Graphes : modélisation et parcours

OPTION INFORMATIQUE - TP n° 3.3 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ modéliser un graphe par liste d'adjacence
- ☞ modéliser un graphe par matrice d'adjacence
- ☞ passer d'une modélisation à une autre
- ☞ parcourir un graphe en largeur et en profondeur
- ☞ implémenter l'algorithme de Dijkstra

A Modélisation d'un graphe

Dans ce qui suit on peut considérer le graphe :

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

A1. Sous quelle forme le graphe g est-il donné ?

> **Solution :** Le graphe est donné sous la forme d'une liste d'adjacence.

A2. Dessiner le graphe g. Comment peut-on qualifier ce graphe ?

> **Solution :** C'est un graphe non orienté et non pondéré : un graphe simple, connexe et acyclique (donc un arbre ;-).

A3. On dispose d'un graphe sous la forme d'une liste d'adjacence. Écrire une fonction de signature `liste_vers_matrice : int list array -> bool array array` qui transforme cette représentation en une matrice d'adjacence. Les valeurs de la matrice sont des booléens : il y a une arête ou il n'y en a pas.

> **Solution :**
>
> ```
> let liste_vers_matrice g =
>   let n = Array.length g in
>   let matrice = Array.make_matrix n n false in
>   let rec lire_ligne voisins row =
>         match voisins with
>             | [] -> ()
> ```

```
            | u::t -> matrice.(row).(u) <- true; lire_ligne t row in
  for v = 0 to n - 1 do
    lire_ligne g.(v) v;
  done;
  matrice ;;
```

A4. On dispose d'un graphe sous la forme d'une matrice d'adjacence. Écrire une fonction de signature `matrice_vers_liste : bool array array -> int list array` qui transforme cette représentation en une liste d'adjacence.

Solution :

```
let matrice_vers_liste matrice =
  let n = Array.length matrice in
  let g = Array.make n [] in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      if matrice.(i).(j) then g.(i) <- j::g.(i)
    done;
  done;
  g ;;
```

A5. On dispose d'un graphe orienté sous la forme d'une liste d'adjacence. Écrire une fonction de signature `desorienter_liste : int list array -> unit` qui transforme ce graphe en un graphe non orienté. Cette fonction travaille en place.

Solution :

```
let desorienter_liste g =
  let n = Array.length g in
  let rec symetriser voisins v =
        match voisins with
            | [] -> ()
            | u::t -> if not (List.mem v g.(u)) then g.(u) <- v::g.(u);
                symetriser t v in
  for v = 0 to n - 1 do
        symetriser g.(v) v
  done;;
```

A6. On dispose d'un graphe orienté sous la forme d'une matrice d'adjacence. Écrire une fonction de signature `desorienter_matrice : bool array array -> unit` qui transforme ce graphe en un graphe non orienté. Cette fonction travaille en place.

Solution :

```
let desorienter_matrice m =
  let n = Array.length m in
```

```
    for i = 0 to n - 1 do
        for j = i + 1 to n - 1 do
            if m.(i).(j) && (not m.(j).(i)) then m.(j).(i) <- true;
            if m.(j).(i) && (not m.(i).(j)) then m.(i).(j) <- true;
        done ;
done;;
```

B Parcourir un graphe

Le parcours d'un graphe (cf. algorithme 1) est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. Les sommets passent dans une **pile** de type Last In First Out.
3. L'algorithme de **Dijkstra** passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'est un parcours en largeur qui utilise une **file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance. La plus petite distance en tête donc.

Dans cette section, on suppose qu'on manipule un graphe sous la forme d'une liste d'adjacence.

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file d'attente vide                       ▷ F comme file
3 :     D ← ∅                                             ▷ D ensemble des sommets découverts
4 :     P ← une liste vide                                ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)                  ▷ Pour la preuve de la correction, on précise que d[s] = 0
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                 ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ D alors                            ▷ x n'a pas encore été découvert
12 :                AJOUTER(D, x)
13 :                ENFILER(F, x)     ▷ Pour la preuve de la correction, on ajoute ici d[x] = d[u] + 1
14 :    renvoyer P                    ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

B1. Démontrer la terminaison du parcours en largeur.

Solution : La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle

$$v = |F| + |\overline{D}| \tag{1}$$

c'est-à-dire la somme des éléments présents dans la file et du nombre de sommets non découverts.

Démonstration. À l'entrée de la boucle, si n est l'ordre du graphe, on a $|F| + |\overline{D}| = 1 + n - 1 = n$. Puis, à chaque itération, on retire un élément de la file et on ajoute ses p voisins en même temps qu'on marque les p voisins comme découverts. L'évolution du variant au cours d'une itération s'écrit :

$$v = (|F|_d - 1 + p) + (|\overline{D}|_d - p) = |F|_d + |\overline{D}|_d - 1 \tag{2}$$

où l'on note $|F|_d$ et $|\overline{D}|_d$ les valeurs au début de l'itération de $|F|$ et $|\overline{D}|$.

À chaque tour de boucle, ce variant v décroit donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\overline{D}| = 0$, on a donc $|F| = 0$ et $|\overline{D}| = 0$. La file est nécessairement vide et tous les sommets ont été découverts. L'algorithme se termine puisque la condition de sortie est atteinte. ■

B2. Démontrer la correction du parcours en largeur.

Solution :

Démonstration. On montre dans un premier temps que le parcours en largeur à partir du sommet de départ visite tous les sommets par l'absurde.

Imaginons qu'il existe un sommet x accessible depuis le sommet de départ s et non visité par l'algorithme du parcours en largeur. Supposons que x est le premier sommet accessible depuis s et non visité par l'algorithme. Si ce sommet est accessible depuis s, c'est qu'il existe un chemin de s à x dans le graphe. x possède donc un sommet adjacent parent sur ce chemin que l'on note p. Comme x est le premier sommet non visité, alors p a nécessairement été découvert précédemment par l'algorithme. Mais comme le parcours procède en largeur par sommets adjacents, si p a été découvert, alors x a été découvert également. Ce qui est absurde. Il n'existe donc pas de sommet x non visité par l'algorithme. **Donc tous les sommets sont visités.**

Puis il faut montrer qu'on parcourt les sommets selon l'ordre des voisins d'abord, c'est-à-dire dans l'ordre de la distance au sommet de départ. Le graphe n'est pas pondéré, mais on peut considérer que tous les poids du graphe valent 1 ou bien qu'on compte les sauts (franchissement des arêtes) pour atteindre un sommet. Il suffit d'ajouter la notion de distance comme précisé en commentaire sur l'algorithme 1.

On procède en utlisant l'invariant de boucle `while` : $\mathcal{I}$: **« Pour chaque sommet v qui se trouve dans la file, sa distance d[v] qu sommet de départ s est correcte et tous les sommets déjà découverts ont été découverts dans l'ordre croissant de distance depuis s. »**

Initialisation à l'entrée de la boucle `while`, le seul sommet dans la file est le sommet de départ qui est situé à une distance nulle de lui-même. La distance est correcte. Comme aucun autre sommet n'a encore été découvert, l'ordre est correct également. Donc $\mathcal{I}$ est vérifié à l'entrée de la boucle.

Conservation Supposons que $\mathcal{I}$ soit vérifié à l'entrée de la boucle. D'après cette hypothèse, la distance du sommet v défilé est correcte car v a été enfilé et marqué comme découvert à la précédente itération. À la fin de l'itération, on a marqué comme découvert tous les voisins de v qui se trouve à une distance de 1 de v. Ce sont les voisins les plus proches car ils sont directement connectés à v. Donc les sommets découverts le sont bien dans l'ordre croissant de distance. $\mathcal{I}$ est conservé par les instructions du corps de la boucle.

Conclusion Comme l'invariant est vérifié à l'entrée de la boucle et qu'il est conservé par les instructions de la boucle, $\mathcal{I}$ est vérifié à la fin de la boucle. Par conséquent, le parcours en largeur est correct : tous les sommets ont été découverts dans l'ordre croissant des distances.

Le parcours en largeur est donc correct. ■

B3. Écrire une fonction de signature `parcours_largeur : int list array -> int -> int list` qui renvoie un parcours en largeur du graphe à partir du sommet de départ passé en paramètre. En mélangeant les approches impératives et récursives, on peut proposer des versions à la fois fluides et efficaces. On s'appuie sur une structure file d'attente programmée comme suit :

```
type 'a file = {
  mutable tete : 'a list;
  mutable queue : 'a list;
}

let filevide () = { tete = []; queue = [] };;

let est_vide file = (file.tete = [] && file.queue = []);;

let enfiler x file = file.queue <-  x :: file.queue;;

let defiler file=
  let rec rev rlst lst =
    match lst with
    | [] -> rlst
    | h :: t -> rev (h::rlst) t in
  match file.tete with
    | [] ->  ( match rev [] file.queue with
                | [] -> failwith "Erreur : file vide !"
                | e :: t -> file.tete <- t; file.queue <- []; e )
    | e :: t -> file.tete <- t; e;;

let tete file=
  match file.tete with
  | [] ->
      (match List.rev file.queue with
       | [] -> None
       | x :: _ -> Some x)
  | x :: _ -> Some x;;
```

Solution :

```
let parcours_largeur g s0 =
    let n = Array.length g in
    let decouverts = Array.make n false in
    let file = filevide () in
    let parcours = ref [s0] in
    enfiler s0 file;
    decouverts.(s0) <- true;
    while not (est_vide file) do
      let s = defiler file in
      let m = List.length g.(s) in
      let voisins = ref g.(s) in
      for k = 0 to m - 1 do
        let v = List.hd !voisins in
        if  not decouverts.(v) then (decouverts.(v) <- true; parcours := v
            :: !parcours; enfiler v file);
        voisins := List.tl !voisins;
      done;
    done;
    List.rev !parcours;;

let parcours_largeur g s0 =
  let n = Array.length g in
  let decouverts = Array.make n false in
  let file = filevide () in
  let parcours = ref [s0] in
  enfiler s0 file;
  decouverts.(s0) <- true;
  while not (est_vide file) do
    let s = defiler file in
    let rec parcours_voisins voisins =
      match voisins with
       | [] -> ()
       | v :: t -> if not decouverts.(v) then (decouverts.(v) <- true;
           parcours := v :: !parcours; enfiler v file);  parcours_voisins t
           in parcours_voisins g.(s)
  done;
  List.rev !parcours;;

let parcours_largeur g s0 =
  let rec explorer file decouverts =
      match file with
            | [] -> []
            | v::t when List.mem v decouverts -> explorer t decouverts
            | v::t -> v::(explorer (t @ g.(v)) (v::decouverts))
  in explorer [s0] [] ;;

let parcours_largeur g s0 =
  let n = Array.length g in
  let decouverts = Array.make n false in
  let rec explorer file =
      match file with
            | [] -> []
            | v::t when decouverts.(v) -> explorer t
            | v::t -> decouverts.(v) <- true; v::(explorer (t @ g.(v)))
  in explorer [s0];;
```

La première version est totalement itérative. En utilisant l'opérateur ajouter en tête `::` suivi d'un retournement de la liste plutôt que la concaténation de liste `@`, les opérations sont optimisées.
La seconde version évite la boucle `for` en utilisant la récursivité. La complextié est la même, la syntaxe plus fluide.
La troisième version possède une syntaxe très fluide. Cependant l'utilisation de `List.mem` n'est pas optimal, car en $O(n)$ si n est la longueur de la liste.
Dans la quatrième version, on remédie à ce problème en mélangeant approche impérative et récursive.

B4. Que vaut la complexité de `parcourir_largeur` ?

Solution : La complexité vaut $\mathcal{O}(|S|+|A|) = \mathcal{O}(n+m)$. Les opérations dans la boucle sont effectuées en un temps constant. On défile `file` jusqu'à ce que la file soit vide. Or, on enfile une seule fois un sommet (n) et ses voisins (m) s'ils n'ont pas été découverts. D'où le résultat.

B5. Écrire une fonction de signature `parcours_profondeur : int list array -> int -> int list` qui parcours en profondeur un graphe et qui renvoie la liste des sommets parcourus.

Solution :

```
let parcours_profondeur g s0 =
  let n = Array.length g in
  let decouverts = Array.make n false; in
  let rec explorer parcours s =
    if decouverts.(s)
    then parcours
    else (let suivants = g.(s) in decouverts.(s) <- true; List.fold_left
        explorer (parcours @ [s]) suivants)
  in explorer [] s0;;
```

C Plus courts chemins : algorithme de Dijkstra

C1. Démontrer la terminaison de l'algorithme de Dijkstra.

Solution : Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n-1$ éléments, si $n \in \mathbb{N}^\star$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro.

C2. Démontrer la correction de l'algorithme de Dijkstra.

Solution : On note :

- δ_u la distance la plus courte du sommet s_0 au sommet u.

Algorithme 2 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (S, A, w), s_0)        ▷ Trouver les plus courts chemins à partir de s_0 ∈ V
2 :     Δ ← s_0                                  ▷ Δ est le dictionnaire des sommets dont on connaît la distance à s_0
3 :     Π ← ∅                                    ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
4 :     d ← ∅                                    ▷ l'ensemble des distances au sommet s_0
5 :     ∀s ∈ V, d[s] ← w(s_0, s)                 ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
6 :     tant que Δ̄ n'est pas vide répéter        ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)   ▷ Choix glouton !
8 :         Δ = Δ ∪ {u}                          ▷ On prend la plus courte distance à s_0 dans Δ̄
9 :         pour x ∈ V_G(u) répéter              ▷ pour tous les voisins de u
10 :            si d[x] > d[u] + w(u, x) alors
11 :                d[x] ← d[u] + w(u, x)        ▷ Mises à jour des distances des voisins
12 :                Π[x] ← u                     ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d, Π
```

- $d[u]$ la distance trouvée par l'algorithme entre le sommet s_0 le sommet u.

On souhaite démontrer la correction en montrant que $\mathcal{I} : \forall x \in \Delta, d[x] = \delta_x$ est un invariant de boucle.

1. Avant la boucle : Δ ne contient que le sommet s_0. Or, $d[s_0] = 0$ et $\delta_{s_0} = 0$. Donc, l'invariant est vérifié à l'entrée de la boucle.
2. Pour une itération quelconque, on suppose que l'invariant $\mathcal{I}$ est vérifié à l'entrée de la boucle. Un sommet u est sélectionné dans $\bar{\Delta}$. Pour ce sommet u, qui n'appartient pas encore à Δ, on souhaite montrer qu'à la fin de l'itération $d[u] = \delta_u$.

   On procède par l'absurde en supposant que $d[u] \neq \delta_u$ et qu'il existe un plus court chemin P de s_0 à u tel que la longueur de ce chemin $\lambda(P)$ soit **strictement** plus petite que $d[u]$:

   $$\lambda(P) < d[u]$$

   Ce chemin P démarre d'un sommet de Δ et le quitte au bout d'un certain temps pour atteindre u. Soit (x, y) la première arête quittant Δ de ce chemin P : $x \in \Delta$ et $y \in \bar{\Delta}$. Soit P_x le chemin de s_0 à x. Ce chemin est un plus court chemin, par hypothèse d'induction et $d[x] = \delta_x$. **Comme y est un sommet adjacent à x qui a été découvert précédemment, la distance à s_0 a été mise à jour par l'algorithme**. Donc, la distance à y est telle que :

   $$d[y] \leqslant \delta_x + w(x, y) \tag{3}$$

   Or, la distance de s_0 à y est une partie du chemin P. Donc :

   $$d[y] \leqslant \delta_x + w(x, y) \leqslant \lambda(P) < d[u] \tag{4}$$

   De plus, comme on a sélectionné u dans $\bar{\Delta}$ tel que la distance soit minimale au sommet de départ, et comme $y \in \bar{\Delta}$, on a également :

   $$d[u] \leqslant d[y] \tag{5}$$

En combinant ces équations, on aboutit à la contradiction suivante : $d[u] < d[u]$, ce qui est absurde. Un chemin tel que P n'existe donc pas et $d[u] = \delta_u$. L'invariant $\mathcal{I}$ est donc vérifié à la fin de l'itération.

3. Comme l'invariant $\mathcal{I}$ est vérifié à l'entrée de la boucle et qu'il n'est pas modifié par les instructions de la boucle, on en déduit qu'il est vrai à la fin de la boucle. Comme, à la fin de la boucle, Δ contient tous les sommets du graphe, on a : $\forall x \in S, d[x] = \delta_x$, c'est-à-dire l'algorithme de Dijsktra est correct.

C3. Quelle est la complexité de l'algorithme de Dikjstra ?

Solution :

La complexité de l'algorithme de Dijsktra dépend de l'ordre n du graphe considéré et de sa taille m. La boucle *tant que* effectue exactement $n-1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à $2m$ (si le graphe n'est pas orienté), donc proportionnelle à son nombre d'arêtes.

En notant le coût du transfert c_t, le coût de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n,m) = (n-1)c_t + mc_d \tag{6}$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n \log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n-1)O(n \log n) + mO(1) = O(n^2 \log n)$.

Si on utilise une file de priorités, on peut atteindre une bien meilleure complexité : $C(n) = (n-1)\log n + m \log n = O(()n+m)\log n)$, car le coût de transfert est en $\log n$ (retrirer le min, défiler) et le coût de mise à jour en $\log n$ (tamiser le tas).

C4. Exécuter à la main l'algorithme de Dijsktra sur le graphe orienté suivant en complétant à la fois le tableau des distances et le tableau des parents qui permet de reconstruire le chemin a posteriori. Le tableau parent à la case `i` contient le sommet précédent sur le chemin.

```
let g = [| [(1,7);(2,1)] ;
           [(0,7);(2,5);(3,4);(4,2);(5,1)] ;
           [(0,1);(1,5);(4,2);(5,7)];
           [(1,4);(4,5)];
           [(1,2);(2,2);(3,5);(5,3)];
           [(1,2);(2,7);(4,3)] |] ;;
```

Solution :

```
val d : int array = [|0; 5; 1; 8; 3; 6|]
  val p : int array = [|0; 4; 0; 1; 2; 1|]
```

Algorithme 3 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(g, s_0)                          ▷ Trouver les plus courts chemins à partir de s_0
2 :     n ← nombre de sommets de G
3 :     fp ← une file de priorités                      ▷ contient les tuples (distance, sommet)
4 :     ENFILER(fp,(0,s_0))                             ▷ initialisation de la file de priorités
5 :     d ← un tableau de taille n                      ▷ recense les distances de chaque sommet à s_0
6 :     ∀s ∈ S, d[s] ← w(s_0, s)                        ▷ w(s_0, s) = +∞, 0 si s = s_0
7 :     parents ← un tableau de taille n                ▷ Recense le père d'un sommet sur le chemin le plus court
8 :     parents[s_0] ← s_0                              ▷ Le sommet de départ est son propre père sur le chemin le plus court
9 :     Δ ← ∅                                           ▷ Les sommets découverts
10 :    pour _ de 1 à n répéter                         ▷ Répéter n fois
11 :        δ, u ← DÉFILER(fp)                          ▷ Choix glouton ! On prend le minimum.
12 :        AJOUTER(Δ, u)
13 :        pour v ∈ g[u] répéter                       ▷ Pour chaque voisin de u
14 :            si v ∉ Δ et d[u] + δ < d[v] alors       ▷ Si la distance est meilleure en passant par u
15 :                d[v] ← d[u] + δ                     ▷ Mettre à jour la distance au voisin
16 :                ENFILER(fp, (d[v],v))               ▷ Enfiler dans la file de priorité la nouvelle priorité
17 :                parents[v] ← u                      ▷ Pour garder la trace du chemin le plus court
18 :    renvoyer d, parents
```

C5. Compléter la fonction `dijkstra : (int * int)list array -> int array * int array` en suivant l'algorithme de Dijkstra 3. Cette fonction renvoie les plus courtes distances à partir d'un sommet d'un graphe ainsi que les directions à prendre. Cette fonction s'appuie sur une file de priorités implémentée par un tas binaire.

```
type 'a qdata = {valeur: 'a; priorite: int};;
type 'a file_priorites = {mutable taille: int; tas: 'a qdata array};;

let echanger t i j = let tmp = t.(i) in t.(i) <- t.(j); t.(j) <- tmp;;

let rec faire_monter tas k = match k with
    | 0 -> ()
    | _ -> let p = (k - 1)/2 in
           if tas.(k).priorite < tas.(p).priorite
           then (echanger tas k p ; faire_monter tas p);;

let rec faire_descendre tas taille k = match k with
    | n when 2*n + 1 >= taille -> () (* pas d'enfants *)
    | n when 2*n + 1 = (taille - 1) -> (* un seul enfant *)
       if tas.(n).priorite > tas.(2*n + 1).priorite
       then echanger tas (2*n + 1) n
    | n -> begin (* deux enfants *)
            let f = if tas.(2*n + 1).priorite < tas.(2*n + 2).priorite then  2*n
                 +  1 else 2*n + 2 in
```

```
            if tas.(n).priorite > tas.(f).priorite then (echanger tas n f;
                faire_descendre tas taille f;)
        end;;

let creer_file_priorites n (v,p) = {taille = 0; tas = Array.init n (fun _ -> {
    valeur = v; priorite=p})};;

let inserer filep (v,p) =
  let size = Array.length filep.tas in
  if filep.taille + 1 > size then failwith "FULL_PRIORITY_QUEUE";
  filep.tas.(filep.taille) <- {valeur=v; priorite=p}; (* Placer l'élément à la
      fin *)
  faire_monter filep.tas filep.taille; (* recréer la structure de tas *)
  filep.taille <- filep.taille + 1;;

let retirer_minimum filep =
  if filep.taille = 0 then failwith "EMPTY_PRIORITY_QUEUE";
  let premier = filep.tas.(0) in (* sauvegader l'élément le plus prioritaire *)
    filep.taille <- filep.taille - 1;
    filep.tas.(0) <- filep.tas.(filep.taille);  (* Placer le dernier élément au
        sommet *)
    faire_descendre filep.tas filep.taille 0;   (* Recréer la structure de tas
        *)
    premier;;   (* renvoyer le résultat *)

let show_path start stop d parents =
    print_string "Cost -> "; print_int d.(stop); print_newline ();
    print_string "Path -> ";
    let rec aux current path =
        if current = start
        then List.iter (fun e -> print_int e; print_string " ") path
        else let father = parents.(current) in aux father ( father :: path )
    in aux stop [ stop ];;

let fp_dijkstra g start stop =
    let fp = creer_file_priorites 10 (max_int,max_int) in
    let n = Array.length g in
    let d = Array.make n max_int in
    d.(start) <- 0;
    let parents = Array.make n start in
    let delta = Array.make n false in
    inserer fp (start, d.(start));
    for _ = 1 to n do
        (* transfert de Delta_bar à Delta *)
        (* mise à jour des distances et de la file de priorités *)
    done;
    show_path start stop d parents;;
```

Solution :

```
  let fp_dijkstra g start stop =
      let fp = creer_file_priorites 10 (max_int,max_int) in
      let n = Array.length g in
      let d = Array.make n max_int in
      d.(start) <- 0;
```

```
        let parents = Array.make n start in
        let delta = Array.make n false in
        inserer fp (start, d.(start));
        for _ = 1 to n do
            let u = (retirer_minimum fp).valeur in
            delta.(u) <- true; (* transfert de Delta_bar à Delta *)
            let update (v, w) =
                if d.(v) > d.(u) + w then
                begin
                    d.(v) <- d.(u) + w;     (* mise à jour de la distance *)
                    parents.(v) <- u;       (* mémoriser le père sur le chemin *)
                    inserer fp (v, d.(v))   (* la nouvelle priorité dans file *)
                end
            in List.iter update (List.filter (fun (v,p) -> not delta.(v)) g.(u))
            (* mise à jour des distances et de la file de priorités *)
        done;
        show_path start stop d parents;;
```

C6. Quelle la complexité de l'algorithme de Dijkstra ainsi implémenté ?

Solution : Si la file de priorité est implémentée par un tas, alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n+m)\log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux!