Conception et analyse d'algorithmes

Robert Cori, Guillaume Hanrot, Claire Kenyon et
Jean-Marc Steyaert

Deuxième partie

Complexité

Préambule

La première partie de ce cours a présenté des paradigmes qui conduisent à des algorithmes efficaces. 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 de la théorie de la complexité est d'obtenir des résultats négatifs. Les résultats absolus (comme dans le cas du tri) sont très rares ; on aborde donc 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 en classes de complexité selon leur niveau de difficulté ; à l'intérieur d'une classe, on essaie de même d'isoler les problèmes les plus difficiles de cette classe, représentatifs de la difficulté inhérente à la classe. Ces problèmes sont dits *complets*.

On s'intéressera essentiellement à la classe **NP** des problèmes solubles 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.

P et NP

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 qui sur toute entrée de longueur n, résout le problème en $O(f(n))$ pas de calcul.

Les classes de complexité s'intéressent habituellement 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, c'est-à-dire que cette question appelle une réponse oui ou non. FACTORISATION, en revanche, n'est pas un problème de décision.[2]

Soit $\mathbf{P} = \cup_{k\geq 0}\mathbf{DTIME}(n \mapsto n^k)$ la class des problèmes de décision solubles en temps polynomial. Cette classe de complexité est une notion très robuste en ce sens qu'elle est indépendante du choix de modèle de calcul raisonnable, tel que RAM, machine de Turing, ou pseudo-code par exemple. On l'interprète comme la classe des problèmes qui ont une solution de complexité "raisonnable".

On définit également la classe $\mathbf{NP} = \cup_{k\geq 0}\mathbf{NTIME}(n \mapsto n^k)$, où le "N" signifie "non-déterministe". Ici le modèle de calcul est enrichi par une instruction de "choix" (par exemple : "choisir une permutation σ de $\{1, \ldots, n\}$), et on a une instance *positive* du problème si et seulement si il existe une façon de faire les choix qui conduise à la réponse "oui".

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 « $\mathbf{P} = \mathbf{NP}$? ».

Il existe 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 recherche dichotomique, 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.

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

[2]Les problèmes de calcul ont leurs propres classes de complexité **FP** et **FNP**.

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 $w = O(\log(mn))$.

NP-complétude

On dit qu'un problème P se *réduit polynomialement* à un problème Q si l'on peut construire une fonction τ transformant les entrées de P en des entrées de Q, telle que $P(x)$ soit vrai si et seulement si $Q(\tau(x))$ est vrai, et si τ est calculable en temps déterministe polynomial.[3] Conséquence importante : 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**-*difficile* si tout problème de **NP** se réduit polynomialement à lui, et **NP**-*complet* si de plus il appartient à **NP**.

L'intérêt de cette définition réside dans le fait suivant. Si un problème Q est **NP**-difficile, et si on sait le résoudre en temps polynomial, alors tous les problèmes de **NP** sont solubles en temps polynomial, et donc **P**= **NP**. Inversement, si **P**$\neq$ **NP** et si Q est un problème **NP**-difficile, alors il ne peut pas exister d'algorithme polynomial pour Q : si l'on croit à la conjecture **P** $\neq$ **NP**, la **NP**-complétude capture donc une certaine notion de problème "difficile".

THÉORÈME 4.4 (Cook). *Le problème* SAT *défini ci-dessous est* **NP**-*complet.*

SAT

Données : un ensemble de variables $\{x_1, \ldots, x_n\}$ et une formule logique sous forme normale conjonctinve : $F = C_1 \wedge C_2 \wedge \cdots \wedge C_\ell$, avec la chaque clause $C_i = (y_{i,1} \vee y_{i,2} \vee y_{i,k})$, où chacun des $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j} = x_p$ ou $y_{i,j} = \overline{x_p}$.

Résultat : "oui" si et seulement si F est satisfaisable, c'est-à-dire s'il existe une assignation de valeurs de vérité V/F aux variables qui rende F vraie.

La preuve de Cook, omise dans ce cours, permet en fait de montrer que le problème 3-SAT, où les clauses contiennent chacune trois littéraux au plus, est en fait déjà **NP**-complet. On montrera ce fait indépendamment. En revanche, le problème 2-SAT, où les clauses contiennent chacune deux littéraux au plus, est dans **P**. Cette remarque illustre un fait important : *ce n'est pas parce qu'un problème est* **NP**-*complet que toutes ses instances sont*

[3] Sur la définition de la notion de réduction, voir les compléments au chapitre 5.

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 pourra consulter 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

NP-complétude

Introduction

Ce chapitre décrit plusieurs exemples de réductions, et présente un certain nombre de problèmes **NP**-complets classiques.

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, le fait que SAT soit **NP**-complet ne prouve en rien que l'instance particulière 3-SAT le soit. Il faut prouver que cette instance est aussi difficile que le cas général.

3-SAT.
Données : un ensemble de variables $\{x_1, \ldots, x_n\}$ et une formule $F = C_1 \wedge \cdots C_\ell$ avec $C_i = (y_{i,1} \vee y_{i,2} \vee y_{i,3})$, où $y_{i,j}$ est égal soit à l'un des x_i, soit à l'un des $\overline{x}_i$.

Résultat : décider si F est satisfaisable par une affectation de valeurs de vérité V/F aux variables.

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

DÉMONSTRATION. 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 va réduire SAT à 3-SAT. Soit F une formule SAT. Soit C une clause de F, par exemple $C = x \vee y \vee z \vee u \vee v \vee w \vee t$. On introduit de nouvelles variables a, b, c, d associées à cette clause, et on remplace C par la formule

$$(x \vee y \vee a) \wedge (\overline{a} \vee z \vee b) \wedge (\overline{b} \vee u \vee c) \wedge (\overline{c} \vee v \vee d) \wedge (\overline{d} \vee w \vee t).$$

Il est facile de voir qu'une assignation de x, y, z peut être complétée par une assignation de a, b, c, d de façon à rendre cette formule vraie, si et seulement si $(x \vee y \vee z)$ est vraie. En appliquant cette construction à toutes les clauses de F et en prenant la conjonction des formules ainsi produites, on obtient ainsi une formule 3-SAT F' équivalente à F.

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. $\square$

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 soluble en temps polynomial.

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

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 : décider s'il existe une affectation de valeurs de vérité aux x_i de sorte que chaque clause contienne au moins un littéral vrai et au moins un littéral faux (c'est-à-dire : $\forall i,\ \exists j, k$ tels que $y_{ij} = V$ et $y_{ik} = F$)

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

DÉMONSTRATION. NAESAT appartient à **NP** car la donnée d'une affectation valide est un certificat vérifiable en temps polynomial.

On va réduire SAT à NAESAT. Soit F une formule 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 $C_i = y_{i1} \vee \cdots \vee y_{ik_i}$ de F en $C'_i = y_{i1} \vee \cdots \vee y_{ik_i} \vee z$.

Le caractère polynomial de cette réduction est évident.

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

Inversement, supposons que l'instance construite de NAESAT soit satisfaisable. Si la valeur de vérité de z dans l'affectation correspondante est F, les valeurs des variables x_i correspondent à une affectation valide pour l'instance de SAT d'origine (dans la clause considéré, au moins l'un des y_{ij} ou z vaut V, et ce n'est pas z). Si au contraire z vaut V, 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 F dans l'affectation initiale, et vaut donc maintenant V, tandis que z vaut F. On a donc construit une affectation dans laquelle z vaut F, 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. Donc NAESAT est **NP**-complet. □

Noter qu'en effectuant la même réduction sur 3-SAT, on prouve que NAE4SAT est **NP**-complet ; on va utiliser cette remarque pour prouver :

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

DÉMONSTRATION. On va réduire NAE4SAT à NAE3SAT. Soit $C = x \vee y \vee z \vee t$ une clause à 4 littéraux. On introduit une nouvelle variable u_C, et on forme les deux clauses $C_1 = x \vee y \vee \overline{u_C}$ et $C_2 = z \vee t \vee u_C$. En procédant ainsi pour toutes les clauses, on construit une instance F' de NAE3SAT.

Supposons que F' soit une instance positive de NAE3SAT, et considérons l'affectation de valeurs de vérité correspondante. Si $u = V$, alors x ou y est F, et z ou t est V, donc

$(x \vee y \vee z \vee t)$ a au moins un littéral V et au moins un littéral F ; de même si $u = F$; donc F est une instance positive de NAE4SAT.

Inversement, si F est une instance positive de NAE4SAT, considérons l'affectation de valeurs de vérité correspondante. Dans $(x \vee y \vee z \vee t)$, si x et y sont tous deux à V, on affecte u à V ; sinon, si x et y sont tous deux à F, on affecte u à F ; sinon, on affecte à u la valeur de vérité adéquate pour la clause $(u \vee z \vee t)$. On voit que ceci donne une assignation démontrant que F' est une instance positive de NAE3SAT.

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

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, c'est-à-dire de la forme $(x_1 \wedge x_2 \wedge \cdots \wedge x_n \Rightarrow y)$, ou encore $(\overline{x}_1 \vee \overline{x}_2 \vee \cdots \vee \overline{x}_n \vee y)$, avec tous les littéraux négatifs sauf un). 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 V par clause) ; CIRCUIT-SAT (la formule n'est pas donnée sous forme normale conjonctive, mais sous la forme d'un circuit à n entrées, avec des portes logiques $\vee, \wedge, \bar{}$ décrivant le calcul de la sortie en fonction des variables d'entrée).

2. Stable

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

Résultat : Décider s'il existe $V' \subset V$, $\mathrm{card}(V') = k$, tel que $u, v \in V' \;\Rightarrow\; (u, v) \notin E$.

THÉORÈME 5.3. STABLE *est un problème* **NP**-*complet.*

DÉMONSTRATION. STABLE 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 STABLE, 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 de valeurs de vérité satisfaisant F.

Soit $F = \bigwedge_{1 \leq j \leq p}(x_{1j} \vee x_{2j} \vee x_{3j})$. Définissons un graphe G. G a $3k$ sommets, un pour chaque occurence d'un littéral dans une clause. Contraintes de cohérence : pour tout i, G a une arête entre chaque sommet associés à un littéral x_i et chaque sommet associé à un littéral $\overline{x_i}$ (ainsi, un stable de G correspond à une affectation de valeurs de vérité à une partie des variables). Contraintes associées aux clauses : si par exemple $C = (x_1 \vee \overline{x_2} \vee x_3)$ est une clause de F, alors G a les arêtes $\{x_1, \overline{x_2}\}, \{\overline{x_2}, x_3\}, \{x_3, x_1\}$ formant un triangle (ainsi un stable de G contient au plus un des trois sommets associés à la clause C).

Soit k le nombre de clauses de F. On démontre maintenant que F est satisfiables si et seulement si G possède une clique de taille k.

Si F est satisfiable, on considère une assignation des variables satisfaisant F. Pour chaque clause C de F, on choisit ℓ_C un littéral de C rendu vrai par l'assignation. Ceci définit k sommets formant un stable de G.

Inversement, si G a un stable de taille k, alors il a nécessairement exactement un sommet dans chaque triangle ; ce sommet correspond à un littéral rendant la clause associée vraie, et forme une assignation des variables cohérente par construction des arêtes formant les contraintes de cohérence.

La réduction est manifestement polynomiale. □

Deux problèmes classiques sont liés à STABLE :

CLIQUE
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 $(i, j) \in V_0 \times V_0$, $(i, j) \in E$?

COROLLAIRE 5.2. *Le problème* CLIQUE *est* **NP**-*complet.*

DÉMONSTRATION. Passage au complémentaire. □

RECOUVREMENT PAR 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 toute arête de E ait au moins une extrémité dans V_0 ?

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

DÉMONSTRATION. Complémentaire du stable de G. □

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 entre V_1 et V_2 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 $(u \vee t_1 \vee t_2) \wedge (v \vee w \vee \overline{t_1}) \wedge (v \vee z \vee \overline{t_2})$. 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 F.

On va construire un graphe $G = (V, E)$ de la façon suivante : G a $2n$ sommets. 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. Ceci définit G. Les réductions opérées dans le premier paragraphe de la preuve permettent de voir qu'à chaque clause correspond un triangle et que deux de ces triangles ont des arêtes distinctes.

Si l'on note n le nombre de variables et m le nombre de clauses, G a donc $2n$ sommets et $3m + n$ arêtes. 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 chaque clause $(u \vee v \vee w)$. Inversement, toute coupure de G 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.

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.

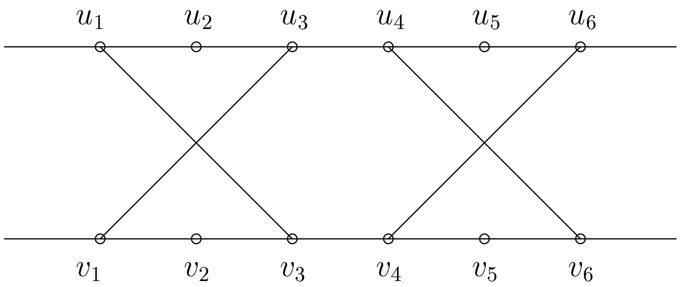
CIRCUIT HAMILTONIEN
Données : un graphe $G = (V, E)$.

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 $\forall (u,v) \in E, \chi(u) \neq \chi(v)$?

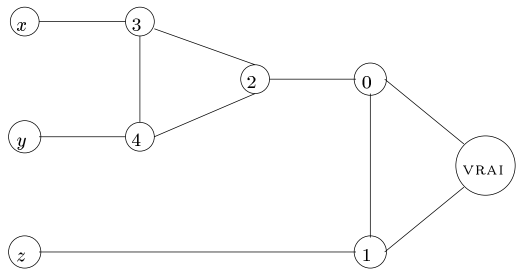
THÉORÈME 5.6. 3-COLORABILITÉ *est* **NP**-*complet.*

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 un graphe ayant $3 + 2n + 5m$ sommets, les trois permiers sont notés VRAI, FAUX, NSP. Ces trois sommets sont reliés deux à deux en triangle, de sorte qu'ils doivent être tous trois de couleurs différentes. On appelera les couleurs correspondantes VRAI, FAUX, NSP.

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

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 exprimant les entiers dans 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 , 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**-difficile. La difficulté, pour une fois, revient à prouver qu'il est dans **NP**.

Données : une matrice $m \times n$ notée A, un vecteur de dimension m 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**-difficile.

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* m*, 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 linéairement 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 **NP** = **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 :

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

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 **NPSPACE** = **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 $\mathbf{NP} = \mathbf{PCP}(0, \text{poly}(n))$. 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 $\mathbf{PCP}(\text{poly}(n),\ \text{poly}(n)) = \mathbf{NEXP}$...

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

Nous avons étudié de nombreux problèmes et montré qu'ils sont **NP**-complets. Cela implique que si $\mathbf{P} \neq \mathbf{NP}$, ces problèmes sont difficiles à résoudre. Ces problèmes consistent 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 petits, 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 ramènent. 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 $> B/3$ 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 $> B/2$. 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 $A \times B$. $\square$

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 , 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 A pour $(\mathrm{P},\ c)$ qui calcule un élément $A(x)$ tel que $\mathrm{P}(x)(y) = \texttt{true}$, est ε-approché si :

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

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 $k_{\text{opt}} \geq 2$, et donc que $3k_{\text{opt}}/2 + 1 \leq 2k_{\text{opt}}$. 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 chapitre 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 les sommets u et v ainsi que les arêtes adjacentes. 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 $11k_{\mathrm{opt}}/9+4$ 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 $(1-\varepsilon)^{-1}$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 $(k+1)\cdot A_{\text{opt}}$. Mais toute solution non optimale a un coût multiple de $k+1$ et strictement supérieur à $(k+1)\cdot A_{\text{opt}}$, donc distant de $(k+1)\cdot A_{\text{opt}}$ 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 connaît depuis 2002 un algorithme polynomial déterministe permettant de décider si un nombre est premier, mais il est de complexité élevée pour le moment. On se contente ici 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.

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 $X = \sum_{1 \leq i \leq n^{3/4}} X_i$. 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 $n^{3/4}(1 - k/n)(k/n) \leq n^{3/4}/4$.

On peut alors utiliser l'inégalité de Tchebitchev, qui dit que $P(|X - E[X]| > \varepsilon) \leq \operatorname{var}(X)/\varepsilon^2$. Dans le cas présent, pour $\varepsilon = \sqrt{n}$, il vient

$$P(|X - kn^{-1/4}| > \sqrt{n}) \leq n^{-1/4}/4,$$

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 $O(n^{-1/4})$. $\square$

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. □

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 $M=\{0,\dots,m-1\}$ l'ensemble duquel proviennent les éléments à hacher, et soit $V \subset M$ de cardinal v l'ensemble à hacher. Quitte à augmenter[1] m, on peut supposer que $p=m+1$ est un nombre premier. On fixe un entier $r \geq v$, et on introduit la fonction

$$h_k(x) = (kx \bmod p) \bmod r,$$

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.

[1]on peut montrer que cette augmentation est d'un facteur linéaire ≤ 2 au plus.

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 procède 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.

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] Tarjan Robert Endre, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.

[20] Vazirani Vijay V., *Approximation Algorithms*, Springer 2001.

[21] Welsh Dominic, *Matroid Theory*, Academic Press, 1976.