# Préparation agrégation : Algorithmique

Schémas algorithmiques et graphes
Benjamin Monmege

Année 2010/2011

Ce document est composé de brefs rappels sur les schémas algorithmiques classiques ainsi que sur les graphes, munis de références bibliographiques, et est agrémenté d'exercices qui peuvent être des inspirations pour de futurs développements de leçon.

## 1 Schémas algorithmiques classiques

**902** Diviser pour régner : exemples et applications.
**903** Exemples d'algorithmes de tri. Complexité.
**906** Programmation dynamique : exemples et applications.
**925** Graphes : représentations et algorithmes.

### 1.1 Diviser pour régner [CLRS04, Sec. 2.3.1]

La méthode *diviser pour régner* s'applique pour des problèmes qui peuvent se diviser en plusieurs sous-problèmes semblables au problème initial mais de taille moindre. La paradigme implique trois étapes : diviser, régner et combiner.

**Tris.** Les premiers exemples simples à mentionner sont les tris : **tri rapide** [CLRS04, Ch. 7] et **tri par fusion**.

**Exercice 1.** Calculer la complexité en moyenne du tri rapide. (confronter [BBC92, Sec. 5.1.1] et [CLRS04, Sec. 7.4.2])

Lors de l'analyse du tri par fusion, ou plus généralement de tout algorithme diviser pour régner, on se retrouve confronter à des équations de récurrence, devant lesquelles il est important de savoir réagir vite. Voilà le théorème rappelant les solutions asymptotiques de ces équations.

**Théorème 1.** *[CLRS04, Thm. 4.1] Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour tout entier positif par la récurrence*

$$T(n) = a\,T(n/b) + f(n)$$

*où l'on interprète $n/b$ comme signifiant $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. $T(n)$ peut alors être bornée asymptotiquement de la façon suivante :*

1. *Si $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$ alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $a\,f(n/b) \leq c\,f(n)$ pour une certaine constante $c < 1$ et pour tout $n$ suffisamment grand, alors $T(n) = \Theta(f(n))$.*

Ce théorème très utile ne doit pas vous faire oublier qu'il est toujours possible de s'en sortir *à la main* par des méthodes de substitution ou d'arbre récursif (au moins pour conjecturer une réponse).

**Exercice 2** (Récurrences)**.** Puisque quelques exemples valent toujours mieux qu'un long discours... Résoudre les équations de récurrence suivante (sans utiliser le théorème précédent).

1. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$
2. $T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$
3. $T(n) = T(n/3) + T(2n/3) + cn$

**Applications numériques.** Les méthodes diviser pour régner sont très utilisées dans les algorithmes numériques : exponentiation rapide, multiplication de polynômes ou de matrices, transformée rapide de Fourier...

**Exercice 3** (Exponentiation rapide)**.**

1. Proposer un algorithme simple pour le calcul de $a^n$ ; analyser sa complexité.
2. Proposer un algorithme diviser pour régner pour le calcul de $a^n$ ; analyser sa complexité. On peut généraliser cette méthode pour écrire une exponentiation modulaire [CLRS04, Sec. 31.6].

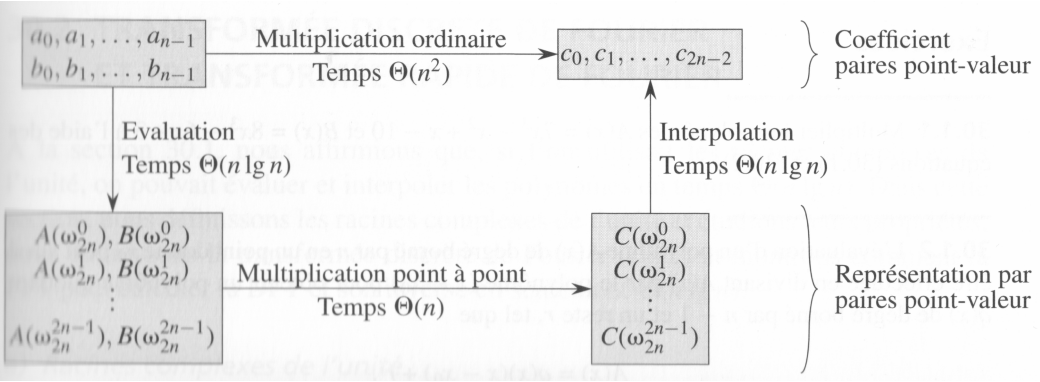
**Exercice 4** (Multiplication de polynômes)**.** On représente un polynôme $P = \sum_{i=0}^{n-1} a_i X^i$ par la liste de ses coefficients $(a_0, \ldots, a_{n-1})$. On s'intéresse au problème de la multiplication de deux polynômes : étant donnés deux polynômes $P$ et $Q$ de degré au plus $n-1$, calculer $PQ = \sum_{i=0}^{2n-2} c_i X^i$.

1. Donner un algorithme qui calcule directement chaque coefficient $c_i$. Quelle est sa complexité ?
2. On découpe chaque polynôme en deux parties de tailles égales :

$$P = P_1 + X^{\lceil n/2 \rceil} P_2 \qquad Q = Q_1 + X^{\lceil n/2 \rceil} Q_2$$

   En utilisant la relation $ab + cd = (a+c)(b+d) - ad - bc$, en déduire un algorithme de calcul du produit $PQ$, de complexité $\mathcal{O}(n^{\log_2 3})$.
3. Trouver une méthode similaire en découpant les polynômes selon leur coefficients pairs et impairs.

Il existe un autre moyen plus rapide pour calculer le produit de deux polynômes. Ceci utilise la **transformée rapide de Fourier** [CLRS04, Ch. 30]. Cette méthode est basée sur une interprétation des polynômes en certains points adaptés, puis utilise une méthode d'interpolation (bien retenir le schéma p. 801 qu'on a reproduit ici).




La partie coûteuse du processus est l'évaluation d'un polynôme $P = \sum_{j=0}^{n-1} a_j X^j$ en les $n$ racines $n$-ièmes de l'unité : $\omega_n^0, \ldots, \omega_n^{n-1}$, avec $\omega_n = \mathrm{e}^{2i\pi/n}$. Dans la suite, on suppose, sans perte de généralité, que $n$ est une puissance de 2. Pour $j \in \{0, \ldots, n-1\}$, on note $y_j = P(\omega_n^j)$.

**Exercice 5** (Évaluation et interpolation)**.**

1. Donner une méthode simple permettant de calculer les $(y_j)_{0 \leq j \leq n-1}$ ; quelle est sa complexité ?

2. On divise le polynôme $P$ en deux parties, selon les coefficients pairs et impairs, de sorte que $P = P_0(X^2) + XP_1(X^2)$. En remarquant que l'ensemble $\{(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2\}$ est composé de $n/2$ valeurs distinctes, en déduire un algorithme diviser pour régner permettant de calculer les $(y_j)_{0\leq j\leq n-1}$ en temps $\Theta(n \log n)$.
3. On s'intéresse finalement au problème inverse : trouver les coefficients $(a_j)_{0\leq j\leq n-1}$ à partir des valeurs $(y_j)_{0\leq j\leq n-1}$. Relier matriciellement ces deux vecteurs de complexes : $y = V_n a$. Montrer que la composante $(j,k)$ de la matrice $V_n^{-1}$ est $\omega_n^{-kj}/n$. Modifier finalement l'algorithme de la question précédente pour résoudre le problème demandé.

Une application intéressante est donnée par les matrices de Tœplitz. On reprend ci-dessous l'exercice 30.2 de [CLRS04].

**Exercice 6** (Matrices de Tœplitz)**.** Une matrice de Tœplitz est une matrice $A = (a_{i,j})$ carrée de taille $n$ telle que $a_{i,j} = a_{i-1,j-1}$ pour $i, j \in \{2, 3, \ldots, n\}$.

1. La somme de deux matrices de Tœplitz est-elle nécessairement une matrice de Tœplitz ? Et le produit ?
2. Décrire une manière de représenter une matrice de Tœplitz de sorte que deux matrices de Tœplitz de taille $n$ puissent être additionnées en temps $\mathcal{O}(n)$.
3. Donner un algorithme en temps $\mathcal{O}(n \log n)$ pour la multiplication d'une matrice de Tœplitz de taille $n$ par un vecteur de longueur $n$.

Un autre algorithme célèbre utilisant la méthode diviser pour régner est l'algorithme de Strassen [CLRS04, Sec. 28.2] pour la multiplication de matrices. Il est bon de le connaître, en particulier le résultat de complexité (multiplication de 2 matrices carrées en temps $\Theta(n^{\log_2 7})$ au lieu de $\Theta(n^3)$ pour la méthode naïve), mais il semble dangereux de le présenter en développement.

**Géométrie algorithmique.** Finalement, un dernier champ où les méthodes diviser pour régner apparaissent concerne la géométrie algorithmique. Citons dans un premier temps la **recherche de l'enveloppe convexe** de $n$ points donnés. De nombreuses méthodes existent : l'une d'elle (qu'on peut trouver dans [BY95, Sec. 9.2]) utilise une méthode diviser pour régner, et s'exécute en temps $\mathcal{O}(n \log n)$. La présentation de cet algorithme est à réserver aux passionnés de géométrie... L'exercice suivant présente quant à lui un algorithme permettant de trouver les deux points les plus rapprochés d'un ensemble de points donnés : là où la méthode naïve réaliserait $\mathcal{O}(n^2)$ tests, la méthode diviser pour régner (qu'on trouve présentée dans [CLRS04, Sec. 33.4]) affiche une complexité en $\mathcal{O}(n \log n)$.

**Exercice 7** (Recherche des deux points les plus rapprochés)**.** Une instance du problème consiste en un ensemble $Q$ de $n \geq 2$ points du plan euclidien. On cherche les deux points de $Q$ dont la distance euclidienne est minimale. Chaque appel récursif prend en entrée un sous-ensemble $P \subseteq Q$ et des tableaux $X$ et $Y$, contenant chacun tous les points de $P$ : les points du tableau $X$ sont triés par ordre croissant des abscisses, alors que le tableau $Y$ est trié par ordre croissant des ordonnées.

1. Décrire un algorithme dans le cas où $|P| \leq 3$.
2. On suppose maintenant que $|P| \geq 3$. Voilà la procédure employée :
   - **Diviser** : Trouver une droite verticale $\ell$ qui sépare $P$ en deux sous-ensembles $P_G$ (les points à gauche de $\ell$) et $P_D$ (les points à droite de $\ell$) approximativement de même taille. Le tableau $X$ est ainsi divisé en deux tableaux $X_G$ et $X_D$, triés dans l'ordre croissant, de même pour le tableau $Y$.
   - **Régner** : Appels récursifs sur les deux entrées $(P_G, X_G, Y_G)$ et $(P_D, X_D, Y_D)$. On note $\delta_G$ et $\delta_D$ les distances trouvées, et $\delta = \min(\delta_G, \delta_D)$.
   - **Combiner** : La distance minimale est soit $\delta$, soit la distance (alors inférieure à $\delta$) entre un point de $P_G$ et un point $P_D$. On décrit comment trouver ces deux points, s'ils existent. On crée un tableau $Y'$, trié dans l'ordre croissant des ordonnées, qui contient les points qui se trouvent

dans un ruban vertical de largeur $2\delta$ centré sur $\ell$. Pour chaque point $p$ de $Y'$, l'algorithme essaye de trouver des points de $Y'$ se trouvant à une distance de $p$ inférieure à $\delta$ : pour cela, il compare les 7 points de $Y'$ qui suivent $p$. On mémorise alors la distance minimale $\delta'$ trouvée. Si $\delta' < \delta$, alors on renvoie $\delta'$ (ainsi que les deux points de $P_G$ et $P_D$) sinon on renvoie $\delta$.

Prouver la validité de cet algorithme.

3. Proposer une implémentation et calculer la complexité de votre méthode.

## 1.2 Programmation dynamique ([CLRS04, Ch. 15], [DPV06, Ch. 6])

La programmation dynamique, comme la méthode diviser pour régner, résout des problèmes en combinant des solution de sous-problèmes. Cependant, les sous-problèmes ne sont plus traités de manière indépendante : au contraire, on mémorise dans un tableau les solutions des problèmes déjà résolus pour éviter d'avoir à les calculer plusieurs fois.

On utilise souvent la programmation dynamique dans des problèmes d'optimisation : étant donnée une fonction $f : S \longrightarrow V$ trouver $\max_S f$ ou $\text{argmax}_S f$, dans un espace $S$ *grand*. On procède en 4 étapes :

1. Analyse de la solution optimale.
2. Écriture de la valeur optimale en fonction de valeurs optimales de sous-problèmes.
3. Calcul de la valeur d'une solution optimale en suivant un ordre topologique du graphe des appels (de manière ascendante, ou *bottom-up*).
4. Si besoin, ajouter des informations pour trouver la solution optimale (en plus de sa valeur).

Un algorithme représentatif de la programmation dynamique est l'algorithme de Floyd-Warshall, qui calcule des plus courts chemins dans un graphe : nous l'étudierons dans la section 2.5. Dans la suite, on propose trois exercices... On peut ajouter à cette liste le problème de la **plus longue sous-séquence commune** [CLRS04, Sec. 15.4].

**Exercice 8** (Distance d'édition)**.** On considère deux mots $x$ et $y$ fixés, sur un alphabet $\Sigma$ fini. On cherche à *aligner* ces deux mots, c'est-à-dire à trouver la plus petite suite d'opérations qui permettent de transformer un mot en l'autre. Les opérations disponibles sont l'insertion d'une lettre, la suppression d'une lettre ou la substitution d'une lettre par une autre. Voici deux manières d'aligner les mots EXPONENTIEL et POLYNOMIAL :

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| E | X | P | O | N | E | N | - | T | I | E | L |
| - | - | P | O | L | Y | N | O | M | I | A | L |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| E | X | P | O | N | E | N | T | I | E | - | L |
| P | O | L | Y | N | O | M | - | I | - | A | L |

Les tirets représentent l'insertion ou la suppression d'un caractère. Chaque opération coûte 1. Évidemment lorsque deux lettres sont identiques (on pourra parler dans la suite de vérification), le coût de la vérification est nul. Ainsi, les coûts des deux transformations précédentes sont respectivement 7 et 9. La distance d'édition de deux mots $x$ et $y$ est le coût minimal d'une suite d'opérations transformant l'un en l'autre.

1. Donner un algorithme simple, ainsi que sa complexité dans le pire cas, permettant de trouver la distance d'édition entre deux mots $x$ et $y$, de longueurs respectives $m$ et $n$.
2. Afin d'améliorer cet algorithme, considérons le sous-problème consistant à trouver la distance d'édition d'un préfixe $x[1 \dots i]$ du mot $x$ à un préfixe $y[1 \dots j]$ du mot $y$. Notons $E(i, j)$ la solution de ce sous-problème. En supposant connus $E(i-1, j)$, $E(i, j-1)$ et $E(i-1, j-1)$, exprimer la valeur de $E(i, j)$.
3. En déduire un algorithme de programmation dynamique permettant de calculer $E(m, n)$. Il s'agit de donner un tableau bidimensionnel et de fournir les instructions permettant de le remplir (en particulier, comment initialiser ce tableau ?). Quelle est la place utilisée en mémoire ?

4. En complétant l'algorithme précédent, faites imprimer la liste des opérations à faire pour transformer un mot en un autre.
5. Finalement, on veut économiser de la place mémoire. Trouver une manière de calculer $E(m,n)$ en utilisant une place mémoire à tout moment inférieure à $\min(m,n)+3$. Peut-on trouver la suite d'opérations à produire afin d'aligner les mots avec une économie telle sur la mémoire ?

**Exercice 9** (Problème du sac-à-dos)**.** On considère le problème du sac-à-dos suivant :

| |
|---|
| **Données** : $(v_i, w_i)_{1\leq i\leq n}$ suite de paires d'entiers positifs (la valeur et le poids des $n$ objets) et un entier positif $W$ (la capacité du sac-à-dos) |
| **Objectif** : trouver un sous-ensemble $I \subseteq \{1,\dots,n\}$ tel que $\sum_I w_i \leq W$ et $\sum_I v_i$ maximale |

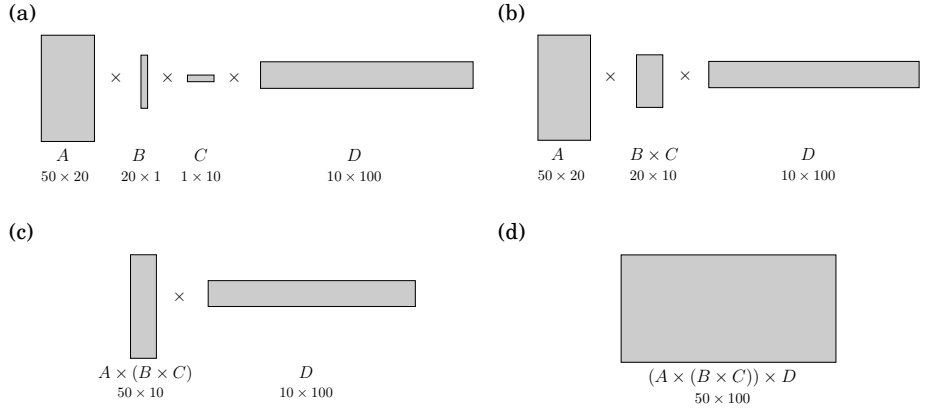
On introduit pour deux paramètres $w \in \{0,\dots,W\}$ et $j \in \{0,\dots,n\}$ le sous-problème dans lequel le sac-à-dos a une capacité de $w$ et les objets disponibles sont les objets d'index $1,\dots,j$. On note $K(w,j)$ la solution de ce sous-problème : c'est donc la valeur maximale qu'un sac-à-dos de capacité $w$ peut contenir, en choisissant parmi les objets $1,\dots,j$.

1. Pour $j \geq 1$, donner la valeur de $K(w,j)$ en fonction des valeurs $K(w',j-1)$ pour $w' \leq w$.
2. En déduire un algorithme de programmation dynamique calculant la valeur de $K(W,n)$. Quelle est sa complexité ? Compléter votre algorithme pour qu'il fournisse la liste des objets à choisir.
3. Donner une version de votre algorithme calculant la valeur de $K(W,n)$ n'utilisant qu'une place mémoire à tout moment inférieure à $W+3$.
4. Montrer que le problème de décision suivant (Somme-Sous-Ensemble) est NP-complet :

   | |
   |---|
   | **Données** : $S \subseteq \mathbb{N}$ fini, $t \in \mathbb{N}$ |
   | **Question** : Existe-t-il $S' \subseteq S$ tel que $\sum_{x\in S'} x = t$ ? |

   En déduire un problème de décision NP-complet adapté du problème du sac-à-dos. A-t-on contredit ce résultat avec les algorithmes présentés précédemment ?
5. (Facultatif) On peut également étudier la variante du problème du sac-à-dos dans laquelle on a un stock illimité de chacun des objets présentés...

**Exercice 10** (Multiplications matricielles)**.** Supposons qu'on veuille multiplier 4 matrices $A\times B\times C\times D$ de dimensions respectives $50\times 20$, $20\times 1$, $1\times 10$ et $10\times 100$ (cf schéma **a**). La multiplication matricielle n'est pas commutative, mais elle est associative. Ainsi, on peut calculer le produit des 4 matrices de plusieurs manières différentes, selon la manière dont on le parenthèse.




1. Au vue du schéma précédent, que pensez-vous d'une approche gloutonne permettant de résoudre ce problème ?

Généralisons le problème : on cherche à déterminer l'ordre optimal pour calculer le produit matriciel $A_1 \times A_2 \times \cdots \times A_n$ où les matrices $A_i$ sont de dimensions respectives $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$.
Précisons ici que l'algorithme que nous cherchons n'est qu'un *prétraitement* à la multiplication en elle-même : les différentes complexités demandées dans la suite ne correspondent donc qu'à ce prétraitement et ne doivent aucunement contenir le coût des multiplications.

2. Quelle est la complexité d'un algorithme qui teste tous les ordres possibles ?
3. Proposer un algorithme de programmation dynamique permettant de donner le nombre minimal de multiplications matricielles nécessaires pour effectuer un produit matriciel. Cet algorithme prendra en entrée la suite des tailles $m_i$. Quelle est la complexité (temporelle et spatiale) de votre algorithme ?

### 1.3 Approche gloutonne [CLRS04, Ch. 16]

Un algorithme glouton détermine une solution optimale pour un problème d'optimisation après avoir effectué une série de choix. Pour chaque point de décision de l'algorithme, le choix qui semble le meilleur à l'instant est effectué. On conçoit généralement les algorithmes gloutons selon les étapes suivantes :

1. Transformation du problème d'optimisation en un problème dans lequel on fait un choix (glouton) à la suite duquel on se retrouve avec *un seul* sous-problème à résoudre.
2. Démonstration qu'il y a toujours une solution optimale du problème initial qui fait le choix glouton, de sorte que le choix glouton est toujours approprié.
3. Démonstration que, après avoir fait le choix glouton, on se retrouve avec un sous-problème tel que, si l'on combine une solution optimale de celui-ci et le choix glouton que l'on a fait, on arrive à une solution optimale du problème originel.

**Exercice 11** (Variante fractionnaire du problème du sac-à-dos)**.** On considère le problème suivant :

> **Données** : $(v_i, w_i)_{1 \leq i \leq n}$ suite de paires d'entiers positifs et un entier positif $W$
> **Objectif** : trouver un vecteur $(\alpha_i)_{1 \leq i \leq n}$ de rationnels entre 0 et 1
> tels que $\sum \alpha_i w_i \leq W$ et $\sum \alpha_i v_i$ maximale

1. Résoudre ce problème de manière gloutonne en une complexité $\mathcal{O}(n \log n)$. Pour rappel, la version entière du problème du sac-à-dos est un problème NP-complet.
2. (On s'inspire du problème 9.2 de [CLRS04]) Rappeler une méthode de complexité linéaire permettant de déterminer le $k$-ème plus petit élément d'un tableau à $n > 1$ éléments. Généraliser cet algorithme pour qu'il résolve le problème du médian pondéré :

   > **Données** : $z_1, \ldots, z_n \in \mathbb{Q}$, $w_1, \ldots, w_n \in \mathbb{Q}_+$ et $W \in \mathbb{Q}$ tel que $0 < W \leq \sum_{i=1}^{n} w_i$
   > **Objectif** : trouver le $(w_1, \ldots, w_n; W)$-médian pondéré par rapport à $(z_1, \ldots, z_n)$
   > c'est-à-dire l'unique nombre $z^*$ pour lequel $\sum_{z_i < z^*} w_i < W \leq \sum_{z_i \leq z^*} w_i$

   En déduire que la variante fractionnaire du problème du sac-à-dos peut être résolue en $\mathcal{O}(n)$.

Une autre manière d'illustrer les méthodes gloutonnes est d'étudier le **codage de Huffman** [CLRS04, Sec. 16.3]. On verra dans la suite du cours d'autres algorithmes gloutons célèbres : algorithme de Dijkstra pour trouver un plus court chemin, algorithmes de Prim et Kruskal pour construire un arbre couvrant de poids minimum.

Remarquons pour finir que la stratégie gloutonne ne produit pas toujours une solution optimale. On utilise en effet souvent cette stratégie pour trouver une *approximation* correcte d'un problème difficile, à moindre coût. On peut par exemple étudier le **problème de la couverture d'ensemble** [CLRS04, Sec. 35.3] décrit ci-dessous :

> **Données** : $X$ un ensemble fini, $\mathcal{F}$ une famille de sous-ensembles de $X$ tel que $\bigcup_{S \in \mathcal{F}} S = X$
> **Objectif** : trouver un sous-ensemble de taille minimale $\mathcal{C} \subseteq \mathcal{F}$ tel que $X = \bigcup_{S \in \mathcal{C}} S$

# 2 Algorithmes de graphes

**901** Exemples de structures de données et de leurs applications.
**904** Problèmes NP-complets : exemples.
**925** Graphes : représentations et algorithmes.

## 2.1 Généralités sur les graphes et les arbres

| Graphes orientés | Graphes non orientés |
|---|---|
| Arcs : $(u, v) \in S \times S$ | Arêtes : $\{u, v\} \in \mathcal{P}_2(S)$ paire de sommets |
| Chemin | Chaîne |
| Circuit | Cycle |
| Forte connexité | Connexité |
| Cocycle $w(T) = \{(u, v) \in A \mid (u \in T \wedge v \in S - T) \vee (u \in S - T \wedge v \in T)\}$ | $w(T) = \{\{u, v\} \in A \mid u \in T \wedge v \in S - T\}$ |
| Arborescence (arbre enraciné) | Arbre : graphe connexe acyclique |

Autres notions communes : sommets adjacents, graphe biparti (s'il existe $T \subseteq S$ tel que $\omega(T) = A$), sous-graphe, chemin ou chaîne élémentaire...

**Lemme 1** (König)**.** *De tout chemin dans un graphe orienté, on peut extraire un chemin élémentaire. (Le lemme s'étend aux circuits d'un graphe orienté et aussi aux chaînes et aux cycles d'un graphe non orienté.)*

**Proposition 1** (définitions équivalentes des arbres)**.** *[CLRS04, Thm. B.2] Soit $G = (S, A)$ un graphe non orienté. Les assertions suivantes sont équivalentes :*

*(i) $G$ est un arbre (par définition, graphe connexe acyclique)*
*(ii) Deux sommets quelconques de $G$ sont reliés par une chaîne élémentaire unique.*
*(iii) $G$ est connexe mais, si l'on enlève un sommet quelconque à $A$, le graphe résultant n'est plus connexe.*
*(iv) $G$ est connexe et $|A| = |S| - 1$*
*(v) $G$ est acyclique et $|A| = |S| - 1$*
*(vi) $G$ est acyclique, mais si une arête quelconque est ajoutée à $A$, le graphe résultant contient un cycle.*

**Exercice 12.** Prouver la proposition précédente.

Dans ce contexte, on appelle souvent arborescence un arbre dont un sommet est choisi pour devenir la racine : les sommets sont alors souvent appelés nœuds.

## 2.2 Représentation des graphes [BBC92, Sec. 4.1]

Il existe deux représentations classiques pour les graphes (orientés ou non). La première, particulièrement utile pour les graphes peu denses, est la liste d'adjacence : elle est très compacte $\mathcal{O}(|S| + |A|)$, mais la recherche d'un arc est coûteuse $\mathcal{O}(|A|)$. Grâce aux listes d'adjacence, on peut facilement connaître le degré (sortant) des sommets d'un graphe (orienté). La seconde est la matrice d'adjacence : elle utilise de façon plus importante la mémoire $\mathcal{O}(|S|^2)$, mais la recherche d'un arc s'exécute en temps constant. Avec la matrice d'adjacence $M$ d'un graphe, on peut connaître le nombre de chemins de longueur $k$ entre deux sommets $u$ et $v$ en calculant le coefficient d'indice $(u, v)$ de la matrice $M^k$.

| Liste d'adjacence | Matrice d'adjacence |
|---|---|
| parcours en profondeur et en largeur<br>algorithme de Prim<br>algorithme de Dijkstra | algorithme de Roy-Warshall<br>algorithme de Floyd-Warshall |

Ces deux structures s'étendent très facilement au cas des graphes pondérés (définition dans [CLRS04, Sec. 22.1]). Citons finalement une dernière structure permettant de représenter les graphes orientés : la matrice d'incidence. Pour un graphe orienté $G = (S, A)$, c'est la matrice $B = (b_{ij})_{i \in S, j \in A}$ définie par

$$b_{ij} = \begin{cases} -1 & \text{si l'arc } j \text{ sort du sommet } i \\ 1 & \text{si l'arc } j \text{ arrive au sommet } i \\ 0 & \text{sinon} \end{cases}$$

Elle est souvent utile pour formaliser les problèmes de flots, ainsi que pour résoudre des systèmes de contraintes de potentiel [CLRS04, Sec. 24.4.b].

### 2.3 Accessibilité : algorithme de Roy-Warshall [BBC92, Sec. 4.2.1]

Étant donné un graphe orienté $G = (S, A)$, le sommet $v$ est dit *accessible* à partir du sommet $u$ s'il existe un chemin de $u$ à $v$. On cherche ainsi à calculer la relation d'accessibilité : en d'autres termes, on cherche à construire la clôture réflexive et transitive du graphe. Pour ce faire, on utilise la matrice d'adjacence du graphe.

**Exercice 13.** [Algorithme de Roy-Warshall] On pose $S = \{1, 2, \ldots, n\}$ et pour $k \in S$, on note $E(k)$ l'ensemble $\{1, 2, \ldots, k\}$. Si $c = (v_1, v_2, \cdots, v_k)$ est un chemin de $G$ (on notera $c : v_1 \to v_k$), l'intérieur $I(c)$ de ce chemin est l'ensemble des sommets $\{v_2, \ldots, v_{k-1}\}$. On note $G_k = (S, A_k)$ le graphe défini par

$$(i, j) \in A_k \iff \exists c : i \to j \quad I(c) \subseteq E(k)$$

1. Prouver que pour tout $i, j, k \in S$, l'arc $(i, j)$ appartient à $A_k$ si et seulement si $(i, j) \in A_{k-1}$ ou $((i, k) \in A_{k-1}$ et $(k, j) \in A_{k-1})$. En déduire un algorithme simple calculant la relation d'accessibilité du graphe $G$. Quelle est sa complexité (spatiale et temporelle) ?
2. On cherche à améliorer la complexité spatiale de cet algorithme afin qu'il n'utilise qu'une seule matrice booléenne, initialisée avec la matrice d'adjacence. Modifier votre algorithme pour qu'il en soit ainsi, après avoir démontré les deux équivalences suivantes, valables pour tout $i, j, k \in S$ :

$$(i, k) \in A_{k-1} \iff (i, k) \in A_k \qquad \text{et} \qquad (k, j) \in A_{k-1} \iff (k, j) \in A_k$$

On étudiera une généralisation aux graphes pondérés avec l'algorithme de Floyd-Warshall (cf. section 2.5).

### 2.4 Parcours de graphes [CLRS04, Ch. 22]

**Parcours en largeur** Le parcours en largeur est utilisé pour calculer des distances de plus court chemin à partir d'un point. Il n'est donc pas étonnant qu'on retrouve des idées similaire dans le cas des graphes pondérés, avec l'algorithme de Dijkstra par exemple. Ce type d'exploration de graphes se retrouve également dans l'algorithme de Prim.

On rappelle ci-dessous la procédure de parcours en largeur. Soit $G = (S, A)$ un graphe représenté par une liste d'adjacence et soit $s$ un sommet arbitraire. Pour chaque sommet $u \in S$ du graphe, on maintient à jour plusieurs informations supplémentaires :

- une couleur $\chi(u)$ : noir (visite terminée), gris (en cours de visite), blanc (pas encore visité),
- un parent $\pi(u)$ : si $u$ n'a pas de parent, alors $\pi(u) =$ nil,
- la distance calculée de $s$ à $u$, $d(u)$.

L'algorithme utilise une file $F$ pour gérer l'ensemble des sommets gris.

```
Pour chaque sommet u ∈ S − {s}
    faire χ(u) ← BLANC, d(u) ← ∞, π(u) ← NIL
finPour
χ(s) ← GRIS, d(s) ← 0, π(s) ← NIL, F ← {s}
Tant que F ≠ ∅
    faire u ← tête(F)
    Pour chaque v adjacent à u
        faire Si χ(v) = BLANC
            alors χ(v) ← GRIS, d(v) ← d(u) + 1, π(v) ← u, ENFILE(F, v)
            finSi
    finPour
    DÉFILE(F), χ(u) ← NOIR
finTantque
```

**Exercice 14** (Analyse du parcours en largeur)**.**

1. Quelle est la complexité du parcours en largeur ?
2. On note $\delta(s, u)$ la distance d'un plus court chemin de $s$ à $u$ (éventuellement $\infty$ s'il n'existe aucun chemin de $s$ à $u$). Montrer que pour tout arc $(u, v) \in A$, $\delta(s, v) \leq \delta(s, u) + 1$.
3. Montrer qu'après exécution du parcours en largeur, pour tout $u \in S$, $d(u) \geq \delta(s, u)$.
4. Soit $F = (u_1, u_2, \cdots, u_r)$ la file à un moment de l'exécution (avec $u_1$ la tête). Montrer que $d(u_r) \leq d(u_1) + 1$ et $d(u_i) \leq d(u_{i+1})$ pour tout $i \in \{1, 2, \ldots, r-1\}$.
5. Montrer qu'à la fin de l'exécution, tous les sommets accessibles à partir de $s$ sont découverts par l'algorithme, et que $d(u) = \delta(s, u)$ pour tout $u \in S$. Montrer aussi que pour tout sommet $u \neq s$, accessible à partir de $s$, l'un des plus courts chemins de $s$ à $u$ est le plus court chemin de $s$ à $\pi(u)$ complété par l'arc $(\pi(u), u)$.

Les informations $\pi(u)$ permettent de construire une arborescence de racine $s$, $G_\pi$ : dans le cas non orienté, cette arborescence contient tous les sommets du graphe uniquement si le graphe est connexe.

**Exercice 15** (File en Caml)**.** Pour vous préparer aux épreuves de programmation, savez-vous programmer une file en Caml ? Utilisez-la pour implémenter le parcours en largeur...

**Parcours en profondeur** Contrairement au parcours en largeur, le parcours en profondeur ne désigne pas de racine arbitraire. L'exploration se fait à partir d'un sommet quelconque puis continue en profondeur dans le graphe, jusqu'à ce que tous les sommets accessibles aient été visités. S'il reste des sommets non découverts, on en choisit un qui servira de nouvelle origine, et le parcours reprend à partir de cette origine. Le processus complet est répété jusqu'à ce que tous les sommets aient été découverts.

Comme dans le parcours en largeur, chaque fois qu'un sommet $v$ est découvert pendant le balayage d'une liste d'adjacence d'un sommet $u$, le parcours en profondeur enregistre cet évènement en donnant la valeur $u$ à $\pi(v)$. Contrairement au parcours en largeur, le sous-graphe formé par les champs $\pi(u)$ ne possède pas une seule arborescence, mais forme une forêt d'arborescences (c'est-à-dire que les différentes arborescences sont disjointes : pourquoi ?) : il est défini par $G_\pi = (S, A_\pi)$ avec

$$A_\pi = \{(\pi(u), u) \mid u \in S, \pi(u) \neq \text{NIL}\}$$

En plus de créer une forêt de parcours en profondeur, le parcours en profondeur date chaque sommet avec deux dates : la première, $d(u)$, marque le moment où $u$ a été découvert pour la première fois (et colorié en GRIS), et la seconde $f(u)$ enregistre le moment où le parcours a fini d'examiner la liste d'adjacence de $u$ (et le colorie en NOIR). Les dates sont des entiers compris entre 1 et $2|S|$, puisque découverte et fin de traitement se produisent une fois et une seule pour chacun des $|S|$ sommets.

```
PP(G)
    Pour chaque sommet u ∈ S
        faire χ(u) ← BLANC, π(u) ← NIL
    finPour
    date ← 0
    Pour chaque sommet u ∈ S
        faire Si χ(u) = BLANC alors VISITER-PP(u) finSi
    finPour
```

```
VISITER-PP(u)
    χ(u) ← GRIS, date ← date + 1, d(u) ← date
    Pour chaque v adjacent à u
        faire Si χ(v) = BLANC alors π(v) ← u, VISITER-PP(v) finSi
    finPour
    χ(u) ← NOIR, date ← date + 1, f(u) ← date
```

**Exercice 16** (Analyse du parcours en profondeur)**.**

1. Quelle est la complexité du parcours en profondeur ?
2. On suppose l'exécution de $PP(G)$ terminée. Montrer que pour tous sommets $u$ et $v$ de $G$, une et une seule des trois conditions suivantes est vérifiée :
   - les intervalles $[d(u), f(u)]$ et $[d(v), f(v)]$ sont disjoints, et ni $u$ ni $v$ n'est un descendant de l'autre dans la forêt d'arborescence de parcours en profondeur,
   - l'intervalle $[d(u), f(u)]$ est entièrement inclus dans l'intervalle $[d(v), f(v)]$, et $u$ est un descendant de $v$ dans une arborescence de parcours en profondeur, ou
   - l'intervalle $[d(v), f(v)]$ est entièrement inclus dans l'intervalle $[d(u), f(u)]$, et $v$ est un descendant de $u$ dans une arborescence de parcours en profondeur.
3. Montrer que le sommet $v$ est un descendant (dans la forêt de parcours en profondeur) d'un sommet $u$ si et seulement si, à la date $d(u)$ où le parcours découvre $u$, il existe un chemin de $u$ à $v$ dans $G$ composé uniquement de sommets blancs.

La parcours en profondeur possède de nombreuses applications. Dans un graphe orienté sans circuit, les dates de fin de traitement permettent de réaliser un tri topologique des sommets (voir [CLRS04, Sec. 22.4]). Par ailleurs, l'algorithme $PP$ permet aussi de détecter simplement les cycles (également [CLRS04, Sec. 22.4]). Le parcours en profondeur permet également de décomposer un graphe en composantes fortement connexes : une composante fortement connexe d'un graphe orienté $G = (S, A)$ est un ensemble maximal de sommets $R \subseteq S$ tel que pour chaque paire de sommets $u, v \in R$, $u$ et $v$ sont mutuellement accessibles.

**Exercice 17** (Composantes fortement connexes)**.** Soit $G = (S, A)$ un graphe orienté. On définit le graphe transposé de $G$ par ${}^{\mathrm{T}}G = (S, {}^{\mathrm{T}}A)$ où ${}^{\mathrm{T}}A = \{(u, v) | (v, u) \in A\}$.

1. Étant donnée une représentation par listes d'adjacence de $G$, quel est le coût de la création de ${}^{\mathrm{T}}G$ ?
2. Soit $f$ la fonction des dates de fin de traitement relative au parcours en profondeur du graphe $G$ : on étend ces dates aux ensembles $U \subseteq S$ de sommets par $f(U) = \max_{u \in U} f(u)$. Soient $C$ et $C'$ des composantes fortement connexes distinctes de $G$. On suppose qu'il y a un arc $(u, v) \in A$ tel que $u \in C$ et $v \in C'$. Montrer que $f(C) > f(C')$.
3. Soient $C$ et $C'$ des composantes fortement connexes distinctes de $G$. On suppose qu'il y a un arc $(u, v) \in {}^{\mathrm{T}}A$ tel que $u \in C$ et $v \in C'$. Montrer que $f(C) < f(C')$.
4. En déduire un algorithme réalisant deux parcours en profondeur qui calcule les composantes fortement connexes de $G$. Quelle est sa complexité ?

Le parcours en profondeur possède de nombreuses autres applications : citons par exemple la recherche de sommets d'articulation, de ponts et de composantes biconnexes [CLRS04, Prob. 22.2].

**Parcours généralisé** De manière générale, on appelle parcours d'un graphe connexe $G$ à partir de l'un de ses sommets $s$ toute liste $\ell$ de sommets telle que :

- le premier sommet de $\ell$ est $s$,
- chaque sommet de $S$ apparaît une fois et une seule dans $\ell$,
- tout sommet de la liste (sauf le premier) est adjacent à au moins un sommet placé avant lui dans la liste.

Ainsi, en notant $\mathcal{B}(T)$ la bordure d'un ensemble (ou d'une liste) $T$ non vide de sommets, défini par $\mathcal{B}(T) = \{x \in S - T \mid x \text{ adjacent à un sommet de } T\}$, on peut écrire une procédure de parcours générique :

```
ℓ := [s]
Pour k de 1 à |S| faire
   choisir un sommet u dans B(ℓ)
   ℓ := u :: ℓ
finPour
```

On peut alors montrer [BBC92, Prop. 4.1] que les parcours de $G$ sont exactement les listes construites par cette procédure. Finalement, les parcours en largeur et en profondeur sont des instances de cet algorithme, dans lequel on utilise respectivement une file et une pile (cachée dans les appels récursifs) pour choisir les sommets de la bordure.

Grâce à ce type de parcours, mentionnons qu'il est aussi possible de réaliser des parcours aléatoires modélisant par exemple un surfeur sur le web. L'algorithme de Dijkstra peut être vu comme un parcours généralisé dans lequel on utilise une file de priorité min pour choisir les sommets.

## 2.5 Algorithmes de plus courts chemins [CLRS04, Ch. 24 et 25]

On possède en entrée un graphe orienté pondéré $G = (S, A)$, avec une fonction de pondération $w : A \longrightarrow \mathbb{R}$ qui fait correspondre à chaque arc un poids de valeur réelle. La longueur du chemin $c = (v_0, v_1, \ldots, v_k)$ est la somme des poids de arcs qui le constituent : $w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. On définit la longueur du plus court chemin entre $u$ et $v$ par

$$\delta(u,v) = \begin{cases} \min\{w(c) \mid c : u \to v\} & \text{s'il existe un chemin de } u \text{ à } v \\ \infty & \text{sinon.} \end{cases}$$

Un plus court chemin de $u$ à $v$ est alors défini comme un chemin $c$ de longueur $w(c) = \delta(u, v)$.

Dans la suite, nous allons développer des algorithmes gloutons et de programmation dynamique, qui demandent donc d'exhiber une propriété de sous-structure optimale. C'est le but de ce lemme [CLRS04, Lem. 24.1].

**Lemme 2** (Sous-chemins de plus court chemin)**.** *Soit $c = (v_1, \ldots, v_k)$ un plus court chemin de $v_1$ à $v_k$. Pour tout $1 \leq i \leq j \leq k$, on note $c_{ij} = (v_i, \ldots, v_j)$ le sous-chemin de $c$ entre le sommet $v_i$ et le sommet $v_j$. Alors $c_{ij}$ est un plus court chemin de $v_i$ à $v_j$.*

**Plus courts chemins à origine unique** Dans un premier temps, on fixe une source unique $s \in S$ et on cherche à calculer les plus courts chemins de $s$ à chaque sommet de $G$. On étudie deux algorithmes : l'algorithme de Bellman-Ford et l'algorithme de Dijkstra.

Pour chaque sommet $v \in S$, on gère un attribut $d(v)$ qui est un majorant de la longueur d'un plus court chemin de $s$ à $v$. Par ailleurs, comme dans le cas du parcours en largeur, on représente les plus courts chemins par une arborescence de racine $s$ (qu'on manipule grâce aux informations $\pi(v)$). La procédure principale est celle de relâchement qu'on propose ci-dessous :

```
Relâcher(u, v, w)
    Si d(v) > d(u) + w(u, v)
    alors d(v) ← d(u) + w(u, v), π(v) ← u
    finSi
```

Par ailleurs, on suppose écrite une procédure SOURCE-UNIQUE-INITIALISATION$(G, s)$, qui initialise chaque parent $\pi(v)$ à NIL et chaque distance $d(v)$ à $\infty$, sauf pour l'origine où $d(s) = 0$.

**Exercice 18** (Algorithme de BELLMAN-FORD)**.** Dans cet exercice, il est important de noter que les poids des arcs peuvent avoir des valeurs négatives.

1. Existe-t-il nécessairement un plus court chemin entre deux sommets ?
2. On donne ci-dessous l'algorithme de Bellman-Ford :

```
BELLMAN-FORD(G, w, s)
    SOURCE-UNIQUE-INITIALISATION(G, s)
    Pour i de 1 à |S| − 1
        faire Pour chaque arc (u, v) ∈ A faire RELÂCHER(u, v, w) finPour
    finPour
    Pour chaque arc (u, v) ∈ A
        faire Si d(v) > d(u) + w(u, v) alors retourner FAUX finSi
    finPour
    retourner VRAI
```

   Quelle est la complexité de cet algorithme ?
3. On suppose ici que $G$ ne contient aucun circuit de longueur strictement négative accessible depuis $s$. Montrer qu'après les $|S| - 1$ itérations de la première boucle Pour, on a $d(v) = \delta(s, v)$ pour tous les sommets accessibles depuis $s$. En déduire que dans ce cas, l'algorithme retourne VRAI et que le sous-graphe des prédécesseurs $G_\pi$ est une arborescence de plus courts chemins de racine $s$.
4. On suppose ici que $G$ contient un circuit de longueur strictement négative accessible depuis $s$. Montrer que l'algorithme retourne FAUX.

**Exercice 19** (Algorithme de DIJKSTRA)**.** Dans cet exercice, on suppose que tous les arcs ont des poids positifs ou nuls. Le point faible de l'algorithme de BELLMAN-FORD est qu'il relâche plusieurs fois chaque arête. C'est ce point qu'on améliore ici, en utilisant une file de priorités min $F$, en prenant comme clé la valeur de l'attribut $d$ [CLRS04, Sec. 6.5]. C'est un algorithme glouton.

```
DIJKSTRA(G, w, s)
    SOURCE-UNIQUE-INITIALISATION(G, s)
    E ← ∅, F ← S
    Tant que F ≠ ∅
        faire u ← EXTRAIRE-MIN(F), E ← E ∪ {u}
            Pour chaque sommet v adjacent à u faire RELÂCHER(u, v, w) finPour
    finTantque
```

1. Montrer qu'à la fin de l'exécution de l'algorithme, pour tout sommet $u \in S$, $d(u) = \delta(s, u)$. Pour ce faire, on pourra employer l'invariant de boucle : au début de chaque itération de la boucle Tant que, pour chaque sommet $v \in E$, $d(v) = \delta(s, v)$.
2. Étudier la complexité de l'algorithme, selon le choix d'implémentation de la file de priorité min (tableau indexé de 1 à $|S|$, tas min binaire ou tas de Fibonacci).

**Plus courts chemins pour tout couple de sommets** On cherche ici à trouver les plus courts chemins entre tous les couples de sommets du graphe $G$. Si tous les arcs ont des poids positifs ou nuls, on peut utiliser l'algorithme de DIJKSTRA, en itérant l'algorithme pour toute origine : si on implémente la file de priorité à l'aide d'un tas de Fibonacci, on engendre alors un temps d'exécution en $\mathcal{O}(|S|^2 \log |S| + |S||A|)$. Si l'on autorise les arcs de poids négatif, l'algorithme de DIJKSTRA n'est plus utilisable. Il faut alors exécuter l'algorithme de BELLMAN-FORD, ce qui engendre un temps d'exécution en $\mathcal{O}(|S|^2|A|)$. On va montrer qu'il est possible de faire mieux dans l'exercice suivant.

**Exercice 20** (Algorithme de FLOYD-WARSHALL)**.** On suppose dans tout cet exercice qu'il n'existe pas de circuit de longueur strictement négative. On reprend les notations de l'algorithme de ROY-WARSHALL. Notons $d_{ij}^{(k)}$ la longueur d'un plus court chemin du sommet $i$ au sommet $j$ dont tous les sommets intérieurs sont dans l'ensemble $E(k)$.

1. Fournir une définition récursive de $d_{ij}^{(k)}$.
2. En déduire un algorithme de programmation dynamique calculant les longueurs des plus courts chemins entre tous sommets. Quelle est sa complexité ?
3. Expliquer comment on peut construire les plus courts chemins à partir de l'algorithme de FLOYD-WARSHALL.

## 2.6 Arbres couvrants de poids minimum [CLRS04, Ch. 23]

Dans cette section, on considère $G = (S, A)$ un graphe non orienté connexe pondéré, avec $w : A \longrightarrow \mathbb{R}$ la fonction de pondération. On souhaite trouver un arbre couvrant $T = (S, E)$ qui soit de poids $w(T) = \sum_{\{u,v\}\in E} w(u, v)$ minimum. On peut facilement concevoir un algorithme générique (suivant l'approche gloutonne) construisant un arbre couvrant de poids minimum (acm) :

```
ACM-GÉNÉRIQUE(G, w)
    E ← ∅
    Tant que E ne forme pas un arbre couvrant
        choisir une arête {u, v} tel que (S, E ∪ {u, v}) soit inclus dans un acm
        E ← E ∪ {u, v}
    finTantque
    retourner (S, E)
```

Comment trouver des arêtes $\{u, v\}$ sûres ? Pour un sous-ensemble $P \subseteq S$, on dit que $(P, S - P)$ est une coupe du graphe $G$ : c'est une partition de $S$. On dit qu'une arête $\{u, v\} \in A$ traverse la coupe $(P, S-P)$ si l'une de ses extrémités est un sommet de $P$ et l'autre un sommet de $S-P$. On dit de plus qu'elle la traverse minimalement, si elle est de poids minimal parmi les arêtes traversante. Finalement, on dit qu'une coupe respecte un ensemble d'arête $E$ si aucune arête de $E$ ne traverse la coupe.

**Exercice 21** (Arêtes sûres)**.**

1. Soit $E$ un sous-ensemble de $A$ inclus dans un acm de $G$ et soit $(P, S - P)$ une coupe de $G$ qui respecte $E$. Soit $\{u, v\} \in A$ une arête minimale traversant $(P, S - P)$. Montrer que $E \cup \{u, v\}$ est inclus dans un acm.
2. Soit $E$ un sous-ensemble de $A$ inclus dans un acm de $G$ et soit $C$ une composante connexe de la forêt $G_E = (S, E)$. Si $\{u, v\} \in A$ est une arête minimale reliant $C$ à une autre composante de $G_E$, alors $E \cup \{u, v\}$ est inclus dans un acm.

À partir de l'algorithme générique, on peut construire deux algorithmes : l'algorithme de KRUSKAL et l'algorithme de PRIM.

**Exercice 22** (Algorithme de KRUSKAL)**.** À chaque étape, cet algorithme trouve l'arête à ajouter à la forêt $(S, E)$ en choisissant une arête de poids minimal qui relie deux arbres de la forêt.

1. En commançant par trier les arêtes du graphe par ordre croissant de poids, décrire précisément l'algorithme.
2. Après avoir brièvement rappeler comment implémenter une structure d'ensemble disjoints, donner la complexité de cet algorithme.

**Exercice 23** (Algorithme de PRIM)**.** Contrairement à l'algorithme de KRUSKAL, l'algorithme de PRIM maintient toujours une arborescence avec une racine $r$ arbitraire (on représentera cette arborescence par des champs $\pi(u)$). À chaque étape, une arête minimale est ajoutée à cet arborescence pour relier celle-ci à un sommet isolé de $G_E = (S, E)$.

1. En utilisant une file de priorité min $F$ dont vous préciserez les clés, décrire précisément l'algorithme.
2. Étudier la complexité de l'algorithme, selon le choix d'implémentation de la file de priorité min (tas min binaire ou tas de Fibonacci). Comparer ces deux implémentations pour des graphes peu denses ($|A| = \Theta(|S|)$) ou des graphes denses ($|A| = \Theta(|S|^2)$).

## 2.7 Réseaux et flots [CLRS04, Ch. 26]

Un réseau de transport $G = (S, A)$ est un graphe orienté dans lequel chaque arc $(u, v) \in A$ se voit attribuer une capacité $c(u, v) \geq 0$, avec deux sommets particuliers : la source $s$ et le puits $t$. On suppose que chaque sommet se trouve sur un chemin reliant la source au puits. Un flot $f$ de $G$ est une fonction à valeurs réelles $f : S \times S \longrightarrow \mathbb{R}$ qui satisfait aux trois propriétés suivantes :

- Contrainte de capacité : pour tout $u, v \in S$, $f(u, v) \leq c(u, v)$.
- Symétrie : pour tout $u, v \in S$, $f(u, v) = -f(v, u)$.
- Conservation du flot : pour tout $u \in S - \{s, t\}$, on exige $\sum_{v \in S} f(u, v) = 0$.

On peut étendre $f$ à des couples d'ensembles de sommets $X, Y \subseteq S$ par $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$. La valeur d'un flot $f$ est définie par $|f| = \sum_{v \in S} f(s, v) = f(s, S)$, qui est par ailleurs égal à $f(S, t)$ (pourquoi ?). On cherche à résoudre le problème de la recherche de flot de valeur maximale.

**Exercice 24** (Algorithme de Ford-Fulkerson)**.**

1. Soient $u, v$ deux sommets de $G$. La capacité résiduelle de $(u, v)$ est $c_f(u, v) = c(u, v) - f(u, v)$. Le réseau résiduel de $G$ induit par $f$ est $G_f = (S, A_f)$ où $A_f = \{(u, v) \in S \times S \mid c_f(u, v) > 0\}$. Montrer que si $f'$ est un flot de $G_f$, alors la somme $f + f'$, définie par $(f + f')(u, v) = f(u, v) + f'(u, v)$, est un flot de $G$ de valeur $|f + f'| = |f| + |f'|$.
2. Un chemin améliorant est un chemin élémentaire de $s$ vers $t$ dans le réseau résiduel $G_f$. La capacité résiduelle d'un chemin améliorant $p$ est définie par $c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ appartient à } p\}$. Soit $f_p : S \times S \longrightarrow \mathbb{R}$ définie par

$$f_p(u, v) = \begin{cases} c_f(p) & \text{si } (u, v) \text{ appartient à } p \\ -c_f(p) & \text{si } (v, u) \text{ appartient à } p \\ 0 & \text{sinon.} \end{cases}$$

   Montrer que $f + f_p$ est un flot de $G$ de valeur $|f| + |f_p| > |f|$.
3. Une coupe $(E, T)$ d'un réseau de transport $G$ est une partition de $S$ dans $E$ et $T = S - E$ telle que $s \in E$ et $t \in T$. Si $f$ est un flot, alors le flot net à travers la coupe $(E, T)$ est défini par $f(E, T)$. Montrer que $f(E, T) = |f|$.
4. On appelle capacité de la coupe $(E, T)$ la valeur $c(E, T) = \sum_{u \in E} \sum_{v \in T} c(u, v)$. Une coupe minimum d'un réseau est une coupe dont la capacité est minimale rapportée à toutes les coupes du réseau. Montrer que la valeur d'un flot $f$ est majorée par la capacité d'une coupe quelconque de $G$.
5. Soit $f$ un flot dans un réseau $G$. Montrer que les conditions suivantes sont équivalentes :
   (*i*) $f$ est un flot maximum dans $G$.
   (*ii*) Le réseau résiduel $G_f$ ne contient aucun chemin améliorant.
   (*iii*) $|f| = c(E, T)$ pour une certaine coupe $(E, T)$ de $G$.
6. En déduire un algorithme calculant un flot maximum, en initialisant un flot $f$ à 0 puis à augmenter ce flot le long d'un chemin améliorant, tant qu'il en existe un.
7. Quelle est la complexité de cet algorithme dans le cas où les capacités du réseau sont des valeurs entières.

8. Dans le cas général, on améliore l'algorithme en cherchant les chemins améliorant par une recherche en largeur, c'est-à-dire si le chemin améliorant est un plus court chemin de $s$ vers $t$ dans le réseau résiduel : c'est l'algorithme d'Edmonds-Karp. On note $\delta_f(u, v)$ la distance d'un plus court chemin de $u$ à $v$ dans $G_f$, où chaque arc possède une distance unitaire. Montrer que si l'algorithme d'Edmonds-Karp est exécuté sur un réseau de transport $G$ de source $s$ et de puits $t$, alors pour tous les sommets $v \in S - \{s, t\}$, la distance du plus court chemin $\delta_f(s, v)$ dans le réseau résiduel $G_f$ augmente de façon monotone avec chaque augmentation de flot. Montrer que le nombre total d'augmentations de flot effectuées est $\mathcal{O}(|S||A|)$. En déduire la complexité de l'algorithme d'Edmonds-Karp.

On peut généraliser le problème de la recherche de flot maximum au cas où le réseau comporte plusieurs sources et/ou plusieurs puits. De plus, notons que ce problème n'est pas plus difficile que la restriction au cas de source et puits uniques : en effet, on peut toujours ajouter une supersource $s$ reliée à toutes les sources par un arc de capacité $\infty$ et ajouter un superpuits $t$ tels que tous les puits sont reliés à $t$ par un arc de capacité $\infty$.

On peut trouver des applications combinatoires étonnantes pouvant se ramener à un problème de flot maximum. Le problème du couplage maximum dans un graphe biparti en fait partie.

**Exercice 25** (Couplage biparti maximum par l'algorithme de Ford-Fulkerson)**.** Étant donné un graphe non orienté $G = (S, A)$, un *couplage* est un sous-ensemble d'arêtes $M \subseteq A$ tel que pour tous les sommets $v \in S$, au plus une arête de $M$ est incidente à $v$. On dit qu'un sommet $v$ est *couvert* par le couplage $M$ si une certaine arête de $M$ est incidente à $v$. Un couplage *maximum* est un couplage de cardinalité maximum.

On s'intéresse uniquement au problème de couplage maximum dans un graphe biparti. On suppose donc donné une partition $(L, R)$ de l'ensemble des sommets, tel que toutes les arêtes de $A$ passent entre $L$ et $R$. On suppose également que chaque sommet de $S$ a au moins une arête incidente.

1. Construire à partir de $G$ un réseau de transport $G' = (S', A')$ tel que tout couplage $M$ de $G$ induise un flot à valeurs entières $f$ de $G'$ tel que $|f| = |M|$.
2. Montrer réciproquement que si $f$ est un flot à valeurs entières de $G'$, alors il existe un couplage $M$ dans $G$ de cardinalité $|M| = |f|$.
3. Montrer que si la fonction de capacité d'un réseau de transport ne prend que des valeurs entières, alors le flot maximum $f$ produit par la méthode de Ford-Fulkerson est à valeurs entières.
4. En déduire que la cardinalité d'un couplage maximum d'un graphe biparti $G$ est la valeur d'un flot maximum du réseau de transport correspondant $G'$, ainsi qu'un algorithme permettant de générer un tel couplage maximum. Donner la complexité de cet algorithme.

Montrons pour finir que le problème de couplage biparti maximum peut être résolu plus efficacement à l'aide d'un algorithme dû à Hopcroft et Karp (adapté de [CLRS04, Prob. 26.7])

**Exercice 26** (Couplage biparti maximum par l'algorithme de Hopcroft-Karp)**.** Soit $G = (S, A)$ un graphe non orienté biparti, avec $S = L \cup R$ et où toutes les arêtes ont une et une seule extrémité dans $L$. Soit $M$ un couplage de $G$. On dit qu'une chaîne élémentaire $P$ de $G$ est une *chaîne améliorante* par rapport à $M$ si elle commence en un sommet non couplé de $L$, si elle finit en un sommet non couplé de $R$, et si ses arêtes appartiennent alternativement à $M$ et $A - M$. Dans cet exercice, on traite une chaîne comme une séquence d'arêtes, plutôt que comme une séquence de sommets. Une plus courte chaîne améliorante par rapport à un couplage $M$ est donc une chaîne améliorante ayant un nombre minimum d'arêtes. On définit la différence symétrique de deux ensembles $A$ et $B$ comme $A \Delta B = (A - B) \cup (B - A)$.

1. La notion de chaîne améliorante est liée, bien que très différente, à la notion de chemin améliorant d'un réseau de transport. En vous aidant d'un parcours en largeur, montrer qu'on peut trouver efficacement un ensemble maximum de plus courtes chaînes améliorantes sans sommet commun pour un couplage $M$ donné.

2. Montrer que si $M$ est un couplage et $P$ une chaîne améliorante par rapport à $M$, alors la différence symétrique $M\Delta P$ est un couplage et $|M\Delta P| = |M|+1$. Montrer que si $P_1, \ldots, P_k$ sont des chaînes améliorantes par rapport à $M$ sans sommet commun, alors la différence symétrique $M\Delta(P_1 \cup \cdots \cup P_k)$ est un couplage de cardinalité $|M| + k$.

   On en déduit l'algorithme correct suivant :

```
HOPCROFT-KARP(G)
    M ← ∅
    Répéter
        Soit P ← {P_1, ..., P_k} un ensemble maximum de plus courtes
              chaînes améliorantes par rapport à M sans sommet commun
        M ← MΔ(P_1 ∪ ··· ∪ P_k)
    Jusqu'à P = ∅
    retourner M
```

   On cherche désormais à borner le nombre d'itérations de la boucle Répéter pour prouver la terminaison de l'algorithme, et majorer sa complexité en temps.

3. Étant donnés deux couplages $M$ et $M'$ tels que $|M| \leq |M'|$, montrer que le graphe $G' = (S, M\Delta M')$ contient $|M'| - |M|$ chaînes améliorantes par rapport à $M$ sans sommet commun.

4. On va montrer qu'à chaque itération de la boucle, la longueur commune des plus courtes chaînes améliorantes par rapport à $M$ croît strictement. Plus précisément, soit $\ell$ la longueur d'une plus courte chaîne améliorante par rapport à $M$ et soient $\mathcal{P} = \{P_1, \ldots, P_k\}$ un ensemble maximum de chaînes améliorantes de longueur $\ell$ par rapport à $M$, sans sommet commun. Soit $M' = M\Delta(P_1 \cup \cdots \cup P_k)$, et supposons que $P$ est une plus courte chaîne améliorante par rapport à $M'$.
   (a) Montrer que si $P$ n'a pas de sommet commun avec $\mathcal{P}$, alors $P$ a plus de $\ell + 1$ arêtes.
   (b) Supposons maintenant que $P$ a au moins un sommet commun avec $\mathcal{P}$. Soit $B$ l'ensemble des arêtes $(M\Delta M')\Delta P$. Montrer que $B = (P_1 \cup \cdots \cup P_k)\Delta P$ et que $|B| \geq (k+1)\ell$. En conclure que $P$ a plus de $\ell + 1$ arêtes.

5. En conclure que le nombre d'itérations de la boucle est majoré par $2\sqrt{|S|}$, et donner le temps d'exécution total de l'algorithme.

### 2.8 Exemples de problèmes de graphes NP-complets

Mentionnons pour finir qu'un grand nombre de problèmes de graphe sont des problèmes NP-complets : CLIQUE, VERTEX-COVER, CYCLE-HAMILTONIEN, VOYAGEUR-COMMERCE, $k$-COLOR...

## Références


[BBC92] Danièle Beauquier, Jean Berstel, and Philippe Chrétienne. *Eléments d'algorithmique*. Masson, Disponible en ligne sur http://denif.ens-lyon.fr/data/elements_algorithmique/1992/contenu/Elements.pdf, 1992.

[BY95] Jean-Daniel Boissonnat and Mariette Yvinec. *Géométrie Algorithmique*. Ediscience, 1995.

[CLRS04] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction à l'algorithmique : Cours et exercices (2ème édition)*. Dunod, 2004.

[DPV06] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh V. Vazirani. *Algorithms*. Disponible en ligne sur http://www.cs.berkeley.edu/~vazirani/algorithms/all.pdf, 2006.