Optimisation combinatoire

Springer
Paris
Berlin
Heidelberg
New York
Hong Kong
Londres
Milan
Tokyo

Bernhard Korte
Jens Vygen

Optimisation combinatoire

Théorie et algorithmes

Traduit de l'anglais par Jean Fonlupt et Alexandre Skoda

Springer

Bernhard Korte
Research Institute
for Discrete Mathematics
University of Bonn
Lennéstraße 2
53113 Bonn
Germany
dm@or.uni-bonn.de

Jens Vygen
Research Institute
for Discrete Mathematics
University of Bonn
Lennéstraße 2
53113 Bonn
Germany
vygen@or.uni-bonn.de

Traducteurs

Jean Fonlupt
Professeur émérite
Université Paris-VI
Faculté de mathématiques
175, rue du Chevaleret
75013 Paris
Jean.Fonlupt@math.jussieu.fr

Alexandre Skoda
Université Paris-I
Panthéon-Sorbonne
Centre d'économie de la Sorbonne
106-112, boulevard de l'Hôpital
75013 Paris
alexandre.skoda@univ-paris1.fr

ISBN : 978-2-287-99036-6 Springer Paris Berlin Heidelberg New York

Maquette de couverture : Jean-François MONTMARCHÉ
Illustration de couverture : Ina PRINZ

Collection *IRIS*
Dirigée par Nicolas Puech

Ouvrages parus :

– *Méthodes numériques pour le calcul scientifique*. *Programmes en Matlab*
A. Quarteroni, R. Sacco, F. Saleri, Springer-Verlag France, 2000

– *Calcul formel avec MuPAD*
F. Maltey, Springer-Verlag France, 2002

– *Architecture et micro-architecture des processeurs*
B. Goossens, Springer-Verlag France, 2002

– *Introduction aux mathématiques discrètes*
J. Matousek, J. Nesetril, Springer-Verlag France, 2004

– *Les virus informatiques : théorie, pratique et applications*
É. Filiol, Springer-Verlag France, 2004

– *Introduction pratique aux bases de données relationnelles*. *Deuxième édition*
A. Meier, Springer-Verlag France, 2006

– *Bio-informatique moléculaire. Une approche algorithmique*
P.A. Pevzner, Springer-Verlag France, 2006

– *Algorithmes d'approximation*
V. Vazirani, Springer-Verlag France, 2006

– *Techniques virales avancées*
É. Filiol, Springer-Verlag France, 2007

– *Codes et turbocodes*
C. Berrou, Springer-Verlag France, 2007

– *Introduction à Scilab. Deuxième édition*
J.P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikouhah, S. Steer,
Springer-Verlag France, 2007

– *Maple : règles et fonctions essentielles*
N. Puech, Springer-Verlag France, 2009

– *Les virus informatiques : théorie, pratique et applications. Deuxième édition*
É. Filiol, Springer-Verlag France, 2009

À paraître :

– *Concepts et méthodes en phylogénie moléculaire*
G. Perrière, Springer-Verlag France, 2010

Préface

Ce livre est la traduction française de la quatrième édition du livre *Combinatorial Optimization : Theory and Algorithms* écrit par deux éminents spécialistes, Bernhard Korte et Jens Vygen, professeurs à l'université de Bonn. Considéré comme un ouvrage de référence, il s'adresse à des chercheurs confirmés qui travaillent dans le champ de la recherche fondamentale ou de ses applications (R&D). Il donne une vision complète de l'optimisation combinatoire et peut donc aussi intéresser de nombreux scientifiques non spécialistes ayant une bonne culture en mathématiques et des connaissances de base en informatique.

L'optimisation combinatoire est un domaine assez récent des mathématiques appliquées, qui plonge ses racines dans la combinatoire (principalement la théorie des graphes), la recherche opérationnelle et l'informatique théorique. Une des raisons de son développement est liée au nombre considérable de problèmes concrêts qu'elle permet de formuler. Il s'agit en grande partie de problèmes pour lesquels on connaît de «bons» algorithmes de résolution ; ceux-ci sont étudiés dans la première partie de ce livre. Une des originalités de cet ouvrage, par rapport à d'autres traités, est de présenter les algorithmes de résolution ayant la meilleure borne de complexité connue à ce jour.

La seconde partie traite des problèmes difficiles à résoudre sur le plan algorithmique et connus sous le nom de problèmes *NP-difficiles*. Le plus célèbre d'entre eux, celui du voyageur de commerce, fait l'objet, au chapitre 21, d'une étude particulièrement approfondie. D'autres tout aussi importants, comme les problèmes de conception de réseaux, de multi-flots, de localisation de services, etc., bénéficient également d'une présentation détaillée, ce qui est peu fréquent dans la littérature et mérite d'être signalé.

Dans la traduction que nous proposons, nous avons cherché à traduire en français toutes les expressions et tous les termes anglo-saxons même quand aucune traduction n'existait ; il y a cependant quelques exceptions pour des termes très techniques qui ne sont universellement connus que sous leur dénomination anglaise. Nous avons en outre inclus quelques améliorations et corrections écrites par les auteurs après la parution de l'édition originale actuelle ; celles-ci seront intégrées dans la cinquième édition anglaise, actuellement en préparation.

Paris, juillet 2009 *Jean Fonlupt et Alexandre Skoda*

Avant-propos à la quatrième édition originale

Avec quatre éditions anglaises et quatre traductions en cours, nous sommes très heureux de l'évolution de notre livre ; celui-ci a été révisé, actualisé et amélioré de manière significative pour cette quatrième édition. Nous y avons inclus des matières classiques, parfois manquantes dans les éditions précédentes, notamment sur la programmation linéaire, la méthode network simplex et le problème de la coupe maximum. Nous avons également ajouté de nouveaux exercices et mis à jour les références.

Nous sommes reconnaissants à l'Union des académies allemandes des sciences et des lettres et à l'Académie des sciences du Land Rhénanie-du-Nord-Westphalie pour leur soutien permanent par l'intermédiaire du projet «Mathématiques discrètes et applications». Nous remercions également pour leurs commentaires précieux tous ceux qui nous ont contacté après la troisième édition, en particulier Takao Asano, Christoph Bartoschek, Bert Besser, Ulrich Brenner, Jean Fonlupt, Satoru Fujishige, Marek Karpinski, Jens Maßberg, Denis Naddef, Sven Peyer, Klaus Radke, Rabe von Randow, Dieter Rautenbach, Martin Skutella, Markus Struzyna, Jürgen Werber, Minyi Yue, et Guochuan Zhang. Nous continuerons à fournir des informations actualisées sur cet ouvrage à l'adresse :
`http ://www.or.uni-bonn.de/~vygen/co.html`

Bonn, août 2007 *Bernhard Korte et Jens Vygen*

Sommaire

Chapitre 1

Introduction

Commençons cet ouvrage par deux exemples.

Une machine est utilisée pour percer des trous dans des plaques de circuits imprimés. Comme de nombreux circuits sont produits, il est souhaitable que chaque circuit soit fabriqué aussi rapidement que possible. Nous ne pouvons agir sur le temps de perçage de chaque trou qui est fixé, mais nous pouvons chercher à minimiser le temps total de déplacement de la perceuse. Habituellement, les perceuses effectuent des déplacements dans deux directions : la table se déplace horizontalement tandis que le bras de la machine se déplace verticalement. Comme ces deux mouvements peuvent se faire simultanément, le temps nécessaire pour ajuster la machine entre deux positions est proportionnel au maximum des distances horizontales et verticales parcourues. Cette quantité est souvent appelée distance de la norme infini. (Les vieilles machines ne peuvent se déplacer que dans une direction à la fois ; le temps d'ajustement est alors proportionnel à la 1-distance, somme des distances horizontale et verticale.)

Un parcours optimal pour le perçage est donné par un ordre des positions des trous $p_1, \ldots, p_n$ qui rend minimum la quantité $\sum_{i=1}^{n-1} d(p_i, p_{i+1})$, d étant la distance de la norme infini : si $p = (x, y)$ et $p' = (x', y')$ sont deux points du plan, alors $d(p, p') := \max\{|x - x'|, |y - y'|\}$. Un ordre des trous peut être représenté par une permutation, c.-à-d. une bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.

La meilleure permutation dépend bien entendu de la position des trous ; pour chaque ensemble de positions, nous aurons une instance spécifique (suivant l'usage, nous utiliserons le terme «instance» de préférence à «exemple»). Nous dirons qu'une instance est une liste de points du plan, c.-à-d. une liste des coordonnées des trous à percer. Le problème peut alors se formuler de la manière suivante :

PROBLÈME DE PERÇAGE	
Instance	Un ensemble de points $p_1, \ldots, p_n \in \mathbb{R}^2$.
Tâche	Trouver une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ telle que $\sum_{i=1}^{n-1} d(p_{\pi(i)}, p_{\pi(i+1)})$ soit minimum.

Décrivons maintenant notre deuxième exemple. Nous devons effectuer un ensemble de tâches dont nous connaissons les temps d'exécution. Chaque tâche peut être confiée à une partie des employés. Plusieurs employés peuvent être affectés à une même tâche et chaque employé peut travailler sur plusieurs tâches mais pas simultanément. Notre objectif est d'exécuter l'ensemble des tâches aussi rapidement que possible.

Dans ce modèle, il suffira de déterminer le temps d'affectation de chaque employé aux différentes tâches. Le temps d'exécution de l'ensemble des tâches est alors égal au temps de travail de l'employé le plus occupé. Nous devons donc résoudre le problème suivant :

PROBLÈME D'AFFECTATION DES TÂCHES

Instance Un ensemble de nombres $t_1, \ldots, t_n \in \mathbb{R}_+$ (les temps d'exécution des n tâches), un nombre $m \in \mathbb{N}$ d'employés, et un sous-ensemble non vide $S_i \subseteq \{1, \ldots, m\}$ d'employés pour chaque tâche $i \in \{1, \ldots, n\}$.

Tâche Trouver des nombres $x_{ij} \in \mathbb{R}_+$ pour tout $i = 1, \ldots, n$ et $j \in S_i$ tels que $\sum_{j \in S_i} x_{ij} = t_i$ pour $i = 1, \ldots, n$ et tel que $\max_{j \in \{1,\ldots,m\}} \sum_{i : j \in S_i} x_{ij}$ soit minimum.

Voilà deux exemples typiques de problèmes d'optimisation combinatoire. La manière de modéliser un problème pratique en un problème abstrait d'optimisation combinatoire n'est pas l'objet de ce livre ; il n'y a d'ailleurs aucune recette pour réussir dans cette démarche. Outre la précision des données et des résultats attendus, il est souvent important pour un modèle d'ignorer certains paramètres non significatifs (par exemple, le temps de perçage qui ne peut être optimisé ou l'ordre suivant lequel les employés exécutent les tâches).

Notons enfin qu'il ne s'agit pas de résoudre un cas particulier d'un problème, comme celui du perçage ou celui d'affectation des tâches, mais de résoudre tous les cas possibles de ces problèmes. Étudions d'abord le PROBLÈME DE PERÇAGE.

1.1 Énumération

Quelle est l'allure d'une solution du PROBLÈME DE PERÇAGE ? Ce problème a un nombre infini d'instances possibles (tout ensemble fini de points du plan) et nous ne pouvons donc faire la liste des permutations optimales associées à toutes les instances. Ce que nous recherchons, c'est un algorithme qui associe, à chaque instance, une solution optimale. Un tel algorithme existe : étant donné un ensemble de n points, calculer la longueur du chemin associé à chacune des $n!$ permutations.

Il y a de nombreuses manières de formuler un algorithme, la différence se faisant principalement par le niveau de détails ou par le langage formel utilisé. Nous n'accepterons pas la proposition suivante comme définissant un algorithme : étant donné un ensemble de n points, trouver un chemin optimal qui sera l'output, c.-à-d. le résultat, car rien n'est dit sur la manière de trouver la solution optimale. La sugges-

tion précédente, d'énumérer l'ensemble des $n!$ permutations, est plus utile à condition de préciser la manière d'énumérer ces permutations. Voici une méthode : énumérons par comptage tous les n-uplets des nombres $1, \dots, n$, c.-à-d. les n^n vecteurs de $\{1, \dots, n\}^n$: partons de $(1, \dots, 1, 1)$, $(1, \dots, 1, 2)$ jusqu'à $(1, \dots, 1, n)$, passons à $(1, \dots, 1, 2, 1)$, et ainsi de suite : à chaque étape, nous ajoutons 1 à la dernière composante sauf si celle-ci vaut n, auquel cas nous revenons à la dernière composante plus petite que n, lui ajoutons 1 et réinitialisons à 1 toutes les composantes suivantes. Cette technique est parfois appelée «*backtracking*» (en français, retour arrière). L'ordre selon lequel les vecteurs de $\{1, \dots, n\}^n$ sont énumérés est appelé ordre lexicographique.

Définition 1.1. *Soient $x, y \in \mathbb{R}^n$ deux vecteurs. Nous dirons qu'un vecteur x est* **lexicographiquement plus petit** *que y s'il existe un indice $j \in \{1, \dots, n\}$ tel que $x_i = y_i$ pour $i = 1, \dots, j-1$ et $x_j < y_j$.*

Il nous suffit maintenant de vérifier si, au cours de l'énumération, chaque vecteur de $\{1, \dots, n\}^n$ a des composantes différentes et voir dans ce cas si le chemin représenté par cette permutation est plus court que le meilleur chemin trouvé précédemment.

Comme cet algorithme énumère n^n vecteurs, il nécessitera au moins n^n étapes. Cela n'est pas très efficace puisque le nombre de permutations de $\{1, \dots, n\}$ est $n!$ qui est bien plus petit que n^n. (Par la formule de Stirling $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$ (Stirling [1730]) ; voir exercice 1.) Montrons comment énumérer tous les chemins en approximativement $n^2 \cdot n!$ étapes grâce à l'algorithme suivant qui énumère toutes les permutations suivant un ordre lexicographique :

ALGORITHME D'ÉNUMÉRATION DES CHEMINS

Input	Un nombre naturel $n \geq 3$. Un ensemble $\{p_1, \dots, p_n\}$ de points dans le plan.
Output	Une permutation $\pi^* : \{1, \dots, n\} \to \{1, \dots, n\}$ telle que $\mathit{coût}(\pi^*) := \sum_{i=1}^{n-1} d(p_{\pi^*(i)}, p_{\pi^*(i+1)})$ soit minimum.

① $\pi(i) := i$ et $\pi^*(i) := i$ pour $i = 1, \dots, n$. Posons $i := n - 1$.

② Soit $k := \min(\{\pi(i) + 1, \dots, n + 1\} \setminus \{\pi(1), \dots, \pi(i-1)\})$.

③ **If** $k \leq n$ **then** :
  $\pi(i) := k$.
  If $i = n$ et $\mathit{coût}(\pi) < \mathit{coût}(\pi^*)$ **then** $\pi^* := \pi$.
  If $i < n$ **then** $\pi(i+1) := 0$ et $i := i + 1$.
If $k = n + 1$ **then** $i := i - 1$.
If $i \geq 1$ **then go to** ②.

Partant de $(\pi(i))_{i=1,\dots,n} = (1, 2, 3, \dots, n-1, n)$ et $i = n - 1$, l'algorithme trouve à chaque étape la meilleure valeur possible suivante de $\pi(i)$ (sans utiliser $\pi(1), \dots, \pi(i-1)$). S'il n'existe plus aucune possibilité pour $\pi(i)$ (c.-à-d. $k =$

$n+1$), alors l'algorithme décrémente i (*backtracking*). Sinon il affecte à $\pi(i)$ la nouvelle valeur k. Si $i = n$, la nouvelle permutation est évaluée, sinon l'algorithme évaluera toutes les valeurs possibles pour $\pi(i+1), \ldots, \pi(n)$, en affectant à $\pi(i+1)$ la valeur 0 et en incrémentant i.

Ainsi tous les vecteurs de permutation $(\pi(1), \ldots, \pi(n))$ sont générés suivant un ordre lexicographique. Par exemple, les premières itérations dans le cas $n = 6$ sont décrites comme suit :

$$\begin{array}{llll}
 & \pi := (1,2,3,4,5,6), & i := 5 & \\
k := 6, & \pi := (1,2,3,4,6,0), & i := 6 & \\
k := 5, & \pi := (1,2,3,4,6,5), & & \textit{coût}(\pi) < \textit{coût}(\pi^*)\,? \\
k := 7, & & i := 5 & \\
k := 7, & & i := 4 & \\
k := 5, & \pi := (1,2,3,5,0,5), & i := 5 & \\
k := 4, & \pi := (1,2,3,5,4,0), & i := 6 & \\
k := 6, & \pi := (1,2,3,5,4,6), & & \textit{coût}(\pi) < \textit{coût}(\pi^*)\,?
\end{array}$$

Puisque l'algorithme compare le coût de la solution courante à π^*, le meilleur chemin actuel, il fournit bien le chemin optimal. Mais quel est le nombre d'étapes ? La réponse dépendra de ce que nous appelons un «pas» de l'algorithme. Comme le nombre de pas ne doit pas dépendre de l'implémentation, nous devons ignorer les facteurs constants. Ainsi, ① nécessitera au moins $2n+1$ étapes et au plus cn étapes, c étant une constante. La notation suivante sera utile pour ignorer les constantes :

Définition 1.2. *Soient $f, g : D \to \mathbb{R}_+$ deux fonctions. Nous dirons que f est $O(g)$ (et nous écrirons parfois $f = O(g)$) s'il existe des constantes $\alpha, \beta > 0$ telles que $f(x) \leq \alpha g(x) + \beta$ pour tout $x \in D$. Si $f = O(g)$ et $g = O(f)$ nous dirons alors que $f = \Theta(g)$ (et bien entendu $g = \Theta(f)$). Dans ce cas, f et g auront le même* **taux de croissance**.

Remarquons que la relation $f = O(g)$ n'implique aucune symétrie entre f et g. Pour illustrer cette définition, prenons $D = \mathbb{N}$ et soit $f(n)$ le nombre de pas ou d'étapes élémentaires de ①. En posant $g(n) = n$ ($n \in \mathbb{N}$), il est évident que $f = O(g)$ (et que, également, $f = \Theta(g)$) ; nous dirons que ① s'exécute en un temps $O(n)$ ou en temps linéaire. L'exécution de ③ se fait en un nombre constant de pas (nous dirons aussi en temps $O(1)$ ou en temps constant) sauf dans le cas où les coûts de deux chemins doivent être comparés, ce qui nécessitera un temps $O(n)$.

Que peut-on dire de ② ? Vérifier si $j = \pi(h)$ pour tout $j \in \{\pi(i)+1, \ldots, n\}$ et tout $h \in \{1, \ldots, i-1\}$ se fait en $O((n - \pi(i))i)$ étapes, c.-à-d. en un temps $\Theta(n^2)$. On peut améliorer ce temps en utilisant un tableau auxiliaire indexé par $1, \ldots, n$:

② **For** $j := 1$ **to** n **do** $aux(j) := 0$.
For $j := 1$ **to** $i-1$ **do** $aux(\pi(j)) := 1$.
$k := \pi(i) + 1$.
While $k \leq n$ et $aux(k) = 1$ **do** $k := k + 1$.

De cette manière, ② s'exécute en un temps $O(n)$. Nous n'étudierons pas dans ce livre ce genre d'améliorations algorithmiques, laissant au lecteur le choix des bonnes mises en œuvre.

Examinons maintenant le temps total d'exécution de l'algorithme. Puisque le nombre de permutations est $n!$, il nous faut trouver le temps de calcul entre deux permutations. Le compteur i peut décroître de la valeur n à un indice i', une nouvelle valeur de $\pi(i') \leq n$ étant trouvée. Puis le compteur est réincrémenté jusqu'à la valeur $i = n$. Tant que le compteur i est constant, chacune des étapes ② et ③ est exécutée une seule fois, sauf dans le cas $k \leq n$ et $i = n$; dans ce cas ② et ③ sont exécutées deux fois. Ainsi le nombre de pas entre deux permutations est au plus $4n$ fois ② et ③, c.-à-d. $O(n^2)$. Le temps total d'exécution de l'ALGORITHME D'ÉNUMÉRATION DES CHEMINS est $O(n^2 n!)$.

On peut faire encore mieux ; une analyse plus fine montre que le temps de calcul est seulement $O(n \cdot n!)$ (exercice 4).

Cependant, le temps de calcul de l'algorithme est trop important quand n devient grand, car le nombre de chemins croît d'une manière exponentielle avec le nombre de points ; déjà pour 20 points, on a $20! = 2\ 432\ 902\ 008\ 176\ 640\ 000 \approx 2,4 \cdot 10^{18}$ chemins différents et même les ordinateurs les plus puissants auraient besoin de plusieurs années pour tous les examiner. Ainsi une énumération complète est impossible à envisager même pour des instances de taille modeste.

L'objet de l'optimisation combinatoire est de trouver de meilleurs algorithmes pour ce type de problèmes. Nous devrons souvent trouver le meilleur élément d'un ensemble fini de solutions réalisables (dans nos exemples : chemins de perçage ou permutations). Cet ensemble n'est pas défini explicitement, mais dépend implicitement de la structure du problème. Un algorithme doit pouvoir exploiter cette structure.

Dans le PROBLÈME DE PERÇAGE une instance avec n points sera décrite par $2n$ coordonnées. Alors que l'algorithme précédent énumère les $n!$ chemins, on peut imaginer qu'il existe un algorithme trouvant le chemin optimal plus rapidement, disons en n^2 étapes de calcul. On ne sait pas si un tel algorithme existe (on verra cependant au chapitre 15 que cela est improbable). Il existe cependant des algorithmes bien meilleurs que ceux fondés sur la méthode d'énumération.

1.2 Temps d'exécution des algorithmes

On peut donner une définition formelle d'un algorithme, et c'est ce que nous ferons au chapitre 15.1. Cependant, de tels modèles conduisent à des descriptions longues et fastidieuses. Il en est de même pour les preuves mathématiques : bien que le concept de preuve puisse être formalisé, personne n'utilise un tel formalisme pour décrire des preuves, car elles deviendraient trop longues et presque illisibles.

Ainsi les algorithmes présentés dans ce livre seront-ils écrits dans un langage informel. Cependant, ils seront suffisamment détaillés pour qu'un lecteur ayant un peu d'expérience puisse les programmer sur un ordinateur sans trop d'effort.

Puisque nous ne prenons pas en compte les facteurs constants quand nous mesurons le temps de calcul, nous n'avons pas à spécifier un modèle concret d'ordinateur. Nous comptons les pas élémentaires sans nous soucier du temps d'exécution de ces pas. Comme exemples de pas élémentaires, citons les affectations de variables,

l'accès aléatoire à une variable dont l'adresse est stockée dans un autre registre, les sauts conditionnels (if – then – go to), ainsi que les opérations arithmétiques élémentaires telles que l'addition, la soustraction, la multiplication, la division, la comparaison de nombres.

Un algorithme consiste en un ensemble d'inputs valides et une suite d'instructions composées d'opérations élémentaires, de telle sorte que pour chaque input valide, le déroulement de l'algorithme soit une suite bien définie d'opérations élémentaires fournissant un output. La question essentielle sera alors d'obtenir une borne satisfaisante du nombre d'opérations, en fonction de la taille de l'input.

L'input est en général une liste de nombres. Si tous ces nombres sont des entiers, nous pouvons les coder dans une représentation binaire en réservant un emplacement de $O(\log(|a|+2))$ bits pour stocker un entier a. Les nombres rationnels peuvent être stockés en codant séparément leur numérateur et leur dénominateur. La **taille de l'input** notée $\text{taille}(x)$ d'une instance x avec des données rationnelles est le nombre total de bits utilisés dans la représentation binaire.

Définition 1.3. *Soit A un algorithme qui accepte des inputs d'un ensemble X, et soit $f : \mathbb{N} \to \mathbb{R}_+$. S'il existe une constante $\alpha > 0$ telle que A se termine après au plus $\alpha f(\text{taille}(x))$ pas élémentaires (en incluant les opérations arithmétiques) pour chaque input $x \in X$, nous dirons alors que A* **s'exécute en un temps** *$O(f)$. Nous dirons également que $O(f)$ est le* **temps de calcul** *ou la* **complexité** *de A.*

Définition 1.4. *Un algorithme acceptant des inputs rationnels est dit* **polynomial** *s'il s'exécute en un temps $O(n^k)$ quand la taille de l'input est n, k étant fixé, et si la taille de tous les nombres intermédiaires calculés n'excède pas $O(n^k)$ bits.*

Un algorithme acceptant des inputs arbitraires est dit **fortement polynomial** *si son temps de calcul est $O(n^k)$ pour tout input de n nombres, k étant une constante fixée, et s'il se termine en temps polynomial dans le cas d'inputs rationnels. Si $k = 1$, nous dirons que l'algorithme est* **linéaire**.

Notons que le temps de calcul peut être différent pour des instances distinctes de même taille (ce n'était pas le cas pour l'ALGORITHME D'ÉNUMÉRATION DES CHEMINS). Nous considérerons le temps de calcul dans le pire des cas, c.-à-d. la fonction $f : \mathbb{N} \to \mathbb{N}$ où $f(n)$ est le maximum du temps de calcul d'une instance de taille n. Pour certains algorithmes, nous ne connaissons pas le taux de croissance de f, mais nous avons seulement une borne supérieure.

Il se peut que le temps de calcul dans le pire des cas soit une mesure pessimiste si le pire des cas se produit rarement. Dans certaines situations, un temps de calcul moyen fondé sur des modèles probabilistes serait plus adéquat, mais nous n'aborderons pas cette question dans ce livre.

Si A est un algorithme qui, pour chaque input $x \in X$, calcule l'output $f(x) \in Y$, nous dirons que A **calcule** $f : X \to Y$. Une fonction calculée par un algorithme polynomial sera dite **calculable en temps polynomial**.

Les algorithmes polynomiaux sont quelquefois appelés «bons» ou «efficaces». Ce concept a été introduit par Cobham [1964] et Edmonds [1965]. La table 1.1 illustre cela en fournissant les temps de calcul pour divers temps de complexité.

Table 1.1.

n	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
10	3 μs	1 μs	3 μs	2 μs	1 μs	4 ms
20	9 μs	4 μs	36 μs	420 μs	1 ms	76 années
30	15 μs	9 μs	148 μs	20 ms	1 s	$8 \cdot 10^{15}$ a.
40	21 μs	16 μs	404 μs	340 ms	1100 s	
50	28 μs	25 μs	884 μs	4 s	13 jours	
60	35 μs	36 μs	2 ms	32 s	37 années	
80	50 μs	64 μs	5 ms	1075 s	$4 \cdot 10^7$ a.	
100	66 μs	100 μs	10 ms	5 heures	$4 \cdot 10^{13}$ a.	
200	153 μs	400 μs	113 ms	12 années		
500	448 μs	2.5 ms	3 s	$5 \cdot 10^5$ a.		
1000	1 ms	10 ms	32 s	$3 \cdot 10^{13}$ a.		
10^4	13 ms	1 s	28 heures			
10^5	166 ms	100 s	10 années			
10^6	2 s	3 heures	3169 a.			
10^7	23 s	12 jours	10^7 a.			
10^8	266 s	3 années	$3 \cdot 10^{10}$ a.			
10^{10}	9 heures	$3 \cdot 10^4$ a.				
10^{12}	46 jours	$3 \cdot 10^8$ a.				

Pour différentes tailles d'inputs n, nous indiquons les temps de calcul de six algorithmes qui nécessitent $100n \log n$, $10n^2$, $n^{3.5}$, $n^{\log n}$, 2^n, et $n!$ opérations élémentaires ; nous supposons qu'une opération élémentaire s'effectue en une nanoseconde. Comme partout dans ce livre, «log» est le logarithme en base 2.

Ainsi que la table 1.1 le montre, les algorithmes polynomiaux sont plus rapides pour les instances de taille suffisamment importante. Cette table indique également que les facteurs constants de taille modérée ne sont pas très importants si on considère la croissance asymptotique du temps de calcul.

La table 1.2 indique la taille maximum d'inputs résolubles en une heure pour les six algorithmes précédents. Pour (a) nous supposons qu'une opération élémentaire s'effectue en une nanoseconde ; (b) donne les résultats pour une machine dix fois plus rapide. Les algorithmes polynomiaux peuvent traiter de grandes instances en des temps raisonnables. Cependant, même en multipliant par 10 la rapidité de calcul des ordinateurs, on n'augmente pas de manière significative la taille des instances que l'on peut résoudre pour des algorithmes exponentiels, ce qui n'est pas le cas pour les algorithmes polynomiaux.

Les algorithmes (fortement) polynomiaux et si possible linéaires sont ceux qui nous intéressent. Il existe des problèmes pour lesquels il n'existe aucun algorithme polynomial et d'autres pour lesquels il n'existe aucun algorithme. (Par exemple, un problème qui peut se résoudre en un temps fini mais pas en temps polynomial

Table 1.2.

	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
(a)	$1.19 \cdot 10^9$	60000	3868	87	41	15
(b)	$10.8 \cdot 10^9$	189737	7468	104	45	16

est celui de décider si une expression régulière définit l'ensemble vide ; voir Aho, Hopcroft et Ullman [1974]. Un problème pour lequel il n'existe aucun algorithme est le «HALTING PROBLEM», décrit dans l'exercice 1 du chapitre 15.)

Cependant, presque tous les problèmes étudiés dans ce livre appartiennent à une des deux classes suivantes : pour les problèmes de la première classe, il existe un algorithme polynomial ; pour les problèmes de la seconde, l'existence d'un algorithme polynomial est une question ouverte. Néanmoins, nous savons que si un de ces problèmes peut se résoudre en temps polynomial, alors tous les problèmes appartenant à cette seconde classe sont également résolubles en temps polynomial. Une formulation et une preuve de cette affirmation seront données au chapitre 15.

Le PROBLÈME DE L'AFFECTATION DES TÂCHES appartient à la première classe, le PROBLÈME DE PERÇAGE appartient à la seconde.

Ces deux classes divisent à peu près ce livre en deux parties. Nous étudierons d'abord les problèmes pour lesquels on connaît des algorithmes polynomiaux. Puis, à partir du chapitre 15, nous nous intéresserons aux problèmes difficiles. Bien qu'on ne connaisse aucun algorithme polynomial dans ce cas, il existe souvent de bien meilleures méthodes que l'énumération complète. De plus, pour de nombreux problèmes (incluant le PROBLÈME DE PERÇAGE), on peut trouver des solutions approchées à un certain pourcentage de l'optimum en temps polynomial.

1.3 Problèmes d'optimisation linéaire

Revenons sur notre deuxième exemple, le PROBLÈME D'AFFECTATION DES TÂCHES, pour illustrer brièvement un sujet central de ce livre.

Le PROBLÈME D'AFFECTATION DES TÂCHES est totalement différent du PROBLÈME DE PERÇAGE puisque chaque instance non triviale a un nombre infini de solutions. Nous pouvons reformuler ce problème en introduisant une variable T qui sera le temps nécessaire à l'achèvement de toutes les tâches :

$$\begin{array}{lll} \min & T & \\ \text{s.c.} & \sum_{j \in S_i} x_{ij} = t_i & (i \in \{1, \ldots, n\}) \\ & x_{ij} \geq 0 & (i \in \{1, \ldots, n\},\ j \in S_i) \\ & \sum_{i : j \in S_i} x_{ij} \leq T & (j \in \{1, \ldots, m\}) \end{array} \tag{1.1}$$

(s.c. est une abréviation pour «sous les contraintes»)

Les nombres t_i et les ensembles S_i ($i = 1, \ldots, n$) sont donnés, et nous cherchons à calculer les variables x_{ij} et T. Un problème d'optimisation de ce type, avec une fonction objectif linéaire et des contraintes linéaires, est appelé **programme linéaire**. L'ensemble des solutions réalisables de (1.1) est un **polyèdre** ; cet ensemble convexe a un nombre fini de points extrêmes qui inclut la **solution optimale** de ce programme linéaire. Un programme linéaire peut donc, en théorie, se résoudre par énumération complète, mais de bien meilleures méthodes existent comme nous le verrons ultérieurement.

Bien que de nombreux algorithmes existent pour résoudre des programmes linéaires, les techniques générales sont souvent moins performantes que les algorithmes spécifiques qui exploitent la structure du problème. Dans notre exemple, il est judicieux de modéliser les ensembles S_i, $i = 1, \ldots, n$, à l'aide d'un **graphe** : associons à chaque tâche i et à chaque employé j un point (appelé sommet) et relions par une arête un employé i et une tâche j si i peut être affecté à j (c.-à-d. si $j \in S_i$). Les graphes constituent une structure combinatoire fondamentale ; de nombreux problèmes d'optimisation combinatoire se décrivent de manière naturelle dans le contexte de la théorie des graphes.

Supposons que le temps d'exécution de chaque tâche soit de une heure et que nous voulions savoir si toutes les tâches seront terminées en une heure. Ce problème revient à trouver des nombres x_{ij} ($i \in \{1, \ldots, n\}, j \in S_i$) tels que $0 \leq x_{ij} \leq 1$ pour tout i et j, $\sum_{j \in S_i} x_{ij} = 1$ pour $i = 1, \ldots, n$, et $\sum_{i : j \in S_i} x_{ij} \leq 1$ pour $j = 1, \ldots, n$. On peut montrer que si ce problème a une solution, celle-ci peut être choisie entière, les quantités x_{ij} valant alors 0 ou 1. Cela revient à affecter chaque tâche à un seul employé qui effectuera au plus une seule tâche. Dans le langage de la théorie des graphes, nous cherchons un **couplage** couvrant toutes les tâches. Le problème de la recherche d'un couplage optimal est un des problèmes classiques de l'optimisation combinatoire.

L'étude et le rappel de notions de base en théorie des graphes et en programmation linéaire sera l'objet des chapitres 2 et 3. Au chapitre 4 nous montrerons comment résoudre les programmes linéaires en temps polynomial, et au chapitre 5 nous étudierons les polyèdres entiers. Les chapitres suivants seront consacrés à l'étude de problèmes classiques en optimisation combinatoire.

1.4 Tri

Concluons ce chapitre en nous intéressant à un cas particulier du PROBLÈME DE PERÇAGE ; plus précisément, nous supposerons que tous les trous doivent être percés sur une même ligne horizontale. Il suffit de connaître une seule coordonnée pour chaque point p_i, $i = 1, \ldots, n$. Une solution du problème de perçage est alors facile à trouver : il s'agit de faire le tri des points selon cette coordonnée, le bras de la machine se déplaçant alors de la gauche vers la droite. Nous n'aurons donc pas à examiner les $n!$ permutations, pour trouver le chemin optimal : il est en effet très facile de trier n nombres dans un ordre non décroissant en un temps $O(n^2)$.

Trier n nombres en un temps $O(n \log n)$ demande un peu plus de réflexion. Il y a de nombreux algorithmes ayant cette complexité ; nous présentons ici l'algorithme bien connu de TRI-FUSION (en anglais, *merge-sort*) : la liste initiale est d'abord divisée en deux sous-listes de même taille approximative. Puis chaque sous-liste est triée (récursivement, par le même algorithme). Enfin, les deux sous-listes triées sont fusionnées. Cette méthode, appelée «diviser pour régner» (*divide and conquer* en anglais) est souvent utilisée. Voir le paragraphe 17.1 pour une autre illustration.

Nous n'avons pas présenté ce qu'on appelle les algorithmes récursifs. Ce ne sera pas nécessaire ici ; il nous suffira de savoir que tout algorithme récursif peut être transformé en un algorithme séquentiel sans accroître le temps de calcul. Cependant, certains algorithmes sont plus faciles à formuler (et à implémenter) en utilisant la récursivité, et c'est ce que nous ferons quelquefois dans cet ouvrage.

ALGORITHME TRI-FUSION

Input Une liste $a_1, \ldots, a_n$ de nombres réels.

Output Une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ telle que $a_{\pi(i)} \leq a_{\pi(i+1)}$ pour tout $i = 1, \ldots, n-1$.

① **If** $n = 1$ **then** $\pi(1) := 1$ et **stop** (**return** π).

② $m := \lfloor \frac{n}{2} \rfloor$.
Soit $\rho :=$TRI-FUSION$(a_1, \ldots, a_m)$.
Soit $\sigma :=$TRI-FUSION$(a_{m+1}, \ldots, a_n)$.

③ $k := 1, l := 1$.
While $k \leq m$ et $l \leq n - m$ **do** :
If $a_{\rho(k)} \leq a_{m+\sigma(l)}$ **then** $\pi(k + l - 1) := \rho(k)$ et $k := k + 1$
else $\pi(k + l - 1) := m + \sigma(l)$ et $l := l + 1$.
While $k \leq m$ **do** : $\pi(k + l - 1) := \rho(k)$ et $k := k + 1$.
While $l \leq n - m$ **do** : $\pi(k + l - 1) := m + \sigma(l)$ et $l := l + 1$.

Comme exemple, considérons la liste «69, 32, 56, 75, 43, 99, 28». L'algorithme divise d'abord cette liste en deux listes, «69, 32, 56» et «75, 43, 99, 28» puis trie récursivement chacune des deux sous-listes. Nous obtenons les deux permutations $\rho = (2, 3, 1)$ et $\sigma = (4, 2, 1, 3)$ correspondant aux listes triées «32, 56, 69» et «28, 43, 75, 99». Ces deux listes sont alors fusionnées de la manière suivante :

					$k := 1,$	$l := 1$
$\rho(1) = 2,$	$\sigma(1) = 4,$	$a_{\rho(1)} = 32,$	$a_{\sigma(1)} = 28,$	$\pi(1) := 7,$		$l := 2$
$\rho(1) = 2,$	$\sigma(2) = 2,$	$a_{\rho(1)} = 32,$	$a_{\sigma(2)} = 43,$	$\pi(2) := 2,$	$k := 2$	
$\rho(2) = 3,$	$\sigma(2) = 2,$	$a_{\rho(2)} = 56,$	$a_{\sigma(2)} = 43,$	$\pi(3) := 5,$		$l := 3$
$\rho(2) = 3,$	$\sigma(3) = 1,$	$a_{\rho(2)} = 56,$	$a_{\sigma(3)} = 75,$	$\pi(4) := 3,$	$k := 3$	
$\rho(3) = 1,$	$\sigma(3) = 1,$	$a_{\rho(3)} = 69,$	$a_{\sigma(3)} = 75,$	$\pi(5) := 1,$	$k := 4$	
	$\sigma(3) = 1,$		$a_{\sigma(3)} = 75,$	$\pi(6) := 4,$		$l := 4$
	$\sigma(4) = 3,$		$a_{\sigma(4)} = 99,$	$\pi(7) := 6,$		$l := 5$

Théorème 1.5. *L'*ALGORITHME TRI-FUSION *répond correctement et s'exécute en un temps* $O(n \log n)$.

Preuve. Il est évident que cet algorithme répond correctement. Si $T(n)$ est le temps de calcul (nombre de pas) sur des instances ayant n nombres, observons que $T(1) = 1$ et que $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 3n + 6$. (Les constantes dans l'expression $3n + 6$ dépendent de la manière dont est défini un pas de l'algorithme.)

Nous affirmons que cela implique que $T(n) \leq 12n \log n + 1$. Le cas $n = 1$ étant trivial, nous procéderons par induction. Pour $n \geq 2$, en supposant que l'inégalité est vraie pour $1, \ldots, n-1$, nous avons

$$\begin{aligned} T(n) &\leq 12 \left\lfloor \frac{n}{2} \right\rfloor \log \left(\frac{2}{3} n \right) + 1 + 12 \left\lceil \frac{n}{2} \right\rceil \log \left(\frac{2}{3} n \right) + 1 + 3n + 6 \\ &= 12n(\log n + 1 - \log 3) + 3n + 8 \\ &\leq 12n \log n - \frac{13}{2} n + 3n + 8 \;\leq\; 12n \log n + 1, \end{aligned}$$

parce que $\log 3 \geq \frac{37}{24}$. □

Cet algorithme s'applique aussi au tri d'éléments d'un ensemble totalement ordonné, pourvu que l'on puisse comparer deux éléments quelconques en temps constant. Peut-il exister un algorithme plus rapide, disons linéaire ? Si on ne peut trouver l'ordre qu'à la suite de comparaisons successives de deux éléments, il est possible de montrer que tout algorithme nécessite au moins $\Theta(n \log n)$ comparaisons dans le pire des cas. En effet, on peut représenter le résultat d'une comparaison par zéro ou un. Le résultat de toutes les comparaisons est donc une chaîne binaire (une suite de zéro et de un). Deux ordres différents pour l'input de l'algorithme produisent deux chaînes binaires différentes (sinon on ne pourrait distinguer ces deux ordres). Pour un input ayant n éléments, il y a donc $n!$ ordres possibles et $n!$ chaînes binaires susceptibles d'être produites. Comme le nombre de chaînes binaires de longueur plus petite que $\lfloor \frac{n}{2} \log \frac{n}{2} \rfloor$ est $2^{\lfloor \frac{n}{2} \log \frac{n}{2} \rfloor} - 1 < 2^{\frac{n}{2} \log \frac{n}{2}} = (\frac{n}{2})^{\frac{n}{2}} \leq n!$, le nombre nécessaire de comparaisons est au moins $\frac{n}{2} \log \frac{n}{2} = \Theta(n \log n)$.

On voit donc que le temps de calcul de l'ALGORITHME TRI-FUSION est optimal à un facteur constant près. On peut cependant trier des entiers ou des chaînes suivant un ordre lexicographique grâce à des algorithmes linéaires ; voir l'exercice 7. Han [2004] a proposé un algorithme pour trier n entiers en $O(n \log \log n)$.

Il y a très peu de problèmes pour lesquels des bornes inférieures non triviales de ce type existent. On aura souvent besoin d'un minorant de l'ensemble des opérations élémentaires pour obtenir une borne inférieure superlinéaire.

Exercices

1. Montrer que pour tout $n \in \mathbb{N}$:

$$e \left(\frac{n}{e} \right)^n \leq n! \leq en \left(\frac{n}{e} \right)^n .$$

Indication : utiliser la relation $1 + x \leq e^x$ pour tout $x \in \mathbb{R}$.

2. Montrer que $\log(n!) = \Theta(n \log n)$.
3. Montrer que $n \log n = O(n^{1+\epsilon})$ pour tout $\epsilon > 0$.
4. Montrer que le temps de calcul de l'ALGORITHME D'ÉNUMÉRATION DES CHEMINS est $O(n \cdot n!)$.
5. Soit un algorithme dont le temps de calcul est $\Theta(n(t + n^{1/t}))$, n étant la taille de l'input et t un paramètre positif arbitraire. Comment choisir t en fonction de n pour que le temps de calcul qui est une fonction de n ait un taux de croissance minimum ?
6. Soient s, t deux chaînes binaires de longueur m. Nous dirons que s est lexicographiquement plus petite que t s'il existe un indice $j \in \{1, \ldots, m\}$ tel que $s_i = t_i$ pour $i = 1, \ldots, j-1$ et $s_j < t_j$. Soient alors n chaînes de longueur m que nous souhaitons trier suivant un ordre lexicographique. Montrer qu'on peut résoudre ce problème en un temps $O(nm)$.
   Indication : regrouper les chaînes selon leur premier bit et trier chaque groupe.
7. Proposer un algorithme qui trie une liste de nombres naturels $a_1, \ldots, a_n$, c.-à-d. qui trouve une permutation π telle que $a_{\pi(i)} \leq a_{\pi(i+1)}$ $(i = 1, \ldots, n-1)$ en un temps $O(\log(a_1 + 1) + \cdots + \log(a_n + 1))$.
   Indication : trier d'abord les chaînes codant les entiers suivant leur longueur. Appliquer ensuite l'algorithme de l'exercice 6.
   Note : l'algorithme étudié ici et dans l'exercice précédent est quelquefois appelé le tri radix.

Références

Littérature générale :

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. [2001] : Introduction to Algorithms. Second Edition. MIT Press, Cambridge 2001

Knuth, D.E. [1968] : The Art of Computer Programming ; Vol. 1. Fundamental Algorithms. Addison-Wesley, Reading 1968 (third edition : 1997)

Références citées :

Aho, A.V., Hopcroft, J.E., Ullman, J.D. [1974] : The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading 1974

Cobham, A. [1964] : The intrinsic computational difficulty of functions. Proceedings of the 1964 Congress for Logic Methodology and Philosophy of Science (Y. Bar-Hillel, ed.), North-Holland, Amsterdam 1964, pp. 24-30

Edmonds, J. [1965] : Paths, trees, and flowers. Canadian Journal of Mathematics 17 (1965), 449-467

Han, Y. [2004] : Deterministic sorting in $O(n \log \log n)$ time and linear space. Journal of Algorithms 50 (2004), 96-105

Stirling, J. [1730] : Methodus Differentialis, London 1730

Chapitre 2

Graphes

Les graphes seront utilisés tout au long de ce livre. Dans ce chapitre nous donnerons les définitions de base et nous préciserons nos notations. Nous présenterons également quelques théorèmes classiques et quelques algorithmes fondamentaux.

Après les définitions du paragraphe 2.1, nous étudierons quelques structures essentielles souvent rencontrées dans ce livre : les arbres, les cycles, les coupes. Nous démontrerons quelques propriétés importantes, et nous considérerons des systèmes d'ensembles reliés aux arbres au paragraphe 2.2. L'algorithme de recherche des composantes connexes ou fortement connexes sera présenté au paragraphe 2.3. Nous démontrerons le théorème d'Euler relatif aux parcours fermés qui passent une seule fois par chaque arête au paragraphe 2.4. Enfin, nous étudierons les graphes dessinables sur un plan sans que les arêtes se croisent aux paragraphes 2.5 et 2.6.

2.1 Définitions fondamentales

Un **graphe non orienté** est un triplet (V, E, Ψ) constitué de deux ensembles finis V et E et d'une application $\Psi : E \to \{X \subseteq V : |X| = 2\}$[1]. Un **graphe orienté** est un triplet (V, E, Ψ), constitué de deux ensembles finis V et E et d'une application $\Psi : E \to \{(v, w) \in V \times V : v \neq w\}$. V est l'ensemble des **sommets** du graphe ; E est l'ensemble de ses **arêtes** si le graphe est non orienté et de ses **arcs** s'il est orienté. Suivant un usage assez répandu, nous noterons également une arête $e = \{v, w\}$ par $e = (v, w)$ ou $e = (w, v)$.

Deux arêtes (arcs) e, e' seront dites **parallèles** si $\Psi(e) = \Psi(e')$. Un graphe sans arêtes ou arcs parallèles est un graphe **simple**. Quand un graphe est simple, nous pouvons identifier $e \in E$ avec son image $\Psi(e)$ et écrire $G = (V(G), E(G))$, avec $E(G) \subseteq \{X \subseteq V(G) : |X| = 2\}$ ou $E(G) \subseteq V(G) \times V(G)$. Nous utiliserons souvent cette notation même en présence d'arêtes (d'arcs) parallèles. Ainsi l'ensemble $E(G)$ pourra contenir plusieurs éléments «identiques». $|E(G)|$

[1] Nous utiliserons tout au long de cet ouvrage les notations ensemblistes anglophones (*ndt*).

est le nombre d'arêtes (arcs) ; si E et F sont deux ensembles d'arêtes (arcs), $|E \dot\cup F| = |E| + |F|$ même si des arêtes (arcs) parallèles apparaissent dans cette union.

Nous dirons qu'une arête (resp. un arc) $e = (v, w)$ **joint** v et w (resp. v à w) et que v et w sont **adjacents** ou mutuellement **voisins**. v et w seront les **extrémités** de e et nous dirons que v (resp. e) est **incident** à e (resp. à v). si $e = (v, w)$ est un arc, v est l'**origine** de e, w est l'**extrémité terminale** de e ; nous dirons que e est **sortant de** v et **entrant dans** w. Nous dirons aussi que v (resp. w) est le **voisin entrant** (resp. **voisin sortant**) de w (resp. v). Deux arcs ou arêtes ayant une extrémité commune seront dits **adjacents**.

La terminologie de la théorie des graphes n'est pas complètement figée. Par exemple, les sommets sont parfois appelés nœuds ou points ; en anglais, *edge* signifie arête ou arc. Un graphe avec des arêtes ou arcs parallèles est parfois appelé multigraphe. On peut également autoriser les boucles (extrémités identiques).

Si G est un graphe orienté, nous considérerons parfois son **graphe non orienté associé** G' obtenu en enlevant l'orientation de chaque arc de G. Nous dirons alors que G est une **orientation** de G'.

Un **sous-graphe** de $G = (V(G), E(G))$ est un graphe $H = (V(H), E(H))$ avec $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$. Nous dirons que G contient H. Le graphe H est un **sous-graphe induit** de G si H est un sous-graphe de G et si $E(H) = \{(x, y) \in E(G) : x, y \in V(H)\}$; $H = G[V(H)]$ est le sous-graphe de G **induit par** $V(H)$. Un sous-graphe H de G est appelé **couvrant** si $V(H) = V(G)$.

Si $v \in V(G)$, $G - v$ est le sous-graphe de G induit par $V(G) \setminus \{v\}$. Si $e \in E(G)$, $G - e := (V(G), E(G) \setminus \{e\})$ est le graphe obtenu en supprimant e de E. $G + e := (V(G), E(G) \dot\cup \{e\})$ est le graphe obtenu en ajoutant une nouvelle arête (un nouvel arc) e à E. Si G et H sont deux graphes, $G + H$ est le graphe tel que $V(G + H) = V(G) \cup V(H)$ et tel que $E(G + H)$ est l'union disjointe de $E(G)$ et $E(H)$.

Deux graphes G et H sont appelés **isomorphes** s'il existe deux bijections $\Phi_V : V(G) \to V(H)$ et $\Phi_E : E(G) \to E(H)$ telles que $\Phi_E((v, w)) = (\Phi_V(v), \Phi_V(w))$ pour tout $(v, w) \in E(G)$. Nous ne distinguerons pas deux graphes isomorphes ; ainsi nous dirons que G contient H si G a un sous-graphe isomorphe à H.

Soit G un graphe non orienté et soit $X \subseteq V(G)$. Le graphe résultant de la **contraction** de X, et noté G/X, s'obtient en supprimant X et les arêtes de $G[X]$, puis en ajoutant un nouveau sommet x et en remplaçant enfin chaque arête (v, w) avec $v \in X$, $w \notin X$ par une arête (x, w) (notons que cette construction pourra créer des arêtes parallèles). Cette définition s'étend naturellement aux graphes orientés.

Soit un graphe G et $X, Y \subseteq V(G)$. Nous poserons : $E(X, Y) := \{(x, y) \in E(G) : x \in X \setminus Y,\ y \in Y \setminus X\}$ si G est non orienté et $E^+(X, Y) := \{(x, y) \in E(G) : x \in X \setminus Y,\ y \in Y \setminus X\}$ si G est orienté. Si G est non orienté et $X \subseteq V(G)$ nous poserons $\delta(X) := E(X, V(G) \setminus X)$. L'**ensemble des voisins** de X est défini par $\Gamma(X) := \{v \in V(G) \setminus X : E(X, \{v\}) \neq \emptyset\}$. Si G est orienté et $X \subseteq V(G)$ nous poserons : $\delta^+(X) := E^+(X, V(G) \setminus X)$, $\delta^-(X) := \delta^+(V(G) \setminus X)$ et $\delta(X) :=$

$\delta^+(X) \cup \delta^-(X)$. Nous utiliserons des indices (par exemple $\delta_G(X)$) pour spécifier le graphe G, si nécessaire.

Pour les ensembles de sommets ayant un seul élément $\{v\}$ ($v \in V(G)$), que nous appellerons aussi **singletons**, nous écrirons $\delta(v) := \delta(\{v\})$, $\Gamma(v) := \Gamma(\{v\})$, $\delta^+(v) := \delta^+(\{v\})$ et $\delta^-(v) := \delta^-(\{v\})$. Le **degré** d'un sommet v est $|\delta(v)|$, nombre d'arêtes incidentes à v. Dans le cas orienté, le **degré entrant** est $|\delta^-(v)|$, le **degré sortant** est $|\delta^+(v)|$, et le degré est $|\delta^+(v)|+|\delta^-(v)|$. Un sommet v de degré 0 est appelé **isolé**. Un graphe dont tous les sommets ont degré k est appelé ***k*-régulier**.

Si G est quelconque, $\sum_{v\in V(G)} |\delta(v)| = 2|E(G)|$. En particulier le nombre de sommets de G de degré impair est pair. Si G est orienté, $\sum_{v\in V(G)} |\delta^+(v)| = \sum_{v\in V(G)} |\delta^-(v)|$. Pour montrer ces relations, observons que chaque arc ou arête est compté deux fois dans chacun des membres de la première équation et que chaque arc est compté une fois dans chacun des membres de la deuxième équation. On peut aussi démontrer :

Lemme 2.1. *Soit G un graphe orienté et soient $X, Y \subseteq V(G)$:*

(a) $|\delta^+(X)|+|\delta^+(Y)| = |\delta^+(X\cap Y)|+|\delta^+(X\cup Y)|+|E^+(X,Y)|+|E^+(Y,X)|$;

(b) $|\delta^-(X)|+|\delta^-(Y)| = |\delta^-(X\cap Y)|+|\delta^-(X\cup Y)|+|E^+(X,Y)|+|E^+(Y,X)|$.

Soit G est un graphe non orienté et soient $X, Y \subseteq V(G)$:

(c) $|\delta(X)| + |\delta(Y)| = |\delta(X \cap Y)| + |\delta(X \cup Y)| + 2|E(X,Y)|$;

(d) $|\Gamma(X)| + |\Gamma(Y)| \geq |\Gamma(X \cap Y)| + |\Gamma(X \cup Y)|$.

Preuve. Il suffit d'utiliser des arguments de comptage. Soit $Z := V(G)\setminus(X\cup Y)$. Pour (a), observons que $|\delta^+(X)| + |\delta^+(Y)| = |E^+(X,Z)| + |E^+(X,Y \setminus X)| + |E^+(Y,Z)| + |E^+(Y,X \setminus Y)| = |E^+(X\cup Y,Z)| + |E^+(X\cap Y,Z)| + |E^+(X,Y \setminus X)| + |E^+(Y,X \setminus Y)| = |\delta^+(X\cup Y)| + |\delta^+(X\cap Y)| + |E^+(X,Y)| + |E^+(Y,X)|$. (b) se déduit de (a) en inversant l'orientation de chaque arc (remplacer (v,w) par (w,v)). (c) se déduit de (a) en remplaçant chaque arête (v,w) par une paire d'arcs de directions opposées (v,w) et (w,v).

Pour montrer (d), observons que $|\Gamma(X)| + |\Gamma(Y)| = |\Gamma(X \cup Y)| + |\Gamma(X) \cap \Gamma(Y)| + |\Gamma(X) \cap Y| + |\Gamma(Y) \cap X| \geq |\Gamma(X \cup Y)| + |\Gamma(X \cap Y)|$. □

Une fonction $f : 2^U \to \mathbb{R}$ (où U est un ensemble fini et 2^U est l'ensemble des parties de U) est appelée :

- **sous-modulaire** si $f(X\cap Y)+f(X\cup Y) \leq f(X)+f(Y)$ pour tout $X, Y \subseteq U$;
- **supermodulaire** si $f(X\cap Y)+f(X\cup Y) \geq f(X)+f(Y)$ pour tout $X, Y \subseteq U$;
- **modulaire** si $f(X \cap Y) + f(X \cup Y) = f(X) + f(Y)$ pour tout $X, Y \subseteq U$.

Le lemme 2.1 implique que $|\delta^+|$, $|\delta^-|$, $|\delta|$ et $|\Gamma|$ sont sous-modulaires. Cela sera utile ultérieurement.

Un **graphe complet** est un graphe simple non orienté tel que toute paire de sommets est adjacente. Le graphe complet à n sommets sera noté K_n. Le **complément** d'un graphe simple non orienté G est le graphe H tel que $G + H$ est un graphe complet.

Un **couplage** d'un graphe non orienté G est un ensemble d'arêtes deux à deux non adjacentes (c.-à-d. ayant toutes leurs extrémités différentes). Une **couverture par les sommets** de G est un ensemble $S \subseteq V(G)$ tel que chaque arête de G soit incidente à au moins un sommet dans S. Une **couverture par les arêtes** de G est un ensemble $F \subseteq E(G)$ d'arêtes tel que chaque sommet de G soit incident à au moins une arête de F. Un **ensemble stable** dans G est un ensemble de sommets deux à deux non adjacents. Un graphe sans aucune arête (ou arc) est dit **vide**. Une **clique** est un ensemble de sommets deux à deux adjacents.

Proposition 2.2. *Soit un graphe G et $X \subseteq V(G)$. Les propositions suivantes sont équivalentes :*

(a) *X est une couverture par les sommets dans G.*

(b) *$V(G) \setminus X$ est un ensemble stable dans G.*

(c) *$V(G) \setminus X$ est une clique dans le complément de G.* □

Si $\mathcal{F}$ est une famille d'ensembles ou de graphes, nous dirons que F est un élément **minimal** de $\mathcal{F}$ si $\mathcal{F}$ contient F, mais aucun sous-ensemble/sous-graphe propre de F. De même, F est **maximal** dans $\mathcal{F}$ si $F \in \mathcal{F}$ et F n'est pas un sous-ensemble/sous-graphe propre d'un élément de $\mathcal{F}$. Un élément **minimum** ou **maximum** est un élément de cardinalité minimum/maximum.

Une couverture par les sommets minimale n'est pas forcément minimum (voir par exemple figure 13.1), et un couplage maximal n'est en général pas maximum. Les problèmes de la recherche d'un couplage, d'un ensemble stable ou d'une clique maximum, de la couverture par les sommets ou par les arêtes minimum dans un graphe non orienté auront une grande importance dans la suite de ce livre.

Le **line graph**[2] d'un graphe simple non orienté G est le graphe $(E(G), F)$, tel que $F = \{(e_1, e_2) : e_1, e_2 \in E(G), |e_1 \cap e_2| = 1\}$. Notons que les couplages du graphe G correspondent aux ensembles stables du *line graph* de G.

Soit G un graphe orienté ou non. Une suite $P = [v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}]$ est un **parcours de v_1 à v_{k+1}** de G si $k \geq 0$ et si les deux **extrémités** de e_i sont v_i et v_{i+1} pour $i = 1, \ldots, k$. v_1 et v_{k+1} sont les **extrémités** de P. Si G est orienté, nous dirons que P est un **parcours orienté** quand l'arc e_i est sortant de v_i et entrant dans v_{i+1} pour $i = 1, \ldots, k$. v_1 sera l'**origine** du parcours P et v_{k+1} sera son **extrémité**. Si $e_i \neq e_j$ pour $1 \leq i < j \leq k$, P est un **parcours simple** de G. Un parcours simple P est **fermé** si $v_1 = v_{k+1}$. Un parcours simple P tel que $v_i \neq v_j$ pour $1 \leq i < j \leq k+1$ est une **chaîne**. Dans ce cas on pourra identifier P avec le sous-graphe $(\{v_1, \ldots, v_{k+1}\}, \{e_1, \ldots, e_k\})$ de G. Si P est une chaîne et $x, y \in V(P)$ $P_{[x,y]}$ sera l'(unique) sous-graphe de P qui est une chaîne de x à y. Il est évident qu'il existe un parcours d'un sommet v à un sommet $w \neq v$ si et seulement s'il existe une chaîne de v à w.

[2] En français «graphe représentatif des arêtes d'un graphe» ; nous garderons ici l'expression anglaise, plus concise et plus facile à utiliser (*ndt*).

Un parcours orienté qui est également une chaîne est un **chemin**. Deux sommets d'un graphe non orienté sont **connectés** s'il existe une chaîne ayant ces deux sommets comme extrémités ; un sommet t est **connecté** à un sommet s dans un graphe orienté s'il existe un chemin d'origine s et d'extrémité t.

Un graphe $(\{v_1, \ldots, v_k\}, \{e_1, \ldots, e_k\})$ tel que $v_1, e_1, v_2, \ldots, v_k, e_k, v_1$ est un parcours (resp. un parcours orienté) avec $v_i \neq v_j$ si $1 \leq i < j \leq k$ est un **cycle** (resp. un **circuit**). Un argument simple d'induction montre que l'ensemble des arêtes (resp. des arcs) d'un parcours fermé (resp. d'un parcours orienté fermé) peut être partitionné en ensembles d'arêtes de cycles (resp. d'arcs de circuits).

La **longueur** d'une chaîne (resp. d'un chemin) est son nombre d'arêtes (resp. d'arcs). Une chaîne qui couvre les sommets d'un graphe non orienté G est appelée **chaîne hamiltonienne** de G ; un cycle couvrant les sommets de G est appelé **cycle hamiltonien** ou **tour** de G. Un graphe contenant un cycle hamiltonien est appelé **graphe hamiltonien**. Les chemins et circuits hamiltoniens se définissent de la même manière quand les graphes sont orientés.

Si v et w sont deux sommets de G, la **distance** de v à w notée $\mathrm{dist}(v, w)$ ou $\mathrm{dist}_G(v, w)$ est la longueur d'un plus court chemin de v à w si G est orienté, et d'une plus courte chaîne de v à w si G est non orienté. S'il n'existe aucune chaîne (ou chemin dans le cas orienté) de v à w, nous poserons $\mathrm{dist}(v, w) := \infty$. Si G est non orienté, on a toujours $\mathrm{dist}(v, w) = \mathrm{dist}(w, v)$ pour toute paire $v, w \in V(G)$.

Souvent une fonction coût $c : E(G) \to \mathbb{R}$ sera associée aux problèmes que nous étudierons. Si $F \subseteq E(G)$, nous poserons $c(F) := \sum_{e \in F} c(e)$ (et $c(\emptyset) = 0$). La fonction $c : 2^{E(G)} \to \mathbb{R}$ est une fonction modulaire. $\mathrm{dist}_{(G,c)}(v, w)$ sera le minimum de $c(E(P))$ pour toutes les chaînes (chemins) P de v à w.

2.2 Arbres, cycles, coupes

Un graphe orienté ou non orienté G sera dit **connexe** s'il existe une chaîne de v à w pour tous $v, w \in V(G)$; sinon G sera **non connexe**. Les sous-graphes connexes maximaux de G sont les **composantes connexes** de G. Nous identifierons quelquefois les composantes connexes avec les ensembles de sommets qu'elles induisent. Un ensemble de sommets X est connexe si le sous-graphe induit par X est connexe. v est un **sommet d'articulation** si $G - v$ a plus de composantes connexes que G ; $e \in E(G)$ est un **pont** si $G - e$ a plus de composantes connexes que G.

Un graphe non orienté sans cycles est une **forêt**. Une forêt connexe est un **arbre**. Dans un arbre, une **feuille** est un sommet de degré 1. Une **étoile** est un arbre ayant au plus un sommet qui n'est pas une feuille.

Nous allons maintenant donner quelques propriétés des arbres et des arborescences, leurs analogues dans les graphes orientés. Établissons le résultat suivant :

Proposition 2.3.

(a) *Un graphe orienté ou non orienté G est connexe si et seulement si $\delta(X) \neq \emptyset$ pour tout $\emptyset \neq X \subset V(G)$.*

(b) *Soit G un graphe orienté et soit $r \in V(G)$. Il existe un chemin de r à tout sommet $v \in V(G)$ si et seulement si $\delta^+(X) \neq \emptyset$ pour tout $X \subset V(G)$ contenant r.*

Preuve. (a) : s'il existe $X \subset V(G)$ tel que $r \in X$, $v \in V(G) \setminus X$, et $\delta(X) = \emptyset$, il ne peut exister de chaîne de r à v et G n'est pas connexe. Inversement, si G n'est pas connexe, il existe deux sommets r, v non connectés. Si R est l'ensemble des sommets connectés à r, $r \in R$, $v \notin R$ et $\delta(R) = \emptyset$.

(b) : la preuve est analogue. □

Théorème 2.4. *Soit G un graphe non orienté connexe ayant n sommets. Les propositions suivantes sont équivalentes :*

(a) *G est un arbre (G est connexe et sans cycles).*

(b) *G est sans cycles et a $n-1$ arêtes.*

(c) *G est connexe et a $n-1$ arêtes.*

(d) *G est un graphe connexe minimal (chaque arête est un pont).*

(e) *G est un graphe minimal vérifiant la propriété $\delta(X) \neq \emptyset$ pour tout $\emptyset \neq X \subset V(G)$.*

(f) *G est un graphe maximal sans cycles (l'addition d'une arête quelconque crée un cycle).*

(g) *Toute paire de sommets de G est connectée par une chaîne unique.*

Preuve. (a)⇒(g), car l'union de deux chaînes distinctes ayant les mêmes extrémités contient un cycle.

(g)⇒(e)⇒(d) se déduit de la proposition 2.3(a).

(d)⇒(f) : évident.

(f)⇒(b)⇒(c) : cela se déduit du fait que si une forêt a n sommets, m arêtes et p composantes connexes, alors $n = m + p$. (Preuve par induction sur m.)

(c)⇒(a) : soit G connexe ayant $n-1$ arêtes. Si G a un cycle, on peut le supprimer en en retirant une arête. Supposons qu'après avoir retiré k arêtes de cette manière, le graphe résultant G' soit connexe et sans cycles. G' a $m = n-1-k$ arêtes. Comme $n = m + p = n - 1 - k + 1$, on a $k = 0$. □

En particulier, (d)⇒(a) signifie qu'un graphe est connexe si et seulement s'il contient un **arbre couvrant** (un sous-graphe couvrant qui est un arbre).

Un graphe orienté est une **ramification** si le graphe non orienté associé est une forêt et si chaque sommet v a au plus un arc entrant. Une ramification connexe est une **arborescence**. Par le théorème 2.4, une arborescence ayant n sommets a $n-1$ arcs et par conséquent il n'existe qu'un sommet r avec $\delta^-(r) = \emptyset$. Ce sommet est appelé la **racine** de l'arborescence ; nous dirons aussi que l'arborescence est **enracinée en** r. Les **feuilles** sont les sommets v qui vérifient $\delta^+(v) = \emptyset$.