Graphes pondérés

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Graphes pondérés

Graphes pondérés

Dans ce chapitre, on considère des graphes où les arêtes sont munies d'un **poids**. Les applications sont nombreuses.

- Dans le plan euclidien, si on considère un nuage de points, on peut considérer le graphe complet sur ce nuage : les arêtes ont alors pour poids la distance entre ces sommets.
- Une variante s'applique aux problématiques de transport : on a un réseau de routes (des arcs), dont le poids est la distance entre les villes. Les problèmes de plus courts chemins apparaissent naturellement dans ce contexte.
- Un poids sur une arête/un arc peut modéliser une capacité de flux, . . .

Graphes pondérés

On a vu dans le précédent chapitre comment calculer la distance $\delta(s,t)$ entre un sommet source s et un sommet quelconque t dans un **graphe non pondéré**, à l'aide d'un parcours en largeur.

Le but de ce chapitre est de généraliser cet algorithme au cas des **graphes pondérés**.

Définition et représentations des graphes pondérés

Graphes pondérés

Définition (pondération)

On considère une graphe $G = (S, A)$ orienté ou non. Une **fonction de pondération** de G est une fonction $\omega : A \to \mathbb{R}$.
Le réel $\omega(a)$ est appelé le **poids** de l'arête ou de l'arc a.
Le graphe $G = (S, A, \omega)$ est appelé un **graphe pondéré**.

Remarque

On étend naturellement la fonction de pondération à tout couple de sommets pour obtenir une fonction $S^2 \to \mathbb{R} \cup \{+\infty\}$, avec la définition suivante :

$$\omega'(s,t) = \begin{cases} \omega((s,t)) & \text{si } (s,t) \in A \\ 0 & \text{si } s = t \\ +\infty & \text{sinon} \end{cases}$$

Graphes pondérés

```
type graphe_pondere_creux = (int * int) list array ;;

type zbar = Z of int | Inf ;;
type graphe_pondere_dense = zbar array array ;;
```

Implémentation

En pratique, on se ramène très souvent au cas où les poids sont des entiers. L'implémentation en OCaml est assez immédiate :

- dans le cas d'une implémentation creuse, il suffit dans la liste d'adjacence d'un sommet s de stocker des couples (t, p) à la place du seul sommet t : ceci signifie que $(s, t) \in A$ et $\omega(s, t) = p$;
- dans le cas d'une implémentation dense, on utilise la généralisation de la fonction ω à tout couple de sommets : la matrice d'adjacence est maintenant une matrice à valeurs dans $\mathbb{R} \cup +\{\infty\}$, ou $\mathbb{Z} \cup +\{\infty\}$ si les poids sont des entiers.

Type option en OCaml

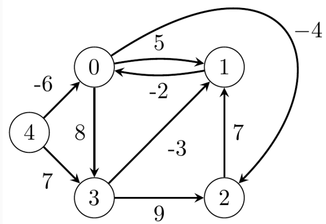
```
1    type 'a option = None | Some of 'a ;;
```

Type option

En fait, on pourrait utiliser le type `'a option` déjà défini en OCaml, qui est définit comme ci-dessus.

La constante `None` servirait alors à représenter $+\infty$, i.e. l'absence d'arête/arc.

Graphes pondérés



$$\begin{pmatrix} 0 & 5 & -4 & 8 & +\infty \\ -2 & 0 & +\infty & +\infty & +\infty \\ +\infty & 7 & 0 & +\infty & +\infty \\ +\infty & -3 & 9 & 0 & +\infty \\ -6 & +\infty & +\infty & 7 & 0 \end{pmatrix}$$

Exemple

Voici un exemple de **graphe pondéré**, et sa **matrice d'adjacence**.

Graphes pondérés

Attention

Attention à ne pas confondre la matrice d'adjacence dans les graphes pondérés et non pondérés : les zéros dans la matrice d'adjacence d'un graphe non pondéré deviennent des $+\infty$ dans la matrice d'un graphe pondéré, sauf la diagonale où il reste des zéros.

Définitions et premières propriétés sur les plus courts chemins

Définitions

Définition (poids)

La notion de **poids** d'un arc s'étend au **poids** d'un chemin : pour $c = s_0, s_1, \ldots, s_n$ un chemin dans un graphe, on définit son **poids** comme :

$$\omega(p) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1}) \in \mathbb{R}$$

Définition (distance)

Pour s et t deux sommets dans le graphe, on définit la **distance** de s à t, notée $\delta(s,t)$ comme suit :

$$\delta(s,t) = \inf\{\omega(p) \mid p \text{ est un chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{+\infty\}$$

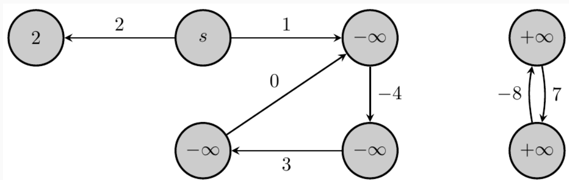
Distance

Remarque

Pour s et t deux sommets du graphe, $\delta(s,t)$ peut prendre les valeurs $+\infty$ et $-\infty$!

- Si t n'est pas accessible depuis s, il n'existe aucun chemin de s à t. L'ensemble définissant $\delta(s,t)$ étant vide, on a bien $\delta(s,t) = \inf = +\infty$.
- Si t est un sommet accessible depuis s, et si sur un chemin de s à t il existe un circuit de poids strictement négatif, le poids d'un chemin de s à t n'est pas borné inférieurement, car on peut boucler autant de fois qu'on veut dans ce circuit. Ainsi, $\delta(s,t) = -\infty$.

Distance



Exemple

Le graphe ci-dessus contient des poids négatifs : on a fait figurer les distances à la source s.

Trois sommets sont accessibles et situés sur un circuit de poids négatif. Les sommets de droite sont également sur un tel circuit, mais non accessibles.

Définitions

Définition (plus court chemin)

Pour s et t deux sommets tels que $\delta(s,t) \notin \{\pm\infty\}$, on appelle **plus court chemin** de s à t un chemin de poids $\delta(s,t)$ entre s et t.

Différents problèmes de plus court chemins

Remarque

Lors des calculs des plus courts chemins, nos algorithmes donneront une réponse erronée pour les sommets situés sur des circuits (accessibles) de poids strictement négatifs. On supposera donc qu'il n'y en a pas, même si en pratique il est facile de les détecter.

Différents problèmes de plus court chemins

Variantes

Dans le problème du calcul de plus courts chemins, on peut énumérer 4 cas :

1. calcul d'un plus court chemin depuis une source s et une destination t (toutes les deux fixées) ;
2. calcul de plus courts chemins depuis une origine unique s fixée ;
3. calcul de plus courts chemins vers une destination unique s fixée ;
4. calcul de plus courts chemins entre deux couples quelconques de sommets s et t.

Différents problèmes de plus court chemins

Variantes

Les problèmes (2) et (3) sont symétriques (il suffit de travailler sur le graphe transposé).

On verra dans la suite des algorithmes pour résoudre les problèmes (2) et (4).

Pour le problème (1), on se contentera du problème (2) : même s'il est *a priori* plus facile, on ne connaît pas à l'heure actuelle d'algorithme de résolution de (1) plus efficace asymptotiquement qu'un algorithme qui résout (2).

Optimalité des solutions aux sous-problèmes

Problème d'optimisation

Le problème du calcul d'un plus court chemin entre deux sommets est un **problème d'optimisation**.

Mais calculer un plus court chemin entre deux sommets fournit une solution à des sous-problèmes.

Proposition (Optimalité des sous-chemins)

Soit $s \overset{c}{\rightsquigarrow} t$ un **plus court chemin** passant par un sommet u.

Alors les deux morceaux de c $s \overset{c_1}{\rightsquigarrow} u$ et $u \overset{c_2}{\rightsquigarrow} t$ sont les plus courts chemins de s à u et de u à t.

Optimalité des solutions aux sous-problèmes

Proposition (Optimalité des sous-chemins)

Soit $s \overset{c}{\rightsquigarrow} t$ un **plus court chemin** passant par un sommet u.

Alors les deux morceaux de c $s \overset{c_1}{\rightsquigarrow} u$ et $u \overset{c_2}{\rightsquigarrow} t$ sont les plus courts chemins de s à u et de u à t.

Preuve

Par l'absurde, s'il existait un chemin $s \overset{c_1'}{\rightsquigarrow} u$ plus court que c_1, alors on aurait un chemin $s \overset{c_1'}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} t$ de poids $\omega(c_1') + \omega(c_2) < \omega(c_1) + \omega(c_2) = \omega(c)$, ce qui est absurde.

De même pour le chemin c_2.

Optimalité des solutions aux sous-problèmes

Sous-problèmes

La propriété précédente montre que dans le problème du calcul de plus courts chemins réside une **optimalité des solutions aux sous-problèmes** : déterminer un plus court chemin de s à t fournit en même temps des plus courts chemins depuis s (ou vers t) pour tous les sommets situés sur le chemin.

C'est une caractéristique des problèmes pour lesquels les méthodes de **programmation dynamique** peuvent s'appliquer.

Optimalité des solutions aux sous-problèmes

Programmation dynamique

Parmi les algorithmes que l'on va voir, trois d'entre eux font usage de **programmation dynamique**, le dernier étant un **algorithme glouton** : un algorithme pour lequel il y a un choix **localement optimal** qui est en fait **globalement optimal**, et évite donc le recours à la programmation dynamique (plus lourde).

Optimalité des solutions aux sous-problèmes

Programmation dynamique

Comme souvent en programmation dynamique, on commence par calculer les valeurs optimales (ici le s$\delta(s,t)$) avant de modifier l'algorithme pour calculer les solutions optimales (pour tout (s,t), un chemin $s \overset{c}{\rightsquigarrow} t$ tel que $\omega(c) = \delta(s,t)$).

On va donc se concentrer sur le calcul des $\delta(s,t)$.

Plus courts chemins à origine unique

Plus courts chemins à origine unique

Origine unique

Le but de cette section est de calculer les distances entre une **origine** s **fixée** et tous les sommets du graphe, donc $(\delta(s,t))_{t\in S}$.

On va voir deux algorithmes : l'algorithme de **Bellman-Ford** et l'algorithme de **Dijkstra**.

Le premier a une moins bonne complexité, mais a le mérite d'être très simple et de s'appliquer même lorsqu'il y a des arcs de poids strictement négatifs, alors que le second nécessite que tous les arcs soient de poids positif.

Les deux fonctionnent sur le même principe : il s'agit de **relâcher** les arcs du graphe, dans un certain ordre.

Plus courts chemins à origine unique

Hypothèse

Dans la suite, on suppose que le graphe ne possède pas de **circuit** de poids **strictement négatif**.

Relâchement d'arcs

Lemme (Inégalité triangulaire)

Soit s, t, t u trois sommets d'un graphe pondéré G.

Alors $\delta(s,t) \leq \delta(s,u) + \omega(u,t)$.

Preuve

- S'il existe un chemin $s \rightsquigarrow u$ et un arc $u \to t$, on a $\delta(s,u) < +\infty$ et $\omega(u,t) < +\infty$. On obtient un chemin $s \rightsquigarrow t$ de poids $\delta(s,u) + \omega(u,t)$ en prenant un plus petit chemin $s \rightsquigarrow u$ et l'arc $u \to t$, donc $\delta(s,t) \leq \delta(s,u) + \omega(u,t)$.
- Sinon, on a $\delta(s,u) + \omega(u,t) = +\infty$, et l'inégalité reste valable.

Relâchement d'arcs

Lemme ($\star$)

Soit t un sommet accessible depuis s.

Alors il existe un chemin de poids $\delta(s,t)$ entre s et t, composé de sommets tous distincts.

Relâchement d'arcs

Preuve

Considérons un chemin $s \overset{c}{\rightsquigarrow} t$ de poids $\delta(s,t)$, et de longueur **minimale** parmi les chemins de poids $\delta(s,t)$ reliant s à t.
S'il existait deux sommets égaux sur le chemin, on obtiendrait un circuit.
Comme il n'y a pas de circuit de poids strictement négatif dans le graphe par hypothèse, ce circuit est de poids nul (sinon on pourrait le supprimer pour obtenir un chemin de s à t de poids strictement inférieur à $\delta(s,t)$, ce qui est impossible).
Mais le supprimer mène alors à un chemin de même poids mais avec strictement moins d'arcs : contradiction avec la minimalité de la longueur de c.
Donc c est composé de sommets distincts.

Relâchement d’arcs

Définition (Relâchement d’arc)

Supposons que le tableau $(d_s[t])_{t \in S}$ soit une estimation des distances $\delta(s,t)$ (i.e. $d_s[t] \geq \delta(s,t)$ pour tout t).

Relâcher l’arc (u,v) consiste à réaliser l’affectation :

$$d_s[v] \leftarrow \min(d_s[v], d_s[u] + \omega(u,v))$$

Lemme

On a encore $d_s[v] \geq \delta(s,v)$ après le relâchement de l’arc (u,v).

Preuve

Avant relâchement, on a $\delta(s,u) \leq d_s[u]$ par hypothèse.
Ainsi, par **inégalité triangulaire** : $\delta(s,v) \leq d_s[u] + \omega(u,v)$.

Relâchement d'arcs

Proposition

Soit t un sommet accessible depuis s, et c un chemin $(s = s_0, s_1, \ldots, s_k = t)$ de poids $\delta(s,t)$ entre s et t.

En partant du tableau $(d_s[u])_{u \in S}$ quelconque tel que $d_s[u] \geq \delta(s,u)$ pour tout u, avec $d_s[s] = 0$, si on relâche successivement les arcs $(s_0, s_1), \ldots, (s_{k-1}, s_k)$ dans cet ordre, alors $\delta_s[t]$ contient $\delta(s,t)$ à la fin du processus.

Relâchement d'arcs

Preuve

La propriété d'optimalité des sous-chemins montre que $\forall i \in \{0, \ldots, k\}$, $(s_0, s_1, \ldots, s_i)$ est un plus court chemin de s à s_i. Montrons par récurrence sur i qu'après relâchement de l'arc (s_{i-1}, s_i), $d_s[s_{i-1}]$ contient $\delta(s, s_{i-1})$.

- $i = 0$: $d_s[s] = 0 = \delta(s, s)$: OK.
- Soit $i > 1$, et supposons la propriété démontrée au rang $i-1$. Alors juste avant le relâchement de (s_{i-1}, s_i), $d_s[s_{i-1}]$ contient $\delta(s, s_{i-1})$.
  Comme $\delta(s, s_i) = \delta(s, s_{i-1}) + \omega(s_{i-1}, s_i)$, on a $d_s[s_i] \leq \delta(s, s_i)$ après relâchement de l'arc (s_{i-1}, s_i).
  Or, le lemme précédent prouve qu'on avait $d_s[s_i] \geq \delta(s, s_i)$ avant relâchement. Donc la propriété est vraie au rang i.

Relâchement d'arcs

Preuve

Ainsi, par principe de récurrence, la propriété est vraie $\forall i \in \{0, \ldots, k\}$, donc en particulier $d_s[t]$ contient $\delta(s,t)$ à la fin du processus.

Remarque

La propriété précédente reste vraie si on relâche d'autres arcs entre les relâchements des (s_i, s_{i+1}) : en effet, relâcher un arc ne peut que faire baisser des $d_s[u]$, mais le lemme précédent assure que l'on de descendra pas en dessous de $\delta(s,u)$.

Algorithme de Bellman-Ford

Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est directement inspiré de la proposition précédente et du lemme $(\star)$: avec $n = |S|$, il suffit de relâcher $n-1$ fois tous les arcs du graphe $G = (S, A, \omega)$ pour calculer les $\delta(s, t)$.

Algorithme de Bellman-Ford

Algorithme 1 : Algorithme de Bellman-Ford

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par listes d'adjacence, un sommet s

Résultat : Les distances $\delta(s, t)$ pour tout $t \in S$

$d_s[t] \leftarrow +\infty$ pour tout $t \in S$;
$d_s[s] \leftarrow 0$;
$n \leftarrow |S|$;
pour i *entre* 1 *et* $n - 1$ **faire**
    pour j *entre* 0 *et* $n - 1$ **faire**
        pour *tout voisin* u *de* j **faire**
            $d_s[u] \leftarrow \min(d_s[u], d_s[j] + \omega(j, u))$;
retourner d_s ;

Algorithme de Bellman-Ford

Correction

Montrons que l'algorithme de Bellman-Ford est correct si le graphe n'a pas de circuit de poids strictement négatif. Soit t un sommet quelconque du graphe, accessible depuis s.

- Comme on ne fait que relâcher des arcs, le lemme sur les relâchements d'arcs implique $d_s[t] \geq \delta(s,t)$ à la fin de l'algorithme.
- Le lemme $(\star)$ assure l'existence d'un plus court chemin $s = s_0, s_1, \ldots, s_k = t$ où tous les sommets sont distincts : on a donc $k \leq n-1$.

Algorithme de Bellman-Ford

Correction

- Ainsi, l'arc (s_0, s_1) est relâché lors du tour 1 de la boucle principale, l'arc (s_1, s_2) est relâché lors du tour 2, et de même jusqu'à l'arc (s_{k-1}, s_k) lors du tour k.
- Puisque les arcs sont relâchés dans l'ordre du chemin, la proposition précédente assure $d_s[t] = \delta(s,t)$ à la fin de l'algorithme.

De plus, on vérifie facilement que la propriété "$d_s[t] = +\infty$ si t n'est pas accessible depuis s" est un invariant des boucles de l'algorithme.

Ainsi, à la fin de l'algorithme, on a bien $d_s[t] = \delta(s,t)$ pour tout $t \in S$.

Algorithme de Bellman-Ford

Complexité

L'algorithme de Bellman-Ford est de complexité $\mathcal{O}(n(a+n))$, puisqu'il relâche $n-1$ fois tous les arcs de G, et un relâchement de tous les arcs nécessitant de parcourir toutes les listes d'adjacence, il a un coût $\mathcal{O}(a+n)$.

Algorithme de Bellman-Ford

Détection des circuits de poids strictement négatif

Il est en fait facile de tester l'existence d'un circuit de poids strictement négatif accessible depuis s.

Proposition

Dans l'algorithme de Bellman-Ford, effectuons un relâchement supplémentaire de tous les arcs (boucle principale de 1 à n).

Alors le tableau d_s est modifié pendant le dernier tour de boucle si et seulement s'il existe un circuit de poids strictement négatif accessible depuis s.

Preuve

Dans le cas où il n'y a pas de tel circuit, on a vu qu'après les $n-1$ tours de la boucle principale, $d_s[t] = \delta(s,t)$ pour tout sommet t. Un relâchement d'arc ne peut donc diminuer aucun $d_s[t]$.

Supposons maintenant qu'aucun $d_s[u]$ ne soit modifié lors du dernier tour de boucle.
Cela signifie qu'avant le dernier tour de boucle, on a $d_s[u] \leq d_s[j] + \omega(j,u)$ pour tout arc (j,u).

Preuve

Considérons un circuit $s_0, \ldots, s_k = s_0$ accessible depuis s. Alors, avant le dernier tour de boucle :
$d_s[s_{i+1}] \leq d_s[s_i] + \omega(s_i, s_{i+1})$, pour tout $i \in [\![0, k-1]\!]$.

On a de plus $d_s[s_i] < +\infty$ pour tout $i \in [\![0, k-1]\!]$, car chaque s_i est accessible, donc $d_s[s_{i+1}] - d_s[s_i] \leq \omega(s_i, s_{i+1})$. Ainsi, en somment ces inégalités, on obtient (comme $s_0 = s_k$) :

$$0 = \sum_{i=0}^{k-1} (d_s[s_{i+1}] - d_s[s_i]) \leq \sum_{i=0}^{k-1} \omega(s_i, s_{i+1})$$

Ce circuit n'est donc pas de poids strictement négatif.

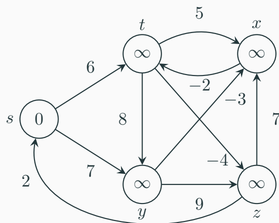
Algorithme de Bellman-Ford

Calcul effectif des plus courts chemins

Pour calculer effectivement des plus courts chemins depuis s, il suffit comme dans le parcours en largeur d'un graphe non pondéré d'ajouter un tableau de prédécesseurs pred : dans la boucle interne de l'algorithme, si $d_s[u] > d_s[j] + \omega(j, u)$, alors $d_s[u]$ prend la valeur $d_s[j]+\omega(j, u)$, et $\mathrm{pred}[u]$ prend la valeur j.

À la fin de l'algorithme, il suffit de remonter depuis un sommet accessible vers s pour obtenir un plus court chemin, à l'envers.

Algorithme de Bellman-Ford



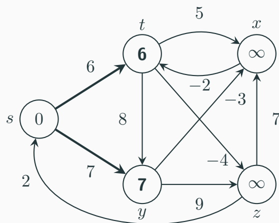
Exemple

Le graphe ci-dessus est constitué de 5 sommets : la source s, ainsi que t, x, y, z ; chaque arc doit donc être relâché 4 fois. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté (s,t) et (s,y) que l'on relâche à la fin :

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Les arcs en **gras** représentent le tableau des prédécesseurs.

Algorithme de Bellman-Ford



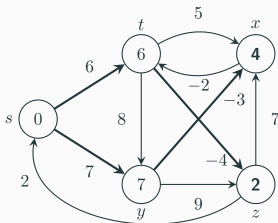
Exemple

Le graphe ci-dessus est constitué de 5 sommets : la source s, ainsi que t, x, y, z ; chaque arc doit donc être relâché 4 fois. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté (s,t) et (s,y) que l'on relâche à la fin :

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Les arcs en **gras** représentent le tableau des prédécesseurs.

Algorithme de Bellman-Ford



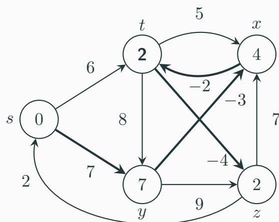
Exemple

Le graphe ci-dessus est constitué de 5 sommets : la source s, ainsi que t, x, y, z ; chaque arc doit donc être relâché 4 fois. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté (s,t) et (s,y) que l'on relâche à la fin :

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Les arcs en **gras** représentent le tableau des prédécesseurs.

Algorithme de Bellman-Ford



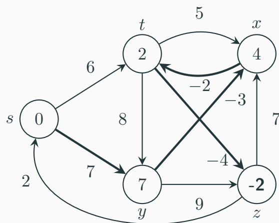
Exemple

Le graphe ci-dessus est constitué de 5 sommets : la source s, ainsi que t, x, y, z ; chaque arc doit donc être relâché 4 fois. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté (s,t) et (s,y) que l'on relâche à la fin :

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Les arcs en **gras** représentent le tableau des prédécesseurs.

Algorithme de Bellman-Ford



Exemple

Le graphe ci-dessus est constitué de 5 sommets : la source s, ainsi que t, x, y, z ; chaque arc doit donc être relâché 4 fois. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté (s,t) et (s,y) que l'on relâche à la fin :

$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Les arcs en **gras** représentent le tableau des prédécesseurs.

Algorithme de Dijkstra

Dijkstra

L'algorithme précédent a une complexité très élevée comparée au parcours en largeur du chapitre précédent, qui s'effectuait en temps linéaire $\mathcal{O}(a + n)$.

L'algorithme de **Dijkstra** que l'on va voir est une généralisation du parcours en largeur : il consiste également à traiter les sommets un par un, par distance à l'origine croissante.

Pour fonctionner, l'algorithme a besoin d'une hypothèse supplémentaire :

On suppose pour l'algorithme de Dijkstra que **tous les arcs** ont un **poids positif**.

Algorithme de Dijkstra

Algorithme 2 : Algorithme de Dijkstra

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par listes d'adjacence avec $\omega(A) \subset \mathbb{R}_+$, un sommet s

Résultat : Les distances $\delta(s, t)$ pour tout $t \in S$

$d_s[t] \leftarrow +\infty$ pour tout $t \in S$;
$d_s[s] \leftarrow 0$;
$H \leftarrow \emptyset$;
$F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
    $u \leftarrow$ Retirer de F un sommet v vérifiant $d_s[v]$ minimal ;
    pour *tout voisin* v *de* u **faire**
        si $v \notin F$ *et* $v \notin H$ **alors**
            Ajouter v à F ;
        $d_s[v] \leftarrow \min(d_s[v], d_s[u] + \omega(u, v))$;
    Ajouter u à H ;
retourner d_s ;

Terminaison

L'algorithme fait un parcours du graphe depuis le sommet s : on peut retrouver le parcours générique du chapitre précédent en supprimant ce qui concerne le tableau d_s, et en remplaçant H par l'ensemble des sommets déjà visités.

Ainsi, l'algorithme de Dijkstra **termine**.

Correction

Faisons déjà plusieurs observations :

- l'algorithme, pour travailler sur le tableau d_s, se contente de relâcher des arcs : on aura donc $d_s[t] \geq \delta(s,t)$ pour tout sommet t à la fin de l'algorithme ;
- l'algorithme faisant notamment un parcours générique, à la fin de l'algorithme, tous les sommets t accessibles depuis s sont dans l'ensemble H, et vérifient tous $d_s[t] < +\infty$.

Montrons maintenant que l'algorithme est **correct**.

Algorithme de Dijkstra

Proposition (correction)

Dans l'algorithme de **Dijkstra**, la propriété suivante est un invariant de boucle :

$$\forall t \in H,\ d_s[t] = \delta(s,t)$$

Preuve

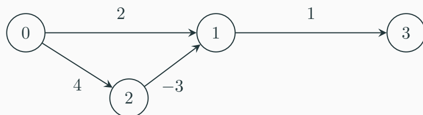
- La propriété est vraie avant la boucle, car $H = \emptyset$.
- Pour montrer l'hérédité, il suffit de montrer qu'un sommet $u \in F$ vérifiant $d_s[u]$ minimal vérifie en fait $d_s[u] = \delta(s,u)$. Distinguons deux cas :
  - si $u = s$ (c'est le premier tour de boucle), on a $d_s[s] = \delta(s,s) = 0$: OK ;

Algorithme de Dijkstra

Preuve

- sinon, considérons un plus court chemin $s \rightsquigarrow u$.
  Considérons sur ce chemin le premier sommet $y \notin H$ (qui existe bien, car $s \in H$ et $u \notin H$), et x son prédécesseur.
  On a donc $s \rightsquigarrow x \to y \rightsquigarrow u$.
  Puisque $x \in H$, on a $\delta(s,x) = d_s[x]$ (HR), et lorsqu'on a traité x, on a rajouté $y \in F$ et relâché l'arc $x \to y$: ainsi $d_s[y] \leq \delta(s,x) + \omega(x,y) = \delta(s,y)$.
  Le morceau $s \rightsquigarrow y$ étant un plus court chemin et les poids étant **positifs**, on a $\delta(s,y) \leq \delta(s,u)$. Puisque $d_s[u]$ est minimal parmi les éléments de F, on a $d_s[u] \leq d_s[y]$.
  Ainsi : $d_s[u] \leq d_s[y] \leq \delta(s,y) \leq \delta(s,u) \leq d_s[u]$.
  Donc $d_s[u] = \delta(s,u)$.

Algorithme de Dijkstra



Remarque

La preuve précédente ne fonctionne pas si les arcs ne sont pas tous de poids **positifs**.

L'exemple ci-dessus montre un graphe pour lequel l'algorithme de Dijkstra ne fonctionne pas (avec $s = 0$) : l'algorithme va relâcher successivement les arcs $(0, 1)$, $(0, 2)$, $(1, 3)$, $(2, 1)$.

Mais il faudrait relâcher à nouveau l'arc $(1, 3)$ pour que le tableau des distances soit correct.

Complexité

La complexité de l'algorithme de Dijkstra dépend de la structure de données utilisée pour gérer l'ensemble F.

- Si on utilise simplement un tableau de booléens pour marquer les éléments de F, retirer l'élément $d[u]$ minimal a un coût $\mathcal{O}(n)$. Cette action est effectuée au plus n fois, pour un coût total en $\mathcal{O}(n^2)$.
  À part cela, la complexité de l'algorithme est la même que celle du parcours en largeur classique : on parcourt au plus une fois toutes les listes d'adjacence, pour un coût total $\mathcal{O}(a+n) = \mathcal{O}(n^2)$ car $a = \mathcal{O}(n^2)$.
  Ainsi, avec une telle implémentation, l'algorithme de **Dijkstra** a une complexité en $\mathcal{O}(n^2)$.

Complexité

- Si on utilise une **file de priorité min** pour gérer F, implémentée avec un **tas-min**, chaque opération sur F a une complexité en $\mathcal{O}(\log n)$. On fera une telle opération :
  - lorsqu'on retire le minimum de F (donc au plus une fois par sommet) ;
  - lorsqu'on diminue $d_s[v]$ (donc au plus une fois par arc).

  Ainsi, la complexité totale est en $\mathcal{O}((a+n)\log n)$.

Attention

Il faut avoir implémenté l'opération de diminution de la clé d'un élément quelconque de la file.

Ici, avec les sommets supposés être dans $[\![0, n-1]\!]$, il suffit d'utiliser un tableau pos dans lequel est stockée la position de chaque sommet i dans le tableau associé au tas.

On accède ainsi facilement à la position de chaque sommet, qu'on peut diminuer en temps $\mathcal{O}(\log n)$, en répercutant les permutations effectuées dans le tableau pos.

Algorithme de Dijkstra

Remarque

L'implémentation à choisir dépend du graphe :

- si le graphe a beaucoup d'arêtes (a proche de n^2), il vaut mieux utiliser un tableau de booléens (graphe "dense") ;
- si le graphe a peu d'arêtes ($a = o(n^2/\log n)$), on aura intérêt à utiliser une file de priorité (graphe "creux").

Remarque

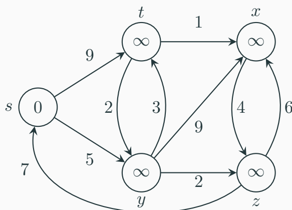
Une implémentation de la structure de **file de priorité min** où les opérations de diminution de clé ont une **complexité amortie constante** existe (et a été en fait inventée historiquement pour l'algorithme de Dijkstra) : les tas de Fibonacci.

Avec cette implémentation, la complexité se réduit à $\mathcal{O}(a + n \log n)$.

Calcul effectif des plus courts chemins

L'adaptation est la même que pour l'algorithme de **Bellman-Ford** : il suffit d'utiliser un tableau des **prédécesseurs**.

Algorithme de Dijkstra



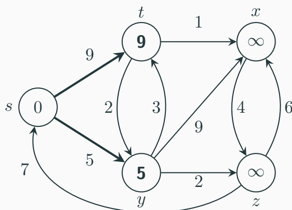
$$H = \emptyset,\ F = \{s\}$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Algorithme de Dijkstra



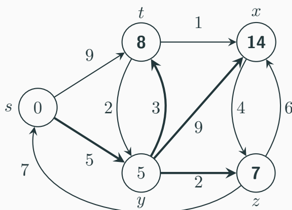
$$H = \{s\},\ F = \{t, y\}$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Algorithme de Dijkstra



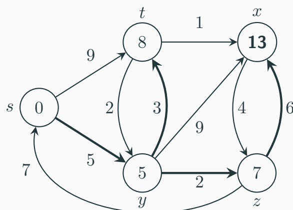
$$H = \{s, y\},\ F = \{t, x, z\}$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Algorithme de Dijkstra



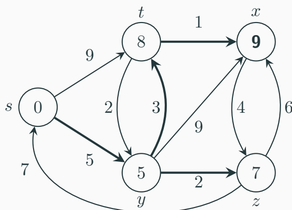
$$H = \{s, y, z\},\ F = \{t, x\}$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Algorithme de Dijkstra



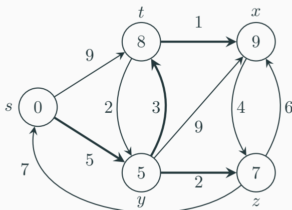
$$H = \{s, y, z, t\},\ F = \{x\}$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Algorithme de Dijkstra



$$H = \{s, y, z, t, x\},\ F = \emptyset$$

Exemple

Voici le déroulement de l'algorithme de **Dijkstra** sur un graphe à 5 sommets, dont la source est s.

Les arcs en gras représentent l'évolution du **tableau des prédécesseurs**.

Plus courts chemins pour tous couples de sommets

Plus courts chemins pour tous couples de sommets

Problème

On souhaite maintenant calculer $\delta(s,t)$ pour tout $(s,t) \in S^2$.

Avec un graphe représenté par **listes d'adjacence**, il est possible d'appliquer n fois les algorithmes de **Bellman-Ford** et **Dijkstra**. Ce dernier reste très efficace si le graphe est "creux" : $\mathcal{O}((a+n)n\log n)$, mais suppose les poids positifs.

On va donner ici deux algorithmes fonctionnant sur des graphes donnés par **matrice d'adjacence** :

- le premier est assez simple, et possède une complexité en $\mathcal{O}(n^3 \log n)$;
- le second (**Floyd-Warshall**) est plus astucieux et plus efficace, car sa complexité se réduit à $\mathcal{O}(n^3)$.

Multiplication de matrices

Matrice d'adjacence

Soit M la **matrice d'adjacence** d'un **graphe pondéré** $G = (S, A, \omega)$.

On considère la suite de matrices $(D_i)_{i \geq 1}$ définie par :

- $D_1 = M$;
- $D_m = (d_{i,j}^m)_{0 \leq i,j \leq n-1}$ avec :

$$\begin{aligned} d_{i,j}^m &= \min\{d_{i,j}^{m-1}\} \cup \{d_{i,k}^{m-1} + \omega(k,j) \mid k \in [\![0, n-1]\!]\} \\ &= \min\{d_{i,k}^{m-1} + \omega(k,j) \mid k \in [\![0, n-1]\!]\} \text{ car } w_{j,j} = 0 \end{aligned}$$

Multiplication de matrices

Proposition

Avec cette construction, $d_{i,j}^m$ est le **poids** d'un **plus court chemin** $i \rightsquigarrow j$ de **longueur au plus** m.

Preuve

Immédiat par récurrence.

Multiplication de matrices

Calcul matriciel

Ainsi, si le graphe ne possède pas de circuit de poids **strictement négatif**, il suffit de calculer la matrice D_{n-1} pour obtenir tous les $\delta(i,j)$.

En suivant la définition, le calcul de D_m en fonction de D_{m-1} se fait avec une complexité en $\mathcal{O}(n^3)$: il y a n^2 coefficients, et le calcul de chaque $d_{i,j}^m$ se fait en $\mathcal{O}(n)$.

On en déduit donc un algorithme de complexité $\mathcal{O}(n^4)$ pour le calcul de D_{n-1}.

On peut en fait accélérer le processus en calquant la multiplication matricielle usuelle.

Multiplication de matrices

Proposition

$(\mathbb{R} \cup \{+\infty\}, \min, +, +\infty, 0)$ est un **semi-anneau commutatif**, c'est à dire que :

- $(\mathbb{R} \cup \{+\infty\}, \min)$ est un **monoïde commutatif**, de neutre $+\infty$;
- $(\mathbb{R} \cup \{+\infty\}, +)$ est également un **monoïde commutatif**, de neutre 0 ;
- $+$ est **distributive** par rapport à $\min$;
- $+\infty$ est **absorbant** pour $+$.

Multiplication de matrices

Preuve

Tout s'écrit facilement. Vérifions la distributivité de $+$ sur $\min$:

$$a + \min(b, c) = \min(a + b, a + c)$$

L'associativité de $\min$ est aussi immédiate :

$$\min(a, \min(b, c)) = \min(a, b, c) = \min(\min(a, b), c)$$

Multiplication de matrices

Multiplication matricielle

La proposition précédente montre que l'on peut définir un produit matriciel associatif pour les matrices de $\mathcal{M}_n(\mathbb{R} \cup \{+\infty\})$.

La loi de multiplication de deux telles matrices $A = (a_i)_{0 \leq i,j \leq n-1}$ et $B = (b_i)_{0 \leq i,j \leq n-1}$ est la suivante : $A \star B = C$ où $C = (c_i)_{0 \leq i,j \leq n-1}$ avec :

$$c_{i,j} = \min\{a_{i,k} + b_{k,j} \mid 0 \leq k \leq n-1\}$$

Le neutre pour ce produit est la matrice avec des zéros sur la diagonale et des $+\infty$ ailleurs.

Multiplication matricielle

Avec ce produit, on a $D_i = M \star \cdots \star M = M^i$.

On peut donc calculer $D_{n-1} = M^{n-1}$ par **exponentiation rapide**, avec une complexité $\mathcal{O}(n^3 \log n)$.

En fait, il nous suffit d'avoir M^k pour un $k \geq n-1$, ce qui fournit l'algorithme suivant.

Multiplication de matrices

Algorithme 3 : Algorithme de multiplication matricielle

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par sa matrice d'adjacence M
Résultat : La matrice $(\delta(i,j))_{0 \leq i,j \leq n-1}$ des plus courtes distances entre deux sommets quelconques du graphe

$D \leftarrow \mathrm{copie}(M)$;
$k \leftarrow 1$;
tant que $k < n\text{-}1$ **faire**
  $D \leftarrow D \star D$;
  $k \leftarrow 2 \times k$;

retourner D ;

Multiplication de matrices

Détection d'un circuit de poids strictement négatif

Pour détecter un circuit de poids strictement négatif, on peut procéder de manière similaire à l'algorithme de **Bellman-Ford** : s'il existe un tel circuit, alors il en existe un tel que tous les sommets soient distincts, excepté le premier sommet qui coïncide avec le dernier, notons le s.

On a alors un chemin $s \rightsquigarrow s$ de poids strictement négatif, et de longueur au plus n.

Ainsi, le **coefficient diagonal** de la matrice M^n en case (s, s) est strictement négatif.

On peut donc légèrement modifier l'algorithme pour calculer M^k avec $k \geq n$ (faire un tour de boucle de plus), et vérifier s'il existe un **coefficient diagonal strictement négatif**.

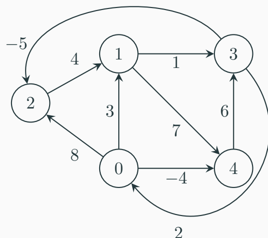
Multiplication de matrices

Calcul effectif de plus court chemins

Pour retrouver les plus courts chemins dans l'algorithme précédent, la méthode la plus économique en mémoire consiste là encore à stocker une **matrice de liaison** les prédécesseurs dans des plus courts chemins entre deux sommets.

On pose donc $\pi_{i,j}$ le prédécesseur de j dans un plus court chemin de i à j, s'il existe. Initialement, $\pi_{i,j} = i$ si $(i, j) \in A$, et a une valeur arbitraire sinon.
Lors du calcul de D^2, si le coefficient en case (i, j) est abaissé (i.e. $a_{i,j} > a_{i,k} + a_{k,j}$), alors $\pi_{i,j}$ prend la valeur $\pi_{k,j}$.
En fin d'algorithme, s'il existe un chemin $i \rightsquigarrow j$ pour $i \neq j$, alors $\pi_{i,j}$ contient l'avant dernier sommet d'un tel plus court chemin.

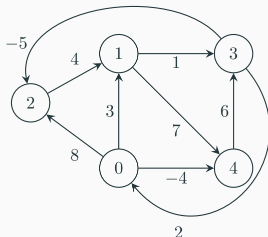
Multiplication de matrices



Exemple

Le graphe ci-dessus est un graphe d'ordre 5, il suffit donc de faire deux multiplications pour obtenir les poids des plus courts chemins (une troisième permet de s'assurer qu'il n'y a pas de circuit de poids strictement négatif).
Seul les coefficients pertinents de la matrice Π sont indiqués.

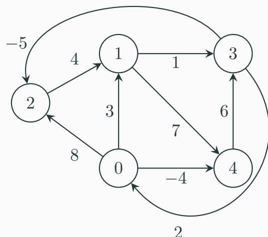
Multiplication de matrices



Exemple : Initialisation

$$D = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & \cdot & 0 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 3 & \cdot & 3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 4 & \cdot \end{pmatrix}$$

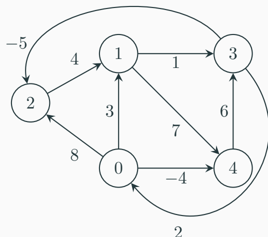
Multiplication de matrices



Exemple : Première multiplication

$$D^2 = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 1 \\ \cdot & 2 & \cdot & 1 & 1 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & \cdot & 3 & 4 & \cdot \end{pmatrix}$$

Multiplication de matrices



Exemple : Deuxième multiplication

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Multiplication de matrices

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Exemple

La matrice D^4 indique par exemple qu'il existe un chemin de poids -1 allant de 1 à 4.

La ligne 1 de Π est le tableau des prédécesseurs es plus courts chemins partant de 1, on peut donc reconstituer le chemin $1 \rightsquigarrow 4$ suivant :

Multiplication de matrices

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Exemple

La matrice D^4 indique par exemple qu'il existe un chemin de poids -1 allant de 1 à 4.

La ligne 1 de Π est le tableau des prédécesseurs es plus courts chemins partant de 1, on peut donc reconstituer le chemin $1 \rightsquigarrow 4$ suivant : 4

Multiplication de matrices

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Exemple

La matrice D^4 indique par exemple qu'il existe un chemin de poids -1 allant de 1 à 4.

La ligne 1 de Π est le tableau des prédécesseurs es plus courts chemins partant de 1, on peut donc reconstituer le chemin $1 \rightsquigarrow 4$ suivant : $0 \to 4$

Multiplication de matrices

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Exemple

La matrice D^4 indique par exemple qu'il existe un chemin de poids -1 allant de 1 à 4.

La ligne 1 de Π est le tableau des prédécesseurs es plus courts chemins partant de 1, on peut donc reconstituer le chemin $1 \rightsquigarrow 4$ suivant : $3 \to 0 \to 4$

Multiplication de matrices

$$D^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Exemple

La matrice D^4 indique par exemple qu'il existe un chemin de poids -1 allant de 1 à 4.

La ligne 1 de Π est le tableau des prédécesseurs es plus courts chemins partant de 1, on peut donc reconstituer le chemin $1 \rightsquigarrow 4$ suivant : $1 \to 3 \to 0 \to 4$

Multiplication de matrices

Remarque

On peut se demander s'il est possible d'utiliser un algorithme de multiplication sous-cubique (comme l'algorithme de **Strassen**, de complexité $\mathcal{O}(n^{\log_2(7)})$) pour améliorer la complexité précédente.

En fait, la réponse est oui, mais pas tel quel, car $\mathbb{R} \cup \{\infty\}$ n'a qu'une structure de **semi-anneau**, et l'algorithme de **Strassen** demande d'effectuer des **soustractions**.

Mais ces questions sortent très largement du programme de MP2I !

Algorithme de Floyd-Warshall

Floyd-Warshall

L'algorithme de **Floyd-Warshall** est un autre algorithme de programmation dynamique qui calcule une suite de matrices (M_i), mais le passage de M_i à M_{i+1} se fait simplement en temps $\mathcal{O}(n^2)$.

Algorithme de Floyd-Warshall

Définition (Floyd-Warshall)

Dans l'algorithme de **Floyd-Warshall** sur un **graphe pondéré** $G = (S, A, \omega)$, de matrice d'adjacence M, on pose $\forall k \in [\![0, n]\!]$, $M_k = (m_{i,j}^k)$ avec $m_{i,j}^k$ le poids minimal d'un chemin de i à j dont tous les **sommets intermédiaires** (i.e. i et j exclus) sont dans $[\![0, k-1]\!]$.

Remarque

On a donc notamment $M_0 = M$ car les chemins sans sommets intermédiaires sont simplement les arcs.

Remarque

Ces matrices sont à valeurs dans $\mathbb{R} \cup \{\pm\infty\}$, mais bien sûr la valeur $-\infty$ ne se produit que dans le cas où il y a un **circuit de poids strictement négatif**.

En excluant ce cas, la proposition suivante indique comment calculer M_{k+1} à partir de M_k.

Proposition

En l'absence de circuits de poids strictement négatif dans le graphe, on a :

$$\forall (i,j,k) \in [\![0, n-1]\!],\ m_{i,j}^{k+1} = \min(m_{i,j}^k, m_{i,k}^k + m_{k,j})$$

Algorithme de Floyd-Warshall

Preuve

- S'il n'y a pas de chemin entre i et j, ne passant que par des sommets de $[\![0, k]\!]$, alors $m_{i,j}^k = m_{i,j}^{k+1} = +\infty$, et l'un des deux $m_{i,k}^k$ ou $m_{k,j}^k$ vaut aussi $+\infty$ (car sinon, on aurait deux chemins $i \rightsquigarrow k \rightsquigarrow j$ dont la concaténation fournirait un chemin $i \rightsquigarrow j$).

Preuve

- Sinon, un chemin de poids minimal entre i et j ne passant que par des sommets de $[\![0, k]\!]$ peut être supposé ne passer qu'au plus une seule fois par k (il suffit de supprimer le circuit–de poids nécessairement nul–entre les première et dernière occurrences de k dans un chemin de poids minimal pour en obtenir un de même poids dans lequel k apparaît au plus une fois).
    - Si ce chemin ne passe pas par k, on a $m_{i,j}^{k+1} = m_{i,j}^{k}$.
    - Sinon, le chemin se décompose en $i \overset{c_1}{\rightsquigarrow} k \overset{c_2}{\rightsquigarrow} j$ avec c_1 et c_2 deux chemins dont les sommets intermédiaires sont dans $[\![0, k-1]\!]$. Ainsi : $m_{i,j}^{k+1} = m_{i,k}^{k} + m_{k,j}^{k}$.

Complexité

On obtient ainsi un algorithme de complexité $\mathcal{O}(n^3)$ pour le calcul de tous les $\delta(i,j)$.

De plus, la remarque suivante indique que l'on peut se contenter de mettre à jour une unique matrice.

Remarque

En l'absence de circuits poids strictement négatifs, on a :

$$\begin{aligned}\forall (i,j,k) \in [\![0, n-1]\!],\ m_{i,j}^{k+1} &= \min(m_{i,j}^{k}, m_{i,k}^{k} + m_{k,j}^{k}) \\ &= \min(m_{i,j}^{k}, m_{i,k}^{k+1} + m_{k,j}^{k}) \\ &= \min(m_{i,j}^{k}, m_{i,k}^{k} + m_{k,j}^{k+1}) \\ &= \min(m_{i,j}^{k}, m_{i,k}^{k+1} + m_{k,j}^{k+1})\end{aligned}$$

En effet, remplacer $m_{i,k}^{k}$ par $m_{i,k}^{k+1}$ ne change rien, car il n'y a pas de circuit de poids strictement négatif bouclant sur k.

Algorithme de Floyd-Warshall

Algorithme 4 : Algorithme de Floyd-Warshall

Données : Un graphe pondéré $G = (S, A, \omega)$ donné par sa matrice d'adjacence M

Résultat : La matrice $(\delta(i,j))_{0 \leq i,j \leq n-1}$ des plus courtes distances entre deux sommets quelconques du graphe

$D \leftarrow \mathrm{copie}(M)$;

pour $k \in [\![0, n-1]\!]$ **faire**

  pour $i \in [\![0, n-1]\!]$ **faire**

    pour $j \in [\![0, n-1]\!]$ **faire**

      $d_{i,j} \leftarrow \min(d_{i,j}, d_{i,k} + d_{k,j})$;

retourner D ;

Algorithme de Floyd-Warshall

Complexité

La complexité de l'algorithme de **Floyd-Warshall** est clairement en $O(n^3)$.

Détection de circuit de poids strictement négatif

S'il existe un **circuit de poids strictement négatif**, prenons-en un sans sommet en double excepté les extrémités.

Soit i le sommet aux extrémités. On aura alors $d_{i,i} < 0$ à la fin de l'algorithme. Il suffit donc de tester l'existence d'un élément **diagonal strictement négatif** à la fin de l'algorithme pour tester l'existence d'un **circuit de poids total strictement négatif**.

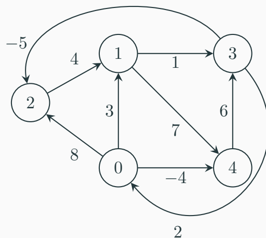
Algorithme de Floyd-Warshall

Calcul effectif des plus courts chemins

Parmi d'autres méthodes, on peut là encore calculer une **matrice de liaison**, comme dans l'algorithme par multiplications matricielles.

Lorsqu'on fait $d_{i,j} \leftarrow d_{i,k} + d_{k,j}$, on effectue parallèlement $\pi_{i,j} \leftarrow \pi_{k,j}$, la matrice $\Pi = (\pi_{i,j})$ étant initialisée comme dans l'algorithme précédent.

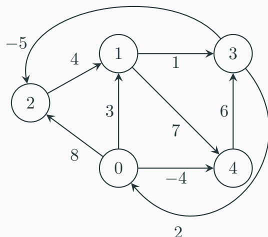
Algorithme de Floyd-Warshall



Exemple

On reprend le même exemple que précédemment, pour naturellement obtenir les mêmes matrices finales.

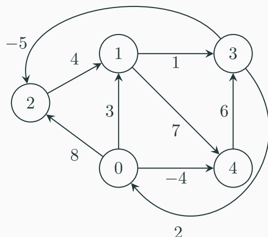
Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_0 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & \cdot & 0 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 3 & \cdot & 3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 4 & \cdot \end{pmatrix}$$

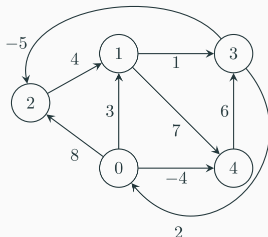
Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_1 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \color{orange}{5} & -5 & 0 & \color{orange}{-2} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & \cdot & 0 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 2 & \cdot & \cdot & \cdot \\ 3 & \color{orange}{0} & 3 & \cdot & \color{orange}{0} \\ \cdot & \cdot & \cdot & 4 & \cdot \end{pmatrix}$$

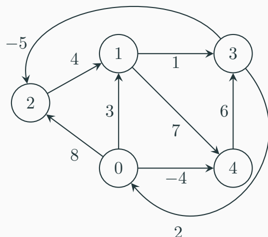
Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_2 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & 1 & 0 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 2 & \cdot & 1 & 1 \\ 3 & 0 & 3 & \cdot & 0 \\ \cdot & \cdot & \cdot & 4 & \cdot \end{pmatrix}$$

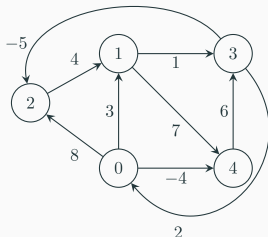
Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_3 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 0 & 1 & 0 \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & 2 & \cdot & 1 & 1 \\ 3 & 2 & 3 & \cdot & 0 \\ \cdot & \cdot & \cdot & 4 & \cdot \end{pmatrix}$$

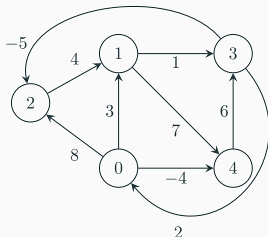
Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_4 = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 0 & 3 & 1 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Algorithme de Floyd-Warshall



Exemple : Initialisation

$$M_5 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} \cdot & 2 & 3 & 4 & 0 \\ 3 & \cdot & 3 & 1 & 0 \\ 3 & 2 & \cdot & 1 & 0 \\ 3 & 2 & 3 & \cdot & 0 \\ 3 & 2 & 3 & 4 & \cdot \end{pmatrix}$$

Algorithme de Floyd-Warshall

Application : fermeture transitive d'un graphe

Une dérivation de l'algorithme de **Floyd-Warshall** permet de répondre facilement au problème de l'accessibilité dans un graphe, et fournit l'algorithme de **Warshall** (historiquement antérieur à l'algorithme de **Floyd-Warshall**).

Définition (fermeture transitive)

Soit $G = (S, A)$ un graphe orienté, non pondéré.

On appelle **fermeture transitive** de G le graphe $\tilde{G} = (S, \tilde{A})$, où pour $u \neq v$ deux sommets de $\tilde{G}$, $(u, v) \in \tilde{A}$ s'il existe un chemin $u \rightsquigarrow v$ dans G.

Résumé des algorithmes de plus courts chemins

Problème (2)	Limitation	Complexité
Bellman-Ford	—	$\mathcal{O}(n(a+n))$
Dijkstra	poids positifs	$\mathcal{O}((a+n)\log n)$ (tas min) $\mathcal{O}(n^2)$ (tableau)
Problème (4)	Limitation	Complexité
Bellman-Ford	—	$\mathcal{O}(n^2(a+n))$
Dijkstra	poids positifs	$\mathcal{O}((a+n)n\log n)$ (tas min) $\mathcal{O}(n^3)$ (tableau)
Multiplication matricielle	—	$\mathcal{O}(n^3\log n)$
Floyd-Warshall	—	$\mathcal{O}(n^3)$

Résumé des algorithmes de plus courts chemins

Remarque

Pour la résolution du problème (4) sur un **graphe dense**, la complexité asymptotique de l'algorithme de **Dijkstra** utilisé n fois (en gérant la file de priorité avec un tableau) est la même que celle de l'algorithme de **Floyd-Warshall** : $\mathcal{O}(n^3)$.

Néanmoins, la constante cachée dans le $\mathcal{O}$ est plus faible pour l'algorithme de **Floyd-Warshall**, et celui-ci a le mérite de s'appliquer même s'il y a des arcs de **poids négatifs**.

Pour un **graphe dense**, on préférera l'algorithme de **Floyd-Warshall** pour calculer une solution au problème (4).

Arbre couvrant de poids minimal

Arbre couvrant de poids minimal

Définition (arbre couvrant)

Soit $G = (S, A, \omega)$ un **graphe pondéré, non orienté, connexe**, dont la fonction de pondération est à valeurs dans $\mathbb{R}_+$.

On appelle **graphe couvrant** de G un sous-graphe **connexe** $G' = (S, A')$ tel que $A' \subseteq A$.

On appelle **poids** du graphe couvrant la somme $\sum_{a \in A'} \omega(a)$.

Si G' est un arbre, on dit que c'est un **arbre couvrant**.

Arbre couvrant de poids minimal

Problème de l'arbre couvrant minimal

Le problème de "**l'arbre couvrant minimal**" est de trouver un graphe couvrant de poids minimal.

La propriété suivante indique que l'on peut chercher un arbre.

Proposition

En reprenant les notations de la définition précédente, il existe un **graphe couvrant** de G de **poids minimal** qui est un **arbre**.

Arbre couvrant de poids minimal

Preuve

Soit G' un graphe couvrant de poids minimal de G (ce graphe existe, car il existe au moins un graphe couvrant de G : lui-même). On peut de plus supposer G' minimal en nombre d'arêtes parmi les graphes couvrants de poids minimal.

S'il existait un **cycle** dans G', on pourrait retirer une arête du cycle sans perdre la connexité et en diminuant le poids du graphe : **absurde**.

Ainsi, G' est connexe et acyclique : c'est donc un arbre.

Arbre couvrant de poids minimal

Remarque

En pratique, si les valeurs de ω sont strictement positives, un graphe couvrant minimal est un arbre.

S'il y a des arêtes de poids nul, un graphe couvrant minimal peut ne pas être un arbre, mais il existe au moins un graphe couvrant qui est un arbre.

L'algorithme de **Prim**, très proche de l'algorithme de **Dijkstra**, permet de trouver un tel arbre.

Arbre couvrant de poids minimal

Algorithme 5 : Algorithme de Prim

Données : Un graphe pondéré, non orienté, connexe $G = (S, A, \omega)$ donné par listes d'adjacence avec $\omega(A) \subset \mathbb{R}_+$, un sommet s

Résultat : Un ensemble d'arêtes formant un arbre couvrant minimal

```
ds[t] ← +∞ pour tout t ∈ S ; p[t] ← t pour tout t ∈ S ;
ds[s] ← 0 ; n ← |S| ; H ← ∅ ; F ← {s} ; A' ← ∅ ;
pour i ∈ ⟦0, n − 1⟧ faire
    u ← Retirer de F un sommet v vérifiant ds[v] minimal ;
    si u ≠ s alors
        A' ← A' ∪ {{u, p[u]}} ;
    pour tout voisin v de u faire
        si v ∉ F et v ∉ H alors
            Ajouter v à F ;
        si ds[v] > ω(u, v) alors
            ds[v] ← ω(u, v) ;
            p[v] ← u ;
    Ajouter u à H ;
retourner A' ;
```

Arbre couvrant de poids minimal

Théorème (Correction de l'algorithme de Prim)

Avec A' l'ensemble renvoyé par l'algorithme de **Prim**, (S, A') est un **arbre couvrant** de G de **poids minimal**.

Arbre couvrant de poids minimal

Preuve

L'algorithme réalise un parcours du graphe, qui atteindra tous les sommets car G est connexe : les n sommets vont donc passer par F.

L'algorithme ajoute à A' une arête pour chaque sommet excepté le sommet s de départ.
Ainsi, à la fin de l'algorithme, on a $|A'| = n - 1$.

De plus, (S, A') est connexe car tous les sommets sont raccordés au sommet initial s.

Donc (S, A') est un arbre.

Arbre couvrant de poids minimal

Preuve

Notons $s = s_0, s_1, \ldots, s_{n-1}$ les sommets dans l'ordre de leur ajout à H, et pour $i \in [\![1, n-1]\!]$ notons a_i l'arête ajoutée à A', reliant s_i au graphe induit par $\{s_j \mid j < i\}$.

Supposons que (S, A') ne soit pas un arbre couvrant de poids minimal, et considérons un arbre $(S, \tilde{A})$ couvrant de poids minimal, et notons :

$$i = \max\{j \mid a_1, \ldots, a_j \in \tilde{A}\}$$

On peut sans perte de généralité supposer que $(S, \tilde{A})$ est un arbre couvrant minimal pour lequel i est **maximal**.

Arbre couvrant de poids minimal

Preuve

L'arête a_{i+1} reliant s_{i+1} à un sommet s_j (avec $j \leq i$) n'est pas dans $\tilde{A}$. Raisonnons sur le graphe $(S, \tilde{A} \cup \{a_{i+1}\})$: ce graphe possède un cycle car $(S, \tilde{A})$ est un arbre.
En considérant le cycle comme un chemin bouclant de s_j sur lui-même en terminant par l'arête a_{i+1}, notons $\tilde{s}$ le premier sommet qui n'est pas dans $\{s_0, \ldots, s_i\}$ et a l'arête le reliant au sommet précédent.

- $\omega(a) \geq \omega(a_{i+1})$, car sinon a_{i+1} n'aurait pas été rajoutée à A'. En effet, on aurait eu $d_s[v] < d_s[v_{i+1}]$ à ce moment là.
- $\omega(a) \leq \omega(a_{i+1})$, car sinon $(S, \tilde{A} \cup \{a_{i+1}\} \setminus \{a\})$ serait un arbre couvrant de poids strictement inférieur à celui de $(S, \tilde{A})$.

Arbre couvrant de poids minimal

Preuve

Ainsi, a a le même poids que a_{i+1}, et $(S, \tilde{A} \cup \{a_{i+1}\} \setminus \{a\})$ est un arbre couvrant de même poids que $(S, \tilde{A})$ mais tel que $\max\{j \mid a_1, \ldots, a_j \in \tilde{A}\} > i$: **absurde**.

Ainsi, (S, A') est un **arbre couvrant de poids minimal**.

Arbre couvrant de poids minimal

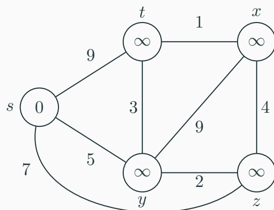
Complexité

La complexité de l'algorithme de **Prim** est en $\mathcal{O}(n^2)$ avec une implémentation avec tableaux, et en $\mathcal{O}(a \log n)$ avec une file de priorité implémentée avec un tas binaire.

Preuve

La complexité est exactement la même que dans l'algorithme de **Dijkstra**, avec ici $n = \mathcal{O}(a)$ car le graphe est **connexe**.

Arbre couvrant de poids minimal



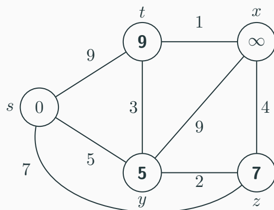
$$H = \emptyset,\ F = \{s\}$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.

Arbre couvrant de poids minimal



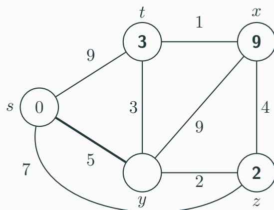
$$H = \{s\},\ F = \{t, y\}$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.

Arbre couvrant de poids minimal



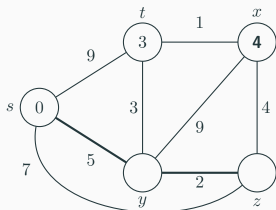
$$H = \{s, y\},\ F = \{t, x, z\}$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.

Arbre couvrant de poids minimal



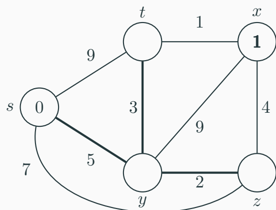
$$H = \{s, y, z\},\ F = \{t, x\}$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.

Arbre couvrant de poids minimal



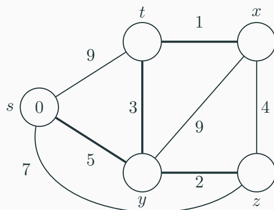
$$H = \{s, y, z, t\},\ F = \{x\}$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.

Arbre couvrant de poids minimal



$$H = \{s, y, z, t, x\},\ F = \emptyset$$

Exemple

On termine par l'exemple du graphe suivant, où l'on part de s pour trouver un **arbre couvrant minimal** (de poids 11).

On fait figurer dans chaque sommet $u \notin H$ la valeur $d_s[u]$; les arêtes en **gras** forment l'arbre couvrant minimal.