

Cours de Complexité et Algorithmique

Partie Complexité

Avant-propos

La complexité est un des enseignements de base en deuxième et troisième cycle d'informatique théorique (Ecoles d'ingénieurs, Masters Professionnelles ou Recherche).

Cet enseignement s'adresse à des étudiants déjà familiarisés avec des méthodes de programmation et l'algorithmique, et a pour but d'élargir leurs connaissances fondamentales en leur apprenant certaines limites inhérentes à l'informatique. Cette matière permet aussi de mieux connaître et comprendre le développement de l'informatique et ses bases théoriques.

L'objectif principal de ce manuscrit est de rendre le sujet lisible pour les futurs ingénieurs, tout en gardant un volume assez réduit. Bien entendu, il ne constitue qu'une introduction.

Je tiens à remercier mes collègues François Doré, Mélanie Ducoffe, Bruno Martin, Dorian Mazauric, Christophe Papazian et Stéphane Perennes, qui ont participé à cet enseignement et ont eu la patience et la gentillesse de relire diverses versions de ce manuscrit, ainsi qu'à Matti Schneider qui a corrigé beaucoup d'erreurs en 2010. Leurs remarques ont permis d'améliorer la lisibilité. Cependant, il est plus que probable qu'il en reste. J'espère que les lecteurs me les feront remarquer. Qu'ils en soient remerciés à l'avance.

Sophia-Antipolis, octobre 2020

CHAPITRE 1

Introduction

Le programme de la deuxième année de l'école comporte deux cours qui traitent des notions de complexité et des modèles de calcul. Dans le cours intitulé «Calculabilité» nous traitons la notion de fonction calculable du point de vue théorique. Le but de ce cours, plus proche de la pratique, est d'établir une notion de fonction «pratiquement calculable», en tenant compte du temps de calcul nécessaire.

Pour mieux situer le problème, prenons un exemple pratique. Considérons un problème de tri selon l'ordre lexicographique de n mots d'un dictionnaire (tous les mots étant de longueur 20 caractères). Notre problème est de taille $20n$. Le temps de calcul nécessaire (dû essentiellement aux comparaisons nécessaires) dépendra de la méthode choisie, et sera selon la méthode

- $2n^5$ pour la méthode A (très inefficace),
- $20n^2$ pour la méthode B (peu efficace),
- $40n\log_2 n$ pour la méthode C (assez efficace)
- $80n$ pour la méthode D (très efficace, tient compte de la nature des données).

Si on suppose, que notre machine peut effectuer 100 000 000 instructions par seconde (machine avec un processeur 100 MHz), alors un problème de taille 10 000 nécessitera

- plus de 63 siècles pour la méthode A,
- 20 secondes pour la méthode B,
- 53 millisecondes pour la méthode C,
- 8 millisecondes pour la méthode D.

Par contre, si on se fixe un temps de calcul de 10 minutes, on peut traiter des problèmes de taille au plus

- 125 pour la méthode A.
- 54 773 pour la méthode B,
- 58 154 443 pour la méthode C,
- 750 000 000 pour la méthode D.

Cet exemple illustre la *mauvaise influence* d'un algorithme peu efficace sur la taille des problèmes qu'on peut traiter.

Dans un cadre plus général on effectue la même démarche mais, au lieu des calculs exacts, on calcule par ordre de grandeur. C'est aussi l'occasion de considérer comme base une machine plus réaliste, capable d'effectuer de l'ordre de 10^{10} opérations par seconde (10 GHz).

Ainsi, on peut estimer l'ordre de grandeur de la taille des problèmes qu'on peut traiter, si on dispose d'algorithmes nécessitant un nombre d'opérations de l'ordre de n , n^2 , n^3 , n^4 , n^5 , 2^n , 3^n et $n!$ respectivement et si l'on veut obtenir le résultat au bout d'au plus 1/10 de seconde, 10 minutes ou une journée :

Complexité	0.1 seconde	10 minutes	1 jour	1 mois
n	1 000 000 000	6 000 000 000 000	864 000 000 000 000	25 920 000 000 000 000
n^2	31 623	2 449 490	29 393 877	160 996 894
n^3	1 000	18 171	95 244	295 945
n^5	63	359	971	1 917
2^n	30	42	50	55
3^n	19	27	31	34
$n!$	12	15	17	18

Les données de ce tableau nous montrent qu'on a intérêt à éviter les algorithmes exponentiels. Bien évidemment il faut qu'on tienne compte de l'évolution des machines. Ainsi, en supposant que les machines actuelles nous permettent de traiter un problème de taille N , le tableau ci-dessous résume la taille des problèmes qu'on peut espérer de traiter par une machine 1000 fois plus rapide (ce qui peut correspondre à une future machine de la prochaine décennie) :

Complexité	taille actuelle	taille future
$O(n)$	N	$N * 1000$
$O(n^2)$	N	$N * 32$
$O(n^3)$	N	$N * 10$
$O(n^5)$	N	$N * 4$
$O(2^n)$	N	$N + 10$
$O(3^n)$	N	$N + 7$

Ainsi on peut conclure que même les futurs progrès de nos machines ne peuvent pas améliorer sensiblement la situation, si nos algorithmes sont exponentiels.

Il est donc naturel d'essayer de montrer qu'un problème qui nécessite du temps et/ou un espace mémoire exponentiel est pratiquement intraitable. Cette approche s'avère ne pas être praticable, car trouver quelle est la complexité d'un problème est très difficile en général (ce qui explique le faible nombre de problèmes ainsi traités au cours des études de deuxième cycle). La difficulté provient de la nécessité de traiter tous les algorithmes possibles.

Cependant, dans la pratique on a besoin d'un tel outil. Citons un exemple donné par M. Garey et D. Johnson dans l'introduction de leur livre (traduction libre) :

Vous êtes le chef du service informatique d'une entreprise. Un nouveau problème se pose, et on fait appel à vous pour construire un programme résolvant le problème. Vous vous mettez au travail, mais au bout de quelques jours, voire quelques semaines de travail vous n'avez toujours pas de solution raisonnable au problème, sinon d'essayer toutes les possibilités, ce qui revient à l'utilisation d'un algorithme exponentiel, et n'est donc pas réaliste. Vous êtes conscient que, si votre réponse est négative, votre emploi sera en jeu. Il est donc souhaitable de trouver un autre moyen de s'en sortir. L'idéal serait de pouvoir montrer qu'il ne peut pas y avoir de solution au problème, mais la preuve en est trop difficile, et vous ne savez pas la faire. C'est ici que la théorie des problèmes NP-complets intervient. Elle permet d'affirmer que, même si vous ne savez pas résoudre le problème, vous licencier ce n'est pas une solution pour l'entreprise, car personne, et donc votre remplaçant en particulier, ne pourra résoudre efficacement le problème.

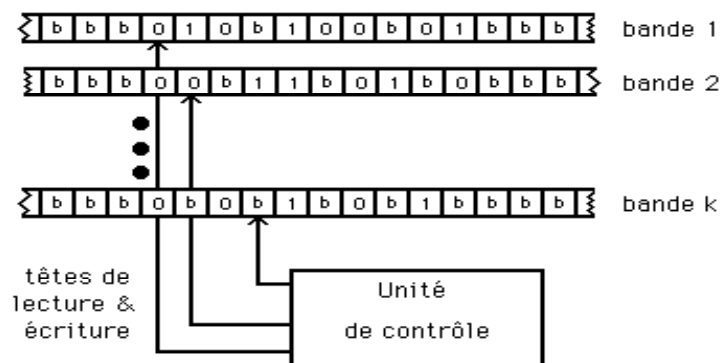
CHAPITRE 2

Les machines de Turing : un modèle de calcul

2.1 Description

Ce modèle de calcul date de 1936 et est dû à Turing. Historiquement, c'est le premier modèle «plutôt informatique» et sûrement le modèle le plus connu.

Une machine de Turing est constituée d'une ou plusieurs bandes de lecture/écriture et d'une unité de contrôle finie (une sorte d'automate fini). Chaque bande est divisée en cases, chaque case contenant un symbole d'un alphabet fini, fixé à priori. Les bandes sont infinies des deux côtés (il y a un nombre infini de cases à gauche et à droite de toute case). On dispose de têtes de lecture et écriture permettant, comme leur nom l'indique, de lire le contenu d'une case et d'écrire un des symboles de l'alphabet dans cette case.



La machine obtenue est une sorte d'automate, mais disposant de mémoire infinie (par écriture sur les bandes). Donnons une définition formelle des machines de Turing :

Définition : une machine de Turing à k bandes est décrite par un septuplet $(Q, T, I, \delta, B, q_0, q_f)$. Les composantes de ce septuplet sont :

- Q - un ensemble fini d'états de la machine.
- T - un ensemble fini de symboles qu'on peut utiliser sur les bandes.
- I - un sous-ensemble de l'ensemble des symboles, les symboles de données.
- δ - la fonction de transition : c'est une fonction de $Q \times T^k$ vers $Q \times (T \times \{G, D, S\})^k$.
- B - un symbole, désignant le blanc, appartenant à $T \setminus I$.
- q_0 - l'état initial.
- q_f - l'état final.

La fonction de transition décrit le fonctionnement de la machine. En cas de lecture d'une certaine suite de données, on la remplace par les nouvelles valeurs données par la fonction de transition et on déplace les têtes de lecture/écriture de manière correspondante sur les différentes bandes. Ainsi G , D et S dénotent les déplacements à gauche, à droite ou stationnaire (pas de déplacement).

Le temps de calcul d'une machine de Turing est une fonction de complexité $T(n)$, qui dépend

de n , la longueur des données en cases (bits ?). Formellement on définit $T(n)$ comme le nombre maximum de transitions pour une entrée de longueur n .

Avant de continuer, nous donnons un exemple de machine de Turing. Il s'agit d'une machine à 3 bandes, qui réalise l'addition binaire de deux nombres inscrits respectivement sur la bande 1 et 2 et dont le résultat sera inscrit sur la 3^{ième} bande (pour raison de simplicité, tout en base 2). On aura $Q=\{q_0, q_1, q_f\}$, $T=\{0,1,B\}$, $I=\{0,1\}$. La table de transitions est comme suit :

état	lecture			écriture			déplacement			nouvel état	remarques
	b ₁	b ₂	b ₃	b ₁	b ₂	b ₃	b ₁	b ₂	b ₃		
q ₀	0	0		B	B	0	G	G	G	q ₀	sans retenue
	0	1		B	B	1	G	G	G	q ₀	
	1	0		B	B	1	G	G	G	q ₀	
	1	1		B	B	0	G	G	G	q ₁	
	0	B		B	B	0	G	S	G	q ₀	
	1	B		B	B	1	G	S	G	q ₀	
	B	0		B	B	0	S	G	G	q ₀	
	B	1		B	B	1	S	G	G	q ₀	
q ₁	B	B		B	B	B	S	S	S	q _f	avec retenue
	0	0		B	B	1	G	G	G	q ₀	
	0	1		B	B	0	G	G	G	q ₁	
	1	0		B	B	0	G	G	G	q ₁	
	1	1		B	B	1	G	G	G	q ₁	
	0	B		B	B	1	G	S	G	q ₀	
	1	B		B	B	0	G	S	G	q ₁	
	B	0		B	B	1	S	G	G	q ₀	
	B	1		B	B	0	S	G	G	q ₁	
	B	B		B	B	1	S	S	G	q _f	

On n'a pas précisé la lecture sur la troisième bande, car il s'agit de la bande résultat, supposée vide (que des blancs) au départ.

Pour finir de décrire le fonctionnement de cette machine, il faut préciser qu'au début les têtes se trouvent sur le bit le plus à droite (le moins significatif) des deux nombres, et qu'à l'issue du calcul, les deux bandes sont vides et la troisième bande contient le résultat, avec la tête de lecture/écriture à gauche du résultat.

En ce qui concerne la complexité de cette machine, $T(n)$, elle est en $O(n)$. En effet il suffit de lire les données pour finir le calcul. On peut remarquer que cette complexité est optimale, car on ne peut pas avoir une complexité inférieure (il faut en effet lire le plus long des deux entiers).

2.2 Variantes de la machine

Dans la littérature on trouve plusieurs variantes de la machine de Turing. On verra que tous ces modèles sont équivalents du point de vue de leur puissance de calcul. Les différents modèles se distinguent surtout selon les trois critères suivants :

- le nombre de bandes
- la nature des bandes
- le déterminisme ou le non-déterminisme de la machine.

• **Le nombre de bandes** : on distingue les machines selon le nombre de bandes utilisés. Principalement on distinguera les machines à une bande de celles à plusieurs bandes.

- **La nature des bandes** : on distingue les machines utilisant des bandes infinies des deux côtés de celles utilisant une bande infinie d'un côté seulement.

- **Le déterminisme ou le non-déterminisme de la machine** : le déterminisme de la machine est défini par le nombre de choix (par couple (*état*, *lecture*)) qu'on peut avoir dans la table de transitions. Dans le cas déterministe il s'agit bien sûr d'un choix unique. Le temps de calcul est défini comme la plus courte suite de choix qui mène au résultat. En ce qui concerne le cas non-déterministe on trouve plusieurs modèles dans la littérature. Il existe des différences entre ces modèles, selon que l'on permet d'avoir plusieurs transitions sans aucune relation entre elles ou si on impose (dans le cas le plus restrictif) que deux transitions peuvent différer seulement sur le nouvel état. Cette version, qui semble *moins puissante*, est aussi puissante que les autres, car en introduisant un nombre suffisant de nouveaux états, on peut transformer toute machine non-déterministe en une de ce type. Une version qui donne l'impression d'être encore plus faible, impose de plus un nombre borné de choix (et dans sa version la plus stricte au plus deux choix). Là encore, il s'agit de versions équivalentes, car l'introduction de nouveaux états en nombre suffisant permet de réduire le nombre de choix.

Dans la littérature on trouve aussi d'autres distinctions. Ainsi, par exemple, il existe un modèle de machine à plusieurs bandes, où les déplacements sont les mêmes sur toutes les bandes. Ce modèle a encore la même puissance que les autres, car de façon évidente il est au moins aussi puissant que le modèle de la machine à une bande, et au plus aussi puissant que le modèle avec les déplacements indépendants et, comme on le verra dans la suite, ces deux modèles sont équivalents.

Une autre différence entre les modèles concerne le résultat. En effet, on a plusieurs manières de définir des fonctions et en conséquence on a plusieurs modèles de machines. Ainsi, on peut parler de machines qui «calculent», comme celle de l'exemple précédent. On peut aussi parler de machines qui reconnaissent un langage. Considérons, par exemple, une machine qui reçoit comme données trois nombres en base 2 et accepte les données si le troisième est la somme des deux précédents (on peut la construire facilement à partir de l'exemple donné). Comme ces deux approches sont fondamentalement équivalentes, on utilisera les deux en alternance.

Comme le non-déterminisme est une notion non évidente, nous souhaitons faire quelques remarques sur la manière dont on peut interpréter cette notion. En effet, comme la seule autre évocation du non-déterminisme dans les programmes de l'école concerne les automates finis non-déterministes, le lecteur peut avoir l'impression qu'il s'agit d'un modèle trivialement équivalent au modèle déterministe. Sans mettre en cause l'équivalence, dont la preuve est donnée dans ce qui suit, nous proposons deux interprétations intuitives à cette notion. Mais remarquons tout d'abord, qu'il s'agit d'un modèle théorique, qui ne peut pas exister dans la pratique (en effet, le comportement de nos machines reste déterministe même dans les cas qui nous paraissent inexplicables ...).

- Une première interprétation intuitive consiste à considérer une machine non-déterministe comme une machine fournie avec un *conseiller* en non-déterminisme. C'est ce conseiller qui dicte les choix au moment où la machine doit en faire, et ainsi on peut supposer que les conseils *sont bons*. Ceci justifie la mesure utilisée pour le temps de calcul, c.a.d. le temps d'une plus courte exécution.
- Une autre interprétation consiste à considérer que la machine non-déterministe n'est rien d'autre qu'une machine parallèle. Chaque fois que plusieurs choix de transition se présentent, le processeur lance *toutes* les exécutions possibles, de la suite, sur différents processeurs. Si le modèle peut ainsi paraître réalisable, ce n'est toujours le cas, car la réalisation nécessiterait un

nombre non borné de processeurs.

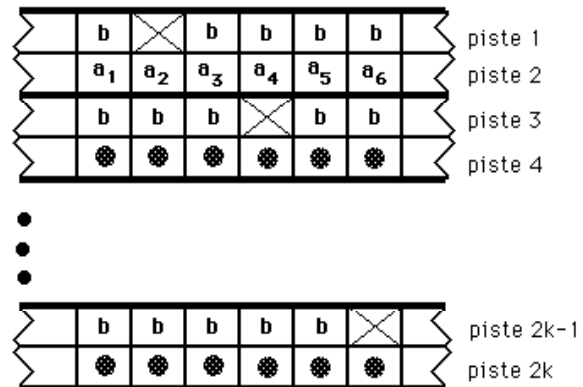
2.3 Equivalence des différents modèles

Montrons d'abord que les machines à une ou plusieurs bandes sont équivalentes.

Théorème : Si le langage L est accepté par une machine de Turing T à k bandes, alors il existe une machine de Turing T' à une bande pour reconnaître L .

Preuve : Pour la preuve nous introduisons la notion de piste. Construisons en effet T' à une bande, mais à $2k$ pistes, simulant T .

Ainsi pour décrire ce qui se passe sur la bande i on dispose de deux pistes. La piste $2i-1$, a des blancs partout, à part la case sur laquelle se trouve la tête de lecture/écriture. D'autre part la piste $2i$ contient exactement l'information de la bande i . Pour que la bande de T' puisse contenir cette information sur les $2k$ pistes, on aura un alphabet plus grand, permettant le codage de chaque $2k$ -uplet possible en une seule lettre.



Pour simuler le fonctionnement de T , la machine T' fonctionne comme suit :

- on «visite» toutes les cases marquées pour lire l'information se trouvant sur chacune des bandes. L'information lue est conservée par l'état.
- on visite à nouveau les cases marquées pour effectuer les changements d'écriture et de marquage dues à la transition.

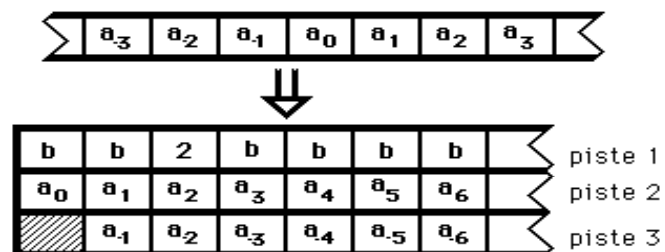
Remarquons que pour trouver les cases marquées, les états doivent aussi contenir l'information sur l'endroit où se trouve la tête (à gauche ou à droite) sur chaque bande.

Ainsi la machine T' obtenue utilise un grand alphabet et un grand nombre d'états pour simuler T . □

La même méthode d'utilisation de pistes permet de prouver le résultat suivant :

Théorème : Si le langage L est accepté par une machine de Turing T à 1 bande infinie des deux côtés, alors il existe une machine de Turing T' à 1 bande infinie d'un seul côté pour reconnaître L .

Preuve : La preuve se fait par simulation, en utilisant une bande à trois pistes, avec la case i correspondant à la case i et à la case $-i$ respectivement, comme sur la figure. La première piste sert comme indicateur, pour savoir si on se trouve (pour la simulation) sur la piste 2 ou 3. □



On peut généraliser ce résultat :

Théorème : Si le langage L est accepté par une machine de Turing T à k bandes infinies des deux côtés, alors il existe une machine de Turing T' à k bandes infinies d'un seul côté pour reconnaître L .

La dernière variante dont nous avons à prouver l'équivalence, est celle des machines de

Turing non-déterministes. De façon évidente, il suffit de montrer que tout langage accepté par une machine non-déterministe peut être accepté par une machine déterministe, (l'inverse est vrai, car une machine déterministe est en particulier une non-déterministe où le nombre de choix est toujours au plus 1). Comme on a déjà vu que les différentes variantes de machines déterministes sont équivalentes, on peut l'énoncer de la façon suivante :

Théorème : Si le langage L est accepté par une machine de Turing non-déterministe T à une bande, alors il existe une machine de Turing déterministe T' à 3 bandes pour accepter L .

Preuve : Comme il faut simuler le non-déterminisme, l'idée est de générer dans un ordre fixé à l'avance tous les choix possibles, jusqu'à ce qu'on arrive à l'état d'acceptation. Ceci correspond à un parcours de l'arborescence qui comporte tous les choix. L'idée d'un parcours en profondeur ne peut pas être retenue, car elle peut poser des problèmes, notamment dans le cas des choix qui font «boucler» la machine. Par contre, un parcours en largeur peut être utilisé. Un problème que l'on rencontre lors de tout parcours est l'impossibilité de revenir en arrière. La simulation d'un parcours en largeur dans ces conditions devient une suite de parcours partiels, de la racine vers les différents sommets, dans l'ordre qui correspond à l'ordre d'un parcours en largeur. Ainsi, si on note les suites de choix retenus, on obtient des mots rangés selon l'ordre lexicographique.

La simulation se déroule comme suit : Supposons que la machine T ait au maximum r choix. Le fonctionnement de la machine T' est comme suit :

- i) On génère un mot de $\{0,1,2,\dots,r-1\}^*$ sur la bande 3.
- ii) On recopie les données de la bande 1 sur la bande 2.
- iii) On exécute le programme de T en travaillant sur la bande 2, en prenant comme choix celui indiqué par le bit correspondant sur la bande 3 (sur laquelle on avance). Si on arrive à un état d'acceptation on arrête, sinon on génère une nouvelle suite sur la bande 3, on efface la bande 2 et on recommence en ii).

Pour s'assurer qu'on teste toutes les possibilités, on génère toutes les suites sur la bande 3 de façon systématique. Ceci en commençant par une bande 3 où il y a un 0, et rajoutant chaque fois 1 (calcul en base r). \square

2.4 Le théorème de l'accélération

Un résultat très important qu'on peut prouver à l'aide de ce modèle est le théorème de l'accélération. Ce théorème donne une justification de plus à l'usage de l'ordre de grandeur asymptotique comme mesure de complexité des algorithmes.

Théorème (de l'accélération - speed up) : Si L est accepté par une machine de Turing à k bandes ($k > 1$) en temps $T(n)$ (avec $\lim T(n)/n = \infty$), alors pour tout nombre réel $c > 0$ il existe une machine de Turing à k bandes acceptant L en temps $cT(n)$.

Comme exemple d'application de ce théorème, notons que si on peut calculer quelque chose en temps $n \log n$ alors, on peut obtenir le même résultat en temps $1/2 n \log n$, ou encore en temps $1/10 n \log n$.

Preuve : La preuve se fait par construction d'une machine de Turing qui simule la machine originale. La machine M' est construite comme suit :

I) On encode les données, de façon que m cases de la machine originale soient codées par une seule lettre - donc une seule case, dans la nouvelle machine. Cette opération se fait en temps linéaire ($2n$ par exemple). Notons que le temps linéaire est possible, car il s'agit d'une machine à plusieurs bandes !

II) Le codage obtenu permet d'effectuer plusieurs transitions à la fois, car une lecture donne l'information de m cases de la machine simulée et donc on peut effectuer tout le travail imposé par ces m lectures. Par contre, il est impossible d'éliminer ainsi les effets de bords, c.a.d. le cas où la machine simulée effectue les aller - retours entre deux cases qui impliquent des allers - retours entre deux «blocs» de m cases. Pour éliminer ce problème, nous allons effectuer la lecture de trois cases voisines (ce qui correspond à la lecture de $3m$ cases de la machine simulée), et ainsi, nous pouvons assurer que chaque «phase de base» correspond à au moins m transitions de la machine simulée.

Pendant la simulation, la machine M' aura les «phases de base» suivantes :

- 1) Lecture et déplacement à gauche
- 2) Lecture et déplacement à droite
- 3) Déplacement à droite
- 4) Lecture et déplacement à gauche

Arrivés à ce point on se trouve sur la case de départ et on connaît le contenu des deux cases voisines. Ainsi, comme on souhaitait, on connaît le contenu de $3m$ cases de la machine simulée, ce qui permet d'effectuer d'un coup toutes les transitions que la machine originale effectue, jusqu'au moment où elle quitte ces $3m$ cases. Selon que l'on quitte les $3m$ cases vers la droite ou vers la gauche on effectue les opérations suivantes.

- 5) Ecriture et déplacement à gauche (droite)
- 6) Ecriture et déplacement à droite (gauche)
- 7) Déplacement à droite (gauche)
- 8) Ecriture et déplacement à droite (gauche)

Ainsi une phase de base comprend 8 pas de calcul de la nouvelle machine et correspond à plus de m pas de calculs de la machine originale. Ainsi la complexité de la nouvelle machine sera

$$T'(n) \leq 2n + \frac{8T(n)}{m}.$$

Maintenant il faut choisir m de façon à obtenir $T'(n) \leq cT(n)$. Soit $m > 16/c$. Alors on a $8T(n)/m < c/2 T(n) < c T(n) - 2n$ (pour n assez grand). \square

En utilisant les mêmes méthodes de preuve on obtient le résultat suivant :

Théorème : Si le langage L est accepté par une machine de Turing à k ($k > 1$) bandes en temps $T(n)$, $\lim T(n)/n = \infty$, alors L est accepté par une machine de Turing à une bande M' en temps au plus $T^2(n)$.

Preuve : D'abord soit M'' une machine à k bandes en temps $1/\sqrt{2} T(n)$. Ceci existe, comme corollaire du Théorème de l'accélération. La machine M' simulera le fonctionnement de M'' . Comme on a déjà vu, la simulation se fait en utilisant une machine à 1 bande, mais $2k$ pistes. Pour simuler un pas de calcul de M'' , M' doit «visiter» deux fois toutes les cases marquées, une fois pour la lecture et une deuxième fois pour l'écriture. Ainsi la distance parcourue est d'au plus $2T''(n)$ chaque fois. En tout il y aura au plus $T''(n)$ telles phases, donc la complexité est d'au plus $2T''^2(n) = T^2(n)$. \square

2.5 Les suites de passages

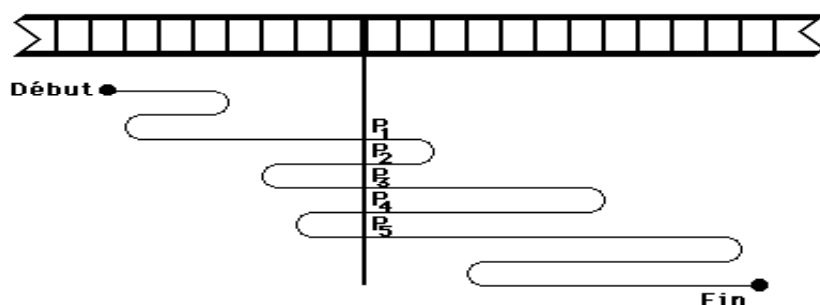
Le fonctionnement d'une machine de Turing est ainsi dirigé par une unité de contrôle fini, ressemblant à un automate fini. Ainsi d'un certain point de vue, le fonctionnement d'une

machine de Turing est comme celui d'un automate fini. Pour étudier ce comportement nous définissons la notion de *suite de passages* (*crossing sequence*).

Soit T une machine de Turing à une bande. On regardera pendant l'exécution le nombre de fois que la tête se déplace entre les cases i et $i+1$ (la ligne épaisse de la figure). On notera

- par p_{2k-1} l'état de T lorsque la tête est sur la case $i+1$ et qu'il s'agit du $k^{\text{ième}}$ passage de la case i à la case $i+1$.
- par p_{2k} l'état de T lorsque la tête est sur la case i et qu'il s'agit du $k^{\text{ième}}$ passage de la case $i+1$ à la case i .

Quand le calcul est fini, la suite (p_1, p_2, \dots, p_q) ainsi définie s'appelle la suite des passages en i (ou entre i et $i+1$). Cette suite peut être de longueur quelconque, paire ou impaire, même nulle.



Lemme 1 : Soit $w = a_1 a_2 \dots a_n$ ($n \geq 3$) accepté par une machine de Turing T à une bande. Soient i et j des entiers tels que $1 \leq i < j \leq n-1$. Si les suites des passages en i et en j sont égales, lorsqu'on exécute T sur w , alors le mot $u = a_1 \dots a_i a_{j+1} \dots a_n$ est accepté par T .

Preuve : Soit p_1, p_2, \dots, p_p la suite des passages en i et en j . Notons les trois parties du mot w par $w_d = a_1 \dots a_i$, $w_m = a_{i+1} \dots a_j$ et $w_f = a_{j+1} \dots a_n$. L'idée est que les comportements de T pour $w_d w_m w_f$ et pour $w_d w_f$ se ressemblent. En effet T fait la même chose pour $w_d w_m w_f$ et pour $w_d w_f$ jusqu'au moment où la tête quitte w_d (dans les deux cas en état p_1). De la même façon T entre en w_f la première fois dans l'état p_1 , pour $w_d w_m w_f$ et pour $w_d w_f$. On peut continuer ainsi pour montrer que chaque fois que la tête arrive en w_f de la gauche ou en w_d de la droite il y a identité pour $w_d w_m w_f$ et pour $w_d w_f$. \square

Remarques :

- 1) Si le nombre de passages est impair alors on finit dans w_f et si le nombre de passages est pair alors on finit dans w_d . En aucun cas on ne finit sur w_m .
- 2) Le résultat obtenu peut être généralisé pour aboutir à un analogue du lemme de l'étoile de la théorie des langages.

Lemme 2 : Soient $w = a_1 a_2 \dots a_n$ et $u = b_1 b_2 \dots b_m$ acceptés par une machine de Turing T à une bande. Si la suite des passages entre a_i et a_{i+1} est égale à la suite des passages entre b_j et b_{j+1} , lorsqu'on exécute T sur w et sur u , alors les mots $a_1 \dots a_i b_{j+1} \dots b_m$ et $b_1 \dots b_j a_{i+1} \dots a_n$ sont aussi acceptés par T .

Preuve : comme la preuve du Lemme 1. \square

Soit T une machine de Turing, de complexité en temps bornée par la fonction $t(n)$. Soit w un mot de longueur n , accepté par T . Soit l_i la longueur de la suite des passages en i . Si on considère l_i pour toutes les cases i , sa valeur est nulle, sauf pour un nombre fini de cases.

Lemme 3 : $\sum_i l_i \leq t(n)$.

Preuve : comme à chaque passage on fait un pas de calcul, la somme des longueurs des suites des passages ne peut dépasser le nombre de pas. \square

Les suites des passages constituent un outil pratique pour l'étude de certaines propriétés des machines de Turing. Un tel exemple d'application est donné dans le paragraphe suivant.

2.6 La reconnaissance des palindromes impairs

Rappelons ce qu'est un palindrome impair : un mot $a_1 a_2 \dots a_n$ de longueur impaire d'un langage, tel que $a_1 a_2 \dots a_n = a_n a_{n-1} \dots a_1$.

Le problème alors est le suivant : quel est la complexité minimale nécessaire pour la reconnaissance d'un palindrome impair ?

La réponse est facile dans le cas d'une machine à deux bandes. En effet, il est clair que n est une borne inférieure, car il faut lire le mot. D'autre part, on peut reconnaître un palindrome en temps $O(n)$, en recopiant d'abord le mot sur la deuxième bande et relisant les deux copies du mot dans le sens inverse en comparant les bits.

Dans le cas d'une machine à une seule bande la question s'avère plus difficile. Un algorithme simple consiste à comparer la première et la dernière lettre en les effaçant, et ainsi de suite. Cet algorithme nécessite un temps $O(n^2)$ à cause des déplacements. Le problème est alors d'essayer de faire mieux ou de prouver que c'est bien la meilleure solution (à un facteur constant près).

Le choix de traiter les palindromes impairs permettra de simplifier les preuves. Les méthodes utilisées permettent bien sûr de prouver les mêmes résultats pour les palindromes pairs ou pour les palindromes tout court.

Soit L le langage des palindromes impairs sur l'alphabet $\{0,1\}$,

$$L = \{wxw^i \mid w \text{ mot de } \{0,1\}^*, x \text{ lettre de } \{0,1\}, w^i \text{ étant le mot inverse de } w\}$$

Soit n la longueur d'un mot. On prouvera le résultat suivant :

Théorème : Si T est une machine de Turing à une bande, reconnaissant L , de complexité en temps $t(n)$, alors il existe une constante $c > 0$, telle que pour une infinité de valeurs de n , $t(n) \geq cn^2$.

Proposition 1 : Soient $w_1 w_2 x w_2^i w_1^i$ et $w_3 w_4 x w_4^i w_3^i$ deux mots de L , tels que w_1 et w_3 sont de même longueur, mais différents. Alors la suite des passages entre w_1 et w_2 est différente de la suite des passages entre w_3 et w_4 .

Preuve : sinon, en appliquant le lemme 2, $w_1 w_4 x w_4^i w_3^i$ serait accepté lui aussi, ce qui est faux. \square

Preuve du théorème : Soit n impair. Notons par L_n les palindromes de longueur n et soit $k = (n-1)/2$. Un mot de L_n s'écrit wxw^i avec w de longueur k . Il y a 2^k mots distincts de longueur k et donc 2^{k+1} mots dans L_n . Pour une case i , $i \leq k$, soit $p(i)$ la longueur moyenne des suites des passages en i , sur tous les mots de L_n . Au moins la moitié des mots de L_n ont des suites des passages en i de longueur au plus $2p(i)$.

Soit s le nombre d'états distincts de T . Le nombre de suites des passages en i , de longueur au plus $2p(i)$, est au plus $1 + s + s^2 + \dots + s^{2p(i)} < s^{2p(i)+1}$ ($s \geq 2$). Comme il y a au moins 2^k (la moitié) mots de L_n dont les suites des passages en i ont une longueur d'au plus $2p(i)$ et qu'il y a au plus $s^{2p(i)+1}$ suites des passages distinctes de longueur au plus $2p(i)$, en i , il y a au moins $2^k / s^{2p(i)+1}$ mots de L_n ayant la même suite des passages en i (et dont la longueur est au

plus $2p(i)$).

Pour la suite, soit $u = w_1 w_2 x w_2^i w_1^i$ un mot de L_n ayant cette suite des passages en i . Comme corollaire de la proposition 1, tout mot de L_n ayant cette suite de passages est de la forme $w_1 z x z^i w_1^i$ avec z de longueur $k-i$. D'autre part il y a 2^{k-i+1} mots distincts de cette forme dans L_n .

Ainsi on obtient $2^k / s^{2p(i)+1} \leq 2^{k-i+1}$, d'où $s^{2p(i)+1} \geq 2^{i-1}$, d'où on obtient la minoration $p(i) \geq (i-1)/3 \log_2 s$.

Comme ceci est vrai pour tout i , on obtient

$$\sum_{1 \leq i \leq k} p(i) \geq \sum_{1 \leq i \leq k} (i-1)/3 \log_2 s \geq 1/3 \log_2 s (n-1)(n-3)/8.$$

En utilisant le lemme 3, on a, pour tout mot u de L_n

$$\sum_i l_i(u) \leq t(n),$$

donc

$$\sum_u \sum_i l_i(u) \leq 2^{k+1} t(n)$$

donc

$$\sum_i 1/2^{k+1} \sum_u l_i(u) \leq t(n)$$

d'où en particulier

$$\sum_i p(i) \leq t(n).$$

Ainsi on peut conclure que

$$t(n) \geq cn^2$$

pour c suffisamment petit et n suffisamment grand (par exemple $c \leq 1/100 \log_2 s$ et $n \geq 5$).

□

CHAPITRE 3

Les classes P et NP

3.1 Quelques notions de complexité

Pour parler de complexité, il faut qu'on précise un certain nombre de termes. Ici, on ne parlera que de complexité en temps, mais on rencontre exactement la même terminologie si on considère la mémoire (espace).

Pour décrire la complexité d'un algorithme, on dit qu'il est en $O(n)$, $O(n^2)$, etc. Est-ce que c'est suffisamment clair ? Le premier problème à se poser est de fixer n . De manière évidente, n mesure d'une certaine façon la taille du problème, mais qu'est-ce que la taille d'un problème ? Pour se fixer un modèle on est amené d'abord à se fixer un modèle de calcul. Le modèle adopté est celui des machines de Turing. Ainsi, le problème de la taille des données est aussi résolu, car il s'agit du nombre de cases nécessaire pour coder le problème. Il est vrai que cette taille dépend de l'alphabet de la machine mais cela n'a pas trop d'importance, car le passage d'un alphabet de taille t à un alphabet de deux lettres n'augmente la taille des données que d'un facteur constant ($\log_2 t$). Le choix de la machine de Turing comme modèle de calcul a un deuxième avantage : les notions de complexité en temps et de complexité en espace deviennent très simples. En effet la complexité en temps compte le nombre de pas de calculs, tandis que la complexité en espace compte le nombre de cases utilisées sur la bande. Par ailleurs, dans ce cas nous pouvons remarquer que la complexité en espace est toujours bornée par la complexité en temps.

Un autre problème concerne la manière d'énoncer les problèmes. Nous utiliserons un formalisme simple, apte à ne traiter que des problèmes de décision, forme à laquelle il est presque toujours possible de se ramener. Ainsi la façon standard d'énoncer un problème sera composée de trois parties :

- le nom du problème
- les données du problème (description complète, codage inclus)
- une question n'admettant comme réponse que oui ou non.

Exemple :

- NOM** : **KNAPSACK** (sac à dos)
- DONNÉES** : un ensemble fini d'objets, E , avec deux fonctions entières, v et p , associant à chaque objet une valeur et un poids. Un poids total autorisé P et une valeur totale minimale V .
- QUESTION** : peut-on choisir des objets (à mettre dans le sac) de manière à ne pas dépasser le poids total autorisé, et que le total des valeurs soit supérieur ou égal à V ?

3.2 La réduction polynomiale

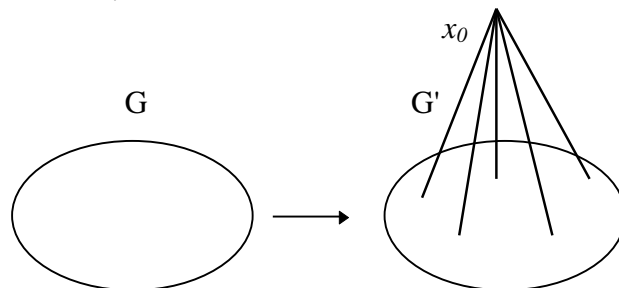
Soient P_1 et P_2 deux problèmes. On dira que le problème P_1 peut être *réduit* au problème P_2 s'il existe une transformation associant à chaque instance Π de P_1 une instance $f(\Pi)$ de P_2 , de manière que la réponse à Π est oui si et seulement si la réponse à $f(\Pi)$ est oui. On utilise le terme *réduction*, car en disposant d'une telle transformation, si on sait résoudre le problème P_2 , alors on sait aussi résoudre le problème P_1 (on le transforme, puis on répond à la question par oui ou non). Notons le paradoxe linguistique que P_1 se réduit en un problème plus dur P_2 . Si, de plus, la transformation est polynomiale (peut se faire en temps polynomial) alors la réduction est dite polynomiale. On notera $P_1 \propto P_2$.

Exemple : Donnons d'abord deux problèmes, puis la réduction polynomiale :

- NOM** : **CHAINEDHAM** (chaîne hamiltonienne)
DONNÉES : un graphe fini $G(V,E)$, représenté sous forme de listes d'adjacence.
QUESTION : est-ce que le graphe admet une chaîne hamiltonienne (une chaîne passant une fois et une seule par tous les sommets) ?
- NOM** : **CYCLEDHAM** (cycle hamiltonien)
DONNÉES : un graphe fini $G(V,E)$, représenté sous forme de listes d'adjacence.
QUESTION : est-ce que le graphe admet un cycle hamiltonien (un cycle passant une fois et une seule par tous les sommets) ?

Montrons que **CHAINEDHAM** \propto **CYCLEDHAM**.

Décrivons d'abord la transformation : le graphe $G'(V',E')$ pour le problème **CYCLEDHAM** est obtenu en rajoutant au graphe $G(V,E)$ donné pour le problème **CHAINEDHAM** un sommet relié à tous les autres sommets. Cette transformation peut se faire en temps polynomial.



Montrons que cette transformation polynomiale est une réduction. Il faut donc montrer que $G(V,E)$ admet une chaîne hamiltonienne si et seulement si $G'(V',E')$ admet un cycle hamiltonien.

Si : supposons que $G'(V',E')$ admet un cycle hamiltonien. Soient x_1, x_2, \dots, x_n les sommets de V et x_0 le sommet rajouté pour obtenir G' . Le cycle hamiltonien moins le sommet x_0 constitue une chaîne hamiltonienne dans G .

Seulement si : supposons que G admet une chaîne hamiltonienne. Cette chaîne, avec le sommet x_0 et les deux arêtes qui le relient aux deux extrémités de la chaîne constitue en G' un cycle hamiltonien. \square

On peut remarquer que la relation \propto est une relation d'ordre. On notera par \approx l'équivalence associée : $P_1 \approx P_2$ si et seulement si $P_1 \propto P_2$ et $P_2 \propto P_1$.

3.3 Quelques autres exemples de réduction polynomiale

Nous donnons ici quelques autres exemples de réduction polynomiale qui nous permettront d'établir que tous les problèmes classiques d'hamiltonisme sont polynomialement équivalents. Énonçons d'abord ces problèmes :

- NOM** : **CHEMINHAM** (chemin hamiltonien)
DONNÉES : un graphe orienté fini $G(V,E)$, représenté sous forme de listes de successeurs.
QUESTION : est-ce que le graphe admet un chemin hamiltonien (un chemin passant une fois et une seule par tous les sommets) ?
- NOM** : **CIRCUITHAM** (circuit hamiltonien)
DONNÉES : un graphe orienté fini $G(V,E)$, représenté sous forme de listes de successeurs.
QUESTION : est-ce que le graphe admet un circuit hamiltonien (un circuit passant une fois et une seule par tous les sommets) ?

Remarquons tout d'abord que la preuve utilisée pour la réduction de **CHAINEDHAM** vers **CYCLEHAM** peut être adaptée pour prouver la réduction de **CHEMINHAM** vers **CIRCUITHAM**. En effet, il suffit de remplacer les arêtes rajoutées entre les anciens sommets et le nouveau sommet par des arcs dans les deux sens. Cette même idée peut être réutilisée pour démontrer :

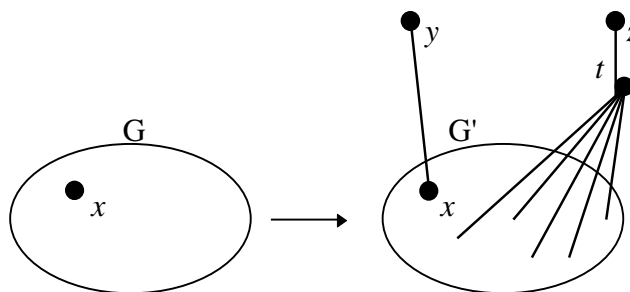
Proposition : i) **CYCLEHAM** \propto **CIRCUITHAM**.
 ii) **CHAINEDHAM** \propto **CHEMINHAM**.

Il est un peu plus difficile de prouver le résultat suivant :

Proposition : **CYCLEHAM** \propto **CHAINEDHAM**.

Preuve : En effet, si on connaît un cycle, la transformation est facile (il suffit de supprimer une arête du cycle), mais ce n'est pas vrai dans notre cas, le problème étant exactement l'existence du cycle.

Décrivons d'abord la transformation : le graphe $G'(V',E')$ pour le problème **CHAINEDHAM** est obtenu en rajoutant au graphe $G(V,E)$ donné pour le problème **CYCLEHAM** un sommet t relié à tous les voisins d'un sommet x choisi. Ensuite on rajoute un sommet z ayant comme seul voisin t et un sommet y ayant comme seul voisin x .



Cette transformation peut se faire en temps polynomial. Montrons qu'il s'agit d'une réduction. Il faut donc montrer que $G(V,E)$ admet un cycle hamiltonien si et seulement si $G'(V',E')$ admet une chaîne hamiltonienne.

Si : supposons que $G'(V',E')$ admet une chaîne hamiltonienne. Comme G' admet deux

sommets de degré un (y et z), ces sommets sont nécessairement les deux extrémités de la chaîne. De plus, le voisin de z dans la chaîne est nécessairement t . Ainsi, la partie de la chaîne obtenue par la suppression de y , z et t est une chaîne hamiltonienne dans le graphe original G , qui a comme extrémités deux sommets voisins. Ce qui implique l'existence d'un cycle hamiltonien dans G .

Seulement si : supposons que G admet un cycle hamiltonien. Si on supprime dans ce cycle une des deux arrêtes d'extrémité x , nous obtenons une chaîne hamiltonienne ayant comme extrémités x et un voisin de x , donc dans G' on peut le compléter en une chaîne hamiltonienne. \square

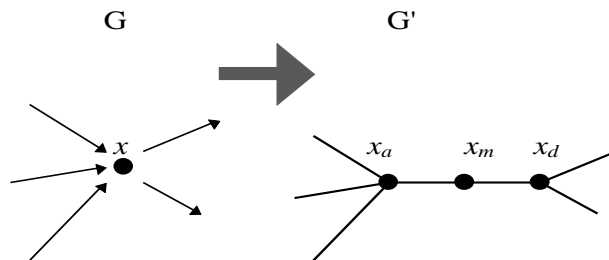
Presque de la même manière on peut prouver la réduction de **CIRCUITHAM** vers **CHEMINHAM**. En effet, en tenant compte de l'orientation, il suffit que le sommet t ajouté ait les mêmes successeurs que x . Ensuite y sera le prédécesseur de t et y le successeur de x .

Pour compléter ce tour d'horizon de problèmes d'hamiltonisme, il nous reste à prouver que le problème de **CIRCUITHAM** se réduit vers **CYCLEHAM** (ou bien **CHEMINHAM** vers **CHAINEHAM**).

Proposition : CIRCUITHAM \propto CYCLEHAM.

Preuve : La toute première idée, consistant en la suppression de l'orientation, n'est pas bonne. En effet, on peut ainsi obtenir un cycle, sans que le graphe de départ admette un circuit.

La transformation proposée est comme suit : chaque sommet x du graphe orienté est remplacé par trois sommets x_a (a comme arrivée), x_m (m comme milieu) et x_d (d comme départ). Un arc xy devient une arrête $x_d y_a$ dans G' . De plus, on rajoute les arrêtes reliant x_a à x_m et x_d à x_m , pour tout x .



Cette transformation peut se faire en temps polynomial. Montrons qu'il s'agit d'une réduction. Il faut donc montrer que $G(V,E)$ admet un circuit hamiltonien si et seulement si $G'(V',E')$ admet un cycle hamiltonien.

Si : supposons que $G'(V',E')$ admet un cycle hamiltonien. Comme les sommets x_m sont de degré deux dans G' , un cycle hamiltonien doit forcément parcourir les arrêtes dans l'ordre $x_a, x_m, x_d, y_a, y_m, y_d, \dots$ (ou dans l'ordre inverse), ce qui correspond à l'existence d'un circuit hamiltonien dans G .

Seulement si : supposons que G admet un circuit hamiltonien. Ce circuit donne lieu à un cycle hamiltonien dans G' . \square

Nous pouvons remarquer que cette transformation peut aussi être utilisée pour la réduction de **CHEMINHAM** vers **CHAINEHAM**. Ainsi, les quatre problèmes sont polynomialement équivalents.

3.4 P et NP

On distinguera deux classes de problèmes particuliers :

Définition : P est la classe des problèmes qu'on sait résoudre en temps polynomial sur une machine de Turing déterministe (vu les résultats du chapitre précédent, il n'est pas nécessaire de préciser le nombre de bandes).

Définition : NP est la classe des problèmes qu'on sait résoudre en temps polynomial sur une machine de Turing non-déterministe.

Deux remarques immédiates s'imposent :

Proposition : Tous les problèmes inclus dans **P** sont polynomialement équivalents.

Preuve : Soient P_1 et P_2 deux problèmes dans **P**.

Montrons que $P_1 \propto P_2$. On choisit deux instances de P_2 , Π_O pour laquelle la réponse est oui et Π_n pour laquelle la réponse est non.

La transformation se fait en deux étapes :

- i) On résout l'instance donnée de P_1 .
- ii) Selon la réponse le résultat de la transformation sera Π_O ou Π_n . □

Proposition : P est inclus dans **NP**.

Preuve : comme on a vu qu'une machine déterministe peut être considérée comme une machine non-déterministe avec un choix unique pour chaque transition, la conclusion est évidente. □

Cette dernière proposition nous pose un des problèmes ouverts les plus importants en informatique théorique :

Problème : est-ce que l'inclusion de **P** dans **NP** est stricte ?

Si on savait répondre à cette question, alors on pourrait définir les problèmes comme difficiles s'ils sont dans **NP** et non dans **P**. Malheureusement, on n'a pas de réponse à cette question et il paraît peu probable d'en avoir une sous peu¹.

Nous pouvons remarquer que la simulation qu'on a faite d'une machine non-déterministe par une machine déterministe ne répond pas à la question. En effet, la durée de la simulation était exponentielle.

¹ C'est dans ce contexte que l'Institut Clay propose un prix d'un million de dollars pour une réponse à cette question (voir <http://www.claymath.org/prizeproblems/pvsnp.htm>).

CHAPITRE 4

NP-complétude

4.1 La notion

Comme on ne sait pas si $P = NP$, il fallait trouver un autre moyen de définir les problèmes difficiles. Une réponse à cette question est fournie par le théorème de Cook. Avant de l'énoncer, essayons d'abord de présenter l'esprit de ce résultat exceptionnel.

Comme on a vu, les réductions polynomiales nous donnent un ordre partiel sur NP . Un ordre partiel assez particulier, car dans cet ordre les éléments de P sont tous équivalents. La communauté des informaticiens est divisée en ce qui concerne le problème de l'inclusion stricte. Certains croient que l'inclusion est stricte, d'autres que non. Si on admet que l'inclusion est stricte, c'est qu'il y a plusieurs classes d'équivalence. Dans ce cas on peut étudier l'ordre quotient. En particulier, existe-t-il un élément maximum (clairement un minimum existe, c'est la classe P). Si un maximum existe, alors au cas où la réponse au problème ouvert serait positive, les problèmes de cette classe seraient dans NP et non dans P . On appellera cette classe d'équivalence la classe des problèmes NP -complets.

Cette idée nous permet d'obtenir une appréciation de la difficulté des problèmes à étudier, sans avoir à résoudre le problème ouvert. Ainsi, pour montrer qu'un problème est NP -complet, on doit :

- montrer que le problème est dans NP .
- trouver un problème NP -complet qui se réduit polynomialement au problème étudié.

Cette preuve peut ainsi devenir relativement simple. Le problème majeur est par où doit-on commencer ? Est-ce qu'un tel élément maximal existe (a priori il n'y a aucune raison pour qu'il existe) ? La difficulté provient du fait qu'on doit prouver que tout problème de NP peut être réduit polynomialement à un tel élément maximal. La réponse est donnée par le théorème de Cook.

4.2 Le théorème de Cook

Pour pouvoir énoncer le théorème de Cook, faisons d'abord quelques rappels de logique. Une *variable* logique est une variable qui peut prendre une des deux valeurs vrai ou faux. Un *littéral* est une variable ou la négation d'une variable. Une *formule* logique est une expression contenant des variables, reliées par les opérations de négation, conjonction et disjonction. Une *clause* est la disjonction de littéraux. Le degré de la clause est le nombre de littéraux qu'elle contient. Une *formule en forme normale conjonctive* est une conjonction de clauses.

A toutes les notions ci-dessus on associe une évaluation, donnée par la valeur de l'expression, compte tenu de la valeur des variables. Une formule est dite *satisfiable*, s'il existe une affectation des variables pour laquelle la formule est vraie.

Il est bien connu que toute formule peut être mise sous forme normale conjonctive. C'est important, car une telle formule de longueur n , peut être évaluée en temps $O(n)$.

Soient v_1, v_2, \dots, v_m des variables logiques. Nous souhaitons utiliser la formule logique suivante : $exactement_un(v_1, v_2, \dots, v_m)$ qui est vraie si et seulement si une (et une seule) des variables est vraie.

Proposition : $exactement_un(v_1, v_2, \dots, v_m)$ peut s'écrire en forme normale conjonctive, avec une longueur $O(m^2)$.

Preuve : Tout d'abord notons que

$$exactement_un(v_1, v_2, \dots, v_m) = au_moins_un(v_1, v_2, \dots, v_m) \wedge au_plus_un(v_1, v_2, \dots, v_m).$$

En ce qui concerne la première formule, nous avons

$$au_moins_un(v_1, v_2, \dots, v_m) = \bigvee_{i=1}^m v_i.$$

Nous pouvons écrire la deuxième formule sous la forme

$$au_plus_un(v_1, v_2, \dots, v_m) = \bigwedge_{1 \leq i < j \leq m} (\neg v_i \vee \neg v_j).$$

Ainsi, la longueur totale de la formule obtenue (sous forme normale conjonctive) est

$$m + \frac{m(m-1)}{2} = \frac{m(m+1)}{2}.$$

□

Pour énoncer le théorème de Cook, formulons d'abord le problème de la satisfiabilité.

NOM : **SAT** (satisfiabilité)

DONNÉES : une formule sous forme normale conjonctive

QUESTION : est-ce que la formule est satisfiable ?

Théorème [Cook, 1971] : **SAT** est **NP**-complet.

Preuve : La preuve est composée de deux parties. Tout d'abord nous devons montrer que **SAT** \in **NP**. Par la suite nous montrerons que tout problème $\Pi \in$ **NP** peut être réduit à **SAT**.

i) **SAT** \in **NP**. En effet, en utilisant une machine non-déterministe à deux bandes, on peut d'abord associer une valeur de vérité aux variables de façon non-déterministe (en temps $O(n^2)$ - ce qui correspond à un passage pour chaque variable), et évaluer la formule ainsi obtenue en temps linéaire ensuite. Si la formule est vraie, on passe dans l'état final d'acceptation. Comme la complexité est définie par le temps minimum nécessaire, ce minimum est atteint lors du bon choix dès le début des valeurs des variables. Ainsi, le temps est polynomial, donc **SAT** \in **NP**.

ii) Soit $\Pi \in$ **NP**. La difficulté vient du fait qu'il faut prouver qu'un problème quelconque Π de **NP** peut être réduit polynomialement à **SAT**. La généralité de l'énoncé ne nous donne que peu d'indications en ce qui concerne Π . La seule information dont on dispose, est l'existence d'une machine de Turing non-déterministe, permettant d'accepter Π en temps polynomial. C'est donc cette information qui est codée en formule. Par la définition de **NP**, il existe une machine de Turing M , non-déterministe, qui accepte une donnée de longueur n en temps $p(n)$. Nous pouvons supposer qu'il s'agit d'une machine à une bande, car la simulation d'une machine à k bandes sur une machine à une bande se fait en temps polynomial. Soient q_0, q_1, \dots, q_s les états de la machine, avec q_0 l'état initial et les états q_r, q_{r+1}, \dots, q_s finaux (états d'acceptation). Pour simplifier notre travail, nous modifions la machine de sorte qu'une fois un état d'acceptation atteint, elle reste «stationnaire» dans cet état. Ainsi, il n'est plus utile de vérifier après chaque transition si la machine a terminé, il suffit de faire $p(n)$ étapes de calcul et vérifier ensuite. Soient l_0, l_1, \dots, l_u les lettres de l'alphabet de M , avec l_0 le blanc. La table de transition de la machine nous donne une valeur m des nombres de lignes de la table. Soit x_1, x_2, \dots, x_n une instance de longueur n du problème Π . Nous allons construire une formule

logique $F(x_1, x_2, \dots, x_n)$, de longueur polynomiale en n , qui est satisfiable si et seulement si la machine M accepte l'instance x_1, x_2, \dots, x_n . Nous disposerons des variables logiques suivantes :

- $s_{i,j,t}$ avec $-p(n) \leq i \leq p(n)$, $0 \leq j \leq u$ et $0 \leq t \leq p(n)$. Si $s_{i,j,t}$ est vraie, c'est que la case i de la bande contient la lettre l_j au temps t .
- $z_{i,t}$ avec $0 \leq i \leq s$ et $0 \leq t \leq p(n)$. La vérité de $z_{i,t}$ signifie qu'au temps t la machine se trouve dans l'état q_i .
- $h_{i,t}$ avec $-p(n) \leq i \leq p(n)$ et $0 \leq t \leq p(n)$. Si $h_{i,t}$ est vraie, c'est que la tête de lecture/écriture se trouve sur la case i de la bande au temps t .
- $b_{i,t}$ avec $1 \leq i \leq m$ et $1 \leq t \leq p(n)$. La vérité de $b_{i,t}$ signifie que pour passer du temps $t-1$ au temps t nous avons choisi la $i^{\text{ième}}$ ligne de la table de transition. Une transition représente donc une ligne de la table des transitions, de la forme :

ligne	lecture	état	écriture	déplacement	état
i	$lect_i$	$état_i$	$écrit_i$	$dépl_i$	$nétat_i$

avec $1 \leq i \leq m$, $lect_i$ et $écrit_i$ dans l'intervalle $[0...u]$, $état_i$ et $nétat_i$ dans l'intervalle $[0...s]$ et $dépl_i$ dans $\{-1, 0, 1\}$.

Ces variables choisies, nous devons tout d'abord nous assurer de leur consistance. Ainsi, nous aurons les formules suivantes :

- chaque case contient une seule lettre à la fois :

$$Cons(bande) = \bigwedge_{-p(n) \leq i \leq p(n)} \left(\bigwedge_{0 \leq t \leq p(n)} \text{exactement_un}(s_{i,0,t}, s_{i,1,t}, \dots, s_{i,u,t}) \right)$$

ce qui donne une formule de longueur $(2p(n) + 1)(p(n) + 1) \frac{u(u+1)}{2}$, c.a.d. en $O(p^2(n))$.

- on est dans un seul état à la fois :

$$Cons(état) = \bigwedge_{0 \leq t \leq p(n)} \text{exactement_un}(z_{0,t}, z_{1,t}, \dots, z_{s,t})$$

ce qui donne une formule de longueur $(p(n) + 1) \frac{s(s+1)}{2}$, c.a.d. en $O(p(n))$.

- la tête se trouve en un seul endroit à la fois :

$$Cons(tête) = \bigwedge_{0 \leq t \leq p(n)} \text{exactement_un}(h_{-p(n),t}, h_{-p(n)+1,t}, \dots, h_{p(n),t})$$

ce qui donne une formule de longueur $(p(n) + 1)(2p(n) + 1)(p(n) + 1)$, c.a.d. en $O(p^3(n))$.

- un seul choix de transition à la fois :

$$Cons(trans) = \bigwedge_{0 \leq t \leq p(n)} \text{exactement_un}(b_{1,t}, b_{2,t}, \dots, b_{m,t})$$

ce qui donne une formule de longueur $(p(n) + 1) \frac{m(m+1)}{2}$, c.a.d. en $O(p(n))$.

La formule de consistance

$$Cons(x) = Cons(bande) \wedge Cons(état) \wedge Cons(tête) \wedge Cons(trans)$$

est de coût total en $O(p^3(n))$.

Il faut s'assurer que l'initialisation est correcte. Ainsi il faut que les n cases de la bande contenant les données soit correctes (c.a.d. la case i contient x_i qui est la lettre l_i , que les autres cases contiennent des blancs, que la tête se trouve sur la première case et que l'état initial soit q_0 :

$$Init(x) = \left(\bigwedge_{-p(n) \leq i \leq 0} s_{i,0,0} \right) \wedge \left(\bigwedge_{1 \leq i \leq n} s_{i,l_i,0} \right) \wedge \left(\bigwedge_{n+1 \leq i \leq p(n)} s_{i,0,0} \right) \wedge z_{0,0} \wedge h_{0,0}$$

Tout ceci rajoute des formules de longueur $O(p(n))$.

Un autre problème dont il faut se soucier concerne la terminaison. En effet, pour

l'acceptation il faut que la machine se trouve en temps $p(n)$ dans un des états finaux q_r, q_{r+1}, \dots, q_s . Ainsi

$$Termine(x) = z_{r,p(n)} \vee z_{r+1,p(n)} \vee \dots \vee z_{s,p(n)}$$

ce qui n'est qu'une formule supplémentaire de longueur constante.

Il nous reste la partie la plus difficile de notre travail. En effet, il faut assurer que les transitions sont correctes. Il faut donc vérifier, que les «transitions se passent bien». Pour cela, on définit

$$Transition(x) = \bigwedge_{1 \leq t \leq p(n)} Trans(t).$$

Pour chaque unité de temps, il faut que la bande ne change pas, sauf à l'endroit, où se trouve la tête, que le changement d'état et que le déplacement soient conformes à la transition choisie.

$$Trans(t) = \bigwedge_{-p(n) \leq i \leq p(n)} \left\{ \bigwedge_{1 \leq j \leq u} (h_{i,t-1} \vee \neg s_{i,j,t-1} \vee s_{i,j,t}) \wedge BTrans(i) \right\}$$

avec $BTrans$ qui exprime que la transition s'est bien déroulée, en ce qui concerne le changement d'état et le déplacement, c.a.d.

$$BTrans(i) = \bigwedge_{1 \leq l \leq m} \left\{ (h_{i,t-1} \wedge b_{l,t}) \Rightarrow (z_{t-1,état_l} \wedge z_{t,nétat_l} \wedge s_{i,lect_l,t-1} \wedge s_{i,écrit_l,t} \wedge h_{i+dépl_l,t}) \right\}.$$

Ceci peut être traduit en

$$BTrans(i) = \bigwedge_{1 \leq l \leq m} \left\{ \neg h_{i,t-1} \vee \neg b_{l,t} \vee (z_{t-1,état_l} \wedge z_{t,nétat_l} \wedge s_{i,lect_l,t-1} \wedge s_{i,écrit_l,t} \wedge h_{i+dépl_l,t}) \right\}.$$

Comme cette formule n'est pas sous forme normale conjonctive, on est obligé de la transformer en transformant $\alpha \vee (\beta \wedge \lambda)$ en $(\alpha \vee \beta) \wedge (\alpha \wedge \lambda)$ et on obtient

$$BTrans(i) = \bigwedge_{1 \leq l \leq m} \left\{ \begin{array}{l} \left(\neg h_{i,t-1} \vee \neg b_{l,t} \vee z_{t-1,état_l} \right) \wedge \\ \left(\neg h_{i,t-1} \vee \neg b_{l,t} \vee z_{t,nétat_l} \right) \wedge \\ \left(\neg h_{i,t-1} \vee \neg b_{l,t} \vee s_{i,lect_l,t-1} \right) \wedge \\ \left(\neg h_{i,t-1} \vee \neg b_{l,t} \vee s_{i,écrit_l,t} \right) \wedge \\ \left(\neg h_{i,t-1} \vee \neg b_{l,t} \vee h_{i+dépl_l,t} \right) \end{array} \right\}$$

La longueur totale des formules de transition est en $O(p^2(n))$, donc la longueur totale de la formule construite est en $O(p^3(n))$.

Pour terminer la preuve, il faut encore prouver que la formule ainsi obtenue est satisfiable si et seulement si la donnée du problème Π donne lieu à une réponse favorable (acceptation) par la machine de Turing M . Pour cela il suffit de vérifier les différentes étapes de la construction de la formule. \square

Une fois le théorème de Cook prouvé (1970), la théorie s'est très rapidement développée. Au début il y avait 6 autres problèmes prouvés comme étant **NP-complets**, leur liste détaillée est donnée dans la section suivante. Depuis la liste s'est allongée, et aujourd'hui on connaît des milliers de problèmes **NP-complets**.

4.3 Variantes de SAT

Plusieurs variantes du problème **SAT** sont connues. Nous évoquerons ici deux familles de versions:

- NOM** : **k-SAT** (k - Satisfiabilité)
DONNÉES : une formule logique sous forme normale conjonctive, composée de clauses de degré au plus k.
QUESTION : est-ce que la formule est satisfiable ?
- NOM** : **Xk-SAT** (k - Satisfiabilité exacte)
DONNÉES : une formule logique sous forme normale conjonctive, composée de clauses de degré exactement k.
QUESTION : est-ce que la formule est satisfiable ?

Nous montrons dans ce qui suit que ces problèmes sont **NP-complets** pour $k \geq 3$, alors que le cas $k=2$ est polynomial. Cette démarche est donnée ici à titre d'exemple et parce que les problèmes **3-SAT** et **X3-SAT** sont largement utilisés dans la littérature. En général, la recherche ne se contente pas de constater qu'un problème donné soit **NP-complet**. Ainsi, les travaux continuent pour essayer de répondre à des questions plus précises, est-ce que le problème reste **NP-complet**, même si les données disposent de propriétés supplémentaires. Le travail est considéré réellement fini lorsque la frontière entre **NP-complet** et polynomial est trouvée pour le problème étudié. C'est dans ce cadre qu'on peut inscrire l'étude de **3-SAT** (encore **NP-complet**) et de **2-SAT** (polynomial).

Théorème [Cook, 1971] : 3-SAT est NP-complet.

Preuve : Tout d'abord **3-SAT** \in **NP**, car une instance de **3-SAT** est une instance de **SAT**. Pour prouver la **NP-complétude** il suffit de montrer que **SAT** \propto **3-SAT**, car par transitivité on pourra déduire que tout problème de **NP** admet une réduction polynomiale vers **3-SAT**. La transformation : pour transformer une instance de **SAT** en instance de **3-SAT**, il faut limiter le degré des clauses.

$ C $	C	$f(C)$
1	l_1	l_1
2	$(l_1 \vee l_2)$	$(l_1 \vee l_2)$
3	$(l_1 \vee l_2 \vee l_3)$	$(l_1 \vee l_2 \vee l_3)$
4	$(l_1 \vee l_2 \vee l_3 \vee l_4)$	$(l_1 \vee l_2 \vee l_3) \wedge (\neg l_3 \vee l_4)$
5	$(l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5)$	$(l_1 \vee l_2 \vee l_3) \wedge (\neg l_3 \vee l_4 \vee l_5) \wedge (\neg l_4 \vee l_5)$
≥ 6	$(l_1 \vee l_2 \vee \dots \vee l_n)$	$(l_1 \vee l_2 \vee l_{n+1}) \wedge (\neg l_{n+1} \vee l_3 \vee l_{n+2}) \wedge \dots \wedge (\neg l_{2n-4} \vee l_{n-2} \vee l_{2n-3}) \wedge (\neg l_{2n-3} \vee l_{n-1} \vee l_n)$

Evidemment les nouvelles variables introduites ne sont utilisées que dans une formule à la fois. On peut facilement constater que la transformation s'effectue en temps polynomial. Il reste à prouver que la formule obtenue (instance de **3-SAT**) est satisfiable si et seulement si la formule originale (instance de **SAT**) est satisfiable. En effet il est facile de vérifier, que les nouvelles variables introduites jouent un rôle de «colle» entre les clauses de degré 3. Une fois, qu'on a la clause de degré trois vraie, les autres clauses peuvent être satisfaites par les valeurs des nouvelles variables. \square

Théorème : X3-SAT est NP-complet.

Preuve : Comme dans le cas de **3-SAT**, il suffit de rappeler qu'une instance de **X3-SAT** est une instance de **SAT**, pour conclure que **X3-SAT** \in **NP**. Pour prouver la **NP-complétude** il

suffit de réduire **3-SAT** à **X3-SAT**, et par transitivité déduire que tout problème de **NP** admet une réduction polynomiale vers **X3-SAT**.

La transformation : pour transformer une instance de **3-SAT** en instance de **X3-SAT**, il faut augmenter le degré des clauses.

$ C $	C	$f(C)$
1	l_1	$(l_1 \vee l_2 \vee l_3) \wedge (l_1 \vee l_2 \vee \neg l_3) \wedge (l_1 \vee \neg l_2 \vee l_3) \wedge (l_1 \vee \neg l_2 \vee \neg l_3)$
2	$(l_1 \vee l_2)$	$(l_1 \vee l_2 \vee l_3) \wedge (l_1 \vee l_2 \vee \neg l_3)$
3	$(l_1 \vee l_2 \vee l_3)$	$(l_1 \vee l_2 \vee l_3)$

Dans ce cas aussi, les nouvelles variables introduites ne sont utilisées que dans une formule à la fois. On peut facilement constater que la transformation s'effectue en temps polynomial. Il reste à prouver que la formule obtenue (instance de **3-SAT**) est satisfiable si et seulement si la formule originale (instance de **SAT**) est satisfiable, mais c'est trivialement le cas. \square

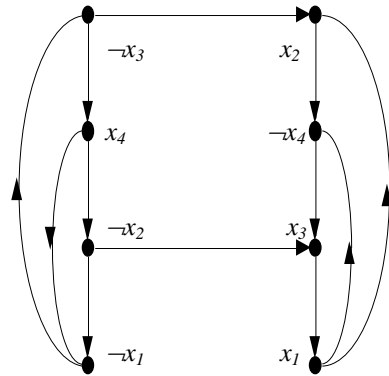
Théorème : 2-SAT \in P.

Preuve : L'idée est la suivante : on associe un graphe orienté à la formule, ayant comme sommets les variables utilisés et leurs négations. Une clause de la forme $(l_1 \vee l_2)$ donne lieu à deux arcs : $(\neg l_1 \rightarrow l_2)$ et $(\neg l_2 \rightarrow l_1)$. Ces arcs correspondent aux implications dues à la satisfiabilité, c.a.d. que si l_1 est faux alors l_2 doit être vraie (et l'inverse). On peut calculer la fermeture transitive de ce graphe. On montrera que la formule est satisfiable si et seulement si cette fermeture ne contient pas de double implication entre une variable et sa négation. Ce qui suggère le test sur le graphe original, consistant en la recherche des composantes fortement connexes et la vérification qu'une variable et sa négation se trouvent dans des composantes connexes différentes. En effet, s'il existe un chemin d'un littéral l_i vers un littéral l_t passant par les sommets l_2, l_3, \dots, l_{t-1} , alors la formule contient les clauses $(\neg l_1 \vee l_2)$, $(\neg l_2 \vee l_3)$, $(\neg l_3 \vee l_4)$, \dots , $(\neg l_{t-1} \vee l_t)$. Ainsi, si $\neg l_1$ est faux, alors l_2 est vrai, ce qui implique à son tour la vérité de $l_3, l_4 \dots l_t$. Supposons maintenant qu'il existe un chemin de l_i vers $\neg l_i$ et aussi un chemin de $\neg l_i$ vers l_i . Comme l'un des deux est vrai, l'autre aussi doit être vrai, ce qui est impossible. Donc dans ce cas la formule n'est pas satisfiable. Supposons maintenant que, quel que soit le littéral l_i choisi, l_i et $\neg l_i$ ne sont pas dans la même composante fortement connexe. Montrons qu'on peut attribuer des valeurs de vérité qui satisfont la formule. On peut remarquer qu'il suffit de trouver une attribution de valeurs de vérité aux littéraux, pour laquelle aucun arc dans le graphe n'a pas comme extrémité initiale vrai et comme extrémité terminale faux. Pour affecter les valeurs de vérité, on choisit un littéral l_i , tel qu'il n'y a pas de chemin de l_i vers $\neg l_i$. On affecte à l_i la valeur vraie ainsi qu'à tous les sommets qu'on peut atteindre à partir de l_i . De même on attribue la valeur fausse à $\neg l_i$ ainsi qu'aux sommets à partir desquels on peut atteindre $\neg l_i$. S'il reste des littéraux auxquels nous n'avons pas encore attribué de valeur, on recommence. Il reste à montrer que ce procédé est correct, c.a.d. ne mène pas à contradiction. Tout d'abord, nous remarquons que si on attribue une valeur de vérité à l_j , alors on attribue aussi une valeur (le contraire) à $\neg l_j$. En effet, il est facile de remarquer que, s'il existe un chemin de l_i vers l_j , alors il existe aussi un chemin de $\neg l_i$ vers $\neg l_j$. Ainsi les attributions de valeurs se font en couples. Est-il possible qu'on souhaite lors d'une phase attribuer à la fois la valeur vrai et la valeur fausse à un littéral l_j ? Dans ce cas, on devait avoir un chemin de l_i vers l_j et un chemin de l_j vers $\neg l_i$, c.a.d. un chemin de l_i vers $\neg l_i$, ce qui est impossible. L'autre situation qu'on doit évoquer est celle où on souhaite attribuer une valeur contraire à l'existante à un littéral ayant déjà une valeur. Heureusement, ce cas de figure ne se présente pas. En effet, supposons par exemple qu'on souhaite attribuer la valeur vrai à un sommet l_j ayant déjà la valeur faux. Comme on souhaite attribuer la valeur vrai, c'est qu'il existe un chemin de l_i vers l_j . Mais dans ce cas, lors d'une phase précédente, lors

de l'affectation de la valeur faux à l_j on aurait du aussi attribuer la valeur faux à l_i , car il existe un chemin de l_i vers l_j . Ce qui termine la preuve. \square

Voici un exemple de notre démarche sur la formule suivante :

$$F = (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_4)$$



On détecte facilement les deux composantes fortement connexes qui correspondent aux deux cotés (gauche et droite) sur la figure. Ainsi, il est facile de déduire de ce graphe, que pour satisfaire la formule, il faut choisir la valeur faux pour x_4 et la valeur vraie pour x_1 , x_2 et x_3 . \square

CHAPITRE 5

Problèmes NP-complets connus

5.1 Quelques problèmes connus

Nous donnons ici une liste de problèmes **NP**-complets, parmi les plus connus. Le cadre réduit ne permet pas de faire toutes les preuves, mais il nous paraît important d'au moins donner les énoncés. Ces problèmes s'ajoutent à ceux dont les énoncés se trouvent dans les chapitres précédents ou vus en TD (**SAT**, **3-SAT**, **X3-SAT**, **KNAPSACK**, **SSP**, **CYCLEHAM**, **CHAINEDHAM**, ...).

NOM : **3DM** (Couplage en 3 dimensions)
DONNÉES : un ensemble M de triplets (w,x,y) , avec w, x et y des éléments de trois ensembles W, X, Y de même cardinalité q .
QUESTION : M contient-il un couplage (un sous-ensemble de triplets contenant tous les éléments une fois et une seule) ?

NOM : **VC** (transversal)
DONNÉES : un graphe fini $G(V,E)$, et un entier positif $K \leq |V|$.
QUESTION : le graphe admet-il un transversal (un ensemble de sommets contenant au moins une extrémité de toute arête) de cardinalité au plus K ?

NOM : **CLIQUE**
DONNÉES : un graphe fini $G(V,E)$, et un entier positif $C \leq |V|$.
QUESTION : le graphe admet-il une clique (sous-graphe complet) de cardinalité au moins C ??

NOM : **STABLE**
DONNÉES : un graphe fini $G(V,E)$, et un entier positif $J \leq |V|$.
QUESTION : le graphe admet-il un stable (sous-graphe vide) de cardinalité au moins J ?

NOM : **PARTITION**
DONNÉES : un ensemble fini d'entiers non-négatifs A .
QUESTION : existe-t-il une partition de A en deux ensembles A' et A'' , telle que la somme des éléments de A' soit égale à la somme des éléments de A'' ?

Pour une liste plus complète, nous proposons de consulter le livre de Garey et Johnson, et les articles de mise à jour qui paraissent régulièrement dans « Journal of Algorithms ».

5.2 Les problèmes transversal, clique et stable

Nous donnons ici quelques preuves classiques. En particulier, il s'agit en particulier de certaines parmi les toutes premières preuves par ordre chronologique.

Théorème [Karp, 1972] : VC est NP-complet.

Preuve : Il est facile de montrer que $VC \in NP$. En effet, si on dispose d'une prétendue solution, nous pouvons facilement la vérifier. Pour montrer que ce problème est NP-complet, nous allons montrer que $X3-SAT \propto VC$.

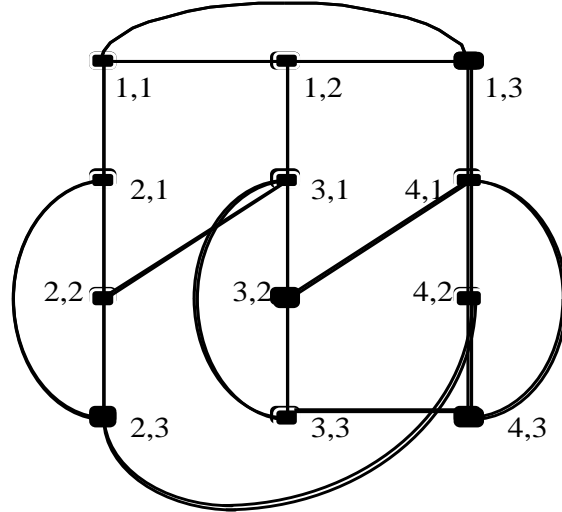
La transformation : Soit $F = C_1 \wedge C_2 \wedge \dots \wedge C_q$ une instance de **X3-SAT**, avec $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, les $l_{i,j}$ étant des littéraux. Nous construisons un graphe ayant $3q$ sommets qui correspondent aux $3q$ littéraux de la formule. Le sommet (i,j) , $1 \leq i \leq q$, $1 \leq j \leq 3$, qui correspond au littéral $l_{i,j}$ sera relié au sommet (m,n) si $i=m$ et $j \neq n$ ou si $i \neq m$ et $l_{i,j} = \neg l_{m,n}$. De plus la valeur attribuée à K sera $2q$. La construction se fait en temps polynomial.

Exemple de transformation : Considérons la formule suivante :

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_5) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_5).$$

Le graphe obtenu est le suivant :

Les « petits sommets » représentent les sommets du transversal, alors que les « grands sommets » correspondent aux littéraux satisfaits dans les clauses. Ainsi, dans notre exemple les littéraux qui ne sont pas dans le transversal sont : $x_3, x_4, x_3, \neg x_5$.



Nous pouvons remarquer que le graphe obtenu dispose de q triangles deux à deux disjoints, ce qui nous permet de conclure qu'il ne peut pas exister de transversal de cardinalité inférieure à $2q$. D'autre part, à cause des arêtes *entre les triangles*, il n'est pas évident qu'un tel transversal existe. Supposons que la formule F est satisfiable, et choisissons un littéral vrai dans chaque clause. Alors l'ensemble des $2q$ sommets associés aux littéraux non choisis est un transversal. Cet ensemble comprend deux sommets par triangle, donc les arêtes des triangles sont bien « couvertes ». Les arêtes qui ne sont pas dans les triangles relient toujours un littéral à sa négation, d'où nécessairement qu'une des deux extrémités est associé à un littéral faux, donc non sélectionné. Ainsi l'ensemble est un transversal. Le même raisonnement fonctionne aussi dans l'autre sens. En effet, si on dispose d'un transversal de cardinalité $2q$, il comprend nécessairement deux sommets par triangle. De plus, pour chaque littéral dont la négation figure aussi dans la formule, le littéral ou sa négation figurent dans le transversal, à cause de l'arête qui les relie. Les q littéraux non choisis peuvent être rendus vrais sans contradiction, ce qui entraîne la satisfaction de la formule. \square

Théorème [Karp, 1972] : STABLE est NP-complet.

Preuve : Il est facile de montrer que **STABLE** \in **NP**. En effet, si on dispose d'une prétendue solution, c.à.d. d'un ensemble de sommets candidats à être un stable de cardinalité au moins J , il suffit de vérifier qu'il s'agit en effet d'un stable et que l'ensemble est de bonne cardinalité. Pour terminer la preuve, il suffit de réduire un problème connu comme **NP**-complet au problème **STABLE**. Nous allons utiliser le problème du transversal, et nous montrons **VC** \propto **STABLE**. La transformation n'est rien d'autre que la transformation identité pour le graphe et $J \leftarrow |V| - K$. Cette transformation se fait de manière évidente en temps polynomial. Le « oui » si et seulement si « oui » se déduit de la propriété des graphes selon laquelle le complémentaire d'un ensemble stable est un transversal, et vice versa. En effet, si le complémentaire d'un transversal contenait une arête, cela contredirait le fait que l'ensemble était un stable. De la même façon, toute arête doit avoir au moins une extrémité dans le complémentaire d'un stable, ce qui prouve qu'il s'agit d'un transversal. \square

Remarque : c'est une bonne occasion de donner un exemple d'utilisation de la transitivité dans les preuves de **NP**-complétude. En effet, la composition des deux transformations données dans les réductions **X3-SAT** \propto **VC** et **VC** \propto **STABLE** nous donnent la transformation qui correspond à la réduction **X3-SAT** \propto **STABLE**. La transformation que nous obtenons par la composition est comme suit :

Soit $F = C_1 \wedge C_2 \wedge \dots \wedge C_q$ une instance de **X3-SAT**, avec $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, les $l_{i,j}$ étant des littéraux. Nous construisons un graphe ayant $3q$ sommets qui correspondent aux $3q$ littéraux de la formule. Le sommet (i,j) qui correspond au littéral $l_{i,j}$ sera relié au sommet (m,n) si $i=m$ et $j \neq n$ ou si $i \neq m$ et $l_{i,j} = \neg l_{m,n}$. De plus la valeur attribuée à J sera q .

La transformation est bien évidemment polynomiale et la preuve se déduit des deux preuves.

Théorème [Karp, 1972] : **CLIQUE** est **NP**-complet.

Preuve : Il est facile de montrer que **CLIQUE** \in **NP**. En effet, si on dispose d'une prétendue solution, c.à.d. d'un ensemble de sommets candidats à être une clique de cardinalité au moins C , il suffit de vérifier qu'il s'agit en effet d'une clique et que l'ensemble est de bonne cardinalité. Pour terminer la preuve, il suffit de réduire un problème connu comme **NP**-complet au problème **CLIQUE**. Nous allons utiliser le problème du stable, et nous montrons **STABLE** \propto **CLIQUE**. La transformation consiste en la complémentarité du graphe, c.à.d. on supprime les arêtes existantes et on ajoute celles qui n'existaient pas ; en ce qui concerne C , $C \leftarrow J$. En effet, les mêmes sommets qui formaient un stable qui forment la clique. Cette transformation se fait de manière évidente en temps polynomial. Le « oui » si et seulement si « oui » se déduit de la propriété des graphes selon laquelle un ensemble stable est une clique dans le graphe complémentaire, et vice versa. \square

5.3 Les problèmes de hamiltonisme

Nous avons vu lors dans les chapitres précédents que les différents problèmes de hamiltonisme sont polynomialement équivalents. Nous pourrions montrer maintenant que tous ces problèmes sont **NP**-complets. Pour ce faire, en tenant compte du fait que tous ces problèmes sont dans **NP**, il suffit de montrer qu'un de ces problèmes est **NP**-complet.

Nous donnons ici deux preuves différentes pour la **NP**-complétude du hamiltonisme. Tout d'abord nous donnons la preuve « classique », puis une preuve plus récente et plus simple.

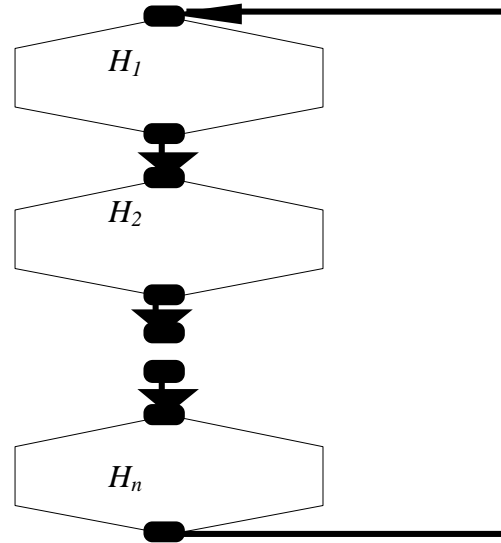
Théorème [Karp, 1972] : **CIRCUIT HAMILTONIEN** est **NP**-complet.

Preuve : Comme nous venons de le remarquer, il est facile de montrer que

CIRCUIT HAMILTONIEN \in **NP**. En effet, si on dispose d'une prétendue solution, nous pouvons facilement la vérifier. Pour montrer que ce problème est **NP-complet**, nous allons montrer que **X3-SAT** \propto **CIRCUIT HAMILTONIEN**.

La transformation : Soit $F = C_1 \wedge C_2 \wedge \dots \wedge C_q$ une instance de **X3-SAT**, avec $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, les $l_{i,j}$ étant des littéraux. Soient x_1, x_2, \dots, x_n les variables utilisées dans la formule. Nous construisons un graphe orienté G composé de deux types de sous-graphes. A chaque variable x_i on associe un sous-graphe H_i comme sur la figure ci-contre :

Les sous-graphes H_i disposent chacun d'un point d'entrée unique a_i et un point de sortie unique d_i et sont connectés en circuit, c.a.d. on a un arc du sommet d_i vers le sommet $a_{(i+1) \bmod n}$



Le nombre de clauses dans la formule est q . Chaque sous-graphe H_i est composé de sommets $a_i, b_{i,j}, c_{i,j}$ et d_i (avec $0 \leq j \leq q$). Ces sous-graphes ont une structure assez particulière. En effet, comme la seule «entrée» de H_i est a_i , le parcours commence par ce sommet.

Un chemin hamiltonien de H_i doit donc commencer en a_i et se terminer en d_i .

De plus, si un sommet $b_{i,j}$ est suivi par le sommet $c_{i,j+1}$, alors la seule manière d'assurer que le sommet $c_{i,j}$ soit sur le chemin, est qu'il soit le prédécesseur du sommet $b_{i,j}$.

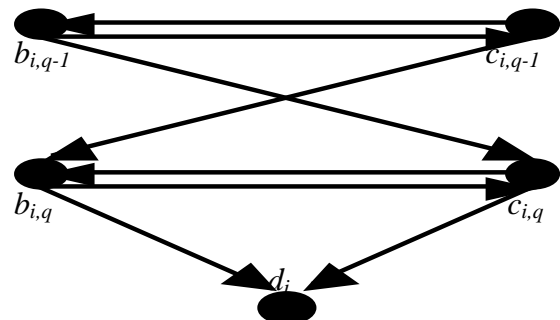
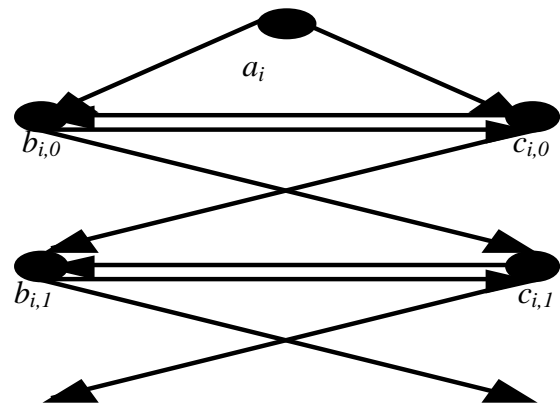
Ainsi, on ne peut parcourir les autres sommets que selon l'un des deux chemins hamiltoniens possibles :

$$a_i, b_{i,0}, c_{i,0}, b_{i,1}, c_{i,1}, \dots, b_{i,q}, c_{i,q}, d_i$$

ou

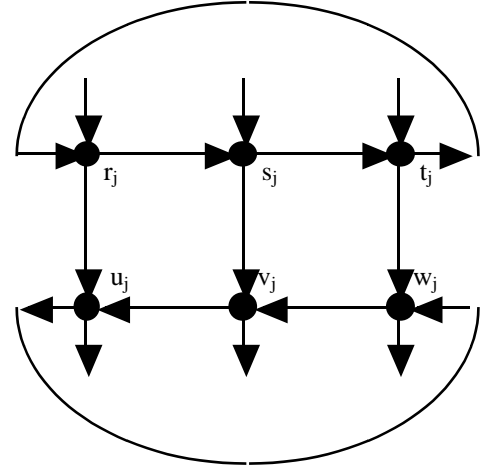
$$a_i, c_{i,0}, b_{i,0}, c_{i,1}, b_{i,1}, \dots, c_{i,q}, b_{i,q}, d_i.$$

Nous préconisons le choix du premier parcours si la variable x_i représentée par le sous-graphe H_i est vraie et le deuxième parcours dans le cas contraire.



Pour finir la transformation nous avons besoin d'une information supplémentaire, qui correspondra aux clauses de la formule. A chaque clause $C_j = l_{j,1} \vee l_{j,2} \vee l_{j,3}$, nous associons une copie du graphe G_j suivant :

Dans G_j , chacun des sommets r_j , s_j et t_j a un prédécesseur dans le graphe (hors G_j) et chacun des sommets u_j , v_j et w_j a un successeur dans le graphe (hors G_j). Ce sous-graphe a la particularité suivante : s'il existe un chemin hamiltonien du graphe, et si ce chemin arrive dans G_j par le sommet r_j (respectivement s_j ou t_j), alors ce chemin doit quitter G_j par le sommet qui se trouve en dessous le sommet r_j , le sommet u_j (respectivement v_j ou w_j).



En effet si le chemin arrive par le sommet r_i , on peut avoir les chemins suivants de l'entrée du sous-graphe jusqu'à une sortie (dans les lignes suivantes nous ne précisons pas les indices, quand cela n'est pas strictement nécessaire) :

ru , ruw , $ruwv$, rsv , $rsvu$, $rsvuw$, $rstw$, $rstwv$ et $rstvwu$. Il suffit de montrer que les chemins ruw , $ruwv$, rsv , $rsvuw$, $rstw$ et $rstwv$ ne peuvent pas faire partie d'un circuit hamiltonien du graphe :

- Si le chemin est ruw , $ruwv$ ou $rsvuw$ alors le sommet t ne pourra plus être visité, faute de successeur non encore visité.
- Si le chemin est rsv ou $rstwv$, alors le sommet u ne pourra plus être visité, faute de prédécesseur non encore visité.
- Si le chemin est $rstw$, alors le sommet v ne pourra plus être visité, faute de prédécesseur non encore visité.

Maintenant, nous pouvons donner la manière utilisée pour connecter les sous-graphes G_j au reste du graphe construit. Soit x_i le premier littéral de C_j . Dans ce cas nous rajoutons les arcs de $c_{i,j-1}$ vers r_j et de u_j vers $b_{i,j}$. Au cas où le premier littéral de C_j est $-x_i$ nous rajoutons les arcs de $b_{i,j-1}$ vers r_j et de u_j vers $c_{i,j}$. Ceci termine la construction de notre graphe. Ce graphe

a $2n(q+2)+6q$ sommets et $n(4q+7)+12q$ arcs (n est le nombre de variables de la formule transformée et q le nombre de clauses). La construction se fait en temps polynomial.

Pour comprendre la transformation, considérons pour l'instant le graphe sans les sous-graphes G_i . Supposons que le graphe admet un circuit hamiltonien. Nous pouvons supposer que ce circuit «commence» au sommet a_1 . Par la suite nous devons retrouver les deux sommets $b_{1,0}$ et $c_{1,0}$ dans un ordre quelconque, puis les sommets $b_{1,1}$ et $c_{1,1}$ toujours dans un ordre quelconque et ainsi de suite. En effet, a_1 ne dispose que de deux successeurs, $b_{1,0}$ et $c_{1,0}$, donc l'un des deux doit succéder à a_1 . Par ailleurs, comme chacun des deux sommets n'a d'autres prédécesseurs que a_1 et l'autre sommet, ils doivent nécessairement se succéder l'un à l'autre. Par la suite, comme $b_{i,1}$ n'a comme prédécesseurs que $c_{i,0}$ et $c_{i,1}$ et de même $c_{i,1}$ n'a comme prédécesseurs que $b_{i,0}$ et $b_{i,1}$, l'ordre entre $b_{i,0}$ et $c_{i,0}$ impose tout le parcours de H_i , qui ne peut être qu'un des deux suivants : $a_1, b_{i,0}, c_{i,0}, b_{i,1}, c_{i,1}, \dots, b_{i,q}, c_{i,q}, d_i$ ou bien $a_1, c_{i,0}, b_{i,0}, c_{i,1}, b_{i,1}, \dots, c_{i,q}, b_{i,q}, d_i$. Faisons correspondre la première version à un x_i vrai et la deuxième à un x_i faux. Ainsi les 2^n différents circuits hamiltoniens correspondent aux 2^n différentes valuations de notre formule de départ. Lors d'un parcours de H_i , on peut aussi parcourir (en entier ou partiellement) les sommets d'un G_j , si on a choisi le parcours correspondant de H_i . Ainsi si on a une affectation de valeurs de vérité qui satisfait la formule, on peut en déduire un circuit hamiltonien du graphe construit. De la même manière, si on dis-

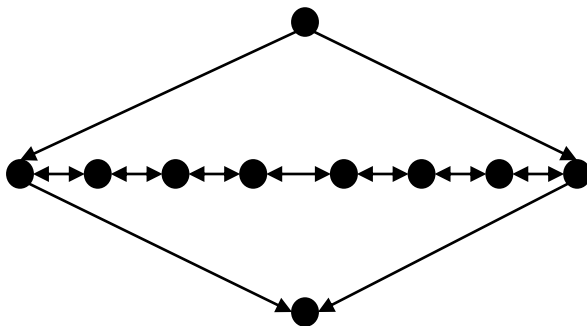
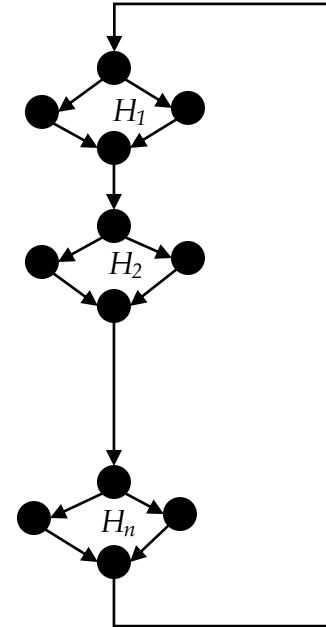
pose d'un circuit hamiltonien du graphe, c'est qu'on a visité chacun des sous-graphes G_j , ce qui implique que dans le sous-graphe H associé à un des variables de la clause, nous avons choisi une valeur de vérité qui satisfait la clause. \square

Seconde preuve (de la NP-difficulté uniquement) : Pour montrer que ce problème est NP-complet, nous allons montrer que **X3-SAT** \propto **CIRCUIT HAMILTONIEN**.

La transformation : Soit $F = C_1 \wedge C_2 \wedge \dots \wedge C_q$ une instance de **X3-SAT**, avec $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, les $l_{i,j}$ étant des littéraux. Soient x_1, x_2, \dots, x_n les variables utilisées dans la formule. Nous construisons un graphe orienté G composé de deux types de sous-graphes. A chaque variable x_i on associe un sous-graphe H_i comme sur la figure ci-contre :

Les sous-graphes H_i disposent chacun d'un point d'entrée unique a_i et un point de sortie unique d_i et sont connectés en circuit, c.a.d. on a un arc du sommet d_i vers le sommet $a_{(i+1) \bmod n}$.

Le nombre de clauses dans la formule est q . Chaque sous-graphe H_i a la forme d'un « diamant » composé de sommets a_i, b_i, c_i , et d_i . Ces quatre sommets sont reliés par quatre arcs, de a_i vers b_i et c_i , et de b_i et c_i vers d_i . De plus on a un chemin orienté dans les deux sens entre b_i et c_i . Ce chemin est composé de sommets $u_{i,j}$ et $v_{i,j}$ (avec $1 \leq j \leq q$), dans l'ordre $u_{i,0}, v_{i,0}, u_{i,1}, \dots$. Comme dans la première preuve, ces sous-graphes ont une structure assez particulière. En effet, comme la seule



« entrée » de H_i est a_i , le parcours commence par ce sommet. Un chemin hamiltonien de H_i doit donc commencer en a_i et se terminer en d_i , en passant par b_i , le chemin et c_i ou bien commencer en a_i et se terminer en d_i , en passant par c_i , le chemin et b_i . Ainsi, on ne peut parcourir H_i que selon l'un des deux chemins hamiltoniens possibles :

$a_i, b_i, u_{i,1}, v_{i,1}, u_{i,2}, v_{i,2}, \dots, u_{i,q}, v_{i,q}, c_i, d_i$ ou
 $a_i, c_i, v_{i,q}, u_{i,q}, v_{i,q-1}, u_{i,q-1}, \dots, v_{i,1}, u_{i,1}, b_i, d_i$.

Nous préconisons le choix du premier parcours si la variable x_i représentée par le sous-graphe H_i est vraie et le deuxième parcours dans le cas contraire.

Pour finir la transformation nous avons besoin d'une information supplémentaire, qui correspondra aux clauses de la formule. A chaque clause $C_j = l_{j,1} \vee l_{j,2} \vee l_{j,3}$, nous associons un sommet g_j . Comment ce sommet sera relié au reste du graphe ? Si $l_{j,k} = x_i$, ($k=1,2,3$) alors il y a un arc de $u_{i,j}$ vers g_j et un arc de g_j vers $v_{i,j}$. Sinon, si $l_{j,k} = \neg x_i$, ($k=1,2,3$) alors il y a un arc de $v_{i,j}$ vers g_j et un arc de g_j vers $u_{i,j}$.

Ceci termine la construction de notre graphe. Ce graphe a $n(2q+4)+q$ sommets et $n(2q+6)+6q$ arcs (n est le nombre de variables de la formule transformée et q le nombre de clauses). La construction se fait en temps polynomial.

Pour comprendre la transformation, il faut poser la question quelle est la forme des circuits hamiltoniens que ce graphe peut admettre. Un premier type évident consiste en l'enchaînement des chemins hamiltoniens dans les H_i , avec des « détours » par des sommets g_j . Peut-on avoir d'autres types ? Supposons qu'un circuit contient un arc d'un H_i vers g_j avec retour vers un H_k ($k \neq i$). Soit donc $u_{i,j}$ le prédécesseur de g_j sur le circuit. Comment peut-on passer par le sommet $v_{i,j}$? Ce sommet n'a que trois prédécesseurs, $u_{i,j}$, g_j et $u_{i,j+1}$,

dont les deux premiers déjà visités, donc son prédécesseur sur le circuit doit être $u_{i,j+1}$. De même, ce sommet n'a que trois successeurs, $u_{i,j}$, g_{j+1} et $u_{i,j+2}$. Comme tous les trois sont déjà visités, on ne peut plus « quitter » ce sommet, donc on ne peut pas avoir de circuit hamiltonien de ce type. Comme dans la preuve précédente, nous avons deux choix de parcours pour chaque H_i , et ainsi les 2^n différents circuits hamiltoniens correspondent aux 2^n différentes valuations de notre formule de départ. Lors d'un parcours de H_i , on peut aussi parcourir un sommet g_j , si on a choisi le parcours correspondant de H_i . Ainsi si on a une affectation de valeurs de vérité qui satisfait la formule, on peut en déduire un circuit hamiltonien du graphe construit. De la même manière, si on dispose d'un circuit hamiltonien du graphe, c'est qu'on a visité chacun des sous-graphes G_j , ce qui implique que dans le sous-graphe H associé à un des variables de la clause, nous avons choisi une valeur de vérité qui satisfait la clause. \square

5.4 Autres problèmes

Nous traiterons ici quelques problèmes NP-complets supplémentaires parmi les plus connus : le *couplage en trois dimensions* (**3DM**), la *partition* (**Partition**), les *sommes partielles* (**SSP**) et le *sac à dos* (**KNAPSACK**). Rappelons d'abord les énoncés de ces problèmes :

- NOM** : **3DM** (Couplage en 3 dimensions)
DONNÉES : un ensemble M de triplets (w,x,y) , avec w , x et y des éléments de trois ensembles W , X , Y de même cardinalité q .
QUESTION : M contient-il un couplage (un sous-ensemble de triplets contenant tous les éléments une fois et une seule) ?
- NOM** : **PARTITION**
DONNÉES : un ensemble fini d'entiers non négatifs A .
QUESTION : existe-t-il une partition de A en deux ensembles A' et A'' , telle que la somme des éléments de A' soit égale à la somme des éléments de A'' .
- NOM** : **KNAPSACK** (sac à dos)
DONNÉES : un ensemble fini d'objets, E , avec deux fonctions entières, v et p , associant à chaque objet une valeur et un poids. Un poids total autorisé P et une valeur totale minimale V .
QUESTION : peut-on choisir des objets (à mettre dans le sac) de manière à ne pas dépasser le poids total autorisé, et que le total des valeurs soit supérieur ou égal à V ?
- NOM** : **SSP** (somme de sous-ensembles - *subset sum problem*)
DONNÉES : un ensemble fini d'entiers non négatifs A et un entier naturel S .
QUESTION : existe-t-il un sous-ensemble A' de A , tel que la somme des éléments de A' soit égale à S ?

Théorème [Karp, 1972] : 3DM est NP-complet.

Preuve : L'appartenance à NP est évidente. En effet une machine non-déterministe peut choisir les triplets du couplage de manière non-déterministe. Par la suite il ne reste que la

vérification, qu'il s'agit en effet d'une solution, et ceci peut se faire sur une machine déterministe. Le tout en temps polynomial.

Pour la NP-difficulté, nous allons réduire le problème X3-SAT à 3DM.

La transformation :

Soit $F = C_0 \wedge C_1 \wedge \dots \wedge C_{k-1}$ une instance de 3-SAT qui utilise les variables x_1, x_2, \dots, x_n .

Nous construisons (données de 3DM) :

Les triplets de M seront dans trois groupes

- le premier pour « choisir » une valeur de vérité
- le second pour assurer la satisfiabilité
- le troisième pour « arrondir les angles »

Le premier groupe : \forall variable x_i , $1 \leq i \leq n$ on a $2k$ triplets :

- $G_i = \{ (a_{ij}, x_{ij}, b_{ij}), (b_{ij}, \neg x_{ij}, a_{i,(j+1) \bmod k}) \mid 0 \leq j < k \}$
- pour les éléments a_{ij} et b_{ij} ($1 \leq i \leq n$ et $0 \leq j < k$)
- ce sont les seuls triplets qui les contiennent.

On peut remarquer qu'il y a exactement deux manières de recouvrir les éléments a_{ij} et b_{ij} ($1 \leq i \leq n$ et $0 \leq j < k$) :

- soit on utilise les triplets qui contiennent les x_{ij} mais pas les $\neg x_{ij}$, (ce qui correspond à x_i faux) ;
- soit l'inverse (ce qui correspond à x_i vrai).

Ainsi, les triplets choisis impliquent un choix de valeur de vérité.

Le deuxième groupe : pour vérifier la satisfiabilité, on construit pour chaque clause C_j , $0 \leq j < k$, et pour chaque littéral l de C_j , un triplet $(c_{j,1}, l, c_{j,2})$.

Ce sont les seuls triplets contenant $c_{j,1}$ et $c_{j,2}$. Le nombre de triplets : dépend du degré de la clause.

Si les triplets de type 1 sont choisis (ce qui revient à un choix de vérité des variables), alors $c_{j,1}$ et $c_{j,2}$ peuvent être couverts si $\exists x$ variable de C_j , qui n'est pas couverte par les triplets du type 1. Ainsi $c_{j,1}$ et $c_{j,2}$ peuvent être couverts si et seulement si la clause est vraie.

Pour compléter la preuve il reste à compléter avec des triplets qui permettent de couvrir les x_{ij} non encore couverts.

Le troisième groupe : on veut couvrir les x_{ij} (et $\neg x_{ij}$) non encore couverts. Combien de tels éléments existent ? Nous avons n variables, k clauses, donc non couverts par des triplets de type 1 nk éléments, dont k sont couverts par des triplets de type 2. En tout $nk-k$.

On introduit les triplets :

- $\{(h_r, x_{ij}, g_r) \mid 1 \leq r < nk-k, 1 \leq i \leq n, 0 \leq j < k\}$
- $\{(h_r, \neg x_{ij}, g_r) \mid 1 \leq r < nk-k, 1 \leq i \leq n, 0 \leq j < k\}$

Les trois ensembles contiennent chacun $q = 2nk$ éléments.

- $W = \{ a_{ij}, c_{j,1}, h_r \mid 1 \leq i \leq n, 0 \leq j < k, 1 \leq r < nk-k \}$
- $X = \{ x_{ij}, \neg x_{ij} \mid 1 \leq i \leq n, 0 \leq j < k \}$
- $Y = \{ n_{ij}, c_{j,2}, g_r \mid 1 \leq i \leq n, 0 \leq j < k, 1 \leq r < nk-k \}$

Le nombre de triplets est :

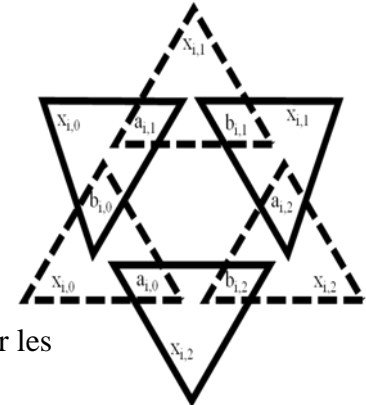
- type 1 : $2nk$
- type 2 : entre k et $3k$, en fonction des degrés
- type 3 : $2n(n-1)k^2$

La transformation est donc polynomiale. Le « si et seulement si » est assuré par la construction. \square

Nous montrons maintenant que les trois autres problèmes mentionnés sont NP-complets.

Théorème [Karp, 1972] : Partition est NP-complet.

Preuve : i) il est facile de voir que **Partition** \in **NP**. En effet, on peut choisir de manière



non-déterministe pour chaque élément de \mathbf{A} , s'il doit se trouver dans \mathbf{A}' ou dans \mathbf{A}'' (en temps linéaire). Ensuite, il ne reste qu'à vérifier que la somme des éléments de \mathbf{A}' est égale à la somme des éléments de \mathbf{A}'' , calcul qui se résume au calcul d'un nombre linéaire de sommes partiels consécutifs.

ii) Nous réduisons le problème **3DM** au problème **Partition**. La transformation sera la suivante :

Soient W, X, Y les trois ensembles de **3DM**, de cardinalité q chacun, et soit M l'ensemble des triplets, de cardinalité k . Nous allons construire un ensemble d'entiers naturels A de cardinalité $k+2$. Soit $p = \lceil \log_2(k+1) \rceil$. Nous allons construire à partir du triplet m_i de M un nombre naturel a_i (sous sa forme binaire), comme suit : a_i sera de longueur $3qp$, étant composé de $3q$ blocs de longueur p , les blocs correspondant aux $3q$ éléments de W, X et Y . Les blocs seront composés que de 0, sauf les trois blocs correspondants aux trois éléments de m_i , dans lesquels le bit le moins significatif sera 1, les autres des 0. Ce choix nous permet de détecter facilement si un sous-ensemble de nombres correspond à un couplage en trois dimensions, car il suffit de tester que la somme des nombres est composée de $3q$ blocs identiques, de la forme $00\dots 01$ (nombre qu'on notera par B dans la suite). Comme les blocs ont été choisis suffisamment longs, il ne peut y avoir de débordement d'un bloc vers un autre même si on calcule la somme de tous les k nombres obtenus. Soit $S = \sum_{1 \leq i \leq k} a_i$. Pour finir la transformation, nous introduisons deux nombres supplémentaires, $a_{k+1} = 2S - B$ et $a_{k+2} = S + B$.

La transformation est polynomiale, car elle peut s'effectuer en temps $O(n^2)$, pour une donnée de taille n . En effet, un premier parcours permet de calculer k et q . Lors d'un deuxième parcours on calcule les a_i . Montrons qu'il s'agit bien d'une réduction. Remarquons d'abord, que la somme des éléments de A est $S + 2S - B + S + B = 4S$.

Si : supposons que M admet un sous-ensemble M' , constituant une solution au problème **3DM**. Soit A' constitué de l'image des triplets de M' et de l'élément a_{k+1} . La somme des éléments de A' sera donc $B + 2S - B = 2S$, ce qui est la moitié de la somme totale.

Seulement si : Supposons avoir une partition de A en deux parties de somme égale. Soit A' la partie contenant a_{k+1} . Comme la somme des éléments de A' doit être la moitié de la somme totale, $2S$, la somme des autres éléments de A' doit être B , ce qui implique que ces éléments sont issus d'un couplage en trois dimensions. \square

Nous pouvons maintenant utiliser le problème **Partition** pour montrer que le problème **SSP** est **NP-complet**.

Théorème [Karp, 1972] : **SSP** est **NP-complet**.

Preuve : Il suffit de réaliser que le problème **Partition** n'est autre qu'un cas particulier du problème **SSP**. En effet, dans le problème **partition** on doit répondre à la question, est-ce qu'il existe un sous-ensemble dont la somme est la moitié de la somme totale, alors que dans le problème **SSP** on doit répondre à la même question, pour une somme donnée, mais quelconque (évidemment on peut présenter ceci comme une transformation, avec la fonction identité pour les nombres et l'affectation de $1/2 \sum a_i$ pour S). \square

La même approche nous permet de prouver le résultat suivant :

Théorème : **KNAPSACK** est **NP-complet**.

Preuve : En effet il suffit de considérer que le problème **SSP** est un cas particulier du problème du sac à dos, dans lequel la valeur et le poids des objets sont identiques, et la somme totale et la valeur totale sont aussi identiques et égales à S . Ainsi on cherche un sous-ensemble de somme totale ayant comme borne inférieure et supérieure S , donc égale à S . \square

5.5 Problèmes NP-complets au sens fort

Comme nous l'avons vu, pour pouvoir parler de la complexité d'un problème il faut absolument préciser la manière dont les données sont codées et ainsi disposer d'une estimation de l'espace mémoire utilisé. Par exemple, nous avons montré en TD que le problème **Nombre composé** est en **NP** et aussi que si les données sont en unaire alors le problème est dans **P**.

Précisons donc un peu plus les mesures de complexité. Soit n la taille des données et M la plus grande valeur des données. Ainsi nous savons déjà que M est en $O(2^{n^c})$ pour un certain constant c . Etant donné un algorithme, nous distinguerons les trois cas suivants :

- Algorithme *fortement polynomial* : l'algorithme est polynomial en n .
- Algorithme *faiblement polynomial* : l'algorithme est polynomial en n et $\log(M)$.
- Algorithme *pseudo-polynomial* : l'algorithme est polynomial en n et M .

Par exemple, l'algorithme intuitif qui teste tous les nombres inférieurs à un nombre s'ils divisent la donnée est un algorithme pseudo-polynomial. Pour certains problèmes, comme par exemple le **KNAPSACK** ou **Partition** on a des algorithmes de programmation dynamiques qui sont pseudo-polynomiaux.

L'ensemble des problèmes **NP-complets** peut à son tour être partitionné en deux catégories :

- Les problèmes **NP-complets au sens faible** (ou *faiblement NP-complets*) : les problèmes qu'on peut résoudre en temps polynomial si les données sont en unaire.
- Les problèmes **NP-complets au sens fort** (ou *fortement NP-complets*) : les problèmes qui restent **NP-complets**, même si les données sont en unaire.

On remarque que les problèmes dont les données ne sont pas numériques sont ainsi fortement **NP-complets**. Comme exemple, les versions de **SAT**, les problèmes de graphes étudiés dans ce cours, **3DM** sont fortement **NP-complets**.

En ce qui concerne les problèmes numériques, cela dépend du problème. Un premier exemple de problème **NP-complet au sens fort** :

NOM : triplets-Partition²

DONNÉES : Un nombre nature T , un ensemble de nombres naturels $A = \{a_1, a_2, \dots, a_{3s}\}$, dont la somme est Ts et tels que $T/4 < a_i < T/2$. Les nombres a_i sont donnés en unaire.

QUESTION : Peut-on partitionner l'ensemble en s ensembles A_1, A_2, \dots, A_s , tels que la somme des nombres dans chaque A_i est T ?

On peut remarquer la condition sur la taille des a_i implique que si une telle partition existe alors chaque A_i contient exactement trois nombres.

Le résultat suivant qui prouve la NP-complétude forte du problème est donné sans preuve :

Théorème [Garey & Johnson, 1975] : triplets-Partition est NP-complet.

² Dans la littérature ce problème s'appelle souvent 3-Partition. Comme nous avons déjà en TD porté ce nom, nous l'appellerons triplets-partition.

5.6 Jeux NP-complets

Bien souvent les jeux intéressants s'avèrent être NP-complets. Citons, comme exemple les jeux démineur, mastermind, othello (reversi) et go. Nous proposons ici la preuve de la NP-complétude du problème **DEMINEUR**³ et celui de **TETRIS**.

NOM : **DEMINEUR**

DONNÉES : un rectangle fini avec certaines cases contenant des bombes ou des valeurs.

QUESTION : est-ce qu'il existe une solution à ce problème de démineur ?

Théorème [Kaye, 2000⁴] : **DEMINEUR** est NP-complet.

Preuve : Il est facile de vérifier (en temps polynomial, sur une machine déterministe) une solution du problème. Ainsi le problème est dans **NP**.

Pour la **NP**-difficulté, nous réduisons 3-SAT à **DEMINEUR**. L'idée est d'associer un problème de démineur à une formule, de manière à assurer que le problème admet une solution si et seulement si la formule est satisfiable. La construction se fait à l'aide de briques de « *LEGO* » (qu'on appellera les connecteurs) qui permettent d'assurer les différentes opérations.

Le connecteur le plus simple est un fil (fig. 1), qui permet simplement de « propager » la valeur x . D'autres fils, sont ceux qui tournent à 90 degrés et ceux ayant un bout (fig. 2 et 3).

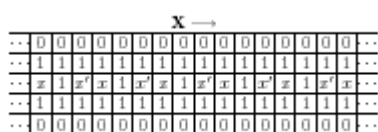


fig. 1

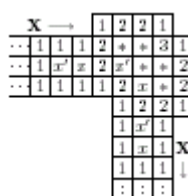


fig. 2

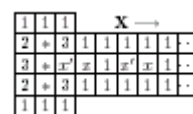


fig. 3

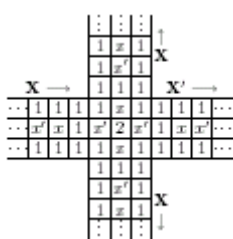


fig. 4

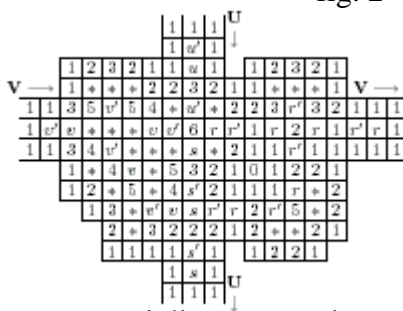


fig. 5










fig. 6

Les figures 4, 5 et 6 présentent un triplique, un croisement de fils et une négation.

³ C'est le nom commercial de l'implémentation qui fait partie de la distribution Windows.

⁴ Nous utilisons les figures de la preuve originale de R. Kaye [8].

Dans le jeu on utilise les pièces suivantes :

Forme	Notation	Description
	I	Quatre carrés alignés.
	Sq	Carré de 2x2.
	T	Trois carrés en ligne et un carré sous le centre.
	RG	Trois carrés en ligne et un carré sous le côté gauche.
	LG	Trois carrés en ligne et un carré sous le côté droit.
	LS	Carré de 2x2, dont la rangée supérieure est glissée à gauche.
	RS	Carré de 2x2, dont la rangée supérieure est glissée à droite.

Pour pouvoir parler de complexité du problème on est amenés à passer à une version information parfaite :

- On connaît à l'avance la suite entière des blocks
- On a une position initiale donnée

Ainsi on obtient le problème suivant :

NOM : **TETRIS**

DONNÉES : Un rectangle fini avec des blocs placés, ce qui correspond à un état initial. Une liste de blocs qui tomberont dans l'ordre.

QUESTION : Est-ce qu'il existe une solution à ce problème de démineur qui permet d'aboutir à un rectangle vide ?

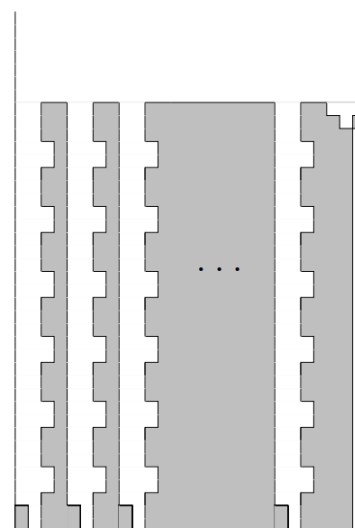
Théorème [Demaine et autres, 2004] : TETRIS est NP-complet.

Preuve : Une prétendue solution consiste en le placement des différents blocks avec la manière dont il faut le tourner. Comme il est facile de vérifier (en temps polynomial, sur une machine déterministe) une solution du problème, le problème est dans **NP**.

Pour la **NP**-difficulté, nous réduisons **triplets-Partition** à **TETRIS**. L'idée est de construire un tableau de jeux original et une liste de blocs associés à chaque nombre a_i . On suppose-
ra que T est un multiple de 4 (autrement on multiplie tous les nombres par 4).

Le tableau original sera le suivant :

Les parties grisées sont occupées par des blocs et les parties en blanc sont « libres ». La hauteur des parties grisées est de $5T+18$ lignes et au-dessus il y a encore $2s+O(1)$ lignes sans blocks. La largeur est de $4s+3$ colonnes. Dans ce schéma nous avons « à remplir » les colonnes $(1,2,3)$, $(5,6,7)$, ..., $(4s-3,4s-2,4s-1)$ s ensembles de colonnes de forme identique et aussi la case supérieure dans la colonne $4s+1$, les deux cases supérieures dans la colonne $4s+2$ et la colonne $4s+3$ qui est presque complètement vide (à part la ligne $5T+17$).



Pour les pièces on transforme chaque a_i dans une liste de blocs :

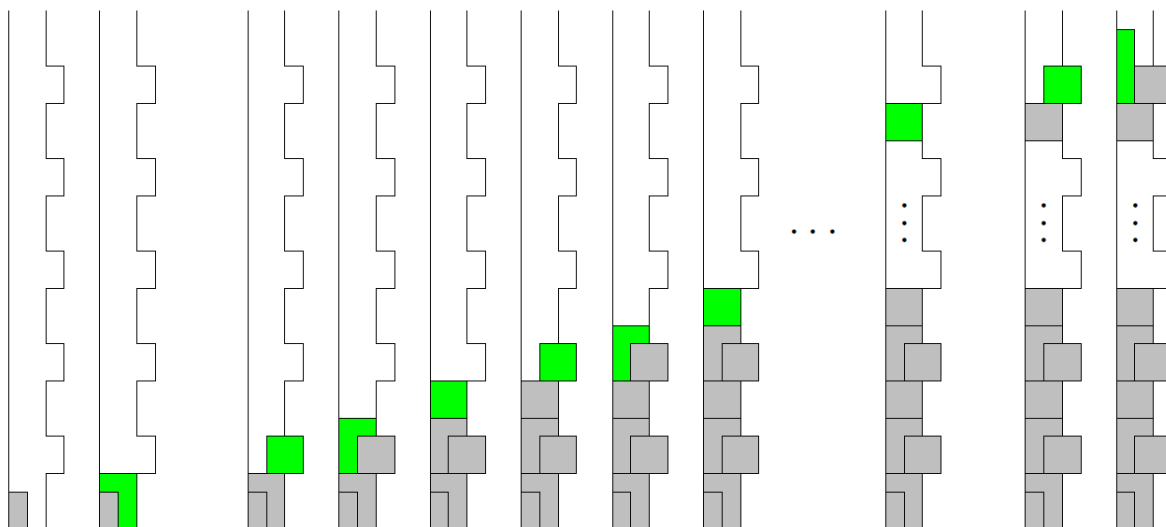
$$f(a_i) = \text{RG} (\text{Sq LG Sq})^{a_i} \text{Sq I}$$

A la fin on rajoute les pièces suivantes :

$$\text{RG}^s \text{T I}^{(5T+16)/4}$$

Ainsi la liste est d'une longueur linéaire en la taille des données **triplets-Partition**, et comme ces données sont en unaire nous avons une transformation polynomiale.

Supposons que **triplets-Partition** admet une solution. On décidera de mettre les pièces qui correspondent aux nombres dans A_i dans la i -ième ensemble de 3 colonnes, c'est-à-dire les colonnes $(4i-3, 4i-2, 4i-1)$. Voici comment on peut placer les blocks qui correspondent à un nombre :



On remarque qu'on ne laisse aucun vide dans l'ensemble des 3 colonnes et qu'après les avoir posés, la configuration reste la même qu'avant, donc on peut continuer avec la liste qui correspond au nombre suivant dans le même A_i .

Comme la somme des nombres dans un n'importe quel A_i est la même, ces ensembles de colonnes seront remplis de la même manière et jusqu'à la même hauteur et on obtiendra le tableau suivant lorsqu'on a placé les blocks issus des nombres a_i :

Il faut continuer avec les s blocs RG qui permettent de remplir complètement toutes les colonnes, sauf les trois dernières.

Le bloc T qui arrive permet de remplir et donc supprimer les deux lignes du haut. Ainsi il reste uniquement la dernière colonne à remplir, de hauteur $5T+16$.

C'est à cela que serviront les $(5T+16)/4$ blocs de type I qui arrivent, et ainsi le tableau sera vide.

Donc si **triplets-Partition** admet une solution alors le problème **TETRIS** admet une solution.

La preuve dans l'autre sens est presque la même. Supposons que **TETRIS** admet une solution. Alors, on ne peut à aucun moment placer un block à une place plus haut que la ligne $5T+18$, car selon le décompte on ne pourra pas tout vider ensuite. Donc on doit tout placer dans les $4S$ premières colonnes avant l'arrivée du seul bloc T dans les données. Comment peut-on placer les blocs issus d'un a_i ? Comme on ne doit pas laisser du vide, il faut choisir un des ensembles de 3 colonnes et placer le block RG. Ce block est suivi par un carré. Si on place ce carré dans un autre ensemble de 3 colonnes, cela crée du vide. Donc il faut placer

le carré juste au-dessus. Arrive un bloc LG, qui par le même raisonnement doit être placé au-dessus et ainsi de suite, tous les blocs issus d'un a_i doivent être ensemble. Et l'ensemble de 3 colonnes est « prêt » à recevoir la transformée d'un autre a_i . Ainsi dans chaque groupe de colonnes nous aurons des images d'éléments de A . Et comme on n'a pas le droit de dépasser la hauteur et il faut laisser la place à la fin pour les s blocks de type RG, tous doivent terminer à la même hauteur. Il suffit donc de créer les ensembles A_i par les nombres dont l'image est dans le groupe de colonne correspondant pour trouver la solution de **triple-Partition**. \square

CHAPITRE 6

Conclusions

6.1 Méthodes de preuve de NP-complétude

En considérant la longue liste de problèmes **NP**-complets connus, la tâche de prouver qu'un problème est **NP**-complet est devenue plus facile. En effet, pour la preuve, une fois qu'on a prouvé que le problème est dans **NP**, on a un large choix de problèmes **NP**-complets connus pour la réduction. Bien souvent, on peut trouver un problème qui est un cas particulier de celui qu'on essaye de prouver, et dans ce cas la preuve devient facile, car la transformation est l'identité.

Comme exemple, nous donnons ici le problème du planning des multiprocesseurs :

- NOM** : **PM** (planning de multiprocesseur)
DONNÉES : un ensemble de tâches à réaliser (avec le temps nécessaire pour chacune), le nombre de processeurs et un temps total **T**.
QUESTION : peut-on répartir les tâches sur les processeurs de manière à ce que toutes les tâches soient finies en temps **T** ?

Théorème : **PM** est **NP**-complet.

Preuve :

- i) **PM** est dans **NP**. En effet il suffit de répartir de manière non-déterministe les tâches sur les processeurs et vérifier ensuite que le temps total ne dépasse pas **T**.
- ii) **PM** est **NP**-complet. Pour la réduction nous choisissons le problème **NP**-complet **PARTITION**. Il faut donc montrer que **PARTITION** \propto **PM**. La réduction est facile, car en quelque sorte **PARTITION** est généralisé par **PM**. En effet il s'agit du cas particulier de **PM**, où le nombre de processeurs est deux et le temps **T** est la moitié du temps total.

□

Evidemment, il serait faux de dire que toutes les preuves sont aussi faciles. Dans certains cas on doit construire des preuves qui utilisent des transformations dites « locales » et parfois des transformations globales, qui nécessitent le changement de toute la structure.

On peut parler en général de trois types de preuves de **NP**-complétude :

- i) *les transformations identité (restrictions)* : il s'agit de cas où le problème à réduire peut être considéré comme un cas particulier du problème qu'on cherche à résoudre. Dans ce cours nous avons vu de tels exemples, lors des preuves des problèmes **CLIQUE**, **SSP**, **KNAPSACK**, **PM**. C'est dans cette catégorie que se trouvent la majorité des problèmes traités lors des séances de travaux dirigés. Sans doute, ce sont les cas les plus faciles.
- ii) *les transformations locales* : il s'agit de cas où la transformation se résume à une assez légère modification des structures, qui conservent leur caractère « local ». Les preuves sont assez simples dans ces cas, mais, contrairement au

cas précédent, nécessaires quand même. Comme exemples de preuves de ce type citons les preuves des différentes versions de **SAT**, **Partition**, **PPET** (le plus petit ensemble de tests - « *minimum test collection* »).

- iii) *les transformations globales* : il s'agit de transformations qui nécessitent une modification profonde d'un problème vers un autre. Nous pouvons citer dans cette catégorie les preuves de **SAT**, **VC**, **CIRCUITHAM**.

Evidemment ces trois catégories ne sont pas définies de manière suffisamment précise, mais permettent de donner une idée sur les approches possibles ou de la difficulté d'une preuve.

6.2 Que fait-on quand même?

On ne peut pas conclure ce chapitre sans évoquer les solutions de rechange. On a vu que les problèmes **NP**-complets sont difficilement abordables, à cause du temps de calcul. Que peut-on faire quand même ? Ils existent des *heuristiques*, des méthodes assez rapides pour trouver des solutions approchées. De plus, certaines méthodes permettent d'obtenir pour certains problèmes des solutions aussi proches de l'optimal qu'on veut.

Ces méthodes seront (brièvement) abordées lors du dernier cours.

Bibliographie

La littérature sur la complexité est abondante. Nous citerons ici quelques ouvrages, recommandés soit pour leur présentation particulièrement claire soit pour leur accessibilité.

- [1] **M.D. Davis** et **E.J. Weyuker** : «*Computability, Complexity & Languages*», Academic Press, 1983.
- [2] **J.H. Hopcroft** et **J.D. Ullman** : «*Formal languages and their relation to automata*», Addison-Wesley, 1969.
- [3] **M.R. Garey** et **D.S. Johnson** : «*Computers and intractability. A guide to the theory of NP-completeness*», Freeman, New York, 1979.
- [4] **K. Mehlorn** : «*Data Structures and Algorithms*» vol 2 : "*Graph Algorithms and NP-completeness*", Springer-Verlag, 1984.
- [5] **T.H. Cormen, C.E. Leiserson** et **R.L. Rivest** : «*Introduction to Algorithms*», MIT Press, 1990.
- [6] **C.H. Papadimitriou** : «*Computational Complexity*», Addison-Wesley, 1994.
- [7] **M. Sipser** : «*Introduction to the Theory of Computation*», PWS, 1997.
- [8] **R. Kaye** : «*Minesweeper is NP-complete*», Mathematical Intelligencer, 22:9-15, 2000

Table des matières

Avant-propos	1
1 Introduction	3
2 Les machines de Turing : un modèle de calcul	5
2.1 Description	5
2.2 Variantes de la machine	6
2.3 Equivalence des différents modèles	8
2.4 Le théorème de l'accélération	9
2.5 Les suites de passages	11
2.6 La reconnaissance des palindromes impairs	12
3 Les classes P et NP	15
3.1 Quelques notions de complexité	15
3.2 La réduction polynomiale	16
3.3 Quelques autres exemples de réduction polynomiale	17
3.4 P et NP	19
4 NP-complétude	21
4.1 La notion	21
4.2 Le théorème de Cook	21
4.3 Variantes de SAT	25
5 Problèmes NP-complets connus	29
5.1 Quelques problèmes connus	29
5.2 Les problèmes de transversal, clique et stable	30
5.3 Les problèmes de hamiltonisme	31
5.4 Autres problèmes	35
5.5 Problèmes NP-complets au sens fort	38
5.6 Jeux NP-complets	39
6 Conclusions	45
6.1 Méthodes de preuve de NP-complétude	45
6.2 Que fait-on quand même?	46
Bibliographie	47
Table des matières	49

