

Notes de cours

Algorithmique avancée

Ecole Centrale

Pierre Fraigniaud

CNRS et Université Paris Diderot
pierre.fraigniaud@irif.fr



Version 17 octobre 2017

Résumé

Ce cours a pour objectif de donner aux élèves un aperçu de quelques unes des techniques modernes de conception et d'analyse d'algorithmes. Ainsi, tous les grands thèmes de l'algorithme seront abordés dans le cours : calculabilité, complexité, récursivité, programmation dynamique, programmation linéaire, algorithmes d'approximation, algorithmes paramétrés, algorithmes probabilistes, etc. A l'issue de ce cours, les élèves devraient être capables d'identifier la ou les méthodes les plus appropriées pour la résolution des problèmes algorithmiques qu'ils pourront rencontrer dans leurs carrières. Il ne sera bien sûr pas possible de rentrer dans les détails de toutes les thématiques abordées dans le cours, qui mériteraient chacune un cours à part entière. Toutefois, les ouvrages de référence qui seront indiqués durant le cours devraient permettre de satisfaire tous les élèves désirant en savoir plus sur tels ou tels thèmes du cours.

Ouvrages de référence

- Complexité algorithmique [11]
- Techniques de conception et d'analyse d'algorithmes [3]
- Algorithmes d'approximation [14]
- Algorithmes paramétrés [5]
- Algorithmes probabilistes [9]
- Méthodes heuristiques [8]
- Algorithmes parallèles [6]
- Algorithmes distribués [1, 7, 12]
- Algorithmes *online* [2]

Table des matières

1 Préambule : les graphes et PageRank de Google	5
1.1 Des objets au cœur de l’algorithmique : les graphes	5
1.2 Codage et notations asymptotiques	7
1.3 Matrices et graphes	8
1.4 Composantes connexes	9
2 Calculabilité et complexité algorithmique	10
2.1 Machine de Turing	10
2.1.1 Définition	10
2.1.2 Indécidabilité	11
2.2 Machines RAM	13
2.2.1 Les modèles de machines RAM	14
2.2.2 Le modèle du cours	15
2.3 Les classes P et NP	16
2.3.1 Les problèmes de décision	16
2.3.2 La classe P	17
2.3.3 La classe NP	18
2.3.4 Réduction polynomiale et problème NP-difficiles	21
2.4 Exemples de preuves de NP-complétude	23
2.4.1 Réduction à partir de 3-SAT	23
2.4.2 Réduction à partir de COUVERTURE ENSEMBLE	26
2.5 Complexité spatiale	26
2.5.1 Espace polynomial	26
2.5.2 Espace logarithmique	27
2.5.3 Complexité spatiale non déterministe	27
2.5.4 Les preuves interactives	28
3 Techniques de conception d’algorithmes	29
3.1 Analyse et preuves d’algorithmes	29
3.1.1 Parcours DFS	30
3.1.2 Parcours BFS	31
3.1.3 Algorithme de Dijkstra	32
3.2 Algorithmes glouton	34
3.2.1 Arbre couvrant de poids minimum	35
3.2.2 Matroïdes	37
3.3 Algorithmes récursifs	38
3.4 Programmation dynamique	41
3.4.1 Plus longue sous-séquence commune	42
3.4.2 Programmation dynamique dans les arbres	43

3.4.3	Problème du sac à dos	44
3.5	Flots et applications	45
3.5.1	Réseaux de transport	45
3.5.2	Algorithme de Ford-Fulkerson	46
3.5.3	Application au couplage	49
3.6	Programmation linéaire	50
3.6.1	Définition	50
3.6.2	Programmation linéaire en nombres entiers	51
4	Algorithmes d'approximation	51
4.1	Définition	52
4.1.1	2-approximation de la couverture sommet minimum	52
4.1.2	La classe APX	53
4.2	Approximation « gloutonne »	54
4.2.1	Couverture-sensemble	54
4.2.2	Couverture-ensemble pondérée	55
4.3	Approximation à partir d'arbres couvrant minimaux	56
4.3.1	Problème du voyageur de commerce	56
4.3.2	Inapproximabilité	57
4.3.3	Le voyageur de commerce métrique	57
4.3.4	Arbres de Steiner	59
4.4	Algorithmes d'élagage	59
4.5	Les schémas d'approximation	60
4.5.1	Définition	60
4.5.2	L'exemple du sac à dos	60
4.5.3	Les classes FPTAS, PTAS et APX-complet	61
4.6	Méthode de l'arrondi	61
5	Algorithmes paramétrés	62
5.1	La classe FPT	63
5.1.1	Algorithmes polynomiaux à paramètre fixé	63
5.1.2	Arbre de recherche pour COUVERTURE SOMMET	64
5.1.3	La classe FPT	65
5.2	Les noyaux : l'exemple de MAXSAT	66
5.3	Algorithmes de branchement	67
5.3.1	Algorithme paramétré de branchement pour COUVERTURE SOMMET	67
5.3.2	Analyse des algorithmes de branchement	68
5.4	Classes de complexité paramétrée	68
5.4.1	Le problème SAT contraint	68
5.4.2	Formules booléennes t -normalisées	69
5.4.3	Les classes $w[t]$	69

6 Algorithmes probabilistes	69
6.1 Analyse probabiliste	69
6.1.1 Rappels élémentaires de la théorie des probabilités	69
6.1.2 Exemple de la diffusion	71
6.1.3 Analyse d'algorithmes	73
6.2 Las Vegas et Monté Carlo	74
6.2.1 Un algorithme Las Vegas pour la recherche de répétitions	74
6.2.2 Un algorithme de Monté Carlo pour MAXSAT	74
6.2.3 Echantillonnage probabiliste	75
6.3 Méthode de l'arrondi probabiliste	75
6.4 Concentration : bornes de Chernoff	76
6.4.1 Borne de Chernoff	77
6.4.2 Application à la diffusion probabiliste	77
6.5 Les classes de complexité probabiliste	77
7 Conclusion	78
A Correction des exercices	80
B Devoir à la maison	86
C Examen du 29 janvier 2010, 11h30-13h00	89
D Examen du 21 janvier 2011, 09h30-11h00	92
E Examen du 29 octobre 2011, 09h00-11h00	95
F Examen de rattrapage 8 février 2012, 10h00-11h00	98
G Examen du 30 octobre 2012, durée 3h	100
H Examen de rattrapage 27 février 2013, durée 1h30	104
I Examen du 18 octobre 2013, durée 3h	106
J Examen de rattrapage 27 février 2014, durée 1h30	109

Introduction

L’algorithmique est la discipline de l’informatique traitant de la conception et l’analyse d’algorithmes. A ce titre, l’algorithmique se concentre principalement sur des problèmes abstraits dont la résolution permet de mettre en évidence de grands principes pouvant ensuite être déclinés pour la résolution de problèmes spécifiques issus d’applications pratiques. L’algorithmique se distingue de la programmation au sens où l’algorithmique permet de concevoir des algorithmes à un certain niveau d’abstraction, susceptibles ensuite d’être programmés dans n’importe quel langage de programmation. La traduction d’un algorithme en programme nécessite néanmoins de tenir compte du language de programmation, de l’environnement d’exécution du programme, de ses entrées, de ses sorties, etc., toutes choses qui ne sont pas au centre des intérêts de l’algorithmique. L’exemple de l’algorithme PageRank, au cœur de nombreux moteurs de recherche, illustre la différence entre algorithmique et programmation. En effet, autant il est relativement aisés de comprendre les principes de base de PageRank, voire de décrire PageRank sous forme d’un algorithme relativement compact, autant le passage de la description algorithmique de PageRank à la conception d’un moteur de recherche nécessite un travail considérable.

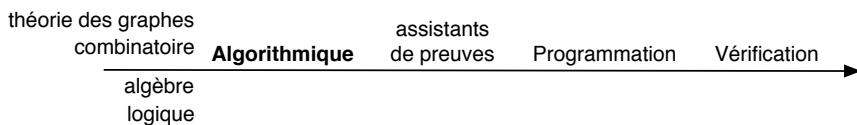


FIGURE 1 – *L’algorithmique au sein de la chaîne de conception de logiciels*

La figure 1 illustre la position de l’algorithmique au sein de la chaîne de conception de logiciels. La conception et l’analyse d’algorithmes se nourri de résultats et de concepts à l’interface entre mathématique et informatique. Des concepts issus de l’algorithmique, il est possible de dériver des programmes, soit manuellement, soit via l’utilisation d’outils tels que les assistants de preuves, ces derniers permettant à la fois de concevoir un programme et d’en générer une preuve de correction. Une fois le programme conçu, il convient ensuite de le vérifier, soit statiquement, soit à l’exécution.

L’algorithmique est donc essentielle à la conception de logiciels efficaces. Sans algorithmique, point de PageRank, et donc point de Google. Ce document effectue un survol des résultats algorithmiques les plus importants. Il débute par des rappels de calculabilité et de complexité, et se poursuit par la description de techniques de conception et d’analyse d’algorithmes. Il aborde les algorithmes d’approximation, les algorithmes paramétrés, et les algorithmes probabilistes.

1 Préambule : les graphes et PageRank de Google

1.1 Des objets au cœur de l’algorithmique : les graphes

Ce cours utilisera abondamment la notion de *graphe*, un des modèles les plus utilisés pour formaliser les problèmes réels. Par exemple, Internet peut être vu comme un graphe connectant des routeurs, et le Web comme un graphe connectant des pages html. Les interactions moléculaires se modèlent également comme un graphe, tout comme les relations entre concepts linguistiques ou les relations entre les espèces dans la théorie de l’évolution, et caetera. On pourrait multiplier les exemples à l’infini.

Rappelons donc qu’un graphe G est défini comme une paire (V, E) où V est l’ensemble

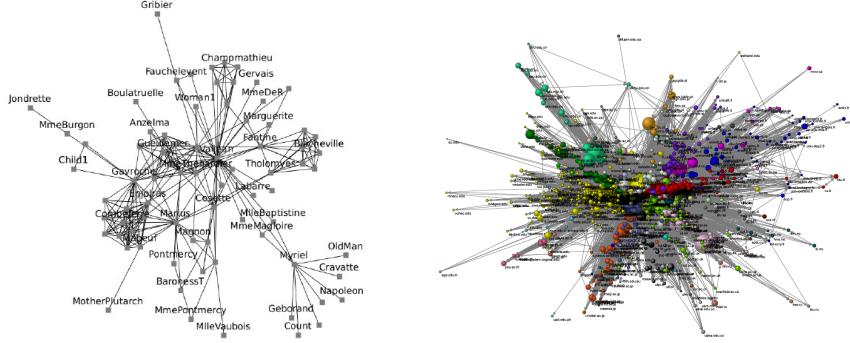


FIGURE 2 – Gauche : graphe relatif à des concepts issus de la linguistique (cf. N. Villa-Vialaneix et al., 2011). Droite : une partie du graphe du Web (cf. Ortega et Aguillo, 2009).

(généralement fini) des *sommets* et E est l’ensemble des *arêtes*. Une arête est une paire $\{u, v\}$ de sommets. Si une arête est orientée, de u vers v , on parle alors d’un *arc*, et on le note par une paire ordonnée (u, v) . Un graphe (non orienté) est donc défini par ses sommets et ses arêtes notées par des paires non ordonnées $\{u, v\}$ de sommets, et un graphe orienté est défini par ses sommets et ses arcs notés par des paires ordonnée (u, v) de sommets.

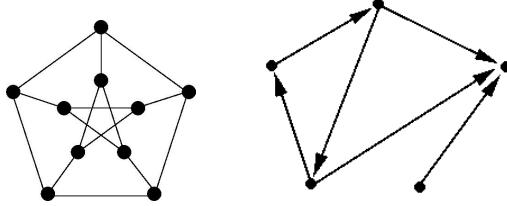


FIGURE 3 – Gauche : le graphe (non orienté) de Petersen. Droite : un graphe orienté

Les *cycles* forment une famille de graphes très utilisée en algorithmique : un cycle à n sommets, C_n , $n \geq 3$, est défini par

$$V(C_n) = \{0, 1, \dots, n - 1\} \text{ et } E(C_n) = \{\{i, i + 1 \bmod n\}, i = 0, \dots, n - 1\}.$$

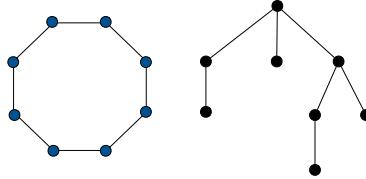


FIGURE 4 – Gauche : un cycle. Droite : un arbre

Les *arbres* forment une autre famille de graphes très utilisée en algorithmique : un arbre est un graphe connexe (on peut aller de tout sommet à tout autre sommet en traversant des arêtes) sans cycle (il n’est pas possible de traverser une suite d’arêtes et de revenir au sommet d’où l’on est parti sans traverser une même arête plus d’une fois). Un sous-graphe de $G = (V, E)$ est un graphe $G' = (V', E')$ où $V' \subseteq V$ et $E' \subseteq E$. Un *arbre couvrant* d’un graphe connexe $G = (V, E)$ est un sous graphe $T = (V, E')$ de G qui est un arbre. Notez qu’un arbre peut se définir de plusieurs façons. En particulier, les définition suivantes sont équivalentes :

- T est un arbre de n sommets ;
- T est un graphe connexe sans cycle de n sommets ;
- T est un graphe connexe de n sommets et $n - 1$ arêtes ;
- T est un graphe sans cycle de n sommets et $n - 1$ arêtes.
- T est un graphe sans sommet isolé (i.e., tout sommet est incident à au moins une arête) de n sommets et $n - 1$ arêtes.

Le *degré* $\deg(u)$ d'un sommet u est le nombre d'arêtes incidentes à ce sommet. Ainsi, tous les sommets ont degré 2 dans un cycle. Un graphe dont tous les sommets ont même degré est dit *régulier*. Tous les cycles, ainsi que le graphe de Petersen sont donc réguliers. Notez également que, dans un graphe $G = (V, E)$, on a

$$\sum_{u \in V} \deg(u) = 2|E|$$

car chaque arête contribue de 1 aux degrés de ses deux extrémités. Il en découle que $|E| \leq \frac{n(n-1)}{2}$ car chaque sommet a un degré au plus $n - 1$. (Dans ce cours, tous les graphes considérés sont *simple*, c'est-à-dire qu'il n'y a au plus qu'une arête entre deux sommets, et *sans boucle*, c'est-à-dire qu'il n'y a pas d'arête connectant un sommet à lui-même). Un *chemin* est un arbre dont tous les sommets ont degré au plus 2. Un graphe connexe est donc un graphe tel que, pour toute paire de sommets $s \neq t$, il existe un chemin P dans G d'extrémité s et t , c'est-à-dire

$$P = (v_0, v_1, v_2, \dots, v_r)$$

où $v_0 = s$, $v_r = t$, et $\{v_i, v_{i+1}\} \in E(G)$ pour tout $i = 0, \dots, r - 1$. Un graphe *pondéré* est un graphe $G = (V, E)$ tel que chaque arête $e \in E$ à un *poids* $\omega(e) \geq 0$. Si ce poids est interprété comme une longueur ℓ alors la longueur de P est

$$\ell(P) = \sum_{i=0}^{r-1} \ell(\{v_i, v_{i+1}\}).$$

La distance $\text{dist}(u, v)$ entre deux sommets s et t de G est la plus petite longueur d'un chemin entre s et t dans G . Dans un graphe non pondéré, on suppose la longueur de chaque arête égale à 1, et la distance entre deux sommets s et t est donc le nombre d'arêtes du plus court chemin d'extrémités s et t . Le *diamètre* d'un graphe $G = (V, E)$ est $D = \max_{u,v \in V} \text{dist}(u, v)$.

Le cours ne nécessite pas de connaissances approfondies en théorie des graphes. Tout étudiant souhaitant en savoir plus sur cette théorie féconde est néanmoins invité à consulter [4].

1.2 Codage et notations asymptotiques

Il existe plusieurs façons de coder un graphe G de n sommets. On peut le faire par sa matrice d'*adjacence* booléenne M de taille $n \times n$. A cette fin, on numérote les sommets de 1 à n , et $M[i, j] = 1$ si et seulement si il existe une arête entre les sommets i et j (ou, dans le cas orienté, s'il existe un arc de i vers j). On peut aussi coder le même graphe G par ses listes d'adjacence L_i , $i = 1, \dots, n$, où L_i est la liste de tous les sommets voisins du sommet i . On peut enfin coder G par sa matrice d'*incidence* M . Pour un graphe de n sommets et m arêtes, cette matrice est une matrice booléenne $n \times m$ telle que $M[i, j] = 1$ si et seulement si le sommet i est incident à l'arête j . Ces codages sont équivalents à une réduction en temps polynomial près. Voici par exemple une réduction du codage en listes d'adjacence au codage en matrice d'adjacence.

```

Algorithme matrix2list( $M$ ) :
début
     $n \leftarrow \text{dimension}(M)$ 
    pour  $i = 1$  à  $n$  faire
         $k \leftarrow 1$ 
        pour  $j = 1$  à  $n$  faire
            si  $M[i, j] = 1$  alors
                 $L_i[k] \leftarrow j$ 
                 $k \leftarrow k + 1$ 
            fin si
         $L_i[k] \leftarrow \perp$  /* symbole de fin de liste */
    retourner  $L = (L_1, L_2, \dots, L_n)$ 
fin

```

Notations asymptotiques. Afin de discuter des avantages et inconvénients de ces deux codages de graphes, on rappelle que la notation $f = O(g)$ indique qu'il existe $c > 0$ et x_0 tel que $f(x) \leq c \cdot g(x)$ pour tout $x \geq x_0$. De même, $f = \Omega(g)$ indique qu'il existe $c > 0$ et x_0 tel que $f(x) \geq c \cdot g(x)$ pour tout $x \geq x_0$, et $f = \Theta(g)$ indique que $f = O(g)$ et $f = \Omega(g)$.

Le codage d'un graphe par liste utilise une mémoire de $\Theta(m \log n)$ bits pour un graphe de m arêtes, ce qui peut être significativement plus petit que l'espace $\Theta(n^2)$ bits nécessaire au codage par matrice d'adjacence, pour un graphe peu dense. Cependant vérifier si deux sommets sont voisins réclame de chercher dans une liste, ce qui peut être coûteux en temps (de l'ordre de la longueur de la liste), surtout si la liste n'est pas triée.

1.3 Matrices et graphes

La matrice d'adjacence M d'un graphe fournit quantité d'informations sur ce graphe (cf. la théorie algébrique des graphes). En particulier, de simples manipulations matricielles permettent d'effectuer des opérations complexes sur les graphes et de déduire des propriétés de ces graphes. Par exemple, l'existence d'un chemin de longueur ℓ de i à j dans un graphe se traduit par l'existence d'une valeur positive à la position (i, j) de la matrice M^ℓ . (Cette valeur est de fait égale au nombre de chemins de longueur ℓ entre i et j .)

Pagerank de Google. Les procédures mises en oeuvre par Google pour classer les pages web contenant les mots clés relatifs à une requête à ce système ne sont pas publiques. En revanche, on sait qu'elles sont bâties autour de l'algorithme Pagerank. Ce dernier cherche à simuler le comportement d'un surfeur du web passant au hasard de page en page. La probabilité pour ce surfeur de se retrouver sur une page spécifique est d'autant plus grande que cette page est référencée par un grand nombre de pages elles mêmes référencées par un grand nombre de pages, etc. Cette propriété est capturée par la notion de *marche aléatoire* dans un graphe, consistant à quitter le sommet courant par un arc incident choisi uniformément aléatoirement parmi les arcs sortants de ce sommet. Une telle marche correspond à un *processus Markovien* puisque la probabilité $p_t(i)$ que le surfeur soit en un sommet i au temps t ne dépend que de la position du surfeur au temps précédent.

Plus spécifiquement, si M désigne la matrice d'adjacence du graphe, on définit la matrice de transition Q par $Q = D^{-1}M$ où D est la matrice diagonale définie par $D_{i,i} = \sum_{j=1}^n M_{i,j}$. La matrice de transition n'est donc rien d'autre que la matrice d'adjacence normalisée par le degré de chaque sommet, de façon à ce que $\sum_{j=1}^n Q_{i,j} = 1$ pour tout $i = 1, \dots, n$. Par définition de la

marche aléatoire, on obtient

$$p_{t+1} = Q^\top p_t$$

où Q^\top dénote la transposée de Q . Sous certaines conditions sur M (et donc sur Q), cette chaîne de Markov correspondant à la marche aléatoire dans le graphe G correspondant à M admet une distribution stationnaire p^* où p_i^* indique la probabilité stationnaire d'être au sommet i . Le vecteur p^* satisfait

$$p^* = Q^\top p^*.$$

Il est donc possible de calculer p^* comme point d'attraction par itération $q^{(k+1)} = Q^\top q^{(k)}$ en partant d'un vecteur $q^{(0)}$ proche de p^* . C'est ce que fait Pagerank sur le graphe du web : chaque page est un sommet du graphe, et il y a un arc d'une page à une autre si la première contient un lien hypertexte vers la seconde. Si p^* est la distribution stationnaire alors la page i obtient un rang proportionnel à la valeur de p_i^* .

L'immense difficulté de la tâche se cache dans la mise en oeuvre de cette technique, qui requiert tout un savoir faire. Pagerank repose en effet sur la manipulation d'un graphe de plusieurs centaines de millions de sommets et de plusieurs milliards de liens qu'il convient potentiellement de modifier afin de satisfaire aux hypothèses de convergence et d'assurer une convergence rapide. En pratique, Pagerank effectue quelques itérations de $q^{(k+1)} = Q^\top q^{(k)}$.

On pourrait exprimer $q^{(k+1)} = Q^\top q^{(k)}$ sous la forme $q^{(k+1)} = (Q^\top)^k q^{(0)}$. Effectuer un produit matrice-matrice n'est toutefois en rien trivial. Pour information :

- L'algorithme standard de produit matriciel a une complexité $O(n^3)$ car chacune des n^2 entrée i, j de la matrice produit est calculée par le produit scalaire de la i ème ligne par la j ème colonne, et ce produit scalaire implique n produits et $n - 1$ additions.
- L'algorithme dû à Strassen (1969), basé sur un produit de matrices 2×2 n'utilisant que sept multiplications au lieu de huit, a une complexité $O(n^{\log_2 7}) = O(n^{2.81})$.
- Un des meilleurs algorithmes connus, dû à Coppersmith et Winograd (1990), a une complexité $O(n^{2.376})$, amélioré en 2012 par Stothers puis Williams en $O(n^{2.373})$.

Il est conjecturé que le produit matriciel pourrait s'exécuter en $O(n^2 \log^c n)$ avec c constant. Si cette conjecture s'avère vraie, alors un produit matrice-matrice ne serait guère plus couteux qu'un produit matrice-vecteur (qui nécessite un temps $\Omega(n^2)$ puisqu'il faut bien lire toutes les entrées de la matrice). On note ω l'infimum de l'ensemble des $\alpha > 0$ pour lesquels il existe un algorithme de multiplication de matrice de complexité $O(n^\alpha)$.

1.4 Composantes connexes

Comme évoqué précédemment, on dit qu'un graphe $G = (V, E)$ est *connexe* s'il existe un chemin entre toute paire de sommets, c'est-à-dire si, pour tout $\{u, v\} \in V^2$, $u \neq v$, il existe une suite x_0, x_1, \dots, x_k de sommets, telle que $x_0 = u$, $x_k = v$, et $\{x_i, x_{i+1}\} \in E$ pour tout $i = 0, \dots, k - 1$. Si un graphe n'est pas connexe, il peut être décomposé en plusieurs *composantes connexes*, où une composante connexe désigne un ensemble C de sommets, maximal pour l'inclusion, tel que le sous-graphe de G induit par C , noté $G[C]$, est connexe. Ces définitions se généralisent aux cas des graphes orientés. En ce cas, le chemin x_0, x_1, \dots, x_k entre u et v doit respecter l'orientation des arcs, c'est-à-dire $(x_i, x_{i+1}) \in E$ pour tout $i = 0, \dots, k - 1$. On introduit alors la notion de connexité *forte* : un graphe orienté $G = (V, E)$ est fortement connexe si pour toute paire $\{u, v\}$ de sommets, il existe un chemin de u à v , et un chemin de v à u . Une composante fortement connexe est un ensemble C de sommets, maximal pour l'inclusion, tel que le sous-graphe de G induit par C , noté $G[C]$ est fortement connexe.

2 Calculabilité et complexité algorithmique

La *calculabilité* est le domaine scientifique traitant de la capacité de résoudre un problème algorithmiquement. La *complexité* est le domaine scientifique traitant de la difficulté des problèmes, en les classant dans différentes *classes* définies selon différents critères d'efficacité (temps, mémoire, etc.), et en étudiant les relations entre ces classes. Cette partie du cours décrit les bases des théories de la calculabilité et de la complexité, ainsi que deux classes essentielles : P et NP. D'autres classes de complexité seront introduites au fur et à mesure de l'avancée du cours.

2.1 Machine de Turing

Pour parler de la difficulté à calculer tel ou tel résultats ou à résoudre tel ou tel problème, il convient de se mettre d'accord sur ce que veut dire « calculer ». Alan Turing (1937) a défini une machine abstraite, appelée depuis *machine de Turing*, permettant de donner un sens précis à la notion de calcul.

Informellement, une machine de Turing est constituée d'un ruban (ou d'une bande) formé(e) d'un nombre infini de cases auxquels accède une tête de lecture et d'écriture. Intuitivement, le ruban modélise la mémoire d'un ordinateur, et la tête de lecture modélise le processeur. Initialement, la tête de lecture est supposée positionnée en face du premier caractère du ruban infini dans un sens. Les données d'entrée x de la machine sont codées sur les $|x|$ premières caractères de la bande, où $|x|$ dénote le nombre de caractères utilisés pour coder la donnée x . A chaque étape de calcul, le symbole sous la tête de lecture est lu. La machine change alors potentiellement d'état, et écrit un nouveau symbole sur la bande en lieu et place du symbole lu. La tête de lecture se déplace ensuite d'un symbole vers la droite ou vers la gauche. Le nouvel état de la machine, le symbole écrit, et le déplacement dépendent du symbole lu et de l'état courant de la machine lorsque le symbole est lu. Les données de la machine sont décrites par une suite de symboles initialement écrits en début de ruban. Lorsque la machine entre dans un état final spécifié, elle s'arrête, et le résultat du calcul est l'ensemble des symboles écrits sur la bande. Je vous invite à taper « *the lego Turing machine Youtube* » dans Google pour observer une machine de Turing en action !

2.1.1 Définition

La définition formelle d'une machine de Turing est la suivante (voir aussi la figure 5) :

Definition 1 Une machine de Turing est un quintuplet $M = (Q, \Gamma, \delta, q_0, F)$ où Q dénote l'ensemble fini des états de la machine, Γ l'ensemble fini de symboles pouvant être écrit sur la bande de la machine (incluant le symbole « blanc » pouvant apparaître infiniment sur la bande de la machine), $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1, +1\}$ est la fonction de transition, q_0 est l'état initial, et $F \subseteq Q$ est l'ensemble des états finaux.

Exercice 1 Décrire une machine de Turing M qui, étant donné un entier $x \in \mathbb{N}$ quelconque écrit sur le ruban, décide si x est pair ou non. (Dans le premier cas, M doit terminer dans l'état « oui », et, dans le second cas, M doit terminer dans l'état « non »).

Il convient de noter que la machine de Turing n'est pas la seule façon de donner une définition formelle au calcul. Ainsi, le λ -calcul, introduit par Alonzo Church dans les années 30 permet

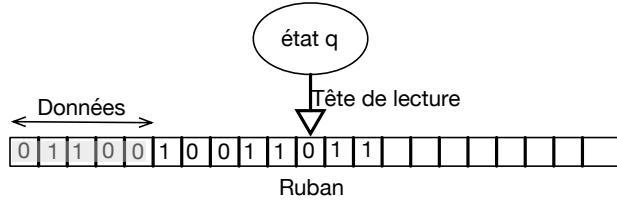


FIGURE 5 – Une machine de Turing. Dans cet exemple, il est indiqué que le ruban contient initialement la donnée 01100. La tête de lecture se situe en position 11 du ruban, et lit 0. Si l'état de la machine est dans l'état $q \in Q$, elle calcule le triplet $(q', x, d) = \delta(q, 0)$. Cela correspond au fait que la machine passe dans l'état q' , écrit x sur le ruban en position courante (c'est-à-dire la position 11), et se déplace à droite (si $d = +1$) ou à gauche (si $d = -1$).

aussi de formaliser la notion de calcul, tout comme les *fonctions récursives* dues à Kleene, ou les *automates cellulaires*. La machine de Turing peut paraître limitée et primitive. Elle se révèle au contraire d'une grande richesse, certes difficile à percevoir lorsque l'on est confronté à cet objet pour la première fois. En fait, l'hypothèse, ou *thèse*, de Church-Turing, exprimée dans les années 40, conjecture que la machine de Turing capture exactement la notion de calcul.

Thèse de Church-Turing : les machines de Turing formalisent correctement la notion de méthode effective de calcul.

Informellement, cette thèse énonce que tout calcul fondé sur une méthode effective peut être effectué par une machine de Turing. La notion de thèse fait ici référence au fait que l'adjectif « effective » est vague. Ce qu'elle prétend est que, quelque soit la façon d'imaginer un calcul, ce dernier pourra être effectué par une machine de Turing. Depuis plus de 70 ans, cette thèse n'a pas été remis en cause. Notez que la thèse de Church-Turing reste vraie par exemple pour des ordinateurs quantiques. Ces derniers (encore hypothétiques physiquement à grande échelle, mais effectifs en terme de modèle de calcul) sont plus rapides que les machines de Turing, mais il ne calculent pas plus de choses que ce que peuvent calculer ces dernières¹.

Machines universelles. La puissance d'expressivité des machines de Turing se révèle également au travers de la notion de machines de Turing *universelles*. Turing a démontré qu'il existait des machines de Turing $M = (Q, \Gamma, \delta, q_0, F)$ pouvant simuler les calculs de toutes machines de Turing. Informellement, pour que M puisse simuler M' il s'agit de montrer comment encoder le quintuplet $M' = (Q', \Gamma', \delta', q'_0, F')$ décrivant M' sur le ruban de M . En fait c'est exactement ce que fait un ordinateur : le programmeur rentre un programme et des données, et le programme est exécuté sur les données. Mais la notion de machine de Turing universelle dit plus : elle montre qu'il n'y a pas de différence formelle entre données et programme. Les deux sont en fait des données pour une machine de Turing universelle.

2.1.2 Indécidabilité

Existe-t-il une méthode algorithmique permettant de résoudre tous les problèmes ? Non ! Il existe en effet des problèmes qui ne sont pas *décidables*, c'est-à-dire pour lesquels il n'existe pas

1. Remarquons toutefois que les ordinateurs quantiques sont capables générer des nombres purement aléatoires, ce qui est au delà de ce que peut faire un ordinateur classique. Néanmoins, cela ne contredit pas la thèse de Church-Turing puisque la génération d'un nombre aléatoire n'est pas considéré comme un calcul.

de machine de Turing permettant de les résoudre. Par exemple, étant donnés deux présentations finies de groupes, décider si ces deux présentations correspondent à deux groupes isomorphes est indécidable. Attention, ce n'est pas la puissance de la machine de Turing qui est à mettre en cause pour son incapacité à résoudre certains problèmes (voir la thèse de Church-Turing). L'existence de problèmes indécidables est intrinsèque à l'informatique. Le fait qu'il existe des problèmes pour lesquels il n'existe pas de machine de Turing, et donc pas d'algorithmes permettant de les résoudre indique une limitation de la capacité intrinsèque à calculer. Cette limitation s'applique aussi bien aux ordinateurs qu'à n'importe quel façon de calculer (dont par exemple notre cerveau, si la thèse de Church-Turing est vraie).

Montrer l'existence d'un langage indécidable, c'est-à-dire tel que aucune machine de Turing peut le décider est une conséquence directe du théorème de Cantor (1891) stipulant que l'ensemble $\mathcal{P}(\mathbb{N})$ des parties de \mathbb{N} est d'une cardinalité strictement supérieure à celle des entiers \mathbb{N} . (En effet, on peut énumérer les machines de Turing, par exemple en les classant par ordre alphabétique de leur représentation en binaire, alors que l'ensemble des problèmes est en bijection avec $\mathcal{P}(\mathbb{N})$ – nous verrons pourquoi cela un peu plus loin dans ce document). L'originalité et l'intérêt du théorème ci-dessous est d'exhiber un tel langage. Un problème central de l'informatique consiste, étant donné un programme quelconque (au sens machine de Turing), à déterminer s'il finira par s'arrêter ou non. Ce problème est appelé *problème de l'arrêt*.

Théorème 1 (Turing, 1936) *Etant donnée une machine de Turing M quelconque et une entrée x quelconque, déterminer si M s'arrête ou pas sur l'entrée x est indécidable.*

Dit autrement, il n'existe pas d'algorithme **halt** qui, pour toute machine de Turing M et pour toute entrée x de cette machine, retourne « oui » si $M(x)$ s'arrête, et « non » sinon.

Il existe de nombreuses preuves de ce théorème. Le sketch de la preuve ci-dessous est la preuve classique par argument « diagonal » dont les grandes lignes sont semblables à celles de la preuve d'indénombrabilité des réels (Cantor, 1891) et du théorème d'incomplétude de Gödel (1931). Pour mémoire, le théorème de Gödel annonce qu'une théorie mathématique suffisante pour y exprimer l'arithmétique est nécessairement incomplète, au sens où il existe des énoncés qui n'y sont ni démontrables, ni réfutables. Le théorème de Turing est une forme « mécanique » du théorème de Gödel. Notez toutefois que tout énoncé vrai « M s'arrête sur x » est démontrable (en donnant le nombre d'étapes auxquels la machine s'arrête), et tout énoncé faux « M ne s'arrête pas sur x » est réfutable (en donnant le nombre d'étapes auxquels la machine s'arrête). De fait, le théorème de Turing n'affirme pas qu'il existe une machine M et une entrée x pour laquelle l'énoncé « M s'arrête sur x » n'est ni démontrable ni réfutable, mais affirme qu'il n'existe pas de protocole mécanique permettant de décider « M s'arrête sur x » pour toute paire (M, x) . Enfin, le théorème de Turing est explicite au sens où il présente une propriété spécifique qui est non décidable, au lieu de simplement montrer l'existence d'une telle propriété.

Le sketch de la preuve du théorème de Turing ci-dessous repose sur l'auto-référence, qui conduit souvent à des paradoxes. A titre d'exemple, considérer l'énoncé suivant « cette phrase est fausse ». Si cette phrase est effectivement fausse, alors elle est vraie, ce qui impossible, et si cette phrase est vraie, alors elle ne peut être fausse. C'est ce genre de technique qu'utilise Gödel pour démontrer son théorème d'incomplétude, en montrant comment formuler une version du paradoxe du menteur dans toute théorie mathématique incluant l'arithmétique. C'est également ce que nous allons faire ci-dessous en exécutant un programme sur lui-même.

Sketch de la preuve du théorème de Turing. Supposons par l'absurde qu'il existe un algorithme **halt** tel que $\text{halt}(M, x)$ retourne 1 si $M(x)$ s'arrête, et 0 sinon. Nous avons vu qu'il

n'existe pas de différence formelle entre programmes et données. Nous pouvons donc construire le programme **diag** prenant en entrée une chaîne de caractères x , comme suit :

```
Algorithme diag( $x$ ) :
début
    si halt( $x, x$ ) = 1 alors
        tant que vrai faire  $b \leftarrow 0$  /* boucle infinie */
    sinon stop
fin
```

Le programme ci-dessus n'est rien d'autre qu'une chaîne de caractères, **diag**. On peut donc étudier la valeur de **halt**(**diag**, **diag**) :

- Si **halt**(**diag**, **diag**) = 1 alors, par définition de **halt**, le programme **diag** s'arrête sur le chaîne de caractères **diag**. Cela est en contradiction avec la définition de **diag** qui boucle indéfiniment sur une chaîne de caractère x telle que **halt**(x, x) = 1.
- Si **halt**(**diag**, **diag**) = 0 alors, par définition de **halt**, le programme **diag** ne s'arrête pas sur le chaîne de caractères **diag**. Cela est en contradiction avec la définition de **diag** qui s'arrête immédiatement sur une chaîne de caractère x telle que **halt**(x, x) = 0.

Les deux cas conduisant à une contradiction, il ne peut donc pas exister de programme **halt**. \square

En d'autres termes, le théorème de l'arrêt indique qu'il n'existe pas de méthode générique permettant de prouver qu'un programme est correcte. Ainsi, vérifier si le programme de contrôle d'une centrale nucléaire est correct est une tâche extrêmement ardue, voire potentiellement impossible si ce programme n'offre pas des caractéristiques spécifiques. Un moyen de prouver des programmes consiste à restreindre l'expressivité du langage de programmation (cf. certaines versions du λ -calcul). Ainsi, la correction des programmes contrôlant les métros automatiques repose soit sur des procédures de vérifications formelles de programmes peu expressifs, soit sur des techniques spécifiques du programme à vérifier.

Notons que le problème de l'arrêt de la machine de Turing, ou celui de l'isomorphisme de groupes, peuvent sembler abstraits, et on peut croire que l'indécidabilité de ces problèmes provient de ce caractère abstrait. Il n'en est rien, et des problèmes très simples peuvent se révéler indécidables ! Par exemple, le problème suivant est indécidable :

Le problème PRODUIT MATRICE NUL :

Entrée : un ensemble S de six matrices 3×3 à coefficients rationnels ;

Question : peut-on obtenir la matrice nulle par produit de matrices de S entre elles (avec répétition) ?

Notons que le problème PRODUIT MATRICE NUL est décidable pour deux matrices 2×2 (avec un algorithme déjà non trivial). La décidabilité du problème pour trois matrices 2×2 n'est pas connu, ni pour deux matrices 3×3 . En revanche, le problème est indécidable pour deux matrices 15×15 . Notons que dans tous ces cas, la longueur de la séquence de produits permettant d'obtenir la matrice nulle (si une telle séquence existe) peut être arbitrairement longue.

2.2 Machines RAM

Si la machine de Turing donne un cadre formel à la théorie de la complexité, ce n'est généralement pas avec ce modèle que l'on analyse les algorithmes ou que l'on juge au quotidien de la difficulté d'un problème. On utilise plutôt un autre modèle, d'eplus haut niveau et plus

malléable : le modèle RAM, pour *random-acces machine*. Ce modèle est suffisant pour juger de la difficulté d'un problème, à une fonction polynomiale près de la taille des données du problème.

2.2.1 Les modèles de machines RAM

Le modèle RAM inclut :

- une *mémoire* infinie dont chaque cellule peut contenir un entier,
- un nombre fini de *registres* pouvant contenir un entier, et
- une *unité de calcul* exécutant une suite finie d'instructions élémentaires.

L'instruction effectuée à chaque étape de calcul dépend de l'état des registres (c'est-à-dire de leur contenu), ce qui inclu le compteur de programme. Une telle instruction peut consister en charger le contenu d'une cellule mémoire dans un registre, ou écrire le contenu d'un registre dans une cellule mémoire. Le modèle RAM permet également l'utilisation des instructions arithmétiques élémentaires $+, -, *, /$, et de comparaisons $<, =, >$, entre contenus de registres. Il permet également d'effectuer toutes les instructions usuelles comme les branchements conditionnels (si/alors/sinon), les boucles (tant que/faire, pour/faire, faire/jusqu'à), ainsi que l'adressage indirect (c'est-à-dire l'accès à une cellule dont la position est décrite par le contenu d'un registre).

On distingue alors deux types de machines RAM :

1) Le modèle de la machine RAM *uniforme* suppose que chacune des opérations arithmétiques listées ci-dessus prend un temps 1. Comme tel, ce modèle est considéré trop puissant, principalement parce qu'il est capable d'effectuer des opérations arithmétiques sur des entiers de taille arbitrairement grande en temps constant. De fait, on ne connaît pas de simulation en temps polynomial de la machine RAM uniforme par une machine de Turing. (En revanche, si l'on se restreint aux opérations arithmétiques $+$ et $-$, alors il est possible de montrer que la machine RAM uniforme est équivalente à la machine de Turing par simulations réciproques en temps polynomial).

2) Le modèle de la machine RAM *logarithmique* suppose que chaque opération arithmétique $+, -, *, /$ et de comparaison $<, =, >$ prend un temps proportionnel au logarithme de la taille des entiers impliqués dans l'opération. Ce modèle est équivalent à la machine de Turing par simulations réciproques en temps polynomial. C'est-à-dire tout programme RAM s'exécutant en temps $t(n)$ peut être simulé par une machine de Turing s'exécutant en temps $O(t(n)^c)$ pour $c > 0$ constant.

Exemple. Considérons les deux algorithmes suivants :

```

Algorithme puissance( $n$ ) :
début
     $y \leftarrow 1$ 
    pour  $i = 1$  à  $n$  faire  $y \leftarrow 2y$ 
fin
```

Ce premier algorithme calcule $y = 2^n$. Le second algorithme ci-dessous calcule $y = 2^{2^n}$.

```

Algorithme puissance2( $n$ ) :
début
     $y \leftarrow 2$ 
    pour  $i = 1$  à  $n$  faire  $y \leftarrow y^2$ 
fin
```

Chacun de ces deux algorithmes effectue $O(n)$ opérations, dont n multiplications. Ils ont donc

tous les deux une complexité $O(n)$ dans le modèle RAM uniforme. L'algorithme **puissance** manipule des entiers sur $O(n)$ bits. Sa complexité dans le modèle RAM logarithmique est donc de $O(n \log n)$. En revanche, l'algorithme **puissance2** manipule des entiers sur $O(2^n)$ bits. Sa complexité dans le modèle RAM logarithmique est donc de $O(n^2)$.

2.2.2 Le modèle du cours

Dans ce cours, nous adopterons le modèle usuel en algorithmique consistant à utiliser le modèle RAM uniforme (afin d'éviter la présence continue de facteurs logarithmiques dans le temps d'exécution des algorithmes) pour tous les algorithmes n'impliquant des opérations élémentaires que sur des entiers de taille polynomiale en leur valeur. Dans ce modèle, la complexité de l'algorithme **puissance** est $O(n)$. En revanche, pour analyser l'algorithme **puissance2**, on ne devra pas négliger le coût de la manipulation d'entiers sur un nombre exponentiel de bits, et il conviendra d'utiliser le modèle RAM logarithmique, plus précis, pour de tels algorithmes.

Un *algorithme* est donc une suite finie d'instructions RAM. On ne précisera pas nécessairement le codage des données en mémoire. Par exemple, l'algorithme **prod** suivant effectue le produit AB de deux matrices A et B quelconques $n \times n$, et stocke le résultat dans une matrice C :

```
Algorithme prod( $A, B$ ) :
début
    pour  $i = 1$  à  $n$  faire
        pour  $j = 1$  à  $n$  faire
             $C[i, j] \leftarrow 0$ 
            pour  $k = 1$  à  $n$  faire
                 $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
    fin
```

Le *temps d'exécution* d'un algorithme A sur une entrée x est le nombre d'instructions élémentaires $t(A, x)$ effectuées par la machine RAM exécutant A sur x . La complexité d'un algorithme s'exprime généralement en fonction de la taille des entrées. On s'intéresse ainsi à la fonction

$$t_A(N) = \max_{|x| \leq N} t(A, x)$$

et on cherche à en donner une borne asymptotique. Dans le cas de **prod**, on a un temps $O(n^3)$ puisque cet algorithme effectue de l'ordre de n^3 additions et multiplications. Notons que la taille N des données satisfait ici $N = \Theta(n^2)$, donc $t_{\text{prod}}(N) = O(N^{3/2})$. Comme mentionné précédemment, il est conjecturé qu'il existe un algorithme quasi linéaire pour le produit de matrice (c'est-à-dire en $O(n^2 \text{ polylog}(n))$ pour des matrices $n \times n$).

Codage des nombres. Remarquons que la machine RAM stocke des entiers $n \in \mathbb{N}$ dans les cellules mémoire et dans les registres. Il est possible de signer ces entiers en stockant leurs signes, et donc de manipuler des entiers relatifs $n \in \mathbb{Z}$. En revanche, la machine ne manipule pas des réels $r \in \mathbb{R}$, et il n'est donc pas possible d'utiliser des nombres transcendants tels que π dans un programme RAM². Néanmoins, il est tout à fait possible de manipuler des nombres rationnels $r \in \mathbb{Q}$ en stockant le rationnel $r = p/q$ sous la forme des deux entiers p et q , ou en approximant r en ne stockant que ses premières décimales après la virgule³. Dans ce cours, nous

2. Les ordinateurs personnels modernes manipulent des approximations de nombres réels, avec une certaine précision (cf. la norme IEEE 754).

3. Il est même potentiellement possible de stocker tout nombre algébrique en stockant son polynôme minimal $P(x) = \sum_{i=0}^k a_i x^i$ sous la forme de la suite des entiers relatifs a_0, a_1, \dots, a_k . Préserver une représentation

nous contenterons de manipuler principalement des entiers relatifs codés en binaire, et parfois des rationnels approximés par une représentation binaire tronquée à $\epsilon = 2^{-k}$ près. Par exemple

$$\frac{1}{3} = 0.010101010101\dots\dots = \sum_{i=1}^{+\infty} 2^{-2i}$$

sera approximé en $x = 0.010101 = \sum_{i=1}^{k/2} 2^{-2i}$ satisfaisant $|\frac{1}{3} - x| \leq \epsilon = \frac{1}{2^k}$.

2.3 Les classes P et NP

Un des objectif de la *complexité algorithmique* est de classer chaque problème selon sa difficulté, c'est-à-dire selon la complexité du meilleur algorithme permettant de résoudre ce problème. Une des classes les plus importantes est la classe P des problèmes pouvant se résoudre en temps polynomial en la taille des données. Cette classe est considérée comme celle des problèmes « facile », au sens où si la donnée est de taille N alors le temps pour résoudre le problème sur cette donnée sera au plus $O(N^c)$ avec $c \geq 0$. Bien sûr, si $c = 100$ alors il semble absurde de dire que le problème est facile. Toutefois, pour la très grande majorité des problèmes « naturels » de P, il existe un algorithme les résolvant s'exécutant en temps $O(N^c)$ avec c petit.

2.3.1 Les problèmes de décision

En fait, la complexité fait référence à des *problèmes de décision* relatifs à décider si un *mot* appartient à un *langage*. Soit Σ un alphabet fini, par exemple $\Sigma = \{0, 1\}$, et soit Σ^* l'ensemble de tous les mots finis sur cet alphabet (y compris le mot vide ϵ). Un langage est un sous-ensemble L de Σ^* . Un problème de décision s'exprime alors sous la forme suivante. Soit L un langage de Σ^* :

Le problème DÉCIDER L :

Entrée : un mot x de Σ^* ;

Question : $x \in L$?

Bien que cette présentation des problèmes puisse sembler très abstraite, il n'en est rien. Prenons par exemple le cas du problème PRODUIT MATRICE NUL introduit précédemment. Chaque matrice 3×3 nécessite 9 entiers. Une instance du problème consiste donc en 54 rationnels (ou entiers, une fois normalisés). Le langage L considéré dans le problème est celui des chaînes de 54 entiers, correspondant donc à un ensemble de six matrices 3×3 , telles qu'il existe une façon de combiner ces matrices par produit de façon à obtenir la matrice nulle.

A titre d'autre exemple, considérons le problème USTCON (pour *undirected s-t-connectivity*) qui, étant donné un graphe $G = (V, E)$ et deux sommets $s, t \in V$, doit décider s'il existe une chemin dans G entre s et t . Le langage considéré est celui des graphes G et des paires de sommets (s, t) pour lesquels il existe un chemin entre s et t . Si l'on code tout graphe $G = (V, E)$ de n sommets en numérotant ses sommets de 1 à n , et en utilisant une matrice d'adjacence M , c'est-à-dire $M_{i,j} = 1 \iff \{i, j\} \in E$, alors les données de USTCON sont une matrice booléenne $n \times n$, et deux entiers $s, t \in \{1, \dots, n\}$. Le langage L correspondant à USTCON peut être défini comme suit : $(G, s, t) \in L$ si et seulement si il existe une suite d'indices j_0, j_1, \dots, j_k dans $\{1, \dots, n\}$ tel que $s = j_0$, $t = j_k$, et pour tout $i = 0, \dots, k-1$ on a $M_{j_i, j_{i+1}} = 1$. Notons que, une fois

compacte exacte de telles nombres sujets à des opérations arithmétiques complexes est l'objet d'une domaine de recherche propre : le *calcul formel*.

« codées en machine », les données G , s et t forment une chaîne de bits, c'est-à-dire un mot de $\Sigma^* = \{0, 1\}^*$.

2.3.2 La classe P

Pour un mot $x \in \Sigma^*$, on note $|x|$ la taille de x , c'est-à-dire le nombre de symboles (ou lettres) de x (souvent, simplement le nombre de bits). On dit qu'un algorithme A résolvant un problème de décision est *polynomial* s'il existe $c \geq 0$ tel que pour tout $x \in \Sigma^*$, A s'exécute en temps $O(|x|^c)$. On note $A(x)$ la décision retournée par A sur la donnée x : $A(x) \in \{\text{oui}, \text{non}\}$. Lorsque $A(x) = \text{oui}$, on dit que A *accepte* x , et sinon on dit que A *rejette* x .

Exemple. Considérons le langage L formé par l'ensemble des matrices inversibles. Etant donné une matrice M , on souhaite savoir si $M \in L$, c'est-à-dire si M est inversible. Une matrice est inversible si et seulement si son déterminant est non nul. Pour décider L , un algorithme consiste donc à calculer le déterminant de M par la formule de Leibniz, et à accepter M si et seulement si ce déterminant est non nul. Cet algorithme n'est toutefois pas polynomial car calculer la formule de Leibniz pour une matrice $n \times n$ nécessite de l'ordre de $n!$ opérations arithmétiques. Un autre algorithme consiste à utiliser l'algorithme de l'élimination de Gauss-Jordan pour inverser M (qui, en particulier signale si M n'est pas inversible). Cet algorithme s'exécute en au plus $O(n^3)$ opérations élémentaires. Il est donc polynomial en la taille de M (décrite sous la forme de n^2 entiers).

Definition 2 La classe P consiste en tous les langages L pour lesquels il existe un algorithme A polynomial en la taille de x tel que pour tout $x \in \Sigma^*$: A accepte x si et seulement si $x \in L$, c'est-à-dire :

$$x \in L \iff A \text{ accepte } x$$

Informellement, la classe P est donc la classe de tous les problèmes de décision qui peuvent se résoudre en temps polynomial. Ainsi, par exemple, le langage $L = \{\text{matrices inversibles}\}$ est dans P (l'algorithme de décision est celui de l'élimination de Gauss-Jordan). Par exemple, un graphe coloré est un graphe $G = (V, E)$ muni d'une fonction $c : V \rightarrow \mathbb{N}$ affectant à chaque sommet u une couleur $c(u)$. Un graphe est proprement coloré si, pour chaque sommet, la couleur de ce sommet est différente de la couleur de ses voisins. Soit $L = \{\text{graphes proprement colorés}\}$. On a $L \in P$, comme le démontre l'algorithme ci-dessous qui suppose que G est décrit par sa matrice d'adjacence M :

```

Algorithme decide-coloration-propre( $G, c$ ) :
début
    col-propre  $\leftarrow$  vrai
    pour  $i = 1$  à  $n$  faire
        pour  $j = 1$  à  $n$  faire
            si  $M[i, j] = 1$  et  $c(i) = c(j)$  alors col-propre  $\leftarrow$  faux
        retourner col-propre
fin
```

La taille d'une instance est $\Omega(n^2)$ puisque la matrice d'adjacence M de G est de taille $n \times n$, et l'algorithme effectue $O(n^2)$ opérations élémentaires. Cet algorithme est donc polynomial (il est en fait même linéaire en la taille des données). On peut écrire un algorithme en temps $O(m)$ si G est décrit par listes d'adjacence L :

```

Algorithme decide-coloration-propre2( $G, c$ ) :
début
    col-propre  $\leftarrow$  vrai
    pour  $i = 1$  à  $n$  faire
         $j \leftarrow 1$ 
        tant que  $L[i, j] \neq \perp$  faire
            si  $c(i) = c(L[i, j])$  alors col-propre  $\leftarrow$  faux
            sinon  $j \leftarrow j + 1$ 
        retourner col-propre
    fin

```

La taille d'une instance est dans ce cas $\Omega(m \log n)$ bits puisque la liste d'adjacence L de G est de taille $2m$ (chaque arête apparaît deux fois), et l'algorithme effectue $O(m)$ opérations élémentaires. Cet algorithme est donc polynomial (il est en fait également linéaire en la taille des données).

Optimisation vs. décision. Souvent, les problèmes que l'on a à résoudre sont des problèmes d'*optimisation*, où l'on cherche une meilleure solution parmi un ensemble de solutions admissibles. Par exemple, le problème DISTANCE : étant donné un graphe $G = (V, E)$, et deux sommets $s, t \in G$, calculer la distance entre s et t dans G , c'est-à-dire le nombre minimum d'arête à traverser pour aller de s à t . Bien que ce problème ne soit pas un problème de décision, on peut en donner une version sous la forme d'un problème de décision : étant donné un graphe $G = (V, E)$, deux sommets $s, t \in G$, et un entier k , est-ce que la distance entre s et t dans G est au plus k ? Par abus de langage on dit parfois qu'un problème d'optimisation est dans P pour dire rapidement que le problème de décision associé à ce problème est dans P. Il pourra en être de même pour d'autres classes de complexité que nous verrons plus loin.

2.3.3 La classe NP

S'il est possible de montrer que de nombreux problèmes sont dans P, montrer qu'un problème *n'est pas* dans P se révèle une tâche extrêmement ardue en général. Ceci est en partie dû à l'énorme difficulté de trouver des bornes inférieures sur le temps nécessaire à la résolution d'un problème. Prenons par exemple le problème SAT défini comme suit.

Un *littéral* est un élément de $\{x, \bar{x}\}$ où x est une variable booléenne, c'est-à-dire $x \in \{\text{vrai}, \text{faux}\}$. Si $x = \text{vrai}$ alors $\bar{x} = \text{faux}$, et si $x = \text{faux}$ alors $\bar{x} = \text{vrai}$. En particulier, $\bar{\bar{x}} = x$.

Rappelons que toute formule booléenne F , c'est-à-dire toute combinaison de littéraux par des opérateurs « ou » (noté \vee) et « et » (noté \wedge), peut s'écrire sous *forme normale conjonctive* (FNC), c'est-à-dire sous la forme

$$F = \bigwedge_{i=1}^m \bigvee_{j=1}^{k_i} \ell_{i,j}$$

ou les $\ell_{i,j}$ sont des littéraux d'un ensemble de n variables booléennes x_1, \dots, x_n . Par exemple :

$$F_1 = (x_1) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$$

ou

$$F_2 = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4).$$

L'utilisation de formules booléennes sous FNC est fréquente en bases de données, où l'on a fréquemment recours à des requêtes de la forme : retourner la liste des éléments de la base

satisfaisant les propriétés P_1, \dots, P_m où chaque propriété P_i est de la forme alternative, q_1 ou q_2 ou ... ou q_k . Par exemple, retourner tous les auteurs de romans espagnols, italiens, ou français ayant vécu au XVII ou au XVIIIème siècle. Les $C_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$ sont les *clauses* de la formule F . C_i est de *longueur* k_i .

Notons que, pour toute affectation de x_1, x_2 , et x_3 à vrai ou faux, la formule F_1 est toujours fausse. En revanche, la formule F_2 est vraie pour $x_1 = \text{vrai}$, $x_3 = \text{faux}$, $x_4 = \text{vrai}$, et x_2 quelconque. On peut stocker une formule F en FNC de plusieurs façons équivalentes à une réduction polynomiale près (de la même façon que les codages d'un graphe en matrice d'adjacence et en liste d'adjacence sont équivalents). Par exemple, une formule de n variables x_1, \dots, x_n sur m clauses C_1, \dots, C_m peut être codée comme une matrice F de taille $m \times n$, où $F_{i,j} \in \{-1, 0, 1\}$ telle que :

$$F_{i,j} = \begin{cases} 0 & \text{si le littéral } x_j \text{ n'apparaît pas dans la clause } C_i \\ 1 & \text{si le littéral } x_j \text{ apparaît pas dans la clause } C_i \text{ sous forme positive } (x_i) \\ -1 & \text{si le littéral } x_j \text{ apparaît dans la clause } C_i \text{ sous forme négative } (\bar{x}_i) \end{cases}$$

Le problème SAT est alors le suivant :

Le problème SAT :

Entrée : une formule booléenne F sous FNC;

Question : existe-t-il une affectation des variables booléennes de F telle que F soit vraie ?

Pour tout entier $k \geq 0$, le problème k -SAT est le problème SAT restreint aux FNC dont les clauses sont de longueur au plus k .

Exercice 2 Montrer que le problème 2-SAT est dans P.

Alors qu'il est trivial de donner un algorithme exponentiel résolvant SAT par tests successifs des 2^n valeurs possibles des n variables booléennes de la formule, on ne connaît pas d'algorithme polynomial pour ce problème. Notez que tester 2^n valeurs est irréaliste dès que n dépasse quelques dizaines, même sur un ordinateur ultra-puissant. Ce fait est illustré dans la figure 6.

En fait, il est fortement suspecté qu'il n'existe pas d'algorithme polynomial pour SAT, mais cela reste toutefois au titre de conjecture, malgré près de cinquante années d'efforts de la communauté. Démontrer la non existence d'un algorithme polynomial pour un problème (ou un langage) arbitraire est une tâche extrêmement complexe. Il convient toutefois de remarquer un fait intriguant à propos de SAT. Trouver rapidement (c'est-à-dire en temps polynomial) les valeurs des variables booléennes susceptibles de rendre vraie la formule semble très difficile. En revanche, si la formule est satisfiable et que quelqu'un (disposant des capacités de l'*oracle* de Delphes) devine l'affectation correspondante des variables, alors vérifier que cette affectation rend effectivement vraie la formule se fait en temps polynomial (en temps au plus le nombre de clauses fois le nombre maximum de littéraux par clauses), par exemple au moyen de l'algorithme suivant :

Algorithme verification(F, x) :

début

$a \leftarrow \text{vrai}$

pour $i = 1$ à m **faire**

$b \leftarrow \text{faux}$

	1	2	3	4	10	20	30	40	100
n	1×10^{-6} seconds	2×10^{-6} seconds	3×10^{-6} seconds	4×10^{-6} seconds	1×10^{-5} seconds	2×10^{-5} seconds	3×10^{-5} seconds	4×10^{-5} seconds	1×10^{-4} seconds
2 ⁿ	1×10^{-6} seconds	4×10^{-6} seconds	9×10^{-6} seconds	1.6×10^{-5} seconds	1×10^{-4} seconds	4×10^{-4} seconds	9×10^{-4} seconds	1.6×10^{-3} seconds	1×10^{-2} seconds
3 ⁿ	1×10^{-6} seconds	8×10^{-6} seconds	2.7×10^{-5} seconds	6.4×10^{-5} seconds	1×10^{-3} seconds	8×10^{-3} seconds	2.7×10^{-3} seconds	6.4×10^{-2} seconds	1 seconds
4 ⁿ	1×10^{-6} seconds	1.6×10^{-5} seconds	8.1×10^{-6} seconds	2.6×10^{-4} seconds	1×10^{-2} seconds	0.16 seconds	0.81 seconds	2.56 seconds	100 seconds
2 ²ⁿ	2×10^{-6} seconds	4×10^{-6} seconds	8×10^{-6} seconds	1.6×10^{-5} seconds	1×10^{-3} seconds	1.05 seconds	17.9 minutes	12.7 days	4×10^{14} years
3 ²ⁿ	3×10^{-6} seconds	9×10^{-6} seconds	2.7×10^{-5} seconds	8.1×10^{-5} seconds	5.9×10^{-2} seconds	58 minutes	6.52 years	3.9×10^5 5.9×10^{37} years	
4 ²ⁿ	4×10^{-6} seconds	1.6×10^{-5} seconds	6.4×10^{-5} seconds	2.6×10^{-4} seconds	1.05 seconds	12.7 days	3.7×10^4 years	3.8×10^{10} 5.1×10^{46} years	

FIGURE 6 – Différence entre le temps de calcul polynomial et exponentiel. On suppose qu'une opération élémentaire s'effectue en 10^{-6} secondes. Un algorithme linéaire (i.e., s'exécutant en temps n) prendra 10^{-4} secondes pour traiter un problème de taille 100, alors qu'un algorithme quadratique prendra un temps 10^{-2} secondes. Un algorithme exponentiel, disons en temps 2^n , prendra... 400.000 milliard d'années, soit environ 20.000 fois l'âge estimé de l'univers.

```

pour  $j = 1$  à  $n$  faire
    si  $((F[i, j] = 1 \text{ et } x[j] = \text{vrai}) \text{ ou } (F[i, j] = -1 \text{ et } x[j] = \text{faux}))$  alors  $b \leftarrow \text{vrai}$ 
     $a \leftarrow a \wedge b$ 
retourner  $a$ 
fin

```

Cet algorithme prend en entrée une instance codée sur $O(nm)$ bits, et effectue $O(mn)$ opérations élémentaire. Il est donc polynomial.

Dit autrement, si la réponse à SAT est oui, c'est-à-dire si la formule en entrée est satisfiable, alors il existe un *certificat* qui vous permet de vous convaincre que la formule est effectivement satisfiable. De plus, l'algorithme de vérification est fiable, au sens que si quelqu'un essaie de vous tromper en prétendant qu'une formule non satisfiable est satisfiable, il ne sera pas en mesure de vous fournir un certificat accepté par l'algorithme.

La classe NP est la classe des problèmes pour lesquels, comme pour SAT, il existe un certificat de taille polynomial permettant de vérifier la solution en temps polynomial et de façon fiable (on ne peut pas tromper l'algorithme de vérification avec un faux certificat). Plus formellement :

Definition 3 La classe NP consiste en tous les langages L pour lesquels il existe un algorithme polynomial A tel que pour tout $x \in \Sigma^*$:

- si $x \in L$ alors il existe $y \in \Sigma^*$ de taille polynomiale en la taille de x tel que $A(x, y)$ accepte ;
- si $x \notin L$ alors pour tout $y \in \Sigma^*$, $A(x, y)$ rejette.

Dis autrement, il existe $c \geq 0$ tel que, pour tout $x \in \Sigma^*$:

$$x \in L \iff \exists y \in \Sigma^* : |y| = O(|x|^c) \text{ et } A(x, y) \text{ accepte.}$$

Le mot $y \in \Sigma^*$ est appelé certificat.

On impose que la taille $|y|$ du certificat soit polynomiale en la taille $|x|$ de l'entrée x , car il est

crucial que A s'exécute en temps polynomial en la taille de x . Or la complexité d'un algorithme est exprimée en la taille de ses entrées, et la taille de l'entrée de A est $|x| + |y|$.

Notons qu'on obtient immédiatement de la définition ci-dessus que

$$P \subseteq NP$$

puisque tout langage de P est reconnaissable avec comme certificat le mot vide ϵ . Mais attention ! NP ne veux **pas** dire « non-polynomial » ! NP est pour *non-deterministic polynomial*. On peut en effet définir le certificat comme le résultat d'une machine de Turing *non déterministe* qui fait à chaque étape des choix non déterministes pouvant amener à trouver une solution en temps polynomial parmi l'éventail potentiellement exponentiel des choix possibles.

A propos de la question P versus NP . Serait-il possible que $P = NP$? Cette question est de fait la question la plus importante de l'informatique contemporaine, et ses termes dépassent largement le cadre de cette discipline scientifique. Il est en effet possible de reformuler cette question différemment. Un certificat y pour l'appartenance de x à un langage L peut être vu comme une preuve du théorème « $x \in L$ ». Or, l'expérience montre empiriquement que démontrer un théorème et significativement plus difficile que vérifier une preuve de ce théorème. $P = NP$ reviendrait à nier cette intuition empirique, puisque $P = NP$ impliquerait que démontrer un théorème et vérifier une démonstration existente de ce théorème seraient sensiblement de même difficulté !

Egalement, pour les lecteurs ayant quelques notions de logique, on peut définir la classe P comme les langages descriptibles par un certain fragment F de la logique du premier ordre, et la classe NP comme le fragment existentiel de la logique monadique du second ordre (EMSO). L'égalité $P = NP$ impliquerait donc que le fragment EMSO, pourtant vu comme un vaste fragment du second ordre, n'inclut pas plus de langages que le fragment F de la logique du premier ordre.

2.3.4 Réduction polynomiale et problème NP -difficiles

On a vu que $SAT \in NP$. Ce problème résiste depuis des décennies à sa résolution en temps polynomial. En présence d'un problème Π pour lequel on échoue à trouver un algorithme polynomial, une façon de montrer que cet échec est causé par la difficulté intrinsèque du problème Π et non par un manque d'aptitude à la conception d'algorithmes efficaces, consiste à montrer que Π est au moins aussi difficile que SAT . Une façon de procéder consiste à montrer que s'il existait un algorithme polynomial pour résoudre Π alors on en déduirait un algorithme polynomial pour résoudre SAT . Ceci se traduit formellement au moyen de *réductions* polynomiales.

Definition 4 Une réduction polynomiale d'un langage $L_1 \subseteq \Sigma^*$ vers un langage $L_2 \subseteq \Sigma^*$ est une fonction $f : \Sigma^* \rightarrow \Sigma^*$ telle que :

1. f est calculable en temps polynomial (c'est-à-dire, il existe un algorithme A qui, pour tout $x \in \Sigma^*$, calcule $f(x)$ en temps polynomial en $|x|$) ;
2. $\forall x \in \Sigma^*, x \in L_1 \iff f(x) \in L_2$.

Ainsi, s'il existe une réduction polynomiale f de L_1 vers L_2 , l'existence d'un algorithme polynomial A_2 décidant L_2 implique l'existence d'un algorithme polynomial A_1 décidant L_1 . Cet algorithme est le suivant : pour tout $x_1 \in \Sigma^*$ donné, calculer $x_2 = f(x_1)$ en appliquant A

(l'algorithme calculant f), puis appliquer A_2 à x_2 . (Notez que $|x_2|$ est polynomial en $|x_1|$ car f s'exécute en temps polynomial en $|x_1|$, et donc ne peut que construire un mot x_2 de taille au plus polynomiale en la taille de x_1).

Le concept de réduction polynomial permet en fait de définir une sous-classe cruciale de NP :

Definition 5 *Un langage (i.e., problème) L est dit NP-difficile si, pour tout $L' \in \text{NP}$, il existe une réduction polynomiale de L' vers L . Si de plus $L \in \text{NP}$ alors L est dit NP-complet. On note NPC la classe des langages NP-complets.*

Un résultat surprenant est qu'il existe des problèmes NP-difficiles. Ce résultat est surprenant puisqu'un problème NP-difficile Π a la caractéristique que si on sait décider de tout instance x de Π en temps $t(|x|)$, alors on sait décider de tout instance x' de tout problème $\Pi' \in \text{NP}$ en temps $t(|f(x')|) + p(|x'|)$ où p est le temps (polynomial) nécessaire au calcul de la réduction polynomiale f de L' vers L . Ainsi, si l'on connaissait un algorithme polynomial pour résoudre un problème NP-difficile, on obtiendrait $\text{P} = \text{NP}$. Un tel algorithme n'est pas connu, et il est conjecturé qu'un tel algorithme n'existe pas. Le théorème suivant est donc un des résultats les plus importants de la théorie de la complexité :

Théorème 2 (Levin-Cook, 1971) SAT est NP-difficile.

Comme $\text{SAT} \in \text{NP}$, on obtient que SAT est NP-complet : $\text{SAT} \in \text{NPC}$. En 1972, Richard Karp a publié 21 problèmes NP-complets, et la liste ne cesse de s'allonger depuis (voir l'ouvrage de Garey et Johnson pour une très longue liste de problème NP-complets). En particulier, k -SAT est NP-complet pour tout $k \geq 3$.

Voici un autre exemple de problème NP-complet :

Le problème COUVERTURE ENSEMBLE :

Entrée : un ensemble fini U et une collection $\mathcal{S} = \{S_1, \dots, S_r\}$ de sous-ensembles de U ;

Objectif : trouver le plus petit nombre d'éléments de \mathcal{S} dont l'union contient tous les éléments de U .

En fait, il s'agit bien sûr du problème de décision correspondant qui est NP-complet, c'est-à-dire, étant donné U , \mathcal{S} , et un entier k , est-il possible de « couvrir » U avec au plus k sous-ensembles dans \mathcal{S} ? Plus précisément :

Entrée : un ensemble fini U , une collection $\mathcal{S} = \{S_1, \dots, S_r\}$ de sous-ensembles de U , et $k \geq 0$;

Question : existe-t-il $I \subseteq \{1, \dots, r\}$, avec $|I| \leq k$ tel que $\cup_{i \in I} S_i = U$?

Nous savons maintenant ce qu'il convient de faire lorsque l'on échoue à trouver un algorithme polynomial pour un problème donné Π : il faut essayer de montrer que Π est NP-difficile. Pour cela, la démarche à suivre est la suivante :

1. chercher dans la liste des problèmes NP-complets connus à ce jour celui, disons Π' , qui semble le plus « ressembler » à Π (par exemple, $\Pi' = \text{SAT}$ ou $\Pi' = \text{COUVERTURE ENSEMBLE}$) ;

2. trouver une réduction polynomiale de Π' vers Π .

La seconde étape de la démarche ci-dessus s'effectue comme suit. Plutôt que de parler de langages, revenons à une description des problèmes de décision en terme d'instances. Soit \mathcal{I} l'ensemble des instances de Π , et \mathcal{I}' l'ensemble des instances de Π' . Il s'agit de trouver une construction systématique, c'est-à-dire un algorithme A , qui, en temps polynomial en la taille de chaque instance $I' \in \mathcal{I}'$, construit une instance $I = A(I') \in \mathcal{I}$, tel que $\Pi'(I')$ vrai si et seulement si $\Pi(I)$ vrai.

Remarque. L'expérience montre qu'on arrive presque toujours à montrer qu'un problème Π de NP satisfait $\Pi \in P$ ou $\Pi \in NPC$. Il existe toutefois quelques problèmes de NP dont on ne sait s'ils sont dans P ou NPC. Il est conjecturé que certains de ces problèmes sont strictement entre P et NPC. Un exemple de ce type de problèmes est le test d'isomorphisme de graphes. On dit que deux graphes $G = (V, E)$ et $G' = (V', E')$ sont isomorphes s'il existe une bijection $\phi : V \rightarrow V'$ qui préserve l'adjacence, c'est-à-dire telle que, pour tout $u, v \in V$, $\{u, v\} \in E \iff \{\phi(u), \phi(v)\} \in E'$. Le test d'isomorphisme est le problème suivant :

Le problème ISOMORPHISME DE GRAPHES (ISOGRAPHES) :

Entrée : deux graphes G et G' ;

Question : G et G' sont-ils isomorphes ?

On a ISOGRAPHES $\in NP$. En revanche, il est conjecturé que ISOGRAPHES $\notin P \cup NPC$. Si cette conjecture est vraie, alors $P \neq NP$.

2.4 Exemples de preuves de NP-complétude

2.4.1 Réduction à partir de 3-SAT

Soit $G = (V, E)$ un graphe orienté, et soient $s \in V$. On cherche à concevoir un protocole de communication permettant d'envoyer un message à partir de la source s vers tous les sommets de G , en suivant les arcs de G . Les communications procèdent par étape. A chaque étape, chaque sommet u informé du message peut le transmettre à au plus un voisin sortant de u . On note $b(G, s)$ le temps de diffusion dans G à partir de s (b est pour « broadcast »), c'est-à-dire le nombre minimal d'étapes de communication nécessaires pour acheminer le message de s à tous les sommets de G . Notons que $b(G, s) \geq \lceil \log_2 n \rceil$ car le nombre total de sommets informés ne peut que doubler à chaque étape, du fait que chaque sommet informé ne peut informer qu'au plus un sommet à chaque étape. Calculer $b(G, s)$ est a priori difficile. En effet, nous allons montrer que le problème (de décision) suivant est NP-complet. (Il est NP-complet aussi bien pour les graphes non-orientés que pour les graphes orientés ; Dans la suite, nous considérons le cas des graphes orientés.)

Le problème DIFFUSION :

Entrée : un graphe orienté $G = (V, E)$, $s \in V$, et un entier $k \geq 0$;

Question : $b(G, s) \leq k$?

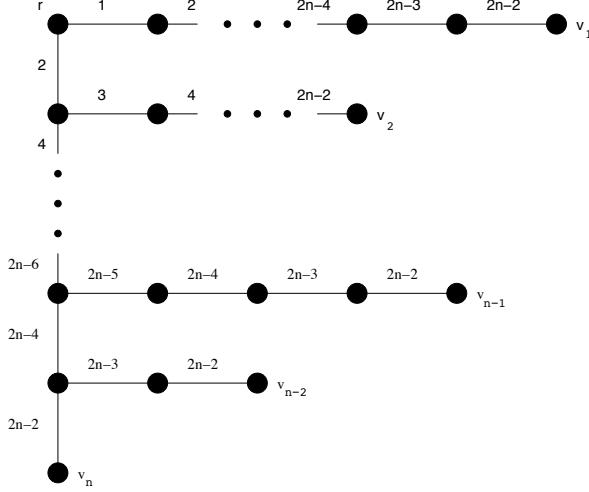


FIGURE 7 – Le gadget A_n et le protocole de diffusion canonique dans A_n . Les arcs sont orientés de la racine r vers les feuilles v_i , $i = 1, \dots, n$.

Pour montrer que DIFFUSION est NP-complet, il faut montrer d'une part que DIFFUSION \in NP, et d'autre part que DIFFUSION est NP-difficile. DIFFUSION \in NP car on peut décrire un certificat vérifiable en temps polynomial. En effet, si $b(G, s) \leq k$, alors le certificat consiste en le graphe G dont chaque arcs $e = (u, v)$ est étiqueté par un entier $t_e \in \{0, 1, \dots, k\}$. Les étiquettes s'interprètent de la façon suivante : $t_e = 0$ indique que e n'est pas utilisé pour communiquer, et, pour $1 \leq i \leq k$, $t_e = i$ indique que e est utilisé à l'étape i du protocole. Un tel certificat se code en $O(m \log k)$ bits dans un graphe de m arcs. Il est donc de taille polynomiale en la taille $\Theta(n^2 + \log k)$ de l'entrée puisque $m \leq \frac{n(n-1)}{2}$ dans un graphe de n sommets.

Afin de vérifier que le protocole de vérification est correct, il suffit de vérifier les propriétés suivantes :

- Pour tout arc e , $t_e \in \{0, 1, \dots, k\}$;
- Chaque sommet $u \neq s$ doit posséder un arc entrant e tel que $t_e > 0$;
- Il n'y a qu'un arc étiqueté 1, et cet arc sort de s ;
- Si un arc e étiqueté $i > 1$ sort d'un sommet u , alors un arc étiqueté $j < i$ doit entrer en u ;
- Les arcs étiquetés positivement sortant d'un même sommet doivent posséder des étiquettes deux à deux distinctes.

Ces propriétés peuvent être vérifiées en temps $O(m)$, c'est-à-dire en temps $O(n^2)$ dans un graphe de n sommets. Si $b(G, s) \leq k$, alors il existe un étiquetage satisfaisant les cinq propriétés ci-dessus. En revanche, si $b(G, s) > k$, alors aucun étiquetage ne peut satisfaire ces cinq propriétés.

Pour montrer que DIFFUSION est NP-difficile, nous allons construire une réduction polynomiale de 3-SAT à DIFFUSION. Pour définir cette réduction, on définit tout d'abord le *gadget* A_n consistant en un arbre enraciné en r , et dont les feuilles sont v_1, \dots, v_n (voir la figure 7). L'orientation des arcs n'est pas indiquée, mais ceux-ci sont orientés de la racine vers les feuilles. Cet arbre possède $\sum_{i=0}^{n-1} (2i + 1) = n^2$ sommets. Etant donnée une formule

$$F = \bigwedge_{i=1}^m C_i = \bigwedge_{i=1}^m \bigvee_{j=1}^3 \ell_{i,j}$$

de m clauses C_1, \dots, C_m d'au plus trois littéraux chacune, et de n variables v_1, \dots, v_n , la réduction

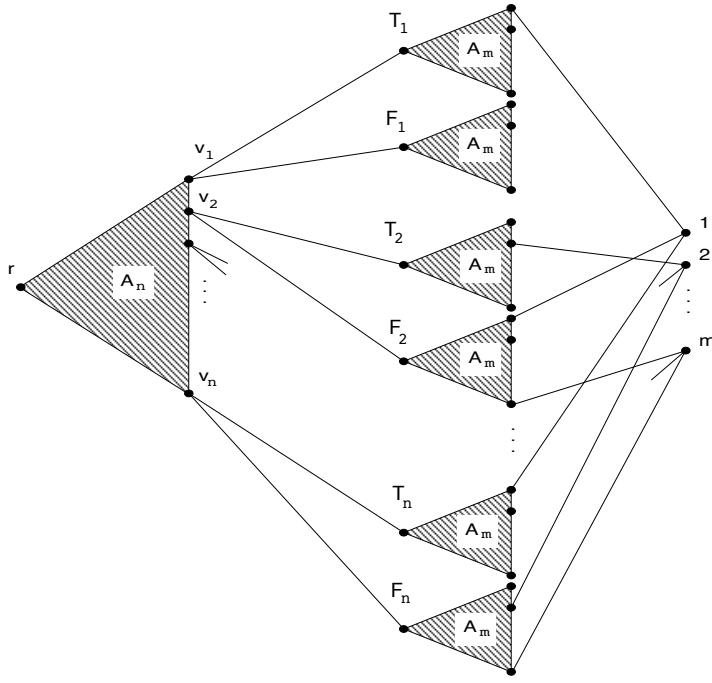


FIGURE 8 – Graphe orienté (de la gauche vers la droite) construit à partir d'une formule 3-SAT

consiste à construire le graphe orienté G de la figure 8.

Le graphe G consiste en un gadget A_n enraciné en r , et dont chacune des n feuilles correspondent à des variables v_1, \dots, v_n , plus $2n$ gadgets A_m . Les feuilles de chaque gadget A_m sont en correspondance bijective avec les m clauses. Chaque v_i est connecté aux racines de deux arbres A_m notés T_i et F_i , pour « true » et « false ». En plus de A_n et des $2n$ copies de A_m , G inclut m sommets représentant les clauses C_1, \dots, C_m . La règle de connexion entre tel un sommet i et les feuilles des A_m dépend de la clause C_i . Soit $C_i = \bigvee_{j=1}^3 \ell_{i,j}$. Pour $j \in \{1, 2, 3\}$, si $\ell_{i,j} = v_k$ pour $k \in \{1, \dots, n\}$, alors i est connecté par un arc de la i ème feuille de l'arbre T_k , sinon (c.-à-d. $\ell_{i,j} = \bar{v}_k$ pour $k \in \{1, \dots, n\}$) i est connecté par un arc de la i ème feuille de l'arbre F_k . Remarquons que chaque sommet i a donc au plus 3 arcs entrants, puisque chaque clause ne contient qu'au plus 3 littéraux. Par ailleurs, chaque feuille des A_m n'a qu'au plus un arc sortant.

Enfin, on fixe $k = 2n + 2m - 2$.

L'image de l'instance F de 3-SAT par la transformation est l'instance (G, r, k) de DIFFUSION.

Le nombre de sommets de G est $N = 2nm^2 + n^2 + m$. Le nombre m de clauses distinctes impliquant 3 littéraux sur n variables est au plus $8 \binom{n}{3} \leq O(n^3)$. Notons par ailleurs que l'on doit avoir $m \geq \frac{n}{3}$ car sinon une variable n'apparaît dans aucun littéral. La formule F est encodée sous la forme d'une table d'entiers positifs ou négatifs de m lignes (les m clauses) et de 3 colonnes (les 3 variables), tel que, par exemple, une ligne $(i, -j, k)$ code la clause $v_i \vee \bar{v}_j \vee v_k$. F est donc encodée sur $\Theta(m \log n)$ bits, soit au moins $\Omega(n)$ bits. On a donc bien N de taille polynomiale en la taille de l'instance. Construire le graphe G consiste à construire sa matrice d'adjacente, de taille N^2 . Chaque entrée de cette matrice est calculée selon les règles simples énoncées ci-dessus, en temps $O(N^2)$. La réduction est donc bien polynomiale. La première propriété de la définition 4 est donc satisfaite.

Afin de s'assurer que la seconde propriété de la définition 4 est satisfaite, il faut montrer l'équivalence suivante :

$$F \text{ est satisfiable} \iff b(G, r) \leq 2m + 2n - 2.$$

Supposons donc que F est satisfiable. La figure 7 indique également un protocole de diffusion dans A_n s'exécutant en $2n - 2$ étapes. De même, on peut diffuser dans un arbre A_m en $2m - 2$ étapes. Considérons une affectation des variables à vrai ou faux de façon à satisfaire F . Si $v_i = \text{vrai}$ alors le sommet v_i de A_n envoie d'abord à la racine de T_i , puis à la racine de F_i . Ainsi, toutes les feuilles de T_i recevront le message à l'étape $(2n - 2) + 1 + (2m - 2) = 2n + 2m - 3$, et toutes les feuilles de F_i recevront le message à l'étape $(2n - 2) + 2 + (2m - 2) = 2n + 2m - 2$. Si la variable v_i rend une clause C_j vraie, alors le sommet j reçoit le message du j ème sommet de T_i . Le raisonnement est identique si $v_i = \text{faux}$ en informant d'abord F_i puis T_i . Donc $b(G, r) \leq 2m + 2n - 2$.

Supposons réciproquement que $b(G, r) \leq 2m + 2n - 2$. Si le protocole canonique de la figure 7 n'est pas employé dans A_n alors un sommet v_i reçoit le message à une étape $\geq 2n - 1$. En conséquence, la racine de T_i ou de F_i reçoit le message au plus tôt au temps $2n + 1$. Diffuser dans cet arbre prend encore $2m - 2$ étapes, entraînant un temps de diffusion $> 2n + 2m - 2$. Donc le protocole canonique est utilisé dans A_n . En conséquence, pour chaque i , un des deux arbres T_i ou F_i a toutes ses feuilles informées à l'étape $2n + 2m - 3$. Le protocole canonique doit nécessairement être utilisé dans l'autre arbre. Ainsi, l'arbre premier informé (F_i ou T_i) dicte une unique affectation des variables à vrai ou faux, rendant F satisfiable.

2.4.2 Réduction à partir de COUVERTURE ENSEMBLE

Soit $G = (V, E)$ un graphe orienté, et soient $s \in V$ et $S \subseteq V$ avec $s \in S$. Les sommets de $D = V \setminus S$ sont supposés potentiellement hostiles, et on les appelle sommets *dangereux*. On cherche à concevoir un protocole de communication permettant d'informer tous les sommets de S à partir de s , en suivant les arcs de G mais en traversant le moins possible de sommets dangereux. Le modèle de communication est différent que dans l'exemple précédent. Ici, on cherche simplement à construire $|S| - 1$ chemins dans G , connectant s à chacun des sommets de $S \setminus \{s\}$, tels que l'ensemble des sommets dangereux traversé par ces chemins soit le plus petit possible.

Exercice 3 Montrer, par réduction à partir de COUVERTURE ENSEMBLE, que le problème suivant est NP-complet :

Le problème DIFFUSION SÛRE :

Entrée : un graphe orienté $G = (V, E)$, $S \subseteq V$, $s \in S$, et un entier $k \geq 0$;

Question : existe-t-il $|S| - 1$ chemins dans G , connectant s à chacun des sommets de $S \setminus \{s\}$, tels que l'ensemble des sommets dangereux traversés par ces chemins soit de taille au plus k ?

2.5 Complexité spatiale

2.5.1 Espace polynomial

Informellement, la classe PSPACE est la classe des problèmes pouvant être résolu en utilisant une mémoire de taille polynomiale en la taille des données, c'est-à-dire de taille au plus $|x|^c$ pour une données x , où $c \geq 0$ est une constante quelconque.

On a $P \subseteq PSPACE$ car un algorithme exécutant un nombre polynomial d'étapes ne peut utiliser qu'un nombre polynomial de cellules mémoire.

2.5.2 Espace logarithmique

Informellement, la classe L est la classe des problèmes pouvant être résolus (par une machine de Turing) en utilisant une mémoire (c'est-à-dire un ruban de travail) de taille logarithmique en la taille des données, c'est-à-dire de taille au plus $c \log |x|$, où $c \geq 0$ est une constante quelconque. Alternativement, on peut voir la classe L comme la classe des problèmes pouvant être résolus par une machine RAM sans mémoire et n'utilisant qu'un nombre *constant* de registres.

Un algorithme utilisant une nombre constant c de registres de taille au plus $\log |x|$ bits où $|x|$ est la taille de l'entrée x ne peut effectuer qu'au plus $2^{c \log |x|}$ étapes de calcul. En effet, l'ensemble des registres ne peut prendre que $2^{c \log |x|}$ valeurs possibles, et donc la machine n'a que $2^{c \log |x|}$ états possibles. Or si la machine se trouve deux fois dans le même état, alors elle se trouvera une infinité de fois dans ce même état, et donc ne terminera pas. La machine ne doit donc effectuer qu'au plus $2^{c \log |x|} = n^c$ étapes de calcul. L'algorithme s'exécute donc en temps polynomial. On a donc $L \subseteq P$. Notez que la question $L = P$? est un autre question ouverte majeure de l'informatique. Si $L = P$, cela impliquerait qu'il serait possible, pour tout problème Π de P , de mettre en œuvre sur une RAM un algorithme résolvant Π en utilisant un nombre constant de registres, sans écriture en mémoire.

2.5.3 Complexité spatiale non déterministe

Les classes $NPSPACE$ et NL sont à $PSPACE$ et L ce que NP est à P . Par définition, $PSPACE \subseteq NPSPACE$ et $L \subseteq NL$. Nous avons vu par ailleurs que $2\text{-SAT} \in P$. Or, il est connu que 2-SAT est NL -complet (pour la réduction log-space). On obtient ainsi $L \subseteq NL \subseteq P$. On ne sait pas si ces inclusions sont strictes ou pas.

Un autre problème NL -complet est le suivant :

Le problème (s, t) -connectivité (STCON) :

Entrée : un graphe orienté $G = (V, E)$ et deux sommets s et t de

G ;

Question : existe-t-il un chemin de s à t dans G ?

On a $STCON \in P$, en effectuant un parcours BFS ou DFS à partir de s (voir les sections 3.1.1 et 3.1.2). Savitch (1970) a montré que STCON pouvait être résolu en espace $O(\log^2 n)$ où n est le nombre de sommets du graphe, par l'algorithme ci-dessous qui décide s'il existe un chemin dans G entre u et v de longueur au plus d :

Algorithme **SQUARE**(G, u, v, d)

début

si ($d = 0$ et $u = v$) **alors** retourner OUI

sinon si ($d = 1$ et $(u = v$ ou $(u, v) \in E)$) **alors** retourner OUI

sinon si $\exists w \in V$ tel que $SQUARE(G, u, w, \lceil d/2 \rceil)$ et $SQUARE(G, w, v, \lfloor d/2 \rfloor)$

alors retourner OUI

sinon retourner NON

fin

En appelant $\text{SQUARE}(G, s, t, n)$, on décide ainsi s'il existe un chemin entre s et t . Cet algorithme, inefficace en temps, utilise une mémoire $O(\log^2 n)$. L'intuition de ce fait est la suivante : la profondeur des appels récursifs est $\lceil \log_2 n \rceil$ car la distance testée d est divisée par 2 à chaque appel. La liste des sommets en cours d'appels récursifs ne contient donc qu'au plus $\lceil \log_2 n \rceil$ sommets. Or, chaque sommet peut être identifié par un entier entre 1 et n , donc sur $\lceil \log_2 n \rceil$ bits. L'algorithme ne stocke donc jamais plus que $O(\log^2 n)$ bits à chaque étapes. Comme STCON est NL -complet, on obtient ainsi $\text{NL} \subseteq \text{L}^2$.

En fait, le théorème de Savitch (1970), et l'algorithme ci-dessus, permet de montrer que $\text{NPSPACE} = \text{PSPACE}$. Le non déterministe n'aide donc pas à gagner de l'espace mémoire au delà de l'espace polynomial. Comme $\text{NP} \subseteq \text{NPSPACE}$, pour les mêmes raisons que $\text{P} \subseteq \text{PSPACE}$, on obtient la suite d'inclusion suivante :

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}.$$

En haut de cette hiérarchie, les problèmes PSPACE -complets sont les problèmes de PSPACE plus difficiles que tous les autres problèmes de PSPACE (par réduction polynomiale). Un exemple de problème PSPACE -complet est le problème de la satisfiabilité de formules booléennes quantifiées (QSAT, pour SAT quantifié). Etant donné une formule booléenne F (en forme normale conjonctive) avec n variable x_1, \dots, x_n , SAT demande de décider si

$$\exists x_1 \dots \exists x_n \mid F.$$

QSAT demande de décider F mais avec des quantificateurs existentiels (\exists) et universels (\forall). QSAT demande donc de décider des expressions logiques de la forme

$$\exists x_1 \forall x_2 \exists x_3 \exists x_4 \forall x_5 \dots, \exists x_n \mid F.$$

Pour finir ce bref tour d'horizon de la théorie de la complexité, soulignons au moins un résultat de discrimination entre classe :

$$\text{NL} \subset \text{PSPACE} \text{ (inclusion stricte).}$$

2.5.4 Les preuves interactives

La théorie de la complexité moderne est souvent basée sur l'étude de *preuves interactives*, impliquant des interactions entre un *prouveur* et un *vérifieur*⁴. Dans le cas de NP , l'interaction est minimalist : le prouveur devine le certificat (ou la preuve) et la donne au vérifieur qui est en charge de la vérifier. La classe de complexité $\text{PCP}[r(n), q(n)]$ réfère à l'ensemble des problèmes de décision qui ont des preuves vérifiables en temps polynomial par un algorithme probabiliste utilisant au plus $r(n)$ bits aléatoires, et en lisant au plus $q(n)$ bits de la preuve. Intuitivement, on peut voir PCP en termes de preuves interactives, où le vérifieur demande au prouveur de lui fournir $q(n)$ bits de la preuve à des positions choisies aléatoirement en utilisant $r(n)$ bits aléatoires. Bien sûr, la probabilité que le vérifieur se trompe est non nulle. Elle peut toutefois être rendue arbitrairement petite.

Un résultat récent (2001) surprenant est le théorème PCP. Ce théorème est considéré comme une des avancées majeures en théorie de la complexité depuis le théorème de Cook sur la NP -complétude de SAT. Il stipule que

$$\text{PCP}[O(\log n), O(1)] = \text{NP}.$$

4. Le prouveur est souvent appelé Merlin, et le vérifieur Arthur. Une classe de complexité est d'ailleurs notée AM pour Arthur-Merlin.

Ainsi, il est possible de vérifier une preuve avec une bonne garantie de ne pas se tromper, en lisant qu'un nombre constant de bits de la preuve, choisis à des positions aléatoires dans la preuve.

3 Techniques de conception d'algorithmes

Ce chapitre traite des principales méthodes de conception d'algorithmes. Il ne rentrera pas dans les détails des structures de données, c'est-à-dire dans les détails de la façon dont les données d'un problème sont codées et stockées, même si ces structures peuvent avoir un impact significatif sur la complexité en temps et en espace d'un problème. Ainsi, coder un graphe G par sa matrice d'adjacence booléenne A permet de savoir en temps constant (sur une RAM) si deux sommets sont voisins, mais utilise une mémoire $\Omega(n^2)$ pour un graphe de n sommets. En revanche, coder le même graphe G par listes d'adjacence utilise une mémoire $O(m)$ pour un graphe de m arêtes, ce qui peut être significativement plus petit que $O(n^2)$ pour une graphe peu dense. Cependant vérifier si deux sommets sont voisins réclame dans ce cas de chercher dans une liste, ce qui peut être coûteux (ordre du longueur de la liste), surtout si la liste n'est pas triée.

L'exemple ci-dessus illustre l'importance du codage des données. Même si nous ne rentrerons pas dans les détails de ce codage, il convient de se souvenir que codage et manipulation de données vont de paire, parfois de conserve, parfois en conflit. Il convient également de garder en mémoire que tout entier est par défaut codé en binaire. L'entier positif n est donc codé sur $O(\log n)$ bits. Il existe toutefois des contextes pour lesquels on utilise un codage en unaire, où l'entier n est codé en une suite $11\dots11$ de n 1. Néanmoins, le codage unaire ne sera évoqué que pour caractériser les performances d'algorithmes, et n'est pas utilisé pour la conception d'algorithmes.

3.1 Analyse et preuves d'algorithmes

Un algorithme est formellement une séquence non ambiguë d'instructions dans le modèle RAM. Toutefois, appliquer à la lettre cette définition serait équivalent à programmer en assembleur, ou par analogie, à revenir systématiquement à la construction de la mesure de Lebesgue dès que l'on voudrait parler d'intégrales. En pratique, on décrit un algorithme à plus haut niveau, en se basant sur des concepts plus élaborés. En fait, au fur et à mesure de l'avancement du cours, on ne rentrera que de moins en moins dans les détails de l'écriture des algorithmes. Pour illustrer ce propos, imaginez la recherche d'un plus petit élément dans un tableau T de taille n . On n'écrira certainement pas le détail de la recherche de cet élément sous forme fastidieuse d'une boucle

```
min  $\leftarrow +\infty$ 
pour  $i = 1$  à  $n$  faire
    si  $T[i] \leq min$  alors  $min \leftarrow T[i]$ 
```

mais l'on dira simplement « trouver le plus petit élément de T ». (Sans compter que la boucle ci-dessus demande d'initialiser min , et de traiter quantité de détails du type tableau vide, etc.). Trois algorithmes sont toutefois décrits ci-dessous relativement finement afin d'illustrer la description d'algorithmes et l'analyse de son temps d'exécution. Dans la suite du cours, nous ne rentrerons pas dans de tels détails. Deux de ces trois algorithmes effectuent deux types essentiels de parcours de graphes (le troisième est le fameux algorithme de Dijkstra). Vous êtes invités à suivre le comportement de ces deux algorithmes sur un graphe “en huit”, c'est-à-dire deux cycles de quatre sommets, ayant un sommet en commun.

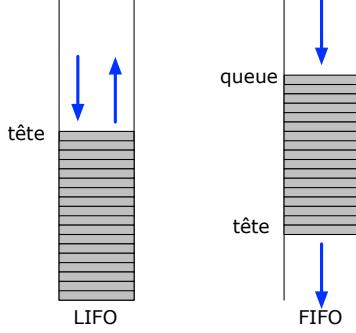


FIGURE 9 – Listes LIFO et FIFO

3.1.1 Parcours DFS

Le parcours en *profondeur d'abord* (ou DFS pour *depth-first search*) explore un graphe $G = (V, E)$ à partir d'un sommet $s \in V$ en privilégiant la visite de nouvelles arêtes. On peut utiliser un parcours DFS pour, par exemple, chercher l'ensemble des sommets x tels qu'il existe un chemin partant d'un sommet de départ s et aboutissant à x . Afin de décrire un algorithme DFS, nous allons utiliser une procédure **visite**, qui rajoute les arêtes incidentes au sommet u actuellement visité dans une liste d'arêtes L (voir la figure 9) :

```
Procédure visite( $w$ )
début
    marquer  $w$ 
    pour toute arête  $\{w, w'\}$  faire ajouter  $(w, w')$  à la liste  $L$ 
fin
```

Le parcours DFS d'un graphe G construit un arbre couvrant T de G enraciné au sommet s duquel débute le parcours, comme suit :

```
Algorithme DFS( $G, s$ )
début
     $L \leftarrow \emptyset$           /*  $L$  is a LIFO list */
     $T \leftarrow \emptyset$ 
    visite( $s$ )
    tant que  $L \neq \emptyset$  faire
        supprimer  $(u, v)$  de  $L$  /* la paire la plus récente est retirée de  $L$  */
        si  $v$  n'est pas marqué alors
            ajouter  $\{u, v\}$  à  $T$ 
            visite( $v$ )
    fin
```

DFS ajoute et supprime les paires (u, v) en tête de la liste L . Cette liste est donc utilisée sous forme LIFO (pour *last-in first-out*), ou « dernier entré, premier sorti », également appelé *pile*. Voir la figure 9.

Montrons tout d'abord que DFS construit bien un arbre couvrant T du graphe G sur lequel est exécuté DFS, c'est-à-dire que l'ensemble des arêtes $\{u, v\}$ ajoutées successivement à T forme un arbre couvrant de G , c'est-à-dire un arbre tel que $V(T) = V(G)$ et $E(T) \subseteq E(G)$.

DFS construit un arbre couvrant de G . Pour établir ce fait, remarquons tous d'abord qu'un sommet v n'est jamais marqué plus d'une fois, puisque l'instruction $\text{visiter}(v)$ n'est effectuée que si v n'est pas marqué.

Montrons que tout sommet v est marqué au moins une fois. C'est le cas de s qui est visité avant la première itération de la boucle « tant que ». Supposons qu'il existe un sommet qui n'est pas marqué par DFS. Puisque G est connexe et qu'il existe au moins un sommet marqué une fois, il existe donc une arête $\{x, y\}$ de G telle que x est marqué et y n'est pas marqué. Lorsque x est marqué, la paire (x, y) est rajoutée à L . Puisque qu'aucun sommet n'est marqué plus d'une fois, chaque paire (u, v) n'est ajoutée à L qu'au plus une fois (lorsque u est marqué). Puisque DFS supprime une paire (u, v) à chaque itération de la boucle « tant que », il suit que la paire (x, y) sera supprimée de L lors d'une itération de cette boucle. Or lorsque (x, y) est supprimée de L , comme y n'est pas marqué, DFS visite y , ce qui implique que y sera marqué, contradiction.

Donc DFS marque chaque sommet de G une et une seule fois.

En conséquence, le test « v n'est pas marqué » est donc exécuté exactement $n - 1$ fois (une fois par sommet $v \neq s$). Il en découle que T consiste en une collection d'au plus $n - 1$ arêtes. Or, une arête $\{x, y\}$ ne peut être ajoutée à T plus d'une fois. En effet, $\{x, y\}$ ne peut être ajoutée que lorsque (x, y) ou (y, x) est récupéré de la tête de L . Supposons que (x, y) soit récupéré en premier. Le sommet ayant placé (x, y) dans L ne peut être que le sommet x lorsqu'il a été visité, et donc marqué. Ainsi, lorsque (y, x) sera ensuite récupéré, le test « v n'est pas marqué » ne sera pas satisfait, et $\{x, y\}$ ne sera pas rajouté à T une seconde fois.

Donc T consiste en une collection d'exactement $n - 1$ arête distincte. Enfin, aucun sommet n'est isolé dans T puisqu'une arête incidente à tout sommet v est ajouté à T lorsque v est visité. Donc, T est un graphe sans sommet isolé, de n sommets et $n - 1$ arêtes. C'est donc un arbre.

Analyse de la complexité de l'algorithme DFS. Dans un parcours DFS, chaque arête n'est ajoutée à la liste L au plus deux fois, une fois lors de la visite de chacune de ses extrémités. Dans un graphe de m arêtes, le parcours DFS ci-dessus ne fait qu'au plus $2m$ ajouts et $2m$ suppressions d'arêtes dans L . Par ailleurs, chaque sommet est marqué exactement une fois. L'algorithme ci-dessus s'exécute donc en un temps au plus $O(n + m)$. La constante cachée dans le O prend compte de toute les mise à jour du marquage des sommets et d'ajout des arêtes à T . Pour que chaque opération du parcours DFS prenne un temps constant, il convient d'utiliser un codage du graphe en liste d'adjacence afin de trouver tous les voisins d'un sommet en temps linéairement proportionnel à ce nombre de voisins.

3.1.2 Parcours BFS

Le parcours en *largeur d'abord* (BFS pour *breadth-first search*) explore un graphe $G = (V, E)$ à partir d'un sommet $s \in V$ en privilégiant la visite “par couche”, à distance croissante. Il peut donc être utilisé pour calculer la distance entre le sommet de départ s et chaque autre sommet, dans un graphe non pondéré (la distance est simplement le nombre d'arêtes traversées — voir la section suivante pour le cas des graphes pondérés). Notez la seule différence entre l'algorithme ci-dessous pour le BFS et l'algorithme vu pour le DFS : dans le DFS, on prend les arêtes en tête de liste alors que, dans le BFS, on prend les arêtes en queue de liste. BFS utilise donc une liste sous forme FIFO (pour *first-in first-out*), ou « premier entré, premier sorti », également appelé *file*. Voir la figure 9.

```

Algorithme BFS( $G, s$ )
début
   $L \leftarrow \emptyset$       /*  $L$  is a LIFO list */
   $T \leftarrow \emptyset$ 
  visite( $s$ )
  tant que  $L \neq \emptyset$  faire
    supprimer l'arête  $(u, v)$  de  $L$  /* la plus vieille paire est retirée de  $L$  */
    si  $v$  n'est pas marqué alors
      ajouter  $\{u, v\}$  à  $T$ 
      visiter  $v$ 
  fin

```

Pour les même raisons que pour le DFS, le parcours BFS ci-dessus s'effectue en temps $O(n + m)$ dans un graphe de m arêtes.

3.1.3 Algorithme de Dijkstra

Un graphe *pondéré* est un graphe pour lequel la traversée d'une arête e a un coût $w(e) \geq 0$ (une distance, un péage, etc.). L'algorithme de Dijkstra permet de calculer la distance entre un sommet et tous les autres sommets d'un graphe pondéré. Il est typiquement utilisé dans Internet sous une version distribuée, pour le routage OSPF (*Open Shortest Path First*) à l'intérieur d'un système autonome.

Soit donc $G = (V, E)$ un graphe pondéré connexe, et $s \in V$ un sommet quelconque. On note $\ell(e)$ le poids de l'arête e . (La notation ℓ remplace ici la notation usuelle w pour bien signifier que le poids représente une longueur). On suppose $\ell(e) > 0$ pour tout $e \in E$. Rappelons qu'un chemin $P = (e_1, e_2, \dots, e_k)$ est une suite ordonnée d'arêtes de G telles que e_{i+1} est incidente à e_i , $i = 1, \dots, k-1$. La longueur de P est $\sum_{i=1}^k \ell(e_i)$. La distance $d_G(u, v)$ de u à v est la longueur d'un plus court chemin reliant u à v . Notez que ℓ n'est pas forcément une distance (l'inégalité triangulaire peut ne pas être satisfaite). En revanche, d_G est une distance. L'algorithme de Dijkstra fixe la distance de s à chacun des autres sommets u de manière séquentielle. Les sommets sont initialement placés dans un ensemble S . A chaque itération, l'algorithme choisit un sommet $u \in S$, le supprime de S , et met à jour la distance estimée $D[v]$ de s à v pour tous les voisins v de u . Lorsque u est supprimé de S , sa distance $D[u]$ ne sera plus modifiée par la suite. Le choix de u est crucial : u est choisi comme le sommet de S ayant la plus petite valeur $D[u]$ parmi tous les sommets de S . Un voisin v de u encore présent dans S est mise à jour si la distance de v à s serait plus courte que sa distance courante “en passant par u ”, c'est-à-dire si $D[u] + \ell(u, v) < D[v]$. Si tel est le cas, la valeur de $D[v]$ est changée en $D[u] + \ell(u, v)$. L'algorithme de Dijkstra est décrit en détail ci-après.

```

Algorithme Dijkstra ( $G, s$ )
début
  pour tout  $v \in V$  faire  $D[v] \leftarrow +\infty$ 
   $D[s] \leftarrow 0$ 
   $S \leftarrow V$ 
  tant que  $S \neq \emptyset$  faire
     $u \leftarrow \operatorname{argmin}\{D[v], v \in S\}$ 
     $S \leftarrow S \setminus \{u\}$ 
    pour tout  $v \in S$  voisin de  $u$  faire
      si  $D[u] + \ell(u, v) < D[v]$  alors  $D[v] \leftarrow D[u] + \ell(u, v)$ 
  fin

```

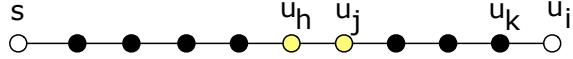


FIGURE 10 – Chemin P dans la preuve de correction de l’algorithme de Dijkstra

Preuve de la correction de l’algorithme. Pour prouver que l’algorithme de Dijkstra est correct, notons $\delta(v) = \text{dist}_G(s, v)$ la distance de v à s dans G . Notons que lorsqu’un sommet u sort de S , la valeur de $D[u]$ restera identique jusqu’à la fin de l’exécution de l’algorithme. Soit u_1, u_2, \dots, u_n les sommets de G indexés par l’ordre dans lequel ils sortent de S durant l’exécution de l’algorithme. On a donc $u_1 = s$.

- Nous allons tout d’abord montrer qu’à la fin de l’exécution de l’algorithme, on a

$$D[v] \geq \delta(v) \text{ pour chaque sommet } v \text{ de } G.$$

La démonstration repose sur une notion d’*invariant* : on suppose une propriété vraie avant l’exécution d’une itération de la boucle « tant que », et on montre qu’elle reste vraie après l’itération de cette boucle.

Supposons que, jusqu’à la k ème itération de la boucle « tant que », $D[v] \geq \delta(v)$ pour tout sommet v , et considérons la $(k+1)$ ème itération. Lors de cette itération, seules les valeurs de D des voisins de u_{k+1} dans S sont mises à jour. Soit $v \in S$ un voisin de u_{k+1} .

- Si $D[u_{k+1}] + \ell(u_{k+1}, v) \geq D[v]$, alors la valeur de $D[v]$ n’est pas modifiée, et on a bien toujours $D[v] \geq \delta(v)$ après l’itération $k+1$.
- Si $D[u_{k+1}] + \ell(u_{k+1}, v) < D[v]$, alors, après la $(k+1)$ ème itération, on a

$$D[v] = D[u_{k+1}] + \ell(u_{k+1}, v) \geq \delta(u_{k+1}) + \ell(u_{k+1}, v) \geq \delta(v)$$

où la dernière inégalité provient du fait qu’un chemin de s à v passant par u_{k+1} a une longueur au moins égale à celle d’un plus court chemin de s à v .

La propriété $D[v] \geq \delta(v)$ pour chaque sommet v est donc bien préservée à chaque itération.

- Nous allons maintenant montrer qu’à la fin de l’exécution de l’algorithme, on a

$$D[v] = \delta(v) \text{ pour chaque sommet } v \text{ de } G.$$

Supposons qu’il existe un sommet v tel que $D[v] > \delta(v)$ à la fin de l’exécution de l’algorithme. Soit alors $i > 1$ le plus petit indice tel que $D[u_i] > \delta(u_i)$. Soit P un plus court chemin de s à u_i .

Montrons tout d’abord que le chemin P ne passe pas par une arête $\{u_k, u_i\}$ avec $1 \leq k \leq i-1$. En effet, la sortie de S du sommet u_k a entraîné une mise à jour de $D[u_i]$, de telle façon que

$$D[u_i] \leq D[u_k] + \ell(u_k, u_i) = \delta(u_k) + \ell(u_k, u_i) \leq \delta(u_i)$$

en contradiction avec l’hypothèse $D[u_i] > \delta(u_i)$. Le sommet u_k juste avant u_i sur P satisfait donc $k > i$.

Soit donc u_j le premier sommet rencontré en parcourant P en partant de s tel que $j > i$. Par définition, le sommet u_j se trouve donc entre s et u_k sur le chemin P . Soit $h \in \{1, 2, \dots, i-1\}$ l’indice du sommet u_h juste avant u_j dans P en partant de s . On a donc $h < i < j$. Par ailleurs, $D[u_h] = \delta(u_h)$ car $h \leq i-1$. Enfin, comme u_j et u_h sont sur un plus court chemin de s à u_i , on a

$$\delta(u_j) = \delta(u_h) + \ell(u_j, u_h).$$

Or, lorsque u_h est sorti de S à la h ème itération, la valeur de $D[u_j]$ a été traitée, impliquant que

$$D[u_j] \leq D[u_h] + \ell(u_j, u_h) = \delta(u_k) + \ell(u_j, u_h) = \delta(u_j).$$

Donc $D[u_j] = \delta(u_j)$ à la h ème itération, et donc également à la i ème itération. Or $\delta(u_i) > \delta(u_j)$ car les arêtes ont des poids strictement positifs. Cela implique que, à la i ème itération, on a

$$D[u_i] > \delta(u_i) > \delta(u_j) = D[u_j].$$

Cette dernière inégalité est en contradiction avec le choix de u_i par l'algorithme à l'étape i , puisqu'il aurait dû préférer u_j à u_i car le sommet choisi est un sommet de plus petite valeur de D .

Supposer qu'il existe un sommet v tel que $D[v] > \delta(v)$ à la fin de l'exécution de l'algorithme conduit donc à une contradiction. En conséquence, on a bien $D[v] = \delta(v)$ pour tout sommet v à la fin de l'algorithme. \square

Temps d'exécution. Si G possède n sommets, alors l'initialisation de D prend un temps $O(n)$. La boucle « tant que » est répétée n fois puisque $|S| = n$ avant la première itération, et chaque itération fait décroître $|S|$ de un sommet. Notons $t_{min}(n)$ le temps pour trouver la valeur minimum d'un tableau d'au plus n valeurs. Les n recherches de minimum prennent donc un temps $O(n t_{min}(n))$. Enfin, chaque arête $\{u, v\}$ n'est considérée qu'un fois en tout dans l'algorithme. Donc, si le nombre d'arêtes de G est m , le temps d'exécution de l'algorithme de Dijkstra est $O(n t_{min}(n) + m)$ avec le graphe stocké en listes d'adjacence. On a $t_{min}(n) = O(n)$, et donc le temps d'exécution de l'algorithme de Dijkstra est $O(n^2 + m) = O(n^2)$. En stockant S sous forme d'une liste de sommets, en maintenant cette liste triée par ordre croissant des valeurs dans D (en utilisant des structures de données telles qu'un tas binaire ou un tas de Fibonacci), la complexité de l'algorithme peut être réduite à $O(m + n \log n)$. Notez que trier une liste de n entiers peut s'effectuer en temps $O(n \log n)$. Le coût de l'algorithme de Dijkstra est donc essentiellement le temps de lire les valeurs des m longueurs d'arêtes, plus le temps d'un tri.

3.2 Algorithmes glouton

Etant donné un problème à résoudre, un algorithme glouton pour ce problème construit une solution globale en une suite de *choix*. A chaque étapes, un algorithme glouton effectue le meilleur choix possible parmi les choix disponibles.

Attention, faire le meilleurs choix à chaque étape ne conduit pas nécessairement à calculer une solution optimale. Par exemple, considérons le problème consistant à rendre la monnaie avec le moins de pièces possible. Un commerçant aguerri applique l'algorithme glouton consistant à appliquer la règle suivante :

Rendre la pièce de plus grande valeur inférieure ou égale à ce qu'il reste à rendre.

Cela semble intuitivement la méthode la plus efficace puisqu'à chaque étape on fait le meilleurs choix, au sens de celui qui conduit à minimiser le montant qu'il reste à rendre. Cette méthode est effectivement optimale pour les euros, les dollars, les yens, etc. (Montrez cela à titre d'exercice). Elle n'est néanmoins pas optimale pour tout type de systèmes monétaires ! Par exemple, les monnaies anglaises comptaient encore au début des années 70 :

- la livre, valant 20 shillings,
- quatre subdivisions du shilling : le penny ($1/12$ de shilling), et trois pièces de valeurs respectives $1/4$, $1/3$, et $1/2$ shillings, soit respectivement 3, 4, et 6 pence (pluriel de penny).

Dans ce système au charme carolingien, considérons la stratégie optimale pour rendre 8 pence. On a 8 pence $= \frac{8}{12}$ de shilling, soit $\frac{2}{3}$ de shilling.

- l'optimal consiste à rendre deux pièces de $1/3$ de shilling (deux tiers de shilling font bien 8 pence !) ;
- l'algorithme glouton rend une pièce $1/2$ shilling, puis deux pièces d'un penny, soit trois pièces en tout.

Si les algorithmes gloutons se sont pas systématiquement optimaux, ils se révèlent néanmoins très performants dans de nombreux contextes (même outre Manche, en tout cas depuis la réforme de décimalisation de 1971).

3.2.1 Arbre couvrant de poids minimum

Le problème de l'arbre couvrant de poids minimum (MST, pour *minimum spanning tree*) est le suivant :

Le problème MST :

Entrée : un graphe $G = (V, E)$ et une fonction de poids $w : E \rightarrow \mathbb{N}$;

Objectif : trouver dans G un arbre couvrant tous les sommets, et dont la somme des poids des arêtes est minimum.

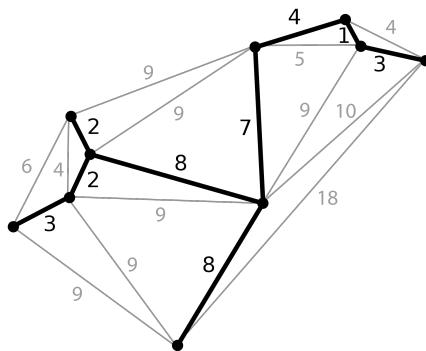


FIGURE 11 – Un arbre couvrant de poids minimum dans un graphe (de wikipedia)

Nous allons décrire un algorithme glouton (en fait une famille d'algorithmes gloutons) pour résoudre ce problème. Ces algorithmes reposent sur le concept de *coupe*. Une coupe dans un graphe $G = (V, E)$ est définie par une paire (S, S') d'ensembles de sommets non vides et disjoints telle que $S \cup S' = V$. Les arêtes ayant une extrémité dans S et une autre dans S' sont les arêtes de la coupe. On dit qu'une coupe *préserve* un ensemble d'arêtes $A \subseteq E$ si aucune arête de A n'est dans la coupe.

Algorithme générique de calcul de MST :

```

 $A \leftarrow \emptyset$ 
tant que  $A$  n'est pas un arbre couvrant faire
    Soit  $(S, V \setminus S)$  une coupe quelconque de  $G$  qui préserve  $A$  ;
    Soit  $e$  une arête de la coupe, de poids minimum ;
     $A \leftarrow A \cup \{e\}$  ;

```

Il y a un nombre potentiellement exponentiel de coupes dans un graphe. Il convient donc de se restreindre à des coupes particulières pour obtenir un algorithme polynomial. Les algorithmes de Borůvka, de Prim et de Kruskal décrits brièvement ci-dessous sont trois mises en oeuvre possibles de l'algorithme générique.

- L'algorithme de Borůvka (1926) maintient une forêt couvrante (un ensemble d'arbres dont l'ensemble de tous les sommets couvre le graphe), et ajoute l'arête de poids minimum joignant deux arbres. En d'autres termes, la coupe considérée est celle entre deux forêts dont l'union couvre les sommets du graphe. Initialement, l'algorithme de Kruskal part d'une forêt de n arbres, où chaque arbre est réduit à un unique sommet. La première arête choisie par Borůvka est donc celle de poids minimum.
- L'algorithme de Prim (1957) maintient un arbre, et ajoute à l'arbre courant l'arête incidente de poids minimum. En d'autres termes, la coupe considérée est celle entre les sommets couverts par l'arbre partiel A , et ceux non encore couverts. Initialement, l'algorithme de Prim part de l'arbre réduit à un seul sommet u_0 , choisi arbitrairement. La première arête choisie par Prim est donc l'arête de poids minimum parmi celles incidente à u_0 .
- L'algorithme de Kruskal (1956) maintient une forêt non nécessairement couvrante, et ajoute l'arête de poids minimum permettant d'agrandir la forêt en couvrant au moins un sommet de plus. Plus précisément, les arêtes sont triées par ordre croissant de leur poids. Elles sont ensuite considérées une par une dans cet ordre. Lorsque l'on considère l'arête e , on agit comme suit : si e ne forme pas un cycle avec des arêtes déjà retenues, alors on retient cette arête, sinon on ne la retient pas et on passe à l'arête suivante dans l'ordre croissant des poids. En d'autres termes, la coupe considérée est celle entre les sommets de la forêt courante et les sommets non encore couverts. La première arête choisie par Kruskal est donc celle de poids minimum.

Ces trois algorithmes sont polynomiaux. Pour s'en convaincre, il suffit d'une analyse très grossière. Dans un graphes de n sommets, chacun des algorithmes s'exécute en $n - 1$ étapes puisqu'un arbre de n sommets possède $n - 1$ arêtes. Chacune des étapes est une simple recherche d'une arête de poids minimum dans un ensemble d'au plus m arêtes, où m est le nombre d'arêtes du graphe. On obtient donc un temps d'au plus $O(nm)$. En fait, cette analyse est très grossière, et on peut montrer que Prim et Kruskal s'exécutent en temps $O(m \log n)$, et Borůvka en temps $O(m \log m)$.

Preuve de la correction de l'algorithme générique. Cette preuve démontre également la justesse de Borůvka, Prim et Kruskal, puisque ces derniers ne sont que des mises en oeuvre particulières de l'algorithme générique. Considérons un ensemble (potentiellement vide) d'arêtes A inclus dans l'ensemble des arêtes d'un arbre couvrant de poids minimum. Soit (S, S') une coupe quelconque de G qui préserve A . Soit $e = \{u, v\}$ une arête de la coupe, de poids minimum. Nous allons montrer que $A \cup \{e\}$ est inclus dans l'ensemble des arêtes d'un arbre couvrant de poids minimum. Soit T un arbre couvrant de poids minimum tel que $A \subseteq E(T)$, et supposons que $A \cup \{e\} \not\subseteq E(T)$. Les sommets u et v sont de part et d'autre de la coupe $(S, V \setminus S)$. Considérons le chemin P entre u et v dans T . Ce chemin doit traverser la coupe $(S, V \setminus S)$ au moins une fois.

Soit donc $e' = \{u', v'\}$ une arête de la coupe empruntée par P . Soit T' l'arbre obtenu à partir de T en remplaçant e' par e . T' est bien un arbre car, d'une part, si $P \cup \{e\}$ forme un cycle, la suppression de e' élimine ce cycle, et, d'autre part, T' est bien connexe puisqu'on peut aller de u' à v' en suivant une partie de P , l'arête e , puis une autre partie de P . Enfin, on a

$$w(T') = w(T) - w(e') + w(e).$$

Or e est une arête de poids minimum parmi les arêtes de la coupe $(S, V \setminus S)$, et e' est une arête de cette coupe. Donc $w(e) \leq w(e')$, donc $w(T') \leq w(T)$, et donc T' est un arbre couvrant de poids minimum. Ainsi $A \cup \{e\}$ est bien inclus dans l'ensemble d'arêtes d'un arbre couvrant de poids minimum.

3.2.2 Matroïdes

En fait, l'algorithme glouton donne toujours une solution optimale dès qu'il existe une structure de *matroïde* sous-jacente. Les matroïdes ont été définis par Whitney (1935) comme structure formelle permettant de capturer des propriétés telles que le théorème de la base incomplète, et plus généralement les propriétés des familles indépendantes de vecteurs.

Definition 6 Soit $M = (S, \mathcal{I})$, où S est un ensemble et $\mathcal{I} \subseteq \mathcal{P}(S)$. M est un matroïde si les deux propriétés ci-dessous sont satisfaites :

1. *héritage* : si $A \subseteq B$ et $B \in \mathcal{I}$ alors $A \in \mathcal{I}$;
2. *échange* : si $A \in \mathcal{I}$, $B \in \mathcal{I}$, et $|A| < |B|$ alors il existe $x \in B \setminus A$ tel que $A \cup \{x\} \in \mathcal{I}$.

Les ensembles de \mathcal{I} sont dit indépendants.

Un exemple de matroïde est $S = \{\text{vecteurs de } \mathbb{R}^d\}$ et $\mathcal{I} = \{\text{familles libres de vecteurs}\}$. Nous nous intéresserons ici principalement aux matroïdes finis, c'est-à-dire S est un ensemble fini. Dans ce cadre, un exemple de matroïde est celui des matroïdes *graphiques*. Soit $G = (V, E)$ un graphe. On lui associe alors le matroïde : $S = E$, et $\mathcal{I} = \{\text{forêts}\}$. (Rappelons qu'un forêt est un ensemble d'arbres disjoints.)

Le lemme suivant est une conséquence triviale de la propriété d'échange :

Lemma 1 Tous les ensembles maximaux d'un matroïde ont la même cardinalité.

Nous allons nous intéresser à une propriété plus générale. Considérons un matroïde $M = (S, \mathcal{I})$ pondéré, c'est-à-dire muni d'un fonction de poids $w : S \rightarrow \mathbb{Q}^+$. On définit le poids de $I \in \mathcal{I}$ comme $w(I) = \sum_{x \in I} w(x)$. Nous allons montrer que l'algorithme ci-dessous trouve un ensemble indépendant de poids maximum.

Algorithme générique de calcul d'un ensemble indépendant de poids maximum :

```

 $A = \emptyset$ ;
tant que il existe  $x \in S$  t.q.  $A \cup \{x\} \in \mathcal{I}$  faire
    Choisir l'élément  $x \in S$  t.q.  $A \cup \{x\} \in \mathcal{I}$  de poids maximum ;
     $A \leftarrow A \cup \{x\}$ ;
```

La complexité de cet algorithme est $O(n \log n + n f(n))$ où $n = |S|$ et $f(n)$ est le temps requis pour vérifier que $A \cup \{x\} \in \mathcal{I}$ où $A \in \mathcal{I}$ et $x \in S$. En effet, on commence par trier les n éléments de S par ordre décroissant de poids (ce tri prend un temps $O(n \log n)$), puis on les

considère un après l'autre, en écartant les éléments qui ne satisfont pas le test $A \cup \{x\} \in \mathcal{I}$ (il y a n tests, et chacun prend un temps $f(n)$).

Montrons que l'algorithme générique calcule effectivement un ensemble indépendant de poids maximum. On effectue une preuve par récurrence sur la taille de S . Si $S = \emptyset$ alors $A = \emptyset$ est bien le seul ensemble indépendant. Son poids est 0, ce qui est bien le poids d'un ensemble indépendant dans le matroïde \emptyset , et l'algorithme générique est donc bien optimal. Supposons donc maintenant que l'algorithme générique soit optimal pour tous les matroïdes de taille au plus n , et montrons qu'il est optimal pour les matroïdes de taille $n + 1$. Soit $M = (S, \mathcal{I})$ un matroïde de tailles $n + 1$. Soit A un ensemble indépendant de poids maximum. Par la propriété d'hérédité, pour tout $x \in A$, on a $\{x\} \in \mathcal{I}$. Ainsi il existe des éléments $x \in S$ tels que $\{x\} \in \mathcal{I}$. Soit donc $x \in S$ un élément de poids max t.q. $\{x\} \in \mathcal{I}$, et définissons $M' = (S', \mathcal{I}')$ par :

$$S' = \{y \in S \setminus \{x\} \mid \{x, y\} \in \mathcal{I}\} \text{ et } \mathcal{I}' = \{A' \subseteq S \setminus \{x\} \mid A' \cup \{x\} \in \mathcal{I}\}.$$

On montre facilement que M' est bien un matroïde (exercice). Par hypothèse de récurrence, l'algorithme générique calcule un ensemble $A' \in \mathcal{I}'$ de poids maximum dans M' . Il ne reste plus qu'à montrer que $A' \cup \{x\}$ est de poids maximum dans M . Pour montrer que $A' \cup \{x\}$ est de poids maximum dans M , montrons tout d'abord que si $x \in S$ est l'élément de poids max t.q. $\{x\} \in \mathcal{I}$, alors il existe un ensemble indépendant A de poids maximum t.q. $x \in A$. A cet effet, soit A un ensemble indépendant de poids maximum, et supposons que $x \notin A$. Soit $k = |A|$. Si $k = 1$, alors $A = \{y\}$ et $w(x) \geq w(y)$, donc $B = \{x\}$ est un ensemble indépendant de poids maximum t.q. $x \in B$. Si $k > 1$, alors, par échanges successifs, on peut rajouter $k - 1$ éléments y_1, \dots, y_{k-1} de $A = \{y_1, \dots, y_{k-1}, y_k\}$ à $\{x\}$ de façon à obtenir un ensemble $B = \{y_1, \dots, y_{k-1}, x\}$. On a

$$w(B) = w(x) + \sum_{i=1}^{k-1} w(y_i) = w(x) + w(A) - w(y_k) \geq w(A).$$

Donc B est un ensemble indépendant de poids maximum t.q. $x \in B$. Fixons donc ensemble indépendant A de poids maximum t.q. $x \in A$. Soit $A'' = A \setminus \{x\}$. On a $A'' \in \mathcal{I}$, donc $w(A'') \leq w(A')$ et donc $w(A) = w(A'' \cup \{x\}) \leq w(A' \cup \{x\})$, ce qui implique que $A' \cup \{x\}$ est de poids maximum dans M .

3.3 Algorithmes récursifs

L'exemple d'école est le calcul de la factoriel d'un entier : $n! = n(n-1)(n-2)\dots 1$. En effet, la définition de factoriel est récursive : $n! = n \cdot (n-1)!$ pour $n > 1$, et $1! = 1$.

Fonction factoriel(entier $n \geq 1$)
si $n = 1$ **alors** retourner 1
sinon retourner $n \cdot$ factoriel($n - 1$).

Notez que la récursivité est une facilité d'écriture, mais n'augment pas la puissance du modèle RAM. De fait, il est toujours possible de « dé-récurser » un algorithme récursif (c'est d'ailleurs ce que fait tout compilateur transformant un programme potentiellement récursif écrit en langage de haut niveau en un programme écrit en assembleur). On peut par exemple réécrire la fonction factoriel ci-dessus en

Fonction factoriel(entier $n \geq 1$)

```

 $F \leftarrow 1$ 
pour  $i = 1$  à  $n$  faire
     $F \leftarrow F \times i$ 
retourner  $F$ 

```

La récursivité est une technique algorithmique utile lorsqu'il est possible d'obtenir une solution à un problème de taille n à partir de la combinaison de solutions de problèmes indépendant de tailles plus petites que n . Les algorithmes récursifs reposent sur le principe *diviser pour régner* (« divide and conquer » en anglais) consistant à (1) diviser le domaine de recherche en sous-domaines indépendants plus petits, (2) traiter séparément ces sous-domaines, puis (3) construire la solution du problème original par concaténation des solutions des sous-problèmes. Un exemple classique est le tri d'une liste L de n entiers quelconques (n est la *taille* $|L|$ de la liste) :

Le problème TRI :

Entrée : une liste L d'entiers ;

Objectif : trier L dans l'ordre croissant des entiers de la liste.

Fonction mergesort(liste L de taille ≥ 1)

si $|L| = 1$ **alors** retourner L

sinon

partitionner L en deux listes L_1 et L_2 de tailles égales (à ± 1 près)

trier(L_1)

trier(L_2)

$L \leftarrow \text{fusion}(L_1, L_2)$

retourner L

Cet algorithme fait appel à la fonction « fusion » qui crée une liste ordonnée en fusionnant deux listes ordonnées. On peut aisément vérifier que la fusion de deux listes de tailles k_1 et k_2 se fait en au plus $k_1 + k_2$ comparaisons. L'algorithme ci-après effectue cela, en supposant $L_1[k_1 + 1] = L_2[k_2 + 1] = +\infty$:

Fonction fusion(L_1 de longueur k_1 , L_2 de longueur k_2)

$i \leftarrow 1$

$j \leftarrow 1$

pour $k = 1$ à $k_1 + k_2$ faire

si $L_1[i] \leq L_2[j]$ **alors** ($L[k] \leftarrow L_1[i]$; $i \leftarrow i + 1$)

sinon ($L[k] \leftarrow L_2[j]$; $j \leftarrow j + 1$)

retourner L

La complexité de l'algorithme récursif de tri ci-dessus s'exprime sous la forme

$$T(2k) = 2 \cdot T(k) + 2k$$

où $T(\ell)$ est le nombre de comparaisons de l'algorithme appliqué à une liste de taille au plus ℓ . En effet, l'algorithme divise la liste de taille $2k$ en deux listes de taille k , les trie en effectuant $T(k)$ comparaisons, puis les fusionne en $2k$ comparaisons. Donc, si $T(k) \leq k \log_2 k$, on obtient

$$T(2k) \leq 2k \log_2 k + 2k = 2k(\log_2 k + 1) = 2k \log_2(2k).$$

Le nombre de comparaisons de `mergesort` est donc au plus $O(n \log_2 n)$. Notez que la borne $O(n \log_2 n)$ comparaisons est asymptotiquement optimale. En effet, le théorème ci-dessous montre

que tout algorithme n'effectuant que des comparaisons entre les valeurs de la liste à trier doit effectuer $\Omega(n \log n)$ comparaisons.

Théorème 3 *Tout algorithme de tri n'effectuant que des comparaisons entre les éléments (et aucune autre opération arithmétique) nécessite $\Omega(n \log n)$ comparaisons au pire cas pour trier un tableau de n entiers.*

Preuve. Soit A un algorithme de tri. Soit Σ_n l'ensemble des permutations possibles de n éléments. Chaque permutation $\pi \in \Sigma_n$ correspond à une entrée possible de l'algorithme A . L'algorithme A remplace les éléments de π par ordre croissant de leurs valeurs en les permutant. De fait, l'algorithme A effectue π^{-1} . La question est : combien de comparaisons A doit-il effectuer au pire cas sur π pour réaliser π^{-1} ?

Une première comparaison entre deux éléments $L[i]$ et $L[j]$ de la liste permet à A de distinguer les permutations en entrée pour lesquelles $L[i] < L[j]$ de celles pour lesquelles $L[i] > L[j]$. Un de ces deux ensembles a une taille au moins $\frac{1}{2}n!$. Supposons que l'entrée soit dans le plus grand de ces deux ensembles de permutations.

Effectuer une seconde comparaison entre deux éléments $L[i']$ et $L[j']$ de la liste permettra de distinguer, parmi cet ensemble d'au moins $\frac{1}{2}n!$ permutations, les permutations pour lesquelles $L[i'] < L[j']$ de celles pour lesquelles $L[i'] > L[j']$. Un de ces deux ensembles a une taille au moins $\frac{1}{4}n!$. Supposons que l'entrée soit dans le plus grand de ces deux ensembles de permutations.

Ainsi de suite : après k comparaisons, l'algorithme A peut-être amené à réaliser le tri au sein d'un ensemble d'au moins $\frac{1}{2^k}n!$ permutations. Or, pour être certain d'avoir trié la liste en entrée, il faut obtenir un ensemble réduit à une unique permutation. En effet, si l'algorithme termine sans avoir distingué deux permutations $\pi_1 \neq \pi_2$, chacune compatible avec toutes les comparaisons effectuées jusqu'alors par A , alors A ne trie pas correctement au moins une de ces deux permutations. Trier nécessite donc au moins k comparaisons, où k vérifie $\frac{n!}{2^k} \leq 1$. On a donc $2^k \geq n!$, et donc $k \geq \log_2(n!) = \Omega(n \log n)$ comparaisons. \square

Attention : la borne $\Omega(n \log_2 n)$ porte sur le nombre de comparaisons. Par ailleurs, elle ne vaut que si aucune information n'est connue a priori sur la liste à trier. Par exemple, si l'on sait que les n éléments de la liste à trier L sont deux à deux distincts dans $[1, N]$, alors on peut trier L en temps $O(N)$ par un algorithme qui n'effectue aucune comparaison :

Fonction `boundsort`(liste L de taille n , avec des éléments distincts dans $[1, N]$)

```

pour  $i = 1$  à  $N$  faire  $T[i] \leftarrow 0$ 
pour  $i = 1$  à  $n$  faire  $T[L[i]] \leftarrow 1$ 
 $j \leftarrow 1$ 
pour  $i = 1$  à  $N$  faire
    si  $T[i] = 1$  alors
         $L[j] \leftarrow i$ 
         $j \leftarrow j + 1$ 
retourner  $L$ 
```

L'algorithme `mergesort` est optimal en nombre de comparaisons, mais il nécessite $\Theta(n \log_2 n)$ comparaisons systématiquement, même si la liste L en entrée est déjà triée ! L'algorithme `quicksort` ci-dessous nécessite $O(n^2)$ comparaisons au pire cas, mais $O(n \log n)$ en moyenne sur toute les permutations possibles. Egalement, il s'avère plus rapide que l'algorithme par fusion récursive, en partie pour des questions liées à une meilleure utilisation des caches. Il est de fait

souvent préféré aux autres algorithmes récursifs⁵.

Fonction quicksort(liste L de taille ≥ 1)

si $|L| = 1$ **alors** retourner L

sinon

 choisir un élément x de la liste

 placer tous les éléments de la liste plus petits que x dans la liste L_{petits}

 placer tous les éléments de la liste plus grands que x dans la liste L_{grands}

$L \leftarrow (\text{quicksort}(L_{petits}), x, \text{quicksort}(L_{grands}))$

 retourner L

Le choix de x , appelé *pivot*, peut se faire de différentes façons, arbitrairement en choisissant un élément d'indice quelconque dans la liste (le premier, celui du milieu, etc.) ou bien aléatoirement.

Remarque. La récursivité est une technique puissante de conception d'algorithmes lorsque le principe diviser pour régner s'applique. Cette technique attrayant par sa simplicité peut toutefois conduire à des algorithmes inefficaces lorsque ce principe ne s'applique pas. Par exemple, dans le cas du calcul des coefficients binomiaux

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1},$$

utiliser la récursivité conduit à un algorithme exponentiel. Cela provient du fait que la récursivité conduira à recalculer plusieurs fois certains coefficients binomiaux : le principe diviser pour régner ne s'applique pas car les sous-problèmes ne sont pas indépendants. En revanche, remplir le triangle de Pascal jusqu'à la ligne n peut se faire en temps $O(n^2)$.

3.4 Programmation dynamique

L'idée de la programmation dynamique est d'identifier des sous-problèmes, et de combiner les solutions à ces sous-problèmes pour obtenir des solutions à des sous-problèmes de plus grande taille. Intuitivement, la programmation dynamique procède de façon duale de la méthode gloutonne. Cette dernière considère toutes les possibilités, et se restreint au fur et à mesure en choisissant à chaque étape la meilleure possibilité courante. A l'inverse, la programmation dynamique part d'un ensemble de solutions triviales à des sous-problèmes élémentaires, et procède en fusionnant les solutions de sous-problèmes jusqu'à obtenir la solution globale. La paternité de la programmation dynamique est attribuée à Bellman, intéressé, dans les années 1940, à la résolution de problèmes d'optimisations dont nous allons voir plusieurs exemples ci-après. Notons que la signification de « programmation dynamique » et l'historique de ce choix terminologique sont sujets à débats. Le mot « programmation » n'avait certes pas exactement la même signification dans les années 40 que de nos jours (le terme faisait alors référence à « planification » plutôt qu'à programmation au sens informatique). Par ailleurs, le terme « dynamique » fut semble-t-il introduit pour évoquer l'idée d'évolution dans le temps. Les algorithmes de programmation dynamique présentés dans ce chapitre effectuent en effet des décisions sur des systèmes dont la taille augmente, ce qui peut être vu comme une évolution d'un paramètre temporel.

5. Quicksort est par exemple utilisé dans le système d'exploitation Linux. L'invention de quicksort est attribuée à Tony Hoare (l'inventeur de la logique éponyme) qui développa cet algorithme en 1960 lors d'un séjour à l'université de Moscou, motivé par le besoin de trier alphabétiquement les mots d'un fichier afin de les comparer à ceux d'un dictionnaire Russe-Anglais.

3.4.1 Plus longue sous-séquence commune

Une molécule d'ADN peut être vue comme séquence sur l'alphabet $\{a, c, g, t\}$. On dit qu'une séquence S est une sous-séquence de S' s'il existe un suite croissante d'indices i_j , $j = 1, \dots, |S|$ t.q. $S_j = S'_{i_j}$. La proximité entre espèce, ou entre membres d'une même espèce, peut être mesurée en estimant des distances entre molécules d'ADN. Une distance possible est la longueur de la plus longue sous-séquence commune (PLSSC) :

Le problème PLUS LONGUE SOUS-SÉQUENCE COMMUNE (PLSSC) :

Entrée : X et Y deux séquences de lettres dans un alphabet Σ ;

Objectif : trouver une sous-séquence commune à X et Y de longueur maximale.

Une façon de procéder pourrait consister à tester toutes les sous-séquences possibles. Il y a cependant un nombre exponentiel de sous-séquences à tester, ce qui rend cette approche inefficace. Etudions plutôt ce qu'il est possible de faire à partir de la solution à des sous-problèmes. Soit $X^{(i)}$ le préfixe de X défini par les i premiers éléments de X . Définissons alors $\ell_{i,j}$ comme la longueur d'une PLSSC à $X^{(i)}$ et $Y^{(j)}$. On a :

$$\ell_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ \ell_{i-1,j-1} + 1 & \text{si } i, j > 0 \text{ et } X_i = Y_j \\ \max\{\ell_{i,j-1}, \ell_{i-1,j}\} & \text{si } i, j > 0 \text{ et } X_i \neq Y_j \end{cases}$$

Calculer les $\ell_{i,j}$ au moyen d'un algorithme récursif conduirait à une complexité exponentiel. En effet, soit $t(k)$ le temps de la procédure récursive appliquée à des sous-séquences $X^{(i)}$ et $Y^{(j)}$ de longueur totale $i + j \leq k$. On a $t(k) \leq 2t(k-1)$ puisqu'il faut calculer $\ell_{i,j-1}$ et $\ell_{i-1,j}$, ce qui conduit à la borne exponentielle $t(n) \leq 2^n$ où n est la somme des longueurs des séquences en entrées. Le caractère exponentiel de cette approche provient du fait que l'algorithme récursif recalcule plusieurs fois les mêmes sous-problèmes. Il faut en effet remarquer ici les nombreuses redondances dans les résolutions de sous-problèmes. Ainsi, le sous-problème (i, j) dépend des sous-problèmes $(i-1, j)$, $(i, j-1)$ et $(i-1, j-1)$. En conséquence, le sous-problème $(i-1, j-1)$ intervient dans la résolution des deux sous-problèmes : $(i-1, j)$ et $(i, j-1)$. L'algorithme récursif est exponentiel car il recalcule plusieurs fois les solutions aux mêmes sous problèmes. Une solution plus économique consiste à stocker les solutions partielles, par exemple dans un tableau L , de la façon suivante. Soit $|X|$ et $|Y|$ les longueurs respectives des séquences X et Y .

```

pour  $i = 0$  à  $|X|$  faire  $L[i, 0] \leftarrow 0$  ;
pour  $j = 0$  à  $|Y|$  faire  $L[0, j] \leftarrow 0$  ;
pour  $i = 1$  à  $|X|$  faire
    pour  $j = 1$  à  $|Y|$  faire
        si  $X_i = Y_j$  alors  $L[i, j] \leftarrow L[i - 1, j - 1] + 1$ 
        sinon  $L[i, j] \leftarrow \max\{L[i, j - 1], L[i - 1, j]\}$  ;

```

L'algorithme ci-dessus calcule bien la longueur de la plus longue sous-séquence commune à X et Y , mais il ne retourne pas cette séquence. Pour obtenir la PLSSC explicitement, on peut procéder comme suit :

```

pour  $i = 1$  à  $|X|$  faire  $L[i, 0] \leftarrow 0$  ;
pour  $j = 1$  à  $|Y|$  faire  $L[0, j] \leftarrow 0$  ;

```

```

pour  $i = 1$  à  $|X|$  faire
    pour  $j = 1$  à  $|Y|$  faire
        si  $X_i = Y_j$  alors
             $L[i, j] \leftarrow L[i - 1, j - 1] + 1$ 
             $p[i, j] \leftarrow \text{diag}$ 
        sinon si  $L[i - 1, j] \leq L[i, j - 1]$  alors
             $L[i, j] \leftarrow L[i, j - 1]$ 
             $p[i, j] \leftarrow \text{haut}$ 
        sinon
             $L[i, j] \leftarrow L[i - 1, j]$ 
             $p[i, j] \leftarrow \text{gauche}.$ 

```

Pour obtenir la PLSSC, on peut alors utiliser la procédure récursive suivante, initialement appelée avec le quadruplet $(p, X, |X|, |Y|)$:

```

Imprimer-PLSSC( $p, X, i, j$ ) :
    si  $i = 0$  ou  $j = 0$  alors print(.)
    sinon si  $p[i, j] = \text{diag}$  alors
        Imprimer-PLSSC( $p, X, i - 1, j - 1$ )
        print( $X_i$ )
    sinon si  $p[i, j] = \text{haut}$  alors
        Imprimer-PLSSC( $p, X, i, j - 1$ )
    sinon Imprimer-PLSSC( $p, X, i, j - 1$ )

```

Tous ces algorithmes prennent un temps $O(|X| \cdot |Y|)$, et donc un temps $O(n^2)$ pour des séquences de longueur au plus n .

3.4.2 Programmation dynamique dans les arbres

Soit $b(G, s)$ le nombre minimum d'étapes nécessaire pour faire une diffusion dans G à partir de s vers tous les autres sommets, dans le modèle suivant : à chaque étape, un sommet informé ne peut informer qu'au plus un seul de ses voisins.

Le problème DIFFUSION :

Entrée : un graphe $G = (V, E)$ et un sommet $s \in V$;
Objectif : calculer $b(G, s)$.

On a vu que problème de décision associé à DIFFUSION est NP-complet (on en en fait vu cela pour les graphes orientés, mais le problème est également NP-complet dans les graphes non-orientés). En revanche, l'exercice suivant permet de montrer que DIFFUSION est polynomial lorsque l'on se restreint aux arbres.

Exercice 4 Proposer un algorithme polynomial de programmation dynamique permettant de calculer $b(T, s)$ pour tout arbre T et tout sommet s de T .

3.4.3 Problème du sac à dos

Le problème SAC À DOS :

Entrée : un ensemble de n objets, de poids positifs w_1, \dots, w_n et de valeurs positives p_1, \dots, p_n , et un entier $W \geq 0$;

Objectif : trouver un ensemble S d'objets dont la somme des poids est au plus W , et dont la valeur totale est maximum.

L'objectif du problème SAC À DOS est donc de maximiser $\sum_{i \in S} p_i$ sous la contrainte $\sum_{i \in S} w_i \leq W$. Rendre la monnaie avec le nombre minimum de pièce peut s'exprimer sous la forme de celui du sac à dos. On doit en effet rendre la monnaie pour une valeur de W . On dispose de n différentes valeurs de pièces v_1, \dots, v_n . Soit $x_i \geq 0$ le nombre de pièce de valeur v_i utilisé pour rendre la monnaie. Il s'agit de minimiser $\sum_{i=1}^n x_i$ sous la contrainte $\sum_{i=1}^n x_i v_i = W$.

L'algorithme glouton consistant à toujours choisir l'objet j qui maximise le gain relatif p_j/w_j ne marche pas pour résoudre le problème du sac à dos. Par exemple, considérons trois objets de caractéristique (10kg, 60€), (20kg, 100€), (30kg, 120€) avec sac de $W=50\text{kg}$. L'algorithme glouton choisira d'abord le premier objet, d'une valeur de 60€, pesant 10kg, puis le deuxième objet d'une valeur de 10€, pesant 20kg. A ce stade, l'algorithme glouton stoppe, puisqu'il n'est plus possible de rajouter d'objet sans excéder le poids de 50kg. Remarquant qu'il n'est pas possible de mettre les trois objets dans le sac, on constate que la solution optimale consiste à prendre les deux objets de poids 20 et 30kg, pour une valeur totale de 220€, contre 160€ pour le glouton. Le problème du glouton vient qu'il termine avec un sac disposant d'espace, mais insuffisant pour rajouter un objet de plus. En revanche, le glouton marche parfaitement pour le sac à dos fractionnaire :

Exercice 5 Montrer qu'il existe un algorithme glouton pour le sac à dos fractionnaire, où l'on est autorisé à ne prendre qu'une fraction $\alpha \in [0, 1]$ d'un objet, auquel cas on ne charge le sac que de la fraction α du poids de l'objet, pour un bénéfice d'une fraction α de la valeur de l'objet.

Revenons au sac à dos standard (c'est-à-dire non fractionnaire). Il existe un algorithme de programmation efficace pour ce problème, à la restriction près que cet algorithme n'est pas polynomial, mais seulement *pseudo-polynomial*. Un algorithme est pseudo-polynomial s'il est polynomial en la *valeur* des entiers donnés en entrée, ou encore polynomial en la taille des entrées si les entiers sont donnés en *unaire* au lieu d'être naturellement codés en binaire. Faire l'exercice ci-dessous et consulter sa correction pour un exemple d'algorithme pseudo-polynomial.

Exercice 6 Donner un algorithme utilisant la programmation dynamique pour résoudre SAC À DOS utilisant des sous-problèmes impliquant des sacs de capacités c , pour tout $c \in \{1, \dots, W\}$.

Notez qu'il semble a priori difficile de concevoir un algorithme polynomial pour résoudre le problème du sac à dos car ce problème est NP-complet par réduction polynomiale à partir du problème NP-complet COUVERTURE EXACTE PAR TRIPLETS. Ce problème consiste à décider si un ensemble donné U peut être couvert par un sous-ensemble de triplets d'éléments de U deux-à-deux disjoints, sélectionnés dans une famille donnée \mathcal{S} de triplets d'éléments de U . Plus spécifiquement, le problème est le suivant :

Le problème COUVERTURE EXACTE PAR TRIPLETS :

Entrée : un ensemble fini U et une collection $\mathcal{S} = \{S_1, \dots, S_r\}$ de triplets d'éléments de U ;

Question : Existe-t-il $I \subseteq \{1, \dots, r\}$ tel que $\cup_{i \in I} S_i = U$ et $S_i \cap S_j = \emptyset$ pour tout $i, j \in I, i \neq j$?

Exercice 7 Montrer, par réduction polynomiale à partir de COUVERTURE EXACTE PAR TRIPLETS que SAC À DOS est NP-complet.

3.5 Flots et applications

3.5.1 Réseaux de transport

Un *réseau de transport* est un graphe orienté $G = (V, E)$, munis de capacités entières $c(u, v) \geq 0$ pour chaque $(u, v) \in E$ (si $(u, v) \notin E$ alors $c(u, v) = 0$), d'une *source* $s \in V$, et d'un *puits* $t \in V$.

Un *flot* f dans G est une fonction $f : V \times V \rightarrow \mathbb{Q}$ telle que :

$$P1 : \forall (u, v) \in V \times V, f(u, v) \leq c(u, v);$$

$$P2 : \forall (u, v) \in V \times V, f(u, v) = -f(v, u);$$

$$P3 : \forall u \in V \setminus \{s, t\}, \sum_{v \in V} f(u, v) = 0.$$

La première contrainte (P1) exprime le fait qu'on ne peut pas acheminer plus de flot de u vers v que la capacité de l'arc (u, v) . La deuxième contrainte (P2) exprime le fait que si une quantité x de flot est acheminé de u vers v alors on doit acheminer une quantité inverse de flot de v vers u . Combiné avec la troisième contrainte (P3), on obtient la loi de Kirchhoff exprimant la conservation du flot en chaque sommet différent de la source et du puits : pour tout sommet $u \in V \setminus \{s, t\}$, la somme des flots positifs entrants est égales à la somme des flots positifs sortants. En effet, soit $u \in V \setminus \{s, t\}$. On a

$$\sum_v f(u, v) = \sum_v f^+(u, v) + \sum_v f^-(u, v),$$

où $f^+(x) = \max\{0, f(x)\}$ et $f^-(x) = \min\{0, f(x)\}$. De cette égalité combinée à P3, on obtient

$$\sum_v f^+(u, v) = -\sum_v f^-(u, v),$$

d'où il découle par P2 que $\sum_v f^+(u, v) = \sum_v f^+(v, u)$, tel qu'énoncé par la loi de Kirchhoff.

La *valeur d'un flot* f est définie comme : $|f| = \sum_{v \in V} f(s, v)$. Par la conservation du flot induit par la loi de Kirchhoff, on obtient que $|f| = \sum_{v \in V} f(v, t)$. En effet, d'une part P2 implique que

$$\sum_u \sum_v f(u, v) = 0,$$

et, d'autre part, P3 implique que

$$\sum_u \sum_v f(u, v) = \sum_v f(s, v) - \sum_v f(v, t).$$

Le problème FLOT MAXIMUM :

Entrée : un réseau de transport (G, c, s, t) ;

Objectif : trouver un flot f dans G de valeur maximum.

3.5.2 Algorithme de Ford-Fulkerson

Cet algorithme date de la fin des années 50. Etant donné un flot f , on définit la *capacité résiduelle*

$$c_f(u, v) = c(u, v) - f(u, v).$$

Le *réseau résiduel* G_f est le réseau G muni des capacités résiduelles. Attention, G_f n'est pas nécessairement égal à G car, comme nous allons le voir, le calcul des capacités résiduelles peut faire apparaître des arcs « retour » de capacités résiduelles positives. Par ailleurs, lorsque $c_f(u, v) = 0$, l'arc (u, v) disparaît de G_f .

La preuve du lemme suivant est une conséquence directe des définitions, et est laissée en exercice.

Lemma 2 Si f' est un flot dans le réseau résiduel G_f induit par le flot f dans G , alors $f + f'$ est un flot dans G de valeur $|f + f'| = |f| + |f'|$.

L'algorithme de Ford-Fulkerson ci-dessous procède de manière gloutonne :

Algorithme de Ford-Fulkerson (F&F) :

```

pour toute paire  $(u, v) \in V \times V$  faire  $f(u, v) \leftarrow 0$  ;
tant que  $\exists$  chemin  $P$  de  $s$  à  $t$  dans  $G_f$  tel que  $c_f(e) > 0$  pour tout  $e \in E(P)$  faire
    soit  $c_f(P) = \min_{(u,v) \in E(P)} c_f(u, v)$  ;
    pour chaque arc  $(u, v) \in E(P)$  faire
         $f(u, v) \leftarrow f(u, v) + c_f(P)$ 
         $f(v, u) \leftarrow -f(u, v)$ 
         $c(u, v) \leftarrow c(u, v) - c_f(P)$ 
         $c(v, u) \leftarrow c(v, u) + c_f(P)$ 

```

La figure 12 présente un exemple d'exécution de l'algorithme. Sur cette figure, l'algorithme s'exécute en trois itérations. A chaque itération, le chemin choisi est indiqué en gras et en couleur. La paire x/y indiquée sur chaque arc indique que le flot passant par cette arc a une valeur x , et que sa capacité résiduelle est y . Au début, le flot est nul sur tous les arcs, et la capacités résiduelles sont égales aux capacités. Le flot croît à chaque étape le long du chemin P . Chaque étape sature au moins un arc, indiqué en très gras sur la figure. Les arcs en pointillé sont ceux saturés à une étape précédente. Après trois étapes, il n'y a plus de chemin de s à t dans G_f . Le flot est de valeur $|f| = 5$ (+2 à l'étape 1, +1 à l'étape 2, +2 à l'étape 3).

Dans l'exemple de la figure 12 ne présente pas les arcs retour, créés lors de la mise à jour des capacités résiduels le long des chemins P sélectionnés par l'algorithme. Cela n'a pas posé de problème dans cet exemple, mais peut poser des problèmes lors d'autres exécutions de l'algorithme, même sur le même réseau de transport. La figure 13 présente le même réseau de transport, mais une exécution différente de l'algorithme.

Complexité. A chaque itération, le flot f voit sa valeur augmentée d'au moins une unité. Le nombre d'itérations de l'algorithme est donc au plus $|f_{max}|$ où f_{max} est un flot maximum.

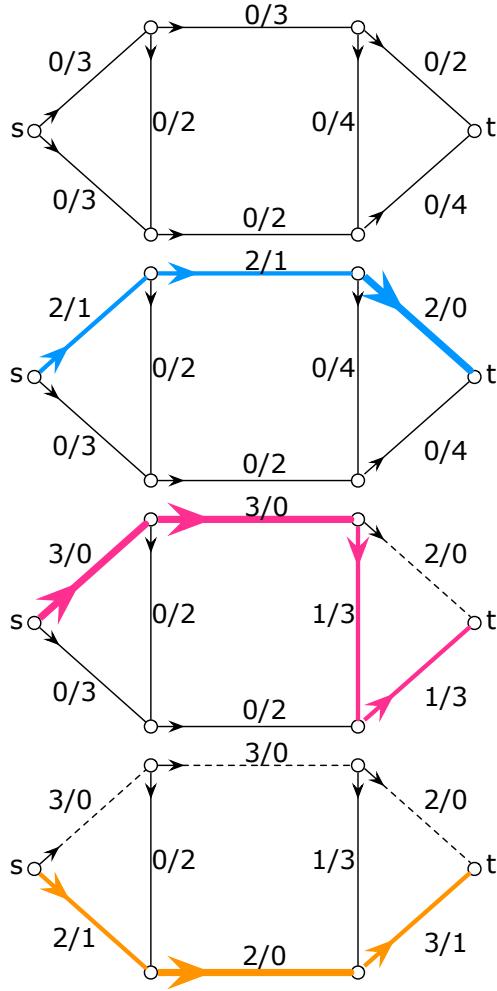


FIGURE 12 – Exemple d'exécution de l'algorithme de Ford et Fulkerson

La complexité de chaque itération est dominée par la recherche du chemin P dans G_f . Cette recherche peut se faire par un BFS ou un DFS, donc en temps $O(|E_f|)$. Or $|E_f| \leq 2|E|$ car les seuls arcs créés dans G_f sont des arcs retour d'arcs existants dans G . La complexité de l'algorithme est donc $O(|E| \cdot |f_{max}|)$. Il existe toutefois des mises en oeuvre de l'algorithme de Ford-Fulkerson de complexités polynomiales indépendantes de $|f_{max}|$. Par exemple, l'algorithme d'Edmonds-Karp (1972) s'exécute en temps $O(nm^2)$ dans un réseau de transport de n noeuds et m arcs. Le meilleur algorithme connu s'exécute en temps $O(nm)$.

Montrons maintenant que le flot obtenu par l'algorithme de Ford et Fulkerson est optimal. Pour cela, nous allons montrer que lorsqu'il n'y a plus de chemin « augmentant » P , le flot est maximum. Pour cela, définissons une (s, t) -coupe comme une coupe (S, T) avec $s \in S$ et $t \in T = V \setminus S$. Soit $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$. L'optimalité de Ford et Fulkerson découle du résultat suivant :

Théorème 4 (L. Ford et D. Fulkerson) *Le flot obtenu par l'algorithme de Ford et Fulkerson est au moins égal à la valeur minimum d'une (s, t) -coupe dans G . La valeur maximum d'un flot de s à t dans G ne peut excéder valeur minimum d'une (s, t) -coupe dans G .*

Preuve. Soit f le flot lorsque l'algorithme F&F termine, c'est-à-dire lorsqu'il n'y a plus de

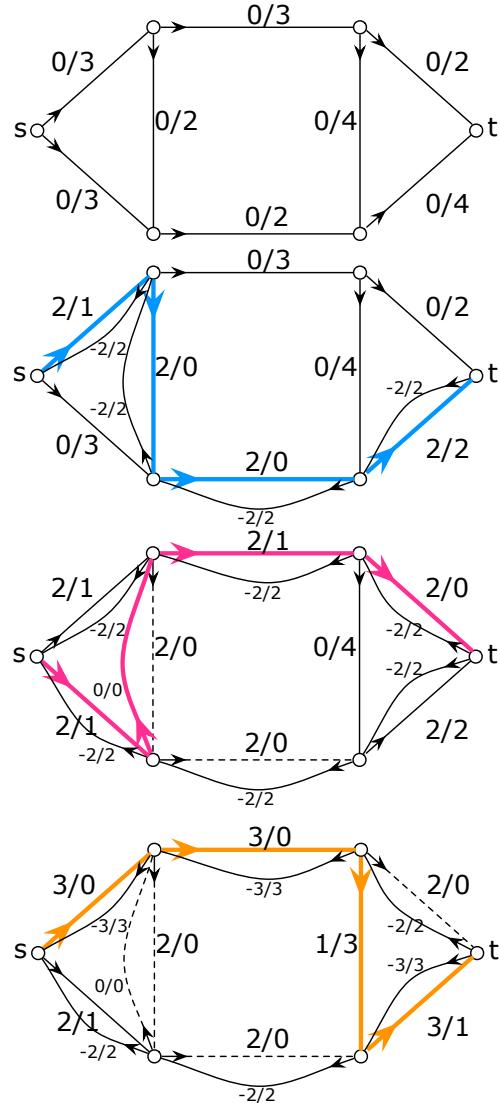


FIGURE 13 – Second exemple d'exécution de l'algorithme de Ford et Fulkerson, avec arcs retours

chemin augmentant de s à t . On considère alors

$$S = \{v \in V \mid \text{il existe un chemin de } s \text{ à } v \text{ dans } G_f\}$$

et $T = V \setminus S$. Si $u \in S$ et $v \in T$, alors $f(u, v) = c(u, v)$ car $c_f(u, v) = 0$. Donc, en définissant $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v)$, on obtient $f(S, T) = c(S, T)$. Or, par P2,

$$\sum_{u \in S} \sum_{v \in T} f(u, v) = 0,$$

et donc

$$f(S, T) = \sum_{u \in S} \sum_{v \in V} f(u, v) = \sum_{v \in V} f(s, v) = |f|.$$

On obtient alors $|f| = c(S, T)$, et donc $|f| \geq c_{min}$ où c_{min} dénote la valeur d'une (s, t) -coupe minimum.

Réiproquement, soit $|f^*|$ la valeur d'un flot optimal. Soit (S, T) une (s, t) -coupe telle que $c(S, T) = c_{min}$. Pour chaque arc (u, v) de la coupe avec $u \in S$ et $v \in T$, on a $f^*(u, v) \leq c(u, v)$ et donc

$$|f^*| = \sum_{v \in V} f^*(s, v) = \sum_{u \in S} \sum_{v \in V} f^*(u, v) = \sum_{u \in S} \sum_{v \in T} f^*(u, v) \leq \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T) = c_{min}.$$

Il découle de ce qui précède que $c_{min} \leq |f| \leq |f^*| \leq c_{min}$, et donc $|f| = |f^*|$, c'est-à-dire f est optimal. \square

3.5.3 Application au couplage

Un *couplage* dans un graphe $G = (V, E)$ est un ensemble d'arêtes $M \subseteq E$ tel que deux arêtes distinctes de M ne partagent aucun sommet. Un couplage M est *maximal* s'il n'existe pas d'arête $e \in E$ telle que $M \cup \{e\}$ soit un couplage. Un couplage M est *maximum* s'il a la plus grande cardinalité parmi tous les couplages de G (ou le plus grand poids dans le cas où les arêtes de G auraient un poids). Un couplage est *parfait* si tout sommet est couplé, c'est-à-dire tout sommet est adjacent à une arête du couplage. Trouver un couplage maximal peut se faire par un algorithme linéaire glouton :

Fonction couplage-maximal(graphe $G = (V, E)$)

```

 $C \leftarrow \emptyset$ 
 $F \leftarrow E$ 
tant que  $F \neq \emptyset$  faire
    choisir  $e \in F$  quelconque
     $C \leftarrow C \cup \{e\}$ 
    supprimer de  $F$  l'arête  $e$  et toutes ses arêtes incidentes dans  $G$ 
    retourner  $C$ 

```

La recherche d'un couplage maximum requiert plus d'efforts. Nous allons ici appliquer une technique de flot pour calculer un couplage maximum dans un graphe biparti.

Definition 7 Un graphe $G = (V, E)$ est biparti s'il existe une partition (V_1, V_2) des sommets ($V_1 \cup V_2 = V$ et $V_1 \cap V_2 = \emptyset$) telles que chacune des arêtes de E relie un sommet de V_1 à un sommet de V_2 .

Par exemple : les hypercubes et les grilles sont des graphes bipartis. Un cycle est biparti seulement s'il contient un nombre pair de sommets. La recherche d'un couplage dans un graphe biparti correspond à de multiple problèmes se modélisant par la satisfaction d'un maximum de « relations », comme par exemple entre des ressources et des consommateurs.

Exercice 8 Montrer comment il est possible d'utiliser une algorithme de flot maximum pour calculer un couplage maximum dans un graphe biparti.

Attention, cette approche ne se généralise pas à un calcul de couplage de poids maximum. Le problème du couplage de poids maximum peut toutefois se résoudre dans les graphes quelconques par l'algorithme des *chemins augmentés* dû à Edmonds (1965). Cet algorithme peut être raffiné pour s'exécuter en temps $O(m\sqrt{n})$ dans les graphes de n sommets et m arêtes. Notez que tous les problèmes de couplage ne sont pas nécessairement polynomiaux. Par exemple, la recherche d'un couplage maximal de cardinalité minimum est NP-complet.

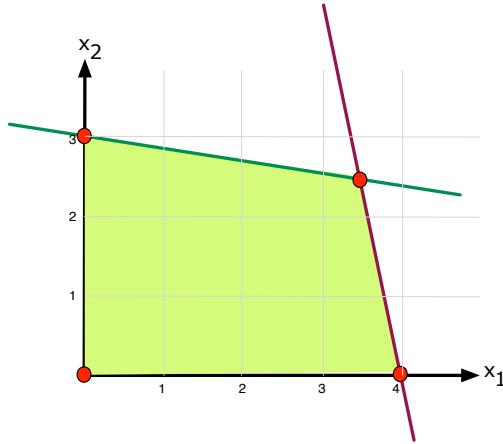


FIGURE 14 – En dimension 2, les contraintes définissent un polygone convexe.

3.6 Programmation linéaire

3.6.1 Définition

Considérons une entreprise qui dispose de m matières premières en quantité respectives b_i , $i = 1, \dots, m$, à partir desquelles elle peut produire n produits. Le produit $j \in \{1, \dots, n\}$ nécessite une quantité $a_{i,j}$ de la matière première i . L'entreprise peut vendre le produit j au prix c_j . L'entreprise souhaite maximiser ses profits $\sum_{j=1}^n c_j x_j$ où x_j dénote la quantité produite du produit j . Ses contraintes sont liées à la quantité dont l'entreprise dispose de chaque matière première, s'exprimant sous la forme $\sum_{j=1}^n a_{i,j} x_j \leq b_i$ pour toute matière première $i = 1, \dots, m$. La programmation linéaire fait référence au problème de la minimisation ou maximisation d'une fonction objective linéaire, sous des contraintes linéaires. Dans le cadre de l'exemple ci-dessus, la programmation linéaire fait référence à la recherche de n réels (positifs) x_1, \dots, x_n qui

$$\begin{aligned} & \text{maximisent} && \sum_{j=1}^n c_j x_j \\ & \text{sous la contrainte} && \sum_{j=1}^n a_{i,j} x_j \leq b_i \text{ for } i = 1, \dots, m \end{aligned}$$

où les c_j , b_i et $a_{i,j}$ sont rationnels, pour $i = 1, \dots, m$ et $j = 1, \dots, n$. Plus généralement, sous sa forme standard, la programmation linéaire consiste à résoudre des problèmes d'optimisation du type :

$$\max_{x \in \mathbb{R}^n | Ax \leq b} c^T x \quad \text{ou} \quad \min_{x \in \mathbb{R}^n | Ax \geq b} c^T x$$

où $A \in \mathbb{Q}^{n \times m}$, $b \in \mathbb{Q}^m$, and $c \in \mathbb{Q}^n$.

En dimension 2 (i.e., lorsque $n = 2$), le domaine défini par les m contraintes forme un polygone convexe. Voir par exemple la figure 14 qui indique le domaine défini par deux contraintes linéaires. La solution optimale ne peut se situer qu'en un des sommets marqués en rouge de ce polygone. En effet, étant donné un point p sur une arête du polygone, il existe au moins une direction de déplacement le long de cette arête qui permet d'augmenter la fonction objective. Plus généralement, en dimension n , le domaine est un polytope⁶ convexe, et la solution optimale se situe également en un sommet de ce polytope.

6. Ou polyèdre (la terminologie diffère entre pays).

Il existe deux classes d’algorithmes polynomiaux en la taille $\Theta(nm)$ du problème et en $1/\epsilon$ où ϵ désigne la précision souhaitée sur les valeurs réelles x_j , $j = 1, \dots, n$: la méthode de l’ellipsoïde (Khachian, 1979) et la méthode du point intérieur (Karmarkar, 1984). En pratique, la méthode la plus souvent utilisée est celle du Simplex due à Dantzig (1947). Cette méthode explore les sommets du polytope des contraintes en se déplaçant de sommet en sommet selon une stratégie gloutonne en privilégiant le meilleur gain. Le Simplex est exponentiel dans le pire cas car le nombre de sommets du polytope peut être exponentiel. Le Simplex est néanmoins polynomial en moyenne⁷ et se montre rapide sur les instances de problèmes rencontrées en pratique.

La programmation linéaire est une vaste théorie qui a de fait conduit au développement de nombreux logiciels du commerce basés sur des combinaisons d’algorithmes évoqués ci-dessus avec des heuristiques permettant de raffiner ou d’accélérer la recherche de la solution. Le cours n’a fait ici qu’effleurer le sujet (sans même évoquer la dualité).

3.6.2 Programmation linéaire en nombres entiers

Le problème du sac à dos peut se réécrire sous la forme suivante :

$$\begin{aligned} & \text{maximiser} && \sum_{i=1}^n p_i x_i \\ & \text{sous la contrainte} && \sum_{i=1}^n w_i x_i \leq W \\ & && x_i \in \{0, 1\} \text{ pour } i = 1, \dots, n \end{aligned}$$

Les variables x_j ne sont plus réelles, mais entières. On parle alors de *programmation entière*, ou ILP pour *integer linear programming* :

Le problème ILP :

Entrée : des rationnels $a_{i,j}, b_i, c_j$, $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$;

Objectif : maximiser $\sum_{j=1}^n c_j x_j$ sous les contraintes $\sum_{j=1}^n a_{i,j} x_j \leq b_i$ pour $i = 1, \dots, m$ et $x_j \in \mathbb{Z}$ pour $j = 1, \dots, n$.

La programmation linéaire en nombres entiers est NP-difficile, c’est-à-dire ILP \in NPC. Exprimer un problème sous la forme d’un programme linéaire en nombres entiers trouve toutefois de nombreuses applications, en particulier pour concevoir des algorithmes d’approximation et des algorithmes probabilistes. Nous revenons plus loin dans le cours.

Il existe néanmoins des cas où le programme linéaire relaxé, c’est-à-dire lorsque l’on relaxe la contrainte d’intégralité des variables, a une solution optimale entière. C’est en particulier le cas lorsque les contraintes $Ax \leq b$ satisfont b entier et *A totalement unimodulaire*. (Une matrice A est totalement unimodulaire si le déterminant de chacune de ses sous-matrices carrées est égal à 0, 1, ou -1).

4 Algorithmes d’approximation

Lorsqu’un problème est dans NPC, cela indique qu’il sera difficile (voire impossible s’il s’avère que $P \neq NP$) de trouver un algorithme polynomial pour le résoudre. Cela ne veut toutefois pas

7. Surtout, sa complexité *lissée* (« smoothed complexity » en anglais) est polynomiale.

dire qu'il n'est pas possible de trouver une solution approchée. Plus précisément, même si la version décisionnelle d'un problème de minimisation ou de maximisation est NP-difficile, il est souvent possible de décider une version relaxé de ce problème. Par exemple, nous avons vu qu'il est NP-difficile de décider si, pour le problème COUVERTURE ENSEMBLE, U peut être couvert avec au plus k sous-ensembles de \mathcal{S} . En revanche, il peut être possible de décider en temps polynomial le problème relaxé dont les spécifications ci-dessous dépendent d'un paramètre $\rho > 1$:

- si U ne peut pas être couvert avec au plus k sous-ensembles de \mathcal{S} alors retourner "non" ;
- si U peut être couvert avec au plus $\rho \cdot k$ sous-ensembles de \mathcal{S} alors retourner "oui".

Cette spécification ne stipule pas la réponse à donner pour un nombre de sous-ensembles entre k et $\rho \cdot k$. Il s'agira donc soit de chercher un algorithme polynomial pour résoudre le problème relaxé avec ρ le plus proche de 1 possible, soit de montrer que tout algorithme polynomial ne peut résoudre le problème relaxé que pour des valeurs de ρ en deçà d'une certaine borne. Dans ce dernier cas, cela ne pourra souvent être montré que sous des hypothèses plausibles liées à la théorie de la complexité, typiquement sous l'hypothèse que $P \neq NP$.

4.1 Définition

Un problème d'optimisation est défini par une paire $\Pi = (\mathcal{I}, f)$ où \mathcal{I} désigne l'ensemble des instances et f est une fonction à valeurs rationnelles mesurant la qualité des solutions. Pour $I \in \mathcal{I}$, on note $S(I)$ l'ensemble des solutions admissibles pour I . L'objectif est alors de trouver, pour tout $I \in \mathcal{I}$, une solution $x^* \in S(I)$, tel que

- $f(x^*) = \min_{x \in S(I)} f(x)$ s'il s'agit d'un problème de minimisation, ou
- $f(x^*) = \max_{x \in S(I)} f(x)$ s'il s'agit d'un problème de maximisation.

On note $\text{OPT}(I)$ la valeur de $f(x^*)$.

Definition 8 Soit $\rho \geq 1$. Un algorithme A est un algorithme de ρ -approximation pour un problème de minimisation $\Pi = (\mathcal{I}, f)$ si, pour tout $I \in \mathcal{I}$,

$$A(I) \in S(I) \text{ et } f(A(I)) \leq \rho \cdot \text{OPT}(I).$$

Dans le cas d'un problème de maximisation, la dernière inégalité est remplacée par

$$f(A(I)) \geq \frac{1}{\rho} \cdot \text{OPT}(I).$$

4.1.1 2-approximation de la couverture sommet minimum

Definition 9 Soit $G = (V, E)$ un graphe. Une couverture-sommet de G est un ensemble $C \subseteq V$ tel que pour toute arête $e \in E$, au moins une des deux extrémités de e soit dans C .

Le problème COUVERTURE-SOMMET :

Entrée : un graphe G ;

Objectif : trouver une plus petite couverture-sommet de G .

Dans ce problème, l'ensemble \mathcal{I} des instances est l'ensemble des graphes. Etant donné un graphe G , une solution admissible est une couverture sommet. La fonction f retourne la taille

de la couverture. Il s'agit donc de minimiser f sur l'ensemble des solutions admissibles. Notons que tout graphe possède au moins une solution admissible, en prenant la couverture égale à l'ensemble de tous les sommets. Le problème COUVERTURE-SOMMET est NP-complet. Il est toutefois facile d'approximer une solution optimale à une constante près en temps polynomial.

Algorithme d'approximation pour COUVERTURE-SOMMET :

début

trouver un couplage maximal M dans G ;
retourner l'ensemble C des sommets couplés dans M ;

fin.

Cet algorithme est polynomial puisque trouver un couplage maximal dans un graphe est polynomial par une algorithme glouton. (Trouver un couplage maximal de cardinalité minimum est en revanche NP-difficile.) Montrons que c'est un algorithme de 2-approximation pour COUVERTURE-SOMMET. Tout d'abord, toutes les arêtes sont couvertes par les sommets de C . En effet, s'il existait une arête $e = \{u, v\}$ non couverte par C , alors ni u ni v sont couplés, et on pourrait alors rajouter e à M , contredisant la maximalité de M .

Nous allons maintenant montrer que $|C| \leq 2 \text{OPT}$ où OPT est la taille d'une couverture-sommet minimum dans G . Comme on ne connaît pas OPT, montrer une telle inégalité requiert en fait deux choses :

1. trouver une borne inférieure pour OPT, du genre $\text{OPT} \geq \alpha$, et
2. trouver une borne supérieure pour $|C|$, du genre que $|C| \leq c\alpha$.

Si l'on peut faire cela, alors on aura montré que $|C| \leq c \text{OPT}$, est que donc l'algorithme est c -approximant.

On obtient une borne inférieure adéquate en observant que toute couverture-sommet doit contenir au moins un sommet de chacune des arêtes de n'importe quel couplage M . Donc $\text{OPT} \geq |M|$. Pour la borne supérieure, il suffit de noter que $|C| = 2|M|$. En conséquence, $|C| \leq 2 \text{OPT}$, et donc l'algorithme est 2-approximant.

4.1.2 La classe APX

Definition 10 *On note APX la classe des problèmes (d'optimisation) Π pour lesquels il existe une constante c et un algorithme polynomial A de c -approximation pour Π .*

Par exemple, COUVERTURE-SOMMET \in APX. Malheureusement, on sait qu'il y a des problèmes pour lesquels il est peu probable qu'il soit dans APX. (Peu probable, au sens ou si c'était le cas, alors on aurait P = NP, ce qui est considéré comme peu vraisemblable.) En particulier :

Lemma 3 (Arora et Sudan, et Raz et Safra) *Sauf si P = NP, la solution optimale COUVERTURE ENSEMBLE ne peut pas être approximée à moins d'un facteur $\Omega(\log n)$ où n est la cardinalité de l'ensemble donné en entrée.*

Une conséquence de ce lemme est que, sauf si P = NP, COUVERTURE ENSEMBLE \notin APX.

4.2 Approximation « gloutonne »

4.2.1 Couverture-sensemble

Nous avons vu que COUVERTURE ENSEMBLE est NP-complet. Nous allons voir qu'il permet toutefois d'approximer COUVERTURE ENSEMBLE à un facteur logarithmique près, en temps polynomial. D'après le Lemme 3, il n'est pas possible de faire mieux, sauf si P = NP.

Algorithme glouton pour COUVERTURE-ENSEMABLE :

début

```

 $C \leftarrow \emptyset$ 
 $V \leftarrow \emptyset$ 
tant que  $V \neq U$  faire
    choisir  $S \in \mathcal{S}$  maximisant  $|S \setminus V|$  ;
     $C \leftarrow C \cup \{S\}$  ;
     $V \leftarrow V \cup S$  ;
    Retourner  $C$  ;

```

fin.

Afin d'analyser cet algorithme, il convient d'observer que si S est l'ensemble ajouté à C à l'étape t , et si S' est l'ensemble ajouté à C à l'étape $t' > t$, alors

$$|S \setminus V_t| \geq |S' \setminus V_{t'}|,$$

où V_t et $V_{t'}$ dénotent respectivement l'ensemble V au début des itérations t et t' . Dis autrement, le nombre de nouveaux éléments couverts décroît (non nécessairement strictement) à chaque étape. Pour montrer que l'algorithme retourne une approximation à un facteur logarithmique près de la solution optimale, on montre la propriété $P(k)$ suivante, en notant $n = |U|$:

$P(k)$: après $k \cdot \text{OPT}$ itérations, la collection construite couvre au moins $(1 - \frac{1}{2^k})n$ éléments de U .

Ce fait est établi par récurrence sur k . $P(k)$ est trivialement vrai pour $k = 0$. Soit $k \geq 1$. Supposons $P(k-1)$, c'est-à-dire, après $(k-1) \cdot \text{OPT}$ itérations, la collection construite couvre au moins $(1 - \frac{1}{2^{k-1}})n$ éléments de U . Supposons, dans un objectif de contradiction, que $P(k)$ soit faux, c'est-à-dire, après $k \cdot \text{OPT}$ itérations, la collection C construite couvre strictement moins de $(1 - \frac{1}{2^k})n$ éléments de U . Dit autrement, le nombre d'éléments de U non couverts par C est supérieur à $\frac{n}{2^k}$.

- Soit C^* une collection optimale, i.e., $|C^*| = \text{OPT}$. Les OPT ensembles de C^* couvrent U , dont en particulier les au moins $\frac{n}{2^k}$ éléments non couverts par C . Il existe donc un ensemble $S^* \in C^*$ parmi les OPT ensembles de C^* couvrant au moins $\frac{n}{2^k \cdot \text{OPT}}$ éléments de U non couverts par C .
- Par ailleurs, puisqu'après $(k-1) \cdot \text{OPT}$ itérations, la collection construite couvrait au moins $(1 - \frac{1}{2^{k-1}})n$ éléments de U , il découle que les OPT ensembles ajoutés entre l'itération $(k-1) \cdot \text{OPT}$ et l'itération $k \cdot \text{OPT}$ couvrent strictement moins de $\frac{n}{2^k}$ nouveaux éléments. En conséquence, l'ensemble S choisi par l'algorithme durant la $(k \cdot \text{OPT})$ ème itération, qui est celui qui couvre le moins de nouveaux éléments parmi les OPT derniers ensembles ajoutés, ne couvre que strictement moins de $\frac{n}{2^k \cdot \text{OPT}}$ nouveaux éléments parmi ceux non couverts avant cette itération.

L'existence de S^* couvrant au moins $\frac{n}{2^k \cdot \text{OPT}}$ éléments de U non couverts par C et en contradiction avec le choix de S , car l'algorithme glouton aurait dû choisir S^* au lieu de S . Donc, $P(k)$ est vrai.

Puisqu'après $k \cdot \text{OPT}$ itérations, la collection construite couvre au moins $(1 - \frac{1}{2^k})n$ éléments de U , il découle que l'algorithme effectue au plus $\lceil \log_2 n \rceil \cdot \text{OPT} + 1$ itérations, et donc retourne une solution de taille au plus $\lceil \log_2 n \rceil \cdot \text{OPT} + 1$. Il s'agit donc d'un algorithme de $(\lceil \log_2 n \rceil + 1)$ -approximation.

4.2.2 Couverture-ensemble pondérée

Nous allons maintenant montrer qu'une stratégie gloutonne permet également d'approximer une version pondérée de COUVERTURE ENSEMBLE.

Le problème COUVERTURE ENSEMBLE PONDÉRÉ :

Entrée : un ensemble fini U , une collection $\mathcal{S} = \{S_1, \dots, S_k\}$ de sous-ensembles de \mathcal{S} , et une fonction de coût $c : \mathcal{S} \rightarrow \mathbb{Q}^+$;
Objectif : trouver un sous-ensemble de \mathcal{S} dont l'union contient tous les éléments de U , et de coût total minimum.

L'algorithme glouton pour approximer la solution à ce problème est le suivant :

Algorithme glouton pour COUVERTURE-ENSEMBLE PONDÉRÉ :
début

```

 $C \leftarrow \emptyset$ 
 $V \leftarrow \emptyset$ 
tant que  $V \neq U$  faire
    choisir  $S \in \mathcal{S}$  de plus petit « coût relatif »  $c(S)/|S \setminus V|$  ;
     $C \leftarrow C \cup \{S\}$  ;
     $V \leftarrow V \cup S$  ;
    Retourner  $C$  ;

```

fin.

Nous allons montrer que cet algorithme retourne une approximation à un facteur logarithmique près de la solution optimale. Pour cela, on rajoute un paramètre de « prix » aux éléments couverts au fur et à mesure de l'algorithme, correspondant au coût relatif de chaque élément au moment où il est choisi, comme suit :

```

/* boucle modifiée */
tant que  $V \neq U$  faire
    choisir  $S \in \mathcal{S}$  de plus petit coût relatif  $c(S)/|S \setminus V|$  ;
    pour chaque  $e \in S \setminus V$  faire  $\text{prix}(e) \leftarrow c(S)/|S \setminus V|$  ;
     $C \leftarrow C \cup \{S\}$  ;
     $V \leftarrow V \cup S$  ;

```

Soit e_1, e_2, \dots, e_n les éléments de U énumérés dans l'ordre dans lequel ils sont choisis par l'algorithme. Le coût total de la solution de l'algorithme est $\sum_{k=1}^n \text{prix}(e_k)$. Nous allons donc borner cette somme en fonction de OPT . Soit C^* une couverture optimale, c'est-à-dire

$$\sum_{S \in C^*} c(S) = \text{OPT}.$$

A chaque itération, les ensembles de C^* qui n'ont pas été choisis couvrent $U \setminus V$. Donc, à chaque itération, il existe un de ces ensembles, $S^* \in C^* \setminus C$, tel que

$$c(S^*)/|S^* \setminus V| \leq \text{OPT}/|U \setminus V|.$$

En effet, supposons, par contradiction, que, pour tous les ensembles $S \in C^* \setminus C$, on ait

$$c(S)/|S \setminus V| > \text{OPT}/|U \setminus V|.$$

Alors, en sommant sur ces ensembles, on obtient

$$\sum_{S \in C^* \setminus C} c(S) > \text{OPT}/|U \setminus V| \cdot \sum_{S \in C^* \setminus C} |S \setminus V|.$$

Or $C^* \setminus C$ couvre $U \setminus V$, donc $\cup_{S \in C^* \setminus C} (S \setminus V) = U \setminus V$. Cela implique que $\sum_{S \in C^* \setminus C} |S \setminus V| \geq |U \setminus V|$. Il en découle que $\sum_{S \in C^* \setminus C} c(S) > \text{OPT}$, contradiction.

Soit donc S choisi à l'itération courante. Par le choix de S minimisant $c(S)/|S \setminus V|$, on a

$$c(S)/|S \setminus V| \leq c(S^*)/|S^* \setminus V| \leq \text{OPT}/|U \setminus V|.$$

Soit $e_k \in S$ un élément ajouté à cet itération. On obtient

$$\text{prix}(e_k) = c(S)/|S \setminus V| \leq \text{OPT}/|U \setminus V| \leq \text{OPT}/(n - k + 1).$$

Donc

$$\sum_{k=1}^n \text{prix}(e_k) \leq \left(1 + \frac{1}{2} + \dots + \frac{1}{n}\right) \text{OPT} = H_n \cdot \text{OPT}$$

où H_n est le n ème nombre harmonique. On a $H_n \sim \ln n$, d'où une approximation à $\ln n$ près.

Exercice 9 Donner une instance de COUVERTURE ENSEMBLE montrant que l'analyse du glou-ton ci-dessus est exacte.

4.3 Approximation à partir d'arbres couvrant minimaux

4.3.1 Problème du voyageur de commerce

Le problème du « voyageur de commerce » est motivé par l'optimisation de la tâche visant à visiter un ensemble de sites en parcourant une distance minimum.

Problème VOYAGEUR DE COMMERCE :

Entrée : un graphe complet de n sommets, dont chaque arête e est munie d'un poids $w(e) > 0$;

Objectif : construire un cycle visitant chaque sommet une et une seule fois, et dont la somme des poids des arêtes est minimum.

Le problème de décision associé au problème VOYAGEUR DE COMMERCE est NP-complet. Nous allons voir que ce problème est en fait très difficile au sens qu'il ne peut pas être approximé. Nous étudierons donc ensuite une version restreinte du problème dans le cas où les poids satisfont l'inégalité triangulaire.

4.3.2 Inapproximabilité

Afin de montrer que le problème VOYAGEUR DE COMMERCE ne peut pas être approximé, nous allons montrer que si l'on pouvait l'approximer, il serait alors possible de résoudre un problème NP-complet en temps polynomial. Ce problème est celui celui du cycle Hamiltonien.

Problème CYCLE HAMILTONIEN :

Entrée : un graphe G ;

Question : existe-t-il un cycle respectant les arêtes de G visitant chaque sommet de G une et une seule fois ? (Un tel cycle est dit *Hamiltonien*, et un graphe possédant un cycle Hamiltonien est lui-même dit « Hamiltonien ».)

Par exemple, l'hypercube Q_d est Hamiltonien. En revanche la grille 3×3 n'est pas Hamiltonienne. Le problème CYCLE HAMILTONIEN est NP-complet.

Théorème 5 *Sauf si $P = NP$, pour toute fonction f calculable en temps polynomial, il n'existe pas d'algorithme polynomial $f(n)$ -approximant pour VOYAGEUR DE COMMERCE.*

Preuve. Soit f une fonction calculable en temps polynomial, et supposons qu'il existe un algorithme polynomial A , $f(n)$ -approximant pour VOYAGEUR DE COMMERCE. Soit G un graphe de n sommets. Etiquetons les sommets de G par des entiers deux à deux distinct dans $[1, n]$. On construit le graphe complet K_n de n sommets dont les sommets sont également étiquetés par des entiers deux à deux distinct dans $[1, n]$, et dont les arêtes ont deux types de poids possibles. Si $e = \{u, v\} \in E(G)$ alors $w(e) = 1$, sinon $w(e) = n \cdot f(n)$. Notons que puisque f est calculable en temps polynomial, cette réduction est polynomiale.

Remarquons une propriété essentielle de cette réduction. Si G est Hamiltonien alors l'instance $I = (K_n, w)$ du VOYAGEUR DE COMMERCE a une solution de poids $\leq n$ (une telle solution est fournie par un cycle Hamiltonien de G), donc $OPT \leq n$. Réciproquement, si G n'est pas Hamiltonien alors au moins une non-arête de G doit être utilisée pour construire un cycle du VOYAGEUR DE COMMERCE, et donc l'instance $I = (K_n, w)$ du VOYAGEUR DE COMMERCE n'a pas de solution de poids $< n \cdot f(n)$, et donc $OPT \geq n \cdot f(n)$.

L'inapproximabilité résulte directement de la remarque ci-dessus. Une fois la réduction effectuée, on applique A . Si A retourne une solution $A(I)$ de poids p , alors $p \leq f(n) \cdot OPT$, et donc $OPT \geq p/f(n)$. Ainsi, si $p > n \cdot f(n)$, alors $OPT > n$, et donc G n'est pas Hamiltonien. En revanche, si $p \leq n \cdot f(n)$, alors comme soit $OPT \leq n$, soit $OPT > n \cdot f(n)$, on a nécessairement $OPT \leq n$ et donc G est Hamiltonien. On conclut donc que la composition de A et de la réduction polynomiale permet de résoudre le problème NP-complet CYCLE HAMILTONIEN, ce qui est impossible si $P \neq NP$. \square

4.3.3 Le voyageur de commerce métrique

Au vu du Théorème 5, il convient de restreindre la généralité de VOYAGEUR DE COMMERCE. Une restriction naturelle consiste à ne considérer que des instances $I = (K_n, w)$ correspondant à des métriques, c'est-à-dire pour lesquelles les poids satisfont l'inégalité triangulaire $w(\{u, v\}) \leq w(\{u, w\}) + w(\{w, v\})$ pour tout triplet de sommets u, v, w . On peut montrer que

VOYAGEUR DE COMMERCE MÉTRIQUE reste NP-complet dans ce contexte. Notons en revanche que la construction dans la preuve du Théorème 5 ne respecte pas nécessairement l'inégalité triangulaire. De fait, VOYAGEUR DE COMMERCE MÉTRIQUE est dans APX, comme montre le théorème suivant :

Théorème 6 *Il existe un algorithme polynomial 2-approximant pour VOYAGEUR DE COMMERCE MÉTRIQUE.*

Preuve. Nous allons donner explicitement l'algorithme. Soit $I = (K_n, w)$ une instance de VOYAGEUR DE COMMERCE MÉTRIQUE. L'algorithme procède d'abord en construisant un arbre couvrant de poids minimum de (K_n, w) (par exemple en utilisant Borůvka, Kruskal, ou Prim). Soit T cet arbre. Nous allons montrer comment construire à partir de T une solution de VOYAGEUR DE COMMERCE MÉTRIQUE approximant OPT à un facteur 2.

Notons d'abord que T est le sous-graphe connexe le plus léger couvrant tous les sommets de K_n , donc $\text{OPT} \geq w(T)$. On effectue un parcours en profondeur d'abord (DFS) dans T . Notons C le parcours obtenu, et remarquons que C traverse chaque arête exactement deux fois. Donc $w(C) \leq 2w(T)$. On transforme C en un tour dans K_n visitant une et une seule fois chaque sommet, comme suit. Soit $C = (x_0, x_1, \dots, x_{2n-1})$, où un même sommet peut avoir plusieurs occurrences dans cette suite. On construit $C' = (y_0, y_1, \dots, y_{n-1})$ à partir de C en supprimant toutes les occurrences d'un même sommet, à partir de la deuxième occurrence. Par construction C' est une solution à VOYAGEUR DE COMMERCE MÉTRIQUE (chaque sommet apparaît une et une seule fois dans C'). De plus, l'inégalité triangulaire assure que lorsque une portion $x_i, x_{i+1}, \dots, x_{j-1}, x_j$ de C est remplacée par un « raccourci » x_i, x_j , on a $w(\{x_i, x_j\}) \leq \sum_{k=i}^{j-1} w(\{x_k, x_{k+1}\})$. Donc $w(C') \leq w(C)$. On obtient donc la chaîne d'inégalités $w(C') \leq w(C) \leq 2w(T) \leq 2\text{OPT}$. \square

On peut améliorer de 2 à $3/2$ le facteur d'approximation de l'algorithme partant d'un arbre couvrant de poids minimum T . A cette fin, considérons l'ensemble S des sommets de degré impair dans T . Notons que $|S|$ est pair car, d'une part, $\sum_v \deg_T(v) = 2(n - 1)$, et, d'autre part $\sum_v \deg_T(v) = \sum_{v \in S} \deg_T(v) + \sum_{v \notin S} \deg_T(v)$ et le second terme $\sum_{v \notin S} \deg_T(v)$ est pair par définition de S . On construit alors un couplage parfait M de poids minimum entre les sommets de S . (Cela peut se faire en temps polynomial par l'algorithme d'Edmonds, 1965.)

On montre maintenant que $w(M) \leq \text{OPT}/2$. Pour cela, considérons un tour optimal τ , et construisons τ' restreint à S en créant des raccourcis, comme précédemment. Du fait de l'inégalité triangulaire, on a $w(\tau') \leq w(\tau) = \text{OPT}$. Comme τ' couvre exactement S , c'est un cycle de longueur pair. Un tel cycle se décompose en deux couplages parfaits dans S . Un de ces deux couplages à nécessairement un poids au plus $w(\tau')/2 \leq \text{OPT}/2$. Donc le couplage parfait M satisfait $w(M) \leq \text{OPT}/2$.

Soit $H = T \cup M$. (Noter que H peut avoir des doubles arêtes, c'est-à-dire la présence de deux arêtes entre deux mêmes sommets). Par construction, H a tous ses sommets de degré pair. H est donc Eulérien⁸. Considérons un cycle Eulérien C dans H . On a $w(C) \leq w(T) + w(M) \leq \text{OPT} + \text{OPT}/2$. On construit une solution pour le voyageur de commerce en créant des raccourcis. Le tour obtenu a un poids au plus celui de C , et donc au plus $\frac{3}{2}\text{OPT}$.

8. Un cycle Eulérien dans un graphe (orienté) est un cycle passant une et une seule fois par chaque arête (arc). Dans un graphe non orienté (resp., orienté), une condition nécessaire et suffisante pour l'existence d'un cycle Eulérien est que le degré de chaque sommet soit pair (resp., que le degré entrant de chaque sommet soit égale à son degré sortant).

Remarque. On vient de voir qu'on peut approximer la solution de VOYAGEUR DE COMMERCE MÉTRIQUE à un facteur $\frac{3}{2}$. La situation est très différente si l'on considère le problème dans un cadre non symétrique, c'est-à-dire lorsque l'on a pas nécessairement $w(u, v) = w(v, u)$ pour toute paire de sommets u, v . La meilleure approximation polynomiale pour le problème VOYAGEUR DE COMMERCE MÉTRIQUE ASYMÉTRIQUE est $O(\frac{\log n}{\log \log n})$. L'article montrant ce résultat a d'ailleurs obtenu le prix du meilleur article à la principale conférence internationale annuelle en algorithmique (ACM-SIAM Symposium on Discrete Algorithms, 2010).

4.3.4 Arbres de Steiner

Le problème suivant généralise le problème de l'arbre couvrant minimum :

Le problème ARBRE DE STEINER :

Entrée : un graphe $G = (V, E)$, un sous-ensemble $S \subseteq V$, et $w : E \rightarrow \mathbb{Q}^+$;
Objectif : Trouver un arbre couvrant S de poids minimum.

Malheureusement, à l'inverse de l'arbre couvrant minimum, le problème ARBRE DE STEINER est NP-difficile. Nous allons montrer que, comme pour le voyageur de commerce, ARBRE DE STEINER est 2-approximable.

Exercice 10 Montrer qu'il existe une réduction polynomial préservant le facteur d'approximation de ARBRE DE STEINER vers ARBRE DE STEINER MÉTRIQUE, c'est-à-dire où $G = K_n$ et w satisfait l'inégalité triangulaire. En déduire un algorithme de 2-approximation pour ARBRE DE STEINER.

Il existe de meilleures approximations, dont en particulier une approximation à un facteur $1 + \frac{1}{2} \ln 3 \simeq 1.55$.

4.4 Algorithmes d'élagage

Etant donné le graphe complet K_n dont les arêtes sont pondérées positivement par w , $S \subseteq V$, et $v \in V$, on définit $\text{dist}(v, S)$ comme le poids de l'arête de poids minimum entre v et S . On considère le problème de positionnement de ressources suivant :

Le problème k -CENTRE :

Entrée : $K_n = (V, E)$, $w : E \rightarrow \mathbb{Q}^+$ satisfaisant l'inégalité triangulaire, et $k \geq 0$;
Objectif : Trouver un ensemble S de cardinalité k minimisant $\max_{v \in V} \text{dist}(v, S)$.

Le devoir en annexe vous demande de montrer qu'il existe un algorithme polynomial de 2-approximation pour ce problème, mais que, sauf si $P = NP$, il n'existe pas d'algorithme polynomial permettant d'approximer la solution optimale de ce problème à un facteur $2 - \epsilon$ pour $\epsilon > 0$ aussi petit que l'on souhaite. L'algorithme de 2-approximation est basée sur la technique dite d'*élagage* (« pruning » en anglais).

4.5 Les schémas d'approximation

4.5.1 Définition

Definition 11 Un schéma d'approximation polynomial (PTAS, pour polynomial-time approximation scheme) pour un problème de minimisation est une famille d'algorithmes $\{A_\epsilon, \epsilon > 0\}$, paramétrée par un rationnel $\epsilon > 0$, tel que, pour tout $\epsilon > 0$, A_ϵ est un algorithme polynomial $(1 + \epsilon)$ -approximant.

Note : pour un problème de maximisation, on cherche des algorithmes polynomiaux $(1 - \epsilon)$ -approximant.

Notons que A_ϵ est polynomial en la taille de l'instance x du problème pour ϵ fixé, c'est-à-dire de la forme $O(|x|^c)$. Toutefois, une fonction de ϵ peut être « cachée » dans le O et dans la constante c . Nous reviendrons sur cela dans la suite du chapitre, mais voyons d'abord un exemple.

Il existe de nombreuses approches pour concevoir un PTAS pour un problème donné. Un stratégie classique consiste à réduire l'espace de recherche en le considérant à une granularité moins fine. Plus spécifiquement, pour une instance I , on cherche une solution admissible dans $S(I)$. Typiquement, on cherche à décomposer $S(I)$ en $|S(I)|/\epsilon H$ sous-ensembles de taille ϵH , où H permet de réduire le temps de recherche, et ϵ contrôle la précision de la solution.

4.5.2 L'exemple du sac à dos

On a vu plus tôt un algorithme *pseudo-polynomial* pour le calcul d'une solution optimale au problème du sac à dos (voir la solution de l'exercice 6). Nous en donnons ici autre qui nous servira de base pour la construction d'un schéma d'approximation polynomial. Soit o_1, \dots, o_n les n objets du problème du sac à dos, de poids respectifs w_i , $i = 1, \dots, n$, et de profits respectifs p_i , $i = 1, \dots, n$. On peut découper le problème en sous-problèmes $W_{min}(i, p)$ dénotant le poids d'une solution de poids minimum rapportant exactement p lorsque l'on se restreint aux objets o_1, \dots, o_i . (S'il n'y a pas de solution, alors $W_{min}(i, p) = +\infty$). On a

$$W_{min}(i+1, p) = \begin{cases} \min\{W_{min}(i, p), W_{min}(i, p - p_{i+1}) + w_{i+1}\} & \text{si } p_{i+1} \leq p; \\ W_{min}(i, p) & \text{sinon.} \end{cases}$$

On en déduit un programme dynamique de la même façon que celui décrit dans la solution de l'exercice 6. Le profit maximum que l'on peut espérer est $\max\{p \mid W_{min}(n, p) \leq W\}$. Le programme dynamique utilise deux boucles, une sur i et une sur p . On a $1 \leq i \leq n$. Par ailleurs, si $p_{sum} = \sum_{i=1}^n p_i$, on a $0 \leq p \leq p_{sum}$. Le programme dynamique s'exécute donc en temps $O(np_{sum}) \leq O(n^2 p_{max})$ où $p_{max} = \max_{i=1}^n p_i$. On retrouve un temps pseudo-polynomial puisqu'il dépend polynomiallement de la valeur des profits p_i . Nous allons toutefois utiliser ce programme dynamique pour décrire un schéma d'approximation polynomial, c'est-à-dire une famille d'algorithmes paramétrés par un $\epsilon > 0$ arbitrairement petit, tel que, pour ϵ fixé, A_ϵ approxime la solution optimale en temps polynomial, à un facteur $1 - \epsilon$ près.

Selon la technique générale exposée ci-dessus, l'idée pour obtenir un schéma d'approximation polynomial est de regrouper les p_{sum} valeurs possibles de la solution optimale, en paquets de μ valeurs consécutives pour un μ bien choisi. Ainsi, il ne s'agira plus que de traiter de p_{sum}/μ valeurs possibles. En fait, pour le schéma ci-dessous, il est préférable de jouer sur p_{max} . En effet, si l'on filtre les objets en éliminant ceux dont leur seul poids excède la borne W , alors on a $\text{OPT} \geq p_{max}$. De plus, si l'on utilise des profits $p'_i \propto p_i/p_{max}$, alors on obtiendra la résolution

du programme dynamique correspondant en $O(n^c p'_{max}) = O(n^c)$, c'est-à-dire polynomial en n , et non-plus pseudo-polynomial. Ces arguments vont apparaître explicitement dans l'analyse de l'algorithme ci-dessous. Soit $\mu > 0$ dont nous fixerons la valeur lors de l'analyse de l'algorithme.

Schéma d'approximation A_ϵ pour SAC à DOS :

pour chaque objet i , soit $p'_i = \lfloor \frac{p_i}{\mu} \rfloor$;
 résoudre SAC à DOS par programmation dynamique avec profits p'_i ;
 retourner la solution S_μ obtenue.

Soit OPT la valeur de la solution optimale, correspondant à l'ensemble d'objets S^* . Pour tout i , on a $\frac{p_i}{\mu} \geq p'_i \geq \frac{p_i}{\mu} - 1$, c'est-à-dire

$$p_i \geq \mu p'_i \geq p_i - \mu.$$

Donc, pour tout ensemble d'objets S :

$$\sum_{i \in S} p_i \geq \mu \sum_{i \in S} p'_i \geq \sum_{i \in S} p_i - \mu |S|.$$

Par ailleurs, l'ensemble d'objet S_μ est optimal pour les p'_i , et S_μ est admissible pour les valeurs p_i puisque les poids w_i ne sont pas modifiés. En conséquence :

$$\sum_{i \in S_\mu} p'_i \geq \sum_{i \in S^*} p'_i.$$

Donc

$$\sum_{i \in S_\mu} p_i \geq \mu \sum_{i \in S_\mu} p'_i \geq \mu \sum_{i \in S^*} p'_i \geq \sum_{i \in S^*} p_i - \mu |S^*| \geq \sum_{i \in S^*} p_i - n\mu = \text{OPT} - n\mu.$$

Soit $\epsilon > 0$ arbitrairement petit. En fixant $\mu = \frac{\epsilon p_{max}}{n}$, et en notant que $p_{max} \leq \text{OPT}$, on obtient

$$\sum_{i \in S_\mu} p_i \geq \text{OPT} - \epsilon p_{max} \geq (1 - \epsilon)\text{OPT}.$$

Ainsi, la solution S_μ approxime donc la solution optimale à un facteur $1 - \epsilon$ près.

Le temps d'exécution de l'algorithme A_μ est $O(n^2 \frac{p_{max}}{\mu}) = O(n^3/\epsilon)$. Il est donc polynomial en n . Il est même polynomial en $1/\epsilon$, c'est donc un FPTAS, pour *fully polynomial-time approximation scheme*. Un FPTAS est un schéma d'approximation qui s'exécute en $(\frac{1}{\epsilon})^{O(1)} n^{O(1)}$. Un temps en, par exemple, $2^{1/\epsilon} n^{O(1)}$ ne donne pas un FPTAS, mais simplement un PTAS.

4.5.3 Les classes FPTAS, PTAS et APX-complet

Non documenté.

4.6 Méthode de l'arrondi

Considérons un problème de programmation linéaire en nombres entiers, tel que :

$$\begin{array}{ll} \text{minimiser} & \mathbf{c}^T \mathbf{x} \\ & \text{sous la contrainte } A \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \in \{0, 1\}^n \end{array}$$

La programmation linéaire en nombres entiers est NP-difficile. on considère la relaxation

$$\begin{array}{ll} \text{minimiser} & \mathbf{c}^T \mathbf{x} \\ \text{sous la contrainte} & A \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

L'idée de la méthode de l'arrondi est de calculer (en temps polynomial) une solution du problème relaxé, et d'essayer d'en déduire une bonne solution du problème d'origine. Prenons l'exemple de COUVERTURE ENSEMBLE, et soit (U, \mathcal{S}) une instance de ce problème. Voici ci-dessous un programme linéaire en nombre entier correspondant à ce problème :

$$\begin{array}{ll} \text{minimiser} & \sum_{S \in \mathcal{S}} x_S \\ \text{sous la contrainte} & \sum_{S|u \in S} x_S \geq 1 \text{ pour tout } u \in U \\ & x_S \in \{0, 1\} \text{ pour tout } S \in \mathcal{S} \end{array}$$

Sa relaxation est :

$$\begin{array}{ll} \text{minimiser} & \sum_{S \in \mathcal{S}} x_S \\ \text{sous la contrainte} & \sum_{S|u \in S} x_S \geq 1 \text{ pour tout } u \in U \\ & x_S \geq 0 \text{ pour tout } S \in \mathcal{S} \end{array}$$

Voici comment, dans le cas de COUVERTURE ENSEMBLE, construire une « bonne » solution entière à partir d'une solution optimale $\{x_S, S \in \mathcal{S}\}$ du programme linéaire relaxé. On définit la *fréquence* d'un élément de U comme le nombre d'ensembles de \mathcal{S} auxquels cet élément appartient. Soit f_{max} la fréquence maximum de tout élément de U .

On prend alors comme couverture tous les ensembles $S \in \mathcal{S}$ tels que $x_S \geq 1/f_{max}$.

Soit \mathcal{C} l'ensemble des S sélectionnés. \mathcal{C} est bien une couverture. En effet, d'une part, pour tout $u \in U$, la solution du programme linéaire relaxé satisfait $\sum_{S|u \in S} x_S \geq 1$. D'autre part, soit f la fréquence de u . On a $|\{S \mid u \in S\}| = f$ et donc au moins un ensemble S tel que $u \in S$ satisfait $x_S \geq 1/f$. Comme $f \leq f_{max}$, on a donc $x_S \geq 1/f_{max}$, ce qui implique que cet ensemble est choisi par l'arrondi, et donc que u est couvert.

Comme chaque $S \in \mathcal{C}$ satisfait $x_S \geq 1/f_{max}$, la valeur du coût de la solution \mathcal{C} qui consiste à affecter $x_S = 1$ pour tout $S \in \mathcal{C}$ est au plus f_{max} fois la valeur de la solution optimale du programme linéaire relaxé, qui est elle même au plus la valeur de la solution optimale du programme linéaire en nombre entier. Cet arrondi donne donc un algorithme de f_{max} -approximation de COUVERTURE ENSEMBLE.

Application. Les instances $G = (V, E)$ de COUVERTURE SOMMET sont des instances (U, \mathcal{S}) de COUVERTURE ENSEMBLE avec $f_{max} = 2$. Pour cela, poser $U = E$ et $\mathcal{S} = \{\text{star}(u), u \in V\}$ où $\text{star}(u)$ représente l'ensemble des arêtes incidentes au sommet u . La méthode de l'arrondi ci-dessus donne donc un autre algorithme de 2-approximation de COUVERTURE SOMMET .

5 Algorithmes paramétrés

Afin d'illustrer l'intérêt des algorithmes paramétrés, considérons l'exemple d'un problème Π_f d'optimisation, d'ensemble d'instances \mathcal{I} . Pour $I \in \mathcal{I}$, rappelons que l'on note $S(I)$ l'ensemble

des solutions admissibles pour I , et que l'objectif est de trouver, pour tout $I \in \mathcal{I}$, une solution $x^* \in S(I)$, tel que $f(x^*) = \min_{x \in S(I)} f(x)$. Considérons la « version décision » de ce problème :

Entrée : $I \in \mathcal{I}$, et un entier $k \geq 0$;

Question : existe-t-il $x \in S(I)$ tel que $f(x) \leq k$?

Supposons que le problème ci-dessus soit NP-complet. Sauf si $P = NP$, une résolution exacte de ce problème ne peut se faire en temps $(|I| + \log k)^c$ où c est une constante. On peut par exemple imaginer l'existence d'algorithmes de temps d'exécution $O(k2^{|I|})$, ou $O(|I|^k)$, ou $O(2^k|I|^c)$. Aucun de ces algorithmes n'est polynomial, ni même pseudo-polynomial. Toutefois, les performances de ces algorithmes sont très différentes, et ces différences apparaissent cruciales lorsque l'on fixe le paramètre k .

Fixer le paramètre k peut avoir un intérêt dans les circonstances suivantes. Imaginons qu'un responsable d'entreprise ait à prendre un choix stratégique entre deux options A et B . Supposons que l'option A soit préférable à B si pour toute les branches I de l'entreprise, il existe $x \in S(I)$ tel que $f(x) \leq 3$. Cela peut par exemple être le cas si B nécessite un gros investissement, alors que A pourrait se faire avec les ressources actuelles de l'entreprise s'il existe x tel que $f(x) \leq 100$. En revanche, si un tel x n'existe pas, alors on suppose que l'investissement pour A serait encore plus élevé que pour B , et B serait donc préférable. Dans le cadre de cet exemple, à paramètre k fixé, les algorithmes ci-dessus ont des performances $O(2^{|I|})$, $O(|I|^k)$, et $O(|I|^c)$, respectivement. Le deuxième algorithme apparaît donc comme le plus approprié à la situation afin de permettre au responsable d'entreprise de faire son choix.

L'objectif de la complexité paramétrée est précisément de distinguer ce qui rend un problème difficile. Est-ce la taille de l'instance, ou est-ce plutôt un paramètre lié à ces instances ? Dans l'exemple ci-dessus, la difficulté du problème provient du paramètre k : pour un k fixé, le problème est polynomial. Le paramètre à considérer n'est pas forcément une borne sur la valeur de la solution optimale, comme dans l'exemple ci-dessus. On peut par exemple, dans le cadre de problèmes de graphes, citer de nombreux exemples de problèmes NP-complets en général, mais polynomiaux si le degré des graphes est borné, ou si leur genre est borné, ou si leur largeur arborescente est bornée, etc. Ainsi la difficulté de problèmes (ou, pour simplifier, leur caractéristique « exponentielle ») peut être liée à des caractéristiques des instances distinctes de leur simple taille.

Sous sa forme paramétrée, le problème de décision du début de ce chapitre se présente sous la forme :

Entrée : $I \in \mathcal{I}$;

Paramètre : $k \geq 0$;

Question : existe-t-il $x \in S(I)$ tel que $f(x) \leq k$?

La question que pose la complexité paramétrée est : ce problème est-il polynomial à paramètre k fixé, et, si oui, quelle est la forme de la dépendance de la complexité du problème en le paramètre ?

5.1 La classe FPT

Introduction non documentée.

5.1.1 Algorithmes polynomiaux à paramètre fixé

Considérons les trois problèmes suivants :

Le problème ENSEMBLE DOMINANT MINIMUM (MINDOMSET) :

Entrée : un graphe $G = (V, E)$, un entier $k \geq 0$;

Question : existe-t-il un ensemble $D \subseteq V$ tel que $\forall u \notin D, \exists v \in D \mid \{u, v\} \in E$, et $|D| \leq k$?

Le problème CLIQUE MAXIMUM (CLIQUEMAX) :

Entrée : un graphe $G = (V, E)$, un entier $k \geq 0$;

Question : existe-t-il un ensemble $K \subseteq V$ tel que $\forall u, v \in K, \{u, v\} \in E$, et $|K| \geq k$?

Le problème ENSEMBLE INDÉPENDANT MAXIMUM (MIS) :

Entrée : un graphe $G = (V, E)$, un entier $k \geq 0$;

Question : existe-t-il un ensemble $I \subseteq V$ tel que $\forall u, v \in I, \{u, v\} \notin E$, et $|I| \geq k$?

Tous ces problèmes ont un algorithme (exact) consistant à tester tous les ensembles de k sommets possibles. Il y a $\binom{n}{k}$ tels ensembles, où n représente le nombre de sommets du graphe. Comme la vérification des propriétés souhaitées dans chacun des trois exemples de problèmes se fait en temps polynomial, on obtient donc un algorithme de complexité $n^{O(k)}$. A paramètre k fixé, ces trois problèmes sont polynomiaux. Cependant, la dépendance en k est forte. Le chapitre ci-dessous présente l'exemple d'un problème pouvant se traité par un algorithme dont la dépendance en k est nettement plus faible.

5.1.2 Arbre de recherche pour COUVERTURE SOMMET

Tout comme les trois problèmes mentionnés ci-dessus, le problème COUVERTURE SOMMET admet un algorithme de temps d'exécution $n^{O(k)}$. Dans ce chapitre, nous allons montrer comment obtenir un algorithme paramétré dont les dépendances en k et en n de la complexité peuvent être séparées.

Considérons un algorithme consistant à prendre les sommets un par un dans un ordre arbitraire, et à explorer pour chacun d'entre eux les deux sous-problèmes consistant à ajouter le sommet actuellement considéré à la couverture, ou à ne pas l'ajouter. Cet algorithme appliqué à un graphe $G = (V, E)$ se présente de la façon suivante. Pour $v \in V$, notons $N(v)$ l'ensemble des voisins de v dans G . Pour $W \subseteq V$, on note $G \setminus W$ le graphe obtenu en supprimant de G les sommets de W ainsi que les arêtes incidentes à ces sommets. Les deux branches de l'exploration correspondent à l'alternative : mettre v dans la couverture ou pas. Si v est dans la couverture, alors il faut et il suffit de couvrir $G \setminus \{v\}$ avec au plus $k - 1$ sommets. En revanche, si v n'est pas dans la couverture, alors il faut mettre tous les voisins de v dans la couverture afin de couvrir les arêtes incidentes à v , et, il faut et il suffit alors de couvrir $G \setminus (\{v\} \cup N(v))$ avec au plus $k - |N(v)|$ sommets. Remarquons le l'algorithme ne fait pas de choix. En particulier, il ne procède pas comme un algorithme glouton qui essaierait d'effectuer un bon choix local afin de trouver une bonne solution globale. L'algorithme considère les deux choix, et effectue l'exploration de ces deux choix. Schématiquement, cela se traduit par l'algorithme suivant :

Algorithme de branchement I pour COUVERTURE SOMMET :

début

```

    choisir  $v \in V(G)$  quelconque
    si  $G \setminus \{v\}$  à une couverture-sommet  $C$  de taille  $\leq k - 1$ 
    alors
        retourner  $C \cup \{v\}$ 
    sinon
        si  $G \setminus (\{v\} \cup N(v))$  à une couverture-sommet  $C$  de taille  $\leq k - |N(v)|$ 
        alors
            retourner  $C \cup N(v)$ 
        sinon
            retourner « non »

```

fin

Dans l'algorithme ci-dessus, on sous entend que la vérification de l'existence d'une couverture-sommet dans $G \setminus \{v\}$ et $G \setminus (\{v\} \cup N(v))$ s'effectue selon le même processus, récursivement. A l'exécution de cet algorithme correspond un arbre binaire. La racine de cet arbre est G , qui possède deux fils $G \setminus \{v\}$ et $G \setminus (\{v\} \cup N(v))$. Ces derniers possèdent à leur tour deux fils, respectivement,

$$G \setminus \{v, v'\} \text{ et } G \setminus (\{v, v'\} \cup N(v'))$$

où v' est un sommet quelconque de $G \setminus \{v\}$, et

$$G \setminus (\{v, v''\} \cup N(v)) \text{ et } G \setminus (\{v, v''\} \cup N(v) \cup N(v''))$$

où v'' est un sommet quelconque de $G \setminus (\{v\} \cup N(v))$. Donc, au niveau r de la récursivité, l'arbre possède 2^r sommets. La croissance de l'arbre, et donc de la complexité de l'algorithme est donc exponentielle en la hauteur de l'arbre d'exploration. Notons toutefois que cet arbre ne peut pas avoir une profondeur plus de k . En effet, à chaque niveau de l'arbre la taille de la couverture considérée décroît au moins de 1. Lorsque l'on arrive à une feuille, il suffit de vérifier que le graphe courant n'a plus d'arêtes, test s'effectuant trivialement en temps polynomial. Par ailleurs, chaque branchement nécessite de mettre à jour le graphe en supprimant des sommets et des arêtes, ce qui s'effectue également en temps polynomial. La complexité de cet algorithme est donc au plus

$$2^k n^{O(1)}.$$

Ainsi, on est passé d'une complexité $n^{O(k)}$ en une complexité $2^k n^{O(1)}$, « séparant » ainsi la complexité en k et en n . Le chapitre suivant discute de ce type de complexité.

5.1.3 La classe FPT

Definition 12 *Une problème paramétré, d'ensemble d'instances \mathcal{I} et de paramètre k , appartient à FPT (pour fixed-parameter tractable) s'il existe un algorithme décidant en temps $f(k) n^{O(1)}$ de tout instance $I \in \mathcal{I}$ de taille n , où f est une fonction calculable qui ne dépend que de k .*

Ainsi, un problème FPT dépend polynomialement de la taille n de l'instance, quoique potentiellement exponentiellement du paramètre k . Le chapitre précédent montrer que COUVERTURE SOMMET ∈ FPT. En revanche, nous verrons dans le chapitre 5.4.3 que, sauf si P = NP, aucun de problèmes MIS, CLIQUEMAX et MINDOMSET n'appartient à FPT. Lorsqu'un problème appartient à FPT, il convient d'essayer de minimiser la fonction f dictant la dépendance au paramètre. Par exemple, dans le cas de COUVERTURE SOMMET, on peut être intéressé au développement d'algorithmes de temps d'exécution $(1 + \epsilon)^k n^{O(1)}$ pour $\epsilon < 1$. En revanche, il est connu que, sauf si P = NP, il n'existe pas d'algorithmes de temps d'exécution $2^{o(k)} n^{O(1)}$ pour COUVERTURE SOMMET.

5.2 Les noyaux : l'exemple de MAXSAT

Les problèmes FPT possède une structure particulière, dont l'existence d'un *noyau*, c'est-à-dire un sous-problème dont le taille ne dépend que de du paramètre k . Nous allons illustrer cette notion dans la cas du problème MAXSAT :

Le problème MAXSAT :

Entrée : n variables booléennes x_1, \dots, x_n , et m clauses disjonctives C_1, \dots, C_m ;

Paramètre : $k \geq 0$.

Question : existe-t-il une affectation des variables telle que $\geq k$ clauses sont satisfaites ?

Notons tout d'abord que la réponse au problème MAXSAT est oui si k est petit, plus précisément si $k \leq \frac{m}{2}$. En effet, considérons une affectation quelconque des n variables à vrai ou faux. Remarquons que si une clause C_i n'est pas satisfaites, alors inverser l'affectation des variables (c'est-à-dire affecter à faux les variable affectées à vrai, et réciproquement) permet de satisfaire la clause C_i . Ainsi, si moins de k clauses sont satisfaites, alors, en inversant l'affectation des variables, on obtient $m - k \geq m - \frac{m}{2} = \frac{m}{2}$ clauses satisfaites.

On suppose donc que $k > \frac{m}{2}$, donc $m < 2k$. L'ensemble F des m clauses peuvent être séparées en deux classes, celle des classes « longue », et celle des classes « courtes ». Les clauses longues sont celles dont le nombre de littéraux est au moins k . Les classes courtes sont celles qui ne sont pas longues. On a ainsi $F = F_\ell \cup F_c$. Soit ν le nombre de clauses longues, c'est-à-dire $\nu = |F_\ell|$. Notons maintenant que la réponse au problème MAXSAT est oui si $k \leq \nu$. En effet, un simple algorithme glouton permet de satisfaire k clauses. L'algorithme procède de la façon suivante. Soit $F_\ell = \{C_1, \dots, C_\nu\}$. L'algorithme considère les clauses C_1, \dots, C_k une par une, en commençant pas la clause C_1 . Lorsqu'il considère la clause C_i , il choisit un littéral non encore affecté à vrai dans la clause C_i , et affecte la variable booléenne correspondante afin de rendre vrai ce littéral. Cela est possible car lorsque la clause C_i est considérée, $1 \leq i \leq k$, d'une part au plus $i - 1$ variables ont été affectées, et, d'autre part, chaque clause longue possède au moins k littéraux.

Nous sommes ainsi rendu au traitement du problème lorsque $k > \nu$. En ce cas, au moins k clauses de F peuvent être satisfaites si et seulement si au moins $k - \nu$ clauses de F_c peuvent être satisfaites. En effet, supposons que $k - \nu$ clauses de F_c peuvent être satisfaites. Il ne reste qu'à satisfaire ν clauses de F_ℓ . La satisfaction de $k - \nu$ clauses de F_c ne nécessite de fixer qu'au plus $k - \nu$ variables. Cela suffit à permettre à l'algorithme glouton de satisfaire ν clauses de F_ℓ . Réciproquement, s'il n'est pas possible de satisfaire au moins $k - \nu$ clauses de F_c , alors satisfaire l'ensemble de toutes les clauses longues ne suffit pas à satisfaire au moins k clauses.

Remarquons maintenant que $|F_c| = m - \nu \leq m < 2k$. Par ailleurs, chaque clause courte ne contient qu'au plus k littéraux. Donc, le nombre de littéraux, et donc le nombre total de variables intervenant dans les clauses courtes est $< 2k^2$. Il est donc possible de décider si $k - \nu$ clauses de F_c peuvent être satisfaites en temps $2^{O(k^2)}n$ en testant toutes les affectations possibles des au plus $2k^2$ variables intervenant dans les clauses courtes.

Ainsi, MAXSAT ∈ FPT. En fait, le problème MAXSAT $(F_c, k - \nu)$ est un noyau du problème MAXSAT (F, k) . En effet, le problème $(F_c, k - \nu)$ ne fait intervenir qu'au plus $2k$ clauses et $2k^2$ variables. Il est donc d'une taille ne dépendant que du paramètre k , indépendant de la taille $m + n$ du problème initial.

5.3 Algorithmes de branchement

Introduction non documentée.

5.3.1 Algorithme paramétré de branchement pour COUVERTURE SOMMET

Un algorithme élémentaire de branchement pour COUVERTURE SOMMET a été présenté au chapitre 5.1.2. Nous allons présenter un algorithme plus sophistiqué, quoique reposant sur la même logique, permettant de diminuer la dépendance au paramètre k . L'algorithme ne fait que distinguer trois cas particuliers : le cas où le sommet traité v est de degré 1, le cas $\deg(v) = 2$ et le cas $\deg(v) \geq 3$.

- Dans le cas $\deg(v) = 1$, notons v' l'unique voisin de v dans le graphe G . Observons alors que s'il existe une couverture C de G de taille $\leq k$ contenant v , alors l'ensemble $(C \setminus \{v\}) \cup \{v'\}$ est également une couverture de G de taille $\leq k$. On ne branchera donc pas dans le cas $\deg(v) = 1$.
- Dans le cas $\deg(v) = 2$, notons v', v'' les deux voisins de v dans le graphe G . Observons alors que s'il existe une couverture C de G de taille $\leq k$ contenant v et les deux sommets v', v'' , alors l'ensemble $(C \setminus \{v\})$ est également une couverture de G de taille $\leq k$. Egale-ment, s'il existe une couverture C de G de taille $\leq k$ contenant v et un des deux sommets v', v'' (disons v'), alors l'ensemble $(C \setminus \{v\}) \cup \{v''\}$ est également une couverture de G de taille $\leq k$. Lorsque l'on branchera dans le cas $\deg(v) = 2$, on rajoutera donc dans la couverture courante soit $\{v\} \cup N(N(v))$, soit $N(v)$.

On en déduit l'algorithme ci-dessous.

Algorithme de branchement II pour COUVERTURE SOMMET :

début

```
choisir  $v \in V(G)$  quelconque et considérer les trois cas suivants :  
si  $\deg(v) = 1$  alors  
    si  $G \setminus (\{v\} \cup N(v))$  une couverture  $C$  de taille  $\leq k - 1$  alors retourner  $C \cup N(v)$   
    sinon retourner « non »  
si  $\deg(v) = 2$  alors  
    si  $G \setminus (\{v\} \cup N(v) \cup N(N(v)))$  a une couverture  $C$  de taille  $\leq k - (1 + |N(N(v))|)$   
    alors retourner  $C \cup \{v\} \cup N(N(v))$   
    sinon  
        si  $G \setminus (\{v\} \cup N(v))$  a une couverture  $C$  de taille  $\leq k - |N(v)|$   
        alors retourner  $C \cup N(v)$   
        sinon retourner « non »  
si  $\deg(v) \geq 3$  alors  
    si  $G \setminus \{v\}$  a une couverture  $C$  de taille  $\leq k - 1$   
    alors retourner  $C \cup \{v\}$   
    sinon  
        si  $G \setminus (\{v\} \cup N(v))$  a une couverture  $C$  de taille  $\leq k - |N(v)|$   
        alors retourner  $C \cup N(v)$   
        sinon retourner « non »
```

fin

Dans l'algorithme ci-dessus, on sous entend que, comme dans le cas de la version élémentaire du branchement (voir chapitre 5.1.2), la vérification de l'existence d'une couverture-sommet dans les sous-graphes s'effectue selon le même processus, récursivement. Les mises à jour des graphes et des ensembles couvrant se font en temps polynomial. Le temps d'exécution de l'algorithme

ci-dessus ne dépend que du nombre de feuilles de l'arbre de branchement. Rappelons que cet arbre a une profondeur au plus k puisque chaque branchement conduit à rajouter un sommet dans la couverture.

Notons T_h le nombre de feuilles dans un arbre de branchement relatif à l'existence d'une couverture de taille h . Le cas $\deg(v) = 1$ conduit à une récurrence $T_h = T_{h-1}$ car ce cas n'induit pas de branchement. Le cas $\deg(v) = 2$ conduit à une récurrence $T_h = 2T_{h-2}$ car chacun des deux branches conduit à ajouter soit exactement deux sommets (la branche $C \cup N(v)$), soit au moins deux sommets (la branche $C \cup \{v\} \cup N(N(v))$). Finalement, le cas $\deg(v) \geq 3$ conduit à une récurrence $T_h = T_{h-1} + T_{h-3}$. Le temps d'exécution (au pire cas) de l'algorithme est lié à T_k , pour la récurrence conduisant à la plus grande valeur. Ce calcul est décrit dans le chapitre suivant.

5.3.2 Analyse des algorithmes de branchement

Soit d_1, \dots, d_r i entiers positifs, et notons $d = \max_i d_i$. Considérons une récurrence de la forme $T_0 = T_1 = \dots = T_{d-1} = 0$ et

$$T_h = T_{h-d_1} + T_{h-d_2} + \dots + T_{h-d_r}.$$

Alors, asymptotiquement, $T_k \propto \alpha^k$, où α est la racine du polynôme caractéristique

$$p(x) = x^d - x^{d-d_1} - \dots - x^{d-d_r}$$

de plus grande valeur absolue.

Considérons, à titre d'exemple le cas des trois récurrences liées au calcul du temps d'exécution de l'algorithme de branchement II pour COUVERTURE SOMMET décrit dans le chapitre précédent. La récurrence $T_h = T_{h-1}$ a comme équation caractéristique $x - 1 = 0$, de racine $\alpha_1 = 1$. La récurrence $T_h = 2T_{h-2}$ a comme équation caractéristique $x^2 - 2 = 0$, de racine maximale $\alpha_2 = \sqrt{2}$. Finalement, la récurrence $T_h = T_{h-1} + T_{h-3}$ a comme équation caractéristique $x^3 - x^2 - 1 = 0$. Cette dernière équation a comme racine maximale $\alpha_3 \simeq 1,47$. Ainsi, il est possible d'affirmer que le nombre de feuille de l'algorithme de branchement est asymptotiquement au plus $1,47^k$. La complexité paramétrée de l'algorithme est donc $1,47^k n^{O(1)}$. Pour information, le meilleur algorithme paramétré pour COUVERTURE SOMMET a une complexité $1,28^k n^{O(1)}$.

5.4 Classes de complexité paramétrée

Introduction non documentée.

5.4.1 Le problème SAT constraint

Soit $F(x_1, \dots, x_n)$ une formule booléenne, et soit $x_i = x_i^*$ une affectation des variables x_i telle que $F(x_1^*, \dots, x_n^*)$ soit vraie. On définit la *force* de (x_1^*, \dots, x_n^*) le nombre d'indices i tels que $x_i^* = \text{vrai}$.

Le problème SAT CONSTRAINT :

Entrée : une formule booléenne F sous FNC, et un entier $k \geq 0$;

Question : existe-t-il une affectation des variables booléennes telle que F soit vraie avec une force exactement k ?

Différence entre « exactement k » et « au plus k » : le cas 2-SAT contraint. Non documenté.

5.4.2 Formules booléennes t -normalisées

Definition 13 Une formule booléenne F est appelée t -normalisée si elle peut être écrite sous la forme alternée $\bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \bigwedge_{i_3 \in I_3} \dots$ de littéraux, avec t occurrences de \bigwedge ou \bigvee .

Ainsi, une formule F sous forme normale conjonctive est 2-normalisée.

Un graphe $G = (V, E)$ admet un ensemble indépendant de taille exactement k si et seulement si la formule

$$\bigwedge_{\{u,v\} \in E} (\overline{x_u} \vee \overline{x_v})$$

est vraie avec force exactement k . (On note que cette réduction se fait en temps polynomial.) Cette formule est 1-normalisée. Plus généralement, une formule de r -SAT est 1-normalisée, pour tout r fixé. En revanche, un graphe $G = (V, E)$ admet un ensemble dominant de taille exactement k si seulement si la formule

$$\bigwedge_{u \in V} \bigvee_{v \in \{u\} \cup N(u)} x_v$$

est vraie avec force exactement k . (De nouveau, on note que cette réduction se fait en temps polynomial.) Cette formule est 2-normalisée.

5.4.3 Les classes $W[t]$

Definition 14 $W[t]$ est la classe de tous les problèmes paramétrés qui peuvent être transformés par une réduction polynomiale paramétrée en le problème de la satisfaction contrainte (i.e., avec force spécifiée) de formules booléennes présentées sous forme t -normalisée

Non documenté.

6 Algorithmes probabilistes

Introduction non documentée.

Machine RAM avec capacité de choisir un élément uniformément aléatoirement dans tout ensemble de taille polynomiale en la taille des données.

6.1 Analyse probabiliste

6.1.1 Rappels élémentaires de la théorie des probabilités

Un espace probabiliste discret est défini par une paire (Ω, μ) où Ω est un ensemble dénombrable (fini ou infini) et $\mu : \Omega \rightarrow [0, 1]$ est telle que $\sum_{\omega \in \Omega} \mu(\omega) = 1$. Pour $S \subseteq \Omega$, la probabilité de S est $\Pr[S] = \sum_{\omega \in S} \mu(\omega)$. En particulier, $\mu(\omega)$ est la probabilité de $\omega \in \Omega$.

Une variable aléatoire sur (Ω, μ) à valeurs entières est une fonction $X : \Omega \rightarrow \mathbb{N}$ ou $X : \Omega \rightarrow \mathbb{Z}$. On a alors

$$\Pr[X = i] = \sum_{w \text{ t.q. } X(\omega)=i} \mu(\omega) = \sum_{\omega \in X^{-1}(i)} \Pr(\omega).$$

L'espérance $\mathbf{E}X$ d'une variable aléatoire X est définie par

$$\mathbf{E}X = \sum_{i \in \mathbb{N}} i \Pr[X = i]$$

souvent abrégée en $\mathbf{E}X = \sum_{i=1}^{+\infty} i \Pr[i]$. Notez que cette somme peut très bien ne pas être définie (i.e., diverger vers $+\infty$). Toutes les variables aléatoires n'ont donc pas d'espérance finie. Evidemment, toute variable aléatoire définie sur un espace probabiliste Ω fini a une espérance finie. On préfèrera la terminologie « espérance » à « moyenne » car la moyenne fait essentiellement référence à des calculs empiriques ayant pour objectif d'approximer l'espérance au moyen de la loi des grands nombres.

Une variable aléatoire de Bernouilli de paramètre $p \in [0, 1]$ est une variable aléatoire X à valeur 0 ou 1 telle $\Pr[X = 1] = p$. On a donc $\mathbf{E}X = p$.

Propriétés élémentaires. Un ensemble $S \subseteq \Omega$ est appelé *événement*. Par extension, on parlera également d'événement « $X \in U$ » où $U \subseteq \mathbb{N}$ ou \mathbb{Z} pour faire référence à l'événement $S = \{\omega \in \Omega : X(\omega) \in U\}$.

— *Sous-additivité des probabilités.* Pour deux événements quelconques A et B , on a

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B).$$

— *Disjonction.* Deux événements A et B sont dits *disjoints* si et seulement si $A \cap B = \emptyset$. Si A et B sont disjoints, alors $\Pr(A \cup B) = \Pr(A) + \Pr(B)$. Notez qu'on peut avoir $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ sans que A et B soient disjoints. En effet $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ implique simplement que $\Pr[A \cap B] = 0$.

— *Indépendance.* Deux événements A et B sont dits indépendants si et seulement si

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B].$$

Egalement, deux variables aléatoires X et Y sont indépendantes si, pour tout x et tout y ,

$$\Pr[(X = x) \wedge (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y].$$

Attention, ne pas confondre l'indépendance mutuelle avec l'indépendance deux-à-deux. Par exemple, soit X_1 et X_2 deux variables aléatoires booléennes indépendantes, et soit $X_3 = X_1 \oplus X_2$ où \oplus dénote le ou-exclusif. Ces trois variables aléatoires sont deux-à-deux indépendantes, mais elles ne sont pas mutuellement indépendantes puisque les valeurs de X_1 et X_2 déterminent entièrement celle de X_3 .

— *Linéarité de l'espérance.* Pour toutes variables aléatoires X et Y , on a

$$\mathbf{E}(X + Y) = \mathbf{E}X + \mathbf{E}Y.$$

En particulier, on sera souvent amener dans le cours à considérer une séquence X_i , $i \geq 1$, de variables aléatoires identiquement distribuées, et à étudier $X = \sum_{i=1}^n X_i$. En ce cas, $\mathbf{E}X = \sum_{i=1}^n \mathbf{E}X_i = n\nu$ où ν est l'espérance de chaque X_i .

Abréviation. On note souvent « i.d. » pour identiquement distribué, et « i.i.d. » pour indépendante identiquement distribué.

Probabilité conditionnelles (non documentées)

6.1.2 Exemple de la diffusion

Considérons le problème de la diffusion d'une rumeur dans un graphe non orienté, tel que présenté au chapitre 2.4.1. Nous avons vu que calculer un protocole de diffusion optimal est difficile (le problème de décision correspondant est NP-complet). Nous allons étudier le protocole de diffusion probabiliste suivant dans un graphe $G = (V, E)$ à partir de la source $s \in V$:

Protocole de diffusion probabiliste :

début

```

répéter /* chaque itération de la boucle correspond à une étape du protocole de diffusion */
    pour tout  $u \in V$  ayant reçu l'information faire
        choisir un voisin  $v$  de  $u$  aléatoirement uniformément parmi tous les voisins de  $u$ 
        transmettre l'information de  $u$  à  $v$  ;
    jusqu'à tous les sommets ont reçu l'information
fin.
```

Notez que, dans le protocole de diffusion probabiliste, un sommet peut être informé plusieurs fois, à la même étape ou à des étapes différentes.

a) **Analyse en moyenne.** Soit P un chemin de longueur $\ell = \text{dist}_G(s, u)$ entre la source s et un sommet u . Soit $e = \{x, y\}$ une arête de ce chemin. Soit $B_i, i \geq 1$ la séquence de variables aléatoires i.i.d. de Bernoulli de paramètre $p = 1/\deg(x)$. C'est-à-dire :

$$B_i = \begin{cases} 1 & \text{avec probabilité } 1/\deg(x) \\ 0 & \text{avec probabilité } 1 - 1/\deg(x) \end{cases}$$

On a $\mathbb{E}B_i = 1/\deg(x)$. Soit T_e la variable aléatoire définie comme égale au plus petit t tel que $\sum_{i=1}^t B_i = 1$. Nous avons

$$\Pr[T_e = k] = p(1-p)^{k-1}.$$

Il en découle que

$$\mathbf{ET}_e = \sum_{k \geq 1} k \Pr[T_e = k] = p \sum_{k \geq 1} k(1-p)^{k-1} = 1/p.$$

Donc $\mathbf{ET}_e = \deg(x)$, et donc $\mathbf{ET}_e \leq \Delta$ où Δ dénote le degré maximum des sommets du graphe. Ainsi, le nombre espéré d'étapes de l'algorithme probabiliste pour progresser d'une arête le long de P est au plus Δ .

Soit T la variable aléatoire définie comme égale aux nombre d'étapes pour traverser P . Par la linéarité de la moyenne, nous obtenons :

$$\mathbf{ET} = \sum_{e \in E(P)} \mathbf{ET}_e \leq \ell \Delta.$$

Le *diamètre* d'un graphe est la longueur maximum d'un plus court chemin, c'est-à-dire

$$D = \max_{(u,v) \in V \times V} \text{dist}_G(u, v).$$

On déduit du calcul de l'espérance de T que, pour tout sommet u d'un graphe de diamètre D et de degré maximum Δ , le nombre espéré d'étapes pour que le sommet u reçoive la rumeur est $O(\Delta D)$. Dans le cas des graphes de degré constant, c'est asymptotiquement optimal.

En revanche, cela n'informe pas sur le pire cas, parmi tous les sommets. Par exemple, dans l'étoile de $n + 1$ sommets (l'arbre contenant 1 centre connecté à n feuilles), si la source est le centre, alors chaque feuille reçoit l'information en au plus n étapes en moyenne. Cependant,

l'espérance du nombre d'étapes pour que *toutes* les feuilles soient informées est $\Omega(n \log n)$. (Ceci correspond au problème du « collecteur de coupons »). En effet, pour $i = 1, \dots, n$, soit T_i la variable aléatoire égale aux nombre d'étapes pour informer i feuilles après que $i-1$ feuilles aient déjà été informées. Si $i-1$ feuilles sont déjà informé, la probabilité d'informer une nouvelle feuille est $p_i = (n-i+1)/n$. Ainsi, de la même manière que pour le calcul de \mathbf{ET}_e ci-dessus, on a $\mathbf{ET}_i = 1/p_i = n/(n-i+1)$. Le temps pour informer toutes les feuilles est $T = T_1 + T_2 + \dots + T_n$. Par linéarité de l'espérance, on obtient

$$\mathbf{ET} = \sum_{i=1}^n \mathbf{ET}_i = \sum_{i=1}^n \frac{n}{n-i+1} = n \sum_{i=1}^n \frac{1}{i} = n \cdot H_n$$

où H_n dénote le n ième nombre harmonique.

b) Analyse avec forte probabilité. Nous allons maintenant borner le temps de diffusion, en explicitant la probabilité d'être en deçà de ce cette borne. Nous utiliserons l'outil suivant :

Inégalité de Markov. Pour une variable aléatoire entière positive ou nulle X , on a

$$\Pr[X \geq k] \leq \mathbf{E}X/k$$

pour tout entier $k \geq 1$. En particulier, pour tout entier $k \geq 1$,

$$\Pr[X \geq k | \mathbf{E}X] \leq 1/k.$$

Soit $P = x_0, x_1, \dots, x_\ell$ un plus court chemin de longueur $\ell = \text{dist}(s, u)$ entre la source s et le sommet u , avec $x_0 = s$ et $x_\ell = u$. On a $\sum_{i=0}^\ell \deg(x_i) \leq 3n$ car :

- chaque sommet n'appartenant pas à P ne peut être connecté qu'à au plus trois sommets de P (sinon, cela créerait un raccourci, contredisant le fait que P est un plus court chemin), et
- chaque sommet appartenant à P n'est connecté qu'à au plus deux sommets de P .

En conséquence, si T_P dénote la variable aléatoire égale au nombre d'étapes pour traverser P , alors par linéarité de l'espérance, $\mathbf{ET}_P = \sum_{i=0}^{\ell-1} \deg(x_i) \leq 3n$. Soit donc \mathcal{E}_u l'événement « u n'a pas reçu le message après $6n$ étapes ». D'après Markov, on obtient :

$$\Pr[\mathcal{E}_u] = \Pr[T_P > 6n] \leq \Pr[T_P \geq 2\mathbf{ET}_P] \leq 1/2.$$

C'est une probabilité trop grande pour pouvoir affirmer quelque chose sur l'ensemble de *tous* les sommets u . Afin d'obtenir une probabilité plus faible, on considère l'événement \mathcal{E}'_u défini comme « u n'a pas reçu le message après $12n \log n$ étapes ». En considérant ces $12n \log n$ étapes comme $2 \log n$ phases indépendantes de $6n$ étapes, on obtient :

$$\Pr(\mathcal{E}_v) \leq 1/2^{2 \log n} = 1/n^2.$$

Cette probabilité est maintenant suffisamment petite pour sommer sur l'ensemble des sommets.

Soit \mathcal{E}' l'événement « il existe un sommet u qui n'a pas reçu le message après $12n \log n$ étapes », on a $\mathcal{E}' = \cup_u \mathcal{E}'_u$. Donc, par sous-linéarité des probabilités, $\Pr(\mathcal{E}') \leq \sum_u \Pr(\mathcal{E}'_u)$. Donc $\Pr(\mathcal{E}') \leq n/n^2 = 1/n$.

Donc la diffusion se fait en au plus $O(n \log n)$ étapes avec forte probabilité $1 - O(\frac{1}{n})$. C'est optimal au pire cas (de graphe), en considérant l'étoile de $n+1$ sommets mentionnée ci-dessus.

6.1.3 Analyse d'algorithmes

a) Algorithmes déterministes. Dans les analyses d'algorithmes déterministes vues jusqu'à présent dans ce document, nous nous sommes principalement focalisés sur une analyse au pire cas, c'est-à-dire, pour un algorithme A , et une taille n des données, sur la recherche de

$$T_{\max}(A) = \max_{|x|=n} T(A, x)$$

où $T(A, x)$ dénote le temps d'exécution de l'algorithme A sur la donnée x . Notons que nous aurions aussi pu considérer une analyse en moyenne d'algorithmes déterministes définie à partir d'une distribution des entrées x par

$$T_{moyen}(A) = \sum_{|x|=n} \Pr(x) \cdot T(A, x).$$

Exemple : les graphes aléatoires. $\mathcal{G}_{n,p}$ (Edgar Gilbert 1959), et $\mathcal{G}_{n,m}$ (Erdős and Rényi, 1959). G de n sommets et m arêtes : $\Pr(G) = p^m(1-p)^{\binom{n}{2}-m}$. Attention les propriétés d'un graphe aléatoire dépendent de p (constant, $\ln n/n$, $1/n$, etc.). Est-ce un bon modèle ? (densité globale, locale, loi de puissance, etc.) Non documenté.

Attachement préférentiel. Non documenté.

b) Algorithmes probabilistes. L'analyse d'algorithmes probabilistes consiste également à effectuer une analyse au pire cas, mais sur la moyenne du temps d'exécution. On s'intéresse ainsi, pour une donnée x , à

$$\mathbf{ET}(A, x) = \sum_t t \cdot \Pr[T(A, x) = t]$$

et à

$$\max_{|x|=n} \mathbf{ET}(A, x).$$

La moyenne n'étant pas toujours un bon indicateur⁹, on cherche souvent à borner le temps d'exécution en probabilité. Pour $t \geq 0$, on dira par exemple que l'algorithme A a un temps d'exécution au plus t

— asymptotiquement presque sûrement (a.p.s.) si, pour tout x tel que $|x| = n$, on a

$$\Pr[T(A, x) \leq t] \geq 1 - o(1)$$

— et avec forte probabilité (a.f.p.) s'il existe $c > 0$ tel que pour tout x tel que $|x| = n$, on a

$$\Pr[T(A, x) \leq t] \geq 1 - O(1/n^c).$$

Dans le premier cas, la notation $o(1)$ indique toute fonction de la taille n de l'entrée dont la valeur tend vers 0 lorsque $n \rightarrow +\infty$.

c) Principe Min-Max. Non documenté.

9. Par exemple, la moyenne de la taille des hommes en France donne une indication assez précise de la taille d'un français quelconque, alors que la moyenne du salaire des français ne donne qu'une idée vague du salaire d'un français quelconque car beaucoup de français ont un salaire très significativement supérieure à la moyenne des salaires.

6.2 Las Vegas et Monté Carlo

Un algorithme Las Vegas est un algorithme probabiliste qui calcule systématiquement la solution optimale, mais dont le temps d'exécution est aléatoire. Un algorithme Monté Carlo est un algorithme probabiliste dont le temps d'exécution est borné par une borne déterministe, mais qui ne calcule pas forcément systématiquement la solution optimale.

6.2.1 Un algorithme Las Vegas pour la recherche de répétitions

Soit T un tableau de longueur n pair contenant $\frac{n}{2}$ éléments distincts et $\frac{n}{2}$ occurrences d'un même élément distinct des $\frac{n}{2}$ autres. L'objectif est de trouver cet élément dans le tableau T . Si l'on passe en revu les cases du tableau, cela prend un temps $\frac{n}{2} + 2$ au pire cas. **On peut montrer (exercice)** que tout algorithme déterministe nécessite un temps au moins $\frac{n}{2} + 2$ au pire cas. Nous allons montrer qu'utiliser l'aléa permet de réduire exponentiellement cette complexité.

Considérons l'algorithme probabiliste suivant. La recherche procède par phase. A chaque phase, on choisit deux entiers i et j aléatoirement uniformément dans $[1, n]$, de façon indépendante. On vérifie alors si $i \neq j$ et si $T[i] = T[j]$. Si c'est le cas, alors on a trouvé l'élément aux multiples occurrences. Sinon, on passe à la phase suivante.

Montrons que le nombre de phases exécutées par cet algorithme est $O(\log n)$ avec forte probabilité (a.f.p.). A chaque phase, la probabilité de trouver l'élément cherché est $p = \frac{n/2(n/2-1)}{n^2}$. On a $p \geq 1/5$ dès que $n \geq 10$. La probabilité que l'algorithme ne trouve pas l'élément cherché après $c\alpha \log n$ phases est au plus

$$\left(\frac{4}{5}\right)^{c\alpha \log n} = n^{-c\alpha \log(5/4)}.$$

En choisissant $\alpha \geq 1/\log(5/4)$, on obtient que l'algorithme effectue au plus $\frac{c \log n}{\log(5/4)}$ phase, avec probabilité au moins $1 - \frac{1}{n^c}$.

Application du principe min-max.

6.2.2 Un algorithme de Monté Carlo pour MAXSAT

On se restreint ici à MAX3SAT, c'est-à-dire MAXSAT avec des clauses de longueur 3. Il s'agit, étant donnée une formule booléenne ϕ en FNC, de trouver une affectation des variables de façon à rendre le plus grand nombre de clauses de ϕ satisfaites.

On suppose sans perte de généralité qu'aucune clause ne contient un littéral et son complément, et qu'aucune clause ne contient une répétition d'un même littéral. Soit n le nombre de variables et m le nombre de clauses. Considérons l'algorithme probabiliste suivant. Chacune des variables est indépendamment fixée à 0 ou 1, aléatoirement uniformément avec probabilité $\frac{1}{2}$. Nous allons montrer que cet algorithme (qui s'exécute en temps $O(n)$) trouve en moyenne une $\frac{7}{8}$ -approximation de la solution optimale.

Soit X_i la variable aléatoire binaire telle que $X_i = 1$ si et seulement si la i ème clause est satisfait. On a $\Pr[X_i = 0] = (\frac{1}{2})^3 = 1/8$. Donc $\Pr[X_i = 1] = 7/8$, et ainsi $\mathbf{E}X_i = 7/8$. Le nombre de clauses satisfaites est $X = \sum_{i=1}^m X_i$. Par linéarité de l'espérance, on a

$$\mathbf{E}X = \sum_{i=1}^m \mathbf{E}X_i = 7m/8.$$

Par ailleurs, le nombre de clauses satisfaites ne peut pas excéder m . Le nombre moyen de clauses satisfaites trouvé par l'algorithme probabiliste est donc au moins $\frac{7}{8} \cdot \text{OPT}$.

Remarque. Notez que puisque $\mathbf{E}X \geq 7m/8$ pour toute formule 3SAT ϕ de m clauses ne contenant pas un littéral et son complément, ni une répétition d'un même littéral, on obtient qu'il est toujours possible de satisfaire au moins $7m/8$ clauses de formule de ce type.

6.2.3 Echantillonnage probabiliste

Intro non documenté.

Soit T un tableau de longueur n contenant n entiers deux à deux distincts. On recherche une approximation de la médiane de ces valeurs. Pour tout élément x de T , on définit le rang de x comme $1 + |\{y \in T / y < x\}|$. On cherche un élément $z \in T$ dont le rang est dans l'intervalle $[(\frac{1}{2} - \epsilon)n, (\frac{1}{2} + \epsilon)n]$.

Exercice 11 Montrer qu'il existe un algorithme de Monté Carlo tel que, pour tout $\alpha > 0$, cette algorithme retourne un tel élément z avec probabilité $\geq 1 - n^{-\alpha}$.

6.3 Méthode de l'arrondi probabiliste

Prenons l'exemple de COUVERTURE ENSEMBLE pondérée (chaque ensemble $S \in \mathcal{S}$ à un coût $c(S)$). Soit (U, \mathcal{S}, c) une instance de ce problème. On a vu au chapitre 4.6 un programme linéaire en nombre entier correspondant à la version non-pondérée du problème. Il se généralise trivialement à la version pondérée :

$$\begin{aligned} \text{minimiser} \quad & \sum_{S \in \mathcal{S}} x_S \cdot c(S) \\ \text{sous la contrainte} \quad & \sum_{S|u \in S} x_S \geq 1 \text{ pour tout } u \in U \\ & x_S \in \{0, 1\} \text{ pour tout } S \in \mathcal{S} \end{aligned}$$

dont la relaxation est :

$$\begin{aligned} \text{minimiser} \quad & \sum_{S \in \mathcal{S}} x_S \cdot c(S) \\ \text{sous la contrainte} \quad & \sum_{S|u \in S} x_S \geq 1 \text{ pour tout } u \in U \\ & x_S \geq 0 \text{ pour tout } S \in \mathcal{S} \end{aligned}$$

Nous allons construire une $O(\log n)$ -approximation de la solution optimale de COUVERTURE ENSEMBLE PONDÉRÉ à partir d'une solution optimale $\{x_S^*, S \in \mathcal{S}\}$ du programme linéaire relaxé. L'arrondi probabiliste procède comme suit : pour chaque $S \in \mathcal{S}$,

$$\text{on prend } S \text{ dans la couverture } \mathcal{C} \text{ avec probabilité } x_S^*. \quad (\star)$$

Estimons la probabilité qu'un élément soit couvert par \mathcal{C} . Soit $u \in U$, et supposons que u appartient à f ensembles de \mathcal{S} . Soit S_1, \dots, S_f , ces ensembles, et soit p_1, \dots, p_f les probabilités associées à chacun de ces ensembles selon la solution optimale $\{x_S^*, S \in \mathcal{S}\}$, c'est-à-dire $p_i = x_{S_i}^*$. La probabilité qu'aucun de ces ensembles soient choisis est $\prod_{i=1}^f (1 - p_i)$. Comme la solution optimale du problème relaxé satisfait les contraintes, on a $\sum_{i=1}^f p_i \geq 1$. La fonction $\prod_{i=1}^f (1 - p_i)$ est maximisée pour $p_i = 1/f$ pour tout i , et donc

$$\Pr[u \text{ est couvert par } \mathcal{C}] \geq 1 - \left(1 - \frac{1}{f}\right)^f \geq 1 - 1/e.$$

Chaque élément est donc couvert avec une probabilité constante. Cela ne suffit cependant pas pour certifier une couverture de *tous* les éléments avec une probabilité constante. Pour remédier à ce problème, on répète le choix (\star) $\alpha \ln n$ fois. C'est-à-dire : pour chaque $S \in \mathcal{S}$,

on prend S dans la couverture \mathcal{C} avec probabilité $1 - (1 - x_S^*)^{\alpha \ln n}$.

Cette solution est notée \mathcal{C}' . La constante α est choisie de façon à ce que

$$\left(\frac{1}{e}\right)^{\alpha \ln n} \leq \frac{1}{4n} .$$

Ainsi,

$$\Pr[u \text{ est non couvert par } \mathcal{C}'] \leq \left(\frac{1}{e}\right)^{\alpha \ln n} \leq \frac{1}{4n} .$$

En conséquence, par sous-additivité des probabilités,

$$\Pr[\mathcal{C}' \text{ n'est pas une couverture valide}] \leq n \cdot \frac{1}{4n} = \frac{1}{4} . \quad (1)$$

Estimons maintenant le coût espéré de la solution \mathcal{C}' . Commençons par calculer le coût espéré de \mathcal{C} :

$$\mathbf{E}[\text{coût}(\mathcal{C})] = \sum_{S \in \mathcal{S}} \Pr[S \text{ est pris}] \cdot c(S) = \sum_{S \in \mathcal{S}} x_S^* \cdot c(S) = \text{OPT}_r$$

où OPT_r dénote la valeur d'une solution optimale du problème relaxé. On obtient donc

$$\mathbf{E}[\text{coût}(\mathcal{C}')'] \leq \alpha \ln n \cdot \text{OPT}_r .$$

On a $\text{OPT}_r \leq \text{OPT}$ où OPT dénote la valeur optimale d'une solution optimale du problème initial (c'est-à-dire non relaxé). Donc $\mathbf{E}[\text{coût}(\mathcal{C}')'] \leq \alpha \ln n \cdot \text{OPT}$. En appliquant l'inégalité de Markov, on obtient :

$$\Pr[\text{coût}(\mathcal{C}') \geq 4\alpha \ln n \cdot \text{OPT}] \leq \frac{1}{4} . \quad (2)$$

En combinant les équations 1 et 2 on obtient

$$\Pr[(\mathcal{C}' \text{ n'est pas une couverture valide}) \text{ ou } (\text{coût}(\mathcal{C}') \geq 4\alpha \ln n \cdot \text{OPT})] \leq \frac{1}{2} .$$

Donc

$$\Pr[(\mathcal{C}' \text{ est une couverture valide}) \text{ et } (\text{coût}(\mathcal{C}') \leq 4\alpha \ln n \cdot \text{OPT})] \geq \frac{1}{2} .$$

En recommençant k fois ces opérations, on peut accroître exponentiellement la probabilité de succès, avec une probabilité au moins $1 - \frac{1}{2^k}$ d'obtenir une couverture valide dont le coût est au plus $4\alpha \ln n \cdot \text{OPT}$. En particulier, pour $k = \lceil \log_2 n \rceil$, on obtient une couverture valide $O(\log n)$ -approximante, avec forte probabilité.

6.4 Concentration : bornes de Chernoff

Intro non documenté.

6.4.1 Borne de Chernoff

Soit X_i , $i = 1, \dots, n$, des variables de Bernoulli indépendantes et identiquement distribuées (i.i.d.). Soit $X = \sum_{i=1}^n X_i$.

Lemma 4 Pour tout δ , $0 < \delta \leq 1$, on a $\Pr[X < (1 - \delta)\mathbf{E}X] < e^{-\delta^2 \mathbf{E}X/2}$.

Autres bornes de Chernoff non documentées.

6.4.2 Application à la diffusion probabiliste

Considérons la diffusion d'épidémie dans un réseau G de degré maximum Δ et de diamètre D . On a vu dans le chapitre 6.1.1 que $\mathbb{E}T_v \leq \Delta D$ pour tout v car en chaque sommet d'un plus court chemin de la source u à v (donc de longueur au plus D), l'espérance du temps mis pour avancer au sommet suivant du chemin est au plus Δ . Soit X_i la variable de Bernoulli qui indique si le message a avancé d'un pas le long du plus court chemin à la i ème étape. On veut borner en probabilité le temps qu'il faut attendre avant d'avoir D succès.

Soit $k = 2\Delta(D + 8 \ln n)$. En notant $Y_v = \sum_{i=1}^k X_i$, on obtient $\mathbb{E}Y_v = k/\Delta = 2(D + 8 \ln n)$. On a

$$\Pr(Y_v < D) \leq \Pr(Y_u < D + 8 \ln n).$$

Or, $\Pr(Y_v < D + 8 \ln n) = \Pr(Y_v < \frac{1}{2}\mathbb{E}Y_u)$. Donc, d'après Chernoff,

$$\Pr\left[Y_v < \frac{\mathbb{E}Y_v}{2}\right] < e^{-\frac{1}{8}\mathbb{E}Y_v} = e^{-\frac{1}{4}(D+8\ln n)} \leq e^{-2\ln n} = 1/n^2.$$

Donc, la probabilité que v n'ait pas reçu le message après k étapes est au plus $1/n^2$. Donc, par sous-linéarité des probabilités, il découle que la probabilité qu'un sommet quelconque n'ait pas reçu le message après k étapes est au plus $1/n$. Donc la diffusion se fait en au plus $O(\Delta(D + \log n))$ étapes, avec probabilité $1 - 1/n$.

6.5 Les classes de complexité probabiliste

Il est possible de définir des classes de complexité liées aux différents usages possibles de l'aléa.

Definition 15 La classe BPP consiste en tous les langages L pour lesquels il existe un algorithme probabiliste A polynomial en la taille de x tel que pour tout $x \in \Sigma^*$:

$$\begin{cases} x \in L & \Rightarrow \Pr[A \text{ accepte } x] \geq 2/3 \\ x \notin L & \Rightarrow \Pr[A \text{ rejette } x] \geq 2/3 \end{cases}$$

Par définition, on a $\text{P} \subseteq \text{BPP}$. La question de savoir si $\text{BPP} \neq \text{P}$, c'est-à-dire de savoir si l'aléa aide à résoudre en temps polynomial des problèmes non déterministiquement polynomiaux, est ouverte. À l'inverse de la question P versus NP pour laquelle il est conjecturé que $\text{NP} \neq \text{P}$, la situation n'est pas aussi tranchée pour BPP versus P . Notez que la constante $2/3$ est arbitraire dans la définition. Il est en effet possible d'amplifier toute probabilité de succès $p > 1/2$ d'un algorithme A en exécutant suffisamment de fois A et en retournant la réponse majoritaire – **non documenté**.

BPP implique des algorithmes faisant des erreurs aussi bien pour les instances $x \in L$ que pour les instances $x \notin L$ (« 2-sided error » en anglais). Les classes RP et co-RP sont plus restrictives, en impliquant des algorithmes qui ne font des erreurs que sur un seul type d'instances (« 1-sided error » en anglais).

Definition 16 La classe RP (*randomized polynomial time*) consiste en tous les langages L pour lesquels il existe un algorithme probabiliste A polynomial en la taille de x tel que pour tout $x \in \Sigma^*$:

$$\begin{cases} x \in L \Rightarrow \Pr[A \text{ accepte } x] \geq 1/2 \\ x \notin L \Rightarrow \Pr[A \text{ rejette } x] = 1 \end{cases}$$

La classe co-RP consiste en tous les langages L pour lesquels il existe un algorithme probabiliste A polynomial en la taille de x tel que pour tout $x \in \Sigma^*$:

$$\begin{cases} x \in L \Rightarrow \Pr[A \text{ accepte } x] = 1 \\ x \notin L \Rightarrow \Pr[A \text{ rejette } x] \geq 1/2 \end{cases}$$

On définit ZPP (*zero-error probabilistic polynomial-time*) comme $ZPP = RP \cap co-RP$.

Comme pour le choix de la constante $2/3$ dans la définition de BPP, le choix de la constante $1/2$ dans les définitions de RP et co-RP est arbitraire, et on pourrait choisir toute constante $p \in]0, 1[$, **par amplification**.

Par définition, on a $P \subseteq ZPP \subseteq RP \subseteq BPP$.

7 Conclusion

Ce cours a eu pour objectif de donner un aperçu des techniques algorithmiques les plus classiques. Il n'est malheureusement pas possible d'aborder tous les sujets en liaison avec l'algorithmique en un cours, et plusieurs thématiques importantes n'ont pas été présentées. Nous renvoyons ainsi le lecteur à des références pour les thématiques suivantes :

Méthodes heuristiques : Voir [8].

Algorithmes online Voir [2].

Algorithmes de streaming Voir [10].

Théorie algorithmique des jeux Voir [15].

Ordonnancement Voir [13].

Algorithmes distribués Voir [1, 7, 12].

Algorithmes parallèles Voir [6].

Références

- [1] Hagit Attiya and Jennifer Welch. *Distributed Computing : Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc. 2006.
- [2] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms* (2nd Edition). The MIT Press and McGraw-Hill Book Company, 2001.
- [4] R. Diestel. *Graph Theory*. Springer, 2006.
- [5] Förg Flum, Martin Grohe. *Parameterized Complexity Theory*. Springer Verlag, 2006.
- [6] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.
- [7] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [8] Z. Michalewicz and D. Fogel. *How to Solve It : Modern Heuristics*. Springer, 2004.
- [9] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [10] S. Muthu Muthukrishnan. *Data Stream Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science 1(2), pp117-236 (2005).
- [11] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993
- [12] David Peleg. *Distributed Computing : A Locality-Sensitive Approach*. SIAM Monographs on Discrete Maths and Applications, 2000.
- [13] Michael Pinedo. *Scheduling : Theory, Algorithms, and Systems*. Springer, 2012.
- [14] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.
- [15] Vijay Vazirani, Noam Nisan, Tim Roughgarden, Éva Tardos *Algorithmic Game Theory*. Cambridge University Press (2007)

ANNEXES

A Correction des exercices

Correction de l'exercice 1

Nous décrivons le quintuplet $M = (Q, \Gamma, \delta, q_0, F)$. On suppose l'entier n écrite en binaire sur la bande. On fixe

$$\Gamma = \{0, 1, b\}$$

et

$$Q = \{\text{cherche, trouve, oui, non}\} \text{ avec } q_0 = \text{cherche et } F = \{\text{oui, non}\}.$$

On définit la fonction de transition $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1, +1\}$ comme suit :

$$\delta(\text{cherche}, \gamma) = \begin{cases} (\text{cherche}, \gamma, +1) & \text{si } \gamma \in \{0, 1\} \\ (\text{trouve}, \gamma, -1) & \text{si } \gamma = b \end{cases}$$

et

$$\delta(\text{trouve}, \gamma) = \begin{cases} (\text{oui}, \gamma, 0) & \text{si } \gamma = 0 \\ (\text{non}, \gamma, 0) & \text{si } \gamma = 1 \end{cases}$$

Cette machine effectue donc les opérations suivantes. Dans l'état « cherche », elle lit tous les bits de n , du poids fort au poids faible, en se déplaçant vers la droite. Lorsqu'elle a lu tous les bits de n , elle lit le symbole blanc b , et, dans ce cas, passe dans l'état « trouve », et se déplace à gauche d'une case. La réponse de la machine dépend alors de la valeur du dernier bit de n lu à l'étape suivante : oui si celui-ci est nul, et non sinon (les deux états sont terminaux).

Notons que cette machine n'est pas définie pour toute les paires de $Q \times \Gamma$ car ce n'est pas nécessaire. Plusieurs machines différentes peuvent donc résoudre le problème. Notons également que cette machine ne modifie pas la bande. On pourrait modifier la machine pour qu'elle nettoie la mémoire en effaçant ce qu'elle vient de lire, en définissant

$$\delta(\text{cherche}, \gamma) = \begin{cases} (\text{cherche}, b, +1) & \text{si } \gamma \in \{0, 1\} \\ (\text{trouve}, \gamma, -1) & \text{si } \gamma = b \end{cases}$$

Néanmoins, en ce cas, la valeur du dernier bit de n est effacé de la bande, et $\delta(\text{trouve}, \gamma)$ demande également à être redéfini. En fait, il convient de sauvegarder la valeur du dernier bit lu dans les états de la machine, et donc de légèrement modifier Q . Cela est laissé en exercice.

Correction de l'exercice 2

Etant donnée une formule SAT ψ , on suppose sans perte de généralité que ψ ne contient pas de clauses triviales, c'est-à-dire du type x , du type \bar{x} , ou du type $x \vee \bar{x}$. Dans les deux premiers cas, la valeur de la variable x est nécessairement fixée (à « vrai » dans le premier cas, et à « faux » dans le second). Dans le troisième cas, la clause $x \vee \bar{x}$ est toujours vraie, et peut donc être supprimée. Dans le cas d'une formule 2-SAT ψ , on supposera donc sans perte de généralité que toutes les clauses contiennent exactement deux littéraux de variables différentes.

Etant donnée une formule 2-SAT ψ , construire le graphe orienté $G = (V, E)$ dont les sommets sont les littéraux positifs et négatifs de la formule, et dans lequel chaque clause induit une paire

d'arcs correspondant aux implications logiques liées à l'affectation des variables. Par exemple, la clause $x \vee y$ induit deux implications $\bar{x} \Rightarrow y$ et $\bar{y} \Rightarrow x$ auxquelles correspondent les arcs (\bar{x}, y) et (\bar{y}, x) . De même la clause $\bar{u} \vee v$ induit les arcs (u, v) et (\bar{v}, \bar{u}) .

S'il existe un littéral x pour lequel il existe dans G un chemin de x à \bar{x} et un chemin de \bar{x} à x (dit autrement, si x et \bar{x} sont dans la même composante *fortement* connexe de G), alors la formule ψ ne peut être satisfaisante. En effet, par transitivité, ces chemins impliquent qu'on ne peut avoir ni x , ni \bar{x} vrai car $x \Rightarrow \bar{x}$ et $\bar{x} \Rightarrow x$.

En revanche, si aucun littéral x n'est dans la même composante fortement connexe que \bar{x} alors la formule est satisfiable. En effet, supposons sans perte de généralité que $x \not\Rightarrow \bar{x}$. Notons qu'on ne peut avoir $x \Rightarrow y$ et $x \Rightarrow \bar{y}$ car sinon $x \Rightarrow \bar{x}$. Donc, fixer le littéral x à vrai dans ψ n'induit aucune contradiction par cascade d'implications. Fixer ce littéral détermine la valeur vrai ou faux de tout un ensemble d'autres littéraux y selon que l'on rencontre y où \bar{y} dans la traversé de tous les chemins d'origine x dans G . A la suite de cette opération, on obtient une formule 2-SAT φ n'incluant plus le littéral x . Si $\varphi \neq \emptyset$, on continuant en considérant un littéral x' présent dans φ . On fixe x' à vrai, et, comme précédemment, on cascade cette contrainte à tous les littéraux de la composante fortement connexe de x' . Notons que cela n'induit aucune contradiction avec les littéraux ayant été fixés par cascade à partir de x . En effet, si $x \Rightarrow y$ et $x' \Rightarrow \bar{y}$ alors on obtiendrait $x \Rightarrow \bar{x}'$, en contradiction avec le fait que le littéral x' n'a pas été fixé par le littéral x . On peut donc procéder ainsi de suite jusqu'à obtenir la formule vide, qui est vraie.

On en déduit ainsi un algorithme A décidant de la véracité de toute formule 2-SAT ψ . L'algorithme consiste à construire le graphe G à partir de la formule ψ en entrée, et à vérifier que toutes les paires de sommets x et \bar{x} sont bien dans des composantes fortement connexes différentes. A cette fin, considérer chacun des littéraux positifs et négatifs de ψ et explorer G par un parcours en profondeur d'abord (voir le chapitre 3.1.1) à partir de ce littéral.

A s'exécute bien en temps polynomial. En effet, si la formule implique n variables booléennes, alors construire G , ou plus précisément construire la matrice d'adjacence de G , se fait en temps au plus $O(n^2)$ puisque G a $2n$ sommets. Par ailleurs, A effectue au plus $2n$ parcours en profondeur. Comme le nombre d'arcs de G est linéaire en le nombre m de clauses (chaque clauses induit deux arcs), chaque parcours s'effectue en temps $O(m)$. La complexité totale de A est donc $O(n(n + m))$.

Correction de l'exercice 3

Le problème DIFFUSION SÛRE est dans NP. En effet, si l'on vous donne un ensemble de chemins, leur description prendra un espace que $O(n|S|)$ car chacun des $|S| - 1$ chemin aura une longueur au plus $n - 1$. Ce certificat est donc de taille polynomiale. De plus, vérifier si ces chemins passent bien par au plus k sommets dangereux se fait en temps polynomial en parcourant chacun des chemins et en comptant le nombre de sommets dangereux traversés. On peut donc vérifier en temps polynomial le certificat. On a donc DIFFUSION SÛRE \in NP.

Pour montrer que le problème DIFFUSION SÛRE est NP-difficile, nous construisons une réduction polynomiale de COUVERTURE ENSEMBLE vers DIFFUSION SÛRE.

Soit (U, \mathcal{S}, k) une instance de COUVERTURE ENSEMBLE, où $U = \{1, \dots, n\}$, $\mathcal{S} = \{S_1, \dots, S_m\}$, et $k \geq 0$. Nous construisons une instance (G, S, s, k') de DIFFUSION SÛRE comme suit. Le graphe G consiste en $n + m + 1$ sommets. Ces sommets sont dénotés $u_1, \dots, u_n, s_0, s_1, \dots, s_m$. On définit $S = \{s_0, u_1, \dots, u_n\}$ et $s = s_0$. Les sommets dangereux sont donc les sommets s_1, \dots, s_m . Les arcs du graphe G sont les suivants. Le sommet s_0 a un arc vers chacun des sommets s_1, \dots, s_m .

Pour $i = 1, \dots, m$, le sommet s_i a un arc vers le sommet u_j si et seulement si $j \in S_i$. Il n'y a pas d'autres arc dans G . Pour finir, $k' = k$.

Cette construction se fait en temps polynomial. En effet, d'une part la taille de l'instance (U, \mathcal{S}, k) est au moins $\Omega(n + m + \log k)$ bits. D'autre part, le graphe G construit est de taille $n' = n + m + 1$. Il peut donc être représenté par une matrice d'adjacence $n' \times n'$ de taille $O(n^2 + m^2)$ bits. L'ensemble S se décrit par une liste de $n + 1$ sommets, c'est-à-dire par $n + 1$ entiers de $\{1, \dots, n + m + 1\}$, pour une taille totale n'excédant pas $O(n \log(n + m))$ bits. La taille de l'instance de DIFFUSION SÛRE construite est donc polynomiale en la taille de l'instance de COUVERTURE ENSEMBLE donnée. Le temps pour construire l'instance de DIFFUSION SÛRE est ici directement proportionnel à sa taille. La construction de (G, S, s, k') se fait donc en un temps n'excédant pas $O(n^2 + m^2)$, donc polynomial en la taille de l'instance (U, \mathcal{S}, k) .

Il reste à vérifier qu'il existe une couverture de U par au plus k sous-ensembles de \mathcal{S} si et seulement si il est possible de trouver $|S| - 1$ chemins de s vers les sommets de $S \setminus \{s\}$ traversant au plus k sommets dangereux.

Supposons tout d'abord qu'il existe une couverture $S_{i_1}, \dots, S_{i_\ell}$ de U par $\ell \leq k$ sous-ensembles de \mathcal{S} . Le chemin de s_0 vers u_j est : $s_0 \rightarrow s_{i_t} \rightarrow u_j$ où i_t satisfait $j \in S_{i_t}$. L'existence de cet indice i_t est garantie par l'existence de la couverture : au moins un des ensembles $S_{i_1}, \dots, S_{i_\ell}$ doit contenir j . Ces chemins ne traversent qu'au plus ℓ sommets dangereux, et $\ell \leq k$.

Réciproquement, supposons qu'il est possible de trouver $|S| - 1$ chemins de s vers les sommets de $S \setminus \{s\}$ traversant $\ell \leq k$ sommets dangereux. Soit $s_{i_1}, \dots, s_{i_\ell}$ l'ensemble des sommets dangereux traversés par ces chemins. Un sommet $u_j \in S$ ne peut être atteint à partir de s qu'en allant de s vers un sommet dangereux s_i puis en allant de s_i à u_j . Donc, pour tout $j \in \{1, \dots, n\}$, si i_t est le sommet dangereux traversé par le chemin de s à u_j , on a une arc de s_{i_t} à u_j , et donc $j \in S_{i_t}$. Donc $S_{i_1}, \dots, S_{i_\ell}$ est une couverture de U , de taille au plus k .

DIFFUSION SÛRE \in NP et DIFFUSION SÛRE est NP-difficile, donc DIFFUSION SÛRE est NP-complet, c'est-à-dire DIFFUSION SÛRE \in NPC.

Correction de l'exercice 4

L'algorithme procède en enracinant l'arbre T en la source s de la diffusion, puis en remontant des feuilles vers la racine. Lors de la remontée, l'algorithme calcule le temps $b(T_u, u)$ de diffusion à partir de u dans le sous-arbre T_u de T enraciné en u . (T_u est obtenu en supprimant de T l'arête incidente à u en direction de s). Ce temps $b(T_u, u)$ est calculé à partir des temps $b(T_{v_i}, v_i)$ de diffusion à partir de chacun des $k = \deg(u)$ enfants de u dans T_u , dans leur sous-arbre respectif. Par un tri des temps $b(T_{v_i}, v_i)$, $i = 1, \dots, k$, on peut renommer les k enfants de u dans T_u de façon à ce que

$$b(T_{v_1}, v_1) \geq b(T_{v_2}, v_2) \geq \dots \geq b(T_{v_k}, v_k).$$

Il est facile de vérifier que l'équation suivante est satisfaite :

$$b(T_u, u) = \max_{i=1, \dots, k} (i + b(T_{v_i}, v_i)).$$

Elle correspond au protocole de diffusion dans lequel u informe ses enfants dans l'ordre décroissant de leur temps de diffusion. Initialement, le temps 0 est affecté à chaque feuille de T , et la programmation dynamique consiste à calculer le temps des sommets de l'arbre, de bas en haut, en suivant la méthode ci-dessus.

En chaque sommet u , il faut trier $\deg(u)$ entiers. La complexité de l'algorithme est donc $O(\sum_u (\deg(u) \cdot \log(\deg(u))))$. Or, pour tous sommet u , $\deg(u) \leq n - 1$. Par ailleurs, dans un

arbre, $\sum_u \deg(u) = 2(n - 1)$. La complexité de l'algorithme est donc $O(n \log n)$.

Correction de l'exercice 5

Ici, inutile de faire appel à la théorie des matroïdes. Considérons simplement l'algorithme consistant à trier les n objets par valeur décroissante de p_i/w_i . Renumérotions les objets afin que

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}.$$

On prend alors les k objets $1, \dots, k$ tels que $\sum_{i=1}^k w_i \leq W$, puis la fraction $(W - \sum_{i=1}^k w_i)/w_{k+1}$ du $(k + 1)$ ème objet. Dit autrement, ce choix revient à prendre une fraction α_i du i ème objet, avec $\alpha_1 = \dots = \alpha_k = 1$, $\alpha_{k+1} = (W - \sum_{i=1}^k w_i)/w_{k+1}$, et $\alpha_i = 0$ pour $i > k + 1$. Cet algorithme demande de trier les n objets, ce qui prend un temps $O(n \log n)$.

Montrons que cet algorithme retourne une solution optimale. La valeur de la solution retournée est $\sum_{i=1}^n \alpha_i p_i$. Considérons une solution optimale. Pour tout i , soit β_i la fraction de l'objet i dans cette solution. La solution optimale est donc de valeur $\text{OPT} = \sum_{i=1}^n \beta_i p_i$. Supposons qu'il existe a tel que $\beta_a < \alpha_a$. Soit alors a le plus petit indice de la sorte. Il existe donc $b > a$ tel que $\beta_b > \alpha_b$. Soit $\gamma = \min\{(1 - \beta_a)p_a, \beta_b p_b\}$, et considérons la solution obtenue à partir de l'optimale en affectant les fractions $\beta'_a = \beta_a + \frac{\gamma}{p_a}$ et $\beta'_b = \beta_b - \frac{\gamma}{p_b}$ aux objets a et b , respectivement. (Le choix de γ garantit que cette solution est valide.) Le poids de cette solution est

$$\sum_{i=1}^n \beta'_i w_i = (\sum_{i=1}^n \beta_i w_i) + \gamma(\frac{w_a}{p_a} - \frac{w_b}{p_b}) \leq \sum_{i=1}^n \beta_i w_i \leq W,$$

et son prix est

$$\sum_{i=1}^n \beta'_i p_i = (\sum_{i=1}^n \beta_i p_i) + \gamma(\frac{p_a}{p_a} - \frac{p_b}{p_b}) = \sum_{i=1}^n \beta_i p_i = \text{OPT}.$$

On a donc construit une solution de même valeur OPT , en permutant des objets. Pour montrer qu'on peut répéter cela et obtenir la solution construite par notre algorithme après une suite finie de permutations, on note que chaque permutation (a, b) résulte en une solution β' telle qu'ou bien $\beta'_a = 1$ ou bien $\beta'_b = 0$. Une telle permutation n'est donc jamais plus à reconsidérer. De proche en proche, on arrive en au plus n permutations à une solution optimale pour laquelle aucune permutation est possible. Cette solution est donc la solution α . Cette dernière est donc optimale.

Correction de l'exercice 6

Étant donnée une indice $i \in \{0, \dots, n\}$ et une contenance $c \in \{0, \dots, W\}$, on considère le sous-problème (i, c) consistant à résoudre le problème du sac à doc avec i objets, pour une borne supérieure de la taille du sac c . Une solution optimale de $\text{OPT}(i, c)$ peut être obtenue de deux façons possibles :

- à partir d'une solution optimale $\text{OPT}(i - 1, c)$ du problème à $i - 1$ variables, avec la même contenance c , à laquelle on ne rajoute pas l'objet i ;
- à partir d'une solution optimale $\text{OPT}(i - 1, c - w_i)$ du problème à $i - 1$ variables, avec la contenance $c - w_i$, à laquelle on ajoute l'objet i .

Le problème du sac à dos à zéro objet $OPT(0, c)$ a une solution optimale de valeur nulle pour tout c . De même, le problème du sac à dos à poids maximum 0, $OPT(i, 0)$ a une solution optimale de valeur nulle pour tout i . On en déduit donc le programme dynamique suivant.

```

pour  $i$  de 0 à  $n$  faire  $T[i, 0] \leftarrow 0$  ;
pour  $c$  de 0 à  $W$  faire  $T[0, c] \leftarrow 0$  ;
pour  $i$  de 1 à  $n$  faire
    pour  $c$  de 0 à  $W$  faire
        si  $w_i \leq c$  alors  $T[i, c] \leftarrow \max\{T[i - 1, c], p_i + T[i - 1, c - w_i]\}$ 
        sinon  $T[i, c] \leftarrow T[i - 1, c]$ 
```

Une fois la table T construite, la solution optimale est égale à $T[n, W]$. Pour trouver la suite d'objet mis dans le sac pour la solution optimale, il suffit de démarrer de la case de $T[n, W]$ et de déduire l'état des objets (mis dans le sac, ou non mis dans le sac) en remontant jusqu'à une case $T[0, \cdot]$.

Le temps d'exécution de cet algorithme est $O(nW)$. Attention, cet algorithme n'est donc pas polynomial. En effet, une instance du problème se décrit par n paires d'entiers (p_i, w_i) , plus la valeur de W . La taille d'une instance est donc

$$\Theta\left(\sum_{i=1}^n (\log p_i + \log w_i) + \log W\right).$$

Ainsi, $O(nW)$ peut être exponentiel en la taille de l'instance, en particulier pour des instances telles que, pour tout i , $p_i \leq W$ et $w_i \leq W$. On dit toutefois qu'un tel algorithme est *pseudo-polynomial* : il serait polynomial si on codait les entiers en unaire plutôt qu'en binaire.

Correction de l'exercice 7

Formellement, il s'agit de montrer que la version décisionnelle de SAC À DOS ci-dessous est NP-complet.

Le problème SAC À DOS (décisionnel) :

Entrée : un ensemble \mathcal{O} de n objets, de poids positifs w_1, \dots, w_n et de valeurs positives p_1, \dots, p_n , un entier $W \geq 0$, et un entier $k \geq 0$;

Question : existe-t-il un ensemble S d'objets dont la somme des poids est au plus W , et dont la valeur totale est au moins k ?

Nous allons montrer que SAC À DOS est NP-complet. Tout d'abord, montrons que SAC À DOS \in NP. Rappelons qu'appartenir à NP correspond à la capacité de *vérifier* une solution en temps polynomial. (L'appartenance à P correspond à la capacité de *calculer* une solution).

- Le certificat, ou, de manière équivalente, la preuve, est simplement la donnée d'une solution $J \subseteq \{1, \dots, n\}$ décrivant les ensembles sélectionnés dans le sac. Un tel certificat se code par un mot binaire J de n bits où $J[i] = 1$ si et seulement si le i ème objet est sélectionné. La taille de J est donc au plus polynomiale (en fait linéaire) en la taille $\Theta(\sum_{i=1}^n (\log p_i + \log w_i) + \log W) \geq \Omega(n + \log W)$ d'une instance de SAC À DOS.
- L'algorithme de vérification consiste effectuer les sommes $\sum_{i \in J} w_i$ et $\sum_{i \in J} p_i$ afin de vérifier que la première est au plus W , et la seconde au moins k . Cela peut se faire en temps $O(n)$ étapes RAM, donc en temps au plus polynomial en la taille d'une instance de SAC À DOS.

Pour montrer que SAC À DOC est NP-difficile, nous montrons qu'il existe une réduction polynomiale de COUVERTURE EXACTE PAR TRIPLETS à SAC À DOC. Soit (U, \mathcal{S}) une instance de COUVERTURE EXACTE PAR TRIPLETS. Sans perte de généralité, on peut supposer que $|U|$ est multiple de 3, et que $|\mathcal{S}| \geq |U|/3$ car sinon la réponse à COUVERTURE EXACTE PAR TRIPLETS est forcément négative. On peut également supposer sans perte de généralité que $U = \{1, \dots, 3n\}$. Nous construisons l'instance $f(U, \mathcal{S}) = (\mathcal{O}, W, k)$ de SAC À DOS de la façon suivante. Soit donc $U = \{1, \dots, 3n\}$ et $\mathcal{S} = \{S_1, \dots, S_r\}$ une instance de COUVERTURE EXACTE PAR TRIPLETS. Les objets du SAC À DOC correspondant sont les r triplets de \mathcal{S} . Le poids et la valeur de $S_i \in \mathcal{S}$ sont fixés à $\sum_{j \in S_i} (r+1)^j$. Enfin, on fixe $W = k = \sum_{j=1}^{3n} (r+1)^j$. Vérifions que la transformation f est une réduction polynomiale.

Pour vérifier que la construction de l'instance (\mathcal{O}, W, k) de SAC À DOS peut se faire en temps polynomial en la taille de l'instance (U, \mathcal{S}) , il suffit de vérifier que les valeurs et les poids sont bien de taille polynomial en la taille de l'instance (U, \mathcal{S}) . Cette taille est $\Theta(r \log n)$ car il faut coder r triplets d'entiers entre 1 et n . Elle est donc au moins r . Les valeurs et les poids des objets, ainsi que les valeurs de W et k sont tous au plus $\sum_{j=1}^{3n} (r+1)^j \simeq (r+1)^{3n}$. Cet entier se code sur $O(n \log r)$ bits, donc sur au plus $O(r \log r)$ bits, donc polynomial en la taille de l'instance (U, \mathcal{S}) .

Il reste plus qu'à vérifier que U admet une couverture exacte par \mathcal{S} si et seulement si on peut choisir un sous-ensemble d'objets de \mathcal{O} de poids total au plus W et de valeur totale au moins k . Supposons que U admet une couverture exacte par \mathcal{S} . Soit $I \subseteq \{1, \dots, r\}$ cette couverture, c'est-à-dire $U = \cup_{i \in I} S_i$ et $S_i \cap S_{i'} = \emptyset$ pour tout $i \neq i'$ de I . La valeur totale et le poids total de la solution correspondant à retenir les objets de I dans le sac-à-dos sont égaux à $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j$. Puisque $S_i \cap S_{i'} = \emptyset$ pour tout $i \neq i'$ de I , tout entier $j \in \{1, \dots, 3n\}$ n'apparaît qu'une fois au plus dans cette double somme. Egalement, puisque $U = \cup_{i \in I} S_i$, tout entier $j \in \{1, \dots, 3n\}$ apparaît une fois au moins dans cette double somme. Donc tout entier $j \in \{1, \dots, 3n\}$ apparaît exactement une fois dans cette double somme, dont la valeur est donc $\sum_{j=1}^{3n} (r+1)^j$. Il existe donc bien une solution de poids au plus W et valeur au moins k .

Réciproquement, supposons qu'on peut choisir un sous-ensemble d'objets de \mathcal{O} de poids total au plus W et de valeur totale au moins k . Soit $I \subseteq \{1, \dots, r\}$ ces objets. On a $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j \leq W$ et $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j \geq k$, donc $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j = \sum_{j=1}^{3n} (r+1)^j$. Soit $\alpha_j \geq 0$ la multiplicité de $j \in U$ dans $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j$, c'est-à-dire $\sum_{i \in I} \sum_{j \in S_i} (r+1)^j = \sum_{i=1}^r \alpha_j (r+1)^j$. On a $\sum_{i=1}^r \alpha_j (r+1)^j = \sum_{j=1}^{3n} (r+1)^j$. L'écriture d'un entier en base $r+1$ étant unique, on obtient $\alpha_j = 1$ pour tout $j \in U$, ce qui revient à dire que tout entier $j \in U$ apparaît une et une seule fois dans $\cup_{i \in I} S_i$. I est donc une couverture exacte de U . La transformation f est donc bien une réduction polynomiale.

Correction de l'exercice 8

Soit $G = (V, W, E)$ un graphe biparti. V et W sont les deux partie de la partitions des sommets. Soit $n = |V|$ et $m = |W|$. On construit le réseau de flot (G', c, s, t) comme suit. G' consiste en un sommet s reliés par n arc à n sommets v_1, \dots, v_n , et en un sommet t vers lequel pointent m arcs de m sommets w_1, \dots, w_m . Les sommets v_i et w_j sont reliés selon E de la façon suivante : il existe un arc de v_i à w_j si et seulement si $\{v_i, w_j\} \in E$. Chaque arc de G' a une capacité 1. Si on ne considère que des flots entiers, il est facile de montrer que G a un couplage de taille k si et seulement si G' a un flot de valeur k .

Correction de l'exercice 9

Une instance possible consiste en n ensembles réduits chacun à un unique éléments, et de coût $1/k$, $k = 1, \dots, n$, plus un $(n + 1)$ ème ensemble regroupant tous les éléments, et de coût $1 + \epsilon$.

Correction de l'exercice 10

Tout d'abord, montrons l'équivalence entre le problème ARBRE DE STEINER général, et sa restriction dans une métrique. Soit $G = (V, E)$, $S \subseteq V$, et $w : E \rightarrow \mathbb{Q}^+$ une instance quelconque de ARBRE DE STEINER. On définit $w'(\{u, v\})$ dans K_n comme le coût d'un chemin de coût total minimum entre u et v dans G . Notons que w' satisfait l'inégalité triangulaire. On définit également $S' = S$. On obtient ainsi l'instance (K_n, S, w') de ARBRE DE STEINER dans une métrique.

Montrons que les solutions des deux instances sont identiques. D'une part, on a $w'(e) \leq w(e)$ pour tout $e \in E$. Donc le coût d'une solution optimale T' de (K_n, w', S') est au plus le coût d'une solution optimale T de (G, w, S) . Réciproquement, soit T' un arbre de Steiner optimal pour (K_n, w', S') . En remplaçant chaque arête de T' par un chemin de coût minimum dans G entre les deux extrémités de l'arête, on obtient un sous-graphe H de G couvrant S , de poids total au plus celui de T' . On enlève des arêtes du sous-graphe H pour obtenir un arbre couvrant T de S dans G . Cet arbre est de poids total au plus celui de H , et donc au plus celui de T' .

Nous décrivons maintenant un algorithme polynomial de 2-approximation. Soit (G, S, w) une instance d'ARBRE DE STEINER, avec G graphe de n sommets. En premier, on transforme cette instance en une instance (K_n, S, w') comme ci-dessus. Cette transformation est polynomiale car chercher un plus court chemin entre deux sommets dans un graphe pondéré peut se faire en temps polynomial par l'algorithme de Dijkstra évoqué dans le chapitre 3.1.2. En fait, ici on a besoin d'un plus court chemin pour toutes les paires de sommets. Pour cela, il existe des algorithmes spécialisés qui évitent d'appeler $O(n^2)$ fois Dijkstra, par exemple l'algorithme de Floyd-Warshall. Comme on a vu que les valeurs des solutions optimales de (K_n, S, w') et de (G, S, w) sont identiques, et que l'on peut transformer en temps polynomial un arbre optimale d'une de ces instances en un arbre optimal de l'autre instance, il suffit de résoudre le problème (K_n, S, w') .

Pour résoudre (K_n, S, w') , on se restreint aux sommets de S , pour obtenir le sous-graphe $K_n[S]$ de K_n . On construit alors un arbre couvrant S dans $K_n[S]$, de poids minimum. Ceci peut se faire en temps polynomial, par Prim ou Kruskal. Notons T_S cet arbre. Soit T^* un arbre de Steiner optimal dans K_n . Comme pour le voyageur de commerce, on effectue un parcours DFS de T^* . Ce parcours a un poids au plus $2w'(T^*)$, donc au plus 2 OPT. Ensuite, toujours comme pour le voyageur de commerce, on utilise des raccourcis pour obtenir un chemin C passant uniquement par les sommets de S , et ne passant jamais deux fois par le même sommet. L'inégalité triangulaire satisfaite par la métrique assure que le poids $w'(C)$ de ce chemin est au plus celui du parcours DFS, et donc au plus 2 OPT. On note maintenant que ce chemin est un arbre couvrant de $K_n[S]$. Son poids est donc au moins égal au poids de l'arbre couvrant minimum T . Donc $w'(T_S) \leq w'(C) \leq 2 \text{OPT}$.

B Devoir à la maison

Considérons le problème du k -CENTRE tel que défini dans le chapitre 4.4.

Entrée : $K_n = (V, E)$, $w : E \rightarrow \mathbb{Q}^+$ satisfaisant l'inégalité triangulaire, et $k \geq 0$;

Objectif : Trouver un ensemble S de cardinalité k minimisant $\max_{v \in V} \text{dist}(v, S)$.

Dans un graphe $G = (V, E)$, un *ensemble dominant* est un sous-ensemble $D \subseteq V$ tel que tout sommet u de V satisfait $u \in D$ ou u a un voisin $v \in D$. Un ensemble *indépendant* est un sous-ensemble $I \subseteq V$ tel que si $u \in I$ alors aucun voisin v de u n'est dans I . Le *carré* de G est le graphe $G^2 = (V, F)$ tel que $\{u, v\} \in F$ si et seulement si $\{u, v\} \in E$ ou il existe un sommet $w \in V$ tel que $\{u, w\} \in E$ et $\{w, v\} \in E$.

Partie I. Algorithme d'approximation

Question I.1. Soit e_1, \dots, e_m les arêtes du graphe complet K_n triées de façon à ce que $w(e_1) \leq \dots \leq w(e_m)$. Soit G_i le graphe obtenu à partir de K_n en supprimant toutes les arêtes sauf e_1, \dots, e_i . Montrer que la valeur de la solution optimale de k -centre est égale à $w(e_i)$ où i est le plus petit indice tel que G_i ait un ensemble dominant D de cardinalité $\leq k$, et que D est l'ensemble S recherché.

La question 1 ne résout pas le problème k -CENTRE car calculer un ensemble dominant minimum est NP-difficile. On considère l'algorithme suivant, basé sur des ensembles indépendants maximaux (pas maximum) :

Algorithme A :

```

 $j \leftarrow 0$ ;
répéter
     $j \leftarrow j + 1$ ;
    construire  $H_j = G_j^2$ ;
    calculer un ensemble indépendant maximal  $I_j$  dans  $H_j$ ; (maximal, pas maximum)
jusqu'à  $|I_j| \leq k$ ;
Retourner  $I_j$ .

```

Question I.2. Montrer que si I est un ensemble indépendant maximal d'un graphe G , alors I est aussi un ensemble dominant. En déduire que l'algorithme A retourne un k -centre (non nécessairement optimal).

Question I.3. Soit I_j la solution retournée par A. Montrer que $\max_{v \in V} \text{dist}(v, I_j) \leq 2w(e_j)$.

Question I.4. Soit $G = (V, E)$ un graphe quelconque. Soit I un ensemble indépendant dans G^2 , et soit D un ensemble dominant minimum dans G . Montrer que $|I| \leq |D|$.

Question I.5. Soit I_j la solution retournée par A. Montrer que $\text{OPT} \geq w(e_j)$. En déduire que A est un algorithme polynomial de 2-approximation pour k -CENTRE.

Partie II. Inapproximabilité.

Question II.1. Formaliser le problème de décision NP-complet MINDOMSET associé au problème de la recherche d'un ensemble dominant de cardinalité minimum.

Question II.2. Proposer une transformation polynomiale de toute instance de MINDOMSET en une instance de k -CENTRE, qui construit un graphe complet dont les poids sur les arêtes valent 1 ou 2.

Question II.3. En utilisant la construction de la question précédente, montrer que, sauf si $P = NP$, il ne peut exister d'gorithme polynomial approximant la solution optimale de k -CENTRE à un facteur $2 - \epsilon$, pour tout $\epsilon > 0$ arbitrairement petit.

Correction du devoir

Question I.1. Soit i est le plus petit indice tel que G_i ait un ensemble dominant D de cardinalité $\leq k$. Tout d'abord, puisque G_i ait un ensemble dominant D de cardinalité $\leq k$, si l'on choisit D comme k -centre, alors $\max_{v \in V} \text{dist}(v, D) \leq w(e_i)$ puisque tous les sommets sont soit dans D , soit connectés directement par une arête de G_i avec un sommet de D , auquel cas le poids de cette arête est $\leq w(e_i)$. Donc $\text{OPT} \leq w(e_i)$. Réciproquement, pour $j < i$, tout ensemble dominant de G_j a cardinalité $> k$. Donc, pour tout ensemble S de k sommets, au moins un sommet v ne sera pas dominé par S dans G_j . Donc v est connecté à S par une arête de poids $\geq w(e_i)$ dans K_n . Donc $\text{OPT} \geq w(e_i)$.

Question I.2. Soit I un ensemble indépendant maximal d'un graphe G . Soit $u \notin I$ un sommet non dominé par I , c'est-à-dire dont aucun voisin est dans I . Alors $I \cup \{u\}$ est un ensemble indépendant, contredisant la maximalité de I .

Question I.3. I_j est un ensemble indépendant maximal de H_j . D'après la question précédente, c'est donc un ensemble dominant de H_j . En conséquence, tout sommet v est connecté dans H_j à un sommet de I_j par une arête. Comme $H_j = G_j^2$, on obtient que tout sommet v est connecté dans G_j à un sommet de I_j par un chemin de longueur au plus 2. Puisque les arêtes de G_j ont un poids au plus $w(e_j)$, on en déduit que ce chemin à un poids au plus $2w(e_j)$.

Question I.4. Soit $G = (V, E)$ un graphe quelconque. Soit I un ensemble indépendant dans G^2 , et soit D un ensemble dominant dans G . Soit $u \in D$, et soit $\Gamma(u)$ les voisins de u dans G . Dans G^2 , les sommets de $\Gamma[u] = \Gamma(u) \cup \{u\}$ forment une clique C_u . Comme D est dominant, tout sommet $v \notin D$ a au moins un voisin $u \in D$, et donc v est un sommet d'au moins une clique C_u dans G^2 . L'union des cliques C_u , $u \in D$ couvre donc tous les sommets de G^2 . Chacune de ces $|D|$ cliques ne peut contenir qu'au plus un sommet d'un ensemble indépendant, donc $|I| \leq |D|$.

Question I.5. Soit I_j la solution retournée par A . Pour tout $i < j$, $|I_i| > k$. Donc, d'après la question précédente, pour tout $i < j$, tout ensemble dominant D_i de G_i vérifie $|D_i| > k$. Donc $j \leq i$ où i est le plus petit indice tel que G_i ait un ensemble dominant D de cardinalité $\leq k$. D'après la question I.1, $\text{OPT} = w(e_i) \geq w(e_j)$. D'après la question II.3, $\max_{v \in V} \text{dist}(v, I_j) \leq 2w(e_j) \leq 2\text{OPT}$. Donc A est un algorithme de 2-approximation pour k -CENTRE. A est polynomial car calculer G_i^2 correspond à prendre le carré de la matrice d'incidence de G_i , et calculer un ensemble indépendant maximal est polynomial par un algorithme glouton.

Partie II. Inapproximabilité.

Question II.1.

Le problème ENSEMBLE DOMINANT MINIMUM (MINDOMSET) :

Entrée : un graphe $G = (V, E)$, un entier $k \geq 0$;

Question : existe-t-il un ensemble $D \subseteq V$ tel que $\forall u \notin D, \exists v \in D \mid \{u, v\} \in E$, et $|D| \leq k$?

Question II.2. Soit $G = (V, E)$ un graphe quelconque de n -sommet. On construit une instance (K_n, w) de k -CENTRE comme suit : pour tout arête e de K_n , $w(e) = 1$ si $e \in E$, et $w(e) = 2$ sinon. Cette transformation est trivialement polynomiale puisque K_n a $O(n^2)$ arêtes, ce qui est au plus le carré de la taille du graphe G .

Question II.3. Si G a un ensemble dominant de taille $\leq k$ alors la solution optimale pour l'instance de k -CENTRE construite est de valeur 1. En revanche, si G n'a pas d'ensemble dominant de taille $\leq k$ alors la solution optimale pour l'instance de k -CENTRE construite est de valeur 2. Soit A un algorithme de facteur d'approximation $2 - \epsilon$. Si G a un ensemble dominant de taille $\leq k$ alors A retourne une valeur au plus $(2 - \epsilon)\text{OPT}$, et donc au plus $2 - \epsilon$. En revanche, si G n'a pas d'ensemble dominant de taille $\leq k$ alors A retourne au moins OPT , donc au moins 2. A serait donc capable de distinguer si G a un ensemble dominant de taille $\leq k$ ou pas. Donc, sauf si $P = NP$, l'algorithme A ne peut pas s'exécuter en temps polynomial.

C Examen du 29 janvier 2010, 11h30-13h00

Exercice I. Schéma d'approximation polynomial

Considérons le problème ADDMAX défini comme suit :

Données : un ensemble $\mathcal{E} = \{x_1, \dots, x_n\}$ de n entiers positifs, et un entier $t \geq 0$;

Objectif : maximiser $\sum_{i \in I} x_i$ parmi tous les sous-ensembles d'indices $I \subseteq \{1, \dots, n\}$ tels que $\sum_{i \in I} x_i \leq t$.

L'exercice a pour but de construire un schéma d'approximation polynomial pour le problème ADDMAX. Sans perte de généralité, on pourra supposer l'ensemble \mathcal{E} initialement trié par ordre croissant des entiers, de façon à ce que $x_i \leq x_{i+1}$ pour tout $i = 1, \dots, n - 1$.

Etant donnés deux sous-ensembles A et B d'un ensemble E , on note $A \setminus B$ l'ensemble

$$\{x \in E / x \in A \text{ et } x \notin B\}.$$

Pour $\alpha > 1$, on note $\log_\alpha x$ le logarithme en base α de x , i.e., $\log_\alpha x = \ln x / \ln \alpha$.

Question I.1. Combien y a-t-il de sous ensembles $I \subseteq \{1, \dots, n\}$ distincts ? En déduire un algorithme exponentiel résolvant le problème ADDMAX.

Question I.2. Cette question conduit à développer un algorithme exponentiel plus sophistiqué que celui de la question 1, dans un but d'analyse du schéma d'approximation à construire dans les questions suivantes. Pour un ensemble d'entiers $Y = \{y_1, \dots, y_m\}$ et un entier x , on note $Y + x$ l'ensemble $\{y_1 + x, \dots, y_m + x\}$. On considère l'algorithme \mathcal{A} suivant.

```

 $E_0 \leftarrow \{0\};$ 
pour  $i = 1$  à  $n$  faire
     $E_i \leftarrow \text{fusionner}(E_{i-1}, E_{i-1} + x_i);$ 
    supprimer de  $E_i$  tout entier  $> t$ ;
    retourner l'entier  $y$  le plus grand de  $E_n$ .

```

Montrer que \mathcal{A} retourne la solution optimale y^* du problème ADDMAX. Pourquoi \mathcal{A} est-il exponentiel en temps ?

Question I.3. Soit $\delta \in]0, 1[$ un réel quelconque. On dit que \mathcal{E}' est un δ -seuillage de \mathcal{E} si pour tout élément $y \in \mathcal{E} \setminus \mathcal{E}'$, il existe $\hat{x} \in \mathcal{E}'$ tel que $(1 - \delta)y \leq \hat{x} \leq y$. Décrire une procédure **seuiller**(\mathcal{E}, δ) qui, en temps polynomial en n , produit un δ -seuillage \mathcal{E}' de \mathcal{E} de petite taille, c'est-à-dire tel que, pour tout couple d'entiers x et x' de \mathcal{E}' avec $x \leq x'$, on ait $x \neq x'$ et $x/x' < (1 - \delta)$.

Question I.4. Soit $\mathcal{E}' = \text{seuiller}(\mathcal{E}, \delta)$ un δ -seuillage de \mathcal{E} vérifiant les conditions de la question précédente. Montrer que \mathcal{E}' ne contient pas plus de $1 + \log_{1/(1-\delta)} t$ éléments.

Question I.5. Soit $\epsilon \in]0, 1[$ un réel quelconque. On considère l'algorithme \mathcal{A}_ϵ suivant.

```

 $E'_0 \leftarrow \{0\};$ 
pour  $i = 1$  à  $n$  faire
   $E'_i \leftarrow \text{fusionner}(E'_{i-1}, E'_{i-1} + x_i);$ 
   $E'_i \leftarrow \text{seuiller}(E'_i, \epsilon/n);$ 
  supprimer de  $E'_i$  tout entier  $> t$ ;
retourner l'entier le plus grand de  $E'_n$ .

```

Nous allons comparer les ensembles E_i construits en question 2 avec les ensembles E'_i construits par l'algorithme ci-dessus. Montrer par récurrence sur i que pour tout $x \in E_i$, il existe un $x' \in E'_i$ tel que $(1 - \epsilon/n)^i x \leq x' \leq x$.

Question I.6. En utilisant le fait que $f(x) = (1 - \epsilon/x)^x$ est une fonction croissante de $x \geq 1$, déduire de la question 5 que \mathcal{A}_ϵ est retourne un $(1 - \epsilon)$ -approximation de la solution optimale de ADDMAX.

Question I.7. En utilisant le fait que $\ln(1 - x) < -x$ pour $x \in]0, 1[$, déduire de la question 4 que l'algorithme \mathcal{A}_ϵ s'exécute en temps polynomial en la taille des données (PTAS).

Question I.8. Le schéma construit ci-dessus est-il complètement polynomial (FPTAS) ?

Exercice II. Théorie de la complexité

Question II.1. Si l'on cherche à vous vendre un logiciel capable de dire si un programme quelconque termine, est-ce crédible ?

Question II.2. Afin de réaliser un sondage, vous souhaitez trouver dans une population de n individus, par exemple les n élèves de Centrale, un sous-ensemble d'individus le plus grand possible tel que deux individus quelconques de ce sous-ensemble ne se connaissent pas personnellement. On cherche à vous vendre un logiciel qui prétend résoudre ce problème en une seconde. Est-ce crédible ?

Correction de l'examen

Question I.1. Il y a 2^n sous ensembles $I \subseteq \{1, \dots, n\}$ distincts, un pour chaque mot binaire dans $\{0, 1\}^n$. On peut résoudre ADDMAX par un algorithme exponentiel qui passe en revu chacun de ces 2^n sous-ensembles I , teste pour chacun d'eux si $\sum_{i \in I} x_i \leq t$, et, parmi ceux-là, prend le maximum des sommes considérées.

Question I.2. Pour tout $i \geq 1$, l'ensemble E_i contient toutes les sommes inférieures ou égales à t pouvant être obtenues à partir des i premiers éléments de \mathcal{E} (référence triviale si i). Ainsi E_n contient toutes les sommes inférieures ou égales à t pouvant être obtenues à partir de tous les éléments de \mathcal{E} . Et donc la plus grande d'entre elles est la solution cherchée.

\mathcal{A} est exponentiel en temps car il peut être amené à traiter $\min\{t, 2^n\}$ sommes distinctes, et ces deux termes sont exponentiels en la taille des données (sauf à considérer que t soit codé en unaire sur t bits, plutôt que d'être codé de manière classique en binaire sur $\lceil \log_2 t \rceil$ bits.)

Question I.3. On procède comme suit. On suppose que les éléments de \mathcal{E} sont triés. On place x_1 dans \mathcal{E}' , puis on passe en revue les n éléments de \mathcal{E} un par un, de x_2 à x_n . Lorsque l'on considère x_i , on le compare au dernier entier \hat{x} placé dans \mathcal{E}' . (On a $\hat{x} \leq x_i$ puisque les x_i sont triés.) Si $\hat{x} < (1 - \delta)x_i$ alors on place x_i dans \mathcal{E}' . Par construction, \mathcal{E}' est un δ -seuillage de \mathcal{E} .

L'algorithme ci-dessus satisfait bien que pour tout couple d'entiers (x, x') de E' on a $x \neq x'$. D'autre part, si $x \leq x'$ et \hat{x} dénote le dernier élément de \mathcal{E}' entré juste avant d'entrer x' dans \mathcal{E}' , alors $x \leq \hat{x} < (1 - \delta)x'$. D'où $x/x' < (1 - \delta)$.

Question I.4. Soit $\mathcal{E}' = \{y_1, \dots, y_m\}$. D'après la question précédente, on a $y_i/y_{i+1} < (1 - \delta)$ pour tout $i = 1, \dots, m - 1$. D'où $y_1/y_m < (1 - \delta)^{m-1}$. Dit autrement, $y_m/y_1 > 1/(1 - \delta)^{m-1}$. Comme $y_m \leq t$ et $y_1 \geq 1$, on obtient $t > 1/(1 - \delta)^{m-1}$. D'où $m - 1 < \log_{1/(1-\delta)} t$.

Question I.5. La propriété est trivialement vrai pour $i = 0$. Supposons la propriété vraie pour $i - 1$ et montrons la pour i . Fixons donc $x \in E_i$.

Si $x \in E_{i-1}$ alors, par induction, il existe $x' \in E'_{i-1}$ tel que $(1 - \epsilon/n)^{i-1} x \leq x' \leq x$. Par la propriété de seuillage, il existe $x'' \in E'_i$ tel que $(1 - \epsilon/n) x' \leq x'' \leq x'$. On en déduit alors $(1 - \epsilon/n)^i x \leq x'' \leq x$.

Si $x \notin E_{i-1}$ alors $x = y + x_i$ où $y \in E_{i-1}$. Par induction, il existe $y' \in E'_{i-1}$ tel que $(1 - \epsilon/n)^{i-1} y \leq y' \leq y$. On en déduit, $(1 - \epsilon/n)^{i-1} (y + x_i) \leq y' + x_i \leq y + x_i$. Par la propriété de seuillage, il existe $y'' \in E'_i$ tel que $(1 - \epsilon/n) (y' + x_i) \leq y'' \leq y' + x_i$ d'où l'on déduit $(1 - \epsilon/n)^i (y + x_i) \leq y'' \leq y + x_i$.

Question I.6. D'après la question précédente, la solution optimale, c'est-à-dire l'entier x^* le plus grand de E_n , est approximé par un entier x' de E'_n satisfaisant $(1 - \epsilon/n)^n x^* \leq x' \leq x^*$. Comme $f(n) = (1 - \epsilon/n)^n$ est une fonction croissante de $n \geq 1$, on en déduit que $(1 - \epsilon) x^* \leq x' \leq x^*$. Donc $1 - \epsilon \leq \frac{x'}{x^*} \leq 1$, et donc \mathcal{A}_ϵ calcule une $(1 - \epsilon)$ -approximation de la solution optimale de ADDMAX.

Question I.7. L'algorithme \mathcal{A}_ϵ procède en n itérations. D'après la question I.4, la taille des ensembles \mathcal{E}'_i considérés à chaque itération est $O(\ln(t)/\ln(1/(1 - \epsilon/n))) = O(-\ln(t)/\ln(1 - \epsilon/n))$. Comme $\ln(1 - x) < -x$ pour $x \in]0, 1[$, on obtient que la taille de ces ensembles est $O(\ln(t)/(\epsilon/n)) = O(\frac{1}{\epsilon} n \ln(t))$. Les opérations de fusion et de seuillage s'effectuent en temps linéaire en fonction de la taille des ensembles considérés. Ainsi, chaque boucle s'effectue en temps $O(\frac{1}{\epsilon} n \ln(t))$, ce qui donne un temps total $O(\frac{1}{\epsilon} n^2 \ln(t))$. La taille de l'entrée est au moins $\Omega(n + \ln t)$. La complexité de \mathcal{A}_ϵ est donc polynomiale en la taille des entrées et en $\frac{1}{\epsilon}$.

Question I.8. Le schéma est complètement polynomial car il est polynomial en $1/\epsilon$.

Exercice II. Théorie de la complexité

Question II.1. Cela n'est pas crédible car cela contredit la non décidabilité de l'arrêt de la machine de Turing. Si l'on accepte la thèse de Church énonçant que la notion d'algorithme est bien capturée par la notion de machine de Turing, il ne peut y avoir d'algorithme capable de décider si une machine de Turing s'arrête ou pas. Autrement dit, il ne peut pas y avoir d'algorithme décidant si un programme quelconque s'arrête. Ainsi, si le logiciel que l'on cherche à vous vendre fonctionne correctement, cela remettrait en cause la thèse de Church.

Question II.2. Trouver dans une population de n individus, un sous-ensemble d'individus le plus grand possible tel que deux individus quelconques de ce sous-ensemble ne se connaissent pas personnellement revient à calculer un ensemble indépendant maximum dans le graphe défini comme suit : il y a une arête entre deux individus x et y si et seulement si ceux-ci se connaissent personnellement. Or calculer un ensemble indépendant maximum est un problème NP-difficile. En conséquence, sauf si P=NP, il n'existe pas d'algorithme polynomial résolvant ce problème. Un temps d'exécution de 1 seconde est donc peu crédible dès que n est grand (comme par exemple le nombre d'étudiants de Centrale).

D Examen du 21 janvier 2011, 09h30-11h00

L'examen traite principalement du même problème algorithmique, mais certaines questions peuvent être sautées si l'on peine à les résoudre. Chaque question ne demande une réponse que de quelques lignes.

Rappelons que le problème COUVERTURE SOMMET consiste à, étant donné un graphe quelconque $G = (V, E)$, trouver un ensemble $C \subseteq V$ de cardinalité minimum tel que toute arête $e \in E$ a au moins une de ses deux extrémités dans C .

Pour deux ensembles A et B , on note $A \setminus B = \{x \in A \mid x \notin B\}$.

Question 1. Donner une formulation de COUVERTURE SOMMET sous la forme d'un programme linéaire en nombres entiers (PLNE) \mathcal{P} .

Réponse : On définit le PLNE \mathcal{P} suivant :

$$\begin{aligned} & \text{minimiser} && \sum_{v \in V} x_v \\ & \text{sous les contraintes} && x_u + x_v \geq 1 \text{ pour toute arête } \{u, v\} \in E \\ & && x_v \in \{0, 1\} \text{ pour tout sommet } v \in V \end{aligned}$$

Les variables $x_v \in \{0, 1\}$ sont interprétées comme suit :

$$x_v = 1 \iff v \in C.$$

La contrainte $x_u + x_v \geq 1$ force chaque arête $\{u, v\}$ à être couverte. Minimiser $\sum_{v \in V} x_v$ revient bien à minimiser la cardinalité de la couverture.

Question 2. Si $n = |V|$ et $m = |E|$, combien de variables utilisez-vous ? Combien de contraintes ?

Réponse : \mathcal{P} contient n variables et $m + n$ contraintes.

Question 3. Donner une formulation relaxée \mathcal{R} de votre programme linéaire en nombres entiers et expliquer pourquoi il est possible de résoudre ce programme relaxé en temps polynomial.

Réponse : On relaxe \mathcal{P} en relaxant la contrainte d'intégralité :

$$\begin{aligned} \text{minimiser} \quad & \sum_{v \in V} x_v \\ \text{sous les contraintes} \quad & x_u + x_v \geq 1 \text{ pour toute arête } \{u, v\} \in E \\ & x_v \geq 0 \text{ pour tout sommet } v \in V \end{aligned}$$

Il est possible de résoudre ce programme relaxé en temps polynomial en la taille de l'instance G car le nombre de variables et le nombre de contraintes sont polynomiaux en la taille de G .

Question 4. Soit $\{x_v^*, v \in V\}$ une solution optimale du programme linéaire \mathcal{R} . On définit les ensembles suivants :

$$\begin{aligned} S^+ &= \{v \in V \mid x_v^* > \frac{1}{2}\} \\ S^= &= \{v \in V \mid x_v^* = \frac{1}{2}\} \\ S^- &= \{v \in V \mid x_v^* < \frac{1}{2}\} \end{aligned}$$

Soit C une couverture-sommet de $G = (V, E)$. Montrer que $C' = S^+ \cup (C \setminus S^-)$ est une couverture-sommet de G .

Réponse : C étant privé de S^- , il suffit de vérifier que toutes les arêtes ayant une extrémité dans S^- sont bien couvertes. Soit $e = \{u, v\}$ avec $u \in S^-$. On a $v \notin S^+ \cup S^-$ car $x_u^* + x_v^* \geq 1$. Donc $v \in S^+$. Il suit que e est bien couverte dans C' car $S^+ \subseteq C'$.

Question 5. Montrer que $\{y_v, v \in V\}$ où

$$y_v = \begin{cases} 1 & \text{si } v \in S^+ \cup S^= \\ 0 & \text{si } v \in S^- \end{cases}$$

est une solution admissible pour \mathcal{P} dont la valeur $\sum_{v \in V} y_v$ est au plus deux fois celle de la solution optimale $\text{OPT} = \sum_{v \in V} x_v^*$ de \mathcal{R} .

T.S.V.P.

Réponse : Dans la solution $\{y_v, v \in V\}$, seules les arêtes $e = \{u, v\}$ entre deux sommets de S^- ne sont pas couvertes. Mais, nous avons vu qu'il n'existe pas de telles arêtes (puisque $x_u^* + x_v^* \geq 1$). Donc $\{y_v, v \in V\}$ est une solution admissible pour \mathcal{R} . On a

$$\text{OPT} = \sum_{v \in V} x_v^* \geq \sum_{v \in S^+ \cup S^=} x_v^* \geq \frac{1}{2} (|S^+| + |S^=|) = \frac{1}{2} \sum_{v \in V} y_v .$$

Donc $\sum_{v \in V} y_v \leq 2\text{OPT}$.

Question 6. Concevoir le squelette d'un algorithme polynomial de 2-approximation pour COUVERTURE SOMMET utilisant la programmation linéaire et la méthode de l'arrondi.

Réponse : L'algorithme consiste simplement en résoudre le programme linéaire relaxé \mathcal{R} , puis à arrondir la solution trouvée $\{x_v^*, v \in V\}$ selon la règle : $y_v = 1 \iff x_v^* \geq \frac{1}{2}$. C'est une solution admissible pour \mathcal{P} dont la valeur est au plus 2OPT où OPT dénote la valeur de la solution optimale de \mathcal{R} . Cette dernière est au plus la valeur d'une solution optimale de \mathcal{P} .

Question 7. Soit $\epsilon = \min_{v \in S^+} (x_v^* - \frac{1}{2})$ et soit $(x'_v)_{v \in V}$ défini par $x'_v = x_v^* - \epsilon$ si $v \in S^+ \setminus C$, $x'_v = x_v^* + \epsilon$ si $v \in C \cap S^-$, et $x'_v = x_v^*$ sinon. Montrez que $(x'_v)_{v \in V}$ est une solution admissible pour \mathcal{R} .

Réponse : Une arête e ne voit sa contrainte diminuer que si l'une de ses extrémités v est dans $S^+ \setminus C$, mais on préserve $x'_v \geq \frac{1}{2}$ pour tout $v \in S^+$. Donc, si l'autre extrémité u de e est dans $S^+ \cup S^-$, la contrainte $x'_u + x'_v \geq 1$ reste satisfaite. Si $u \in S^-$ alors on note que, puisque $v \notin C$, on a nécessairement $u \in C$, et donc $u \in S^- \cap C$. Il en découle que

$$x'_v + x'_u = (x_v^* - \epsilon) + (x_u^* + \epsilon) = x_v^* + x_u^* \geq 1 .$$

Question 8. En comparant $\sum_{v \in V} x'_v$ et $\sum_{v \in V} x_v^*$, déduire de la question précédente que $|S^+ \setminus C| \leq |C \cap S^-|$.

Réponse : Considérons la solution admissible $(x'_v)_{v \in V}$ définie dans la question 7. On a

$$\begin{aligned} \sum_{v \in V} x'_v &= \sum_{v \in S^+ \setminus C} x'_v + \sum_{v \in C \cap S^-} x'_v + \sum_{v \in V \setminus ((S^+ \setminus C) \cup (C \cap S^-))} x'_v \\ &= \sum_{v \in V} x_v^* + \epsilon \left(|C \cap S^-| - |S^+ \setminus C| \right) \end{aligned}$$

Or $\{x_v^*, v \in V\}$ est optimal, donc $\sum_{v \in V} x'_v \geq \sum_{v \in V} x_v^*$. Donc $|C \cap S^-| - |S^+ \setminus C| \geq 0$.

Question 9. Déduire des questions précédentes qu'il existe une solution optimale C^* de COUVERTURE SOMMET pour $G = (V, E)$ telle que $S^+ \subseteq C^*$ et $C^* \cap S^- = \emptyset$.

Réponse : Soit C une couverture-sommet de $G = (V, E)$, de cardinalité minimum. Soit $C^* = S^+ \cup (C \setminus S^-)$. D'après la question 4, C^* est une couverture-sommet de G . D'après la question 8, $|S^+ \setminus C| \leq |C \cap S^-|$, donc $|C^*| \leq |C|$, et donc C^* une couverture-sommet de $G = (V, E)$, de cardinalité minimum. Par construction, on a bien $S^+ \subseteq C^*$ et $C^* \cap S^- = \emptyset$.

Question 10. On considère maintenant la version paramétrée de COUVERTURE SOMMET :

données : un graphe $G = (V, E)$;

paramètre : un entier $k \geq 0$;

question : existe-t-il une couverture-sommet C de G telle que $|C| \leq k$?

En considérant le programme linéaire relaxé, montrer que si la réponse à COUVERTURE SOMMET de paramètre k est positive alors $|S^+| \leq k$.

Réponse : Cela découle directement de la question précédente puisqu'il existe une solution optimale C^* telle que $S^+ \subseteq C^*$, car alors $|S^+| \leq |C^*|$, et $|C^*| \leq k$ si la réponse à COUVERTURE SOMMET de paramètre k est positive .

Question 11. Montrer que si la réponse à COUVERTURE SOMMET de paramètre k est positive alors $|S^-| \leq 2k$.

Réponse : Si la réponse à COUVERTURE SOMMET de paramètre k est positive, alors, en particulier, la valeur optimale du programme linéaire relaxé est au plus k . Donc $\sum_{v \in V} x_v^* \leq k$. Donc $\sum_{v \in S^-} x_v^* \leq k$. Et donc $\frac{|S^-|}{2} \leq k$.

Question 12. En déduire un noyau de COUVERTURE SOMMET paramétré.

Réponse : D'après les questions précédentes, il existe une couverture sommet C^* optimale telle que $S^+ \subseteq C^*$ et $C^* \cap S^- = \emptyset$. On se restreint donc à vérifier l'existence d'une telle couverture, de taille au plus k . On peut donc se restreindre au graphe $G' = G \setminus (S^+ \cup S^-)$ obtenu en supprimant de G tous les sommets de S^+ et tous les sommets de S^- . (On peut définir également $G' = G[S^=]$ comme le sous-graphe de G induit par les sommets de $G^=$). G a une couverture-sommet de taille au plus k si et seulement si G' a une couverture-sommet de taille au plus $k - |S^+|$. En effet, les sommets de S^+ seront placés dans la couverture et les sommets de S^- ne sont pas dans la couverture, mais leurs arêtes sont couvertes par les sommets de S^+ . D'après la question 11, G' n'a qu'au plus $2k$ sommets. On est donc ramené à décider s'il existe une couverture-sommet de taille au plus $h = k - |S^+|$ dans un graphe de taille au plus $2k$. Comme $h \leq k$, ce problème ne dépend plus que de k , et est indépendant de la taille n du graphe initial. C'est donc un noyau. On résout le noyau en testant chacune des $\binom{2k}{h}$ possibilités de couverture.

— Fin —

E Examen du 29 octobre 2011, 09h00-11h00

L'examen est découpé en trois problèmes indépendants.

Une clique dans un graphe $G = (V, E)$ est un sous-ensemble de sommets $K \subseteq V$ tel que, pour toute paire de sommets distincts $(u, v) \in K \times K$, on a $\{u, v\} \in E$ (il y a une arête entre u et v dans G). On définit le problème CLIQUEMAX comme suit :

CLIQUE MAXIMUM (CLIQUEMAX) :

Donnée : un graphe $G = (V, E)$, un entier $k \geq 0$;

Question : G possède-t-il une clique K telle que $|K| \geq k$?

— Problème I —

Ce problème a pour objet d'étudier la difficulté du problème CLIQUEMAX. Soit

$$\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$$

une formule booléenne en forme normale conjonctive, impliquant n variables et m clauses de trois littéraux chacune. Soit G_ϕ le graphe de $3m$ sommets obtenu de la façon suivante. Chaque littéral ℓ de chaque clause C_i induit un sommet $u_{i,\ell}$. Il y une arête dans G_ϕ entre $u_{i,\ell}$ et $u_{j,\ell'}$ si et seulement si $i \neq j$ et $\ell' \neq \bar{\ell}$.

Question I.1. Supposons que ϕ est satisfiable, et considérons une affectation des n variables permettant de satisfaire ϕ . De façon arbitraire, choisir dans chaque clause C_i un littéral ℓ vrai pour cette affectation. Ce choix revient à choisir m sommets de G_ϕ . Montrer que ces m sommets forment une clique K .

Réponse : Soit $\{u_{i_j, \ell_j}, j = 1, \dots, m\}$ les m sommets choisis. Soit $j, j' \in \{1, \dots, m\}$, avec $j \neq j'$. Puisqu'on a choisi un sommet par clause, on obtient $i_j \neq i_{j'}$. Par ailleurs, puisque ℓ_j et $\ell_{j'}$ sont vraies, on a $\ell_j \neq \bar{\ell}_{j'}$. De ces deux inégalités découle l'existence d'une arête entre les sommets u_{i_j, ℓ_j} et $u_{i_{j'}, \ell_{j'}}$ dans G_ϕ . L'ensemble $\{u_{i_j, \ell_j}, j = 1, \dots, m\}$ forme donc une clique K dans G_ϕ .

Question I.2. Supposons que G_ϕ possède une clique de taille m . Montrer que ϕ est satisfiable.

Réponse : Soit $\{u_{i_j, \ell_j}, j = 1, \dots, m\}$ les m sommets de la clique. On affecte les variables de ϕ de façon à ce que les littéraux ℓ_j soient tous vrais, pour $j = 1, \dots, m$. Cela est possible car l'existence d'une arête entre u_{i_j, ℓ_j} et $u_{i_{j'}, \ell_{j'}}$ garantit que $\ell_j \neq \bar{\ell}_{j'}$ pour tout $j \neq j'$, et garantit donc l'absence d'une situation du type $\ell_j = x$ et $\ell_{j'} = \bar{x}$, qui forme le seul scénario pour lequel on ne pourrait assigner ℓ_j et $\ell_{j'}$ à vrai simultanément. Cette affectation des variables satisfait ϕ car chaque clause est satisfaite du fait que l'existence d'une arête entre u_{i_j, ℓ_j} et $u_{i_{j'}, \ell_{j'}}$ garantit que $i_j \neq i_{j'}$, et donc chaque clause contient exactement un littéral correspondant à un sommet de la clique.

Question I.3. En déduire que CLIQUEMAX est NP-complet.

Réponse : La construction du graphe G_ϕ à partir d'une formule ϕ est une réduction de 3SAT vers CLIQUEMAX. En effet, les deux questions précédentes montrent qu'une formule ϕ de m clauses est satisfiable si et seulement si G_ϕ possède une clique de taille m . Cette réduction est polynomiale car la construction de G_ϕ peut s'effectuer en temps $O(m^2)$. 3SAT étant NP-difficile, on en déduit que CLIQUEMAX est NP-difficile. Par ailleurs, CLIQUEMAX est dans NP avec comme certificat la clique de taille k elle-même. La vérification qu'un graphe de taille k est une clique peut en effet s'effectuer en temps $O(k^2)$.

— Problème II —

Nous avons vu dans le problème précédent que le problème CLIQUEMAX est NP-complet. En fait, il est connu que, pour tout $\epsilon > 0$, ce problème n'est pas approximable en temps polynomial à un facteur multiplicatif $n^{1-\epsilon}$, sauf si P = NP.

Question II.1. Quel est le facteur d'approximation de l'algorithme trivial consistant à choisir un sommet de façon arbitraire et à retourner la clique formée par cet unique sommet ?

Réponse : Une telle clique a une taille 1. Comme la taille maximum OPT d'une clique dans un graphe de n sommets est au plus n , on en déduit que le facteur d'approximation $\max_G \text{OPT}(G)/\text{trivial}(G)$ de l'algorithme trivial est au plus n . Par ailleurs, la clique maximum dans un graphe complet de n sommets est de taille n . On obtient donc que le facteur d'approximation de l'algorithme trivial est exactement n .

Question II.2. La suite du problème a pour objet de concevoir un algorithme d'approximation modeste, mais toutefois meilleur que l'algorithme trivial de la question précédente. A cette fin, considérons l'algorithme \mathcal{A}_k en trois phases, paramétré par un entier k :

1. Décomposer l'ensemble des sommets V en k sous-ensembles B_1, \dots, B_k de même cardinalité (à 1 près), appelés *blocs* ;
2. Pour chaque bloc B_i trouver une clique K_i de cardinalité maximum dans $G[B_i]$, où $G[B_i]$ dénote le sous-graphe de G engendré par les sommets de B_i ;
3. Retourner la plus grande des cliques trouvées.

Donner une borne inférieure sur k (fonction de n) afin que la phase 2 de \mathcal{A}_k puisse se faire aisément en temps polynomial.

Réponse : Tout bloc possède au plus $b = \lceil n/k \rceil$ sommets. Par ailleurs, il y a au plus 2^b sous-ensembles deux-à-deux distincts de sommets dans un graphe de b sommets. Pour chacun d'entre eux, vérifier s'il forme un clique prend un temps au plus $O(b^2)$ car chacun de ces sous-ensembles induit un sous-graphe d'au plus b sommets. Trouver une clique maximum dans chacun des blocs prend donc un temps au plus $O(k \cdot (\frac{n}{k})^2 \cdot 2^{n/k})$. Choisir $k = n/(c \log n)$ permet d'obtenir un temps $O(n \cdot \log n \cdot 2^{c \log n})$, c'est-à-dire au plus $n^{O(1)}$, donc polynomial en n .

Question II.3. Soit K_{max} une clique de cardinalité maximum dans G . Montrer que la clique calculée par l'algorithme \mathcal{A}_k a une taille au moins $|K_{max}|/k$.

Réponse : Il y a k blocs, donc il existe un bloc B qui contient au moins $|K_{max}|/k$ sommets de K_{max} . Ces $|K_{max}|/k$ sommets forment une clique dans B . La taille de la clique maximum calculée par \mathcal{A}_k dans B est donc au moins $|K_{max}|/k$. \mathcal{A}_k retourne donc une clique de taille au moins $|K_{max}|/k$.

Question II.4. Déduire de ce qui précède un algorithme polynomial \mathcal{A} de $f(n)$ -approximation, avec $f(n) = o(n)$ (c'est-à-dire négligeable devant n).

Réponse : En prenant $k = n/(c \log n)$, on obtient un algorithme \mathcal{A}_k s'exécutant en temps $n^{O(1)}$ (d'après la question II.2), et de factuer d'approximation $n/(c \log n)$ (d'après la question II.3). On peut donc prendre $\mathcal{A} = \mathcal{A}_{n/(c \log n)}$. Plus l'on fixe c grand, meilleur est le facteur d'approximation, mais plus grand est le temps d'exécution.

— Problème III —

Une personne dispose de n produits financiers P_1, \dots, P_n de valeurs respectives v_1, \dots, v_n . Son objectif est de répartir ces n produits en un nombre minimum de portefeuilles de valeur maximum V_{max} où la valeur d'un portefeuille F contenant les produits P_i pour $i \in I \subseteq \{1, \dots, n\}$ est $\sum_{i \in I} v_i$. Les données de ce problème de minimisation sont donc les n valeurs v_1, \dots, v_n et la borne supérieure V_{max} . (On suppose sans perte de généralité que $v_i \leq V_{max}$ pour tout $i = 1, \dots, n$).

On considère l'algorithme “glouton” Gl suivant. Gl passe en revu les produits financiers un par un, dans un ordre arbitraire. Lorsque Gl traite un produit P_i , il procède de deux façons différentes selon que P_i peut être placé ou non dans un portefeuille ouvert F_j sans que la valeur de F_j excède la limite V_{max} . Si oui, alors P_i est placé dans F_j . Si non, alors un nouveau portefeuille est ouvert, et P_i est placé dans ce nouveau portefeuille.

Question III.1. Montrer que Gl assure qu'il n'existe pas deux portefeuilles ouverts, chacun de valeur au plus $V_{max}/2$.

Réponse : Par contradiction, supposons que Gl ouvre deux portefeuilles F_i et F_j , chacun de valeur au plus $V_{max}/2$. Sans perte de généralité, supposons que F_j a été ouvert après F_i . Plaçons nous à l'instant où Gl ouvre F_j . Il le fait parce qu'aucun des portefeuilles ouverts à cet instant ne peuvent contenir un certain P_k . Or, puisque la valeur de F_j n'excède pas $V_{max}/2$ à la fin, cela implique que $v_k \leq V_{max}/2$. Donc, v_k aurait pu être placé dans F_i . On obtient donc une contradiction.

Question III.2. Montrer que Gl est un algorithme de 2-approximation.

Réponse : Soit F_1, \dots, F_g les g portefeuilles ouverts par Gl , triés par ordre de valeur décroissante. Soit F_1^*, \dots, F_{OPT}^* une solution optimale. Par contradiction, supposons que $g > OPT$. La question précédente implique que, pour tout $i = 1, \dots, OPT$, on a $valeur(F_{2i-1}) + valeur(F_{2i}) \geq V_{max}$. Par ailleurs, pour tout $i = 1, \dots, OPT$, on a $valeur(F_i^*) \leq V_{max}$. Donc

$$\sum_{i=1}^n v_i = \sum_{i=1}^g valeur(F_i) > \sum_{i=1}^{OPT} (valeur(F_{2i-1}) + valeur(F_{2i})) \geq \sum_{i=1}^{OPT} valeur(F_i^*) = \sum_{i=1}^n v_i,$$

contradiction.

Question III.3. En définissant les variables booléennes y_j , $j = 1, \dots, n$, indiquant si le portefeuille F_j est ouvert ($y_j = 1$) ou pas ($y_j = 0$), et les variables booléennes $x_{i,j}$, $i, j = 1, \dots, n$,

telles que $x_{i,j} = 1$ si et seulement si P_i est placé dans le portefeuille F_j , écrire un programme linéaire en nombres entiers pour le problème.

Réponse : La contrainte que chaque produit P_i soit placé dans un portefeuille se traduit par $\sum_{j=1}^n x_{i,j} \geq 1$. La contrainte que le portefeuille F_j dans lequel P_i est placé soit ouvert se traduit par $y_j \geq x_{i,j}$. Finalement, la contrainte qu'un portefeuille ne puisse pas excéder la valeur V_{max} se traduit par $\sum_{i=1}^n x_{i,j} v_i \leq V_{max}$.

$$\begin{array}{ll} \text{minimiser} & \sum_{j=1}^n y_j \\ \text{sous les contraintes} & \sum_{j=1}^n x_{i,j} \geq 1 \text{ pour tout } i = 1, \dots, n \\ & y_j \geq x_{i,j} \text{ pour tout } i, j = 1, \dots, n \\ & \sum_{i=1}^n x_{i,j} v_i \leq V_{max} \text{ pour tout } j = 1, \dots, n \\ & x_{i,j} \in \{0, 1\} \text{ pour tout } i, j = 1, \dots, n \\ & y_j \in \{0, 1\} \text{ pour tout } j = 1, \dots, n \end{array}$$

Question subsidiaire : montrer que la version décision du problème de minimisation étudié ici est NP-complet.

F Examen de rattrapage 8 février 2012, 10h00-11h00

Cet examen traite du produit $A_1 A_2 \dots A_n$ de n matrices à coefficients réels, où la matrice A_i est une matrice $p_{i-1} \times p_i$ (p_{i-1} lignes et p_i colonnes). La matrice produit est donc $p_0 \times p_n$. Rappelons que si le produit de matrices n'est pas commutatif, il est associatif. Il y a malheureusement de nombreuses façons de parenthésier le produit, par exemple $(A_1(A_2(A_3 \dots A_n)))$ ou $(A_1 \dots A_{\lfloor n/2 \rfloor})(A_{\lfloor n/2 \rfloor + 1} \dots A_n)$. Le produit d'une matrice $p \times q$ par une matrice $q \times r$ prenant pqr multiplications de réels (par l'algorithme classique de multiplication de matrices), chaque parenthésage induit un nombre de multiplications potentiellement différent d'autres parenthésages. L'objectif de ce problème est de construire un algorithme qui, étant donnée une suite p_0, p_1, \dots, p_n d'entiers positifs, calcule un parenthésage minimisant le nombre de multiplications pour le produit $A_1 A_2 \dots A_n$ de n matrices réelles, où A_i est de dimension $p_{i-1} \times p_i$.

Question 1. Soit A_1 de dimension 10×100 , A_2 de dimension 100×5 , et A_3 de dimension 5×50 . Calculer le parenthésage optimal pour le produit $A_1 A_2 A_3$.

Réponse : Le parenthésage $((A_1 A_2) A_3)$ induit 7.500 multiplications, alors que le parenthésage $(A_1 (A_2 A_3))$ induit 75.000 multiplications. Le premier parenthésage est donc 10 fois plus efficace que le second !

Question 2. Montrer que le nombre de parenthésages pour n matrices peut être exponentiel.

Réponse : Fixons n une puissance de 2. En considérant le parenthésage $(A_1 \dots A_{n/2})(A_{n/2+1} \dots A_n)$, on en déduit que si $\pi(n)$ dénote le nombre de parenthésages pour un produit de n matrices, on a $\pi(n) \geq \pi(n/2)^2$. Donc $\pi(n) \geq \pi(n/4)^4 \geq \dots$. En itérant $\log_2 n - 2$ fois, on obtient $\pi(n) \geq \pi(4)^{n/4}$. Comme $\pi(4) \geq 2$, on obtient le résultat désiré. (En fait, on peut facilement montrer que $\pi(n)$ est égale au $(n - 1)$ ème nombre de Catalan, pour tout n).

Question 3. Montrer que si $P = (A_1 \dots A_k)(A_{k+1} \dots A_n)$ est un parenthésage optimal pour le produit $A_1 A_2 \dots A_n$ alors le parenthésage induit par P pour le produit $A_1 \dots A_k$ et le parenthésage induit par P pour le produit $A_{k+1} \dots A_n$ sont tous deux optimaux pour chacun de ces deux produits.

Réponse : Le nombre N de multiplications pour le parenthésage $P = (A_1 \dots A_k)(A_{k+1} \dots A_n)$ est $N' + N'' + p_0 p_k p_n$ où N' est le nombre de multiplications pour $A_1 \dots A_k$, et N'' est le nombre de multiplications pour $A_{k+1} \dots A_n$. Si le parenthésage induit par P pour le produit $A_1 \dots A_k$ n'était pas optimal, c'est-à-dire si N' n'était pas le nombre minimal de multiplications pour $A_1 \dots A_k$, alors remplacer ce parenthésage par le parenthésage optimal pour $A_1 \dots A_k$ réduirait le nombre de multiplications, en contradiction avec l'optimalité de P , i.e., la minimalité de N . Même raisonnement pour $A_{k+1} \dots A_n$.

Question 4. Pour $i \leq j$, notons $m_{i,j}$ le nombre de multiplications réelles induites par un parenthésage optimal du produit $A_i \dots A_j$. Donner une expression de $m_{i,j}$ en fonction d'autres valeurs $m_{i',j'}$, dans un objectif de conception de programmes dynamiques.

Réponse : On a $m_{i,j} = 0$ si $i = j$. Si $i < j$, alors un parenthésage optimal du produit $A_i \dots A_j$ est de la forme $(A_i \dots A_k)(A_{k+1} \dots A_j)$ pour $i \leq k < j$, et donc $m_{i,j} = \min_{i \leq k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j\}$.

Question 5. Ecrire un algorithme issue de la programmation dynamique permettant, étant donnée une séquence p_0, p_1, \dots, p_n d'entiers positifs, calcule le nombre minimal de multiplications pour le produit $A_1 A_2 \dots A_n$ de n matrices réelles, où A_i est de dimension $p_{i-1} \times p_i$.

Réponse : On va remplir un tableau $mult$ tel que $mult[i, j] = m_{i,j}$. La difficulté provient de la façon dont on doit remplir $mult$. On doit le faire par diagonale, en commençant par $mult[i, i] = 0$. La matrice triangulaire inférieure de $mult$ n'a pas à être considérée car on n'a besoin des valeurs de $mult[i, j]$ que pour $j \geq i$. En effet, pour affecter $mult[i, j]$, il suffit d'avoir stocké les valeurs $mult[i, k]$ et $mult[k, j]$ pour $i \leq k < j$.

ALGORITHME PARENTHÈSE

début

```

pour  $i = 1$  à  $n$  faire  $mult[i, i] \leftarrow 0$ ; (mise à jour de la diagonale  $i = j$ )
pour  $diag = 2$  à  $n$  faire (parcours des  $n - 1$  sur-diagonales)
    pour  $i = 1$  à  $n - diag + 1$  faire (pour tous les indices  $i$  sur la diagonale  $diag$ )
         $j \leftarrow i + diag - 1$ ; (on fixe le  $j$  correspondant à  $i$  sur la diagonale  $diag$ )
         $mult[i, j] \leftarrow \min_{i \leq k < j} \{mult[i, k] + mult[k + 1, j] + p_{i-1} p_k p_j\}$ 
```

fin.

Question 6. Ecrire un algorithme issue de la programmation dynamique réalisant le produit $A_1 A_2 \dots A_n$ avec un nombre optimal de multiplications.

Réponse : En rajoutant l'instruction “ $\text{parenthesis}[i, j] \leftarrow k_{\min}$ ” après la dernière instruction dans l'algorithme de la question précédente, où k_{\min} est l'argument de la minimisation, on retient ainsi l'endroit où est parenthésé le produit $A_i \dots A_j$. On peut alors construire l'algorithme récursif suivant, où $A = (A_1, \dots, A_n)$:

ALGORITHME MULTIPLIER-CHAINE(A, i, j)

début

```

si  $j > i$  alors
     $X \leftarrow \text{MULTIPLIER-CHAINE}(A, i, \text{parenthesis}[i, j])$ ;
     $Y \leftarrow \text{MULTIPLIER-CHAINE}(A, \text{parenthesis}[i, j] + 1, j)$ ;
    retourner  $XY$ ;
sinon retourner  $A$ ;
fin.
```

En appellant *MULTIPLIER-CHAINE*($A, 1, n$), on obtient le produit de la chaîne $A = A_1 \dots A_n$ en un nombre optimal de multiplications.

G Examen du 30 octobre 2012, durée 3h

L'examen est découpé en trois problèmes indépendants.

— Problème I —

Soit $G = (V, E)$ un graphe. Un couplage dans G est un ensemble d'arêtes $C \subseteq E$ n'ayant deux-à-deux aucune extrémité en commun. Un couplage C est maximal s'il n'existe pas de couplage C' tel que $C \subset C'$. Un couplage C est maximum s'il n'existe pas de couplage C' tel que $|C| < |C'|$ (où $|X|$ dénote la cardinalité d'un ensemble X).

On note $[n] = \{1, \dots, n\}$.

Question I.1. Donner une description sous forme d'un programme linéaire en nombre entier du problème du couplage maximum (CM).

Réponse : On introduit une variable booléenne x_e pour chaque arête $e \in E$, où $x_e = 1$ si et seulement e est incluse dans le couplage. Le PL est alors le suivant :

$$\begin{array}{ll} \text{maximiser} & \sum_{e \in E} x_e \\ \text{sous les contraintes} & (1) \quad x_e \in \{0, 1\} \text{ pour tout } e \in E \\ & (2) \quad \sum_{e \in E(u)} x_e \leq 1 \text{ pour tout } u \in V \end{array}$$

où $E(u)$ dénote l'ensemble des arêtes incidentes à u . La contrainte (2) exprime bien que chaque sommet ne peut être couplé qu'au plus une fois.

Question I.2. Donner une description sous forme d'un programme linéaire en nombre entier du problème du couplage maximal de cardinalité minimum (CMCM). C'est-à-dire, on cherche un couplage maximal C tel que $|C| \leq |C'|$ pour tout couplage maximal C' .

Réponse : On utilise les mêmes variables x_e . Le PL est alors le suivant :

$$\begin{array}{ll} \text{minimiser} & \sum_{e \in E} x_e \\ \text{sous les contraintes} & (1) \quad x_e \in \{0, 1\} \text{ pour tout } e \in E \\ & (2) \quad \sum_{e \in E(u)} x_e \leq 1 \text{ pour tout } u \in V \\ & (3) \quad \sum_{e' \in E(u)} x_{e'} + \sum_{e' \in E(v)} x_{e'} \geq 1 \text{ pour tout } e = \{u, v\} \in E \end{array}$$

La contrainte (3) exprime la maximalité du couplage : parmi e et les arêtes incidentes à e , une doit nécessairement être dans le couplage.

Question I.3. Montrer que tout couplage maximal a une cardinalité au moins moitié de celle d'un couplage maximum.

Réponse : Un couplage maximal satisfait la contrainte (3) de la question I.2. Donc, pour chaque arête $e = \{u, v\}$, au moins une variable $x_{e'} = 1$ pour $e' \in E(u) \cup E(v)$. Un couplage maximum satisfait la contrainte (2). Donc, parmi les arêtes $e' \in E(u) \cup E(v)$ au plus deux peuvent faire $x_{e'} = 1$. Le nombre d'arêtes c_{\max} dans un couplage maximum ne peut donc excéder deux fois le nombre d'arêtes d'un couplage maximal quelconque.

Il est également possible de montrer cela sans utiliser la formulation en programme linéaire. Soit C un couple maximal, et soit C^* in couplage maximum. Supposons que $|C| < |C^*|/2$. Alors le nombre de sommets couplés dans C est $2|C| < |C^*|$. Ainsi, il existe au moins une arête $e \in C^*$ dont aucune extrémité est dans C . En conséquence, $C \cup \{e\}$ est un couplage, contredisant la maximalisé de C .

Question I.4. Décrire un algorithme polynomial de 2-approximation du problème CMCM.

Réponse : Il suffit de calculer un calculer une couplage maximal C quelconque. En effet, on a $\text{OPT} \leq |C| \leq c_{\max}$ et, d'après la question précédente, on a $c_{\max} \leq 2 \text{OPT}$ puisque OPT est en particulier maximal. Le calcul d'un couplage maximal peut se faire par un algorithme glouton, tel que vu en cours :

```

début
   $C \leftarrow \emptyset$ 
   $X \leftarrow E$ 
  tant que  $X \neq \emptyset$  faire
    choisir  $e \in X$ 
     $C \leftarrow C \cup \{e\}$ 
    enlever  $e$  de  $X$  ainsi que toute ses arêtes incidentes
    retourner  $C$ .
fin
```

— Problème II —

Soit $G = (V, E)$ un graphe. Une k -coloration de G est une affectation d'une couleur $c(u) \in \{1, \dots, k\}$ à chaque sommet $u \in V$ tel que, pour toute arête $e = \{u, v\} \in E$, on a $c(u) \neq c(v)$. Le problème de la coloration (COL) consiste, étant donné un graphe G , à calculer une k -coloration de G avec k aussi petit que possible. Pour tout $u \in V$, on note $N(u) = \{v \in V : \{u, v\} \in E\}$.

Question II.1. Donner une description sous forme d'un programme linéaire en nombre entier du problème COL.

Réponse : On introduit une variable booléenne $x_{u,c}$ pour chaque sommet $u \in V$ et chaque couleur $c \in [n]$, telle que $x_{u,c} = 1$ si et seulement si le sommet u est coloré c . On introduit de plus une variable booléenne y_c pour chaque couleur, qui indique si la couleur c est utilisée. Le PL est alors le suivant :

$$\begin{aligned}
&\text{minimiser} && \sum_{c \in [n]} y_c \\
&\text{sous les contraintes} &&
\begin{aligned}
(1) \quad &y_c \in \{0, 1\} \text{ pour tout } c \in [n] \\
(2) \quad &x_{u,c} \in \{0, 1\} \text{ pour tout } u \in V \text{ et } c \in [n] \\
(3) \quad &\sum_{c \in [n]} x_{u,c} = 1 \text{ pour tout } u \in V \\
(4) \quad &x_{u,c} \leq y_c \text{ pour tout } u \in V \text{ et tout } c \in [n] \\
(5) \quad &x_{u,c} + x_{v,c} \leq 1 \text{ pour tout } c \in [n] \text{ et tout } \{u, v\} \in E
\end{aligned}
\end{aligned}$$

La contrainte (3) spécifie que tout sommet est coloré avec une unique couleur. La contrainte (4) spécifie qu'un sommet ne peut être coloré c sans que la couleur c soit utilisée. Enfin, la contrainte (5) spécifie que deux sommets adjacents ont des couleurs différentes.

Question II.2. Soit $\Delta = \max_{u \in V} |N(u)|$ le degré maximum des sommets de G . Déduire de la question précédentes que tout graphe de degré maximum Δ admet une $(\Delta + 1)$ -coloration.

Réponse : Il suffit de vérifier que le PL de la question précédente admet une solution admissible en restreignant $c \leq \Delta + 1$, par exemple avec $y_1 = y_2 = \dots = y_{\Delta+1} = 1$ et $y_c = 0$ pour tout $c > \Delta + 1$. Il suffit de vérifier que l'on peut satisfaire (5) avec des couleurs entre 1 et $\Delta + 1$. On effectue cette vérification de façon gloutonne. Pour chaque sommet u , chacun de ses voisins v ne peut que saturer qu'une équation du type (5), celle qui correspond à la couleur de v , pour un nombre maximum de Δ équations saturées, contrignant au plus Δ couleurs. Comme on dispose de $\Delta + 1$ couleurs, une équation reste donc non saturée pour au moins une couleur, et on peut donc affecter à u cette couleur. Les contraintes (3) et (4) sont alors également satisfaites.

Question II.3. Décrire un algorithme glouton de $(\Delta + 1)$ -coloration pour tout graphe de degré maximum Δ .

Réponse :

début

$X \leftarrow \emptyset$

tant que $X \neq V$ faire

 choisir $u \in V \setminus X$

$X \leftarrow X \cup \{u\}$

$C \leftarrow$ ensemble des couleurs utilisées par les voisins de u dans X

 Donner à u la plus petite couleur entre 1 et $\Delta + 1$ non présente dans C .

fin

L'algorithme est correct ca $|C| \leq \Delta$, et donc il reste toujours une couleur disponible.

— Problème III —

Le problème PARTITION est le suivant :

Données : n objets O_1, \dots, O_n de poids respectifs w_1, \dots, w_n ;

Question : existe-t-il une partition des objets en deux groupes I et J tels que $\sum_{i \in I} w_i = \sum_{i \in J} w_i$?

Par “partition”, il faut comprendre $I \cap J = \emptyset$ et $I \cup J = [n]$. Le problème PARTITION est NP-complet.

Le problème PAQUETAGE est le suivant :

Données : n boites B_1, \dots, B_n de volumes respectifs v_1, \dots, v_n , deux entiers positifs V et k ;

Question : Peut-on ranger les n boites dans k sacs, chacun de volume au plus V ?

Question III.1. Montrer que PAQUETAGE \in NP.

Réponse : Remarquons tout d'abord que si $v_i > V$ alors PAQUETAGE est trivialement faux. On suppose donc sans perte de généralité que $v_i \leq V$ pour tout $i \in [k]$. Sous cette hypothèse, si $k \geq n$, alors PAQUETAGE est trivialement vrai. Ainsi, on peut également supposer sans perte de généralité que $k < n$. Sous ces hypothèses, un certificat consiste simplement en le vecteur “de rangement” r , composés d'entiers $r_i \in [k]$ pour $i \in [n]$, indiquant le sac dans lequel est rangée la boite i . La taille du certificat est $O(n \log k)$, donc polynomial en la taille des entrées. L'algorithme de vérification consiste à effectuer les k sommes $\sum_{i:r_i=j} v_j$ pour $j = 1, \dots, k$, et vérifier que chacune d'entre elles vaut au plus V . Le nombre d'additions est au plus $O(n)$ puisqu'il y a n boites. Le nombre de comparaisons est $O(k)$, donc également polynomial en la taille des données puisque $k < n$.

Question III.2. Montrer que PAQUETAGE est NP-complet.

Réponse : On effectue une réduction de PARTITION à PAQUETAGE. Soit n objets O_1, \dots, O_n de poids respectifs w_1, \dots, w_n une instance P de PARTITION. On construit l'instance $f(P) = Q$ de PAQUETAGE suivante : $B_i = O_i$ et $w_i = v_i$ pour tout $i = 1, \dots, n$, $V = \frac{1}{2} \sum_{i=1}^n w_i$ et $k = 2$.

– La transformation f est bien polynomiale puisque la seule opération effectuée est le calcul de la somme pour l'affectation de V , ce qui prend un temps $O(n)$.

– Supposons que P admet une partition I, J . Alors $\sum_{i \in I} w_i = \sum_{i \in J} w_j$, donc $\sum_{i \in I} v_i = \sum_{i \in J} v_j = \frac{V}{2}$, ce qui revient à dire que l'on peut ranger les n boîtes B_i dans les deux sacs.

– Réciproquement, supposons que les n boîtes de Q peuvent être rangées dans deux sacs. Chacun de ces deux sacs ayant un volume au plus V , et le volume total des boîtes étant $2V$, il en découle que chaque sac a un volume exactement V , et qu'ils correspondent donc à une partition des n objets de P en deux ensembles.

On a donc bien P admet une partition si et seulement si $f(P)$ peut être rangé. PAQUETAGE est donc NP-difficile. Puisque PAQUETAGE est dans NP d'après la question précédente, on obtient que PAQUETAGE est NP-complet.

Question III.3. On considère l'algorithme glouton suivant pour la minimisation du nombre k de sacs pour PAQUETAGE :

```

début
    trier les boîtes par ordre décroissant de volume
    on suppose donc maintenant que  $v_1 \geq v_2 \geq \dots \geq v_n$ 
     $k \leftarrow 0$ 
    pour  $i = 1$  à  $n$  faire
        si  $B_i$  peut être placé dans un sac  $S_j$  avec  $j \leq k$  alors
            placer  $B_i$  dans  $S_j$ 
        sinon
             $k \leftarrow k + 1$ 
            placer  $B_i$  dans un nouveau sac  $S_k$ 
    fin

```

Soit B_j la boîte ayant amenée la création du dernier sac S_k par l'algorithme glouton. Montrer que si $v_j > V/2$ alors l'algorithme glouton est optimal.

Réponse : Puisque les boîtes sont ordonnées en ordre décroissant de volume, toutes les boîtes B_1, \dots, B_j ont un volume $> V/2$. Chacune d'entre elles nécessite donc un sac différent des $j - 1$ autres. Donc OPT doit ouvrir j sacs pour ces j boîtes, donc au moins j sacs au total, i.e., $\text{OPT} \geq j$. Or, puisque c'est la boîte j qui a causé l'ouverture de la dernière boîte de glouton, ce dernier n'ouvre pas plus de j sacs, i.e., $k \leq j$. Ainsi, $k = \text{OPT}$, et Glouton est donc optimal.

Question III.4. On suppose maintenant que $v_j \leq V/2$. La capacité total des $k - 1$ premiers sacs est de $(k - 1)V$. Cette capacité se répartie en une partie *occ* occupée par des boîtes, et une partie *vide* non occupée par des boîtes. Montrer que $\text{volume}(\text{occ}) \leq \sum_{i=1, i \neq j}^n v_i$ et que $\text{volume}(\text{vide}) \leq (k - 1)v_j$. En déduire que l'algorithme glouton est un algorithme de 2-approximation pour PAQUETAGE.

Réponse : Puisque B_j n'est pas placée dans les $k - 1$ premiers sacs, on a $\text{volume}(\text{occ}) \leq (\sum_{i=1}^n v_i) - v_j$. Puisque B_j a causé l'ouverture d'un nouveau sac, il n'y avait pas la place pour cette boîte dans aucun des $k - 1$ premiers sacs, donc chacun d'entre eux avait un volume disponible

strictement plus petit que v_j , et donc $\text{volume}(\text{vide}) < (k - 1)v_j$. On a donc

$$(k - 1)V = \text{volume}(\text{occ}) + \text{volume}(\text{vide}) < \sum_{i=1}^n v_i + (k - 1)v_j \leq \sum_{i=1}^n v_i + (k - 1)V/2.$$

A ce stade, il suffit de remarquer que $\text{OPT} \geq \frac{1}{V} \sum_{i=1}^n v_i$ car au mieux le volume total $\sum_{i=1}^n v_i$ se répartit équitablement entre les sacs en les occupant chacun entièrement. Donc

$$(k - 1)V < \text{OPT} V + (k - 1)V/2$$

d'où l'on déduit que $(k - 1)/2 < \text{OPT}$, et donc $k \leq 2 \text{OPT}$.

Question III.5. En utilisant une transformation semblable à celle utilisée dans la question III.2, montrer que, sauf si $P=NP$, il n'existe pas d'algorithme polynomial d'approximation de PAQUETAGE à un facteur $\frac{3}{2} - \epsilon$, pour tout $\epsilon > 0$.

Réponse : Par contradiction, supposons qu'il existe un algorithme A polynomial de $(\frac{3}{2} - \epsilon)$ -approximation pour PAQUETAGE. On utilise la transformation polynomiale f de la question III.2 pour obtenir un algorithme polynomial B pour PARTITION comme suit. Soit P une instance de PARTITION.

début

$$Q = f(P)$$

appliquer A à Q

si le nombre de sacs ouverts est au moins 3 alors rejeter P sinon accepter P

fin

Montrons que B est correct. Soit P une instance positive pour PARTITION. Alors l'optimal pour $Q = f(P)$ est de deux sacs. Comme $2(\frac{3}{2} - \epsilon) < 3$, l'algorithme A n'ouvrira que deux sacs également, et B acceptera donc P . Inversement, soit P une instance négative pour PARTITION. Alors l'optimal pour $Q = f(P)$ est d'au moins trois sacs, et donc A ouvrira forcément au moins trois sacs, et B rejettéra Q . B est donc un algorithme polynomial décidant PARTITION, en contradiction avec le fait que PARTITION est NP-complet, sous l'hypothèse $P \neq NP$.

H Examen de rattrapage 27 février 2013, durée 1h30

Le problème PAQUETAGE sous forme décisionnelle est le suivant :

Données : n boites B_1, \dots, B_n de volumes respectifs v_1, \dots, v_n , deux entiers positifs V et k ;

Question : Peut-on ranger les n boites dans k sacs, chacun de volume au plus V ?

Question 1. Exprimer deux problèmes d'optimisation pouvant être associés à PAQUETAGE.

Réponse : On peut, pour un volume fixé V , chercher à minimiser le nombre de sacs nécessaires au stockage des n boites. Alternativement, on peut, pour un nombre de sacs fixé k , minimiser le volume de ces sacs afin de pouvoir y stocker les n boites.

Dans la suite, PAQUETAGE désignera l'une ou l'autre des versions décisionnelle ou d'optimisation du nombre de sacs de ce problème.

Question 2. Donner une formulation de PAQUETAGE sous forme d'un programme linéaire en nombre entier (PLNE).

Réponse : On peut supposer que $v_i \leq V$ pour tout i car sinon on ne peut pas stocker les objets, quelque soit le nombre de sacs. Soit x_i , $i = 1, \dots, n$, la variable booléenne indiquant si le sac i est utilisé. Soit $y_{i,j}$, $i, j = 1, \dots, n$, la variable booléenne indiquant si la B_j est placé dans le sac i . Un PLNE répondant à la question est le suivant :

$$\text{minimiser} \quad \sum_{i=1}^n x_i$$

- sous les contraintes
- (1) $x_i \in \{0, 1\}$ pour tout $i \in \{1, \dots, n\}$
 - (2) $y_{i,j} \in \{0, 1\}$ pour tout $i \in \{1, \dots, n\}$ et $j \in \{1, \dots, n\}$
 - (3) $y_{i,j} \leq x_i$ pour tout $i \in \{1, \dots, n\}$ et $j \in \{1, \dots, n\}$
 - (4) $\sum_{j=1}^n y_{i,j} v_j \leq V$ pour tout $i \in \{1, \dots, n\}$
 - (5) $\sum_{i=1}^n y_{i,j} \geq 1$ pour tout $j \in \{1, \dots, n\}$

La contrainte (3) force à utiliser le sac i si une boîte j y est placée. La contrainte (4) stipule qu'un sac ne peut contenir des boîtes de volume total excédant V . Enfin, la contrainte (5) stipule que toute boîte j doit être placée dans au moins 1 sac i .

Question 3. Déduire de cette expression sous forme PLNE qu'une solution optimale ne peut avoir plus d'un sac à moitié vide.

Réponse : Supposons que les sacs i_1 et i_2 sont tous deux à moitié vide dans une certaine solution admissible S . C'est-à-dire $\sum_{j=1}^n y_{i_1,j} v_j \leq V/2$ et $\sum_{j=1}^n y_{i_2,j} v_j \leq V/2$ dans S . Remplaçons toutes les occurrences de $y_{i_2,j} = 1$ de S par $y_{i_1,j} = 1$ et $y_{i_2,j} = 0$, afin d'obtenir une nouvelle solution S' où $x_{i_2} = 0$. S' est admissible car la somme $\sum_{j=1}^n y_{i_1,j} v_j$ dans S' est égale à la somme $\sum_{j=1}^n y_{i_1,j} v_j + \sum_{j=1}^n y_{i_2,j} v_j \leq V/2 + V/2 = V$ dans S . La valeur de la solution admissible S' est strictement inférieure à celle de S (car $x_{i_2} = 0$ dans S'). Donc S n'est pas optimale.

Question 4. Proposer un algorithme glouton simple pour PAQUETAGE pour lequel aucun pré-calcul sur les données n'est possible (en particulier, cet algorithme ne doit pas nécessiter de trier les volumes des boîtes). Votre algorithme doit traiter les données à la volée : les boîtes B_i sont données l'une après l'autre et, pour chaque boîte, votre algorithme doit affecter un sac à cette boîte (soit un sac existant, soit un nouveau sac).

Réponse : Pour $j = 1$ à n effectuer les instructions suivantes : chercher, parmi les sacs couramment utilisés, s'il existe un sac auquel il est possible de ajouter B_j sans excéder le volume total V du sac ; si oui, y placer B_j ; si non, ouvrir un nouveau sac et y placer B_j .

Question 5. Montrer que votre algorithme glouton est un algorithme de 2-approximation. (Si ce n'est pas le cas, revoir votre algorithme afin qu'il retourne une solution 2-approximante).

Réponse : Supposons que l'algorithme retourne deux sacs à moitié vide i_1 et i_2 , et supposons que i_2 ait été ouvert après i_1 . Soit B_j la boîte causant l'ouverture de i_2 . Cette boîte ne pouvait pas être placée dans i_1 . Donc $v_j > V/2$ car au moment où l'algorithme considère B_j , le volume utilisé de i_1 est au plus le volume utilisé de i_1 à la fin de l'exécution de l'algorithme, soit au plus $V/2$. Il découle que le volume utilisé de i_2 est au moins $v_j > V/2$, contradiction. En conséquence, l'algorithme ne retourne qu'au plus un sac à moitié vide. Soit ALG le nombre de sacs utilisé par l'algorithme glouton. On a donc $(\text{ALG} - 1)V/2 < \sum_{j=1}^n v_j$. Or $\text{OPT} \geq (\sum_{j=1}^n v_j)/V$. Donc $\text{OPT} > (\text{ALG} - 1)/2$, et donc $\text{ALG} \leq 2 \text{ OPT}$.

Question 6. Est-il possible de concevoir un schéma d'approximation polynomial pour PAQUETAGE ?

Réponse : L'examen du 30 octobre 2012, question III.5, a montré que PAQUETAGE ne pouvait être approximé à tout facteur $\frac{3}{2} - \epsilon$, pour tout $\epsilon > 0$, sauf si $P=NP$. En conséquence, sauf si $P=NP$, il n'existe pas de schéma d'approximation polynomial pour PAQUETAGE.

Question 7. Proposer une version bi-dimensionnelle de PAQUETAGE.

Réponse : Une généralisation triviale est la suivante. Les boites sont remplacées par des rectangles R_j de hauteur h_j et largeur ℓ_j . Les sacs ont deux dimensions H et L . On ne peut placer des rectangles dans une même boite que si la somme de leur hauteur n'excède pas H , et la somme de leur largeur n'excède pas L . Il existe des généralisations plus sophistiquées (voir par exemple Two-dimensional packing problems : A survey par Lodi, Martello, et Monaci).

I Examen du 18 octobre 2013, durée 3h

L'examen est découpé en deux problèmes indépendants. Pour tout entier positif n , on note $[n] = \{1, \dots, n\}$.

— Problème I —

Vous êtes à la tête de la fameuse compagnie automobile de luxe Roule-Roux. Le fleuron de votre compagnie, la Deux-Anes, est produit sur k chaînes de montage, chacune composée de n machines, chacune exécutant une tâche T_i , $i \in [n]$. On note $M_{i,j}$ la i ème machine de la j ème chaîne, $i \in [n]$, $j \in [k]$. Pour tout $i \in [n]$, les machines de l'ensemble $\{M_{i,j}, j \in [k]\}$, ont exactement la même fonction : effectuer la tâche T_i . Construire une Deux-Anes nécessite d'effectuer le tâche T_1 , puis T_2 , etc., jusqu'à la tâche T_n , l'une après l'autre, dans cet ordre. Vos chaînes de montage ont cependant subit des modifications avec le temps, et certaines machines sont plus récentes que d'autres. Ainsi, le temps d'exécution de la i ème tâche n'est pas nécessairement le même sur deux machines $M_{i,j}$ et $M_{i,j'}$ pour $j \neq j'$.

Vous recevez soudain une commande urgente d'un client prestigieux qui souhaite acquérir une Deux-Anes, mais seulement si vous êtes capable de la produire en moins de temps que pour produire une voiture de votre concurrent italien Firrore. Pour produire une Deux-Anes, vous pouvez bien sûr la produire sur une unique chaîne de montage j en temps $\sum_{i=1}^n m_{i,j}$ où $m_{i,j}$ est de temps pour exécuter la i ème tâche de construction de la Deux-Anes sur la machine $M_{i,j}$. Vous pouvez toutefois passer de chaîne en chaîne si par exemple vous voulez utiliser pour T_i une machine $M_{i,j}$ très moderne de la chaîne j , et pour $T_{i'}, i' \neq i$, une machine $M_{i',j'}$ d'une autre chaîne $j' \neq j$.

On note $t_{i,j,j'}$ le temps pour faire passer une Deux-Anes partiellement construite de la sortie de la machine $M_{i,j}$ à l'entrée de la machine $M_{i+1,j,j'}$, pour tout $i \in [n-1]$ et $j, j' \in [k]$.

Question I.1. Soit $c_{i,j}$ le temps minimum pour effectuer les i premières tâches de construction d'une Deux-Anes, en passant potentiellement de chaîne en chaîne en finissant par exécuter la tâche i sur la chaîne j . Exprimer $c_{i,j}$ en fonction des $t_{i',j,j'}$, $m_{i',j,j'}$ et $c_{i',j,j'}$, avec $i' \leq i$.

Réponse : On a $c_{1,j} = m_{1,j}$ pour tout j . Pour $i > 1$, on a la récurrence suivante :

$$c_{i,j} = \min_{j' \in [k]} (c_{i-1,j'} + t_{i-1,j',j}) + m_{i,j}$$

(Notez que l'énoncé ne stipule pas que $t_{i,j,j} = 0$).

Question I.2. Décrire un algorithme de programmation dynamique vous permettant de savoir si vous pouvez répondre favorablement ou pas à la requête de votre client.

Réponse : On remplit le tableau C afin d'obtenir $C[i, j] = c_{i,j}$:

```

début
    pour  $j = 1$  à  $k$  faire  $C[1, j] = m_{1,j}$ 
    pour  $i = 2$  à  $n$  faire
        pour  $j = 1$  à  $k$  faire
             $C[i, j] \leftarrow \min_{j' \in [k]} (C[i - 1, j'] + t_{i-1, j', j} + m_{i,j})$ 
         $D \leftarrow \min_{j \in [k]} C[n, j]$ 
fin

```

La valeur de D correspond au temps minimum nécessaire pour produire une Deux-Anes.

Question I.3. Quel est le temps d'exécution de votre algorithme ? Comparez le avec celui que l'on pourrait obtenir par une adaptation de l'algorithme de Dijkstra.

Réponse : Le temps de l'algorithme est dominé par celui de la double boucle « pour ». Il y a $k(n - 1)$ itérations. Chaque itération demande de calculer un minimum de k valeurs, en temps $O(k)$. Le temps total de l'algorithme est donc $O(nk^2)$.

Le problème résolu n'est rien d'autre qu'une recherche de plus court chemin dans un graphe. On aurait pu adapter l'algorithme de Dijkstra à un graphe orienté (et en rajoutant des coûts aux sommets) pour calculer ce plus court chemin. L'algorithme de Dijkstra dans un graphe $G = (V, E)$ s'exécute en temps $O(|E| + |V| \log |V|)$, soit $O(nk^2 + nk \log(nk))$ dans notre contexte. Pour k petit devant n , notre algorithme est donc plus rapide.

Question I.4. Supposons que, pour tout i et tous j, j' , on ait $t_{i,j,j'} = 0$. Donner une description sous forme d'un programme linéaire en nombre entier de votre problème.

Réponse : On introduit une variable booléenne $x_{i,j}$ pour chaque machine $M_{i,j}$, tel que $x_{i,j} = 1$ si et seulement si la tâche T_i est exécutée sur la machine $M_{i,j}$. Le PLNE est alors le suivant :

$$\begin{aligned}
&\text{minimiser} && \sum_{(i,j) \in [n] \times [k]} x_{i,j} \cdot m_{i,j} \\
&\text{sous les contraintes} && (1) \quad x_{i,j} \in \{0, 1\} \text{ pour tout } (i, j) \in [n] \times [k] \\
&&& (2) \quad \sum_{j \in [k]} x_{i,j} \geq 1 \text{ pour tout } i \in [n]
\end{aligned}$$

La contrainte (2) exprime que toute tâche T_i doit être effectuée sur au moins une machine.

Question I.5. Montrer que votre programme linéaire a en fait une solution triviale. (Si vous n'avez pas réussi à écrire le programme linéaire, montrer que votre problème a une solution triviale sous les hypothèses de la question I.4).

Réponse : Il suffit, pour tout $i \in [n]$, d'exécuter la tâche T_i sur la machine M_{i,j_i} telle que

$$m_{i,j_i} = \min_{j \in [k]} m_{i,j}.$$

En effet la contrainte (2) est bien satisfaite, et cette solution est optimale car $\text{OPT} \geq \sum_{i=1}^n m_{i,j_i}$ puisqu'exécuter la tâche T_i nécessite un temps au moins m_{i,j_i} .

— Problème II —

On rappelle qu'une coupe d'un graphe $G = (V, E)$ est une partition¹⁰ (S, T) des sommets de G (i.e., $S \cup T = V$ et $S \cap T = \emptyset$). Par extension, une coupe (S, T) désigne également l'ensemble des arêtes $\{u, v\} \in E$ telles que $u \in S$ et $v \in T$. Le problème MAXCUT consiste, étant donné

10. non triviale ! C-à-d. S et T sont tous deux non vides.

un graphe $G = (V, E)$, à trouver une coupe (S, T) dont la cardinalité (i.e., le nombre d'arêtes) est maximum parmi toutes les coupes de G . Le problème MINCUT consiste, étant donné un graphe $G = (V, E)$, à trouver une coupe (S, T) dont la cardinalité est minimum parmi toutes les coupes de G .

On considère la stratégie itérative ITER suivante pour MAXCUT. Partant d'une coupe quelconque (S, T) dans G , on l'améliore successivement comme suit : étant donnée la coupe courante (S, T) , chercher s'il existe un sommet u tel que la coupe $(S \cup \{u\}, T \setminus \{u\})$ ou la coupe $(S \setminus \{u\}, T \cup \{u\})$ a une cardinalité strictement supérieure à celle de (S, T) ; si oui, alors passer u d'un côté de la coupe à l'autre.

Question II.1. Montrer que la stratégie ITER termine. Donnez une version explicite de la stratégie ITER.

Réponse : La cardinalité d'une coupe maximum dans $G = (V, E)$ est au plus $|E|$. A chaque étape de la stratégie ITER, la cardinalité de la coupe courante augmente au moins de 1. La stratégie ITER termine donc après au plus $|E|$ itérations.

Pour donner une version explicite de la stratégie, il faut formaliser le « quelconque » de sa description. Il suffit par exemple de partir de la coupe triviale (\emptyset, V) .

Question II.2. Montrer que l'algorithme ITER est un algorithme polynomial de 2-approximation de MAXCUT.

Réponse : Il y au plus $m = |E|$ itérations. A chaque itération, on passe en revue les $n = |V|$ sommets, et, pour chacun, on regarde si la coupe peut être améliorée. Le test pour un sommet u se fait en au plus $O(\deg(u))$ opérations. Le temps total est donc polynomial, au plus $O(m \sum_u \deg(u)) = O(m^2)$.

En terme d'approximation, il suffit de remarquer que, à la fin de l'algorithme, la coupe (S, T) est telle que, pour tout sommet u , $\deg_{in}(u) \leq \deg_{out}(u)$ où $\deg_{in}(u)$ et $\deg_{out}(u)$ désignent respectivement le nombre d'arêtes incidentes de u qui ne traversent pas et qui traversent la coupe. En effet, si cela n'était pas le cas, alors il faudrait faire passer u de l'autre côté de la coupe. La cardinalité de la coupe renversée par l'algorithme ITER est de cardinalité $C = \frac{1}{2} \sum_{u \in V} \deg_{out}(u)$ car, en sommant les degrés extérieurs, on compte deux fois chaque arêtes de la coupe. Or,

$$OPT \leq |E| = \frac{1}{2} \sum_{u \in V} \deg(u) = \frac{1}{2} \sum_{u \in V} (\deg_{out}(u) + \deg_{in}(u)) \leq \sum_{u \in V} \deg_{out}(u) = 2C.$$

Question II.3. Proposez un algorithme polynomial glouton GL pour approximer MAXCUT et analyser son facteur d'approximation. (Cet algorithme considère les sommets un par un).

Réponse : GL peut procéder comme suit. On part de $(S, T) = (\emptyset, \emptyset)$ et on ajoute les sommets un par un, soit à S , soit à T . Lorsque l'on considère un nouveau sommet u , avec une coupe partielle courante (S, T) , on considère les deux coupes partielles $(S \cup \{u\}, T)$ et $(S, T \cup \{u\})$. On ajoute u à S si la coupe partielle $(S \cup \{u\}, T)$ contient plus d'arêtes que la coupe partielle $(S, T \cup \{u\})$, et à T sinon.

Soit $d_{in}(u)$ et $d_{out}(u)$ les nombres respectifs d'arêtes incidentes de u qui ne traversent pas et qui traversent la coupe partielle juste après que GL ait placé u dans l'un ou l'autre des deux

ensembles. La cardinalité C de la coupe construite par GL satisfait $C = \sum_{u \in V} d_{out}(u)$, et on a $d_{out}(u) \geq d_{min}(u)$ pour tout $u \in V$. Or,

$$OPT \leq |E| = \sum_{u \in V} (d_{out}(u) + d_{in}(u)) \leq 2 \sum_{u \in V} d_{out}(u) = 2C.$$

Le temps de cet algorithme est $O(\sum_u \deg(u)) = O(m)$.

Question II.4. Décrire un algorithme probabiliste (simple!) RAND pour MAXCUT tel que l'espérance de la cardinalité de la coupe calculée soit au moins $OPT/2$.

Réponse : L'algorithme RAND consiste simplement à placer chaque sommet, de manière indépendante, soit dans S , avec probabilité $\frac{1}{2}$, soit dans T , avec probabilité $\frac{1}{2}$. Soit u un sommet, et soit X_u la variable aléatoire indiquant le nombre d'arêtes incidentes à u dans la coupe obtenue par RAND. On a $X_u = \sum_{v \in N(u)} B_{u,v}$ où $N(u)$ dénote l'ensemble des voisins de u , et $B_{u,v}$ est la variable aléatoire valant 1 si $\{u, v\}$ est dans la coupe, et 0 sinon. On a $\mathbf{E}X_u = \sum_{v \in N(u)} \mathbf{E}B_{u,v}$ par linéarité de l'espérance. Or $\Pr[B_{u,v} = 1] = \frac{1}{2}$, donc $\mathbf{E}B_{u,v} = \frac{1}{2}$. Donc $\mathbf{E}X_u = \sum_{v \in N(u)} \mathbf{E}B_{u,v} = \frac{|N(u)|}{2} = \frac{\deg(u)}{2}$, toujours par la linéarité de l'espérance. Enfin, l'espérance de la cardinalité de la coupe calculée est $\mathbf{E}X$ où $X = \frac{1}{2} \sum_u X_u$, donc $\mathbf{E}X = \frac{1}{2} \sum_u \mathbf{E}X_u = \frac{1}{4} \sum_u \deg u = \frac{1}{2}|E| \geq \frac{1}{2}OPT$.

Question II.5. Montrer que le problème MINCUT peut être résolu en temps polynomial.

Réponse : (Notez que le sujet supposait implicitement que $S \neq \emptyset$ et $T \neq \emptyset$, sinon cette question serait triviale). Le théorème flot-max coupe-min dit que le flot maximum de s à t est égal à la cardinalité minimale d'une (s, t) -coupe. (Ici les capacités sont toutes égales à 1). Pour résoudre MINCUT, on peut donc procéder en calculant le flot maximum $f_{s,t}$ entre s et t , pour toutes les paires $(s, t) \in V \times V$. Le temps d'exécution d'un tel algorithme est $\frac{n(n-1)}{2}$ fois le temps de calcul d'un flot maximum, qui peut être effectué en temps polynomial. (Remarquez que la coupe minimum peut parfaitement être inférieure à toute coupe $(\{u\}, V \setminus \{u\})$ pour tout $u \in V$).

J Examen de rattrapage 27 février 2014, durée 1h30

Pour tout entier $n \geq 1$, on note $[n] = \{1, 2, \dots, n\}$. Soit p et q deux entiers strictement positifs. Le graphe orienté arc-pondéré $G_{p,q}$ est défini comme suit. Chaque noeud de $G_{p,q}$ est une paire (i, j) d'entiers avec $i \in [p]$ et $j \in [q]$. Pour $i \in [p-1]$ et $j \in [q]$, chaque noeud (i, j) possède un arc sortant vers chacun des noeuds $(i+1, k)$, pour tout $k \in [q]$. La longueur de l'arc $((i, j), (i+1, k))$ est notée $w(i, j, k) \geq 0$.

Les trois questions ci-dessous sont indépendantes.

Réponse : Les réponses aux trois questions suivent essentiellement les réponses aux questions similaires de l'examen de 1ère session, sauf la question 2 qui demande un peu plus de réflexion.

Question 1. Décrire un algorithme de programmation dynamique permettant de calculer la distance du noeud $(1, 1)$ au noeud (p, q) dans $G_{p,q}$ (c'est-à-dire la plus petite longueur d'un chemin de $(1, 1)$ à (p, q) où la longueur d'un chemin est la somme de la longueur de ses arcs).

Réponse : Notons $d_{i,j}$ la distance du noeud $(1, 1)$ au noeud (i, j) . On a $d_{1,1} = 0$ et $d_{1,j} = +\infty$ for every $j > 1$. Par ailleurs, pour $i > 1$, on a la récurrence

$$d_{i,j} = \min_{k \in [q]} \{d_{i-1,k} + w(i-1, k, j)\}.$$

On remplit le tableau D afin d'obtenir $D[i, j] = d_{i,j}$ comme suit :

début

```

 $D[1, 1] \leftarrow 0$ 
pour  $j = 2$  à  $q$  faire  $D[1, j] \leftarrow +\infty$ 
pour  $i = 2$  à  $p$  faire
    pour  $j = 1$  à  $q$  faire
         $D[i, j] \leftarrow \min_{k \in [q]} (D[i - 1, k] + w(i - 1, k, j))$ 
fin

```

Question 2 Donner une description sous forme d'un programme linéaire en nombre entier du calcul de la distance du noeud $(1, 1)$ et le noeud (p, q) dans $G_{p,q}$. Indication : utiliser une variable booléenne $x_{i,j,k}$ pour chaque arc $((i, j), (i + 1, k))$, et, à l'aide de ces variables, exprimer en particulier qu'un arc doit sortir du noeud $(1, 1)$, qu'un arc doit entrer au noeud (p, q) , que si un arc entre au noeud (i, j) alors un arc doit sortir du noeud (i, j) , et que si un arc sort du noeud (i, j) alors un arc doit entrer au noeud (i, j) .

Réponse : On cherche un chemin de $(1, 1)$ à (p, q) . Ce chemin doit donc partir de $(1, 1)$, traverser chaque niveau $i = 1, \dots, p - 1$ en traversant un arc de la forme $((i, j), (i + 1, k))$, et finalement finir en (p, q) . La minimisation va exprimer que l'on cherche un sous-graphe dont la somme de la longueur des arcs est le plus petite possible, et les contraintes auront pour objectif de forcer à ce que ce sous-graphe connecte bien le noeud $(1, 1)$ au noeud (p, q) . Le PLNE est alors le suivant :

$$\begin{aligned}
\text{minimiser} \quad & \sum_{(i,j,k) \in [p-1] \times [q] \times [q]} x_{i,j,k} \cdot w(i, j, k) \\
\text{sous les contraintes} \quad & (1) \quad x_{i,j,k} \in \{0, 1\} \text{ pour tout } (i, j, k) \in [p-1] \times [q] \times [q] \\
& (2) \quad \sum_{\ell \in [q]} x_{1,1,\ell} \geq 1 \\
& (2') \quad \sum_{\ell \in [q]} x_{1,j,\ell} = 0 \text{ pour tout } j \in [2, q] \\
& (3) \quad \sum_{\ell \in [q]} x_{i,j,\ell} \geq x_{i-1,k,j} \text{ pour tout } (i, j, k) \in [2, p-1] \times [q] \times [q] \\
& (4) \quad \sum_{\ell \in [q]} x_{i-1,\ell,j} \geq x_{i,j,k} \text{ pour tout } (i, j, k) \in [2, p-1] \times [q] \times [q] \\
& (5) \quad \sum_{\ell \in [q]} x_{p-1,\ell,q} \geq 1 \\
& (5') \quad \sum_{\ell \in [q]} x_{p-1,\ell,j} = 0 \text{ pour tout } j \in [q-1]
\end{aligned}$$

La contrainte (2) exprime que le chemin quitte le noeud $(1, 1)$, et la contrainte (2') exprime qu'aucun chemin ne quitte un noeud $(1, j)$ pour $j \neq 1$. La contrainte (3) exprime que si un arc du chemin entre au noeud (i, j) alors un arc du chemin doit sortir du noeud (i, j) . La contrainte (4) exprime que si un arc du chemin sort du noeud (i, j) alors un arc du chemin doit entrer au noeud (i, j) . Enfin, la contrainte (5) exprime que le chemin entre au noeud (p, q) , et la contrainte (5') exprime qu'aucun arc ne rentre dans un noeud (p, j) pour $j \neq q$. La fonction à minimiser, ainsi que les contraintes à satisfaire sont bien des fonctions linéaires des variables $x_{i,j,k}$.

On considère maintenant le problème MAXCUT dans un graphe arête-pondéré quelconque G , chaque arête e ayant un poids $w(e) \geq 0$. On ne cherche alors plus à maximiser le nombre d'arêtes dans la coupe (comme dans l'examen de 1ère session), mais la somme des poids des arêtes dans la coupe.

Question 3. Décrire un algorithme probabiliste PROB pour MAXCUT tel que l'espérance du poids de la coupe calculée soit au moins $OPT/2$.

Réponse : Notons que $OPT \leq \sum_{e \in E} w(e)$. On construit de manière probabiliste une coupe (S, T) . L'algorithme PROB consiste simplement à placer chaque sommet, de manière indépendante, soit dans S , avec probabilité $\frac{1}{2}$, soit dans T , avec probabilité $\frac{1}{2}$. Soit u un sommet, et soit X_u la variable aléatoire indiquant la somme des poids des arêtes incidentes à u dans la coupe obtenue par PROB. On a

$$X_u = \sum_{v \in N(u)} w(\{u, v\}) \cdot B_{u,v}$$

où $N(u)$ dénote l'ensemble des voisins de u , et $B_{u,v}$ est la variable aléatoire valant 1 si $\{u, v\}$ est dans la coupe, et 0 sinon. Par linéarité de l'espérance, on a

$$\mathbf{E}X_u = \sum_{v \in N(u)} w(\{u, v\}) \cdot \mathbf{E}B_{u,v}.$$

Or $\Pr[B_{u,v} = 1] = \frac{1}{2}$, et donc $\mathbf{E}B_{u,v} = \frac{1}{2}$. On en déduit ainsi que

$$\mathbf{E}X_u = \frac{1}{2} \sum_{v \in N(u)} w(\{u, v\}).$$

Enfin, l'espérance $\mathbf{E}X$ du poids $X = \frac{1}{2} \sum_{u \in V} X_u$ de la coupe obtenue par PROB satisfait, par linéarité de l'espérance,

$$\mathbf{E}X = \frac{1}{2} \sum_{u \in V} \mathbf{E}X_u = \frac{1}{4} \sum_{u \in V} \sum_{v \in N(u)} w(\{u, v\}) = \frac{1}{2} \sum_{e \in E} w(e) \geq \frac{1}{2} OPT.$$