Conception et analyse d'algorithmes

Robert Cori, Guillaume Hanrot et Jean-Marc Steyaert

Table des matières

Introduction

Ce cours est principalement consacré à l'étude de la complexité des algorithmes de nature combinatoire et à quelques applications. Ce domaine est particulièrement vivant depuis les années 70 après la mise en évidence de la classe des algorithmes **NP** et du célèbre problème $\mathbf{P} = \mathbf{NP}$?. En effet, on peut dire que dans l'état actuel de nos connaissances théoriques et étant donné l'état de l'art en matière de conception et de réalisation de machines, seuls les algorithmes dont le comportement à l'exécution demande un nombre d'opérations majoré par une fonction polynomiale de la taille des données sont utilisables dans la pratique : ils constituent la classe **P**.
Le fait de ne pas connaître d'algorithmes en temps polynomial pour résoudre certains problèmes d'optimisation combinatoire est à la fois une question intrigante sur le plan conceptuel, puisqu'on n'a pas réussi à démontrer que de tels algorithmes n'existaient pas et cruciale sur le plan pratique puisque de nombreux problèmes de cette famille se posent tous les jours dans les applications et que l'on espère pouvoir les résoudre effectivement : un grand nombre de ces problèmes sont réunis dans la classe **NP**.

On verra que placé devant l'impossibilité de répondre à la question théorique posée et n'ayant pas à sa disposition des algorithmes exacts efficaces, l'utilisateur se retourne vers des algorithmes probabilistes permettant en un temps polynomial d'obtenir une solution ayant une probabilité raisonable d'être optimale ou des algorithmes approchés qui donnent, avec une quasi certitude, une solution située à une faible distance de l'optimum.

Au long des neuf chapitres qui composent ce cours on s'intéresse d'abord à des méthodes qui donnent naissance à des algorithmes efficaces : la stratégie gloutonne et la programmation dynamique permettent ainsi de résoudre bon nombre de problèmes de façon élégante ; ces deux procédés font l'objet des premiers chapitres. On s'intéresse ensuite à la programmation linéaire et à ses diverses facettes ; une mention particulière doit être faite aux problèmes qui s'expriment comme la recherche d'un flot optimal dans un réseau, pour lesquels des techniques efficaces sont connues et dont les applications sont nombreuses en particulier en matière de certains problèmes d'affectation optimale.
Les questions générales de complexité sont abordées sous l'angle de classification des problèmes pratiques : on illustrera sur des exemples des propriétés partagées par tous les systèmes de calcul et on accordera un peu de temps au problème fameux $\mathbf{P} = \mathbf{NP}$? On abordera rapidement le concept d'algorithme probabiliste, qui constitue le développement le plus récent du domaine.
Pour terminer on s'interessera à deux domaines d'applications des techniques mises en

oeuvre : d'une part les algorithmes sur les graphes planaires et leur dessin, et d'autre part les problèmes de nature algorithmique qui se posent en physique statistique.

Nous pensons que le lecteur aura ainsi développé un point de vue sur une problématique riche et utile, qui fait aujourd'hui l'objet de nombreux travaux de recherche et de développements industriels.

Terminologie de théorie des graphes

Un *graphe orienté* est donné par un ensemble X de *sommets* et un ensemble A d'*arcs* ; chaque arc est un couple de sommets. Les définitions de base sur les graphes orientés sont données dans le polycopié de tronc commun d'informatique : algorithmes et programmation.

Un *graphe symétrique* $G = (X, E)$ (on dit aussi non-orienté) est constitué d'un ensemble (fini) X de *sommets* et d'un ensemble E *d'arêtes*, chaque arête étant une paire de sommets. On note en général n le nombre de sommets et m le nombre d'arêtes d'un graphe. On peut aussi considérer un graphe symétrique comme un graphe orienté (X, A) tel que pour tout arc $a \in A$, il existe un arc opposé $\bar{a}$ dont l'origine est l'extrémité de a et dont l'extrémité est l'origine de a. Le couple $\{a, \bar{a}\}$ forme alors une arête. La terminologie sur les graphes *orientés* est est donnée dans le chapitre 5 du polycopié du cours de tronc commun d'informatique.

Un *chemin* dans un graphe symétrique est une suite d'arêtes $e_1, e_2, \ldots, e_k$ telle qu'il existe une suite de sommets $x_1, x_2, \ldots, x_k, x_{k+1}$ vérifiant $e_i = \{x_i, x_{i+1}\}$, on dit que le chemin *joint* x_1 à x_{k+1} ; k est la longueur du chemin. Le chemin est dit *élémentaire* si les $x_1, x_2, \ldots, x_k$ sont distincts. Un chemin est un *cycle* si $x_1 = x_{k+1}$ (on dit aussi parfois *circuit* mais nous reservons ce mot aux graphes orientés). Un graphe est *connexe* si pour tout couple de sommets $x, y \in X$, il existe un chemin qui joint x à y.

Un *arbre* est un graphe symétrique connexe et sans cycle. Dans un arbre on a $m = n-1$; d'autre part pour tout couple de sommets il existe un chemin élémentaire unique qui les joint. On peut montrer que les conditions suivantes sont équivalentes pour un graphe symétrique $G = (X, T)$:

(1) $G = (X, T)$ est connexe et sans cycle

(2) $G = (X, T)$ est connexe et $X = |T| + 1$

(3) $G = (X, T)$ est sans cycle et $X = |T| + 1$

(4) $G = (X, T)$ est tel que tout couple de sommets est relié par un chemin unique.

Les arbres utilisés en structure de données sont légèrement différents de ceux considérés ici dans la mesure où une racine est distinguée, ce qui induit la notion de père et de fils qui est absente ici.

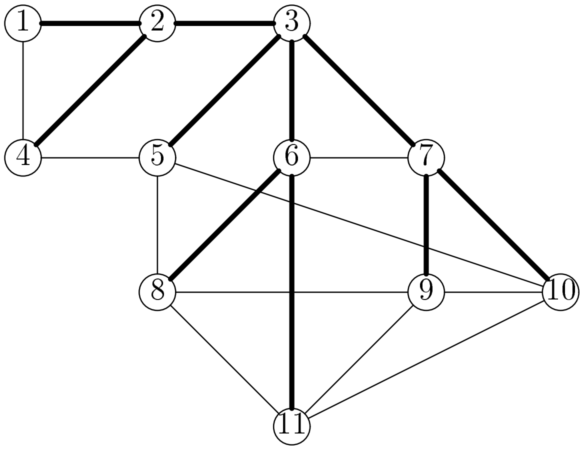


FIG. 1. Un graphe et un de ses arbres recouvrants en traits gras

Pour un graphe symétrique $G = (X, E)$ quelconque, un *arbre recouvrant* est donné par un sous-ensemble de l'ensemble des arêtes qui forme un arbre ayant X pour ensemble de sommets (voir figure). Pour posséder un arbre recouvrant, un graphe doit être connexe. Les algorithmes de parcours de graphes vus en Algorithmique et Programmation permettent de construire des arbres recouvrants. Le nombre d'arbres recouvrants d'un graphe est en général une fonction exponentielle de n : en particulier un graphe dans lequel toutes les paires de sommets forment une arête possède n^{n-2} arbres recouvrants.

Si $G = (X, E)$ est un graphe symétrique et $A = (X, T)$ un arbre recouvrant de G, toute arête de G qui n'est pas dans T crée un cycle unique avec des arêtes de T, on l'appelle parfois *cycle principal*. Il y a $m-n+1$ telles arêtes et donc $m-n+1$ cycles principaux. Ces cycles permettent de construire tous les autres cycles du graphe par différence symétrique.

Plusieurs sous-ensembles de sommets ou d'arêtes d'un graphe symétrique jouent un rôle particulier . Un *stable* est un sous-ensemble S de X tel que deux sommets de S ne sont pas joints par une arête. Soit formellement

$$\{x, y\} \in E, \quad x \in S \quad \Rightarrow \quad y \notin S$$

Un ensemble *couvrant* Y de sommets est un sous-ensemble de X tel que toute arête possède au moins une extremité dans Y, soit :

$$\{x, y\} \in E, \quad x \notin Y \quad \Rightarrow \quad y \in Y$$

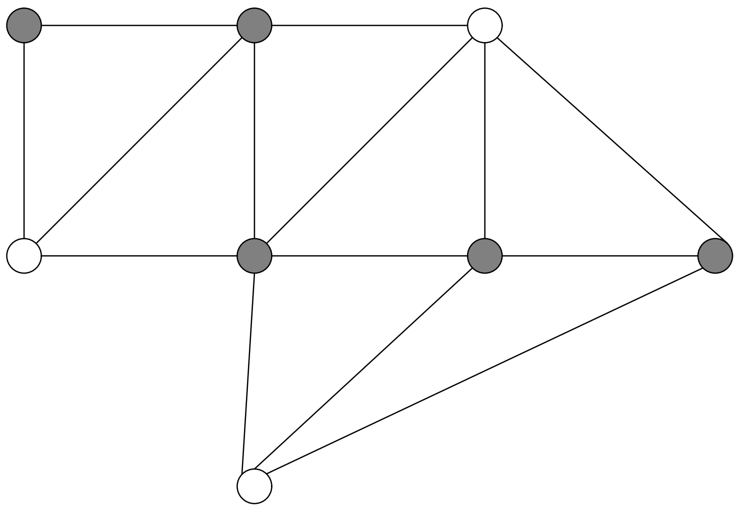


FIG. 2. Un graphe et un de ses ensembles stables en clair et couvrants en noir

Un *couplage* est un sous-ensemble C de E tel que deux arêtes de C ne possèdent pas d'extrémité commune, soit :

$$e, f \in C, \quad \Rightarrow \quad e \cap f = \emptyset$$

La notion de *coloration* joue un rôle important en théorie des graphes, il s'agit d'affecter des couleurs aux sommets de façon telle que deux sommets reliés par une arête aient des couleurs différentes. Formellement une k-coloration est une application γ de X dans l'ensemble $\{1, 2, \ldots, k\}$ telle que :

$$\{x, y\} \in E \quad \Rightarrow \quad \gamma(x) \neq \gamma(y)$$

Le plus petit entier k tel qu'il existe une k-coloration de G est appelé *nombre chromatique* de G.

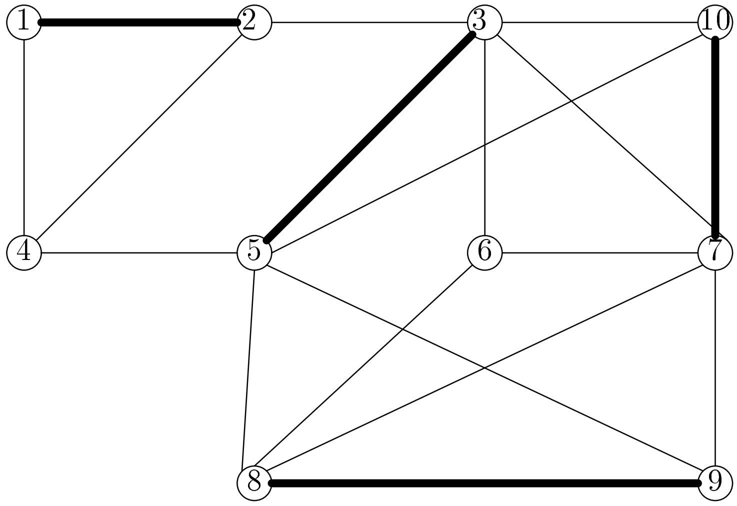


Fig. 3. Un graphe et un de ses couplage en traits gras

Première partie

Algorithmes

CHAPITRE 1

Algorithmes gloutons et matroïdes

De nombreux problèmes d'optimisation combinatoire se présentent sous la forme suivante :

> On se donne un ensemble E fini et à chaque élément e de E est affectée une valeur $v(e)$ qui est un réel positif ; on se donne de plus une condition C sur l'ensemble des parties de E. Le problème consiste à construire un sous-ensemble F de E tel que :
>
> $\sum_{e \in F} v(e)$ est maximal parmi les parties F satisfaisant C.

Dans plusieurs cas la condition C est telle que si F satisfait C alors il en est de même pour tout sous-ensemble de F. Il est alors naturel de proposer un algorithme très simple consistant à initialiser F par $F = \emptyset$, puis à ajouter successivement des éléments à F suivant un certain critère (par exemple par valeurs décroissantes), jusqu'à obtenir un ensemble auquel on ne peut plus ajouter d'élément sans contredire C. On appelle *algorithme glouton* ce type d'algorithme, ce qualificatif décrit le fait qu'on se précipite sur l'élément le plus intéressant en premier sans se préoccuper des conséquences de ce choix pour la suite.

Cet algorithme donne très rapidement un résultat, en revanche ce résultat n'est pas toujours la solution optimale. Nous allons voir que ce n'est le cas que pour des familles de parties satisfaisant une condition particulière, on appelle ces familles *matroïdes*. L'étude de la structure algébrique des matroïdes a fait l'objet de plusieurs travaux en combinatoire car elle donne un cadre mathématique unificateur à des résultats de théorie des graphes et d'algorithmique. Plusieurs ouvrages lui ont été consacrés, nous recommandons en particulier celui de D. Welsh [**19**] ; une bonne introduction au sujet est aussi donnée par le chapitre 17 du manuel [**4**].

Les questions relatives à l'affectation d'une ou plusieurs ressources à des utilisateurs (clients, processeurs, etc.) constituent une classe importante de problèmes qu'il est tentant de résoudre par un algorithme glouton. Nous allons montrer que dans certains cas très simples, l'algorithme glouton donne une solution optimale. Dans des cas plus complexes, l'algorithme donne une solution approchée, dont on se contente parfois, vu le temps de calcul prohibitif de la recherche de l'optimum exact. Après avoir traité ce cas simple nous considérons un problème classique sur les graphes la recherche d'un arbre recouvrant de poids maximal (ou de manière équivalente minimal). La résolution de celui-ci par l'algorithme glouton amène à la structure de matroïde dont on donne quelques propriétés.

1. Exemple élémentaire

Le problème décrit précisément ci-dessous peut être résolu par l'algorithme glouton (mais, comme on le verra, l'algorithme glouton ne donne pas la solution optimale pour une autre formulation du problème, pourtant proche de celle-ci). Il s'agit d'affecter une ressource unique, non partageable, successivement à un certain nombre d'utilisateurs qui en font la demande en précisant la période exacte pendant laquelle ils souhaitent en disposer.

On peut illustrer ceci en considérant la question de la location d'une voiture. Des clients formulent un ensemble de demandes de location et, pour chaque demande sont donnés le jour du début de la location et le jour de restitution du véhicule ; le but est d'affecter le véhicule de façon à satisfaire le maximum de clients (et non pas de maximiser la somme des durées de location). On vérifie que ce problème rentre dans le cadre général considéré plus haut. L'ensemble E est celui des demandes de location, pour chaque élément e de E, on note $d(e)$ la date du début de la location et $f(e) > d(e)$ la date de fin. La valeur $v(e)$ de tout élément e de E est égale à 1 et la contrainte à respecter pour le sous-ensemble F à construire est la suivante, (cette contrainte exprime que deux clients ne peuvent disposer en même temps du véhicule) :

$$\forall e_1, e_2 \in F \quad d(e_1) \leq d(e_2) \Rightarrow f(e_1) \leq d(e_2)$$

L'algorithme glouton est décrit comme suit :

- *Etape 1 :*
  Classer les éléments de E par ordre de dates de fin croissantes. Les éléments de E forment alors une suite $e_1, e_2, \ldots, e_n$ telle que $f(e_1) \leq f(e_2) \leq \ldots \leq f(e_n)$
- *Etape 2 :*
  Initialiser $F := \emptyset$
  Pour i variant de 1 à n, ajouter la demande e_i à F si celle-ci ne chevauche pas la dernière demande appartenant à F.

Montrons que l'algorithme glouton donne la solution optimale.
Soit $F = \{x_1, x_2, \ldots, x_p\}$ la solution obtenue par l'algorithme glouton et soit $G = \{y_1, y_2, \ldots, y_q\}, q \geq p$ une solution optimale. Dans G et dans F nous supposons que les x_i et y_j sont classées par dates de fin croissantes. Nous allons montrer que $p = q$. Soit k le plus petit entier tel que $x_k \neq y_k$; ainsi $\forall i < k$, on a $x_i = y_i$. Par construction de F on a alors $f(y_k) \geq f(x_k)$. On peut alors remplacer G par $G' = \{y_1, y_2, \ldots y_{k-1}, x_k, y_{k+1}, y_q\}$ tout en satisfaisant à la contrainte de non chevauchement des demandes, ainsi G' est une solution optimale, de même cardinalité que G, ayant plus d'éléments en commun avec F que n'en avait G. En répétant cette opération suffisamment de fois on trouve un ensemble H de même cardinalité que G et qui contient F. L'ensemble H ne peut contenir d'autres éléments car ceux-ci auraient été ajoutés à F par l'algorithme glouton, ceci montre bien que $p = q$.

Remarques

1. Noter que le choix de classer les demandes par dates de fin croissantes est important. Si on les avait classées, par exemple, par dates de début croissantes, on n'aurait pas obtenu le résultat. On le voit sur l'exemple suivant avec trois demandes e_1, e_2, e_3 dont les dates de début et de fin sont données par le tableau suivant :

	e_1	e_2	e_3
d	2	3	5
f	8	4	8

Bien entendu, pour des raisons évidentes de symétrie, le classement par dates de début décroissantes donne aussi un résultat optimal.

2. On peut noter aussi que si le but est de maximiser la durée totale de location du véhicule l'algorithme décrit ci-dessus ne donne pas l'optimum. En particulier, il ne considérera pas comme prioritaire une demande de location de durée très importante. Une idée consiste à le modifier en classant les demandes par durées décroissantes et d'appliquer l'algorithme glouton, malheureusement cette technique ne donne pas non plus le bon résultat (il suffit de considérer une demande de location de 3 jours et deux demandes qui ne se chevauchent pas mais qui sont incompatibles avec la première chacune de durée égale à 2 jours).

3. S'il y a plus d'une ressource à affecter, par exemple deux voitures à louer, l'algorithme glouton consistant à classer les demandes suivant les dates de fin et à affecter la première ressource disponible, ne donne pas l'optimum, comme le montre l'exemple suivant :

	e_1	e_2	e_3	e_4
d	1	2	6	4
f	3	5	7	8

En effet les deux premières demandes sont satisfaites par des véhicules différents si on attribue le premier véhicule à la troisième demande, il n'est plus possible de satisfaire la quatrième.

2. Arbre de poids maximal d'un graphe

2.1. Exposé du problème. Nous renvoyons aux préliminaires sur les graphes pour la terminologie utilisée ici.

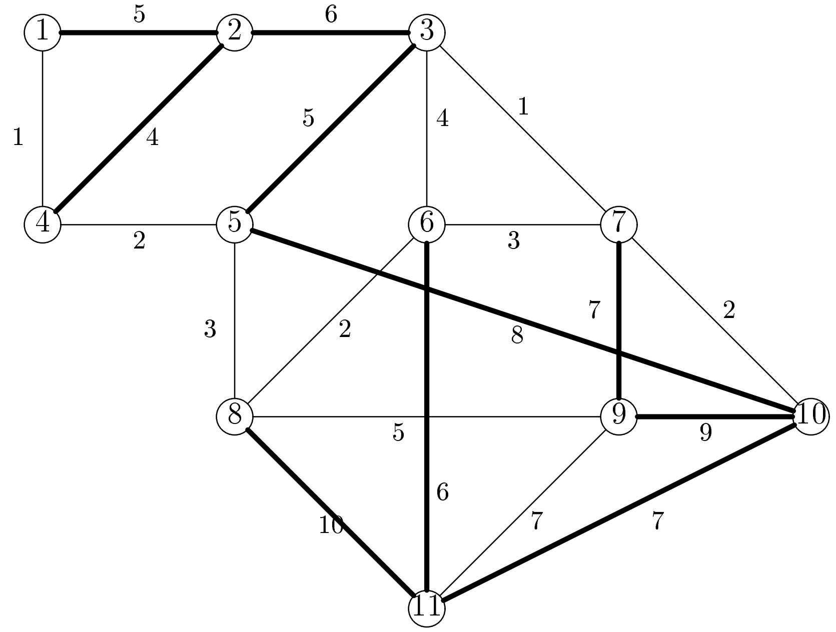


FIG. 1. Un arbre recouvrant de poids maximum

Un exemple classique d'utilisation de l'algorithme glouton est la recherche d'un arbre recouvrant de poids maximal dans un graphe symétrique, il prend dans ce cas particulier le nom d'algorithme de Kruskal. Ce problème se rencontre de fait en pratique sous la forme de la recherche d'un arbre de coût minimal pour réaliser une connexion de tous les points d'un réseau ; nous avons choisi de conserver une cohérence générale à ce chapitre en présentant tous les problèmes sous forme de recherche d'un objet qui rend maximal une certaine fonction, ceci n'a toutefois pas une grande importance car il est facile dans ce cas de transformer une recherche d'un poids maximal en celle d'un coût minimal par une transformation simple. Décrivons brièvement le problème et l'algorithme.

On affecte aux arêtes d'un graphe symétrique un poids qui est un entier positif, le poids d'un arbre est égal à la somme des poids des arêtes qui le composent ; on cherche un arbre recouvrant de poids maximal. Une illustration de ce problème est la réalisation d'un réseau électrique ou informatique entre différents points, deux points quelconques doivent toujours être reliés entre eux (connexité) et on doit minimiser le coût de la réalisation. Le poids d'une arête est, dans ce contexte, la différence entre un entier constant assez grand et le coût de construction de la portion du réseau reliant ses deux extrémités. L'arbre de poids maximal va ainsi donner le réseau dont le coût de construction est minimal.

On peut facilement formuler le problème dans le cadre général donné en début de chapitre : E est l'ensemble des arêtes du graphe, la condition C à satisfaire par F est de former un graphe sans cycle, enfin il faut maximiser la somme des poids des éléments de F.

Un exemple d'arbre recouvrant de poids maximal est donné sur la figure 1.

2.2. Algorithme de Kruskal. Ce problème peut être résolu très efficacement par l'algorithme glouton suivant :

– *Etape 1 :*

Classer les arêtes par ordre de poids décroissants. Elles constituent alors une suite

$$e_1, e_2, \ldots, e_n$$

telle que

$$v(e_1) \geq v(e_2) \geq \ldots \geq v(e_n)$$

– *Etape 2 :*

Initialiser $F := \emptyset$

Pour i variant de 1 à n, ajouter l'arête e_i à F si celle-ci ne crée pas de cycle avec celles appartenant à F.

On montre que l'algorithme glouton donne l'arbre de poids maximal en utilisant la propriété suivante des arbres recouvrants d'un graphe :

LEMME 1.1. *Soient T et U deux arbres recouvrants d'un graphe G et soit a une arête de U qui n'est pas dans T. Alors il existe une arête b du cycle γ créé par a dans T telle que $U \setminus \{a\} \cup \{b\}$ soit aussi un arbre recouvrant de G.*

Preuve : La suppression de a dans l'arbre U divise cet arbre en deux composantes connexes U_1 et U_2. Le cycle γ créé par a dans T contient nécessairement un nombre pair d'arêtes reliant ces deux composantes, comme a est une telle arête il en existe une autre dans T. On note b cette arête et on vérifie qu'elle satisfait bien aux conditions du lemme.

Preuve de validité de l'algorithme : Soit T l'arbre donné par l'algorithme glouton. Si l'algorithme ne donne pas l'optimimum c'est qu'il existe des arbres recouvrants de poids supérieurs à celui de T. Notons U celui parmi ceux-ci dont le nombre d'arêtes communes avec T est maximum. Soit a l'arête de U de plus grand poids qui n'est pas dans T, et soit b l'arête de T donnée par le lemme. Comme a n'a pas été choisie par l'algorithme glouton c'est qu'elle crée un cycle avec les éléments de T, formé d'arêtes toutes de poids supérieur ou égal à $v(a)$. Puisque b est dans ce cycle on a $v(b) \geq v(a)$, ainsi $v(U - \{a\} \cup \{b\}) \geq v(U) > v(T)$, mais $U - \{a\} \cup \{b\}$ est un arbre qui a une arête en plus en commun avec T que n'en a U ce qui contredit le choix de U. □

3. Condition d'optimalité de l'algorithme glouton

Les deux problèmes considérés ci-dessus peuvent être formulés dans un cadre général. On se donne un ensemble E fini et une famille non vide $\mathcal{F}$ de parties de E qui satisfait la condition suivante :

$$F \in \mathcal{F}, F' \subset F \Rightarrow F' \in \mathcal{F}$$

On se donne aussi une application v de E dans l'ensemble des entiers naturels.

On cherche $F \in \mathcal{F}$ tel que

– $v(F) = \sum_{e \in F} v(e)$ soit maximal

L'algorithme glouton s'exprime alors par :

– *Etape 1 :* Classer les éléments de E par ordre des $v(e)$ décroissants. Ils constituent alors une suite $e_1, e_2, \dots e_n$, telle que :

$$v(e_1) \geq v(e_2), \dots \geq v(e_n)$$

– Initialiser $F := \emptyset$

– *Etape 2 :* Pour i variant de 1 à n, ajouter l'élément e_i à F si $F \cup \{e_i\} \in \mathcal{F}$.

On a vu que l'algorithme glouton donne le résultat optimal quelles que soient les valeurs des $v(e)$ pour l'arbre recouvrant, par contre pour celui de l'affectation de ressources on n'obtient l'optimum que lorsque la fonction v est constante ($\forall e, v(e) = 1$). Il est ainsi naturel de rechercher un critère sur la famille $\mathcal{F}$ qui permette de déterminer les cas où l'algorithme glouton donne la solution optimale quelles que soient les valeurs des $v(e)$. On obtient alors le résultat suivant :

THÉORÈME 1.1. *Si l'algorithme glouton donne une solution optimale quelle que soit la fonction v alors la famille $\mathcal{F}$ vérifie la propriété suivante :*

Si F et G sont deux ensembles de $\mathcal{F}$ tels que G a un élément de plus que F, alors il existe un élément x de G qui n'est pas dans F et tel que $F \cup \{x\} \in \mathcal{F}$.

Preuve : Pour démontrer ce résultat on suppose donnés F et G et on construit une fonction de poids v telle que l'algorithme glouton donne une solution qui contient strictement F.

Notons p le nombre d'éléments communs à F et G et q le nombre d'éléments de F qui ne sont pas dans G. On définit la fonction v suivante :

– Pour tout élément f de F, $v(f) = 2q + 2$
– Pour tout élément g de G qui n'est pas dans F, $v(g) = 2q + 1$
– Pour tout élément h qui n'est ni dans F ni dans G, $v(h) = 0$

L'algorithme glouton appliqué pour ces valeurs des poids donne un ensemble F' qui contient nécessairement F puisque les éléments de F se présentent en premier. Or on a $v(F) = (p + q)(2q + 2)$ et $v(G) = p(2q + 2) + (q + 1)(2q + 1)$ Ainsi $v(G) = v(F) + q + 1$, comme l'algorithme glouton donne un ensemble F' ayant un poids maximal le poids de F' doit être supérieur ou égal à celui de G donc strictement supérieur à celui de F. Ainsi F' contient un élément qui n'est pas dans dans F et qui a un poids non nul, c'est donc un élément de G qui est le x cherché. □

En fait cette condition est aussi suffisante comme le montre :

Réciproque Si $\mathcal{F}$ est une famille de sous-ensembles d'un ensemble E fini, satisfaisant les 2 conditions suivantes :

– $F \in \mathcal{F}, F' \subset F, \ \Rightarrow F' \in \mathcal{F}$
– Si F et G sont deux ensembles de $\mathcal{F}$ tels que G contient plus dé'léments que F alors il existe un élément x de $G \setminus F$ tel que $F \cup \{x\} \in \mathcal{F}$.

Alors l'algorithme glouton appliqué à la famille $\mathcal{F}$ donne l'optimum.

Preuve : Supposons que l'algorithme glouton construise l'ensemble F et notons G un des ensembles de la famille $\mathcal{F}$.

On montre, par récurrence sur le nombre p d'éléments de G qui ne sont pas dans F, que

$$v(F) \geq v(G)$$

Si $p = 0$ c'est immédiat. Soit $p > 0$, il existe un $x \in F$ tel que $x \notin G$, sinon $F \subset G$ et l'algorithme glouton aurait sélectionné les éléments de G lorsqu'ils se seraient présentés.

Ainsi soit x le premier élément sélectionné par l'algorithme glouton qui n'est pas dans G. Les éléments y de G qui ne sont pas dans F sont tels que $v(y) \leq v(x)$, sinon ils auraient été retenus par l'algorithme glouton. Soit l'ensemble $F' = (F \cap G) \cup \{x\}$, comme c'est un sous ensemble de F il appartient à $\mathcal{F}$. La deuxième condition ci-dessus implique que l'on peut y ajouter successivement des éléments de G pour obtenir un ensemble G' de même cardinal que G et appartenant à $\mathcal{F}$. Comme x est le seul élément de G' qui n'est pas dans G, on a alors $G' = G \cup \{x\} \setminus \{y\}$ et :

$$v(G') = v(G) + v(x) - v(y) \geq v(G)$$

De plus G' a un élément de plus en commun avec F que n'en a G, l'hypothèse de récurrence implique donc $v(F) \geq v(G') \geq v(G)$. $\square$

4. Matroïdes : définition et exemples

Les propriétés figurant dans le théorème ci-dessus et sa réciproque suggèrent l'introduction de la notion de matroïde qui est une structure définie par une famille de sous-ensembles d'un ensemble E, famille qui vérifie certaines conditions. Cette démarche est assez courante en mathématiques, citons part exemple la topologie définie par des ouverts, la théorie de la mesure et ses ensembles mesurables, les espaces vectoriels et leurs ensembles indépendants de vecteurs. Dans le cadre de la combinatoire, principalement concernée par des structures mathématiques *finies*, on a développé ces dernières années des outils unificateurs qui tendent à présenter sous une forme générale des résultats qui paraissent dans un premier abord très épars. Une de ces structure est celle de matroïde, il n'est pas étonnant de voir apparaître le nombre d'éléments d'un sous-ensemble dans la définition puisque la combinatoire s'intéresse aux ensembles finis.

Définition Un matroïde est un couple $\mathcal{M} = (E, \mathcal{F})$ où E est un ensemble fini et où $\mathcal{F}$ est une famille de parties (sous-ensembles) de E satisfaisant les conditions (on note dans la suite $|F|$ le nombre d'éléments de l'ensemble F) :

(I1) $\mathcal{F}$ contient au moins une partie de E
(I2) $F' \subset F,\ F \in \mathcal{F} \ \Rightarrow F' \in \mathcal{F}$
(I3) $F, G \in \mathcal{F}, |G| > |F| \ \Rightarrow \exists x \in G \setminus F, \quad F \cup \{x\} \in \mathcal{F}$

Les éléments d'une telle famille $\mathcal{F}$, sont appelés ensembles *indépendants* du matroïde $\mathcal{M}$.

Les matroïdes ont fait l'objet d'études mathématiques importantes sans référence à la raison pour laquelle ils ont été introduits dans ce cours. Pour ce qui concerne l'algorithmique il est intéressant de reconnaître si un problème de recherche d'un optimum rentre dans le cadre de la recherche d'un ensemble indépendant maximal d'un matroïde, auquel cas il existe un algorithme simple permettant de trouver cet optimum.

Nous nous proposons de donner quelques exemples de matroïdes :

4.1. Familles de vecteurs indépendants dans un espace vectoriel. Soit E un espace vectoriel sur un corps K, et soit $\mathcal{F}$ la famille des ensembles de vecteurs linéairement indépendants de E. Il est facile de vérifier que cette famille satisfait les conditions (I1), (I2) et (I3), pour les deux premières c'est immédiat. Vérifions la dernière ; soit $F = \{u_1, u_2, \ldots, u_k\}$ et $G = \{v_1, v_2, \ldots, v_k, v_{k+1}\}$ deux ensembles chacun formé de vecteurs indépendants. Si chaque v_i dépendait linéairement des u_j l'ensemble G serait de dimension k ce qui contredirait le fait qu'il est formé de $k+1$ vecteurs indépendants, il existe donc un v_i qui ne dépend pas des u_j et $F \cup \{v_i\} \in \mathcal{F}$.

4.2. Familles des parties sans cycle dans un graphe. Soit $\Gamma = (X, E)$ un graphe symétrique, on définit un matroïde formé par des sous-ensembles de l'ensemble E des arêtes. Un ensemble indépendant est sous-ensemble de E qui ne contient pas de cycle. De manière équivalente on peut définir un ensemble indépendant comme une union d'arbres $F = \cup F_i$ $(i = 1, \ldots, p)$ dont les ensembles de sommets X_i sont disjoints et dont l'union est égale à X. Les conditions I1 et I2 sont faciles à vérifier par contre I3 est un peu plus complexe. Si G et F sont deux parties indépendantes de E telles que G possède une arête de plus que $F = \cup F_i$ on divise G en $G = \cup G_i \cup G'$ où G_i est constitué des arêtes de G ayant leurs deux extrémités dans X_i et G' contient les arêtes qui ont des extrémités dans deux X_i distincts. Pour démontrer I3, il suffit de vérifier que G' n'est pas vide. Or on remarque que G_i ne peut pas contenir plus d'éléments que F_i (sinon il y aurait un cycle dans X_i, G_i). Comme $|G| = |F| + 1$ et $|G| = \sum |G_i| + |G'|$ on a $|G'| > 0$ et le résultat.

4.3. Graphes bipartis. Un graphe $G = (X, E)$ est dit *biparti* si l'ensemble X des sommets est réunion de deux sous-ensembles disjoints Y et Z tels que toute arête $e \in E$ a une extrémité dans Y et l'autre dans Z. On définit un matroïde en prenant comme ensembles indépendants les sous-ensembles F de E tels que deux arêtes de F ne sont pas incidentes à un même sommet appartenant à Y. On peut par symétrie définir un autre matroïde en remplaçant Y par Z.

Un *couplage* dans un graphe biparti est un sous-ensemble M de l'ensemble des arêtes tels que tout couple d'arêtes de M n'a pas de sommet commun. On vérifie que l'ensemble des couplages ne forme pas un matroïde, par contre un *couplage* est un élément qui appartient à la fois aux deux matroïdes définis ci-dessus, ainsi l'ensemble des couplages est l'intersection de deux matroïdes.

4.4. Transversal d'une famille d'ensembles. Soit $\mathcal{A} = A_1, A_2, \ldots, A_k$ une famille de sous-ensembles de E, les *transversaux* de $\mathcal{A}$ sont les parties de $T = \{x_1, x_2 \ldots, x_l\}$ de E telles qu'il existe des entiers deux à deux distincts $i_1, i_2, \ldots i_l$ satisfaisant $x_j \in A_{i_j}$.

Un exemple simple est le suivant : $E = \{1, 2, 3, 4, 5, 6\}$,

$$A_1 = \{1,3,4\},\ A_2 = \{3,5,6\},\ A_3 = \{1,4\},\ A_4 = \{2,5\}, A_5 = \{3,4\}, A_6 = \{1,3\}$$

On vérifie alors facilement que tout sous-ensemble de E qui ne contient pas à la fois $2, 5$ et 6 est un transversal.

Un exemple de transversal est donné par la constitution d'un ensemble de dirigeants pour différentes associations et groupes dans une population. Ainsi si E est l'ensemble des habitants majeurs d'un village, soit A_1 l'ensemble des parents d'élèves, A_2 l'ensemble des pécheurs à la ligne, A_3 les membres du cinéclub, A_4 les joueurs de pétanque etc. On peut alors interpr'eter un transversal est comme un ensemble d'individus tel que chacun d'entre eux dirige une association ou un groupe (il n'y a pas deux personnes qui dirigent le même groupe). La détermination d'un responsable pour chacune des associations de façon telle qu'une même personne ne soit pas à la tête de deux d'entre elles relève de la recherche d'un transversal de taille maximale.

On peut vérifier les axiomes de matroïdes pour ces ensembles transversaux mais ce n'est pas immédiat, ceci pourra être fait au chapitre 3 après l'étude des couplages.

4.5. Réalisation de taches sur un processeur séquentiel. Voici un exemple de matroïde issu d'une situation pratique. Des tâches $T_1, T_2, \dots, T_k$, sont à réaliser sur un processeur séquentiel, la réalisation de chacune de celles-ci prend une unité de temps et on donne pour chaque T_i un délai d_i à respecter (en supposant que l'on commence à l'instant 0, la tâche T_i doit être réalisée avant l'instant d_i). De plus une pénalité p_i est fixée pour le cas où la tache T_i ne serait pas réalisée dans les délais impartis (cette pénalité est fixe et ne dépend pas de l'instant de la réalisation). Il s'agit de déterminer l'ordre sur les tâches qui minimise la somme des pénalités. L'exemple suivant illustre le problème : il y a 7 tâches $T_1, \dots, T_7$ dont les délais et pénalités sont données par le tableau :

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
d	4	3	2	2	6	3	5
p	70	80	90	50	30	40	20

Si on effectue les tâches dans l'ordre donné par T_1, T_2, T_3, T_4, T_5, T_6, T_7 les tâches T_3, T_4, T_6, T_7 ne sont pas effectuées dans les délais et la pénalité est de $90+50+40+20 = 200$. Pour l'ordre : T_3, T_4, T_2, T_5, T_7, T_1, T_6 seules les tâches T_1 et T_6 sont hors délais et la pénalité est 110, noter que l'optimum est atteint pour T_4, T_3, T_2, T_1, T_7, T_5, T_6 avec une pénalité de 40.

On considère la famille $\mathcal{F}$ de sous-ensembles F de $T = \{T_1, T_2, \dots, T_k\}$ donnée par $F \in \mathcal{F}$ si les tâches contenues dans F peuvent être toutes réalisées dans les délais. On montre que cette famille satisfait les conditions sur les matroïdes. (Laissé en exercice, considérer les nombres $U(i, F)$ de tâches de délai inférieur ou égal à i). Ainsi l'algorithme glouton consistant à ajouter les tâches par ordre de pénalités décroissantes donne la solution optimale au problème posé.

5. Définitions équivalentes des matroïdes

Nous avons défini les matroïdes par des ensembles indépendants mais il y a d'autres ensembles remarquables dans un matroïde : les bases et les circuits. Ceux-ci peuvent être définis à partir des ensembles indépendants mais on peut aussi inversement retrouver les ensembles indépendants à l'aide des bases ou à l'aide des circuits. Une notion intéressante est aussi celle de rang d'un sous-ensemble qui associe un entier à tout sous-ensemble de E. L'équivalence entre ces différents concepts peut alors être utilisée pour vérifier qu'une famille constitue bien un matroïde, il est en effet parfois plus simple de retrouver une famille d'ensembles indépendants, d'autres fois une famille de bases, ou de circuits. Les caractérisations sont données par 3 théorèmes dont les preuves techniques sont omises.

Soit $\mathcal{M} = (E, \mathcal{F})$ un matroïde, on appelle base de $\mathcal{M}$ une partie maximale de $\mathcal{F}$. Ainsi la famille $\mathcal{B}$ des bases de $\mathcal{M}$ est composée des ensembles B tels que

$$B \in \mathcal{F}, B \subset F, F \in \mathcal{F} \quad \Rightarrow B = F$$

On vérifie le théorème suivant :

THÉORÈME 1.2. *La famille $\mathcal{B}$ des bases d'un matroïde $\mathcal{M}$ satisfait la condition :*

$$(Ba) \quad B_1, B_2 \in \mathcal{B}, x \in B_1 \setminus B_2, \Rightarrow \exists y \in B_2 \setminus B_1 \quad \text{tel que} \quad (B_1 \cup y) \setminus x \in \mathcal{B}$$

Réciproquement Si $\mathcal{B}$ satisfait la condition (Ba), alors la famille $\mathcal{F}$ des parties F de E telles qu'il existe $B \in \mathcal{B}$ satisfaisant $F \subset B$ vérifie les conditions $(I1), (I2), (I3)$.

La notion de *circuit* joue aussi un rôle important dans cette théorie. Soit $\mathcal{F}$ une famille de parties indépendantes de $\mathcal{M}$, on appelle circuit de $\mathcal{M}$ une partie minimale qui n'appartient pas à $\mathcal{F}$, ainsi la famille $\mathcal{C}$ des circuits de $\mathcal{M}$ est composée des ensembles C tels que

$$C \notin \mathcal{F}, \quad \text{et} \qquad F \subset C, F \neq C \ \Rightarrow F \in \mathcal{F}$$

On vérifie alors la caractérisation suivante :

THÉORÈME 1.3. *La famille $\mathcal{C}$ des circuits d'un matroïde $\mathcal{M}$ vérifie les conditions :*
- *Si C_1 et C_2 sont deux ensembles distincts de $\mathcal{C}$ alors C_1 n'est pas inclus dans C_2.*
- *Si C_1 et C_2 sont deux ensembles distincts de $\mathcal{C}$ et si $x \in C_1 \cap C_2$, alors il existe C_3 tel que $C_3 \subset (C_1 \cup C_2) \setminus x$.*

Réciproquement, soit $\mathcal{C}$ une famille de parties de E satisfaisant les deux conditions ci-dessus, on construit une famille $\mathcal{F}$ d'ensembles indépendants. Un ensemble est indépendant si et seulement s'il ne contient pas de circuit. On vérifie sans trop de peine que $\mathcal{F}$ satisfait les conditions I1, I2 et I3.

La notion de *rang* permet, lorsqu'on a défini un matroïde $\mathcal{M}$ sur E, d'associer un entier à toute partie de l'ensemble de base E. Soit $\mathcal{F}$ une famille de parties indépendantes définissant $\mathcal{M}$, la fonction rang, notée ρ de $\mathcal{M}$ est telle que pour tout A, $A \subset E$, $\rho(A)$ est le cardinal maximal d'un ensemble F de $\mathcal{F}$ inclus dans A, soit :

$$\rho(A) \; = \; \max_{F \subset A, F \in \mathcal{F}} \; |F|$$

On a alors le théorème suivant :

THÉORÈME 1.4. *Une application ρ de l'ensemble des parties de E dans l'ensemble $\mathcal{N}$ des entiers naturels est la fonction rang d'un matroïde $\mathcal{M}$ si et seulement si :*
- $\rho(\emptyset) = 0$,
- $A \subset B \quad \Rightarrow \quad \rho(A) \leq \rho(B)$,
- $\rho(A \cup \{x\}) = \rho(A \cup \{y\}) = \rho(A) \quad \Rightarrow \quad \rho(A \cup \{x, y\}) = \rho(A)$.

Dans les exemples donnés plus haut on retrouve ces différentes notions :

Pour les ensembles indépendants de vecteurs, les bases correspondent à la notion habituelle de base d'un sous-espace vectoriel, la fonction rang correspond à la dimension, enfin les circuits sont les ensembles de vecteurs dépendants tels que la suppression de l'un quelconque d'entre eux donne une famille indépendante.

Pour les graphes connexes, on vérifie facilement que les cycles élémentaires de G satisfont les conditions sur les circuits d'un matroïde. Les les bases sont alors les arbres recouvrants, et le rang d'un sous-ensemble A de l'ensemble E des arêtes est égal à $n - c$. Ici n est le nombre total de sommets du graphe, c est le nombre de composantes connexes du graphe ayant X comme ensemble de sommets et A comme ensemble d'arêtes un sommet sans arête dans A compte pour une composante connexe).

6. Jeu de Shannon

6.1. Présentation. Le jeu de Shannon illustre l'importance de la notion d'arbre recouvrant sur un graphe, et donc celle de base sur un matroïde. On voit aussi apparaître les opérations de suppression et de contraction d'arête qui jouent un rôle crucial en théorie algébrique des graphes et qui ont aussi une généralisation intéressante dans le cadre des matroïdes. Les règles de ce jeu sont les suivantes :

Soit G un graphe symétrique donné par un ensemble X de sommets, où deux sommets u et v sont distingués, et un ensemble E d'arêtes. Un jeu de Shannon sur $G = (X, E, u, v)$ se pratique à deux joueurs. Au départ toute les arêtes sont incolores ; chaque joueur à tour de rôle colorie une arête incolore, l'un *(le conquérant)* en vert, l'autre *(le défenseur)* en rouge. Le conquérant a gagné s'il a réussi à construire un chemin composé d'arêtes vertes reliant u à v ; par contre si toutes les arêtes sont coloriées sans qu'un tel chemin existe c'est le défenseur qui est déclaré gagnant . La victoire du défenseur peut se constater avant la fin du jeu s'il a réussi à disconnecter le graphe par des arêtes rouges, en deux composantes dont l'une contient u et l'autre v.

6.2. Classification. On classe les graphes où deux sommets sont distingués en trois catégories :
- Les graphes *faibles* pour lesquels il existe une stratégie du conquérant qui lui permet de gagner contre toute défense et ceci quel que soit le joueur qui joue en premier.
- Les graphes *résistants* pour lesquels il existe une stratégie permettant au résistant de vaincre et ceci même dans le cas où le conquérant joue en premier.

– Les graphes *neutres* pour lesquels il existe une stratégie gagnante pour tout joueur qui a le trait en premier.

À tout graphe $G = (X, E, u, v)$ on associe le graphe $G_1 = (X, E', u, v)$ où $E' = E \cup \{u, v\}$; on vérifie alors que G est résistant si et seulement si G_1 est neutre et que G est neutre si et seulement si G_1 est faible et G ne l'est pas. Ainsi un algorithme qui permet de vérifier si un graphe est faible peut être utilisé sur G_1 pour tester si G est résistant ou neutre.

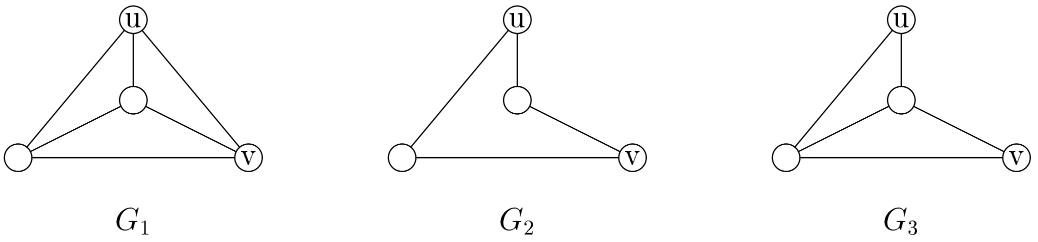


FIG. 2. Trois graphes, G_1 faible, G_2 résistant, G_3 neutre

6.3. Caractérisation des situations gagnantes. On a le résultat suivant dû à A. Lehman :

THÉORÈME 1.5. *Le graphe G avec les sommets u et v est faible si et seulement s'il existe deux arbres $A_1 = (Y, T_1)$, $A_2 = (Y, T_2)$ tels que :*

$$Y \subset X, \quad T_1 \subset E, \quad T_2 \subset E, \quad T_1 \cap T_2 = \emptyset, \quad u, v \in Y$$

On ne démontre pas ce théorème complètement ; on se contente de donner une stratégie pour le conquérant, dans un graphe qui satisfait les conditions ci-dessus, lorsqu'il joue en second. L'idée générale est que les arêtes coloriées en rouge par le défenseur sont supprimées du graphe alors que celles coloriées en vert sont contractées. On donne le sens suivant à la contraction :

Soit $G = (X, E)$ un graphe symétrique et $e = \{x, y\}$ une arête de G, le contracté G_e de G suivant e est le graphe obtenu en supprimant le sommet y et remplaçant chaque arête $\{y, z\}$ dont l'une des extrémités est y par une arête $\{x, z\}$, une telle opération peut amener à construire des arêtes multiples.

La stratégie de l'attaquant consiste lorsque le défenseur supprime une arête qui disconnecte un des arbres T_1 ou T_2 à choisir une arête e dans l'autre arbre de telle façon que que G_e contienne deux arbres disjoints pour les arêtes ayant même ensemble de sommets, ce qui se traduit par l'algorithme suivant :

Si l'arête rendue rouge par le défenseur n'appartient à aucun des deux arbres T_1 ou T_2, colorier en vert n'importe quelle arête.

Si l'arête rendue rouge appartient à T_1, on considère les deux parties connexes Y_1 et Y_2 de T_1 ainsi créées, on choisit une arête f de T_2 qui relie un sommet de Y_1 à un sommet de Y_2, et l'on colorie$f$ en vert.

Si l'arête rendue rouge appartient à T_2, on effectue un choix similaire en échangeant T_1 et T_2.

On poursuit alors le jeu dans le graphe obtenu en supprimant l'arête rouge et en contractant l'arête verte, ce graphe possède encore deux arbres qui satisfont les mêmes conditions que T_1 et T_2, le jeu se termine lorsque les deux sommets u et v sont reliés par une arête, lorsque c'est le tour du conquérant de jouer.

Ce jeu sur des graphes se généralise sur des matroïdes et on montre un résultat du même type car la preuve qui est faite sur les graphes utilise des propriétés des arbres recouvrants qui sont celles d'un matroïde.

CHAPITRE 2

Chemins minimaux, programmation dynamique

Dans ce chapitre on s'intéresse à la recherche de chemins de longueur minimale dans des graphes orientés ou symétriques en supposant que chaque arc (ou chaque arête) est munie d'une longueur. On suppose connus les algorithmes sur les parcours de graphes qui construisent des chemins entre les sommets sans la contrainte de minimalité, ceux-ci figurent dans le cours d'algorithme et programmation du tronc commun (Chapitre 5).

On verra que la recherche de chemins minimaux partant d'un sommet distingué donné fait appel à des techniques du style de l'algorithme glouton. Par contre si l'on recherche l'ensemble des chemins minimaux entre tous les couples de sommets une nouvelle technique est mise en œuvre. Celle-ci se rencontre dans des problèmes de nature différente on l'appelle pour des raisons historiques *programmation dynamique*, le terme programmation est employé ici dans le sens de méthode et non pas dans celui de réalisation d'un programme informatique.

La recherche de plus courts chemins peut être vue aussi comme la résolution d'un système d'équations linéaires dans un anneau particulier pour lequel cette résolution se limite au calcul de la puissance d'une matrice. Une introduction à cette théorie qui est largement développée par ailleurs est donnée.

La méthode de la programmation dynamique est expliquée dans ses grandes lignes au paragraphe suivant, elle est appliqué ensuite pour résoudre plusieurs problèmes d'optimisation dont l'un intervient dans le cadre de l'analyse du génome.

1. Plus courts chemins d'origine donnée

Dans la suite, on considère un graphe orienté $G = (X, A)$ ayant X comme ensemble de sommets et A comme ensemble d'arcs. Les applications *or* et *ext* associent à tout arc a de A son origine et son extrémité respectivement. On se donne une application ℓ de A dans l'ensemble des entiers naturels, $\ell(a)$ est la *longueur* de l'arc a. La longueur d'un chemin est égale à la somme des longueurs des arcs qui le composent. Étant donné un sommet x_0, le problème consiste à déterminer pour chaque sommet $x_i \in X$, le plus court chemin, s'il existe, qui joint x_0 à x_i. Par exemple dans le graphe donné ci-dessous les plus courts chemins à partir du sommet x_2 sont dessinés en traits gras.

Le caractère très simple des algorithmes de recherche de plus courts chemins partant de x_0 repose sur la propriété suivante : si un chemin f de longueur minimale joint x_0 à x tout facteur gauche de ce chemin se terminant en l'arc a est un chemin minimal qui joint x_0 à $ext(a)$.

On utilise dans les deux algorithmes donnés ci-dessous une application d de l'ensemble des sommets dans l'ensemble des entiers. L'image par d du sommet x est notée $d[x]$ car dans la réalisation d'un programme celle-ci sera représentée par un tableau. D'autre part la représentation des plus courts chemins s'effectue par un vecteur `pere` tel que `pere[x]` est le sommet situé immédiatement avant `x` dans un chemin de longueur minimale qui joint x_0 à x. Une opération élémentaire utilisée pour modifier d est l'opération dite de *relaxation*, elle est décrite par l'algorithme suivant :

```
 Relaxation (Arc a) {
      if (d[ext[a]] > d[or[a]] + l(a)){
          d[ext[a]] = d[or[a]] + l(a);
          pere[ext[a]] = or[a]; }
}
```

1.1. Algorithme de Dijkstra. Cet algorithme commence par initialiser l'application d en posant $d[x_0] = 0$ et pour $x \neq x_0$, $d[x] = \infty$. Ensuite, pour chaque sommet tel que $d[x]$ est minimum, il effectue la relaxation de tous les arcs issus de x. On construit une file de priorité `F` qui contient au départ tous les sommets et sur laquelle on effectue deux opérations :

– recherche et suppression de l'élément x tel que $d[x]$ soit minimum ;
– modification de la valeur de certains $d[x]$.

Il s'exprime de façon plus détaillée comme suit :

```
Pour tout sommet x faire {
    pere[x] = x; d[x] = Maxint;
    Ajouter (x, F);
    }
d[x0] = 0;
Tant que (F non vide) {
    Rechercher x de F tel que d[x] minimum;
    Supprimer x de F;
    Pour tout (a tel que (or[x] == a)) faire Relaxation (a);
    }
```

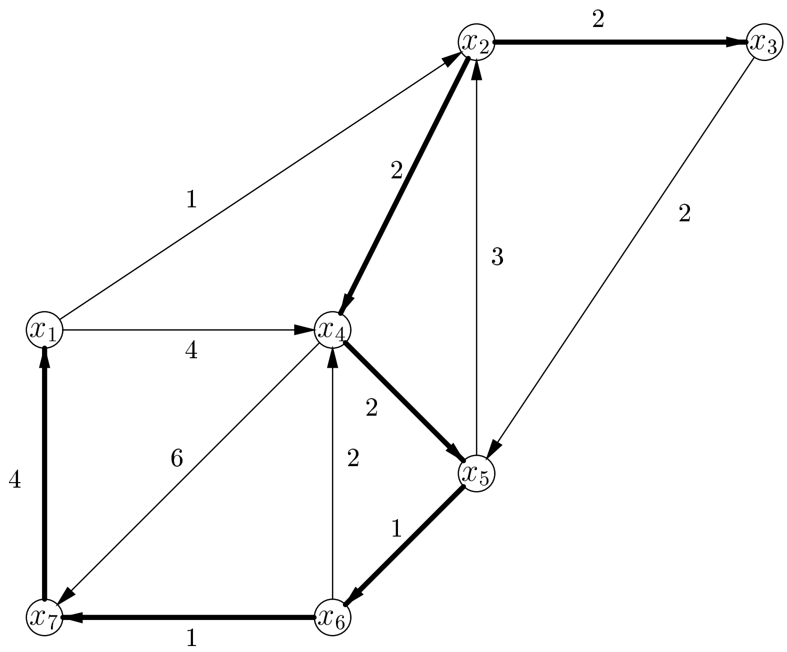


FIG. 1. Plus courts chemins issus du sommet x_2

On peut considérer l'algorithme de Dijkstra comme un algorithme glouton (on traite à chaque étape le sommet le plus intéressant) mais qui remet à jour la valeur de chaque élément après la réalisation d'une étape.

Remarques :

1. Une réalisation efficace de `F` consiste à utiliser la notion de *tas* introduite dans le cours d'algorithmes. Pour un graphe ayant n sommets et m arcs, on effectue n suppressions et m modifications de valeurs de d, chaque suppression demande $\log n$ opérations de même que chaque modification de valeur, la complexité de l'algorithme est en $O(n \log n + m \log n)$.

2. On peut terminer l'algorithme dès que tous les éléments x situés dans `F` sont tels que $d[x] = \infty$; aucun de ces éléments ne sera atteint par un chemin d'origine x.

3. L'algorithme a été décrit pour des graphes orientés ; si le graphe est symétrique il faut considérer une arête comme étant formée de deux arcs opposés.

4. L'algorithme ne donne pas le bon résultat si certains arcs sont négatifs. Il est clair qu'on ne peut s'attendre à obtenir un résultat s'il existe des circuits dont la somme des longueurs des arcs est négative ; par contre en l'absence de tels circuits certains algorithmes donnent les plus courts chemins. C'est le cas de l'algorithme de Ford ci-dessous.

1.2. Algorithme de Ford. Cet algorithme est particulièrement simple : il consiste à effectuer pour un graphe à n sommets, $n - 1$ fois la procédure de relaxation sur tous les arcs. Soit plus précisément :

```
        Pour tout sommet x {
            Pere[x] = x; d[x] = Maxint;
            }
        d[x0] = 0;
        Pour i=1 jusqu'a n-1 {
             Pour tout arc a du graphe G faire Relaxation(a)
};
```

On peut démontrer que cet algorithme donne le chemin minimal d'origine x_0. Plus exactement, on vérifie qu'après l'exécution de la i-ème itération de la boucle on a obtenu tous les chemins minimaux composés de i arcs. Ce résultat est valide même en présence d'arcs de longueur négative, la seule condition est que tout circuit soit de longueur positive. Ce critère se vérifie une fois l'algorithme de Ford exécuté en testant la présence d'un arc `a` tel que `d[ext[a]] > d[or[a]] + l[a]`. On trouve un tel arc si et seulement si le graphe possède un circuit de longueur négative.

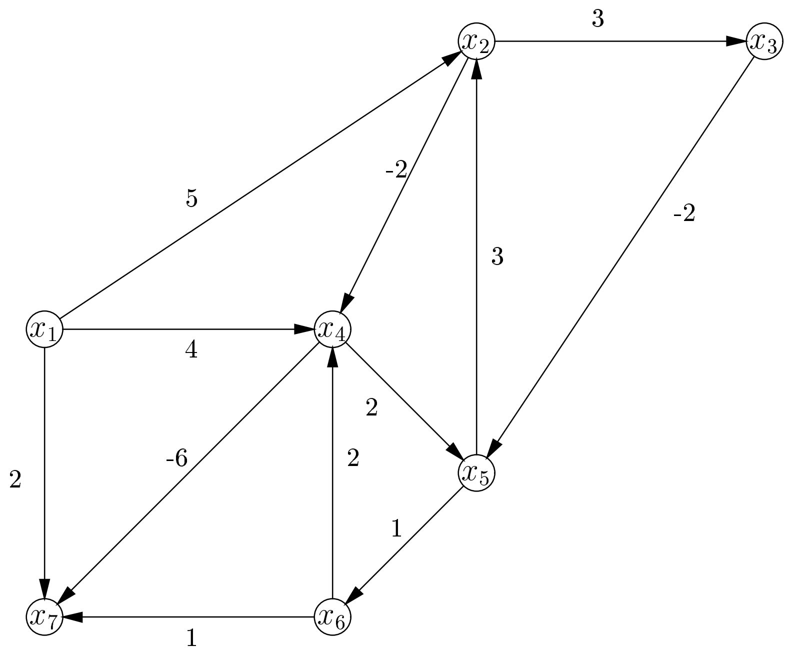


FIG. 2. Un graphe avec des arcs négatifs

Sur l'exemple ci-dessus on obtient les plus courts chemins depuis x_1 à tous les sommets ils sont de longueurs respectives 5, 8,3,5,6,-3 pour $x_2, x_3, x_4, x_5, x_6, x_7$ on vérifie que toute relaxation sur ces valeurs ne modifie pas le résultat trouvé.

La complexité de l'algorithme est en $O(nm)$, ce qui est moins bon que celui de Dijkstra.

2. Plus courts chemins entre tous les couples de sommets

On cherche ici à déterminer tous les plus courts chemins et non pas uniquement ceux d'origine x_0. Nous commençons par donner un algorithme qui détermine les longueurs de ces plus courts chemins. La longueur du plus court chemin entre x_i et x_j est notée $\delta(x_i, x_j)$. On convient de poser $\delta(x_i, x_j) = \infty$ s'il n'existe pas de chemin entre x_i et x_j (en pratique il suffit dans la suite de remplacer ∞ par un nombre suffisamment grand par exemple la somme des longueurs de tous les arcs du graphe). La construction effective des chemins sera examinée ensuite, elle se déduit simplement du calcul de δ. On suppose qu'entre deux sommets il y a au plus un arc. En effet, s'il en existe plusieurs, il suffit de ne retenir que le plus court. On note $\ell(i, j)$ la longueur de l'arc a le plus court qui a pour origine x_i et pour extrémité x_j, $\ell(i, j) = \infty$ s'il n'existe pas de tel arc.

Les algorithmes de recherche de chemins les plus courts reposent sur l'observation très simple mais importante suivante :

Remarque *Si f est un chemin de longueur minimale joignant x à y et qui passe par z, alors il se décompose en deux chemins de longueur minimale l'un qui joint x à z et l'autre qui joint z à y.*

Dans la suite, on suppose les sommets numérotés $x_1, x_2, \ldots x_n$ et, pour tout $k > 0$ on considère la propriété P_k suivante pour un chemin f :

> $P_k(f)$ Tous les sommets de f, autres que son origine et son extrémité, ont un numéro strictement inférieur à k.

On peut remarquer qu'un chemin vérifie P_1 si et seulement s'il se compose d'un unique arc, d'autre part la condition P_{n+1} est satisfaite par tous les chemins du graphe. Notons $\delta_k(x_i, x_j)$ la longueur du plus court chemin qui vérifie P_k et qui a pour origine x_i et pour extrémité x_j. Cette valeur est ∞ si aucun tel chemin n'existe. Ainsi $\delta_1(x_i, x_j) = \infty$ s'il n'y a pas d'arc entre x_i et x_j et vaut $\ell(a)$ si a est cet arc. D'autre part $\delta_{n+1} = \delta$. Le lemme suivant permet de calculer les δ_{k+1} connaissant les $\delta_k(x_i, x_j)$. On en déduira un algorithme itératif.

LEMME 2.1. *Les relations suivantes sont satisfaites par les δ_k :*

$$\delta_{k+1}(x_i, x_j) = \min(\delta_k(x_i, x_j), \delta_k(x_i, x_k) + \delta_k(x_k, x_j))$$

DÉMONSTRATION. Soit un chemin de longueur minimale satisfaisant P_{k+1}, ou bien il ne passe pas par x_k et on a

$$\delta_{k+1}(x_i, x_j) = \delta_k(x_i, x_j)$$

ou bien il passe par x_k et, d'après la remarque préliminaire, il est composé d'un chemin de longueur minimale joignant x_i à x_k et satisfaisant P_k et d'un autre minimal aussi joignant x_k à x_j. Il a donc pour longueur : $\delta_k(x_i, x_k) + \delta_k(x_k, x_j)$. □

L'algorithme pour la recherche du plus court chemin met à jour une matrice `delta[i,j]` qui a été initialisée à $\ell(i, j)$. À chaque itération de la boucle externe, on fait croître l'indice k du δ_k calculé.

```
Pour k = 1 jusqu'a n {
    Pour  i = 1 jusqu'a n {
        Pour  j = 1 jusqu'a n {
          Si delta[i][j] > (delta[i][k] + delta[k][j])

              delta[i][j]:=  delta[i][k] + delta[k][j]; }}}
```

Sur l'exemple du graphe donné sur la figure 4, on part de la matrice δ_1 donnée par

$$\delta_1 = \begin{pmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ \infty & 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 0 & 2 & \infty & 6 \\ \infty & 3 & \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 2 & \infty & 0 & 1 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Après le calcul on obtient :

$$\delta = \begin{pmatrix} 0 & 1 & 4 & 3 & 5 & 6 & 7 \\ 10 & 0 & 3 & 2 & 4 & 5 & 6 \\ 8 & 5 & 0 & 5 & 2 & 3 & 4 \\ 8 & 5 & 8 & 0 & 2 & 3 & 4 \\ 6 & 3 & 6 & 3 & 0 & 1 & 2 \\ 5 & 6 & 9 & 2 & 4 & 0 & 1 \\ 4 & 5 & 8 & 7 & 9 & 10 & 0 \end{pmatrix}$$

Pour le calcul effectif des chemins les plus courts, on utilise une matrice `suivant` telle que `suivant[i][j]` est le sommet qui suit i dans le chemin le plus court qui va de i à j. Les valeurs `suivant[i][j]` sont initialisées à j s'il existe un arc de i vers j et à -1 sinon, `suivant[i][i]` est lui initialisé à i. Le calcul précédent qui a donné δ peut s'accompagner de celui de `suivant` en ajoutant la ligne :

```
suivant[i][j] = suivant[i][k]
```

dans le cas où le test sur `delta[i][j]` est positif.

Une fois le calcul des deux matrices effectué on peut retrouver le chemin le plus court qui joint i à j par la procédure :

```
k = i;
while (k != j) k= suivant[k][j];
```

Sur l'exemple précédent on trouve pour `suivant` :

$$\begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 2 & 3 & 4 & 4 & 4 & 4 \\ 5 & 5 & 3 & 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 4 & 5 & 5 & 5 \\ 6 & 2 & 2 & 6 & 5 & 6 & 6 \\ 7 & 7 & 7 & 4 & 4 & 6 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 7 \end{pmatrix}$$

3. Généralisation de l'algorithme

L'algorithme précédent a une structure qui rappelle celui du calcul de la puissance p-ième d'une matrice. Cela se voit mieux sur l'algorithme ci-dessous qui donne la longueur $d^{(p)}(i,j)$ du plus court chemin de longueur inférieure ou égale à p entre les sommets x_i et x_j d'un graphe $G = (X, A, \ell)$. Dans ce qui suit $\ell(i,j)$ désigne la longueur de l'arc le plus court entre x_i et x_j, égal à ∞ s'il n'existe pas de tel arc.

$$\begin{aligned} d^{(1)}(i,j) &= \ell(i,j), \\ d^{(p)}(i,j) &= \min[d^{(p-1)}(i,j), \min_{1\leq k\leq n}[d^{(p-1)}(i,k) + \ell(k,j)]]. \end{aligned}$$

Ceci est exactement le calcul de la puissance de la matrice contenant les $\ell(i,j)$ si on utilise des opérations particulières pour la somme et le produit des coefficients. Il faut en effet remplacer la somme par min et le produit par $+$. On est ainsi amené à introduire la notion de semi-anneau.

3.1. Semi-anneaux. Définition. Un *semi-anneau* est un ensemble S muni de deux lois de compositions notées ici $\oplus$ et $\odot$ telles que ;

(1) La loi $\oplus$ est associative, commutative et possède un élément neutre noté $\underline{0}$
(2) La loi $\odot$ est associative, et possède un élément neutre noté $\underline{1}$
(3) La loi $\odot$ est distributive (à gauche et à droite) par rapport à $\oplus$ et pour tout s dans S on a $s \odot \underline{0} = \underline{0}$

Tout anneau est bien entendu un semi-anneau, en effet en plus de ces conditions on a aussi dans un anneau l'existence d'un inverse pour $\oplus$. Les exemples de semi-anneaux sont nombreux en voici quelques-uns qui sont utiles en algorithmique.

1. Le semi-anneau de *Boole* $\mathcal{B}$ a deux éléments $\{0, 1\}$, $\oplus$ est l'opération `ou`, $\odot$ est l'opération `et`.

$\oplus$	0	1
0	0	1
1	1	1

$\odot$	0	1
0	0	0
1	0	1

2. L'ensemble des entiers naturels $\mathbb{N}$, pour les lois usuelles d'addition et de multiplication.

3. L'ensemble $\mathbb{N} \cup \{\infty\}$ des entiers naturels augmenté de ∞, pour les lois min et $+$. Les éléments neutres sont respectivement ∞ et 0 pour ces deux lois. On vérifie simplement que min est associative, commutative et possède ∞ comme élément neutre, d'autre part $+$ est associative d'élément neutre 0 ; enfin on a bien :

$$a + \min(b, c) = \min(a + b, a + c), \quad \forall a \in S, \ a + \infty = \infty$$

4. Si on remplace min par max dans l'exemple ci-dessus, on n'a pas tout à fait un semi-anneau, car l'élément neutre de max qui est 0, ne satisfait pas la relation voulue avec $+$, par contre c'est le cas pour max et la multiplication.

5. Le dernier exemple est un peu plus nouveau. On considère un alphabet A et l'ensemble A^* des mots sur A, S est alors l'ensemble des parties $\mathcal{P}(A^*)$ de A^*. L'opération $\oplus$ est la réunion et l'opération $\odot$ est définie à partir de la concaténation . Pour deux parties U et V de A^* :

$$U \odot V \quad = \{u.v | u \in U, v \in V\}$$

Les éléments neutres sont respectivement $\emptyset$ pour l'union et l'ensemble composé du seul mot vide ε pour $\odot$.

3.2. Puissances de matrices. On peut définir des matrices carrées $n \times n$ dont les coefficient sont dans un semi-anneau S ; le produit de deux telles matrices P et Q est alors la matrice M dont le coefficient $M_{i,j}$ est donné par :

$$M_{i,j} = \oplus_{k=1}^{n} (P_{i,k} \odot Q_{k,j})$$

Le calcul des plus courts chemins dans un graphe se généralise à celui de graphes dont les arcs sont valués par des éléments d'un semi-anneau S. Soit $G = (X, A, \lambda)$, où (X, A) est un graphe orienté et λ une application de A dans un semi-anneau S. La valuation λ se généralise des arcs aux chemins en posant, pour tout chemin $f = a_1, a_2, \ldots, a_k$, $\lambda(f)$ égale au « produit » $\odot_{i=1,k} \lambda(a_i)$. Soit M la matrice dont le coefficient $M_{i,j}$ est égal à la « somme » (par $\oplus$) des $\lambda(a)$ pour tous les arcs a d'origine x_i et d'extrémité x_j, $M_{i,j}$ est égal à $\underline{0}$ s'il n'existe pas de tel arc. On a alors le théorème suivant :

THÉORÈME 2.1. *L'élément $M^k_{i,j}$ de la puissance k-ième de la matrice M est égal à la « somme » par $\oplus$ des $\lambda(f)$ pour tous les chemins f d'origine x_i, d'extrémité x_j et de longueur k.*

La preuve de ce résultat s'effectue par récurrence sur k.

Remarques :

1. On a souvent à calculer la valeur donnée par le théorème pour tous les chemins de longueur inférieure ou égale à k et non pas seulement pour ceux de longueur k, il faut alors faire la somme des puissances M^i de M pour $i = 0$ (M^0 est la matrice identité) jusqu'à k.

2. Dans de nombreux cas on constate pour un certain entier k que pour tout $k' > k$:

$$\sum_{i=0,k} M^i = \sum_{i=0,k'} M^i$$

On dit alors que M est quasi-inversible, son quasi-inverse est $M^* = \sum_{i=1,k} M^i$ en effet on a :

$$I \oplus (M \odot M^*) = M^*$$

3. Le théorème précédent donne avec les différents exemples cités plus haut des résultats intéressants d'un point de vue algorithmique. Dans le cas du semi-anneau de Boole $\mathcal{B}$, $M^k_{i,j} = 1$ indique l'existence d'un chemin de longueur k entre x_i et x_j ; dans celui de $S = \mathbb{N}$ avec les opérations usuelles $M^k_{i,j}$ est le nombre de tels chemins. Lorsque l'on prend $S = \mathbb{N} \cup \{\infty\}$ et les opérations $\min, +$ on obtient les plus courts chemins, enfin pour $S = \mathcal{P}(A^*)$ et les opérations $\cup$ et ., le coefficient $M^k_{i,j}$ est égal à l'ensemble de toutes les étiquettes des chemins (l'étiquette d'un chemin est la concaténation des étiquettes des arcs qui le composent) de longueur k qui ont pour origine x_i et pour extrémité x_j. Cette propriété est utilisée en théorie des automates et des langages rationnels.

4. Programmation dynamique

La recherche d'un chemin minimal dans un graphe entre tous les couples de sommets est un cas particulier d'une méthode générale de recherche d'optimum que l'on appelle *programmation dynamique*.

4.1. Présentation de la méthode. Les problèmes que l'on peut traiter à l'aide de cette méthode ont deux caractéristiques principales

- Une solution optimale à un problème de taille k contient dans un sens à préciser des solutions optimales à des problèmes de tailles plus petites que k.
- On peut construire une solution optimale à un problème de taille k en connaissant un petit nombre de solutions optimales à des problèmes de tailles plus petites que k.

Une méthode de résolution de problème par la programmation dynamique consiste à résoudre itérativement des problèmes de tailles de plus en plus grandes en conservant en mémoire les résultats obtenus pour les problèmes de petites tailles. En général une mémoire de l'ordre de n^2 pour un problème de taille n est nécessaire.

4.2. Un problème générique. L'exemple qui suit est un exemple typique de tel problème, on l'exprime souvent sous la forme suivante :

Un système est caractérisé par son *état* qui est un élément σ d'un ensemble fini Σ, on considère des suites $\sigma_0, \sigma_1, \ldots, \sigma_k$ d'états successifs du système, σ_i est l'état au temps $i = 1, \ldots, k$. Le système est commandé par des *décisions* à prendre représentées par des entiers $x_1, x_2, \ldots, x_k$: x_i est la décision prise au temps i. Le comportement du système est régi par k fonctions de transitions ϕ_i, une pour chaque valeur de i. La fonction de transition ϕ_i a pour donnée un état σ et une décision x, elle a pour résultat un état ; $\phi_i(\sigma, x)$ est l'état atteint au temps i si l'état au temps $i-1$ est σ et si la décision prise au temps i est x.

Une fonction de coût c_i pour chaque i compris entre 1 et k permet de mesurer les coûts des décisions prises. La fonction de coût a les mêmes données que la fonction de transition et a pour résultat un entier $c_i(\sigma, x)$ qui est le coût de la décision x au temps i si l'état du système est σ au temps $i-1$.

Le problème consiste à trouver une suite de transitions entre deux états donnés l'état initial et l'état final qui minimise la somme des coûts. Quelquefois l'état final n'est pas précisé, il faut alors obtenir celui qui donne le minimum.

Une autre version du problème consiste à considérer non pas des coûts c_i mais des gains g_i ; il faut alors rendre le gain maximum plutôt que de chercher un coût minimal.

Dans l'exemple de la recherche du chemin minimal la décision à prendre consiste à choisir un des arcs ou à rester sur place, l'état du système est le sommet où l'on se trouve dans le graphe. Les transitions possibles sont celles où l'on a choisi un arc ayant pour origine le sommet où l'on se trouve, le résultat de la transition consiste à passer au sommet extrémité de l'arc choisi, et le coût de la transition est égal à la longueur de l'arc.

4.3. Algorithme. L'algorithme de programmation dynamique s'exprime de la façon suivante : pour chaque couple d'états σ, τ on note $\delta_k(\sigma, \tau)$ le coût minimal d'une suite de transitions conduisant de σ à l'instant 0 à τ à l'instant k ; ce coût est infini si une telle suite n'existe pas. On a alors la formule de récurrence suivante qui permet de résoudre le problème de manière itérative :

$$\delta_k(\sigma, \tau) = \min_{\theta \in \Sigma}[\delta_{k-1}(\sigma, \theta) + c_k(\theta, x_k)]$$

Dans cette formule x_k satisfait $\phi_k(\theta, x_k) = \tau$.

On remarque qu'il suffit alors de conserver les valeurs de δ_{k-1} pour obtenir celles de δ_k, il n'est pas nécessaire de connaître les valeurs de δ_i pour $i < k-1$, la mémoire nécessaire est bien en $O(n^2)$ où n est le nombre d'états du système.

4.4. Le sac à dos. Le problème du sac à dos est connu comme étant NP-complet en général, toutefois dans le cas particulier où les poids des objets à emporter sont des entiers on peut obtenir un algorithme polynomial en fonction de la capacité totale du sac. Précisons le problème : on se donne des entiers $p_1, p_2, \ldots, p_k$ représentant les poids de certains objets et un nombre P représentant le poids maximal des objets que l'on peut emporter dans le sac ; pour chaque objet i ($i = 1, \ldots, k$), a_i est l'indice de satisfaction obtenu si on emporte i dans le sac. Le problème consiste à trouver le sous-ensemble d'objets qui maximise la somme des indices de satisfaction et tel que la somme des poids des objets qui le composent est inférieure ou égale à P.

Avec la terminologie du paragraphe précédent on considère des gains et non pas des coûts. L'état du système est la capacité qui reste dans le sac à dos, ainsi l'état initial est P, lorsque l'on prend l'objet i on passe d'une capacité p à la capacité $p - p_i$. Les différentes décisions sont représentées par une variable qui prend les valeurs 0 et 1, $x_i = 1$ si on met l'objet i dans le sac $x_i = 0$ sinon. Ainsi $\phi_i(p, x) = p - xp_i$ et $g_i(p, x) = a_i$. L'algorithme de programmation dynamique consiste à gérer un tableau `benefice[p]` de taille P qui donne à chaque étape i le bénéfice maximum que l'on peut réaliser en prenant au plus i objets et en laissant une capacité résiduelle de p dans le sac. Les poids de chaque objet i sont donnés dans un tableau `poids[i]` et les nombres a_i dans un tableau `a[i]`.

```
Pour p = 1 jusqu'a  P faire benefice[p] = 0;
Pour i = 1 jusqu'a k faire {
         q = poids[i];
         Pour p = q jusqu'a P faire
            Si (benefice[p-q] < benefice[p] + a[i])
                 benefice[p-q] = benefice[p] + a[i];
        }
```

5. Traitement de séquences

La comparaison de deux séquences afin de déterminer leur similarité joue un rôle central en bio-informatique. On utilise souvent la programmation dynamique pour résoudre ces problèmes qui proviennent de l'analyse du génome humain et de la compréhension de sa structure primaire.

5.1. Sous-séquence communes. Un exemple simple consiste à déterminer la plus longue sous-séquence commune à deux séquences données. Précisons tout d'abord quelques définitions. Une *séquence* (ou un *mot*) est une suite finie de symboles (ou *lettres*) pris dans un ensemble fini (ou *alphabet*). Si $u = a_1 \cdots a_n$ est une séquence, où $a_1, \ldots, a_n$ sont des lettres, l'entier n est la *longueur* de u. Une séquence $v = b_1 \cdots b_m$ est une *sous-séquence* de $u = a_1 \cdots a_n$ s'il existe des entiers $i_1, \ldots, i_m$, $(1 \leq i_1 < \cdots i_m \leq n)$ tels que $a_{i_k} = b_k$ $(1 \leq k \leq m)$. Une séquence w est une *sous-séquence commune* aux séquences u et v si w est sous-séquence de u et de v. Une sous-séquence commune est dite *maximale* si sa longueur est maximale parmi toutes les sous-séquences communes.

On cherche d'abord à déterminer la longueur d'une sous-séquence commune maximale à $u = a_1 \cdots a_n$ et $v = b_1 \cdots b_m$. Pour cela, on note $L(i, j)$ la longueur d'une sous-séquence commune maximale aux mots $a_1 \cdots a_i$ et $b_1 \cdots b_j$, $(0 \leq j \leq m, 0 \leq i \leq n)$. On peut montrer que :

$$L(i,j) = \begin{cases} 1 + L(i-1, j-1) & \text{si } a_i = b_j \\ \max(L(i, j-1), L(i-1, j)) & \text{sinon.} \end{cases}$$

En effet, soit w une sous séquence de longueur maximale, commune à $a_1 \cdots a_{i-1}$ et à $b_1 \cdots b_{j-1}$ si $a_i = b_j$, wa_i est une sous-séquence commune maximale à $a_1 \cdots a_i$ et $b_1 \cdots b_j$. Si $a_i \neq b_j$ alors une sous-séquence commune à $a_1 \cdots a_i$ et $b_1 \cdots b_j$ est ou bien commune à $a_1 \cdots a_i$ et $b_1 \cdots b_{j-1}$ (si elle ne se termine pas par b_j) ; ou bien à $a_1 \cdots a_{i-1}$ et $b_1 \cdots b_j$, (si elle ne se termine par a_i). On obtient ainsi l'algorithme qui permet de déterminer la longueur d'une sous séquence commune maximale à $a_1 \cdots a_n$ et $b_1 \cdots b_m$

Il est assez facile de transformer l'algorithme pour retrouver une sous-séquence maximale commune au lieu de simplement calculer sa longueur. Pour cela, on met à jour un tableau `provient` qui indique lequel des trois cas a permis d'obtenir la longueur maximale.

Une fois ce calcul effectué il suffit de remonter à chaque étape de `i,j` vers `i-1, j-1`, vers `i, j-1` ou vers `i-1,j` en se servant de la valeur de `provient[i][j]`. On peut améliorer cette méthode et rendre la complexité plus faible grâce à des techniques astucieuses de programmation, c'est ce qui est fait pour la commande `diff` du système UNIX.

5.2. Alignement de séquences. Soit u et v deux mots écrits à l'aide de l'alphabet $\mathbb{A}$ (on prend souvent $\mathbb{A} = \{G, T, A, C\}$ en bioinformatique), on appelle alignement de u, v un couple de mots u', v' sur l'alphabet $\mathbb{A} \cup \{e\}$ (où $e \notin \mathbb{A}$ représente le caractère d'espacement), couple qui satisfait les conditions suivantes

– les deux mots u' et v' ont même longueur ($|u'| = |v'|$) ;
– la suppression de e dans u' et dans v' donne u et v respectivement ;
– les lettres e ne se retrouvent pas en même position dans u' et v'.

Un exemple d'alignement de $u = CGATTAG$ et $v = GATCGA$ est

$$\begin{array}{lcl} u' & = & C\,G\,A\,T\,T\,e\,A\,G \\ v' & = & e\,G\,A\,T\,G\,G\,A\,e \end{array}$$

Il y a bien entendu de très nombreux alignements possibles pour deux séquences, en fait on cherche ceux qui sont les plus intéressants dans le sens précis que l'on décrit ici :

On suppose donnée une fonction de similarité p entre les couples de lettres. La valeur de $p(a, b)$ est un nombre reél d'autant plus grand que les lettres sont considérées comme similaires d'un point de vue biologique, ainsi on prendra des valeurs négatives quand les lettres sont différentes, soit : $b \neq a$, $p(a, a) > 0 > p(a, b)$, de plus la symétrie impose $p(a, b) = p(b, a)$. D'autre part on se donne aussi un réel $q < 0$ qui exprime la similarité d'une lettre avec l'espacement, elle est indépendante de la lettre, ainsi pour tout a on a $p(a, e) = p(e, a) = q$.

Le score d'un alignement est alors donné par

$$\sum_{i=1}^{n} p(u'_i, v'_i)$$

où n est la longueur commune à u' et v'.

Par exemple, pour l'exemple précédent lorsque $p(a, b) = -1$ pour $a \neq b$, $p(a, a) = 5$ et $q = -2$ on trouve un score de 13.

La détermination du score le plus élevé d'un alignement s'effectue grâce à un algorithme de programmation dynamique basé sur la remarque suivante :

Le meilleur alignement de ua et vb s'obtient soit à partir de celui de u et v en alignant a et b, soit à partir de celui de ua et v en alignant b avec un espacement, soit enfin à partir de celui de u et vb en alignant a avec un espacement.

On construit ainsi pas à pas un tableau ms (pour meilleur score) à l'aide de la récurrence suivante :

Soit n la longueur de u et m celle de v, on note par $ms_{i,j}$ (pour $0 \leq i \leq n,\ 0 \leq j \leq m$) le score d'un meilleur alignement entre $u_1 u_2 \cdots u_i$ et $v_1 v_2 \cdots v_j$. Par convention on pose

$ms_{0,j} = qj$, $ms_{i,0} = qi$ correspondant à un alignement dans lequel tous les caractères de l'une des séquences sont des espacements.

$$ms_{i,j} = \max \begin{cases} ms_{i-1,j-1} + p(u_i, v_j) \\ ms_{i-1,j} + q \\ ms_{i,j-1} + q \end{cases}$$

On note que le calcul de ms s'effectue en temps $O(nm)$ et semble nécessiter un espace de taille $O(nm)$, en fait on constate que l'on peut se limiter à conserver en mémoire une seule ligne de la matrice, la ligne suivante remplace alors la précédente lors du calcul.

Un dernier problème consiste à déterminer l'alignement optimal lui-même une fois calculé son score, c'est à dire connaisant le tableau ms, ceci est relativement simple si l'on connaît la totalité du tableau : on part de $ms_{m,n}$ et on remonte dans le tableau de $ms_{i,j}$ vers celui des $ms_{i-1,j-1}, ms_{i-1,j}$ ou $ms_{i,j-1}$ qui a déterminé sa valeur.

Dans le cas où l'on ne connaît que la dernière ligne du tableau, il faut utiliser une technique procédant du principe de *diviser pour regner* et qui pourra être vue en exercice.

Notons deux problèmes voisins de celui qui est considéré ici qui peuvent être traités par les mêmes techniques.

Le premier consiste étant données deux séquences u et v à trouver un facteur de chacune de celles-ci formant un couple qui présente un alignement de score maximal (rappelons qu'un facteur d'une séquence est une sous-séquence constituée de caractères consécutifs). Pour résoudre ce problème (on utilise ici le fait que $q < 0$ et $p(a, b) < 0$) si $a \neq b$ on applique l'algorithme ci-dessus en considérant $ms_{i,j}$ comme le meilleur score d'un alignement entre un suffixe de $u_1 u_2 \ldots u_i$ et un suffixe de $v_1 v_2 \ldots v_j$; ce suffixe pouvant être le mot vide, on voit que $m_{i,j}$ ne peut pas être négatif, ainsi 0 doit figurer comme quatrième possibilité dans la formule de récurrence donnant $m_{i,j}$.

Le second problème consiste à déterminer le meilleur alignement pour une fonction de score qui ne tient pas compte des espacements situés en début ou en fin de mot. L'algorithme pour ce problème ne différe du précédent que dans l'initialisation des $m_{i,0}$ et $m_{j,0}$ qui doivent prendre la valeur 0. On retient alors la valeur maximale de la dernière ligne ou de la dernière colonne.

5.3. Distance d'édition. Un problème voisin du précédent consiste à déterminer ce qu'on appelle la *distance d'édition* entre deux mots f et g. Cette notion est introduite en considérant les trois opérations suivantes permettant de transformer une séquence en une autre :

(1) La suppression d'une lettre (notée S)
(2) L'ajout d'une lettre (notée A)
(3) Le remplacement d'une lettre par une autre (notée R)

Chacune de ces opérations a un coût et il s'agit de déterminer la suite d'opérations ayant un coût minimum permettant de passer d'un mot à un autre. Ainsi pour passer de ACGATC à CAGTCAA on peut faire un ajout (en tête), qui donne CACGATC, puis une suppression pour obtenir CAGATC, une nouvelle suppression et deux ajouts en fin de mot.

On peut aussi faire deux remplacements pour obtenir CAGATC, puis une suppression et deux ajouts. La meilleure suite d'opérations dépend bien entendu des valeurs des coûts, mais on voit bien que des techniques très voisines de ce qui a été fait ci-dessus donnent le résultat.

CHAPITRE 3

Flots et couplages

Les graphes permettent de modéliser des relations de dépendance et de donner un cadre mathématique (algèbre, combinatoire) à de nombreux problèmes pratiques. La résolution de ces problèmes se traduit alors par la recherche d'algorithmes efficaces sur les graphes, nous avons ainsi développé les algorithmes de parcours, de construction de chemins ou d'arbres couvrants optimaux. Dans ce chapitre, nous nous préoccupons de propriétés globales d'un autre type qui mettront en jeu *simultanément* l'ensemble des arêtes du graphe ; il s'agit de modéliser la circulation de personnes, voitures, informations ou de fluides à travers un réseau de façon à rendre maximum l'acheminement global tout en respectant des contraintes locales de capacités des liaisons entre deux sites, ces notions se modélisent à l'aide du concept de *flot*.

La recherche d'un flot maximal est déjà intéressante par elle-même, mais il y a aussi plusieurs applications de la notion de flot. Une des applications, parmi les plus répandues, est la recherche d'un *couplage* maximum dans un graphe biparti. Le couplage modélise le problème de l'affectation (machine/ouvrier, processus/unité de calcul, stagiaire/entreprise).

1. Flots

On considère un graphe orienté dont chaque arc est affecté d'une capacité, nombre entier positif ou nul. Étant donnés deux sommets s et t du graphe, appelés respectivement *source* et *puits*, on cherchera à faire passer dans le graphe un flot maximal de s vers t en respectant les capacités de chaque arc et en appliquant les lois de Kirchoff aux sommets qui interdisent en quelque sorte l'accumulation en un sommet.

Plus précisément on considère un graphe avec capacité $G = (X, A, \gamma)$, où X est l'ensemble des sommets, A l'ensemble des arcs et γ la fonction de capacité qui à chaque arc a

associe un entier[1] positif ou nul $\gamma(a)$. On suppose donnés aussi deux sommets s et t entre lesquels on cherche à faire passer un flot maximal. On suppose qu'il y a au plus un arc d'origine et d'extrémité données, ainsi on notera (x, y) l'arc a ayant x comme origine et y comme extrémité. Par souci de simplicité on supposera qu'il n'existe pas d'arc de type (x, x) bouclant sur un sommet. On appelle *flot* de source s et de puits t dans le graphe $G = (X, A)$, muni de la capacité γ, une fonction positive entière ϕ sur A qui vérifie les deux propriétés suivantes :

$$0 \leq \phi(x, y) \leq \gamma(x, y) \quad \text{, pour tout arc } (x, y) \in A, \tag{1}$$

$$\sum_{(x,y)\in A} \phi(x, y) = \sum_{(y,z)\in A} \phi(y, z) \quad \text{, pour tout sommet } y \neq s, t. \tag{2}$$

On définit une *coupure* d'un graphe (*cut* dans la littérature anglo-saxonne) comme une partition des sommets de G en deux sous-ensembles disjoints dont l'un, S, contient s et l'autre, T, contient t. La valeur du flot associée à une coupure (S, T) est la quantité : $\Phi(S, T) = \sum_{(\sigma,\tau)\in A} \phi(\sigma, \tau) - \sum_{(\tau,\sigma)\in A} \phi(\tau, \sigma)$, où $\sigma \in S$ et $\tau \in T$. On définit également la *capacité*, $\gamma(S, T)$, d'une coupure comme la quantité :

$$\gamma(S, T) = \sum_{(\sigma,\tau)\in A} \gamma(\sigma, \tau)$$

, où $\sigma \in S$ et $\tau \in T$.

On vérifie quelques propriétés simples.

PROPOSITION 3.1. *Étant donné un flot ϕ, toutes les coupures (S, T) ont la même valeur de flot $\Phi \leq \gamma(S, T)$. Si, de plus, s n'a pas d'arc entrant et t n'a pas d'arc sortant, alors $\Phi = \sum_{(s,x)\in A} \phi(s, x) = \sum_{(y,t)\in A} \phi(y, t)$.*

La preuve se fait en considérant une coupure (S, T) et en la transformant en $(S \cup \{x\}, T \backslash \{x\})$ pour $x \in T$. On vérifie alors que la nouvelle coupure possède la même valeur de flot Φ en appliquant la loi de Kirchoff sur le sommet x.

Une conséquence immédiate est alors que le flot maximal ne saurait excéder la capacité d'une coupure.

PROPOSITION 3.2. $\max_\phi \Phi \leq \min_{(S,T)} \gamma(S, T)$.

Il reste à obtenir une borne inférieure pour un flot maximal. Pour ce faire une notion naturelle est celle de *chemin saturé*. Etant donné un graphe avec capacités $G = (X, A, \gamma)$ et un flot ϕ, un arc a est *saturé* si $\phi(a) = \gamma(a)$. Un chemin est saturé s'il contient au moins un arc saturé. Il est clair que s'il existe un chemin non saturé qui joint s à t alors on peut améliorer le flot en ajoutant à chaque arc de ce chemin une quantité de flot égale au minimum de $\gamma(a) - \phi(a)$ pour l'ensemble des arcs du chemin. Un *flot est saturé* si tout chemin qui mène de s à t est saturé. Une première intuition engage à penser qu'un flot saturé

[1] on peut en fait considérer des réels, mais il est plus simple de supposer les capacités et les flots entiers et cette restriction ne gène pas dans la pratique

est maximal, or il n'en est rien comme le montre l'exemple de la figure ci-dessous, où les capacités sont indiquées en caractère gras, et le flot ϕ à côté des capacités. Le flot est saturé mais on peut l'améliorer en ajoutant une unité de flot sur les arcs $(s, x_3), (x_3, x_4), (x_1, x_2)$ et (x_2, t) et en faisant passer un flot nul sur l'arc (x_1, x_4).

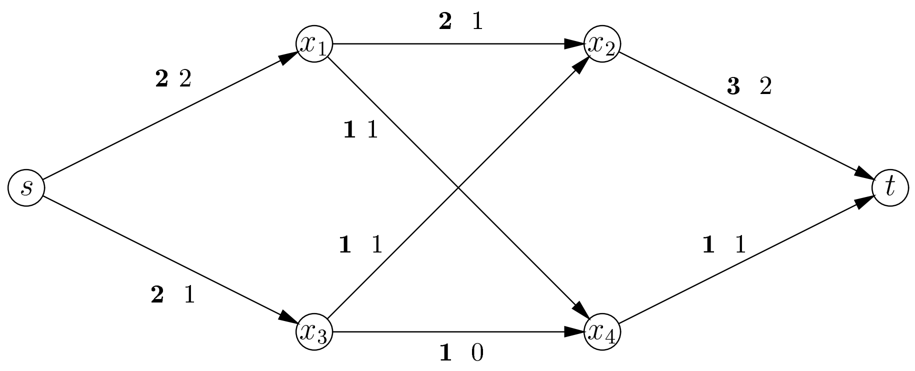


FIG. 1. Un flot saturé non maximal

2. Méthode de Ford-Fulkerson

Pour am'eliorer un flot saturé il faut, comme on l'a vu sur l'exemple plus haut , diminuer sa valeur sur certains arcs pour pouvoir l'augmenter sur d'autres. Pour cela on construit un graphe G_ϕ associé au flot ϕ sur le graphe G, qui a même sommets que G et dont les arcs sont obtenus à partir de ceux de G en conservant ceux qui ne sont pas saturés et en *retournant* les arcs contenant une quantité non nulle de flot. On appelle parfois ce graphe le *graphe des améliorations*

Plus précisément, $G_\phi = (X, A_\phi, \gamma_\phi)$ est tel que :

$$(x, y) \in A_\phi \quad \text{si} \quad \begin{cases} (x, y) \in A & \text{et} \quad \phi(x, y) < \gamma(x, y) \\ (y, x) \in A & \text{et} \quad \phi(x, y) > 0 \end{cases}$$

De plus on a les nouvelles capacités γ_ϕ données par

$$\gamma_\phi(x, y) = \quad \begin{cases} \gamma(x, y) - \phi(x, y) & \text{si} \quad \phi(x, y) < \gamma(x, y) \\ \phi(y, x) & \text{si} \quad \phi(y, x) > 0 \end{cases}$$

Un chemin c dans G_ϕ joignant s à t permet d'obtenir un flot ϕ' qui améliore ϕ. Soit u la plus petite valeur de $\gamma_\phi(x, y)$ pour tous les arcs appartenant au chemin c, on calcule alors ϕ' par

$$\phi'(x, y) = \quad \begin{cases} \phi(x, y) & \text{si} \quad (x, y) \notin c \\ \phi(x, y) + u & \text{si} \quad (x, y) \in A \quad \text{et} \quad (x, y) \in c \\ \phi(y, x) - u & \text{si} \quad (x, y) \in A \quad \text{et} \quad (y, x) \in c \end{cases}$$

Ceci fournit une technique très générale de construction par amélioration successive des flots, paradigme dû à Ford et Fulkerson :

```
initialiser le flot à 0
tant qu'il existe un chemin améliorable p dans G_φ
faire améliorer le flot sur p.
```

Il suffit de vérifier que pour chaque arc la capacité n'est pas dépassée et que le flot sur une arête de G n'est pas négatif. La valeur du nouveau flot est évidemment la somme des valeurs et est en croissance, car on ne saurait diminuer le flot partant de s.

La validité de la méthode est garantie par le fait qu'on obtient ainsi un nouveau flot compatible avec les contraintes de capacité sur G. En effet, il suffit de vérifier que pour chaque arc la capacité n'est pas dépassée et que le flot sur une arête de G n'est pas négatif. La valeur du nouveau flot est évidemment la somme des valeurs et est en croissance, car on ne saurait diminuer le flot partant de s. Il reste à montrer que le flot obtenu à la fin de l'algorithme est maximum pour cela on utilise :
Preuve : Il suffit de vérifier que pour chaque arc la capacité n'est pas dépassée et que le flot sur une arête de G n'est pas négatif. La valeur du nouveau flot est évidemment la somme des valeurs et est en croissance, car on ne saurait diminuer le flot partant de s.

Théorème 3.1 (Maxflow-Mincut). *Soit ϕ un flot de s vers t dans un graphe G. Les trois conditions suivantes sont équivalentes :*
1. le flot ϕ est maximal ;
2. le graphe d'amélioration G_ϕ ne contient pas de chemin joignant s à t ;
3. la valeur Φ de ϕ est égale à la capacité $\Gamma(S,T)$ pour une certaine coupure (S,T).

Preuve : (1) entraine (2) car si G' avait un chemin améliorable on pourrait augmenter le flot d'après la proposition précédente.
(2) entraine (3) ; considérons l'ensemble S des sommets accessibles de s dans G' par un chemin formé d'arcs de capacité *non nulle*, et soit $T = X \backslash S$, son complémentaire, qui est bien sûr non vide puisque contenant t. S étant maximal, les arêtes de S à T ont toutes une capacité nulle dans G'. Et donc $\Phi = \Gamma(S,T)$.
Enfin, (3) entraine (1) puisque le maximum possible est atteint.

Pour résumer l'algorithme de Ford et Fulkerson s'écrit comme suit et a pour complexité $O(m\Phi^\star)$, où m est le nombre d'arcs, $\Phi^\star$ est le flot maximal et en supposant que toutes les capacités sont entières.

```
FORD-FULKERSON(G,s,t)
pour chaque (x,y) de V faire φ(x,y) :=0 ;
tant qu'il existe un chemin p de s à t dans G' faire
        h :=min capacité d'amélioration des arcs de p
        pour chaque arc a de p faire φ(a) :=φ(a)+h ;
```

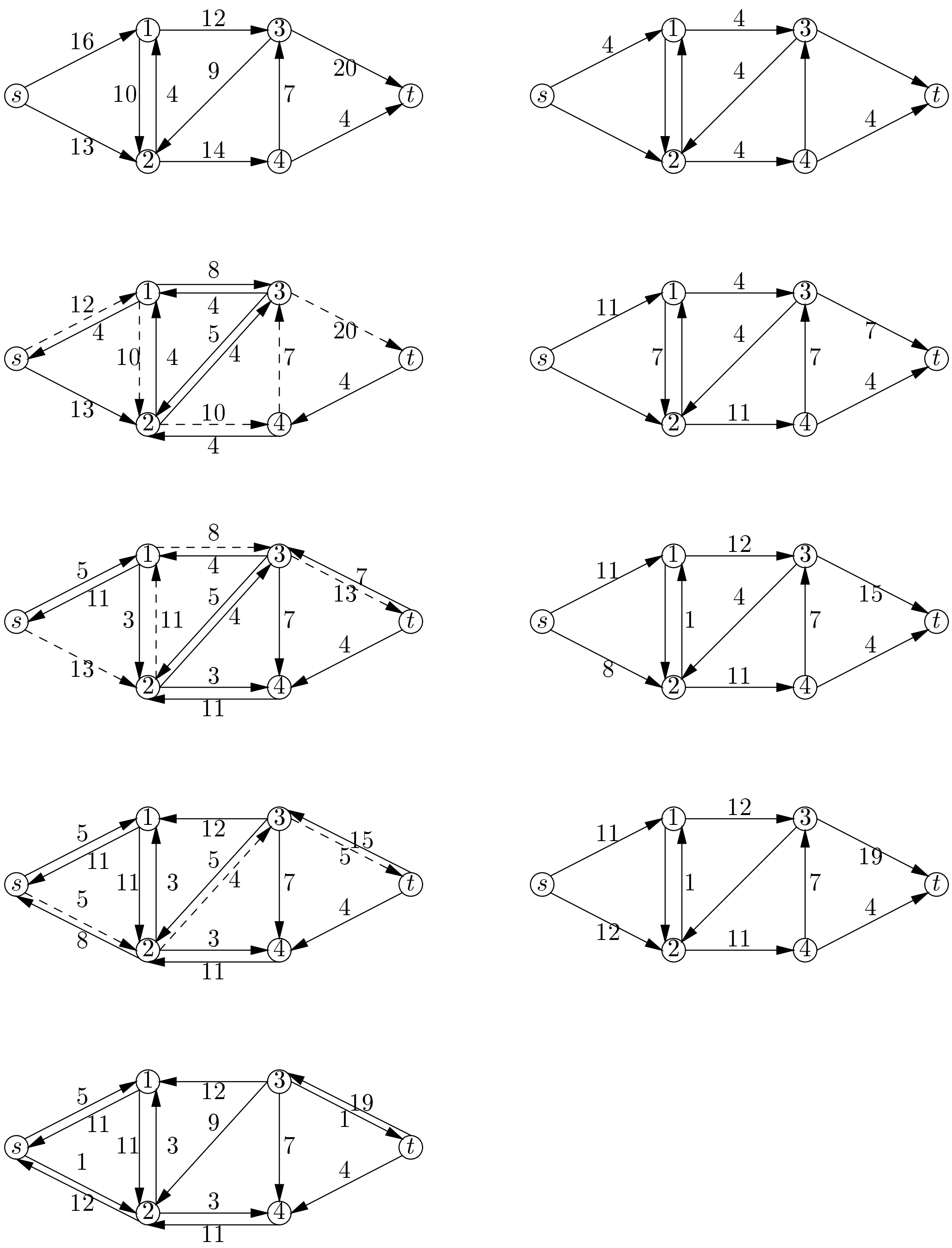
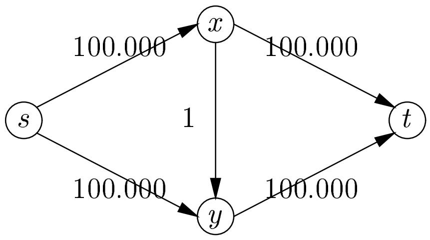


FIG. 2. Application de l'algorithme

L'exemple plus haut montre une des étapes de l'algorithme sur un graphe simple, sur la figure un graphe avec les capacités est donné, puis on a fait figuré sur la partie gauche le graphe des améliorations et sur la partie droite les flots calculés au fur et à mesure des améliorations.

Remarque La complexité de cette méthode dépend beaucoup de la façon dont les améliorations sont effectuées. Sur le graphe pathologique suivant dont le flot maximal est de 200.000, le choix systématique d'un chemin passant par la diagonale (x, y) de capacité 1 conduit à faire 200.000 d'itérations alors que le choix direct des chemins (s, x, t) puis (s, y, t) permettrait de conclure en 2 étapes.



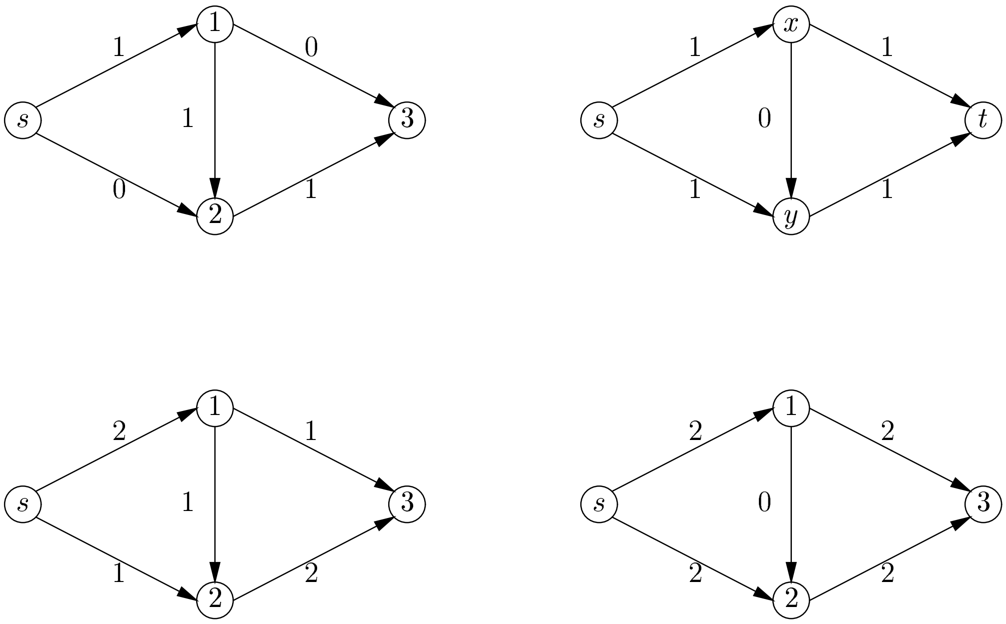


Fig. 3. Un exemple de calcul très long

3. Des algorithmes en temps polynomial

Pour rendre le comportement de l'algorithme moins dépendant des données il faut préciser la stratégie de choix des chemins sur lesquels se font les augmentations du flot. Nous en décrivons deux, et il est même possible de calculer le flot maximal en temps $O(n^3)$.

3.1. L'algorithme de Edmonds-Karp. Dans cette implantation, les chemins améliorables sont énumérés selon leur longueur au moyen d'une recherche en *largeur* des sommets accessibles depuis s dans le graphe G'.

PROPOSITION 3.3. *Dans l'algorithme de Edmonds-Karp, la longueur du chemin amélioré ne décroît jamais.*

Preuve : Supposons qu'à une étape la longueur du chemin améliorable vienne à diminuer. Ceci veut donc dire qu'il existe un sommet x dont les distances à s dans les deux graphes successifs G et G' sont telles que $d(s,x) > d'(s,x)$ et choisissons, à cette étape, un sommet x qui soit le plus proche de s. Remarquons que sur le chemin améliorable p de s à t qui passe par x, le prédécesseur y de x satisfait $d(s,y) \leq d'(s,y)$.
Considérons alors le flot $\phi(y,x)$ avant l'augmentation. Si $\phi(y,x) < \gamma(y,x)$, capacité de l'arête, alors $d(s,x) \leq d(s,y)+1 \leq d'(s,y)+1 = d'(s,x)$. Ce qui contredit le choix de x. Donc $\phi(y,x) = \gamma(y,x)$ et (y,x) n'est pas une arête du graphe d'amélioration de G ; mais comme (x,y) en est une dans celui de G', il faut donc que (x,y) soit une arête du chemin amélioré dans le passage de G à G'. Alors et comme ce chemin est de longueur minimale : $d(s,x) = d(s,y)-1 \leq d'(s,y)-1 = d'(s,x)-2 < d'(s,x)$. Contradiction.

Appelons *phase* une succession d'augmentations du flot pendant lesquelles la longueur des chemins améliorés reste fixe. Le nombre de phases est au plus le nombre n de sommets du graphe. De plus dans chaque phase une arête (x,y) ne peut être critique (déterminant l'augmentation du chemin choisi) qu'une seule fois, puisqu'il aura fallu modifier entre temps (y,x) pour permettre un nouvel usage de (x,y) : du même coup x s'éloigne de s et la longueur des chemins croît. Il y a donc $O(nm)$ améliorations apportées au flot, d'où le théorème :

THÉORÈME 3.2. *Le coût de l'algorithme de Edmonds-Karp est $O(nm^2)$ opérations.*

En effet le coût de la recherche en largeur d'un chemin améliorable est proportionnel au nombre d'arêtes du graphe.

4. Couplage dans un graphe biparti

Soit $G = (X,E)$ un graphe symétrique, un *couplage* (on dit *matching* en anglais) est un sous-ensemble C de E tel que deux arêtes de C n'aient pas de sommet commun. Un exemple de couplage est donné sur la figure ci-dessous (les arêtes du couplage sont en traits gras). Dans la suite on dira qu'un sommet est *saturé* s'il appartient à une arête du couplage.

Une remarque importante sur le couplage est la suivante :

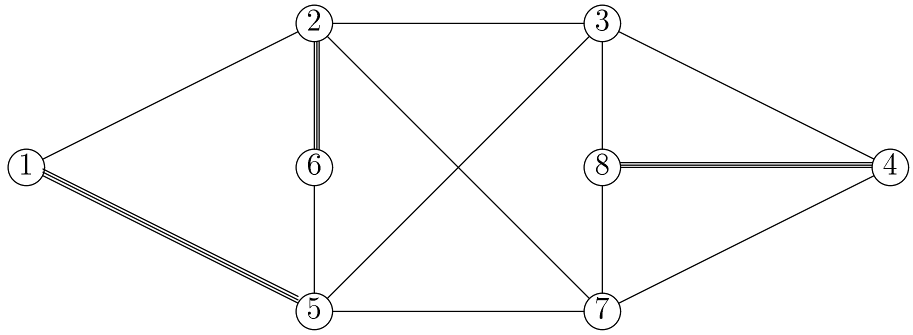


FIG. 4. Un couplage (non maximum)

Remarque Soient C et D deux couplages de $G = (X, E)$, alors la différence symétrique $C \oplus D$:

$$C \oplus D = \{e \in E | e \in Ce, \notin D\} \cup \{e \in E | e \in D, e \notin C\}$$

est formée de chemins et de cycles. Chacun de ces chemins ou cycles est composé alternativement d'une arête de C et d'une arête de D. Si $|C| > |D|$ alors il existe une composante connexe du graphe $(X, C \oplus D)$ constituée d'un chemin de longueur impaire $e_1, e_2, \ldots, e_{2p+1}$ tel que $e_1, e_3 \ldots e_{2p+1} \in C$, et$e_2, e_4, \ldots, e_{2p} \in D$. Une des extrémités de e_1 et une de e_{2p+1} ne sont incidents à aucun arc de D. On dit que le chemin est augmentable pour D, car on peut augmenter le nombre d'éléments de D en le remplaçant par $D' = D \cup \{e_1, e_3, \ldots, e_{2p+1}\} \setminus \{e_2, e_4, \ldots, e_{2p}\}$.

Dans ce paragraphe on s'intéresse à la recherche d'un couplage maximum dans le cas où le graphe G est biparti. L'ensemble des sommets est alors composé de deux sous ensembles disjoints X et Y et chaque arête joint un sommet de X et un sommet de Y. Les graphes bipartis modélisent le problème de *l'affectation*. Un exemple typique d'un tel problème est celui de l'affectation de stages à des étudiants, l'ensemble X représente les étudiants, Y les stages proposés et les arêtes du graphe relient chaque étudiant aux stages qui font partie de sa liste de choix. Le but est d'obtenir un couplage maximal dans lequel chaque étudiant est couplé avec un stage qui fait partie de sa liste.

Dans le cas particulier des graphes bipartis la recherche d'un couplage maximal dans un graphe $G = (X, E)$ se ramène à celle d'un flot maximum dans un graphe orienté G' qui se déduit de G de la façon suivante : L'ensemble des sommets de G' est $X' = X \cup Y \cup \{s, t\}$ où s et t sont deux nouveaux sommets. L'ensemble des arcs de G' est constitué des arêtes de G orientées de X vers Y, d'arcs ayant pour origine s et pour extrémité chaque sommet de X, d'arcs reliant chaque sommet de Y à t. On suppose de plus que toutes les capacités sont égales à 1. Il est assez facile de vérifier qu'à tout couplage C dans G on associe un flot ϕ_C dans G' en posant $\phi_c(u, v) = 1$ si et seulement l'une des conditions suivantes est satisfaite :

- $\{u, v\} \in C$
- $u = s$ et $v \in X$ est couplé (il existe $y \in Y$ tel que $\{v, y\} \in C$

– $v = t$ et $u \in Y$ est couplé (il existe $x \in X$ tel que $\{x, u\} \in C$
si aucune de ces conditions n'est vérifiée $\phi_C(u, v) = 0$. On vérifie que ϕ_C est un flot et que sa valeur est égale au nombre d'arêtes dans C, réciproquement tout flot ϕ dans G' détermine un couplage de G.

La notion de chemin améliorable se traduit simplement en terme de couplage. Un chemin $P = e_1, e_2, \ldots e_k$ est *alternant* si $e_{2i+1} \notin C$ et si $e_{2i} \in C$ pour $i \leq k/2$. En reprenant la notion de sommet saturé (extrémité d'une arête du couplage) :

THÉORÈME 3.3. *Un couplage est maximal si et seulement s'il n'existe pas de chemin alternant reliant deux sommets non saturés.*

Preuve : Si $P = e_1, e_2, \ldots e_k$ est un chemin alternant reliant deux sommets non saturés alors

$$C \oplus P = C \cup \{e_1\} - e_2 \cup \{e_3\} - e_4 \ldots \cup \{e_k\}$$

est un couplage contenant une arête de plus que C. Réciproquement, soit C un couplage non maximal et soit D un couplage tel que $|D| > |C|$ alors d'après la remarque préliminaire $C \oplus D$ contient une composante connexe qui est chemin alternant les arêtes de C et D où il y a une arête de plus dans C que dans D. Ce chemin constitue un chemin alternant reliant deux sommets non saturés dans C.

Le théorème et sa preuve suggèrent l'algorithme suivant

```
Initialiser C par l'ensemble vide
Tant qu'(il existe un chemin P ameliorable)
       { Ameliorer (P);}
```

5. Mariages stables.

Une question qui généralise celle du couplage est celle de la recherche de couplages (ou mariages) *stables*. Cette notion constitue un chapitre important de la théorie des jeux.

Les données du problème sont un peu plus complexes que celles du simple couplage, on se donne deux ensembles $X = \{x_1, x_2, \ldots, x_p\}$ et $Y = \{y_1, y_2, \ldots, y_p\}$, pour chaque $i = 1, \ldots, p$ une liste ordonnée F_i d'éléments de Y et pour chaque $j = 1, \ldots, q$ une liste ordonnée G_j d'éléments de X. Ces listes ne sont pas vides mais ne contiennent pas nécessairement tous les éléments de Y et de X respectivement. On note $y \prec_i y'$ si y est avant y' dans la liste F_i ou si y' ne figure pas dans la liste F_i alors que y s'y trouve. Un couplage est alors un sous-ensemble de couples (x, y) tel que y figure dans la liste F_x et x dans la liste G_y. La recherche d'un couplage maximal consiste à travailler dans le graphe biparti (X, Y, E) où les arêtes sont tous les couples $\{x, y\}$ tels que y figure dans la liste F_x et x dans la liste G_y, et à y appliquer les algorithmes vus au paragraphe **4**.

Pour introduire la notion de stabilité il est commode de considérer un couplage comme une application partielle α de X dans Y, on note alors β son inverse, ainsi les éléments du couplage sont les couples $(x, \alpha(x))$ ou de manière équivalente les couples $(\beta(y), y)$. Un couplage C est stable si et seulement si pour tout couple x, y qui n'est pas dans C on a

$$\beta(y) \prec_y x \qquad \text{ou} \qquad \alpha(x) \prec_x y$$

On interprète ceci en disant qu'un couplage est stable s'il n'existe pas deux individus qui se préfèrent mutuellement à leur conjoints respectifs.

Exemple : Considérons le cas $p = q = 3$ et les listes

$$\begin{array}{lcccc} F_1 & : & y_2 & y_1 & y_3 \\ F_2 & : & y_1 & y_3 & y_2 \\ F_3 & : & y_1 & y_2 & y_3 \end{array} \qquad\qquad \begin{array}{lcccc} G_1 & : & x_1 & x_3 & x_2 \\ G_2 & : & x_3 & x_1 & x_2 \\ G_3 & : & x_1 & x_3 & x_2 \end{array}$$

Le couplage $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ n'est pas stable en raison de x_1 et y_2 qui se préfèrent à y_1 et x_2 respectivement. Par contre le couplage $(x_1, y_1), (x_2, y_3), (x_3, y_2)$ est stable.

On montre que l'algorithme suivant donne un couplage stable.

On initialise $\alpha(x)$ au premier élément de la liste F_x et $\beta(y)$ indéfini. Ensuite on effectue des modifications itérativement de β et de α comme suit :

(1) Pour chaque y choisir l'élément x le plus intéressant dans la liste G_y parmi les x tels que $\alpha(x) = y$. et poser $\beta(y) = x$
(2) Pour chaque x,
   - S'il existe y tel que $\beta(y) = x$, alors $\alpha(x) = y$.
   - S'il n'existe pas de y tel que $\beta(y) = x$ et si $\alpha(x)$ n'est pas le dernier élément de la liste F_x alors remplacer $\alpha(x)$ par l'élément qui le suit dans dans la liste F_x,
   - Si aucune des deux conditions précédentes n'est remplie alors $\alpha(x)$ est indéfini.

6. Couplages et matroïdes

Soit $G = (X, Y, E)$, un graphe biparti et soit $\mathcal{C}$ la famille des sous-ensembles C de E qui sont des couplages. Alors la famille $E, \mathcal{C}$ ne constitue pas un matroïde. En effet, considérons le graphe biparti G composé de 4 sommets $1, 2, 3, 4$, $X = \{1, 3\}$ et $Y = \{2, 4\}$, et de 3 arêtes $(1, 2), (2, 3), (3, 4)$, alors sur les ensembles $C_1 = \{(1, 2), (3, 4)\}$ et $C_2 = \{(2, 3)\}$ on constate que C_1 possède un élément de plus que C_2 et pour tout $e \in C_1$, $C_2 \cup \{e\}$ n'est pas un couplage ce qui contredit les axiomes des matroïdes. Par contre on a :

Proposition 3.4. *La famille $\mathcal{C}$ est l'intersection de deux matroïdes.*

Preuve : Considérons les deux familles de parties $\mathcal{F}$ et $\mathcal{G}$ de E telles que $F \in \mathcal{F}$ si et seulement si deux arêtes de F n'ont pas une extrémité commune dans X. De même on définit $\mathcal{G}$ comme l'ensemble des parties G de E telles que $G \in \mathcal{F}$ si et seulement si deux arêtes de G n'ont pas une extrémité commune dans Y. Il est alors clair que C est un couplage si et seulement si $C \in \mathcal{F}$ et $C \in \mathcal{G}$.

L'algorithme de recherche du couplage maximum se généralise pour déterminer étant donnés deux matroïdes $(E, \mathcal{F})$ et $(E, \mathcal{G})$ l'ensemble C de cardinal maximal qui appartienne à la fois à $\mathcal{F}$ et $\mathcal{G}$. On utilise la notion de *circuit* dans un matroïde $(E, \mathcal{F}$. Rappelons qu'un ensemble C est un circuit de $(E, \mathcal{F})$ si $C \notin \mathcal{F}$ et si tout sous-ensemble strict de C est dans $\mathcal{F}$. Il est alors clair que tout sous ensemble U de E qui n'est pas dans $\mathcal{F}$ contient un circuit.

Pour obtenir le sous-ensemble de cardinal maximal de on définit la notion suivante : Soit F un ensemble qui appartient à $\mathcal{F}$ et $\mathcal{G}$, on associe à une suite d'éléments $e_1, e_2, \ldots, e_p$ les circuits C_i et D_i. C_i est un circuit inclus dans $F \cup \{e_i\}$ pour le matroïde $(E, \mathcal{F})$; il est défini chaque fois que cet ensemble n'est pas dans $\mathcal{F}$. Symétriquement D_i est un circuit inclus dans $F \cup \{e_i\}$ pour le matroïde $E, \mathcal{G}$; il est défini chaque fois que cet ensemble n'est pas dans $\mathcal{G}$.

Une suite $P = \{e_1, e_2, \ldots e_p\}$ est augmentante si les conditions suivantes sont satisfaites :
- $F \cup \{e_1\} \in \mathcal{F}$
- Pour tout i pair compris entre 2 et k $e_i \in F \cap D_{i-1}$. De plus e_i n'appartient pas à D_j pour j impair strictement plus petit que $i - 1$.
- Pour tout i impair compris entre 3 et k, $e_i \notin F$ et $e_{i-1} \in C_i$. De plus e_{j-1} n'appartient pas à C_i pour j impair strictement plus petit que i.
- $F \oplus P$ est un élément de $\mathcal{G}$.

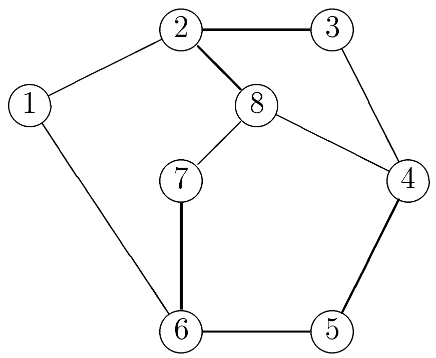
On constate alors que $F \oplus P$ appartient aussi à $\mathcal{F}$ et possède un élément de plus que F. L'itération de cette augmentation donne un ensemble appartenant à l'intersection de taille maximum.

7. Couplages généraux

On considère maintenant des graphes généraux, pour lesquels on veut déterminer un ensemble d'arêtes deux à deux disjointes et de cardinalité maximale. On généralise ainsi les problèmes d'affectation, en prenant en compte par exemple les problèmes de tournée (postier chinois) ou de chaînes. Des applications typiques sont les transitions de phase en magnétisme, le regroupement de particules dans des phénomènes de désintégrations simultanées, etc.

7.1. Principes. Étant donné un graphe *non orienté* $G = (X, A)$, on appelle couplage un sous-ensemble C de A tel que deux arêtes quelconques de C n'ont pas de point commun. Le couplage C étant donné, on dit qu'il *sature* les sommets extrémités de ses arêtes.

Appelons *chaîne* de G toute suite d'arêtes formant un chemin sans boucle. Une chaîne S est dite *alternée* par rapport à un couplage C si ses arêtes sont successivement dans C et dans $A \backslash C$. Supposons de plus que S soit telle que ses extrémités sont ou bien insaturées ou bien extrémité d'une arête commune à C et S. Alors la différence symétrique $C' = C \Delta S$ forme encore un couplage, dans lequel les arêtes de S ont simplement échangé leur rôle. Cette opération est appelée *transfert* le long de S. On remarquera que si les *deux* extrémités de S sont insaturées le nouveau couplage C' est plus riche d'une arête. Une telle chaîne est dite *augmentante*.



Exemple : Donner une chaîne augmentante sur le graphe suivant où le couplage est formé des arêtes $\{2,3\}, \{4,5\}, \{6,7\}$.

7.2. Couplages maximaux. Un théorème de Berge caractérise les couplages maximaux. Démontrons d'abord un lemme technique.

LEMME 3.1. *Soient C et C' deux couplages distincts d'un graphe $G = (X, A)$. Soit $B = C \oplus C'$ et $\Gamma = (X, B)$ le graphe construit sur cet ensemble B d'arêtes. Les composantes connexes de Γ sont soit des points isolés, soit des cycles ou des chaînes alternés d'arêtes de C et C'.*

Preuve : Le degré maximal de Γ est 2.

Ce lemme permet de prouver la condition suffisante.

THÉORÈME 3.4. *Un couplage C est maximal si et seulement s'il n'existe pas de chaîne alternée augmentante.*

Preuve : Compte tenu de la remarque ci-dessus l'existence d'une chaîne alternée augmentante permet de construire un couplage plus grand par transfert le long de cette chaîne, ce qui réfute l'hypothèse de maximalité.

Réciproquement, soit C n'ayant pas de chaîne alternée augmentante et soit C' un couplage maximal (donc n'ayant pas de chaîne alternée augmentante). Considérons le graphe d'arêtes $B = C \Delta C'$ évoqué dans le lemme. Il ne peut posséder de chaîne alternée impaire et donc C est maximal.

On remarquera enfin qu'un couplage maximal ne peut laisser qu'un seul sommet insaturé et que les couplages maximaux s'échangent par transfert sur des cycles ou des chaînes alternées pairs.

7.3. Algorithme de calcul d'un couplage maximal. Cet algorithme est relativement complexe car il repose sur l'assemblage de couplages locaux et sur de nombreux transferts. On retrouvera donc deux types de recherche sur des cycles et des chaînes ; on essaiera d'organiser la croissance des couplages selon des arbres alternés reliant des cycles. Dans cette perspective, le concept de contraction d'un graphe est utile.

Soit $G = (X, A)$ un graphe ; pour $X' \subset X$ on définit un nouveau graphe $G' = G/X'$ dont l'ensemble des sommets est $X \setminus X' \cup \{\star\}$ et les arêtes sont les mêmes en dehors de X' et pointent toutes sur le nouveau sommet $\star$ sinon. Le sommet $\star$ représente tous les sommets de X'.

L'algorithme s'appuie sur les propriétés suivantes.

LEMME 3.2. *Soit $X' \subset X$ un ensemble de sommets formant un cycle impair K. Si C est un couplage de G/X' il existe un couplage maximal C_K de K tel que $C \cup C_K$ est un couplage de G.*

LEMME 3.3. *Soit C est un couplage laissant deux sommets insaturés. Soit K un cycle impair sur les sommets X' et C_K un de ses couplages maximaux. Alors C est un couplage maximal de G si et seulement si $C \setminus C'$ est un couplage maximal de G/X'.*

Le lemme 2 est immédiat. Le lemme 3 se prouve en considérant la nature de la chaîne alternée qui mène du sommet non saturé de K à l'autre sommet non saturé de G.

7.4. Recherche des arbres alternés. Le problème est de déterminer dans le cas d'un couplage qui laisse plusieurs sommets insaturés s'il existe un chemin augmentant ou non. Le concept d'arbre alterné étend immédiatement celui de chaîne : chacune des branches de l'arbre est une chaîne alternée de longueur paire et la racine est un sommet insaturé (de fait le seul pour cet arbre).

Le processus construit progressivement des arêtes de l'arbre tout en contractant le graphe.

L'algorithme détaillé est un peu complexe et se trouve dans les ouvrages de référence.

CHAPITRE 4

Programmation linéaire

Nous nous attaquons à l'un des chapitres les plus classiques de l'optimisation combinatoire.

Le problème traité peut se présenter sous la forme imagée suivante : une entreprise fabrique des produits $P_1, P_2, \ldots, P_n$ à l'aide de matières premières $M_1, M_2, \ldots, M_m$. Chaque unité du produit P_j rapporte un bénéfice c_j et utilise $a_{i,j}$ quantité de la matière première M_i ; la matière M_i est disponible au total en quantité b_i pour l'ensemble des produits. La question question posée consiste à déterminer pour tout j la quantité x_j de produit P_j à produire de façon à réaliser un bénéfice maximal tout en satisfaisant aux contraintes de quantité.

On doit ainsi maximiser $\sum_{j=1}^{n} c_j x_j$ en satisfaisant les m contraintes $\sum_{j=1}^{n} a_{i,j} x_i \leq b_j$. On voit bien que cette question se rencontre très souvent dans la pratique, d'aileurs les equipes de chercheurs qui ont réalisé des logiciels efficaces dans ce domaine ont obtenu d'excellents résultats commerciaux.

George B. Dantzig a été un des pionniers du domaine et a le premier proposé une méthode de résolution pour ce qui reste un des grands paradigmes de la modélisation et de la programmation (ce terme désigne dans le contexte l'organisation des calculs et non pas la réalisation d'un programme informatique).

Si l'on se place d'un point de vue de mathématicien, il s'agit de trouver un point dans un polyèdre de R^n (donné par un système de m inéquations) point qui optimise une fonction linéaire des coordonnées (appelée fonction économique). Ce problème peut donc paraître particulièrement élémentaire à ceux des mathématiciens qui ne se posent pas la question de connaître l'efficacité des méthodes pour résoudre un problème.

En effet, quelques considérations simples montrent que si le polyèdre est borné, l'optimum est atteint sur un de ses sommets. Il suffirait donc de résoudre tous les systèmes non singuliers de n équations tirées de ces m inéquations, et après avoir vérifié que la solution satisfait les autres inéquations, de tabuler la valeur de la fonction économique en ce point. Le problème est alors terminé en recherchant la valeur maximale de la table.

Pour ce qui concerne notre point de vue algorithmique, on ne saurait se contenter de cette façon de traiter le problème car il faut se poser des questions de complexité, l'algorithme envisagé demande un nombre exponentiel de calculs : on doit résoudre un nombre de systèmes égal au coefficient binomial C_m^n qui croît exponentiellement avec m et n.

Après plusieurs années de recherches dans ce domaine on a pu montrer que ce problème est exemplaire car il figure parmi les cas limites qui incarnent les barrières et sauts de complexité, on a longtemps pensé qu'il n'existait pas d'algorithme polynomial pour le résoudre et maintenant on sait que tout en étant polynomial il reste un des plus compliqués de sa catégorie. Par ailleurs, l'algorithme non polynomial classique, la *méthode du simplexe*, reste expérimentalement excellent (on montre d'ailleurs qu'avec une certaine distribution de probabilité des données la méthode est en moyenne polynomiale). Par contre les méthodes polynomiales demandent un énorme effort de compréhension des géométries sous-jacentes et sont pour les valeurs usuelles de m et n inutilisables. Il faudrait que m et n croissent considérablement pour qu'elles deviennent meilleures que la méthode du simplexe, mais dans ce cas toutes les méthodes connues sont impratiquables.

1. Présentation du problème

Etant donnés des nombres $a_{i,j}, b_i, c_j$, pour $1 \leq i \leq m,\ 1 \leq j \leq n$, soient $x_1, x_2, \ldots, x_n$ des inconnues astreintes à un ensemble de m contraintes pour $1 \leq i \leq m$:

$$\sum_{1 \leq j \leq n} a_{ji} x_j \leq b_i$$

On peut supposer que l'on travaille soit avec des reéls soit avec des rationnels, les résultats théoriques donnés ci-dessous sont valables dans les deux cas ; toutefois, lorsqu'on s'interesse aux questions de complexité, il vaut mieux se placer dans le corps des rationnels car on peut alors mesurer la taille des données par le nombre de bits nécessaires pour représenter le numérateur et le dénominateur.

On cherche dans le domaine ainsi défini les points qui optimisent une forme linéaire, appelée fonction économique ou fonction d'objectif,

$$\phi(x) = \sum_{j=1}^{n} c_j x_j$$

On écrit usuellement le problème de façon synthétique au moyen d'une notation matricielle :

$$(\mathcal{L}P) \begin{cases} \max\ cx \\ Ax \leq b \end{cases}$$

où $c, x \in \mathtt{R}^n$, $b \in \mathtt{R}^m$ et $A \in \mathtt{R}^{m \times n}$.

Dans la suite on adopte cette notation pratique qui consiste à ecrire $u \leq v$ pour deux vecteurs de $\mathtt{R}^n$ pour dire que pour tout $1 \leq j \leq n$ on a $u_i \leq v_j$, ainsi $u \geq 0$ signifiera que toutes les coordonnées de u sont positives ou nulles.

Géométriquement, chaque contrainte défini un hyperplan, l'ensemble de points qui satisfont la condition $Ax \leq b$ est alors l'intersection d'un nombre fini de demis espaces définis par les hyperplans donnés par les contraintes. On appelle un tel domaine un *polyèdre* le problème se traduit en la recherche du maximum de la forme linéaire $\phi(x) = cx$ à l'intérieur du polyèdre $\mathcal{C}$ des contraintes $Ax \leq b$.

1.1. Remarques géométriques. Rappelons quelques concepts simples :
– un sous-ensemble C de $\mathtt{R}^n$ est convexe si et seulement si il contient tout segment reliant deux de ses points, soit formellement : $\forall x, y \in C,\ \forall \lambda \in [0,1],\ \lambda x + (1-\lambda)y \in C$;
– l'enveloppe convexe d'un ensemble de points M est le plus petit convexe qui contient M, on le note ici $CH(M)$: si M est fini, cette enveloppe est un appelée *un polytope* dont les sommets sont des points de M ;
– le cône convexe engendré par M est l'ensemble des combinaisons linéaires positives de M ;
– étant donnés un ensemble X et H un hyperplan de $\mathtt{R}^n$, on dit que H est délimitant pour X si et seulement si $H \cap X \neq \emptyset$ et tous les points de X sont placés d'un même côté de H : si de plus $X \not\subset H$, on dit que $X \cap H$ est une face de X ;
– un sommet de X est une face réduite à un point, une facette une face de dimension $n-1$;

Les propriétés du système $\mathcal{L}P$ peuvent être résumées géométriquement comme suit.

PROPOSITION 4.1. *L'ensemble R des points qui satisfont les contraintes $Ax \leq b$ d'un système $\mathcal{L}P$, vérifie les propriétés suivantes :*
– R est un cone engendré par un polytope convexe ;
– chaque sommet est point d'intersection d'au moins n hyperplans de contrainte ;
– si R est non vide, l'ensemble S des points optimaux de $\mathcal{L}P$ contient un sommet de R ou un point à l'infini ;
– tout sommet non optimal de R est adjacent à un sommet qui améliore la fonction d'objectif.

DÉMONSTRATION. Les deux premiers points sont des conséquences immédiates des définitions.
Le troisième se démontre en deux observations :
i) si S était intérieur à R strictement ($S \cap CH(R) = \emptyset$), l'hyperplan d'équation $vx = r$ qui le contient, soit contiendrait aussi R et donc ses sommets, soit séparerait R en deux sous-ensembles non vides dont l'un contiendrait un point y vérifiant $vy > r$;
ii) donc, et à moins d'un problème dégénéré où S est de dimension n, $S \subset CH(X)$ et l'hyperplan $vx = r$ est délimitant pour R (sinon on pourrait améliorer la fonction d'objectif dans la face correspondante), et par conséquent S contient un sommet de R.
Le dernier point repose sur le fait que si la fonction d'objectif n'est pas constante sur R, et si le point considéré n'est pas un sommet dégénéré, l'hyperplan $vx = r$ qui passe par le sommet sépare R en deux parties non vides, et ainsi de suite. □

1.2. L'algorithme de base. Les énoncés précédents nous permettent de donner les grandes lignes d'un algorithme (c'est en fait un pseudo-algorithme car la terminaison n'en est pas nécessairement assurée) de résolution du problème $\mathcal{L}P$.

1. SI R EST VIDE ALORS PAS DE SOLUTION SINON DÉTERMINER UN SOMMET P_0 DE L'ENVELOPPE CONVEXE DE L'ENSEMBLE R DES POINTS RÉALISABLES

2. Calculer une suite de points P_i , sur lesquels la fonction d'objectif $\phi(P_i)$ soit croissante et tels que P_{i+1} est adjacent à P_i. Jusqu'à ce que :
- ou bien P_i est tel que $\phi(P_i) \geq \phi(Q)$ pour tout Q voisin de P_i ;
alors P_i est la solution optimale ;
- ou bien P_i est origine d'une direction non bornée sur laquelle ϕ est croissante ; alors l'optimum est à l'infini.

Il reste encore pas mal de points à préciser pour transformer ceci en un algorithme précis, il prend alors le nom d'algorithme du simplexe. La correction est évidente car il s'agit d'une transposition quasi mot à mot de la proposition 4.1

L'étape 1 est un point délicat de tous ces problèmes d'optimisation linéaire, quadratique ou autre. On donnera une méthode un peu plus loin, en se ramenant au cas particulier suivant :

$$(\mathcal{L}P^+) \left\{ \begin{array}{l} \max \ vx \\ Ax \leq b \end{array} \right.$$

où $v, x \in \mathtt{R}^n$, $b \in (\mathtt{R}^+)^m$ et $A \in \mathtt{R}^{m\times n}$. Pour ce cas l'ensemble des points réalisables R contient de manière évidente l'origine.

2. Le cas $b = 0$

Pour détailler l'algorithme du simplexe on doit donc tester s'il existe une direction le long de laquelle on peut faire croître indéfiniment la fonction économique. On s'interesse d'abord au cas $b = 0$ qui tout en étant plus simple se situe déjà au cœur du problème et donne aussi des résultats fondamentaux qui seront utilisés par la suite.

Etant donnée une matrice A et un vecteur c on cherche donc à savoir s'il existe un x tel que $Ax \leq 0$ et $cx > 0$. Dans ce cas en multipliant x par un entier λ plus grand que 1, on peut faire croître indéfiniment la fonction économique tout en satisfaisant les contraintes, en effet $(A\lambda x = \lambda Ax \leq 0, \quad c\lambda x = \lambda cx)$.

Noter qu'un tel x n'existe pas si c est combinaison linéaire à coefficients positifs des lignes $L_1, L_2, \ldots, L_m$ de A, car alors cx est négatif quand Ax a toutes ses coordonnées négatives. L'algorithme ci-dessous permet de tester ces deux conditions à la fois et montre réciproquement l'existence d'une combinaison linéaire des lignes à coefficients positifs égale à c si un tel x n'existe pas.

Soit I un ensemble de d'indices qui forme une base de l'espace engendré par les lignes de A, exprimer c comme une combinaison linéaire des lignes $L_i, \ i \in I$.

$$c = \sum_{i \in I} y_i L_i$$

Tant qu'il existe $y_i < 0$ faire

Soit h le plus petit indice tel que $y_i < 0$ et soit u le vecteur de R^n tel que $L_j u = 0$ pour $j \in I, j \neq h$ et $L_h u = -1$

Si $L_i u \leq 0$ pour $i = 1, \ldots m$ alors u est le x cherché

SINON CHOISIR s LE PLUS PETIT TEL QUE $L_s u > 0$; POSER $I := I - \{h\} \cup \{s\}$ ET EXPRIMER c COMME PLUS HAUT.

THÉORÈME 4.1. *Si c s'exprime comme une combinaison des lignes de A alors l'algorithme ci dessus donne ou bien un u tel que $Au \leq 0$ et $cu > 0$ ou bien un $y \geq 0$ tel que $c = yA$. Les deux cas sont exclusifs.*

DÉMONSTRATION. Si c ne s'exprime pas comme une combinaison des lignes de A il existe alors un x tel que $Ax = 0$ et $cx > 0$, sinon l'algorithme calcule à chaque étape des y_i et un u car la sous matrice des lignes dont les indices sont dans I est toujours non singulière. Cet u satisfait $cu > 0$ à chaque étape puisque $cu = y_h L_h u$.

Il nous reste à montrer que l'algorithme termine en vérifiant à chaque étape que le sous ensemble I calculé est différent de ceux calculés avant lui. Supposons qu'au contraire on obtienne le même I à deux étapes différentes, montrons que l'on aboutit à une contradiction. Notons r le plus grand indice qui sort de I entre ces deux étapes et posons $I = I_p$ au moment où r sort, comme on retrouve le même I plus tard c'est que r est aussi rentré entre les deux étapes considérées, notons I_q la valeur de I lorsque r entre dans I.

Posons $c = \sum_{i \in I_p} y_i L_i$ et soit u' la valeur de u trouvée lorsque r entre dans I, on a alors $u'c > 0$.

Mais on vérifie que pour $i \in I_p,\ i < r$ on a $y_i > 0$ puisque r étant sorti de $I = I_p$ c'est le plus petit tel que $y_r < 0$, de plus $u'L_i < 0$ puisque r, lors qu'il entre dans I_q est le plus petit tel que $u'L_r > 0$. Ainsi pour $i \in I_p, \quad i \leq r$ on a $y_i u' L_i \leq 0$.

D'autre part si $i \in I_p,\ i > r$, puisque r est le plus grand indice sorti, i est aussi dans I_q ; et u' a été choisi de façon telle que $L_i u' = 0$. Ainsi pour $i \in I_p, \quad i > r$ on a $y_i u' L_i = 0$.

On obtient la contradiction en vérifiant que $0 < u'c = \sum_{i \in I_p} y_i u' L_i \leq 0$. □

Le corollaire qui suit est une version simplifiée du principe fondamental de dualité qui sera vu dans le paragraphe suivant.

COROLLAIRE 4.1. *Si l'un des deux problèmes suivants*

$$\begin{array}{lll} (\mathcal{P}) & \max & \{cx \mid Ax \leq 0\} \\ (\mathcal{D}) & \exists y, & y \geq 0 \mid yA = c \end{array}$$

admet une solution finie alors il en est de même pour l'autre et $0 = \max\{cx \mid Ax \leq 0\}$.

DÉMONSTRATION. Si $\max\{cx \mid Ax \leq 0\}$ admet une solution finie alors elle est telle que $cx = 0$, en effet $x = 0$ est tel que $Ax \leq 0$ et donne $cx = 0$. S'il existe un x tel que $Ax \leq 0$ et $cx > 0$ alors en multipliant x par un réel $\lambda > 1$ on vérifie que l'ensemble des valeurs de cx n'est pas borné et il n'y a pas de solution finie. Le théorème montre enfin que l'existence d'un $y \geq 0$ tel que $yA = c$ est équivalente à $Ax \leq 0 \Rightarrow cx \leq 0$. □

Le corollaire qui suit nous sera utile dans la suite :

COROLLAIRE 4.2. *Le système d'inéquations $Ax \leq b$ admet une solution si et seulement si pour tout y :*

$$y \geq 0,\ yA = 0 \ \Rightarrow \ yb \geq 0$$

DÉMONSTRATION. On considère la matrice $A' = [I\ A\ \ -A]$.

On montre d'abord que $Ax \leq b$ a une solution si et seulement s'il existe $u \geq 0$ tel que $A'u = b$. En effet, s'il existe un $u \geq 0$ tel que $A'u = b$ alors u s'ecrit $[v\ u'\ u'']$ et $x = u' - u''$ satisfait $Ax + v = b$ soit $Ax \leq b$. Réciproquement si x satisfait $Ax \leq b$ on peut poser $x = u' - u''$ avec $u' \geq 0, u'' \geq 0$, en posant $v = Ax - b$ on observe que $[v\ u'\ u'']A' = b$.

D'après le corollaire 4.1 $A'u = b$ si et seulement si $yA' \geq 0$ implique $yb \geq 0$. Mais comme les colonnes de A' sont consistituées de celles de A, leurs opposées et celles de la matrice identité, $yA' \geq 0$ est équivalent à $y \geq 0$ et $yA = 0$ ce qui permet de conclure. □

3. Le cas b quelconque

3.1. Théorème de dualité. On a le résultat fondamental suivant qui permettra de remplacer le recherche d'un maximum par celle d'une solution à un système d'inéquations.

THÉORÈME 4.2. *Si l'un des deux problèmes suivants*

$$\begin{array}{ll} (\mathcal{P}) & \max\{cx \mid Ax \leq b\} \\ (\mathcal{D}) & \min\{by \mid y \geq 0,\ yA = c\} \end{array}$$

admet une solution finie alors il en est de même pour l'autre et on a :

$$\min\{by \mid y \geq 0,\ yA = c\} = \max\{cx \mid Ax \leq 0\}.$$

DÉMONSTRATION. Si $\mathcal{P}$ n'admet pas de solution parce que $Ax \leq b$ n'en a pas, alors d'après le corollaire 4.2 il existe $z \geq$ tel que $zA = 0$ et $zb \geq 0$. Ceci montre que pour $\lambda > 1$, l'ajout de λz a toute solution de $yA = c$ donne une autre solution y' avec $y'b < yb$, ce qui interdit une solution finie à $\mathcal{D}$.

Si $\mathcal{P}$ n'admet pas de solution parce qu'il existe x tel que $Ax \leq b$ et cx plus grand que toute valeur donnée, il ne peut y avoir de $y \geq 0$ tel que $yA = c$ car alors $cx = yAx \leq yb$.

Si $\mathcal{D}$ n'admet pas de solution parce que $y \geq 0,\ yA = c$ n'en a pas alors, d'après le théorème du paragraphe précédent, il existe un z tel que $zc \geq 0$ et $Az \leq 0$ et l'ajout de λz pour $\lambda > 1$ à un x qui satisfait $Ax \leq b$ donne $x' = x + \lambda z$ tel que $Ax' \leq b$ et $cx' > cx$.

Si $\mathcal{D}$ n'admet pas de solution parce qu'il existe $y \geq 0$ tels que $yA = c$ et yb plus petit que toute valeur donnée, il ne peut exister de x tel que $Ax \leq b$ car alors $yAx = cx \leq yb$. □

3.2. Algorithme du simplexe. On se propose d'abord de donner un algorithme qui résout le problème $\mathcal{P}$ dans le cas où l'on connaît un point x tel que $Ax \leq b$ et x est solution du système d'equations $L_i x = b_i$ pour $i \in I$. On verra plus loin comment calculer une tel point de depart. Dans la suite I désigne un ensemble d'indices de lignes et A_I la sous matrice de A qui contient les lignes dont les indices sont dans I, au cours de tout l'algorithme cette matrice est inversible.

On itére la boucle suivante jusqu'à obtenir $y \geq 0$ ou $v < 0$

SOIT y TEL QUE $y_i = (cA_I^{-1})_i$ POUR $i \in I$ ET $y_j = 0$ POUR $j \notin I$.

SI $y \geq 0$ ALORS x EST SOLUTION DU SYSTÈME.

SINON SOIT i LE PLUS PETIT INDICE TEL QUE $y_i < 0$ ET SOIT U_i LA COLONNE CORRESPONDANTE DE A_I^{-1} ; DÉTERMINER POUR CHAUQE LIGNE L_j DE LA MATRICE A

LA VALEUR $v_j = -L_j U_i$.

SI $\forall j,\ v_j < 0$ ALORS Cx EST NON BORNÉ POUR $Ax \leq b$

SINON SOIT j TEL QUE $\frac{b_j - L_j x}{v_j}$ SOIT MINIMUM. ON POSE ALORS $I := I - \{i\} \cup \{j\}$ ET ON CALCULE x, INTERSECTION DES HYPERPLANS CORRESPONDANTS.

La preuve de ce que l'algorithme du simplexe termine se fait de la même façon que la preuve du théorème 4.1. Pour que l'algorithme soit complet il nous faut obtenir un point réalisable. Pour cela on utilise le programme linéaire construit comme suit :

Soit b' le vecteur obtenu à partir de b en remplaçant les coordonnées négatives par des 0 et soit $b'' = b' - b$ de sorte que b'' est aussi à coordonnées positives ou nulles . Soit A_1 la matrice obtenue à partir de A en ne retenant que les lignes i où $b_i \geq 0$ et A_2 celle formée avec les autres lignes. Soit $\tilde{x}$ un nouveau vecteur inconnu ayant un nombre de coordonnées égal au nombre de lignes de A_2. Soit $\phi(x, \tilde{x})$ la fonction économique égale à la somme des coordonnées de $A_2 x - \tilde{x}$, soit enfin β la somme des coordonnées de b_2

PROPOSITION 4.2. *Il existe un x satisfaisant $Ax \leq b$ si et seulement si le programme*

$$\max\{\phi(x, \tilde{x})\ |A_1 x \leq b_1,\ A_2 x - \tilde{x} \leq b_2,\ -\tilde{x} \leq 0\}$$

admet un maximum égal à β.

DÉMONSTRATION. Il n'est pas difficile de voir que la valeur de $\phi(x, \tilde{x})$ est inférieure ou égale à β puisque chaque coordonnée de $A_2 x - \tilde{x}$ est inférieure ou égale à une de celle de b_2. Ce maximum est atteint si et seulement si toutes les coordonnées sont égales à celles de b_2 c'est à dire si et seulement si $A_2 x - \tilde{x} = b_2$ peut être obtenu avec $\tilde{x} \geq 0$ et $A_1 x \leq b_1$ mais ceci est équivalent à $Ax \leq b$.

□

De la proposition on déduit un procédé permettant de construire un point réalisable s'il en existe un. On applique l'algorithme du simplexe au programme défini plus haut en notant que le vecteur nul est un point réalisable pour ce nouveau programme. Si on atteint pour maximum β on a un point réalisable pour le programme initial, si le maximum atteint est inférieur à β c'est qu'il n'en existe pas.

4. Un exemple

On considère le problème :

$$\left\{\begin{array}{lcl} -x_1 & \leq & 0 \\ -x_3 & \leq & 0 \\ 2x_1 + x_2 - x_3 & \leq & 4 \\ 3x_1 - 2x_2 + 2x_3 & \leq & 2 \\ x_1 - 2x_2 & \leq & 5 \\ -x_1 + x_3 & \leq & 8 \end{array}\right.$$

$$\max x_1 + x_2 + x_3$$

On pose $I = \{1, 2, 3\}$ et on a le point $(0, 4, 0)$ qui est réalisable

$$A_I = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 2 & 1 & -1 \end{pmatrix} \quad {A_I}^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 2 & -1 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

Etape 1.

On calcule $c{A_I}^{-1}$ qui vaut $(1, -2, 1)$, on devra donc supprimer la ligne 2 la colonne correspondante dans ${A_I}^{-1}$ est alors $\begin{pmatrix} 0 \\ -1 \\ -1 \end{pmatrix}$. Le produit de cette colonne par l'opposée de chacune des lignes de la matrice A donne $0, -1, 0, -1, -2, 1$. Seul le produit avec la ligne L_6 donne un nombre positif, on fait donc entrer 6 dans I et on obtient : $I = \{1, 3, 6\}$. On obtient

$$A_I = \begin{pmatrix} -1 & 0 & 0 \\ 2 & 1 & -1 \\ -1 & 0 & -1 \end{pmatrix} \quad {A_I}^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 1 & 1 & 1 \\ -1 & 0 & -1 \end{pmatrix}$$

Etape 2.

Le point x est égal à $(0, 12, 8)$

On calcule $c{A_I}^{-1}$ qui vaut $(-1, 1, 2)$, on devra donc supprimer la ligne 1, la colonne correspondante dans ${A_I}^{-1}$ est alors $\begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}$. Le produit de cette colonne par l'opposée de chacune des lignes de la matrice A donne $-1, -1, 0, 6, 3, 0$. Il y a deux valeurs positives pour les lignes 4 et 5, on doit donc calculer :

$$\frac{b_j - L_j x}{v_j}$$

qui vaut 3 pour $j = 4$ et 9.66 pour $j = 5$ on retient donc la ligne 4 et on obtient $I = 3, 4, 6$.

$$A_I = \begin{pmatrix} 2 & 1 & -1 \\ 3 & -2 & 1 \\ -1 & 0 & -1 \end{pmatrix} \quad {A_I}^{-1} = \frac{1}{6} \begin{pmatrix} 2 & 1 & 1 \\ 4 & -1 & 5 \\ 2 & 1 & 7 \end{pmatrix}$$

Etape 3.

On trouve le point $x = (3, 9, 11)$ et on a $A_I^{-1} c = (8, 1, 13)$ qui est positif on a donc atteint l'optimum qui vaut 23 pour ce point x.

5. Problèmes équivalents

On montre que si l'on a un algorithme polynomial qui résout l'un des problèmes suivants, alors il en existe un pour tous les autres. De fait on donne des réductions entre problèmes, réductions qui peuvent être très utiles lorsque l'on veut résoudre l'un d'entre eux. Dans ce qui suit on suppose que tous les nombres qui interviennent dans les matrices, vecteurs ou autres sont des rationnels ; maximum des logarithmes de tous les numérateurs et du dénominateurs détermine la taille du problème.

Dans l'ennoncé qui suit *résoudre* $\max\{cx \mid Ax \leq b\}$ signifie déterminer si ce maximum est fini, infini ou si le problème est non réalisable. Dans le premier cas donner un x qui fournit le maximum, dans le second donner un x tel que $Ax \leq b$ et un $y \geq 0$ tel que $yA = 0$ et $cy > 0$.

THÉORÈME 4.3. *Les problèmes suivants sont polynomialement équivalents*
- *Déterminer s'il existe un x tel que $Ax \leq b$*
- *Déterminer s'il existe un $x \geq 0$ tel que $Ax = b$*
- *Trouver un x, s'il existe, tel que $Ax \leq b$*
- *Trouver un $x \geq 0$, s'il existe, tel que $Ax = b$*
- *Etant donné un nombre γ, déterminer s'il existe un x tel que $Ax \leq b$ et $cx \geq \gamma$*
- *Résoudre* $\max\{cx \mid Ax \leq b\}$.
- *Résoudre* $\max\{cx \mid x \geq 0, \quad Ax \leq b\}$
- *Résoudre* $\max\{cx \mid x \geq 0, \quad Ax = b\}$
- *Résoudre* $\min\{cx \mid \quad Ax \geq b\}$
- *Résoudre* $\min\{cx \mid x \geq 0, \quad Ax \geq b\}$
- *Résoudre* $\min\{cx \mid x \geq 0, \quad Ax = b\}$

DÉMONSTRATION. Laissée en exercice en utilisant le théorèmes donnés plus haut.

6. L'algorithme Primal-Dual

On considère donc le problème primal sous forme normale.

$$(\mathcal{P}) \left\{ \begin{array}{l} \max \ vx \\ Ax = b \\ x \geq 0 \end{array} \right. .$$

Considérons le système dual

$$(\mathcal{D}) \left\{ \begin{array}{l} \min \ yb \\ yA \geq v \end{array} \right. ,$$

et supposons que nous possédons pour $\mathcal{D}$ une solution réalisable u. On peut alors transformer $\mathcal{P}$ en un système $\mathcal{P}'$ ayant les mêmes solutions optimales, car seule la fonction d'objectif a été translatée de la constante ub :

$$(\mathcal{P}') \left\{ \begin{array}{l} \max \ (v - uA)x \\ Ax = b \\ x \geq 0 \end{array} \right. .$$

Ceci pourra advenir si, par exemple, une solution réalisable de $\mathcal{P}$ est telle que

$$x_i > 0 \Rightarrow v_i - yA_i = 0 \ \text{ et } \ v_i - yA_i < 0 \Rightarrow x_i = 0$$

remplissant ainsi les conditions du théorème de complémentation.

On essaye de se mettre dans cette situation et de trouver une solution réalisable de $\mathcal{P}$ telle que $x_i > 0 \Rightarrow v_i - yA_i = 0$.

Pour ce faire on considère le système, dit primal restreint, défini par :

$$(\mathcal{P}R(u))\begin{cases}\min\varphi=\sum y_i\\ Ax+y=b\\ x\geq 0\\ x_i=0 \text{ si } (v-uA)_i<0\\ y>0\end{cases}$$

et réalisable à partir de la solution de base $y = b$ pour laquelle on aura rendu tous les b_i positifs possiblement en multipliant l'équation correspondante par -1.

La solution optimale peut être de deux types.
1. Elle vérifie $\varphi = 0$, signifiant qu'il existe une solution de base x réalisable pour $\mathcal{P}$ et satisfaisant les conditions de complémentarité avec le u de départ.
2. À l'optimum de $\mathcal{P}R(u)$, la fonction φ est toujours strictement positive : il n'existe donc pas de solution réalisable de $\mathcal{P}$ qui soit complémentaire avec u. On modifie alors u en u' pour obtenir un $\mathcal{P}R(u')$ dont le domaine de solutions est un peu plus grand. Considérons le dual de $\mathcal{P}R(u)$:

$$(\mathcal{D}(u))\begin{cases}\max\vartheta b\\ \vartheta A\leq 0, \text{ pour les colonnes où } (v-uA)_i=0\\ \vartheta\leq 1\end{cases}$$

sans contrainte de signe sur ϑ.

On a donc maintenant $\min\varphi=\varphi^\star=\vartheta^\star b=\max\vartheta b>0$. Ces variables duales optimales $\vartheta^\star$ vérifient, elles, les conditions de complémentarité : $x_i>0\Rightarrow\vartheta^\star A_i=0$.
On considère alors le nouveau vecteur générique $u'=u-\lambda\vartheta$: pour toutes les composantes de la base, le vecteur $v-u'A$ reste nul comme précédemment puisque $\vartheta^\star A$ l'est. Pour $\lambda>0$ on a $u'b=ub-\lambda\vartheta^\star b<ub$ et donc on peut diminuer l'écart entre la fonction primale et la fonction duale en choisissant la plus grande valeur de λ compatible avec les contraintes qui assurent que u' reste une fonction réalisable.

Si la valeur de λ n'est pas bornée, on en déduit que le primal $\mathcal{P}$ n'a pas de solution, sinon on remplace u par u' dans $\mathcal{P}'$ et on itère le processus. En raison de la décroissance de la fonction d'objectif duale et en l'absence de dégénérescence, le processus s'arrête en un nombre fini d'étapes.

7. Complexité de la Programmation linéaire

Conceptuellement simple, la méthode évoquée depuis le début du chapitre s'est révélée délicate à implanter dans le détail. Il se trouve de plus que certains problèmes vont forcer le programme à passer par quasiment tous les sommets du simplexe qui, en dimension n, peuvent être au nombre de 2^n.

7.1. Un cas extrême. V. Klee et G. J. Minty ont proposé le problème suivant en dimension n.

$$(\mathcal{K}M)\begin{cases} \max \sum_{j=1}^{n} 10^{n-j}x_j \\ (2\sum_{j=1}^{i-1} 10^{i-j}x_j) + x_i \leq 100^{i-1}, \ \text{pour } (1 \leq i \leq n) \\ x_j \geq 0, \ \text{pour } (1 \leq j \leq n) \end{cases}.$$

Prenons par exemple les cas $n = 3$ et résolvons le système en faisant systématiquement le pivot sur la variable dont le coefficient dans la fonction d'objectif est le plus élevé. Le problème s'énonce donc :

$$(\mathcal{K}M)\begin{cases} \max \ 100x_1 + 10x_2 + x_3 & \\ x_1 & \leq 1 \\ 20x_1 + x_2 & \leq 100 \\ 200x_1 + 20x_2 + x_3 & \leq 10000 \\ x_1, x_2, x_3 & \geq 0 \end{cases}.$$

On commence par mettre ce système en forme normale ajoutant les variables positives x_4, x_5, x_6. On obtient alors mis dans une forme un peu différente

$$(\mathcal{K}M)\begin{cases} & \max \ 100x_1 + 10x_2 + x_3 \\ x_4 = & 1 - x_1 \\ x_5 = & 100 - 20x_1 - x_2 \\ x_6 = & 10000 - 200x_1 - 20x_2 - x_3 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases}.$$

On effectue donc le pivot sur x_1, variable qui est dans la base, ce qui donne d'après la première équation

$$(\mathcal{K}M1)\begin{cases} & \max \ 100 - 100x_4 + 10x_2 + x_3 \\ x_1 = & 1 - x_4 \\ x_5 = & 80 + 20x_4 - x_2 \\ x_6 = & 9800 + 200x_4 - 20x_2 - x_3 \end{cases}.$$

Le pivot maintenant se fait sur x_2 et la deuxième équation :

$$(\mathcal{K}M2)\begin{cases} & \max \ 900 + 100x_4 - 10x_5 + x_3 \\ x_1 = & 1 - x_4 \\ x_2 = & 80 + 20x_4 - x_5 \\ x_6 = & 8200 - 200x_4 + 20x_5 - x_3 \end{cases}.$$

et il faudra encore 5 pivots, que le lecteur est convié à effectuer, avant d'obtenir

$$(\mathcal{K}M7)\begin{cases} & \max \ 10000 - 100x_1 - 10x_2 - x_6 \\ x_4 = & 1 - x_1 \\ x_5 = & 100 - 20x_1 - x_2 \\ x_3 = & 10000 - 200x_1 - 20x_2 - x_6 \end{cases},$$

sur lequel il n'est plus possible d'améliorer la fonction d'objectif et qui donne l'optimum.

Il est un peu plus dur de montrer en toute généralité que le coût est exponentiel. Cependant c'est bien ce qui se passe.

Plus généralement, il a été montré pour toutes les stratégies naturelles de sélection du pivot qu'il existait des systèmes pour lesquels la stratégie obligeait à parcourir tous les sommets du simplexe, ruinant ainsi les espoirs de comportement polynomial.

7.2. Des algorithmes polynomiaux. Depuis de nombreux travaux ont été publiés sur le sujet. Popularisée par la contribution de S. Smale, l'approche stochastique à la complexité moyenne du problème mit en évidence que ce cas extrême est bien peu fréquent et que le nombre moyen de pivots reste polynomial en la dimension du problème.
De plus, Katchian puis Karmarkar ont donné des algorithmes polynomiaux. On a même montré que la programmation linéaire était complète dans la classe des problèmes polynomiaux pour la réduction en espace logarithmique, plaçant ainsi très précisément le problème dans les hiérarchies de complexité.

7.2.1. *La méthode de Katchian.* L'algorithme a été conçu pour résoudre des systèmes d'inéquations. Il peut également s'appliquer à la programmation linéaire soit en récrivant le systèmes d'équations comme système d'inéquations, soit en utilisant une méthode dite de *fonction objectif glissante*, qui est une simple variante de la méthode. En voici une description simplifiée.
On veut donc résoudre le système :

$$Ax \leq b, \qquad (1)$$

où $x \in \mathtt{R}^n$, $b \in (\mathtt{R}^+)^m$ et $A \in \mathtt{R}^{m \times n}$.
On notera a_i le vecteur correspondant à la i-ème ligne de A, et qui est donc un vecteur ligne.

Le principe. On travaille ici avec les réels des ordinateurs, et donc on peut normaliser les entrées du système afin d'en faire des entiers, par simple multiplication par une puissance convenable de 10. On ne se pose pas ici la question de la qualité de l'approximation que l'on va obtenir, et des risques d'erreur consécutifs à un mauvais conditionnement de la matrice. Soit L le nombre total de bits nécessaire à l'écriture du système.
On montre que le système (2) suivant :

$$Ax \leq b + 2^{-L}, \qquad (2)$$

où on ajoute donc la quantité 2^{-L} à chaque membre droit d'inéquation, a une solution si et seulement si (1) en a une. Intuitivement, la perturbation est tellement faible qu'elle n'affecte pas la nature du résultat. De plus, on peut obtenir la solution de (1) en temps polynomial à partir de celle de (2). Enfin, le système (2) est plus facile à traiter car s'il a une solution, il existe une boule de rayon 2^{-2L} dont tous les points sont solution.

La méthode de recherche des solutions s'exprime géométriquement par la construction d'une suite d'ellipsoïdes emboîtés, dans lesquels, à chaque étape, on sélectionne un point dont on espère qu'il satisfait le système. Si tel n'est pas le cas, la k-ième contrainte étant violée par exemple, on restreint l'ellipsoïde parallèlement à cette facette : le lecteur remarquera une certaine analogie avec la méthode du simplexe.
On construit donc une suite de points $z_0, z_1, \ldots, z_p \in \mathtt{R}^n$ et de matrices définies positives

$E_0, E_1, \ldots, E_p \in \mathtt{R}^{n\times n}$, par récurrence, comme suit.
Le premier ellipsoïde est défini par :

$$z_0 = 0, E_0 = 2^{-L}I.$$

À la k-ième étape on procède aux opérations suivantes :
– tester si z_k satisfait le système (2) et alors s'arrêter ;
– sinon, soit i l'indice d'une inégalité violée par z_k, $a_i^T z_k > b[i]$; calculer

$$z_{k+1} = z_k - \frac{1}{n+1}\frac{E_k a_i^T}{\sqrt{a_i E_k a_i^T}},$$

$$E_{k+1} = \frac{n^2}{n^2-1}\left(E_k - \frac{2}{n+1}\frac{(E_k a_i^T)^T(E_k a_i^T)}{a_i E_k a_i^T}\right).$$

On définit ainsi un ellipsoïde de centre z_{k+1}, dont les points satisfont :

$$\mathcal{E}_{k+1} = \{x \mid (x - z_{k+1})\ E_{k+1}^{-1}\ (x - z_{k+1})^T \leq 1\};$$

cet ellipsoïde contient les points de $\mathcal{E}_k$ qui sont encore susceptibles d'être solution. En particulier, si l'ensemble des solutions est non vide, tous les $\mathcal{E}_k$ contiennent la boule de solutions de rayon 2^{-2L} initialement située dans $\mathcal{E}_0$.
On observe que le volume des ellipsoïdes décroît à chaque étape d'un facteur majoré par $e^{-1/(2n+2)}$. En conséquence, si rien n'arrête l'itération ce volume deviendra plus petit que celui de la boule cherchée de rayon 2^{-2L}, à savoir après Cn^2L itérations, pour une constante C.
Si l'algorithme n'a pas permis à ce stade de trouver un centre d'ellipsoïde solution du système, c'est que le système n'en a pas.

Détails de réalisation. Dans la pratique, il faut prendre en compte les erreurs d'arrondi et augmenter significativement la précision des calculs. On ne peut cependant pas aller jusqu'à calculer en nombres rationnels, car alors la taille des fractions risquant de croître exponentiellement, on remettrait en cause le coût polynomial en nombre d'itérations. De fait, il est possible de trouver un compromis et de conserver le caractère polynomial de l'algorithme. Remarquons que cette méthode est presque tout le temps plus coûteuse que celle du simplexe.

7.2.2. *La méthode des fonctions d'objectif glissantes.* On considère de nouveau un problème d'optimisation :

$$(\mathcal{L}P^+)\left\{\begin{array}{l}\max\ vx\\ Ax \leq b\end{array}\right. .$$

On commence par déterminer une solution du système d'inégalités associé de type (2), comme décrit plus haut. On détermine ainsi un point $\overline{z}$.
On ajoute alors l'inégalité $vx \geq v\overline{z}$ aux contraintes et on continue d'itérer le processus, y compris pour améliorer le $\overline{z}$ de départ.
Chaque fois qu'une nouvelle solution x_k qui satisfait $vx_k \geq v\overline{z}$ est trouvée, on pose $\overline{z} := x_k$ et on continue comme plus haut.
On résout ainsi le problème (3) en temps polynomial, mais ce polynôme dépend de la

précision initiale L. De plus, comme déjà mentionné, cette dépendance fait qu' en pratique la méthode de Katchian n'est pas vraiment compétitive.

La méthode de Karmarkar. Après une amélioration de Megiddo, le pas décisif fut effectué par Karmarkar qui améliore notablement la méthode de Katchian en proposant une vision géométrique de type gradient, dont on donne une idée.
La procédure de base est un algorithme rapide pour résoudre les problèmes du type :

$$(\mathcal{K}^+) \begin{cases} \max\ vx, \text{ pour } x \in U \cap S, \\ U = \{x \mid Ax = 0\} \\ S = \{x \mid x \geq 0, ||x|| = 1\}, \end{cases}.$$

où $v, x \in \mathtt{R}^n$, $b \in \mathtt{R}^m$ et $A \in \mathtt{R}^{m \times n}$ et $||x|| = \sum x[i]$ est la somme des coordonnées de x.
On suppose de plus que :
– U est défini par des équations homogènes ;
– la valeur minimale de x est 0 ;
– le centre de la sphère S, le point de coordonnées $1/n$ dans chaque dimension, est solution du système ; – on connaît un paramètre entier q qui permet d'arrêter les itérations dès que $vz_k \leq 2^{-q} vz_0$.
La suite des solutions z_k est telle que la fonction dite de potentiel $f(x) = \sum \log \frac{vx}{x[i]}$, diminue à chaque itération d'une constante (d'où la convergence rapide).
Géométriquement, la transformation projective appliquée à chaque étape de S sur S applique z_k sur le point $\frac{1}{n}e$ qui dans ce nouveau système de coordonnées permet de minimiser la fonction d'objectif.
On obtient ainsi une solution optimale pour le système $\mathcal{K}$ en un temps $O(n^{3.5}L^2)$.

Deuxième partie

Complexité

Préambule

La première partie de ce cours a présenté et étudié un certain nombre de paradigmes de l'algorithmique qui conduisent à des algorithmes, sinon polynomiaux, du moins très efficaces en pratique. L'objet de cette seconde partie est de présenter des situations moins agréables, où l'on peut légitimement supposer qu'il n'existe aucun algorithme fonctionnant en temps polynomial pour résoudre le problème.

Le chapitre 5 étudie la complexité proprement dite. Alors que l'algorithmique tente d'établir des résultats positifs, à savoir « ce problème peut être résolu en temps polynomial », l'objectif avoué de la théorie de la complexité est d'obtenir des résultats négatifs. Hélas, les résultats absolus (comme dans le cas du tri) sont très rares ; on se résout donc à aborder la question de la borne inférieure en *comparant la difficulté de différents problèmes.* C'est tout le sens de la notion de réduction, fondamentale dans le chapitre 5. Cela conduit à une classification des problèmes, selon leur niveau de difficulté, en classes de complexité ; à l'intérieur d'une classe, on essaie de même d'isoler les problèmes les plus difficiles de cette classe, qui seuls sont vraiment représentatifs de la difficulté inhérente à la classe. Ces problèmes sont dits *complets.*

On s'intéressera essentiellement à la classe **NP** des problèmes résolubles en temps polynomial non déterministe. L'objectif principal du chapitre 5 est de présenter un échantillon de problèmes **NP**-complets, ce qui permet d'illustrer largement la technique de réduction.

Comme à l'heure actuelle on ne connaît aucun algorithme efficace pour les problèmes **NP**-complets, et que ces derniers interviennent pourtant fréquemment dans la vie courante, on étudiera dans le chapitre 6 divers angles d'attaque pour les problèmes difficiles. D'une part, nous présenterons une méthode complètement générique pour tous les problèmes de la classe **NP**, le *backtracking*, qui, dans le cas d'un problème de décision, peut conduire à une réponse dans un temps raisonnable si une bonne proportion des choix non-déterministes conduisent à une réponse positive. Le reste du chapitre est consacré à l'étude de problèmes d'optimisation, qui cherchent, en un certain sens, à trouver la meilleure « solution » à un problème **NP**-complet. Une variante du *backtracking*, le branch-and-bound sera présentée dans ce contexte ; nous étudierons ensuite les algorithmes d'approximation, qui permettent, en temps polynomial, de trouver non pas la solution optimale, mais une solution dont la qualité peut être estimée. Tous les problèmes **NP**-complets ne sont pas égaux vis-à-vis de l'approximation, car cette dernière n'est pas préservée par les réductions. Nous montrerons ainsi que certains problèmes s'approchent de façon satisfaisante, tandis que d'autres sont rétifs à toute forme d'approximation.

Le chapitre 7, quant à lui, aborde le problème d'un point de vue différent. Un certain nombre d'algorithmes pour des problèmes **NP**-complets – voire polynomiaux– ont un tout petit ensemble de pire cas ; en probabilisant l'algorithme et l'espace des données, il est souvent possible de donner un sens à cette assertion, et de construire des algorithmes dits *de type Las Vegas*, dont la complexité est polynomiale en moyenne. En poussant ces idées encore plus loin, on peut définir les algorithmes *de type Monte-Carlo*, dont la complexité est toujours polynomiale, mais qui s'autorisent à se tromper avec une probabilité faible.

Rappels et notations

On utilise dans toute cette partie un *modèle de calcul*, par exemple la machine de Turing ou la machine *RAM*. Étant donné une fonction $f : \mathbb{N} \to \mathbb{N}$, on dira qu'un problème appartient à la classe $\mathbf{DTIME}(f)$ s'il existe une machine déterministe et une constante c qui sur toute entrée X de longueur n, résout le problème en au plus $cf(n)$ pas de calcul. On insiste sur le fait que l'on parle bien de la *longueur* des données, et que cette dernière dépend en principe de leur représentation, qui devrait toujours être partie intégrante de la description du problème. En particulier, quand « un entier n » figure parmi les données du problème, sa longueur est $\max(0, \lceil \log n \rceil) + 1$, – en supposant une représentation binaire classique – et à ce titre un algorithme dont la complexité est proportionnelle à n n'est pas polynomial. En revanche, si « une matrice $n \times m$ à coefficients $\{0, 1\}$ »figure parmi les données, la taille de l'entrée est nm, et un algorithme polynomial peut avoir une complexité proportionnelle à nm tout en étant polynomial. Certains problèmes peuvent même spécifier des données d'entrée sous une forme particulière ; ainsi la donnée d'une matrice M de taille $n \times m$ sous forme creuse est la donnée de tous les couples (i, j) tels que le coefficient M_{ij} soit non nul. Si l'on suppose que les coefficients sont entiers et bornés par A, la taille de l'entrée est alors $O(w \log A)$, où w est le nombre de coefficients non nuls de M, tandis que si les données du problème indiquent juste une matrice M, cette taille est $O(mn \log A)$. La notion d'algorithme polynomial peut alors différer suivant le format de l'entrée, par exemple si $w = O(\log(mn))$.

On note $\mathbf{P} = \cup_{k \geq 0} \mathbf{DTIME}(n \mapsto n^k)$. On définit de la même façon la classe $\mathbf{NTIME}(f)$ et la classe **NP** dans le cadre des machines non-déterministes. On a bien entendu l'inclusion $\mathbf{DTIME}(f) \subset \mathbf{NTIME}(f)$, et donc $\mathbf{P} \subset \mathbf{NP}$. Savoir si cette inclusion est stricte ou non constitue le fameux problème « $\mathbf{P} = \mathbf{NP}$? ».

Enfin, on peut remarquer – et cette caractérisation est beaucoup plus proche de l'intuition, donc plus maniable en pratique – que la classe **NP** peut se définir comme la classe des problèmes dont toute réponse « oui » admet un certificat vérifiable en temps polynomial (l'ensemble des choix de la machine non-déterministe).

Il est important de noter que ces classes de complexité s'intéressent à des *problèmes de décision*, par opposition aux *problèmes d'optimisation ou problèmes de calcul* tels ceux qui ont été étudiés dans la première partie. Par exemple, PRIMALITÉ, qui décide la primalité d'un nombre entier est un problème de décision – cette question appelle une réponse oui ou non –, dont on peut montrer qu'il est[1] dans **NP**.[2] FACTORISATION, en revanche, n'est pas un problème de décision ; il n'y a donc pas de sens à se demander si FACTORISATION est dans **P** ou **NP**.[3]

Il existe toutefois un lien entre problème d'optimisation et problème de décision. Si le problème d'optimisation est à valeurs entières, et consiste à rendre un bénéfice maximal , on peut construire un problème de décision en rajoutant une donnée entière K et en demandant de décider s'il existe une solution au problème de décision dont le bénéfice est au moins K. Dans ce contexte, notons que dans le cas où ce problème de décision peut être résolu en temps $O(P(n, \log K))$, et où l'on peut obtenir une borne M pour la meilleure solution, le problème d'optimisation peut être résolu en temps $O(f(n, \log M) \log M)$ par une simple dichotomie, ce qui montre qu'en un certain sens, si le problème admet une borne polynomiale, il y a équivalence entre existence d'un algorithme polynomial pour le problème de décision et le problème d'optimisation.

NP-complétude

On dit qu'un problème P se réduit polynomialement à un problème Q si l'on peut construire une fonction τ de l'espace $\mathcal{P}$ des entrées de P dans l'espace $\mathcal{Q}$ des entrées de Q, telle que P accepte x si et seulement si Q accepte $\tau(x)$, et si τ est calculable par une machine de Turing (ou une machine RAM) déterministe fonctionnant en temps polynomial.[4] La fonction τ est alors une réduction du problème P au problème Q. Une conséquence importante de cette définition est le fait que s'il existe un algorithme polynomial pour Q, alors il existe un algorithme polynomial pour P, obtenu en décidant Q sur $\tau(x)$.

Un problème est dit **NP**-dur si tout problème de **NP** se réduit polynomialement à lui. Un problème est dit **NP**-complet s'il est **NP**-dur et dans **NP**. Noter que si l'*on sait résoudre ne serait ce qu'un seul problème* **NP**-*dur en temps polynomial, alors* $\mathbf{P} = \mathbf{NP}$. En effet, si le problème D **NP**-dur admet un algorithme en temps polynomial, et si Q est un problème de **NP**, il existe une réduction τ de Q à D. Pour une entrée x de τ, décider D sur $\tau(x)$ peut alors se faire en temps polynomial, et revient à décider Q.

Le théorème de Cook affirme que le problème suivant, SAT, est **NP**-complet.

SAT
Données : un ensemble de variables $\{x_1, \ldots, x_n\}$ et un ensemble de clauses de la forme $y_{i,1} \vee y_{i,2} \vee y_{i,k}$, où chacun des $y_{i,j}$ est soit l'un des x_l, soit l'un des $\overline{x}_l$.

Résultat : la conjonction des clauses données peut-elle être satisfaite par une assignation de valeurs de vérité V/F aux variables ?

La preuve de Cook permet en fait de montrer que le problème 3-SAT, cas où les clauses contiennent chacune trois littéraux au plus, est en fait déjà **NP**-complet. On montrera ce

[1]Noter qu'on ne sait toujours pas si PRIMALITÉ est dans **P**.
[2]Ce n'est pas trivial. Pour le prouver, noter que p est premier ssi. $(\mathbb{Z}/p\mathbb{Z})^*$ est cyclique d'ordre $p-1$.
[3]Les problèmes de calcul ont leurs propres classes de complexité **FP** et **FNP**.
[4]Sur la définition de la notion de réduction, voir les compléments au chapitre 5.

fait indépendamment. En revanche, le problème 2-SAT est dans **P**. Cette remarque illustre un fait important, à ne pas perdre de vue : *ce n'est pas parce qu'un problème est* **NP**-*complet que toutes ses instances sont difficiles*; en particulier, relâcher une hypothèse ou fixer un paramètre peut fréquemment conduire à un problème de **P**.

Pour aller plus loin dans l'étude de la complexité, on consultera avec grand profit le livre de C. Papadimitriou [**16**]. L'ouvrage [**7**] répertorie de façon systématique un grand nombre de problèmes **NP**-complets connus en 1979. C'est un passage obligé quand on s'intéresse à un problème dont on pense qu'il est **NP**-complet.

CHAPITRE 5

La réduction entre problèmes

Introduction

Ce chapitre décrit plusieurs exemples de réductions, et présente donc par là-même un certain nombre de problèmes **NP**-complets classiques, de caractère assez varié pour pouvoir être eux-mêmes utilisés dans un vaste ensemble de preuves de **NP**-complétude.

1. 3-SAT

Nous allons nous intéresser dans cette partie au problème 3-SAT, cas particulier du problème SAT . Si 3-SAT est **NP**-complet, il est alors clair que SAT, qui le contient, l'est aussi. En revanche, comme souligné dans le préambule, le fait que SAT soit **NP**-complet ne prouve en rien que l'instance particulière 3-SAT le soit. Cela requiert une preuve que cette instance est aussi difficile que le cas général.

3-SAT.
Données : un ensemble de variables $\{x_1, \ldots, x_n\}$ et un ensemble de clauses de la forme $(y_{i,1} \vee y_{i,2} \vee y_{i,3})$, où $y_{i,j}$ est égal soit à l'un des x_i, soit à l'un des $\overline{x}_i$.

Résultat : la conjonction des clauses données peut-elle être satisfaite par un assignement de valeurs de vérité V/F aux variables ?

THÉORÈME 5.1. *Le problème* 3-SAT *est* **NP***-complet.*

DÉMONSTRATION. Nous donnons volontairement une preuve détaillée de **NP**-complétude, en vérifiant soigneusement l'appartenance à **NP** et le caractère polynomial de la réduction. Le lecteur est invité à détailler les affirmations équivalentes des preuves suivantes.

Notons tout d'abord que 3-SAT est bien dans **NP**. En effet, la donnée d'une affectation de valeurs de vérité aux variables constitue bien un certificat vérifiable en temps polynomial.

On cherche à réduire SAT à 3-SAT. On se donne donc une conjonction de clauses à un nombre quelconque de littéraux, et il faut trouver une formule booléenne, conjonction de clauses à (au plus) 3 littéraux, qui lui soit équivalente, calculable en temps polynomial à partir de la formule initiale.

Partons d'une clause $x_1 \vee \cdots \vee x_{2l}$. On peut supposer que l'on ne rencontre pas dans la même clause à la fois une variable et son complémentaire, sans quoi on peut ignorer cette clause (qui est automatiquement satisfaite). Toutes les variables sont donc indépendantes. On introduit une nouvelle variable y_i par paire de variables (x_{2i-1}, x_{2i}) et l'on va réécrire

$x_1 \vee \cdots \vee x_l = (y_1 = x_1 \vee x_2) \wedge (y_2 = x_3 \vee x_4) \wedge \cdots \wedge (y_l = x_{2l-1} \vee x_{2l}) \wedge (y_1 \vee \ldots y_l)$. Il reste à exprimer la formule $(y_1 = x_1 \vee x_2)$ comme une conjonction de clauses à 3 littéraux. Pour ce faire, on observe que $(y_1 = x_1 \vee x_2)$ est équivalente à la conjonction

- $y_1 = 1 \Rightarrow x_1 = 1 \vee x_2 = 1$;
- $y_1 = 0 \Rightarrow x_1 = 0 \wedge x_2 = 0$,

soit encore $(y_1 \Rightarrow (x_1 \vee x_2)) \wedge (\overline{y}_1 \Rightarrow (\overline{x}_1 \wedge \overline{x}_2)) = (\overline{y}_1 \vee x_1 \vee x_2) \wedge (y_1 \vee (\overline{x}_1 \wedge \overline{x}_2)) = (\overline{y}_1 \vee x_1 \vee x_2) \wedge (y_1 \vee \overline{x}_1) \wedge (y_1 \vee \overline{x}_2)$.

Ce faisant, on a donc

- réduit la longueur de la plus longue clause à $\max(3, l)$,
- multiplié par $3/2$ le nombre de variables,
- multiplié la longueur totale de la formule au plus par une constante c.

Si l'on itère le processus, après $\log_2 l + 1$ étapes, on aura donc uniquement des clauses à 3 littéraux, un nombre de variables de l'ordre de $O(l^{1+\log_2(3/2)})$ et une formule de longueur totale $O(l^{1+\log_2(c)})$. On a donc réduit une clause quelconque à une conjonction de clauses à 3 littéraux. Le temps de calcul se réduit à écrire les clauses, dont la longueur est polynomiale. Il s'ensuit que l'ensemble du processus de réduction se réalise donc bien en temps polynomial. On a donc montré, par réduction de SAT, que 3-SAT est **NP**-complet. □

On peut remarquer que 3 est la limite de la méthode : la transformation de la formule $y_1 = (x_1 \vee x_2)$ produit automatiquement une clause à trois littéraux. De fait, comme mentionné dans le préambule, on peut montrer que 2-SAT est dans **P**.

Dans le même esprit, on peut étudier le problème NAESAT (*not all equal* SAT) qui peut paraître anecdotique, mais qui nous sera utilie dans la suite pour démontrer que d'autres problèmes sont **NP**-complet.

NAESAT

Données : un ensemble de variables $\{x_1, \ldots, x_n\}$ et un ensemble de clauses $y_{i1} \vee y_{i2} \cdots \vee y_{ik_i}$, où pour tout (i, j), y_{ij} est soit un des x_i, soit un des $\overline{x}_i$.

Réponse : existe-t-il une affectation de valeurs de vérité aux x_i de sorte que pour chaque i il existe j et k tels que $y_{ij} = 1$ et $y_{ik} = 0$?

THÉORÈME 5.2. NAESAT *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire SAT à NAESAT. Considérons un ensemble de clauses pour SAT sur les variables $\{x_1, \ldots, x_n\}$. On ajoute à ces variables une nouvelle variable z et l'on forme des clauses pour NAESAT en transformant chaque clause de l'instance de SAT donnée $y_{i1} \vee \cdots \vee y_{ik_i}$ en $y_{i1} \vee \cdots \vee y_{ik_i} \vee z$.

Si l'instance donnée de SAT est satisfaisable, la même affectation de valeurs aux variables, tout en affectant à z la valeur 0, fournit une affectation valide pour NAESAT.

Inversement, supposons que l'instance construite de NAESAT est satisfaisable. Si la valeur de vérité de z dans l'affectation correspondante est 0, les valeurs des variables x_i correspondent à une affectation valide pour l'instance de SAT d'origine (au moins un des y_{ij} ou z vaut 1 par clause de NAESAT pour une affectation valide, et ce n'est pas z). Si au contraire z vaut 1, on change les valeurs de toutes les variables dans l'affectation. L'affectation reste valide pour NAESAT, car au moins une variable par clause valait 0 dans l'affectation initiale, et vaut donc maintenant 1, tandis que z vaut 0. On a donc construit une affectation dans laquelle z vaut 0, et en vertu du cas précédent l'instance de SAT initiale est satisfaisable.

On a donc bien prouvé l'équivalence entre satisfaisabilité de l'instance initiale de SAT et de l'instance correspondante de NAESAT. Le caractère polynomial de la réduction est évident, comme l'appartenance de NAESAT à **NP** (la donnée d'une affectation valide est un certificat vérifiable en temps polynomial). Par suite, on a bien prouvé que NAESAT est **NP**-complet. Noter qu'en effectuant la même réduction sur 3-SAT, on prouve que NAE4SAT est **NP**-complet. □

COROLLAIRE 5.1. NAE3SAT *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire NAE4SAT à NAE3SAT. Soit $x \vee y \vee z \vee t$ une clause à 4 littéraux. On introduit une nouvelle variable u, et on forme les deux clauses $C_1 = x \vee y \vee \overline{u}$ et $C_2 = z \vee t \vee u$.

Supposons qu'il existe une affectation de valeurs de vérité qui rende simultanément satisfaisables ces deux clauses. Alors x, y, z, t ne peuvent pas toutes avoir la même valeur dans l'affectation, sans quoi soit $\overline{u}$, soit u a aussi cette valeur et la clause correspondante n'est pas satisfaisable au sens NAESAT.

Inversement, si la clause initiale est satisfaisable, l'une des variables de C_1 a une valeur différente d'une des variables de C_2 dans l'affectation correspondante (sans quoi toutes les variables ont la même valeur dans l'affectation). Il suffit alors de choisir la valeur de u en conséquence pour satisfaire simultanément C_1 et C_2 au sens NAESAT.

Là encore, la réduction est polynomiale et NAE3SAT est dans **NP** de façon triviale. □

Instances particulières et variantes. Pour chacun des problèmes décrits dans ce chapitre, on s'efforcera de joindre un « bestiaire » discutant la **NP**-complétude de ses variantes, et d'éventuelles instances (cas particuliers) polynomiales ou **NP**-complètes. Dans le cas de SAT, cela donne :

Instances et variantes polynomiales : 2-SAT, HORN-SAT (toutes les clauses doivent être des clauses de Horn, i.e. de la forme $(x_1 \wedge x_2 \wedge \cdots \wedge x_n \Rightarrow y)$, ou encore $(\overline{x}_1 \vee \overline{x}_2 \vee \cdots \vee \overline{x}_n \vee y))$. Le cas de SAT où chaque variable apparaît au plus deux fois sur l'ensemble des clauses est également polynomial ; on peut même autoriser chaque variable à apparaître 3 fois dans le cas de 3-SAT.

Instances et variantes **NP**-*complètes* : k-SAT, $k \neq 2$; MAX2SAT ou plus généralement MAXSAT (étant donné l, existe-t-il une affectation de valeurs de vérité qui permet de satisfaire au moins l clauses de la formule ? – c'est bien entendu la version « de décision » de ce problème qui est **NP**-complète) ; ONE-IN-THREE SAT (une seule variable à 1 par clause) ; CIRCUIT-SAT (la formule n'est pas donnée sous forme normale conjonctive, mais sous la forme d'un circuit à n entrées, les variables, et une sortie ; des portes logiques $\vee, \wedge, \bar{\ }$ décrivent le calcul de la sortie en fonction de l'entrée).

2. Clique

L'étape suivante est d'établir la **NP**-complétude de problèmes de graphes. Le problème CLIQUE consiste à décider, étant donné un graphe $G = (V, E)$ non orienté et un entier k, s'il existe un sous-graphe complet de G à k sommets.

Clique.
Données : un graphe $G = (V, E)$ et un entier k.

Résultat : existe-t-il $V' \subset V$, $\text{card}(V') = k$, et $u, v \in V' \ \Rightarrow \ (u, v) \in E$.

Théorème 5.3. CLIQUE *est un problème* **NP**-*complet.*

Démonstration. Notons que CLIQUE est bien dans **NP**, car la donnée de V' est un certificat aisément vérifiable en temps polynomial.

On va réduire le problème 3-SAT à CLIQUE, c'est-à-dire étant donné une formule booléenne F du type 3-SAT, construire en temps polynomial un graphe G, de sorte que l'existence d'une clique dans G soit équivalente à l'existence d'une affectation valide de valeurs de vérité dans la formule initiale.

On écrit F sous la forme $\bigwedge_{1 \leq j \leq J}(x_{1j} \vee x_{2j} \vee x_{3j})$. L'idée de la construction du graphe G est la suivante : ses sommets sont les variables x_{ij}, et la clique sera formée en choisissant dans chaque formule un des x_{ij} dont la valeur est 1. On va donc chercher dans G une clique de taille J, et construire l'affectation de valeurs de vérité en mettant toutes les variables de la clique à 1, les autres de façon quelconque (noter qu'il faut toutefois tenir compte du fait que x_{ij} et x_{kl} peuvent être liés pour établir l'affectation finale).

Reste à placer les arêtes, de sorte qu'on ait bien équivalence entre sommets d'une J-clique de G et variables à 1 d'une affectation valide de F.

La première contrainte est une contrainte de cohérence. Si deux des variables x_{ij} et x_{kl} sont la négation l'une de l'autre – rappelons que les variables x_{ij} représentent soit un y_l, soit un $\overline{y}_l$ –, aucune affectation valide des x_{ij} ne pourra avoir $x_{ij} = x_{kl} = 1$. Il faut donc empêcher qu'une clique ne contienne à la fois ces deux variables, ce qui se fait en interdisant la présence d'une arête entre x_{ij} et x_{kl} si $x_{ij} = \overline{x}_{kl}$.

La seconde contrainte est une contrainte de complétude. La J-clique doit contenir une variable par clause exactement. Il faut donc empêcher que deux variables de la même clause ne figurent dans une même clique, et donc ne pas placer d'arête entre x_{ij} et x_{lj}.

On place en revanche des arêtes entre tous les couples de sommets qui ne sont pas éliminés par ces deux contraintes, obtenant ainsi le graphe J.

Montrons que la satisfaisabilité de la formule F est équivalente à l'existence d'une J-clique dans G. Si la formule F est satisfaisable, il existe un sous-ensemble de sommets $(x_{i(j),j})_{1 \leq j \leq J}$ tel que $x_{i(j),j}$ est vrai. Tous les j sont distincts, et les variables $x_{i(j),j}$ étant à 1, donc n'étant pas négation deux à deux l'une de l'autre, le sous-graphe correspondant est complet, et est donc une J-clique.

Inversement, s'il existe une J-clique, on construit une affectation de valeurs de vérité de la façon suivante :

– $x_{ij} = 1$ si x_{ij} est un des sommets de la J-clique ou s'il existe i, j tels que $x_{kl} = x_{ij}$ et x_{ij} sommet de la j-clique ;
– $x_{kl} = 0$ s'il existe i, j tels que $x_{kl} = \overline{x}_{ij}$ et x_{ij} sommet de la j-clique ;
– x_{kl} quelconque autrement, en tenant compte des relations entre eux de même que ci-dessus.

Il est facile de voir que, grâce aux deux contraintes introduites, cette affectation, d'une part est cohérente (on n'attribue pas à une même variable deux valeurs différentes), mais conduit bien à une affectation valide (toutes les clauses ont exactement une de leurs variables dans la clique).

On a donc bien prouvé que 3-SAT se réduit à CLIQUE. La réduction est manifestement polynomiale, et cette remarque conclut la preuve. □

Deux problèmes classiques sont liés à CLIQUE :

ENSEMBLE INDÉPENDANT
Données : un graphe $G = (V, E)$ et un entier k.

Résultat : existe-t-il $V_0 \subset V$ de cardinal k tel que pour tout $(i, j) \in V_0 \times V_0$, $(i, j) \notin E$?

RECOUVREMENT DE SOMMETS
Données : un graphe $G = (V, E)$ et un entier k.

Résultat : existe-t-il $V_0 \subset V$ de cardinal $\leq k$ tel que pour tout $(i, j) \in E$, $i \in V_0$ ou $j \in V_0$?

COROLLAIRE 5.2. *Le problème* ENSEMBLE INDÉPENDANT *est* **NP***-complet.*

DÉMONSTRATION. Il suffit de constater que

$$\text{ENSEMBLE INDÉPENDANT}(V, E, k) = \text{CLIQUE}(V, V \times V - E, k).$$

□

COROLLAIRE 5.3. *Le problème* RECOUVREMENT DE SOMMETS *est* **NP***-complet.*

DÉMONSTRATION. ENSEMBLE INDÉPENDANT(V, E, k) = RECOUVREMENT DE SOMMETS$(V, E, |V| - k)$, car un recouvrement s'obtient simplement comme le complémentaire d'un ensemble indépendant. Il existe donc un recouvrement de taille $\leq k$ si et seulement s'il existe un ensemble indépendant de taille $\geq k$, ssi il existe un ensemble indépendant de taille k. □

On étudie maintenant un problème assez différent, mais restant dans le même esprit.
COUPURE MAXIMALE :

Données : un graphe $G = (V, E)$ et un entier k.

Réponse : existe-t-il une partition $V = V_1 \cup V_2$ telle que le nombre d'arêtes (u, v) de E avec $u \in V_1$ et $v \in V_2$, ou $u \in V_2$ et $v \in V_1$ soit au moins k ?

THÉORÈME 5.4. COUPURE MAXIMALE *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire NAE3SAT à COUPURE MAXIMALE. Supposons donnée une instance de NAE3SAT, dans laquelle sans perte de généralité nous supposerons qu'une clause ne contient pas simultanément une variable et son complémentaire ; quitte à remplacer la clause $u \vee v$ par $(u \vee v \vee w) \wedge (u \vee v \vee \overline{w})$, on peut en outre supposer que chaque clause contient exactement 3 littéraux (si une clause contient un seul littéral, la formule n'est pas satisfaisable). Enfin si l'on a deux clauses $(u \vee v \vee w)$ et $(u \vee v \vee z)$, on peut, en introduisant deux variables t_1 et t_2 et en procédant comme pour réduire NAE4SAT à NAE3SAT, réécrire ces deux clauses comme $(u \vee t_1) \wedge (v \vee w \wedge \overline{t}_1)$ et $(u \vee t_2) \wedge (v \vee z \vee \overline{t}_2)$. On peut donc supposer que deux clauses données ont au plus une variable en commun.

Notons $\{x_1, \ldots, x_n\}$ les variables de la formule.

On va construire un graphe $G = (V, E)$ construit de la façon suivante : on a $V = \{x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n\}$. Le but est maintenant qu'une coupure de grande taille corresponde à une affectation de valeurs de vérité qui satisfasse la formule au sens NAE3SAT. Pour ce faire, on va bien entendu choisir comme variable à 0 les variables appartenant à V_1, et comme variable à 1 les variables appartenant à V_2 (où V_1, V_2 est la coupure obtenue par COUPURE MAXIMALE).

On construit alors l'ensemble d'arêtes E ; on place une arête entre chaque couple de sommets (u, v) tels que u et v apparaissent dans la même clause, et une arête entre les sommets u et $\overline{u}$ pour toute variable u. Les réductions opérées dans le premier paragraphe de la preuve permettent de voir que les triangles correspondant à des clauses ont toutes leurs arêtes distinctes. Si l'on note n le nombre de variables et m le nombre de clauses, il est alors facile de voir que le nombre d'arêtes dans une coupure correspondant à une affectation NAE3SAT-valide est $2m+n$ (l'arête entre u et $\overline{u}$ pour chaque variable u, et deux des arêtes du triangle uvw pour une clause $u \vee v \vee w$; c'est pour cela que l'on doit veiller à ce que les triangles correspondant aux clauses soient distincts, sans quoi on compterait ici certaines arêtes deux fois). Inversement, toute coupure a au plus cette valeur, car une coupure ne peut inclure que deux arêtes par triangle associé à une clause ; il s'ensuit qu'une coupure de valeur $2m + n$ fournit immédiatement une affectation NAESAT valide (le $+n$ garantit que u et $\overline{u}$ figurent bien chacun dans un sous-ensemble de la partition, et garantit donc la cohérence de l'affectation de valeurs).

On a donc montré que résoudre NAE3SAT sur le problème initial revenait à résoudre COUPURE MAXIMALE sur $(G, 2m + n)$. Il s'ensuit donc, la réduction étant polynomiale et COUPURE MAXIMALE étant manifestement dans **NP**, que COUPURE MAXIMALE est **NP**-complet. □

Instances particulières et variantes. *Instances et variantes polynomiales* : RECOUVREMENT D'ARÊTES (recouvrement d'un graphe par des arêtes) ; CLIQUE est polynomial si l'on se restreint à des graphes de degré borné par un d donné, ou à des graphes planaires. ENSEMBLE INDÉPENDANT est polynomial pour les graphes bipartis ou les graphes de degré ≤ 2.

Instances et variantes **NP**-*complètes* : ENSEMBLE INDÉPENDANT reste **NP**-complet si le graphe est supposé de degré donné $d \geq 4$ (la réduction donnée ci-dessus construit une instance de degré 4).

3. Circuit hamiltonien

Le problème décrit dans cette section sert souvent à prouver la **NP**-complétude de propriétés liées aux chemins dans les graphes.

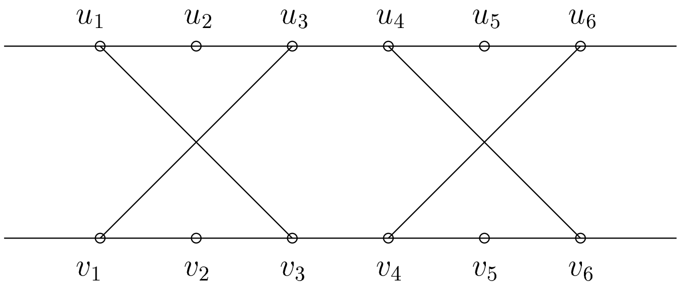
CIRCUIT HAMILTONIEN
Données : un graphe $G = (V, E)$.

Résultat : existe-t-il un *circuit hamiltonien*, c'est-à-dire un chemin de G passant une fois et une seule par chacun des sommets et revenant à son point de départ ?

THÉORÈME 5.5. CIRCUIT HAMILTONIEN *est* **NP**-*complet.*

DÉMONSTRATION. On va démontrer ce fait en réduisant RECOUVREMENT DE SOMMETS à CIRCUIT HAMILTONIEN. La technique consiste à construire, partant d'une instance de RECOUVREMENT DE SOMMETS, un graphe dans lequel chaque arête initiale sera remplacée par un « motif » admettant exactement deux chemins hamiltoniens, c'est-à-dire des chemins visitant une fois et une seule chaque sommet. L'un de ces deux chemins correspondra au cas où le sommet correspondant appartient à la couverture, l'autre au cas où il n'appartient pas. Il reste ensuite à préciser comment recoller ces différents motifs pour qu'un circuit hamiltonien global corresponde exactement à une réunion de chemins hamiltoniens à travers chaque motif, et pour assurer qu'on obtient bien le bon résultat au problème RECOUVREMENT DE SOMMETS initial en résolvant CIRCUIT HAMILTONIEN dans ce graphe.

Notons $(G = (V, E), k)$ l'instance de RECOUVREMENT DE SOMMETS étudiée. Le motif que nous allons utiliser est le suivant :



Pour obtenir un parcours de ce motif traversant une fois et une seule chaque sommet, seules deux solutions sont possibles : soit un passage en deux fois, une fois $u_1u_2u_3u_4u_5u_6$ puis ultérieurement $v_1v_2v_3v_4v_5v_6$ (dans un sens ou dans l'autre) ; ou alors un passage en une seule fois $u_1u_2u_3v_1v_2v_3v_4v_5v_6u_4u_5u_6$ (ou le même en inversant les u et les v). À chaque arête (u, v) du graphe de départ, on associe un motif de ce type, en faisant correspondre le sommet u au côté (u) et le sommet v au côté (v). On raccorde ensuite entre eux bout-à-bout tous les côtés de tous les motifs correspondant à un même sommet ; on forme donc une chaîne associée à un sommet donné, avec encore deux « sorties » libres. On raccorde alors chacune de ces deux sorties à k nouveaux sommets $s_1, \ldots, s_k$. On note le graphe ainsi construit H dans la suite.

Supposons maintenant donné un recouvrement du graphe initial de taille k, dont les sommets sont $\{g_1, \ldots, g_k\}$. On peut alors construire un circuit hamiltonien de H de la façon suivante :

- partir de s_1 ;
- parcourir la chaîne g_1 de la façon suivante. Quand on traverse une arête (g_1, h), si h est aussi dans la couverture, on traverse simplement le côté g_1, sinon, on parcourt les deux côtés simultanément.
- une fois la chaîne g_1 finie, on revient en s_2 et on repart par g_2, ainsi de suite.

Il est clair que tous les sommets s_k sont atteints une fois et une seule.

Considérons un sommet h du motif correspondant à une arête (u, v). On peut toujours supposer que u est dans la couverture, disons $u = g_1$. Il s'ensuit que si h est du côté de u, h sera atteint une fois au moins. On voit en vertu du second item qu'il ne sera plus atteint dans la suite. Si h est du côté de v, et que v n'est pas dans la couverture, h est parcouru lors du parcours de u. Si $v = g_i$, h est parcouru lors du parcours de la chaîne correspondant à g_i et à ce moment-là seulement. On a donc bien un circuit hamiltonien.

Réciproquement, supposons que l'on dispose d'un circuit hamiltonien de H. La construction de notre motif impose que venant d'un sommet s_i, on traverse entièrement une chaîne u puis l'on passe à un autre des sommets s_j. On traverse ainsi k chaînes ; les k sommets correspondants forment alors une couverture. En effet, si (u, v) est une arête, le motif correspondant est parcouru par le chemin hamiltonien ; or il ne peut l'être que lors du parcours d'une boucle correspondant à une des deux extrémités de l'arête.

Enfin, la réduction est trivialement polynomiale. CIRCUIT HAMILTONIEN est donc bien **NP**-complet. □

Comme dans la partie précédente, on peut déduire aussitôt la **NP**-complétude de nombreuses variantes :

VOYAGEUR DE COMMERCE

Données : un couple (n, M), où M est une matrice $n \times n$ de réels et un réel k.

Réponses : existe-t-il une permutation π de $[1, n]$ telle que $\sum_{1 \leq i \leq n-1} M_{\pi(i)\pi(i+1)} + M_{\pi(n)\pi(1)} \leq k$?

On peut voir ce problème comme l'établissement de la tournée d'un voyageur de commerce devant visiter n villes, dont les distances sont données par la matrice M, de façon à faire moins de k kilomètres.

COROLLAIRE 5.4. VOYAGEUR DE COMMERCE *est* **NP***-complet.*

DÉMONSTRATION. On réduit CIRCUIT HAMILTONIEN à VOYAGEUR DE COMMERCE. Pour ce faire, étant donné $G = (V, E)$, on pose $V = \{x_1, \ldots, x_n\}$. On considère alors la matrice $n \times n$ telle que

$$M_{ij} = \begin{cases} 1 & \text{si } (i, j) \in E \\ 2 & \text{sinon} \end{cases}$$

Montrons alors que

$$\text{VOYAGEUR DE COMMERCE}(n, M, n) = \text{CIRCUIT HAMILTONIEN}(V, E).$$

S'il existe un circuit hamiltonien dans G, on peut en effet construire la permutation π comme décrivant l'ordre dans lequel les sommets de G sont parcourus. La construction de M implique alors que $\sum_{1 \leq i \leq n-1} M_{\pi(i)\pi(i+1)} + M_{\pi(n)\pi(1)} = n$. Inversement, étant donné une permutation avec cette propriété, le fait que les $n-1$ termes de la somme soient au moins égaux à 1 implique qu'ils sont en fait tous égaux à 1, et que donc les arêtes $(x_\pi(i), x_{\pi(i+1)})$ existent dans le graphe G ; on a donc bien construit un circuit hamiltonien. □

CIRCUIT LE PLUS LONG
Données : un graphe G valué, et un entier r.

Résultat Existe-t-il un circuit de G ne passant pas deux fois par le même sommet dont la longueur est $\geq r$?

COROLLAIRE 5.5. CIRCUIT LE PLUS LONG *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire CIRCUIT HAMILTONIEN à CIRCUIT LE PLUS LONG. Pour ce faire, étant donné un graphe G pour circuit hamiltonien, on associe à chaque arête le poids 1. La recherche d'un circuit hamiltonien est alors trivialement identique à la recherche d'un circuit de longueur $\geq n$ dans le graphe. □

Instances particulières et variantes. *Instances et variantes polynomiales* : CIRCUIT HAMILTONIEN est polynomial dans un graphe de degré au plus 2.

Instances et variantes **NP**-*complètes* : CHEMIN HAMILTONIEN, cas où l'on ne demande pas à l'itinéraire de revenir au point de départ, est **NP**-complet. CIRCUIT HAMILTONIEN et CHEMIN HAMILTONIEN restent **NP**-complets même si le graphe est supposé planaire et de degré 3, ou si le graphe est supposé biparti. CHEMIN HAMILTONIEN reste **NP**-complet même si les deux extrémités sont données. CHEMIN LE PLUS COURT est polynomial, voir la première partie.

VOYAGEUR DE COMMERCE reste **NP**-complet même si les distances sont choisies dans $\{1, 2\}$ (cf. la preuve donnée ci-dessus). Reste **NP**-complet si la distance vérifie l'inégalité triangulaire, est la distance L^1, la distance L^∞, la distance L^2 arrondie supérieurement (si l'on n'arrondit pas, on doit manipuler des réels, ce qui est toujours malaisé avec notre modèle de calcul). Le cas où l'on demande seulement que l'arête la plus longue soit plus courte que B reste **NP**-complet, même si toutes les arêtes sont de poids 1 ou 2.

Le problème FACTEUR(K), où l'on demande que chaque arête soit traversée, avec un coût total $\leq K$ est aussi **NP**-complet, même si toutes les arêtes sont de coût 1.

4. 3-COLORABILITÉ

Le problème de la colorabilité consiste à déterminer, étant donné un graphe, le nombre de couleurs à utiliser pour pouvoir colorier les sommets du graphe, de façon à ce que deux sommets adjacents soient toujours coloriés de deux couleurs différentes. Le plus petit tel nombre est appelé le nombre chromatique du graphe. Dans le cas des graphes planaires (par exemple les cartes géographiques, si l'on suppose les pays connexes et simplement connexes), on peut montrer – c'est un résultat très profond – que 4 couleurs suffisent toujours.

3-COLORABILITÉ

Données : un graphe $G = (V, E)$.

Réponse : existe-t-il une application $\chi : V \to \{1, 2, 3\}$ telle que $\forall (u, v) \in E, \chi(u) \neq \chi(v)$?

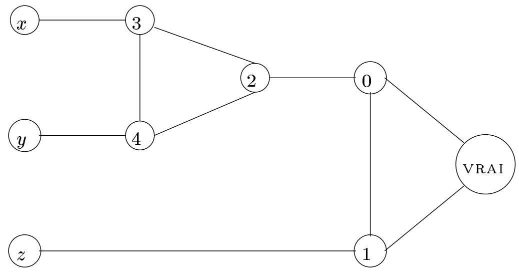
THÉORÈME 5.6. 3-COLORABILITÉ *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire 3-SAT à 3-COLORABILITÉ. On se donne donc une conjonction de clauses à 3 littéraux, et il nous faut à partir de là construire un graphe. Comme dans les autres réductions de 3-SAT, il faut parvenir à traduire deux contraintes : une variable peut prendre la valeur 0 ou 1 d'une part, et les règles d'évaluation d'une clause d'autre part.

On construit trois sommets VRAI, FAUX, NSP. Ces trois sommets sont reliés deux à deux en triangle, de sorte qu'ils sont tous trois de couleurs différentes. On appelera les couleurs correspondantes VRAI, FAUX, NSP.

On va commencer par associer un sommet à chaque variable et au complémentaire de chaque variable. Pour assurer qu'une variable prenne la valeur 0 ou 1, pour chaque variable x_i on construit un triangle dont les sommets sont x_i, $\overline{x}_i$ et NSP. Cela impose que soit $\chi(x_i) =$ VRAI et $\chi(\overline{x}_i) =$ FAUX, soit $\chi(x_i) =$ FAUX et $\chi(\overline{x}_i) =$ VRAI.

Il nous reste donc à encoder les règles d'évaluation d'une clause. Pour ce faire, on introduit le sous-graphe suivant, qui correspond à une clause $x \vee y \vee z$:



Il est facile de voir que si ce motif – où les trois sommets distingués et les triangles construits sur les variables sont implicites – est 3-coloriable, alors les sommets 0 et 1 sont FAUX et NSP. Si 1 est FAUX, comme un sommet correspondant à une variable doit être VRAI ou FAUX, on a $\chi(z)$=VRAI. Si 0 est FAUX, alors 2 ne peut pas être FAUX, donc 3 ou 4 l'est, et la variable correspondante est coloriée VRAI.

Réciproquement, si l'une des variables est vraie, on peut facilement construire une 3-coloration du motif.

Considérons alors le graphe formé des trois sommets distingués, des triangles formés sur les variables, et des motifs donnés. Si ce graphe est 3-coloriable, alors en particulier tout sous-graphe est coloriable. Les triangles de variables sont en particulier coloriables.

À partir d'une 3-coloration du graphe, on construit une affectation de valeurs de vérité en mettant à 1 toutes les variables coloriées par VRAI. Cette affectation est cohérente – une variable et son complémentaire ont bien une valeur opposée – et au moins une variable par clause est à 1, en vertu des propriétés du motif ci-dessus.

Inversement, étant donné une affectation de valeurs de vérité, il est aisé de déduire un 3-coloration du graphe.

L'existence d'une 3-coloration du graphe est donc équivalente à la satisfaisabilité de la formule initiale. La réduction est manifestement polynomiale ; on a donc bien prouvé que 3-SAT se réduisait à 3-COLORABILITÉ ; ce dernier est donc bien **NP**-complet. □

Noter que l'on peut obtenir une preuve légèrement plus simple en utilisant une réduction de NAE3SAT ; on laisse ce fait en exercice au lecteur.

Instances particulières et variantes. *Instances et variantes polynomiales* : 2-COLORABILITÉ, 4-COLORABILITÉ PLANAIRE (toujours vrai). k-COLORABILITÉ pour les graphes de degré ≤ 3.

Instances et variantes **NP***-complètes* : 3-COLORABILITÉ reste **NP**-complet dans le cas planaire, même si l'on suppose le degré ≤ 4. k-COLORABILITÉ est **NP**-complet pour tout $k \geq 3$.

5. Somme de sous-ensemble

Le problème dit SOMME DE SOUS-ENSEMBLE est le suivant :

SOMME DE SOUS-ENSEMBLE

Données : un ensemble fini d'entiers E et un but $t \in \mathbb{N}$.

Réponse : existe-t-il $E' \subset E$ tel que $\sum_{x \in E'} x = t$?

THÉORÈME 5.7. SOMME DE SOUS-ENSEMBLE *est* **NP***-complet.*

DÉMONSTRATION. Le fait que SOMME DE SOUS-ENSEMBLE est dans **NP** provient du fait que la donnée de E' constitue un certificat pouvant être vérifié en temps polynomial.

On va maintenant réduire une version de RECOUVREMENT DE SOMMETS dans laquelle on cherche à construire une couverture de taille exactement k à SOMME DE SOUS-ENSEMBLE. Il est facile de voir que cette version de RECOUVREMENT DE SOMMETS est essentiellement équivalente à la version classique, car il suffit de décider pour chaque $l \leq l_0$ s'il existe une couverture de taille l pour savoir s'il existe une couverture de taille au plus l_0.

On se donne un graphe $G = (V, E)$ dans lequel on souhaite construire un recouvrement de sommets de taille k. On numérote les sommets et arêtes ; soit $B = (b_{ij})$ la matrice d'incidence sommets-arêtes, c'est-à-dire que $b_{ij} = 1$ si l'arête i est incidente au sommet j, $b_{ij} = 0$ sinon.

On va construire un ensemble F d'entiers, tel que chaque sommet corresponde à un entier de F. On veut pouvoir lire sur la somme des éléments d'un sous-ensemble les arêtes dont un des sommets au moins appartient au sous-ensemble, de façon à ce qu'on ait bien une couverture de sommets si et seulement si c'est le cas pour toutes les arêtes.

Cela peut se faire en choisissant une base B à fixer ultérieurement, et en faisant correspondre au sommet s_j l'entier $\sum b_{ij} B^i$. On veut aussi pouvoir compter le nombre de sommets présents dans la couverture – qui doit être égal à k ; pour ce faire, il suffit d'ajouter un terme B^n, si les arêtes sont numérotées de 0 à $n-1$.

Il subsiste une difficulté : pour qu'un sous-ensemble de somme $t = \sum_{i=0}^{n} a_i B^i$ corresponde à une couverture de sommets, le choix juste fait impose que – noter que tous les a_i sauf a_n valent au plus 2, car une arête est incidente à deux sommets exactement –

- $a_n = k$;
- $a_i \geq 1$.

Les spécifications du problème ne permettent pas de prendre en compte le second item. On peut contourner le problème en ajoutant à F les entiers B^i correspondant à l'arête i. On peut alors choisir comme but

$$t = kB^n + \sum_{i=0}^{n-1} 2B^i. \tag{3}$$

À ce stade, on va imposer que la base B soit au moins égale à 4 ; comme dans toute somme d'éléments de F il y a au plus trois termes B^i pour $i < n$, cela entraîne qu'aucune retenue ne peut se produire dans l'addition (on permet toujours au coefficient de B^n d'excéder $B-1$), et donc que l'on peut lire sur l'équation (3) le nombre d'occurences du terme B^i.

Pour tout sous-ensemble F' de somme t, on a donc :

- il y a k termes correspondant à des sommets dans F' (cf. le terme en B^n,
- pour chaque arête i, F' contient au moins un terme correspondant à une des deux extrémités de i ; en effet, t contenant 2 termes B^i, il peut contenir soit les deux extrémités de l'arête, soit l'entier correspondant à l'arête et celui correspondant à l'une des deux extrémités.

Inversement, étant donné une couverture de sommets, on construit l'ensemble F' en prenant les sommets de la couverture, et les arêtes dont un seul des deux sommets est dans la couverture. On a donc bien réduit RECOUVREMENT DE SOMMETS à SOMME DE SOUS-ENSEMBLE.

Il est facile de voir que la réduction s'effectue en temps polynomial, et donc que l'on a bien prouvé le théorème. □

On peut en déduire la **NP**-complétude du classique problème du sac-à-dos :

SAC-À-DOS

Données : un ensemble de poids $a_1, \ldots, a_n$, un ensemble de valeurs $v_1, \ldots, v_n$, un poids limite A, et un entier V.

Réponse : existe-t-il une suite $\epsilon_i \in \{0,1\}$ telle que $\sum \epsilon_i a_i \leq A$ et $\sum \epsilon_i v_i \geq V$?

COROLLAIRE 5.6. SAC-À-DOS *est* **NP**-*complet.*

DÉMONSTRATION. On se donne une instance de SOMME DE SOUS-ENSEMBLE, soit $E = \{e_1, \ldots, e_n\}, t$. Il suffit alors de prendre $v_i = a_i = e_i$, et $V = A = t$ dans SAC-À-DOS. □

On obtient aussi de la même façon la **NP**-complétude du problème de la partition :
PARTITION
Données : un ensemble A fini d'entiers.

Réponse : existe-t-il $A' \subset A$ tel que $\sum_{x\in A'} x = \sum_{x\notin A'} x$?

THÉORÈME 5.8. PARTITION *est* **NP**-*complet.*

DÉMONSTRATION. On va réduire SOMME DE SOUS-ENSEMBLE à PARTITION. Soit (E,t) une instance de SOMME DE SOUS-ENSEMBLE. Posons $S = \sum_{x\in E} x$. Quitte à changer t en $S-t$ (ce qui revient à changer l'ensemble obtenu en son complémentaire), on peut supposer que $2t \leq S$.

L'idée naturelle consisterait à ajouter l'élément $u = S - 2t$ à E ; le résultat de partition serait alors deux sous-ensembles (A' et sont complémentaire) de somme $S-t$; l'un des deux contient l'élément $S - 2t$, donc en enlevant ce dernier, on trouve un ensemble de termes de E de somme t. Malheureusement, cette technique échoue si $S - 2t$ est déjà dans E.

Au lieu de cela, on prend le nombre $X = 2S$ et $X' = S+2t$, et on applique PARTITION à $E' = E \cup \{X, X'\}$. Il existe une partition de E' si et seulement s'il existe un sous-ensemble de E de somme t. En effet, s'il existe une partition de E', il existe deux sous-ensembles complémentaires de somme $2S + t$. Chacun des deux sous-ensembles doit contenir soit X, soit X', car sinon sa somme ne peut excéder S ; donc un des deux ensembles contient X et non X', et on obtient en enlevant X un sous-ensemble F de E de taille t. Réciproquement, étant donné un tel F, $(F \cup \{X\}, E - F \cup \{X'\})$ constitue une partition de E'.

On a donc bien réduit SOMME DE SOUS-ENSEMBLE à PARTITION. Reste à justifier que la réduction est bien polynomiale. L'essentiel de la réduction est le calcul de X et X', qui est bien polynomial en la taille des entrées (l'addition de k nombres de n bits se fait en temps $O(k \log n)$). □

RANGEMENT OPTIMAL
Données : n objets de poids $s_i \in \mathbb{Z}$, une capacité B et un entier k.

Réponse : Est-il possible de ranger les objets dans k boîtes de capacité B ?

COROLLAIRE 5.7. RANGEMENT OPTIMAL *est* **NP**-*complet.*

DÉMONSTRATION. Il est bien clair que RANGEMENT OPTIMAL est dans **NP**, car la donnée de la liste des éléments allant dans chaque boîte constitue un certificat de taille polynomiale.

Il suffit alors de remarquer que le cas $k = 2$, $B = (\sum_{x\in E} x)/2$ est exactement PARTITION. □

Instances particulières et variantes. *Instances et variantes polynomiales* : SOMME DE SOUS-ENSEMBLE, PARTITION et SAC-À-DOS sont polynomiales si les éléments de l'ensemble (les deux premiers cas) et les valeurs des objets (SAC-À-DOS) sont bornés. Voir à ce sujet le paragraphe 2 du chapitre 6.

Instances et variantes **NP**-*complètes* : PRODUIT DE SOUS-ENSEMBLE.

6. Programmation entière

Le problème de la programmation entière est l'exact analogue du problème étudié au chapitre 7.2.2, à ceci près que les variables sont restreintes à être entières. C'est un problème extrêmement général, et on n'a que l'embarras du choix du problème à lui réduire pour prouver qu'il est **NP**-dur. La difficulté, pour une fois, revient à prouver qu'il est dans **NP**.

Données : une matrice $k \times n$ notée A, un vecteur de dimension k noté b.

Réponse : existe-t-il des entiers $x_1, \ldots, x_n$ positifs tels que $Ax = b$?

Remarquons que nous décrivons ici une version de la programmation linéaire en nombres entiers n'incluant même pas de fonctionnelle à optimiser. La « seule » difficulté du problème semble être la contrainte de positivité des variables.

Théorème 5.9. Programmation entière *est* **NP**-*complet.*

Démonstration. On va réduire 3-sat à programmation entière. On introduit deux variables entières y_i et w_i par variable booléenne x_i, liées par $y_i + w_i = 1$. L'une des deux variables représente la valeur de vérité de x_i, l'autre celle de $\overline{x}_i$. Les contraintes de satisfaisabilité s'écrivent alors toutes sous la forme $F_j(y_i, w_i) \geq 1$, où F_j est la somme des 3 variables correspondant aux différents termes d'une clause. On peut transformer ces inégalités en égalités, de la même façon que pour la programmation linéaire classique (en réécrivant chaque inégalité comme $F_j(y_i, w_i) - z_j = 1$). Ce processus de réduction est manifestement polynomial, donc programmation entière est **NP**-dur.

Montrons maintenant que programmation entière est dans **NP**. La difficulté *a priori* vient du fait qu'on ne dispose d'aucune borne sur la taille des x_i, et qu'on ne peut donc conclure que la donnée de ces derniers constitue un certificat de taille polynomiale. On doit donc prouver le lemme suivant.

Lemme 5.1. *Si* $A = (a_{ij}$ *est une matrice* $m \times n$ *et* $b = (b_j)$ *un vecteur de taille* n*, tels que* $\max(|a_{ij}|, |b_j|) \leq A$*, et si le système* $Ax = b$ *a une solution à coefficients entiers positifs, alors il a une solution* (x_i) *à coefficients entiers positifs telle que* $\max x_i \leq 2n^2(ma)^{2m+1}$.

Il existe donc un certificat de taille $m \log_2 (2n^2(ma)^{2m+1}) = O(m^2 \log_2(mna))$, qui est bien polynomial en la taille de la matrice de données $(O(mn \log_2(a)))$.

Démonstration. La présente preuve du lemme est due à Papadimitriou.

Posons $M = (ma)^m$, et soit x une solution en entiers positifs telle que $\max x_i$ est minimal. Si cette quantité est au plus M, le résultat est prouvé. Dans le cas contraire, supposons que les k premières composantes de x sont plus grandes que M. Soit $v_1, \ldots, v_k$ les k premières colonnes de A.

Supposons d'abord qu'il existe des α_i entiers dans $[0, M]$ tels que $\sum \alpha_i v_i = 0$, on peut alors remplacer x par $(x_1 - \alpha_1, \ldots, x_k - \alpha_k, x_{k+1}, \ldots, x_n)$, ce qui contredit la minimalité de x.

Dans le cas contraire, les v_k sont linéairements indépendants. Soit V une matrice $k \times k$ inversible extraite des v_k ; on peut résoudre le système ${}^tVu = 1$; en complétant le vecteur u obtenu par des 0, et en transposant, on obtient un vecteur ligne h de taille m tel que $h \cdot (v_k) = 1$. Les formules de Cramer montrent que les coordonnées de h sont des rationnels, qui, réduits au même dénominateur (le déterminant de V) ont des numérateurs au plus égaux à $(ka)^k \leq M$. En multipliant tous les termes par le dénominateur commun, on a donc un vecteur h d'entiers de $[0, M]$ tel que $h \cdot (v_k) \geq 1$.

On multiplie alors $Ax = b$ à gauche par h ; il vient

$$\sum_{j=1}^{k} h \cdot (v_j) \cdot x_j = hb - \sum_{j=k+1}^{n} h \cdot (v_j) \cdot x_j,$$

et donc

$$\sum_{j=1}^{k} x_j \leq \sum_{j=1}^{k} h \cdot (v_j) \cdot x_j \leq mMa + n^2 maM^2 \leq 2n^2(ma)^{2m+1},$$

ce qui conclut la preuve du lemme et du théorème. □

Instances particulières et variantes. *Instances et variantes polynomiales* : La version la plus générale de la programmation entière (avec inégalités et fonctionnelle à optimiser) admet un algorithme polynomial à n (dimension) fixée.

Instances et variantes **NP***-complètes* : La variante où les variables sont dans $\{0, 1\}$ est **NP**-complète (cf. la preuve donnée ci-dessus), même si tous les coefficients des données sont contraints à être dans $\{0, 1\}$. Le cas où l'on admet des inégalités, les variantes de type optimisation sont bien sûr aussi **NP**-complètes, car plus générales que la variante étudiée ici. Les trois problèmes du paragraphe précédent peuvent être vues comme des instances de PROGRAMMATION ENTIÈRE.

Compléments

On indique ici de façon très rudimentaire quelques prolongements de la théorie de la complexité. L'objectif est de montrer l'extrême richesse de cette théorie et le grand nombre de problèmes ouverts subsistant.

6.1. Classes de complexité. Outre **P** et **NP**, il est possible de définir d'autres classes de complexité, comme les classes **EXP** et **NEXP**. On peut également définir des classes de complexité en termes d'espace, telles la classe **L** des problèmes pouvant être résolus en *espace* $\log n$ sur une entrée de taille n par une machine de Turing déterministe, ainsi que son équivalent non-déterministe **NL** ; on peut aussi définir ainsi la classe **PSPACE**. Il n'est pas très difficile de montrer la suite d'inclusions suivante :

$$\mathbf{L} \subset \mathbf{NL} \subset \mathbf{P} \subset \mathbf{NP} \subset \mathbf{PSPACE}. \tag{4}$$

On peut envisager, pour chacune de ces classes, de définir des problèmes complets comme on l'a fait pour **NP**. Toutefois, pour les classes de « faible » complexité, la réduction en temps polynomial n'a pas de sens, car la réduction contient alors déjà toute la capacité de calcul de la classe, voire plus. On demande alors usuellement que les réductions se fassent en *espace logarithmique* (on ne sait d'ailleurs pas si les deux notions de réduction diffèrent ; ce n'est en particulier pas le cas si $\mathbf{L} = \mathbf{P}$... le lecteur vérifiera que toutes les réductions présentées dans ce chapitre sont bien en espace logarithmique, de sorte que les problèmes restent **NP**-complet sous cette notion de réduction).

Toutes ces classes admettent alors des problèmes complets. Par exemple, le problème de la programmation linéaire classique est **P**-complet, comme le problème d'évaluer la valeur de sortie d'un circuit booléen, les entrées étant données. Dans la même lignée, on peut montrer que le problème QSAT est **PSPACE**-complet :

Données : Une formule $Q_1x_1Q_2x_2Q_3x_3\dots Q_nx_n\varphi(x_1,\dots,x_n)$, où les Q_i sont des quantificateurs $\forall$ ou $\exists$, et φ une formule booléenne.

Résultat : La formule est-elle vraie ou fausse ?

Le fait que ce problème soit dans **PSPACE** est facile à voir : on peut en espace polynomial essayer toutes les affectations possibles des variables, en « effaçant » à chaque étape les calculs précédents. En revanche, la présence de quantificateurs universels fait qu'on voit mal comment offrir un certificat compact de la satisfaisabilité, puisque si n quantificateurs universels sont présents il faut a priori tester 2^n valeurs de vérité au moins (noter que la question $\mathbf{NP} = \mathbf{PSPACE}$ est ouverte, et qu'il est donc possible qu'il existe un certificat de taille polynomiale pour QSAT, mais cela serait extrêmement surprenant).

On peut montrer également que pour divers jeux, le problème de décider si une position donnée entraîne automatiquement la victoire d'un des deux joueurs est **PSPACE**-complet (voir [**16**] pour le cas du jeu de go un peu modifié).

6.2. Oracles. La notion de réduction que nous avons étudiée, dite *réduction de Karp*, ou *polynomial-time many-one reduction*, n'est pas la seule à exister. On s'impose en particulier de n'effectuer qu'un seul appel au problème auquel on veut réduire, qui doit avoir lieu à l'extrême fin ; le résultat ne doit en outre pas être modifié.

On peut considérer une notion de réduction un peu différente, dite *réduction de Turing* ou *réduction de Cook*. Dans cette réduction, on suppose que l'on dispose d'un « oracle » résolvant le problème auquel on cherche à réduire, en temps polynomial. On s'autorise alors à faire autant d'appels que nécessaire – un nombre polynomial dans le cas de **P**, bien sûr – à cet oracle lors de la réduction. La classe de complexité obtenue en résolvant un problème dans une classe de complexité **C** moyennant des appels à un oracle O est notée $\mathbf{C}^{\mathrm{O}}$.

6.3. Inclusions strictes. On peut enfin s'interroger sur l'existence d'inclusions strictes dans nos classes de complexité, et en particulier dans l'identité (4). La situation est, *grosso modo*, la suivante ; si l'on compare deux classes de complexité de même nature (déterministe en temps, déterministe en espace, non déterministe en temps, non déterministe en espace), la hiérarchie est bien comprise :

$$\mathbf{TIME}(f(n)) \neq \mathbf{TIME}(f(n)\log^2 f(n)), \quad \mathbf{SPACE}(f(n)) \neq \mathbf{SPACE}(f(n)\log n), \tag{5}$$

pour toute fonction f « raisonnable » croissant plus vite que n. Des résultats analogues existent dans le cas non-déterministe.

En revanche, dès lors que l'on mélange deux types de classes de complexité, très peu de choses sont connues ; citons toutefois l'inclusion $\mathbf{NSPACE}(f(n)) \subset \mathbf{SPACE}(f(n)^2)$, qui implique que $\mathbf{NPSPACE} = \mathbf{PSPACE}$, dès que $f(n)$ croît plus vite que $\log n$. Notons en particulier que la suite d'inclusions (4) est stricte prise dans son ensemble, puisque (5) montre en particulier que $\mathbf{L} \neq \mathbf{PSPACE}$. On ne sait à l'heure actuelle prouver le caractère strict d'aucune des 4 inclusions la composant...

6.4. Vérifieurs. Il est bon, pour conclure ce chapitre, de mentionner quelques beaux et difficiles résultats obtenus récemment par la théorie de la complexité dans le domaine des preuves interactives. Dans ce contexte, on substitue au modèle de calcul habituel un modèle basé sur la notion de *vérifieur*, machine qui étant donné un problème et une entrée x, y, décide si y constitue bien un certificat du fait que x est une instance à réponse positive du problème de départ. Ainsi, on peut voir la classe **NP** comme la classe des problèmes admettant un vérifieur en temps polynomial.

La classe $\mathbf{PCP}(f(n),\ g(n))$ est définie comme la classe des machines polynomiales probabilistes fonctionnant sur une entrée x de taille n en utilisant $O(f(n))$ bits aléatoires, accédant à $O(g(n))$ bits de la preuve fournie ; on exige en outre que ces machines valident toutes les preuves correctes, et rejettent les preuves incorrectes avec probabilité $1/2$ au moins. Ce que dit la remarque de la fin du paragraphe suivant, c'est que $\mathbf{NP} = \mathbf{PCP}(0, \mathrm{poly}(n))$. Le résultat suivant est beaucoup plus profond et illustre la puissance des méthodes probabilistes (que nous étudierons plus en détail au chapitre 3) :

$$\mathbf{NP} = \mathbf{PCP}(\log(n),\ 1).$$

En d'autres termes, en utilisant $O(\log(n))$ bits aléatoires, et en accédant à *un nombre de bits borné* de la preuve, on peut vérifier le certificat de toute instance d'un problème de **NP**, avec une probabilité arbitrairement faible de se tromper (en itérant k fois la vérification de façon indépendante, on ramène la probabilité d'accepter une instance invalide à 2^{-k}). S'il est encore utile d'illustrer la puissance du modèle **PCP**, mentionnons qu'on peut montrer que $\mathbf{PCP}(\mathrm{poly}(n),\ \mathrm{poly}(n)) = \mathbf{NEXP}$...

L'autre classe dont nous voulons parler ici a été introduite en relation avec la cryptographie, et est une classe de preuve interactive, où le prouveur et le vérifieur échangent des messages par lesquels le prouveur cherche à persuader le vérifieur que la solution qu'il détient est bien valide.

Dans le cas de la classe **IP**, le prouveur P a une puissance de calcul exponentielle, le vérifieur V une puissance de calcul polynomiale probabiliste. Chacun des deux protagonistes envoie à son tour un message, en commençant par le prouveur, qui peut dépendre des messages précédents. Finalement, le vérifieur décide d'accepter ou non. Un problème est dans **IP** s'il admet un algorithme de preuve interactive de ce type, où si x est une instance valide, alors le couple (P, V) accepte avec probabilité au moins $1 - 1/2^{|x|}$, et si x n'est pas une instance valide, alors pour tout prouveur P' de puissance de calcul exponentielle, (P', V) refuse avec probabilité au moins $1 - 1/2^{|x|}$. Il faut rapprocher cette définition de la situation cryptographique où une carte à puce à puissance limitée souhaiterait sous-traiter des calculs à un serveur très puissant, mais auquel elle ne fait pas confiance. Elle pose alors des questions pour tenter de vérifier le résultat. La classe **IP** est, dans ce contexte, la classe des problèmes ou la carte peut être convaincue avec une bonne probabilité que le serveur n'est pas en train de lui mentir. L'extrême puissance de ces preuves interactives est illustrée par le théorème de Shamir :

$$\mathbf{IP} = \mathbf{PSPACE},$$

c'est-à-dire que tout calcul pouvant s'effectuer en espace polynomial peut être effectué dans un tel contexte.

CHAPITRE 6

Traiter la NP-complétude

Introduction

Dans le chapitre précédent, nous avons étudié de nombreux problèmes et prouvé leur caractère **NP**-complet. Cela implique en particulier que si $\mathbf{P} \neq \mathbf{NP}$, ces problèmes sont certainement difficiles à résoudre. Ces problèmes consistaitent parfois (MAXSAT, SAC-À-DOS, ...) en une réécriture d'un problème d'optimisation.

Dans ce chapitre, nous allons nous intéresser à ces problèmes d'optimisation. Dans le cas où le problème de décision associé est **NP**-complet, on a le choix entre deux stratégies. Soit les instances que l'on souhaite étudier sont petites et/ou peu nombreuses, et l'on peut se contenter de programmer très soigneusement le meilleur algorithme – non polynomial – connu pour ce problème ; soit on est amené à étudier le problème de façon plus extensive, et l'on peut alors être amené à relâcher la contrainte selon laquelle on souhaite trouver *la meilleure* solution. Dans ce cas, on peut parfois – mais hélas pas toujours – concevoir des algorithmes dits d'approximation, qui fournissent une solution dont la valeur diffère de la solution optimale d'un coefficient constant. Nous verrons également que tous les problèmes **NP**-complets ne peuvent pas être approchés de la même façon, et que sous réserve que $\mathbf{P} \neq \mathbf{NP}$, on peut obtenir des résultats de non-approximabilité assez précis.

Face à un problème d'optimisation **NP**-complet que l'on doit résoudre, diverses stratégies doivent être étudiées. La première question qui doit venir à l'esprit est : ai-je besoin de résoudre ce problème en toute généralité ? En effet, comme on l'a déjà vu, toutes les instances d'un problème **NP**-complet ne sont pas forcément **NP**-complètes ; parfois même, très peu d'entre elles le sont. Ainsi, beaucoup de problèmes **NP**-complets sur les graphes admettent des algorithmes polynomiaux simples dans le cas où le graphe est sans cycle (arbre). De même, le passage de 3-SAT à 2-SAT transforme le problème de **NP**-complet à polynomial. Enfin, parfois, admettre que l'algorithme puisse être exponentiel en certains des paramètres (souvent des paramètres entiers) peut permettre d'obtenir un algorithme de type *pseudo-polynomial*, voir le paragraphe 2.

La seconde question à se poser est : ai-je vraiment besoin d'une réponse exacte, de la meilleure réponse ? Dans le cas contraire, il faut penser aux algorithmes d'approximation (voir le paragraphe 3), ou encore aux algorithmes probabilistes de type Monte-Carlo, qui fournissent souvent de bons résultats. On peut mentionner à part les algorithmes probabilistes de type Las Vegas, dont le résultat est garanti, mais dont le temps d'exécution n'est polynomial qu'avec une bonne probabilité. Voir le chapitre 7 pour les méthodes probabilistes.

Une fois ces possibilités écartées (il faut la meilleure solution, dans un contexte assez général), on peut alors jeter son dévolu sur une méthode « générique » de résolution des problèmes NP-complets, le *backtracking* (ou son analogue pour les problèmes d'optimisation, *branch-and-bound*).

1. Backtracking et branch-and-bound

La méthode de backtracking exploite l'existence d'un arbre de calcul de profondeur au plus polynomiale résolvant le problème, et s'efforce de parcourir cet arbre de calcul jusqu'à avoir trouvé une solution. Dans la mesure où l'arbre de calcul a généralement un nombre exponentiel de sommets, on peut s'attendre à effectuer un nombre exponentiel d'états, et cela risque fort d'être le cas, surtout si une seule branche de l'arbre conduit à une solution. Dans le cas, toutefois, ou de nombreuses branches de l'arbre conduisent à une solution, on peut s'attendre à obtenir un résultat de façon relativement efficace.

Il convient de bien garder à l'esprit que, comme toute stratégie de type générique, le backtracking ne doit être utilisé qu'en *dernier recours*. Dans le cas de la non-primalité des entiers, le *backtracking* correspond par exemple à l'algorithme naïf consistant à essayer de diviser l'entier donné par tous les nombres plus petits que lui, algorithme qui n'est efficace que pour des entiers de très petite taille.

Toutefois, pour de nombreux problèmes, aucune autre stratégie n'existe. Le backtracking est donc en pratique d'une réelle utilité, d'autant qu'il est conceptuellement simple et facilement implanté. Nous allons tenter d'en formaliser un peu le cadre et de donner les principales heuristiques permettant d'en améliorer l'efficacité.

1.1. Le backtracking. Le *backtracking* s'intéresse aux problèmes de décision, c'est-à-dire aux problèmes où il faut répondre oui/non (avec preuve, par exemple la primalité) ou aux problèmes dont on doit trouver *une* solution. L'exposé général du backtracking, comme toujours en algorithmique (penser aux algorithmes gloutons, à la programmation dynamique[1]) repose sur la capacité à savoir décomposer le problème courant en un certain nombre de problèmes de même nature, mais plus petit, en faisant à chaque étape un choix.

Un pseudo-code « générique » pour le backtracking peut s'écrire de la façon suivante :

- Entrée : problème S ;
- Faire $\mathcal{A} := \{S\}$;
- Tant que $\mathcal{A} \neq \emptyset$ faire
    - Choisir $T \in \mathcal{A}$;
    - Décomposer T en les sous-problèmes $\{T_0, \ldots, T_r\}$;
    - Pour chaque i
        - Appliquer un test heuristique à T_i ; si le test répond qu'il y a une solution, la renvoyer, si le test ne répond rien, faire $\mathcal{A} := \mathcal{A} \cup \{T_i\}$.
- Renvoyer « Pas de solution ».

[1]On peut voir le backtracking comme une programmation dynamique ratée, parce que les sous-problèmes se reproduisent peu fréquemment, et donc que leur nombre est trop grand (non polynomial).

Expliquons d'abord ce que l'on entend par TEST HEURISTIQUE. Il faut voir chacun des sous-problèmes T_i obtenus ci-dessus comme un sous-arbre de l'arbre de calcul. La décomposition en sous-problèmes plus petits revient à remplacer le sous-arbre par ses propres sous-arbres. L'idée d'appliquer un test heuristique est d'essayer d'exploiter la structure du problème pour prédire à l'avance quels sont ceux de ces sous-arbres qui ne donneront de toute façon pas de solution, afin d'éviter d'avoir à les explorer. On suppose donc que l'on dispose d'un test *efficace* qui répond oui s'il y a une solution (et donne alors cette dernière), non s'il n'y a pas de solution (on peut alors couper cette branche), et ne répond pas s'il ne sait pas décider. Typiquement, la situation la plus simple est celle d'un problème où l'on ne sait dire quelque chose sur le problème qu'une fois celui-ci réduit à une instance de taille 1. Le test est alors : si l'instance est de taille 1, résoudre le problème ; sinon ne pas répondre.

Une fois ceci précisé, examinons de plus près notre métaalgorithme. Trois des points sont mentionnés en petites capitales ; c'est ceux auxquels le programmeur doit réfléchir, en fonction de son problème, pour obtenir une solution de la façon la plus rapide possible. Signalons que si l'on recherche toutes les solutions, toutes les stratégies (sauf pour le test heuristique) se valent, car il faudra de toute façon parcourir l'arbre dans son intégralité.

On a donc deux choix à faire. D'une part, comment choisir celui des problèmes que l'on va à nouveau décomposer (en particulier, comment parcourir l'arbre ? En largeur, en profondeur, de façon hybride) ? Enfin, une fois le problème choisi, comment va-t-on le décomposer (il y a généralement plusieurs choix, voir les exemples...) ?

En ce qui concerne le parcours de l'arbre, pour les instances de taille importante, le parcours en largeur est à éviter, car il impose trop de stockage (tous les renseignements associés aux différentes branches de l'arbre doivent être stockés simultanément). On préfèrera donc en général parcourir en profondeur, quitte, une fois ramené à une petite instance, à passer à un parcours en largeur. Dans certaines circonstances, cette stratégie peut être modifiée, par exemple, s'il est facile de calculer les informations associées à une branche connaissant celles associées à sa voisine (rare).

Pour ce qui est du choix du problème à décomposer (ainsi que de la façon de le décomposer), la seule règle générale consiste à choisir le sous-arbre qui a des chances d'être le plus petit ; cela ne veut pas dire que le problème a moins de chances d'avoir des solutions, mais plutôt que s'il n'en a pas, on s'en apercevra plus vite et on n'aura pas perdu beaucoup de temps (c'est une stratégie plutôt pessimiste). Il est toutefois bon d'essayer différentes stratégies sur de petites instances pour approcher mieux le comportement (et la structure de l'arbre de calcul) de chaque problème particulier.

1.2. Exemples.

1.2.1. 3-SAT. On rappelle que 3-SAT est le problème (NP-complet) consistant en la donnée d'une formule, conjonction de clauses qui sont elles-même disjonction de trois variables ou de leur négation, et dont la solution est une affectation de valeurs de vérité aux variables $x_1, \ldots, x_n$ telle que la formule donnée soit vraie.

Montrons comment faire les choix discutés au-dessus. D'une part, quel test heuristique choisir ? Prenons le plus simple : un sous-problème vide est faux (on verra pourquoi dans la décomposition), un sous-problème à une seule clause conduit à une solution, et un sous-problème de la forme $z \wedge \overline{z}$ est faux. Comment réduire ? En choisissant un des sous-problèmes couranst, et en le décomposant en deux sous-problèmes selon qu'une des variables intervenant a pour valeur `vrai` ou `faux`. Quel problème et quelle variable choisir ? En vertu du principe général énoncé ci-dessus, les clauses les plus restrictives étant les clauses ayant le moins de littéraux, le problème choisi sera un de ceux contenant la plus petite clause, et la variable une de celles intervenant dans une de ces clauses.

1.2.2. CIRCUIT HAMILTONIEN. Pour illustrer le *backtracking*, donnons une méthode s'appliquant au cas du circuit hamiltonien.

Un sous-problème est défini par un chemin allant de a à b, et passant par une suite de noeuds $T \subset V - \{a, b\}$. L'objectif consiste alors à trouver un autre chemin de a à b ne passant que par les noeuds de $V - T$. On initialise le processus en choisissant $a = b$ un noeud arbitraire et $T = \emptyset$.

La décomposition en sous-problèmes est simple : on choisit au hasard un sommet de $V - T$, disons c, relié par une arête à b. Le nouveau sous-problème est alors défini par a, c et $T \cup \{b\}$.

Un exemple de test possible pour décider si une branche est condamnée par avance peut être de regarder si le graphe subsistant est connexe, ou s'il admet un noeud de degré 1 autre que a et c – auquel cas on ne peut pas à la fois entrer et sortir de ce noeud, donc pas construire un chemin de a à c. Inversement, s'il ne reste dans le graphe qu'un chemin de a à c, on a gagné.

1.3. Optimisations. On décrit dans cette partie quelques optimisations possibles (liste non limitative). La plupart des optimisations sont toutefois à inventer au cas par cas.

Symétries.– Les symétries du problème se reportent souvent dans l'arbre de calcul. Savoir les exploiter peut conduire à une diminution substantielle la taille de la partie de l'arbre à explorer, et donc du temps de calcul.

Recensement.– La remarque faite plus haut concernant le lien avec la programmation dynamique n'est pas innocente. On peut s'en inspirer pour tenter d'optimiser un peu le parcours de l'arbre, en essayant de détecter des sous-arbres identiques (ou symétriques) à ceux déjà traités. Pour ce faire, le plus pratique est en général de remplir un tableau, indexé par un haché des sous-problèmes rencontrés, pour signaler ceux qui ont déjà été traités.

Randomisation.– Souvent, et plus particulièrement dans les problèmes ayant une certaine « structure », un parcours de l'arbre aléatoire (par opposition à un parcours déterministe le parcourant de gauche à droite) peut amener à un résultat dans des délais bien plus brefs. C'est l'un des paradigmes qui fait le succès des algorithmes probabilistes, voir le chapitre 7.

Petites tailles.– Il peut s'avérer payant de programmer très proprement une résolution spécifique du problème dans les cas de petite taille, et de la substituer au test heuristique dans ces cas. Il faut se rappeller que si l'arbre est de degré 2 par exemple, 15/16 (soit 94%) des nœuds de l'arbre correspondent à des problèmes de taille 4 ou moins.

1.4. Branch and bound. La version *branch-and-bound* du *backtracking* correspond au cas des problèmes d'optimisation. La philosophie générale en est la même, à savoir se reposer sur le parcours de l'arbre de calcul. Toutefois, alors que dans le cas précédent, on pouvait, lorsque l'on ne cherchait qu'une seule solution, se dispenser d'examiner une bonne partie de l'arbre une fois une solution trouvée, dans le cas présent, dans la mesure où l'on souhaite obtenir la meilleure solution, il peut sembler nécessaire de parcourir l'intégralité de l'arbre.

On peut toutefois mettre en place un mécanisme analogue au TEST HEURISTIQUE utilisé dans le BACKTRACKING. Simplement, il faut construire un procédé d'ÉVALUATION HEURISTIQUE, qui, étant donné une branche, peut garantir que toute solution issue de cette branche a un coût (ou un objectif) associé égal au plus (ou au moins) à une valeur donnée.

Comment utiliser cette évaluation heuristique. Supposons que l'on dispose *a priori* d'une solution, même de mauvaise qualité. Cela nous permet de disposer d'une borne supérieure sur le coût (ou inférieure sur l'objectif). On peut donc se dispenser de s'intéresser aux solutions donc le coût ou l'objectif sont moins bons que ces bornes. C'est là que l'ÉVALUATION HEURISTIQUE intervient. On peut se dispenser d'examiner une branche de l'arbre de calcul si la procédure d'évaluation retourne une valeur moins bonne que la borne déjà connue.

Dans le cas où l'on n'a pas de solution initiale, il faut parcourir l'arbre exhaustivement jusqu'à en trouver une.

Il va sans dire que les bornes connues sur la meilleure solution doivent être remises à jour en fonction et au fur et à mesure des solutions trouvées jusqu'alors, i.e. si l'on trouve une solution qui améliore la borne, cette dernière est remplacée par la valeur de ladite solution.

1.5. Exemple : programmation linéaire en nombres entiers. L'exemple sans doute le plus générique de *branch-and-bound* est le problème de la programmation linéaire en nombres entiers, auquel bon nombre de problèmes d'optimisation difficiles se ramène. On rappelle que cette question consiste à résoudre un programme linéaire, en ajoutant une contrainte d'intégralité sur un certain nombre de variables. Ni l'algorithme du simplexe, ni l'algorithme primal-dual ne savent gérer ce type de contraintes.

On va toutefois pouvoir utiliser une stratégie de *branch-and-bound*, l'estimation de la solution maximale dans une branche donnée consistant à utiliser le simplexe usuel (ou toute autre méthode), en relâchant la contrainte d'intégralité, pour obtenir une borne inférieure (dans le cas de la minimisation) sur le minimum possible en nombres entiers.

Comment choisir la division du problème initial ? Une heuristique pour la programmation linéaire en nombres entiers consiste à appliquer le simplexe usuel, puis tenter d'arrondir de façon plus ou moins aléatoire. On a vu en exercice qu'on n'obtenait pas ainsi forcément une solution optimale. Plus significative est la stratégie qui consiste alors à découper le domaine du simplexe en deux régions disjointes $x_i \leq \lfloor x_i^0 \rfloor$ et $x_i \geq \lfloor x_i^0 \rfloor + 1$, où x_i est une valeur telle que le simplexe usuel a produit une valeur x_i^0 non entière. Observons que cela conduit à éliminer une bande de largeur 1 dans la direction i du domaine réalisable, et que donc la réunion des deux domaines plus petits est moins grande que le domaine initial. Qui plus est, imposer ce type de contrainte force une coordonnée entière sur certains sommets du nouveau simplexe, ce qui augmente les chances d'obtenir une solution entière.

En résumé, on procède de la façon suivante :

(1) Résoudre le problème courant par l'algorithme du simplexe, et obtenir une solution x^0. Si la fonction d'objectif est supérieure ou égale à la meilleure valeur obtenue jusqu'alors associée à une solution entière, *backtracker*.

(2) Si toutes les coordonnées de x^0 sont entières, mettre à jour la valeur optimale de l'objectif, et *backtracker*.

(3) Sinon, choisir une coordonnée i telle que x_i^0 soit non entière ; créer deux sous-problèmes en ajoutant dans l'un la contrainte $x_i \leq \lfloor x_i^0 \rfloor$, dans l'autre $x_i \geq \lfloor x_i^0 \rfloor + 1$, et les rajouter à la liste.

2. Algorithmes pseudo-polynomiaux

Il est bon, à propos de problèmes **NP**-complets liés aux nombres, de signaler la notion d'*algorithme pseudo-polynomial*. De façon informelle, un algorithme *pseudo-polynomial* est un algorithme dont la complexité est polynomiale en la taille des entrées et en le maximum des nombres apparaissant dans ces entrées. L'intérêt de cette notion vient du fait que les instances que l'on peut rencontrer des problèmes de nombres seront fréquemment des instances mettant en jeu de petits entiers. Cette notion n'a de sens que si certaines des entrées sont des nombres, comme pour les problèmes considérés dans le présent paragraphe, ou pour le problème PRIMALITÉ évoqué dans le préambule.

Cela revient à dire que l'on relâche l'exigence de caractère polynomial (rappelons que la longueur du nombre a est $\max(0, \lceil \log_2(a) \rceil) + 1$ pour les variables entières. En particulier, si un problème n'a pas d'entiers parmi ses données (cf. circuit hamiltonien), les notions d'algorithmes polynomial et pseudopolynomial coïncident.

Ce n'est en revanche pas toujours le cas. Noter par exemple que PRIMALITÉ admet trivialement un algorithme pseudo-polynomial (énumérer tous les diviseurs possibles de n se fait en un temps borné par n). En particulier, la plupart des problèmes à donnée numérique étudiés au chapitre 5 admettent des algorithmes pseudo-polynomiaux.

THÉORÈME 6.1. SAC-À-DOS, PARTITION *et* SOMME DE SOUS-ENSEMBLE *admettent des algorithmes pseudo-polynomiaux.*

DÉMONSTRATION. PARTITION et SOMME DE SOUS-ENSEMBLE sont des cas particuliers de SAC-À-DOS. Il suffit donc de le prouver pour ce dernier. Définissons $P(i, v)$ le poids minimum d'un sac rempli avec un sous-ensemble de $\{1, \ldots, i\}$ et de valeur au moins v. On a la récurrence

$$P(i + 1, v) = \min(P(i, v), P(i, v - v_{i+1}) + p_{i+1}),$$

selon que l'on prend ou ne prend pas le $(i+1)$-ème objet.

La taille de la table de programmation dynamique est alors $O(n \sum_i v_i)$, soit encore $O(n^2 \max v_i)$; comme chaque case prend $O(1)$ à remplir à partir des précédentes, le présent algorithme s'exécute en temps $O(n^2 \max v_i)$. Cette complexité n'est *pas polynomiale* en général (l'entrée est de taille $\sum lg(a_i) + \sum lg(v_i)$, avec $lg(x) = \log_2(x) + 1$ si x entier non nul. La complexité montre bien, en revanche, que cet algorithme rentre dans le cadre des algorithmes pseudo-polynomiaux étudiés dans ce paragraphe. □

2.1. NP-complétude au sens fort.

DÉFINITION 6.1. *On dira qu'un problème est* **NP**-*complet au sens fort si l'instance de ce problème où les données entières sont de taille bornée par un polynôme en les autres données est* **NP**-*complète.*

Le théorème suivant est une conséquence facile de la définition.

THÉORÈME 6.2. *Si un problème* **NP**-*complet au sens fort admet un algorithme pseudo-polynomial, alors* **P**=**NP**.

Outre les problèmes sans donnée entière, on peut montrer qu'il existe des problèmes **NP**-complets au sens fort.

THÉORÈME 6.3. PROGRAMMATION ENTIÈRE *est* **NP**-*complet au sens fort.*

DÉMONSTRATION. Il suffit de noter que la réduction de 3-SAT à PROGRAMMATION ENTIÈRE construit une instance où tous les coefficients de la matrice et du vecteur sont dans $\{-1, 0, 1\}$, et que cette instance est donc déjà **NP**-complète. □

On peut aussi montrer :

THÉORÈME 6.4. RANGEMENT OPTIMAL *est* **NP**-*complet au sens fort.*

Ce résultat contraste avec celui obtenu pour les trois autres problèmes de nombres du paragraphe 5 du chapitre 5. Si l'on examine la preuve de la **NP**-complétude de RANGEMENT OPTIMAL, on s'aperçoit que le cas $k = 2$ équivaut essentiellement à PARTITION, et admet donc un algorithme pseudo-polynomial. Il n'en est en revanche rien quand on admet que k est plus grand, en vertu du théorème qui précède.

3. Algorithmes d'approximation

Pour un état de l'art en 1999 sur les problèmes d'optimisation, on pourra consulter l'ouvrage [**1**] ou l'adresse `http://www.nada.kth.se/~viggo/wwwcompendium/` (attention, la définition du facteur d'approximation est différente de la nôtre, et donne en particulier des valeurs dans $]0, \infty[$.

3.1. Techniques de conception. Il est utile de noter que la plupart des algorithmes d'approximation étudiés dans ce chapitre sont basés sur des idées très simples, qui sont essentiellement celle de la première partie de ce cours. Les méthodes de conception les plus fréquemment rencontrées sont

(1) la méthode gloutonne, qui fournit très souvent des heuristiques simples et efficaces mais hélas pas toujours de bonne qualité ;
(2) la programmation dynamique, comme on l'a déjà vu dans le paragraphe précédent ;
(3) la programmation linéaire dont on arrondit les solutions, utilisée pour approcher les solutions d'un problème linéaire en nombres entiers ;
(4) la recherche locale, que nous discuterons brièvement ;
(5) enfin, les algorithmes probabilistes.

Nous illustrons les méthodes (1), (3) et (4) par des exemples. Pour (2), voir le Théorème 6.1 et le Théorème 6.10. Pour (5), voir le chapitre 7.

Construisons une heuristique gloutonne pour le problème RANGEMENT OPTIMAL. L'idée naturelle consiste à ranger tout d'abord les objets les plus encombrants, donc à les classer par ordre de poids décroissant. Un objet est ensuite rangé dans la boîte de plus petit numéro dont la capacité est suffisante.

THÉORÈME 6.5. *Soit k le nombre de boîtes nécessaire par cette méthode, et k_{opt} le nombre optimal. On a alors*

$$k \leq 3k_{\text{opt}}/2 + 1.$$

DÉMONSTRATION. Soit B la capacité d'une boîte. Supposons qu'il existe une boîte b_0 ne contenant que des objets de poids $\leq B/3$. Cela implique qu'hormis cette boîte, toutes les autres sont pleines au moins aux deux tiers, sans quoi on aurait commencé par ranger certains des objets de b_0 dans ces boîtes avant de remplir b_0. Le poids total des objets est donc au moins $2(k-1)B/3$; il est au plus $k_{\text{opt}}B$, et l'inégalité s'ensuit.

Si aucune boîte ne contient que des objets de poids $\leq B/3$, on peut supposer que tous les objets de l'instance sont de poids $\geq B/3$ sans modifier la solution obtenue (on rappelle qu'on commence par traiter les objets de poids le plus grand). Montrons que dans cette situation, on obtient effectivement la solution optimale. Comme tout objet de poids $\geq 2B/3$ est nécessairement seul dans une boîte, on peut à nouveau se limiter au cas où tous les objets sont de poids compris entre $2B/3$ et $B/3$. Dans ce dernier cas, chaque boîte contient au plus deux objets, dont un seul de poids $\geq B/2$. L'affectation optimale correspond alors nécessairement à réaliser le maximum de couples d'objets de $A \times B$ de poids total au plus B, où A est l'ensemble des objets de poids de $]B/2, 2B/3[$ et B l'ensemble des objets de poids $]B/3, B/2]$; les objets restants sont traités de façon évidente (laissés seuls pour les éléments de A, appariés deux à deux de façon quelconque pour les éléments de B). Comme notre algorithme place les éléments de A dans des boîtes d'ordre croissant, et tente ensuite de placer tout élément de B avec un élément de A de poids le plus grand possible, il est facile de voir que l'on obtient bien le nombre maximal de couples de $A \times B$. □

Noter que l'argument peut être raffiné par une étude plus détaillée pour obtenir la borne $11k_{\text{opt}}/9 + 4$. Ce résultat est en un sens optimal, comme le montre l'exemple où l'on a n objets de taille $1/2 + \varepsilon$, n de taille $1/4 + 2\varepsilon$, n de taille $1/4 + \varepsilon$, et $2n$ de taille $1/4 - 2\varepsilon$, où l'heuristique nécessite $11n/6$ boîtes [n boîtes $(1/2+\varepsilon)+(1/4+2\varepsilon)$, $n/3$ boîtes $3 \cdot (1/4+\varepsilon)$, $n/2$ boîtes $4 \cdot (1/4 - 2\varepsilon)$] contre $3n/2$ [n boîtes $(1/2+\varepsilon)+(1/4+\varepsilon)+(1/4-2\varepsilon)$, $n/2$ boîtes $2 \cdot (1/4+2\varepsilon) + 2 \cdot (1/4 - 2\varepsilon)$] pour un résultat optimal.

On peut également concevoir un algorithme d'approximation pour SAC-À-DOS en tentant de formaliser le compromis entre poids et bénéfice de façon approchée, en introduisant le *rapport qualité-poids* d'un objet a_i/v_i ; on classe les objets par rapport qualité-poids décroissant, et l'on ajoute les éléments au sac dans cet ordre lorsque c'est possible sans violer la contrainte de poids. Enfin, pour une raison technique, on compare en fin de calcul le bénéfice obtenu à celui obtenu en mettant dans le sac le seul élément de bénéfice maximal parmi ceux de poids inférieur à la capacité du sac, et on choisir le meilleur résultat des deux.

THÉORÈME 6.6. *Si A est le bénéfice obtenu par l'heuristique gloutonne et A_{opt} le résultat optimal, on a*

$$A < 2A_{\text{opt}}.$$

DÉMONSTRATION. Soit j le premier élément dans l'ordre décroissant de rapport qualité-poids qui n'est pas pris dans le sac. On note A_j le bénéfice apporté par les éléments déjà dans le sac, et V_j le poids des éléments présents dans le sac.

La stratégie gloutonne est conçue de sorte à ce qu'enlever des éléments déjà dans le sac par des éléments d'ordre au moins égal à j ne peut améliorer le bénéfice que si le poids total est augmenté, et d'un facteur au plus égal à a_j/v_j fois l'augmentation de poids. Cette augmentation de poids peut être au plus $C - V_j$, où C est la capacité du sac. Comme en outre $V_j + v_j > C$ (sinon le j-ème élément est rajouté dans le sac par l'heuristique gloutonne), on a donc l'inégalité

$$A_{\text{opt}} \leq A_j + (C - V_j)a_j/v_j < A_j + a_j.$$

Si $a_j \leq A_j$, le résultat s'ensuit trivialement. Dans le cas contraire, $a_j \leq a_{\max}$, où $a_{\max}$ est l'élément de bénéfice maximal parmi ceux qui rentrent dans le sac, et donc $A_{\text{opt}} < 2a_j \leq 2a_{\max}$, et l'on obtient encore le résultat, grâce à la modification technique apportée en fin d'algorithme. □

On illustre maintenant l'usage de techniques de programmation linéaire. Considérons le problème RECOUVREMENT DE SOMMET PONDÉRÉ, qui est identique à RECOUVREMENT DE SOMMET, hormis que l'on attache un poids à chacun des sommets et que l'on cherche à minimiser ce poids. RECOUVREMENT DE SOMMET correspond au cas particulier où tous les poids valent 1, et RECOUVREMENT DE SOMMET PONDÉRÉ (ou tout au moins le problème de décision associé) est donc bien **NP**-complet. Le programme linéaire en nombres entiers suivant est équivalent à RECOUVREMENT DE SOMMET PONDÉRÉ$((V, E), p_i)$, où les variables x_{v_i} valent 1 si v_i est dans le recouvrement optimal, 0 sinon :

$$\begin{cases} \min \sum_{v_i \in V} p_i x_{v_i}, & \\ x_{v_i} + x_{v_j} \geq 1, & \forall (v_i, v_j) \in E, \\ x_{v_i} \in \{0, 1\}, & \forall v_i \in V. \end{cases}$$

Le cas où les p_i sont tous égaux à 1 correspond en particulier exactement au problème RECOUVREMENT DE SOMMETS du chapitre 5.

On peut résoudre ce problème par une des méthodes vues au chapitre 7.2.2, en transformant la dernière contrainte en $x_{v_i} \geq 0$ et retenir dans le recouvrement les variables de valeur $\geq 1/2$.

THÉORÈME 6.7. *La solution renvoyée par l'algorithme ci-dessus est bien un recouvrement. En outre, si* p_{opt} *est la valeur optimale et* p_0 *la valeur obtenue par l'algorithme, alors on a*

$$p_0 \leq 2p_{\text{opt}}.$$

DÉMONSTRATION. Si la solution n'est pas un recouvrement, alors il existe une arête telle que x_{v_i} et x_{v_j} soient strictement inférieurs à $1/2$, ce qui contredit la contrainte correspondante.

Maintenant, la solution optimale est toujours au plus égale à la solution du problème linéaire général, car ce dernier est identique à ceci près qu'il relâche les contraintes d'intégralité. Notant que l'on a $2x_{v_i} \geq 1$ pour tout i, il s'ensuit que

$$p_0 = \sum_{v_i \in V; x_{v_i} \geq 1/2} p_i \leq \sum_{v_i \in V} 2p_i x_{v_i} \leq 2\tilde{p} \leq 2p_{\text{opt}},$$

où $\tilde{p}$ est la valeur de l'objectif obtenue pour la solution optimale du programme linéaire. □

D'autres exemples d'algorithmes approchés se trouvent dans les exercices du cours sur la programmation linéaire ; enfin, le chapitre suivant montrera comment l'on peut parfois utiliser une stratégie probabiliste pour choisir l'arrondi.

On présente enfin un exemple où l'on utilise une technique d'amélioration locale. Cette technique consiste à partir d'une solution réalisable, et à tenter d'améliorer cette solution par une modification locale. On recommence à nouveau sur la nouvelle solution obtenue, etc. jusqu'à ce que l'on ait obtenu un maximum local. Cette technique est très voisine dans l'esprit des techniques de recherche d'extrema par des méthodes de type Newton (suivre la ligne de plus grande pente) en analyse numérique, en ce sens qu'elle conduit en général à un optimum local, qui peut être significativement moins bon que l'optimum global. Dans le cas présent, en revanche, il n'y a pas de stratégie « générique »d'amélioration locale, et il faut en construire une au coup par coup.

On rappelle l'énoncé de COUPURE MAXIMALE, qui a été prouvé **NP**-complet au Théorème 5.4 du chapitre 5.

COUPURE MAXIMALE :

Données : un graphe $G = (V, E)$ et un entier k.

Résultat : existe-t-il une partition $V = V_1 \cup V_2$ telle que le nombre d'arêtes (v_1, v_2) avec $v_1 \in V_1$ et $v_2 \in V_2$ soit au moins k ?

On construit, partant d'une coupure donnée (par exemple $V_1 = V$, $V_2 = \emptyset$) une méthode d'amélioration comme suit : les voisins d'une situation (V_1, V_2) donnés sont obtenus soit comme $(V_1 - \{a\}, V_2 \cup \{a\})$ pour $a \in V_1$, soit comme $(V_1 \cup \{a\}, V_2 - \{a\})$ pour $a \in V_2$. On choisit à chaque étape le a qui améliore le plus la valeur de la coupure.

THÉORÈME 6.8. *Si c_{opt} est la valeur optimale et c la valeur de la coupure obtenue par l'algorithme, alors on a*

$$c_{\text{opt}} \leq 2c.$$

DÉMONSTRATION. Il suffit de prouver que la coupure localement optimale construite contient au moins la moitié des arêtes du graphe. Soit, pour cette coupure optimale, a_1 le nombre d'arêtes de V_1, a_2 le nombre d'arêtes de V_2. On a alors trivialement que $a_1 + a_2 + c$ est le nombre d'arêtes du graphe.

La propriété d'optimalité locale signifie que tout élément de V_1 a plus de voisins dans V_1 que dans V_2, et de même pour les éléments de V_2. Par conséquent, en sommant sur tous les sommets de V_1, le nombre total d'arêtes de V_1, compté avec un facteur 2 (une arête est comptée pour chacune de ses extrémités) est au plus égal à c, et de même pour V_2. Il en résulte que $a_1 + a_2 \leq c$, ce qui, combiné avec le fait que $a_1 + a_2 + c$ est le nombre d'arêtes du graphe, conclut la preuve. □

3.2. Définitions. Ce paragraphe formalise quelques-unes des notions étudiées dans les paragraphes précédents autour des algorithmes d'approximation.

DÉFINITION 6.2. *Considérons un problème d'optimisation prenant une entrée x et donné par un problème de décision $\mathrm{P}(x)$ et une fonction de coût $c(x, y)$ à valeurs réelles positives ; la solution au problème d'optimisation est alors le nombre*

$$\phi_{P,c}(x) = \min_{\{y;\mathrm{P}(x)(y)=\texttt{true}\}} c(x, y).^2$$

Un algorithme ε-approché pour $(\mathrm{P},\, c)$ calcule un élément $A(x)$ tel que $\mathrm{P}(x)(y) = \texttt{true}$, et que

$$\frac{|c(A(x)) - \phi_{P,c}(x)|}{\max(c(A(x)), \phi_{P,c}(x))} \leq \varepsilon.$$

Noter que le max au dénominateur sert uniquement à traiter simultanément le cas de problèmes de maximisation et de minimisation. La définition de $\phi_{P,c}(x)$ permet de choisir aisément le bon terme selon le contexte. En particulier, le quotient du membre gauche est toujours compris entre 0 et 1.

Le seuil d'approximation d'un problème d'optimisation est la borne inférieure de l'ensemble des ε pour lesquels il existe un algorithme ε-approché pour ce problème. Intuitivement, plus ce seuil est proche de 0, meilleure est l'approximation, et plus l'algorithme approché fournira une solution satisfaisante.

Il est bon de noter qu'on trouve également d'autres définitions de la notion d'algorithme θ-approché, la plus fréquente consistant à dire que l'on a

$$\max(c(A(x))/\phi_{P,c}(x), \phi_{P,c}(x)/c(A(x))) \leq \theta.$$

[2] On peut aussi considérer des problèmes de maximisation ; le coût est alors traditionnellement baptisé objectif.

Dans ce cas, θ est toujours au moins égal à 1, ce qui permet de lever l'ambiguïté dans tous les cas.

Les résultats obtenus depuis le début du chapitre s'énoncent alors :

THÉORÈME 6.9. RANGEMENT OPTIMAL, SAC-À-DOS, RECOUVREMENT DE SOMMETS PONDÉRÉ, COUPURE MAXIMALE *admettent des algorithmes (1/2)-approchés.*

DÉMONSTRATION. Seul le cas de RANGEMENT OPTIMAL est non trivial, car le $+1$ de la borne supérieure n'est *a priori* pas pris en compte dans la définition d'un algorithme ε-approché. On peut toutefois noter que l'algorithme donné est exact dans le cas ou l'on n'a qu'une seule boîte ; cette remarque montre que l'on peut supposer $k_{\mathrm{opt}} \geq 2$, et donc que $3k_{\mathrm{opt}}/2 + 1 \leq 2k_{\mathrm{opt}}$. Il peut paraître décevant de n'être pas capable de faire mieux qu'une borne de 2 pour le nombre de boîtes, mais rappelons que décider si 2 boîtes exactement suffisent contient le problème PARTITION, lui-même **NP**-complet. Voir encore sur ce point le Théorème 6.13. □

3.3. Exemples. Ce paragraphe a pour objectif de montrer le comportement très variable des problèmes **NP**-complet vis-à-vis de l'approximation. La raison fondamentale de cette différence réside dans le fait que, lors de la réduction de A à B, on ne contrôle absolument pas le lien entre coût de l'instance de A et coût de l'instance de B correspondante. En particulier, l'instance optimale de B peut correspondre à une très mauvaise instance de A, ou inversement. On peut contourner cette difficulté en considérant la notion de *L-réduction*, où l'on impose à la fonction τ de réduction de vérifier en plus que $\phi_{B,c'}(\tau(x)) \leq \alpha\phi_{A,c}(x)$ (dans le cas d'un problème de maximisation). Nous n'explorerons pas cette notion plus avant, mais étudierons plutôt divers exemples.

Commençons par les succès.

THÉORÈME 6.10. *Le seuil d'approximation de* SAC-À-DOS *est 0.*

DÉMONSTRATION. Ce résultat vient du fait que la difficulté de SAC-À-DOS provient essentiellement de la taille des valeurs $v_1, \ldots, v_n$. Dans le cas où ces valeurs sont bornées par une quantité fixée, il existe un algorithme polynomial simple et efficace, basé sur la programmation dynamique, voir le paragraphe 2. L'algorithme d'approximation repose sur cette idée ; on va en fait simplement *approcher le problème*, en remplaçant les valeurs v_i par des valeurs tronquées. On forme les $w_i = \lfloor v_i 2^{-k} \rfloor$, k étant un entier à choisir, et l'on étudie le problème du sac-à-dos donné par les a_i et les w_i. On sait obtenir une solution optimale $W = \sum \varepsilon_i w_i$, $\varepsilon_i \in \{0, 1\}$ de ce problème en temps $O(n^2 2^{-k} \max v_i)$.

Il nous faut maintenant comparer $V^* = \sum \varepsilon_i v_i$ et la solution optimale du problème de départ $V = \sum \varepsilon'_i v_i$.

On a $V = 2^k \sum \varepsilon'_i v_i 2^{-k} \leq 2^k \left(\sum \varepsilon'_i (\lfloor v_i 2^{-k} \rfloor + 1)\right) \leq 2^k \sum \varepsilon'_i w_i + 2^k n$.

En vertu de la définition des ε_i, le premier terme est au plus égal à $2^k \sum \varepsilon_i w_i$; or $w_i \leq v_i 2^{-k}$, donc ce terme est majoré par $\sum \varepsilon_i v_i = V^*$. On a donc

$$V^* \leq V \leq V^* + 2^k n.$$

Par suite, le facteur d'approximation vaut donc $2^k n/V$. Imposer un seuil d'approximation de θ revient donc à choisir

$$k \geq \log_2 \left(\frac{\theta V}{n} \right) \geq \log_2 \left(\frac{\theta \max_i v_i}{n} \right).$$

On vérifie que ce choix conduit à un algorithme θ-approché en temps $O(n^3/\theta)$. Par conséquent, pour tout $\theta > 0$, SAC-À-DOS admet un algorithme polynomial θ-approché. Son seuil d'approximation est donc 0. □

Voici maintenant un cas médian.

THÉORÈME 6.11. RECOUVREMENT DE SOMMETS *a un seuil d'approximation au plus égal à* $1/2$.

DÉMONSTRATION. On peut déduire ce résultat du théorème 6.7, mais nous en donnons une preuve différente.

Rappelons que le problème consiste à trouver un ensemble minimal de sommets S d'un graphe non-orienté $G = (V, E)$ tel que toute arête ait un sommet dans S.

On propose l'algorithme simple suivant : à une étape donnée de l'algorithme, on choisit une arête (u, v) de G, on ajoute *à la fois* u et v à S, et l'on supprime l'arête (u, v) de G. On s'arrête quand G ne contient plus aucune arête.

Montrons que cet algorithme est un algorithme 1/2-approché. Le nombre d'arêtes de G reliant des sommets de S est $|S|/2$. Tout recouvrement de E doit contenir au moins un des sommets de chacune de ces arêtes, donc est de cardinal au moins $|S|/2$; cela implique que, si $|O|$ est le cardinal d'un recouvrement minimal, on a $||O| - |S|| \leq |S|/2$, ce qui est précisément la définition d'un algorithme de minimisation 1/2-approché. □

THÉORÈME 6.12. *Le problème du voyageur de commerce dans le cas où la distance vérifie l'inégalité triangulaire a un seuil d'approximation d'au plus* $1/2$.

DÉMONSTRATION. On considère le graphe G correspondant, et l'on construit au moyen d'un algorithme glouton (cf. chapitre 1) un arbre recouvrant de poids minimal en temps $O(n^2)$. On construit alors une tournée en faisant le parcours en profondeur de l'arbre recouvrant minimal. Dans ce parcours, chaque arête est décrite deux fois. Si c est le coût de l'arbre recouvrant minimal, on obtient donc une tournée de coût $2c$. On peut, dans cette tournée, remplacer chaque parcours de type $a \to b \to a \to c$, correspondant au sous-arbre de racine a dont les fils sont b et c, par le parcours $a \to b \to c$; cela ne peut que diminuer le coût de la tournée, en vertu de l'inégalité triangulaire.

Soit maintenant une tournée optimale ; si l'on enlève une arête à cette dernière, on obtient un arbre recouvrant pour le graphe, de coût inférieur au coût de la tournée, et de coût supérieur au coût de l'arbre recouvrant minimal. Le coût de la tournée que nous avons construite au paragraphe précédent, est donc au plus le double du coût d'une tournée optimale, ce qui conclut la preuve. □

3.4. Résultats négatifs. Enfin, on peut, dans ce contexte, obtenir des résultats négatifs – toujours sous réserve que **P**= **NP**.

THÉORÈME 6.13. *Si* **P**≠ **NP**, *il n'existe pas d'algorithme ε-approché pour* RANGEMENT OPTIMAL *avec* $\varepsilon < 1/3$.

DÉMONSTRATION. On va montrer que dans le cas contraire, on sait décider de PARTITION en temps polynomial. La réduction construite dans le chapitre 5 montre que partant d'une instance de PARTITION, on peut construire une instance de RANGEMENT OPTIMAL, de sorte que la réponse à PARTITION soit 1 si et seulement si le résultat de RANGEMENT OPTIMAL est 2. Un algorithme ε-approché renvoie alors comme réponse à RANGEMENT OPTIMAL une valeur au plus égale à $2/(1-\varepsilon)$. En particulier, si $\varepsilon < 1/3$, RANGEMENT OPTIMAL a pour résultat 2 si et seulement si l'algorithme ε-approché renvoie une valeur strictement inférieure à 3. On peut donc décider PARTITION en temps polynomial. Si **P**≠ **NP**, il n'existe donc pas d'algorithme ε-approché pour RANGEMENT OPTIMAL avec $\varepsilon < 1/3$. □

Ce résultat peut s'énoncer comme le fait que le problème RANGEMENT OPTIMAL a un seuil d'approximation d'au moins 1/3. Noter que ce résultat ne contredit pas le Théorème 6.5, en raison du facteur additif qui fait qu'un algorithme de borne supérieure $11k_{\text{opt}}/9+4$ n'est généralement pas ε-approché.[3] La technique de preuve est assez courante, et repose sur le fait qu'il existe un problème **NP**-complet de décision qui se réduit en un sens à notre problème d'optimisation, de sorte que les réponses positives au problème de décision donnent une valeur A au plus, et les réponses négatives une valeur B au moins ; dans ce contexte, la généralisation de la preuve précédente montre qu'il ne peut exister d'algorithme ε-approché en temps polynomial avec $\varepsilon < 1 - A/B$, sauf si **P**= **NP**. Ainsi, la **NP**-complétude de 3-COLORABILITÉ montre que le problème du coloriage d'un graphe avec un nombre minimal de couleurs a un seuil d'au moins 1/4 (en fait, on peut montrer que ce seuil est égal à 1).

THÉORÈME 6.14. *Si* VOYAGEUR DE COMMERCE *admet un algorithme ε-approché pour* $\varepsilon < 1$, *alors* **P** = **NP**.

DÉMONSTRATION. La preuve repose sur le fait que tout algorithme ε-approché pour VOYAGEUR DE COMMERCE peut être utilisé pour résoudre le problème CIRCUIT HAMILTONIEN.

Supposons en effet qu'il existe un algorithme polynomial ε-approché pour VOYAGEUR DE COMMERCE, et posons $n = \lceil (1-\varepsilon)^{-1} \rceil$. Soit maintenant une instance du problème du circuit hamiltonien, donnée par un graphe $G = (V, E)$ non orienté. On considère alors le graphe $G' = (V, V \times V)$ valué par $v(x, y) = 1$ si $(x, y) \in E$, $v(x, y) = |V|n + 1$ sinon. Il est bien clair dans ces conditions que G admet un circuit hamiltonien dans si et seulement si le problème du voyageur de commerce pour G' a une solution de coût inférieur à $|V|$. Dans le cas contraire, le coût d'une tournée de G' est au moins égal à $|V|n + 1$.

[3]Dans le cas présent, on peut montrer en imitant la preuve du Théorème 6.9 que l'algorithme est 20/29-approché.

Si l'on applique l'algorithme ε-approché à G', on obtient une tournée dont le coût est au plus $(1-\varepsilon)^{-1}\mathrm{opt}$. Par suite,

– Si G admet un circuit hamiltonien, on obtient donc une tournée de coût inférieur à $n|V|$;
– si G n'admet pas de circuit hamiltonien, on obtient une tournée de coût au moins égal à $n|V|+1$.

L'algorithme ε-approché appliqué à G' permet donc de décider s'il existe un circuit hamiltonien en temps polynomial, ce qui conclut la preuve. □

Noter que la preuve précédente repose encore une fois sur le même principe, encore que la technique de réduction depuis le problème **NP**-complet sous-jacent soit un peu plus subtile.

Concernant SAC-À-DOS, on peut se demander s'il est possible d'améliorer encore la situation ; on sait déjà qu'on peut rendre, en proportion, l'erreur aussi petite que l'on veut. Est-il possible de borner cette erreur par une constante ?

THÉORÈME 6.15. *Si* SAC-À-DOS *admet un algorithme polynomial donnant un résultat distant de l'optimum d'au plus une constante fixée, alors* **P**$=$ **NP**.

DÉMONSTRATION. Supposons qu'il existe un algorithme polynomial donnant un résultat distant de l'optimum d'au plus k. On construit alors une nouvelle instance $(p_i, a_i(k+1))$. Le choix optimal n'est pas modifié, seul l'objectif a été multiplié par $k+1$. Le nouvel optimal est donc maintenant $(k+1)\cdot A_{\mathrm{opt}}$. Mais toute solution non optimale a un coût multiple de $k+1$ et strictement supérieur à $(k+1)\cdot A_{\mathrm{opt}}$, donc distant de $(k+1)\cdot A_{\mathrm{opt}}$ de strictement plus de k. Par suite, la seule valeur pouvant être renvoyée par l'algorithme est l'optimum. Par suite, si l'on sait trouver un algorithme polynomial donnant un résultat distant de l'optimum d'au plus k, on peut l'utiliser pour construire un algorithme polynomial pour SAC-À-DOS, et **P**$=$ **NP**. □

CHAPITRE 7

Méthodes probabilistes

Introduction et paradigmes

L'étude des algorithmes probabilistes est d'importance cruciale dans l'algorithmique. Ils permettent fréquemment d'échapper, ou du moins de réduire, les situations de pire cas, et fournissent donc fréquemment de bons algorithmes d'approximation pour des problèmes **NP**-complets (voire de complexité encore supérieure). Il arrive aussi que l'on préfère implanter un algorithme probabiliste, de complexité en moyenne correspondant à la complexité dans le cas le pire du meilleur algorithme déterministe, mais bien plus simple à implanter.

La vision de la théorie de la complexité sur les algorithmes probabilistes consiste à considérer une machine de Turing non-déterministe que l'on normalise de sorte que chaque choix non-déterministe soit fait parmi deux possibilités ; on construit ainsi, dans le cas polynomial, un arbre binaire (que l'on peut toujours supposer complet) de calcul, ayant $2^{p(|x|)}$ feuilles sur une entrée x (p polynôme), et l'on considère chacun de ces calculs comme équiprobables, c'est-à-dire que chacun des branchements de l'algorithme se fait avec probabilité $1/2$. Plus simplement, on peut considérer un algorithme probabiliste comme un algorithme disposant en plus d'une fonction `alea`, qui retourne des valeurs 0 ou 1 avec probabilité $1/2$, cette fonction étant utilisée pour opérer des choix.

Le principal intérêt d'utiliser le hasard vient du fait que ce dernier permet de se placer dans une situation générique. Ce principe sous-tend la majorité des algorithmes probabilistes. On peut illustrer ceci sur un exemple. Le pire des cas de l'algorithme de tri `quicksort` correspond à la situation où la liste donnée en entrée est triée. On peut alors imaginer, avant de démarrer `quicksort`, d'appliquer une permutation aléatoire à la liste à trier. Cela permet de transférer le cas le pire vers une situation n'ayant pas de structure « a priori ». Même si cela ne modifie pas la répartition des complexités, d'un point de vue pratique cette idée s'avère souvent utile.

Il importe toutefois de rester circonspect face à des résultats de complexité bien supérieurs via des algorithmes probabilistes ; il est en effet peu éclairant de comparer complexité dans le pire des cas de l'algorithme déterministe et complexité en moyenne de l'algorithme probabiliste, même si ladite moyenne porte sur la suite d'aléa utilisée par l'algorithme et non sur les données d'entrée. Quand l'algorithme probabiliste est juste un algorithme déterministe probabilisé de façon plus ou moins artificielle, on pourrait obtenir le même résultat en fixant une distribution de probabilité des entrées et en étudiant la complexité en moyenne (noter toutefois que l'algorithme probabiliste laisse toute latitude pour le choix de la distribution de l'aléa).

Énumérons succinctement quelques grandes idées qui font le succès des algorithmes probabilistes et sont au cœur de la conception de ces derniers.

- *généricité*. Comme décrit ci-dessus, le hasard fait qu'il est quasi-impossible de construire une entrée du problème qui réalise le pire des cas de l'algorithme probabiliste pour une proportion significative des aléas possibles. En d'autres termes, l'algorithme probabiliste nous place en situation générique avec forte probabilité.
- *échantillonage*. Le choix d'un « petit » échantillon aléatoire d'un grand ensemble permet de comprendre la structure du grand ensemble sur le petit, sous réserve que ce dernier soit représentatif, ce qui advient en pratique avec probabilité 1. Il n'y a pas vraiment de moyen d'atteindre cet objectif (sous-ensemble « représentatif ») avec un algorithme déterministe. C'est l'idée principale de l'algorithme de sélection de k-ième élément d'une liste, voir plus bas.
- *abondance de témoins*. Supposons que l'on souhaite tester une propriété $\mathcal{P}(x)$ pour x dans un grand ensemble X. On peut se contenter de la tester pour x dans un plus petit ensemble E. Si la propriété s'avère fausse une fois, on peut répondre sans risque de se tromper. Dans le cas contraire, l'idée d'échantillonage incite à penser que la propriété est vraie avec bonne probabilité. C'est le principe des tests de primalité, voir plus bas.
- *rupture de symétrie et répartition de charge*. Dans l'algorithmique distribuée, un problème fréquemment rencontré est le fait que si les différentes machines exécutent simultanément le même code déterministe, il est très difficile de briser la symétrie qui s'ensuit. En particulier, cela peut conduire à une répartition très hétérogène de la charge, voire à des situations de blocage.
- *existence probabiliste*. Si l'on cherche à construire un objet vérifiant une certaine propriété, il suffit de montrer qu'un objet aléatoire du même type vérifie cette propriété avec probabilité positive. Cela peut aussi s'appliquer à la recherche du maximum d'une fonction f sur un ensemble X ; si l'on tire des éléments de X au hasard pour obtenir une variable aléatoire Y et que l'on montre que $E(f(Y)) \geq a$, il s'ensuit qu'il existe un élément de X au moins tel que $f(y) \geq a$.

1. Quelques inégalités probabilistes

On renvoie le lecteur à son polycopié de probabilités où à l'un ou l'autre des ouvrages du commerce traitant de la théorie des probabilités, ou encore à l'appendice C de [**14**] pour les rappels élémentaires sur variables aléatoires (nous n'étudierons dans le cas présent que le cas discret), mesure de probabilité, moments, indépendance, probabilités conditionnelles, etc. Notons toutefois qu'une approche intuitive de ces notions nous semble largement suffisante dans le cadre du présent cours, même si l'étude plus approfondie des algorithmes probabilistes nécessite une maîtrise assez fine des techniques de la théorie des probabilités.

L'objet de ce paragraphe est juste de rappeler trois résultats élémentaires permettant de borner la taille des queues de distribution, suivant les renseignements dont on dispose.

En règle générale, lorsqu'on analyse un algorithme probabiliste, on peut obtenir assez facilement des renseignements sur la moyenne et la variance du résultat ou du temps de calcul, mais il est beaucoup plus délicat d'obtenir un résultat de distribution précis. Il se trouve que les renseignements sur les deux premiers moments suffisent à montrer que la probabilité de rencontrer des cas très éloignés du cas moyen (c'est-à-dire en pratique des cas où l'algorithme fournit un mauvais résultat, ou encore est très lent) est faible.

C'est précisément ce qu'affirment les inégalités de Markov et de Tchebitchev, que nous donnons sans preuves (au demeurant faciles) :

THÉORÈME 7.1. *(inégalité de Markov) Soit X une variable aléatoire positive de moyenne μ ; on a*

$$\Pr(X \geq t\mu\} \leq 1/t.$$

(inégalité de Tchebitchev) Soit X une variable aléatoire de moyenne μ et d'écart-type σ ; alors on a

$$\Pr(|X - \mu| \geq a\sigma\} \leq \frac{1}{a^2}.$$

Dans le cas particulier d'une somme de variables aléatoires indépendantes à valeurs $\{0, 1\}$ que l'on rencontre très fréquemment, on peut être beaucoup plus précis.

THÉORÈME 7.2 (Chernov). *Soit $(X_i)_{1\leq i\leq n}$ des variables aléatoires indépendantes prenant la valeur 0 avec probabilité p_i et la valeur 1 avec probabilité $1 - p_i$, où $0 < p_i < 1$. Alors, si $\mu = \sum_{i=1}^{n} p_i$, on a, pour tout $\delta > 0$,*

$$\Pr\left(\sum_{i=1}^{m} X_i > (1+\delta)\mu\right) < \left(\frac{\exp(\delta)}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

Noter que le terme intérieur à la parenthèse est toujours inférieur à 1, et vaut par exemple 0,68... pour $\delta = 1$, et $0,27...$ pour $\delta = 0$. Quand $\delta \to \infty$, il décroît comme $\exp(-\delta^2)$.

2. Las-Vegas et Monte-Carlo

On distingue classiquement deux classes d'algorithmes probabilistes, les algorithmes de type *Las-Vegas* et les algorithmes de type *Monte-Carlo*. On exige des algorithmes Las-Vegas qu'ils fournissent une réponse toujours exacte ; leur temps d'exécution doit, lui, être polynomial en moyenne (pour une définition plus précise, voir la définition de la classe **ZPP** dans les compléments en fin de chapitre).

À l'inverse, les algorithmes de type Monte-Carlo doivent avoir un temps d'exécution polynomial dans tous les cas, mais peuvent, avec une probabilité strictement inférieure à 1, fournir un résultat faux. Remarquons qu'en itérant plusieurs fois un algorithme de type Monte-Carlo ayant une certaine probabilité d'échec, on peut ramener cette probabilité aussi près de 0 qu'on le veut.

Noter que dans le cas d'un problème de calcul ou d'optimisation, si l'on dispose d'un algorithme polynomial permettant de vérifier le résultat, il est possible de transformer

« génériquement » un algorithme Monte-Carlo en algorithme Las Vegas (tant que la vérification montre qu'on s'est trompé, réappliquer l'algorithme ; noter que l'itération peut ne pas terminer, mais cela n'arrive qu'avec probabilité nulle).

3. Primalité

Le problème de décider si un nombre est premier intervient de façon fondamentale dans de nombreux algorithmes liés de près ou de loin à la théorie des nombres. On ne connaît à l'heure actuelle aucun algorithme polynomial déterministe permettant de décider si un nombre est premier, sauf si l'on admet une conjecture de théorie des nombres extrêmement profonde et difficile, l'hypothèse de Riemann généralisée[1], voir ci-après. Le meilleur algorithme déterministe connu, dit des sommes de Jacobi, dû à Cohen-Lenstra-Pomerance-Rumely, est de complexité $O((\log n)^{O(\log\log\log n)})$; le meilleur algorithme probabiliste (Las-Vegas) est ECPP, basé sur les courbes elliptiques. Sa complexité est polynomiale. Ces deux algorithmes nécessitant des développements théoriques considérables dépassant largement le cadre de ce cours, on se contente de décrire un test rapide et efficace, qui est celui implémenté – de façon plus ou moins heureuse – dans la plupart des systèmes de calcul formel actuels.

Ce test repose sur la proposition suivante :

Proposition 7.1. *Soit n un nombre premier, et posons $n-1 = 2^s t$ avec t impair. Alors pour tout $1 \leq a < n$, on a*
- *soit $a^t = 1 \bmod n$,*
- *soit il existe $k < s$ tel que $a^{2^k t} = -1 \bmod n$.*

Démonstration. La démonstration consiste essentiellement à factoriser l'expression $a^{n-1}-1$, en remplaçant $n-1$ par $2^s t$; le petit théorème de Fermat montre alors que n divise ce produit, et si n est premier il doit donc diviser l'un des facteurs, ce qui est exactement la conclusion du lemme. □

On dit qu'un entier n est pseudo-premier fort en base a si n vérifie l'une des deux conclusions du lemme tout en étant composé. On dit alors que a est un faux témoin pour n. Tester si a est un faux témoin pour un n donné peut se faire en temps polynomial. En effet, il suffit de calculer a^t par l'algorithme suivant (méthode binaire droite-gauche), en temps $O(\log t(\log a)^2)$ (la multiplication de deux nombres de taille m peut se faire en temps $O(m^2)$; on peut même descendre à $O(m \log m \log\log m)$) :
- $z \leftarrow 1$; $y \leftarrow a$.
- **Tant que** $t \neq 0$
  - Si t impair $z \leftarrow zy$
  - $y \leftarrow y^2$; $t \leftarrow \lfloor t/2 \rfloor$.
- Renvoyer z.

[1] mise à prix à un million de dollars.

On laisse au lecteur le soin de prouver que cet algorithme est correct.
Les puissances $a^{2^k t}$ sont ensuite calculés par carrés successifs.

L'efficacité de ce test simple est démontrée par la proposition suivante, due à Monier :

Proposition. Si n est composé, il y a au plus $n/4$ faux témoins pour n dans l'intervalle $[1, n-1]$.

Il s'ensuit qu'un test effectué avec un a aléatoire indique que n est premier avec une probabilité de 3/4. Il est très délicat de donner un sens à une suite de tests, mais on peut montrer, sous des conditions techniques, que si n survit à k tests consécutifs, alors il est premier avec probabilité $1 - (1/4)^k$. Le lecteur aura compris que l'algorithme présenté ici est de type Monte-Carlo.

Citons enfin le théorème suivant, qui montre comment transformer cet algorithme en un algorithme déterministe polynomial si l'hypothèse de Riemann généralisée est vraie :

Théorème. Si l'hypothèse de Riemann généralisée est vraie, il existe une constante c telle que si n est pseudo-premier fort en base a pour $a \leq c \log^2 n$, alors n est premier.

4. Sélection du k-ième élément d'une liste

On présente dans cette partie un algorithme de recherche du k-ième élément d'une liste basé sur une technique d'échantillonage. Une idée analogue peut être développée par exemple pour le problème de la recherche de la distance minimale entre deux points d'un nuage.

L'idée de l'algorithme est la suivante : extraire un sous-ensemble aléatoire de petite taille (échantillon) de l'ensemble S initial, en étudier la structure (en l'occurrence le trier), et en déduire un petit sous-ensemble de S dans lequel se trouve la réponse au problème initial. On peut alors trier ce petit sous-ensemble pour obtenir la réponse. Le but est, bien entendu, d'éviter d'avoir à trier S tout entier, opération dont le coût ne serait pas linéaire en la taille de S.

On donne l'algorithme sous forme brute, et on le commentera ensuite.

(1) Choisir un ensemble R de S en tirant $n^{3/4}$ éléments de ce dernier, au hasard et avec remise.
(2) Trier R. On pose $x = kn^{-1/4}$, $l = \max(\lfloor x - \sqrt{n} \rfloor, 1)$, $h = \min(\lfloor x + \sqrt{n} \rfloor, n^{3/4})$.
(3) Déterminer a et b, les l et h-ièmes éléments de R. Parcourir S pour déterminer le rang de a et b comme éléments de S.
(4) Construire P selon

$$P = \begin{cases} \{y \in S, y \leq a\} & \text{si } k < n^{1/4} \\ \{y \in S, a \leq y \leq b\} & \text{si } n^{1/4} \leq k < n - n^{1/4} \\ \{y \in S, y \geq b\} & \text{si } k \geq n - n^{1/4} \end{cases}$$

Si P est de cardinal supérieur à $4n^{3/4}+2$, retourner en 1, et de même si P ne contient pas le k-ième élément de S.

(5) Trier P et en déduire le k-ième élément de S.

Signalons que tous les calculs effectués ici sont de complexité $O(n^{3/4}\log n)$, à l'exception de l'évaluation des rangs de a et b dans S, de complexité $2n$.

Expliquons un peu le fonctionnement de l'algorithme. On tire à la première étape un ensemble R de cardinal $n^{3/4}$, « représentatif » de S. L'hypothèse « avec remise » n'est ici que pour simplifier l'analyse. Cela implique que R a une « densité » $n^{-1/4}$ dans S. Par suite, on peut s'attendre que le k-ième élément de S ait un rang de l'ordre de $kn^{-1/4}$ dans R. On considère alors un intervalle de largeur au plus $2\sqrt{n}$ de R encadrant cette valeur. Il doit correspondre à l'échantillonage d'un intervalle de S de taille environ $2n^{3/4}$, qui est désigné par P dans l'algorithme (on ne s'intéresse dans cette explication qu'au second cas de la définition de P ; les deux autres cas sont analogues). À ce stade, on vérifie que les hypothèses heuristiques ci-dessus sont valides, à savoir que P contient bien le k-ième élément et qu'il n'est pas trop gros. La preuve ci-dessous montrera que c'est bien le cas avec probabilité $1-O(n^{-1/4})$. On peut ensuite trier P, et en déterminer le $k-r(A)+1$-ième élément.

L'étude de la complexité de cet algorithme se fait via le théorème suivant :

THÉORÈME 7.3. *Le nombre de passages par la boucle* 1–4 *est* 1 *avec probabilité* $1-O(n^{-1/4})$.

DÉMONSTRATION. On se limite là encore au second cas de la définition de P.

Il peut y avoir deux sources d'échec, qu'il faudra étudier séparément : d'abord, le k-ième élément peut ne pas appartenir à P (soit $S\cap[a,b]$ dans notre cas). Sinon, si ce k-ième élément est bien dans P, il peut advenir que P soit trop gros.

Définissons une suite de variables aléatoires X_i par $X_i=1$ si le i-ème élément de R est au moins égal à l'élément cherché S_k, $X_i=0$ sinon, et posons $X=\sum_{1\le i\le n^{3/4}} X_i$. Comme l'espérance de X_i est manifestement k/n, il s'ensuit que $E[X]=kn^{-1/4}$. De même la variance de X_i étant $(1-k/n)k/n$, et les X_i étant indépendantes, la variance de X vaut $n^{3/4}(1-k/n)(k/n)\le n^{3/4}/4$.

On peut alors utiliser l'inégalité de Tchebitchev, qui dit que $P(|X-E[X]|>\varepsilon)\le \mathrm{var}(X)/\varepsilon^2$. Dans le cas présent, pour $\varepsilon=\sqrt{n}$, il vient

$$P(|X-kn^{-1/4}|>\sqrt{n})\le n^{-1/4}/4,$$

ce qui revient exactement à l'assertion initiale. Notons que X est exactement la variable aléatoire désignant le rang de S_k dans R.

Soit maintenant $k_l=\max(1,k-2n^{3/4})$ et $k_h=\min(n,k+2n^{3/4})$. Si P soit trop grand, alors soit a est plus petit que le k_l-ième élément de S, ou b est plus grand que le k_h-ième élément de S (rappelons que l'on peut supposer à ce stade que le k-ième élément de S est dans P). On traite seulement le cas de a, l'autre cas étant analogue. Définissons alors la suite Y_i par $Y_i=1$ si le i-ème tirage est au plus le k_l-ième élément de S, 0 sinon. Alors $E[Y_i] = k_l/n$, $var(Y_i) = (1 - k_l/n)k_l/n$, et posant $Y = \sum Y_i$, on a $E[Y] = k_l n^{-1/4}, var(Y) = n^{3/4}(1 - k_l/n)(k_l/n) \leq n^{3/4}/4$. La probabilité que a soit plus petit que le k_l-ième élément de S est la probabilité que Y soit plus petit que l, i.e., soit plus petit que $kn^{-1/4} - \sqrt{n} = n^{-1/4}k_l - 3\sqrt{n}$ (on peut trivialement écarter le cas $k_l = 1$). L'inégalité de Tchebitchev montre que cette probabilité est encore $O(n^{-1/4})$. □

5. Vérification d'identités polynomiales

Supposons donnés deux polynômes P et Q, sous deux formes différentes (par exemple, l'un sous forme développée usuelle, l'autre sous forme factorisée ou sous forme de déterminant), et supposons que nous voulions tester l'égalité de ces deux polynômes. Dans le cas de deux polynômes en une variable, il suffit pour ce faire de majorer leurs degrés par M et de les évaluer en $M + 1$ points. Dans le cas de plusieurs variables, on ne peut transposer l'algorithme tel quel : si $P(X, Y) = X^2 - Y^2$ et $Q(X, Y) = X - Y$, alors pour tout point (x, x), $P(x, x) = Q(x, x) = 0$, tandis que $Q \neq P$. En revanche, un choix *aléatoire* de points (x, y) permet de décider avec une probabilité d'erreur faible si $P = Q$. L'algorithme consiste donc à tirer k points, à comparer $P(x_i, y_i)$ et $Q(x_i, y_i)$ pour tout i et à déclarer les deux polynômes égaux s'il y a égalité pour tout i. Reste à analyser la probabilité d'erreur (qui ne peut intervenir que dans le cas où $P \neq Q$).

THÉORÈME 7.4. *Soit $Q \in \mathbb{K}[X_1, \ldots, X_n]$ un polynôme de degré total d. Pour tout ensemble fini E de cardinal k de $\mathbb{K}$, et pour $(x_1, \ldots, x_n)$ choisi uniformément dans E^n, on a*

$$\Pr\left(Q(x_1, \ldots, x_n) = 0 | Q \neq 0\right) \leq \frac{d}{k}.$$

DÉMONSTRATION. Par récurrence sur n. Le cas $n = 1$ revient à dire qu'un polynôme de degré d a au plus d racines. Supposons le résultat vrai pour les polynômes en $n - 1$ variables. On écrit

$$Q(x_1, \ldots, x_n) = \sum_{l=0}^{j} Q_l(x_1, \ldots, x_{n-1}) x_n^l.$$

Si l'on prend pour j le degré en x_n de Q, et si $Q \neq 0$, on a $Q_j(x_1, \ldots, x_{n-1}) \neq 0$. Par l'hypothèse de récurrence, on a

$$\Pr(Q_j(r_1, \ldots, r_{n-1}) = 0) \leq \frac{d - j}{k}.$$

Supposons maintenant que $Q_j(r_1, \ldots, r_{n-1}) \neq 0$. Alors on peut former le polynôme $q(x) = Q(r_1, \ldots, r_{n-1}, x_n)$, polynôme de degré j en x_n. Il s'ensuit que la probabilité que $q(r_n) = 0$ vaut au plus j/k.

Comme

$$\Pr(Q(r_1, \ldots, r_n) = 0) \leq \Pr(Q(r_1, \ldots, r_n) = 0 | Q_k(r_2, \ldots, r_n) \neq 0) + \Pr(Q_k(r_2, \ldots, r_n) \neq 0),$$

le résultat s'ensuit par récurrence. □

Il convient de signaler une difficulté dans la mise en œuvre de cette méthode. Dans le cas où $\mathbb{K} = \mathbb{Q}$, par exemple, l'évaluation des polynômes peut mettre en jeu des calculs très lourds. On peut à nouveau probabiliser cette évaluation en la faisant modulo plusieurs petits nombres premiers aléatoires et en comparant les résultats.

On laisse l'analyse de cette variante au lecteur.

6. Routage dans un hypercube

On considère dans ce paragraphe le problème du routage de données dans un réseau de machines. Le problème est le suivant : N machines $\{m_1, \ldots, m_n\}$ sont connectées à un réseau dont la topologie est décrite par la donnée des arêtes (m_i, m_j) correspondant aux liaisons existantes. À l'instant initial, chaque machine m_i dispose d'un message à envoyer à la machine $m_{f(i)}$. Durant chaque étape, un message peut être acheminé par une liaison donnée. L'objectif est de construire un algorithme de routage par lequel chaque message arrive en un temps minimum à sa destination.

Dans le cas où le graphe est complet, il est bien clair que l'acheminement des messages peut se faire en une seule étape, mais cela suppose une infrastructure importante (N^2 liaisons...). Dans toute autre configuration, il est susceptible de se produire des « bouchons », c'est-à-dire que plusieurs messages passent par le nœud m_i à un moment donné.

On se limitera dans cette partie au *routage inconscient*, dans lequel le chemin suivi par le message M_i ne dépend que de i et de la destination $f(i)$, sans tenir compte des destinations des messages émis par les voisins. On suppose aussi que f est une permutation, c'est-à-dire qu'il n'y a pas deux messages émis pour le même destinataire. On peut montrer dans ce cadre le théorème suivant :

THÉORÈME 7.5. *Tout algorithme déterministe inconscient pour ce problème nécessite au moins $c\sqrt{N/d}$ étapes dans le cas le pire, où c est une constante absolue, dans le cas où tous les nœuds du graphe sont de degré d.*

Considérons en particulier le cas de l'hypercube. Dans ce cas, il y a $N = 2^n$ sommets et il y a une arête du graphe entre i et j si i et j diffèrent d'un chiffre dans leur représentation binaire. En particulier, chaque nœud est de degré n ; le théorème affirme donc qu'il faut $c\sqrt{2^n/n}$ étapes dans le cas le pire.

Cette situation contraste avec la situation probabiliste :

THÉORÈME 7.6. *Dans le cas de l'hypercube, il existe un algorithme probabiliste inconscient utilisant $O(nN)$ bits aléatoires tel qu'étant donné θ, avec probabilité supérieure à $1 - 2/N^\theta$, chaque message atteint sa destination en au plus $(2\theta + 4)n$ étapes.*

DÉMONSTRATION. On commence par décrire l'algorithme. La façon naturelle d'acheminer un message M de i vers j consiste à écrire les représentations binaires de i et de j ; on les lit ensuite de gauche à droite et on modifie à chaque étape le bit de i le plus à gauche qui diffère du bit correspondant de j pour obtenir le sommet suivant du parcours. Ainsi, le chemin de $12 = 01100$ à $31 = 11111$ passera par $28 = 11100$ et $30 = 11110$.

L'algorithme probabiliste que nous allons utiliser est alors le suivant : pour chaque sommet i, tirer un entier aléatoire de $[1, N]$, noté $t(i)$. Acheminer le message de i à $t(i)$ par la stratégie déterministe ci-dessus, puis le message de $t(i)$ à $f(i)$ par la stratégie déterministe ci-dessus. Noter que t n'est a priori pas une permutation.

Le but du passage par $t(i)$ est de rendre les différents chemins « plus indépendants » pour éliminer les goulots d'étranglement pouvant être créés par une instance du problème ; pour la stratégie déterministe ci-dessus, on pourra ainsi réfléchir à ce qui se passe si la destination du sommet $i = u2^{n/2} + v$ est $j = v2^{n/2} + u$.

Il faut aussi spécifier dans quel ordre nos messages bloqués à un sommet quittent ce dernier ; on adopte une structure de file, c'est-à-dire que le premier message arrivé est aussi le premier à partir.

Pour estimer le nombre d'étapes nécessaire, commençons par estimer le nombre d'étapes pour acheminer chaque message jusqu'en $t(i)$. Ce nombre est au plus n plus le nombre d'étapes passées à attendre à un sommet du graphe qu'une arête se libère.

LEMME 7.1. *Considérons un message M dont le chemin est $a_1, \ldots, a_k$. Soit S l'ensemble des messages dont le trajet passe par au moins une de ces arêtes. La somme des temps d'attente subis par M aux sommets est alors au plus $|S|$.*

DÉMONSTRATION. Notons que la stratégie déterministe fait qu'une fois que deux routes se séparent, elles ne se rencontrent plus ultérieurement. Chaque message quitte donc cette suite d'arêtes à une étape a_m et ne repasse plus par une de ces arêtes ensuite.

L'idée de la preuve du lemme consiste à dire que quand le retard de M augmente, c'est qu'il est en concurrence avec un message M' pour traverser une des arêtes $a_1, \ldots, a_k$, et donc que M' est dans S ; on peut donc associer à chaque message de S une unité de retard de M, il faut maintenant le faire de façon à ce que la correspondance soit injective.

Définissons le retard d'un message avant de traverser l'arête a_l comme le nombre d'étapes écoulées depuis le début de l'algorithme moins l. On va montrer que le retard de notre message en sortie de a_k est au plus $|S|$.

Notons d'abord que quand le retard de M passe de r à $r+1$, cela implique que le message est en attente pour franchir l'arête a_l, et donc c'est qu'un message (qui est nécessairement dans S) est en train de traverser l'arête a_l, avec un retard qui est donc égal à r.

Soit t l'étape maximale à laquelle il existe un message de S de retard r donné (noter que t est borné car tous les messages arrivent (heureusement) à leur destination) ; il existe donc un message prêt à franchir l'arête $t - r$. Ce message ne peut suivre l'arête $t - r + 1$ à l'étape suivante, et doit donc quitter à ce stade le chemin $a_1, \ldots, a_k$; sans quoi, soit il n'est pas bloqué et son retard à cette arête est à nouveau r ; soit il est bloqué par un message quittant a_{t-r+1} à l'instant $t+1$, dont le retard est donc r. Ces deux possibilités contredisent la maximalité de t. À chaque unité de retard, on peut donc associer un message dont la route rencontre $a_1, \ldots, a_k$, de façon injective. Le retard du message M est donc au plus $|S|$, ce qui conclut la preuve du lemme. □

Considérons la variable aléatoire H_{ij} qui vaut 1 si les routes de i à $t(i)$ et de j à $t(j)$ se rencontrent. Le paragraphe précédent montre que le retard de i est $\sum_{j=1}^{N} H_{ij}$. Les routes étant choisies indépendamment et aléatoirement (le choix de la destination peut être vu comme le choix de la route), on va pouvoir appliquer la borne de Chernov à $\sum_{j=1}^{N} H_{ij}$.

Notons maintenant que $\sum_{j=1}^{N} H_{ij} \leq \sum_{l=1}^{k} R(a_l)$, où $R(a_l)$ est le nombre de routes contenant l'arête a_l. Par symétrie de l'hypercube, $E[R(a_l)]$ est indépendant de l'arête. La longueur moyenne d'une route est $n/2$, le nombre de routes est N, donc $\sum_{l=1}^{Nn} E[R(a_l)] = Nn/2$ (noter que l'hypercube a Nn arêtes) ; il s'ensuit que $E[R(a_l)] = 1/2$, et donc que $E[\sum_{j=1}^{N} H_{ij}] \leq k/2 \leq n/2$.

L'application de la borne de Chernov à $\sum_{j=1}^{N} H_{ij}$ montre alors que la probabilité que cette variable dépasse $(\theta + 1)n$ est au plus $2^{-(\theta+1)n}$, et donc la probabilité qu'un des 2^n messages ait un retard dépassant $(\theta + 1)n$ est au plus $2^{-\theta n}$. Par suite, avec probabilité inférieure à $1/N^\theta$, un message atteint sa destination temporaire en $(\theta + 2)n$ étapes au plus (rajouter la longueur de la route au retard). Par suite, avec probabilité au plus $2/N^\theta$, chaque message atteint sa destination finale en $(2\theta + 4)n$ au plus. $\square$

Noter que pour être rigoureux, il faut bloquer les paquets à leur destination temporaire jusqu'à ce que tous les autres paquets aient atteint leur destination temporaire (donc jusqu'à ce que $(\theta+2)n$ étapes se soient écoulées), sans quoi l'interférence possible entre les deux phases peut rendre la preuve ci-dessus incorrecte.

7. Hachage parfait

L'un des rôles principaux des fonctions de hachage consiste à offrir la possibilité de décider très rapidement de l'appartenance d'un élément donné à un ensemble E. Pour ce faire, on construit un tableau de booléens T, et pour chaque élément x de E, on met la case $h(x)$ de T à `vrai`. Moyennant ce précalcul, on peut décider si x est dans E en consultant simplement $T[h(x)]$.

Ceci étant dit, on peut constater que les deux objectifs implicitement décrits ci-dessus sont contradictoires. On voudrait d'une part avoir un tableau le plus petit possible, ce qui implique que $h(x)$ ne prenne que de petites valeurs entières ; on voudrait par ailleurs que h soit injective ce qui entraîne automatiquement que l'ensemble des valeurs de h soit plus grand que son espace de définition.

On dira qu'une fonction de hachage h est parfaite pour un sous-ensemble S de son domaine de définition si la restriction de h à S est injective. L'objet de ce paragraphe est de présenter une méthode probabiliste pour construire une telle fonction de hachage parfaite. On peut aussi introduire la notion de hachage quasi-parfait, pour lequel le nombre d'antécédents d'une valeur donnée est borné par une constante indépendante de $|S|$. Dans ce cas, on peut à chaque case du tableau T associer un tableau de booléens, qui correspondent à la valeur d'une seconde fonction de hachage, censée discriminer entre les « collisions » de h (ou encore, on stocke dans la ℓ-ième case de T tous les $y \in E$ tels que $h(y) = \ell$.

On va montrer ici comment résoudre le problème du hachage parfait. Comme décrit ci-dessus, cette solution – à la fois économique en temps et en espace – utilise deux niveaux de hachage. Soit $M = \{0, \ldots, m-1\}$ l'ensemble duquel proviennent les éléments à hacher, et soit $V \subset M$ de cardinal v l'ensemble à hacher. Quitte à augmenter[2] m, on peut supposer que $p = m + 1$ est un nombre premier. On fixe un entier $r \geq v$, et on introduit la fonction

[2] on peut montrer que cette augmentation est d'un facteur linéaire ≤ 2 au plus.

$$h_k(x) = (kx \bmod p) \bmod r,$$

où par abus de notation $x \bmod y$ désigne le reste de la division euclidienne de x par y.

Il nous faut maintenant compter le nombre de collisions.

LEMME 7.2. *Le nombre de triplets (k, x, y) de $[1, p-1] \times [0, m-1]^2$ tels que $x \neq y$ et $h_k(x) = h_k(y)$ est au plus $mv(v-1)/r$.*

DÉMONSTRATION. L'identité $h_k(x) = h_k(y)$ équivaut au fait que r divise $(kx \bmod p) - (ky \bmod p)$. En raison de notre abus de notation, cette dernière quantité ne vaut pas toujours $k(x-y) \bmod p$, mais peut être son opposé.

On sait donc que r divise $\pm(k(x-y) \bmod p)$. Pour chacun des $2\lfloor (p-1)/r \rfloor$ multiples non nuls de r de $[1-p, p-1]$, si x et y sont fixés, on peut trouver un seul et unique k tel que $k(x-y) \bmod p$ ait la valeur prescrite. Le nombre de couples (x, y) avec $x \neq y$ étant $v(v-1)/2$, le nombre total de triplets (k, x, y) est donc $\leq (p-1)v(v-1)/r = mv(v-1)/r$. □

Le corollaire suivant est immédiat (c'est l'inégalité de Markov déguisée) :

COROLLAIRE 7.1. *Pour au moins la moitié des k de $[1, m-1]$, le nombre de couples (x, y) tels que $h_k(x) = h_k(y)$ est au plus égal à $2v(v-1)/r$.*

On peut maintenant construire notre schéma de hachage en deux temps. On choisit tout d'abord un k tel que dans le corollaire précédent, en tirant au hasard et en dénombrant le couple (x, y) correspondants. Deux étapes en moyenne seront nécessaires pour obtenir un k valide. Pour chaque i de $[0, r-1]$, on choisit alors une fonction $h_{k_i} = (k_i x \bmod p) \bmod r_i$, où $r_i \geq 2\#\{x/h_k(x) = i\}^2$, pour hacher à nouveau l'ensemble des x tels que $h_k(x) = i$. Le corollaire précédent montre que pour la moitié des choix de k_i, le nombre de couples (u, v) tels que $h_{k_i}(u) = h_{k_i}(v)$ est nul, et donc le second hachage est parfait.

THÉORÈME 7.7. *Le procédé de hachage ci-dessus hache en temps $O(1)$ et en espace $O(r)$.*

DÉMONSTRATION. Le hachage d'un élément consiste à évaluer deux fonctions h_ℓ, ce qui se fait en temps $O(1)$. Reste à estimer l'espace occupé. La table de hachage primaire occupe un espace $O(s)$; posons $s_i = \#\{x/h_k(x) = i\}$. pour l'ensemble des tables de hachage secondaire, il nous faut estimer $\sum_{i=0}^{r-1} s_i^2$.

Cette quantité est reliée au nombre de collisions : une collision (x, y) est un couple avec $x \neq y$ et $h_k(x) = i$, $h_k(y) = i$ pour un certain i. Le nombre total de collisions est donc $\sum_{i=0}^{r-1} s_i(s_i - 1)/2 = \sum_{i=0}^{r-1} s_i^2/2 - \sum_{i=0}^{r-1} s_i/2$. La seconde somme vaut au plus le nombre total de x de $[0, r-1]$, soit r, et le nombre total de collisions est, en vertu du corollaire dans le cas $v = r$, au plus $2(r-1)$. Il s'ensuit que la taille totale des tables de hachage secondaire est au plus $6r - 4$, soit $O(r)$. Le stockage des clés k_i est aussi en espace $O(r)$, ce qui conclut la preuve. □

Notons que l'argument présenté ici est probabiliste dans l'esprit à défaut de l'être dans la forme : on estime la moyenne du nombre de collisions, et on utilise cette estimation pour montrer qu'un choix aléatoire des k_i conduit en peu d'étapes à un choix correct. Les clés de hachage k_i peuvent être construites et réactualisées « à la volée », c'est-à-dire que si l'on s'aperçoit d'une collision en route, on essaie un nouveau k_i, et on re-hache la table secondaire. Cela se produit en principe au plus une fois en moyenne, et n'augmente donc la complexité que d'un facteur linéaire.

8. MAXSAT

On rappelle le cadre du problème MAXSAT. On se donne un certain nombre n de variables $x_1, \ldots, x_n$, et m clauses de la forme $y_{l,1} \vee y_{l,2} \vee \cdots \vee y_{l,k_l}$, où chacun des $y_{l,j}$ est soit x_t, soit $\overline{x}_t$. On suppose de plus qu'une variable et son complément n'interviennent pas dans la même clause (sans quoi celle-ci est automatiquement vraie). On cherche à déterminer le plus grand nombre de clauses pouvant être satisfaites par une affectation de variables.

On va présenter deux algorithmes probabilistes, l'un se comportant particulièrement bien quand le nombre de variables par clauses est élevé, l'autre se comportant bien quand ce nombre est faible. On prouvera ensuite que l'un des deux algorithmes donne toujours, en moyenne, un résultat qui est au moins les $3/4$ du nombre maximal de clauses pouvant être valides. En d'autres termes, on construit un algorithme probabiliste $(1/4)$-approché pour MAXSAT.

8.1. Premier algorithme. Le premier algorithme suit l'idée la plus naïve qui soit. Pour chaque variable, on choisit une valeur de vérité 0 ou 1 au hasard, avec probabilité $1/2$, et l'on regarde le nombre de clauses satisfaites. Il est bien clair que si l'on prend une clause à k littéraux, elle sera alors satisfaite si et seulement si l'une des variables (ou son complémentaire éventuellement) est vraie, et donc avec probabilité $1-2^{-k}$. Si pour chaque clause on définit alors une variable aléatoire C_k par $C_k = 1$ si la clause k est satisfaite, on a $E[C_k] = 1 - 2^{-k}$. Comme le nombre de clauses satisfaites est $E\left[\sum_{j=1}^m C_j\right] \geq m/2$, on a une approximation à un facteur $1/2$ près.

Notons que le problème de cet algorithme réside dans les clauses à 1 élément. En l'absence de ces dernières on a déjà un algorithme donnant au moins une proportion $3/4$ du vrai résultat.

8.2. Second algorithme. Le second algorithme repose sur des idées plus raffinées. On traduit le problème MAXSAT en un problème linéaire en nombres entiers en introduisant une variable entière x_i par variable, et une variable y_j par clause. Le problème MAXSAT peut alors s'écrire

$$\begin{cases} \max \sum_{j=1}^m z_j \\ 0 \leq y_i \leq 1 \\ 0 \leq z_j \leq 1 \\ \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} \geq z_j \end{cases},$$

où les variables sont de plus contraintes à être entières.

Dans la dernière contrainte, C_j^+ est l'indice des variables apparaissant dans la clause j, et C_j^- l'indice des variables dont le complémentaire apparaît dans la clause j. Cette dernière contrainte traduit que si $z_j = 1$, alors soit un des y_i de C_j^+ vaut 1, soit un des y_i de C_j^- vaut 0, donc que la clause j est effectivement satisfaite.

On oublie alors la contrainte d'intégralité, et l'on résout le système par la méthode classique du simplexe (ou tout autre méthode type ellipsoïde). On obtient alors des solutions de $[0, 1]$ comme valeur pour les y_i et z_j, que l'on notera respectivement $\hat{y}_i$ et $\hat{z}_j$.

On va alors arrondir les $\hat{y}_i$ pour obtenir les valeurs à affecter aux variables, en choisissant d'arrondir $\hat{y}_i$ vers 0 avec probabilité $1 - \hat{y}_i$. Cette technique est une probabilisation extrêmement fréquente pour les problèmes de programmation linéaire en nombres entiers : supprimer les contraintes d'intégralités, et arrondir les résultats du programme linéaire en utilisant les parties fractionnaires pour définir les probabilités des deux entiers encadrant la valeur réelle.

Considérons la j-ième clause. On a $\hat{y}_1 + \cdots + \hat{y}_k \geq \hat{z}_j$ (en supposant, sans perte de généralité, que les variables de cette clause sont non complémentées). La probabilité que cette clause soit fausse est la probabilité que toutes les variables soient arrondies vers 0. Cette probabilité est $\prod_{i=1}^k (1-\hat{y}_i)$. La clause est donc satisfaite avec probabilité $1-\prod_{i=1}^k (1-\hat{y}_i)$. Or cette dernière quantité est minimale, sous la contrainte que la somme des $\hat{y}_i$ est au moins $\hat{z}_j$, quand tous les $\hat{y}_i$ sont égaux à $\hat{z}_j/k$, et vaut alors

$$1 - \left(1 - \frac{\hat{z}_j}{k}\right)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \hat{z}_k \geq (1 - 1/e)\hat{z}_k.$$

Le nombre moyen de clauses satisfaites est donc $(1 - 1/e) \sum \hat{z}_k$. Noter que la somme est un majorant pour le nombre de clauses satisfaites (c'est le maximum de la somme sur tous les points réels du simplexe, au lieu des points entiers).

Noter aussi que cette méthode fonctionne moins bien pour les grandes valeurs de k, mais très bien pour les petites valeurs.

8.3. Combinaison des deux algorithmes. Soit n_1 le nombre moyen de clauses satisfaites dans le premier algorithme, et n_2 ce nombre pour le second algorithme. Si l'on peut prouver que l'on a toujours $(n_1 + n_2)/2 \geq 3/4$, alors l'exécution simultanée des deux algorithmes donne au moins 3/4 du nombre maximal de clauses satisfaites, en moyenne.

On a

$$n_1 \geq \sum_k \sum_{c_j \in S^k} (1 - 2^{-k}) \geq \sum_k \sum_{c_j \in S^k} (1 - 2^{-k})\hat{z}_j,$$

où l'on a séparé les clauses selon leur nombre de variables.

Par ailleurs

$$n_2 \geq \sum_k \sum_{c_j \in S^k} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \hat{z}_j.$$

Il vient

$$(n_1 + n_2)/2 \geq \sum_k \sum_{c_j \in S^k} \frac{\hat{z}_j}{2} \left(2 - 2^{-k} - \left(1 - \frac{1}{k}\right)^k\right),$$

et il n'est pas difficile de montrer que le terme intérieur à la parenthèse est au moins égal à $3/2$ pour tout k ce qui conclut la preuve.

Il faut noter que le résultat d'approximation n'est valide qu'en moyenne. Ce problème peut en pratique – moyennant une analyse probabiliste plus précise – être contourné en itérant plusieurs fois chacun des deux algorithmes, et en conservant la plus grande de toutes les valeurs.

8.4. Dérandomisation. Il est bon de dire un mot à ce niveau de la notion de *dérandomisation*. On peut voir un algorithme probabiliste pour un problème de **NP** comme un chemin dans l'arbre de calcul de la machine non-déterministe correspondante, chaque bit aléatoire correspondant à un choix non-déterministe. Dire que l'on dispose d'un algorithme polynomial probabiliste pour résoudre un problème revient à estimer la probabilité de tomber sur une feuille de l'arbre de calcul apportant une réponse, ce qui, si tous les choix sont équiprobables, revient à estimer le nombre de telles feuilles de l'arbre de calcul. C'est en particulier ainsi que l'on définit les classes de complexité probabilistes, voir le complément.

Notre propos dans ce paragraphe est de montrer comment l'on peut parfois exploiter cette remarque pour transformer un algorithme probabiliste en algorithme déterministe.

À chaque sommet s de l'arbre de calcul, on peut associer la probabilité $\Pr(s)$ qu'un calcul issu de ce sommet conduise à une réponse valide. Lors de l'utilisation d'un bit aléatoire (ou d'un choix non-déterministe), on peut alors descendre d'un nœud n vers un nœud n_0 ou n_1, et l'on a $\Pr(n) = p_0 \Pr(n_0) + p_1 \Pr(n_1)$, où p_0 et p_1 sont les probabilités que le bit aléatoire prenne la valeur 0 ou la valeur 1. En particulier, l'une des deux probabilités $\Pr(n_0)$ et $\Pr(n_1)$ est au moins aussi grande que $\Pr(n)$. Si l'on est capable, à chaque nœud de l'arbre de calcul, de décider laquelle des deux branches vérifie cette propriété, on peut alors construire un chemin de calcul de la racine à une feuille le long duquel les probabilités croissent ; comme la probabilité de la racine est non nulle (au moins, c'est préférable), la probabilité de la feuille est non nulle. Mais la probabilité d'une feuille ne peut être que 0 (elle ne fournit pas de solution) ou 1 (elle fournit une solution). La seule possibilité est donc que cette probabilité vaille 1, et donc on obtient bien une solution.

On peut illustrer cette méthode dans le cas de MAXSAT. Le premier algorithme probabiliste donné correspond, à chaque étape, à choisir la valeur 0 ou 1 d'une variable donnée. En fonction de ce choix, il est facile, par le même raisonnement que dans l'analyse de l'algorithme, d'estimer la probabilité du sous-arbre correspondant, et de choisir la valeur donnant la probabilité la plus forte. En suivant ce raisonnement, on obtient un algorithme déterministe $(1/2)$-approché pour MAXSAT.

Un exemple de détermination d'une borne inférieure pour la complexité

La question de détermination de bornes inférieures pour la complexité d'un problème est des plus difficiles en algorithmique. Une des techniques utilisées consiste à considérer des *stratégies adverses* , c'est à dire à imaginer qu'une fois votre algorithme mis au point un adversaire trouvera les données les plus génantes qui rendront le temps de calcul le plus élevé possible. On illustre ici cette stratégie de preuve sur un exemple simple.

Le problème suivant est un classique, il s'agit étant donnée une suite de nombres $a_1, a_2, \ldots, a_n$ de trouver le plus petit de ces nombres et le plus grand en effectuant un nombre minimum de comparaisons.

Nous supposons que $n = 2p$, dans le cas $n = 2p + 1$ on utilise l'algorithme pour déterminer le maximum et le minimum parmi les $n - 1$ premiers éléments on compare ensuite le n-ième aux extrema trouvés.

L'algorithme se divise en deux phases ; au cours de la première on compare les nombres deux à deux : a_1 avec a_2, a_3 avec a_4 et ainsi de suite jusqu'à a_{n-1} avec a_n ; après chaque comparaison on met le plus grand des deux nombres dans un ensemble A et le plus petit dans un ensemble B.

On a ainsi fait p comparaisons et on a constitué deux sous-ensembles A et B de p éléments chacun. Pour trouver le maximum on examine tous les éléments de A en les comparant à un maximum partiel, ceci se fait en $p - 1$ comparaisons ; symétriquement on obtient le minimum en parcourant B et en effectuant aussi $p-1$ comparaisons. Au total on aura fait $3p-2$ comparaisons. Si $n = 2p+1$, il faudra comparer le nombre a_n au maximum et eventuellement au minimum ce qui donne $3p$ comparaisons dans le cas le pire.

La question est maintenant de savoir s'il y a un algorithme qui utilise moins de comparaisons. Nous allons démontrer qu'il n'en est rien, pour cela on utilise un argument de stratégie adverse. Après la i-eme comparaison, notons A_i, B_i, C_i et D_i (pour $i = 1, \ldots, p$, où p est le nombre total de comparaisons), les ensembles donnés par :

- A_i contient les éléments qui ont été comparés à au moins un autre élément et qui sont supérieurs à tous les éléments auxquels ils ont été comparés,
- B_i contient les éléments qui ont été comparés à au moins un autre élément et qui sont inférieurs à tous les éléments auxquels ils ont été comparés,
- C_i contient les éléments pour lesquels on a trouvé un élément plus grand et un élément plus petit.

– enfin D_i contient les éléments qui n'ont pas été comparés à un autre élément et qui peuvent donc être maximum ou minimum.

Dans la suite on note par une lettre miniscule x_i le nombre d'éléments de l'ensemble X_i. Il est clair que $a_0 = b_0 = c_0 = 0$ et $d_0 = n$ d'autre part lorsque l'algorithme termine après p comparaisons, on a $a_p = b_p = 1$; $c_p = n - 2$ et $d_p = 0$. Pour mesurer le degré d'avancement de l'algorithme on introduit la fonction $U_i = 2a_i + 2b_i + 3d_i$ qui passe de $3n$ à 4 au cours de l'exécution de l'algorithme ($U_0 = 3n$, $U_p = 4$).

Lorsque qu'une comparaison est effectuée la différence entre U_i et U_{i+1} prend une valeur dans l'ensemble $\{0, 1, 2, 3, 4\}$. La valeur est égale à 4 si on compare un élément x de A_i à un élément y de B_i et que l'on obtient $y > x$ car dans ce cas A_i et B_i perdent un élément. Cette différence est égale à 3 lorsqu'un élément de D_i est trouvé supérieur à un élément de A_i, ou symétriquement s'il est trouvé inférieur à un élement de B_i : en effet l'élement de D_i remplace alors dans A_i l'élément auquel il a été comparé, ce dernier passant dans C_{i+1} et on vérifie que U_i diminue de 3. Dans tous les autres cas la différence est inférieure ou égale à 2.

LEMME 7.3. *Pour tout algorithme donné, il existe une suite* $= a_1, a_2, \ldots, a_n$ *telle que si la* $i+1$*-eme opération consiste à comparer un élément de* D_i *à un élément de* A_i *ou* B_i*, ou un élément de* A_i *à un élément de* B_i *on a alors* $U_i - U_{i+1} \leq 1$.

Preuve On suppose l'algorithme donné et on imagine un *adversaire* tout puissant qui connaît les A_i, B_i, C_i, D_i et qui chosit les nombres à comparer de façon telle que U_i diminue le plus faiblement possible tout en restant consistant avec les comparaisons déjà effectuées. Si une comparaison concerne un élément x de D_i et un élément y de A_i l'adversaire s'arrange pour que l'élément de x soit plus petit que y, ceci est toujours possible puisque x n'a encore été comparé à aucun autre élément, aucune condition de consistance n'est à vérifier ; la propriété est aussi satisfaite par symétrie pour la comparaison d'un élément de D_i et B_i. Lorsque la comparaison concerne un élément de A_i et un élément de B_i, l'adversaire peut augmenter l'élément de A_i sans modifier les résultats précédents, en effet cet élément a été trouvé supérieur à tous les éléments auxquels il a été comparés et le fait de l'augmenter ne modifie pas cette propriété.

THÉORÈME 7.8. *Pour tout algorithme déterminant le plus petit et le plus grand élément d'une suite de taille* n *il existe une telle suite pour laquelle sont effectuées au moins* $\frac{3n-4}{2}$ *comparaisons.*

Preuve On vérifie qu'il existe une suite telle que chaque comparaison fait diminuer la valeur de U_i d'au plus 2, en effet toutes les comparaisons donnent une différence de 0,1 ou 2 sauf le cas d'une comparaison d'un élément de D_i à un de A_i ou B_i ou de la comparaison d'éléments de A_i et B_i ; mais le lemme assure que dans ce cas il existe une suite qui donne une différence de 1. Comme on passe de $3n$ à 4 le nombre de comparaisons est au moins égal à $\frac{3n-4}{2}$.

Troisième partie

Quelques applications

CHAPITRE 8

Graphes planaires

Un graphe est planaire si on peut le dessiner dans le plan sans que ses arêtes se coupent. Les graphes planaires jouent un rôle important tant en théorie que pour les applications. Le problème de la coloration en *quatre couleurs* des pays sur une carte de géographie a été probablement le problème de mathématiques combinatoires le plus célèbre depuis Euler et jusqu'à sa résolution vers 1980. Sa solution a été tout d'abord donnée par un très long calcul sur ordinateur, puis plus récemment par une preuve directe. Ce problème s'exprime très simplement en terme de graphe en construisant le *graphe dual* de la carte de géographie de la façon suivante : à chaque pays on associe un sommet et deux sommets sont reliés par une arête si et seulement si les pays correspondants ont une frontière commune. Il s'agit alors de montrer que pour tout graphe planaire on peut partitionner l'ensemble des sommets en quatre parties, de façon telle que deux sommets reliés par une arête n'appartiennent pas à la même partie.

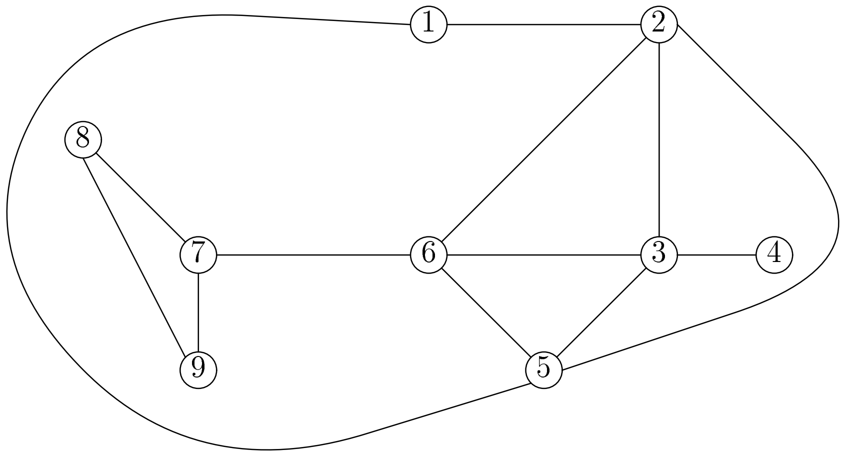


FIG. 1. Un graphe planaire

Les problèmes algorithmiques posés en terme de graphes planaires sont nombreux, un grand nombre provenant des circuits et de leur tracé. On recherche par exemple un tracé minimisant la surface utilisée et satisfaisant des contraintes sur la distance de voisinage entre nœuds (qui doit être supérieure à un seuil donné), ou à minimisant le nombre d'intersections. Un autre domaine qui a vu fleurir des modèles combinatoires basés sur les graphes planaires est la synthèse d'images pour laquelle on se pose des questions de délimitation des

régions, de structuration de données pour la représentation topologique, de la réalisation de diverses fonctions de rendu (incrustation, dégradés).
Dans ce chapitre, après avoir rappelé quelques propriétés classiques des graphes planaires on donne les grandes lignes d'un algorithme de test de planarité d'un graphe, on présente ensuite une structure de données simple qui permet de représenter un graphe dessiné dans le plan et on démontre enfin qu'il est toujours possible de représenter un graphe planaire de façon telle que les arêtes soient des segments de droite.

1. Propriétés élémentaires classiques

Un graphe est planaire s'il peut être tracé dans le plan de façon telle que :
- Les sommets sont représentés par des points
- Chaque arête $\{x, y\}$ est un arc de Jordan dont les extrémités sont les points qui représentent x et y
- Les arcs qui représentent les arêtes ne se coupent pas en dehors de leurs extrémités.

Si le graphe est connexe les domaines délimités par les arêtes sont simplement connexes (toute courbe fermée est homotope à un point), sauf l'un d'entre-eux qui est non-borné. On appelle ces domaines les *faces*, la *face infinie* est le seul domaine non-borné.

Le tracé d'un graphe planaire est donné ci-dessus, il possède 6 faces, la face infinie est limitée par les arêtes $\{1, 2\}, \{2, 5\}, \{1, 5\}$.

On constate qu'un graphe planaire peut avoir plusieurs tracés différents dont certains peuvent ne pas être planaires :

Un des premiers résultats obtenus pour les graphes planaires est la formule classique suivante dite *relation d'Euler* :

PROPOSITION 8.1. *Soit $G = (X, E)$ un graphe planaire connexe à n sommets et m arêtes, tous les tracés planaires de G ont le même nombre f de faces donné par :*

$$n + f = m + 2$$

Preuve : On effectue une preuve par récurrence sur m, si aucune arête de E ne se trouve sur la frontière de deux faces distinctes alors le tracé n'a qu'une face, le graphe est un arbre et $n = m + 1$. S'il existe une arête qui limite deux faces la suppression de cette arête laisse un graphe connexe qui a une arête et une face en moins, par l'hypothèse de récurrence on a :

$$n + (f - 1) = (m - 1) + 2,$$

ce qui est la relation cherchée. $\square$

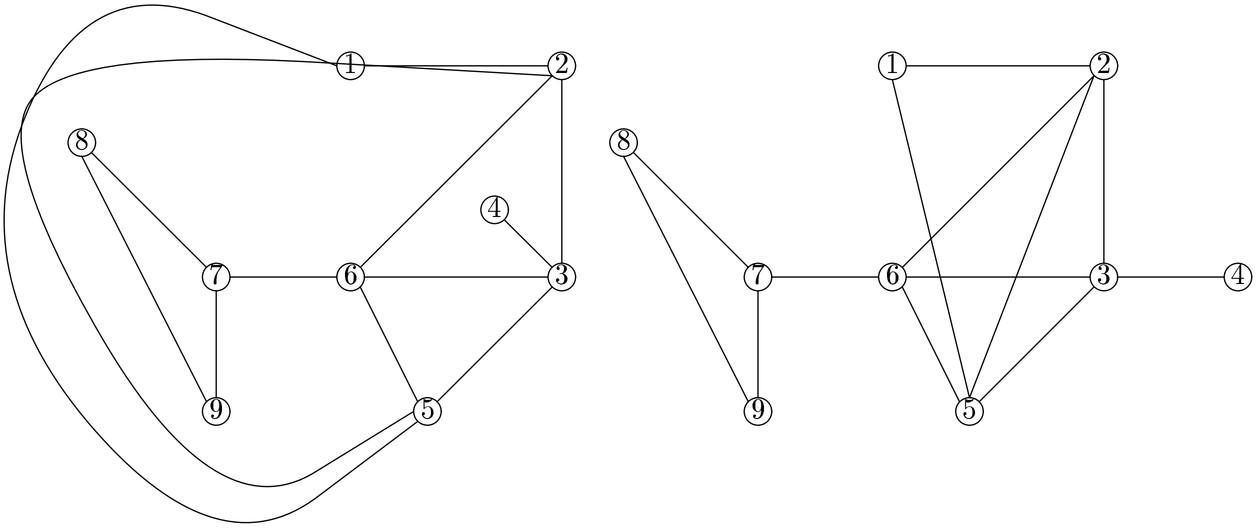


FIG. 2. Plusieurs représentations d'un même graphe planaire

De cette relation on tire :

COROLLAIRE 8.1. *Dans un graphe planaire connexe à n sommets et m arêtes on a :*

$$m \leq 3n - 6$$

Preuve : Une face est limitée par 3 arêtes ou plus et chaque arête se trouve sur la frontière de 1 ou 2 faces ainsi :

$$2m \geq 3f.$$

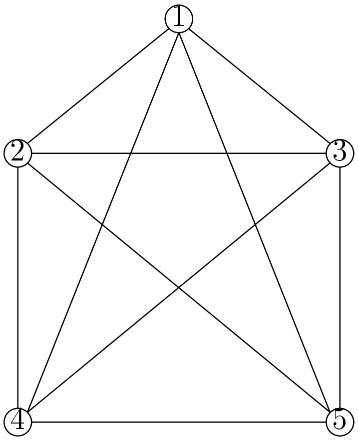
Soit en reportant dans la formule d'Euler :

$$2m \geq 3m + 6 - 3n,$$

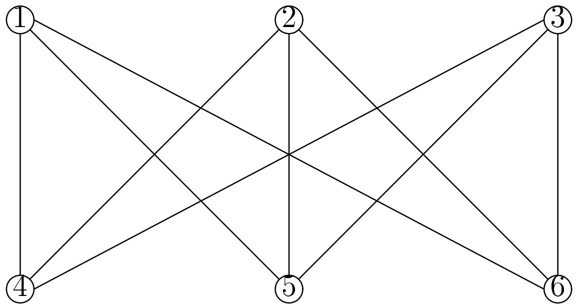
ce qui donne le résultat. □

De ces résultats on déduit :

PROPOSITION 8.2. *Les deux graphes K_5 et $K_{3,3}$ représentés ci-dessous n'admettent pas de tracé planaire :*



K_5



$K_{3,3}$

En effet le premier a 5 sommets et 10 arêtes, il ne satisfait pas les conditions du corollaire, le second ne peut avoir de face de longueur 3 car il ne contient pas de triangle ainsi les faces sont de longueurs ou égales à 4 ce qui donne puisqu'il y a 9 arêtes : $4f \leq 18$ qui est en contradiction avec la relation d'Euler $f + 6 = 11$. $\square$

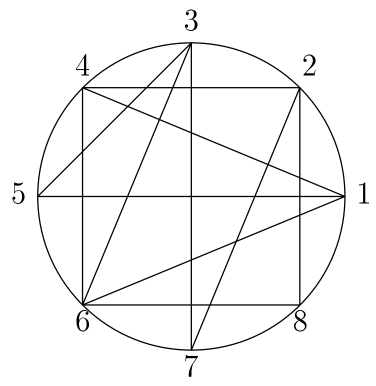
On montre en fait que tout graphe qui n'est pas planaire *contient*, en un sens à préciser, l'un des deux graphes ci-dessus. C'est un théorème difficile dû à Kuratowski qui sort du cadre de cette introduction.

On peut noter aussi que le Corollaire 1 a pour conséquence que tout algorithme en de complexité linéaire suivant le nombre d'arêtes d'un graphe planaire a aussi une complexité linéaire en fonction du nombre de sommets, ce qui n'est pas le cas pour les graphes quelconques.

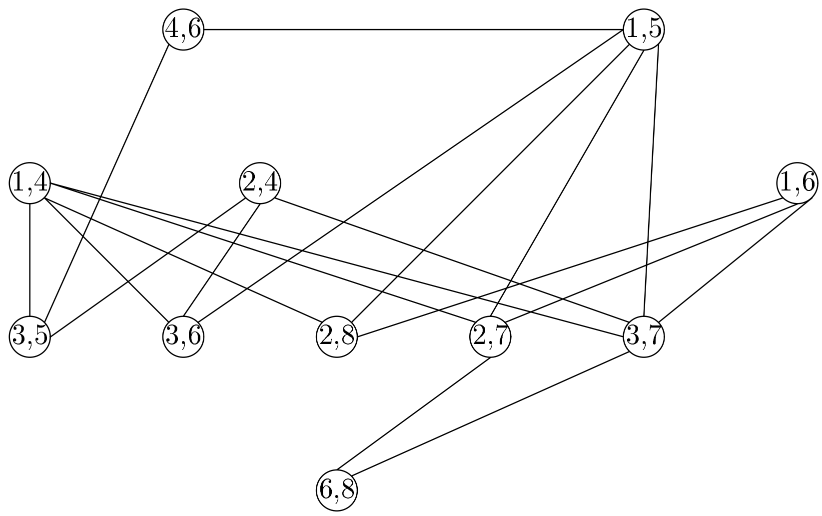
2. Un algorithme de test de planarité

On se pose ici la question suivante étant donné un graphe symétrique donné par un ensemble X de sommets et un ensemble E d'arêtes existe-t-il un tracé planaire de G ? Nous nous proposons de donner les grandes lignes d'un algorithme qui teste la planarité sans entrer dans les détails techniques qui en constituent la difficulté.

2.1. Graphes hamiltoniens. Une bonne introduction au principe de l'algorithme que nous allons étudier consiste à considérer d'abord le cas où l'on connaît un cycle hamiltonien du graphe dont on veut tester la planarité. Un exemple est donné sur la figure ci-dessous :

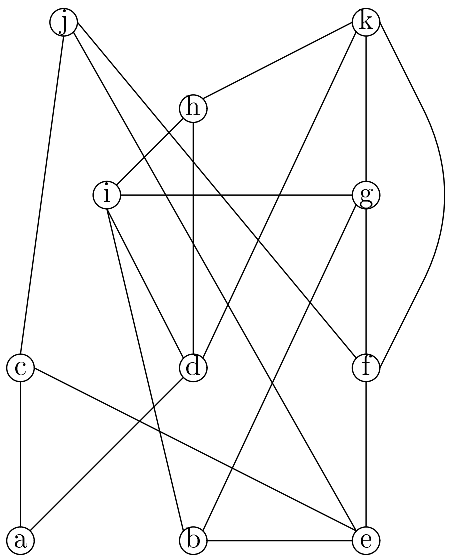


Pour un tel graphe G on peut ramener la recherche de la planarité G à celle de la bicoloration du graphe des cordes $C(G)$ associé à G. On obtient le graphe des cordes de la façon suivante, on trace le cycle hamiltonien de G sous forme d'un cercle dans le plan et on considère les arêtes de G qui ne sont pas sur le cycle hamiltonien comme des cordes du cercle. Le graphe $C(G)$ a comme ensemble de sommets les cordes de G et chaque arête de $C(G)$ correspond à deux cordes qui se coupent dans G. Pour le graphe ci-dessus on obtient :

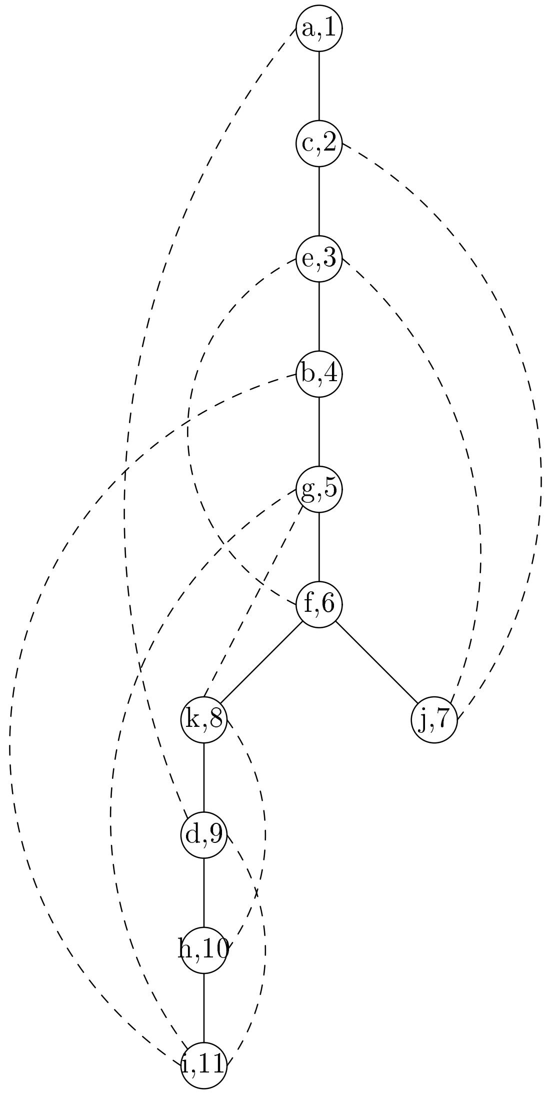


On cherche alors à dessiner G en plaçant certaines cordes à l'intérieur du cercle et d'autres à l'extérieur, ce qui revient à trouver une coloration des sommets de $C(G)$ en deux couleurs. L'algorithme est alors assez simple à réaliser : on construit $C(G)$ et c'est là l'étape la plus coûteuse (car il faut examiner tous les couples de cordes pour vérifier si elles se coupent) ; on applique ensuite un algorithme de bicoloration que l'on peut déduire simplement d'un algorithme de parcours en coloriant les sommets rencontrés. Pour un graphe planaire ayant n sommets on sait que le nombre d'arêtes est inférieur ou égal à $3n-6$ ainsi le nombre de cordes est inférieur ou égal à $2n-6$ et la complexité de construction de $C(G)$ est en $O(n^2)$, par contre celle de sa bicoloration est en $O(n)$. Tarjan a mis au point un algorithme qui détermine si le graphe $C(G)$ est bicoloriable sans construire toutes les arêtes de $C(G)$, l'algorithme qu'il propose tient compte de la nature particulière de ce graphe ; ceci lui permet d'obtenir une efficacité très grande en $O(n)$. Les détails techniques sont assez ingénieux et ne peuvent être décrits en détail ici. L'algorithme général est basé sur les mêmes techniques.

2.2. Les principales étapes de l'algorithme général. La *première étape* de l'algorithme consiste à décomposer le graphe en *composantes biconnexes.* Décrivons brièvement de quoi il s'agit, un graphe est biconnexe s'il est connexe et si la suppression d'un sommet le laisse connexe ; il est facile d'obtenir pour un graphe G non biconnexe les sommets d'articulation (ceux qui supprimés disconnectent le graphe) et une décomposition de G en une réunion de graphes biconnexes, c'est ce qu'on appelle les composantes biconnexes. Celles-ci n'ont que des points d'articulation en commun. On voit alors que G est planaire si et seulement si ses composantes biconnexes le sont. On supposera donc que le graphe dont on teste la planarité est biconnexe. On illustrera l'algorithme à l'aide du graphe suivant :



La *deuxième étape* de l'algorithme consiste à appliquer l'algorithme de Trémaux (ou algorithme de parcours en profondeur d'abord), de numéroter les sommets dans l'ordre où ils ont été rencontrés, de construire un arbre recouvrant et de diviser les arêtes en deux sous-ensembles les arêtes de l'arbre et les arêtes dites de retour (qui joignent un sommet à un de ses ancêtres dans l'arbre). Cette procédure à été décrite dans le chapitre 5 du cours de tronc commun Algorithmes et Programmation. Dans la suite on considérera que les sommets sont numérotés dans l'ordre où on les rencontre dans ce parcours de Trémaux et lorsqu'il sera question du sommet x, x sera en fait le numéro d'ordre du sommet obtenu lorsque l'on effectue ce parcours. Sur le graphe ci-dessus on obtient :



Lors de la *troisième étape* de l'algorithme on construit tout d'abord le point d'attache $at[x]$ de chaque sommet x. Cette notion a été introduite dans le Chapitre sur les graphes polycopié de tronc commun, le point d'attache de x est le sommet de numéro le plus faible que l'on peut atteindre par un chemin d'origine x et composé d'une suite d'arêtes descendants dans l'arbre et d'un seul arc remontant. Après avoir calculé ces points d'attache, on réordonne la liste des successeurs de chacun des sommets x du graphe de la façon décrite comme suit. A chaque arête $\{x, y\}$ on associe un entier $v(x, y)$ égal à

– $2y$ si $\{x, y\}$ est un arc de retour de x vers y, ainsi $y < x$

– $2at[y]$ si $\{x, y\}$ est une arête de l'arbre ($x < y$) et s'il existe un arc de retour qui part du sous arbre de racine y et qui remonte vers x ou vers un sommet situé en dessous de $at[y]$ et en dessus de x.
– $2at[y] + 1$ si $\{x, y\}$ est une arête de l'arbre et si tout arc de retour qui part du sous arbre de racine y remonte vers $at[y]$ ou vers un sommet situé en dessous de x.

La liste d'adjacence de x est alors réorganisée suivant les $d(x, y)$ croissants : elle se compose des sommets y tels que $\{x, y\}$ est dans l'arbre et $x < y$ et des sommets z tels que $\{x, z\}$ n'est pas dans l'arbre et $z < x$. Avec cette dernière convention, chaque arête n'intervient qu'une fois dans l'ensemble des listes de successeurs. On peut considérer que l'on a ainsi obtenu un graphe orienté, chaque arête de l'arbre est orientée du sommet le plus petit vers le plus grand et pour les arêtes en dehors de l'arbre l'orientation est inverse.

Sur l'exemple donné ci dessus on obtient les listes :

1	:	2	2	:	3	3	:	4
4	:	5	5	:	6	6	:	8, 7, 3
7	:	2, 3	8	:	9, 5	9	:	1, 10
10	:	11, 8	11	:	4, 5, 9			

Au cours de la *quatrième étape* de l'algorithme on divise l'ensemble des arêtes du graphe en chemins par la procédure récursive `Chemins (Sommet x, Liste u)` que l'on initialise par l'appel `Chemins (1, vide)`.

```
void Chemins (Sommet x, Liste u) {
      Pour tout y dans la liste d'adjacence de x {
         u = Ajouter(y,u);
         if (x < y) Chemins(y, u);
           else {
             Afficher(u);
             u = vide;}
      }
}
```

On remarque que cette procédure affiche un chemin dès qu'elle rencontre une arête qui n'est pas dans l'arbre, il y a donc autant de chemins que de telles arêtes. D'autre part on peut montrer que, si le graphe est biconnexe, le premier chemin obtenu est un cycle.

Sur l'exemple donné ci-dessus on obtient les 9 chemins suivants :

$(1, 2, 3, 4, 5, 6, 8, 9, 1)$	$(9, 10, 11, 4)$
$(11, 5)$	$(11, 9)$
$(10, 8)$	$(8, 5)$
$(6, 7, 2)$	$(7, 3)$
$(6, 3)$	

Les chemins construits permettent de se replacer dans la situation rencontrée dans le paragraphe précédent, c'est à dire dans le cas où le graphe est hamiltonien. D'une certaine façon le premier chemin γ_0 obtenu (qui est un cycle) joue le rôle du cycle hamiltonien les autres chemins jouent le rôle des cordes, mais ceci n'est pas tout à fait exact, il faut en effet introduire la notion de segment qui est un sous-ensemble de l'ensemble des chemins construits par l'algorithme précédent.

Un *segment* est soit
- un chemin constitué uniquement d'un arc de retour et dont les deux sommets sont sur γ_0 ,
- un ensemble de chemins comportant un chemin dont la première arête $\{u, v\}$ fait partie de l'arbre recouvrant et telle que $u \in \gamma_0$, et tous les autres chemins dont l'origine est dans le sous-arbre de racine v.

Dans l'exemple ci-dessus il y a quatre segments, deux sont du premier type $\{(7,3)\}$ et $\{(8,5)\}$, les deux autres sont $\{(9,10,11,4),(11,9),(11,5),(10,8)\}$ et $\{(6,7,2),(6,3)\}$ On peut alors construire un graphe d'incompatibilité dont les sommets sont les segments et les arêtes relient deux segments qui ne peuvent être tracés tous les 2 à l'intérieur du cercle γ_0 sans intersection. On alors le résultat :

THÉORÈME 8.1. *Le graphe G est planaire si et seulement si chacun des graphes constitué d'un segment et du cycle γ_0 est planaire et si le graphe d'incompatibilité est bicoloriable.*

Le théorème donne donc un algorithme récursif de planarité qui à chaque étape vérifie la bicoloration d'un graphe de segments, de la même façon que pour le cas hamiltonien ce n'est pas le test de bicoloration qui est coûteux mais la construction du graphe d'incompatibilité. L'algorithme de planarité dû à Hopcroft et Tarjan évite la construction complète de ce graphe et donne le résultat en un temps $O(n)$.

3. Représentation par permutations : cartes

Une fois que l'on a déterminé qu'un graphe est planaire il reste encore à le tracer dans le plan et à définir une structure de données efficace et concise pour manipuler cette représentation. La notion de carte combinatoire constitue une réponse à cette demande. Pour définir la notion de carte on donne une nouvelle définition des graphes, qu'il sera ensuite facile de transformer en carte. On part d'un ensemble B contenant un nombre pair de *brins*, intuitivement chaque brin représente l'incidence d'une arête et d'un sommet. De manière plus précise on a :

Un *graphe symétrique* G sur un ensemble B de brins est déterminé par deux partitions S et A de B telles que toutes les classes de A contiennent *exactement deux brins*. Les classes de S sont appelées *les sommets* de G et celles de A les arêtes de G, deux sommets u et v sont adjacents s'il existe une arête a de A tel que $u \cap a \neq \emptyset$ et $v \cap a \neq \emptyset$ Une *boucle* est une arête dont les deux brins appartiennent au même sommet, deux arêtes a et a' sont parallèles s'il existe deux sommets u et v tels que $u \cap a$, $v \cap a$, $u \cap a'$ et $v \cap a'$ ne sont pas vides. Un graphe sans boucle ni arêtes parallèles est dit *simple*.

À partir du tracé d'un graphe planaire dans le plan on constitue un ensemble de brins en divisant en deux les arcs de Jordan correspondant aux arêtes, chacun des deux brins est alors incident à l'un des sommets extrémités de l'arête en question. Deux brins appartiennent à une même classe de la partition A s'ils forment une arête ; ils appartiennent à une même classe de la partition S s'ils sont incidents au même sommet.

Le plan étant orienté canoniquement par le sens trigonométrique direct, on peut replacer les partitions par des permutations, à chaque sommet x d'un tracé planaire du graphe G on fait correspondre le cycle des brins que l'on rencontre en tournant autour de x dans le sens trigonométrique positif. L'ensemble de tous les sommets donne ainsi une permutation notée σ dont chaque cycle correspond à un sommet du graphe. De même on peut associer à chaque arête un cycle formé de 2 brins, le produit de ces cycles donnant une permutation α. La topologie du dessin est alors complètement déterminée par les deux permutations σ et α sur l'ensemble B de brins.

On définit une *carte combinatoire* comme un couple de permutations (σ, α) sur un ensemble B de brins, telles que tous les cycles de α sont de longueur 2 et tout sous ensemble strict B' de B qui est union de cycles de σ n'est pas union de cycles de α. Cette dernière propriété exprime la connexité du graphe représenté par la carte.

On alors le résultat suivant qui est une traduction de la relation d'Euler :

PROPOSITION 8.3. *Dans une carte combinatoire σ, α qui représente un graphe planaire les faces correspondent aux cycles de la permutation produit $\alpha\sigma$, ainsi en notant $z(\alpha)$, $z(\sigma)$, $z(\alpha\sigma)$, le nombre de cycles des permutations $\alpha, \sigma, \alpha\sigma$ on a :*

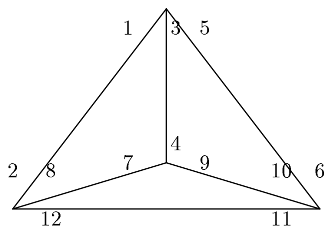
$$z(\sigma) + z(\alpha\sigma) = 2 + z(\alpha)$$

Sur la figure ci-dessous on a représenté le tracé de la carte sur l'ensemble de brins $B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$:

$$\begin{array}{lcl} \sigma & = & (1,3,5)(2,12,8)(4,7,9)(6,10,11) \\ \alpha & = & (1,2)(3,4)(5,6)(7,8)(9,10)(11,12) \end{array}$$

Ce qui donne les faces :

$$\alpha\sigma = (1, 12, 6)(2, 3, 7)(4, 5, 10)(8, 9, 11)$$

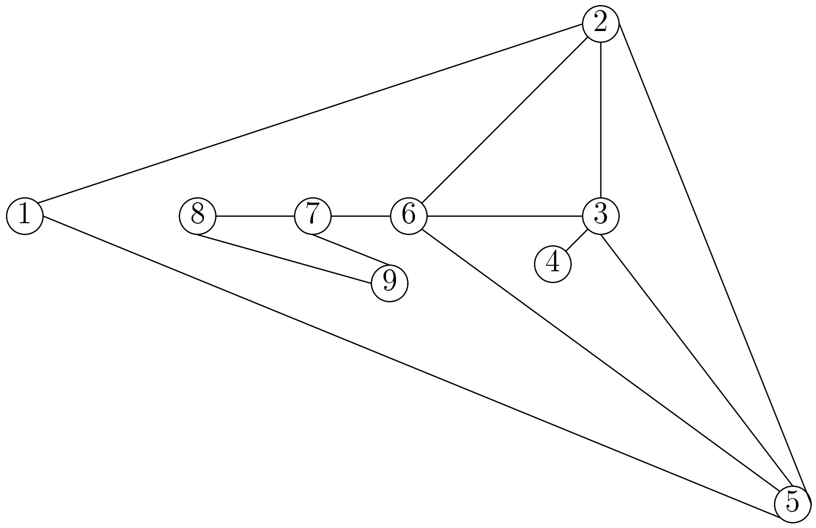


4. Tracé par des segments de droites

Dans ce paragraphe on souhaite montrer le résultat suivant :

THÉORÈME 8.2. *Tout graphe planaire admet un tracé dans lequel les arêtes sont représentées par des segments de droites.*

Ainsi le graphe donné au début du chapitre peut être dessiné par



La preuve s'effectue par récurrence sur le nombre de sommets et utilise le fait que l'on peut toujours redessiner un polygone convexe de façon à le rendre inscriptible dans un triangle.

CHAPITRE 9

Algorithmes des modèles de la physique

Plusieurs questions de physique statistique amènent à effectuer de longs calculs, en général il s'agit d'obtenir des fonctions de partition et des valeurs critiques. Il est ainsi naturel d'étudier la complexité des algorithmes mis en œuvre pour ces calculs. Depuis les travaux d'Onsager, le modèle d'Ising est considéré comme un des grands classiques du domaine, sa solution (c'est-à-dire le calcul de la fonction de partition) pour des graphes (ou réseaux) particuliers n'a pu être menée à bien que dans certains cas particuliers, il reste encore à faire pour venir complètement à bout, il en est ainsi pour les réseaux à 3 dimensions. Dans ce qui suit on va montrer que plusieurs des problèmes posés par les calculs de la physique statistique sont **NP**-complets en particulier les calculs liés au modèle d'Ising. Une exception notable est celle du calcul de la configuration d'énergie optimale dans le problème du verre de spin lorsque le graphe est planaire. Enfin on verra que si la physique a fourni des problèmes très intéressants à l'algorithmique elle lui a aussi apporté des heuristiques efficaces, en effet la méthode dite du *recuit simulé* donne dans de nombreux cas des solutions approchées à des problèmes d'optimisation combinatoire.

1. Modèle d'Ising

Ce modèle tente d'expliquer la physique des transitions de phase, il est appelé ainsi après les travaux réalisés dans les années 20 par Ernst Ising . Les transitions de phase, c'est à dire les changements qualitatifs profonds qui se produisent dans certains états d'un système lorsque l'on modifie légèrement les conditions de température ou de pression ne sont pas toujours bien expliqués. Le modèle d'Ising permet de décrire les interactions entre molécules dans un cristal et de calculer certains paramètres de changement de phase, comme la température critique par exemple. Il ne s'agit pas ici de développer les aspects physiques de la question mais de montrer comment le problème peut se traduire en termes de recherche de certains paramètres sur les graphes.

On considère un graphe symétrique $G = (X, E)$, où X est un ensemble de *sommets* et E un ensemble d'arêtes. Dans la suite on notera comme d'habitude n le nombre de sommets et m le nombre d'arêtes de G. Le graphe G est le plus souvent très régulier.

Par exemple dans le modèle à une dimension tous les sommets du graphe sauf 2 ont deux voisins, les deux sommets restant n'en ayant qu'un seul.

FIG. 1. Un réseau à une dimension

Dans le réseau à deux dimensions le graphe considéré est une grille :

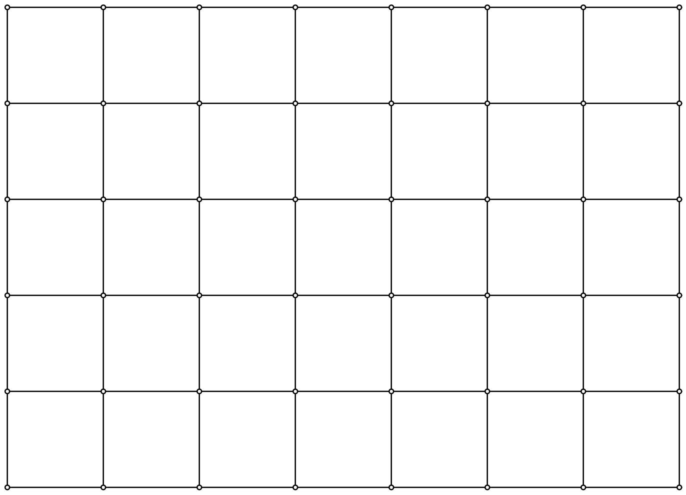


FIG. 2. Un réseau à deux dimensions

De plus on se donne une application J de E dans l'ensemble des réels, à chaque arête e est ainsi associée une énergie d'interaction $J(e)$ qui est souvent considérée comme constante ($\forall e, \quad J(e) = J$) mais qui dans certains cas peut dépendre de e. Une *configuration* σ du graphe est une application de l'ensemble X des sommets dans l'ensemble à deux éléments $\{-1, 1\}$ l'énergie de la configuration notée $H(\sigma)$ est égale à

$$H(\sigma) = \sum_{e=\{x,y\}\in E} -J(e)\sigma(x)\sigma(y) - M \sum_{x\in X} \sigma(x)$$

où M est une énergie due à un champ magnétique extérieur.

Un des problèmes principaux consiste à déterminer la fonction de partition Z donnée par :

$$Z = \sum_{\sigma} e^{-\beta H(\sigma)}$$

où la somme est prise sur toutes les 2^n configurations et où $\beta = \frac{1}{kT}$ est un paramètre qui est une fonction inverse de la température absolue T et qui fait intervenir la constante k

de Boltzmann. L'intérêt de la fonction de partition Z est que la probabilité de rencontrer une configuration σ donnée est égale à

$$P(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z}$$

Sur cette formule on constate que qu'à haute température les valeurs de β sont faibles et les configurations ont toutes des probabilités voisines, par contre à basse température les états à faible énergie ont une probabilité d'apparition bien plus élevée que ceux à énergie plus grande.

On va voir que le calcul de la fonction de partition se ramène au calcul du nombre de coupures de taille donnée du graphe G. Rappelons qu'une coupure est déterminée par une décomposition de l'ensemble des sommets X en deux parties disjointes X_1 et X_2, $X = X_1 \cup X_2$ et $X_1 \cap X_2 = \emptyset$, la coupure $C(X_1, X_2)$ est alors constituée de toutes les arêtes qui ont une extrémité dans X_1 et l'autre dans X_2. Pour un graphe G et pour $j = 1, \dots, m$ on note b_j le nombre de coupures constituées de j arêtes. On note aussi $B(x)$ le polynôme

$$b(x) = \sum_{j=0}^{m} b_j x^j$$

On remarque que chaque sous ensemble X_1 de X donne deux coupures de même taille ainsi $B(1) = 2^{n-1}$ où n est le nombre de sommets du graphe. On a alors

Théorème 9.1. *Dans un graphe G à m arêtes où l'énergie d'interaction est une constante J sur chacune des arêtes et où le champ magnétique M est nul, la fonction de partition et le polynôme des coupures sont reliés par :*

$$Z = \frac{2}{\theta^m} P(\theta^2)$$

où $\theta = e^{-\beta J}$

Preuve : Chaque configuration σ détermine une coupure (X_1, X_2) en posant

$$X_1 = \{x | \sigma(x) = 1\} \quad X_2 = \{x | \sigma(x) = -1\}$$

En notant m_1 et m_2 le nombre d'arêtes qui joignent respectivement les sommets de X_1 et ceux de X_2 entre-eux et $c(X_1, X_2)$ le nombre d'arêtes de la coupure $C(X_1, X_2)$ on a :

$$H(\sigma) = -J(m_1 + m_2 - c(X_1, X_2))$$

Or le nombre m d'arêtes de G vérifie

$$m = m_1 + m_2 + c(X_1, X_2)$$

ce qui donne :

$$H(\sigma) = -mJ + 2Jc(X_1, X_2)$$

et pour la fonction de partition :

$$Z = \sum_{\sigma} e^{\beta m J} e^{-2\beta c(X_1, X_2)}$$

Or chaque décomposition de X en $X_1 \cup X_2$ donne deux configurations de même énergie ainsi :

$$Z = 2e^{\beta m J} \sum_{j=0}^{m} b_j e^{2j\beta J}$$

et en posant $\theta = e^{-\beta J}$ on a :

$$Z = \frac{2}{\theta^m} \sum_{j=0}^{m} b_j \theta^{2j}$$

d'où le résultat. □

Corollaire 9.1. *Le calcul de la fonction de partition est de complexité équivalente à la détermination du polynôme des coupures P du graphe G.*

Preuve : On a vu que la connaissance de P permet d'obtenir Z, réciproquement si on connaît la valeur de Z, pour m valeurs de β une formule d'interpolation classique permet d'obtenir les coefficients de P. □

2. Modèle verre de spin

Dans le cas des verres de spin le cristal est un alliage contenant différents composants ainsi les différentes arêtes e du graphe ont des énergies d'interaction $J(e)$ diverses. Il s'agit de trouver la configuration pour laquelle $H(\sigma)$ est minimale. Par exemple si l'on considère le graphe constitué de 6 sommets et 7 arêtes formant deux carrés voisins (voir Figure), dans lequel $J(e) = 1$ sauf $J\{1,2\} = J\{5,6\} = -3$. Une solution optimale avec $H(\sigma) = -9$ est donnée par :

$$\sigma(1) = \sigma(3) = \sigma(5) = +1 \qquad \sigma(2) = \sigma(4) = \sigma(6) = -1$$

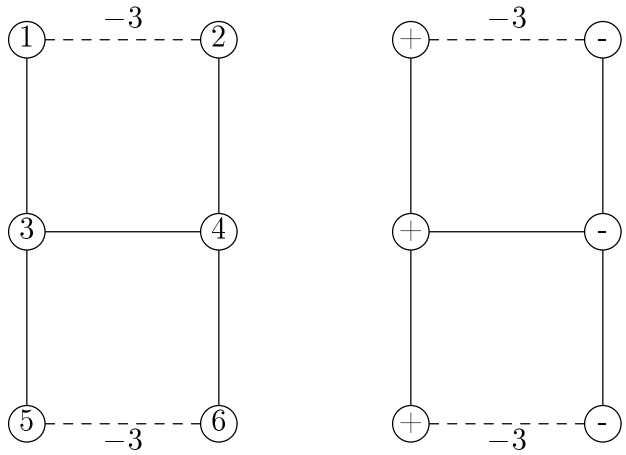


Fig. 3. Configuration d'énergie optimale

Dans le cas général le problème est NP-complet et il est donc illusoire de chercher un algorithme qui donne rapidement une solution exacte. Un cas particulier intéressant donne lieu à un algorithme polynomial il s'agit du cas où le graphe est planaire.

Lorsqu'il existe une configuration σ_0 telle que $\sigma_0(x)\sigma_0(y)J(e)$ est positive pour toute arête e du graphe on dit que le cas est équivalent au modèle ferro-magnétique. Il est relativement facile de déceler ce type de situation.

Soit $G = (X, E, J)$ un graphe aux arêtes valuées, une arête est dite négative si $J(e) < 0$, on a alors la proposition suivante :

PROPOSITION 9.1. *Le graphe valué $G = (X, E, J)$ admet une solution optimale telle que $H(\sigma) = -\sum_{e \in E} |J(e)|$ si et seulement si tout cycle de G contient un nombre pair d'arêtes négatives.*

Preuve : A partir d'un graphe valué $G = (X, E, J)$ on construit le graphe G' obtenu en ajoutant un nouveau sommet sur chacune des arêtes e de G d'image par σ positive. Ces nouveaux sommets sont de degré 2. On vérifie alors que (G, J) est équivalent au modèle ferromagnétique si et seulement si les sommets de G' peuvent être partagés en deux classes X_1 et X_2 telles que toute arête de G' a une extrémité dans X_1 et une autre dans X_2. Le graphe G' est ainsi bicoloriable, or une condition nécessaire et suffisante pour qu'un graphe soit bicoloriable est que tous ses cycles soient de longueur paire. La preuve se termine en remarquant que G' a tous ses cycles de longueur paire si et seulement si tout cycle de G contient un nombre pair d'arêtes négatives. $\square$

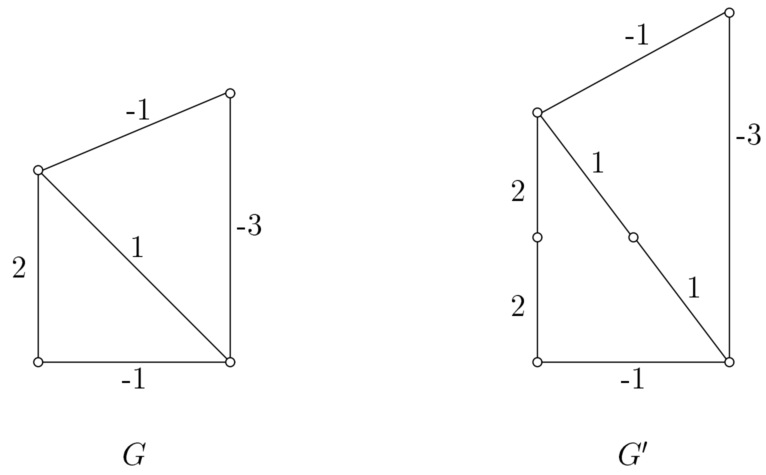


FIG. 4. Un graphe valué et le graphe à bicolorier associé

La proposition qui précède donne un critère théorique difficile à vérifier. Par contre il est assez facile de vérifier qu'un graphe est équivalent au modèle ferromagnétique en utilisant l'algorithme suivant :

– On attribue la valeur $\sigma(x_0) = +1$ à l'un quelconque des sommets x_0
– on calcule de proche en proche la valeur des $\sigma(x)$, en utilisant la règle :
Si une arête $e = \{x, y\}$ telle que $\sigma(y)$ est connu alors que $\sigma(x)$ ne l'est pas alors on choisit $\sigma(x)$ de façon à ce que :

$$\sigma(x)\sigma(y)J(e) \geq 0$$

– Une fois que l'on a déterminé les valeurs de $\sigma(x)$ pour tous les sommets du graphe on vérifie que la condition $\sigma(x)\sigma(y)J(e) \geq 0$ est satisfaite pour toutes les arêtes.

3. Algorithme dans le cas planaire

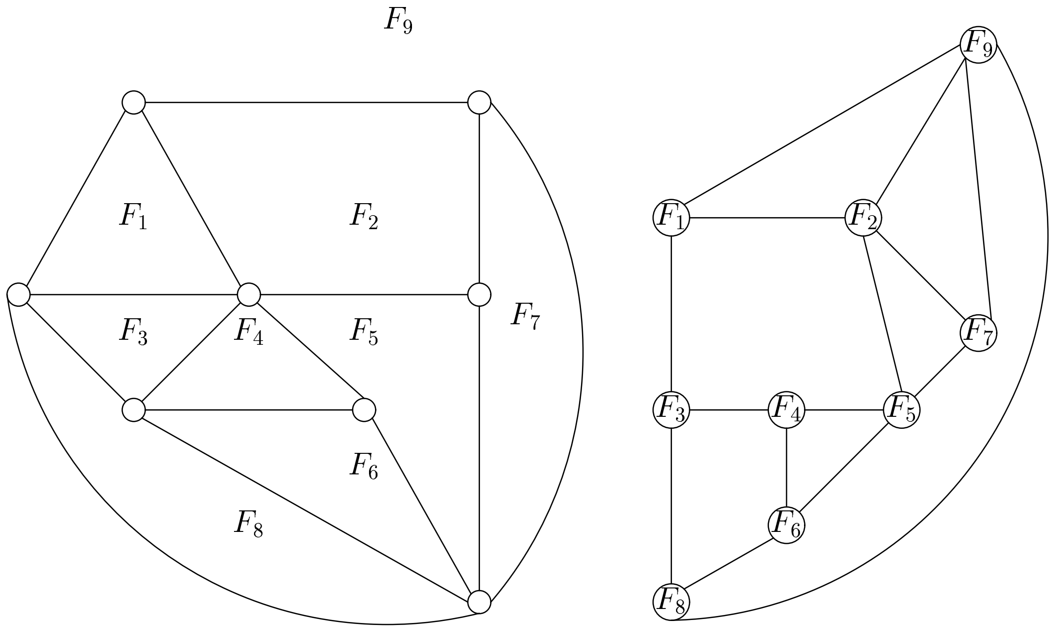


FIG. 5. Un graphe planaire et son dual

Les arêtes d'un graphe planaire connexe délimitent des domaines simplement connexes : *les faces*. Une de ces faces est non bornée on l'appelle la *face infinie*. A chacune de ces faces on associe le cycle formé des arêtes qui la séparent d'une autre face. Ces cycles jouent un rôle important, car tout cycle d'un graphe planaire peut s'obtenir comme différence symétrique d'une famille de cycles associés aux faces. Ainsi la proposition donnée plus haut se simplifie pour les graphes planaires par :

PROPOSITION 9.2. *Un graphe valué $G = (X, E, J)$ planaire est équivalent au modèle ferromagnétique si et seulement si toute face de G contient un nombre pair d'arêtes négatives.*

Une face qui contient un nombre impair d'arêtes négatives est dite *frustrée*. Un graphe associé à ces faces va permettre de calculer une configuration d'énergie minimale.

Pour toute configuration σ on dit qu'une arête $e = \{x, y\}$ est dite *contrariée* si $J(e)\sigma(x)\sigma(y) < 0$. Il est clair que l'énergie d'une configuration est déterminée par l'ensemble des arêtes contrariées. Rechercher une configuration d'énergie minimale revient à rechercher un ensemble d'arêtes contrariées dont la somme des valeurs absolues des valuations soit minimale. L'ensemble des arêtes contrariées associées à une configuration σ est noté C_σ.

PROPOSITION 9.3. *Soit $G = (X, E, J)$ un graphe valué planaire et U un sous-ensemble d'arêtes de E, U forme un ensemble d'arêtes contrariées pour une configuration σ si et seulement si :*

(1) *Toute face frustrée contient un nombre impair d'arêtes dans U.*
(2) *Toute face non frustrée contient un nombre pair d'arêtes dans U.*

Preuve : Le fait qu'un ensemble d'arêtes contrariées vérifient 1. et 2. est immédiat. Considérons la réciproque : soit J' défini par :
$J'(e) = -J(e)$ si e appartient à U, $J'(e) = J(e)$ sinon.
Alors J' défini une valuation dans laquelle aucune face n'est frustrée, il existe alors une configuration de spin σ pour laquelle aucune arête n'est contrariée pour J'. Dans cette configuration les arêtes contrariées pour J sont exactement les arêtes de U. $\square$

On associe à un graphe planaire son graphe dual G^* construit de la façon suivante : les sommets de G^* sont les faces de G ,(noter que la face infinie fait partie des faces de G). Deux sommets sont reliés dans G^* par une arête si et seulement si ils correspondent à deux faces de G adjacentes, c'est à dire possédant une arête de G en commun sur leurs frontières.

Les arêtes du graphe dual G^* de G correspondent aux arêtes de G, les faces frustrées dans G donnent lieu à des sommets frustrés dans G^*. La proposition précédente se traduit dans G^* par.

PROPOSITION 9.4. *Un ensemble U d'arêtes forme un ensemble d'arêtes contrariées pour le graphe valué planaire $G = (X, E, J)$ si et seulement si :*

(1) *Tout sommet frustré de G^* est incident à un nombre impair d'arêtes de U.*
(2) *Tout sommet non frustré de G^* est incident à un nombre pair d'arêtes de U.*

Comme conséquence immédiate on obtient que le nombre de sommets frustrés de G^* (et donc que le nombre de faces frustrées de G) est pair. On a aussi :

PROPOSITION 9.5. *Si σ est une configuration d'énergie minimale alors l'ensemble des arêtes contrariées forme un sous graphe sans cycle de G^*.*

Preuve : S'il existe un cycle dans U on considère l'ensemble U' obtenu en supprimant dans U les arêtes du cycle, U' vérifie les conditions de la proposition car la suppression d'un cycle ne modifie pas la parité des degrés des sommets, et la configuration définie par U' est d'énergie plus faible. □

Au graphe G on associe le graphe $\hat{G}$ construit de la façon suivante :
- L'ensemble des sommets de $\hat{G}$ est constitué des sommets frustrés de G^*
- L'ensemble des arêtes de $\hat{G}$ est constitué de toutes les paires $\{x, y\}$ de sommets de $\hat{G}$.
- Chaque arête $\{x, y\}$ est valuée par la longueur du plus court chemin qui joint x à y dans G^*, où la longueur d'un chemin est la somme des valeurs absolues $|J(e_i)|$ des arêtes e_i qui en font partie.

Un sous ensemble C de l'ensemble des arêtes d'un graphe est un *couplage parfait*, si deux arêtes de C n'ont pas d'extrémité commune et si tout sommet de G est extrémité d'une arête de C. La *valuation* d'un couplage parfait est la somme des valuations des arêtes.

Un couplage parfait C de valuation minimale dans $\hat{G}$ détermine un ensemble d'arêtes U_C de G^*, en effet à chaque arête de C est associé un chemin dans G^* et U_C et constitué de l'union de tous les chemins ainsi formés.

THÉORÈME 9.2. *Un couplage parfait C de valuation minimale dans $\hat{G}$ détermine un ensemble d'arêtes U_C qui forme un ensemble d'arêtes contrariées pour une configuration σ de G, cette configuration est d'énergie minimale.*

Preuve : Il est assez facile de voir que l'on peut décomposer les arêtes d'un graphe sans cycle en une famille de chemins disjoints qui relient deux à deux les sommets de degré impairs. Ces chemins déterminent donc un couplage dans le graphe des faces frustrées. Si le couplage minimal est de valuation plus faible, celui-ci détermine des chaînes reliant des sommets frustrés de G^*, il est facile de voir que ces chaînes n'ont pas d'arêtes en commun (sinon on pourrait améliorer le couplage) en choisissant l'ensemble des arêtes de ces chaînes comme arêtes contrariées on obtient une configuration d'énergie inférieure.

Conséquence Déterminer une configuration d'énergie minimale revient à déterminer un couplage minimal dans le graphe $\hat{G}$ il existe des algorithmes permettant d'effectuer ceci en temps polynomial. Ils ont été évoqués dans le chapitre 3 (Flots et Couplages), toutefois ils sont assez difficiles à mettre en œ et leur complexité bien que polynomiale est assez élevée.

Sur la figure ci-dessus on a représenté un graphe régulier à 20 sommets, les interactions sont telles que $J(e) = 1$ (traits normaux) ou $J(e) = -1$ (traits en pointillés).

4. Recuit simulé

Pour résoudre certains problèmes d'optimisation combinatoire, on procède par améliorations successives d'une configuration. Dans le contexte considéré ici cela se traduit par partir d'une configuration quelconque et tenter de l'améliorer en modifiant l'une de ses composantes.

On obtient, en réitérant ce procédé une configuration qui est meilleure que toutes ses "voisines", on dit que l'on a atteint un *optimum local*. Rien ne dit que cet optimum local

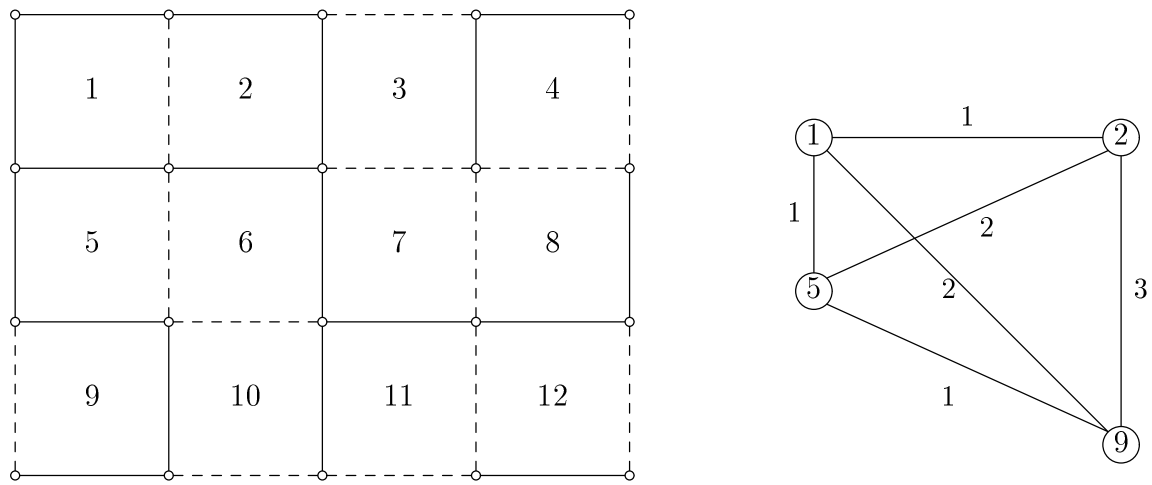


FIG. 6. Un graphe planaire valué G et le graphe $\hat{G}$

soit l'optimum général ou même qu'il en soit proche. Et c'est là l'inconvénient de cette méthode.

Pour tenter de pallier cet inconvénient, on modifie l'algorithme en faisant intervenir une part de hasard. On part d'une configuration quelconque σ, on cherche une configuration voisine σ' si celle ci est telle que $H(\sigma') < H(\sigma)$ on remplace σ par σ' et on recommence le procédé. Si $H(\sigma') > H(\sigma)$, on tire au hasard un nombre λ, compris entre des valeurs à déterminer, et on remplace σ par σ' si $H(\sigma') > H(\sigma) - \lambda$ puis on répète le procédé. Tout l'art réside dans la détermination du λ, celui-ci peut d'ailleurs varier au cours de l'exécution de l'algorithme. L'algorithme se termine quand on est resté suffisamment longtemps sur la même configuration. Cette façon de procéder s'appelle la méthode du recuit simulé, elle est utilisée dans certains cas pour résoudre de façon approchée des problèmes d'optimisation combinatoire NP complets.

Bibliographie

[1] Ausiello Giorgio, Crescenzi Pierluigi, Gambosi Giorgio, Kann Viggo, Marchetti-Spaccamela Alberto, Protasi Marco, Complexity and Approximation. Combinatorial optimization problems and their approximability properties. ISBN 3-540-65431-3. Springer-Verlag, 1999.

[2] Berge Claude, *Graphes et Hypergraphes*, Bordas, 1973.

[3] Chvátal Vasek, *Linear Programming* McGraw-Hill, 1983.

[4] Cormen Thomas, Leiserson Charles, Rivest Ronald, *Introduction to Algorithms*, MIT Press, 1990.

[5] Crochemore Maxime, Rytter W. *Text Algorithms.* Oxford University Press, 1994.

[6] Even Shimon, *Graph Algorithms*, Computer Science Press, 1979.

[7] Garey Michael R., Johnson David S., Computers and Intractability : A Guide to the Theory of NP-completeness. ISBN 0-7167-1045-5. Freeman, 1979.

[8] Gibbons Alan, *Algorithmic Graph Theory* Cambridge University Press, 1985.

[9] Goldberg David, *Genetic Algorithms*, Addison Wesley 1989.

[10] Gondran Michel, Minoux Michel, *Graphes et Algorithmes*, Eyrolles, 1995.

[11] Knuth Donald, *The Art of Computer Programming, Vol 3 : Sorting and Searching*, Addison Wesley, 1973.

[12] Lawler, E. L, Lenstra J.K, Rinnooy K, Shmoys D. B, *The Travelling Salesman Problem : a Guided Tour of Combinatorial Optimization*, John Wiley and sons, 1985 .

[13] Lint J. van, Wilson Robert, *A course in Combinatorics*, Cambridge University Press, 1992.

[14] Motwani R., Raghavan P. Randomized Algorithms. ISBN 0-521-47465-5. Cambridge University Press, 1995.

[15] Nemhauser Georges, Wolsley Laurence, *Integer and Combinatorial Optimization*, John Wiley and sons 1988

[16] Papadimitriou C. H. Computational complexity. ISBN 0-201-53082-1. Addison-Wesley, 1994.

[17] Papadimitriou C. H., Steiglitz K. Combinatorial Optimization. Prentice-Hall, 1982.

[18] Roth Alvin, Sotomayor Marilda, *Two sided matching*, Cambridge University Press, 1990.

[19] Welsh Dominic, *Matroid Theory*, Academic Press, 1976.

Index