

Emmanuel Jeandel

COMPLEXITÉ

TABLE DES MATIÈRES

1	Machines de Turing	1
1.1	Machine de Turing classique	1
1.2	Variantes du modèle	5
1.3	Représentation des données	7
2	Complexité en temps et en espace	9
2.1	Définitions	9
2.2	Théorèmes de hiérarchie - Premières relations	11
2.3	Théorèmes de hiérarchie - Diagonalisation	12
2.3.1	Espace	14
2.3.2	Temps	16
3	Non déterminisme	19
3.1	Définitions	19
3.2	Premières propriétés	20
3.3	Liens avec les classes déterministes	21
4	Classes de complexité	23
5	Quelques conséquences de $P = NP$	25
5.1	Factorisation	25
5.2	Certificats polynomiaux	26
5.3	Cryptographie - Fonction One-Way	28
5.4	Padding	29
6	Complétude	31
6.1	Réductions	31
6.2	Complétude	33
7	Quelques problèmes NP-complets	39
7.1	Méthode	39
7.2	Variantes de SAT	39
7.3	CLIQUE	41
7.4	Couplages	43
8	Problèmes NP-complets sur les nombres	45
8.1	Une autre variante de SAT	45
8.2	SUBSET-SUM et les variantes	46

8.2.1	NP-complétude	46
8.2.2	Un algorithme pour SUBSETSUM	47
8.3	Problèmes NP-complets au sens fort	48
9	Algorithmes d'approximation	51
9.1	BINPACKING	51
9.2	PARTITION	51
9.3	KNAPSACK	51
A	Index	53
B	Liste des définitions	55

INTRODUCTION

Voici une liste de livres dont je m'inspire pour le cours

- Carton, *Langages formels, calculabilité et complexité*
- Garey, Johnson, *Computers and Intractability, A guide to the Theory of NP-Completeness*
- Balcazar, Diaz, Gabarro, *Structural Complexity I & II*
- Hopcroft, Ullman, *Introduction to Automata Theory, Languages and Computation*
- Wegener, *Complexity Theory*
- Kozen, *Theory of Computation*
- Wagner, Wechsung, *Computational Complexity*

MACHINES DE TURING

Le formalisme des machines de Turing permet de définir un modèle de calcul précis, pour lequel la notion de complexité, et la notion de pas de calcul sera clairement définie. Par exemple, il est difficile d'évaluer exactement combien de temps et de mémoire un programme C va utiliser sur une entrée donnée, et il est encore plus difficile de définir ce qu'est exactement la notion de temps ou de mémoire.

Le fonctionnement d'une machine de Turing est très proche de ce qui se passe à bas niveau dans un ordinateur ordinaire, mais il est énormément simplifié.

1.1 Machine de Turing classique

Informellement, une machine de Turing consiste en la donnée d'un ruban muni d'une tête de lecture, et d'un état. La machine de Turing agira en fonction de cet état et de ce qui se trouve sur la tête de lecture.

Donnons un exemple concret : On considère la table suivante

	a	b	B
0	$(1, b, \leftarrow)$	$(0, b, \rightarrow)$	$(1, B, \rightarrow)$
1	$(2, b, \leftarrow)$	$(0, b, \rightarrow)$	$(3, B, \leftarrow)$
2	$(1, a, \leftarrow)$	$(3, a, \rightarrow)$	$(1, B, \rightarrow)$
3	$(1, a, \leftarrow)$	$(3, a, \rightarrow)$	$(1, B, \rightarrow)$

Sa signification est la suivante : Si on regarde à la ligne 2, colonne b , on voit marqué $(3, a, \rightarrow)$ ce qui signifie : si je suis dans l'état 2 et que je lis un b sur ma tête de lecture, alors :

- Je passe dans l'état 3
- Je mets un a à la place du b
- Je déplace ma tête à droite

Un premier exemple d'utilisation est décrit dans la figure 1.1. L'état est indiqué à l'intérieur de la tête de lecture. La machine s'arrête lorsqu'elle atteint l'état 3, puisqu'aucune règle ne spécifie ce qui se passe dans ce cas.

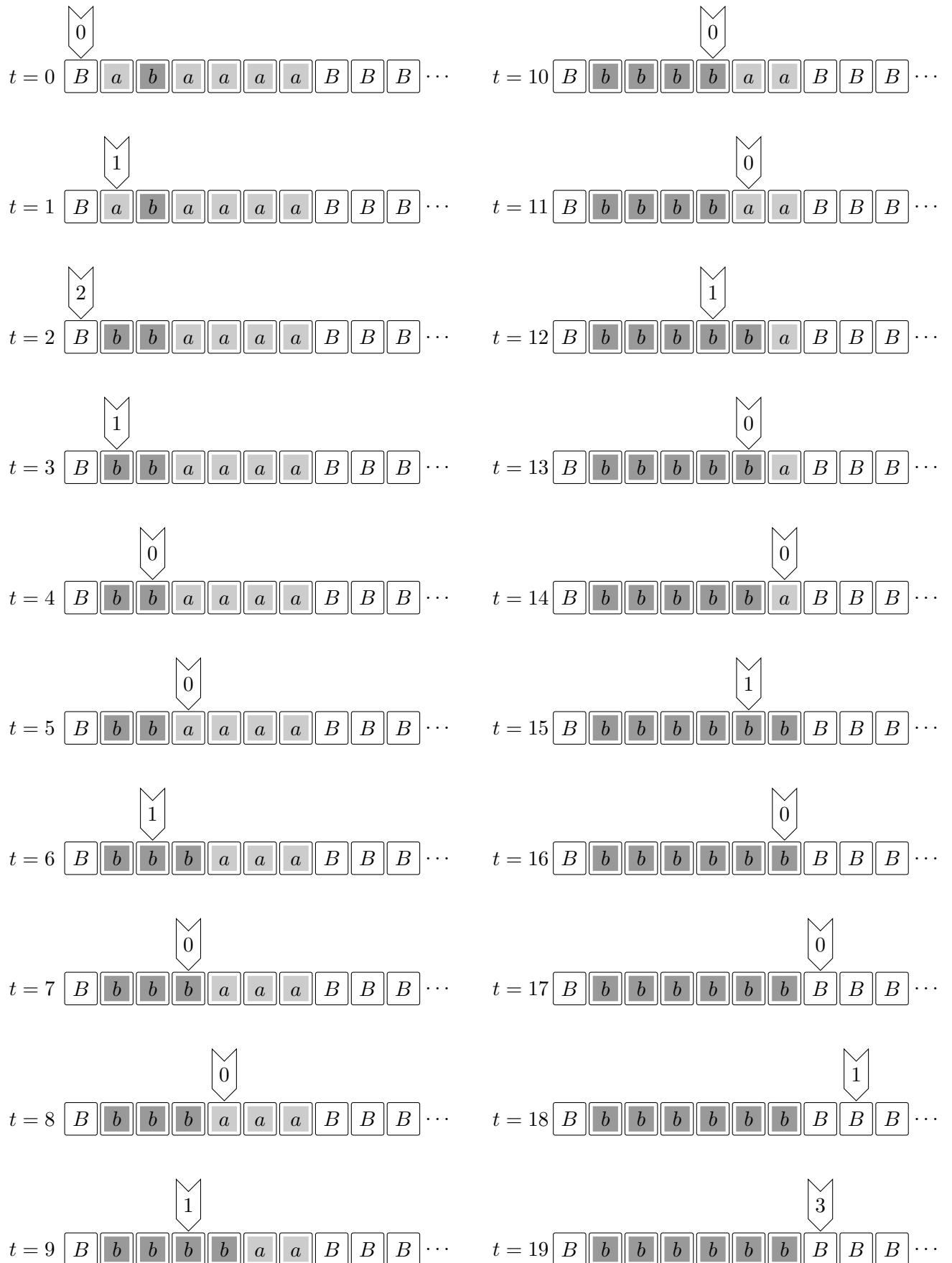


FIGURE 1.1 – Exécution d'une machine de Turing

Plus formellement, on définit une machine de Turing ainsi :

Définition 1.1 (Machine de Turing)

Une machine de Turing à un ruban est la donnée :

- D'un ensemble fini Q , appelé ensemble des états de la machine. Q contient deux états particulier, un état initial noté q_0 et un état final noté q_f .
- D'un alphabet Γ , qui est l'ensemble des symboles qui peuvent apparaître sur le ruban. Γ contient un symbole particulier, noté B , correspondant au blanc, c'est à dire à une case vide.
- D'un alphabet $\Sigma \subset \Gamma$ appelé alphabet d'entrée
- D'une fonction $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\leftarrow, \rightarrow\}$ appelé fonction de transition.

◆ **Exemple**

Formellement, l'exemple précédent s'écrit ainsi :

- $Q = \{0, 1, 2, 3\}$. $q_0 = 0$, $q_f = 3$
- $\Gamma = \{a, b, B\}$
- $\Sigma = \{a, b\}$
- δ est donnée par le tableau ci-dessus

Définition 1.2

Une *configuration* d'une machine de Turing est la donnée d'un couple constitué d'un état et d'un mot sur Γ contenant exactement une lettre soulignée.

Au vu de la définition, on ne reconnaît pas forcément l'objet qu'on a étudié sur la figure. L'endroit souligné correspond à l'endroit où se trouve la tête de lecture. Sur la figure, le ruban est infini : il contient à la première étape $BabaaaaBBBB \dots$. Dans tous les raisonnements, le ruban sera toujours constitué d'un mot (qui peut contenir des blancs) suivi par une infinité de symboles blancs. De sorte que la configuration $(3, \underline{B}aaB)$ et la configuration $(3, \underline{B}aaBBBBB)$ représentent en fait la même configuration, c'est à dire un ruban où se trouve $BaaBBBB \dots$, où la tête de lecture est en deuxième position, et où l'état de la machine de Turing est l'état 3.

◆ **Exemple**

La première configuration de l'exemple 1.1 est donnée par $(0, \underline{B}abaaaaB)$ et la dernière par $(3, Bbbbbb\underline{B}B)$.

Définition 1.3

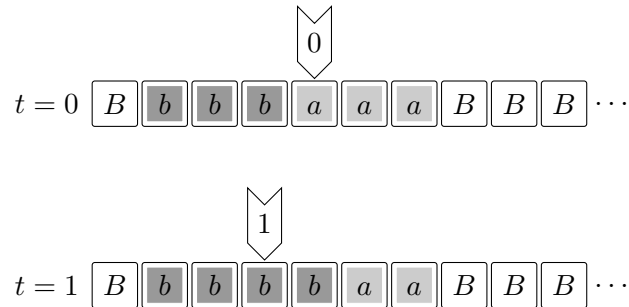
Soit $M = (Q, q_0, q_f, \Gamma, B, \Sigma, \delta)$ une machine de Turing. Si (q, w) et (q', w') sont des configurations, on dit que la machine de Turing passe de (q, w) à (q', w') , ce qu'on note $(q, w) \vdash (q', w')$ si :

- La lettre soulignée dans w est la lettre numéro k .
- $\delta(q, w_k) = (q', w'_k, D)$
- $\forall i \neq k, w_i = w'_i$ (Autrement dit, la seule lettre qui change est la k ème)
- Si $D = \leftarrow$ alors la lettre soulignée dans w' est la lettre de numéro $k - 1$. Si au contraire $D = \rightarrow$, c'est la lettre de numéro $k + 1$.

On appelle *pas de calcul* le passage d'une configuration à la configuration suivante.

◆ Exemple

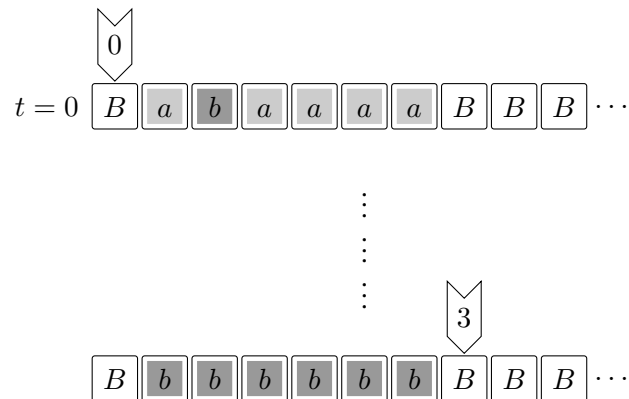
$(0, Bbb\underline{b}aaBB) \vdash (1, Bbb\underline{b}baaBB)$. C'est à dire :

**Définition 1.4**

Si (q, w) et (q', w') sont deux configurations, on dit que $(q, w) \vdash^* (q', w')$ s'il existe des configurations $(q_1, w_1) \dots (q_k, w_k)$ telles que $(q_1, w_1) = (q, w)$, $(q_k, w_k) = (q', w')$ et pour tout i , $(q_i, w_i) \vdash (q_{i+1}, w_{i+1})$. Autrement dit, la machine de Turing passe de (q, w) à (q', w') en plusieurs étapes.

◆ Exemple

$(0, \underline{B}abaaaaBB) \vdash^* (3, Bbb\underline{b}baaBB)$. C'est à dire :



Définition 1.5

Si w est un mot sur Σ , la configuration initiale associée à w est la configuration $(q_0, \underline{B}w)$.
La fonction calculée par la machine de Turing est définie de la façon suivante :

$$f(w) = \begin{cases} x & \text{si } (q_0, \underline{B}w) \vdash^* (q_f, Bx) \\ \perp & \text{sinon} \end{cases}$$

◆ Exemple

$$f(aba\text{aaaa}) = b\text{bbbbbb}$$

Résumé 1

Une machine de Turing part d'une configuration initiale, et effectue des pas de calcul jusqu'à tomber sur une configuration finale, c'est à dire dont l'état est l'état final.

1.2 Variantes du modèle

Le modèle à un ruban est pratique pour les raisonnements, mais on lui préfère en pratique d'autres modèles, qui lui sont équivalents (au sens où on peut calculer les mêmes fonctions, et en utilisant à *peu près* les mêmes ressources) :

Ruban infini Au lieu du ruban semi-infini (infini seulement dans une direction), on peut lui préférer un ruban infini dans les deux directions.

Proposition 1.1

Pour toute machine de Turing M à un ruban infini, on peut fabriquer une machine M' à un ruban semi-infini tel que $M(x) = M'(x)$ pour tout x et M' effectue sur l'entrée x à peu près autant d'étapes que M .

Preuve : Au tableau. ■

Tête qui reste sur place On peut également considérer un modèle dans lequel la tête de lecture de la machine a le droit de rester sur place : au lieu de \leftarrow, \rightarrow , on s'autorise également \downarrow .

Proposition 1.2

Pour toute machine de Turing M qui peut rester sur place, on peut fabriquer une machine M' "classique" tel que $M(x) = M'(x)$ pour tout x et M' effectue sur l'entrée x à peu près autant d'étapes que M .

Preuve (Idée): En ajoutant des états supplémentaires, on simule une machine qui reste sur place par une machine qui va à droite une fois puis à gauche. ■

Les fonctions calculées par les machines de Turing présentent des tas de propriétés que nous n'avons pas le temps d'explorer en cours.

Modèle à plusieurs rubans Un modèle qu'on va beaucoup rencontrer est le modèle à plusieurs rubans : Au lieu d'avoir un seul ruban, la machine de Turing en possède plusieurs : Sa fonction de transition tient compte maintenant de ce qu'elle lit sur chacun des rubans. De plus, les têtes de lecture sont indépendantes : la tête du premier ruban peut par exemple bouger sans que les têtes des autres rubans le fassent. Dans ce cas, l'entrée est donnée uniquement sur le premier ruban, et le deuxième est rempli initialement avec des symboles blancs.

La principale différence tient donc dans la fonction δ qui est maintenant du type :

$$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{\leftarrow, \rightarrow, \downarrow\}$$

Donnons-en un exemple : Il s'agit d'une machine à deux rubans. Toutes les transitions qui ne sont pas écrites dans le tableau ne servent pas dans le calcul. L'état initial est 0, l'état final 5.

	0	1	2	3	4	5
B B	$(1, B \rightarrow)$	$(2, B \leftarrow)$	$(3, B \rightarrow)$	$(5, a \rightarrow)$	$(5, b \rightarrow)$	
a B		$(1, a \rightarrow)$	$(2, a \leftarrow)$			
b B		$(1, b \rightarrow)$	$(2, b \leftarrow)$			
a a				$(3, B \rightarrow)$	$(4, B \rightarrow)$	
b b				$(3, B \rightarrow)$	$(4, B \rightarrow)$	
a b				$(4, B \rightarrow)$	$(4, B \rightarrow)$	
b a				$(4, B \rightarrow)$	$(4, B \rightarrow)$	
B a						
B b						

◆ **Exemple**

- $M(abbba) = \dots$
- $M(aabba) = \dots$

Autres On peut considérer des variantes de ces modèles, et également considérer un modèle qui utilise toutes les variantes simultanément.

En fait on démontre, sur le même principe que les démonstrations précédentes, que l'ensemble des fonctions calculées ne dépend pas de la définition précise

1.3 Représentation des données

Une machine de Turing travaille uniquement sur des mots. Cependant, la plupart des problèmes naturels ne s'expriment pas sur des mots, mais plutôt sur des graphes, des formules ou des entiers. Il faut donc transformer ces différentes données en des données utilisables par une machine de Turing, donc par des mots.

Le codage exact n'est pas important. Ce qu'il faut retenir, c'est qu'on peut facilement coder toutes les données dont on a besoin. Le paramètre important est la *taille* du codage : Dans tout le cours, la complexité s'exprime en effet en fonction de la taille de l'entrée :

- La taille d'un mot est son nombre de lettres. Par exemple, si on dit qu'un algorithme est en $2 \times n^5$, cela veut dire que sur le mot *aaabba*, il va mettre de l'ordre de 2×6^5 opérations.
- Comme un graphe est donnée par une matrice de 0 et de 1, la taille d'un graphe à k sommets est k^2 . Ainsi un algorithme en $2 \times n^5$ mettra de l'ordre de 2×36^5 opérations sur un graphe à 6 sommets.
- La taille d'une formule sous forme CNF est de l'ordre de son nombre total de littéraux.
- La taille d'un entier prête à discussion : Il y a plusieurs façons de représenter un entier sur machine de Turing.
 - On peut représenter l'entier k par k symboles 1 consécutifs : 12 sera représenté par 111111111111.
 - On dit alors que l'entier est représenté en unaire.
 - On peut représenter l'entier n en binaire : 13 sera représenté par 1101.

Dans le premier cas, la taille de l'entrée est de la même taille que l'entier : Un algorithme en $2 \times n^5$ sur l'entrée 13 mettra de l'ordre de 2×13^5 opérations. Dans le deuxième cas, la taille de l'entrée est *logarithmique* en la taille de l'entier : Un algorithme en $2 \times n^5$ sur l'entrée 13 mettra de l'ordre de 2×4^5 opérations, puisque 13 s'écrit avec 4 chiffres seulement en binaire.

La représentation la plus utilisée est celle en binaire. Il faut y faire attention : Ainsi si un algorithme met k étapes sur l'entier k , cela veut donc dire que l'algorithme est exponentiel en la longueur de l'entrée, et donc pas très performant.

C'est le cas par exemple de l'algorithme classique pour savoir si un nombre est premier :

```
int prime(int x){
int i;
for (i = 2; i < x; i++)
{
if (x % i == 0)
return 1;
}
return 0;
}
```

Le temps d'exécution de cet algorithme est de l'ordre de k sur l'entier k , il n'est donc pas efficace.

Pour obtenir un bon algorithme, il faudrait que le temps d'exécution soit de l'ordre de $\log k$, ou de $\log^2 k$ etc, c'est à dire polynomial en la taille de l'entrée.

COMPLEXITÉ EN TEMPS ET EN ESPACE

2.1 Définitions

Le modèle de machine de Turing précis qu'on va utiliser est le suivant : On considère des machines de Turing qui ont $k + 2$ rubans, tous semi-infinis (infinis juste à droite) :

- Un ruban où on place l'entrée, et qui est en lecture seule : la tête de lecture ne peut pas modifier l'entrée et ne peut pas se déplacer en dehors de l'entrée et des blancs qui l'entoure ;
- Un ruban où on placera la sortie. Ce ruban est en écriture seule : la tête de lecture ne peut pas réécrire là où elle a déjà écrit, et ne peut pas lire ce qu'elle a écrit ;
- k rubans "normaux".

Lorsqu'on s'intéresse à des machines de Turing qui acceptent des langages, on peut le faire de deux façons :

- Supposer que la machine écrit 1 sur le ruban de sortie si le mot est accepté, et 0 sinon ;
- Supposer que la machine a en fait deux états finaux différents, un qui correspond à un état d'acceptation, l'autre à un état de refus.

Les deux modes de reconnaissance ne changent pas grand chose, et on ne s'attardera pas là dessus.

Dans la suite, on s'intéresse à la complexité des problèmes : Un problème, en complexité, est juste la donnée de l'ensemble des solutions de ce problème, c'est à dire un langage (ensemble de mots, de graphes, etc). Par exemple, on peut considérer le langage L_1 des graphes G tels que G est 3-coloriable.

On dit qu'une machine de Turing reconnaît un langage L (répond au problème) si elle répond "oui" lorsque son entrée est dans le langage, et "non" sinon.

On peut maintenant définir tout ceci

Définition 2.1

Si M est une machine de Turing et w un mot, le temps utilisé par M sur l'entrée w est le nombre de pas de calcul qu'effectue la machine M sur l'entrée w avant qu'elle ne s'arrête. Le temps est dit infini si la machine ne s'arrête pas.

Définition 2.2

Si M est une machine de Turing et w un mot, l'espace utilisé par M sur l'entrée w est le nombre de cases des différents rubans de travail sur laquelle la machine est passée. Il est infini si la machine utilise un nombre infini de cases (et donc ne s'arrête pas).

Définition 2.3

On dit que la machine de Turing M fonctionne en temps f si pour toute entrée w de taille n , le temps utilisé par M est au plus $f(n)$.

Définition 2.4

On dit que la machine de Turing M fonctionne en espace f si pour toute entrée w de taille n , l'espace utilisé par M est au plus $f(n)$.

Définition 2.5 (DTIME(f))

DTIME(f) est l'ensemble des langages qu'on peut calculer par une machine de Turing fonctionnant en temps f : $L \in \mathbf{DTIME}(f)$ s'il existe une machine de Turing M fonctionnant en temps f et qui reconnaît L .

Définition 2.6 (DSPACE(f))

DSPACE(f) est l'ensemble des langages qu'on peut calculer par une machine de Turing fonctionnant en espace f .

A priori, on pourrait penser que l'espace nécessaire pour accepter un langage peut varier suivant la machine de Turing utilisée, et par exemple son nombre de rubans. Ce n'est pas le cas :

Proposition 2.1

Si L est accepté en espace f et en temps g par une machine de Turing à k rubans de travail, alors L est accepté en espace f et en temps g^2 par une machine de Turing à un seul ruban de travail

L'idée de la preuve est la suivante : On regroupe les k rubans en un seul ruban. Le problème est évidemment que l'on n'a le droit maintenant qu'à une seule tête (la difficulté est dans la réduction du nombre de têtes, pas du nombre de rubans). On représente donc sur notre ruban également la position des têtes. On voit un exemple sur la figure 2.1. Le problème pour simuler la machine est que maintenant il faut, avant de pouvoir faire une étape de la machine d'origine, aller regarder sous chaque tête de lecture ce qu'il y a d'écrit afin de savoir quelle transition effectuer. Cette opération est coûteuse et on peut borner son temps par g , d'où le coefficient g^2 (g étapes, on prend g pas de calcul par étape)

On peut en fait faire mieux :

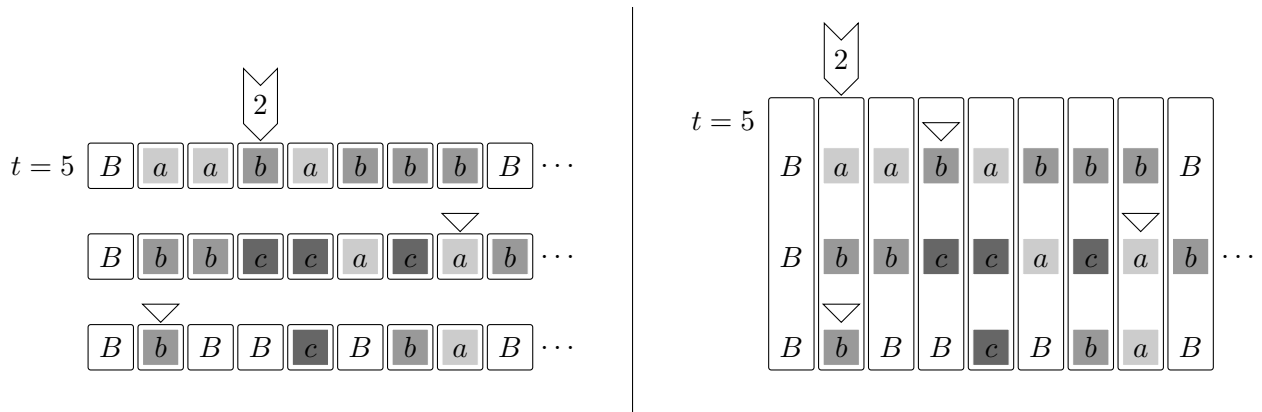


FIGURE 2.1 – 3 rubans en un ruban. On voit d’abord la configuration sur 3 rubans, et ensuite comment on la représente sur un seul ruban.

Proposition 2.2 (Hennie-Stearns)

Si L est accepté en espace f et en temps g par une machine de Turing à k rubans de travail, alors L est accepté en espace f et en temps $g \log g$ par une machine de Turing à deux rubans de travail.

mais la preuve est beaucoup plus compliquée.

2.2 Théorèmes de hiérarchie - Premières relations

Théorème 2.3

- $\text{DSPACE}(f) = \text{DSPACE}(O(f))$
- $\text{DTIME}(f) = \text{DTIME}(O(f))$ si $n = o(f(n))$.

On ne prouvera pas ce théorème. Il signifie que si on a un algorithme en temps $2n^3$ pour résoudre un problème, alors on a également un algorithme en temps n^3 .

Proposition 2.4

- Si $f \leq g$ (c’est à dire $\forall n, f(n) \leq g(n)$)
- $\text{DSPACE}(f) \subseteq \text{DSPACE}(g)$
 - $\text{DTIME}(f) \subseteq \text{DTIME}(g)$

Preuve : Immédiat. ■

Temps et espace sont liés :

Proposition 2.5

$$\mathbf{DTIME}(f) \subseteq \mathbf{DSPACE}(f)$$

Preuve : Pour aller jusqu'à la n ème cellule d'un ruban de travail, la machine de Turing doit faire au moins n étapes ; En conséquence, une machine de Turing qui fonctionne en temps f va visiter au maximum f cellules. ■

Proposition 2.6

$$\mathbf{DSPACE}(f) \subseteq \mathbf{DTIME}(2^{O(f)}) \text{ si } f(n) \geq \log n.$$

Preuve : Soit $L \in \mathbf{DSPACE}(f)$. L est donc reconnu par une machine de Turing qui fonctionne en espace f . On peut supposer grâce à la proposition 2.1 que cette machine n'a qu'un seul ruban de travail. Soit w un mot de taille n . Lorsque la machine de Turing fonctionne sur l'entrée w , elle n'utilise qu'un espace $f(n)$.

Comptons le nombre de configurations dans laquelle la machine peut se trouver. Une configuration est la donnée :

- De l'état de la machine, il y en a $|Q|$ possibles ;
- De la position de la tête sur le ruban d'entrée, il y en a $n + 2$ possibles (sur chacune des lettres de w , et sur les deux bords) ;
- De la position de la tête sur le ruban de travail : il y a $f(n)$ positions possibles, puisque la tête ne visite que $f(n)$ cellules ;
- De ce qui est écrit sur le ruban de travail. Etant donné que seulement $f(n)$ cellules ont été visitées, et donc éventuellement modifiées, on a $|\Gamma|^{f(n)}$ possibilités différentes.

En conclusion, le nombre de configurations différentes dans laquelle la machine peut se trouver est $|Q| (n + 2) f(n) |\Gamma|^{f(n)}$.

Sur l'entrée w , la machine ne peut pas se trouver au cours du calcul deux fois dans la même configuration, sans quoi la machine va boucler (ce qui n'est pas le cas, puisqu'elle va répondre "oui" ou "non"). En conséquence, le nombre de pas de calcul de la machine est borné par le nombre de configurations, et la machine fonctionne en temps $|Q| (n + 2) f(n) |\Gamma|^{f(n)} = 2^{O(f(n))}$. En conséquence $L \in \mathbf{DTIME}(2^{O(f(n))})$. ■

2.3 Théorèmes de hiérarchie - Diagonalisation

On va ici utiliser une technique de diagonalisation pour montrer que différentes classes de complexité sont différentes.

On va commencer par admettre le résultat suivant, qui explique intuitivement qu'il existe une machine de Turing qui peut simuler toutes les autres.

Fait 2.7

Il existe une machine de Turing U et une énumération des machines de Turing M_i à deux rubans tel que

- Chaque machine apparaît infiniment souvent dans l'énumération ;
- U sur l'entrée i et x simule la machine M_i sur l'entrée x de sorte que
 - U effectue chaque pas de calcul de M_i en k pas de calcul, où k est une constante ne dépendant que de la machine (et pas de sa place dans l'énumération)
 - U utilise un espace au plus h fois celui qu'utilise M_i , où h est une constante ne dépendant que de la machine (et pas de sa place dans l'énumération)

La méthode pour prouver les différents résultats va alors toujours être la même : Supposons donné une énumération M_i des machines de Turing fonctionnant en temps f . Construisons une machine N de la façon suivante : $N(i) = 1$ si $M_i(i) = 0$ et $N(i) = 0$ sinon.

Par construction, le langage L reconnu par la machine N est différent de tous les langages reconnus par les M_i . Si on arrive à prouver que N fonctionne en temps g on aura donc montré $\mathbf{DTIME}(f) \neq \mathbf{DTIME}(g)$ (puisque $L \in \mathbf{DTIME}(g)$ mais $L \notin \mathbf{DTIME}(f)$).

2.3.1 Espace

On va pouvoir maintenant prouver :

Proposition 2.8

$\mathbf{DSPACE}(n) \neq \mathbf{DSPACE}(n^2)$

Preuve : On va construire une machine de Turing qui reconnaît un langage L qui est dans $\mathbf{DSPACE}(n^2)$ mais pas dans $\mathbf{DSPACE}(n)$. Voici le programme de cette machine de Turing, qu'on va appeler T :

- Sur l'entrée w de longueur n , noircir n^2 cases,
- Exécuter la machine de Turing U sur l'entrée (w, w) (c'est à dire : simuler l'exécution de la machine numéro w sur l'entrée w)
 - Si la machine U veut dépasser une des cases noircies, s'arrêter en répondant "non" ;
 - Si la machine U s'arrête en répondant "non", répondre "oui"
 - Si la machine U s'arrête en répondant "oui", répondre "non"

Soit L le langage reconnu par T .

On vérifie aisément que T fonctionne en espace n^2 , puisqu'on a limité le calcul artificiellement à n^2 cellules. Donc $L \in \mathbf{DSPACE}(n^2)$.

Montrons maintenant que $L \notin \mathbf{DSPACE}(n)$. Supposons $L \in \mathbf{DSPACE}(n)$. Il existe donc une machine de Turing, qu'on va appeler N qui reconnaît L et qui fonctionne en espace n et on peut supposer que N a deux rubans de travail.

Lorsqu'on simule N par la machine U on utilise un espace h fois plus grand pour une certaine constante h . Comme N apparaît infiniment souvent dans l'énumération M_i , elle apparaît avec un numéro i qu'on peut supposer grand, c'est à dire : $hi < i^2$.

Regardons ce qui se passe lorsqu'on exécute T sur l'entrée i . Comme U n'utilise que h fois plus d'espace que N , qui n'en utilise que i , U n'en utilise que $hi \leq i^2$, de sorte que lorsqu'on exécute T , la machine U ne va dépasser aucune des cases noircies, on est donc dans un des autres cas. Si N répond "oui" sur i , alors on voit que T va répondre "non". De même, si N répond "non", T va répondre "oui". En conséquence, N et T ne répondent pas la même chose sur l'entrée i , elles ne peuvent donc pas reconnaître le même langage. Donc L n'est pas le langage reconnu par N , contradiction.

Et $L \notin \mathbf{DSPACE}(n)$. ■

Remarque : Il y a un petit problème dans cette démonstration, puisqu'il est possible que sur certaines entrées la machine T ne termine pas. Ceci est facile à corriger, mais n'est pas expliqué ici pour faciliter la compréhension.

Cette démonstration s'adapte aisément à d'autres fonctions que n et n^2 . Cependant, un point est essentiel : On utilise le fait qu'on est capable, sur une entrée de taille n , de noircir n^2 cases. La définition suivante s'en suit :

Définition 2.7

Une fonction f est dite constructible en espace s'il existe une machine de Turing qui sur toute entrée de taille n visite exactement $f(n)$ cellules.

La plupart des fonctions naturelles ($f(n) = n^2$, $f(n) = 2^n$, $f(n) = \log n$) sont constructibles en espace. Certaines trop petites ($f(n) = \log \log \log n$ par exemple) ne le sont pas.

On peut maintenant refaire la preuve :

Théorème 2.9

Soit f et g deux fonctions telles que g est constructible en espace, $g(n) \geq \log n$, et $f = o(g)$. Alors $\mathbf{DSPACE}(f) \neq \mathbf{DSPACE}(g)$

La démonstration a été recopiée :

Preuve : Considérons la machine de Turing suivante, qu'on va appeler T :

- Sur l'entrée w de longueur n , noircir $g(n)$ cases,
- Exécuter la machine de Turing U sur l'entrée (w, w) .
 - o Si la machine U veut dépasser une des cases noircies, s'arrêter en répondant "non";
 - o Si la machine U s'arrête en répondant "non", répondre "oui"
 - o Si la machine U s'arrête en répondant "oui", répondre "non"

Soit L le langage reconnu par T .

On vérifie aisément que T fonctionne en espace $g(n)$, puisqu'on a limité le calcul artificiellement à $g(n)$ cellules et qu'on est capable de noircir exactement $g(n)$ cellules par hypothèse de constructibilité en espace. Donc $L \in \mathbf{DSPACE}(g(n))$.

Montrons maintenant que $L \notin \mathbf{DSPACE}(f(n))$. Supposons $L \in \mathbf{DSPACE}(f(n))$. Il existe donc une machine de Turing, qu'on va appeler N qui reconnaît L et qui fonctionne en espace $f(n)$ et on peut supposer que N a deux rubans de travail.

Lorsqu'on simule N par la machine U on utilise un espace h fois plus grand pour une certaine constante h . Comme N apparaît infiniment souvent dans l'énumération M_i , elle apparaît avec un numéro i qu'on peut supposer grand, c'est à dire qui vérifie : $hf(i) < g(i)$ (comme $f = o(g)$ tout i suffisamment grand va vérifier cette condition)

Regardons ce qui se passe lorsqu'on exécute T sur l'entrée i . Comme U n'utilise que h fois plus d'espace que N , qui n'en utilise que $f(i)$, U n'en utilise que $hf(i) \leq g(i)$, de sorte que lorsqu'on exécute T , la machine U ne va dépasser aucune des cases noircies, on est donc dans un des autres cas. Si N répond "oui" sur i , alors on voit que T va répondre "non". De même, si N répond "non", T va répondre "oui". En conséquence, N et T ne répondent pas la même chose sur l'entrée i , elles ne peuvent donc pas reconnaître le même langage. Donc L n'est pas le langage reconnu par N , contradiction.

Et $L \notin \mathbf{DSPACE}(g(n))$. ■

Remarque : La condition $g(n) \geq \log n$ n'est pas utilisée dans la démonstration mais est nécessaire lorsqu'on fait une démonstration soigneusement : Elle permet de s'arranger pour que la machine de Turing T termine toujours.

2.3.2 Temps

Le théorème pour le temps est relativement identique

Proposition 2.10

$$\mathbf{DTIME}(n) \neq \mathbf{DTIME}(n^2)$$

Preuve : On va construire une machine de Turing qui reconnaît un langage L qui est dans $\mathbf{DTIME}(n^2)$ mais pas dans $\mathbf{DTIME}(n)$. Voici le programme de cette machine de Turing, qu'on va appeler T :

- Sur l'entrée w de longueur n , Exécuter en parallèle les deux processus suivants, et répondre comme le premier qui s'arrête :
 - Faire n^2 pas de calcul puis s'arrêter
 - Exécuter la machine de Turing U sur l'entrée w, w .
 - Si la machine U s'arrête en répondant "non", répondre "oui"
 - Si la machine U s'arrête en répondant "oui", répondre "non"

Soit L le langage reconnu par T .

On vérifie aisément que T fonctionne en temps n^2 , puisqu'un des deux processus lancés en parallèle est en temps n^2 , donc on répondra au pire en temps n^2 . Donc $L \in \mathbf{DTIME}(n^2)$.

Montrons maintenant que $L \notin \mathbf{DTIME}(n)$. Supposons $L \in \mathbf{DTIME}(n)$. Il existe donc une machine de Turing qui reconnaît L en temps n . Par la proposition 2.2, il existe une machine de Turing N à deux rubans de travail qui reconnaît L et qui fonctionne en temps $n \log n$.

Lorsqu'on simule N par la machine U on simule un pas en k pas, pour une certaine constante k . Comme N apparaît infiniment souvent dans l'énumération M_i , elle apparaît avec un numéro i qu'on peut supposer grand, c'est à dire : $ki \log i < i^2$.

Regardons ce qui se passe lorsqu'on exécute T sur l'entrée i . Comme U n'utilise que k fois plus de temps que N , qui n'en utilise que $i \log i$, U n'en utilise que $ki \log i < i^2$, de sorte que le processus qui va répondre en premier est le deuxième processus, celui qui calcule U . Si N répond "oui" sur i , alors on voit que T va répondre "non". De même, si N répond "non", T va répondre "oui". En conséquence, N et T ne répondent pas la même chose sur l'entrée i , elles ne peuvent donc pas reconnaître le même langage. Donc L n'est pas le langage reconnu par N , contradiction.

Et $L \notin \mathbf{DTIME}(n)$. ■

Cette démonstration s'adapte aisément à d'autres fonctions que n et n^2 . La encore, un point est essentiel : On utilise le fait qu'on est capable, sur une entrée de taille n , de s'arrêter après exactement n^2 pas de calcul. La définition suivante s'en suit :

Définition 2.8

Une fonction f est dite constructible en temps s'il existe une machine de Turing qui sur toute entrée de taille n s'arrête après exactement $f(n)$ pas de calcul.

La plupart des fonctions naturelles ($f(n) = n^2$, $f(n) = 2^n$) sont constructibles en temps. Certaines trop petites ($f(n) < n$) ne le sont pas.

On peut maintenant refaire la preuve :

Proposition 2.11

Soit f et g deux fonctions telles que $f \log f = o(g)$ et g constructible en temps. Alors $\text{DTIME}(f) \neq \text{DTIME}(g)$

Remarque : Attention l'hypothèse est $f \log f = o(g)$ et non plus $f = o(g)$.

Preuve : Considérons la machine de Turing T suivante :

- Sur l'entrée w de longueur n , Exécuter en parallèle les deux processus suivants, et répondre comme le premier qui s'arrête :
 - Faire $g(n)$ pas de calcul puis s'arrêter
 - Exécuter la machine de Turing U sur l'entrée w , w .
 - Si la machine U s'arrête en répondant "non", répondre "oui"
 - Si la machine U s'arrête en répondant "oui", répondre "non"

Soit L le langage reconnu par T .

On vérifie aisément que T fonctionne en temps $g(n)$, puisqu'un des deux processus lancés en parallèle est en temps exactement $g(n)$ puisque g est constructible en temps, donc on répondra au pire en temps $g(n)$. Donc $L \in \text{DTIME}(g(n))$.

Montrons maintenant que $L \notin \text{DTIME}(f(n))$. Supposons $L \in \text{DTIME}(f(n))$. Il existe donc une machine de Turing qui reconnaît L en temps $f(n)$. Par la proposition 2.2, il existe une machine de Turing N à deux rubans de travail qui reconnaît L et qui fonctionne en temps $f(n) \log f(n)$.

Lorsqu'on simule N par la machine U on simule un pas en k pas, pour une certaine constante k . Comme N apparaît infiniment souvent dans l'énumération M_i , elle apparaît avec un numéro i qu'on peut supposer grand, c'est à dire : $k f(i) \log f(i) < g(i)$ (Tout i suffisamment grand va vérifier cette condition)

Regardons ce qui se passe lorsqu'on exécute T sur l'entrée i . Comme U n'utilise que k fois plus de temps que N , qui n'en utilise que $f(i) \log f(i)$, U n'en utilise que $k f(i) \log f(i) < g(i)$, de sorte que le processus qui va répondre en premier est le deuxième processus, celui qui calcule U . Si N répond "oui" sur i , alors on voit que T va répondre "non". De même, si N répond "non", T va répondre "oui". En conséquence, N et T ne répondent pas la même chose sur l'entrée i , elles ne peuvent donc pas reconnaître le même langage. Donc L n'est pas le langage reconnu par N , contradiction.

Et $L \notin \text{DTIME}(f(n))$. ■

NON DÉTERMINISME

3.1 Définitions

On considère ici des machines de Turing non déterministes, ce qui signifie que la fonction δ est maintenant non déterministe, c'est à dire $\delta : Q \times \Sigma \mapsto P(Q \times \Sigma \times \{\leftarrow, \rightarrow\})$ pour la version un ruban, et les analogues pour les versions plus compliquées.

$(q, w) \vdash (q', w')$ signifie toujours la même chose, mais maintenant il peut y avoir plusieurs (q', w') qu'on peut atteindre à partir de (q, w) .

Définition 3.1

Un mot w est accepté par une machine de Turing non-déterministe M si à partir de la configuration initiale correspondant à w , on peut atteindre une configuration finale acceptante.

Considérons l'exemple de la machine de Turing non-déterministe suivante, que l'on donne comme un programme

- Sur l'entrée w de longueur n , choisir un entier i entre 1 et n .
- Accepter si $w_i = 'a'$.

Cette machine de Turing non déterministe accepte les mots qui contiennent au moins un 'a'.

- Sur l'entrée w de longueur n , choisir un entier i entre 0 et 1 ;
- Si $i = 0$, s'arrêter, sinon chercher de nouveau un entier entre 0 et 1 et recommencer

Cette machine de Turing non déterministe accepte tous les mots, mais le temps qu'elle met avant de répondre est arbitraire, et la machine peut même ne pas répondre !

L'acceptation pour une machine de Turing non déterministe est plus compliquée à décrire que pour une machine normale.

Lorsqu'une machine de Turing fonctionne sur un mot w , elle choisit à chaque étape un des choix possibles, et finit (ou non) par s'arrêter en disant "oui" ou "non". Le fait que la machine dise "non" ne signifie pas que le mot w n'est pas dans le langage L , mais plutôt que la machine n'a pas réussi à le montrer ; peut-être qu'en faisant d'autres choix, elle aurait répondu "oui".

On dit donc qu'une machine de Turing M accepte un mot w s'il existe un chemin acceptant, c'est à dire si la machine peut, si elle fait des choix corrects, répondre "oui". Il est refusé s'il n'existe pas de chemin acceptant c'est à dire si tous les chemins refusent c'est à dire si quels que soient les choix que fait la machine, elle va répondre "non".

◆ Exemple

Considérons la machine de Turing suivante :

- Sur l'entrée w de longueur n , choisir i entre 1 et n .
- Accepter si $w_i \neq w_{n-i}$, refuser sinon.

Cette machine de Turing reconnaît les non-palindromes : Si w n'est pas un palindrome, il existe i tel que $w_i \neq w_{n-i}$. Si la machine choisit correctement ce i , elle acceptera w . Si w est un palindrome, quel que soit le choix de la machine, elle va refuser le mot.

◆ **Exemple**

Considérons la machine de Turing suivant, qui prend en entrée un graphe G

- La machine choisit pour tout sommet sa couleur parmi rouge, vert et bleu
- La machine accepte alors si le graphe est colorié de sorte que deux sommets reliés par une arête aient des couleurs différentes.

Cette machine reconnaît les graphes 3-coloriables : Si le graphe est 3-coloriable et que la machine choisit correctement sa coloration, elle va accepter. Si le graphe n'est pas 3-coloriable, quel que soit la coloration qu'elle choisit, elle va refuser.

Définition 3.2

Si M est une machine de Turing non déterministe et w un mot, le temps (resp. espace) utilisé par M sur l'entrée w est le maximum du temps (resp. espace) utilisé par la machine M sur tous les chemins non-déterministes.

Par exemple, la première machine est en temps $O(n)$ (le temps de se placer sur la i ème cellule) et en espace constant. La deuxième est en espace constant et en temps ∞ (puisque la machine peut ne pas s'arrêter).

Définition 3.3 (NTIME(f))

NTIME(f) est l'ensemble des langages calculés par une machine de Turing non déterministe fonctionnant en temps f .

Définition 3.4 (NSPACE(f))

NSPACE(f) est l'ensemble des langages calculés par une machine de Turing non déterministe fonctionnant en espace f .

3.2 Premières propriétés

Proposition 3.1

- DTIME(f) \subseteq NTIME(f)
 - DSPACE(f) \subseteq NSPACE(f)
- Si $f \leq g$, alors
- NTIME(f) \subseteq NTIME(g)
 - NSPACE(f) \subseteq NSPACE(g)

Preuve : Immédiat



Proposition 3.2

Soit L accepté par une machine de Turing non déterministe M . Alors il existe une machine de Turing déterministe M' qui accepte un langage L' telle que $x \in L \iff \exists y, (x, y) \in L'$

Preuve : Pour simplifier, on va supposer que la machine de Turing M ne fait que des choix binaires.

M' fait comme M , mais à chaque fois qu'elle doit faire un choix, disons que c'est la i ème fois, M' demande à y ce qu'elle doit faire : Si $y_i = 0$ elle prend le premier choix, si $y_i = 1$ elle prend le deuxième. ■

Si M est en temps f , on voit qu'on peut choisir $|y| \leq f(|x|)$.

La plupart des algorithmes non-déterministes qu'on construit utilise cette caractérisation : L'algorithme choisit d'abord, grâce au choix non déterministe, un y pour l'aider à répondre au problème, puis "vérifie" (déterministiquement) que ce y permet de résoudre le problème

◆ **Exemple**

Pour le problème de 3-coloriabilité, y correspond au choix, pour chaque sommet, de sa couleur. On vérifie ensuite que la coloration choisie est convenable.

◆ **Exemple**

Pour le problème de savoir si un nombre n n'est pas premier, y correspond au choix d'un nombre inférieur à n . On vérifie ensuite que y divise n .

3.3 Liens avec les classes déterministes**Proposition 3.3**

$\text{NTIME}(f) \subseteq \text{DSPACE}(f)$.

Preuve : Soit $L \in \text{NTIME}(f)$, L est accepté par une machine de Turing M non déterministe.

Considérons alors la machine T qui sur l'entrée ω exécute tous les calculs possibles de la machine M , l'un après l'autre. Pour faire cela, elle doit garder en mémoire les choix qu'elle a déjà fait, pour qu'elle puisse en faire des autres ; il lui faut ainsi un espace f , en plus de l'espace utilisé pour le calcul, qui est aussi de l'ordre de f . ■

Corollaire 3.4

$\text{NTIME}(f) \subseteq \text{DTIME}(2^{O(f)})$

Théorème 3.5 (Savitch)

Si $f(n) \geq \log n$ est constructible en espace, alors $\mathbf{NSPACE}(f) \subseteq \mathbf{DSPACE}(f^2)$

Preuve : Soit $L \in \mathbf{NSPACE}(f)$, L est accepté par une machine de Turing M en espace f . On peut supposer que lorsque L accepte le mot, elle efface tout son ruban de travail, et remet les deux têtes au début, ceci afin qu'il existe une et une seule configuration acceptante.

On sait comme dans le cas déterministe qu'on peut supposer que tous les calculs vont mettre un temps inférieur à $2^{O(f)}$ (sans quoi on pourrait entrer dans une boucle). Soit $g = 2^{cf}$ le nombre de configurations différentes possibles.

On considère alors le programme suivant, qui prend en entrée deux configurations et qui dit si on peut aller de la première à la deuxième en moins de i étapes : Sur l'entrée c_1 et c_2 et i

- Tester récursivement pour toutes les configurations c_3 si on peut aller de c_1 à c_3 en $\lfloor i/2 \rfloor$ étapes, puis de c_3 à c_2 en $i - \lfloor i/2 \rfloor$ étapes
- Si $i = 0$ ou $i = 1$ répondre suivant la table δ .

On appelle cette machine à partir de la configuration initiale, de la configuration finale et de $i = 2^{cf}$: Elle répond donc oui si et seulement si on peut aller de la configuration initiale à la configuration finale, donc si et seulement si le mot peut être accepté.

Chaque étape de la récursion nécessite un espace de l'ordre de $O(f)$ (pour retenir c_1, c_2, c_3, i , et si on est en train de tester de c_1 à c_3 ou de c_3 à c_1) la profondeur de récursion est $\log g = cf$, donc on peut exécuter cette machine en espace $O(f^2)$. Donc $L \in \mathbf{DSPACE}(f^2)$. ■

Proposition 3.6

$\mathbf{NSPACE}(f) \subseteq \mathbf{DTIME}(2^{O(f)})$.

Preuve : On reprend la même technique que dans la preuve précédente.

Construisons le graphe de l'ensemble des configurations où on relie une configuration à une autre si la machine de Turing peut passer de l'un à l'autre en une étape.

Un mot w est alors reconnu s'il existe un chemin de la configuration initiale à la configuration finale.

Le graphe contient $n = 2^{cf}$ sommets, il faut (approximativement) un temps n^2 pour le construire et un temps n^2 pour savoir s'il existe un chemin de la configuration initiale à la configuration finale.

Ainsi $L \in \mathbf{DTIME}(2^{O(f)})$. ■

Note : Cette proposition n'est pas une conséquence de la précédente (Pourquoi ?)

CLASSES DE COMPLEXITÉ

Définition 4.1 (P et NP)

- $\mathbf{P} = \cup_i \mathbf{DTIME}(n^i)$
- $\mathbf{E} = \cup_i \mathbf{DTIME}(2^{in})$
- $\mathbf{EXPTIME} = \cup_i \mathbf{DTIME}(2^{n^i})$
- $\mathbf{LOGSPACE} = \mathbf{DSPACE}(\log n)$
- $\mathbf{PSPACE} = \cup_i \mathbf{DSPACE}(n^i)$
- $\mathbf{NP} = \cup_i \mathbf{NTIME}(n^i)$
- $\mathbf{NLOGSPACE} = \mathbf{NSPACE}(\log n)$
- $\mathbf{NPSPACE} = \cup_i \mathbf{NSPACE}(n^i)$

Définition 4.2

Si C est une classe de complexité, on note $\text{co}C$ l'ensemble des langages L dont le complémentaire est dans C .

Proposition 4.1

- $\mathbf{P} \subseteq \mathbf{E} \subseteq \mathbf{EXPTIME}$
- $\mathbf{LOGSPACE} \subseteq \mathbf{PSPACE}$
- $\mathbf{P} \subseteq \mathbf{NP}$
- $\mathbf{LOGSPACE} \subseteq \mathbf{NLOGSPACE}$
- $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$

Preuve : Immédiat. ■

Proposition 4.2

- $\mathbf{LOGSPACE} \subseteq \mathbf{NLOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$

Preuve : Il faut uniquement prouver $\mathbf{NLOGSPACE} \subseteq \mathbf{P}$ et $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

$\mathbf{NLOGSPACE} = \mathbf{NSPACE}(\log n) \subseteq \mathbf{DTIME}(2^{O(\log n)}) \subseteq \mathbf{P}$ par la proposition 3.6.

$\text{NP} = \cup_i \text{NTIME}(n^i) \subseteq \cup_i \text{DSPACE}(n^i)$ par la proposition 3.3. ■

Proposition 4.3

┆ $\text{PSPACE} = \text{NPSPACE}$

Preuve : Il suffit de prouver $\text{NPSPACE} \subseteq \text{PSPACE}$.

$\text{NPSPACE} = \cup_i \text{NSPACE}(n^i) \subseteq \cup_i \text{DSPACE}(n^{2i}) \subseteq \text{PSPACE}$ par le théorème de Savitch (3.5). ■

Proposition 4.4

┆ $\text{LOGSPACE} \neq \text{PSPACE}$

Preuve : On reprend la preuve de 2.9 avec $g(n) = n$, et on voit qu'il existe un langage calculable en espace n et qui n'est pas calculable en espace $f(n)$ pour tout $f(n) = o(n)$. Ce langage vérifie donc $L \in \text{PSPACE}$ et $L \notin \text{LOGSPACE}$. ■

Corollaire 4.5

┆ Soit $\text{LOGSPACE} \neq \text{P}$, soit $\text{P} \neq \text{PSPACE}$.

Preuve : On a $\text{LOGSPACE} \subseteq \text{P} \subseteq \text{PSPACE}$ et $\text{LOGSPACE} \neq \text{PSPACE}$. ■

Théorème 4.6

┆ Soit $L \in \text{NP}$.

┆ Alors il existe $C \in \text{P}$ et un polynôme p tel que $x \in L \iff \exists y, |y| \leq p(|x|), (x, y) \in C$.

Preuve : y est la liste des choix que fait l'algorithme non déterministe. ■

QUELQUES CONSÉQUENCES DE $P = NP$

5.1 Factorisation

Factoriser est le problème, connaissant un nombre n , de trouver un de ses diviseurs. Ce problème est fondamental en cryptographie ; en particulier, le protocole RSA utilise le fait qu’il n’existe pas, à l’heure actuelle, d’algorithme polynomial pour factoriser.

La théorie des classes de complexité ne s’applique pas directement au problème de la factorisation : En effet, factoriser n’est pas un problème de décision (un problème où il faut juste répondre “oui” ou “non”). Si on veut le regarder, il faut considérer le problème suivant :

$$L_{fac} = \{(n, k) \mid n \text{ a un diviseur strictement plus petit que } k\}$$

Par exemple $(29874949, 15) \in L_{fac}$ puisque 29874949 est divisible par 13, qui est plus petit que 15, mais en revanche $(29874949, 11) \notin L_{fac}$.

Pour analyser ce problème, nous allons avoir besoin du résultat suivant. Soit L_p le langage

$$L_p = \{n \mid n \text{ est premier}\}$$

Théorème 5.1 (Agrawal, Kayal, Saxena (2002))

$$L_p \in P$$

Autrement dit, on peut tester en temps polynomial si un nombre est premier.

Proposition 5.2

$$L_{fac} \in NP$$

Preuve : Clair ■

Proposition 5.3

$$L_{fac} \in coNP$$

Preuve : Il faut donc donner un algorithme dans **NP** qui décide, étant donné n et k , si n n’a pas de diviseur plus petit que k . Pour cela, il suffit de considérer l’algorithme suivant, sur l’entrée n et k

- Choisir des entiers a_i inférieurs à n
- Vérifier qu’ils sont tous premiers
- Vérifier que le produit des a_i vaut n
- Vérifier qu’aucun des a_i n’est inférieur à k

Autrement dit, deviner la décomposition en facteurs premiers de n , et vérifier qu'aucun des facteurs premiers n'est inférieur à k .

Cet algorithme est bien dans NP . Pour cela, il faut borner le nombre de a_i à prendre à la première étape (si on en prend trop, l'algorithme ne sera plus polynomial). Cependant, le nombre de facteurs premiers d'un entier n est borné par $\log n$ (puisque tous ses facteurs premiers sont supérieurs à 2, il ne peut en avoir plus de $\log n$). Lors de la première étape, il suffit donc de choisir au plus $\log n$ entiers a_i , c'est à dire un nombre linéaire en la taille de l'entrée (qui est $\log n + \log k$). La première étape de vérification utilise le théorème ci-dessus. ■

Proposition 5.4

Si $P = NP$, il existe un algorithme polynomial pour factoriser.

Preuve : $L_{fac} \in NP$, donc sous l'hypothèse, il existe un algorithme polynomial pour décider, étant donné, n et k si n a un diviseur plus petit que k .

On va utiliser cet algorithme pour trouver le plus petit diviseur par dichotomie.

Voici l'algorithme :

- Si n est premier, répondre "ce nombre est premier"
- $a := 2, b := n$
- Tant que $a \neq b$
 - Poser $c := (a + b)/2$
 - Demander si n a un diviseur plus petit que c
 - Si c'est le cas, poser $b := c$, sinon poser $a := c$

Cet algorithme effectue un nombre d'étapes de l'ordre de $\log n$, donc polynomial en la longueur de l'entrée, et chaque étape s'effectue, par hypothèse, en temps polynomial. ■

5.2 Certificats polynomiaux

Rappelons la propriété :

Théorème 4.6

Soit $L \in NP$.

Alors il existe $C \in P$ et un polynôme p tel que $x \in L \iff \exists y, |y| \leq p(|x|), (x, y) \in C$.

Soit $L \in NP$, et C, p qui y correspondent. On dit que y est un certificat de l'appartenance de x à L s'il vérifie les conditions de l'énoncé.

◆ Exemple

- Pour L_{fac} un certificat est un diviseur p de n plus petit que k . Le langage C vérifie que c'est bien le cas
- Pour ${}^cL_{fac}$ le certificat est la décomposition en facteurs premiers de n , le langage C vérifie que c'est bien le cas et qu'aucun des facteurs premiers n'est inférieur à k .
- Pour 3-color, l'ensemble des graphes 3-coloriables, un certificat est une bonne coloration du graphe en 3 couleurs, et C vérifie que c'est bien une bonne coloration.

Théorème 5.5

Soit $L \in \mathbf{NP}$, p un polynôme et $C \in \mathbf{P}$ tel que

$$x \in L \iff \exists y, |y| \leq p(|x|), (x, y) \in C$$

Si $\mathbf{P} = \mathbf{NP}$ alors il existe une fonction f calculable en temps polynomial tel que

- Si $x \in L$ alors $f(x)$ est un certificat pour x .
- Sinon $f(x)$ vaut “faux”.

Note : nous n’avons pas encore défini précisément ce qu’on appelle les fonctions calculables en temps polynomial, on verra un peu plus loin.

Preuve : On va procéder par dichotomie. On va supposer que dans la formulation du problème, y est un mot sur l’alphabet $\{a, b\}$. Si u et v sont deux mots, on note $u \sqsubset v$ pour dire que u commence par v .

On introduit le langage

$$L' = \{(x, w) \mid \exists y, y \sqsubset w, |y| \leq p(|x|), (x, y) \in C\}$$

Autrement dit, $(x, w) \in L'$ s’il existe un certificat que $x \in L$ qui commence par w .

Remarquons que $L' \in \mathbf{NP}$ (clair). Donc $L' \in \mathbf{P}$ par hypothèse.

Considérons maintenant le programme suivant, sur l’entrée x

- Si $x \notin L$, répondre faux.
- Soit w le mot vide, de longueur 0.
- Tant que $(x, w) \notin C$ (tant que w n’est pas un certificat)
 - Si $(x, wa) \in L'$ (s’il existe un certificat qui commence par wa), prendre $w := wa$
 - Sinon (il existe un certificat qui commence par wb), prendre $w := wb$.
- Renvoyer w .

Cet algorithme fonctionne en temps polynomial : Le premier test peut se faire en temps polynomial puisque $L \in \mathbf{NP} \subseteq \mathbf{P}$ par hypothèse. Chaque étape de la boucle s’effectue en temps polynomial puisque C et L' sont dans \mathbf{P} et le nombre d’étapes est borné par le polynôme p par hypothèse.

Et il est clair que cet algorithme répond au problème. ■

Il est important de noter que ce n’est pas le fait que $L \in \mathbf{P}$ qui permet d’obtenir l’algorithme, mais bien le fait qu’un langage très proche, L' , y soit. On notera également que la preuve du résultat est très proche de celle de l’algorithme polynomial pour factoriser. Dire “ n a un diviseur plus petit que k ” ou “ n a un diviseur dont les premiers chiffres sont ... ” n’est pas très différent.

5.3 Cryptographie - Fonction One-Way

Définition 5.1

$\mathbf{FTIME}[g]$ est l'ensemble des fonctions f pour lesquelles il existe une machine de Turing calculant f fonctionnant en temps inférieur à g .

$\mathbf{FP} = \cup \mathbf{FTIME}[n^i]$ est l'ensemble des fonctions calculables en temps polynomial.

On définit de même $\mathbf{FLOGSPACE}$ et $\mathbf{FPSPACE}$. A noter, lorsqu'on définit l'espace utilisé par une machine, que sa sortie, écrite dans un ruban spécial et non pas sur le ruban de travail, n'est pas pris en compte.

Les fonctions qu'on définit sont partielles, c'est à dire ne sont pas forcément définies sur toutes les entrées.

La définition suivante n'est pas officielle

Définition 5.2 (Fonction one-way)

Une fonction one-way est une fonction de f de A dans B telle que

- $f \in \mathbf{FP}$ (f est calculable rapidement)
- il n'existe pas de $g \in \mathbf{FP}$ tel que $f(g(x)) = x$ pour x dans l'image de f (il n'existe pas d'inverse calculable)
- f est honnête : $f(x)$ est du même ordre de grandeur que x : c'est à dire il existe un polynôme p tel que $|x| \leq p(|f(x)|)$

Attention, on ne définit pas le comportement de g si on lui donne une entrée qui n'est pas de la forme $f(x)$.

Théorème 5.6

Si $P = NP$, il n'existe pas de fonction one-way. Plus exactement, pour toute fonction $f \in \mathbf{FP}$ honnête, on peut calculer un inverse $g \in \mathbf{FP}$

Preuve : Soit L le langage

$$L = \{x | \exists y, f(y) = x\}$$

Soit C le langage

$$C = \{(x, y) | f(y) = x\}$$

Alors $x \in L \iff \exists y, (x, y) \in C$. De plus $x \in L \iff \exists y, |y| \leq p(|x|), (x, y) \in C$, puisque f est honnête.

$C \in \mathbf{P}$ puisque $f \in \mathbf{FP}$. Trouver y revient à trouver un certificat que $x \in L$, donc l'algorithme précédent permet de résoudre le problème. ■

Proposition 5.7

Soit $f \in \mathbf{FP}$ honnête d'inverse $g \in \mathbf{FP}$. Alors l'image de f , $L_f = \{y | \exists x, f(x) = y\}$ est dans \mathbf{P} .

Preuve : $y \in L_f \iff f(g(y)) = y$ ■

Théorème 5.8

Si toute fonction $f \in \mathbf{FP}$ honnête a un inverse dans \mathbf{FP} , alors $\mathbf{P} = \mathbf{NP}$.

Preuve : Soit $L \in \mathbf{NP}$, p un polynôme et $C \in \mathbf{P}$ tel que

$$x \in L \iff \exists y, |y| \leq p(|x|), (x, y) \in C$$

Considérons la fonction f définie comme suit :

– $f(x, y) = x$ si $(x, y) \in C$ et $|y| \leq p(|x|)$

– $f(x, y)$ n'est pas défini sinon.

f est honnête et $f \in \mathbf{FP}$. Par la proposition précédente, L_f est dans \mathbf{P} . Mais $L_f = L$ ce qui finit la preuve. ■

5.4 Padding

Soit L un langage quelconque et $f : \mathbb{N} \rightarrow \mathbb{N}$ une fonction.

Le langage L_f est défini par

$$L_f = \{w\#1^{f(|w|)} \mid w \in L\}$$

Autrement dit, on ajoute $f(|w|)$ fois le symbole 1 à l'entrée w .

Théorème 5.9

Soit f constructible en temps.

Si $L \in \mathbf{NTIME}[f]$, alors $L_f \in \mathbf{NP}$.

Preuve : La machine de Turing, sur l'entrée x , commence par vérifier que x est de la forme $w\#1^{f(|w|)}$. Comme f est constructible en temps, il lui suffit, pour vérifier que le nombre de 1 est exactement de $f(|w|)$ que d'un temps $f(|w|)$ donc d'un temps linéaire en $|x|$.

Ensuite, il lui faut un temps de l'ordre de $f(|w|)$ pour tester si $w \in L$, donc un temps linéaire en $|x|$.

Au final, la machine de Turing, non déterministe, fonctionne en temps linéaire donc le langage qu'elle reconnaît est dans $\mathbf{NTIME}[n] \subseteq \mathbf{NP}$. ■

Théorème 5.10

Soit f constructible en temps, si $L_f \in \mathbf{P}$ alors $L \in \mathbf{TIME}[O(f^k)]$ pour un certain k

Preuve : $L_f \in \mathbf{P}$, donc $L_f \in \mathbf{TIME}[n^k]$ pour un certain k .

Considérons maintenant l'algorithme qui sur l'entrée w construit $w\#1^{f(|w|)}$ puis lance l'algorithme qui résout L_f en temps n^k .

La construction de l'entrée prend un temps de l'ordre de $f(|w|)$. L'algorithme qui résout L_f mets un temps n^k sur une entrée de taille $n = f(|w|) + |w| + 1$, donc le nouvel algorithme est en temps $f(|w|) + (f(|w|) + |w| + 1)^k = O(f(|w|)^k)$. ■

Corollaire 5.11

- Si $\mathbf{P} = \mathbf{NP}$, alors pour tout f constructible en temps, $\mathbf{NTIME}[f] \subseteq \cup_k \mathbf{TIME}[O(f^k)]$
- Si $\mathbf{P} = \mathbf{NP}$, alors $\mathbf{E} = \mathbf{NE}$

Théorème 5.12

Si $\mathbf{Linspace} \subseteq \mathbf{P}$ alors $\mathbf{P} = \mathbf{PSPACE}$

$\mathbf{Linspace} = \mathbf{SPACE}[O(n)]$ est l'ensemble des langages reconnus en espace linéaire.

Preuve : Soit $L \in \mathbf{PSPACE}$. Donc L est reconnu pour un certain p par une machine de Turing fonctionnant en espace $p(n)$ sur une entrée de longueur n .

Prenons alors le langage L_p .

$$L_p = \{w\#1^{p(|w|)} \mid w \in L\}$$

L_p est accepté en espace linéaire : Il suffit en effet d'abord de tester que l'entrée est bien de la bonne forme $w\#1^{p(|w|)}$ puis d'exécuter l'algorithme en espace $p(|w|)$ donc en espace linéaire en la taille de l'entrée $|w| + p(|w|) + 1$.

Par hypothèse $L_p \in \mathbf{P}$. Mais alors $L \in \mathbf{P}$: Il suffit, sur l'entrée w de construire $w\#1^{p(|w|)}$ puis d'appeler L_p .

Donc pour tout $L \in \mathbf{PSPACE}$, on a $L \in \mathbf{P}$, donc $\mathbf{PSPACE} \subseteq \mathbf{P}$ ■

Corollaire 5.13

$\mathbf{P} \neq \mathbf{Linspace}$

Preuve : Si $\mathbf{P} = \mathbf{Linspace}$, on a $\mathbf{PSPACE} = \mathbf{P}$, donc on aurait $\mathbf{Linspace} = \mathbf{PSPACE}$ ce qui est faux (on sait que $\mathbf{SPACE}[n^2] \neq \mathbf{SPACE}[O(n)]$ d'après le théorème 2.9. ■

COMPLÉTUDE

6.1 Réductions

Définition 6.1

Une fonction f est dite calculable en espace logarithmique s'il existe une machine de Turing qui sur l'entrée x , calcule $f(x)$ en utilisant un espace $O(\log |x|)$.

On ne compte l'espace que sur le ruban de travail, mais pas sur le ruban de sortie. En conséquence, la taille de la sortie peut être polynomiale en $|x|$.

Définition 6.2 (Réduction \leq_m)

Soit A et B deux langages. On dit que A se réduit à B , ce qu'on note $A \leq_m B$ s'il existe une fonction f calculable en espace logarithmique telle que $x \in A \iff f(x) \in B$.

On dira souvent dans ce cas que A est plus "simple" que B .

◆ Exemple

Considérons les deux langages suivants, PARTITION et SUBSETSUM

- PARTITION est l'ensemble des listes d'entiers a_i qu'on peut partager en deux parties de même somme.
- SUBSET est l'ensemble des listes d'entiers a_i et d'entiers K tels qu'on peut atteindre exactement K en sommant certains des a_i

On a alors

- PARTITION \leq_m SUBSETSUM. En effet, PARTITION est plus simple que SUBSETSUM : PARTITION correspond au cas particulier de SUBSETSUM où $K = \frac{a_1 + \dots + a_n}{2}$. Plus formellement, posons la fonction f suivante

$$f : (a_1, \dots, a_n) \mapsto ((a_1, \dots, a_n), \frac{a_1 + \dots + a_n}{2})$$

f vérifie bien

$$(a_1, \dots, a_n) \in PARTITION \iff f(a_1, \dots, a_n) \in SUBSETSUM$$

et se calcule en espace logarithmique de sorte qu'on a bien prouvé PARTITION \leq_m SUBSETSUM.

- SUBSETSUM \leq_m PARTITION. En effet, considérons la transformation suivante : On part de (a_1, \dots, a_n) et de K . Notons $M = a_1 + \dots + a_n$.

Définissons $b_1 = a_1 \dots b_n = a_n$ et $b_{n+1} = 2M, b_{n+2} = M + 2K$. Autrement dit, on rajoute deux nouveaux entiers à la liste des a_i .

Montrons alors

On peut atteindre exactement K en sommant certains des a_i si et seulement si on peut partager les b_i en deux parties de même somme.

Il y a deux implications :

- Supposons qu'on arrive à atteindre K en sommant certains des a_i .
Mettons tous ces a_i d'un côté (enfin, plus exactement les b_i qui correspondent), en leur ajoutant b_{n+1} . On obtient donc un ensemble de somme $K + 2M$. L'autre partie est constitué du reste des a_i , dont la somme vaut $M - K$ et de b_{n+2} qui vaut $M + 2K$ de sorte qu'on obtient un ensemble de taille $2M + K$.

On a donc bien réussi à partager les b_i en deux parties de même somme.

- Supposons avoir partagé les b_i en deux parties de même somme. La somme totale de tous les b_i fait $M + b_{n+1} + b_{n+2} = 4M + 2K$. Donc il faut faire deux parties de somme $2M + K$. On ne peut pas mettre b_{n+1} et b_{n+2} dans la même partie, puisqu'on aurait une partie de somme au moins $3M + 2K > 2M + K$. Donc b_{n+1} et b_{n+2} sont dans des parties différentes. Regardons la partie qui contient b_{n+1} . Comme la somme totale de la partie fait $2M + K$ et que $b_{n+1} = 2M$, on en déduit que la somme totale des autres b_i (qui sont en fait certains des objets a_i du départ) fait K .

On a donc bien réussi à trouver un certain ensemble des a_i dont la somme fait exactement K .

On peut voir que la transformation de $a_1 \dots a_n$ vers $b_1 \dots b_{n+2}$ se fait en espace logarithmique de sorte qu'on a bien démontré SUBSETSUM \leq_m PARTITION.

Proposition 6.1

\leq_m est une relation de préordre partiel :

- $A \leq_m A$
- Si $A \leq_m B$ et $B \leq_m C$, alors $A \leq_m C$.

$A \leq_m A$ est clair, il suffit de prendre $f(x) = x$. Pour la deuxième partie, il faut montrer que si f et g sont calculables en espace logarithmique, alors $f \circ g$ aussi. Ce n'est pas trivial, et c'est admis.

Proposition 6.2

- Si $A \leq_m B$ et $B \in \mathbf{LOGSPACE}$, alors $A \in \mathbf{LOGSPACE}$.
- Si $A \in \mathbf{LOGSPACE}$, alors $A \leq_m B$.

Ces deux propriétés signifient que $\mathbf{LOGSPACE}$ est en bas de l'ordre défini par \leq_m .

- Preuve :**
- Soit f qui témoigne que $A \leq_m B$. Alors le programme suivant montre que $A \in L$: sur l'entrée x calculer $f(x)$ puis demander si $f(x) \in B$.
 - On suppose que B n'est pas trivial, c'est à dire qu'il existe $y \in B$ et $z \notin B$. On considère la fonction f suivante : Sur l'entrée x , chercher si $x \in A$. Si c'est le cas, on pose $f(x) = y$. Si ce n'est pas le cas, on pose $f(x) = z$. f est calculable en espace polynomial, puisqu'il suffit juste de savoir décider si $x \in A$. Comme $x \in A \iff f(x) \in B$, on en déduit $A \leq_m B$. ■

Proposition 6.3

Si \mathcal{C} est la classe **LOGSPACE** ou **NLOGSPACE** ou **P** ou **NP** ou **coNP** ou **PSPACE**, et si $A \leq_m B$ avec $B \in \mathcal{C}$, alors $A \in \mathcal{C}$.

Preuve : On reprend la démonstration ci-dessus. ■

Proposition 6.4

Si $A \leq_m B$, alors $\bar{A} \leq_m \bar{B}$

6.2 Complétude**Définition 6.3 (Problèmes \mathcal{C} -complets)**

Si \mathcal{C} est une classe de complexité, on dit que L est \mathcal{C} -dur si pour tout $A \in \mathcal{C}$, $A \leq_m L$. on dit que L est \mathcal{C} -complet si L est \mathcal{C} -dur et $L \in \mathcal{C}$

Théorème 6.5

Il existe un problème **NLOGSPACE**-complet.

Plus précisément, le problème suivant est **NLOGSPACE**-complet

NOM: GAP (Graph Accessibility Problem)

ENTREE: Un graphe orienté G et deux sommets x et y

QUESTION: Est-ce qu'il existe un chemin de x à y ?

Preuve : *GAP* est dans **NLOGSPACE**, puisqu'il suffit de trouver le chemin de x à y , donc de deviner les sommets $x_1 \dots x_k$ qui permettent de passer de x à y . De plus, on n'a besoin que de stocker un sommet à la fois, d'où l'espace logarithmique.

Soit maintenant A un problème dans **NLOGSPACE** et M la machine de Turing en espace logarithmique qui reconnaît A . On va dire qu'il est en espace $c \log n$. Quitte à modifier un peu la machine, on peut supposer que lorsqu'elle accepte un mot, elle efface tout son ruban de travail, et remet les deux têtes au début, de sorte qu'il n'existe qu'une seule configuration acceptante.

Sur l'entrée w , considérons le graphe G_w dont les sommets sont les configurations de taille $c \log |w|$ et où on relie deux sommets c_1 et c_2 si la machine de Turing passe de la configuration c_1 à la configuration c_2 en une étape. Soit i_w le sommet du graphe correspond à l'état initial de la machine sur l'entrée w et f_w le sommet du graphe correspond à l'état final acceptant. Alors w est accepté si et seulement si il existe un chemin dans le graphe de i_w à f_w .

Notons $f(w) = (G_w, i_w, f_w)$.

Alors $w \in A \iff f(w) \in \text{GAP}$. f se calcule facilement, en espace logarithmique, de sorte qu'on vient de prouver $A \leq_m \text{GAP}$. ■

Théorème 6.6

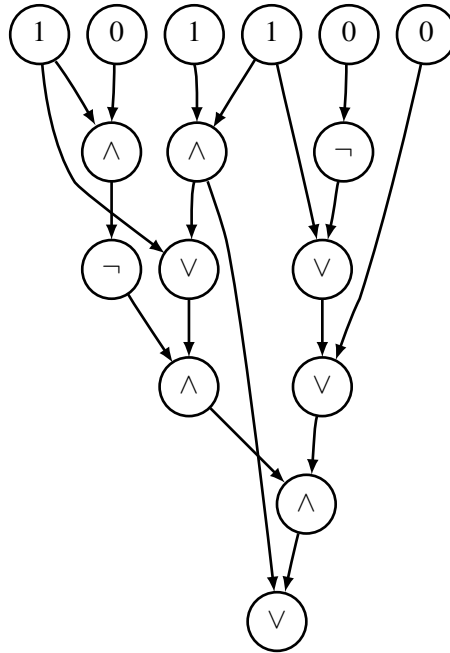
Il existe un problème **P**-complet. Plus précisément, le problème suivant est **P**-complet.

NOM: CIRCUIT

ENTREE: Un circuit

QUESTION: Est-ce que la valeur du circuit est 1 ?

On ne va pas donner la définition précise d'un circuit. Un circuit est un graphe particulier comme celui-ci :



Dans ce cas, la valeur du circuit est 1.

Preuve : Soit A un problème dans P et M la machine de Turing qui en témoigne. M est en temps $p(n)$, et donc en espace $p(n)$. On va supposer pour simplifier que quand un mot est accepté, la tête est revenue au début.

On peut représenter le calcul de la machine de Turing par un tableau de taille $p(n) \times p(n)$: la case numéro (i, j) contient le symbole présent sur la i -ème cellule de la machine de Turing au temps j . Dans le cas où la tête du ruban est au temps j dans la cellule i , on insère également l'état de la machine de Turing dans la case. Chaque case contient donc une information finie, qu'on peut représenter sur un nombre fini de bits, disons k . De plus, on s'aperçoit que la case (i, j) ne dépend que de 3 cases : il existe une fonction g (qui ne dépend pas de i et j) telle que $T[i, j] = g(T[i - 1, j - 1], T[i, j - 1], T[i + 1, j - 1])$.

On peut maintenant représenter chaque case par k variables, et transformer cette fonction g en un circuit qui explique comment passer des $3k$ variables correspondants aux cellules $(i - 1, i, i + 1)$ au temps $j - 1$ aux k variables de la cellule i au temps j .

Si on note t la taille du circuit correspondant à g , on va donc transformer notre tableau en un circuit de taille approximativement $tkp(n)p(n)$, de sorte que le mot est accepté si lorsqu'on calcule tout le circuit, l'état d'acceptation se trouve tout en haut,

en position $(0, p(n))$. Ajoutons au dessus de ce gros circuit un petit circuit qui vaut 1 si l'état en position $(0, p(n))$ est bien égal à l'état d'acceptation.

Dans ce cas, on voit que le mot est accepté si et seulement si ce circuit total vaut 1. On peut voir qu'en fait la transformation qui au tableau associe le circuit se fait en espace logarithmique (cela vient du fait que la fonction g qui apparaît est la même partout), de sorte qu'on a prouvé $A \leq_m CIRCUIIT$ ■

Théorème 6.7

Il existe un problème **NP**-complet. Plus précisément le problème suivant est **NP**-complet.

NOM: CIRCUITSAT

ENTREE: Un circuit avec des variables

QUESTION: Est-ce qu'il existe un choix des variables pour lequel la valeur du circuit est 1 ?

Preuve : Ce problème est dans **NP**.

Si A est dans **NP**, on sait d'après le théorème 4.6 qu'il existe B dans **P** et un polynôme p tel que $x \in A \iff \exists y, |y| \leq p(|x|), (x, y) \in B$.

En reprenant la démonstration précédente sur le problème B , on voit qu'on peut construire un circuit tel que $x \in A$ si et seulement si il existe un choix des variables $y_1 \dots y_k$ (représentant y) tel que le circuit est vrai. Ainsi $A \leq CIRCUIITSAT$. ■

Notons que ce résultat n'est PAS un corollaire du théorème précédent. Il s'obtient cependant en utilisant la même preuve.

Théorème 6.8

Le problème SAT est **NP**-complet.

NOM: SAT

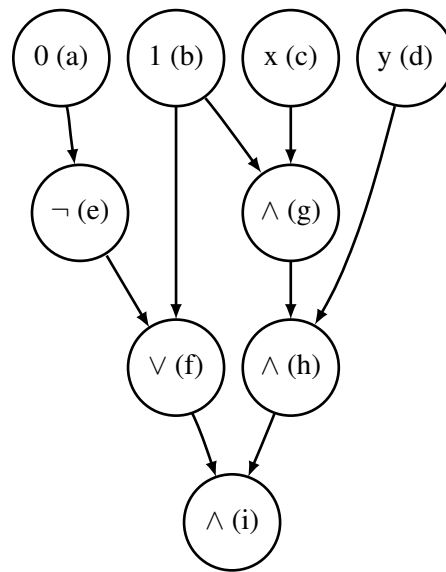
ENTREE: Une formule φ contenant des variables

QUESTION: Est-ce qu'il existe un choix des variables tel que φ soit vrai ?

On dit que φ est satisfaisable s'il existe un choix de variable tel que φ est vraie.

Preuve : Il est clair qu'il est dans **NP**. Pour montrer qu'il est **NP**-dur, il suffit de montrer que $CIRCUIITSAT \leq_m SAT$.

Pour cela, on va transformer un circuit en une formule : Chaque élément du circuit est remplacé par une variable. Illustrons la transformation par un exemple.



On obtient les formules suivantes :

$$\varphi_1 = \neg a$$

$$\varphi_2 = b$$

$$\varphi_3 = (c \vee \neg x) \wedge (\neg c \vee x)$$

$$\varphi_4 = (d \vee \neg y) \wedge (\neg d \vee y)$$

$$\varphi_5 = (e \vee a) \wedge (\neg e \vee \neg a)$$

$$\varphi_6 = (\neg f \vee b \vee e) \wedge (\neg b \vee f) \wedge (\neg e \vee f)$$

$$\varphi_7 = (g \vee \neg b \vee \neg c) \wedge (b \vee \neg g) \wedge (c \vee \neg g)$$

$$\varphi_8 = (h \vee \neg g \vee \neg d) \wedge (g \vee \neg h) \wedge (d \vee \neg h)$$

$$\varphi_9 = (i \vee \neg f \vee \neg h) \wedge (f \vee \neg i) \wedge (h \vee \neg i)$$

$$\varphi_{10} = i$$

φ_1 signifie que $a = 0$, φ_4 signifie que $d = y$, φ_7 signifie que $g = b \wedge c$, φ_5 signifie que $e = \neg a$, etc. φ_{10} est particulière, et signifie que la sortie du circuit vaut 1.

De sorte que $\bigwedge_j \varphi_j$ est vraie si et seulement si il existe un choix des variables x_i tel que le circuit vaut 1. Le calcul de $\bigwedge_j \varphi_j$ connaissant le circuit est en espace logarithmique, ce qui finit la preuve. ■

En examinant la preuve, on s'aperçoit que la formule finale est d'une forme particulière : c'est un "et" (\wedge) de "ou" (\vee) de formules atomiques. Cette forme particulière est appelée forme normale conjonctive.

Théorème 6.9

Le problème *CNF – SAT* est **NP**-complet.

NOM: CNF-SAT

ENTREE: Une formule φ en forme normale conjonctive

QUESTION: Est-ce que φ est satisfaisable ?

Dans la forme normale conjonctive, $\phi = \bigwedge C_i$, les C_i s'appellent des clauses. $C_i = \bigvee l_{ij}$, les l_{ij} s'appellent des littéraux. l_{ij} est soit une variables x , soit une négations de variable $\neg x$.

Considérons maintenant les deux problèmes suivants

NOM: 3SAT

ENTREE: Une formule φ en forme normale conjonctive, ayant 3 littéraux par clause

QUESTION: Est-ce que φ est satisfaisable ?

NOM: (≤ 3) SAT

ENTREE: Une formule φ en forme normale conjonctive, ayant au plus 3 littéraux par clause

QUESTION: Est-ce que φ est satisfaisable ?

En examinant la preuve du théorème précédent, on s'aperçoit qu'on a démontré :

Corollaire 6.10

(≤ 3) SAT est NP-dur (et en fait NP-complet).

Théorème 6.11

3SAT est NP-complet

Preuve : 3SAT est dans NP.

Pour montrer que 3SAT est NP-dur, il suffit de montrer (≤ 3) SAT \leq_m 3SAT.

Prenons donc ϕ une formule avec moins de 3 littéraux par clause, et transforme ϕ en ϕ' .

- On ne transforme pas les clauses ayant 3 littéraux.
- Si une clause C_i n'a que deux littéraux, $C_i = l_1 \vee l_2$, on la transforme en deux clauses $l_1 \vee l_2 \vee x$ et $l_1 \vee l_2 \vee \neg x$, pour une nouvelle variable x .
- Si une clause C_i n'a qu'un littéral, $C_i = l_1$, on la transforme en quatre clauses $l_1 \vee x \vee y$, $l_1 \vee x \vee \neg y$, $l_1 \vee \neg x \vee y$ et $l_1 \vee \neg x \vee \neg y$, pour deux nouvelles variables x et y .

En réfléchissant, on voit que $\phi \in (\leq 3)$ SAT $\iff \phi' \in 3SAT$. La transformation $\phi \mapsto \phi'$ est bien en espace logarithmique, ce qui finit la preuve. ■

QUELQUES PROBLÈMES NP-COMPLETS

7.1 Méthode

Pour montrer qu'un problème A est NP-complet :

- Montrer que A est dans NP (c'est souvent assez facile).
- Trouver un problème B proche de A dont on sait déjà qu'il est NP-dur et montrer que A est plus dur que B ($B \leq_m A$). Il faut donc transformer le problème B en A .

7.2 Variantes de SAT

Théorème 7.1

Le problème suivant est NP-complet

NOM: 3-3-SAT

ENTREE: Une formule φ avec 3 littéraux par clause, chaque littéral apparaissant au plus 3 fois

QUESTION: Est-ce que φ est satisfaisable ?

Preuve : Le problème est évidemment dans NP. Réduisons 3SAT à 3-3-SAT. Partons d'une formule φ avec 3 littéraux par clause. Prenons un littéral, qu'on appelle x qui apparaît le nombre maximum de fois. Soit n le nombre de fois où il apparaît. Sa négation, \bar{x} apparaît donc aussi moins de n fois. Construisons les formules suivantes :

$$\left\{ \begin{array}{l} x_1 \vee \bar{x}_2 \vee y_1 \\ x_1 \vee \bar{x}_2 \vee \bar{y}_1 \\ x_2 \vee \bar{x}_3 \vee y_2 \\ x_2 \vee \bar{x}_3 \vee \bar{y}_2 \\ \vdots \\ x_n \vee \bar{x}_1 \vee y_n \\ x_n \vee \bar{x}_1 \vee \bar{y}_n \end{array} \right.$$

Il est assez clair que si cet ensemble de formules est satisfaisable, alors $x_1 = x_2 = \dots = x_n$. Si on remplace alors dans les autres formules la première occurrence de x (resp. \bar{x}) par x_1 (resp. \bar{x}_1), la deuxième occurrence de x par x_2 , etc, et qu'on y ajoute l'ensemble de formules ci-dessus, on obtient une nouvelle formule ψ telle que ψ est satisfaisable si et seulement φ est satisfaisable. De plus, la variable x n'apparaît plus, et les nouvelles variables introduites sont telles que chaque littéral apparaît au plus 3 fois.

Il suffit alors de répéter cette construction jusqu'à ce qu'aucune variable n'apparaisse plus de 3 fois. ■

Théorème 7.2

Le problème suivant est NP-complet

NOM: NAE (Not-All-Equal)

ENTREE: Une formule φ avec 3 littéraux par clause

QUESTION: Est-ce que φ est satisfaisable de sorte que aucune clause ne contienne 3 littéraux vrais ?

Avant de passer à la preuve, donnons quelques remarques :

- $\varphi \in \text{NAE}$ si et seulement si φ est satisfaisable de sorte que chaque clause contienne un littéral vrai et un littéral faux.
- $\varphi \in \text{NAE}$ si et seulement si φ est satisfaisable simultanément par une assignation $x_1 \dots x_n$ et par son assignation opposée $\bar{x}_1 \dots \bar{x}_n$ où on échange la valeur de toutes les variables.
- $\varphi \in \text{NAE}$ si et seulement si $\varphi \wedge \bar{\varphi}$ est satisfaisable, où $\bar{\varphi}$ dénote la formule opposée de φ où on a remplacé tous les littéraux par leur négation.

Notons que la dernière remarque prouve $\text{NAE} \leq_m \text{3SAT}$, ce qu'on savait déjà (pourquoi ?).

Preuve : NAE est évidemment dans NP. Montrons que NAE est NP-dur en montrant $\text{3SAT} \leq_m \text{NAE}$.

Partons d'une formule φ . Introduisons une variable spéciale F . Transformons chaque clause $l_1 \vee l_2 \vee l_3$ en

$$(l_1 \vee l_2 \vee c) \wedge (l_3 \vee \bar{c} \vee F)$$

en introduisant une variable nouvelle c par clause.

Appelons ψ la nouvelle formule.

Montrons que $\varphi \in \text{3SAT} \iff \psi \in \text{NAE}$

- Supposons $\varphi \in \text{3SAT}$. Prenons une assignation des variables qui le prouve. On construit une assignation pour ψ de la façon suivante : On met F à 0. On met c à la valeur de $\bar{l}_1 \vee l_2$.
 - Si l_1 ou l_2 est vrai, la première formule $(l_1 \vee l_2 \vee c)$ est vraie et contient un littéral vrai (l_1 ou l_2) et un littéral faux (c). La deuxième formule $(l_3 \vee \bar{c} \vee F)$ contient un littéral vrai (\bar{c}) et un littéral faux (F)
 - Si l_1 et l_2 sont faux, alors l_3 est vraie puisque $l_1 \vee l_2 \vee l_3$ est vrai dans la formule φ de départ. la première formule $(l_1 \vee l_2 \vee c)$ contient un littéral vrai (c) et un littéral faux (l_1). La deuxième formule $(l_3 \vee \bar{c} \vee F)$ contient un littéral vrai (l_3) et un littéral faux (F).

Donc $\psi \in \text{NAE}$.

- Supposons $\psi \in \text{NAE}$. Prenons une assignation qui le prouve. Comme on l'a indiqué plus haut, l'assignation et l'assignation opposée prouvent que ψ est satisfaisable. On peut donc supposer sans perte de généralité que la variable F est à la valeur 0. Dans ce cas, comme $(l_1 \vee l_2 \vee c) \wedge (l_3 \vee \bar{c} \vee F)$ est vraie, on en déduit, en raisonnant sur la valeur de c , que $l_1 \vee l_2 \vee l_3$ est vraie.

Donc $\varphi \in \text{3SAT}$.

La réduction s'effectue bien en espace logarithmique de sorte qu'on a prouvé $\text{3SAT} \leq_m \text{NAE}$. ■

7.3 CLIQUE

Théorème 7.3

Le problème suivant est NP-complet

NOM: CLIQUE

ENTREE: Un graphe G et un entier k

QUESTION: Est-ce que G a une clique de taille k ?

On rappelle qu'une clique de taille k est un ensemble de k sommets du graphe tel que tous les points de cet ensemble soient reliés deux à deux.

Preuve : CLIQUE est dans NP, il suffit de considérer un algorithme non-déterministe qui choisit k sommets et vérifie qu'ils forment une clique.

Pour montrer que CLIQUE est NP-dur, on va montrer qu'il est plus dur que 3SAT dont on sait déjà qu'il est NP-dur.

On va donc convertir une formule φ de 3SAT en un graphe G et un entier k de sorte que φ est satisfaisable si et seulement si G a une clique de taille k .

La transformation s'effectue comme suit : Soit φ une formule avec p clauses.

- Pour chaque clause $l_1 \vee l_2 \vee l_3$, créer trois sommets l_1, l_2, l_3 .
- Relier les sommets x à tous les sommets qui ne sont pas dans sa clause et qui ne sont pas \bar{x}

Le graphe G obtenu pour l'exemple $\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w) \wedge (\bar{x} \vee z \vee \bar{w})$ est donné dans la figure 7.1

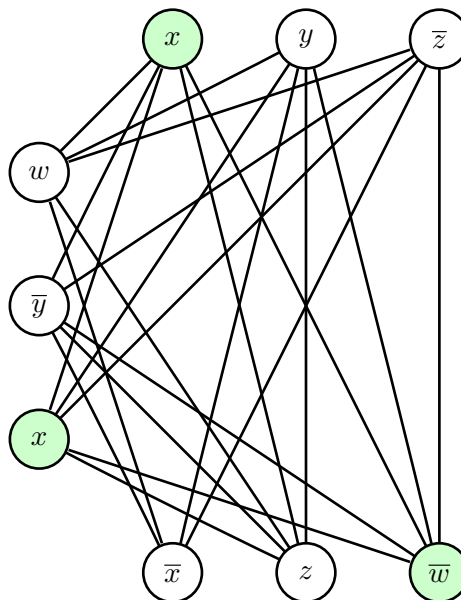


FIGURE 7.1 – Graphe correspondant à la formule $\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w) \wedge (\bar{x} \vee z \vee \bar{w})$. Les trois sommets grisés forment une clique de taille 3, ce qui est cohérent avec le fait que l'assignation $x = 1, w = 0$ permet de satisfaire la formule.

Montrons alors

φ est satisfaisable si et seulement si G a une clique de taille p .

- Supposons φ satisfaisable. Prenons donc une assignation des variables tels que φ soit vraie. Chaque clause contient donc au moins un littéral vrai. Choisissons, dans chaque clause, un littéral vrai. L'ensemble E ainsi formé est une clique de taille p : On a choisi un littéral par clause, il y a donc p sommets. De plus, deux sommets quelconques de E sont reliés par une arête : En effet, on ne peut pas avoir en même temps x et \bar{x} dans E puisque x et \bar{x} ne peuvent être simultanément vrais dans l'assignation qu'on a choisi. E est donc bien une clique.
- Supposons que G a une clique de taille p , qu'on va noter F . Comme deux sommets d'une même clause ne sont pas reliés entre eux, F contient donc un sommet par clause. Si F contient un sommet de la forme x , on pose $x = 1$. Si F contient un sommet de la forme \bar{x} , on pose $x = 0$. On ne peut pas poser en même temps $x = 1$ et $x = 0$ puisque x et \bar{x} ne sont pas reliés par une arête donc ne peuvent pas appartenir tous les deux à F .

On vérifie alors aisément que l'assignation (partielle) ainsi choisie permet de satisfaire chaque clause.

La transformation f de φ vers (G, p) se fait en espace logarithmique de sorte qu'on a prouvé $3SAT \leq_m CLIQUE$. ■

Corollaire 7.4

Les deux problèmes suivants sont aussi NP-durs (et même NP-complets)

NOM: INDEPENDANT

ENTREE: Un graphe G et un entier k

QUESTION: Est-ce que G admet un ensemble indépendant de taille k ?

NOM: VERTEX COVER

ENTREE: Un graphe G et un entier k

QUESTION: Est-ce que G admet une couverture de taille k ?

Le corollaire est laissé en exercice.

7.4 Couplages

L'objet qu'on va maintenant étudier a une définition un peu compliquée, commençons par donner un exemple.

On s'intéresse à l'organisation d'un festival de musique dans la ville de Marseille. On dispose de n artistes, chaque artiste acceptant de faire un seul concert. On souhaite évidemment faire participer chacun des artistes. On dispose de n lieux de concerts différents, et on souhaite que chacun des lieux organise un concert. Enfin, le concert se déroule sur n dates différentes, et on ne veut pas que deux concerts aient lieu en même temps.

On cherche donc à résoudre ce problème, c'est à dire à trouver comment mettre chaque artiste dans un lieu donné à une date donnée. Cependant, le problème se corse : Certaines salles ne sont pas disponibles à certains moments, de même que certains artistes, certains artistes refusent de jouer dans certaines salles, sauf si c'est un samedi, etc.

Appelons A les artistes, C les lieux de concerts, et D les dates. Chaque contrainte possible se traduit par un triplet $(a, c, d) \in A \times C \times D$ telle que l'artiste a a la possibilité de jouer dans la salle c le jour d .

◆ Exemple

Artistes	Lieux	Dates
Alexis (A)	Le Passe-Temps (PT)	Jeudi (J)
Béatrice (B)	Le CCO	Vendredi (V)
Christophe (C)	Le Théâtre Sous Vide (T)	Samedi (S)

Et voilà les différentes possibilités :

(A, PT, V)
 (A, CCO, J)
 (B, CCO, J)
 (B, T, S)
 (B, T, V)
 (C, PT, S)
 (C, PT, J)
 (C, T, V)

Comment faire ?

Théorème 7.5

Le problème suivant est NP-complet :

NOM: 3-DM (3-Dimensional-Matching)

ENTREE: Des ensembles A, C, D disjoints de cardinal n et un ensemble de contraintes $W \subset A \times C \times D$

QUESTION: Est-ce qu'on peut construire un couplage parfait ?

Il s'agit de 3-Dimensional Matching puisqu'on se donne 3 ensembles. Lorsqu'il n'y en a que deux, le problème est dans **P**, et s'appelle, entre autres, le problème des mariages.

Preuve : Le problème est évidemment dans **NP**. Pour montrer qu'il est **NP-dur**, on va montrer qu'il est plus dur que 3-3-SAT.

Partons donc d'une formule φ . Notons n le nombre de variables, et p le nombre de clauses. Comme chaque littéral n'apparaît qu'au plus 3 fois, on va les numéroter dans

la formule. Par exemple, si on part de

$$\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w) \wedge (x \vee z \vee w)$$

On écrira plutôt :

$$\varphi = (x_1 \vee y_1 \vee \bar{z}_1) \wedge (x_2 \vee \bar{y}_1 \vee w_1) \wedge (x_3 \vee z_1 \vee w_2)$$

- Pour chaque variable x , on construit trois artistes A, B, C , trois salles de concerts D, E, F et 6 dates possibles, x_1, x_2, x_3 et $\bar{x}_1, \bar{x}_2, \bar{x}_3$.

Les contraintes qui les régissent sont les suivantes

$$\begin{aligned} &(A, D, x_1) \\ &(B, E, x_2) \\ &(C, F, x_3) \\ &(A, E, \bar{x}_1) \\ &(B, F, \bar{x}_2) \\ &(C, D, \bar{x}_3) \end{aligned}$$

Ces contraintes sont les seules où apparaissent A, B, C, D, E, F . On s'aperçoit alors que pour obtenir un couplage parfait, il n'y a que deux possibilités. Dans la première, les horaires $x_1 \dots x_3$ n'ont pas été utilisés. Dans la seconde, les horaires $\bar{x}_1 \dots \bar{x}_3$ n'ont pas été utilisés.

- Pour chaque clause $C = l_1 \vee l_2 \vee l_3$ on crée un artiste A et un lieu D , et on crée les trois contraintes

$$\begin{aligned} &(A, D, l_1) \\ &(A, D, l_2) \\ &(A, D, l_3) \end{aligned}$$

Pour pouvoir écouter l'artiste A , il n'y a donc pas le choix : il faut nécessairement avoir accès à l'une des trois dates $l_1 \dots l_3$. La conjonction des deux ensembles de contraintes ci-dessus nous assure donc qu'on a choisi pour chaque variable si elle est vraie ou fausse (en se donnant la possibilité d'accéder à x_1, x_2, x_3 ou $\bar{x}_1 \dots \bar{x}_3$), et ensuite que chaque clause contient un littéral vrai.

- Enfin, tous les horaires n'ont pas nécessairement été pris : On a pris $3n$ horaires dans la première partie, et p dans la deuxième. Il reste donc exactement $6n - (3n + p) = 3n - p = k$ horaires qui n'ont pas été pris. On introduit pour finir des artistes $A_1 \dots A_k$, des salles $B_1 \dots B_k$ et les contraintes

$$(A_i, B_i, h)$$

pour tout i et pour tout horaire h . Autrement dit il s'agit de k artistes qui sont capables de jouer dans leur salle à n'importe quelle date.

On peut se convaincre que les ensembles d'artistes, salles et horaires obtenus ainsi sont tels qu'ils admettent une solution si et seulement si la formule ϕ de départ est satisfaisable, et qu'on peut construire ces ensembles en espace polynomial, ce qui termine la réduction. ■

PROBLÈMES NP-COMPLETS SUR LES NOMBRES

8.1 Une autre variante de SAT

Théorème 8.1

Le problème suivant est NP-complet

NOM: OIT (One In Three)

ENTREE: Une formule φ avec 3 littéraux par clause

QUESTION: Est-ce que φ est satisfaisable de sorte que chaque clause contienne un et un seul littéral vrai

Preuve : Le problème est évidemment dans NP. Réduisons 3SAT à OIT. Partons d'une formule φ avec 3 littéraux par clause.

Transformons chaque clause $\varphi = l_1 \vee l_2 \vee l_3$ en trois clauses

$$(l_1 \vee a \vee b) \wedge (\bar{l}_2 \vee a \vee c) \wedge (\bar{l}_3 \vee b \vee d)$$

Appelons ψ le résultat de la transformation.

Par exemple, sur la formule $\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w)$ on obtient donc

$$\psi = (x \vee a_1 \vee b_1) \wedge (\bar{y} \vee a_1 \vee c_1) \wedge (z \vee b_1 \vee d_1) \wedge (x \vee a_2 \vee b_2) \wedge (y \vee a_2 \vee c_2) \wedge (\bar{w} \vee b_2 \vee d_2)$$

Montrons que $\phi \in 3SAT \iff \psi \in OIT$.

– Supposons $\phi \in 3SAT$, et prenons donc un assignement qui le prouve. On va construire un assignement pour ψ qui va prouver que $\psi \in OIT$. Prenons une clause de ϕ , c'est à dire $l_1 \vee l_2 \vee l_3$, qu'on va transformer en $(l_1 \vee a \vee b) \wedge (\bar{l}_2 \vee a \vee c) \wedge (\bar{l}_3 \vee b \vee d)$

Fixons les valeurs de a, b, c, d en fonction des valeurs de l_1, l_2, l_3 grâce à la table suivante

l_1	l_2	l_3	a	b	c	d
0	0	1	0	1	0	0
0	1	0	1	0	0	0
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	0	1	0
1	1	1	0	0	1	1

On remarque alors que dans chacun des cas, les trois clauses $(l_1 \vee a \vee b)$, $(\bar{l}_2 \vee a \vee c)$ et $(\bar{l}_3 \vee b \vee d)$ sont bien satisfaites par un assignement pour lequel un seul littéral est vrai par clause.

- Supposons maintenant que $\psi \in OIT$ et prenons un assignement qui le prouve. Prenons une clause de ϕ , c'est à dire $l_1 \vee l_2 \vee l_3$, qu'on va transformer en $(l_1 \vee a \vee b) \wedge (\bar{l}_2 \vee a \vee c) \wedge (\bar{l}_3 \vee b \vee d)$
 - Si $a = 1$, alors comme $\bar{l}_2 \vee a \vee c$ ne contient qu'un littéral vrai, on en déduit $\bar{l}_2 = 0$ donc $l_2 = 1$ donc $l_1 \vee l_2 \vee l_3 = 1$.
 - Si $b = 1$ on aboutit de même à $l_3 = 1$ donc $l_1 \vee l_2 \vee l_3 = 1$.
 - Le dernier cas est donc $a = 0$ et $b = 0$. Dans ce cas comme $l_1 \vee a \vee b$ contient un littéral vrai, on en déduit $l_1 = 1$, donc $l_1 \vee l_2 \vee l_3 = 1$.
- Dans tous les cas $l_1 \vee l_2 \vee l_3 = 1$ de sorte que l'assignation prouve que $\phi = 1$.
Donc $\phi \in 3SAT$.

La transformation de ϕ à ψ se fait bien en espace logarithmique et prouve bien $3SAT \leq_m OIT$. ■

8.2 SUBSET-SUM et les variantes

8.2.1 NP-complétude

Théorème 8.2

Le problème suivant est NP-complet

NOM: SUBSETSUM

ENTREE: Des entiers a_i et un entier K

QUESTION: Est-ce qu'on peut sommer certains des a_i de sorte à obtenir exactement K ?

Preuve : SUBSETSUM est dans NP, on va montrer que SUBSETSUM est NP-dur en montrant qu'il est plus dur que OIT.

Partons d'une formule ϕ avec 3 littéraux par clause. Numérotons les clauses de 1 à m et les variables de 1 à n .

Si x est un littéral, on note $c_j(x) = 1$ si x appartient à la j ème clause et $c_j(x) = 0$ sinon.

- Si x est la i ème variable, on lui associe l'entier a_x défini par

$$a_x = \sum c_j(x)10^j + 10^{i+m}$$

- Si x est la i ème variable, on lui associe l'entier $a_{\bar{x}}$ défini par

$$a_{\bar{x}} = \sum c_j(\bar{x})10^j + 10^{i+m}$$

Posons finalement $K = \sum_{j=1}^m 10^j + \sum_{i=1}^n 10^{i+m}$.

Pour mieux comprendre, effectuons la transformation sur l'entrée

$$\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w) \wedge (\bar{x} \vee z \vee \bar{w}) = c_1 \wedge c_2 \wedge c_3$$

On construit le tableau suivant :

	x	y	z	w	c_1	c_2	c_3
a_x	1	0	0	0	1	1	0
$a_{\bar{x}}$	1	0	0	0	0	0	1
a_y	0	1	0	0	1	0	0
$a_{\bar{y}}$	0	1	0	0	0	1	0
a_z	0	0	1	0	0	0	1
$a_{\bar{z}}$	0	0	1	0	1	0	0
a_w	0	0	0	1	0	1	0
$a_{\bar{w}}$	0	0	0	1	0	0	1
K	1	1	1	1	1	1	1

Les nombres qu'on obtient sont donc 1000110, 1000001, 0100100, 0100000, 0010001, 001100, 0001010, 0001001 (Note : bien que les nombres construits ne contiennent que des 0 et des 1 ils sont bien en base 10).

On s'aperçoit alors que $\varphi \in OIT$ si et seulement si certains de ces nombres somment à $K = 111\dots 1$. En effet, les nombres sont choisis de telle façon qu'on soit obligé de choisir a_x ou $a_{\bar{x}}$ mais pas les deux à la fois, et qu'on soit obligé de choisir exactement un littéral par clause. ■

Corollaire 8.3

Le problème suivant est NP-complet.

NOM: PARTITION

ENTREE: Des entiers a_i

QUESTION: Est-ce qu'on peut partager les a_i en deux parties de même somme ?

Preuve : On a montré dans un cours précédent que SUBSETSUM \leq_m PARTITION de sorte que PARTITION est NP-dur. Il est clair que PARTITION est dans NP ce qui conclut le raisonnement. ■

8.2.2 Un algorithme pour SUBSETSUM

Soit $(a_i)_{1 \leq i \leq n}$ des entiers et K un entier.

Considérons le tableau $T[j, s]$ tel que $T[j, s] = 1$ si et seulement si on peut atteindre la somme s en sommant certains des j premiers entiers a_i .

L'algorithme suivant permet de calculer T :

- Pour s de 1 à K poser $T[0, s] = 0$
- poser $T[0, 0] = 1$
- Pour j de 1 à n , pour s de 0 à K .
 - Si $T[j-1, s] = 1$ ou $T[j-1, s - a_j] = 1$ poser $T[j, s] = 1$

En effet, on peut atteindre s en sommant certains des j premiers a_i soit en ne prenant pas le j ème (auquel cas il faut atteindre s avec les $j - 1$ premiers) soit en le prenant (auquel cas il faut atteindre $s - a_j$ avec les $j - 1$ premiers).

Cet algorithme permet de résoudre SUBSETSUM : Une fois que le tableau T est constitué, il suffit de savoir si $T[n, K]$ est égal à 1 ou non.

La complexité de cet algorithme est polynomial en n et en K . Bien entendu, si K est donné en binaire, cela nous donne un algorithme exponentiel pour SUBSETSUM. En revanche, si K est donné en unaire, l'algorithme est polynomial. De sorte que le fait que SUBSETSUM est NP-complet provient uniquement du fait que K est en binaire.

Définition 8.1 (NP-complet au sens faible/fort)

Un problème est NP-complet au sens faible s'il est NP-complet et que sa variante où tous les nombres sont donnés en unaire est polynomiale. Dans le cas où la variante reste NP-complet, on dit que le problème est NP-complet au sens fort.

◆ Exemple

- SUBSETSUM et PARTITION sont NP-complets au sens faible
- 3SAT est NP-complet au sens fort (puisque'il n'y a pas de nombres dans 3 SAT, les donner en unaire ou en binaire ne change rien)
- CLIQUE est NP-complet au sens fort (donner la taille de la clique en unaire ou en binaire n'a aucune importance).

8.3 Problèmes NP-complets au sens fort

Théorème 8.4

Le problème suivant est NP-complet au sens fort

NOM: ILP (Integer Linear Programming)

ENTREE: Un système d'équations et d'inéquations du type $\sum a_i x_i \leq N$
où tous les coefficients sont des entiers positifs

QUESTION: Est-ce qu'il existe des entiers positifs x_i solutions du système ?

Remarque : Si on omet les restrictions sur la forme particulière des inégalités, le problème reste NP-complet, mais il est plus délicat de montrer qu'il est dans NP. Si on cherche une solution avec x_i rationnels plutôt qu'entier, il existe un algorithme polynomial.

Preuve : Montrons que le problème est dans NP. L'algorithme est le suivant : Deviner la valeur de chacun des entiers x_i puis tester si le système est satisfait. Du fait de la forme très particulière des équations, on connaît une borne (polynomiale) sur chacun des x_i , de sorte que deviner chaque x_i prend un temps polynomial (Par exemple, si on a l'équation $3x + 4y \leq 18$ on sait qu'on peut se restreindre aux x qui vérifient $x \leq 18/3 = 9$).

Pour montrer que le problème est NP-dur, on montre que $OIT \leq_m ILP$.

Donnons uniquement l'idée sur un exemple, le raisonnement général s'en déduisant.
Partant de la formule

$$\varphi = (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee w) \wedge (\bar{x} \vee z \vee \bar{w})$$

on construit le système d'équations :

$$\left\{ \begin{array}{l} x + y + \bar{z} = 1 \\ x + \bar{y} + w = 1 \\ \bar{x} + z + \bar{w} = 1 \\ x + \bar{x} = 1 \\ y + \bar{y} = 1 \\ z + \bar{z} = 1 \\ w + \bar{w} = 1 \end{array} \right.$$

Il est clair que le système d'équations a une solution si et seulement si la formule est dans OIT.

De plus, comme les coefficients valent 0 ou 1, les encoder en binaire ou en unaire ne change pas la taille de la formule, de sorte que le problème est NP-complet même lorsque les entiers sont donnés en unaire. ■

Théorème 8.5

Le problème suivant est NP-complet au sens fort.

NOM: BINPACKING

ENTREE: Des entiers a_i , deux entiers K et S

QUESTION: Peut-on partager les a_i en K paquets, chacun de somme inférieure à S ?

Remarque : si K est une constante, le problème devient polynomial.

Preuve : La preuve, pas très facile, a été esquissée en cours, mais n'est pas écrite ici. ■

ALGORITHMES D'APPROXIMATION

9.1 BINPACKING

Algorithme First-Fit : On met l'objet dans le premier sac où il passe. Soit (a_i) les objets.

Analyse de l'algorithme :

- Chaque sac (sauf éventuellement un) est à moitié rempli. Prenons en effet les deux sacs les moins remplis à la fin de l'exécution. Appelons les A et B et supposons que A soit avant B. Lorsqu'on a mis un objet dans B, c'est qu'on n'a pas pu le mettre dans A. Si A et B sont à moitié vides, alors l'objet qu'on a mis dans B fait moins de la moitié de la taille d'un sac, contradiction.
- Pour simplifier, on oublie ce sac qui n'est pas forcément à moitié rempli.
- Soit $N = \sum a_i$ et K la capacité d'un sac. Appelons p le nombre de sacs dans notre algorithme. Comme chaque sac dans l'algorithme est à moitié rempli, le nombre de sac est inférieur à $p \leq 2N/K$
- Soit p^* le nombre de sacs qu'il faut au minimum. Il faut au moins N/K sacs, donc $p^* \geq N/K$.
- On en déduit $p^* \leq p \leq 2p^*$.

9.2 PARTITION

MIN-WEIGHT : minimiser le plus grand entre $\sum_{i \in I} a_i$ et $\sum_{i \notin I} a_i$. Dit autrement, on a deux sacs, on veut minimiser le poids du sac le plus lourd.

Idée : On se donne ϵ . Posons $L = \sum_{i \in I} a_i$. Considérons les objets de poids supérieurs à ϵL . Il y en a peu, genre au plus $K = 1/\epsilon$. Tester toutes les affectations possibles de ces objets. Pour chaque affectation, on affecte les autres objets de manière gloutonne (mettre chaque objet dans le sac actuellement le moins lourd). On prend alors le minimum parmi toutes les K affectations testées. Par construction, on se trompera au plus de ϵL .

9.3 KNAPSACK

On sait que si le prix P à atteindre est petit, on a un algo polynomial. Appelons $f(n, P)$ sa complexité.

Prenons un vrai problème (p_i, v_i, V) . Notons $Q = \max p_i$.

On pose $K = \epsilon Q/n$. Remplaçons chaque p_i par $p'_i = p_i/K$ (division entière). On peut donc utiliser l'algorithme décrit précédemment (programmation dynamique). Sa complexité est maintenant polynomiale.

Maintenant, notons P le prix obtenu, P^l le prix obtenu lorsqu'on reprend les valeurs d'origine et gardons le même choix que celui qui nous a permis d'atteindre P . Notons enfin et P^* le meilleur prix possible. Le but est de comparer P^* à P^l .

- $P^* \leq KP + Kn$. Prenons la solution qui permet d'atteindre le meilleur prix. Lorsqu'on divise tout par K et qu'on arrondit inférieurement, cette solution nous donne un prix d'au moins $P^*/K - n$. Or P correspond au prix maximum possible, donc $P \geq P^*/K - n$.
- $P^l \geq KP$. (Clair)

D'où $P^l \geq KP \geq P^* - Kn \geq P^* - \epsilon Q \geq P^*(1 - \epsilon)$.
donnée.

INDEX

∞

(≤ 3)SAT, 37
3-3-SAT, 39
3-DM (3-Dimensional-Matching), 43
3SAT, 37

V

VERTEX COVER, 42

B

BINPACKING, 49

C

CIRCUIT, 34
CIRCUITSAT, 35
CLIQUE, 41
CNF-SAT, 36

G

GAP (Graph Accessibility Problem), 33

I

ILP (Integer Linear Programming), 48
INDEPENDANT, 42

N

NAE (Not-All-Equal), 40

O

OIT (One In Three), 45

P

PARTITION, 47

S

SAT, 35
SUBSETSUM, 46

LISTE DES DÉFINITIONS

1.1 - Machine de Turing	3
2.5 - $\text{DTIME}(f)$	10
2.6 - $\text{DSPACE}(f)$	10
3.3 - $\text{NTIME}(f)$	20
3.4 - $\text{NSPACE}(f)$	20
4.1 - P et NP	23
5.2 - Fonction one-way	28
6.2 - Réduction \leq_m	31
6.3 - Problèmes \mathcal{C} -complets	33
8.1 - NP -complet au sens faible/fort	48