

CLASSES PRÉPARATOIRES

Informatique MPI

Cours, exercices, travaux pratiques, problèmes et devoirs

Marc de Falco

Un cours figé, remis en ligne pour consultation

Ce livre reproduit le cours que j'ai écrit lorsque j'étais professeur d'informatique en MP2I/MPI. Il n'est plus tenu à jour au fil des programmes.

Il a été **complété et corrigé grâce à Claude** (Anthropic) dans le seul but de le remettre à disposition dans un état propre et consultable. Les passages rédigés ou modifiés par Claude sont signalés par un trait vertical et un symbole dans la marge gauche :

 en orange () : contenu rédigé par Claude (sections, preuves, corrigés, code) ;

 en violet () : texte de l'auteur corrigé, reformulé ou complété par Claude.

Les corrections ponctuelles (coquilles, mise en forme) ne sont pas signalées.

Ce livre n'a **aucune valeur prescriptive** : ce n'est ni une référence officielle du programme, ni une recommandation pédagogique.

Table des matières

I	Algorithmique	5
1	Union-Find	7
2	Algorithmique avancée des graphes	13
3	TP Algorithme de Kruskal, union-find	31
4	TP Algorithme de Kosaraju et 2-SAT	39
5	TP Couplage maximal dans un graphe biparti	47
6	TP Algorithme de Ford-Fulkerson en C	55
7	Algorithmes d'approximation, algorithmes probabilistes	67
8	TP Algorithmes d'approximation (Vertex Cover et TSP)	81
9	TP Problème du sac à dos	87
10	TP Sac à dos et algorithmes probabilistes	93
II	Langages formels	101
11	Langages réguliers et automates finis	103
12	Fiche Langages réguliers et automates finis	137
13	Exercices Langages réguliers et automates finis	145
14	TP Berry-Sethi en C	153
15	Grammaires non contextuelles	159
16	TP Parser des expressions	173
III	Logique	179
17	Déduction naturelle	181
18	Introduction au lambda-calcul	197
IV	Calculabilité	205
19	Calculabilité	207
20	Exercices Calculabilité	213
V	Système	217
21	Concurrence et parallélisme	219
22	TP Threads, Peterson, mutex et sémaphores	245
VI	Intelligence artificielle	259
23	Intelligence artificielle	261
24	TP KNN - Identifier des chiffres	275
25	TP Classification en basse dimension	279
26	TP Fonctions booléennes et ID3	287
27	TP Morpion	291
28	TP Othorro	299
29	TP Résolution du 15-puzzle	309
VII	Problèmes	317
30	Problème Devoir calculabilité	319
31	Problème Devoir - Logique et Grammaire	327
32	Problème Exercice - Tableaux autoréférents	337
VIII	Corrigés de concours	341
33	Problème DS06 - Sujet CCINP	343
34	Problème Mines - Sujet 0 - MPI	347
35	Problème Mines 2023 - Épreuve 2	357

36	Problème	Centrale 2022 - Option informatique	367
37	Problème	Agrégation 2023 - Épreuve 3b	379
IX	Devoirs MPI 2022-2023		385
38	Devoir	Interro 1 - Graphes	387
39	Devoir	DS 4 - Algorithmes d'approximation, algorithmes probabilistes	389

PARTIE I

Algorithmique

1	Union-Find	7
2	Algorithmique avancée des graphes	13
3	TP Algorithme de Kruskal, union-find	31
4	TP Algorithme de Kosaraju et 2-SAT	39
5	TP Couplage maximal dans un graphe biparti	47
6	TP Algorithme de Ford-Fulkerson en C	55
7	Algorithmes d'approximation, algorithmes probabilistes	67
8	TP Algorithmes d'approximation (Vertex Cover et TSP)	81
9	TP Problème du sac à dos	87
10	TP Sac à dos et algorithmes probabilistes	93





CHAPITRE 1

Union-Find

DANS CE CHAPITRE

- 1.1 Structure de données abstraite d'ensembles disjoints
- 1.2 Implémentation naïve : listes chaînées
- 1.3 Implémentation efficace
- 1.4 Implémentation en c avec un tableau `parent`

1.1 Structure de données abstraite d'ensembles disjoints

1.1.1 Objectif et définition

On cherche à réaliser une structure permettant de manipuler des ensembles disjoints de manière dynamique, c'est-à-dire en fusionnant des ensembles à la volée.

La structure abstraite union-find est ainsi définie par :

- un type `'a uf` pour manipuler la collection d'ensembles dont les éléments sont de type `'a`.
- une opération `makeset` qui étant donné un élément x va rajouter $\{x\}$ à la collection.
- une opération `find` qui associe à chaque élément x , l'ensemble qui le contient dans la collection.
- une opération `union` qui fusionne deux ensembles disjoints, le plus souvent donnés par des éléments qu'ils contiennent, donc implicitement en utilisant `find`.

1.1.2 Exemple emblématique d'utilisation

On considère le problème déjà vu du calcul des composantes connexes d'un graphe, mais avec le graphe donné à la volée, arête après arête, par exemple lors d'un téléchargement.

La solution qu'on connaît consiste à itérer des parcours quelconques d'un sommet non découvert, à chaque parcours on calcule une nouvelle composante connexe. Cet algorithme est ainsi en $O(|S| + |A|)$, donc linéaire, ce qui est a priori très satisfaisant.

Cependant, si on rajoute une arête, on change la nature des composantes et il est nécessaire de refaire un parcours.

Avec une structure union-find, on peut procéder ainsi :

- on appelle `makeset` pour chaque sommet
- pour chaque arête (x, y) où $find(x) \neq find(y)$, on appelle `union(x, y)`.

Si jamais une nouvelle arête est à considérer, il suffit d'appeler `union`.

Pour n éléments dans tous les ensembles, si on suppose que la complexité est $m(n)$ pour `makeset`, $u(n)$ pour `union` et $f(n)$ pour `find`, on a alors une complexité du calcul en $O(m(|S|) |S| + (f(|S|) + u(|S|)) |A|)$.

Pour que cet algorithme soit intéressant, il faut que cette complexité soit proche de la précédente. Cela signifie que les trois fonctions devraient être en $O(1)$ ou presque.

1.1.3 Classe d'équivalence et représentant

On peut définir une relation d'équivalence sur les éléments en posant :

$$x \sim y \Leftrightarrow \text{find}(x) = \text{find}(y)$$

Les ensembles sont alors des classes d'équivalence dont il suffit de donner un représentant pour les retrouver. Ainsi, `find` pourra se contenter de renvoyer un représentant **sous réserve** que `find(x)` et `find(y)` renvoient le même quand $x \sim y$.

Remarque 1.1 Comme on l'a vu en mathématiques, on peut toujours représenter une classe de congruence modulo p avec un représentant dans $\llbracket 0, p-1 \rrbracket$.

1.2 Implémentation naïve : listes chaînées

1.2.1 Première implémentation

On peut représenter chaque élément par un maillon et chaque ensemble par une liste chaînée. Le dernier maillon de chaque liste correspondra alors à l'élément distingué qui représente l'ensemble.

Plus précisément :

- `makeset` crée un maillon $O(1)$
 - pour pouvoir manipuler directement des maillons depuis leurs valeurs, il peut être nécessaire de maintenir un dictionnaire permettant d'associer à une valeur son maillon, on peut le faire en $O(1)$ avec des tables de hachage (en moyenne).
- `find` va descendre le long de la liste correspondant à l'ensemble pour déterminer le maillon final. En pire cas, il faut parcourir toute la liste, c'est donc en $O(n)$ où n est le nombre total d'éléments.
- `union` doit réaliser une concaténation de listes, ce qui est en $O(n)$ en pire cas.

1.2.2 Quick-find

On peut améliorer les performances de `find` en rajoutant un pointeur direct depuis chaque maillon vers le maillon final de la liste qui les contient. L'opération `find` est alors en $O(1)$. Cependant `union` sera en $O(n)$, même en faisant en sorte d'avoir une concaténation rapide comme expliqué dans le paragraphe suivant, car il faut maintenir les pointeurs vers le maillon final.

Cette implémentation est en général dénommée `Quick-find` car `find` est rapide.

1.2.3 Quick-union

On peut faire en sorte que l'union soit rapide en faisant en sorte que le maillon distingué pointe sur le premier et dernier maillon de chaque liste. Ainsi, pour l'union, il suffit de faire un nombre constant d'opérations.

Attention, cela suppose ici que `union` ne va pas réaliser des `find`, qui eux, resteront coûteux.

1.3 Implémentation efficace

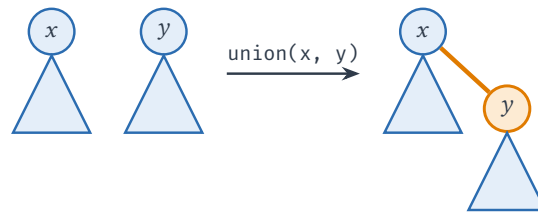
1.3.1 Forêt disjointe

On va représenter une structure union-find par une forêt d'arbres dont les étiquettes sont toutes distinctes. Chaque arbre représentera alors l'ensemble des valeurs de ses étiquettes et la forêt sera naturellement une représentation d'une collection d'ensembles disjoints.

On peut alors identifier une valeur au nœud de l'arbre dont c'est l'étiquette et naïvement effectuer les opérations :

- `makeset` crée un arbre avec une feuille et le rajoute à la forêt, ce qui se fait en temps constant en supposant que la forêt est une liste d'arbres

- `find` va remonter du nœud vers sa racine et renvoyer sa racine, ce qui est en $O(h)$ où h est la hauteur de la forêt, c'est-à-dire la plus grande hauteur des arbres qu'elle contient.
- `union` va supprimer deux arbres et les remplacer par un unique arbre ; la racine de l'un devient un enfant de la racine de l'autre. Ci-dessous, `union(x, y)` où x et y sont des racines :



Ainsi, `union` est naturellement en $O(1)$ sous réserve de partir des racines. On retrouve donc la même complexité que Quick-union avec les listes chaînées pour `union`, et `find` est en $O(h)$.

Cependant, il est possible d'avoir des arbres très profonds et en pire cas, `find` est en $O(n)$ où n est le nombre d'éléments.

Remarque 1.2 Si on considère les entiers de 0 à n et que `union(x, y)` place la racine de l'arbre de y comme enfant de la racine de l'arbre de x (comme sur la figure), la suite d'unions $(1, 0), (2, 0), \dots, (n, 0)$ va créer un peigne avec n à la racine et une unique branche $0 \rightarrow 1 \rightarrow \dots \rightarrow (n-1) \rightarrow n$ (où $x \rightarrow y$ signifie que y est le père de x) donc 0 est à profondeur n et ainsi `find(0)` devra remonter les n arêtes. Comme chaque `union(i, 0)` commence par un `find(0)` qui remonte $i-1$ arêtes, cette suite d'opérations coûte au total $\Theta(n^2)$.

1.3.2 Union pondérée

Quand on réalise une union, on a deux choix sur qui sera la racine de l'arbre fusionné. Si on s'assure de toujours faire le choix qui minimise la hauteur, on peut garantir que la forêt soit de hauteur en $O(\log n)$.

Calculer les tailles ou les hauteurs de chaque arbre n'est pas utile, on peut se contenter de définir le rang d'un arbre noté $rg(a)$:

- au départ, chaque feuille est de rang 0
- quand on réalise une union entre deux arbres a et a'
 - si $rg(a) > rg(a')$ on n'a pas intérêt à placer a plus profondément donc on rajoute a' comme enfant de la racine de a et on préserve les rangs
 - sinon, on fait la fusion dans l'autre sens et dans le cas où les deux rangs étaient égaux, on augmente le rang de a' de 1.

Ainsi, les rangs indiquent le nombre de fusions successives qui ont augmenté la hauteur : en l'absence d'autre modification des arbres, le rang d'un arbre est exactement sa hauteur (on le vérifie par récurrence sur le nombre d'unions : si $rg(a) > rg(a')$, la hauteur de l'arbre fusionné est $\max(h(a), h(a') + 1) = h(a)$ car $h(a') + 1 \leq h(a)$; si les rangs sont égaux à r , elle vaut $r + 1$).

Théorème 1.3

Avec l'union par rang, un arbre de rang r possède au moins 2^r nœuds. En conséquence, pour n éléments, tout arbre est de rang, donc de hauteur, au plus $\log_2 n$, et `find` et `union` sont en $O(\log n)$ dans le pire cas.

Preuve Par récurrence sur le nombre d'opérations. Au départ, chaque arbre est une feuille de rang 0 ayant $1 = 2^0$ nœud. Lors d'une union de a et a' :

- si $rg(a) \neq rg(a')$, le rang de l'arbre obtenu est $\max(rg(a), rg(a'))$ et il contient plus de nœuds que l'arbre de plus grand rang, donc au moins $2^{\max(rg(a), rg(a'))}$ nœuds par hypothèse de récurrence ;
- si $rg(a) = rg(a') = r$, l'arbre obtenu est de rang $r + 1$ et contient au moins $2^r + 2^r = 2^{r+1}$ nœuds.

Les autres arbres ne sont pas modifiés. Comme un arbre a au plus n nœuds, on en déduit $2^r \leq n$, soit $r \leq \log_2 n$. Le rang étant égal à la hauteur, `find` est en $O(\log n)$, et `union`, qui fait deux `find` puis un nombre constant d'opérations, aussi.

□

1.3.3 Compression de chemins

La structure de l'arbre est souvent inintéressante, on peut donc se permettre de la supprimer et d'aplatir les arbres. Pour cela, si la parenté est déterminée par un tableau `parent` avec `parent[x] = x` à la racine, on peut procéder récursivement pour `find` :

Calcul de `find(x)`

- Si $x \neq \text{parent}[x]$
 - `parent[x] = find(parent[x])`
- Renvoyer `parent[x]`

Ainsi, après un appel à `find(x)`, chaque nœud du chemin de x à la racine pointe directement vers la racine. Cela revient à aplatir le chemin parcouru : tous ses nœuds deviennent des enfants de la racine. On parle de *compression de chemins* puisque ces chemins, qui seront de nouveau parcourus lors des appels suivants, deviennent de longueur 1.

La compression de chemins ne modifie pas les rangs : le rang d'un arbre n'est alors plus égal à sa hauteur, mais en reste un majorant, et le théorème précédent reste valable (les nombres de nœuds des arbres ne changent pas).

Théorème 1.4 — (Tarjan, admis)

Avec l'union par rang et la compression de chemins, une suite de m opérations `makeset`, `find` et `union`, dont n opérations `makeset`, s'exécute dans le pire cas en $O(m\alpha(n))$, où α est la réciproque de la fonction d'Ackermann. Autrement dit, chaque opération est en $O(\alpha(n))$ en complexité **amortie**.

La fonction $\alpha(n)$ croît tellement lentement que $\alpha(n) \leq 4$ pour $n \leq 10^{80}$: en pratique, les opérations sont en temps constant amorti. Il s'agit bien d'une complexité amortie (sur une suite d'opérations dans le pire cas) et non d'une complexité en moyenne : un `find` isolé peut coûter $\Theta(\log n)$.

1.4 Implémentation en c avec un tableau `parent`

Remarque 1.5 Cette implémentation peut également se faire facilement avec `OCaml`.

On va considérer que les éléments sont les entiers de 0 à $n - 1$ et on va représenter la forêt par un tableau `parent` où `parent[x]` indique le père de x si x n'est pas une racine, et `parent[x] = x` pour les racines.

Remarque 1.6 On peut donc tester si un élément est la racine d'un arbre en $O(1)$.

On va représenter la forêt par le type suivant :

```
#include <stdlib.h>
#include <assert.h>

struct union_find {
    int *parent;
    int *rang;
    int nelements;
};
typedef struct union_find union_find;
```

Pour commencer, on crée une forêt de feuilles qui sont toutes de rang 0.

```

union_find uf_makesets(int n)
{
    union_find uf;
    uf.parent = malloc(sizeof(int) * n);
    uf.rang = malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++)
    {
        uf.parent[i] = i;
        uf.rang[i] = 0;
    }
    uf.nelements = n;
    return uf;
}

```

On implémente alors directement find et union, en prenant garde au fait que union est un mot-clé en C :

```

int uf_find(union_find uf, int x)
{
    if (uf.parent[x] != x)
        uf.parent[x] = uf_find(uf, uf.parent[x]);
    return uf.parent[x];
}

```

```

void uf_union(union_find uf, int x, int y)
{
    int rx = uf_find(uf, x);
    int ry = uf_find(uf, y);
    if (rx == ry)
        return; // x et y sont déjà dans le même ensemble

    if (uf.rang[rx] > uf.rang[ry])
        uf.parent[ry] = rx;
    else {
        uf.parent[rx] = ry;
        if (uf.rang[rx] == uf.rang[ry])
            uf.rang[ry] = uf.rang[ry] + 1;
    }
}

```

Remarque 1.7 Le test `rx == ry` est indispensable. Si `x` et `y` sont déjà dans le même ensemble, il n'y a rien à fusionner : sans ce test, on exécuterait `uf.parent[rx] = ry`, c'est-à-dire `uf.parent[rx] = rx`, ce qui est sans effet, mais aussi `uf.rang[ry] = uf.rang[ry] + 1` puisque les deux rangs sont trivialement égaux. Le rang augmenterait alors sans que l'arbre change, et on perdrait la propriété « un arbre de rang r possède au moins 2^r nœuds » sur laquelle repose la complexité en $O(\log n)$.

Plutôt que d'interdire cet appel, on le rend inoffensif : l'appelant peut ainsi appeler `uf_union(uf, x, y)` pour chaque arête (x, y) sans tester au préalable si `uf_find(uf, x) != uf_find(uf, y)`.

Comme pour toute structure allouée sur le tas, on fournit une fonction de libération de la mémoire, à appeler une fois la structure devenue inutile.

```

void uf_detruire(union_find uf)
{
    free(uf.parent);
    free(uf.rang);
}

```


Algorithmique avancée des graphes

DANS CE CHAPITRE

- 2.1 Arbre couvrant minimal
- 2.2 Kosaraju et composantes fortement connexes
- 2.3 Couplage maximal dans un graphe biparti
- 2.4 Exercices
- 2.5 Travaux pratiques

On présente ici des algorithmes qui complètent les notions vues en première année dans le chapitre [Graphes](#) :

- le calcul d'un arbre couvrant de poids minimal dans un graphe pondéré ;
- le calcul des composantes fortement connexes dans un graphe orienté ;
- le calcul d'un couplage maximal dans un graphe biparti, qui est un cas particulier d'un problème plus général de flot maximal.

2.1 Arbre couvrant minimal

2.1.1 Présentation du problème

Description informelle : on a un ensemble de maisons reliées par des routes, on cherche à poser des câbles le long des routes pour que toute paire de maisons soit connectée. Quelle est la longueur minimale de câble à utiliser ?

On comprend assez vite qu'il faut que les câbles forment un graphe connexe et que, pour des questions de réduction de coût, on peut supposer que ce graphe est acyclique. On cherche donc un arbre qui couvre chaque maison et dont la somme des poids des arêtes, les longueurs des câbles, est minimale.

Définition 2.1

Soit $G = (S, A)$ un graphe non orienté connexe. On dit que $T \subset A$ est un **arbre couvrant** de G lorsque :

- $\forall x \in S, \exists a \in T, x \in a$: chaque sommet appartient à au moins une arête de T ;
- (S, T) est un arbre, c'est-à-dire un graphe acyclique et connexe.

On notera ici $\mathcal{T}(G)$ l'ensemble des arbres couvrants de G .

Remarque 2.2 Comme (S, T) a pour sommets **tous** les sommets de G , la seconde condition entraîne la première dès que $|S| \geq 2$: un sommet d'un graphe connexe ayant au moins deux sommets appartient à une arête. Pour $|S| = 1$, l'unique arbre couvrant est $T = \emptyset$ et on ignore la première condition. Un arbre couvrant possède toujours exactement $|S| - 1$ arêtes (voir les [exercices sur les graphes](#)).



Exemple 2.3

Dans le graphe pondéré suivant, les arêtes en trait épais forment un arbre couvrant, de poids $1 + 2 + 5 + 2 + 3 = 13$. On verra qu'il est de poids minimal.

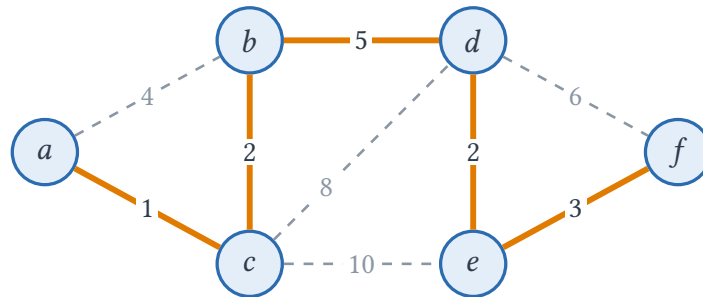


Figure 2.1 — Un arbre couvrant de poids minimal (13), en orange

En remplaçant l'arête $b \smile d$ par l'arête $c \smile d$, on obtient un autre arbre couvrant, de poids 16. En revanche, les arêtes $a \smile b$, $a \smile c$ et $b \smile c$ ne peuvent pas être toutes les trois dans un arbre couvrant car elles forment un cycle.

Une manière naturelle d'obtenir un arbre couvrant est de faire un parcours quelconque ou de calculer les composantes connexes avec une structure [union-find](#). Dans ce dernier cas, comme le graphe est connexe, on obtiendra directement un unique arbre dans la forêt, qui est un arbre couvrant. C'est l'ordre de traitement des arêtes qui va aiguiller vers un arbre de $\mathcal{T}(G)$ ou un autre.

Définition 2.4

Soit $G = (S, A, w)$ un graphe non orienté connexe et pondéré par $w : A \rightarrow \mathbb{R}$. On note $w(T) = \sum_{a \in T} w(a)$ le poids d'un arbre couvrant T de G .

Comme $\mathcal{T}(G)$ est fini et non vide (voir ci-dessus), il existe, au moins, un arbre T_0 tel que

$$w(T_0) = \min_{T \in \mathcal{T}(G)} w(T)$$

On dit que T_0 est un **arbre couvrant de poids minimal**.

Remarque 2.5 En anglais, on parle de *minimum spanning tree*. Les poids peuvent être négatifs : comme tous les arbres couvrants ont le même nombre d'arêtes, ajouter une même constante à tous les poids ne change pas les arbres couvrants de poids minimal.

2.1.2 Algorithme de Kruskal

Pour calculer un arbre couvrant de poids minimal, on va considérer le calcul des composantes connexes avec une structure [union-find](#) mais en traitant les arêtes dans l'ordre croissant de leurs poids : c'est l'algorithme de Kruskal.

Algorithme 2.6 — : Kruskal

Entrée : un graphe non orienté pondéré connexe $G = (S, A, w)$.

Déroulement :

- Pour chaque $x \in S$
 - `makeset(x)`
- On trie A par ordre croissant de poids.
- $T \leftarrow \emptyset$
- Pour chaque $\{x, y\} \in A$ dans cet ordre
 - Si `find(x) ≠ find(y)`

- $\text{union}(x, y)$
- $T \leftarrow T \cup \{\{x, y\}\}$
- On renvoie T , l'unique arbre de la forêt.

Exemple 2.7

Sur le graphe de l'exemple précédent, les arêtes triées par poids croissant sont $a \smile c$ (1), $b \smile c$ (2), $d \smile e$ (2), $e \smile f$ (3), $a \smile b$ (4), $b \smile d$ (5), $d \smile f$ (6), $c \smile d$ (8) et $c \smile e$ (10).

- On ajoute $a \smile c$, $b \smile c$, $d \smile e$ et $e \smile f$: la forêt a alors deux arbres, $\{a, b, c\}$ et $\{d, e, f\}$.
- $a \smile b$ est rejetée car a et b sont déjà dans le même arbre.
- On ajoute $b \smile d$ qui relie les deux arbres.
- Les arêtes suivantes sont toutes rejetées.

On obtient l'arbre en trait épais de la figure, de poids 13.

2.1.3 Correction de l'algorithme de Kruskal

On va montrer la correction d'une famille d'algorithmes à laquelle Kruskal appartient.

Définition 2.8

On dit que $T \subset A$ est une **forêt minimale** de G s'il existe T' arbre couvrant de poids minimal de G tel que $T \subset T'$.

Remarque 2.9 $\emptyset$ est ainsi une forêt minimale. Une forêt minimale est acyclique puisqu'elle est incluse dans un arbre.

Définition 2.10

Soit T une forêt minimale et $a \in A \setminus T$. On dit que a est une arête **sûre** pour T si $T \cup \{a\}$ est encore une forêt minimale.

Lemme 2.11

Soit T une forêt minimale qui n'est pas un arbre couvrant. Il existe une arête sûre pour T .

Preuve Comme T est une forêt minimale, il existe T' arbre couvrant de poids minimal tel que $T \subset T'$. Comme T lui-même n'est pas un arbre couvrant, $T \neq T'$ et il existe $a \in T' \setminus T$. On a alors $T \cup \{a\} \subset T'$ donc a est une arête sûre.

□

On en déduit un proto-algorithme de calcul d'un arbre couvrant de poids minimal :

Algorithme 2.12 — : ArbreMinQuelconque

Entrée : un graphe non orienté pondéré connexe $G = (S, A, w)$.

Déroulement :

- On pose $T = \emptyset$
- Tant que T n'est pas un arbre couvrant
 - Déterminer une arête sûre $a \in A$ pour T
 - $T \leftarrow T \cup \{a\}$
- Renvoyer T

Théorème 2.13

Cet algorithme renvoie un arbre couvrant de poids minimal.

Preuve Cet algorithme vérifie directement l'invariant suivant : T est une forêt minimale. En effet, c'est vrai au départ et le choix d'une arête sûre, qui existe d'après le lemme, permet de prolonger l'invariant.

Cet algorithme termine car T gagne une arête à chaque itération et reste inclus dans un arbre couvrant, qui possède $|S| - 1$ arêtes : il y a donc au plus $|S| - 1$ itérations.

En sortie de boucle, T est un arbre couvrant et une forêt minimale : il est inclus dans un arbre couvrant de poids minimal T' ayant le même nombre d'arêtes, donc $T = T'$. □

Remarque 2.14 Il s'agit d'un proto-algorithme car la partie critique est de déterminer une arête sûre et c'est la partie qui n'est pas explicitée pour le moment.

L'outil principal pour exhiber des arêtes sûres est la notion de coupure.

Définition 2.15

Une **coupure** de $G = (S, A)$ est une partition $S = X \cup (S \setminus X)$ avec $X \neq \emptyset$ et $X \neq S$. On dit qu'une arête $\{x, y\}$ **traverse** la coupure si elle a une extrémité dans X et l'autre dans $S \setminus X$, et qu'une partie $T \subset A$ **respecte** la coupure si aucune arête de T ne la traverse.

Théorème 2.16 — (propriété de coupure)

Soit F une forêt minimale et $(X, S \setminus X)$ une coupure que F respecte. Si e est une arête de poids minimal parmi les arêtes qui traversent la coupure, alors e est sûre pour F .

Preuve Comme F est une forêt minimale, il existe T arbre couvrant de poids minimal tel que $F \subset T$.

Si $e \in T$, alors $F \cup \{e\} \subset T$ est encore une forêt minimale.

Sinon, notons $e = \{x, y\}$ avec $x \in X$ et $y \notin X$. Comme T est un arbre, il existe un unique chemin élémentaire φ de x à y dans T et $e \notin \varphi$. Ce chemin part de X et arrive dans $S \setminus X$: il possède donc une arête e' qui traverse la coupure. Comme F respecte la coupure, $e' \in T \setminus F$, et par minimalité de e , $w(e) \leq w(e')$.

On pose $T' = (T \setminus \{e'\}) \cup \{e\}$. Retirer e' de T déconnecte l'arbre en exactement deux composantes connexes, l'une contenant x et l'autre y puisque e' est sur l'unique chemin de x à y . L'arête e reconnecte ces deux composantes : (S, T') est connexe et possède $|S| - 1$ arêtes, c'est donc un arbre couvrant (voir les exercices). Autrement dit, on a cassé l'unique cycle de $T \cup \{e\}$, formé de φ et de e , en enlevant e' .

Reste que $w(T') = w(T) - w(e') + w(e) \leq w(T)$, donc T' est un arbre couvrant de poids minimal.

Ainsi $F \cup \{e\} \subset T'$ car $e' \notin F$: c'est une forêt minimale et e est sûre. □

On en déduit le critère utilisé par Kruskal.

Théorème 2.17

Si F est une forêt minimale non couvrante et e une arête de plus petit poids reliant deux arbres distincts de F , alors e est sûre pour F .

Preuve Notons C l'ensemble des sommets de l'arbre de F contenant l'une des extrémités de e . Toute arête traversant la coupure $(C, S \setminus C)$ relie deux arbres distincts de F et F respecte cette coupure. L'arête e est donc de poids minimal parmi les arêtes traversant la coupure et la propriété de coupure permet de conclure. □

Corollaire 2.18

Kruskal renvoie un arbre couvrant de poids minimal.

Preuve On montre que l'ensemble T des arêtes ajoutées est toujours une forêt minimale, les arbres de la forêt union-find ayant pour sommets ceux des arbres de T . Au départ, $T = \emptyset$ est une forêt minimale.

Quand l'algorithme ajoute l'arête $e = \{x, y\}$, x et y sont dans deux arbres distincts de T . Toute arête e' de poids strictement inférieur à celui de e a déjà été examinée : soit elle a été ajoutée, soit ses extrémités étaient déjà dans le même arbre. Dans les deux cas, ses extrémités sont dans le même arbre de T au moment où on examine e , car les arbres ne font que grossir. Ainsi e est une arête de plus petit poids reliant deux arbres distincts de T et elle est sûre d'après le théorème précédent.

À la fin, T est une forêt minimale. Si elle n'était pas couvrante, comme G est connexe, une arête de G relierait deux arbres distincts de T : elle aurait été ajoutée lors de son examen, puisque ses extrémités étaient alors dans des arbres distincts. Donc T est un arbre couvrant de poids minimal. □

2.1.4 Complexité de l'algorithme de Kruskal

On peut décomposer l'algorithme :

- La création avec `makeSet` est en $O(|S|)$.
- Le tri des arêtes est en $O(|A| \log |A|)$.
- La boucle est en $O(|A| \alpha(|S|))$ avec $\alpha(|S|) = o(\log |A|)$, en utilisant une structure union-find avec union par rang et compression de chemins.

Comme G est connexe, on a $|A| \geq |S| - 1$ et ainsi $|S| = O(|A|)$. La complexité globale est donc en $O(|A| \log |A|)$, c'est-à-dire en $O(|A| \log |S|)$ car $|A| \leq |S|^2$.

Remarque 2.19 Kruskal fait partie de ces algorithmes qui sont linéaires, ou presque, après avoir fait un tri. C'est un cas que l'on a déjà vu avec les algorithmes gloutons. D'ailleurs, on peut dire que Kruskal est un choix glouton d'arête sûre.

2.1.5 Implémentation de Kruskal

On reprend la structure union-find du chapitre [Union-Find](#). En OCaml, le graphe est donné par la liste de ses arêtes sous la forme de triplets (x, y, poids) ; en C, par un tableau de structures.

OCaml

```
let rec find parent x =
  if parent.(x) = x
  then x
  else begin
    let r = find parent parent.(x) in
    parent.(x) <- r; (* compression de chemins *)
    r
  end

(* aretes : liste de triplets (x, y, poids) d'un graphe
   non orienté connexe dont les sommets sont 0, ..., n-1 *)
let kruskal n aretes =
  let parent = Array.init n (fun x -> x) in
  let rang = Array.make n 0 in
  let triees =
    List.sort (fun (_, _, p1) (_, _, p2) -> compare p1 p2) aretes in
  let arbre = ref [] in
  List.iter (fun (x, y, p) ->
    let rx = find parent x in
    let ry = find parent y in
    if rx <> ry
    then begin
      arbre := (x, y, p) :: !arbre;
      (* union par rang *)
```

```

        if rang.(rx) < rang.(ry) then parent.(rx) <- ry
        else if rang.(rx) > rang.(ry) then parent.(ry) <- rx
        else begin
            parent.(ry) <- rx;
            rang.(rx) <- rang.(rx) + 1
        end
    end
end
) trieess;
!arbre

```

C

```

#include <stdlib.h>
#include <assert.h>

// union_find, uf_makesets, uf_find, uf_union et uf_detruire
// sont ceux du chapitre Union-Find

struct arete_ponderee {
    int x;
    int y;
    int poids;
};
typedef struct arete_ponderee arete_ponderee;

int compare_aretes(const void *a1, const void *a2) {
    int p1 = ((const arete_ponderee *)a1)->poids;
    int p2 = ((const arete_ponderee *)a2)->poids;
    return (p1 > p2) - (p1 < p2);
}

// Trie le tableau aretes (de taille m) d'un graphe connexe à n sommets
// et renvoie un tableau des n-1 arêtes d'un arbre couvrant de poids minimal
arete_ponderee *kruskal(int n, arete_ponderee *aretes, int m) {
    qsort(aretes, m, sizeof(arete_ponderee), compare_aretes);
    arete_ponderee *arbre = malloc((n > 1 ? n - 1 : 1) * sizeof(arete_ponderee));
    int taille = 0;
    union_find uf = uf_makesets(n);
    for (int i = 0; i < m && taille < n - 1; i++) {
        if (uf_find(uf, aretes[i].x) != uf_find(uf, aretes[i].y)) {
            uf_union(uf, aretes[i].x, aretes[i].y);
            arbre[taille++] = aretes[i];
        }
    }
    assert(taille == n - 1); // sinon le graphe n'est pas connexe
    uf_detruire(uf);
    return arbre;
}

```

En C, on peut s'arrêter dès que $|S| - 1$ arêtes ont été ajoutées. Le comparateur renvoie le signe de $p_1 - p_2$ sans calculer cette différence, qui pourrait dépasser la capacité d'un int.

2.1.6 Algorithme de Prim

La propriété de coupure permet de justifier un autre algorithme : au lieu de faire grossir une forêt, on fait grossir **un seul arbre** à partir d'un sommet de départ, en lui ajoutant à chaque étape l'arête la plus légère qui le relie à un sommet extérieur.

Algorithme 2.20 — : Prim

Entrée : un graphe non orienté pondéré connexe $G = (S, A, w)$ et un sommet $r \in S$.

Déroulement :

- $X \leftarrow \{r\}$ et $T \leftarrow \emptyset$
- Tant que $X \neq S$
 - Choisir une arête $\{x, y\}$ de poids minimal parmi celles qui vérifient $x \in X$ et $y \notin X$

- $X \leftarrow X \cup \{y\}$ et $T \leftarrow T \cup \{(x, y)\}$
- Renvoyer T

Théorème 2.21

L'algorithme de Prim renvoie un arbre couvrant de poids minimal.

Preuve On montre l'invariant : T est une forêt minimale dont les arêtes ont leurs deux extrémités dans X , et (X, T) est un arbre.

C'est vrai au départ. Si $X \neq S$, $(X, S \setminus X)$ est une coupure, traversée par au moins une arête car G est connexe, et T la respecte puisque ses arêtes ont leurs deux extrémités dans X . L'arête choisie est de poids minimal parmi celles qui traversent la coupure : elle est sûre d'après la propriété de coupure. De plus, ajouter à l'arbre (X, T) un nouveau sommet y et une arête le reliant à X donne encore un arbre.

Chaque itération ajoute un sommet à X : l'algorithme termine après $|S| - 1$ itérations avec $X = S$. Alors T est un arbre couvrant et une forêt minimale, donc un arbre couvrant de poids minimal comme dans la preuve de ArbreMinQuelconque.

□

Exemple 2.22

Sur le graphe de l'exemple, en partant de a , on ajoute successivement $a \smile c$ (1), $c \smile b$ (2), $b \smile d$ (5), $d \smile e$ (2) et $e \smile f$ (3). On retrouve le même arbre qu'avec Kruskal. On remarque que les poids des arêtes ne sont pas ajoutés dans l'ordre croissant.

Pour réaliser efficacement le choix de l'arête, on procède comme pour l'algorithme de Dijkstra du chapitre [Graphes](#) : on maintient pour chaque sommet $y \notin X$ la valeur $d(y)$, poids de l'arête la plus légère connue reliant y à X , et le sommet $parent(y) \in X$ correspondant. Quand on ajoute un sommet x à X , on met à jour $d(y)$ pour ses voisins y . La seule différence avec Dijkstra est la mise à jour : $d(y) \leftarrow \min(d(y), w(x, y))$ au lieu de $d(y) \leftarrow \min(d(y), d(x) + w(x, y))$.

OCaml

```
(* g : listes d'adjacence (voisin, poids) d'un graphe non orienté connexe *)
let prim g =
  let n = Array.length g in
  (* d.(y) : poids de la plus légère arête connue reliant y à l'arbre *)
  let d = Array.make n max_int in
  let parent = Array.make n (-1) in
  let dans_arbre = Array.make n false in
  let arbre = ref [] in
  if n > 0 then d.(0) <- 0;
  for _ = 1 to n do
    (* sommet x hors de l'arbre avec d.(x) minimal *)
    let x = ref (-1) in
    for y = 0 to n - 1 do
      if not dans_arbre.(y) && (!x < 0 || d.(y) < d.(!x))
      then x := y
    done;
    let x = !x in
    dans_arbre.(x) <- true;
    if parent.(x) >= 0
    then arbre := (parent.(x), x, d.(x)) :: !arbre;
    List.iter (fun (y, p) ->
      if not dans_arbre.(y) && p < d.(y)
      then begin
        d.(y) <- p;
        parent.(y) <- x
      end
    ) g.(x)
  ) g.(x)
```

```
done;
!arbre
```

C

```
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

// arete et graphe_pondere sont ceux de l'algorithme de Dijkstra
// (chapitre Graphes)

// g est non orienté (chaque arête apparaît dans les deux sens) et connexe.
// Renvoie le tableau parent de l'arbre obtenu, enraciné en 0 :
// les arêtes de l'arbre sont les parent[y] -- y pour y != 0.
int *prim(graphe_pondere g) {
    int *d = malloc(g.n * sizeof(int));
    int *parent = malloc(g.n * sizeof(int));
    bool *dans_arbre = malloc(g.n * sizeof(bool));
    for (int y = 0; y < g.n; y++) {
        d[y] = INT_MAX;
        parent[y] = -1;
        dans_arbre[y] = false;
    }
    if (g.n > 0) d[0] = 0;
    for (int k = 0; k < g.n; k++) {
        // sommet x hors de l'arbre avec d[x] minimal
        int x = -1;
        for (int y = 0; y < g.n; y++) {
            if (!dans_arbre[y] && (x < 0 || d[y] < d[x])) {
                x = y;
            }
        }
        dans_arbre[x] = true;
        for (int i = 0; i < g.degree[x]; i++) {
            int y = g.voisins[x][i].but;
            int p = g.voisins[x][i].poids;
            if (!dans_arbre[y] && p < d[y]) {
                d[y] = p;
                parent[y] = x;
            }
        }
    }
    free(d);
    free(dans_arbre);
    return parent;
}
```

Ici, on part du sommet 0 et la recherche du minimum est linéaire : chaque itération coûte $O(|S|)$ plus le parcours des voisins de x , d'où une complexité en $O(|S|^2 + |A|) = O(|S|^2)$, intéressante pour un graphe dense. Avec une file de priorité, comme pour Dijkstra, on obtient $O((|S| + |A|) \log |S|)$, soit la même complexité que Kruskal.

Remarque 2.23 Ces programmes ont été testés en comparant le poids de l'arbre obtenu avec celui d'un arbre de poids minimal obtenu par énumération de toutes les parties à $|S| - 1$ arêtes, sur des milliers de petits graphes aléatoires (avec des poids égaux ou négatifs).

2.2 Kosaraju et composantes fortement connexes

2.2.1 Rappel des définitions

Soit $G = (S, A)$ un graphe **orienté**. On note, pour $x, y \in S$, $x \rightsquigarrow y$ quand il existe un chemin dans G reliant x à y . On dit que y est *accessible* depuis x .

On note $x \leftrightarrow y \Leftrightarrow x \rightsquigarrow y \wedge y \rightsquigarrow x$. C'est la restriction symétrique de $\rightsquigarrow$. Comme $\rightsquigarrow$ est réflexive et transitive, $\leftrightarrow$ est une relation d'équivalence. Les classes d'équivalence pour $\leftrightarrow$ sont appelées les **composantes fortement connexes** de G (voir le chapitre [Graphes](#)). On note leur ensemble $CFC(G) = S / \leftrightarrow$.

Par exemple, sur le graphe :

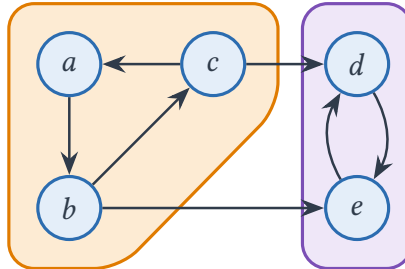


Figure 2.2 — Les deux composantes fortement connexes $\{a, b, c\}$ et $\{d, e\}$

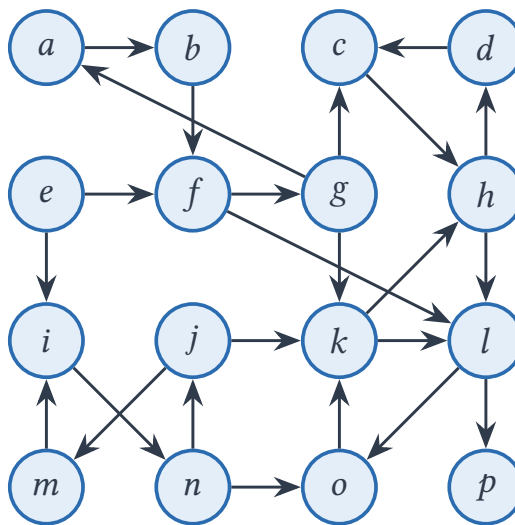
on a les deux composantes fortement connexes $\{a, b, c\}$ et $\{d, e\}$.

On remarque une différence fondamentale avec les composantes connexes : il peut y avoir des arêtes entre deux composantes fortement connexes distinctes, mais seulement dans un sens. Ici, $c \rightarrow d$ et $b \rightarrow e$ vont de la première composante vers la seconde, et il ne peut pas y avoir d'arête de la seconde vers la première sans fusionner les deux composantes.

On cherche ici à déterminer un algorithme pour calculer les composantes fortement connexes d'un graphe.

2.2.2 Exemple

On considère le graphe orienté suivant, qu'on retrouvera dans les exercices et en TP :



Ses composantes fortement connexes sont $\{a, b, f, g\}$, $\{c, d, h, k, l, o\}$, $\{e\}$, $\{i, j, m, n\}$ et $\{p\}$. Par exemple, $f \rightarrow l$ mais on ne peut pas revenir de l vers f : depuis $\{c, d, h, k, l, o\}$, on ne peut atteindre que p .

Si on considère chaque composante comme un seul sommet, on obtient un graphe appelé **graphe des composantes** ; il est acyclique car un cycle entre composantes les fusionnerait. Ici, ses arêtes sont $\{e\} \rightarrow \{a, b, f, g\}$, $\{e\} \rightarrow \{i, j, m, n\}$, $\{a, b, f, g\} \rightarrow \{c, d, h, k, l, o\}$, $\{i, j, m, n\} \rightarrow \{c, d, h, k, l, o\}$ et $\{c, d, h, k, l, o\} \rightarrow \{p\}$.

On va voir que les temps de sortie d'un parcours en profondeur permettent de retrouver un ordre topologique de ce graphe des composantes, et ainsi de calculer les composantes.

2.2.3 Rappels sur le parcours en profondeur

Le parcours en profondeur et ses temps d'entrée et de sortie sont très importants ici. On reprend le parcours complet de la partie [Temps d'entrée et de sortie](#) du chapitre Graphes, avec ses notations : un sommet est

inconnu, découvert ou **traité** ; l'appel `parcours` sur x marque x comme découvert, note son temps d'entrée $t_e(x)$, relance le parcours sur ses voisins inconnus, puis marque x comme traité et note son temps de sortie $t_s(x)$.

Définition 2.24

On considère un parcours complet d'un graphe $G = (S, A)$ et $x \in S$.

Soit $y \in S$. On dit que y est accessible par un **chemin inconnu** depuis x s'il existe un chemin de x à y ne passant que par des sommets inconnus au moment où on lance l'appel sur x (y compris x lui-même, qui n'est pas encore marqué). On note $x \xrightarrow{I} y$.

Théorème 2.25 — (chemin inconnu)

$x \xrightarrow{I} y$ si et seulement si y est découvert pendant l'appel sur x , c'est-à-dire entre le début et la fin de cet appel.

Preuve

- $\Rightarrow$) Soit $x = u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_k = y$ un chemin dont tous les sommets sont inconnus au début de l'appel sur x . Supposons par l'absurde que l'un des u_i ne soit pas découvert pendant cet appel, et prenons i minimal. On a $i \geq 1$ car x est découvert au début de son appel. Le sommet u_{i-1} est découvert pendant l'appel sur x , donc son propre appel se déroule pendant celui de x (voir le lemme du chapitre Graphes sur la pile d'appels). Lors de cet appel, l'arête $u_{i-1} \rightarrow u_i$ est examinée : soit u_i est encore inconnu et il est découvert à ce moment, soit il ne l'est plus et, comme il était inconnu au début de l'appel sur x , il a été découvert entre-temps. Dans les deux cas, u_i est découvert pendant l'appel sur x : c'est absurde.
- $\Leftarrow$) Si y est découvert pendant l'appel sur x , on considère la chaîne des appels en cours au moment de sa découverte, de x à y : chaque appel a été lancé par le précédent sur un voisin, qui était alors inconnu, et qui l'était donc aussi au début de l'appel sur x . Ces sommets forment un chemin de x à y , qui est donc un chemin inconnu.

□

Théorème 2.26

Soient $x, y \in S$ tels que $t_e(x) < t_e(y)$.

- Si $x \xrightarrow{I} y$, alors $t_s(y) < t_s(x)$.
- Sinon, $t_s(x) < t_e(y)$.

Autrement dit, soit $[t_e(y); t_s(y)] \subset [t_e(x); t_s(x)]$, soit $[t_e(y); t_s(y)] \cap [t_e(x); t_s(x)] = \emptyset$.

On dit que les temps sont bien parenthésés. En effet, si on considère le mot sur S avec les lettres $(_x$ et $)_x$ pour chaque sommet x et tel qu'on écrive la lettre $(_x$ quand on note le temps d'entrée et $)_x$ quand on note le temps de sortie, alors ce mot est bien parenthésé. C'est le théorème des parenthèses du chapitre Graphes, vu ici sous l'angle des chemins inconnus.

Exemple 2.27

Sur le graphe $\{a, b, c, d, e\}$ du début de cette partie, en lançant le parcours depuis a et en examinant les voisins dans l'ordre alphabétique, on obtient le mot $(_a(b(c(d(e)_e)_d)_c)_b)_a$.

Preuve Comme $t_e(x) < t_e(y)$, on a commencé l'appel sur x avant de découvrir y . Si $x \xrightarrow{I} y$, alors par le théorème précédent, y est découvert pendant l'appel sur x : l'appel sur y se déroule pendant celui sur x et termine donc avant lui. Ainsi $t_s(y) < t_s(x)$.

Sinon, toujours par le théorème précédent, y n'est pas découvert pendant l'appel sur x , et comme il est découvert après le début de cet appel, il l'est après sa fin. Donc $t_s(x) < t_e(y)$.

□

Définition 2.28

Soit $C \in CFC(G)$. On note

$$t_e(C) = \min\{t_e(x) \mid x \in C\}$$

$$t_s(C) = \max\{t_s(x) \mid x \in C\}$$

On a alors une propriété des temps de sortie sur les composantes fortement connexes elles-mêmes.

Théorème 2.29

Soient $C, C' \in CFC(G)$ **distinctes**.

S'il existe $x \in C, y \in C'$ avec $x \rightarrow y$, alors $t_s(C') < t_s(C)$.

Preuve On suppose qu'il existe $x \in C, y \in C'$ avec $x \rightarrow y$.

Premier cas, $t_e(C) < t_e(C')$. On considère $u \in C$ tel que $t_e(C) = t_e(u)$: c'est le premier sommet de $C \cup C'$ découvert, donc tous les sommets de $C \cup C'$ sont inconnus au début de l'appel sur u . Or tout sommet $v \in C \cup C'$ est accessible depuis u en restant dans $C \cup C'$: directement si $v \in C$, en passant par $x \rightarrow y$ si $v \in C'$. On a donc $u \rightsquigarrow_I v$ et, par le théorème précédent, $t_s(v) < t_s(u)$ pour tout $v \neq u$. Ainsi $t_s(C) = t_s(u) > t_s(C')$.

Second cas, $t_e(C') < t_e(C)$. Soit $u \in C'$ tel que $t_e(C') = t_e(u)$. Comme précédemment, tous les sommets de C' sont découverts pendant l'appel sur u , donc $t_s(u) = t_s(C')$. En revanche, aucun sommet de C n'est accessible depuis u : sinon, avec $x \rightarrow y$, on aurait $u \rightsquigarrow x \rightsquigarrow u$ et $C = C'$. Par le théorème du chemin inconnu, aucun sommet de C n'est découvert pendant l'appel sur u ; comme ils sont tous inconnus au début de cet appel, ils sont découverts après sa fin. On a bien $t_s(C) > t_e(C) > t_s(u) = t_s(C')$.

□

Remarque 2.30 En particulier, en triant les composantes par temps de sortie décroissant, on obtient un tri topologique du graphe des composantes. Dans l'exemple précédent, en lançant les parcours depuis a puis e et en examinant les voisins dans l'ordre alphabétique, on obtient $t_s(\{e\}) = 32$, $t_s(\{i, j, m, n\}) = 31$, $t_s(\{a, b, f, g\}) = 22$, $t_s(\{c, d, h, k, l, o\}) = 18$ et $t_s(\{p\}) = 15$.

2.2.4 Graphe miroir**Définition 2.31**

Soit $G = (S, A)$ un graphe orienté. On appelle **graphe miroir** de G le graphe $G^R = (S, A^R)$ où

$$\forall x, y \in S, (x, y) \in A \Leftrightarrow (y, x) \in A^R$$

Cela revient à renverser toutes les flèches du graphe G .

Théorème 2.32

$$CFC(G) = CFC(G^R)$$

Preuve Un chemin de x à y dans G donne, en le parcourant à l'envers, un chemin de y à x dans G^R , et réciproquement. Ainsi $x \rightsquigarrow_G y \Leftrightarrow y \rightsquigarrow_{G^R} x$, donc $x \leftrightarrow_G y \Leftrightarrow x \leftrightarrow_{G^R} y$. Les deux relations sont égales et ont donc les mêmes classes d'équivalence.

□

2.2.5 Algorithme de Kosaraju

Algorithme 2.33 — : Kosaraju

Entrée : un graphe orienté $G = (S, A)$.

Déroulement :

- On effectue un parcours complet de G : tant qu'il y a des sommets inconnus, on lance un parcours depuis un sommet inconnu.
- On trie S par **ordre décroissant** de temps de sortie.
- On initialise l'état d'un parcours pour G^R .
- $Comp \leftarrow \emptyset$
- Pour chaque sommet x dans cet ordre, s'il est inconnu (dans G^R)
 - On lance un parcours dans G^R à partir de x en notant les sommets découverts dans la liste C .
 - $Comp \leftarrow Comp \cup \{C\}$
- On renvoie $Comp$.

Exemple 2.34

Sur l'exemple précédent, l'ordre décroissant des temps de sortie est $e, i, n, j, m, a, b, f, g, c, h, l, p, o, k, d$. Dans G^R :

- le parcours depuis e ne découvre que e , qui n'a pas d'arête entrante dans G ;
- le parcours depuis i découvre $\{i, m, j, n\}$ (l'arête $i \rightarrow e$ de G^R mène à un sommet déjà traité) ;
- celui depuis a découvre $\{a, g, f, b\}$;
- celui depuis c découvre $\{c, d, h, k, o, l\}$;
- celui depuis p découvre $\{p\}$.

On retrouve bien les cinq composantes fortement connexes.

Théorème 2.35

$Comp = CFC(G)$.

Preuve On montre l'invariant suivant pour la dernière boucle : $Comp \subset CFC(G)$ et les sommets déjà découverts dans G^R sont exactement les éléments des composantes de $Comp$. Au départ, comme $Comp = \emptyset$, il est trivialement vérifié et à la fin, comme tous les sommets auront été découverts, on aura nécessairement $Comp = CFC(G)$.

Supposons donc l'invariant vérifié et qu'on lance un parcours dans G^R à partir d'un sommet inconnu x . Notons $\bar{x}$ la composante contenant x . D'après l'invariant, aucun sommet de $\bar{x}$ n'est découvert. De plus, comme on traite les sommets par temps de sortie décroissant, tous les sommets de temps de sortie supérieur à $t_s(x)$ sont déjà découverts. Ainsi $t_s(x) = t_s(\bar{x})$ et $t_s(\bar{x}) > t_s(C')$ pour toute composante C' non encore découverte.

Comme $\bar{x}$ est fortement connexe dans G^R et que tous ses sommets sont inconnus, le théorème du chemin inconnu assure que le parcours depuis x découvre tout $\bar{x}$: $\bar{x} \subset C$.

Si, par l'absurde, C contient un sommet hors de $\bar{x}$, le chemin inconnu de x à ce sommet dans G^R sort de $\bar{x}$ par une arête $u \rightarrow v$ de G^R avec $u \in \bar{x}$, $v \notin \bar{x}$ et v inconnu. Dans G , c'est une arête $v \rightarrow u$ de la composante $\bar{v}$ vers $\bar{x}$, donc $t_s(\bar{v}) > t_s(\bar{x})$ par le théorème précédent. Mais v est inconnu, donc $\bar{v}$ n'est pas encore découverte d'après l'invariant, et $t_s(\bar{v}) < t_s(\bar{x})$: contradiction.

Ainsi $C = \bar{x}$ et l'invariant est préservé.

□

Remarque 2.36 La complexité de cet algorithme est dominée par les deux parcours complets et le calcul de G^R , on est donc en $O(|S| + |A|)$. Il n'est pas nécessaire de trier : il suffit d'ajouter chaque sommet en tête d'une liste au moment où il devient traité, cette liste est alors dans l'ordre décroissant des temps de sortie. C'est ce qui est fait dans le TP Kosaraju et 2-SAT.

Remarque 2.37 Une application importante est la résolution de 2-SAT en temps linéaire, évoquée dans le chapitre Calcul propositionnel : une formule dont les clauses ont deux littéraux est satisfiable si et seulement si aucune variable n'est dans la même composante fortement connexe que sa négation dans le graphe d'implication. La preuve et la construction d'une valuation sont l'objet du TP.

2.3 Couplage maximal dans un graphe biparti

2.3.1 Problème

Définition 2.38

Soit $G = (S, A)$ un graphe non orienté. On appelle **couplage** de G une partie $C \subset A$ telle que $\forall e, e' \in C, e \neq e' \Rightarrow e \cap e' = \emptyset$: deux arêtes distinctes du couplage n'ont pas d'extrémité commune.

On dit qu'un couplage est **maximal** pour G quand il est de cardinal maximal.

Remarque 2.39 Attention, dans de nombreux ouvrages, on parle de couplage **maximum** pour un couplage de cardinal maximal, et de couplage **maximal** pour un couplage maximal pour l'inclusion, c'est-à-dire auquel on ne peut pas ajouter d'arête. Un couplage de cardinal maximal est maximal pour l'inclusion, mais la réciproque est fautive : dans le chemin $a \sim b \sim c \sim d$, le couplage $\{\{b, c\}\}$ est maximal pour l'inclusion mais $\{\{a, b\}, \{c, d\}\}$ est de cardinal plus grand. Ici, maximal signifie toujours de cardinal maximal.

Rappel (voir le chapitre Graphes) :

Définition 2.40

Soit $G = (S, A)$ un graphe non orienté. On dit que G est **biparti** lorsqu'il existe $S_1, S_2 \subset S$ avec $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$ et toutes les arêtes relient un sommet de S_1 et un sommet de S_2 .

On se pose alors la question de déterminer un couplage maximal dans un graphe biparti. C'est un problème classique d'appariement. On peut ainsi citer le cas où on a des élèves et des écoles. On met une arête entre un élève et une école quand les deux veulent de l'autre. Un couplage maximal est alors une manière de placer le maximum d'élèves dans une école (en supposant que chaque école n'a qu'une place).

2.3.2 Chemin augmentant

Définition 2.41

Soit C un couplage d'un graphe et x un sommet. On dit que x est **libre** vis-à-vis de C si x n'appartient à aucune arête de C .

Définition 2.42

Soit $C \subset A$ un couplage d'un graphe.

Un chemin **élémentaire** de x à y composé des arêtes $(e_1, \dots, e_{2n+1})$, avec $n \in \mathbb{N}$, est dit **augmentant** si les arêtes de rang pair sont dans le couplage ($e_{2i} \in C$), les arêtes de rang impair n'y sont pas ($e_{2i+1} \notin C$), et si x et y sont libres pour C .

Exemple 2.43

Dans le graphe biparti suivant, avec $S_1 = \{0, 1, 2\}$ et $S_2 = \{3, 4\}$, le couplage $C = \{(1, 3)\}$ est représenté à gauche en trait épais. Le chemin $\varphi = 0 \sim 3 \sim 1 \sim 4$, dont les arêtes hors de C sont en pointillés, est augmentant : 0 et 4 sont libres et les arêtes alternent hors de C , dans C , hors de C . En échangeant le statut des arêtes de φ , on obtient le couplage de droite, qui a une arête de plus. Il est maximal puisque chaque arête contient un sommet de S_2 , qui n'a que deux sommets.

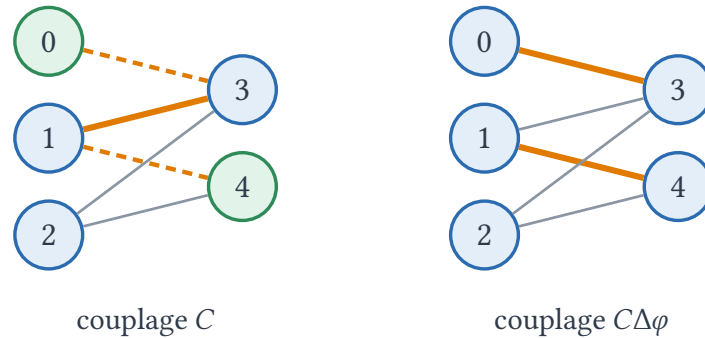


Figure 2.3 — Couplage C et chemin augmentant φ , puis couplage $C \Delta \varphi$

Lemme 2.44

Soit C, C' des couplages de $G = (S, A)$. On considère $G' = (S, C \Delta C')$, où $C \Delta C' = (C \setminus C') \cup (C' \setminus C)$ est la différence symétrique.

Les composantes connexes de G' sont :

- soit des sommets isolés ;
- soit des cycles **de longueur paire** alternant entre arêtes de C et de C' ;
- soit des chemins élémentaires alternant entre arêtes de C et de C' ayant des extrémités distinctes.

Preuve Les sommets de G' sont de degré au plus 2 car ils sont de degré au plus 1 dans (S, C) et dans (S, C') . Or une composante connexe d'un graphe dont les sommets sont de degré au plus 2 est un sommet isolé, un chemin élémentaire ou un cycle : on part d'un sommet de degré 1 s'il y en a un, sinon de n'importe quel sommet, et on avance d'arête en arête sans jamais pouvoir bifurquer.

De plus, deux arêtes de C (ou deux arêtes de C') ne peuvent avoir une extrémité en commun : un chemin de G' alterne donc forcément entre arêtes de C et de C' . Les cycles sont ainsi nécessairement de longueur paire.

□

Théorème 2.45 — (Berge, 1957)

C n'a pas de chemin augmentant si et seulement si C est un couplage maximal.

Preuve On va montrer la contraposée : C non maximal si et seulement si C a un chemin augmentant.

⇐) Supposons que G dispose d'un chemin augmentant φ . On considère C' la différence symétrique de C et de l'ensemble des arêtes de φ . Ainsi, C' est obtenu en retirant de C les n arêtes de φ qui sont dans C et en ajoutant les $n + 1$ arêtes de φ qui n'y sont pas : C' a une arête de plus que C .

De plus, C' est un couplage. En effet, comme φ est élémentaire, chaque sommet intérieur de φ appartenait à exactement une arête de C , qui était dans φ et a été retirée, et appartient à exactement une arête ajoutée. Les deux extrémités de φ étaient libres et appartiennent à exactement une arête ajoutée. Les autres sommets ne sont pas concernés. Ainsi C n'est pas maximal.

⇒) Supposons que C n'est pas maximal : il existe un couplage C' tel que $|C'| > |C|$. Le graphe $G' = (S, C \Delta C')$ contient plus d'arêtes de C' que de C , il a donc une composante qui contient plus d'arêtes de C' que de C . D'après le lemme, ce ne peut pas être un cycle : c'est un chemin élémentaire alternant, qui

commence et finit par une arête de $C' \setminus C$. Ses extrémités sont libres pour C : si une extrémité u appartenait à une arête $e \in C$, on aurait $e \notin C'$ puisque u appartient déjà à une arête de C' distincte de e (celle du chemin, qui n'est pas dans C). On aurait alors $e \in C \Delta C'$ et u serait de degré 2 dans G' , ce qui est impossible pour une extrémité. Ce chemin est donc augmentant pour C . □

Remarque 2.46 Ce théorème et sa preuve sont valables dans **tout** graphe, biparti ou non. C'est la recherche d'un chemin augmentant qui est plus simple dans le cas biparti : dans un graphe quelconque, il faut un algorithme plus subtil (algorithme d'Edmonds, hors programme).

2.3.3 Déterminer un chemin augmentant dans un graphe biparti

On considère ici un graphe biparti avec $S = S_1 \cup S_2$ et un couplage C .

Pour déterminer un chemin augmentant pour C , on oriente les arêtes $\{x, y\}$, avec $x \in S_1$ et $y \in S_2$, ainsi :

- $y \rightarrow x$ si $\{x, y\} \in C$;
- $x \rightarrow y$ sinon.

On rajoute également deux sommets :

- un sommet source noté s avec $s \rightarrow x$ pour tout sommet **libre** x de S_1 ;
- un sommet but noté t avec $y \rightarrow t$ pour tout sommet **libre** y de S_2 .

On appelle ce graphe orienté le **graphe résiduel** associé à C , comme dans le TP. On remarque qu'un sommet non libre y de S_2 est nécessairement de degré sortant 1 : son unique arête sortante est $y \rightarrow x$ où $\{x, y\} \in C$.

S'il existe un chemin de s à t dans ce graphe orienté, alors il existe un chemin **élémentaire** de s à t (on peut supprimer les portions entre deux passages par un même sommet ; le chemin donné par l'arbre d'un parcours est d'ailleurs directement élémentaire). Un tel chemin est de la forme :

$$s \rightarrow x_1 \rightarrow y_1 \rightarrow x_2 \rightarrow y_2 \dots \rightarrow x_n \rightarrow y_n \rightarrow t$$

avec :

- x_1 libre dans S_1 , car l'arête $s \rightarrow x_1$ n'existe que dans ce cas ;
- y_n libre dans S_2 , pour la même raison ;
- tous les autres x_i et y_j non libres : un y_j libre n'a pas d'arête sortante vers S_1 et un x_i libre n'a pas d'arête entrante depuis S_2 ;
- $\{x_i, y_i\} \notin C$, car ces arêtes vont de S_1 vers S_2 ;
- $\{y_i, x_{i+1}\} \in C$, car ces arêtes vont de S_2 vers S_1 ;
- les x_i et y_j deux à deux distincts, car le chemin est élémentaire.

Le chemin $x_1 \smile y_1 \smile \dots \smile y_n$ est donc augmentant. Réciproquement, un chemin augmentant, parcouru depuis son extrémité dans S_1 , donne un chemin de s à t dans le graphe résiduel.

Théorème 2.47

C admet un chemin augmentant si et seulement si t est accessible depuis s dans le graphe résiduel associé à C . On peut donc en trouver un, ou déterminer qu'il n'en existe pas, en $O(|S| + |A|)$ par un parcours.

On en déduit un algorithme de calcul d'un couplage maximal : on part du couplage vide et, tant qu'on trouve un chemin augmentant, on échange le statut de ses arêtes. Chaque augmentation fait grossir le couplage d'une arête et un couplage a au plus $\min(|S_1|, |S_2|)$ arêtes : on fait au plus $|S|/2$ augmentations, d'où une complexité en $O(|S|(|S| + |A|))$. D'après le théorème de Berge, le couplage obtenu est maximal.

En pratique, comme dans le **TP Couplage**, on cherche successivement un chemin augmentant depuis chaque sommet libre de S_1 , une seule fois chacun. C'est justifié par la propriété suivante.

Propriété 2.48

Soit $u \in S_1$ un sommet libre pour C duquel ne part aucun chemin augmentant. Si C' est obtenu à partir de C en échangeant le statut des arêtes d'un chemin augmentant φ , alors aucun chemin augmentant pour C' ne part de u .

Preuve On travaille dans le graphe résiduel sans s ni t . Un chemin augmentant pour C partant de u correspond à un chemin dans ce graphe de u vers un sommet libre de S_2 . Soit R l'ensemble des sommets accessibles depuis u dans le graphe résiduel associé à C : par hypothèse, R ne contient aucun sommet libre de S_2 et, par définition, aucune arête de ce graphe ne sort de R .

Le chemin φ correspond à un chemin du graphe résiduel allant d'un sommet libre de S_1 à un sommet libre v de S_2 . Il ne passe par aucun sommet de R : sinon v serait accessible depuis ce sommet, donc depuis u , et serait dans R .

Le graphe résiduel associé à C' s'obtient en renversant les arêtes de φ , dont aucune extrémité n'est dans R . Aucune arête ne sort donc de R dans ce nouveau graphe : les sommets accessibles depuis u sont encore dans R . Enfin, un sommet de S_2 libre pour C' l'était pour C , donc R ne contient toujours aucun sommet libre de S_2 , et u est toujours libre car $u \notin \varphi$. □

Remarque 2.49 En cherchant un plus court chemin augmentant par un parcours en largeur, et en augmentant plusieurs chemins disjoints à chaque phase, on obtient l'algorithme de Hopcroft-Karp en $O(\sqrt{|S|} |A|)$, hors programme.

2.4 Exercices

Exercice

On considère un chemin entre deux sommets x et y dans un graphe non orienté pondéré. La **largeur** de ce chemin est le plus petit poids des arêtes présentes dans ce chemin. Le chemin vide de x à x est de largeur $+\infty$.

La **distance de goulot d'étranglement** entre x et y est la largeur maximale d'un chemin de x à y . S'il n'en existe pas, cette distance est $-\infty$.

1. Prouver qu'un arbre couvrant de poids **maximal** d'un graphe connexe contient un chemin le plus large entre toute paire de sommets.
2. Décrire un algorithme pour résoudre en temps $O(|S| + |A|)$ le problème suivant : étant donné un graphe non orienté pondéré $G = (S, A)$, $x, y \in S$ et $W \in \mathbb{R}$, est-ce que la distance de goulot d'étranglement entre x et y est inférieure ou égale à W ?
3. On suppose que la distance de goulot d'étranglement entre $x \neq y$ est $B \in \mathbb{R}$.
 1. Prouver que la suppression d'une arête de poids strictement inférieur à B ne change pas cette distance.
 2. Prouver que la contraction d'une arête de poids strictement supérieur à B ne change pas cette distance. La contraction d'une arête $\{u, v\}$ revient à identifier u et v ; si cette contraction crée des arêtes parallèles, on ne conservera que l'arête de plus grand poids.

Corrigé

1. On considère un arbre couvrant de poids maximal T et deux sommets distincts x, y . Supposons par l'absurde que l'unique chemin de x à y dans T ne soit pas un chemin le plus large : sa largeur est strictement inférieure à la distance de goulot d'étranglement B . Son arête de plus petit poids e vérifie donc $w(e) < B$. On considère $T' = T \setminus \{e\}$, qui a exactement deux composantes connexes, l'une contenant x et l'autre y . Un chemin le plus large de x à y , de largeur B , contient forcément une arête

e' reliant ces deux composantes, avec $e' \notin T$ et $w(e') \geq B > w(e)$. Alors $T'' = T' \cup \{e'\}$ est connexe à $|S| - 1$ arêtes, c'est un arbre couvrant avec $w(T'') = w(T) + w(e') - w(e) > w(T)$. Contradiction.

2. La distance est strictement supérieure à W si et seulement s'il existe un chemin de x à y dont toutes les arêtes sont de poids strictement supérieur à W (pour $x = y$, le chemin vide convient et la distance vaut $+\infty$). On effectue donc un parcours depuis x en ignorant les arêtes de poids inférieur ou égal à W , et on répond « oui » si et seulement si y n'est pas atteint. C'est un parcours, en $O(|S| + |A|)$.
3. Pour les deux sous-questions :
 - 3.1. Supprimer une arête ne fait que retirer des chemins, donc la distance ne peut que diminuer. Mais un chemin de largeur B ne contient aucune arête de poids strictement inférieur à B : il existe encore après la suppression, et la distance reste B .
 - 3.2. L'arête contractée $e = \{u, v\}$ ne relie pas x et y : sinon on aurait $B \geq w(e) > B$. Tout chemin de G donne un chemin du graphe contracté passant par des arêtes de poids au moins aussi grand (seule e disparaît, et on garde le plus grand poids parmi les arêtes parallèles), donc de largeur au moins égale : la distance ne diminue pas. Réciproquement, un chemin du graphe contracté se relève en un chemin de G en insérant éventuellement l'arête e là où il passe par le sommet fusionné. Sa largeur est au moins le minimum de la largeur initiale et de $w(e) > B$. Si le chemin contracté avait une largeur strictement supérieure à B , on obtiendrait dans G un chemin de largeur strictement supérieure à B : c'est impossible. La distance reste donc B .

2.5 Travaux pratiques

- Algorithme de Kruskal, union-find (C)
- Algorithme de Kosaraju et 2-SAT (OCaml)
- Couplage maximal dans un graphe biparti (OCaml)
- Algorithme de Ford-Fulkerson (C)

Algorithme de Kruskal, union-find

3.1 Algorithme de Kruskal, union-find

Ce TP accompagne le chapitre *Algorithmique avancée des graphes* et utilise la structure du chapitre *Union-Find*.

3.1.1 Union-Find

On va définir un type

```
struct union_find {
    int *parent;
    int *rank;
    int nelements;
};
typedef struct union_find union_find;
```

Question 1 Écrire des fonctions de prototypes :

```
union_find uf_makesets(int n);
int uf_find(union_find uf, int x);
void uf_union(union_find uf, int x, int y);
void uf_free(union_find uf);
```

Ces fonctions vont réaliser les opérations vues en classe. Pour makesets on va réaliser des makeset des éléments de 0 à n-1 d'un coup en créant la valeur de type union_find (et notamment en allouant la mémoire). uf_free va alors libérer toute la mémoire allouée dans une telle structure.

Corrigé

```
union_find uf_makesets(int n)
{
    union_find uf;
    uf.parent = malloc(sizeof(int) * n);
    uf.rank = malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++)
    {
        uf.parent[i] = i;
        uf.rank[i] = 0;
    }
    uf.nelements = n;
    return uf;
}

int uf_find(union_find uf, int x)
{
    if (uf.parent[x] != x)
        uf.parent[x] = uf_find(uf, uf.parent[x]);
}
```

```

    return uf.parent[x];
}

void uf_union(union_find uf, int x, int y)
{
    int rx = uf_find(uf, x);
    int ry = uf_find(uf, y);
    assert(rx != ry);

    if (uf.rank[rx] > uf.rank[ry])
        uf.parent[ry] = rx;
    else {
        uf.parent[rx] = ry;
        if (uf.rank[rx] == uf.rank[ry])
            uf.rank[ry] = uf.rank[ry] + 1;
    }
}

void uf_free(union_find uf)
{
    free(uf.parent);
    free(uf.rank);
}

```

Ici, on a choisi d'interdire l'union de deux éléments déjà dans le même ensemble avec un assert : c'est à l'appelant de tester `uf_find(uf, x) != uf_find(uf, y)` au préalable. Le chapitre Union-Find fait le choix inverse de rendre cet appel inoffensif. Il faut inclure `stdlib.h` et `assert.h`.

Question 2 Écrire des jeux de tests pour ces fonctions. Avec des assert par exemple.

Question 3 On veut comparer les différentes optimisations de cette structure de données. Pour cela, on rajoute deux nouveaux paramètres booléens `compress` et `weighted` qui vont indiquer respectivement si on effectue de la compression de chemin ou si on utilise les rangs pour équilibrer lors des unions. Reprendre les fonctions précédentes pour qu'elles tiennent compte de ces paramètres. On pourra remplacer le prototype de `makesets` pour initialiser les valeurs : `union_find uf_makesets(int n, bool compress, bool weighted);`

Corrigé

On ajoute les deux booléens à la structure (et on inclut `stdbool.h`) :

```

struct union_find {
    int *parent;
    int *rank;
    int nelements;
    bool compress;
    bool weighted;
};
typedef struct union_find union_find;

union_find uf_makesets(int n, bool compress, bool weighted)
{
    union_find uf;
    uf.compress = compress;
    uf.weighted = weighted;
    uf.parent = malloc(sizeof(int) * n);
    uf.rank = malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++)
    {
        uf.parent[i] = i;
        uf.rank[i] = 0;
    }
    uf.nelements = n;
    return uf;
}

```



```

}

int uf_find(union_find uf, int x)
{
    if (uf.compress) {
        if (uf.parent[x] != x)
            uf.parent[x] = uf_find(uf, uf.parent[x]);
        return uf.parent[x];
    } else {
        while(uf.parent[x] != x)
            x = uf.parent[x];
        return x;
    }
}

void uf_union(union_find uf, int x, int y)
{
    int rx = uf_find(uf, x);
    int ry = uf_find(uf, y);
    assert(rx != ry);

    if (!uf.weighted || uf.rank[rx] > uf.rank[ry])
        uf.parent[ry] = rx;
    else {
        uf.parent[rx] = ry;
        if (uf.rank[rx] == uf.rank[ry])
            uf.rank[ry] = uf.rank[ry] + 1;
    }
}

```

uf_free est inchangée.

Question 4 Écrire une fonction

```
int *random_tests(int n, int m);
```

qui génère un jeu de test aléatoire de m unions pour n valeurs. Ce jeu de test sera représenté par un tableau t de 2m entiers indiquant qu'on souhaite effectuer des unions entre t[2*i] et t[2*i+1]. **On ne demande pas que les unions soient valides ici.**

Rappel : on peut utiliser rand() % A pour générer une valeur aléatoire entière entre 0 et A-1.

Corrigé

```

int *random_tests(int n, int m)
{
    int *fusions = malloc(sizeof(int) * 2 * m);

    for(int i = 0; i < 2*m; i++)
    {
        fusions[i] = rand() % n;
    }

    return fusions;
}

```

Question 5 Écrire une fonction

```
void apply_tests(union_find uf, int *tests, int m);
```

qui va appliquer les tests qu'on vient de générer sur un union_find qui aura été initialisé par le bon appel à uf_makesets.

Attention ici il faudra faire attention à ne faire l'union que si elle est valide, c'est-à-dire uniquement lorsqu'elle fusionne deux arbres de la forêt.

Corrigé

```
void apply_tests(union_find uf, int *tests, int m)
{
    for(int i = 0; i < m; i++)
    {
        int x = tests[2*i];
        int y = tests[2*i+1];

        if (uf_find(uf, x) != uf_find(uf, y))
            uf_union(uf, x, y);
    }
}
```

Il est possible de mesurer des temps simplement en C à l'aide de la fonction `clock_t clock()`; définie dans `time.h`.

Cette fonction renvoie le temps processeur consommé par le programme, exprimé en ticks d'horloge ; une constante `CLOCKS_PER_SEC` permet alors d'en déduire un nombre de secondes ainsi :

```
clock_t start = clock();
// tâche à mesurer en temps
clock_t end = clock();
// le nombre flottant de secondes écoulées
double elapsed = ((double)(end-start)/CLOCKS_PER_SEC);
```

Question 6 En déduire des tracés de comparaison du coût moyen en fonction de n pour $2n$ unions. On comparera avec/sans l'union pondérée et la compression de chemins (quatre choix donc). Attention, il sera sûrement impossible de mesurer le temps pris par la version sans optimisation pour de grandes valeurs de n .

Remarque 3.1 Pour tracer des courbes, on peut utiliser `gnuplot` ou faire un export vers Python avec `matplotlib`. On peut même faire un export en CSV dans la console et copier-coller dans un tableur.

La partie suivante présente rapidement `gnuplot`.



Remarque 3.2 À titre indicatif, pour $n = 20\,000$ et $2n$ unions aléatoires, on a mesuré environ 0,6 s sans optimisation contre moins de 0,01 s dès qu'une des deux optimisations est activée. Sans optimisation, la hauteur des arbres peut devenir linéaire et chaque `find` coûte alors $O(n)$.

Gnuplot en bref

Si on a un fichier `data.dat` avec des valeurs séparées en colonnes par des tabulations (`'\t'`). Par exemple :

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

on peut procéder ainsi pour afficher un tracé avec `gnuplot` :

```
$ gnuplot
gnuplot> plot "data.dat"
```

Par défaut, il va tracer la deuxième colonne en fonction de la première.

Pour tracer la colonne j en fonction de la colonne i :

```
gnuplot> plot "data.dat" using i:j
```

On peut demander de tracer des lignes avec :

```
gnuplot> plot "data.dat" using i:j with linespoint
```

Tout ou presque peut être abrégé donc la commande précédente peut s'écrire :

```
gnuplot> p "data.dat" u i:j w lp
```

Pour tracer plusieurs courbes, on peut mettre les descriptions à la suite :

```
gnuplot> p "data.dat" u 1:2 w lp, "data.dat" u 1:3 w lp
```

On peut aussi utiliser des boucles pour faire ça :

```
gnuplot> p for [col=2:3] "data.dat" using 1:col w lp
```

Pour écrire un fichier, on peut préciser un terminal de sortie, par exemple en png, avec :

```
gnuplot> set terminal pngcairo size 500,500
gnuplot> set output "fichier.png"
```

3.1.2 Arbre couvrant de poids minimal

On considère un graphe de points du plan dont les arêtes sont données dans un tableau :

```
struct edge {
    int u;
    int v;
};
typedef struct edge edge;

struct vertex {
    double x;
    double y;
};
typedef struct vertex vertex;

struct graph {
    vertex *vertices;
    int nvertices;
    edge *edges;
    int nedges;
};
typedef struct graph graph;
```

Ainsi, l'arête e relie les sommets d'indice $e.u$ et $e.v$. Un sommet v correspond au point du plan $(v.x, v.y)$.

Question 7 Écrire une fonction

```
double w(graph *g, edge *e);
```

qui renvoie la distance euclidienne entre les deux sommets donnés par une arête.

Attention pour avoir `sqrt` il faudra `math.h` et lier avec `-lm`.

Corrigé

```
double w(graph *g, edge *e)
{
    vertex a = g->vertices[e->u];
    vertex b = g->vertices[e->v];
    return sqrt( (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) );
}
```

Question 8 Écrire une fonction

```
graph randomize(int n, int m, double w, double h);
```

qui crée un graphe aléatoirement avec n sommets ($n \geq 2$) et m arêtes. Les sommets auront des coordonnées dans $[0, w] \times [0, h]$.

Note : pour N assez grand, on peut générer aléatoirement un double dans $[0, w]$ ainsi : $w * ((\text{double})(\text{rand}() \% (N+1)) / N)$

Corrigé

```
graph randomize(int n, int m, double w, double h)
{
    int N = 10000;
    graph g;

    g.nvertices = n;
    g.vertices = malloc(sizeof(vertex) * n);
    for(int i = 0; i < n; i++)
    {
        g.vertices[i].x = w * ((double)(rand() % (N+1)) / N);
        g.vertices[i].y = h * ((double)(rand() % (N+1)) / N);
    }
    g.nedges = m;
    g.edges = malloc(sizeof(edge) * m);
    for(int i = 0; i < m; i++)
    {
        int u = rand() % n;
        int v = rand() % (n-1);
        if (v == u) v = v + 1;
        g.edges[i].u = u;
        g.edges[i].v = v;
    }

    return g;
}
```

On évite ainsi les boucles $u \sim u$: si $v = u$, alors $u \leq n - 2$ et $v + 1 \leq n - 1$. Le paramètre w masque ici la fonction w de la question précédente, ce qui ne pose pas de problème puisqu'on ne l'utilise pas dans cette fonction. Le graphe obtenu peut contenir des arêtes multiples et n'est pas forcément connexe.

Question 9 Pour trier les arêtes on va utiliser `qsort` de `stdlib.h`. Cette fonction pose plusieurs problèmes :

- il faut faire des conversions depuis `void *` explicitement ;
- comme la fonction de comparaison ne prend pas de paramètre supplémentaire, on doit utiliser des variables globales.

On vous demande ainsi d'ajouter dans votre fichier :

```
graph *global_graph = NULL;
int compare_edges(const void *e1, const void *e2)
{
    double w1 = w(global_graph, (edge *)e1);
    double w2 = w(global_graph, (edge *)e2);
    if (w1 < w2)
        return -1;
    if (w1 > w2)
        return 1;
    return 0;
}
```

et pour trier les arêtes du graphe g , on pourra alors écrire :

```
global_graph = g;
qsort(g->edges, g->nedges, sizeof(edge), compare_edges);
```

Question 10 On sait qu'un arbre couvrant contient n vertices-1 arêtes, on va donc stocker un arbre couvrant comme un tableau d'entiers `mst` de cette longueur. `mst[i]` sera l'indice de la i -ème arête dans l'arbre.

Écrire ainsi une fonction

```
int *kruskal(graph *g);
```

qui renvoie le tableau `mst` en l'ayant alloué préalablement.

On s'assurera que le graphe est connexe en vérifiant que la structure `union_find` utilisée ne contient qu'un seul arbre.

Corrigé

On compte les arbres de la forêt en comptant les racines :

```
int uf_count_trees(union_find uf)
{
    int count = 0;
    for(int i = 0; i < uf.nelements; i++)
        if (uf.parent[i] == i)
            count++;
    return count;
}

int *kruskal(graph *g)
{
    global_graph = g;
    qsort(g->edges, g->nedges, sizeof(edge), compare_edges);

    int *mst = malloc(sizeof(int) * (g->nvertices - 1));
    union_find uf = uf_makesets(g->nvertices, true, true);
    int mst_size = 0;

    for (int i = 0; i < g->nedges; i++)
    {
        int ru = uf_find(uf, g->edges[i].u);
        int rv = uf_find(uf, g->edges[i].v);
        if (ru != rv)
        {
            uf_union(uf, ru, rv);
            mst[mst_size] = i;
            mst_size++;
        }
    }
    assert(uf_count_trees(uf) == 1);
    uf_free(uf);

    return mst;
}
```

Attention, les indices stockés dans `mst` sont ceux des arêtes **après** le tri : `kruskal` modifie l'ordre du tableau `g->edges`.



Remarque 3.3 Pour tester `kruskal`, on peut comparer, sur de nombreux petits graphes aléatoires connexes (4 à 6 sommets), le poids de l'arbre obtenu avec le minimum des poids des parties à $n - 1$ arêtes qui forment un arbre, obtenu par énumération de toutes ces parties. C'est ainsi qu'ont été testés les corrigés de ce TP.

Question 11 Faire des essais de graphes aléatoires en affichant le graphe et par-dessus dans une autre couleur l'arbre couvrant de poids minimal obtenu.

Algorithme de Kosaraju et 2-SAT

4.1 Algorithme de Kosaraju et 2-SAT

Ce TP accompagne le chapitre [Algorithmique avancée des graphes](#). Il est disponible sous forme de notebook Jupyter.

4.1.1 Kosaraju

On va considérer un type graphe où les sommets ne sont pas forcément des entiers et où on stocke une table de hachage pour faire la correspondance entre la valeur d'un sommet et son indice.

```
type 'a graphe = {  
  sommets : 'a array;  
  sommets_indices : ('a, int) Hashtbl.t;  
  ladj : int list array  
}
```

Question 1 Écrire une fonction qui étant donné un tableau `t` va construire la table des associations `t.(i) -> i` qui permettra ainsi de remonter du tableau à l'indice.

Note C'est le moment de retourner apprendre par cœur la [documentation du module Hashtbl](#).

Corrigé

```
let table_indices (t:'a array) : ('a, int) Hashtbl.t =  
  let h = Hashtbl.create (Array.length t) in  
  for i = 0 to Array.length t - 1 do  
    Hashtbl.add h t.(i) i  
  done;  
  h
```

Question 2 Écrire une fonction `indice` qui prend un graphe et un sommet et renvoie son indice en utilisant la table `sommets_indices`.

Corrigé

```
let indice g a = Hashtbl.find g.sommets_indices a
```

Question 3 En déduire une fonction `cree_graphe` qui va prendre un tableau `sommets` et créer un graphe sans arêtes dont ce sont les sommets.

Corrigé

```
let cree_graphe sommets =  
  {
```

```

sommets = sommets;
sommets_indices = table_indices sommets;
ladj = Array.make (Array.length sommets) []
    }
    
```

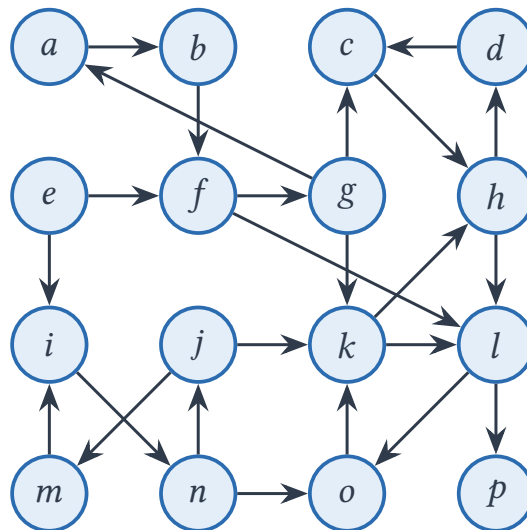
Question 4 Écrire une fonction `ajoute_arete` telle que `ajoute_arete g a b` ajoute l'arête $a \rightarrow b$ où les sommets sont donnés par leur valeur et non pas par leur indice.

Corrigé

```

let ajoute_arete g a b =
  let ia = indice g a in
  let ib = indice g b in
  if not (List.mem ib g.ladj.(ia))
  then g.ladj.(ia) <- ib :: g.ladj.(ia)
    
```

Les deux cellules suivantes permettent de définir le graphe vu dans le cours en exemple :



```

let ex_cours = cree_graphe [|'a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p'|]
    
```

```

List.iter (fun (a,b) -> ajoute_arete ex_cours a b)
  [ ('a','b'); ('b','f'); ('g','a'); ('f','g');
    ('c','h'); ('g','c'); ('d','c'); ('h','d');
    ('e','f'); ('f','l'); ('e','i'); ('g','k');
    ('h','l'); ('i','n'); ('j','m'); ('j','k');
    ('k','l'); ('k','h'); ('l','o'); ('l','p');
    ('m','i'); ('n','j'); ('n','o'); ('o','k')]
    
```

Question 5 Écrire une fonction `graphe_miroir` qui renvoie le graphe miroir du graphe donné, c'est-à-dire qui renverse toutes les arêtes.

Corrigé

```

let graphe_miroir g =
  let gr = {
    sommets = g.sommets;
    sommets_indices = g.sommets_indices;
    ladj = Array.make (Array.length g.sommets) []
  } in
  for i = 0 to Array.length g.sommets - 1 do
    
```



```
List.iter (fun j -> gr.ladj.(j) <- i :: gr.ladj.(j)) g.ladj.(i)
done;
gr
```

Question 6 Écrire une fonction `dfs` qui va parcourir en profondeur un graphe et appliquer une fonction de **post-traitement**. Pas besoin de noter les temps de sortie puisqu'ils correspondent au moment où on effectue ce traitement.

Corrigé

```
let rec dfs g trait visites x =
  if not visites.(x)
  then begin
    visites.(x) <- true;
    List.iter (fun y ->
      if not visites.(y)
      then dfs g trait visites y
    ) g.ladj.(x);
    trait x
  end
end
```

Question 7 En déduire une fonction `kosaraju` qui renvoie un tableau indiquant le numéro de la composante fortement connexe de chaque sommet.

Attention, cette fonction va être un peu compliquée, on indique les grandes lignes ici :

- on initialise un tableau de visites ;
- on va construire une liste `ordre` des indices de sommets rencontrés lors d'un premier DFS de tout le graphe grâce à un post-traitement (avec une référence) ;
- on calcule G^R et on initialise un tableau de visites ;
- on fait des DFS dans G^R mais en prenant les sommets depuis la liste `ordre`. En post-traitement, on va remplir un tableau `comp` où `comp.(x)` sera le numéro de la composante fortement connexe de x .

On commence par numéroter les composantes à 1.

Corrigé

```
let kosaraju g =
  let n = Array.length g.sommets in
  let ordre = ref [] in
  let visites = Array.make n false in
  for i = 0 to n-1 do
    if not visites.(i)
    then dfs g (fun x -> ordre := x :: !ordre) visites i
  done;
  let gr = graphe_miroir g in
  let visites = Array.make n false in
  let ncomp = ref 0 in
  let comp = Array.make n 0 in
  List.iter (fun ia ->
    if not visites.(ia)
    then begin
      incr ncomp;
      dfs gr (fun x -> comp.(x) <- !ncomp) visites ia
    end
  ) !ordre;
  comp
```

Comme on ajoute chaque sommet en tête de `ordre` au moment où il devient traité, `ordre` contient les sommets par temps de sortie décroissant.

Pour tester :

```
assert (kosaraju ex_cours = [|3; 3; 4; 4; 1; 3; 3; 4; 2; 2; 4; 4; 2; 2; 4; 5|]);
"OK"
```

Question 8 Cette représentation des composantes va nous être utile mais elle n'est pas très maniable. Écrire une fonction `listes_comp` telle que `listes_comp sommets comp` renvoie le tableau des composantes données sous forme de listes de sommets.

Corrigé

```
let listes_comp sommets comp =
  let n = Array.fold_left max 0 comp in
  let comp_l = Array.make n [] in
  for i = 0 to Array.length comp - 1 do
    let n = comp.(i) in
    comp_l.(n-1) <- sommets.(i) :: comp_l.(n-1)
  done;
  comp_l
```

Pour tester :

```
assert (listes_comp ex_cours.sommets (kosaraju ex_cours) =
  [|['e']; ['n'; 'm'; 'j'; 'i'];
   ['g'; 'f'; 'b'; 'a']; ['o'; 'l'; 'k'; 'h'; 'd'; 'c']; ['p']|]);
"OK"
```



Remarque 4.1 Pour tester `kosaraju` plus largement, on peut comparer son résultat sur des petits graphes aléatoires avec une méthode naïve : on calcule pour chaque sommet x l'ensemble des sommets accessibles par un parcours, et on vérifie que $\text{comp.}(x) = \text{comp.}(y)$ si et seulement si y est accessible depuis x et x depuis y . On peut aussi vérifier que pour toute arête $x \rightarrow y$, $\text{comp.}(x) \leq \text{comp.}(y)$: c'est la propriété de numérotation utilisée pour 2-SAT ci-dessous.

4.1.2 2-SAT

On va résoudre 2-SAT en utilisant l'algorithme de Kosaraju. Tout d'abord, on rappelle que les formules de 2-SAT sont des conjonctions de clauses comportant chacune deux littéraux (voir le chapitre Calcul propositionnel).

Exemple : $f = (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$

On reprend un type proche de ce qui a été fait l'an dernier, sauf que les clauses sont des couples de littéraux.

```
type lit = Pos of char | Neg of char
type clause = lit * lit
type formule = clause list
```

La formule donnée au-dessus s'écrit alors :

```
let f = [ (Pos 'a', Neg 'b'); (Neg 'a', Pos 'b'); (Neg 'a', Neg 'b'); (Pos 'a', Neg 'c') ]
```

Question 9 Écrire une fonction `label : lit -> char` qui renvoie l'étiquette d'un littéral et une fonction `neg : lit -> lit` qui renvoie le littéral opposé d'un littéral donné.

Corrigé

```
let label = function Pos a | Neg a -> a

let neg = function Pos a -> Neg a | Neg a -> Pos a
```

Question 10 Écrire une fonction `variables` qui renvoie les étiquettes de tous les littéraux **sans répétitions**. On pourra utiliser `List.sort_uniq Stdlib.compare`.

Corrigé

```
let variables f =
  let rec var_aux f =
    match f with
    | [] -> []
    | c::f' -> var_clause c @ var_aux f'
  and var_clause (l1,l2) = [label l1; label l2]
  in List.sort_uniq Stdlib.compare (var_aux f)
```

Question 11 On peut écrire $a \vee b$ sous la forme implicative $(\neg a \rightarrow b) \wedge (\neg b \rightarrow a)$.

En faisant cela, on peut alors construire un graphe sur les littéraux où $l \rightarrow l'$ quand cette implication apparaît dans la formule.

Écrire une fonction `implication_graphe` qui étant donnée `f` renvoie ce graphe.

Corrigé

```
let implication_graphe f =
  let v = variables f in
  let s = List.concat (List.map (fun x -> [Pos x; Neg x]) v) in
  let sommets = Array.of_list s in
  let g = cree_graphe sommets in
  List.iter (fun (a,b) ->
    ajoute_arete g (neg a) b;
    ajoute_arete g (neg b) a
  ) f;
  g
```

Les sommets sont rangés par variable : le sommet d'indice $2i$ est `Pos x` et celui d'indice $2i + 1$ est `Neg x` pour la i -ème variable `x`.

On cherche alors une valuation des variables telle qu'il n'y ait pas d'arête $\top \rightarrow \perp$ d'un littéral vrai vers un littéral faux. En effet, toutes les autres affectations vérifient la clause correspondante. Ainsi, tous les littéraux d'une composante fortement connexe doivent nécessairement avoir la même valeur de vérité. Si $C \rightarrow C'$ pour deux composantes, on ne peut pas affecter la valeur vraie à C et fausse à C' . On remarque que le graphe est symétrique par contraposition, donc il existe une composante niée $\neg C$ pour chaque composante C .

L'idée pour résoudre 2-SAT est donc de calculer les composantes fortement connexes et de parcourir leur graphe acyclique en partant des puits, en affectant la valeur vraie à la première composante rencontrée entre C et $\neg C$.

Pour résoudre 2-SAT, on va appliquer Kosaraju sur le graphe d'implication de la formule.

```
kosaraju (implication_graphe f)
```

Sur ce graphe, on remarque que si deux littéraux opposés sont dans la même composante fortement connexe, alors la formule est insatisfiable. Si ce n'est pas le cas, on peut démontrer que la valuation suivante fonctionne :

$$\forall a \in V, v(a) = \begin{cases} \top & \text{si } comp(a) > comp(\neg a) \\ \perp & \text{sinon} \end{cases}$$

Cette affirmation repose sur deux propriétés :

- les composantes sont numérotées dans l'ordre croissant d'un tri topologique de leur graphe, qui est acyclique : pour toute arête $x \rightarrow y$, $comp(x) \leq comp(y)$;

- le graphe d'implication possède une symétrie par contraposition : si $a_1 \rightarrow a_2 \dots \rightarrow a_n$ alors $\neg a_n \rightarrow \neg a_{n-1} \dots \rightarrow \neg a_1$.

Théorème 4.2

Une formule de 2-SAT est satisfiable si et seulement si aucune variable a n'est dans la même composante fortement connexe que $\neg a$ dans son graphe d'implication. Dans ce cas, la valuation v ci-dessus est un modèle.

Preuve Si v est un modèle, chaque arête $l \rightarrow l'$ provient d'une clause $\neg l \vee l'$ satisfaite par v : si l est vrai, l' aussi. Par récurrence, deux littéraux reliés par un chemin $l \rightsquigarrow l'$ vérifient : si l est vrai, l' l'est. Si a et $\neg a$ étaient dans la même composante, on aurait $a \rightsquigarrow \neg a$ et $\neg a \rightsquigarrow a$, donc a vrai entraîne $\neg a$ vrai et réciproquement : impossible.

Réciproquement, supposons que $\text{comp}(a) \neq \text{comp}(\neg a)$ pour toute variable a . Montrons d'abord que la valuation v rend vrai le littéral l (positif ou négatif) si et seulement si $\text{comp}(l) > \text{comp}(\neg l)$. C'est la définition si $l = a$; si $l = \neg a$, l est vrai si et seulement si $\text{comp}(a) < \text{comp}(\neg a)$, c'est-à-dire $\text{comp}(l) > \text{comp}(\neg l)$ puisque les deux numéros sont distincts.

Soit une clause $l \vee l'$ et supposons par l'absurde que v rende l et l' faux : $\text{comp}(l) < \text{comp}(\neg l)$ et $\text{comp}(l') < \text{comp}(\neg l')$. Les arêtes $\neg l \rightarrow l'$ et $\neg l' \rightarrow l$ du graphe donnent, par la première propriété, $\text{comp}(\neg l) \leq \text{comp}(l')$ et $\text{comp}(\neg l') \leq \text{comp}(l)$. On obtient $\text{comp}(l) < \text{comp}(\neg l) \leq \text{comp}(l') < \text{comp}(\neg l') \leq \text{comp}(l)$, ce qui est absurde. Toutes les clauses sont donc satisfaites.

La première propriété découle de l'étude de Kosaraju dans le cours : la deuxième passe découvrir les composantes par temps de sortie décroissant, et une arête $x \rightarrow y$ entre deux composantes distinctes impose $t_s(\bar{y}) < t_s(\bar{x})$, donc $\text{comp}(x) < \text{comp}(y)$.

□

On remarque que la preuve n'utilise pas directement la symétrie : celle-ci garantit simplement que $\neg C$ est une composante, ce qui donne l'interprétation précédente en termes de composantes.

Question 12 En déduire une fonction `resout_2sat` qui étant donné une formule renvoie `None` si elle est insatisfiable, et `Some l` où `l` est une liste de couples (variable, valeur) représentant une valuation solution.

Corrigé

```
let resout_2sat f =
  let g = implication_graphe f in
  let comp = kosaraju g in
  let n = Array.length g.sommets / 2 in
  let valuation = ref [] in
  let insoluble = ref false in
  for i = 0 to n-1 do
    (* g.sommets.(2*i) est Pos a et g.sommets.(2*i+1) est Neg a *)
    let a = label g.sommets.(2*i) in
    if comp.(2*i) = comp.(2*i+1)
    then insoluble := true
    else valuation := (a, comp.(2*i) > comp.(2*i+1)) :: !valuation
  done;
  if !insoluble
  then None
  else Some (List.rev !valuation)
```

Sur l'exemple, `resout_2sat f` renvoie `Some [('a', false); ('b', false); ('c', false)]`, qui est bien un modèle de f .

Remarque 4.3 On peut tester `resout_2sat` contre une méthode naïve qui essaie toutes les valuations des variables : sur des milliers de formules aléatoires à au plus 6 variables, quand `resout_2sat` renvoie `Some`

 v , on vérifie que v satisfait la formule, et quand elle renvoie `None`, on vérifie qu'aucune valuation ne la satisfait.

Couplage maximal dans un graphe biparti

5.1 Couplage maximal dans un graphe biparti

Ce TP accompagne la partie [Couplage maximal dans un graphe biparti](#) du chapitre Algorithmique avancée des graphes.

5.1.1 Graphe biparti

On va considérer ici un graphe biparti donné par une liste d'arêtes sous la forme (source, but). Voici un exemple :

```
[ (0,3); (1,3); (1,4); (2,3); (2,4) ]
```

En commençant à numéroté à 0, on peut donc supposer que le nombre de sommets est égal au plus grand entier apparaissant dans un couple plus un.

Question 1 Écrire une fonction `nombre_sommets : (int * int) list -> int` qui calcule ce nombre.

Corrigé

```
let rec nombre_sommets l =
  match l with
  | [] -> 0
  | (a,b)::q -> let n = nombre_sommets q in
    max (1 + max a b) n
```

Pour tester :

```
assert(nombre_sommets [(0,3);(1,3);(1,4);(2,3);(2,4)] = 5);
assert(nombre_sommets [(0,8);(0,7);(2,6);(2,9);(3,8);(4,8);(4,9);(5,11)] = 12);
"OK"
```

On va maintenant récupérer la liste des premières composantes et la liste des secondes composantes **sans répétition**.

Pour cela, on commence par écrire une fonction permettant de réaliser `t :: q` en omettant `t` en cas de répétition.

Question 2 Écrire une fonction `cons_uniq : 'a -> 'a list -> 'a list` telle que `cons_uniq t q` renvoie `t::q` si `t` n'est pas dans la liste `q` et `q` sinon.

Corrigé

```
let cons_uniq t q =
  if List.mem t q
```

```

    then q
  else t::q

```

Pour tester :

```

assert(cons_uniq 0 [] = [0]);
assert(cons_uniq 0 [0] = [0]);
assert(cons_uniq 0 [1] = [0;1]);
"OK"

```

Question 3 Écrire une fonction `separe_liste : (int * int) list -> int list * int list` qui prend en argument une liste de couples d'entiers et renvoie le couple des listes **sans répétitions** des premières et secondes composantes.

Corrigé

```

let rec separe_liste l =
  match l with
  | [] -> [], []
  | (a,b)::q -> let l1, l2 = separe_liste q in
    (cons_uniq a l1, cons_uniq b l2)

```

Pour tester :

```

assert(separe_liste [(0,3);(1,3);(1,4);(2,3);(2,4)] = ([0;1;2],[3;4]));
assert(separe_liste [(0,8);(0,7);(2,6);(2,9);(3,8);(4,8);(4,9);(5,11)] =
  ([0;2;3;4;5],[7;6;8;9;11]));
"OK"

```

Question 4 Écrire une fonction `intersection : 'a list -> 'a list -> 'a list` qui prend en arguments deux listes sans répétitions et renvoie une liste contenant les éléments présents dans les deux listes.

Corrigé

```

let intersection l1 l2 =
  List.filter (fun x -> List.mem x l2) l1

```

Pour tester :

```

assert(intersection [1;2;3] [0;1] = [1]);
assert(intersection [1;2;3] [1;2] = [1;2]);
assert(intersection [1;2;3] [4;5] = []);
assert(intersection [] [3;4] = []);
assert(intersection [1;2;3] [] = []);
"OK"

```

Question 5 En déduire une fonction `biparti : (int * int) list -> bool` qui vérifie qu'une liste d'arêtes permet effectivement de représenter un graphe biparti où les sommets sont d'un côté les premières composantes et de l'autre les secondes.

Corrigé

```

let biparti l =
  let l1, l2 = separe_liste l in intersection l1 l2 = []

```

Pour tester :


```
assert(biparti [(0,3);(1,3);(1,4);(2,3);(2,4)]);
assert(not (biparti [(0,3);(1,0);(1,4);(2,3);(2,4)]));
"OK"
```

De cette liste d'arêtes représentant un graphe biparti à n sommets, on va en déduire le graphe lui-même en le représentant sous la forme d'un couple $(\text{ladj}, \text{sources})$ où ladj est une représentation en listes d'adjacence et sources est un tableau de booléens indiquant si un sommet appartient à la première partie du graphe biparti, c'est-à-dire s'il est la première composante d'un des couples d'arêtes.

Ainsi, les sommets qui n'apparaissent pas dans les arêtes sont associés implicitement à la seconde partie, ce qui ne sera pas gênant dans la suite.

Question 6 Écrire une fonction `graphe_arettes : (int * int) list -> int list array` qui renvoie le tableau des listes d'adjacence d'un graphe **non orienté** donné sous la forme d'une liste d'arêtes.

Corrigé

```
let graphe_arettes l =
  let n = nombre_sommets l in
  let ladj = Array.make n [] in
  List.iter (fun (a,b) ->
    ladj.(a) <- b :: ladj.(a); ladj.(b) <- a :: ladj.(b)) l;
  ladj
```

Pour tester :

```
assert(graphe_arettes [(0,3);(1,3);(1,4);(2,3);(2,4)]
  = [| [3]; [4; 3]; [4; 3]; [2; 1; 0]; [2; 1] |]);
assert(graphe_arettes [(0,8);(0,7);(2,6);(2,9);(3,8);(4,8);(4,9);(5,11)]
  = [| [7; 8]; []; [9; 6]; [8]; [9; 8]; [11]; [2];
    [0]; [4; 3; 0]; [4; 2]; []; [5] |]);
"OK"
```

Question 7 En déduire une fonction `graphe_biparti : (int * int) list -> int list array * bool array` qui renvoie le couple $(\text{ladj}, \text{sources})$ représentant le graphe biparti.

Corrigé

```
let graphe_biparti l =
  let g = graphe_arettes l in
  let s1, _ = separe_liste l in
  let sources = Array.make (Array.length g) false in
  List.iter (fun i -> sources.(i) <- true) s1;
  g, sources
```

Pour tester :

```
assert (snd(graphe_biparti [(0,3);(1,3);(1,4);(2,3);(2,4)])
  = [| true; true; true; false; false |]);
assert (snd(graphe_biparti [(0,8);(0,7);(2,6);(2,9);(3,8);(4,8);(4,9);(5,11)])
  = [| true; false; true; true; true; true; false; false;
    false; false; false; false |]);
"OK"
```

5.1.2 Couplage maximal

On va programmer l'algorithme pour déterminer un couplage maximal par des bascules successives de chemins augmentants (voir la preuve du théorème de Berge dans le cours).

On va adopter deux représentations d'un couplage. La première est la représentation élémentaire comme sous-liste de la liste d'arêtes du graphe.

Par exemple $[(0,3); (1,4)]$ est un couplage pour la liste d'arêtes donnée plus haut.

Question 8 Écrire une fonction `est_couplage : (int * int) list -> bool` qui vérifie si une liste d'arêtes est un couplage en vérifiant qu'il n'y a pas d'arêtes ayant une extrémité commune.

Votre fonction devra être de complexité **linéaire**.

Corrigé

```
let est_couplage el =
  let n = nombre_sommets el in
  let libres = Array.make n true in
  let couplage = ref true in
  List.iter (fun (x,y) ->
    couplage := !couplage && libres.(x) && libres.(y);
    libres.(x) <- false; libres.(y) <- false) el;
  !couplage
```

La complexité est linéaire en la longueur de la liste plus le plus grand numéro de sommet, à cause de la création du tableau.

Pour tester :

```
assert( est_couplage [(0,1);(2,3)] );
assert( not(est_couplage [(0,1);(1,3)]) );
"OK"
```

L'autre représentation est un tableau indiquant pour un sommet i , soit `Some j` quand on a une arête $\{i, j\}$ dans le couplage et `None` sinon. En effet, un couplage définit une fonction partielle involutive de $S \rightarrow S$.

Ainsi, le tableau `[| Some 3; Some 4; None; Some 0; Some 1 |]` permet de représenter le couplage $[(0,3); (1,4)]$ en considérant que les sommets sont dans $\llbracket 0, 4 \rrbracket$.

Question 9 Écrire des fonctions permettant de passer d'une représentation à une autre :

- `couplage_liste_vers_tab : (int * int) list -> int option array ;`
- `couplage_tab_vers_liste : bool array -> int option array -> (int * int) list` ici, on a besoin de connaître les sources pour choisir comment placer les arêtes.

Corrigé

```
let couplage_liste_vers_tab l =
  let n = nombre_sommets l in
  let couplage = Array.make n None in
  List.iter (fun (a,b) ->
    couplage.(a) <- Some b;
    couplage.(b) <- Some a) l;
  couplage

let couplage_tab_vers_liste sources couplage =
  let cpl_aretes = ref [] in
  let n = Array.length sources in
  for i = 0 to n-1 do
    if sources.(i)
    then match couplage.(i) with
      | None -> ()
      | Some j -> cpl_aretes := (i,j) :: !cpl_aretes
  done;
  !cpl_aretes
```

Attention, `couplage_liste_vers_tab` renvoie un tableau dont la taille est le nombre de sommets apparaissant dans le couplage, qui peut être inférieur au nombre de sommets du graphe.

Pour tester :

```
assert( couplage_liste_vers_tab [ (0,3); (1,4) ]
      = [| Some 3; Some 4; None; Some 0; Some 1 |]);
assert( couplage_tab_vers_liste
      [| true; true; true; false; false |]
      [| Some 3; Some 4; None; Some 0; Some 1 |]
      = [ (1,4); (0,3) ]);
"OK"
```

On va définir ici le graphe résiduel associé à un couplage (voir le cours) de manière implicite avec une fonction permettant de décider si une arête $x \rightarrow y$ est dans le graphe résiduel. Les arêtes hors du couplage vont des sources vers les buts, celles du couplage des buts vers les sources ; on n'ajoute pas ici les sommets s et t du cours.

Attention il peut y avoir des arêtes $x \rightarrow z$ quand (x, y) est dans le couplage.

Question 10 Écrire une fonction

`arete_residuelle : bool array -> int option array -> int -> int -> bool`

telle que l'appel `arete_residuelle sources couplage x y` permette de décider si $x \rightarrow y$ est dans le graphe résiduel, en supposant que $\{x, y\}$ est une arête du graphe.

Corrigé

```
let arete_residuelle sources couplage x y =
  if sources.(x)
  then couplage.(x) <> Some y
  else couplage.(x) = Some y
```

Pour tester :

```
assert(arete_residuelle [|true;false|] [|None;None|] 0 1);
assert(not (arete_residuelle [|true;false|] [|None;None|] 1 0));
assert(arete_residuelle [|true;false|] [|Some 1; Some 0|] 1 0);
assert(not (arete_residuelle [|true;false|] [|Some 1; Some 0|] 0 1));
assert(arete_residuelle [|true;true;false;false|] [|Some 2; None; Some 0; None|] 0 3);
assert(not (arete_residuelle [|true;true;false;false|] [|Some 2; None; Some 0; None|] 0 2));
"OK"
```

Pour chercher un chemin augmentant, on va commencer par effectuer un parcours en profondeur récursif dans un graphe avec une notion d'arête implicite, une fonction comme `arete_residuelle`, et on va remplir un tableau de prédécesseurs où x est le prédécesseur de y si c'est le DFS depuis x qui a appelé le DFS depuis y .

Dans toute la suite, on suppose que le prédécesseur de x est x lui-même quand on a lancé le DFS initialement depuis le sommet x .

Question 11 Écrire une fonction

`dfs : int list array -> (int -> int -> bool) -> int option array -> int -> unit`

telle que `dfs g est_arete pred x` effectue un DFS depuis le sommet x dans le graphe donné sous forme de listes d'adjacence par g , avec une fonction `est_arete` permettant de tester si une arête est à considérer dans le parcours et en remplissant le tableau des prédécesseurs `pred`, qui est un `int option array` car la valeur vaut `None` tant que le sommet n'est pas découvert.

Rappel : on a spécifié que `pred.(x)` valait `Some x` si x était un des sommets initiaux sur lesquels on a commencé le DFS.

Corrigé

```
let rec dfs g est_arete pred x =
  List.iter (fun y ->
    if pred.(y) = None && est_arete x y
    then begin
      pred.(y) <- Some x;
      dfs g est_arete pred y
    end)
  g.(x)
```

C'est à l'appelant d'initialiser `pred.(x)` à `Some x` avant l'appel initial.

Question 12 Écrire une fonction

`remonte : int option array -> int -> int list`

telle que `remonte pred x` renvoie la liste des sommets allant du point de départ d'un DFS ayant rempli `pred` jusqu'à `x` en remontant la relation de prédécesseurs.

Corrigé

```
let remonte pred x =
  let rec aux pred y acc =
    match pred.(y) with
    | None -> failwith "Impossible"
    | Some x when x = y -> y::acc
    | Some x -> aux pred x (y::acc)
  in
  aux pred x []
```

On considère qu'on a effectué un DFS dans le graphe résiduel associé à un couplage depuis un sommet et on cherche maintenant à écrire une fonction qui permet de déterminer s'il existe un chemin depuis ce sommet vers un sommet **libre** parmi les buts, c'est-à-dire les sommets **qui ne sont pas des sources**.

Question 13 Écrire une fonction

`cherche_chemin : int option array -> bool array -> int option array -> int -> int list option`

telle que `cherche_chemin pred sources couplage depart` renvoie `Some phi` si `phi` est la liste des sommets visités dans un chemin du graphe résiduel de `depart` à un sommet but **libre** pour couplage et `None` sinon.

Corrigé

```
let cherche_chemin pred sources couplage depart =
  let rec aux i =
    if i = Array.length sources
    then None
    else if not sources.(i) && couplage.(i) = None
          && pred.(i) <> None
    then let phi = remonte pred i in
         if List.hd phi = depart
         then Some phi
         else aux (i+1)
    else aux (i+1)
  in
  aux 0
```

Si le DFS n'a été lancé que depuis `depart`, le test `List.hd phi = depart` est toujours vrai ; il protège le cas où `pred` proviendrait de plusieurs DFS.

Question 14 Écrire une fonction

`chemin_augmentant : int list array -> bool array -> int option array -> int -> int list option`

telle que `chemin_augmentant g sources couplage depart` renvoie `Some phi` où `phi` est la liste des sommets d'un chemin augmentant issu de la source **libre** `depart` (et donc arrivant nécessairement dans un sommet but **libre**), et `None` s'il n'en existe pas.

Corrigé

```
let chemin_augmentant g sources couplage depart =
  let n = Array.length g in
  let pred = Array.make n None in
  pred.(depart) <- Some depart;
  dfs g (arete_residuelle sources couplage) pred depart;
  cherche_chemin pred sources couplage depart
```

Le chemin obtenu est élémentaire car il suit l'arbre du parcours, et il alterne entre arêtes hors du couplage (des sources vers les buts) et arêtes du couplage (des buts vers les sources) : c'est bien un chemin augmentant d'après la partie du cours sur le graphe résiduel.

Question 15 Écrire une fonction

`basculer_chemin : int option array -> int list -> unit`

telle que `basculer_chemin couplage chemin`, où `chemin` est un chemin augmentant pour le couplage, va faire la bascule de toutes les arêtes de `chemin` : celles dans le couplage sont enlevées et celles qui n'y sont pas sont rajoutées.

Indice : seule la parité permet de déterminer celles qui y sont dans la mesure où on a un chemin augmentant.

Corrigé

```
let rec basculer_chemin couplage chemin =
  match chemin with
  | s::t::q -> couplage.(s) <- Some t; couplage.(t) <- Some s; basculer_chemin couplage q
  | [] -> ()
  | [_] -> failwith "Le chemin a forcément un nombre pair de sommets"
```

Il suffit d'ajouter les arêtes de rang impair : chaque sommet du chemin reçoit un nouveau partenaire, ce qui écrase automatiquement les anciennes arêtes du couplage, de rang pair.

Question 16 En déduire une fonction

`couplage_maximal : (int * int) list -> (int * int) list`

telle que `couplage_maximal aretes`, où `aretes` est un graphe biparti donné sous forme d'une liste d'arêtes, renvoie un couplage maximal sous la forme d'une liste d'arêtes.

Pour cela, on va itérer sur chaque sommet source en cherchant un chemin augmentant depuis celui-ci et en le basculant. Quand on aura traité toutes les sources, on est certain qu'il n'y a plus de chemin augmentant (c'est l'objet de la dernière propriété de la partie [Déterminer un chemin augmentant dans un graphe biparti](#) du cours).

Corrigé

```
let couplage_maximal l =
  let g, sources = graphe_biparti l in
  let n = Array.length g in
  let couplage = Array.make n None in
  for i = 0 to n-1 do
    if sources.(i) && couplage.(i) = None
    then match chemin_augmentant g sources couplage i with
         | None -> ()
         | Some ch -> basculer_chemin couplage ch
```

```
done;
couplage_tab_vers_liste sources couplage
```

Il y a au plus $|S|$ recherches de chemin, chacune en $O(|S| + |A|)$, d'où une complexité en $O(|S| (|S| + |A|))$.

Pour tester :

```
assert(couplage_maximal [(0,3);(1,3);(1,4);(2,3);(2,4)]
      = [(1,4);(0,3)]);
assert(couplage_maximal [(0,8);(0,7);(2,6);(2,9);(3,8);(4,8);(4,9);(5,11)]
      = [(5, 11); (4, 9); (3, 8); (2, 6); (0, 7)]);
"OK"
```



Remarque 5.1 Pour tester plus sérieusement `couplage_maximal`, on peut le comparer avec une fonction naïve qui calcule récursivement la taille maximale d'un couplage : pour la première arête $\{a, b\}$ de la liste, soit on ne la prend pas, soit on la prend et on retire toutes les arêtes touchant a ou b . Sur des milliers de petits graphes bipartis aléatoires, on vérifie que le résultat est un couplage, formé d'arêtes du graphe, et de même taille que celui de la fonction naïve.

Algorithme de Ford-Fulkerson en C

6.1 Algorithme de Ford-Fulkerson en C

Ce TP accompagne les parties [Couplage maximal dans un graphe biparti](#) du chapitre Algorithmique avancée des graphes. Il reprend en C le [TP Couplage](#) avant de passer aux flots.

Tous les programmes de ce TP utilisent `stdio.h`, `stdlib.h`, `string.h`, `stdbool.h` et, pour la dernière partie, `limits.h`.

6.1.1 Manipulations de base sur les graphes

On pourra sauter cette partie si on est à l'aise sur ces notions et partir du fichier `tp_couplage_1.c`.

On va considérer le type suivant pour des graphes avec listes chaînées d'adjacence :

```
struct edge_list {
    int dst;
    struct edge_list *next;
};
typedef struct edge_list edge_list;

struct graph {
    int nvertices;
    bool directed;
    edge_list **edge_lists;
};
typedef struct graph graph;
```

Question 1 Écrire une fonction de prototype

```
graph *graph_create(int nvertices, bool directed);
```

qui crée un graphe vide à `nvertices` sommets et dont le caractère orienté est donné par le booléen `directed`.

Corrigé

```
graph *graph_create(int nvertices, bool directed) {
    graph *g = malloc(sizeof(graph));

    g->nvertices = nvertices;
    g->edge_lists = malloc(sizeof(edge_list *) * nvertices);
    for(int i = 0; i < nvertices; i++)
        g->edge_lists[i] = NULL;
    g->directed = directed;

    return g;
}
```

Question 2 Écrire des fonctions de prototypes

```
void edge_list_free(edge_list *l);
void graph_free(graph *g);
```

permettant de libérer la mémoire occupée par un graphe.

Corrigé

```
void edge_list_free(edge_list *l)
{
    while(l != NULL)
    {
        edge_list *t = l->next;
        free(l);
        l = t;
    }
}

void graph_free(graph *g)
{
    for(int i = 0; i < g->nvertices; i++)
        edge_list_free(g->edge_lists[i]);
    free(g->edge_lists);
    free(g);
}
```

Question 3 Écrire une fonction de prototype

```
bool is_connected(graph *g, int x, int y);
```

qui teste s'il existe une arête $x \rightarrow y$.

Corrigé

```
bool is_connected(graph *g, int x, int y)
{
    edge_list *l = g->edge_lists[x];
    while(l != NULL)
    {
        if(l->dst == y)
            return true;
        l = l->next;
    }
    return false;
}
```

Question 4 Écrire une fonction de prototype

```
void add_edge(graph *g, int src, int dst);
```

qui ajoute une arête de src à dst dans le graphe. On ne fera aucune vérification quant au fait que l'arête soit déjà présente.

Corrigé

```
void add_edge(graph *g, int src, int dst)
{
    edge_list *e = malloc(sizeof(edge_list));
    e->dst = dst;
    e->next = g->edge_lists[src];
    g->edge_lists[src] = e;
    if (!g->directed)
    {

```



```

    edge_list *e = malloc(sizeof(edge_list));
    e->dst = src;
    e->next = g->edge_lists[dst];
    g->edge_lists[dst] = e;
}
}

```

Question 5 Écrire une fonction de prototype

```
void add_edges(graph *g, int *vtx_pairs, int npairs);
```

qui prend en entrée un tableau t de 2*npairs entiers et qui rajoute des arêtes de t[2i] à t[2i+1].

Corrigé

```

void add_edges(graph *g, int *vtx_pairs, int npairs)
{
    for(int i = 0; i < npairs*2; i+=2)
    {
        add_edge(g, vtx_pairs[i], vtx_pairs[i+1]);
    }
}

```

Question 6 Écrire une fonction de prototype

```
void graph_print(graph *g);
```

qui affiche le graphe sous un format lisible. On s'en servira pour faire des vérifications.

Corrigé

```

void graph_print(graph *g)
{
    for(int i = 0; i < g->nvertices; i++)
    {
        printf("%d : ", i);
        edge_list *l = g->edge_lists[i];
        while(l != NULL)
        {
            printf(" %d", l->dst);
            l = l->next;
        }
        printf("\n");
    }
}

```

Question 7 Écrire une fonction de prototype

```
int *dfs_init(graph *g, int src);
```

qui renvoie un tableau pred valant -1 pour tous les sommets autres que src et valant src pour src.

Corrigé

```

int *dfs_init(graph *g, int src)
{
    int *pred = malloc(sizeof(int) * g->nvertices);
    memset(pred, -1, sizeof(int) * g->nvertices);
    pred[src] = src;
}

```

```
    return pred;
}
```

memset remplit octet par octet : cela fonctionne pour -1 car tous les octets de l'entier -1 valent 0xFF, mais ce ne serait pas le cas pour une autre valeur comme 1.

Question 8 Écrire une fonction de prototype

```
void dfs(graph *g, int *pred, int x);
```

qui effectue une étape récursive d'un DFS en partant du sommet x et en remplissant le tableau de prédécesseurs $pred$: $pred[y] = x$ si y est découvert par le sommet x .

Corrigé

```
void dfs(graph *g, int *pred, int x)
{
    edge_list *l = g->edge_lists[x];
    while(l != NULL)
    {
        int y = l->dst;
        if (pred[y] < 0)
        {
            pred[y] = x;
            dfs(g, pred, y);
        }
        l = l->next;
    }
}
```

Question 9 Écrire une fonction de prototype

```
int *dfs_launch(graph *g, int src);
```

qui renvoie le tableau $pred$ à l'issue d'un DFS depuis src .

Corrigé

```
int *dfs_launch(graph *g, int src)
{
    int *pred = dfs_init(g, src);
    dfs(g, pred, src);
    return pred;
}
```

6.1.2 Retour sur les couplages de graphe biparti

On pourra sauter cette partie si on est à l'aise sur ces notions et partir du fichier `tp_couplage_2.c`.

On va reprendre les notations du TP précédent. Pour indiquer les deux parties des sommets d'un graphe biparti, on va appeler les premiers sommets des sources et les autres des buts. L'appartenance à un ensemble ou à l'autre est déterminée par un tableau de booléens `sources`.

```
graph *g1 = graph_create(5, false);
int vtx_pairs1[] = { 0, 3, 1, 3, 1, 4, 2, 3, 2, 4 };
bool sources1[] = { true, true, true, false, false };
add_edges(g1, vtx_pairs1, 5);

graph *g2 = graph_create(12, false);
int vtx_pairs2[] = { 0, 8, 0, 7, 2, 6, 2, 9, 3, 8, 4, 8, 4, 9, 5, 11 };
bool sources2[] = { true, false, true, true, true, true,
```

```

    false, false, false, false, false, false };
add_edges(g2, vtx_pairs2, 8);

```

Pour les couplages, on va considérer un tableau d'entiers `matching` où `matching[x] = y` si $\{x, y\}$ est dans le couplage, et `matching[x] = -1` si x est libre.

Question 10 Écrire une fonction de prototype

```
graph *residual_graph(graph *g, bool *sources, int *matching);
```

qui renvoie le graphe résiduel **orienté** associé au couplage du graphe biparti.

Corrigé

```

graph *residual_graph(graph *g, bool *sources, int *matching)
{
    graph *gr = graph_create(g->nvertices, true);
    for(int i = 0; i < g->nvertices; i++)
    {
        edge_list *l = g->edge_lists[i];
        while(l != NULL)
        {
            if((sources[i] && matching[i] != l->dst)
                || (!sources[i] && matching[i] == l->dst))
                add_edge(gr, i, l->dst);
            l = l->next;
        }
    }
    return gr;
}

```

Question 11 Écrire une fonction de prototype

```
void swap_along_path(int *matching, int *pred, int x);
```

qui considère un sommet but libre x atteint par un DFS depuis un sommet source libre dans le graphe résiduel et bascule les arêtes du chemin correspondant en utilisant le tableau `pred` des prédécesseurs.

Corrigé

```

void swap_along_path(int *matching, int *pred, int x)
{
    bool swap = true;
    while(pred[x] != x)
    {
        if(swap)
        {
            matching[x] = pred[x];
            matching[pred[x]] = x;
        }
        swap = !swap;
        x = pred[x];
    }
}

```

Question 12 Écrire une fonction de prototype

```
bool augment(graph *g, bool *sources, int *matching, int *pred);
```

qui considère un graphe biparti, un couplage et un tableau `pred` issu d'un DFS depuis une source libre et renvoie `true` après avoir basculé un chemin augmentant s'il en existe un, et `false` sinon.

Corrigé

```
bool augment(graph *g, bool *sources, int *matching, int *pred)
{
    for(int i = 0; i < g->nvertices; i++)
    {
        if(!sources[i] && matching[i] < 0 && pred[i] >= 0)
        {
            swap_along_path(matching, pred, i);
            return true;
        }
    }
    return false;
}
```

Question 13 Écrire une fonction de prototype

```
int *maximal_matching(graph *g, bool *sources);
```

qui renvoie un tableau matching décrivant un couplage maximal.

Corrigé

```
int *maximal_matching(graph *g, bool *sources)
{
    int *matching = malloc(sizeof(int) * g->nvertices);
    memset(matching, -1, sizeof(int) * g->nvertices);
    for(int i = 0; i < g->nvertices; i++)
    {
        if(sources[i] && matching[i] < 0)
        {
            graph *gr = residual_graph(g, sources, matching);
            int *pred = dfs_launch(gr, i);
            augment(g, sources, matching, pred);
            free(pred);
            graph_free(gr);
        }
    }
    return matching;
}
```

Comme dans le TP en OCaml, il suffit de chercher une seule fois un chemin augmentant depuis chaque source (voir le cours).

6.1.3 Ford-Fulkerson

Note si vous vous sentez à l'aise, il peut être intéressant de directement essayer d'implémenter Ford-Fulkerson. On fera attention à préserver la complexité, ce qui implique de bien réfléchir à l'organisation de votre code. C'est un exercice délicat mais très instructif. Cette partie vous propose un choix d'implémentation parmi d'autres.

Pour pouvoir travailler sur l'algorithme de Ford-Fulkerson, il va être nécessaire de reporter des changements de flot le long d'un chemin dans le graphe résiduel vers le graphe initial. On va donc avoir besoin de considérer les arêtes dans un chemin et pas seulement les sommets donnés par la relation de prédécesseurs **et** on va avoir besoin d'accéder directement à une arête du graphe initial depuis une des deux arêtes créées dans le graphe résiduel.

C'est pour cela qu'on enrichit le type précédent :

```
struct edge_list {
    int src;
```

```

    int dst;
    int flow; // le flot sur l'arete
    int capacity; // la capacite de l'arete
    struct edge_list *ref; // NULL ou l'arete associee
    struct edge_list *next;
};
typedef struct edge_list edge_list;

struct graph {
    int s; /* graphe s-->*-->t */
    int t;
    int nvertices;
    edge_list **edge_lists;
};
typedef struct graph graph;

```

Question 14 Reprendre la fonction `graph_create` pour tenir compte de `s` et `t`.

```
graph *graph_create(int nvertices, int s, int t);
```

Ici, le graphe est forcément orienté.

Corrigé

```

graph *graph_create(int nvertices, int s, int t) {
    graph *g = malloc(sizeof(graph));

    g->nvertices = nvertices;
    g->s = s;
    g->t = t;
    g->edge_lists = malloc(sizeof(edge_list *) * nvertices);
    for(int i = 0; i < nvertices; i++)
        g->edge_lists[i] = NULL;

    return g;
}

```

Les fonctions `edge_list_free` et `graph_free` sont inchangées.

Question 15 Reprendre la fonction `add_edge` pour tenir compte de cela, cette fonction affectera un flot nul et une référence NULL et renverra l'arête créée :

```
edge_list *add_edge(graph *g, int src, int dst, int capacity);
```

Corrigé

```

edge_list *add_edge(graph *g, int src, int dst, int capacity)
{
    edge_list *e = malloc(sizeof(edge_list));
    e->src = src;
    e->dst = dst;
    e->next = g->edge_lists[src];
    g->edge_lists[src] = e;
    e->capacity = capacity;
    e->flow = 0;
    e->ref = NULL;

    return e;
}

```

Question 16 Reprendre la fonction `graph_print` pour afficher le flot.

```
void graph_print(graph *g);
```

Corrigé

```
void graph_print(graph *g)
{
    for(int i = 0; i < g->nvertices; i++)
    {
        edge_list *l = g->edge_lists[i];
        while(l != NULL)
        {
            printf("%d ---%d/%d---> %d\n", i, l->flow, l->capacity, l->dst);
            l = l->next;
        }
    }
}
```

Question 17 Reprendre les fonctions de parcours pour qu'elles correspondent aux prototypes suivants :

```
edge_list **dfs_init(graph *g);
void dfs(graph *g, edge_list **pred, int x);
edge_list **dfs_launch(graph *g);
```

Tous les DFS démarreront de $g \rightarrow s$.

Ici, le tableau des prédécesseurs indique l'arête empruntée pour remonter. Comme la source $g \rightarrow s$ du DFS aura une valeur NULL dans ce tableau, il est nécessaire de la tester explicitement pour ne pas la considérer à tort comme non visitée.

Corrigé

```
edge_list **dfs_init(graph *g)
{
    edge_list **pred = malloc(sizeof(edge_list *) * g->nvertices);
    memset(pred, 0, sizeof(edge_list *) * g->nvertices);
    return pred;
}

void dfs(graph *g, edge_list **pred, int x)
{
    edge_list *l = g->edge_lists[x];
    while(l != NULL)
    {
        int y = l->dst;
        if (pred[y] == NULL && y != g->s)
        {
            pred[y] = l;
            dfs(g, pred, y);
        }
        l = l->next;
    }
}

edge_list **dfs_launch(graph *g)
{
    edge_list **pred = dfs_init(g);
    dfs(g, pred, g->s);
    return pred;
}
```

Question 18 Écrire une fonction de prototype

```
graph *residual_graph(graph *g);
```

qui renvoie le graphe résiduel associé au flot présent sur les arêtes de g .

On fera en sorte que les arêtes du graphe résiduel fassent référence aux arêtes de g dont elles proviennent. On va utiliser le champ `capacity` pour la capacité résiduelle et le champ `flow` aura la valeur 1 si on est dans l'orientation de g et -1 si on est dans l'orientation inverse (retour du flot).

Corrigé

```
graph *residual_graph(graph *g)
{
    graph *gr = graph_create(g->nvertices, g->s, g->t);
    for(int i = 0; i < g->nvertices; i++)
    {
        edge_list *l = g->edge_lists[i];
        while(l != NULL)
        {
            if (l->flow > 0)
            {
                edge_list *e = add_edge(gr, l->dst, i, l->flow);
                e->ref = l;
                e->flow = -1;
            }
            if (l->flow < l->capacity)
            {
                edge_list *e = add_edge(gr, i, l->dst, l->capacity-l->flow);
                e->ref = l;
                e->flow = 1;
            }
            l = l->next;
        }
    }
    return gr;
}
```

Question 19 Écrire une fonction de prototype

```
void adjust_flow_along_path(edge_list **pred, int t);
```

qui, étant donné un tableau `pred` issu d'un DFS depuis la source du graphe de flot et le sommet t destination **visité**, va remonter le chemin augmentant pour ajuster le flot dans le graphe initial.

Il faudra deux parcours ici pour obtenir le minimum puis pour ajuster le flot.

Corrigé

```
void adjust_flow_along_path(edge_list **pred, int t)
{
    int min_flow = INT_MAX;
    int y = t;
    while(pred[y] != NULL)
    {
        if(min_flow > pred[y]->capacity)
            min_flow = pred[y]->capacity;
        y = pred[y]->src;
    }
    y = t;
    while(pred[y] != NULL)
    {
        pred[y]->ref->flow += pred[y]->flow * min_flow;
        y = pred[y]->src;
    }
}
```

La remontée s'arrête en $g->s$, seul sommet visité dont le prédécesseur vaut `NULL`.

Question 20 Écrire une fonction de prototype

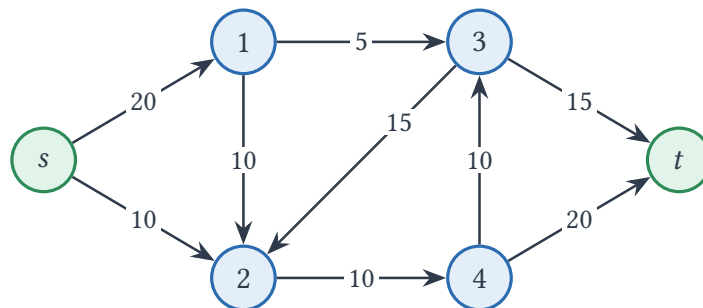
```
void maximal_flow(graph *g);
```

qui calcule un flot maximal depuis un flot nul.

Corrigé

```
void maximal_flow(graph *g)
{
    bool has_augmented = true;
    while(has_augmented)
    {
        has_augmented = false;
        graph *gr = residual_graph(g);
        edge_list **pred = dfs_launch(gr);
        if (pred[g->t] != NULL)
        {
            has_augmented = true;
            adjust_flow_along_path(pred, g->t);
        }
        free(pred);
        graph_free(gr);
    }
}
```

Chaque itération coûte $O(|S| + |A|)$ et augmente la valeur du flot d'au moins 1 : on retrouve la complexité $O(f^* (|S| + |A|))$ du cours.

Question 21 Testez votre fonction sur le réseau suivant, où la source s est le sommet 0 et le puits t le sommet 5 (les étiquettes sont les capacités) :


```
int main(void)
{
    graph *g = graph_create(6, 0, 5);
    add_edge(g, 0, 1, 20);
    add_edge(g, 0, 2, 10);
    add_edge(g, 1, 2, 10);
    add_edge(g, 1, 3, 5);
    add_edge(g, 3, 2, 15);
    add_edge(g, 2, 4, 10);
    add_edge(g, 4, 3, 10);
    add_edge(g, 3, 5, 15);
    add_edge(g, 4, 5, 20);
    maximal_flow(g);
    graph_print(g);

    graph_free(g);
}
```

Corrigé

On obtient un flot de valeur 15, par exemple celui de la figure du cours :

```
0 ---0/10---> 2
0 ---15/20----> 1
1 ---5/5---> 3
1 ---10/10---> 2
2 ---10/10---> 4
3 ---10/15---> 5
3 ---0/15---> 2
4 ---5/20---> 5
4 ---5/10---> 3
```

Le flot obtenu dépend de l'ordre des listes d'adjacence, mais sa valeur est toujours 15, la capacité de la coupe $\{0, 1, 2\}$.

Question 22 Assurez-vous de n'avoir aucune fuite mémoire.

Corrigé

On peut compiler avec `gcc -Wall -Wextra -fsanitize=address -g` : à la fin du programme, l'outil signale les blocs alloués non libérés. On peut aussi utiliser `valgrind --leak-check=full ./a.out` sous Linux.

Question 23 Appliquez cela au cas d'un graphe biparti pour retrouver la méthode vue précédemment.

Corrigé

On construit le réseau du cours : les sommets $0, \dots, n-1$ du graphe biparti, une source $s = n$ et un puits $t = n+1$, des arêtes de capacité 1 de s vers chaque source, de chaque but vers t , et de la source vers le but pour chaque arête du graphe. La valeur d'un flot maximal est la taille d'un couplage maximal, et les arêtes entre sources et buts portant un flot 1 forment ce couplage.

```
int max_matching_size(int nvertices, bool *sources, int *vtx_pairs, int npairs)
{
    int s = nvertices, t = nvertices + 1;
    graph *g = graph_create(nvertices + 2, s, t);
    for(int i = 0; i < nvertices; i++)
    {
        if (sources[i])
            add_edge(g, s, i, 1);
        else
            add_edge(g, i, t, 1);
    }
    for(int i = 0; i < npairs; i++)
    {
        int x = vtx_pairs[2*i];
        int y = vtx_pairs[2*i+1];
        if (sources[x])
            add_edge(g, x, y, 1);
        else
            add_edge(g, y, x, 1);
    }
    maximal_flow(g);
    int value = 0;
    for(edge_list *l = g->edge_lists[s]; l != NULL; l = l->next)
        value += l->flow;
    graph_free(g);
    return value;
}
```

Sur le graphe g_2 précédent, on obtient 5. Le graphe résiduel de ce réseau est exactement celui de la partie précédente, complété par s et t : chaque itération de Ford-Fulkerson bascule un chemin augmentant du couplage.



Remarque 6.1 Les corrigés de ce TP ont été testés, avec `-fsanitize=address,undefined`, en comparant sur des milliers de petits graphes aléatoires la valeur du flot obtenu avec la capacité minimale d'une coupe calculée en énumérant toutes les coupes, et la taille des couplages obtenus avec celle d'un couplage maximal calculé par énumération.

Algorithmes d'approximation, algorithmes probabilistes

DANS CE CHAPITRE

- 7.1 Algorithmes d'approximation
- 7.2 Algorithmes probabilistes
- 7.3 Travaux dirigés et pratiques

7.1 Algorithmes d'approximation

Source principale de cette partie : *Approximation Algorithms* de Vijay Vazirani.

7.1.1 Définition

Dans le contexte d'un problème d'optimisation fixé (voir le chapitre [Calculabilité](#) pour la définition), on va noter $OPT(e)$ la valeur de la solution optimale pour l'entrée e . Dans tout ce chapitre, on s'intéresse au cas où le problème de décision à seuil associé est NP-complet. Il est donc illusoire de chercher à calculer $OPT(e)$ pour des entrées de grande taille. On va plutôt chercher à définir un algorithme calculant non pas $OPT(e)$ mais une approximation.

Définition 7.1

On considère un problème de minimisation défini par :

- les instances I ;
- les sorties S ;
- les sorties valides pour une instance, définies par $v : I \rightarrow \mathcal{P}(S) \setminus \{\emptyset\}$;
- et une fonction de coût $c : S \rightarrow \mathbb{R}_+$.

On note $OPT(i) = \min_{s \in v(i)} c(s)$ le coût minimal pour l'instance i , et on suppose $OPT(i) > 0$.

Étant donné un algorithme qui calcule une fonction $\phi : I \rightarrow S$ telle que $\forall i \in I, \phi(i) \in v(i)$, une fonction $\rho : \mathbb{N} \rightarrow \mathbb{R}$ telle que

$$\forall n \in \mathbb{N}, \rho(n) \geq \max_{i \in I, |i|=n} \frac{c(\phi(i))}{OPT(i)}$$

est appelée un **facteur d'approximation** de l'algorithme.

Quand ρ est une fonction constante λ , on dit que l'algorithme est une λ -approximation.

Remarque 7.2

- Le fait d'avoir $\rho(n) \geq$ et pas $\rho(n) =$ permet de se contenter d'une majoration : on n'a pas besoin de calculer exactement le pire cas.

- Pour un problème de minimisation, on a forcément $c(\phi(i)) \geq OPT(i)$ or si $|i| = n$, on a

$$\frac{c(\phi(i))}{OPT(i)} \leq \rho(n) \Leftrightarrow c(\phi(i)) \leq \rho(n)OPT(i)$$

Et donc l'encadrement :

$$OPT(i) \leq c(\phi(i)) \leq \rho(n)OPT(i)$$

Ainsi, $\rho(n)$ assure un contrôle sur l'approximation qui garantit qu'on est à moins de cette proportion de l'optimal.

- Il n'y a aucune raison qu'un problème NP-complet possède une λ -approximation (voir le cas du TSP général ci-dessous).
- On définit naturellement la même notion pour les problèmes de maximisation :

$$\forall n \in \mathbb{N}, \rho(n) \geq \max_{i \in I, |i| = n} \frac{OPT(i)}{c(\phi(i))}$$

on a alors $c(\phi(i)) \geq \frac{1}{\rho(n)}OPT(i)$.

- Ainsi, une 2-approximation est au pire :
 - 2 fois plus grande pour un problème de minimisation ($\times 2$) ;
 - 2 fois plus petite pour un problème de maximisation ($\times 1/2$).



7.1.2 2-approximation de **MinVertexCover**

On reprend le problème **MinVertexCover** du chapitre **Calculabilité** : étant donné un graphe non orienté $G = (S, A)$, on cherche une couverture par sommets $C \subset S$, c'est-à-dire telle que toute arête a au moins une extrémité dans C , de cardinal minimal. Le problème à seuil associé, **VertexCover**, est NP-complet.

Une idée gloutonne naturelle consiste à prendre une arête non couverte et à mettre ses deux extrémités dans la couverture. On ne sait pas laquelle des deux est utile, alors on prend les deux : c'est ce choix qui coûte un facteur 2.

Algorithme 7.3

- $C \leftarrow \emptyset$
- Pour chaque arête $\{x, y\} \in A$:
 - si $x \notin C$ et $y \notin C$, alors $C \leftarrow C \cup \{x, y\}$.
- Renvoyer C .

On note M l'ensemble des arêtes $\{x, y\}$ qui ont provoqué un ajout dans C .

Théorème 7.4

Cet algorithme est une 2-approximation de **MinVertexCover**, de complexité $O(|S| + |A|)$ avec des listes d'adjacence.

Preuve

- C est une couverture : si une arête $\{x, y\}$ n'avait aucune extrémité dans C à la fin, elle n'en avait pas non plus au moment où on l'a examinée (on ne retire jamais de sommet de C), donc on aurait ajouté x et y à ce moment-là.
- M est un couplage : quand on ajoute $\{x, y\}$ à M , ni x ni y ne sont dans C , donc ils ne sont extrémités d'aucune arête déjà dans M . Ainsi $|C| = 2|M|$.
- Soit C^* une couverture optimale. Elle contient au moins une extrémité de chaque arête de M , et ces extrémités sont distinctes car les arêtes de M sont deux à deux disjointes. Donc $OPT(G) = |C^*| \geq |M|$.

On en déduit $|C| = 2|M| \leq 2OPT(G)$. Si G n'a pas d'arête, $C = \emptyset$ est optimale. Chaque arête est examinée un nombre constant de fois avec des opérations en $O(1)$ d'où la complexité. $\square$

Remarque 7.5 Le facteur 2 est atteint : pour un graphe réduit à une arête, l'algorithme renvoie les deux sommets alors qu'un seul suffit. Plus généralement, sur un graphe biparti complet $K_{n,n}$, la couverture renvoyée contient les $2n$ sommets alors que $OPT = n$.

La preuve utilise une technique très générale : on ne connaît pas OPT , mais on dispose d'un **minorant** de OPT (ici la taille d'un couplage) qu'on sait relier à la valeur de la solution calculée.

Avec la représentation par listes d'adjacence du chapitre sur les graphes, chaque arête est vue deux fois, ce qui ne change rien :

OCaml

```
let couverture_2approx (g : int list array) : int list =
  let n = Array.length g in
  let couvert = Array.make n false in
  let c = ref [] in
  for x = 0 to n - 1 do
    List.iter (fun y ->
      if not couvert.(x) && not couvert.(y) then begin
        couvert.(x) <- true;
        couvert.(y) <- true;
        c := x :: y :: !c
      end)
      g.(x)
  done;
  !c
```

C

```
struct graphe {
  int n;          // nombre de sommets
  int *deg;       // deg[x] = nombre de voisins de x
  int **voisins;  // voisins[x][0], ..., voisins[x][deg[x] - 1]
};
typedef struct graphe graphe;

/* renvoie un tableau de booléens : dans_c[x] indique si x est dans la couverture */
bool *couverture_2approx(graphe *g)
{
  bool *dans_c = malloc(sizeof(bool) * g->n);
  for (int x = 0; x < g->n; x++)
    dans_c[x] = false;
  for (int x = 0; x < g->n; x++) {
    for (int k = 0; k < g->deg[x]; k++) {
      int y = g->voisins[x][k];
      if (!dans_c[x] && !dans_c[y]) {
        dans_c[x] = true;
        dans_c[y] = true;
      }
    }
  }
  return dans_c;
}
```

Le TP associé en propose une version sur une liste d'arêtes.

7.1.3 2-approximation de MetricTSP

Problème 7.6 — : MetricTSP

- Entrée : un graphe complet non orienté $G = (S, A)$ avec $|S| \geq 2$, et une distance $d : S^2 \rightarrow \mathbb{R}_+$ symétrique vérifiant l'inégalité triangulaire $\forall x, y, z \in S, d(x, y) \leq d(x, z) + d(z, y)$.

- Sortie : un cycle hamiltonien de G (passant une et une seule fois par chaque sommet) de longueur minimale.

Ce problème est NP-difficile. On considère l'algorithme suivant :

Algorithme 7.7

- On construit un arbre couvrant de poids minimal T de G (par exemple avec l'algorithme de Kruskal, voir le chapitre [Algorithmique des graphes avancée](#)).
- On choisit une racine r et on renvoie le cycle qui visite les sommets dans l'ordre préfixe d'un parcours en profondeur de T , puis revient en r .

Théorème 7.8

Cet algorithme est une 2-approximation de MetricTSP .

Preuve Notons $d(T)$ le poids de l'arbre et H^* un cycle hamiltonien optimal.

- $d(T) \leq \text{OPT}$: en retirant une arête de H^* , on obtient un chemin hamiltonien, qui est un arbre couvrant de G , de poids $\leq d(H^*) = \text{OPT}$ car les distances sont positives. Comme T est de poids minimal, $d(T) \leq \text{OPT}$.
- Le parcours en profondeur complet de T , qui note un sommet à chaque passage (en descendant et en remontant), emprunte chaque arête de T exactement deux fois : c'est un circuit fermé P passant par tous les sommets, de longueur $2d(T)$.
- Le cycle renvoyé H s'obtient à partir de P en sautant les sommets déjà visités : on remplace un morceau $x, z_1, \dots, z_k, y$ de P par l'arête $\{x, y\}$. Par inégalité triangulaire (appliquée k fois), $d(x, y) \leq d(x, z_1) + \dots + d(z_k, y)$, donc $d(H) \leq d(P) = 2d(T)$.

Ainsi $d(H) \leq 2d(T) \leq 2\text{OPT}$.

□

Remarque 7.9

- L'hypothèse métrique est indispensable. Sans elle, s'il existait une λ -approximation polynomiale du TSP général, on pourrait décider en temps polynomial si un graphe $G = (S, A)$ possède un cycle hamiltonien (problème NP-complet) : on complète G en donnant le poids 1 à ses arêtes et le poids $\lambda |S| + 1$ aux autres. S'il existe un cycle hamiltonien dans G , $\text{OPT} = |S|$ et l'approximation renvoie un cycle de longueur $\leq \lambda |S|$, qui ne peut utiliser que des arêtes de G ; sinon, tout cycle utilise une arête de poids $\lambda |S| + 1$. Il suffit donc de comparer la longueur renvoyée à $\lambda |S|$. Ainsi, sauf si $P = NP$, le TSP général n'a pas de λ -approximation polynomiale.
- L'algorithme de Christofides (hors programme) améliore ce facteur en $\frac{3}{2}$ pour MetricTSP .

Ce problème est mis en œuvre dans le [TP associé](#).



7.1.4 Sac à dos

Problème 7.10 — : Knapsack

- Entrée : n objets numérotés de 1 à n , l'objet i ayant un poids $w_i \in \mathbb{N}^*$ et une valeur $v_i \in \mathbb{N}^*$, et un poids maximal $W \in \mathbb{N}$.
- Sortie : $S \subset \llbracket 1, n \rrbracket$ tel que $w(S) = \sum_{i \in S} w_i \leq W$ et de valeur $v(S) = \sum_{i \in S} v_i$ maximale.

Le [TD sur le sac à dos](#) montre que ce problème est NP-complet, que sa variante fractionnaire est résolue optimalement par un algorithme glouton, et construit un schéma d'approximation : pour tout $\epsilon > 0$, un algorithme de complexité $O(n^3/\epsilon)$ qui renvoie une solution de valeur $\geq (1 - \epsilon)\text{OPT}$.

On présente ici une approximation plus simple. Quitte à les éliminer, on suppose que chaque objet rentre seul dans le sac : $w_i \leq W$.

Le glouton par efficacité v_i/w_i décroissante seul peut être arbitrairement mauvais : avec $W = M$, un objet de poids 1 et de valeur 2 et un objet de poids M et de valeur M , il prend le premier objet, ne peut plus prendre le second, et renvoie 2 au lieu de M . On corrige cela en comparant au meilleur objet pris seul.

Algorithme 7.11

- Trier les objets par efficacité v_i/w_i décroissante et les ajouter dans cet ordre dès qu'ils rentrent dans le sac : on obtient S_g .
- Soit i_0 un objet de valeur maximale.
- Renvoyer la meilleure des deux solutions S_g et $\{i_0\}$.

Théorème 7.12

Cet algorithme est une 2-approximation de Knapsack, de complexité $O(n \log n)$.

Preuve On numérote les objets par efficacité décroissante. Si tous les objets rentrent ensemble, S_g est l'ensemble des objets, qui est optimal. Sinon, soit j le premier objet tel que $w_1 + \dots + w_j > W$. Les objets $1, \dots, j-1$ ont tous été pris par le glouton, donc $v(S_g) \geq v_1 + \dots + v_{j-1}$.

Soit S^* optimal. On a, en notant $e = v_j/w_j$:

$$v(S^*) - \sum_{i=1}^j v_i = \sum_{i \in S^*, i > j} v_i - \sum_{i \leq j, i \notin S^*} v_i \leq e \left(\sum_{i \in S^*, i > j} w_i - \sum_{i \leq j, i \notin S^*} w_i \right) = e \left(w(S^*) - \sum_{i=1}^j w_i \right) < 0$$

car $v_i \leq ew_i$ pour $i > j$, $v_i \geq ew_i$ pour $i \leq j$, et $w(S^*) \leq W < \sum_{i=1}^j w_i$. Ainsi

$$OPT = v(S^*) < (v_1 + \dots + v_{j-1}) + v_j \leq v(S_g) + v_{i_0} \leq 2 \max(v(S_g), v_{i_0})$$

La complexité est dominée par le tri.

□

L'implémentation compare les efficacités sans flottants : $v_i/w_i < v_j/w_j \Leftrightarrow v_i w_j < v_j w_i$.

OCaml

```
let sac_glouton (w : int array) (v : int array) (cap : int) : int list =
  let n = Array.length w in
  let ordre = Array.init n (fun i -> i) in
  (* tri par efficacité v.(i) / w.(i) décroissante *)
  Array.sort (fun i j -> compare (v.(j) * w.(i)) (v.(i) * w.(j))) ordre;
  let reste = ref cap in
  let pris = ref [] in
  let valeur = ref 0 in
  Array.iter (fun i ->
    if w.(i) <= !reste then begin
      reste := !reste - w.(i);
      valeur := !valeur + v.(i);
      pris := i :: !pris
    end)
    ordre;
  (* meilleur objet seul *)
  let meilleur = ref (-1) in
  for i = 0 to n - 1 do
    if w.(i) <= cap && (!meilleur = -1 || v.(i) > v.(!meilleur)) then
      meilleur := i
  done;
  if !meilleur <> -1 && v.(!meilleur) > !valeur then [!meilleur] else !pris
```

Remarque 7.13 La boucle gloutonne ne s'arrête pas au premier objet qui ne rentre pas : cela ne peut qu'améliorer la solution, et la preuve reste valable. La fonction ignore d'elle-même les objets trop lourds.

7.2 Algorithmes probabilistes

Source principale de cette partie : *Randomized Algorithms* de Rajeev Motwani et Prabhakar Raghavan.

7.2.1 Las Vegas et Monte-Carlo

On va considérer dans ce paragraphe des algorithmes ayant accès à une forme parfaite de hasard. On ne soulève pas de difficulté sur cette définition, et ce hasard pourra, selon les cas, se matérialiser par le fait de pouvoir choisir uniformément et aléatoirement dans un ensemble de valeurs :

- le cas le plus simple est celui du bit aléatoire : une valeur aléatoire 0 ou 1. Ici, il est crucial que tous les tirages soient indépendants, cela permet naturellement de construire des valeurs mémoires aléatoires sur n bits : entiers, flottants...
- un cas usuel dans les algorithmes est celui du choix d'un élément dans un tableau ou dans une liste.

On considère le plus souvent que le tirage s'effectue en temps et en espace constant. Cela suffit pour la plupart des analyses.

En pratique, les ordinateurs utilisent des générateurs pseudo-aléatoires : des suites déterministes qui semblent aléatoires (voir le TP). En OCaml, `Random.int n` renvoie un entier uniforme dans $\llbracket 0, n-1 \rrbracket$; en C, on utilise `rand()`.

Pour une entrée e fixée, l'exécution d'un tel algorithme dépend des tirages effectués : son résultat et sa durée $T(e)$ sont donc des **variables aléatoires**. On distingue deux grandes familles d'algorithmes probabilistes selon que c'est la durée ou la correction qui est aléatoire.

Définition 7.14

Un algorithme probabiliste résolvant un problème est dit :

- de **Las Vegas** lorsque, pour toute entrée e , il termine avec probabilité 1 et que le résultat renvoyé est **toujours** correct. Seule sa durée $T(e)$ est aléatoire : on mesure alors sa complexité par l'**espérance** $E(T(e))$, et on parle de complexité en moyenne dans le pire cas pour $\max_{|e|=n} E(T(e))$.
- de **Monte-Carlo** lorsqu'il existe une fonction t telle que, pour toute entrée e , il termine toujours en au plus $t(|e|)$ étapes, mais que le résultat peut être faux. On impose alors une majoration de la **probabilité d'erreur** : il existe $\epsilon < 1$ tel que pour toute entrée e , la probabilité que le résultat soit incorrect est au plus ϵ .

Pour un problème de décision, un algorithme de Monte-Carlo est dit à **erreur unilatérale** lorsqu'une des deux réponses est toujours correcte (par exemple : s'il répond « non », la bonne réponse est « non » ; s'il répond « oui », il peut se tromper avec probabilité $\leq \epsilon$), et à **erreur bilatérale** sinon, avec alors $\epsilon < \frac{1}{2}$.

Remarque 7.15 Attention à ne pas confondre l'espérance d'un algorithme de Las Vegas avec la complexité en moyenne d'un algorithme déterministe : ici, l'espérance porte sur les tirages de l'algorithme, l'entrée étant fixée et pouvant être la pire possible. Aucune hypothèse n'est faite sur la répartition des entrées.

Pour mémoriser : à Las Vegas, on joue jusqu'à gagner, mais on ne sait pas combien de temps cela va prendre ; à Monte-Carlo, on joue un temps fixé, sans garantie de gagner.

On peut passer d'une famille à l'autre, et réduire la probabilité d'erreur en répétant l'algorithme.

Propriété 7.16

1. **Amplification** : si un algorithme de Monte-Carlo à erreur unilatérale (réponse « non » toujours correcte) a une probabilité d'erreur $\leq \epsilon$, l'algorithme qui l'exécute k fois de manière indépendante et répond « oui » si et seulement si toutes les exécutions ont répondu « oui » a une probabilité d'erreur $\leq \epsilon^k$.

2. **De Monte-Carlo à Las Vegas** : si un algorithme de Monte-Carlo renvoie une réponse correcte avec probabilité $\geq p > 0$ et qu'on sait **vérifier** une réponse, l'algorithme qui le répète jusqu'à obtenir une réponse vérifiée est de Las Vegas, et le nombre moyen de répétitions est au plus $\frac{1}{p}$.
3. **De Las Vegas à Monte-Carlo** : si un algorithme de Las Vegas vérifie $E(T(e)) \leq t(|e|)$, l'algorithme qui l'interrompt après $2t(|e|)$ étapes en répondant « échec » termine en temps borné et échoue avec probabilité $\leq \frac{1}{2}$.

Preuve

1. Si la bonne réponse est « non », chaque exécution répond « non » donc l'algorithme répété aussi. Si la bonne réponse est « oui », il se trompe si et seulement si les k exécutions indépendantes se trompent, ce qui arrive avec probabilité $\leq \epsilon^k$.
2. Comme chaque réponse renvoyée est vérifiée, elle est correcte. Notons $q \in [0, 1 - p]$ la probabilité qu'une exécution échoue et N le nombre d'exécutions. Par indépendance, $P(N = k) = q^{k-1}(1 - q)$: N suit une loi géométrique. En particulier $P(N > k) = q^k \rightarrow 0$ donc l'algorithme termine avec probabilité 1, et

$$E(N) = \sum_{k=1}^{+\infty} k(1 - q)q^{k-1} = \frac{1 - q}{(1 - q)^2} = \frac{1}{1 - q} \leq \frac{1}{p}$$

3. C'est l'inégalité de Markov : pour une variable aléatoire X à valeurs dans $\mathbb{N}$ et $a > 0$, $E(X) = \sum_k kP(X = k) \geq \sum_{k \geq a} aP(X = k) = aP(X \geq a)$. Avec $X = T(e)$ et $a = 2t(|e|)$, on obtient $P(T(e) \geq 2t(|e|)) \leq \frac{E(T(e))}{2t(|e|)} \leq \frac{1}{2}$.

□

Remarque 7.17

- Dans le point 3, en relançant k fois l'algorithme interrompu, on obtient une probabilité d'échec $\leq 2^{-k}$ en temps $2kt(|e|)$.
- Pour une erreur bilatérale $\epsilon < \frac{1}{2}$, on amplifie en répondant la réponse majoritaire de $2k + 1$ exécutions indépendantes : la probabilité d'erreur décroît alors exponentiellement avec k (admis ; cela découle d'inégalités de concentration hors programme).
- L'exemple classique d'algorithme de Monte-Carlo à erreur unilatérale est l'algorithme de Freivalds qui vérifie si $AB = C$ pour des matrices A, B, C en temps $O(n^2)$ en testant $A(Bx) = Cx$ pour un vecteur x aléatoire.

7.2.2 Problème des N reines pour N grand

On cherche à placer N reines sur un échiquier $N \times N$ sans que deux reines soient sur la même ligne, la même colonne ou la même diagonale. Le retour sur trace résout ce problème, mais explore beaucoup de positions quand N grandit. On sait par ailleurs qu'il existe des solutions pour tout $N \geq 4$ (admis), et qu'il n'y en a pas pour $N \in \{2, 3\}$.

Une idée probabiliste très simple : on place une reine par ligne, dans une colonne choisie uniformément parmi celles qui ne sont pas attaquées par les reines précédentes. Si aucune colonne n'est disponible, on recommence tout.

OCaml

```
(* tente de placer une reine par ligne dans une colonne libre tirée
   au hasard ; renvoie None en cas d'impasse *)
let essai_reines n =
  let col = Array.make n (-1) in
  let compatible l c =
    let ok = ref true in
    for l' = 0 to l - 1 do
      if col.(l') = c || abs (col.(l') - c) = l - l' then ok := false
    end
  in
```

```

done;
!ok
in
let rec place l =
  if l = n then Some col
  else begin
    let possibles = List.filter (compatible l) (List.init n (fun c -> c)) in
    match possibles with
    | [] -> None
    | _ ->
      col.(l) <- List.nth possibles (Random.int (List.length possibles));
      place (l + 1)
    end
  in
  place 0

let rec reines_las_vegas n =
  match essai_reines n with
  | Some col -> col
  | None -> reines_las_vegas n

```

essai_reines est un algorithme de Monte-Carlo : il termine en $O(N^3)$ avec cette implémentation naïve, mais peut échouer. Comme une solution renvoyée est toujours correcte, reines_las_vegas est un algorithme de Las Vegas pour $N = 1$ ou $N \geq 4$: la probabilité p_N qu'un essai réussisse est strictement positive, puisqu'il existe une solution et que chacun de ses choix a une probabilité non nulle d'être tiré. D'après la propriété précédente, le nombre moyen d'essais est $\frac{1}{p_N}$. Pour $N \in \{2, 3\}$, la fonction ne termine pas.

Des mesures sur 20 000 essais donnent :

N	4	5	8	10	20	30
p_N estimée	0,50	0,76	0,13	0,06	0,022	0,012

Remarque 7.18 Pour $N = 8$, il faut donc environ 8 essais en moyenne. La probabilité de succès diminue avec N : en pratique, on combine les deux approches en plaçant aléatoirement les reines des premières lignes puis en complétant par retour sur trace, ce qui est beaucoup plus efficace que chacune des deux méthodes seule pour N grand.

7.2.3 Algorithme de Karger

Définition 7.19

Soit $G = (S, A)$ un graphe non orienté avec $|S| \geq 2$, on dit que $\Gamma \subset A$ est une coupure de G lorsque le graphe $(S, A \setminus \Gamma)$ n'est pas connexe.

Remarque 7.20 Cette définition n'a donc de pertinence que pour un graphe connexe. Voire, pour un graphe **très connexe**, c'est-à-dire un graphe pour lequel il faut supprimer beaucoup d'arêtes pour le déconnecter.

Pour une partie $X \neq \emptyset, S$, l'ensemble des arêtes reliant X à $S \setminus X$ est une coupure. Réciproquement, si Γ est une coupure et X une composante connexe de $(S, A \setminus \Gamma)$, les arêtes reliant X à $S \setminus X$ sont toutes dans Γ : une coupure de cardinal minimal est donc de cette forme.

Lemme 7.21

Si la plus petite coupure, en cardinal, de G a k arêtes, alors

$$|A| \geq k \frac{|S|}{2}$$

Preuve Si G admet un sommet de degré $< k$, alors il suffit de supprimer toutes ses arêtes incidentes pour l'isoler. On obtient ainsi une coupure de taille strictement plus petite que k , ce qui est impossible par hypothèse. Tous les sommets sont donc de degré $\geq k$.

On sait que $\sum_{x \in S} d(x) = 2 |A|$ et on a donc $2 |A| \geq |S| k$.

□

Cette remarque, qui ici nous permet de constater qu'il faut un nombre conséquent d'arêtes pour ne pas avoir de coupures trop petites, sera cruciale dans la suite. Elle reste valable, avec la même preuve, pour les multigraphes définis ci-dessous.

On cherche à résoudre le problème :

Problème 7.22 — : MinCut

- Entrée : $G = (S, A)$ un graphe non orienté connexe avec $|S| \geq 2$.
- Sortie : $\Gamma \subset A$ une coupure de cardinal minimal.

Il existe des algorithmes déterministes et polynomiaux pour résoudre ce problème, ils reposent sur des graphes de flot et rappellent l'algorithme vu pour un couplage maximum d'un graphe biparti.

On va présenter ici un algorithme de Monte-Carlo très facile à mettre en œuvre

l'algorithme de Karger. Il repose sur la notion de multigraphes : des graphes où plusieurs arêtes peuvent relier une paire de sommets.

Définition 7.23

Soit $G = (S, A)$ un multigraphe sans boucle et $a \in A$ une arête reliant x et y . La **contraction** de G par a est le multigraphe $G/a = (S \setminus \{y\}, A')$ obtenu en fusionnant y dans x :

- les arêtes reliant x et y (dont a) sont supprimées ;
- chaque arête reliant y à un sommet $z \neq x$ est remplacée par une arête reliant x et z ;
- les autres arêtes sont conservées.

Les arêtes multiples sont conservées : G/a a exactement $|S| - 1$ sommets et $|A| - m$ arêtes, où m est le nombre d'arêtes entre x et y .

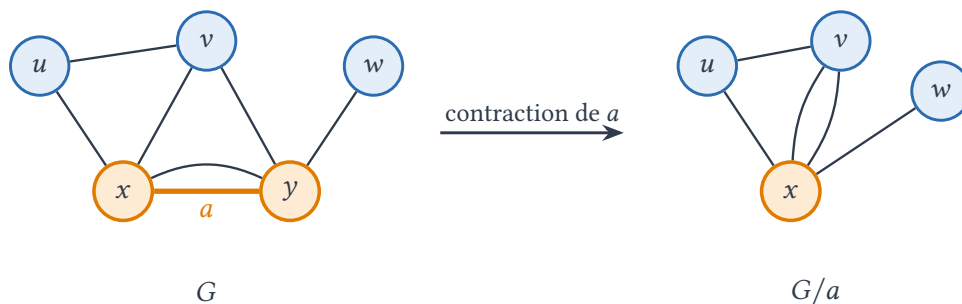


Figure 7.1 — Contraction de l'arête a : y est fusionné dans x

Les arêtes de G/a correspondent à des arêtes de G : on identifie chaque arête à l'arête de G dont elle provient. Une coupure de G/a est alors une coupure de G (les sommets fusionnés restant du même côté), donc la plus petite coupure de G/a est de taille au moins celle de G .

L'algorithme de Karger est alors très simple :

Algorithme 7.24 — de Karger

- Tant que le multigraphe a plus de deux sommets :
 - choisir une arête a uniformément parmi les arêtes ;

- remplacer le multigraphe par sa contraction par a .
- Renvoyer l'ensemble des arêtes (de G) reliant les deux sommets restants.

Chaque sommet restant représente un ensemble de sommets X et $S \setminus X$ de G ; les arêtes restantes sont exactement les arêtes de G entre X et $S \setminus X$: c'est une coupure de G . Comme le graphe est connexe, il reste toujours au moins une arête tant qu'il y a au moins deux sommets.

Théorème 7.25

Soit G un graphe connexe à $n \geq 2$ sommets et Γ une coupure minimale de G . L'algorithme de Karger renvoie Γ avec une probabilité au moins $\frac{2}{n(n-1)}$.

Preuve Soit $k = |\Gamma|$. L'algorithme renvoie Γ si et seulement si aucune arête de Γ n'est contractée au cours des $n-2$ étapes : en effet, Γ est l'ensemble des arêtes entre une partie X et $S \setminus X$ (voir la remarque ci-dessus) et, si aucune de ces arêtes n'est contractée, les sommets de X et ceux de $S \setminus X$ ne sont jamais fusionnés ensemble, et il reste à la fin les deux sommets X et $S \setminus X$.

Notons E_i l'événement « aucune arête de Γ n'est contractée lors des i premières étapes ». Si E_i est réalisé, le multigraphe courant a $n-i$ sommets, contient toutes les arêtes de Γ , et sa coupure minimale est de taille $\geq k$ (c'est aussi une coupure de G) et $\leq k$ (Γ en est une). Par le lemme, il a au moins $k \frac{n-i}{2}$ arêtes. La probabilité de choisir une arête de Γ à l'étape $i+1$ est donc au plus $\frac{k}{k(n-i)/2} = \frac{2}{n-i}$, d'où

$$P(E_{i+1} | E_i) \geq 1 - \frac{2}{n-i} = \frac{n-i-2}{n-i}$$

En enchaînant les probabilités conditionnelles :

$$P(E_{n-2}) \geq \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}$$

après simplification télescopique. Pour $n=2$, il n'y a aucune contraction et le résultat est direct. □

Corollaire 7.26

En répétant l'algorithme $L = \frac{n(n-1)}{2} \ln n$ fois de manière indépendante et en gardant la plus petite coupure obtenue, on obtient une coupure minimale avec probabilité au moins $1 - \frac{1}{n}$.

Preuve On échoue si et seulement si aucune des L exécutions ne renvoie Γ (ou une autre coupure minimale). Par indépendance et avec $1-x \leq e^{-x}$:

$$P(\text{échec}) \leq \left(1 - \frac{2}{n(n-1)}\right)^L \leq e^{-\frac{2L}{n(n-1)}} = e^{-\ln n} = \frac{1}{n}$$
□

Remarque 7.27 C'est un algorithme de Monte-Carlo : il termine toujours, mais peut renvoyer une coupure qui n'est pas minimale, et on ne sait pas vérifier efficacement qu'une coupure est minimale. Corollaire amusant de la preuve : comme les événements « renvoyer Γ » sont disjoints pour des coupures minimales Γ distinctes, un graphe a au plus $\frac{n(n-1)}{2}$ coupures minimales.

Pour l'implémentation, choisir une arête uniforme puis contracter revient à parcourir les arêtes dans un ordre aléatoire uniforme en ignorant celles qui sont devenues des boucles (leurs extrémités sont déjà fusionnées) : à chaque étape, la première arête non boucle restante est uniforme parmi les arêtes non boucles. On retrouve l'algorithme de Kruskal avec un ordre aléatoire, arrêté à deux composantes, et on gère les fusions avec une structure [Union-Find](#) :

OCaml

```
(* union-find minimal avec compression de chemin *)
let rec trouve parent x =
  if parent.(x) = x then x
  else begin
    let r = trouve parent parent.(x) in
    parent.(x) <- r;
    r
  end

let echange t i j =
  let x = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- x

(* mélange de Fisher-Yates : permutation uniforme *)
let melange t =
  for i = Array.length t - 1 downto 1 do
    echange t i (Random.int (i + 1))
  done

(* graphe connexe de sommets 0, ..., n - 1 donné par ses arêtes *)
let karger n (aretes : (int * int) array) : (int * int) list =
  let parent = Array.init n (fun x -> x) in
  let a = Array.copy aretes in
  melange a;
  let nb_sommets = ref n in
  Array.iter (fun (x, y) ->
    let rx = trouve parent x and ry = trouve parent y in
    if !nb_sommets > 2 && rx <> ry then begin
      parent.(ry) <- rx; (* contraction de l'arête *)
      decr nb_sommets
    end
  ) a;
  List.filter (fun (x, y) -> trouve parent x <> trouve parent y)
    (Array.to_list aretes)
```

Sur deux graphes complets K_5 reliés par deux arêtes ($n = 10$, coupure minimale de taille 2), on obtient expérimentalement la coupure minimale dans environ 26 % des exécutions, bien au-dessus de la borne $\frac{2}{90} \approx 2,2\%$. Le TP en propose une implémentation en C qui réalise réellement les contractions.



7.2.4 Quicksort avec pivot aléatoire

Le tri rapide avec le premier élément comme pivot est en $O(n^2)$ dans le pire cas, par exemple sur un tableau déjà trié. En choisissant le pivot uniformément au hasard, on obtient un algorithme de Las Vegas : le tableau renvoyé est toujours trié, seule la durée est aléatoire, et son espérance est en $O(n \log n)$ **pour toute entrée**.

OCaml

```
let echange t i j =
  let x = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- x

(* trie t.(d), ..., t.(f - 1) *)
let rec tri_rapide t d f =
  if f - d > 1 then begin
    echange t d (d + Random.int (f - d)); (* pivot aléatoire placé en tête *)
    let pivot = t.(d) in
    let m = ref d in
    (* invariant : t.(d+1..!m) < pivot <= t.(!m+1..i-1) *)
    for i = d + 1 to f - 1 do
      if t.(i) < pivot then begin
        incr m;
        echange t !m i
      end
    done;
    echange t d !m;
```

```

    tri_rapide t d !m;
    tri_rapide t (!m + 1) f
end

let tri t = tri_rapide t 0 (Array.length t)

```

C

```

void echange(int *t, int i, int j)
{
    int x = t[i];
    t[i] = t[j];
    t[j] = x;
}

/* trie t[d], ..., t[f - 1] */
void tri_rapide(int *t, int d, int f)
{
    if (f - d <= 1)
        return;
    echange(t, d, d + rand() % (f - d)); // pivot aléatoire placé en tête
    int pivot = t[d];
    int m = d;
    // invariant : t[d+1..m] < pivot <= t[m+1..i-1]
    for (int i = d + 1; i < f; i++) {
        if (t[i] < pivot) {
            m++;
            echange(t, m, i);
        }
    }
    echange(t, d, m);
    tri_rapide(t, d, m);
    tri_rapide(t, m + 1, f);
}

```

Théorème 7.28

Pour un tableau de n éléments distincts, l'espérance du nombre de comparaisons effectuées par le tri rapide à pivot aléatoire est au plus $2n \ln n$ (pour $n \geq 1$).

Preuve Notons $z_1 < z_2 < \dots < z_n$ les éléments du tableau dans l'ordre croissant. Deux éléments ne sont comparés que si l'un des deux est le pivot, et un pivot n'est plus jamais comparé après son placement : z_i et z_j sont donc comparés au plus une fois. Soit X_{ij} la variable aléatoire valant 1 s'ils sont comparés, 0 sinon. Le nombre de comparaisons est $X = \sum_{i < j} X_{ij}$.

Pour $i < j$, considérons $Z_{ij} = \{z_i, \dots, z_j\}$. Tant qu'aucun élément de Z_{ij} n'a été choisi comme pivot, tous les éléments de Z_{ij} sont dans le même sous-tableau. Au moment où un premier élément de Z_{ij} est choisi comme pivot :

- si c'est z_i ou z_j , ils sont comparés ;
- sinon, c'est un z_k avec $i < k < j$, et z_i et z_j sont envoyés dans des sous-tableaux différents : ils ne seront jamais comparés.

Comme le pivot est uniforme dans son sous-tableau, qui contient Z_{ij} , chaque élément de Z_{ij} a la même probabilité d'être ce premier pivot, donc $P(X_{ij} = 1) = \frac{2}{j-i+1}$. Par linéarité de l'espérance :

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n \sum_{k=2}^n \frac{1}{k} \leq 2n \ln n$$

car $\sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{dt}{t} = \ln n$.

□

Remarque 7.29

- La complexité totale est proportionnelle au nombre de comparaisons, elle est donc en $O(n \log n)$ en espérance. Il s'agit bien d'une espérance sur les tirages pour une entrée fixée quelconque, et non d'une moyenne sur les entrées : aucun tableau particulier n'est mauvais.
- Avec des éléments répétés, la partition ci-dessus peut se dégrader (un tableau constant donne un temps quadratique) : on utilise alors une partition en trois zones $<$, $=$, $>$.
- En C, `rand() % k` n'est pas exactement uniforme quand k ne divise pas `RAND_MAX + 1`, ce qui est sans importance ici.

7.3 Travaux dirigés et pratiques

- TD : [problème du sac à dos](#) (glouton fractionnaire, programmation dynamique pseudo-polynomiale, schéma d'approximation, NP-complétude).
- TP : [couverture par sommets et TSP métrique](#) (OCaml).
- TP : [sac à dos, générateur aléatoire et algorithme de Karger](#) (C).
- DS 4 (MPI 2022-2023) : 3-coloration par marche aléatoire, partition approchée.

Algorithmes d'approximation (Vertex Cover et TSP)

8.1 Vertex Cover et TSP

On va reprendre ici certains algorithmes étudiés dans le [cours](#) pour réaliser des approximations.

8.1.1 MinVertexCover

On considère un graphe non orienté donné en OCaml uniquement par ses listes d'adjacence en identifiant un sommet à son indice.

```
let g = [[1];[0;2];[1;3;4];[2;4;5;6];[2;3;5];[3;4;6];[3;5]]
```

Question 1 Écrire une fonction `aretes` qui prend un graphe et renvoie la liste de ses arêtes sous la forme d'une liste de couples (i, j) où $i < j$.

Corrigé

```
let aretes ladj =
  ladj
  |> Array.mapi (fun i l -> (* pour chaque sommet i et liste adj l *)
    l |> List.filter ((<)i) (* on ne garde que les voisins > i *)
    |> List.map (fun j -> (i,j))) (* on transforme les voisins en aretes *)
  |> Array.fold_left (@) [] (* on concatene le tableau de listes *)
```

Sur `g`, on obtient $[(0,1); (1,2); (2,3); (2,4); (3,4); (3,5); (3,6); (4,5); (5,6)]$.

Question 2 Écrire une fonction `retire_coincidentes` qui prend une arête a et une liste d'arêtes l et renvoie la liste $l' \subset l$ des arêtes qui ne partagent pas d'extrémités avec a .

Corrigé

```
let retire_coincidentes (x,y) =
  List.filter (fun (x',y') -> x <> x' && y <> y' && x <> y' && y <> x')
```

Question 3 En déduire une fonction `min_vertex_cover_2a` qui renvoie une couverture de sommets par la 2-approximation du cours.

Corrigé

```
let min_vertex_cover_2a ladj =
  let rec aux a =
    match a with
    | [] -> []
    | (x,y)::q -> x::y::aux (retire_coincidentes (x,y) q)
  in aux (aretes ladj)
```

Question 4 Testez vos fonctions.**Corrigé**

Sur g , on obtient $[0; 1; 2; 3; 4; 5]$, soit 6 sommets, alors qu'une couverture optimale en a 4 (par exemple $\{1, 3, 4, 5\}$; il en faut au moins un pour l'arête $\{0, 1\}$ et au moins trois parmi 2, ..., 6 pour couvrir les triangles 2, 3, 4, 3, 4, 5 et 3, 5, 6 et l'arête $\{5, 6\}$). On a bien $6 \leq 2 \times 4$.

Pour tester plus systématiquement, on peut comparer sur des graphes aléatoires à une recherche exhaustive de la couverture optimale (on teste les 2^n sous-ensembles de sommets pour $n \leq 10$).

8.1.2 Union-find**Question 5** Reprendre la définition de la structure Union-Find en OCaml (voir le [cours](#)).

On pourra par exemple considérer le type :

```
type 'a union_find = {
  values_to_index : ('a, int) Hashtbl.t;
  parent : int array;
  rank : int array
}
```

où `values_to_index` permet d'accéder aux éléments par leur valeur.

On pourra alors définir des fonctions :

```
(* crée un union_find *)
let make_sets (l:'a list) : 'a union_find = ...
(* renvoie l'indice de la racine de l'ensemble contenant x *)
let find (uf:'a union_find) (x:'a) : int = ...
(* fusionne deux ensembles donnés par leur racine *)
let union (uf:'a union_find) (i:int) (j:int) = ...
```

Corrigé

```
(* crée un union_find *)
let make_sets (l:'a list) : 'a union_find =
  let n = List.length l in
  let h = Hashtbl.create n in
  List.iteri (fun i a -> Hashtbl.add h a i) l;
  { values_to_index = h; parent = Array.init n Fun.id; rank = Array.make n 0 }

(* renvoie l'indice de la racine de l'ensemble contenant x *)
let find (uf:'a union_find) (x:'a) : int =
  let rec remonte i =
    if uf.parent.(i) = i
    then i
    else remonte uf.parent.(i)
  in remonte (Hashtbl.find uf.values_to_index x)

(* fusionne deux ensembles donnés par leur racine *)
let union (uf:'a union_find) (i:int) (j:int) =
  if i <> j
  then begin
    let ri = uf.rank.(i) in
    let rj = uf.rank.(j) in
    if ri = rj
    then begin
      uf.parent.(j) <- i;
      uf.rank.(i) <- ri + 1
    end else if ri < rj then uf.parent.(i) <- j else uf.parent.(j) <- i
  end
```

```
let size uf = Array.length uf.parent
let mem uf x = Hashtbl.mem uf.values_to_index x
```

Application à 3SatAllOrNothing

On rappelle le problème 3SatAllOrNothing (voir les [exercices de calculabilité](#)) : il s'agit étant donné une formule de 3-SAT de déterminer s'il existe une valuation telle que chaque clause contient soit trois littéraux vrais, soit trois littéraux faux.

Pour résoudre ce problème, on va définir une relation d'équivalence sur les littéraux : la plus petite relation d'équivalence telle que $l \equiv l'$ dès que l et l' sont dans une même clause de la formule. Deux littéraux équivalents doivent avoir la même valeur. Il suffira alors de vérifier dans un second temps qu'on peut donner une valeur à chaque classe de façon cohérente : les classes de x et de $\neg x$ doivent recevoir des valeurs opposées, ce qui revient à 2-colorier le graphe dont les sommets sont les classes et dont les arêtes relient la classe de x à celle de $\neg x$.

On considère le type suivant :

```
type lit = Pos of char | Neg of char
type sat3 = (lit * lit * lit) list
```

Question 6 Résoudre le problème en utilisant une structure union_find.

Corrigé

```
let litteraux f =
  List.map (fun (i,j,k) -> [i;j;k]) f
  |> List.concat
  |> List.sort_uniq Stdlib.compare

let variables f =
  litteraux f
  |> List.map (function Pos x | Neg x -> x)
  |> List.sort_uniq Stdlib.compare

let construit_classes f =
  let uf = make_sets (litteraux f) in
  List.iter (fun (l1, l2, l3) ->
    union uf (find uf l1) (find uf l2);
    union uf (find uf l1) (find uf l3)) f;
  uf

let deux_coloriable ladj =
  let n = Array.length ladj in
  let echec = ref false in
  let couleurs = Array.make n (-1) in
  let rec parcours couleurs col x =
    if couleurs.(x) = -1
    then begin
      couleurs.(x) <- col;
      List.iter (fun j -> parcours couleurs ((col+1) mod 2) j) ladj.(x)
    end else if couleurs.(x) <> col then echec := true
  in
  for i = 0 to n - 1 do
    if couleurs.(i) = -1
    then parcours couleurs 0 i
  done;
  not !echec, couleurs

let resout f =
  let var = variables f in
```

```

let classes = construit_classes f in
let ladj = Array.make (size classes) [] in
List.iter (fun v ->
  if mem classes (Pos v) && mem classes (Neg v)
  then begin
    let pi = find classes (Pos v) and ni = find classes (Neg v) in
    if not (List.mem ni ladj.(pi)) then ladj.(pi) <- ni :: ladj.(pi);
    if not (List.mem pi ladj.(ni)) then ladj.(ni) <- pi :: ladj.(ni)
  end) var;
deux_coloriable ladj

resout [ (Pos 'a', Pos 'b', Pos 'c');
        (Pos 'a', Pos 'b', Neg 'd');
        (Pos 'd', Neg 'b', Neg 'c')]

```

La première composante du résultat vaut true ici. Si x et $\neg x$ sont dans la même classe, la classe est reliée à elle-même et le 2-coloriage échoue, comme attendu. Le tableau `couleurs` (indexé par les racines) donne une valuation : par exemple, les littéraux de couleur 0 sont vrais.

8.1.3 TSP métrique

Le problème du TSP métrique consiste, étant donné un graphe complet $G = (S, A)$ non orienté et une fonction de distance $d : S^2 \rightarrow \mathbb{R}_+$ vérifiant l'inégalité triangulaire

$$\forall x, y, z \in S, d(x, y) \leq d(x, z) + d(z, y)$$

à déterminer la longueur minimale d'un cycle hamiltonien de G .

Ce problème est NP-difficile. On va donner une 2-approximation en considérant l'algorithme suivant :

- on construit un arbre couvrant de poids minimal (par exemple, avec Kruskal) ;
- on considère le cycle induit par le préordre d'un parcours de l'arbre couvrant.

Question 7 On vous demande ici de mettre en place cette approximation pour résoudre l'instance suivante :

```

let points = [
  (525,187); (295,946); (364,787); (533,5); (203,862); (625,11); (824,675);
  (369,58); (308,785); (726,766); (155,22); (197,879); (881,599); (862,953);
  (328,391); (983,744); (426,695); (478,583); (821,911); (79,60); (665,272);
  (191,773); (114,281); (800,567); (222,693); (846,961); (215,740); (353,769);
  (750,726); (908,909); (309,658); (867,649); (592,106); (16,637); (54,796);
  (449,649); (980,654); (594,819); (764,101); (921,307); (928,647); (738,196);
  (550,578); (478,489); (449,755); (189,415); (433,545); (967,478); (149,416);
  (144,117); (298,972); (854,144)
]

```

Corrigé

```

type 'a arbre = Noeud of 'a * 'a arbre list

let arbre valeurs ladj racine =
  let visite = Array.make (Array.length ladj) false in
  let rec construit x =
    visite.(x) <- true;
    Noeud(valeurs.(x),
      List.map construit (List.filter (fun y -> not visite.(y)) ladj.(x)))
  in construit !racine

let rec preordre n =
  let Noeud(x,l) = n in
  x :: List.concat (List.map preordre l)

```

```

let kruskal dist points =
  let n = Array.length points in
  let aretes = ref [] in
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      aretes := (i, j, dist points.(i) points.(j))::!aretes
    done
  done;
  let aretes = List.sort (fun (_,_,a) (_,_,b) -> Stdlib.compare a b) !aretes in
  let ladj = Array.make n [] in
  let uf = make_sets (List.init n Fun.id) in
  List.iter (fun (i,j,_ ) ->
    let ri = find uf i and rj = find uf j in
    if ri <> rj
    then begin
      union uf ri rj;
      ladj.(i) <- j :: ladj.(i);
      ladj.(j) <- i :: ladj.(j)
    end) aretes;
  let racine = ref 0 in
  while uf.parent(!racine) <> !racine do
    incr racine
  done;
  arbre points ladj racine

let longueur dist l =
  let p0 = List.hd l in
  let rec aux l =
    match l with
    | [] -> 0.
    | [p] -> dist p0 p
    | p1::p2::q -> dist p1 p2 +. aux (p2::q)
  in aux l

let dist_euclidienne (x,y) (x',y') =
  sqrt (float_of_int ((x-x')*(x-x') + (y-y')*(y-y')))

let approx_tsp dist =
  longueur dist (preordre (kruskal dist points))

approx_tsp dist_euclidienne

```

On obtient un cycle de longueur environ 7521,27, pour un arbre couvrant minimal de poids environ 4809,02 : la longueur optimale est donc comprise entre 4809 et 7521.

Question 8 Même question, mais cette fois-ci, on considère que les points sont sur un tore. Concrètement, cela signifie qu'ils sont sur un carré de dimension $\llbracket 0, 999 \rrbracket^2$ dont les bords gauche/droite et haut/bas sont identifiés.

Note : il s'agit d'un challenge présent sur le site <http://caesum.com/game> pour avoir le droit d'être anonyme sur ses forums. Le record sur le site est de 5107,36 le 9 novembre 2001.

Corrigé

```

let dist_tore (x,y) (x',y') =
  let dx = min (min (abs (x-x')) (abs (x-x'-1000))) (abs (1000+x-x')) in
  let dy = min (min (abs (y-y')) (abs (y-y'-1000))) (abs (1000+y-y')) in
  sqrt (float_of_int (dx * dx + dy * dy))

approx_tsp dist_tore

```

On obtient environ 6556,18 (arbre couvrant minimal de poids environ 4508,06), assez loin du record.

Problème du sac à dos

9.1 Problème du sac à dos

Ce TD accompagne le chapitre *Algorithmes d'approximation, algorithmes probabilistes*.

On considère le problème suivant :

Problème 9.1 — : Knapsack

- Entrée :
 - n objets numérotés de 1 à n ;
 - à chaque objet on associe deux entiers : $w_i > 0$ son poids et $v_i > 0$ sa valeur ;
 - un poids maximal $W \in \mathbb{N}$.
- Sortie : pour $S \subset \llbracket 1, n \rrbracket$, on note $w(S) = \sum_{x \in S} w_x$ et $v(S) = \sum_{x \in S} v_x$. On cherche

$$\max\{v(S) \mid S \subset \llbracket 1, n \rrbracket, w(S) \leq W\}$$

On cherche ainsi la valeur maximale d'un ensemble d'objets dont le poids est sous la limite.

9.1.1 Cas fractionnaire et algorithme glouton

On considère ici une variante où on peut prendre partiellement les objets.

Problème 9.2 — : FractionalKnapsack

- Entrée : les mêmes données que pour Knapsack.
- Sortie : pour $\sigma : \llbracket 1, n \rrbracket \rightarrow \mathbb{Q}_+ \cap [0; 1]$, on note $w(\sigma) = \sum_{x=1}^n w_x \sigma(x)$ et $v(\sigma) = \sum_{x=1}^n v_x \sigma(x)$. On cherche

$$\max\{v(\sigma) \mid \sigma : \llbracket 1, n \rrbracket \rightarrow \mathbb{Q}_+ \cap [0; 1], w(\sigma) \leq W\}$$

Pour chaque objet, on calcule son efficacité $e_i = v_i/w_i$. On trie alors les objets par efficacité décroissante et on sélectionne tous les objets tant qu'on reste sous le poids, le premier objet qui force un dépassement est placé de manière fractionnelle.

C'est un algorithme glouton. On va montrer qu'il renvoie une solution optimale.

On va se placer dans le cas où $\sum_{i=1}^n w_i > W$ (sinon, prendre tous les objets en entier est optimal).

Question 1 En déduire que si τ est une solution optimale, alors $w(\tau) = W$.

Corrigé

Si $w(\tau) < W$, comme $\sum w_i > W$, il existe un objet i avec $\tau(i) < 1$. On peut alors augmenter $\tau(i)$ de $\min\left(1 - \tau(i), \frac{W - w(\tau)}{w_i}\right) > 0$ sans dépasser W , ce qui augmente strictement la valeur car $v_i > 0$: contradiction.

On suppose qu'on a obtenu une solution gloutonne donnée par σ et qu'on dispose également d'une solution optimale τ . Pour simplifier, on peut supposer les objets rangés par efficacité décroissante, ainsi 1 est l'objet d'efficacité maximale, et $\sigma(1) = \min\left(1, \frac{W}{w_1}\right)$.

Question 2 Montrer que si $\sigma(1) > \tau(1)$, alors on peut en déduire une solution optimale τ' telle que $\tau'(1) = \sigma(1)$.

Corrigé

On pose $\delta W = (\sigma(1) - \tau(1))w_1 > 0$. Comme $w(\tau) = W$, on a $\sum_{i=2}^n \tau(i)w_i = W - \tau(1)w_1$, et $W - \tau(1)w_1 = (W - \sigma(1)w_1) + \delta W > 0$ car $\sigma(1)w_1 \leq W$. On pose alors $\chi = \frac{\delta W}{W - \tau(1)w_1}$, on a

$$\chi \sum_{i=2}^n \tau(i)w_i = \chi(W - \tau(1)w_1) = \delta W$$

or, $\sigma(1)w_1 \leq W$ donc $\delta W \leq W - \tau(1)w_1$, c'est-à-dire $\chi \leq 1$.

On définit ainsi τ' par

$$\tau'(1) = \sigma(1)$$

$$\forall i > 1, \tau'(i) = (1 - \chi)\tau(i)$$

qui est bien à valeurs dans $[0; 1]$. On a

$$w(\tau') = w(\tau) + (\sigma(1) - \tau(1))w_1 - \chi \sum_{i=2}^n \tau(i)w_i = w(\tau) + \delta W - \delta W = w(\tau) = W$$

Et

$$\begin{aligned} v(\tau') &= v(\tau) + (\sigma(1) - \tau(1))v_1 - \chi \sum_{i=2}^n \tau(i)v_i \\ &= v(\tau) + \delta W \frac{v_1}{w_1} - \chi \sum_{i=2}^n \tau(i)v_i \\ &= v(\tau) + \chi \frac{v_1}{w_1} \sum_{i=2}^n \tau(i)w_i - \chi \sum_{i=2}^n \tau(i)v_i \\ &= v(\tau) + \chi \sum_{i=2}^n \tau(i) \left(w_i \frac{v_1}{w_1} - v_i \right) \\ &\geq v(\tau) \end{aligned}$$

$$\text{car } \frac{v_1}{w_1} \geq \frac{v_i}{w_i}.$$

Ainsi, on a τ' solution admissible et par maximalité, $v(\tau') = v(\tau)$ et τ' est optimale. Or $\tau'(1) = \sigma(1)$.

Question 3 Justifier que si τ est optimale pour les objets de 1 à n , alors τ^* la restriction de τ à $\llbracket 2, n \rrbracket$ est également optimale pour cette instance avec $W - \tau(1)w_1$ comme poids limite.

Corrigé

τ^* est admissible : $w(\tau^*) = w(\tau) - \tau(1)w_1 \leq W - \tau(1)w_1$. Si une solution ρ des objets 2 à n avec ce poids limite vérifiait $v(\rho) > v(\tau^*)$, en la complétant par $\tau(1)$ pour l'objet 1, on obtiendrait une solution admissible de l'instance initiale de valeur $\tau(1)v_1 + v(\rho) > v(\tau)$: contradiction.

Question 4 Justifier que σ^* , la restriction de σ à $\llbracket 2, n \rrbracket$, est également donnée par l'algorithme glouton pour les objets de 2 à n et le poids limite $W - \sigma(1)w_1$.

Question 5 En déduire, par récurrence sur le nombre d'objets, que le glouton renvoie une solution optimale et ainsi que FractionalKnapsack est dans P.

Corrigé

Par récurrence sur n , le cas $n = 0$ étant trivial. Si $\sum w_i \leq W$, le glouton prend tout, ce qui est optimal. Sinon, soit τ optimale ; on a $\tau(1) \leq \sigma(1)$ car $\sigma(1)$ est la plus grande valeur admissible, et par la question 2 on peut supposer $\tau(1) = \sigma(1)$. Par la question 3, τ^* est optimale pour les objets 2 à n et le poids $W - \sigma(1)w_1$; par la question 4 et l'hypothèse de récurrence, σ^* aussi, donc $v(\sigma^*) = v(\tau^*)$ et $v(\sigma) = \sigma(1)v_1 + v(\sigma^*) = v(\tau)$.

Le glouton se résume à un tri et un parcours : il est en $O(n \log n)$, donc FractionalKnapsack est dans P.

9.1.2 Pseudo-polynomialité

On fixe ici une instance de Knapsack.

On considère $V = \max_{i=1}^n v_i$ la valeur maximale d'un objet. On sait que nV est un majorant direct de la solution optimale.

On va alors poser pour $i \in \llbracket 0, n \rrbracket$ et $p \in \llbracket 0, nV \rrbracket$, $A(i, p)$ le plus petit poids d'un sous-ensemble des objets $\leq i$ dont la valeur est **exactement** p . Quand un tel sous-ensemble n'existe pas, on pose $A(i, p) = \infty$.

Question 6 Déterminer les valeurs pour lesquelles on peut directement calculer $A(i, p)$.

Corrigé

On a $A(i, 0) = 0$ pour tout i et $A(0, p) = \infty$ pour $p > 0$.

Question 7 Déterminer une équation de Bellman pour A .

Corrigé

Soit $i \geq 1$ et $p \in \llbracket 1, nV \rrbracket$, si $v_i > p$, alors on ne peut pas utiliser l'objet i , donc $A(i, p) = A(i-1, p)$. Sinon, utiliser l'objet i va permettre au mieux un poids de $w_i + A(i-1, p - v_i)$, qui n'a pas de raison d'être *a priori* meilleur que $A(i-1, p)$, on prend donc le minimum :

$$A(i, p) = \begin{cases} \min(A(i-1, p), w_i + A(i-1, p - v_i)) & \text{si } v_i \leq p \\ A(i-1, p) & \text{sinon.} \end{cases}$$

Question 8 En déduire un algorithme calculant la solution de Knapsack en $O(n^2V)$.

Corrigé

On remplit le tableau A de taille $(n+1) \times (nV+1)$ ligne par ligne avec l'équation précédente, chaque case en $O(1)$, soit $O(n^2V)$. La valeur optimale est $\max\{p \in \llbracket 0, nV \rrbracket \mid A(n, p) \leq W\}$. Pour obtenir les objets, on remonte depuis la case (n, p) optimale : si $A(i, p) = A(i-1, p)$, on passe à $(i-1, p)$, sinon on prend l'objet i et on passe à $(i-1, p - v_i)$. Une implémentation en C est donnée dans le TP.

Question 9 On a Knapsack dans P si l'instance est donnée sous quelle représentation ?

Corrigé

Si les valeurs sont données en unaire, alors V est de l'ordre de la taille des données, donc la solution en programmation dynamique est polynomiale. En binaire, la taille de l'entrée est de l'ordre de $n \log V$ et n^2V est exponentiel en cette taille : on dit que l'algorithme est **pseudo-polynomial**.

9.1.3 Schéma d'approximation polynomial complet

Soit $0 < \epsilon < 1$. On va montrer qu'il est possible d'obtenir en temps polynomial en n et $\frac{1}{\epsilon}$ une solution de valeur au moins $(1 - \epsilon)OPT$, c'est-à-dire une $\frac{1}{1-\epsilon}$ -approximation au sens du chapitre, où $\frac{1}{1-\epsilon} = 1 + \epsilon + o(\epsilon)$. Quitte à les éliminer, on suppose que chaque objet rentre seul dans le sac : $w_i \leq W$ pour tout i .

On pose $K = \frac{\epsilon V}{n}$ et pour chaque objet i , on définit une nouvelle valeur $v'_i = \lfloor \frac{v_i}{K} \rfloor$. On calcule alors une solution optimale S' avec ces nouvelles valeurs par l'algorithme de programmation dynamique de la partie précédente et on renvoie S' .

Question 10 Calculer la complexité de cette résolution.

Corrigé

On a $V' = \max v'_i \leq \max \frac{v_i}{K} = \frac{V}{K} = \frac{n}{\epsilon}$, donc la complexité est en $O(n^2 V') = O(n^3 \frac{1}{\epsilon})$.

Question 11 Montrer que $v(S') \geq (1 - \epsilon)OPT$ où OPT est la valeur optimale.

Corrigé

Soit S^* une solution optimale pour les valeurs v_i : c'est aussi une solution admissible pour les valeurs v'_i (les poids n'ont pas changé). Comme S' est optimale pour les v'_i et que $K v'_x \leq v_x$ et $K v'_x > v_x - K$:

$$v(S') \geq K \sum_{x \in S'} v'_x \geq K \sum_{x \in S^*} v'_x \geq \sum_{x \in S^*} (v_x - K) \geq OPT - nK = OPT - \epsilon V$$

Or chaque objet rentre seul dans le sac, donc $OPT \geq V$, et ainsi $v(S') \geq (1 - \epsilon)OPT$.

Les schémas d'approximation polynomiaux complets sont très fortement liés à la pseudo-polynomialité.

9.1.4 NP-complétude

On s'intéresse ici au problème de décision associé à Knapsack : étant donné de plus un seuil k , existe-t-il un sous-ensemble S tel que $w(S) \leq W$ et $v(S) \geq k$?

Question 12 Montrer que Knapsack est dans NP.

Pour montrer que Knapsack est NP-complet, on va utiliser un problème intermédiaire ExactCover et effectuer la chaîne de réduction :

$$\text{Fira Sans(Coloration)} \rightarrow \text{Fira Sans(ExactCover)} \rightarrow \text{Fira Sans(Knapsack)}$$

On rappelle Coloration dont on a montré qu'il était NP-complet par réduction de 3-Coloriable lui-même par réduction de 3-SAT.

Problème 9.3 — : Coloration

- Entrée : un graphe non orienté $G = (S, A)$ et un entier k .
- Sortie : existe-t-il une application $\gamma : S \rightarrow \llbracket 1, k \rrbracket$ telle que $\forall \{x, y\} \in A, \gamma(x) \neq \gamma(y)$?

Le problème ExactCover est une variante des problèmes déjà rencontrés de couverture :

Problème 9.4 — : ExactCover

- Entrée : un ensemble fini Σ et une suite de sous-ensembles $S_1, \dots, S_n \subset \Sigma$.
- Sortie : existe-t-il une sous-famille $1 \leq i_1 < \dots < i_k \leq n$ telle que les S_{i_j} soient deux à deux disjoints et que $\bigcup_{j=1}^k S_{i_j} = \bigcup_{j=1}^n S_j$? (Autrement dit, peut-on extraire une partition de $\bigcup_j S_j$ parmi les S_i ?)

Question 13 Montrer que ExactCover est dans NP.

Question 14 On considère une instance de Coloration donnée par un graphe $G = (V, A)$ et un entier k . On fixe un ordre total sur V et on note chaque arête $a = \{x_a, y_a\}$ avec $x_a < y_a$. On pose

$$\Sigma = V \cup A \cup \{(x, a, c) \mid a \in A, x \in a, c \in \llbracket 1, k \rrbracket\}$$

et on considère les sous-ensembles suivants :

$$\forall c \in \llbracket 1, k \rrbracket, \forall x \in V, S_{x,c} = \{x\} \cup \{(x, a, c) \mid a \in A \text{ où } x \in a\}$$

$$\forall a \in A, \forall c_1, c_2 \in \llbracket 1, k \rrbracket \text{ tels que } c_1 \neq c_2, S_{a,c_1,c_2} = \{a\} \cup \{(x_a, a, c) \mid c \neq c_1\} \cup \{(y_a, a, c) \mid c \neq c_2\}$$

Montrer qu'on a défini ainsi une réduction polynomiale de Coloration vers ExactCover.

Question 15 On considère une instance de ExactCover donnée par Σ et $S_1, \dots, S_n \subset \Sigma$. Sans perte de généralité, on peut supposer que $\Sigma = \bigcup_{j=1}^n S_j = \llbracket 1, p \rrbracket$.

Pour $i \in \llbracket 1, p \rrbracket, j \in \llbracket 1, n \rrbracket$, on pose $\epsilon_{i,j} = \begin{cases} 0 & \text{si } i \notin S_j \\ 1 & \text{sinon.} \end{cases}$

On pose également $d = n + 1$ et à chaque sous-ensemble S_j , on associe l'entier $a_j = \sum_{i=1}^p \epsilon_{i,j} d^{i-1}$.

On pose également $W = d^0 + \dots + d^{p-1}$.

On va considérer l'instance du sac à dos avec W comme poids maximal et pour chaque objet $j \in \llbracket 1, n \rrbracket$, $w_j = v_j = a_j$. On considère alors W comme seuil.

- Détailler la construction pour $S_1 = \{1\}, S_2 = \{3, 4\}, S_3 = \{1, 2, 4\}, S_4 = \{1, 2\}$ et $S_5 = \{2, 3, 4\}$.
- Montrer qu'on a ainsi défini une réduction polynomiale de ExactCover à Knapsack et en déduire que Knapsack est NP-complet.

Corrigé (1.)

Ici $n = 5, d = 6$ et $\Sigma = \llbracket 1, 4 \rrbracket$ (la source prenait $\Sigma = \llbracket 1, 5 \rrbracket$, mais 5 n'appartient à aucun S_j). On écrit les a_j en base 6, le chiffre de rang $i - 1$ indiquant si $i \in S_j$:

j	S_j	base 6	a_j
1	$\{1\}$	0001	1
2	$\{3, 4\}$	1100	252
3	$\{1, 2, 4\}$	1011	223
4	$\{1, 2\}$	0011	7
5	$\{2, 3, 4\}$	1110	258

et $W = 1111_6 = 259$. Les sous-ensembles d'objets de poids total exactement 259 sont $\{1, 5\}$ ($1 + 258$) et $\{2, 4\}$ ($252 + 7$), qui correspondent aux deux partitions $S_1 \cup S_5$ et $S_2 \cup S_4$. Comme on additionne au plus $n < d$ chiffres égaux à 1, il n'y a jamais de retenue : c'est le rôle du choix $d = n + 1$.

Sac à dos et algorithmes probabilistes

10.1 Sac à dos et algorithmes probabilistes

Ce TP en C accompagne le chapitre Algorithmes d'approximation, algorithmes probabilistes et le TD sur le sac à dos.

10.1.1 Implémentation du sac à dos

On va supposer que l'instance du sac à dos est décrite par :

- `int n` le nombre d'éléments ($n \geq 1$) ;
- `int *w` le tableau des poids (strictement positifs) ;
- `int *v` le tableau des valeurs (strictement positives) ;
- `int w` le poids max.

Cas fractionnaire

Question 1 Écrire une fonction de prototype

```
float *resout_sacados_frac(int n, int *w, int *v, int W);
```

qui alloue et renvoie un tableau contenant la solution gloutonne du sac à dos fractionnaire.

On pourra utiliser le tri à bulle suivant qui place dans la variable `ordre` un tableau des indices des objets dans l'ordre décroissant de l'efficacité.

```
int *ordre = malloc(sizeof(int) * n);
for(int i = 0; i < n; i++)
    ordre[i] = i;
for(int i = 0; i < n; i++)
{
    for(int j = 0; j < n-1; j++)
    {
        float ej = (float)v[ordre[j]]
            / (float)w[ordre[j]];
        float ejn = (float)v[ordre[j+1]]
            / (float)w[ordre[j+1]];
        if (ej < ejn) {
            int temp = ordre[j];
            ordre[j] = ordre[j+1];
            ordre[j+1] = temp;
        }
    }
}
```

Corrigé

```
float *resout_sacados_frac(int n, int *w, int *v, int W)
{
```

```

int *ordre = malloc(sizeof(int) * n);
for(int i = 0; i < n; i++)
    ordre[i] = i;
for(int i = 0; i < n; i++)
{
    for(int j = 0; j < n-1; j++)
    {
        float ej = (float)v[ordre[j]]
            / (float)w[ordre[j]];
        float ejn = (float)v[ordre[j+1]]
            / (float)w[ordre[j+1]];
        if (ej < ejn) {
            int temp = ordre[j];
            ordre[j] = ordre[j+1];
            ordre[j+1] = temp;
        }
    }
}

float *f = malloc(sizeof(float) * n);
for(int i = 0; i < n; i++)
    f[i] = 0.0;

int o = 0;
while(o < n && W >= w[ordre[o]]) {
    W -= w[ordre[o]];
    f[ordre[o]] = 1.0;
    o++;
}

if (o < n)
    f[ordre[o]] = (float)W/(float)w[ordre[o]];

free(ordre);

return f;
}
    
```

Programmation dynamique

Question 2 Écrire une fonction de prototype

```
bool *resout_sacados(int n, int *w, int *v, int W);
```

qui alloue et renvoie un tableau contenant la solution exacte par programmation dynamique sous la forme d'un tableau de n booléens indiquant si on a pris le i -ème élément. On utilisera le tableau $A(i, p)$ du TD (plus petit poids d'un sous-ensemble des i premiers objets de valeur exactement p).

Corrigé

Le tableau A est stocké à plat : la case (i, p) est à l'indice $p + i*(n*V+1)$. La valeur $w + 1$ joue le rôle de ∞ (seules les cases $\leq W$ nous intéressent). Le tableau `used` mémorise si l'objet i est utilisé dans le minimum, pour reconstruire la solution.

```

bool *resout_sacados(int n, int *w, int *v, int W)
{
    bool *f = malloc(sizeof(bool) * n);
    for(int i = 0; i < n; i++)
        f[i] = false;

    int V = v[0];
    for (int i = 1; i < n; i++)
        if(v[i] > V)
            V = v[i];
    
```

```

int *tab = malloc(sizeof(int) * (n+1)*(1+n*V));
bool *used = malloc(sizeof(bool) * (n+1)*(1+n*V));
for(int i = 0; i < (n+1)*(n*V+1); i++)
{
    used[i] = false;
    tab[i] = W + 1;
}
tab[0] = 0;
for(int i = 1; i <= n; i++) {
    tab[i * (n*V+1)] = 0;
    for(int p = 1; p <= n * V; p++) {
        int val = tab[p + (i-1)*(n*V+1)];
        if (p >= v[i-1]) {
            int valw = w[i-1] + tab[(i-1)*(n*V+1)+p-v[i-1]];
            if (valw < val) {
                used[p + i*(n*V+1)] = true;
                val = valw;
            }
        }
        tab[p + i*(n*V+1)] = val;
    }
}

// plus grande valeur atteignable avec un poids <= W
int p = n * V;
while(p > 0 && tab[n*(n*V+1) + p] > W) p--;

// reconstruction de la solution
int i = n;
while(p != 0) {
    if (used[p+i*(n*V+1)]) {
        f[i-1] = true;
        p = p - v[i-1];
    }
    i--;
}

free(tab);
free(used);

return f;
}

```

Schéma d'approximation (FPTAS)

Question 3 Écrire une fonction de prototype

```
bool *approx_sacados(int n, int *w, int *v, int W, float eps);
```

qui renvoie la solution de l'algorithme d'approximation du TD pour la précision ϵ .

Corrigé

On suppose, comme dans le TD, que chaque objet rentre seul dans le sac (sinon on l'élimine au préalable).

```

bool *approx_sacados(int n, int *w, int *v, int W, float eps)
{
    int v = 0;
    for (int i = 0; i < n; i++)
    {
        if (v[i] > v)
            v = v[i];
    }
    float factor = (v * eps) / n;

```

```
int *va = malloc(sizeof(int) * n);
for (int i = 0; i < n; i++)
{
    va[i] = (int)(v[i] / factor);
}
bool *f = resout_sacados(n, w, va, W);
free(va);
return f;
}
```

Des tests

```
int n = 10;
int W = 842;
int w[] = {415, 241, 555, 842, 841, 59, 626, 535, 715, 541};
int v[] = {515, 341, 655, 942, 941, 159, 726, 635, 815, 641};
```

Qui doit renvoyer

```
// pour fractionnaire
{ 1.0, 1.0, 0.0, 0.0, 0.0,
  1.0, 0.0, 0.237, 0.0, 0.0 }
// pour la programmation dynamique
// objets 1, 5, 9 pour une valeur 1141
0 1 0 0 0 1 0 0 0 1
```

Pour $\epsilon = 0,5$, l'approximation renvoie par exemple les objets 1, 5 et 7, de valeur $1135 \geq 0,5 \times 1141$.

On pourra aussi tester `resout_sacados` sur de petites instances aléatoires en comparant avec une recherche exhaustive sur les 2^n sous-ensembles, et vérifier que `approx_sacados` renvoie une solution admissible de valeur $\geq (1 - \epsilon)OPT$.

On peut trouver une calculatrice en ligne ici pour vérifier les résultats : <https://augustineaykara.github.io/Knapsack-Calculator/>

10.1.2 Fonctionnement d'un générateur aléatoire

On va créer ici notre propre générateur aléatoire. Il s'agit d'une famille de suites, comme peuvent l'être les suites de Syracuse, qui dépend de la valeur de u_0 , appelée le **germe** du générateur.

Voici le générateur RANDU utilisé par IBM dans les années 1960, de très mauvaise qualité, qui a induit en erreur des centaines de scientifiques :

$$u_{n+1} \equiv (65539u_n) \bmod 2^{31}$$

avec u_0 un nombre impair.

Note : on peut écrire $x \& 0x7fffffff$ pour calculer le modulo 2^{31} d'un entier positif.

Ici, ce générateur ne renvoie que des nombres impairs. On en déduit un générateur de `float` dans $]0;1$ par division : $f_n = \frac{u_n}{2^{31}}$.

Question 4 On utilisera une variable globale de type `int` pour contenir le dernier terme calculé de la suite.

Écrire des fonctions de prototype

```
void srand_ibm(int seed);
int rand_ibm();
```

qui initialise le germe (on rajoute 1 s'il est pair) pour `srand_ibm` et renvoie le prochain terme de la suite (et change donc la valeur de la variable globale).

Corrigé

Le produit $65539 * u$ dépasse la capacité d'un `int` : en C, un dépassement sur un entier signé est un comportement indéfini. On calcule donc en `unsigned int`, où les calculs se font modulo 2^{32} , avant de réduire modulo 2^{31} .

```
int u = 1;
void srand_ibm(int seed)
{
    u = seed + 1 - seed % 2;
}

int rand_ibm()
{
    u = (int)((65539u * (unsigned int)u) & 0x7fffffffu);
    return u;
}
```

On en déduit, grâce à la fonction `ldexpf(f, n)` de `math.h` qui calcule $f \times 2^n$, une fonction `randu` renvoyant un flottant :

```
float randu()
{
    return ldexpf(rand_ibm(), -31);
}
```

On peut voir que ce générateur est mauvais en traçant les points (f_k, f_{k+1}, f_{k+2}) formés de trois valeurs successives : ils se répartissent sur 15 plans parallèles.

! [Triplets successifs (f_k, f_{k+1}, f_{k+2}) de `randu` : ils se répartissent sur 15 plans parallèles. {width=75%}]

Question 5 Quelle est la proportion de nombres $< \frac{1}{2}$ renvoyée par ce générateur pour un grand nombre de tirages ?

Corrigé

Expérimentalement, on obtient une proportion très proche de $\frac{1}{2}$ (par exemple 0,5005 sur 10^6 tirages depuis le germe 1) : ce test unidimensionnel ne détecte pas le défaut du générateur. Le défaut apparaît sur les triplets : comme $65539 = 2^{16} + 3$, on a $65539^2 = 2^{32} + 6 \cdot 2^{16} + 9 \equiv 6 \times 65539 - 9 \pmod{2^{31}}$, donc

$$u_{k+2} \equiv 6u_{k+1} - 9u_k \pmod{2^{31}}$$

Ainsi $9f_k - 6f_{k+1} + f_{k+2}$ est toujours un entier, compris entre -5 et 9 : d'où les 15 plans de la figure.

À partir de maintenant, on va utiliser `srand` et `rand` donnés par la bibliothèque standard du C.

À noter qu'il y a de **meilleures** fonctions disponibles avec `srand48` et `lrand48`.

La fonction `rand` renvoie une valeur entre 0 et `RAND_MAX`, qui vaut sûrement $2^{31} - 1$ dans votre implémentation.

Question 6 Écrire une fonction `int randint(int a, int b)` qui renvoie un entier uniformément entre a et $b - 1$, ici, on supposera que $0 \leq a < b \leq \text{RAND_MAX}$.

Corrigé

La solution naïve $a + \text{rand}() \% (b - a)$ n'est pas exactement uniforme quand $b - a$ ne divise pas `RAND_MAX + 1` : les petits restes sont un peu plus fréquents. On rejette donc les tirages au-delà du plus grand multiple de $b - a$ (moins un) inférieur ou égal à `RAND_MAX`. En notant $N = \text{RAND_MAX} + 1$ et $k = b - a$, il y a $r = N \bmod k$ valeurs rejetées, avec $r \leq k - 1$ et $r \leq N - k$ (au moins un bloc de k valeurs est conservé), donc $r < \frac{N}{2}$: chaque tirage est accepté avec probabilité $> \frac{1}{2}$ et le nombre moyen de tirages est inférieur à 2. C'est un algorithme de Las Vegas.

```
int randint(int a, int b)
{
    int n = b - a;
    // (RAND_MAX % n + 1) % n vaut (RAND_MAX + 1) % n sans débordement
    int limite = RAND_MAX - (RAND_MAX % n + 1) % n;
    int r = rand();
    while (r > limite)
        r = rand();
    return a + r % n;
}
```

10.1.3 Karger contracte, Karger remplace

On va implémenter l'algorithme de Karger vu en classe.

Pour cela, on considère une représentation des graphes différente de celle qu'on a vue en classe.

```
struct edge {
    int x; // arête x -- y
    int y;
    int id; // identifiant de l'arête
};
typedef struct edge edge;

struct multigraph {
    edge *edges;
    int nedges;
};
typedef struct multigraph multigraph;
```

Un multigraphe est alors un tableau d'arêtes et on identifie un sommet à un entier. L'identifiant permettra de suivre l'arête après les différentes contractions. On pourra utiliser l'indice de l'arête dans le graphe de départ.

Question 7 Écrire une fonction de prototype

```
multigraph *contract(multigraph *g, edge e);
```

qui renvoie le multigraphe obtenu en contractant l'arête e . On va allouer et créer un nouveau multigraphe.

On pourra choisir $\min(x, y)$ pour le nouveau sommet.

Corrigé

On renomme le sommet $\max(x, y)$ en $\min(x, y)$ et on supprime les boucles ainsi créées (dont e elle-même).

```
multigraph *contract(multigraph *g, edge e)
{
    int garde = e.x < e.y ? e.x : e.y;
    int supprime = e.x < e.y ? e.y : e.x;
    multigraph *h = malloc(sizeof(multigraph));
    h->edges = malloc(sizeof(edge) * g->nedges);
    h->nedges = 0;
    for (int i = 0; i < g->nedges; i++) {
        edge f = g->edges[i];
        if (f.x == supprime) f.x = garde;
        if (f.y == supprime) f.y = garde;
        if (f.x != f.y) {
            h->edges[h->nedges] = f;
            h->nedges++;
        }
    }
    return h;
}
```

Question 8 Écrire une fonction de prototype

```
multigraph *karger_coupure(multigraph *g);
```

qui renvoie une coupure obtenue par une passe de l'algorithme de Karger.

Corrigé

On suppose le graphe connexe. Un multigraphe connexe sans boucle a exactement deux sommets si et seulement si toutes ses arêtes relient la même paire de sommets (avec au moins trois sommets, il y a au moins deux paires reliées), ce qui évite de maintenir le nombre de sommets.

```
bool deux_sommets(multigraph *g)
{
    for (int i = 1; i < g->nedges; i++) {
        edge f = g->edges[i];
        edge e = g->edges[0];
        if (!(f.x == e.x && f.y == e.y) || (f.x == e.y && f.y == e.x))
            return false;
    }
    return true;
}

multigraph *karger_coupure(multigraph *g)
{
    // copie de g pour ne pas le libérer
    multigraph *h = malloc(sizeof(multigraph));
    h->nedges = g->nedges;
    h->edges = malloc(sizeof(edge) * g->nedges);
    for (int i = 0; i < g->nedges; i++)
        h->edges[i] = g->edges[i];
    while (!deux_sommets(h)) {
        edge e = h->edges[randint(0, h->nedges)];
        multigraph *h2 = contract(h, e);
        free(h->edges);
        free(h);
        h = h2;
    }
    return h;
}

void free_multigraph(multigraph *g)
{
    free(g->edges);
    free(g);
}
```

Les champs `id` des arêtes restantes donnent les arêtes de la coupure dans le graphe de départ. Chaque contraction est en $O(m)$ et il y en a $n - 2$: une passe est en $O(nm)$ (plus le test `deux_sommets`, lui aussi en $O(m)$ par tour).

Question 9 Testez votre fonction sur le graphe exemple donné en classe. Estimez le nombre d'essais avant d'obtenir une coupure minimale.

On peut réaliser d'autres tests avec des recollements de deux graphes complets.

Corrigé

Avec deux graphes complets K_5 (sommets 0 à 4 et 5 à 9) reliés par les deux arêtes $\{0, 5\}$ et $\{1, 6\}$, la coupure minimale est formée de ces deux arêtes. Sur 10 000 passes, on l'obtient environ 2 600 fois, soit une probabilité d'environ 0,26 et donc en moyenne un peu moins de 4 essais (loi géométrique), bien mieux que la borne $\frac{2}{n(n-1)} = \frac{1}{45}$ du cours, qui donnerait 45 essais.

PARTIE II

Langages formels

11	Langages réguliers et automates finis	103
12	Fiche Langages réguliers et automates finis	137
13	Exercices Langages réguliers et automates finis	145
14	TP Berry-Sethi en C	153
15	Grammaires non contextuelles	159
16	TP Parser des expressions	173

Langages réguliers et automates finis

DANS CE CHAPITRE

- 11.1 Langages réguliers
- 11.2 Automates finis
- 11.3 Théorème de Kleene
- 11.4 Construction de l'automate de Glushkov, algorithme de Berry-Sethi
- 11.5 Conséquences du théorème de Kleene

11.1 Langages réguliers

11.1.1 Alphabet et mots

Définition 11.1

On considère un ensemble fini appelé *alphabet* et noté traditionnellement Σ . Ses éléments sont appelés des *lettres* ou des *symboles*.

Un mot sur Σ est une suite finie d'éléments de Σ . On note ϵ le mot vide. Un mot non vide $(a, b, c, \dots)$ sera juste noté $abc\dots$

On note Σ^* l'ensemble des mots.

Exemple 11.2

Si $\Sigma = \{a, b, \dots, z\}$ alors *bonjour* est un mot.

On utilisera souvent l'alphabet réduit à deux lettres : $\{a, b\}$. Il est particulièrement important dans le contexte du binaire.

Remarque 11.3

- On a donc $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ en posant $\Sigma^0 = \{\epsilon\}$.
- On note $|u|$ la longueur du mot u . On a donc $|\epsilon| = 0$.

Définition 11.4

On définit une loi de composition interne appelée concaténation ainsi :

$$\forall x_1 \dots x_n, y_1 \dots y_m \in \Sigma^*, x_1 \dots x_n \cdot y_1 \dots y_m = x_1 \dots x_n y_1 \dots y_m$$

Remarque 11.5

- ϵ est le neutre de la concaténation

- La concaténation est associative. On dit que $(\Sigma^*, .)$ est un **monoïde**.
- On omet le plus souvent le symbole de la concaténation. Dans le contexte où on mélange des mots et des lettres, cela revient à identifier les lettres et les mots à une lettre qui leur sont associées. Par exemple, si on écrit awb avec $a, b \in \Sigma$ et $w \in \Sigma^*$.
- $|uv| = |u| + |v|$.
- Si $\Sigma = \{\star\}$ l'alphabet à une lettre, alors $(\Sigma^*, .)$ est isomorphe à $(\mathbb{N}, +)$ en tant que monoïde.

Conséquence directe de la définition de Σ^* :

Théorème 11.6

Soient $w, w' \in \Sigma^*$.

$$w = w' \Leftrightarrow \{|w| = |w'| = n \\ w = w_1 \dots w_n, w' = w'_1 \dots w'_n \text{ et } \forall i \in \llbracket 1, n \rrbracket, w_i = w'_i\}$$

11.1.2 Suffixe, Préfixe, facteurs, sous-mots

Définition 11.7

Soit $u, v \in \Sigma^*$. On dit que v est un

- **suffixe** de u lorsqu'il existe $w \in \Sigma^*$ tel que $u = wv$. Quand de plus $w \neq \epsilon$, c'est-à-dire $v \neq u$, on dit que v est un suffixe propre de u .
- **préfixe** de u lorsqu'il existe $w \in \Sigma^*$ tel que $u = vw$. Quand de plus $w \neq \epsilon$, on dit que v est un préfixe propre de u .
- **facteur** de u lorsqu'il existe $w, w' \in \Sigma^*$ tels que $u = wvw'$. Quand de plus $v \neq u$, on dit que v est un facteur propre de u .
- **sous-mot** de $u = x_1 \dots x_n$, où $x_1, \dots, x_n \in \Sigma$, lorsqu'il existe des indices $1 \leq i_1 < i_2 < \dots < i_k \leq n$ tels que $v = x_{i_1} \dots x_{i_k}$. Pour $k = 0$, on obtient que ϵ est un sous-mot de tout mot.

Exemple 11.8

Si on considère le mot *triskaidekaphobia*,

- *tri* est un préfixe propre
- *phobia* est un suffixe propre
- *aide* est un facteur propre
- *trap* est un sous-mot

11.1.3 Langage

Définition 11.9

Un **langage** sur l'alphabet Σ est une partie de Σ^* .

Exemple 11.10

Sur l'alphabet $\{a, b\}$, les ensembles suivants sont des langages :

- les mots ne contenant que des a : $\{a^n \mid n \in \mathbb{N}\}$
- les mots ayant le même nombre de a que de b
- les palindromes, c'est-à-dire les mots pouvant se lire de la même manière de gauche à droite et de droite à gauche

Remarque 11.11 Les langages peuvent être faciles à définir mais durs à expliciter. Il existe ainsi un langage contenant tous les énoncés que vous aurez aux concours, mais ce sera difficile de le déterminer avant les épreuves !

Définition 11.12

Soient L, L' des langages sur un même alphabet.

On note

- $L + L' = L \cup L'$ le **langage union** de ces langages.
- $LL' = \{uv \mid u \in L, v \in L'\}$ la **concaténée** de ces langages.
- $L^n = L \dots L$ la concaténée de n copies de L et $L^0 = \{\epsilon\}$.
- $L^* = \bigcup_{n \in \mathbb{N}} L^n$ appelée **l'étoile de Kleene** de L .

Théorème 11.13

L^* est clos par concaténation : $\forall w, w' \in L^*, ww' \in L^*$.

Preuve Soit $w, w' \in L^*$. Il existe n, m tels que $w = w_1 \dots w_n \in L^n$ et $w' = w'_1 \dots w'_m \in L^m$, on a donc $ww' = w_1 \dots w_n w'_1 \dots w'_m \in L^{n+m} \subset L^*$. □

Exemple 11.14

$$\emptyset^* = \{\epsilon\}^* = \{\epsilon\}$$

Remarque 11.15 On peut démontrer de nombreuses relations sur les langages :

- $\cup$ est associative, commutative, idempotente et d'élément neutre $\emptyset$.
- la concaténation est associative et d'élément neutre $\{\epsilon\}$ mais non commutative dès que Σ contient au moins deux lettres. Elle distribue sur l'union.
- L'étoile est idempotente : $(L^*)^* = L^*$.

Preuve On a $L^* = (L^*)^1 \subset (L^*)^*$. Réciproquement, soit $w \in (L^*)^*$: il existe $n \in \mathbb{N}$ et $w_1, \dots, w_n \in L^*$ tels que $w = w_1 \dots w_n$. On montre par récurrence sur n que $w_1 \dots w_n \in L^*$: pour $n = 0$, c'est le mot vide, qui est dans $L^0 \subset L^*$, et le passage de n à $n + 1$ découle de la clôture de L^* par concaténation. Ainsi, on a l'égalité par double inclusion. □

11.1.4 Langages réguliers

Définition 11.16

L'ensemble $\text{Reg}(\Sigma)$ des langages réguliers sur Σ est défini inductivement ainsi :

- $\emptyset \in \text{Reg}(\Sigma)$
- $\forall a \in \Sigma, \{a\} \in \text{Reg}(\Sigma)$
- $\forall A \in \text{Reg}(\Sigma), A^* \in \text{Reg}(\Sigma)$
- $\forall A, B \in \text{Reg}(\Sigma), A + B, AB \in \text{Reg}(\Sigma)$

Remarque 11.17

- Autrement dit, $\text{Reg}(\Sigma)$ est le plus petit ensemble contenant $\emptyset$ et les langages réduits à une lettre qui soit stable par concaténation, union et étoile de Kleene.
- Les langages réguliers correspondent à la classe des langages facilement reconnaissables, comme on le verra dans la partie suivante, tout en étant suffisamment expressifs.

Exemple 11.18

Soit L le langage des mots sur $\{a, b\}$ sans facteurs aa ou bb .

Il s'agit donc des mots qui alternent les lettres.

L est régulier car on peut l'écrire :

$$L = (\{b\} + \{\epsilon\})\{ab\}^*(\{a\} + \{\epsilon\})$$

où $\{\epsilon\} = \emptyset^*$ et $\{ab\} = \{a\}\{b\}$ sont bien réguliers. En effet, un tel mot est de quatre formes selon qu'il commence ou pas par un b et qu'il finisse ou pas par un a .

11.1.5 Expression régulière

On a pu voir dans le dernier exemple que la notation d'un langage régulier à l'aide de la définition inductive est un peu lourde. Afin d'améliorer cela, on va introduire une dénotation particulière des langages réguliers : les expressions régulières.

Définition 11.19

Les expressions régulières sur Σ sont des arbres d'expressions dont les feuilles sont étiquetées par $\{\emptyset, \epsilon\} \cup \Sigma$ et ayant des nœuds binaires étiquetés par $\{ |, \cdot \}$ et des nœuds unaires étiquetés par $\{ * \}$.

On adopte la notation infixe usuelle pour ces expressions en plaçant $*$ en opérateur unaire postfixe et en omettant de noter $\cdot$.

Exemple 11.20

Les expressions suivantes sont des expressions régulières sur $\{a, b\}$:

$$a, b, ab, a | b, (a | b)a^*, \emptyset b$$

Remarque 11.21 On suppose que les opérateurs ont des précédences dans l'ordre $* > . > |$ et ainsi, on pourra omettre certaines parenthèses : $(a(b^*)) | c = ab^* | c$.

Définition 11.22

On associe à toute expression régulière e sur Σ un langage $\text{lang}(e)$:

- $\text{lang}(\emptyset) = \emptyset$
- $\text{lang}(\epsilon) = \{\epsilon\}$
- $\forall a \in \Sigma, \text{lang}(a) = \{a\}$.
- $\text{lang}(ee') = \text{lang}(e) \text{ lang}(e')$
- $\text{lang}(e | e') = \text{lang}(e) + \text{lang}(e')$
- $\text{lang}(e^*) = \text{lang}(e)^*$

On dit que e est une dénotation du langage $\text{lang}(e)$.

La définition calquant celle des langages réguliers, on a le théorème suivant, qui se prouve par une induction immédiate dans chaque sens (le seul cas qui n'est pas une traduction directe est $\text{lang}(\epsilon) = \{\epsilon\}$, qui est régulier car $\{\epsilon\} = \emptyset^*$) :

Théorème 11.23

Les langages admettant une dénotation par une expression régulière sont exactement les langages réguliers.

Remarque 11.24 Un langage régulier n'admet pas une unique dénotation. Par exemple, $\{a\}^*$ peut se noter a^* mais également $aa^* | \epsilon$

Afin de simplifier les expressions régulières, on est amené à identifier deux expressions régulières dénotant un même langage. Ainsi, on écrira directement $(a \mid b)c = ac \mid bc$.

11.1.6 Expressions régulières étendues POSIX

On distingue trois types d'expressions régulières dans les outils Unix :

- les expressions simples acceptées par un shell
- les expressions régulières POSIX
- les expressions régulières POSIX étendues.

On présente ici les expressions étendues. Elles sont composées :

- de caractères, pour ceux qui sont utilisés dans les définitions suivantes, on les *échappe* avec un $\backslash$ comme dans $\backslash.$ ou $\backslash*$
- de listes de caractères sous la forme $[abc]$ pour représenter $a \mid b \mid c$. On peut aussi utiliser $a-z$ pour indiquer tous les caractères de a à z .
- de listes niées de la forme $[\^{\dots}]$ où on considère tous les caractères sauf ceux qui sont indiqués.
- le caractère $.$ représente n'importe quel caractère.
- le caractère $^$ est un caractère spécial représentant le début d'une chaîne. De même, $\$$ représente la fin.
- de parenthèses pour regrouper des sous-expressions
- des opérateurs $*$ et $|$ comme dans la définition des expressions régulières

On a également des ajouts pratiques :

- $+$ et $?$ qui sont des sucres syntaxiques pour les définitions suivantes : $e+ = ee^*$ et $e? = (e|)$. Il n'y a pas de ϵ , on se contente d'utiliser l'expression vide comme ici.
- pour $m, n \in \mathbb{N}$ et e une expression on peut écrire :
 - $e\{n\}$ pour exactement n copies de e
 - $e\{m, \}$ pour au moins m copies de e
 - $e\{, n\}$ pour au plus n copies de e
 - $e\{m, n\}$ pour entre m et n copies de e

Remarque 11.25 Il s'agit ici aussi de sucres. En effet, on pourrait écrire :

- $e\{3\} = eee$
- $e\{3, \} = eeee^*$
- $e\{, 3\} = (e|ee|eee)$
- $e\{2, 3\} = (ee|eee)$

11.2 Automates finis

On va présenter ici un modèle de calcul permettant de reconnaître des langages réguliers.

11.2.1 Automates finis déterministes

Définition 11.26

Un **automate fini déterministe** est un quintuplet $A = (Q, \Sigma, \delta, q_i, F)$ où

- Σ est un alphabet fini
- Q est un ensemble fini appelé l'ensemble des **états** de l'automate
- $q_i \in Q$ est appelé l'**état initial**
- $F \subset Q$ est appelé l'ensemble des **états finaux**
- δ est une application partielle de $Q \times \Sigma$ dans Q et est appelée la **fonction de transition** de l'automate.

Remarque 11.27 Une application partielle de E dans F est une application $E' \rightarrow F$ où $E' \subset E$ est le domaine de l'application.

Quand le domaine de δ est $Q \times \Sigma$, on dit que l'automate est complet.

Exemple 11.28

Considérons l'automate fini $A = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_3\})$ où :

$$\delta(q_0, a) = q_1, \delta(q_0, b) = q_2$$

$$\delta(q_1, a) = q_1, \delta(q_1, b) = q_3$$

$$\delta(q_2, a) = q_3, \delta(q_2, b) = q_2$$

et $\delta(q_3, a), \delta(q_3, b)$ ne sont pas définies.

On peut représenter δ sous forme tabulaire, on parle alors de **table de transition**. L'état initial est indiqué par une flèche et les états finaux par un astérisque.

	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_3
q_2	q_3	q_2
$* q_3$		

On adopte une notation graphique pour présenter les automates. Les états sont des nœuds et les arêtes permettent de représenter δ . Les états finaux sont indiqués par une flèche sortante ou un double trait comme ici :

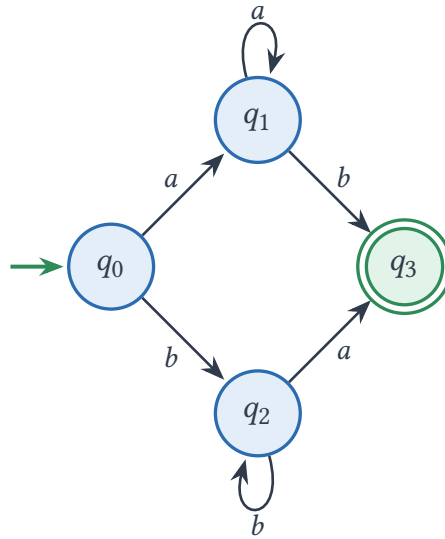


Figure 11.1 — Automate déterministe : état initial q_0 (flèche entrante), état final q_3 (double cercle)

11.2.2 Accessibilité, Co-accessibilité

On peut préciser la notation précédente d'un automate sous la forme d'un graphe. Comme on peut avoir deux transitions $\delta(q, a) = \delta(q, b) = q'$ avec $a \neq b$ ainsi que des boucles, le graphe n'est pas simple et parler d'ensemble de couples de sommets pour les arêtes ne suffit plus.

Définition 11.29

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate fini.

On pose $E_A \subset Q \times \Sigma \times Q$ défini par $(q, a, q') \in E_A \Leftrightarrow \delta(q, a) = q'$.

Le couple $G_A = (Q, E_A)$ est le graphe, dont les arêtes sont étiquetées, associé à l'automate A .

Un **chemin** de q à q' dans A est une suite de transitions $q = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n = q'$ avec $(q_{k-1}, a_k, q_k) \in E_A$ pour tout $k \in \llbracket 1, n \rrbracket$. Le mot $a_1 \dots a_n$ est appelé l'**étiquette** de ce chemin. Pour $n = 0$, on obtient le chemin vide de q à q , d'étiquette ϵ .

On note $q \rightsquigarrow q'$ lorsqu'il existe un chemin de q à q' .

Grâce à ce graphe, on peut utiliser des notions de chemins entre états :

Définition 11.30

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate fini et $q \in Q$.

On dit que q est

- **accessible** lorsque $q_i \rightsquigarrow q$.
- **co-accessible** lorsqu'il existe $q_f \in F$ tel que $q \rightsquigarrow q_f$.

Quand tous les états d'un automate sont accessibles et co-accessibles, on dit que l'automate est **émondé**.

11.2.3 Langage reconnu par un automate fini

Définition 11.31

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate fini.

On dit que $m \in \Sigma^*$ est **reconnu** par A si c'est l'étiquette d'un chemin de q_i à un état final.

On appelle **langage reconnu** par l'automate l'ensemble des mots reconnus, on le note $\text{lang}(A)$.

Exemple 11.32

On reprend l'exemple précédent d'automate. Pour qu'il reconnaisse un mot, on voit que celui-ci doit être soit de la forme $a^n b$ soit de la forme $b^n a$ avec $n > 0$.

On peut décrire précisément ce langage par l'expression régulière $aa^*b \mid bb^*a$.

Théorème 11.33

Étant donné un automate fini, il est possible d'en déduire un automate complet et, si ce langage est non vide, un automate émondé ayant tous les trois le même langage reconnu.

Preuve Complétion. On rajoute un état supplémentaire, communément appelé *état puits*, $q_p \notin Q$ et on pose $A' = (Q \cup \{q_p\}, \Sigma, \delta', q_i, F)$ avec, pour $q \in Q \cup \{q_p\}$ et $a \in \Sigma$, $\delta'(q, a) = \delta(q, a)$ si la transition était définie et $\delta'(q, a) = q_p$ sinon. En particulier, $\delta'(q_p, a) = q_p$. L'automate A' est complet. Tout chemin de A est un chemin de A' , donc $\text{lang}(A) \subset \text{lang}(A')$. Réciproquement, un chemin de A' qui passe par q_p ne peut plus en sortir, et $q_p \notin F$: un chemin de q_i à un état final dans A' ne passe donc pas par q_p , c'est un chemin de A .

Émondage. Tout état d'un chemin de q_i à un état final est accessible (on suit le début du chemin) et co-accessible (on suit sa fin). Si on supprime les états qui ne sont pas accessibles ou pas co-accessibles, ainsi que les transitions qui les concernent, on conserve donc tous les chemins de q_i à un état final, et le langage reconnu est inchangé. Si ce langage est non vide, il existe un tel chemin, donc q_i est conservé et on obtient bien un automate. L'automate obtenu est émondé : un état conservé a un chemin depuis q_i et un chemin vers un état final, et tous les états de ces chemins sont eux-mêmes accessibles et co-accessibles, donc conservés.

□

11.2.4 Fonction de transition étendue

Il est souvent utile de raisonner directement sur des mots pour effectuer plusieurs transitions d'affilée. On introduit alors la notion suivante :

Définition 11.34

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate fini déterministe, on définit sa **fonction de transition étendue** δ^* : $Q \times \Sigma^* \rightarrow Q$ par récurrence sur la longueur des mots ainsi :

$$\forall q \in Q, \delta^*(q, \epsilon) = q$$

$$\forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

Comme δ , l'application δ^* est partielle : $\delta^*(q, wa)$ n'est définie que si $\delta^*(q, w)$ est définie et que δ est définie en $(\delta^*(q, w), a)$.

Remarque 11.35 $\delta^*(q, w)$ est défini si et seulement s'il existe un chemin d'étiquette w partant de q , et c'est alors l'état d'arrivée de ce chemin, qui est unique car l'automate est déterministe. On a ainsi

$$\text{lang}(A) = \{w \in \Sigma^* \mid \delta^*(q_i, w) \in F\}$$

11.2.5 Modélisation par des automates

On vient de voir que les automates finis permettent de caractériser certains langages, mais ils permettent également de modéliser des systèmes à plusieurs états.

Exemple 11.36

On considère un distributeur de boissons qui permet de distribuer des espressos ou des cafés longs. Un expresso coûte 1€ et un long coûte 2€. On considère qu'on peut mettre des pièces de 50c, 1€ ou 2€. On souhaite modéliser l'échange amenant à valider la quantité d'argent insérée. On suppose que la boisson qui sera distribuée est stockée ailleurs que dans un état, donc un état final signifiera uniquement que le bon montant a été inséré.

On ignore également toutes les pièces dépassant le montant dû, on pourrait ainsi rajouter des boucles sur les états avec ces valeurs. On les omet dans le dessin suivant pour être plus lisible :

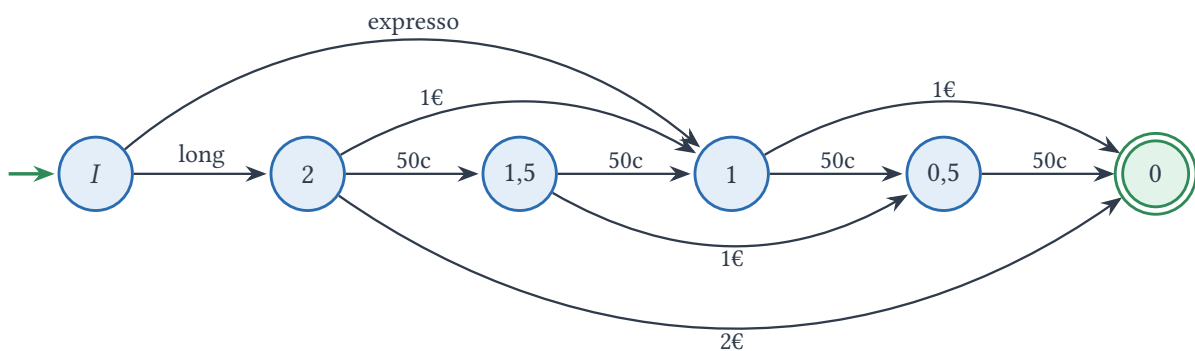


Figure 11.2 — Automate du distributeur de boissons : l'état indique la somme restant due

Les mots reconnus sont alors :

- * long-2€
- * long-1€-1€
- * long-1€-50c-50c
- * long-50c-1€-50c
- * long-50c-50c-1€
- * long-50c-50c-50c-50c

* espresso-1€
* espresso-50c-50c

Ce qui correspond exactement au comportement souhaité.

Remarque 11.37 On a parlé de mots, mais on parle en général de **traces d'exécution** qui permettent de s'assurer que le comportement est celui que l'on souhaite.

11.2.6 Automates non déterministes

On va modifier la définition afin de donner plus de souplesse et donc *a priori* plus de pouvoir expressif aux automates.

Définition 11.38

Un **automate fini non déterministe** est un quintuplet $A = (Q, \Sigma, \delta, I, F)$ où

- Σ est un alphabet fini
- Q est un ensemble fini appelé l'ensemble des **états** de l'automate
- $I \subset Q$ est appelé l'ensemble des **états initiaux**
- $F \subset Q$ est appelé l'ensemble des **états finaux**
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ est appelée la **fonction de transition** de l'automate.

Remarque 11.39 Dans cette définition, les seuls changements par rapport aux automates déterministes sont la présence de plusieurs états initiaux et le fait qu'on ait des transitions vers 0, 1 ou plus d'états.

Si $\delta(q, a) = \emptyset$ c'est équivalent à ne pas avoir défini la transition dans l'application partielle d'un automate déterministe. La nouveauté est donc dans le fait de pouvoir avoir plusieurs états d'arrivée.

On représente naturellement ces automates en rajoutant des flèches étiquetées par a depuis un état q vers chaque état de $\delta(q, a)$.

On définit de même la fonction de transition étendue d'un automate non-déterministe.

Définition 11.40

Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini non déterministe, on définit sa **fonction de transition étendue** $\delta^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ par récurrence sur la longueur des mots ainsi :

$$\forall q \in Q, \delta^*(q, \epsilon) = \{q\}$$

$$\forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, wa) = \bigcup_{q' \in \delta^*(q, w)} \delta(q', a)$$

Définition 11.41

Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini non-déterministe, on appelle *langage reconnu* par A l'ensemble

$$\text{lang}(A) = \bigcup_{q_i \in I} \{w \in \Sigma^* \mid \delta^*(q_i, w) \cap F \neq \emptyset\}$$

Autrement dit, w est reconnu lorsqu'il existe **au moins un** chemin d'étiquette w d'un état initial à un état final.

Exemple 11.42

L'automate non-déterministe suivant reconnaît les mots dont l'avant-dernière lettre est un a .

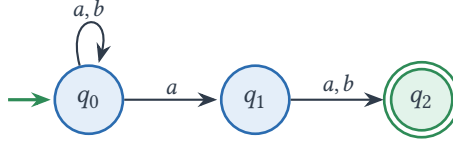


Figure 11.3 — Automate non déterministe des mots dont l'avant-dernière lettre est un a

On voit qu'il y a un choix non-déterministe ici qui permet de deviner la fin d'un mot.

Théorème 11.43

Soit A un automate fini non-déterministe, il existe un automate fini déterministe A' reconnaissant le même langage que A .

De plus, si A contient n états, alors A' contient au plus 2^n états.

Preuve On va, en fait, présenter une méthode permettant de **déterminiser** un automate fini. Cette construction est appelée **l'automate des parties**.

Si $A = (Q, \Sigma, \delta, I, F)$ on pose $A' = (\mathcal{P}(Q), \Sigma, \delta', I, F')$ où

$$F' = \{p \subset Q \mid p \cap F \neq \emptyset\}$$

et, pour $p \subset Q$ et $a \in \Sigma$,

$$\delta'(p, a) = \bigcup_{q \in p} \delta(q, a)$$

On peut émonder l'automate pour diminuer son nombre d'états, mais on voit tout de suite qu'il en contient au plus $2^{|Q|}$.

Pour conclure, on doit montrer que $\text{lang}(A) = \text{lang}(A')$. Pour cela, on va montrer le résultat suivant par récurrence sur $|w|$:

$$\forall w \in \Sigma^*, \bigcup_{q_i \in I} \delta^*(q_i, w) = \delta'^*(I, w)$$

- **Initialisation** : $\bigcup_{q_i \in I} \delta^*(q_i, \epsilon) = \bigcup_{q_i \in I} \{q_i\} = I = \delta'^*(I, \epsilon)$
- **Hérédité** : si la propriété est vraie pour des mots de longueur n , soit $wa \in \Sigma^*$ où $a \in \Sigma$ avec $|w| = n$.

On a

$$\begin{aligned} \bigcup_{q_i \in I} \delta^*(q_i, wa) &= \bigcup_{q_i \in I} \bigcup_{q \in \delta^*(q_i, w)} \delta(q, a) \\ &= \bigcup_{q \in \bigcup_{q_i \in I} \delta^*(q_i, w)} \delta(q, a) \\ &= \bigcup_{q \in \delta'^*(I, w)} \delta(q, a) \\ &= \delta'(\delta'^*(I, w), a) \\ &= \delta'^*(I, wa) \end{aligned}$$

Ainsi, on a l'égalité voulue et donc

$$\begin{aligned}
\text{lang}(A) &= \bigcup_{q_i \in I} \{w \in \Sigma^* \mid \delta^*(q_i, w) \cap F \neq \emptyset\} \\
&= \left\{ w \in \Sigma^* \mid \bigcup_{q_i \in I} \delta^*(q_i, w) \cap F \neq \emptyset \right\} \\
&= \{w \in \Sigma^* \mid \delta'^*(I, w) \cap F \neq \emptyset\} \\
&= \{w \in \Sigma^* \mid \delta'^*(I, w) \in F'\} \\
&= \text{lang}(A')
\end{aligned}$$

□

Exemple 11.44

Pour réaliser ce procédé de déterminisation en pratique, on établit une table de transition en partant de I et en rajoutant une ligne pour chaque nouvel état rencontré. Ce faisant, on obtient un automate partiellement émondé.

Si on reprend l'exemple précédent, on va définir la table suivante :

		a	b
$\rightarrow$	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
*	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
*	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

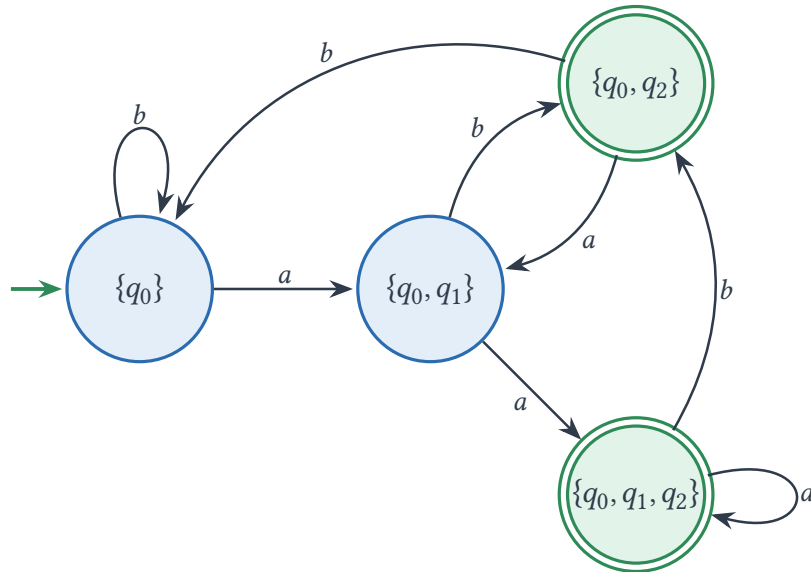


Figure 11.4 — Déterminisé de l'automate des mots dont l'avant-dernière lettre est un a

On remarque avec cette construction qu'on a deux états acceptants qui correspondent aux suffixes $\$aa\$$ et $\$ab\$$. Plus généralement, l'état atteint ne dépend que des deux dernières lettres lues.

❏ Réciproquement, un automate déterministe $(Q, \Sigma, \delta, q_i, F)$ peut être vu comme l'automate non-déterministe $(Q, \Sigma, \delta', \{q_i\}, F)$ en posant $\delta'(q, a) = \{\delta(q, a)\}$ quand $\delta(q, a)$ est défini et $\delta'(q, a) = \emptyset$ sinon. On a alors le corollaire :

Corollaire 11.45

Les langages reconnus par des automates déterministes sont exactement les langages reconnus par des automates non-déterministes.

11.2.7 Transitions spontanées

On peut également étendre les automates non-déterministes en rajoutant des transitions spontanées, également appelées ϵ -transitions, entre les états en considérant une fonction de transition prenant également le mot vide en entrée. Comme on va le voir, cela permet de n'avoir qu'un seul état initial.

Définition 11.46

Un **automate fini non-déterministe à transitions spontanées** est un quintuplet $A = (Q, \Sigma, \delta, q_i, F)$ où

- Σ est un alphabet fini
- Q est un ensemble fini appelé l'ensemble des **états** de l'automate
- $q_i \in Q$ est appelé l'**état initial**
- $F \subset Q$ est appelé l'ensemble des **états finaux**
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ est appelée la **fonction de transition**.

Exemple 11.47

L'automate suivant, d'état initial q_0 et d'état final q_2 , présente des transitions spontanées indiquées par ϵ :

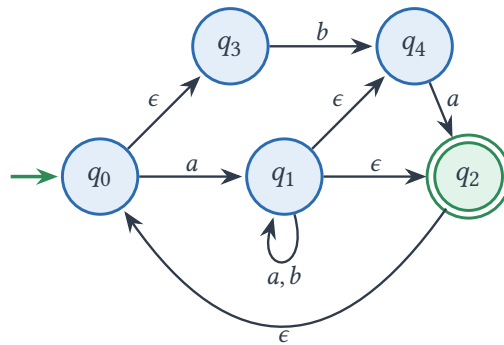


Figure 11.5 — Automate avec transitions spontanées, d'état initial q_0 et d'état final q_2

Définition 11.48

Soit A un automate fini non-déterministe possédant des transitions spontanées et $q \in Q$. On définit l'ensemble $\bar{q}$, appelé **clôture** de q , comme le plus petit ensemble tel que $q \in \bar{q}$ et

$$\forall q' \in \bar{q}, \delta(q', \epsilon) \subset \bar{q}$$

Si $E \subset Q$, on pose $\bar{E} = \bigcup_{q \in E} \bar{q}$.

Remarque 11.49 $\bar{q}$ est donc l'ensemble des états accessibles dans le sous-graphe des états obtenus en ne conservant que les transitions spontanées.

Cette complétion est nécessaire pour comprendre l'action de l'automate sur les états dans la mesure où $q' \in \delta(q, a)$ signifie qu'on peut également accéder aux états accessibles depuis q' par transition spontanée.

On remarque qu'on a $\overline{E \cup E'} = \bar{E} \cup \bar{E'}$ et $\bar{\bar{E}} = \bar{E}$.

Exemple 11.50

Dans l'automate précédent, on a $\bar{q}_0 = \{q_0, q_3\}$, $\bar{q}_2 = \{q_0, q_2, q_3\}$ et $\bar{q}_1 = \{q_0, q_1, q_2, q_3, q_4\}$.

On peut alors définir la fonction de transition étendue d'un tel automate ainsi :

$$\forall q \in Q, \delta^*(q, \epsilon) = \bar{q}$$

$$\forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, wa) = \overline{\bigcup_{q' \in \delta^*(q, w)} \delta(q', a)}$$

et on définit alors le langage reconnu par un tel automate ainsi :

$$\text{lang}(A) = \{w \in \Sigma^* \mid \delta^*(q_i, w) \cap F \neq \emptyset\}$$

Théorème 11.51

Soit A un automate fini non-déterministe possédant des transitions spontanées. Il existe un automate non-déterministe A' tel que $\text{lang}(A) = \text{lang}(A')$.

Preuve On pose $A' = (Q, \Sigma, \delta', \bar{q}_i, F)$ où

$$\delta'(q, a) = \overline{\delta(q, a)}$$

On va montrer par récurrence sur $|w|$ que

$$\forall w \in \Sigma^*, \delta^*(q_i, w) = \bigcup_{q \in \bar{q}_i} \delta'^*(q, w)$$

- *Initialisation* $\delta^*(q_i, \epsilon) = \bar{q}_i = \bigcup_{q \in \bar{q}_i} \{q\} = \bigcup_{q \in \bar{q}_i} \delta'^*(q, \epsilon)$
- *Hérédité* supposons la propriété vraie pour les mots de longueur n et soit $wa \in \Sigma^*$ où $|w| = n$ et $a \in \Sigma$. On a, en utilisant $\overline{E \cup E'} = \bar{E} \cap \bar{E'}$ puis l'hypothèse de récurrence :

$$\begin{aligned} \delta^*(q_i, wa) &= \overline{\bigcup_{q' \in \delta^*(q_i, w)} \delta(q', a)} \\ &= \overline{\bigcup_{q' \in \delta^*(q_i, w)} \delta(q', a)} \\ &= \bigcup_{q' \in \bigcup_{q \in \bar{q}_i} \delta'^*(q, w)} \delta'(q', a) \\ &= \bigcup_{q \in \bar{q}_i} \bigcup_{q' \in \delta'^*(q, w)} \delta'(q', a) \\ &= \bigcup_{q \in \bar{q}_i} \delta'^*(q, wa) \end{aligned}$$

On peut alors conclure sur les langages reconnus :

$$\begin{aligned} \text{lang}(A) &= \{w \in \Sigma^* \mid \delta^*(q_i, w) \cap F \neq \emptyset\} \\ &= \left\{ w \in \Sigma^* \mid \bigcup_{q \in \bar{q}_i} \delta'^*(q, w) \cap F \neq \emptyset \right\} \\ &= \bigcup_{q \in \bar{q}_i} \{w \in \Sigma^* \mid \delta'^*(q, w) \cap F \neq \emptyset\} \\ &= \text{lang}(A') \end{aligned}$$

□

Exemple 11.52

On va éliminer les transitions spontanées de l'automate précédemment donné en exemple. Pour cela, on peut commencer par établir une table complétée :

q	c	$\delta(q, c)$	$\overline{\delta(q, c)}$
q_0	a	$\{q_1\}$	$\{q_0, q_1, q_2, q_3, q_4\}$
q_1	a	$\{q_1\}$	$\{q_0, q_1, q_2, q_3, q_4\}$
q_1	b	$\{q_1\}$	$\{q_0, q_1, q_2, q_3, q_4\}$
q_3	b	$\{q_4\}$	$\{q_4\}$
q_4	a	$\{q_2\}$	$\{q_0, q_2, q_3\}$

Les états initiaux sont ceux de $\overline{q_0} = \{q_0, q_3\}$ et l'unique état final reste q_2 .

On a donc l'automate non-déterministe équivalent suivant :

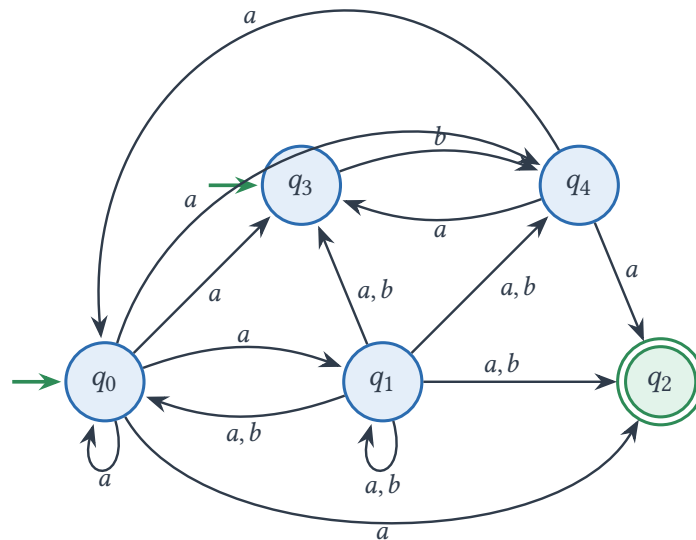


Figure 11.6 — Automate non déterministe équivalent, sans transition spontanée

On remarque donc que cette élimination des transitions spontanées ne change pas les sommets mais uniquement les arêtes du graphe associé à l'automate en rajoutant par complétion.

Remarque 11.53 Pour passer d'un automate non déterministe à un automate non déterministe à transitions spontanées, il suffit de rajouter un nouvel état initial et des transitions spontanées vers les anciens états initiaux.

On a donc également le corollaire suivant en composant les théorèmes précédents.

Corollaire 11.54

Les langages reconnus par un automate fini non-déterministe possédant des transitions spontanées sont exactement les langages reconnus par un automate fini déterministe.

Exemple 11.55

En fait, il est possible de déterminer directement un automate avec transitions spontanées.

On va déterminer l'automate présenté au dessus en utilisant là aussi la méthode de la table de transition construite à la volée. Il faut juste bien faire attention à prendre la clôture des états obtenus, en partant de $\overline{q_0}$. Afin de simplifier la représentation, on a attribué une lettre aux états.

			a	b
$\rightarrow$	A	$\{q_0, q_3\}$	$\{q_0, q_1, q_2, q_3, q_4\}$	$\{q_4\}$
$*$	B	$\{q_0, q_1, q_2, q_3, q_4\}$	$\{q_0, q_1, q_2, q_3, q_4\}$	$\{q_0, q_1, q_2, q_3, q_4\}$
	C	$\{q_4\}$	$\{q_0, q_2, q_3\}$	
$*$	D	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3, q_4\}$	$\{q_4\}$

Cet automate peut se représenter graphiquement ainsi :

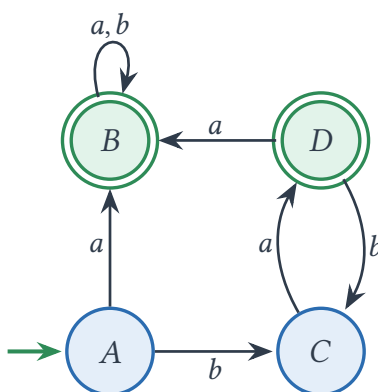


Figure 11.7 – Déterminisé de l'automate avec transitions spontanées

11.3 Théorème de Kleene

Dans cette partie, on va démontrer le théorème de Kleene :

Théorème 11.56

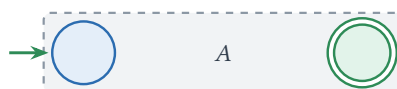
(Kleene) Les langages réguliers sont exactement les langages reconnaissables par un automate fini déterministe.

La preuve de ce théorème est suffisamment complexe pour qu'on la sépare en deux résultats. On va montrer ici comment passer d'une expression régulière à un automate, et réciproquement. Dans les deux sens, on dispose de constructions inefficaces, faciles à comprendre, et d'algorithmes efficaces mais plus délicats.

11.3.1 Des expressions régulières aux automates non-déterministes avec transitions spontanées

On commence ici par montrer une construction très simple d'un automate non-déterministe à transitions spontanées associé à une expression régulière par induction sur l'expression.

On note $A(e)$ l'**automate de Thompson** associé à l'expression e , il va s'agir dans tous les cas d'un automate à un état initial et un état final. Un tel automate aura donc la forme suivante :



• $A(\epsilon) =$

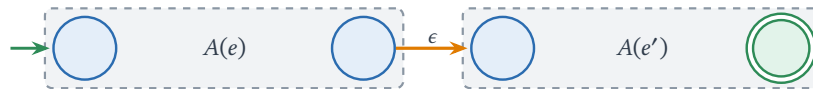


• $A(\emptyset) =$

- $\forall a \in \Sigma, A(a) =$



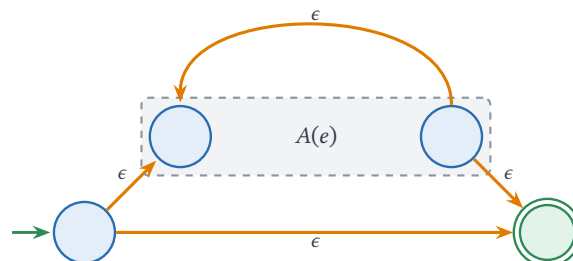
- $A(ee') =$



- $A(e \mid e') =$



- $A(e^*) =$



On peut alors passer à un automate déterministe avec la méthode de l'automate des parties.

Exemple 11.57

On considère l'expression régulière $(a \mid b)^*b(a \mid b)$.

L'automate de Thompson associé est donc :

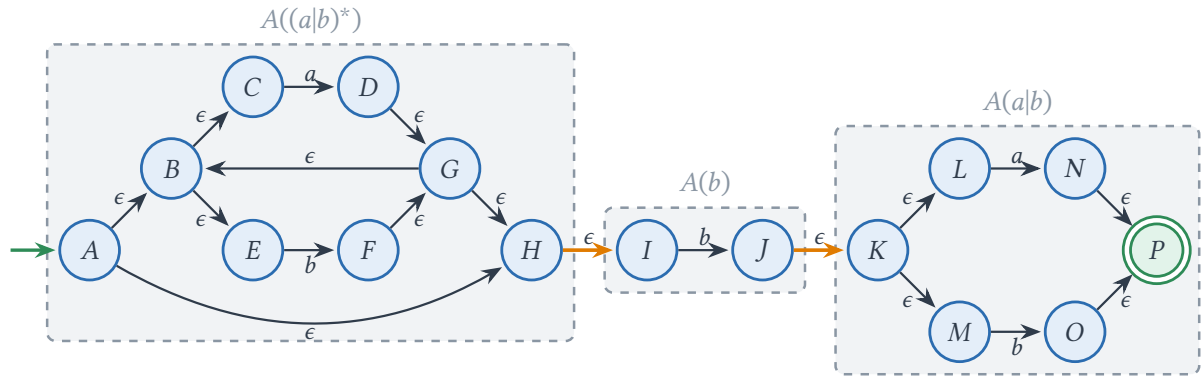


Figure 11.8 — Automate de Thompson de $(a | b)^*b(a | b)$; en orange, les transitions de concaténation

Si on retire les transitions spontanées, on obtient l'automate :

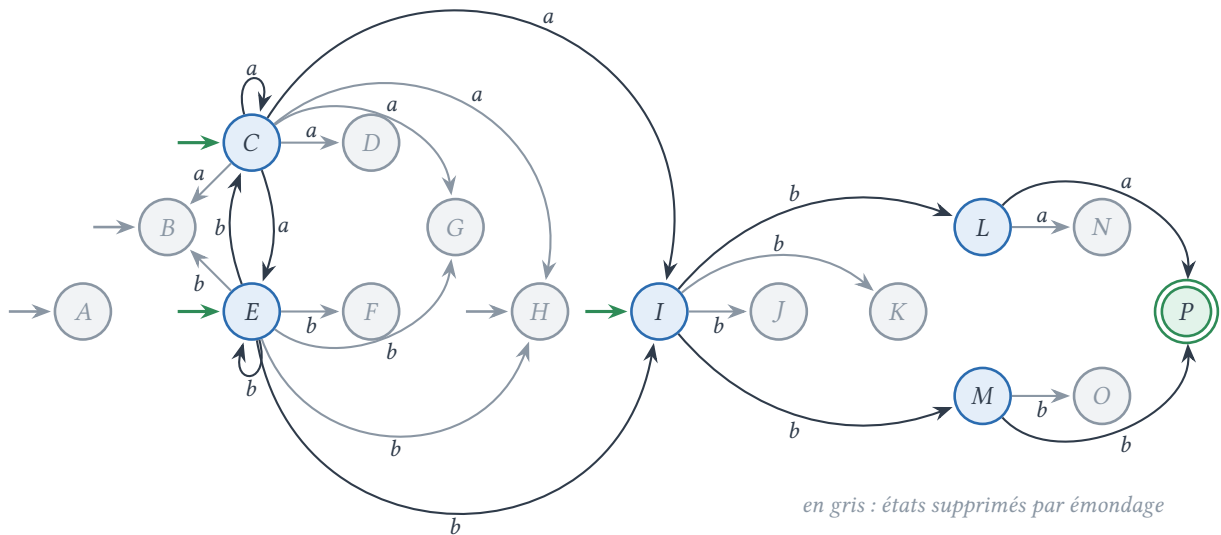


Figure 11.9 — Après suppression des transitions spontanées ; en gris, les états supprimés par émondage

qui en émondant les états qui ne sont pas accessibles ou pas co-accessibles devient :

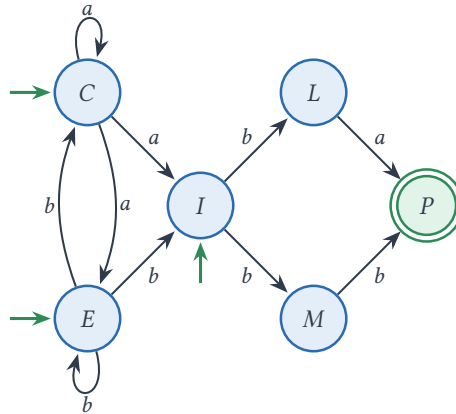
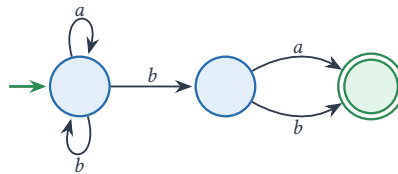


Figure 11.10 — Automate émondé

Notons qu'il est possible d'obtenir directement un automate comportant moins d'états :

Figure 11.11 — Un automate plus petit pour $(a | b)^* b(a | b)$

On a alors le résultat suivant :

Théorème 11.58

Pour toute expression régulière e , on a $\text{lang}(e) = \text{lang}(A(e))$.

Preuve Dans chaque construction, les états ajoutés sont nouveaux, et $\text{lang}(A(e))$ est l'ensemble des étiquettes des chemins de l'état initial i à l'état final f de $A(e)$, les transitions spontanées n'ajoutant rien à l'étiquette. On procède par induction sur e .

- Pour ϵ , $\emptyset$ et $a \in \Sigma$, les seuls chemins de i à f sont respectivement d'étiquette ϵ , inexistants, et d'étiquette a .
- Pour ee' : il n'y a aucune transition de $A(e')$ vers $A(e)$, donc un chemin de l'état initial à l'état final emprunte exactement une fois la transition spontanée qui relie l'état final de $A(e)$ à l'état initial de $A(e')$. Il se décompose en un chemin réussi de $A(e)$ suivi d'un chemin réussi de $A(e')$, et réciproquement toute telle concaténation est un chemin réussi. Par hypothèse d'induction, le langage reconnu est donc $\text{lang}(e) \text{ lang}(e')$.
- Pour $e | e'$: un chemin réussi commence par une transition spontanée vers l'état initial de $A(e)$ ou de $A(e')$, et comme il n'y a aucune transition entre $A(e)$ et $A(e')$, il reste dans cet automate jusqu'à son état final, puis rejoint le nouvel état final. Le langage reconnu est donc $\text{lang}(e) \cup \text{lang}(e')$.
- Pour e^* : un chemin réussi est soit la transition spontanée directe entre les deux nouveaux états (étiquette ϵ), soit entre dans $A(e)$ par son état initial et en ressort par son état final. En le coupant à chaque utilisation de la transition spontanée de retour (de l'état final de $A(e)$ vers son état initial), on obtient $k \geq 1$ chemins réussis de $A(e)$. Réciproquement, de tels chemins se recollent grâce à cette transition de retour. Le langage reconnu est donc $\{\epsilon\} \cup \bigcup_{k \geq 1} \text{lang}(e)^k = \text{lang}(e)^*$.

□

Avec l'élimination des transitions spontanées puis la déterminisation par l'automate des parties, on en déduit que tout langage régulier est reconnu par un automate fini déterministe. C'est le premier sens du théorème de Kleene ; le second sens est l'objet du théorème suivant.

11.3.2 Calcul d'une expression régulière depuis un automate - *McNaughton et Yamada*

Théorème 11.59

Soit A un automate fini déterministe, il existe une expression régulière e telle que $\text{lang}(A) = \text{lang}(e)$.

Remarque 11.60 On va s'inspirer de Floyd-Warshall pour définir de manière incrémentale des expressions régulières.

Preuve On pose $A = (\llbracket 1, n \rrbracket, \Sigma, \delta, 1, F)$ en numérotant les états de A et en supposant que l'état initial a le numéro 1. On va construire une suite de matrices d'expressions régulières $(R^{(k)})_{k \in \llbracket 0, n \rrbracket}$ telle que $\forall i, j \in \llbracket 1, n \rrbracket, R_{i,j}^{(k)}$ soit une expression régulière dénotant exactement l'ensemble des étiquettes des chemins de l'état i à l'état j dont les états intermédiaires sont $\leq k$. Les états intermédiaires d'un chemin $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_m$ sont $q_1, \dots, q_{m-1}$: on exclut le départ et l'arrivée.

On va construire les $R^{(k)}$ par récurrence.

Initialisation : pour $k = 0$, il s'agit des transitions de l'automate. On va noter ici $\alpha(i, j) = \{a \in \Sigma \mid \delta(i, a) = j\}$. On pose donc, pour $i \neq j$,

$$R_{ij}^{(0)} = \begin{cases} \emptyset & \text{si } \alpha(i, j) = \emptyset \\ a_1 \mid a_2 \mid \dots \mid a_p & \text{si } \alpha(i, j) = \{a_1, \dots, a_p\} \end{cases}$$

Quand $i = j$, il faut également considérer le chemin vide de i vers i qui est d'étiquette ϵ , on a donc :

$$R_{ii}^{(0)} = \begin{cases} \epsilon & \text{si } \alpha(i, i) = \emptyset \\ \epsilon \mid a_1 \mid a_2 \mid \dots \mid a_p & \text{si } \alpha(i, i) = \{a_1, \dots, a_p\} \end{cases}$$

Hérédité : supposons qu'on ait construit les $R^{(l)}$ pour $0 \leq l < k$. Un chemin de i à j dont les états intermédiaires sont $\leq k$ est de deux sortes :

- Soit k n'est pas un de ses états intermédiaires : ils sont donc tous $< k$, et son étiquette est dans le langage dénoté par $R_{ij}^{(k-1)}$.
- Soit k apparaît au moins une fois comme état intermédiaire. En le découpant à chaque passage par k , il a donc la forme :

$$i \xrightarrow{\phi} k \xrightarrow{\psi_1} k \rightarrow \dots \rightarrow k \xrightarrow{\psi_p} k \xrightarrow{\tau} j$$

avec $p \in \mathbb{N}$ qui correspond au nombre de fois où on revient sur l'état k dans le chemin, et où les morceaux $\phi, \psi_1, \dots, \psi_p, \tau$ n'ont que des états intermédiaires $< k$. L'étiquette de ϕ est donc dans le langage dénoté par $R_{ik}^{(k-1)}$, celles des ψ_l dans celui de $R_{kk}^{(k-1)}$ et celle de τ dans celui de $R_{kj}^{(k-1)}$. Ainsi, le chemin global a une étiquette dans le langage dénoté par

$$R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)} \right)^* R_{kj}^{(k-1)}$$

Ainsi, on pose

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \mid R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)} \right)^* R_{kj}^{(k-1)}$$

On vient de voir que toute étiquette d'un chemin de i à j d'états intermédiaires $\leq k$ est dans le langage dénoté par $R_{ij}^{(k)}$. Réciproquement, un mot de ce langage est soit l'étiquette d'un chemin d'états intermédiaires $< k$, soit l'étiquette d'un chemin obtenu en recollant en k des chemins d'états intermédiaires $< k$, dont les

états intermédiaires sont donc bien $\leq k$. Ces raisonnements restent valables quand $i = k$ ou $j = k$, car $R_{kk}^{(k-1)}$ contient le chemin vide.

On déduit de cette construction la matrice $R^{(n)}$ et on peut donc poser

$$e = R_{1f_1}^{(n)} \mid \dots \mid R_{1f_p}^{(n)}$$

où $F = \{f_1, \dots, f_p\}$, avec $e = \emptyset$ si $F = \emptyset$. Comme $R_{1f}^{(n)}$ désigne les étiquettes de tous les chemins de 1 à f (il n'y a pas de contrainte sur les états intermédiaires quand $k = n$), e désigne exactement $\text{lang}(A)$.

□

Exemple 11.61

On considère l'automate :

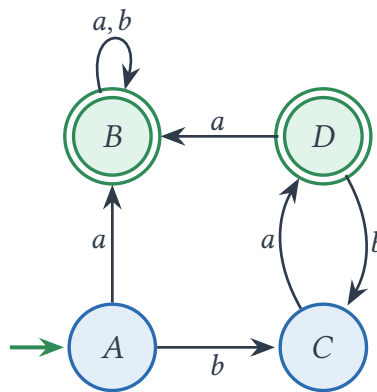


Figure 11.12 — Déterminisé de l'automate avec transitions spontanées

En numérotant dans l'ordre alphabétique $1 = A$, $2 = B$, $3 = C$ et $4 = D$, on a, en simplifiant au fur et à mesure (en utilisant par exemple $\epsilon \mid \emptyset = \emptyset$, $\epsilon \mid \emptyset = \epsilon$ et $(\epsilon \mid e)^* = e^*$), les matrices suivantes. Comme aucune transition n'arrive sur l'état 1 , on a $R^{\{(1)\}} = R^{\{\emptyset\}}$.

```

$$
R^{\{\emptyset\}} = R^{\{(1)\}} = \begin{pmatrix}
\epsilon & a & b & \emptyset \\
\emptyset & \epsilon \mid a \mid b & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon & a \\
\emptyset & a & b & \epsilon
\end{pmatrix}
$$

$$
R^{\{(2)\}} = \begin{pmatrix}
\epsilon & a \mid a(\epsilon \mid a \mid b)^*(\epsilon \mid a \mid b) & b & \emptyset \\
\emptyset & (\epsilon \mid a \mid b) \mid (\epsilon \mid a \mid b)(\epsilon \mid a \mid b)^*(\epsilon \mid a \mid b) & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon & a \\
\emptyset & a \mid a(\epsilon \mid a \mid b)^*(\epsilon \mid a \mid b) & b & \epsilon
\end{pmatrix}
= \begin{pmatrix}
\epsilon & a(a \mid b)^* & b & \emptyset \\
\emptyset & (a \mid b)^* & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon & a \\
\emptyset & a(a \mid b)^* & b & \epsilon
\end{pmatrix}
$$

```

```

R^{(3)}= \begin{pmatrix}
\epsilon & a(ab)^* & b & ba \\
\emptyset & (ab)^* & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon & a \\
\emptyset & a(ab)^* & b & \epsilon \mid b \epsilon \mid a
\end{pmatrix} =
\begin{pmatrix}
\epsilon & a(ab)^* & b & ba \\
\emptyset & (ab)^* & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon & a \\
\emptyset & a(ab)^* & b & \epsilon \mid ba
\end{pmatrix}
$$

$$
\begin{aligned}
R^{(4)} &= \begin{pmatrix}
\epsilon & a(ab)^* \mid ba(ba)^* a(ab)^* & b \mid ba(ba)^* b & ba \mid ba(ba)^* (\epsilon \mid ba) \\
\emptyset & (ab)^* & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon \mid a(ba)^* b & a \mid a(ba)^* (\epsilon \mid ba) \\
\emptyset & a(ab)^* \mid (\epsilon \mid ba)(ba)^* a(ab)^* & b \mid (\epsilon \mid ba)(ba)^* b & \epsilon \mid ba \mid (\epsilon \mid ba)(ba)^* (\epsilon \mid ba)
\end{pmatrix} \\
&= \begin{pmatrix}
\epsilon & (ba)^* a(ab)^* & (ba)^* b & ba(ba)^* \\
\emptyset & (ab)^* & \emptyset & \emptyset \\
\emptyset & \emptyset & \epsilon \mid a(ba)^* b & a(ba)^* \\
\emptyset & (ba)^* a(ab)^* & (ba)^* b & (ba)^*
\end{pmatrix}
\end{aligned}
$$

```

On en déduit donc l'expression régulière associée à l'automate de départ :

```

$$
(ba)^* a(ab)^* \mid ba(ba)^*
= (ba)^* (a(ab)^* \mid ba)
$$

```

On peut voir dans l'exemple précédent que la détermination de cette expression régulière est laborieuse : chaque coefficient de $R^{(k)}$ fait intervenir quatre coefficients de $R^{(k-1)}$. Sans simplification, les expressions de la matrice finale peuvent donc être de taille $O(4^n)$. On va voir une autre méthode permettant de déterminer de manière plus efficace des expressions régulières.

11.3.3 Élimination des états - Brzowski et McCluskey

On présente ici le principe de l'élimination des états. Il s'agit de construire un automate dont les transitions ne sont plus étiquetées par des lettres mais par des expressions régulières : un mot est alors reconnu s'il s'écrit $w_1 \dots w_n$ où chaque w_k est dans le langage dénoté par l'étiquette de la k -ième transition d'un chemin d'un état initial à un état final. Cette souplesse permet de supprimer des états en gardant le même langage reconnu. On procède donc, étant donné un état final q , par élimination jusqu'à obtenir un automate dont les états sont $\{q_i, q\}$ où q_i est l'état initial. On en déduit simplement une expression régulière équivalente. On conclut alors en faisant cela pour chaque état final et en prenant l'union des expressions régulières.

Exemple 11.62

On reprend l'automate précédent :

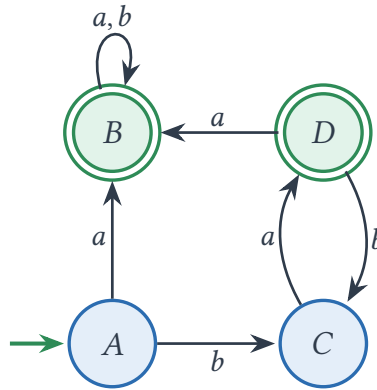
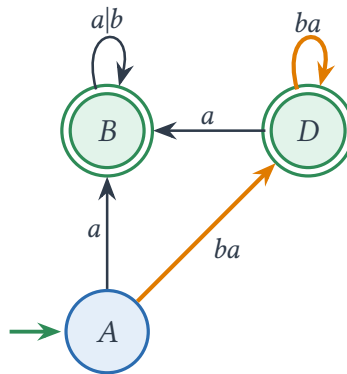
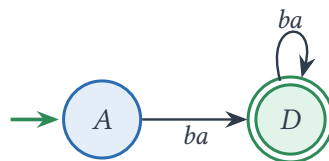


Figure 11.13 – Déterminisé de l'automate avec transitions spontanées

On commence par retirer les états qui ne sont ni initiaux ni finaux, ce qui permet de repartir de l'automate obtenu pour tous les états finaux. Ici, seul l'état CD est dans ce cas.

Figure 11.14 – Après élimination de C : transitions créées en orange

En considérant l'état final BD , comme aucun chemin de AS à DS ne passe par l'état BS , on peut l'éliminer directement pour obtenir l'automate à deux états suivant :



On en déduit directement l'expression régulière $ba(ba)^*a$.

Maintenant, on va considérer l'état final BS et on souhaite éliminer DS . On doit prendre en compte le chemin qui va de AS à BS en passant par DS , sachant qu'il y a une boucle potentielle sur DS qu'on peut prendre un nombre quelconque de fois. On obtient alors l'automate :



Qui admet $(a \mid ba (ba)^* a)(a \mid b)^* = (ba)^* a (a \mid b)^*$ comme expression régulière.

Au final, on obtient l'expression régulière :

\$\$
 (ba)^* a (a \mid b)^* \mid ba(ba)^* = (ba)^* (a (a \mid b)^* \mid ba)
 \$\$

On retrouve donc l'expression régulière obtenue avec la méthode précédente.

Remarque 11.63 Quitte à rajouter des transitions spontanées, on peut se ramener à n'avoir qu'un unique état final et donc à ne faire qu'une seule suite d'élimination.

L'automate précédent avec ce nouvel état devient :

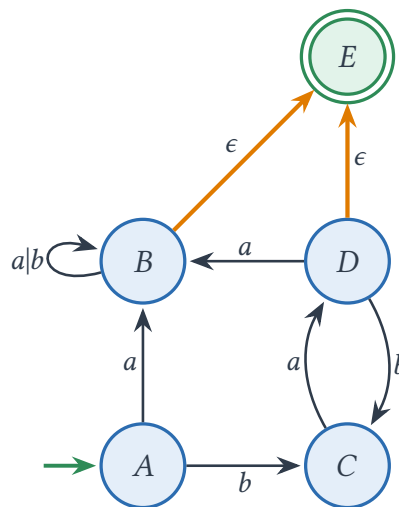
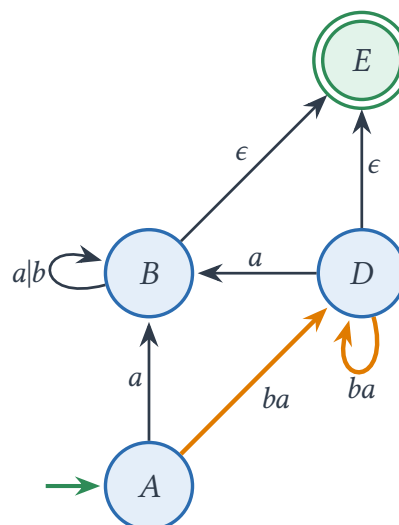


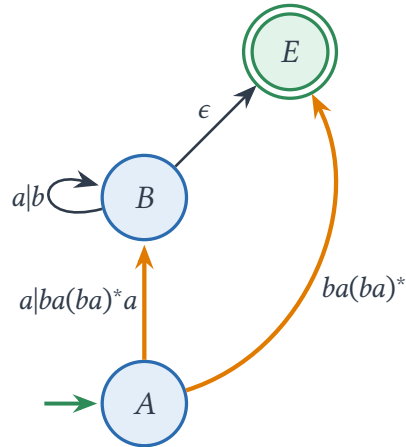
Figure 11.15 — Ajout d'un unique état final E par des transitions ϵ (en orange)

On note que ϵ ici désigne une expression régulière, on n'a donc pas à sortir du cadre précédent.

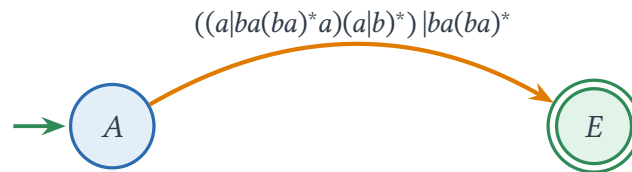
On peut alors éliminer les sommets B , C et D . On commence par éliminer C :



On élimine alors D :



Puis en éliminant B :



On retrouve ainsi l'expression régulière précédente après simplification.

On remarque que cette méthode n'est pas forcément plus pratique à la main sur des petits automates et qu'on risque de faire apparaître de longues expressions régulières et de faire des erreurs de calcul.

On va formaliser l'élimination. Si un état s a des prédécesseurs $p_1, \dots, p_n$ et des successeurs $q_1, \dots, q_m$, certains pouvant être à la fois successeurs et prédécesseurs, on a la situation locale en s suivante :

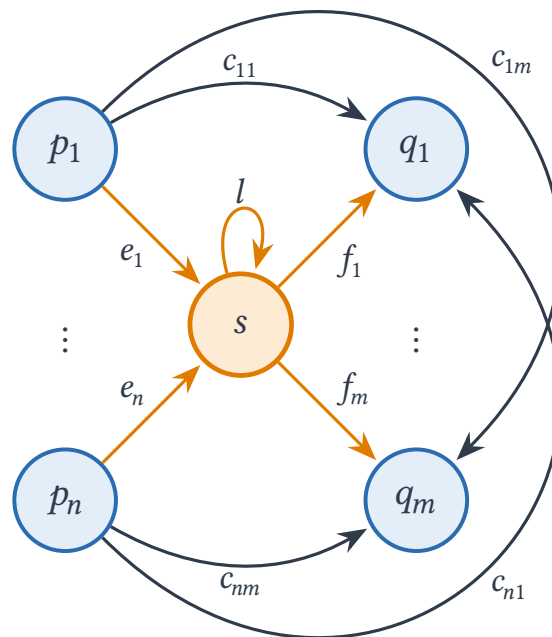


Figure 11.16 — Situation locale autour de l'état s à éliminer (en orange)

Pour passer de p_i à q_j on a donc soit l'expression déjà présente c_{ij} , soit on effectue un chemin en passant par s donc de la forme

$$p_i \rightarrow s \rightarrow s \rightarrow \dots \rightarrow s \rightarrow q_j$$

ce qui correspond à l'expression régulière $e_i l^* f_j$. On peut donc en déduire l'automate suivant avec s éliminé, qui reconnaît le même langage car tout chemin passant par s est remplacé par une transition dont l'étiquette dénote exactement les étiquettes de ces passages. Si $p_i = q_j$, la transition obtenue est une boucle sur p_i .

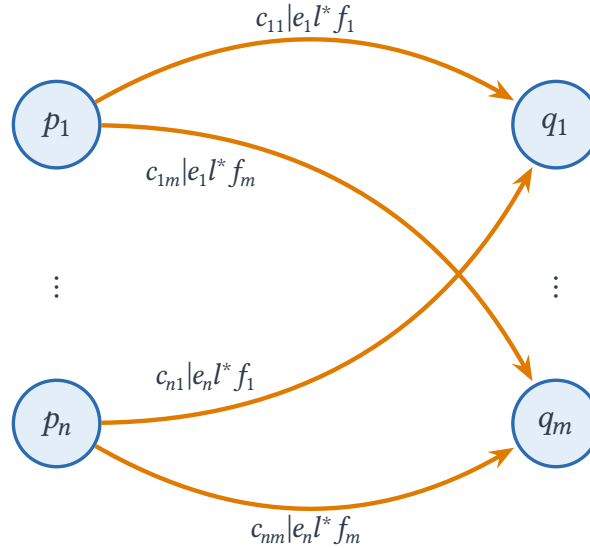


Figure 11.17 — Après élimination de s : les nouvelles étiquettes, en orange, intègrent les passages par s

Après élimination de tous les états sauf l'état initial et un état final, on obtient un automate de la forme :

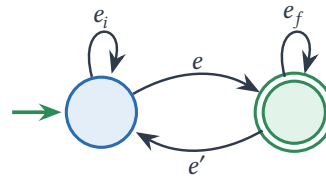


Figure 11.18 — Forme finale après élimination de tous les autres états

On lui associe alors l'expression régulière

$$(e_i | ee_f^* e')^* ee_f^*$$

En effet, un chemin de l'état initial à l'état final se découpe selon ses retours sur l'état initial : il est formé d'une suite de boucles sur l'état initial, chacune étant soit la transition e_i , soit un aller-retour $ee_f^* e'$ par l'état final, puis d'un dernier aller ee_f^* . Une transition absente est étiquetée par $\emptyset$. Si l'état final considéré est l'état initial lui-même, on élimine tous les autres états et on obtient un unique état portant une boucle e_i , d'expression e_i^* .

11.4 Construction de l'automate de Glushkov, algorithme de Berry-Sethi

On présente ici une construction directe d'un automate non déterministe reconnaissant le même langage qu'une expression régulière donnée. La construction qu'on présente ici s'effectue en plusieurs temps :

- définition et reconnaissance des langages locaux par les automates déterministes locaux
- définition d'une expression régulière linéaire et localité de son langage dénoté
- obtention d'un automate non déterministe par linéarisation d'une expression régulière puis oubli du marquage de lettres

11.4.1 Langages locaux

Définition 11.64

Soit Σ un alphabet fini et $L \subset \Sigma^*$ un langage.

On dit que L est **local** lorsqu'il existe $P, S \subset \Sigma$ et $N \subset \Sigma^2$ tels que les mots non vides de L sont exactement les mots qui

- commencent par une lettre de P
- finissent par une lettre de S
- ne contiennent aucun facteur dans N

Autrement dit, L vérifie l'égalité :

$$L \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$$

Remarque 11.65 Pour savoir si un mot est dans un langage local, il suffit donc d'avancer avec une fenêtre de lecture de deux caractères, c'est en ce sens qu'il est local.

Si on sait qu'un langage est local, il est facile de déterminer des ensembles P, S et N convenables.

Théorème 11.66

Soit L un langage. On pose

$$P(L) = \{a \in \Sigma \mid \exists w \in \Sigma^*, aw \in L\}$$

$$S(L) = \{a \in \Sigma \mid \exists w \in \Sigma^*, wa \in L\}$$

$$N(L) = \{ab \in \Sigma^2 \mid \forall w \in L, ab \text{ n'est pas un facteur de } w\}$$

Alors L est local si et seulement si $L \setminus \{\epsilon\} = (P(L)\Sigma^* \cap \Sigma^*S(L)) \setminus \Sigma^*N(L)\Sigma^*$.



Preuve Notons $L' = (P(L)\Sigma^* \cap \Sigma^*S(L)) \setminus \Sigma^*N(L)\Sigma^*$. Par définition de ces trois ensembles, on a toujours $L \setminus \{\epsilon\} \subset L'$.

Le sens réciproque de l'équivalence est immédiat. Supposons L local pour des ensembles P, S, N . Tout mot non vide de L commence par une lettre de P , donc $P(L) \subset P$. De même $S(L) \subset S$. Enfin, si $ab \in N$, ab n'est facteur d'aucun mot de L , donc $N \subset N(L)$. Ainsi $L' \subset (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^* = L \setminus \{\epsilon\}$, d'où l'égalité. $\square$

Exemple 11.67

- Σ^* est local avec $P = S = \Sigma$ et $N = \emptyset$.
- $L = a(aa)^*b$ n'est pas local car on aurait alors, d'après le théorème précédent, $P = \{a\}$, $S = \{b\}$ et $N = \{ba, bb\}$. Or, $(P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$ contient les mots de la forme $a^n b$ avec $n \in \mathbb{N}^*$ et donc contient aab qui n'est pas dans L , car les mots de L contiennent un nombre impair de a .

11.4.2 Automates locaux

Définition 11.68

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate fini déterministe.

On dit que A est

- **local** lorsque $\forall a \in \Sigma, |\{\delta(q, a) \mid q \in Q, \delta(q, a) \text{ défini}\}| \leq 1$
- **standard** lorsque $\forall a \in \Sigma, \forall q \in Q, \delta(q, a) \neq q_i$.

Remarque 11.69 Ainsi, un automate est local si toutes les transitions étiquetées par une même lettre arrivent sur le même état, et standard quand on ne peut pas revenir sur l'état initial.

Théorème 11.70

Soit L un langage, les propriétés suivantes sont équivalentes :

1. L est local
2. L est reconnaissable par un automate déterministe local
3. L est reconnaissable par un automate déterministe local et standard

Preuve On a directement $3 \Rightarrow 2$. On va montrer deux autres implications pour avoir l'équivalence.

$1 \Rightarrow 3$ Soit L un langage local et S, P, N tels que

$$L \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$$

On définit $A = (\Sigma \cup \{\epsilon\}, \Sigma, \delta, \epsilon, S')$ où $S' = \begin{cases} S & \text{si } \epsilon \notin L \\ S \cup \{\epsilon\} & \text{sinon} \end{cases}$, $\forall a \in P, \delta(\epsilon, a) = a$ et $\forall a, b \in \Sigma, ab \notin N \Rightarrow \delta(a, b) = b$. Cet automate est déterministe local et standard par définition.

On va montrer que $\text{lang}(A) = L$.

- $\text{lang}(A) \subset L$. Soit w reconnu par A . Si $w = \epsilon$, c'est forcément que ϵ est final donc que $\epsilon \in L$. Sinon $w = a_1 \dots a_n$. On a donc un chemin $\epsilon \xrightarrow{a_1} a_1 \dots \xrightarrow{a_n} a_n$ or :
 - la transition depuis ϵ implique $a_1 \in P$;
 - a_n est final, donc nécessairement $a_n \in S$;
 - pour $i \in \llbracket 1, n-1 \rrbracket$, $a_i \xrightarrow{a_{i+1}}$ implique que $a_i a_{i+1} \notin N$.

Ainsi, par définition $w \in L$.

- $L \subset \text{lang}(A)$. Réciproquement, soit $w \in L$. Soit $w = \epsilon$ et alors ϵ est initial et final, donc $\epsilon \in \text{lang}(A)$, soit $w = a_1 \dots a_n$ et, comme $a_1 \in P$, on a la transition $\epsilon \xrightarrow{a_1} a_1$. Ensuite, pour chaque $i \in \llbracket 1, n-1 \rrbracket$, comme $a_i a_{i+1} \notin N$, on a $a_i \xrightarrow{a_{i+1}}$. Ainsi, on a donc un chemin de ϵ à a_n , or $a_n \in S$ donc il est final et w est reconnu.

$2 \Rightarrow 1$. On va définir des ensembles P, S, N associés à l'automate local $A = (Q, \Sigma, \delta, q_i, F)$:

$$P = \{a \in \Sigma \mid \delta(q_i, a) \text{ est défini} \}$$

$$S = \{a \in \Sigma \mid \exists q \in Q, \delta(q, a) \in F\}$$

$$N = \left\{ ab \in \Sigma^2 \mid \text{il n'existe pas } q, q', q'' \in Q, q \xrightarrow{a} q' \xrightarrow{b} q'' \right\}$$

On pose alors $L' = (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$, qui ne contient pas ϵ , et on va montrer que $L' = \text{lang}(A) \setminus \{\epsilon\}$, ce qui prouvera que $\text{lang}(A)$ est local.

- $L' \subset \text{lang}(A)$. Soit $w = a_1 \dots a_n \in L'$. Par définition de L' , on a $a_1 \in P$ et $a_n \in S$. On va montrer par récurrence bornée que $\delta^*(q_i, a_1 \dots a_j)$ est bien défini pour $j \in \llbracket 1, n \rrbracket$.
 - Initialisation : on a $\delta^*(q_i, a_1) = \delta(q_i, a_1)$ défini par définition de P .
 - Hérédité : Soit $j \in \llbracket 1, n-1 \rrbracket$ tel que $\delta^*(q_i, a_1 \dots a_j)$ est bien défini. On pose $q' = \delta^*(q_i, a_1 \dots a_j)$: c'est un état sur lequel arrive une transition étiquetée par a_j . Par hypothèse, $a_j a_{j+1} \notin N$ donc il existe $p \xrightarrow{a_j} p' \xrightarrow{a_{j+1}} p''$. Comme l'automate est local, toutes les transitions étiquetées par a_j arrivent sur le même état, donc $p' = q'$ et $\delta^*(q_i, a_1 \dots a_{j+1}) = \delta(q', a_{j+1}) = p''$ est bien défini.

Pour $j = n$, on a $q = \delta^*(q_i, a_1 \dots a_n)$ bien défini. Comme $a_n \in S$, il existe une transition étiquetée par a_n arrivant sur un état final, et, par localité, cet état est q . Donc $q \in F$ et w est reconnu.

- $\text{lang}(A) \setminus \{\epsilon\} \subset L'$. Soit $w = a_1 \dots a_n$ un mot non vide reconnu par A . On a donc un chemin

$$q_i \xrightarrow{a_1} \dots \xrightarrow{a_n} q \in F$$

Par définition, on a donc $a_1 \in P$, $a_n \in S$ et comme on a pu réaliser les transitions, pour tout $j \in \llbracket 1, n-1 \rrbracket$, $a_j a_{j+1} \notin N$. Ainsi $w \in L'$.

□

On peut déduire de cette caractérisation des propriétés de stabilité des langages locaux.

Théorème 11.71

Soit L un langage local, le langage L^* est local.

Preuve Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate local et standard pour L , qui existe d'après le théorème précédent.

On pose $A' = (Q, \Sigma, \delta', q_i, F \cup \{q_i\})$ avec $\delta'(q, a) = \delta(q, a)$ pour (q, a) dans le domaine de δ et pour toute transition $\delta(q_i, a) = q$ et $q' \in F$, on pose $\delta'(q', a) = q$. Cela ne crée pas de conflit : si $\delta(q', a)$ était déjà défini, il valait q par localité. Ces nouvelles transitions arrivent sur l'unique état cible des transitions étiquetées par a , ce qui préserve la localité de l'automate. □

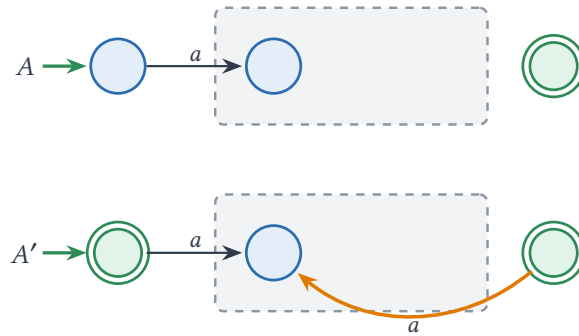


Figure 11.19 — Construction de A' : l'état initial devient final, transition ajoutée en orange

Montrons que A' reconnaît L^* .

- * $\text{lang}(A') \subset L^*$. Les nouvelles transitions arrivent sur des états $\delta(q_i, a) \neq q_i$ car A est standard, donc A' est encore standard : le seul chemin réussi aboutissant en q_i est le chemin vide, d'étiquette ϵ . Sinon, le chemin aboutit dans F . On le coupe juste avant chaque utilisation d'une nouvelle transition $q' \xrightarrow{a} q$ (avec $q' \in F$). Chaque morceau part de q_i ou commence par une telle transition, qu'on remplace par $q_i \xrightarrow{a} q$, et n'utilise ensuite que des transitions de A jusqu'à un état de F . Chaque morceau est donc l'étiquette d'un chemin réussi de A , et le mot est dans L^* .
- * $L^* \subset \text{lang}(A')$. ϵ est reconnu car q_i est final. Soit $w = u_1 \dots u_k$ avec $u_1, \dots, u_k \in L \setminus \{\epsilon\}$ et $k \geq 1$. Le mot u_1 étiquette un chemin de q_i à un état de F dans A . Pour $l \geq 2$, on écrit $u_l = a v$; il existe dans A un chemin $q_i \xrightarrow{a} q \xrightarrow{v} f$ avec $f \in F$, et dans A' on peut le reprendre depuis l'état final atteint précédemment grâce à la nouvelle transition étiquetée par a vers q . On recolle ainsi un chemin réussi d'étiquette w .

Théorème 11.72

Soient L_1, L_2 des langages locaux sur des alphabets disjoints Σ_1 et Σ_2 les langages $L_1 \cup L_2$ et $L_1 L_2$ sont locaux sur l'alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$.

Preuve On va juste donner les constructions correspondantes d'automates locaux.

Soient $A_1 = (Q_1, \Sigma_1, \delta_1, i_1, F_1)$ et $A_2 = (Q_2, \Sigma_2, \delta_2, i_2, F_2)$ des automates standards locaux reconnaissant respectivement L_1 et L_2 , avec $Q_1 \cap Q_2 = \emptyset$.

On pose

$$A_u = ((Q_1 \cup Q_2 \cup \{i\}) \setminus \{i_1, i_2\}, \Sigma, \delta_u, i, F_u)$$

avec

$$F_u = \begin{cases} F_1 \cup F_2 & \text{si } i_1 \notin F_1 \wedge i_2 \notin F_2 \\ (F_1 \cup F_2 \cup \{i\}) \setminus \{i_1, i_2\} & \text{sinon} \end{cases}$$

$$\forall q \in Q_1 \setminus \{i_1\}, \forall a \in \Sigma_1, \delta_u(q, a) = \delta_1(q, a) \text{ si défini}$$

$$\forall q \in Q_2 \setminus \{i_2\}, \forall a \in \Sigma_2, \delta_u(q, a) = \delta_2(q, a) \text{ si défini}$$

$$\forall a \in \Sigma_1, \delta_u(i, a) = \delta_1(i_1, a) \text{ si défini}$$

$$\forall a \in \Sigma_2, \delta_u(i, a) = \delta_2(i_2, a) \text{ si défini}$$

Comme les automates A_1 et A_2 sont standards, aucune transition n'arrivait sur i_1 ou i_2 : on n'a pas perdu de transitions ici.

A_u est déterministe car les alphabets sont disjoints. Il est local : une lettre $a \in \Sigma_1$ ne mène que vers l'unique état cible des transitions étiquetées par a dans A_1 , et de même pour Σ_2 . Il est standard car aucune transition n'arrive sur i .

Enfin, un chemin non vide partant de i commence par une lettre de Σ_1 , et reste alors dans $Q_1 \setminus \{i_1\}$ car les états de Q_1 n'ont que des transitions vers $Q_1 \setminus \{i_1\}$, ou par une lettre de Σ_2 , et reste alors dans $Q_2 \setminus \{i_2\}$. Les chemins réussis non vides de A_u correspondent donc exactement à ceux de A_1 et de A_2 , en remplaçant i par i_1 ou i_2 , et le mot vide est reconnu si et seulement si $\epsilon \in L_1 \cup L_2$. Ainsi A_u reconnaît $L_1 \cup L_2$.

De même, on pose

$$A_p = ((Q_1 \cup Q_2) \setminus \{i_2\}, \Sigma, \delta_p, i_1, F_p)$$

où

$$F_p = \begin{cases} F_2 & \text{si } i_2 \notin F_2 \\ F_1 \cup (F_2 \setminus \{i_2\}) & \text{sinon} \end{cases}$$

$$\forall q \in Q_1, \forall a \in \Sigma_1, \delta_p(q, a) = \delta_1(q, a) \text{ si défini}$$

$$\forall q \in Q_2 \setminus \{i_2\}, \forall a \in \Sigma_2, \delta_p(q, a) = \delta_2(q, a) \text{ si défini}$$

$$\forall q \in F_1, \forall q' \in Q_2, \forall a \in \Sigma_2, \delta_2(i_2, a) = q' \Rightarrow \delta_p(q, a) = q'$$

Les nouvelles transitions partent d'états de Q_1 avec des lettres de Σ_2 , donc A_p reste déterministe. Elles arrivent sur l'unique état cible des transitions étiquetées par a dans A_2 , donc A_p est local, et il est standard car aucune transition n'arrive sur i_1 .

Un chemin réussi de A_p lit d'abord des lettres de Σ_1 en restant dans Q_1 , puis, s'il lit une lettre de Σ_2 , passe d'un état $q \in F_1$ à $Q_2 \setminus \{i_2\}$ par une nouvelle transition et ne revient plus dans Q_1 . S'il ne passe jamais dans Q_2 , il finit dans F_1 , ce qui n'est possible que si $\epsilon \in L_2$: son étiquette est dans $L_1 \cdot \{\epsilon\} \subset L_1 L_2$. Sinon, son étiquette s'écrit uv avec u l'étiquette d'un chemin réussi de A_1 et v celle d'un chemin réussi de A_2 , en remplaçant la nouvelle transition par la transition correspondante partant de i_2 . La réciproque se montre en recollant de la même façon un chemin réussi de A_1 et un de A_2 . Ainsi A_p reconnaît $L_1 L_2$. □

11.4.3 Expressions régulières linéaires

Définition 11.73

Une expression régulière est dite **linéaire** si chaque lettre de Σ y apparaît au plus une fois.

Exemple 11.74

L'expression régulière $a(b \mid c)^*$ est linéaire mais $a(a \mid b)^*$ ne l'est pas.

On déduit des deux théorèmes précédents le théorème suivant :

Théorème 11.75

Soit e une expression régulière linéaire, $\text{lang}(e)$ est un langage local.

Preuve Par induction sur l'expression régulière.

- *Cas de base* : Si $e = \emptyset, \epsilon$ ou a avec $a \in \Sigma$, c'est direct : par exemple $\{a\}$ est local avec $P = S = \{a\}$ et $N = \Sigma^2$.
- Pour $e^*, e \mid e'$ ou ee' , l'hypothèse d'induction et les théorèmes précédents permettent de conclure, en voyant $\text{lang}(e)$ et $\text{lang}(e')$ comme des langages sur les alphabets des lettres apparaissant respectivement dans e et e' . Ces alphabets sont disjoints par linéarité : une même lettre ne peut pas être présente dans e et dans e' .
- Enfin, un langage local sur un alphabet $\Sigma' \subset \Sigma$ est local sur Σ : il suffit de garder les mêmes P et S , et d'ajouter à N tous les facteurs de longueur 2 contenant une lettre de $\Sigma \setminus \Sigma'$.

□

Remarque 11.76 La réciproque est fautive, il existe des langages locaux qui ne sont pas dénotés par des expressions linéaires. Par exemple $(ab)^*a$ est local avec $P = \{a\}, S = \{a\}$ et $N = \{aa, bb\}$, mais une expression régulière le dénotant devra forcément contenir au moins deux a (on l'admet ici).

Corollaire 11.77

Pour toute expression régulière linéaire e , il existe un automate local déterministe A tel que $\text{lang}(e) = \text{lang}(A)$.

Pour déterminer cet automate, on a vu une preuve explicite en déterminant les ensembles P, S et N pour $\text{lang}(e)$. Pour calculer ces ensembles pour une expression régulière, on peut le faire par induction.

Tout d'abord, comme on a besoin de savoir si l'expression reconnaît le mot vide, on peut définir la propriété $\text{accv}(e)$:

$$\text{accv}(\emptyset) = \perp \quad \forall a \in \Sigma, \text{accv}(a) = \perp \quad \text{accv}(\epsilon) = \top \quad \text{accv}(e^*) = \top$$

$$\text{accv}(e_1 \mid e_2) = \text{accv}(e_1) \vee \text{accv}(e_2) \quad \text{accv}(e_1 e_2) = \text{accv}(e_1) \wedge \text{accv}(e_2)$$

On définit ensuite P, S et N directement. On suppose ici que e ne contient pas $\emptyset$, ce qui n'est pas restrictif : une expression régulière est équivalente soit à $\emptyset$, soit à une expression ne contenant pas $\emptyset$, grâce aux simplifications $e\emptyset = \emptyset e = \emptyset$, $e \mid \emptyset = \emptyset \mid e = e$ et $\emptyset^* = \epsilon$. Sans cette hypothèse, la formule donnant $S(e_1 e_2)$ serait fautive pour $e_1 = \emptyset$.

$$P(\emptyset) = \emptyset \quad \forall a \in \Sigma, P(a) = \{a\} \quad P(\epsilon) = \emptyset \quad P(e^*) = P(e)$$

$$P(e_1 \mid e_2) = P(e_1) \cup P(e_2) \quad P(e_1 e_2) = \begin{cases} P(e_1) \cup P(e_2) & \text{si } \text{accv}(e_1) \\ P(e_1) & \text{sinon} \end{cases}$$

$$S(\emptyset) = \emptyset \quad \forall a \in \Sigma, S(a) = \{a\} \quad S(\epsilon) = \emptyset \quad S(e^*) = S(e)$$

$$S(e_1 \mid e_2) = S(e_1) \cup S(e_2) \quad S(e_1 e_2) = \begin{cases} S(e_1) \cup S(e_2) & \text{si } \text{accv}(e_2) \\ S(e_2) & \text{sinon} \end{cases}$$

$$N(\emptyset) = N(\epsilon) = \Sigma^2 \quad \forall a \in \Sigma, N(a) = \Sigma^2 \quad N(e^*) = N(e) \setminus S(e)P(e)$$

$$N(e_1 \mid e_2) = N(e_1) \cap N(e_2) \quad N(e_1 e_2) = (N(e_1) \cap N(e_2)) \setminus S(e_1)P(e_2)$$

Ces formules calculent les ensembles $P(L)$, $S(L)$ et $N(L)$ du théorème de caractérisation pour $L = \text{lang}(e)$. Pour N , il est plus simple de raisonner sur son complémentaire, l'ensemble des facteurs de longueur 2 des

mots de L . Par exemple, un facteur de longueur 2 d'un mot uv avec $u \in \text{lang}(e_1)$ et $v \in \text{lang}(e_2)$ est soit un facteur de u , soit un facteur de v , soit formé de la dernière lettre de u et de la première de v . L'hypothèse d'absence de $\emptyset$ garantit que les langages $\text{lang}(e_1)$ et $\text{lang}(e_2)$ sont non vides, ce qui permet d'obtenir tous ces facteurs.

11.4.4 Application aux expressions régulières

Étant donné une expression régulière e , on peut en déduire une expression régulière linéaire en indiquant les occurrences de chaque lettre. Par exemple, $(ab)^*aba$ deviendra $(a_1b_1)^*a_2b_2a_3$.

On peut alors en déduire un automate déterministe local par la méthode précédente. En oubliant les indices sur les transitions de cet automate, on en déduit un automate *non déterministe* reconnaissant le même langage que l'expression régulière initiale. En effet, si on note h l'application qui efface les indices d'un mot, on a $\text{lang}(e) = h(\text{lang}(e'))$ (par une induction immédiate sur e), et les chemins réussis de l'automate sans indices sont exactement ceux de l'automate local, avec des étiquettes dont on a effacé les indices.

C'est l'algorithme de *Berry-Sethi*.

Exemple 11.78

On va considérer l'expression régulière $e = (a \mid b)^*b(a \mid b)$.

On commence par la linéarisation $e' = (a_1 \mid b_1)^*b_2(a_2 \mid b_3)$.

On calcule les ensembles P, S et N :

$$P((a_1 \mid b_1)^*b_2(a_2 \mid b_3)) = \{a_1, b_1, b_2\}$$

$$S((a_1 \mid b_1)^*b_2(a_2 \mid b_3)) = \{a_2, b_3\}$$

Pour N , il est plus rapide de donner son complémentaire :

$$\Sigma^2 \setminus N((a_1 \mid b_1)^*b_2(a_2 \mid b_3)) = \{a_1b_2, b_1b_2, a_1a_1, b_1b_1, a_1b_1, b_1a_1, b_2a_2, b_2b_3\}$$

On en déduit l'automate local :

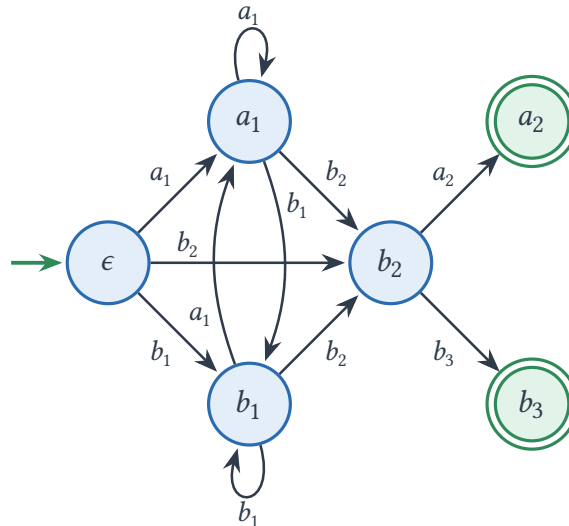


Figure 11.20 — Automate local obtenu par Berry-Sethi pour $(a_1 \mid b_1)^*b_2(a_2 \mid b_3)$

En oubliant les indices, on obtient l'automate :

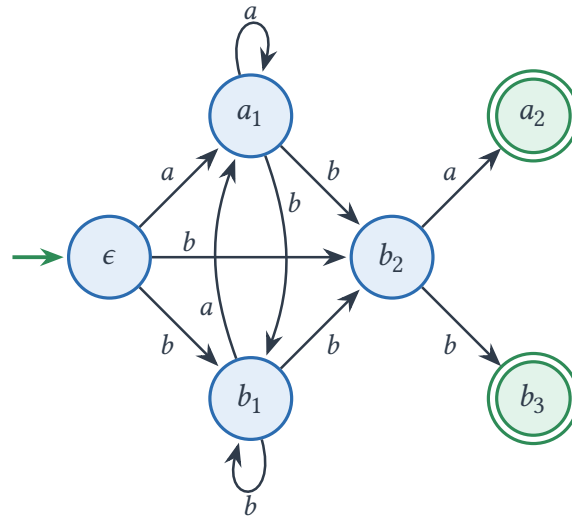


Figure 11.21 — Automate obtenu en oubliant les indices : il reconnaît $(a \mid b)^*b(a \mid b)$

11.5 Conséquences du théorème de Kleene

11.5.1 Stabilité des langages réguliers

Par union finie

Théorème 11.79

Soient $L_1, \dots, L_n$ des langages réguliers sur Σ . Le langage $L_1 \cup \dots \cup L_n$ est régulier.

Preuve Du théorème de Kleene on déduit des expressions régulières $e_1, \dots, e_n$ dénotant respectivement $L_1, \dots, L_n$. Le langage dénoté par $e_1 \mid \dots \mid e_n$ est $L_1 \cup \dots \cup L_n$ et par le théorème de Kleene, il est régulier.

□

Par complémentaire

Lemme 11.80

Soit $A = (Q, \Sigma, \delta, q_i, F)$ un automate déterministe **complet**, l'automate $A' = (Q, \Sigma, \delta, q_i, Q \setminus F)$ reconnaît $\Sigma^* \setminus \text{lang}(A)$.



Preuve Comme A est complet, $\delta^*(q_i, w)$ est défini pour tout $w \in \Sigma^*$, et comme il est déterministe, c'est l'unique état atteint en lisant w . Ainsi $w \in \text{lang}(A') \Leftrightarrow \delta^*(q_i, w) \in Q \setminus F \Leftrightarrow \delta^*(q_i, w) \notin F \Leftrightarrow w \notin \text{lang}(A)$.

Les deux hypothèses sont nécessaires : si A n'est pas complet, un mot qui bloque n'est reconnu ni par A ni par A' ; si A n'est pas déterministe, un mot peut atteindre à la fois un état de F et un état de $Q \setminus F$, et être reconnu par les deux.

□

Théorème 11.81

Les langages réguliers sont stables par complémentaire. Si L est régulier sur Σ^* alors $\Sigma^* \setminus L$ est régulier.

Preuve Comme L est régulier il est reconnaissable par un automate déterministe qu'on rend complet en rajoutant un état puits. En appliquant le lemme et le théorème de Kleene, on en déduit que $\Sigma^* \setminus L$ est régulier.

□

Par intersection finie

Théorème 11.82

Soient $L_1, \dots, L_n$ des langages réguliers sur Σ . Le langage $L_1 \cap \dots \cap L_n$ est régulier.

Preuve On a $L_1 \cap \dots \cap L_n = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup \dots \cup (\Sigma^* \setminus L_n))$. Il s'agit donc d'une conséquence directe des deux théorèmes précédents. □

11.5.2 Lemme de l'étoile

Théorème 11.83

Lemme de l'étoile Soit L un langage régulier sur Σ . Il existe une constante $n \in \mathbb{N}^*$ telle que pour tout $w \in L$, si $|w| \geq n$ alors il existe $x, y, z \in \Sigma^*$ tels que $w = xyz$ et

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $xy^*z \subset L$

Remarque 11.84 Cela signifie que pour tout mot suffisamment long, on peut identifier un facteur non vide qui peut être répété autant qu'on le souhaite.

Preuve Soit $A = (Q, \Sigma, \delta, q_0, F)$ un automate déterministe reconnaissant L . On pose $n = |Q|$.

Soit $w \in L$ tel que $|w| \geq n$. Comme w est reconnu, il existe un chemin $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_m \in F$ d'étiquette w . Comme $m = |w| \geq n = |Q|$, les $n + 1$ états $q_0, \dots, q_n$ ne peuvent pas être deux à deux distincts : il existe $0 \leq i < j \leq n$ tels que $q_i = q_j$. On pose alors x l'étiquette de $q_0 \rightsquigarrow q_i$, y l'étiquette de $q_i \rightsquigarrow q_j$ et z celle de $q_j \rightsquigarrow q_m$.

On a $w = xyz$, $|y| = j - i > 0$ donc $y \neq \epsilon$ et $|xy| = j \leq n$.

Comme $q_i = q_j$, le chemin correspondant à y est une boucle, on peut donc l'emprunter autant qu'on le souhaite et ainsi tous les mots de la forme xy^kz sont reconnus par A . □

Ce lemme est très utile par sa contraposée qui permet d'affirmer que si un langage ne vérifie pas cette propriété, alors il n'est pas régulier :

Corollaire 11.85

Soit L un langage sur Σ . Si pour tout $n \in \mathbb{N}^*$, il existe $w \in L$ tel que $|w| \geq n$ et que pour tout $x, y, z \in \Sigma^*$ tels que $w = xyz$, $y \neq \epsilon$ et $|xy| \leq n$ il existe $k \in \mathbb{N}$ avec $xy^kz \notin L$, alors L n'est pas régulier.

Exemple 11.86

Le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier.

Soit $n \in \mathbb{N}^*$, on considère $w = a^n b^n \in L$, qui est bien de longueur $\geq n$. Soient x, y, z tels que $w = xyz$ avec $y \neq \epsilon$ et $|xy| \leq n$, on a donc xy préfixe de a^n : $x = a^k$ et $y = a^l$ avec $k + l \leq n$ et $l > 0$ et ainsi $z = a^{n-k-l} b^n$.

Le mot $xz = a^k a^{n-k-l} b^n = a^{n-l} b^n \notin L$ car $n - l < n$.

Ainsi, selon le lemme de l'étoile, L n'est pas régulier.

Remarque 11.87 Cet exemple est très important car c'est un cas particulier très simple des expressions bien parenthésées. On sait qu'on peut les reconnaître avec une pile, et on vient de voir que c'est impossible de le faire avec un automate fini.

Remarque 11.88 Attention, le lemme de l'étoile n'est pas une équivalence. Un langage peut très bien le satisfaire et être non régulier comme on va le voir dans l'exemple suivant.

Exemple 11.89

On considère le langage

$$L = \{ab^n c^n \mid n \in \mathbb{N}\} \cup \{a^k w \mid k \neq 1, w \in \{b, c\}^*\} = L_1 \cup L_2$$

On a $L \cap ab\Sigma^* = \{ab^n c^n \mid n \in \mathbb{N}^*\}$, qui n'est pas régulier : c'est, à la lettre a en préfixe près, le même exemple que le précédent, et la même preuve s'applique. Or, $ab\Sigma^*$ est régulier donc nécessairement L n'est pas régulier par clôture des langages réguliers par intersection.

On va montrer que L vérifie le lemme de l'étoile pour $n = 2$.

Soit $w \in L$ tel que $|w| \geq 2$. On a quatre cas possibles pour w :

- $w = ab^n c^n$ pour $n \in \mathbb{N}^*$, on peut alors poser $x = \epsilon$, $y = a$ et $z = b^n c^n$. On a pour $k \in \mathbb{N}$, $xy^k z = a^k b^n c^n$ qui est soit dans L_1 si $k = 1$ ou dans L_2 sinon.
- $w = a^k v$ avec $k \geq 2$, $k \neq 3$ et $v \in \{b, c\}^*$. On pose $x = \epsilon$, $y = aa$ et $z = a^{k-2} v$. On a $xy^l z = a^{2l} a^{k-2} v$. Si $l = 0$, alors $xy^0 z = a^{k-2} v \in L$ car $k - 2 \neq 1$. Si $l > 0$, alors $2l + k - 2 \geq 2l \geq 2$ donc $xy^l z \in L$.
- $w = a^3 v$ avec $v \in \{b, c\}^*$. On pose $x = \epsilon$, $y = a$ et $z = a^2 v$. Ainsi $xy^l z = a^{l+2} v \in L$ car $l + 2 \geq 2$.
- $w \in \{b, c\}^*$: on écrit $w = \ell v$ avec $\ell \in \{b, c\}$ la première lettre de w et on pose $x = \epsilon$, $y = \ell$ et $z = v$. On a alors $xy^k z = \ell^k v \in \{b, c\}^* \subset L$.

Ainsi, on a bien prouvé que L satisfait le lemme de l'étoile.

Langages réguliers et automates finis

12.1 Fiche de synthèse

Définition 12.1

AFD $A = (Q, \Sigma, \delta, q_i, F)$

- un seul état initial
- pas de choix : si $q \xrightarrow{a} q'$ et $q \xrightarrow{a} q''$ alors $q' = q''$

AFND $A = (Q, \Sigma, \delta, I, F)$: plusieurs états initiaux et choix

AFND ϵ $A = (Q, \Sigma, \delta, q_i, F)$: un seul état initial, choix et transitions spontanées $q \xrightarrow{\epsilon} q'$

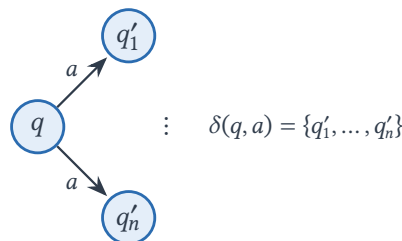
complet : toutes les transitions sont définies, $\forall q \in Q, \forall a \in \Sigma, \exists q', q \xrightarrow{a} q'$

émondé : $\forall q \in Q, \exists q_f \in F, q_i \rightsquigarrow q \rightsquigarrow q_f$

Fonction de transition AFD $\delta : Q \times \Sigma \rightarrow Q$ partielle

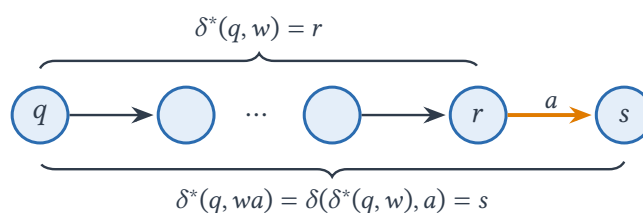
$(q, a) \in \text{dom}(\delta) \Leftrightarrow q \xrightarrow{a} \delta(q, a)$

AFND $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$



AFND ϵ $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$
 transition spontanée $q \xrightarrow{\epsilon} q'$

Fonction de transition étendue AFD $q_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_n} q_n \Rightarrow \delta^*(q_0, a_1 \dots a_n) = q_n$



```


$$\delta^*(q, \epsilon) = q \quad \text{w} \in \text{mathop{lang}}(A) \iff \delta^*(q_i, w) \in F$$


**AFND**  $q_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_n} q_n$ 
 $\xrightarrow{\text{Rightarrow}} q_n \in \delta^*(q_0, a_1 \dots a_n)$ 

 $\delta^*(q, \epsilon) = \{q\}$ 
 $\delta^*(q, wa) = \bigcup_{q' \in \delta^*(q, w)} \delta(q', a)$ 

 $w \in \text{mathop{lang}}(A) \iff \left( \bigcup_{q_i \in I} \delta^*(q_i, w) \right) \cap F \neq \emptyset$ 

```

AFND → AFD

$$A = (Q, \Sigma, \delta, I, F) \Rightarrow A' = (\mathcal{P}(Q), \Sigma, \delta', I, F')$$

$$\delta'(E, a) = \bigcup_{q \in E} \delta(q, a)$$

$$F' = \{E \subset Q \mid E \cap F \neq \emptyset\}$$



Figure 12.1 — Un AFND (à gauche) et l'AFD obtenu par l'automate des parties (à droite)

```

::: {.mtable cols="4"}
::: cellule
\
:::
::: cellule
\
:::
::: cellule
$a$
:::
::: cellule
$b$
:::
::: cellule
$\rightarrow *$
:::
::: cellule
$A = \{ 1, 2 \}$
:::
::: cellule
$B$
:::
::: cellule
$C$
:::
::: cellule
$*$
:::
::: cellule
$B = \{2, 3 \}$
:::
::: cellule
$D$
:::

```

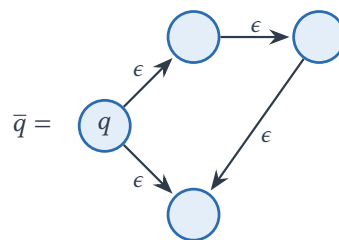
```

::: cellule
$C$
:::
::: cellule
$*$
:::
::: cellule
$C = \{1, 3 \}$
:::
::: cellule
$B$
:::
::: cellule
$E$
:::
::: cellule
$*$
:::
::: cellule
$D = \{ 2 \}$
:::
::: cellule
$\emptyset$
:::
::: cellule
$C$
:::
::: cellule
$*$
:::
::: cellule
$E = \{ 3 \}$
:::
::: cellule
$D$
:::
::: cellule
$\emptyset$
:::

```

AFND ϵ : clôture

$$\bar{q} = \{q' \mid q \xrightarrow{\epsilon} \dots \xrightarrow{\epsilon} q'\} \text{ (avec } q \in \bar{q})$$



$$\delta^*(q, \epsilon) = \overline{\{q\}} \quad \delta^*(q, wa) = \overline{\bigcup_{q' \in \delta^*(q, w)} \delta(q', a)}$$

AFND $\epsilon \rightarrow$ AFND

$$A = (Q, \Sigma, \delta, q_i, F) \Rightarrow A' = (Q, \Sigma, \delta', \bar{q}_i, F)$$

$$\delta'(q, a) = \overline{\delta(q, a)}$$

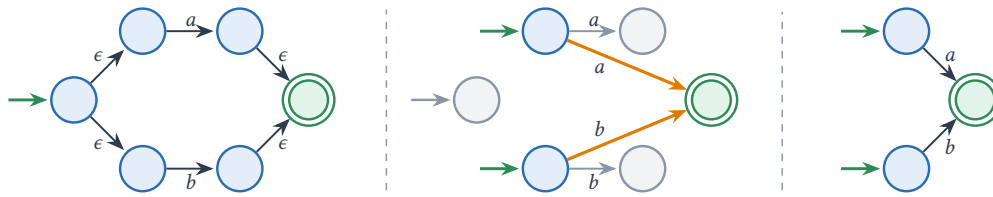


Figure 12.2 – Suppression des transitions ϵ : transitions ajoutées en orange, états émondés en gris

Thompson : expression régulière $\rightarrow$ AFND ϵ

$A(\emptyset) =$



$A(\epsilon) =$



$A(a) =$



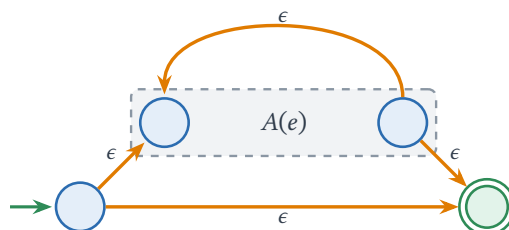
$A(ee') =$



$A(e|e') =$



$A(e^*) =$

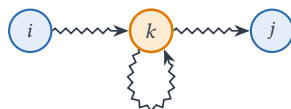


Méthode de McNaughton-Yamada $Q = \llbracket 1, n \rrbracket$, état initial 1.

$R_{ij}^{(k)}$ = expression régulière dénotant les étiquettes des chemins $i \rightsquigarrow j$ dont les états intermédiaires sont $\leq k$.

$$R_{ii}^{(0)} = \epsilon \text{ si } i \rightarrow i \quad R_{ii}^{(0)} = \epsilon \mid a_1 \mid \dots \mid a_p \text{ si } i \xrightarrow{a_1} i, \dots, i \xrightarrow{a_p} i$$

$$i \neq j : R_{ij}^{(0)} = \emptyset \text{ si } i \not\rightarrow j \quad R_{ij}^{(0)} = a_1 \mid \dots \mid a_p \text{ si } i \xrightarrow{a_1} j, \dots, i \xrightarrow{a_p} j$$



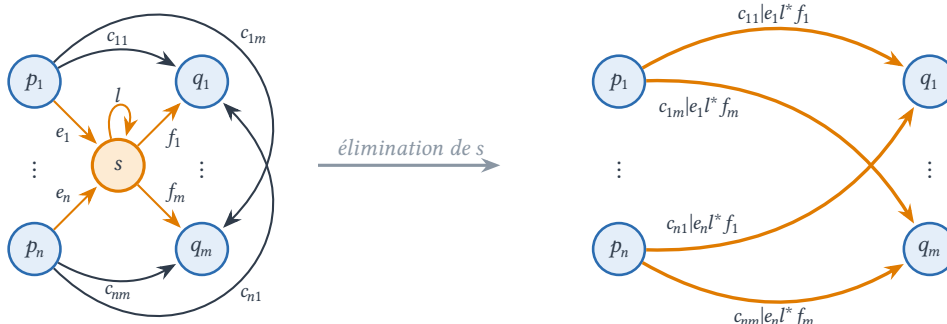
$$R^{(k)}_{ij} = R^{(k-1)}_{ij} \mid R^{(k-1)}_{ik} R^{(k-1)*}_{kk} R^{(k-1)}_{kj}$$

Si $F = \{f_1, \dots, f_p\}$:

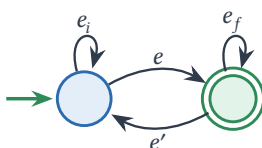
$$R^{(n)} = R^{(n-1)}_{f_1} \mid \dots \mid R^{(n-1)}_{f_p}$$

Élimination des états - Brzowski-McCluskey EX-AF(N)D : automates avec des expressions régulières comme étiquettes.

Élimination de s :



On aboutit à :



Expression régulière associée : $(e_i \mid e e_f^* e')^* e e_f^*$

Théorème 12.2 — de Kleene

régulier (ou rationnel) : dénoté par une expression régulière

reconnaissable : reconnu par un AFD

Théorème : un langage est **régulier** si et seulement s'il est **reconnaissable par un AFD** (ou par un AFND, ou par un AFND ϵ).

Stabilité des langages réguliers

Les langages réguliers sont stables par union, intersection, complémentaire, étoile, concaténation, miroir.

Lemme 12.3 — de l'étoile

Si L est régulier, il existe $n \in \mathbb{N}^*$ tel que tout $w \in L$ avec $|w| \geq n$ s'écrit $w = xyz$ avec $y \neq \epsilon$, $|xy| \leq n$ et $xy^*z \subset L$.

Contraposée : soit L un langage sur Σ . Si pour tout $n \in \mathbb{N}^*$, il existe $w \in L$ tel que $|w| \geq n$ et que pour tout $x, y, z \in \Sigma^*$ tels que $w = xyz$, $y \neq \epsilon$ et $|xy| \leq n$ il existe $k \in \mathbb{N}$ avec $xy^kz \notin L$, alors L n'est pas régulier.

Langages locaux

L local ssi $\exists P, S \subset \Sigma, N \subset \Sigma^2$,

$$L \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$$

Si L est local, on peut prendre :

$$P = \{a \in \Sigma \mid \exists w \in \Sigma^*, aw \in L\}$$

$$S = \{a \in \Sigma \mid \exists w \in \Sigma^*, wa \in L\}$$

$$\Sigma^2 \setminus N = \{ab \in \Sigma^2 \mid \exists w, v \in \Sigma^*, wabv \in L\}$$

Automates locaux

AFD local : si $q \xrightarrow{a} q_1$ et $q' \xrightarrow{a} q_2$ alors $q_1 = q_2$.

AFD standard : aucune transition n'arrive sur l'état initial.

langage local $\Leftrightarrow$ reconnu par un AFD local $\Leftrightarrow$ reconnu par un AFD local et standard.

Si P, S, N sont donnés : $A = (\Sigma \cup \{\epsilon\}, \Sigma, \delta, \epsilon, S')$ avec

$$S' = \begin{cases} S & \text{si } \epsilon \notin L \\ S \cup \{\epsilon\} & \text{sinon} \end{cases}$$

$$\forall a \in P, \delta(\epsilon, a) = a \quad \forall a, b \in \Sigma, ab \notin N \Rightarrow \delta(a, b) = b$$

Berry-Sethi

Expression régulière linéaire : chaque lettre apparaît au plus une fois.

Linéarisation : $a(a \mid b)cb^*a \Rightarrow a_1(a_2 \mid b_1)cb_2^*a_3$

e linéaire $\Rightarrow$ lang (e) local : on calcule P, S, N par induction, on construit l'AFD local, puis on efface les indices pour obtenir un AFND reconnaissant l'expression de départ.

$$\text{accv}(\emptyset) = \perp \quad \forall a \in \Sigma, \text{accv}(a) = \perp \quad \text{accv}(\epsilon) = \top \quad \text{accv}(e^*) = \top$$

$$\text{accv}(e_1 \mid e_2) = \text{accv}(e_1) \vee \text{accv}(e_2) \quad \text{accv}(e_1 e_2) = \text{accv}(e_1) \wedge \text{accv}(e_2)$$

Pour e ne contenant pas $\emptyset$ (ou $e = \emptyset$) :

$$P(\emptyset) = \emptyset \quad \forall a \in \Sigma, P(a) = \{a\} \quad P(\epsilon) = \emptyset \quad P(e^*) = P(e)$$

$$P(e_1 \mid e_2) = P(e_1) \cup P(e_2) \quad P(e_1 e_2) = \begin{cases} P(e_1) \cup P(e_2) & \text{si } \text{accv}(e_1) \\ P(e_1) & \text{sinon} \end{cases}$$

$$S(\emptyset) = \emptyset \quad \forall a \in \Sigma, S(a) = \{a\} \quad S(\epsilon) = \emptyset \quad S(e^*) = S(e)$$

$$S(e_1 | e_2) = S(e_1) \cup S(e_2) \quad S(e_1 e_2) = \begin{cases} S(e_1) \cup S(e_2) & \text{si } \text{accv}(e_2) \\ S(e_2) & \text{sinon} \end{cases}$$

$$N(\emptyset) = N(\epsilon) = \Sigma^2 \quad \forall a \in \Sigma, N(a) = \Sigma^2 \quad N(e^*) = N(e) \setminus S(e)P(e)$$

$$N(e_1 | e_2) = N(e_1) \cap N(e_2) \quad N(e_1 e_2) = (N(e_1) \cap N(e_2)) \setminus S(e_1)P(e_2)$$

12.1.1 Schéma global

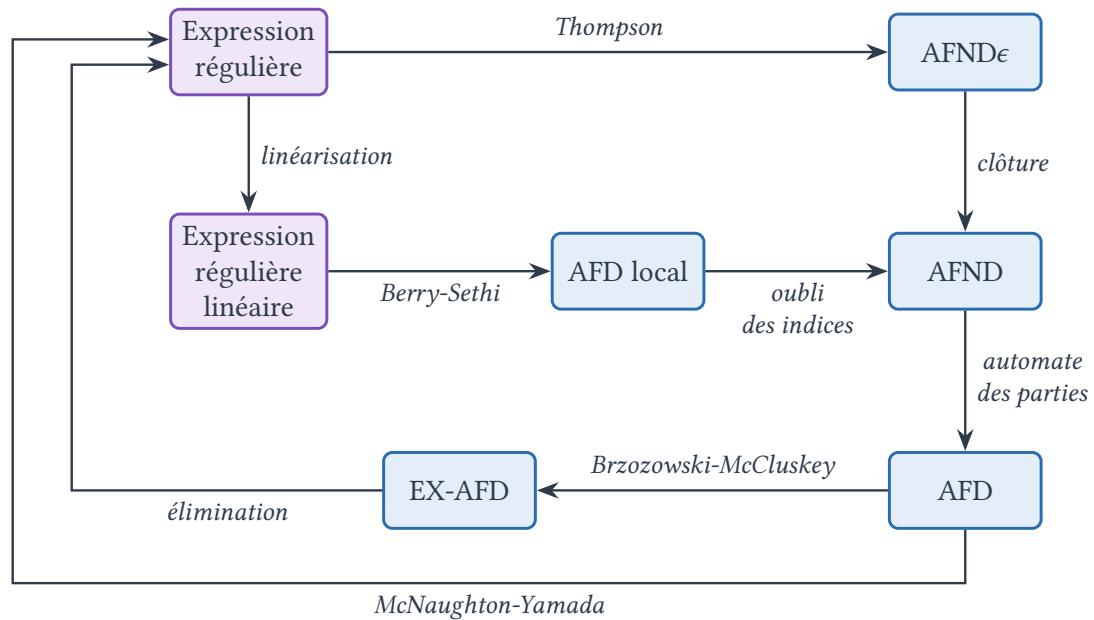


Figure 12.3 – Passages entre expressions régulières (violet) et automates (bleu)

Langages réguliers et automates finis

DANS CE CHAPITRE

13.1 Exercices

13.2 Exercices de colle

13.1 Exercices

13.1.1 Exercices d'application

Exercice

Si $L = \{a\}$ et $L' = \{b\}$, déterminer les langages suivants : L^* , L'^* , LL' , $L + L'$, $(LL')^*$, $(L + L')^*$, $L^*L'^*$, $L^* + L'^*$.

Exercice

Déterminer des expressions régulières dénotant les langages suivants.

1. L'ensemble des mots sur $\{a, b, c\}$ contenant au moins un a et au moins un b .
2. L'ensemble des mots sur $\{a, b\}$ dont la dixième lettre en partant de la fin est un b .
3. L'ensemble des mots sur $\{a, b\}$ contenant au plus deux b consécutifs.
4. L'ensemble des mots sur $\{a, b\}$ tels qu'aucun facteur aa ne soit précédé d'un facteur bb . *aabaabb* convient mais pas *aabbbaa*.
5. L'ensemble des mots sur $\{a, b\}$ dont le nombre de a est divisible par 5.

Exercice

Décrire informellement les langages dénotés par les expressions régulières suivantes.

1. $(a \mid \epsilon)(aa^*b)^*a^*$
2. $(a^*b^*)^*aaa(a \mid b)^*$
3. $(a \mid ba)^*b^*$

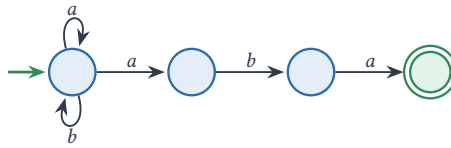
Exercice

Montrer qu'il n'existe que deux langages dont l'étoile est finie.

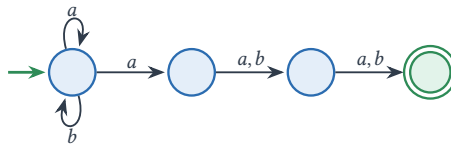
Exercice

Déterminer les automates suivants :

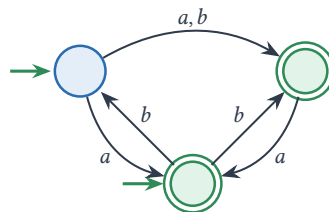
(a)



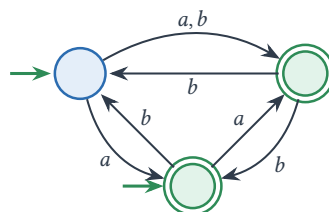
(b)



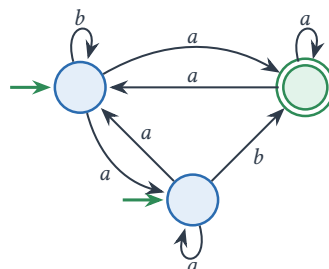
(c)



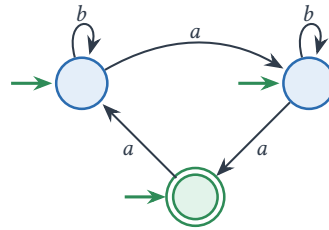
(d)



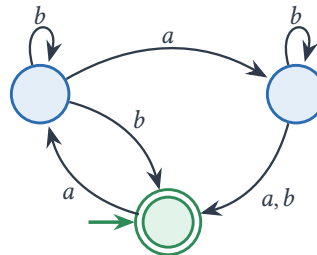
(e)



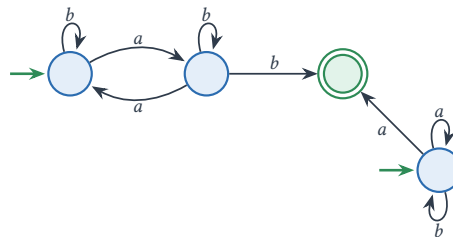
(f)



****(g)****



****(h)****



Exercice

Pour chacune des expressions régulières suivantes, déterminez un automate fini non déterministe avec transitions spontanées qui reconnaît le langage qu'elle dénote, éliminez les transitions spontanées pour en déduire un automate non déterministe, puis, déterminez un automate directement par Berry-Sethi.

1. ab^*
2. $(a \mid b)ab$
3. $aa(a \mid b)^*$

Exercice

On considère la table de transition :

		a	b
$\rightarrow$	q_1	q_2	q_1
	q_2	q_3	q_1
*	q_3	q_3	q_2

1. Déterminer la matrice des expressions régulières $R^{(0)}$.
2. Déterminer la matrice des expressions régulières $R^{(1)}$ en la simplifiant.
3. Déterminer la matrice des expressions régulières $R^{(2)}$ en la simplifiant.
4. En déduire une expression régulière dénotant le langage reconnu par l'automate.
5. Appliquer la méthode d'élimination des états pour obtenir une expression régulière.

Exercice

Montrer que les langages suivants ne sont pas réguliers.

1. L'ensemble des expressions bien parenthésées : P défini par induction avec $\epsilon \in P$, $\forall e_1, e_2 \in P, e_1 e_2 \in P$ et $\forall e \in P, (e) \in P$.
2. $\{a^n b c^n \mid n \geq 1\}$.
3. $\{a^n b^m c^n \mid n, m \in \mathbb{N}\}$
4. $\{a^n b^m \mid n \leq m\}$
5. $\{a^n b^{2n} \mid n \geq 1\}$

Exercice

Même exercice que le précédent, mais les preuves sont plus délicates.

1. $\{a^{n^2} \mid n \in \mathbb{N}\}$
2. $\{a^{2^n} \mid n \in \mathbb{N}\}$
3. $\{w \in \Sigma^* \mid \exists n \in \mathbb{N}, |w| = n^2\}$
4. $\{w^2 \mid w \in \Sigma^*\}$, où Σ contient au moins deux lettres
5. $\{w \in \{0, 1\}^* \mid w \text{ est l'écriture d'un premier en binaire}\}$
6. $\{a^i b^j \mid i \wedge j = 1\}$.

13.1.2 Lemme d'Arden**Théorème 13.1**

Soit A, B deux langages sur Σ .

Le langage $L = A^* B$ est le plus petit langage solution de l'équation

$$X = AX \cup B$$

De plus, si $\epsilon \notin A$, cette solution est unique.

Exercice

Démontrer ce théorème.

13.1.3 Explosion exponentielle de la détermination**Exercice**

On va montrer dans cet exercice qu'il est possible de trouver un automate à $n + 1$ états dont le déterminisé par l'automate des parties, même émondé, contient au moins 2^n états.

1. Soit $n \in \mathbb{N}^*$, définir un automate fini non-déterministe à $n + 1$ états sur l'alphabet $\{a, b\}$ qui reconnaît les mots de la forme waw' avec $|w'| = n - 1$, c'est-à-dire les mots qui ont un a en n -ième position en partant de la fin.
2. Montrer que le déterminisé de cet automate contient au moins 2^n états.

13.1.4 Miroir d'un langage**Exercice**

Soit Σ un alphabet fini.

Si $w = a_1 \dots a_n \in \Sigma^*$ est un mot non vide, on note $w^R = a_n \dots a_1$ le mot w lu à l'envers, et on pose $\epsilon^R = \epsilon$. On l'appelle le miroir de w .

Soit $L \subset \Sigma^*$ un langage. On note $L^R = \{w^R \mid w \in L\}$ le miroir de L .

1. On suppose que Σ contient au moins deux lettres. Montrer que le langage des palindromes

$$P = \{w \in \Sigma^* \mid w = w^R\}$$

n'est pas régulier. Que se passe-t-il si Σ ne contient qu'une lettre ?

2. On va donner deux preuves du fait que les langages réguliers sont stables par miroir.

1. Soit L un langage régulier et A un automate reconnaissant L , en déduire un automate non déterministe avec transitions spontanées A^R , en renversant les transitions, permettant de reconnaître L^R .
2. Soit e une expression régulière, déterminer par induction une expression régulière e^R dénotant $\text{lang}(e)^R$.

13.2 Exercices de colle

13.2.1 Moitié d'un langage

Exercice

Soit $L \subset \Sigma^*$, on note

$$\frac{1}{2}L = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, |u| = |v| \text{ et } uv \in L\}$$

Montrer que si L est régulier, alors $\frac{1}{2}L$ est régulier.

Preuve Soit $A = (Q, \Sigma, \delta, i, F)$ un automate déterministe reconnaissant L . L'idée ici est que si $u \in \frac{1}{2}L$ et que $\delta^*(i, u) = q$, unique car A est déterministe, alors il existe un chemin de longueur $|u|$ tel que $q \rightsquigarrow q' \in F$.

Donc, pour déterminer un automate reconnaissant $\frac{1}{2}L$, on peut simuler **tous les chemins possibles** depuis tous les états en même temps qu'on lit u . Quand on a fini de lire, on n'a qu'à vérifier si dans les chemins énumérés il y en a un issu de q qui aboutit sur un état dans F .

On va ainsi noter $Q = \{q_1, \dots, q_n\}$ et supposer que $i = q_1$. On suppose de plus que A est complet, quitte à ajouter un état puits.

On pose l'automate non déterministe

$$A' = (Q \times Q^n, \Sigma, \delta', \{(q_1, (q_1, \dots, q_n))\}, F')$$

avec

$$\forall (q, (s_1, \dots, s_n)) \in Q \times Q^n, \forall a \in \Sigma, \delta'((q, (s_1, \dots, s_n)), a) = \{(\delta(q, a), (\delta(s_1, a), \dots, \delta(s_n, a))) \mid a \in \Sigma\}$$

ce qui revient à faire exactement comme A sur la première composante et à faire tous les chemins possibles sur la seconde de manière non déterministe. On remarque, par une récurrence immédiate sur $|u|$, que

$$\delta'^*((q_1, (q_1, \dots, q_n)), u) = \{(\delta^*(q_1, u), (\delta^*(q_1, v), \dots, \delta^*(q_n, v))) \mid v \in \Sigma^*, |v| = |u|\}$$

Pour les états finaux, on pose

$$F' = \{(q_j, (s_1, \dots, s_n)) \mid j \in \llbracket 1, n \rrbracket, s_j \in F\}$$

ce sont les états tels qu'on puisse atteindre un état final depuis ceux-ci en prenant le même temps que pour arriver sur l'état.

Soit $u \in \Sigma^*$,

$$\begin{aligned}
u \in \text{lang}(A') &\Leftrightarrow \delta'^*((q_1, (q_1, \dots, q_n)), u) \cap F' \neq \emptyset \\
&\Leftrightarrow \exists v \in \Sigma^*, |v| = |u| \text{ et } (\delta^*(q_1, u), (\delta^*(q_1, v), \dots, \delta^*(q_n, v))) \in F' \\
&\Leftrightarrow \exists v \in \Sigma^*, |v| = |u| \text{ et } \delta^*(\delta^*(q_1, u), v) \in F \\
&\Leftrightarrow \exists v \in \Sigma^*, |v| = |u| \text{ et } \delta^*(q_1, uv) \in F \\
&\Leftrightarrow \exists v \in \Sigma^*, |v| = |u| \text{ et } uv \in L \\
&\Leftrightarrow u \in \frac{1}{2}L
\end{aligned}$$

□

Exercice

Soit $L, L' \subset \Sigma^*$, on note

$$L : L' = \{u \in L \mid \exists v \in L', |u| = |v|\}$$

Montrer que si L et L' sont réguliers, alors $L : L'$ est régulier.

Preuve C'est un peu comme l'exercice précédent, on va considérer $A = (Q, \Sigma, \delta, i, F)$ et $A' = (Q', \Sigma, \delta', i', F')$ déterministes et complets reconnaissant respectivement L et L' , et simuler tous les chemins possibles dans A' pendant qu'on lit un mot dans A . Si on peut aboutir en même temps sur deux états finaux, on aura reconnu le mot.

On pose donc l'automate non déterministe $A'' = (Q \times Q', \Sigma, \delta'', \{(i, i')\}, F \times F')$ et

$$\delta''((q, q'), a) = \{(\delta(q, a), \delta'(q', b)) \mid b \in \Sigma\}$$

On a ainsi, par une récurrence immédiate sur $|u|$, $\delta''^*((i, i'), u) = \{(\delta^*(i, u), \delta'^*(i', v)) \mid v \in \Sigma^*, |v| = |u|\}$, d'où

$$\begin{aligned}
u \in \text{lang}(A'') &\Leftrightarrow \delta''^*((i, i'), u) \cap F \times F' \neq \emptyset \\
&\Leftrightarrow \{(\delta^*(i, u), \delta'^*(i', v)) \mid v \in \Sigma^*, |v| = |u|\} \cap F \times F' \neq \emptyset \\
&\Leftrightarrow \exists v \in \Sigma^*, |v| = |u| \text{ et } \delta^*(i, u) \in F, \delta'^*(i', v) \in F' \\
&\Leftrightarrow u \in L \text{ et } \exists v \in L', |u| = |v| \\
&\Leftrightarrow u \in L : L'
\end{aligned}$$

□

Exercice

Soit $L \subset \Sigma^*$.

1. Soit $a \in \Sigma$, on pose $L \setminus a = \{w \in \Sigma^* \mid wa \in L\}$.
2. $\min(L) = \{w \in L \mid \forall w = uv, v \neq \epsilon \Rightarrow u \notin L\}$ les mots de L dont aucun préfixe propre n'est dans L .
3. $\max(L) = \{w \in L \mid \forall u \in \Sigma^*, u \neq \epsilon \Rightarrow wu \notin L\}$ les mots de L qui ne sont préfixes propres d'aucun mot de L .

Montrer que si L est régulier, alors $L \setminus a$, $\min(L)$ et $\max(L)$ sont réguliers.

Preuve Dans tous les cas, on va procéder de la même manière : on part d'un AFD complet A reconnaissant L et on en déduit un AFD A' reconnaissant le langage qui nous intéresse.

Soit $A = (Q, \Sigma, \delta, i, F)$ reconnaissant L .

Construction 1 On pose $A' = (Q, \Sigma, \delta, i, F')$ où

$$F' = \{q \in Q \mid \delta(q, a) \in F\}$$

Montrons que $\text{lang}(A') = L \setminus a$:

Soit $w \in \Sigma^*$,

$$\begin{aligned} w \in \text{lang}(A') &\Leftrightarrow \delta^*(i, w) \in F' \\ &\Leftrightarrow \delta(\delta^*(i, w), a) = \delta^*(i, wa) \in F \\ &\Leftrightarrow wa \in L \\ &\Leftrightarrow w \in L \setminus a \end{aligned}$$

On a bien $\text{lang}(A') = L \setminus a$ donc ce langage est régulier.

Construction 2 On pose $A' = (Q, \Sigma, \delta', i, F)$ où

$$\forall q \in Q \setminus F, \forall a \in \Sigma, \delta'(q, a) = \delta(q, a)$$

et $\delta'(q, a)$ n'est pas défini pour $q \in F$. On supprime ainsi les transitions issues d'un état final.

Comme $q \rightarrow q'$ dans A' entraîne $q \notin F$, on remarque qu'un chemin dans A' passe au plus par un état final, et nécessairement comme dernier état du chemin. Ainsi, $\delta'^*(i, w)$ est défini et dans F si et seulement si $\delta^*(i, w) \in F$ et aucun état $\delta^*(i, u)$, pour u préfixe propre de w , n'est dans F .

Soit $w \in \Sigma^*$,

$$\begin{aligned} w \in \text{lang}(A') &\Leftrightarrow \delta'^*(i, w) \in F \\ &\Leftrightarrow \delta^*(i, w) \in F \text{ et } \forall w = uv, v \neq \epsilon \Rightarrow \delta^*(i, u) \notin F \\ &\Leftrightarrow w \in L \text{ et } \forall w = uv, v \neq \epsilon \Rightarrow u \notin L \\ &\Leftrightarrow w \in \min(L) \end{aligned}$$

On a bien $\text{lang}(A') = \min(L)$ donc ce langage est régulier.

Construction 3 On pose $A' = (Q, \Sigma, \delta, i, F')$ où

$$F' = \{q \in F \mid \forall u \in \Sigma^*, u \neq \epsilon \Rightarrow \delta^*(q, u) \notin F\}$$

les états finaux depuis lesquels aucun chemin non vide ne mène à un état final (y compris q lui-même : un état final sur un cycle n'est pas dans F').

Soit $w \in \Sigma^*$, on va noter $q = \delta^*(i, w)$.

Supposons $q \in F'$ et soit $u \in \Sigma^*$ tel que $u \neq \epsilon$. Par définition de F' , $\delta^*(q, u) = \delta^*(i, wu) \notin F$ donc $wu \notin L$. Comme de plus $q \in F$, on a $w \in L$ et ainsi $w \in \max(L)$.

Réciproquement, si $w \in \max(L)$, on a $q \in F$ car $w \in L$. Soit $u \in \Sigma^*$ tel que $u \neq \epsilon$: on a $\delta^*(q, u) = \delta^*(i, wu)$ or $wu \notin L$ par hypothèse sur w . Donc $\delta^*(q, u) \notin F$. On vient de montrer que $q \in F'$.

On a donc,

$$w \in \text{lang}(A') \Leftrightarrow q \in F' \Leftrightarrow w \in \max(L)$$

On a bien $\text{lang}(A') = \max(L)$ donc ce langage est régulier.

□

Berry-Sethi en C

Ce TP met en œuvre la [construction de l'automate de Glushkov par l'algorithme de Berry-Sethi](#) vue en cours. On y reprend les notations du cours : pour une expression régulière e , $accv(e)$ indique si $\epsilon \in \text{lang}(e)$, $P(e)$ et $S(e)$ sont les ensembles des premières et dernières lettres des mots de $\text{lang}(e)$, et $N(e)$ l'ensemble des facteurs de longueur 2 qui n'apparaissent dans aucun mot de $\text{lang}(e)$.

On considère des expressions régulières représentées sous forme de chaînes de caractères de manière postfixe avec :

- le caractère `'\0'` comme terminateur de la chaîne ;
- des constantes pour les opérations (PLUS représente l'union, notée $|$ dans le cours, et EPSILON l'expression ϵ) :

```
char ETOILE = '*';
char PLUS = '+';
char POINT = '.';
char EPSILON = '~';
```

- les lettres minuscules entre 'a' et 'z' forment l'alphabet.

Il n'y a pas de symbole pour $\emptyset$: comme dans le cours, on suppose que les expressions ne contiennent pas $\emptyset$.

Exemple 14.1

L'expression régulière $(ac)^* | b$ sera définie par :

```
char s[] = "ac.*b+";
```

Question 1 Représenter $b^*(a | a^*b)$.

Corrigé

"b*aa*b.+." : on écrit b^* , puis a , puis a^*b ("a*b."), on fait l'union des deux derniers ('+') et enfin la concaténation avec b^* ('.').

On va calculer les ensembles P, S et $\Sigma^2 \setminus N$ associés à une expression régulière en représentant une partie de Σ par un tableau de 26 booléens et une partie de Σ^2 par un tableau de 26^2 booléens (une matrice à plat).

On regroupe ces informations dans une structure :

```
#define NL 26 // nombre de lettres

struct infos {
    bool vide;           // l'expression reconnaît-elle le mot vide ?
    bool p[NL];          // P(e)
    bool s[NL];          // S(e)
    bool f[NL * NL];     // Σ² \ N(e) : f[26 * a + b] pour le facteur ab
```

```
};
typedef struct infos infos;
```

Question 2 Écrire une fonction `infos calcule(char *es)` qui calcule $accv(e)$, $P(e)$, $S(e)$ et $\Sigma^2 \setminus N(e)$ pour l'expression postfixe `es`, à l'aide des formules inductives du cours.

Indication : comme pour l'évaluation d'une expression postfixe (voir le TP sur les piles), on utilise une pile de `infos` : une lettre ou `EPSILON` empile ses informations, une opération dépile ses arguments et empile le résultat. Pour $\Sigma^2 \setminus N$, les formules du cours donnent, en passant au complémentaire :

$$\Sigma^2 \setminus N(e_1 \mid e_2) = (\Sigma^2 \setminus N(e_1)) \cup (\Sigma^2 \setminus N(e_2))$$

$$\Sigma^2 \setminus N(e_1 e_2) = (\Sigma^2 \setminus N(e_1)) \cup (\Sigma^2 \setminus N(e_2)) \cup S(e_1)P(e_2) \quad \Sigma^2 \setminus N(e^*) = (\Sigma^2 \setminus N(e)) \cup S(e)P(e)$$

Corrigé

```
// calcule les informations de l'expression postfixe es
infos calcule(char *es)
{
    int n = strlen(es);
    infos *pile = malloc(n * sizeof(infos));
    int sommet = 0; // nombre d'éléments dans la pile
    for (int i = 0; i < n; i++)
    {
        char c = es[i];
        infos r;
        if ('a' <= c && c <= 'z')
        {
            memset(&r, 0, sizeof(infos)); // tout à false
            r.p[c - 'a'] = true;
            r.s[c - 'a'] = true;
        }
        else if (c == EPSILON)
        {
            memset(&r, 0, sizeof(infos));
            r.vide = true;
        }
        else if (c == ETOILE)
        {
            assert(sommet >= 1);
            r = pile[sommet - 1];
            sommet--;
            r.vide = true;
            for (int a = 0; a < NL; a++)
                for (int b = 0; b < NL; b++)
                    if (r.s[a] && r.p[b])
                        r.f[NL * a + b] = true;
        }
        else
        {
            assert(sommet >= 2 && (c == PLUS || c == POINT));
            infos e1 = pile[sommet - 2];
            infos e2 = pile[sommet - 1];
            sommet -= 2;
            for (int k = 0; k < NL * NL; k++)
                r.f[k] = e1.f[k] || e2.f[k];
            if (c == PLUS)
            {
                r.vide = e1.vide || e2.vide;
                for (int a = 0; a < NL; a++)
                {
                    r.p[a] = e1.p[a] || e2.p[a];
                    r.s[a] = e1.s[a] || e2.s[a];
                }
            }
        }
    }
}
```

```

        else
        {
            r.vide = e1.vide && e2.vide;
            for (int a = 0; a < NL; a++)
            {
                r.p[a] = e1.p[a] || (e1.vide && e2.p[a]);
                r.s[a] = e2.s[a] || (e2.vide && e1.s[a]);
            }
            for (int a = 0; a < NL; a++)
                for (int b = 0; b < NL; b++)
                    if (e1.s[a] && e2.p[b])
                        r.f[NL * a + b] = true;
        }
        pile[sommet] = r;
        sommet++;
    }
    assert(sommet == 1); // expression bien formée
    infos res = pile[0];
    free(pile);
    return res;
}

```

Pour "ac.*b+", on obtient $accv = \top$, $P = \{a, b\}$, $S = \{b, c\}$ et $\Sigma^2 \setminus N = \{ac, ca\}$.

D'après le théorème de caractérisation du cours, si e est **linéaire** (chaque lettre y apparaît au plus une fois), $\text{lang}(e)$ est local et $\text{lang}(e) \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^*S) \setminus \Sigma^*N\Sigma^*$.

Question 3 Écrire une fonction `bool reconnait_local(infos *inf, char *w)` qui teste si le mot w appartient au langage local défini par ces ensembles (et par $accv$ pour le mot vide). Vérifier sur des exemples qu'elle reconnaît $\text{lang}(e)$ quand e est linéaire, et trouver une expression non linéaire pour laquelle ce n'est pas le cas.

Corrigé

```

// teste si w est dans le langage local défini par inf
bool reconnait_local(infos *inf, char *w)
{
    int n = strlen(w);
    if (n == 0)
        return inf->vide;
    if (!inf->p[w[0] - 'a'] || !inf->s[w[n - 1] - 'a'])
        return false;
    for (int i = 0; i + 1 < n; i++)
        if (!inf->f[NL * (w[i] - 'a') + (w[i + 1] - 'a')])
            return false;
    return true;
}

```

Pour $e = a(aa)^*$, c'est-à-dire "aaa.*", on obtient $P = S = \{a\}$ et $\Sigma^2 \setminus N = \{aa\}$: le mot aa est accepté alors qu'il n'est pas dans $\text{lang}(e)$, qui n'est pas local.

Pour une expression quelconque, on la **linéarise** en numérotant les occurrences de lettres de 0 à $n - 1$ (dans l'ordre de la chaîne) : ce sont les *positions*. On calcule alors P , S et $\Sigma^2 \setminus N$ sur l'alphabet des positions, avec des tableaux de taille n et n^2 alloués dynamiquement. L'automate de Glushkov a pour états un état initial et les n positions ; il a une transition de l'état initial vers $i \in P$ et de i vers j pour $ij \in \Sigma^2 \setminus N$, étiquetées par la lettre de la position d'arrivée ; ses états finals sont les positions de S , et l'état initial si $accv(e)$.

Question 4 Écrire une fonction `glushkov linearise(char *es)` qui calcule ces informations, puis une fonction `bool reconnait(glushkov *g, char *w)` qui teste si w est reconnu par l'automate de Glushkov, en simulant l'automate non déterministe par l'ensemble des états atteints.

Corrigé

Comme les positions de deux sous-expressions sont disjointes, on peut remplir un unique tableau global de facteurs au fur et à mesure.

```

struct glushkov {
    int n;           // nombre de positions (occurrences de lettres)
    char *lettre;    // lettre[i] : lettre de la position i
    bool vide;
    bool *p;        // p[i] : la position i est dans P
    bool *s;        // s[i] : la position i est dans S
    bool *f;        // f[n * i + j] : le facteur ij est possible
};
typedef struct glushkov glushkov;

// informations sur une sous-expression, pour des positions
// stockées dans des tableaux de taille n alloués dynamiquement
struct infos_pos {
    bool vide;
    bool *p;
    bool *s;
};
typedef struct infos_pos infos_pos;

glushkov linearise(char *es)
{
    int m = strlen(es);
    glushkov g;
    // on numérote les occurrences de lettres : c'est la linéarisation
    g.n = 0;
    for (int i = 0; i < m; i++)
        if ('a' <= es[i] && es[i] <= 'z')
            g.n++;
    int n = g.n;
    g.lettre = malloc(n * sizeof(char));
    g.f = calloc(n * n + 1, sizeof(bool));
    infos_pos *pile = malloc(m * sizeof(infos_pos));
    int sommet = 0;
    int pos = 0; // prochaine position à attribuer
    for (int i = 0; i < m; i++)
    {
        char c = es[i];
        infos_pos r;
        if (c == ETOILE)
        {
            r = pile[--sommet];
            r.vide = true;
            for (int a = 0; a < n; a++)
                for (int b = 0; b < n; b++)
                    if (r.s[a] && r.p[b])
                        g.f[n * a + b] = true;
        }
        else if (c == PLUS || c == POINT)
        {
            infos_pos e2 = pile[--sommet];
            infos_pos e1 = pile[--sommet];
            // les ensembles de positions de e1 et e2 sont disjointes :
            // on peut calculer les facteurs dans un tableau global
            if (c == POINT)
                for (int a = 0; a < n; a++)
                    for (int b = 0; b < n; b++)
                        if (e1.s[a] && e2.p[b])
                            g.f[n * a + b] = true;
            r.p = e1.p;
            r.s = e1.s;
            for (int a = 0; a < n; a++)
            {

```

```

        if (c == PLUS)
        {
            r.p[a] = e1.p[a] || e2.p[a];
            r.s[a] = e1.s[a] || e2.s[a];
        }
        else
        {
            r.p[a] = e1.p[a] || (e1.vide && e2.p[a]);
            r.s[a] = e2.s[a] || (e2.vide && e1.s[a]);
        }
    }
    r.vide = (c == PLUS) ? (e1.vide || e2.vide) : (e1.vide && e2.vide);
    free(e2.p);
    free(e2.s);
}
else
{
    r.p = calloc(n + 1, sizeof(bool));
    r.s = calloc(n + 1, sizeof(bool));
    r.vide = (c == EPSILON);
    if (c != EPSILON)
    {
        g.lettre[pos] = c;
        r.p[pos] = true;
        r.s[pos] = true;
        pos++;
    }
}
pile[sommet++] = r;
}
assert(sommet == 1);
g.vide = pile[0].vide;
g.p = pile[0].p;
g.s = pile[0].s;
free(pile);
return g;
}

void libere_glushkov(glushkov *g)
{
    free(g->lettre);
    free(g->p);
    free(g->s);
    free(g->f);
}

// Automate de Glushkov : un état initial et un état par position.
// On simule l'automate non déterministe en maintenant l'ensemble
// des positions atteintes (l'état initial n'est actif qu'au départ).
bool reconnait(glushkov *g, char *w)
{
    int n = g->n;
    int m = strlen(w);
    if (m == 0)
        return g->vide;
    bool *cour = malloc((n + 1) * sizeof(bool));
    bool *suiv = malloc((n + 1) * sizeof(bool));
    for (int i = 0; i < n; i++)
        cour[i] = g->p[i] && g->lettre[i] == w[0];
    for (int k = 1; k < m; k++)
    {
        for (int j = 0; j < n; j++)
        {
            suiv[j] = false;
            if (g->lettre[j] == w[k])
                for (int i = 0; i < n && !suiv[j]; i++)
                    suiv[j] = cour[i] && g->f[n * i + j];
        }
    }
}

```

```
    }  
    bool *t = cour; cour = suiv; suiv = t;  
}  
bool res = false;  
for (int i = 0; i < n; i++)  
    res = res || (cour[i] && g->s[i]);  
free(cour);  
free(suiv);  
return res;  
}
```

On a testé ces fonctions en comparant, pour des milliers d'expressions aléatoires sur $\{a, b, c\}$ et tous les mots de longueur au plus 5, le résultat de `reconnait` avec un filtrage naïf sur l'arbre de l'expression.

Grammaires non contextuelles

DANS CE CHAPITRE

15.1 Introduction

15.2 Grammaire non contextuelle

15.3 Analyse syntaxique par descente récursive

15.1 Introduction

Ce chapitre fait directement suite au chapitre sur les langages réguliers.

Si on considère un programme OCaml comme :

```
let f x =
  let y = x + 3 in
  x * y
```

On a appris à détecter des suites de caractères particulières, on pourrait notamment identifier dans le programme précédent :

- des mots-clés comme `let` ou `in` ;
- des opérateurs comme `=`, `*`, `+` ;
- des identificateurs comme `f`, `x` ou `y` ;
- des nombres comme `3`.

Ces types correspondent à des langages réguliers. Dans le contexte de la reconnaissance de syntaxe, on parle de *lexèmes* (ou de *token* en anglais) et un programme qui découpe une suite de caractères en une liste de lexèmes est appelé un *analyseur lexical* (*lexer* en anglais).

Ici, on pourrait considérer un type OCaml `token` :

```
type token =
  | Keyword of string
  | Identifier of string
  | Number of int
  | Equals | Plus | Times
```

Et le programme précédent deviendrait alors la liste :

```
[ Keyword "let"; Identifier "f"; Identifier "x";
  Equals; Keyword "let"; Identifier "y";
  Equals; Identifier "x"; Plus; Number 3; Keyword "in";
  Identifier "x"; Times; Identifier "y" ]
```

Cependant, il ne suffit pas de savoir que les éléments sont bien des lexèmes. En effet, $x + 3$ est une expression mais $+ \times 3$ n'en est pas une car on considère que $+$ est un opérateur infixé.

La question qu'on se pose maintenant est de savoir comment décider ce qui est une phrase valide du langage ou non. Il s'agit ainsi de raisonner sur la manière dont les lexèmes sont positionnés les uns par rapport aux autres : c'est la grammaire.

Tout d'abord, on peut commencer par abstraire le contenu des lexèmes car au moment où on analyse la phrase, il n'est pas important. On peut alors essayer de reconnaître par des règles d'inférence les phrases valides :

$$\frac{\text{ident}}{\text{expr}} \quad \frac{\text{number}}{\text{expr}} \quad \frac{\text{expr} \quad \frac{+}{\text{expr}} \quad \text{expr}}{\text{expr}}$$

De gauche à droite, ces règles signifient :

- qu'un identificateur est une expression valide ;
- qu'un nombre est une expression valide ;
- que deux expressions séparées par un $+$ forment une expression valide.

On peut alors construire un arbre de dérivation comme dans le chapitre sur la déduction naturelle.

$$\frac{\frac{\text{ident}}{\text{expr}} \quad + \quad \frac{\text{number}}{\text{expr}}}{\text{expr}}$$

Figure 15.1 — Arbre de dérivation de $\text{ident} + \text{number}$

Il est traditionnel de voir cet arbre à plat dans le contexte des grammaires comme une suite de raffinements successifs permettant de passer de ce que l'on veut lire : une expression, à la suite des lexèmes qui la constituent :

$$\text{expr} \Rightarrow \text{expr} + \text{expr} \Rightarrow \text{ident} + \text{expr} \Rightarrow \text{ident} + \text{number}$$

On va ainsi s'intéresser dans les prochains paragraphes à la définition précise d'une grammaire non contextuelle et de son langage, ainsi qu'à la manière dont on peut en déduire un programme permettant d'effectuer cette reconnaissance.

15.2 Grammaire non contextuelle

15.2.1 Définition



Dans l'introduction, on a manipulé deux sortes de symboles : les lexèmes, qui apparaissent effectivement dans la phrase, et des symboles comme *expr* qui désignent une catégorie de phrases et qu'on remplace progressivement. Les règles d'inférence deviennent des règles de réécriture.

Définition 15.1

Une **grammaire non contextuelle** est un quadruplet $G = (\Sigma, V, P, S)$ où :

- Σ est un alphabet fini dont les éléments sont appelés les symboles **terminaux** ;
- V est un ensemble fini, disjoint de Σ , dont les éléments sont appelés les symboles **non terminaux** (ou **variables**) ;
- $P \subset V \times (\Sigma \cup V)^*$ est un ensemble fini de **règles de production**. On note $A \rightarrow \alpha$ la règle (A, α) ;
- $S \in V$ est le **symbole initial**.

Remarque 15.2

- Quand plusieurs règles ont le même membre gauche, $A \rightarrow \alpha_1, \dots, A \rightarrow \alpha_k$, on les regroupe en écrivant $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$.

- Par convention, les non terminaux sont notés avec des majuscules et les terminaux avec des minuscules ou des symboles. Le plus souvent, on se contente de donner les règles, le symbole initial étant le membre gauche de la première règle.
- *Non contextuelle* signifie que le membre gauche d'une règle est un unique non terminal : on peut remplacer A par α quel que soit ce qui l'entoure, c'est-à-dire son contexte.

Exemple 15.3

- La grammaire de l'introduction s'écrit, en notant E le symbole $expr : E \rightarrow E + E \mid id \mid num$. Ici $\Sigma = \{+, id, num\}$, $V = \{E\}$ et $S = E$.
- Sur $\Sigma = \{a, b\}$, on considère la grammaire $S \rightarrow aSb \mid \epsilon$ qui possède deux règles : $S \rightarrow aSb$ et $S \rightarrow \epsilon$.
- La grammaire $expr \rightarrow num \mid num + expr$ du TP sur l'analyse des expressions est donnée sous la forme textuelle $\tau = \dots$, $NT = \dots$, $s = \dots$ qui liste explicitement Σ , V et S .

15.2.2 Dérivation et langage**Définition 15.4**

Soit $G = (\Sigma, V, P, S)$ une grammaire et $u, v \in (\Sigma \cup V)^*$.

- On dit que u se **dérive immédiatement** en v , et on note $u \Rightarrow v$, lorsqu'il existe $x, y \in (\Sigma \cup V)^*$ et une règle $A \rightarrow \alpha \in P$ tels que $u = xAy$ et $v = x\alpha y$.
- On note $u \xRightarrow{k} v$ lorsqu'il existe $u_0 = u, u_1, \dots, u_k = v$ avec $u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_k$: c'est une **dérivation** de longueur k .
- On note $u \xRightarrow{*} v$ la clôture réflexive et transitive de $\Rightarrow : u \xRightarrow{*} v$ si et seulement s'il existe $k \in \mathbb{N}$ tel que $u \xRightarrow{k} v$.

Définition 15.5

Le **langage engendré** par la grammaire G est

$$\text{lang}(G) = \{w \in \Sigma^* \mid S \xRightarrow{*} w\}$$

Un langage est dit **non contextuel** (ou **algébrique**) lorsqu'il est engendré par une grammaire non contextuelle. Deux grammaires engendrant le même langage sont dites **faiblement équivalentes**.

Remarque 15.6 Seuls les mots de Σ^* , donc **sans** non terminal, sont dans le langage. Les mots intermédiaires de $(\Sigma \cup V)^*$ apparaissant dans une dérivation sont parfois appelés des *proto-mots*.

Exemple 15.7

Avec la grammaire $S \rightarrow aSb \mid \epsilon$, on a $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$, donc $aabb$ est dans le langage engendré.

Comme les règles s'appliquent indépendamment du contexte, une dérivation à partir d'un mot $X_1 \dots X_n$ se décompose en n dérivations indépendantes. Ce lemme est l'outil de base pour raisonner sur les grammaires non contextuelles.

Lemme 15.8

Soient $X_1, \dots, X_n \in \Sigma \cup V$, $w \in \Sigma^*$ et $k \in \mathbb{N}$. On a $X_1 \dots X_n \xRightarrow{k} w$ si et seulement s'il existe $w_1, \dots, w_n \in \Sigma^*$ et $k_1, \dots, k_n \in \mathbb{N}$ tels que $w = w_1 \dots w_n$, $k = k_1 + \dots + k_n$ et pour tout $i \in \llbracket 1, n \rrbracket$, $X_i \xRightarrow{k_i} w_i$.

Preuve Le sens réciproque s'obtient en enchaînant les dérivations $X_1 \dots X_n \xRightarrow{k_1} w_1 X_2 \dots X_n \xRightarrow{k_2} \dots \xRightarrow{k_n} w_1 \dots w_n$. Pour le sens direct, on procède par récurrence sur k (pour tous les mots $X_1 \dots X_n$ simultanément).

- Si $k = 0$, $X_1 \dots X_n = w$ ne contient que des terminaux et on pose $w_i = X_i$, $k_i = 0$.

- Si $k > 0$, la première étape remplace un certain $X_j = A$ par $\alpha = Y_1 \dots Y_p$ avec $A \rightarrow \alpha \in P$: $X_1 \dots X_{j-1} Y_1 \dots Y_p X_{j+1} \dots X_n \xRightarrow{k-1} w$. Par hypothèse de récurrence, w se découpe en $w_1 \dots w_{j-1} v_1 \dots v_p w_{j+1} \dots w_n$ avec des dérivations de chaque symbole dont les longueurs sont de somme $k-1$. On pose $w_j = v_1 \dots v_p$: par le sens réciproque, $Y_1 \dots Y_p \xRightarrow{l} w_j$ où l est la somme des longueurs des dérivations des Y_i , donc $X_j \xRightarrow{l} \alpha \Rightarrow w_j$ et $k_j = l + 1$ convient.

□

Théorème 15.9

Le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ est engendré par la grammaire $S \rightarrow aSb \mid \epsilon$. Il est donc non contextuel mais, d'après le chapitre précédent, il n'est pas régulier.

Preuve Notons G cette grammaire.

- $L \subset \text{lang}(G)$: on montre par récurrence sur n que $S \xRightarrow{*} a^n S b^n$. C'est vrai pour $n = 0$ et si $S \xRightarrow{*} a^n S b^n$ alors $S \xRightarrow{*} a^n a S b b^n = a^{n+1} S b^{n+1}$. Ainsi $S \xRightarrow{*} a^n S b^n \Rightarrow a^n b^n$.
- $\text{lang}(G) \subset L$: on montre par récurrence sur k que si $S \xRightarrow{k} u$ alors u est de la forme $a^n S b^n$ ou $a^n b^n$. C'est vrai pour $k = 0$ avec $u = S$. Si $S \xRightarrow{k} u \Rightarrow v$, comme u contient un non terminal, $u = a^n S b^n$ et v vaut $a^{n+1} S b^{n+1}$ ou $a^n b^n$ selon la règle appliquée. Les mots de Σ^* obtenus sont donc de la forme $a^n b^n$.

Le fait que L ne soit pas régulier a été démontré avec le lemme de l'étoile.

□

On a ainsi trouvé un langage qui est non contextuel sans être régulier. Inversement, tout langage régulier est non contextuel : les grammaires non contextuelles sont strictement plus expressives que les automates finis.

Théorème 15.10

Tout langage régulier est non contextuel. L'inclusion est stricte.

Preuve Soit L un langage régulier. D'après le théorème de Kleene, il est reconnu par un automate fini déterministe $A = (Q, \Sigma, \delta, q_i, F)$, où l'on suppose, quitte à renommer les états, que $Q \cap \Sigma = \emptyset$.

On considère la grammaire $G = (\Sigma, Q, P, q_i)$ dont les non terminaux sont les états et dont les règles sont :

- $q \rightarrow a q'$ pour chaque transition $\delta(q, a) = q'$;
- $q \rightarrow \epsilon$ pour chaque état final $q \in F$.

Montrons que pour $q, q' \in Q$ et $w \in \Sigma^*$, on a $q \xRightarrow{*} w q'$ si et seulement si $\delta^*(q, w) = q'$. Une dérivation à partir de q ne contient jamais qu'un seul non terminal, placé à la fin : tant qu'on applique des règles $q \rightarrow a q'$ on obtient des mots de la forme $u p$ avec $u \in \Sigma^*$ et $p \in Q$, et une règle $q \rightarrow \epsilon$ supprime le dernier non terminal, après quoi plus rien n'est possible. Ainsi une dérivation $q \xRightarrow{*} w q'$ avec $w = a_1 \dots a_n$ est exactement de la forme

$$q = p_0 \Rightarrow a_1 p_1 \Rightarrow a_1 a_2 p_2 \Rightarrow \dots \Rightarrow a_1 \dots a_n p_n = w q'$$

avec $\delta(p_{k-1}, a_k) = p_k$ pour tout k , c'est-à-dire un chemin d'étiquette w de q à q' dans l'automate.

Enfin, la dernière étape d'une dérivation $q_i \xRightarrow{*} w$ avec $w \in \Sigma^*$ est nécessairement l'application d'une règle $q' \rightarrow \epsilon$ avec $q' \in F$. Donc $w \in \text{lang}(G)$ si et seulement s'il existe $q' \in F$ tel que $q_i \xRightarrow{*} w q'$, c'est-à-dire si et seulement si $\delta^*(q_i, w) \in F$, soit $w \in \text{lang}(A)$.

L'inclusion est stricte d'après le théorème précédent.

□

Exemple 15.11

L'automate de l'exemple du chapitre précédent

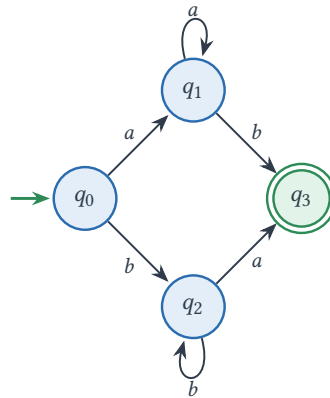


Figure 15.2 — Automate déterministe reconnaissant $aa^*b \mid bb^*a$

donne la grammaire

```

$$
q_0 \rightarrow a q_1 \mid b q_2 \quad \text{qqquad}
q_1 \rightarrow a q_1 \mid b q_3 \quad \text{qqquad}
q_2 \rightarrow a q_3 \mid b q_2 \quad \text{qqquad}
q_3 \rightarrow \epsilon
$$

```

et la dérivation $q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_3$ et $q_0 \xrightarrow{b} q_2 \xrightarrow{b} q_2 \xrightarrow{a} q_3$ correspond au chemin réussi d'étiquette aab et baa .

Remarque 15.12

- Les grammaires obtenues ainsi, où toutes les règles sont de la forme $A \rightarrow aB$ ou $A \rightarrow \epsilon$, sont dites *linéaires à droite*. On peut montrer qu'elles engendrent exactement les langages réguliers.
- On aurait aussi pu partir de la définition inductive de $\text{Reg}(\Sigma)$: les langages non contextuels contiennent $\emptyset$ et les $\{a\}$ et sont stables par union ($S \rightarrow S_1 \mid S_2$), concaténation ($S \rightarrow S_1 S_2$) et étoile ($S \rightarrow S_1 S \mid \epsilon$), en prenant des non terminaux disjoints pour les deux grammaires.
- Contrairement aux langages réguliers, les langages non contextuels ne sont pas stables par intersection ni par complémentaire (admis).

Exemple 15.13

Le langage des mots bien parenthésés sur $\{ (,) \}$ est engendré par la grammaire $S \rightarrow (S)S \mid \epsilon$. On le prouve, comme pour $a^n b^n$, par une double inclusion : par récurrence sur la longueur de la dérivation dans un sens, et en découpant un mot bien parenthésé non vide selon la parenthèse fermante associée à la première parenthèse ouvrante dans l'autre. Ce langage n'est pas régulier : son intersection avec le langage régulier $\{ \{ \}^* \}^*$ est $\{ ({}^n)^n \mid n \in \mathbb{N} \}$.

15.2.3 Dérivations à gauche et à droite



Un même mot peut être obtenu par de nombreuses dérivations qui ne diffèrent que par l'ordre dans lequel on remplace les non terminaux. Avec la grammaire $E \rightarrow E + E \mid \text{id} \mid \text{num}$, on a par exemple :

$$E \Rightarrow E + E \Rightarrow \text{id} + E \Rightarrow \text{id} + \text{num} \quad E \Rightarrow E + E \Rightarrow E + \text{num} \Rightarrow \text{id} + \text{num}$$

Pour éviter ces répétitions inutiles, on fixe un ordre.

Définition 15.14

Une dérivation $u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_k$ est une **dérivation à gauche** (resp. **à droite**) lorsqu'à chaque étape, c'est le non terminal le plus à gauche (resp. le plus à droite) de u_i qui est remplacé pour obtenir u_{i+1} .

Exemple 15.15

Dans l'exemple précédent, la première dérivation est à gauche, la seconde à droite. La dérivation de l'introduction était une dérivation à gauche.

Remarque 15.16 Pour la grammaire de la preuve précédente issue d'un automate, il n'y a jamais qu'un seul non terminal : toute dérivation est à la fois à gauche et à droite.

On verra à la section suivante que se restreindre aux dérivations à gauche ne change pas le langage engendré : tout mot de $\text{lang}(G)$ admet au moins une dérivation à gauche. C'est ce qui rend ces dérivations naturelles pour un analyseur qui lit le mot de gauche à droite.

15.2.4 Arbre d'analyse

Les dérivations portent encore trop d'information : l'ordre de remplacement des non terminaux n'a pas d'importance. Ce qui compte, c'est la règle choisie pour chaque non terminal, ce que l'on représente par un arbre, comme l'arbre de dérivation de l'introduction.

Définition 15.17

Soit $G = (\Sigma, V, P, S)$ une grammaire et $A \in V$. Un **arbre d'analyse** de racine A est un arbre tel que :

- la racine est étiquetée par A ;
- les nœuds internes sont étiquetés par des non terminaux ;
- les feuilles sont étiquetées par des terminaux ou par ϵ ;
- si un nœud interne étiqueté par B a pour fils, de gauche à droite, des nœuds étiquetés par $X_1, \dots, X_n$, alors $B \rightarrow X_1 \dots X_n \in P$. Une feuille ϵ est l'unique fils d'un nœud B tel que $B \rightarrow \epsilon \in P$.

La **frontière** de l'arbre est le mot de Σ^* obtenu en concaténant les étiquettes des feuilles de gauche à droite.

Exemple 15.18

Pour la grammaire $S \rightarrow aSb \mid \epsilon$, voici l'arbre d'analyse de racine S et de frontière $aabb$:

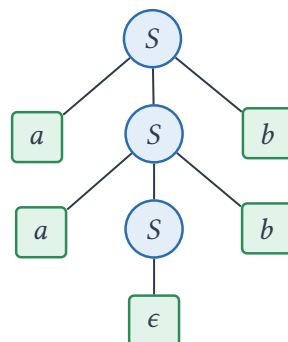


Figure 15.3 — Arbre d'analyse de $aabb$: non-terminaux en bleu, terminaux en vert

Théorème 15.19

Soit $A \in V$ et $w \in \Sigma^*$. On a $A \xRightarrow{*} w$ si et seulement s'il existe un arbre d'analyse de racine A et de frontière w .

En particulier, $\text{lang}(G)$ est l'ensemble des frontières des arbres d'analyse de racine S .

Preuve

- Soit un arbre d'analyse de racine A et de frontière w . On montre $A \Rightarrow^* w$ par récurrence sur la hauteur de l'arbre. Si la racine a pour unique fils une feuille ϵ , on a $A \rightarrow \epsilon \in P$ et $w = \epsilon$. Sinon, la racine a pour fils $X_1, \dots, X_n$ avec $A \rightarrow X_1 \dots X_n \in P$. Pour chaque i , si X_i est un terminal, c'est une feuille et on pose $w_i = X_i$; sinon le sous-arbre de racine X_i est un arbre d'analyse de hauteur strictement plus petite, de frontière w_i , et par hypothèse de récurrence $X_i \Rightarrow^* w_i$. Comme $w = w_1 \dots w_n$, le lemme de décomposition donne $A \Rightarrow^* X_1 \dots X_n \Rightarrow^* w$.
- Réciproquement, on montre par récurrence forte sur $k \geq 1$ que si $A \Rightarrow^k w$, alors il existe un arbre d'analyse de racine A et de frontière w ($k = 0$ est impossible car $A \notin \Sigma^*$). La première étape utilise une règle $A \rightarrow X_1 \dots X_n$ et $X_1 \dots X_n \Rightarrow^{k-1} w$. Si $n = 0$, $w = \epsilon$ et l'arbre A avec une feuille ϵ convient. Sinon, d'après le lemme de décomposition, $w = w_1 \dots w_n$ avec $X_i \Rightarrow^{k_i} w_i$ et $k_i \leq k - 1$. Pour un terminal X_i , on prend la feuille X_i ; pour un non terminal, on obtient par hypothèse de récurrence un arbre de racine X_i et de frontière w_i . On accroche ces arbres sous une racine étiquetée par A .

□

Un arbre d'analyse oublie l'ordre des remplacements. Pour retrouver une dérivation à partir d'un arbre, il suffit de choisir un ordre : si on remplace toujours le non terminal le plus à gauche, on parcourt les nœuds internes dans l'ordre d'un parcours en profondeur préfixe et on obtient une dérivation à gauche.

Théorème 15.20

(admis) Pour tout $w \in \Sigma^*$, l'application qui associe à une dérivation à gauche $S \Rightarrow^* w$ l'arbre d'analyse qu'elle construit est une bijection entre les dérivations à gauche de w et les arbres d'analyse de racine S et de frontière w . Il en va de même pour les dérivations à droite.

Remarque 15.21 En particulier, tout mot de lang (G) admet une dérivation à gauche. L'idée de la bijection est la suivante : dans une dérivation à gauche, le non terminal remplacé à chaque étape est imposé, seule la règle est à choisir, et c'est précisément cette suite de règles, lue dans l'ordre préfixe, que l'arbre mémorise.

Remarque 15.22 L'arbre d'analyse contient tous les détails de la grammaire : non terminaux intermédiaires, parenthèses, etc. Un analyseur syntaxique construit en général un arbre plus compact, l'**arbre de syntaxe abstraite**, comme les valeurs du type expression du TP, où une expression $(1 + 2) * x$ est représentée par Mult (Add (Constant 1, Constant 2), Variable "x") sans trace des parenthèses.

15.2.5 Ambiguïté**Définition 15.23**

Une grammaire G est **ambiguë** lorsqu'il existe un mot $w \in \text{lang}(G)$ admettant au moins deux arbres d'analyse distincts de racine S , ou de manière équivalente, au moins deux dérivations à gauche distinctes.

L'exemple typique est celui des expressions arithmétiques du TP, avec les lexèmes Id, Num, Plus, Minus, Times, Div, LPar, RPar notés respectivement id, num, +, -, *, /, (,) :

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \mid \text{num}$$

Exemple 15.24

Le mot $\text{id} + \text{id} * \text{id}$ admet deux dérivations à gauche :

$$E \Rightarrow E + E \Rightarrow \text{id} + E \Rightarrow \text{id} + E * E \Rightarrow \text{id} + \text{id} * E \Rightarrow \text{id} + \text{id} * \text{id}$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow \text{id} + E * E \Rightarrow \text{id} + \text{id} * E \Rightarrow \text{id} + \text{id} * \text{id}$$

qui correspondent aux deux arbres d'analyse :

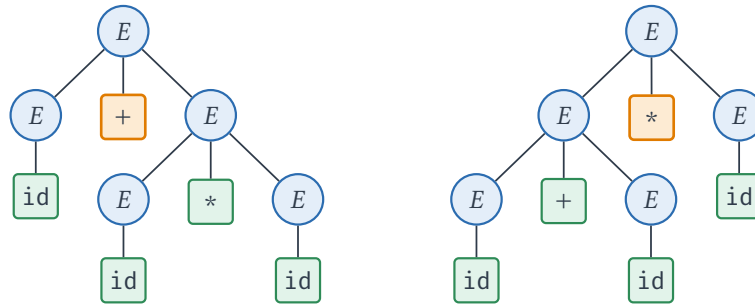


Figure 15.4 — Deux arbres d'analyse de $\text{id} + \text{id} * \text{id}$; en orange, l'opérateur appliqué en dernier

Le premier correspond à $\text{id} + (\text{id} * \text{id})$, le second à $(\text{id} + \text{id}) * \text{id}$. Si on évalue l'expression à partir de l'arbre, avec $1 + 2 * 3$, on obtient 7 dans un cas et 9 dans l'autre : l'ambiguïté n'est pas qu'un problème théorique.

De même, $\text{id} - \text{id} - \text{id}$ a deux arbres d'analyse, correspondant à $(8 - 2) - 1 = 5$ et $8 - (2 - 1) = 7$.

Remarque 15.25

- L'ambiguïté est une propriété de la grammaire et non du langage : on va voir une grammaire non ambiguë engendrant le même langage. Il existe cependant des langages non contextuels dont toutes les grammaires sont ambiguës, comme $\{a^i b^j c^k \mid i = j \text{ ou } j = k\}$ (admis).
- On ne peut pas décider algorithmiquement si une grammaire est ambiguë (admis, voir le chapitre de calculabilité).
- Pour la grammaire $S \rightarrow aSb \mid \epsilon$, il n'y a pas d'ambiguïté : à chaque étape d'une dérivation, il n'y a qu'un seul non terminal, et la règle est imposée par le nombre de a restant à produire.

Pour lever l'ambiguïté, on encode dans la grammaire les conventions usuelles :

- **priorité** : $*$ et $/$ sont prioritaires sur $+$ et $-$;
- **associativité** : les quatre opérateurs sont associatifs à gauche, $a - b - c$ se lit $(a - b) - c$.

Pour cela, on introduit un non terminal par niveau de priorité : E pour une expression (une somme de termes), T pour un terme (un produit de facteurs) et F pour un facteur.

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \text{id} \mid \text{num} \end{aligned}$$

Remarque 15.26

- **Priorité** : un opérande de $*$ est un F , il ne peut donc contenir un $+$ qu'entre parenthèses. Dans $\text{id} + \text{id} * \text{id}$, le $+$ est nécessairement à la racine.
- **Associativité** : dans $E \rightarrow E - T$, l'opérande droit est un T qui ne peut pas contenir de $-$ hors parenthèses. Dans $\text{id} - \text{id} - \text{id}$, c'est donc le dernier $-$ qui est à la racine, ce qui donne bien $(\text{id} - \text{id}) - \text{id}$. La récursivité **à gauche** de la règle donne l'associativité **à gauche**.
- On admet que cette grammaire n'est pas ambiguë et qu'elle est faiblement équivalente à la précédente.
- La grammaire complète du TP, $\text{expr} \rightarrow \text{term} + \text{expr} \mid \text{term} - \text{expr} \mid \text{term}$, est récursive à droite : elle est elle aussi non ambiguë, mais rend les opérateurs associatifs à droite, ce qui est faux pour $-$ et $/$. On va voir pourquoi elle a été choisie ainsi et comment corriger ce défaut.

15.3 Analyse syntaxique par descente récursive

- Étant donné une grammaire et une suite de lexèmes, l'**analyse syntaxique** consiste à déterminer si la suite est dans le langage engendré, et le cas échéant à construire un arbre (d'analyse ou de syntaxe abstraite). La

méthode la plus simple à écrire à la main est la **descente récursive** : on construit l'arbre depuis la racine, en lisant les lexèmes de gauche à droite, ce qui revient à produire une dérivation à gauche.

On reprend les types du TP :

```
type token =
  | Id of string
  | Num of int
  | Plus | Times | Minus | Div
  | LPar | RPar

type expression =
  | Variable of string
  | Constant of int
  | Add of expression * expression
  | Subtract of expression * expression
  | Mult of expression * expression
  | Divide of expression * expression

exception ParsingError
```

15.3.1 Premier exemple

On considère la grammaire des mots bien parenthésés $S \rightarrow (S)S \mid \epsilon$ sur les lexèmes LPar et RPar. On écrit une fonction `parse_s` qui lit un préfixe de la liste de lexèmes dérivant de S et renvoie le suffixe non lu.

Pour choisir la règle, on regarde le premier lexème : un mot dérivé par $S \rightarrow (S)S$ commence par LPar, et sinon on applique $S \rightarrow \epsilon$, qui ne consomme rien. Lire le membre droit $(S)S$, c'est lire LPar, puis un S par un appel récursif, puis vérifier la présence de RPar, puis lire un dernier S .

```
(* S -> ( S ) S | epsilon *)
let rec parse_s (tl : token list) : token list =
  match tl with
  | LPar :: q ->
    begin match parse_s q with
    | RPar :: q' -> parse_s q'
    | _ -> raise ParsingError
    end
  | _ -> tl

let bien_parenthesee (tl : token list) : bool =
  try parse_s tl = [] with ParsingError -> false
```

Exemple 15.27

- `bien_parenthesee [LPar; LPar; RPar; RPar; LPar; RPar]` renvoie `true`.
- `bien_parenthesee [LPar]` renvoie `false` : l'appel interne à `parse_s` renvoie `[]` et on ne trouve pas RPar.
- `bien_parenthesee [LPar; RPar; RPar]` renvoie `false` : `parse_s` renvoie `[RPar]`, qui n'est pas vide.

Remarque 15.28 Il ne faut pas oublier de vérifier, à la fin, que tous les lexèmes ont été consommés : `parse_s` s'arrête dès qu'elle ne peut plus appliquer la règle $S \rightarrow (S)S$, et `[LPar; RPar; RPar]` a pour préfixe le mot bien parenthésé `[LPar; RPar]`.

15.3.2 Principe général

Pour une grammaire G , on écrit :

- une fonction `parse_A` par non terminal A , ces fonctions étant mutuellement récursives. Elle prend une liste de lexèmes, lit un préfixe qui dérive de A et renvoie un couple formé de l'arbre construit et du suffixe restant ;
- pour lire un membre droit $X_1 \dots X_n$, on lit successivement chaque symbole : pour un terminal, on vérifie que c'est le premier lexème de la liste et on le consomme, sinon on lève `ParsingError` ; pour un non terminal B , on appelle `parse_B` sur la liste restante ;

- pour choisir entre les règles $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$, on regarde **uniquement le premier lexème** de la liste ;
- enfin, l'analyse complète appelle `parse_s` et vérifie que la liste restante est vide.

C'est exactement ce que fait le TP avec `parse_expression`, `parse_term` et `parse_factor`. Une telle analyse ne revient jamais en arrière : chaque lexème est examiné un nombre borné de fois et l'analyse est en temps linéaire en le nombre de lexèmes.

Cette méthode ne s'applique pas à toutes les grammaires. Deux obstacles apparaissent dès la grammaire non ambiguë des expressions :

- $E \rightarrow E + T$: pour lire un E , `parse_expression` commencerait par s'appeler elle-même sur la même liste, ce qui boucle indéfiniment ;
- $E \rightarrow T + E \mid T$ (version du TP) : le premier lexème ne permet pas de savoir quelle règle appliquer, les deux membres droits commençant par un T .

On va voir deux transformations de grammaires, qui préservent le langage engendré, permettant de lever ces obstacles.

15.3.3 Élimination de la récursivité à gauche

Une règle $A \rightarrow A\alpha$ est dite **récursive à gauche**. Avec une telle règle, la fonction `parse_A` ressemblerait à

```
let rec parse_expression tl =
  let e, q = parse_expression tl in (* même liste : boucle infinie *)
  ...
```

et l'exécution se terminerait par un dépassement de pile (`Stack_overflow`).

Propriété 15.29

Soient $\alpha, \beta \in (\Sigma \cup V)^*$ ne contenant pas A . Remplacer les règles $A \rightarrow A\alpha \mid \beta$ par les règles

$$A \rightarrow \beta A' \quad A' \rightarrow \alpha A' \mid \epsilon$$

où A' est un nouveau non terminal, ne change pas le langage engendré.

Preuve Pour $\gamma \in (\Sigma \cup V)^*$, notons $L(\gamma) = \{w \in \Sigma^* \mid \gamma \xRightarrow{*} w\}$, qui ne dépend que des règles des non terminaux autres que A et A' si γ ne contient ni A ni A' . Il suffit de montrer que, dans les deux grammaires, les frontières des arbres d'analyse de racine A sont les mots de $L(\beta)L(\alpha)^*$; le lemme de décomposition permet ensuite de conclure pour toute dérivation à partir de S .

- Dans la première grammaire, considérons un arbre d'analyse de racine A et suivons depuis la racine la branche des premiers fils étiquetés A : on obtient $n \geq 0$ nœuds utilisant la règle $A \rightarrow A\alpha$ puis un nœud utilisant $A \rightarrow \beta$. La frontière est donc la concaténation d'une frontière d'arbres pour β puis de n frontières d'arbres pour α : c'est un mot de $L(\beta)L(\alpha)^n$. Réciproquement, tout mot de $L(\beta)L(\alpha)^n$ s'obtient ainsi.
- Dans la seconde grammaire, la racine utilise $A \rightarrow \beta A'$ et on suit la branche des derniers fils étiquetés A' : n nœuds utilisent $A' \rightarrow \alpha A'$ puis un nœud $A' \rightarrow \epsilon$. La frontière est encore un mot de $L(\beta)L(\alpha)^n$ et réciproquement.

□

Remarque 15.30 Plus généralement, $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_p$ se transforme en $A \rightarrow \beta_1 A' \mid \dots \mid \beta_p A'$ et $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \epsilon$.

Appliquée à la grammaire non ambiguë des expressions, cette transformation donne :

$$\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid -TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid /FT' \mid \epsilon \\
F &\rightarrow (E) \mid \text{id} \mid \text{num}
\end{aligned}$$

Remarque 15.31 Les arbres d'analyse de cette nouvelle grammaire penchent à droite : on a perdu la forme qui donnait l'associativité à gauche. Mais l'analyseur n'est pas obligé de renvoyer l'arbre d'analyse : la fonction associée à E' peut recevoir l'expression déjà lue dans un accumulateur et construire l'arbre de syntaxe abstraite de gauche à droite. C'est ce qui est fait dans l'exemple complet ci-dessous et qui répond à la remarque finale du TP.

15.3.4 Factorisation à gauche

Quand deux règles d'un même non terminal commencent par le même préfixe, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ avec $\alpha \neq \epsilon$, le premier lexème ne permet pas de choisir. On met alors ce préfixe en facteur avec un nouveau non terminal A' :

$$A \rightarrow \alpha A' \quad A' \rightarrow \beta_1 \mid \beta_2$$

Le langage engendré est clairement inchangé : une étape $A \Rightarrow \alpha\beta_i$ correspond aux deux étapes $A \Rightarrow \alpha A' \Rightarrow \alpha\beta_i$.

Exemple 15.32

La grammaire du TP $E \rightarrow T + E \mid T - E \mid T$ devient

$$E \rightarrow TE' \quad E' \rightarrow +E \mid -E \mid \epsilon$$

C'est exactement ce que fait la fonction `parse_expression` du TP : elle appelle d'abord `parse_term`, puis regarde si le lexème suivant est `Plus`, `Minus` ou autre chose.

15.3.5 Exemple plus complet

On écrit maintenant l'analyseur de la grammaire obtenue après élimination de la récursivité à gauche. Les fonctions `parse_expression_suite` et `parse_term_suite` correspondent à E' et T' ; elles reçoivent dans `acc` l'arbre de ce qui a déjà été lu à gauche, ce qui rend les opérateurs associatifs à gauche.

```

let parser (tl : token list) : expression =
  (* E -> T E' *)
  let rec parse_expression tl =
    let t, q = parse_term tl in
    parse_expression_suite t q
  in
  (* E' -> + T E' | - T E' | epsilon *)
  and parse_expression_suite acc tl =
    match tl with
    | Plus :: q ->
      let t, q = parse_term q in
      parse_expression_suite (Add (acc, t)) q
    | Minus :: q ->
      let t, q = parse_term q in
      parse_expression_suite (Subtract (acc, t)) q
    | _ -> (acc, tl)
  in
  (* T -> F T' *)
  and parse_term tl =
    let f, q = parse_factor tl in
    parse_term_suite f q
  in
  (* T' -> * F T' | / F T' | epsilon *)
  and parse_term_suite acc tl =
    match tl with
    | Times :: q ->
      let f, q = parse_factor q in

```

```

    parse_term_suite (Mult (acc, f)) q
  | Div :: q ->
    let f, q = parse_factor q in
    parse_term_suite (Divide (acc, f)) q
  | _ -> (acc, tl)
(* F -> ( E ) | id | num *)
and parse_factor tl =
  match tl with
  | Id x :: q -> (Variable x, q)
  | Num n :: q -> (Constant n, q)
  | LPar :: q ->
    begin match parse_expression q with
    | e, RPar :: q -> (e, q)
    | _ -> raise ParsingError
    end
  | _ -> raise ParsingError
in
let e, q = parse_expression tl in
if q <> [] then raise ParsingError;
e

```

Exemple 15.33

- `parser [Num 8; Minus; Num 2; Minus; Num 1]` renvoie `Subtract (Subtract (Constant 8, Constant 2), Constant 1)`, c'est-à-dire $(8 - 2) - 1$.
- `parser [Num 1; Plus; Num 2; Times; Num 3]` renvoie `Add (Constant 1, Mult (Constant 2, Constant 3))`.
- `parser [LPar; Num 1; Plus; Num 2; RPar; Times; Id "x"]` renvoie `Mult (Add (Constant 1, Constant 2), Variable "x")`.
- `parser [], parser [Num 1; Plus], parser [LPar; Num 1]` et `parser [Num 1; Num 2]` lèvent `ParsingError`.

Remarque 15.34 Dans les cas ϵ , on renvoie la liste sans rien vérifier (`| _ -> (acc, tl)`). Une erreur comme dans `[Num 1; Num 2]` n'est alors détectée que plus haut, par le test final `q <> []`. On pourrait la détecter plus tôt en vérifiant que le lexème suivant peut effectivement suivre un E' : c'est ce que formalise la section suivante.

15.3.6 Aller plus loin

Cette section n'est plus au programme de MPI : elle formalise la condition « on peut choisir la règle en regardant le premier lexème » utilisée ci-dessus. On ajoute à la fin de l'entrée un terminal spécial `#` qui marque la fin de la liste.

FIRST et FOLLOW

Définition 15.35

Soit $G = (\Sigma, V, P, S)$ une grammaire.

- Pour $\alpha \in (\Sigma \cup V)^*$, $\text{FIRST}(\alpha)$ est l'ensemble des terminaux a tels que $\alpha \xRightarrow{*} au$ pour un certain u , auquel on ajoute ϵ si $\alpha \Rightarrow \epsilon$.
- Pour $A \in V$, $\text{FOLLOW}(A)$ est l'ensemble des terminaux de $\Sigma \cup \{\#\}$ qui peuvent suivre A dans une dérivation, c'est-à-dire les a tels que $S\# \xRightarrow{*} uAav$ pour certains u, v .

On calcule ces ensembles en partant d'ensembles vides et en appliquant les règles suivantes jusqu'à ce que plus rien ne change :

- $\text{FIRST}(\epsilon) = \{\epsilon\}$ et $\text{FIRST}(a\beta) = \{a\}$ pour $a \in \Sigma$;
- $\text{FIRST}(A\beta) = \text{FIRST}(A)$ si $\epsilon \notin \text{FIRST}(A)$, et $(\text{FIRST}(A) \setminus \{\epsilon\}) \cup \text{FIRST}(\beta)$ sinon ;
- $\text{FIRST}(A)$ est la réunion des $\text{FIRST}(\alpha)$ pour les règles $A \rightarrow \alpha$;
- $\# \in \text{FOLLOW}(S)$ et pour chaque règle $B \rightarrow \alpha A \beta$: $\text{FIRST}(\beta) \setminus \{\epsilon\} \subset \text{FOLLOW}(A)$ et, si $\epsilon \in \text{FIRST}(\beta)$, $\text{FOLLOW}(B) \subset \text{FOLLOW}(A)$.

Exemple 15.36

Pour la grammaire $E \rightarrow TE'$, $E' \rightarrow +TE' \mid -TE' \mid \epsilon$, $T \rightarrow FT'$, $T' \rightarrow *FT' \mid /FT' \mid \epsilon$, $F \rightarrow (E) \mid \text{id} \mid \text{num}$:

Non terminal	FIRST	FOLLOW
E	(, id, num	), #
E'	+, -, ϵ	), #
T	(, id, num	+, -,), #
T'	*, / , ϵ	+, -,), #
F	(, id, num	*, / , +, -,), #

Par exemple, $\text{FOLLOW}(T)$ contient $\text{FIRST}(E') \setminus \{\epsilon\} = \{+, -\}$ à cause de la règle $E \rightarrow TE'$ et, comme $E' \Rightarrow \epsilon$, il contient aussi $\text{FOLLOW}(E) = \{), \#\}$.

Table de choix de production

On range dans une table M , indexée par un non terminal A et un terminal $a \in \Sigma \cup \{\#\}$, la règle à appliquer pour lire A quand le prochain lexème est a : la règle $A \rightarrow \alpha$ est placée dans la case $M[A, a]$ lorsque $a \in \text{FIRST}(\alpha)$, ou lorsque $\epsilon \in \text{FIRST}(\alpha)$ et $a \in \text{FOLLOW}(A)$. Une case vide signifie une erreur de syntaxe.

Exemple 15.37

Sur l'exemple précédent, en n'indiquant que le membre droit de la règle :

	id	num	(	)	+	-	*	/	#
E	TE'	TE'	TE'						
E'				ϵ	$+TE'$	$-TE'$			ϵ
T	FT'	FT'	FT'						
T'				ϵ	ϵ	ϵ	$*FT'$	$/FT'$	ϵ
F	id	num	(E)						

Chaque fonction de l'analyseur précédent correspond à une ligne de cette table : les cases ϵ sont le cas $| _ \rightarrow (\text{acc}, \text{tl})$ et les cases vides de la ligne F le cas $| _ \rightarrow \text{raise ParsingError}$.

Grammaires LL(1)**Définition 15.38**

Une grammaire est **LL(1)** lorsque chaque case de sa table de choix contient au plus une règle. De manière équivalente, pour toutes règles distinctes $A \rightarrow \alpha$ et $A \rightarrow \beta$:

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$;
- si $\epsilon \in \text{FIRST}(\alpha)$, alors $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$.

Le nom signifie : lecture de gauche à droite (*Left to right*), construction d'une dérivation à gauche (*Leftmost derivation*), avec 1 lexème d'avance. Ce sont exactement les grammaires pour lesquelles la descente récursive présentée ici fonctionne sans retour en arrière.

Remarque 15.39

- La grammaire précédente est LL(1), comme on le lit sur la table.
- Une grammaire récursive à gauche n'est jamais LL(1) (si $A \rightarrow A\alpha \mid \beta$ avec $\text{FIRST}(\beta)$ contenant un terminal, il est aussi dans $\text{FIRST}(A\alpha)$), ni une grammaire non factorisée comme $E \rightarrow T + E \mid T$. Une grammaire ambiguë n'est jamais LL(1) (admis).
- Les outils comme *ocamllyacc* ou *menhir* utilisent une autre famille de méthodes, l'analyse ascendante LR, qui accepte notamment les grammaires récursives à gauche.

Parser des expressions

DANS CE CHAPITRE

- 16.1** Analyse lexicale
- 16.2** Premier analyseur syntaxique
- 16.3** Deuxième analyseur syntaxique
- 16.4** La grammaire complète
- 16.5** Et ensuite ?

Dans ce TP, on va analyser syntaxiquement des expressions arithmétiques composées de variables, d'entiers, de parenthèses et d'un des opérateurs `+` `-` `*` `/`.

16.1 Analyse lexicale

On va commencer par écrire une fonction qui convertit une chaîne de caractères en une liste de caractères pour pouvoir faire du filtrage dessus.

```
let string_to_charlist (s:string) : char list =
  (* renvoie la liste des caractères d'une chaîne *)
  List.of_seq (String.to_seq s)
```

On définit maintenant un type pour les jetons :

```
type token =
  | Id of string
  | Num of int
  | Plus | Times | Minus | Div
  | LPar | RPar
```

La fonction suivante décide si un caractère est une lettre minuscule ou majuscule dans `abc...z` ou `ABC...Z`.

```
let is_alpha c =
  match c with
  | 'a'..'z' | 'A'..'Z' -> true
  | _ -> false
```

La fonction suivante décide si un caractère est un chiffre de 0 à 9.

```
let is_digit c =
  match c with
  | '0'..'9' -> true
  | _ -> false
```

On considère maintenant une liste de caractères `l` et la fonction suivante renvoie un couple `(l1,l2)` tel que :

- $l1$ est le plus grand préfixe de l qui satisfait l'expression régulière $0|[1-9][0-9]^*$, ce qui interdit les zéros non significatifs tout en acceptant la constante 0.
- $l1 @ l2 = l$

S'il n'y a pas de préfixe numérique valide, on va donc renvoyer $([], l)$.

```
let extract_num s =
  let rec aux s =
    match s with
    | ('0'..'9' as c)::q ->
        let i, s' = aux q in
        (c::i, s')
    | _ -> [], s
  in match s with
  | '0'::q -> ([ '0' ], q)
  | ('1'..'9' as c)::q ->
      let i, s' = aux q in
      (c::i, s')
  | _ -> [], s
```

Remarque 16.1 Sur 012, le plus grand préfixe valide est 0 : le lexer produit alors les jetons Num 0 puis Num 12, et c'est l'analyse syntaxique qui rejettera l'expression. Comme tout chiffre commence un préfixe valide, extract_num ne renvoie jamais [] dans lexer ; le test $x = []$ y est conservé par prudence.

La fonction suivante procède comme la précédente mais avec un préfixe satisfaisant l'expression régulière $[a-zA-Z][a-zA-Z0-9]^*$.

```
let extract_ident s =
  let rec extract_aux s =
    match s with
    | c::q when is_digit c || is_alpha c || c = '_' ->
        let i, s' = extract_aux q in
        (c::i, s')
    | _ -> [], s
  in
  match s with
  | c::_ when is_alpha c ->
      extract_aux s
  | _ -> [], s
```

La fonction lexer convertit une liste de caractères en une liste de tokens. Si jamais on ne peut pas produire de tokens, on produit l'exception LexingError.

Cette fonction est naturellement récursive, on va ignorer tous les caractères ' '. Ici, il s'agit simplement d'avoir une règle

```
| ' '::q -> lexer q
```

```
exception LexingError

let rec lexer s =
  match s with
  | [] -> []
  | ' '::q -> lexer q
  | '('::q -> LPar :: lexer q
  | ')'::q -> RPar :: lexer q
  | '+'::q -> Plus :: lexer q
  | '-'::q -> Minus :: lexer q
  | '*'::q -> Times :: lexer q
  | '/'::q -> Div :: lexer q
  | x :: _ when is_digit x ->
      let x, q = extract_num s in
      if x = []
      then raise LexingError
      else Num (int_of_string
```

```

        (String.concat "" (List.map (String.make 1) x)))
    :: lexer q
| _ ->
    let x, q = extract_ident s in
    if x = []
    then raise LexingError
    else let x = String.concat "" (List.map (String.make 1) x) in
        Id x :: lexer q

```

16.2 Premier analyseur syntaxique

On considère maintenant que l'entrée est une liste de tokens obtenue par un appel valide à `lexer`.

On va considérer un premier langage d'expression qui correspond à la grammaire :

```

T = + num
NT = expr
s = expr

expr -> num | num + expr

```

Le langage contient donc des expressions comme `1 + 33 + 23` ou `12`.

Une telle expression se représente naturellement par la liste des entiers. Donc pour l'exemple `[1;33;23]` ou `[12]`.

On va écrire une fonction qui produit cette liste :

```

exception ParsingError

let rec parser1 (tl:token list) : int list =
  match tl with
  | [Num n] -> [n]
  | Num n :: Plus :: q -> n :: parser1 q
  | _ -> raise ParsingError

```

16.3 Deuxième analyseur syntaxique

On considère maintenant un type arborescent expression complet :

```

type expression =
  Variable of string
| Constant of int
| Add of expression * expression
| Subtract of expression * expression
| Mult of expression * expression
| Divide of expression * expression

```

Les analyseurs syntaxiques vont ainsi essayer de reconstruire un arbre.

On commence avec une grammaire assez simple :

```

T = ( ) + / - * id num
NT = expr operand
s = expr

expr -> operand
    | operand + operand
    | operand - operand
    | operand * operand
    | operand / operand

operand -> id | num | ( expr )

```

Pour réaliser cet analyseur, on vous conseille d'avoir deux fonctions auxiliaires récursives `parse_operand` et `parse_expression` qui vont prendre une liste de token et renvoyer un couple (e, q) où e est l'expression lue et q les tokens non utilisés.

Vu la dernière production, ces fonctions sont nécessairement mutuellement récursives.

À la fin, il faudra s'assurer qu'on a utilisé tous les tokens.

```
let parser2 (tl:token list) : expression =
  let rec parse_operand tl =
    match tl with
    | Id x :: q -> (Variable x, q)
    | Num n :: q -> (Constant n, q)
    | LPar :: q ->
      begin
        match parse_expression q with
        | e, RPar::q -> (e,q)
        | _ -> raise ParsingError
      end
    | _ -> raise ParsingError
  and parse_expression tl =
    let op1, q = parse_operand tl in
    match q with
    | Plus::q -> let op2, q = parse_operand q in (Add(op1, op2), q)
    | Times::q -> let op2, q = parse_operand q in (Mult(op1, op2), q)
    | Minus::q -> let op2, q = parse_operand q in (Subtract(op1, op2), q)
    | Div::q -> let op2, q = parse_operand q in (Divide(op1, op2), q)
    | _ -> (op1, q)
  in

  let e, q = parse_expression tl in

  if q <> []
  then raise ParsingError;
  e
```

16.4 La grammaire complète

On rappelle la grammaire complète vue en cours qui gère la précédence entre opérateurs sans parenthèses :

```
T = ( ) + / - * id num
NT = expr term factor
s = expr

expr -> term + expr | term - expr | term
term -> factor * term | factor / term | factor
factor -> id | num | ( expr )
```

```
let parser tl =
  let rec parse_factor tl =
    match tl with
    | Id x :: q -> (Variable x, q)
    | Num n :: q -> (Constant n, q)
    | LPar :: q ->
      begin
        match parse_expression q with
        | e, RPar::q -> (e,q)
        | _ -> raise ParsingError
      end
    | _ -> raise ParsingError
  and parse_term tl =
    match parse_factor tl with
    | (f, Times :: q) ->
      let t, q = parse_term q in
      Mult(f, t), q
    | (f, Div :: q) ->
```



```

        let t, q = parse_term q in
        Divide(f, t), q
    | (f, q) -> (f, q)
and parse_expression tl =
  match parse_term tl with
  | (t, Plus :: q) ->
      let e, q = parse_expression q in
      Add(t, e), q
  | (t, Minus :: q) ->
      let e, q = parse_expression q in
      Subtract(t, e), q
  | (t, q) -> (t, q)
in

let e, q = parse_expression tl in

if q <> []
then raise ParsingError;
e

```



Remarque 16.2 Avec cette grammaire, les opérateurs sont associatifs **à droite** : $8 - 2 - 1$ est analysé comme $8 - (2 - 1)$, ce qui ne correspond pas à la convention usuelle pour $-$ et $/$. Pour obtenir l’associativité à gauche sans grammaire récursive à gauche (qu’une descente récursive ne sait pas traiter), on peut lire une suite `term op term op ... term` avec une boucle (ou une fonction auxiliaire à accumulateur) qui construit l’arbre de gauche à droite.

16.5 Et ensuite ?

Si vous êtes ici, vous n’avez pas vraiment besoin d’être guidé. Voici ce qu’on peut faire ensuite :

- ajouter des productions et modifier l’analyseur pour gérer le $-$ unaire comme dans $-(1+3)$
- écrire une fonction d’évaluation d’expression prenant en entrée une `Hashtbl` donnant la valeur numérique des variables (on parle d’environnement)
- ajouter la définition de variables locales avec `let ... in` dans la syntaxe et dans l’évaluation d’expressions.
- implémenter le successeur de `OCaml` et écrire un compilateur vers l’assembleur `x86_64`.

PARTIE III

Logique

17	Déduction naturelle	181
18	Introduction au lambda-calcul	197

Déduction naturelle

DANS CE CHAPITRE

- 17.1 Règles d'inférence, dérivations
- 17.2 Déduction naturelle
- 17.3 Utilisation d'un assistant de preuve (Bonus)
- 17.4 Sémantique de Gödel (Bonus)
- 17.5 Lambda-calcul simplement typé (Bonus)
- 17.6 Exercices

17.1 Règles d'inférence, dérivations

On présente ici la notion formelle de règle d'inférence et de dérivations. L'optique de ce chapitre étant de se placer dans un système déductif précis et contraint : la *déduction naturelle*, on pourra sauter ce paragraphe en première lecture.

On considère ici des données appelées **jugements**, il s'agit d'affirmations, on va voir dans la partie suivante le principal type de jugements qui vont nous intéresser : les *séquents* qui affirment qu'une formule est vraie sous certaines hypothèses.

Mais il ne s'agit que d'une facette des jugements que l'on peut rencontrer en informatique. On peut ainsi citer les jugements de types, qui affirment qu'un programme est d'un certain type.

Définition 17.1

On appelle **règle d'inférence** une règle qui relie des jugements, appelés *prémisses*, à un jugement appelé *conclusion*.

Il s'agit d'une formalisation abstraite d'une règle de déduction.

On la note à la manière d'un nœud dans un arbre en utilisant des formes génériques de jugements. Ainsi une règle d'inférence a la forme

$$\frac{\text{Prémisse 1} \quad \dots \quad \text{Prémisse } n}{\text{Conclusion}}$$

Figure 17.1 — Forme générale d'une règle d'inférence : prémisses au-dessus, conclusion au-dessous.

Une règle sans prémisse est appelée un **axiome**.

On donne parfois un nom symbolique à une règle pour pouvoir les distinguer plus facilement, il est alors indiqué à droite du trait.

Remarque 17.2 Ce qu'on signifie ici par le terme **forme générique**, c'est en fait, une forme de filtrage de motifs comme on le connaît en OCaml : si les prémisses ont cette forme, voilà comment on construit la conclusion.

Exemple 17.3

On considère ici que les jugements sont des ensembles et on pose les règles suivantes :

$$\frac{}{\emptyset} \quad \frac{}{\{x\}} \text{ singl} \quad \frac{A \quad B}{A \cup B} \cup$$

Figure 17.2 — Règles de construction d'ensembles : vide, singleton, union.

Les deux dernières règles sont génériques :

- * la règle *singl* fait référence à un élément x quelconque, cela signifie qu'elle s'applique dès que le jugement contient un unique élément.
- * la seconde règle s'applique pour deux ensembles quelconques.

Définition 17.4

On dit qu'on peut **dériver** un jugement depuis un ensemble fixé de règles d'inférences, quand il est possible de réaliser un arbre **fini** dont les nœuds sont des applications des règles.

Un tel arbre est appelé une **dérivation**.

Remarque 17.5

1. Ici, il s'agit d'applications concrètes des règles et non plus de formes génériques.
2. Comme l'arbre est fini, ses feuilles sont nécessairement des règles axiomes.

Exemple 17.6

Dans le jeu de règles précédent, on peut dériver le jugement $\{1, 2, 3\}$:

$$\frac{\frac{\frac{}{\emptyset}}{\{1\}} \cup \frac{\frac{}{\{1\}} \text{ singl}}{\{1\}}}{\{1\}} \cup \frac{\frac{\frac{}{\{3\}} \text{ singl}}{\{3\}} \cup \frac{\frac{}{\{2\}} \text{ singl}}{\{2\}}}{\{2, 3\}}}{\{1, 2, 3\}} \cup$$

Figure 17.3 — Un arbre de dérivation du jugement $\{1, 2, 3\}$.

On remarque qu'il existe ici de nombreuses dérivations de ce jugement. En fait, comme on peut le voir avec $\{1\} = \{1\} \cup \{1\}$, il est possible d'augmenter artificiellement l'arbre et donc l'ensemble de ces dérivations est infini.

On peut même prouver le résultat suivant : un ensemble est dérivable si et seulement si il est fini.

Remarque 17.7 On rajoute parfois des conditions sur les prémisses pour que la règle s'applique. Il ne s'agit pas d'une nouvelle prémisses, mais juste d'une restriction sur la construction de l'arbre. On peut rapprocher cela de la condition de garde dans un filtrage en OCaml (les when).

Par exemple, pour le jeu de règle sur l'union, on aurait pu demander à ce que l'union soit disjointe :

$$\frac{A \quad B \quad A \cap B = \emptyset}{A \cup B} \cup$$

Figure 17.4 — Règle d'union disjointe, avec sa condition (en violet).

Exemple 17.8

En remplaçant la règle $\cup$ par la règle $\sqcup$ et en enlevant la règle $\emptyset$, le nombre de dérivations d'un ensemble fini est fini : pour un ensemble non vide à n éléments, c'est le nombre d'arbres binaires dont les feuilles sont étiquetées de manière bijective par les éléments de l'ensemble (l'ensemble vide n'est alors plus dérivable). On peut le calculer précisément avec les nombres de Catalan : il y a $C_{n-1} = \frac{1}{n} \binom{2n-2}{n-1}$ arbres binaires à n feuilles, qu'on multiplie par $n!$ pour toutes les affectations possibles aux feuilles.

17.2 Dédution naturelle**17.2.1 Séquent, preuves**

La déduction naturelle est un jeu de règles d'inférence associées à des jugements logiques : les *séquents*.

Définition 17.9

Étant donné un ensemble de formules logiques F , par exemple le calcul propositionnel, on appelle **séquent** pour F , un couple (Γ, f) où $\Gamma \subset F$ et $f \in F$. On le note usuellement $\Gamma \vdash f$, le symbole $\vdash$ se lisant *thèse* ou *donne*.

Remarque 17.10 Un séquent est un jugement qui peut se lire sous la forme intuitive suivante : étant données les hypothèses Γ , je peux déduire f .

Définition 17.11

Dans le contexte des séquents, les dérivations sont appelées des **preuves**.

Quand on peut dériver le jugement $\vdash f$, on dit que f est prouvable.

17.2.2 Système de règles pour le calcul propositionnel

Les règles de la déduction naturelle sont de trois types :

- la règle axiome qui signifie que les jugements $\Delta \vdash f$ où $f \in \Delta$ sont prouvables :

$$\frac{}{\Gamma, A \vdash A} \text{ ax}$$

Figure 17.5 — Règle axiome.

On remarque qu'ici, on a fait le choix de le présenter simplement avec Δ, A , cela ne signifie pas que A est la dernière formule ou que Δ soit non vide, mais juste que le contexte est de la forme $\Delta = \Gamma \cup \{A\}$ ce qui est la forme générale d'un ensemble Δ tel que $A \in \Delta$.

- les règles d'**introduction** qui font apparaître un symbole à la racine de l'arbre de la formule du séquent conclusion ;
- les règles d'**élimination** qui font disparaître un symbole présent à la racine de l'arbre de la formule d'un des séquents prémisses.

Dans la suite, on étiquette par i ou e ces différents types de règles.

Remarque 17.12 Si on construit un arbre de preuve en partant de la conclusion, ce qui est le cas usuel, les règles d'introduction sont assez automatiques : il suffit de regarder le symbole à la racine de la formule à prouver pour en déduire la règle à appliquer. Toute la difficulté repose donc sur la bonne utilisation des règles d'élimination.

On peut aussi raisonner à l'envers en partant des axiomes.

- Connecteur $\rightarrow$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e$$

Figure 17.6 — Règles d'introduction et d'élimination de $\rightarrow$.

Dans la règle $\rightarrow_i$, l'hypothèse A est *déchargée* : elle figure dans le contexte de la prémisse mais plus dans celui de la conclusion. La règle $\rightarrow_e$ est plus communément appelée le *Modus Ponens* dans le cadre de l'étude des raisonnements en philosophie. C'est celle qui permet d'utiliser un résultat intermédiaire A , du moment qu'il est prouvable, pour raisonner. C'est *la principale manière* de faire apparaître des hypothèses intermédiaires quand elles ne sont pas directement dans le contexte.

- Connecteur $\wedge$:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_{e,g} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_{e,d}$$

Figure 17.7 — Règles d'introduction et d'élimination de $\wedge$.

Ici, il y a deux règles d'élimination, une règle gauche et une règle droite, qui indiquent la partie de la conjonction conservée dans le séquent conclusion.

- Connecteur $\vee$:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{i,g} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{i,d} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

Figure 17.8 — Règles d'introduction et d'élimination de $\vee$.

Ici, la règle d'introduction est latéralisée car elle correspond au côté de la disjonction que l'on prouve. *Attention* il y a une perte d'information ici et ce n'est pas toujours une bonne idée de faire cette règle sans avoir une bonne raison de choisir la gauche ou la droite.

La règle d'élimination est plus communément appelée une *disjonction de cas* en mathématiques. Ici, il faut s'assurer que la formule sur laquelle s'effectue la disjonction est prouvable puis faire la preuve dans chacun des deux cas, les hypothèses A et B étant déchargées.

- Connecteur $\neg$:

$$\frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} \neg_i \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \perp} \neg_e \qquad \frac{\Gamma \vdash \perp}{\Gamma \vdash C} \perp_e$$

Figure 17.9 — Règles pour $\neg$ et élimination de $\perp$.

La règle d'introduction de la négation revient en fait à considérer que $\neg A = A \rightarrow \perp$: c'est une règle $\rightarrow_i$ déguisée, où l'hypothèse A est déchargée. La règle d'élimination de la négation est souvent confondue avec le raisonnement par l'absurde. Il s'agit juste de montrer une formule et sa négation pour en déduire une contradiction.

La règle d'élimination de l'absurde $\perp_e$ (*ex falso quodlibet*) permet de déduire n'importe quelle formule C d'une contradiction : c'est elle qui permet d'exploiter une contradiction obtenue par $\neg_e$.

Attention, le raisonnement par l'absurde proprement dit, qui consiste à déduire $\neg A$ de $\neg A \rightarrow \perp$, n'est pas une règle de ce système : avec les règles ci-dessus, on obtient seulement $\neg A$ par $\neg_i$. On verra qu'il faut pour cela se placer en logique classique.

Exemple 17.13

Voici une preuve de $(A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)$:

$$\begin{array}{c}
 \frac{(A \vee B) \wedge C \vdash (A \vee B) \wedge C}{(A \vee B) \wedge C \vdash A \vee B} \wedge_{e,g} \quad \frac{\frac{A, (A \vee B) \wedge C \vdash A}{A, (A \vee B) \wedge C \vdash C} \wedge_i \quad \frac{A, (A \vee B) \wedge C \vdash (A \vee B) \wedge C}{A, (A \vee B) \wedge C \vdash C} \wedge_{e,d}}{A, (A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)} \vee_{i,g} \quad \frac{\frac{B, (A \vee B) \wedge C \vdash B}{B, (A \vee B) \wedge C \vdash C} \wedge_i \quad \frac{B, (A \vee B) \wedge C \vdash (A \vee B) \wedge C}{B, (A \vee B) \wedge C \vdash C} \wedge_{e,d}}{B, (A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)} \vee_{e,d} \\
 \hline
 (A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)
 \end{array}$$

Figure 17.10 — Preuve de $(A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)$; conclusion en vert.

Le théorème suivant affirme la **correction** de la déduction naturelle : elle ne prouve rien qui ne soit pas une conséquence logique. Ici, pour Γ éventuellement infini, on note $\Gamma \models f$ lorsque toute valuation rendant vraies toutes les formules de Γ rend vraie f (pour Γ fini, cela coïncide avec la définition du chapitre sur le calcul propositionnel).

Théorème 17.14

Soit $Prop$ l'ensemble des formules du calcul propositionnel et $\Gamma \subset Prop$, $f \in Prop$.

Si $\Gamma \vdash f$ est prouvable alors $\Gamma \models f$.

Preuve On montre par induction structurale sur la preuve que tout séquent prouvable $\Delta \vdash \varphi$ vérifie $\Delta \models \varphi$. L'hypothèse d'induction porte donc sur les séquents prémisses, dont le contexte peut différer de celui de la conclusion (règles $\rightarrow_i$, $\vee_e$, $\neg_i$).

Dans chaque cas, on note Γ le contexte du séquent conclusion et on considère une valuation v telle que pour tout $f \in \Gamma$, $[f]_v = V$.

- Si la preuve est réduite à un axiome $\Gamma, A \vdash A$, alors comme A est dans le contexte, $[A]_v = V$. Ainsi $\Gamma, A \models A$.
- Si la preuve est obtenue par une règle $\rightarrow_i$ à partir de $\Gamma, A \vdash B$, on raisonne sur $[A]_v$:

Si $[A]_v = V$, alors v rend vraies toutes les formules de $\Gamma \cup \{A\}$ et comme, par hypothèse d'induction, $\Gamma, A \models B$, on a forcément $[B]_v = V$ et ainsi $[A \rightarrow B]_v = V$.

Si $[A]_v = F$, alors trivialement $[A \rightarrow B]_v = V$.

Dans tous les cas, on a bien $[A \rightarrow B]_v = V$, ainsi $\Gamma \models A \rightarrow B$.

- Si la preuve est obtenue par une règle $\rightarrow_e$, comme $\Gamma \vdash A \rightarrow B$ et $\Gamma \vdash A$ sont prouvables, par hypothèses d'induction on a $[A \rightarrow B]_v = [A]_v = V$ et cela force $[B]_v = V$ selon la table de vérité de l'implication. Ainsi $\Gamma \models B$.
- Si la preuve est obtenue par une règle $\wedge_i$, de $\Gamma \vdash A$ et $\Gamma \vdash B$ prouvables, on peut en déduire par hypothèse d'induction que $[A]_v = [B]_v = V$, et donc en vertu de la table de vérité de $\wedge$, on a $[A \wedge B]_v = V$. Donc, $\Gamma \models A \wedge B$.
- Si la preuve est obtenue par une règle $\wedge_{e,g}$ ou $\wedge_{e,d}$, on a $\Gamma \vdash A \wedge B$ prouvable donc par hypothèse d'induction $[A \wedge B]_v = V$; le seul cas de la table de vérité est celui où $[A]_v = [B]_v = V$ et donc $\Gamma \models A$ ainsi que $\Gamma \models B$.
- Pour $\vee_{i,g}$ (resp. $\vee_{i,d}$), $[A]_v = V$ (resp. $[B]_v = V$) entraîne $[A \vee B]_v = V$.
- Pour $\vee_e$, par hypothèse d'induction $[A \vee B]_v = V$, donc $[A]_v = V$ ou $[B]_v = V$. Dans le premier cas, v rend vraies les formules de $\Gamma \cup \{A\}$ et $\Gamma, A \models C$ donne $[C]_v = V$; le second cas est symétrique.
- Pour $\neg_i$, si on avait $[A]_v = V$, l'hypothèse d'induction $\Gamma, A \models \perp$ donnerait $[\perp]_v = V$, ce qui est impossible. Donc $[A]_v = F$ et $[\neg A]_v = V$.

- Pour $\neg_e$ et $\perp_e$, les hypothèses d'induction donneraient $[A]_v = [\neg A]_v = V$ (resp. $[\perp]_v = V$), ce qui est impossible : il n'existe donc aucune valuation v rendant vraies les formules de Γ , et $\Gamma \models \varphi$ pour toute formule φ .

□

Corollaire 17.15

Les formules prouvables sont des tautologies.

17.2.3 Logique classique

Certaines formules dont on sait qu'elles sont des tautologies du calcul propositionnel ne semblent pas prouvables.

La formule la plus emblématique dans ce cas est le tiers exclu : $\vdash A \vee \neg A$ n'est pas prouvable.

Remarque 17.16 Comment montrer qu'une formule n'est pas prouvable ?

Ce n'est pas une question facile à traiter directement, car cela nécessite de raisonner sur l'ensemble des arbres de preuve. La manière usuelle de le montrer est de le faire de manière indirecte :

- on construit un modèle de la déduction, c'est-à-dire une sémantique qui vérifie les règles. Dans la partie précédente, on a construit un modèle booléen en liant la prouvabilité à la conséquence logique.
- on montre que la formule n'est pas vraie dans ce modèle (attention, sur des modèles non booléens, cela ne signifie pas qu'elle est fausse).
- si la formule était prouvable, la correction du modèle entraînerait que la formule serait vraie.
- donc, la formule n'est pas prouvable.

Pour le tiers exclu, il faut utiliser un modèle non booléen, par exemple une sémantique à trois valeurs de vérité comme celle de Gödel : c'est l'objet de la partie bonus [Sémantique de Gödel](#).

On parle de déduction naturelle classique quand on ajoute la règle axiome suivante (tiers exclu) :

$$\frac{}{\Gamma \vdash A \vee \neg A} \text{cl}$$

Figure 17.11 — Règle du tiers exclu.

👉 Sans cette règle, on parle de déduction naturelle *intuitionniste*. Les formules prouvées dans ce cadre sont encore des tautologies : la preuve de correction s'étend directement, car $A \vee \neg A$ est une tautologie.

Remarque 17.17 L'usage usuel de cette règle est de la combiner avec une disjonction par cas :

$$\frac{\frac{}{\Gamma \vdash A \vee \neg A} \text{cl} \quad \frac{\Gamma, A \vdash B \quad \Gamma, \neg A \vdash B}{\Gamma \vdash B} \vee_e}{\Gamma \vdash B}$$

Figure 17.12 — Tiers exclu combiné à une disjonction par cas.

👉 **Remarque 17.18** En logique classique, le raisonnement par l'absurde devient une règle dérivée : si $\Gamma, \neg A \vdash \perp$ est prouvable, on obtient $\Gamma, \neg A \vdash A$ par $\perp_e$, et $\Gamma, A \vdash A$ est un axiome ; la disjonction de cas sur $A \vee \neg A$ ci-dessus donne alors $\Gamma \vdash A$. Réciproquement, on peut montrer qu'en ajoutant la règle du raisonnement par l'absurde (ou l'élimination de la double négation $\neg\neg A \vdash A$) au système intuitionniste, on peut prouver le tiers exclu.

On peut alors démontrer le théorème suivant auquel on fait référence sous le terme de **complétude** :

Théorème 17.19

Soit $Prop$ l'ensemble des formules du calcul propositionnel et $\Gamma \subset Prop, f \in Prop$.

Si $\Gamma \models f$ alors le séquent $\Gamma \vdash f$ est prouvable en déduction naturelle classique.

La preuve de ce théorème est hors de portée de ce cours. Ce théorème est faux pour la déduction naturelle intuitionniste, puisque le tiers exclu est une tautologie qui n'y est pas prouvable.

17.2.4 Règles pour la logique du premier ordre

On considère ici F les formules de la logique du premier ordre.

Variables libres

Si $f \in F$, on dit qu'une variable est libre dans f quand elle a au moins une occurrence dans f qui n'est pas liée à un quantificateur. On note $FV(f)$ les variables libres de f .

Exemple 17.20

Dans $f \equiv y = x \wedge \forall x. \exists z. x < z$, les variables x et y sont libres, la variable x est aussi liée à droite dans la formule, mais elle est libre si au moins une de ses occurrences est libre.

On a donc $FV(f) = \{x, y\}$.

Remarque 17.21 On utilise ici la notation FV pour *free variables* afin d'éviter l'ambiguïté d'un VL pouvant faire référence également aux variables liées.

α -équivalence

Si on considère une formule $\forall x.P$ et que l'on note P' la formule obtenue en changeant toutes les occurrences libres de x dans P par la variable y **supposée non libre dans P** (et telle qu'aucune occurrence libre de x dans P ne se trouve sous un quantificateur liant y , sinon y serait capturée), on dit que la formule $\forall y.P'$ est α -équivalente à $\forall x.P$. On étend cela naturellement au quantificateur $\exists$ ainsi qu'à des quantificateurs à profondeur quelconque. On pourra noter $f \equiv_{\alpha} g$ pour dire que f et g sont α -équivalentes.

Bien définir cette équivalence est délicat, on passe outre toute difficulté ici.

Exemple 17.22

$$\forall x.P(x, y) \rightarrow \exists y.R(y, x) \equiv_{\alpha} \forall t.P(t, y) \rightarrow \exists z.R(z, t)$$

Ici, la première occurrence de y est libre et ne peut changer par α -équivalence.

Remarque 17.23 On connaît déjà cette notion dans deux contextes :

- en mathématiques quand on change la variable d'intégration
- en informatique quand on renomme les variables

On considère ici que les règles s'appliquent modulo α -équivalence. Ainsi l'application suivante est bien un axiome :

$$\frac{}{\forall x.P(x) \vdash \forall y.P(y)} \text{ax}$$

Figure 17.13 — Un axiome modulo α -équivalence.

Substitution

Soit f une formule, x une variable et t un terme, on note $f[x \setminus t]$ la formule obtenue en remplaçant chaque occurrence libre de x dans f par t , après avoir renommé (par α -équivalence) les variables liées de f qui sont libres dans t : ainsi aucune variable de t n'est capturée par un quantificateur de f .

Exemple 17.24

Si $f \equiv \forall x.P(x, y) \rightarrow P(x, z)$ et g est un symbole de fonction, alors $f[y \setminus g(x, z)] = \forall x'. P(x', g(x, z)) \rightarrow P(x', z)$: on a d'abord renommé la variable liée x en x' , sinon l'occurrence libre de x dans $g(x, z)$ aurait été capturée par le quantificateur $\forall x$.

On remarque que $f[x \setminus t] = f$ car toutes les occurrences de x sont liées.

Règles pour $\forall, \exists$

On conserve les règles précédentes et on ajoute les règles suivantes pour les quantificateurs.

- Règles pour $\forall$:

$$\frac{\Gamma \vdash A \quad x \notin FV(\Gamma)}{\Gamma \vdash \forall x. A} \forall_i \quad \frac{\Gamma \vdash \forall x. A}{\Gamma \vdash A[x/t]} \forall_e$$

Figure 17.14 — Règles pour $\forall$; la condition sur les variables libres est en violet.

Pour bien comprendre ces règles, il est nécessaire de les rapprocher de l'usage qu'on en fait en mathématiques. La règle d'introduction correspond au moment où on introduit une variable dans une preuve en écrivant **Soit x ...*. La condition de non présence de x comme variable libre dans le contexte est très importante. Sinon, on pourrait prouver que tous les nombres sont irrationnels en commençant par **Soit π ** et en utilisant le fait que π est irrationnel. Quand on introduit un nom avec **Soit** en mathématiques, on veut le faire en toute généralité.

La règle d'élimination est au contraire une règle de spécialisation : si on sait démontrer un résultat pour tous les x , on sait en déduire le résultat dans le cas d'un x particulier, remplacé par un terme t quelconque.

- Règles pour $\exists$:

$$\frac{\Gamma \vdash A[x/t]}{\Gamma \vdash \exists x. A} \exists_i \quad \frac{\Gamma \vdash \exists x. A \quad \Gamma, A \vdash B \quad x \notin FV(\Gamma, B)}{\Gamma \vdash B} \exists_e$$

Figure 17.15 — Règles pour $\exists$; la condition sur les variables libres est en violet.

La règle d'introduction correspond en mathématiques au fait de donner un témoin d'existence d'une propriété. Ici, on affirme que c'est $x = t$ qui fonctionne.

La règle d'élimination correspond au raisonnement **Soit x tel que A ** : sachant qu'il existe un x vérifiant A , on introduit un tel x et l'hypothèse A (qui possède x comme variable libre) pour prouver B . L'hypothèse A est alors déchargée. La condition $x \notin FV(\Gamma, B)$ assure que x est un nom nouveau, sur lequel on ne sait rien d'autre que A , et qu'il n'apparaît pas dans la conclusion.

Remarque 17.25 Les quantificateurs rendent complexe l'application des règles. Il faut s'assurer que ce que l'on écrit a un sens mathématique.

17.2.5 Règles hypothèses

Attention on ne peut pas utiliser ces règles au concours sans démontrer qu'elles fonctionnent avant.

On peut rajouter des **méta-règles**, c'est-à-dire des configurations pour lesquelles, quitte à appliquer une succession de règles, on peut passer d'un séquent conclusion à des séquents prémisses.

On a ainsi pu voir une telle méta-règle avec la disjonction sur le tiers exclu, on pourrait ainsi la noter :

$$\frac{\Gamma, A \vdash B \quad \Gamma, \neg A \vdash B}{\Gamma \vdash B} \vee_{ex}$$

Figure 17.16 — Méta-règle de disjonction sur le tiers exclu.

Le cas usuel de règles intéressantes sont les règles hypothèses qui permettent de raisonner directement sur les hypothèses :

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge_h$$

Figure 17.17 — Méta-règle hypothèse pour $\wedge$.

17.3 Utilisation d'un assistant de preuve (Bonus)

Construire un arbre de preuve à la main est fastidieux et il est facile d'y glisser une erreur, notamment sur les contextes. Un **assistant de preuve** est un logiciel qui construit l'arbre avec nous et vérifie que chaque règle est correctement appliquée. L'assistant **dedunat** mentionné dans les exercices suit exactement les règles de ce chapitre. Les assistants utilisés en recherche et dans l'industrie, comme **Rocq** (anciennement **Coq**, développé en France) ou **Lean**, reposent sur les mêmes principes.

On y travaille comme on l'a conseillé plus haut : en partant de la conclusion. On énonce le séquent à prouver, appelé le **but**, puis on applique des **tactiques** qui remplacent le but par les prémisses d'une règle. L'assistant affiche à chaque étape le contexte et le ou les buts restants. La preuve est terminée quand il n'y a plus de but.

Exemple 17.26

Voici en Rocq la preuve de $(A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)$ donnée en exemple plus haut, présentée ici sous la forme $\vdash (A \vee B) \wedge C \rightarrow (A \wedge C) \vee (B \wedge C)$. En commentaire, on indique la règle correspondante.

```
Theorem distrib (A B C : Prop) : (A ∨ B) ∧ C -> (A ∧ C) ∨ (B ∧ C).
Proof.
  intro H.                (* ->i : H : (A ∨ B) ∧ C dans le contexte *)
  destruct H as [HAB HC].  (* /\e,g et /\e,d sur l'hypothèse H *)
  destruct HAB as [HA | HB]. (* \e : deux buts, avec HA : A puis HB : B *)
  - left.                  (* \i,g *)
    split.                 (* /\i : deux buts A et C *)
    + exact HA.            (* ax *)
    + exact HC.            (* ax *)
  - right.                 (* \i,d *)
    split.                 (* /\i *)
    + exact HB.            (* ax *)
    + exact HC.            (* ax *)
Qed.
```

Les puces - et + correspondent aux branches de l'arbre de preuve.

Le tableau suivant donne la correspondance entre les tactiques de base de Rocq et les règles de la déduction naturelle.

Tactique	Règle
exact H ou assumption	ax
intro H	$\rightarrow_i$, et aussi $\neg_i$ car $\neg A$ est défini comme $A \rightarrow \perp$
apply H avec H : A -> B	$\rightarrow_e$: le but B est remplacé par A
split	$\wedge_i$
destruct H as [H1 H2] avec H : A /\ B	$\wedge_{e,g}$ et $\wedge_{e,d}$
left, right	$\vee_{i,g}$, $\vee_{i,d}$
destruct H as [H1 H2] avec H : A \/ B	$\vee_e$
exfalso	$\perp_e$
intro x sur un but $\forall x.A$	$\forall_i$
exists t	$\exists_i$

Remarque 17.27

- Les tactiques **destruct** agissent sur une hypothèse du contexte plutôt que sur le but : ce sont les règles hypothèses vues plus haut.
- Rocq est intuitionniste par défaut : le tiers exclu n'est pas prouvable sans l'ajouter explicitement comme axiome (par exemple avec `Require Import Classical.`, qui fournit `classic : forall P, P ∨ ~ P`).

- En fait, Rocq ne stocke pas l'arbre mais un programme : la commande `Show Proof`. affiche un terme fonctionnel qui ressemble beaucoup à du OCaml, avec un `fun` pour `intro` et un `match` pour chaque `destruct`. C'est la correspondance de Curry-Howard présentée dans le chapitre [Introduction au lambda-calcul](#).

17.4 Sémantique de Gödel (Bonus)



On a vu que le tiers exclu $A \vee \neg A$ est une tautologie mais qu'il n'est pas prouvable en déduction naturelle intuitionniste. Pour le démontrer, on suit la méthode de la remarque *Comment montrer qu'une formule n'est pas prouvable ?* : on construit un modèle à trois valeurs de vérité, dû à Gödel (1932), dans lequel toutes les formules prouvables sont vraies mais pas le tiers exclu.

L'idée est d'ajouter aux valeurs 0 (faux) et 1 (vrai) une valeur intermédiaire $\frac{1}{2}$, qu'on peut lire comme « pas encore démontré ».

Définition 17.28

On pose $G = \{0, \frac{1}{2}, 1\}$ muni de l'ordre usuel. Une **valuation de Gödel** est une application v des variables propositionnelles dans G . On l'étend aux formules par :

- $[\perp]_v = 0$;
- $[A \wedge B]_v = \min([A]_v, [B]_v)$ et $[A \vee B]_v = \max([A]_v, [B]_v)$;
- $[A \rightarrow B]_v = 1$ si $[A]_v \leq [B]_v$ et $[A \rightarrow B]_v = [B]_v$ sinon ;
- $[\neg A]_v = [A \rightarrow \perp]_v$, c'est-à-dire 1 si $[A]_v = 0$ et 0 sinon.

Pour un ensemble de formules Γ , on pose $[\Gamma]_v = \min\{[f]_v \mid f \in \Gamma\}$, avec la convention $[\emptyset]_v = 1$ (ce minimum existe car G est fini).

Remarque 17.29 Restreinte à $\{0, 1\}$, cette sémantique coïncide avec les tables de vérité usuelles. Une formule qui vaut 1 pour toute valuation de Gödel est donc en particulier une tautologie.

Exemple 17.30

Pour $[A]_v = \frac{1}{2}$, on a $[\neg A]_v = 0$ et donc $[A \vee \neg A]_v = \max(\frac{1}{2}, 0) = \frac{1}{2} \neq 1$.

De même $[\neg \neg A]_v = 1$ et ainsi $[\neg \neg A \rightarrow A]_v = \frac{1}{2}$.

La propriété clé de l'implication est la suivante, à rapprocher des règles $\rightarrow_i$ et $\rightarrow_e$.

Lemme 17.31

Pour tous $a, b, c \in G$, en notant $a \Rightarrow b$ la valeur de l'implication, on a

$$c \leq (a \Rightarrow b) \Leftrightarrow \min(c, a) \leq b$$

Preuve

- Si $a \leq b$, alors $a \Rightarrow b = 1$ donc l'inégalité de gauche est vraie, et $\min(c, a) \leq a \leq b$ donc celle de droite aussi.
- Si $a > b$, alors $a \Rightarrow b = b$. Si $c \leq b$, alors $\min(c, a) \leq c \leq b$. Réciproquement, si $\min(c, a) \leq b$, comme $a > b$, ce minimum ne peut pas être a , c'est donc c et $c \leq b$.

□

Théorème 17.32

Si $\Gamma \vdash f$ est prouvable en déduction naturelle intuitionniste, alors pour toute valuation de Gödel v , $[\Gamma]_v \leq [f]_v$.

En particulier, si f est prouvable, alors $[f]_v = 1$ pour toute valuation de Gödel v .

Preuve Comme pour la correction booléenne, on procède par induction sur la preuve. On fixe v , on note Γ le contexte de la conclusion, $c = [\Gamma]_v$ et on note simplement $a = [A]_v$, $b = [B]_v$, $d = [C]_v$. On remarque que $[\Gamma, A]_v = \min(c, a)$.

- Axiome : $\min(c, a) \leq a$.
- $\rightarrow_i$: l'hypothèse d'induction donne $\min(c, a) \leq b$, donc $c \leq (a \Rightarrow b)$ par le lemme.
- $\rightarrow_e$: on a $c \leq (a \Rightarrow b)$ et $c \leq a$. Par le lemme, $\min(c, a) \leq b$ et comme $\min(c, a) = c$, $c \leq b$.
- $\wedge_i$, $\wedge_{e,g}$, $\wedge_{e,d}$: $c \leq a$ et $c \leq b$ équivaut à $c \leq \min(a, b)$.
- $\vee_{i,g}$, $\vee_{i,d}$: si $c \leq a$ ou $c \leq b$, alors $c \leq \max(a, b)$.
- $\vee_e$: on a $c \leq \max(a, b)$, $\min(c, a) \leq d$ et $\min(c, b) \leq d$. Si $a \geq b$, alors $c \leq a$ donc $c = \min(c, a) \leq d$. Sinon $c \leq b$ et $c = \min(c, b) \leq d$.
- $\neg_i$ et $\neg_e$: ce sont les cas $\rightarrow_i$ et $\rightarrow_e$ avec $B = \perp$, puisque $[\neg A]_v = [A \rightarrow \perp]_v$.
- $\perp_e$: on a $c \leq [\perp]_v = 0$ donc $c = 0 \leq d$.

□

Corollaire 17.33

Les formules $A \vee \neg A$, $\neg\neg A \rightarrow A$ et $((A \rightarrow B) \rightarrow A) \rightarrow A$ (loi de Peirce) ne sont pas prouvables en déduction naturelle intuitionniste.

Preuve D'après le théorème, il suffit de trouver une valuation de Gödel qui ne donne pas la valeur 1 à ces formules. L'exemple précédent traite les deux premières avec $[A]_v = \frac{1}{2}$. Pour la loi de Peirce, avec $[A]_v = \frac{1}{2}$ et $[B]_v = 0$, on a $[A \rightarrow B]_v = 0$, puis $[(A \rightarrow B) \rightarrow A]_v = 1$ et enfin $[((A \rightarrow B) \rightarrow A) \rightarrow A]_v = \frac{1}{2}$.

□

Remarque 17.34

- Ce modèle n'est pas complet pour la logique intuitionniste : certaines formules valent toujours 1 sans être prouvables, par exemple $(A \rightarrow B) \vee (B \rightarrow A)$. Il sert uniquement à montrer que des formules **ne** sont **pas** prouvables.
- On peut obtenir des modèles complets de la logique intuitionniste en remplaçant G par des structures plus riches (algèbres de Heyting, modèles de Kripke), ce qui dépasse le cadre de ce cours.
- Ce résultat ne doit pas être confondu avec le *théorème de complétude de Gödel* (1929), qui affirme que toute formule du premier ordre vraie dans tous les modèles est prouvable en logique classique : c'est l'analogue au premier ordre du théorème de complétude énoncé plus haut.

17.5 Lambda-calcul simplement typé (Bonus)

Cette partie a été déplacée dans son propre chapitre [Introduction au lambda-calcul](#).

17.6 Exercices

Pour réaliser les preuves, vous pouvez utiliser l'assistant de preuve dédié au programme [dedunat](#).

Exercice

Démontrer en déduction naturelle :

1. $(A \rightarrow B) \wedge (B \rightarrow C) \rightarrow A \rightarrow C$
2. $A \wedge B \rightarrow B \wedge A$
3. $(A \vee B \rightarrow C) \rightarrow A \rightarrow C$
4. $A \vee (A \wedge B) \rightarrow A$
5. $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
6. $\neg A \vee \neg B \rightarrow \neg(A \wedge B)$
7. $\neg A \wedge \neg B \rightarrow \neg(A \vee B)$
8. $(A \rightarrow (B \rightarrow (C \rightarrow D))) \rightarrow (B \rightarrow C \wedge A \rightarrow D)$

9. $(\neg A \vee B) \rightarrow (A \rightarrow B)$
 10. $A \rightarrow \neg \neg A$

Réponse 1.

$$\frac{\frac{\frac{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash (A \rightarrow B) \wedge (B \rightarrow C)}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash B \rightarrow C} \text{ax}}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash B \rightarrow C} \wedge_{e,d} \quad \frac{\frac{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash (A \rightarrow B) \wedge (B \rightarrow C)}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash A \rightarrow B} \text{ax}}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash A \rightarrow B} \wedge_{e,g} \quad \frac{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash A \rightarrow B}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash B} \text{ax}}{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash B} \rightarrow_e$$

$$\frac{(A \rightarrow B) \wedge (B \rightarrow C), A \vdash B}{(A \rightarrow B) \wedge (B \rightarrow C) \vdash A \rightarrow B} \rightarrow_i$$

$$\frac{(A \rightarrow B) \wedge (B \rightarrow C) \vdash A \rightarrow B}{\vdash (A \rightarrow B) \wedge (B \rightarrow C) \rightarrow A \rightarrow B} \rightarrow_i$$

2.

$$\frac{\frac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash B} \text{ax}}{A \wedge B \vdash B} \wedge_{e,d} \quad \frac{\frac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A} \text{ax}}{A \wedge B \vdash A} \wedge_{e,g}$$

$$\frac{A \wedge B \vdash B \wedge A}{\vdash (A \wedge B) \rightarrow (B \wedge A)} \wedge_i$$

$$\frac{A \wedge B \vdash B \wedge A}{\vdash (A \wedge B) \rightarrow (B \wedge A)} \rightarrow_i$$

3.

$$\frac{\frac{A, (A \vee B) \rightarrow C \vdash (A \vee B) \rightarrow C}{A, (A \vee B) \rightarrow C \vdash C} \text{ax}}{A, (A \vee B) \rightarrow C \vdash C} \wedge_{e,d} \quad \frac{\frac{A, (A \vee B) \rightarrow C \vdash A}{A, (A \vee B) \rightarrow C \vdash A \vee B} \text{ax}}{A, (A \vee B) \rightarrow C \vdash A \vee B} \wedge_{e,g}$$

$$\frac{A, (A \vee B) \rightarrow C \vdash C}{(A \vee B) \rightarrow C \vdash A \rightarrow C} \rightarrow_i$$

$$\frac{(A \vee B) \rightarrow C \vdash A \rightarrow C}{\vdash ((A \vee B) \rightarrow C) \rightarrow (A \rightarrow C)} \rightarrow_i$$

4.

$$\frac{\frac{A \vee (A \wedge B) \vdash A \vee (A \wedge B)}{A \vee (A \wedge B) \vdash A} \text{ax}}{A \vee (A \wedge B) \vdash A} \wedge_{e,d} \quad \frac{\frac{A, A \vee (A \wedge B) \vdash A}{A \vee (A \wedge B) \vdash A} \text{ax}}{A \vee (A \wedge B) \vdash A} \wedge_{e,g}$$

$$\frac{A \vee (A \wedge B) \vdash A}{\vdash (A \vee (A \wedge B)) \rightarrow A} \rightarrow_i$$

5.

$$\frac{\frac{A, \neg(A \vee B) \vdash A}{A, \neg(A \vee B) \vdash A \vee B} \text{ax}}{A, \neg(A \vee B) \vdash A \vee B} \wedge_{e,d} \quad \frac{\frac{A, \neg(A \vee B) \vdash \neg(A \vee B)}{A, \neg(A \vee B) \vdash \perp} \text{ax}}{A, \neg(A \vee B) \vdash \perp} \wedge_{e,g}$$

$$\frac{A, \neg(A \vee B) \vdash \perp}{\neg(A \vee B) \vdash \neg A} \rightarrow_i$$

$$\frac{\neg(A \vee B) \vdash \neg A}{\neg(A \vee B) \vdash \neg A \wedge \neg B} \wedge_i$$

$$\frac{\neg(A \vee B) \vdash \neg A \wedge \neg B}{\vdash \neg(A \vee B) \rightarrow (\neg A \wedge \neg B)} \rightarrow_i$$

6.

$$\frac{\frac{A \wedge B, \neg A \vee \neg B \vdash \neg A \vee \neg B}{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A} \text{ax}}{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A} \wedge_{e,d} \quad \frac{\frac{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A}{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A} \text{ax}}{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A} \wedge_{e,g}$$

$$\frac{\neg A, A \wedge B, \neg A \vee \neg B \vdash \neg A}{\neg A, A \wedge B, \neg A \vee \neg B \vdash \perp} \wedge_{e,d}$$

$$\frac{\neg A, A \wedge B, \neg A \vee \neg B \vdash \perp}{\neg A \vee \neg B \vdash \neg(A \wedge B)} \rightarrow_i$$

$$\frac{\neg A \vee \neg B \vdash \neg(A \wedge B)}{\vdash (\neg A \vee \neg B) \rightarrow \neg(A \wedge B)} \rightarrow_i$$

7.

$$\begin{array}{c}
\frac{\frac{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash A \rightarrow (B \rightarrow (C \rightarrow D))}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash B \rightarrow (C \rightarrow D)} \text{ax} \quad \frac{\frac{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash C \wedge A}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash A} \wedge_{e,d}}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash C \rightarrow D} \rightarrow_e \\
\\
\frac{\frac{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash B}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash B \rightarrow (C \rightarrow D)} \text{ax} \quad \frac{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash C \wedge A}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash C} \wedge_{e,g}}{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash C \rightarrow D} \rightarrow_e \\
\\
\frac{\frac{C \wedge A, B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash D}{B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash (C \wedge A) \rightarrow D} \rightarrow_i \quad \frac{B, A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash (C \wedge A) \rightarrow D}{A \rightarrow (B \rightarrow (C \rightarrow D)) \vdash B \rightarrow ((C \wedge A) \rightarrow D)} \rightarrow_i}{\vdash (A \rightarrow (B \rightarrow (C \rightarrow D))) \rightarrow (B \rightarrow ((C \wedge A) \rightarrow D))} \rightarrow_i
\end{array}$$
$$\frac{\frac{\frac{A, \neg A \vee B \vdash \neg A \vee B}{\neg A, A, \neg A \vee B \vdash \neg A \vee B} ax}{\neg A, A, \neg A \vee B \vdash \perp} \perp_e}{\neg A, A, \neg A \vee B \vdash B} \rightarrow_i \quad \frac{\frac{\frac{B, A, \neg A \vee B \vdash B}{\neg A, A, \neg A \vee B \vdash B} ax}{\neg A, A, \neg A \vee B \vdash \neg A} \neg_e}{\neg A, A, \neg A \vee B \vdash A} \rightarrow_i \quad \frac{\frac{\frac{A, \neg A \vee B \vdash B}{\neg A \vee B \vdash A \rightarrow B} \rightarrow_i}{\vdash (\neg A \vee B) \rightarrow (A \rightarrow B)} \rightarrow_i}{\vdash (\neg A \vee B) \rightarrow (A \rightarrow B)} \vee_e$$
$$\frac{\frac{\frac{}{\neg A, A \vdash A} ax \quad \frac{}{\neg A, A \vdash \neg A} ax}{\neg A, A \vdash \perp} \neg_e}{A \vdash \neg \neg A} \neg_i \quad \rightarrow_i$$

- a) $\neg\neg A \rightarrow A$
- b) $(A \rightarrow B) \rightarrow \neg A \vee B$
- c) $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- d) $((A \rightarrow B) \rightarrow A) \rightarrow A$ (loi de Peirce)

$$\frac{\frac{\frac{}{\neg A \vdash A \vee \neg A} cl \quad \frac{\frac{}{A, \neg A \vdash A} ax}{\neg A \vdash A} \rightarrow_i}{\vdash \neg A \rightarrow A} \rightarrow_i \quad \frac{\frac{\frac{\frac{}{\neg A, \neg A \vdash \neg A} ax \quad \frac{}{\neg A, \neg A \vdash \neg \neg A} ax}{\neg A, \neg A \vdash \perp} \neg_e \quad \frac{}{\neg A, \neg A \vdash A} \bot_e}{\neg A, \neg A \vdash A} \vee_e}{\vdash \neg A \rightarrow A} \rightarrow_i$$

100

$$\begin{array}{c}
\frac{A \rightarrow B \vdash A \vee \neg A}{A \rightarrow B \vdash A \vee \neg A} cl \\
\frac{\frac{A, A \rightarrow B \vdash A \rightarrow B}{A, A \rightarrow B \vdash B} ax \quad \frac{A, A \rightarrow B \vdash A}{A, A \rightarrow B \vdash \neg A \vee B} \rightarrow_e \quad \frac{\frac{A, A \rightarrow B \vdash \neg A}{A, A \rightarrow B \vdash \neg A \vee B} \vee_{i,g} \quad \frac{A, A \rightarrow B \vdash \neg A \vee B}{A \rightarrow B \vdash \neg A \vee B} \vee_e}{\vdash (A \rightarrow B) \rightarrow (\neg A \vee B)} \rightarrow_i
\end{array}$$

C.

$$\begin{array}{c}
\frac{\frac{B, A, \neg(A \wedge B) \vdash A}{B, A, \neg(A \wedge B) \vdash A \wedge B} ax \quad \frac{B, A, \neg(A \wedge B) \vdash B}{B, A, \neg(A \wedge B) \vdash A \wedge B} ax \quad \frac{B, A, \neg(A \wedge B) \vdash A \wedge B}{B, A, \neg(A \wedge B) \vdash \neg(A \wedge B)} \neg_e}{\frac{B, A, \neg(A \wedge B) \vdash \perp}{A, \neg(A \wedge B) \vdash \neg B} \neg_i \quad \frac{A, \neg(A \wedge B) \vdash \neg B}{A, \neg(A \wedge B) \vdash \neg A \vee \neg B} \vee_{i,d} \quad \frac{\frac{A, \neg(A \wedge B) \vdash \neg A}{A, \neg(A \wedge B) \vdash \neg A \vee \neg B} ax \quad \frac{A, \neg(A \wedge B) \vdash \neg A \vee \neg B}{\neg(A \wedge B) \vdash \neg A \vee \neg B} \vee_e}{\vdash \neg(A \wedge B) \rightarrow (\neg A \vee \neg B)} \rightarrow_i
\end{array}$$

d.

$$\begin{array}{c}
\frac{A \rightarrow B \rightarrow A \vdash A \vee \neg A}{A \rightarrow B \rightarrow A \vdash A \vee \neg A} cl \quad \frac{A, (A \rightarrow B) \rightarrow A \vdash A}{A, (A \rightarrow B) \rightarrow A \vdash A} ax \quad \frac{\frac{A, (A \rightarrow B) \rightarrow A \vdash A}{A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow A} ax \quad \frac{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow \neg A}{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow B} \neg_e \quad \frac{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow B}{\neg A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow B} \rightarrow_i}{\frac{A, (A \rightarrow B) \rightarrow A \vdash A \rightarrow B}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A} \rightarrow_i}
\end{array}$$

Exercice

Démontrez en déduction naturelle :

- a) $(\forall x.P(x)) \rightarrow \exists x.P(x)$
- b) $\exists x.\forall y.P(x, y) \rightarrow \forall y.\exists x.P(x, y)$
- c) $P(a) \rightarrow Q(a) \rightarrow \exists x.P(x) \wedge Q(x)$
- d) $(\forall x.P(x) \rightarrow Q(x)) \rightarrow P(a) \rightarrow Q(a)$
- e) $(\forall x.P(x) \rightarrow Q(x)) \rightarrow (\forall x.Q(x) \rightarrow R(x)) \rightarrow (\forall x.P(x) \rightarrow R(x))$

Réponse J'ai fait le choix ici d'utiliser des nouveaux noms, comme x_0 , pour bien marquer l'application des règles. Techniquement, on peut tout faire en utilisant les mêmes noms de variables que ceux qui sont quantifiés.

a.

$$\begin{array}{c}
\frac{\forall x.P(x) \vdash \forall x.P(x)}{\forall x.P(x) \vdash P(y)} ax \\
\frac{\forall x.P(x) \vdash P(y)}{\forall x.P(x) \vdash \exists x.P(x)} \vee_e \\
\frac{\forall x.P(x) \vdash \exists x.P(x)}{\vdash (\forall x.P(x)) \rightarrow (\exists x.P(x))} \rightarrow_i
\end{array}$$

b.

$$\begin{array}{c}
\frac{\frac{\forall y.P(x_0, y), \exists x.\forall y.P(x, y) \vdash \forall y.P(x_0, y)}{\forall y.P(x_0, y), \exists x.\forall y.P(x, y) \vdash P(x_0, y)} ax \quad \frac{\forall y.P(x_0, y), \exists x.\forall y.P(x, y) \vdash P(x_0, y)}{\forall y.P(x_0, y), \exists x.\forall y.P(x, y) \vdash \exists x.P(x, y_0)} \vee_e \quad \frac{\forall y.P(x_0, y), \exists x.\forall y.P(x, y) \vdash \exists x.P(x, y_0)}{\exists x.\forall y.P(x, y) \vdash \exists x.P(x, y_0)} \exists_i}{\frac{\exists x.\forall y.P(x, y) \vdash \exists x.P(x, y_0)}{\exists x.\forall y.P(x, y) \vdash \forall y.\exists x.P(x, y)} \vee_i \quad \frac{\exists x.\forall y.P(x, y) \vdash \forall y.\exists x.P(x, y)}{\vdash (\exists x.\forall y.P(x, y)) \rightarrow (\forall y.\exists x.P(x, y))} \rightarrow_i
\end{array}$$

****C.****

$$\begin{array}{c}
\frac{Q(a), P(a) \vdash P(a)}{Q(a), P(a) \vdash P(a) \wedge Q(a)} \text{ax} \quad \frac{Q(a), P(a) \vdash Q(a)}{Q(a), P(a) \vdash P(a) \wedge Q(a)} \text{ax} \\
\frac{Q(a), P(a) \vdash P(a) \wedge Q(a)}{Q(a), P(a) \vdash \exists x.P(x) \wedge Q(x)} \exists_i \\
\frac{Q(a), P(a) \vdash \exists x.P(x) \wedge Q(x)}{P(a) \vdash Q(a) \rightarrow (\exists x.P(x) \wedge Q(x))} \rightarrow_i \\
\frac{P(a) \vdash Q(a) \rightarrow (\exists x.P(x) \wedge Q(x))}{\vdash P(a) \rightarrow (Q(a) \rightarrow (\exists x.P(x) \wedge Q(x)))} \rightarrow_i
\end{array}$$

****d.****

$$\begin{array}{c}
\frac{\forall x.P(x) \rightarrow Q(x) \vdash \forall x.P(x) \rightarrow Q(x)}{\forall x.P(x) \rightarrow Q(x) \vdash P(a) \rightarrow Q(a)} \text{ax} \\
\frac{\forall x.P(x) \rightarrow Q(x) \vdash P(a) \rightarrow Q(a)}{\vdash (\forall x.P(x) \rightarrow Q(x)) \rightarrow (P(a) \rightarrow Q(a))} \rightarrow_i
\end{array}$$

****e.****

$$\begin{array}{c}
\frac{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash \forall x.Q(x) \rightarrow R(x)}{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash Q(x0) \rightarrow R(x0)} \text{ax} \\
\frac{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash Q(x0) \rightarrow R(x0)}{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash Q(x0)} \forall_e \\
\frac{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash Q(x0)}{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash R(x0)} \rightarrow_e \\
\frac{P(x0), \forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash R(x0)}{\forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash P(x0) \rightarrow R(x0)} \rightarrow_i \\
\frac{\forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash P(x0) \rightarrow R(x0)}{\forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash \forall x.P(x) \rightarrow R(x)} \forall_i \\
\frac{\forall x.Q(x) \rightarrow R(x), \forall x.P(x) \rightarrow Q(x) \vdash \forall x.P(x) \rightarrow R(x)}{\vdash (\forall x.P(x) \rightarrow Q(x)) \rightarrow ((\forall x.Q(x) \rightarrow R(x)) \rightarrow (\forall x.P(x) \rightarrow R(x)))} \rightarrow_i
\end{array}$$

Exercice

On parle de méta-règle

$$\frac{P_1 \quad \dots \quad P_n}{C}$$

Quand étant donné des preuves de $P_1, \dots, P_n$, on peut toujours appliquer des règles pour démontrer C .

C'est une sorte de sucre syntaxique qui permet des raccourcis de raisonnement.

Démontrez les méta-règles hypothèses du formulaire.

Réponse Hypothèse $\wedge$:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

Figure 17.18 — Méta-règle hypothèse $\wedge$.

$$\begin{array}{c}
\frac{\Gamma, A, B \vdash C}{A, B, \Gamma, A \wedge B \vdash C} \text{struct} \\
\frac{B, \Gamma, A \wedge B \vdash A \rightarrow C}{\Gamma, A \wedge B \vdash B \rightarrow (A \rightarrow C)} \rightarrow_i \\
\frac{\Gamma, A \wedge B \vdash B \rightarrow (A \rightarrow C)}{\Gamma, A \wedge B \vdash A \rightarrow C} \rightarrow_i \\
\frac{\Gamma, A \wedge B \vdash A \rightarrow C}{\Gamma, A \wedge B \vdash C} \rightarrow_e \\
\frac{\Gamma, A \wedge B \vdash A \wedge B}{\Gamma, A \wedge B \vdash B} \text{ax} \\
\frac{\Gamma, A \wedge B \vdash B}{\Gamma, A \wedge B \vdash A \wedge B} \wedge_{e,d} \\
\frac{\Gamma, A \wedge B \vdash A \wedge B}{\Gamma, A \wedge B \vdash A} \text{ax} \\
\frac{\Gamma, A \wedge B \vdash A}{\Gamma, A \wedge B \vdash C} \wedge_{e,g}
\end{array}$$

Figure 17.19 — Justification de la méta-règle hypothèse $\wedge$ par les règles usuelles.

La règle **struct** utilisée ici est l'affaiblissement : si $\Gamma \vdash C$ est prouvable, alors $\Gamma, D \vdash C$ l'est aussi. Ce n'est pas une règle du système, mais elle se démontre par induction sur la preuve (en ajoutant D à tous les contextes, quitte à renommer les variables liées par $\forall_i$ ou $\exists_e$ pour respecter les conditions sur les variables libres).

Introduction au lambda-calcul

DANS CE CHAPITRE

18.1 Lambda-calcul non typé

18.2 Typage

18.3 Conclusion : la correspondance de Curry-Howard

18.1 Lambda-calcul non typé

On va présenter ici le λ -calcul en utilisant au maximum la syntaxe d'ocaml.

On considère un ensemble dénombrable $\mathcal{V}$ dont les éléments sont appelés des variables. On va appeler les programmes des termes, et l'ensemble $\mathcal{T}$ des termes est défini inductivement par

$$\mathcal{T} = \mathcal{V} \mid \text{fun } \mathcal{V} \rightarrow \mathcal{T} \mid \mathcal{T} \mathcal{T}$$

- Un terme $u \ v$ est appelé une **application** : c'est l'application de la fonction u à l'argument v .
- Un terme $\text{fun } x \rightarrow t$ est appelé une **abstraction** : cela correspond au fait de considérer que la variable x apparaissant (ou non) dans t est maintenant un paramètre.

Remarque 18.1 Comme on l'a vu précédemment, on définit ainsi un langage d'arbres qui correspond à l'arbre syntaxique des termes. Une notation textuelle va nécessairement être imprécise et on est obligé de mettre des parenthèses pour distinguer $u \ (v \ w)$ de $(u \ v) \ w$. Ces parenthèses ne font pas partie de la syntaxe.

Exemple 18.2

Voici des exemples de termes :

- x
- $\text{fun } y \rightarrow x \ y$
- $(\text{fun } x \rightarrow x) \ x$
- $\text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } x \rightarrow f \ (g \ x)$

18.1.1 Variables libres et liées

En fait, $\text{fun } x \rightarrow$ est de même nature que $\forall x.$, c'est un lieu qui va capturer le nom x pour la suite du terme.

Une variable qui n'est pas capturée par un fun est dite **libre**.

Si t est un terme, on note $FV(t)$ ses variables libres, pour **free variables** et en évitant l'ambiguïté de VL avec les variables liées.

On peut définir simplement FV ainsi :

- $\forall x \in \mathcal{V}, FV(x) = \{x\}$

- $\forall u, v \in \mathcal{T}, FV(uv) = FV(u) \cup FV(v)$
- $\forall x \in \mathcal{V}, \forall t \in \mathcal{T}, FV(\text{fun } x \rightarrow t) = FV(t) \setminus \{x\}$

On dit parfois que $\text{fun } x$ capture ou masque x .

18.1.2 Substitution

Si $u, t \in \mathcal{T}$ et $x \in \mathcal{V}$, on peut définir le terme où on a remplacé chaque occurrence libre de x dans u par le terme t .

Par exemple, dans $(\text{fun } x \rightarrow x) x$ si on remplace x par $y z$, le terme devient $(\text{fun } x \rightarrow x) (y z)$ vu que l'occurrence sous l'abstraction est liée.

On note $u[x \setminus t]$ le terme substitué. On peut le définir par induction ainsi :

- $\forall y \in \mathcal{V}, y[x \setminus t] = \begin{cases} t & \text{si } x = y \\ y & \text{sinon.} \end{cases}$
- $\forall u, v \in \mathcal{T}, (uv)[x \setminus t] = u[x \setminus t] v[x \setminus t]$
- $\forall y \in \mathcal{V}, \forall s \in \mathcal{T}, (\text{fun } y \rightarrow s)[x \setminus t] = \text{fun } y \rightarrow \begin{cases} s & \text{si } x = y \\ s[x \setminus t] & \text{si } x \neq y \text{ et } y \notin FV(t) \end{cases}$
- si $x \neq y$ et $y \in FV(t)$, on renomme d'abord la variable liée : $(\text{fun } y \rightarrow s)[x \setminus t] = \text{fun } z \rightarrow (s[y \setminus z])[x \setminus t]$ où z est une variable *fraîche*, c'est-à-dire n'apparaissant ni dans s ni dans t , et distincte de x .

Le dernier cas évite la **capture** des variables libres de t . Par exemple, $(\text{fun } y \rightarrow x)[x \setminus y]$ ne doit pas donner $\text{fun } y \rightarrow y$ (la fonction identité) mais $\text{fun } z \rightarrow y$ (une fonction constante) : la variable libre y doit rester libre.

Ce renommage repose sur l' α -**conversion** : deux termes qui ne diffèrent que par le nom de leurs variables liées, comme $\text{fun } x \rightarrow x$ et $\text{fun } y \rightarrow y$, sont dits α -équivalents. Formellement, $\text{fun } y \rightarrow s$ est α -équivalent à $\text{fun } z \rightarrow s[y \setminus z]$ dès que z n'apparaît pas dans s , et on étend cela à toute profondeur. Dans la suite, on identifie les termes α -équivalents.

18.1.3 β -réduction

Un terme de la forme $(\text{fun } x \rightarrow u) t$ est appelé un β -redex. Cela correspond à une expression qui va s'évaluer en `OCaml` : on s'attend à ce que l'expression devienne u dans lequel x est remplacé par t . Cela correspond exactement à la notion précédente de substitution et on définit une relation ainsi :

$$(\text{fun } x \rightarrow u) t \xrightarrow{\beta} u[x \setminus t]$$

On peut naturellement prolonger $\xrightarrow{\beta}$ par congruence en faisant en sorte que cette réduction s'applique à toute profondeur dans l'arbre syntaxique du terme :

- $\forall u, t, t' \in \mathcal{T}, t \xrightarrow{\beta} t' \Rightarrow \begin{cases} u t \xrightarrow{\beta} u t' & \text{et} \\ t u \xrightarrow{\beta} t' u \end{cases}$
- $\forall x \in \mathcal{V}, \forall t, t' \in \mathcal{T}, t \xrightarrow{\beta} t' \Rightarrow \text{fun } x \rightarrow t \xrightarrow{\beta} \text{fun } x \rightarrow t'$

Exemple 18.3

- Si $t = (\text{fun } x \rightarrow x) y$, ici c'est directement un β -redex, donc $t \xrightarrow{\beta} y$.
- Si $t = ((\text{fun } f \rightarrow \text{fun } x \rightarrow f (f x)) (\text{fun } y \rightarrow y)) z$ alors on peut réduire le redex sous l'application : $t \xrightarrow{\beta} t' = (\text{fun } x \rightarrow (\text{fun } y \rightarrow y) ((\text{fun } y \rightarrow y) x)) z$.

Ici, on a trois β -redex : un pour chaque $\text{fun } y \rightarrow y$ et un pour $\text{fun } x \rightarrow$.

Choisissons de réduire celui qui est le plus en profondeur dans le terme : $t' \xrightarrow{\beta} (\text{fun } x \rightarrow (\text{fun } y \rightarrow y) x) z$, puis en réduisant à l'intérieur, $\xrightarrow{\beta} (\text{fun } x \rightarrow x) z \xrightarrow{\beta} z$.

Au final, on a obtenu $t \xrightarrow{\beta} \xrightarrow{\beta} \xrightarrow{\beta} z$.

- Si $\Omega = (\text{fun } x \rightarrow x x) (\text{fun } x \rightarrow x x)$ alors on peut poser $\delta = \text{fun } x \rightarrow x x$ et on a alors $\Omega = \delta \delta = (\text{fun } x \rightarrow x x) \delta \xrightarrow{\beta} \delta \delta = \Omega = (\text{fun } x \rightarrow x x) (\text{fun } x \rightarrow x x)$

Ainsi $\Omega \xrightarrow{\beta} \Omega$: Ω est un point fixe de la réduction.

Ces exemples permettent d'identifier deux points importants :

- la β -réduction est un petit pas de réduction, mais on souhaite pouvoir en réaliser tant que c'est possible. Le dernier exemple montre qu'il ne suffit pas de réduire en espérant tomber sur un terme sans β -redex : le processus n'a aucune raison de terminer.

On note $\xrightarrow{\beta}$ la fermeture réflexive et transitive de $\rightarrow_{\beta}$, c'est-à-dire la plus petite relation réflexive et transitive contenant $\rightarrow_{\beta}$. On parle de réduction à grand pas, cela revient à enchaîner un nombre quelconque, éventuellement nul, de réductions. Ainsi, dans le second exemple, on a montré $t \xrightarrow{\beta} z$.

- il est possible d'avoir le choix dans les β -redex que l'on peut réduire. On peut se poser la question légitime de l'ordre et de son importance. Le théorème suivant montre qu'il n'y a pas de problèmes.

Théorème 18.4

Church-Rosser Soient $t, t_1, t_2 \in \mathcal{T}$ tels que $t \xrightarrow{\beta} t_1$ et $t \xrightarrow{\beta} t_2$.

Il existe $s \in \mathcal{T}$ tel que $t_1 \xrightarrow{\beta} s$ et $t_2 \xrightarrow{\beta} s$.

Note on identifie ici les termes à α -équivalence près, comme $\text{fun } x \rightarrow x = \text{fun } y \rightarrow y$.

Ce théorème est admis.

On note ce théorème sous la forme d'un diagramme :

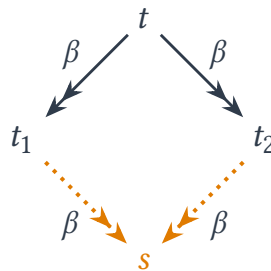


Figure 18.1 — Confluence : si t se réduit en t_1 et en t_2 , il existe s commun (en pointillé).

Définition 18.5

Si $t \xrightarrow{\beta} t'$ et que t' ne contient pas de β -redex, on dit que t' est une forme normale de t .

Conséquence du théorème précédent :

Corollaire 18.6

Si un terme a une forme normale, alors elle est unique.

Preuve Si t_1 et t_2 sont deux formes normales de t , le théorème de Church-Rosser fournit s tel que $t_1 \xrightarrow{\beta} s$ et $t_2 \xrightarrow{\beta} s$. Comme t_1 ne contient aucun redex, la seule réduction possible depuis t_1 est la suite vide, donc $s = t_1$. De même $s = t_2$, et ainsi $t_1 = t_2$.

□

Remarque 18.7 Le terme $t = (\text{fun } x \rightarrow y) \Omega$ a une forme normale y en réduisant le redex le plus à gauche, mais on peut également réduire Ω et avoir $t \xrightarrow{\beta} t$.

Ainsi, un terme peut avoir une forme normale et ne pas y aboutir en réduisant de manière quelconque.

18.2 Typage

18.2.1 Types simples

Pour résoudre le problème des termes comme Ω dont la réduction ne termine pas, on va introduire une restriction sur les termes : une discipline de typage.

On note ici V un ensemble dénombrable de variables de type et on définit l'ensemble T des types inductivement ainsi :

$$T = V \mid T \rightarrow T$$

Remarque 18.8 Il s'agit des formules du calcul propositionnel avec uniquement le connecteur $\rightarrow$.

On note $t : \tau$ pour affirmer que le terme $t \in \mathcal{T}$ est du type $\tau \in T$. On va considérer des jugements de la forme $\Gamma \vdash t : \tau$ où Γ , appelé *contexte*, est un ensemble d'hypothèses de la forme $x : \tau'$ avec $x \in \mathcal{V}$, chaque variable apparaissant au plus une fois dans Γ .

On définit alors les règles d'inférence suivantes :

$$\frac{}{\Gamma, x : A \vdash x : A} \text{ var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \text{fun } x \rightarrow t : A \rightarrow B} \text{ abs} \qquad \frac{\Gamma \vdash u : A \rightarrow B \quad \Gamma \vdash v : A}{\Gamma \vdash uv : B} \text{ app}$$

Figure 18.2 – Règles de typage du λ -calcul simplement typé : variable, abstraction, application.

Remarque 18.9 Ce sont exactement les règles de la déduction naturelle αx , $\rightarrow_i$ et $\rightarrow_e$.

Définition 18.10

Un terme t a le type τ s'il existe un contexte Γ tel que $\Gamma \vdash t : \tau$ est dérivable.

Un terme t est **bien typé**, s'il existe τ tel que t a le type τ .

On dit qu'un type τ est habité s'il existe au moins un terme t tel que $\vdash t : \tau$ soit dérivable dans le contexte vide (sinon, tout type serait habité par une variable : $x : \tau \vdash x : \tau$).

Dans la suite, on dira $t : \tau$ pour affirmer que t a le type τ .

Remarque 18.11 On peut se poser la question du contexte dans la définition précédente : il est là pour traiter des variables libres.

Le théorème suivant affirme que le typage est compatible avec la réduction, on parle de correction, ou *soundness*, du typage.

Théorème 18.12

Si $\Gamma \vdash t : \tau$ est dérivable et $t \xrightarrow{\beta} t'$, alors $\Gamma \vdash t' : \tau$ est dérivable.

Ce théorème est admis.

Le théorème suivant affirme que l'objectif est bien rempli : la réduction des termes bien typés termine toujours.

Théorème 18.13

Tait 1967 Si t est bien typé, alors il a une forme normale.

De plus, ici, toute suite de réductions $\xrightarrow{\beta}$ partant de t est finie, et sa longueur est majorée par une constante ne dépendant que de t : c'est impossible de boucler en réduisant. On dit qu'il y a normalisation forte. *Ce théorème est admis.*

Remarque 18.14 Même si les types sont simples et que cela correspond à la déduction naturelle, il a fallu attendre 35 ans entre la définition du λ -calcul et ce théorème.

Exemple 18.15

Voici un exemple de preuve du fait que $((\text{fun } f \rightarrow \text{fun } x \rightarrow f \ x) (\text{fun } x \rightarrow x)) \ y$ est bien typé :

$$\begin{array}{c}
 \frac{}{f : A \rightarrow A \vdash f : A \rightarrow A} \text{var} \quad \frac{}{x : A \vdash x : A} \text{var} \\
 \frac{}{f : A \rightarrow A, x : A \vdash f \ x : A} \text{app} \\
 \frac{}{f : A \rightarrow A \vdash \text{fun } x \rightarrow f \ x : A \rightarrow A} \text{abs} \\
 \frac{}{\vdash \text{fun } f \rightarrow \text{fun } x \rightarrow f \ x : (A \rightarrow A) \rightarrow A \rightarrow A} \text{abs} \quad \frac{}{x : A \vdash x : A} \text{var} \\
 \frac{}{\vdash \text{fun } x \rightarrow x : A \rightarrow A} \text{abs} \quad \frac{}{y : A \vdash y : A} \text{var} \\
 \frac{}{\vdash (\text{fun } f \rightarrow \text{fun } x \rightarrow f \ x) (\text{fun } x \rightarrow x) : A \rightarrow A} \text{app} \quad \frac{}{y : A \vdash y : A} \text{var} \\
 \frac{}{y : A \vdash ((\text{fun } f \rightarrow \text{fun } x \rightarrow f \ x) (\text{fun } x \rightarrow x)) \ y : A} \text{app}
 \end{array}$$

Figure 18.3 — Dérivation de typage de $((\text{fun } f \rightarrow \text{fun } x \rightarrow f \ x) (\text{fun } x \rightarrow x)) \ y$.

On a retiré des contextes les hypothèses inutiles pour alléger la preuve.

Si on représente le terme sous la forme d'un arbre, on constate que cet arbre a exactement la même structure que la preuve.

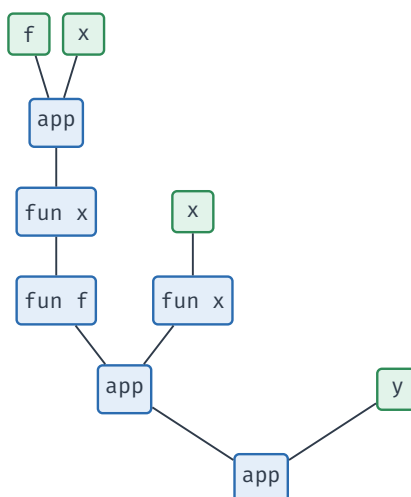


Figure 18.4 — Arbre du terme : même structure que la dérivation de typage (lue de bas en haut).

Le terme est en quelque sorte une représentation à plat de la preuve.

Une fois le terme connu, il n'est pas possible de changer les règles à appliquer. Elles sont déjà présentes dans le terme.

Par contre, on peut ****oublier**** les termes dans l'arbre de typage et ne conserver que les formules. On obtient alors une preuve valide en déduction naturelle.

$$\begin{array}{c}
 \frac{}{A, A \rightarrow A, A \vdash A \rightarrow A} \text{ax} \quad \frac{}{A, A \rightarrow A, A \vdash A} \text{ax} \\
 \frac{}{A, A \rightarrow A, A \vdash A} \rightarrow_e \\
 \frac{}{A, A \rightarrow A \vdash A \rightarrow A} \rightarrow_i \\
 \frac{}{A \vdash (A \rightarrow A) \rightarrow A \rightarrow A} \rightarrow_i \\
 \frac{}{A \vdash A \rightarrow A} \text{ax} \quad \frac{}{A \vdash A \rightarrow A} \rightarrow_e \\
 \frac{}{A \vdash A} \text{ax}
 \end{array}$$

Figure 18.5 — La même dérivation sans les termes : une preuve en déduction naturelle.

Bien entendu, cette preuve semble très étrange vu qu'une simple règle axiome permet de conclure. Mais on pourrait également remarquer que le terme admet `y`

comme forme normale et que la preuve du typage $\lambda y. \lambda x. A$ ne sera également qu'une seule règle λ .

Comme on vient de le voir, vérifier un typage ne demande aucune *invention* : le choix des règles à appliquer ainsi que les formules utilisées pour les éliminations sont déjà connus. La vérification est purement mécanique par induction.

La question qui se posera est plutôt de savoir **inventer** un bon type pour un terme typable donné. C'est la question de l'inférence de type.

18.2.2 Enrichir le langage

On va enrichir le langage avec de nouvelles constructions.

Les termes sont maintenant définis par :

$$\begin{aligned} \mathcal{T} = & \mathcal{V} \mid \text{fun } \mathcal{V} \rightarrow \mathcal{T} \mid \mathcal{T} \mathcal{T} \\ & \mid (\mathcal{T}, \mathcal{T}) \mid \text{fst } \mathcal{T} \mid \text{snd } \mathcal{T} \\ & \mid \text{Right } \mathcal{T} \mid \text{Left } \mathcal{T} \\ & \mid \text{match } \mathcal{T} \text{ with Left } \mathcal{V} \rightarrow \mathcal{T} \mid \text{Right } \mathcal{V} \rightarrow \mathcal{T} \end{aligned}$$

Et les types par :

$$T = V \mid T \rightarrow T \mid T \wedge T \mid T \vee T$$

On retrouve ainsi les connecteurs $\wedge$ et $\vee$ du calcul propositionnel.

On considère également quatre nouveaux types de redex pour la réduction. C'est un peu abusif de parler de β -réduction, donc on utilisera uniquement $\rightarrow$ ici et on parlera juste de réduction.

- $\text{fst } (t, t') \rightarrow t$
- $\text{snd } (t, t') \rightarrow t'$
- $\text{match Left } t \text{ with Left } x \rightarrow u \mid \text{Right } y \rightarrow v \rightarrow u[x \backslash t]$
- $\text{match Right } t \text{ with Left } x \rightarrow u \mid \text{Right } y \rightarrow v \rightarrow v[y \backslash t]$

Ces règles correspondent à l'exécution en `OCaml` en rajoutant la définition de type :

```
type ('a, 'b) type_or = Left of 'a | Right of 'b
```

Remarque 18.16 Il peut sembler que ce que l'on propose ici est un enrichissement de la syntaxe, mais en fait, c'est surtout le système de type qu'on enrichit.

En effet, on pourrait supposer l'existence de termes fonctionnels réalisant les éléments de syntaxe. Ce qui compte, c'est surtout de pouvoir les typer.

L'exemple caractéristique, ce sont les couples. Une fois acquise une construction $A \wedge B$ (ou $A * B$ en `OCaml`) pour le type d'un couple, on pourrait admettre l'existence de trois fonctions polymorphes :

- $\text{make_pair} : A \rightarrow B \rightarrow A * B$
- $\text{fst} : A * B \rightarrow A$
- $\text{snd} : A * B \rightarrow B$

Reste à demander à ce que $\text{fst } (\text{make_pair } x \ y) = x$ pour tout x, y , et la relation symétrique pour snd .

On rajoute également des nouvelles règles de typage :

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash t' : B}{\Gamma \vdash (t, t') : A \wedge B} \text{ couple} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \text{fst } t : A} \text{ fst} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \text{snd } t : B} \text{ snd}$$

Figure 18.6 — Règles de typage des couples (conjonction).

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \text{Left } t : A \vee B} \text{Left} \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \text{Right } t : A \vee B} \text{Right}$$

Figure 18.7 — Règles de typage des constructeurs Left et Right (disjonction).

$$\frac{\Gamma \vdash t : A \vee B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \text{match } t \text{ with Left } x \rightarrow u \mid \text{Right } y \rightarrow v : C} \text{match}$$

Figure 18.8 — Règle de typage du filtrage (élimination de la disjonction).

On retrouve exactement les règles $\wedge_i$, $\wedge_{e,g}$, $\wedge_{e,d}$, $\vee_{i,g}$, $\vee_{i,d}$ et $\vee_e$ de la déduction naturelle.

Exemple 18.17

On considère le terme

```
fun p -> match fst p with Left a -> Left (a, snd p) | Right b -> Right (b, snd p)
```

qui prend un couple dont la première composante est un Left ou un Right et fait « rentrer » la seconde composante dans le Left ou le Right.

On note t le corps de la fonction (le match), Γ le contexte $p : (A \vee B) \wedge C$ et $D = (A \wedge C) \vee (B \wedge C)$. Voici une dérivation de typage de ce terme :

$$\frac{\frac{\Gamma \vdash p : (A \vee B) \wedge C}{\Gamma \vdash \text{fst } p : A \vee B} \text{fst} \quad \frac{\frac{\Gamma, a : A \vdash a : A}{\Gamma, a : A \vdash (a, \text{snd } p) : A \wedge C} \text{var} \quad \frac{\Gamma, a : A \vdash p : (A \vee B) \wedge C}{\Gamma, a : A \vdash \text{snd } p : C} \text{snd}}{\Gamma, a : A \vdash \text{Left } (a, \text{snd } p) : D} \text{couple} \quad \frac{\frac{\Gamma, b : B \vdash b : B}{\Gamma, b : B \vdash (b, \text{snd } p) : B \wedge C} \text{var} \quad \frac{\Gamma, b : B \vdash p : (A \vee B) \wedge C}{\Gamma, b : B \vdash \text{snd } p : C} \text{snd}}{\Gamma, b : B \vdash \text{Right } (b, \text{snd } p) : D} \text{couple} \quad \frac{\Gamma \vdash t : D}{\vdash \text{fun } p \rightarrow t : (A \vee B) \wedge C \rightarrow D} \text{abs}$$

Figure 18.9 — Dérivation de typage du terme associé à la distributivité ; t est le match.

En oubliant les termes, on obtient exactement la preuve de $\$(A \vee B) \wedge C \vdash (A \wedge C) \vee (B \wedge C)$ donnée en exemple dans le chapitre [Dédution naturelle](#chap-logique-dedunat), suivie d'une règle $\rightarrow_i$. Chaque règle de la preuve se lit dans le terme : le `match` est la disjonction de cas $\vee_e$, `fst` et `snd` sont les éliminations $\wedge_{e,g}$ et $\wedge_{e,d}$, le couple est $\wedge_i$, `Left` et `Right` sont $\vee_{i,g}$ et $\vee_{i,d}$.$

En OCaml, avec le type `type_or` précédent, cette fonction est acceptée et le type inféré est bien la formule attendue :

```
``ocaml
let distrib = fun p ->
  match fst p with
  | Left a -> Left (a, snd p)
  | Right b -> Right (b, snd p)
(* val distrib : ('a, 'b) type_or * 'c -> ('a * 'c, 'b * 'c) type_or *)
``
```

18.3 Conclusion : la correspondance de Curry-Howard

Ce qu'on a observé tout au long de ce chapitre porte le nom de **correspondance de Curry-Howard** (ou correspondance preuves-programmes) : on peut lire chaque terme bien typé comme une preuve, et chaque type comme la formule qu'il prouve.

Logique (déduction naturelle)	Programmation (λ -calcul typé)
formule	type
preuve de $\Gamma \vdash A$	terme t tel que $\Gamma \vdash t : A$
hypothèse, règle <i>ax</i>	variable, règle <i>var</i>
$\frac{\vec{p}}{i} / \frac{\vec{e}}{e}$	<i>fun</i> $x \rightarrow t$ / application
$\wedge_i / \wedge_{e,g}, \wedge_{e,d}$	couple / <i>fst</i> , <i>snd</i>
$\vee_{i,g}, \vee_{i,d} / \vee_e$	<i>Left</i> , <i>Right</i> / <i>match</i>
formule prouvable	type habité
simplification d'une preuve	réduction d'un terme

Le dernier point mérite une explication. Un redex, comme *fst* (t, t') , correspond dans la preuve à une introduction immédiatement suivie d'une élimination du même connecteur : on prouve $A \wedge B$ pour en déduire aussitôt A , ce qui est un détour inutile puisqu'on avait déjà une preuve de A . Réduire le terme, c'est supprimer ce détour dans la preuve. Le théorème de correction du typage affirme que la preuve obtenue prouve toujours la même formule, et le théorème de Tait affirme qu'on finit toujours par obtenir une preuve sans détour.

Remarque 18.18 La correspondance permet de transporter des résultats d'un monde à l'autre. Par exemple, la formule de Peirce $((A \rightarrow B) \rightarrow A) \rightarrow A$ est une tautologie, mais elle n'est pas prouvable en déduction naturelle intuitionniste (on peut le voir avec la [sémantique de Gödel](#)). Comme une dérivation de typage donne une preuve en oubliant les termes, le type $((\lambda a. \lambda b. \lambda a. a) \rightarrow \lambda a. a)$ n'est habité par aucun terme de notre λ -calcul. On ne peut donc pas écrire de fonction `OCaml` de ce type avec les seules constructions de ce chapitre : il faudrait des traits impératifs comme les exceptions, ou une fonction qui ne termine pas.

Remarque 18.19 On n'a pas traité ici $\perp$, $\neg$, ni les quantificateurs. On peut prolonger la correspondance : $\perp$ correspond à un type vide, et les quantificateurs à des types dépendants. C'est sur ce principe que reposent les assistants de preuve comme `Rocq` (anciennement `Coq`) ou `Lean`, dans lesquels écrire une preuve revient à écrire un programme bien typé.

PARTIE IV

Calculabilité

19	Calculabilité	207
20	Exercices Calculabilité	213

Calculabilité

DANS CE CHAPITRE

19.1 Décidabilité

19.2 Complexité et problème de décision

19.3 NP-complétude

19.1 Décidabilité

19.1.1 Représentation des entrées

Tout ce qu'on construit au sein d'un ordinateur appartient à un ensemble dénombrable. En effet, les programmes sont des mots sur un alphabet fini, ils forment donc un ensemble dénombrable. Les exécutions finies de ces programmes, comprenant alors l'état de la mémoire accédée et modifiée par le programme, forment également un ensemble dénombrable.

Les entrées et les sorties d'un programme sont des valeurs représentables en mémoire, elles forment donc également un ensemble dénombrable.

Cela signifie qu'on peut associer à tout programme et toute donnée, un numéro ou une représentation textuelle unique pour l'identifier.

Dans la suite, on va manipuler des fonctions ou des programmes prenant en entrée d'autres programmes ou données, on pourra considérer ces représentations sous la forme d'une chaîne de caractère.

Remarque 19.1 On pourrait considérer des entiers, mais en C comme en OCaml, les entiers n'étant pas de précision arbitraire, ils ne peuvent que représenter un ensemble **fini** de valeurs.

Exemple 19.2

On a manipulé l'interprète OCaml, c'est un programme dans lequel on peut taper un texte correspondant à un programme OCaml et qui va répondre par un texte représentant l'évaluation de ce programme. Plus généralement, les compilateurs et les interprètes voient les programmes comme une forme de donnée.

Dans la suite, on identifiera un ensemble à ses représentations pour simplifier les énoncés.

19.1.2 Fonction calculable

Définition 19.3

Soient E, F deux ensembles dénombrables, on dit que $f : E \rightarrow F$ est **calculable** s'il existe un programme prenant en entrée $e \in E$ et renvoyant **en temps fini**, mais en pouvant accéder à une mémoire infinie, $f(e)$.

Théorème 19.4

L'ensemble des fonctions calculables de $\mathbb{N}$ dans $\mathbb{N}$ est dénombrable.



Preuve Pour chaque fonction calculable f , on choisit un programme P_f qui la calcule. Deux fonctions distinctes ne peuvent pas être calculées par le même programme, donc $f \mapsto P_f$ est une injection de l'ensemble des fonctions calculables dans l'ensemble des programmes, qui est dénombrable. Cet ensemble est de plus infini car il contient les fonctions constantes, qui sont calculables. Il est donc dénombrable. $\square$

Corollaire 19.5

Il existe une infinité non dénombrable de fonctions de $\mathbb{N}$ dans $\mathbb{N}$ non calculables.



Preuve L'ensemble $\mathbb{N}^{\mathbb{N}}$ des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas dénombrable (argument diagonal de Cantor) : si $(f_n)_{n \in \mathbb{N}}$ est une suite de fonctions de $\mathbb{N}$ dans $\mathbb{N}$, la fonction $g : n \mapsto f_n(n) + 1$ diffère de chaque f_n en n , donc aucune énumération ne contient toutes les fonctions.

Si l'ensemble des fonctions non calculables était au plus dénombrable, $\mathbb{N}^{\mathbb{N}}$ serait la réunion de deux ensembles au plus dénombrables, donc serait dénombrable. Absurde. $\square$

19.1.3 Machine universelle**Théorème 19.6**

On considère la fonction qui étant donné un programme P , et une entrée e pour P pour laquelle P renvoie une valeur en temps fini, renvoie le résultat de P sur e .

Cette fonction est calculable (admis).

On parle alors de programme universel ou de machine universelle dans ce contexte. Dans le contexte de la programmation, on parle plus simplement d'**interprète**.

On étend naturellement ce programme universel sur des entrées pour lesquelles P ne termine pas en le faisant lui-même ne pas terminer.

Notons qu'il n'est pas possible que ce programme puisse détecter et renvoyer une valeur spécifique dans les cas où P ne termine pas. C'est en effet l'objet du paragraphe suivant.

19.1.4 Problème de l'arrêt**Définition 19.7**

On appelle **problème de décision** sur E dénombrable la donnée d'une fonction $f : E \rightarrow \mathbb{B}$ où $\mathbb{B} = \{\top, \perp\}$ est l'ensemble des booléens. Un élément de E est alors appelé une **instance** du problème.

On dit alors que le problème consiste à décider la propriété f .

On cherche alors à savoir si le problème est **décidable**, c'est-à-dire, à savoir si f est calculable.

Remarque 19.8 On peut identifier le problème p à la donnée de E et de $f^{-1}(\{\top\}) \subset E$, l'ensemble de ses instances **positives**. Ainsi, on pourra noter $e \in p \Leftrightarrow e \in f^{-1}(\{\top\}) \Leftrightarrow f(e) = \top$.

Comme on ne considère que des entrées dénombrables, on peut considérer que $E = \Sigma^*$ pour Σ un alphabet. On peut alors identifier un problème de décision à la fonction indicatrice d'un langage. On parle ainsi assez souvent de langages à la place de parler de problèmes de décision.

Définition 19.9

On appelle **problème de l'arrêt** le problème qui étant donné un couple (P, e) où P est un programme et e une entrée pour celui-ci, décide si P termine en temps fini sur l'entrée e .

Théorème 19.10

Le problème de l'arrêt est indécidable.

Preuve Par l'absurde, supposons qu'on dispose d'une fonction

```
val arret : string -> string -> bool
```

telle que `arret p e` renvoie `true` si et seulement si le programme représenté par p termine sur l'entrée représentée par e .

On définit alors

```
let bogus p =
  if arret p p
  then let rec f () = f () in
       f ()
  else 42
```

Si on note `sbogus` une représentation textuelle du programme complet contenant `arret` et `bogus`, on peut alors demander à évaluer `bogus sbogus`.

Deux cas :

- `bogus sbogus` termine, on a alors `arret sbogus sbogus` qui doit renvoyer `true` par définition. Mais alors, dans `bogus`, on rentre dans la branche `then` qui ne termine pas. Impossible.
- `bogus sbogus` ne termine pas, on a alors `arret sbogus sbogus` qui renvoie `false` et, dans `bogus`, on passe par le `else` et la fonction termine. Impossible.

Les deux cas étant impossibles, on en déduit une contradiction.

Il s'agit d'un argument **diagonal** : on fait agir le programme `bogus` sur sa propre représentation, de la même manière que Cantor évalue f_n en n .

□

Remarque 19.11 Pour l'arrêt, le langage associé au problème est l'ensemble des mots (pr, e) où pr est un programme, e une entrée et pr termine sur e .

Remarque 19.12 Le problème de l'arrêt est cependant **semi-décidable** : il existe un programme qui, sur l'entrée (P, e) , termine en renvoyant $\top$ quand P termine sur e , et ne termine pas sinon. Il suffit en effet d'utiliser la machine universelle pour simuler P sur e , puis de renvoyer $\top$. Ce programme ne permet pas de **décider** l'arrêt car on ne sait jamais, au cours de la simulation, si P va finir par terminer ou non.

19.2 Complexité et problème de décision

Jusqu'ici, on s'est intéressé à la complexité d'un algorithme. Plusieurs algorithmes résolvant un même problème, par exemple le tri de données, pouvant avoir des complexités différentes.

Ici, on va prendre un peu de hauteur et s'intéresser au problème lui-même.

19.2.1 Définition

Définition 19.13

On appelle **taille d'une instance** d'un problème, la taille de sa représentation dans un système de représentation raisonnable, comme la représentation binaire sérialisée de cette donnée.

Remarque 19.14 Ici, on est obligé d'être un peu spécifique. En effet, si on ne précise rien de plus pour un entier $n \in \mathbb{N}$, on pourrait dire qu'il est sa propre représentation, et donc que tout entier est de taille 1, quelle que soit sa grandeur. À l'opposé, on pourrait le représenter en unaire, ce qui obligerait à considérer une taille n , ce qui est inefficace. En passant au binaire, on s'assure une représentation de taille $\lfloor \log_2 n \rfloor + 1$ pour $n \geq 1$. Le choix d'une autre base $b \geq 2$ ne fait que multiplier cette taille par une constante ($\log_b n = \log_2 n / \log_2 b$), et ne change donc pas l'ordre de grandeur.

On a déjà vu qu'on définissait la complexité temporelle en pire cas comme étant la suite $(C_n)_{n \in \mathbb{N}}$ où C_n est le plus grand nombre d'opérations élémentaires sur une entrée de taille n .

Définition 19.15

On dit que le problème de décision p est dans la classe **P** quand il existe un programme décidant p dont la complexité pour des instances de taille n est en $O(f(n))$ avec f fonction polynomiale en n .

Exemple 19.16

On a vu beaucoup de problèmes dans la classe **P** comme l'existence d'un cycle dans un graphe, la détection de palindromes...

Cependant, on sent bien que de nombreux problèmes qu'on a résolus sont polynomiaux, mais comme ce ne sont pas des problèmes de décisions, ils ne sont pas techniquement dans la classe **P**.

19.2.2 Réduction polynomiale

Définition 19.17

Soient p, p' deux problèmes de décision sur les entrées respectives E et E' . Une fonction **calculable**

$$f : E \rightarrow E'$$

est appelée une **réduction** de p à p' lorsque

$$\forall e \in E, e \in p \Leftrightarrow f(e) \in p'$$

Lorsque f est calculable en temps polynomial, on dit que f est une réduction polynomiale et on note $p \leq_P p'$.

Théorème 19.18

Soit f une réduction de p à p' .

- Si p' est décidable, alors p est décidable.
- Si f est une réduction polynomiale et $p' \in \mathbf{P}$, alors $p \in \mathbf{P}$.

Preuve

- Si D' est un programme décidant p' , le programme qui sur l'entrée e calcule $f(e)$ puis renvoie $D'(f(e))$ termine toujours et renvoie $\top$ si et seulement si $f(e) \in p'$, c'est-à-dire si et seulement si $e \in p$.
- Avec le même programme, si f est calculée en temps $O(n^k)$ et D' en temps $O(n^l)$, alors $|f(e)| = O(|e|^k)$ car un programme ne peut écrire qu'un nombre borné de symboles par opération élémentaire. Le temps total est donc en $O(|e|^k + |e|^{kl})$, qui est polynomial.

□

Remarque 19.19 Attention au **sens** de la réduction : par contraposée, si p est indécidable et que p se réduit à p' , alors p' est indécidable. Pour montrer qu'un **nouveau** problème p' est indécidable, on réduit donc un problème **connu** pour être indécidable (par exemple l'arrêt) **vers** p' . Réduire p' vers l'arrêt ne prouve rien : cela montre seulement que p' n'est pas plus difficile que l'arrêt.

On a vu des passages polynomiaux d'un problème à un autre, mais pas dans le cadre restrictif des problèmes de décision. On peut ainsi citer la réduction de 2-SAT au calcul des composantes fortement connexes.

19.3 NP-complétude

19.3.1 Problème NP

Définition 19.20

On dit qu'un problème de décision p sur des entrées E est dans la classe **NP** lorsqu'il existe une fonction polynomiale f sur $\mathbb{N}$ et un problème de décision $v(p)$ de classe **P** sur $E \times B^*$ tel que

$$\forall e \in E, \quad e \in p \Leftrightarrow \exists c \in B^*, |c| \leq f(|e|) \text{ et } (e, c) \in v(p)$$

On dit que $v(p)$ est le problème de vérification associé à p .

Ici, on note $B = \{0, 1\}$ l'ensemble des booléens et B^* l'ensemble des mots binaires. Les éléments de B^* sont des candidats certificats et lorsque $(e, c) \in v(p)$ on dit que le mot binaire c est un certificat pour la validité de l'entrée e .

Il s'agit de preuves permettant de vérifier **rapidement**, à l'aide de son certificat, qu'une instance est positive pour p . Attention, la définition n'est pas symétrique : on ne demande pas de certificat pour les instances négatives.

Remarque 19.21 On utilise ici un mot binaire comme certificat et non un ensemble plus spécifique pour que la définition soit valide. Cependant, toute étape de vérification commence par une désérialisation du certificat, c'est-à-dire par la construction d'informations relatives à l'entrée. Si l'entrée est un graphe, un candidat certificat peut être une coloration de ce graphe à l'aide d'un codage des tableaux finis d'entiers. Seuls quelques mots binaires ont alors un sens compatible avec une entrée donnée.

Théorème 19.22

Fira Sans(*SAT*), le problème qui étant donnée une formule du calcul propositionnel décide si elle est satisfiable, est dans la classe **NP**.

Preuve Un certificat pour une formule φ est une valuation de ses variables, codée par un mot binaire de longueur le nombre de variables, donc au plus $|\varphi|$. Le problème de vérification consiste à évaluer φ pour cette valuation, ce qui se fait en temps polynomial (et même linéaire) en $|\varphi|$ par un parcours de son arbre syntaxique. Enfin φ est satisfiable si et seulement si un tel certificat existe. □

Théorème 19.23

P $\subset$ **NP**

Preuve Il suffit ici de considérer un certificat factice, par exemple le mot vide pour tout e , et le problème de vérification $v(p) = \{(e, c) \mid e \in p\}$, qui est dans **P** car il s'agit juste de décider p en ignorant c . □

Exercice

difficile a-t-on l'inclusion réciproque ?

Note : m'envoyer la preuve pour que je la vérifie avant d'en parler.

19.3.2 Problèmes NP-complets

Définition 19.24

Un problème de décision p' est dit **NP-difficile** lorsque pour tout problème $p \in \text{NP}$, on a $p \leq_p p'$.

Il est dit **NP-complet** lorsqu'il est NP-difficile et dans NP.

Théorème 19.25 — (Cook-Levin)

Fira Sans(SAT) est NP-complet (admis).

Remarque 19.26

- La relation $\leq_p$ est transitive (la composée de deux fonctions calculables en temps polynomial l'est encore, avec le même argument de taille que précédemment). Ainsi, pour montrer qu'un problème $p' \in \text{NP}$ est NP-complet, il suffit de réduire polynomialement un problème **connu** pour être NP-complet, comme Fira Sans(SAT), **vers** p' .
- Si un seul problème NP-complet est dans P, alors, d'après le théorème sur les réductions, $\text{P} = \text{NP}$.

19.3.3 Problème d'optimisation et seuil

On considère ici des problèmes d'optimisation : on considère ainsi des entrées E et des sorties S avec pour tout $e \in E$, un ensemble $V(e) \subset S$. On dispose de plus d'une fonction de coût $f : S \rightarrow \mathbb{R}$ (ou $\mathbb{N}$ ou n'importe quel ensemble totalement ordonné) et on cherche pour chaque $e \in E$ l'élément de $V(e)$ qui maximise (ou minimise) f .

Exemple 19.27

Considérons par exemple le problème **MinVertexCover** :

- Entrée : un graphe non orienté $G = (S, A)$.
- Sortie : un sous-ensemble $V \subset S$ de plus petit cardinal tel que $\forall a \in A, V \cap a \neq \emptyset$.

Avec la présentation précédente, les entrées sont les graphes non orientés, les sorties sont les ensembles de sommets. Pour un graphe $G = (S, A)$, on a donc $V(G) = \{V \subset S \mid \forall a \in A, V \cap a \neq \emptyset\}$, l'ensemble des couvertures par sommets de G . La fonction de coût est alors juste le cardinal, qu'on cherche à minimiser.

À un tel problème d'optimisation (ici de minimisation), on associe un problème de décision à **seuil** : étant donnés $e \in E$ et k , existe-t-il $v \in V(e)$ tel que $f(v) \leq k$? (on remplace par $f(v) \geq k$ pour un problème de maximisation). Pour **MinVertexCover**, on obtient le problème **VertexCover** : étant donnés un graphe G et un entier k , existe-t-il une couverture par sommets de G de cardinal au plus k ?

Si on sait résoudre le problème d'optimisation, on sait résoudre le problème à seuil associé ; le problème à seuil est donc au plus aussi difficile que le problème d'optimisation.

Calculabilité

Exercice

Soit $G = (S, A)$ un graphe non orienté.

On dit que $X \subset S$ est :

- une clique lorsque $\forall x, y \in X, x \neq y \Rightarrow \{x, y\} \in A$
- un ensemble indépendant lorsque $\forall x, y \in X, \{x, y\} \notin A$
- une couverture de sommets lorsque $\forall \{x, y\} \in A, x \in X \vee y \in X$

On note ici $\bar{G} = (S, \bar{A})$ où $\forall x, y \in S, x \neq y \Rightarrow (\{x, y\} \in \bar{A} \Leftrightarrow \{x, y\} \notin A)$.

On s'intéresse aux problèmes suivants :

Problème MaxClique

- Entrée : un graphe non orienté $G = (S, A)$ et un entier k .
- Sortie : existe-t-il une clique de G de cardinal $\geq k$?

Problème MaxIndSet

- Entrée : un graphe non orienté $G = (S, A)$ et un entier k .
- Sortie : existe-t-il un ensemble indépendant de G de cardinal $\geq k$?

Problème MinVertexCover

- Entrée : un graphe non orienté $G = (S, A)$ et un entier k .
- Sortie : existe-t-il une couverture de sommets de G de cardinal $\leq k$?

Questions :

1. Montrer que MaxIndSet est NP-complet par réduction de 3-SAT.
2. Soit $X \subset S$, montrer que les propriétés suivantes sont équivalentes :
 - (a) X est un ensemble indépendant de G ;
 - (b) X est une clique de $\bar{G}$;
 - (c) $S \setminus X$ est une couverture de sommets de G .
3. En déduire que MinVertexCover et MaxClique sont NP-complets.

Exercice

On vient de voir des problèmes de décision avec seuil. Ce sont des problèmes de décision classiques qu'on peut déduire de problèmes d'optimisation comme :

Problème MaxCliqueOpt

- Entrée : un graphe non orienté $G = (S, A)$.
- Sortie : quel est le cardinal maximal d'une clique de G ?

Montrer que si MaxClique est dans P alors MaxCliqueOpt peut également être résolu en temps polynomial.

Exercice

Problème SubsetSum

- Entrée : un multiensemble fini X d'éléments de $\mathbb{N}^*$ et $T \in \mathbb{N}$.
- Sortie : existe-t-il un sous-multiensemble $V \subset X$ tel que $\sum_{x \in V} x = T$?

Questions :

1. Montrer que SubsetSum est dans NP .
2. On va montrer que SubsetSum est NP -complet par réduction de MinVertexCover . Soit (G, k) une instance de MinVertexCover avec $G = (S, A)$ et $k \leq |S|$ (sinon l'instance est trivialement positive). On considère une numérotation dans $\llbracket 0, |A| - 1 \rrbracket$ des arêtes et on pose pour la i -ème arête $b_i = 4^i$. Pour chaque sommet $x \in S$, on pose $a_x = 4^{|A|} + \sum_{i \text{ incidente à } x} 4^i$. On pose également $T = k4^{|A|} + \sum_{i=0}^{|A|-1} 2 \cdot 4^i$ et on note X le multiensemble formé des $b_0, \dots, b_{|A|-1}$ et des a_x pour $x \in S$. Montrer qu'on a bien une réduction polynomiale permettant de prouver que SubsetSum est NP -difficile, et donc NP -complet.
3. Résoudre SubsetSum par programmation dynamique. On obtient alors un algorithme en $O(|X| \cdot T)$. A-t-on prouvé $P=NP$?

Exercice

On considère le problème BoxDepth qui étant donnés n rectangles dont les côtés sont parallèles aux axes d'un repère orthonormé, détermine le plus grand sous-ensemble de rectangles ayant un point en commun.

1. Décrire une réduction polynomiale de BoxDepth vers MaxClique .
2. Montrer que BoxDepth est dans P .
3. Pourquoi n'a-t-on pas résolu $P=NP$?

Exercice

On considère DNF-SAT qui est le pendant de CNF-SAT avec une formule en entrée donnée sous forme normale disjonctive.

1. Montrer que DNF-SAT est dans P .
2. Pourquoi ne peut-on pas en déduire que 3-SAT est dans P en mettant la formule sous forme normale disjonctive ?

Exercice

Le problème $3\text{-SAT-AllOrNothing}$ cherche à déterminer s'il existe une valuation d'une formule de 3-SAT dans laquelle chaque clause contient soit trois littéraux vrais, soit trois littéraux faux.

1. Montrer que ce problème est dans P .
2. Pourquoi n'a-t-on pas résolu $P=NP$?

Exercice

Le problème Partition consiste, étant donné un multiensemble S de n entiers naturels, à déterminer s'il existe deux sous-multiensembles $A, B \subset S$ tels que $A \cup B = S, A \cap B = \emptyset$ et

$$\sum_{a \in A} a = \sum_{b \in B} b$$

1. Montrer que Partition est NP -complet.
2. Donner un algorithme en temps $O(nM)$ où $M = \sum_{s \in S} s$ résolvant Partition .

3. A-t-on démontré que $P=NP$?
4. On s'intéresse maintenant au problème 2-Partition : étant donnés $2n$ entiers, peut-on les séparer en n paires de même somme ? Montrer que 2-Partition est dans P.

PARTIE V

Systeme

21	Concurrence et parallélisme	219
22	TP Threads, Peterson, mutex et sémaphores	245

Concurrence et parallélisme

DANS CE CHAPITRE

- 21.1 Processus, threads
- 21.2 Section critique et exclusion mutuelle
- 21.3 Verrou, mutex
- 21.4 Sémaphores
- 21.5 Problèmes classiques

L'objectif de ce chapitre est de considérer des programmes qui s'exécutent au même moment, soit chacun leur tour, soit simultanément.

On va ainsi observer deux phénomènes majeurs :

- la **concurrence** quand plusieurs programmes veulent accéder à une même ressource
- le **parallélisme** quand deux programmes s'exécutent au même moment sur des unités de calcul différentes (des processeurs ou des cœurs)

Il faut donc, pour que les interactions se passent bien, qu'on soit capable de gérer la synchronisation entre les programmes.

Remarque 21.1 La caractérisation de la concurrence et du parallélisme faite ici est assez naïve mais convient pour une première exposition.

21.1 Processus, threads

21.1.1 Processus

Un processus est un programme s'exécutant dans un environnement. Il dispose du code binaire du programme, et des bibliothèques qu'il utilise, ainsi qu'un espace mémoire dont on a vu l'organisation dans un précédent chapitre.

Quand un programme s'exécute, il a plein accès à une unité de calcul du processeur. Il peut ainsi modifier les registres du processeur.

Remarque 21.2 Suite à la limite d'évolution des fréquences d'horloge de processeur, il y a eu un développement des processeurs multi-cœurs (*cores* en anglais) permettant de proposer des unités de calculs indépendantes mais reliées. Mettant ainsi à profit des mécanismes partagés entre les différents cœurs.

C'est pour correspondre à cette réalité qu'on a fait le choix ici de parler d'unité de calcul d'un processeur.

Pour autant, un programme ne s'exécute pas d'une traite à chaque lancement. Si c'était le cas, une boucle `while` pourrait ainsi bloquer le processeur. En fait, le système d'exploitation définit des interruptions qui

permettent dans certains cas au processeur d'interrompre l'exécution du programme en cours pour basculer dans le système d'exploitation.

C'est ainsi que le système d'exploitation va pouvoir distribuer l'accès au processeur à différents programmes. Chaque fois qu'il faut passer d'un programme à un autre, il est nécessaire de restaurer tout son environnement d'exécution : on parle de contexte d'exécution, il contient l'organisation de la mémoire mais aussi des copies des valeurs des registres avant la bascule. Ce procédé est assez coûteux en temps et s'appelle la bascule de contexte, ou *context switching*.

Pour qu'un processus en lance un autre, par exemple pour réaliser une autre tâche en parallèle, il est nécessaire qu'il se duplique complètement. Un appel système est disponible pour cela `fork()`. Toute mémoire partagée entre les deux processus doit alors avoir été créée explicitement avec des appels systèmes. Ce modèle d'exécution n'est pas au programme en MPI.

21.1.2 Threads

Le point de départ des *threads*, ou *fils d'exécution* en français, est qu'une séparation complète d'un processus en deux copies est inutilement lourde :

- elle demande de copier toutes les données d'un processus, bien qu'on puisse imaginer que la copie est superficielle
- elle limite les possibilités de partage immédiat de mémoire

Un *thread* est juste une exécution d'un processus, c'est-à-dire où on est dans l'exécution et quel est le contexte, i.e. la pile d'exécution. Il est toujours nécessaire de sauvegarder le contexte des registres pour passer d'un thread à un autre, mais deux threads lancés par un même processus partageront tout le reste de la mémoire.

Ainsi, les threads d'un même processus partagent :

- les variables globales
- le tas

mais ont un accès exclusif et personnel

- au processeur et à ses registres
- à leur pile d'exécution

Pour passer d'un thread à un autre, il est encore nécessaire de faire une bascule de contexte. Cependant, on peut imaginer que la bascule de contexte entre deux threads d'un même processus peut être réalisée plus rapidement qu'entre deux processus.

Le fait que la mémoire soit partagée permet de communiquer entre les threads de manière directe.

Les différents threads vont pouvoir être interrompus par le système comme les processus. C'est le système d'exploitation qui va ainsi être responsable du choix du thread à exécuter. Quand on dispose de plusieurs unités de calcul, les threads peuvent s'exécuter en parallèle.

C'est le cadre choisi pour ce chapitre en MPI.

21.1.3 Threads coopératifs

Comme pour les processus, les threads sont gérés au niveau du système d'exploitation. Cela n'est donc pas forcément portable d'un système à un autre.

Il est possible de permettre la concurrence qu'offrent les threads sans qu'ils correspondent à de vrais threads systèmes. On parle de threads coopératifs car, en l'absence d'intervention du système d'exploitation, ce sont ces threads eux-mêmes qui vont laisser la main, soit manuellement avec une opération de *yield*, soit en redéfinissant les appels systèmes.

Des bibliothèques comme `lwj` (en OCaml) permettent de gérer cela.

21.1.4 Quelle utilité pratique ?

Si l'essentiel du cours de MP2I/MPI a été traité dans le cadre d'une exécution séquentielle d'un unique programme, il serait peu avisé d'en déduire qu'il s'agit du cadre standard. La plupart des programmes peuvent tirer parti de la concurrence et/ou du parallélisme.

Les programmes devant effectuer de lourdes tâches de calcul ont intérêt à exploiter le matériel au maximum, et sur un processeur à huit cœurs, il semble peu efficace d'en utiliser un seul. Au premier abord, on pourrait penser obtenir un facteur x8 dans l'efficacité dans ce cas. Cependant, pour que les calculs se déroulent bien, le parallélisme a un coût et utiliser à 100% le processeur pour un calcul est un objectif inaccessible sauf dans des cas très spécifiques où on effectue des calculs sans liens, comme pour du lancer de rayon sommaire.

Les programmes interactifs vont bénéficier de la concurrence pour ne pas bloquer l'interface. Afin d'avoir une interface réactive, il ne faut pas qu'un calcul ou un appel système prenne du temps et empêche d'interagir avec elle.

21.1.5 Lancer des threads en C

Pour créer des threads, on utilise la bibliothèque `pthread` dont l'en-tête est `pthread.h`.

Remarque 21.3 Si vous avez un problème à l'édition de lien, il faut rajouter `-lpthread`.

Pour créer un thread :

- on définit une valeur `pthread_t t` qui va contenir les données du thread
- on appelle `pthread_create(&t, NULL, f, arg)`; où `f` est une fonction de prototype `void *f(void *arg)`; et `arg` est une valeur de type `void *`

Remarque 21.4 En C, le type `void *` joue un rôle particulier :

- toute valeur de type `T *a` peut être convertie en un `void *`
- toute valeur de type `void *v` peut être convertie en un `T *`

Ces conversions sont **implicites** en C, il n'y a rien à écrire.

Ainsi, le code suivant est correct :

```
#include <stdio.h>

void incremente(void *a)
{
    int *t = a;
    *t = *t + 1;
}

int main(void)
{
    int n = 3;
    incremente(&n);
    printf("n = %d\n", n);
}
```

Ainsi, il est possible de faire des fonctions génériques en utilisant le type `void *` comme une sorte de 'a en OCaml.

Comme le type `int` est, sur les architectures usuelles, plus petit que le type `void *` en termes de taille, il est possible de faire des conversions explicites pour ne pas avoir à définir une variable comme `n` ici.

Par exemple, on définit la fonction `incremente` qui va prendre un entier en argument par pointeur et l'incrémenter.

```
void *incremente(void *arg)
{
    int *n = arg;
```

```

    *n = *n + 1;

    return NULL; // pas de retour
}

```

On peut alors lancer cette fonction dans un thread :

```

pthread_t t;
int v = 3;
pthread_create(&t, NULL, incremente, &v);

```

Pour attendre qu'un thread `t` termine on appelle `pthread_join(t, NULL)` :

```

pthread_join(t, NULL);
printf("%d\n", v);

```

Remarque 21.5 Il est également possible d'avoir une valeur de retour mais c'est hors programme.

Pour lancer plusieurs threads, il suffit de faire plusieurs appels à `pthread_create`.

Remarque 21.6 Lancer deux threads

Si on écrit :

```

pthread_t t1, t2;
pthread_create(&t1, NULL, f, NULL);
pthread_join(t1, NULL);
pthread_create(&t2, NULL, f, NULL);
pthread_join(t2, NULL);

```

Les deux threads ne vont pas s'exécuter en même temps car on attend avec le `join` que le premier thread termine pour lancer le second.

Il faut donc créer les threads puis les attendre :

```

pthread_t t1, t2;
pthread_create(&t1, NULL, f, NULL);
pthread_create(&t2, NULL, f, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);

```

Ici, on privilégie l'attente sur le premier, mais on pourrait le faire sur le second.

Il arrive qu'on ait besoin de passer un identificateur pour chaque thread, son indice dans le tableau convient alors simplement. Avec les limites du programme, il faut alors écrire :

```

pthread_t t1, t2;
int i1 = 0, i2 = 1;
pthread_create(&t1, NULL, f, &i1);
pthread_create(&t2, NULL, f, &i2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);

```

Remarque 21.7 Lancer n threads au démarrage

Pour lancer un grand nombre de threads, il va être nécessaire de stocker les `pthread_t` pour pouvoir appeler `join`. Le plus simple est alors de passer par un `pthread_t[]`.

Si le nombre est connu à l'avance, on peut réaliser un tableau statique :

```

pthread_t t[10];
for(int i = 0; i < 10; i++) // on les lance

```

```
pthread_create(&t[i], NULL, f, NULL);
// ensuite quand on veut les attendre, il faut refaire une boucle
for(int i = 0; i < 10; i++)
    pthread_join(t[i], NULL);
```

Si le nombre n n'est pas connu, on peut effectuer un tableau avec `malloc`, mais il ne faut pas oublier de libérer après les `join`.

Si on a besoin d'avoir les identificateurs pour les threads, dans les limites du programme, c'est un peu maladroit, il faut réaliser un tableau d'entiers pour les indices :

```
pthread_t t[10];
int t_idx[10];
for(int i = 0; i < 10; i++) { // on les lance
    t_idx[i] = i;
    pthread_create(&t[i], NULL, f, &t_idx[i]);
}
```

Remarque 21.8 *Avancé* : Lancer des threads avec un pool

Il n'est pas rare d'avoir besoin d'effectuer des tâches à la volée et avoir des tableaux de `pthread_t` n'est pas le plus souple. On utilise plutôt un sac de `pthread_t` :

- quand on lance un nouveau thread, on l'ajoute dans le sac
- en sortie de programme, on réalise des `join(retire(sac))` tant que le sac est non vide.

21.2 Section critique et exclusion mutuelle

21.2.1 Définition

Définition 21.9

On appelle **section critique** une suite d'instructions qui ne peut pas être exécutée par deux threads en même temps. Cette propriété est appelée l'**exclusion mutuelle**.

Remarque 21.10 Cela suppose une symétrie entre les threads qui n'est pas toujours souhaitable. Parfois, deux threads doivent exécuter deux suites d'instructions différentes mais il faut tout de même s'assurer qu'il y a exclusion mutuelle, on parlera alors encore abusivement de section critique.

21.2.2 Exemple de section critique : compteur partagé

On considère ici deux threads t_0 et t_1 qui vont chacun incrémenter n fois un compteur entier global initialement à 0.

Le pseudo-code de chaque thread est donc :

```
Répéter n fois
    compteur = compteur + 1
```

On s'attendrait ainsi au fait qu'au bout de $2n$ incréments, la valeur du compteur soit à $2n$. Cependant, en exécutant ce programme en présence de parallélisme, on constate que la valeur du compteur n'est pas déterministe et qu'elle est inférieure à $2n$.

Notons lit_i et $ecrit_i$ les événements indiquant que le thread t_i va lire ou écrire dans le compteur. Ces événements ont un temps de début et un temps de complétion, si le temps de complétion de e est inférieur au temps de départ de e' , on notera $e \rightarrow e'$.

Remarque 21.11 La relation $\rightarrow$ est une relation d'ordre stricte partielle. Deux événements sont incomparables quand il existe un instant pendant lequel les deux événements se produisent.

On a donc deux suites d'événements : $lit_i \rightarrow écrit_i$ qui vont s'entrelacer.

Si on observe

$$lit_0 \rightarrow écrit_0 \rightarrow lit_1 \rightarrow écrit_1$$

tout se passe bien, si la valeur du compteur était x au départ, le thread 0 va lire x , écrire $x + 1$, le thread 1 va lire $x + 1$ et écrire $x + 2$.

Mais, si on observe

$$lit_1 \rightarrow écrit_0 \text{ et } lit_0 \rightarrow écrit_1$$

alors les deux threads vont lire x et écrire $x + 1$, ce qui ne va pas réaliser les deux incrémentations.

Ces incrémentations perdues sont celles qui induisent une valeur plus faible du compteur.

Remarque 21.12 Même si ça a peu de chances de se produire en pratique, car cela signifierait qu'on a interrompu un thread entre l'évaluation et l'affectation, il peut être intéressant de considérer que le problème se pose même en l'absence de vrai parallélisme.

21.2.3 Modèle mémoire

Afin de raisonner, on va adopter un modèle mémoire simple, même s'il n'est pas forcément réaliste.

On fait ainsi plusieurs hypothèses :

- **Écritures simultanées** : si deux écritures en mémoire ont lieu en même temps à la même adresse, on suppose que la dernière écriture est complète. Si les deux finissent en même temps, celle qui est effectivement complétée est non déterministe.
- **Séquentialité des accès** : si deux accès ont lieu à la suite, ils sont effectivement réalisés dans cet ordre en mémoire.
- **Pas d'optimisation** : toutes les instructions d'écriture ou de lecture sont effectuées dans le code final.

Remarque 21.13 Ces hypothèses peuvent sembler très naturelles mais elles sont peu réalistes dans un contexte moderne. On cite ici ce qui pourrait se passer en pratique :

- **Écritures simultanées** : rien ne dit qu'il y a atomicité des écritures de données pour certaines tailles de valeurs. Ainsi, un entier sur 64 bits pourrait être décomposé en deux entiers sur 32 bits, et l'écriture des deux parties qui en résulte pourrait ne pas être cohérente. Pour faire un exemple plus visuel, imaginons qu'un thread écrive AA pendant qu'un autre écrit BB, on pourrait se retrouver avec AB ou BA ce qui ne correspond à aucune écriture réelle.
- **Séquentialité des accès** : chaque écriture mémoire n'est pas faite directement, elle peut passer par un cache propre au thread et n'être réalisée qu'à la fin d'une suite d'opérations. Pour ne pas avoir ce problème, on peut utiliser des barrières mémoires.
- **Pas d'optimisation** : vu du point de vue d'un seul thread, certaines instructions que l'on va écrire semblent être inutiles, elles ne sont là que pour gérer l'interaction entre plusieurs threads. Si on ne fait pas attention, le compilateur risque de les optimiser.

21.2.4 Instructions atomiques

Afin de faciliter la synchronisation, le processeur dispose d'instructions **atomiques** : elles effectuent une lecture suivie d'une écriture en mémoire sans qu'aucun autre accès à cette case ne puisse s'intercaler, comme si elles étaient exécutées en section critique.

On a notamment les deux instructions :

- *fetch and add* pour incrémenter une valeur en mémoire sans problème entre la lecture des données et l'écriture
- *test and set* pour écrire true dans une case mémoire tout en renvoyant l'ancienne valeur de cette case (ce qui permet de tester et de modifier un drapeau en une seule opération).

21.3 Verrou, mutex

21.3.1 Définition

Définition 21.14

Un **verrou** (en anglais *lock*) est une primitive qui permet d'assurer l'exclusion mutuelle entre deux ou plusieurs threads.

Il s'agit d'un type verrou et de quatre opérations :

- `void verrou_init(verrou *)` alloue et initialise le verrou
- `void verrou_detruit(verrou *)` détruit le verrou
- `void verrou_verrouille(verrou *)` attend que le verrou soit déverrouillé et le verrouille
- `void verrou_deverrouille(verrou *)` déverrouille le verrou

En anglais, on utilise les termes *lock* et *unlock* pour les deux opérations. Un verrou est plus souvent appelé un *mutex*, car sa raison d'être est de garantir l'exclusion mutuelle (**mutual exclusion**).

Le cas usuel d'utilisation du verrou est :

```
verrou_verrouille(&v);
// SECTION CRITIQUE
verrou_deverrouille(&v);
```

Quand on parlera des propriétés d'un verrou, on supposera qu'il est toujours utilisé dans ce cadre-là.

Remarque 21.15 L'attente du verrou peut être réalisée :

- par une boucle active, un `while`, on parle alors d'**attente active**
- en utilisant un mécanisme du système d'exploitation pour qu'il réveille le thread quand le verrou est disponible.

21.3.2 Propriétés d'un verrou

On peut demander plusieurs propriétés pour un verrou :

- l'exclusion mutuelle, un seul thread peut exécuter la section critique à un moment donné
- l'absence d'interblocage : si un thread effectue un appel à `verrouille` alors au moins un thread, peut-être un autre, va rentrer en section critique ensuite, on parle aussi de *vivacité*
- l'absence de famine : si un thread effectue un appel à `verrouille`, il ne va pas attendre indéfiniment
- *premier arrivé, premier servi* : si un thread demande à verrouiller avant un autre, il prendra le verrou avant lui

👉 Pour les deux propriétés de vivacité, on suppose naturellement que chaque thread en section critique finit par en sortir (et donc par appeler `deverrouille`) et que chaque thread finit toujours par être exécuté par l'ordonnanceur.

Remarque 21.16 Pour donner un sens précis à *demandeur avant*, on découpe `verrouille` en une première partie, appelée **porte d'entrée**, qui s'exécute en un nombre borné d'étapes (sans attente), suivie de la partie d'attente. Un thread *A* demande le verrou avant un thread *B* quand *A* a terminé sa porte d'entrée avant que *B* ne commence la sienne.

On a alors le théorème suivant :

Théorème 21.17

- Si un verrou vérifie *premier arrivé, premier servi* et l'absence d'interblocage, alors il y a absence de famine.
- S'il y a absence de famine, alors il y a absence d'interblocage.

- On considère n threads et un thread A qui termine sa porte d'entrée. Chaque thread n'ayant qu'une demande en cours à la fois, au plus $n - 1$ threads ont commencé leur porte d'entrée avant que A n'ait fini la sienne sans être encore entrés en section critique. Tout autre thread qui demande le verrou commence sa porte d'entrée après la fin de celle de A , donc passe après A par la propriété *premier arrivé, premier servi*. Par absence d'interblocage, tant que A attend, des threads entrent en section critique : ce ne peuvent être que les au plus $n - 1$ threads précédents, chacun au plus une fois avant A . Donc A finit par entrer en section critique.
- Si un thread appelle verrouille, par absence de famine, ce thread lui-même finit par entrer en section critique, ce qui est plus fort que l'absence d'interblocage.

□

Remarque 21.18 L'hypothèse d'absence d'interblocage est nécessaire dans le premier point : un verrou qui ne laisse jamais entrer personne vérifie trivialement *premier arrivé, premier servi*.

21.3.3 Le verrou de Peterson

On considère un problème de section critique à deux threads, on suppose que chaque thread a un identifiant 0 ou 1 qu'il connaît. On peut reprendre l'exemple du compteur partagé et on va chercher à remplir la partie en commentaire avec les variables définissant le verrou et le code des fonctions associées. Ici, on passe l'identificateur du thread au verrou.

```
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>

int compteur = 0;
int N = 100000;

/*
 * les variables globales permettant de gérer le verrou
 * Le code de verrouille et de deverrouille
 */

void *incrimente(void *arg)
{
    int *i_arg = arg;
    int id = *i_arg; // le numéro du thread

    for(int i = 0; i < N; i++) {
        verrouille(id);
        compteur = compteur + 1;
        deverrouille(id);
    }
    return NULL;
}

int main(void)
{
    pthread_t t0, t1;
    int i0 = 0, i1 = 1;
    pthread_create(&t0, NULL, incrimente, &i0);
    pthread_create(&t1, NULL, incrimente, &i1);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
    printf("compteur = %d (attendu %d)\n", compteur, 2 * N);
}
```

Première idée : des drapeaux spécifiques

On considère deux drapeaux indiquant si un thread veut verrouiller ainsi qu'un système de politesse : tant qu'un autre thread veut verrouiller, on attend.

```

bool flags[2] = {}; // initialise deux flags à false

void verrouille(int id)
{
    int autre = 1 - id;

    flags[id] = true;
    while(flags[autre]) { // attente active
    }
}

void deverrouille(int id)
{
    flags[id] = false;
}

```

Théorème 21.19

Ce verrou satisfait l'exclusion mutuelle.

Preuve Par l'absurde, supposons que les deux threads sont simultanément dans la section critique. On va raisonner sur le dernier appel à `verrouille` ayant été effectué par chaque thread.

Comme le thread 0 a réussi à obtenir le verrou, il a réalisé la suite d'événements : $ecrit_0(flags[0] = true) \rightarrow lit_0(flags[1] == false) \rightarrow SC_0$ où SC_0 est la section critique pour le thread 0.

De même, le thread 1 a réalisé : $ecrit_1(flags[1] = true) \rightarrow lit_1(flags[0] == false) \rightarrow SC_1$.

Comme le thread 0 a lu $flags[1] == false$ alors que le thread 1 ne remet $flags[1]$ à `false` qu'en sortant de sa section critique, c'est forcément que l'écriture $flags[1] = true$ a été réalisée ensuite : $lit_0(flags[1] == false) \rightarrow ecrit_1(flags[1] = true)$.

Par transitivité, on a donc $ecrit_0(flags[0] = true) \rightarrow ecrit_1(flags[1] = true)$.

Or, par symétrie, on peut faire le même raisonnement en échangeant les threads 0 et 1, donc on a également $ecrit_1(flags[1] = true) \rightarrow ecrit_0(flags[0] = true)$.

Comme la relation $\rightarrow$ est un ordre strict, c'est impossible.

Contradiction.

□

Si jamais deux threads appellent `verrouille` au même moment (plus précisément, si chacun écrit son drapeau avant que l'autre ne lise le sien), chacun va attendre dans la boucle `while` sans espoir que la valeur `flags` qu'ils observent ne puisse changer vu que l'autre thread responsable sera également bloqué.

Ainsi, il y a un interblocage.

Deuxième idée : un drapeau partagé

On peut se dire que le mécanisme utilisant deux drapeaux échoue en raison de l'indépendance des écritures. On va maintenant utiliser un drapeau partagé `victim` qui indique quel thread doit attendre. Comme il y a une unique valeur, on est certain que le second thread à verrouiller va changer la valeur.

```

int victim = 0;

void verrouille(int id)
{
    victim = id;

    while(victim == id) { // attente active
    }
}

void deverrouille(int id)

```

```
{
}
```

Théorème 21.20

Ce verrou satisfait l'exclusion mutuelle.

Preuve Par l'absurde, supposons que les deux threads sont simultanément dans la section critique et considérons le dernier appel à `verrouille` de chaque thread.

Le thread 0 a réalisé $ecrit_0(victim = 0) \rightarrow lit_0(victim == 1) \rightarrow SC_0$. Comme il a lu la valeur 1 après avoir écrit 0, le thread 1 a écrit `victim = 1` entre les deux. Or le thread 1 n'écrit dans `victim` qu'une fois par appel à `verrouille` et il est en section critique : il s'agit donc de l'écriture de son dernier appel et $ecrit_0(victim = 0) \rightarrow ecrit_1(victim = 1)$.

Par symétrie, on a également $ecrit_1(victim = 1) \rightarrow ecrit_0(victim = 0)$, ce qui est impossible car $\rightarrow$ est un ordre strict.

Contradiction. □

Cependant, on remarque que pour être débloqué, un thread doit attendre qu'un autre thread verrouille. Il y a donc interblocage quand un thread est seul à verrouiller, ce qui est toujours le cas quand un des deux threads termine avant.

Algorithme de Peterson

Le verrou de Peterson consiste à fusionner les deux comportements.

```
bool flags[2] = {}; // initialise deux flags à false
int victim = 0;

void verrouille(int id)
{
    int autre = 1 - id;

    flags[id] = true;
    victim = id;
    while(flags[autre] && victim == id) { // attente active
    }
}

void deverrouille(int id)
{
    flags[id] = false;
}
```

Remarque 21.21 L'ordre des deux écritures `flags[id] = true;` puis `victim = id;` est essentiel. Si on les inverse, l'exécution suivante viole l'exclusion mutuelle : le thread 0 écrit `victim = 0`, puis le thread 1 écrit `victim = 1` et `flags[1] = true`, lit `flags[0] == false` et entre en section critique ; le thread 0 écrit alors `flags[0] = true`, lit `victim == 1` et entre aussi en section critique.

Théorème 21.22

Ce verrou satisfait l'exclusion mutuelle.

Preuve Par l'absurde, supposons que les deux threads sont simultanément dans la section critique et considérons le dernier appel à `verrouille` de chaque thread. Sans perte de généralité, on suppose que le thread 0 est le dernier à avoir écrit dans `victim` : $ecrit_1(victim = 1) \rightarrow ecrit_0(victim = 0)$.

Le thread 0 est sorti de la boucle après `ecrit0(victim = 0)`, en lisant soit `flags[1] == false`, soit `victim == 1`. Comme personne n'écrit dans `victim` après `ecrit0` (le thread 1 est en section critique), il a nécessairement lu `flags[1] == false`.

Mais on a `ecrit1(flags[1] = true) → ecrit1(victim = 1) → ecrit0(victim = 0) → lit0(flags[1])` et le thread 1 ne remet `flags[1]` à `false` qu'en sortant de sa section critique : le thread 0 a donc lu `flags[1] == true`.

Contradiction.

□

Théorème 21.23

Ce verrou satisfait l'absence de famine.

Preuve Par l'absurde, supposons qu'il y ait famine. Sans perte de généralité, on peut supposer que le thread 0 reste indéfiniment bloqué dans la boucle de verrouille. Il lit donc indéfiniment `flags[1] == true` et `victim == 0`.

- Si le thread 1 ne demande plus le verrou, `flags[1]` vaut `false` à partir d'un certain moment et le thread 0 sort de la boucle.
- Le thread 1 ne peut pas rester en section critique indéfiniment (hypothèse) ni être bloqué dans sa boucle, car `victim` vaut 0 et la condition `victim == 1` est fausse.
- Il reste le cas où le thread 1 sort de sa section critique puis rappelle verrouille. Il écrit alors `victim = 1`, et comme seul le thread 1 pourrait remettre `victim` à 0, le thread 0 lit ensuite `victim == 1` et sort de sa boucle.

Contradiction.

□

Remarque 21.24 En lançant le programme précédent d'incrémentement avec ce verrou, il est probable de ne pas observer 200000 dans le compteur bien que les deux threads fassent chacun cent mille incréments dans une section critique.

En effet, les hypothèses sur la gestion de la mémoire ne sont pas satisfaites sur des ordinateurs récents. De plus, en langage C, ces accès concurrents non protégés à `flags` et `victim` constituent des *data races* dont le comportement n'est pas défini : avec des optimisations (-O2), le compilateur peut par exemple ne lire `victim` qu'une seule fois avant la boucle d'attente. On peut *réparer* ce verrou en forçant le processeur à gérer la mémoire comme on l'attend en plaçant des barrières mémoire à l'aide de l'instruction :

```
asm volatile ("mfence" ::: "memory");
```

En pratique, il semble que placer une barrière avant d'entrer dans l'attente active suffit. Cependant, on devrait la placer avant chaque accès mémoire, ce qui ralentirait fortement le programme. En effet, ces instructions ont un coût qui peut parfois dépasser le bénéfice obtenu par l'exécution parallèle.

21.3.4 L'algorithme de la boulangerie de Lamport

L'algorithme de la boulangerie de Lamport est une généralisation du verrou de Peterson pour n threads. L'idée est d'associer à chaque thread un numéro de ticket et de débloquent à chaque fois le thread de plus petit numéro de ticket, comme lorsqu'on fait la queue dans un commerce.

Ici, on n'aura aucune garantie que les numéros de tickets soient uniques, mais en considérant un couple (ticket, identificateur) on peut toujours déterminer le thread prioritaire.

On va conserver les drapeaux comme avec le verrou de Peterson, les numéros de tickets vont eux permettre de généraliser le rôle qu'avait `victim` précédemment.

Ici, les numéros de tickets sont stockés dans le tableau `labels` et, comme pour Peterson, on a un tableau `flags` de booléens.

```
bool flags[10] = {};
int labels[10] = {};
```

Pour verrouiller, on passe le drapeau du thread à `true` et on choisit pour numéro de ticket $1 + \max_k \text{labels}[k]$ qui est calculé dans une boucle. Ces deux étapes constituent la porte d'entrée.

Ensuite, on attend tant qu'il existe un thread k tel que `flags[k] == true` et dont le couple $(\text{labels}[k], k) < (\text{labels}[i], i)$ dans l'ordre lexicographique.

```
void lock(int i)
{
    flags[i] = true;
    int m = labels[0];
    for(int k = 1; k < 10; k++)
        if(m < labels[k])
            m = labels[k];
    labels[i] = 1 + m;

    while(true) {
        bool cango = true;
        for(int k = 0; k < 10; k++) {
            if (k != i && flags[k]
                && (labels[k] < labels[i]
                    || (labels[k] == labels[i] && k < i)))
                cango = false;
        }
        if (cango)
            break;
    }
}
```

Comme annoncé, les `labels` généralisant `victim`, il y a la même fonction que précédemment pour déverrouiller.

```
void unlock(int i)
{
    flags[i] = false;
}
```

Théorème 21.25

Ce verrou satisfait l'exclusion mutuelle, l'absence d'interblocage et la propriété *premier arrivé, premier servi*. Il satisfait donc l'absence de famine.



Preuve On remarque d'abord que les valeurs successives de `labels[i]` sont strictement croissantes, car le maximum calculé par le thread i prend en compte son propre ancien ticket.

- *Absence d'interblocage* : supposons que des threads attendent indéfiniment sans qu'aucun thread n'entre en section critique. Comme les portes d'entrée sont de longueur bornée, à partir d'un certain moment plus aucun drapeau ni ticket ne change. Le thread en attente A dont le couple $(\text{labels}[A], A)$ est minimal parmi les threads ayant leur drapeau levé effectue alors une itération complète de la boucle sans trouver de thread prioritaire : il sort de la boucle. Contradiction.
- *Premier arrivé, premier servi* : si A termine sa porte d'entrée avant que B ne commence la sienne, alors B lit `labels[A]` après son écriture par A , donc `labels[B] > labels[A]`. Tant que A n'est pas sorti de section critique, `flags[A]` vaut `true` et $(\text{labels}[A], A) < (\text{labels}[B], B)$: B attend.
- *Exclusion mutuelle* : supposons A et B simultanément en section critique, avec $(\text{labels}[A], A) < (\text{labels}[B], B)$ pour les tickets de leurs derniers appels. Lors de sa dernière vérification de A , B a lu `flags[A] == false` ou un ticket de A tel que le couple correspondant soit plus grand que $(\text{labels}[B], B)$. Les tickets de A étant croissants, un ancien ticket de A donne un couple encore plus petit : B a

donc $\text{flags}[A] == \text{false}$, avant que A ne lève son drapeau lors de son dernier appel. On a ainsi : $\text{ecrit}_B(\text{labels}[B]) \rightarrow \text{lit}_B(\text{flags}[A]) \rightarrow \text{ecrit}_A(\text{flags}[A] = \text{true}) \rightarrow \text{lit}_A(\text{labels}[B]) \rightarrow \text{ecrit}_A(\text{labels}[A])$. Le thread A a donc lu le ticket de B et choisi $\text{labels}[A] > \text{labels}[B]$, ce qui contredit $(\text{labels}[A], A) < (\text{labels}[B], B)$.

L'absence de famine découle alors du théorème sur les propriétés des verrous.

□

21.3.5 Utiliser les mutex de pthread

Les deux mécanismes de verrou que l'on vient de voir ont plusieurs défauts :

- ils sont basés sur des attentes actives
- ils supposent des hypothèses fortes sur la gestion de la mémoire

Comme souvent, il vaut mieux reposer sur des primitives déjà programmées et qui ont souvent été adaptées pour coller au mieux aux architectures sur lesquelles on développe.

Remarque 21.26 Une des premières versions sous Linux de la bibliothèque pthread que l'on va utiliser a été écrite par Xavier Leroy, qui n'est autre que l'architecte principal derrière Caml Light puis OCaml.

Cela prouve qu'on peut à la fois écrire l'histoire en C et en OCaml !

Pour avoir accès à ces fonctions, il faut également inclure `pthread.h`.

Le type des verrous est `pthread_mutex_t`.

Initialisation

On peut initialiser un verrou déjà alloué de deux manières :

- si c'est un verrou statique, il suffit de lui donner la valeur initiale `PTHREAD_MUTEX_INITIALIZER` :

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- dans tous les cas, si on a un pointeur `pthread_mutex_t *m`, on va appeler `pthread_mutex_init(m, NULL)`. Ce pointeur peut être obtenu depuis une allocation statique :

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

et dans ce cas c'est équivalent à l'initialisation précédente, ou alors dynamiquement :

```
pthread_mutex_t *m = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(m, NULL);
```

Verrouiller et déverrouiller un mutex

On utilise les deux fonctions suivantes pour un mutex `m` :

```
pthread_mutex_lock(&m); // verrouille
pthread_mutex_unlock(&m); // déverrouille
```

Destruction

Quand on a fini d'utiliser un mutex, on le détruit avec l'appel suivant :

```
pthread_mutex_destroy(&m);
```

Exemple d'utilisation

On reprend le code précédent de l'incrément d'un compteur partagé :

```
#include <pthread.h>
#include <stdlib.h>
```

```

#include <stdio.h>
#include <stdbool.h>

int compteur = 0;
int N = 100000;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *incrémente(void *arg)
{
    for(int i = 0; i < N; i++) {
        pthread_mutex_lock(&m);
        compteur = compteur + 1;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

int main(void)
{
    pthread_t t0, t1;
    pthread_create(&t0, NULL, incrémente, NULL);
    pthread_create(&t1, NULL, incrémente, NULL);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);

    printf("compteur = %d (attendu %d)\n",
        compteur, 2 * N);
    pthread_mutex_destroy(&m);
}

```

21.4 Sémaphores

21.4.1 Principe

Les sémaphores généralisent les verrous. Il s'agit de compteurs partagés de ressources avec deux opérations :

- rendre une ressource en incrémentant un compteur
- attendre qu'une ressource soit disponible et décrémenter le compteur

Une manière intuitive de voir cela est de considérer une pièce dans laquelle seulement n personnes peuvent être présentes à la fois :

- on considère devant un réceptacle avec n jetons
- pour entrer on attend que le réceptacle ne soit pas vide, on prend un jeton et on entre
- en sortant, on repose son jeton dans le réceptacle.

21.4.2 Cas initial $n = 1$: mutex

Quand le sémaphore dispose initialement d'un seul jeton, une seule personne peut rentrer dans la pièce à la fois : on retrouve l'exclusion mutuelle :

Un sémaphore de valeur initiale 1 permet de réaliser un mutex.

21.4.3 Cas initial $n = 0$: signal

Si le sémaphore a une valeur initiale de 0, personne ne peut prendre de ressource initialement. Cela peut sembler inutile, mais rien n'impose que la valeur initiale d'un sémaphore soit sa valeur maximale. Ainsi, on peut incrémenter le sémaphore ce qui va permettre de débloquer un thread attendant pour la décrémenter.

Ce mécanisme permet ainsi de réaliser un signal. Si le thread 1 veut attendre que le thread 2 ait fini de faire une action pour continuer :

- on crée un sémaphore de valeur 0
- le thread 1 attend le sémaphore et bloque
- quand il le souhaite, le thread 2 peut incrémenter le sémaphore

- cela débloquent alors le thread 1

On parle de signal, c'est très pratique de raisonner en termes de signaux pour réaliser des synchronisations complexes.

21.4.4 Utiliser les sémaphores avec pthread

Pour utiliser les sémaphores, on va inclure l'en-tête `semaphore.h` qui définit le type `sem_t` des sémaphores.

Initialiser un sémaphore

On initialise un sémaphore `s` avec la valeur initiale `n` ainsi :

```
sem_init(&s, 0, n);
```

Le second paramètre sera toujours à 0 pour nous.

Libérer un sémaphore

Une fois qu'on a fini de l'utiliser, on le libère avec

```
sem_destroy(&s);
```

Attendre une ressource

Pour attendre un sémaphore, et donc décrémenter ensuite sa valeur, on utilise

```
sem_wait(&s);
```

Rendre une ressource

L'opération inverse de l'attente, appelée *post* en anglais, qui va incrémenter la valeur est :

```
sem_post(&s);
```

21.4.5 Réalisation d'un mutex

On reprend l'exemple précédent pour montrer la richesse des sémaphores en réalisant un mutex :

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

int compteur = 0;
int N = 100000;

sem_t s;

void *incrimente(void *arg)
{
    for(int i = 0; i < N; i++) {
        sem_wait(&s);
        compteur = compteur + 1;
        sem_post(&s);
    }
    return NULL;
}

int main(void)
{
    pthread_t t0, t1;
    sem_init(&s, 0, 1);
    pthread_create(&t0, NULL, incrimente, NULL);
    pthread_create(&t1, NULL, incrimente, NULL);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
}
```

```

    printf("compteur = %d (attendu %d)\n",
           compteur, 2 * N);
    sem_destroy(&s);
}

```

21.4.6 Utilisation d'un signal

On va considérer deux threads très simples : le thread 1 affiche A puis B, le thread 2 affiche C puis D. Contrainte : on demande que le B ne soit pas affiché avant que le thread 2 ait affiché C. Ainsi la trace, ce qui va être affiché sur la sortie, peut être par exemple ACDB ou CADB mais pas ABCD.

Pour cela, on va faire en sorte que le thread 2 signale qu'il a fini au thread 1 avec un sémaphore à 0.

```

#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

sem_t s;

void *threadAB(void *arg)
{
    putchar('A');
    sem_wait(&s);
    putchar('B');

    return NULL;
}

void *threadCD(void *arg)
{
    putchar('C');
    sem_post(&s);
    putchar('D');

    return NULL;
}

int main(void)
{
    pthread_t t1, t2;
    sem_init(&s, 0, 0); // à 0 car c'est un signal

    pthread_create(&t1, NULL, threadAB, NULL);
    pthread_create(&t2, NULL, threadCD, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&s);
}

```

On a fait l'hypothèse en parlant des signaux qu'ils seraient attendus avant d'être produits, mais ce n'est pas nécessaire.

Que se passe-t-il si le thread 2 poste le sémaphore avant que le thread 1 ne l'attende ? Il va s'incrémenter et passer à 1, ainsi le thread 1 pourra l'acquiescer sans attente.

21.5 Problèmes classiques

21.5.1 Rendez-vous à deux threads

On considère un problème proche du problème du paragraphe précédent : deux threads veulent écrire AB mais tous les threads doivent avoir écrit A avant qu'un thread écrive B. Ainsi, la trace doit forcément être AABB.

Il est ainsi nécessaire que les deux threads s'attendent mutuellement. On va donc définir un signal *thread 0 a fini* et un signal *thread 1 a fini*.

Bien entendu, pour ne pas bloquer, il faut poster ces signaux avant d'attendre l'autre signal. On obtient donc le code suivant :

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

sem_t s[2];

void *threadAB(void *arg)
{
    int *i_arg = arg;
    int id = *i_arg;

    putchar('A');
    sem_post(&s[id]);
    sem_wait(&s[1 - id]);
    putchar('B');

    return NULL;
}

int main(void)
{
    pthread_t t[2];
    int ids[] = { 0, 1 };

    sem_init(&s[0], 0, 0); // à 0 car c'est un signal
    sem_init(&s[1], 0, 0); // à 0 car c'est un signal

    pthread_create(&t[0], NULL, threadAB, &ids[0]);
    pthread_create(&t[1], NULL, threadAB, &ids[1]);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);

    sem_destroy(&s[0]);
    sem_destroy(&s[1]);
}
```

21.5.2 Rendez-vous à n threads : barrière

On souhaite généraliser le mécanisme précédent à n threads **mais sans que la solution ne dépende du nombre de threads**.

Avec n threads, on doit donc avoir chaque thread qui affiche A puis attend **tous les autres threads** pour ensuite afficher B. Ainsi, la trace sera forcément A..AB..B.

Comme on dispose d'un nombre constant de primitives de synchronisation, les threads doivent partager un signal. La solution classique consiste, à la manière d'une chute de dominos, à faire en sorte que chaque thread réveille libère le suivant.

Celui qui lance la réaction en chaîne est le dernier thread ayant écrit un A. Pour le connaître, on va rajouter une incrémentation de compteur avec une section critique.

Remarque 21.27 Cela implique que les threads connaissent le nombre de threads dans la barrière.

On va considérer un sémaphore barrière et chaque thread va donc exécuter :

```
sem_wait(&barrier);
sem_post(&barrier);
```

Le dernier thread à finir va également réaliser un post de la barrière, la barrière est donc rendue $n + 1$ fois et acquise n fois, chaque fois dans l'ordre :

rendue $\rightarrow$ acquise $\rightarrow$ rendue $\rightarrow \dots \rightarrow$ rendue

On obtient ainsi le programme suivant :

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

int count = 0;

pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
sem_t barrier;

void *f(void *arg)
{
    putchar('A');

    pthread_mutex_lock(&count_lock);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);

    if (count == 10)
        sem_post(&barrier);

    sem_wait(&barrier);
    sem_post(&barrier);

    putchar('B');

    return NULL;
}

int main(void)
{
    pthread_t t[10];

    sem_init(&barrier, 0, 0);

    for(int i = 0; i < 10; i++)
        pthread_create(&t[i], NULL, f, NULL);

    for(int i = 0; i < 10; i++)
        pthread_join(t[i], NULL);

    sem_destroy(&barrier);
}
```

Note Faire le dessin de la réaction en chaîne.

Remarque 21.28 On peut s'étonner que la solution proposée accède à la valeur du compteur en dehors d'une section critique. En effet, il n'est pas inenvisageable d'avoir le comportement suivant :

- l'avant-dernier thread à prendre la section critique incrémente le compteur à $n-1$
- le dernier thread en section critique le passe à n
- les deux threads lisent en même temps la nouvelle valeur du compteur à n
- les deux threads postent le sémaphore

Cependant, on est assuré qu'on ne poste jamais le sémaphore avant qu'il y ait eu effectivement n exécutions de la section critique. Ainsi, le seul risque est de trop incrémenter le sémaphore, ce qui n'a aucune influence sur notre algorithme.

Notons toutefois qu'en C, cette lecture non protégée d'une variable modifiée par d'autres threads est une *data race*, dont le comportement n'est formellement pas défini. Il est plus sûr de lire `count` dans la section critique, par exemple en y mémorisant la valeur `count == 10` dans une variable locale testée ensuite.

21.5.3 Barrière réutilisable

On cherche maintenant à faire en sorte que la barrière soit réutilisable après une synchronisation complète. Le problème est lié à la remarque précédente : en l'absence d'accès au compteur en section critique, on ne connaît pas sa valeur à la fin et on ne peut pas le réinitialiser.

Pour bien comprendre le problème, on va modifier le code précédent :

```
void *f(void *arg)
{
    for(int i = 0; i < N; i++)
    {
        putchar('A');

        // ATTENTE D'ARRIVÉE

        putchar('B');

        // ATTENTE DE DÉPART
    }

    return NULL;
}
```

Ici, on veut donc effectuer plusieurs boucles AAABBBAAABBB... il y a donc deux barrières implicites avant et **après** l'affichage de B. On va ainsi considérer deux sémaphores et deux réactions en chaînes : `inbarrier` initialement à 0 et `outbarrier` initialement à 1.

Le code va devenir :

```
void *f(void *arg)
{
    for(int i = 0; i < N; i++) {
        putchar('A');

        pthread_mutex_lock(&count_lock);
        count = count + 1;
        if (count == 10) {
            sem_wait(&outbarrier);
            sem_post(&inbarrier);
        }
        pthread_mutex_unlock(&count_lock);

        // réaction en chaîne 1
        sem_wait(&inbarrier);
        sem_post(&inbarrier);

        putchar('B');

        pthread_mutex_lock(&count_lock);
        count = count - 1;
        if (count == 0) {
            sem_wait(&inbarrier);
            sem_post(&outbarrier);
        }
        pthread_mutex_unlock(&count_lock);

        // réaction en chaîne 2
        sem_wait(&outbarrier);
        sem_post(&outbarrier);
    }
}
```

```

    return NULL;
}

```

Note Faire un beau dessin ici.

Remarque 21.29 On a fait le choix ici de placer le `putchar('B')` entre les deux phases de la barrière. Cela a pour conséquence que tous les B seront affichés avant la nouvelle itération d'un thread.

Cependant, ce comportement qui est un bonus de notre implémentation peut être ignoré si on place `putchar('B')` après la seconde réaction en chaîne.

En faisant cela, on aura AAA-BBABAA-BABBAA... c'est-à-dire des séries de trois B et de trois A des itérations suivantes mélangés.

Encapsuler la barrière

On va définir un type `barrier_t` en C pour les barrières réutilisables. L'interface sera :

```

void barrier_init(barrier_t *b, int n);
void barrier_destroy(barrier_t *b);
void barrier_sync_in(barrier_t *b);
void barrier_sync_out(barrier_t *b);
void barrier_sync(barrier_t *b);

```

Les deux fonctions d'attente `sync_in` et `sync_out` permettent la granularité de la partie précédente pour séparer les deux phases de la synchronisation.

```

struct barrier {
    int n;
    int count;
    pthread_mutex_t count_lock;
    sem_t inbarrier;
    sem_t outbarrier;
};
typedef struct barrier barrier_t;

```

```

void barrier_init(barrier_t *b, int n)
{
    b->n = n;
    b->count = 0;
    pthread_mutex_init(&b->count_lock, NULL);
    sem_init(&b->inbarrier, 0, 0);
    sem_init(&b->outbarrier, 0, 1);
}

void barrier_destroy(barrier_t *b)
{
    pthread_mutex_destroy(&b->count_lock);
    sem_destroy(&b->inbarrier);
    sem_destroy(&b->outbarrier);
}

```

```

void barrier_sync_in(barrier_t *b)
{
    pthread_mutex_lock(&b->count_lock);
    b->count = b->count + 1;
    if (b->count == b->n) {
        sem_wait(&b->outbarrier);
        sem_post(&b->inbarrier);
    }
    pthread_mutex_unlock(&b->count_lock);

    sem_wait(&b->inbarrier);
    sem_post(&b->inbarrier);
}

```

```

void barrier_sync_out(barrier_t *b)
{
    pthread_mutex_lock(&b->count_lock);
    b->count = b->count - 1;
    if (b->count == 0) {
        sem_wait(&b->inbarrier);
        sem_post(&b->outbarrier);
    }
    pthread_mutex_unlock(&b->count_lock);

    sem_wait(&b->outbarrier);
    sem_post(&b->outbarrier);
}

```

```

void barrier_sync(barrier_t *b)
{
    barrier_sync_in(b);
    barrier_sync_out(b);
}

```

21.5.4 Producteurs-consommateurs

On considère maintenant le problème suivant :

- p threads producteurs vont produire des tâches à traiter
- c threads consommateurs vont traiter des tâches non traitées
- les tâches à traiter sont stockées dans un tableau de capacité N sous la forme d'une file de tâches
- si un producteur veut produire une tâche mais que la file est pleine, il doit attendre
- si un consommateur veut consommer une tâche mais que la file est vide, il doit attendre

La manière dont on a réalisé une file dans un tableau peut être simplifiée à l'aide de sémaphores : en effet, si on utilise des sémaphores pour gérer le nombre de places libres et le nombre de places occupées, il sera impossible de réaliser un ajout ou une suppression illicite.

Afin de mesurer des performances, les tâches seront des empreintes de temps telles qu'on peut en prendre avec la fonction `gettimeofday` de `sys/time.h`. En effet, cette fonction permet de remplir une `struct timeval` avec le temps actuel et une très bonne précision (à la microseconde).

```

struct timeval buffer[N];
int buffer_capacity = N;
int buffer_start = 0;
int buffer_end = 0;

void add_task(struct timeval task)
{
    buffer[buffer_end] = task;
    buffer_end = (buffer_end + 1) % N;
}

struct timeval take_task()
{
    struct timeval d = buffer[buffer_start];
    buffer_start = (buffer_start + 1) % N;
    return d;
}

```

Les fonctions qu'on aimerait protéger sont celles-ci :

```

void *produce(void *arg)
{
    while(true) {
        struct timeval tm;
        gettimeofday(&tm, NULL);
        add_task(tm);
    }
}

```

```

    }

    return NULL;
}

void *consume(void *arg)
{
    while(true) {
        struct timeval tm1 = take_task();
        struct timeval tm2;
        gettimeofday(&tm2, NULL);
        unsigned long long t = 1000000 * (tm2.tv_sec - tm1.tv_sec)
            + (tm2.tv_usec - tm1.tv_usec);

        printf("Task waited %llu us\n", t);
    }
}

```

Comme annoncé précédemment, on suppose que la file n'est jamais pleine (pour `add_task`) ou vide (pour `take_task`) dans ces fonctions, on pourra s'en assurer de manière indirecte par la synchronisation.

On va ainsi considérer deux sémaphores et un verrou :

- un sémaphore `items` indiquant le nombre d'éléments de la file, initialement à 0
- un sémaphore `spaces` indiquant le nombre de places libres dans la file, initialement à `N`
- un verrou `m` permettant d'assurer l'exclusion mutuelle dans l'accès à la file

Un producteur va ainsi :

- attendre une place libre avec une acquisition de `spaces`
- ajouter la tâche dans une section critique protégée par `m`
- incrémenter le nombre d'objets en signalant le sémaphore `items`

Cela correspond au code :

```

void *produce(void *arg)
{
    while(true) {
        struct timeval tm;
        gettimeofday(&tm, NULL);

        sem_wait(&spaces);
        pthread_mutex_lock(&m);
        add_task(tm);
        pthread_mutex_unlock(&m);
        sem_post(&items);
    }

    return NULL;
}

```

Un consommateur va faire l'opération inverse :

- attendre un objet avec une acquisition de `items`
- retirer la tâche dans une section critique protégée par `m`
- incrémenter le nombre de places libres en signalant le sémaphore `spaces`

```

void *consume(void *arg)
{
    while(true) {
        sem_wait(&items);
        pthread_mutex_lock(&m);
        struct timeval tm1 = take_task();
        pthread_mutex_unlock(&m);
        sem_post(&spaces);
    }
}

```



```

    struct timeval tm2;
    gettimeofday(&tm2, NULL);

    unsigned long long t = 1000000 * (tm2.tv_sec - tm1.tv_sec)
        + (tm2.tv_usec - tm1.tv_usec);

    printf("Task waited %llu us\n", t);
}
}

```

21.5.5 Le dîner des philosophes

On considère n philosophes assis autour d'une table ronde, chacun a une baguette à sa gauche et une baguette à sa droite. Les philosophes alternent entre penser et manger. Pour manger un philosophe a besoin de deux baguettes.

Le code d'un philosophe est donc :

```

while(true) {
    penser();
    attraper_baguette();
    manger();
    reposer_baguette();
}

```

On demande de trouver une solution pour la synchronisation de la prise des baguettes afin d'assurer :

- qu'au plus un philosophe tienne une baguette donnée
- qu'il n'y ait pas de famine
- que si $n \geq 4$, au moins deux philosophes puissent manger simultanément.

On va numéroter les philosophes de 0 à $n - 1$, et on numérote les baguettes par le numéro du philosophe à leur droite.

Le philosophe i doit donc attraper les baguettes i et $(i + 1) \bmod n$ pour manger.

Il va de soi qu'on ne peut pas mettre un verrou sur la table elle-même, car cela aurait pour effet de ne permettre qu'à un seul philosophe de manger. Il faut donc rajouter plus de granularité.

Ce problème est bien plus difficile qu'il n'en a l'air. On va voir différentes propositions mais aucune ne sera satisfaisante.

Premier essai

On va considérer ici un sémaphore par baguette qu'on notera `baguettes[i]`. On suppose que chaque sémaphore est initialisé à 1. On propose les programmes suivants :

```

void attraper_baguette(int i)
{
    sem_wait(&baguettes[i]);
    sem_wait(&baguettes[(i+1)%n]);
}

void reposer_baguette(int i)
{
    sem_post(&baguettes[i]);
    sem_post(&baguettes[(i+1)%n]);
}

```

- ✏ Ce code est simple mais va provoquer un interblocage si tous les philosophes attrapent leur première baguette (la baguette i) en même temps : chacun attend alors indéfiniment la baguette tenue par son voisin.

États des philosophes

On considère maintenant que les philosophes ont trois états : en train de penser, en train de manger, ayant faim. On a ainsi rajouté un état intermédiaire qui est un état d'attente.

Initialement, chaque philosophe est en train de penser.

On considère également n sémaphores signalant que le i ème philosophe mange, ils sont initialisés à 0, car aucun ne mange au départ.

Pour manipuler les états, on rajoute un verrou m .

```
int PENSER = 0;
int MANGER = 1;
int FAIM = 2;

int etats[n] = {};
sem_t mange[n];
pthread_mutex_t m;
```

On va définir une fonction permettant de laisser le philosophe i manger s'il le désire. Cette fonction permettra à la fois d'être utilisée par le philosophe désirant manger et par un philosophe ayant fini de manger et voulant le signaler aux philosophes à sa gauche et à sa droite.

```
void test(int i)
{
    if(etats[i] == FAIM && etats[gauche(i)] != MANGER
        && etats[droite(i)] != MANGER) {
        etats[i] = MANGER;
        sem_post(&mange[i]);
    }
}
```

Pour attraper les baguettes, un philosophe va changer son état en section critique en déclarant avoir faim, prendre les baguettes avec le test si c'est possible et attendre d'être effectivement en train de manger avec son sémaphore :

```
void attraper_baguettes(int i)
{
    pthread_mutex_lock(&m);
    etats[i] = FAIM;
    test(i);
    pthread_mutex_unlock(&m);
    sem_wait(&mange[i]);
}
```

Pour reposer les baguettes, on modifie l'état du philosophe en section critique et on débloque les philosophes à gauche et à droite.

```
void reposer_baguettes(int i)
{
    pthread_mutex_lock(&m);
    etats[i] = PENSER;
    test(gauche(i));
    test(droite(i));
    pthread_mutex_unlock(&m);
}
```

Ainsi, pour manger, un philosophe va

- soit directement prendre les baguettes et s'auto-signaliser, s'il a faim et que personne ne mange à sa gauche et à sa droite
- soit attendre son sémaphore, ce qui implique qu'un philosophe à gauche ou à droite mange, attendre qu'il finisse de manger et qu'on lui signale qu'il peut manger.

Ici, on considère qu'un philosophe tient ses deux baguettes exactement quand son état est MANGER.

Théorème 21.30

À tout moment, au plus un philosophe tient une baguette donnée et il n'y a pas de situation d'interblocage.

Preuve Les états ne sont modifiés que lorsque le verrou m est pris, on peut donc raisonner sur des invariants vérifiés à chaque libération du verrou.

- *Invariant 1* : deux philosophes voisins ne sont jamais tous les deux dans l'état MANGER. C'est vrai initialement, et un philosophe ne passe dans l'état MANGER que dans `test`, après avoir vérifié que ses deux voisins ne sont pas dans cet état. Comme une baguette n'est partagée que par deux voisins, au plus un philosophe la tient.
- *Invariant 2* : si un philosophe est dans l'état FAIM, un de ses voisins est dans l'état MANGER. En effet, quand un philosophe passe dans l'état FAIM, il appelle immédiatement `test` : il ne reste dans l'état FAIM que si un voisin mange. Et quand un philosophe cesse de manger, il appelle `test` sur ses deux voisins : un voisin affamé passe alors dans l'état MANGER sauf si son autre voisin mange.
- Supposons qu'il y ait interblocage. L'attente du verrou ne peut pas durer indéfiniment car les sections critiques sont courtes et le verrou ne présente pas d'interblocage. Un philosophe bloqué indéfiniment l'est donc sur `sem_wait(&mange[i])` ; il n'est pas dans l'état MANGER, sinon son sémaphore aurait été incrémenté, donc il est dans l'état FAIM. Par l'invariant 2, un de ses voisins mange, et comme les philosophes ne mangent pas indéfiniment, celui-ci finit par appeler `reposer_baguettes` : le système n'est donc pas bloqué.

□

Malheureusement, cette solution n'est pas parfaite car il y a bien une possibilité de famine.

On considère 5 philosophes et on va affamer le philosophe 0.

- au départ les philosophes 2 et 4 commencent à manger, puis le philosophe 0 a faim
- puis les philosophes 1 et 3 ont faim
- le philosophe 2 finit de manger, ce qui libère 1, puis le philosophe 4 finit de manger, ce qui libère 3
- puis les philosophes 2 et 4 ont faim
- le philosophe 3 finit de manger, ce qui libère 4, puis le philosophe 1 finit de manger, ce qui libère 2
- et on répète indéfiniment

Avec cet ordre, à tout instant l'un des voisins 1 ou 4 du philosophe 0 est en train de manger : chaque appel à `test(0)` échoue et on affame le philosophe 0 qui ne pourra jamais manger.

Threads, Peterson, mutex et sémaphores

DANS CE CHAPITRE

- 22.1 Type universel de pointeurs
- 22.2 Création de threads
- 22.3 Algorithme de Peterson
- 22.4 Sémaphores
- 22.5 Problèmes

Ce TP accompagne le chapitre [Concurrence et parallélisme](#).

Remarque 22.1 On compile les programmes de ce TP avec l'option `-pthread` :

```
gcc -Wall -Wextra -g -pthread -o prog prog.c
```

Si vous avez un problème à l'édition de liens, il faut rajouter `-lpthread`. L'option `-fsanitize=thread` (à la place de `-fsanitize=address`) détecte à l'exécution les accès concurrents non protégés à une même variable (*data races*).

Attention, sous macOS, les sémaphores anonymes de `semaphore.h` ne sont pas implémentés : `sem_init` renvoie `-1` et les programmes de la seconde partie ne fonctionnent pas. Il faut alors travailler sous Linux (machine virtuelle, serveur de l'établissement...).

22.1 Type universel de pointeurs

En C, le type `void *` joue un rôle particulier :

- toute valeur de type τ `*a` peut être convertie en un `void *` ;
- toute valeur de type `void *v` peut être convertie en un τ `*`.

Ces conversions sont **implicites** en C, il n'y a rien à écrire.

Ainsi, le code suivant est correct :

```
#include <stdio.h>

void incremente(void *a)
{
    int *t = a;
    *t = *t + 1;
}

int main(void)
{
    int n = 3;
```

```

    incremente(&n);
    printf("n = %d\n", n);
}

```

Ainsi, il est possible de faire des fonctions génériques en utilisant le type `void *` comme une sorte de 'a en OCaml. Mais contrairement à OCaml, aucune vérification n'est faite : c'est au programmeur de savoir vers quel type pointe réellement un `void *`.

Remarque 22.2 Comme le type `int` est, sur les architectures usuelles, plus petit que le type `void *` en termes de taille, il est possible de faire des conversions explicites pour ne pas avoir à définir une variable comme `n` ici.

22.2 Création de threads

Pour créer des threads, on utilise la bibliothèque `pthread` dont l'en-tête est `pthread.h`.

Pour créer un thread :

- on définit une valeur `pthread_t t` qui va contenir les données du thread ;
- on appelle `pthread_create(&t, NULL, f, arg)`; où `f` est une fonction de prototype `void *f(void *arg)`; et `arg` est une valeur de type `void *`.

Par exemple, on définit la fonction `incremente` qui va prendre un entier en argument par pointeur et l'incrémenter.

```

void *incremente(void *arg)
{
    int *n = arg;
    *n = *n + 1;

    return NULL; // pas de retour
}

```

On peut alors lancer cette fonction dans un thread :

```

pthread_t t;
int v = 3;
pthread_create(&t, NULL, incremente, &v);

```

Pour attendre qu'un thread `t` termine, on appelle `pthread_join(t, NULL)` :

```

pthread_join(t, NULL);
printf("%d\n", v); // affiche 4

```

Remarque 22.3 Il est également possible d'avoir une valeur de retour mais c'est hors programme.

22.3 Algorithme de Peterson

On considère le programme à deux threads suivant :

```

#include <pthread.h>
#include <stdio.h>

int compteur = 0;
int N = 100000;

void *incremente(void *arg)
{
    int *id = arg; // le numéro du thread, inutilisé pour l'instant
    (void)id;
    for(int i = 0; i < N; i++) {
        compteur = compteur + 1;
    }
}

```

```

    }
    return NULL;
}

int main(void)
{
    pthread_t t0, t1;
    int i0 = 0;
    int i1 = 1;
    pthread_create(&t0, NULL, incremente, &i0);
    pthread_create(&t1, NULL, incremente, &i1);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
    printf("compteur = %d (attendu %d)\n",
           compteur, 2 * N);
}

```

Question 1

Observez le comportement en changeant la valeur de N .

Corrigé

Pour de petites valeurs de N , on obtient souvent $2 * N$: le premier thread a fini avant que le second ne démarre. Pour de grandes valeurs, le résultat est inférieur à $2 * N$ et varie d'une exécution à l'autre : des incrémentations sont perdues quand les deux threads lisent la même valeur de `compteur` avant d'écrire (voir [Exemple de section critique](#)).

Avec l'option `-O2`, on peut même obtenir exactement N : le compilateur remplace la boucle par une unique instruction `compteur = compteur + N`, et les deux threads lisent 0 en même temps. En C, ces accès concurrents non protégés sont des *data races*, dont le comportement n'est pas défini, et `-fsanitize=thread` les signale.

On va essayer de réaliser des fonctions de prototypes :

```

void verrouille(int id);
void deverrouille(int id);

```

où `id` est le numéro du thread, 0 ou 1 ici, et telles qu'entre les appels à `verrouille(id)` et `deverrouille(id)`, on puisse placer une section critique et garantir :

- l'exclusion mutuelle : un seul thread peut exécuter la section critique à un moment donné ;
- l'absence d'interblocage : si un thread effectue un appel à `verrouille`, alors au moins un thread, peut-être un autre, va rentrer en section critique ensuite ;
- l'absence de famine : si un thread effectue un appel à `verrouille`, il ne va pas attendre indéfiniment.

Remarque 22.4 L'absence de famine implique l'absence d'interblocage (voir [Propriétés d'un verrou](#)).

On va alors, dans l'exemple précédent, remplacer `incremente` par :

```

void *incremente(void *arg)
{
    int *id = arg; // le numéro du thread
    for(int i = 0; i < N; i++) {
        verrouille(*id);
        compteur = compteur + 1;
        deverrouille(*id);
    }
    return NULL;
}

```

22.3.1 Premier essai

On propose le pseudo-code suivant :

```
flag[0] = false
flag[1] = false

verrouille(i):
    flag[i] = true
    tant que flag[j] == true où j est le numéro de l'autre thread
        attendre

deverrouille(i):
    flag[i] = false
```

Question 2

Implémentez et testez cet algorithme.

Que se passe-t-il ? Est-ce prévisible ?

Corrigé

```
bool flags[2] = { false, false };

void verrouille(int id)
{
    flags[id] = true;
    while(flags[1-id]) { // attente active
    }
}

void deverrouille(int id)
{
    flags[id] = false;
}
```

Le programme se bloque : les deux threads finissent par lever leur drapeau en même temps, puis chacun attend indéfiniment que l'autre baisse le sien. C'est un interblocage, prévisible : voir [Première idée](#).

Question 3

Montrez que ce verrou satisfait l'exclusion mutuelle.

Corrigé

Voir la preuve du cours dans [Première idée](#) : si les deux threads étaient en section critique, chacun aurait écrit son drapeau avant que l'autre n'écrive le sien, ce qui est impossible.

22.3.2 Second essai

On propose maintenant le pseudo-code suivant :

```
victime = 0

verrouille(i):
    victime = i
    tant que victime == i
        attendre

deverrouille(i):
    ne fait rien
```

Question 4

Implémentez et testez cet algorithme.

Que se passe-t-il ? Est-ce prévisible ?

Corrigé

```
int victime = 0;

void verrouille(int id)
{
    victime = id;
    while(victime == id) { // attente active
    }
}

void deverrouille(int id)
{
}
```

Un thread n'est débloquent que lorsque l'autre verrouille à son tour. Dès qu'un thread a fini ses N incrémentations, l'autre reste bloqué indéfiniment : le programme ne termine pas. C'est prévisible, voir [Deuxième idée](#).

Question 5

Montrez que ce verrou satisfait l'exclusion mutuelle.

Corrigé

Voir la preuve dans [Deuxième idée](#) : le dernier thread à avoir écrit dans `victime` ne peut pas en être sorti.

22.3.3 Algorithme de Peterson

On va combiner les deux mauvaises idées précédentes pour en déduire une bonne idée.

On va utiliser `flags` et `victime`, mais cette fois, on attend tant que `flags[j] == true` et `victime == i`.

Question 6

Implémentez et testez cet algorithme.

Que se passe-t-il ? Est-ce prévisible ?

Corrigé

```
bool flags[2] = { false, false };
int victime = 0;

void verrouille(int id)
{
    int autre = 1 - id;
    flags[id] = true;
    victime = id;
    while(flags[autre] && victime == id) { // attente active
    }
}

void deverrouille(int id)
{
    flags[id] = false;
}
```

Le programme termine, mais le compteur peut être légèrement inférieur à $2 * N$ (par exemple 199999), alors que l'algorithme est prouvé correct : les hypothèses du **modèle mémoire** ne sont pas satisfaites par les processeurs modernes, qui peuvent réordonner l'écriture de `flags[id]` et la lecture de `flags[autre]`. Avec -O2, le programme peut même se bloquer : le compilateur, qui ne sait pas que d'autres threads modifient ces variables, lit `flags[autre]` et victime une seule fois avant la boucle.

Remarque 22.5 Il est possible que ce verrou marche **presque** en pratique ; la raison est liée aux optimisations de la gestion de la mémoire, où les écritures ne sont pas réalisées au bon moment.

Sur un processeur x86, on peut rajouter l'instruction :

```
asm volatile ("mfence" ::: "memory");
```

avant le `while` pour forcer l'écriture avant la lecture. Cette instruction n'existe pas sur d'autres processeurs (ARM, comme les Mac récents) ; on peut utiliser à la place la fonction `__sync_synchronize()` ; de `gcc` et `clang`, qui place la barrière mémoire adaptée au processeur.

Normalement, ça suffit ici à réparer le verrou sans optimisations (-O0). Pour un programme correct au sens de la norme, il faut déclarer les variables partagées comme atomiques avec `stdatomic.h` (`atomic_bool flags[2]` ; et `atomic_int victime`) : leurs accès sont alors réalisés dans l'ordre du programme et ne sont plus optimisés. C'est hors programme.

Question 7

Prouvez que ce verrou satisfait toutes les propriétés voulues.

Corrigé

Voir les preuves de l'exclusion mutuelle et de l'absence de famine dans **Algorithme de Peterson** ; l'absence d'interblocage découle de l'absence de famine.

22.3.4 Mutex

Il y a dans `pthread` un mécanisme de verrou fiable : les mutex.

On déclare un mutex `m` avec une variable globale :

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

On peut également l'initialiser avec

```
pthread_mutex_init(&m, NULL);
```

On peut alors verrouiller et déverrouiller `m` avec :

```
pthread_mutex_lock(&m); // verrouille
pthread_mutex_unlock(&m); // déverrouille
```

Même si ici cela ne va rien faire, c'est une bonne pratique d'appeler

```
pthread_mutex_destroy(&m);
```

quand on a fini d'utiliser le mutex.

Question 8

Reprenez le code d'incrémentation avec un mutex.

Corrigé

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *incremente(void *arg)
{
    (void)arg;
    for(int i = 0; i < N; i++) {
        pthread_mutex_lock(&m);
        compteur = compteur + 1;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}
```

On obtient toujours $2 * N$, y compris avec `-O2`, et `-fsanitize=thread` ne signale plus rien. On appelle `pthread_mutex_destroy(&m)`; à la fin de main, après les join.

22.3.5 Incrémentation atomique

Une autre manière de réparer ce programme est d'utiliser l'instruction atomique non portable suivante :

```
__sync_add_and_fetch(&compteur, 1);
```

Ici, on va réaliser l'incrémentation d'un seul coup sans avoir de possibilité d'accès concurrent (voir [Instructions atomiques](#)).

Question 9

Vérifiez que cela fonctionne.

Corrigé

Il suffit de remplacer, dans la boucle de `incremente`, l'instruction `compteur = compteur + 1`; (et les appels de verrouillage) par `__sync_add_and_fetch(&compteur, 1)`. On obtient bien $2 * N$. C'est l'instruction *fetch and add* du processeur ; elle est plus rapide qu'un mutex mais ne protège qu'une seule opération sur une seule variable.

22.4 Sémaphores

pthread définit également des sémaphores, dans l'en-tête `semaphore.h`. Ce sont des compteurs de ressources qui permettent :

- d'être initialisés avec un nombre de ressources ;
- d'être acquis en attendant que le compteur soit > 0 et en le décrémentant ;
- d'être rendus en incrémentant le compteur.

Ce mécanisme permet de transmettre des signaux, puisque un thread peut attendre un sémaphore qui est nul et qui va passer à 1 quand un autre thread veut lui envoyer un signal.

Avec pthread, on utilise :

```
sem_t s; // définit un sémaphore

sem_init(&s, 0, v); // l'initialise avec la valeur
                  // initiale v

sem_wait(&s); // attend pour décrémenter (acquisition)

sem_post(&s); // incrémente (rendu)

sem_destroy(&s); // quand on a fini de l'utiliser
```

22.4.1 Rendez-vous

Question 10

Réalisez deux threads effectuant chacun l’affichage AB de deux caractères, mais en faisant en sorte que la trace soit toujours AABB.

Indication On pourra utiliser deux sémaphores initialement à 0, intitulés, c’est un indice, `t_est_la[0]` et `t_est_la[1]`.

Corrigé

Chaque thread signale qu’il est arrivé, puis attend l’autre (voir aussi [Rendez-vous à deux threads](#)).

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t t_est_la[2];

void *f(void *arg)
{
    int *id = arg;
    putchar('A');

    sem_post(&t_est_la[*id]); // je signale que je suis là
    sem_wait(&t_est_la[1 - *id]); // j'attends l'autre

    putchar('B');
    return NULL;
}

int main(void)
{
    pthread_t t0, t1;
    int i0 = 0;
    int i1 = 1;
    sem_init(&t_est_la[0], 0, 0);
    sem_init(&t_est_la[1], 0, 0);
    pthread_create(&t0, NULL, f, &i0);
    pthread_create(&t1, NULL, f, &i1);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
    putchar('\n');
    sem_destroy(&t_est_la[0]);
    sem_destroy(&t_est_la[1]);
    return 0;
}
```

Il faut faire le `sem_post` **avant** le `sem_wait` : dans l’autre ordre, les deux threads s’attendraient mutuellement (interblocage).

22.4.2 Rendez-vous généralisé : la barrière

Question 11

Même question que la précédente mais pour 10 threads.

Il ne faut pas que le nombre de primitives de synchronisation dépende du nombre de threads.

Indication Une variable compte, un mutex pour la gérer et un sémaphore nul pour signaler qu’on peut passer à la suite.

Corrigé

Le dernier thread arrivé ouvre la barrière, puis chaque thread qui passe laisse passer le suivant (voir Rendez-vous à n threads : barrière).

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdbool.h>

#define NB_THREADS 10

int compte = 0;
pthread_mutex_t verrou_compte = PTHREAD_MUTEX_INITIALIZER;
sem_t barriere; // initialement à 0

void *f(void *arg)
{
    (void)arg;
    putchar('A');

    pthread_mutex_lock(&verrou_compte);
    compte = compte + 1;
    bool dernier = (compte == NB_THREADS); // lu dans la section critique
    pthread_mutex_unlock(&verrou_compte);

    if (dernier)
        sem_post(&barriere);

    // tourniquet : chaque thread qui passe laisse passer le suivant
    sem_wait(&barriere);
    sem_post(&barriere);

    putchar('B');
    return NULL;
}

int main(void)
{
    pthread_t t[NB_THREADS];
    sem_init(&barriere, 0, 0);
    for(int i = 0; i < NB_THREADS; i++)
        pthread_create(&t[i], NULL, f, NULL);
    for(int i = 0; i < NB_THREADS; i++)
        pthread_join(t[i], NULL);
    putchar('\n');
    sem_destroy(&barriere);
    return 0;
}
```

On teste `compte == NB_THREADS` dans la section critique : lire `compte` après l'avoir déverrouillé serait une *data race*.

22.4.3 Barrière réutilisable

Question 12

On souhaite créer un type barrière et des fonctions :

```
// initialise la barrière pour synchroniser n threads
void init_barriere(barriere *b, int n);

// attend que les n threads aient atteint la barrière
void attends_barriere(barriere *b);
```

Contrairement au cas précédent, on souhaite que la barrière soit réutilisable.

Il ne faut pas que le nombre de primitives de synchronisation dépende du nombre de threads.

Corrigé

Avec un seul tourniquet, un thread rapide pourrait repasser la barrière avant que les autres ne l'aient franchie. On utilise donc deux tourniquets, l'un fermé quand l'autre est ouvert, comme dans [Barrière réutilisable](#).

```
struct barriere {
    int n;                // nombre de threads à synchroniser
    int compte;           // nombre de threads arrivés
    pthread_mutex_t verrou;
    sem_t entree;         // tourniquet d'entrée, initialement fermé (0)
    sem_t sortie;         // tourniquet de sortie, initialement ouvert (1)
};
typedef struct barriere barriere;

void init_barriere(barriere *b, int n)
{
    b->n = n;
    b->compte = 0;
    pthread_mutex_init(&b->verrou, NULL);
    sem_init(&b->entree, 0, 0);
    sem_init(&b->sortie, 0, 1);
}

void detruit_barriere(barriere *b)
{
    pthread_mutex_destroy(&b->verrou);
    sem_destroy(&b->entree);
    sem_destroy(&b->sortie);
}

void attends_barriere(barriere *b)
{
    // phase 1 : on attend que les n threads soient arrivés
    pthread_mutex_lock(&b->verrou);
    b->compte = b->compte + 1;
    if (b->compte == b->n) {
        sem_wait(&b->sortie); // on ferme la sortie
        sem_post(&b->entree); // on ouvre l'entrée
    }
    pthread_mutex_unlock(&b->verrou);
    sem_wait(&b->entree);
    sem_post(&b->entree);

    // phase 2 : on attend que les n threads soient passés
    pthread_mutex_lock(&b->verrou);
    b->compte = b->compte - 1;
    if (b->compte == 0) {
        sem_wait(&b->entree); // on referme l'entrée
        sem_post(&b->sortie); // on ouvre la sortie
    }
    pthread_mutex_unlock(&b->verrou);
    sem_wait(&b->sortie);
    sem_post(&b->sortie);
}
```

Avec 10 threads exécutant 5 fois `putchar('A');` `attends_barriere(&b);` `putchar('B');` `attends_barriere(&b);`, la trace est toujours formée de 10 A, 10 B, 10 A, etc.

22.5 Problèmes

Dans tous les problèmes suivants, on demande d'implémenter une solution qui se comporte bien.

22.5.1 Producteurs-consommateurs

On considère des tâches qui sont gérées dans un tampon de n cases.

On a :

- p threads producteurs qui veulent écrire une nouvelle tâche dans le tampon, quitte à attendre s'il est plein ;
- c threads consommateurs qui veulent récupérer une tâche et libérer la place dans le tampon, quitte à attendre s'il est vide.

Corrigé

On gère le tampon comme une file circulaire, avec un sémaphore comptant les places libres, un autre comptant les tâches présentes et un mutex protégeant les indices (voir [Producteurs-consommateurs](#)).

```
#define N 4           // taille du tampon

int tampon[N];
int debut = 0;        // indice de la prochaine tâche à retirer
int fin = 0;          // indice de la prochaine case libre

sem_t places;         // places libres, initialisé à N
sem_t taches;         // tâches présentes, initialisé à 0
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void depose(int tache) // appelé par les producteurs
{
    sem_wait(&places);
    pthread_mutex_lock(&m);
    tampon[fin] = tache;
    fin = (fin + 1) % N;
    pthread_mutex_unlock(&m);
    sem_post(&taches);
}

int retire(void)       // appelé par les consommateurs
{
    sem_wait(&taches);
    pthread_mutex_lock(&m);
    int tache = tampon[debut];
    debut = (debut + 1) % N;
    pthread_mutex_unlock(&m);
    sem_post(&places);
    return tache;
}
```

L'ordre des opérations compte : si un producteur prenait le mutex avant d'attendre une place libre, il bloquerait les consommateurs qui pourraient la libérer.

22.5.2 Lecteurs-écrivains

On considère maintenant une ressource comme une base de données dont l'accès est une section critique, et deux types de threads :

- autant de lecteurs qu'on le souhaite peuvent être dans la section critique ;
- les écrivains ont un accès exclusif à la section critique (ce qui bloque également les lecteurs).

Corrigé

Un sémaphore `salle_vides`, initialisé à 1, est pris par chaque écrivain et par le **premier** lecteur qui entre ; il est rendu par chaque écrivain et par le **dernier** lecteur qui sort. Le nombre de lecteurs présents est protégé par un mutex.

```
int nb_lecteurs = 0; // lecteurs en section critique
pthread_mutex_t verrou_lecteurs = PTHREAD_MUTEX_INITIALIZER;
```

```

sem_t salle_vide;      // initialisé à 1

void debut_lecture(void)
{
    pthread_mutex_lock(&verrou_lecteurs);
    nb_lecteurs = nb_lecteurs + 1;
    if (nb_lecteurs == 1) // le premier lecteur ferme la salle aux écrivains
        sem_wait(&salle_vide);
    pthread_mutex_unlock(&verrou_lecteurs);
}

void fin_lecture(void)
{
    pthread_mutex_lock(&verrou_lecteurs);
    nb_lecteurs = nb_lecteurs - 1;
    if (nb_lecteurs == 0) // le dernier lecteur la rouvre
        sem_post(&salle_vide);
    pthread_mutex_unlock(&verrou_lecteurs);
}

void debut_ecriture(void)
{
    sem_wait(&salle_vide);
}

void fin_ecriture(void)
{
    sem_post(&salle_vide);
}

```

Cette solution présente un risque de famine pour les écrivains : si des lecteurs se relaient sans cesse, `nb_lecteurs` ne revient jamais à 0. On peut y remédier en ajoutant un sémaphore « tourniquet » que les écrivains ferment en arrivant et que chaque lecteur doit franchir avant `debut_lecture`.

22.5.3 Dîner des philosophes

n philosophes sont autour d'une table avec une baguette entre deux d'entre eux. Ils ont besoin de deux baguettes pour manger.

On cherche une solution garantissant les bonnes propriétés et maximisant le parallélisme, c'est-à-dire le nombre de philosophes mangeant en même temps.

Corrigé

Voir l'étude détaillée dans [Le dîner des philosophes](#) : la solution naïve (un sémaphore par baguette) peut provoquer un interblocage, et la solution par états évite l'interblocage tout en permettant à des philosophes non voisins de manger simultanément, mais présente un risque de famine.

22.5.4 Dîner convivial

n personnes sont autour d'une marmite pouvant servir m rations avant d'être vide. Quand elle est vide, elles vont réveiller le cuisinier pour qu'il remplisse la marmite et elles attendent que la marmite soit remplie pour se resservir.

Le cuisinier ne doit remplir la marmite que lorsqu'elle est vide.



Corrigé

Le nombre de rations est protégé par un `mutex`. Le convive qui trouve la marmite vide réveille le cuisinier avec un premier sémaphore, puis attend, **en gardant le verrou**, le signal du second sémaphore : les autres convives attendent ainsi sur le verrou et le cuisinier n'est réveillé qu'une fois par marmite vide.


```

#define M 5                // rations par marmite

int rations = 0;           // la marmite est vide au départ
pthread_mutex_t verrou_marmite = PTHREAD_MUTEX_INITIALIZER;
sem_t marmite_vide;        // signal vers le cuisinier, initialisé à 0
sem_t marmite_pleine;      // signal vers le convive, initialisé à 0

void se_servir(void)       // appelé par les convives avant de manger
{
    pthread_mutex_lock(&verrou_marmite);
    if (rations == 0) {
        sem_post(&marmite_vide); // on réveille le cuisinier
        sem_wait(&marmite_pleine); // et on attend qu'il ait rempli
    }
    rations = rations - 1;
    pthread_mutex_unlock(&verrou_marmite);
}

void *cuisinier(void *arg)
{
    (void)arg;
    while (true) {
        sem_wait(&marmite_vide);
        rations = M;           // la marmite est vide ici
        sem_post(&marmite_pleine);
    }
    return NULL;
}

```

Le cuisinier modifie rations sans prendre le verrou, mais sans concurrence : le seul convive qui pourrait y accéder attend marmite_pleine et les autres attendent le verrou qu'il détient. Les sémaphores assurent que l'écriture du cuisinier est visible du convive.

PARTIE VI

Intelligence artificielle

23	Intelligence artificielle	261
24	TP KNN - Identifier des chiffres	275
25	TP Classification en basse dimension	279
26	TP Fonctions booléennes et ID3	287
27	TP Morpion	291
28	TP Othorro	299
29	TP Résolution du 15-puzzle	309

Intelligence artificielle

DANS CE CHAPITRE

- 23.1 Apprentissage supervisé
- 23.2 Apprentissage non-supervisé
- 23.3 Recherche informée



Remarque 23.1 On parle d'**apprentissage automatique** quand un programme construit lui-même, à partir d'exemples, une fonction qu'on ne sait pas écrire directement : reconnaître un chiffre manuscrit, prédire si un message est indésirable, regrouper des clients aux comportements proches... On présente ici les algorithmes au programme. Les codes sont en Python sans bibliothèque, ou en OCaml avec les mêmes noms de fonctions que dans les TP **KNN**, **KNN avec arbres k-d** et **ID3**, qui mettent ces algorithmes en œuvre sur des données réelles.

23.1 Apprentissage supervisé



23.1.1 Classification

On dispose d'un ensemble X d'objets, souvent $X = \mathbb{R}^d$ où chaque coordonnée est un **attribut** mesuré sur l'objet, et d'un ensemble fini Y de **classes** (ou étiquettes). On suppose qu'il existe une fonction inconnue $f : X \rightarrow Y$ qu'on cherche à approcher.

Définition 23.2

Un **jeu de données étiqueté** est une suite finie $((x_1, y_1), \dots, (x_n, y_n))$ d'éléments de $X \times Y$, où y_i est la classe de x_i .

Un algorithme d'**apprentissage supervisé** construit à partir d'un tel jeu de données un **classifieur** $\hat{f} : X \rightarrow Y$ qui doit prédire la classe d'objets qui ne sont pas dans le jeu de données.

Exemple 23.3

- X est l'ensemble des images de 28×28 pixels en niveaux de gris, vues comme des éléments de $\mathbb{R}^{784}$, et $Y = \llbracket 0, 9 \rrbracket$: on veut reconnaître des chiffres manuscrits.
- X est l'ensemble des courriels et $Y = \{\text{indésirable}, \text{légitime}\}$.

Pour évaluer un classifieur, il ne faut pas le tester sur les données qui ont servi à le construire : un classifieur qui apprend par cœur le jeu de données serait parfait sur celui-ci et pourrait être très mauvais ailleurs. On sépare donc les données en un **jeu d'entraînement**, qui sert à construire $\hat{f}$, et un **jeu de test**, qui sert à l'évaluer.

Dans les codes qui suivent, on utilise des points de $\mathbb{R}^2$ répartis autour de trois centres, la classe d'un point étant l'indice de son centre.

Python

```
import random

def genere(n, centres, sigma):
    """n points autour de chaque centre, étiquetés par l'indice du centre"""
    donnees = []
    for c, (cx, cy) in enumerate(centres):
        for _ in range(n):
            x = random.gauss(cx, sigma)
            y = random.gauss(cy, sigma)
            donnees.append((x, y), c)
    random.shuffle(donnees)
    return donnees
```

23.1.2 Algorithme des k plus proches voisins

L'idée est très simple : deux objets proches ont sans doute la même classe. On munit $X = \mathbb{R}^d$ de la distance euclidienne

$$d(x, y) = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$$

Définition 23.4

Soit $k \in \mathbb{N}^*$. Le classifieur des **k plus proches voisins** associé au jeu de données $((x_1, y_1), \dots, (x_n, y_n))$ associe à $x \in X$ la classe majoritaire parmi les classes des k points x_i les plus proches de x .

Il n'y a ici aucune phase d'entraînement : tout le travail est fait au moment de la prédiction. En cas d'égalité des votes, il faut fixer une règle ; ici, on garde la plus petite classe.

Comme dans le **TP KNN avec arbres k-d**, un vecteur est un tableau de flottants, les données d'entraînement sont un tableau de couples (vecteur, classe) et la distance est passée en paramètre.

OCaml

```
type vector = float array

let distance_2 v1 v2 =
  let s = ref 0. in
  for i = 0 to Array.length v1 - 1 do
    s := !s +. (v1.(i) -. v2.(i)) *. (v1.(i) -. v2.(i))
  done;
  sqrt !s

(* élément le plus fréquent de l, le plus petit en cas d'égalité *)
let majority l =
  let occ = Hashtbl.create 16 in
  List.iter (fun c ->
    let o = try Hashtbl.find occ c with Not_found -> 0 in
    Hashtbl.replace occ c (o + 1)) l;
  let best = ref (List.hd l) in
  Hashtbl.iter (fun c o ->
    let ob = Hashtbl.find occ !best in
    if o > ob || (o = ob && c < !best) then best := c) occ;
  !best

(* les k plus proches voisins de v dans training *)
let k_nearest dist k training v =
  let a = Array.copy training in
  Array.sort (fun (v1, _) (v2, _) -> compare (dist v1 v) (dist v2 v)) a;
  Array.to_list (Array.sub a 0 (min k (Array.length a)))
```

```
let nearest_neighbors dist k training v =
  majority (List.map snd (k_nearest dist k training v))
```

Python

```
from math import sqrt

def distance_2(x, y):
    return sqrt(sum((a - b) ** 2 for a, b in zip(x, y)))

def majority(classes):
    # en cas d'égalité, on renvoie la plus petite classe
    return min(set(classes), key=lambda c: (-classes.count(c), c))

def nearest_neighbors(dist, k, training, x):
    """training est une liste de couples (point, classe)"""
    voisins = sorted(training, key=lambda d: dist(d[0], x))[:k]
    return majority([c for _, c in voisins])
```

Une prédiction coûte $O(nd)$ pour calculer les distances et $O(n \log_2 n)$ pour le tri. On pourrait se contenter de sélectionner les k plus petites distances, par exemple avec un tas de taille k , en $O(n \log_2 k)$.

Remarque 23.5

- Le choix de k est important. Avec $k = 1$, le classifieur est très sensible aux points mal étiquetés ou atypiques. Avec $k = n$, il renvoie toujours la classe majoritaire du jeu de données. On choisit k en comparant les résultats sur un jeu de test. Avec deux classes, un k impair évite les égalités de votes.
- La distance euclidienne suppose que les attributs sont comparables. Si un attribut est une taille en mètres et un autre un revenu en euros, le second écrase le premier. On commence alors par **normaliser** les attributs, par exemple en les ramenant dans $[0, 1]$: si m et M sont les valeurs minimale et maximale d'un attribut, on remplace chaque valeur v par $\frac{v-m}{M-m}$. C'est ce qui est fait dans le **TP KNN avec arbres k-d**, où l'on compare des statistiques de pokémons de natures très différentes.

23.1.3 Arbres k-dimensionnels

Quand le jeu d'entraînement est grand et la dimension d petite, on peut éviter de calculer les n distances à chaque prédiction en rangeant les points dans un **arbre k-d** (arbre k -dimensionnel ; ici le k désigne la dimension, notée d , et n'a rien à voir avec le nombre de voisins). C'est l'objet du **TP KNN avec arbres k-d**, dont on reprend les notations.

Définition 23.6

Un **arbre k-d** sur des points de $\mathbb{R}^d$ est un arbre binaire dont chaque nœud contient un point w et une dimension $\dim \in \llbracket 0, d-1 \rrbracket$, tel que tous les points x du sous-arbre gauche vérifient $x_{\dim} \leq w_{\dim}$ et tous ceux du sous-arbre droit vérifient $x_{\dim} \geq w_{\dim}$.

```
type 'a kd_tree =
  | Nil
  | Node of (vector * 'a) * int * 'a kd_tree * 'a kd_tree
```

Le type `'a` est celui des données associées aux points, ici les classes.

Construction. Pour construire un arbre k-d à partir d'un tableau de points en partant de la dimension $\dim$, on trie le tableau selon la coordonnée $\dim$, on place l'élément médian à la racine et on construit récursivement les sous-arbres avec les éléments strictement à gauche et strictement à droite de la médiane, en passant à la dimension $(\dim + 1) \bmod d$. L'arbre obtenu est équilibré : sa hauteur est $\lceil \log_2 n \rceil + 1$. Avec un tri en $O(m \log_2 m)$ pour un tableau de taille m , la construction coûte $O(n \log_2^2 n)$.

OCaml

```

let rec kd_build a d n =
  let len = Array.length a in
  if len = 0 then Nil
  else begin
    let a = Array.copy a in
    Array.sort (fun (v1, _) (v2, _) -> compare v1.(d) v2.(d)) a;
    let m = len / 2 in
    let d' = (d + 1) mod n in
    Node (a.(m), d,
          kd_build (Array.sub a 0 m) d' n,
          kd_build (Array.sub a (m + 1) (len - m - 1)) d' n)
  end

```

Recherche des k plus proches voisins. On utilise une distance dist telle que $|x_i - y_i| \leq \text{dist}(x, y)$ pour toute coordonnée i , ce qui est le cas des distances d_1 et d_2 . Pour chercher les k plus proches voisins de v dans un nœud (w, dim, g, d) :

- on cherche récursivement dans le sous-arbre du côté de v (g si $v_{\text{dim}} \leq w_{\text{dim}}$, d sinon), ce qui donne une liste vois1 ;
- si vois1 contient k points tous à distance au plus $|v_{\text{dim}} - w_{\text{dim}}|$ de v , on renvoie vois1 ;
- sinon, on cherche aussi dans l'autre sous-arbre, ce qui donne vois2 , et on renvoie les k plus proches de v parmi vois1 , vois2 et w .

OCaml

```

(* les k premiers éléments d'une liste *)
let rec prefix k l =
  match l with
  | x :: q when k > 0 -> x :: prefix (k - 1) q
  | _ -> []

let rec kd_search_knn dist k t v =
  match t with
  | Nil -> []
  | Node ((w, c), dim, g, d) ->
    let first, other = if v.(dim) <= w.(dim) then g, d else d, g in
    let vois1 = kd_search_knn dist k first v in
    let sep = abs_float (v.(dim) -. w.(dim)) in
    if List.length vois1 = k
    && List.for_all (fun (v', _) -> dist v v' <= sep) vois1
    then vois1
    else begin
      let vois2 = kd_search_knn dist k other v in
      let l = List.sort
        (fun (v1, _) (v2, _) -> compare (dist v1 v) (dist v2 v))
        ((w, c) :: vois1 @ vois2) in
      prefix k l
    end

```

Théorème 23.7

$\text{kd_search_knn dist k t v}$ renvoie $\min(k, |t|)$ points de t tels qu'aucun point de t non renvoyé ne soit strictement plus proche de v qu'un point renvoyé.

Preuve Par induction sur l'arbre. C'est immédiat pour Nil . Pour un nœud, on suppose, par hypothèse d'induction, que vois1 est formée de k plus proches voisins de v dans le sous-arbre exploré (ou de tous ses points s'il en a moins de k), et de même pour vois2 .

- Si on explore les deux sous-arbres, tout point de t est w , ou un point de vois1 ou de vois2 , ou un point d'un sous-arbre qui est à distance au moins égale à celle de chacun des k points de la liste correspondante. En gardant les k plus proches de l'union, on obtient bien k plus proches voisins dans t (ou tous les points de t s'il y en a au plus k).
- Sinon, vois1 contient k points à distance au plus $s = |v_{\text{dim}} - w_{\text{dim}}|$ de v . Supposons par exemple $v_{\text{dim}} \leq w_{\text{dim}}$: tout point x du sous-arbre droit, ainsi que w , vérifie $x_{\text{dim}} \geq w_{\text{dim}}$, donc $\text{dist}(v, x) \geq x_{\text{dim}} - v_{\text{dim}} \geq$

s. Ces points ne sont donc pas strictement plus proches que ceux de v_{vois1} , qui convient. Le cas $v_{\text{dim}} > w_{\text{dim}}$ est symétrique.

□

On en déduit les k plus proches voisins en appliquant `majority` aux classes des points renvoyés. Dans le pire cas, on explore tout l'arbre et la recherche coûte $O(n)$ calculs de distances, plus le coût des tris des listes. En pratique, en petite dimension, une grande partie des sous-arbres est éliminée : dans le TP, en dimension 6 avec 750 points d'entraînement, on évite environ deux tiers des calculs de distances pour $k = 1$. Cet avantage disparaît quand la dimension augmente (pour les images MNIST, $d = 784$) : on parle de **fléau de la dimension**.

23.1.4 Matrice de confusion

Pour évaluer un classifieur sur un jeu de test, le taux de réussite ne suffit pas toujours : on veut savoir **quelles** classes sont confondues.

Définition 23.8

Soit $Y = \{c_1, \dots, c_p\}$. La **matrice de confusion** d'un classifieur $\hat{f}$ sur un jeu de test $((x_1, y_1), \dots, (x_m, y_m))$ est la matrice $M \in \mathcal{M}_p(\mathbb{N})$ où $M_{i,j}$ est le nombre d'indices k tels que $y_k = c_i$ et $\hat{f}(x_k) = c_j$.

La somme des coefficients vaut m , les coefficients diagonaux comptent les bonnes prédictions et le **taux de réussite** est $\text{tr}(M)/m$. Un classifieur parfait a une matrice de confusion diagonale.

Python

```
def matrice_confusion(k, training, test, nb_classes):
    m = [[0] * nb_classes for _ in range(nb_classes)]
    for x, c in test:
        m[c][nearest_neighbors(distance_2, k, training, x)] += 1
    return m
```

Exemple 23.9

Dans le **TP KNN**, on reconnaît des chiffres manuscrits de la base MNIST : des images de 28×28 pixels, avec 60 000 images d'entraînement et 10 000 images de test. Avec la distance euclidienne et $k = 3$, on obtient la matrice de confusion suivante (ligne : vraie classe, colonne : classe prédite).

	0	1	2	3	4	5	6	7	8	9
0	974	1	1	0	0	1	2	1	0	0
1	0	1133	2	0	0	0	0	0	0	0
2	10	9	996	2	0	0	0	13	2	0
3	0	2	4	976	1	13	1	7	3	3
4	1	6	0	0	950	0	4	2	0	19
5	6	1	0	11	2	859	5	1	3	4
6	5	3	0	0	3	3	944	0	0	0
7	0	21	5	0	1	0	0	991	0	10
8	8	2	4	16	8	11	3	4	914	4
9	4	5	2	8	9	2	1	8	2	968

La trace vaut 9705, soit un taux de réussite de 97,05 %. Au-delà de ce taux global, on lit les confusions les plus fréquentes : 21 chiffres 7 classés comme des 1, 19 chiffres 4 classés comme des 9, 16 chiffres 8 classés comme des 3. Avec $k = 1$, le taux de réussite est de 96,91 %.

Remarque 23.10 Pour deux classes, positive et négative, on parle de vrais positifs, faux positifs, vrais négatifs et faux négatifs. Selon l'application, les deux types d'erreurs n'ont pas la même gravité : pour un test médical de dépistage, un faux négatif est bien plus grave qu'un faux positif.

23.1.5 Arbres de décision et algorithme ID3

On considère maintenant des attributs qui prennent un nombre fini de valeurs : $X = A_1 \times \dots \times A_d$ où chaque A_i est fini.

Définition 23.11

Un **arbre de décision** est un arbre dont chaque nœud interne est étiqueté par un attribut i et possède un fils pour chaque valeur de A_i , et dont chaque feuille est étiquetée par une classe.

Pour classer $x = (a_1, \dots, a_d)$, on part de la racine et, sur un nœud étiqueté par i , on descend dans le fils correspondant à la valeur a_i jusqu'à atteindre une feuille, dont l'étiquette est la classe prédite.

Exemple 23.12

On veut prédire si on va jouer au tennis selon la météo, avec les attributs Ciel $\in \{\text{soleil, couvert, pluie}\}$, Température $\in \{\text{chaud, doux, frais}\}$, Humidité $\in \{\text{élevée, normale}\}$ et Vent $\in \{\text{faible, fort}\}$. On dispose de 14 exemples (jeu de données classique dû à Quinlan, l'auteur d'ID3) :

Ciel	Température	Humidité	Vent	Tennis
soleil	chaud	élevée	faible	non
soleil	chaud	élevée	fort	non
couvert	chaud	élevée	faible	oui
pluie	doux	élevée	faible	oui
pluie	frais	normale	faible	oui
pluie	frais	normale	fort	non
couvert	frais	normale	fort	oui
soleil	doux	élevée	faible	non
soleil	frais	normale	faible	oui
pluie	doux	normale	faible	oui
soleil	doux	normale	fort	oui
couvert	doux	élevée	fort	oui
couvert	chaud	normale	faible	oui
pluie	doux	élevée	fort	non

On peut toujours construire un arbre de décision qui classe correctement tous les exemples, s'ils ne sont pas contradictoires, mais on cherche un arbre **petit**, qui généralise mieux. Trouver un plus petit arbre est un problème difficile, et l'algorithme ID3 utilise une heuristique gloutonne : on place à la racine l'attribut qui sépare *le mieux* les exemples, puis on recommence dans chaque sous-arbre. Pour mesurer la qualité d'une séparation, on utilise l'entropie, qui mesure le désordre d'un ensemble d'exemples.

Définition 23.13

Soit S un ensemble fini et non vide d'exemples étiquetés et, pour chaque classe c , p_c la proportion d'exemples de S de classe c . L'**entropie** de S est

$$H(S) = - \sum_{c, p_c > 0} p_c \log_2 p_c$$

Propriété 23.14

Si S contient des exemples de p classes, alors $0 \leq H(S) \leq \log_2 p$. De plus, $H(S) = 0$ si et seulement si tous les exemples de S ont la même classe, et $H(S) = \log_2 p$ si et seulement si les p classes ont la même proportion.

Preuve Comme $0 < p_c \leq 1$, chaque terme $-p_c \log_2 p_c$ est positif, et nul si et seulement si $p_c = 1$, d'où le premier point. Pour le second, on utilise la concavité stricte de $\log_2$ (inégalité de Jensen) avec les poids p_c qui sont de somme 1 :

$$H(S) = \sum_c p_c \log_2 \frac{1}{p_c} \leq \log_2 \left(\sum_c p_c \frac{1}{p_c} \right) = \log_2 p$$

avec égalité si et seulement si tous les $1/p_c$ sont égaux. □

Remarque 23.15 Avec deux classes, si p est la proportion d'exemples de la première classe, on a $H(S) = B(p)$ où

$$B(p) = -p \log_2 p - (1 - p) \log_2 (1 - p)$$

en prolongeant par continuité $B(0) = B(1) = 0$. C'est cette fonction qu'on utilise dans le **TP ID3**, où les classes sont les booléens `true` et `false`.

Définition 23.16

Soit S un ensemble d'exemples et i un attribut. Pour $a \in A_i$, on note S_a l'ensemble des exemples de S dont l'attribut i vaut a . Le **gain d'information** de i sur S est

$$G(S, i) = H(S) - \sum_{a \in A_i, S_a \neq \emptyset} \frac{|S_a|}{|S|} H(S_a)$$

C'est la diminution de l'entropie moyenne obtenue en séparant S selon l'attribut i . L'algorithme ID3 construit l'arbre récursivement à partir d'un ensemble d'exemples S et d'un ensemble d'attributs disponibles A :

- si tous les exemples de S ont la même classe, on renvoie une feuille étiquetée par cette classe ;
- si A est vide, on renvoie une feuille étiquetée par la classe majoritaire de S ;
- sinon, on choisit l'attribut $i \in A$ de gain $G(S, i)$ maximal et on crée un nœud étiqueté par i dont les fils sont les arbres construits récursivement à partir des S_a et de $A \setminus \{i\}$.

Un ensemble S_a peut être vide quand aucun exemple n'a la valeur a pour l'attribut i : on renvoie alors une feuille étiquetée par la classe majoritaire de l'ensemble S du nœud parent. C'est le rôle du paramètre `prec_s` dans le TP. La version python ci-dessous ne crée pas ces fils et traite ce cas au moment de la prédiction.

L'algorithme termine car le nombre d'attributs disponibles diminue à chaque appel récursif, et les feuilles construites classent correctement les exemples dès qu'il n'y en a pas deux de mêmes attributs et de classes différentes.

Dans la version `ocaml`, qui est celle du **TP ID3**, les attributs sont booléens : un exemple est un couple (values, b) où `values` est un `bool array` et `b` la classe, et un nœud `Node (i, t_yes, t_no)` teste la valeur de l'attribut i . Dans la version `python`, un exemple est un couple (x, c) où x est un tuple de valeurs d'attributs quelconques, et un arbre est soit une classe, soit un couple formé d'un indice d'attribut et d'un dictionnaire associant à chaque valeur le sous-arbre correspondant.

OCaml

```
type samples = (bool array * bool) list
```

```

type decision_tree =
  | Node of int * decision_tree * decision_tree
  | Leaf of bool

let rec eval_tree t values =
  match t with
  | Leaf b -> b
  | Node (i, t_yes, t_no) ->
    if values.(i) then eval_tree t_yes values else eval_tree t_no values

(* classe majoritaire, true en cas d'égalité *)
let majority s =
  let n = List.length s in
  let p = List.length (List.filter snd s) in
  2 * p >= n

let b p =
  if p <= 1e-12 || p >= 1. -. 1e-12 then 0.
  else -. p *. Float.log2 p -. (1. -. p) *. Float.log2 (1. -. p)

let entropy s =
  let n = List.length s in
  let p = List.length (List.filter snd s) in
  if n = 0 then 0. else b (float_of_int p /. float_of_int n)

let gain s i =
  let s_yes, s_no = List.partition (fun (v, _) -> v.(i)) s in
  let n = float_of_int (List.length s) in
  entropy s
  -. float_of_int (List.length s_yes) /. n *. entropy s_yes
  -. float_of_int (List.length s_no) /. n *. entropy s_no

(* s : exemples courants, prec_s : exemples du parent,
   free : attributs disponibles *)
let rec id3_rec s prec_s free =
  if s = [] then Leaf (majority prec_s)
  else if List.for_all snd s then Leaf true
  else if not (List.exists snd s) then Leaf false
  else if free = [] then Leaf (majority s)
  else begin
    let best = List.fold_left
      (fun i j -> if gain s j > gain s i then j else i)
      (List.hd free) free in
    let s_yes, s_no = List.partition (fun (v, _) -> v.(best)) s in
    let free' = List.filter (fun j -> j <> best) free in
    Node (best, id3_rec s_yes s free', id3_rec s_no s free')
  end

let id3 s vars = id3_rec s s (List.init vars (fun i -> i))

```

Python

```

from math import log2

def entropie(exemples):
    """exemples est une liste de couples (attributs, classe)"""
    n = len(exemples)
    effectifs = {}
    for _, c in exemples:
        effectifs[c] = effectifs.get(c, 0) + 1
    return -sum(e / n * log2(e / n) for e in effectifs.values())

def partition(exemples, a):
    """regroupe les exemples selon la valeur de l'attribut a"""
    parts = {}
    for x, c in exemples:
        parts.setdefault(x[a], []).append((x, c))

```

```

    return parts

def gain(exemples, a):
    n = len(exemples)
    parts = partition(exemples, a)
    return entropie(exemples) - sum(len(p) / n * entropie(p)
                                    for p in parts.values())

def majoritaire(exemples):
    classes = [c for _, c in exemples]
    return max(set(classes), key=classes.count)

def id3(exemples, attributs):
    """renvoie soit une classe (feuille),
    soit un couple (attribut, dictionnaire valeur -> sous-arbre)"""
    classes = set(c for _, c in exemples)
    if len(classes) == 1:
        return classes.pop()
    if attributs == []:
        return majoritaire(exemples)
    a = max(attributs, key=lambda b: gain(exemples, b))
    restants = [b for b in attributs if b != a]
    return (a, {v: id3(p, restants)
                for v, p in partition(exemples, a).items()})

def predict(arbre, x, default):
    while isinstance(arbre, tuple):
        a, fils = arbre
        if x[a] not in fils: # valeur jamais vue à l'entraînement
            return default
        arbre = fils[x[a]]
    return arbre

```

Exemple 23.17

Sur les 14 exemples précédents, 9 sont positifs et 5 négatifs, donc $H(S) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} \approx 0,940$. Les gains des attributs sont environ :

Ciel	Température	Humidité	Vent
0,247	0,029	0,152	0,048

L'attribut Ciel est placé à la racine. Les exemples où le ciel est couvert sont tous positifs. On obtient finalement l'arbre suivant, qui classe correctement les 14 exemples en n'utilisant pas la température :

- Ciel = soleil : Humidité = élevée → non, Humidité = normale → oui ;
- Ciel = couvert → oui ;
- Ciel = pluie : Vent = faible → oui, Vent = fort → non.

Remarque 23.18

- Un arbre trop profond colle aux exemples, y compris à leurs erreurs, et généralise mal : on parle de **sur-apprentissage**. On peut limiter la profondeur de l'arbre ou élaguer les branches qui n'améliorent pas les résultats sur un jeu de test.
- Si une valeur d'attribut n'apparaît jamais dans les exemples d'un nœud, la fonction `predict` renvoie une classe par défaut, par exemple la classe majoritaire du jeu d'entraînement.

23.2 Apprentissage non-supervisé



23.2.1 Partitionnement

On ne dispose plus des classes : on a seulement des points $x_1, \dots, x_n \in \mathbb{R}^d$ et on cherche à les regrouper en **groupes** (ou **clusters**) de points proches. Par exemple, on peut vouloir regrouper des clients selon leurs achats sans savoir à l'avance quels profils existent.

Définition 23.19

Un **partitionnement** de $\{x_1, \dots, x_n\}$ en k groupes est une application $c : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, k \rrbracket$, $c(j)$ étant le groupe du point x_j .

Il n'y a pas ici de « bonne réponse » à laquelle se comparer : la qualité d'un partitionnement dépend d'un critère qu'on se fixe.

23.2.2 Algorithme des k -moyennes

On mesure la qualité d'un partitionnement c associé à des centres $\mu_1, \dots, \mu_k \in \mathbb{R}^d$ par l'**inertie**

$$J(c, \mu) = \sum_{j=1}^n \|x_j - \mu_{c(j)}\|^2$$

qu'on cherche à minimiser. Minimiser exactement J est un problème difficile et on utilise l'heuristique suivante, due à Lloyd.

- On choisit k centres initiaux, par exemple k points du jeu de données, et on affecte chaque point au centre le plus proche.
- Tant que l'affectation change :
 - on remplace chaque centre μ_i par le barycentre des points du groupe i ;
 - on affecte chaque point au centre le plus proche, en ne changeant le groupe d'un point que si un autre centre est **strictement** plus proche.

Comme dans le **TP KNN avec arbres k-d**, la fonction `k_means dist data k` prend la distance en paramètre et renvoie le couple `(classes, centers)` où `classes.(j)` est le groupe de `data.(j)`.

OCaml

```
let barycentre points =
  let n = List.length points in
  let dim = Array.length (List.hd points) in
  let b = Array.make dim 0. in
  List.iter (fun p ->
    Array.iteri (fun i v -> b.(i) <- b.(i) +. v /. float_of_int n) p) points;
  b

let plus_proche dist centers x =
  let meilleur = ref 0 in
  for i = 1 to Array.length centers - 1 do
    if dist centers.(i) x < dist centers.(!meilleur) x
    then meilleur := i
  done;
  !meilleur

(* on suppose 1 <= k <= Array.length data ; les k premiers points
servent de centres initiaux *)
let k_means dist data k =
  let centers = Array.init k (fun i -> Array.copy data.(i)) in
  let classes = Array.map (plus_proche dist centers) data in
  let change = ref true in
  while !change do
    (* mise à jour des centres *)
    for i = 0 to k - 1 do
      let groupe = ref [] in
      Array.iteri (fun j x -> if classes.(j) = i then groupe := x :: !groupe)
        data;
      if !groupe <> [] then centers.(i) <- barycentre !groupe
```

```

done;
(* affectation de chaque point au centre le plus proche *)
change := false;
Array.iteri (fun j x ->
  let i = plus_proche dist centers x in
  if dist centers.(i) x < dist centers.(classes.(j)) x then begin
    classes.(j) <- i;
    change := true
  end) data
done;
classes, centers

```

Python

```

def barycentre(points):
    n = len(points)
    return tuple(sum(p[i] for p in points) / n for i in range(len(points[0])))

def plus_proche(dist, centers, x):
    return min(range(len(centers)), key=lambda i: dist(centers[i], x))

def k_means(dist, data, k):
    # on suppose 1 <= k <= len(data) ; les k premiers points
    # servent de centres initiaux
    centers = list(data[:k])
    classes = [plus_proche(dist, centers, x) for x in data]
    change = True
    while change:
        # mise à jour des centres
        for i in range(k):
            groupe = [x for x, c in zip(data, classes) if c == i]
            if groupe != []:
                centers[i] = barycentre(groupe)
        # affectation de chaque point au centre le plus proche
        change = False
        for j, x in enumerate(data):
            i = plus_proche(dist, centers, x)
            if dist(centers[i], x) < dist(centers[classes[j]], x):
                classes[j] = i
                change = True
    return classes, centers

```

Remarque 23.20 Un groupe peut devenir vide. Dans ce cas, on conserve son ancien centre.

Lemme 23.21

Soit C un ensemble fini et non vide de points de $\mathbb{R}^d$, de barycentre m . Pour tout $\mu \in \mathbb{R}^d$,

$$\sum_{x \in C} \|x - \mu\|^2 = \sum_{x \in C} \|x - m\|^2 + |C| \|m - \mu\|^2$$

En particulier, le barycentre minimise $\mu \mapsto \sum_{x \in C} \|x - \mu\|^2$.

Preuve On développe $\|x - \mu\|^2 = \|(x - m) + (m - \mu)\|^2 = \|x - m\|^2 + 2\langle x - m, m - \mu \rangle + \|m - \mu\|^2$, puis on somme sur $x \in C$: le terme central est nul car $\sum_{x \in C} (x - m) = 0$ par définition du barycentre.

□

Théorème 23.22

L'algorithme des k -moyennes termine, et l'inertie décroît au cours de l'algorithme.

Preuve Pour une affectation c , on note $m(c)$ les centres obtenus après la mise à jour à partir de c et $I(c) = J(c, m(c))$. Considérons un tour de boucle qui commence avec l'affectation c et se termine avec l'affectation c' .

- Lors de l'affectation, chaque point qui change de groupe se rapproche strictement de son centre, et les autres termes ne changent pas. Donc si $c' \neq c$, $J(c', m(c)) < J(c, m(c)) = I(c)$.
- Lors de la mise à jour suivante, d'après le lemme appliqué à chaque groupe non vide de c' , $I(c') = J(c', m(c')) \leq J(c', m(c))$.

Ainsi, tant que l'affectation change, I décroît strictement. Une même affectation ne peut donc pas apparaître deux fois. Comme il n'y a que k^n affectations possibles, l'algorithme termine. □

Remarque 23.23

- Chaque tour de boucle coûte $O(nkd)$. La borne k^n sur le nombre de tours est très pessimiste : en pratique, l'algorithme converge en peu de tours.
- L'algorithme aboutit à un minimum *local* : aucun point ne gagne à changer de groupe, mais ce n'est pas forcément un minimum de J . Le résultat dépend des centres initiaux, et on relance souvent l'algorithme avec plusieurs initialisations aléatoires en gardant le résultat de plus petite inertie.
- Il faut choisir k à l'avance. On peut comparer les inerties obtenues pour plusieurs valeurs de k , sachant que l'inertie optimale décroît quand k augmente.

Exemple 23.24

Sur 90 points générés autour de $(0, 0)$, $(5, 0)$ et $(2, 5; 4)$ avec un écart-type de 0,7, l'algorithme avec $k = 3$ retrouve des centres proches de $(0, 15; -0, 01)$, $(4, 87; -0, 15)$ et $(2, 57; 4, 32)$.

23.2.3 Classification hiérarchique ascendante

Plutôt que de fixer k dès le départ, on peut construire une hiérarchie de regroupements :

- au départ, chaque point forme un groupe ;
- tant qu'il reste plus d'un groupe, on fusionne les deux groupes les plus proches.

On obtient un arbre binaire, appelé **dendrogramme**, dont les feuilles sont les points et dont chaque nœud interne correspond à une fusion. En arrêtant les fusions quand il reste k groupes, on obtient un partitionnement en k groupes.

Il faut définir une distance entre groupes. Les choix classiques pour deux groupes G et G' sont :

- le **saut minimal** : $\min\{d(x, y) \mid x \in G, y \in G'\}$;
- le **saut maximal** : $\max\{d(x, y) \mid x \in G, y \in G'\}$;
- la distance entre les barycentres de G et G' .

Le code suivant utilise le saut minimal et s'arrête quand il reste k groupes.

Python

```
def distance_min(g1, g2):
    return min(distance_2(x, y) for x in g1 for y in g2)

def cah(points, k):
    groupes = [[x] for x in points]
    while len(groupes) > k:
        # recherche des deux groupes les plus proches
        _, i, j = min((distance_min(groupe1, groupe2), i, j)
                      for i in range(len(groupe1))
                      for j in range(i + 1, len(groupe2)))
        groupes[i] = groupes[i] + groupes[j]
```



```
groupes.pop(j)
return groupes
```

Cette implémentation directe est très coûteuse : chaque fusion recalcule toutes les distances entre points de groupes différents, soit $O(n^2)$ calculs de distances, et il y a $n - k$ fusions, d'où une complexité en $O(n^3)$. On peut faire mieux en conservant une matrice des distances entre groupes et en la mettant à jour après chaque fusion.

Remarque 23.25 Avec le saut minimal, fusionner les deux groupes les plus proches revient à ajouter l'arête la plus courte entre deux groupes différents du graphe complet pondéré par les distances : c'est exactement l'algorithme de Kruskal, arrêté quand il reste k composantes connexes.

23.3 Recherche informée

👉 | L'algorithme A^* est mis en œuvre sur le taquin dans le [TP Résolution du 15-puzzle](#).

KNN - Identifier des chiffres

DANS CE CHAPITRE

24.1 Données MNIST

24.2 Plus proches voisins

24.3 Aller plus loin

Ce TP met en œuvre l'algorithme des *k* plus proches voisins et la *matrice de confusion* du cours d'apprentissage.

24.1 Données MNIST

Note : ce TP **en C** est volontairement très peu guidé. Si toutefois vous avez vraiment du mal à démarrer, le squelette du code, sans le corps des fonctions demandées, avec la visualisation en raylib, se trouve ici.

On va considérer ici le jeu de données MNIST qui fournit des milliers d'images en basse résolution de chiffres écrits à la main par des Américains (ça a son importance sur la forme des chiffres).

Source des données : base MNIST de Yann LeCun, Corinna Cortes et Christopher J. C. Burges.

On dispose de deux jeux de données, des données d'entraînement et des données de tests.

Chaque jeu de données est composé de deux fichiers :

- un fichier finissant par `images-idx3-ubyte` qui contient les images dans le format suivant :
 - un entier sur 4 octets en grand boutiste indiquant le format du fichier, c'est ce qu'on appelle un nombre magique, comme `PK` au début des fichiers zip. On va l'ignorer ;
 - un entier dans le même format indiquant le nombre N d'images ;
 - deux entiers dans le même format indiquant la hauteur H (le nombre de lignes) puis la largeur W (le nombre de colonnes) des images, ici $H = W = 28$;
 - une suite de $W \times H$ octets pour chacune des N images, ligne par ligne, indiquant l'image en niveaux de gris.
- un fichier finissant par `labels-idx1-ubyte` :
 - un entier sur 4 octets en grand boutiste indiquant le format du fichier (le nombre magique), qu'on va ignorer ;
 - un entier dans le même format indiquant le nombre N d'images ;
 - une suite de N octets où le i -ème octet indique la classe de l'image, entre 0 et 9.

Vous pouvez trouver les données ici :

- Entraînement :
 - `train-images-idx3-ubyte`
 - `train-labels-idx1-ubyte`

- Tests :
 - t10k-images-idx3-ubyte
 - t10k-labels-idx1-ubyte

Remarque 24.1 Si on considère l'entier s'écrivant $0x12345678$ en hexadécimal, il consiste en 4 octets qui en hexadécimal s'écrivent 12, 34, 56 et 78. L'octet 12 est celui de poids fort (il est multiplié par 2^{24}) et l'octet 78 celui de poids faible (multiplié par 2^0).

En mémoire, on peut représenter cet entier de deux manières :

- soit en **grand boutiste** (*big endian*), où les octets sont dans cet ordre dans le fichier ou en mémoire : 12 34 56 78 ;
- soit en **petit boutiste** (*little endian*), où les octets sont dans cet ordre dans le fichier ou en mémoire : 78 56 34 12.

Le nom indique par quel « bout » on commence : par l'octet de poids fort dans le premier cas, par l'octet de poids faible dans le second. Les processeurs de nos machines (x86, ARM) sont petit boutistes : il faudra donc convertir les entiers lus dans les fichiers MNIST.

Question 1 Écrire une fonction réalisant la conversion d'une représentation à l'autre.

On pourra utiliser des divisions et des modulus pour extraire les octets, puis des multiplications pour recomposer.

Par exemple, $(x / 256) \% 256$ va être le deuxième octet en partant du poids faible.

Question 2 Définir un type permettant de représenter une donnée : la classe et l'image.

Attention : il est crucial d'avoir des `unsigned char` pour les pixels et pas des `char`, sinon rien ne va marcher !

Question 3 Décoder les fichiers pour obtenir des tableaux de données.

On va utiliser ici la fonction `fread` :

```
BIBLIOTHÈQUE
    Bibliothèque C standard (libc, -lc)

SYNOPSIS
    #include <stdio.h>

    size_t fread(void *restrict ptr, size_t taille, size_t nbelem,
                  FILE *restrict flux);

DESCRIPTION
    La fonction fread() lit nbelem éléments de données de
    taille octets chacun depuis le flux pointé par flux, et les
    stocke à l'emplacement pointé par ptr.
```

On rappelle qu'on obtient un `FILE*` avec `fopen` (attention, peut-être en "rb" si vous êtes sous Windows) et qu'on le ferme avec `fclose`.

24.2 Plus proches voisins

Question 4 Écrire une fonction permettant de calculer la distance euclidienne entre deux images.

Question 5 Implémenter l'algorithme KNN.

Question 6 Calculer les matrices de confusion pour $k = 1, 3, 5, 7$ et le taux d'erreur sur le jeu de test.



Remarque 24.2 Pour vérifier vos résultats : avec la distance euclidienne, les 60 000 images d'entraînement et les 10 000 images de test, on obtient un taux d'erreur d'environ 3 % pour chacune de ces valeurs de k (3, 09 % pour $k = 1$ et 2, 95 % pour $k = 3$, en départageant les égalités de votes par la plus petite classe). Le

calcul complet prend de l'ordre de la minute : on pourra commencer par tester sur les premières images de test seulement.

24.3 Aller plus loin

Question 7 Réaliser une interface minimaliste avec `raylib` pour écrire à la souris et voir la reconnaissance.

On affichera notamment les images des k plus proches voisins.

Classification en basse dimension

DANS CE CHAPITRE

25.1 Principe et extraction de données

25.2 Normalisation des coordonnées

25.3 k plus proches voisins naïf

25.4 k-moyennes

25.5 Piste d'exploration : visualisation par la réduction de la dimension

25.1 Principe et extraction de données

Dans ce TP, on va reprendre un problème de classification, comme dans le **TP KNN**, mais avec une dimension suffisamment faible pour que les **arbres k-d** permettent une accélération notable.

On va considérer un jeu de données de 898 pokémons (pas sûr de tous les avoir attrapés...), qui comporte 1032 lignes car certains pokémons y apparaissent sous plusieurs formes. Ce jeu de données est dans le fichier `All_Pokemon_knn.csv`.

C'est un fichier CSV donc *Comma Separated Values*. Sa première ligne indique les noms des colonnes séparés par des `,` : `Number,Name,Type 1,...`

Ensuite, chaque ligne suivante renseigne un pokémon donné, comme par exemple ce pokémon :

```
707,Klefki,Steel,Fairy,"['Magician'; 'Prankster']",57,80,91,80,87,75,470,
78.33333333333333,10.85766498326822,6.0,Fast,800000,1.0,75,0.0,0.0,0.0,
0.0,0.5,2.0,1.0,1.0,0.5,0.5,1.0,0.0,2.0,0.5,0.5,0.25,0.5,1.0,0.0,0.5,1.0,
0.5,0.2,3.0,75.0
```

(la ligne a été coupée ici pour l'affichage). Dans le fichier d'origine, la colonne `Abilities` contient des virgules entre guillemets : elles ont été remplacées par des `;` dans le fichier fourni, ce qui permet de découper chaque ligne simplement selon les virgules.

On va s'intéresser dans ce TP à étudier les liens entre certaines statistiques pour identifier des motifs. Par exemple : est-ce que les statistiques `HP,Att,Def,Spa,Spd,Spe` permettent de deviner `Height` ou `Weight` ?

Tout d'abord, il va falloir lire ce fichier. Pour cela, on rappelle la manière dont on peut écrire des fonctions de lecture des lignes d'un fichier (on va adapter cette fonction dans la suite) :

```
let rec read_lines f =
  try
    let line = input_line f in
    line :: read_lines f
  with End_of_file -> []
```

On rappelle les fonctions suivantes d'OCaml :

```
(* pour obtenir un canal de lecture sur un fichier *)
val open_in : string -> in_channel
(* pour lire une ligne sans le saut de ligne final *)
val input_line : in_channel -> string
(* pour séparer une chaîne selon un caractère :
 * avec '-', "a-bb-c" donne [ "a"; "bb"; "c" ] *)
val String.split_on_char : char -> string -> string list
```

Question 1 Écrire une fonction

```
val list_to_dict : string list -> string list -> (string, string) Hashtbl.t
```

qui prend en entrée deux listes de chaînes de **même longueur**, les champs et les valeurs, et construit un dictionnaire avec.

Par exemple, si on appelle

```
list_to_dict ["nom"; "prenom"; "age"] ["Tajiri"; "Satoshi"; "57"]
```

on va obtenir une table de hachage avec les correspondances comme "age" -> "57".

Question 2 Écrire une fonction

```
val input_csv : in_channel -> string list
```

qui lit une ligne dans le fichier donné et renvoie la liste des sous-chaînes obtenues en séparant selon les virgules.

On définit

```
type pokemon = (string, string) Hashtbl.t
```

Question 3 En déduire une fonction

```
val csv_to_dicts : string -> pokemon array
```

qui prend un nom de fichier CSV, lit la ligne d'en-tête et construit tous les dictionnaires représentant les valeurs des lignes suivantes.

Question 4 Définir un array pokemons qui contient le tableau de tous les dictionnaires du fichier.

25.2 Normalisation des coordonnées

Dans le TP précédent, les données étaient semblables en magnitude sur toutes les dimensions car elles étaient de même nature. Mais ici, il ne va pas être raisonnable de calculer des distances avec des HP entre 50 et 150 quand les tailles sont souvent autour de 1 m.

On va ainsi procéder à une renormalisation des données. Pour normaliser une dimension, on calcule m la valeur minimale et M la valeur maximale, et on remplace chaque valeur v par $\frac{v-m}{M-m} \in [0; 1]$.

Question 5 Écrire une fonction

```
val field_spread : string -> float * float
```

telle que `field_spread field` renvoie le couple (`min_value`, `max_value`) associé à l'attribut `field`, **qui sera nécessairement un attribut de valeur numérique**.

On pourra ainsi convertir la valeur avec `float_of_string`.

Question 6 Écrire une fonction

```
val normalize : float * float -> float -> float
```

telle que l'appel `normalize spread v` réalise le calcul précédent, où `spread` est le couple $(\text{min_value}, \text{max_value})$.
On définit

```
type vector = float array
```

Question 7 Écrire une fonction

```
val data_vector : string list -> (float * float) list -> pokemon -> vector
```

qui prend une liste de champs en entrée dont les valeurs sont **toutes numériques**, la liste de leurs étendues (min, max) associées, un dictionnaire d , et renvoie un tableau de flottants où le i -ème flottant est la valeur dans d du i -ème champ, convertie en `float` puis **normalisée**.

Pour classer les données, on va commencer par choisir un champ cible et déterminer les quartiles (q_1, q_2, q_3) associés : le quartile q_i est une valeur telle qu'environ une proportion $\frac{i}{4}$ des valeurs de ce champ sont $< q_i$. Si on trie les n valeurs $v_0 \leq v_1 \leq \dots \leq v_{n-1}$, on pourra prendre $q_i = v_{\lfloor in/4 \rfloor}$. C'est une généralisation de la médiane : on coupe en quatre et on regarde les trois valeurs de séparation.

Question 8 Écrire une fonction

```
val field_quartiles : string -> float * float * float
```

qui renvoie les quartiles associés à un champ dans le tableau complet.

On va associer, pour des quartiles (q_1, q_2, q_3) :

- la classe 0 aux valeurs $v < q_1$;
- la classe 1 aux valeurs dans $[q_1; q_2[$;
- la classe 2 aux valeurs dans $[q_2; q_3[$;
- la classe 3 aux valeurs $v \geq q_3$.

Question 9 Écrire une fonction

```
val classify : float * float * float -> float -> int
```

qui renvoie la classe d'une valeur.

Question 10 En déduire une fonction

```
val data_pair : string list -> (float * float) list
-> string -> float * float * float
-> pokemon -> vector * int
```

telle que `data_pair fields fields_spreads target target_quartiles pokemon` :

- extrait le vecteur `vec` des données associé à `pokemon` pour les champs `fields` avec les étendues `fields_spreads` ;
- calcule la classe de `pokemon` à l'aide des quartiles `target_quartiles` pour le champ `target` ;
- renvoie alors le couple $(\text{vec}, \text{classe})$.

25.3 k plus proches voisins naïf

On va chercher maintenant, dans les couples $(\text{vec}, \text{classe})$ précédents, à deviner classe à partir de `vec` en utilisant la méthode des plus proches voisins sur une partie des données.

On va reprendre ce qu'on a fait en C pour les k plus proches voisins.

Afin de déterminer des données d'entraînement et de test, on ne peut pas prendre un préfixe. En effet, comme les pokémons sont rangés dans l'ordre officiel, cela serait sûrement biaisé.

Question 11 Écrire une fonction

```
val shuffle_array : 'a array -> unit
```

qui mélange le tableau.



Remarque 25.1 Le mélange de Knuth (ou Fisher-Yates) consiste, pour i allant de 1 à $n - 1$, à échanger $a.(i)$ avec $a.(j)$ où j est tiré uniformément dans $\llbracket 0, i \rrbracket$ (`Random.int (i+1)`). Il produit chacune des $n!$ permutations avec la même probabilité. Attention : tirer j dans $\llbracket 0, i - 1 \rrbracket$ ne produit que des permutations circulaires.

Question 12 Écrire une fonction

```
val split_data : 'a array -> int -> 'a array * 'a array
```

telle que `split_data t clip` mélange `t` puis renvoie un couple (t_1, t_2) où t_1 est le sous-tableau des `clip` premiers éléments et t_2 le sous-tableau du reste.

Pour récupérer un sous-tableau, on rappelle l'existence de la fonction `Array.sub`.

Question 13 Écrire une fonction

```
val distance_2 : vector -> vector -> float
```

qui prend en entrée deux vecteurs **de même dimension** et renvoie la distance euclidienne entre eux.

Question 14 Écrire une fonction

```
val distance_1 : vector -> vector -> float
```

qui prend en entrée deux vecteurs **de même dimension** et renvoie la distance 1 entre eux : $d_1(v_1, v_2) = \sum_{i=0}^{n-1} |v_1[i] - v_2[i]|$.

Question 15 Écrire une fonction

```
val majority : int list -> int
```

qui prend en entrée une liste d'entiers et renvoie un élément majoritaire (en cas d'égalité, on prend un élément quelconque).

Note : on pourra réaliser une fonction générique fonctionnant avec des valeurs de type `'a` en utilisant une table de hachage si on le souhaite.

Question 16 Écrire une fonction

```
val nearest_neighbors : (vector -> vector -> float)
  -> int -> (vector * 'a) array -> vector -> 'a
```

telle que l'appel

```
nearest_neighbors dist k training v
```

renvoie la classe, ici génériquement de type `'a`, de `v` obtenue par les k plus proches voisins depuis les données `training` en utilisant la distance `dist`.

Question 17 Écrire une fonction

```
val eval_naive : int -> int -> string list -> string -> float
```

telle que `eval_naive k clip fields target` :

- construit le tableau des (vecteur, classe) où :
 - les vecteurs sont extraits à l'aide des attributs `fields` ;
 - la classe se déduit des quartiles des valeurs de l'attribut `target` ;
- mélange le tableau ;
- extrait les `clip` premiers dans `training` et le reste dans `tests` ;
- applique les k plus proches voisins sur chaque donnée de `tests` ;
- renvoie le taux d'erreur, c'est-à-dire la proportion de données de `tests` mal classées.

On pourra dès maintenant tester cette fonction pour obtenir un affichage tel que celui-ci (les valeurs dépendent du mélange aléatoire) :

```
Guessing pokemon Height quartile from HP,Att,Def,Spa,Spd,Spe,Weight.
[250training] 1NN : error rate 51.023018%
[250training] 3NN : error rate 51.534527%
[250training] 5NN : error rate 45.524297%
[500training] 1NN : error rate 52.067669%
[500training] 3NN : error rate 46.992481%
[500training] 5NN : error rate 44.172932%
[750training] 1NN : error rate 48.226950%
[750training] 3NN : error rate 50.354610%
[750training] 5NN : error rate 46.099291%
```



Remarque 25.2 Avec quatre classes de même effectif, répondre au hasard donne un taux d'erreur de 75 %. Les plus proches voisins font mieux, mais les statistiques de combat ne suffisent pas à bien prédire la taille.

25.3.1 k plus proches voisins accéléré avec un arbre k-d

On définit le type suivant :

```
type 'a kd_tree =
| Nil
| Node of (vector * 'a) * int * 'a kd_tree * 'a kd_tree
```

Le type `'a` est le type des éléments qu'on veut ranger dans un arbre k-d et le `int` désigne la dimension selon laquelle on a découpé les données dans ce nœud.

Pour construire un arbre k-d contenant les vecteurs du tableau `a` ayant `n` dimensions, on va adopter l'algorithme récursif suivant :

`ConstruitKD(a, d, n)`

- `a` est un tableau de N paires (vecteur, donnée) ;
- `n` est un entier indiquant la dimension des vecteurs et $d \in \llbracket 0, n-1 \rrbracket$;
- si le tableau `a` est vide, on renvoie l'arbre vide ;
- si le tableau `a` est un singleton, on renvoie une feuille contenant son unique élément ;
- sinon :
 - pour bien séparer en deux, on trie `a` selon la coordonnée `d` ;
 - la médiane `m` est alors au milieu du tableau, on va raisonner sur les éléments **strictement** à sa gauche et **strictement** à sa droite ;
 - on construit `tg` l'arbre de la première moitié avec la dimension $d' = (d+1) \bmod n$;
 - on construit `td` l'arbre de la seconde moitié avec la dimension d' ;
 - on renvoie un nœud avec l'élément médian, la dimension `d`, `tg` à gauche et `td` à droite.

Question 18 Écrire une fonction

```
val kd_build : (vector * 'a) array -> int -> int -> 'a kd_tree
```

qui réalise cet algorithme.

Pour effectuer une recherche des k plus proches voisins dans un arbre k-d, on adopte là aussi un algorithme récursif :

RechercheKD(dist, k, t, v)

- dist est une fonction distance, telle qu'on a toujours $|x_i - y_i| \leq \text{dist}(x, y)$;
- k est un entier ;
- t est un arbre k-d ;
- v est le vecteur dont on cherche les k plus proches voisins.

L'algorithme est alors le suivant :

- si t est Nil, on renvoie la liste vide ;
- sinon, on est sur un nœud (w,dim,g,d) pour le vecteur w et la dimension dim :
 - si la coordonnée $v[\text{dim}] \leq w[\text{dim}]$, on effectue récursivement une recherche dans g qui donne une liste vois1 ;
 - sinon, on effectue récursivement une recherche dans d qui donne une liste vois1 ;
 - si vois1 contient k éléments et que pour tout v' dans vois1 on a $\text{dist}(v, v') \leq |v[\text{dim}] - w[\text{dim}]|$, alors on renvoie vois1 ;
 - sinon, on effectue une recherche dans le sous-arbre non exploré et on obtient une liste vois2 ;
 - on renvoie alors les k plus proches voisins de $\text{vois1} \cup \text{vois2} \cup \{w\}$ (si cette union a au plus k éléments, on la renvoie telle quelle).

Question 19 Écrire une fonction

```
val kd_search_knn : (vector -> vector -> float) -> int -> 'a kd_tree
    -> vector -> (vector * 'a) list
```

qui prend en entrée une fonction de distance, un entier k, un arbre k-d t et un vecteur v, et renvoie les k plus proches voisins de v dans l'arbre t.

Question 20 En déduire une fonction

```
val eval_kd : int -> int -> string list -> string -> float
```

qui procède comme eval_naive mais en appliquant l'algorithme des k plus proches voisins à l'aide d'un arbre k-d construit sur les données d'entraînement, et renvoie le taux d'erreur.

Question 21 Modifiez vos fonctions en rajoutant une référence globale pour mesurer l'efficacité de l'arbre k-d, c'est-à-dire la proportion de calculs évités par le fait de ne parcourir qu'un seul côté. Cela revient à compter le nombre d'appels à kd_search_knn sur un nœud et à le comparer au nombre de calculs de la méthode naïve, c'est-à-dire au produit des tailles des jeux de test et d'entraînement.

On pourra faire en sorte d'avoir l'affichage suivant :

```
Guessing pokemon Height quartile from HP,Att,Def,Spa,Spd,Spe.
KD-tree efficiency 50.348338%
[250training] 1NN : error rate 53.964194%
KD-tree efficiency 24.361637%
[250training] 3NN : error rate 51.918159%
KD-tree efficiency 16.653708%
[250training] 5NN : error rate 52.173913%
KD-tree efficiency 60.491353%
[500training] 1NN : error rate 50.563910%
KD-tree efficiency 37.242481%
[500training] 3NN : error rate 52.067669%
KD-tree efficiency 27.301504%
[500training] 5NN : error rate 49.436090%
KD-tree efficiency 67.787707%
```

```
[750training] 1NN : error rate 50.354610%
KD-tree efficiency 45.421749%
[750training] 3NN : error rate 48.936170%
KD-tree efficiency 32.979669%
[750training] 5NN : error rate 43.617021%
```

25.4 k-moyennes

Question 22 Écrire une fonction

```
val k_means : (vector -> vector -> float) -> (vector array) -> int ->
  int array * vector array
```

telle que l'appel `k_means dist data k` renvoie un couple `(classes, centers)` où `classes.(i)` est l'entier de 0 à $k - 1$ qui indique la classe de `data.(i)` et `centers` est un tableau à k éléments qui indique les centroïdes (voir l'[algorithme des k-moyennes](#)).

Question 23 Comment pourrait-on étudier les pokémons avec cette fonction pour valider ou invalider la démarche précédente ?

25.5 Piste d'exploration : visualisation par la réduction de la dimension

Il peut être intéressant de tracer nos données dans le plan. Pour cela, on peut chercher deux vecteurs unitaires $\vec{u}$, $\vec{v}$ et, étant donnée une donnée $\vec{x}$, obtenir le point du plan $(\vec{u} \cdot \vec{x}, \vec{v} \cdot \vec{x})$.

Comment pourrait-on s'assurer d'avoir de **bons** vecteurs pour ces projections ?

On effectuera l'exploration en affichant les points avec `Graphics` ou `raylib`.

Fonctions booléennes et ID3

DANS CE CHAPITRE

26.1 Génération de formules

26.2 Arbres de décision

26.3 Algorithme ID3

26.4 Mesure de performance

26.5 Passer des arbres de décision aux formules et réciproquement

Dans ce **long** TP, on va revenir sur le calcul propositionnel et mettre en œuvre l'**algorithme ID3** pour *deviner* des formules.

26.1 Génération de formules

On considère le type de formules suivant :

```
type bool_expr =
  Var of int
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
  | Not of bool_expr
```

Les variables sont des entiers entre 0 et $n - 1$, ce qui va permettre de représenter directement une valuation booléenne par un `bool array` de longueur n .

Question 1 Écrire une fonction

```
val eval : bool_expr -> bool array -> bool
```

telle que `eval e values` évalue l'expression `e` en considérant que la variable `i` a la valeur `values.(i)`.

Note : on supposera le tableau suffisamment long pour contenir toutes les variables présentes dans `e`.

Pour générer une expression aléatoirement, on reprend le principe déjà vu sur les arbres binaires avec un algorithme de génération récursif d'une formule à n nœuds internes :

- si $n = 0$, on génère une feuille avec un numéro de variable aléatoire ;
- sinon, on produit un des trois types de nœuds avec probabilité $1/3$:
 - pour le nœud unaire, on se contente de générer une expression de taille $n - 1$;
 - pour les nœuds binaires, on choisit $k \in \llbracket 0, n - 1 \rrbracket$ aléatoirement et on génère une sous-expression à gauche de k nœuds et à droite de $n - 1 - k$ nœuds.

Question 2 Écrire une fonction

```
val random_expr : int -> int -> bool_expr
```

telle que `random_expr n vars` génère ainsi une expression aléatoire à n nœuds internes et dont les variables sont $< vars$.

Pour effectuer des tests, on va également générer des valuations aléatoires.

Question 3 Écrire une fonction

```
val random_test : int -> bool array
```

telle que `random_test vars` renvoie un tableau de booléens aléatoires de taille `vars`.

On considère le type

```
type samples = (bool array * bool) list
```

Question 4 En déduire une fonction

```
val gen_samples : bool_expr -> int -> int -> samples
```

telle que `gen_samples e vars nsamples` génère aléatoirement `nsamples` tests composés de couples $(values, b)$ où b est la valeur de vérité de l'expression e pour la valuation donnée par `values`.

26.2 Arbres de décision

On considère le type suivant pour les arbres de décision :

```
type decision_tree =  
  Node of int * decision_tree * decision_tree  
  | Leaf of bool
```

Un nœud `Node(i, t_yes, t_no)` représente un choix sur la valeur de vérité de la variable i . Si la variable i est vraie, on continue dans `t_yes`, sinon dans `t_no`.

Question 5 Écrire une fonction

```
val eval_tree : decision_tree -> bool array -> bool
```

telle que `eval_tree t values` renvoie la valeur de vérité de l'arbre de décision `t` pour une valuation donnée par `values`.

26.3 Algorithme ID3

On va implémenter maintenant l'algorithme ID3 vu en cours pour déduire un arbre de décision de valeurs telles que renvoyées par `gen_samples`. Ici, les attributs sont les variables et il y a deux classes, `true` et `false`.

Question 6 Écrire une fonction

```
val majority : samples -> bool
```

qui renvoie la valeur de vérité majoritaire des tests donnés en entrée.

On rappelle que cette valeur est la seconde composante des couples. En cas d'égalité, on pourra prendre une valeur fixée comme `true` ou une valeur aléatoire.

Question 7 Écrire des fonctions


```
val b : float -> float
val entropy : samples -> float
val gain : samples -> int -> float
```

calculant respectivement la valeur $B(p) = -p \log_2 p - (1-p) \log_2 (1-p)$, en prenant garde aux cas 0 et 1 qui ne seront pas forcément atteints exactement avec des `float`, la valeur de l'entropie $H(S)$ et le gain $G(S, i)$ de la variable i vue comme attribut.

Note : on pourra utiliser `Float.log2`.

Question 8 Écrire une fonction

```
val id3_rec : samples -> samples -> int list -> decision_tree
```

telle que `id3_rec s prec_s free`, où :

- `s` est l'ensemble courant des tests ;
- `prec_s` est l'ensemble de tests du nœud parent, qui sert à choisir la valeur de la feuille quand `s` est vide ;
- `free` est la liste des variables non encore utilisées comme attributs ;

renvoie l'arbre de décision associé par l'algorithme ID3.

Question 9 En déduire une fonction d'interface réduite

```
val id3 : samples -> int -> decision_tree
```

telle que `id3 s vars` renvoie un arbre de décision pour les tests `s` sur `vars` variables.

26.4 Mesure de performance

Pour mesurer les performances de l'algorithme ID3, on propose la stratégie suivante :

- on génère une expression `e` à `vars` variables ;
- pour `k` allant de 0 à `N-1` donné :
 - on génère `k` tests ;
 - on construit l'arbre de décision associé par ID3 ;
 - on génère toutes les valuations possibles de `vars` variables ;
 - on calcule la proportion de valuations pour lesquelles l'arbre de décision renvoie la même valeur que `e` ;
- on réalise un graphe des taux de succès en fonction de `k`.

Remarque : vous pouvez factoriser en générant les valuations une fois pour tous les `k` ; ça complique un peu les types, c'est pour ça que je ne l'ai pas fait ci-dessous. Libre à vous d'optimiser !

Question 10 Écrire une fonction

```
val enumerate_liste_values : int -> bool list list
```

telle que `enumerate_liste_values vars` renvoie la liste des valuations, données sous forme de listes de booléens, pour `vars` variables.

Remarque : le fait d'avoir des listes va permettre de réaliser simplement une énumération récursive.

Question 11 En déduire une fonction

```
val enumerate_values : int -> bool array list
```

qui les renvoie sous la forme de tableaux de booléens.

Question 12 Écrire une fonction

```
val performance : bool_expr -> decision_tree -> int -> float
```

telle que `performance e t vars` calcule le taux de correspondance entre une expression et un arbre de décision sur l'ensemble des valuations de `vars` variables.

Question 13 En déduire une fonction

```
val test : int -> int -> int -> float array
```

telle que l'appel `test size vars max_samples` applique la stratégie de tests précédente pour une formule de `size` nœuds sur les variables `< vars` et en ayant `k` tests pour `k` allant de 0 à `max_samples - 1`. On renverra les résultats sous la forme d'un tableau.

Question 14 On pourra réaliser des graphes pour mesurer l'impact du nombre de tests sur la performance.



Remarque 26.1 Par exemple, pour des formules à 10 nœuds sur 8 variables, la correction obtient un taux de succès qui passe d'environ 50 % sans test à 80 %–95 % dès une dizaine de tests, puis progresse lentement. Les valeurs varient beaucoup d'une formule à l'autre.

26.5 Passer des arbres de décision aux formules et réciproquement

Il est possible de passer d'un arbre de décision à une formule et réciproquement.

Question 15 Écrire deux fonctions effectuant ces conversions.

Morpion

DANS CE CHAPITRE

- 27.1** Représentation des données
- 27.2** Construction du graphe des états
- 27.3** Calcul des attracteurs
- 27.4** Jouer

Dans ce TP, on va implémenter la stratégie parfaite de résolution du morpion par calcul des attracteurs.

27.1 Représentation des données

On va considérer des nombres ternaires pour représenter les plateaux du morpion. C'est assez inefficace car beaucoup de nombres ne vont pas correspondre à des parties atteintes, mais $3^9 = 19683$ n'étant pas très grand, il sera facile d'en faire un graphe ensuite.

On considère les types OCaml

```
type case = Vide | Croix | Rond
type etat = case array array
type joueur = J1 | J2
```

On va associer en ternaire 0 à une case vide, 1 à une croix et 2 à un rond. Un état est une matrice 3×3 de cases, on la représentera en ternaire ligne par ligne ainsi :

```
[| [| c0; c1; c2 |];
   [| c3; c4; c5 |];
   [| c6; c7; c8 |] |]
```

est représenté par le nombre $a_0 + 3a_1 + \dots + 3^8a_8$ où a_i est le chiffre associé à ci .

Question 1 Écrire des fonctions :

```
val nombre_vers_case : int -> case
val case_vers_nombre : case -> int
val nombre_vers_etat : int -> etat
val etat_vers_nombre : etat -> int
```

effectuant ces conversions.

Corrigé

```
let nombre_vers_case c =
  match c with
```

```

| 0 -> Vide
| 1 -> Croix
| _ -> Rond

let case_vers_nombre c =
  match c with
  | Vide -> 0
  | Croix -> 1
  | _ -> 2

let nombre_vers_etat n =
  let m = Array.make_matrix 3 3 Vide in
  let v = ref 0 in
  for i = 0 to 8 do
    m.(i / 3).(i mod 3) <- nombre_vers_case (!v mod 3);
    v := !v / 3
  done;
  m

let etat_vers_nombre m =
  let v = ref 0 in
  for i = 2 downto 0 do
    for j = 2 downto 0 do
      v := 3 * !v + case_vers_nombre m.(i).(j)
    done
  done;
  !v

```

Exemple 27.1

Le nombre 12410 correspond ainsi à `[|Rond; Rond; Croix|]; [|Vide; Vide; Vide|]; [|Rond; Rond; Croix|]`.

Question 2 Écrire une fonction

```
val compte_pions : etat -> int * int * int
```

qui renvoie un triplet (m, n, p) où m est le nombre de cases vides, n est le nombre de croix de l'état et p le nombre de ronds.

Corrigé

```

let compte_pions m =
  let vides = ref 0 in
  let croix = ref 0 in
  let ronds = ref 0 in
  for i = 0 to 2 do
    for j = 0 to 2 do
      match m.(i).(j) with
      | Croix -> incr croix
      | Rond -> incr ronds
      | _ -> incr vides
    done
  done;
  (!vides, !croix, !ronds)

```

Question 3 En déduire une fonction

```
val etat_controle : etat -> joueur
```

qui renvoie le joueur contrôlant un état, c'est-à-dire celui qui doit jouer. Le joueur 1 joue les croix et commence.

Corrigé

```
let etat_controle m =
  let _, croix, ronds = compte_pions m in
  if croix = ronds
  then J1
  else J2
```

Pour déterminer si un état est valide ou gagnant, il va être nécessaire de compter le nombre d'alignements de trois valeurs identiques.

Question 4 Écrire une fonction

```
val trois_en_ligne : etat -> case -> int
```

qui compte le nombre d'alignements de trois valeurs identiques égales à la valeur passée en paramètre.

Corrigé

```
let trois_en_ligne m c =
  let trois = ref 0 in
  for i = 0 to 2 do
    if m.(i).(0) = c && m.(i).(1) = c && m.(i).(2) = c
    then incr trois;
    if m.(0).(i) = c && m.(1).(i) = c && m.(2).(i) = c
    then incr trois
  done;
  if m.(0).(0) = c && m.(1).(1) = c && m.(2).(2) = c
  then incr trois;
  if m.(2).(0) = c && m.(1).(1) = c && m.(0).(2) = c
  then incr trois;
  !trois
```

Question 5 En déduire des fonctions

```
val etat_valide : etat -> bool
val etat_terminal : etat -> bool
val gagnant : etat -> joueur option
```

qui vont respectivement :

- déterminer si un état est valide, c'est-à-dire si on l'atteint naturellement depuis la grille vide en commençant par le joueur Croix et sans continuer à jouer dans une partie gagnée ;
- déterminer si un état est terminal, c'est-à-dire qu'il est valide et qu'un joueur a gagné ou qu'il n'y a plus de cases libres ;
- étant donné un état terminal, renvoyer le gagnant s'il y en a un, ou None sinon.

Corrigé

Un état est valide quand il y a autant de croix que de ronds, ou une croix de plus, et que le dernier joueur à avoir joué est le seul à pouvoir avoir un alignement.

```
let etat_valide m =
  let vides, croix, ronds = compte_pions m in
  (croix = ronds || croix = ronds+1)
  && (trois_en_ligne m Croix = 0 || croix = ronds+1)
  && (trois_en_ligne m Rond = 0 || croix = ronds)

let etat_terminal m =
  let vides, _, _ = compte_pions m in
  etat_valide m
```

```

&& (vides = 0 || trois_en_ligne m Croix >= 1 || trois_en_ligne m Rond >= 1)

let gagnant m =
  assert (etat_terminal m);
  if trois_en_ligne m Croix >= 1
  then Some J1
  else if trois_en_ligne m Rond >= 1
  then Some J2
  else None

```

On obtient ainsi 5478 états valides, dont 958 terminaux.

27.2 Construction du graphe des états

Question 6 Écrire une fonction

```
val etat_copie : etat -> etat
```

qui copie un état (attention, ce sont des matrices).

Corrigé

```

let etat_copie m =
  Array.init 3 (fun i -> Array.copy m.(i))

```

Question 7 Écrire une fonction

```
val suivants : etat -> etat list
```

qui renvoie la liste des états accessibles depuis un état donné par un coup du joueur qui le contrôle.

Si l'état est terminal, on renverra la liste vide.

Corrigé

```

let suivants m =
  let c = match etat_controle m with J1 -> Croix | J2 -> Rond in
  let s = ref [] in
  if not (etat_terminal m)
  then
    for i = 0 to 2 do
      for j = 0 to 2 do
        if m.(i).(j) = Vide
        then begin
          let m' = etat_copie m in
          m'.(i).(j) <- c;
          s := m' :: !s
        end
      done
    done;
  !s

```

Question 8 Écrire une fonction

```
val graphe_etats : unit -> int list array
```

qui renvoie la représentation en listes d'adjacence du graphe des états.

Le graphe comporte ainsi 3^9 sommets. On ne calculera pas les états suivants des états non valides, ils auront donc une liste de voisins vide.

Corrigé

```

let graphe_etats () =
  let netats = 19683 in
  let gr = Array.make netats [] in
  for e = 0 to netats - 1 do
    let m = nombre_vers_etat e in
    if etat_valide m
    then let ml = suivants m in
         gr.(e) <- List.map etat_vers_nombre ml
  done;
  gr

```

27.3 Calcul des attracteurs

Question 9 Écrire une fonction

```

val attracteur_initial : int list array -> joueur -> bool array

```

qui prend le graphe des états et un joueur et renvoie un tableau de booléens indiquant si un état est terminal et gagnant pour ce joueur.

Corrigé

```

let attracteur_initial g j =
  let n = Array.length g in
  Array.init n
    (fun e -> let m = nombre_vers_etat e in
              etat_terminal m && gagnant m = Some j)

```

Question 10 Écrire une fonction

```

val attracteur_suivant : int list array -> joueur -> bool array -> bool

```

qui prend le graphe des états, un joueur, un attracteur A_i donné sous la forme d'un tableau de booléens indiquant si $e \in A_i$, place dans ce tableau A_{i+1} et renvoie un booléen indiquant si $A_i \neq A_{i+1}$.

Corrigé

Un état valide non terminal est ajouté s'il est contrôlé par j et qu'un de ses successeurs est dans l'attracteur, ou s'il est contrôlé par l'adversaire et que tous ses successeurs sont dans l'attracteur.

```

let attracteur_suivant g j a =
  let nouveaux = ref false in
  let n = Array.length g in
  for e = 0 to n-1 do
    if not a.(e) && etat_valide (nombre_vers_etat e)
    && not (etat_terminal (nombre_vers_etat e))
    then begin
      let controleur = etat_controle (nombre_vers_etat e) in
      if controleur = j
      && List.exists (fun e' -> a.(e')) g.(e)
      then (a.(e) <- true; nouveaux := true);
      if controleur <> j
      && List.for_all (fun e' -> a.(e')) g.(e)
      then (a.(e) <- true; nouveaux := true);
    end
  done;
  !nouveaux

```

Comme on modifie le tableau pendant le parcours, un état ajouté peut servir à en ajouter d'autres lors du même appel : on obtient un ensemble qui contient A_{i+1} et qui est inclus dans l'attracteur. Cela ne change pas le résultat de la question suivante.

Question 11 En déduire une fonction

```
val positions_gagnantes : int list array -> joueur -> bool array
```

qui renvoie un tableau de booléens indiquant si un état est une position gagnante.

Corrigé

```
let positions_gagnantes g j =
  let a = attracteur_initial g j in
  while attracteur_suivant g j a do
    ()
  done;
  a
```

On constate que l'état initial n'est gagnant pour aucun des deux joueurs : le morpion bien joué aboutit à un match nul.

27.4 Jouer

Question 12 En déduire une fonction

```
val joue_bien : int list array -> bool array -> bool array -> int -> int
```

qui prend un graphe d'états, le tableau des positions gagnantes d'un joueur, le tableau des positions gagnantes de l'adversaire et un état non terminal e qu'il contrôle.

Si e est gagnant pour le joueur, on renvoie un coup encore gagnant pour le joueur. Sinon, on essaie de renvoyer un coup qui n'est pas gagnant pour l'adversaire (on vise le nul).

Corrigé

```
let joue_bien g pj po e =
  let l = List.filter (fun e -> pj.(e)) g.(e) in
  let l' = List.filter (fun e -> not po.(e)) g.(e) in
  match l, l', g.(e) with
  | t::_ , _ , _
  | [], t::_ , _
  | [], [], t::_ -> t
  | _ -> failwith "etat terminal"
```

Question 13 On peut maintenant écrire une petite boucle interactive pour jouer au morpion contre notre IA.

Corrigé

Le joueur humain joue les croix. Si son coup termine la partie, l'IA ne joue pas.

```
let affiche_etat m =
  for i = 0 to 2 do
    for j = 0 to 2 do
      let c = match m.(i).(j) with
      | Croix -> 'X'
      | Rond -> 'O'
      | Vide -> ' ' in
      print_char c;
```



```

        if j <> 2 then print_char '|'
      done;
      print_newline ()
    done;
    print_newline ()

let boucle_interactive () =
  let g = graphe_etats () in
  let w1 = positions_gagnantes g J1 in
  let w2 = positions_gagnantes g J2 in

  let p = ref 0 in
  while not (etat_terminal (nombre_vers_etat !p)) do
    affiche_etat (nombre_vers_etat !p);
    let l = g.(!p) in
    List.iteri (fun i e ->
      Printf.printf "Coup %d) %s%s%s :\n"
        i
        (if w1.(e) then "gagnant pour vous" else "")
        (if w2.(e) then "gagnant pour l'adversaire" else "")
        (if not w1.(e) && not w2.(e) then "au mieux nul" else ""));
    affiche_etat (nombre_vers_etat e)) l;
    Printf.printf "Saisir numero coup> ";
    flush stdout;
    let i = Scanf.scanf " %d" (fun i -> i) in
    Printf.printf "%d\n" i;
    let e' = List.nth l i in
    if etat_terminal (nombre_vers_etat e')
    then p := e'
    else p := joue_bien g w2 w1 e'
  done;
  affiche_etat (nombre_vers_etat !p);
  match gagnant (nombre_vers_etat !p) with
  | Some J1 -> print_endline "Vous avez gagné !"
  | Some J2 -> print_endline "Vous avez perdu !"
  | None -> print_endline "Match nul."

```


Othorro

DANS CE CHAPITRE

- 28.1 Description du jeu Othorro
- 28.2 Représentation en OCaml et implémentation des règles
- 28.3 Évaluation et stratégie gloutonne
- 28.4 Minmax et élagage α - β
- 28.5 Exemples de sorties
- 28.6 Exemple de partie

28.1 Description du jeu Othorro

Dans ce TP, on va faire une intelligence artificielle pour une variante du jeu *Othello* : le jeu **Othorro** !

En cas de doute, vous pouvez consulter [les règles d'Othello ici](#).

Othorro est un jeu à deux joueurs, noir et blanc, qui jouent tour à tour sur un plateau de $n \times n$ cases qui contient initialement 4 pions en haut à gauche. Exemple pour un plateau de taille 6 :

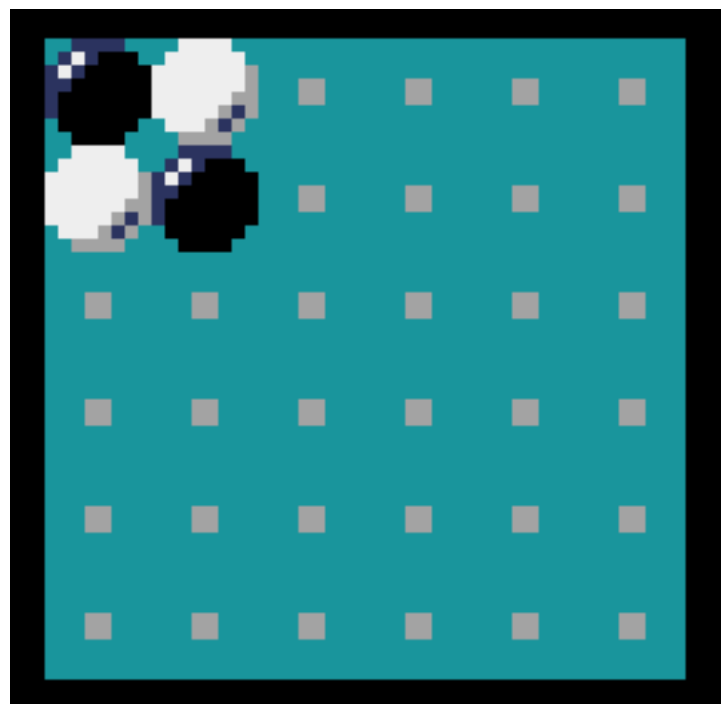


Figure 28.1 — Plateau initial de taille 6

En commençant par le joueur noir, chaque joueur place un pion, ou passe si c'est impossible, en faisant en sorte d'entourer des pions adverses entre deux de ses pions. Tous les pions entourés **dans les huit directions** (haut, bas, gauche, droite et les quatre diagonales) sont alors remplacés par des pions de la couleur du joueur courant.

Exemple : le coup central ici par noir va tout prendre :

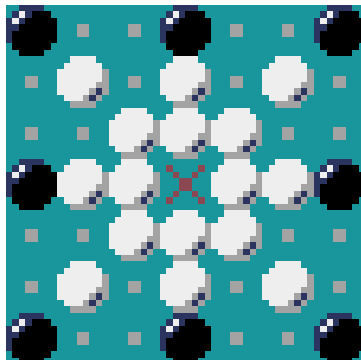


Figure 28.2 — Avant le coup central

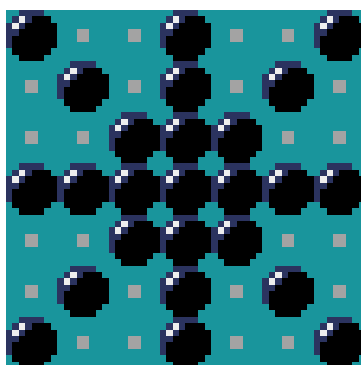


Figure 28.3 — Après le coup central

Dans **Othorro**, le plateau n'est pas un carré mais un **tore** ! Cela signifie qu'il n'y a pas de bords : si on sort par le côté gauche, on revient par le côté droit, même chose pour les directions verticales.

Voici indiqués sur le plateau par des croix les coups possibles du joueur noir, le premier joueur, sur le plateau initial :

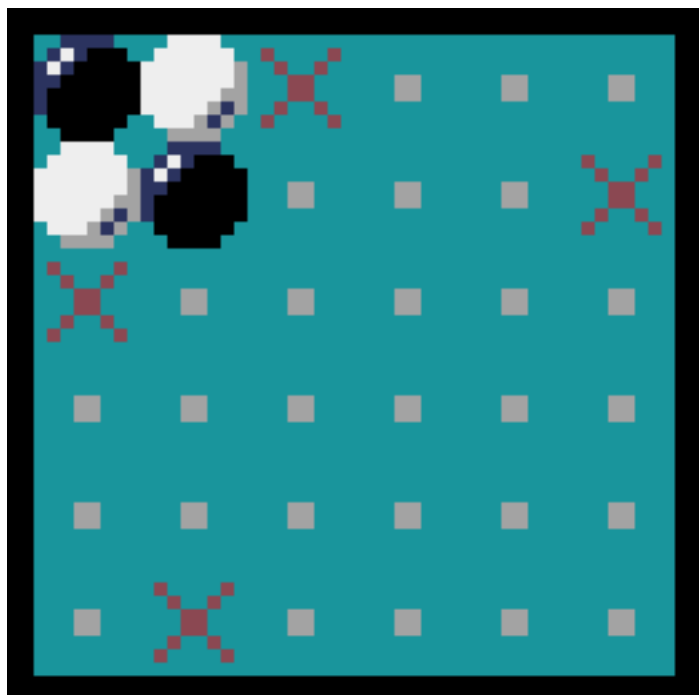


Figure 28.4 — Coups possibles de noir sur le plateau initial

S'il choisit de faire le coup en haut à droite, sur la deuxième ligne, on obtient alors le plateau suivant, sur lequel on a indiqué par des croix les coups possibles du joueur blanc :

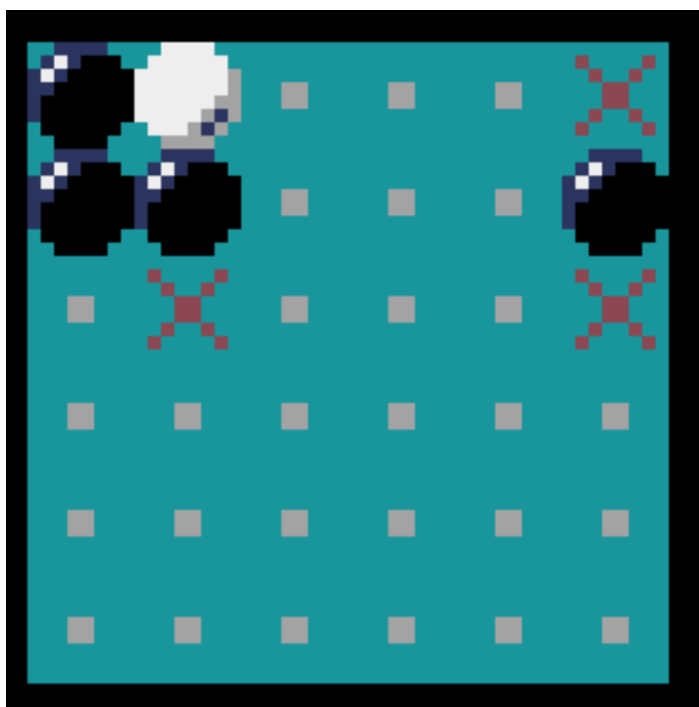


Figure 28.5 — Coups possibles de blanc après le premier coup de noir

Ici, ce sont, mis à part les conditions du tore, les mêmes règles qu'Othello. On rajoute une nouvelle règle : on permet de faire des coups circulaires en rajoutant un pion dans une ligne complète de pions de la couleur adverse :

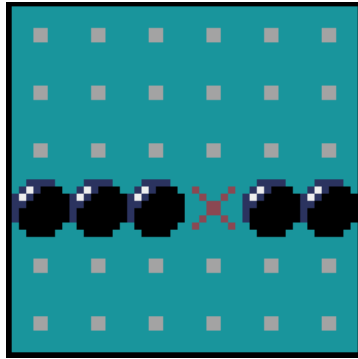


Figure 28.6 — Ligne de pions adverses avec une case libre

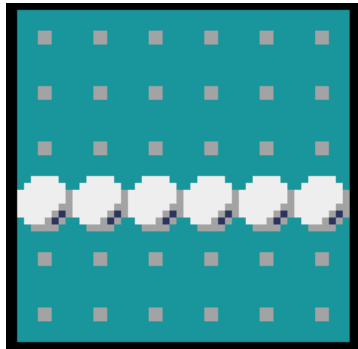


Figure 28.7 — Toute la ligne est retournée

La partie se termine quand aucun joueur ne peut jouer, le gagnant est alors celui qui a le plus de pions de sa couleur.

28.2 Représentation en OCaml et implémentation des règles

On va considérer les types

```
type joueur = Blanc | Noir
type case = Vide | Pion of joueur
type plateau = case array array
```

Un plateau est **toujours** une matrice carrée, donc il suffit de calculer la taille du array englobant pour en déduire sa dimension. La case $p.(x).(y)$ est sur la ligne x et la colonne y .

Question 1 Écrire une fonction

```
val plateau_initial : int -> plateau
```

qui prend en entrée une taille $t \geq 2$ et crée un plateau avec la configuration initiale des pions.

Question 2 *Optionnel, mais ça vous sera utile.*

Écrire une fonction

```
val plateau_affiche : plateau -> unit
```

qui affiche un plateau sous une représentation textuelle lisible, comme :

```
#0...
0#...
.....
.....
.....
.....
```

Question 3 Écrire une fonction

```
val opposant : joueur -> joueur
```

qui renvoie l'opposant du joueur donné.

Question 4 Écrire des fonctions

```
val plateau_case : plateau -> int -> int -> case
val plateau_place : plateau -> int -> int -> case -> unit
```

qui renvoient, ou modifient, la case associée aux coordonnées passées en entrée. Ces fonctions peuvent prendre des coordonnées qui ne sont pas dans $\llbracket 0, t-1 \rrbracket^2$, où t est la taille du plateau, compte tenu du fait qu'on est sur un tore.

Indication : $(-3) \bmod 5$ vaut -3 en OCaml, car le reste r de la division de a par b y vérifie seulement $|r| < |b|$ et a le signe de a . Pour avoir un nombre positif, on peut rajouter des valeurs, ici $(-3 + 5) \bmod 5$ qui vaut bien 2. C'est une remarque cruciale pour les bords.

On supposera ici que les valeurs v données sont toujours telles que $v + t \geq 0$.

Question 5 Écrire une fonction

```
val coups : plateau -> joueur -> (int * int) list
```

qui renvoie la liste des coups, c'est-à-dire la liste des positions possibles où le joueur donné peut placer un pion. Compte tenu des remarques précédentes, ces positions n'ont pas besoin d'être dans $\llbracket 0, t-1 \rrbracket^2$.

Question 6 Écrire une fonction

```
val plateau_copie : plateau -> plateau
```

qui effectue une copie complète d'un plateau. C'est-à-dire que si on modifie le plateau copie, on ne change rien au plateau d'origine.

Question 7 Écrire une fonction

```
val joue_coup : plateau -> joueur -> int * int -> plateau
```

qui prend en entrée un plateau, un joueur et un coup valide, et l'applique sur le plateau, c'est-à-dire place le pion et retourne tous les pions adverses qu'il faut retourner.

Cette fonction ne modifie pas le plateau en entrée mais renvoie un **nouveau** plateau.

28.3 Évaluation et stratégie gloutonne**Question 8** Écrire une fonction

```
val mobilite : joueur -> plateau -> int
```

qui renvoie la différence entre le nombre de coups du joueur sur le plateau donné et le nombre de coups de son adversaire. On parle de score de mobilité.

Question 9 Écrire une fonction

```
val materiel : joueur -> plateau -> int
```

qui renvoie la différence entre le nombre de pions du joueur et le nombre de pions de son adversaire. On parle de score matériel.

Question 10 En déduire une fonction

```
val eval : joueur -> plateau -> joueur -> int
```

où `eval j p j'` renvoie la somme des scores de mobilité et matériel vis-à-vis du joueur `j`. Le paramètre `j'` sera ici ignoré, mais il sera utile dans la suite.

Cette fonction sera celle qu'on va utiliser pour les stratégies du cours.

Question 11 Écrire une fonction

```
val joue : (joueur -> plateau -> joueur -> int) ->
          plateau -> joueur -> plateau option
```

où `joue eval p j` renvoie :

- `None` si le joueur `j` n'a pas de coup possible dans le plateau `p` ;
- `Some p'` sinon, où `p'` est, de tous les plateaux atteints, celui qui a la plus grande valeur `eval j p' j'` où `j'` est l'opposant de `j`. En cas d'égalité, on garde le premier coup dans l'ordre de la liste renvoyée par coups.

Question 12 Écrire une fonction

```
val combat : int ->
            (joueur -> plateau -> joueur -> int) ->
            (joueur -> plateau -> joueur -> int) -> plateau
```

où `combat taille eval_noir eval_blanc` va créer un plateau initial et faire jouer tour à tour noir et blanc selon les fonctions d'évaluation passées en entrée, avec la fonction `joue` précédente.

Quand les deux joueurs passent successivement, la partie s'arrête et la fonction renvoie le plateau correspondant.

Note : je vous conseille d'afficher la suite des plateaux pour pouvoir mettre au point le code ensuite.

Question 13 Écrire une fonction

```
val gagnant : plateau -> joueur
```

qui renvoie le joueur gagnant sur un plateau (en cas d'égalité, c'est noir qui gagne).

28.4 Minmax et élagage α - β **Question 14** Écrire une fonction

```
val minmax : (joueur -> plateau -> joueur -> int) -> int
            -> joueur -> plateau -> joueur -> int
```

où `minmax eval prof j p courant` renvoie la valeur obtenue par la stratégie minmax visant à maximiser l'évaluation de `j` dans le cas où c'est au joueur courant de jouer dans le plateau `p`.

Le paramètre `prof` régit la profondeur de recherche : quand `prof` est nul, on renvoie l'évaluation directe de la partie.



Remarque 28.1 On peut alors utiliser `minmax eval prof` comme fonction d'évaluation pour `joue` : c'est pour cela que `eval` prend en dernier paramètre le joueur qui doit jouer. Il faut aussi penser au cas où courant doit passer, sans que la partie soit terminée pour autant.

Question 15 Écrire une fonction


```
val alpha_beta : (joueur -> plateau -> joueur -> int) -> int
  -> joueur -> plateau -> joueur -> int
```

qui procède comme la précédente mais en utilisant l'élagage α - β . Ici, ces deux paramètres ne sont pas présents, on pourra les faire apparaître dans une fonction auxiliaire récursive.

On pourra s'inspirer du pseudo-code présent sur [la page Wikipédia](#).

28.5 Exemples de sorties

On vous donne la sortie de la correction pour les commandes suivantes :

```
let p = plateau_initial 5
val p : plateau =
  [| [| Pion Noir; Pion Blanc; Vide; Vide; Vide|];
    [| Pion Blanc; Pion Noir; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|] |]

let _ = plateau_case p 0 1
- : case = Pion Blanc

let _ = plateau_case p (-5) 6
- : case = Pion Blanc

let _ = plateau_place p 0 1 (Pion Noir); p
- : plateau =
  [| [| Pion Noir; Pion Noir; Vide; Vide; Vide|];
    [| Pion Blanc; Pion Noir; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|] |]

let _ = coups p Blanc
- : (int * int) list = [(1, 2); (4, 0); (4, 2)]

let _ = coups p Noir
- : (int * int) list = [(1, 4); (2, 0); (2, 4)]

let _ = joue_coup p Blanc (4,0)
- : plateau =
  [| [| Pion Blanc; Pion Noir; Vide; Vide; Vide|];
    [| Pion Blanc; Pion Noir; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Pion Blanc; Vide; Vide; Vide; Vide|] |]

let _ = mobilite Noir p
- : int = 0

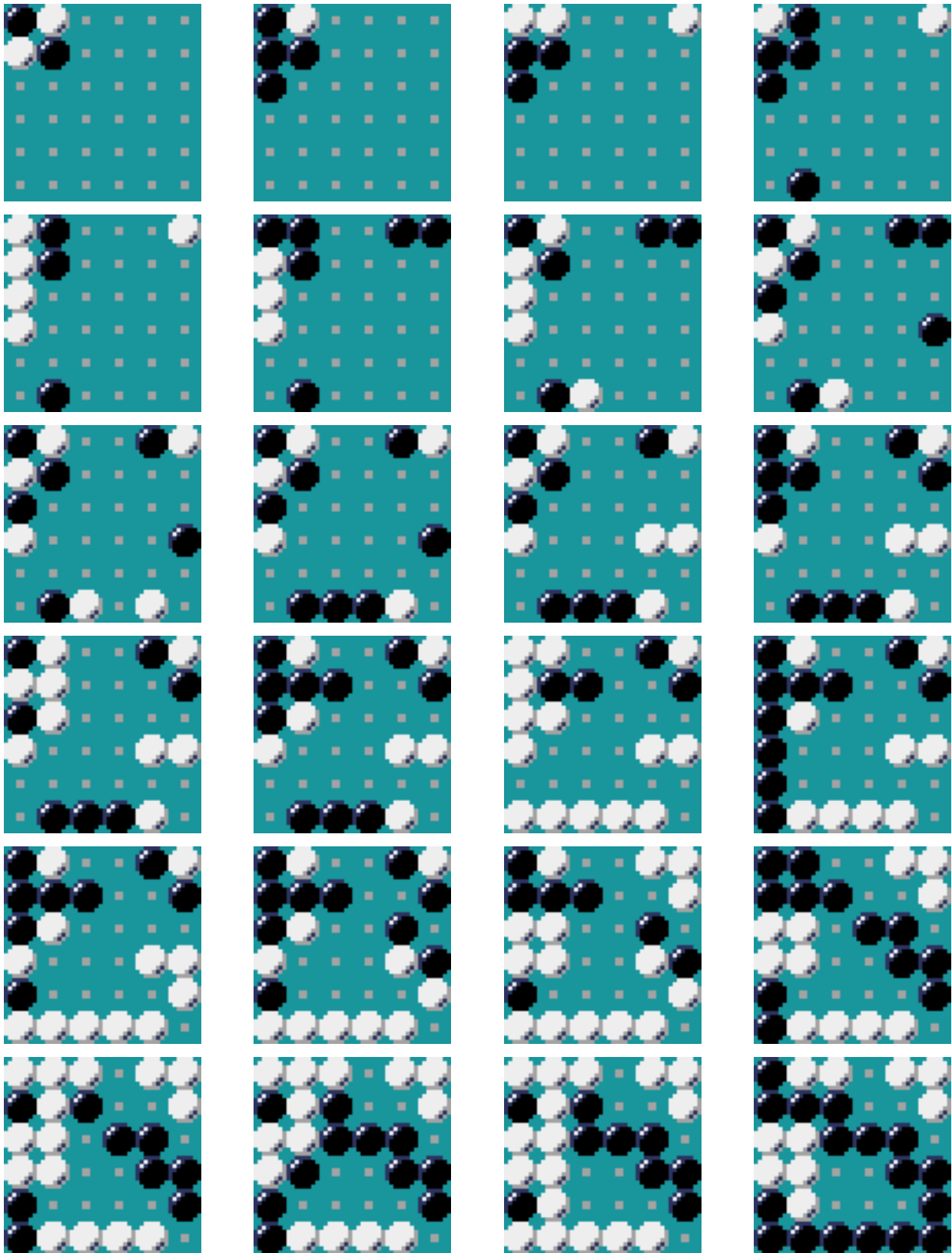
let _ = materiel Noir p
- : int = 2

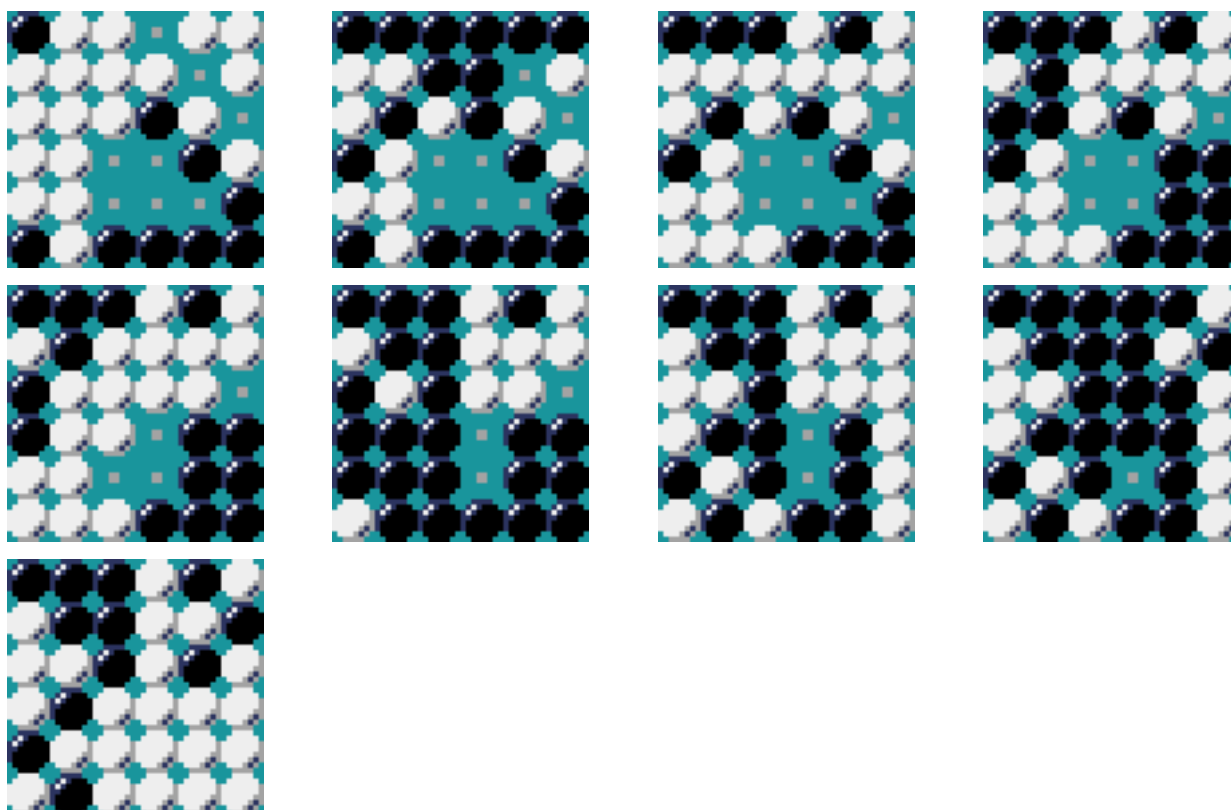
let _ = evaluate Noir p Blanc
- : int = 2

let _ = joue evaluate p Noir
- : plateau option =
Some
  [| [| Pion Noir; Pion Noir; Vide; Vide; Vide|];
    [| Pion Noir; Pion Noir; Vide; Vide; Pion Noir|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|];
    [| Vide; Vide; Vide; Vide; Vide|] |]
```

Dans le dernier exemple, les trois coups possibles de noir donnent la même évaluation (5) : c'est le premier, (1, 4), qui est retenu.

28.6 Exemple de partie





Résolution du 15-puzzle

DANS CE CHAPITRE

- 29.1** Présentation du puzzle et fonctions sur le plateau
- 29.2** Files de priorité
- 29.3** Tables de hachage sur les plateaux
- 29.4** Algorithme A*
- 29.5** Stratégies par force brute
- 29.6** Optimisations

Dans ce TP, on propose l'architecture suivante sur plusieurs fichiers :

- `main.c` contient le lancement des résolutions ;
- `plateau.(h|c)` contiennent le type des plateaux et les fonctions liées au graphe implicite des mouvements ;
- `tablehachage.(h|c)` des tables de hachage en adressage ouvert (sondage linéaire) avec pour clés des plateaux et des valeurs associées quelconques ;
- `liste.(h|c)` une implémentation des listes chaînées avec ajout sur le dernier maillon en $O(1)$;
- `file.(h|c)` une implémentation des files avec des listes chaînées ;
- `fileprio.(h|c)` une implémentation des files de priorité avec des tas ;
- `bruteforce.(h|c)` du retour sur trace pour résoudre le puzzle ;
- `aetoile.(h|c)` l'algorithme A* et le parcours en largeur pour la résolution.

Tous les `.h` sont fournis, et on vous demande d'écrire les `.c`. Ce document décrit ce qui est attendu des différentes fonctions.

On pourra ainsi récupérer l'archive `TP_15puzzle.zip` qui contient les `.h`. Les `.c` solutions sont dans le répertoire `solutions` de l'archive et vous pouvez les récupérer pour sauter certaines parties.

29.1 Présentation du puzzle et fonctions sur le plateau

On va écrire ici `plateau.c`.

Le 15-puzzle (ou taquin) est un puzzle constitué de 15 tuiles numérotées de 1 à 15 et d'une case vide agencées dans un carré 4×4 .

Le puzzle résolu correspond au plateau suivant, la case vide étant en gris :

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 29.1 — Plateau résolu ; la case vide est en gris.

Un mouvement valide consiste à déplacer une tuile voisine de la case vide dans celle-ci, ce qui place la case vide à sa place.

Les deux mouvements valides sur le plateau précédent mènent à :

1	2	3	4
5	6	7	8
9	10	11	12
13	14		15

Figure 29.2 — Le 15 a glissé vers la droite.

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

Figure 29.3 — Le 12 a glissé vers le bas.

Un plateau se représente naturellement par un tableau de 16 entiers, la valeur 0 étant donnée pour la case vide, en lisant le plateau ligne par ligne. Le plateau résolu est donc le tableau { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0 }.

On va faire beaucoup de manipulations de plateaux, et il est plus simple d'avoir des plateaux immuables passés par copie. Pour cela, on pourrait les définir uniquement comme des tableaux statiques :

```
#define N 4 // pour pouvoir changer la taille
typedef int plateau[N*N];
```

Problème : tout passage d'un plateau en paramètre se fera alors par référence, en tant que pointeur, et sans aucune copie du tableau lui-même.

Il existe une astuce classique en C : les structures peuvent être passées par valeur et recopiées. On va donc définir :

```
#define N 4
struct plateau {
    int t[N*N];
};
typedef struct plateau plateau;
```

Cela permettra d'avoir directement des plateau comme argument ou valeur de retour sans poser aucun problème.

Question 1 Écrire une fonction

```
void plateau_affiche(plateau p);
```

qui affiche un plateau sur la sortie standard.

Question 2 Écrire une fonction

```
plateau plateau_resolu(void);
```

qui renvoie le plateau résolu.

Question 3 Écrire une fonction

```
bool plateau_eq(plateau p1, plateau p2);
```

qui teste si deux plateaux sont égaux.

Question 4 Écrire une fonction

```
int position_vide(plateau p);
```

qui renvoie l'indice de la case vide.

Question 5 Écrire une fonction

```
bool mouvement_valide(plateau p, int src, int dst);
```

qui indique si échanger les cases d'indices `src` et `dst` est un mouvement valide, où `src` est l'indice de la case vide.

Question 6 Écrire une fonction

```
plateau mouvement_effectue(plateau p, int src, int dst);
```

qui effectue l'échange de deux cases. On suppose que le mouvement est valide.

Question 7 Écrire une fonction

```
void mouvements(plateau p, int *mouv, int *n);
```

qui prend en entrée un pointeur sur un tableau de 4 entiers et place dedans les indices des cases qu'on peut échanger avec la case vide, en indiquant, par passage par référence, le nombre de mouvements valides dans l'entier `n` passé par pointeur.

Question 8 Écrire une fonction

```
plateau mouvement_aleatoire(plateau p);
```

qui effectue un mouvement valide aléatoire et renvoie le nouveau plateau.

Question 9 Écrire une fonction

```
plateau plateau_melange(int n);
```

qui effectue `n` mouvements aléatoires depuis le plateau résolu et renvoie le plateau correspondant.

Remarque 29.1 En partant du plateau résolu, on est sûr d'obtenir un plateau qu'on peut résoudre en au plus n mouvements. Ce n'est pas le cas de n'importe quelle permutation des tuiles : on peut montrer que la moitié seulement des configurations sont accessibles depuis le plateau résolu.

29.2 Files de priorité

On va écrire ici `fileprio.c`.

On implémente ici des files de min-priorité avec un tas redimensionnable, où on double la capacité en cas de dépassement.

Pour cela, on considère les types :

```
struct element {
    T valeur;
    int priorite;
};

typedef struct element element;

struct fileprio {
    element *elements;
    unsigned int taille;
    unsigned int capacite;
};

typedef struct fileprio fileprio;
```

Le type T ne peut être polymorphe comme en OCaml, on a donc défini en tête de `fileprio.h` :

```
typedef plateau T;
```

Question 10 Écrire une fonction

```
fileprio *fileprio_cree(unsigned int capacite);
```

qui crée une file de priorité de capacité initiale `capacite`.

Question 11 Écrire une fonction

```
void fileprio_libere(fileprio *f);
```

qui libère la file de priorité `f`.

Question 12 Écrire une fonction

```
void fileprio_insere(fileprio *t, T valeur, int prio);
```

qui insère la valeur `valeur` avec la priorité `prio`.

Question 13 Écrire une fonction

```
element fileprio_extraire_minimum(fileprio *t);
```

qui extrait et renvoie le couple (`valeur`, `priorité`) de priorité minimale.

Note : ici, on aurait pu renvoyer un T et placer la priorité dans un pointeur optionnel.

29.3 Tables de hachage sur les plateaux

On va écrire ici `tablehachage.c`.

On va utiliser des plateaux comme clés, mais pour les valeurs associées, on propose du polymorphisme *light* à l'aide de `void *` :

```
struct baquet {
    bool rempli;
    plateau valeur;
    void *assoc;
};
typedef struct baquet baquet;

struct table {
    baquet *elements;
    unsigned int taille;
    unsigned int capacite;
};
typedef struct table table;
```

Et on fournit la fonction de hachage suivante sur les plateaux :

```
uint64_t plateau_hache(table *t, plateau p)
{
    uint64_t n = 0;
    uint64_t m = 1;
    for(int i = 0; i < N*N; i++)
    {
        n += p.t[i] * m;
        m *= N*N;
    }
    return n % t->capacite;
}
```

Remarque 29.2 Avant le modulo, n est l'entier dont l'écriture en base $N^2 = 16$ est donnée par les cases du plateau. Comme un plateau comporte 16 chiffres hexadécimaux, soit 64 bits, deux plateaux différents donnent deux valeurs de n différentes : les collisions ne viennent que du modulo.

Attention : on supposera que les valeurs `void *assoc` ne sont pas possédées par la table. C'est donc à l'utilisateur de s'assurer de les libérer. C'est la raison pour laquelle la fonction de remplacement ci-dessous va renvoyer l'ancienne valeur.

Question 14 Écrire une fonction

```
table *table_cree(int capacite);
```

qui crée une table de hachage de capacité initiale `capacite`.

Question 15 Écrire une fonction

```
void table_libere(table *t, bool libere_assoc);
```

qui libère la table `t`, ainsi que les valeurs associées si `libere_assoc` est vrai.

Question 16 Écrire une fonction

```
bool table_contient(table *t, plateau p);
```

qui indique si le plateau `p` est une clé de la table `t`.

Question 17 Écrire une fonction

```
void table_ajoute(table *t, plateau p, void *assoc);
```

qui ajoute une association.

Si la taille de la table dépasse la moitié de la capacité, on doublera la capacité en créant une nouvelle table et en réinsérant toutes les paires.

Question 18 Écrire une fonction

```
void *table_remplace(table *t, plateau p, void *assoc);
```

qui remplace l'association si p est dans la table et renvoie l'ancienne valeur. Si p n'est pas dans la table, cette fonction appellera juste `table_ajoute` et renverra `NULL`.

Question 19 Écrire une fonction

```
void *table_cherche(table *t, plateau p);
```

qui renvoie la valeur `void *` associée au plateau p .

29.4 Algorithme A*

On va écrire ici `aetoile.c`.

Question 20 Écrire une fonction

```
void resout_aetoile(plateau p);
```

qui résout le puzzle avec l'algorithme A*.

On affichera la solution sous la forme de la succession des plateaux et on indiquera la longueur de la solution.

On pourra procéder ainsi :

- définir une fonction d'heuristique qui compte le nombre de cases non vides mal placées ;
- on va utiliser une file de priorité ;
- pour stocker les distances minimales et les parents, on pourra utiliser une table de hachage avec des valeurs associées qui seront des `donnee *` où `donnee` est :

```
struct donnee {
    int distance;
    plateau parent;
};

typedef struct donnee donnee;
```



Remarque 29.3 Un mouvement déplace une seule tuile : il change donc le nombre de tuiles mal placées d'au plus 1. Cette heuristique ne surestime jamais le nombre de mouvements restants (chaque tuile mal placée doit bouger au moins une fois) et vérifie $h(p) \leq 1 + h(p')$ pour tout mouvement de p vers $p' : A^*$ renvoie donc bien une solution de longueur minimale. La somme des distances de Manhattan de chaque tuile à sa place est une heuristique plus précise qui a les mêmes propriétés.

29.5 Stratégies par force brute

On va écrire ici `bruteforce.c`.

Question 21 Écrire une fonction

```
void resout_retoursurtrace(plateau p, int recursion_limit);
```

qui trouve une solution par retour sur trace sans jamais descendre dans l'arbre à une profondeur plus grande que `recursion_limit`.

On affichera la solution.

Question 22 Écrire une fonction

```
int resout_retoursurtrace_meilleur(plateau p, int recursion_limit);
```

qui renvoie la longueur minimale d'une solution par retour sur trace sans jamais descendre dans l'arbre à une profondeur plus grande que `recursion_limit`.

Si on ne peut trouver une solution dans cette limite, on renverra -1.

Note : ici, pas la peine d'afficher la solution, ce qui demanderait beaucoup plus de travail.

29.6 Optimisations**Question 23** Écrire une fonction

```
int resout_retoursurtrace_meilleur_borne(plateau p, int recursion_limit);
```

qui optimise la fonction précédente en ne descendant jamais plus profondément que la profondeur minimale d'une solution déjà trouvée.

Question 24 À rajouter dans `aetoile.c` pour avoir les fonctions communes.

Écrire une fonction

```
void resout_largeur(plateau p);
```

qui résout le puzzle par un parcours en largeur.

Note : on pourra implémenter `liste.c` et `file.c`, et utiliser le type des tables de hachage en plaçant directement des entiers castés en `void *` pour les distances.



Remarque 29.4 Pour comparer les deux approches : sur un plateau obtenu par 40 mouvements aléatoires dont la solution optimale est de longueur 16, le parcours en largeur de la correction a évalué environ 440 000 positions contre 166 pour A*.

PARTIE VII

Problèmes

30	Problème	Devoir calculabilité	319
31	Problème	Devoir - Logique et Grammaire	327
32	Problème	Exercice - Tableaux autoréférents	337

PROBLÈME 30

Devoir calculabilité

30.1 Autour de SubsetSum

Dans cette partie, tous les programmes sont à écrire en C.

On considère le problème Fira Sans (*SubsetSum*) suivant :

Problème 30.1 — : SubsetSum

Entrées : un ensemble fini $X \subset \mathbb{N}^*$ et $T \in \mathbb{N}$.

Sortie : existe-t-il $V \subset X$ tel que $\sum_{x \in V} x = T$?

Les notions de problème de décision, de classe NP et de réduction polynomiale sont celles du chapitre de calculabilité.

30.1.1 Vérification

Question 1 Écrire une fonction de prototype

```
bool subsetsum_solution(int *val, int nval, int tgt, bool *sol);
```

qui prend en entrée X donné sous la forme d'un tableau `val` de `nval` entiers et une solution donnée sous la forme d'un tableau `sol` de `nval` booléens où `sol[i]` vaut `true` si `val[i]` appartient à l'ensemble V , et renvoie un booléen indiquant si la somme des éléments de V vaut `tgt`.

Corrigé

```
bool subsetsum_solution(int *val, int nval, int tgt, bool *sol)
{
    int s = 0;
    for (int i = 0; i < nval; i++)
    {
        if(sol[i])
            s += val[i];
    }
    return s == tgt;
}
```

Question 2 Que peut-on en déduire pour le problème Fira Sans (*SubsetSum*) ?

Corrigé

La solution donnée sous la forme d'un tableau de booléens est donc un certificat de taille polynomiale (ici linéaire : un bit par élément de X) en la taille de l'entrée, dont la vérification peut s'effectuer en temps polynomial : la fonction précédente fait n additions d'entiers dont la taille est majorée par celle de l'entrée.

On a donc $\text{Fira Sans}(\text{SubsetSum}) \in \text{NP}$.

30.1.2 Recherche par force brute

Question 3 Écrire une fonction de prototype

```
void subsetsum_affiche(int *val, int nval, bool *sub);
```

qui prend en entrée X donné sous la forme d'un tableau val de nval entiers et un sous-ensemble $V \subset X$ donné sous la forme d'un tableau sub de nval booléens où $\text{sub}[i]$ vaut `true` si $\text{val}[i]$ appartient à l'ensemble V , et affiche V sur une ligne.

Exemple :

```
int val[] = { 1, 2, 4, 10 };
bool sub[] = { true, false, false, true };
subsetsum_affiche(val, 4, sub);
```

va afficher

```
1 10
```

Corrigé

```
void subsetsum_affiche(int *val, int nval, bool *sub)
{
    for(int i = 0; i < nval; i++)
    {
        if(sub[i])
            printf("%d ", val[i]);
    }
    printf("\n");
}
```

Question 4 Écrire une fonction de prototype

```
void subsetsum_cherche_rec(int *val, int nval,
    int tgt, bool *sub, int i);
```

qui va considérer l'ensemble V décrit comme dans les questions précédentes mais avec

$$V \subset \{\text{val}[k] \mid 0 \leq k < i\}$$

et affiche récursivement $V \cup V'$ pour toutes les solutions

$$V' \subset \{\text{val}[k] \mid i \leq k < \text{nval}\}$$

à $\text{Fira Sans}(\text{SubsetSum})$ pour la cible tgt .

Note : il s'agit donc de faire une simple recherche récursive par retour sur trace.

Corrigé

```
void subsetsum_cherche_rec(int *val, int nval,
    int tgt, bool *sub, int i)
{
```



```

    if (i == nval) {
        if (subsetsum_solution(val, nval, tgt, sub))
            subsetsum_affiche(val, nval, sub);
    } else {
        sub[i] = true;
        subsetsum_cherche_rec(val, nval, tgt, sub, i+1);
        sub[i] = false;
        subsetsum_cherche_rec(val, nval, tgt, sub, i+1);
    }
}

```

Question 5 En déduire une fonction de prototype

```
void subsetsum_resout(int *val, int nval, int tgt);
```

qui affiche tous les sous-ensembles de val de somme tgt.

Corrigé

Il n'est pas nécessaire d'initialiser sub : chaque case sub[i] est affectée avant l'appel de rang i+1, donc avant toute lecture.

```

void subsetsum_resout(int *val, int nval, int tgt)
{
    bool *sub = malloc(sizeof(bool) * nval);
    subsetsum_cherche_rec(val, nval, tgt, sub, 0);
    free(sub);
}

```

Par exemple, avec val contenant 1, 2, ..., 10 et tgt valant 12, on obtient les 13 lignes 1 2 3 6, 1 2 4 5, 1 2 9, ..., 5 7.

Question 6 Quelle est la complexité temporelle en pire cas de cette fonction ?

Corrigé

On suppose qu'on a n valeurs.

On énumère tous les sous-ensembles (2^n feuilles dans l'arbre des appels récurrents, et $2^n - 1$ nœuds internes traités en temps constant) et chaque sous-ensemble est traité en $O(n)$ pour vérifier qu'il convient (et éventuellement l'afficher). L'algorithme est donc en $O(n2^n)$.

30.1.3 NP-complétude

Dans cette partie, on note $\mathcal{V}$ l'ensemble des variables propositionnelles et $B = \{\top, \perp\}$ l'ensemble des valeurs de vérité.

- On va montrer que Fira Sans(*SubsetSum*) est NP-complet en réduisant polynomialement Fira Sans(3SAT) à Fira Sans(*SubsetSum*). On admet que Fira Sans(3SAT) est NP-complet.

Problème 30.2 — : 3SAT

Entrées : une formule φ du calcul propositionnel donnée en forme normale conjonctive dont chaque clause comporte **exactement** trois littéraux.

Sortie : existe-t-il une valuation $v : \mathcal{V} \rightarrow B$ telle que $[\varphi]_v = \top$?

Question 7 Justifier qu'on peut supposer, sans perte de généralité, que la formule à résoudre par Fira Sans(3SAT) est telle qu'aucune clause ne comporte à la fois un littéral et son littéral opposé.

Corrigé

Une telle clause est toujours satisfaite, quelle que soit la valuation (elle contient $x \vee \neg x$) : on peut donc l'omettre sans changer la satisfiabilité de la formule. Supprimer ces clauses se fait en temps linéaire et la formule obtenue est encore une entrée de Fira Sans(3SAT).

On considère une formule $\varphi = C_0 \wedge \dots \wedge C_{n-1}$ où $\text{Var}(\varphi) = \{x_0, \dots, x_{p-1}\}$ est l'ensemble des variables dont un littéral associé est présent dans au moins une clause. En vertu de la question précédente, on suppose également qu'aucune clause ne contient à la fois un littéral et son littéral opposé.

On associe :

- à chaque variable x_i , deux entiers a_i, a'_i définis par :

$$a_i = 10^i + \sum \{10^{p+j} \mid x_i \text{ littéral de } C_j, j \in \llbracket 0, n-1 \rrbracket\}$$

$$a'_i = 10^i + \sum \{10^{p+j} \mid \neg x_i \text{ littéral de } C_j, j \in \llbracket 0, n-1 \rrbracket\}$$

- à chaque clause C_j , deux entiers $c_j = 10^{p+j}$ et $c'_j = 2 \cdot 10^{p+j}$.

On pose alors $X(\varphi) = \{a_0, a'_0, \dots, a_{p-1}, a'_{p-1}, c_0, c'_0, \dots, c_{n-1}, c'_{n-1}\}$ et

$$T(\varphi) = \sum_{i=0}^{p-1} 10^i + 4 \sum_{j=0}^{n-1} 10^{p+j}$$

Question 8 On considère la formule $\varphi = C_0 \wedge C_1 \wedge C_2 \wedge C_3$ où $C_0 = x_0 \vee \neg x_1 \vee \neg x_2$, $C_1 = \neg x_0 \vee \neg x_1 \vee \neg x_2$, $C_2 = \neg x_0 \vee \neg x_1 \vee x_2$ et $C_3 = x_0 \vee x_1 \vee x_2$.

- Calculer $X(\varphi)$ et $T(\varphi)$.
- Résoudre Fira Sans(*SubsetSum*) pour cette entrée.
- En déduire, si c'est possible, une valuation satisfaisant φ .

Corrigé

- Ici $p = 3$ et $n = 4$. On a

$$a_0 = 1001001 \quad a'_0 = 110001$$

$$a_1 = 1000010 \quad a'_1 = 111010$$

$$a_2 = 1100100 \quad a'_2 = 11100$$

$$\text{et } c_0 = 1000, c'_0 = 2000, c_1 = 10000, c'_1 = 20000, c_2 = 100000, c'_2 = 200000, c_3 = 1000000, c'_3 = 2000000.$$

$$T(\varphi) = 4444111.$$

- On trouve par exemple

$$\begin{aligned} 4444111 &= a_0 + a'_1 + a'_2 + c_3 + c'_3 + c_2 + c'_2 + c'_1 + c_0 \\ &= 1001001 + 111010 + 11100 + 1000000 \\ &\quad + 2000000 + 100000 + 200000 + 20000 + 1000 \end{aligned}$$

L'instance est donc positive. Une recherche exhaustive (avec `subsetsum_resout`) donne en fait quatre solutions, qui correspondent aux quatre valuations satisfaisant φ .

- On en déduit la valuation $v(x_0) = \top$ et $v(x_1) = v(x_2) = \perp$ (a_0, a'_1 et a'_2 sont dans la somme). On vérifie rapidement qu'elle satisfait φ : x_0 satisfait C_0 et C_3 , $\neg x_1$ satisfait C_1 et C_2 .

Question 9 Justifier que cette réduction est polynomiale.

Corrigé

Chaque entier s'écrit avec au plus $p + n$ chiffres et se calcule en temps linéaire en la taille de φ (un parcours des clauses), et il y en a $2p + 2n$, nombre linéaire en la taille de φ (car $p \leq 3n$), à calculer. De même pour $T(\varphi)$. La réduction est donc calculable en temps quadratique.

Remarquons que ces $2p + 2n$ entiers sont deux à deux distincts : a_i et a'_i sont les seuls à avoir un chiffre non nul en position $i < p$, et ils diffèrent car x_i apparaît dans au moins une clause et n'y apparaît pas avec son opposé ; c_j et c'_j n'ont qu'un chiffre non nul. L'ensemble $X(\varphi)$ a donc bien $2p + 2n$ éléments.

Question 10 Montrer que pour tout $V \subset X(\varphi)$, il n'y a pas de retenue sur les chiffres de l'écriture décimale en calculant $\sum_{v \in V} v$.

Corrigé

On remarque qu'il y a deux nombres ayant un 1×10^i dans leur écriture décimale pour $i < p$: a_i et a'_i . Donc ce chiffre vaut 0, 1 ou 2 dans toute somme.

Le chiffre en position $p + j$ vaut :

- 1 dans au plus trois a_i ou a'_i (un par littéral de C_j) ;
- 1 dans c_j ;
- 2 dans c'_j ;
- 0 dans tous les autres nombres.

On a donc au plus $3 + 1 + 2 = 6$ pour ce chiffre dans toute somme. Comme chaque colonne a une somme inférieure ou égale à $6 < 10$, il n'y a aucune retenue.

Ainsi, on pourra identifier le codage à des $(p + n)$ -uplets, si on le souhaite pour simplifier les preuves.

Question 11 Montrer que φ est satisfiable si et seulement s'il existe $V \subset X(\varphi)$ tel que $\sum_{v \in V} v = T(\varphi)$.

Corrigé

- $\Rightarrow$) Soit v une valuation satisfaisant φ , on considère V tel que

- $\forall i \in \llbracket 0, p-1 \rrbracket, a_i \in V \Leftrightarrow v(x_i) = \top$;
- $\forall i \in \llbracket 0, p-1 \rrbracket, a'_i \in V \Leftrightarrow v(x_i) = \perp$;
- pour chaque clause C_j ,

$$V \cap \{c_j, c'_j\} = \begin{cases} \{c_j\} & \text{si } C_j \text{ a trois littéraux satisfaits} \\ \{c'_j\} & \text{si } C_j \text{ a deux littéraux satisfaits} \\ \{c_j, c'_j\} & \text{sinon} \end{cases}$$

Comme la clause est satisfaite, ce « sinon » couvre uniquement le cas d'un seul littéral satisfait.

Par construction, le i -ème chiffre de $\sum V$ pour $i < p$ est exactement 1 car chaque variable a une unique valeur de vérité. Pour les clauses, comme il y a un 1 par littéral satisfait dans les a_i, a'_i , on rajoute exactement ce qu'il faut pour avoir 4 comme chiffre. Comme il n'y a pas de retenue, on a bien $\sum V = T(\varphi)$.

- $\Leftarrow$) S'il existe un tel ensemble V , comme il n'y a pas de retenue, chaque chiffre de $\sum V$ est la somme des chiffres correspondants des éléments de V . Pour tout $i \in \llbracket 0, p-1 \rrbracket$, on a forcément $a_i \in V$ ou $a'_i \in V$ mais pas les deux, car il y a un 1 au chiffre correspondant dans $T(\varphi)$. On pose la valuation $v(x_i) = \top$ si $a_i \in V$, et $\perp$ sinon. Ainsi, pour tout i , $a_i \in V$ si et seulement si x_i est satisfait par v , et $a'_i \in V$ si et seulement si $\neg x_i$ est satisfait par v .

Soit C_j une clause : comme il y a le chiffre 4 en position $p + j$ et que $c_j + c'_j$ n'apporte que 3 à ce chiffre, c'est forcément qu'on a au moins un 1 provenant d'un des a_i ou a'_i présents dans V , et donc $[C_j]_v = \top$ en considérant le littéral correspondant qui est nécessairement satisfait.

Comme toutes les clauses sont satisfaites, $[\varphi]_v = \top$ et φ est satisfiable.

Question 12 En déduire que Fira Sans(*SubsetSum*) est NP-complet.

Corrigé

On vient de montrer que $\varphi \mapsto (X(\varphi), T(\varphi))$ (précédée de la suppression des clauses de la question 7) est une réduction polynomiale de Fira Sans(3SAT) vers Fira Sans(*SubsetSum*). Or Fira Sans(3SAT) est NP-complet, donc pour tout $p \in \text{NP}$, $p \leq_P \text{Fira Sans}(3\text{SAT}) \leq_P \text{Fira Sans}(\text{SubsetSum})$ et, par transitivité, Fira Sans(*SubsetSum*) est NP-difficile. Comme on a montré que Fira Sans(*SubsetSum*) était dans NP, c'est un problème NP-complet.

30.1.4 Résolution par programmation dynamique

On considère ici un ensemble $X = \{x_0, \dots, x_{n-1}\} \subset \mathbb{N}^*$ et une valeur cible $T \in \mathbb{N}$ et on cherche à calculer $f(i, t)$ pour $i \in \llbracket 0, n \rrbracket$ et $t \in \llbracket 0, T \rrbracket$ tel que

$$f(i, t) = \begin{cases} \top & \text{si } \exists I \subset \llbracket i, n-1 \rrbracket, \sum_{k \in I} x_k = t \\ \perp & \text{sinon} \end{cases}$$

Question 13 Quelle valeur de f faut-il calculer pour résoudre cette instance de Fira Sans(*SubsetSum*) ?

Corrigé

Il faut calculer $f(0, T)$.

Question 14 Montrer qu'il y a des valeurs calculables directement pour f et d'autres valeurs qui satisfont une unique équation de récurrence, donc dans ce contexte une équation de Bellman.

Corrigé

On connaît les valeurs quand $\llbracket i, n-1 \rrbracket = \emptyset$, i.e. $i = n$, car soit $t = 0$ et la somme vide convient, soit c'est impossible.

$$f(n, 0) = \top \quad \forall t \in \llbracket 1, T \rrbracket, f(n, t) = \perp$$

On peut remarquer que la somme vide convient pour tout i donc

$$\forall i \in \llbracket 0, n \rrbracket, f(i, 0) = \top$$

Si $i < n$ et $t > 0$, on a deux possibilités pour réaliser t :

- soit on prend x_i et on complète avec un sous-ensemble dont les indices sont dans $\llbracket i+1, n-1 \rrbracket$ et réalisant $t - x_i$. **Attention** ce cas n'est valide que si $t \geq x_i$.
- soit on ignore x_i et on réalise t avec $\llbracket i+1, n-1 \rrbracket$.

Ainsi :

$$f(i, t) = f(i+1, t) \vee \begin{cases} f(i+1, t - x_i) & \text{si } x_i \leq t \\ \perp & \text{sinon} \end{cases}$$

Question 15 Écrire une fonction récursive qui calcule f de manière naïve.

Corrigé

```

bool dyn_rec(int *val, int nval, int i, int t)
{
    if (t == 0)
        return true;
    if (i == nval)
        return false;
    bool b = dyn_rec(val, nval, i+1, t);
    if (val[i] <= t)
        b = b || dyn_rec(val, nval, i+1, t-val[i]);
    return b;
}

bool resout_dynrec(int *val, int nval, int tgt)
{
    return dyn_rec(val, nval, 0, tgt);
}

```

Chaque appel en fait au plus deux, donc cette fonction est en $O(2^n)$. Cette borne est atteinte, par exemple pour $x_k = 2(k+1)$ et $T = n(n+1) + 1$: la cible restante est toujours impaire, donc jamais nulle, et strictement plus grande que la somme des valeurs restantes, donc l'arbre des appels est complet.

Question 16 On suppose donnés un type table et des fonctions :

```

table *create_table(); // crée une table vide
// teste si (i,t) est dans la table
bool has_couple(table *d, int i, int t);
// renvoie la valeur associée à (i,t) dans la table d
bool table_value(table *d, int i, int t);
// ajoute une correspondance (i, t) -> b
void table_add(table *d, int i, int t, bool b);
// libère la table
void free_table(table *d);

```

En déduire une version récursive efficace de f .

Corrigé

```

bool dyn_memo(int *val, int nval, table *d, int i, int t)
{
    if (has_couple(d, i, t))
        return table_value(d, i, t);

    bool b;
    if (t == 0)
        b = true;
    else if (i == nval)
        b = false;
    else {
        b = dyn_memo(val, nval, d, i+1, t);
        if (val[i] <= t)
            b = b || dyn_memo(val, nval, d, i+1, t-val[i]);
    }

    table_add(d, i, t, b);
    return b;
}

bool resout_dynmemo(int *val, int nval, int tgt)
{
    table *d = create_table();
    bool b = dyn_memo(val, nval, d, 0, tgt);
    free_table(d);
    return b;
}

```

Chaque couple $(i, t) \in \llbracket 0, n \rrbracket \times \llbracket 0, T \rrbracket$ n'est calculé qu'une fois : si les opérations sur la table sont en temps constant, on obtient une complexité en $O(nT)$.

Question 17 Écrire une version tabulée de f .

Corrigé

On range $f(i, t)$ dans la case $\text{tab}[i \cdot (\text{tgt} + 1) + t]$ d'un tableau de $(n + 1)(T + 1)$ booléens, rempli par i décroissant.

```
bool resout_dyntab(int *val, int nval, int tgt)
{
    bool *tab = malloc(sizeof(bool) * (nval+1) * (tgt+1));

    for(int t = 0; t <= tgt; t++)
    {
        tab[nval * (tgt+1) + t] = t == 0;
    }

    for(int i = nval-1; i >= 0; i--)
    {
        tab[i * (tgt+1)] = true;
        for(int t = 1; t <= tgt; t++)
        {
            bool b = tab[(i+1)*(tgt+1)+t];
            if(t >= val[i]) {
                b = b || tab[(i+1)*(tgt+1)+t-val[i]];
            }
            tab[i*(tgt+1)+t] = b;
        }
    }

    bool b = tab[tgt];
    free(tab);
    return b;
}
```

Question 18 On a ainsi calculé f avec une complexité en $O(nT)$. A-t-on montré que $P = NP$?

Corrigé

Non : $O(nT)$ est polynomial en la **valeur** de T , mais pas en la taille de l'entrée. Dans une représentation des entiers autre qu'en unaire (binaire, décimale...), T n'occupe que $O(\log T)$ symboles : T est donc exponentiel en la taille de son écriture. On dit que cet algorithme est *pseudo-polynomial*. C'est précisément ce qui se passe pour la réduction précédente, où $T(\varphi)$ vaut environ $4 \cdot 10^{p+n-1}$.

PROBLÈME 31

Devoir – Logique et Grammaire

DANS CE CHAPITRE

31.1 Preuves en déduction naturelle usuelle

31.2 Grammaires

31.3 ★ Lemme d'Ogden

MPI – DS 5 – Informatique. Durée : 4h.

Tous documents interdits.

31.1 Preuves en déduction naturelle usuelleLes règles et notations sont celles du chapitre de [déduction naturelle](#).**31.1.1 Logique propositionnelle intuitionniste**On rappelle les règles de la déduction naturelle pour les connecteurs $\wedge, \neg, \rightarrow$ et pour la règle axiome.

$$\frac{}{\Gamma, A \vdash A} \text{ ax}$$

Figure 31.1 – Règle axiome.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e$$

Figure 31.2 – Règles d'introduction et d'élimination de $\rightarrow$.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_{e,g} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_{e,d}$$

Figure 31.3 – Règles d'introduction et d'élimination de $\wedge$.

$$\frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} \neg_i \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \perp} \neg_e \qquad \frac{\Gamma \vdash \perp}{\Gamma \vdash C} \perp_e$$

Figure 31.4 – Règles pour $\neg$ et élimination de $\perp$.**Question 1** Donnez les règles pour $\vee$.

Corrigé

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{i,g} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{i,d} \quad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

Figure 31.5 – Règles d'introduction et d'élimination de $\vee$.**Question 2** Prouvez en déduction naturelle :

1. $A \wedge B \rightarrow B \wedge A$
2. $(A \vee B \rightarrow C) \rightarrow A \rightarrow C$
3. $A \rightarrow \neg\neg A$

Corrigé

1.
$$\frac{\frac{\frac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash B} ax \quad \frac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A} ax}{A \wedge B \vdash B \wedge A} \wedge_{e,d} \quad \wedge_{e,g} \quad \rightarrow_i$$
2.
$$\frac{\frac{\frac{A, (A \vee B) \rightarrow C \vdash (A \vee B) \rightarrow C}{A, (A \vee B) \rightarrow C \vdash C} ax \quad \frac{A, (A \vee B) \rightarrow C \vdash A \wedge B}{A, (A \vee B) \rightarrow C \vdash A \vee B} ax}{A, (A \vee B) \rightarrow C \vdash C} \vee_{i,g} \quad \rightarrow_e \quad \rightarrow_i$$
3.
$$\frac{\frac{\frac{\neg A, A \vdash A}{\neg A, A \vdash \perp} ax \quad \frac{\neg A, A \vdash \neg A}{\neg A, A \vdash \neg A} ax}{A \vdash \neg\neg A} \neg_i \quad \rightarrow_i$$

31.1.2 Logique propositionnelle classique**Question 3** On rajoute la règle :

$$\frac{}{\Gamma \vdash A \vee \neg A} cl$$

Figure 31.6 – Règle du tiers exclu.

Démontrez :

1. $\neg\neg A \rightarrow A$
2. $((A \rightarrow B) \rightarrow A) \rightarrow A$

Corrigé

1.
$$\frac{\frac{\frac{\neg\neg A \vdash A \vee \neg A}{\neg\neg A \vdash A} cl \quad \frac{\neg\neg A \vdash A}{\neg\neg A \vdash A} ax}{\neg\neg A \vdash A} \rightarrow_i$$
2.
$$\frac{\frac{\frac{\frac{A \rightarrow B \rightarrow A \vdash A \vee \neg A}{A, (A \rightarrow B) \rightarrow A \vdash A} cl \quad \frac{A, (A \rightarrow B) \rightarrow A \vdash A}{A, (A \rightarrow B) \rightarrow A \vdash A} ax}{(A \rightarrow B) \rightarrow A \vdash A} \rightarrow_i \quad \frac{\frac{\frac{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \vdash A}{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \vdash \perp} ax \quad \frac{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \vdash \perp}{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \vdash B} \perp_e}{A, \neg A, (A \rightarrow B) \rightarrow A \vdash A \vdash B} \rightarrow_e \quad \vee_e$$

31.1.3 Logique du premier ordre

On rappelle les règles sur les quantificateurs :

$$\frac{\Gamma \vdash A \quad x \notin FV(\Gamma)}{\Gamma \vdash \forall x.A} \forall_i \quad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \setminus t]} \forall_e$$

Figure 31.7 – Règles pour $\forall$; la condition sur les variables libres est en violet.

$$\frac{\Gamma \vdash A[x \setminus t]}{\Gamma \vdash \exists x.A} \exists_i \quad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B \quad x \notin FV(\Gamma, B)}{\Gamma \vdash B} \exists_e$$

Figure 31.8 – Règles pour $\exists$; la condition sur les variables libres est en violet.

Question 4 Démontrez :

1. $P(a) \rightarrow Q(a) \rightarrow \exists x.P(x) \wedge Q(x)$
2. $(\forall x.P(x)) \rightarrow \exists x.P(x)$

Corrigé

1.
$$\frac{\frac{\frac{Q(a), P(a) \vdash P(a)}{Q(a), P(a) \vdash P(a)} ax \quad \frac{Q(a), P(a) \vdash Q(a)}{Q(a), P(a) \vdash Q(a)} ax}{Q(a), P(a) \vdash P(a) \wedge Q(a)} \wedge_i \quad \frac{Q(a), P(a) \vdash P(a) \wedge Q(a)}{Q(a), P(a) \vdash \exists x.P(x) \wedge Q(x)} \exists_i \quad \frac{Q(a), P(a) \vdash \exists x.P(x) \wedge Q(x)}{P(a) \vdash Q(a) \rightarrow (\exists x.P(x) \wedge Q(x))} \rightarrow_i \quad \frac{P(a) \vdash Q(a) \rightarrow (\exists x.P(x) \wedge Q(x))}{\vdash P(a) \rightarrow (Q(a) \rightarrow (\exists x.P(x) \wedge Q(x)))} \rightarrow_i$$
2.
$$\frac{\frac{\frac{\forall x.P(x) \vdash \forall x.P(x)}{\forall x.P(x) \vdash \forall x.P(x)} ax \quad \frac{\forall x.P(x) \vdash P(y)}{\forall x.P(x) \vdash P(y)} \forall_e}{\forall x.P(x) \vdash \exists x.P(x)} \exists_i \quad \frac{\forall x.P(x) \vdash \exists x.P(x)}{\vdash (\forall x.P(x)) \rightarrow (\exists x.P(x))} \rightarrow_i$$

On a utilisé $\forall_e$ avec le terme y , puis $\exists_i$ avec ce même terme.

31.2 Grammaires

On emploiera ici le mot grammaire par simplification pour désigner des grammaires non contextuelles. Les définitions et notations sont celles du chapitre sur les [grammaires](#).

31.2.1 Étude d'une chaîne dans une grammaire

On considère la grammaire donnée par $G = (\{+, *, a\}, \{S\}, P, S)$ où P est l'ensemble des productions :

$$S \rightarrow S S + \mid S S * \mid a$$

et un mot $m = aa + a *$.

Question 5 Donnez une dérivation à gauche de m .

Corrigé

$$S \Rightarrow SS * \Rightarrow SS + S * \Rightarrow aS + S * \Rightarrow aa + S * \Rightarrow aa + a *$$

Question 6 Donnez une dérivation à droite de m .

Corrigé

$$S \Rightarrow SS * \Rightarrow Sa * \Rightarrow SS + a * \Rightarrow Sa + a * \Rightarrow aa + a *$$

Question 7 Donnez un arbre d'analyse de m .

Corrigé

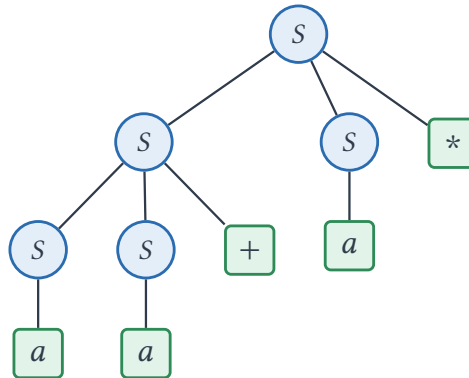


Figure 31.9 — Arbre d'analyse de $m = a a + a *$ (feuilles : terminaux, en vert).

Question 8 Cette grammaire est-elle ambiguë ? Justifiez votre réponse.

Corrigé

Cette grammaire n'est pas ambiguë car chaque production va faire apparaître un terminal à droite qui fixe la règle à appliquer.

✚ Pour que l'argument soit complet, il faut aussi justifier que le découpage d'un mot $w = w_1 w_2 +$ (ou $w_1 w_2 *$) avec $w_1, w_2 \in \text{lang}(G)$ est unique. On associe à chaque mot son *poids* : le nombre de a moins le nombre d'opérateurs. Par récurrence sur la dérivation, tout mot w de $\text{lang}(G)$ est de poids 1 et tous ses préfixes non vides sont de poids ≥ 1 : c'est vrai pour a et, pour $w_1 w_2 +$, un préfixe non vide de w_1 est de poids ≥ 1 , un préfixe $w_1 p$ avec p préfixe non vide de w_2 est de poids $1 + (\geq 1) \geq 2$ et $w_1 w_2 +$ est de poids $1 + 1 - 1 = 1$.

Ainsi, w_1 est le plus long préfixe de poids 1 de $w_1 w_2$: il est déterminé par w . On montre alors par récurrence forte sur la longueur que tout mot de $\text{lang}(G)$ a un unique arbre d'analyse : la dernière lettre fixe la règle appliquée à la racine, le découpage $w_1 w_2$ est unique et, par hypothèse de récurrence, w_1 et w_2 ont chacun un unique arbre d'analyse.

Question 9 Décrivez le langage reconnu par cette grammaire.

Corrigé

Ce sont les expressions postfixes sur a avec deux opérations binaires : $*$ et $+$.

31.2.2 Écriture de grammaires

Question 10 Déterminez une grammaire reconnaissant l'ensemble des palindromes sur l'alphabet $\{0, 1\}$.

Rappel : un palindrome est un mot qui se lit de la même manière de gauche à droite et de droite à gauche.

Corrigé

On considère la grammaire G définie par

$$S \rightarrow 0 S 0 \mid 1 S 1 \mid 0 \mid 1 \mid \epsilon$$

Les règles $S \rightarrow 0$ et $S \rightarrow 1$ sont indispensables pour obtenir les palindromes de longueur impaire, comme 0 ou 101.

Il est assez immédiat que $\text{lang}(G)$ est l'ensemble des palindromes, par récurrence sur la longueur d'un mot : un mot de longueur au plus 1 est un palindrome et est engendré directement ; sinon, on considère

la première lettre qui est forcément égale à la dernière, soit par application d'une production, soit par définition, et ainsi on peut faire la récurrence sur le mot intérieur.

Question 11 Déterminez une grammaire reconnaissant l'ensemble des mots sur $\{0, 1\}$ qui ne contiennent pas 011.

Corrigé

Ici, c'est un langage régulier, on peut donc raisonner un peu à la manière des automates (voir la construction d'une grammaire linéaire à droite à partir d'un automate dans le chapitre sur les grammaires) :

$$S \rightarrow 0S' \mid 1S \mid \epsilon$$

$$S' \rightarrow 1S'' \mid 0S' \mid \epsilon$$

$$S'' \rightarrow 0S' \mid \epsilon$$

Le non terminal S correspond à un préfixe lu ne se terminant pas par un début de 011, S' à un préfixe se terminant par 0 et S'' à un préfixe se terminant par 01 : depuis S'' , lire un 1 est interdit.

On montre que le langage est bien celui demandé par récurrence sur la longueur d'un mot, en raisonnant sur le préfixe.

31.2.3 Écriture d'un analyseur syntaxique

On considère la grammaire G donnée par :

- les terminaux **if, then, else, begin, end, print, ,, +, -, *, /, (,), id, num**
- les non terminaux *stmt, stmtList, expr, term, factor*
- le symbole initial *stmt*
- et les productions :

$$\begin{aligned} \text{stmt} &\rightarrow \text{if } \text{expr} \text{ then } \text{stmt} \text{ else } \text{stmt} \\ &\quad | \text{if } \text{expr} \text{ then } \text{stmt} \\ &\quad | \text{begin } \text{stmtList} \text{ end} \\ &\quad | \text{print } \text{expr} \\ \text{stmtList} &\rightarrow \text{stmt} ; \text{stmtList} \mid \text{stmt} \\ \text{expr} &\rightarrow \text{term} + \text{expr} \mid \text{term} - \text{expr} \mid \text{term} \\ \text{term} &\rightarrow \text{factor} * \text{term} \mid \text{factor} / \text{term} \mid \text{factor} \\ \text{factor} &\rightarrow \text{id} \mid \text{num} \mid (\text{expr}) \end{aligned}$$

Les terminaux sont ainsi donnés, dans l'ordre précédent, par un type token :

```
type token =
  | Tif | Tthen | Telse | Tbegin | Tend | Tprint
  | Tcolon | Tplus | Tminus | Ttimes | Tdiv
  | Tlpar | Trpar | Tid of string | Tnum of int
```

On définit un type pour représenter l'arbre syntaxique :

```
type expr =
  Variable of string
  | Constant of int
  | Add of expr * expr
  | Subtract of expr * expr
  | Mult of expr * expr
  | Divide of expr * expr
```

```
type stmt =
| IfThenElse of expr * stmt * stmt option
| Block of stmt list
| Print of expr
```

On vous donne également une fonction :

```
val parse_expr : token list -> expr * token list
```

où `parse_expr tl` prend en entrée une liste de tokens et renvoie un couple (e, tl') où e est l'expression lue et tl' sont les tokens restants.

Question 12 Pour cette question, on pourra, avec une note qui en dépendra, considérer :

- soit la grammaire complète précédente ;
- soit la grammaire simplifiée sans le `else` ;
- soit la grammaire simplifiée sans les deux productions du `if`.

Écrivez une fonction

```
val parse_stmt : token list -> stmt * token list
```

où `parse_stmt tl` prend en entrée une liste de tokens et renvoie un couple (st, tl') où st est un `stmt` lu et tl' sont les tokens restants.

Corrigé

On lève une exception `ParsingError` en cas d'erreur de syntaxe.

```
exception ParsingError

let rec parse_stmt tl =
  match tl with
  | Tif :: q -> begin
    let cond, tl1 = parse_expr q in
    match tl1 with
    | Tthen :: q1 -> begin
      let sthen, tl2 = parse_stmt q1 in
      match tl2 with
      | Telse :: q2 ->
        let selse, tl3 = parse_stmt q2 in
        IfThenElse(cond, sthen, Some selse), tl3
      | _ -> IfThenElse(cond, sthen, None), tl2
    end
    | _ -> raise ParsingError
  end
  | Tbegin :: q -> begin
    let s_list, tl1 = parse_stmt_list q in
    match tl1 with
    | Tend :: q1 -> Block s_list, q1
    | _ -> raise ParsingError
  end
  | Tprint :: q ->
    let e, tl1 = parse_expr q in
    Print e, tl1
  | _ -> raise ParsingError
and parse_stmt_list tl =
  let s, tl1 = parse_stmt tl in
  match tl1 with
  | Tcolon :: q ->
    let sl, tl2 = parse_stmt_list q in
    s :: sl, tl2
  | _ -> [s], tl1
```



La grammaire est ambiguë : `if  $e_1$  then if  $e_2$  then  $s_1$  else  $s_2$` a deux arbres d'analyse (problème du « else pendant »). Cet analyseur, qui lit un `else` dès qu'il le peut, rattache le `else` au `if` le plus proche, ce qui est la convention usuelle.

31.2.4 Extension de la syntaxe

On se propose de rajouter deux extensions pour écrire les productions :

- $A \rightarrow \alpha[\beta]\gamma$ signifiant que β est *optionnel*. Cela équivaut à $A \rightarrow \alpha\gamma \mid \alpha\beta\gamma$;
- $A \rightarrow \alpha\{\beta\}\gamma$ signifiant que β peut être répété un nombre quelconque de fois, y compris 0 fois. Cela équivaut à $A \rightarrow \alpha\gamma \mid \alpha\beta\gamma \mid \alpha\beta\beta\gamma \mid \dots$

Question 13 Donnez une forme simplifiée de la grammaire précédente en utilisant cette extension.

On ne changera pas, et on ne répètera pas, les productions pour les expressions.

Corrigé

```
stmt → if expr then stmt [else stmt]
      | begin stmt {; stmt} end
      | print expr
```

Question 14 Démontrez que toute grammaire utilisant ces extensions peut se mettre sous une forme faiblement équivalente qui ne les utilise pas.

Corrigé

La réponse est dans la question pour la première extension.

Pour la deuxième extension, il suffit de remplacer

$$A \rightarrow \alpha\{\beta\}\gamma$$

par

$$A \rightarrow \alpha A' \gamma \quad A' \rightarrow \epsilon \mid \beta A'$$



où A' est un **nouveau** non terminal, propre à cette occurrence. On a $A' \xRightarrow{*} \beta^k A'$ pour tout k , donc A' se dérive en β^k pour tout $k \in \mathbb{N}$, et ce sont les seules possibilités : cette transformation ne change pas le langage engendré.

Si les extensions sont imbriquées ou multiples, on les élimine une par une. Pour que le nombre d'extensions diminue strictement à chaque étape, il vaut mieux traiter aussi la première extension avec un nouveau non terminal, $A \rightarrow \alpha A' \gamma$ et $A' \rightarrow \beta \mid \epsilon$: la réécriture $A \rightarrow \alpha\gamma \mid \alpha\beta\gamma$ dupliquerait les extensions éventuellement présentes dans α et γ . Chaque étape supprime ainsi exactement une paire de crochets ou d'accolades, et on obtient au bout d'un nombre fini d'étapes une grammaire sans extension faiblement équivalente.

31.3 ★ Lemme d'Ogden

On va présenter ici un lemme qui sera l'équivalent du lemme de l'étoile pour les grammaires non contextuelles.

31.3.1 Lemme combinatoire sur les arbres

On considère ici un arbre d'arité quelconque dont certaines feuilles sont *distinguées*.

On dit alors que :

- un nœud est *distingué* lorsque le sous-arbre dont il est racine contient des feuilles distinguées ;
- un nœud est *spécial* lorsqu'il a au moins deux fils distingués.

Une *branche* est un chemin de la racine à une feuille.

Remarque 31.1 Cette notion de *distingué* est juste une manière d'introduire une sorte de coloration qui va nous servir pour rajouter de la structure dans un arbre de dérivation dans la suite. C'est une forme de ressource, comme lorsqu'on comptait des crédits pour la complexité amortie.

Question 15 On dit qu'un nœud est le plus petit ancêtre commun de deux nœuds x et y s'il est le premier nœud où les branches de x et y vers la racine se rejoignent.

Que peut-on dire du plus petit ancêtre commun de deux nœuds distingués ?

Corrigé

Soit z le plus petit ancêtre commun de deux nœuds distingués x et y . Il est distingué, car son sous-arbre contient celui de x .

Si de plus aucun des deux n'est ancêtre de l'autre (par exemple si ce sont deux feuilles distinctes), alors $z \neq x$ et $z \neq y$: les chemins de z vers x et vers y passent par deux fils **distincts** de z (sinon ce fils serait un ancêtre commun plus bas). Ces deux fils sont distingués car leurs sous-arbres contiennent respectivement x et y : z est donc **spécial**.

Si x est un ancêtre de y , le plus petit ancêtre commun est x , qui n'a aucune raison d'être spécial.

On considère le lemme :

Lemme 31.2

Soit t un arbre dont tous les nœuds sont d'arité au plus $m \geq 1$ et ayant k feuilles distinguées.

Si chaque branche de t contient au plus r nœuds spéciaux, alors $k \leq m^r$.

Question 16 En raisonnant par récurrence sur r , démontrez ce lemme.

Corrigé

On remarque d'abord que le caractère distingué ou spécial d'un nœud ne dépend que du sous-arbre dont il est racine : on peut donc l'examiner indifféremment dans t ou dans un sous-arbre de t .

On montre par récurrence sur r la propriété « pour tout arbre t d'arité au plus m dont chaque branche contient au plus r nœuds spéciaux, t a au plus m^r feuilles distinguées ».

- $r = 0$: si t avait deux feuilles distinguées distinctes, leur plus petit ancêtre commun serait spécial d'après la question précédente, et la branche menant à l'une de ces feuilles contiendrait un nœud spécial. Donc $k \leq 1 = m^0$.
- Supposons la propriété vraie pour r et soit t dont chaque branche contient au plus $r + 1$ nœuds spéciaux. Si $k \leq 1$, on a bien $k \leq m^{r+1}$. Sinon, t contient un nœud spécial (le plus petit ancêtre commun de deux feuilles distinguées) ; soit s un nœud spécial de profondeur minimale.

Toutes les feuilles distinguées sont dans le sous-arbre de s : si une feuille distinguée f n'y était pas, en prenant une feuille distinguée g sous s , le plus petit ancêtre commun de f et g serait un ancêtre strict de s et serait spécial, ce qui contredit la minimalité de la profondeur de s .

Soient $c_1, \dots, c_q$ les fils de s , avec $q \leq m$. Toute branche du sous-arbre de racine c_i se prolonge en une branche de t passant par s , qui est spécial et n'est pas dans ce sous-arbre : elle contient donc au plus r nœuds spéciaux. Par hypothèse de récurrence, le sous-arbre de racine c_i a au plus m^r feuilles distinguées, et donc

$$k \leq q \cdot m^r \leq m^{r+1}$$

31.3.2 Lemme d'Ogden

On considère maintenant le lemme suivant dû à Ogden :

Lemme 31.3

Soit $G = (\Sigma, V, P, S)$ une grammaire non contextuelle et $X \in V$.

Il existe $K \in \mathbb{N}$ tel que tout mot $f \in (\Sigma \cup V)^*$ vérifiant $X \xRightarrow{*} f$ et ayant au moins K lettres, donc des éléments de $\Sigma \cup V$, distinguées se factorise en $f = \alpha u \beta v \gamma$ où $\alpha, u, \beta, v, \gamma \in (\Sigma \cup V)^*$, avec :

1. $X \xRightarrow{*} \alpha A \gamma$, $A \xRightarrow{*} u A v$, $A \xRightarrow{*} \beta$ où $A \in V$;
2. soit α, u et β , soit β, v et γ contiennent des lettres distinguées ;
3. $u \beta v$ contient moins de K lettres distinguées.

Question 17 Montrer que des conditions du lemme, on peut en déduire que pour tout $n \in \mathbb{N}$, on a $X \xRightarrow{*} \alpha u^n \beta v^n \gamma$.

Note : c'est une forme d'étoile symétrique.

Corrigé

On montre par récurrence sur n que $A \xRightarrow{*} u^n A v^n$. Pour $n = 0$, c'est la dérivation de longueur nulle. Si $A \xRightarrow{*} u^n A v^n$, comme les règles s'appliquent indépendamment du contexte, on peut dériver le A central en $u A v$: $A \xRightarrow{*} u^n A v^n \xRightarrow{*} u^n u A v v^n = u^{n+1} A v^{n+1}$.

Ainsi, pour tout n :

$$X \xRightarrow{*} \alpha A \gamma \xRightarrow{*} \alpha u^n A v^n \gamma \xRightarrow{*} \alpha u^n \beta v^n \gamma$$

Question 18 Démontrez le lemme d'Ogden en appliquant le lemme précédent à l'arbre correspondant à la dérivation.

Corrigé

Arbre d'une dérivation. Comme dans le cours, à une dérivation $X \xRightarrow{*} f$ on associe un arbre de racine X dont les nœuds internes sont étiquetés par des non terminaux et respectent les règles de production, mais dont les feuilles peuvent maintenant être étiquetées par des terminaux, par ϵ ou par des **non terminaux** (ceux qui n'ont pas été réécrits). Sa frontière est f , et réciproquement tout tel arbre de frontière f donne une dérivation $X \xRightarrow{*} f$ (même preuve que dans le cours). Les feuilles distinguées sont celles qui portent les lettres distinguées de f .

Choix de K . Soit $h = |V|$ le nombre de non terminaux et $m \geq 2$ un majorant de la longueur des membres droits des règles, qui majore donc l'arité des nœuds. On pose $K = m^{2h+3} + 1$.

Soit f avec $X \xRightarrow{*} f$ ayant au moins K lettres distinguées, et t un arbre de cette dérivation. Comme t a plus de m^{2h+2} feuilles distinguées, d'après le lemme combinatoire, une branche de t contient au moins $2h + 3$ nœuds spéciaux.

Choix d'un sous-arbre. Soit x un nœud de profondeur maximale tel que le sous-arbre de racine x contienne une branche (de x à une feuille) avec au moins $2h + 3$ nœuds spéciaux. Aucun fils de x n'a cette propriété, donc chaque branche du sous-arbre de racine x contient au plus $1 + (2h + 2) = 2h + 3$ nœuds spéciaux. D'après le lemme combinatoire, ce sous-arbre a au plus $m^{2h+3} < K$ feuilles distinguées.

Choix de A . Fixons une branche b issue de x contenant au moins $2h + 3$ nœuds spéciaux. Un nœud spécial s de b a au moins deux fils distingués, dont au plus un est sur b : il a donc un fils distingué à gauche ou à droite du fils par lequel passe b . On dit que s est *gauche* ou *droit* en conséquence (au choix s'il est les deux). Parmi $2h + 3$ nœuds spéciaux, au moins $h + 2$ sont du même côté, disons gauche (l'autre cas est symétrique). Notons-les $t_0, t_1, \dots, t_{h+1}$ de haut en bas. Parmi les $h + 1$ nœuds $t_1, \dots, t_{h+1}$, deux portent la même étiquette $A \in V$ d'après le principe des tiroirs : t_a au-dessus de t_c , avec $1 \leq a < c$.

Factorisation. On note β la frontière du sous-arbre de racine t_c , $u\beta v$ celle du sous-arbre de racine t_a , et $f = \alpha u\beta v\gamma$.

1. En coupant l'arbre sous t_a , on obtient un arbre de racine X de frontière $\alpha A\gamma$, donc $X \xrightarrow{*} \alpha A\gamma$. De même, le sous-arbre de racine t_a coupé sous t_c donne $A \xrightarrow{*} uAv$, et le sous-arbre de racine t_c donne $A \xrightarrow{*} \beta$.
2. β contient une lettre distinguée car t_c , spécial, est distingué. Le fils distingué gauche de t_a est à gauche de la branche b , qui mène à t_c : ses feuilles sont dans u , qui contient donc une lettre distinguée. De même, le fils distingué gauche de t_0 est à gauche de t_a : α contient une lettre distinguée.
3. $u\beta v$ est la frontière du sous-arbre de racine t_a , inclus dans celui de racine x : il contient au plus $m^{2h+3} < K$ lettres distinguées.

Dans le cas symétrique, on obtient de même que β , v et γ contiennent des lettres distinguées.

Le corollaire suivant suffit en pratique pour montrer qu'un langage n'est pas non contextuel :

Question 19 En déduire que si L est un langage non contextuel, il existe $N \geq 0$ tel que pour tout mot $f \in L$, si $|f| \geq N$ alors on peut trouver une factorisation $f = \alpha u\beta v\gamma$ telle que $|uv| > 0$, $|u\beta v| < N$ et $\alpha u^n \beta v^n \gamma \in L$ pour tout $n \geq 0$.

Corrigé

Soit $G = (\Sigma, V, P, S)$ une grammaire engendrant L et K la constante donnée par le lemme d'Ogden pour $X = S$. On pose $N = K$.

Soit $f \in L$ avec $|f| \geq N$. On distingue **toutes** les lettres de f : il y en a au moins K , et $S \xrightarrow{*} f$. On obtient une factorisation $f = \alpha u\beta v\gamma$ vérifiant les trois conditions du lemme.

- D'après la condition 2, u ou v contient une lettre distinguée, donc $|uv| > 0$.
- Toutes les lettres étant distinguées, la condition 3 donne $|u\beta v| < K = N$.
- D'après la question 17, $S \xrightarrow{*} \alpha u^n \beta v^n \gamma$ pour tout $n \geq 0$. Comme $f \in \Sigma^*$, les mots $\alpha, u, \beta, v, \gamma$ ne contiennent que des terminaux, donc $\alpha u^n \beta v^n \gamma \in \text{lang}(G) = L$.

31.3.3 Application

Question 20 Démontrez que le langage $L = \{a^n b^n c^n \mid n \geq 0\}$ n'est pas reconnaissable par une grammaire non contextuelle.

Corrigé

Supposons L non contextuel et soit N donné par la question précédente. On considère $f = a^N b^N c^N \in L$, de longueur $3N \geq N$, et une factorisation $f = \alpha u\beta v\gamma$ comme dans la question précédente.

Le facteur $u\beta v$ est de longueur strictement inférieure à N : il ne peut pas contenir à la fois un a et un c , sinon il contiendrait les N lettres b situées entre les deux. Supposons par exemple qu'il ne contienne pas de c (l'autre cas est symétrique).

Pour $n = 0$, on a $\alpha\beta\gamma \in L$. Or ce mot est obtenu en retirant de f les lettres de u et v , qui ne sont pas des c : il contient donc encore N lettres c , et il est de longueur $3N - |uv| < 3N$. Un mot de L contenant N lettres c est $a^N b^N c^N$, de longueur $3N$: contradiction.

Ainsi L n'est pas non contextuel.

Exercice – Tableaux autoréférents

32.1 Tableaux autoréférents

✍ Exercice d'après le sujet 0 du concours CCINP (filiale MPI).

Définition 32.1

Soit $n \in \mathbb{N}^*$. Un tableau t de taille n à valeurs dans $\llbracket 0, n-1 \rrbracket$ est dit **autoréférent** lorsque, pour tout $k \in \llbracket 0, n-1 \rrbracket$, t_k est le nombre d'occurrences de k dans t :

$$\forall k \in \llbracket 0, n-1 \rrbracket, \quad t_k = |\{i \in \llbracket 0, n-1 \rrbracket \mid t_i = k\}|$$

Exemple 32.2

Le tableau $[1; 2; 1; 0]$ est autoréférent : il contient une fois la valeur 0, deux fois la valeur 1, une fois la valeur 2 et aucune fois la valeur 3.

Les programmes sont à écrire en OCaml. On utilisera la fonction suivante qui renvoie la somme $t_0 + \dots + t_{i-1}$ des i premières cases d'un tableau.

```
let somme t i =
  let s = ref 0 in
  for k = 0 to i-1 do
    s := !s + t.(k)
  done;
  !s
```

32.1.1 Vérification

Question 1 Montrer qu'il n'existe pas de tableau autoréférent de taille 1, 2 ou 3, et trouver un second tableau autoréférent de taille 4.

Corrigé

- On a toujours $t_0 \geq 1$: si $t_0 = 0$, la case 0 contient un 0, donc 0 apparaît au moins une fois, d'où $t_0 \geq 1$, contradiction. De plus, si $t_0 = a$ alors la valeur $a \geq 1$ apparaît, donc $t_a \geq 1$.
- $n = 1$: le seul tableau est $[0]$, exclu par le point précédent.
- $n = 2$: $t_0 = 1$, il y a donc un unique 0, nécessairement en case 1. Mais la valeur 1 apparaît (en case 0), donc $t_1 \geq 1$: contradiction.
- $n = 3$: si $t_0 = 2$, les deux 0 sont en cases 1 et 2, donc $t_2 = 0$ alors que la valeur 2 apparaît. Si $t_0 = 1$, la valeur 1 apparaît donc $t_1 \geq 1$, et l'unique 0 est en case 2 : $t = [1; 1; 0]$, qui contient deux 1 alors que $t_1 = 1$, ou $t = [1; 2; 0]$, qui contient un 2 alors que $t_2 = 0$.
- $n = 4$: $[1; 2; 0; 2]$ est aussi autoréférent. On verra à la question 8 qu'il n'y en a pas d'autre.

Question 2 Écrire une fonction `est_auto : int array -> bool` qui teste si un tableau à valeurs dans $\llbracket 0, n - 1 \rrbracket$ est autoréférent, en temps $O(n)$.

Corrigé

On compte les occurrences dans un second tableau puis on compare.

```
let est_auto t =
  let n = Array.length t in
  let t2 = Array.make n 0 in
  for i = 0 to n-1 do
    t2.(t.(i)) <- 1 + t2.(t.(i))
  done;
  t = t2
```

La boucle et la comparaison structurelle de deux tableaux de taille n sont en $O(n)$.

Question 3 Montrer que si t est autoréférent de taille n , alors $\sum_{k=0}^{n-1} t_k = n$.

Corrigé

Chaque case i contient exactement une valeur $t_i \in \llbracket 0, n - 1 \rrbracket$, elle est donc comptée dans exactement un des t_k , celui pour $k = t_i$. Ainsi $\sum_k t_k = \sum_k |\{i \mid t_i = k\}| = n$.

32.1.2 Génération par retour sur trace

Question 4 Écrire une fonction `gen_auto : int -> int array list` qui renvoie la liste des tableaux autoréférents de taille n , en essayant toutes les valeurs possibles pour chaque case, de la gauche vers la droite. Quelle est sa complexité ?

Corrigé

```
let gen_auto n =
  let solutions = ref [] in
  let t = Array.make n 0 in
  let rec aux i =
    if i = n
    then (if est_auto t
          then solutions := Array.copy t :: !solutions)
    else
      for k = 0 to n-1 do
        t.(i) <- k;
        aux (i+1)
      done
  in aux 0;
  !solutions
```

L'arbre des appels a n^n feuilles, chacune effectuant un test en $O(n)$, et $\sum_{i=0}^{n-1} n^i \leq n^n$ nœuds internes : la complexité est en $O(n^{n+1})$.

Question 5 En utilisant la question 3, proposer un premier élagage : une condition sur $t_0, \dots, t_{i-1}$ qui permet d'abandonner l'exploration. Écrire la fonction `gen_auto_opt1` correspondante.

Corrigé

Comme les cases sont positives, si $t_0 + \dots + t_{i-1} > n$ aucune complétion ne peut vérifier la question 3 : on peut abandonner.

```
let gen_auto_opt1 n =
  let solutions = ref [] in
  let t = Array.make n 0 in
```

```

let rec aux i =
  if i = n
  then (if est_auto t
        then solutions := Array.copy t :: !solutions)
  else
    let elague =
      somme t i > n
    in
      if not elague
      then begin
        for k = 0 to n-1 do
          t.(i) <- k;
          aux (i+1)
        done
      end
in aux 0;
!solutions

```

Question 6 Lorsqu'on veut placer la valeur k en case i avec $k < i$, la valeur t_k est déjà fixée. En déduire un second élagage, en maintenant un tableau `occ` tel que `occ.(k)` soit le nombre d'occurrences de k parmi $t_0, \dots, t_{i-1}$. Écrire `gen_auto_opt2`.

Corrigé

Dans un tableau autoréférent, la valeur k apparaît exactement t_k fois. Si $k < i$, on ne peut donc placer une nouvelle occurrence de k que si `occ.(k) < t.(k)`. Si $k \geq i$, t_k n'est pas encore choisi et il n'y a pas de contrainte. On met à jour `occ` avant l'appel récursif et on le restaure après.

```

let gen_auto_opt2 n =
  let solutions = ref [] in
  let t = Array.make n 0 in
  let occ = Array.make n 0 in
  let rec aux i =
    if i = n
    then (if est_auto t
          then solutions := Array.copy t :: !solutions)
    else
      let elague =
        somme t i > n
      in
        if not elague
        then begin
          for k = 0 to n-1 do
            if k >= i || occ.(k) < t.(k)
            then begin
              t.(i) <- k;
              occ.(k) <- occ.(k) + 1;
              aux (i+1);
              occ.(k) <- occ.(k) - 1
            end
          done
        end
  in aux 0;
  !solutions

```

Question 7 Soit t autoréférent et $i \in \llbracket 1, n \rrbracket$. On note $s = t_0 + \dots + t_{i-1}$ et o le nombre de 0 parmi $t_0, \dots, t_{i-1}$. Montrer que $s \leq i + t_0 - o$. En déduire `gen_auto_opt3`.

Corrigé

D'après la question 3, $n = \sum_{j=0}^{n-1} t_j = s + \sum_{j=i}^{n-1} t_j$. Parmi les $n - i$ cases $t_i, \dots, t_{n-1}$, exactement $t_0 - o$ valent 0 (il y a t_0 zéros en tout), les $(n - i) - (t_0 - o)$ autres valent au moins 1. Ainsi $\sum_{j \geq i} t_j \geq n - i - t_0 + o$, d'où $s \leq n - (n - i - t_0 + o) = i + t_0 - o$.

On peut donc abandonner dès que $s > i + t_0 - o$, ce qui raffine la condition $s > n$ (pour $i = 0$, $s = 0$ et la condition est fausse : $t.(0)$ vaut encore sa valeur initiale 0, sans conséquence).

```
let gen_auto_opt3 n =
  let solutions = ref [] in
  let t = Array.make n 0 in
  let occ = Array.make n 0 in
  let rec aux i =
    if i = n
    then (if est_auto t
          then solutions := Array.copy t :: !solutions)
    else
      let elague =
        let s = somme t i in
        s > n || s > i + t.(0) - occ.(0)
      in
      if not elague
      then begin
        for k = 0 to n - 1 do
          if k >= i || occ.(k) < t.(k)
          then begin
            t.(i) <- k;
            occ.(k) <- occ.(k) + 1;
            aux (i + 1);
            occ.(k) <- occ.(k) - 1
          end
        done
      end
  in aux 0;
  !solutions
```

Remarque 32.3 somme $t\ i$ coûte $O(i)$ à chaque appel. On pourrait passer la somme partielle en argument de aux pour la maintenir en temps constant.

Question 8 Donner les tableaux autoréférents pour $n \leq 11$. Que conjecturer ?

Corrigé

Les quatre fonctions renvoient les mêmes solutions pour $n \leq 7$ (et les deux dernières jusqu'à $n = 11$) :

n	tableaux autoréférents
1, 2, 3, 6	aucun
4	[1;2;1;0], [2;0;2;0]
5	[2;1;2;0;0]
7	[3;2;1;1;0;0;0]
8	[4;2;1;0;1;0;0;0]
9	[5;2;1;0;0;1;0;0;0]
10	[6;2;1;0;0;0;1;0;0;0]
11	[7;2;1;0;0;0;0;1;0;0;0]

On conjecture que pour $n \geq 7$ l'unique tableau autoréférent est $t_0 = n - 4$, $t_1 = 2$, $t_2 = 1$, $t_{n-4} = 1$ et $t_k = 0$ sinon (on vérifie facilement qu'il est autoréférent ; l'unicité est admise).

PARTIE VIII

Corrigés de concours

33	Problème DS06 - Sujet CCINP	343
34	Problème Mines - Sujet 0 - MPI	347
35	Problème Mines 2023 - Épreuve 2	357
36	Problème Centrale 2022 - Option informatique	367
37	Problème Agrégation 2023 - Épreuve 3b	379

PROBLÈME 33

DS06 – Sujet CCINP

DANS CE CHAPITRE

33.1 Logique

33.2 Dominos

Sujet Corrigé d'un sujet du concours CCINP donné en DS 6 de MPI. L'énoncé n'est pas reproduit ici : seules les réponses sont données, dans l'ordre des questions du sujet.

33.1 Logique

Oh chouette, du foot...

Question 1 ::: {adm kind=« corrige » title=« Corrigé »}

Quatre variables :

- F : faire la fête
- G : mon équipe gagne
- M : je marque
- C : je suis content, et quand je suis content je... :::

Question 2 ::: {adm kind=« corrige » title=« Corrigé »}

- $P_1 = M \rightarrow C \wedge F$
- $P_2 = G \rightarrow C \vee F$
- $P_3 = \neg G \rightarrow \neg C \vee M$
- $P_4 = \neg M \wedge F \rightarrow C$
- $P_5 = \neg C$:::

Question 3 ::: {adm kind=« corrige » title=« Corrigé »}

- $P_1 \equiv \neg M \vee (C \wedge F) \equiv (\neg M \vee C) \wedge (\neg M \vee F)$
- $P_2 \equiv \neg G \vee C \vee F$ (une seule clause)
- $P_3 \equiv G \vee \neg C \vee M$
- $P_4 \equiv \neg(\neg M \wedge F) \vee C \equiv M \vee \neg F \vee C$
- $P_5 = \neg C$:::

Question 4 ::: {adm kind=« corrige » title=« Corrigé »}

$$\begin{aligned} F &= P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5 \\ &\equiv (\neg M \vee C) \wedge (\neg M \vee F) \wedge (\neg G \vee C \vee F) \wedge (G \vee \neg C \vee M) \wedge (M \vee \neg F \vee C) \wedge \neg C \end{aligned}$$

qui est une formule en forme normale conjonctive dont chaque clause a au plus trois littéraux : c'est une instance de 3-SAT. :::

Question 5 ::: {adm kind=« corrige » title=« Corrigé »}

- On commence par remplacer $C \leftarrow \perp$ dans F et on appelle ainsi Quine sur $\neg M \wedge (\neg M \vee F) \wedge (\neg G \vee F) \wedge (M \vee \neg F)$.
- On effectue alors le remplacement $M \leftarrow \perp$ et on appelle ainsi Quine sur $(\neg G \vee F) \wedge \neg F$.
 - On remplace $F \leftarrow \perp$ et on appelle Quine sur $\neg G$.
 - On remplace $G \leftarrow \perp$ et on obtient l'ensemble vide de clauses, donc l'appel renvoie vrai.
 - On renvoie vrai.
 - On renvoie vrai.
- On renvoie vrai.

La formule est donc satisfiable. On note que la valuation qui la satisfait est : $F = C = M = G = \perp$ (c'est d'ailleurs la seule, comme on le vérifie sur la table de vérité). Il ne fait pas la fête, il n'est pas content, il n'a pas marqué et il a perdu. Youpi. :::

33.2 Dominos

Remarque 33.1 Attention : le typedef sur une structure anonyme est hors programme. Aussi, Domino mais element, c'est horrible.

Question 1 ::: {adm kind=« corrige » title=« Corrigé »}

```
struct element {
    Domino *valeur;
    struct element *suivant;
};
typedef struct element element;
```

:::

Question 2 ::: {adm kind=« corrige » title=« Corrigé »}

Je ne comprends pas bien la question, vu qu'on ne demande rien de plus, il suffirait de dire typedef element chaine. Pourtant, la question suivante incite à avoir un pointeur sur le dernier maillon :

```
struct chaine {
    element *debut;
    element *fin;
};
typedef struct chaine chaine;
```

:::

Question 3 ::: {claude kind=« modifie » raison=« le code d'origine déréférençait l quand la liste est vide (l == NULL). »}

Corrigé

J'imagine qu'il faut parcourir l ici. On renvoie le maillon ajouté, qui est aussi la nouvelle liste si l était vide.

```
element *ajouteElement(element *l, Domino *d)
{
    element *e = malloc(sizeof(element));
    e->valeur = d;
    e->suivant = NULL;
    if (l == NULL)
        return e;
```



```

while (l->suivant != NULL)
    l = l->suivant;
l->suivant = e;
return e;
}

```

...

Question 4 ::: {.claude kind=« modifie » raison=« le code d'origine comparait des structures avec != (interdit en C), ne renvoyait rien, déréférençait prec == NULL quand le domino est en tête et ne gérait pas la liste vide. »}

Corrigé

En C, on ne peut pas comparer deux structures avec == ou != : on compare ici les pointeurs, c'est-à-dire qu'on cherche ce domino précis. On renvoie la tête de la liste, qui change si on retire le premier maillon. Le domino lui-même n'est pas libéré : il appartient à l'appelant.

```

element *retireElement(element *l, Domino *d)
{
    if (l == NULL)
        return NULL;
    if (l->valeur == d)
    {
        element *suite = l->suivant;
        free(l);
        return suite;
    }
    element *prec = l;
    while (prec->suivant != NULL && prec->suivant->valeur != d)
        prec = prec->suivant;
    if (prec->suivant != NULL)
    {
        element *e = prec->suivant;
        prec->suivant = e->suivant;
        free(e);
    }
    return l;
}

```

...

Question 5 ::: {.claude kind=« modifie » raison=« la condition de boucle d'origine (|| au lieu de &&) sortait de la liste ; liste vide non gérée. »}

Corrigé

```

bool rechercheElement(element *l, Domino *d)
{
    while (l != NULL)
    {
        if (l->valeur == d)
            return true;
        l = l->suivant;
    }
    return false;
}

```

...

PROBLÈME 34

Mines – Sujet 0 – MPI

Sujet Corrigé du sujet 0 (épreuve de test) de la première épreuve d’informatique de la filière MPI du concours Mines-Ponts, publié par le Concours commun Mines-Ponts (sujet sous licence CC BY-NC-ND 3.0 FR, propriété du GIP CCMP). L’énoncé n’est pas reproduit ici : seules les réponses figurent, avec les numéros de questions du sujet. Le code est en C (C99).

Dans tout ce corrigé, on utilise les déclarations du sujet :

```
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include <limits.h>

struct maillon {
    int donnee;
    struct maillon *suivant;
};
typedef struct maillon maillon_t;
```

34.0.1 Listes triées simplement chaînées

Autour des opérations de base

Question 1 ::: {adm kind=« corrige » title=« Corrigé »}

```
maillon_t *init(void)
{
    maillon_t *l = malloc(sizeof(maillon_t));
    l->donnee = INT_MIN;
    l->suivant = malloc(sizeof(maillon_t));
    l->suivant->donnee = INT_MAX;
    l->suivant->suivant = NULL;
    return l;
}
```

:::

Question 2 ::: {adm kind=« corrige » title=« Corrigé »}

```
maillon_t *localise(maillon_t *t, int v)
{
    // pas d'erreur d'accès à suivant ici en raison de la
    // sentinelle finale
    while(t->suivant->donnee < v)
    {
        t = t->suivant;
    }
    return t;
}
```

...

Question 3 ::: {adm kind=« corrige » title=« Corrigé »}

On présente les listes sous le format usuel pour ce corrigé pour présenter les tests.

- Test 1 : insertion de 0 dans une liste vide $[-\infty, \infty] \rightarrow [-\infty, 0, \infty]$ et renvoie true.
- Test 2 : insertion en première position de 0 dans une liste non vide : $[-\infty, 1, \infty] \rightarrow [-\infty, 0, 1, \infty]$ et renvoie true.
- Test 3 : insertion en dernière position de 2 dans une liste non vide : $[-\infty, 1, \infty] \rightarrow [-\infty, 1, 2, \infty]$ et renvoie true.
- Test 4 : insertion d'une valeur présente, ici 0 : $[-\infty, 0, 1, \infty]$ ne change pas et renvoie false. :::

Question 4 ::: {adm kind=« corrige » title=« Corrigé »}

Le problème vient du `et` : comme `t` est un `maillon_t *`, on a `et` qui est un `maillon_t **`, il n'y a pas besoin de prendre l'adresse de `t`. D'ailleurs, cette adresse est sur la pile même si le maillon désigné par `t` est sur le tas.

```
bool insere_errone(maillon_t *t, int v)
{
    maillon_t *p = localise(t, v);
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    p->suivant = n;
    return true;
}
```

...

Question 5 ::: {adm kind=« corrige » title=« Corrigé »}

Le test 4 n'est pas réussi car on va ajouter sans discernement : on obtiendra avec la version erronée $[-\infty, 0, 0, 1, \infty]$ et la valeur true.

```
bool insere(maillon_t *t, int v)
{
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee == v)
        return false;
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    p->suivant = n;
    return true;
}
```

...

Question 6 ::: {adm kind=« corrige » title=« Corrigé »}

```
bool supprime(maillon_t *t, int v)
{
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee != v)
        return false;
    // ici on a p->suivant->donnee = v < oo donc
    // forcément ce maillon a un maillon suivant
    maillon_t *n = p->suivant->suivant;
    free(p->suivant);
    p->suivant = n;
    return true;
}
```

...

Question 7 ::: {adm kind=« corrige » title=« Corrigé »}

Mis à part `localise`, ces deux fonctions n'effectuent que des opérations élémentaires. Or, `localise` parcourant la liste, elle est en pire cas de complexité linéaire en la taille de la liste, en $O(|l|)$, et ainsi `insere` et `supprime` le sont également. :::

Question 8 ::: {claudie kind=« modifie » raison=« la source ne citait que la zone des variables globales ; la valeur 717 est aussi copiée sur la pile (paramètres) et dans le tas (nouveau maillon). »}

Corrigé

On distingue trois parties :

- les variables globales qui sont placées en mémoire en même temps que le programme ;
- le tas qui contient la mémoire allouée dynamiquement ;
- la pile qui se construit avec les appels de fonctions.

Ici, la variable `v` est définie en dehors de toute fonction, c'est une variable globale : la valeur 717 réside donc dans cette zone. Mais le passage par valeur la recopie aussi sur la pile (paramètre `v` de `insere`, puis de `localise`), et `insere` l'inscrit enfin dans le tas, dans le champ `donnee` du maillon alloué par `malloc`. La valeur 717 est donc inscrite dans les trois zones.

Note : il y a une distinction entre variables globales constantes et non constantes, les premières sont dans une zone marquée en lecture seule. Mais, comme `const` n'est pas au programme, je pense que c'est la réponse raisonnable de ne pas en parler.

...

Extensions des opérations de base

Question 9 ::: {adm kind=« corrige » title=« Corrigé »}

Un arbre binaire de recherche est un arbre binaire permettant de stocker un ensemble ordonné de valeurs. Dans un tel arbre, la valeur d'un nœud est toujours supérieure ou égale à toutes les valeurs de son sous-arbre gauche et inférieure ou égale à toutes les valeurs de son sous-arbre droit. Cela permet ainsi d'effectuer une recherche le long d'une branche et donc, avec une complexité linéaire en la hauteur de l'arbre.

Pour que cette complexité soit logarithmique dans le nombre de nœuds, il faut que l'arbre soit compact, de hauteur logarithmique : on parle d'arbre équilibré. On a vu que les arbres binaires de recherche rouges et noirs garantissaient cette propriété. :::

Question 10 ::: {adm kind=« corrige » title=« Corrigé »}

On effectue un parcours infixe de l'arbre binaire de recherche. Ici, l'enjeu est d'être capable de rajouter des maillons, en tête pour avoir une faible complexité.

On va donc faire en sorte que les insertions s'effectuent de droite à gauche dans l'arbre :

- on initialise une liste vide avant de réaliser un parcours récursif ;
- quand on est sur un nœud (g, x, d) :
 - on effectue un appel récursif sur `d` (**important**) ;
 - on insère `x` juste après la sentinelle de tête ;
 - on effectue un appel récursif sur `g` ;
- quand on est sur une feuille, on ne fait rien.

Chaque insertion se fait en temps constant, puisque `x` est plus petit que toutes les valeurs déjà insérées. On construit alors la liste triée en $O(|t|)$ où $|t|$ est le nombre de nœuds de l'arbre binaire de recherche `t`. L'espace utilisé est celui de la pile d'appels, borné par la hauteur de l'arbre, et la liste construite. On a donc dans tous les cas une complexité en espace également en $O(|t|)$. :::

Question 11 ::: {.claude kind=« modifie » raison=« la source donnait $1/2^{(n-i+1)}$ pour tout i , ce qui est faux pour $i = 1$ (probabilité $1/2^{(n-1)}$). »}

Corrigé

On considère ici par i -ième qu'on commence à compter à 1. Comme il y a n éléments en tout, il y en a $i - 1$ avant le i -ième et $n - i$ après.

Pour $i \geq 2$, pour que le i -ième élément soit renvoyé, il faut avoir obtenu un résultat non nul au test effectué sur le i -ième élément et 0 aux $n - i$ tests suivants, on a donc

$$P(i\text{-ième renvoyé}) = \frac{1}{2^{n-i+1}}.$$

Le premier élément est la valeur initiale de `ret` : il n'y a pas de test le concernant, et il est renvoyé si les $n - 1$ tests suivants donnent 0, avec probabilité $\frac{1}{2^{n-1}}$. On vérifie que la somme vaut bien 1 : $\frac{1}{2^{n-1}} + \sum_{i=2}^n \frac{1}{2^{n-i+1}} = \frac{1}{2^{n-1}} + 1 - \frac{1}{2^{n-1}} = 1$.

...

Question 12 ::: {adm kind=« corrige » title=« Corrigé »}

On va choisir ici de générer uniformément un indice d'un maillon puis de le chercher.

```
int random_elt(maillon_t *t)
{
    maillon_t *c = t->suivant;
    int ret = c->donnee;
    assert(ret != INT_MAX);
    int n = 0;
    maillon_t *p = c;
    while(p->donnee != INT_MAX)
    {
        n = n+1;
        p = p->suivant;
    }
    int k = random() % n;

    for(int i = 0; i < k; i++)
    {
        c = c->suivant;
    }

    return c->donnee;
}
```

...

Question 13 ::: {adm kind=« corrige » title=« Corrigé »}

Se pose le problème de la précision pour les tests d'égalité dans les insertions et suppressions. Doit-on considérer qu'il faut insérer une valeur très proche de 0 quand 0 est déjà présent dans la liste ?

Une manière de résoudre ce problème serait d'introduire une notion de seuil ou précision ϵ et de remplacer le test d'égalité entre x et y par $|x - y| \leq \epsilon$.

👉 Il faut aussi choisir de nouvelles sentinelles : `INT_MIN` et `INT_MAX` n'ont plus de sens, on peut utiliser $-\infty$ et $+\infty$ (`-INFINITY` et `INFINITY` de `math.h`), en excluant la valeur `NAN` qui n'est comparable à rien.

...

34.0.2 Listes à enjambement

Note : on parle de *skip-list* en anglais.

Question 14 ::: {adm kind=« corrige » title=« Corrigé »}

Las Vegas. ...

Question 15 ::: {adm kind=« corrige » title=« Corrigé »}

On utilise la déclaration du sujet :

```
struct chainon {
    int donnee;
    int hauteur;
    struct chainon **suivants;
};
typedef struct chainon chainon_t;
```

```
chainon_t *enjmb_init(void)
{
    chainon_t *l = malloc(sizeof(chainon_t));
    l->donnee = INT_MIN;
    l->hauteur = 1;
    l->suivants = malloc(sizeof(chainon_t *));

    chainon_t *p = malloc(sizeof(chainon_t));
    p->donnee = INT_MAX;
    p->hauteur = 1;
    p->suivants = malloc(sizeof(chainon_t *));
    p->suivants[0] = NULL;

    l->suivants[0] = p;

    return l;
}
```

...

Question 16 ::: {claude kind=« modifie » raison=« la boucle de la source s'arrêtait sur la sentinelle de fin sans la libérer (fuite de deux blocs) ; on boucle désormais tant que t != NULL. »}

Corrigé

Ici, il suffit de faire une libération normale de liste chaînée grâce au niveau 0. On doit juste faire attention à ne pas oublier la libération des tableaux de suivants, ni celle de la sentinelle de fin.

```
void enjmb_detruit(chainon_t *t)
{
    while(t != NULL)
    {
        chainon_t *temp = t->suivants[0];
        free(t->suivants);
        free(t);
        t = temp;
    }
}
```

...

Question 17 ::: {claude kind=« modifie » raison=« indicatrices A_n / A_n^c inversées dans la seconde partie de la source ; bornes des sommes précisées (a_n n'est pas entier). »}

Corrigé

Note : question de maths et pas d'informatique.

Comme H_i suit une loi géométrique de paramètre $\frac{1}{2}$, on a $P(H_i = k) = (1 - \frac{1}{2})^{k-1} \frac{1}{2} = \frac{1}{2^k}$.

Notons $a_n = 3 \log_2 n$, de sorte que $2^{a_n} = n^3$, et b_n le plus petit entier strictement supérieur à a_n : on a $a_n < b_n \leq a_n + 1$. Comme A_n est l'événement $H_{\max} > a_n$, c'est-à-dire $H_{\max} \geq b_n$, on a

$$\begin{aligned} \mathbb{E}(H_{\max} \cdot 1_{A_n}) &= \sum_{k=b_n}^{+\infty} k \mathbb{P}(H_{\max} = k) \\ &\leq \sum_{k=b_n}^{+\infty} \sum_{i=1}^n k \mathbb{P}(H_i = k) \\ &= \sum_{k=b_n}^{+\infty} n \frac{k}{2^k} \\ &= n \frac{2(b_n + 1)}{2^{b_n}} \\ &\leq n \frac{2(a_n + 2)}{2^{a_n}} = \frac{2n(3 \log_2 n + 2)}{n^3} \end{aligned}$$

Or $\frac{2(3 \log_2 n + 2)}{n^2 \log_2 n} \sim \frac{6}{n^2} \rightarrow 0$, donc cette dernière majoration est un $o(\log_2 n)$ et par majoration, on a $\mathbb{E}(H_{\max} \cdot 1_{A_n}) = o(\log_2 n)$.

Comme $1 = 1_{A_n} + 1_{A_n^c}$, on a $\mathbb{E}(H_{\max}) = o(\log_2 n) + \mathbb{E}(H_{\max} \cdot 1_{A_n^c})$.

Or, sur A_n^c , $H_{\max} \leq a_n$, donc

$$\mathbb{E}(H_{\max} \cdot 1_{A_n^c}) = \sum_{1 \leq k \leq a_n} k \mathbb{P}(H_{\max} = k) \leq a_n \sum_{1 \leq k \leq a_n} \mathbb{P}(H_{\max} = k) \leq a_n$$

Donc $\mathbb{E}(H_{\max} \cdot 1_{A_n^c}) = O(\log_2 n)$ et par additivité des O , on a

$$\mathbb{E}(H_{\max}) = O(\log_2 n)$$

...

Question 18 ::: {adm kind=« corrige » title=« Corrigé »}

Note : question de maths et pas d'informatique.

En comptant une case par donnée et par pointeur de suivant, chaque sentinelle occupe $H_{\max} + 1$ cases et le chaînon contenant u_i en occupe $1 + H_i$ (à une constante multiplicative près si on compte aussi le champ hauteur et le pointeur vers le tableau). On a donc $S = 2(H_{\max} + 1) + \sum_{i=1}^n (1 + H_i) = 2H_{\max} + n + 2 + \sum_{i=1}^n H_i$ donc

$$\mathbb{E}(S) = 2\mathbb{E}(H_{\max}) + n + 2 + \sum_{i=1}^n \mathbb{E}(H_i) = 2\mathbb{E}(H_{\max}) + n + 2 + 2n = O(\log_2 n + n) = O(n)$$

Comme de plus $S \geq n$, l'espérance de S est bien linéaire en n

Question 19 ::: {adm kind=« corrige » title=« Corrigé »}

Notons qu'on a plusieurs possibilités pour implémenter cette fonction. Le choix que je fais ici ne semble pas être celui qu'avait en tête le concepteur du sujet mais cela me semble plus simple.

```
bool enjmb_contient(chainon_t *t, int v)
{
    int k = t->hauteur - 1;

    while(t->donnee < v)
    {
        if(k > 0 && t->suivants[k]->donnee > v) {
```



```

        k = k-1;
    } else {
        t = t->suivants[k];
    }
}

return t->donnee == v;
}

```

...

Question 20 ::: {adm kind=« corrige » title=« Corrigé »}

Note : question de maths avec *un peu* d'informatique.

La première partie de la question n'est pas si facile à justifier. On remarque que les maillons du niveau h sont aussi dans le niveau $h + 1$ avec probabilité $\frac{1}{2}$. Pour compter R_h , il faut compter le nombre de maillons de niveau h entre deux maillons du niveau $h + 1$, or cela revient effectivement à effectuer une loi géométrique tronquée de paramètre $\frac{1}{2}$ avec Y indiquant ici le nombre maximum de maillons induit par le niveau supérieur. On a alors $E(R_h) \leq 1$.

On avance dans les maillons en suivant les pointeurs, comme il y a n maillons, on a au maximum $R' \leq n$.

Sur A_n^c , il y a au plus $3 \log_2 n$ niveaux, donc $E(R' \cdot 1_{A_n^c}) \leq \sum_{0 \leq h < 3 \log_2 n} E(R_h) \leq 3 \log_2 n$. Ainsi,

$$E(R) \leq E(H_{max}) + 1 + E(R' \cdot 1_{A_n^c}) + E(R' \cdot 1_{A_n}) \leq O(\log_2 n) + 1 + 3 \log_2 n + nP(A_n)$$

Or

$$nP(A_n) = n \sum_{k > 3 \log_2 n} P(H_{max} = k) \leq n \sum_{k > 3 \log_2 n} \frac{n}{2^k} \leq \frac{2n^2}{2^{3 \log_2 n}} = \frac{2n^2}{n^3} = \frac{2}{n} \rightarrow 0$$

donc ce dernier terme est négligeable devant $O(\log_2 n)$ et on a bien $E(R) = O(\log_2 n)$.

Note : ce choix du $3 \log_2 n$ pour avoir une limite nulle ici ne semble pas très important. Avec $2 \log_2 n$, on avait un $O(1)$ qui était suffisant. C'est d'ailleurs ce choix qui est fait dans la preuve de *Randomized Algorithms*. ...

Question 21 ::: {adm kind=« corrige » title=« Corrigé »}

Dans le cas des arbres binaires de recherche, on a une complexité en pire cas en $O(n)$, mais avec des ABR équilibrés, elle passe en $O(\log_2 n)$ de manière déterministe. Ici, on a une complexité aléatoire, en $O(\log_2 n)$ en moyenne.

Note : difficile d'en dire plus ici, on ne compare vraiment pas les mêmes choses. ...

34.0.3 Utilisation concurrente de listes chaînées

Question 22 ::: {claud kind=« modifie » raison=« la source affirmait que l'invariant 2 est indirectement violé ; les sentinelles ne sont jamais modifiées, il est préservé. Scénarios précisés. »}

Corrigé

Si deux fils distincts exécutent `insere` en même temps au même endroit, on peut trouver des scénarios invalidant l'invariant 1 ou l'invariant 3.

- **Invariant 1** : si les deux fils insèrent la même valeur v absente, qu'ils localisent le même maillon p et effectuent tous deux le test `p->suivant->donnee == v` (qui échoue), puis que le premier fil finit de rajouter son maillon avant que le second ne lise `p->suivant`, le pointeur `p->suivant` auquel accèdera le second fil sera le maillon nouvellement ajouté : on obtient deux maillons consécutifs de valeur v , ce qui fait doublon.

- **Invariant 3** : si on ajoute deux maillons a et b après le maillon m qui était suivi de m', on peut avoir dans l'ordre : a->suivant = m'; b->suivant = m'; m->suivant = b; m->suivant = a. On obtient alors m->a->m' et le maillon b, qui pointe vers m', n'est plus accessible depuis la tête : sa valeur n'est pas dans la liste alors que l'insertion a renvoyé true.

L'invariant 2, lui, est préservé : les sentinelles ne sont jamais modifiées ni déplacées par insere.

...

Exclusion mutuelle à gros grains

Question 23 ::: {adm kind=« corrige » title=« Corrigé »}

Attention à bien lire le type ici et à ne pas tout recoder. On utilise la déclaration du sujet :

```
#include <pthread.h>

struct liste_grosgrain {
    maillon_t *tete;
    pthread_mutex_t verrou;
};
typedef struct liste_grosgrain liste_grosgrain_t;

bool grosgrain_insere(liste_grosgrain_t *l, int v)
{
    pthread_mutex_lock(&l->verrou);
    bool res = insere(l->tete, v);
    pthread_mutex_unlock(&l->verrou);
    return res;
}
```

...

Question 24 ::: {adm kind=« corrige » title=« Corrigé »}

Comme la structure liste_grosgrain contient un pthread_mutex_t directement, celui-ci sera copié en même temps que la liste sur la pile d'appels en cas de passage par valeur. Chaque fil aura alors sa propre copie du mutex sans aucune garantie de synchronisation entre les deux copies.

En passant la liste par pointeur, on s'assure que les deux fils utilisent le même mutex. Notons qu'on aurait pu mettre un pthread_mutex_t * dans la liste pour ne pas avoir ce problème. :::

Question 25 ::: {claude kind=« modifie » raison=« la trace de la source ne fonctionne pas pour l = 0 (elle laisse 0 dans la liste) et l'impossibilité de l = n n'était pas justifiée ; ajout des deux. »}

Corrigé

Note : attention, le cast de la ligne 50 est hors programme **et inutile en C**.

Il n'y a qu'un seul verrou et aucun fil n'attend autre chose que ce verrou, qui est toujours relâché à la fin de chaque opération : il ne peut pas y avoir d'interblocage, chaque fil va ainsi terminer et le programme lui-même termine.

Chaque entier i est inséré une seule fois (par le fil i) et on tente de le supprimer une seule fois (par le fil $i - 1 \bmod n$). Il est donc présent à la fin si et seulement si la tentative de suppression a eu lieu avant l'insertion.

Le cas $l = n$ est impossible. Sinon, pour tout i , la suppression de i par le fil $i - 1$ précède l'insertion de i par le fil i , qui précède elle-même (ordre du programme) la suppression de $i + 1$ par le fil i . En faisant le tour des indices modulo n , on obtiendrait que la suppression de 0 par le fil $n - 1$ se précède strictement elle-même, ce qui est absurde.

Le cas $\ell = 0$. Si chaque fil effectue son insertion, puis que chaque fil effectue sa suppression, toutes les suppressions réussissent et la liste est vide.

Les cas $\ell \in \llbracket 1, n - 1 \rrbracket$. Pour que la liste contienne ℓ éléments à la sortie, il faut avoir échoué sur ℓ suppressions. En considérant une exécution séquentielle des fils, les ℓ premiers fils (de 0 à $\ell - 1$) peuvent, dans cet ordre, insérer leur valeur puis échouer à supprimer la suivante, qui n'est pas encore présente. Ensuite, les fils suivants font la même chose mais en partant du dernier ($n - 1$) jusqu'au fil ℓ , ce qui garantit le succès de leurs suppressions (le fil $n - 1$ supprime 0, inséré par le fil 0). À la fin, la liste contient exactement $1, \dots, \ell$.

...

Question 26 ::: {claude kind=« modifie » raison=« ajout de la réponse sur le contenu de la liste quand l'exécution termine. »}

Corrigé

Chaque fil peut supprimer son premier entier et il est donc possible qu'ils se retrouvent tous simultanément dans la deuxième boucle `while`, chacun attendant de pouvoir supprimer l'entier qu'un autre fil a déjà supprimé et ne réinsérera qu'après être lui-même sorti de cette boucle : il y a ainsi interblocage (une attente active infinie) et le programme ne termine pas.

Lorsque l'exécution termine, en revanche, la liste contient forcément les n entiers : pour chaque entier i , chacune des deux suppressions effectives de i (par les fils i et $i - 1$) est suivie, dans le même fil, d'une insertion de i . La dernière opération sur i est donc une insertion, et i est présent à la fin. Seul $\ell = n$ est donc possible.

...

Exclusion mutuelle à grains fins

Question 27 ::: {adm kind=« corrige » title=« Corrigé »}

On utilise la déclaration du sujet :

```
struct maillon_protege {
    int donnee;
    struct maillon_protege *suivant;
    pthread_mutex_t verrou;
};
typedef struct maillon_protege maillon_protege_t;

maillon_protege_t *grainfin_localise(maillon_protege_t *t, int v)
{
    pthread_mutex_lock(&t->verrou);

    // attention à le faire avant le
    // test, sinon il a pu changer

    pthread_mutex_lock(&t->suivant->verrou);
    while(t->suivant->donnee < v)
    {
        pthread_mutex_unlock(&t->verrou);
        t = t->suivant;
        pthread_mutex_lock(&t->suivant->verrou);
    }
    pthread_mutex_unlock(&t->suivant->verrou);
    return t;
}
```

...

Question 28 ::: {adm kind=« corrige » title=« Corrigé »}

Si un fil A attend un verrou acquis par le fil B, c'est forcément que B est plus loin dans `localise`, donc il ne peut dépendre de maillons acquis par A. Notons que cet argument ne fonctionnerait plus avec une liste circulaire.

Remarque : on peut facilement prouver le cas général.

Pour qu'il y ait interblocage, comme les verrous ont de bonnes propriétés, c'est nécessairement qu'il y a un fil A qui attend un verrou possédé par un fil B qui lui-même attend un autre verrou pour le déverrouiller.

Cela signifierait qu'on ait une chaîne $m_1 \rightarrow m_2 \rightarrow m_3$ avec m_1 acquis par A et m_2 acquis par B, A qui attend m_2 et B qui attend m_3 . B lui-même, pour être bloqué, a besoin d'un fil C qui a acquis m_3 et attend le maillon suivant...

Comme la liste est finie, cette situation ne peut perdurer, il y a forcément un fil qui rompt la chaîne de blocage car il ne veut pas attendre le maillon suivant car la condition du `while` n'est plus vérifiée (elle l'est au plus tard sur la sentinelle de fin, de valeur `INT_MAX > v`). :::

Question 29 ::: {adm kind=« corrige » title=« Corrigé »}

- Invariant 1 : si on insère la même valeur, les deux fils ne peuvent sortir de `localise` avec le même maillon en même temps, car un des deux fils l'aura verrouillé avant de faire l'insertion puis de le libérer. Suite à cela, l'autre fil, qui est encore dans le `while`, va voir le maillon nouvellement inséré comme suivant du maillon renvoyé. Cela va alors placer le fil dans le cas de non insertion.
- Invariant 3 : on rajoute un nouveau maillon en verrouillant le maillon précédent, ce qui oblige à ce que les deux raccords soient valides et à ce que la structure de la liste soit préservée. :::

Question 30 ::: {claude kind=« modifie » raison=« l'exemple de la source faisait supprimer 12 par un fil « des multiples de 4 », or 4 n'est jamais rencontré par le fil principal ; remplacé par le fil des multiples de 2. »}

Corrigé

Tout d'abord, on remarque que l'utilisation d'une liste à gros grains revient à supprimer toute possibilité de concurrence dans les parcours : cela revient à réaliser une file de valeurs à traiter dont on doit barrer les multiples.

Avec une liste à grains fins, plusieurs fils peuvent supprimer des nombres en même temps : pendant que le fil supprime le multiple 9 de 3, un autre fil peut supprimer le multiple 12 de 2. Quand le fil des multiples de 3 avancera, il ne trouvera pas 12 à supprimer. Le désavantage ici va être la synchronisation forcée entre deux fils n'ayant aucun multiple en commun, qui doivent se suivre maillon par maillon.

Pour que le fil principal ne rencontre que des nombres premiers, il doit réaliser une attente du thread barrant les multiples de p avant de rencontrer p^2 (les autres multiples ayant déjà été barrés si on procède ainsi).

Les suppressions effectives ne vont pas être réparties équitablement car un nombre composé avec un petit diviseur premier sera barré très tôt. Or, il y a une moitié de multiples de 2, un tiers de multiples de 3... Les threads barrant les multiples de grands premiers pourraient n'avoir que très peu de travail effectif de suppression.

:::

PROBLÈME 35

Mines 2023 - Épreuve 2

Sujet Corrigé partiel (questions 1 à 23 sur 38) de la deuxième épreuve d'informatique de la filière MPI du concours Mines-Ponts 2023, consacrée à la complexité de Kolmogoroff (sujet du Concours commun Mines-Ponts, sous licence CC BY-NC-ND 3.0 FR, propriété du GIP CCMP). L'énoncé n'est pas reproduit ici : seules les réponses figurent, avec les numéros de questions du sujet. Le code est en OCaml.

On rappelle seulement les notations : $K(y)$ est la longueur de la plus courte chaîne x dont l'évaluation comme code OCaml renvoie la chaîne y , et y_0 est l'écriture décimale de $10^{(10^{10})}$, un 1 suivi de 10^{10} zéros.

35.0.1 Complexité de Kolmogoroff

Un premier exemple

Question 1 :: {.claude kind=« modifie » raison=« la source donnait $|x_0| = |y_0| = 2 + 10^{(10^{10})}$, confondant la valeur de l'entier et la longueur de son écriture ; y_0 a $10^{10} + 1$ caractères. »}

Corrigé

La chaîne y_0 comporte $10^{10} + 1$ caractères. La description x_0 est le littéral de chaîne correspondant, guillemets compris, donc $|x_0| = |y_0| + 2$ et on a ainsi $K(y_0) \leq |x_0| = 10^{10} + 3$ (**attention** à ne pas oublier les guillemets).

Plus généralement, si y est une chaîne de longueur n (sans caractère à échapper), on a $K(y) \leq 2 + n$.

:::

Question 2 :: {.claude kind=« modifie » raison=« la source affirmait que le programme fait moins de 100 caractères ; écrit sur une ligne sans espace superflu, il en fait 121 (vérifié). »}

Corrigé

En notant $n' = \lfloor n/2 \rfloor$, on a

$$10^n = \begin{cases} (10^{n'})^2 & \text{si } n \text{ pair} \\ 10 \times (10^{n'})^2 & \text{sinon} \end{cases}$$

La question met sur la voie d'une implémentation récursive non terminale, mais l'absence du `rec` dans le code fourni par l'énoncé fait douter. Je le fais ainsi dans la mesure où l'hypothèse de mémoire infinie devrait se transmettre à la capacité de la pile d'appels. On aurait souhaité que le sujet soit plus précis ici car c'est un élément important d'OCaml : on pouvait très bien supposer une mémoire infinie mais une pile bornée.

```
let rec exp10 n =
  if n = 0
  then 1
```

```

else let x = exp10 (n/2) in
  if n mod 2 = 0 then x * x else 10 * x * x
in string_of_int (exp10 (exp10 10))

```

Écrit sur une seule ligne et sans espace superflu, ce programme fait 121 caractères :

```

let rec exp10 n = if n = 0 then 1 else let x = exp10 (n/2) in if n mod 2 = 0 then x * x else 10 * x * x in
string_of_int (exp10 (exp10 10))

```

On peut donc affirmer $K(y_0) \leq 121$, de l'ordre de 10^2 , ce qui est considérablement mieux que la borne $10^{10} + 3$ de la question 1.

...

Question 3 ::: {.claude kind=« modifie » raison=« la source invoquait un dépassement de pile ; la profondeur de récursion de exp10 n'est que d'environ $\log_2(10^{10}) \approx 34$. Le problème est le dépassement des entiers (et la taille du résultat). »}

Corrigé

La récursivité non terminale n'est pas un problème ici : exp10 divise son argument par deux à chaque appel, la profondeur de récursion est donc d'environ $\log_2(10^{10}) \approx 34$. En revanche, l'utilisation des entiers en OCaml rend ce calcul très vite inexact quand on va dépasser la capacité maximale d'un entier machine (*pas la peine de parler du bit réservé aux unboxed int*) : les calculs se font modulo 2^{63} et, comme 2^{63} divise 10^k dès que $k \geq 63$, le programme renvoie en pratique la chaîne "0".

Même avec des entiers de taille arbitraire, le résultat comporterait $10^{10} + 1$ chiffres, soit une dizaine de gigaoctets, et son calcul demanderait beaucoup de temps et de mémoire.

...

Quelques propriétés

Question 4 ::: {.claude kind=« modifie » raison=« la formule de la source était inachevée ; complétée d'après le code de l'auteur, avec la justification de la bijectivité. »}

Corrigé

On peut identifier un caractère à un chiffre en base 256 ; ainsi, chaque entier non nul sera représenté par son écriture propre en base 256 (chiffre de poids faible en premier). Pour 0, il suffit de le représenter par la chaîne vide. On a naturellement une bijection, par unicité de l'écriture en base 256, avec les écritures propres, c'est-à-dire ne terminant pas par le caractère 0.

Cela pose un problème ici, car on veut toutes les chaînes, même celles finissant par le caractère 0. Pour résoudre cela, en notant c_k le k -ième caractère de Σ pour $k \in \llbracket 0, 255 \rrbracket$, on considère une version modifiée

$$\varphi(n) = \begin{cases} "" & \text{si } n = 0 \\ c_{(n-1) \bmod 256} \cdot \varphi\left(\left\lfloor \frac{n-1}{256} \right\rfloor\right) & \text{si } n \geq 1 \end{cases}$$

C'est une bijection : toute chaîne non vide s'écrit de manière unique $c_a \cdot s$ avec $a \in \llbracket 0, 255 \rrbracket$ et $s \in \Sigma^*$, et $n \mapsto ((n-1) \bmod 256, \lfloor (n-1)/256 \rfloor)$ est une bijection de $\mathbb{N}^*$ sur $\llbracket 0, 255 \rrbracket \times \mathbb{N}$. Par récurrence sur la longueur, la réciproque est donnée par $\varphi^{-1}("") = 0$ et $\varphi^{-1}(c_a \cdot s) = 1 + a + 256 \varphi^{-1}(s)$.

Le code OCaml correspondant est :

```

let rec phi n =
  if n = 0
  then ""
  else String.make 1 (Char.chr ((n-1) mod 256)) ^ phi ((n-1) / 256)

```

...

Question 5 ::: {adm kind=« corrige » title=« Corrigé »}

Le code est :

```
let psi m =
  let rec cherche n =
    if kolmogoroff (phi n) >= m
    then n
    else cherche (n+1)
  in cherche 0
```

La recherche termine : il n'y a qu'un nombre fini de chaînes de longueur $< m$, donc un nombre fini de chaînes y telles que $K(y) < m$, et φ est une bijection. :::

Question 6 ::: {adm kind=« corrige » title=« Corrigé »}

Soit $m \in \mathbb{N}$, on a $\psi(m) \in \{n \in \mathbb{N} \mid K(\varphi(n)) \geq m\}$ donc $K(\varphi(\psi(m))) \geq m$.

On peut considérer le code

```
let kolmogoroff y = <code de kolmogoroff> in
let rec phi n = <code de phi> in
let psi m = <code de psi> in
phi (psi <m>)
```

où $\langle m \rangle$ est une représentation décimale de m . Ce code est une description de $\varphi(\psi(m))$ de longueur $c + \lfloor \log_{10} m \rfloor + 1$, où la constante c ne dépend pas de m : on a donc $K(\varphi(\psi(m))) = O(\log m)$.

Par majoration, on en déduit $m = O(\log m)$, ce qui est impossible. La seule hypothèse ayant été l'existence de kolmogoroff, on en déduit qu'une telle fonction ne peut exister. :::

Question 7 ::: {adm kind=« corrige » title=« Corrigé »}

Il s'agit d'un compilateur. On parle parfois de *transpileur* dans ce contexte source vers source. :::

Remarque 35.1 C'est un peu étonnant de ne pas avoir d'hypothèse ici sur l'existence d'une description. Vu l'énoncé, la fonction `let d x = "argh"` est un décompresseur qui n'admet de description que pour une seule chaîne.

Question 8 ::: {adm kind=« corrige » title=« Corrigé »}

On peut considérer le programme suivant :

```
let decompresseur x = <code du decompresseur>
in decompresseur "<z>"
```

où $\langle z \rangle$ est une description minimale de y par rapport à D , en supposant que cela existe (sinon $K_D(y) = +\infty$ et il n'y a rien à montrer).

On a alors un code de longueur $K_D(y) + c_D$ avec c_D la longueur du reste du code, guillemets compris, qui ne dépend pas de z , et donc pas de y .

Ainsi, $K(y) \leq K_D(y) + c_D$.

👉 On suppose ici, comme à la question 1, que z s'écrit tel quel entre guillemets. Si z contient des guillemets ou des $\backslash$, il faut les échapper ($\backslash$ et $\backslash \backslash$), ce qui peut doubler sa longueur : on obtient seulement $K(y) \leq 2K_D(y) + c_D$ dans le pire cas. L'énoncé néglige visiblement ce point.

...

35.0.2 Estimation de la complexité grâce au décompresseur de Huffman

Question 9 ::: {adm kind=« corrige » title=« Corrigé »}

On réécrit avec des sauts de ligne et des indentations pour une meilleure lisibilité :

```
let rec t e =
  if e = 0
  then 1
  else let n = e - 1 in
        10 * t n
in
let n = t 10 in
t n
```

On remarque tout de suite avec le cas de base que $t : \text{int} \rightarrow \text{int}$ et donc l'expression $t\ n$ est de type int . :::

Question 10 ::: {adm kind=« corrige » title=« Corrigé »}

La fonction t effectue un calcul naïf de 10^n , on renvoie donc $10^{(10^{10})}$. :::

Question 11 ::: {adm kind=« corrige » title=« Corrigé »}

Impénétrable ? Ça se voit que le rédacteur ne corrige pas souvent des copies d'élèves de prépa, ça me semble très compréhensible.

Pour une réponse sérieuse : la nomenclature des fonctions et variables (noms d'une lettre, n utilisé pour deux variables différentes), et l'absence de sauts de ligne et d'indentation. :::

Question 12 ::: {claude kind=« modifie » raison=« le code de la source oubliait la clé dans Hashtbl.add et utilisait une table h inexistante. »}

Corrigé

Le code OCaml :

```
let count x =
  let occ = Hashtbl.create 256 in
  for i = 0 to String.length x - 1 do
    Hashtbl.add occ x.[i]
    (1 + match Hashtbl.find_opt occ x.[i] with
      | None -> 0
      | Some k -> k)
  done;
  occ
```

:::

Remarque 35.2 Le but ici était d'évaluer votre pratique des tables de hachage, mais avec des caractères, autant faire un tableau de 256 cases.

Par contre, l'usage de `add` (la seule fonction d'ajout rappelée par l'énoncé) et non de `replace` va conserver les associations précédentes : `find_opt` renvoie bien la plus récente, donc le bon nombre d'occurrences, mais la table de hachage sera de taille $|x|$ et non bornée par 256.

Question 13 ::: {adm kind=« corrige » title=« Corrigé »}

Un arbre est donné ici, je n'ai pas respecté la consigne car je ne l'ai pas fait à la main... :::

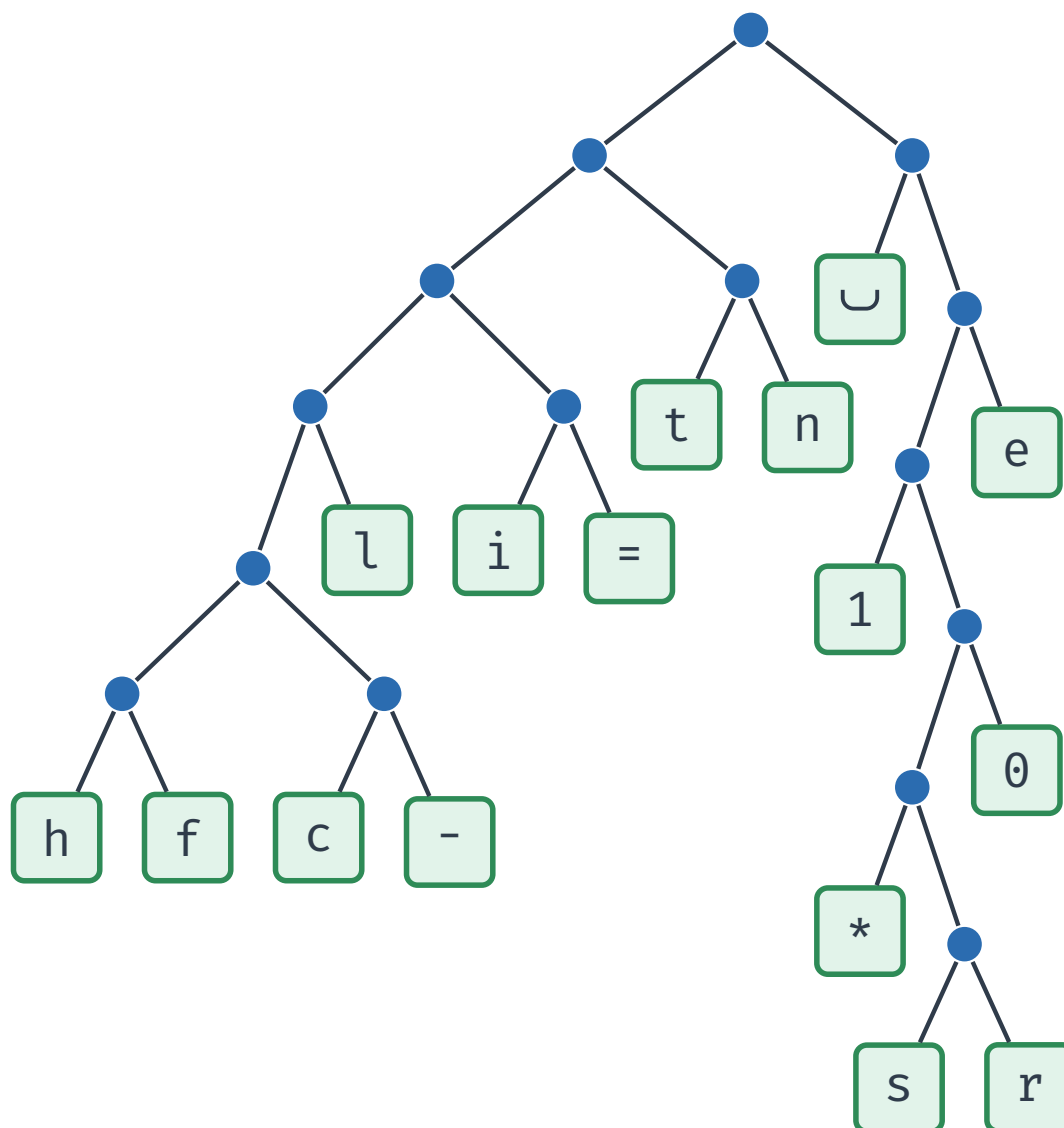


Figure 35.1 — Un arbre de Huffman de x_0 (gauche : 0, droite : 1) ; $_$ est l'espace.

La feuille $_$ correspond au caractère espace. En lisant un $_0$ pour une branche gauche et un $_1$ pour une branche droite, la profondeur d'une feuille est la longueur du code de son caractère : 2 pour l'espace, 3 pour $_t$, $_n$ et $_e$, 4 pour $_l$, $_i$, $_='$ et $_1$, 5 pour $_0$, 6 pour $_h$, $_f$, $_c$, $_-'$ et $_*$, 7 pour $_s$ et $_r$.

Question 14 ::: {adm kind=« corrige » title=« Corrigé »}

On a $|z_0| = \sum_c |x_0|_c \text{prof}(c) = 239$ où $\text{prof}(c)$ est la profondeur de la feuille c dans l'arbre : avec les occurrences données par l'énoncé, $19 \times 2 + (8 + 8 + 10) \times 3 + (4 + 4 + 4 + 4) \times 4 + 3 \times 5 + 5 \times 6 + 2 \times 7 = 239$.

Comme $H(z_0) = y_0$, on a alors directement $K_H(y_0) \leq 239$ avec cette description, mais c'est un peu étonnant comme choix de placer l'arbre dans H , car H dépend de y_0 . :::

35.0.3 Interlude

Question 15 ::: {claude kind=« modifie » raison=« la source définissait la portée comme la durée de vie ; les deux notions sont distinguées. »}

Corrigé

On reproduit le code :

```
let inc s = s := "a" ^ (!s); !s

let new_string1 =
  let seed1 = ref "#" in
  fun () -> inc seed1

let new_string2 () =
  let seed2 = ref "#" in
  inc seed2

let seed3 = ref "#"
let new_string3 () =
  inc seed3
```

La portée d'une variable est la partie du programme où son nom est accessible ; il ne faut pas la confondre avec la durée de vie en mémoire de la valeur qu'elle désigne.

La portée de `seed1` est la fonction anonyme `fun () -> inc seed1` : `seed1` est inaccessible ailleurs. Mais la référence reste vivante dans la clôture de la fonction anonyme, pendant tout le reste du programme : elle est créée une seule fois, à la définition de `new_string1`, et partagée par tous les appels.

Dans `new_string2`, la variable `seed2` n'existe que dans le corps de la fonction et une nouvelle référence est créée à chaque appel : sa portée, comme sa durée de vie, s'arrête à la fin de l'appel.

Pour `seed3`, elle est définie globalement et est donc vivante **et accessible** tout au long du programme à partir de sa définition.

...

Question 16 ... {adm kind=« corrige » title=« Corrigé »}

La fonction `new_string2` est constante (elle renvoie toujours "a#") comme on l'a expliqué précédemment : elle ne respecte pas la spécification. Un test pourrait être :

```
let test_non_constant f =
  f () <> f ()
```

qui renvoie `true` pour `new_string1` et `new_string3` et `false` pour `new_string2`. ...

Question 17 ... {claude kind=« modifie » raison=« le code de la source se terminait par « in » suivi d'un commentaire, ce qui ne compile pas. »}

Corrigé

En exposant la variable `seed3`, on prend le risque qu'elle soit modifiée en dehors de `new_string3`, ce qui pourrait modifier son comportement : `new_string1` est donc préférable.

Par exemple :

```
let () =
  let x = new_string3 () in
  seed3 := "#";
  let y = new_string3 () in
  (* ici x = y car on a réinitialisé seed3 *)
  assert (x = y)
```

(si c'est le premier appel à `new_string3`, sinon `x` vaut "aa...a#").

...

35.0.4 Estimation de la complexité grâce au décompresseur de De Bruijn

Construction syntaxique d'un langage

Question 18 ::: {adm kind=« corrige » title=« Corrigé »}

On peut considérer $(a \mid b)^* \#$. :::

Question 19 ::: {claude kind=« modifie » raison=« la source donnait l'automate minimal à deux états au lieu de l'automate de Glushkov demandé ; ajout de l'automate de Glushkov (figure mines_1_2_19_glushkov), l'automate de l'auteur est conservé comme remarque. »}

Corrigé

On linéarise l'expression en $(a_1 \mid b_2)^* \#_3$. Les premières lettres sont a_1, b_2 et $\#_3$, la seule dernière lettre est $\#_3$, et les facteurs de longueur 2 sont $a_1 a_1, a_1 b_2, b_2 a_1, b_2 b_2, a_1 \#_3$ et $b_2 \#_3$. L'automate de Glushkov a donc quatre états : l'état initial 0 et un état par lettre linéarisée, $\#_3$ étant le seul état final.

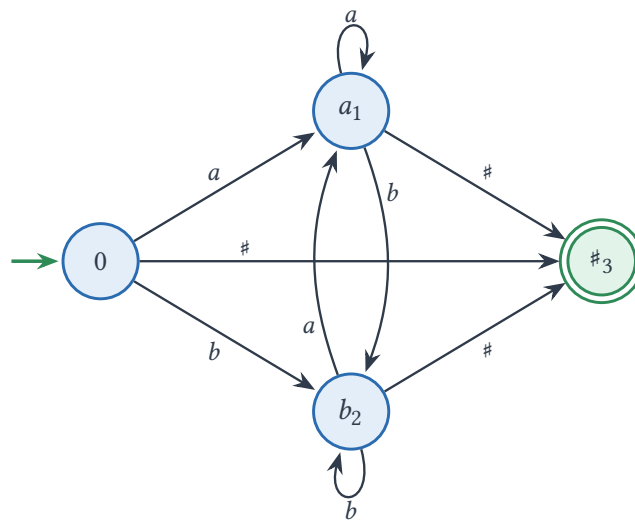


Figure 35.2 — Automate de Glushkov de $(a_1 \mid b_2)^* \#_3$.

La précision sur Glushkov est un peu étonnante vu l'expression régulière élémentaire : les états $\$0$, $\$a_1$ et $\$b_2$ ont exactement les mêmes transitions sortantes et ne sont pas finaux. En les fusionnant, on obtient l'automate suivant, qui convient aussi :

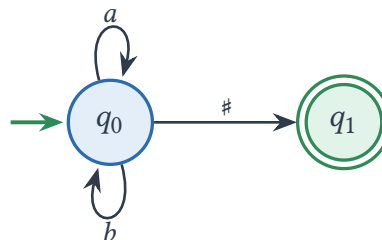


Figure 35.3 — Automate à deux états obtenu en fusionnant 0, a_1 et b_2 .

...

Question 20 ::: {adm kind=« corrige » title=« Corrigé »}

Le code OCaml :

```
exception SyntaxError
```

```

let rec parseV w =
  match w with
  | 'b' :: q
  | 'a' :: q ->
    let v, s = parseV q in
    (String.make 1 (List.hd w) ^ v, s)
  | '#' :: q -> ("#", q)
  | _ -> raise SyntaxError

```

:::

Question 21 ::: {.claude kind=« modifie » raison=« la source n'énonçait que le plan de la preuve ; preuve complétée. »}

Corrigé

L'indication du sujet est étrange, on parle d'induction structurelle sur deux nombres ?

Pour un mot u , notons $h(u) = |u|_(' - |u|_(')$ le nombre de parenthèses ouvrantes moins le nombre de parenthèses fermantes.

📌 **Lemme.** Pour tout $w \in L(G_1)$, on a $h(w) = 0$, tout préfixe u de w vérifie $h(u) \geq 0$, et tout préfixe non vide et strict u de w vérifie $h(u) \geq 1$.

On le montre par récurrence forte sur la longueur de w (c'est une induction structurelle sur la dérivation).

- Si $w = _$, on a $h(w) = 0$ et w n'a pas de préfixe non vide strict.
- Sinon $w = (w_1 w_2)$ avec $w_1, w_2 \in L(G_1)$ plus courts. On a $h(w) = 1 + 0 + 0 - 1 = 0$. Un préfixe non vide strict u de w s'écrit $u = (v$ où v est un préfixe de $w_1 w_2$ (le dernier caractère $)$ de w n'est pas dans u). Soit v est un préfixe de w_1 , et $h(v) \geq 0$ par hypothèse de récurrence ; soit $v = w_1 v'$ avec v' préfixe de w_2 , et $h(v) = 0 + h(v') \geq 0$. Dans les deux cas, $h(u) = 1 + h(v) \geq 1$.

Soient alors $w, w' \in L(G_1)$ non vides avec w préfixe strict de w' . Par le lemme appliqué à w' , on aurait $h(w) \geq 1$, alors que $h(w) = 0$: c'est absurde. Le langage $L(G_1)$ est donc sans préfixe.

:::

Question 22 ::: {.claude kind=« modifie » raison=« la source affirmait une erreur d'énoncé (grammaire ambiguë) en lisant le terminal $_$ comme le mot vide et en ajoutant une règle $T \rightarrow (T)$ qui n'existe pas ; la grammaire G_1 n'est pas ambiguë, preuve donnée. »}

Corrigé

📌 On montre par récurrence forte sur $|w|$ que tout mot $w \in L(G_1)$ admet un unique arbre d'analyse.

- Si w commence par $_$, la première règle appliquée ne peut être que $T \rightarrow _$ (l'autre produit un mot commençant par $($), donc $w = _$ et l'arbre est une feuille.
- Si w commence par $($, la première règle appliquée est forcément $T \rightarrow (TT)$ et $w = (w_1 w_2)$ avec $w_1, w_2 \in L(G_1)$. Cette décomposition est unique : si $w_1 w_2 = w'_1 w'_2$ avec $w_1, w'_1, w_2, w'_2 \in L(G_1)$, alors l'un des mots w_1, w'_1 est préfixe de l'autre ; ces mots étant non vides, la question 21 impose $w_1 = w'_1$, puis $w_2 = w'_2$. Par hypothèse de récurrence, w_1 et w_2 ont un unique arbre d'analyse, donc w aussi.

Attention, $_$ est ici un symbole terminal et non le mot vide : avec une règle $T \rightarrow \epsilon$, le langage ne serait plus sans préfixe.

:::

Question 23 ::: {.claude kind=« modifie » raison=« la source répondait seulement « grammaire du lambda-calcul » sans démontrer la non-ambiguïté demandée ; preuve ajoutée. »}

Corrigé

Il s'agit de la grammaire du λ -calcul : $\text{var} \rightarrow \tau$ est une abstraction et $(\tau \ \tau)$ une application.

👉 Ici var et $\rightarrow$ sont des symboles terminaux (de longueur 1). Le langage $L(G_2)$ n'est plus sans préfixe (var est préfixe de $\text{var} \rightarrow \text{var}$), il faut donc adapter l'argument précédent.

Lemme. Si $u, u' \in L(G_2)$ et u est un préfixe strict de u' , alors le symbole qui suit u dans u' est $\rightarrow$.

On reprend la fonction h de la question 21 : comme précédemment, tout mot de $L(G_2)$ vérifie $h = 0$, ses préfixes vérifient $h \geq 0$, et les préfixes non vides stricts d'un mot de la forme $(t_1 t_2)$ vérifient $h \geq 1$ (la règle $\text{var} \rightarrow \tau$ n'ajoute pas de parenthèse). On montre alors le lemme par récurrence forte sur $|u'|$:

- si $u' = \text{var}$, il n'a pas de préfixe strict non vide, or u est non vide ;
- si $u' = (t_1 t_2)$, alors $h(u) \geq 1$, ce qui contredit $u \in L(G_2)$: ce cas n'arrive pas ;
- si $u' = \text{var} \rightarrow t$, alors u commence par var : soit $u = \text{var}$, suivi de $\rightarrow$ dans u' ; soit $u = \text{var} \rightarrow v$ avec $v \in L(G_2)$ préfixe strict de t (un mot de $L(G_2)$ ne se termine jamais par $\rightarrow$, donc $u \neq \text{var} \rightarrow$), et par hypothèse de récurrence v est suivi de $\rightarrow$ dans t , donc u aussi dans u' .

Non-ambiguïté. On procède par récurrence forte sur la longueur d'un mot $w \in L(G_2)$.

- Si w commence par var , la première règle est $T \rightarrow \text{var}$ si $w = \text{var}$, et $T \rightarrow \text{var} \rightarrow T$ sinon ; dans ce second cas $w = \text{var} \rightarrow w'$ avec w' unique, qui a un unique arbre par hypothèse de récurrence.
- Si w commence par $($, la première règle est $T \rightarrow (TT)$ et $w = (w_1 w_2)$. Si $w_1 w_2 = w'_1 w'_2$ avec $w_1 \neq w'_1$, l'un, par exemple w_1 , est préfixe strict de l'autre, et par le lemme le symbole suivant, qui est le premier symbole de w_2 , est $\rightarrow$. Or un mot de $L(G_2)$ commence par var ou $($: c'est absurde. La décomposition est donc unique et on conclut par hypothèse de récurrence.

...

PROBLÈME 36

Centrale 2022 - Option informatique

36.1 Centrale 2022 - Option informatique

Sujet Corrigé du sujet d'informatique (option informatique) du concours Centrale-Supélec 2022, filière MP. L'énoncé n'est pas reproduit ici : il faut le consulter à côté de ce corrigé. Les questions sont numérotées comme dans l'énoncé.

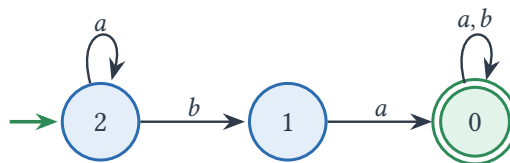
36.1.1 Mots et automates

Miroir d'un mot et automate transposé

Question 1 ::: {adm kind=« corrige » title=« Corrigé »}

On va décrire ces langages par des expressions rationnelles : L_1 est dénoté par $(a|b)^*aba^*$ et $\tilde{L}_1$ par $a^*ba(a|b)^*$. :::

Question 2 ::: {adm kind=« corrige » title=« Corrigé »} :::



Question 3 ::: {adm kind=« corrige » title=« Corrigé »}

On pose $I' = F$, $F' = I$ et pour $q, q' \in Q$, $a \in \Sigma$, $(q, a, q') \in T \Leftrightarrow (q', a, q) \in T'$.

Soit encore, $T' = \{(q', a, q) \mid (q, a, q') \in T\}$. On a donc $q \xrightarrow{a} q'$ dans A ssi $q' \xrightarrow{a} q$ dans $\tilde{A}$ et, par une récurrence rapide sur la longueur des chemins, on en déduit qu'un chemin est dans A ssi son renversement est dans $\tilde{A}$.

Comme on a échangé les états finaux et initiaux, on en déduit que $L_{\tilde{A}} = \tilde{L}$. :::

Question 4 ::: {adm kind=« corrige » title=« Corrigé »}

```

let transpose a =
{
  nb = a.nb;
  init = a.final;
  final = a.init;
  trans = List.map
    (fun (q,a,q') -> (q',a,q)) a.trans
}
  
```

:::

Question 5 ::: {adm kind=« corrige » title=« Corrigé »}

La complexité de cette fonction est déterminée par le parcours de la table de transitions, on est donc en $O(|T|)$

Palindromes et rationalité

Question 6 ::: {.adm kind=« modifie » raison=« la boucle allait jusqu'à $n / 2$ et lisait $w[0]$ sur le mot vide (Invalid_argument) ; borne remplacée par $n / 2 - 1$. »}

Corrigé

```
let palindrome w =
  let n = String.length w in
  let pal = ref true in
  for i = 0 to n / 2 - 1 do
    if w.[i] <> w.[n-i-1]
    then pal := false
  done;
  !pal
```

...

Question 7 ::: {.adm kind=« corrige » title=« Corrigé »}

Si $\Sigma = \{a\}$ alors $\text{Pal}(\Sigma) = \Sigma^*$ car tous les mots sont des palindromes. Or, Σ^* est évidemment rationnel. ...

Question 8 ::: {.adm kind=« corrige » title=« Corrigé »}

S'il existe $a, b \in \Sigma$ tels que $a \neq b$, on sait que le langage $L = \{a^n b a^n \mid n \in \mathbb{N}\}$ n'est pas rationnel (voir cours).

Or, $a^* b a^*$ est rationnel et si $\text{Pal}(\Sigma)$ était rationnel, alors par stabilité des langages rationnels par intersection, on aurait $\text{Pal}(\Sigma) \cap a^* b a^* = L$ rationnel. Donc, $\text{Pal}(\Sigma)$ est non rationnel. ...

Question 9 ::: {.adm kind=« corrige » title=« Corrigé »}

$L_{q,q'}$ est directement reconnu par l'automate $(Q, \{q\}, \{q'\}, T)$.

On a

$$L_A = \bigcup_{q \in I} \bigcup_{q' \in F} L_{q,q'}$$

car un mot reconnu est l'étiquette d'un chemin d'un état de I vers un état de F

Question 10 ::: {.adm kind=« corrige » title=« Corrigé »}

On remarque que $(\Sigma^2)^*$ est l'ensemble des mots de longueur paire car ceux-ci peuvent se découper en facteurs disjoints de longueur 2.

On a $\tilde{u}\tilde{u} = \tilde{u}\tilde{u} = u\tilde{u}$ et $|u\tilde{u}| = 2 \mid u|$ est pair, donc $\{u\tilde{u} \mid u \in \Sigma^*\} \subset \text{Pal}(\Sigma) \cap (\Sigma^2)^*$.

Soit $w \in \text{Pal}(\Sigma) \cap (\Sigma^2)^*$, si on note $w = a_1 \dots a_{2p}$ et $w' = a_1 \dots a_p$, $w'' = a_{p+1} \dots a_{2p}$, comme w est un palindrome, $a_i = a_{2p-i+1}$ et donc $w'' = \tilde{w}'$. Ainsi, $w = w' w'' = w' \tilde{w}'$ et on en déduit l'autre inclusion. ...

Question 11 ::: {.adm kind=« corrige » title=« Corrigé »}

$$D(a^* b) = \{a^n b b a^n \mid n \in \mathbb{N}\}$$

Ici, on remarque que les palindromes dans $a^* b^* a^*$ sont de la forme $a^n b^m a^n$ et ils sont donc de longueur paire ssi m est pair. On en déduit qu'ils s'écrivent alors $a^n b^{2p} a^n = a^n b^p a^n \tilde{b}^p$.

$$R(a^* b^* a^*) = \{a^n b^p \mid n \in \mathbb{N}, p \in \mathbb{N}\} = a^* b^*$$

...

Question 12 ::: {.claude kind=« modifie » raison=“la condition sur les transitions de A_R portait sur (q_1,a,q_2) au lieu de (q_1,a,q'_1) ; corrigée.”}

Corrigé

On remarque que a^*b est rationnel mais $D(a^*b)$ ne l'est pas, on se ramène encore à $\{a^n b^n \mid n \in \mathbb{N}\}$.

On peut considérer l'automate où on avance en même temps dans A depuis un état initial dans un sens et depuis un état final dans l'autre sens. On pose ainsi $A_R = (Q \times Q, I \times F, \Delta, T_R)$ où $\Delta = \{(q, q) \mid q \in Q\}$ et

$$T_R = \{((q_1, q_2), a, (q'_1, q'_2)) \mid (q_1, a, q'_1) \in T \wedge (q'_2, a, q_2) \in T\}$$

Un mot w est alors reconnu par A_R ssi il existe $i \in I, f \in F$ et $q \in Q$ avec $i \xrightarrow{w} q$ et $q \xrightarrow{\tilde{w}} f$ dans A , c'est-à-dire ssi $w\tilde{w} \in L$.

On peut aussi remarquer que si $w = a_1 \dots a_n \in R(L)$, on a un chemin

$$i \xrightarrow{a_1} \dots \xrightarrow{a_n} q \xrightarrow{a_n} \dots \xrightarrow{a_1} f$$

pour $i \in I$ et $f \in F$. On a donc $w \in L_{i,q}$ et $\tilde{w} \in L_{q,f} \Leftrightarrow w \in \widetilde{L_{q,f}}$. Réciproquement, un tel w vérifie $w\tilde{w} \in L$.

Ainsi,

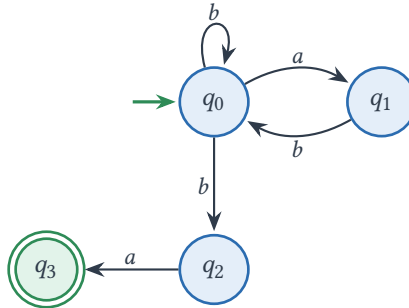
$$R(L) = \bigcup_{i \in I} \bigcup_{f \in F} \bigcup_{q \in Q} L_{i,q} \cap \widetilde{L_{q,f}}$$

qui est rationnel par stabilité par unions finies et intersection des langages rationnels (et par miroir, question 3).

...

Déterminisation

Question 13 ::: {.adm kind=« corrige » title=« Corrigé »} :::



Question 14 ::: {.adm kind=« corrige » title=« Corrigé »}

L'automate miroir est :

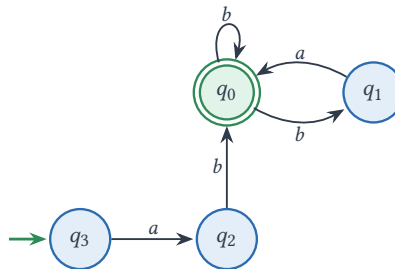


Figure 36.1 — Automate miroir : transitions retournées, q_3 initial et q_0 final.

On applique la méthode vue en classe :

```

::: {.mtable cols="4"}
::: cellule
\
:::
::: cellule
\
:::
::: cellule
$a$
:::
::: cellule
$b$
:::
::: cellule
 $\rightarrow$ 
:::
::: cellule
 $e_0 = \{q_3\}$ 
:::
::: cellule
 $e_1$ 
:::
::: cellule
 $\emptyset$ 
:::
::: cellule
\
:::
::: cellule
 $e_1 = \{q_2\}$ 
:::
::: cellule
 $\emptyset$ 
:::
::: cellule
 $e_2$ 
:::
::: cellule
 $*$ 
:::
::: cellule
 $e_2 = \{q_0\}$ 
:::
::: cellule
 $\emptyset$ 
:::
::: cellule
 $e_3$ 
:::
::: cellule
 $*$ 
:::
::: cellule
 $e_3 = \{q_0, q_1\}$ 
:::
::: cellule
 $e_2$ 
:::
::: cellule
 $e_3$ 
:::
:::

```

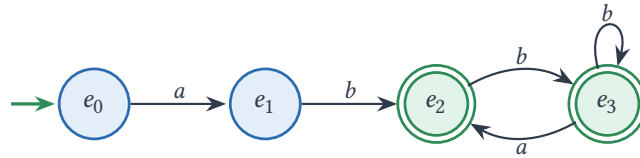


Figure 36.2 — Automate déterminisé (états e_0 à e_3 du tableau).

Question 15 ::: {adm kind=« corrige » title=« Corrigé »}

L'automate miroir est :

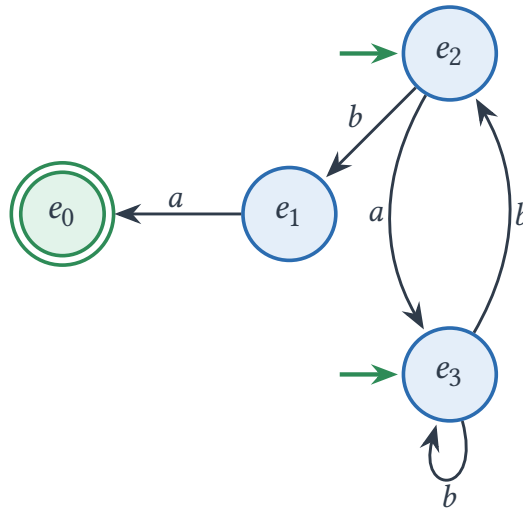


Figure 36.3 — Automate miroir : transitions retournées, e_2 et e_3 initiaux, e_0 final.

On détermine :

```

::: {.mtable cols="4"}
::: cellule
\
:::
::: cellule
\
:::
::: cellule
$a$
:::
::: cellule
$b$
:::
::: cellule
$\rightarrow$
:::
::: cellule
$q_0 = \{ e_2, e_3 \}$
:::
::: cellule
$q_1$
:::
::: cellule
$q_2$
:::
::: cellule
\
:::
::: cellule
$q_1 = \{ e_3 \}$

```

```

:::
::: cellule
$\emptyset$
:::
::: cellule
$q_0$
:::
::: cellule
\
:::
::: cellule
$q_2 = \{ e_1, e_2, e_3 \}$
:::
::: cellule
$q_3$
:::
::: cellule
$q_2$
:::
::: cellule
$*$
:::
::: cellule
$q_3 = \{ e_0, e_3 \}$
:::
::: cellule
$\emptyset$
:::
::: cellule
$q_0$
:::

```

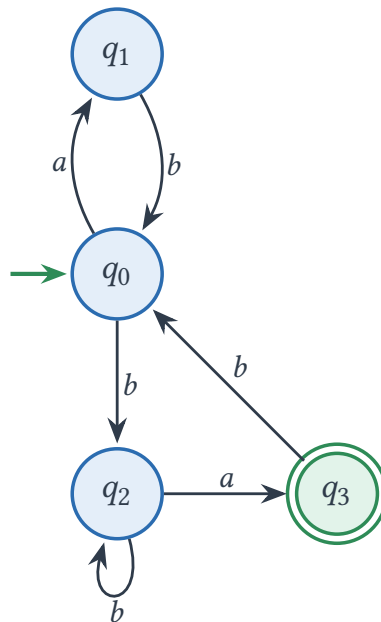


Figure 36.4 — Automate déterminisé (états q_0 à q_3 du tableau).

Question 16 ::: {adm kind=« corrige » title=« Corrigé »}

On sait que détermineriser préserve le langage, donc $L_{A_4} = L_{\widetilde{A_3}} = \widetilde{L_{A_3}} = \widetilde{\widetilde{L_{A_2}}} = \widetilde{\widetilde{\widetilde{L_{A_2}}}} = L_{A_2}$. :::

Question 17 ::: {adm kind=« corrige » title=« Corrigé »}

```
let rec supprimer l =
  match l with
  | [] -> []
  | t::q -> let q' = supprimer q in
             if List.mem t q' then q' else t::q'
```

:::

Question 18 ::: {adm kind=« corrige » title=« Corrigé »}

On effectue un parcours de la liste et à chaque élément, on appelle `List.mem` qui est également linéaire. On a donc un algorithme en $O(n^2)$ où n est le nombre d'éléments de la liste. :::

Question 19 ::: {adm kind=« corrige » title=« Corrigé »}

On utilise un tableau `pow` précalculé tel que `pow.(i)` vaut 2^i pour $0 \leq i \leq 20$.

```
let pow =
  let t = Array.make 21 1 in
  for i = 1 to 20 do
    t.(i) <- t.(i-1) * 2
  done;
  t

let est_dans q k =
  (k / pow.(q)) mod 2 = 1
```

:::

Question 20 ::: {adm kind=« corrige » title=« Corrigé »}

Deux versions possibles :

```
let numero l =
  let rec aux l =
    match l with
    | [] -> 0
    | i::q -> pow.(i) + aux q
  in
  aux (supprimer l)

let numero l =
  List.fold_left (+) 0
    (List.map (fun i -> pow.(i)) (supprimer l))
```

:::

Question 21 ::: {adm kind=« corrige » title=« Corrigé »}

```
let rec intersecte l k =
  match l with
  | [] -> false
  | t::q -> est_dans t k || intersecte q k
```

:::

Question 22 ::: {adm kind=« corrige » title=« Corrigé »}

```
let etat_suivant k t =
  let suivants c =
    List.map (fun (_,_,q) -> q)
      (List.filter (fun (q,a,_) -> c = a && est_dans q k) t) in
  let la = suivants 'a' in
```

```
let lb = suivants 'b' in
(numero la, numero lb)
```

:::

Question 23 ::: {adm kind=« corrige » title=« Corrigé »}

```
let rec cherche k l =
  match l with
  | [] -> -1 (* beurk *)
  | (k',v)::q when k' = k -> v
  | _::q -> cherche k q
```

:::

Question 24 ::: {adm kind=« corrige » title=« Corrigé »}

Note. Cette fonction est beaucoup trop délicate pour être donnée ainsi sans plus d'indications.

```
let determinise a =
  let corres = ref [] in
  let final = ref [] in

  let int_of_num k =
    let v = cherche k !corres in
    if v = -1
    then let v' = List.length !corres in
         corres := (k, v') :: !corres;
         if intersekte a.final k
         then final := v' :: !final;
         v'
    else v
  in

  let i = numero a.init in
  let init = int_of_num i in

  let rec parcours q =
    let nq = int_of_num q in
    let la, lb = etat_suitant q a.trans in
    let res = ref [] in
    if cherche la !corres = -1
    then res := !res @ parcours la;
    if cherche lb !corres = -1
    then res := !res @ parcours lb;
    (nq, 'a', int_of_num la)
    :: (nq, 'b', int_of_num lb) :: !res
  in
  let trans = parcours i in

  {
    nb = List.length !corres;
    init = [init];
    final = !final;
    trans = trans
  }
```

L'automate obtenu est complet : l'ensemble vide (numéro 0) est un état puits dès qu'il est accessible. Appliquée au miroir de l'automate de la question 13, elle produit ainsi les quatre états de la question 14 plus l'état puits. :::

Question 25 ::: {claudé kind=« modifie » raison=« la source attribuait le coût $O(n^2)$ à intersekte, qui est en $O(n)$; le coût dominant vient de supprimer (dans numero) et de cherche. »}

Corrigé

On effectue autant d'appels à parcours que d'états dans A_{det} , soit N appels. Dans chacun :

- `etat_suivant` parcourt la liste des transitions puis appelle `numero`, donc supprimer, sur des listes de longueur au plus $|T| : O(|T|^2)$ opérations ;
- les appels à `cherche` parcourent `corres`, de longueur au plus $N : O(N)$ (de même pour les concaténations `⊔`) ;
- `intersecte` coûte $O(n)$.

On est donc en $O(N(|T|^2 + N))$. Comme $N = O(2^n)$, c'est coûteux en pire cas.

...

Brzozowski

Remarque 36.1 Dans cette partie, il semble y avoir un problème apparent de notation vu que $\delta^* : Q' \times \Sigma \rightarrow Q'$. Cependant, $Q' = \mathcal{P}(Q)$ et c'est ainsi qu'il faut le comprendre. Par contre, il y a bien une erreur avec l'emploi de $\{I\}$ à la place de I .

Question 26 ::: {adm kind=« corrige » title=« Corrigé »}

Comme l'automate miroir est accessible, cela entraîne que l'automate lui-même est co-accessible, il existe ainsi un chemin de tout état vers un état final qu'il suffit de renverser pour en déduire w . Plus précisément :

Comme $q \in \delta^*(I, u)$ dans le déterminisé, cela signifie qu'il y a forcément un chemin $q \xrightarrow{\tilde{u}} i$ avec $i \in I$ dans $\tilde{A}$. Comme q est accessible depuis un état initial f de cet automate accessible, on sait qu'il existe w étiquette d'un chemin $f \xrightarrow{w} q$. On a donc $w\tilde{u} \in \tilde{L}$, donc $u\tilde{w} \in L$. :::

Question 27 ::: {adm kind=« corrige » title=« Corrigé »}

Soit $q \in \delta^*(I, u)$. Par la question précédente, il existe $w \in \Sigma^*$ tel que $f \xrightarrow{\tilde{w}} q$ dans $\tilde{A}$, où f est l'unique état initial de $\tilde{A}$, et $uw \in L$. Donc $w \in u^{-1}L = v^{-1}L$ et $vw \in L$. On en déduit deux chemins

$$f \xrightarrow{\tilde{w}} q' \xrightarrow{\tilde{v}} i' \in I \quad f \xrightarrow{\tilde{w}} q \xrightarrow{\tilde{u}} i \in I$$

dans $\tilde{A}$. Mais comme cet automate est déterministe, on a nécessairement $q = q'$. Or, comme $q \xrightarrow{\tilde{v}} i' \in I$, c'est qu'on a bien $q \in \delta^*(I, v)$.

Par symétrie, on a bien l'égalité. :::

Question 28 ::: {adm kind=« corrige » title=« Corrigé »}

C'est direct ici car $B = \tilde{\tilde{B}}$ est déterministe et accessible (c'est un déterminisé), donc $(\tilde{\tilde{B}})_{det}$ vérifie $*$ par la question précédente. :::

Remarque 36.2 Le sujet n'en dit pas plus, mais la propriété $*$ signifie que les états de l'automate sont bien minimaux car tous les états sont bien distingués les uns des autres : tous les états permettent de reconnaître depuis ceux-ci des mots différents.

Question 29 ::: {adm kind=« corrige » title=« Corrigé »}

```
let minimal a =
  determinise (transpose (determinise (transpose a)))
```

...

36.1.2 Expression rationnelle associée à un automate

Cette partie (questions 30 à 43) n'est pas corrigée.

36.1.3 Automate des dérivées d'Antimirov

Question 44 ::: {adm kind=« corrige » title=« Corrigé »}

$$\begin{aligned}
 \partial_a((ab|b)^*ba) &= \partial_a(ab|b) \cdot \{(ab|b)^*\} \cdot \{ba\} \cup \partial_a(ba) \\
 &= (\partial_a(a)\{b\} \cup \partial_a(b)) \cdot \{(ab|b)^*\} \cdot \{ba\} \cup \partial_a(b)\{a\} \\
 &= (\{\epsilon\}\{b\} \cup \emptyset) \{(ab|b)^*ba\} \cup \emptyset\{a\} \\
 &= \{b(ab|b)^*ba\}
 \end{aligned}$$

$$\begin{aligned}
 \partial_b((ab|b)^*ba) &= \partial_b(ab|b) \cdot \{(ab|b)^*\} \cdot \{ba\} \cup \partial_b(ba) \\
 &= (\partial_b(a)\{b\} \cup \partial_b(b)) \cdot \{(ab|b)^*\} \cdot \{ba\} \cup \partial_b(b)\{a\} \\
 &= (\emptyset\{b\} \cup \{\epsilon\}) \{(ab|b)^*ba\} \cup \{\epsilon\}\{a\} \\
 &= \{(ab|b)^*ba, a\}
 \end{aligned}$$

...

Question 45 ::: {adm kind=« corrige » title=« Corrigé »}

On construit la table des dérivations comme dans le cas de la table de l'automate des parties.

		∂_a	∂_b
$\rightarrow$	$(ab b)^*ba$	$b(ab b)^*ba$	$(ab b)^*ba, a$
	$b(ab b)^*ba$	$\emptyset$	$(ab b)^*ba$
	a	ϵ	$\emptyset$
$*$	ϵ	$\emptyset$	$\emptyset$

...

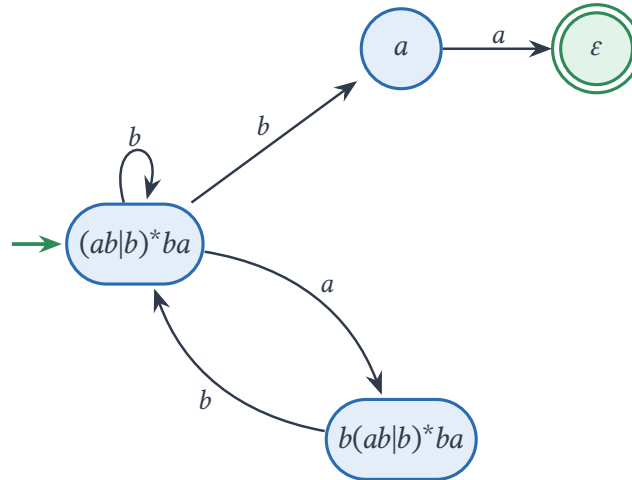


Figure 36.5 – Automate dont les états sont les dérivées de $(ab|b)^*ba$.

Question 46 ::: {adm kind=« corrige » title=« Corrigé »}

$$w \in (uv)^{-1}L \Leftrightarrow uvw \in L \Leftrightarrow vw \in u^{-1}L \Leftrightarrow w \in v^{-1}(u^{-1}L) = v^{-1}u^{-1}L. \quad \text{:::}$$

Question 47 ::: {adm kind=« corrige » title=« Corrigé »}

Par la question précédente et une récurrence sur $|w|$, on a $\mathcal{L}(\partial_w(E)) = w^{-1}\mathcal{L}(E)$. On conclut alors directement par passage à l'union :

$$\mathcal{L}(\partial_w(S)) = \mathcal{L}\left(\bigcup_{E \in S} \partial_w(E)\right) = \bigcup_{E \in S} \mathcal{L}(\partial_w(E)) = \bigcup_{E \in S} w^{-1} \mathcal{L}(E) = w^{-1} \bigcup_{E \in S} \mathcal{L}(E) = w^{-1} \mathcal{L}(S)$$

:::

Question 48 ::: {.claude kind=« modifie » raison=“la question 49 utilise les deux sens ($q \in \partial_w(E) \iff$ chemin de E à q d’étiquette w) ; la réciproque est ajoutée.”}

Corrigé

Si on a un chemin $E \xrightarrow{a_1} \dots \xrightarrow{a_n} q$ avec $w = a_1 \dots a_n$ alors, par construction de l’automate, on a $q \in \partial_w(E)$. Réciproquement, tout $q \in \partial_w(E)$ est l’extrémité d’un tel chemin.

Il s’agit d’une récurrence facile sur $|w|$: les transitions issues d’un état E' en lisant a mènent exactement aux éléments de $\partial_a(E')$, et $\partial_{wa}(E) = \bigcup_{E' \in \partial_w(E)} \partial_a(E')$.

:::

Question 49 ::: {.adm kind=« corrige » title=« Corrigé »}

Soit $w \in \mathcal{L}(E)$, on a donc $\epsilon \in w^{-1} \mathcal{L}(E) = \mathcal{L}(\partial_w(E))$ ce qui signifie qu’il existe un état acceptant dans $\partial_w(E)$, l’ensemble des états accessibles depuis E en lisant w . On vient de voir que l’automate d’Antimirov reconnaît alors w .

Réciproquement, si w est reconnu, cela signifie qu’il existe une expression rationnelle E' telle que $\epsilon \in \mathcal{L}(E')$ avec $E' \in \partial_w(E)$ donc $\epsilon \in \mathcal{L}(\partial_w(E)) = w^{-1} \mathcal{L}(E)$, soit encore $w \in \mathcal{L}(E)$.

Note. Il y a un raisonnement par équivalence possible ici. :::

Question 50 ::: {.adm kind=« corrige » title=« Corrigé »}

On raisonne par induction sur E . Ça se fait bien. :::

PROBLÈME 37

Agrégation 2023 - Épreuve 3b

37.1 Agrégation 2023 - Épreuve 3b

Sujet Corrigé partiel du sujet de l'épreuve 3b (fondements de l'informatique) de l'agrégation externe d'informatique, session 2023. L'énoncé n'est pas reproduit ici : il faut le consulter à côté de ce corrigé. Les questions sont numérotées comme dans l'énoncé ; certaines ne sont pas (encore) corrigées.

37.1.1 Partie I. Le cas d'un neurone à seuil

Fonctions booléennes linéaires à seuil

Question 1 ::: {adm kind=« corrige » title=« Corrigé »}

On remarque tout de suite que $x_1 + \dots + x_n$ est le nombre de x_i égaux à 1 car ils sont dans $\mathbb{B} = \{0, 1\}$.

- a. $x_1 + \dots + x_n \geq 1 \Leftrightarrow \exists i, x_i = 1 \Leftrightarrow OR(x_1, \dots, x_n) = 1$.
- b. $-x \geq 0 \Leftrightarrow x \leq 0 \Leftrightarrow x = 0 \Leftrightarrow NOT_1(x) = 1$.
- c. $x_1 + \dots + x_n \geq n/2$ si et seulement si il y a au moins $n/2$ x_i égaux à 1 et donc si et seulement si $MAJORITY(x_1, \dots, x_n) = 1$. :::

Question 2 ::: {claude kind=« modifie » raison=« $\beta + h$ » au lieu de « $\beta \geq h$ », et $XOR(\alpha, \beta)$ au lieu de $XOR(x, y)$. »}

Corrigé

Par l'absurde, s'il existe α, β, h tels que $\alpha x + \beta y \geq h \Leftrightarrow XOR(x, y) = 1$, alors, comme $XOR(0, 1) = XOR(1, 0) = 1$, on aura $\alpha \geq h$ et $\beta \geq h$, ainsi $\alpha + \beta \geq 2h$.

Ici, il faut raisonner selon le signe de h : si $h \geq 0$, alors $2h \geq h$, on en déduit $\alpha + \beta \geq h$ et donc $XOR(1, 1) = 1$.

Si $h < 0$, alors $0 \times \alpha + 0 \times \beta = 0 \geq h$ et donc $XOR(0, 0) = 1$.

Dans les deux cas, on aboutit à une contradiction.

:::

Question 3 ::: {claude kind=« modifie » raison=« $\alpha_1 + \alpha_2 < 2h < h$ » remplacé par « $< 2h \leq h$ » ($h \leq 0$), et conclusion rétablie (la fonction vaudrait 0 alors que PARITY vaut 1). »}

Corrigé

C'est ici une généralisation de l'argument précédent. On suppose $n \geq 2$.

Supposons, par l'absurde, qu'il existe $\alpha_1, \dots, \alpha_n, h$ tels que $\alpha_1 x_1 + \dots + \alpha_n x_n \geq h \Leftrightarrow PARITY_n(x_1, \dots, x_n) = 1$.

Comme $PARITY_n(0, \dots, 0) = 1$, on a $0 \geq h$.

Mais comme $PARITY_n(x_1, \dots, x_n) = 0$ dans le cas où seul $x_i = 1$, on a $\alpha_i < h$.

Ainsi $\alpha_1 + \alpha_2 < 2h \leq h$ car $h \leq 0$, donc la fonction à seuil vaut 0 en $(1, 1, 0, \dots, 0)$, alors que $PARITY_n(1, 1, 0, \dots, 0) = 1$.

Contradiction.

...

Cas de domaines bornés

Les questions 4 à 8 ne sont pas corrigées.

Apprentissage d'un neurone à seuil

Remarque 37.1 Dans cette partie on manipule des ensembles finis d'éléments de D^n , il y a donc un double indiciage.

On va donc noter $\underline{x}_i$ les éléments indicés de D^n pour bien marquer le double indiciage $\underline{x}_{ij}$.

Question 9 ::: {.claude kind=« modifie » raison=« notation \varphi rétablie dans la définition de X' . »}

Corrigé

Soit $X = \{(\underline{x}_1, y_1), \dots, (\underline{x}_N, y_N)\}$ un jeu d'exemples où les $\underline{x}_i \in D^n$.

On va poser $\varphi(\underline{x}_i) = (\underline{x}_{i_1}, \dots, \underline{x}_{i_n}, -1)$ et on considère $X' = \{(\varphi(x), y) \mid (x, y) \in X\}$.

Soit G solution du problème exact d'apprentissage à seuil nul sur X' .

On pose $F : D^n \rightarrow \mathbb{B}$ définie par $F(a_1, \dots, a_n) = G(a_1, \dots, a_n, -1)$.

Pour tout élément $(\underline{x}, y) \in X$, on a donc $F(\underline{x}) = G(\varphi(\underline{x})) = y$.

Par définition de G , il existe $(w_1, \dots, w_n, h)$ tels que $F(a_1, \dots, a_n) = G(a_1, \dots, a_n, -1) = 1 \Leftrightarrow w_1 a_1 + \dots + w_n a_n - h \geq 0 \Leftrightarrow w_1 a_1 + \dots + w_n a_n \geq h$.

On a donc résolu le problème exact d'apprentissage à seuil en dimension n .

...

Question 10 ::: {.claude kind=« modifie » raison=« suppression d'une affirmation « $F(x) = y$ » placée avant sa démonstration ; dans le i), « $\sum w_i (-a_i) \geq 0$ » remplacé par « > 0 » (équivalence exacte). »}

Corrigé

i) On va considérer ici une fonction linéaire à seuil nul H sur $[-1, 1]^n$ et établir quelques résultats. On considère que les poids sont $(w_1, \dots, w_n)$ et on considère $(a_1, \dots, a_n) \in [-1, 1]^n$.

Tout d'abord un résultat propre aux fonctions linéaires à seuil nul : si $\sum w_i a_i = 0$ on a forcément $H(a_1, \dots, a_n) = H(-a_1, \dots, -a_n) = 1$.

Si $\sum_i w_i a_i \neq 0$, on a :

$$H(a_1, \dots, a_n) = 0 \Leftrightarrow \sum_i w_i a_i < 0 \Leftrightarrow \sum_i w_i (-a_i) > 0 \Leftrightarrow H(-a_1, \dots, -a_n) = 1$$

Ainsi, dans tous les cas, si $H(a_1, \dots, a_n) = 0$ alors $H(-a_1, \dots, -a_n) = 1$.

ii) Soit $X = \{(\underline{x}_1, y_1), \dots, (\underline{x}_N, y_N)\}$ un jeu d'exemples où les $\underline{x}_i \in D^n$.

On reprend les notations de la question précédente, on pose

$$X'' = \{(\varphi(x), 0) \mid (x, 0) \in X\} \cup \{(-\varphi(x), 0) \mid (x, 1) \in X\}$$

Soit H solution du problème exact d'apprentissage à seuil nul sur X'' qui est en dimension $n + 1$ sur le domaine $[-1, 1]$.

On pose $F : D^n \rightarrow \mathbb{B}$ définie par $F(a_1, \dots, a_n) = H(a_1, \dots, a_n, -1)$.

Par définition de H , il existe $(w_1, \dots, w_n, h)$ tels que $F(a_1, \dots, a_n) = H(a_1, \dots, a_n, -1) = 1 \Leftrightarrow w_1 a_1 + \dots + w_n a_n - h \geq 0 \Leftrightarrow w_1 a_1 + \dots + w_n a_n \geq h$.

Reste à montrer que F satisfait les exemples X .

Soit $(\underline{x}, y) \in X$.

Si on a $y = 1$, alors $H(-\varphi(\underline{x})) = 0$ et donc $F(\underline{x}) = H(\varphi(\underline{x})) = 1 = y$ par le i).

Si on a $y = 0$, alors $F(\underline{x}) = H(\varphi(\underline{x})) = 0 = y$.

Dans tous les cas $F(\underline{x}) = y$ donc F est solution au problème d'apprentissage à seuil.

...

Question 11 ::: {adm kind=« corrige » title=« Corrigé »}

Note : il est inutile de passer la dimension en paramètre vu que c'est la longueur commune des vecteurs (on suppose qu'il y a au moins un exemple).

```
type vecteur = float array

let vectorize op v1 v2 =
  let n = Array.length v1 in
  Array.init n (fun i -> op v1.(i) v2.(i))

let difference = vectorize (-.)

let produit_scalaire v1 v2 =
  vectorize ( *. ) v1 v2
  |> Array.fold_left (+.) 0.

let apprentissage exemples =
  let n = Array.length exemples.(0) in
  let w = Array.make n 0. in
  let rec epoque () =
    let modifie = ref false in
    Array.iter
      (fun x -> if produit_scalaire w x >= 0.
        then begin
          modifie := true;
          Array.blit (difference w x) 0 w 0 n
        end) exemples;
    if !modifie
    then epoque ()
  in
  epoque ();
  Array.append w [|0.|]
```

Par exemple, sur les exemples $[| [-1.; -1.; 1.]; [1.; 0.; -1.]; [0.; 1.; -1.]; [0.; 0.; -1.] |]$, la fonction renvoie $[2.; 3.; 4.; 0.]$

Question 12 ::: {adm kind=« corrige » title=« Corrigé »}

Par condition d'arrêt (une époque sans modification), le vecteur $(w_1, \dots, w_n, 0)$ renvoyé vérifie $\sum w_i x_i < 0$ pour chaque exemple $(x, 0)$ donc si F est la fonction associée, on a $F(x) = 0$ et on a bien une solution au problème. ...

Question 13 ::: {adm kind=« corrige » title=« Corrigé »}

Soit $(v_1, \dots, v_n, 0)$ une représentation n -séparable de F .

On a donc :

- $F(x_1, \dots, x_n) = 0 \Leftrightarrow \sum_i v_i x_i < 0 \Leftrightarrow \sum_i v_i x_i \leq -n$;
- en particulier $\forall (x, 0) \in X, \sum_i v_i x_i \leq -n$.

On considère pour un candidat représentation $w : d(v, w) = ||v - w||^2 = \sum_i (v_i - w_i)^2$.

Soit w la valeur à une époque et $(x, 0) \in X$ un exemple pour lequel on a l'erreur $\sum_i w_i x_i \geq 0$.

On va poser $w' = w - x$.

On a

$$\begin{aligned} d(v, w') &= ||v - w + x||^2 = d(v, w) + ||x||^2 + 2 \sum_i (v_i - w_i)x_i \\ &= d(v, w) + ||x||^2 + 2 \sum_i v_i x_i - 2 \sum_i w_i x_i \\ &\leq d(v, w) + ||x||^2 - 2n - 2 \sum_i w_i x_i \end{aligned}$$

Or, $x \in D^n$ donc $||x||^2 \leq n$ et $\sum_i w_i x_i \geq 0$, donc $d(v, w) - d(v, w') \geq n > 0$. Ainsi la distance diminue d'au moins n à chaque modification ; comme elle est positive et vaut $||v||^2$ au départ, il y a au plus $||v||^2 / n$ modifications. Une époque sans modification arrête la boucle, donc la boucle s'arrête. :::

Question 14 :: {.claude kind=« modifie » raison=« la source utilisait $||x||^2 \leq 1$ (au lieu de $||x||^2 \leq n$, comme à la question 13) et $||v|| \leq n\mu$, confondait M (nombre d'erreurs) et N (nombre d'exemples) en fin de calcul ; la borne finale $O(n^2 \mu^2)$ est inchangée. »}

Corrigé

On note μ la masse de v pour ne pas avoir de confusion avec les poids w . Chaque $|v_i|$ est au plus μ , donc $||v||^2 \leq n\mu^2$.

On a $x \cdot v \leq -1$ car v est 1-séparable.

On reprend les calculs précédents. On a donc $w' \cdot v = w \cdot v - x \cdot v \geq w \cdot v + 1$. Comme au départ $w = \vec{0}$, après M erreurs, on a donc $w \cdot v \geq M$.

On a également $||w'||^2 = ||w - x||^2 = ||w||^2 - 2(w \cdot x) + ||x||^2$. Or, $||x||^2 \leq n$ et $-(w \cdot x) \leq 0$, ainsi $||w'||^2 \leq ||w||^2 + n$: la norme au carré des poids augmente au maximum de n à chaque erreur. Après M erreurs, on a donc $||w|| \leq \sqrt{nM}$.

Or, par l'inégalité de Cauchy-Schwarz, $w \cdot v \leq ||w|| ||v|| \leq \sqrt{nM} \sqrt{n\mu} = n\mu \sqrt{M}$. Donc $M \leq n\mu \sqrt{M}$ et ainsi $M \leq n^2 \mu^2$, donc $M = O(n^2 \mu^2)$.

Chaque époque, sauf la dernière, comporte au moins une erreur : il y a donc au plus $M + 1$ époques. Une époque coûte $O(nN)$ calculs pour les tests de produits scalaires sur les N exemples et $O(n)$ par modification de w . Au total, on est ainsi en $O(n^3 \mu^2 N)$.

:::

371.2 D'un neurone à seuil à un réseau de neurones

Question 15 :: {adm kind=« corrige » title=« Corrigé »}

Ce circuit est de profondeur 2.

- Couche 0 : x_1, x_2, x_3
- Couche 1 : NOT_1 de gauche, OR_2 et AND_2
- Couche 2 : NOT_1 en bas :::

Des circuits booléens aux réseaux de neurones à seuil

Question 16 ::: {adm kind=« corrige » title=« Corrigé »}

```
type sommet = int
type seuil = float
type poids = float
type reseau_adjacence = ((sommet * poids) list * seuil) array

let creer n = Array.make n ([], 0.)

let teste_arc r x y =
  fst r.(x) |> List.map fst |> List.mem y

let ajoute_arc r x y w =
  let l, t = r.(x) in
  r.(x) <- ((y,w) :: l, t)
```

:::

Question 17 ::: {claude kind=« modifie » raison=« « poids 0 ou 1 » remplacé par « poids 1 ou -1 » (la porte NOT_1 a le poids -1, question 1). »}

Corrigé

Chacune des portes logiques utilisées admet une représentation entière avec des poids égaux à 1 ou -1 (question 1), donc de poids unitaires. Il suffit alors de remplacer les portes par les neurones associés.

On préserve ainsi la structure du réseau, a fortiori sa profondeur et sa taille.

:::

Question 18 ::: {adm kind=« corrige » title=« Corrigé »}

Il suffit de rajouter un neurone *identité* pour rendre les connexions plus profondes. :::

PARTIE IX

Devoirs MPI 2022-2023

38	Devoir Interro 1 - Graphes	387
39	Devoir DS 4 - Algorithmes d'approximation, algorithmes probabilistes	389

Interro 1 - Graphes

38.1 Interro 1 — Graphes

MPI, année scolaire 2022-2023 — durée 45 min.

Question 1 1. Rappeler ce que calcule l'algorithme de Kruskal avec le maximum de précision. 2. Rappeler l'algorithme de Kruskal. 3. Prouver que l'algorithme de Kruskal est correct. 4. Détailler le calcul de la complexité de cet algorithme.

Question 2 1. Rappeler ce que calcule l'algorithme de Kosaraju avec le maximum de précision. 2. Rappeler l'algorithme de Kosaraju avec précision. 3. Prouver que l'algorithme de Kosaraju est correct. 4. Détailler le calcul de la complexité de Kosaraju.

Question 3 Rappeler ce qu'est un graphe bipartite.

Question 4 Rappeler ce qu'est un couplage dans un graphe, non nécessairement bipartite.

DS 4 - Algorithmes d'approximation, algorithmes probabilistes

39.1 DS 4 — Algorithmes d'approximation, algorithmes probabilistes

MPI, année scolaire 2022-2023.

 *Sujet d'après deux TD de l'Université de Montpellier : « Marche aléatoire pour la coloration de graphe » (B. Grenet, HMIN119m, TD 10, 2018-2019) et « Partition » (HAI503I, TD 5, 2022-2023). Les énoncés sont ici résumés ; le corrigé est celui de l'auteur.*

Ce devoir porte sur le chapitre Algorithmes d'approximation, algorithmes probabilistes.

39.1.1 3-Coloration probabiliste

Une k -coloration d'un graphe $G = (S, A)$ attribue à chaque sommet une couleur parmi k ; elle est valide si les deux extrémités de chaque arête ont des couleurs différentes. On cherche une 3-coloration valide d'un graphe 3-coloriable à n sommets et m arêtes.

Question 1 Les graphes G_1 et G_2 ci-dessous sont-ils 3-coloriables ? Le cas échéant, donner une 3-coloration valide.

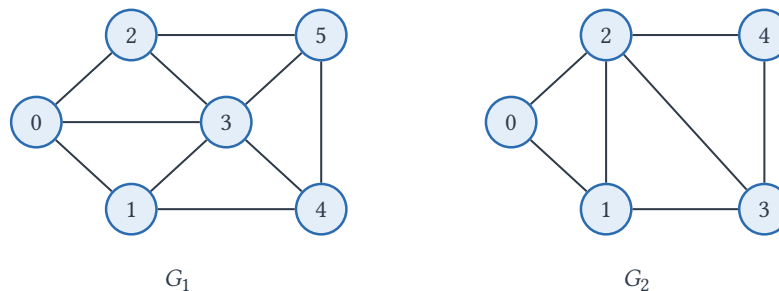


Figure 39.1 — Les graphes G_1 (à gauche) et G_2 (à droite).

Corrigé

1) G_1 n'est pas 3-coloriable : le sommet 3 est relié à tous les autres, donc seul 3 peut avoir sa couleur, et le cycle autour de 3 composé de 0, 2, 5, 4, 1 devrait être 2-coloriable, ce qui est impossible car sa longueur 5 est impaire.

2) G_2 est 3-coloriable avec la coloration suivante :

0	1	2	3	4
0	1	2	0	1

Question 2 On tire indépendamment et uniformément une couleur parmi trois pour chaque sommet.

i. Quelle est la probabilité qu'une arête fixée soit valide ?

- ii. Montrer que la probabilité d'échec (coloration non valide) est au plus $\frac{m}{3}$.
- iii. Pourquoi cet algorithme n'est-il pas satisfaisant ?

Corrigé

i) Une fois la première couleur choisie, il y a deux couleurs possibles sur trois rendant l'arête valide, donc la probabilité est de $\frac{2}{3}$.

Note pour ceux qui trouveraient ça peu rigoureux : vous pouvez énumérer les couples de couleurs ici si ça vous amuse, mais ça revient à ignorer la symétrie évidente de ces cas.

ii) On considère m arêtes et pour l'arête numéro i , l'événement V_i indiquant que la coloration de cette arête est valide. On a alors

$$P(\text{échec}) = P\left(\bigcup_{i=1}^m \overline{V_i}\right) \leq \sum_{i=1}^m P(\overline{V_i}) = \sum_{i=1}^m \frac{1}{3} = \frac{m}{3}$$

iii) Pour $m \geq 3$, cette majoration n'apporte aucune garantie.

On utilise maintenant une marche aléatoire : on part d'une coloration aléatoire et, tant qu'une arête n'est pas valide, on la rend valide en modifiant localement la coloration. La **distance** entre deux colorations est le nombre de sommets qui n'ont pas la même couleur dans les deux.

Question 3 Écrire le pseudo-code de cet algorithme.

Corrigé

- On génère aléatoirement une 3-coloration.
- Tant qu'il y a une arête non valide :
 - on choisit aléatoirement un des sommets de cette arête ;
 - on remplace sa couleur par une des deux autres couleurs en la choisissant aléatoirement.

On fixe une 3-coloration valide (la coloration cible) et on note d la distance entre la coloration courante et la cible, et Δd sa variation lors d'une étape.

Question 4 Montrer que $\Delta d \in \{-1, 0, 1\}$.

Corrigé

On ne change la couleur que d'un sommet, donc en faisant cela il y a trois possibilités :

- la nouvelle couleur est celle du sommet dans la coloration cible : on s'est rapproché et $\Delta d = -1$;
- la nouvelle couleur n'est pas celle du sommet dans la coloration cible alors que c'était le cas avant : on s'est éloigné et $\Delta d = 1$;
- la nouvelle couleur n'est pas celle du sommet dans la coloration cible et ce n'était déjà pas le cas avant : rien n'a changé sur la distance et $\Delta d = 0$.

Question 5 Lors d'une étape, soit aucun des deux sommets de l'arête traitée n'a sa couleur cible, soit l'un des deux l'a (pas les deux, car la cible est valide et l'arête ne l'est pas). Calculer les probabilités que Δd vaille -1 , 0 et 1 dans ces deux cas. Quel est le pire cas ?

Corrigé

On suppose ici que les couleurs sont dans $\{0, 1, 2\}$. On va noter $0 \sim 0$ la coloration invalide initiale de l'arête.

- Cas où la coloration cible est $1 \sim 2$ ($2 \sim 1$ symétrique) : on a quatre colorations possibles dans l'algorithme, équiprobables, et voici les cas de Δd correspondant :

... {,mtable cols=« 5 »} ... cellule col ... :: cellule 1 $\smile$ 0 ... :: cellule 2 $\smile$ 0 ... :: cellule 0 $\smile$ 1 ... :: cellule 0 $\smile$ 2 ... :: cellule Δd ... :: cellule -1 ... :: cellule 0 ... :: cellule 0 ... :: cellule -1 ... ::

Comme ces cas sont équiprobables, on en déduit : $P(\Delta d = -1) = P(\Delta d = 0) = \frac{1}{2}$ et $P(\Delta d = 1) = 0$.

- Cas où la coloration cible est $0 \smile 1$ (les autres cas sont symétriques) : on a quatre colorations possibles dans l'algorithme, équiprobables, et voici les cas de Δd correspondant :

... {,mtable cols=« 5 »} ... cellule col ... :: cellule 1 $\smile$ 0 ... :: cellule 2 $\smile$ 0 ... :: cellule 0 $\smile$ 1 ... :: cellule 0 $\smile$ 2 ... :: cellule Δd ... :: cellule 1 ... :: cellule 1 ... :: cellule -1 ... :: cellule 0 ... ::

Comme ces cas sont équiprobables, on en déduit : $P(\Delta d = -1) = P(\Delta d = 0) = \frac{1}{4}$ et $P(\Delta d = 1) = \frac{1}{2}$.

C'est le deuxième cas qui est le pire cas, car c'est celui où la probabilité de s'éloigner est strictement positive et où la probabilité de se rapprocher est la plus faible.

Notons ici que la question est facilitée par ses probabilités mais qu'il n'est pas toujours possible d'identifier un pire cas sans précisions sur ce qui est comparé.

On note $p(d)$ la probabilité que l'algorithme atteigne la coloration cible en partant d'une coloration à distance d , en supposant qu'on est toujours dans le pire cas.

Question 6 Montrer que $p(d-1) - 3p(d) + 2p(d+1) = 0$ pour $1 \leq d < n$, et donner les équations des cas limites $p(0)$ et $p(n)$.

On admet que la solution vérifie $p(d) \geq 2^{-d}$.

Corrigé

On va appliquer la formule des probabilités totales au pire cas en conditionnant par la valeur de Δd :

$$\begin{aligned} p(d) &= P(\Delta d = 0)p(d) + P(\Delta d = -1)p(d-1) + P(\Delta d = 1)p(d+1) \\ &= \frac{1}{4}p(d) + \frac{1}{4}p(d-1) + \frac{1}{2}p(d+1) \end{aligned}$$

qui équivaut à

$$3p(d) = p(d-1) + 2p(d+1)$$

On a $p(0) = 1$ car si on part d'une coloration à distance 0, c'est qu'elle est valide et l'algorithme la renvoie.

Si on part d'une coloration à distance n , aucun sommet n'a sa couleur cible : les arêtes invalides sont forcément dans le premier cas, et on a donc $p(n) = \frac{1}{2}p(n-1) + \frac{1}{2}p(n) \Leftrightarrow p(n) = p(n-1)$.

Question 7 Combien y a-t-il de colorations à distance d de la coloration cible ?

Corrigé

On adopte une approche constructive du dénombrement : on définit une manière univoque de construire les objets et on identifie pour chaque choix le nombre de possibilités.

Pour construire une coloration à distance d , il faut choisir les sommets qui changent, il y a $\binom{n}{d}$ choix possibles, et pour chacun de ces sommets il y a deux couleurs possibles pour différer de la cible.

On a donc $\binom{n}{d}2^d$ colorations.

Question 8 En déduire que la probabilité de succès de l'algorithme est au moins $\left(\frac{2}{3}\right)^n$, et calculer l'espérance du nombre d'exécutions nécessaires avant de trouver une 3-coloration valide.

Corrigé

On conditionne sur la distance d de la coloration initiale, qui est uniforme parmi les 3^n colorations :

$$\begin{aligned}
 P(\text{succès}) &\geq \sum_{k=0}^n P(d=k)p(k) \\
 &= \sum_{k=0}^n \frac{\binom{n}{k} 2^k}{3^n} p(k) \\
 &\geq \sum_{k=0}^n \frac{\binom{n}{k} 2^k}{3^n} 2^{-k} \\
 &= \frac{1}{3^n} \sum_{k=0}^n \binom{n}{k} \\
 &= \left(\frac{2}{3}\right)^n
 \end{aligned}$$

(la première inégalité car atteindre la cible n'est qu'une des façons de réussir).

On calcule ensuite l'espérance du nombre T d'exécutions de l'algorithme jusqu'à la première qui réussit, les exécutions étant indépendantes.

Il s'agit ici d'une loi géométrique : répéter une épreuve de Bernoulli tant qu'on n'a pas d'issue favorable.

La probabilité d'un succès au bout de k étapes signifie qu'on a eu $k-1$ échecs puis un succès, donc $P(T=k) = pq^{k-1}$ où $p = P(\text{succès})$ et $q = 1-p = P(\text{échec})$.

On a donc

$$\begin{aligned}
 E(T) &= \sum_{k=1}^{\infty} kP(T=k) \\
 &= \sum_{k=1}^{\infty} kpq^{k-1} \\
 &= p \frac{1}{(1-q)^2} \\
 &= p \frac{1}{p^2} \\
 &= \frac{1}{p} \\
 &\leq \left(\frac{3}{2}\right)^n
 \end{aligned}$$

Question 9 On ne peut pas savoir *a priori* si une exécution va s'arrêter. Proposer une ou plusieurs stratégies pour obtenir un algorithme de type Las Vegas.

Corrigé

La stratégie la plus simple est donc de lancer l'algorithme tant qu'on n'a pas de coloration valide, cependant, il faut décider d'un nombre d'itérations limite pour gérer les exécutions infinies sans toutefois perdre de vue la possibilité de converger vers une solution.

On peut, par exemple, considérer qu'on converge en moins de m itérations.

 **Remarque.** L'algorithme obtenu est bien de Las Vegas (une coloration renvoyée est toujours valide, puisqu'on la vérifie), à condition que le graphe soit 3-coloriable ; sinon il ne termine pas. En revanche, l'analyse précédente porte sur une marche non bornée : pour garantir l'espérance $\left(\frac{3}{2}\right)^n$ avec une limite

d'itérations, il faudrait minorer la probabilité d'atteindre la cible **en moins de** cette limite, ce que le sujet ne fait pas. Le choix de la limite m est donc une heuristique.

Un exemple de code réalisant cet algorithme est donné ici :

```
let g = [|
  [1;2];
  [0;2;3];
  [0;1;3;4];
  [1;2;4];
  [2;3]
|]

let coloration_aleatoire g =
  let n = Array.length g in
  Array.init n (fun i -> Random.int 3)

let coloration_valide g col =
  let n = Array.length g in
  let bvalide = ref true in
  for i = 0 to n - 1 do
    bvalide := !bvalide
    && List.for_all (fun j -> col.(i) <> col.(j)) g.(i)
  done;
  !bvalide

exception Trouve of (int * int)
let arete_invalide g col =
  let n = Array.length g in
  try
    for i = 0 to n-1 do
      List.iter (fun j ->
        if col.(i) = col.(j)
        then raise (Trouve (i,j)))
        g.(i)
      done;
    raise Not_found
  with Trouve (i,j) -> (i,j)

let arete_switch col (i,j) =
  let flip i =
    let k = Random.int 2 in
    col.(i) <- if col.(i) > k then k else 1 + k
  in flip (if Random.int 2 = 0 then i else j)

exception TropIterations

let coloration_marche g limite =
  let col = coloration_aleatoire g in
  let n = ref 0 in
  try
    while not (coloration_valide g col) do
      let i, j = arete_invalide g col in
      arete_switch col (i,j);
      incr n;
      if !n > limite
      then raise TropIterations
    done;
    Some col
  with TropIterations -> None

let rec coloration_marche_las_vegas g k =
  let m = Array.fold_left (+) 0
    (Array.map List.length g) / 2 in
  match coloration_marche g m with
  | Some col -> col, k
  | None -> coloration_marche_las_vegas g (k+1)
```

Le graphe g est le graphe G_2 de la question 1 ; `coloration_marche_las_vegas g 1` renvoie une coloration valide et le nombre d'essais effectués. Dans `arete_switch`, si k vaut 0 ou 1, la nouvelle couleur est la $(k + 1)$ -ième couleur différente de `col.(i)` : elle est donc uniforme parmi les deux autres couleurs.

39.1.2 2-Partition approchée

Étant donné un ensemble de n entiers positifs $A = \{a_0, \dots, a_{n-1}\}$, on cherche une partition $A = X \sqcup Y$ minimisant $\max(\Sigma_X, \Sigma_Y)$, où Σ_S désigne la somme des éléments de S . On considère l'algorithme glouton qui part de $X = Y = \emptyset$ et, pour i allant de 0 à $n - 1$, ajoute a_i à X si $\Sigma_X < \Sigma_Y$, et à Y sinon.

Question 1 a) Appliquer l'algorithme à $A = \{4, 2, 3, 2, 7\}$. La solution est-elle optimale ? b) Quelle est sa complexité ?

Corrigé

a) On présente le déroulé dans un tableau :

i	X	Y
	$\emptyset$	$\emptyset$
0	$\emptyset$	$\{4\}$
1	$\{2\}$	$\{4\}$
2	$\{2, 3\}$	$\{4\}$
3	$\{2, 3\}$	$\{4, 2\}$
4	$\{2, 3, 7\}$	$\{4, 2\}$

Et ainsi, le max vaut 12.

La solution optimale est ici atteinte par une division parfaite en deux : $\{4, 2, 3\}, \{2, 7\}$ où chaque somme vaut 9. Une telle division, si elle existe, est forcément optimale. L'algorithme n'est donc pas optimal ici.

b) L'algorithme est linéaire en $O(n)$ (en maintenant les sommes).

Dans la suite, (X^*, Y^*) est une solution optimale, $OPT = \max(\Sigma_{X^*}, \Sigma_{Y^*})$, et (X, Y) la solution du glouton, avec, quitte à échanger les noms, $\Sigma_X \geq \Sigma_Y$.

Question 2 Montrer que a) $a_i \leq OPT$ pour tout i ; b) $\Sigma_A \leq 2OPT$.

Corrigé

a) Si $a_i \in X^*$, alors $a_i \leq \Sigma_{X^*} \leq OPT$. Le cas Y^* est symétrique.

b) $\Sigma_A = \Sigma_{X^*} + \Sigma_{Y^*} \leq 2OPT$ car $a + b \leq 2 \max(a, b)$.

Question 3 Soit a_k le dernier élément ajouté à X . a) Montrer que $\Sigma_X - a_k \leq \frac{1}{2}(\Sigma_A - a_k) \leq OPT - \frac{1}{2}a_k$. b) En déduire que le glouton est une $\frac{3}{2}$ -approximation. c) Construire un exemple où la solution vaut exactement $\frac{3}{2}OPT$.

Corrigé

a) Notons X^0 et Y^0 les valeurs de X et de Y au moment où on a choisi a_k . On a donc $\Sigma_X = a_k + \Sigma_{X^0}$ et $\Sigma_{X^0} \leq \Sigma_{Y^0}$ par le test de l'algorithme (inégalité stricte si X est l'ensemble de la première branche, large sinon, quand on a échangé les noms).

On a $\Sigma_A - a_k = \Sigma_X + \Sigma_Y - a_k = \Sigma_{X^0} + \Sigma_Y \geq \Sigma_{X^0} + \Sigma_{Y^0} \geq 2\Sigma_{X^0}$.

Donc $\Sigma_X - a_k = \Sigma_{X^0} \leq \frac{1}{2}(\Sigma_A - a_k) \leq OPT - \frac{1}{2}a_k$ avec 2b.

b) $\Sigma_X \leq OPT + \frac{1}{2}a_k \leq \frac{3}{2}OPT$ avec le 2a.

c) $\{1, 1, 2\}$ va produire $X = \{1\}, Y = \{1, 2\}$ de valeur 3 alors que l'optimale $\{1, 1\}, \{2\}$ vaut 2. Donc exactement $3 = \frac{3}{2} \times 2$.

On trie maintenant A par ordre décroissant avant d'appliquer le glouton : $a_0 \geq a_1 \geq \dots \geq a_{n-1}$.

Question 4 a) Montrer que sur $\{10, 10, 9, 9, 2\}$, le glouton avec tri n'est pas optimal. b) Montrer que si X ne contient qu'un élément, la solution est optimale. c) Si X contient au moins deux éléments, montrer que le dernier élément a_k ajouté à X vérifie $a_k \leq \frac{2}{3}OPT$. d) En déduire que le glouton avec tri est une $\frac{4}{3}$ -approximation.

Corrigé

Une remarque sur l'énoncé ici, ce n'est pas très clair car il y a un choix de comparaison dans le test de l'algorithme qui défavorise X , donc la condition $\Sigma_X \geq \Sigma_Y$ a peu de chances d'être remplie dans des cas limites. On aurait pu mettre un $\leq$ pour simplifier.

a) On obtient $X = \{10, 9\}$ et $Y = \{10, 9, 2\}$ de valeur 21 alors que $\{10, 10\}, \{9, 9, 2\}$ est une division parfaite de valeur 20, donc optimale.

b) Si $X = \{a_i\}$, la valeur de la solution est $\max(\Sigma_X, \Sigma_Y) = \Sigma_X = a_i \leq OPT$ d'après 2a. La solution est donc optimale.

c) Soit $a_l \in X$ avec $l \neq k$ (existe par hypothèse). Comme a_k est le dernier élément ajouté à X , on a $l < k$ et donc $a_k \leq a_l$ grâce au tri. Avec 3a : $a_k \leq a_l \leq \Sigma_X - a_k \leq OPT - \frac{1}{2}a_k$.

C'est l'étape clé pour pouvoir poursuivre l'encadrement : $\frac{3}{2}a_k \leq OPT \Leftrightarrow a_k \leq \frac{2}{3}OPT$.

d) Ainsi, $\Sigma_X \leq OPT + \frac{1}{2}a_k \leq OPT + \frac{1}{3}OPT = \frac{4}{3}OPT$, et dans le cas b la solution est optimale.