Théorie des graphes - Résumé Juin 2010

Dubuc Xavier

7 décembre 2018

Table des matières

1 Théorie de la complexité

La première idée qui vient à l'esprit pour évaluer et comparer des algorithmes est de les programmer, puis de mesurer leurs durées d'exécution. En fait, le temps de calcul est une mesure imparfaite car elle dépend trop de la machine, du langage de programmation, du compilateur et des données. On préfère en pratique compter le nombre d'opérations caractéristiques de l'algorithme à évaluer : ce nombre est indépendant de la machine et du langage. Evalué dans certaines conditions, on l'appelle complexité de l'algorithme. La complexité d'un algorithme A est une fonction non décroissante $f_A(k)$, donnant le nombre d'instructions caractéristiques exécutées par A dans le pire cas, pour une donnée de taille k. La taille d'une donnée est la quantité de mémoire pour la stocker. Pour être rigoureux, il faudrait la mesurer en nombre de bits. On se limitera à compter le nombre de mots-mémoire nécessaires. La nature de la fonction $f_A(k)$, et particulièrement son comportement asymptotique, donne une indication importante sur la manière dont l'algorithme A permet de résoudre le problème considéré. Selon que $f_A(k)$ est de type (ou d'ordre) polynomial en k ou exponentiel en k, le temps de calcul de A évoluera de manière totalement différente.

Il apparaît illusoire de résoudre de manière exacte un problème de très grande dimension pour lequel le meilleur algorithme connu (complexité d'un problème) aurait une complexité en temps non polynomiale. La distinction polynomiale ou non reste donc essentielle pour analyser l'évolution du temps de résolution d'un algorithme en fonction de la dimension du problème. Une classification grossière mais reconnue distingue les algorithmes polynomiaux, des autres, dits exponentiels. **Un bon algorithme est polynomial**. La puissance de la machine ne résoud rien pour des complexités exponentielles. En particulier, il faut se méfier des méthodes énumératives consistant à énumérer tous les cas possibles. Elles conduisent à des algorithmes exponentiels. Le caractère, pire cas, de la théorie de la complexité ne donne pas toujours une image fidèle de l'efficacité réelle d'un algorithme. Malgré cette faiblesse, ce type d'analyse présente deux avantages : elle fournit une borne supérieure du temps de résolution et elle est indépendante des données de l'exemple à traiter. Une complexité moyenne serait plus difficile à calculer, il faudrait se livrer à une analyse probabiliste.

1.1 Les problèmes

Un problème d'optimisation combinatoire consiste à chercher le minimum s^* d'une application f, le plus souvent à valeurs entières ou réelles, sur un ensemble fini S. f est la fonction économique ou fonction objectif :

$$f(s^*) = \min_{s \in S} \{f(s)\}$$

Un problème d'existence (ou problème de décision) consiste à chercher dans un ensemble fini S s'il existe un élément s vérifiant une certaine propriété P. Les problèmes d'existence peuvent toujours être formulés par un énoncé et une question à réponse Oui-Non. Les problèmes d'existence peuvent être considérés comme des problèmes d'optimisation particuliers : on définit la fonction objectif :

$$f : S \mapsto \{0, 1\}$$
$$f(s) = 0 \Leftrightarrow s \text{ vérifie } P \textit{ (en minimisation)}$$

Il existe aussi un problème d'existence associé à tout problème d'optimisation. Il suffit d'ajouter à la donnée S et f un entier k. La propriété P est alors $f(s) \leq k$. Un problème d'optimisation est au moins aussi difficile que le problème d'existence associé ; mais pas beaucoup plus ! Si le problème d'existence associé à un problème d'optimisation est difficile, le problème d'optimisation l'est alors au moins autant. Si on dispose d'un algorithme efficace pour le problème d'existence, on peut l'utiliser un nombre polynomial de fois pour résoudre le problème d'optimisation, par dichotomie sur k.
On connaît pour certains problèmes d'optimisation combinatoire des algorithmes efficaces, c'est-à-dire polynomiaux. Ces problèmes sont dits faciles.
Pour d'autres problèmes, on n'a pas réussi à trouver d'algorithmes polynomiaux. On ne dispose pour les résoudre de manière optimale que d'algorithmes exponentiels, sortes d'énumérations complètes améliorées. Ces problèmes sont dits difficiles.

La théorie de la complexité ne traite que des problèmes d'existence, car leur réponse Oui-Non, Vrai-Faux a permis de les étudier avec des outils de la logique mathématique. Ceci n'est pas très gênant car, comme tout problème d'optimisation est au moins aussi difficile que son problème d'existence associé, tout résultat établissant la difficulté d'un problème d'existence concerne à fortiori le problème d'optimisation.

La **classe** $\mathcal{P}$ est formée de l'ensemble des problèmes d'existence qui admettent des algorithmes déterministes polynomiaux. Cette classe inclut tous les problèmes d'existence associés aux problèmes d'optimisation faciles.
Un algorithme déterministe est une séquence d'opérations à exécuter dans un ordre déterminé ; un tel algorithme c'est donc rien d'autre qu'un programme informatique constitué d'une suite finie d'instructions correspondant à des opérations élémentaires. La théorie de la complexité se concentre sur les problèmes d'existence qui ont une importance pratique. Le critère de praticabilité retenu est de pouvoir vérifier en temps polynomial une proposition de réponse Oui.
La **classe** $\mathcal{NP}$ est celle des problèmes d'existence dont une proposition de solution Oui est vérifiable polynomialement. On dit également que la classe $\mathcal{NP}$ est celle des problèmes d'existence pour lesquels il existe un algorithme non déterministe de temps polynomial. Tout problème d'existence avec un algorithme polynomial est dans $\mathcal{P}$, et donc dans $\mathcal{NP}$. $\mathcal{NP}$ inclut donc la classe $\mathcal{P}$.
Pour un problème d'existence sans algorithme efficace, il faut faire les choses suivantes pour prouver l'appartenance à $\mathcal{NP}$:

— proposer un codage de la solution (certificat) ;
— proposer un algorithme qui va vérifier la solution au vu des données et du certificat ;
— montrer que cet algorithme a une complexité polynomiale.

Exemple :

Considérons le problème d'empilement : étant donné un ensemble S de n nombres entiers et un entier b, existe-t-il un sous-ensemble T de S dont la somme des éléments soit égale à b ? On ne connaît pas d'algorithme polynomial pour résoudre ce problème. Il n'empêche qu'il est dans $\mathcal{NP}$. Un certificat de solution Oui peut être une liste des éléments trouvés pour T . L'algorithme de vérification consiste à vérifier que les entiers de cette liste correspondent effectivement à des entiers de S, et à vérifer que leur somme vaut bien b. Cette vérification est possible polynomialement : le problème est bien dans $\mathcal{NP}$. Les problèmes qui ne sont pas dans $\mathcal{NP}$ existent, mais ne présentent qu'un intérêt théorique pour la plupart. Des exemples sont fournis par des jeux comme les échecs, les dames et le go.

Un problème d'existence P_1 se transforme polynomialement en un autre P_2 s'il existe un algorithme polynomial A transformant toute donnée pour P_1 en une pour P_2 , en conservant la réponse Oui ou Non. On note une telle relation $P_1 t_p P_2$; elle est réflexive et transitive, et constitue un préordre. A l'évidence, si $P_1 t_p P_2$, le problème P_2 est au moins aussi difficile que le problème P_1 . Plus précisément :

$$P_1 t_p P_2 \quad P_2 \in \mathcal{P} \Rightarrow \quad P_1 \in \mathcal{P}$$
$$P_1 t_p P_2 \quad P_1 \notin \mathcal{P} \Rightarrow \quad P_2 \notin \mathcal{P}$$

De ce préordre, on déduit une relation d'équivalence : P_1 est équivalent (du point de vue complexité en temps) à P_2 si et seulement si $P_1 t p P_2$ et $P_2 t p P_1$. Sur les classes d'équivalence engendrées par cette relation, il existe donc un ordre. La classe $\mathcal{P}$ forme la classe d'équivalence des problèmes les plus simples au sein de $\mathcal{NP}$. Les problèmes $\mathcal{NP}$-complets sont les problèmes les plus difficiles de $\mathcal{NP}$, le "noyau dur" en quelque sorte. La classe des problèmes $\mathcal{NP}$-complets est la classe d'équivalence qui est l'élément maximal selon l'ordre induit par t_p , c'est-à-dire formée des problèmes les plus difficiles (du point de vue complexité en temps) au sein de $\mathcal{NP}$. Un problème $\mathcal{NP}$-complet est un problème de $\mathcal{NP}$ en lequel se transforme polynomialement tout autre problème de $\mathcal{NP}$. La classe des problèmes $\mathcal{NP}$-complets est notée $\mathcal{NPC}$. Dans la littérature on rencontre le terme $\mathcal{NP}$-difficile pour les problèmes d'optimisation : un problème d'optimisation combinatoire est $\mathcal{NP}$-difficile si le problème d'existence associé est $\mathcal{NP}$-complet. La technique utilisée pour prouver qu'un problème X de $\mathcal{NP}$ est $\mathcal{NP}$-complet consiste à montrer qu'un problème $\mathcal{NP}$-complet connu Y peut se transformer polynomialement en X (et pas le contraire !).

Cook a montré en 1971 que le problème de satisfiabilité est $\mathcal{NP}$-complet : tout autre problème de $\mathcal{NP}$ peut s'y transformer polynomialement. Le problème de satisfiabilité s'énonce comme suit : Les données du problème consistent en n variables booléennes x_i et un ensemble de m clauses C_i (unions de variables complémentées ou non, comme $x_1 \vee \bar{x_3} \vee x_7$). *La question à se poser est donc : peut-on affecter à chaque variable une valeur (vrai ou faux) de façon à rendre vraies toutes les clauses ?*

Une question cruciale est de savoir si les problèmes $\mathcal{NP}$-complets peuvent être résolus polynomialement, auquel cas $\mathcal{P} = \mathcal{NP}$, ou bien si on ne leur trouvera jamais d'algorithmes polynomiaux, auquel cas $\mathcal{P} \neq \mathcal{NP}$. Cette question n'est pas définitivement tranchée. la découverte d'un algorithme polynomial pour un seul problème $\mathcal{NP}$-complet permettrait de les résoudre facilement tous. Comme des centaines de problèmes N P-complets résistent en bloc à l'assaut des chercheurs, on conjecture que $\mathcal{P} \neq \mathcal{NP}$. La preuve de cette conjecture nécessitera probablement l'invention de nouvelles mathématiques.Dans l'ensemble des

problèmes $\mathcal{NP}$, on trouve les classes d'équivalences extrêmes $\mathcal{P}$ et $\mathcal{NPC}$. Entre les deux s'étend le marais des problèmes à statut indéterminé. Un exemple de problème du marais est celui de l'isomorphisme de 2 graphes. Deux graphes G et H sont isomorphes si on peut transformer l'un en l'autre par renumérotation des sommets. Personne n'a réussi à montrer que ce problème est $\mathcal{NP}$-complet.

A un problème d'existence P, il est possible d'associer un problème d'existence complémentaire $\bar{P}$ en «inversant» la question posée, c'est-à-dire en s'interrogant sur la non-existence d'une solution admissible vérifiant une certaine propriété. La classe $\mathcal{CoNP}$ des problèmes d'existence est formée des problèmes P dont le complément $\bar{P}$ appartient à la classe $\mathcal{NP}$. Si un problème appartient à $\mathcal{CoNP}$, il existe donc un algorithme non déterministe de temps polynomial pour vérifier que sa réponse est non.

2 Définitions de base

- Un graphe orienté est défini par un couple $G = (X, U)$ de 2 ensembles : X est un ensemble fini de sommets et U est une famille de couples ordonnés de sommets appélés arcs *(ou flèches)*.
- Un graphe orienté simple (ou *1-graphe*) est un graphe qui possède au plus un arc dans chaque sens entre deux sommets, un graphe orienté multiple (ou *p-graphe*) est un graphe qui peut posséder plusieurs arcs dans le même sens. Dans un *p-graphe*, un couple de sommets ne peut apparaître plus de p fois.
- Un graphe pondéré ou valué est muni de **poids** ou **coûts** sur ses arcs par application $C : U \mapsto \mathbb{R}$. On le note $G = (X, U, C)$.
- Un poids peut représenter toute valeur numérique associée à un arc : distance kilométrique, coût de transport, temps de parcours, ...
- Une boucle est un arc reliant un sommet à lui même.
- Soit un arc $u = (x, y)$: x est l'origine et y est l'extrémité, on dit aussi que y est successeur de x et x est prédecesseur de y. On dit également que u est incident intérieurement à y et incident extérieurement à x ; deux arcs sont adjacents s'ils ont **au moins** un sommet en commun. On introduit également les notations suivantes :

Notation	Sémantiques
$w^+(x)$	*l'ensemble des arcs incidents extérieurement à x*
$w^-(x)$	*l'ensemble des arcs incidents intérieurement à x*
$w(x)$	*l'ensemble des arcs incidents à x*
$d^+(x)$	*le nombre d'arcs incidents extérieurements à x (**demi-degré extérieur**)*
$d^-(x)$	*le nombre d'arcs incidents intérieurements à x (**demi-degré intérieur**)*
$d(x)$	*le **degré** de x tel que $d(x) = d^+(x) + d^-(x)$*

- La densité d'un graphe orienté simple est le quotient du nombre d'arcs du graphe par le nombre maximal d'arcs théoriquement possible. Ce nombre peut varier de 0 à 1, il est souvent exprimé en pourcents. Les graphes sont généralement peu denses.
- Un graphe non-orienté est défini par un $G = (X, E)$ telle que X est un ensemble fini de sommets et E est une famille de couples de sommets appelés arêtes. On ne distingue plus les successeurs des prédécesseurs, on parle à présent de voisins et le degré d'un sommet devient le nombre d'arêtes incidentes à ce sommet.
- Un graphe non-orienté simple est un graphe qui possède au plus une arête entre deux sommets et ne possède pas de boucle. Un graphe non-orienté multiple *(ou multigraphe)* est un graphe qui peut posséder plusieurs arêtes entre deux sommets.

3 Parties de graphes

Soit un graphe orienté $G = (X, U)$, soit un ensemble $A \subseteq X$ de sommets et un ensemble $V \subseteq U$ d'arcs, alors on dit :

- Le sous-graphe de G engendré par A est le graphe possédant commes sommets ceux de l'ensemble A et comme arcs les arcs dont les deux extrémités sont dans A.
- Le graphe partiel de G engendré par V est le graphe $G' = (X, V)$.

4 Connexité

Un graphe est connexe s'il existe une chaîne entre toutes paires de sommets. Si le graphe n'est pas connexe, on peut identifier plusieurs sous-graphes connexes maximaux au sens de l'inclusion, appelés composantes connexes. Un point d'articulation est un sommet qui augmente le nombre de composantes connexes si on l'enlève. Un isthme est un arc ou une arête qui possède cette même propriété. Un graphe est k-connexe s'il a k chaînes disjointes (sans sommets en commun, sauf aux extremités) entre toute paire de sommets. On dit qu'un graphe orienté est fortement connexe si pour tout paire $\{x, y\}$ de noeuds distincts il existe un chemin de x à y et un chemin de y à x, cette propriétée orientée étant plus forte que la connexité (un graphe orienté connexe n'est pas forcément fortement connexe).

4.1 Composantes connexes

Cet algorithme a la même complexité qu'une seule exploration. L'examen des arêtes est réparti sur les explorations successives. On initialise les marques seulement une fois au début.

```
NCC <- 0
Initialiser CC à 0
Pour s allant de 1 à N faire
   Si CC[s] = 0 alors
      NCC <- NCC + 1
      CC[x] <- NCC
```

4.2 Composantes fortement connexes

Cet algorithme simple lance 2 explorations. Sa complexité est en $O(sommets \times arcs)$, en effet, une exploration peut marquer tous les sommets et l'autre un seul.

```
Construire H avec BuildPreds
NSCC <- 0
Initialiser le tableau SCC à 0
Pour s allant de 1 à N
   Si SCC[s] = 0 alors
      Lancer une exploration au départ de s dans G
      Lancer une exploration au départ de s dans H
      NSCC <- NSCC + 1
      Faire SCC[x] <- NSCC pour tout x marqué dans les 2 explorations.
```

L'algorithme suivant est un algorithme astucieux pour extraire toutes les composantes fortement connexes en seulement $O(arcs)$, cette complexité résulte du partage des arcs entre des explorations élémentaires. L'algorithme utilise deux piles, P et Q, il stocke les *numéros de visite* de chaque sommet dans un tableau **DFN***(Depth First Number*. Un second tableau **Low** est utilisé et il est tel que pour tout sommet x, $Low[x]$ soit le *plus petit numéro de visite* des sommets déjà détectés dans la composante de x.

```
Initialiser P et Q à vide, NSCC, Count et DFN à 0
Pour s allant de 1 à N faire
   Si DFN[s] = 0 alors
      // s non visité --> on lance une exploration
      Count <- Count + 1
      DFN[s] <- Count
      Low[s] <- Count
      Next <- Head
      Push(P,s)
      Push(Q,s)
      Répéter
         x <- Front(Q)
         Si Next[x] = Head[x+1] alors    // La descendance de x a été visitée
            Si Low[x] = DFN[x] alors   // x est une entrée de composante
               NSCC <- NSCC + 1
               Répéter
```

```
            Pop(P,y)
            SCC[y] <- NSCC
         Jusqu'à ce que y = x
      Pop(Q,x)
      Si Q est non-vide alors
         Low[Front(Q)] <- Min(Low[Front(Q)],Low[x])
   Sinon
      y <- Succ[Next[x]]
      Next[x] <- Next[x] + 1
      Si DFN[y] = 0 alors // y non marqué (x,y) arc de liaison
         Count <- Count + 1
         DFN[y] <- Count
         Low[y] <- Count
         Push(P,y)
         Push(Q,y)
      Sinon Si DFN[y] < DFN[x] et y dans P alors
         // (x,y) transverse dans même composante, ou arc arrière
         Low[x] <- Min(Low[x],DFN[y])
   Jusqu'à ce que Q soit vide.
```

5 Graphes particuliers

— Un graphe orienté est symétrique si l'existence de l'arc (x, y) implique l'existence de l'arc (y, x).
— Un graphe orienté est anti-symétrique si l'existence de l'arc (x, y) implique la non-existence de l'arc (y, x).
— Un graphe est complet si toute paire de sommets est reliée par arc ou une arête. Une clique d'un graphe simple est un sous-graphe complet.
— Un graphe $G = (X, U)$ est biparti si on peut diviser ses sommets en deux sous-ensembles X_1 et X_2 avec aucun arc entre deux sommets de X_1 ou de X_2, on le note $G = (X_1, X_2, U)$.

1. Un graphe est biparti s'il est 2-colorable.
2. Un graphe est biparti s'il n'y a aucun cycle impair (cycle à nombre impair d'arêtes).
3. L'algorithme pour prouver qu'un arbre est biparti se base sur le parcours en largeur ($O(arcs)$), il utilise une file Q de sommets et un tableau $Color$:

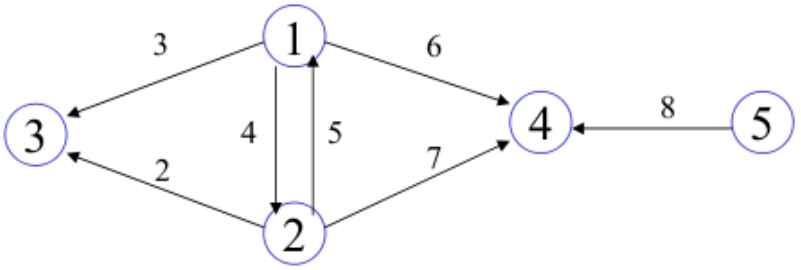
```
Bip <- Vrai
Initialiser le tableau Color à 0
Pour s allant de 1 à N
   Si Color[s] = 0 ET Bip alors
      Q <- {s}
      Color[s] <- 2
      Répéter
         DeQueue(Q,x)
         Pour tout successeur y de x
            Si Color[y] = Color[x] alors   // cycle impair détecté
               Bip <- Faux
            Sinon Si Color[y] <- 0 alors   // si y pas encore marqué
               Color[y] <- 3 - Color[x]   // On alterne les marques 1-2
               EnQueue(Q,y)
      Jusqu'à ce que (Q vide) OU (Non Bip)
```

— Un graphe est planaire si on peut le dessiner dans le plan sans croisement d'arcs. Tous les graphes de moins de 5 sommets sont planaires, ainsi que tous les graphes bipartis de moins de 6 sommets.
— Un arbre est un graphe connexe et sans cycle. Le nombre d'arêtes d'un arbre est égal au nombre de sommets -1, on dit aussi qu'un arbre est un *graphe avec juste ce qu'il faut d'arêtes pour être connexe.*
— Une arborescence est un arbre orienté.
— Une forêt est un graphe sans sycle pas forcément connexe.
— Si un graphe partiel T d'un graphe simple G est un arbre, on dit que T est un arbre recouvrant de G. Il s'agit d'un arbre qui relie tous les noeuds de G, en utilisant uniquement des arêtes de G.

6 Les structures de données

1. **Matrices d'adjacence** *(voir SDD)*, elles ont quelques avantages :
   — elles sont simples et adaptables aux graphes valués,
   — les boucles et la symétrie sont facilement détectables,
   — on peut savoir si un arc (i, j) existe,
   — les prédecesseurs et les successeurs d'un sommet sont facilement obtenables.
   Cependant, elles ont un inconvénient assez conséquent, c'est qu'elles consomment trop de mémoire si le graphe est peu dense. Les algorithmes basés sur ces matrices ne sont pas très efficaces.

2. **Listes d'adjacence**, l'idée ici est de stocker l'ensemble des successeurs pour chaque sommet. On utilise alors un tableau Succ est rempli dans l'ordre des listes des successeurs des sommets *(C'est le tableau des listes de successeurs)*. Pour délimiter ultérieurement ces listes, un tableau Head donne pour tout sommet l'indice dans Succ où comment ses successeurs *(C'est le tableau des têtes des listes)*. Pour les graphes valués, on crée un tableau de poids W en regard de Succ.
   Exemple :



Head :

1	4	7	7	7	8

Succ :

2	3	4	1	3	4	4	

W :

4	3	6	5	2	7	8

Le principal avantage de cette manière de coder est sa compacité, les algorithmes utilisant cette structure de données ont des complexités assez bonnes. Par contre on peut déplorer le test non immédiat de l'existence d'un arc et le passage non immédiat aux précédesseurs (si cette opération est fréquente, il est intêressant de construire également la liste des prédécesseurs.

6.1 Construction de la liste des prédécesseurs

(N est le nombre de sommets, M est le nombre d'arcs)

```
Pour x allant de 1 à N
   InDeg[x] <- 0
Pour k allant de 1 à M
   InDeg[Succ[k]] <- InDeg[Succ[k]] + 1

Head[0] <- 1
Pour x allant de 1 à N
   Head[x] <- Head[x-1]+InDeg[x]

Pour x allant de 1 à N
   Pour k allant de Head[x] à Head[x+1]-1
      y <- Succ[k]
      Head[y] <- Head[y] - 1
      Succ[Head[y]] <- x
```

3. Files & Piles : *voir SDD.*
4. Tas : *voir SDD2* (attention, ici la valeur tout en haut de l'arbre est le minimum, et non le maximum)

7 Décomposition d'un graphe en niveaux

On considère un graphe orienté, connexe et sans circuit, on souhaite classer les sommets dans un tableau **Sorted** de manière à avoir $i < j$ pour tout arc ($Sorted[i]$,$Sorted[j]$), autrement dit, effectuer un tri topologique. L'interêt de ce tri est d'accélérer certains algorithmes, en leur faisant traiter tout sommet avant ses successeurs. Une façon simple d'obtenir le tri topologique est appelée *décomposition en niveaux*.

```
NLayer <- 0
Tant qu'il existe des sommets non classés
   NLayer <- NLayer + 1
   Pour tout sommet x sans prédecesseur
      Layer[x] <- NLayer
   Enlever de G les sommets de niveau Layer
```

La décomposition en niveaux n'a normalement de sens que si le graphe est sans circuit, car un sommet d'un circuit est à la fois descendant et ascendant de lui-même. Si on tente de traiter un graphe avec des circuits, on arrive à une étape où aucun sommet sans prédécesseur ne peut être trouvé. L'algorithme peut donc être adapté pour détecter si un graphe possède au moins un circuit.

```
NLayer <- 0
Q <- vide
Pour x allant de 1 à N
   Si InDeg[x] = 0 alors EnQueue(Q,x)
Tant que Q est non vide
   NLayer <- NLayer + 1
   Pour iter allant de 1 à |Q|
      DeQueue(Q,x)
      Layer[x] <- NLayer
      Pour k allant de Head[x] à Head[x+1] - 1
         y <- Succ[k]
         InDeg[y] <- InDeg[y] - 1
         Si InDeg[y] = 0 alors
            EnQueue(Q,y)
```

8 Exploration de graphes

L'exploration d'un graphe à partir d'un sommet est une opération utilisée dans de nombreux algorithmes de graphes. Partant d'un sommet de départ donné, explorer un graphe c'est déterminer l'ensemble des descendants de ce sommet, c'est-à-dire l'ensemble des sommets situés sur des chemins d'origine de ce sommet. Au début, on marque le seul sommet de départ. Ensuite, on propage les marques dans le graphe. Il est essentiel d'implémenter le processus de propagation avec la meilleure complexité possible.

```
Initialiser Mark à Faux
Mark[s] <- Vrai
Z <- {s}
Next <- Head
Répéter
   Soit un sommet quelconque x de z
   Si Next[x]  = Head[x+1] alors
      Z <- Z -{x}
   Sinon
      y <- Succ[Next[x]]
      Next[x] <- Next[x] + 1
      Si non Mark[y] alors
         Mark[y] <- Vrai
         Z <- Z UNION {y}
Jusqu'à ce que Z soit vide.
```

Il y a 2 façons de gérer l'ensemble Z, soit par une file, on effectue alors un parcours en largeur (ou BFS *(Breadth-First Search)*), soit par une pile, on effectue alors un parcours en profondeur (ou DFS *(Depth-First Search)*).

8.1 Parcours en largeur (DFS)

```
Initialiser Mark à Faux
Mark[s] <- Vrai
ClearSet(Z)
EnQueue(Z,s)
Répéter
   DeQueue(Z,x)
   Pour tout successeur y non marqué de x faire
      Mark[y] <- Vrai
      EnQueue(Z,y)
Jusqu'à ce que SetIsEmpty(Z).
```

8.2 Parcours en profondeur (BFS)

```
Initialiser Mark à Faux
Mark[s] <- Vrai
ClearSet(Z)
Push(Z,s)
Next <- Head
Répéter
   x <- Front(z)
   Si Next[x] >= Head[x+1] alors
      Pop(Z,x)
   Sinon
      y <- Succ[Next[x]]
      Next[x] <- Next[x] + 1
      Si non Mark[y]  alors
         Mark[y] <- Vrai
         Push(Z,y)
Jusqu'à ce que SetIsEmpty(Z).
```

9 Détection de circuit

La décomposition en niveaux peut être modifiée pour détecter un circuit mais pas pour en donner les sommets. L'exploration en longueur, elle, le permet et ce, avec la même complexité.

```
Found <- Faux
Initialiser Mark à Faux
s <- 0
Tant que (non Found) ET (s  < N)
   s <- s + 1
   Si non Mark[s] alors
      Mark[s] <- Vrai
      ClearSet(Z)
      Push(Z,s)
      Next <- Head
      Répéter
         x <- Front(Z)
         Si Next[x] >= Head[x+1] alors
            Pop(Z,x)
         Sinon
            y <- Succ[Next[x]]
            Next[x] <- Next[x] + 1
            Si non Mark[y] alors
               Mark[y] <- Vrai
               Push(Z,y)
            Sinon Si InSet(Z,y) alors Found <- Vrai
      Jusqu'à ce que Found OU SetIsEmpty(Z)
```

10 Arbres et Arborescences

On considère un graphe non orienté simple $G = (X, E)$ avec N sommets et M arêtes, les 6 définitions suivantes désignent un arbre :

1. G est connexe et sans cycle,
2. G est sans cycle et $M = N - 1$,
3. G est connexe et $M = N - 1$,
4. G est sans cycle et l'ajout d'une arête crée un seul cycle,
5. G est connexe et tout arête est un isthme,
6. il existe une et une seule chaîne entre 2 sommets quelconque de G.

10.1 Arbre recouvrant de poids minimal

Soit $G = (X, E, W)$ un graphe simple, connexe et valué sur par des poids sur les arêtes. Le problème de l'arbre recouvrant de poids minimal consiste à trouver un trouver un arbre recouvrant G dont le **poids total des arêtes est minimal** (Si l'arbre n'est pas connexe, on cherche une forêt). Tous les algorithmes pour trouver cet arbre exploitent plus ou moins la propriété suivante :

«Soit $G = (X, E, W)$ un graphe orienté et valué, soit (X_1, X_2) une partition quelconque en deux classes de l'ensemble des sommets de X, ***un arbre recouvrant de poids minimal contiendra toujours l'arête de valeur minimal joignant X_1 et X_2.»***

10.1.1 Algorithme de Prim

On part d'un arbre initial réduit à un seul sommet arbitraire, ensuite, à chaque étape on connecte ce sommet à un voisin avec l'arête de poids minimal. Cet algorithme est insensible à la densité du graphe, sa complexité est en $O(Sommets \times Sommets)$.

```
Weight <- 0
NEdge <- 0
Pour y allant de 1 à N
   Nearest[y] <- 0
   LinkCost <- +Infini
x <- 1
Répéter
   Nearest[x] <- x
   Pour k allant de Head[x] à Head[x+1]-1
      y <- Succ[k]
      Si (Nearest[y] != y)  ET (W[k] < LinkCost[y]) alors
         LinkCost[y] <- W[k]
         Nearest[y] <- x
   WMin <- + infini
   Pour y allant de 2 à N
      Si (Nearest[y] != y)  ET (LinkCost[y] < WMin) alors
         WMin <- LinkCost[y]
         x <- y
   Si WMin < MaxCost
      Weight <- Weight + WMin
      NEdge <- NEdge + 1
      Node1[NEdge] <- x
      Node2[NEdge] <- Nearest[x]
Jusqu'à ce que (WMin = +Infini) OU (NEdge = N-1)
```

10.1.2 Algorithme de Kruskal

Pour cet algorithme, on part d'une forêt d'arbres réduits chacun à un sommet isolé. A chaque itération, on ajoute à cette forêt la plus petite arête ne créant pas de cycle avec celles déjà choisies. On stoppe quand l'arbre est couvrant (graphe connexe) ou quand on ne trouve plus d'arête (graphe non connexe). Contrairement à l'algorithme de Prim, celui de Kruskal trouve directement une forêt couvrante de poids minimal si le graphe est non connexe. La complexité est en $O(aretes \times aretes)$ pour un graphe complet.

10.1.3 Comparaison des 2 algorithmes

Dans l'algorithme de Prim, c'est un arbre qui grandit au fur et à mesure que l'algorithme s'exécute et si le graphe n'est pas connexe, il ne sait pas continuer. Pour l'algorithme de Kruskal, si le graphe n'est pas connexe, vu qu'il part de chaque noeuds, il trouvera une forêt de poids minimal. Cependant ce dernier est très lié à la densité du graphe de par 2 opérations critiques : vérifier s'il y a un cycle et trier les arcs. Pour cela, on utilise le tri par tas pour améliorer la complexité. Ces 2 algorithmes sont des algorithmes gloutons car ils prennent une décision mais ne se remettent jamais en question jusqu'au moment où la solution est trouvée.

10.1.4 Amélioration de l'algorithme de Kruskal

On utilise une structure de données appellée Union-Find, dans ce problème, on considère un ensemble d'éléments partitionné en classe d'équivalence (calculée grâce au tri par tas). Le problème d'Union-Find consiste à suivre les évolutions, en modifiant la partition en classes le plus efficacement. $Find(x)$ est une fonction qui renvoie la classe à laquelle appartient x et $Union(u, v)$ fusionne les deux classes de numéros u et v. Dans notre cas, à chaque sommet on définit une classe (au départ chaque sommet appartient à sa propre classe), ensuite quand on prend un arc reliant 2 sommets, on regarde si les 2 sommets que l'on veut connecter appartiennent à la même classe. Sinon c'est OK pour l'arc et il faut les mettre dans la même classe. L'algorithme consiste donc à chercher le plus petit sommet, on test si c'est OK et si c'est OK on charge la classe de plus petite "taille". A la fin de l'algorithme, si c'est un arbre, tous les sommets appartiennent à la même classe. On va utiliser un tableau **Next** dans lequelle on va indiquer le successeur de chaque sommet dans sa classe (par exemple si 1 et 6 sont dans la même classe et que 6 est le successeur de 1, alors la case 1 de **Next** contiendra 6).

```
Fonction Find(x)
   u <- x
   Tant que Next[u] > 0
      u <- Suivant[u]
   Retourner u

Fonction Union(u,v)
   NNoeuds <- Next[u] + Next[v]
   Si Next[u] > Next[v] alors
      Next[u] <- v
      Next[v] <- NNoeuds
   Sinon
      Next[v] <- u
      Next[u] <- NNoeuds
```

L'algorithme devient donc :

```
Weight <- 0
NEdge <- 0
Inter <- 0
InitClasses
MakeHeap
Répéter
   // Enlève arête de poids minimum [x,y] et récupère classes de x et y
   Iter <- Iter + 1
   HeapMin(x,k)
   y <- Succ[k]
   xClass <- Find(x)
   yClass <- Find(y)
   // Si pas de cycle on garde [x,y] et on fusionne leurs classes
   Si xClass != yClass alors
      Union(xClass,yClass)
      Weight <- Weight + W[k]
      NEdge <- NEdge + 1
```

```
      Node1[NEdge] = x
      Node2[NEdge] = y
Jusqu'à ce que (Iter = M/2) OU (NEdge = M-1)
```

10.1.5 Algorithme d'Edmonds

Cet algorithme permet de trouver l'arborescence de poids minimal, une heuristique fournit un graphe partiel, cette heuristique connecte chaque sommet sauf la racine à son plus proche prédécesseur (en terme de poids) :

```
V <- vide
Pour tout sommet y != root
   Chercher l'arc (x,y) de U de poids minimal
   V <- V UNION {(x,y)}
```

Et l'algorithme général :

```
G' <- G
Répéter
   Calculer pour tout sommet y != root un arc (x,y) de poids minimal
   Soit H le graphe partiel obtenu
   Pour chaque circuit c de H faire
      Contracter G' par rapport à c
      Ajuster les poids des arcs entrants dans c
Jusqu'à ce que H soit sans circuit
Reconstituer l'arborescence de poids minimal sur le graphe initial G.
```

11 Chemins optimaux

Considérons un graphe orienté valué $G = (X, A, W)$ où $W[i, j]$ est la valuation de l'arc (i, j). Le problème consiste à calculer des chemins de coûts minimal (chemins minimaux ou plus courts chemins), ce problème n'a un sens que s'il n'y a pas de circuit de coût négatif (circuit absorbant). En l'absence de circuit absorbant, on peut restreindre la recherche des plus courts chemins aux seuls chemins élémentaires. Le problème de recherche d'un chemin optimal en présence de circuits absorbants existe, mais il est NP-difficile. On distingue 3 types de problèmes de ce genre :

- A : Etant donné deux sommets s et t, trouver un plus court chemin de s à t.
- B : Etant donné un sommet de départ s, trouver un plus court chemin de s vers tout autre sommet.
- C : Trouver un plus court chemin entre tout couple de sommets, c'est-à-dire calculer une matrice (sommets $\times$ sommets) appelée **distancier**.

Le problème A est en général résolu par un algorithme conçu pour résoudre le problème B que l'on stoppe dès que le sommet de destination choici est traité définitivement. Il existe cependant un algorithme pour ce problème mais qui est valable pour les graphes euclidiens, c'est-à-dire des graphes où les sommets sont des points de l'espace euclidien et où les poids sont les distances euclidiennes entre les points.

Le problème B peut se décliner en différents cas :

1. $W =$ ***constante*** : Le problème revient à trouver des plus courts chemins en nombre d'arcs et peut se résoudre par une exploration de graphe en largeur en $O(arcs)$.
2. $W > 0$: On peut le résoudre en $O(sommets \times sommets)$ par un algorithme à fixation d'étiquettes dû à Dijkstra. Avec une structure de tas, on obtient une variante en $O(arcs \times \log sommets)$ intéressante si G est peu dense. Si les coûts sont entiers et si leur valeur maximale U n'est pas trop grande, une structure de données appelée bucket permet une complexité en $O(arcs + U)$ et au pire cas en $O(sommets \times sommets + U)$.
3. W ***quelconque*** : L'algorithme à correction d'étiquettes de Bellman, de type programmation dynamique, résoud ce cas général en $O(arcs \times sommets)$, pire cas en $O(sommets^3)$. Il peut servir à détecter la présence d'un circuit de coût négatif.
4. W ***quelconque et sans circuit*** : L'algorithme de Bellman peut alors se simplifier grâce à une décomposition de G en niveaux de complexité en $O(arcs)$.

Pour le problème C, il existe un algorithme à correction d'étiquettes très simple, dû à Floyd, calculant en $O(sommets^3)$ un distancier. Nécessitant une représentation matricielle du graphe G, il consomme trop de temps de calcul et de mémoire sur des graphes de faible densité. Dans ce cas, il vaut mieux calculer chaque ligne du distancier avec un algorithme pour le problème B.

11.1 Algorithme de Dijkstra

L'itération principale sélectionne le sommet x d'étiquette minimale parmi ceux déjà atteints par un chemin provisoire d'origine s. Pour tout successeur y de x, on regarde si le chemin passant par x améliore le chemin déjà trouvé de s à y : si oui, on remplace $V[y]$ par $Min(V[y], V[x] + W[x, y])$ et on mémorise qu'on parvient en y via x en posant $P[y] = x$. Si tous les sommets sont accessibles au départ de s, l'algorithme prend *sommets* itérations (des sommets peuvent ne pas être accessibles).

11.1.1 Algorithme de base

```
Initialiser le tableau V à +Infini
Initialiser le tableau P à 0
Initialiser le tableau Done à faux
V[s] <- 0
P[s] <- s
Répéter
   VMin <- +Infini
   Pour y allant de 1 à N faire
      Si (non Done[y]) et (V[y] < VMin) alors
         x <- y
         VMin <- V[y]
   Si VMin < +Infini alors
      Done[x] <- Vrai
      Pour k allant de Head[x] à Head [x+1]-1
         y <- Succ[k]
         Si V[x] + W[k] < V[y] alors
            V[y] <- V[x] + W[k]
            P[y] <- x
Jusqu'à ce que VMin = +Infini
```

11.1.2 Version avec tas

L'inconvénient majeur de l'algorithme de Dijkstra est qu'il est insensible à la densité du graphe. Le nombre d'itérations, au plus = *sommets*, ne peut pas être amélioré par construction de l'algorithme. En revanche, l'essentiel du travail est dû à la boucle interne trouvant le prochain sommet à fixer. On suggère d'utiliser un tas pour l'avoir plus rapidement. Le tas contient à toute itération les sommets non fixés de valeur non infinie. L'algorithme est formulé pour traiter les problèmes A et B. Pour le problème B, il faut appeler l'algorithme avec $t = 0$.

```
Initialiser le tableau V à +Infini
Initialiser le tableau P à 0
V[s] <- 0
P[s] <- s
ClearHeap(H)
HeapInsert(H,V,s)
Répéter
   HeapMin(H,V,x)
   Pour k allant de Head[x] à Head [x+1]-1
      y <- Succ[k]
      Si V[x] + W[k] < V[y] alors
         V[y] <- V[x] + W[k]
          P[y] <- x
          Si non InHeap(H,y) alors HeapInsert(H,V,y)
          Sinon MoveUp(H,V,y)
Jusqu'à ce que HeapIsEmpty(H) ou x = t
```

11.1.3 Algorithmes à buckets

Les algorithmes à buckets sont des variantes de l'algorithme de Dijkstra intéressantes quand les coûts des arcs sont entiers et leur valeur maximum U n'est pas très grand. On partitionne l'intervalle des valeurs des étiquettes en B intervalles de largeur commune L, numérotés de 0 à $B-1$, et on associe à chacun d'eux un ensemble de sommets appelé bucket. Ce système est codable comme un tableau **Buck** de B listes. $Buck[k]$ stocke des sommets d'étiquettes entre $k \times L$ et $(k+1) \times L - 1$. Pour trouver x à fixer, on cherche d'abord le bucket non vide de plus petit indice k. On balaie ensuite $Buck[k]$ pour localiser et extraire le sommet x d'étiquette minimale. Le bucket d'un sommet y est $Buck[V[y]/L]$. Les sommets sont en pratique bien répartis et les listes-buckets sont courtes, ce qui donne de très bonnes performances moyennes.

Au pire, tous les sommets vont dans le même bucket, et on serait tenté de déclarer B tableaux de *sommets* éléments pour le système de buckets. En fait, il suffit d'un seul tableau **Next** de *sommets* éléments partagé par les buckets. $Next[i]$ indique le suivant du sommet i dans le même bucket et vaut 0 si i est dernier de son bucket. Le tableau **Buck** sert alors à indiquer le premier sommet de chaque bucket. Il faut aussi définir le prédécesseur, tableau **Prev**, de chaque sommet dans son bucket pour pouvoir réaliser efficacement l'opération d'enlèvement. Les plus simples algorithmes à bucket ont une largeur $L = 1$. Les buckets contiennent des sommets de même valeur, et on accède au bucket d'un sommet x par simple indiçage : $V[x]$. Les buckets sont implémentés par des listes circulaires codées par des tableaux : **First**, **Next** et **Prev**. Voici les primitives pour gérer les buckets :

```
Fonction Initialize(Buckets)
   Pour b allant de 0 à UMax
      First[b] <- 0

Fonction PopFrom(Buckets,b,x)
   x <- First[b]
   Si Next[x] = x alors
      First[b] <- 0
   Sinon
      Next[Prev[x]] <- Next[x]
      Prev[Next[x]] <- Prev[x]
      First[b] <- Next[x]

Fonction PushInto(Buckets,b,x)
   Si First[b] = 0 alors
      Next[x] <- x
      Prev[x] <- x
   Sinon
      Next[x] <- First[b]
      Prev[x] <- Prev[First[b]]
      Next[Prev[x]] <- x
      Prev[Next[x]] <- x
   First[b] <- x

Fonction RemoveFrom(Buckets, b, x)
   Si Next[x] = x alors
      First[b] <- 0
   Sinon
      Next[Prev[x]] <- Next[x]
      Prev[Next[x]] <- Prev[x]
      Si x = First[b] alors
         First[b] <- Next[x]
```

Et l'algorithme avec buckets :

```
Initialize(Buckets)
Initialiser le tableau V à +Infini
Initialiser le tableau P à 0
V[s] <- 0
P[s] <- s
PushInto(Buckets,0,s)   // Mets s dans le bucket 0
LB <- 0
CB <- 0
Done <- Faux
Répéter
   Si First[CB] = 0 alors   // Cherche bucket non-vide
      Répéter
         CB <- (CB+1) ET UMax
      Jusqu'à ce que (First[CB] > 0) OU (CB = LB)
      Si CB = LB alors
         Done <- Vrai   // Tour complet -> Fin.
   Si First[CB] > 0 alors
      LB <- CB
      PopFrom(Buckets,CB,x)   // Enleve le premier sommet bucket CB
      Pour k allant de Head[x] à Head[x+1] - 1
         y <- Succ[k]
         Si V[x] + W[k] < V[y] alors
            Si P[y] > 0 alors
               RemoveFrom(Buckets,(CB+V[y]-V[x]) ET UMax,y)   // Enleve y de son bucket actuel
            PushInto(Buckets,(CB+W[k]) ET UMax,y) // Insère y en tête de son nouveau bucket
            V[y] <- V[x] + W[k]
            P[y] <- x
Jusqu'à ce que Done ou x = t.
```

11.2 Algorithme de Bellman

Cet algorithme à correction d'étiquettes est prévu pour des valuations quelconques et peut être adapté pour détecter un circuit de coût négatif. Il s'agit d'une méthode de programmation dynamique, c'est-à-dire d'optimisation récursive, décrite par les relations de récurrence suivante :

$$\begin{cases} V_0(s) = 0 & \\ V_0(y) = +\infty & y \neq s \\ V_k(y) = \min_{x \in \Gamma^{-1}(y)} \{V_{k-1}(x) + W(x,y)\} & k > 0 \end{cases}$$

Les deux premières relations servent à stopper la récursion. Le sommet s peut être considéré comme un chemin de 0 arc et de coût nul. La troisième relation signifie qu'un chemin optimal de k arcs de s à y s'obtient à partir des chemins optimaux de $k-1$ arcs de s vers tout prédécesseur x de y. En effet, tout chemin optimal est formé de portions optimales, sinon on pourrait améliorer le chemin tout entier en remplaçant une portion non optimale par une portion plus courte.

En pratique, on calcule le tableau des étiquettes V itérativement. Les étiquettes en fin d'étape k sont calculées dans un tableau V_{new} à partir du tableau V des étiquettes disponibles en début d'étape. Pour tout sommet y, on regarde si $V[y]$ est améliorable en venant d'un prédécesseur de y. En fin d'étape on écrase V par V_{new} et on passe à l'itération suivante. On peut accélèrer la méthode en ne travaillant que dans le seul vecteur V. En l'absence de circuit absorbant, on peut se restreindre aux chemins élémentaires pour trouver un plus court chemin de s vers tout autre sommet. Un tel chemin n'a pas plus de $(sommets - 1)$ arcs. Les étiquettes sont donc stabilisées en au plus $(sommets - 1)$ itérations. En pratique, elles peuvent se stabiliser plus tôt, et un meilleur test est de voir si les étiquettes à la fin de l'itération k sont identiques à celles de l'itération $k-1$. La complexité est en $O(sommets \times arcs)$.

Pour détecter un circuit négatif, on fait *sommets* itérations : si alors les étiquettes à la fin de l'itération *sommets* sont différentes des étiquettes à la fin de l'itération (*sommets* − 1), il y a un circuit. Dans l'algorithme, le booléen **Stable** est mis à faux quand on détecte une différence entre V et V_{new}, ce qui signifie que les étiquettes ne sont pas encore stabilisées.

11.2.1 Algorithme de base

```
Initialiser V à +Infini et P à 0
V[s] <- 0
P[s] <- s
k <- 0
Répéter
   k <- k+1
   Initialiser VNew à +Infini
   Stable <- Vrai
   Pour y allant de 1 à N
      Pour tout x prédecesseur de y
         Si (V[x] != +Infini) et (V[x]+W(x,y) < VNew[y]) alors
            VNew[y] <- V[x] + W(x,y)
            P[y] <- x
            Stable <- Faux
      V <- VNew
Jusqu'à ce que Stable OU (k = N)
```

11.2.2 Algorithme FIFO

Cet algorithme à correction d'étiquettes examine les sommets dans l'ordre **FIFO** grâce à une file Q de sommets. L'algorithme se termine quand la file Q est vide. L'algorithme **FIFO** est un dérivé de l'algorithme de Bellman, très intéressant car n'utilisant pas les prédécesseurs. Pour détecter un circuit négatif, il suffit de compter les itérations : il y a un circuit négatif si la file Q n'est pas vide après *sommets* itérations. La complexité est la même que celle de l'algorithme de Bellman.

```
Initialiser V à +Infini et P à 0
V[s] <- 0
P[s] <- s
ClearSet(Q)
EnQueue(Q,s)
Répéter
   Pour Iter allant de 1 à CardOfSet(Q)
      DeQueue(Q,x)
      Pour k allant de de Head[x] à Head[x+1]-1
         y <- Succ[k]
         Si V[x] + W[k] < V[y] alors
            V[y] <- V[x] + W[k]
            P[y] <- x
            EnQueue(Q,y)
Jusqu'à ce que SetIsEmpty(Q)
```

11.2.3 Algorithme d'Esopo et Pape

L'algorithme de D'Esopo et Pape utilise une pile-file **Next**. Il s'agit d'une file où on peut ajouter un élément à la fin ou en tête. Comme dans **FIFO**, un sommet atteint la première fois est mis en fin de file. Par contre, si on le revisite, on l'insère en tête de file (à condition qu'il ne soit pas déjà en file). Ce critère heuristique s'explique intuitivement. Quand on atteint pour la première fois un sommet, il n'est pas urgent de développer ses successeurs car les chemins obtenus risquent d'être mauvais dans un premier temps. En revanche, un sommet déjà visité et dont l'étiquette vient de diminuer doit être développé en priorité, pour propager l'amélioration. Cet algorithme n'est pas polynomial cependant il est très rapide sur les graphes peu denses.

```
Initialiser V à +Infini et P à 0
V[s] <- 0
P[s] <- s
ClearSet(Next)
EnQueue(Next,s)
Répéter
   DeQueue(Next,x)
   Pour k allant de Head[x] à Head[x+1] - 1
      y <- Succ[k]
      Si V[x] + W[k] < V[y] alors
         V[y] <- V[x] + W[k]
         Si P[y] = 0 alors   // y n'a jamais été dans Next --> on l'ajoute en fin de file
            EnQueue(Next,y)
         Sinon Si non InSet(Next,y) alors // y n'est plus dans Next --> on l'ajoute en tête
            Push(next,y)
         P[y] <- x
Jusqu'à ce que SetIsEmpty(Next)
```

11.2.4 Graphes sans circuits

Si on décompose le graphe en niveaux, on peut calculer les étiquettes définitivement par numéro croissant de niveau. En effet, tout sommet y a ses prédécesseurs dans les niveaux inférieurs et les étiquettes des ces prédécesseurs sont prêtes quand on calcule $V[y]$:

```
Initialiser le tableau V à +Infini et le tableau P à 0
V[s] <- 0
P[s] <- s
Pour i allant de 1 à N
   y <- Sorted[i]   // Sommets par numéro croissant de niveau
   Pour tout prédecesseur x de y
      Si (V[x] != +Infini) ET (V[x] + W(x,y) < V[y]) alors
         V[y] <- V[x] + W(x,y)
         P[y] <- x
```

Cet algorithme est en $O(arcs)$, en fait, on n'a même pas besoin des prédécesseurs : pour tout sommet x (par niveau croissant), on peut procéder en améliorant les étiquettes des successeurs. Cette version est aussi en $O(arcs)$ mais nécessite les vecteurs **Sorted** et **Layer** (numéro de niveau de chaque sommet).

```
Initialiser le tableau V à +Infini et le tableau P à 0
V[s] <- 0
P[s] <- s
Pour tout successeur x de s
   V[x] <- W(s,x)
Pour i allant de 1 à N
   x <- Sorted[i]
   Si Layer[x] > Layer[s] alors
      Pour tout successeur y de x tel que V[x] + W(x,y) < V[y]
         V[y] <- V[x] + W(x,y)
         P[y] <- x
```

11.3 Algorithme de Floyd

L'algorithme de Floyd calcule un distancier ($sommets \times sommets$) donnant les valeurs des plus courts chemins entre tout couple de sommets (problème C). Pour cet algorithme, le tableau V des étiquettes devient une matrice. $V[i,j]$ désigne le coût des plus courts chemins de i à j. Au début, la matrice est initialisée avec les coûts des arcs. A la première itération, la matrice contient les coûts des plus courts chemins ayant le seul sommet 1 comme sommet intermédiaire. A l'itération k, la matrice donne le coûts des plus courts chemins dont les sommets intermédiaires sont dans l'ensemble $\{1, 2, ..., k\}$. A l'itération $sommets$, la matrice fournit le distancier désiré. La complexité est due aux trois boucles emboîtées en $O(sommets^3)$. L'algorithme est insensible à la densité du graphe, mais il détecte les circuits négatifs ($V[i,i] < 0$).

```
Initialiser la matrice V à +Infini
Initialiser la diagonale de V à 0
Initialiser la matrice P à 0
Pour i allant de 1 à N
   P[i,i] <- i
   Pour a allant de Head[i] à Head[i+1] - 1 tel que Succ[a] != i
      j <- Succ[a]
      V[i,j] <- W[a]
      P[i,j] <- i
Pour k allant de 1 à N
   Pour i allant de 1 à N
      Pour j allant de 1 à N
         Si (V[i,k] != +Infini) ET (V[k,j] != +Infini) ET (V[i,k]+V[k,j] < V[i,j]) alors
            V[i,j] <- V[i,k] + V[k,j]
            P[i,j] <- P[k,j]
```

12 Ordonnancement et plus longs chemins

Considérons le problème d'ordonnancement de projet défini par un ensemble de tâches de durées connues. Ce projet doit être exécuté en présence de contrainte d'enchaînement, de façon à minimiser la date d'achèvement du projet. Les contraintes d'enchaînement sont données par un graphe valué $G = (X, A, P)$. Un arc (i, j) relie la tâche i à la tâche j si i doit être terminée avant de démarrer j. L'arc est valué par la durée minimale devant séparer les dates de début des deux tâches. Un tel graphe est dit potentiels-tâches et il est sans circuit. Il est d'usage d'inclure deux tâches fictives de durées nulles, désignant le début et la fin du projet.

Le problème d'ordonnancement de projet revient à minimiser la date de début au plus tôt de la tâche fictive de fin du projet. La date de début au plus tôt de toute tâche i est la durée du plus long enchaînement de tâches du début à i. Le plus long chemin trouvé du début à la fin du projet, responsable de la durée totale du projet, est appelé chemin critique. Un retard sur les tâches dites critiques de ce chemin se répercute sur le projet tout entier. Pour toute tâche non critique, on peut s'offrir un petit retard sans affecter la durée totale du projet, du moins jusqu'à une date de début au plus tard.

13 Parcours

Un chemin de longueur q, est une séquence de q arcs, un chemin fermé est appelé circuit, dans les graphes non-orientés on parle de chaînes de longueur q pour une séquence de q arêtes, une chaîne fermée est appelée un cycle. Finalement, le terme de parcours regroupe les chemins, circuits, chaîne et arête, on dit qu'un parcours est élémentaire s'il n'emprunte qu'une seule fois ses sommets.

1. Un parcours est dit eulérien s'il passe exactement une fois par chaque arc ou arête du graphe (il peut passer plusieurs fois par le même sommet),
2. Un parcours est dit hamiltonien s'il passe exactement une fois par chaque sommet du graphe.

Contrairement aux parcours eulériens, on ne connaît pas d'algorithme efficace pour savoir dans tous les cas si un graphe admet un parcours hamiltonien.

Théorème d'Euler : Soit un graphe non orienté $G = (X, E)$. Il admet un parcours eulérien si et seulement s'il est connexe et a 0 ou 2 sommets de degré impair. S'il a 0 sommet impair, G admet un cycle eulérien et on peut partir d'un sommet quelconque pour y revenir. S'il a 2 sommets impairs s et t, G admet une chaîne eulérienne joignant s et t.

Démonstration :
Un parcours traverse un sommet x si, en le suivant, on emprunte deux arêtes consécutives d'extrémité x. La traversée de x diminue de 2 son degré dans le graphe partiel des arêtes à visiter. S'il existe un parcours eulérien, le graphe est forcément connexe. Si le parcours est un cycle, il traverse au moins une fois chaque sommet, et tous les sommets sont pairs. Si le parcours est une chaîne ouverte, elle ne peut joindre que deux sommets impairs, car les sommets intermédiaires sont tous pairs. Réciproquement, supposons que le graphe soit connexe et ait 0 ou 2 sommets impairs s et t. On ramène le second cas au premier en ajoutant

un sommet z de degré 2 relié à s et t. On montre que ce graphe admet un cycle eulérien avec un procédé constructif.

Construisons à partir de z, aussi longtemps que possible, un parcours ne repassant jamais par une arête déjà visitée. Comme le graphe est connexe, les sommets sont tous pairs, et comme il reste une arête incidente à z à visiter, le parcours va finir en z. Si on emprunte toutes les arêtes, on a un cycle eulérien. Sinon, le graphe partiel des arêtes non visitées a tous ses sommets pairs mais il n'est pas forcément connexe. Comme le graphe est connexe, chaque composante connexe du graphe partiel a au moins un sommet sur le parcours. Pour chaque composante connexe du graphe partiel on applique le même procédé au départ du sommet sur le parcours et on greffe le cycle obtenu sur le parcours. En répétant le procédé tant qu'il reste des arêtes à visiter, on finit par rendre le parcours eulérien. Ce théorème s'applique aussi bien à des graphes qu'à des multigraphes.

Le procédé constructif de la démonstration du théorème d'Euler donne un algorithme pour construire un parcours eulérien dans un graphe non orienté $G = (X, E)$ qui vérifie les conditions d'existence. On construit le parcours sous forme d'une liste **Walk** de $Last$ sommets. On note s et t les deux sommets impairs. Si tous les sommets sont pairs, $s = t = 1$. L'itération principale consiste à partir du premier sommet x de **Walk** ayant des arêtes non encore visitées, et à faire une sorte d'exploration de graphe, mais en marquant les arêtes au lieu des sommets. Cette exploration emprunte de proche en proche des arêtes libres, aussi longtemps que possible. L'exploration finit par se bloquer en x ou en t. Le parcours trouvé par l'exploration est greffé sur x dans **Walk**.

```
Chercher les 2 sommets impairs s et t
Si tous les sommets sont pairs, poser s = t = 1
Initialiser le parcours Walk avec le seul sommet s
p <- 1
Last <- 1
Répéter
   x <- Walk[p]
   S'il existe des arêtes non visitées incidentes à x alors
      Construire une chaîne c d'origine x, la plus longue possible,
      n'empruntant que des arêtes non visitées
      Greffer la liste de sommets c dans Walk, sur le sommet x
   Sinon p <- p + 1
Jusqu'à ce que p > Last.
```

L'algorithme est implémentable en $O(arcs)$.

La propriété suivante est l'équivalent du théorème d'Euler, mais pour les graphes orientés. Un graphe orienté $G = (X, U)$ est dit équilibré ou pseudo-symétrique, si pour tout sommet du graphe le degré extérieur est égal au degré intérieur. G admet un circuit eulérien si et seulement s'il est connexe et équilibré. Il admet un chemin eulérien si et seulement s'il est équilibré en tout sommet, sauf en un sommet s vérifiant $d_{ext} = d_{in} + 1$ et un sommet t vérifiant $d_{in} = d_{ext} + 1$. Pour un graphe sans parcours eulérien, on peut chercher des parcours chinois, qui visitent au moins une fois chaque arc ou arête. Pour une graphe valué $G = (X, U, C)$, il y a alors un problème de minimisation du coût total.

La recherche de chemins ou de circuits hamiltoniens est un problème généralement difficile à résoudre. Un cas célèbre où de tels chemins ou circuits interviennent est le problème du voyageur de commerce.

Théorème 1 :
Un graphe complet et fortement connexe possède un circuit hamiltonien.

Théorème 2 :
Un graphe admet un circuit chinois si et seulement s'il est fortement connexe.

14 Problèmes de coloration

Un graphe simple $G = (X, E)$ est k-colorable si on peut colorer ses sommets avec k couleurs distinctes, sans que deux sommets voisins aient la même couleur. Le plus petit k pour lequel G est k-colorable est le nombre chromatique de G, noté $\gamma(G)$. On peut aussi définir une coloration des arêtes. Dans ce cas, deux arêtes adjacentes de G doivent avoir des couleurs différentes. Le plus petit k pour lequel G admet une k-coloration des arêtes est appelé indice chromatique de G, noté $\Psi(G)$. Le calcul du nombre chromatique et celui de l'indice chromatique d'un graphe sont des problèmes NP-difficiles. Le problème du nombre chromatique devient polynomial si les sommets du graphe ont un degré d'au plus 2 ou si le graphe est **biparti**.

14.1 Coloration des arêtes

Soit Δ le degré maximal des sommets d'un graphe G. Il est clair que Δ est une borne inférieure simple de l'indice chromatique $\Psi(G)$. L'indice chromatique ne peut prendre que deux valeurs Δ ou $\Delta + 1$. On connaît des algorithmes polynomiaux pour trouver une coloration des arêtes en au plus $\Delta + 1$ couleurs, mais l'existence d'une Δ-coloration est un problème **NP-complet**.

14.2 Coloration des sommets

Kempe a montré dès 1879 que **tout graphe planaire est 5-colorable** (s'il a bien sûr au moins 5 sommets) et on connaît des algorithmes polynomiaux pour exhiber la coloration. Kempe a conjecturé qu'il existait toujours une 4-coloration, ce qui n'a été prouvé qu'en 1977 par Appel et Haken. La démonstration est très longue et fait appel à un ordinateur. Un nombre maximal de 4 couleurs est nécessaire pour colorier une carte géographique (graphe planaire) de façon à ce que deux pays limitrophes soient toujours de couleurs différentes.

Remarque :

Soit $G = (X, E)$ un graphe non-orienté. Le graphe des arêtes $A(G)$ a pour sommet les arêtes de G : deux sommets de $A(G)$ sont reliés par une arête si les arêtes correspondantes dans G sont adjacentes au même sommet. Il est clair que colorer les arêtes de G de façon à ce que deux arêtes adjacentes au même sommet ne reçoivent pas la même couleur, revient à colorer les sommets de $A(G)$.

14.2.1 Méthode heuristique séquentielle

Le calcul du nombre chromatique d'un graphe est un problème **NP-difficile** qui se résout en général à l'aide d'heuristiques, c'est-à-dire d'algorithmes qui ne garantissent pas de trouver une solution optimale, mais qui sont en principe conçus pour en trouver une bonne. Soit un tableau V définissant un ordre quelconque des sommets d'un graphe. Une méthode séquentielle consiste à colorer V_1,V_2,..., en lui affectant la plus petite couleur non utilisée par ses voisins. Le choix du sommet est imposé à chaque itération. L'algorithme suivant est simple et compact en $O(arcs \times arcs)$. Cette heuristique peut donner de très mauvais résultats *(il existe un ordre optimal)*.

```
Initialiser Color et NC à 0
Pour x allant de 1 à N
   Pour c allant de 1 à NC + 1
      Used[c] <- Faux
   Pour k allant de Head[V[x]] à Head[V[x]+1] - 1
      y <- Succ[k]
      Si Color[y] > 0 alors Used[Color[y]] <- Vrai
   c <- 0
   Répéter
      c <- c+1
   Jusqu'à ce que non Used[c]
   Color[V[x]] <- c
   NC <- Max(NC,c)
```

La façon la plus simple d'implémenter la méthode heuristique séquentielle est d'utiliser l'ordre naturel des numéros de sommets, c'est-à-dire $V_i = i$ pour tout sommet i. Cette heuristique est appelée **FFS** *(First Fit Sequential)*. On peut penser utiliser l'odre des degrés décroissants **LF** *(Largest Fit)*. Ceci donne l'heuristique **LFS** *(Largest First Sequential)*, proposée par Welsh et Powell. Pour l'heuristique **SLS** *(Smallest Last Sequential)*, l'ordre est :

— Le dernier sommet est le sommet de degré minimal.
— Le sommet V_i $(i = (sommets - 1), ..., 1)$ est celui de degré minimal dans le sous graphe induit par X–$\{V_{i+1}, ..., V_{sommets}\}$.

14.3 DSatur

Brélaz a proposé une méthode gloutonne inspirée des méthodes séquentielles, appelée DSatur. A l'itération i, on définit le degré de saturation $DS_i(x)$ comme le nombre de couleurs déjà utilisées par les voisins de x. L'heuristique consiste à :

— colorer le sommet de degré maximal avec la couleur 1.
— pour chaque étape suivante, prendre le sommet libre de DS maximal (prendre celui de degré maximal en cas d'ex aequo) et lui donner la plus petite couleur possible.

Cette méthode peut être considérée comme une méthode séquentielle mais dont l'ordre est construit au fur et à mesure au lieu d'être donné au début (méthode dynamique). L'ordre est tel que les sommets colorés forment à chaque itération un sous-graphe connexe. Bien que Dsatur soit très bonne en moyenne, il existe des graphes pour lesquels aucun ordre de ce type ne donne l'optimum.

14.4 Méthode exacte : Bactrack

L'idée de la méthode heuristique séquentielle peut être utilisée dans une énumération arborescente. Cette méthode est appelée Bactrack. Il ne s'agit pas d'une vraie méthode arborescente car il n'y a pas de fonction d'évaluation. L'algorithme commence par colorer en 1 le sommet V_1, ce qui forme la racine (niveau 1) de l'arborescence. Cette méthode est séquentielle en ce sens qu'au niveau i, elle colore le sommet V_i avec toutes les couleurs possibles. On note U_i l'ensemble des couleurs possibles en un nœud i. La méthode procède en profondeur d'abord, en colorant V_i avec la plus petite couleur non encore utilisée de U_i. Le but est de trouver rapidement une première solution, ce qui est primordial car seule la meilleure solution déjà trouvée va servir à élaguer (il n'y a pas de fonction d'évaluation). Pour gagner du temps il faut éviter de générer des colorations redondantes, c'est-à-dire égales à une permutation près. Admettons le théorème suivant :

Soit une coloration Color des sommets V_1, V_2, ..., V_{i-1} utilisant toutes les couleurs 1 à p. Si on veut générer des colorations non redondantes, la couleur pour V_i doit vérifier : $1 \leq Color(V_i) \leq p + 1$.

On peut maintenant préciser U_i. Soit p_{i-1} la plus grande couleur utilisée par V_1, V_2, ..., V_{i-1}, toute couleur j de U_i pour colorer V_i doit donc vérifier :

1. $j \leq p_{i-1} + 1$ (Théorème de Brown)
2. $j \leq \min(i, d(V_i) + 1)$ (d'après la borne BS_2)
3. j n'est utilisée par aucun voisin de V_i (méthode séquencielle)
4. si une k-coloration a déjà été trouvée, $j \leq k - 1$ (élagage)

15 Problèmes de flots

Un réseau de transport est un graphe orienté valué $G = (X, A, C, s, t)$ avec :
— X un ensemble de sommets,
— A un ensemble d'arcs,
— à chaque arc (i, j) est associé un nombre $C_{ij} \geq 0$ qui désigne la capacité maximale de l'arc,
— deux sommets particuliers s et t, appelés la **source** et le **puits**.
Un flot de valeur v est une application φ qui associe à chaque arc (i, j) une valeur φ_{ij} telle que :
— $0 \leq \varphi_{ij} \leq C_{ij} \quad \forall (i, j)$, c'est-à-dire que le flot doit être compatible avec la capacité,
— $\sum_j \varphi_{ij} = \sum_j \varphi_{ij} \quad \forall i \neq s, t$, c'est-à-dire que le flot doit être conservé en chaque sommet,
— $v = \sum_j \varphi_{sj} = \sum_j \varphi_{jt}$, la valeur v est égale au flot entrant par s et sortant par t.
Un arc tel que $C_{ij} = \varphi_{ij}$ est dit saturé.

Le problème de flot maximal consiste à trouver un flot maximisant la valeur v. On suppose que le graphe n'a pas d'arcs multiples d'un sommet vers un autre. Si cela se produit, on remplace tous les arcs par un arc unique, dont la capacité cumule celles des arcs remplacés. On suppose que les capacités sont entières, sinon certains algorithmes peuvent avoir des problèmes de convergence. Dans les problèmes pratiques, on peut toujours choisir une unité de mesure assez fine pour assurer l'intégrité des capacités. Il peut exister des arcs sans capacité, ce qui équivaut à une capacité infinie. On suppose que le graphe ne contient pas de chemin de capacité infinie de s à t, pour avoir une solution bornée. Dans le cas d'un graphe non orienté on remplace toute arête (i, j) par une par 2 arcs (i, j) et (j, i) de même capacité. Dans le cas où il y a plusieurs sources ou puits : on crée une super-source qu'on relie à toute source par un arc de capacité égale à la disponibilité de cette source, et on relie tout puits à un super-puits par un arc de capacité égale à la demande du puits. Dans le cas où les capacités sont exprimées sur les sommets, on remplace tout sommet x de ce type par deux sommets x' et x'', reliés par un arc (x', x'') de même capacité.

Les graphes d'écart sont souvent utilisés dans les raisonnements sur les flots. Pour un flot donné φ de G, le graphe d'écart $Ge(\varphi) = (X, A(\varphi))$ décrit les possibilités d'augmentation de flot. Il a les mêmes sommets que G et ses arcs sont définis tels que :
— tout arc (i, j) non-saturé de A est reporté dans $A(\varphi)$,
— pour tout arc (i, j) de flot non-nul dans A, on construit l'arc (j, i) dans $A(\varphi)$.
Avec cette théorie, on voit que l'on peut augmenter le flot dans un graphe G s'il existe un chemin de s à t dans le graphe d'écart $Ge(\varphi)$. A ce chemin correspond une chaîne augmentante de s à t dans le graphe G. Cette chaîne se compose d'un sous-ensemble d'arcs non saturés et empruntés dans le sens du parcours (arcs avant) et d'un ensemble d'arcs de flot non nul et empruntés en sens inverse du parcours (arcs arrière). On montre facilement que G contient une chaîne augmentante pour un flot φ si et seulement si le graphe d'écart $Ge(\varphi)$ contient un chemin de s à t. Les algorithmes ne construisent pas explicitement le graphe d'écart. Connaissant les listes de prédécesseurs, ils explorent le graphe d'écart implicitement : à partir d'un sommet i, ils balaient les arcs (i, j) non saturés et les arcs (j, i) de flot non nul.

Considérons un réseau de transport $G = (X, A, C, s, t)$. Une coupe de G est un sous-ensemble de sommets qui inclut la source s, mais pas le puits t. Cette coupe définit une partition de l'ensemble des sommets de X, on la note (S, T) avec $T = X–S$. Une coupe peut aussi être définie par l'ensemble des arcs ayant une extrémité dans S et l'autre dans T. Les arcs sortants sont ceux orientés de S vers T, les arcs entrants, ceux orientés de T vers S. On appelle capacité d'une coupe (S, T) la somme des capacités des arcs sortants de la coupe :

$$C(S, T) = \sum_{i \in S} \sum_{j \in T} C_i j$$

. Au vu de cette nouvelle notion, on introduit 4 théorèmes :

1. La capacité de toute coupe est toujours supérieure à la valeur de tout flot.
2. Un flot est maximal si et seulement si il n'admet pas de chaîne augmentante de s à t.
3. Si les capacités sont des nombres entiers, il existe un flot maximal qui est entier.
4. La valeur maximale d'un flot est égale à la capacité minimale d'une coupe.

15.1 Algorithme de Ford & Fulkerson

L'algorithme est basé sur la recherche de chemins successifs dans le graphe d'écart ou de chaînes augmentantes successives du réseau d'origine. L'algorithme travaille directement dans G et explore virtuellement le graphe d'écart en empruntant au départ d'un sommet i des arcs (i, j) non saturés (arc avant) et des arcs (j, i) de flot non nul (arc arrière). L'algorithme part d'un flot nul. Chaque itération principale consiste à chercher une chaîne augmentante de s à t dans G. Si on trouve une telle chaîne, on calcule l'augmentation δ de flot qu'elle permet. On peut alors augmenter le flot sur les arcs avant et le diminuer le flot sur les arcs arrière.

$$\delta = \min \left\{ \min_{arcs\ avant} \{C_{ij} - \varphi_{ij}\}, \min_{arcs\ arriere} \{\varphi_{ij}\} \right\}$$

La structure générale de l'algorithme est donc :

```
Initialiser F et le flot PHI à 0
Répéter
   Chercher une chaîne améliorante c de s à t dans G
   Si c est trouvée alors
      Calculer DELTA, augmentation de flot possible sur c
      Augmenter F et les flux des arcs avant de c de DELTA unités
      Diminuer les flux des arcs arrières de c de DELTA unités
Jusqu'à ce qu'il n'existe plus de chaîne améliorante.
```

La recherche d'une chaîne s'effectue avec une exploration de graphe adaptée au graphe d'écart sous-jacent. On marque au début le sommet s. Puis on propage les marques au départ d'un sommet i en empruntant des arcs (i, j) non saturés ou des arcs (j, i) de flot non nul. On a trouvé une chaîne si on peut marquer t. Voici cette exploration, si on utilise un file Q de sommets pour procéder en largeur :

```
Marquer s
Q <- {s}
Répéter
   DeQueue(Q,i)
   Pour tout successeur j de i tel que PHI_ij  < Cij
      Marquer j
      EnQueue(Q,j)
   Pour tout prédecesseur j de i tel que PHI_ij < 0
      Marquer j
      EnQueue(Q,j)
Jusqu'à ce que Q soit vide.
```

L'algorithme s'articule donc comme suit :

```
F <- 0
Initialiser le tableaux Phi des flux à 0
Construire le graphe inverse H et le tableau de correspondance Inv
Répéter avec G
   // Initalisation de l'exploratio en largeur, Father sert aussi de marques
   ClearSet(Q)
   EnQueue(Q,s)
   Initialiser le tableau Father à 0
   Father[s] <- s
   AugVal[s] <- +Infini
   // Visite virtuelle du graphe d'écart
   Répéter
      DeQueue(Q,x)
      ScanSuccs   // Propage le marquage par les (x,y) non saturés
      ScanPreds   // Propage le marquage par (y,x) de flux <= 0
   Jusqu'à ce que Q soit vide OU (Father[t] != 0)
   // Augmentation du flot si une chaîne améliorante est trouvée
   Si Father[t] != 0 alors
      Augment // augmente le flot
      F <- F + AugVal[t]
Jusqu'à ce que Father[t] = 0.
```

Les fonctions utilisées sont définies comme suit :

```
Fonction ScanSuccs
   Pour k allant de Head[x] à Head[x+1] - 1
      y <- Succ[k]
      Si (Father[y] = 0) ET (Phi[k] < C[k]) alors
         Father[y] <- x
         ArcTo[y] <- k
         AugVal[y] <- Min (AugVal[x], C[k] - Phi[k])
         EnQueue(Q,y)

Fonction ScanPreds
   Avec H pour k allant de Head[x] à Head[x+1] - 1
      y <- Succ[k]
      Si (Father[y] = 0) ET (Phi[Inv[k]] > 0) alors
         Father[y] <- x
         ArcTo[y] <- Inv[k]
         AugVal[y] <- Min (AugVal[x], C[k] - Phi[ArcTo[y]])
         EnQueue(Q,y)

Fonction Augment
   Delta <- AugVal[t]
   x <- t
   Répéter
      k <- ArcTo[x]
      Si Succ[x] = k alors
         Phi[k] <- Phi[k] + Delta
      Sinon
         Phi[k] <- Phi[k] - Delta
      x <- Father[x]
   Jusqu'à ce que x = s
```

Remarque : Si on recherche des plus courtes chaînes en nombre d'arcs, la complexité de l'algorithme est en $O(sommets \times arcs^2)$. Cette complexité est élevée pour des graphes denses.

15.2 Problème de flot compatible

Nous avons considéré des problèmes de flot où le flux sur chaque arc peut être nul. Ce flot nul est toujours admissible et sert de flot initial dans tous les algorithmes. On s'intéresse plus généralement au calcul d'un flot maximal dans un réseau de transport $G = (X, A, B, C, s, t)$ où B_{ij} désigne une borne inférieure de flux sur l'arc (i, j), appelée capacité minimale. De tels réseaux posent le problème de l'existence d'un flot φ compatible avec les capacités minimales et maximales : $B_{ij} \leq \varphi_{ij} \leq C_{ij}$.

Pour calculer un flot compatible de G, s'il existe, le réseau donné $G = (X, A, B, C, s, t)$ est transformé en un réseau G' avec capacités minimales nulles. On note $B^+(i)$ la somme des capacités minimales des arcs ayant i pour extrémité initiale et $B^-(i)$ la somme des capacités minimales des arcs ayant i pour extrémité terminale.

1. Si G n'a pas d'arc de retour (t, s), on crée cet arc avec une capacité infinie.
   Si (t, s) existe déjà, on rend sa capacité infinie.
2. Pour tout arc (i, j), sa capacité devient $C'_{ij} = C_{ij} - B_{ij}$.
3. On ajoute une source u et un puits v (indépendamment de s et t).
4. Pour tout sommet i, d'extrémité initiale d'au moins un arc, on ajoute un arc (i, v) de capacité $B^+(i)$.
5. Pour tout sommet i, d'extémité terminale d'au moins un arc, on ajoute un arc (u, i) de capacité $B^-(i)$.

On cherche ensuite un flot maximal de u à v dans G'. Le problème d'origine est résolu grâce à la propriété suivante :

G admet un flot compatible φ si et seulement si G' admet un flot maximal φ' de débit $B^+(u) = B^-(v)$, c'est-à-dire saturant les arcs d'origine u et d'extrémité v. De plus, pour tout arc (i, j) de G, on a :

$$\varphi_{ij} = \varphi'_{ij} + B_{ij}$$

⇒ On peut donc appliquer un algorithme de flot maximal au problème de flot compatible sans affecter sa complexité.

Si on dispose d'un flot compatible, le calcul d'un flot maximal ne demande qu'une modification mineure de l'algorithme de Ford et Fulkerson : pour prolonger une chaîne augmentante parvenue en un sommet i, on pourra emprunter un arc arrière (j, i) si et seulement si $\varphi_{ji} > B_{ji}$. L'algorithme part d'un flot compatible. Chaque itération principale consiste à chercher une chaîne augmentante de s à t dans G. Si on trouve une telle chaîne, on calcule l'augmentation δ de flot qu'elle permet. On peut alors augmenter le flot sur les arcs avant et le diminuer le flot sur les arcs arrière.

$$\delta = \min \left\{ \min_{arcs\ avant} \{C_{ij} - \varphi_{ij}\}, \min_{arcs\ arriere} \{\varphi_{ij} - B_{ij}\} \right\}$$

En conclusion, le problème de flot maximal avec capacités minimales n'est pas plus difficile que le problème de flot maximal.

15.3 Problème de flot de coût minimal

Soit un réseau de transport plus compliqué $G = (X, A, C, W, s, t)$ avec :
- X un ensemble de sommets,
- A un ensemble d'arcs
- à chaque arc (i, j) est associé un nombre $Cij \geq 0$ qui désigne la capacité maximale de l'arc,
- à chaque arc (i, j) est associé un nombre $Wij \geq 0$ qui désigne le coût de passage d'une unité de flux sur l'arc,
- deux sommets particuliers s et t, appelés la source et le puits.

Le problème du flot de coût minimal consiste à acheminer un flot de valeur v de s à t, de façon à minimiser le coût total. En particulier, si v est la valeur d'un flot maximal : on a alors un problème de flot maximal et de coût minimal.

Pour un flot φ , le graphe d'écart $Ge(\varphi)$ ressemble à celui défini pour le flot maximal, sauf qu'on doit tenir compte des coûts, il a les mêmes sommets que G, et ses arcs sont définis comme suit :
- tout arc (i, j) non saturé de G est conservé dans le graphe d'écart avec son coût,
- tout arc (i, j) de flot non nul de G donne lieu à un arc (j, i) de coût $-W_{ij}$ dans $Ge(\varphi)$.

Comme pour le flot maximal, un chemin de s à t dans le graphe d'écart correspond à une chaîne augmentante de G avec des arcs avant et arrière, le long de laquelle on peut augmenter le flot d'une certaine quantité δ. La variation de coût par unité supplémentaire de flot est égale au coût du chemin. On peut savoir si un flot est de coût minimal grâce à la propriété suivante :

Un flot φ de s à t et de valeur v est de coût minimal parmi tous les flots de valeur v si et seulement s'il n'existe pas de circuit négatif dans le graphe d'écart.

L'algorithme suivant a l'avantage d'être à la fois simple et efficace. Il existe des algorithmes de meilleure complexité théorique, mais leur implémentation est bien plus compliquée. On part d'un flot nul de valeur nulle et de coût total nul. A chaque itération, on cherche un chemin de coût minimal de s à t dans le graphe d'écart. Comme dans l'algorithme de Ford et Fulkerson, on travaille directement dans G sans construire explicitement ce graphe d'écart. Au chemin du graphe d'écart correspond une chaîne augmentante de G. L'algorithme se termine s'il n'existe pas de chemin de s à t : on a alors un flot de valeur maximale et de coût minimal. A chaque itération de l'algorithme, le flot en cours est de coût minimal parmi les flots de valeur F. Voici la structure générale de l'algorithme :

```
Initialiser F, K et le flot à 0
Répéter
   Chercher un chemin P de coût minimal z de s à t dans le graphe d'écart
   Soit c la chaine correspondante de G
   Si c est trouvée alors
      Calculer Delta, l'augmentation possible du flot sur c
      Delta <- Min(Delta, ReqF-F) // ReqF est une valeur finie si une valeur est imposée au flot
      Augmenter F et les flux des arcs avant de c de Delta unités
      Diminuer les flux des arcs arrière de c de Delta unités
      K <- K + Delta * z   // Incrémentation du poids total
Jusqu'à ce que (c non-trouvée) OU (F = ReqF)
```

Pour calculer un chemin de coût minimal en présence de coûts négatifs dans le graphe d'écart, il faut utiliser un algorithme de plus court chemin ad hoc, comme l'algorithme de Bellman ou sa version FIFO. L'algorithme a donc une complexité en $O(sommets \times arcs^2 \times U)$ (U est le maximun des capacités), en pratique, cet algorithme est efficace.

```
Fonction FIFO // Algorithme de plus court chemin dans le graphe d'écart.
   Initialiser le tableau P à 0 et le tableau d'étiquettes V à +Infini
   V[s] <- 0
   P[s] <- s
   AugVal[s] <- +Infini
   ClearSet(Q)
   EnQueue(Q,s)
   Répéter
      DeQueue(Q,x)
      Pour k allant de Head[x] à Head[x+1] - 1   // Dans G
         Si Phi[k] < C[k]
            y <- Succ[k]
            Si V[k] + W[k] < V[y] alors
               V[y] <- V[k] + W[k]
               P[y] <- x
               ArcTo[y] <- k
               AugVal[y] <- Min (AugVal[x], C[k] - Phi[k])
               EnQueue(Q,y)
      Pour k allant de Head[x] à Head[x+1] - 1
         Si Phi[Inv[k]] > 0
            y <- Succ[k]
            Si V[k] - W[Inv[k]] < V[y] alors
               V[y] <- V[k] - W[Inv[k]]
               P[y] <- x
               ArcTo[y] <- Inv[k]
               AugVal[y] <- Min (AugVal[x], Phi[ArcTo[y]])
               EnQueue(Q,y)
Jusqu'à ce que SetIsEmpty(Q)
```

Et l'algorithme est donc (La fonction **Augment** est la même que précédemment) :

```
F <- 0
K <- 0
Initialiser le tableau Phi des flux à 0
Construire le graphe inverse H et le tableau de correspondance Inv
Répéter
   FIFO           // Recherche de la chaîne de poids minimal dans Ge
   Si P[t] != 0   // On augmente le flot si une chaîne est trouvée
      AugVal[t] <- Min(AugVal[t], ReqF - F)
      Augment
      F <- F + AugVal[t]
      K <- K + AugVal[t]*V[t]
Jusqu'à ce que (P[t] = 0) OU (F = ReqF)
```

16 Problème de couplage

Un couplage C d'un graphe simple $G = (X, E)$ est un sous-ensemble d'arêtes de E deux à deux sans sommet commun. Un sommet x est dit saturé par C s'il est extrémité d'une arête de C. Sinon, il est dit libre ou insaturé. Un couplage parfait sature tous les sommets de X, ceci implique qu'il est de cardinal maximal et que le nombre de sommets est pair. Un couplage maximal de G est un couplage de cardinal maximal. Si les arêtes sont munies de coûts, on peut aussi chercher des couplages de coût minimal ou maximal, de cardinal imposé ou maximal. Le terme de problème d'affectation désigne la recherche d'un couplage maximal de coût minimal ou maximal, dans un graphe biparti.

Pour un graphe biparti $G = (X, Y, E)$, on peut convertir les problèmes de couplage en problèmes de flots dans un réseau associé $R = (X, Y, U, C, s, t)$. Ce réseau se déduit de G de la manière suivante :

— On oriente les arêtes de G en arcs de X vers Y, et on leur donne une capacité infinie,
— on ajoute une entrée s, reliée à tout sommet de X par un arc de capacité 1,
— on ajoute une sortie t, à laquelle tout sommet de Y est relié par un arc de capacité 1.

A un couplage de G correspond un flot dans R.
⇒ **On peut donc résoudre directement les problèmes de couplages bipartis avec des algorithmes de flots**.

— Pour le couplage biparti maximal, on utilise un algorithme pour le flot maximal.
— Pour le couplage biparti de coût minimal, on peut résoudre le problème de flot de coût minimal.

Les techniques de flots ne s'adaptent pas simplement aux problèmes de couplage dans les graphes non bipartis. Ces problèmes sont plus difficiles.

Soit un couplage C de G. Une chaîne est alternée si elle emprunte alternativement des arêtes de C et de $E - C$. Une arête unique est une chaîne alternée de longueur 1. Pour un couplage C, une chaîne alternée est **augmentante** si ses deux extrémités sont insaturées par C. L'opération de transfert consiste à enlever de C les arêtes de la chaîne qui faisient partie du couplage et à ajouter à C les arêtes de la chaîne qui n'en faisaient pas partie. Le problème de couplage maximal dans un graphe biparti ou non peut être résolu grâce aux deux propriétés suivantes :

1. *Soit un graphe simple $G = (X, E)$, un couplage C de G et une chaîne alternée augmentante, alors un transfert donne un nouveau couplage C' avec $|C'| = |C| + 1$,*
2. *Un couplage est maximal si et seulement s'il n'admet plus aucune chaîne alternée augmentante.*

L'algorithme suivant est très rapide et utilise les chaînes alternées augmentantes pour trouver un couplage maximal dans un graphe biparti $G = (X, Y, E)$. La complexité est $O(arcs \times sommets)$. Un couplage est codé par un tableau **Mate** tel que $Mate[i] = 0$ si le sommet i n'est pas saturé par le couplage et $Mate[i] = j$ si l'arête (i, j) est dans le couplage. Les chaînes augmentantes sont trouvées par une sorte d'exploration du graphe. Le but étant de trouver une chaîne alternée augmentante quelconque, on s'arrête à la plus courte trouvée, grâce à une exploration en largeur lancée à partir de tous les sommets insaturés. Cette exploration utilise une file Q de sommets. Elle doit emprunter des arêtes alternativement hors du couplage puis dans le couplage.

```
Initialiser Card et Mate à 0
Répéter
   // Recherche d'une chaine alternée augmentante quelconque par exploration en largeur
   Trouvé <- Faux
   Q <- vide
   Initialiser le tableau Father à 0
   Pour x allant de 1 à NX
      Si Mate[x] = 0 alors   // Part de tout x insaturé de X
         EnQueue(Q,x)
   Si Q n'est pas vide alors
      Pour y allant de NX-1 à NX-NY   // NX (resp NY) = nombre de noeuds dans X (resp Y)
         Father[i] <- 0
      Répéter
         DeQueue(Q,x)   // Note : i appartient à X
         k <- Head[x]   // Balaie les successeurs de x
         Tant que (Non-trouvé) et (k < Head[x+1])
            y <- Succ[k]
            Si Mate[y] = 0 alors   // y insaturé --> Chaine alternée augmentante trouvée
```

```
            Augment (y)   // On augmente le couplage
         Sinon Si Father[y] = 0 alors
            Father[y] <- x   // y non marqué, on va sur son Mate
            EnQueue(Q,Mate[y])   // Sommet non marqué mais dans le couplage
         k <- k+1
      Jusqu'à ce que Trouvé ou que Q soit vide
Jusqu'à ce que non Trouvé.

Fonction Augment (y)
   Trouvé <- Vrai
   Card <- Card + 1
   Father[y] <- x
   Répéter
      Mate[y] <- Father[y]
      x <- Mate[Father[y]]
      Mate[Father[y]] <- y
      y <- x
   Jusqu'à ce que y = 0
```

Cette dernière fonction (qui représente l'opération transfert) augmente la cardinalité du couplage.

Un sous-ensemble Z de sommets de X et Y forme une couverture du graphe biparti $G = (X, Y, E)$ si tout arc du graphe est incident à au moins un sommet de Z. Il existe une relation entre coupe de valeur finie et couverture. Soit (S, T) une coupe de valeur finie alors $Z = (Y \cap S) \cup (X \cap T)$ est une **couverture** de G.Réciproquement, soit Z une couverture de G, alors $S = \{s\} \cup (X - Z) \cup (Z - X)$ et $T = \{t\} \cup (Y - Z) \cup (Z - Y)$ forment une **coupe de valeur finie**.