

Parcours de graphes suite

Michel Habib

M1 Informatique, Algorithmique Avancée
Université Paris Diderot 2012

11 février 2012

Plan

- 1 Parcours générique
- 2 Parcours classiques
 - Algorithme de Tarjan 1972
 - Parcours en largeur lexicographique
- 3 Algorithmes de plus courts chemins
 - Algorithme de Dijkstra
 - L'algorithme A^*
 - Algorithme de Bellman - Ford
 - Algorithme de Prim
- 4 Un cadre général
- 5 Les graphes sans circuits
- 6 Fermeture transitive

Graph searches are very well known and used

- 1 Euler (1735) for solving the famous walk problem in Kœnisberg
- 2 Tremaux (1882) and Tarry (1895) using DFS to solve maze problems
- 3 Computer scientists from 1950
- 4 But also : "Fil d'ariane" in the Greek mythology.

LE PROBLÈME DES LABYRINTHES;

PAR M. G. TARRY.

Tout labyrinthe peut être parcouru en une seule course, en passant deux fois en sens contraire par chacune des allées, sans qu'il soit nécessaire d'en connaître le plan.

Pour résoudre ce problème, il suffit d'observer cette règle unique :

Ne reprendre l'allée initiale qui a conduit à un carrefour pour la première fois que lorsqu'on ne peut pas faire autrement.

Nous ferons d'abord quelques remarques.

Umberto Eco, "Il nome della rosa", Roman, 1980

« Pour trouver la sortie d'un labyrinthe, récita en effet Guillaume, il n'y a qu'un moyen. A chaque nœud nouveau, autrement dit jamais visité avant, le parcours d'arrivée sera marqué de trois signes. Si, à cause de signes précédents sur l'un des chemins du nœud, on voit que ce nœud a déjà été visité, on placera un seul signe sur le parcours d'arrivée. Si tous les passages ont été déjà marqués, alors il faudra reprendre la même voie, en revenant en arrière. Mais si un ou deux passages du nœud sont encore sans signes, on en choisira un quelconque, pour y apposer deux signes. Quand on s'achemine par un passage qui porte un seul signe, on en apposera deux autres, de façon que ce passage en porte trois dorénavant. Toutes les parties du labyrinthe devraient avoir été parcourues si, en arrivant à un nœud, on ne prend jamais le passage avec trois signes, sauf si d'autres passages sont encore sans signes.

– Comment le savez-vous? Vous êtes expert en labyrinthes?

– Non, je récite un extrait d'un texte antique que j'ai lu autrefois.

– Et selon cette règle, on sort?

– Presque jamais, que je sache. Mais nous tenterons quand même. Et puis dans les prochains jours j'aurai des verres et j'aurai le temps de mieux me pencher sur les livres. Il se peut

Three main aspects for a graph search :

- 1 its principle or its algorithm
(i.e. the description of the tie-break rules for the choice of the next vertex (edge) to be explored)
- 2 The study of the underlying tree structure
- 3 The complexity analysis and its implementation or its program

Parcours générique

ParcoursGénérique(G, x_0);

Données: un graphe orienté $G = (X, U)$, un sommet $x_0 \in X$

Résultat: une arborescence de chemins issus de x_0

$OUVERTS \leftarrow \{x_0\}$; $FERMES \leftarrow \emptyset$; $Parent(x_0) \leftarrow NIL$;

$\forall y \neq x_0, Parent(y) \leftarrow y$;

tant que $OUVERTS \neq \emptyset$ **faire**

$z \leftarrow Choix(OUVERTS)$;

$Ajout(z, FERMES)$;

 Explorer(z);

$Retrait(z, OUVERTS)$;

Exploration d'un sommet

Explorer(z) :

pour *Tous les voisins y de z* **faire**

si $y \in \text{FERMES}$ **alors**

 └ Ne rien faire

si $y \in \text{OUVERTS}$ **alors**

 └ Ne rien faire

sinon

 └ $\text{Ajout}(y, \text{OUVERTS})$;

 └ $\text{Parent}(y) \leftarrow z$;

Invariants :

- La fonction Parent définit une arborescence de racine x_0 ,
- $\forall x \in OUVERTS$, il existe un chemin de x_0 à x .

Ainsi à la fin du parcours, l'ensemble des sommets FERMES est égal à l'ensemble des sommets atteignables dans le graphe G à partir de x_0 . Cet ensemble est décrit à l'aide de la fonction Parent. Un sommet est exploré au plus une fois et donc l'algorithme est en $O(n + m)$ si le graphe est représenté par ses listes d'adjacence.

Lorsque tous les sommets du graphe ne sont atteignables à partir de x_0 si l'on veut parcourir tout le graphe, il suffit d'ajouter les instructions suivantes et de représenter l'ensemble des sommets FERMES à l'aide d'un tableau de booléens :

Parcours(G) :

pour tous les $x \in X$ faire

└ $Ferme(x) \leftarrow Faux$

pour tous les $x \in X$ faire

└ **si $Ferme(x) = Faux$ alors**
└└ $ParcoursGénérique(G, x)$

Dans ce cas l'algorithme calcule une forêt d'arborescences recouvrante de G .

Historique

- Défini en premier par Tremaux 1882 puis par Tarry 1895 pour des problèmes de parcours dans les labyrinthes.
- Redécouvert en 1972 par Tarjan et Hopcroft pour écrire des algorithmes de résolution des problèmes de recherche dans un graphe : le calcul des composantes fortement connexes, celui des composantes 2-arêtes connexes, le test de planarité.

Parcours en profondeur

On obtient un parcours en profondeur dès que l'on gère l'ensemble des sommets OUVERTS comme une pile (dernier entré, premier sorti). Cette gestion d'une pile se prête à une écriture récursive du programme, comme suit :

Et les deux fonctions suivantes utilisant deux variables : comptpre et comptpost étant initialisées à 1.

DFS(G, x) :

forall $v \in X$ **do**

└ $Ferme(x) \leftarrow Faux$

forall $v \in X$ **do**

└ **si** $Ferme(x) = Faux$ **alors**
└└ $Explorer(G, x)$

Explorer(G, x) :

Ferme(x) \leftarrow *Vrai* ;

pre(x) ;

forall $xy \in U$ **do**

si *Ferme*(y) = *Faux* **alors**
 └ *Explorer*(G, y)

post(x) ;

pre(x) :

```
pre(x) ← comptpre ;  
comptpre ← comptpre + 1 ;
```

post(x) :

```
post(x) ← comptpost ;  
comptpost ← comptpost + 1 ;
```

Les variables *comptpre* et *comptpost* étant initialisées à 1

Comme tous les parcours, un parcours en profondeur permet de construire une forêt d'arborescences.

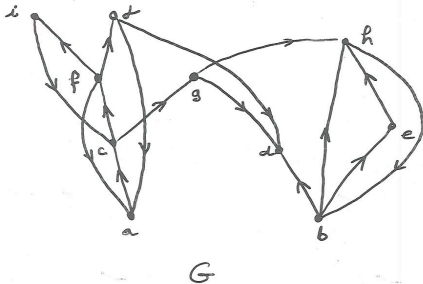
On distingue trois types d'arcs :

- Arcs de retour
- Arcs traversiers
- Arcs de transitivité de la forêt d'arborescences

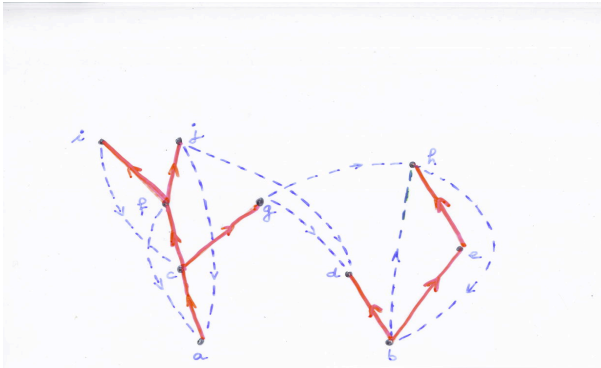
Les ordres pre et post permettent de les caractériser, comme l'indiquent les trois équivalences suivantes :

- xy arc de retour ssi
 $pre(y) \leq pre(x)$ and $post(x) \leq post(y)$
- xy arc traversier ssi
 $pre(y) \leq pre(x)$ and $post(y) \leq post(x)$
- xy arc de transitivité de la forêt d'arborescences ssi
 $pre(x) \leq pre(y)$ and $post(y) \leq post(x)$
- Le cas où $pre(x) \leq pre(y)$ and $post(x) \leq post(y)$ étant impossible par le principe du parcours en profondeur.

Ces trois types d'arcs partitionnent les arcs du graphe.



	a	b	c	d	e	f	g	h	i	j
pre	5	1	6	4	2	7	10	3	8	9
post	10	4	9	3	2	7	8	1	5	6
racine	5	1	5	4	1	5	10	1	5	5
post ^d	1	7	2	8	9	4	3	10	6	5



Définition

Une **extension linéaire** d'un graphe sans circuit $G = (X, U)$ est un ordre total sur les sommet de G , noté τ , vérifiant :

$\forall x, y \in X, xy \in U$ implique $x \leq_{\tau} y$.

Certains auteurs appellent une extension linéaire un **tri topologique** ou encore **un ordre total compatible**.

Théorème

Si G est un graphe sans circuit, alors le parcours en profondeur appliqué sur G vérifie la propriété :

L'ordre inverse de dépilement (noté $post^d$) est une extension linéaire de G .

preuve

Considérons deux sommets $x, y \in X$ et supposons $xy \in U$. Deux cas sont possibles :

- 1 $pre(x) < pre(y)$. Mais alors l'exploration de y se terminera donc avant celle de x et nous avons bien : $post^d(x) < post^d(y)$.
- 2 $pre(y) < pre(x)$. Comme il n'existe pas de chemin de y à x , car sinon G aurait un circuit, nous avons que l'exploration de y se terminera avant celle de x . Et donc $post^d(x) < post^d(y)$.



Ce théorème peut aussi s'écrire comme suit :

Théorème

Après un parcours en profondeur sur un graphe G les 3 propriétés suivantes sont équivalentes :

- 1 G est sans circuit
- 2 Il n'y a pas d'arc de retour
- 3 $post^d$ est une extension linéaire de G

Corollaire

$post^d$ est une extension linéaire de G/P le graphe G quotienté par la partition en composantes fortement connexes de G .

Le parcours en profondeur permet donc de calculer une extension linéaire d'un graphe sans circuit. Il existe d'autres algorithmes, le plus classique étant celui basé sur la remarque que tout graphe sans circuit admet une source (i.e. un sommet tel que $d^-(x) = 0$).

Algorithme de Tarjan 1972

Il suffit d'ajouter deux nouvelles structures de données :
un tableau de booléens Pile définissant la présence d'un sommet dans une pile appelée Resultat de transformer la fonction d'exploration du parcours en profondeur comme suit :

DFS(G) : $comptpre \leftarrow 1$; $Resultat \leftarrow \emptyset$;

forall $x \in X$ **do**

┌ $Ferme(x) \leftarrow Faux$; $Pile(x) = Faux$;

forall $x \in X$ **do**

┌ **si** $Ferme(x) = Faux$ **alors**

┌ $Explorer(G, x)$

Explorer(G, x) : $Empiler(Resultat, x)$; $Pile(x) = Vrai$; $Ferme(x) \leftarrow Vrai$; $pre(x) \leftarrow comptpre$;

$comptpre \leftarrow comptpre + 1$; $racine(x) \leftarrow pre(x)$; **forall** $xy \in U$ **do**

┌ **si** $Ferme(y) = Faux$ **alors**

┌ $Explorer(G, y)$; $racine(x) \leftarrow \min\{racine(x), racine(y)\}$;

┌ **sinon**

┌ **si** $Pile(y) = Vrai$ **alors**

┌ $racine(x) \leftarrow \min\{racine(x), racine(y)\}$

si $racine(x) = pre(x)$ **alors**

┌ Dépiler $Resultat$ jusqu'à x inclus et écrire ces sommets; mettre à jour le tableau $Pile$;

Les composantes fortement connexes se repèrent récursivement avec le test $racine(x) = pre(x)$ à la fin de l'exploration de x
Plus précisément voici les invariants placés lorsqu'on a fini l'exploration d'un sommet x .

- 1 $\forall z$ tel que $pre(x) \leq pre(z)$ et $Pile(z) = Vrai$ nous avons :
 $racine(z) < pre(z)$
- 2 $\forall z$ tel que $Pile(z) = Vrai$ il existe un chemin de z au sommet y tel que $pre(y) = racine(z)$ n'utilisant que des sommets de Resultat.

théorème

Quand on trouve x t.q. $pre(x) = racine(x)$, $A = \{z \in X \mid pre(x) \leq pre(z)$ et $Pile(z) = Vrai\}$ constitue l'ensemble des sommets de la composante fortement connexe qui contient x .

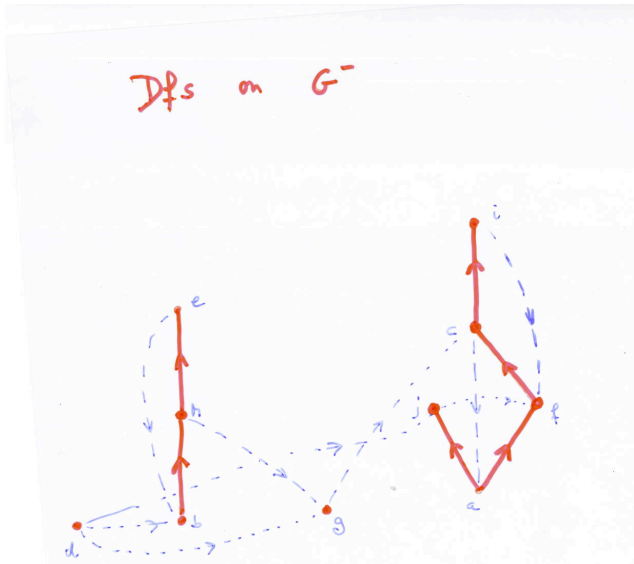
preuve

En utilisant les invariants il est facile de vérifier que $\forall z \in A$ il existe un chemin allant de z à x . Par ailleurs quand on a fini d'explorer x ce qui reste dans Resultat après x est constitué de descendants de x . Et donc ainsi $G(A)$ est fortement connexe.

La maximalité de A est aussi facile à vérifier.

Algorithme de Kosaraju 1978, Sharir 1981

- 1 Exécuter un parcours en profondeur sur G .
- 2 Faire un autre parcours en profondeur sur G^- avec pour ordre initial l'ordre *post*^d
- 3 Les arbres de la forêt recouvrante de G^- , sont les composantes fortement connexes de G



Numéros :	a	b	c	d	e	f	g	h	i	j
pre	5	1	6	4	2	7	10	3	8	9
post	10	4	9	3	2	7	8	1	5	6
$post^d$	1	7	2	8	9	4	3	10	6	5

Implémentations

Dans un parcours en profondeur nous avons le choix de parcourir les successeurs d'un sommet dans un ordre qui peut soit provenir :

- d'un parcours précédent comme dans l'algorithme de Kosaraju et Sharir.
- d'un ordre explicite de préférence sur les liens (utilisé pour les algorithmes d'héritage en programmation objet)
- Un parcours en profondeur traverse chaque arête une fois,, alors qu'un parcours en largeur examine tout le voisinage d'un sommet à la fois voisinage noté $N(x)$

Parcours en largeur lexicographique

Il suffit de prendre $T = \{1, 2, \dots, n\}^*$ (l'ensemble des mots construits avec les nombres 1,2 jusqu'à n), et de choisir comme sommet à explorer celui dont l'étiquette est maximale lexicographiquement. Cet algorithme s'utilise sur les graphes non orientés. La fonction Explorer devenant :

Explorer(z) :

$numero(z) \rightarrow numerocourant - 1$;

$numerocourant \rightarrow numerocourant - 1$;

pour Tous les voisins y de z **faire**

si $y \in FERMES$ **alors**

 └ Ne rien faire

si $y \in OUVERTS$ **alors**

 | $d(y) \rightarrow d(y) \bullet numero(z)$

sinon

 └ $d(y) \rightarrow d(y) \bullet numero(z)$ Ajout(y , OUVERTS);

le symbole "•" représentant l'opération de concaténation, on étiquette ainsi les sommets du graphe avec des mots sur $\{1, 2, \dots, n\}^*$.
La variable globale `numerocourant` doit être initialisée à n dans les initialisations de la procédure principale.

Le problème 1 est-il toujours bien défini ?

On considère un graphe $G = (X, U)$ orienté et l'on suppose les arcs munis d'étiquettes, i.e. une valuation $\omega : U \rightarrow R$.

Non en cas de valuation de signe qq, il peut exister des circuits négatifs (dits **absorbants**).

Exemple :

$$\text{valuation}(ij) = \text{cout}(ij) + \text{taxe} - \text{subvention}$$

Pour un graphe $G = (X, U)$ orienté dont les arcs sont valués

- Etant donné deux sommets a et b , trouver un plus court chemin allant de a jusqu'à b dans G .
- Hélas ce problème est NP-complet !

S'il existe une plus courte marche orientée allant de a vers b dans G , alors il existe un plus court chemin de a vers b dans G .
C'est pourquoi nous pouvons parler des problèmes de plus courts chemins.

Applications

Le problème de la recherche d'une plus court *chemin* dans un graphe orienté a de très nombreuses applications pratiques, telles :

- Routages de paquets dans les réseaux, routages de véhicules dans les réseaux de transports
- Diamètre d'un réseau de télécommunications (qualité de service)
- Problèmes de Transport
- Jeux (graphe dont les sommets sont les états du jeu et les arcs les transitions légales)
- Investissements, ordonnancements
- Navigation (routeurs de course au large : Vendée Globe Challenge, Route du Rhum, ...)

Les différents problèmes

Pour un graphe $G = (X, U)$ orienté dont les arcs sont valués En général la valuation d'un chemin est la **somme des valuations** des arcs qui constituent le chemin.

- 1 Etant donné deux sommets a et b , trouver une plus courte marche allant de a jusqu'à b dans G .
- 2 Etant donné un sommet a trouver pour chaque $x \in X$ une plus courte marche allant de a jusqu'à x dans G .
- 3 Pour tout $x, y \in G$, trouver une plus courte marche allant de x jusqu'à y .

Dans l'énumération ci-dessus, la complexité du problème est croissante. En effet un algorithme qui résout le problème 3, permet de résoudre le problème 2 et un algorithme qui résout le problème 2 permet de résoudre le problème 1.

Cependant lorsque les valuations choisies sont de signe quelconque, la solution au problème 1 peut-être une marche infinie comprenant un circuit de valeur négative (circuit dit **absorbant**).

Exemple : $valuation(ij) = cout(ij) + taxe - subvention$

Il est tentant de transformer le problème en :

Etant donné deux sommets a et b , trouver le plus court chemin allant de a jusqu'à b dans G .

Mais alors le problème devient NP-difficile.

Algorithme de Dijkstra 1959

Données: un graphe orienté $G = (X, U)$, une fonction de coût
 $\omega : U \rightarrow \mathcal{R}^+$

Résultat: une arborescence de chemins issus de x_0

$OUVERTS \leftarrow \{x_0\}$; $FERMES \leftarrow \emptyset$; $Parent(x_0) \leftarrow NIL$;

$\forall y \neq x_0$, $Parent(y) \leftarrow y$ $d(x_0) \leftarrow 0$;

$\forall y \neq x_0$, $d(y) \leftarrow +\infty$;

tant que $OUVERTS \neq \emptyset$ **faire**

 Choisir un sommet $z \in OUVERTS$ tel que

$d(z) = \min_{y \in OUVERTS} \{d(y)\}$;

 Ajout(z , $FERMES$) ;

 Explorer(z);

 Virer(z , $OUVERTS$) ;

Explorer(z);

pour Tous les voisins y de z **faire**

si $y \in \text{FERMES}$ **alors**

 Ne rien faire

sinon

si $y \in \text{OUVERTS}$ **alors**

si $d(z) + \omega(z, y) < d(y)$ **alors**

$\text{Parent}(y) \leftarrow z$;

$d(y) \leftarrow d(z) + \omega(z, y)$

sinon

$\text{Ajout}(y, \text{OUVERTS})$;

$\text{Parent}(y) \leftarrow z$;

$d(y) \leftarrow d(z) + \omega(z, y)$

Theorem

Lorsque la valuation ω est à valeurs positives, l'algorithme calcule bien une arborescence des plus courts chemins issus de x_0 .

Preuve :

Considérons les invariants principaux de l'algorithme.

Invariants

- 1 $\forall x \in OUVERTS$, il existe un chemin μ de x_0 à x tel que, $d(x) = \omega(\mu)$
- 2 À la fin de la i ème exploration d'un sommet,
 $\forall x \in OUVERTS \cup FERMES$, $d(x)$ est égale à la valuation minimale d'un chemin de G de x_0 à x n'utilisant que des sommets $FERMES$ sauf éventuellement x .
- 3 $\forall u \in FERMES$ and $\forall v \in OUVERTS$, $d(u) \leq d(v)$.

L'invariant 1 se montre facilement par récurrence, montrons l'invariant 2 aussi par récurrence. Pour se faire indexons par i tous les objets de l'algorithme à la fin de l'exploration du i ème sommet.

A l'initialisation lorsque $i = 0$, l'invariant est trivialement vrai. Supposons le maintenant vrai jusqu'à l'exploration du $(i-1)$ ème sommet. Soit x le sommet exploré à la i ème itération.

Considérons $z \in OUVERTS_i \cup FERMES_i$, il y a plusieurs cas possibles.

- Soit $xz \notin U$, mais alors il n'existe pas de chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z . Comme $d_i(z) = d_{i-1}(z)$, l'hypothèse de récurrence permet de conclure.
- Soit $xz \in U$ et $z \notin OUVERTS_{i-1} \cup FERMES_{i-1}$. Dans ce cas le seul chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z passe par x et l'algorithme affecte bien à $d_i(z)$ la valeur de ce chemin.
- Soit $xz \in U$ et $z \in OUVERTS_{i-1}$. Quand le sommet x est fermé à la i ème étape, il existe un chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z passant par x . L'algorithme prend en compte la valeur de ce chemin pour calculer $d_i(z)$, on peut conclure grâce à l'hypothèse de récurrence.

- Soit $xz \in U$ et $z \in FERMES_{i-1}$. Dans ce cas l'algorithme ne fait rien (i.e. $d_i(z) = d_{i-1}(z)$) montrons que c'est justifié. En effet supposons qu'il existe un chemin $\mu = [x_0, x_1, \dots, x_k, x_{k+1} = x, z]$ allant de x_0 à z tel que $\omega(\mu) < d_i(z)$.
Soit $j \in [0, k]$ l'indice du dernier sommet de μ qui ait été exploré avant que z ne soit exploré à l'étape h . Par hypothèse de récurrence $d_h(x_{j+1}) = \omega(\mu[x_0, x_{j+1}])$. Comme les valuations des arcs sont positives, on en déduit : $d_h(x_{j+1}) < d_h(z)$ ce qui contredit l'hypothèse sur j , car l'algorithme aurait dû explorer x_{j+1} avant z .



L'algorithme A*

Si la fonction choix fournit un sommet z vérifiant :

$d(z) + h(z) = \min_{y \in OUVERTS} \{d(y) + h(y)\}$ où $h(y)$ est une information "heuristique" sur la distance qui reste à parcourir.

et si l'instruction : "Ne rien faire" de l'algorithme de Dijkstra est remplacée par :

si $d(z) + \omega(z, y) < d(y)$ **alors**
 $Parent(y) \leftarrow z;$
 $d(y) \leftarrow d(z) + \omega(z, y);$
 Ajout($y, OUVERTS$);

On suppose que cette fonction $h(y)$ est une information disponible en chaque sommet du graphe. L'usage de cet algorithme est adapté au cas où l'on cherche un plus court chemin d'un sommet x_0 à un sommet t . La valeur de $h(x)$ pour un sommet x donné, étant une estimation de la distance qu'il reste à parcourir pour aller de x à t .

Dans les deux instanciations précédentes, le goulot d'étranglement de complexité provient de la gestion de l'ensembles des OUVERTS. Il faut utiliser une structure de données qui permette le calcul du minimum en $O(\log n)$ ou mieux.

Algorithme de Bellman - Ford

Données: un graphe orienté $G = (X, U)$, une fonction de coût
 $\omega : U \rightarrow \mathcal{R}$

Résultat: une arborescence de chemins issus de x_0 ou un circuit négatif

$Parent(x_0) \leftarrow NIL \ \forall y \neq x_0, \ Parent(y) \leftarrow y$

$d(x_0) \leftarrow 0; \ \forall y \neq x_0, \ d(y) \leftarrow \text{infini}$

répéter

$\forall xy \in U$
si $d(y) < d(x) + \omega(xy)$ **alors**
 $d(y) \leftarrow d(x) + \omega(xy)$ et $pere(y) \leftarrow x$

jusqu'à $|X| - 1$ fois

pour chaque $xy \in U$ faire
┌ **si $d(y) < d(x) + \omega(xy)$ alors**
└ **Il existe un circuit négatif**

- Complexité en $O(n.m)$
- Invariant :
 $\forall i, x,$
 $d_i(x)$ = la longueur minimum d'une marche orientée de s à x ayant au plus i arcs.
- A la fin de l'algo la fonction père représente une arborescence des plus courts chemins.

Algorithme de Prim

L'algorithme de Prim que nous avons vu au chapitre sur les arbres recouvrants de poids minimum peut aussi se voir comme un algorithme de plus court chemin. Pour cela il suffit de remarquer que le graphe est non orienté, les arêtes sont valuées et si le sommet à explorer est celui qui vérifie :

$$d(z) = \min_{y \in \text{OUVERTS}} \{d(y)\}$$

et si l'exploration devient :

Explorer(z);

pour Tous les voisins y de z faire

si $y \in \text{FERMES}$ alors

 | Ne rien faire

sinon

si $y \in \text{OUVERTS}$ alors

si $\omega(z, y) < d(y)$ alors

 | $\text{Parent}(y) \leftarrow z$;

 | $d(y) \leftarrow \omega(z, y)$

sinon

 | $\text{Ajout}(y, \text{OUVERTS})$;

 | $\text{Parent}(y) \leftarrow z$;

 | $d(y) \leftarrow \omega(z, y)$

L'algorithme ci-dessus calcule un arbre de poids minimum de G et c'est une implémentation de l'algorithme de Prim. En outre, il est facile de montrer que l'arborescence obtenue est celle des distances minimum (suivant la distance du max) issue de x_0 .

Tous les algorithmes de plus courts chemins précédemment présentés peuvent être vus comme des implémentations d'un algorithme générique, défini comme suit. On considère un graphe $G = (X, U)$ orienté et l'on suppose les arcs munis d'étiquettes, i.e. une valuation $\omega : U \rightarrow T$, où T est un ensemble muni d'une relation d'ordre total \ll , et de deux éléments distingués \top (resp. \perp) un unique élément maximal (resp. minimal).

On étend cette valuation aux chemins comme suit :

$$\mu = [x_1, \dots, x_k], \omega(\mu) = \bigoplus_{i=0}^{i=k-1} \omega(x_i, x_{i+1}).$$

La relation \oplus binaire associative sur T , étant interprétée suivant les exemples, comme :

- l'addition dans le cas usuel où T est l'ensemble des entiers, des rationnels ou des réels,
- le maximum,
- un produit ou la concaténation dans un monoïde,
- ...

La relation d'ordre total \ll s'interprétant comme :

- l'ordre usuel quand T est l'ensemble des entiers,
- l'ordre lexicographique lorsque T est un langage,
- ...

Algorithme de Parcours Générique;

Données: un graphe orienté $G = (X, U)$, une fonction de coût
 $\omega : U \rightarrow T$

Résultat: une arborescence de chemins issus de x_0

$OUVERTS \leftarrow \{x_0\}$; $FERMES \leftarrow \emptyset$;

$Parent(x_0) \leftarrow NIL$; $\forall y \neq x_0$, $Parent(y) \leftarrow y$;

$d(x_0) \leftarrow \perp$ (l'élément minimal de T);

$\forall y \neq x_0$, $d(y) \leftarrow \top$ (l'élément maximal de T);

tant que $OUVERTS \neq \emptyset$ **faire**

$z \leftarrow Choix(OUVERTS)$;

$Ajout(z, FERMES)$;

 Explorer(z);

$Virer(z, OUVERTS)$

Explorer(z) :

pour Tous les voisins y de z **faire**

si $y \in \text{FERMES}$ **alors**

 Ne rien faire

sinon

si $y \in \text{OUVERTS}$ **alors**

si $d(z) \oplus \omega(z, y) < d(y)$ **alors**

$\text{Parent}(y) \leftarrow z$;

$d(y) \leftarrow d(z) \oplus \omega(z, y)$

sinon

$\text{Ajout}(y, \text{OUVERTS})$;

$\text{Parent}(y) \leftarrow z$;

$d(y) \leftarrow d(z) \oplus \omega(z, y)$

Les graphes sans circuits

Extension *Lineaire*(G, S);

Données: Un graphe orienté $G = (X, U)$ sans circuit, S l'ensemble des sources de G

Résultat: Une numérotation des sommets

$OUVERTS \leftarrow \{S\}$; $FERMES \leftarrow \emptyset$; $compteur \leftarrow 1$;

tant que $OUVERTS \neq \emptyset$ **faire**

$z \leftarrow \text{Choix}(OUVERTS)$;

$numero(z) \leftarrow compteur$; $compteur \leftarrow compteur + 1$;

$Ajout(z, FERMES)$;

 Explorer(z);

$Retrait(z, OUVERTS)$;

Explorer(z);

pour *Tous les successeurs y de z* **faire**

┌ $d^-(y) \leftarrow d^-(y) - 1$
└ **si** $d^-(y) = 0$ **alors**
 └ Ajout(y , OUVERTS)

Cette procédure fournit une numérotation des sommets qui est une extension linéaire de G . Si l'on gère l'ensemble des sommets OUVERTS comme une File, l'extension linéaire obtenue est appelée numérotation par niveau, ou par rang.

Plus courts chemins dans les graphes sans circuit

```
pour  $i = 1$  à  $n$  faire  
┌ pour  $j = i + 1$  à  $n$  faire  
└  $d(j) = \min(d(j), d(i) + \omega(ij))$   
  
┌ si modification de  $d(j)$  alors  
└  $Parent(j) \leftarrow i$ 
```

L'algorithme fonctionne si la numérotation des sommets est une extension linéaire du graphe.

- Graphes orientés considérés comme des relations binaires.
- Etant donné $G = (X, U)$ un graphe orienté, calculer $G^t = (X, U^t)$ le graphe de la plus petite relation transitive contenant U .

Propriété

$xy \in U^t$ ss'il existe un chemin de x à y dans G

2 procédures en $O(n^3)$

Algo A

```
pour  $i = 1$  à  $n$  faire  
  pour  $j = 1$  à  $n$  faire  
    pour  $k = 1$  à  $n$  faire  
      si  $A[i, j] = 1$  et  $A[j, k] = 1$  alors  
         $A[i, k] \leftarrow 1$ 
```

Algo B

```
pour  $i = 1$  à  $n$  faire  
  pour  $j = 1$  à  $n$  faire  
    pour  $k = 1$  à  $n$  faire  
      si  $A[j, i] = 1$  et  $A[i, k] = 1$  alors  
         $A[j, k] \leftarrow 1$ 
```

L'un des deux calcule la fermeture transitive

A ou B ?

La deuxième procédure est connue sous le nom d'algorithme de Roy-Warshall

L'invariant :

A la fin de la boucle en i ,

Pour tout $i' \leq i$, le graphe G_i n'admet pas de triplet du type $ji'eti'k \in G_i$ mais $jk \notin G_i$.

Via des produits de matrices

Soit A la matrice d'incidence de G .

- $A^2[i, j] = 1$ ss'il existe un chemin de longueur exactement 2 entre i et j dans le graphe.
- $(A + A^2)[i, j] = 1$ ss'il existe un chemin de longueur ≤ 2 entre i et j dans le graphe.
- Il suffit donc de calculer si l'on considère la fermeture réflexo-transitive (on ajoute toutes les boucles)

$$I + A + \dots + A^{n-1} = (I + A)^{n-1}$$

- Calcul de X^n
- Le meilleur algorithme de produit de Matrices
- $O(n^\alpha \log n)$

Produit de matrices

- Considérons deux matrices carrées (n, n) , A, B . Leur produit $C = A.B$ vérifie :
- $C[i, j] = \bigoplus_{k=1}^{k=n} A[i, k] \otimes B[k, j]$
- Les opérateurs \bigoplus et \otimes doivent être associatifs et distributifs.

Algèbres tropicales

Dans les applications ces opérateurs \oplus et \otimes , vont être surchargés comme suit :

- Existence d'un chemin, fermeture transitive

Opérateurs logiques : $\oplus = \vee$, $\otimes = \wedge$

- Calcul du nombre de chemins :

Opérateurs : $\oplus = +$ et $\otimes = \times$

- Calcul des plus courts chemins

Opérateurs : $\oplus = \min$ et $\otimes = +$

(avec la convention $A[i, j] = a_{ij}$ la valuation de l'arc ij s'il existe et ∞ si ij n'est pas un arc de G)