Chapitre 5

Graphes valués

Option informatique MP1, MP2 & MP3

Les graphes que nous avons étudiés permettent de modéliser les relations entre entités.
Nous allons introduire la possibilité d'une valeur des relations : les arêtes peuvent avoir un poids.
On peut alors prolonger ce poids aux chemin et introduire un questionnement sur l'optimisation de ces valeurs :

- *quel sont les chemins de poids minimum ?*
- *comment recouvrir à moindre coût le graphe ?*
- *quel est le flux maximum entre deux points ?*

I Introduction

I.1 Définitions

Définition 1 : Graphe valué

n graphe valué sur $\mathcal{B}$ est un triplet (S, A, w) où

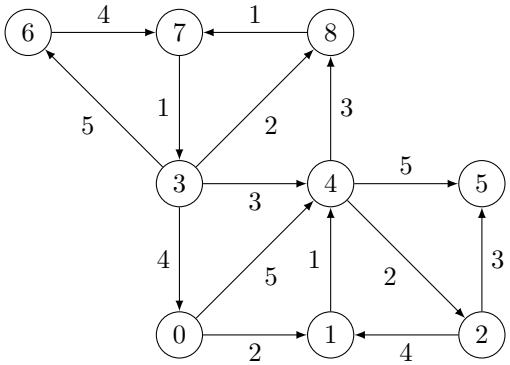
- (S, A) est un graphe
- w est une application de A vers $\mathcal{B}$; $w(i, j)$ est le **poids** de l'arête (i, j).

Les poids des arêtes seront souvent des entiers ou des réels.

Dans ce chapitre, nous nous restreindrons à des poids strictement positifs.
La plupart des résultats seront généralisables dans le cas de poids positifs ou nuls, les démonstrations seraient parfois plus difficiles.

Dans le cas d'un graphe non orienté on impose $w(a, b) = w(b, a)$.

Exemple : $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ et
$A = \{(0,1), (0,4), (1,4), (2,1), (2,5), (3,0), (3,4), (3,6), (3,8), (4,2), (4,5), (4,8), 6,7), (7,3), (8,7)\}$
On détermine le poids avec $w(i, j) = (i + j) \bmod 5 + 1$.



Définition 2 : Poids d'un chemin

Le poids d'un chemin entre deux sommets a et b,

$$a = s_0 \to s_1 \to s_2 \to s_3 \to \ldots \to s_{k-1} \to s_k = b$$

est la somme des poids des arêtes qui le composent, $W(c) = \sum_{i=1}^{k} w(a_{i-1}, a_i)$.

Le chemin vide (direct) entre a et a a le poids 0.

I.2 Implémentations

Les type devront gérer les poids, dont le type sera symbolisé par `'a`. En particulier on introduit la fonction `poids : int -> int -> 'a` telle que `poids i j` renvoie $w(i, j)$.
Les fonctions `voisins` et `aretes` ajouteront le poids dans leurs retours et `ajouter` prendra le poids en paramètre.

Matrice d'adjacence

Une matrice d'adjacence permet de stocker facilement le poids.
Cependant il faut gérer l'absence d'arc.
On marquera souvent la non-existence d'un arc par une arête de poids infini.
Cet infini pourra être représenté par la constante `max_int` dans le cas de poids entiers ou, plus simplement, à l'aide d'un type composé.

```
type 'a arete  = Infini|Poids of 'a;;

type 'a graphe = 'a arete array array;;
```

L'exemple ci-dessus est implémenté par

$$\begin{pmatrix}
\infty & 2 & \infty & \infty & 5 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 1 & \infty & \infty & \infty & \infty \\
\infty & 4 & \infty & \infty & \infty & 3 & \infty & \infty & \infty \\
4 & \infty & \infty & \infty & 3 & \infty & 5 & \infty & 2 \\
\infty & \infty & 2 & \infty & \infty & 5 & \infty & \infty & 3 \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 4 & \infty \\
\infty & \infty & \infty & 1 & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & \infty
\end{pmatrix}$$

Exercice 1 - Implémentation
Écrire les fonctions `creerGraphe`, `taille`, `existe`, `poids`, `ajouter`, `retirer`, `voisins` et `aretes` à l'aide de matrices d'adjacence.

Listes d'adjacence

Pour l'implémentation sous forme de listes d'adjacence on peut maintenir la liste des voisins avec leur poids.

```
type 'a graphe == (int*'a) list array;;
```

L'exemple ci-dessus est implémenté par le tableau ci-dessous. On notera l'alternance de virgules et de points-virgules due à l'omission des parenthèses (cela diminue la lisibilité).

```
[|[1, 2; 4, 5];              [4, 1];              [1, 4; 5, 3];
  [0, 4; 4, 3; 6, 5; 8, 2]; [2, 2; 5, 5; 8, 2]; [];
  [7, 4];                    [3, 1];              [7, 1]|]
```

Exercice 2 - Implémentation
Écrire les fonctions `creerGraphe`, `taille`, `existe`, `poids`, `ajouter`, `retirer`, `voisins` et `arêtes` à l'aide de tableaux d'adjacence.

II Plus court chemin

Dans cette partie nous allons chercher à répondre à la question
"*Quel est le poids minimum d'un chemin entre deux sommets a et b d'un graphe ?*".
On rappelle que les poids sont supposés strictement positifs.

Théorème 1 : Existence d'un chemin de poids minimum

S'il existe un chemin c entre a et b dans un graphe valué à poids positifs alors il existe un chemin de poids minimum entre a et b.

Exercice 3
Démontrer ce résultat.

Lorsqu'il existe un chemin de a vers b, on notera $d(a, b)$ le poids minimum d'un chemin de a vers b, on parle de **distance** entre a et b. Ce n'est pas une distance au sens mathématique dans le cas d'un graphe orienté car on peut avoir $d(a, b) \neq d(b, a)$.
S'il n'existe pas de chemin de a vers b, on posera $d(a, b) = +\infty$.

Exercice 4 - Restriction des chemins minimaux
Montrer que si $a = s_0 \to s_1 \to s_2 \to s_3 \to \ldots \to s_{k-1} \to s_k = b$ est un chemin de poids minimum entre a et b alors $s_i \to s_{i+1} \to \ldots \to s_{j-1} \to s_j$ est un chemin de poids minimum entre s_i et s_j pour $i < j$.

Exercice 5 - Pseudo-distance
Prouver que d vérifie l'inégalité triangulaire : $d(a, c) \leqslant d(a, b) + d(b, c)$ pour des sommets a, b et c tels qu'il existe un chemin de a vers b et un chemin de b vers c.

II.1 Algorithme de Floyd-Warshall

La première méthode que nous allons étudier détermine les poids minimum entre toute paire de sommet. L'algorithme va utiliser l'idée de la programmation dynamique : on va calculer les poids minimum en autorisant les chemins de a vers b à passer par un ensemble de sommets de plus en plus grand.

- On note $\Gamma_k(i, j)$ l'ensemble des chemins simples (c'est-à-dire sans boucle) de i à j qui ne passent, entre i et j, que par des sommets appartenant à $\{0, 1, \ldots, k-1\}$. En particulier $\Gamma_0(i, j)$, pour $i \neq j$, est vide si (i, j) n'est pas un arc et est restreint à (i, j) si $(i, j) \in A$.
  Par convention $\Gamma_k(i, i)$ n'est pas vide, il est réduit au chemin vide de poids nul.
- Les chemins de à $\Gamma_k(i, j)$ sont soit les chemins de $\Gamma_{k-1}(i, j)$ s'ils ne passent pas par k, soit les concaténations d'un chemin de $\Gamma_{k-1}(i, k)$ et d'un chemin de $\Gamma_{k-1}(k, j)$ sinon.
  On a donc $\Gamma_k(i, j) = \Gamma_{k-1}(i, j) \cup \Gamma_{k-1}(i, k).\Gamma_{k-1}(k, j)$.

- On note $d_k(i,j)$ le poids minimum, éventuellement infini, des chemin de $\Gamma_k(i,j)$.
  Ainsi $d_n(i,j)$ est le poids minimum des chemin de a vers b.

Le résultat sur les chemins donne le calcul $d_k(i,j) = \min\{d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)\}$

Exercice 6 - Modification en place
Prouver que $d_{k-1}(i,k) = d_k(i,k)$ ainsi que $d_{k-1}(k,j) = d_k(k,j)$.

D'après l'exercice ci-dessus la matrice `p` dont les coefficients sont les $p_k(i,j)$ peut être modifiée "en place". On peut alors implémenter le calcul de la matrice des poids minimums par modifications successives. Le programme commence par initialiser la matrice à partir des voisins pondérés de chaque sommet puis effectue une boucle pour le calcul des p_k.

Algorithme de Floyd-Warshall

```
let poidsMin g =
   let n = taille g in
   let d = Array.make_matrix n n Infini in
   for i = 0 to (n-1) do
      List.iter (fun (j,k) -> d.(i).(j) <- Poids k)
                (voisins g i);
      d.(i).(i) <- Poids 0 done;
   for k = 0 to (n-1) do
      for i = 0 to (n-1) do
         for j = 0 to (n-1) do
            let new_d = plus d.(i).(k) d.(k).(j) in
            if inferieur new_d  d.(i).(j)
            then  d.(i).(j) <- new_d
         done done done;
   d;;
```

Exercice 7 - Arithmétique avec l'infini
Écrire les fonctions `plus` qui calcule la somme de deux éléments de type `int arete` et `inferieur` qui compare deux éléments de type `int arete`.

Théorème 2 : Complexité de l'algorithme de Floyd-Warshall

La complexité de `poidsMin g` est en $\mathcal{O}(|S|^3)$ pour un graphe $G = (S, A)$.

Démonstration : l'initialisation a une complexité au pire en $\mathcal{O}(n^2)$ et les trois boucles imbriquées donnent la complexité.

Exercice 8 - Calcul du chemin
Modifier le programme de Floyd-Warshall pour qu'il renvoie une fonction `chemin` telle que `chemin a b` retourne un chemin de poids minimal entre a et b.
Cette fonction renverra une liste vide si un tel chemin n'existe pas.

On pourra maintenir une autre matrice dont les éléments on un type

```
type  passage =  Sans | Direct | Milieu of int;;
```

qui signifie l'absence de chemin, un chemin direct ou un chemin passant par k, ce point étant inséré lors de la diminution de la distance au rang k.

II.2 Algorithme de Dijkstra

La deuxième méthode ne calcule que les chemins minimums depuis un sommet s_0 fixé.
Le principe est de calculer, à chaque étape, le sommet le plus proche de s_0 non encore trouvé : c'est un algorithme **glouton** (*greedy* en anglais) qui progresse en cherchant un optimum local et qui permet à la fin d'obtenir un optimum global.
On cherche donc à calculer la suite $\mathbb{S} = (s_0, s_1, s_2, \ldots, s_{r-1})$ des sommets accessibles depuis s_0 selon l'ordre des distances : $0 = d(s_0, s_0) \leqslant d(s_0, s_1) \leqslant d(s_0, s_2) \leqslant \cdots \leqslant d(s_0, s_r)$.

On notera $S_k = \{s_0, s_1, \ldots, s_{k-1}\}$ et $S'_k = S \setminus S_k$.
En particulier $S_1 = \{s_0\}$ et S_r est l'ensemble des sommets accessibles depuis s_0.

Exercice 9 - Croissance
Montrer que les chemins de poids minimum de s_0 à s_k sont de la forme $s_0 \to s_{i_1} \to s_{i_2} \to \cdots \to s_{i_q} \to s_k$ avec $0 < i_1 < i_2 < \cdots i_q < k$.

On définit alors $d_k(s)$ comme le poids minimum d'un chemin de s_0 à s dont les sommets intermédiaires appartiennent à S_k. $d_k(s)$, comme $d(s_0, s)$ peut être infini.

Exercice 10 - Comparaisons
Montrer qu'on a $d(s_0, s) \leqslant d_k(s)$ pour tout s.
Montrer que $d_q(s_p) = d(s_0, s_p)$ pour $q \geqslant p$.

Exercice 11 - Modification de d_k
Montrer que $d_{k+1}(s) = \min\{d_k(s), d_k(s_k) + w(s_k, s)\}$.

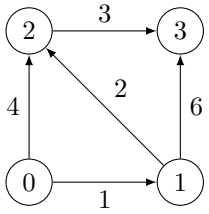
En particulier $d_{k+1}(s) = d_k(s)$ pour s non accessible ou s non voisin de s_k.

Exercice 12 - Choix de s_k
Montrer que s_k vérifie $d_k(s_k) = \inf\{d_k(s)\ ;\ s \in S'_k\}$.

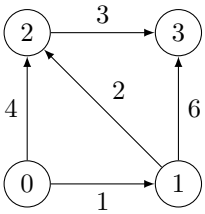
On peut donc définir un algorithme de calcul des distances ; n est le nombre de sommets

1. Définir les distances d_0 avec $d_0(s_0) = 0$ et $d_0(s) = +\infty$ pour les autres points.
2. Définir l'ensemble des sommets vus comme vide.
3. Pour k variant de 0 à $n-1$:
   (a) choisir t tel que $d_k(t)$ est minimum parmi les sommets non vus,
   (b) indiquer t comme vu,
   (c) mettre à jour d_k en d_{k+1} avec $d_{k+1}(s) = \min\{d_k(s), d_k(t) + w(t, s)\}$ pour tous les voisins s de t

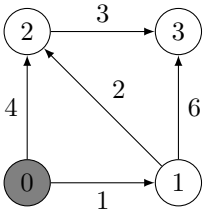
Exemple
On considère le graphe simple suivant, dont on cherche les distance depuis 0.



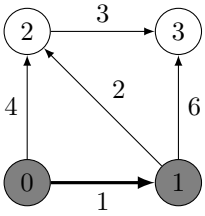
Initialisation `d = [|Poids 0; Infini; Infini; Infini|]`



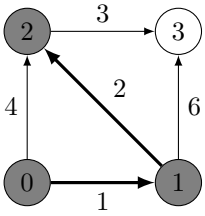
0 est vu `d = [|Poids 0; Poids 1; Poids 4; Infini|]`



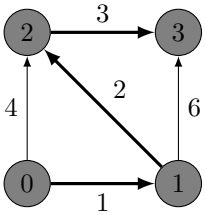
1 est vu `d = [|Poids 0; Poids 1; Poids 3; Poids 6|]`



2 est vu `d = [|Poids 0; Poids 1; Poids 3; Poids 5|]`



3 est vu `d = [|Poids 0; Poids 1; Poids 3; Poids 5|]`



Algorithme de Dijkstra

```
let poids_min g s0 =
    let n = taille g in
    let vus = Array.make n false in
    let p = Array.make n Infini in
    p.(s0) <- Poids 0;
    for i = 0 to (n-1) do
        let s = ind_min p vus in
        let traiter (t, w) =
            let new_p = plus p.(s) (Poids w) in
            if not vus.(t) && inferieur new_p p.(t)
            then p.(t) <- new_p in
        List.iter traiter (voisins g s);
        vus.(s) <- true done;
    p;;
```

On a besoin de la recherche du sommet non vu de poids minimal

```
let ind_min p vus =
    let n = Array.length p in
    let i_min = ref 0 in
    while vus.(!i_min) do incr i_min done;
       for i = !i_min to (n-1) do
       if not vus.(i) && inferieur p.(i) p.(!i_min) then i_min :=
          i done;
    !i_min;;
```

Théorème 3 : Complexité de l'algorithme de Dijkstra classique

La complexité de `poidsMin g s0` est en $\mathcal{O}(|S|^2)$ pour un graphe $G = (S, A)$.

Démonstration : on effectue n passages d'une boucle dans laquelle on recherche un élément dans un tableau de taille n et on modifie au plus n éléments.
Dans le cas de l'algorithme de Floyd-Warshall, on obtenait les poids minimums pour toutes les paires de sommets ; pour nobtenir un résultat semblable, on doit exécuter n fois l'algorithme de Dijkstra. Les complexités sont donc comparables.

La boucle du point 3. de la page 5 devrait être interrompue dès que le minimum est infini, cela advient lorsqu'il existe des sommets non accessibles.
Pour cela on remplace la boucle **`for`** par une boucle **`while`**.

Exercice 13
Modifier la fonction en tenant compte de cette remarque

Exercice 14 - Calcul du chemin
Écrire une fonction `cheminDijkstra g` qui renvoie une fonction `chemin` telle que `chemin a b` retourne un chemin de poids minimal entre a et b.

II.3 Utilisation d'une file de priorité

Dans l'écriture de l'algorithme de Dijkstra, la complexité quadratique provient de la recherche du poids minimum qui est linéaire à chaque étape. Sans cette recherche le nombre d'opérations est simplement linéaire en $|A|$, le nombre d'arêtes du graphe.
Or nous avons étudié une structure de données qui permet d'extraire un extremum en un temps logarithmique par rapport au nombre d'éléments : la file de priorité implémentée par un tas.

Rappel de l'interface des files de priorité

```
fpVide  : 'a  -> 'a fdp
estVide : 'a fdp -> bool
ajouter : 'a -> int -> 'a fdp -> unit
premier : 'a fdp -> 'a
enlever : 'a fdp -> unit
```

On placera dans la file de priorité les couples (b, d) où s est le sommet et d est le poids du chemin de s_0 à t qui commence par un chemin de poids minimum de s_0 à s et qui finit par l'arête (s, t).
On peut modifier l'algorithme comme suit :

1. Définir les distances d avec $d_0(s_0) = 0$ et $d(s) = -1$ pour les autres points.
2. Définir l'ensemble des sommets vus comme vide.
3. Créer une file de priorité vide, f.

4. Ajouter $(s_0, 0)$ dans la file de priorité.
5. Tant que la file n'est pas vide :
   (a) retirer le couple (s, δ) avec δ minimal,
   (b) si $d(s)$ vaut -1,
   (c) alors affecter $d(s) = \delta$
   (d) et placer $\big(t, \delta + w(s,t)\big)$ dans la file pour tous les voisins de s

On notera qu'un sommet t peut être placé plusieurs fois dans la file d'attente.
La clé de comparaison pour la file est définie par le second élément, comme on veut sortir le poids minimum, la clé sera l'opposé du poids.

```
let cle (s, w) = -w;;
```

Algorithme de Dijkstra avec recherche améliorée

```
let poids_min g s0 =
   let n = taille g in
   let fp = creeVide (0, 0) in
   let d = Array.make n (-1) in
   ajouter (s0, 0) fp;
   while not (estVide fp) do
      let (s, dist) =  premier fp in
      enlever fp;
      if d.(s) = -1
      then begin distance.(s) <- dist;
           List.iter (fun (t, w) -> ajouter (t, w + dist) fp)
                     (voisins g s) end done;
   d;;
```

Dans le pire des cas on place les voisins de chaque sommet dans la file d'attente, on y met donc au plus $|A|$ éléments. Comme l'ajout et le retrait d'un élément sont logarithmiques en la taille, on a ici une complexité en $\mathcal{O}\big(|A|\log_2(|A|)\big)$. Comme $|A| \leqslant |S|$, on obtient un $\mathcal{O}\big(|A|\log_2(|S|)\big)$.

Théorème 4 : Complexité de l'algorithme de Dijkstra amélioré

La complexité de `poidsMin g s0` est en $\mathcal{O}\big(|A|\log_2(|S|)\big)$ pour un graphe $G = (S, A)$.

On notera que, pour un graphe dense où $|A|$ est de l'ordre de $|S|^2$, la complexité asymptotique est moins bonne que dans le cas classique car on obtient un $\mathcal{O}\big(|S|^2\log_2(|S|)\big)$.

La preuve de l'algorithme amélioré peut se déduire de la preuve de l'algorithme classique mais on peut le faire d'une autre façon.

Exercice 15 - Lemme de Sedgewick
On suppose qu'on a calculé, pour tout sommet accessible depuis s_0, le poids $\delta(s)$ d'un chemin de s_0 vers s ($(s) = +\infty$ si s n'est pas accessible). $\delta(s)$ est le poids minimal d'un chemin de s_0 à s si et seulement si

1. $\delta(s_0) = 0$
2. $\delta(t) \leqslant \delta(s) + w(s,t)$ pour toute arête (s,t).

Exercice 16 - Preuve de l'algorithme de Dijkstra amélioré
Prouver que l'algorithme de Dijkstra amélioré calcule bien les poids minimums.

Solutions

Solution de l'exercice 1 - Implémentation

```
let creerGraphe n =
   Array.make_matrix n n Infini;;
```

```
let taille g =
   Array.length g;;
```

```
let existe s1 s2 g =
   g.(s1).(s2) <> Infini;;
```

```
let poids s1 s2 g =
   match g.(s1).(s2) with
   |Infini -> failwith "L'arête n'existe pas"
   |Poids x -> x;;
```

```
let ajouter s1 s2 w g =
   g.(s1).(s2) <- (Poids poids);;
```

```
let retirer s1 s2 g =
   g.(s1).(s2) <- Infini;;
```

```
let voisins s g =
   let v = ref [] in
   for i = 0 to (taille g - 1) do
      match g.(s).(i) with
      |Infini -> ()
      |Poids k -> v := (i,k) :: !v  done;
   !v;;
```

```
let aretes g =
   let a = ref [] in
   for i = 0 to (taille g - 1) do
       a := (voisins i g) @ !a done;
   !a;;
```

Solution de l'exercice 2 - Implémentation

```
let creerGraphe n =
   Array.make n [];;
```

```
let taille g =
   Array.length g;;
```

```
let existe s1 s2 g =
   let rec dans s liste =
      match liste with
      |[] -> false
      |(t, w) :: q -> (t = s) || (dans s q)
   in dans s2 g.(s1);;
```

La fonction auxiliaire teste l'appartenance d'un couple `(s,w)` avec `s` donné dans la liste

```
let poids s1 s2 g =
   let rec assoc s liste =
      match liste with
      |[] -> failwith "L'arête n'existe pas"
      |(t, w) :: q when t = s -> w
      |(t, w) :: q -> assoc s q
   in assoc s2 g.(s1);;
```

La fonction auxiliaire calcule le second membre d'un couple `(s,w)` avec `s` donné dans la liste

```
let ajouter s1 s2 w g =
   if not existe s1 s2 g
   then g.(s1) <- (s2, w) :: g.(s1) ;;
```

```
let retirer s1 s2 g =
   let rec moins s liste =
      match liste with
      |[] -> []
      |(t, w)::q when t = s -> q
      |c::q -> c::(moins s q)
   in g.(s1) <- moins s2 g.(s1);;
```

```
let voisins s g =  g.(s);;
```

```
let aretes g =
   let a = ref [] in
   for i = 0 to (taille g -1) do
      a :=  (voisins i g) @ !a done;
    !a;;
```

Solution de l'exercice 3

La longueur (nombre d'arêtes le composant) d'un chemin de a vers b est un entier positif ; il existe donc un chemin de longueur minimale : $c : a = s_0 \to s_1 \to \cdots \to s_{p-1} \to s_p = b$.

Si ce chemin n'est pas simple, il existe deux sommets égaux dans le chemin, $s_i = s_j$, et on obtient un chemin de poids strictement inférieur entre a et b en retirant la portion $s_i \to s_{i+1} \to \cdots \to s_{j-1} \to s_j = s_i$.

Après un nombre fini de telles opérations, on aboutit à un chemin simple : le poids d'un chemin non simple est strictement minoré par le poids d'un chemin simple.

On en déduit que la borne inférieure des poids des chemins de a vers b est égale à la borne inférieure des poids des chemins simples de a vers b

Les chemins simples passent par des sommets distincts donc sont de longueur $n-1$ au plus.

Ainsi il existe un nombre fini de chemins simples de a vers b, et la borne inférieure est atteinte : il existe donc un chemin de poids minimum, c_0.

Solution de l'exercice 4 - Restriction des chemins minimaux
Si $s_i \to s_{i+1} \to \ldots \to s_{j-1} \to s_j$ n'était pas un chemin de poids minimum entre s_i et s_j alors, en considérant un chemin de poids minimum $s_i = t_0 \to t_1 \to \ldots \to t_{q-1} \to t_q = s_j$, on obtiendrait un chemin de poids strictement inférieur entre a et b avec

$$a = s_0 \to s_1 \to \cdots \to s_i = t_0 \to t_1 \to \ldots \to t_{q-1} \to t_q = s_j \to \ldots \to s_{k-1} \to s_k = b$$

ce qui contredirait la minimalité du poids du chemin.
Le chemin partiel est bien un chemin de poids minimum entre s_i et s_j.

Solution de l'exercice 5 - Pseudo-distance
Un chemin de poids minimum de a à b concaténé avec un chemin de poids minimum de b à c donne un chemin de a à c de poids $d(a,b) + d(b,c)$. Ce poids est minoré par le poids minimum $d(a,c)$.

Solution de l'exercice 6 - Modification en place
On a $d_k(i,k) = \min\{d_{k-1}(i,k), d_{k-1}(i,k) + d_{k-1}(k,k)\} = d_{k-1}(i,k)$ car $d_{k-1}(k,k) = 0$.
Les formules de passages peuvent donc utiliser indistinctement les termes $p_{k-1}(i,k)$ ou $p_k(i,k)$; on peut donc travailler sur les termes de la matrice en utilisant les valeurs des $p_{i,k}$ qu'ils aient été transformés ou non.

Solution de l'exercice 7 - Arithmétique avec l'infini

```
let plus a1 a2 =
   match (a1, a2) with
   |(Infini, _) -> Infini
   |(_, Infini) -> Infini
   |(Poids p1, Poids p2) -> Poids (p1 + p2);;
```

```
let inferieur a1 a2 =
   match (a1, a2) with
   |(Infini, _) -> false
   |(_, Infini) -> true
   |(Poids p1, Poids p2) -> p1 < p2;;
```

Solution de l'exercice 8 - Calcul du chemin

```
let passages g =
   let n = taille g in
   let d = Array.make_matrix n n Infini in
   let ch = Array.make_matrix n n Sans in
   for i = 0 to (n-1) do
      List.iter (fun (j,k) -> d.(i).(j) <- Poids k) (voisins g i);
      List.iter (fun (j,k) -> ch.(i).(j) <- Direct) (voisins g i);
      d.(i).(i) <- Poids 0;
      ch.(i).(i) <- Direct done;
   for k = 0 to (n-1) do
      for i = 0 to (n-1) do
         for j = 0 to (n-1) do
            let d1 = d.(i).(j) in
            let d2 = plus d.(i).(k) d.(k).(j) in
            if inferieur d2 d1
            then begin d.(i).(j) <- d2;
                       ch.(i).(j) <- Milieu (k) end
         done done done;
   ch;;
```

```
let cheminFloyd g =
   let ch = passages g in
   let rec chemin i j =
      match ch.(i).(j) with
      |Sans -> []
      |Direct -> if i = j then [i] else [i; j]
      |Milieu k -> (chemin i k)@(List.tl (chemin k j)) in
   chemin;;
```

Solution de l'exercice 9 - Croissance
On considère un chemins de poids minimum de s_0 à s_k : $s_0 \to t_1 \to t_2 \to \cdots \to t_q \to s_k$.
On a vu à l'exercice 4 que $s_0 \to t_1 \to t_2 \to \cdots \to t_i$ définit un chemin minimum de s_0 à t_i donc $d(s_0, t_i) < d(s_0, s_k)$. D'après la définition de $\mathbb{S}$ cela implique que t_i appartient à S_k.
Ce chemin minimum $s_0 \to t_1 \to t_2 \to \cdots \to t_i = s_m$ implique, de la même manière, que $t_j \in S_m$ pour $j < i$.
On a prouvé que les sommets d'un chemin de poids minimum forment une extraction strictement croissante de $\mathbb{S}$.

Solution de l'exercice 10 - Comparaisons
Un chemin de poids minimal de s_0 à s dans S_k est un chemin de s_0 à s dans S donc son poids est minoré par celui d'un chemin de poids minimal : $d(s_0, s) \leqslant d_k(s)$.
L'exercice précédent montre que $d(s_0, s_p) = d_p(s_p)$ car tout chemin de poids minimal est dans S_p.
De même un chemin de poids minimal de s_0 à s_p dans S_p est un chemin de s_0 à s_p dans S_q car $S_p \subset S_q$ donc $d_q(s_p) \leqslant d_p(s_p)$.
On a ainsi $d(s_0, s_p) \leqslant d_q(s_p) \leqslant d_p(s_p) = d(s_0, s_p)$ d'où l'égalité.

Solution de l'exercice 11 - Modification de d_k
$d_{k+1}(s)$ est le poids d'un chemin $c = s_0 \to s_{i_1} \to s_{i_2} \to \cdots \to s_{i_m} \to s$ avec $i_j \leqslant k$. Alors $s_0 \to s_{i_1} \to s_{i_2} \to \cdots \to s_{i_m}$ doit être un chemin de poids minimal de s_0 vers s_{i_m} dans S_{k+1}. Son poids est donc $d_{k+1}(s_{i_m})$ qui est égal à $d(s_0, s_{i_m})$ d'après l'exercice 10.
L'exercice 9 permet de conclure qu'on a $0 < i_1 < \cdots < i_m$.
Si $i_m < k$ alors c est un chemin dans S_k donc $d_{k+1}(s) \geqslant d_k(s)$.
Si $i_m = k$ alors le chemin c est le prolongement d'un chemin de s_0 vers s_k dans S_k et de l'arête (s_k, s) donc son poids vérifie $d_{k+1}(s) \geqslant d_k(s_k) + w(s_k, s)$.
Dans les deux cas on a $d_{k+1}(s) \geqslant \min\{d_k(s), d_k(t) + w(t, s)\}$.
Inversement, tout chemin de s_0 vers s dans S_k est un chemin de s_0 vers s dans S_{k+1} donc $d_{k+1}(s) \leqslant d_k(s)$, les deux valeurs étant infini.
Tout chemin de tout chemin de s_0 vers s_k dans S_k prolongé par (s_k, s) est un chemin de s_0 vers s dans S_{k+1} donc $d_{k+1}(s) \leqslant d_k(s_k) + w(s_k, s)$. Cette égalité est triviale si (s_k, s) n'est pas une arête.
Ainsi $d_{k+1}(s) \geqslant \min\{d_k(s), d_k(t) + w(t, s)\}$ d'où l'égalité.

Solution de l'exercice 12 - Choix de s_k
On doit montrer que, si $d_k(t) = \inf\{d_k(s)\ ;\ s \in S'_k\}$ alors, pour tout $s \in S'_k$, $d(s_0, t) \leqslant d(s_0, s)$.
$d_k(t)$ est le poids d'un chemin de s_0 à t donc on a $d(s_0, t) \leqslant d_k(t)$.
Si s n'est pas accessible depuis s_0 alors $d(s_0, s) = +\infty \geqslant d(s_0, t)$.
Si s est accessible, il existe l tel que $s = s_l$. Comme $s \notin S_k$, on a $l \geqslant k$.
On a vu qu'alors un chemin de poids minimum est de s_0 à s_p est de la forme
$s_0 \to s_{i_1} \to s_{i_2} \to \cdots \to s_{i_q} \to s_p$ avec $0 = i_0 < i_1 < i_2 < \cdots < i_q < l = i_{q+1}$.
Il existe m tel que $i_{m-1} < k \leqslant i_m$ et on a alors $s_0 \to s_{i_1} \to s_{i_2} \to \cdots \to s_{i_{m-1}} \to s_{i_m}$ qui est un chemin de poids minimum dont les sommets intermédiaires appartiennent à S_k d'où $d(s_0, s) = d(s_0, s_l) \geqslant d(s_0, s_{i_m}) = d_k(s_{i_m}) \geqslant d_k(t)$; par minimalité de $d_k(t)$.
On a bien $d(s_0, t) \geqslant d(s_0, s)$ pour tout $s \in S'_k$ et t peut être choisi comme point s_k.

Solution de l'exercice 13

```
let poids_min g s0 =
    let n = taille g in
    let vus = Array.make n false in
    let p = Array.make n Infini in
    p.(s0) <- Poids 0;
    let continue = ref true in
    let i = ref 0 in
    while !continue && !i < n do
        let s = ind_min p vus in
        if p.(s) = Infini
        then continue := false
        else begin
             let traiter (t, w) =
                let new_p = plus p.(s) (Poids w) in
                if not vus.(t) && inferieur new_p p.(t)
                then p.(t) <- new_p in
             List.iter traiter (voisins g s) end;
             vus.(s) <- true;
             incr i done;
    p;;
```

Solution de l'exercice 14 - Calcul du chemin
On modifie la fonction précédente en construisant le tableau des pères.

```
let peres g a=
    let n = taille g in
    let p = Array.make n Infini in
    let vus = Array.make n false in
    let pere = Array.make n (-1) in
    p.(a) <- Poids 0;
    pere.(a) <- a;
    let continue = ref true in
    let i = ref 0 in
    while !continue && !i < n do
        let s = ind_min p vus in
        if p.(s) = Infini
        then continue := false
        else begin
                let traiter (t, w) =
                let new_p = plus p.(s) (Poids w) in
                if not vus.(t) && inferieur new_p p.(t)
                then (p.(t) <- new_p; pere.(t) <- s) in
                List.iter traiter (voisins g s) end;
                vus.(s) <- true;
                incr i done;
    pere;;
    let rec construire s fait =
        match pere.(s) with
        |(-1) -> []
        |t when t = s -> fait
        |t -> construire t (t::fait) in
    construire b [b];;
```

```
let peres g a=
    let n = taille g in
    let p = Array.make n Infini in
    let vus = Array.make n false in
    let pere = Array.make n (-1) in
    p.(a) <- Poids 0;
    pere.(a) <- a;
    let continue = ref true in
    let i = ref 0 in
    while !continue && !i < n do
        let s = ind_min p vus in
        if p.(s) = Infini
        then continue := false
        else begin
                let traiter (t, w) =
                let new_p = plus p.(s) (Poids w) in
                if not vus.(t) && inferieur new_p p.(t)
                then (p.(t) <- new_p; pere.(t) <- s) in
                List.iter traiter (voisins g s) end;
                vus.(s) <- true;
                incr i done;
    let rec construire s fait =
        match pere.(s) with
        |(-1) -> []
        |t when t = s -> fait
        |t -> construire t (t::fait) in
    construire b [b];;
```

Solution de l'exercice 15 - Lemme de Sedgewick

Si $\delta(s)$ est le poids minimal d'un chemin de s_0 à s alors, trivialement, $\delta(s_0) = 0$.
De plus, pour tous sommets accessibles s et t tels que (s,t) est une arête, $\delta(s) + w(s,t)$ est le poids d'un chemin de s_0 à t obtenu en prolongeant un chemin de poids minimal de s_0 à s par l'arête (s,t) ; ce poids est minoré par le poids minimal, $\delta(t)$.

Inversement, supposons que les distances vérifie les propriétés. Soit s accessible depuis s_0 ; on considère un chemin de poids minimal de s_0 à s : $s_0 \to s_1 \to s_2 \to \cdots \to s_{q_1} \to s_q = s$.

Le poids minimal est donc $d(s) = \sum_{k=1}^{q} w(s_{k-1}, s_k) \geqslant \sum_{k=1}^{q} \big(\delta(s_k) - \delta(s_{k-1})\big)$ d'après la propriété 2).
En téléscopant on obtient donc $d(s) \geqslant \delta(s_q) - \delta(s_0) = \delta(s) - 0$ d'après la propriété 2).
Comme $\delta(s)$ est le poids d'un chemin on a $\delta(s) \geqslant d(s)$ d'où l'égalité.
$\delta(s)$ est bien le poids minimal d'un chemin de s_0 à s pour tout s accessible..

Solution de l'exercice 16 - Preuve de l'algorithme de Dijkstra amélioré

- On commence par montrer que `d.(s)` est bien le poids d'un chemin de `s0` à `s` lorsque sa valeur est différente de -1.

  Pour cela on montre l'invariant de boucle : pour tout couple `(s, dist)` ajouté dans la file de priorité, `dist` est le poids d'un chemin de `s0` à `s`.

  Initialement la file de priorité ne contient que `(s0, 0)`, l'invariant est donc vérifié.

  Si cela est vrai avant le traitement d'un couple dans la boucle **`while`**, on n'ajoute que des couples `(t, dist + w)` où `dist + w` est bien le poids du chemin de `s0` à `t` obtenu en prolongeant le chemin de `s0` à `s` de poids `dist` par l'arête `(s, t)` de poids `w`. L'invariant est donc vrai à la fin de la boucle.

  Ainsi, comme on affecte `d.(s) <- dist`, `d.(s)` est le poids d'un chemin de `s0` à `s`.

- Comme dans les autres preuves de parcours, on montre que tous tous les points accessibles sont placés au moins une fois dans la file de priorité donc auront une valeur différente de -1 pour `d.(s)`.
- Lors du premier passage dans la boucle **`while`**, la file de priorité ne contient que `(s0, 0)` donc on traite `s0` en affectant 0 à `d.(s0)`.
- Lorsque l'on donne une valeur `dist` à `d.(s)`, on place dans la file de priorité les couples `(t, dist + w)` où w est le poids de (s, t).