

Dans ce TP, on va voir les bases de la manipulation des graphes. L'objectif est de réaliser des fonctions qu'on pourra réutiliser dans d'autres TPs. Vous pouvez faire le TP avec des fichiers `.ml` comme le TP18 ou directement sur Jupyter pour bénéficier des jeux de tests.

La partie bonus finale sera réalisable avec graphics, donc il faudra faire une transition à ce moment-là.

Pour simplifier, on ne va considérer dans un premier temps que des graphes $G = ([0; n - 1], A)$.

Matrice d'adjacence

Ainsi, on pourra considérer la matrice d'adjacence `mat : bool array array` telle que `mat.(i).(j)` soit égal à `true` si et seulement si $(i, j) \in A$.

On utilisera `make_matrix` puis remplir les arêtes une à une :

Entrée[1]: `type mat_adj = bool array array`

Sortie[1]:

Entrée[2]: `let mat_vide n : mat_adj =
 (* Renvoie la matrice d'adjacence du graphe vide sans arêtes *)
 (* BEGIN SOLUTION *)
 Array.make_matrix n n false
 (* END SOLUTION *)`

Sortie[2]:

Entrée[3]: `let mat_ajout_arete (m:mat_adj) x y : unit =
 (* Ajoute une arete x -> y *)
 (* BEGIN SOLUTION *)
 m.(x).(y) <- true
 (* END SOLUTION *)`

Sortie[3]:

Entrée[4]: `let m = mat_vide 3 in
mat_ajout_arete m 0 1;
mat_ajout_arete m 0 2;
mat_ajout_arete m 1 2;
assert (m = [| [|false; true; true|]; [|false; false; true|]; [|false
"OK"`

Sortie[4]:

Comme ce n'est pas forcément très pratique de définir des graphes ainsi, on va commencer par réaliser une fonction `init_matrix` qui permettra d'appliquer une fonction `int -> int -> 'a` pour définir une matrice à la manière de `Array.init`.

```
Entrée[5]: let init_matrix n p f =  
  (* BEGIN SOLUTION *)  
  Array.init n (fun i -> Array.init p (f i))  
  (* END SOLUTION *)
```

Sortie[5]:

```
Entrée[6]: assert(init_matrix 2 2 (fun i j -> i+j) = [| [|0; 1|]; [|1; 2|]|]);  
assert(init_matrix 2 2 (fun i j -> i = j) = [| [|true; false|]; [|fal  
assert(init_matrix 1 3 (fun i j -> j) = [| [|0; 1; 2|]|]);  
assert(init_matrix 3 1 (fun i j -> i) = [| [|0|]; [|1|]; [|2|]|]);  
"OK"
```

Sortie[6]:

La fonction suivante renvoie la matrice d'adjacence pour
 $A = \{(0, 1), (1, 2), \dots, (n-2, n-1), (n-1, 0)\}$.

```
Entrée[7]: let mat_cycle n =  
  (* BEGIN SOLUTION *)  
  init_matrix n n (fun i j -> (i+1) mod n = j)  
  (* END SOLUTION *)
```

Sortie[7]:

```
Entrée[8]: assert (mat_cycle 4 = [| [|false; true; false; false|]; [|false; fals  
assert (mat_cycle 2 = [| [|false; true|]; [|true; false|]|]);  
"OK"
```

Sortie[8]:

```
Entrée[9]: let mat_complet n =  
  (* Renvoie la matrice d'adjacence du graphe orienté complet  
  à n arêtes, c'est-a-dire celui ayant toutes les arêtes possibles  
  (* BEGIN SOLUTION *)  
  init_matrix n n (<>)  
  (* END SOLUTION *)
```

Sortie[9]:

```
Entrée[10]: assert (mat_complet 3 = [| [|false; true; true|]; [|true; false; true  
"OK"
```

Sortie[10]:

Pour les fonctions suivantes, on demande d'extraire les voisins entrants et sortants sous forme d'une liste. On pourra utiliser une fonction auxiliaire récursive ou une référence sur une liste pour simuler une pile.

```
Entrée[11]: let mat_voisins_entrants m x =
  (* BEGIN SOLUTION *)
  let v = ref [] in
  let n = Array.length m in
  for y = 0 to n-1 do
    if m.(y).(x) then v := y :: !v
  done;
  !v
  (* END SOLUTION *)
```

Sortie[11]:

```
Entrée[12]: let mat_voisins_sortants m x =
  (* BEGIN SOLUTION *)
  let v = ref [] in
  let n = Array.length m in
  for y = 0 to n-1 do
    if m.(x).(y) then v := y :: !v
  done;
  !v
  (* END SOLUTION *)
```

Sortie[12]:

```
Entrée[13]: let mat_degre_entrant m x =
  (* BEGIN SOLUTION *)
  List.length (mat_voisins_entrants m x)
  (* END SOLUTION *)
```

Sortie[13]:

```
Entrée[14]: let mat_degre_sortant m x =
  (* BEGIN SOLUTION *)
  List.length (mat_voisins_sortants m x)
  (* END SOLUTION *)
```

Sortie[14]:

```
Entrée[19]: let mat_ex = [| [| false; true; true; true; false |]; [| false; false; t
  [| true; false; false; false; true |]; [| false; false; false; false;
  [| false; false; true; true; false |] |] in
assert (List.map (mat_degre_entrant mat_ex) [0;1;2;3;4] = [1; 1; 3;
assert (List.map (mat_degre_sortant mat_ex) [0;1;2;3;4] = [3; 1; 2;
"OK"
```

Sortie[19]:

Graphes par matrice d'adjacence

Pour représenter un graphe, on va utiliser un tableau `'a array` pour les sommets et utiliser l'indice d'un élément pour accéder à la matrice d'adjacence.

Ainsi le graphe $A \rightarrow B \leftrightarrow C$ sera représenté par le tableau `[|"A";"B";"C"|]`

et la matrice $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. Notons qu'on aurait pu permuter les éléments de S et la

matrice associée sera également permutée : $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ et $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$.

On va devoir ainsi gérer la cohérence entre le tableau des sommets et la matrice d'adjacence le long de transformation.

On en déduit le type suivant. On va supposer **que chaque sommet est identifié uniquement par sa valeur**. Ainsi, dans l'exemple précédent "A" a un unique indice pour l'identifier. On remarque que le fait que les champs ne soient pas mutable signifie qu'il faudra créer un nouveau graphe pour changer l'ensemble de sommets.

Entrée[63]: **type** 'a graphem = { som : 'a array; mat : bool array array }

Sortie[63]:

Entrée[64]: **let** graphem_vide () : 'a graphem =
(renvoie un graphe sans sommet ni arêtes *)*
(BEGIN SOLUTION *)*
 { som = [||]; mat = [||] }
(END SOLUTION *)*

Sortie[64]:

Entrée[65]: **let** graphem_isoles (s:'a array) : 'a graphem =
(Renvoie un graphe sans arêtes de sommets s *)*
(BEGIN SOLUTION *)*
 { som = s; mat = mat_vide (Array.length s) }
(END SOLUTION *)*

Sortie[65]:

Entrée[66]: **let** graphem_dim (g:'a graphem) : int * int =
(Renvoie le couple (n,p) où n est le nombre de sommets*
*et p le nombre d'arêtes *)*
(BEGIN SOLUTION *)*
let n = Array.length g.som **in**
let p = ref 0 **in**
for i = 0 **to** n-1 **do**
 for j = 0 **to** n-1 **do**
if g.mat.(i).(j) **then** incr p
done
done;
 n, !p
(END SOLUTION *)*

Sortie[66]:

Entrée[67]: **let** g_ex = { som = ['A';'B';'C';'D';'E']; mat = [|||false; true; true; false; false; true; true; false|||] }

Sortie[67]:

Entrée[68]: **assert** (graphem_dim g_ex = (5, 9));
 "OK"

Sortie[68]:

La représentation n'étant pas très lisible, on aimerait pouvoir afficher les graphes sous la forme suivante :

Sommets: A B C D E

Arêtes: A -> B, A -> C, A -> D, B -> C, C -> A, C -> E, D -> E, E -> C, E -> D,

On rappelle qu'avec Jupyter il faut un `printf` finissant par `\n%!` pour s'assurer de l'affichage.

```
Entrée[103]: let graphem_print g =  
  (* Affiche un char graphem sous le format précédent *)  
  (* BEGIN SOLUTION *)  
  let n, _ = graphem_dim g in  
  Printf.printf "Sommets: ";  
  for i = 0 to n-1 do  
    Printf.printf "%c " g.som.(i)  
  done;  
  Printf.printf "\nArêtes: ";  
  for i = 0 to n-1 do  
    for j = 0 to n-1 do  
      if g.mat.(i).(j)  
      then Printf.printf "%c -> %c, " g.som.(i) g.som.(j)  
    done  
  done;  
  Printf.printf "\n%! "  
  (* END SOLUTION *)
```

Sortie[103]:

```
Entrée[104]: graphem_print g_ex
```

Sommets: A B C D E

Arêtes: A -> B, A -> C, A -> D, B -> C, C -> A, C -> E, D -> E, E -> C, E -> D,

Sortie[104]:

A partir de maintenant, on va travailler sur les sommets puis en déduire leurs indices, on supposera que toutes les fonctions sont appelées avec des sommets valides.

Entrée[69]: **exception** Trouve **of** int

```
let graphem_indice (g:'a graphem) (x:'a) : int =  
  (* Renvoie l'indice de x dans le graphe *)  
  (* BEGIN SOLUTION *)  
  try  
    for i = 0 to Array.length g.som - 1 do  
      if g.som.(i) = x  
        then raise (Trouve i)  
    done;  
    raise Not_found  
  with Trouve i -> i  
  (* END SOLUTION *)
```

Sortie[69]:

Sortie[69]:

Entrée[70]: **assert** (graphem_indice g_ex 'C' = 2);
"OK"

Sortie[70]:

Entrée[71]: **let** graphem_test_arete (g:'a graphem) (x:'a) (y:'a) : bool =
 (* Test si le graphe contient une arete x -> y *)
 (* BEGIN SOLUTION *)
 let i = graphem_indice g x **in**
 let j = graphem_indice g y **in**
 g.mat.(i).(j)
 (* END SOLUTION *)

Sortie[71]:

Entrée[74]: **let** graphem_ajoute_arete (g:'a graphem) (x:'a) (y:'a) : unit =
 (* Ajoute une arete x -> y *)
 (* BEGIN SOLUTION *)
 let i = graphem_indice g x **in**
 let j = graphem_indice g y **in**
 g.mat.(i).(j) <- true
 (* END SOLUTION *)

Sortie[74]:

Entrée[75]: **let** graphem_suppr_arete (g:'a graphem) (x:'a) (y:'a) : unit =
 (* Ajoute une arete x -> y *)
 (* BEGIN SOLUTION *)
 let i = graphem_indice g x **in**
 let j = graphem_indice g y **in**
 g.mat.(i).(j) <- false
 (* END SOLUTION *)

Sortie[75]:

```
Entrée[76]: let g = graphem_isoles [|'A';'B';'C'|] in  
graphem_ajoute_arete g 'A' 'B';  
graphem_ajoute_arete g 'A' 'C';  
graphem_ajoute_arete g 'C' 'A';  
graphem_ajoute_arete g 'B' 'C';  
assert (g.mat = [| [|false; true; true|]; [|false; false; true|]; [|t
```

Sortie[76]:

Pour les fonctions suivantes, on pourra utiliser toutes les fonctions disponibles dans le module Array : <https://ocaml.org/api/Array.html> (<https://ocaml.org/api/Array.html>) .
Notamment Array.append .

```
Entrée[77]: let graphem_ajoute_sommet (g:'a graphem) (x:'a) : 'a graphem =  
  (* Ajoute un nouveau sommet au graphe a la fin du tableau,  
    il faudra changer la matrice *)  
  (* BEGIN SOLUTION *)  
  let n, _ = graphem_dim g in  
  let som = Array.append g.som [|x|] in  
  let mat = Array.append g.mat [| Array.make (n+1) false |] in  
  for i = 0 to n-1 do  
    mat.(i) <- Array.append mat.(i) [|false|]  
  done;  
  { som = som; mat = mat }  
  (* END SOLUTION *)
```

Sortie[77]:

```
Entrée[78]: let g = graphem_ajoute_sommet g_ex 'F' in  
assert (g = {som = [|'A'; 'B'; 'C'; 'D'; 'E'; 'F'|];  
  mat =  
    [| [|false; true; true; true; false; false|];  
      [|false; false; true; false; false; false|];  
      [|true; false; false; false; true; false|];  
      [|false; false; false; false; true; false|];  
      [|false; false; true; true; false; false|];  
      [|false; false; false; false; false; false|] |] } );  
"OK"
```

Sortie[78]:

Entrée[80]: **let** graphem_suppr_sommet (g:'a graphem) (x:'a) : 'a graphem =
(Supprime le sommet x. Attention cela va surement casser les c*
Une technique pourra consister à partir d'un graphe isole et a
*depuis g *)*
(BEGIN SOLUTION *)*
let n, _ = graphem_dim g **in**
let i = graphem_indice g x **in**
let som = Array.append (Array.sub g.som 0 i) (Array.sub g.som (i
let g' = graphem_isoles som **in**
for k = 0 **to** n-1 **do**
 for l = 0 **to** n-1 **do**
 if i <> k && i <> l && g.mat.(k).(l)
 then graphem_ajoute_arete g' g.som.(k) g.som.(l)
 done
done;
g'
(END SOLUTION *)*

Sortie[80]:

Entrée[82]: **assert** (graphem_suppr_sommet g_ex 'C' = {som = [|'A'; 'B'; 'D'; 'E'|]
mat = [| [|false; true; true; false|]; [|false; false; false; false|]
[|false; false; false; true|]; [|false; false; true; false|] |] }
"OK")

Sortie[82]:

Entrée[91]: **let** graphem_sous_graphe (g:'a graphem) (s:'a array) : 'a graphem =
(Renvoie le sous graphe induit par s sous-ensemble de g.som*
on pourra commencer par calculer le complémentaire de s dans g.s
*puis supprimer tous ces sommets *)*
(BEGIN SOLUTION *)*
let g' = ref g **in**
let n, _ = graphem_dim g **in**
for i = 0 **to** n - 1 **do**
 if not (Array.mem g.som.(i) s)
 then g' := graphem_suppr_sommet !g' g.som.(i)
done;
!g'
(END SOLUTION *)*

Sortie[91]:

Entrée[92]: **assert** (graphem_sous_graphe g_ex [|'A'; 'D'|] = {som = [|'A'; 'D'|];
"OK")

Sortie[92]:

Listes d'adjacence

On va maintenant représenter les arêtes par un `ladj : int list array` où
`ladj.(i)` donne la liste des indices des **voisins sortants** du sommet d'indice *i*.

On va reprendre les fonctions précédentes.

Entrée[108]: **type** liste_adj = int list array

Sortie[108]:

Entrée[109]: **let** ladj_vide n =
 (** BEGIN SOLUTION **)
 Array.make n []
 (** END SOLUTION **)

Sortie[109]:

Entrée[110]: **let** ladj_ajout_arete l x y =
 (** BEGIN SOLUTION **)
 l.(x) <- y :: l.(x)
 (** END SOLUTION **)

Sortie[110]:

Entrée[111]: **let** ladj_cycle n =
 (** BEGIN SOLUTION **)
 Array.init n (**fun** i -> [(i+1) mod n])
 (** END SOLUTION **)

Sortie[111]:

Entrée[113]: **assert** (ladj_cycle 4 = [| [1]; [2]; [3]; [0] |]);
"OK"

Sortie[113]:

Entrée[119]: **let** ladj_complet n =
 (** BEGIN SOLUTION **)
 let rec aux i j =
 if i < 0
 then []
 else if i <> j **then** i :: aux (i-1) j **else** aux (i-1) j
 in
 Array.init n (**fun** i -> aux (n-1) i)
 (** END SOLUTION **)

Sortie[119]:

Entrée[118]: **assert** (ladj_complet 4 = [| [3; 2; 1]; [3; 2; 0]; [3; 1; 0]; [2; 1; 0] |]);
"OK"

Sortie[118]:

Entrée[120]: **let** ladj_voisins_sortants l x = l.(x)

Sortie[120]:

Entrée[124]: **let** ladj_voisins_entrants l x =
 (* C'est plus compliqué ! On pourra utiliser List.filter et List
 (* BEGIN SOLUTION *)
 let n = Array.length l **in**
 List.filter (**fun** i -> List.mem x l.(i)) (List.init n (**fun** i -> i
 (* END SOLUTION *)

Sortie[124]:

Entrée[131]: **let** l = [| [1;2;3]; [2]; [0;4]; [4]; [2;3] |] **in**
 let l2 = ladj_voisins_entrants l 4 **in**
 assert (l2 = [2;3] || l2 = [3;2]);
 "OK"

Sortie[131]:

Entrée[132]: **let** ladj_degre_sortant l x = List.length l.(x)

Sortie[132]:

Entrée[133]: **let** ladj_degre_entrant l x =
 (* BEGIN SOLUTION *)
 List.length (ladj_voisins_entrants l x)
 (* END SOLUTION *)

Sortie[133]:

Entrée[137]: **let** ladj_ex = [| [3; 2; 1]; [2]; [4; 0]; [4]; [3; 2] |] **in**
 assert (List.map (ladj_degre_entrant ladj_ex) [0;1;2;3;4] = [1; 1; 3
 assert (List.map (ladj_degre_sortant ladj_ex) [0;1;2;3;4] = [3; 1; 2
 "OK"

Sortie[137]:

Entrée[135]: **let** mat_to_ladj m =
 (* Convertit une matrice d'adjacence en liste d'adjacence *)
 (* BEGIN SOLUTION *)
 let n = Array.length m **in**
 let l = ladj_vide n **in**
 for i = 0 **to** n-1 **do**
 for j = 0 **to** n-1 **do**
 if m.(i).(j)
 then l.(i) <- j :: l.(i)
 done
 done;
 l
 (* END SOLUTION *)

Sortie[135]:

Entrée[139]: **let** mat_ex = [| [| false; true; true; true; false |]; [| false; false; t
 [| true; false; false; false; true |]; [| false; false; false; false;
 [| false; false; true; true; false |] |] **in**
 assert (mat_to_ladj mat_ex = [| [3; 2; 1]; [2]; [4; 0]; [4]; [3; 2] |])
 "OK"

Sortie[139]:

```
Entrée[140]: let ladj_to_mat l =
  (* Convertit une liste d'adjacence en matrice d'adjacence *)
  (* BEGIN SOLUTION *)
  let n = Array.length l in
  let m = mat_vide n in
  for i = 0 to n-1 do
    List.iter (fun j -> m.(i).(j) <- true) l.(i)
  done;
  m
  (* END SOLUTION *)
```

Sortie[140]:

```
Entrée[142]: let mat_ex = [| [| false; true; true; true; false |]; [| false; false; t
  [| true; false; false; false; true |]; [| false; false; false; false;
  [| false; false; true; true; false |] |] in
assert (ladj_to_mat [| [3; 2; 1]; [2]; [4; 0]; [4]; [3; 2] |] = mat_ex
"OK"
```

Sortie[142]:

Graphe par liste d'adjacence

```
Entrée[93]: type 'a graphel = { soml : 'a array; ladj : int list array }
```

Sortie[93]:

```
Entrée[195]: (* Le g_ex sera alors représenté par : *)
let g_ex_l = { soml = [| 'A'; 'B'; 'C'; 'D'; 'E' |]; ladj = [| [1;2;3]; [2
```

Sortie[195]:

On va maintenant reprendre **toutes** les fonctions précédentes. **Attention** sans faire de conversions brutales depuis et vers les matrices d'adjacences.

```
Entrée[106]: let graphel_vide () =
  (* BEGIN SOLUTION *)
  { soml = [| |]; ladj = [| |] }
  (* END SOLUTION *)
```

Sortie[106]:

```
Entrée[143]: let graphel_isoles s =
  (* BEGIN SOLUTION *)
  { soml = s; ladj = ladj_vide (Array.length s) }
  (* END SOLUTION *)
```

Sortie[143]:

```
Entrée[147]: let graphel_dim g =  
    (* BEGIN SOLUTION *)  
    let n = Array.length g.soml in  
    let p = Array.fold_left (+) 0 (Array.map List.length g.ladj) in  
    n, p  
    (* END SOLUTION *)
```

Sortie[147]:

```
Entrée[189]: assert (graphel_dim g_ex_1 = (5, 9) );  
"OK"
```

Sortie[189]:

```
Entrée[152]: let graphel_print g =  
    (* BEGIN SOLUTION *)  
    let n, _ = graphel_dim g in  
    Printf.printf "Sommets: ";  
    for i = 0 to n-1 do  
        Printf.printf "%c " g.soml.(i)  
    done;  
    Printf.printf "\nAretes: ";  
    for i = 0 to n-1 do  
        List.iter (fun j -> Printf.printf "%c -> %c, " g.soml.(i) g.  
    done;  
    Printf.printf "\n%!"  
    (* END SOLUTION *)
```

Sortie[152]:

```
Entrée[190]: graphel_print g_ex_1  
  
Sommets: A B C D E  
Aretes: A -> B, A -> C, A -> D, B -> C, C -> A, C -> E, D -> E, E -  
> C, E -> D,
```

Sortie[190]:

```
Entrée[154]: exception Trouve of int  
  
let graphel_indice (g:'a graphel) (x:'a) : int =  
    (* Renvoie l'indice de x dans le graphe *)  
    (* BEGIN SOLUTION *)  
    try  
        for i = 0 to Array.length g.soml - 1 do  
            if g.soml.(i) = x  
            then raise (Trouve i)  
        done;  
        raise Not_found  
    with Trouve i -> i  
    (* END SOLUTION *)
```

Sortie[154]:

Sortie[154]:

Entrée[191]: **assert** (graphel_indice g_ex_l 'C' = 2);
"OK"

Sortie[191]:

Entrée[158]: **let** graphel_test_arete (g:'a graphel) (x:'a) (y:'a) : bool =
 (* Test si le graphe contient une arete x -> y *)
 (* BEGIN SOLUTION *)
 let i = graphel_indice g x **in**
 let j = graphel_indice g y **in**
 List.mem j g.ladj.(i)
 (* END SOLUTION *)

Sortie[158]:

Entrée[165]: **let** graphel_ajoute_arete (g:'a graphel) (x:'a) (y:'a) : unit =
 (* Ajoute une arete x -> y . ATTENTION ici il faut vérifier qu'e *)
 (* BEGIN SOLUTION *)
 let i = graphel_indice g x **in**
 let j = graphel_indice g y **in**
 if not (List.mem j g.ladj.(i)) **then** g.ladj.(i) <- j :: g.ladj.(i)
 (* END SOLUTION *)

Sortie[165]:

Entrée[166]: **let** graphel_suppr_arete (g:'a graphel) (x:'a) (y:'a) : unit =
 (* Supprime une arete x -> y *)
 (* BEGIN SOLUTION *)
 let i = graphel_indice g x **in**
 let j = graphel_indice g y **in**
 g.ladj.(i) <- List.filter ((<>) j) g.ladj.(i)
 (* END SOLUTION *)

Sortie[166]:

Entrée[171]: **let** g = graphel_isoles [|'A';'B';'C'|] **in**
graphel_ajoute_arete g 'A' 'B';
graphel_ajoute_arete g 'A' 'C';
graphel_ajoute_arete g 'C' 'A';
graphel_ajoute_arete g 'B' 'C';
assert(g = {soml = [|'A'; 'B'; 'C'|]; ladj = [| [2; 1]; [2]; [0] |] })
graphel_suppr_arete g 'C' 'A';
assert(g = {soml = [|'A'; 'B'; 'C'|]; ladj = [| [2; 1]; [2]; [] |] });
"OK"

Sortie[171]:

Entrée[193]: **let** graphel_ajoute_sommet (g:'a graphel) (x:'a) : 'a graphel =
 (* Ajoute un nouveau sommet au graphe a la fin du tableau *)
 (* BEGIN SOLUTION *)
 let soml = Array.append g.soml [|x|] **in**
 let ladj = Array.append g.ladj [| [] |] **in**
 { soml = soml; ladj = ladj }
 (* END SOLUTION *)

Sortie[193]:

Entrée[197]: `graphel_ajoute_sommet g_ex_l 'F' (* doit valoir {soml = [/'A'; 'B';
ladj = [/[1; 2; 3]; [2]; [0; 4]; [4]; [2; 3]; []/]} *)`

Sortie[197]:

Entrée[179]: `let rec suppr_adj l i =
 (* supprime le sommet i de la liste d'adjacence en décalant
 à j-1 les indices j > i *)
 (* BEGIN SOLUTION *)
 match l with
 | [] -> []
 | t::q when t = i -> suppr_adj q i
 | t::q when t < i -> t :: suppr_adj q i
 | t::q -> (t-1) :: suppr_adj q i
 (* END SOLUTION *)`

Sortie[179]:

Entrée[182]: `assert (suppr_adj [1;2;3;4;5] 3 = [1;2;3;4]);
assert (suppr_adj [1;3;5] 3 = [1;4]);
"OK"`

Sortie[182]:

Entrée[186]: `let graphel_suppr_sommet (g:'a graphel) (x:'a) : 'a graphel =
 (* Supprime le sommet x *)
 (* BEGIN SOLUTION *)
 let n = Array.length g.soml in
 let i = graphel_indice g x in
 let soml = Array.append (Array.sub g.soml 0 i) (Array.sub g.soml
 let ladj = Array.append (Array.sub g.ladj 0 i) (Array.sub g.ladj
 for j = 0 to n-2 do
 ladj.(j) <- suppr_adj ladj.(j) i
 done;
 { soml = soml; ladj = ladj }
 (* END SOLUTION *)`

Sortie[186]:

Entrée[200]: `graphel_suppr_sommet g_ex_l 'C'
(* doit valoir :
{soml = [/'A'; 'B'; 'D'; 'E']/]; ladj = [/[1; 2]; []; [3]; [2]/]}
*)`

Sortie[200]: