Cours de Complexité/Algorithmique

HMIN329 Année 2017-2018

Version 1.0

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

ANNIE CHATEAU ET RODOLPHE GIROUDEAU
161, RUE ADA
34392 MONTPELLIER CEDEX 5
TEL : 04-67-41-85-40
MAIL : {CHATEAU,RGIROU}@LIRMM.FR

Table des matières

Table des figures

Liste des Algorithmes

Chapitre

1 Notions de bases

Contents

Résumé

Ce chapitre est dédié aux notions de base en informatique théorique.

1.1 Logique et raisonnements

Il est important d'avoir un langage rigoureux. La langue française est souvent ambigüe. Prenons l'exemple de la conjonction « ou » ; au restaurant « fromage ou dessert » signifie l'un ou l'autre mais pas les deux. Par contre si dans un jeu de carte on cherche « les as ou les cœurs » alors il ne faut pas exclure l'as de cœur. Autre exemple : que répondre à la question « As-tu 10 euros en poche ? » si l'on dispose de 15 euros ?
Autre exemple « Est-ce qu'une augmentation de 20%, puis de 30% est plus intéressante qu'une augmentation de 50% ? ». Vous pouvez penser « oui » ou « non », mais pour en être sûr il faut suivre une démarche logique qui mène à la conclusion. Cette démarche doit être convaincante pour tout le monde. On parle de raisonnement. Les mathématiques sont un langage pour s'exprimer rigoureusement, adapté aux phénomènes complexes, qui rend les calculs exacts et vérifiables. Le raisonnement est le moyen de valider – ou d'infirmer– une hypothèse et de l'expliquer à autrui.

1.1.1 Logique des propositions

Une *assertion* est une phrase soit vraie, soit fausse, pas les deux en même temps. Exemples :

- « Il pleut. »
- « Je suis plus grand que toi. »
- « $2 + 2 = 4$ »
- « $2 \times 3 = 7$ »
- « Pour tout $x \in \mathbb{R}$, on a $x^2 \geq 0$. »
- « Pour tout $z \in \mathbb{C}$, on a $|z| = 1$. »

Si P est une assertion et Q est une autre assertion, nous allons définir de nouvelles assertions construites à partir de P et de Q.

1.1.1.1 L'opérateur logique « et »

L'assertion « P et Q » est vraie si P est vraie et Q est vraie. L'assertion « P et Q » est fausse sinon. On résume ceci en une table de vérité :
Par exemple si P est l'assertion « Cette carte est un as » et Q l'assertion « Cette carte est cœur » alors l'assertion « P et Q » est vraie si la carte est l'as de cœur et est fausse pour toute autre carte.

P \ Q	V	F
V	V	F
F	F	F

FIGURE 1.1 – Table de vérité de « P et Q »

1.1.1.2 L'opérateur logique « ou »

L'assertion « P ou Q » est vraie si l'une des deux assertions P ou Q est vraie. L'assertion « P ou Q » est fausse si les deux assertions P et Q sont fausses. Si P est l'assertion « Cette carte est un as » et Q l'assertion « Cette carte est cœur » alors l'assertion « P ou Q » est vraie si la carte est un as ou bien un cœur (en particulier elle est vraie pour l'as de cœur).
Remarque Pour définir les opérateurs « ou », « et » on fait appel à une phrase en français utilisant les mots ou, et ! Les tables de vérités permettent d'éviter ce problème.

1.1.1.3 La négation « non »

L'assertion « non P » est vraie si P est fausse, et fausse si P est vraie.

1.1.1.4 L'implication $\Rightarrow$

La définition mathématique est la suivante : L'assertion « (non P) ou Q » est notée « $P \Rightarrow Q$ ». L'assertion « $P \Rightarrow Q$ » se lit en français « P implique Q ». Elle se lit souvent aussi « si P est vraie alors Q est vraie » ou « si P alors Q ». Par exemple :

- « $0 \leq x \leq 25 \Rightarrow \sqrt{x} \leq 5$ » est vraie (prendre la racine carrée).
- « $x \in]-\infty, -4[\Rightarrow x^2 + 3x - 4 > 0$ » est vraie (étudier le binôme).
- « $\sin(\theta) = 0 \Rightarrow \theta = 0$ » est fausse (regarder pour $\theta = 2\pi$ par exemple).
- « $2 + 2 = 5 \Rightarrow 2 = 2$ » est vraie ! Eh oui, si P est fausse alors l'assertion « $P \Rightarrow Q$ » est toujours vraie.

1.1.1.5 L'équivalence $\Leftrightarrow$

L'équivalence est définie par : « $P \Leftrightarrow Q$ » est l'assertion « $(P \Rightarrow Q)$ et $(Q \Rightarrow P)$ ». On dira « P est équivalent à Q » ou « P équivaut à Q » ou « P si et seulement si

Q ». Cette assertion est vraie lorsque P et Q sont vraies ou lorsque P et Q sont fausses. La table de vérité est :
Exemples :

- Pour $x, x_0 \in \mathbb{R}$, l'équivalence « $x \times x_0 = 0 \Leftrightarrow (x = 0$ ou $x_0 = 0)$ » est vraie.
- Voici une équivalence toujours fausse (quelque soit l'assertion P) : « $P \Leftrightarrow$ non(P) ».

Lemme 1.1.1 *Soient P,Q, R trois assertions. Nous avons les équivalences (vraies) suivantes :*

1. $P \Leftrightarrow$ *non*$($*non*$(P))$
2. $(P$ *et* $Q) \Leftrightarrow (Q$ *et* $P)$
3. $(P$ *ou* $Q) \Leftrightarrow (Q$ *ou* $P)$
4. *non*$(P$ *et* $Q) \Leftrightarrow ($*nonP*$)$ *ou* $($*nonQ*$)$
5. *non*$(P$ *ou* $Q) \Leftrightarrow ($*nonP*$)$ *et* $($*nonQ*$)$
6. P *et* $(Q$ *ou* $R) \Leftrightarrow (P$ *et* $Q)$ *ou* $(P$ *et* $R)$
7. P *ou* $(Q$ *et* $R) \Leftrightarrow (P$ *ou* $Q)$ *et* $(P$ *ou* $R)$
8. $P \Rightarrow Q \Leftrightarrow$ *non*$(Q) \Rightarrow$ *non*(P)

1.1.2 Quantificateurs

1.1.2.1 Le quantificateur $\forall$: « pour tout »

Une assertion P peut dépendre d'un paramètre x, par exemple « $x^2 \geq 1$ », l'assertion $P(x)$ est vraie ou fausse selon la valeur de x. L'assertion $\forall x \in E \;\; P(x)$ est une assertion vraie lorsque les assertions $P(x)$ sont vraies pour tous les éléments x de l'ensemble E. On lit « Pour tout x appartenant à E, $P(x)$ », sous-entendu « Pour tout x appartenant à E, $P(x)$ est vraie ». Par exemple :

- « $\forall x \in [1, +\infty[\;\; (x^2 \geq 1)$ » est une assertion vraie.
- « $\forall x \in \mathbb{R} \;\; (x^2 \geq 1)$ » est une assertion fausse.
- « $\forall n \in \mathbb{N} \;\; n(n+1)$ est divisible par 2 » est vraie.

1.1.2.2 Le quantificateur $\exists$: « il existe »

L'assertion $\exists x \in E \quad P(x)$ est une assertion vraie lorsque l'on peut trouver au moins un x de E pour lequel $P(x)$ est vraie. On lit « il existe x appartenant à E tel que $P(x)$ (soit vraie) ». Par exemple :

- « $\exists x \in \mathbb{R} \quad (x(x-1) < 0)$ » est vraie (par exemple $x = \frac{1}{2}$ vérifie bien la propriété).
- « $\exists n \in \mathbb{N} \quad n^2 - n > n$ » est vraie (il y a plein de choix, par exemple $n = 3$ convient, mais aussi $n = 10$ ou même $n = 100$, un seul suffit pour dire que l'assertion est vraie).
- « $\exists x \in \mathbb{R} \quad (x^2 = -1)$ » est fausse (aucun réel au carré ne donnera un nombre négatif).

1.1.2.3 La négation des quantificateurs

La négation de « $\forall x \in E \quad P(x)$ » est « $\exists x \in E \quad \text{non}P(x)$ ». Par exemple la négation de « $\forall x \in [1, +\infty[\quad (x^2 \geq 1)$ » est l'assertion « $\exists x \in [1, +\infty[\quad (x^2 < 1)$ ». En effet la négation de $x^2 \geq 1$ est non$(x^2 \geq 1)$ mais s'écrit plus simplement $x^2 < 1$.
La négation de « $\exists x \in E \quad P(x)$ » est « $\forall x \in E \quad \text{non}P(x)$ ». Voici des exemples :

- La négation de « $\exists z \in \mathbb{C} \quad (z^2+z+1 = 0)$ » est « $\forall z \in \mathbb{C} \quad (z^2+z+1 \neq 0)$ ».
- La négation de « $\forall x \in \mathbb{R} \quad (x+1 \in \mathbb{Z})$ » est « $\exists x \in \mathbb{R} \quad (x+1 \notin \mathbb{Z})$ ».
- Ce n'est pas plus difficile d'écrire la négation de phrases complexes. Pour l'assertion : $\forall x \in \mathbb{R} \exists y > 0 \quad (x + y > 10)$ sa négation est $\exists x \in \mathbb{R} \forall y > 0 \quad (x + y \leq 10)$.

Remarques
L'ordre des quantificateurs est très important. Par exemple les deux phrases logiques $\forall x \in \mathbb{R} \exists y \in \mathbb{R} \quad (x + y > 0)$ et $\exists y \in \mathbb{R} \forall x \in \mathbb{R} \quad (x + y > 0)$. sont différentes. La première est vraie, la seconde est fausse. En effet une phrase logique se lit de gauche à droite, ainsi la première phrase affirme « Pour tout réel x, il existe un réel y (qui peut donc dépendre de x) tel que $x + y > 0$. » (par exemple on peut prendre $y = x + 1$). C'est donc une phrase vraie. Par contre la deuxième se lit : « Il existe un réel y, tel que pour tout réel x, $x + y > 0$. » Cette phrase est fausse, cela ne peut pas être le même y qui convient pour tous les x ! On retrouve la même différence dans les phrases en français suivantes. Voici une phrase vraie « Pour toute personne, il existe un numéro de téléphone », bien sûr le numéro dépend de la personne. Par contre cette phrase est fausse : « Il existe

un numéro, pour toutes les personnes ». Ce serait le même numéro pour tout le monde !
Terminons avec d'autres remarques.

- Quand on écrit « $\exists x \in \mathbb{R} \quad (f(x) = 0)$ » cela signifie juste qu'il existe un réel pour lequel f s'annule. Rien ne dit que ce x est unique. Dans un premier temps vous pouvez lire la phrase ainsi : « il existe au moins un réel x tel que $f(x) = 0$ ». Afin de préciser que f s'annule en une unique valeur, on rajoute un point d'exclamation : $\exists! x \in \mathbb{R} \ \ (f(x) = 0)$.
- Pour la négation d'une phrase logique, il n'est pas nécessaire de savoir si la phrase est fausse ou vraie. Le procédé est algorithmique : on change le « pour tout » en « il existe » et inversement, puis on prend la négation de l'assertion P.
- Pour la négation d'une proposition, il faut être précis : la négation de l'inégalité stricte « $<$ » est l'inégalité large « $\geq$ », et inversement.
- Les quantificateurs ne sont pas des abréviations. Soit vous écrivez une phrase en français : « Pour tout réel x, si $f(x) = 1$ alors $x \geq 0$. » , soit vous écrivez la phrase logique : $\forall x \in \mathbb{R} \ \ (f(x) = 1 \Rightarrow x \geq 0)$. Mais surtout n'écrivez pas « $\forall x$ réel, si $f(x) = 1 \Rightarrow x$ positif ou nul ». Enfin, pour passer d'une ligne à l'autre d'un raisonnement, préférez plutôt « donc » à « $\Rightarrow$ ».
- Il est défendu d'écrire $\nexists$, $\not\Rightarrow$. Ces symboles n'existent pas !

1.2 Raisonnements

1.2.1 Raisonnement direct

On veut montrer que l'assertion « $P \Rightarrow Q$ » est vraie. On suppose que P est vraie et on montre qu'alors Q est vraie. C'est la méthode à laquelle vous êtes le plus habitué.

Exemple Montrer que si $a, b \in \mathbb{Q}$ alors $a + b \in \mathbb{Q}$.

Démonstration
Prenons $a \in \mathbb{Q}, b \in \mathbb{Q}$. Rappelons que les rationnels $\mathbb{Q}$ sont l'ensemble des réels s'écrivant $\frac{p}{q}$ avec $p \in \mathbb{Z}$ et $q \in \mathbb{N}^*$.
Alors $a = \frac{p}{q}$ pour un certain $p \in \mathbb{Z}$ et un certain $q \in \mathbb{N}^*$. De même $b = \frac{p_0}{q_0}$ avec

$p_0 \in \mathbb{Z}$ et $q_0 \in \mathbb{N}^*$. Maintenant

$$a + b = \frac{p}{q} + \frac{p_0}{q_0} = \frac{pq_0 + qp_0}{qq_0}$$

Or le numérateur $pq_0 + qp_0$ est bien un élément de $\mathbb{Z}$; le dénominateur qq_0 est lui un élément de $\mathbb{N}^*$. Donc $a + b \in \mathbb{Q}$.

1.2.2 Contraposée

Le raisonnement par contraposition est basé sur l'équivalence suivante (voir la proposition 1) : L'assertion « $P \Rightarrow Q$ » est équivalente à « non$(Q) \Rightarrow$ non(P) ». Donc si l'on souhaite montrer l'assertion « $P \Rightarrow Q$ », on montre en fait que si non(Q) est vraie alors non(P) est vraie.

Exemple
Soit $n \in \mathbb{N}$. Montrer que si n^2 est pair alors n est pair.

Démonstration
Nous supposons que n n'est pas pair. Nous voulons montrer qu'alors n^2 n'est pas pair. Comme n n'est pas pair, il est impair et donc il existe $k \in \mathbb{N}$ tel que $n = 2k+1$. Alors $n^2 = (2k+1)^2 = 4k^2+4k+1 = 2k'+1$ avec $k' = 2k^2+2k \in \mathbb{N}$. Et donc n^2 est impair.
Conclusion : nous avons montré que si n est impair alors n^2 est impair. Par contraposition ceci est équivalent à : si n^2 est pair alors n est pair.

1.2.3 Absurde

Le raisonnement par l'absurde pour montrer « $P \Rightarrow Q$ » repose sur le principe suivant : on suppose à la fois que P est vraie et que Q est fausse et on cherche une contradiction. Ainsi si P est vraie alors Q doit être vraie et donc « $P \Rightarrow Q$ » est vraie.

Exemple
Soient $a, b \geq 0$. Montrer que si $\frac{a}{1+b} = \frac{b}{1+a}$ alors $a = b$.

Démonstration
Nous raisonnons par l'absurde en supposant que $\frac{a}{1+b} = \frac{b}{1+a}$ et $a \neq b$. Comme $\frac{a}{1+b} = \frac{b}{1+a}$ alors $a(1+a) = b(1+b)$ donc $a + a^2 = b + b^2$ d'où $a^2 - b^2 = b - a$. Cela conduit à $(a-b)(a+b) = -(a-b)$. Comme $a \neq b$ alors $a - b \neq 0$ et donc

en divisant par $a-b$ on obtient $a+b=-1$. La somme de deux nombres positifs ne peut être négative. Nous obtenons une contradiction.
Conclusion : si $\frac{a}{1+b} = \frac{b}{1+a}$ alors $a = b$. Dans la pratique, on peut choisir indifféremment entre un raisonnement par contraposition ou par l'absurde. Attention cependant de bien écrire quel type de raisonnement vous choisissez et surtout de ne pas changer en cours de rédaction !

1.2.4 Récurrence

Le principe de récurrence permet de montrer qu'une assertion $P(n)$, dépendant de n, est vraie pour tout $n \in \mathbb{N}$. La démonstration par récurrence se déroule en trois étapes : lors de l'initialisation on prouve $P(0)$. Pour l'étape d'hérédité ou de récurrence, on suppose $n \geq 0$ donné avec $P(n)$ vraie, et on démontre alors que l'assertion $P(n+1)$ au rang suivant est vraie. Enfin dans la conclusion, on rappelle que par le principe de récurrence $P(n)$ est vraie pour tout $n \in N$.

Exemple
Montrer que pour tout $n \in N$, $2^n > n$.

Démonstration
Pour $n \geq 0$, notons $P(n)$ l'assertion suivante : $2^n > n$. Nous allons démontrer par récurrence que $P(n)$ est vraie pour tout $n \geq 0$. Initialisation. Pour $n = 0$ nous avons $2^0 = 1 > 0$. Donc $P(0)$ est vraie. Hérédité. Fixons $n \geq 0$. Supposons que $P(n)$ soit vraie. Nous allons montrer que $P(n+1)$ est vraie.

$$\begin{aligned} 2^{n+1} &= 2^n + 2^n \\ &> n + 2^n \quad \text{car par } P(n) \text{ nous savons } 2^n > n, \\ &> n + 1 \quad \text{car } 2^n \geq 1. \end{aligned}$$

Donc $P(n+1)$ est vraie. Conclusion. Par le principe de récurrence $P(n)$ est vraie pour tout $n \geq 0$, c'est-à-dire $2^n > n$ pour tout $n \geq 0$.

- La rédaction d'une récurrence est assez rigide. Respectez scrupuleusement la rédaction proposée : donnez un nom à l'assertion que vous souhaitez montrer (ici $P(n)$), respectez les trois étapes (même si souvent l'étape d'initialisation est très facile). En particulier méditez et conservez la première ligne de l'hérédité « Fixons $n \geq 0$. Supposons que $P(n)$ soit vraie. Nous allons montrer que $P(n+1)$ est vraie. »
- Si on doit démontrer qu'une propriété est vraie pour tout $n \geq n_0$, alors on commence l'initialisation au rang n_0.
- Le principe de récurrence est basé sur la construction de $\mathbb{N}$. En effet un

des axiomes pour définir $\mathbb{N}$ est le suivant : « Soit A une partie de $\mathbb{N}$ qui contient 0 et telle que si $n \in A$ alors $n+1 \in A$. Alors $A = \mathbb{N}$ ».

1.2.5 Principe d'induction

Soit un prédicat $P(n)$ sur $\mathbb{N}$. Si les deux propriétés suivantes sont vérifiées :

1. Base : $P(0)$ est vraie.
2. Induction : $\forall n \in \mathbb{N}$, si $(\forall k \in \mathbb{N} \,|\, k \leqslant n,\ P(k)$ est vraie), alors $P(n+1)$ est vraie.

alors $\forall n \in \mathbb{N} : P(n)$ est vraie.

Remarque : Les 2 principes (induction et récurrence) sont équivalents sur $\mathbb{N}$, tout ce qui peut se démontrer par l'un peut se démontrer par l'autre.

1.3 Ensembles, applications, dénombrement

1.3.1 Définitions de base

Un ensemble est une collection d'objets distincts où l'ordre n'a pas d'importance. L'ensemble vide, noté $\emptyset$, n'a aucun élément. Si E est un ensemble non vide il a au moins un élément x qui *appartient* à E et l'on note $x \in E$. La négation de cette relation : x *n'appartient pas* à E se note $x \notin E$.
Un ensemble *discrêt* est un ensemble qui n'est pas *continu* (l'ensemble $\mathbb{R}$ est continu, l'ensemble $\mathbb{N}$ est discrêt). Un ensemble qu'on peut décrire par la suite de ses *éléments* est *fini* donc discret. On peut le définir en *extension*. *Exemple* : $\{1,2,3\}, \{\} = \emptyset, \{\texttt{vrai}, \texttt{faux}\}$. Un ensemble infini ne peut être donné en extension. On doit donc le définir autrement, par une propriété. Ils sont alors définis en *compréhension* (ou *intention*). *Exemple :* $\mathbb{N}$, tous les entiers qui sont pairs, plus formellement : $\{n \in \mathbb{N} \,|\, \exists k \in \mathbb{N}, n = 2 \times k\}$.

Comparaison d'ensembles Dans la suite E est l'ensemble de référence et A et B des parties de E
Inclusion Un ensemble A est dit contenu dans ou inclus dans un ensemble B si *chaque* élément de A est élément de B. On note : $A \subseteq B$ si $\forall x \in A, x \in B$. On dit A est un *sous-ensemble* de B, ou encore A est une partie de B. A est strictement inclus dans B si $A \subseteq B$ et $A \neq B$. A est dit sous-ensemble *propre* ou *strict* de B.

Non inclusion $A \not\subseteq B$ s'il y a au moins un élément de A qui n'est pas élément de B, ce qui s'écrit $\exists x \in A \,|\, x \notin B$.
Égalité $A = B$ si et seulement si $A \subseteq B$ et $B \subseteq A$ (Manière très classique de prouver l'égalité entre 2 ensembles).

Parties d'un ensemble On note $\mathcal{P}(E)$ l'ensemble des parties de E. $\mathcal{P}(E)$ est l'ensemble exhaustif de toutes les parties de E. On a par définition : $A \subseteq E$ ssi $A \in \mathcal{P}(E)$.
Exemple : $\mathcal{P}(\{1,2,3\}) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

1.3.2 Opérations sur les ensembles

$A \cup B = \{x \,|\, x \in A \text{ ou } x \in B\}$	union
$A \cap B = \{x \,|\, x \in A \text{ et } x \in B\}$	intersection
$B \setminus A = \{x \,|\, x \in B \text{ et } x \notin A\}$	différence
$\overline{A}^E = E \setminus A \quad$ pour $A \subseteq E$	complémentaire
$A \times B = \{(x,y) \,|\, x \in A \text{ et } y \in B\}$	produit cartésien

Propriétés de ces opérations

1. L'union et l'intersection sont des opérations
   — associatives : $(A \cup B) \cup C = A \cup (B \cup C) = A \cup B \cup C$
   — commutatives : $A \cap B = B \cap A$
   — distributives l'une par rapport à l'autre : $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ et $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
   — Loi de De Morgan : $\overline{A \cap B} = \overline{A} \cup \overline{B}$ et $\overline{A \cup B} = \overline{A} \cap \overline{B}$.
2. Deux ensembles dont l'intersection est vide sont appelés *disjoints*.
3. Le produit cartésien se généralise à une famille finie d'ensembles :
   $E_1 \times E_2 \times \ldots \times E_n = \{(e_1, e_2, \ldots, e_n) \,|\, e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$
   $(e_1, e_2, \ldots, e_n)$ est appelé un *n-uplet*. Autre notation souvent utilisée : E^m pour $E \times E \times \ldots \times E$ m fois.
4. Attention : une *paire* est un ensemble à 2 éléments, par exemple $\{a, b\}$, alors qu'un *couple* est un 2-uplet, par exemple (a, b). Deux couples (a, b) et (c, d)

sont égaux ssi $a = c$ et $b = d$, alors que deux paires $\{a, b\}$ et $\{c, d\}$ sont égales ssi ($a = c$ et $b = d$) ou ($a = d$ et $b = c$).

Partition d'un ensemble Une *partition* P d'un ensemble E est :

1. un ensemble non vide P (pas forcément fini) de parties non vides de E
2. les parties sont toutes disjointes : si $A_i \neq A_j$ sont deux parties éléments de P, alors $A_i \cap A_j = \emptyset$.
3. les parties élément de P recouvrent E : si $P = \{A_1, A_2, \ldots, A_n\}$ alors $A_1 \cup A_2 \cup \ldots \cup A_n = E$, se note aussi $\bigcup_{i \in 1..n} A_i = E$.

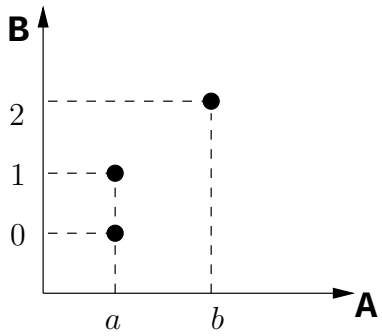
Trois partitions remarquables :

— *les singletons :* on associe à chaque élément $e \in E$ un ensemble à 1 élément, le singleton $E_e = \{e\}$;
— *la partition pleine :* contient un seul ensemble $E_1 = E$.
— Pour un sous-ensemble $A \subsetneq E$ non vide, $\{A, \overline{A}^E\}$ est une partition.

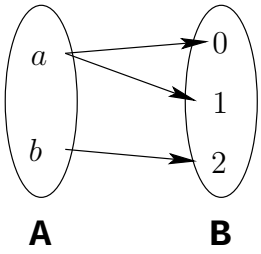
1.3.3 Relations binaires

Une *relation binaire* $\mathcal{R}$ d'un ensemble X vers un ensemble Y est définie par un sous-ensemble du produit cartésien $\mathcal{R} \subseteq X \times Y$, appelé le *graphe* de la relation. Pour $(x, y) \in \mathcal{R}$, on note $x\mathcal{R}y$ et on dit que x et y sont en *relation*. (On dit aussi y est associé à x par $\mathcal{R}$, ou encore que y est image de x par $\mathcal{R}$).
Exemple : $A = \{a, b\}$ et $B = \{0, 1, 2\}$. $\mathcal{R} = \{(a, 0), (a, 1), (b, 2)\}$ est une relation binaire de A vers B. $a\mathcal{R}0$, $a\mathcal{R}1$, ...
On *représente* une relation binaire de A vers B, par différentes sortes de diagrammes :

— diagramme cartésien
— diagramme sagittal (comme on fait dans les diagrammes de Venn : patatoïdes et flèches).



Le diagramme cartésien de $\mathcal{R}$



Le diagramme sagittal de $\mathcal{R}$

Relation réciproque La relation *réciproque* d'une relation $\mathcal{R}$ de A vers B est notée $\mathcal{R}^{-1}$. $\mathcal{R}^{-1}$ est une relation de B vers A, c-à-d que $\mathcal{R}^{-1} \subseteq B \times A$. $(b, a) \in \mathcal{R}^{-1}$ ssi $(a, b) \in \mathcal{R}$, autrement dit $a\mathcal{R}b \Leftrightarrow b\mathcal{R}^{-1}a$.
Exemple : $\mathcal{R}^{-1} = \{(0, a), (1, a), (2, b)\}$

1.3.4 Fonctions

Une relation binaire de X vers Y est *fonctionnelle* si pour tout $x \in X$, il existe *au plus un* élément $y \in Y$ en relation avec x.
En notation fonctionnelle pour une fonction f de X vers Y, on note
$f : \begin{array}{ccc} X & \longrightarrow & Y. \\ x & \longmapsto & f(x) \end{array}$

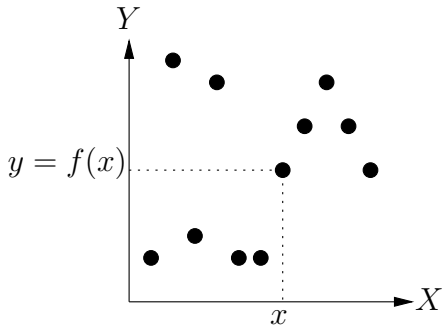
- f est incluse dans $X \times Y$: $f \subseteq X \times Y$.
- X est l'ensemble de *départ* et Y l'ensemble *d'arrivée*.
- L'ensemble $Dom(f) = \{x \in X \mid \exists y \in Y \text{ avec } y = f(x)\}$ est le *domaine* ou *ensemble de définition* de f, $Dom(f) \subseteq X$.
- L'ensemble $Im(f) = \{y \in Y \mid \exists x \in X \text{ avec } y = f(x)\}$ est l'*image* de f, $Im(f) \subseteq Y$.
- **L**'*image* de $x \in X$ par f est l'élément y de Y tel que $y = f(x)$.
- **Un** *antécédent* par f d'un élément y de Y est un élément x tel que $y = f(x)$.

Image directe et réciproque On se donne une fonction $f : X \longrightarrow Y$. Soit A une partie de X et B une partie de Y. On définit *l'image directe* de A par f : $f(A) = \{f(x) \mid x \in A\} = \{y \in Y \mid \exists x \in A \text{ avec } y = f(x)\}$ ($f(A)$ est l'ensemble des images par f des éléments de A), et *l'image réciproque* de B par

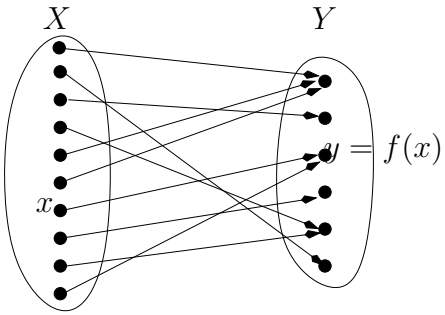
$f : f^{-1}(B) = \{x \in X \mid f(x) \in B\}$ ($f^{-1}(B)$ est l'ensemble des antécédents par f des éléments de B).
Avec ces notations, on a $f(X) = Im(f)$ et $f^{-1}(Y) = Dom(f)$.

1.3.5 Applications

Une fonction f de X vers Y est une *application* si son domaine est l'ensemble X tout entier : $Dom(f) = X$.
L'ensemble des applications de X vers Y est noté Y^X ou $\{X \longrightarrow Y\}$.
L'ensemble $\{(x, f(x)) \in X \times Y\}$ est appelé le *graphe* de l'application f. Il définit l'application f en donnant tous les couples $(x, f(x))$.



Le graphe d'une application f de X vers Y représentée par son diagramme cartésien.



Le diagramme sagittal d'une application $f : X \to Y$.

Attention : une application est une fonction mais une fonction n'est pas toujours une application.

Restriction, co-restriction, prolongement Si $A \subseteq X$, on peut *restreindre* l'ensemble de départ de f au sous-ensemble A, on note $\mathbf{f|_A} : A \longrightarrow Y$, $x \longmapsto f(x)$ — $f|_A$ *coïncide* avec f sur A.
La *co-restriction* est l'opération analogue sur un sous-ensemble de l'ensemble d'arrivée (ici Y).
Soit $f : X \longrightarrow Y$ et $g : X \cup Z \longrightarrow Y$, g est un *prolongement* de f si g coïncide avec f sur $Dom(f)$. Noter que la restriction du prolongement g au domaine de définition de la fonction prolongée f est f elle-même : $g|_{Dom(f)} = f$. *Exemple :*

$\mathbf{r} : \mathbb{N}^* \longrightarrow \mathbb{Q}$, s définie par : $\mathbf{s} : \mathbb{N} \longrightarrow \mathbb{Q}$ est un

$x \longmapsto \frac{1}{x}$

$$x \longmapsto \begin{cases} \text{r(x)} & \text{si } x \neq 0 \\ 12 & x = 0 \end{cases}$$

prolongement de r au point 0. En général on définit un prolongement pour remplacer la fonction prolongée.

Injection, surjection, bijection, réciproque L'application est *injective* si chaque élément $y \in Y$ a au plus un antécédent. Elle est *surjective* si chaque élément $y \in Y$ a au moins un antécédent. Elle est *bijective* si c'est une application injective et surjective, c-à-d chaque élément $y \in Y$ a exactement un antécédent. On dit alors que les ensembles de départ et d'arrivée sont *équipotents*.
Dans le cas où $f : X \longrightarrow Y$ est une application bijective, on peut définir l'application réciproque de f, qui est aussi bijective :

$$\mathbf{f^{-1}} : Y \longrightarrow X \quad ,$$
$$y \longmapsto \text{L'unique } x \text{ tel que } f(x) = y$$

1.3.6 Cardinalité

Les ensembles considérés ici sont

- ou bien finis, comportant n éléments. On dit alors que leur *cardinal* est n. On note $card(E) = |E| = n$ pour un tel ensemble E. Dans ce cas particulier, E est *équipotent* à $[1, n]_{\mathbb{N}} = \{1, 2, \ldots, n\}$. Une telle équipotence ou sa réciproque $f : [1, n]_{\mathbb{N}} \longrightarrow E$ définit une énumération des éléments de E (le premier $f(1)$, ..., le n^e $f(n)$).
- ou bien infinis, mais restant équipotents à $\mathbb{N}$. On dit alors que E est infini dénombrable. L'existence de la bijection $f : \mathbb{N} \longrightarrow E$ définit encore une énumération des éléments de E.

Propriétés :

1. *Égalité* Si A, B sont des ensembles finis : $|A| = |B|$ ssi A et B sont équipotents.
2. *Additivité* Si A, B sont des ensembles finis disjoints : $|A \cup B| = |A| + |B|$
3. *Multiplication* Si A, B sont des ensembles finis : $|A \times B| = |A|.|B|$

Lemme 1.3.1 *Soient A, B deux ensembles finis : $|A| \leqslant |B|$ si et seulement si il existe une application injective $f : A \longrightarrow B$.*

Lemme 1.3.2 *$\mathbb{N}$ est le plus petit ensemble infini. Il est stable par addition, multi-*

plication et exponentiation. Il est bien ordonné *: Toute partie non vide admet un plus petit élément.*
L'argument diagonal : *L'ensemble des parties de $\mathbb{N}$ n'est pas dénombrable.*

1.3.7 Dénombrement élémentaire

Une tâche essentielle en informatique est de connaître/calculer le nombre de cas rencontrés dans l'exécution d'un algorithme, d'estimer la taille mémoire de l'implantation d'un type de données, d'estimer le temps d'exécution d'un programme. Le dénombrement est alors un outil essentiel pour réaliser cette phase d'analyse de la complexité des algorithmes.

- $|A \cup B| = |A| + |B| - |A \cap B|$, et si A et B sont disjoints : $|A \cup B| = |A| + |B|$. Se généralise à une partition, soit $\{A_1, \ldots, A_n\}$ une partition de E, alors $|E| = |A_1| + \ldots + |A_n|$
- $|A \times B| = |A|.|B|$,
- $\{A \to B\} = B^A$ ensemble des applications de A vers B. $|B^A| = |B|^{|A|}$,
- $|\mathcal{P}(E)| = 2^{|E|}$.

On suppose dans la suite que $|A| = n$ et $|B| = p$, avec $n \geq p$:

Nombre d'applications injectives de B vers A (Arrangement) $n \times (n-1) \times \ldots \times (n-p+1) = \frac{n!}{(n-p)!} = \mathcal{A}_n^p$.
Parfois appelé *arrangement* de p éléments d'un ensemble à n éléments et noté $\mathcal{A}_n^p$.

Nombre d'applications bijectives de A vers A (Permutation) Ce nombre est $n! = n \times (n-1) \times \cdots \times 2 \times 1$. Une application bijective de A vers A est aussi appelée une *permutation* de A.

Nombre de parties de A ayant pour cardinal p (Coeff. binomiaux, combinaisons) On dénombre l'ensemble des parties de A comportant p éléments par une valeur notée $\binom{n}{p}$:

$$\binom{n}{p} = \frac{n \times (n-1) \times \ldots \times (n-p+1)}{p \times (p-1) \times \ldots \times 1} = \frac{n!}{p!(n-p)!} = \mathcal{C}_n^p$$

Parfois appelé *combinaison* de p éléments d'un ensemble à n éléments et noté $\mathcal{C}_n^p$.

On rappelle que $|\mathcal{P}(A)| = 2^{|A|}$. Mais on peut partitionner $\mathcal{P}(A)$ en les parties

comportant 0 éléments, les parties à 1 élément, ..., les parties à n éléments. D'où la formule :

$$\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{n} = 2^n$$

Principe des tiroirs/cages à pigeons/ ... $\lceil x \rceil \in \mathbb{N}$ désigne la partie entière supérieure de $x \in \mathbb{Q}$.

1. Si $n > p$ alors il n'existe pas d'injection de A vers B. On a déjà vu ce résultat sous une forme positive.

   En d'autre termes : « si on veut ranger n chaussettes dans p tiroirs », et si on veut de plus que chaque « chaussette » soit dans un tiroir différent, en fait on veut construire une application injective de A vers B. Quand $n > p$ c'est impossible.

2. $n = p + 1$ « Si p tiroirs sont occupés par $p + 1$ chaussettes, alors au moins un tiroir contient au moins 2 objets »

   Plus formellement, soit $f : C \longrightarrow T$, avec $|T| = p, |C| = n = p+1$, alors il existe $t \in T$ tel que $|f^{-1}(\{t\})| \geqslant 2$.

3. Plus généralement, $n = p.q + r, 0 \leqslant r < p$. Si p tiroirs sont occupés par n objets, alors il existe au moins un tiroir qui contient $\lceil n/p \rceil$ objets. (Attention quand $r \neq 0$, $\lceil n/p \rceil = q + 1$).

1.4 Relations binaires, relations d'ordre, d'équivalence

1.4.1 Rappels

Définition 1.4.1 *Soient X et Y deux ensembles, une* relation *binaire $\mathcal{R}$ de X vers Y est une partie de $X \times Y$, càd $\mathcal{R} \subseteq X \times Y$. Pour $(x, y) \in \mathcal{R}$, on note $x\mathcal{R}y$, sinon $x \not\mathcal{R} y$.*

1.4.2 Autour des relations binaires

Définition 1.4.2 *On construit d'autres relations binaires à partir de $\mathcal{R}$:*

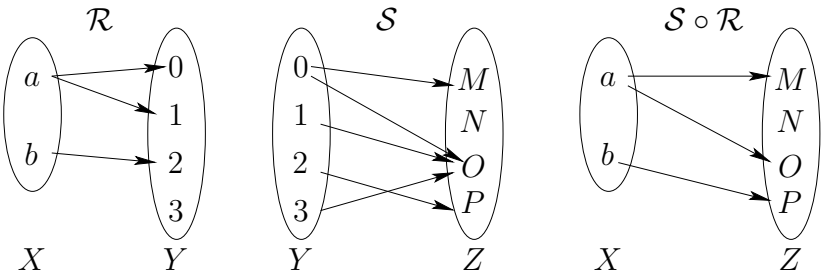
— $\mathcal{R}^{-1}$ *la relation* réciproque *de Y vers X définie, pour $(y,x) \in Y \times X$ par $y\mathcal{R}^{-1}x$ ssi $x\mathcal{R}y$ (parfois notée ${}^t\mathcal{R}$).*

Lemme 1.4.1 *Si $\mathcal{R}$ est une application injective, $\mathcal{R}^{-1}$ est fonctionnelle, si $\mathcal{R}$ est application bijective, $\mathcal{R}^{-1}$ l'est aussi.*

— $\bar{\mathcal{R}}$ *la relation* complémentaire *de X vers Y définie, pour $(x,y) \in X \times Y$ par $x\bar{\mathcal{R}}y$ ssi $x \not\mathcal{R}\, y$ (parfois notée $\neg\mathcal{R}$).*

— *Pour deux relations $\mathcal{R}$ et $\mathcal{S}$ de X dans Y, on construit les relations union, intersection et différence par opération ensembliste sur les parties de $X \times Y$ correspondant à $\mathcal{R}$ et $\mathcal{S}$.*

— *On opère sur les relations binaires par* composition *notée $\circ$. C'est un cas très particulier qui nécessite (de manière simplifiée), que les ensembles d'arrivée et de départ des relations se comportent bien !*
$\mathcal{R}$ relation de X vers Y se compose avec $\mathcal{S}$ de Y vers Z en $\mathcal{S} \circ \mathcal{R}$ de X vers Z. Et $(x,z) \in X \times Z$: $(x,z) \in \mathcal{S} \circ \mathcal{R}$ si $\exists y \in Y$ tel que $(x,y) \in \mathcal{R}$ et $(y,z) \in \mathcal{S}$



Composée de relations binaires

Plus formellement, pour deux relations $\mathcal{R} \subseteq X \times Y$ et $\mathcal{S} \subseteq Y \times Z$, leur composée $\mathcal{S} \circ \mathcal{R} \subseteq X \times Z$ est définie, pour tout $x \in X, z \in Z$,

$$x \ (\mathcal{S} \circ \mathcal{R}) \ z \text{ ssi } \exists y \in Y \text{ tel que } \ x\mathcal{R}y \ \text{ et } \ y\mathcal{S}z.$$

Relations binaires d'un ensemble vers lui-même Soit une *relation* binaire $\mathcal{R}$ de X vers X. On dit alors que $\mathcal{R}$ est définie de X *dans* X (ou de X *sur* X), donc une partie de $X \times X$.

Lorsque X est fini, et $|X|$ est suffisamment petit, on représente graphiquement une relation binaire $\mathcal{R}$ sur X par un GRAPHE, un dessin dont les sommets sont

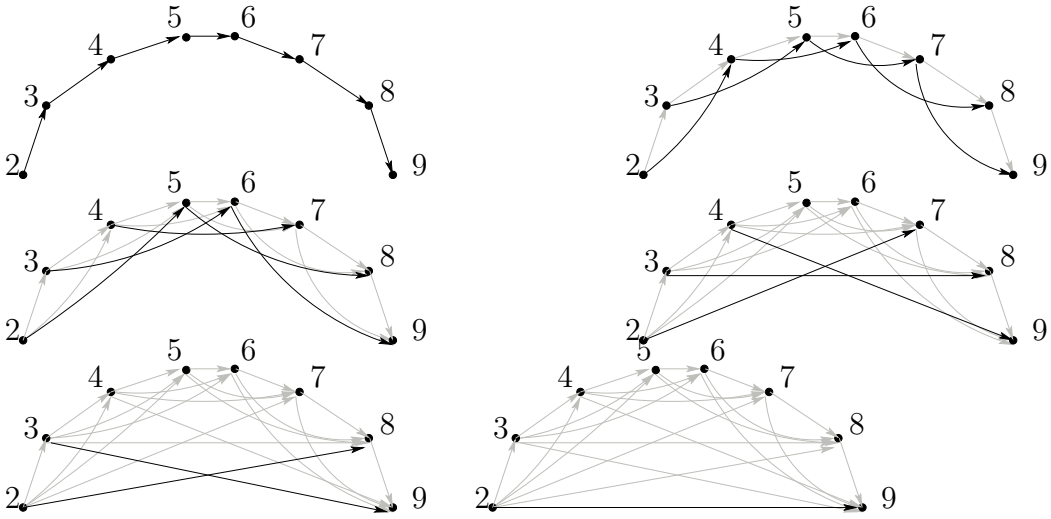


FIGURE 1.2 – La relation $\mathcal{R}$ représentant le successeur sur $[2,9]_{\mathbb{N}}$ et ses itérées. On représente successivement : $\bigcup_{k=1}^{n} \mathcal{R}^k$ pour $n = 1,2,3,4,5,6$. La dernière est la fermeture transitive $\mathcal{R}^+$.

les éléments de X et on dessine un arc (une flèche) entre deux sommets x et y si $(x,y) \in \mathcal{R}$.

Définition 1.4.3 *Soit une* relation *binaire $\mathcal{R}$ de X dans X.*

- $\mathcal{R}$ *est* réflexive *si* $\forall x \in X, x\mathcal{R}x$,
- $\mathcal{R}$ *est* antisymétrique *si* $\forall x,y \in X, (x\mathcal{R}y \text{ et } y\mathcal{R})x \Rightarrow x = y$,
- $\mathcal{R}$ *est* symétrique *si* $\forall x,y \in X, x\mathcal{R}y \Rightarrow y\mathcal{R}x$,
- $\mathcal{R}$ *est* transitive *si* $\forall x,y,z \in X$, $(x\mathcal{R}y \text{ et } y\mathcal{R}z) \Rightarrow x\mathcal{R}z$.

Remarque : Notation :

- une relation symétrique est souvent notée par les symboles $\sim, \simeq, \approx, \equiv \ldots$
- une relation anti-symétrique est souvent notée par les symboles $\geq, \geqslant, \succeq, \sqsupseteq, \ni \ldots$

Relation itérée On définit la relation *itérée* $\mathcal{R}^n = \mathcal{R} \circ \cdots \circ \mathcal{R}$.

Prolongement/restriction de relations binaires On peut *prolonger* une relation :

— en une relation réflexive $\mathcal{R}_R = \mathcal{R} \cup \Delta_X$ en ajoutant la diagonale $\Delta_X = \{(x, x) \in X \times X\}$ (parfois notée I). La relation obtenue est dite *fermeture réflexive* de $\mathcal{R}$
— en une relation symétrique en prenant l'union avec la relation inverse $\mathcal{R}_S = \mathcal{R} \cup \mathcal{R}^{-1}$,
— en une relation transitive en prenant l'union des puissances positives, $\mathcal{R}^+ = \cup_{n>0}\mathcal{R}^n$. C'est la *fermeture transitive* de $\mathcal{R}$.
— en une relation réflexive et transitive : $R^* = R^+ \cup \Delta$. C'est la *fermeture réflexo-transitive* de $\mathcal{R}$.

1.4.3 Relations d'équivalence

Définition 1.4.4 *Une relation $\sim : X \longrightarrow X$ qui est* réflexive, symétrique et transitive *est appelée relation* d'équivalence*. La relation $x \sim y$ se lit «x est équivalent à y».*
Pour un élément $x \in X$ donné, l'ensemble des éléments qui sont en relation avec lui est appelée sa classe d'équivalence*, notée $\bar{x} = \{z \in X \,|\, x \sim z\}$. Un élément $z \in \bar{x}$ est appelé un* représentant *de la classe $\bar{x}$. L'ensemble des classes d'équivalence est appelé* l'ensemble quotient *noté $X/\sim := \{\bar{x} \,|\, x \in X\}$*

Chaque classe peut être vue soit comme un sous-ensemble de X. L'ensemble de toutes les classes forme une partition de X.

1.4.4 Relations d'ordre et ensembles ordonnés

Ranger des objets au mieux est un autre processus mental fondamental, on ordonne une paire d'objets *comparables* en un plus petit et un plus grand, c'est ça construire une relation d'ordre sur un ensemble.

Définition 1.4.5 *Une relation $\leqslant : X \longrightarrow X$ qui est* réflexive, anti-symétrique et transitive *est appelée une relation* d'ordre*. On dit que $(X, \leqslant)$ est* ordonné*.*
Pour une paire d'éléments $(x, y) \in X^2$, on dira que x et y sont comparables *si $x \leqslant y$ ou $y \leqslant x$. Si tous les éléments sont comparables, on dit que l'ordre est* total *sinon il n'est que* partiel*. D'où un ensemble ordonné $(X, \leqslant)$ est* totalement ordonné *si $\leqslant$ est un ordre total, c-à-d si $\forall x, y,\ x \leqslant y$ ou $y \leqslant x$. Il est* partiellement ordonné *sinon, c-à-d si $\exists x$ et y avec $x \neq y$ tels que $x \not\leqslant y$ et $y \not\leqslant x$.*
La relation $x \leqslant y$ se lit « x est plus petit ou égal à y » ou « y est plus grand

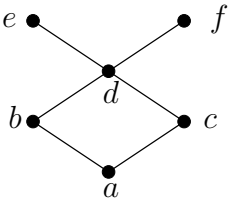
ou égal à x », qu'on note également $y \geqslant x$. Pour deux éléments comparables et différents, $x \leqslant y$ et $x \neq y$, on note $x < y$.
On dit que y couvre *x si $x < y$ et s'il n'existe pas d'éléments entre eux :*

$$x \leqslant z \leqslant y \Rightarrow \begin{cases} x = z \text{ ou} \\ z = y. \end{cases}$$

Définition 1.4.6 *Si X est fini, son* diagramme de Hasse *est le graphe orienté dont les sommets sont les éléments de X et les arêtes (représentées du bas vers le haut) les couples (x, y) où y couvre x.*

Exemple 1.4.1 $X = \{a, b, c, d, e, f\}, \leqslant = \{(d, e), (d, f), (b, d), (c, d), (a, b), (a, c), (a, d), (a, e), (a, f), (b, e), (b, f), (c, e), (c, f), (a, a), (b, b), (c, c), (d, d), (e, e), (f, f)\}$

$d \leq e$ $d \leq f$ $b \leq d$ $c \leq d$ $a \leq b$ $a \leq c$



□

Définition 1.4.7 *Une partie $Y \subseteq X$ d'un ensemble partiellement ordonné hérite de l'ordre partiel. Si l'ordre est total sur Y, on l'appelle une* chaîne*. Un sous-ensemble ou aucune paire n'est comparable est une* anti-chaîne*.*
L'intervalle $[x, y] \subseteq X$ *est l'ensemble des éléments comparables à x et y et compris entre eux :*

$$[x, y] = \{z \in X \mid x \leqslant z \leqslant y\}.$$

Définition 1.4.8 (Éléments particuliers)

1. *Le* minimum *ou* plus petit élément *d'un ensemble Y est un élément qui est plus petit ou égal à tous les autres :*

$$m = \min(Y) \iff \begin{cases} m \in Y \text{ et} \\ \forall y \in Y,\ m \leqslant y. \end{cases}$$

Un ensemble X est bien ordonné *si toute partie non vide admet un plus petit élément. En considérant les paires, on voit que X est alors totalement ordonné.*

2. *Le* maximum *ou* plus grand élément *d'un ensemble Y est un élément qui est plus grand ou égal à tous les autres :*

$$M = \max(Y) \iff \begin{cases} M \in Y \text{ et} \\ \forall y \in Y,\ y \leqslant M. \end{cases}$$

3. *Un élément $m \in Y$ est* minimal *s'il est plus petit ou égal à tous ceux qui lui sont comparables dans Y :*

$$\forall y \in Y,\ y \leqslant m \Rightarrow y = m.$$

4. *De même pour la notion d'élément* maximal.
5. *Un élément $m \in X$ est un* minorant *de Y dans X s'il est plus petit que tous les éléments de Y,*

$$\forall y \in Y,\ m \leqslant y.$$

6. *De même pour la notion de* majorant.
7. *La* borne inférieure *de Y dans X, notée $\inf_X(Y)$, est (s'il existe) le plus grand des minorants de Y :*

$$\forall x \in X,\ (\forall y \in Y, x \leqslant y) \Rightarrow x \leqslant \inf_X(Y).$$

   Si Y admet un minimum, c'est également la borne inférieure.
8. *De même pour la notion de* borne supérieure.

Définition 1.4.9 *Un* treillis *est un ensemble partiellement ordonné ou tout couple $(x, y) \in X^2$ admet une borne supérieure et une borne inférieure*

$$\exists m, M \in X,\ \ m = \inf_X(\{x, y\}) \leqslant x, y \leqslant M = \sup_X(\{x, y\}).$$

Remarque : Si $(X, \leqslant)$ est partiellement ordonné, l'ordre inverse $\geqslant = \leqslant^{-1}$ est également un ordre partiel sur X où les notions de plus petit et plus grand sont inversées.

Chapitre

2 Introduction à l'optimisation Combinatoire

Contents

Résumé

Ce chapitre a pour but de présenter l'optimisation combinatoire en général, et quelques outils existants pour résoudre ou au moins de donner une solution acceptable (terme qui sera défini plus tard).

2.1 Introduction

L'optimisation combinatoire est l'une des plus jeunes et l'une des branches les plus actives des mathématiques discrêtes de ces cinquante dernières années. Pour ce cours, nous allons décrire les idées les plus importantes, démontrer des résultats théoriques et proposer des algorithmes pour l'optimisation combinatoire. L'optimisation combinatoire a ses racines dans la combinatoire, dans la recherche opérationnelle et dans théorie de l'informatique.

2.1.1 Définitions

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

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

Pour éviter des discussions sur les problèmes de précision, nous supposons, sauf exception, que f est à valeurs entières. Par exemple, mesurer des durées de tâches avec des nombres entiers de secondes offre une précision largement suffisante pour un problème d'ordonnancement industriel.
f est la fonction économique ou fonction objectif. La définition concerne la minimisation, mais il suffit de remarquer que maximiser f équivaut à minimiser $-f$..
Parfois, on s'intéresse seulement aux éléments de S vérifiant certaines contraintes, ce sont les solutions réalisables.
Ecrire un algorithme pour un problème combinatoire pour un ensemble S et une application f serait peu utile. On etudie en pratique tout problème d'optimisation combinatoire sous sa forme générale dans laquelle S et f sont des données. Une donnée (S, f) est alors appelé un cas du problème d'optimisation combinatoire (instance).
Un problème d'existence consiste à chercher dans un ensemble fini S, s'il existe un élément s vérifiant une certaine propriété P. Les termes problèmes de décision et problème de reconnaissance sont également utilisés. Les problèmes d'existence peuvent toujours être formulé par un enoncé et une question à réponse oui-non. Nous pouvons les considérer comme des problèmes d'optimisation particulier, en définissant conmme fonction-objectif :

$$f : S \quad \rightarrow \quad \{0,1\} \tag{2.1}$$
$$f(s) = 0 \iff s \text{ vérifie } P \text{ en minimisation} \tag{2.2}$$

De cette façon, un nombre énorme de problèmes peuvent être abordés par l'optimisation combinatoire. Un problème d'existence peut donc être ramené à un problème d'optimisation, mais il existe aussi un problème d'existence associé à tout problème d'optimisation. Il suffit d'ajouter à la donnée de S et f un nombre entier k. La propriété P est alors $f(s) \leq k$. Autrement dit, on ne cherche pas une solution de coût minimal, mais une de coût au plus k.
Noter que si le problème d'existence associé à un problème d'optimisation est difficile, le problème d'optimisation combinatoire l'est alors au moins autant. En effet, si on disposait d'un algorithme efficace pour le problème d'existence, on pourrait l'utiliser pour résoudre le problème d'optimisation combinatoire par dichotomie sur k.

2.2 Formulation et exemples

Un problème d'optimisation combinatoire peut se formuler de la manière suivante : **Données :**

- un ensemble fini d'éléments ou configuration
- une fonction de coût ou poids sur ces éléments

Question : Déterminer un élément de coût minimum ou de poids maximum,...
Voici quelques exemples :

1. *Plus court chemins :*
   Données :
   - un graphe orienté $G = (X, E)$
   - une valuation $v : E \rightarrow \mathbb{R}$
   - s, t deux sommets distincts de X.

   Question : Trouver le plus court chemin de s à t ?

   Solution : Algorithme de Dijkstra, Bellman-Ford. Algorithmes polynomiaux.
2. *Arbre de poids minimum :*
   Données :
   - un graphe orienté $G = (X, E)$
   - une valuation $v : E \rightarrow \mathbb{R}^+$

Question : Trouver un arbre couvrant de poids minimum ?

Solution : Algorithme de Kruskal, Prim. Algorithmes polynomiaux.

3. *Voyageur de commerce :*

   Données :
   — un ensemble de m villes X,
   — un ensemble de routes entre les villes E.
   — une fonction de coût $v : E \to \mathbb{R}$ où $v(x, y)$ est le coût de déplacement de x à y.

   Question : Trouver une tournée de coût (chemin hamiltonien de coût minimum).

   Solution : $\mathcal{NP}$-complet

4. *Système de rues à sens unique.* modélisation du trafic en ville. Problème d'augmentation de trafic (flot).

Remarque : Pourquoi le mot combinatoire ?

— Traditionnellement le mot combinatoire désigne le dénombrement de combinaisons d'éléments en relation avec le calcul des probabilités.
— Ici ce mot désigne l'existence d'une structure finie permettant de générer une quantité astronomique de situations à comparer. On parle d'explosion combinatoire.

Exemple 2.2.1 Combien y-a-t'il d'arbres dans un graphe complet à n sommets. il y a $C_{n^2-n}^{n-1}$. Pour cela il suffirait de prendre un algorithme qui génère tous les arbres recouvrants et choisir celui de poids minimum. Pourtant cette approche n'est pas opérationnelle.

□

2.3 Outils de l'optimisation combinatoire

Voici les grands outils utilisés pour la résolution des problèmes en optimisation combinatoire :

1. **Graphe**
   — La modélisation par les graphes est naturelle dans certains problèmes (plus court chemins,...)
   — Avantage : le concept de graphe permet de considérer des relations sur des structures.
2. **Programmation linéaire en nombres entiers**

Données :
— une matrice A de type $n \times n$
— un vecteur b de taille m
— un vecteur c de taille n

Question : On appelle programmation linéaire en nombres entiers le problème d'optimisation suivant : $(P) \begin{cases} Ax & \leq & b,\ x_j \in \mathbb{N} \\ cx & = & Z(max) \end{cases}$

Si $x_j \in \{0, 1\}$ on dit qu'on a un programme linéaire en $0, 1$. Par exemple le problème du sac à dos peut-être représenté par un programme linéaire en nombres entiers.

3. **Programmation linéaire** $(P') \begin{cases} Ax & \leq & b,\ x_j \in \mathbb{R} \\ cx & = & Z(max) \end{cases}$

— Une solution entière de (P') conduit à une solution de (P).
— Le fait d'arrondir une solution de (P') ne donne pas toujours une solution à (P).
— L'algorithme du simplexe de Dantzig (1947) est efficace dans la pratique.

Le principe est de parcourir un ensemble de sommets formant l'ensemble des solutions. Les sommets sont les points extrémaux issus d'intersection de polyèdres.

Avantages : On peut s'appuyer sur la programmation linéaire qui possède des algorithmes de résolutions efficaces. Le problème devient la transformation de solution réelle en solution discrète.

Chapitre

3 Rappels en théorie des graphes

Contents

Résumé

Ce chapitre est un bref rappel en théorie des graphes.

3.1 Introduction

Les graphes représentent un instrument puissant pour modéliser de nombreux problèmes combinatoires, qui seraient sans cela difficilement abordables par des techniques classiques comme l'analyse mathématique.
Ils sont des structures combinatoires permettant de représenter de nombreuses situations rencontrées dans des applications réelles faisant intervenir des mathématiques discrètes et nécessitant une solution informatique.
Utilisés comme une structure de données en informatique, ils permettent de décrire efficacement la structure d'un ensemble complexe et d'exprimer les relations et les dépendances entre ses éléments.
Les applications qui font appel à la théorie des graphes sont nombreuses. On peut citer à titre d'exemple, les circuits électriques, les réseaux de transport (ferrés, routiers, aériens), les réseaux d'ordinateurs, les diagrammes hiérarchiques en sociologie, les diagrammes de successions de tâches en gestion de projet, etc.

3.2 Point de vue formel

Considérons le problème suivant :
On a trois villes, notées a, b, c, que l'on veut relier par des conduites à une usine de production d'eau (notée D), à une usine de production de gaz (notée E) et à une usine de production d'électricité (notée F).
Peut-on placer sur un plan les trois villes, les trois usines et les trois conduites, de sorte que deux conduites ne se croisent pas en dehors de leurs extremités ?
Pour répondre à la question on désigne les trois villes par trois points a, b, c, et les trois usines par trois autres points D, E, F. Les conduits sont représentées par des liens entre les points a, b, c et D, E, F. Ceci nous amène à construire un graphe avec six points (appelés sommets) et neuf liens (appelés arêtes). On peut facilement constater que huit conduites peuvent être placées mais la neuvième coupera toujours une des huit premières (voir la figure 3.1).
L'arête entre les sommets c et E coupe l'arête entre les sommets a et F.

Définition 3.2.1 *Un graphe $G = (X, A)$ est donné par un ensemble X de sommets et par un sous-ensemble A du produit cartésien $X \times X$ appelé ensemble des arcs de G. Si les ensembles X et A sont finis le graphe sera appelé graphe fini.*

Définition 3.2.2 *Un arc $a = (x, y)$ a pour* origine *le sommet x et pour* extrémité

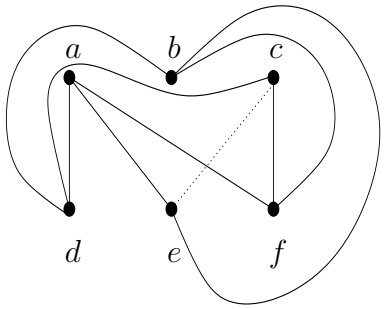


FIGURE 3.1 –

le sommet y. On note $Or(a) = x$ et $Ext(a) = y$. Le sommet y est un successeur *de x, et x un* prédécesseur *de y.*

Dans les définitions ci-dessus les arcs doivent être parcourus dans un sens déterminé (de x vers y, mais pas de y vers x). L'orientation donc est significative et le graphe G est appelé un graphe *orienté*. Si les arcs peuvent être parcourus dans les deux sens, le graphe G est appelé non orienté et les arcs sont appelés *arêtes*. On note $[x, y]$ une arête entre les sommets x et y. Chaque sommet x ou y de l'arête $[x, y]$ peut être vu comme origine ou extrémité de l'arête.
Notons qu'il n'y a pas deux théories des graphes, l'une pour les graphes orientés et l'autre pour les graphes non orientés. Il y a seulement des concepts et des applications dans lesquels l'orientation est importante et d'autres où elle ne l'est pas. Les structures de données sont les mêmes dans l'ordinateur. Les algorithmes de parcours développés pour les graphes orientés s'appliquent en particulier aux graphes non orientés. Il suffit de construire à partir d'un graphe non orienté G un nouveau graphe G' orienté avec les mêmes sommets et remplacer une arête $[x, y]$ par deux arcs (x, y) et (y, x).

Remarque 3.1 Un graphe orienté est aussi appelé souvent "*digraph*" et un graphe non orienté est appelé tout simplement graphe.

Définition 3.2.3 *Un p-*graphe *est un graphe orienté où un arc peut apparaître au plus p fois entre deux sommets.*

Définition 3.2.4 *Un* multigraphe *est un graphe non orienté où une arête peut apparaître plusieurs fois entre deux sommets.*

Définition 3.2.5 *Un graphe* simple *est un graphe non orienté sans boucles (c'est-à-dire $[x, x] \notin E$).*

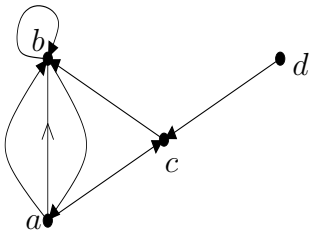


FIGURE 3.2 – un 2-graphe

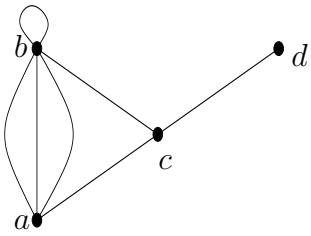


FIGURE 3.3 – un multigraphe

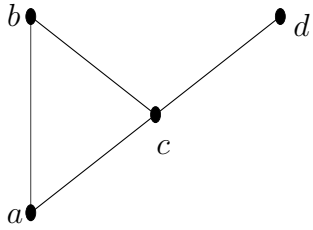


FIGURE 3.4 – un graphe simple

Par la suite on va considérer uniquement les 1-graphes pour les graphes orientés, et les graphes simples pour les graphes non orientés. On notera un graphe simple par $G = (V, E)$, où V est l'ensemble des sommets et E l'ensemble des arêtes de G. Pour un graphe orienté on utilisera la notation précédente, c'est-à-dire $G = (X, A)$, où X est l'ensemble des sommets et A l'ensemble des arcs de G.

Exemple 3.2.1 La figure 3.2 représente un 2-graphe $G = (X, A)$ avec $X = \{a, b, c, d\}$ et $A = \{(a, b), (b, a), (a, b), (a, c), (b, b), (c, b), (d, c)\}$. □

Exemple 3.2.2 La figure 3.3 représente un multigrahe $G = (V, E)$ avec $V = \{a, b, c, d\}$ et $E = \{[a, b], [b, a], [a, b], [a, c], [b, b], [c, b], [d, c]\}$. □

Exemple 3.2.3 La figure 3.4 représente un graphe simple $G = (V, E)$ avec $V = \{a, b, c, d\}$ et $E = \{[a, b], [a, c], [c, b], [d, c]\}$. □

Exemple 3.2.4 Représentation des rues d'une ville par un graphe $G = (X, A)$ où X est l'ensemble des carrefours et $A = \{(x, y) \,|\, x \in X, y \in X,$ et il existe une rue de x vers $y\}$. Pour une rue à double sens on a $(x, y) \in A$ et $(y, x) \in A$. □

Exemple 3.2.5 Considérons l'ensemble $V = \{1, 2, \ldots, n\}$. La relation "l'entier i divise l'entier j" peut être représentée à l'aide d'un graphe de la manière suivante. On définit le graphe $G = (X, A)$ avec $X = \{2, 3, \ldots, n\}$ et $A = \{(x, y) \,|\, x \in A, y \in A$ et $x|y\}$.
Avec $n = 12$ on obtient le graphe 3.5. □

Exemple 3.2.6 Le graphe de De Bruijn $G(X, A)$ est contruit de la manière suivante. L'ensemble $X = \{$suites binaires de longueur$k\}$ et $A = \{(x, y) \,|\, x = ax_{k-1}x_{k-2} \ldots x_1,\ y = x_{k-1}x_{k-2} \ldots x_1 b,\ a, b \in \{0, 1\}\}$.
Le graphe 3.6 représente le grape de De Bruijn avec $k = 3$. □

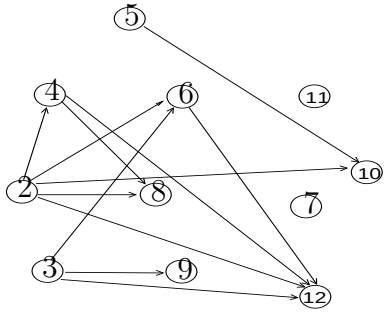


FIGURE 3.5 – Le graphe des diviseurs de 12

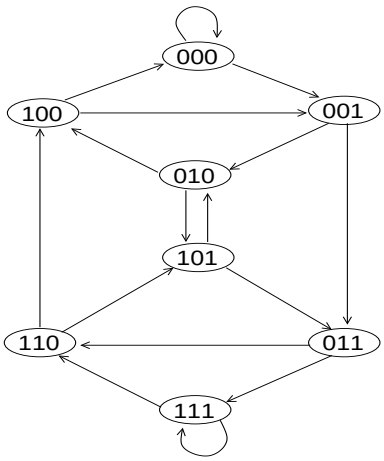


FIGURE 3.6 – Le graphe de De Bruijn avec $k = 3$.

Définition 3.2.6 *Le* degré *d'un sommet x dans un graphe $G = (X, A)$ est le nombre d'arcs incidents*[1] *à x et il est noté par $d_G(x)$, ou tout simplement $d(x)$.*

Donc $d(x)$ est la somme du nombre d'arcs sortants et du nombre d'arcs entrants.

Définition 3.2.7 *Le nombre d'arcs sortants est noté $d^+(x)$ et est appelé* demi degré extérieur. *Le nombre d'arcs entrants est noté $d^-(x)$ et est appelé* demi degré intérieur.

Pour un graphe non orienté (simple) le degré $d(x)$ est le nombre de voisins de x, c'est-à-dire le nombre de sommets adjacents à x.

1. ayant pour extrémité x

Définition 3.2.8 *Un sommet* isolé *est un sommet de degré 0.*

Exemple 3.2.7 Dans le graphe de l'exemple 3.2 on a : $d(6) = 3$, $d^+(6) = 1$, $d^-(6) = 2$, $d(7) = 0$, $d(11) = 0$, $d(2) = 5$, $d^+(2) = 5$ et $d^-(2) = 0$. □

3.3 Graphes particuliers

Définition 3.3.1 *Un graphe est dit* complet *si toute paire de sommets est liée par un arc ou une arête. On note K_n le graphe simple complet à n sommets (on dit aussi* clique *à n sommets).*

Exemple 3.3.1

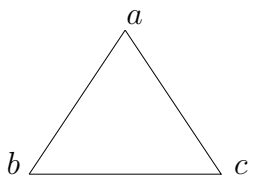


FIGURE 3.7 – K_3

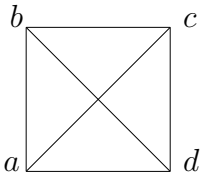


FIGURE 3.8 – K_4

□

Définition 3.3.2 *Un graphe $G = (X, A)$ est dit* biparti *si on peut diviser l'ensemble de ses sommets en deux sous-ensembles X_1 et X_2 tels que $X_1 \mathcal{X}_{\in} = \emptyset$, $X_1 \cup X_2 = X$ et tout arc $a \in A$ a une extrémité dans X_1 et l'autre dans X_2. On note $G = (X_1, X_2, A)$.*

Exemple 3.3.2

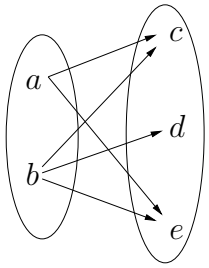


FIGURE 3.9 – *Un graphe biparti*

□

Définition 3.3.3 *Un graphe $G = (V_1, V_2, E)$ est dit* biparti complet *si $\forall v \in V_1$ on a $d(v) = |V_2|$. Si $|V_1| = a$ et $|V_2| = b$ on note ce graphe $K_{a,b}$.*

Exemple 3.3.3

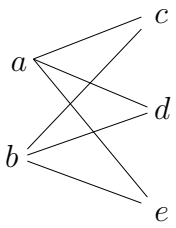


FIGURE 3.10 – *Le graphe biparti complet $K_{2,3}$*

□

Définition 3.3.4 *Un graphe $G = (V, E)$ est dit k* régulier *si $\forall v \in V$ on a $d(v) = k$, avec $k \in \mathbb{N}$ une constante. Autrement dit, tous les sommets ont le même degré. On a évidemment $k \leq n - 1$, avec $n = |V|$.*

Définition 3.3.5 *Le graphe* complémentaire *d'un graphe $G = (X, A)$ est le graphe $\overline{G} = (X, X^2 - A)$.*

Définition 3.3.6 *Un graphe $G = (V, E)$ est dit* planaire *si on peut le dessiner dans le plan sans croisement des arêtes.*

Formuler le problème des trois villes rencontrées précédemment dans le langage de la théorie des graphes.

3.4 Partie d'un graphe

Définition 3.4.1 *Soit un graphe $G = (X, A)$ et X' un sous ensemble de X. Le* sous-graphe de G engendré par X' est le graphe G' tel que $G' = (X', (X' \times X') \cap A)$.

Définition 3.4.2 *Soit un graphe $G = (X, A)$ et A' un sous ensemble de A. Le* graphe partiel engendré par A' est le graphe G' tel que $G' = (X, A')$.

Exemple 3.4.1

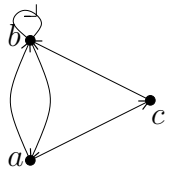


FIGURE 3.11 – $G(X, A)$

FIGURE 3.12 – *sous graphe de G engendré par* $X' = \{x_1, x_2\}$.

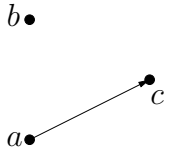


FIGURE 3.13 – *graphe partiel engendré par* $A' = \{(x_1, x_3)\}$

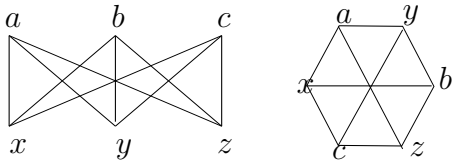
□

3.5 Isomorphisme de graphes

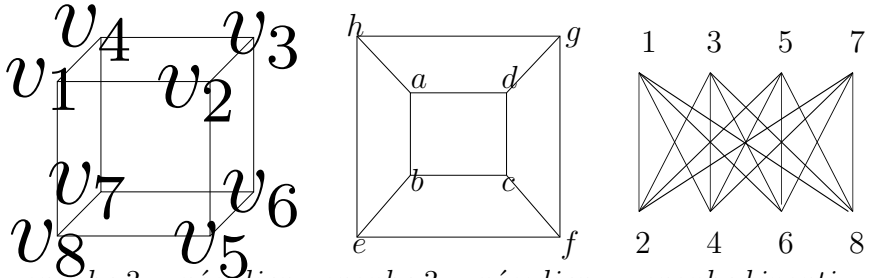
Un graphe est déterminé par la connaissance de ses sommets et de ses arêtes. Donc, pour décrire un graphe on doit donner la liste de ses sommets et de ses arêtes, peut importe dans quel ordre.
Dans une représentation d'un graphe par un diagramme, il est possible que différents diagrammes représentent le même objet mathématique.

Définition 3.5.1 *Deux graphes* $G_1 = (V_1, E_1)$ *et* $G_2 = (V_2, E_2)$ *sont dits* isomorphes*, s'il existe une bijection* $f : V_1 \to V_2$ *telle que* $[f(x_1), f(x_2)] \in E_2$ *si et seulement si* $[x_1, x_2] \in E_1$.

Les deux graphes ci-dessous semblent différents, mais ils donnent la même information. Ils sont identiques et donc isomorphes.



Les trois graphes ci-dessous sont isomorphes.



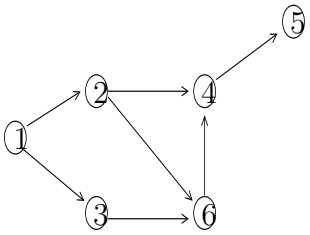
graphe $3-$*régulier* *graphe* $3-$*régulier* *graphe biparti*

3.6 Représentation des graphes

Une structure de données simple pour représenter un graphe est la *matrice d'adjacence* M. Pour obtenir M, on numérote les sommets du graphe de façon quelconque. Soit $G = (X, A)$ un graphe orienté et $X = \{x_1, x_2, \ldots x_n\}$ l'ensemble de ses sommets. La matrice d'adjacence $M = (M_{ij})$ est une matrice carrée $n \times n$, dont les coefficients sont 0 ou 1, telle que :

$$M_{ij} = \begin{cases} 1 \text{ si } (x_i, x_j) \in A \\ 0 \text{ si } (x_i, x_j) \notin A \end{cases}$$

Exemple 3.6.1



□

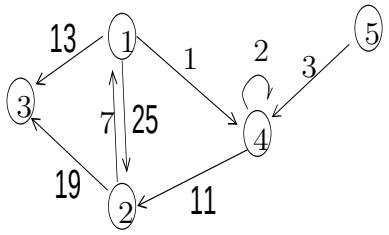
$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Remarque 3.2 Dans le cas des graphes non orientés la matrice M est symétrique. Pour les graphes valués (graphes avec des poids sur les arcs ou arêtes), la matrice d'adjacence M est définie comme suit :

$$M_{ij} = \begin{cases} c_{ij} \text{ si } (x_i, x_j) \in A \\ -1 \text{ si } (x_i, x_j) \notin A \end{cases},$$

avec c_{ij} le poids sur l'arc (x_i, x_j) ou l'arête $[x_i, x_j]$. Il est sous-entendu que la valeur -1 ne fait pas partie des valeurs que peuvent prendre les poids c_{ij}, sinon il faut choisir une autre valeur.

Exemple 3.6.2



□

$$M = \begin{pmatrix} -1 & 25 & 13 & 1 & 0 \\ 7 & -1 & 19 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 11 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 & -1 \end{pmatrix}$$

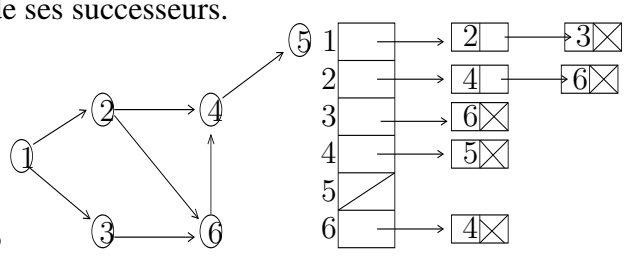
Remarque 3.3 Cette représentation convient pour des graphes pour lesquels l'ensemble des sommets n'évolue pas. Elle a l'avantage de la simplicité, et permet en temps constant de tester l'existence d'un arc (x_i, x_j) et de trouver facilement les prédécesseurs (balayage de la colonne) d'un sommet ou les successeurs (balayage de la ligne) en temps $\mathcal{O}(n)$ où n est l'ordre du graphe. Une consultation complète de la matrice requiert un temps d'ordre n^2 et l'implantation exige $\theta(n^2)$ espace mémoire. Pour des graphes peu denses, c'est-à-dire avec un nombre d'arcs (ou d'arêtes) faible ($m \ll n^2$, avec $m = |A|$) ceci représente un inconvénient majeur. C'est pour cela qu'on préferera dans ces cas une autre représentation : celle des listes d'adjacence.

Liste d'adjacence

Cette représentation consiste à représenter d'abord les sommets (par exemple dans un tableau) et à associer ensuite à chaque sommet la liste de ses successeurs dans un ordre quelconque.

Dans le cas ou l'ensemble des sommets n'évolue pas, ces listes sont accessibles à partir du tableau contenant pour chaque sommet un pointeur vers le début de la liste associée de ses successeurs.

Exemple 3.6.3

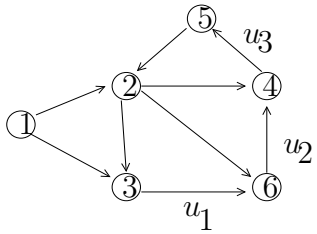


□

Remarque 3.4 Avec cette représentation, l'espace mémoire utilisé pour un graphe avec n sommets et m arcs est en $\theta(n+m)$. Dans le cas d'un graphe non orienté il est en $\theta(n+2m)$. Un algorithme qui traite tous les arcs d'un graphe est en $\mathcal{O}(m)$. Malheureusement, tester l'existence d'un arc (ou d'une arête), nécessite un temps en $\mathcal{O}(m)$. Ajouter un arc ou une arête (avec un test de non répétition) nécessite un temps en $\mathcal{O}(m)$.

3.7 Chemins, chaînes, circuits, cycles

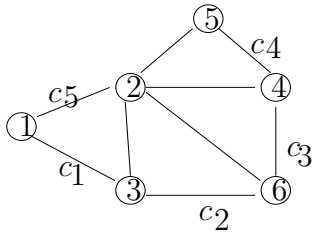
Définition 3.7.1 *Soit un graphe $G=(X,A)$. Un* chemin μ *dans G est une suite finie d'arcs $a_1, a_2, \ldots, a_p$ telle que $\forall i,\ 1 \leq i < p$ or $(a_{i+1}) = ext(a_i)$. L'*origine *de μ, notée $or(\mu)$, est l'origine du premier arc ($or(a_1)$) et l'*extrémité *de μ, notée $ext(\mu)$, est celle du dernier arc ($ext(a_p)$). La* longueur *du chemin μ est égale au nombre d'arcs p. Un chemin μ tel que $or(\mu) = ext(\mu)$ est appelé un* circuit.



Exemple 3.7.1 □

$\mu = u_1u_2u_3$
$or(\mu) = 3$, $ext(\mu) = 5$
$\mu(5,2)(2,3)$ est un circuit

Définition 3.7.2 *Une* chaîne c *dans un graphe non orienté est une suite finie d'arêtes $c_1, c_2, \ldots, c_p$ telle que $\forall i,\ 1 \leq i < p$ $c_i = [v_i, v_{i+1}]$ et $c_{i+1} = [v_{i+1}, v_{i+2}]$. Une chaîne c telle que $v_{p+1} = v_1$ est appelée un* cycle.



Exemple 3.7.2 □

$c = c_1c_2c_3c_4$ est une chaîne
$c = c_2c_3c_4[5,2][2,3]$ est un cycle

3.8 Existence d'un chemin

Soit un graphe $G = (X, A)$. Pour tester l'existence d'un chemin entre deux sommets x_i et x_j de G, on peut utiliser la matrice d'adjacence M de G.
En effet, si on calcule la p-ième puissance de M, le coefficient M_{ij}^p est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et l'extrémité le sommet x_j.
Donc puisque la longeur d'un chemin dans un graphe G d'ordre n est au plus $n-1$, on doit calculer M^{n-1} pour vérifier l'existence d'un chemin entre deux sommets quelconques. Si les deux sommets x_i et x_j sont fixés, on peut ne calculer qu'une ligne de la matrice M^{n-1}, ce qui diminue notablement la complexité.
Nous avons le théorème suivant :

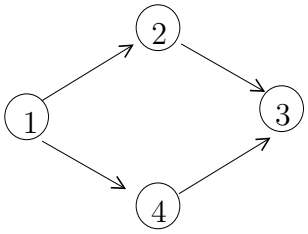
Théorème 3.8.1 *Soit M^p la puissance $p-ime$ de la matrice M, le coéfficient $M_{i,j}^p$ est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j.*

Preuve
On effectue une récurrence sur p. Pour $p = 1$ le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Le calcul de M^p, pour $p > 1$ donne :

$$M_{i,j}^p = \sum_{k=1}^{k=n} M_{i,k}^{p-1} \times M_{k,j}$$

Or tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur $p-1$ entre x_i et un certain x_k suivi d'un arc reliant x_k et x_j. Le résultat découle alors de l' hypothèse de récurrence suivant laquelle $M_{i,k}^{p-1}$ est le nombre de chemins de longueur $p-1$ joignant x_i à x_k. □

Exemple 3.8.1



$$M = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$M^2 = \begin{pmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$M_{13} = 0$ "pas d'arc entre les sommets 1 et 3"
$M_{43} = 1$ "1 chemin de longueur 1 entre 4 et 3"
$M^2_{13} = 2$ "2 chemins de longueur 2 entre 1 et 3"
$M^2_{43} = 0$ "aucun chemin de longueur 2 entre 4 et 3" □

Chapitre 4

Complexité

Contents

Résumé

Nous allons voir la notion de complexité d'un algorithmique et ces conséquences sur l'implémentation.

4.1 Introduction

Les problèmes qui nous intéressent ici sont ceux dont nous connaissons une représentation mathématiques. Ces modèles mathématiques sont facilement implémentés en informatique et peuvent ainsi « résolus » par des algorithmes. Un modèle mathématique pour un problème décrit formellement les propriétés qu'une solution pour ce problème doit vérifier. La `résolution` d'un problème, est le calcul d'une telle solution que l'on appellera `solution réalisable` ; une telle solution n'est pas nécessairement optimale, c'est à dire la meilleure parmi les solutions vérifiant les propriétés que le modèle mathématique pour le problème impose. Pour aborder formellement la question de la résolution informatique d'un problème, nous avons besoin d'un certain nombre d'éléments préliminaires. Ils sont présentés dans la suite.
Un problème peut-être résolu par plusieurs algorithmes, encore plus de programmes (chaque algorithme pouvant être implémenté dans plusieurs langages et de différentes façons).
Comment comparer les diverses solutions ? Quels sont les critères importants pour comparer et donc choisir entre plusieurs algorithmes ou plusieurs programmes ? Diverses propriétés ou paramètres physiques peuvent caractériser l'efficacité d'un programme (supposé juste) : place mémoire nécessaire, durée d'exécution, simplicité du code, . . .
Dans ce cours (faute de temps et pour des raisons de simplification) nous allons nous limiter, aux deux premiers paramètres et nous intéresser plus aux aspects algorithmiques que programmation (dans la mesure du possible car il est évident que la réflexion autour de la mise en œuvre d'un algorithme à un impact parfois déterminant sur son efficacité).
Comment mesurer la place mémoire ? : le nombre de bits utilisés. Comment mesurer la durée d'exécution ? : le nombre d'unités de temps pour la durée d'exécution. Mais alors le temps d'exécution dépend de la donnée, du compilateur, de l'ordinateur, du langage utilisé . . . Bref la mesure devient difficilement utilisable. Il faut simplifier, modéliser. En effet, des énoncés du type : L'algorithme A implémenté dans le langage P sur l'ordinateur O, et exécuté sur la donnée D utilise k secondes et j bits de mémoire, sont d'une portée très limitée.
Que se passe-t-il si l'on exécute sur la donnée D' ? puis si l'on change d'ordinateur ? Nous allons donc chercher des informations plus générales : l'algorithme A est toujours meilleur que l'algorithme B dès que D est grand.
En fait, à chaque problème, on peut en général associer une ou des opérations élémentaires : c'e s t - à-dire une (ou des) opération caractéristique ou élémen-

taire que l'on exécutera au moins autant de fois que les autres. Par exemple si on recherche un élément dans un tableau le nombre de comparaisons entre éléments sera l'opération élémentaire, si on multiplie deux matrices les opérations élémentaires seront les additions et les multiplications d'entiers, si l'on trie des éléments les opérations élémentaires seront les comparaisons entre éléments et les déplacements d'éléments etc ... Axiome : le temps d'exécution de l'algorithme est proportionnel au nombre d'exécutions de l'opération fondamentale (ou élémentaire). On se contentera donc à chaque fois après l'avoir déterminé, de compter le nombre de fois où cette opération est effectuée.

4.1.1 Importance de formaliser la complexité

Les algorithmes prévus pour résoudre le même problème différent souvent énormément par leur efficacité. Ces différences peuvent être beaucoup plus significatives que la différence deux ordinateurs : soit un algorithme de tri par insertion sur un ordinateur A, et un algorihtme de tri par fusion sur un ordinateur B. Les deux doivent trier un tableau d'un million de nombres. Supposons que l'ordinateur A exécute 100 millions d'instructions à la seconde, tandis que l'ordinateur B n'en exécute que un million. Pour accroitre la différence, supposons que le tri par insertion pour l'ordinateur A est écrit en langage machine par le programmeur le plus habile du monde, et que le code résultant ait besoin que de $2n^2$ instructions de l'ordinateur A pour trier n nombres. De son côté, le tri par fusion est programmé sur un oridnateur B par un étudiant à l'aide d'un langage de haut niveau et d'un compilateur inefficace, ce qui produit un code consommant $50n \log n$ instructions de l'ordinateur B.
Ainsi nous obtenons pour le temps de calculs pour l'ordinateur A :

$$\frac{2.(10^6)^2 \text{ instructions}}{10^8 \text{ instructions/seconde}} = 20000\text{secondes} \equiv 5,56 \text{ heures}$$

et pour l'ordinateur B :

$$\frac{50.(10^6) \log 10^6 \text{ instructions}}{10^6 \text{ instructions/seconde}} = 1000\text{secondes} \equiv 16,67 \text{ minutes}$$

Cet exemple montre que les algorithmes, comme les matériels informatiques s'apparentent à la technologie. L'efficacité totale d'un système dépend du choix du bon algorithme autant que du choix du matériel le plus rapide. De même que des avancées rapides ont lieu dans les autres technologies informatiques, elles ont lieu également en algorithmique.

4.1.2 Problème - instance

Dans la théorie algorithmique, un problème est une question générale pour laquelle on veut obtenir une réponse. Cette question possède généralement des paramètres ou des variables dont la valeur reste à fixer. Un problème est spécifié en donnant la liste de ces paramètres ainsi que les propriétés que doit vérifier la réponse. Une instance d'un problème est obtenue en explicitant la valeur de chacun des paramètres du problème instancié.

Algorithme 4.1 La recherche d'un élément dans un tableau. Nous supposerons que $x \in t$

```
i = 1;
while t[i] ≠ x do
   i = i + 1;
end while
retourner i;
```

On remarque que le nombre de comparaisons dépend de la taille des données et de leur organisation (l'élément recherché est-il absent ou présent dans l'une des cases et s'il est présent dans laquelle ?). D'où la nécessité d'introduire la notion de fonctions qui prennent en paramètre la taille des données et donnent comme valeur le nombre d'opérations effectuées : soit le nombre moyen ou le nombre maximum. Le calcul du nombre moyen de comparaisons est souvent un problème très difficile (Quelle est la probabilité d'avoir la donnée d ?). Dans la suite nous limiterons ce problème en faisant l'hypothèse que toutes les données de même taille ont la même probabilité (distribution uniforme). Ainsi à chaque algorithme A, on essayera d'associer une fonction $f : A \to \mathbb{N}$ dans $\mathbb{N}$ qui associe le nombre d'exécutions de l'opération élémentaire si l'on exécute A sur une donnée de taille n (l'unité de taille pourraêtre le bit, l'entier, le flottant, ...). Sur l'exemple, il est facile de voir que le nombre de comparaisons d'éléments est mieux 1 (l'élément x à la première case), au pire n (l'élément x à la dernière case), en moyenne $(n+1)/2$ (i comparaisons si l'élément x est dans la case i, avec une probabilité $1/n$). Pour traiter le cas où $x \notin t$: on ajoute le test $i < n$. Exploitation de la fonction : En fait les valeurs de ces fonctions ne sont évidemment pas égales à la durée d'exécution du programme (la mesure qui intéresse l'utilisateur) par contre elles ont de bonnes chances de représenter une valeur proportionnelle à la durée d'exécution. C'est- à-dire ici proportionnelle à n. Ainsi si pour $n = 100$, l'algorithme s'exécute en $1ms$, il mettra environ 10 ms pour $n = 1000$.

4.1.3 Algorithme et complexité

Un algorithme est une suite d'opérations élémentaires (affectations de variables, tests, branchements, ...) qui, quand on lui fournit une instance d'un problème en entrée, s'arrête après exécution de la dernière opération en nous renvoyant la solution. Le déroulement et le résultat de l'algorithme dépendent de la valeur de tous les paramètres de l'instance du problème. A chaque type de problème correspond un type de solution ; par exemple :

- une liste de nombre, un graphe, un chemin dans un graphe, ... ;
- un ensemble de variables, un tableau de nombres, ...
- oui$\backslash non$;
- une valeur ;
- l'affirmation de l'absence de solution.

Le temps d'exécution d'un algorithme est compté en nombre d'instructions nécessaires à son déroulement. L'utilisation du nombre d'instructions comme unité de temps est justifiée par le fait qu'un même programme utilisera le même nombre d'instruction sur deux machines différentes mais prendra plus ou moins de temps selon leurs rapidités respectives. Il est généralement considéré qu'une instruction correspond à multiplication, un marquage, ... En informatique, toute opération plus compliquée peut plus ou moins se ramener à l'une des précédentes.
Ce qu'on appelle la complexité d'un algorithme correspond à peu près à une indication du temps qu'il prendra pour résoudre un problème d'une taille donnée. C'est en réalité une fonction qui associe, à la taille d'une instance d'un problème donné, une approximation du nombre d'instruction nécessaires à sa résolution. nous supposons, pour des raisons de simplicité, que la complexité d'un algorithme A s'écrit de la façon suivante : $O(f(n))$ et se lit « la complexité de l'algorithme A est en ordre de $f(n)$ » où f est une fonction et n la taille de l'instance du problème.

Définition 4.1.1 *Nous définissons par O et θ de la manière suivante :*
Soient f et g, deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$.

- $f = O(g)$ *si* $\exists c \in \mathbb{R}^{*,+}, \exists \mathrm{n}_0 \in \mathbb{N}$ *tel que* $\forall n > n_0,\ cg(n) \geq f(n)$. *Dans ce cas f croit moins vite que g.*
- $f = \theta(g)$ *si* $\exists c, d \in \mathbb{R}^{*,+}, \exists \mathrm{n}_0 \in \mathbb{N}$ *tel que* $\forall n > n_0,\ dg(n) \geq f(n) \geq cg(n)$. *Dans ce cas f et g croissent à la même vitesse.*

Si on reprend l'exemple de la recherche, le nombre de comparaisons d'éléments en moyenne $(n+1)/2$ donc en $\theta(n)$. On peut donc en déduire que la durée de l'exécution de l'algorithme sera proportionnelle à n. Ainsi si en moyenne on met

1ms pour chercher un élément dans un tableau de taille 100, on mettra environ 1s pour chercher un élément dans un tableau de taille 100000. Exercice : $f(n) = 2n^2 + 3n$ et $g(n) = n^2 + 7n$. On a $f = \theta(g)$ en prenant $c = 1, d = 3, n_0 = 5$. Un exemple plus complet :

Algorithme 4.2 la recherche du plus grand élément dans un tableau (retourne l'élément le plus grand du tableau)

```
m = t[1];
for i de 2 à n do
  if t[i] > m then
    m = t[i];
  end if
end for
retourner m;
```

L'algorithme nécessite $(n-1)$ comparaisons d'éléments : une comparaison à chaque passage dans la boucle. Ce résultat est a priori optimal car il y a $(n-1)$ éléments qui ne sont pas optimaux et pour le savoir il faut comparer chacun avec un élément plus grand. Si l'on désire calculer le nombre d'affectations, le problème est plus délicat car le résultat ne dépend pas que du nombre d'éléments. Pour l'analyse on suppose que les éléments sont tous disjoints et que toutes les positions pour les éléments dans le tableau sont équiprobables (ainsi si le tableau contient les éléments $2, 3$ et 5 alors il y a 6 tableaux possibles :

1. $2|3|5$ avec un nombre d'affectation à 3,
2. $2|5|3$ avec un nombre d'affectation à 2,
3. $3|2|5$ avec un nombre d'affectation à 2,
4. $3|5|2$ avec un nombre d'affectation à 2,
5. $5|2|3$ avec un nombre d'affectation à 1,
6. $5|3|2$ avec un nombre d'affectation à 1,

Seul l'ordre des éléments est important ainsi on peut limiter l'analyse au cas où le tableau ne contient que les entiers de 1 à n (on peut donc se limiter dans l'analyse aux permutations des entiers de 1 à n). Et chacune apparaît avec la même probabilité : $1/6$. Dans le cas général, il y a donc $n!$ possibilités (le nombre de permutations), chacune ayant la probabilité $1/n!$ d'apparaître. On remarque donc que le nombre minimum d'affectations est égal à 1 (avec la probabilité $1/n$, le

plus grand élément est dans la case 1), le nombre maximum est égal à n (avec la probabilité $1/n!$, les éléments sont triés par ordre croissant) et le nombre moyen est égal H_n (à l'itération i, il y a une probabilité $1/i$ d'avoir une affectation) c'est à dire $\log(n)$.

Algorithme 4.3 La recherche d'u n élément X dans un tableau trié t par le recherche dichotomique.

```
i = 1; j = n;
while i ≠ j do
   aux = (i + j)div2
   if X > t[aux] then
      i = aux + 1 ;
   else
      j = aux
   end if
end while
if X = t[i] then
   retourne i
else
   « X n'est pas dans le tableau »
end if
retourner m ;
```

Pour simplifier supposons que $n = 2^k$. Le nombre de comparaisons est égal à k, c'est- à-dire $\log_2(n)$, si l'o n ne compte pas la comparaison extérieure à la boucle tant que. Ainsi une trentaine de comparaisons suffisent pour chercher un élément dans un tableau contenant 109 éléments. Ce résultat est à comparer à celui de la recherche séquentielle (de l'ordre de n comparaisons dans le pire des cas). Améliorer la recherche séquentielle en utilisant le fait que les éléments sont triés ne permet de gagner qu'un facteur 2 en moyenne.
Peut-on améliorer la recherche dichotomique dans le pire des cas ? A priori non car il y a n conclusions possibles et donc il est nécessaire d'avoir n exécutions possibles et chaque comparaison ne peut multiplier au plus par deux le nombre d'exécutions différentes.
De façon générale : si on a un algorithme basé sur les comparaisons avec p exécutions différentes nécessaires alors le nombre de comparaisons au pire est supérieur à $\log_2(p)$. Une façon de se convaincre de ce résultat est de construire l'arbre de décision. L'arbre de décision est un arbre binaire qui représente toutes les exécu-

	Taille de l'instance (n)				
Fonction	20	30	40	50	60
n	$0,00002$ sec.	$0,00003$ sec.	$0,00004$ sec.	$0,00005$ sec.	$0,00006$ sec.
n^2	$0,0004$ sec.	$0,0009$ sec.	$0,0016$ sec.	$0,0025$ sec.	$0,0036$ sec.
n^3	$0,008$ sec.	$0,27$ sec.	$0,064$ sec.	$0,125$ sec.	$0,216$ sec.
n^5	$3,2$ sec.	$24,3$ sec.	$1,7$ min.	$5,2$ min.	13 min.
2^n	1 sec.	$17,9$ min.	$12,7$ jours	$35,7$ jours	366 siècles
3^n	58 min.	$6,5$ années	3855 siècles	2×10^8 siècles	$1,3 \times 10^{13}$ siècles

TABLE 4.1 – Comparaison de temps d'exécution selon la complexité de l'algorithme, pour une machine effectuant un million d'opérations par seconde.

Complexité	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
$T * 10$	t	$t + 3,32$	$10 * t$	$(10 + \epsilon) * t$	$100 * t$	$1000 * t$	t^{10}
$t * 10$	∞	n^{10}	$10 * n$	$(10 - \epsilon) * n$	$3.16 * n$	$2.15 * n$	$n + 3.32$

TABLE 4.2 – Evolutions mutuelles du temps t (ligne 1, la taille des données est multipliée par 10)) et de la taille des données n (le temps est multiplié par 10).

tions possibles d'une certaine taille. Les feuilles de l'arbre indiquent les résultats des différentes exécutions (remarque : deux exécutions différentes peuvent donner le meme résultat). Les noeuds interne correspond à la comparaison de deux éléments effectuées par l'algorithme. Si la comparaison est VRAI (resp. FAUX) alors le sous arbre gauche (resp. droit) représente la suite de l'exécution.
Le tableau ci-dessous permet d'avoir une idée de la durée d'exécution d'un algorithme en fonction de l'ordre de grandeur de son exécution. Si la complexité est inférieure à $\log(n)$ alors il n'y a aucune contrainte sur la taille des données, si la complexité est inférieure à n, seules des données de taille très grande peuvent poser des problèmes (données codées sur quelques dizaines de bits), si la complexité est inférieure à nk avec $k > 1$ seules les données de taille moyenne peuvent être traitées, si la complexité est exponentielle seules les données de petite taille (quelques dizaines) peuvent être traitées.
Dans ce qui suit, étant données deux fonctions f et g, nous utiliserons les propriétés suivantes :

- $f = o(g)$ si et seulement si $\lim_{n\to\infty}(f/G) = 0$
- $f = \gamma(g)$ si et seulement si $g = o(f)$
- $f = \theta(g)$ si et seulement si $f = O(g)$ et $g = O(f)$.

Dans la pratique, les notations $O()$ et $\theta()$ sont utilisées de la même façon et signi-

fient plutôt $\theta()$. Pour des raisons de conformité avec le reste de la littérature nous allons employer $O()$, même si les deux fonctions sont de même ordre.
La taille de l'instance d'un problème peut-être considérée comme étant l'un des ordres de grandeur suivants : le nombre de ses variables ou de ses variables ou de ses contraintes, ou le nombre de sommets (arêtes, arcs) dans le graphe qui représente l'instance, ... Les résultats, dans le cadre de l'étude de la complexité d'un problème, changent pas si on définit la taille d'une instance d'une manière ou d'une autre. En pratique, la taille correspond au nombre de bits nécessaires à un codage de toutes les informations qui caractérisent de manière univoque le problème.

4.2 Comment calculer la complexité d'un algorithme

C'est un problème complexe (en fait il est indécidable). Prenons par exemple la fonction suivante $f(1) = 1, f(n) = n/2$ si n est pair, $f(n) = 3n + 1$ sinon. On conjecture que cette fonction est définie pour tout n et que sa valeur est 1 pour tout n. Donc à l'heure actuelle personne ne connait la complexité du calcul de cette fonction (en utilisant le programme récursif induit par la définition). On ne sait même pas si elle boucle ou non.
Soit $P(X)$: le nombre de fois que l'on exécute l'opération élémentaire sur la structure X. Dans la suite nous allons voir quelques règles (malheureusement non complètes) qui permettront souvent de calculer $P(X)$:

- $X = X1; X2$ alors $P(X) = P(X1) + P(X2)$.
- X = si C alors $X1$ sinon $X2$ alors $P(X) \leq P(C) + max(P(X1), P(X2))$.
- Et si $p(C)$ est la probabilité que C soit vrai alors $moy(P(X)) = p(C) \times moy(P(X1)) + (1 - p(C)) \times moy(P(X2))$. Si X est une boucle pour i allant de a à b faire Xi alors $P(X) = \sum P(Xi)$.

Lors d'un appel à une fonction avec comme paramètre une donnée d, l'on nombre d'exécutions de l'opération élémentaire effectuées pendant l'appel. Si l'appel est récursif alors nous sommes confrontés à la résolution d'une équation de récurrence.
Exemples :
Soit t_n le nombre de multiplications pour calculer $fact(n)$ (c'est évidemment

Algorithme 4.4 Calcul de factorielle $n!$ $fact(n)$

if $n = 0$ **then**
  retourner 1 ;
else
  retourner $n * fact(n-1)$
end if

l'opération fondamentale pour le calcul de $n!$). On a $t_0 = 0$ et pour $n > 0, t_n = 1 + t_{n-1}$. La résolution est immédiate et la solution est $t_n = n$.

Algorithme 4.5 Calcul de fibonacci

if $n < 2$ **then**
  retourner n ;
else
  retourner $fibo(n-1) + fibo(n-2)$;;
end if

Soit t_n le nombre d'additions pour calculer fondamentale pour le calcul de Fibonacci de n).
On a $t_0 = t_1 = 0$ et pour $n > 1, t_n = 1 + t_{n-1} + t_{n-2}$.
Alors la solution est égal à $\alpha \times x_1 + \beta \times x_2$ sinon à $(\alpha \times n + \beta)x_1$. Les valeurs initiales de u permettent de calculer α et β.
La résolution est moins triviale que pour la formule précédente mais elle est faisable : $t_n = \theta(\rho^n)$ où ρ est le nombre d'or $((1 + \sqrt{5})/2)$. En pratique dans de nombreux cas on ne sait pas résoudre l'équation de récurrence et l'on se contente de l'encadrer ou de la majorer.

Algorithme 4.6 Calcul de la fonction $mystere$

if $n < 2$ **then**
  retourner 1 ;
else
  retourner $mystere(n-1) + mystere(ndiv2) + 1$;
end if

Soit t_n le nombre d'additions pour calculer $mystere(n)$ (c'est évidemment l'opération fondamentale pour le calcul de mystère de n). On a $t_0 = t_1 = 0$ et pour $n > 1, t_n = 2 + t_{n-1} + t_n \div 2$.

Le résultat sera compris entre les 2 résultats précédents (on peut évidemment améliorer cet encadrement).
Un autre exemple plus difficile :
Soit $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$. A première vue, cela semble délicat de résoudre directement, mais en utilisant un changement de variables, en posant $n = 2^m$.
On trouve $T(2^m) = 2T(2^{m/2}) + m \log 2$.
Maintenant on peut renommer $S(m)$ l'expression $T(2^m)$ pour produire la nouvelle récurrence $S(m) = 2S(m/2)+m \log 2$. La résolution donne $S(m) = 0(m \log m)$.
En redonnant $S(m)$ la valeur $T(n)$ on trouve $T(n) = T(2^m) = S(m) = 0(m \log m) = 0(\log n \log \log(n))$.

4.2.1 Rappel sur la résolution des équations de récurrence : équation linéaire d'ordre 1 et d'ordre 2, diviser pour régner

Quelques exemples de résolution simple :

- Calcul du nombre de nœuds d'un arbre binaire complet de hauteur n avec 2 stratégies différentes : niveau par niveau ou une racine plus 2 arbres binaires parfait de hauteur $n-1$: $u_0 = 1$, avec $u_n = u_{n-1} + 2n$ si $n > 0$ ou $u_0 = 1$ avec $u_n = 2 * u_{n-1} + 1$ si $n > 0$
  Ce sont des équations de récurrence linéaire d'ordre 1 : résolution par substitution ou par récurrence si l'on a une idée de la solution.
- $u_1 = c$ et $u_n = a * u_n/2 + b$ si $n > 0$. Indication : on pose $n = 2^k$ et l'on se ramène à une équation d'ordre 1.
- Résolution de $u_n = a * u_{n-1} + b * u_{n-2}$ avec $(u_0, u_1, a, b) = (1, 2, 2, 3)$ et $(2, 4, 4, 4)$ et de $u_n = a*u_{n-1}+b*u_{n-2}+4$ avec $(u_0, u_1, a, b) = (1, 3, 2, 3)$.
  On calcule les racines e $P(x) = x^2 - ax - b$. Si les deux racines x_1 et x_2 sont différentes alors la solution est égal à $\alpha x_1^n + \beta x_2^n$ sinon à $(\alpha n + \beta)x_1^n$.
  Les valeurs initiales de u permettent de calculer α et β.
  Soit t_n le nombre d'additions pour calculer pour le calcul de Fibonacci de n. On a $t_0 = t_1 = 0$ et pour $n > 1, t_n = 1 + t_{n-1} + t_{n-2}$. Alors la solution est égal à $\alpha x_1 + \beta x_2$ sinon à $(\alpha n + \beta)x_1$. Les valeurs initiales de u permettent de calculer α et β.
  $fibo(n)$ (c'est évidemment l'opération La résolution est moins triviale que pour la formule précédente mais elle est faisable : $t_n = \theta\rho n$ oùρ est le nombre d'or $\frac{1+\sqrt{5}}{2}$.

En pratique dans de nombreux cas on ne sait pas résoudre l'équation de récurrence et l'on se contente de l'encadrer ou de la majorer.

Exemple :
Un algorithme qui nécessite n^2 additions, par exemple, est considéré de la même complexité (dans notre cadre) qu'un autre qui aurait besoin de n^2 additions et n^2 multiplications. Cette complexité est en $O(n^2)$.
On se place dorénavant dans le cadre dit du « pire des cas », c'est à dire que la complexité d'un algorithme est le nombre d'opérations effectuées sur l'instance qui représente la pire configuration pour son déroulement.
Exemple :
Considérons un algorithme qui tri les éléments d'une liste en ordre croissant en les comparant deux à eux et en les interchangeant, si nécessaire, pour les placer dans le bon ordre relatif et supposons que la mesure de sa complexité soit le nombre des échanges effectués. La pire configuration pour un tel algorithme est une liste triée en ordre décroissant.

Définition 4.2.1 *La complexité d'un problème est la complexité du meilleur (plus rapide) algorithme qui le résout correctement.*

Exemple :
La complexité du tri d'une liste de n éléments est $O(n \log n)$, la complexité de la recherche d'élément dans une liste triée à n élément est en $O(\log n)$, la complexité du parcours dans un graphe $G(V, E)$ est en $O(|E|)$, ...
La complexité en temps n'est pas la seule mesure de performance d'un algorithme. Une autre mesure, moins fréquemment utilisée (mais pas moins intéressante mathématiquement), est la complexité en espace. Elle spécifie l'espace mémoire utilisé par l'algorithme pour résoudre un problème.
Fixons une taille n et une fonction $f(n)$ et définissons les classes de problèmes suivantes :

- **TIME f(n)** la classe de problèmes dont la complexité (en temps) sur une instance de taille n est en $O(f(n))$.
- **SPACE f(n)** : la classe de problèmes qui peuvent être résolus, sur une instance de taille n, en utilisant un espace mémoire en $O(f(n))$.

En utilisant ces notations, nous pouvons spécifier les classes génarales suivantes :
PTIME : la classe de tous les problèmes qui peuventêtre résolus en temps polynomial en la taille de leurs instances

$$PTIME = \cup_{k=0}^{\infty} TIMEn^k$$

EXPTIME : la classe de tous les problèmes qui peuvent être résolus en temps exponentiel en la taille de leurs instances

$$EXPTIME = \cup_{k=0}^{\infty} TIME 2^{n^k}$$

PSPACE : la classe de tous les problèmes qui peuvent être résolus en utilisant un espace mémoire polynomial en la taille de leurs instances

$$PSPACE = \cup_{k=0}^{\infty} SPACE n^k$$

Les relations suivantes eixstent entre les trois classes que nous venons de définir :

$$PTIME \subseteq PSPACE$$

$$PSPACE \subseteq EXPTIME$$

$$PTIME \subset EXPTIME$$

Par ailleurs, le fait de savoir si les deux premières inclusions sont strictes restent un problème ouvert.

Chapitre

5 Parcours dans un graphe

Contents

Résumé

Ce chapitre traite du problème de parcours dans un graphe.

5.1 Introduction

Le parcours d'un graphe $G = (X, A)$ consiste à déterminer l'ensemble des sommets situés sur des chemins d'origine un sommet fixé s. Ceci revient à calculer certaines arborescences dont l'ensemble des sommets et des arcs sont inclus dans X et A respectivement.
Le but est de développer des méthodes de faible complexité, afin de pouvoir ensuite réduire la complexité de plusieurs algorithmes qui résolvent des problèmes plus complexes.
Le principe de la méthode est le suivant :

- Marquer le sommet de départ s,
- tant qu'on trouve un arc (x, y) tel que le sommet x est marqué et le sommet y non marqué, alors on marque le sommet y.
- Les sommets marqués sont les descendants du sommet s.

5.2 Algorithme général

Plus précisément la méthode est décrite comme suit :
Condidérons d'abord que notre graphe G est représenté par les listes de successeurs. On aura besoin de deux structures de données supplémentaires.

- un tableau `mark` indiquant les sommets visités (marqués) et
- un ensemble `reste_succ` contenant les sommets dont il reste des successeurs à examiner.

La question qui se pose est comment l'ensemble `reste_succ` est-il géré ?

Théorème 5.2.1 *La complexité de l'algorithme d'exploration est en $\mathcal{O}(m)$.*

Preuve Remarquer qu'un sommet x marqué ne peut pas retourner dans l'ensemble `reste_succ`. La boucle répéter examine tous les successeurs d'un sommet et ensuite enlève ce sommet de l'ensemble `reste_succ`. Donc on a

$$\begin{aligned}\sum_{x=1}^{n}(d^+(x)+1) &= n+\sum_{x=1}^{n} d^+(x)\\ &= n+m\end{aligned}$$

On considère en général que $m > n$, la complexité de l'algorithme est donc en $\mathcal{O}(m)$. □

Algorithme 5.1 Algorithme général d'exploration

initialiser `mark` à faux
mark[s] := vrai, reste_succ={s}
repeat
  prendre un sommet x dans `reste_succ`
  if tous les successeurs de x sont examinés **then**
    `reste_succ = reste_succ - {`x`}`
  else
    prendre y le successeur suivant de x
    if y non encore marqué **then**
      marquer le sommet y
      mettre y dans `reste_succ`
    end if
  end if
until `reste_succ` soit vide

5.3 Exploration en largeur

Si cet ensemble est géré comme une file alors l'algorithme traite d'abord tous les successeurs de s, ensuite les successeurs des successeurs et ainsi de suite. La méthode qu'on obtient est appelée exploration en largeur (breadth-first-search). L'algorithme précédent prend la forme suivante :

Algorithme 5.2 Algorithme d'exploration en largeur

initialiser `mark` à faux
mark[s] := vrai
mettre s dans la file
repeat
  prendre x en tête de la file
  for tout successeur y non marqué de x **do**
    marquer le sommet y
    mettre y à la fin de la file
  end for
until la file soit vide

Exemple 5.3.1 Soit $G = (V, E)$ le graphe non orienté représenté à la figure 5.1.

Puisque le graphe est non orienté tous les voisins d'un sommet x sont considérés comme successeurs de ce sommet.

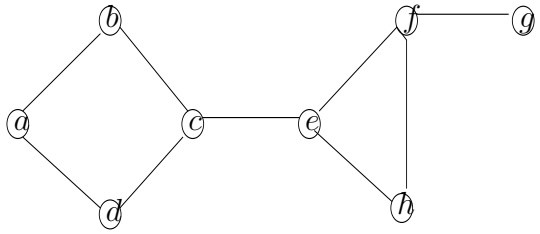


FIGURE 5.1 –

L'évolution de la file en partant du sommet a est indiquée à la figure 5.2.

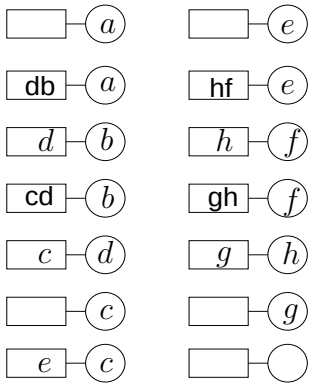


FIGURE 5.2 –

On obtient l'arborescence de la figure 5.3.

□

5.4 Exploration en profondeur

Si l'ensemble `reste_succ` est géré comme une pile alors l'algorithme traite d'abord un seul successeur de s, ensuite un successeur du successeur et ainsi de suite, allant le plus loin possible le long d'un chemin d'origine s. Si il n'y a plus de successeurs à traiter, on revient en arrière pour continuer sur un autre chemin. La méthode qu'on obtient est appelée exploration en profondeur (depth-first search). L'algorithme initial prend la forme suivante :

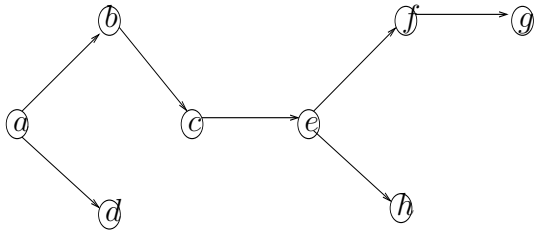


FIGURE 5.3 –

Algorithme 5.3 Algorithme d'exploration en profondeur

```
initialiser mark à faux
mark[s] := vrai
empiler(reste_succ, s)
repeat
   prendre x (le sommet de reste_succ)
   if tous les successeurs de x sont examinés then
      supprimer(reste_succ,x)
   else
      prendre le sommet y successeur de x
      if y non encore marqué then
         marquer le sommet y
         empiler(reste_succ, y)
      end if
   end if
until reste_succ soit vide
```

Exemple 5.4.1

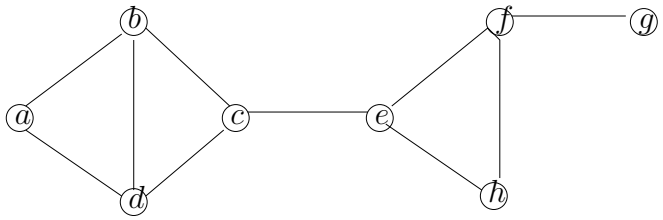


FIGURE 5.4 – *ff*

Soit le graphe de la figure 5.4. La figure 5.4 montre une évolution possible de la

pile en partant du sommet a. L'arborescence obtenue est représentée à la figure 5.4.

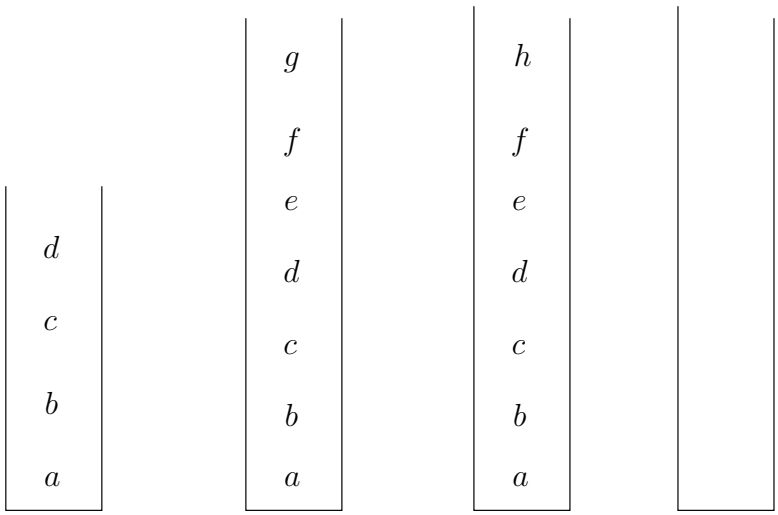


FIGURE 5.5 – *Evolution de la pile*

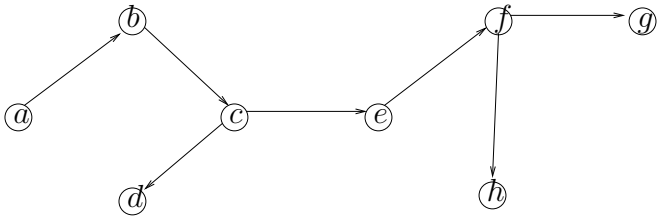


FIGURE 5.6 – *Arboresence associé au parcours en profondeur en suivant la pile donné par la figure 5.5*

□

5.5 Remarques

Remarque 5.1 La complexité des deux méthodes d'exploration est en $\mathcal{O}(m)$, avec m le nombre d'arcs du graphe G.

Remarque 5.2 Pour mémoriser l'arborescence il suffit d'utiliser un tableau `pere` initialisé à 0, et mettre à jour après le marquage d'un sommet y le sommet d'où l'on vient, par exemple `pere[y] = x`.

Remarque 5.3 Puisque dans l'exploration en largeur, les sommets sont atteints par ordre de distance croissant au sommet de départ s (en nombre d'arcs), on peut l'utiliser pour trouver l'arborescence des plus courts chemins en nombre d'arcs. En effet, il suffit d'utiliser un tableau `distance` initialisé à 0 et le mettre à jour lorsqu'on marque un successeur y de x par `distance[y] = distance[x]+1`.

Remarque 5.4 Si on s'intéresse à trouver le chemin le plus court à partir du sommet s vers un sommet fixé s', on peut utiliser la remarque ... et arrêter la procédure lorsque le sommet s' est atteint. Le tableau `distance` fournit la longueur du chemin et le tableau `pere` permet de récupérer le chemin.

5.6 Connexité

5.6.1 Introduction et définitions

La vérification de la connexité d'un réseau électrique ou d'un réseau téléphonique est un problème pratique majeur : on veut savoir si deux points sont reliés entre eux sans vouloir connaître comment. Exprimé en termes de graphes non orientés, ce problème consiste à déterminer des composantes connexes : tous les sommets d'une même composante sont mutuellement accessibles et deux sommets de deux composantes connexes distinctes ne peuvent être joints par aucune chaîne.

Définition 5.6.1 *Un graphe non orienté est dit connexe si pour toute paire de sommet u et v, il existe une chaîne reliant u et v.*

Définition 5.6.2 *Un graphe orienté est dit fortement connexe si pour toute paire de sommets distincts u et v il existe un chemin de u vers v et un chemin de v vers u.*

Définition 5.6.3 *On appelle composante fortement connexe d'un graphe orienté un sous-graphe fortement connexe maximal, c'est à dire un sous-graphe fortement connexe qui n'est pas strictement contenu dans un sous-graphe fortement connexe.*

Définition 5.6.4 *On appelle composante connexe dans graphe non orienté un sous-graphe connexe maximal*

Dans cette partie, nous allons proposer quelques algorithmes qui permettent de trouver les composantes d'un graphe non orienté, puis nous nous plaçons dans le cadre des graphes orientés pour présenter un algorithme qui donnent les composantes fortement connexes.

5.6.2 Fermeture transitive

Nous allons utiliser la notion de fermeture transitive pour vérifier si un graphe est connexe. Pour cela, nous utiliserons à l'algorithme de Roy-Warshall.

Définition 5.6.5 *Nous appelons fermeture transitive d'un graphe non orienté $G = (S, A)$ le graphe $G^* = (S, A^*)$ tel que pour tous les sommets x et y de S, il existe une arête entre x et y dans G^* si et seulement si, il existe une chaîne entre x et y dans G.*

Un graphe G est dit connexe si et seulement si sa fermeture transitive G^* est un graphe complet (Nous rappelons qu'un graphe non orienté est complet si et seulement si il existe une arête entre deux sommets quelqconques). De même, deux sommets x et y sont dans la même composante connexe de G si et seulement si il existe une arête entre x et y dans la fermeture transitive G^* de G.

Principe de l'algorithme :

Soit C la matrice d'adjacence de G. On peut calculer la matrice d'adjacence C^* de G^* de la manière suivante : Supposons que $C_k[i, j]$ représente l'existence d'une chaîne de i à j passant par des sommets inférieurs ou égaux à k. Il existe une chaîne de i à j passant seulement par des sommets inférieurs ou égaux à k si,

- soit il existe une chaîne de i à j passant seulement par des sommets inférieurs ou égaux à $k - 1$,
- soit il existe une chaîne de i à k passant par des sommets inférieurs ou égaux à $k - 1$ et une chaîne de k à j passant par des sommets inférieurs ou égaux à $k - 1$.

Nous avons donc :

$$C_k[i, j] = C_{k-1}[i, j] \text{ ou } (C_{k-1}[i, k] \text{ et } C_{k-1}[k, j])$$

Remarque 5.5 Il est important de noter que :

Algorithme 5.4 L'algorithme de Roy-Warshall

for $k = 1$ à n **do**
  for $i = 1$ à n **do**
    for $j = 1$ à n **do**
      $C_{[i,j]} := C_{[i,j]}$ ou ($C_{[i,k]}$ et $C_{[k;j]}$)
    end for
  end for
end for

— la matrice ne peut évoluer entre deux k consécutifs ou non.
— Quand le graphe est orienté, il faut remplacer le mot arête par arc et chaîne par chemin.
— Si il existe un arc direct entre x_k et x_j alors la chaîne existe entre x_k et x_j passant par des sommets inférieurs à $k - 1$.

Lemme 5.6.1 *L'algorithme 5.6.2 détermine la fermeture transistive en $\theta(n^3)$ opérations.*

Preuve C'est évident. □

Pour tester si deux sommets i et j sont dans la même composante connexe, il suffit de regarder si $C_{[i,j]} = 1$, ce qui se fait en un temps constant. Cependant, vu le coût de la construction de la matrice C, il est plus avantageux d'utiliser les parcours en profondeur ou en largeur, pour trouver les composantes connexes. L'algorithme 5.6.2 est surtout utilisé lorsque nous voulons construire effectivement la fermeture transitive d'un graphe.
Remarque : Il est important de souligner que l'algorithme 5.6.2 calcule aussi la fermeture transitive d'un graphe orienté.
Exemple 5.6.1

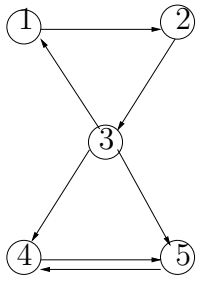


FIGURE 5.7 – *Exemple de graphe pour l'algorithme de Roy-Warshall*

$$C^1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

Il y a un 1 sur la diagonale car il existe un chemin de longueur 0 entre x_i et x_i.

$$C^2 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

$$C^3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

et ensuite $C^5 = C^4 = C^3$ □

5.7 Composantes fortement connexes

5.7.1 Préliminaires et Algorithmes

Rappelons dans un premier temps, deux algorithmes 5.5 et 5.6.

Chaque fois qu'un sommet v est découvert pendant le balayage d'une liste d'adjacence d'un sommet u, le parcours enregistre ce fait en donnant la valeur u à $\pi[v]$, le père de v.
Les sommets sont coloriés pendant le parcours pour indiquer leur état.Chaque sommet est initialement blanc, puis grisé quand il est découvert pendant le parcours, et enfin noirci en fin de traitement. Cette technique permet de s'assurer que chaque sommet appartient à une arborescence en profondeur et une seule, de sorte que ces arborescences restent disjointes.
Le parcours entre profondeur date chaque sommet. Chaque sommet v porte deux dates : la première, $d[v]$ marque le moment où v a été découvert pour la première fois (et colorié en gris), et la seconde, $f[v]$, enregistre le moment où le parcours a fini d'examiner la liste d'adjacence de v (et le colorie en noir).
L'algorithme 5.5 enregistre le moment où elle découvre le sommet u dans la variable $d[u]$, et le moment où elle termine le traitement du sommet u dans la variable $f[u]$. Ces dates sont des entiers compris entre 1 et $2n$, puisque découverte et fin de traitement se produisent une fois et une seule fois chacun des n sommets. $\forall u, d[u] < f[u]$ (la date de découverte et forcément inférieur à la date de fin de traitement . Le sommet u est blanc avant l'instant $d[u]$, gris entre $d[u]$ et $f[u]$, et noir après.

Algorithme 5.5 Algorithme de parcours en profondeur

for $\forall x \in V$ **do**
  $couleur[u] := blanc$;
  $\pi[u] := nul$
end for
$tps := 0$;
for $\forall x \in V$ **do**
  if $couleur[u] := blanc$ **then**
    $VISITER - PP(u)$
  end if
end for

La variable $temps$ est une variable globale qui sert à la datation.
Clairement, la complexité de l'algorithme est en $\theta(V + |E|)$.

Algorithme 5.6 Algorithme de Visite-PP

$couleur[u] := gris$; (le sommet blanc u vient d'être découvert)
$d[u] := tps := tps + 1$;
for $\forall v \in Voisin(u)$ **do**
  if $couleur[u] = blanc$ **then**
    $\pi[v] := u$;
    $VISITER - PP(v)$
  end if
end for
$couleur[u] := noir$;
$f[u] := tps := tps + 1$

5.7.2 Propriétés

Théorème 5.7.1 *Dans un parcours en profondeur d'un graphe $G = (V, E)$ (orienté ou non), pour deux sommets quelconques u et v, les conditions suivantes sont en exclusion mutuelles :*

1. *les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont complétement disjoints*
2. *l'intervalle $[d[u], f[u]]$ est entièrement inclus dans l'intervalle $[d[v], f[v]]$, et u est un descendant de v dans l'arborescence en profondeur correspondante, ou*
3. *l'intervalle $[d[v], f[v]]$ est entièrement inclus dans l'intervalle $[d[u], f[u]]$, et v est un descendant de u dans l'arborescence en profondeur correspondante.*

Preuve

1. Premier cas $d[u] < d[v]$. Etudions les deux sous-cas :
   (a) Si $d[v] < f[u)$, va été découvert pendant que $u$étant encore gris. Cela implique que v est un descendant de u. Par ailleurs, puisque la découverte de v est plus récente que celle de u, tous les arcs qui en partent sont explorés, et le traitement de v se termine avant que le parcours ne revienne à u, et l'intervalle $[d[u], f[u]]$.
   (b) Maintenant $f[u] < d[v]$, et $d[u] < f[u]$ implique les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont complétement disjoints.
2. Pour le cas où $d[v] < d[u]$, il suffit d'inverser les rôles de u et v, ci-avant

□

Corollaire 5.7.1 *Le sommet v est un descendant propre du sommet u dans la forêt en profondeur d'un graphe G (orienté ou non) si et seulement si $d[u] < d[v] < f[v] < f[u]$.*

Preuve
Immédiat en utilisant 5.7.1

□

Théorème 5.7.2 (Théorème du chemin blanc) *Dans un forêt en profondeur d'un grpahe $G = (V, E)$ (orienté ou non) un sommet v est un descendant d'un sommet u si et seulement si à la date $d[u]$ où le parcours a découvert u, il existe un chemin (de G) constitué de sommets blancs de u et v.*

Preuve

— Supposons que v soit un descendant de u. Soit w un sommet quelconque sur le chemin de u à v dans une arborescence en profondeur, de sorte que w soit descendant de u. D'après le Corollaire 5.7.1, $d[u] < d[w]$, donc w est blanc à la date $d[u]$.

— Supposons qu'à la date $d[u]$, il existe un chemin de G constitué de sommets blancs de u à v, mais que v ne devienne pas un descendant de u dans une arborescence en profondeur. On peut, sans perte de généralité, supposer que tous les autres sommets du chemin deviennent descendants e u. (Si ce n'est pas le cas, on choisit pour v le sommet le plus proche de u dans le chemin parmi ceux qui ne deviennent pas descendants de u).
Sit w le père de v dans le chemin, tel que w soit descendant de u, (w et u peuvent être un seul et même sommet); d'après le Corollaire 5.7.1, $f[w] \leq f[u]$. On remarque que v doit être découvert après u, mais avant que le traitement de w soit terminé. Donc, $d[u] < d[v] < f[w] \leq f[u]$. Le théorème 5.7.1. implique alors que l'intervalle $[d[v], f[v]]$ est entièrement inclus dans l'intervalle $[d[u], f[u]]$. D'après le Corollaire 5.7.1, v est finalement un descendant de u.

□

5.7.3 Algorithme pour CFC

Nous allons utuliser les deux algorithmes précédents, pour déterminer les composantes fortement connexes.

Définition 5.7.1 *Une composante fortement connexe d'un graphe orienté $G = (V, E)$ est un ensemble maximal de sommets $R \subseteq V$ tel que pour chaque paire de sommets u et v dans R il existe un chemin allant de u à v et réciproquement ; autrement dit, les sommets u et v sont mutuellement accessibles.*

Notre algorithme de recherche des composantes fortement connexes d'un graphe $G = (V, E)$ utilise le transposé de G, défini par ${}^tG = (V, {}^tE)$ où ${}^tE = \{(u, v) : (v, u) \in E\}$. Autrement dit, ${}^tE$ est constitué des arcs de G dont le sens a été inversé.
Quelle est complexité de la construction, par une représentation par listes d'adjacence de G ?
L'algorithme 5.7 résoud en temps linéaire le problème de la recherche de composante fortement connexe. Le principe est simple : il est intéressant d'observer que G et ${}^tG$ ont exactement les mêmes composantes fortemetn connexes : u et v sont acessibles dans G l'un à partir d e l'autre si et seulement si ils le sont aussi dans ${}^tG$.

Algorithme 5.7 Algorithme de Visite-PP

Appeler $PP(G)$ pour calculer les dates de fin de traitement $f[u]$
Appeler ${}^tG$
Appler $PP({}^tG)$, mais dans la boucle principale de PP, considérer les sommets par ordre de $f[u]$ (calculés à la ligne précédente) décroissant
Imprimer les sommets de chaque arborescence de la forêt préfixe obtenue à l'étape précédente comme une composante fortement connexe séparée

Lemme 5.7.1 *Si deux sommets se trouvent dans la même composante fortement connexe, aucun chemin entre eux ne sort de la composante fortement connexe ;*

Preuve
Soient u et v deux sommets appartenant à la même composante fortement connexe. Ainsi, il existe un chemin de u vers v et un autre de v vers u. Soit w un sommet sur le chemin de $u \rightsquigarrow w \rightsquigarrow v$, w est accessible depuis u. Par ailleurs, $\exists v \rightsquigarrow u$. Sachant que u est accessible depuis w en suivant le chemin $w \rightsquigarrow v \rightsquigarrow u$,

u et w appartiennent donc à la même composante fortement connexe. Le chemin w est quelconque, le lemme est prouvé.

□

Théorème 5.7.3 *Lors d'un parcours en profondeur quelconque, tous les sommets appartenant à la même composante fortement connexe se trouvent dans la même arborescence en profondeur.*

Preuve
Parmi les sommets de la composante fortement connexe, appelons r le premier sommet découvert. Sachant que r est le premier, les autres sommets de la composante fortement connexe sont blancs au moment de sa découverte. Il existe des chemins allant de r vers tous les autres sommets de la composante fortement connexe (d'après le Lemme 5.7.1), tous les sommets qui les composantes sont blancs. Donc, d'après le Théorème 5.7.2, tous les sommets de la composante fortement connexe deviennent des descendants de r dans l'arborescence en profondeur.

□

Définition 5.7.2 *Un aïeul $\phi(u)$ d'un sommet u qui est, parmi les sommets w accessibles à partir de u, celui qui termine le parcours en profondeur de la ligne 1 le dernier.*

Autrement dit $\phi(u) =$ le sommet w tel que $u \rightsquigarrow w$ et $f[u]$ est maximale .
On notera que $\phi(u) = u$ est possible car u est accessible à partir de lui-même, et $f[u] \leq f[\phi(u)]$.
On peut également montrer que $\phi(\phi(u)) = \phi(u)$, en raisonnant de la manière suivante. Pour deux sommets $u, v \in V$ quelconques,
$u \rightsquigarrow v$ implique $f[\phi(v)] \leq f[\phi(u)]$,
puisque $\{w : v \rightsquigarrow w\} \subseteq \{w : u \rightsquigarrow w\}$ et que l'aïeul possède la date de fin de traitement maximum parmi tous les sommets accessibles. $\phi(u)$ étant accessible depuis u, la formule précédente implique $f[\phi(\phi(u))] \leq f[\phi(u)]$.
On a également $f[\phi(u)] \leq f[\phi(\phi(u))]$, par définition ϕ. Donc $f[\phi(\phi(u))] = f[\phi(u)]$, et $\phi(\phi(u)) = \phi(u)$, puisque deux sommets dont le traitement se termine au même moment forment en réalité un seul et même sommet.

Théorème 5.7.4 *Dans un graphe orienté, l'aïeul $\phi(u)$ d'un sommet $u \in V$ quelconque lors d'un parcours en profondeur de G, est un ancêtre de u.*

Preuve

— Si $u = \phi(u)$, c'est vrai.

— Si $\phi(u) \neq u$, considérons les couleurs des sommets à la date $d[u]$.

— Si $\phi(u)$ est noir, alors $f[\phi(u)] < f[u]$, ce qui contredit l'inégalité $f[u] \leq f[\phi(u)]$.

— Si $\phi(u)$ est gris, alors c'est un ancêtre de u, et le théorème est démontré.

Il reste à démontrer que $\phi(u)$ n'est pas blanc. Deux cases se présentent, suivant la couleur des éventuels sommets intermédiaires sur le chemin de u à $\phi(u)$

1. Si tous les sommets intermédiaires sont blancs, alors $\phi(u)$ devient un descendant de u, d'après le Théorème 5.7.2 (du chemin blanc). Mais alors $f[\phi(u)] < f[u]$, ce qui contredit l'inégalité $f[u] \leq f[\phi(u)]$.
2. Si un sommet intermédiaire n'est pas blanc, soit t le dernier sommet non blanc sur le chemin menant de u à $\phi(u)$. Le sommet t doit gris, puisqu'il n'existe aucun arc entre un sommet noir et un sommet blanc, et que le successeur de t est blanc. Mais il existe alors un chemin de sommets blancs de t à $\phi(u)$, ce qui signifie que $\phi(u)$ est un descendant de t d'après le théorème 5.7.2. Cela implique de $f[t] > f[\phi(u)]$, et contredit notre choix pour $\phi(u)$, puisqu'il existe un chemin de u à t.

□

Corollaire 5.7.2 *Dans un parcours en profondeur quelconque d'un graphe orienté $G = (V, E)$, les sommets u et $\phi(u), \forall u \in V$ se trouvent dans la même composante fortement connexe.*

Preuve

On a $u \rightsquigarrow \phi(u)$, d'après la définition de l'aïeul, et $\phi(u) \rightsquigarrow u$, puisque $\phi(u)$ est un ancêtre de u.

□

Théorème 5.7.5 *Dans un graphe orienté $G = (V, E)$, deux sommets $u, v \in V$ se trouvent dans la même composante fortement connexe si et seulement si ils partagent le même aïeul lors d'un parcours en profondeur de G.*

Preuve

— Supposons que u et v se trouvent dans la même composante fortement connexe. Chaque sommet accessible depuis u est accessible depuis v, et

vice versa. D'après la définition de l'aïeul, on peut conclure que $\phi(u) = \phi(v)$.

— Supposons que $\phi(u) = \phi(v)$. D'après le Corollaire 5.7.2, u se trouve dans la même composante fortement connexe que $\phi(u)$, et v se trouve dans la même composante fortement connexe que $\phi(v)$. Donc, u et v appartiennent à la même composante fortement connexe.

□

Théorème 5.7.6 *$CFC(G)$ calcule correctement les composantes fortement connexes d'un graphe orienté G.*

Preuve

La preuve est basée par induction sur le nombre d'arborescences en profondeur obtenues lors du parcours en profondeur de ${}^tG$ pour montrer que les sommets de chaque arborescence forment une composante fortement connexe. Chaque étape du raisonnement par induction démontre qu'une arborescence formée pendant le parcours en profondeur de ${}^tG$ est une composante fortement connexe, en supposant que toutes les arborescences précédentes le soient aussi. La base de l'induction est triviale, puisque pour la première arborescence obtenue, il n'existe aucun précédent, donc l'hypothèse est vérifiée par défaut.

On considère une arborescence en profondeur T de racine r obtenue lors du parcours en profondeur de ${}^tG$. Soit $C(r)$ l'ensemble des sommets d'aïeul r :

$$C(r) = \{v \in S : \phi(v) = r\}$$

— Le théorème 5.7.3 implique que tout sommet de $C(r)$ finit par se retrouver dans la même arborescence en profondeur. Sachant que $r \in C(r)$ et que r est la racine de T, chaque élément de $C(r)$ se retrouve dans T.

— On montre qu'un sommet w quelconque tel que $f[\phi(w)] > f[r]$ ou $f[\phi(w)] < f[r]$ ne se trouve pas dans T, en considérant séparément ces deux cas. Par induction sur le nombre d'arborescences trouvées, aucun sommet w tel que $f[\phi(w)] > f[r]$ on peut se retrouver dans l'arborescence T, puisqu'au moment où r est choisi, w se trouve déjà dans l'arborescence de racine $\phi(w)$. Un sommet w tel que $f[\phi(w)] < f[r]$ ne peut pas non plus se trouver dans T, car cela impliquerait $w \rightsquigarrow r$; donc, d'après la formule ($f[\phi(v)] \leq f[\phi(u)]$) et la propriété selon laquelle $r = \phi(r)$, on obtient $f[\phi(w)] \geq f[\phi(r)] = f[r]$, ce qui contredit $f[\phi(w)] < f[r]$

Ainsi, T contient uniquement les sommets u pour lesquels $\phi(u) = r$. Autrement

dit, T est exactement égal à la composante fortement connexe $C(r)$, ce qui termine la preuve.

□

Chapitre

6 Le problème du tri

Contents

Résumé

Ce chapitre traite du problème de tri.

6.1 Introduction

Le tri d'un ensemble d'objets consiste à les ordonner en fonction de clés et d'une relation d'ordre définie sur cette clé. Le tri est une opération classique et très fréquente. De nombreux algorithmes et méthodes utilisent des tris. Par exemple pour l'algorithme de Kruskal qui calcule un arbre couvrant de poids minimum dans un graphe, une approche classique consiste, dans un premier temps, à trier les arêtes du graphe en fonction de leurs poids. Autre exemple, pour le problème des éléphants, trouver la plus longue séquence d'éléphants pris dans un ensemble donné, telle que les poids des éléphants dans la séquence soient croissants et que leurs Q.I. soient décroissants, une approche classique consiste à considérer une première suite contenant tous les éléphants ordonnés par poids croissants, une deuxième suite avec les éléphants ordonnés par Q.I. décroissants, puis à calculer la plus longue sous- séquence commune à ces deux suites. Trier un ensemble d'objets est aussi un problème simple, facile à décrire, et qui se prête à l'utilisation de méthodes diverses et variées. Ceci explique l'intérêt qui lui est porté et le fait qu'il est souvent présenté comme exemple pour les calculs de complexité. Dans le cas général on s'intéresse à des tris en place, c'est- à -dire des tris qui n'utilisent pas d'espace mémoire supplémentaire pour stocker les objets, et par comparaison, c'est- à -dire que le tri s'effectue en comparant les objets entres eux. Un tri qui n'est pas par comparaison nécessite que les clés soient peu nombreuses et connues à l'avance, et peuvent être indexées facilement. Un tri est stable s'il préserve l'ordre d'apparition des objets en cas d'égalité des clés. Cette propriété est utile par exemple lorsqu'on trie successivement sur plusieurs clés différentes. Si l'on veut ordonner les étudiants par rapport à leur nom puis à leur moyenne générale, on veut que les étudiants qui ont la même moyenne apparaissent dans l'ordre lexicographique de leurs noms.
Dans ce cours nous distinguerons les tris en $O(n^2)$ (tri à bulle, tri par insertion, tri par sélection), les tris en $O(n \times logn)$ (tris par fusion, tri par tas et tri rapide, bien que ce dernier n'ait pas cette complexité dans le pire des cas) et les autres (tris spéciaux instables ou pas toujours applicables). Il convient aussi de distinguer le coût théorique et l'efficacité en pratique : certains tris de même complexité ont des performances très différentes dans la pratique. Le tri le plus utilisé et globalement le plus rapide est le tri rapide (un bon nom-quicksort) ; nous l'étudierons en TD. En général les objets à trier sont stockés dans des tableaux indexés, mais ce n'est pas toujours le cas. Lorsque les objets sont stockées dans des listes chainées, on peut soit les recopier dans un tableau temporaire, soit utiliser un tri adapté comme le tri par fusion.

6.2 Tri par sélection, tri par insersion

6.2.1 Tri par sélection

Le tri par sélection consiste simplement à sélectionner l'élément le plus petit de la suite à trier, à l'enlever, et à répéter itérativement le processus tant qu'il reste des éléments dans la suite. Au fur et à mesure les éléments enlevés sont stockés dans une pile. Lorsque la suite à trier est stockée dans un tableau on s'arrange pour représenter la pile dans le même tableau que la suite : la pile est représentée au début du tableau, et chaque fois qu'un élément est enlevé de la suite il est remplacé par le premier élément qui apparaît à la suite de la pile, et prends sa place. Lorsque le processus s'arrête la pile contient tous les éléments de la suite triés dans l'ordre croissant.

Algorithme 6.1 Tri d'un tableau par sélection

```
for i = 0 à n − 1 do
    min := i
    for j := i + 1 à n do
        if T[j] ≤ T[min] then
            min := j ;
        end if
    end for
    if min ≠ i then
        échanger(T[i], T[min]) ;
    end if
end for
```

6.2.2 Tri par insertion

Le tri par insertion consiste à $inserer$ les éléments de la suite les uns après les autres dans une suite triée initialement vide. Lorsque la suite est stockée dans un tableau la suite triée en construction est stockée au début du tableau. Lorsque la suite est représentée par une liste chainée on insère les maillons les uns après les autres dans une nouvelle liste initialement vide.
Le tri effectue $n-1$ insertions. A la ième itération, dans le pire des cas, l'algorithme effectue $i-1$ recopies. Le coût du tri est donc $\sum_{i=2}^{n}(i-1) = O(n^2)$.

Algorithme 6.2 Tri d'un tableau par insertion

```
for i = 2 à n do
    j := i, v := T[i]
    while j > 1 et v < T[j − 1] do
        T[j] := T[j − 1];
        j := j − 1;
    end while
    T[j] := v;
end for
```

Remarquons que dans le meilleur des cas le tri par insertion requiert seulement $O(n)$ traitements. C'est le cas lorsque l'élément à insérer reste à sa place, donc quand la suite est déja triée (lorsque la suite est stockée dans une liste chainée c'est la cas lorsque la liste est triée à l'envers puisqu'on insère en tête de liste).

6.2.3 Tri par fusion

Le tri par fusion (merge sort en anglais) implémente une approche de type diviser pour régner très simple : la suite à trier est tout d'abord scindée en deux suites de longueurs égales à un élément près. Ces deux suite sont ensuite triées séparément avant d'être fusionnées. L'efficacité du tri par fusion vient de l'efficacité de la fusion : le principe consiste à parcourir simultanément les deux suites triées dans l'ordre croissant de leur éléments, en extrayant chaque fois l'élément le plus petit. Le tri par fusion est bien adapté aux listes chainées : pour scinder la liste il suffit de la parcourir en liant les éléments de rangs pairs d'un coté et les éléments de rangs impairs de l'autre. La fusion de deux listes chainées se fait facilement. Inversement, si la suite à trier est stockée dans un tableau il est nécessaire de faire appel à un tableau annexe lors de la fusion, sous peine d'avoir une complexité en $O(n^2)$.
Dans le cas général, on peut évaluer à $O(n)$ le coût de la scission du tableau T et à $O(n)$ le coût de la fusion des sous-tableaux T_1 et T_2. L'équation récursive du tri par fusion est donc :

$$T(n) = 1 + O(n) + 2 \times T(n/2) + O(n)$$

On en déduit que le tri par fusion est en $O(n \log n)$. On le vérifie en cumulant les nombres de comparaisons effectuées à chaque niveau de l'arbre qui représente

Algorithme 6.3 Tri d'un tableau par fusion

if $|T| \geq 1$ **then**
  Scinder la suite T en deux sous-tableaux T_1 et T_2 de longueurs égales ;
  $T_1 = TriParFusion(T_1)$;
  $T_2 = TriParFusion(T_2)$;
  $T := fusion(T_1, T_2)$;
end if

l'exécution de la fonction (voir figure ci-dessous) : chaque noeud correspond à un appel de la fonction, ses fils correspondent aux deux appels récursifs, et son étiquette indique la longueur de la suite. La hauteur de l'arbre est donc $\log_2 n$ et à chaque niveau le cumul des traitements locaux (scission et fusion) est $O(n)$ et on déduit un coût total de $O(n) \times \log_2 n = O(n \log n)$.

Algorithme 6.4 Tri d'un tableau par fusion

for $i = 2$ à n **do**
  $j := i, v := T[i]$
  while $j > 1$ et $v < T[j-1]$ **do**
    $T[j] := T[j-1]$;
    $j := j - 1$;
  end while
  $T[j] := v$;
end for

6.3 Complexité optimale d'un algorithme de tri par comparaison

L'arbre de décision d'un tri par comparaison représente le comportement du tri dans toutes les configurations possibles. Les configurations correspondent à toutes les permutations des objets à trier, en supposant qu'ils soient tous comparables et de clés différentes. S'il y a n objets à trier, il y a donc $n!$ configurations possibles. On retrouve toutes ces configurations sur les feuilles de l'arbre, puisque deux permutations initiales distinctes ne peuvent pas produire le même comportement du tri : en effet, dans ce cas le tri n'aurait pas fait son travail sur un des deux ordonnancements. Chaque noeud de l'arbre correspond à une comparaison entre

deux éléments et a deux fils, correspondants aux deux ordres possibles entre ces deux éléments.

Théorème 6.3.1 *Tout arbre de décision qui trie n éléments a pour hauteur $\Omega(n \log n)$.*

Preuve
Prenons un arbre de décision de hauteur h qui trie n éléments. Puisqu'il existe $n!$ permutations de n éléments, chaque permutation réprésentant un ordre de tri distinct, l'arbre doit avoir au moins $n!$ feuilles. Sachant qu'un arbre binaire de hauteur h ne comporte pas plus de 2^h feuilles, on a $n! \leq 2^h$, ce qui implique , en prenant les logarithmes, $h \geq \log(n!)$. En utilisant la formule de Stirling $n! > (\frac{n}{e})^n$ avec $e = 2,71828\ldots$.
Ainsi on a $h \geq \log(\frac{n}{e})^n = n \log n - n \log e = \Omega(n \log n)$. □

6.4 Tri par dénombrement, tri par base

6.4.1 Tri par dénombrement

Le tri par dénombrement (counting sort) est un tri sans comparaisons qui est stable, c'est- à -dire qu'il respecte l'ordre d'apparition des éléments dont les clés sont égales. Un tri sans comparaison suppose que l'on sait indexer les éléments en fonction de leur clé. Par exemple, si les clés des éléments à trier sont des valeurs entières comprises entre 0 et 2, on pourra parcourir les éléments et les répartir en fonction de leur clé sans les comparer, juste en utilisant les clés comme index. Le tri par dénombrement utilise cette propriété pour tout d'abord recenser les éléments pour chaque valeur possible des clés. Ce comptage préliminaire permet de connaître, pour chaque clé c, la position finale du premier élément de clé c qui apparaît dans la suite à trier. Sur l'exemple ci-dessous on a recensé dans le tableau T, 3 éléments avec la clé $0, 4$ éléments avec la clé 1 et 3 éléments avec la clé 2. On en déduit que le premier élément avec la clé 0 devra être placé à la position 0, le premier élément avec la clé 1 devra être placé à la position 3, et le prémier élément avec la clé 2 devra être placé à la position 7. Il suffit ensuite de parcourir une deuxième fois les éléments à trier et de les placer au fur et à mesure dans un tableau annexe (le tableau R de la figure), en n'oubliant pas, chaque fois qu'un élément de clé c est placé, d'incrémenter la position de l'objet suivant de clé c. De

cette façon les éléments qui ont la même clés apparaissent nécessairement dans l'ordre de leur apparition dans le tableau initial.

Algorithme 6.5 Tri par dénombrement (T, n)

for $i = 1$ à n **do**
  $Nb[i] := 0$; {Initialisation}
end for
for $i = 1$ à n **do**
  $Nb[T[i]] := Nb[T[i]] + 1$; {Calcul des nombres d'apparitions}
end for
$Nb[k] := n - Nb[k] + 1$; {Calcul des indices du premier}
for $i = k - 1$ à 1 **do**
  $Nb[i] := Nb[i + 1] - Nb[i]$;
end for
for $i = 1$ à n **do**
  $R[Nb[T[i]]] := T[i]$;
  $Nb[T[i]] := Nb[T[i]] + 1$;
end for
Retourner T ;

6.4.2 Tri par base

Le tri par dénombrement est difficilement applicable lorsque les valeurs que peuvent prendre les clés sont très nombreuses. Le principe du tri par base (radix sort) consiste, dans ce type de cas, à fractionner les clés, et à effectuer un tri par dénombrement successivement sur chacun des fragments des clés. Si on considère les fragments dans le bon ordre (i.e. en commençant par les fragments de poids le plus faible), après la dernière passe, l'ordre des éléments respecte l'ordre lexicographique des fragments, et donc la suite est triée. Considérons l'exemple suivant dans lequel les clés sont des nombres entiers à au plus trois chiffres. Le fractionnement consiste simplement à prendre chacun des chiffres de l'écriture décimale des clés. La colonne de gauche contient la suite des valeurs à trier, la colonne suivante contient ces mêmes valeurs après les avoir trié par rapport au chiffre des unités,... Dans la dernière colonne les valeurs sont effectivement triées. Du fait que le tri par dénombrement est stable, si des valeurs ont le même chiffre des centaines, alors elles apparaitront dans l'ordre croissant de leurs chiffres des dizaines,

et si certaines ont le même chiffre des dizaines alors elles apparaitront dans l'ordre croissant des chiffres des unités.

536	592	427	167
893	462	536	197
427	893	853	427
167	853	462	462
853	536	167	536
592	427	592	592
197	167	893	853
462	197	197	893

Supposons que l'on ait n valeurs dont les clés sont fractionnées en c fragments avec k valeurs possibles pour chaque fragment. Le coût du tri par base est alors $O(c \times n + c \times k)$ puisuqe l'on va effectuer c tris par dénombrements sur n éléments avec des clés qui auront k valeurs possibles.
Si $k = O(n)$ on peut dire que le tri par base est linéaire. Dans la pratique, des entiers codés sur 4 octets que l'on framente en 4, le tri par base est aussi rapide que le Tri rapide.

6.5 Tri Shell

C'est une variante du tri par insertion. Le principe du tri shell est de trier séparément des sous-suites de la table formées par des éléments pris de h en h dans la table (on nommera cette opération h-ordonner).

Définition 6.5.1 *La suite $E = (e_1, \ldots, e_n)$ est h-ordonnée si pour tout indice $i \leq n - h, e_i \leq e_{i+h}$.*

Si h vaut 1 alors une suite h-ordonnée est triée.
Pour trier, Donald Shell le créateur de ce tri, propose de h-ordonner la suite pour une série décroissante de valeurs de h. L'objectif est d'avoir une série de valeurs qui permette de confronter tous les éléments entres eux le plus souvent et le plus tôt possible. Dans la procédure ci-dessous la suite des valeurs de h est : $\ldots, 1093, 364, 121, 40, 13, 4, 1$. La complexité de ce tri est $O(n^2)$. Avec la suite de valeurs de h précédente on atteint $O(n^{3/2})$ (admis). Cependant dans la pratique ce tri est très performant et facile à implémenter (conjectures $O(n \times (\log n)^2)$ ou $n^{1,25}$).
On notera que le tri utilisé pour h-ordonner la suite est un tri par insertion. La dernière fois que ce tri est appliqué (avec h qui vaut 1) on exécute donc simplement

Algorithme 6.6 TriShell

```
h := 1;
while h ≤ n/9 do
  h := 3h + 1;
end while
while h > 0 do
  for i := h + 1 à n do
    j := i;
    v := e_i;
    while (j > h) et (v < e_{j−h} do
      e_j := e_{j−h};
      j := j − h;
    end while
    e_j := v;
  end for
  h := h/3;
end while
```

un tri par insertion. Les passes précédentes ont permis de mettre en place les éléments de façon à ce que cette ultime exécution du tri par insertion soit très peu coûteuse.

Chapitre 7

Arbres introduction

Contents

Résumé

Nous allons voir quelques résultats concernant le problème de la recherche d'un arbre couvrant.

7.1 Introduction

L'arbre est le plus simple type de graphe qui soit non trivial. Il a des applications dans de nombreux domaines de la théorie des graphes. Le concept a ses origines historiques dans les travaux de Kirchhoff, sur la théorie des réseaux électriques et dans les travaux de Cayley sur l'énumération des molécules chimiques. Plus récemment, il a trouvé des applications importantes en informatique, en théorie de la décision et dans la linguistique.

7.2 Motivation

Soit une entreprise répartie sur plusieurs sites à travers la France. Chacun de ses sites veut pouvoir communiquer avec les autres, soit directement, soit en passant par l'intermédiaire d'autres sites. La situation actuelle est représenté à la figure 7.1, case 1. Chacune des arêtes représente une ligne de communication loué à l'année. L'entreprise décide de faire des économies et de supprimer les connexions redondantes. Ainsi par exemple, la connexion entre les sites f et h est supprimé car f et h sont déjà reliés en passant par exemple par le site g. On obtient le graphe représenté à la case 3. Si on continue ainsi, on arrive par exemple au graphe de la case 4. Ce graphe est d'un type particulier et est appelé comme nous allons le voir par la suite un arbre. Aucune autre liaison ne peut être supprimé sans perdre la possibilité de communiquer entre les sites.

7.3 Connexité

Définition 7.3.1 *Un graphe est dit* connexe *si toute paire de sommets est relié par une chaîne.*

Définition 7.3.2 *Soit un graphe $G = (V, E)$ connexe. Une arête $[x, y]$ est appelé* isthme *si le graphe $G' = (V, E - [x, y])$ n'est pas connexe.*

Définition 7.3.3 *Soit un graphe G et la relation R telle que xRy ssi les sommets x et y sont reliés par une chaîne. R est une relation d'équivalence dont les classes sont appelés* composantes connexes *de G.*

Un graphe G est donc connexe ssi il n'admet qu'une seule composante connexe.

Lemme 7.3.1 *Tout graphe connexe $G = (V, E)$ d'ordre n a au moins $n-1$ arêtes.*

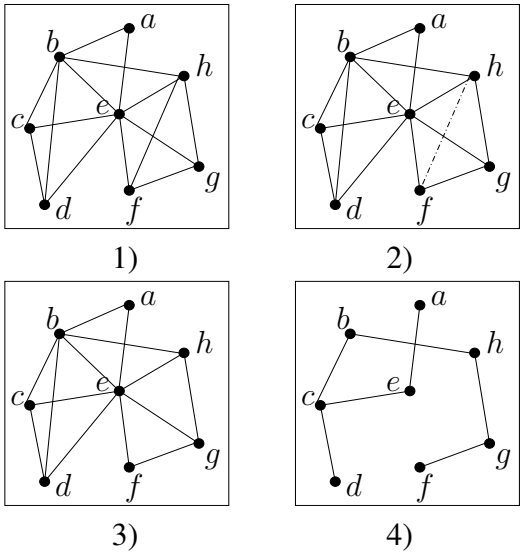


FIGURE 7.1 – Exemples d'arbres couvrant

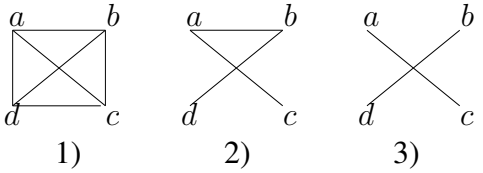


FIGURE 7.2 – Quelles sont les graphes connexes ?

Preuve La preuve est par récurrence sur n. Si $n = 2$ le lemme est vrai. Supposons que le lemme soit vrai pour tout graphe connexe d'ordre $k \leq n-1$. Soit G un graphe connexe d'ordre n, $n \geq 2$. Soit aussi un sommet v de G. Le graphe $G' = (V - \{v\}, E')$ possède l composantes connexes $G_1, G_2, \ldots, G_l$ d'ordres $n_1, n_2, \ldots, n_l$ ($l \geq 1$) chacune ayant $m_1, m_2, \ldots, m_l$ arêtes respectivement. On a donc $m_i \geq n_i - 1$, $\forall i = 1, \ldots, l$. Puisque G est connexe, il existe au moins une arête entre le sommet v et la composante G_i, $\forall i = 1, \ldots, l$. On a

$$|E| \geq l + \sum_{i=1}^{l} m_i \geq l + \sum_{i=1}^{l} (n_i - 1) = l + (n-1) - l = n - 1.$$

□

Lemme 7.3.2 *Soit $G = (V, E)$ un graphe connexe et une arête $e \in E$. Le graphe $G' = (V, E - \{e\})$ est connexe ssi e appartient à un cycle de G.*

Preuve $\Longrightarrow$ Soit G' connexe et $e = [x, y]$. Comme G' est connexe, il existe une chaîne c entre x et y dans G'. Dans G la chaîne c concaténé avec l'arête e forme un cycle entre x et y.
$\Longleftarrow$ Comme G est connexe il existe une chaîne entre toute paire de sommets si cette chaîne passe par e. On peut remplacer e par le cycle moins l'arête e. □

7.4 Arbres

Définition 7.4.1 *Un* arbre *est un graphe (non orienté)* [1]*, connexe et sans cycle.*

Définition 7.4.2 *Une* forêt *est un graphe dont les composantes connexes sont des arbres.*

Définition 7.4.3 *Un arbre* couvrant *d'un graphe est un graphe partiel qui est un arbre.*

Définition 7.4.4 *Un sommet* pendant *ou* feuille *est un sommet de degré 1.*

7.5 Théorème de caractérisation des arbres

Nous commençons par un lemme nécessaire pour la suite.

Lemme 7.5.1 *Tout arbre G, ayant au moins 2 sommets, possède 2 sommets pendants.*

Preuve
Il suffit de prendre la plus longue chaîne de G, et ses voisins extrémités ne peuvent avoir d'autres voisins que leur voisin sur la chaîne. En effet, si une extrémité avait un voisin à l'extérieure de la chaîne on pourrait rallonger la chaîne, si elle avait un voisin sur la chaîne on aurait un cycle.

□

Théorème 7.5.1 *Soit $G = (V, E)$ un arbre simple, d'ordre n et ayant m arêtes. Les propriétés suivantes sont équivalentes :*

1. *G est un arbre (connexe sans cycle)*

1. et simple

2. *G est connexe et $m = n - 1$*
3. *G est sans cycle et $m = n - 1$*
4. *G est connexe et toute arête est un isthme*
5. *G est sans cycle et l'ajout d'une arête crée exactement un cycle*
6. *il existe une chaîne et une seule entre deux sommets a et b, $\forall a, b \in V$.*

Preuve
Montrons que $1 \implies 2$ par récurrence sur n. Pour $n = 1$ le lemme est vrai. Supposons la propriété vraie pour tout arbre d'ordre $k < n$, et soit G un arbre d'ordre n. Soit $G' = (V - \{x\}, E')$ avec x un sommet pendant de G (d'après le lemme 2.5.1 un tel sommet existe). Le graphe G' est connexe, sans cycle, c'est donc un arbre à $n-1$ sommets. Par hypothèse de récurrence, il a $n-2$ arêtes. Par conséquent G possède $n - 1$ arêtes.
Montrons que $2 \implies 3$ par l'absurde. Si G avait un cycle, la suppression d'une arête du cycle laisserait le graphe G connexe d'après le lemme 7.3.2. Appelons G' ce graphe connexe. Le graphe connexe G' est d'ordre n, il possède donc au moins $n - 1$ arêtes d'après le lemme 7.3.1. Mais alors G aurait au moins n arêtes.
Montrons que $3 \implies 2$. Par l'absurde supposons qu'il existe un graphe sans cycle, non connexe avec $m = n-1$. Prenons le graphe G vérifiant cette propriété d'ordre n minimal (remarque $n \geq 2$). Comme la somme des degrés est égal à $2m$, il existe un sommet de degré 0 ou un sommet de degré 1. Dans les deux cas on enlève ce sommet la propriété reste vérifié, ce qui contredit le fait que G est d'ordre minimal. (remarque si l'on retire un sommet de degré 0, on obtient un graphe avec autant de sommets que d'arêtes, on doit alors retirer une arête, ce qui conserve la propriété)
Montrons que $2 \implies 4$ par l'absurde. Soit G est connexe, $m = n-1$ et e une arête qui n'est pas un isthme. e appartient donc a un cycle et on peut la retirer du graphe en gardant la connexité. On obtient donc un graphe connexe ayant n-2 arêtes. Ceci contredit le lemme 2.3.1
Montrons que $4 \implies 1$ par l'absurde. Si il existait un cycle, alors il existerait une arête qui ne serait pas un isthme.
Montrons que $6 \implies 1$. Si G satisfait la condition 6 alors le graphe G est connexe et ne peut contenir de cycle puisque deux sommets distincts d'un cycle sont reliés par deux chaînes du cycle.
Montrons que $1 \implies 6$. Puisque G est connexe, pour toute paire de sommets x et y de G, il existe dans G au moins une chaîne d'extrémités x et y. S'il existe deux chaînes alors on aurait un cycle impossible par hypothèse.
Montrons que $5 \implies 1$. Si G satisfait la condition 5, et x et y en sont deux sommets

non adjacents, alors $G + [x, y]$ contient un cycle dont $[x, y]$ est nécessairement une arête. En supprimant l'arête $[x, y]$ de ce cycle, on obtient une chaîne de G d'extrémités x et y. Donc G est connexe et sans cycle, donc un arbre.
Montrons que $6 \Longrightarrow 5$. G est sans cycle sinon il existerait 2 chaînes entre 2 sommets du graphe. De plus l'ajout d'une arête $[a, b]$ créé avec la chaîne de B à a un cycle.

□

7.6 Arborescence

Les arborescences sont l'analogue des arbres dans le cas des graphes orientés.

Définition 7.6.1 *Une* arborescence *(X,A,r) de racine r est un graphe orienté $G = (X, A)$ ou r est un élément de X tel que pour tout sommet x il existe un unique chemin d'origine r et d'extrémité x.*

Autrement dit, $\forall x \ \exists !\ y_0, y_1, \ldots, y_p$ tels que $y_0 = r,\ y_p = x,\ \forall i,\ 0 \leq i \leq p,\ (y_i, y_{i+1}) \in A$

Définition 7.6.2 *L'entier p est appelé la* profondeur *du sommet x dans l'arborescence.*

Évidemment, dans une arborescence la racine r n'admet pas de prédécesseur, et tout sommet y différent de r admet un prédécesseur et un seul.

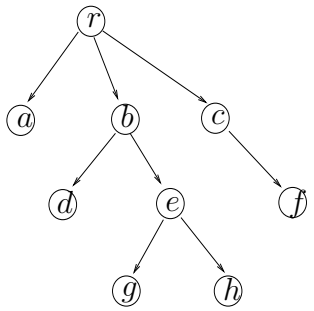


FIGURE 7.3 – Un exemple d'arborescence de racine r

Exercice : Montrer que pour une arborescence (X, A, r) on a $|A| = |X| - 1$.

Chapitre

8 Arbres couvrants de poids minimum

Contents

Résumé

Nous allons voir les conditions nécessaires et suffisantes pour l'existence d'un arbre couvrant de poids minimum. Nous proposons également deux algorithmes classiques qui résolvent ce problème.

8.1 Introduction

Les arbres couvrants jouent un rôle important dans plusieurs applications, par exemple les réseaux électriques, les réseaux routiers et réseaux de communications.

Définition 8.1.1 *Soit un graphe $G = (V, E)$. On appelle* arbre couvrant *de G tout graphe partiel de G qui est un arbre.*

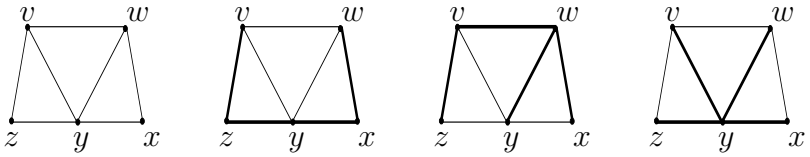


FIGURE 8.1 – 3 arbres couvrants de G

Définition 8.1.2 *Soit un graphe $G = (V, E, w)$ valué. On appelle* arbre couvrant de poids minimum (ou maximum) *de G (noté ACPM) tout arbre couvrant dont la somme des poids des arêtes le constituant, est minimal (maximal).*

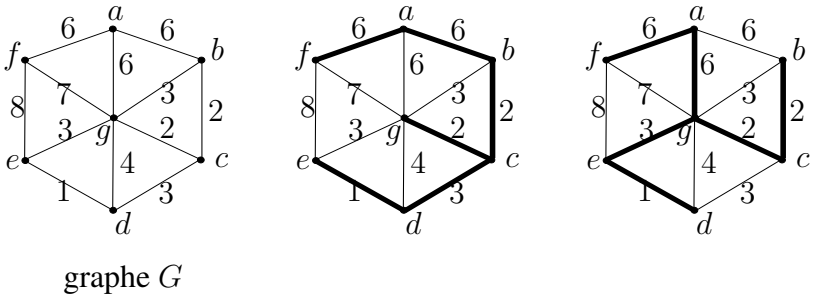


FIGURE 8.2 – 2 arbres couvrants de poids minimal (=20)

Lemme 8.1.1 *Un graphe admet un arbre couvrant ssi il est connexe.*

Preuve Soit $G = (V, E)$ un graphe d'ordre n et $|E| = m$.
$\Longrightarrow$ Si G n'était pas connexe, aucun de ses graphes partiels ne serait connexe et donc G n'admettrait pas un arbre couvrant.

$\Longleftarrow$ La démonstration est par récurrence sur m, l'ordre de G étant fixé à n. Le graphe G étant connexe, on a $m \geq n-1$ d'après le lemme 7.3.1 du chapitre 7. Si $m = n-1$, alors G est lui-même un arbre. Supposons la propriété vraie jusqu'à la taille $m-1$. Soit G connexe d'ordre n avec $m \geq n$ arêtes. Le graphe G n'est donc pas un arbre, et contient donc une arête e appartenant à un cycle. En supprimant e dans G, on obtient un graphe G' connexe, qui par hypothèse de récurrence contient un arbre couvrant qui est aussi un arbre couvrant de G. $\square$

Remarque : L'arbre couvrant de poids minimal n'est pas forcément unique.
Remarque : Si le poids de toutes les arêtes est égal à 1, le problème revient à chercher un arbre couvrant quelconque (de poids $n-1$).
Remarque : Dans un graphe complet d'ordre n le nombre d'arbres couvrants est exactement n^{n-2} (théorème de Cayley). Malgré cela, trouver l'ACPM est un problème *polynomial*.

n	1	2	3	4	5	6	7	8	9	10
nbr AC	1	1	3	16	125	1296	16807	262144	4782969	10^8

8.2 Le théorème d'optimalité pour l'ACPM

Dans cette partie, nous allons caractériser les conditions nécessaires et suffisantes d'optimalité pour un arbre couvrant de poids minimum.

Théorème 8.2.1 (Conditions pour une coupe optimale) *Un arbre couvrant T^* est un arbre couvrant de poids minimum si et seulement si il satisfait la condition suivante de la coupe optimale : $\forall [i,j] \in T^*, w_{ij} \leq w_{kl},\ \forall [k,l]$ contenu dans un coupe formé par la suppression de l'arête $[i,j]$.*

Preuve

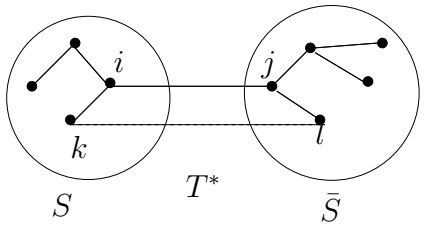


FIGURE 8.3 – *Illustration de la preuve du Théorème 8.2.1 du Théorème 8.2.3*

Il est facile de voir que pour chaque arbre couvrant de poids minimum T^* doit

satisfaire la condition de la coupe optimale. Ainsi, si $w_{ij} > w_{kl}$ et l'arête $[k, l]$ est contenu dans une coupe de la forme en supprimant l'arête $[i, j]$ de T^*, alors en introduisant l'arête $[k, l]$ dans T^* à la place de l'arête $[i, j]$ nous créons un arbre couvrant avec un coût inférieur à T^*, ce qui contredit le caractère optimale de T^*. Réciproquement, nous allons montrer que pour n'importe arbre T^* satisfaisant la condition de la coupe optimale, l'arbre T^* est un arbre couvrant de poids minimum. Supposons que T soit un arbre couvrant de poids minimum et que $T \neq T^*$. Alors T^* contient un arête $[i, j]$ qui n'est pas dans l'arbre T (voir la figure 8.3). La suppression de l'arête $[i, j]$ dans T^* crée une coupe, $[S, \bar{S}]$. Maintenant, notons que si nous ajoutons l'arête $[i, j]$ à T, nous créons un cycle W qui contient l'arête $[k, l]$ (autre arête que $[i, j]$) avec $k \in S$ et $l \in \bar{S}$, sachant que T^* satisfait la condition de la coupe optimale, $w_{ij} \leq w_{kl}$. De plus, sachant que T est un arbre couvrant de poids minimum, $w_{ij} \geq w_{kl}$, sinon on pourrait améliorer la valeur de l'arbre couvrant en remplaçant l'arête $[k, l]$ par $[i, j]$. Ainsi, nous avons $w_{ij} = w_{kl}$. Maintenant si nous introduisons l'arête $[k, l]$ dans l'arbre T^* à la place de $[i, j]$, nous produisons un autre arbre couvrant de poids minimum, admettant un arête en commun de plus avec l'arbre T. En répétant cet argument plusieurs fois, nous pouvons transformé T^* en un arbre couvrant de poids minimum T. La construction précédente montre que T^* est un arbre couvrant de poids minimum.

□

La condition de la coupe optimale implique que chaque arête dans un arbre couvrant de poids minimum est un arête de coût minimum dans une coupe défini par la suppression de cet arête dans l'arbre T. De plus, les conditions de la coupe optimale ceci implique que nous pouvons toujours inclure tout autre arête de coût minimum dans n'importe quel arbre couvrant de poids minimum.

Théorème 8.2.2 *Soit F un sous-ensemble d'arêtes d'un arbre couvrant de poids minimum et soit S l'ensemble des sommets d'une des composantes de F. Supposons que $[i, j]$ est un arête de coût minimum dans la coupe $[S, \bar{S}]$. Alors, parmi tous les arbres couvrant de G incluant la forêt F il en existe un, qui contient l'arête $[i, j]$.*

Preuve
Supposons que F est une forêt d'un arbre couvrant de poids minimum T^*. Si $[i, j] \in T^*$, nous n'avons rien à prouver. Sinon, ajoutons $[i, j]$ à l'arbre T^*, nous créons ainsi un cycle C, et C contient au moins un arête $[p, q] \neq [i, j]$ avec $[p, q] \in [S, \bar{S}]$. Par hypothèse, nous avons $w_{ij} \leq w_{pq}$. Sachant que T^* satisfait la condition de la coupe minimale, $w_{ij} \geq w_{pq}$. Par conséquent, nous avons $w_{ij} = w_{pq}$, ainsi en

ajoutant l'arête $[i, j]$ à T^* et en enlevant l'arête $[p, q]$ produit un arbre couvrant de poids minimum contenant tous les arêtes de F du type $[i, j]$. □

La condition de la coupe optimale, nous propose une caractérisation extérieure d'un arbre couvrant de poids minimum, impliquant une relation entre un arête unique de l'arbre avec des arêtes à l'extérieur de l'arbre. Le théorème suivant, relatif à la condition du chemin optimale, produit une caractérisation interne, entre un arête de ne faisant pas partie de l'arbre et plusieurs arêtes de l'arbre.

Théorème 8.2.3 (Conditions du chemin optimal) *Un arbre couvrant T^* est un arbre couvrant de poids minimum si et seulement si il satisfait la condition suivante du chemin optimal : pour chaque arête $[k, l]$ de G non contenu dans T^*, nous avons $w_{ij} \leq w_{kl}$, $\forall [i, j]$ contenu dans un chemin dans T^* connectant les sommets k et l.*

Preuve
Il est facile de voir de la nécessité de la condition du chemin optimal. Supposons qu'un arbre T^* est minimal satisfaisant les conditions et l'arête $[i, j]$ est contenu dans le chemin de T^* connectant les sommets k et l. Si $w_{ij} > w_{kl}$, en introduisant l'arête $[k, l]$ dans T^* à la place de $[i, j]$ nous créons un arbre avec un coût plus faible que celui de T^*, impossible. Nous allons établir la condition suffisante sur la condition du chemin optimal en utilisant la condition de la coupe optimale. La preuve met en avant l'équivalence entre ces deux conditions. Nous allons montrer que si T^* satisfait la condition du chemin optimal, il satisfait également la conditon de la coupe optimale. Le théorème 8.2.1 implique que T^* est optimal. Soit $[i, j]$ un arête quelqconque de T^*, et soit S et $\bar{S}$ les deux ensembles de sommets produit par la suppression de l'arête $[i, j]$ de T^*. Supposons que $i \in S$ et que $j \in \bar{S}$. Considérons l'arête $[k, l] \in [S, \bar{S}]$ (voir la figure 8.3). Sachant que T^* contient un unique chemin joignant les sommets k et l et sachant que l'arête $[i, j]$ est le seul arête de T^* joignant un sommet de S avec un sommet de $\bar{S}$, l'arête $[i, j]$ doit appartenir à ce chemin. La condition du chemin optimal implique que $w_{ij} \leq w_{kl}$, sachant que cette condition doit être valide pour n'importe quel arête $[k, l]$, faisant partie de la coupe $[S, \bar{S}]$, et $[k, l]$ ne faisant pas partie de l'arbre, formé par la suppression de n'importe quel arête $[i, j]$, T^* satisfait la condition de coupe optimale et il doit être minimum. □

8.3 L'algorithme de Prim

8.3.1 Principe

Nous partons d'un arbre initial T réduit à un seul sommet. Ensuite, à chaque itération, nous augmentons l'arbre T en le connectant au "plus proche" sommet libre au sens des poids.

8.3.2 L'algorithme

Algorithme 8.1 L'algorithme de Prim

$G = (V, E, w)$ avec $|V| = n$ et $T = \{i\}$ avec $i \in V$ et $E' = \varnothing$
while $|E'| \neq n$ **do**
  Choisir $[i, j] \notin E' \backslash w([i, j]) = \min_{\forall i \in T, \forall k \in G \backslash T} w([i, k])$
  $T = T \cup \{i\}$, et $E' = E' \cup \{[i, j]\}$
end while
Retourner T

8.3.3 Exemples

Exemple 1 :

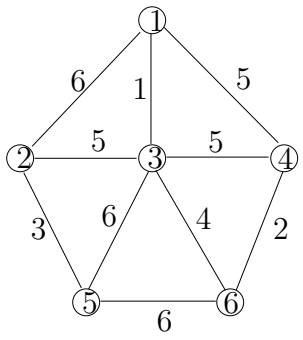


FIGURE 8.4 – *Graphe initial G*

Exemple 2 :

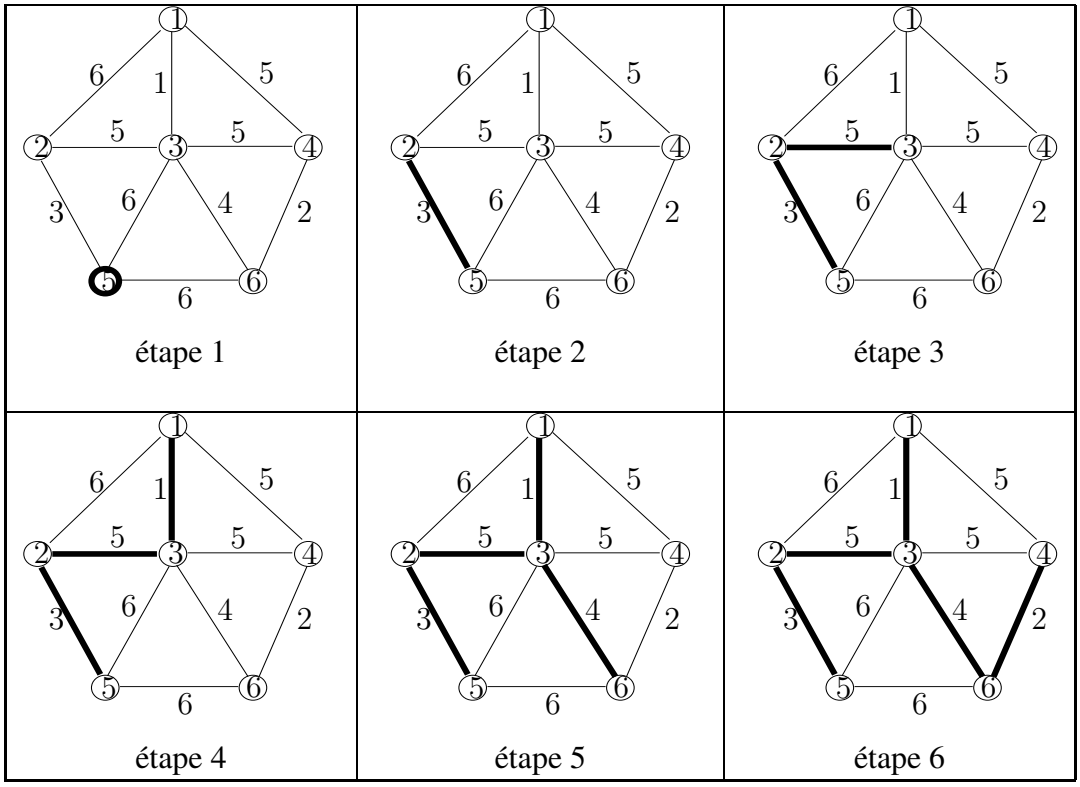


FIGURE 8.5 – L'algorithme de Prim

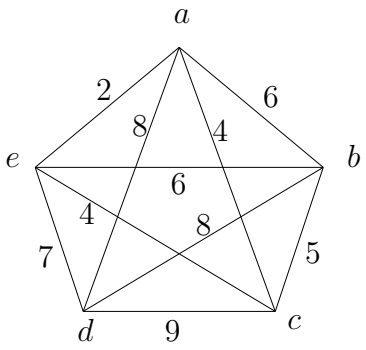


FIGURE 8.6 – *graphe initial*

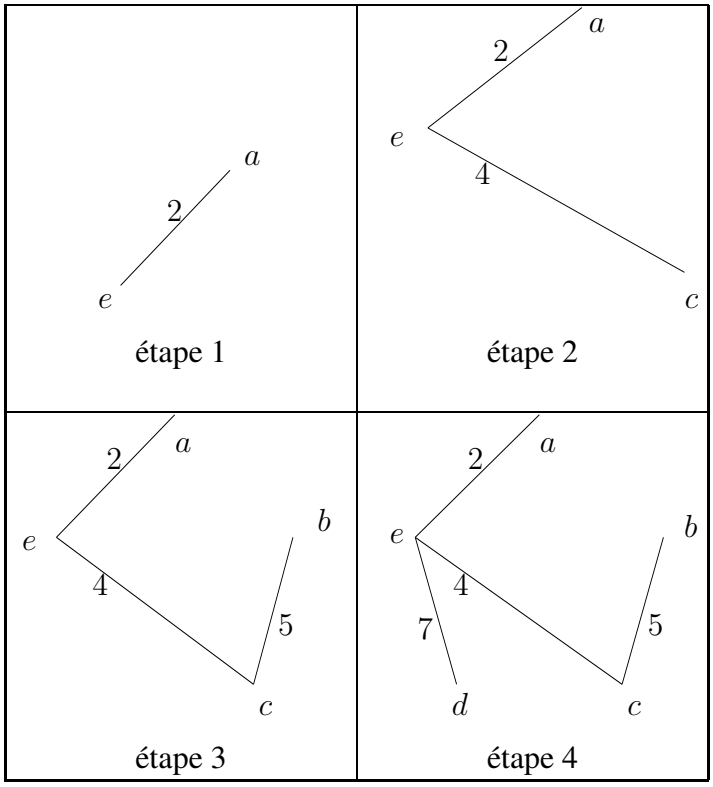


FIGURE 8.7 – L'algorithme de Prim

8.3.4 Remarques

Remarque : Si le graphe G est connexe, alors on obtient l'ACPM en $n-1$ itérations.
Remarque : Si pendant une itération $i < n-1$ on ne trouve pas d'arête $[x, y]$ avec x dans l'arbre et y n'appartenant pas à l'arbre, alors le graphe G n'est pas connexe. L'algorithme peut donc servir comme test de connexité, mais notons que l'on dispose pour cela d'algorithmes plus efficaces en $\mathcal{O}(m)$ (voir la partie sur l'exploration de graphes).
Remarque : Si le graphe n'est pas connexe, on peut obtenir en exécutant l'algorithme sur chaque composante connexe, une forêt couvrante de poids minimal.
Remarque : Le théorème de 8.2.1 garantit l'optimalité de l'algorithme de Prim. En effet, l'algorithme construit un arbre couvrant en ajoutant à chaque étape un sommet et une arête. Les sommets de l'arbre en construction sont contenus dans un ensemble S et nous cherchons un plus proche voisin à un sommet de S. Donc,

nous cherchons un sommet j dans $\bar{S}$ tel que $[i, j]$ soit minimal, avec $i \in S$ dans la coupe $[S, \bar{S}]$. Nous incluons j dans S et nous répétons la procédure jusqu'à que $S = N$. La correction de l'algorithme provient du théorème 8.2.2 sachant que le résultat implique que chaque arête que nous ajoutons dans l'arbre est contenu dans un arbre couvrant de poids minimum constitué des arêtes que nous avons sélectionné précédemment.

8.3.5 Complexité et mise en oeuvre

Lemme 8.3.1 *Avec une implantation naïve, l'algorithme de Prim est en* $\mathcal{O}(mn)$.

Preuve Soit $v_1, v_2, \ldots, v_n$ la suite de sommets qui sont inclus, au fur et à mesure, dans l'arbre couvrant pendant l'exécution de l'algorithme de Prim. Soit $d_1, d_2, \ldots, d_n$ les degrés de $v_1, v_2, \ldots, v_n$. à la première itération, on choisit pour faire entrer dans l'ACPM le plus proche sommet de v_1, en d_1 itérations. A la deuxième itération, on choisit le plus proche sommet de v_1 et de v_2, en $d_1 + d_2$ opérations. A la i-ième itération ($i \leq n-1$), on choisit le plus proche sommet en $\sum_{k=1}^{i} d_k$ opérations. On a donc $d_1+(d_1+d_2)+\ldots+(\sum_{k=1}^{i} d_k)+\ldots$ opérations, ce qui donne au plus $\mathcal{O}(n \sum_{i=1}^{n-1} d_i) = \mathcal{O}(mn)$ opérations, avec m le nombre d'arêtes du graphe. □

Remarque : Si on considère un graphe dense, on a $m = n^2$ et l'algorithme de Prim est en $\mathcal{O}(n^3)$.
Il existe une implantation de l'algorithme de Prim plus efficace ayant une complexité en $\mathcal{O}(n^2)$. On utilise pour cela deux tableaux, de dimension n. Le premier, appelé `plus_proche`, contient pour chaque sommet hors de l'arbre, son plus proche voisin dans l'arbre. On a

$$plus\_proche[x] = \begin{cases} x \text{ si } x \text{ est déjà dans l'arbre} \\ 0 \text{ si } [x, y] \notin E, \forall y \text{ sommet dans l'arbre} \\ y \text{ si } y \text{ est le plus proche voisin de } x \text{ dans l'arbre} \end{cases}$$

Le deuxième tableau sert à mémoriser le poids du plus proche voisin. Pour un sommet x hors de l'arbre, on définit le tableau `poids_plus_proche` de la manière suivante :

$$poids\_plus\_proche[x] = \begin{cases} w_{xy} \text{ si } x \texttt{ plus\_proche[x]=y} \\ +\infty \text{ si } \texttt{plus\_proche[x]=0} \end{cases}$$

à chaque étape on examine les sommets qui ne font pas partie de l'arbre pour choisir le sommet qui va entrer dans l'arbre. Le tableau `poids_plus_proche`

fournit le poids le plus faible, et le tableau `plus_proche` fournit le sommet dans l'arbre avec lequel sera lié le nouveau sommet hors de l'arbre. Le futur sommet est donc choisi en $\mathcal{O}(n)$ opérations. Évidemment, une fois le futur sommet entré dans l'arbre, il faut mettre à jour seulement les valeurs des deux tableaux de ses voisins. Ceci demande d_x opérations, où x est le sommet entrant et d_x son degré (voir la figure 8.8).

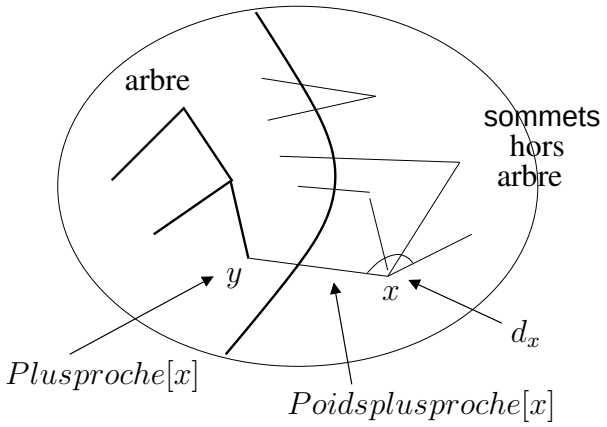


FIGURE 8.8 – *Illustration de la preuve du Lemme 8.3.1*

L'algorithme donc à chaque itération nécessite $\mathcal{O}(n + d_i)$ opérations, où d_i est le degré du sommet entrant dans l'arbre à l'itération i. Puisque l'algorithme demande $n-1$ itérations, la complexité est en $\mathcal{O}(n^2)$. On peut faire en $\mathcal{O}(m+n\log n)$ avec un tas.

8.4 Algorithme de Kruskal

La condition du chemin optimal suggère immédiatement l'algorithme suivant pour résoudre le problème de l'arbre couvrant de poids minimum. Nous commençons par n'importe quel arbre couvrant T, et nous testons la condition du chemin optimal. Si T satisfait cette condition, T est un ACPM, sinon $w_{ij} > w_{pq}$ pour une arête $[k, l]$ n'appartenant à l'arbre et pour une arête $[i, j]$ appartenant à l'arbre et étant sur l'unique chemin dans T reliant les sommets k et l. Dans ce cas, rajoutons l'arête $[k, l]$ et nous enlevons l'arête $[i, j]$ et nous obtenons un arbre de poids plus faible. Nous répétons cette procédure un nombre fini d'itérations. Bien que cet algorithme soit simple, la complexité ne peut être bornée par la taille de l'instance.

8.4.1 Principe

Nous partons d'une forêt de n arbres, chaque arbre étant réduit à un sommet isolé. Puis, à chaque itération, il faut ajouter la plus petite arête ne créant pas de cycle avec les arêtes déjà choisies. Il suffit d'arrêter quand l'arbre est couvrant (le graphe G est connexe) ou quand on ne trouve plus d'arêtes à ajouter (le graphe G est non connexe).

8.4.2 L'algorithme

Algorithme 8.2 L'algorithme de Kruskal

Soit $G = (V, E), T = \{1, \ldots, n\}$ et $E' = \varnothing$
while $|E'| \neq n$ **do**
Choisir $[i, j] \in E \backslash E'$ tel que $w_{ij} = \min_{[k,l] \in E \backslash E'} w_{kl} \wedge E' \cup \{[i, j]\}$ ne forme pas un cycle.
end while

8.4.3 Exemple

Soit le graphe $G = (V, E)$ suivant :

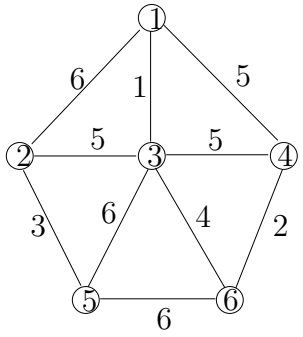


FIGURE 8.9 – *Exemple pour l'algorithme de Kruskal*

On ordonne les arêtes en ordre croissant :
$[1, 3]$ poids 1 $[3, 4]$ poids 5

$[4,6]$ poids 2 $[2,3]$ poids 5
$[2,5]$ poids 3 $[1,2]$ poids 6
$[3,6]$ poids 4 $[3,5]$ poids 6
$[1,4]$ poids 5 $[5,6]$ poids 6

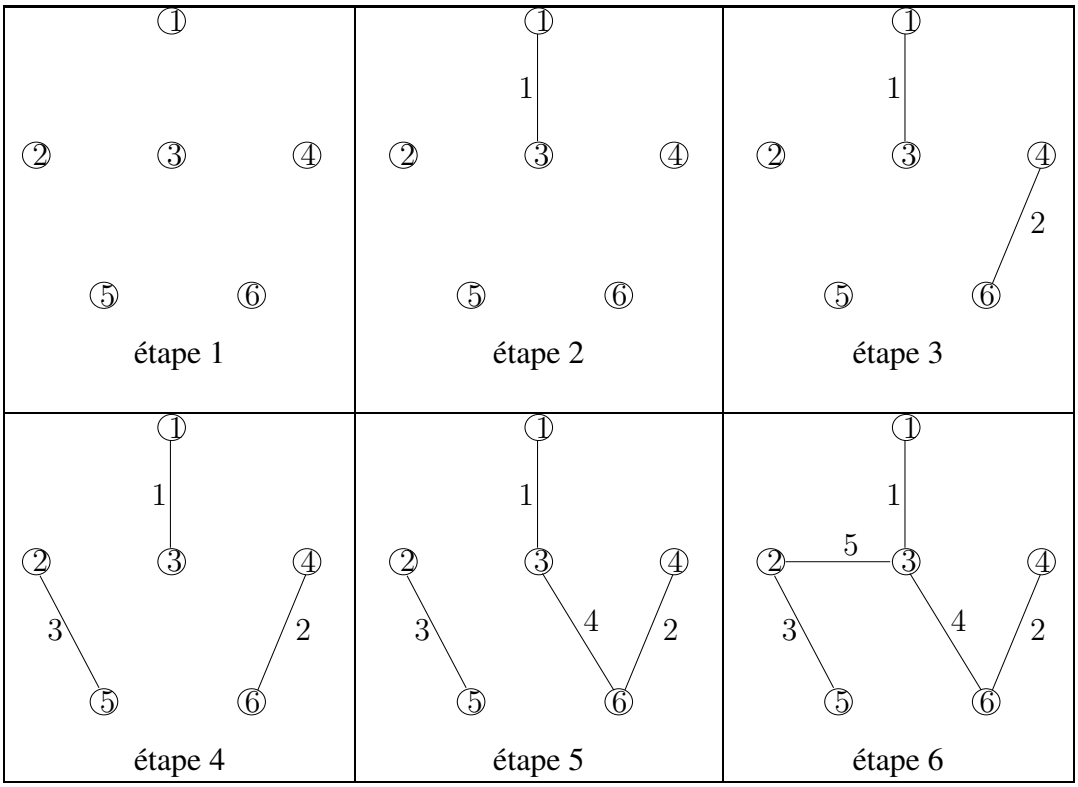


FIGURE 8.10 – L'algorithme de Kruskal

8.4.4 Remarque

Lorsque le graphe est non connexe, pour trouver une forêt couvrante par l'algorithme de Prim il faut lancer l'algorithme sur chaque composante connexe. L'algorithme de Kruskal, par contre, trouve directement une forêt couvrante de poids minimal.
La correction de l'algorithme de Kruskal vient du fait que nous refusons d'inclure une arête dans l'arbre à une itération parce que l'inclusion de cette arête créerait

un cycle avec les arêtes déjà choisies. En observant que le coût de l'arête $[k, l]$ est supérieur ou égal au coût de n'importe quelle arête dans le cycle car nous avons examiné les arêtes par ordre décroissant en terme de coût, alors l'arbre satisfait la condition du chemin optimal (voir théorème 8.2.3).

8.4.5 Complexité et mise en oeuvre

8.4.5.1 Mise en oeuvre de l'algorithme

Avant de choisir les structures de données nécessaires à l'implémentation, nous devons d'abord résoudre le problème suivant : comment déterminer si une arête choisie à l'étape i forme un cycle avec les arêtes précédemment choisies ?
Une solution pour résoudre ce problème est la suivante : pour qu'une arête $[x, y]$ ferme un cycle, il faut que précédémment ses extrémités aient été reliées par une chaîne, et donc aient été dans une même composante connexe. Nous allons donc gérer l'évolution des composantes connexes, au fur et à mesure du choix des arêtes. Initialement, lorsque le graphe partiel ne contient aucune arête, chaque sommet constitue une composante connexe et nous initialisons, pour chaque sommet i, un indice à cette valeur i. Chaque fois qu'une arête $[x, y]$ est candidate on compare les valeurs des indices de x et de y. Si elles sont égales, c'est que les deux sommets sont déjà dans la même composante, l'arête $[x, y]$ créerait donc un cycle et par conséquent on ne la retient pas ; sinon on garde $[x, y]$ et on donne comme valeur à l'indice associé à y celle de l'indice associé à x, ainsi qu'à tout sommet qui avait auparavant le même indice que y (on peut bien sûr intervertir ici le rôle de x et celui de y), ceci voulant dire que tous les sommets de la composante connexe de x et de la composante connexe de y, après le choix de l'arête $[x, y]$, forment désormais une seule composante et portent donc tous le même indice de composante connexe.
Une structure de données raisonnable semble être un tableau (appelé CC dans la suite) d'entiers associés aux sommets numérotés de 1 à n et qui tiendra à jour les indices des composantes connexes associés à ces sommets. Par ailleurs, nous représentons le graphe, donnée du problème, par un tableau A contenant les arêtes, rangées par ordre de poids croissants. Enfin le résultat, c'est à dire l'arbre couvrant de poids minimum, sera représenté par le tableau T de ses $n - 1$ arêtes. L'algorithme de kruskal peut alors être décrit de la façon suivante. Voici le nouvel algorithme de Kruskal avec la prise en compte des remarques précédentes :

Algorithme 8.3 L'algorithme de Kruskal

Trier les arêtes par poids croissants et les ranger dans A selon cet ordre
for $i = 1$ à $n - 1$ **do**
  $CC(i) := i$
end for
$compteurT := 0$
$compteurA := 1$
while $compteurT < n - 1$ **do**
  Soit $[x, y]$ l'arête de A
  $compteurA := compteurA + 1$
  if $CC(x) \neq CC(y)$ **then**
    $compteurT := compteurT + 1$
    $T(compteurT) := [x, y]$
    $auxiliaire := CC(y)$
    for $i = 1$ à n **do**
      if $CC(i) := auxiliaire$ **then**
        $CC(i) =: CC(x)$
      end if
    end for
  end if
end while

8.4.5.2 Complexité

Théorème 8.4.1 *L'algorithme de Kruskal trouve un ACPM en temps* $\mathcal{O}(n^2 \log_2 n)$.

Preuve
Nous devons commencer par trier les arêtes du graphes par poids croissants, ce qui nécessite $\mathcal{O}(m \log_2 m)$ opérations élémentaires, où m est la taille du graphe. Les autres initialisations peuvent se faire à l'aide de $\mathcal{O}(n)$ opérations élémentaires.
Chaque fois qu'on examine une arête candidate, on doit comparer les indicces de composantes connexe de ses extrémités. En cas d'égalité, on la refuse ; on fait alors $\mathcal{O}(1)$ opérations élémentaires par arête rejetée. Au contraire, si les numéros sont différents, on adopte l'arête ; on doit alors mettre à jour les indices des sommets et pour cela effectuer $\mathcal{O}(n)$ opérations élémentaires par arête retenue : or, nous devons retenir $n-1$ arêtes retenues. On doit donc effectuer $\mathcal{O}(n^2)$ opérations élémentaires pour les $n-1$ arêtes retenues et au plus $\mathcal{O}(m)$ pour les arêtes rejetées, soit au total $\mathcal{O}(n^2)$ opérations pour cette partie.
On voit donc, en tenant compte du tri initial, que la complexité de l'algorithme 8.3 est en $\mathcal{O}(n^2 + m \log_2 m)$, ou encore en $\mathcal{O}(n^2 \log_2 n)$. si l'on considère des graphes pour lesquels m est de l'ordre de n^2. □

Exercice : Soit G un graphe connexe. Montrer qu'il admet soit 1, soit k arbres couvrants, avec $k \geq 3$.

Chapitre

9 Problèmes du plus court chemin dans un graphe

Contents

Résumé

Nous allons voir quelques résultats concernant les problèmes du plus court chemin dans un graphe orienté et valué. Nous verrons plusieurs algorithmes classiques.

9.1 Motivations

Les problèmes de plus courts chemins se rencontrent dans de nombreuses applications et souvent comme sous-problème de problèmes plus complexes.
Par exemple, supposons que nous voulions acheminer le plus rapidement possible des marchandises depuis une ville notée 1, jusqu'à des villes notées 2,3,4,5,6 et 7. On dispose pour cela d'une flotte de 6 camions. Le problème est modélisé par le graphe orienté de la figure 9.1. Le coût d'un arc (i, j) représente le temps en heures nécessaire pour relier la ville i à la ville j.
Une solution possible est représentée à la figure 9.2 :

— le camion 1 suit le chemin (1,2) et livre la ville 2 en 16h,

— le camion 2 suit le chemin (1,2,5) et livre la ville 5 en 41h,

— le camion 3 suit le chemin (1,4) et livre la ville 4 en 35h,

— le camion 4 suit le chemin (1,4,6) et livre la ville 6 en 52h,

— le camion 5 suit le chemin (1,4,6,7) et livre la ville 7 en 66h,

— le camion 6 suit le chemin (1,3) et livre la ville 3 en 9h.

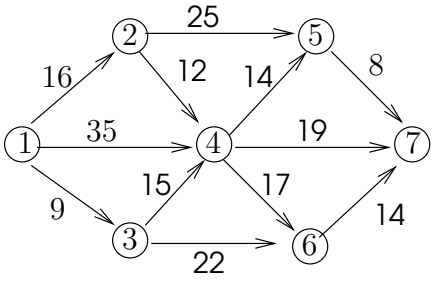


FIGURE 9.1 – *Exemple de graphe*

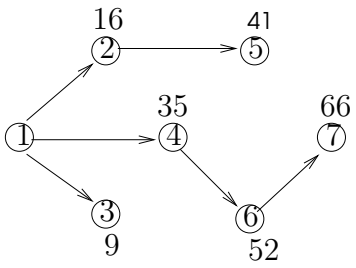


FIGURE 9.2 – *Une solution non optimale*

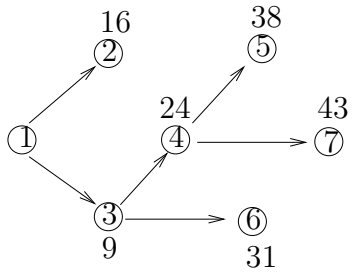


FIGURE 9.3 – *Une solution optimale*

En fait on peut faire beaucoup mieux et la solution représentée à la figure 9.2 est la meilleure possible. On peut remarquer que le graphe de la figure 9.3 est une arborescence de racine 1. Comme nous le verrons par la suite il ne s'agit pas d'un hasard.

9.2 Le problème des plus courts chemins

Soit $G = (X, A, w)$ un graphe orienté avec des coûts (poids) sur ses arcs. On note w_{ij} le coût de l'arc (i, j).

Définition 9.2.1 *Le* coût *ou* longueur *d'un chemin est la somme du coût de ses arcs.*

Définition 9.2.2 *Un chemin est dit* élémentaire *s'il ne passe pas deux fois par un même sommet.*

Les chemins seront toujours supposés élémentaires.
Trois problèmes :

1. Étant donné deux sommets s et t, trouver un plus court chemin (chemin de coût minimum) de s à t.
2. Étant donné un sommet de départ s, trouver un plus court chemin de s vers tout autre sommet.
3. Trouver un plus court chemin entre tout couple de sommets.

Evidemment un algorithme pour le problème 1 peut servir à résoudre les problèmes 2 et 3. De même un algorithme pour le problème 2 peut servir à résoudre le problème 3, et un algorithme pour le problème 2 peut servir à résoudre le problème 1.
Dans la suite nous nous intéressons au problème 2. Il existe deux familles d'algorithmes pour résoudre ce problème : les algorithmes à correction d'étiquettes et les algorithmes à fixation d'étiquettes. Avant de les étudier nous avons besoin de quelques préliminaires.

9.3 L'algorithme à fixation d'étiquettes

9.3.1 L'algorithme de Dijkstra

On suppose que tous les coûts des arcs sont positifs.
Nous notons $\overline{S} = X \setminus S$.
Nous allons prouver dans un premier temps que l'algorithme de Dijkstra est correct, et nous cherchons dans un deuxième temps à déterminer sa complexité.

Théorème 9.3.1 *L'algorithme de Dijkstra est correct c'est-à-dire qu'il détermine un plus court chemin à partir d'un sommet s fixé.*

Preuve
Nous allons montrer l'invariant suivant :
$\forall u \in S, d(u)$ est un plus court chemin entre s et u. Nous procéder par induction sur $|S|$.

— $|S| = 1$, donc $d(s) = d(s, s) = 0$
— Hypothèse $|S| = k \geq 1$

Algorithme 9.1 L'algorithme de Dijkstra

$d(s) := 0$; $pred(s) := 0$
$d(i) := +\infty$; pour tout $i \neq s$
$S = \emptyset$
while $S \neq X$ **do**
  choisir $i \in \overline{S}$ avec $d(i)$ minimum
  $S = S \cup \{i\}$
  for chaque arc (i, j) **do**
    if $d(j) > d(i) + w_{ij}$ **then**
      $d(j) := d(i) + w_{ij}$
    end if
  end for
end while

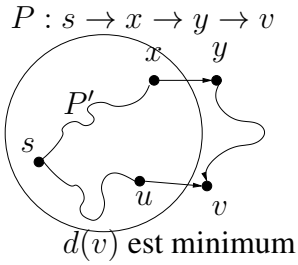


FIGURE 9.4 – *Illustration de la preuve de la correction de l'algorithme Dijkstra*

Soit v le prochain sommet ajouté à S et soit l'arête (u, v) qui est choisi (voir la figure 9.4).
Un plus court chemin $s-u+uv$, de valeur $d(u)+w_{uv}$ un plus court chemin de longueur $d(v)$.
Considérons un autre chemin $s - v$ noté P. Nous allons montrer que la longueur de P ne peut pas être plus petite que $d(v)$:

— Soit y le premier sommet sur le chemin $s-v$ non encore inclus dans S. Soit x son prédécesseur qui appartient à S et P' le sous-chemin entre $s - x$.
En fait P est déjà trop long : $l(P) \geq l(P') + w_{xy} \geq d(x) + w_{xy} \geq d(y) \geq d(v)$

Donc $d(v)$ est un plus court chemin. □

Théorème 9.3.2 *L'algorithme de Dijkstra résout le problème des plus courts chemins en temps $\mathcal{O}(n^2)$.*

Preuve À la i-ième étape on sélectionne le sommet de $\overline{S}$ ayant la plus petite distance. La taille de $\overline{S}$ est alors égale à $n-i$, et cela nécessite donc $n-i$ opérations. À la i-ième étape on corrige également (éventuellement) tous les successeurs du sommet x_i choisi. Cela nécessite $|\Gamma^+(x_i)|$ opérations. On a donc au total $\sum_{i=0}^{n-1}(n-i+|\Gamma^+(x_i)|) = \sum_{i=0}^{n-1}(n-i)+\sum_{i=0}^{n-1}|\Gamma^+(x_i)| = \frac{n(n-1)}{2}+m = \mathcal{O}(n^2)$ opérations. □

Le fonctionnement de l'algorithme de Dijkstra sur le graphe de la figure 9.1 est illustré à la figure 9.5. Une étoile '*' à coté d'un sommet indique qu'il fait partie de l'ensemble S.

9.4 Programmation dynamique

Si $\mu = x_0x_1\dots x_k$ avec $x_0 = s$ et $x_k = t$ est un plus court chemin de s à t, alors $x_0x_1\dots x_{k-1}$ est un plus court chemin de s à x_{k-1}.

$$d^k(i) = \min\{d^{k-1}(i), \min_{v\in\Gamma^-(i)}\{d^{k-1}(v)+w_{vi}\}\}$$

Le tableau suivant donne les valeurs $d^k(i)$ pour le graphe de la figure 9.1.

	1	2	3	4	5	6	7
0	(1,0)	(.,∞)	(.,∞)	(.,∞)	(.,∞)	(.,∞)	(.,∞)
1	(1,0)	(1,16)	(1,9)	(1,35)	(.,∞)	(.,∞)	(.,∞)
2	(1,0)	(1,16)	(1,9)	(3,24)	(2,41)	(3,31)	(4,54)
3	(1,0)	(1,16)	(1,9)	(3,24)	(4,38)	(3,31)	(4,43)
4	"	"	"	"	"	"	"

9.5 Problème du plus court chemin entre toutes paires de sommets

9.5.1 Introduction et hypothèse

Nous supposerons dans la suite que les valuations des arcs peuvent positives ou négatives mais de contient pas de cycle de coût strictement négatif.

Définition 9.5.1 *Une marche dans un graphe orienté $G = (N, A)$ est un sous-graphe de G consistant en une séquence de sommets et d'arcs $i_1 - a_1 - i_2 - a_2 - \ldots - i_{r-1} - a_{r-1} - i_r$ satisfaisant la propriété suivante : $\forall 1 \leq k \leq r-1$ soit $a_k = (i_k, i_{k+1}) \in A$ ou $a_k = (i_{k+1}, i_k) \in A$.*

Définition 9.5.2 *Une marche directe est la version orientée de la marche dans le sens que deux sommets consécutifs i_k et i_{k+1} de la marche $(i_k, i_{k+1}) \in A$.*

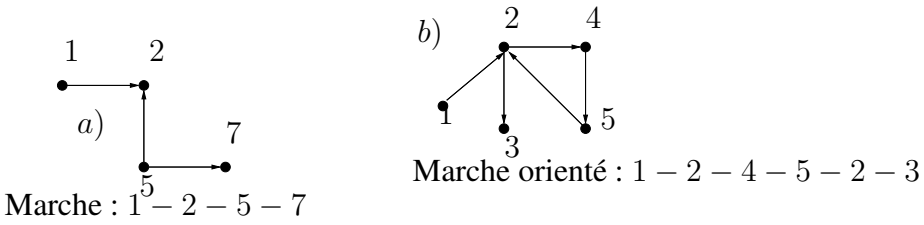


FIGURE 9.6 – *Exemple de deux marches.*

9.5.2 Conditions d'optimalité

Soit $[i, j]$ désigne la paire des sommets i et j. L'algorithme 9.2 maintient un label $d[i, j]$ pour chaque paire de sommets ; les labels représentent la longueur d'une marche entre i et j et donc une borne supérieure d'un plus court chemin entre i et j.

Théorème 9.5.1 *Soit une paire $[i, j] \in V \times V$, et soit $d[i, j]$ représente la longueur d'un chemin orienté d'un sommet i vers un sommet j Ces distances représentent les distances entre chaque paires de sommets si et si seulement elles satisfont les conditions d'optimalité entre chaque paires de sommets*

$$d[i, j] \leq d[i, k] + d[k, j], \forall i, j, k$$

Preuve
Procédons par l'absurde pour établir que le plus court chemin avec pour longueur $d[i, j]$ doit satisfaire la condition du Théorème 9.5.1.
Supposons que $d[i, k] + d[k, j] < d[i, j], i, j, k$. L'union d'un plus court chemin entre les sommets i et k et entre les sommets k et j est une marche de longueur $d[i, k] + d[k, j]$ des sommets i à j. La marche se décompose en un chemin, noté P du sommet i à j et de plusieurs cycles. Sachant que chaque cycle

dans le graphe admet une longueur positive. La longueur du chemin P est au plus $d[i,k] + d[k,j] < d[i,j]$, en contradiction avec l'optimalité de $d[i,j]$.
Maintenant, nous allons montrer que si les labels $d[i,j]$ satisfait la conditoin du Théorème, les labels représentent lezs plus courts chemins. Soit P un chemin, noté $i = i_1 - i_2 - i_3 - \ldots - i_k = j$, de longueur $d[i,j]$. La condition du Théorème implique

$$\begin{aligned} d[i,j] = d[i_1,i_k] &\leq d[i_1,i_2] + d[i_2,i_k] \leq c_{i_1 i_2} + d[i_2,i_k] \\ d[i_2,i_k] &\leq c_{i_2 i_3} + d[i_3,i_k] \\ \vdots &\leq \vdots \\ d[i_{k-1},i_k] &\leq c_{i_{k-1} i_k} \end{aligned}$$

Ces inégalités impliquent

$$d[i,j] \leq c_{i_1 i_2} + c_{i_2 i_3} + \ldots + c_{i_{k-1} i_k} = \sum_{(i,j) \in P} c_{ij}$$

Alors $d[i,j]$ est une borne inférieure sur la longueur d'un chemin d'un sommet i à j. Par hypothèse $d[i,j]$ est également une borne supérieure sur un plus court chemin entre i et j. Par conséquent $d[i,j]$ doit être un plus court chemin entre ces sommets.

□

9.5.3 Algorithme générique entre toutes paires de sommets avec corrections de labels

Utilisant la condition donnée par le Théorème 9.5.1, nous partons du label $d[i,j]$, et nous procédrons à une mise à jour successive jusqu'à que $d[i,j]$ satisfasse les conditions d'optimalité.

Lemme 9.5.1 *L'algorithme 9.2 admet une complexité en $O(n^3 C)$ avec $C = \max_{(i,j) \in E} |c_{ij}|$ si G ne contient pas de cycle de coût négatif.*

Preuve
Supposons que G est sans cycle de coût négatif et que les coûts sont entiers. A chaque étape, l'algorithme maintient la propriété de l'invariant, quand $d[i,j] < \infty$

Algorithme 9.2 Algorithme générique entre toutes paires de sommets avec corrections de labels

for $[i, j] \in V \times V$ **do**
  $d[i, j] = \infty$;
end for
for $i \in V$ **do**
  $d[i, i] = 0$;
end for
for $(i, j) \in E$ **do**
  $d[i, j] = c_{ij}$;
end for
while Le graphe contient trois sommets i, j et k tel que $d[i, j] > d[i, k] + d[k, j]$ **do**
  $d[i, j] = d[i, k] + d[k, j]$
end while

alors le graphe contient une marche de longueur $d[i, j]$ entre les sommets i et j. On peut utiliser le principe d'induction sur le nombre d'itérations pour montrer que la propriété est vraie. Maintenant, nous considérons une marche de longueur $d[i, j]$ entre les sommets i et j quand l'algorithme se termine. Cette marche peut-être décomposée entre un chemin, appelé P de i à j, et des cycles possibles. Aucun de ces cycles ne peuvent avoir des longueurs positives, sinon nous aurions de contradiction sur l'optimalité de $d[i, j]$. Alors, tous les cycles admettent des longueurs nulles. Par conséquent, le chemin P doit avoir une longueur $d[i, j]$. Les labels $d[i, j]$ satisfont les conditions d'optimalités, les conditions garantissent le critère de terminaison de l'algorithme.
Cette conclusion établit le fait que quand l'algortihme se termine les labels représentent les distances les plus courtes.
Maintenant regardons la complexité de l'algorithme : sachant que les coûts sont entiers et que C est le coût maximum, ainsi la borne inférieure (resp. borne supérieure) est $-nC$ (resp. nc). A chaque itération, certains labels $d[i, j]$ décroit. Par conséquent, l'algorithme se termine en $O(n^3C)$ itérations.

□

9.5.4 L'algorithme Floyd-Warshall

Définition 9.5.3 *Soit $d^k[i,j]$ la longueur d'un plus court chemin entre un sommet i et j tel que il utilise que les sommets $1, 2, \ldots, k-1$ comme sommet interne.*

Clairement $d^{n+1}[i,j]$ représente un plus court chemin entre les sommets i et j. L'algorithme de Floyd-Warshall calcule en premier $d^1[i,j], \forall i,j$, calcule $d^2[i,j]$ et ainsi de suite jusqu'à $d^{n+1}[i,j]$.

Lemme 9.5.2 *Nous avons*

$$d^{k+1}[i,j] = \min\{d^k[i,j], d^k[i,k] + d^k[k,i]\}$$

.

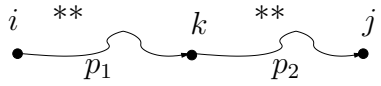
Preuve
Ce lemme est valide pour la raison suivante : un plus court chemin utilise seulement les sommets $1,\ 2, \ldots, k$ comme sommets internes soit

- ne traverse pas le sommet k alors $d^{k+1}[i,j] = d^k[i,j]$ ou,
- traverse le sommet k, alors $d^{k+1}[i,j] = d^k[i,k] + d^k[k,i]$. On a deux chemins p_1 (entre i et k) et p_2 (entre k et j).

Alors

$$d^{k+1}[i,j] = \min\{d^k[i,j], d^k[i,k] + d^k[k,i]\}$$

.



p : tous les sommets internes dans $\{1.2, \ldots, k\}$.
$**$tous les sommets internes dans $\{1.2, \ldots, k-1\}$.

FIGURE 9.7 – *Illustration pour la preuve du Lemme 9.5.2.*

□

L'algorithme donne également la liste des prédécesseurs.

Théorème 9.5.2 *L'algorithm Floyd-Warshall résoud le problème du plus court chemin entre chaque paires de sommets en $0(n^3)$.*

Algorithme 9.3 Algorithme Floyd-Warshall

for $[i, j] \in V \times V$ **do**
  $d[i, j] = \infty$;
  $pred[i, j] = 0$;
end for
for $i \in V$ **do**
  $d[i, i] = 0$;
end for
for $(i, j) \in E$ **do**
  $d[i, j] = c_{ij}$;
  $pred[i, j] = i$;
end for
for $k = 1$ à n **do**
  for $[i, j] \in V \times V$ **do**
    if $d[i, j] > d[i, k] + d[k, j]$ **then**
      $d[i, j] = d[i, k] + d[k, j]$
      $pred[i, j] = pred[k, j]$;
    end if
  end for
end for

Preuve
D'après l'algorithme 9.3, la complexité est en $0(n^3)$.

□

9.5.5 Détection de circuits négatifs

Théorème 9.5.3 *Si durant les itérations la valeur du label $d[i,j]$ admet au moins une des deux valeurs ci-dessous alors le graphe G admet un cycle négatif.*

1. *Si $i = j$, tester si $d[i,i] < 0$*
2. *Si $i \neq j$, tester si $d[i,j] < -nC$*

Preuve

1. Considérons la première itération pour laquelle $d[i,i] < 0$. A cet instant $d[i,i] = d[i;k] + d[k,i]$ pour un sommet $k \neq i$. Cette condition implique dans le graphe G contient une marche directe de i à k et une marche directe de k à i dont la somme est négative. L'union des ces deux marches est une marche fermée. Ainsi, nous pouvons décomposer cette ensemble comme une somme de cycles orientés.
   Sachant que $d[i,i] < 0$, au moins un des cycles orientés admet un coût négatif.
2. Considérons le cas $d[i,j] < -nc$ pour $i \neq j$. Considérons également la première intération pour laquelle cette condition est vraie. Dans ce cas il existe une marche directe du sommet i au sommet j de coût $-nC$. Sachant que cette marche directe peut-être décomposée en un chemin élémentaire P et des cycles orientés. Le coût du chemin est d'au moins $-(n-1)C$, au moins un de ces cycles admet un coût négatif.

Dans l'algorithme de FLoyd-Warshall il suffit de regarder si $d[i,i] < 0$.

□

Question : Comment identifier le cycle de coût négatif ?

9.6 Planification de projets et plus longs chemins

Dans la gestion d'un projet de grande envergure on est souvent amené à décomposer celui-ci en différentes tâches. Chaque tâche k possède une durée w_k. De plus toutes les tâches ne sont pas indépendantes. Certaines doivent être exécutées avant d'autres.

Définition 9.6.1 *On dit qu'il existe une* relation (ou contrainte) de précédence *entre les tâches i et j, si la tâche i doit être accomplie avant que la tâche j ne puisse commencer.*

Le principal objectif est de calculer la date au plus tôt de chaque tâche, c'est-à-dire la date au plus tôt à laquelle elle peut commencer, tout en s'assurant que les contraintes de précédences sont bien respectées.
La *méthode du chemin critique* permet d'atteindre cet objectif.
On commence par construire un graphe orienté qui va représenter les relations de précédence entre les différentes tâches.

Définition 9.6.2 *Le* graphe de tâches *a pour sommets l'ensemble des tâches auxquelles on ajoute deux sommets supplémentaires, que nous notons s et t. Les arcs sont construits en utilisant les trois règles ci-dessous :*

- *Il existe un arc, de poids nul, depuis le sommet s vers toute tâche n'ayant pas de prédécesseur.*
- *Il existe un arc (i, j), de poids w_i si la tâche i est un prédécesseur de la tâche j.*
- *Si la tâche i ne précède aucune autre tâche, alors il existe un arc (i, t) de poids w_i.*

Remarque : Le graphe des tâches doit être acyclique.

Lemme 9.6.1 *La date de début au plus tôt d'une tâche k est égale à la longueur d'un plus long chemin depuis le sommet s jusqu'au sommet k dans le graphe des tâches.*

Preuve C'est évident. C'est une borne inférieure de la date de début au plus tôt. □

Définition 9.6.3 *On appelle* chemin critique *tout plus long chemin depuis la source s jusqu'à un sommet k, ou jusqu'au sommet t.*

Un tel chemin est appelé *critique*, car tout retard dans l'une quelconque des tâches rencontrées le long de ce chemin entraine inévitablement un retard pour la tâche k.

Lemme 9.6.2 *Le temps minimum pour réaliser un projet est égal à la longueur d'un chemin critique de s jusqu'à t.*

Preuve Le chemin critique donne la longueur du plus long chemin dans un graphe. La longueur de ce chemin est incomprésible. □

Si on suppose qu'on a une date d'exécution limite à respecter pour le projet

Définition 9.6.4 *La date au plus tard de toute tâche k est égale à la date de réalisation du projet moins la longueur d'un plus long chemin depuis k jusqu'à t.*

Remarque : Toute tâche exécutée après sa date au plus tard entraine un allongement de la durée du projet. Si on veut minimiser la durée du projet, l'écart entre la date au plus tôt et la date au plus tard est nul sur le chemin critique.

Algorithme 9.4 Algorithme de Bellman-Ford appliqué au problème d'ordonnancement

$d_0(\alpha) = 0$ et $d_0(t) = -\infty$ avec $s \neq t$
while Les valeurs $d_k(t)$ sont modifiées **do**
  Calculer successivement $d_1,\ d_2, \ldots$ en utilisant la formule suivante : $d_{k+1}(t) = \max\{d_k(t), \max_{(u,t)\in E}(d_k(u) + w_{ut}\}\}$
  Marquer le prédécesseur de t
end while

Lemme 9.6.3 *IL existe un plus long chemin de s à x.*

Preuve Pour tout sommmet x, il existe un chemin $(s, \ldots, x)$ car α est une source ; d'autre part, G étant sans circuit tous les chemins sont élémentaires, ils sont donc de longueur finie ; par conséquent il existe un plus long chemin de s à x.

□

Lemme 9.6.4 *L'algorithme 9.6 calcule les plus long chemin.*

Preuve
La preuve se fait par récurrence sur le numéro de l'itération. Soit une numérotation topologique des sommets de G. Nous allons montrer que pour l'itération k, la valeur $d_{k+1}(t)$ du début de l'itération correspond à la longueur d'un plus long chemin de s au sommet $k+1$. qui n'emprunte pas le sommet i et que la valeur $d_{k+1}(t)$ obtenue à la fin de l'itération correspond à la longueur d'un plus long chemin de s au sommet $k+1$.
Notons premièrement que, les sommets étant examinés dans un ordre topologique, pendant l'itération k seules les valeurs $d_{k'},\ k' > k$ sont susceptibles d'être modifiées.

G est sans circuit et à l'initialisation on a $d_\alpha = 0$. Il y a un chemin unique du sommet α au sommet 2 constitué du seul arc $(\alpha, 2)$ de valuation $w_{\alpha 2}$, donc il n'existe pas de chemin de α à 2 n'empruntant pas le sommet α et par convention le plus long chemin d'un ensemble vide est $-\infty$; à l'issue de la première itération on obtient $d_2(t) = w_{\alpha 2}$, la propriété est donc vraie pour $k = 1$.
Supposons cette propriété satisfaite jusqu'à l'itération $k-1$. Considérons l'itération k ; suivant l'hypothèse de récurrence, au début de cette itération, $d_{k+1}(t)$ est la longueur d'un plus long chemin de α à $k+1$ qui ne passe pas par le sommet k et $d_k(t)$ est la longueur d'un plus long chemin de α à k ; si l'arc $(k, k+1)$ appartient au graphe $d_k(t) + w_{k(k+1)}$ est la longueur d'un plus long chemin de α à k passant par le sommet k. Donc , la valeur $d_{k+1}(t)$, modifiée par l'instruction $d_{k+1}(t) = \max\{d_k(t), \max_{(u,t)\in E}(d_k(u) + w_{ut}\}\}$ au cours de l'itération si l'arc $(k, k+1)$ existe, est la longueur d'un plus long chemin reliant α au sommet $k+1$.
□

Exemple 9.6.1
L'exemple suivant concerne la construction d'une maison. Le tableau donne pour chaque tâche un, un libellé, et la liste des prédécesseurs.

Code tâche	Libellé	Durée (en semaines)	Prédécesseurs
1	Maçonnerie	7	aucun
2	Charpente de la toiture	3	1
3	Toiture	1	2
4	Plomberie et électricité	8	1
5	Façade	2	3, 4
6	Fenêtre	1	3, 4
7	Aménagement du jardin	1	3, 4
8	Plafonds	3	6
9	Peintures	2	8
10	Emménagement	1	5, 7, 9

Après ajout des tâches fictives de début et de fin que nous notons $\alpha = 11$ et $\omega = 12$, le graphe du projet se présente comme suit. Les dates au plus tôt trouvées en appliquant l'algorithme 9.6 figurent au-dessus de chaque sommet. La durée totale du projet est $t_\omega = 22$, due au chemin critique $(\alpha, 1, 4, 6, 8, 9, 10, \omega)$.

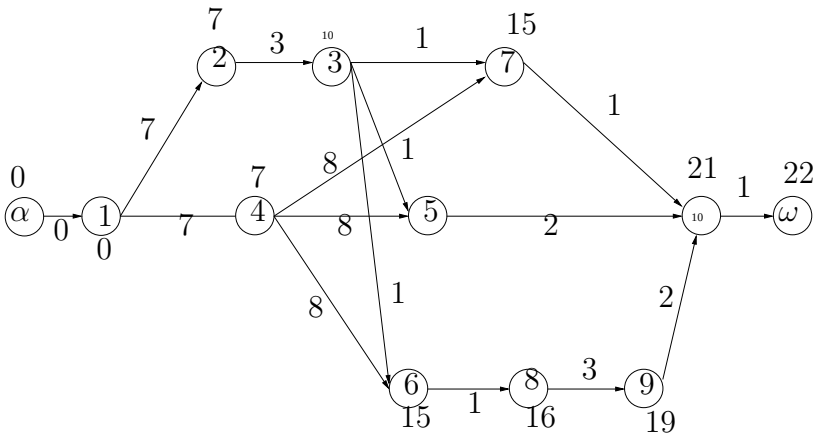
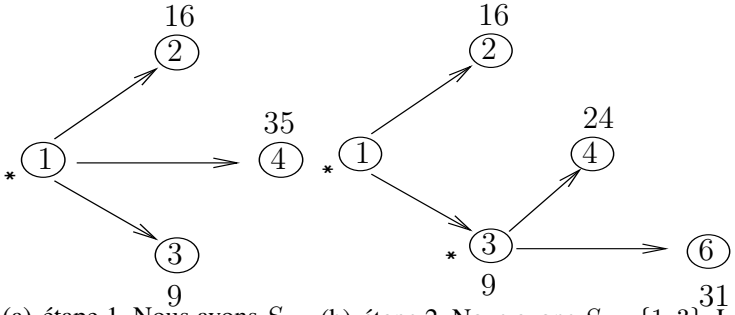


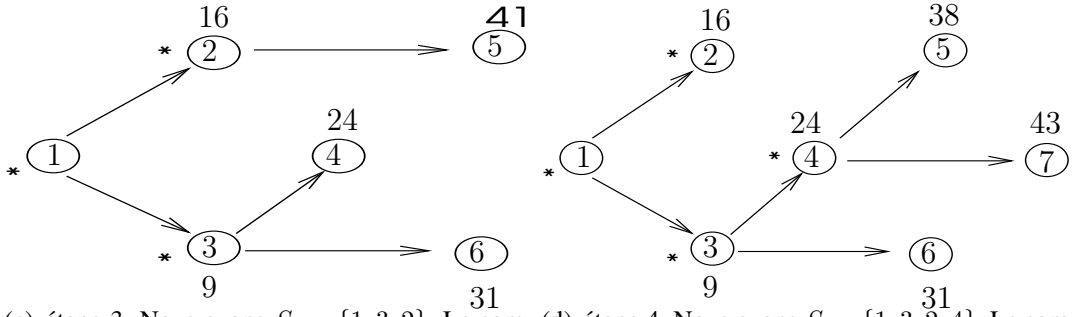
FIGURE 9.8 – *Représentation sous forme de graphe de précédence*

□



(a) étape 1. Nous avons $S = \{1\}$. Le sommet 1 a été choisi car c'est lui qui avait la plus petite des étiquettes ($d(1) = 0$). Les raccourcis $(1,2), (1,3)$ et $(1,4)$ ont été empruntés.

(b) étape 2. Nous avons $S = \{1,3\}$. Le sommet 3 a été choisi car c'est lui qui avait la plus petite étiquette, égale à 9, parmi les sommets 2,3 et 4 ayant respectivement les étiquettes 16,9 et 35. Les raccourcis $(3,4)$ et $(3,6)$ ont été empruntés.



(c) étape 3. Nous avons $S = \{1,3,2\}$. Le sommet 2 a été choisi car c'est lui qui avait la plus petite étiquette, égale à 16, parmi les sommets 2,4 et 6 ayant respectivement les étiquettes 16,24 et 31.

(d) étape 4. Nous avons $S = \{1,3,2,4\}$. Le sommet 4 a été choisi car c'est lui qui avait la plus petite étiquette, égale à 24, parmi les sommets 4,5 et 6 ayant respectivement les étiquettes 24,31 et 41.

FIGURE 9.5 – *Illustration de l'algorithme de Dijkstra*