CHAPITRE 10

La calculabilité

Jean-Marc Alliot

10.1 Introduction

Nous avons introduit les notions fondamentales de calculabilité et de décidabilité dans le chapitre consacré aux systèmes formels. Nous n'y reviendrons pas.

D'autre part, la démonstration d'indécidabilité pour un problème A se fait le plus souvent par référence à un problème B dont on a déjà montré l'indécidabilité : on montre que si le problème A était décidable, alors le problème B le serait aussi. Nous considérons comme acquis le résultat démontré dans le chapitre 5 consacré aux systèmes formels *i.e.*, le prédicat $P_T = \exists n T(i,i,n)$ est indécidable, ou encore le comportement d'une machine de Turing avec comme argument son index est indécidable. Nous nous servirons de ce résultat pour établir notre deuxième théorème d'indécidabilité.

Nous ne parlerons dans ce chapitre que de quelques problèmes de décision classiques. Il aurait sans doute aussi fallu, pour être plus général, parler des problèmes de Thue. Nous ne l'avons pas fait. Le lecteur intéressé peut se reporter à (Salomaa 1989).

10.2 Problème de reconnaissance

On appelle problème de reconnaissance d'un mot m, pour un langage L donné (ici récursivement énumérable), le problème consistant à déterminer si ce mot appartient au langage L. Nous avons vu qu'un langage L est récursivement énumérable ssi il existe une machine de Turing permettant précisément de reconnaître chaque mot de L. Donc si le problème de la reconnaissance était décidable, cela signifierait que le prédicat P_T serait décidable, ce qui est faux. Donc le problème de reconnaissance est indécidable.

Théorème 10.1 – Indécidabilité du problème de reconnaissance – *Le problème consistant à déterminer si un mot m donné appartient au langage généré par une grammaire G donnée est indécidable.*

Il est intéressant de bien réfléchir à ce théorème ; il signifie qu'il est impossible de trouver un algorithme qui garantisse de déterminer en un temps fini, et dans tous les cas, si un mot m appartient à un langage $L(G)$ généré par une grammaire G (ou par un système de Post). Il est évident qu'il existe des grammaires G pour lesquelles il est possible

de trouver un tel algorithme, de même qu'il existe des mots pour lesquels le problème de l'appartenance est trivialement reconnu, mais l'algorithme général de reconnaissance *n'existe pas*.

10.3 Le problème de correspondance de Post

Nous allons nous intéresser de façon plus attentive au problème de correspondance de Post, car il fournit un exemple simple, très étudié, d'un problème indécidable. Nous noterons souvent le problème de correspondance de Post en abrégé PCP.

10.3.1 Énoncé intuitif du problème

Nous allons tout d'abord présenter le problème de Post de façon informelle.

Considérons les deux listes suivantes :

$$(aabb, ab, aab)(bb, abaa, b)$$

Supposons que je concatène les mots de ma première liste dans l'ordre (1,3,2,1). J'obtiens le mot : $m_1 = aabbaababaabb$. Si j'applique la même liste d'indices sur ma seconde liste, j'obtiens le mot $m_2 = bbbabaabb$. Les mots m_1 et m_2 sont différents. Mais est-il possible de construire une liste d'indices telle que $m_1 = m_2$? Cette question est le problème de correspondance de Post, appliqué ici aux listes (abb, ab, aba) et (bb, aba, ba). Dans le cas présent, cette *instance* du PCP a une solution, il suffit de prendre comme liste d'indices (2,3,2,1) qui nous donne : $abaababaabb$.

La question de la décidabilité du problème de correspondance de Post est donc : peut-on trouver un algorithme général permettant de dire si le PCP a une solution pour deux listes quelconques, en un temps fini ?

10.3.2 Démonstration de l'indécidabilité

Nous allons montrer que le PCP est équivalent au problème de l'appartenance à un langage récursivement énumérable. Ce dernier problème étant indécidable, le problème de Post le sera également.

Pour ce faire, nous allons, pour une grammaire donnée G, construire une instance du problème de Post telle que la résolution de cette instance permettra de savoir si un mot m appartient à $L(G)$. Réciproquement, suivant que $m \in L(G)$ ou $m \notin L(G)$, le problème de Post aura, ou n'aura pas, de solution.

Avant de nous lancer dans la construction du PCP pour une grammaire quelconque G, nous allons formaliser le problème de correspondance.

Définition 10.1 – Problème de correspondance de Post – *Soit f et g deux morphismes définis de Σ^* dans Δ^*, avec Σ et Δ deux alphabets.*

Le couple (f, g) est une instance du problème de correspondance de Post. Un mot m de Σ^* est une solution pour cette instance ssi $f(m) = g(m)$.

Ainsi, si nous considérons l'exemple précédent, nous avons :

– $\Sigma = \{1, 2, 3\}$
– $\Delta = \{a, b\}$
– f défini par : $f(1) = abb$, $f(2) = ab$, $f(3) = aba$
– g défini par : $g(1) = bb$, $g(2) = aba$, $g(3) = ba$

La solution de cette instance du problème de Post est le mot de Σ^*, $m = 2321$.

Considérons maintenant une grammaire $G = (\mathcal{R}, \Sigma_N, \Sigma_T, D)$. Nous supposons que les règles de production de $\mathcal{R}$ sont telles que λ n'apparaît ni dans la partie droite ni dans la partie gauche des règles de production[1]. Les règles de production seront : $\{\alpha_i \rightarrow \beta_i, 0 < i \leq n\}$.

Posons que $\Gamma = \Sigma_N \cup \Sigma_T = \{a_1, a_2, \ldots, a_p\}$ et construisons un alphabet $\Lambda = \{a'_i \mid a_i \in \Gamma\}$.

Nous considérons alors l'alphabet $\Delta = \Gamma \cup \Lambda \cup \{I, F, \#, \$\}$. Cet alphabet sera l'alphabet d'arrivée de nos morphismes f et g. L'alphabet de départ sera : $\Sigma = \{1, 2, 3, \ldots, 2n + 2p + 4\}$. Nous notons également α'_i et β'_i les mots α_i et β_i dans lesquels on a substitué les lettres a'_i à toutes les occurrences des lettres a_i.

Nous disons que l'instance du PCP que nous venons de construire admet une solution ssi le mot m appartient à $L(G)$, les morphismes f et g étant définis par le tableau suivant :

	1	2	3	4	$4+i$	$4+p+i$	$4+2p+j$	$4+2p+n+j$
f	$Ia_1\#$	\$	#	F	a'_i	a_i	β'_j	β_j
g	I	#	\$	$\$mF$	a_i	a'_i	α_j	α'_j

La démonstration complète de ce théorème est relativement technique, le lecteur peut se reporter à (Salomaa 1973) pour plus de détails.

Intuitivement, les morphismes f et g simulent le comportement de la grammaire G. Le marqueur I désigne le début de la dérivation, le marqueur F, la fin de la dérivation, et les marqueurs # et \$ séparent les étapes des dérivations. L'astuce consiste à faire en sorte que le morphisme f démarre plus vite que le morphisme g et va effectivement générer le mot m, alors que le morphisme g ne pourra combler son retard que si le mot m est bien le résultat de la construction, en utilisant son indice 4.

Voyons tout cela sur un exemple. Soit la grammaire :
– $\Gamma = \{a_1 = D, a_2 = a, a_3 = b\}$, $(p = 3)$
– les règles de production sont : $D \rightarrow bDb$, $D \rightarrow aa$, $D \rightarrow aD$, $(n = 3)$

Le mot $baaab$ appartient à $L(G)$. On peut en effet l'obtenir par la dérivation : $D \rightsquigarrow bDb \rightsquigarrow baDb \rightsquigarrow baaab$.

Construisons l'instance du problème de Post associée à cette grammaire et à ce mot m :
– $\Delta = \{a_1 = D, a_2 = a, a_3 = b, a'_1 = D', a'_2 = a', a'_3 = b', I, F, \#, \$\}$
– $\Sigma = \{1, 2, 3, \ldots, 15, 16\}$

Nous pouvons alors construire le tableau des morphismes f et g :

1 Nous avons démontré dans le théorème 9.3 qu'il existait toujours une forme normale telle que λ n'apparaissait pas dans la partie gauche des règles de production. Il est également possible, moyennant quelques acrobaties techniques, de le faire disparaître de la partie droite, sans restreindre de façon significative le langage généré.

	1	2	3	4	5	6	7	8
f	$ID\#$	\$	#	F	D'	a'	b'	D
g	I	#	\$	$\$baaabF$	D	a	b	D'
	9	10	11	12	13	14	15	16
f	a	b	$b'D'b'$	$a'a'$	$a'D'$	bDb	aa	aD
g	a'	b'	D	D	D	D'	D'	D'

Il ne nous reste plus qu'à montrer comment nous construisons la solution du problème de correspondance de Post à partir de la dérivation de $baaab$ mentionnée ci-dessus.

Le mécanisme est très simple. On reprend la dérivation, on place le marqueur de tête I (indice 1), puis on remplace chacun des $\rightsquigarrow$ par, alternativement, un # (occurrences impaires : première et troisième occurrence), ou un \$ (occurrence paire : deuxième occurrence).

Les pas pairs sont primés (bDb, $baaab$), les pas impairs (D, $baDb$) conservés inchangés ce qui nous donne :

$$ID\#b'D'b'\$baDb\#b'a'D'b'\$baaabF$$

On arrive alors, en suivant pas à pas le tableau des deux morphismes, à la construction du nombre : 1.11.2.10.16.10.3.7.6.5.7.2.10.9.15.10.4

Ce nombre est solution de l'instance du PCP considérée[2].

On reconnaît facilement dans ce mot la dérivation issue de la grammaire G.

Réciproquement, si le problème de Post a une solution, elle commence impérativement par l'indice 1, se termine par l'indice 4 et permet de reconstruire aisément la dérivation qui générera le mot m pour la grammaire G (nous ne le démontrerons pas).

On constate donc l'équivalence du problème de Post et du problème de reconnaissance pour les langages récursivement énumérables. D'où le résultat :

Théorème 10.2 – Indécidabilité du PCP – *Le problème de correspondance de Post est indécidable.*

10.3.3 Autres résultats sur le PCP

Le problème de correspondance de Post est un problème qui a été, et reste, très étudié.

On s'est posé deux questions principales sur le problème de Post :

1. quelle est l'influence du cardinal de l'alphabet de départ sur la décidabilité du problème ?
2. quelle est l'influence du cardinal de l'alphabet d'arrivée sur la décidabilité du problème ?

La question (2) est relativement facile à résoudre : il est simple de prouver que si l'alphabet d'arrivée est réduit à une seule lettre, le problème est décidable. En effet, nous sommes ramenés à deux listes de la forme : $(a^{p_1}, a^{p_2}, \ldots, a^{p_n})$ et $(a^{q_1}, a^{q_2}, \ldots, a^{q_n})$.

2 Le lecteur ébahi par ce tour de magie est invité à le recommencer lui-même sur un autre exemple afin de mieux comprendre les mécanismes sous-jacents.

On est donc ramené à un problème d'arithmétique élémentaire : déterminer si l'équation

$$\sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n} q_i x_i \tag{10.1}$$

a des solutions entières positives ou nulles, mais non toutes identiquement nulles. La démonstration se fait en trois points :

1. s'il existe un j tel que $p_j = q_j$ alors le problème a trivialement comme solution $x_j = 1$ et tous les autres x_i nuls ;
2. s'il existe un j et un k tels que $p_j > q_j$ et $p_k < q_k$ alors le problème admet comme solution $x_j = q_k - p_k$ et $x_k = p_j - q_j$, tous les autres x_i nuls ;
3. si aucune des deux conditions précédentes n'est vraie alors on a soit (1) $\forall i p_i > q_i$ ou (2) $\forall i p_i < q_i$. L'équation 10.1 étant équivalente à :

$$\sum_{i=1}^{n} (p_i - q_i) x_i = 0 \tag{10.2}$$

Dans le cas (1), $p_i - q_i$ est toujours strictement positif, et strictement négatif dans le cas (2). Donc l'équation admet comme unique solution la solution identiquement nulle. Donc le problème n'est pas soluble.

Le problème de Post ayant un alphabet d'arrivée réduit à une lettre est donc décidable.

Nous allons maintenant prouver que si l'alphabet d'arrivée comprend deux lettres, alors le problème de correspondance est indécidable. Soit une instance (f, g) du problème de Post dont l'alphabet d'arrivée est $\Delta = \{a_1, \ldots, a_p\}$. Codons les lettres de Δ par $h(a_i) = ba^i b$. Considérons maintenant la nouvelle instance du problème de Post (hf, hg). Il s'agit là d'un problème dont l'alphabet d'arrivée est $\{a, b\}$ et qui est équivalent au problème précédent. Donc le problème de Post est indécidable quand l'alphabet d'arrivée contient deux lettres ou plus, car s'il était décidable tout problème de Post le serait également (et nous venons de démontrer le contraire).

Théorème 10.3 – *Le problème de Post est décidable pour un alphabet de départ quelconque ssi l'alphabet d'arrivée est réduit à une lettre.*

Quelle est maintenant l'influence du cardinal de l'alphabet de départ ? Le problème est encore ouvert. On sait seulement qu'il existe un nombre n compris entre 3 et 9 tel que le problème de Post soit décidable lorsque le cardinal de l'alphabet de départ est inférieur strictement à n, et indécidable pour tout autre valeur.

10.3.4 Autres résultats liés au PCP

L'indécidabilité du PCP a une grande importance en théorie des langages. Nous citons ici quelques théorèmes qui en découlent.

Théorème 10.4 – *Il n'existe pas d'algorithme permettant de savoir si deux langages à contexte libre ont une intersection non vide.*

Théorème 10.5 – *Il n'existe pas d'algorithme permettant de déterminer si un langage à contexte libre donné est régulier.*

Théorème 10.6 – *Il n'existe pas d'algorithme permettant de déterminer si un langage à contexte libre donné est égal à un langage régulier donné.*

10.4 Les solutions d'équations diophantiennes

La notion d'équation diophantienne remonte à l'antiquité. Son nom vient du mathématicien grec Diophante. Les équations diophantiennes constituent un des grands problèmes encore ouverts de la théorie arithmétique moderne, alors qu'on les étudie depuis plus de 20 siècles.

10.4.1 Définitions

Définition 10.2 – Ensemble diophantien – *Un ensemble E de n-uplets $(x_1, \ldots, x_n)$ d'entiers positifs ou nuls est dit diophantien si et seulement s'il existe un polynôme $P(x_1, \ldots, x_n, y_1, \ldots, y_m)$ à coefficients entiers positifs ou négatifs tel que :*

$$(x_1, \ldots, x_n) \in E \Leftrightarrow \exists (y_1 \geq 0, \ldots, y_m \geq 0), P(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0 \quad (10.3)$$

Les équations du type 10.3 sont appelées *équations diophantiennes*.

On parle également de *relations diophantiennes* pour désigner les ensembles du type E.

Définition 10.3 – Degré d'un ensemble diophantien – *Le degré d'un ensemble E diophantien est le nombre k tel que k est le plus petit des degrés des polynômes permettant de définir E.*

Définition 10.4 – Dimension d'un ensemble diophantien – *On appelle dimension d de l'ensemble diophantien E le plus petit nombre possible de variables m sur l'ensemble de polynômes P permettant de définir E.*

Un exemple d'ensemble diophantien est :

$$\{x \mid \exists (y, z), x = yz\}$$

En fait, comme nous le verrons plus loin, les équations diophantiennes recouvrent un large ensemble de problèmes mathématiques.

10.4.2 Résultats fondamentaux

Théorème 10.7 – Polynôme décrivant un ensemble diophantien – *Pour tout ensemble diophantien E d'entiers positifs, il existe un polynôme Q décrivant exactement E, c'est-à-dire tel que l'ensemble des valeurs positives de Q, lorsque les variables de Q décrivent $\mathbb{N}$, est exactement E.*

Par définition de E, il existe P tel que

$$x \in E \Leftrightarrow \exists (y_1, \ldots, y_m), P(x, y_1, \ldots, y_m) = 0$$

Il nous suffit alors de poser :

$$Q(x, y_1, \ldots, y_m) = x(1 - (P(x, y_1, \ldots, y_m))^2)$$

Le résultat est évident si l'on a bien en mémoire que les polynômes P et Q ne prennent que des valeurs entières.

Théorème 10.8 – Degré d'un ensemble diophantien – *Le degré d'un ensemble diophantien est inférieur ou égal à 4.*

La démonstration de ce théorème est relativement simple. Elle réside dans l'introduction de variables supplémentaires, que nous utilisons chaque fois qu'une variable v_i apparaît dans P avec un degré supérieur à 2.

On pose ainsi $z_i = v_i^p$. Les termes de la forme v_i^{2p+1} deviennent $v_i z_i^2$ et les termes v_i^{2p} deviennent z_i^2. La relation initiale $P(v_k) = 0$ est alors remplacée par un ensemble de relations :

$$P'(v_k, z_l) = 0, P_i = (z_i - v_i^p) = 0$$

Le polynôme P' est de degré inférieur ou égal à 2 et il est possible d'appliquer la méthode précédente sur tous les polynômes P_i de degré plus grand que 2 pour se ramener finalement à un ensemble de relations de la forme :

$$R_1 = 0, R_2 = 0, \ldots, R_p = 0$$

où les R_i sont tous des polynômes de degré inférieur ou égal à 2. Le polynôme :

$$P \ >>= R_1^2 + R_2^2 + \cdots + R_p^2$$

est un polynôme de degré inférieur ou égal à 4 qui représente le même ensemble que notre polynôme P original. On remarquera que l'opération a finalement consisté à augmenter la dimension de P pour diminuer son degré.

Il faut savoir qu'un théorème presque similaire existe concernant la dimension *i.e.*, on sait qu'il existe un nombre n tel que tout ensemble diophantien est de dimension inférieure à n. On ne connaît pas encore la valeur de n.

Nous en venons maintenant au théorème fondamental concernant les ensembles diophantiens, qui va nous permettre de les mettre en rapport avec la théorie des langages formels et des machines de Turing :

Théorème 10.9 – *Tout ensemble récursivement énumérable est diophantien.*

Nous ne donnerons pas la preuve de ce théorème. On peut trouver des éléments plus élaborés concernant les ensembles diophantiens ainsi que la preuve de ce théorème dans (Davis 1982a) et (Davis 1973).

10.4.3 Le dixième problème de Hilbert

Lors de l'énoncé de l'ensemble de son programme, dont nous avons déjà parlé dans le chapitre consacré aux systèmes formels, le mathématicien David Hilbert énonça le problème suivant : *Trouver un algorithme permettant de déterminer si une équation polynomiale à coefficients entiers admet une solution entière*.

Ce problème resta ouvert jusqu'en 1973 ; le mathématicien américain Davis prouva que ce problème est indécidable. La démonstration de Davis était assez attendue. Les développements de la théorie des équations diophantiennes, ainsi que la théorie des fonctions récursives générales permettaient d'espérer un tel résultat. Nous allons simplement donner une idée de la démonstration d'indécidabilité.

Cette démonstration s'appuie sur le théorème 10.9. Elle se déroule en deux parties :

– on démontre tout d'abord que les ensembles de la forme :

$$S_i = \{x \mid \exists(y_1, y_2, \ldots, y_m), P_i(x, y_1, y_2, \ldots, y_m) = 0\}$$

sont récursivement énumérables ;
– on en déduit qu'il existe un polynôme P_u appelé polynôme universel[3], tel que $x \in S_i$ ssi

$$\exists(y_1, y_2, \ldots, y_m), P_u(i, x, y_1, y_2, \ldots, y_m) = 0$$

– On remarque alors que pour un i_0 particulier il est impossible de savoir si l'équation $P_{i_0} = 0$ possède des solutions positives ou nulles, car si ce problème était décidable, le problème de la reconnaissance des langages récursivement énumérables serait décidable. Ce polynôme P_{i_0} montre que le dixième problème de Hilbert est indécidable.

Il existe quelques résultats supplémentaires intéressants sur les équations diophantiennes. On sait par exemple que tout problème diophantien de degré inférieur ou égal à 2 est décidable, et on sait également que les problèmes de degré 4 sont indécidables. On ne sait encore rien sur les problèmes de degré 3.

10.4.4 Conséquences

L'indécidabilité du problème de Hilbert est un résultat plus important qu'il n'y paraît.

On sait en effet ramener la plupart des problèmes encore ouverts en arithmétique à des problèmes diophantiens. Ainsi l'ensemble des nombres premiers est entièrement décrit par les valeurs positives du polynôme (Jones, Sato, Wada, Wiens) :

$$\begin{aligned}
&Q(a, b, \ldots, x, y, z) = \\
&\quad (k+2)(1 - (wz + h + j - q)^2 \\
&\quad - ((gk + 2g + k + 1)(h + j) + h - z)^2 \\
&\quad - (2n + p + q + z - e)^2 - (16(k+1)^3(k+2)(n+1)^2 + 1 - f^2)^2 \\
&\quad - (e^3(e+2)(a+1)^2 + 1 - o^2)^2 - ((a^2 - 1)y^2 + 1 - x^2)^2 \\
&\quad - (16r^2y^4(a^2 - 1) + 1 - u^2)^2 \\
&\quad - (((a + u^2(u^2 - a))^2 - 1)(n + 4dy)^2 + 1 - (x + cu)^2)^2 \\
&\quad - (n + 1 + v + y)^2 \\
&\quad - ((a^2 - 1)l^2 + 1 - m^2)^2 - (ai + k + 1 - l - i)^2 \\
&\quad - (p + l(a - n - 1) + b(2an + 2a - n^2 - 2n - 2) - m)^2 \\
&\quad - (q + y(a - p - 1) + s(2ap - p^2 - 2p - 2) - x)^2 \\
&\quad - (z + pl(a - p) + t(2ap - p^2 - 1) - pm)^2)
\end{aligned}$$

Toutes les questions encore ouvertes sur les nombres premiers peuvent donc être ramenées à des questions sur les valeurs positives entières du polynôme Q !

On peut également construire de semblables polynômes pour la conjecture de Goldbach[4] ou le grand « théorème » de Fermat.

---

3 On remarquera l'analogie avec la machine de Turing universelle $\mathcal{M}_u$.

4 Tout nombre pair est somme de deux nombres premiers.

10.5 Caractère aléatoire d'un programme

Il est bon de citer les résultats de Gregory Chaitin (Chaitin 1979; Chaitin 1988; Chaitin 1987a; Chaitin 1987b) qui éclairent d'un jour différent les problèmes de calculabilité et d'incomplétude des systèmes formels[5].

10.5.1 Hasard et calculabilité

Le but original des travaux qu'a repris Chaitin était d'étudier la notion de hasard, en particulier dans les séries de nombres. Si l'on considère deux séries de nombres, par exemple :

1	3	5	7	9	11	13	15	17	19
7	2	9	1	19	11	14	3	5	6

La première série de nombre n'est visiblement pas aléatoire, alors que la seconde a été obtenue en tirant au hasard des boules numérotées de 1 à 19 dans un sac et nous paraît clairement aléatoire. Cependant, rien ne nous dit que la première série n'a pas été obtenue de la même façon, car, après tout, les probabilités d'obtention des deux séries sont égales : ce n'est donc pas la provenance qui donne le caractère aléatoire d'une série de nombres. En fait, ce que nous exprimons quand nous disons qu'une série est aléatoire est plutôt lié à l'*incompressibilité* de l'information[6] nécessaire à exprimer la série. On peut aisément formaliser cette idée en codant la série de nombres sous forme binaire et en regardant la taille d'un programme chargé de l'exprimer (taille qui s'exprime également en bits). Dans le premier cas le programme serait simplement :

```
FOR i:=1 TO 10 DO WRITELN(2*i-1);
```

Dans le second cas, nous n'aurions d'autre ressource que de faire écrire au programme l'ensemble des chiffres de la série. La taille en bits du second programme serait donc de l'ordre de la taille en bits de la série qu'il exprime, alors que la taille du premier serait plus petite. Ce caractère serait évidemment renforcé sur des séries plus longues.

La définition de hasard que l'on peut en dégager est la suivante :

Définition 10.5 – Caractère aléatoire d'une série – *Une suite de nombres est aléatoire si le plus petit algorithme capable de la construire est d'une taille équivalente à la taille de la suite.*

L'existence d'un plus petit algorithme semble évidente. En effet, il existe, pour exprimer un nombre N, une infinité de programmes de taille positive et l'on sait que tout ensemble d'entiers naturels admet une borne inférieure. Remarquons que ce *programme minimal* P est lui-même composé d'une séquence aléatoire de bits, car s'il était généré par un autre programme P' de taille plus petite, alors l'exécution de P' générerait P sur lequel il suffirait de continuer l'exécution pour trouver N avec comme seule information initiale P'. Chaitin appelle le nombre de bits du programme minimal capable d'engendrer une série de bits la *complexité* de la série de bits :

---

5 Ces travaux sont étroitement reliés à ceux de A. Kolmogorov (Kolmogorov 1965; Kolmogorov 1968; Li and Vitányi 1990) ou ceux de P. Martin-Lof (Martin-Lof 1966).

6 Cette approche du caractère aléatoire d'une série vient d'ailleurs de la théorie de l'information. Bien entendu, on ne peut négliger les approches classiques pour estimer le caractère aléatoire des séquences de nombres. Voir (Knuth 1981).

Définition 10.6 – Complexité d'une série – *On appelle complexité d'une série la taille en bits du programme minimal qui peut la générer.*

La question que l'on peut maintenant se poser est la suivante : comment démontrer qu'une suite de nombres est aléatoire ? Il est relativement facile de prouver qu'elle ne l'est pas : il suffit de construire un algorithme permettant de l'exprimer plus efficacement. Mais comment trouver effectivement une suite aléatoire ? La seule chose que nous puissions faire est d'écrire un programme T qui serait : « Trouver une série de bits telle que l'on puisse prouver que sa complexité est plus grande que celle de ce programme T[7] ». Il suffit alors de faire s'exécuter le programme T. Il ne peut pas s'arrêter. En effet, s'il trouve une série de complexité supérieure au programme T, alors cela veut dire que le programme T peut la calculer (c'est ce qu'il vient de faire) et donc que la complexité de la série est exactement celle du programme T : le problème n'est pas calculable.

Ce résultat signifie que, dans un système, on ne peut exprimer qu'une série de bits possède une complexité plus grande que le nombre de bits du programme qui l'exprime. Or, on peut regarder un programme et un calculateur comme un système formel dont les axiomes et les règles d'inférence constituent les règles de calcul ; on en déduit le théorème suivant :

Théorème 10.10 – Théorème de Chaitin – *Soit n la complexité (taille en bits) des axiomes et règles d'inférence d'un système formel. Si la complexité du programme T recherchant une série aléatoire est c, on ne peut prouver qu'une série de bits a une complexité supérieure à $n + c$.*

Comme d'autre part, la complexité est définie en terme de hasard, ce théorème prouve que dans un système formel on ne peut prouver qu'un nombre est aléatoire si sa complexité n'est pas inférieure à celle du système formel lui-même. La complexité du système est importante car elle représente la quantité d'information que contient le système et donc la quantité d'information qu'il pourra générer. Les axiomes et règles d'inférence sont aléatoires (ils sont minimaux) et sont les règles qui permettent de tester le caractère aléatoire d'autres nombres (qui codent les formules). Le théorème de Chaitin exprime donc qu'il y a dans tout système formel des nombres (formules) indémontrables car de complexité supérieure à celle du système lui-même. Il donne du théorème de Gödel et des problèmes de calculabilité une approche intuitive liée à la notion d'information contenue dans le système formel.

10.5.2 Probabilité d'arrêt d'un programme

Le but de Chaitin était de construire une série de bits qui soit aléatoire. Il définit alors un nombre Ω qui est la probabilité d'arrêt d'un programme aléatoire pour un ordinateur donné. Le programme est une suite finie de 0 et de 1 qui ont été tirés à pile ou face. Le nombre Ω possède la particularité intéressante d'être aléatoire *i.e.*, de ne pouvoir être exprimé par une séquence de bits plus courte que lui-même.

Il est possible de construire des approximations d'Ω en calculant des nombres Ω_n qui représentent la probabilité pour qu'un programme aléatoire de n bits au plus s'arrête

7 Souvenez-vous de la démonstration du théorème de Gödel : on construit une formule qui exprime sa propre indémontrabilité : ici on construit un programme. Le mécanisme dérive finalement du paradoxe de Berry (le plus petit nombre que l'on peut définir en moins de quinze mots).

en au plus n secondes. Il est alors possible (en théorie) de calculer Ω_n en examinant extensivement combien de programmes n_k de longueur k ($1 \leq k \leq n$) s'arrêtent en n secondes au plus. On a alors :

$$\Omega_n = \sum_{1 \leq k \leq n} 2^{-k} n_k$$

Quand n devient très grand Ω_n tend vers Ω.

Chose amusante, Chaitin, en utilisant les résultats (Jones and Matijasevich 1984) de J. Jones (Université de Calgary) et de Y. Matijasevic (Institut de mathématiques de Léningrad), a construit une famille d'équations diophantiennes qui possèdent la propriété suivante : la k^{e} équation diophantienne de la série admet une infinité de solutions si le k^{e} bit de Ω vaut 1, et n'a qu'un nombre fini de solutions sinon.

10.5.3 Complexité organisée

Le physicien américain Charles Bennett s'est inspiré des travaux de Chaitin pour définir, à coté de la complexité définie par Chaitin qu'il appelle *complexité aléatoire*, une complexité organisée. La complexité organisée d'un objet est le temps nécessaire au programme minimal (au sens du Chaitin) pour calculer cet objet.

Ainsi, alors que la suite des 10^{10} premiers nombres premiers a une complexité aléatoire faible (le programme minimal pour les calculer est petit), sa complexité organisée est forte car le temps de calcul est long.

CHAPITRE 11

La complexité

Jean-Marc Alliot — Thomas Schiex

11.1 Introduction

La théorie de la complexité a pour but de donner un contenu formel à la notion intuitive de difficulté de résolution d'un problème. La théorie de la calculabilité nous a permis d'identifier un certain nombre de problèmes non calculables. Nous sommes, d'autre part, aptes à fournir des algorithmes pour d'autres problèmes. Mais le fait que nous puissions les résoudre théoriquement ne nous permet pas d'affirmer que nous saurons effectuer le calcul en pratique, si, par exemple, le temps d'exécution se révèle prohibitif. La théorie de la complexité permet de dégager plusieurs niveaux de difficulté. C'est une aide très appréciable pour déterminer s'il est raisonnable ou non de se lancer dans la résolution de ce problème, ou s'il faut au contraire chercher à le reformuler pour aboutir à un problème plus simple. Il s'agit d'une théorie encore très ouverte aujourd'hui. Nombre de problèmes, dont certains importants, n'ont pas encore trouvé de réponse.

Avant que le lecteur ne consulte les paragraphes suivants[1], nous lui conseillons de relire rapidement le chapitre consacré aux machines de Turing et spécialement les quelques lignes traitant des machines de Turing non déterministes.

11.2 Définitions fondamentales

11.2.1 Problèmes de décision et langages

De façon informelle, un problème Π est défini par l'ensemble des propriétés que doivent vérifier ses *solutions*. Généralement, ces propriétés font apparaître un ou plusieurs

1 Voici quelques ouvrages traitant de la théorie de la complexité : la référence la plus classique est sans aucun doute (Garey and Johnson 1979). (Salomaa 1989) est assez éclectique, il traite également les problèmes de théorie des langages, de calculabilité, de cryptographie, de réseaux de Pétri. (Papadimitriou 1994) suit une approche similaire, avec des résultats plus récents. Dans (Johnson 1990), le lecteur trouvera un résumé synthétique mais un peu sec des principaux résultats et classes définies. En français, on pourra consulter (Wolper 1991) ou (Barthélemy *et al.* 1992). Ce dernier présente une approche intéressante de la notion de codage raisonnable.

paramètres non spécifiés. Une instance I d'un problème Π est obtenue en spécifiant des valeurs particulières pour tous les paramètres du problème.

Définition 11.1 – *On dira qu'un algorithme résout un problème* Π *s'il peut être appliqué à toute instance* I *de* Π *pour fournir la solution de l'instance* I *en un temps fini.*

Les problèmes qui nous intéresseront sont, pour des raisons de facilité, des problèmes de décisions *i.e.*, des problèmes qui n'ont que deux solutions possibles : la réponse *oui* ou la réponse *non*. Si l'on note D_Π l'ensemble des instances possibles d'un problème *de décision* Π, on peut distinguer le sous-ensemble Y_Π (pour *yes*) des instances pour lesquelles la réponse est *oui* du sous-ensemble N_Π des instances pour lesquelles la réponse est *non*.

Exemple de problème de décision : « un entier positif k étant donné, k est il premier ? »

On définit simplement le complémentaire d'un problème (de décision) :

Définition 11.2 – Complémentaire – *Étant donné un problème de décision* Π*, on définit son complémentaire (noté* Π^c*) par :*

$$D_{\Pi^c} = D_\Pi, Y_{\Pi^c} = N_\Pi, N_{\Pi^c} = Y_\Pi$$

i.e.*, le problème complémentaire* Π^c *d'un problème de décision* Π *est défini par la question opposée de celle définissant* Π*.*

Le complémentaire du problème précédent s'énonce donc : « un entier positif k étant donné, existe-t-il deux entiers $m, n > 1$ tels que $k = m.n$? ».

Se restreindre aux problèmes de décision n'est pas si limitatif qu'il pourrait paraître au premier abord : un problème de recherche pourra se transformer en problème d'existence, un problème d'optimisation en un problème d'existence d'une solution de qualité suffisante. En fait, la raison de cette limitation aux problèmes de décision est qu'il existe une contrepartie naturelle, mais formelle, à un problème de décision, il s'agit du problème de reconnaissance d'un mot dans un langage.

La correspondance entre problèmes de décisions et langages s'effectue au moyen d'une fonction de codage e, permettant de traduire toute instance d'un problème de décision Π en un mot sur un alphabet Σ. L'ensemble des mots de Σ^* est alors partitionné par e et Π en :

- l'ensemble des mots qui ne sont pas des codages d'une instance de Π, qui ne nous intéressent pas ;
- l'ensemble des mots qui résultent du codage d'une instance de N_Π, pour laquelle la réponse est *non* ;
- l'ensemble des mots, noté $L[\Pi, e]$, qui résultent du codage d'une instance de Y_Π, pour laquelle la réponse est *oui*.

La notion informelle d'algorithme permettant de résoudre un problème de décision peut ensuite être ramenée à la notion formelle de programme de machine de Turing (voir chapitre 4) acceptant un mot $x \in \Sigma^*$, x résultant du codage d'une instance I. Plus précisément, nous considérerons des machines de Turing utilisant un alphabet Γ contenant Σ et possédant *deux* états d'arrêt z_Y et z_N. Appliquée à un mot x, une telle machine $\mathcal{M}$ peut :

- s'arrêter dans l'état z_Y, on dira alors que le mot est accepté par la machine. L'ensemble des mots acceptés est noté $L(\mathcal{M})$;

– s'arrêter dans l'état z_N, on dira alors que le mot est rejeté par la machine ;
– ne pas s'arrêter.

Cependant, et pour que la notion de programme de machine de Turing corresponde précisément à la notion d'algorithme qui résout un problème, nous ne considérerons par la suite que des programmes qui s'arrêtent sur toute entrée $x \in \Sigma^*$. Nous ne nous intéresserons donc pas, dans ce chapitre, à des problèmes indécidables.

Définition 11.3 – *On dira qu'une machine de Turing $\mathcal{M}$ résout le problème de décision Π, sous le codage e, si $\mathcal{M}$ s'arrête sur toute entrée $x \in \Sigma^*$ et si l'ensemble des mots acceptés par $\mathcal{M}$ est égal à l'ensemble $L[\Pi, e]$.*

Définition 11.4 – Efficacité d'un algorithme – *Établir l'efficacité d'un algorithme, c'est donner une relation entre le nombre d'opérations effectuées par une machine de Turing $\mathcal{M}$ exécutant l'algorithme A et la taille m de l'argument de la machine de Turing.*

Le problème maintenant est de définir la notion de taille des arguments. En effet, cette taille est dépendante de la fonction de codage et en particulier de l'alphabet utilisé pour le codage d'un argument en un mot. Ainsi le nombre 10 codé en notation décimale à une longueur de 2 caractères, de 4 caractères en numération binaire et de 10 caractères en notation unaire. Toutes les propriétés que nous énoncerons par la suite supposent qu'un codage « raisonnable » est employé. Informellement, il s'agit d'un codage concis, sans redondance, et décodable. Dans le cas des nombres, on peut en donner une définition plus précise[2] :

Définition 11.5 – Codage raisonnable des données – *Le codage des données est raisonnable si les nombres ne sont pas codés sous forme unaire. L'encombrement mémoire doit être de la forme $\log n$. Nous noterons la taille d'un mot codé (sa longueur) $|n|$.*

Ainsi, au sens de la complexité, il est indifférent qu'un nombre soit codé en base 2,3,10 ou 16. En revanche, il *ne doit pas* être codé sous forme unaire. Nous verrons qu'il y a d'excellentes raisons à cela.

Définition 11.6 – Complexité temporelle pour une machine déterministe – *La complexité temporelle d'un programme de machine de Turing déterministe $\mathcal{M}$ est la fonction de n :*

$$T_{\mathcal{M}}(n) = \max_{x \in \Sigma^*, |x|=n} \{k \mid k \textit{ longueur de calcul sur l'entrée } x\}$$

La notion de complexité temporelle est intuitivement pertinente : nous considérons l'ensemble des mots de longueur n et nous regardons quel est celui qui entraîne le calcul le plus long. Nous sommes alors sûrs que tous les autres mots de longueur n seront traités dans un temps inférieur, par définition.

Définition 11.7 – Complexité temporelle pour une machine non déterministe – *La complexité temporelle d'un programme d'une machine de Turing non déterministe $\mathcal{M}$ est la fonction de n :*

$$T_{\mathcal{M}}(n) = \max_{x \in L(\mathcal{M}), |x|=n} \{\min\{k \mid k \textit{ longueur d'un calcul d'acceptation de } x\}\}$$

---

2 Une définition plus large, s'appliquant en sus aux graphes, fonctions et ensembles finis est proposée dans (Garey and Johnson 1979). La notion de codage *sympathique*, plus précise, est présentée dans (Barthélemy *et al.* 1992).

La définition de la complexité temporelle d'un programme d'une machine de Turing non déterministe est plus délicate. On prend le maximum sur l'ensemble des mots de taille n, mais pour chaque mot, on considère que l'on suit le chemin le plus court menant à une acceptation dans l'arbre de résolution de la machine de Turing non déterministe. Il faut noter l'asymétrie, introduite par le non déterminisme, entre acceptation et rejet, le maximum étant calculé sur les mots *acceptés* seulement. Nous y reviendrons.

11.3 Problèmes faciles et intraitables

Dans cette section, nous allons nous intéresser à une première classe de problèmes, celle des problèmes polynomiaux[3].

11.3.1 Problèmes polynomiaux

Définition 11.8 – Problèmes polynomiaux – *L'ensemble des problèmes polynomiaux (ou l'ensemble des langages reconnus en un temps polynomial par une machine de Turing déterministe) est l'ensemble*

$$\mathsf{P} = \{L | L \subset \Sigma^*, \exists \mathcal{M}, L = L(\mathcal{M}), \exists P, \forall x \in L, |x| = n, T_{\mathcal{M}}(x) < P(n)\}$$

où P est une fonction polynomiale et M une machine de Turing déterministe.

Cette classe de problèmes (ou langages) est appelée classe P.

L'ensemble des problèmes[4] polynomiaux est donc l'ensemble des problèmes dont le temps de résolution est borné par un polynôme fonction de la taille des données. On comprend mieux maintenant pourquoi on exige un codage raisonnable, et en particulier, de ne pas utiliser de représentation unaire pour les nombres. Considérons en effet une donnée numérique N pour un problème polynomial codée dans le système décimal, en base 2 et en base 1. Dans le premier cas, la longueur du codage sera de l'ordre de $\log_{10}(N)$, dans le second, de l'ordre de $\log_2(N)$ alors que dans le dernier cas elle sera de l'ordre de N lui-même, qui ne peut pas s'exprimer sous la forme d'un polynôme fonction de $\log_2(N)$ ou de $\log_{10}(N)$. Ainsi, un problème de complexité polynomiale en utilisant un codage en base 1 n'est pas nécessairement polynomial pour des codages en base 2 ou plus, alors que tout problème polynomial en utilisant un codage en base $i \geq 2$ reste polynomial si un codage en une autre base $j \geq 2$ est employé : *l'ensemble des problèmes polynomiaux est invariant par changement de fonction de codage, pourvu que le codage soit un codage raisonnable*.

Il existe une sous-classe des problèmes polynomiaux, les problèmes linéaires (P est une fonction linéaire), qui constitue une classe de problèmes plus simples que ceux de

3 appelés aussi problèmes faciles par certains auteurs.

4 Nous employons indifféremment dans ce chapitre la notion de *langage accepté par une machine de Turing* et la notion de *problème résolu par une machine de Turing* de façon à bien rappeler au lecteur l'interchangeabilité des deux notions. À un problème Π donné, on associe simplement le langage $L[\Pi, e]$ où e est un codage raisonnable.

P. Appartiennent à cette classe des problèmes triviaux comme le calcul de la somme de deux nombres de p chiffres...

On définit également co-P, formé de l'ensemble des problèmes (ou langages) complémentaires de problèmes de P. Une machine de Turing déterministe qui sait dire, en un temps polynomial, si un mot représente, ou non, une instance de Y_{Π}, permet directement de résoudre le problème Π^c en un temps polynomial. Il suffit d'échanger les deux états z_Y et z_N dans son programme. On en déduit donc que :

Théorème 11.1 – P $=$ *co*-P

La classe polynomiale comprend quelques problèmes classiques dont voici des exemples :

- tri d'un ensemble de n nombres ;
- test de planarité d'un graphe[5] ;
- recherche des composantes connexes d'un graphe[6] ;
- recherche de la plus courte chaîne (ou chemin si le graphe est dirigé) simple permettant de se rendre d'un sommet P à un sommet Q dans un graphe dont les arêtes sont valuées positivement[7] ;
- recherche d'une chaîne qui passe par toutes les arêtes d'un graphe[8].

La classe des problèmes polynomiaux est à la fois importante et limitée. Importante parce que nombre de problèmes pratiques indispensables au bon fonctionnement de l'informatique ont des solutions polynomiales, limitée parce que pour nombre de problèmes intéressants, personne n'a réussi à montrer leur caractère polynomial.

11.3.2 Les problèmes prouvés intraitables

L'ensemble des problèmes dont on a prouvé qu'ils étaient calculables mais non polynomiaux est appelé *classe des problèmes prouvés intraitables*[9].

Un problème prouvé intraitable est la théorie formelle de l'addition (théorème de Presburger) : la théorie formelle de l'addition est un sous-système formel de $\mathbb{N}$ comprenant un sous-ensemble du calcul des prédicats et les axiomes de l'addition. Alors que $\mathbb{N}$ (et le calcul des prédicats) est indécidable, ce système est lui décidable, mais intraitable.

11.3.3 Conclusion

Nous avons dégagé plusieurs classes de problèmes, dont les problèmes polynomiaux et les problèmes prouvés intraitables. Mais, il reste tout un ensemble de problèmes que nous ne pouvons pas classer dans P, car on ne connaît pas d'algorithme polynomial

---

5 Un graphe est dit planaire si on peut le dessiner sur une feuille sans que ses arêtes se rencontrent en dehors des sommets.

6 Ensemble de points tel qu'il existe toujours une chaîne permettant de passer d'un point de la composante à un autre. On parle aussi d'ensemble de points reliés entre eux (pas forcément deux à deux).

7 Le problème n'est plus polynomial s'il y a des arêtes valuées négativement. Les algorithmes polynomiaux de calcul d'un plus court chemin ne fournissent un chemin *simple* que si le graphe ne possède pas de circuits absorbants (circuits de longueur négative).

8 Problème dit du cycle eulérien, car Euler fut le premier à le résoudre. Une instance fameuse de ce problème est le « problème des ponts de Kœnigsberg » (Berge 1970).

9 *Intractable* en anglais.

permettant de les résoudre, et que nous ne pouvons pas non plus considérer comme intraitables, car on n'a pas prouvé qu'il n'existe pas un algorithme polynomial permettant de les résoudre. L'introduction de la classe NP et de la notion de problèmes NP-complets permet de pallier, en partie, l'absence de ces résultats.

11.4 Résultats généraux sur les problèmes NP

11.4.1 Problèmes NP

Définition 11.9 – Problèmes NP – *On définit l'ensemble des problèmes non-déterministes polynomiaux (ou ensemble des langages reconnus en un temps polynomial par une machine de Turing non déterministe) par :*

$$\mathsf{NP} = \{L | L \subset \Sigma^*, \exists \mathcal{M}, L = L(\mathcal{M}), \exists P, \forall x \in L, |x| = n, T_{\mathcal{M}}(x) < P(n)\}$$

où P est une fonction polynomiale et $\mathcal{M}$ une machine de Turing non-déterministe.

Cette classe de problèmes est appelée classe NP.

La différence avec la définition de la classe P est que la machine $\mathcal{M}$ est *non-déterministe*. Les problèmes de NP corespondent bien à la notion de casse-tête : il est possible de construire une « solution » en un temps polynomial, mais tout le problème est de trouver cette « solution » (ou plus précisément, un calcul d'acceptation, qui montre qu'il y a une solution). On caractérise donc les problèmes de NP par cette propriété : ce sont les problèmes de décision dont on peut facilement (en un temps polynomial) vérifier, sur une machine déterministe, qu'une solution potentielle est effectivement une solution *i.e.*, qu'un calcul d'acceptation est bien un calcul d'acceptation.

Une machine déterministe étant un cas particulier de machine non déterministe, il ressort de la définition des deux classes P et NP que $\mathsf{P} \subseteq \mathsf{NP}$. *On ne sait pas aujourd'hui si l'inclusion est stricte ou non*. Il s'agit probablement là d'un des problèmes les plus importants de l'informatique théorique moderne, car la classe des problèmes NP est très importante.

On définit, comme pour P, la classe co-NP formée des complémentaires des problèmes (ou langages) de NP. Cependant, la définition de la complexité temporelle d'une machine non-déterministe fait apparaître une asymétrie entre les instances sur lesquelles on répond *oui*, et les autres : on sait que pour les instances pour lesquelles on répond *oui*, il existe un calcul de longueur polynomiale qui permet de répondre *oui* (d'accepter le codage de l'instance), mais on ne sait rien sur les autres. À l'heure actuelle, on ne sait donc pas si $\mathsf{NP} = \text{co-NP}$. En fait, la conjecture $\mathsf{NP} \neq \text{co-NP}$ est plus forte encore que la conjecture $\mathsf{NP} \neq \mathsf{P}$, car du fait $\mathsf{P} = \text{co-P}$, $\mathsf{NP} \neq \text{co-NP}$ impliquerait $\mathsf{NP} \neq \mathsf{P}$ (en effet, si $\mathsf{P} = \mathsf{NP}$, alors $\mathsf{P} = \text{co-P} = \text{co-NP} = \mathsf{NP}$).

11.4.2 Problèmes NP-complets

Définition 11.10 – Transformation polynomiale – *Une* transformation polynomiale[10] *d'un language $L_1 \subset \Sigma_1^*$ en un language $L_2 \subset \Sigma_2^*$ est une fonction $f : \Sigma_1^* \to \Sigma_2^*$ qui satisfait les deux conditions suivantes :*

1. *f peut être calculée en un temps polynomial par une machine de Turing déterministe ;*
2. *pour tout $x \in \Sigma_1^*$, $x \in L_1$ si et seulement si $f(x) \in L_2$.*

Définition 11.11 – Réductibilité – *Un langage $L_1 \subset \Sigma_1^*$ est polynomialement réductible à un langage $L_2 \subset \Sigma_2^*$ ssi il existe une transformation polynomiale de L_1 en L_2. On note alors $L_1 \propto_p L_2$.*

Notons que $\propto_p$ est un préordre (une relation réflexive et transitive). Nous dirons que deux langages L_1 et L_2 sont *polynomialement équivalents* quand $L_1 \propto_p L_2$ et $L_2 \propto_p L_1$. La relation ainsi définie est une relation d'équivalence dont les classes d'équivalence sont ordonnées par $\propto_p$. Les problèmes NP-complets forment alors la classe d'équivalence des problèmes les plus difficiles de NP :

Définition 11.12 – Problème NP-complet – *Soit un langage L. L est dit NP-complet ssi*
– $L \in \mathsf{NP}$
– $\forall L' \in \mathsf{NP}, L' \propto_p L$

Les langages NP-complets[11] représentent donc les problèmes NP les plus difficiles à résoudre puisque tout problème de NP peut se ramener à un problème NP-complet en un temps polynomial. Nous avons d'autre part le théorème suivant :

Théorème 11.2 – *Soit L_0 un langage NP-complet alors :*

$$L_0 \in \mathsf{P} \Leftrightarrow \mathsf{P} = \mathsf{NP}$$

Il est clair que si $\mathsf{P} = \mathsf{NP}$ alors L_0 est dans P.

Réciproquement, soit L_0 NP-complet et $L_0 \in \mathsf{P}$. Nous savons qu'il existe une machine de Turing déterministe $\mathcal{M}$ qui accepte L_0 en un temps borné par un polynôme P. Pour tout langage de L de NP, nous savons qu'il existe f telle que $f(L) = L_0$ et f calculable en un temps borné par un polynôme Q. Soit $x \in L$, nous pouvons alors :

1. calculer $f(x)$ en un temps $Q(|x|)$;
2. ramener la tête de lecture sur le premier symbole en un temps $|f(x)|$;
3. décider si $f(x) \in L_0$ en appliquant $\mathcal{M}$ en un temps $P(|f(x)|)$.

Le temps total de calcul T est

$$T \leq Q(|x|) + |f(x)| + P(|f(x)|)$$

10 Aussi appelée *many-one reduction* dans certaines références en langue anglaise.

11 La catégorie de problèmes qui ne vérifient pas la première partie de la définition (qui ne sont pas forcément dans NP) sont dits (ce ne sont pas les seuls) NP-difficiles, ou NP-durs.

mais nous savons d'autre part que : $|f(x)| \leq |x| + Q(|x|)$ car la machine de Turing en $Q(|x|)$ opérations n'a pas pu écrire plus de $Q(|x|)$ symboles. On a :

$$T \leq 2Q(|x|) + |x| + P(Q(|x|) + |x|)$$

Donc il existe un polynôme R tel que :

$$T \leq R(|x|)$$

Donc tout problème de NP est soluble en un temps polynomial, donc NP = P.

Ce théorème est un théorème capital en théorie de la complexité. Il signifie en effet que, si nous sommes capables de trouver un algorithme polynomial pour un problème NP-complet, alors tous les problèmes NP-complets seront solubles en un temps polynomial. De nombreux mathématiciens et informaticiens travaillent sur le sujet depuis une vingtaine d'années maintenant, et il semble probable, bien que non démontré, que NP $\neq$ P.

Théorème 11.3 – *Soit L_0 NP-complet. Si $L_0 \propto_p L_1$ et $L_1 \in$ NP, L_1 est NP-complet.*

Il nous suffit de démontrer que pour tout $L \in$ NP, $L \propto_p L_1$. Nous savons que L_0 est NP-complet, donc pour tout L de NP, il existe une transformation polynomiale f telle que $f(L) = L_0$. D'autre part, puisque nous avons $L_0 \propto_p L_1$, il existe une transformation polynomiale g telle que $g(L_0) = L_1$. Donc la transformation $h = g \circ f$ est polynomiale et vérifie :

$$h(L) = L_1$$

Q.E.D

Ce théorème est également fondamental, car une fois connu un problème L_0, NP-complet, il suffit pour démontrer qu'un problème L est NP-complet de montrer qu'il est dans NP[12] et de trouver une transformation polynomiale réduisant L_0 à L. En fait, il s'agit là du même type de démonstration que nous avons utilisé en théorie de la calculabilité : nous ramenons toujours un problème nouveau à un problème connu : les démonstrations directes sont rarissimes.

Nous avons donc dégagé une nouvelle classe de problèmes et nous commençons même à avoir des outils pour les étudier. Il nous reste encore une chose bien difficile à faire : trouver un problème qui appartienne effectivement à la classe des problèmes NP-complets. Ce problème ne fut résolu qu'en 1971. C'est l'objet du théorème de Cook.

11.4.3 Le théorème de Cook

Nous rappelons brièvement la définition de la satisfiabilité d'une formule sous forme conjonctive normale.

Définition 11.13 – Satisfiabilité d'une formule – *Soit une formule F du calcul propositionnel sous forme conjonctive normale utilisant les variables propositionnelles $(x_1 \ldots x_n)$.*

Résoudre le problème de la satisfiabilité de F consiste à déterminer s'il existe une interprétation des x_i telle que F soit satisfaite (F vraie). Ce problème est appelé problème SAT (ou langage SAT)[13].

---

12 Cette propriété est souvent simple à établir. Il existe cependant quelques cas difficiles : le problème de satisfiabilité dans le système modal propositionnel S_5 par exemple (cf. (Ladner 1977)), qui est NP-complet.

13 Certains auteurs réservent le terme de SAT au problème de satisfiabilité d'une formule quelconque (habituellement appelé GENERAL SAT). Le problème de satisfiabilité d'un ensemble de clauses est alors appelé CONSAT.

Toutes les notions apparaissant dans cette définition ont été introduites en logique, nous ne revenons pas dessus.

Théorème 11.4 – Théorème de Cook – *Le langage SAT est* NP*-complet.*

La démonstration de ce théorème est assez longue, mais aussi très instructive. Elle ne présente aucune difficulté théorique majeure. Le lecteur pressé ou dégoûté par les mathématiques peut s'en dispenser sans préjudice.

Il nous suffit de prouver que pour tout langage L de NP, on a $L \propto_p$ SAT[14].

Le principe de la méthode est simple. Pour tout mot m, nous allons construire un algorithme polynomial transformant m en une formule F qui sera satisfiable ssi il existe un calcul d'acceptation de m en un temps polynomial $P(n)$ (avec $n = |m|$) par une machine de Turing $\mathcal{M}$ non déterministe. Nous aurons ainsi prouvé $L \propto_p$ SAT.

Nous noterons $i, 0 \leq i \leq p$ les états de la machine de Turing (0 sera l'état de démarrage et p correspondra à l'état d'acceptation z_Y), et $s_j, 1 \leq j \leq q$ les symboles de l'alphabet Σ (le symbole B sera noté B). Les cases de la bande seront désignées par la lettre c et seront numérotées par des entiers relatifs.

Soit un mot $m = (a_1 a_2 \dots a_n) \in \Sigma^*$. Posons $|m| = n$. Le mot m est dans $L \in$ NP ssi il existe une séquence de calcul de $\mathcal{M}$ qui aboutit à un succès en un temps $P(n)$[15]. Nous voyons donc que nous n'aurons à considérer que des instants t tels que :

$$0 \leq t \leq P(n)$$

D'autre part, la machine de Turing étant dans une configuration valide au départ, la tête de la machine de Turing pointe sur la case 0. Comme elle ne pourra pas effectuer plus de $P(n)$ déplacements en $P(n)$ étapes, nous n'avons à considérer que les cases c telles que :

$$-P(n) \leq c \leq P(n)$$

Les deux conditions que nous venons d'énoncer limitent l'espace de notre problème. Il nous reste à énoncer les conditions liées au fonctionnement de la machine de Turing non déterministe et à les traduire en formule logique.

Nous allons utiliser comme atomes de notre formule l'ensemble de variables logiques :

$$[c, t, s, z]$$

Cette variable prend la valeur vraie si la case c est à l'instant t dans l'état (s, z). On définit l'état d'une case c à l'instant t comme étant :

(s,-1) si c contient le symbole s à t et que la machine de Turing pointe sur une autre case que c.

(s,z) si c contient le symbole s à l'instant t et que la machine de Turing pointe sur la case c et est dans l'état z, $0 \leq z \leq p$.

---

14 Il faut aussi démontrer que SAT est dans NP : il suffit de considérer une machine de Turing non-déterministe qui à chaque étape donne à une variable non encore affectée la valeur 0 ou la valeur 1 et évalue la formule lorsque toutes les variables ont été affectées.

15 En toute rigueur, elle aboutit en un temps inférieur ou égal à $P(n)$. Mais on peut toujours supposer que le calcul ne se termine qu'à l'instant $P(n)$ en imposant à la machine de Turing de ne rien faire dès qu'elle a terminé le calcul effectif, jusqu'à $P(n)$, instant de l'arrêt.

Le nombre de variables est égal au nombre de cases $(2P(n)+1)$, multiplié par le nombre d'instants possibles $(P(n)+1)$, multiplié par le nombre de lettres de l'alphabet (q) et le nombre d'états de la machine de Turing plus un, soit $(p+2)$. Le nombre de variables est donc de l'ordre[16] de $(P(n))^2$.

Nous sommes maintenant en mesure de traduire la suite des conditions qui exprime qu'un mot $m \in L$ ssi il existe une séquence de calcul de la machine de Turing non-déterministe $\mathcal{M}$ qui reconnaît m.

1. À l'instant initial, $\mathcal{M}$ est dans la configuration initiale définie par m. Donc les cases $0, 1, \ldots, n-1$ sont occupées par les symboles $a_1, a_2, \ldots, a_n$ et les cases $-P(n), -P(n+1), \ldots, -1$ et $n, n+1, \ldots, P(n)$ sont occupées par le symbole blanc B. D'autre part la machine est dans l'état initial z_0, noté 0. On peut traduire cet état par la formule :

$$F_1 = (\bigwedge_{i=-P(n)}^{-1} [i, 0, B, -1]) \wedge [0, 0, a_1, 0] \wedge (\bigwedge_{i=1}^{n-1} [i, 0, a_{i+1}, -1]) \wedge (\bigwedge_{i=n}^{P(n)} [i, 0, B, -1])$$

   Cette formule peut être écrite en un nombre de pas de l'ordre de $P(n)$[17].

2. À l'instant $P(n)$, la machine de Turing a terminé son calcul. Elle est alors dans l'état final z_Y (noté p) et pointe sur une case quelconque contenant un symbole quelconque. On peut donc le traduire par :

$$F_2 = \bigvee_{i=-P(n)}^{P(n)} \bigvee_{j=1}^{q} [i, P(n), s_j, p]$$

   Cette formule peut être écrite en un nombre de pas de l'ordre de $P(n)$[18] (q est une constante).

Nous avons avec les conditions (1) et (2) traduit l'existence d'un état initial défini par m et d'un état final d'acceptation. Nous allons maintenant exprimer que le changement d'état d'une case dépend de la fonction de transition δ[19] de la machine de Turing $\mathcal{M}$.

3. Considérons une case c dans un état (s, z) avec $0 \leq z \leq p$. Que devient cette case à l'instant $t+1$? Elle peut :
   (a) passer dans l'état (s', z') si $(z', s', I) \in \delta(z, s)$;
   (b) passer dans l'état $(s >>, -1)$ si $(z >>, s >>, G) \in \delta(z, s)$ ou $(z >>, s >>, D) \in \delta(z, s)$.

   Il faut bien se rappeler que la machine de Turing est non déterministe. Donc à chaque étape, elle a plusieurs possibilités d'action (la fonction δ a pour image un ensemble de triplets et non un seul triplet). Ainsi non seulement nous avons deux possibilités principales (I) ou (G ou D), mais pour chacune de ces possibilités, il peut y avoir plusieurs alternatives. Cette condition peut se traduire par la formule :

$$f_3(c, t) = [c, t, s, z] \rightarrow (\bigvee_{i \in \mathcal{D}_1} [c, t+1, s(i), z(i)]) \vee (\bigvee_{j \in \mathcal{D}_2} [c, t+1, s(j), -1])$$

---

16 Nous ne nous intéressons qu'à la complexité en n.

17 En fait, en toute rigueur, la notation des variables étant dyadique, il faut prendre $(\log n)P(n)$ comme majorant ou, pour être tranquille, $nP(n)$. Mais cela ne change en rien le caractère polynomial de l'expression.

18 La note précédente s'applique et continuera de s'appliquer pour les étapes suivantes.

19 Pour le lecteur qui aurait oublié ce qu'est la fonction de transition δ, un petit retour à la définition des machines de Turing non-déterministes est conseillé.

Le domaine $\mathcal{D}_1$ décrit le sous-ensemble de $\delta(z,s)$ comprenant les actions qui laissent la machine immobile, et le domaine $\mathcal{D}_2$ le sous-ensemble des actions qui déplacent la machine à droite ou à gauche. La taille d'un programme donné d'une machine de Turing étant finie, elle est bornée par une constante C, donc la formule f_3 précédente peut être écrite en au plus C pas. D'autre part, nous devons écrire une telle formule pour tous les c et tous les t. La formule F_3 s'écrit donc finalement :

$$F_3 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_3(c,t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

4. Considérons maintenant une case c dans l'état $(s,-1)$ à l'instant t. À l'instant $t+1$, cette case peut :
   (a) rester dans l'état $(s,-1)$;
   (b) passer dans l'état (s,z) si la machine de Turing est à l'instant t dans un état z' et pointe sur la case $c+1$ qui contient un symbole s' tel que : $(z,s,G)\in\delta(z',s')$;
   (c) passer dans l'état (s,z) si la machine de Turing est à l'instant t dans un état z' et pointe sur la case $c-1$ qui contient un symbole s' tel que : $(z,s,D)\in\delta(z',s')$.

   Tout cela peut se traduire par la formule :

$$\begin{aligned} f_4(c,t) = {} & [c,t,s,-1]\rightarrow \\ & [c,t+1,s,-1]\vee \\ & (\bigvee_{i\in\mathcal{D}_1}([c,t+1,s,z(i)]\wedge[c+1,t,s'(i),z'(i)]))\vee \\ & (\bigvee_{j\in\mathcal{D}_2}([c,t+1,s,z(j)]\wedge[c+1,t,s'(j),z'(j)])) \end{aligned}$$

   Le domaine $\mathcal{D}_1$ recouvre l'ensemble des éléments du graphe de δ tels que $\exists z,z'$, $\exists s,s',(z,s,G)\in\delta(z',s')$ et $\mathcal{D}_2$ recouvre l'ensemble des éléments du graphe de δ tels que $\exists z,z',s,s',(z,s,D)\in\delta(z',s')$. Comme précédemment nous pouvons établir que la formule f_4 peut s'écrire en au plus C pas avec C constant. Nous devons là aussi écrire une telle formule pour tous les c et tous les t. La formule F_4 s'écrit donc finalement :

$$F_4 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_4(c,t)$$

   Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

Les deux conditions précédentes sont insuffisantes. Il nous faut aussi exprimer que l'on n'a pas le droit, dans une séquence de calcul, de lire deux cases à la fois, comme le permettent les équations précédentes. Ce sont les conditions que nous allons rédiger maintenant.

5. Nous allons tout d'abord exprimer que pour une case c prise à un instant t, il existe exactement un (s,z) tel que c soit dans l'état (s,z) (z pouvant valoir -1) à l'instant t. Nous voulons donc exprimer un « ou » exclusif sur les variables de la forme $[c,t,s_i,j]$, car nous voulons avoir $[c,t,s_1,-1]\vee[c,t,s_1,0]\vee\ldots\vee[c,t,s_q,p]$ mais nous ne voulons pas avoir simultanément deux $[c,t,s_i,j]$. Rappelons qu'en logique, le ou exclusif sur A,B,C se traduit par $(A\vee B\vee C)\wedge((\neg A\vee\neg B)\wedge(\neg B\vee\neg C)\wedge(\neg A\vee\neg C))$. Cette formule

s'étend aisément à un nombre quelconque de termes. Dans notre cas, la formule que nous obtenons est donc :

$$f_5(c,t) = (\bigvee_{\substack{1\leq i\leq q\\ -1\leq j\leq p}} [c,t,s_i,j]) \wedge (\bigwedge_{\substack{1\leq i,k\leq q\\ -1\leq j,l\leq p}} (\neg[c,t,s_i,j] \vee \neg[c,t,s_k,l]))$$

Cette formule a une taille bornée par une constante (p et q sont constants). D'autre part, nous devons écrire une telle équation pour toutes les cases c à tous les instants t. La formule définitive est donc :

$$F_5 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_5(c,t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

6. Nous voulons maintenant exprimer que pour un t fixé, il existe une case c contenant un symbole quelconque, telle que c est dans un des états $i, 0 \leq i \leq p$, c'est-à-dire un état valide (différent de -1) de la machine de Turing. Nous pouvons l'exprimer par :

$$f_6(t) = \bigvee_{\substack{-P(n)\leq c\leq P(n)\\ 1\leq i\leq q, 0\leq j\leq p}} [c,t,s_i,j]$$

Cette formule contient un nombre d'éléments de l'ordre de $P(n)$. Nous devons d'autre part écrire cette formule pour tous les instants t. Donc, la formule définitive est :

$$F_6 = \bigwedge_{0\leq t\leq P(n)} f_6(t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$ en tout.

7. Il nous reste maintenant à exprimer le fait que, pour un t donné, une seule case c peut être dans un état (s,z) avec z différent de -1. Donc si une case est dans un état (s,z), $z \neq -1$ à t toutes les autres sont dans l'état $(s',-1)$ au même instant. Ceci peut s'écrire :

$$f_7(c,t,s,z) = [c,t,s,z] \rightarrow \bigwedge_{\substack{-P(n)\leq c'\leq P(n)\\ c'\neq c}} \bigvee_{1\leq i\leq q} [c',t,s_i,-1]$$

Cette formule contient un nombre d'éléments de l'ordre de $P(n)$. Nous devons d'autre part écrire cette formule pour tous les instants t, toutes les cases c, tous les symboles s et tous les états z différents de -1. Donc, la formule définitive est :

$$F_7 = \bigwedge_{\substack{0\leq t\leq P(n), 0\leq z\leq p\\ -P(n)\leq c\leq P(n), 1\leq s\leq q}} f_7(c,t,s,z)$$

Nous avons là une opération de construction qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^3$ en tout $(P(n) \times (P(n))^2)$.

Nous voici au bout de nos peines. La formule :

$$F = \bigwedge_{i=1}^{7} F_i$$

est satisfiable ssi il existe un calcul d'acceptation de longueur polynomiale du mot m par la machine $\mathcal{M}$. D'autre part, la transformation du problème de reconnaissance de m en la formule F s'effectue en un nombre de pas de l'ordre de $(P(n))^3$ et est donc polynomiale. Donc tout langage $L \in \mathsf{NP}$ est réductible au problème consistant à savoir si une formule est satisfiable. Donc SAT est NP-complet.

11.4.4 Exemples de problèmes NP-complets

La classe des problèmes NP-complets est très importante et nous ne citerons que quelques-uns de ses éléments, les plus importants ou les plus intéressants. Nous ne donnons pas toujours les démonstrations, mais nous essayons de les indiquer chaque fois qu'elles ne sont pas trop longues.

Le problème 3-SAT : Il s'agit du problème SAT dans lequel on impose à chacune des clauses de ne comporter au plus qu'une disjonction de trois littéraux.
Il nous suffit, pour démontrer que ce problème est NP-complet, de démontrer que le problème SAT est polynomialement réductible à 3-SAT.
La démonstration est là encore très simple. Il suffit de remarquer que toute clause :

$$x_1 \vee x_2 \vee \ldots \vee x_n$$

peut se mettre sous la forme :

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge \ldots \wedge (x_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (x_{n-1} \vee x_n \vee \neg y_{n-3})$$

Donc le problème SAT est polynomialement réductible à 3-SAT. Donc 3-SAT est NP-complet.
Signalons d'autre part que tous les problèmes n-SAT avec $n \geq 3$ sont NP-complets, mais que le problème 2-SAT est lui P[20].

Le problème du sac à dos : Supposons que vous ayez un sac à dos susceptible de porter une charge m, et que vous disposiez d'un ensemble d'objets de poids $p_i, 1 \leq i \leq n$. Votre problème est de montrer qu'il existe un sous-ensemble de ces objets tel que la somme de leur poids soit exactement égale à la capacité de votre sac. C'est le problème du sac à dos[21].
Mathématiquement, on peut exprimer la chose par :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut démontrer que ce problème est NP-complet en montrant que 3-SAT lui est polynomialement réductible (voir (Salomaa 1989; Garey and Johnson 1979)).

---

20 On ne peut donc pas réduire polynomialement SAT à 2-SAT.

21 Signalons la désignation anglaise KNAPSACK que l'on trouve encore dans certaines traductions.

Partition d'un ensemble de nombres : On considère un ensemble formé de n entiers positifs $A = \{a_1, \ldots, a_n\}$. On cherche à partager cet ensemble en deux sous-ensembles tels que les sommes des éléments de chacun des sous-ensembles soient égales. Mathématiquement, on cherche à montrer l'existence d'un sous-ensemble $J \subset A$ tel que :

$$\sum_{a_i \in J} a_i = \sum_{a_i \in A-J} a_i$$

On peut démontrer que ce problème est NP-complet en montrant que 3-SAT lui est polynomialement réductible.

Existence d'un stable dans un graphe : Soit un graphe $G = (X, U)$, X représente l'ensemble des sommets du graphe et U l'ensemble des arcs.

On cherche à savoir s'il existe un sous-ensemble $X' \subset X$ de cardinal m tel que : $\forall x, y \in X', (x, y) \notin U$.

Ce problème est appelé *problème de recherche d'un stable* pour un graphe G.

On démontre que ce problème est NP-complet en prouvant que SAT lui est polynomialement réductible. Voici une idée de la démonstration. Pour une formule de SAT :

$$F = C_1 \wedge C_2 \wedge \ldots C_m$$

et

$$C_i = u_{j_i} \vee \ldots \vee u_{j_{k(i)}}$$

on construit le graphe $G = (V, E)$ (V pour *« vertex »*, ensemble des sommets et E pour *« edges »*, ensemble des arcs) défini par :

$$V = \{(i, u_j) | u_j \text{ apparaît dans la clause } C_i\}$$

$$E = \{((i, u_j), (i', u_{j'})) | (i = i') \vee (u_j = \neg u_{j'})\}$$

Le lecteur curieux pourra vérifier que F est satisfiable si (V, E) possède un stable de cardinal m et que la construction du graphe prend un temps polynomial.

Existence d'un transversal dans un graphe : Soit un graphe $G = (X, U)$. On dit que X' est un transversal de G si $X' \subset X$ et chaque arc de G a un sommet dans X'.

Le problème du transversal est trivialement équivalent à celui du stable. En effet, Y est un stable de $G = (X, U)$ ssi $X - Y$ est un transversal de G.

Existence d'un cycle hamiltonien dans un graphe : Exhiber un cycle hamiltonien de cardinal m dans un graphe G consiste à trouver une chaîne passant une et une seule fois par chaque sommet d'un sous-ensemble X' du graphe, avec X' de cardinal m.

On montre que le problème du transversal est polynomialement réductible à celui du cycle hamiltonien. La démonstration est relativement complexe.

Existence d'un circuit hamiltonien dans un graphe : Trouver un circuit hamiltonien de cardinal m consiste à trouver un chemin passant une et une seule fois par chaque sommet d'un sous-ensemble X' de X dans un graphe (X, U) orienté.

Le problème du circuit hamiltonien est polynomialement réductible à celui du cycle hamiltonien.

On commence par construire un graphe G' non orienté contenant 3 fois plus de points que G, car à chaque point de G on fait correspondre trois points x_1, x_2, x_3.

Pour chaque point x de G, on connecte sur x_1 dans G' les arcs qui arrivent sur x dans G, on connecte sur x_3 les arcs qui partent de x dans G et on crée pour chaque triplet x_1, x_2, x_3 deux arcs : $(x_1, x_2), (x_2, x_3)$.

Le graphe ainsi construit est tel que l'existence d'un circuit hamiltonien dans G est équivalent à l'existence d'un cycle hamiltonien dans G'.

Le problème du voyageur de commerce : Ce problème[22] est un des plus célèbres problèmes de la théorie de la complexité. Nous le retrouverons lorsque nous parlerons des schémas d'approximation.

Imaginons un voyageur de commerce. Il doit passer une et une seule fois dans chaque ville parmi un ensemble de n villes. Chaque ville i est séparée de la ville j par une distance d_{ij}. Quel chemin doit-il suivre afin de parcourir le moins de distance ?

Le problème du voyageur de commerce est un problème appartenant à la classe des problèmes d'optimisation. Un des problèmes de décision qui lui correspond consistera à montrer l'existence d'un circuit dont la longueur est inférieure ou égale à une constante k. Le problème est NP-complet.

On peut aisément le montrer en prouvant que le problème du cycle hamiltonien lui est polynomialement réductible. Pour ce faire, il suffit pour un graphe $G = (X, U)$ donné de construire le problème du voyageur de commerce en posant :
– $d_{ij} = 1$ si (i, j) est une arête de G
– $d_{ij} = 2$ si (i, j) n'est pas une arête de G.
S'il existe une solution au problème du voyageur de commerce pour $n = k$ alors on sait qu'il existe un cycle hamiltonien dans G de cardinal k et réciproquement.

Planification de tâches indépendantes : On dispose de m machines identiques qui ne peuvent exécuter qu'une seule tâche à la fois et l'on doit faire exécuter n tâches, chaque tâche i durant un temps t_i. Le problème consiste à montrer qu'il existe une affectation des tâches à chaque machine telle que le travail soit terminé avant un temps T fixé. Pour démontrer que ce problème est NP-complet, on peut démontrer que le problème du sac à dos lui est polynomialement réductible.

Il existe plusieurs variantes de ce problème. On peut ainsi exiger que certaines des tâches soient effectuées dans un certain ordre, ou que certaines machines ne puissent effectuer que certaines tâches. Toutes ces variantes sont NP-complètes.

Il faut toujours se méfier, lorsque l'on examine un problème, de trop se fier à une ressemblance. Ainsi, le problème consistant à montrer qu'il existe une chaîne simple reliant deux sommets donnés, et de longueur *inférieure* à k, dans un graphe dont les arêtes sont valuées positivement, est un problème polynomial. En revanche, l'existence d'une chaîne simple de longueur *supérieure* à k dans le même type de graphe définit un problème NP-complet. De même pour 2-SAT et 3-SAT. Les exemples de ce type sont nombreux. (Garey and Johnson 1979) présente un large catalogue de problèmes NP-complets (et autres).

---

22 Le problème du voyageur de commerce a de nombreuses applications pratiques sous diverses formes : calcul de routage de circuits imprimés, calcul de réseaux de communication ou de distribution ...

11.4.5 Preuve de NP-complétude

Nous donnons ici deux exemples simples montrant comment démontrer le caractère NP-complet d'un problème à partir d'un problème connu comme étant NP-complet.

– Le premier type de preuve, dite par restriction, consiste à montrer qu'une restriction du problème considéré est déjà connue comme étant un problème NP-complet (il faudrait aussi montrer que le problème est dans NP). Considérons par exemple le problème du sac à dos :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut se restreindre au cas où $m = \frac{1}{2} \sum_{i=1}^{n} p_i$. On a alors :

$$\sum_{i \in J} p_i = \frac{1}{2} \sum_{i=1}^{n} p_i$$

soit, en multipliant par 2 et éliminant à gauche et à droite les termes identiques :

$$\sum_{J} p_i = \sum_{\overline{J}} p_i$$

On reconnaît alors le problème de partition. Si l'on sait que ce problème est NP-complet, alors notre problème de sac à dos l'est également (s'il est dans NP).

– Le second type de preuve, dit par transformation polynomiale, a déjà été évoqué. Considérons à nouveau le problème du sac à dos, que nous supposerons NP-complet :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut, à partir de ce problème considérer le problème de partition de l'ensemble A des p_i auquel on adjoint un $p_{n+1} = 2m - \sum_{i=1}^{n} p_i$:

$$\sum_{p_i \in J} p_i = \sum_{p_i \in A - J} p_i$$

Le terme p_{n+1} est soit dans J, soit dans $A - J$. On supposera, sans perte de généralité qu'il est dans $A - J$. On a alors :

$$\sum_{p_i \in J} p_i = (\sum_{p_i \in A - J - \{p_{n+1}\}} p_i) + 2m - \sum_{i=1}^{n} p_i$$

soit :

$$\sum_{p_i \in J} p_i = m$$

Ce problème a une solution ssi le problème de sac à dos initial a une solution. D'autre part, il est simple de montrer que le passage du problème de sac à dos initial au problème de partition final peut s'effectuer en un temps polynomial. On a donc exhibé une transformation polynomiale du premier problème dans le second : si le problème du sac à dos est NP-complet, le problème de partition l'est également.

11.5 Schémas d'approximation

Lorsque l'on sait qu'un des problèmes de décision associés à un problème d'optimisation est NP-complet, il n'y a aucune chance, dans l'état actuel des connaissances[23], de pouvoir résoudre ce problème d'optimisation, *dans le cas général*, de façon efficace.

On peut alors chercher des algorithmes qui, sans résoudre ce problème de façon optimale, garantissent de trouver une solution dans un temps polynomial et garantissent également que l'écart avec la solution optimale peut être majoré.

11.5.1 Principes généraux

Définition 11.14 – ε-approximation d'un problème Π – *Soit un problème Π résolu de façon optimale par l'algorithme OPT pour toute donnée x.*

L'algorithme A_ε est une ε-approximation résolvant Π ssi :

$$\forall x, \left| \frac{A_\varepsilon(x) - OPT(x)}{OPT(x)} \right| < \varepsilon$$

Définition 11.15 – Schéma d'approximation – *On dit que l'on possède un schéma d'approximation d'un problème Π si pour tout $\varepsilon > 0$, il existe un algorithme A_ε qui soit une ε-approximation de Π.*

Définition 11.16 – Schéma d'approximation polynomial – *Un schéma d'approximation est dit polynomial si pour ε fixé, le temps d'exécution de A_ε est borné par un polynôme $P(n)$ où n représente la taille des données.*

Définition 11.17 – Schéma d'approximation complet – *Un schéma d'approximation est dit complet si le temps d'exécution de A_ε est borné par un polynôme $P(n, 1/\varepsilon)$ où n représente la taille des données.*

Nous allons, dans la suite, nous intéresser au problème des ε-approximations. En particulier, nous allons constater qu'il existe des problèmes qui n'admettent pas d'ε-approximation polynomiale.

11.5.2 Le problème du voyageur de commerce

Théorème 11.5 – *Si $\mathsf{P} \neq \mathsf{NP}$, le problème du voyageur de commerce n'admet aucune ε-approximation polynomiale, quel que soit ε positif.*

Nous allons démontrer complètement ce théorème. Pour ce faire, nous allons raisonner par l'absurde et démontrer que, s'il existe une ε-approximation polynomiale du problème du voyageur de commerce, alors le problème du cycle hamiltonien peut être résolu en temps polynomial.

---

23 Rappelons encore une fois qu'il n'est pas prouvé que les problèmes NP-complets n'aient pas de solution polynomiale.

Supposons donc qu'il existe un ε et un algorithme A_ε tel que A_ε soit une ε-approximation du problème du voyageur de commerce. Comme toute solution du voyageur de commerce ne peut être que supérieure à la solution optimale, nous avons :

$$OPT(x) \leq A_\varepsilon(x) \leq (1+\varepsilon)OPT(x)$$

Considérons maintenant un graphe $G = (X, U)$, avec $X = (x_1, \ldots, x_n)$.

Nous allons prendre comme problème du voyageur de commerce :
- l'ensemble des villes est l'ensemble des sommets du graphe ;
- $d(x_i, x_j) = 1$ si (x_i, x_j) est une arête de G ;
- $d(x_i, x_j) = \lfloor n\varepsilon \rfloor + 2$ sinon. ($\lfloor x \rfloor$ désigne la partie entière de x).

On remarque immédiatement que si G a un cycle hamiltonien, alors la solution du problème du voyageur de commerce est n (qui correspond à la tournée passant par ce cycle).

Si G n'a pas de cycle hamiltonien alors la solution du problème du voyageur de commerce est au moins :

$$n - 1 + \lfloor n\varepsilon \rfloor + 2 = n + (1 + \lfloor n\varepsilon \rfloor) > n(1+\varepsilon)$$

Or l'algorithme A_ε, appliqué à notre problème donne une tournée de longueur :

$$C \leq (1+\varepsilon)n$$

si le problème a un cycle hamiltonien et

$$C > (1+\varepsilon)n$$

sinon.

Donc on sait résoudre le problème du cycle hamiltonien en un temps polynomial, alors que ce problème est NP-complet. Donc, si P $\neq$ NP, le problème du voyageur de commerce n'a pas d'ε-approximation polynomiale.

Ce type de résultat est très important : il n'existe en fait aucune méthode polynomiale permettant de résoudre le problème du voyageur de commerce, ni même d'approcher la solution par un algorithme polynomial en majorant l'erreur.

11.5.3 Problèmes admettant des ε-approximations

Nous allons présenter ici deux problèmes admettant des ε-approximations.

Le problème du Δ-voyageur de commerce : Ce problème est le même que le problème du voyageur de commerce à la différence près que les distances entre les villes vérifient l'inégalité triangulaire :

$$d(i,j) \leq d(i,k) + d(k,j)$$

Il existe au moins deux algorithmes donnant des ε-approximations de ce problème. L'algorithme de Prim donne une approximation avec $\varepsilon = 1$ et réclame un nombre d'opérations de l'ordre de n^2. L'autre algorithme est basé sur la construction d'un cycle eulérien. Il réclame un nombre d'opérations de l'ordre de n^3 et l'on a $\varepsilon = 0.5$.

Planification de tâches indépendantes : Il existe un algorithme trivial qui est une ε-approximation de ce problème. Il consiste à affecter tous les temps $t_1, \ldots, t_n$, en commençant par t_1, systématiquement sur la machine la moins chargée jusque là. On peut démontrer que si le nombre de machines est m, $\varepsilon = \frac{1}{3} - \frac{1}{3m}$.

	0	1	2	3	4	5	6	7	8	9	10
1	V	F	V	F	F	F	F	F	F	F	F
2	V	F	V	V	F	V	F	F	F	F	F
3	V	F	V	V	V	V	V	V	F	V	F
4	V	F	V	V	V	V	V	V	V	V	V
5	V	F	V	V	V	V	V	V	V	V	V

Table 11.1 – Exemple de résolution du problème de partition

11.6 Problèmes pseudo-polynomiaux

Nous allons, dans cette section, présenter rapidement et informellement une classe importante de problèmes reliés aux problèmes NP-complets, les problèmes pseudo-polynomiaux[24].

Prenons une instance du problème de partition[25], et examinons un algorithme permettant de le résoudre.

Soit $A = \{a_1 = 2, a_2 = 3, a_3 = 4, a_4 = 5, a_5 = 6\}$. Nous avons $n = 5$ éléments, et la somme B de l'ensemble des éléments est 20, donc, pour chacun des deux ensembles construits par partition, la somme ne peut être que $B/2 = 10$ Nous allons construire le tableau 11.1 ; en abscisse, nous portons $1 \leq i \leq B/2$, en ordonnée $1 \leq j \leq n$. Nous posons pour chaque case, $b_{i,j} = V$ *s'il existe un sous-ensemble* E *de* $\{a_1, \ldots, a_j\}$ *tel que la somme des éléments de* E *est* i. Sinon, nous posons $b_{i,j} = F$.

Le tableau est construit par l'algorithme suivant :

1. Pour la première ligne, nous avons $b_{i,1} = V$ si $i = 0$ ou $i = a_1$, et $b_{i,1} = F$ sinon.
2. Pour toutes les autres lignes j ($j > 1$), nous avons $b_{i,j} = V$ si :
   - $b_{i,j-1} = V$ ou si
   - $a_j \leq i$ et $b_{i-a_j,j-1} = V$

On sait que le problème de partition a une solution si $b_{B/2,n} = V$.

Si l'on considère attentivement l'algorithme, on remarque qu'il est polynomial en fonction du nombre de cases du tableau $\frac{nB}{2}$. Bien entendu, il n'est pas polynomial en $n \log B$ (taille du codage des données), et nous n'avons donc pas un algorithme polynomial au sens de la complexité. Cependant, si nous imposions une borne sur la taille des nombres utilisables, l'algorithme deviendrait polynomial, puisque $\log B$ serait borné. Il suffirait même d'avoir une borne pour les données qui soit une fonction polynomiale de la taille du problème, pour que notre algorithme reste polynomial. Tout problème vérifiant cette propriété est dit *pseudo-polynomial*.

Un problème NP-complet peut être pseudo-polynomial, ou ne pas l'être. Ainsi, le problème du voyageur de commerce n'est pas pseudo-polynomial, alors que le problème de planification de tâches indépendantes l'est. Un problème NP-complet qui n'est pas

24 Pour plus de détails sur les problèmes pseudo-polynomiaux et les problèmes numériques, voir (Garey and Johnson 1979).

25 Nous savons (section 11.4.4) que ce problème est NP-complet ; donc il n'existe pas d'algorithme polynomial permettant de le résoudre dans l'état actuel des connaissances.

pseudo-polynomial est dit *fortement* NP-complet ou NP-complet dans le sens fort. Tous les problèmes NP-complets ne faisant pas intervenir de nombres sont « naturellement » fortement NP-complets.

Sur un plan pratique, il est important de savoir si un problème est pseudo-polynomial. En effet, très souvent, la taille des données est effectivement bornée, et un algorithme pseudo-polynomial permet alors de résoudre le problème dans un temps polynomial. Par exemple, en ce qui concerne la planification de tâches indépendantes, il existe toujours, dans les cas pratiques, une borne supérieure sur la longueur des tâches.

11.7 La cryptographie : une application

L'utilisation des ordinateurs sur une grande échelle, le remplacement lent mais régulier du courrier classique par le courrier électronique, les problèmes de sécurité et de confidentialité des informations, ont donné à la cryptographie une place de plus en plus importante[26].

C'est en 1975 que Diffie et Hellman (Diffie and Hellman 1976) introduisirent la notion de codage à *clé publique*. Dans ce système, le destinataire d'un message publie la clé de codage, et garde secrète la clé de décodage. Toute personne souhaitant lui envoyer un message le code avec la clé publique du destinataire. Celui-ci n'a alors plus qu'à décoder avec sa clé (privée) de décodage le message reçu. Pour qu'un tel système soit possible, il faut trouver un mécanisme de construction de clés de codage et de décodage tel que la connaissance de la clé de codage ne permette pas de déduire la clé de décodage. La théorie de la calculabilité et la théorie de la complexité sont là d'une grande utilité, car le créateur de la clé doit essayer d'estimer quel est le coût, en temps de calcul, pour un éventuel « briseur de code ». Nous allons voir cela sur un cas pratique, un exemple du système KNAPSACK.

Dans ce système, le destinataire du message va choisir un vecteur de n éléments de $\mathbb{N}$: $A = (a_1, a_2, \ldots, a_n)$ tel que pour tout i, on ait $(\sum_{j<i} a_j) < a_i$. Puis il choisit un nombre M (le modulo) tel que $M > 2a_n$ et un nombre u, $u < M$ et premier avec M. Il construit maintenant le vecteur $B = (b_1, b_2, \ldots, b_n)$ avec $b_i \equiv ua_i\ [M]$. Il publie alors le vecteur B qui est la clé de codage publique. Tout émetteur voulant coder un message le transformera tout d'abord en une suite de 0 et 1 (en remplaçant, par exemple, chaque lettre par son numéro d'ordre dans l'alphabet en binaire). Puis il découpera sa suite de 0 et de 1 en p paquets de n bits chacun. Il codera le paquet k de n bits $(\delta_{1,k}, \ldots, \delta_{n,k})$ par le nombre :

$$1 \leq k \leq p, \quad C_k = \sum_{1 \leq i \leq n} \delta_{i,k} b_i$$

Il ne lui reste plus qu'à diffuser les nombres C_k. Une personne malveillante qui souhaiterait décoder le message devrait, pour chaque nombre C_k, chercher les $\delta_{i,k}$. Il s'agit donc de résoudre le problème du *sac à dos*[27] tel que nous l'avons présenté précédemment. Or

---

26 Pour une présentation générale des systèmes cryptographiques, voir, par exemple, (Robling-Denning 1982). Pour les problèmes d'authentification, voir (Needham 1990).

27 Sac à dos = *knapsack* en anglais, d'où le nom du système de codage.

nous savons que ce problème est NP-complet dans le cas général. Si le vecteur B est long et si les nombres qui le composent sont grands, la théorie de la complexité nous montre que notre espion a fort peu de chance de réussir à décoder le message[28].

Maintenant, il reste au destinataire à décoder le dit message. Cela se fait aisément grâce à quelques bonnes propriétés de l'arithmétique. À partir d'un nombre C_k, le décodeur construit le nombre[29] $D_k \equiv u^{-1}C_k\ [M]$. Il lui suffit alors de résoudre le problème du sac à dos sur D_k avec A comme vecteur de nombres. Mais ce sous-problème est trivial à résoudre en raison de la forme du vecteur A. Comme chaque a_i vérifie la relation $a_i > (\sum_{j<i} a_j)$, il suffit, en commençant par $i = n$, de comparer a_i à D_k. Si $a_i > D_k$, on note $\delta_{i,k} = 0$ et on passe à a_{i-1}. Si $a_i \leq D_k$ on pose $D_k = D_k - a_i$, on note que $\delta_{i,k} = 1$ et on passe à a_{i-1}, et ceci jusqu'à $i = 1$. Le problème est alors résolu. La séquence de bits solution représente le paquet k de n bits de l'expéditeur, décodé.

Il existe de nombreux algorithmes à clé publique, le plus célèbre étant certainement le système RSA (Rivest, Shamir, Adleman). Pourtant, on ne connaît aucun système dont la sécurité soit garantie. Suivant l'expression attribuée à Adi Shamir, la cryptographie reste la lutte éternelle entre les créateurs de code et ceux qui tentent de les briser.

11.8 Complexité en terme d'espace

Nous avons jusqu'ici considéré uniquement le problème de la complexité au sens du temps de calcul. Rien n'empêche de se poser le même type de questions concernant non plus le temps de calcul, mais l'espace de calcul nécessaire à la résolution.

Nous ne citerons dans ce chapitre que quelques résultats. On peut, pour plus de renseignements, se reporter à (Garey and Johnson 1979; Salomaa 1989; Johnson 1990).

Comme nous l'avons fait pour le temps, on définit la classe des langages PSPACE acceptés par une machine de Turing déterministe dont l'espace de calcul[30] sera borné par un polynôme P. De même on définit la classe des langages NPSPACE qui sont acceptés par une machine de Turing non-déterministe dont l'espace de calcul sera borné par un polynôme P.

Il existe un premier résultat intéressant :

Théorème 11.6 – NPSPACE = PSPACE.

---

28 Il faudrait cependant être plus prudent. En fait, la théorie de la complexité nous dit que, dans le cas général, le problème du sac à dos est NP-complet. Mais le cas présent n'est pas exactement le cas général, car le vecteur B n'est pas un vecteur choisi de façon quelconque. D'autre part, nous avons p exemples du problème pour le même vecteur B. En fait, Adi Shamir a démontré en 1982 qu'il était possible de casser un système KNAPSACK du type présenté ci-dessus par des méthodes (polynomiales) d'algèbre linéaire. Voir (Shamir 1982; Salomaa 1989).

29 M et u étant premiers entre eux, u^{-1} existe et il est aisé de le calculer par le théorème d'Euclide (u^{-1} représente l'inverse de u modulo M).

30 On s'intéresse normalement au nombre de cases utilisées pendant le calcul *en sus* des cases contenant l'entrée (le codage du problème). Mais on peut négliger ces dernières car si l'espace occupé est de taille polynomiale en la taille de l'entrée, l'espace consommé ajouté à l'espace occupé par l'entrée est encore de taille polynomiale en la taille de l'entrée. Pour d'autres classes (LOGSPACE par exemple (espace logarithmique), qui est inclus dans P), l'espace effectivement consommé peut être précisément défini en rajoutant deux bandes (et deux têtes) à la machine classique à une bande. Une bande d'entrée en lecture seulement qui contient l'argument et une bande de sortie, en écriture seulement, qui contiendra le résultat. C'est l'espace consommé sur la bande lecture/écriture qui est considéré.

Ce résultat est fort différent de celui que nous avions pour les classes temporelles. Il est assez facile à démontrer, il suffit de prouver qu'une machine de Turing non-déterministe avec un espace borné polynomialement peut être simulée par une machine de Turing déterministe avec un espace borné polynomialement.

Signalons un second résultat :

Théorème 11.7 – $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE}$.

En effet, une machine de Turing (déterministe ou non) utilise un espace toujours inférieur au temps qu'elle utilise (une case au plus peut être lue/écrite par transition). On ne sait pas à l'heure actuelle si les inclusions sont strictes ou s'il s'agit d'égalités. Comme pour la classe NP, on peut définir la classe des problèmes PSPACE-complets. Ces problèmes sont au moins aussi (et *a priori* plus) difficiles que les problèmes NP-complets. Citons quelques exemples de problèmes PSPACE-complets, voire même PSPACE difficiles (au moins aussi difficiles que les problèmes PSPACE-complets) :

- **Modal Logic Provability** : Une formule A propositionnelle d'un système modal $S \in \{K, T, S_4\}$. La formule A est elle prouvable ? Le problème est PSPACE-complet. Rappelons que le problème de satisfiabilité dans S_5 est lui NP-complet (Ladner 1977; Garey and Johnson 1979).
- **Jeux** : Une position de jeu de dames américaines ou de Go (avec des règles convenablement étendues pour que le jeu soit défini sur un damier $n \times n$). Existe-t-il une stratégie gagnante pour le joueur qui a le trait ? Ces problèmes sont PSPACE durs ! Il faut d'ailleurs noter que, tel qu'il est énoncé, le problème n'a jamais été résolu, ni par un homme, ni par une machine, sur les tailles habituelles pour ces deux jeux et sur la position initiale. Il semblerait que le cas des dames américaines ne soit pas loin d'être traité (d'après J. Shaeffer, auteur du logiciel CHINOOK).

11.9 Autres classes

De très nombreuses autres classes de problèmes ont été définies, pour des problèmes ne se limitant pas aux problèmes de décision (problèmes de recherche, d'énumération, d'optimisation…).

On définit ainsi la classe des problèmes (ou langages) EXPTIME dont la complexité temporelle pour une machine déterministe est bornée par $2^{p(n)}$, pour un polynôme p, fonction de la longueur de l'entrée n. A la différence de la classe des problèmes *prouvés intraitables*, cette classe limite la complexité par le haut et non par le bas : un problème de P est dans EXPTIME, mais n'est pas intraitable.

Nous invitons le lecteur à se reporter à (Johnson 1990) pour un exposé complet sur ces classes. La figure 11.1 résume rapidement les relations entre les différentes classes que nous avons définies[31].

31 Ne sont pas présentes les classes des problèmes intraitables et indécidables qui ne sont pas des sur-ensembles des autres classes : il existe des problèmes décidables, ou de EXPTIME qui ne sont pas intraitables et tout problème de P est dans NP, dans PSPACE, dans EXPTIME et est décidable.

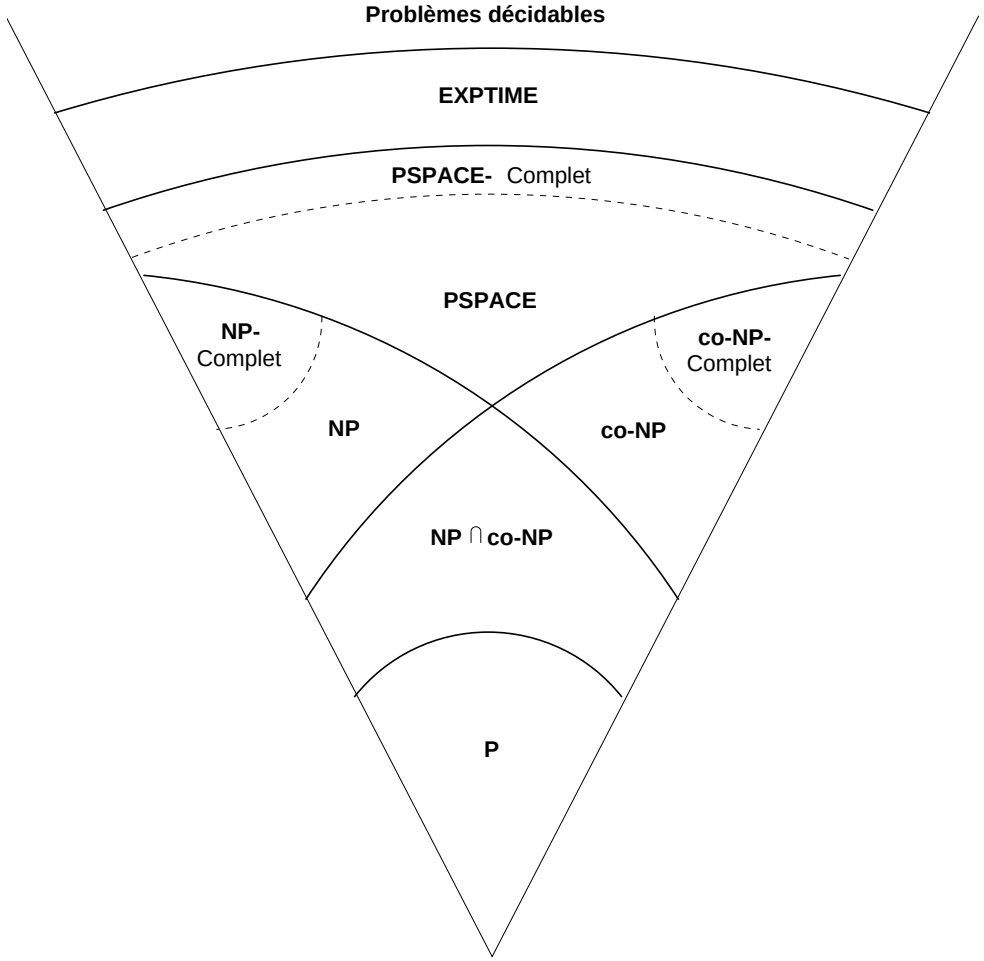

Figure 11.1 – Classes de complexité