

TP 14

September 6, 2022

1 Principe

Le but de ce TP est de pratiquer les parcours itératifs vus en classe. Pour cela, on va suivre assez fidèlement le poly du cours http://mpsi.classeprepa.net/poly/chap_structuresdonnees_graphes.html#sec:parcours (codes d'accès utilisateur *mpsi* et mot de passe *2137*).

L'objectif n'est pas de réécrire un parcours *ex nihilo* mais de savoir modifier les parcours vus en classe pour répondre à des questions.

Dans le TP13, on a vu le parcours en profondeur récursif, ici, on va d'abord revenir sur les piles (LIFO) et les files (FIFO), puis on va utiliser les deux parcours pour répondre à des problèmes.

2 Sac

On rappelle qu'un **sac** permet d'ajouter et de retirer des éléments en utilisant pour cela quatre opérations :

- `sac_vide()` crée un sac vide
- `sac_ajoute(sac, x)` ajoute l'élément `x` au sac
- `sac_retire(sac)` retire un élément du sac et le renvoie. **Produit une erreur si le sac est vide.**
- `sac_estvide(sac)` renvoie un booléen indiquant si le sac est vide

2.1 Pile

On a vu qu'une pile était un sac avec une stratégie **LIFO** (Last In First Out) l'élément retiré par un appel à `retire` est le dernier entré. En **Python**, on peut utiliser directement le type `list` qui fournit deux *méthodes* `.append(x)` et `.pop()`.

On définit ainsi les quatres fonctions suivantes :

```
[3]: def pile_vide():  
      return []  
  
      def pile_ajoute(p, x):  
          p.append(x)  
  
      def pile_retire(p):  
          x = p.pop()
```

```

    return x

def pile_estvide(p):
    return len(p) == 0

```

Il n'est pas rare d'utiliser une pile au sein d'un algorithme : on la crée au début, on la manipule et elle est détruite à la fin. Un exemple classique est la vérification du bon parenthésage. On considère des chaînes utilisant uniquement les caractères `[]()` et on définit le parenthésage usuel :

- `[([)][]]`, `((()())[])`, `()[]`, `()` sont bien parenthésées
- `[(])`, `[(())]` ne le sont pas.

Pour déterminer si une expression est bien parenthésée,

- on crée une pile
- on parcourt chaque caractère :
 - si le caractère courant est un `(` (resp. un `[`) on empile `)` (resp. `]`)
 - sinon c'est un `)` ou un `]`, on dépile si c'est possible (sinon on renvoie `False`) et on vérifie qu'on avait bien empilé le même caractère.
- A la fin, on renvoie un booléen indiquant si la pile est vide.

Question pourquoi doit-on s'assurer que la pile est vide en fin d'algorithme ?

Rappel on peut utiliser directement `for char in s` pour parcourir chaque caractère de la chaîne `s`.

```

[4]: def bien_parenthesee(s):
      # BEGIN SOLUTION
      p = pile_vide()
      for c in s:
          if c == '(':
              pile_ajoute(p, ')')
          elif c == '[':
              pile_ajoute(p, ']')
          else:
              if pile_estvide(p) or pile_retire(p) != c:
                  return False
      return pile_estvide(p)
      # END SOLUTION

```

```

[5]: from civ import *
      init()
      test(bien_parenthesee, '[([])]', True)
      test(bien_parenthesee, '((()())[]', True)
      test(bien_parenthesee, '()[]', True)
      test(bien_parenthesee, '()', True)
      test(bien_parenthesee, '[(])', False)
      test(bien_parenthesee, '[(())]', False)
      test(bien_parenthesee, '[([])]', False)
      valide()

```

```

OK!: bien_parenthesee('([[]])[]') vaut True
OK!: bien_parenthesee('(([]))[]') vaut True
OK!: bien_parenthesee('()[]') vaut True
OK!: bien_parenthesee('()') vaut True
OK!: bien_parenthesee('([[])') vaut False
OK!: bien_parenthesee('()') vaut False
OK!: bien_parenthesee('([[]]') vaut False
Tous les tests sont valides

```

2.2 File

On a vu qu'une file était un sac avec une stratégie **FIFO** (First In First Out) l'élément retiré par un appel à `retire` est le premier entré. En Python, on peut réaliser une file efficace à l'aide du type `deque` défini dans la bibliothèque `collections`.

En effet, ce type permet d'utiliser, en plus d'`append` et de `pop`, les deux méthodes `.appendleft(x)` et `.popleft(x)` qui vont effectuer ces deux opérations à *gauche*, c'est-à-dire au début de la structure. Ces quatre opérations s'effectuant en temps constant, on va pouvoir réaliser une file en ajoutant avec `append` et en retirant avec `popleft`.

On définit ainsi les quatres fonctions suivantes :

```

[10]: from collections import deque

def file_vide():
    return deque()

def file_ajoute(f, x):
    f.append(x)

def file_retire(f):
    x = f.popleft()
    return x

def file_estvide(f):
    return len(f) == 0

```

On dit qu'une chaîne est un palindrome quand on peut la lire dans les deux sens. C'est le cas du mot `kayak`.

Il ne s'agit pas de l'algorithme le plus approprié, mais pour vérifier qu'une chaîne est un palindrome, on peut créer une pile et une file, ajouter chaque caractère de la chaîne, puis dans un second temps tant qu'elles sont non vides, retirer un caractère de la pile et un caractère de la file et vérifier qu'ils sont égaux.

Question Pourquoi la pile et la file sont vides simultanément ?

Question Pour cet algorithme fonctionne ?

Attention il s'agit ici de s'entraîner à utiliser une file, donc on utilisera la méthode ci-dessus.

```
[11]: def palindrome(s):
      # BEGIN SOLUTION
      p = pile_vide()
      f = file_vide()
      for c in s:
          pile_ajoute(p, c)
          file_ajoute(f, c)
      while not file_estvide(f):
          if file_retire(f) != pile_retire(p):
              return False
      return True
      # END SOLUTION
```

```
[12]: from civ import *
      init()
      test(palindrome, 'kayak', True)
      test(palindrome, 'a', True)
      test(palindrome, '', True)
      test(palindrome, 'mpsi', False)
      valide()
```

OK!: palindrome('kayak') vaut True
 OK!: palindrome('a') vaut True
 OK!: palindrome('') vaut True
 OK!: palindrome('mpsi') vaut False
 Tous les tests sont valides

3 Parcours en largeur

On reprend ici directement le code du parcours en largeur présent dans le poly.

```
[13]: def parcours_largeur(ladj, source):
      a_traiter = file_vide()
      file_ajoute(a_traiter, source)
      marque = [ False ] * len(ladj)

      while not file_estvide(a_traiter):
          x = file_retire(a_traiter)
          if not marque[x]:
              marque[x] = True
              for y in ladj[x]:
                  file_ajoute(a_traiter, y)
```

On redonne également les définitions des différents graphes vus en classe.

```
[14]: def complet(n):
      return [ [i for i in range(n) if i != j] for j in range(n) ]
```

```

def cycle(n):
    return [ [ (i+1) % n, (i-1) % n ] for i in range(n) ]

def cycle_oriente(n):
    return [ [ (i+1) % n ] for i in range(n) ]

def nombre_vers_vecteur(N, n, d):
    # réalise l'opération inverse
    return [ (N // n**k) % n for k in range(d) ]

def dist(v1, v2):
    return sum( abs(v1[i]-v2[i]) for i in range(len(v1)) )

def grille(n, d):
    return [ [ k for k in range(n**d)
                if dist(nombre_vers_vecteur(k,n,d),
↪nombre_vers_vecteur(i,n,d)) == 1 ]
                for i in range(n**d) ]

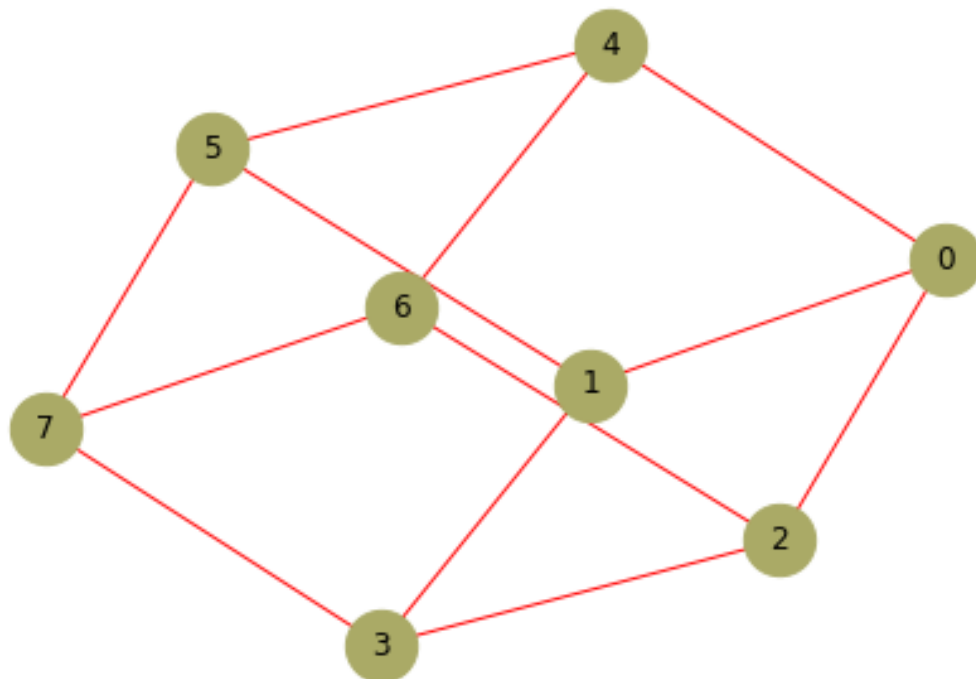
def hypercube(n):
    return grille(2, n)

```

```

[15]: from civ import affiche_graphe_l_no
affiche_graphe_l_no(hypercube(3))

```



L'appel suivant ne fait rien, modifiez le code de `parcours_largeur` pour afficher les sommets dans l'ordre de traitement pour l'hypercube et vérifiez ce qu'il en est.

```
[16]: parcours_largeur(hypercube(3), 0)
```

3.1 Parenté et chemin

On va modifier le parcours pour permettre de construire l'arbre de parenté du parcours. Rappel, le père d'un sommet étant celui qui a permis de le découvrir puis le traiter.

Contrairement au parcours récursif, on ne va pas traiter un sommet tout de suite mais une fois qu'il sera retiré.

On va donc construire un tableau `parent` où `parent[x] = y` si c'est l'arête `y -> x` qui a permis de traiter `x` et `None` sinon. Pour cela, on ne va pas ajouter des sommets dans `a_traiter` mais des couples (`sommet`, `pere`). Pour le sommet `source`, on considère que le père est `None`.

Rappel, il suffit d'écrire `x, y = c` où `c` est un couple pour obtenir ses composantes.

```
[30]: def parcours_largeur_parent(ladj, source):
    a_traiter = file_vide()
    file_ajoute(a_traiter, (source, None) ) # on ajoute un couple
    marque = [ False ] * len(ladj)
    parent = [ None ] * len(ladj)

    # BEGIN SOLUTION
    while not file_estvide(a_traiter):
        x, pere = file_retire(a_traiter)
        if not marque[x]:
            marque[x] = True
            parent[x] = pere
            for y in ladj[x]:
                file_ajoute(a_traiter, (y, x))
    # END SOLUTION

    return parent
```

```
[32]: from civ import *
init()
test(parcours_largeur_parent, hypercube(3), 0, [None, 0, 0, 1, 0, 1, 2, 3])
valide()
```

OK!: `parcours_largeur_parent([[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6], [1, 4, 7], [2, 4, 7], [3, 5, 6]], 0)` vaut `[None, 0, 0, 1, 0, 1, 2, 3]`

Tous les tests sont valides

N'hésitez pas à faire un dessin pour vous convaincre que le tableau est le bon.

On redonne le programme permettant de déterminer le chemin à partir d'un tableau de parenté :

```
[34]: def chemin(y, parent):
    l = [ y ]
    while parent[y] != None:
        y = parent[y]
        l.append(y)
    l.reverse()
    return l
```

```
[35]: chemin(7, [None, 0, 0, 1, 0, 1, 2, 3])
```

```
[35]: [0, 1, 3, 7]
```

On peut donc voir que le chemin dans H_3 entre 0 et $7 = 2^3 - 1$ est $0 \rightarrow 1 \rightarrow 3 \rightarrow 7$. On rappelle que deux sommets sont reliés quand leur écriture binaire diffère uniquement sur un bit, comme $0 = \overline{000}$ et $7 = \overline{111}$, un chemin est forcément de longueur au moins 3. Ici, on a donc obtenu un plus court chemin. On a admis en cours que c'était toujours le cas à l'issue d'un parcours en largeur.

3.2 Distances depuis une source

Si $G = (S, A)$ est un graphe, orienté ou non, on appelle **distance** entre les sommets x et y la longueur d'un plus petit chemin de x à y , on la note $dist(x, y)$. Quand un tel chemin n'existe pas, c'est-à-dire quand y n'est pas accessible depuis x , on note $dist(x, y) = \infty$.

Ainsi, $dist(s, s) = 0$.

Pour manipuler ∞ en Python, on pourra utiliser `float('inf')` qui est la représentation de $+\infty$ dans les flottants. Elle a l'avantage d'être compatible avec l'addition et avec la comparaison.

```
[46]: def distances_source(ladj, source):
    # renvoie le tableau dist tel que dist[x] = dist(source, x) si x est
    ↪ accessible et float('inf') sinon
    # BEGIN SOLUTION
    dist = [ float('inf') ] * len(ladj)
    dist[source] = 0
    parent = parcours_largeur_parent(ladj, source)
    for x in range(len(ladj)):
        if parent[x] != None:
            dist[x] = len(chemin(x, parent))-1
    return dist
    # END SOLUTION
```

```
[47]: from civ import *
init()
test(distances_source, hypercube(3), 0, [0, 1, 1, 2, 1, 2, 2, 3])
valide()
```

```
OK!: distances_source([[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6],
```

[1, 4, 7], [2, 4, 7], [3, 5, 6]], 0) vaut [0, 1, 1, 2, 1, 2, 2, 3]
Tous les tests sont valides

3.3 Distances entre toutes paires et diamètre

On appelle diamètre d'un graphe non vide la plus grande distance $< \infty$ entre deux sommets du graphe. On la note $\Delta(G)$ pour un graphe G .

Ainsi : $\Delta(G) = \max\{dist(x,y) \mid x,y \in S \text{ où } x \rightsquigarrow y\}$.

```
[48]: def distances(ladj):
    # renvoie la matrice dist où dist[x][y] = dist(x,y)
    # BEGIN SOLUTION
    dist = []
    for x in range(len(ladj)):
        dist.append( distances_source(ladj, x) )
    return dist
    # END SOLUTION

[52]: from civ import *
init()
test(distances, hypercube(3), [[0, 1, 1, 2, 1, 2, 2, 3], [1, 0, 2, 1, 2, 1, 3, ↵
↵2], [1, 2, 0, 1, 2, 3, 1, 2],
                                [2, 1, 1, 0, 3, 2, 2, 1], [1, 2, 2, 3, 0, 1, 1, ↵
↵2], [2, 1, 3, 2, 1, 0, 2, 1],
                                [2, 3, 1, 2, 1, 2, 0, 1], [3, 2, 2, 1, 2, 1, 1, ↵
↵0]])
valide()
```

OK!: distances([[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6], [1, 4, 7], [2, 4, 7], [3, 5, 6]]) vaut [[0, 1, 1, 2, 1, 2, 2, 3], [1, 0, 2, 1, 2, 1, 3, 2], [1, 2, 0, 1, 2, 3, 1, 2], [2, 1, 1, 0, 3, 2, 2, 1], [1, 2, 2, 3, 0, 1, 1, 2], [2, 1, 3, 2, 1, 0, 2, 1], [2, 3, 1, 2, 1, 2, 0, 1], [3, 2, 2, 1, 2, 1, 1, 0]]

Tous les tests sont valides

```
[53]: def diametre(ladj):
    # Renvoie le diametre du graphe
    # BEGIN SOLUTION
    m = 0
    for l in distances(ladj):
        for v in l:
            m = max(m, v)
    return m
    # END SOLUTION
```

```
[56]: from civ import *
init()
```

```
test(diametre, hypercube(3), 3)
test(diametre, hypercube(4), 4)
test(diametre, cycle(5), 2)
test(diametre, complet(5), 1)
valide()
```

OK!: diametre([[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6], [1, 4, 7], [2, 4, 7], [3, 5, 6]]) vaut 3

OK!: diametre([[1, 2, 4, 8], [0, 3, 5, 9], [0, 3, 6, 10], [1, 2, 7, 11], [0, 5, 6, 12], [1, 4, 7, 13], [2, 4, 7, 14], [3, 5, 6, 15], [0, 9, 10, 12], [1, 8, 11, 13], [2, 8, 11, 14], [3, 9, 10, 15], [4, 8, 13, 14], [5, 9, 12, 15], [6, 10, 12, 15], [7, 11, 13, 14]]) vaut 4

OK!: diametre([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]]) vaut 2

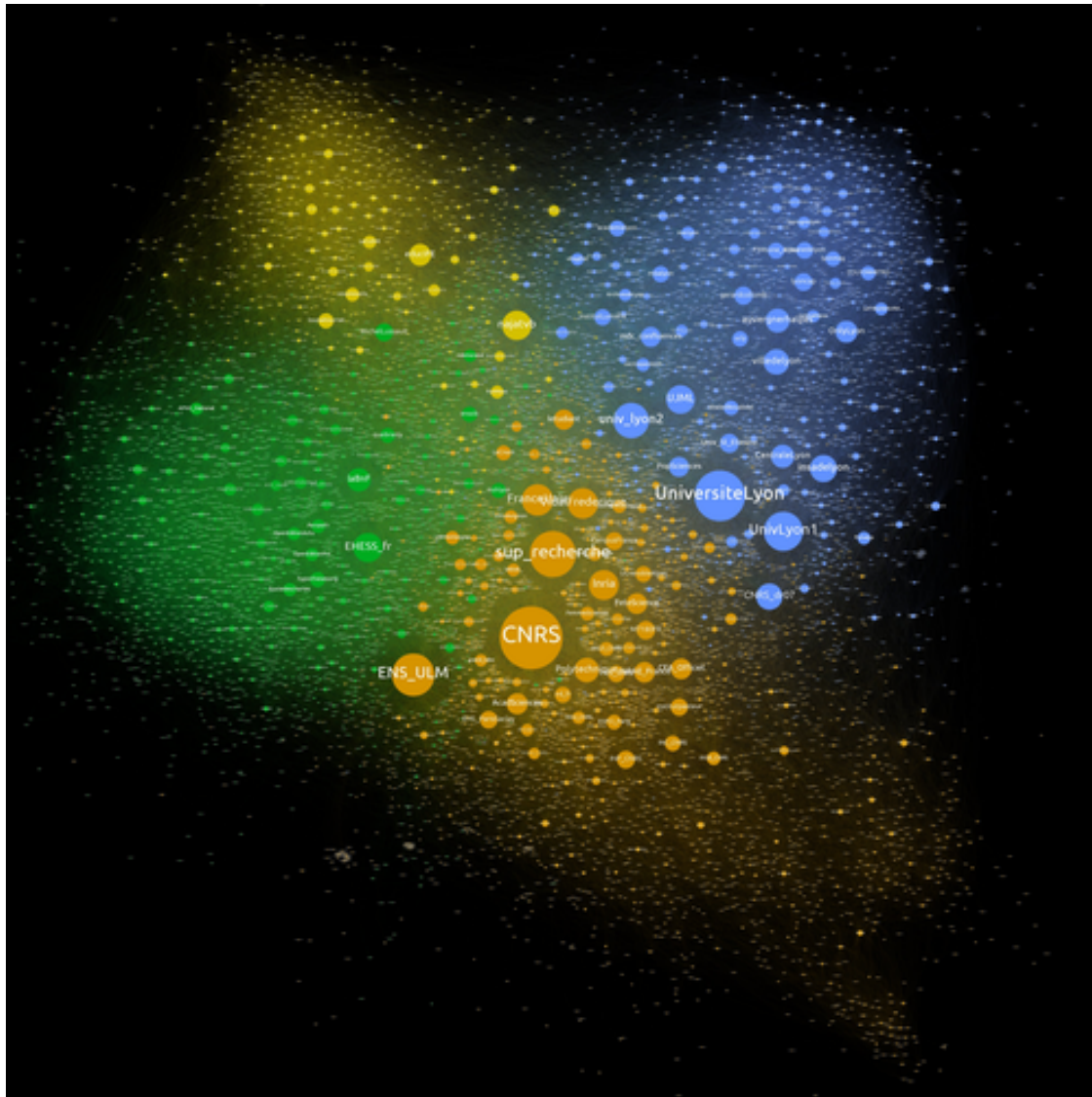
OK!: diametre([[1, 2, 3, 4], [0, 2, 3, 4], [0, 1, 3, 4], [0, 1, 2, 4], [0, 1, 2, 3]]) vaut 1

Tous les tests sont valides

4 Étude d'un graphe issu d'un réseau social

On va étudier le graphe orienté des followers du compte Twitter @ENSdeLyon.

Une représentation graphique de ce graphe est donné dans l'image suivante :



Les sommets sont les comptes donnés par leur identifiant (le @identifiant de Twitter) et une arête $x \rightarrow y$ indique que le compte x est abonné au compte y.

Ce graphe est assez conséquent : il comporte 8418 sommets et 305288 arêtes. Il nous permettra ainsi d'étudier en pratique la complexité des différents algorithmes étudiés.

```
[64]: from pickle import load
      comptes, ladj = load(open('ENSdeLyon.pickle', 'rb'))
```

Ici `comptes` est un tableau d'identifiants :

```
[82]: comptes[1632]
```

```
[82]: 'Patatoid_'
```

et `ladj` est une représentation en liste d'adjacences.

```
[83]: ladj[1632]
```

```
[83]: [8375,  
      8245,  
      4873,  
      7869,  
      6932,  
      5660,  
      4902,  
      2206,  
      7685,  
      3853,  
      2704,  
      6916,  
      3142,  
      5697,  
      4455,  
      6281,  
      8407,  
      3532,  
      3831,  
      8370]
```

Qu'on peut afficher sous un format lisible grace à `comptes` :

```
[84]: [ comptes[v] for v in ladj[1632] ]
```

```
[84]: ['ens_editions',  
      'FAENSL',  
      'nicolachiecchio',  
      'AE_ENS',  
      'scuolanormale',  
      'ECHARDE_ENSL',  
      'MiiniCampbell',  
      'GdrTurbulence',  
      'Asso_conferENS',  
      'AlumniENSdeLyon',  
      'Hydrayunet',  
      'SFP_officiel',  
      'Physique_ENS',  
      'IdexitLyon',  
      'lpensl',  
      'LP_ENS_',  
      'dlouapre',  
      'HommeViande',  
      'ASENSLyon',  
      'ENS_ULM']
```

Question Quel est le compte ayant le plus grand degré sortant ?

Question Quel est le compte ayant le plus grand degré entrant ?

Question Quel est le diamètre du graphe ?

Question Quels sont les chemins réalisant ce diamètre ?

Même question en considérant le symétrisé du graphe, c'est-à-dire celui où on a ajouté une arête $y \rightarrow x$ pour chaque arête $x \rightarrow y$.

[]: # C'est à vous !

[]:

[]:

[]:

[]: