Fondements de l'informatique
Logique, modèles, et calculs

Chapitre: Complexité en temps

Cours CSC_41012_EP

de l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version du 11 juillet 2025

Complexité en temps

Ce chapitre se focalise sur l'étude d'une ressource particulière élémentaire d'un algorithme : le temps qu'il prend pour s'exécuter.

Le chapitre précédent s'applique en particulier à cette mesure : le temps de calcul d'un algorithme est défini comme le temps qu'il prend pour s'exécuter.

Pour appuyer et illustrer l'idée de l'importance de la mesure de cette notion de complexité, intéressons nous au temps correspondant à la complexité des algorithmes n, $n\log_2 n$, n^2, n^3, 1.5^n, 2^n et $n!$ pour des entrées de taille n croissantes, sur un processeur capable d'exécuter un million d'instructions élémentaires par seconde. Nous notons ∞ dans le tableau suivant dès que le temps dépasse 10^{25} années (ce tableau est repris de [Kleinberg & Tardos, 2006]).

Complexité	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} ans
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 ans	∞
$n = 100$	< 1 s	< 1 s	< 1 s	$1s$	12,9 ans	10^{17} ans	∞
$n = 1000$	< 1 s	< 1 s	1s	18 min	∞	∞	∞
$n = 10000$	< 1 s	< 1 s	2 min	12 jours	∞	∞	∞
$n = 100000$	< 1 s	2 s	3 heures	32 ans	∞	∞	∞
$n = 1000000$	1s	20s	12 jours	31,710 ans	∞	∞	∞

On le voit, un algorithme de complexité exponentielle est très rapidement inutilisable, et donc pas *très raisonnable*. Tout l'objet du chapitre est de comprendre ce que l'on appelle un algorithme *raisonnable* en informatique, et de comprendre la théorie de la NP-complétude qui permet de discuter la frontière entre le raisonnable et le non raisonnable.

1 La notion de temps raisonnable

1.1 Convention

Pour différentes raisons, la convention suivante s'est imposée en informatique :

Définition 1 (Algorithme efficace) *Un algorithme est* efficace *si sa complexité en temps est polynomiale, c'est-à-dire en* $\mathcal{O}\left(n^k\right)$ *pour un entier* k.

Il ne s'agit que d'une convention, et on aurait pu en choisir d'autres (et à vrai dire il y en a eu d'autres avant celle-là, qui s'est imposée dans les années 1970).

Remarque 2 *On peut argumenter qu'un algorithme de complexité* $\mathcal{O}\left(n^{1794}\right)$ *n'est pas* très raisonnable. *Certes, mais il faut bien fixer une convention. On considère que c'est raisonnable en théorie de la complexité.*

Remarque 3 *Pourquoi ne pas prendre un temps linéaire, ou un temps quadratique comme notion de "raisonnable" : parce que cela ne fonctionne pas bien. En particulier, ces notions de temps linéaires et quadratiques ne vérifieraient pas la "Deuxième raison : s'affranchir du modèle de calcul" évoquée plus bas : la notion de temps linéaire ou de temps quadratique dépend du modèle de calcul utilisé, contrairement à la notion de calcul en temps polynomial.*

1.2 Première raison : s'affranchir du codage

Une des raisons de cette convention est la remarque suivante : la plupart des objets informatiques usuels peuvent se représenter de différentes façons, mais passer d'une façon de les représenter à l'autre est possible en un temps qui reste polynomial en la taille du codage.

La classe des polynômes étant stable par composition, cela implique qu'un algorithme qui est polynomial et qui travaille sur une représentation se transforme en un algorithme polynomial qui travaille sur toute autre représentation de l'objet.

On peut donc parler d'algorithme *efficace* sur ces objets sans avoir à rentrer dans les détails de comment on écrit ces objets.

Exemple 4 *Un graphe peut se représenter par une matrice, sa* matrice d'adjacence *: si le graphe possède n sommets, on considère un tableau T de taille n par n, dont l'élément* $T[i][j] \in \{0,1\}$ *vaut* 1 *si et seulement s'il y a une arête entre le sommet i et le sommet j.*

On peut aussi représenter un graphe par des listes d'adjacence *: pour représenter un graphe à n sommets, on considère n listes. La liste de numéro i code les voisins du sommet numéro i.*

On peut passer d'une représentation à l'autre en un temps qui est polynomial en la taille de chacune : on laisse le lecteur se persuader de la chose dans son langage de programmation préféré.

Un algorithme efficace *pour l'une des représentations peut toujours se transformer en un algorithme* efficace *pour l'autre représentation : il suffit de commencer éventuellement par traduire la représentation en la représentation sur laquelle travaille l'algorithme.*

Par ailleurs, par les mêmes remarques, puisque chacune de ces représentations est de taille polynomiale en n, en utilisant le fait qu'un graphe à n sommet a au

plus n^2 arêtes, un algorithme polynomial en n n'est rien d'autre qu'un algorithme efficace qui travaille sur les graphes, c'est-à-dire sur les représentations précédentes, ou toutes les représentations usuelles des graphes.

1.3 Deuxième raison : s'affranchir du modèle de calcul

Une deuxième raison profonde est la suivante : revenons sur le chapitre 7. Nous avons montré que tous les modèles de calculs de ce chapitre se simulaient l'un et l'autre : machines RAM, machines de Turing, machines à piles, machines à compteurs.

Si l'on met de côté les machines à compteur dont la simulation est particulièrement inefficace, et dont l'intérêt n'est que théorique, on peut remarquer qu'un nombre t d'instructions pour l'un se simule en utilisant un nombre polynomial en t d'instructions pour l'autre. La classe des polynômes étant stable par composition, cela implique qu'un algorithme qui est polynomial dans un modèle de calcul se transforme en un algorithme polynomial en chacun des autres modèles de calcul (quitte à simuler l'un par l'autre).

On peut donc parler d'algorithme *efficace* sur un objet sans avoir à préciser si l'on programme l'algorithme dans un modèle de calcul ou dans un autre modèle de calcul[1].

En particulier, la notion d'algorithme efficace est indépendante du langage de programmation utilisé : un algorithme efficace en CAML est un algorithme efficace en JAVA, ou un algorithme efficace en C.

Puisque la notion d'efficacité ne dépend pas du modèle, on va donc utiliser celui de la machine de Turing dans tout ce qui suit : lorsque w est un mot, rappelons que l'on note par length w sa longueur.

Définition 5 (TIME(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit alors la classe* TIME(t(n)) *comme la classe des problèmes (langages) décidés par une machine de Turing en temps $\mathcal{O}(t(n))$, où n est la taille de l'entrée.*

Si on préfère, $L \in$ TIME(t(n)) s'il y a une machine de Turing M telle que

- M décide L : pour tout mot w, M accepte w si et seulement si $w \in L$, et M refuse w si et seulement si $w \notin L$;
- M prend un temps borné par $\mathcal{O}(t(n))$:
  - si l'on préfère : il y a des entiers n_0, c, et k tels pour tout mot w, si w est de taille suffisamment grande, c'est-à-dire si length $w \geq n_0$, alors M accepte ou refuse en utilisant au plus $c * t(n)$ étapes, où $n =$ length w désigne la longueur du mot w.

Rappelons que l'on s'intéresse dans ce chapitre et dans le suivant (et en complexité) uniquement à des problèmes décidables.

1. Les plus puristes auront remarqué un problème avec le modèle des machines RAM du chapitre 7 : il faut prendre en compte dans la complexité la taille des entiers dans les opérations élémentaires effectuées et pas seulement le nombre d'instructions. Mais c'est de l'ergotage, et ce qui est écrit au dessus reste totalement vrai, si l'on interdit aux machines RAM de manipuler des entiers de taille arbitraire. De toute façon, ne pas le faire ne serait pas réaliste par rapport aux processeurs qu'ils entendent modéliser qui travaillent sur des entiers codés sur un nombre fini de bits (32 ou 64 par exemple).

1.4 Classe P

La classe des problèmes qui admettent un algorithme raisonnable correspond alors à la classe suivante.

Définition 6 (Classe P) *La classe* P *est la classe des problèmes (langages) définie par :*

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k).$$

Autrement-dit, P est exactement la classe des problèmes qui admettent un algorithme polynomial.

Voici quelques exemples de problèmes de P.

Exemple 7 (Tester le coloriage d'un graphe) ***Donnée****: Un graphe $G = (V, E)$, un ensemble fini C de couleurs, et une couleur $c(v) \in C$ pour chaque sommet $v \in V$.*

Réponse*: Décider si G est (bien)* colorié *avec ces couleurs : c'est-à-dire si il n'y a pas d'arête de G avec deux extrémités de la même couleur.*

Ce problème est dans la classe P*. En effet, il suffit de parcourir les arêtes du graphe et de tester pour chacune si la couleur de ses extrémités est la même.*

Exemple 8 (Evaluation en calcul propositionnel) ***Donnée****: Une formule du calcul propositionnel $F(x_1, x_2, \cdots, x_n)$, des valeurs $x_1, \cdots, x_n \in \{0,1\}$ pour chacune des variables de la formule.*

Réponse*: Décider si la formule F s'évalue en vraie pour ces valeurs des variables.*

Ce problème est dans la classe P*. En effet, étant donnée une formule du calcul propositionnel $F(x_1, \cdots, x_n)$ et des valeurs pour $x_1, x_2, \cdots, x_n \in \{0,1\}$, il est facile de calculer la valeur de vérité de $F(x_1, \cdots, x_n)$. Cela se fait en un temps que l'on vérifie facilement comme polynomial en la taille de l'entrée.*

Beaucoup d'autres problèmes sont dans P.

2 Comparer les problèmes

2.1 Motivation

Il s'avère toutefois qu'il y a toute une classe de problèmes pour lesquels à ce jour on n'arrive pas à construire d'algorithme polynomial, mais sans qu'on arrive à prouver formellement que cela ne soit pas possible.

C'est historiquement ce qui a mené à considérer la classe de problèmes que l'on appelle NP, que nous verrons dans la section suivante.

Des exemples de problèmes dans cette classe sont les suivants :

Exemple 9 (k-COLORABILITE) ***Donnée***: *Un graphe $G = (V, E)$ et un entier k.*
Réponse: *Décider s'il existe un coloriage du graphe utilisant au plus k couleurs : c'est-à-dire décider s'il existe une façon de colorier les sommets de G avec au plus k couleurs pour obtenir un coloriage de G.*

Exemple 10 (SAT**, Satisfaction en calcul propositionnel**) ***Donnée***: *Une formule $F(x_1, \cdots, x_n)$ du calcul propositionnel.*
Réponse: *Décider si F est satisfiable : c'est-à-dire décider s'il existe $x_1, \cdots, x_n \in \{0,1\}^n$ tel que F s'évalue en vraie pour cette valeur de ses variables $x_1, \cdots, x_n$.*

Exemple 11 (CIRCUIT HAMILTONIEN) ***Donnée***: *Un graphe $G = (V, E)$ (non-orienté).*
Réponse: *Décider s'il existe un* circuit hamiltonien*, c'est-à-dire décider s'il existe un chemin de G passant une fois et une seule par chacun des sommets et revenant à son point de départ.*

Pour les trois problèmes on connaît des algorithmes exponentiels : tester tous les coloriages, pour le premier, ou toutes les valeurs de $\{0,1\}^n$ pour le second, ou tous les chemins pour le dernier. Pour les trois problèmes on ne connaît pas d'algorithme efficace (polynomial), et on n'arrive pas à prouver qu'il n'y en a pas.

Comme nous allons le voir, on sait toutefois montrer que ces trois problèmes sont équivalents au niveau de leur difficulté, et cela nous amène à la notion de réduction, c'est-à-dire à l'idée de comparer la difficulté des problèmes.

Avant, quelques précisions.

2.2 Remarques

Dans ce chapitre et le suivant, on va ne parler essentiellement que de problèmes de décisions, c'est-à-dire de problèmes dont la réponse est soit "vraie" ou "fausse" : voir la définition 9.7.

Exemple 12 *"Trier n nombres" n'est pas un problème de décision : la sortie est a priori une liste triée de nombres.*

Exemple 13 *"Étant donné un graphe $G = (V, E)$, déterminer le nombre de couleurs pour colorier G" n'est pas un problème de décision, car la sortie est un entier. On peut toutefois formuler un problème de décision proche, du type "Étant donné un graphe $G = (V, E)$, et un entier k, déterminer si le graphe G admet un coloriage avec moins de k couleurs" : c'est le problème* k-COLORABILITE.

Avant de pouvoir parler de réduction, il faut aussi parler de fonctions calculables en temps polynomial : c'est la notion à laquelle on s'attend, même si on est obligé de l'écrire car on ne l'a encore jamais fait.

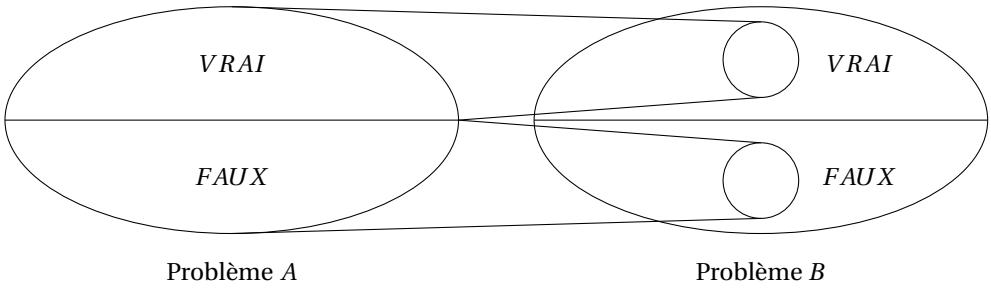

FIGURE 1 – Les réductions transforment des instances positives en instances positives, et négatives en négatives.

Définition 14 (Fonction calculable en temps polynomial) *Soient* Σ *et* Σ' *deux alphabets. Une fonction* $f : \Sigma^* \to \Sigma'^*$ *est* calculable en temps polynomial *s'il existe une machine de Turing A, qui travaille sur l'alphabet* $\Sigma \cup \Sigma'$*, et un entier* k*, telle que pour tout mot* w*, A avec l'entrée* w *termine en un temps* $\mathcal{O}\left(n^k\right)$ *avec lorsqu'elle termine* $f(w)$ *écrit sur son ruban, où* $n = \text{length}\, w$*.*

Le résultat suivant est facile à établir :

Proposition 15 (Stabilité par composition) *La composée de deux fonctions calculables en temps polynomial est calculable en temps polynomial.*

2.3 Notion de réduction

Cela nous permet d'introduire une notion de réduction entre problèmes (similaire à celle du chapitre 9 , si ce n'est que l'on parle de calculable en temps polynomial plutôt que de calculable) : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A : voir la figure 2 et la figure 1.

Définition 16 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs* M_A *et* M_B*. Une* réduction de A vers B *est une fonction* $f : M_A^* \to M_B^*$ *calculable en temps polynomial telle que* $w \in A$ *ssi* $f(w) \in B$*. On note* $A \leq B$ *lorsque A se réduit à B.*

Cela se comporte comme on le souhaite : un problème est aussi facile (et difficile) que lui-même, et la relation "être plus facile que" est transitive. En d'autres termes :

Théorème 17 $\leq$ *est un préordre :*

1. $L \leq L$;

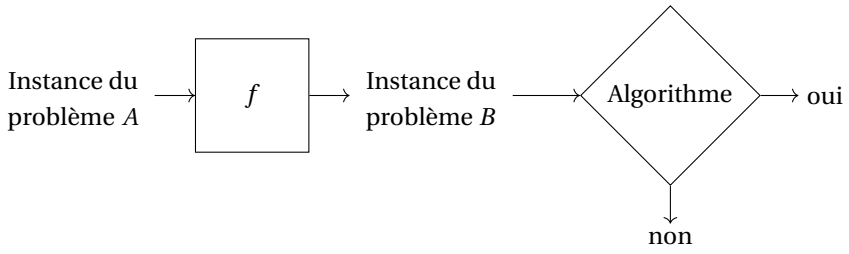

FIGURE 2 – Réduction du problème A vers le problème B. Si l'on peut résoudre le problème B en temps polynomial, alors on peut résoudre le problème A en temps polynomial. Le problème A est donc plus facile que le problème B, noté $A \leq B$.

2. $L_1 \leq L_2$, $L_2 \leq L_3$ impliquent $L_1 \leq L_3$.

Démonstration: Considérer la fonction identité comme fonction f pour le premier point.

Pour le second point, supposons $L_1 \leq L_2$ via la réduction f, et $L_2 \leq L_3$ via la réduction g. On a $x \in L_1$ ssi $g(f(x)) \in L_2$. Il suffit alors de voir que $g \circ f$, en temps que composée de deux fonctions calculables en temps polynomial est calculable en temps polynomial. □

Remarque 18 *Il ne s'agit pas d'un ordre, puisque $L_1 \leq L_2$, $L_2 \leq L_1$ n'implique pas $L_1 = L_2$.*

Il est alors naturel d'introduire :

Définition 19 *Deux problèmes L_1 et L_2 sont équivalents, noté $L_1 \equiv L_2$, si $L_1 \leq L_2$ et si $L_2 \leq L_1$.*

On a alors $L_1 \leq L_2$, $L_2 \leq L_1$ impliquent $L_1 \equiv L_2$.

2.4 Application à la comparaison de difficulté

Intuitivement, si un problème est plus facile qu'un problème polynomial, alors il est polynomial. Formellement :

Proposition 20 (Réduction) *Si $A \leq B$, et si $B \in$ P alors $A \in$ P.*

Démonstration: Soit f une réduction de A vers B. A est décidé par la machine de Turing qui, sur une entrée w, calcule $f(w)$, puis simule la machine de Turing qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, l'algorithme est correct, et fonctionne bien en temps polynomial. □

En prenant la contraposée de la proposition précédente, on obtient la formulation suivante qui dit que si un problème n'a pas d'algorithme polynomial, et qu'il est plus facile qu'un autre, alors l'autre non plus.

Proposition 21 (Réduction) *Si $A \leq B$, et si $A \notin \mathrm{P}$ alors $B \notin \mathrm{P}$.*

Exemple 22 *Nous verrons que le problème d'un coloriage d'un graphe, de la satisfaction du calcul propositionnel, ou de l'existence d'un chemin hamiltonien sont équivalents (et équivalents à tous les problèmes* NP*-complets). Il y a donc un algorithme efficace pour l'un ssi il y en a un pour l'autre.*

2.5 Problèmes les plus durs

Si on considère une classe de problèmes, on peut introduire la notion de problème le plus difficile pour la classe, i.e. maximum pour $\leq$. C'est la notion de *complétude* :

Définition 23 ($\mathscr{C}$-complétude) *Soit $\mathscr{C}$ une classe de problèmes de décisions. Un problème A est dit $\mathscr{C}$-complet, si*

1. *il est dans $\mathscr{C}$;*
2. *tout autre problème B de $\mathscr{C}$ est tel que $B \leq A$.*

On dit qu'un problème A est *$\mathscr{C}$-dur* s'il vérifie la condition 2 de la définition 23. Un problème A est donc $\mathscr{C}$-complet s'il est $\mathscr{C}$-dur et dans la classe $\mathscr{C}$.

Un problème $\mathscr{C}$-complet est donc le plus difficile, ou un des plus difficiles, de la classe $\mathscr{C}$. Clairement, s'il y en a plusieurs, ils sont équivalents :

Corollaire 24 *Soit $\mathscr{C}$ une classe de langages. Tous les problèmes $\mathscr{C}$-complets sont équivalents.*

Démonstration: Soient A et B deux problèmes $\mathscr{C}$-complets. Appliquer la condition 2 de la définition 23 en A relativement à $B \in \mathscr{C}$, et en B relativement à $A \in \mathscr{C}$. □

3 La classe NP

3.1 La notion de vérificateur

Les problèmes k-COLORABILITE, SAT et CIRCUIT HAMILTONIEN évoqués précédemment ont un point commun : s'il n'est pas clair qu'ils admettent un algorithme polynomial, il est clair qu'ils admettent un *vérificateur* polynomial.

Définition 25 (Vérificateur) *Un* vérificateur *pour un problème A est un algorithme V tel que*

$$A = \{w | V \text{ accepte } \langle w, u \rangle \text{ pour un certain mot } u\}.$$

Le vérificateur est polynomial si V se décide en temps polynomial en la longueur de w. On dit qu'un langage est polynomialement vérifiable *s'il admet un*

vérificateur polynomial.

Le mot u est alors appelé un *certificat* (parfois aussi une *preuve*) pour w. Autrement dit, un vérificateur utilise une information en plus, à savoir u pour vérifier que w est dans A.

Remarque 26 *Remarquons que l'on peut toujours se restreindre aux certificats de longueur polynomiale en la longueur de w, puisqu'en temps polynomial en la longueur de w l'algorithme V ne pourra pas lire plus qu'un nombre polynomial de symboles du certificat.*

Exemple 27 *Un certificat pour le problème* k-COLORABILITE *est la donnée des couleurs pour chaque sommet.*

Nous ne donnerons pas toujours autant de détails, mais voici la justification : en effet, un graphe G est dans k-COLORABILITE si et seulement si on peut trouver un mot u qui code des couleurs pour chaque sommet tel que ces couleurs donnent un coloriage correct : l'algorithme V, i.e. le vérificateur, se contente, étant donné $\langle G, u\rangle$ de vérifier que le coloriage est correct, ce qui se fait bien en temps polynomial en la taille du graphe : voir la discussion de l'exemple 7.

Exemple 28 *Un certificat pour le problème* SAT *est constitué de la donnée d'une valeur $x_1, \cdots, x_n \in \{0,1\}$ pour chacune des variables de la formule F : le vérificateur, qui se contente de vérifier que ces valeurs satisfont la formule F, peut bien se réaliser en temps polynomial en la taille de la formule : voir l'exemple 8.*

Exemple 29 *Un certificat pour le problème* CIRCUIT HAMILTONIEN *est la donnée d'un circuit. Le vérificateur se contente de vérifier que le circuit est hamiltonien, ce qui se fait bien en temps polynomial.*

Cela nous amène à la définition suivante :

Définition 30 NP *est la classe des problèmes (langages) qui possèdent un vérificateur polynomial.*

Cette classe est importante car elle s'avère contenir un nombre incroyable de problèmes d'intérêt pratique. Elle contient les problèmes k-COLORABILITE, SAT et CIRCUIT HAMILTONIEN mais aussi beaucoup d'autres problèmes : voir par exemple tout le chapitre qui suit.

Par construction, on a (car le mot vide est par exemple un certificat pour tout problème de P) :

Proposition 31 $\mathrm{P} \subseteq \mathrm{NP}$.

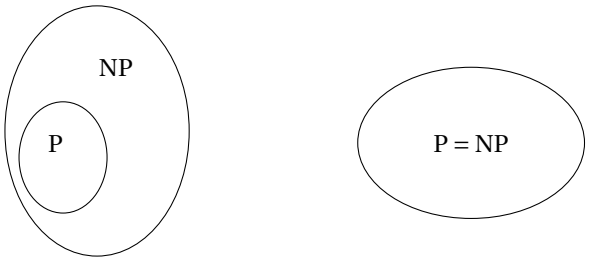

FIGURE 3 – Une des deux possibilités est correcte.

3.2 La question P = NP ?

Clairement, on a soit P = NP soit P $\subsetneq$ NP : voir la figure 3.

La question de savoir si ces deux classes sont égales ou distinctes est d'un enjeu impressionnant. D'une part parce que c'est l'une des questions (voire la question) non résolue les plus célèbres de l'informatique théorique et des mathématiques qui défie les chercheurs depuis plus de 40 ans : elle a été placée parmi la liste des questions les plus importantes pour les mathématiques et l'informatique pour le millénaire en 2000. Le *Clay Mathematics Institute* offre 1 000 000 de dollars à qui déterminerait la réponse à cette question.

Surtout, si P = NP, alors tous les problèmes vérifiables polynomialement seraient décidables en temps polynomial. La plupart des personnes pensent que ces deux classes sont distinctes car il y a un très grand nombre de problèmes pour lesquels on n'arrive pas à produire d'algorithme polynomiaux depuis plus de 40 ans.

Elle a aussi un enjeu économique impressionnant puisque de nombreux systèmes, dont certains systèmes de cryptographie actuels sont basés sur l'hypothèse que ces deux classes sont distinctes : si ce n'était pas le cas, de nombreuses considérations sur ces systèmes s'effondreraient, et de nombreuses techniques de cryptage devraient être revues.

3.3 Temps non déterministe polynomial

Faisons tout d'abord une petite parenthèse sur la terminologie : le "N" dans NP vient de *non déterministe* (et pas de *non*, ou *not* comme souvent beaucoup le croient), en raison du résultat suivant :

Théorème 32 *Un problème est dans* NP *si et seulement s'il est décidé par une machine de Turing non déterministe en temps polynomial.*

Rappelons que nous avons introduit les machines de Turing non déterministes dans le chapitre 7. On dit qu'un langage $L \subset \Sigma^*$ est *décidé par la machine non déterministe M en temps polynomial* si M décide L et M prend un temps borné par $\mathcal{O}(t(n))$: il y a des entiers n_0, c, et k tels que pour tout mot w de taille suffisamment grande, i.e. length $w \geq n_0$, et pour tout mot w, si on note n = length w sa longueur,

pour $w \in L$, M admet un calcul qui accepte en utilisant moins de $c * n^k$ étapes, et pour $w \notin L$, tous les calculs de M mènent à une configuration de refus en moins de $c * n^k$ étapes.

Démonstration: Considérons un problème A de NP. Soit V le vérificateur associé, qui fonctionne en temps polynomial $p(n)$. On construit une machine de Turing M non déterministe qui, sur un mot w, va produire de façon non déterministe un mot u de longueur $p(n)$ puis simuler V sur $\langle w, u \rangle$: si V accepte, alors M accepte. Si V refuse, alors M refuse. La machine M décide A.

Réciproquement, soit A un problème décidé par une machine de Turing non déterministe M en temps polynomial $p(n)$. Comme dans la preuve de la proposition 7.23 dans le chapitre 7, on peut affirmer que le degré de non déterminisme de la machine est borné, et qu'il vaut un certain entier r, et que les suites des choix non déterministes réalisées par la machine M jusqu'au temps t se codent par une suite de longueur t d'entiers entre 1 à (au plus) r.

Par conséquent, une suite d'entiers de longueur $p(n)$ entre 1 et r est un certificat valide pour un mot w : étant donné w et un mot u codant une telle suite, un vérificateur V peut facilement vérifier en temps polynomial si la machine M accepte w avec ces choix non déterministes. □

Plus généralement, on définit :

Définition 33 (NTIME(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* NTIME(t(n)) *comme la classe des problèmes (langages) décidés par une machine de Turing non déterministe en temps $\mathcal{O}(t(n))$, où n est la taille de l'entrée.*

Corollaire 34

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(\text{n}^{\text{k}}).$$

3.4 NP-complétude

Il s'avère que la classe NP possède une multitude de problèmes complets : le prochain chapitre en présente toute une liste.

La difficulté est d'arriver à en produire un premier. C'est l'objet du théorème de Cook et Levin.

Théorème 35 (Cook-Levin) *Le problème* SAT *est* NP*-complet.*

Nous prouverons ce théorème dans la section qui suit.

Commençons par reformuler ce que cela signifie.

Corollaire 36 P = NP *si et seulement si* SAT ∈ P.

Démonstration: Puisque SAT est dans NP, si P = NP, alors SAT ∈ P.

Réciproquement, puisque SAT est complet, pour tout problème $B \in$ NP, $B \leq$ SAT et donc $B \in$ P si SAT ∈ P par la proposition 20. □

Ce que nous venons de faire est vrai pour n'importe quel problème NP-complet.

Théorème 37 *Soit A un problème* NP*-complet.*
P = NP *si et seulement si* $A \in$ P.

Démonstration: Puisque A est complet il est dans NP, et donc si P = NP, alors $A \in$ P. Réciproquement, puisque A est NP-dur, pour tout problème $B \in$ NP, $B \leq A$ et donc $B \in$ P si $A \in$ P par la proposition 20. □

Remarque 38 *On voit donc tout l'enjeu de produire des problèmes* NP*-complets pour la question de prouver* P ≠ NP *: tenter de produire un problème pour lequel on arriverait à prouver qu'il n'existe pas d'algorithme polynomial. A ce jour, aucun des centaines, voir des milliers de problèmes* NP*-complets connus n'ont permis cependant de prouver que* P ≠ NP.

Remarque 39 *Rappelons nous que tous les problèmes complets sont équivalents en difficulté par le corollaire 24.*

3.5 Méthode pour prouver la NP-complétude

La NP-complétude d'un problème s'obtient dans la quasi-totalité des cas de la façon suivante :

Pour prouver la NP*-complétude d'un problème A, il suffit :*

1. *de prouver qu'il est dans* NP*;*
2. *et de prouver que $B \leq A$ pour un problème B que l'on sait déjà* NP*-complet.*

En effet, le point 1. permet de garantir que $B \in$ NP, et le point 2. de garantir que pour tout problème $C \in$ NP on a $C \leq A$: en effet, par la NP-complétude de B on a $C \leq B$, et puisque $B \leq A$, on obtient $C \leq A$.

Remarque 40 *Attention, la* NP*-complétude d'un problème A s'obtient en prouvant qu'il est plus difficile qu'un autre problème* NP*-complet, et pas le contraire. C'est une erreur fréquente dans les raisonnements.*

Le chapitre suivant est consacré à de multiples applications de cette stratégie pour différents problèmes.

3.6 Preuve du théorème de Cook-Levin

Nous ne pouvons pas appliquer la méthode précédente pour prouver la NP-complétude de SAT, car nous ne connaissons encore aucun problème NP-complet.

Il nous faut donc faire une preuve autrement, en revenant à la définition de la NP-complétude : il faut prouver d'une part que SAT est dans NP, et d'autre part que tout problème A de NP vérifie $A \leq$ SAT.

Le fait que SAT est dans NP a déjà été établi : voir l'exemple 28.

Considérons un problème A de NP, et un vérificateur V associé. L'idée (qui a certaines similarités avec les constructions du chapitre 10) est, étant donné un mot

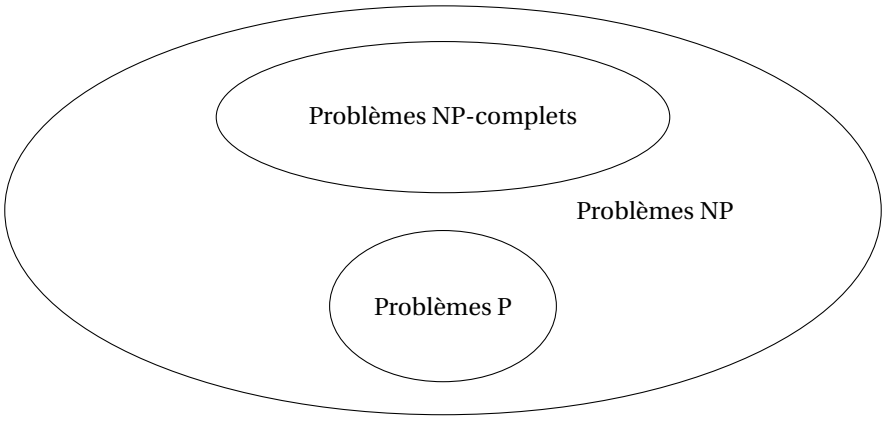

FIGURE 4 – Situation avec l'hypothèse P ≠ NP.

w, de construire une formule propositionnelle $\gamma = \gamma(u)$ qui code l'existence d'un calcul accepteur de V sur $\langle w, u\rangle$ pour un certificat u.

On va en fait construire une série de formules dont le point culminant sera la formule $\gamma = \gamma(u)$ qui codera l'existence d'une suite de configurations $C_0, C_1, \cdots, C_t$ de M telle que :

— C_0 est la configuration initiale de V sur $\langle w, u\rangle$;
— C_{i+1} est la configuration successeur de C_i, selon la fonction de transition δ de la machine de Turing V, pour $i < t$;
— C_t est une configuration acceptante.

En d'autres termes, l'existence d'un diagramme espace-temps valide correspondant à un calcul de V sur $\langle w, u\rangle$.

En observant que la formule propositionnelle obtenue γ reste de taille polynomiale en la taille de w, et peut bien s'obtenir par un algorithme polynomial à partir de w, on aura prouvé le théorème : en effet, on aura $w \in L$ si et seulement s'il existe u qui satisfait $\gamma(u)$, c'est-à-dire $A \leq$ SAT via la fonction f qui à w associe $\gamma(u)$.

Il ne reste plus qu'à donner les détails fastidieux de la construction de la formule $\gamma(u)$. Par hypothèse, V fonctionne en temps polynomial $p(n)$ en la taille n de w. En ce temps là, V ne peut pas déplacer sa tête de lecture de plus de $p(n)$ cases vers la droite, ou de $p(n)$ cases vers la gauche. On peut donc se restreindre à considérer un sous-rectangle de taille $(2 * p(n) + 1) \times p(n)$ du diagramme espace-temps du calcul de V sur $\langle w, u\rangle$: voir la figure 5.

Les cases du tableau $T[i, j]$ correspondant au diagramme espace temps sont des éléments de l'ensemble fini $C = \Gamma \cup Q$. Pour chaque $1 \leq i \leq p(n)$ et $1 \leq j \leq 2 * p(n) + 1$ et pour chaque $s \in C$, on définit une variable propositionnelle $x_{i,j,s}$. Si $x_{i,j,s}$ vaut 1, cela signifie que la case $T[i, j]$ du tableau contient s.

La formule γ est la conjonction de 4 formules CELL $\wedge$ START(u) $\wedge$ MOVE $\wedge$ HALT.

La formule CELL permet de garantir qu'il y a bien un symbole et un seul par case.

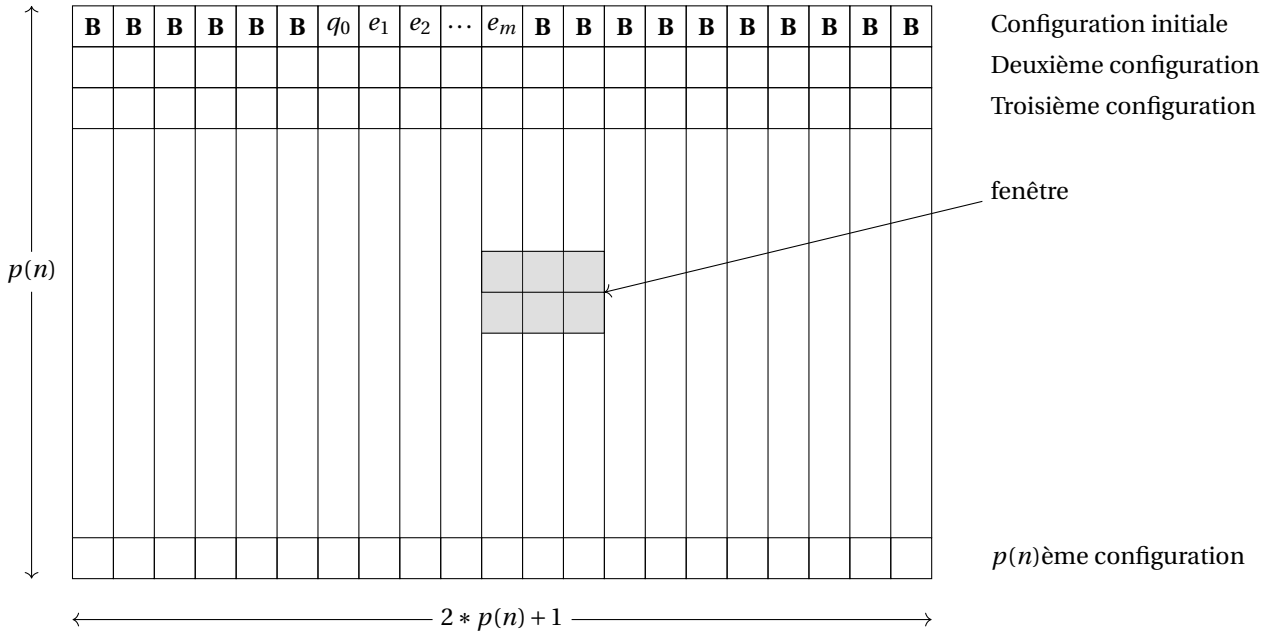

FIGURE 5 – Un tableau $(2p(n)+1) \times p(n)$ codant le diagramme espace-temps du calcul de V sur $\langle w, u\rangle$.

$$\text{CELL} = \bigwedge_{1\le i\le p(n), 1\le j\le 2p(n)+1} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t\in C, s\neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right].$$

Les symboles $\bigwedge$ et $\bigvee$ désignent l'itération des symboles $\wedge$ et $\vee$ correspondants. Par exemple, $\bigvee_{s\in C} x_{i,j,s}$ est un raccourci pour la formule $x_{i,j,s_1} \vee \cdots \vee x_{i,j,s_l}$ si $C = \{s_1, \cdots, s_l\}$.

Si on note le mot $e_1 e_2 \cdots e_m$ le mot $\langle w, u\rangle$, la formule START(u) permet de garantir que la première ligne correspond bien à la configuration initiale du calcul de V sur $\langle w, u\rangle$ pour un certani u.

$$\text{START}(u) = x_{1,1,\mathbf{B}} \wedge x_{1,2,\mathbf{B}} \wedge \cdots \wedge x_{1,p(n)+1,q_0} \wedge x_{1,p(n)+2,e_1} \wedge \cdots \wedge x_{1,p(n)+m+1,e_m}$$

$$\vee x_{1,p(n)+m+2,\mathbf{B}} \wedge \cdots \wedge x_{1,2p(n)+1,\mathbf{B}}.$$

La formule HALT permet de garantir qu'une des lignes correspond bien à une configuration acceptante

$$\text{HALT} = \bigvee_{1\le i\le p(n), 1\le j\le 2p(n)+1} x_{i,j,q_a}.$$

Enfin la formule MOVE écrit que tous les sous-rectangles 3×2 du tableau T sont des fenêtres légales : voir la notion de fenêtre légale du chapitre 7.

$$\text{MOVE} = \bigwedge_{1 \le i \le p(n), 1 \le j \le 2p(n)+1} \text{LEGAL}_{i,j},$$

où $\text{LEGAL}_{i,j}$ est une formule propositionnelle qui exprime que le sous-rectangle 3×2 à la position i, j est une fenêtre légale :

$$\text{LEGAL}_{i,j} = \bigvee_{(a,b,c,d,e,f) \in \text{WINDOW}} (x_{i,j-1,a} \wedge x_{i,j,b} \wedge x_{i,j+1,c} \wedge x_{i+1,j-1,d} \wedge x_{i+1,j,e} \wedge x_{i,j+1,f}),$$

où WINDOW est l'ensemble des 6-uplets (a, b, c, d, e, f) tels que si trois éléments de Σ représentés respectivement par a, b et c apparaissent consécutivement dans une configuration C_i, et si d, e, f apparaissent consécutivement aux mêmes emplacements dans la configuration C_{i+1}, alors cela est cohérent avec la fonction de transition δ de la machine de Turing M.

Ceci termine la preuve, si l'on ajoute que chacune de ces formules sont faciles à écrire (et donc produisibles facilement en temps polynomial à partir de w), et qu'elles restent bien de taille polynomiale en la taille de w.

4 Quelques autres résultats de la théorie de la complexité

Commençons par une remarque à propos de l'hypothèse que nous avons faite sur le fait de restreindre notre discussion aux problèmes de décision.

4.1 Décision vs Construction

Nous avons parlé jusque-là uniquement de problèmes de *décision*, c'est-à-dire dont la réponse est soit vraie ou soit fausse (par exemple : "étant donnée une formule F, décider si la formule F est satisfiable") en opposition aux problèmes qui consisteraient à *produire un objet avec une propriété* (par exemple : étant donnée une formule F, produire une affectation des variables qui la satisfait s'il en existe une).

Clairement, produire une solution est plus difficile que de savoir s'il en existe une, et donc si P $\neq$ NP, aucun de ces deux problèmes n'admet une solution en temps polynomial, et ce pour tout problème NP-complet.

Cependant, si P = NP, il s'avère qu'alors on sait aussi produire une solution :

Théorème 41 *Supposons que* P = NP. *Soit L un problème de* NP *et V un vérificateur associé. On peut construire une machine de Turing qui sur toute entrée $w \in L$ produit en temps polynomial un certificat u pour w pour le vérificateur V.*

Démonstration: Commençons par le prouver pour L correspondant au problème de la satisfaction de formules (problème SAT). Supposons P = NP : on peut donc tester si une formule propositionnelle F à n variables est satisfiable ou non en temps

polynomial. Si elle est satisfiable, alors on peut fixer sa première variable à 0, et tester si la formule obtenue F_0 est satisfiable. Si elle l'est, alors on écrit 0 et on recommence récursivement avec cette formule F_0 à $n-1$ variables. Sinon, nécessairement tout certificat doit avoir sa première variable à 1, on écrit 1, et on recommence récursivement avec la formule F_1 dont la première variable est fixée à 1, qui possède $n-1$ variables. Puisqu'il est facile de vérifier si une formule sans variable est satisfiable, par cette méthode, on aura écrit un certificat.

Maintenant si L est un langage quelconque de NP, on peut utiliser le fait que la réduction produite par la preuve du théorème de Cook-Levin est en fait une réduction de *Levin* : non seulement on a $w \in L$ si et seulement si $f(w)$ est une formule satisfiable, mais on peut aussi retrouver un certificat pour w à partir d'un certificat de la satisfiabilité de la formule $f(w)$. On peut donc utiliser l'algorithme précédent pour retrouver un certificat pour L. □

En fait, on vient d'utiliser le fait que le problème de la satisfiabilité d'une formule est *auto-réductible* à des instances de tailles inférieures.

4.2 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq n\log(n)$ est *constructible en temps*, si la fonction qui envoie 1^n sur la représentation binaire de $1^{f(n)}$ est calculable en temps $\mathcal{O}\big(f(n)\big)$.

La plupart des fonctions usuelles sont constructibles en temps, et en pratique ce n'est pas vraiment une restriction.

Remarque 42 *Par exemple, $n\sqrt{n}$ est constructible en temps : sur l'entrée 1^n, on commence par compter le nombre de 1 en binaire : on peut utiliser pour cela un compteur, qui reste de taille $\log(n)$, que l'on incrémente : cela se fait donc en temps $\mathcal{O}\big(n\log(n)\big)$ puisqu'on utilise au plus $\mathcal{O}\big(\log(n)\big)$ étapes pour chaque lettre du mot en entrée. On peut alors calculer $\lfloor n\sqrt{n}\rfloor$ en binaire à partir de la représentation de n. N'importe quelle méthode pour faire cela fonctionne en temps $\mathcal{O}\big(n\log(n)\big)$, puisque la taille des nombres impliqués est $\mathcal{O}\big(\log(n)\big)$.*

Théorème 43 (Théorème de hiérarchie en temps) *Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ constructible en temps, il existe un langage L qui est décidable en temps $\mathcal{O}\big(f(n)\big)$ mais pas en temps $o(f(n))$.*

Démonstration: Il s'agit d'une généralisation de l'idée de la preuve du théorème 14.30 du chapitre suivant : nous invitons notre lecteur à commencer par cette dernière preuve.

Nous prouverons une version plus faible que l'énoncé plus haut. Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction constructible en temps.

On considère le langage (très artificiel) L qui est décidé par la machine de Turing B suivante :

— sur une entrée w de taille n, B calcule $f(n)$ et mémorise $\langle f(n)\rangle$ le codage en binaire de $f(n)$ dans un compteur binaire c ;

— Si w n'est pas de la forme $\langle A\rangle 10^*$, pour un certaine machine de Turing A, alors la machine de Turing B refuse.
— Sinon, B simule A sur le mot w pendant $f(n)$ étapes pour déterminer si A accepte en un temps inférieur à $f(n)$:
  — si A accepte en ce temps, alors B refuse ;
  — sinon B accepte.

Autrement dit B simule A sur w, étape par étape, et décrémente le compteur c à chaque étape. Si ce compteur c atteint 0 ou si A refuse, alors B accepte. Sinon, B refuse.

Par l'existence d'une machine de Turing universelle, il existe des entiers k et d tels que L soit décidé en temps $d \times f(n)^k$.

Supposons que L soit décidé par une machine de Turing A en temps $g(n)$ avec $g(n)^k = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $d \times g(n)^k < f(n)$.

Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $\langle A\rangle 10^{n_0}$. Puisque cette entrée est de taille plus grande que n_0, B répond l'inverse de la machine de Turing A sur la même entrée. Donc B et A ne décident pas le même langage, et donc la machine de Turing A ne décide pas L, ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en temps $g(n)$ pour toute fonction $g(n)$ avec $g(n)^k = o(f(n))$.

□

Le théorème suivant est une généralisation de l'idée de cette preuve. Le facteur log vient de la construction d'une machine de Turing universelle nettement plus efficace que ceux considérés dans ce document, introduisant seulement un ralentissement logarithmique en temps.

Théorème 44 (Théorème de hiérarchie en temps) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en temps telles que $f(n)\log(f(n)) = o(f'(n))$. Alors l'inclusion* TIME(f) ⊂ TIME(f') *est stricte.*

On obtient par exemple :

Corollaire 45 $\mathrm{TIME}(n^2) \subsetneq \mathrm{TIME}(n^{\log n}) \subsetneq \mathrm{TIME}(2^n)$.

On définit :

Définition 46 *Soit*

$$\mathrm{EXPTIME} = \bigcup_{c\in\mathbb{N}} \mathrm{TIME}(2^{n^c}).$$

On obtient :

Corollaire 47 P $\subsetneq$ EXPTIME.

Démonstration: Tout polynôme devient ultimement négligeable devant 2^n, et donc P est un sous-ensemble de TIME(2^{n}). Maintenant TIME(2^{n}), qui contient tout P est un sous-ensemble strict de, par exemple, TIME(2^{n^3}), qui est inclus dans EXPTIME. □

4.3 EXPTIME **and** NEXPTIME

Considérons

$$\text{EXPTIME} = \bigcup_{c\geq 1} \text{TIME}(2^{n^c})$$

et

$$\text{NEXPTIME} = \bigcup_{c\geq 1} \text{NTIME}(2^{n^c}).$$

On sait prouver le résultat suivant (et ce n'est pas vraiment difficile) :

Théorème 48 *Si* EXPTIME $\neq$ NEXPTIME *alors* P $\neq$ NP.

5 Que signifie la question $P = NP$?

Nous faisons une digression autour de la signification de cette question en lien avec la théorie de la preuve, et les autres chapitres du document.

On peut voir NP comme classe des langages tel qu'en tester l'appartenance revient à déterminer s'il existe un certificat **court** (polynomial). On peut relier cela à l'existence d'une preuve en mathématiques. En effet, dans son principe même, la déduction mathématique consiste à produire des théorèmes à partir d'axiomes.

On s'attend à ce que la validité d'une preuve soit facile à vérifier : il suffit de vérifier que chaque ligne de la preuve soit bien la conséquence des lignes précédentes, dans le système de preuve. En fait, dans la plupart des systèmes de preuves axiomatiques (par exemple dans tous les systèmes de preuve que nous avons vu) cette vérification se fait en un temps qui est polynomial en la longueur de la preuve.

Autrement dit, le problème de décision suivant est NP pour chacun des systèmes de preuve axiomatiques usuels $\mathscr{A}$, et en particulier pour celui $\mathscr{A}$ que nous avons vu pour le calcul des prédicats.

$$\text{THEOREMS} = \{\langle \phi, \mathbf{1}^n \rangle | \phi \text{ possède une preuve de longueur } \leq n \text{ dans le système } \mathscr{A}\}.$$

Nous laisserons à notre lecteur l'exercice suivant :

Exercice 1 *La théorie des ensembles de* Zermelo-Fraenkel *est un des systèmes axiomatiques permettant d'axiomatiser les mathématiques avec une description finie. (Même sans connaître tous les détails de la théorie des ensembles de* Zermelo-Fraenkel*) argumenter à un haut niveau que le problème* THEOREMS *est* NP*-complet pour la théorie des ensembles de* Zermelo-Fraenkel.

Indice : la satisfiabilité d'un circuit booléen est un énoncé.

Autrement dit, en vertu du théorème 41, la question P = NP est celle (qui a été posée par Kurt Gödel) de savoir s'il existe une machine de Turing qui soit capable de produire la preuve mathématique de tout énoncé ϕ en un temps polynomial en la longueur de la preuve.

Cela semble-t-il raisonnable ?

Que signifie la question NP = coNP**?** Rappelons que coNP est la classe des langages dont le complémentaire est dans NP. La question NP = coNP, est reliée à l'existence de preuve courte (de certificats) pour des énoncés qui ne semblent pas en avoir : par exemple, il est facile de prouver qu'une formule propositionnelle est satisfiable (on produit une valuation de ses entrées, que l'on peut coder dans une preuve qui dit qu'en propageant les entrées vers les sorties, le circuit répond 1). Par contre, dans le cas général, il n'est pas facile d'écrire une preuve courte qu'une formule propositionnelle donnée est non satisfiable. Si NP = coNP, il doit toujours en exister une : la question est donc reliée à l'existence d'un autre moyen de prouver la non satisfiabilité d'une formule propositionnelle, que les méthodes usuelles.

On peut reformuler l'équivalent pour chacun des problèmes NP-complets évoqués.

6 Exercices

Exercice 2 *Prouver que la classe* P *est close par union, concaténation et complément.*

Exercice 3 *Prouver que la classe* NP *est close par union et concaténation.*

Exercice 4 *(corrigé page 245) Prouver que si* NP *n'est pas égal à son complément, alors* $P \neq NP$.

Exercice 5 *Prouver que si* P = NP *alors tout langage* $A \in P$ *sauf* $A = \emptyset$ *et* $A = \Sigma^*$ *sont* NP*-complets.*

Exercice 6 *(corrigé page 245) La sécurité des cryptosystèmes est basée sur l'existence de fonctions* à sens unique. *Dans le cadre de cet exercice, on définit une fonction* à sens unique *comme une fonction $f : M^* \to M^*$ calculable en temps polynomial (déterministe) qui préserve les longueurs et qui n'est pas inversible en temps polynomial. Ici M^* désigne les mots sur l'alphabet fini M. Une fonction préserve les longueurs si la longueur de $f(x)$, notée $|f(x)|$, est la même que celle de x, notée $|x|$, pour tout $x \in M^*$. Ici* inversible *signifie : étant donné y, produire soit un x tel que $f(x) = y$, ou dire qu'aucun tel x existe.*

L'objectif de cet exercice est de montrer qu'il existe des fonctions à sens unique *si et seulement si* $\mathrm{P} \neq \mathrm{NP}$.

On souhaite prouver d'abord que si $\mathrm{P} = \mathrm{NP}$ *alors il n'existe pas de fonction* à sens unique.

Supposons l'alphabet $M = \{0,1\}$ soit binaire. Soit f une fonction $f : M^* \to M^*$ calculable en temps polynomial (déterministe) qui préserve les longueurs et qui est inversible.

1. *Expliquer pourquoi le raisonnement suivant est incorrect ou incomplet : "dans* NP*, on peut deviner x et vérifier que $f(x) = y$. Puisque* $\mathrm{P} = \mathrm{NP}$*, on peut le faire de façon déterministe".*
2. *On note $\leq$ pour la relation préfixe sur les mots sur M : autrement dit, $u \leq x$ si et seulement s'il existe v tel que $x = uv$.*

   Montrer que le langage

   $$B = \{\langle y, u\rangle | \exists x\ |x| = |y|,\ f(x) = y,\ u \leq x\}$$

   est dans NP.
3. *En déduire que si* $\mathrm{P} = \mathrm{NP}$ *alors il n'existe pas de fonction* à sens unique.
4. *Soit ϕ un circuit booléen avec, disons, m variables. Soit t une chaîne de longueur m qui dénote une affectation de ses variables qui rend ϕ vraie. On considère la fonction*

   $$f(\phi \# t) = \begin{cases} \phi \# 1^{|t|} & \text{si } \phi(t) \text{ est vrai} \\ \phi \# 0^{|t|} & \text{si } \phi(t) \text{ est faux,} \end{cases}$$

   où # désigne une lettre particulière, et $\phi(t)$ désigne la valeur du circuit ϕ avec l'entrée t. Montrer que f préserve les longueurs et est calculable en temps polynomial.
5. *Montrer que le circuit booléen ϕ est satisfiable si et seulement s'il existe x tel que $f(x) = \phi \# 1^{|t|}$.*
6. *Montrer que f est inversible en temps polynomial si et seulement si* $\mathrm{P} = \mathrm{NP}$.
   Conclure qu'il existe des fonctions à sens unique si et seulement si $\mathrm{P} \neq \mathrm{NP}$.

7 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture des ouvrages [Sipser, 1997], [Papadimitriou, 1994] ainsi que le livre
[Lassaigne & de Rougemont, 2004].

Un ouvrage de référence et contenant les derniers résultats du domaine est [Arora & Barak, 2009].

Bibliographie Ce chapitre contient des résultats standards en complexité. Nous nous sommes essentiellement inspirés des livres [Sipser, 1997], [Poizat, 1995], ainsi que [Papadimitriou, 1994]. La dernière partie discussion est reprise ici de sa formulation dans [Arora & Barak, 2009].

Index

Bibliographie

[Arora & Barak, 2009] Arora, S. & Barak, B. (2009). *Computational Complexity : A Modern Approach*. Cambridge University Press. https://doi.org/10.1017/cbo9780511804090

[Kleinberg & Tardos, 2006] Kleinberg, J. M. & Tardos, É. (2006). *Algorithm design*. Addison-Wesley.

[Lassaigne & de Rougemont, 2004] Lassaigne, R. & de Rougemont, M. (2004). *Logic and complexity*. Discrete Mathematics and Theoretical Computer Science. Springer. https://doi.org/10.1007/978-0-85729-392-3

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Poizat, 1995] Poizat, B. (1995). *Les petits cailloux : Une approche modèle-théorique de l'Algorithmie*. Aléas Editeur.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.