Chapitre 20
Plus courts chemins

On présente dans ce chapitre un problème typique de cheminement dans les graphes : la recherche d'un plus court chemin entre deux sommets. Ce problème a de nombreuses applications. On en a déjà vu un exemple au chapitre 8 : trouver le moyen le plus économique pour aller de Brest à Lyon, connaissant pour chaque ligne aérienne le prix du billet d'avion. Citons quelques autres applications : les problèmes d'optimisation de réseaux (réseaux routiers ou réseaux de télécommunications), les problèmes d'ordonnancement et les problèmes d'intelligence artificielle tels que la circulation dans un labyrinthe.

1. Définition

1.1. Spécification

On s'intéresse ici aux chemins de longueur non nulle. Formellement, la notion de chemin doit être spécifiée, comme au chapitre 19 à l'aide de l'opération de *fermeture transitive*. Plus précisément, il existe un chemin de x vers y dans le graphe orienté G si, et seulement si, il existe un arc de x vers y dans *fermeture-transitive*(G).

La notion de plus court chemin suppose qu'on travaille sur un graphe valué orienté G. On a donc une fonction *coût* définie sur les couples de sommets (x, y) tels que $x \rightarrow y$ est un arc de G, et à valeurs dans $\mathbb{R}$.

On appelle ***coût cumulé d'un chemin*** μ (ou plus simplement coût ou poids de μ), noté *coût*(μ), la somme des coûts des arcs qui le composent. On appelle ***plus court chemin de x vers y*** un chemin allant de x à y de coût minimum. Un tel chemin n'existe pas toujours. On appelle ***plus petite distance de x à y***, le coût de ce chemin s'il existe.

Afin de pouvoir exprimer les algorithmes de construction des plus courts chemins, on ajoute à la spécification d'un graphe orienté valué, donnée au chapitre 8, les opérations suivantes :

ppdistance : Sommet × Sommet × Graphe → Réel
père : Sommet × Sommet × Graphe → Sommet

père(x, y, G) désigne le sommet de G qui précède y dans un plus court chemin de x à y dans G, et *ppdistance*(x, y, G) est le coût de ce chemin.

Ces deux opérations ne sont pas définies partout, mais seulement pour les couples de sommets (x, y) pour lesquels il existe un plus court chemin de x vers y. Quand elles sont définies, elles satisfont les propriétés suivantes :

(1) pour tout chemin $x_0, x_1, \ldots, x_m$ de G,
$$\sum_{i=0,\ldots,m-1} \textit{coût}(x_i, x_{i+1}, G) \geq \textit{ppdistance}(x_0, x_m, G)$$

(2) pour tout chemin $x_0, x_1, \ldots, x_m$ de G
$(\forall i$ tel que $0 \leq i < m-1, x_i = \textit{père}(x_0, x_{i+1}, G)) \Rightarrow$
$$\sum_{i=0,\ldots,m-1} \textit{coût}(x_i, x_{i+1}, G) = \textit{ppdistance}(x_0, x_m, G)$$

Dans ce chapitre, on recherche des plus courts chemins dans un graphe G fixé. Pour simplifier les notations, on omettra, s'il n'y a pas d'ambiguïté, de mentionner G en argument des opérations sur les sommets, les arcs et les chemins.

1.2. Conditions d'existence

A quelles conditions le problème de la recherche d'un plus court chemin admet-il une solution ?

Considérons un chemin μ allant de x à y ; supposons que ce chemin contienne un circuit Γ. Soit μ' le chemin obtenu en supprimant ce circuit de μ. On a :

$$\textit{coût}(\mu) = \textit{coût}(\mu') + \textit{coût}(\Gamma)$$

Si *coût*(Γ) est négatif (strictement inférieur à zéro), il n'existe pas de plus court chemin de x à y, car étant donné un chemin λ de x à y, on peut toujours construire un chemin λ' de coût strictement inférieur à λ, en passant une fois de plus dans Γ.

Si *coût*(Γ) est positif ou nul, μ' est de coût inférieur ou égal à μ ; c'est un meilleur candidat que μ pour être solution du problème si coût(Γ) est strictement positif.

Les circuits de coût négatif sont appelés ***circuits absorbants***. Dans toute la suite de ce paragraphe, on suppose qu'il n'existe pas de tels circuits. S'il existe des circuits, ils sont donc de coût positif ou nul. Or, on vient de voir qu'un plus court chemin ne peut contenir un circuit de coût strictement positif. S'il y a des circuits de coût nul, il y a plusieurs solutions au problème de la recherche d'un plus court chemin.

Comme le nombre de chemins élémentaires de x vers y (rappelons qu'il s'agit de chemins qui ne contiennent pas plusieurs fois un même sommet) est fini, il existe

au moins un plus court chemin élémentaire de x vers y. (Notons qu'il peut exister plusieurs plus courts chemins élémentaires.) Un plus court chemin élémentaire est un plus court chemin de x vers y, s'il n'y a pas de circuit absorbant.

Les algorithmes proposés ici, dans le cas où il n'y a pas de circuit absorbant, donnent des solutions qui sont des ***chemins élémentaires.***

En conclusion, le problème de la recherche d'un plus court chemin de x vers y, admet une solution dès qu'il existe un chemin de x vers y, et qu'il n'y a pas de circuit absorbant.

1.3. Variantes du problème

Le problème de la recherche d'un plus court chemin se rencontre sous l'une des trois formes suivantes :

(i) on recherche un plus court chemin entre deux sommets donnés,

(ii) on recherche les plus courts chemins entre un sommet donné, appelé *source* et tous les autres sommets,

(iii) on recherche les plus courts chemins entre tous les sommets pris deux à deux.

On ne connaît pas de solution meilleure pour le problème (i) que celle qui consiste à passer par les solutions de (ii). On s'intéresse dans les paragraphes suivants aux problèmes (ii) et (iii). Les deux premiers algorithmes présentés résolvent le problème (ii) et le troisième le problème (iii).

Il existe différents algorithmes qui sont plus ou moins bien adaptés selon les propriétés du graphe et la forme du problème étudié ((ii) ou (iii)), mais il n'y a pas d'algorithme général qui soit efficace et intéressant dans tous les cas.

2. Algorithme de Dijkstra (coûts ≥ 0)

Cet algorithme n'est utilisable que dans le cas, très fréquent, où les coûts des arcs sont tous positifs ou nuls. Il calcule un plus court chemin entre une source s et tous les sommets accessibles depuis s : on obtient alors une *arborescence de racine* s formée par ces plus courts chemins.

2.1. Principe de l'algorithme

Soit $G = < S,\ A,\ C >$ un graphe valué orienté de n sommets. Soit s un sommet de G. On construit progressivement un ensemble CC de sommets x pour lesquels on connaît un plus court chemin de s vers x.

Au départ CC ne contient que s. A chaque étape, on ajoute un sommet x à CC pour lequel *ppdistance*(s,x) et *père*(s,x) sont connus. On s'arrête lorsque CC contient tous les sommets accessibles depuis s.

Soit $M = S - CC$. Pour tout sommet y de M on appelle ***chemin de s à y dans CC*** tout chemin de s à y ne comportant que des éléments de CC, sauf y. On note $distance_{s,CC}(y)$ le coût d'un plus court chemin de s à y dans CC et $pred_{s,CC}(y)$ le prédécesseur de y dans ce plus court chemin. A chaque étape, on choisit un sommet m de M tel que :

$$distance_{s,CC}(m) = \inf_{y \in M}\{distance_{s,CC}(y)\}$$

En effet, pour un tel sommet, on montre (voir plus loin la justification de l'algorithme) que :

$distance_{s,CC}(m) = ppdistance(s,m)$
$pred_{s,CC}(m) = père(s,m)$

On supprime m de M et on l'ajoute à CC. Comme m est maintenant élément de CC, pour un sommet y de M, il peut exister de nouveaux chemins de s à y dans CC (ce sont des chemins passant par m tels que m est prédécesseur de y). Pour tout sommet y restant dans M, il faut modifier les valeurs de $distance_{s,CC}(y)$ et de $pred_{s,CC}(y)$ si l'ancienne valeur de $distance_{s,CC}(y)$ est strictement supérieure à $distance_{s,CC}(m)$ + *coût*(m,y).

2.2. Algorithme

On stocke $distance_{s,CC}(i)$ dans la case d'indice i du tableau d, et $pred_{s,CC}(i)$ dans la case d'indice i du tableau pr. Pour tout sommet i, on initialise $pr[i]$ à s et $d[i]$ à *coût*(s,i).

Pour éviter des tests, on prolonge l'opération *coût* pour qu'elle soit définie pour tout couple de sommets (x,y) avec :

coût$(x,x) = 0$ (ce qui est possible car on a supposé les graphes sans boucle)

x *arc* y = faux $\Rightarrow$ *coût*$(x,y) = \infty$, où ∞ désigne une valeur strictement plus grande que tous les coûts du graphe.

On se donne une opération *choisir-min*$(M,\ d)$ qui a pour résultat un élément m de M tel que $d[m]$ est minimum.

Dans la procédure qui suit, on utilise également les opérations *ensemble-vide*, *supprimer*, *ajouter* et $\in$ sur les ensembles, qui ont été spécifiées au chapitre 6 et les opérations sur les graphes spécifiées au chapitre 8.

```
procedure Dijkstra(s : integer; G : Graphe; var d : array[1..n] of real;
                                  var pr : array[1..n] of integer);
{Le graphe G a n sommets : 1, 2,..., n ; s est un sommet de G ; on recherche un
plus court chemin de s à chacun des autres sommets ; les coûts doivent être positifs
ou nuls ; après exécution, si d[y] < ∞, d[y] est le coût d'un plus court chemin de s
vers y et pr[y] est le prédécesseur de y sur ce chemin }
var i, y, m : integer; v : real; M : Ensemble;
{y et m sont des sommets}


begin
    {initialisations}
    M := ensemble-vide;
    for i := 1 to n do begin
        d[i] := coût(s, i, G);
        pr[i] := s;
        M := ajouter(i, M)
    end;
    M := supprimer(s, M); {CC = {s}}
    {enrichissements successifs de CC :}
    while M <> ensemble-vide do begin
        m := choisir-min(M, d);
        if d[m] = ∞ then return; {car les sommets qui restent dans M sont
        inaccessibles depuis s}
        M := supprimer(m, M); {on ajoute m à CC}
        for i:= 1 to d^o+ de m dans G do begin
            {on réajuste les valeurs de distance_{s,CC}(y) et pred_{s,CC}(y) :}
            y := i ème-succ-de m dans G;
            if y ∈ M then begin
                    v := d[m] + coût(m, y, G);
                    if v < d[y] then begin d[y] := v; pr[y] := m end
            end
        end
    end
end Dijkstra;
```

2.3. Exemple

On considère le graphe G orienté valué par des coûts positifs ou nuls de la figure 1. Les nombres au-dessus des arcs indiquent leur coût. On cherche un plus court chemin de $s1$ vers chaque sommet accessible depuis $s1$, ainsi que son coût.

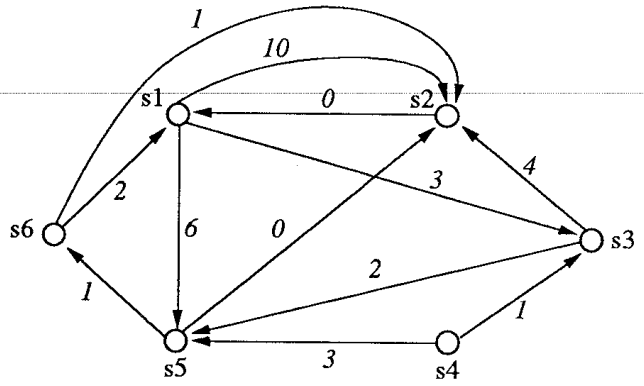

Figure 1. Graphe orienté valué par des coûts positifs ou nuls.

L'exécution de la procédure sur G se fait selon les étapes suivantes :

1) Initialisation : $M = \{s2, s3, s4, s5, s6\}$

d :	0	10	3	∞	6	∞
	1	2	3	4	5	6

pr :	1	1	1	1	1	1
	1	2	3	4	5	6

2) $M = \{s2, s4, s5, s6\}$

d :	0	7	3	∞	5	∞
	1	2	3	4	5	6

pr :	1	3	1	1	3	1
	1	2	3	4	5	6

3) $M = \{s2, s4, s6\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

4) $M = \{s4, s6\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

5) $M = \{s4\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

La procédure s'arrête car *choisir-min*(M, d) $= s4$ et $d[4] = \infty$.

A l'issue de la procédure, on obtient une arborescence de racine s des plus courts chemins de s vers les autres sommets accessibles, «en remontant» dans le tableau pr (voir exercices). Par exemple, un plus court chemin de $s1$ vers $s6$ est :

$$s1 \rightarrow s3 \rightarrow s5 \rightarrow s6 \text{ car } pr[6] = 5,\ pr[5] = 3 \text{ et } pr[3] = 1.$$

On n'a pas trouvé de plus court chemin de $s1$ vers $s4$, ($d[4] = \infty$, à la fin), car $s4$ est inaccessible depuis $s1$.

Remarque : L'hypothèse «les coûts sont tous positifs ou nuls» est fondamentale. L'utilisation de l'algorithme pour le graphe de la figure 2, où les poids ne sont pas tous positifs ou nuls, conduit à un résultat incorrect si on choisit comme source le sommet $s1$. En effet, on ôte d'abord le sommet $s2$ de M, et on n'examine plus $distance_{s1,CC}(s2)$ alors que le chemin de $s1$ vers $s2$ passant par $s3$ est plus court.

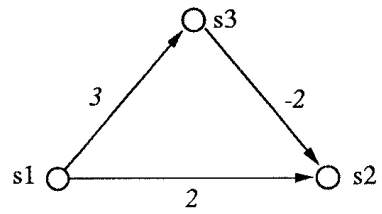

Figure 2. Graphe orienté valué par des coûts quelconques.

2.4. Justification de l'algorithme

Si à la fin de la procédure, pour un sommet i, $d[i] = \infty$, c'est qu'il n'existe pas de chemin s vers i et par suite, pas de plus court chemin de s vers i. Ce cas ne se produit pas si s est une racine du graphe.

On va maintenant montrer que si à la fin de la procédure $d[x] < \infty$, alors $d[x] = ppdistance(s,\ x)$ et $pr[x] = père(s,\ x)$.

Considérons l'ensemble M à une étape donnée. Soit m un sommet de M tel que $distance_{s,CC}(m) = \inf_{y \in M} \{distance_{s,CC}(y)\}$.

$distance_{s,CC}(m)$ est le coût d'un plus court chemin λ de s à m dans CC. Pour montrer que $distance_{s,CC}(m) = ppdistance(s, m)$, il faut montrer qu'il n'existe pas de chemin, qui ne soit pas dans CC, plus court que λ.

Soit μ un chemin de s à m qui n'est pas dans CC ; soit w le premier sommet de μ qui est dans M. Le chemin μ se décompose en deux chemins μ_1 et μ_2 ; μ_1 allant de s à w et μ_2 de w à m (cf. figure 3) ; μ_1 est un chemin de s à w dans CC. Comme m réalise le minimum de $\{distance_{s,CC}(y)\}$ pour y élément de M, $coût(\mu_1) \geq coût(\lambda)$. Or $coût(\mu_2) \geq 0$, puisque ***les coûts sont positifs ou nuls*** (c'est ici que cette

hypothèse intervient). Par conséquent :

$$coût(\mu) = coût(\mu_1) + coût(\mu_2) \geq coût(\lambda).$$

λ est bien un plus court chemin de s vers m :

$$distance_{s,CC}(m) = ppdistance(s,\ m).$$

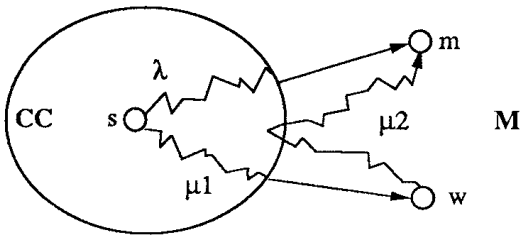

Figure 3. Chemins dans CC.

De plus, $pred_{s,CC}(m)$ qui est le prédécesseur de m dans λ, est bien le prédécesseur de m dans un plus court chemin de s à m :

$$pred_{s,CC}(m) = père(s,\ m).$$

On justifie maintenant les réajustements des tableaux d et pr après chaque modification de CC. Soit M' l'ensemble M privé de m et soit CC' l'ensemble CC auquel on a ajouté m. En ajoutant m à CC on crée la possibilité, pour tout sommet y de M', d'un chemin μ dans CC' plus court que le plus court chemin dans CC. On va montrer qu'un tel chemin μ doit être un chemin dans CC de s vers m suivi de l'arc $m \rightarrow y$.

Considérons un chemin μ dans CC' de s vers y passant par m; il se décompose en un chemin μ_1, chemin de s à m dans CC, suivi d'un arc de m vers un sommet x de CC, puis d'un chemin μ_2 dans CC vers un sommet z, suivi de l'arc $z \rightarrow y$ (cf. figure 4). Montrons que μ n'est pas plus court que le plus court chemin de s à y dans CC.

Puisque z a été ôté de M avant m, il existe un plus court chemin μ_3 de s à z, qui est dans CC et tel que :

$coût(\mu_3) \leq coût(\mu_1) + coût(m,\ x) + coût(\mu_2)$.

Or $coût(\mu) = coût(\mu_1) + coût(m,\ x) + coût(\mu_2) + coût(z,\ y)$.

Donc $coût(\mu) \geq coût(\mu_3) + coût(z,\ y)$.

Le chemin μ est donc de coût supérieur ou égal au coût d'un plus court chemin de s à y dans CC.

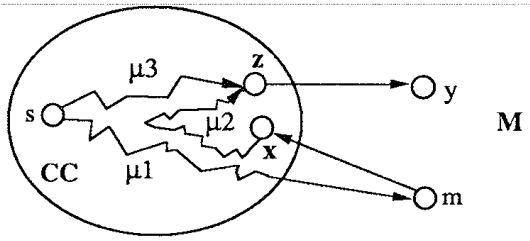

Figure 4. Plus court chemin dans $CC \cup \{m\}$ impossible.

Il en résulte que les seuls chemins de s à y dans CC', qui peuvent être strictement plus courts que le plus court chemin de s à y dans CC, se décomposent en un chemin de s à m dans CC, suivi de l'arc $m \rightarrow y$.

Le plus court de ces chemins a pour coût : $distance_{s,CC}(m)$ + $coût(m, y)$. Si ce coût est strictement inférieur à $distance_{s,CC}(y)$, on a trouvé un chemin de s à y, dans CC', plus court que les chemins de s à y dans CC, et le prédécesseur de y dans ce plus court chemin est m : $pred_{s,CC}(y) = m$.

2.5. Analyse de la complexité

On évalue la complexité au pire de l'algorithme en nombre total d'opérations, en fonction du nombre de sommets n et du nombre d'arcs p du graphe.

• La première boucle, qui correspond à l'initialisation des tableaux, comporte $\Theta(n)$ affectations. L'initialisation de M est également en $\Theta(n)$.

• Il y a au plus $n - 1$ itérations dans la boucle **while**. Le nombre $n - 1$ correspond au cas où s est racine du graphe.

Examinons le coût de l'instruction $m :=$ *choisir-min*(M, d). Son coût au pire est du même ordre que le nombre de comparaisons effectuées.

(i) Si M est représenté par une liste chaînée de longueur $|M|$, la recherche de *choisir-min*(M, d) nécessite $|M| - 1$ comparaisons. Comme $|M|$ diminue de 1 à chaque itération de la boucle **while**, et cela au plus $(n - 1)$ fois, on a au plus $\sum_{i=1}^{n-1}(i - 1)$ comparaisons en tout, soit $\Theta(n^2)$ opérations au pire.

(ii) Si M est représenté par un tableau de n booléens, l'opération *choisir-min* nécessite $n - 1$ comparaisons, soit pour la boucle **while** au plus $(n - 1) \times (n - 1)$ en tout, c'est-à-dire, encore $\Theta(n^2)$ opérations au pire.

Dans le premier cas, la suppression de m revient à supprimer un pointeur, et dans le second à changer un booléen. Il y a au pire $\Theta(n)$ opérations de ce genre.

• Evaluons globalement le nombre de comparaisons, affectations, additions et mises à jour des tableaux d et pr, dues au réajustement des valeurs de d et pr, pour une exécution de l'algorithme. Pour chaque sommet m de M, chacun de ces nombres est majoré par $d^{o+}(m)$. Le nombre total de chacune de ces opérations est donc majoré par le nombre d'arcs p, dans le cas d'une représentation du graphe par listes d'adjacence. Si le graphe est représenté par matrice d'adjacence, cette complexité est en $\Theta(n^2)$.

Comme $p \leq n^2$, la complexité totale de la procédure, dans le pire des cas, est en $\Theta(n^2)$.

Dans le cas de *graphes denses*, où p est assez voisin de n^2, cette complexité est raisonnable. Mais si n^2 est beaucoup plus grand que p, il est légitime de se demander si on peut faire mieux. En fait, on peut dans ce cas améliorer la complexité de l'algorithme en représentant l'ensemble des sommets i de M par un tas ordonné par les valeurs de $d[i]$ (cf. chapitre 15) et le graphe par des listes d'adjacence.

Avec cette représentation, l'opération *choisir-min* est en $\Theta(1)$ et la suppression de m est en $\Theta(\log|M|)$ pour le nombre de comparaisons. Pour toute la boucle **while**, ces opérations impliquent alors au pire $\Theta(n.\log n)$ comparaisons.

Comme on modifie à la fois le cardinal de M et les valeurs associées aux éléments de M, il faut maintenir la structure de tas lorsqu'on réajuste ces valeurs (c'est-à-dire les tableaux d et pr de la procédure Dijkstra). On a vu qu'il y a au pire p réajustements pour une exécution de l'algorithme, et chaque réajustement consiste à supprimer du tas un élément y et son ancienne valeur $d[y]$ et à ajouter au tas cet élément, avec sa nouvelle valeur. Une telle suppression dans un tas de taille i nécessite au pire $\Theta(\log i)$ comparaisons. Il en est de même de l'adjonction. Le nombre total de comparaisons nécessaires, si on utilise un tas, est donc au pire en $O(p.\log n)$. (Il s'agit de la complexité en temps; il faudrait aussi évaluer la complexité en espace dans ce cas.) Par ailleurs, on a toujours $\Theta(p)$ opérations d'additions, de mises à jour des tableaux d et pr, etc.

Avec ces améliorations, si le graphe est représenté par des listes d'adjacence de successeurs, *la complexité au pire de l'algorithme est en* $O(\max(p,n).(\log n))$. Lorsque $p \geq n$ on a une complexité en $O(p.\log n)$ qui est préférable à $\Theta(n^2)$ si $n^2 >> p$.

3. Algorithme de Bellman (graphe sans circuit, coûts quelconques)

Cet algorithme est valable pour des graphes sans circuit, valués par des coûts quelconques. Il donne un plus court chemin d'un sommet r à tous les autres sommets

ainsi que son coût, ***dans le cas où r est racine du graphe*** (une extension au cas où r n'est pas racine est proposée en exercice).

3.1. Algorithme

On note M l'ensemble des sommets x pour lesquels *ppdistance*(r, x) n'est pas encore connue. On ne calcule *ppdistance*(r, x) que lorsque tous les prédécesseurs de x sont dans $CC = S - M$. Il est donc nécessaire de connaître, pour tout sommet x, son demi-degré intérieur et son $i^{ième}$ prédécesseur.

L'algorithme repose sur l'idée suivante : un plus court chemin de r à x doit passer par l'un des prédécesseurs y de x, celui pour lequel *ppdistance*(r, y) + *coût*(y, x) est minimum. L'algorithme choisit donc dans M un sommet x tel que tous ses prédécesseurs sont dans CC. Le choix du sommet x est réalisé par l'opération *choisir-suivant* de profil :

choisir-suivant : Ensemble × Graphe → Sommet

Cette opération n'est pas définie pour l'ensemble vide.

Pour tout ensemble M non vide, pour tout graphe G et pour tout i tel que

$1 \leq i \leq d^{o-}$ *de* (*choisir-suivant*$(M,\ G)$) *dans* G, on a :

(*i ème-pred-de choisir-suivant*$(M,\ G)$ *dans* G) $\in M$ = faux
choisir-suivant$(M,\ G) \in M$ = vrai

La programmation de cette opération varie selon la représentation choisie pour le graphe. Dans tous les cas, elle nécessite le parcours de l'ensemble des prédécesseurs des sommets.

En utilisant cette opération, l'algorithme s'exprime de la manière suivante :

```
procedure Bellman(r : integer; G : Graphe; var d : array [1..n] of real;
                  var pr : array[1..n] of integer);
{Le graphe G a n sommets : 1,2,...,n et est valué par des coûts quelconques;
r est racine du graphe G; on recherche un plus court chemin de r à chacun des
autres sommets; G est supposé sans circuit}
var i, x, y, z : integer; min, aux : real; M : Ensemble;
{ x, y et z sont des sommets}

begin
    {initialisations}
    M := ensemble-vide;
    for i := 1 to n do M := ajouter(i, M);
    M := supprimer(r, M);
    d[r] := 0; pr[r] := r;
```

```
    {choix successifs d'un élément dans M}
    while M <> ensemble-vide do begin
        x := choisir-suivant(M, G);
        M := supprimer(x, M);
        {on cherche y, prédécesseur de x, tel que d[y] + coût(y, x, G) soit
        minimum :}
        y := 1 ème-pred-de x dans G;
        min := d[y] + coût(y, x, G);
        for i := 2 to d°− de x dans G do begin
            z := i ème-pred-de x dans G;
            aux := d[z] + coût(z, x, G);
            if aux < min then begin min := aux; y := z end
        end;
        d[x] := min; {d[x] = ppdistance(r, x)}
        pr[x] := y   {pr[x] = père(r, x)}
    end
end Bellman;
```

3.2. Exemple

On considère le graphe G sans circuit de la figure 5, qui a pour racine le sommet $s1$.

L'exécution de la procédure sur G se fait selon les étapes suivantes :

1) Initialisation : $M = \{s2, s3, s4, s5, s6\}$

d :	0					
	1	2	3	4	5	6

pr :	1					
	1	2	3	4	5	6

2) $M = \{s2, s4, s5, s6\}$

d :	0		−2			
	1	2	3	4	5	6

pr :	1		1			
	1	2	3	4	5	6

3) $M = \{s2, s4, s6\}$. A cette étape, on aurait pu choisir de supprimer le sommet $s2$, au lieu du sommet $s5$.

d :	0		−2		2	
	1	2	3	4	5	6

pr :	1		1		3	
	1	2	3	4	5	6

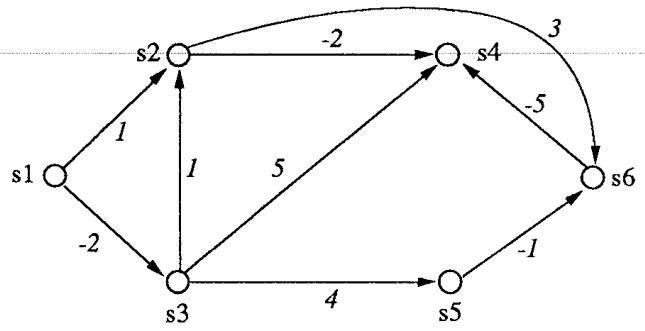

Figure 5. Graphe valué sans circuit de racine $s1$.

4) $M = \{s4, s6\}$

d :	0	−1	−2		2		pr :	1	3	1		3	
	1	2	3	4	5	6		1	2	3	4	5	6

5) $M = \{s4\}$

d :	0	−1	−2		2	1	pr :	1	3	1		3	5
	1	2	3	4	5	6		1	2	3	4	5	6

6) $M = \emptyset$

d :	0	−1	−2	−4	2	1	pr :	1	3	1	6	3	5
	1	2	3	4	5	6		1	2	3	4	5	6

3.3 Justification de l'algorithme

Pour tout sommet intermédiaire z sur un plus court chemin de r à y, la portion de ce chemin comprise entre r et z est un plus court chemin de r à z. Par suite, on obtient bien un plus court chemin de r à tout sommet de CC, ainsi que son coût. Comme à la fin de la procédure, M est vide, on aura montré que l'algorithme est correct, si on montre qu'à chaque itération de la boucle **while**, l'opération *choisir-suivant* est bien définie.

Supposons que M soit non vide et qu'on ne puisse pas trouver un sommet x dans M dont tous les prédécesseurs soient hors de M. Puisque r est racine du graphe, tout sommet autre que r admet un prédécesseur. Soit y_0 un sommet de M, y_0 est différent de r car M est initialisé à $S - \{r\}$; y_0 admet au moins un

prédécesseur y_1 dans M. On peut recommencer avec ce sommet y_1 et construire une suite, $y_0, y_1, \ldots, y_i, y_{i+1}, \ldots$ telle que y_i est élément de M et y_{i+1} est un prédécesseur de y_i qui est dans M. Comme le graphe est sans circuit, cette suite est infinie. On obtient une contradiction avec le fait que M est nécessairement fini. L'opération *choisir-suivant* est donc toujours possible.

3.4. Analyse de la complexité

On évalue la complexité de l'algorithme en nombre total d'opérations, en fonction du nombre n de sommets et du nombre p d'arcs du graphe.

Cette complexité dépend des représentations du graphe, de l'ensemble M et de l'opération *choisir-suivant.*

Plaçons-nous dans le cas, favorable pour la complexité en temps de cet algorithme, où le graphe est représenté par listes de successeurs et par listes de prédécesseurs. La représentation de l'ensemble M doit être choisie en tenant compte de la complexité de l'opération *choisir-suivant* et de celle des mises à jour. L'opération *choisir-suivant* nécessite de connaître, pour chaque sommet de M, le nombre de ses prédécesseurs qui sont dans M. Pour éviter de recalculer ce nombre de prédécesseurs pour chaque nouvelle valeur de l'ensemble M, on peut gérer un tableau des nombres de prédécesseurs, tableau qui doit être mis à jour après chaque suppression d'un élément de M. Dans ce cas, effectuer *choisir-suivant* revient à choisir dans ce tableau un indice de valeur nulle. Pour ne pas avoir à parcourir le tableau, on chaîne dans le tableau les indices de valeurs nulles, comme dans le tri topologique (cf. la procédure *tritopo2* au chapitre 18). L'opération *choisir-suivant* est alors en $\Theta(1)$.

On calcule la complexité de l'algorithme pour ces représentations, en supposant que le graphe est donné par listes de successeurs et listes de prédécesseurs.

L'initialisation de M est en $\Theta(n)$. Lors des $n-1$ exécutions de la boucle **while**, on fait en tout $\sum_{x=1,\ldots,n} d^{o+}(x) = p$ mises à jour du tableau des nombres de prédécesseurs, et pour la recherche de y et de *min*, de l'ordre de $\sum_{x=1,\ldots,n} d^{o-}(x) = p$ opérations.

De plus, à chaque itération, l'exécution de *choisir-suivant* et *supprimer* est en $\Theta(1)$. Si on connaît pour chaque sommet la liste de ses prédécesseurs et la liste de ses successeurs, alors on a une complexité totale de $\Theta(n+p)$ opérations. Comme $n-1 \leq p$ (G admet une racine r), la complexité totale est en $\Theta(p)$.

Si la représentation du graphe rend inefficace le parcours de la liste des successeurs ou de la liste des prédécesseurs d'un sommet, la complexité peut augmenter.

4. Algorithme de Floyd (coûts quelconques)

On s'intéresse maintenant au problème de la recherche d'un plus court chemin pour tout couple de sommets (x, y). On pourrait utiliser les algorithmes précédents en faisant varier la source s. En fait, il existe des algorithmes simples qui calculent directement tous les plus courts chemins en utilisant une représentation des graphes par matrice d'adjacence.

On donne ici une méthode matricielle, l'algorithme de Floyd, valable pour des graphes valués par des coûts quelconques (*dans le cas où il n'y a pas de circuit absorbant*). Une autre méthode matricielle (l'algorithme de Dantzig) est proposée en exercice.

4.1. Principe et algorithme

Le principe de l'algorithme est analogue à celui de l'algorithme de Warshall vu au chapitre 19.

Soit $S = \{1, 2, \ldots, n\}$ l'ensemble des sommets du graphe.

On considère successivement les chemins de i vers j qui ne passent d'abord par aucun autre sommet, puis qui passent éventuellement par le sommet 1, puis par les sommets 1 et 2, etc. A l'étape k, on calcule le coût $d_k(i, j)$ d'un plus court chemin de i à j passant par des sommets inférieurs ou égaux à k. On note $pred_k(i, j)$ le prédécesseur de j dans ce plus court chemin.

On utilise une matrice carrée A d'ordre n, initialisée aux valeurs de l'opération *coût* (on prolonge cette opération comme dans l'algorithme de Dijkstra, de manière à ce qu'elle soit définie pour tout couple de sommets). A chaque étape, cette matrice contient les coûts d_k des plus courts chemins calculés. On utilise également une matrice carrée P d'ordre n telle que $P[i, j] = père(i, j)$, s'il existe un chemin de i vers j. Au départ, $P[i, j] = i$. L'algorithme s'écrit de la manière suivante (en ignorant les problèmes d'additions avec l'infini) :

```
procedure Floyd(var A : array[1..n, 1..n] of real; G : Graphe;
                var P : array[1..n, 1..n] of integer);
{Le graphe G est valué par des coûts quelconques ; on suppose qu'il n'y a pas de circuit absorbant ; on cherche un plus court chemin entre deux sommets quelconques}
var i, j, k : integer;
begin
   {initialisation}
   for i := 1 to n do
      for j := 1 to n do begin
         A[i, j] := coût(i, j, G);
         P[i, j] := i
      end;
```

```
{calcul des plus courts chemins}
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            if A[i,k] + A[k,j] < A[i,j] then begin
                A[i,j] := A[i,k] + A[k,j];
                P[i,j] := P[k,j]
            end
{si A[i,j] < ∞, A[i,j] = ppdistance(i, j) et P[i,j] =père(i, j)}
end Floyd;
```

4.2. Exemple

L'exécution de la procédure sur le graphe de la figure 6 donne les matrices successives suivantes.

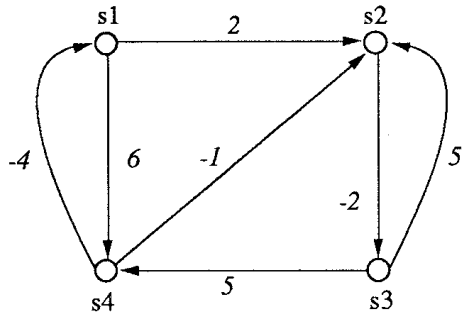

Figure 6. Graphe valué par des coûts quelconques, sans circuit absorbant.

1) Initialisation

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -1 & \infty & 0 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \qquad P = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

$k = 1$

$$A = \begin{bmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & \infty & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 \end{bmatrix}$$

$k = 2$

$$\boldsymbol{A} = \begin{bmatrix} 0 & 2 & 0 & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad \boldsymbol{P} = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

$k = 3$

$$\boldsymbol{A} = \begin{bmatrix} 0 & 2 & 0 & 5 \\ \infty & 0 & -2 & 3 \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad \boldsymbol{P} = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

$k = 4$

$$\boldsymbol{A} = \begin{bmatrix} 0 & 2 & 0 & 5 \\ -1 & 0 & -2 & 3 \\ 1 & 3 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad \boldsymbol{P} = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 4 & 2 & 2 & 3 \\ 4 & 1 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

4.3. Justification de l'algorithme

A l'étape k, on considère les chemins de i à j qui passent par des sommets inférieurs ou égaux à k. On peut éventuellement obtenir un chemin μ de i à j, plus court que le chemin λ obtenu à l'étape $k-1$, en prenant un plus court chemin de i à k, et un plus court chemin de k à j passant tous deux par des sommets inférieurs ou égaux à $k-1$ (voir figure 7). La valeur $d_k(i,j)$ est donc donnée par :

$$d_k(i,j) = \min(d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j))$$

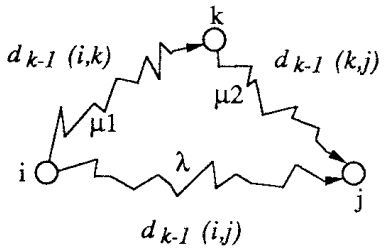

Figure 7. Plus courts chemins passant par des sommets inférieurs ou égaux à k.

On n'obtient un chemin μ de i vers j plus court que λ que si $d_{k-1}(i,k) + d_{k-1}(k,j) < d_{k-1}(i,j)$. Dans ce cas, ce chemin se termine par un plus court chemin $\mu 2$ de k à j calculé à l'étape $k-1$. Le prédécesseur de j dans ce nouveau chemin est donc le prédécesseur de j dans $\mu 2$: $pred_k(i,j) = pred_{k-1}(k,j)$.

Au bout de n étapes, $d_n(i,j) = ppdistance(i,j)$ et $pred_n(i,j) = père(i,j)$.

Comme pour l'algorithme de Warshall, on peut n'utiliser qu'une seule matrice A pour le calcul des $d_k(i,j)$ car entre l'étape $k-1$ et l'étape k les valeurs de $d_{k-1}(i,k)$ et de $d_{k-1}(k,j)$ qui interviennent dans le calcul des $d_k(i,j)$, ne changent pas.

Obtention des plus courts chemins

Pour obtenir un plus court chemin de i à j, il suffit d'utiliser la ligne numéro i de la matrice P. Par exemple, si on veut obtenir le plus court chemin μ de $s4$ à $s3$, dans le graphe de la figure 6, on consulte la matrice P ainsi : $P[4,3] = 2 : s2$ est donc le prédécesseur de $s3$ dans μ; $P[4,2] = 1 : s1$ est le prédécesseur de $s2$ dans μ; $P[4,1] = 4 : s4$ est le prédécesseur de $s1$ dans μ. Finalement, le chemin μ s'écrit : $s4 \rightarrow s1 \rightarrow s2 \rightarrow s3$.

On peut obtenir les plus courts chemins d'une autre façon. A la $k^{ième}$ itération, si $A[i,k] + A[k,j] < A[i,j]$, on indique que le sommet k figure sur un plus court chemin de i à j passant par des sommets inférieurs ou égaux à k. Dans ce cas, on initialise la matrice P partout à 0 et on remplace l'instruction $P[i,j] := P[k,j]$ par l'instruction $P[i,j] := k$. Avec ces deux méthodes, l'impression d'un plus court chemin peut se faire de façon récursive (cf. exercices).

4.4. Analyse de la complexité

Il est clair que le nombre d'opérations (additions, comparaisons, affectations) effectuées par l'algorithme de Floyd est en $\Theta(n^3)$.

Dans le cas où les coûts sont tous positifs ou nuls, on aurait pu aussi utiliser l'algorithme de Dijkstra n fois. Avec une représentation par matrice d'adjacence, et sans utiliser de tas, l'algorithme de Dijkstra est en $\Theta(n^2)$ dans le pire des cas : on obtient ainsi une méthode qui est encore en $\Theta(n^3)$ pour la recherche des plus courts chemins pour tous les couples de sommets.

Si le graphe est peu dense et s'il est représenté par des listes de successeurs, l'utilisation d'un tas permet d'avoir une complexité en $O(p.\log n)$ pour l'algorithme de Dijkstra. Sa répétition n fois est alors en $O(n.p.\log n)$, ce qui est plus intéressant que l'algorithme de Floyd quand $p << n^2$.