Sylvain Perifel

COMPLEXITÉ ALGORITHMIQUE

Avant-propos

Le projet de ce livre a germé au cours d'un groupe de lecture de l'excellent ouvrage d'Arora et Barak [AB09]. Une conjonction de plusieurs facteurs m'a décidé à franchir le pas. Il y avait premièrement la volonté de contribuer modestement au développement de l'enseignement et de la recherche en complexité en France. On pouvait voir l'absence d'ouvrage avancé en langue française sur ce sujet comme la cause ou la conséquence d'un domaine insuffisamment développé. J'ai préféré y voir la cause. Je suis convaincu qu'on a dans notre pays le potentiel pour contribuer significativement à l'avancée de ce beau sujet.

Il y avait aussi le fait que le livre anglophone « de référence » d'Arora et Barak est destiné au public d'outre-Atlantique et ne correspond pas au style d'ouvrage utile aux enseignants de ce côté-ci de l'océan. Les preuves sont trop approximatives, trop de détails sont omis ou trop de petites erreurs gênent la préparation d'un cours formel au tableau. Entendons-nous bien, le livre en question est excellent, donne les bonnes intuitions, couvre un matériel considérable et le présent ouvrage lui doit beaucoup, mais il n'est guère adapté pour les cours formels français. Je ne prétends certes pas que mon livre est exempt d'erreurs, loin de là malheureusement, mais j'espère avoir évité les imprécisions qui peuvent nuire à la compréhension du lecteur.

Il y avait enfin la maison d'édition Ellipses qui m'a contacté au moment opportun et qui a lancé ce projet.

Remerciements

Je tiens donc à remercier Paul de Laboulaye de m'avoir proposé de publier mon ouvrage aux éditions Ellipses. Je remercie également Pascal Koiran pour m'avoir donné le goût de la recherche en complexité.

Je l'ai dit, ce livre doit beaucoup à l'ouvrage d'Arora et Barak, dont s'inspire la présentation de quelques résultats (au chapitre 12 notamment). Je souhaite donc remercier ses auteurs pour avoir écrit un livre couvrant les développement récents de la théorie de la complexité. Je me rends compte maintenant du travail que cela représente.

Évidemment, la rédaction d'un livre n'est pas qu'une aventure solitaire et de nombreuses discussions avec Hervé Fournier, Guillaume Malod et Thomas Colcombet notamment sont venues enrichir cet ouvrage, un grand merci à eux. Merci également à Olivier Serre pour ses lignes de code LaTeX, à Peter Habermehl pour m'avoir débloqué sur deux ou trois questions, et à Christian Choffrut pour avoir donné les TD de mon cours de complexité.

Je dois beaucoup également à tous les relecteurs qui ont débusqué de nombreuses fautes et ont permis d'améliorer sensiblement la présentation de ce livre : je remercie chaleureusement (dans l'ordre alphabétique) Laurent Bienvenu, Sylvain Bouveret, Florent Capelli, Arnaud Durand, Hervé Fournier, Bruno Grenet, Guillaume Lagarde, Mathieu Laurière, Guillaume Malod, Stefan Mengel, Charles Paperman, Natacha Portier, Yann Strozecki et Sébastien Tavenas. S'il reste des erreurs, c'est évidemment ~~leur~~ ma faute.

Enfin, et surtout, merci à mon épouse qui me supporte et me soutient quotidiennement.

Paris, novembre 2013.

Table des matières

Introduction

Pour bien comprendre l'évolution de la discipline, il est utile de débuter par un bref rappel historique. Pendant plusieurs millénaires, les mathématiciens se sont contentés d'une notion intuitive, informelle, d'algorithme : une « méthode effective de calcul », ou encore un « processus de résolution d'un problème par le calcul ». Ce fut un long chemin pour arriver à la formalisation de cette notion au XXème siècle, puis à l'étude de la complexité des algorithmes qui est l'objet de ce livre.

Algorithmes

Les premières traces d'algorithmes ont été retrouvées chez les Babyloniens (l'actuel Irak) au deuxième millénaire avant notre ère et étaient principalement des méthodes de calcul pour le commerce et les impôts. Il faut attendre le troisième siècle avant J.-C. en Grèce pour l'apparition du fameux algorithme d'Euclide pour le calcul du pgcd : on peut considérer qu'il s'agit du premier algorithme « moderne » et il est tout à fait remarquable qu'il soit toujours utilisé de nos jours.

Mille ans plus tard, au IXème siècle ap. J.-C., Al Khuwarizmi, un mathématicien perse (actuel Iran), publie un ouvrage consacré aux algorithmes : l'étymologie du terme « algorithme » vient du nom de ce mathématicien. On commence en effet à étudier les algorithmes en tant que tel, mais il faudra encore 800 ans avant que l'Occident continue cette étude.

Aux XVIIème et au XVIIIème siècles, des savants comme Pascal ou Leibniz construisent des machines à calculer mécaniques (la Pascaline en 1642) ou des automates, ouvrant ainsi la voie à l'automatisation du calcul et à la recherche d'algorithmes efficaces. En 1837, Wantzel résout le problème de savoir quelles longueurs il est possible de construire à la règle et au compas : il s'agit de connaître la capacité des algorithmes dont les opérations de bases sont celles de la règle et du compas. Ce sont les prémices de la calculabilité.

Calculabilité

La volonté de formalisation des mathématiques à la fin du XIXème siècle et au début du XXème amène la notion d'axiomes et de systèmes de preuve, ce qui permet d'envisager une « automatisation » des mathématiques. Dans ce cadre, il est alors naturel de se demander si l'on peut « tout » résoudre par algorithme.

Hilbert pose en 1900 dans son dixième problème la question de trouver un algorithme déterminant si une équation diophantienne a une solution. Malgré des voix plus sceptiques, il est alors clair pour Hilbert qu'un tel algorithme doit exister. Il va plus loin en 1928 avec son *Entscheidungsproblem* (« problème de décision ») : il demande un algorithme capable de décider si un énoncé mathématique est vrai.

En l'absence d'une solution positive, de nombreux mathématiciens tentent de formaliser ce qu'on entend par « algorithme » afin de mieux cerner la question. Plusieurs formalismes voient le jour dans les années 1930 et tous sont prouvés équivalents, notamment le λ-calcul de Church et la machine de Turing. Ce dernier modèle, par sa simplicité et son évidence, emporte l'adhésion et les algorithmes ont maintenant une définition formelle.

Or c'est un choc, ou au moins une surprise, qui attend Hilbert : Church [Chu36] en 1936 avec son modèle, et Turing [Tur37] en 1937 avec le sien, montrent qu'il n'existe pas d'algorithme pour l'Entscheidungsproblem : c'est le début de la calculabilité, qui s'attache à comprendre ce que sont capables de réaliser les algorithmes.

Des techniques de plus en plus sophistiquées permettent d'obtenir d'autres résultats d'indécidabilité (le fait pour un problème de ne pas avoir d'algorithme), comme par exemple Matiyasevich en 1970 qui pose la dernière pierre de la preuve qu'il n'existe pas d'algorithme pour décider si une équation diophantienne a une solution (dixième problème de Hilbert). Mais bien plus que l'indécidabilité, on généralise le modèle de calcul, on compare les différentes classes, on étudie l'aléatoire, etc.

Ces considérations, bien que très théoriques, ont contribué à la construction des premiers ordinateurs au début des années 1940, qui sont finalement très proches de la machine de Turing. Et c'est en quelque sorte le développement de ces ordinateurs qui donne naissance à la théorie de la complexité.

Complexité

La calculabilité s'attache à connaître ce qu'on peut résoudre par algorithme *quel que soit le temps d'exécution.* Or dans les années 1960, si les gros ordinateurs se sont diffusés, il n'en reste pas moins qu'ils sont très lents et ne disposent pas de beaucoup de mémoire. Les chercheurs s'intéressent donc naturellement à *l'efficacité* des algorithmes : quel algorithme puis-je faire tourner sur cette machine sans que le résultat mette un an à arriver ? Quel algorithme puis-je faire tourner sans dépasser la capacité de la mémoire de la machine ?

On trouve des réflexions théoriques très profondes de ce genre dans une lettre de Gödel à von Neumann en 1956, qui demande s'il existe un algorithme quadratique pour le problème SAT : tous les ingrédients de la question « $\mathsf{P} = \mathsf{NP}$? » sont déjà présents dans son esprit. Malheureusement, von Neumann mourant n'a pas pu prendre cette lettre en considération et elle n'a été retrouvée que trente ans plus tard.

Si l'on exclut cette lettre, la première référence explicite aux algorithmes fonctionnant en temps polynomial comme définition d'algorithmes « efficaces » se trouve dans les articles de 1965 de Cobham [Cob65] et d'Edmonds [Edm65]. Puis le papier de Hartmanis et Stearns [HS65] lance réellement le domaine de la complexité en montrant que certains problèmes ne peuvent pas être résolus en un temps donné (théorèmes de hiérarchie).

Quant à la question « P = NP ? », elle prend de l'importance dès le début des années 1970 avec l'article de Cook [Coo71] en 1971 (ou, de l'autre côté du rideau de fer, l'article de Levin [Lev73] en 1973) montrant la NP-complétude de SAT, et avec d'autres résultats de NP-complétude en 1972 par Karp [Kar72] pour de nombreux problèmes ayant une réelle importance.

La théorie de la complexité a commencé en adaptant les méthodes de la calculabilité au cas du temps de calcul borné. Par exemple, on retrouve de nombreux ingrédients issus de la calculabilité dans la démonstration du théorème 3-AK de Ladner datant de 1975. Puis à partir des années 1980, des techniques spécifiques ont vu le jour, notamment pour les bornes inférieures sur les circuits booléens. Les années 1990 ont vu en particulier l'essor des protocoles interactifs qui ont permis de nombreuses avancées, et également une réflexion profonde sur la possibilité de déterminiser des algorithmes probabilistes. Les années 2000 continuent la recherche de méthodes permettant de contourner les obstacles pour résoudre les grandes questions ouvertes comme « P = NP ? », mais plusieurs résultats nouveaux ont aussi réussi à combiner de manière heureuse différentes méthodes conçues au cours des décennies précédentes.

Néanmoins, les grandes questions restent sans réponse. La quête n'est pas finie.

Organisation du livre

Afin de présenter les notions et les résultats dans un ordre cohérent, le livre suit pratiquement la chronologie historique. Le premier chapitre commence donc par définir le modèle de calcul utilisé, la machine de Turing, dans une version qui permettra de traiter à la fois la complexité en temps et celle en espace. Les machines non déterministes sont également introduites. Outre ces définitions, il donne des exemples et plusieurs propriétés des machines (simulations notamment) et construit des machines déterministes universelles plus ou moins efficaces.

Le chapitre 2 contient les premiers résultats sur la complexité en temps, déterministe et non déterministe : définitions, accélération linéaire, machine universelle non déterministe, théorèmes de hiérarchie, classes P et NP et exemples, etc.

Puis on aborde la NP-complétude au chapitre 3 : complétude de SAT, autres réductions permettant de montrer la complétude de différents problèmes, théorème de Ladner et théorème de Mahaney.

Le chapitre 4 fait un travail similaire au chapitre 2 pour la complexité en espace : définitions et résultats de base, PSPACE-complétude du problème QBF et NL-complétude du problème d'accessibilité dans un graphe orienté, le théorème de Savitch et pour finir le théorème d'Immerman-Szelepcsényi.

Le chapitre 5 traite des questions d'uniformité et de circuits : calculs avec conseil, circuits booléens, simulations de machines de Turing, uniformité des circuits, et enfin présentation des circuits arithmétiques pour calculer des polynômes.

Le chapitre 6 concerne les algorithmes probabilistes : définitions, réduction d'erreur, théorème de hiérarchie, comparaison avec les classes déterministes et avec les circuits, présentation du problème d'identité de polynômes que nous reverrons au chapitre 12.

Nous introduisons au chapitre 7 la notion de calcul avec oracle et de relativisation pour explorer notamment les limites de la diagonalisation. Nous montrons que les questions « $\mathsf{P} = \mathsf{NP}$? » et « $\mathsf{EXP} = \mathsf{NP}$? » ne relativisent pas et nous discutons la difficulté de définir ce qu'est une preuve par diagonalisation.

Le calcul avec oracle nous permet aussi d'introduire la hiérarchie polynomiale au chapitre 8. Nous donnons des problèmes complets aux différents niveaux, comparons ces classes avec les classes probabilistes et montrons le théorème de Karp et Lipton. Enfin, nous donnons une borne inférieure sur le temps et l'espace conjugués nécessaires pour résoudre les problèmes de $\mathsf{NTIME}(n)$, en utilisant les techniques vues à ce chapitre.

Avec le chapitre 9 sur la complexité du comptage, nous abordons des sujets plus techniques. Après des propriétés de clôture des classes de comptage, nous montrons le théorème de Toda, puis la complétude du permanent pour la classe $\sharp\mathsf{P}$.

Le chapitre 10 introduit les protocoles interactifs. Nous donnons le théorème de Shamir ($\mathsf{IP} = \mathsf{PSPACE}$), montrons qu'il ne relativise pas, définissons les classes Arthur-Merlin avec toutes leurs propriétés, montrons l'équivalence entre bits aléatoires publics ou privés. Nous appliquons ces techniques au problème de l'isomorphisme de graphe pour montrer qu'il n'est probablement pas NP-complet. Enfin, nous décrivons le théorème PCP sans le prouver et donnons une application à l'inapproximabilité.

Le chapitre 11 présente plusieurs bornes inférieures sur les circuits : des classes qui n'ont pas de circuits de taille n^k pour k fixé, des classes qui n'ont pas de circuits de taille polynomiale sous certaines conditions, des bornes inférieures sur la taille de circuits « restreints » (monotones ou de profondeur constante), et des bornes inférieures sur la taille de circuits arithmétiques pour calculer des polynômes.

Enfin, le chapitre 12, très technique, montre le lien entre dérandomisation et bornes inférieures non uniformes. Pour montrer que des bornes inférieures permettent de dérandomiser des algorithmes, on utilise des codes correcteurs d'erreurs très efficaces dont la construction constitue une bonne part du contenu technique du chapitre. La preuve de la « réciproque » permet de combiner de nombreuses idées vues précédemment dans le livre.

Une annexe (chapitre A) donne les notions de bases en probabilités utiles pour cet ouvrage, ainsi que quelques résultats d'arithmétique.

Des exercices sont proposés tout au long des chapitres lorsqu'ils illustrent directement le propos et quelques autres exercices et problèmes sont proposés en annexe (chapitre B).

Outre les rappels mathématiques et les exercices, l'annexe contient aussi des conseils aux enseignants afin d'utiliser ce livre au mieux (chapitre D), ainsi qu'une liste et des figures récapitulatives des différentes classes de complexité abordées (chapitre C), qu'il ne faut pas hésiter à consulter lorsqu'une nouvelle classe est introduite.

Plusieurs ouvrages peuvent venir compléter celui-ci. En langue anglaise, les livres de Sipser [Sip97], de Papadimitriou [Pap94], et de Balcázar, Díaz et Gabarró [BDG88], bien qu'ils commencent à dater, sont une bonne entrée en matière. Celui de Wagner et Wechsung [WW86], datant des années 1980, est plus complet. Celui d'Hemaspaandra et Ogihara [HO02] traite de nombreux sujets avancés avec tous les détails mais sa lecture est plus ardue. Deux livres récents sont à recommander pour couvrir les avancées de la com-

plexité « moderne » : celui de Goldreich [Gol08] et celui d'Arora et Barak [AB09] dont nous avons parlé à l'avant-propos. Et nous omettons certainement d'autres excellents ouvrages. En langue française, les ouvrages de Dehornoy [Deh93], de Carton [Car08], de Lavallé [Lav08] ou de Rey [Rey04] traitent bien les sujets de base et sont donc à lire sans hésiter, mais ils ne parlent guère de sujets avancés. Le point de vue de la logique sur la complexité, en particulier la complexité descriptive, est plus développé dans les ouvrages de Lassaigne et de Rougemont [LR96] (dans la langue de Molière) et de Immerman [Imm99] (dans celle de Shakespeare). Enfin, si besoin il est certainement valable de compléter la lecture du présent livre par celle d'un ouvrage d'introduction à la calculabilité, comme celui de Wolper [Wol06] ou de Cooper [Coo04], voire d'entreprendre la lecture des volumes plus avancés d'Odifreddi [Odi92].

Nous commençons à la page xv par un préliminaire rappelant quelques notations usuelles que nous utiliserons, avant de débuter le cœur de l'ouvrage proprement dit.

Préliminaires et notations

Nous n'utilisons dans ce livre que des notions et notations usuelles ; néanmoins, pour lever toute ambiguïté, nous rappelons ici les principales conventions.

Alphabets, mots, langages

Σ désignera un alphabet fini, c'est-à-dire un ensemble fini de symboles. À partir de Σ, on construit des mots qui sont des suites finies de symboles : la taille d'un mot x, notée $|x|$, est le nombre de symboles qui le constituent. L'unique mot de taille nulle est le mot vide que l'on désignera par ϵ (mais, selon l'habitude en mathématique, cette notation est souvent utilisée aussi pour désigner une petite constante réelle positive). Si $x = x_1 \dots x_n$ et $y = y_1 \dots y_m$ sont deux mots, on dit que x est un préfixe de y, noté $x \sqsubseteq y$, si $m \geqslant n$ et $x_1 \dots x_n = y_1 \cdots y_n$.

Si A est un ensemble de mots, on note

$$A^\star = \{u_1 \dots u_k \mid k \in \mathbb{N} \text{ et } u_1, \dots, u_k \in A\}$$

l'ensemble de toutes les concaténations possibles d'un nombre arbitraire de mots de A. L'ensemble de tous les mots sur l'alphabet Σ est donc noté $\Sigma^\star$. Un langage est un ensemble de mots $L \subseteq \Sigma^\star$. Pour un entier $n \in \mathbb{N}$, on notera $L^{=n}$ et $L^{\leqslant n}$ les ensembles (finis) des mots de L de longueur respectivement n et au plus n, et $L^{\geqslant n}$ l'ensemble des mots de L de longueur au moins n.

Ordre lexicographique

Si l'alphabet $\Sigma = \{a_1, \dots, a_k\}$ est muni d'un ordre quelconque, disons $a_1 < a_2 < \cdots < a_k$, nous appellerons *l'ordre lexicographique* l'ordre sur $\Sigma^\star$ défini inductivement par :

- si $|x| < |y|$ alors $x <_{\text{lex}} y$;
- si $|x| = |y| = 1$ alors $<_{\text{lex}}$ est égal à l'ordre $<$ sur Σ ;
- sinon, si $x = x_1 \dots x_n$ et $y = y_1 \dots y_n$ alors $x <_{\text{lex}} y$ si $x_1 < y_1$ ou si $x_1 = y_1$ et $x_2 \dots x_n <_{\text{lex}} y_2 \dots y_n$.

En d'autres termes, on ordonne les mots par taille, puis entre des mots de même taille il s'agit de l'ordre du dictionnaire. Cet ordre est parfois appelé « quasi-lexicographique »

car ordonner d'abord par longueurs n'est pas la définition habituelle de l'ordre lexicographique, mais cela nous paraît plus maniable pour la suite.

Suites, intervalles, ensembles

Dans ce livre, on désignera par $(x_i)_{i\in\mathbb{Z}}$ (respectivement $(x_i)_{i\in\mathbb{N}}$) une suite infinie d'éléments x_i indicés par les entiers relatifs (resp. par les entiers naturels). Si le contexte est clair, on pourra omettre la référence à $\mathbb{Z}$ ou $\mathbb{N}$ et parler simplement de la suite (x_i).

La plupart du temps, la notation $[a,b]$ désignera l'intervalle *entier* de bornes a et b, c'est-à-dire l'ensemble $\{\lceil a\rceil, 1+\lceil a\rceil, \ldots, \lfloor b\rfloor\} \subseteq \mathbb{Z}$. Les rares fois où une telle notation désignera un intervalle réel seront claires d'après le contexte.

Si q est une puissance d'un nombre premier, on notera $\mathbb{F}_q$ l'unique corps à q éléments (si q est premier, $\mathbb{F}_q$ est donc l'ensemble des entiers modulo q). Pour un ensemble L et un élément x, on notera $[x \in L]$ (respectivement $[x \notin L]$) le booléen qui vaut 1 ssi $x \in L$ (resp. $x \notin L$) [1].

Fonctions entières

Les informaticiens ont pris l'habitude d'écrire $f(n)$ pour parler d'une fonction $f : \mathbb{N} \to \mathbb{N}$, plutôt que simplement f : ici, n représente la variable et non un entier fixé. Cette notation sert implicitement à savoir que le domaine de f est $\mathbb{N}$, un peu comme en mathématiques lorsqu'on parle d'un polynôme $P(X)$.

Cependant, n est également utilisé pour parler de la taille de l'entrée x d'un problème et peut donc aussi représenter un entier.

Dans tout ce livre, sauf mention contraire on notera log le logarithme *en base* 2. Parfois par abus de notation on pourra même écrire $\log(n)$ pour $\lfloor\log(n)\rfloor$ car nous considérerons la plupart du temps des fonctions entières.

Ordre de grandeur des fonctions

Si $f : \mathbb{N} \to \mathbb{N}$ est une fonction, on note $O(f)$ l'ensemble des fonctions $g : \mathbb{N} \to \mathbb{N}$ telles que

$$\exists C > 0\ \exists N \in \mathbb{N}\ \forall n \geqslant N \quad g(n) \leqslant C f(n)$$

(à partir d'un certain rang, g est plus petite que f à une constante multiplicative près). La notation $o(f)$ désigne l'ensemble des fonctions $g : \mathbb{N} \to \mathbb{N}$ telles que

$$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall n \geqslant N \quad g(n) \leqslant \epsilon f(n)$$

(g est négligeable devant f). Les ensembles $\Omega(f)$, $\omega(f)$ et $\Theta(f)$ sont alors définis par :

$$g \in \Omega(f) \iff f \in O(g)$$

$$g \in \omega(f) \iff f \in o(g)$$

1. On trouve aussi la notation équivalente $\delta_{x\in L}$ qu'on n'utilisera pas ici.

$$g \in \Theta(f) \iff f \in O(g) \text{ et } g \in O(f).$$

Si $h : \mathbb{N} \to \mathbb{N}$ est une fonction, $h(O(f))$ désigne l'ensemble des fonctions $h \circ g$ où $g \in O(f)$; de même pour les autres ensembles ($o(f)$, etc.). Enfin, par abus de notation et selon l'usage habituel, on notera $g = O(f)$ plutôt que $g \in O(f)$, etc.

Mots, uples, variables

Les lettres x, y, z, u, v seront le plus souvent employées pour désigner des mots sur un alphabet Σ, ou des variables ou des uples de variables d'un polynôme ou d'une autre fonction. Les fonctions sur les entiers ou sur les mots seront souvent notées f ou g, les polynômes p, q ou P, Q, mais p et q pourront aussi désigner des entiers premiers.

Les lettres n, m seront plutôt utilisées pour la longueur d'un mot, la taille de l'entrée d'un problème ou la variable d'une fonction $f : \mathbb{N} \to \mathbb{N}$. Les lettres i, j, k désigneront principalement des indices entiers ; α, β, γ et a, b, c des constantes réelles ou entières. On utilisera de préférence t pour un temps d'exécution ou pour la taille d'un circuit et s pour l'espace d'un calcul. Les mots aléatoires seront souvent notés r. Rappelons enfin que ϵ désignera la plupart du temps une petite constante réelle positive, mais parfois aussi le mot vide.

Divers

Un graphe G est donné par l'ensemble V de ses sommets et l'ensemble E de ses arêtes, $E \subseteq V \times V$: on notera donc $G = (V, E)$.

Sauf mention contraire, la notation $\Pr_{r \in E}(A)$ désignera la probabilité de l'événement A *selon la distribution uniforme* sur l'ensemble fini E. Un rappel basique sur les probabilités discrètes est donné à l'annexe A.

Enfin, puisque le cas se produira souvent, nous abrégerons l'expression « la machine M fonctionnant en temps polynomial » par « la machine polynomiale M », malgré l'abus de langage.

1 Le modèle de calcul

Le propos de la complexité algorithmique est de mesurer la difficulté d'un problème à l'aune de l'efficacité des algorithmes pour le résoudre. Dans ce premier chapitre, nous allons définir formellement ce que nous entendons par problèmes et algorithmes.

Aujourd'hui nous avons tous une idée intuitive de ce qu'est un algorithme car nous manipulons chaque jour des ordinateurs ; toutefois, il n'est pas si aisé d'en donner une définition formelle. La définition habituelle que nous allons suivre ici passe par les machines de Turing. Il est remarquable que ce modèle de calcul ait été proposé par Turing dès 1937 dans son article [Tur37], *avant* la naissance des ordinateurs, à laquelle il a d'ailleurs largement contribué.

La définition de Turing est encore utilisée actuellement car, afin d'étudier les algorithmes, on souhaite un modèle théorique aussi simple que possible, d'une part, et d'autre part nous verrons que ce modèle rudimentaire reste capable d'effectuer les mêmes tâches que nos ordinateurs. À la même époque, d'autres formalismes équivalents ont été introduits (λ-calcul par Church [Chu36], fonctions récursives par Herbrand et Gödel, etc.) mais la machine de Turing a l'avantage de permettre la définition aisée du temps et de l'espace utilisés lors d'un calcul. Des modèles plus proches de nos ordinateurs existent également (machines RAM par exemple) mais, pour étudier la complexité algorithmique, la connaissance des machines de Turing est devenue trop usuelle pour pouvoir s'en passer — bien que ce ne soit pas toujours le modèle idéal. [1]

Une telle machine manipule des suites finies de symboles, c'est-à-dire qu'on représente par des « mots » les données à traiter (nous aurons donc besoin d'un codage des données en des mots). Elle effectue alors un calcul sur un mot qu'on lui fournit en entrée et renvoie le résultat du calcul sous la forme d'un nouveau mot de sortie comme illustré à la figure 1.1.

1. On remarquera également qu'il est malaisé d'employer un autre modèle de calcul pour obtenir exactement la définition habituelle des concepts que nous aborderons dans ce livre.

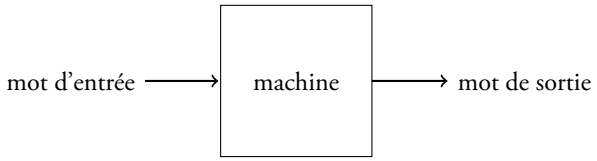


FIGURE 1.1 – Principe d'une machine de Turing.

Nous allons d'abord nous attarder sur la notion de problème avant de définir la machine de Turing et de voir qu'une telle machine permet bien de réaliser toutes les opérations qu'on souhaite possibles pour un algorithme.

1.1 Problèmes, langages et codage

1.1.1 Codage

Chacun sait que les ordinateurs actuels ne manipulent que des 0 (absence de courant électrique) et des 1 (présence du courant). Et pourtant, dans nos programmes nous pouvons parler d'objets de diverse nature : des entiers, des rationnels, des matrices, des graphes, etc. Pour cela, il faut *coder* ces objets par une suite de 0 et de 1.

Dans le cadre plus général de ce livre, nous ne nous contenterons pas de deux symboles 0 et 1, mais d'un *alphabet* fini quelconque. Un alphabet est simplement l'ensemble des symboles autorisés. Ainsi, l'alphabet binaire est $\{0,1\}$ mais on pourrait aussi travailler sur l'alphabet $\{a,b,c,\ldots,z\}$ par exemple, ou bien $\{a,b,\#\}$ pour parler des mots (c'est-à-dire des suites finies de symboles) composés des lettres a, b et # (le symbole # servira la plupart du temps de délimiteur). Dès lors, nous devrons coder les objets à manipuler par des mots sur un alphabet fini Σ.

La taille d'un mot x, notée $|x|$, est le nombre de symboles que x contient et joue une grande importance puisque nous verrons plus tard que le nombre d'étapes de calcul d'une machine sera considéré comme une fonction de la taille du mot d'entrée. Ainsi, nous verrons par la suite que *la difficulté d'un problème dépend du codage utilisé*. Il faudrait donc toujours préciser ce codage. Néanmoins, il y a plusieurs codages usuels que nous utiliserons par défaut, sauf mention contraire :

- les entiers codés en binaire avec les symboles 0 et 1 (par exemple, l'entier 12 sera codé 1100) ;
- de même pour les rationnels, le numérateur et le dénominateur séparés par un délimiteur (par exemple, 4/3 sera codé 100#11) ;
- une matrice donnée par la liste de ses coefficients séparés par des délimiteurs ;
- un graphe donné par sa matrice d'adjacence ;
- un polynôme donné par la liste de ses coefficients (séparés par des délimiteurs), etc.

Mais il existe également des codages moins usuels, en voici deux exemples :

- les entiers codés en unaire avec le seul symbole 1 (le nombre de 1 donne la valeur de l'entier, par exemple 5 sera encodé 11111) : bien que non usuel, nous rencontrerons ce codage plus tard ;
- un polynôme de degré d donné par la liste des valeurs qu'il prend sur les entiers de 0 à d.

Enfin, on aura régulièrement besoin de coder des couples ou des n-uplets de mots. Si x et y sont deux mots sur un alphabet Σ, on désignera par (x,y) le codage du couple (x,y), c'est-à-dire, selon les cas :

- si l'alphabet peut être augmenté (ce qui sera le cas la plupart du temps), (x,y) désignera le mot $x\#y$ sur l'alphabet $\Sigma\cup\{\#\}$ et la taille de (x,y) est alors $|x|+|y|+1$: par exemple, si $x=abaa$ et $y=bba$ alors le code de (x,y) est $abaa\#bba$ de taille 8 ;
- lorsque l'alphabet peut être augmenté, on peut même obtenir un codage de taille $|x|+|y|$ exactement en codant x sur un alphabet suivi de y sur un alphabet disjoint : par exemple, $(abba,caba)$ sera codé $abbaCABA$;
- si l'alphabet ne peut être augmenté, on supposera qu'il possède au moins deux symboles [2] que l'on notera 0 et 1 ; il y a alors au moins deux codages possibles :
  - si $x=x_1\dots x_n$ et $y=y_1\dots y_m$ alors (x,y) désignera le mot
    $$x_1 0 x_2 0 x_3 0 \dots x_{n-1} 0 x_n 1 y_1 y_2 \dots y_m,$$
    c'est-à-dire qu'on intercale 0 entre les lettres de x et 1 pour séparer x et y, puis on écrit y normalement : la taille de (x,y) est alors $2|x|+|y|$,
  - pour un codage plus compact, on donnera d'abord la taille de x (pour savoir où séparer les deux mots) puis la concaténation des deux mots : plus précisément, si $t=|x|$ est la taille de x codée en binaire sur $\{0,1\}$, $t=t_1\dots t_l$, alors (x,y) désignera le mot $t_1 0 t_2 0 \dots t_{l-1} 0 t_l 1 x y$; la taille de (x,y) est alors $|x|+|y|+2\lceil\log(1+|x|)\rceil$. [3]

On peut aisément généraliser ces codages aux n-uplets pour $n>2$. Selon les cas, nous utiliserons le codage le plus adapté sans forcément l'expliciter si le contexte permet de le deviner. Ainsi, la plupart du temps nous supposerons dans nos raisonnements que le codage du couple (x,y) a pour taille $|x|+|y|$.

Maintenant que nous savons coder nos objets sur un alphabet fini, voyons comment formaliser les problèmes qu'on se pose sur ces objets.

1.1.2 Problèmes et langages

Le but d'un programme ou d'un algorithme est de résoudre un problème. La plupart du temps, et nous nous placerons dans ce cadre, le problème est générique mais on le résout

2. Le cas unaire peut être traité par une injection usuelle de $\mathbb{N}^2$ dans $\mathbb{N}$ mais il ne nous intéressera pas ici.
3. Rappelons que, sauf indication contraire, tous les logarithmes utilisés seront en base 2.

sur une *instance* particulière : par exemple, on ne peut répondre au problème « déterminer si un entier est premier » que sur un entier particulier qu'on nous donnerait en entrée ; ou encore, le problème « trier par ordre croissant une liste d'entiers » n'a de sens que sur une liste d'entiers sur laquelle travailler.

Un problème est donc composé de deux éléments : une entrée (ou *instance*) et une question ou une tâche à réaliser. Les deux exemples précédents se reformulent ainsi :

1. `PRIMALITÉ` :
   - *entrée :* un entier N ;
   - *question :* N est-il premier ?
2. `TRI` :
   - *entrée :* une liste d'entiers l ;
   - *tâche :* trier l par ordre croissant.

On distingue ainsi deux types de problèmes : ceux qui consistent à répondre par oui ou par non à une question donnée (dans l'exemple précédent, déterminer si un entier est premier), qu'on appelle *problèmes de décision* ; et ceux qui consistent à produire un nouvel objet (dans l'exemple précédent, produire la nouvelle liste triée), qu'on appelle *problèmes d'évaluation*. En effet, ces derniers consistent à évaluer une fonction : dans l'exemple, il s'agit d'évaluer la fonction qui à une liste d'entiers associe la liste triée.

Pour spécifier un problème de décision, il suffit de donner les mots pour lesquelles la réponse est « oui » : un tel problème est donc caractérisé par un ensemble d'instances positives. Plus précisément, un problème de décision donné sous la forme « entrée/question » peut facilement être transformé sous la forme d'un ensemble

$$L = \{x \in \Sigma^\star \mid x \text{ code une instance valide et la réponse à la question sur } x \text{ est positive}\}$$

qu'on appellera « langage ». Un problème d'évaluation est quant à lui caractérisé par la fonction qu'il calcule. On obtient alors la définition suivante.

1-A Définition

- Un *langage* (ou problème de décision) sur un alphabet Σ est un ensemble de mots sur Σ. Un langage $L \subseteq \Sigma^\star$ définit le problème de décision dont les instances positives sont exactement les mots de L.
- Un problème d'évaluation sur un alphabet Σ est une fonction $f : \Sigma^\star \to \Sigma^\star$.

Lorsqu'un problème de décision est vu comme un langage L, la forme de l'entrée apparaît dans L par « x code une instance valide ». Il faudra donc toujours avoir à l'esprit que tester si l'entrée encode une instance valide fait partie de la complexité du problème ; mais en règle générale, ce test est très facile et rapide à réaliser et est donc omis des raisonnements.

Un peu de recul

Dans la suite, nous étudierons surtout les problèmes de décision (langages). Mais il faut garder à l'esprit qu'on peut la plupart du temps transformer un problème d'évaluation en un langage de « difficulté équivalente ».

Par exemple, pour transformer « trouver le plus petit diviseur non trivial de N », on ajoutera une entrée k pour obtenir la question « existe-t-il un diviseur non trivial de N qui soit inférieur à k ? ». Si l'on sait répondre à cette question, alors on peut trouver rapidement le plus petit diviseur non trivial de N : il suffit d'effectuer une recherche dichotomique en testant s'il existe un diviseur $\leqslant N/2$, puis si oui, s'il existe un divisieur $\leqslant N/4$ ou sinon s'il existe un divisieur $\leqslant 3N/4$, etc.

De manière presque équivalente, plutôt qu'un entier k on pourrait ajouter un mot a à l'entrée pour obtenir la question « existe-t-il un diviseur non trivial de N dont l'écriture binaire commence par a ? ». On effectuerait alors une recherche préfixe à la place de la recherche dichotomique.

Enfin, une autre façon de procéder est de demander le i-ème bit de la réponse. Ainsi, pour transformer « calculer la somme de deux entiers x et y » en un problème de décision, on ajoutera une entrée i pour obtenir la question « le i-ème bit de la somme $x+y$ est-il 1 ? ».

Passons maintenant à l'étude du modèle de calcul qui permet de travailler sur ces problèmes.

1.2 La machine de Turing

Il existe de nombreuses variantes de la machine de Turing : un ruban ou plusieurs, infini vers la droite seulement ou bi-infini, etc. Tous ces modèles sont équivalents mais le montrer n'est pas l'objet de ce livre (on trouvera les simulations entre modèles dans des livres de calculabilité par exemple). Nous nous contenterons donc de définir une unique version de machine de Turing, à plusieurs rubans bi-infinis.

1-B **Remarque** Nous avons pris le parti de définir rigoureusement ce modèle de calcul ; dès lors, les définitions sont relativement laborieuses pour des concepts intuitifs. Une approche possible est de sauter le formalisme et de s'attarder sur la description informelle et les exemples.

1.2.1 Définition

Une machine de Turing (figure 1.2) est composée d'un nombre fini de rubans sur chacun desquels se déplace une tête de lecture/écriture. Les rubans sont composés de cases qui contiennent chacune un symbole : il s'agit de la mémoire de notre machine. C'est l'information contenue sur ces rubans qu'on va lire et modifier. Pour cela, les têtes de

lecture/écriture peuvent se déplacer sur les rubans et lire ou modifier le contenu de la case qu'elles visitent.

On distinguera trois types de rubans [4] :

- le ruban d'entrée, en lecture seule (la tête peut seulement lire les symboles et pas les modifier) et sur lequel est écrit le mot à traiter ;
- le ruban de sortie, en écriture seule (la tête ne peut qu'écrire des symboles sans lire le contenu des cases) et sur lequel on écrira le résultat du calcul ;
- les rubans de travail (en lecture et écriture) permettant de mener à bien le calcul.

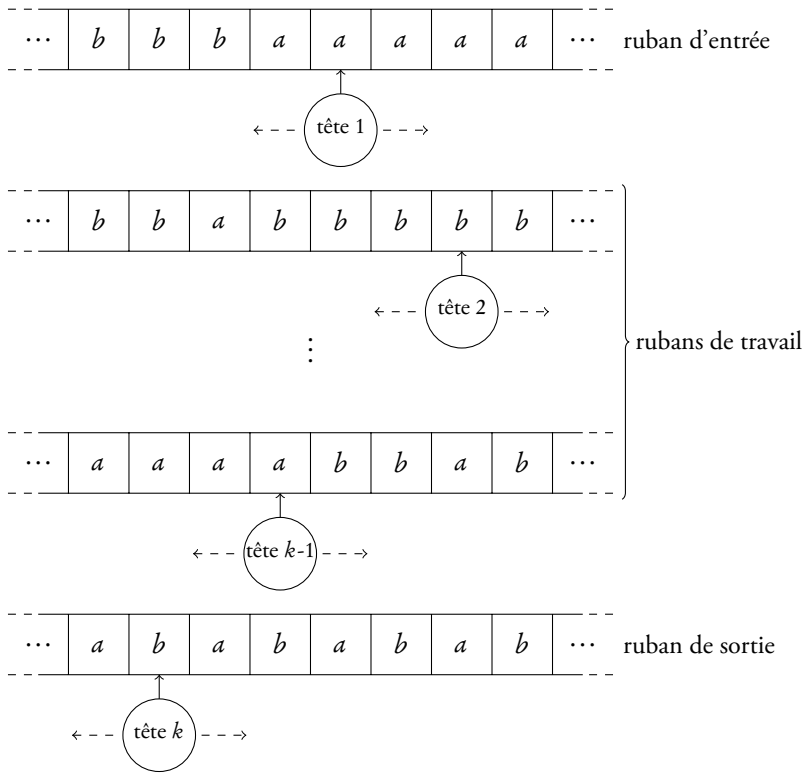


FIGURE 1.2 – Machine de Turing à k rubans.

Les têtes de lecture/écriture ont un pouvoir très limité : elles ne peuvent que se déplacer d'une case vers la gauche ou vers la droite et ont une mémoire finie, c'est-à-dire qu'elles ne peuvent prendre qu'un nombre fini d'états possibles. En d'autres termes, l'ensemble des

4. Ces conventions nous permettront de parler de manière homogène de complexité en espace ; la plupart du temps, nous aurions pu considérer des machines dont les rubans d'entrée et de sortie sont aussi en lecture/écriture.

têtes est un automate fini. Les rubans, quant à eux, sont infinis et représentent la mémoire de la machine. Si l'on indexe les cases d'un ruban par $\mathbb{Z}$, un déplacement à droite d'une tête de lecture correspond à incrémenter sa position ($+1$) et un déplacement à gauche à la décrémenter (-1) ; elle peut aussi rester sur place (0).

Un peu de recul

Nos ordinateurs n'ont pas une mémoire infinie : elle est certes conséquente mais finie. Ce sont donc des automates finis. Mais cette modélisation n'est pas satisfaisante : aucun programmeur n'accepterait d'utiliser un automate fini qui par essence est incapable de compter… La mémoire infinie d'une machine de Turing (ses rubans) semble refléter ce qu'un ordinateur est capable de faire et on peut évoquer deux raisons à cela :

- en pratique, la taille de la mémoire d'un ordinateur est suffisamment grande pour qu'elle soit considérée « infinie » ;
- s'il manque de mémoire pour réaliser un calcul, on peut toujours en ajouter (sous la forme d'un disque dur externe par exemple), donc la place dont on dispose est « virtuellement infinie »…

Au début du calcul, le ruban d'entrée contient le mot sur lequel la machine doit travailler. Ce mot occupe un nombre fini de cases, les autres restant vides : un symbole spécial « blanc » représente une case vide. Tous les autres rubans sont vides (c'est-à-dire ne contiennent que des symboles blancs). La tête du ruban d'entrée pointe sur la première lettre du mot d'entrée et l'état des têtes est un état particulier appelé état initial. Puis, en fonction de l'état des têtes et des symboles qu'elles lisent dans la case visitée, chaque tête remplace éventuellement par un nouveau symbole le contenu de la case, se déplace à gauche ou à droite et change d'état. Le calcul se termine lorsqu'un « état terminal » est atteint. Formalisons cette description avant de voir des exemples.

Pour définir formellement une machine de Turing, nous devons donc spécifier plusieurs éléments : les symboles utilisés sur les rubans, les états des têtes et la façon dont elles agissent sur les rubans.

1-C Définition (machine de Turing)

Une machine de Turing à $k \geqslant 2$ rubans est un octuplet $M = (\Sigma, \Gamma, B, Q, q_0, q_a, q_r, \delta)$ où :

- Σ est un ensemble fini non vide appelé *alphabet d'entrée* (ce sont les symboles utilisés pour écrire le mot d'entrée) ;
- Γ est un ensemble fini appelé *alphabet de travail*, tel que $\Sigma \subset \Gamma$ (ce sont les symboles utilisés dans les cases au cours du calcul) ;
- B est un symbole spécial *blanc* (représentant une case vide) tel que $B \in \Gamma \setminus \Sigma$;
- Q est un ensemble fini appelé *ensemble des états* (les états que peuvent prendre les têtes de lecture/écriture) ;

- $q_0 \in Q$ est un état spécial appelé *état initial* (indiquant l'état dans lequel les têtes commencent le calcul) ;
- $q_a \in Q$ et $q_r \in Q$ sont des états spéciaux appelé *états terminaux* (indiquant la fin du calcul) : q_a est l'état *d'acceptation* et q_r l'état de *rejet* ;
- $\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma^{k-1} \to Q \times \Gamma^{k-1} \times \{G, S, D\}^k$ est la *fonction de transition* décrivant le comportement des têtes :

$$\delta(q, a_1, \ldots, a_{k-1}) = (r, b_2, \ldots, b_k, d_1, \ldots, d_k)$$

si, lorsque les têtes sont dans l'état q et que la i-ème tête lit le symbole a_i ($i \leqslant k-1$ car la tête du ruban de sortie ne lit rien), alors le nouvel état des têtes est r, la tête i écrit le symbole b_i dans la case à la place de a_i ($i \geqslant 2$ car la tête du ruban d'entrée n'écrit rien) et elle se déplace dans la direction d_i (G à gauche, S sur place, D à droite).

1-D Remarques

- La fonction de transition δ n'est pas définie sur les états q_a et q_r puisque ce sont les états terminaux qui marquent la fin du calcul : la machine s'arrête dès que l'un d'eux est atteint.
- De plus, il est commode d'utiliser un alphabet de travail plus étoffé que celui d'entrée afin de disposer de symboles permettant de séparer des mots, de marquer des positions, etc.
- Enfin, insistons une dernière fois sur le fait que, dans le modèle que nous avons choisi, le ruban d'entrée est en lecture seule (donc la fonction de transition ne spécifie aucun symbole à écrire sur le ruban d'entrée) et le ruban de sortie en écriture seule (donc la fonction de transition ne spécifie aucun symbole à lire sur le ruban de sortie).

Les rubans de M sont infinis à droite et à gauche :

- sur chaque ruban, une case est repérée par un entier relatif $i \in \mathbb{Z}$. La *position d'une tête* à un instant donné est le numéro de la case sur laquelle elle est située. Toutes les têtes sont en position 1 au début du calcul.

À chaque transition, la machine évolue d'une configuration à une autre. Pour décrire la configuration de la machine, il faut décrire le contenu des rubans, la position des têtes et leur état :

- une *configuration* C de M est un élément de $Q \times (\Gamma^k)^{\mathbb{Z}} \times \mathbb{Z}^k$: C décrit l'état des têtes, le contenu de chaque case des k rubans et les positions des têtes.

Le calcul prend fin lorsqu'un état terminal est atteint :

– une configuration *finale* est une configuration pour laquelle l'état est un état terminal q_a ou q_r.

Le calcul d'une machine de Turing est une succession de configurations dictée par la fonction de transition : on passe d'une configuration à une autre en changeant l'état des têtes et le contenu des cases visitées, et en déplaçant les têtes. Plus formellement :

– Soit $C = (q, (c_j^{(1)}, \ldots, c_j^{(k)})_{j\in\mathbb{Z}}, (i_1, \ldots, i_k))$ une configuration de M. Si

$$\delta(q, (c_{i_1}^{(1)}, \ldots, c_{i_{k-1}}^{(k-1)})) = (r, (a_2, \ldots, a_k), (d_1, \ldots, d_k))$$

avec $r \in Q$, $(a_2, \ldots, a_k) \in \Gamma^{k-1}$ et $(d_1, \ldots, d_k) \in \{G, S, D\}^k$, alors la configuration qui suit C dans le calcul de M est $C' = (r, (c_j'^{(1)}, \ldots, c_j'^{(k)})_{j\in\mathbb{Z}}, (i_1', \ldots, i_k'))$ où :

  ◦ pour $1 \leqslant m \leqslant k$, $i_m' = i_m - 1$ si $d_m = G$, $i_m' = i_m$ si $d_m = S$ et $i_m' = i_m + 1$ si $d_m = D$;
  ◦ $c_n'^{(1)} = c_n^{(1)}$ pour tout $n \in \mathbb{Z}$
  ◦ pour $2 \leqslant m \leqslant k$, $c_{i_m}'^{(m)} = a_m$ et $c_n'^{(m)} = c_n^{(m)}$ pour $n \neq i_m$.

  On notera $C \to_M C'$.
– Une *étape de calcul* est le passage d'une configuration à la suivante.
– Le *calcul* de M sur l'entrée $x = x_1 \ldots x_n \in \Sigma^\star$ est la suite potentiellement infinie de configurations $C_0, C_1, \ldots$, où :
  ◦ $C_0 = (q_0, (c_j^{(1)}, \ldots, c_j^{(k)})_{j\in\mathbb{Z}}, (1, \ldots, 1))$ avec $c_j^{(1)} = x_j$ si $1 \leqslant j \leqslant n$, $c_j^{(1)} = B$ sinon, et $c_j^{(m)} = B$ pour tous $2 \leqslant m \leqslant k$ et $j \in \mathbb{Z}$ (en d'autres termes : état initial, rubans vides sauf le ruban d'entrée contenant l'entrée x, têtes en position 1) ;
  ◦ pour tout $j \geqslant 0$, $C_j \to_M C_{j+1}$;
  ◦ si la suite $C_0, \ldots, C_t$ est finie, alors C_t est l'unique configuration finale. Le nombre d'étapes de calcul est alors t.

Il se peut que la machine n'atteigne jamais un état final ; dans ce cas, la machine ne s'arrête pas :

– la machine M *s'arrête* sur l'entrée x (on dit aussi que $M(x)$ s'arrête) si le calcul de M sur x atteint une configuration finale.

Si la machine s'arrête, le mot d'entrée est accepté ou rejeté selon si la machine arrive à l'état d'acceptation ou de rejet :

– on dit que M *accepte* x (ou que $M(x)$ accepte) si M s'arrête sur x et que l'état de la configuration finale est q_a. On dit que M *rejette* x (ou que $M(x)$ rejette) si M s'arrête sur x et que l'état de la configuration finale est q_r. Sinon $M(x)$ ne s'arrête pas.

Enfin, outre l'acceptation ou le rejet, le mot écrit à la fin du calcul sur le ruban de sortie constitue le résultat du calcul :

– si M s'arrête sur x, le *résultat* du calcul de M sur x, noté $M(x)$, correspond au contenu du ruban de sortie dans la configuration finale (sans compter les cases vides à gauche et à droite). Plus formellement, c'est le plus petit mot $y = y_1 \dots y_m \in \Gamma^\star$ tel que pour une certaine position $j \in \mathbb{Z}$, $c^{(k)}_{j+p} = y_p$ pour tout $1 \leqslant p \leqslant m$ et $c^{(k)}_{p'} = B$ pour tout $p' > j+m$ ou $p' \leqslant j$.

Ainsi, deux fonctionnements sont possibles pour une machine de Turing : un mode « acceptation » (M accepte ou rejette le mot x selon l'état q_a ou q_r à la fin du calcul) et un mode « calcul » (le résultat du calcul de M sur x est le contenu y du ruban de sortie à la fin du calcul). Dans le mode acceptation, M accepte un ensemble de mots, tandis que dans le mode calcul, M calcule une fonction.

1-E Définition

Soit M une machine de Turing.

– Le *langage accepté* (ou *reconnu*) par M est l'ensemble des mots $x \in \Sigma^\star$ tels que $M(x)$ accepte.

– La fonction calculée par M est la fonction partielle $f_M : \Sigma^\star \to \Gamma^\star$ définie par $f_M(x) = M(x)$ si $M(x)$ s'arrête.

Enfin, une machine peut effectuer un plus ou moins grand nombre d'étapes et visiter un plus ou moins grand nombre de cases pour mener à bien son calcul, d'où la définition suivante.

1-F Définition (temps et espace)

Soit M une machine de Turing et x un mot sur l'alphabet d'entrée. Si $M(x)$ s'arrête, alors :

– le *temps de calcul* de $M(x)$ est le nombre d'étapes effectuées par le calcul $M(x)$;

– *l'espace* utilisé par $M(x)$ est le nombre total de cases différentes visitées au cours du calcul par les têtes *sur les seuls rubans de travail.*

La raison pour laquelle les rubans d'entrée et de sortie ne sont pas comptés dans l'espace deviendra claire dans le chapitre 4 consacré à ce sujet.

1.2.2 Exemples

Afin d'illustrer ces concepts, voyons maintenant quelques exemples de machines de Turing très simples.

1-G Exemple Une machine à trois rubans (un ruban d'entrée, un ruban de travail et un ruban de sortie) qui teste si un entier écrit en binaire sur le ruban d'entrée est pair

peut fonctionner ainsi : déplacer la tête de lecture vers la droite sur le ruban d'entrée jusqu'à la fin du mot et tester si le dernier chiffre est 0. Pour repérer la fin du mot, on va une case trop loin pour trouver le premier symbole blanc (voir la figure 1.3 pour une illustration). La machine sera donc décrite par :

- l'alphabet d'entrée $\Sigma = \{0, 1\}$ (pour écrire l'entrée en binaire) ;
- l'alphabet de travail $\Gamma = \{0, 1, B\}$;
- l'ensemble des états $Q = \{q_0, q_1, q_a, q_r\}$;
- la fonction de transition définie par
  - $\delta(q_0, (u, B)) = (q_0, (B, B), (D, S, S))$ pour tout $u \in \{0, 1\}$ (sur le ruban d'entrée, aller à droite sans changer d'état tant qu'un blanc n'est pas lu, ne pas bouger sur les autres rubans),
  - $\delta(q_0, (B, B)) = (q_1, (B, B), (G, S, S))$ (revenir d'un cran en arrière dès que l'on a trouvé un blanc sur le ruban d'entrée et passer dans l'état q_1),
  - $\delta(q_1, (0, B)) = (q_a, (B, B), (S, S, S))$ (accepter si le dernier chiffre est 0),
  - $\delta(q_1, (1, B)) = (q_r, (B, B), (S, S, S))$ (rejeter si le dernier chiffre est 1),
  - toutes les autres transitions ne devraient pas se produire en fonctionnement normal, donc on peut définir $\delta(q, (u, v)) = (q_r, (B, B), (S, S, S))$ par exemple pour tous les $(q, (u, v))$ non précisés ci-dessus.

On remarquera que les rubans de travail et de sortie ne servent pas ici.

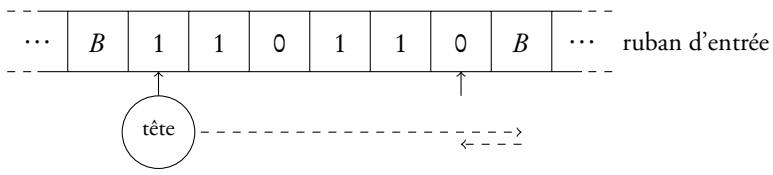


FIGURE 1.3 – Machine pour tester la parité.

1-H **Exemple** Une machine à trois rubans qui multiplie par deux l'entier écrit en binaire sur le ruban d'entrée peut fonctionner ainsi : recopier l'entrée sur le ruban de sortie et ajouter un zéro dès qu'on trouve un blanc. La machine sera donc décrite par :

- l'alphabet d'entrée $\Sigma = \{0, 1\}$ (pour écrire l'entrée en binaire) ;
- l'alphabet de travail $\Gamma = \{0, 1, B\}$;
- l'ensemble des états $Q = \{q_0, q_a, q_r\}$;
- la fonction de transition définie par
  - $\delta(q_0, (u, B)) = (q_0, (B, u), (D, S, D))$ pour tout $u \in \{0, 1\}$ (recopier l'entrée sur le ruban de sortie et aller à droite),

- $\delta(q_0,(B,B)) = (q_a,(B,0),(S,S,S))$ (écrire un zéro à la fin du mot),
- toutes les autres transitions ne devraient pas se produire en fonctionnement normal, donc on peut définir $\delta(q,(u,v)) = (q_r,(B,B),(S,S,S))$ par exemple pour tous les $(q,(u,v))$ non précisés ci-dessus.

On remarquera que le ruban de travail ne sert pas ici.

1-I **Exemple** Une machine à trois rubans qui teste si le mot en entrée a autant de a que de b peut fonctionner ainsi : sur le ruban de travail, écrire au départ le symbole a pour marquer la case initiale ; lire le mot d'entrée de gauche à droite et, sur le ruban de travail, se déplacer à droite lorsqu'on lit un a, à gauche lorsqu'on lit un b ; vérifier qu'à la fin la tête du ruban de travail est revenue à la case initiale. La machine sera donc décrite par :

- l'alphabet d'entrée $\Sigma = \{a,b\}$;
- l'alphabet de travail $\Gamma = \{a,b,B\}$;
- l'ensemble des états $Q = \{q_0,q_1,q_a,q_r\}$;
- la fonction de transition définie par
  - $\delta(q_0,(B,B)) = (q_a,(B,B),(S,S,S))$ (accepter si le mot est vide),
  - $\delta(q_0,(a,B)) = (q_1,(a,B),(D,D,S))$ et
    $\delta(q_0,(b,B)) = (q_1,(a,B),(D,G,S))$ (écrire le symbole a pour repérer la case initiale et se déplacer à droite ou à gauche selon si on lit a ou b en entrée),
  - $\delta(q_1,(a,B)) = (q_1,(B,B),(D,D,S))$,
    $\delta(q_1,(b,B)) = (q_1,(B,B),(D,G,S))$,
    $\delta(q_1,(a,a)) = (q_1,(a,B),(D,D,S))$ et
    $\delta(q_1,(b,a)) = (q_1,(a,B),(D,G,S))$
    (se déplacer à gauche ou à droite selon si on lit b ou a en entrée, en prenant garde à ne pas effacer le symbole a du ruban de travail),
  - $\delta(q_1,(B,a)) = (q_a,(B,B),(S,S,S))$ et $\delta(q_1,(B,B)) = (q_r,(B,B),(S,S,S))$ (accepter ou rejeter selon si on est revenu à la case initiale du ruban de travail à la fin de la lecture du mot d'entrée),
  - toutes les autres transitions ne devraient pas se produire en fonctionnement normal, donc on peut définir $\delta(q,(u,v)) = (q_r,(B,B),(S,S,S))$ par exemple pour tous les $(q,(u,v))$ non précisés ci-dessus.

Enfin, un dernier exemple plus compliqué : l'addition de deux entiers codés en binaire.

1-J **Exemple** Sur le ruban d'entrée se trouvent deux entiers x et y écrits en binaire séparés par un symbole spécial #, c'est-à-dire que le ruban d'entrée contient le mot $x\#y$. Nous voulons écrire en binaire sur le ruban de sortie l'entier z tel que $z = x + y$. Pour cela, nous allons d'abord recopier x sur le ruban de travail, pour disposer de x et de y sur deux rubans séparés (ce qui nous évite de faire des allers-retours pour lire alternativement les chiffres de x et de y). Puis, en partant des bits de poids faibles,

nous allons effectuer l'algorithme d'addition de l'école primaire, en prenant soin de propager la retenue, pour écrire le résultat sur le ruban de sortie. La machine à trois rubans sera donc décrite par :

- l'alphabet d'entrée $\Sigma = \{0, 1, \#\}$;
- l'alphabet de travail $\Gamma = \{0, 1, \#, B\}$;
- l'ensemble des états $Q = \{q_0, q_{\text{bit faible}}, q_{\text{sans retenue}}, q_{\text{avec retenue}}, q_a, q_r\}$: l'état initial q_0 sert à recopier x, $q_{\text{bit faible}}$ à se placer sur les bits de poids faibles de x et y, $q_{\text{sans retenue}}$ et $q_{\text{avec retenue}}$ servent à faire l'addition à partir des bits de poids faibles ($q_{\text{avec retenue}}$ indiquant qu'il y a une retenue) ;
- la fonction de transition définie par
  - $\delta(q_0, (u, B)) = (q_0, (u, B), (D, D, S))$ pour tout $u \in \{0, 1\}$ (recopier x sur le ruban de travail),
  - $\delta(q_0, (\#, B)) = (q_{\text{bit faible}}, (B, B), (D, S, S))$ (changer d'état pour aller à la fin de y),
    $\delta(q_{\text{bit faible}}, (u, B)) = (q_{\text{bit faible}}, (B, B), (D, S, S))$ pour tout $u \in \{0, 1\}$ (aller vers la fin de y),
    $\delta(q_{\text{bit faible}}, (B, B)) = (q_{\text{sans retenue}}, (B, B), (G, G, S))$ (se placer sur le bit de poids faible de y et de la copie de x),
  - $\delta(q_{\text{sans retenue}}, (u, v)) = (q_{\text{sans retenue}}, (v, u + v), (G, G, G))$ si $u, v \in \{0, 1\}$ ne sont pas tous deux égaux à 1 (pas de retenue) et
    $\delta(q_{\text{sans retenue}}, (1, 1)) = (q_{\text{avec retenue}}, (1, 0), (G, G, G))$ (donne lieu à une retenue),
    $\delta(q_{\text{avec retenue}}, (0, 0)) = (q_{\text{sans retenue}}, (0, 1), (G, G, G))$ (fin de la retenue),
    $\delta(q_{\text{avec retenue}}, (1, 1)) = (q_{\text{avec retenue}}, (1, 1), (G, G, G))$ et
    $\delta(q_{\text{avec retenue}}, (u, v)) = (q_{\text{avec retenue}}, (v, 0), (G, G, G))$ si $u + v = 1$ (propagation de la retenue),
  - si x est plus court, on lit B sur le ruban de travail et il faut continuer à lire y :
    $\delta(q_{\text{sans retenue}}, (u, B)) = (q_{\text{sans retenue}}, (B, u), (G, S, G))$ (pas de retenue),
    $\delta(q_{\text{avec retenue}}, (0, B)) = (q_{\text{sans retenue}}, (B, 1), (G, S, G))$ et
    $\delta(q_{\text{avec retenue}}, (1, B)) = (q_{\text{avec retenue}}, (B, 0), (G, S, G))$,
  - si y est plus court que x, alors on lit # sur le ruban d'entrée et il faut continuer à lire x :
    $\delta(q_{\text{sans retenue}}, (\#, u)) = (q_{\text{sans retenue}}, (u, u), (S, G, G))$ (pas de retenue),
    $\delta(q_{\text{avec retenue}}, (\#, 0)) = (q_{\text{sans retenue}}, (0, 1), (S, G, G))$ et
    $\delta(q_{\text{avec retenue}}, (\#, 1)) = (q_{\text{avec retenue}}, (1, 0), (S, G, G))$,
  - enfin, après avoir épuisé x et y, on lit # sur le ruban d'entrée et B sur le ruban de travail :
    $\delta(q_{\text{sans retenue}}, (\#, B)) = (q_a, (B, B), (S, S, S))$ (pas de retenue) et
    $\delta(q_{\text{avec retenue}}, (\#, B)) = (q_a, (B, 1), (S, S, S))$ (écrire la retenue),

- toutes les autres transitions ne devraient pas se produire en fonctionnement normal, donc on peut définir $\delta(q,(u,v)) = (q_r,(B,B),(S,S,S))$ par exemple pour tous les $(q,(u,v))$ non précisés ci-dessus.

On constate la difficulté de définir précisément des machines de Turing pour des tâches pourtant simplistes. C'est la raison pour laquelle nous utiliserons généralement, à partir du prochain chapitre, un langage de plus haut niveau (cf. section 1.2.4).

✎ **1-K** **Exercice**

Donner la description de machines de Turing permettant de :

- tester si le mot en entrée est un palindrome ;
- effectuer la multiplication de deux entiers donnés en binaire.

1.2.3 Code et machine universelle

Un ordinateur peut être programmé pour effectuer différentes tâches, alors qu'il semble à première vue qu'une machine de Turing est spécifique à un problème particulier. Nous allons voir qu'il n'en est rien et qu'il existe une machine *universelle* capable d'exécuter un programme qu'on lui fournit en entrée (selon le vocabulaire informatique, cette machine est donc un interpréteur). L'existence d'une telle machine capable d'exécuter tout « programme » qu'on lui fournit en entrée n'a rien d'évident a priori, et il s'agit d'une des particularités cruciales des machines de Turing (et de nos ordinateurs bien sûr). Cette notion apparaît déjà dans l'article fondateur de Turing [Tur37].

Une telle machine prend en entrée un « programme » et nous devons donc d'abord voir comment coder ce programme, qui n'est rien d'autre que la description d'une machine de Turing. Nous allons donc définir le code d'une machine de Turing : il s'agit essentiellement de coder les alphabets d'entrée et de sortie, l'ensemble des états et la fonction de transition.

1-L **Définition (code d'une machine de Turing)**

Soit $M = (\Sigma,\Gamma,B,Q,q_0,q_a,q_r,\delta)$ une machine de Turing à k rubans. Nous supposerons sans perte de généralité que $\Sigma = \{1,2,\ldots,|\Sigma|\} \subset \mathbb{N}$, $\Gamma = \{1,2,\ldots,|\Gamma|\} \subset \mathbb{N}$ et $B = |\Gamma|$, et que $Q = \{1,2,\ldots,|Q|\} \subset \mathbb{N}$, $q_0 = 1$, $q_a = |Q| - 1$ et $q_r = |Q|$ (ainsi, la seule donnée des tailles de Σ, Γ et Q nous permettra de connaître Σ, Γ, B, Q, q_0, q_a et q_r).

Pour décrire la machine M, on spécifiera en unaire le nombre k de rubans ainsi que $|\Sigma|$, $|\Gamma|$ et $|Q|$, tandis que la fonction de transition δ sera codée par un mot u_δ donnant toutes les transitions possibles.

Plus précisément, le *code* de M est le mot $\langle M\rangle = 1^k 0 1^{|\Sigma|} 0 1^{|\Gamma|} 0 1^{|Q|} 0 u_\delta$ sur l'alphabet $\{0,1\}$, où u_δ est défini comme suit :

– tout d'abord, pour tout élément $s = (q, \gamma_1, \ldots, \gamma_{k-1}) \in (Q \setminus \{q_a, q_r\}) \times \Gamma^{k-1}$, si $\delta(s) = (r, \gamma'_2, \ldots, \gamma'_k, d_1, \ldots, d_k)$ alors on code la transition par le mot

$$u_s = 1^q 0 1^{\gamma_1} 0 1^{\gamma_2} \ldots 0 1^{\gamma_{k-1}} 0 1^r 0 1^{\gamma'_2} 0 1^{\gamma'_3} \ldots 0 1^{\gamma'_k} 0 1^{d'_1} 0 1^{d'_2} \ldots 0 1^{d'_k}$$

où $d'_i = 0$ si $d_i = G$, 1 si $d_i = S$ et 2 si $d_i = D$;

– on note $s_1, \ldots, s_t$ les éléments de $(Q \setminus \{q_a, q_r\}) \times \Gamma^{k-1}$ (disons dans l'ordre lexicographique, mais peu importe) ;

– on définit alors $u_\delta = u_{s_1} 0 u_{s_2} 0 \ldots 0 u_{s_t}$ qui encode toutes les transitions possibles.

Nous avons donné une définition précise du code $\langle M \rangle$ d'une machine de Turing M mais il suffit de retenir qu'un tel code est simple et découle directement de la définition de M. Il y aurait bien d'autres codes possibles, notamment faisant intervenir des codages binaires plus compacts — mais cela n'est pas nécessaire. Il faut simplement que le code, en tant que mot sur $\{0, 1\}$, soit manipulable aisément sur le ruban d'une machine de Turing.

Afin de préparer la construction d'une machine universelle ci-dessous, nous allons voir que l'on peut réduire la taille de l'alphabet sans trop nuire à l'efficacité de la machine : en effet, on peut construire une machine « équivalente » sur un alphabet restreint. Nous devons d'abord définir la notion de machines équivalentes : nous dirons qu'une machine en simule une autre si elle a le même comportement à un codage près des entrées/sorties. Pour coder les entrées/sorties, nous utiliserons un morphisme, c'est-à-dire une application $\varphi : \Sigma^\star \to \Sigma'^\star$ telle que $\varphi(uv) = \varphi(u)\varphi(v)$ (en d'autres termes, il s'agit de coder chaque lettre de Σ par un mot sur Σ', et l'image d'un mot de Σ est alors la concaténation des images de ses lettres).

1-M **Définition (simulation)**

Soit M une machine de Turing d'alphabets d'entrée Σ et de travail Γ. On dit qu'une machine M' (d'alphabets d'entrée Σ' et de travail Γ') *simule* M s'il existe un morphisme « d'entrée » $\varphi : \Sigma^\star \to \Sigma'^\star$ et un morphisme « de sortie » *injectif* $\psi : \Gamma^\star \to \Gamma'^\star$ tels que, pour toute entrée $x \in \Sigma^\star$, $M(x)$ s'arrête ssi $M'(\varphi(x))$ s'arrête, et dans ce cas :

– $M(x)$ accepte ssi $M'(\varphi(x))$ accepte (même acceptation),

– et de plus $M(x) = \psi^{-1}(M'(\varphi(x)))$ (même résultat de calcul).

Bien que la simulation soit une notion concernant les machines, si x est un mot, on dira par abus de langage que le calcul de M' sur $\varphi(x)$ simule le calcul de M sur x (ou plus simplement que $M'(\varphi(x))$ simule $M(x)$).

1-N **Remarque** Le lemme suivant montre qu'on peut se ramener à des machines sur l'alphabet binaire (le symbole blanc en plus bien sûr). On pourrait se contenter de l'idée

de la démonstration esquissée ci-dessous, mais la démonstration relativement formelle que nous proposons ensuite poursuit deux objectifs : d'abord, montrer que l'on peut manipuler formellement les machines de Turing, puis illustrer par la complexité des détails nécessaires comparativement à la simplicité des idées qu'il est inutile de le faire une fois qu'on a appréhendé les capacités des machines de Turing. Par la suite, nous n'entrerons plus dans tant de détails lorsqu'il s'agira de manipuler les machines de Turing car ils obscurcissent les idées sous-jacentes.

1-O Lemme (réduction de l'alphabet)

Si $M = (\Sigma, \Gamma, B, Q, q_0, q_a, q_r, \delta)$ est une machine de Turing, alors il existe une machine M' d'alphabet d'entrée $\Sigma' = \{0,1\}$ et d'alphabet de travail $\Gamma' = \{0,1,B\}$, qui simule M (avec un morphisme d'entrée $\varphi : \Sigma^\star \to \Sigma'^\star$) et telle que :

- si $M(x)$ s'arrête en temps t alors $M'(\varphi(x))$ s'arrête en temps $\leqslant 6t \log|\Gamma|$;
- si $M(x)$ utilise un espace s, alors $M'(\varphi(x))$ utilise un espace $\leqslant 2s \log|\Gamma|$.

De plus, les deux machines ont le même nombre k de rubans et il existe une machine de Turing fixée (indépendante de M) qui calcule le code de M' à partir du code de M.

Idée de la démonstration Chaque lettre de l'alphabet Γ sera codée en binaire et occupera $\log|\Gamma|$ cases sur les rubans de M' (voir la figure 1.4). La machine M' considérera alors ces blocs de cases comme une seule lettre de Γ et pourra effectuer la transition de M qui convient. Les têtes effectuent à la volée les opérations de codage/décodage qui ne nécessitent qu'une mémoire finie puisque le nombre de lettres est fini.

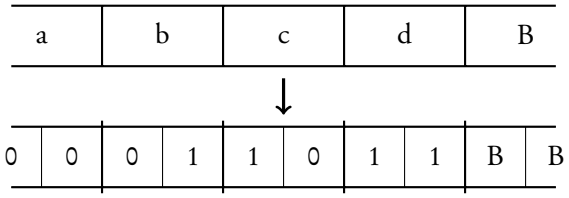


FIGURE 1.4 – Changement de l'alphabet $\{a,b,c,d,B\}$ en $\{0,1,B\}$ sur un ruban : chaque lettre est codée sur deux bits.

Démonstration Soit $\gamma^{(0)}, \ldots, \gamma^{(|\Gamma|-2)}, B$ les éléments de Γ et $\psi : \Gamma^\star \to \Gamma'^\star$ le morphisme qui à $\gamma^{(i)}$ associe le code binaire de i sur $b = \lceil \log(|\Gamma|-1) \rceil$ bits (en ajoutant des zéros en tête si nécessaire) et qui à $B \in \Gamma$ associe $B^b \in \Gamma'^\star$: ce sera le morphisme de sortie, tandis que le morphisme d'entrée $\varphi : \Sigma^\star \to \Sigma'^\star$ est simplement la restriction de ψ à Σ (rappelons en effet que $\Sigma \subset \Gamma$).

La machine M' a le même nombre k de rubans que M et chaque ruban aura le même rôle que dans M (c'est-à-dire qu'on fait une simulation ruban par ruban). Chaque ruban de M' est divisé mentalement en « blocs » de b cases correspondant chacun

au code des lettres se trouvant sur les rubans de M. Pour simuler une transition de M, les têtes de M' se trouveront sur la première case des blocs correspondant aux cases lues par M, elles liront ce bloc de la gauche vers la droite en retenant le mot lu au fur et à mesure, puis reviendront en début de bloc en écrivant le nouveau mot (correspondant à la nouvelle lettre de Γ à écrire) de la droite vers la gauche, et enfin se déplaceront éventuellement à gauche ou à droite de b cases pour se positionner en début du nouveau bloc et simuler la transition suivante.

Lors d'un déplacement à droite de b cases à la fin de la simulation d'une transition, il faut prêter attention au ruban de sortie en écriture seule. Sur ce ruban comme sur les autres, lorsqu'on les survole on aimerait laisser inchangées les cases du bloc que l'on vient de traiter, mais on ne peut pas les lire et on doit quand même écrire quelque chose à la place (par définition, toute transition doit écrire un symbole dans la case où se trouve la tête). Pour remédier à ce problème, dans les états de déplacement on indiquera ce que la tête de sortie doit écrire.

Une étape de M sera alors simulée par au plus $3b-2$ étapes de M'. Il nous faudra de nombreux états supplémentaires pour retenir le mot à lire ou à écrire dans le bloc et pour les déplacements. Ainsi, à chaque état q de M, on associera les états suivants :

- l'état de lecture $q_{\text{lec}}^{u_1,\dots,u_{k-1}}$ (pour tous mots $u_1,\dots,u_{k-1} \in \Gamma'^{\star}$ de taille $\leqslant b-1$) signifie que les préfixes des blocs lus jusqu'à présent de gauche à droite sur les $k-1$ premiers rubans (le ruban de sortie est en écriture seule) sont $u_1,\dots,u_{k-1}$;
- l'état d'écriture $q_{\text{ecr}}^{u_2,\dots,u_k}$ pour tous mots $u_2,\dots,u_k \in \Gamma'^{\star}$ de taille $\leqslant b-1$ signifie qu'il faut encore écrire, de droite à gauche, les préfixes $u_2,\dots,u_k$ dans les blocs des $k-1$ derniers rubans (le ruban d'entrée est en lecture seule) ;
- l'état de déplacement $q_{\text{dep}}^{i_1,\dots,i_k;x}$ pour $-b \leqslant i_j \leqslant b$ et $x \in \Gamma'^{\star}$ ($|x| \leqslant b-1$) signifie qu'il faut encore se déplacer de $|i_j|$ cases (à gauche si $i_j < 0$ et à droite sinon) et que le mot qu'il reste à écrire sur le ruban de sortie dans l'ordre des déplacements est x ;
- on ajoute un état d'acceptation noté q_{acc} et un état de rejet noté q_{rej} (à ne pas confondre avec les états q_a et q_r de M) ;
- enfin, l'état initial est $r_{\text{lec}}^{\epsilon,\dots,\epsilon}$ (où ϵ désigne le mot vide), pour $r = q_0$ l'état initial de M.

Les transitions de M' se déduisent alors aisément :

- pour un état de lecture, tant qu'on n'est pas à la fin du bloc (c'est-à-dire $|u_i| < b-1$), on lit la case courante qui complète le préfixe et on se déplace à droite :

$$\delta'(q_{\text{lec}}^{u_1,\dots,u_{k-1}},(a_1,\dots,a_{k-1})) = (q_{\text{lec}}^{u_1a_1,\dots,u_{k-1}a_{k-1}},(a_2,\dots,a_{k-1},B),(D,\dots,D));$$

- lorsqu'on arrive en fin de bloc (c'est-à-dire $|u_i| = b-1$), il faut effectuer le changement d'état correspondant à la transition de M et commencer à écrire de droite à gauche le nouveau mot sur le bloc.

  Plus précisément, si $\delta(q,(\gamma_1,\dots,\gamma_{k-1})) = (r,(\gamma'_2,\dots,\gamma'_k),(d_1,\dots,d_k))$ et $\psi(\gamma_i) = u_ia_i$ (pour $|u_i| = b-1$ et $a_i \in \Gamma'$) et $\psi(\gamma'_i) = v_ib_i$ (pour $|v_i| = b-1$ et $b_i \in \Gamma'$)

alors :

$$\delta'(q_{\text{lec}}^{u_1,\dots,u_{k-1}},(a_1,\dots,a_{k-1}))=(r_{\text{ecr}}^{v_2,\dots,v_k},(b_2,\dots,b_k),(G,\dots,G));$$

– pour un état d'écriture, tant qu'on n'est pas au début du bloc, on écrit la dernière lettre b_i dans la case courante :

$$\delta'(r_{\text{ecr}}^{v_2b_2,\dots,v_kb_k},(a_1,\dots,a_{k-1}))=(r_{\text{ecr}}^{v_2,\dots,v_k},(b_2,\dots,b_k),(G,\dots,G));$$

– lorsqu'on arrive en début de bloc, si r est un état terminal de M alors on passe dans l'état q_{acc} ou q_{rej} selon que r est acceptant ou rejetant : ainsi, si $b_2,\dots,b_k$ sont des symboles de Γ' alors

$$\delta'(r_{\text{ecr}}^{b_2,\dots,b_k},(a_1,\dots,a_{k-1}))=(q_{\text{acc}},(b_2,\dots,b_k),(S,\dots,S))$$

si $r=q_a$ (et de manière similaire si $r=q_r$).
Si en revanche r n'est pas terminal, alors on écrit la lettre correspondante et on passe dans un état de déplacement, en effectuant le premier mouvement de ce déplacement.
Ainsi, si comme précédemment $\delta(q,(\gamma_1,\dots,\gamma_{k-1}))=(r,(\gamma'_2,\dots,\gamma'_k),(d_1,\dots,d_k))$, alors on définit :

- $i_j=-b+1$ si $d_j=G$,
- $i_j=0$ si $d_j=S$,
- $i_j=b-1$ si $d_j=D$,

et pour gérer le ruban de sortie, si $\psi(\gamma'_k)=\psi_1\dots\psi_b$:

- si $d_k=G$ alors $x=B^{b-1}$ (on écrira des blancs en allant vers la gauche),
- si $d_k=S$ alors $x=\psi_1^{b-1}$ (il faut toujours écrire ψ_1 car on reste sur place),
- si $d_k=D$ alors $x=\psi_2\dots\psi_b$ (il faut écrire $\psi(\gamma'_k)$ car on va à droite).

On a alors

$$\delta'(r_{\text{ecr}}^{b_2,\dots,b_k},(a_1,\dots,a_{k-1}))=(r_{\text{dep}}^{i_1,\dots,i_k;x},(b_2,\dots,b_k),(d_1,\dots,d_k));$$

– pour un état de déplacement, il s'agit de se déplacer d'un bloc vers la gauche ou la droite (b cases) ou de rester sur place.
Si toutes les têtes sont arrivées à destination (début du bloc pour la transition suivante), c'est-à-dire si $i_j=0$ pour tout j et $x=\epsilon$, alors on passe dans l'état de lecture en effectuant le premier déplacement à droite :

$$\delta'(r_{\text{dep}}^{0,\dots,0;\epsilon},(a_1,\dots,a_{k-1}))=(r_{\text{lec}}^{a_1,\dots,a_{k-1}},(a_2,\dots,a_{k-1},B),(D,\dots,D)).$$

Si les têtes sont encore en route, on continue leur trajet : on définit

- $d_j = G$ et $i'_j = i_j + 1$ si $i_j < 0$,
- $d_j = S$ et $i'_j = 0$ si $i_j = 0$,
- $d_j = D$ et $i'_j = i_j - 1$ si $i_j > 0$,

et alors

$$\delta'(r_{\text{dep}}^{i_1,\dots,i_k;x_1\dots x_m},(a_1,\dots,a_{k-1})) = (r_{\text{dep}}^{i'_1,\dots,i'_k;x_2\dots x_m},(a_2,\dots,a_{k-1},x_1),(d_1,\dots,d_k)).$$

Chaque étape de M est ainsi simulée par $3b-2$ étapes de M' ($b-1$ étapes pour lire le bloc de gauche à droite, $b-1$ pour écrire les nouveaux symboles de droite à gauche et b pour se déplacer d'un bloc vers la gauche ou la droite). Puisque $b = \lceil \log(|\Gamma|-1) \rceil \leqslant 2\log|\Gamma|$, le temps mis par M' est $\leqslant 6t\log|\Gamma|$. De même, puisque chaque case de M est remplacée par b cases de M', l'espace utilisé par M' est $\leqslant 2s\log|\Gamma|$.

Nous venons d'écrire explicitement la fonction de transition de M' en fonction de celle de M. La machine de Turing, indépendante de M, qui suivrait ces instructions calculerait donc le code de M' à partir de celui de M. Il serait bien sûr fastidieux mais tout à fait faisable de donner le code d'une telle machine. □

Cette réduction d'alphabet est cruciale pour les machines universelles que nous construisons maintenant. En guise d'échauffement, nous proposons l'exercice suivant.

✎ **1-P Exercice**

Montrer qu'une machine de Turing à $k > 1$ rubans de travail fonctionnant en temps t peut être simulée par une machine de Turing à un seul ruban de travail fonctionnant en temps $O(t^2)$. D'où vient la perte de temps ?

Indication : cf. figure 1.6.

Dans la plupart des cas, nous n'avons pas besoin d'une machine universelle très performante : c'est pourquoi nous en présentons d'abord une dont le principe est simple et qui est capable de simuler une machine avec une perte quadratique en temps. Nous ne donnerons que l'idée de la preuve car nous verrons ensuite une machine plus efficace : on simulera en $O(t\log t)$ étapes une machine fonctionnant en temps t.

1-Q Proposition (machine universelle simple mais non optimale)

Il existe une machine U à 5 rubans, sur l'alphabet d'entrée $\Sigma_U = \{0,1\}$ et l'alphabet de travail $\Gamma_U = \{0,1,B\}$, telle que pour toute machine M sur les alphabets Σ_M et Γ_M :

- il existe un morphisme $\varphi_M : \Sigma_M^\star \to \Sigma_U^\star$ tel que pour tout mot $x \in \Sigma_M^\star$, le calcul de la machine U sur le couple $(\langle M\rangle, \varphi_M(x))$, noté $U(\langle M\rangle, \varphi_M(x))$, simule $M(x)$;
- il existe une constante α_M telle que pour tout $x \in \Sigma_M^\star$, si $M(x)$ s'arrête en temps t et utilise un espace s, alors $U(\langle M\rangle, \varphi_M(x))$ s'arrête en temps $\leqslant \alpha_M(1+st)$ et utilise un espace $\leqslant \alpha_M(1+s)$.

Idée de la démonstration Le lemme 1-O permet de travailler non sur M directement, mais sur la machine équivalente M' ayant pour alphabet $\Gamma_U = \{0, 1, B\}$ (via un morphisme φ_M) : il suffit de calculer dans un premier temps le code de la nouvelle machine. En outre, par commodité, on peut d'abord construire une machine universelle U' sur un alphabet plus grand (utilisant, par exemple, des délimiteurs ou des marqueurs), puis se ramener à l'alphabet Γ_U par le lemme 1-O.

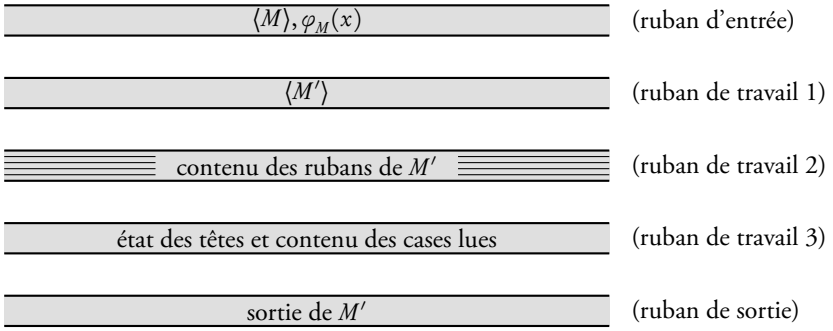


FIGURE 1.5 – Machine universelle à 5 rubans simulant $M(x)$.

Voici le principe de notre machine universelle U' (cf. figure 1.5) :

- le ruban d'entrée sert à lire le mot x encodé sur l'alphabet Γ_U sous la forme $\varphi_M(x)$: il se comportera comme le ruban d'entrée de M ;
- sur le premier ruban de travail, on a calculé le code de la machine M' équivalente à M mais travaillant sur l'alphabet Γ_U ;
- sur le deuxième ruban de travail, on encode les $(k-2)$ rubans de travail de M' en utilisant des blocs consécutifs de $(k-2)$ cases (cf. figure 1.6) et on conserve la position de chaque tête par un marqueur spécial ($\star$ sur la figure, en travaillant donc sur l'alphabet $\Gamma_U \cup (\Gamma_U \times \{\star\})$) : tout se passe comme si l'on disposait de tous les rubans de travail de M' « en parallèle » sur un unique ruban de U ;

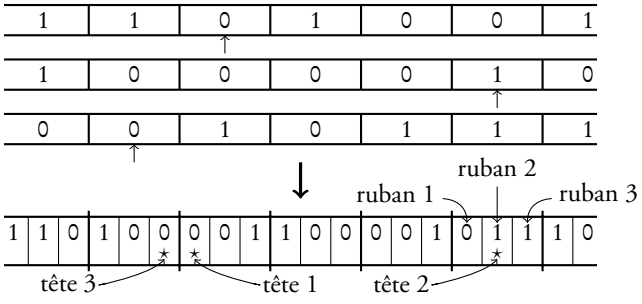


FIGURE 1.6 – Encoder 3 rubans sur un seul.

– à chaque instant, U' conserve sur son troisième ruban de travail l'état des têtes de M' ;

– le ruban de sortie se comporte comme celui de M' ;

– pour simuler une étape de calcul de M', la machine U' parcourt son deuxième ruban de travail à la recherche des têtes de M', recopie sur le troisième ruban de travail le contenu des cases pointées par les têtes de M' : U' dispose maintenant de l'état de M' et du contenu des cases sous les têtes, donc elle va lire dans le code de M' la transition à effectuer et effectue cette transition en changeant l'état inscrit sur le troisième ruban de travail, en modifiant le contenu des cases concernées et en déplaçant le symbole de repérage des têtes de M' si nécessaire.

Pour simuler chaque transition de M', la machine U' fait deux allers-retours sur le ruban codant les rubans de travail de M', ce qui coûte un temps linéaire en le nombre maximal de cases utilisées. Puisque M' utilise au plus $2s\log|\Gamma_M|$ cases, on obtient un temps total $O(st\log|\Gamma_M|)$ pour simuler les t étapes. À ce temps de simulation il convient d'ajouter le temps de calcul du code de M' (une constante dépendant de M), d'où un coût $\alpha_M(st+1)$.

De même pour l'espace, si M a k rubans, l'espace utilisé par notre machine est une constante (dépendant de $|\langle M\rangle|$) sur le premier ruban de travail, $O((k-2)s\log|\Gamma_M|)$ sur le deuxième et une constante (dépendant de $|\langle M\rangle|$) sur le troisième : on obtient au total un espace $\alpha_M(s+1)$. □

1-R **Remarque** Puisqu'une machine fonctionnant en temps t ne peut pas utiliser un espace supérieur à t, la machine universelle de la proposition précédente fonctionne en temps $\leqslant \alpha_M(1+t^2)$, c'est-à-dire $O(t^2)$.

Nous allons maintenant décrire une machine universelle dont le principe est plus compliqué mais qui est capable de simuler une machine avec une perte en temps seulement logarithmique. Cette construction est due à Hennie et Stearns [HS66].

1-S **Théorème (machine universelle avec perte de temps logarithmique)**

Il existe une machine U à 6 rubans, sur l'alphabet d'entrée $\Sigma_U = \{0,1\}$ et l'alphabet de travail $\Gamma_U = \{0,1,B\}$, telle que pour toute machine M sur les alphabets Σ_M et Γ_M :

– il existe un morphisme $\varphi_M : \Sigma_M^\star \to \Sigma_U^\star$ tel que pour tout mot $x \in \Sigma_M^\star$, le calcul $U(\langle M\rangle, \varphi_M(x))$ simule $M(x)$;

– il existe une constante α_M telle que pour tout $x \in \Sigma_M^\star$, si $M(x)$ s'arrête en temps t et utilise un espace s, alors $U(\langle M\rangle, \varphi_M(x))$ s'arrête en temps $\leqslant \alpha_M(1+t\log t)$ et utilise un espace $\leqslant \alpha_M(s+\log t)$.

Idée de la démonstration Nous ne pouvons plus nous permettre comme précédemment de parcourir tous les rubans à chaque étape pour trouver la position des têtes. L'idée est d'effectuer les calculs autour de la position 0 et de ramener à cet emplacement

l'information nécessaire contenue sur les rubans : dès qu'on doit lire une nouvelle case, on va chercher le contenu de la case et tout son voisinage pour le rapatrier autour de la case 0. L'intérêt de ramener également le voisinage est de limiter le nombre de tels rapatriements (car ceux-ci prennent du temps) : si l'on a ramené un voisinage de taille m, alors on peut ensuite effectuer au moins $m/2$ transitions sans rapatrier d'autres blocs [5].

En adaptant la taille du voisinage à chaque transition, on peut simuler une étape de M par $\log t$ étapes de U *en moyenne* : il s'agit d'un coût amorti sur l'ensemble du calcul, certaines transitions requérant plus de temps pour rapatrier de grands blocs, tandis que d'autres ne nécessitent qu'un temps constant.

Démonstration

Description informelle Tous les rubans de travail de M seront regroupés sur un seul ruban de travail de U en les intercalant comme à la figure 1.6, de sorte qu'on peut les traiter les uns après les autres. Le contenu de chaque ruban de travail de M se retrouve sur le ruban de U dans le même ordre, mais un symbole particulier $\square$ peut venir s'intercaler entre les cases (ce qui nous permettra de gérer le rapatriement de blocs). La simulation de toute étape de M s'effectue d'abord en déplaçant un groupe contigu de cases afin d'amener en position 0 la case du ruban de M qui doit être lue : pour cela, on recopie ces cases autour de la position 0 et on efface les cases originales grâce au symbole $\square$ (on dira qu'on « rapatrie un bloc »). On est ensuite à même d'effectuer la transition puisqu'on dispose de la bonne case à lire. Pour éviter de perdre trop de temps dans les rapatriements, plus la case à ramener est loin, plus on rapatrie un bloc de grande taille autour d'elle : cela permet d'avoir autour de la position 0 l'information nécessaire pour effectuer un grand nombre de transitions sans aller chercher une nouvelle case lointaine.

Le ruban est virtuellement découpé en « zones » de plus en plus grandes qui permettent de savoir quels blocs rapatrier. Lorsqu'on rapatrie un bloc, la construction est telle qu'entre ce bloc et la position 0 il n'y a que des cases contenant le symbole $\square$, de sorte qu'on n'écrase aucun contenu lors du rapatriement. Les zones sont définies de sorte que les blocs de taille m sont à distance $O(m)$ et donc leur déplacement prend un temps $O(m)$ en utilisant un ruban de travail auxiliaire : une fois leur rapatriement assuré, on montre qu'il est alors possible de simuler $O(m)$ transitions sans avoir à rapatrier un nouveau bloc si grand, ce qui permet d'obtenir en analyse amortie la complexité de l'énoncé.

Passons maintenant à la démonstration formelle, dans laquelle la gestion des blocs est assez technique.

Alphabet et rubans Le morphisme φ_M sera simplement celui du lemme 1-O codant l'alphabet Γ_M sur $\Gamma_U = \{0, 1, B\}$; la machine travaillant sur Γ_U équivalente à M donnée par ce lemme sera notée M'. Par commodité, nous aurons besoin d'un symbole supplémentaire $\square$ pour représenter une case laissée volontairement vide : nous décrirons donc une machine U' travaillant sur l'alphabet $\{0, 1, \square, B\}$. Il conviendra alors d'appliquer le lemme 1-O pour se ramener à l'alphabet Γ_U.

5. Attention, il s'agit seulement de l'intuition, ce ne sera pas exactement ce qui se passera dans la démonstration car à chaque étape on rapatriera des blocs, petits la plupart du temps.

Dans un premier temps, notre machine U' calcule le code de M' à partir de celui de M et l'écrit sur son premier ruban de travail. Comme à la proposition 1-Q, le ruban d'entrée de U' contenant $(\langle M\rangle, \varphi_M(x))$ servira à lire le mot $\varphi_M(x)$ et se comportera comme le ruban d'entrée de M' ; le premier ruban de travail de U' contiendra le code de M' ; le deuxième ruban de travail contiendra le contenu de tous les rubans de travail de M' ; le troisième ruban de travail contiendra l'état des têtes de M' (ainsi que d'éventuels compteurs pour délimiter les zones mentionnées après) ; le quatrième ruban de travail servira de zone de stockage temporaire pour copier les blocs à rapatrier ; enfin, le ruban de sortie se comportera comme celui de M'. En résumé, le rôle des rubans est comme à la figure 1.5 sauf un ruban de travail supplémentaire pour recopier temporairement les blocs à rapatrier.

Simulation d'une transition Une transition de M' sera simulée comme suit : aller chercher sur le deuxième ruban de travail les cases de M' à lire et les ramener à la position 0 avec un certain nombre de cases les entourant (cette opération détaillée ci-après sera appelée « rapatriement de blocs »), puis lire le code de M' sur le premier ruban de travail pour effectuer la transition correspondante, c'est-à-dire modifier le contenu des cases du deuxième ruban de travail et l'état inscrit sur le troisième ruban de travail. Il nous reste donc à voir comment se comporte le deuxième ruban de travail.

Comme auparavant, les $k-2$ rubans de travail de M' seront encodés sur le deuxième ruban de travail de U par des blocs consécutifs de $k-2$ cases (voir la figure 1.6, mais nous n'aurons pas besoin d'un symbole $\star$ pour repérer la position des têtes). Ainsi on peut accéder aisément au contenu des rubans de travail de M' et les gérer de manière séparée.

Pour chaque ruban de travail de M' simulé par U', on considère les zones suivantes : pour $i \geqslant 0$, D_i (resp. G_i) est l'ensemble des cases de position 2^i à $2^{i+1}-1$ (resp. de $-2^{i+1}+1$ à -2^i). Ces zones recouvrent le ruban sauf à la position 0 qui est traitée spécifiquement (cf. figure 1.7). Notons que les zones D_i et G_i possèdent 2^i cases chacune.

	G_2	G_1	G_0	0	D_0	D_1	D_2	
nombre de cases :	4	2	1	1	1	2	4	

FIGURE 1.7 – Zones sur un ruban de travail.

Ces zones nous permettront de déterminer les blocs que l'on rapatriera si nécessaire lors de la simulation. Afin de gérer le rapatriement de blocs de tailles différentes, il faudra laisser des cases volontairement « vides » et nous utiliserons pour cela le symbole □ : on aura bien le ruban avec ses cases dans l'ordre mais pas toutes contiguës. Cela permet d'éviter de devoir décaler le ruban en entier : on rapatrie le bloc qui nous intéresse en laissant des symboles □ à sa place initiale. Avant la simulation de chaque transition, les invariants suivants seront toujours respectés :

– la case 0 ne contient pas □ ;

- dans la zone D_i (resp. G_i) :
  - soit il n'y a aucun symbole $\square$,
  - soit il y en a exactement la moitié (c'est-à-dire 2^{i-1}),
  - soit toutes les cases contiennent $\square$;
- si D_i et G_i ne contiennent pas seulement le symbole B, alors la somme du nombre de cases $\square$ dans D_i et G_i est 2^i, c'est-à-dire que :
  - D_i (resp. G_i) ne contient pas $\square$ ssi G_i (resp. D_i) ne contient que $\square$,
  - et D_i contient la moitié de $\square$ ssi G_i aussi.

Au départ, toutes les cases contiennent B donc les invariants sont satisfaits. Voici le comportement de notre machine U' pour simuler une étape de la machine M' : supposons qu'il s'agisse d'un déplacement de la tête de M' vers la gauche, c'est-à-dire qu'il faut rapatrier une zone située à gauche (les autres cas sont similaires). La case à lire est la première case sur la gauche qui ne contient pas un symbole $\square$ et on rapatriera autour de la position 0 la moitié de la zone de cette case. Un exemple des différents cas possibles décrits ci-dessous est illustré à la figure 1.8.

1. Se déplacer à gauche jusqu'à trouver la première case ne contenant pas $\square$ (située dans la zone G_i pour un certain $i \geqslant 0$, c'est-à-dire que les zones $G_0, \ldots, G_{i-1}$ contiennent seulement $\square$).
2. Par un aller-retour, tester si les zones G_i et D_i contiennent seulement le symbole B : si c'est le cas et si $i > 0$, on remplit la moitié de G_i et de D_i par le symbole $\square$ et on laisse les autres cases à B (les zones G_0 et D_0 ne contenant qu'une case, si $i = 0$ alors on laisse G_0 inchangée et on écrit $\square$ dans D_0).
3. Puis trois cas sont possibles :
   - si $i = 0$ (c'est-à-dire que la case à la position -1 ne contient pas $\square$), dans ce cas D_0 est vide et on effectue une translation des cases -1 et 0 d'un cran vers la droite : plus précisément, on recopie le contenu des cases -1 et 0 sur les cases 0 et 1 et on écrit $\square$ dans la case -1 ;
   - si $i > 0$ et aucune case de G_i ne contient $\square$, recopier la moitié droite de G_i sur le quatrième ruban de travail et écrire $\square$ à la place (la moitié droite de G_i est maintenant vide) ;
   - si $i > 0$ et la moitié des cases de G_i contiennent $\square$, recopier les cases de G_i ne contenant pas $\square$ sur le quatrième ruban de travail et écrire $\square$ à la place (la zone G_i est maintenant vide).
4. Si $i > 0$, mémoriser le contenu de la case 0 puis recopier les 2^{i-1} cases du quatrième ruban de travail en remplissant exactement la moitié de chaque zone $G_{i-1}, \ldots, G_1$ et en terminant par la case 0 (c'est-à-dire que le dernier symbole de la zone G_i, à droite, se retrouve maintenant en position 0) : tout cela est possible car les zones $G_{i-1}, \ldots, G_1$ sont vides et elles contiennent en tout $\sum_{j=1}^{i-1} 2^j = 2^i - 2$ cases, dont la moitié plus la case 0 font bien 2^{i-1} cases.

5. Enfin, si $i > 0$, effectuer des opérations similaires à droite afin de préserver les invariants : réorganiser le contenu des zones $D_0, \ldots, D_i$ de sorte à remplir la moitié de chaque zone $D_1, \ldots, D_{i-1}$ et, selon si G_i est vide ou non, remplir la totalité de D_i ou seulement la moitié. Tout cela est possible car les zones $D_0, \ldots, D_{i-1}$ contenaient $\sum_{j=0}^{i-1} 2^j = 2^i - 1$ cases pleines, ce qui correspond exactement à la moitié de la capacité des zones $D_1, \ldots, D_i$. Enfin, écrire l'ancien contenu de la case 0 sur la case 1 (zone D_0).

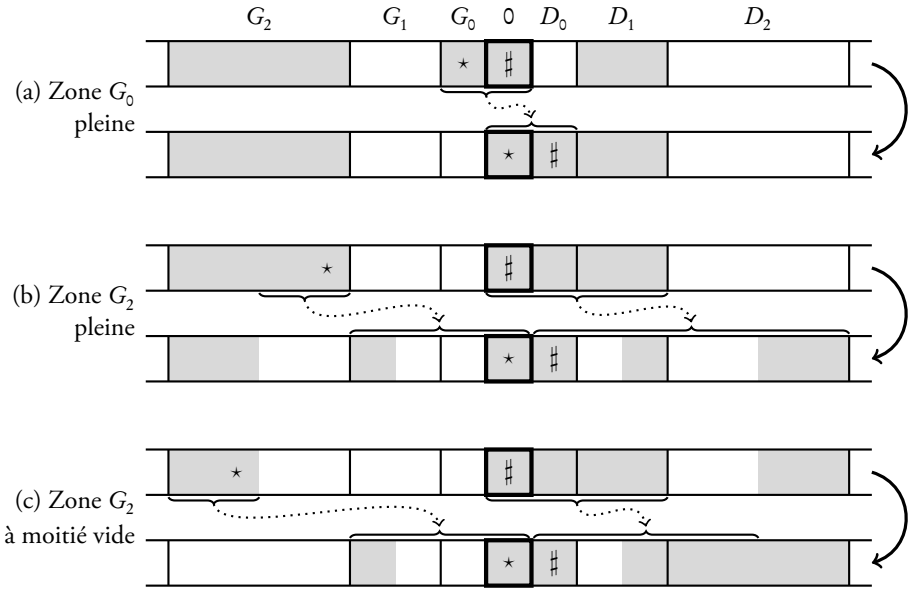


FIGURE 1.8 – Les trois types de rapatriements possibles pour chacun des rubans de travail de M' simulés sur le ruban de travail numéro 2 de U' (les zones grisées représentent les cases non vides). Le symbole ♯ représente la case lue à l'étape précédente, et le symbole ⋆ celle à lire (après l'avoir rapatriée) à l'étape en cours.

Complexité Reste maintenant à analyser le nombre d'étapes et l'espace nécessaires pour simuler le calcul de M'. Remarquons d'abord que le nombre de cases utilisées par U' sur chaque ruban de travail simulé est au plus deux fois celui de M' puisqu'au plus la moitié des cases contiennent $\square$. Par ailleurs, sur le premier ruban de travail, le code de M' prend un espace constant (dépendant de M'), de même pour l'état des têtes sur le troisième ruban de travail. En revanche, pour déterminer les zones G_i et D_i, on a besoin d'un compteur en binaire comptant jusqu'à t (que l'on peut placer sur le troisième ruban de travail), ce qui prend un espace $\log t$. Enfin, il y a la traduction d'alphabet de M à M' et de U' à U, ce qui ajoute une constante multiplicative : en tout, l'espace utilisé par U pour simuler M est donc $\alpha_M(s + \log t)$ pour une constante α_M dépendant de M.

L'analyse du temps d'exécution est une analyse amortie. Il existe une constante α (dépendant du nombre de rubans de M') telle que le rapatriement d'une zone G_i ou D_i prenne $\leqslant \alpha 2^i$ étapes. On supposera α suffisamment grand pour la suite. Que se passe-t-il après avoir rapatrié une zone d'indice i ? Toutes les zones $G_1,\dots,G_{i-1}$ et $D_1,\dots,D_{i-1}$ sont à moitié pleines : $G_1,\dots,G_{i-1}$ (resp. $D_1,\dots,D_{i-1}$) contiennent donc en tout $2^{i-1}-1$ cases non vides. Ainsi, une simulation de 2^{i-1} transitions de M' commençant par le rapatriement d'une zone d'indice i ne rapatrie ensuite que des zones d'indice $\leqslant i-1$.

Montrons par récurrence sur $i \geqslant 0$ la propriété suivante.

Soit t_1 une étape de M' qui nécessite pour sa simulation par U' de rapatrier une zone d'indice i. On note $t_2 > t_1$ la prochaine étape de M' nécessitant de rapatrier une zone d'indice $\geqslant i$. On note $t' = t_2 - t_1$ le nombre d'étapes de t_1 inclus à t_2 exclu : par ce qui précède, $t' \geqslant 2^{i-1}$. Alors la simulation par U' de ces t' transitions prend au plus $2\alpha(i+1)t'$ étapes.

Démonstration — Pour $i = 0$: il n'y a que la transition correspondant au rapatriement de la zone d'indice 0 à simuler, ce qui prend un temps constant. En supposant α suffisamment grand, l'hypothèse est vérifiée au rang 0.

Pour $i > 0$: soit $j_1 \leqslant j_2 \leqslant \cdots \leqslant j_{m-1}$ les indices des zones de plus en plus grandes rapatriées au cours de la simulation des t' étapes, et $j_m \geqslant i$ la zone rapatriée à l'étape t_2. Pour $1 \leqslant n \leqslant m-1$, on appelle $t_{[j_n,j_{n+1}[}$ le nombre de transitions de M' simulées entre le rapatriement de j_n (inclus) et celui de j_{n+1} (exclu). On a : $j_n \leqslant i-1$ pour $n \leqslant m-1$, et $\sum_{n=0}^{m-1} t_{[j_n,j_{n+1}[} = t'$.

Afin de simuler les t' transitions de M', il y a :

- au plus $\alpha 2^i$ étapes de U' pour rapatrier la première zone d'indice i ;
- par récurrence, au plus $2\alpha(j_n+1)t_{[j_n,j_{n+1}[}$ étapes pour simuler les transitions situées entre j_n et j_{n+1}.

Le nombre total d'étapes est donc

$$\leqslant \alpha 2^i + \sum_{n=1}^{m-1} 2\alpha(j_n+1)t_{[j_n,j_{n+1}[} \leqslant \alpha 2^i + 2\alpha i \sum_{n=1}^{m-1} t_{[j_n,j_{n+1}[}$$

$$= \alpha 2^i + 2\alpha i t' \leqslant 2\alpha t' + 2\alpha i t' = 2\alpha(i+1)t',$$

ce qui montre l'hypothèse de récurrence au rang i. ◇

Maintenant, au cours de la simulation de la totalité des t étapes de M', on note de la même manière que précédemment $0 = j_1 \leqslant j_2 \leqslant \cdots \leqslant j_m$ la suite des indices des zones de plus en plus grandes rapatriées. Le dernier intervalle (commençant en j_m) peut être incomplet puisque la fin du calcul peut intervenir avant. Pour éviter de traiter ce cas à part, on supposera sans perte de généralité qu'à la fin du calcul, on simule au plus t déplacements fictifs de M' vers la gauche, ce qui nous assure de rapatrier une zone d'indice $\geqslant j_m$. Ainsi, on simulera $t' \leqslant 2t$ étapes de M' et le dernier intervalle, noté abusivement $t_{[j_m,j_{m+1}[}$, sera complet. Enfin, puisque $2^{j_m-1} \leqslant t_{[j_m,j_{m+1}[} \leqslant t' \leqslant 2t$, on a $j_m \leqslant 2 + \log t$.

Par la propriété montrée ci-dessus, pour $1 \leqslant n \leqslant m$ la simulation de l'intervalle $t_{[j_n,j_{n+1}[}$ prend un temps au plus $2\alpha(j_n+1)t_{[j_n,j_{n+1}[}$ donc la simulation complète nécessite au plus $2\alpha\sum_{n=1}^{m}(j_n+1)t_{[j_n,j_{n+1}[}$ étapes. Puisque $\sum_{n=1}^{m} t_{[j_n,j_{n+1}[} = t' \leqslant 2t$ et $(j_n+1) \leqslant 3+\log t$, le nombre total d'étapes que requiert la simulation est $\leqslant 2\alpha(3+\log t)2t$. En prenant en compte la constante multiplicative venant du changement d'alphabet, on obtient le nombre d'étapes $\alpha_M(1+t\log t)$ de l'énoncé. □

1-T Remarques

– Comme nous le verrons plus tard (proposition 4-W), si une machine s'arrête au bout de t étapes alors elle utilise un espace $s = \Omega(\log t)$. Ainsi, dans l'énoncé du théorème 1-S, l'espace utilisé par la simulation peut en fait s'écrire $\beta_M(1+s)$ pour une constante β_M dépendant de M (c'est-à-dire qu'on peut supprimer le terme $\log t$).

– C'est une question ouverte de construire une machine universelle avec perte de temps moindre que logarithmique (c'est-à-dire fonctionnant en $o(t\log t)$).

– Le théorème précédent montre en particulier que toute machine de Turing fonctionnant en temps $t(n)$ est équivalente à une machine ayant quatre rubans de travail et fonctionnant en temps $O(t(n)\log t(n))$. On peut aisément adapter la preuve pour se ramener à seulement deux rubans de travail, en regroupant les premier, troisième et quatrième rubans de travail en un seul.

1.2.4 Langage de haut niveau

Un peu de recul

Les exemples et résultats qui précèdent illustrent la grande difficulté de définir des machines de Turing pour des opérations pourtant très simples. Pour la suite, la machine de Turing nous servira pour définir proprement les concepts mais nous utiliserons plutôt un langage bien plus compréhensible pour décrire nos algorithmes. Ce chapitre doit nous convaincre que ce langage peut être exécuté par des machines de Turing. Il s'agit d'une des particularités de la complexité (et de la calculabilité) : définir un modèle de calcul basique pour l'oublier ensuite au profit d'un langage plus commode.

Pour décrire un algorithme nous utiliserons dorénavant un langage de haut niveau comme on en a l'habitude, avec des instructions conditionnelles, des boucles, des variables, etc. Il faut donc montrer que ces instructions sont réalisables par une machine de Turing. La proposition suivante est informelle et la preuve est à peine esquissée : nous invitons le lecteur à se convaincre par lui-même qu'une machine de Turing est capable de simuler tout programme qu'on pourrait écrire sur un ordinateur.

1-U **Proposition (informelle)**

Tout programme écrit dans un langage composé de variables entières x_i (en nombre non borné), d'opérations d'affectation $\leftarrow$, d'addition $+$ et de multiplication $\times$, de tests d'égalité $=$ et d'inégalité $\leqslant$, d'instructions conditionnelles `si...alors...sinon` et de boucles `tant que`, peut être simulé par une machine de Turing.

Idée de la démonstration On associe à chaque ligne du programme un certain nombre d'états qui permettront de réaliser l'instruction correspondante. Un ruban de travail sera consacré au stockage des variables : ce ruban contiendra $x_1\#x_2\#\ldots\#x_m$, c'est-à-dire les valeurs des variables codées en binaire et séparées par des symboles #.

L'affectation d'une valeur dans une variable consiste en la recopie au bon endroit de la valeur sur le ruban stockant les variables. Pour cela, il peut être nécessaire de décaler une partie du ruban vers la gauche ou vers la droite pour libérer des cases, ce qui se fait en décalant les cases une par une.

Pour réaliser une addition, il s'agit d'abord d'amener la tête sur la première variable à ajouter, de copier la valeur sur un ruban de travail auxiliaire, de se positionner sur la seconde variable et d'effectuer l'addition comme expliqué à l'exemple 1-J. On réalise une multiplication de manière similaire.

Décider l'égalité de deux valeurs revient à les comparer chiffre à chiffre en effectuant des allers-retours entre elles. Pour un test d'inégalité, il suffit d'abord de comparer le nombre de chiffres des valeurs puis, en cas d'égalité, de comparer les valeurs chiffre à chiffre en partant des bits de poids forts.

Une instruction conditionnelle revient simplement à évaluer un test puis à changer d'état en fonction du résultat pour exécuter la bonne instruction. De même, une boucle `while` n'est rien de plus qu'un test suivi d'un changement d'état, soit pour revenir au début de la boucle, soit pour en sortir. □

1-V **Remarque** Lors de la traduction de ce langage de haut niveau vers une machine de Turing, il faut garder à l'esprit que les opérations d'affectation, d'addition et de comparaison prendront sur la machine de Turing un temps linéaire en la taille binaire des entiers manipulés, et que la multiplication prendra un temps quadratique : contrairement à ce que laisse penser le langage de haut niveau, ce ne sont pas des opérations unitaires.

Nous utiliserons les mots-clés `accepter` et `rejeter` pour signifier que la machine entre dans un état acceptant ou rejetant respectivement. Une telle instruction termine donc le calcul (même s'il reste des instructions à exécuter dans le programme). Par exemple, pour décider si un nombre n est premier, on pourra écrire

- `pour` i de 2 à $n-1$ `faire`
  - si i divise n, `rejeter` ;

```
— accepter.
```

Un peu de recul

Le modèle des machines de Turing est si général qu'il semble être universel. Il s'agit de la thèse de Church-Turing : tout ce qui est « calculable par un système physique » est calculable par machine de Turing.

Attention, ce n'est pas un théorème mais seulement une profession de foi, partagée par la plupart des scientifiques et jamais remise en cause à ce jour depuis sa formulation par Church dans les années 1930. Remarquons qu'on ne peut démontrer un tel énoncé sans définir formellement ce qu'on entend par « calculable par un système physique »…

1.2.5 Indécidabilité

Avant d'aller plus loin, et bien que ceci sorte du cadre de ce livre, il n'est pas inutile de rappeler que certains problèmes ne peuvent pas être résolus par un algorithme, quel que soit le temps dont il dispose. Ce résultat fondamental *d'indécidabilité* remonte aux débuts des machines de Turing (années 1930) et la théorie qui en découle, la calculabilité, a donné ensuite naissance à la complexité algorithmique. Nous suggérons au lecteur de se reporter aux ouvrages cités en introduction, par exemple au livre de Wolper [Wol06], pour approfondir le passionnant sujet de la calculabilité.

La démonstration du résultat suivant utilise le « procédé diagonal », ou *diagonalisation*, inventé par Cantor et que nous étudierons plus en détail par la suite.

1-W Théorème

Il n'existe pas de machine de Turing reconnaissant le langage

$$H = \{\langle M\rangle \mid M \text{ est une machine de Turing et } M(\langle M\rangle) \text{ rejette}\}.$$

Démonstration Par l'absurde, supposons qu'une machine M reconnaisse H. Alors, par définition de M, $\langle M\rangle \in H$ ssi $M(\langle M\rangle)$ accepte. Mais d'autre part, par définition de H, $\langle M\rangle \in H$ ssi $M(\langle M\rangle)$ rejette. Ainsi, $M(\langle M\rangle)$ accepte ssi $M(\langle M\rangle)$ rejette : une contradiction. □

Un peu de recul

Il existe quantité d'autres problèmes indécidables très naturels. Le théorème que nous venons de voir montre qu'il n'existe pas de programme d'ordinateur capable de décider

si un code source quelconque (par exemple un code de programme Java), lors de son exécution sur son propre code, va renvoyer `0` ou `1` ou s'il va boucler indéfiniment.

Plus généralement, on peut montrer (théorème de Rice) qu'aucune propriété non triviale d'un programme ne peut être décidée par algorithme. Pour résumer grossièrement, cela signifie qu'il n'existe pas de méthode automatique pour détecter un bug dans un programme ou pour certifier qu'il exécute bien la tâche voulue.

En calculabilité, les ressources de la machine ne sont pas limitées : elle peut prendre autant de temps et d'espace qu'elle veut. En revanche, les questions que nous nous poserons dans le reste de cet ouvrage concerneront la quantité de ressources nécessaires pour résoudre un problème : ce seront des questions de la forme « quels langages peuvent être reconnus efficacement ? ».

2 Considérations de base sur le temps

Une fois réalisée la formalisation des algorithmes vue au chapitre précédent, les mathématiciens se sont intéressés à des questions liées au type de problèmes que l'on pouvait ainsi résoudre (calculabilité). Plus tard, lors de l'apparition des premiers ordinateurs, la question de l'efficacité de ces algorithmes s'est posée et le domaine de la complexité algorithmique a vu le jour.

Dans ce chapitre, nous allons voir les prémices de la complexité, qui remontent aux années 1960 : définition des classes de complexité en temps, théorèmes de hiérarchie, relations entre classes, etc. Nous introduirons également les machines non déterministes et le problème « $\mathsf{P} = \mathsf{NP}$? » qui a une grande importance depuis les années 1970.

2.1 Temps déterministe

Le modèle des machines de Turing que nous avons vu au chapitre précédent servira à définir les classes de complexité. Rappelons que le *temps* mis par une machine M sur une entrée x est le nombre d'étapes du calcul de $M(x)$ pour arriver à un état final (définition 1-F). Cela revient en quelque sorte à mesurer le temps d'exécution de l'algorithme pour résoudre le problème sur l'entrée x, mais on veut une mesure indépendante de la puissance de l'ordinateur faisant tourner l'algorithme : c'est pourquoi on ne parle pas en secondes mais en nombre d'étapes de calcul. Grâce à cette notion, nous pouvons définir des classes de complexité en temps.

2.1.1 Classes de complexité en temps

Un peu de recul

Pour la plupart des problèmes, plus la taille de l'entrée est grande, plus il faudra du temps pour trouver la solution. Par exemple, il est évident qu'on mettra plus de temps à trier une liste d'un million d'éléments plutôt qu'une liste de 3 éléments.

Il est alors naturel d'évaluer le temps de calcul d'un algorithme en fonction de la taille de son entrée. Cette observation donne lieu à la définition suivante.

La définition suivante apparaît pour la première fois dans l'article [HS65] d'Hartmanis et Stearns.

2-A Définition (classes de complexité en temps déterministe)

- Pour une fonction $t : \mathbb{N} \to \mathbb{N}$, la classe $\mathsf{DTIME}(t(n))$ est l'ensemble des langages reconnus par une machine de Turing M telle qu'il existe une constante α pour laquelle, sur toute entrée x, $M(x)$ fonctionne en temps $\leqslant \alpha t(|x|)$.
- Si $\mathscr{T}$ est un ensemble de fonctions, alors $\mathsf{DTIME}(\mathscr{T})$ désigne $\cup_{t\in\mathscr{T}}\mathsf{DTIME}(t(n))$.

2-B Remarque Dans le domaine de la complexité (contrairement à la calculabilité), toutes les machines que nous considérerons s'arrêteront sur toute entrée. C'est par exemple implicite dans la définition précédente puisque $M(x)$ est censée fonctionner en temps $\leqslant \alpha t(|x|)$. La question n'est plus de savoir si une machine s'arrête, mais en combien de temps elle le fait.

2-C Remarque Pour résoudre la plupart des problèmes non triviaux, il faut lire la totalité de l'entrée x. Ainsi le temps de calcul mis par la machine est au moins $n = |x|$, et les classes $\mathsf{DTIME}(t(n))$ ont peu d'intérêt pour $t(n) = o(n)$.

Avant de poursuivre, mentionnons quelques propriétés évidentes des classes $\mathsf{DTIME}(t(n))$.

2-D Proposition

1. Si pour tout n, $f(n) \leqslant g(n)$ alors $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n))$;
2. pour tout $t(n) \geqslant n$, $\mathsf{DTIME}(t(n))$ est clos par union finie, intersection finie et complémentaire.

Démonstration 1. Si $L \in \mathsf{DTIME}(f(n))$ alors il existe une machine reconnaissant L en temps au plus $\alpha f(n)$ donc a fortiori en temps majoré par $\alpha g(n)$, ce qui implique $L \in \mathsf{DTIME}(g(n))$.

2. Si $L_1, L_2 \in \mathsf{DTIME}(t(n))$, soit M_1 et M_2 deux machines reconnaissant L_1 et L_2 en temps $\leqslant \alpha_1 t(n)$ et $\leqslant \alpha_2 t(n)$ respectivement. Alors une machine pour $L_1 \cup L_2$ (resp. $L_1 \cap L_2$) est la machine $M_\cup(x)$ (resp. $M_\cap(x)$) qui exécute $M_1(x)$ puis $M_2(x)$ et accepte ssi l'une des deux accepte (resp. les deux acceptent). Puisque cette machine doit revenir au début du ruban de lecture entre les calculs de M_1 et de M_2, elle fonctionne en temps $\leqslant \alpha_1 t(n) + \alpha_2 t(n) + O(n)$. Puisque $t(n) \geqslant n$, on a donc $L_1 \cup L_2 \in \mathsf{DTIME}(t(n))$ (resp. $L_1 \cap L_2 \in \mathsf{DTIME}(t(n))$).

   Une machine pour ${}^cL_1$ est la machine $M_c(x)$ qui exécute $M_1(x)$ et accepte ssi $M_1(x)$ rejette : elle fonctionne en temps $\leqslant \alpha_1 t(n)$ donc ${}^cL_1 \in \mathsf{DTIME}(t(n))$. □

Pourquoi définir la classe DTIME à une constante α près ? La raison vient du résultat suivant qui montre que le fonctionnement d'une machine peut être accéléré par n'importe quel facteur constant. En d'autres termes, le temps de calcul d'une machine n'est une mesure pertinente qu'à une constante près.

Un peu de recul

Il n'y a pas de magie dans ce résultat : pour accélérer une machine, il suffit d'agrandir l'alphabet pour faire plus de calculs en une étape. C'est grosso-modo ce qu'on fait en remplaçant les micro-processeurs 32 bits par des micro-processeurs 64 bits.

Comme on l'a vu, on ne peut pas descendre en dessous de n pour le temps d'exécution d'une machine raisonnable, ce qui explique la forme $(1+\epsilon)n + \epsilon t(n)$ utilisée dans le théorème suivant. Si $n = o(t(n))$ alors cela signifie réellement une accélération par n'importe quelle constante. Ce résultat est dû encore à Hartmanis et Stearns [HS65].

2-E Théorème (accélération linéaire)

Pour toute constante $\epsilon > 0$, si un langage L est reconnu par une machine M fonctionnant en temps $\leqslant t(n)$, alors il existe une machine M' reconnaissant L et fonctionnant en temps $\leqslant (1+\epsilon)n + \epsilon t(n)$.

Idée de la démonstration L'astuce consiste à agrandir l'alphabet pour pouvoir réaliser en une seule fois plusieurs étapes de calcul de la machine de départ. Pour réaliser c étapes d'un coup, il faut connaître le voisinage de rayon c autour de la cellule en cours : chaque nouvelle cellule contiendra des $(2c+1)$-uples d'anciennes cellules (ce qui revient à passer à l'alphabet de travail Γ^{2c+1}). En réalité nous n'allons pas simuler c anciennes étapes en une seule nouvelle, mais en 6 nouvelles : les informations nécessaires sont en effet contenues dans les voisines de gauche et de droite qu'il nous faut donc visiter (voir la figure 2.1).

Démonstration La machine M' contient un ruban de travail de plus que M. Soit $c = \lceil 6/\epsilon \rceil$: si l'alphabet de travail de M est Γ, alors celui de M' est $\Gamma \cup \Gamma^c$ (où Γ^c, produit cartésien c fois de Γ avec lui-même, est l'ensemble des c-uples d'éléments de Γ). Sur les rubans de travail de M', chaque cellule contient un bloc de c symboles consécutifs de M. Le ruban d'entrée de M' contient bien sûr l'entrée x sur n cases sur l'alphabet $\Sigma \subset \Gamma$: la première phase de M' consiste à recopier x sur le ruban de travail supplémentaire sur l'alphabet Γ^c : le mot x prend maintenant $\lceil n/c \rceil$ cases. Puisqu'il faut revenir au début du ruban de travail supplémentaire, cette phase requiert $n + \lceil n/c \rceil \leqslant (1+\epsilon)n$ étapes de calcul. Ce ruban de travail supplémentaire simulera le ruban d'entrée de M.

Chaque groupe de c transitions de M est simulé par 5 ou 6 transitions de M' : d'abord, indépendamment du mouvement de M, quatre transitions pour lire les cases voisines (un déplacement à droite suivi de deux à gauche puis un à droite), puis un ou éventuellement deux dernières transitions pour simuler l'écriture et le déplacement de M (voir la figure 2.1).

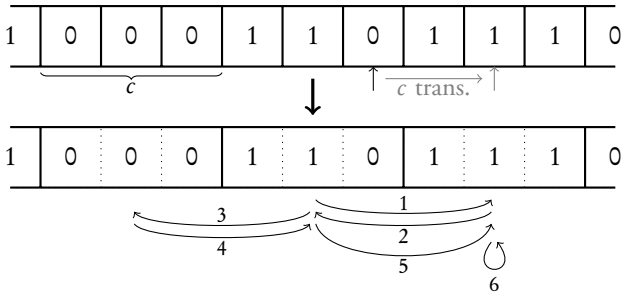


FIGURE 2.1 – Simulation de c transitions en 6 étapes.

Pour simplifier l'explication, nous traiterons chaque ruban de travail séparément : il faudrait en réalité ajouter des états pour retenir le contenu des cases voisines sur tous les rubans.

Pour chaque état q de M, la machine M' contient $c|\Gamma^{2c}| + 2c|\Gamma^c| + 2c$ états pour retenir le contenu des cases voisines et les déplacements à effectuer :

- pour $1 \leqslant i \leqslant c$, un état $q'_{i,D}$ pour débuter la séquence de 5 ou 6 étapes : $q'_{i,D}$ permet de retenir la position i de la tête dans le bloc de c cases et de prévoir un mouvement D ;
- puis M' se déplace à droite en passant dans l'état $q'_{i,G}$ pour prévoir un mouvement G ;
- elle se déplace ensuite à gauche et retient le contenu $u \in \Gamma^c$ de la case qu'elle vient de lire (celle de droite) en passant dans l'état $q'_{i,u,G}$ (le prochain mouvement sera G) ;
- elle se déplace encore à gauche en passant dans l'état $q'_{i,u,D}$ pour prévoir un mouvement D ;

– elle se déplace à droite et retient le contenu $v \in \Gamma^c$ de la case qu'elle vient de lire (celle de gauche) en passant dans l'état $q'_{i,u,v}$.

Après ces 4 déplacements, M' est revenue à la case de départ. Grâce à son état, elle connaît maintenant le contenu de ses deux voisines et, grâce à la lecture de la case courante, celui de la case en cours. En d'autres termes, du point de vue de M, elle connaît le contenu d'au moins c cases vers la gauche et vers la droite : elle peut donc simuler c transitions de M.

Pour cela, il suffit de définir la fonction de transition δ' de M' de sorte que, sur l'état $q'_{i,u,v}$ et en lisant le uple $w \in \Gamma^c$, elle effectue l'équivalent de c transitions de M à partir de la i-ème case du bloc courant. Cela implique de modifier la case en cours en fonction de ce qu'écrit M, puis si nécessaire d'aller modifier la voisine de gauche ou de droite (l'une seulement des deux pouvant être modifiée en c transitions de M) et soit revenir sur la case de départ, soit rester sur la voisine modifiée. Pour effectuer cela, la cinquième transition de M' modifie la case de départ et se déplace du bon côté (G, S ou D) ; si nécessaire, la sixième transition modifie la nouvelle case pointée par la tête et se déplace éventuellement (G ou S si l'on est sur la case de droite, D ou S si on est sur la case de gauche). On passe également dans un nouvel état selon la nouvelle position de la tête de M et son état après c étapes.

Ainsi, $c \geqslant 6/\epsilon$ étapes de M sont simulées par $\leqslant 6$ étapes de M'. En comptant la phase initiale de recopie de l'entrée x sur un ruban de travail supplémentaire, le temps de calcul de $M'(x)$ est majoré par $(1+\epsilon)n + \epsilon\, t(n)$. □

Un peu de recul

Il est crucial de garder à l'esprit que la complexité d'un problème est une **mesure asymptotique**, un temps d'exécution lorsque la taille de l'entrée tend vers l'infini. Ainsi, pour montrer qu'un langage A est dans $\mathsf{DTIME}(n^2)$, il suffit par exemple de donner un algorithme pour A qui fonctionne en temps $3n^2$ *pour n suffisamment grand*. Peu importe son comportement si l'entrée est petite : s'il fonctionne en temps $3n^2$ pour n suffisamment grand, alors il existe une constante c tel qu'il fonctionne en temps $\leqslant cn^2$ pour tout n. Puisque la plupart du temps, la valeur de la constante nous importe peu, nous dirons simplement que l'algorithme fonctionne en temps $O(n^2)$.

En conséquence, dans nombre de raisonnements nous nous intéresserons seulement aux entrées suffisamment grandes.

2.1.2 Théorème de hiérarchie

Nous allons maintenant voir l'un des résultats fondateurs de la complexité : le théorème de hiérarchie en temps déterministe. Il s'agit de montrer que l'on peut résoudre strictement plus de problèmes si l'on dispose de plus de temps, ce qui n'est pas très surprenant. Pour formaliser cela, nous devons introduire le concept de *fonction constructible en temps*.

2-F Définition (fonction constructible en temps)

Une fonction $t : \mathbb{N} \to \mathbb{N}$ est *constructible en temps* s'il existe une constante α et une machine de Turing M qui, sur l'entrée 1^n (l'entier n en unaire) renvoie $1^{t(n)}$ (l'entier $t(n)$ en unaire) en temps $\leqslant \alpha t(n)$.

En quelque sorte, les fonctions constructibles en temps sont des fonctions $t : \mathbb{N} \to \mathbb{N}$ « raisonnables » qui ont de bonnes propriétés. En réalité, la plupart des fonctions que nous utilisons sont constructibles en temps. En voici quelques exemples.

2-G Exemple Les fonctions suivantes $t : \mathbb{N} \to \mathbb{N}$ sont constructibles en temps :

- $t(n) = c$ pour une constante $c \in \mathbb{N}$;
- $t(n) = n$;
- $t(n) = 2^{n^c}$ pour une constante $c \in \mathbb{N}$;
- de plus, si t_1 et t_2 sont constructibles en temps, alors il en est de même de $t_1 + t_2$ et $t_1 t_2$: ainsi, tout polynôme $t \in \mathbb{N}[n]$ est constructible en temps.

✎ **2-H Exercice**

Montrer que les fonctions ci-dessus sont constructibles en temps.

✎ **2-I Exercice**

Montrer qu'une fonction constructible en temps $f(n)$ telle que $f(n) = o(n)$ est ultimement constante.

Indication : une machine fonctionnant en temps $o(n)$ se comporte de la même façon sur les entrées 1^n et 1^{n+1} pour n suffisamment grand.

Le rôle des fonctions constructibles en temps s'illustre par exemple dans le théorème de hiérarchie suivant : une machine disposant de plus de temps peut décider plus de choses. C'est à nouveau un résultat de Hartmanis et Stearns [HS65] mais sous la forme qui suit il est dû à Hennie et Stearns [HS66] grâce à leur machine universelle avec ralentissement logarithmique vue au théorème 1-S.

2-J Théorème (hiérarchie en temps déterministe)

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions telles que $f(n) \neq 0$ (pour tout $n \in \mathbb{N}$), g est constructible en temps et $f \log f = o(g)$. Alors $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.

Idée de la démonstration Il s'agit de construire un langage $L \in \mathsf{DTIME}(g(n))$ tel que $L \notin \mathsf{DTIME}(f(n))$. Pour cela, nous allons faire en sorte que toute machine fonctionnant

en temps $f(n)$ et censée reconnaître L se trompe sur au moins une entrée. Dans la lignée des paradoxes logiques célèbres, on utilisera l'auto-référence par une question de la forme « est-ce qu'une machine M sur son propre code rejette en temps $\leqslant f(n)$? », sur laquelle toute machine fonctionnant en temps $f(n)$ doit se tromper. Modulo quelques détails techniques, le langage L sera l'ensemble des machines M pour lesquelles la réponse à la question précédente est positive.

Pour décider L, il faudra simuler une machine fonctionnant en temps $f(n)$ par une machine universelle, ce qui lui prend $O(f(n)\log f(n))$ étapes (théorème 1-S), d'où la condition du théorème.

Démonstration Nous aimerions définir $L = \{\langle M\rangle \mid M(\langle M\rangle)$ rejette en temps $\leqslant f(n)\}$ où M désigne une machine de Turing, mais nous ne savons pas décider ce langage en temps $g(n)$ (à cause de la constante α_M dépendant de M dans le théorème 1-S). En gardant la même idée, il va falloir être légèrement plus subtil. On va d'abord ajouter un mot x quelconque à l'entrée du problème (donc travailler sur l'entrée $(\langle M\rangle, x)$ à la place de $\langle M\rangle$) pour pouvoir agrandir arbitrairement la taille de l'entrée, puis définir le langage en termes de fonctionnement d'une machine.

Soit V la machine suivante sur l'entrée $(\langle M\rangle, x)$:

- exécuter $U(\langle M\rangle, (\langle M\rangle, x))$ pendant $g(n)$ étapes, où U est la machine universelle du théorème 1-S et $n = |(\langle M\rangle, x)|$ (c'est-à-dire qu'on simule $M(\langle M\rangle, x)$ tant que la simulation prend moins de $g(n)$ étapes) ;
- si U n'a pas terminé son calcul, alors rejeter ;
- sinon, accepter ssi U rejette.

On note L le langage reconnu par V.

On remarquera que V calcule d'abord $g(n)$ pour savoir quand arrêter l'exécution de U : puisque g est constructible en temps, cela prend $O(g(n))$ étapes. Puis l'exécution de U prend à nouveau $g(n)$ étapes, donc en tout V fonctionne en temps $O(g(n))$: ainsi, $L \in \mathsf{DTIME}(g(n))$.

Montrons maintenant que $L \notin \mathsf{DTIME}(f(n))$. Soit M un machine fonctionnant en temps $\alpha f(n)$. Ainsi U simule M en temps $\alpha_M \alpha f(n)\log f(n)$ (où α_M est une constante dépendant seulement de M). Pour n assez grand, par hypothèse

$$g(n) \geqslant \alpha_M \alpha f(n)\log f(n),$$

donc pour x assez grand, la simulation de M par U se termine avant $g(n)$ étapes, donc $V(\langle M\rangle, x)$ accepte ssi $M(\langle M\rangle, x)$ rejette. Ainsi, M « se trompe » sur l'entrée $(\langle M\rangle, x)$ et ne reconnaît donc pas L. □

2-K Remarques

- Nous verrons au théorème 6-O une astuce permettant de réduire un peu l'écart entre f et g dans le théorème précédent (c'est-à-dire obtenir mieux que $f\log f =$

$o(g)$). On pourrait également essayer de construire une machine universelle plus efficace que celle du théorème 1-S, mais il s'agit d'une question ouverte comme on l'a vu (remarque 1-T).

– Si l'on se restreint aux machines de Turing ayant *toutes* exactement k rubans de travail pour $k \geqslant 2$ fixé, alors Fürer [Für82] montre un théorème de hiérarchie optimal, c'est-à-dire sans le facteur $\log(f(n))$.

Le théorème de hiérarchie est très intuitif puisqu'il nous apprend qu'en disposant de plus de temps de calcul, on peut résoudre plus de problèmes. Cependant, l'hypothèse de constructibilité en temps de la fonction g est cruciale : en effet, le résultat suivant montre ce que l'on peut faire sans supposer g constructible. Il est dû à Trakhtenbrot [Tra64] en 1964 (rédigé en russe) et a été redécouvert par Borodin [Bor72] en 1972 pour le monde occidental sous le nom de « Gap theorem ».

2-L Théorème (de la lacune)

Il existe une fonction $f : \mathbb{N} \to \mathbb{N}$ telle que $f(n) \geqslant n$ et $\mathsf{DTIME}(f(n)) = \mathsf{DTIME}(2^{f(n)})$.

Idée de la démonstration L'idée est de définir la fonction f de sorte qu'aucune machine de Turing ne s'arrête en un temps compris entre $f(n)$ et $2^{f(n)}$, ce qui implique bien sûr qu'il n'y a aucun langage dans $\mathsf{DTIME}(2^{f(n)}) \setminus \mathsf{DTIME}(f(n))$.

Démonstration Si M est une machine, on note Σ_M son alphabet d'entrée. La propriété suivante sera vérifiée : pour tout $n \in \mathbb{N}$, aucune machine M telle que $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$ ne s'arrête sur une entrée de taille n en un temps compris entre $f(n)$ et $n2^{f(n)}$.

Posons arbitrairement $f(0) = 0$ et définissons $f(n)$ pour un $n > 0$ donné. Pour cela on considère la suite $(u_i)_{i \geqslant 1}$ définie par $u_1 = n$ et $u_{i+1} = n2^{u_i} + 1$: ainsi, les intervalles $[u_i, n2^{u_i}]$ forment une partition de $[n, +\infty[$. Soit M une machine telle que $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$ et soit $x \in \Sigma_M$ de taille n. Si $M(x)$ ne s'arrête pas, alors M ne décide pas de langage. Sinon, $M(x)$ s'arrête en un temps $t_{M,x}$: alors soit $t_{M,x} < n$, soit $t_{M,x} \in [u_i, 2^{u_i}]$ pour un certain i.

Il y a $\leqslant 2^{n+1} - 1$ machines M telles que $|\langle M \rangle| \leqslant n$ et il y a $\leqslant n^n$ mots de taille n dans Σ_M (car $|\Sigma_M| \leqslant n$), donc il y a $< n^n 2^{n+1}$ valeurs $t_{M,x}$ distinctes. On en déduit que l'un des intervalles $[u_i, n2^{u_i}]$, pour $1 \leqslant i \leqslant n^n 2^{n+1}$, ne contient aucun $t_{M,x}$. Si $[u_k, n2^{u_k}]$ est un tel intervalle, on définit $f(n) = u_k$ (on remarquera que $f(n) \geqslant n$ puisque $u_1 = n$). Ainsi, si $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$, alors pour tout mot $x \in \Sigma_M$ de taille n, $M(x)$ ne s'arrête pas en un temps compris entre $f(n)$ et $n2^{f(n)}$.

Maintenant que f est définie, soit $L \in \mathsf{DTIME}(2^{f(n)})$. Alors une machine M reconnaît L en temps $\leqslant \alpha 2^{f(n)}$ pour une certaine constante α. Pour n suffisamment grand, $|\langle M \rangle| \leqslant n$, $|\Sigma_M| \leqslant n$ et $\alpha 2^{f(n)} \leqslant n2^{f(n)}$. Donc, pour $|x|$ suffisamment grand, $M(x)$ ne s'arrête pas en un temps compris entre $f(|x|)$ et $|x|2^{f(|x|)}$: puisqu'elle s'arrête en un temps $< |x|2^{f(|x|)}$, on en déduit que $M(x)$ doit s'arrêter en un temps $< f(|x|)$. D'où $L \in \mathsf{DTIME}(f(n))$. □

✎ **2-M** **Exercice**

Montrer qu'on peut prendre f strictement croissante dans le théorème ci-dessus.
Indication : modifier la valeur de départ de la suite (u_i) dans la démonstration.

2-N **Remarque** Le nom de ce théorème contre-intuitif vient du « trou » qu'il y a entre $f(n)$ et $2^{f(n)}$, intervalle dans lequel on ne trouve aucun langage. Rappelons que f n'est pas constructible en temps, sinon on contredirait le théorème de hiérarchie.

En réalité, n'importe quelle fonction $s : \mathbb{N} \to \mathbb{N}$ conviendrait à la place de $n \mapsto 2^n$ (c'est-à-dire $\mathsf{DTIME}(f(n)) = \mathsf{DTIME}(s(f(n)))$ pour une fonction f construite comme ci-dessus), c'est-à-dire qu'on peut rendre le « trou » aussi grand que l'on veut. Ce théorème est habituellement donné pour une fonction s calculable quelconque (notion que nous n'avons pas définie) ; dans ce cas, f est elle-même calculable.

2.1.3 Temps polynomial et temps exponentiel

Nous pouvons maintenant définir deux classes de complexité qui joueront un rôle important par la suite. L'idée de considérer comme notion d'algorithmes « efficaces » les algorithmes fonctionnant en temps polynomial, remonte aux articles de Cobham [Cob65] et d'Edmonds [Edm65] en 1965.

2-O **Définition (temps polynomial et exponentiel)**

La classe P est l'ensemble des langages reconnus en temps polynomial, c'est-à-dire

$$\mathsf{P} = \mathsf{DTIME}(n^{O(1)}) = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(n^k).$$

La classe EXP est l'ensemble des langages reconnus en temps exponentiel, c'est-à-dire

$$\mathsf{EXP} = \mathsf{DTIME}(2^{n^{O(1)}}) = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{n^k}).$$

Enfin, une dernière classe que l'on étudiera moins est notée E, il s'agit de l'ensemble des langages reconnus en temps exponentiel avec exposant linéaire, c'est-à-dire :

$$\mathsf{E} = \mathsf{DTIME}(2^{O(n)}) = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{kn}).$$

On a bien sûr $\mathsf{P} \subseteq \mathsf{E} \subseteq \mathsf{EXP}$. Nous pouvons d'ores et déjà appliquer le théorème de hiérarchie 2-J pour séparer ces classes.

2-P **Corollaire**

$$\mathsf{P} \subsetneq \mathsf{EXP}$$

Démonstration Il suffit de remarquer que $\mathsf{P} \subseteq \mathsf{DTIME}(2^n)$ puis, par le théorème 2-J, que $\mathsf{DTIME}(2^n) \subsetneq \mathsf{DTIME}(2^{n^2})$ et enfin que $\mathsf{DTIME}(2^{n^2}) \subseteq \mathsf{EXP}$. □

✎ **2-Q** **Exercice**

Séparer de même E et P.

2-R **Exemple** Voici quelques exemples de problèmes dans P et dans EXP.

1. `MULTIPLICATION D'ENTIERS` :
   - *entrée* : deux entiers a et b donnés en binaire ; un entier k ;
   - *question* : le k-ème bit du produit ab vaut-il 1 ?

   Ce problème est dans P : il suffit d'effectuer la multiplication ab grâce à l'algorithme de l'école primaire et de regarder le k-ème bit du résultat. Cela prend un temps polynomial.
2. `MULTIPLICATION DE MATRICES` :
   - *entrée* : deux matrices A et B de taille $m \times m$ à coefficients entiers donnés en binaire ; un couple (i, j) avec $1 \leqslant i, j \leqslant m$; enfin, un entier k ;
   - *question* : le k-ème bit du coefficient (i, j) du produit AB vaut-il 1 ?

   Ce problème est dans P : le coefficient (i, j) de AB n'est rien d'autre que la somme des m produits $a_{i,k} b_{k,j}$, ce qui se calcule en temps polynomial.
3. `ACCESSIBILITÉ` :
   - *entrée* : un graphe orienté G et deux sommets s et t ;
   - *question* : existe-t-il un chemin dans G de s à t ?

   Ce problème est dans P : il suffit de faire un parcours du graphe en partant de s et de voir si l'on atteint t (parcours en largeur ou en profondeur, peu importe).
4. `COUPLAGE PARFAIT` :
   - *entrée* : un graphe non orienté G ;
   - *question* : G admet-il un couplage parfait ?

   Un couplage parfait est un ensemble d'arêtes tel que chaque sommet est relié à exactement une arête de cet ensemble. Ce problème est dans P mais l'algorithme polynomial n'est pas trivial et est dû à Edmonds [Edm65].
5. `PRIMALITÉ` :

- *entrée* : un entier N donné en binaire ;
- *question* : N est-il premier ?

Ce problème est dans P mais la démonstration n'est pas évidente (l'algorithme naïf consistant à essayer tous les diviseurs potentiels jusqu'à $\sqrt{N}$ est exponentiel puisque la taille de l'entrée est $n = \log N$) : il a fallut attendre Agrawal, Kayal et Saxena [AKS04] pour l'obtenir.

6. `RANGEMENT` :
   - *entrée* : des entiers $p_1, \dots, p_n \in \mathbb{N}$ en binaire (les poids de n objets) et deux entiers m, p en binaire (le nombre de boîtes de rangement et le poids maximal que chacune peut contenir) ;
   - *question* : peut-on ranger les n objets dans les m boîtes en respectant le poids maximal p de chaque boîte ?

   Ce problème est dans EXP car il suffit de tester toutes les possibilités pour ranger les objets (c'est-à-dire pour chaque i, décider dans laquelle des m boîtes mettre l'objet i) et de voir si l'une d'entre elles respecte les contraintes. Il y a $m^n = 2^{n \log m}$ possibilités, donc ce calcul prend un temps exponentiel en la taille de l'entrée. En réalité, bien que ce problème ne soit probablement pas dans P, on peut quand même faire mieux que EXP car il appartient à la classe NP que nous verrons par la suite (il est NP-complet). Il est appelé `BIN PACKING` en anglais.
7. `ARRÊT EXP` :
   - *entrée* : le code d'une machine M et un entier k en binaire ;
   - *question* : est-ce que $M(\epsilon)$ s'arrête en $\leqslant k$ étapes ?

   Ce problème est dans EXP car on peut simuler $M(\epsilon)$ pendant k étapes : puisque la taille de l'entrée k est $\log k$, cela prend un temps exponentiel. Nous n'avons pas encore vu cette notion, mais mentionnons au passage que ce problème est EXP-complet.
8. Un problème de EXP plus « naturel » est, sur l'entrée C, de calculer la valeur du circuit booléen D dont le code est décrit par le circuit booléen C donné en entrée (nous définirons les circuits au chapitre 5). Puisque D peut être de taille exponentielle par rapport à C, ce calcul prend un temps exponentiel. Comme le précédent, ce problème est EXP-complet (exercice B-C).

✎ **2-S** **Exercice**

Justifier plus rigoureusement que ces problèmes sont chacun dans la classe annoncée. Pour le problème du couplage parfait, voir [Edm65] ou un livre d'algorithmique comme [Cor+10]. Pour le problème de la primalité, voir [AKS04].

Un peu de recul

La classe P est généralement présentée comme celle des problèmes résolubles « efficacement ». Bien qu'en pratique on sache résoudre rapidement certains problèmes hors de P, et par ailleurs qu'il soit difficile d'admettre qu'un algorithme fonctionnant en temps $1\,000\,000n^{1\,000}$ soit efficace, il n'en reste pas moins que cette présentation est très souvent pertinente : la plupart des problèmes « naturels » dans P ont un algorithme en n^k pour un petit exposant k et pour la plupart des problèmes « naturels » hors de P, on ne connaît pas d'algorithme efficace.

On verra plus tard que les algorithmes probabilistes polynomiaux ajoutent encore à la confusion. Toujours est-il que la classe P est « robuste » (bonnes propriétés, notamment de clôture) et donne un bon cadre d'étude en complexité.

Notre première propriété de P sera la clôture de P par changements finis. Si A et B sont deux langages, rappelons que leur différence symétrique $A\Delta B$ est l'ensemble des mots contenus dans exactement un seul des deux ensembles, c'est-à-dire $A\Delta B = (A\setminus B)\cup(B\setminus A)$.

2-T Lemme

Soit L et L' deux langages sur un alphabet Σ. Si $L \in \mathsf{P}$ et $L\Delta L'$ est fini (c'est-à-dire que L et L' diffèrent seulement en un nombre *fini* de mots), alors $L' \in \mathsf{P}$.

Idée de la démonstration La machine pour L' va contenir dans son code la liste (finie) des mots de $L\Delta L'$. Pour décider si $x \in L'$, on décide d'abord si $x \in L$ puis on corrige éventuellement la réponse en testant si x appartient à la liste des mots de $L\Delta L'$.

Démonstration Soit $\{x_1,\dots,x_m\}$ l'ensemble des mots de $L\Delta L'$. Soit M une machine polynomiale pour le langage L. Voici une machine M' pour L' sur l'entrée x :

- exécuter $M(x)$;
- si $x \in \{x_1,\dots,x_m\}$, alors `accepter` ssi $M(x)$ rejette ;
- sinon `accepter` ssi $M(x)$ accepte.

L'exécution de $M(x)$ prend un temps polynomial. Pour tester si $x \in \{x_1,\dots,x_m\}$, on compare x à chacun des mots x_i, lesquels sont codés « en dur » dans le code de la machine M'. Chaque comparaison prend un temps linéaire ; puisque le nombre de comparaisons est fixé une fois pour toute, le test $x \in \{x_1,\dots,x_m\}$ prend lui aussi un temps linéaire. Au total, M' fonctionne en temps polynomial et reconnaît L', donc $L' \in \mathsf{P}$. □

2.2 Temps non déterministe

Déjà dans l'article de Turing [Tur37], une variante des machines de Turing est considérée pour modéliser les « choix » ou l'intuition dans la conduite d'une preuve mathématique (en effet, à l'époque Hilbert demandait s'il était possible d'automatiser le processus de preuve). Ce concept n'a pas été développé plus avant par Turing mais il a ensuite permis la définition d'autres classes de complexité en temps ayant des propriétés particulièrement intéressantes. Nous allons donc introduire cette variante de machines appelées « non déterministes » car elles ont plusieurs choix possibles à chaque transition.

2.2.1 Machines non déterministes

Jusqu'à présent, les machines de Turing que nous avons vues n'avaient aucun choix : la transition qu'elles effectuaient était déterminée uniquement par leur état et les cases lues par la tête. C'est pourquoi on les appelle *machines déterministes*. Si en revanche, à chaque étapes plusieurs transitions sont possibles, la machine a plusieurs exécutions possibles et est appelée *non déterministe*.

2-U Définition (machine de Turing non déterministe)

Une machine de Turing $N = (\Sigma, \Gamma, B, Q, q_0, q_a, q_r, \delta)$ est dite *non déterministe* si δ n'est plus une fonction mais une relation :

$$\delta \subseteq ((Q \setminus \{q_a, q_r\}) \times \Gamma^{k-1}) \times (Q \times \Gamma^{k-1} \times \{G, S, D\}^k).$$

À chaque étape, plusieurs transitions sont possibles : tout élément

$$((q, (\gamma_1, \ldots, \gamma_{k-1})), (r, (\gamma'_2, \ldots, \gamma'_k), d)) \in \delta$$

signifie que de l'état q en lisant $(\gamma_1, \ldots, \gamma_{k-1})$ la machine N peut aller dans l'état r, écrire $(\gamma'_2, \ldots, \gamma'_k)$ et se déplacer dans la direction d.

Ainsi, plutôt qu'une suite de configurations, le calcul d'une machine de Turing non déterministe est un *arbre de configurations* (cf. figure 2.2) puisqu'une configuration a plusieurs successeurs possibles. Dans cet arbre, certaines configurations peuvent éventuellement apparaître plusieurs fois si elles sont atteintes par différents chemin.

2-V Remarque En supposant qu'à chaque étape, seulement deux transitions sont possibles (ce qui ne change pas fondamentalement le modèle et permet de définir les mêmes classes), on peut aussi voir une machine non déterministe comme une machine ayant deux fonctions de transition déterministes.

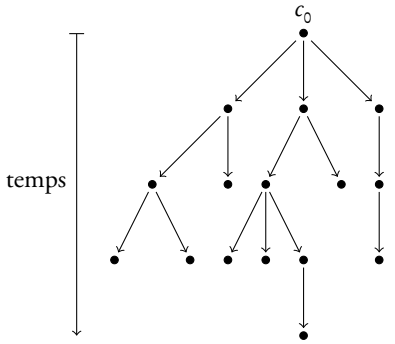


FIGURE 2.2 – Arbre de calcul d'une machine non déterministe.

2-W Définition

- Une *exécution* d'une machine non déterministe N est une suite de configurations compatible avec la relation de transition δ, d'une configuration initiale à une configuration finale.
- Une exécution de N est aussi appelée *chemin* car il s'agit d'un chemin dans l'arbre de calcul, de la racine à une feuille.
- Le temps de calcul d'une machine de Turing non déterministe est le temps maximal d'une de ses exécutions, c'est-à-dire la hauteur de son arbre de calcul.
- De même, l'espace utilisé par une machine de Turing non déterministe est l'espace maximal utilisé par l'une de ses exécutions.

Voici maintenant le mode d'acceptation d'une machine non déterministe.

2-X Définition

Le langage reconnu par une machine non déterministe N sur l'alphabet Σ est l'ensemble des mots $x \in \Sigma^*$ tels qu'il existe un chemin acceptant dans le calcul $N(x)$.

Un peu de recul

Ainsi, pour accepter un mot x, il faut et il suffit qu'il y ait *au moins une* exécution qui mène à un état acceptant dans l'arbre de calcul de $N(x)$. En d'autres termes, un mot x est rejeté ssi *aucun* chemin n'est acceptant dans l'arbre de calcul de $N(x)$. Nous reviendrons sur l'asymétrie entre acceptation et rejet, qui est cruciale pour comprendre le non-déterminisme.

Lorsque x est accepté, on dit de manière imagée que la machine « devine » le chemin menant à l'état acceptant. Cela correspond à l'idée de l'intuition guidant le déroulement d'une preuve mathématique. Mais attention, ce n'est qu'une image, la machine ne devine rien, c'est simplement son mode d'acceptation qui est défini ainsi.

2-Y **Remarque** Contrairement aux machines déterministes pour lesquelles on pouvait aisément définir la notion de fonction calculée par une machine (mot écrit sur le ruban de sortie, voir la définition 1-E), il n'est pas évident d'en faire autant pour les machines non déterministes. En effet, plusieurs chemins peuvent mener à plusieurs résultats différents. Nous nous contenterons donc d'étudier des machines en mode « acceptation ».

2.2.2 Langage de haut niveau

Une machine de Turing déterministe est bien entendu un cas particulier d'une machine non déterministe. Pour capturer la puissance du non-déterminisme, nous allons ajouter à notre langage de haut niveau du paragraphe 1.2.4 une instruction `deviner`. Dans un premier temps, cette instruction permet de deviner un bit $b \in \{0,1\}$. Cela correspond à deux possibilités pour la relation de transition δ : choisir la première transition si $b = 0$ et la seconde si $b = 1$. De cette manière, nous décrivons le calcul de la machine non déterministe le long de chaque chemin, le chemin courant étant décrit par la suite d'embranchements devinés.

2-Z **Exemple** Voici une machine non déterministe pour décider si le nombre x est composé (non premier), c'est-à-dire pour le problème `COMPOSÉ` suivant :

- *entrée* : un entier $x = x_1 \dots x_n$ donné en binaire ;
- *question* : x est-il composé ?

Machine N pour `COMPOSÉ` :

- `Pour` i `de 1 à` n `faire`
  - `deviner` $b_i \in \{0,1\}$,
  - `deviner` $b'_i \in \{0,1\}$;
- on considère les nombres $b = b_1 \dots b_n$ et $b' = b'_1 \dots b'_n$ écrits en binaire ;
- vérifier que $1 < b < x$ et $b \times b' = x$: `accepter` ssi les deux conditions sont réunies.

Le calcul $N(x)$ a un chemin acceptant ssi x est composé.
Remarquons qu'en un algorithme, on a décrit 2^{2n} chemins puisqu'on a deviné $2n$ bits.

Plutôt que deviner des bits $b_i \in \{0,1\}$, nous pourrions plus généralement deviner une lettre de l'alphabet de travail Γ, ce qui correspondrait à $|\Gamma|$ transitions possibles. Par ailleurs, très souvent nous voulons deviner un mot plutôt que les lettres une par une, comme ici où nous étions intéressés par les mots b et b' et non par les lettres b_i et b'_i. Nous abrégerons la boucle précédente par une instruction `deviner` étendue aux mots comme dans l'exemple suivant.

2-AA **Exemple** On réécrira la machine de l'exemple précédent avec une instruction `deviner` étendue aux mots :

- `deviner` $b \in \{0,1\}^n$;
- `deviner` $b' \in \{0,1\}^n$;
- `accepter` ssi $1 < b < x$ et $b \times b' = x$.

2.2.3 Machine non déterministe universelle

Comme pour les machines déterministes, on peut construire une machine non déterministe universelle, c'est-à-dire capable de simuler une machine non déterministe quelconque donnée par son code. Mais en utilisant le non-déterminisme, on peut rendre la machine plus rapide (linéaire) par rapport au cas déterministe, au prix d'un espace potentiellement beaucoup plus grand (proportionnel au temps d'exécution). Notons qu'on ne s'intéresse qu'à l'acceptation ou au rejet du mot par la machine. La démonstration s'inspire en partie de celle de la proposition 1-Q pour le cas déterministe, c'est pourquoi nous ne donnons que l'idée détaillée de la preuve.

2-AB **Proposition (machine non déterministe universelle optimale en temps)**

Il existe une machine non déterministe U_{nondet} à 6 rubans, sur l'alphabet d'entrée $\Sigma_U = \{0,1\}$ et l'alphabet de travail $\Gamma_U = \{0,1,B\}$, telle que pour toute machine non déterministe N sur les alphabets Σ_N et Γ_N :

- il existe un morphisme $\varphi_N : \Sigma_N^\star \to \Sigma_U^\star$ tel que pour tout mot $x \in \Sigma_N^\star$, $U_{\text{nondet}}(\langle N\rangle, \varphi_N(x))$ accepte ssi $N(x)$ accepte ;
- il existe une constante α_N telle que pour tout $x \in \Sigma_N^\star$, si $N(x)$ s'arrête en temps t, alors $U_{\text{nondet}}(\langle N\rangle, \varphi_N(x))$ s'arrête en temps $\leqslant \alpha_N t$ (et utilise un espace potentiellement beaucoup plus grand que l'espace utilisé par $N(x)$).

Idée de la démonstration Ce qui ralentissait la machine universelle déterministe de la proposition 1-Q étaient les allers-retours nécessaires de la tête afin d'aller lire les cases pointées par les différentes têtes de lecture de la machine simulée. Afin d'obtenir ici un temps linéaire, plutôt que d'aller chercher ces cases de part et d'autre du ruban, nous allons deviner leur contenu et vérifier plus tard que ce contenu était valide. Nous

donnons maintenant un peu plus de détails. En particulier, nous allons voir qu'on ne peut se contenter d'effectuer la vérification à la fin du calcul car le contenu deviné pourrait mener à un calcul infini : il faut donc vérifier régulièrement qu'on ne se trompe pas.
De la même manière que dans le cas déterministe avec le lemme 1-O, on calcule dans un premier temps le code d'une machine N' équivalente à N (via un morphisme φ_N) mais ayant pour alphabet de travail $\Gamma_U = \{0, 1, B\}$. Le code de N' est maintenant écrit sur le premier ruban de travail. On note k le nombre de rubans de N'.

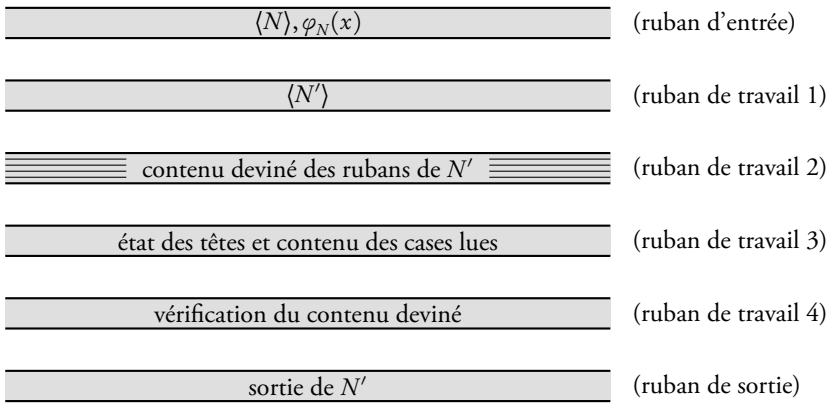


FIGURE 2.3 – Machine universelle non déterministe à 6 rubans simulant $N(x)$.

La fonction des 6 rubans de U_{nondet} est illustrée à la figure 2.3 et ressemble à la construction de la machine universelle déterministe de la proposition 1-Q : en particulier, de la même façon, nous regroupons tous les rubans de travail de N' sur le deuxième ruban de travail de U_{nondet}. Mais ce ruban joue un rôle légèrement différent : il contiendra le contenu *deviné* des rubans de travail de N' et non le contenu réel. Par ailleurs, U_{nondet} possède un quatrième ruban de travail qui servira à vérifier que les valeurs devinées étaient correctes.

Voici comment U_{nondet} simule une étape de N' :

- pour chaque tête d'un ruban de travail de N', U_{nondet} devine le contenu de la case courante ;
- elle note les valeurs devinées sur son deuxième ruban de travail à la suite des contenus précédents (pour pouvoir les vérifier à la fin du calcul) ;
- elle devine un numéro de transition à effectuer ;
- elle inscrit ce numéro de transition sur son deuxième ruban de travail (à la suite du contenu deviné des cases qu'elle vient d'écrire) ;
- selon l'état courant stocké sur le troisième ruban de travail et selon le contenu deviné des cases, elle effectue la transition devinée en lisant le code de N' sur son

premier ruban de travail ;

– enfin, elle met à jour l'état des têtes sur son troisième ruban de travail.

Le deuxième ruban de travail alterne ainsi des blocs de $(k-2)$ cases pour les contenus devinés et des blocs contenant le numéro de la transition effectuée.

À chaque fois que le nombre m d'étapes simulées atteint une puissance de deux (c'est-à-dire après la première étape, après la deuxième, après la quatrième, après la huitième, etc.), on vérifie que les valeurs devinées étaient correctes. Pour cela, on procède de la façon suivante pour chacun des $k-2$ rubans de travail de N' successivement (on désignera par i le numéro du ruban de travail en train d'être traité, pour $1 \leqslant i \leqslant k-2$) :

– en suivant les valeurs devinées et les transitions choisies sur le deuxième ruban de travail, on effectue une nouvelle fois, depuis le début, la simulation des m premières étapes de N' (en lisant son code sur le premier ruban de travail) ;

– à chaque étape, on met à jour le quatrième ruban de travail pour qu'il reflète le contenu du i-ème ruban de travail de N', c'est-à-dire qu'on modifie ses cases et on déplace la tête selon les transitions effectuées ;

– si on rencontre une incompatibilité avec les valeurs devinées (la valeur devinée ne correspond pas à la valeur réelle inscrite sur le ruban), alors on rejette.

Si, après avoir vérifié les $k-2$ rubans de travail, aucune incompatibilité n'a été trouvée : on accepte si le calcul a accepté au cours des m étapes simulées ; on rejette si le calcul a rejeté au cours des m étapes simulées ; sinon on reprend la phase de simulation de N' en devinant le contenu des cases à partir de l'étape $m+1$ (l'historique étant conservé sur le deuxième ruban de travail).
Cette vérification prend un temps $O(mk)$ puisqu'il faut resimuler $(k-2)$ fois les m étapes effectuées.

De cette manière, les seuls chemins acceptants sont ceux pour lesquels les valeurs devinées étaient compatibles avec le vrai calcul de N' (c'est-à-dire que la simulation est correcte) et pour lesquels N' accepte. Ainsi, $N'(x)$ possède un chemin acceptant ssi $U_{\text{nondet}}(\langle N\rangle, \varphi_N(x))$ possède un chemin acceptant.

Enfin, le temps de calcul de U_{nondet} est réparti ainsi :

– un temps constant (dépendant de N mais pas de x) pour calculer $\langle N'\rangle$;

– un temps constant pour chaque étape de N' (deviner le contenu des cases et la transition puis l'effectuer) ;

– un temps $O(k\sum_{j=0}^{d} 2^j)$ pour les vérifications, où d est la plus petite valeur telle que la simulation termine ou provoque une incompatibilité en 2^d étapes : en particulier, $2^d < 2t$.

En tout, U_{nondet} prend un temps linéaire $\alpha_N t$, où α_N dépend de N (et en particulier du nombre de rubans). On notera que l'espace utilisé est également proportionnel à t car on inscrit à chaque étape le contenu des cases devinées sur le deuxième ruban de travail. □

2-AC **Remarque** On pourrait également adapter la preuve du théorème 1-S pour obtenir un temps de simulation $O(t \log t)$ et un espace $O(s)$: on perd en temps mais on gagne en espace.

✎ **2-AD** **Exercice**

Montrer la remarque précédente.

2.2.4 Classes en temps non déterministe

Les machines non déterministes nous permettent de définir les classes NTIME, analogues non déterministes de DTIME.

2-AE **Définition (classes de complexité en temps non déterministe)**

- Pour une fonction $t : \mathbb{N} \to \mathbb{N}$, la classe $\mathsf{NTIME}(t(n))$ est l'ensemble des langages reconnus par une machine de Turing non déterministe N telle qu'il existe une constante α pour laquelle, sur toute entrée x, $N(x)$ fonctionne en temps $\leqslant \alpha t(|x|)$ (c'est-à-dire que *toute* branche de son arbre de calcul sur l'entrée x est de taille majorée par $\alpha t(|x|)$).
- Si $\mathscr{T}$ est un ensemble de fonctions, alors $\mathsf{NTIME}(\mathscr{T})$ désigne $\bigcup_{t\in\mathscr{T}} \mathsf{NTIME}(t(n))$.

Les classes NTIME sont closes par union et intersection.

2-AF **Proposition**

Si $L_1, L_2 \in \mathsf{NTIME}(t(n))$ alors $L_1 \cup L_2 \in \mathsf{NTIME}(t(n))$ et $L_1 \cap L_2 \in \mathsf{NTIME}(t(n))$.

Démonstration Soit N_1 et N_2 des machines non déterministes pour L_1 et L_2 fonctionnant en temps $\alpha_1 t(n)$ et $\alpha_2 t(n)$ respectivement. On trouvera une illustration des arbres de calcul à la figure 2.4.

Pour reconnaître $L_1 \cup L_2$, on considère la machine $N_\cup$ suivante sur l'entrée x :

- `deviner` $b \in \{1,2\}$;
- exécuter $N_b(x)$;
- `accepter` ssi le chemin simulé de $N_b(x)$ accepte.

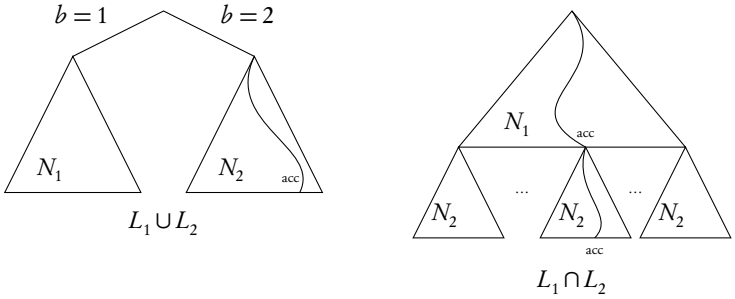


FIGURE 2.4 – Union et intersection pour des machines non déterministes.

On notera que « exécuter $N_b(x)$ » signifie que le comportement (ou l'arbre de calcul) de $N_\cup$ est exactement celui de N_b : cela revient à deviner un chemin et simuler N_b le long de ce chemin.

Si $x \in L_1 \cup L_2$ alors $x \in L_b$ pour un certain $b \in \{1,2\}$. Dans le sous-arbre du calcul de $N_\cup$ correspondant à ce b, il y a un chemin acceptant puisque $N_b(x)$ accepte. Donc $N_\cup(x)$ accepte x puisqu'elle a un chemin acceptant. Réciproquement, si $x \notin L_1 \cup L_2$ alors il n'y a pas de chemin acceptant dans $N_\cup(x)$ car il n'y en a ni dans $N_1(x)$ ni dans $N_2(x)$. Le temps d'exécution de $N_\cup(x)$ est $O(1) + \max(\alpha_1 t(n), \alpha_2 t(n))$ donc $L_1 \cup L_2 \in \mathsf{NTIME}(t(n))$.

Pour reconnaître $L_1 \cap L_2$, on considère la machine $N_\cap$ suivante sur l'entrée x :

- exécuter $N_1(x)$;
- exécuter $N_2(x)$;
- `accepter` ssi les chemins simulés de $N_1(x)$ et $N_2(x)$ acceptent tous les deux.

Si $x \in L_1 \cap L_2$ alors $N_1(x)$ et $N_2(x)$ ont tous deux des chemins acceptants ; dans l'arbre de $N_\cap(x)$, emprunter d'abord le chemin acceptant de $N_1(x)$ puis celui de $N_2(x)$ fournit un chemin acceptant de $N_\cap(x)$. Réciproquement, si $x \notin L_1 \cap L_2$ alors $N_\cap(x)$ n'a pas de chemin acceptant car $N_1(x)$ ou $N_2(x)$ n'en a pas. Le temps d'exécution de $N_\cap(x)$ est $(\alpha_1 + \alpha_2)t(n)$ donc $L_1 \cap L_2 \in \mathsf{NTIME}(t(n))$. □

Un peu de recul

On remarquera que la clôture de NTIME par complémentaire ne figure pas dans la proposition précédente. La raison en est simple : il s'agit d'une question ouverte.

La preuve dans le cas déterministe ne convient pas car le contraire de « il existe un chemin acceptant » est « tous les chemins rejettent », qui n'est plus de la même forme. C'est un point crucial du non-déterminisme : prendre le complémentaire change la nature du

problème car on passe d'un quantificateur existentiel à un quantificateur universel. Nous étudierons cette question plus en détail à la section 2.2.8 et au chapitre sur la hiérarchie polynomiale.

Il **faut** prendre le temps de se convaincre par soi-même qu'il n'est probablement pas possible de passer au complémentaire sans utiliser beaucoup plus de temps de calcul, et bien garder à l'esprit qu'on ne peut pas simplement « inverser » la réponse.

En revanche, si l'on autorise beaucoup plus de temps de calcul alors on peut prendre le complémentaire. Plus généralement, on peut simuler le fonctionnement d'une machine de Turing non déterministe par une machine déterministe fonctionnant en un temps exponentiellement plus élevé.

2-AG Proposition

Pour toute fonction $t : \mathbb{N} \to \mathbb{N}$, $\mathsf{DTIME}(t(n)) \subseteq \mathsf{NTIME}(t(n)) \subseteq \mathsf{DTIME}(2^{O(t(n))})$.

Idée de la démonstration Pour simuler une machine non déterministe fonctionnant en temps $t(n)$ par une machine déterministe, on parcourt tous les chemins de calcul et on détermine si l'un d'entre eux accepte. Puisqu'il y a un nombre exponentiel de chemins, la machine déterministe prend un temps $2^{O(t(n))}$.

Démonstration La première inclusion découle de l'observation qu'une machine déterministe est un cas particulier de machine non déterministe.

Pour la seconde inclusion, on simule une machine non déterministe N fonctionnant en temps $\alpha t(n)$ par une machine déterministe de la façon suivante [1] :

- `pour` tout chemin y de taille $\alpha t(n)$ `faire`
  - simuler $N(x)$ sur le chemin y,
  - `accepter` si la simulation accepte ;
- `rejeter` (ici, tous les chemins ont rejeté).

Pour y fixé, le corps de boucle prend un temps déterministe $O(t(n))$. Il reste à compter le nombre de chemins y à tester. Chaque transition de la machine non déterministe a au plus $R = |Q||\Gamma|^{k-1}3^k$ possibilités (il s'agit du choix d'un état, des lettres à écrire et du déplacement des têtes). Ainsi il y a au plus $R^{\alpha t(n)}$ chemins y possibles, soit $2^{\alpha t(n) \log R}$. Puisque R est une constante indépendante de l'entrée x (elle dépend seulement de N), le temps mis par la machine déterministe est $t(n)2^{O(t(n))}$, soit $2^{O(t(n))}$. □

1. Ici on suppose pour simplifier que $\alpha t(n)$ est constructible en temps ; l'objet de l'exercice 2-AH est de voir que cette condition n'est pas nécessaire.

✎ **2-AH** **Exercice**

Dans la démonstration qui précède, nous avons supposé que la fonction $t(n)$ est constructible en temps. Refaire la démonstration sans cette hypothèse.

Indication : construire chaque chemin y au fur et à mesure jusqu'à ce que la simulation de N s'arrête ; effectuer alors un backtracking *pour reprendre l'énumération des autres chemins y.*

2.2.5 Théorème de hiérarchie en temps non déterministe

Il n'est guère surprenant que les classes en temps non déterministe admettent également un théorème de hiérarchie : on sait résoudre plus de problèmes si l'on dispose de plus de temps. Il est même plus « serré » que son analogue déterministe. Cependant, la preuve est plus subtile : on ne peut plus procéder comme dans le cas déterministe puisqu'on ne sait pas prendre efficacement la réponse opposée à une machine non déterministe, comme nous l'avons déjà vu… Le premier théorème de hiérarchie non déterministe est dû à Cook [Coo72] mais avec un plus grand écart entre f et g ; la version ci-dessous a été montrée par Seiferas, Fischer and Meyer [SFM78] et la preuve que nous donnons est due à Žák [Žák83].

2-AI **Théorème (hiérarchie en temps non déterministe)**

Soit $f, g : \mathbb{N} \to \mathbb{N}$ des fonctions telles que $f(n) \neq 0$ (pour tout $n \in \mathbb{N}$), g est croissante et constructible en temps et $f(n+1) = o(g(n))$. Alors $\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n))$.

Idée de la démonstration Puisqu'on ne sait pas prendre le complémentaire efficacement dans un calcul non déterministe, on va attendre d'avoir assez de temps pour faire une simulation déterministe. Pour cela, il faut attendre de travailler sur une entrée de taille exponentielle ; la partie subtile consiste ensuite à trouver un moyen de propager l'information jusqu'à l'entrée courante. Le fait d'attendre pour prendre le complémentaire donne le nom de « diagonalisation retardée » à cette méthode.

Démonstration Nous allons décrire un langage $L \in \mathsf{NTIME}(g(n)) \setminus \mathsf{NTIME}(f(n))$, non pas directement mais en donnant une machine non déterministe V qui reconnaît L. Il sera constitué de triplets $(\langle N \rangle, 1^i, x)$: un code de machine non déterministe N, un entier i donné en unaire et un mot x quelconque. En supposant qu'une machine N décide L en temps $O(f(n))$, l'objectif est d'aboutir à la chaîne d'égalités suivante (où k est grand entier) et donc à une contradiction :

$$N(\langle N \rangle, \epsilon, x) = N(\langle N \rangle, 1, x) = \cdots = N(\langle N \rangle, 1^k, x) = \neg N(\langle N \rangle, \epsilon, x).$$

Remarquons tout d'abord que pour n assez grand, $2^{g(n)} \geqslant 2^{g(n-1)}$ est une borne supérieure sur le temps de simulation par une machine déterministe fixée d'une machine non déterministe quelconque N fonctionnant en temps $O(f(n))$ et dont le code est

donné en entrée : comme à la proposition 2-AG il suffit de simuler de manière déterministe la machine universelle U_{nondet} (proposition 2-AB) qui exécute N.
La machine V pour le langage L a le comportement suivant sur l'entrée $(\langle N\rangle, 1^i, x)$ de taille n :

- on note m la taille du triplet $(\langle N\rangle, \epsilon, x)$ (correspondant à $i = 0$) ;
- si $i \geqslant 2^{g(m)}$ alors on simule $N(\langle N\rangle, \epsilon, x)$ *de manière déterministe* pendant $2^{g(m)}$ étapes déterministes, et on accepte ssi tous les chemins rejettent (on renvoie donc l'inverse de $N(\langle N\rangle, \epsilon, x)$) ;
- si $i < 2^{g(m)}$ alors on simule *de manière non déterministe*

$$N(\langle N\rangle, 1^{i+1}, x) :$$

plus précisément, on exécute la machine universelle non déterministe $U_{\text{nondet}}(\langle N\rangle, (\langle N\rangle, 1^{i+1}, x))$ pendant $g(n)$ étapes (U_{nondet} est la machine de la proposition 2-AB).

Puisque g est constructible en temps et croissante, on sait calculer $g(m)$ et $g(n)$ en temps $O(g(n))$. Le temps mis par la machine V pour x assez grand est donc :

- si $i \geqslant 2^{g(m)}$ alors la simulation déterministe de $N(\langle N\rangle, \epsilon, x)$ prend $2^{g(m)}$ étapes, c'est-à-dire moins de i, donc V fonctionne en temps linéaire ;
- sinon la machine universelle tourne pendant $g(n)$ étapes.

Dans tous les cas [2], V prend un temps $O(g(n))$ donc $L \in \mathsf{NTIME}(g(n))$.
Supposons maintenant que $L \in \mathsf{NTIME}(f(n))$ pour une fonction f telle que $f(n+1) = o(g(n))$: il existe donc une machine non déterministe N qui reconnaît L en temps $\alpha f(n)$. Par hypothèse, pour x suffisamment grand, la taille n de l'entrée $(\langle N\rangle, 1^i, x)$ satisfait $g(n) > \alpha_N \alpha f(n+1)$, où α_N est la constante de la machine universelle U_{nondet} de la proposition 2-AB. Ainsi, exécuter U_{nondet} pendant $g(n)$ étapes revient à effectuer au moins $\alpha f(n+1)$ étapes de $N(\langle N\rangle, 1^{i+1}, x)$; puisque N est supposée fonctionner en temps αf et que la taille de son entrée est ici $n+1$, cela signifie que la simulation de N est complète.
Donc, par définition du langage L et pour x suffisamment grand, $N(\langle N\rangle, \epsilon, x) = N(\langle N\rangle, 1, x)$ puisque sur l'entrée $(\langle N\rangle, \epsilon, x)$ on simule complètement $N(\langle N\rangle, 1, x)$. De même, tant que $i < 2^{g(m)}$, $N(\langle N\rangle, 1^i, x) = N(\langle N\rangle, 1^{i+1}, x)$. Finalement, pour $i = 2^{g(m)}$ on a $N(\langle N\rangle, 1^i, x) = \neg N(\langle N\rangle, \epsilon, x)$ puisqu'on simule $\neg N(\langle N\rangle, \epsilon, x)$ de manière déterministe. Au final, on a la chaîne d'égalités suivante :

$$N(\langle N\rangle, \epsilon, x) = N(\langle N\rangle, 1, x) = \cdots = N(\langle N\rangle, 1^{2^{g(m)}}, x) = \neg N(\langle N\rangle, \epsilon, x),$$

une contradiction. □

2. En effet, les hypothèses $f(n) \neq 0$, g croissante et constructible en temps, et $f(n+1) = o(g(n))$ impliquent $g(n) = \Omega(n)$: cf. exercice 2-AJ.

✎ **2-AJ** **Exercice**

Montrer la propriété mentionnée en note de bas de page numéro 2 : si g est croissante et constructible en temps, si $f(n+1) = o(g(n))$, et si pour tout $n \in \mathbb{N}$ $f(n) \neq 0$, alors $g(n) = \Omega(n)$.

2.2.6 Temps non déterministe polynomial et exponentiel

Nous passons maintenant à la définition de deux classes de complexité non déterministes importantes, NP et NEXP.

2-AK **Définition (temps non déterministe polynomial et exponentiel)**

La classe NP est l'ensemble des langages reconnus par une machine non déterministe en temps polynomial, c'est-à-dire

$$\mathsf{NP} = \mathsf{NTIME}(n^{O(1)}) = \cup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k).$$

La classe NEXP est l'ensemble des langages reconnus par une machine non déterministe en temps exponentiel, c'est-à-dire

$$\mathsf{NEXP} = \mathsf{NTIME}(2^{n^{O(1)}}) = \cup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{n^k}).$$

Enfin, une dernière classe que l'on étudiera moins est notée NE, il s'agit de l'ensemble des langages reconnus en temps non déterministe exponentiel avec exposant linéaire, c'est-à-dire :

$$\mathsf{NE} = \mathsf{NTIME}(2^{O(n)}) = \cup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{kn}).$$

2-AL **Remarque** Attention, une erreur souvent commise est de croire que NP signifie « non polynomial » alors que c'est une abréviation pour *Nondeterministic Polynomial time*.

2-AM **Exemple** Voici quelques exemples de problèmes dans NP et dans NEXP (nous verrons d'autres problèmes NP au chapitre suivant).

1. `CLIQUE` :
   - *entrée* : un graphe non orienté G et un entier k ;
   - *question* : G a-t-il une clique de taille k ?

   Une clique de taille k est un ensemble de k sommets tous reliés les uns aux autres. Ce problème est dans NP : une machine non déterministe polynomiale pour le reconnaître peut deviner un ensemble de k sommets puis vérifier que ces k sommets sont tous reliés entre eux. Nous verrons que ce problème est « NP-complet ».

2. SOMME PARTIELLE :
   - *entrée* : une liste d'entiers $a_1, \dots, a_m$ et un entier cible t ;
   - *question* : existe-t-il une sous-liste dont la somme vaut t ? En d'autres termes, existe-t-il $S \subseteq \{1, \dots, m\}$ tel que $\sum_{i \in S} a_i = t$?

   Ce problème est dans NP : il suffit de deviner les éléments de l'ensemble S et de vérifier si la somme des entiers correspondants vaut t. Il est appelé SUBSET SUM en anglais. Comme le précédent, ce problème est NP-complet.
3. ARRÊT NEXP :
   - *entrée* : le code d'une machine non déterministe N et un entier k en binaire ;
   - *question* : est-ce que $N(\epsilon)$ s'arrête en $\leqslant k$ étapes ?

   Ce problème est dans NEXP car on peut simuler $N(\epsilon)$ pendant k étapes grâce à une machine universelle non déterministe : puisque la taille de l'entrée k est $\log k$, cela prend un temps exponentiel. Nous n'avons pas encore vu cette notion, mais mentionnons au passage que ce problème est NEXP-complet.
4. Un problème de NEXP plus « naturel » est, sur l'entrée C, de décider s'il existe une instanciation des variables satisfaisant le circuit booléen D dont le code est décrit par le circuit booléen C donné en entrée (nous définirons les circuits au chapitre 5). Puisque D peut être de taille exponentielle, il s'agit de deviner une instanciation de taille exponentielle et d'évaluer le circuit, ce qui se fait dans NEXP. Comme le précédent, c'est un problème NEXP-complet (exercice B-C).

✎ **2-AN** **Exercice**

Justifier plus rigoureusement que ces problèmes sont chacun dans la classe annoncée.

Pour décider les deux exemples précédents de langages de NP, nos machines n'utilisaient leur non-déterminisme que pour deviner un mot, puis elles vérifiaient une propriété facile sur ce mot. Cette remarque est en fait générale : les classes NP et NEXP ont une belle caractérisation utilisant un quantificateur existentiel. On supposera toujours que l'alphabet Σ contient au moins deux symboles notés 0 et 1.

2-AO **Proposition (caractérisation existentielle de NP et NEXP)**

- Un langage A est dans NP ssi il existe un polynôme $p(n)$ et un langage $B \in \mathsf{P}$ tels que
$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)} \ (x, y) \in B.$$
- Un langage A est dans NEXP ssi il existe un polynôme $p(n)$ et un langage $B \in \mathsf{P}$ tels que
$$x \in A \iff \exists y \in \{0,1\}^{2^{p(|x|)}} \ (x, y) \in B.$$

Le mot y justifiant l'appartenance de x à A est appelé *preuve* ou *certificat*.

Idée de la démonstration Le certificat y correspond au chemin acceptant dans la machine non déterministe reconnaissant le langage A.

Démonstration Nous faisons en détail le cas de NP puis verrons comment l'adapter à NEXP.

$\Rightarrow$ Soit N une machine non déterministe polynomiale reconnaissant A : on suppose qu'elle fonctionne en temps $q(n)$. À chaque étape, N a le choix entre un nombre constant R de transitions possibles : R dépend seulement de N, pas de l'entrée x. Ainsi, l'arbre de calcul de $N(x)$ est un arbre de hauteur $q(n)$ et tel que chaque nœud a au plus R fils : un chemin dans un tel arbre peut être encodé par un mot $y \in \{0,1\}^\star$ de taille $q(n)\lceil \log R \rceil$ (une succession de $q(n)$ numéros de fils). Soit p le polynôme défini par $p(n) = q(n)\lceil \log R \rceil$.
On définit alors B l'ensemble des couples (x,y) où $y \in \{0,1\}^{p(|x|)}$ est tel que le chemin encodé par y dans le calcul $N(x)$ accepte (si y n'encode pas un chemin valide, on décrète que $(x,y) \notin B$). Puisque $x \in A$ ssi il existe un chemin acceptant, on a bien

$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)}\ (x,y) \in B.$$

Il ne reste plus qu'à montrer que $B \in \mathsf{P}$. Pour décider si $(x,y) \in B$, il s'agit d'exécuter $N(x)$ le long du chemin y : puisque l'on connaît le chemin, cela revient à exécuter une machine déterministe fonctionnant en temps $q(n)$. Donc $B \in \mathsf{P}$.
$\Leftarrow$ Soit $B \in \mathsf{P}$ reconnu par une machine déterministe M fonctionnant en temps polynomial $q(n)$ et supposons que A soit défini par

$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)}\ (x,y) \in B.$$

Montrons que $A \in \mathsf{NP}$. Voici une machine non déterministe pour A sur l'entrée x :

- `deviner` $y \in \{0,1\}^{p(|x|)}$;
- `accepter` ssi $M(x,y)$ accepte.

Cette machine non déterministe fonctionne en temps polynomial $p(n)+q(n+p(n))$ puisque deviner y prend un temps $p(n)$, puis M fonctionne en temps q sur une entrée (x,y) de taille $n+p(n)$. Donc $A \in \mathsf{NP}$.
Pour NEXP : la même preuve s'applique à ceci près que le certificat y doit être de taille exponentielle puisque la machine N fonctionne maintenant en temps exponentiel. Mais attention, le langage B reste dans P car l'entrée (x,y) de B est maintenant de taille exponentielle et donc l'exécution de $N(x)$ selon le chemin y prend un temps polynomial (et même linéaire) en la taille de (x,y). □

Nous utiliserons souvent cette caractérisation existentielle par la suite.

✎ **2-AP** **Exercice**

Montrer qu'on obtient la même caractérisation de NP si l'on ne demande plus au certificat y d'être de taille exactement $p(n)$ mais « au plus $p(n)$ ».

✎ **2-AQ** **Exercice**

Le fait de demander à ce que le langage sous-jacent soit dans P dans la caractérisation existentielle de NEXP peut sembler étrange à première vue. Mais les deux variantes suivantes ne capturent pas NEXP.

– Soit $\mathscr{C}$ l'ensemble des langages A tels qu'il existe un polynôme $p(n)$ et un langage $B \in \mathsf{EXP}$ vérifiant

$$x \in A \iff \exists y \in \{0,1\}^{2^{p(|x|)}} \, (x,y) \in B.$$

Montrer que $\mathscr{C} = \mathsf{DTIME}(2^{2^{n^{O(1)}}})$ (temps déterministe doublement exponentiel).
Indication : on pourra d'abord se familiariser avec le padding à la proposition 2-AU et à l'exercice 2-AW.

– Soit $\mathscr{C}$ l'ensemble des langages A tels qu'il existe un polynôme $p(n)$ et un langage $B \in \mathsf{EXP}$ vérifiant

$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)} \, (x,y) \in B.$$

Montrer que $\mathscr{C} = \mathsf{EXP}$.

Un peu de recul

Si P est la classe des problèmes pour lesquels on peut trouver une solution efficacement, NP est la classe des problèmes pour lesquels on peut *vérifier* une solution efficacement. En effet, on nous donne une preuve y et on doit *vérifier* en temps polynomial que cette preuve est correcte. En d'autres termes, on nous donne une solution potentielle (un chemin acceptant) et on doit vérifier qu'il s'agit en effet d'une solution.

On retrouve cela dans les deux exemples précédents de problèmes NP : pour `CLIQUE`, on choisissait (de manière non déterministe) l'ensemble formant la clique et il fallait vérifier qu'il formait bien une clique ; pour `SOMME PARTIELLE`, on choisissait (de manière non déterministe) le sous-ensemble et il fallait vérifier que la somme valait bien t. Ces deux exemples illustrent un phénomène général :

NP est la classe des problèmes « faciles à vérifier ».

Nous pouvons maintenant comparer les quatre classes vues jusqu'à présent. La proposition 2-AG nous donne les inclusions suivantes.

2-AR **Corollaire**

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$$

Par ailleurs, le théorème de hiérarchie en temps non déterministe 2-AI nous donne la séparation suivante.

2-AS **Corollaire**

$$\mathsf{NP} \subsetneq \mathsf{NEXP}$$

Démonstration Il suffit de remarquer que $\mathsf{NP} \subseteq \mathsf{NTIME}(2^n)$ puis, par le théorème 2-AI, que $\mathsf{NTIME}(2^n) \subsetneq \mathsf{NTIME}(2^{n^2})$ et enfin que $\mathsf{NTIME}(2^{n^2}) \subseteq \mathsf{NEXP}$. □

✎ **2-AT** **Exercice**

Séparer de même NE et NP.

Pour résumer, nous savons que

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}, \qquad \mathsf{EXP} \neq \mathsf{P} \qquad \text{et} \qquad \mathsf{NEXP} \neq \mathsf{NP}.$$

Cependant, c'est une question ouverte de savoir si chacune des inclusions $\mathsf{P} \subseteq \mathsf{NP}$, $\mathsf{NP} \subseteq \mathsf{EXP}$ et $\mathsf{EXP} \subseteq \mathsf{NEXP}$ est stricte.

2.2.7 Le problème « P = NP? »

En particulier, la question de savoir si $\mathsf{P} = \mathsf{NP}$ est ouverte et centrale en complexité. Pour reprendre la caractérisation précédente, cela revient à savoir si trouver une solution est aussi simple [3] que de vérifier une solution. Par exemple, est-il aussi facile de trouver une démonstration d'un énoncé mathématique que de vérifier qu'une démonstration donnée est correcte ? Est-il aussi facile de remplir une grille de Sudoku que de vérifier qu'un remplissage donné est bien une solution ? On pourrait bien sûr multiplier les exemples.

Notre expérience semble indiquer qu'il est plus difficile de trouver une solution car il faut faire preuve d'imagination ou d'intuition : c'est une des raisons pour lesquelles de nombreuses personnes pensent que $\mathsf{P} \neq \mathsf{NP}$. Nous exposerons une autre raison au prochain chapitre : malgré énormément d'efforts, nous ne connaissons pas d'algorithme polynomial pour une vaste classe de problèmes de NP.

3. Dans cette discussion, « simple » ou « facile » signifie efficace : ce pourrait être un algorithme très compliqué pourvu qu'il soit rapide.

Enfin, nous terminons ce chapitre en montrant les liens entre la question $\mathsf{P} = \mathsf{NP}$ et son analogue exponentiel, $\mathsf{EXP} = \mathsf{NEXP}$.

2-AU Proposition

$$\mathsf{P} = \mathsf{NP} \implies \mathsf{EXP} = \mathsf{NEXP}$$

Idée de la démonstration La technique s'appelle *padding* (qu'on pourrait traduire par « rembourrage »). Il s'agit d'augmenter délibérément la longueur de l'entrée pour que les machines disposent de plus de temps, puisque le temps est calculé en fonction de la taille de l'entrée.

Ici, on augmente exponentiellement l'entrée d'un langage de NEXP pour le ramener dans NP et pouvoir appliquer l'hypothèse.

Démonstration On suppose $\mathsf{P} = \mathsf{NP}$. Soit $L \in \mathsf{NEXP}$. Il est reconnu par une machine non déterministe N fonctionnant en temps 2^{n^k} pour un certain k. On définit $\tilde{L} = \{(x, 1^{2^{|x|^k}}) \mid x \in L\}$, c'est-à-dire qu'on augmente la taille de l'entrée avec un nombre exponentiel de symboles 1. La nouvelle entrée a maintenant une taille $m \geqslant 2^{|x|^k}$.

Voici une machine non déterministe $\tilde{N}$ pour $\tilde{L}$:

- vérifier que l'entrée est de la forme (x, y) (sinon `rejeter`) ;
- vérifier que $y = 1^{2^{|x|^k}}$ (sinon `rejeter`) ;
- exécuter $N(x)$.

Cette machine reconnaît $\tilde{L}$ en temps $O(2^{n^k})$, c'est-à-dire en temps non déterministe linéaire en m (la taille de l'entrée). Ainsi, $\tilde{L} \in \mathsf{NP}$: par hypothèse, on en déduit que $\tilde{L} \in \mathsf{P}$. Il existe donc une machine déterministe $\tilde{M}$ fonctionnant en temps polynomial pour $\tilde{L}$. Cette machine fonctionne en temps $m^{k'}$.

Voici maintenant une machine déterministe M pour L sur l'entrée x :

- écrire $(x, 1^{2^{|x|^k}})$ sur un ruban de travail ;
- exécuter $\tilde{M}(x, 1^{2^{|x|^k}})$.

La première étape prend un temps $O(2^{n^k})$; la seconde étape prend un temps $m^{k'}$ où $m = 2^{O(n^k)}$ est la taille de $(x, 1^{2^{|x|^k}})$. En tout, M fonctionne en temps $2^{O(n^k)}$, donc $L \in \mathsf{EXP}$. □

2-AV **Remarque** La réciproque de la proposition précédente n'est pas connue.

Nous proposons l'exercice suivant pour manipuler les techniques de padding.

✎ **2-AW** **Exercice**

Un langage L est dit *creux* s'il existe un polynôme $p(n)$ tel que pour tout $n \in \mathbb{N}$, $|L^{=n}| \leqslant p(n)$. Il est dit *unaire* si $L \subseteq 1^\star$ (tout mot de L n'est composé que de symboles 1).

Montrer que les trois assertions suivantes sont équivalentes :

- $\mathsf{E} \neq \mathsf{NE}$;
- $\mathsf{NP} \setminus \mathsf{P}$ contient un langage creux ;
- $\mathsf{NP} \setminus \mathsf{P}$ contient un langage unaire.

Indication : montrer $1 \Longrightarrow 2 \Longrightarrow 3 \Longrightarrow 1$. *Pour* $2 \Longrightarrow 3$, *encoder les bits en nombre polynomial des mots de* $A^{=n}$ *dans un langage unaire.*

2.2.8 Complexité du complémentaire

Le caractère asymétrique de la définition de NTIME appelle quelques remarques. Le non-déterminisme est une notion naturelle dont l'importance sera approfondie au chapitre suivant. Mais le fait qu'on ne sache pas passer efficacement au complémentaire amène la définition de nouvelles classes de complexité.

2-AX **Définition**

Si $\mathscr{C}$ est une classe de complexité, alors la classe $\mathsf{co}\mathscr{C}$ est l'ensemble des complémentaires des langages de $\mathscr{C}$. En d'autres termes,

$$\mathsf{co}\mathscr{C} = \{{}^{c}A \mid A \in \mathscr{C}\}.$$

On obtient ainsi la classe coNP par exemple, l'ensemble des langages dont le complémentaire est dans NP. Quitte à transformer les chemins acceptants en chemins rejetant dans un machine NP et inversement, on peut reformuler la définition de coNP comme suit.

2-AY **Proposition** (coNP)

La classe coNP est l'ensemble des langages A tels qu'il existe une machine de Turing non déterministe N fonctionnant en temps polynomial et satisfaisant :

$$x \in A \text{ ssi tous les chemins de calcul de } N(x) \text{ sont acceptants.}$$

Comme on l'a vu auparavant, on ne sait pas décider dans NP le complémentaire de tout langage de NP : cela se traduit par le fait que la question « NP = coNP ? » est ouverte. Il est utile de réfléchir un moment à cette question pour comprendre l'essence du non-déterminisme.

2-AZ **Remarque** Formellement, le complémentaire dans $\Sigma^\star$ du langage CLIQUE par exemple (cf. exemple 2-AM) est l'ensemble des mots x :

- soit qui ne codent pas un couple (G,k) (un graphe et un entier) ;
- soit qui codent un couple (G,k) tel que G ne possède pas de clique de taille k.

Cependant, lorsqu'on parle du complémentaire coCLIQUE du problème CLIQUE, on désigne le langage $\{(G,k) \mid G$ n'a pas de clique de taille $k\}$: on garde ainsi les mêmes entrées et on inverse simplement la question. En effet, on parle du complémentaire dans l'ensemble $\{(G,k) \mid G$ est un graphe et k un entier$\}$ (l'ensemble des entrées valides pour ce problème) et non dans l'ensemble $\Sigma^\star$.

Par rapport à prendre le complémentaire dans $\Sigma^\star$, cela ne change pas la complexité s'il est facile de décider qu'un mot code une entrée « valide » du langage (par exemple un graphe, un couple d'entiers, etc.), comme c'est toujours le cas.

De la même manière qu'à la proposition 2-AO, on peut donner une caractérisation de coNP en termes de quantification universelle : on remarque que par rapport à NP, on remplace simplement le quantificateur $\exists$ par un quantificateur $\forall$.

2-BA **Proposition (caractérisation universelle de coNP)**

Un langage A est dans coNP ssi il existe un polynôme $p(n)$ et un langage $B \in \mathsf{P}$ tels que

$$x \in A \iff \forall y \in \{0,1\}^{p(|x|)}\ (x,y) \in B.$$

Démonstration Soit $A \in$ coNP : par définition, ${}^cA \in$ NP, donc d'après la proposition 2-AO, il existe $B' \in \mathsf{P}$ et un polynôme $p(n)$ tels que

$$x \in {}^cA \iff \exists y \in \{0,1\}^{p(|x|)}\ (x,y) \in B'.$$

On en déduit que

$$x \in A \iff \forall y \in \{0,1\}^{p(|x|)}\ (x,y) \notin B'.$$

Il suffit alors de prendre $B = {}^cB'$ (qui est bien un langage de P) pour obtenir notre caractérisation. □

On déduit en particulier de cette proposition que $\mathsf{P} \subseteq$ coNP. Le fait que NP n'est probablement pas clos par complémentaire donnera naissance à la hiérarchie polynomiale qu'on étudiera au chapitre 8.

On pourra consulter l'exercice B-A en annexe pour s'entraîner davantage sur le padding et les théorèmes de hiérarchie.

3 NP-complétude

Bien que la question « P = NP ? » résiste depuis plusieurs décennies, la quantité d'efforts investis a permis d'avoir une compréhension bien meilleure du problème. La notion de NP-complétude fait partie de ces avancées majeures, une des premières : elle a d'abord montré l'importance incroyable de la classe NP avant de donner un moyen aux algorithmiciens de contourner le problème « P = NP ? ».

En effet, nous allons voir qu'un très grand nombre de problèmes naturels et importants en pratique sont tous équivalents et sont les plus difficiles de la classe NP : cela implique qu'ils ne possèdent pas d'algorithme polynomial si P ≠ NP. Puisque la croyance populaire soutient que P ≠ NP, on se contente de ces résultats comme d'une espèce de « preuve » que le problème est difficile et qu'on ne peut pas espérer obtenir un algorithme efficace [1].

Pour affirmer que certains problèmes « sont les plus difficiles », il faut pouvoir comparer les problèmes entre eux. On définit pour cela la notion de *réduction* qui tire ses origines de la calculabilité. Il existe beaucoup de réductions différentes mais nous ne définirons dans ce chapitre que l'un des deux types les plus importants. Puis nous verrons plusieurs problèmes NP-complets avant de voir le théorème 3-AK de Ladner et celui de Mahaney 3-AQ.

3.1 Réductions

Les notions de réduction permettent de comparer deux problèmes A et B. Grosso-modo, on dira que B est plus facile que A si on peut décider *efficacement* l'appartenance à B dès lors qu'on possède *un moyen* de tester l'appartenance à A : en effet, dans ce cas, résoudre

1. En réalité on a vu que la classe P ne reflétait qu'imparfaitement l'ensemble des problèmes « faciles ». En conséquence les algorithmiciens ne s'arrêtent pas à un résultat de NP-complétude : ils peuvent chercher des solutions approchées, des algorithmes qui fonctionneront sur « la plupart » des instances rencontrées, programmer des méthodes efficaces pour les petites entrées, etc.

A permet de résoudre B et donc B n'est pas plus difficile que A. Les différentes notions de réduction varient par ce qu'on entend par « efficacement » et selon le « moyen » de résoudre A.

En d'autres termes, pour résoudre B on « se ramène à » A : c'est en ce sens que B est plus simple que A. Plutôt que « se ramener à », on utilise le terme *se réduire à* : on dira que B se réduit à A, ou encore qu'il y a une réduction de B à A.

3-A Définition (réductions many-one polynomiales)

Une *réduction many-one* en temps polynomial d'un problème B (sur l'alphabet Σ_B) à un problème A (sur l'alphabet Σ_A) est une fonction $f : \Sigma_B^\star \to \Sigma_A^\star$ calculable en temps polynomial telle que :

$$\forall x \in \Sigma_B^\star, \quad x \in B \iff f(x) \in A.$$

Si une telle fonction f existe, on dira que B *se réduit à* A (via f) et on notera $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$.

3-B Remarques Quelques commentaires sur cette définition :

- La résolution de B se ramène à la résolution de A par le calcul de f : il suffit d'un précalcul simple (polynomial), c'est en ce sens que B est plus simple que A. La réduction transforme une instance x du problème B en une instance x' du problème A telles que $x \in B$ ssi $x' \in A$.
- On notera que la définition est équivalente à $B = f^{-1}(A)$.
- Le nom *many-one* vient de ce que plusieurs mots $x \in \Sigma_B^\star$ peuvent avoir la même image.
- Nous avons défini la réduction many-one en temps polynomial : on pourrait définir d'autres réductions many-one selon l'efficacité du calcul de f. Lorsqu'on parlera de réductions many-one sans préciser la complexité de f, on désignera les réductions many-one en temps polynomial.
- **Attention**, le terme « se réduire à » peut être trompeur puisque habituellement on se ramène à une tâche plus simple. Ici, lorsque B se réduit à A, c'est bien B qui est plus simple !

La plupart des classes de complexité abordées dans cet ouvrage seront closes pour les réduction many-one (c'est-à-dire que $A \in \mathscr{C}$ et $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$ impliquent $B \in \mathscr{C}$), notamment P, NP, EXP et NEXP. Une exception notable sont les classes E et NE et c'est la raison pour laquelle elles sont moins étudiées.

3-C Proposition

Les classes P et NP sont closes pour $\leqslant_{\mathsf{m}}^{\mathsf{p}}$.

Idée de la démonstration Si $B \leqslant_{\mathsf{m}}^{\mathsf{P}} A$ via f, pour décider B il suffit de calculer $f(x)$ et de décider si $f(x) \in A$.

Démonstration Pour la classe P : soit $A \in \mathsf{P}$ et $B \leqslant_{\mathsf{m}}^{\mathsf{P}} A$ via f, montrons que $B \in \mathsf{P}$. Soit k un entier suffisamment grand pour que f soit calculable en temps n^k et A soit décidable en temps n^k. Voici un algorithme pour B sur l'entrée $x \in \Sigma_B^\star$:

- Calculer $f(x)$.
- Décider si $f(x) \in A$.

Cet algorithme décide B par définition de la réduction f de B à A, et il fonctionne en temps polynomial :

- le calcul de f prend un temps $\leqslant n^k$;
- puisque $|f(x)| \leqslant n^k$, décider si $f(x) \in A$ prend un temps $\leqslant (n^k)^k = n^{k^2}$.

Le temps total est $O(n^{k^2})$, donc $B \in \mathsf{P}$.
Pour la classe NP, le raisonnement est exactement le même. Si A est reconnu par une machine non déterministe polynomiale N_A, la machine non déterministe pour B fonctionne ainsi sur l'entrée x :

- Calculer $f(x)$.
- Exécuter $N_A(f(x))$.

□

3-D **Remarque** La clôture de la classe P pour $\leqslant_{\mathsf{m}}^{\mathsf{P}}$ signifie qu'un problème B plus simple qu'un problème A efficacement résoluble est lui-même efficacement résoluble, conformément à l'intuition.

✎ **3-E** **Exercice**

Montrer que les classes EXP et NEXP sont closes pour $\leqslant_{\mathsf{m}}^{\mathsf{P}}$.

✎ **3-F** **Exercice**

Montrer que les classes E et NE ne sont pas closes pour $\leqslant_{\mathsf{m}}^{\mathsf{P}}$.

Indication : par théorème de hiérarchie, on sait que $\mathsf{EXP} \neq \mathsf{E}$. *Utiliser un problème* $L \in \mathsf{EXP} \setminus \mathsf{E}$, *le « padder » (cf. proposition 2-AU) pour obtenir un langage* $L' \in \mathsf{E}$ *: alors* $L \leqslant_{\mathsf{m}}^{\mathsf{P}} L'$.

Il est immédiat de constater que $\leqslant^{\mathrm{p}}_{\mathrm{m}}$ est une relation de pré-ordre sur l'ensemble des langages, c'est-à-dire qu'elle est réflexive et transitive.

3-G Lemme

La relation $\leqslant^{\mathrm{p}}_{\mathrm{m}}$ est réflexive et transitive.

Démonstration Réflexivité : soit A un langage, alors $A \leqslant^{\mathrm{p}}_{\mathrm{m}} A$ via l'identité.
Transitivité : soit A, B et C des langages tels que $A \leqslant^{\mathrm{p}}_{\mathrm{m}} B$ via f et $B \leqslant^{\mathrm{p}}_{\mathrm{m}} C$ via g. Alors $x \in A$ ssi $f(x) \in B$ ssi $g(f(x)) \in C$, donc $A \leqslant^{\mathrm{p}}_{\mathrm{m}} C$ via $g \circ f$. □

On en déduit la relation d'équivalence suivante, signifiant que deux langages ont même difficulté.

3-H Définition

Si deux langages A et B vérifient $A \leqslant^{\mathrm{p}}_{\mathrm{m}} B$ et $B \leqslant^{\mathrm{p}}_{\mathrm{m}} A$, on notera $A \equiv^{p}_{m} B$ et on dira que A et B sont équivalents pour les réductions many-one polynomiales.

Voici un exemple très simple de réductions.

3-I Exemple Soit ENSEMBLE INDÉPENDANT le problème suivant :

- *entrée* : un graphe non orienté G et un entier k ;
- *question* : existe-t-il un ensemble de k sommets indépendants, c'est-à-dire tous non reliés deux à deux ?

Ce problème semble très proche du problème CLIQUE présenté au chapitre précédent (exemple 2-AM). En effet, nous allons montrer que ENSEMBLE INDÉPENDANT $\equiv^{p}_{m}$ CLIQUE.
Pour un graphe $G = (V, E)$, on notera $\bar{G}$ le graphe $(V, {}^{c}E)$ sur les mêmes sommets mais dont les arêtes sont complémentaires à celles de G : x et y sont reliés dans $\bar{G}$ ssi ils ne sont pas reliés dans G (voir figure 3.1). On remarque que le calcul de $\bar{G}$ à partir de G se fait en temps polynomial.
Pour réduire CLIQUE à ENSEMBLE INDÉPENDANT, il s'agit de transformer une instance (G, k) de CLIQUE en une instance (G', k') de ENSEMBLE INDÉPENDANT. La réduction f que nous proposons est définie par $f(G, k) = (\bar{G}, k)$, c'est-à-dire qu'on prend le complémentaire de G et qu'on garde la valeur de k. Cette fonction f est bien calculable en temps polynomial ; pour qu'il s'agisse d'une réduction, il reste à montrer que $(G, k) \in$ CLIQUE ssi $f(G, k) = (\bar{G}, k) \in$ ENSEMBLE INDÉPENDANT. Dans un sens, si $(G, k) \in$ CLIQUE alors G possède une clique de taille k, c'est-à-dire k sommets tous reliés : dans $\bar{G}$, ces k sommets sont tous non reliés et forment un ensemble indépendant de taille k, donc $(\bar{G}, k) \in$ ENSEMBLE INDÉPENDANT. L'autre sens est similaire.
Il s'avère que, par le même raisonnement, f est également une réduction de ENSEMBLE INDÉPENDANT à CLIQUE (ce qui est une propriété assez exceptionnelle et ne reflète pas

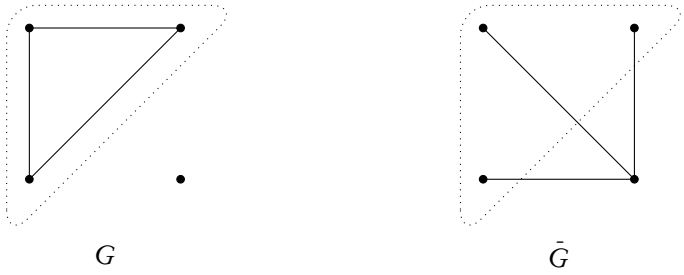


FIGURE 3.1 – Un graphe G avec une clique de taille 3 et son « complémentaire » $\bar{G}$ avec un ensemble indépendant de taille 3.

du tout le cas général des réductions entre problèmes). Donc les deux problèmes sont équivalents.

Il va sans dire que ce premier exemple de réductions est particulièrement simple : les réductions que nous verrons par la suite seront bien plus évoluées.

3.2 Complétude

Les réductions permettent de comparer les problèmes entre eux et donc de parler des problèmes *les plus difficiles* d'une classe de complexité. Nous introduisons pour cela les notions de difficulté et de complétude.

3.2.1 Définition et premières propriétés

3-J Définition (difficulté et complétude)

Soit $\leqslant$ une notion de réduction entre problèmes. Soit A un problème et $\mathscr{C}$ une classe de complexité.

- A est dit *$\mathscr{C}$-difficile* (ou $\mathscr{C}$-dur) pour les réductions $\leqslant$ si pour tout $B \in \mathscr{C}$, on a $B \leqslant A$.
- A est dit *$\mathscr{C}$-complet* pour les réductions $\leqslant$ s'il est $\mathscr{C}$-difficile pour les réductions $\leqslant$ et si $A \in \mathscr{C}$.

Cette notion dépend du type de réductions $\leqslant$; par défaut, si l'on ne précise pas le type de réductions, il s'agira des réductions many-one en temps polynomial $\leqslant^{\mathsf{P}}_{\mathsf{m}}$.

Ainsi, un problème $\mathscr{C}$-difficile est un problème au moins aussi dur que tous ceux de $\mathscr{C}$; on le qualifie en outre de $\mathscr{C}$-complet s'il appartient à $\mathscr{C}$ (il fait donc partie des problèmes les plus durs de $\mathscr{C}$).

3-K Remarques

– La condition est très forte : il faut que *tous* les problèmes de $\mathscr{C}$ se réduisent à A. A priori, rien ne dit qu'il existe des problèmes $\mathscr{C}$-difficiles ni (a fortiori) $\mathscr{C}$-complets.

– Attention, si un langage A est hors de $\mathscr{C}$, cela n'implique pas qu'il est $\mathscr{C}$-difficile ! Nous verrons un contre-exemple à l'exercice 3-AO.

Pour certaines classes usuelles, on ne sait pas si elles contiennent des problèmes complets. Cependant, la magie de cette notion de complétude est l'existence de nombreux problèmes naturels complets pour les classes vues jusqu'à présent, notamment NP. Mais commençons par voir que cette notion est inutile pour la classe P.

3-L Proposition

Tout problème A non trivial (c'est-à-dire $A \neq \emptyset$ et $A \neq \Sigma_A^\star$) est P-difficile pour les réductions many-one en temps polynomial.

Démonstration Soit A non trivial, $x_0 \notin A$ et $x_1 \in A$. Soit $B \in \mathsf{P}$: montrons que $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$. La réduction f de B à A est alors définie comme suit :

$$f(x) = \begin{cases} x_1 \text{ si } x \in B \\ x_0 \text{ sinon.} \end{cases}$$

Puisque $B \in \mathsf{P}$, cette fonction f est calculable en temps polynomial. Par ailleurs, si $x \in B$ alors $f(x) = x_1 \in A$ et si $x \notin B$ alors $f(x) = x_0 \notin A$: ainsi, $x \in B$ ssi $f(x) \in A$, donc f est une réduction de B à A. □

Un peu de recul

Ce résultat vient du fait que la réduction est trop puissante pour la classe P : pour avoir une pertinence, il faudra donc réduire la puissance des réductions. Nous verrons par la suite que nous pouvons considérer pour cela les réduction many-one en espace logarithmique.

Il s'agit d'un phénomène général : plus la classe est petite, plus il faut utiliser des réductions faibles pour comparer les problèmes de la classe.

Les machines universelles vues précédemment permettent de définir un premier problème complet pour les classes vues jusqu'à présent, ce que nous illustrons sur la classe NP.

3-M **Proposition**

Le problème

$$A = \{(\langle N\rangle, x, 1^t) \mid N(x) \text{ accepte en temps } \leqslant t\},$$

où N est une machine non déterministe, x un mot et t un entier (donné ici en unaire), est NP-complet.

Idée de la démonstration $A \in \mathsf{NP}$ car on peut simuler $N(x)$ grâce à la machine universelle. Pour réduire tout problème $B \in \mathsf{NP}$ à A, il suffit de simuler via A la machine non déterministe reconnaissant B.

Démonstration Montrons d'abord que $A \in \mathsf{NP}$. Soit U_{nondet} la machine non déterministe universelle de la proposition 2-AB : voici un algorithme non déterministe pour décider si $(\langle N\rangle, x, 1^t) \in A$.

- Simuler $N(x)$ pendant t étapes en exécutant $U_{\text{nondet}}(\langle N\rangle, x)$.
- Si le chemin simulé a accepté, `accepter`, sinon `rejeter` (le chemin simulé n'a pas terminé ou a rejeté).

Par la proposition 2-AB, cet algorithme fonctionne en temps non déterministe polynomial et possède un chemin acceptant ssi $N(x)$ en possède un en au plus t étapes. Ainsi, $A \in \mathsf{NP}$.

Montrons maintenant que A est NP-difficile. Soit $B \in \mathsf{NP}$: B est reconnu par une machine non déterministe N_B fonctionnant en temps polynomial $p(n)$. La réduction de B à A est alors la fonction f définie par $f(x) = (\langle N_B\rangle, x, 1^{p(|x|)}) : x \in B$ ssi $N_B(x)$ accepte en temps $\leqslant p(|x|)$ ssi $(\langle N_B\rangle, x, 1^{p(|x|)}) \in A$ ssi $f(x) \in A$. Puisque f est clairement calculable en temps polynomial, on a $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$. □

3-N **Remarque** Ce n'est pas un problème NP-complet très intéressant car c'est un problème ad-hoc calqué sur la définition de NP. La NP-complétude devient pertinente lorsqu'elle concerne des problèmes *naturels*.

✎ **3-O** **Exercice**

Montrer que le problème

$$A = \{(\langle M\rangle, x, t) \mid M(x) \text{ accepte en temps } \leqslant t\},$$

où M est une machine déterministe, x un mot et t un entier donné en binaire, est EXP-complet. De même, si M est non déterministe, montrer que le problème correspondant est NEXP-complet.

Un des intérêts des problèmes NP-complets réside dans le résultat suivant.

3-P Proposition

Les affirmations suivantes sont équivalentes :

1. $\mathsf{P} = \mathsf{NP}$;
2. tout problème NP-complet est dans P ;
3. il existe un problème NP-complet dans P.

Démonstration L'implication $1 \Longrightarrow 2$ est évidente, ainsi que $2 \Longrightarrow 3$ car il existe au moins un problème NP-complet (cf. proposition 3-M). Il reste seulement à montrer $3 \Longrightarrow 1$. Soit B un langage de NP, montrons que $B \in \mathsf{P}$. Soit A le langage NP-complet donné par 3 : d'une part, $A \in \mathsf{P}$ par 3 ; d'autre part, $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$ par complétude de A. Puisque P est clos par réduction many-one (proposition 3-C), on en déduit que $B \in \mathsf{P}$. □

3.2.2 Complétude du problème SAT

L'avancée majeure dans la compréhension de la classe NP a été la découverte que de nombreux problèmes naturels réputés difficiles sont NP-complets. Cook [Coo71] et, indépendamment, Levin [Lev73] ont donné de tels problèmes au début des années 1970, complétés ensuite par Karp [Kar72] notamment. Ici notre premier problème sera SAT, la satisfaisabilité de formules booléennes.

3-Q Définition (formules booléennes)

L'ensemble des *formules booléennes sans quantificateur* sur les variables $x_1, \ldots, x_n$ est défini par induction comme suit.

- Une variable x_i est une formule.
- Si $\varphi_1(x_1, \ldots, x_n)$ et $\varphi_2(x_1, \ldots, x_n)$ sont des formules, alors

$$(\varphi_1(x_1, \ldots, x_n) \wedge \varphi_2(x_1, \ldots, x_n)),$$
$$(\varphi_1(x_1, \ldots, x_n) \vee \varphi_2(x_1, \ldots, x_n)) \quad \text{et}$$
$$(\neg\varphi_1(x_1, \ldots, x_n))$$

  sont des formules.

Si $a_1, \ldots, a_n \in \{V, F\}$ (V pour vrai et F pour faux) sont des valeurs pour les variables $x_1, \ldots, x_n$ (on dit aussi une *assignation*), alors la *valeur de vérité* d'une formule $\varphi(x_1, \ldots, x_n)$ en $a_1, \ldots, a_n$, notée $\varphi(a_1, \ldots, a_n)$, est définie comme suit.

- Si $\varphi(x_1, \ldots, x_n) = x_i$ alors $\varphi(a_1, \ldots, a_n) = a_i$.

– Si $\varphi = (\varphi_1 \wedge \varphi_2)$ (conjonction, ET) alors

$$\varphi(a_1,\ldots,a_n) = \begin{cases} V & \text{si } \varphi_1(a_1,\ldots,a_n) = V \textbf{ et } \varphi_2(a_1,\ldots,a_n) = V \\ F & \text{sinon.} \end{cases}$$

– Si $\varphi = (\varphi_1 \vee \varphi_2)$ (disjonction, OU) alors

$$\varphi(a_1,\ldots,a_n) = \begin{cases} V & \text{si } \varphi_1(a_1,\ldots,a_n) = V \textbf{ ou } \varphi_2(a_1,\ldots,a_n) = V \\ F & \text{sinon.} \end{cases}$$

– Si $\varphi = (\neg\varphi_1)$ (négation, NON) alors

$$\varphi(a_1,\ldots,a_n) = \begin{cases} V & \text{si } \varphi_1(a_1,\ldots,a_n) = F \\ F & \text{sinon.} \end{cases}$$

3-R **Remarque** On utilisera la plupart du temps 0 au lieu de F et 1 au lieu de V. On se permettra également les conventions usuelles concernant le parenthésage, ainsi que des abréviations utiles comme

– 1 pour $x_1 \vee \neg x_1$ et 0 pour $x_1 \wedge \neg x_1$;
– $\varphi \to \psi$ pour $\neg\varphi \vee \psi$;
– $\varphi \leftrightarrow \psi$ pour $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

Enfin, un uple de variables $(x_1,\ldots,x_n)$ sera souvent désigné par x si le nombre d'éléments $|x|$ du uple n'a pas besoin d'être précisé.

3-S **Exemple** $\varphi(x_1,x_2,x_3) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge (x_2 \vee x_3))$ est une formule booléenne sans quantificateur qui satisfait $\varphi(0,0,0) = 0$ et $\varphi(0,0,1) = 1$.

3-T **Remarques**

– On peut coder très simplement une formule booléenne en binaire en codant les symboles à la suite, séparés par un délimiteur. Puisqu'il faut donner le numéro des variables, une formule sur n variables s'écrivant avec k symboles aura un codage de taille $O(k \log n)$. La taille du codage d'une formule φ sera noté $|\varphi|$.
– Une formule se représente également sous la forme d'un arbre où les feuilles sont les variables (apparaissant éventuellement plusieurs fois) et les nœuds les opérations $\vee$, $\wedge$ et $\neg$.
– Étant données une formule $\varphi(x_1,\ldots,x_n)$ et une assignation $(a_1,\ldots,a_n)$ pour les variables, on peut décider en temps polynomial si $\varphi(a_1,\ldots,a_n) = 1$. Il suffit pour cela

d'évaluer l'arbre correspondant à la formule depuis les feuilles jusqu'à la racine.

Nous pouvons maintenant introduire le problème SAT.

3-U Définition (SAT)

- Une formule $\varphi(x_1,\dots,x_n)$ est dite *satisfaisable* s'il existe une assignation $(a_1,\dots,a_n)\in\{0,1\}^n$ des variables telle que $\varphi(a_1,\dots,a_n)=1$.
- Le problème SAT est le problème suivant :
  - *entrée* : une formule booléenne sans quantificateur $\varphi(x_1,\dots,x_n)$;
  - *question* : φ est-elle satisfaisable ?

3-V Théorème (Cook 1971, Levin 1973)

Le problème SAT est NP-complet.

Plus précisément, le fonctionnement en temps polynomial d'une machine non déterministe N sur une entrée x est décrit par une formule φ calculable en temps polynomial telle que le nombre d'assignations satisfaisant φ est égal au nombre de chemins acceptants de $N(x)$.

Idée de la démonstration $\texttt{SAT}\in\texttt{NP}$ car il suffit de deviner une assignation des variables et vérifier en temps polynomial qu'elle satisfait la formule.

La complétude vient du fait qu'on peut décrire par une formule de taille polynomiale le diagramme espace-temps d'une exécution d'une machine non déterministe polynomiale car celui-ci répond à des règles *locales*. En d'autres termes, on décrit par une formule $\varphi(y)$ le fonctionnement de la machine le long du chemin (découlant du choix des transitions) décrit par y. Pour savoir s'il existe un chemin acceptant dans le calcul de la machine, il suffit alors de savoir s'il existe une affectation des variables y de la formule pour laquelle l'état final du diagramme décrit est acceptant, ce qui est un problème de type SAT.

Démonstration Voici un algorithme non déterministe fonctionnant en temps polynomial pour SAT sur l'entrée $\varphi(x_1,\dots,x_n)$:

- `deviner` $(a_1,\dots,a_n)\in\{0,1\}^n$;
- vérifier que $\varphi(a_1,\dots,a_n)=1$.

Comme on l'a vu, la vérification prend un temps polynomial : ainsi, $\texttt{SAT}\in\texttt{NP}$.

Il reste donc à montrer que SAT est NP-difficile. Soit $B \in \mathsf{NP}$: à toute instance x du problème B, il s'agit d'associer une formule φ_x telle que $x \in B$ ssi φ_x est satisfaisable. Les variables de φ_x désigneront en quelque sorte le chemin de calcul à suivre. Soit N une machine non déterministe polynomiale à k rubans pour B (ensemble d'états Q, alphabet de travail Γ) : nous allons « simuler » le fonctionnement de $N(x)$ le long d'un chemin arbitraire par φ_x. Pour cela, nous allons considérer le *diagramme espace-temps* de $N(x)$, c'est-à-dire le diagramme représentant l'évolution du contenu des rubans au cours du temps. Par commodité, sur la figure 3.2 nous représentons un seul ruban de travail. La case sur laquelle se situe la tête de lecture est indiquée par le symbole $\star$.

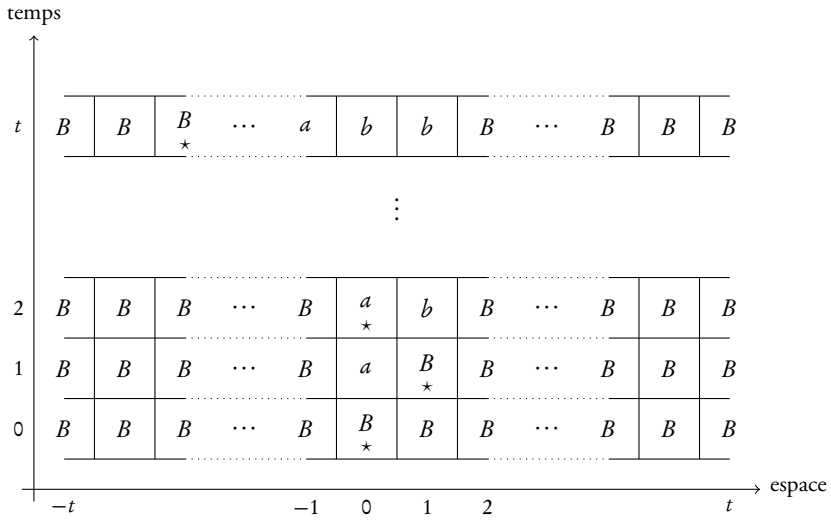


FIGURE 3.2 – Diagramme espace-temps pour un ruban de travail.

Lors des t étapes de calcul de $N(x)$ (t est un polynôme en $|x|$, donc constructible en temps), les cases visitées du ruban se situent entre $-t$ et t. Nous considérons donc seulement les $(2t+1)(t+1)$ cases du diagramme représentées sur la figure.

Notre formule φ_x va affirmer que le calcul commence dans la configuration initiale, que le diagramme espace-temps de la machine est cohérent avec la relation de transition et qu'on termine dans un état acceptant. Elle sera composée de quatre parties :

- `cohérence` signifiant que deux valeurs ne sont pas assignées à la même case, deux positions à la même tête, deux états à la même étape ;
- $\texttt{début}_x$ signifiant que la configuration initiale est la bonne : on est dans l'état initial, les têtes sont en position initiale, x est écrit sur le ruban de lecture et les autres rubans sont vides ;
- pour chaque étape j, $\texttt{transition}_j$ signifiant que la j-ème transition est valide : déplacements des têtes et symboles écrits cohérents avec la relation de transition, pas de changement pour les cellules non concernées par les têtes ;

– `accepte` signifiant qu'on arrive dans un état acceptant à un temps $\leqslant t$.

Voici maintenant les variables qui seront utilisées dans notre formule.

– Pour chacune des étapes, chaque case des rubans va être décrite par plusieurs variables : pour chaque symbole $\gamma \in \Gamma$ et pour chaque ruban r, on utilise une variable $c^r_{\gamma,i,j}$ qui vaut *vrai* ssi la i-ème case du ruban r contient le symbole γ au temps j (ici, $1 \leqslant r \leqslant k$, $-t \leqslant i \leqslant t$ et $0 \leqslant j \leqslant t$).
– La position de la tête au temps j sur chaque ruban r est décrite par $2t+1$ variables : la variable $p^r_{i,j}$ (où $1 \leqslant r \leqslant k$, $-t \leqslant i \leqslant t$ et $0 \leqslant j \leqslant t$) vaut *vrai* ssi la tête du ruban r est à la position i au temps j.
– Pour chaque état q et à chaque instant j, la variable $e_{q,j}$ vaut *vrai* ssi l'état des têtes est q à l'instant j (ici, $q \in Q$ et $0 \leqslant j \leqslant t$).

En tout, le nombre de variables est donc $(t+1)(2t+1)|\Gamma|k+(t+1)(2t+1)k+(t+1)|Q|$, qui est polynomial en la taille de x puisque t est polynomial et Γ, k et Q sont constants (dépendant seulement de la machine N et non de x).
Dans un premier temps, il faut s'assurer qu'à chaque étape, chaque case contient exactement une valeur, chaque tête est exactement à un endroit et on est dans exactement un état. On peut exprimer cela par la conjonction des formules suivantes (ici, $r \in \{1,\dots,k\}$ désigne un ruban, $i \in \{-t,\dots,t\}$ une position, $j \in \{0,\dots,t\}$ une étape et $\gamma \in \Gamma$ un symbole) :

– à tout moment chaque case contient exactement un symbole :

$$\bigwedge_{r,i,j} \bigvee_{\gamma} (c^r_{\gamma,i,j} \wedge \bigwedge_{\gamma' \neq \gamma} \neg c^r_{\gamma',i,j});$$

– à tout moment chaque tête a exactement une position :

$$\bigwedge_{r,j} \bigvee_{i} (p^r_{i,j} \wedge \bigwedge_{i' \neq i} \neg p^r_{i',j});$$

– et à tout moment on est dans exactement un état :

$$\bigwedge_{j} \bigvee_{q} (e_{q,j} \wedge \bigwedge_{q' \neq q} \neg e_{q',j}).$$

La conjonction de ces trois formules est de taille polynomiale et donne la formule `cohérence` mentionnée ci-dessus.
Pour exprimer que la configuration initiale est correcte, il s'agit de tester si l'état au temps 0 est q_0, si tous les rubans sont vides à part le ruban de lecture qui contient $x = x_1 \dots x_n$ à partir de la case 1, et si toutes les têtes sont en position 1. On peut écrire cela sous la forme d'une conjonction des quatre formules suivantes :

– $e_{q_0,0}$ (l'état de départ est l'état initial q_0) ;
– $\bigwedge_{r>1,i} c^r_{B,i,0}$ (au départ toutes les cases des rubans $r > 1$ contiennent le symbole B) ;

– le ruban de lecture contient x entre les positions 1 et n et B ailleurs :

$$\Big(\bigwedge_{i\leqslant 0 \vee i>n} c^1_{B,i,0}\Big) \wedge \Big(\bigwedge_{1\leqslant i\leqslant n} c^1_{x_i,i,0}\Big);$$

– la tête de chaque ruban r est en position 1 au temps 0 :

$$\bigwedge_r (p^r_{1,0}).$$

La conjonction de ces quatre formules est de taille polynomiale et donne la formule $\texttt{début}_x$ mentionnée précédemment.

Pour alléger les notations qui suivent et par abus de notation, on notera pour tout $\gamma \in \Gamma^{k-1}$, $\delta(q_a,\gamma) = \{(q_a,(B,\ldots,B),(S,\ldots,S))\}$ et $\delta(q_r,\gamma) = \{(q_r,(B,\ldots,B),(S,\ldots,S))\}$ (alors qu'en réalité, par définition $\delta(q_a,\gamma) = \delta(q_r,\gamma) = \emptyset$ puisque le calcul s'arrête). Ce n'est qu'une notation pour la suite, mais intuitivement cela revient à considérer que le calcul ne s'arrête pas à partir du moment où on atteint un état terminal, et qu'alors la configuration ne change plus.

Dans ce cas, la formule `accepte` est simple à écrire, il s'agit d'exprimer qu'au temps t on est dans l'état acceptant q_a : cela s'écrit $e_{q_a,t}$.

Il reste le cœur de la simulation : spécifier que le comportement de la machine correspond à la relation de transition. Pour la j-ème transition (où $1 \leqslant j \leqslant t$), il s'agit de s'assurer que :

– pour chaque ruban les cases ne contenant pas la tête n'ont pas changé de contenu :

$$\psi_{\text{contenu}} \equiv \bigwedge_{r,i} (\neg p^r_{i,j-1} \to \bigwedge_{\gamma} (c^r_{\gamma,i,j} \leftrightarrow c^r_{\gamma,i,j-1}));$$

– et la valeur de la case contenant la tête, la position de la tête et l'état ont évolué selon une transition valide : pour tout état q, tout uple de positions i et tout uple de symboles γ, si à l'étape $j-1$, l'état est q, les positions des têtes sont i et les symboles lus sont γ, alors l'évolution doit se dérouler selon une transition $(q',\gamma',d') \in \delta(q,\gamma)$ (nouvel état, nouveau contenu des cases et déplacement des têtes). En symboles :

$$\psi_{\text{trans}} \equiv \bigwedge_{q,i,\gamma} \Bigg(\big(e_{q,j-1} \wedge \bigwedge_r (p^r_{i_r,j-1} \wedge c^r_{\gamma_r,i_r,j-1})\big) \to \bigvee_{(q',\gamma',d')\in\delta(q,\gamma)} \big(e_{q',j} \wedge \bigwedge_r (c^r_{\gamma'_r,i_r,j} \wedge p^r_{i_r+d'_r,j})\big) \Bigg).$$

Ici, i désigne $(i_1,\ldots,i_k)$; γ désigne $(\gamma_1,\ldots,\gamma_k)$ mais, utilisé comme argument de δ, seules les $k-1$ premières composantes sont concernées (puisque le k-ème ruban est en écriture seule) ; de même, γ' désigne $(\gamma'_1,\ldots,\gamma'_k)$ mais, utilisé comme « image » de δ, seules les $k-1$ dernières composantes sont concernées (puisque le premier ruban est en lecture seule) ; d' désigne $(d'_1,\ldots,d'_k)$; enfin, $i_r + d'_r$ signifie $i_r - 1$ si $d'_r = G$, i_r si $d'_r = S$ et $i_r + 1$ si $d'_r = D$.

La formule $\texttt{transition}_j$ vaut alors $(\psi_{\text{contenu}} \wedge \psi_{\text{trans}})$. On remarquera une nouvelle fois que cette formule est de taille polynomiale.
On peut enfin écrire la formule φ_x :

$$\varphi_x = \texttt{cohérence} \wedge \texttt{début}_x \wedge \bigwedge_{1 \leqslant j \leqslant t} \texttt{transition}_j \wedge \texttt{accepte}.$$

La formule φ_x est de taille polynomiale et est satisfaisable si et seulement s'il existe un chemin acceptant dans le calcul de $N(x)$. Par ailleurs, le code de N étant fixé par le choix de la machine décidant le langage B, à partir de l'instance x il est aisé de construire en temps polynomial le code de φ_x puisque nous venons d'écrire cette formule explicitement. Ainsi la fonction $f : x \mapsto \varphi_x$ est une réduction many-one polynomiale de B à SAT.
Enfin, par construction (notamment grâce à la formule cohérence), il y a une bijection entre les assignations satisfaisant φ_x et les chemins acceptants de $N(x)$, donc le nombre de celles-ci est égal au nombre de chemins acceptants de $N(x)$. □

Ce problème est notre premier problème NP-complet naturel et servira de base pour montrer la complétude d'une multitude d'autres problèmes. En effet, pour montrer la NP-difficulté d'un problème A, il suffit maintenant de donner une réduction de SAT à A comme le montre le lemme suivant.

3-W Lemme

Soit C un problème NP-complet et $A \in \mathsf{NP}$. Si $C \leqslant_{\mathsf{m}}^{\mathsf{p}} A$ alors A est NP-complet.

Démonstration Soit $B \in \mathsf{NP}$. Par complétude de C, $B \leqslant_{\mathsf{m}}^{\mathsf{p}} C$. Mais par hypothèse, $C \leqslant_{\mathsf{m}}^{\mathsf{p}} A$: la transitivité de $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ (lemme 3-G) permet de conclure que $B \leqslant_{\mathsf{m}}^{\mathsf{p}} A$ et donc que A est NP-difficile. □

3.2.3 Autres problèmes NP-complets

Notre prochain problème NP-complet est une variante importante de SAT appelée 3SAT : il s'agit de décider la satisfaisabilité de formules données en 3-CNF, forme que nous définissons maintenant.

3-X Définition (forme normale conjonctive)

- Un *littéral* est soit une variable (x_i) soit la négation d'une variable ($\neg x_i$).
- Une *clause* est une disjonction de littéraux : $C = \bigvee_i l_i$ où l_i est un littéral.
- Une formule φ est dite sous *forme normale conjonctive* (CNF) si elle est la conjonction de clauses (c'est-à-dire la conjonction de disjonctions de littéraux) :

$$\varphi = \bigwedge_i C_i \qquad \text{où} \qquad C_i = \bigvee_j l_{i,j} \text{ est une clause.}$$

– Enfin, φ est en 3-CNF si elle est en CNF avec au plus trois littéraux par clause.

3-Y **Exemple** $\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$ est en 3-CNF.

3SAT est alors le problème suivant :

– *entrée* : une formule $\varphi(x_1, \ldots, x_n)$ en 3-CNF ;
– *question* : φ est-elle satisfaisable ?

3-Z **Proposition**

Le problème 3SAT est NP-complet.

De plus, la réduction de SAT à 3SAT préserve le nombre d'assignations satisfaisant la formule.

Idée de la démonstration On voit une instance $\varphi(x)$ de SAT sous la forme d'un arbre : tester sa satisfaisabilité revient alors à deviner les valeurs de x et celles des nœuds de l'arbre et tester la cohérence de ces choix. Ce test est local à chaque nœud de l'arbre et nécessite seulement des clauses contenant trois littéraux car le degré d'un sommet de l'arbre est au plus 3 (un père et au plus deux fils).

Démonstration Que 3SAT soit dans NP est une évidence puisqu'il suffit de tester (en temps polynomial) si l'entrée est en 3-CNF et d'appliquer l'algorithme non déterministe pour SAT.

Afin de montrer sa NP-difficulté, nous donnons une réduction de SAT à 3SAT. Soit $\varphi(x)$ une instance de SAT : nous allons construire une instance $\psi(x, y)$ de 3SAT de sorte que φ soit satisfaisable ssi ψ l'est.

Pour cela, nous considérons l'arbre représentant la formule φ, dont les feuilles (entrées) sont les variables x_i (apparaissant éventuellement plusieurs fois) et les nœuds les opérations $\neg$, $\vee$ et $\wedge$: chaque nœud a donc au plus deux fils. On peut voir l'exemple d'un tel arbre à la figure 3.3.

L'objectif est de décrire la valeur de tous les nœuds de l'arbre afin de connaître la valeur de la racine. On associe donc une nouvelle variable y_s à chaque nœud s de l'arbre. Les clauses de notre formule $\psi(x, y)$ sont les suivantes :

– si s est une feuille correspondant à la variable x_i, il s'agit d'exprimer que $y_s = x_i$, ce qui donne les deux clauses $C_s = (x_i \vee \neg y_s) \wedge (\neg x_i \vee y_s)$;
– si s est une porte $\neg$ ayant pour fils la porte s', il s'agit d'exprimer que $y_s = \neg y_{s'}$, ce qui donne les deux clauses $C_s = (y_s \vee y_{s'}) \wedge (\neg y_s \vee \neg y_{s'})$;
– si s est une porte $\vee$ ayant pour fils les portes s_1 et s_2, il s'agit d'exprimer que $y_s = y_{s_1} \vee y_{s_2}$, ce qui donne les trois clauses $C_s = (\neg y_s \vee y_{s_1} \vee y_{s_2}) \wedge (y_s \vee \neg y_{s_1}) \wedge (y_s \vee \neg y_{s_2})$;

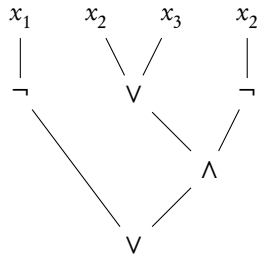


FIGURE 3.3 – Arbre correspondant à la formule $(\neg x_1) \vee ((x_2 \vee x_3) \wedge (\neg x_2))$.

– enfin, si s est une porte $\wedge$ ayant pour fils les portes s_1 et s_2, il s'agit d'exprimer que $y_s = y_{s_1} \wedge y_{s_2}$, ce qui donne les trois clauses $C_s = (y_s \vee \neg y_{s_1} \vee \neg y_{s_2}) \wedge (\neg y_s \vee y_{s_1}) \wedge (\neg y_s \vee y_{s_2})$.

De cette façon, nous avons exprimé le fait que toutes les variables y_s prennent la valeur de la porte s correspondante. Il ne reste plus qu'à vérifier que la porte de sortie, appelons-la t, a la valeur 1, ce qui donne au final la formule

$$\psi(x,y) = y_t \wedge \bigwedge_s C_s,$$

qui est bien en 3-CNF. Ainsi, $\varphi(x)$ est satisfaisable ssi il existe une assignation des variables x telle que la porte t vaille 1, ssi pour cette assignation des variables x, il existe une (unique) assignation des variables y telle que $\psi(x,y)$ vaille 1. De plus, la fonction $\varphi \mapsto \psi$ est bien sûr calculable en temps polynomial puisqu'il suffit d'écrire les clauses que nous avons spécifiées ci-dessus. Au final, nous venons de décrire une réduction de `SAT` à `3SAT` qui conserve le nombre de solutions. □

3-AA **Remarque** Attention, le problème de la satisfaisabilité des formules en 2-CNF (deux littéraux par clause) est résoluble en temps polynomial, bien que l'algorithme ne soit pas trivial.

De même pour les formules en forme normale disjonctive (DNF), où les clauses sont des conjonctions de littéraux et la formule est une disjonction de clauses (cf. exercice ci-dessous).

✎ **3-AB** **Exercice**

Montrer que le problème de décider si une formule en DNF (forme normale disjonctive où les clauses sont des conjonctions de littéraux et la formule est une disjonction de clauses) est dans P.

À partir de cette variante de `SAT`, nous pouvons montrer la complétude de plusieurs problèmes dans des domaines variés : graphes, nombres, etc. Nous verrons seulement un petit

nombre d'exemples mais la liste des problèmes NP-complets est grande. De nombreuses variantes de SAT sont NP-complètes et nous commençons par le problème 1Lit3SAT suivant :

- *entrée* : une formule $\varphi(x_1,\dots,x_n)$ en 3-CNF ;
- *question* : existe-t-il une affectation des variables rendant exactement un littéral vrai par clause ?

Il s'agit bien sûr d'un sous-ensemble de 3SAT dont nous montrons qu'il est également NP-complet.

3-AC Proposition

Le problème 1Lit3SAT est NP-complet.

Idée de la démonstration L'appartenance à NP est facile. La NP-difficulté vient d'une réduction de 3SAT où l'on sépare les littéraux de chaque clause en ajoutant des variables pour garantir une solution avec exactement un littéral vrai par clause.

Démonstration Le problème est dans NP grâce à la machine non déterministe suivante, fonctionnant en temps polynomial sur une formule $\varphi(x_1,\dots,x_n)$ en 3-CNF :

- `deviner` une affectation des variables ;
- vérifier que chaque clause contient exactement un littéral vrai.

Nous donnons maintenant une réduction du problème 3SAT au problème 1Lit3SAT. Soit $\varphi(x_1,\dots,x_n)$ une instance de 3SAT (c'est-à-dire une formule en 3-CNF) ayant m clauses.

Si C est une clause $(x \vee y \vee z)$ de φ, où x,y,z sont des littéraux (variables ou négation de variables), alors on remplace C par les quatre clauses $\psi_C = (\neg x \vee a \vee b) \wedge (\neg y \vee c \vee d) \wedge (\neg z \vee e \vee f) \wedge (a \vee c \vee e)$, où $a,\dots,f$ sont des nouvelles variables.

Si C est satisfaite par une affectation de x,y,z, alors il existe une affectation avec les mêmes valeurs de x,y,z rendant exactement un littéral vrai par clause de ψ_C. Pour montrer cela, raisonnons sur le nombre de littéraux vrais dans C :

- si un seul littéral de C est vrai, par exemple x (les autres cas sont symétriques), on fixe $a=1$ et $b,c,d,e,f=0$;
- si deux littéraux de C sont vrais, par exemple x et y (les autres cas sont symétriques), alors on fixe $a,d=1$ et $b,c,e,f=0$;
- si les trois littéraux de C sont vrais, on fixe $a,d,f=1$ et $b,c,e=0$.

Réciproquement, si ψ_C est satisfaite par une affectation rendant exactement un littéral vrai par clause, alors la même affectation de x,y,z satisfait C. En effet, exactement l'une des variables a,c,e doit être vraie, par exemple a (les autres cas sont symétriques), donc x doit être vrai sinon la première clause aurait deux littéraux vrais.

Le remplacement de chaque clause de φ de cette façon, en utilisant pour chacune de nouvelles variables $a, \dots, f$, donne une formule $\psi = \bigwedge_C \psi_C$ contenant $4m$ clauses et $n+6m$ variables. La formule ψ est évidemment calculable en temps polynomial à partir de φ. Par ce qui précède, puisque les valeurs des littéraux x, y, z sont préservées (ce qui est important si plusieurs clauses utilisent les mêmes littéraux), si φ est satisfaisable alors il existe une affectation rendant exactement un littéral vrai par clause de ψ ; et réciproquement, si ψ possède une affectation rendant exactement un littéral vrai par clause, alors la même affectation des variables de φ satisfait φ. □

Nous proposons en exercice de montrer qu'une autre variante de `SAT` est NP-complète.

✎ **3-AD** **Exercice**

Soit `Half-3SAT` le problème suivant :

- *entrée :* formule $\varphi(x_1, \dots, x_n)$ en 3-CNF ;
- *question :* existe-t-il une affectation des variables $(x_1, \dots, x_n)$ satisfaisant exactement la moitié des clauses ?

Montrer que `Half-3SAT` est NP-complet.

Indication : si φ est une instance de `3SAT` à m clauses, doubler le nombre de clauses de φ en ajoutant m fois une même clause comportant de nouvelles variables. Remarquer en outre que si φ est telle qu'il existe une affectation falsifiant toutes les clauses, alors φ est satisfaisable.

Les graphes sont aussi de grands pourvoyeurs de problèmes NP-complets. Par exemple, le problème `ENSEMBLE INDÉPENDANT` défini précédemment (exemple 3-I), pour lequel il s'agit de décider s'il existe k sommets non reliés deux à deux dans un graphe non orienté.

3-AE **Proposition**

Le problème `ENSEMBLE INDÉPENDANT` est NP-complet.

Idée de la démonstration Le problème est dans NP car vérifier que k sommets sont indépendants se fait en temps polynomial. Pour la NP-difficulté, on réduit `3SAT` : la réduction transforme une formule φ en 3-CNF à i clauses en un graphe contenant i triangles (un pour chaque clause). Chaque sommet d'un triangle correspond à un littéral de la clause. Entre les triangles, on relie ensuite x et $\neg x$. Si φ est satisfaisable, alors les littéraux valant 1 dans une solution forment un ensemble indépendant du graphe, et réciproquement.

Démonstration Voici une machine non déterministe pour `ENSEMBLE INDÉPENDANT`, sur l'entrée (G, k) :

– `deviner` un sous-ensemble de k sommets $x_1, \ldots, x_k$;
– vérifier que $x_1, \ldots, x_k$ sont distincts et sont tous deux à deux non reliés.

Il est clair que cette machine fonctionne en temps polynomial et qu'il existe un chemin acceptant si et seulement s'il existe un ensemble indépendant de taille k. Donc `ENSEMBLE INDÉPENDANT` est dans NP.

Pour montrer la NP-difficulté, nous réduisons `3SAT` à `ENSEMBLE INDÉPENDANT`. Soit $\varphi(x_1, \ldots, x_n)$ une formule en 3-CNF (une instance de `3SAT`) : le but est d'associer à φ en temps polynomial un graphe G et un entier k tel que φ est satisfaisable ssi G a un ensemble indépendant de taille k. La formule φ est constituée de m clauses contenant chacune trois littéraux (si une clause contient moins de trois littéraux, on ajoute l'un des littéraux pour en obtenir exactement trois, par exemple $(x_1 \vee \neg x_2) \mapsto (x_1 \vee x_1 \vee \neg x_2)$). On pose alors $k = m$ (le nombre de clauses). Chaque clause donne lieu à un triangle dans G reliant trois nouveaux sommets correspondant aux trois littéraux apparaissant dans la clause (voir la figure 3.4).

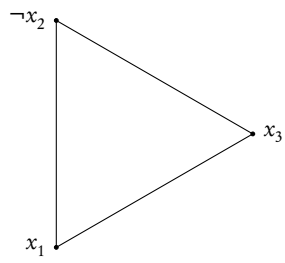


FIGURE 3.4 – Une clause est représentée par un triangle dont les sommets sont les littéraux. Ici, la clause est $(x_1 \vee \neg x_2 \vee x_3)$.

Ainsi, les sommets du graphe G seront tous les littéraux apparaissant dans φ avec leur multiplicité, et il y a des arêtes entre tous les littéraux correspondant à une même clause, formant donc $k = m$ triangles dans G. Mais ce ne sont pas les seules arêtes du graphe : on relie également chaque littéral à sa négation, par exemple on relie toutes les occurences de x_1 à toutes les occurences de $\neg x_1$. On obtient ainsi le graphe G associé à la formule φ, comme illustré à la figure 3.5.

Cette construction de (G, k) à partir de φ est clairement calculable en temps polynomial. Il reste à montrer que φ est satisfaisable ssi G a un ensemble indépendant de taille $k = m$.

Si φ est satisfaisable par une certaine affectation des variables, soit a_i un littéral égal à 1 dans la clause i selon cette affectation (a_i est de la forme x_j ou $\neg x_j$). Par construction, un des sommets de G correspond à ce littéral. Puisque pour $i \neq j$, a_i et a_j sont dans des clauses différentes, ils ne sont pas dans le même triangle de G, et par ailleurs ils ne sont pas la négation l'un de l'autre car il existe une affectation qui les rend tous

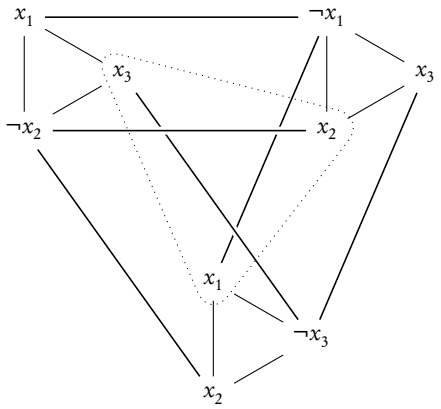


FIGURE 3.5 – Graphe correspondant à la formule $\varphi(x_1,x_2,x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ et ensemble indépendant de taille 3 (en pointillés) montrant que $x_1 = x_2 = x_3 = 1$ satisfait φ.

deux vrais. Ainsi, il n'y a pas d'arête entre eux dans G et $a_1,\ldots,a_m$ est un ensemble indépendant de taille $k = m$.

Réciproquement, supposons que G ait un ensemble indépendant $\{a_1,\ldots,a_m\}$ de taille $k = m$. Si $i \neq j$, a_i et a_j ne peuvent pas être la négation l'un de l'autre sinon ils seraient reliés dans G. Ainsi, il existe une affectation des variables de φ telle que tous les a_i ont valeur 1. De même, il ne peuvent pas correspondre à deux littéraux de la même clause car ceux-ci sont reliés dans G. Cette affectation rend donc au moins un littéral vrai par clause, et φ est satisfaisable. □

3-AF **Remarque** Si k est fixé (et non donné dans l'entrée), le problème se résout en temps polynomial puisqu'il suffit de tester tous les sous-ensembles de k sommets (il y en a moins de n^k).

Par les réductions entre `ENSEMBLE INDÉPENDANT` et `CLIQUE` vues à l'exemple 3-I, nous avons le corollaire suivant.

3-AG **Corollaire**

Le problème `CLIQUE` est NP-complet.

Nous allons maintenant montrer que le problème `SOMME PARTIELLE` vu au chapitre précédent (exemple 2-AM) est NP-complet.

3-AH Proposition

Le problème SOMME PARTIELLE est NP-complet.

Idée de la démonstration Il suffit de deviner un sous-ensemble sommant à t pour résoudre ce problème dans NP. Pour la NP-difficulté, on réduit 3SAT en imposant que chaque clause soit satisfaite grâce à des entiers correspondant aux littéraux et aux clauses.

Démonstration La machine non déterministe polynomiale suivante montre que le problème SOMME PARTIELLE est dans NP :

- deviner un sous-ensemble $S \subseteq \{1, \dots, m\}$;
- vérifier que $\sum_{i \in S} a_i = t$.

Nous réduisons maintenant 3SAT au problème SOMME PARTIELLE pour montrer la NP-difficulté de ce dernier. Soit $\varphi(x_1, \dots, x_n)$ une instance de 3SAT (c'est-à-dire une formule en 3-CNF) ayant m clauses $C_0, \dots, C_{m-1}$.

À chacun des deux littéraux $l \in \{x_i, \neg x_i\}$ sur la variable x_i, on associe l'entier

$$a_l = 6^{i+m-1} + \sum_{j \mid l \in C_j} 6^j.$$

En d'autres termes, en base 6 le chiffre de j-ème position est 1 si l apparaît dans la clause C_j, et le chiffre de position $i + m - 1$ est fixé à 1.

Par ailleurs, à chaque clause C_i ($0 \leqslant i < m$), on associe les deux entiers $b_i = b'_i = 6^i$. En d'autres termes, en base 6 le chiffre de i-ème position est fixé à 1.

Enfin, l'entier cible est défini par $t = 3\sum_{i=0}^{m-1} 6^i + \sum_{i=m}^{m+n-1} 6^i$, c'est-à-dire le nombre dont l'écriture en base 6 est

$$\underbrace{1 \dots 1}_{n \text{ fois}} \; \underbrace{3 \dots 3}_{m \text{ fois}}.$$

Un exemple est donné à la figure 3.6.

Comme sur l'exemple, on visualisera les nombres a, b, b' et t en base 6, les n premières colonnes représentant les variables $x_n, \dots, x_1$ et les m suivantes représentant les clauses $C_{m-1}, \dots, C_0$. On notera que l'ensemble des nombres a ne peut pas apporter plus de trois chiffres 1 par colonne C_i car chaque clause contient trois littéraux. Ainsi, chaque colonne a au plus cinq chiffres 1 et il n'y a donc jamais de retenue qui peut se propager en base 6.

Cette instance de SOMME PARTIELLE est bien sûr calculable en temps polynomial à partir de φ. Supposons que φ soit satisfaisable par l'affectation $(\alpha_1, \dots, \alpha_n)$ de ses variables. Alors on choisit comme solution de SOMME PARTIELLE l'ensemble A des entiers a_l tels que l est vrai (c'est-à-dire $l = x_i$ si $\alpha_i = 1$ ou $l = \neg x_i$ si $\alpha_i = 0$). Dans ce cas, $\sum_{a_l \in A} a_l$ s'écrit en base 6 avec n chiffres 1 en tête (sur les colonnes des variables) et les m chiffres suivants sont compris entre 1 et 3. En effet, puisque chaque clause est

Instance de 3SAT

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

Représentation des entiers en base 6 :

	x_4	x_3	x_2	x_1	C_2	C_1	C_0
a_{x_1}				1	1	0	1
$a_{\neg x_1}$				1	0	1	0
a_{x_2}			1	0	0	0	0
$a_{\neg x_2}$			1	0	1	1	1
a_{x_3}		1	0	0	0	0	1
$a_{\neg x_3}$		1	0	0	1	0	0
a_{x_4}	1	0	0	0	0	1	0
$a_{\neg x_4}$	1	0	0	0	0	0	0
b_1							1
b'_1							1
b_2						1	0
b'_2						1	0
b_3					1	0	0
b'_3					1	0	0
t	1	1	1	1	3	3	3

FIGURE 3.6 – Illustration de la réduction de 3SAT à SOMME PARTIELLE.

satisfaite, il y a au moins un 1 par colonne de clause (et au plus trois par la remarque ci-dessus). Il suffit alors de compléter la somme par les variables b_i et b'_i nécessaires pour amener chaque colonne de clause à la valeur 3. Ainsi, cette instance de SOMME PARTIELLE a une solution.

Réciproquement, si cette instance de SOMME PARTIELLE a une solution, on remarque d'abord que pour chaque variable x_i, soit a_{x_i} soit $a_{\neg x_i}$ a été choisi pour obtenir 1 dans la colonne de x_i. Si a_{x_i} est choisi, on affectera $x_i = 1$; si c'est $a_{\neg x_i}$ qui est choisi, on affectera $x_i = 0$. Puisque chaque colonne de clause vaut 3 et que parmi les nombres b, b', seuls deux peuvent y contribuer, cela signifie qu'il y a au moins un a_l choisi apportant un 1 dans cette colonne. La clause est donc satisfaite par le littéral l. Au total, chaque clause est satisfaite par l'affectation des variables, donc φ est satisfaisable.

La fonction qui à φ associe cette instance de SOMME PARTIELLE est donc une réduction de 3SAT à SOMME PARTIELLE, ce qui montre la NP-difficulté de SOMME PARTIELLE.

□

Un peu de recul

Nous avons vu quelques problèmes NP-complets issus de domaines différents (logique, graphes, nombres) mais il existe des centaines d'autres problèmes NP-complets importants en pratique. C'est ce qui fait de la NP-complétude une notion centrale en complexité et en algorithmique.

On ne connaît d'algorithme polynomial pour aucun d'entre eux. Rappelons que résoudre l'un d'entre eux en temps polynomial revient à tous les résoudre en temps polynomial. Un résultat de NP-complétude est donc considéré comme une « preuve » que le problème est difficile (mais ce n'est pas une vraie preuve tant qu'on n'a pas montré $\mathsf{P} \neq \mathsf{NP}$!).

On pourra trouver de nombreux autres exemples de problèmes NP-complets dans le livre de Garey et Johnson [GJ79] par exemple.

3.2.4 Complémentaire

Nous avons défini la classe coNP à la section 2.2.8. Cette classe aussi admet des problèmes complets. En réalité, tout le travail fait ici se transpose aisément. Nous avons en effet le résultat suivant.

3-AI Lemme

Si A est NP-complet, alors ${}^cA$ est coNP-complet.

Démonstration Comme $A \in \mathsf{NP}$, par définition ${}^cA$ est dans coNP. Il reste à montrer qu'il est coNP-difficile. Soit $B \in \mathsf{coNP}$: par définition, ${}^cB \in \mathsf{NP}$, donc ${}^cB$ se réduit à A via une réduction f calculable en temps polynomial. En d'autres termes, $x \in {}^cB$ ssi $f(x) \in A$. Cela implique que $x \in B$ ssi $f(x) \in {}^cA$, c'est-à-dire que B se réduit à ${}^cA$. Donc ${}^cA$ est coNP-complet. □

La remarque 2-AZ ne remet bien sûr pas en cause ce résultat. On en déduit que les problèmes coSAT, coCLIQUE, etc., sont coNP-complets.

✎ **3-AJ Exercice**

Montrer que le problème TAUTOLOGIE suivant est coNP-complet :

- *entrée* : une formule booléenne $\varphi(x_1, \ldots, x_n)$;
- *question* : φ est-elle une tautologie, c'est-à-dire $\varphi(a_1, \ldots, a_n)$ est-elle vraie pour toute affectation $(a_1, \ldots, a_n)$?

Indication : on pourra faire une réduction de coSAT.

3.2.5 Théorème de Ladner

Dans NP, nous avons donc d'un côté des problèmes NP-complets qui sont difficiles si P ≠ NP, et de l'autre des problèmes P qui sont faciles. Si P ≠ NP, y a-t-il quelque chose entre les deux ? En d'autres termes, si P ≠ NP existe-t-il des problèmes de NP qui soient hors de P mais pas NP-complet (langages « intermédiaires ») ?

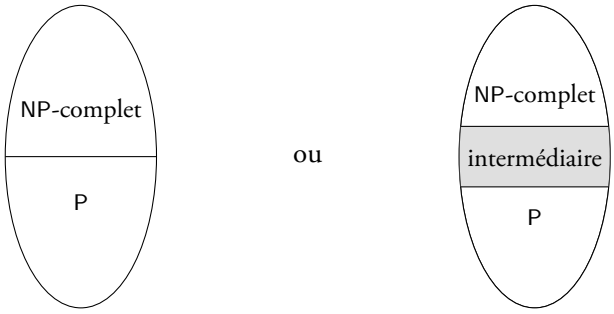


FIGURE 3.7 – Si P ≠ NP, existe-t-il des problèmes « intermédiaires » (figure de droite) ou non (figure de gauche) ?

En s'inspirant de méthodes issues de calculabilité, Ladner [Lad75a] répond à cette question dès 1975.

3-AK Théorème (Ladner, 1975)

Si P ≠ NP alors il existe un problème $A \in$ NP tel que :

- $A \notin$ P ;
- A n'est pas NP-complet.

Idée de la démonstration Le langage A sera SAT avec des « trous ». Plus précisément, pour certains intervalles de longueur de mots, A sera vide (« trous ») tandis que pour les autres intervalles de longueur de mots, A sera égal à SAT (voir figure 3.8). S'ils sont assez grands, les premiers intervalles garantissent que A n'est pas NP-complet, tandis que les seconds garantissent qu'il n'est pas dans P.

3-AL Remarque Avant de voir la démonstration détaillée, il nous faut formaliser le concept *d'énumération des machines de Turing fonctionnant en temps polynomial*. Le code d'une machine de Turing n'est qu'un mot sur un alphabet fini, ce n'est donc pas difficile d'énumérer le code de toutes les machines, $M_1, M_2, \ldots$ On remarquera que dans cette liste, pour chaque i il y a une infinité de machines équivalentes à M_i puisqu'il suffit d'ajouter des instructions inutiles pour augmenter la taille du code.

En revanche, pour énumérer seulement les machines fonctionnant en temps polynomial

il faut une astuce. Il s'agit d'ajouter un ruban à la machine M_i permettant de compter le nombre d'étapes de calcul. Dès que le compteur dépasse $i + n^i$ (où n est la taille de l'entrée), on arrête le calcul en rejetant. On obtient ainsi une nouvelle énumération (M'_i) de certaines machines de Turing, dans laquelle M'_i fonctionne en temps polynomial $(i+n^i)$ mais est éventuellement interrompue au cours de son calcul.

Si M est une machine fonctionnant en temps polynomial $p(n)$, il existe k tel que $p(n) \leqslant k + n^k$ pour tout n. Soit M_i l'une des machines équivalentes à M et telle que $i \geqslant k$. Alors M'_i fonctionne en temps $i+n^i \geqslant p(n)$, donc son calcul n'est pas interrompu et elle est aussi équivalente à M. Ainsi, (M'_i) est une énumération de toutes les machines fonctionnant en temps polynomial.

Démonstration (th. 3-AK) Si $h : \mathbb{N} \to \mathbb{N}$ est une fonction strictement croissante, on définit

$$\mathtt{SAT}_h = \{\varphi \mid \varphi \in \mathtt{SAT} \text{ et } \exists i\ h(2i) \leqslant |\varphi| < h(2i+1)\}.$$

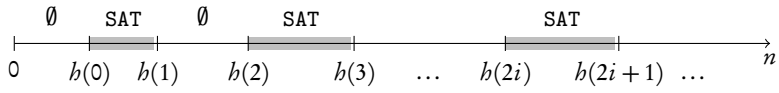


FIGURE 3.8 – Langage $\mathtt{SAT}_h$.

Si, sur l'entrée 1^n, on sait calculer en temps polynomial $h(0), h(1), \ldots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant n$, alors $\mathtt{SAT}_h \in \mathsf{NP}$. En effet, l'algorithme suivant convient pour $\mathtt{SAT}_h$ sur l'entrée φ :

- calculer $h(0), h(1), \ldots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant |\varphi|$;
- si i est impair, `rejeter` ;
- décider (de manière non déterministe) si $\varphi \in \mathtt{SAT}$.

L'objectif est maintenant de définir une fonction h ayant cette propriété, et telle que $\mathtt{SAT}_h$ ne soit ni dans P ni NP-complet. Pour cela, nous allons utiliser les deux observations suivantes.

1. Puisque $\mathsf{P} \neq \mathsf{NP}$ (par hypothèse) et que SAT est NP-complet, $\mathtt{SAT} \notin \mathsf{P}$.
   En d'autres termes, pour toute machine déterministe polynomiale M et tout entier n, il existe φ telle que
   - $|\varphi| \geqslant n$;
   - $M(\varphi)$ accepte ssi $\varphi \notin \mathtt{SAT}$
     (c'est-à-dire [$\varphi \in \mathtt{SAT}$ et $M(\varphi)$ rejette] ou [$\varphi \notin \mathtt{SAT}$ et $M(\varphi)$ accepte] ; en d'autres termes, M se trompe sur φ).

2. Si A est un langage fini (donc dans P), puisque $\mathsf{P} \neq \mathsf{NP}$ par hypothèse, $\mathtt{SAT}$ ne se réduit pas à A.

   En d'autres termes, pour toute réduction polynomiale f et tout entier n, il existe φ telle que

   – $|\varphi| \geqslant n$;
   – $f(\varphi) \in A$ ssi $\varphi \notin \mathtt{SAT}$
   (c'est-à-dire $[\varphi \in \mathtt{SAT}$ et $f(\varphi) \notin A]$ ou $[\varphi \notin \mathtt{SAT}$ et $f(\varphi) \in A]$).

On définit arbitrairement $h(\mathtt{0}) = \mathtt{0}$. Les valeurs suivantes de h vont être définies grâce au temps de fonctionnement d'une certaine machine M, qui prendra en entrée des entiers i et $m_0, \ldots, m_{i-1}$ (plus tard on prendra $m_j = h(j)$).

Soit (M_i) une énumération des machines fonctionnant en temps polynomial (voir remarque 3-AL) et qui calculent une fonction de $\{\mathtt{0},\mathtt{1}\}^\star$ dans $\{\mathtt{0},\mathtt{1}\}^\star$: on verra tantôt M_i comme calculant une réduction, tantôt comme acceptant un langage (dans ce mode « acceptation », le mot est accepté ssi le résultat est 1).

Soit M la machine suivante, sur l'entrée $(i, m_0, \ldots, m_{i-1})$:

– si i est impair, $i = 2k+1$, énumérer par taille croissante toutes les formules φ de taille $> m_{i-1}$ et simuler $M_k(\varphi)$, jusqu'à trouver φ telle que $[M_k(\varphi) = 1$ ssi $\varphi \notin \mathtt{SAT}]$;
– si i est pair, $i = 2k$, énumérer par taille croissante toutes les formules φ de taille $> m_{i-1}$ et simuler $M_k(\varphi)$, jusqu'à trouver φ telle que :
  ◦ soit $m_j \leqslant |M_k(\varphi)| < m_{j+1}$ pour j pair, et $[M_k(\varphi) \in \mathtt{SAT}$ ssi $\varphi \notin \mathtt{SAT}]$;
  ◦ soit $m_j \leqslant |M_k(\varphi)| < m_{j+1}$ pour j impair, et $\varphi \in \mathtt{SAT}$;
  ◦ soit $|M_k(\varphi)| \geqslant m_{i-1}$ et $\varphi \in \mathtt{SAT}$.

Par les observations 1 et 2 précédentes, de telles formules existent forcément.

On définit alors récursivement h comme suit : $h(i)$ est égal à $h(i-1)$ auquel on ajoute le temps de calcul de $M(i, h(\mathtt{0}), \ldots, h(i-1))$. On remarquera notamment que la formule φ recherchée par M vérifie $h(i-1) \leqslant |\varphi| < h(i)$ puisque le temps de calcul de M est évidemment supérieur à la taille de φ. De plus, sur l'entrée 1^n, on sait calculer en temps polynomial $h(\mathtt{0}), h(\mathtt{1}), \ldots, h(i)$ où i est le plus grand entier tel que $h(i) \leqslant n$, puisqu'il suffit de simuler les fonctionnements successifs de M jusqu'à ce que le nombre d'étapes total de la simulation dépasse n. Ainsi, $\mathtt{SAT}_h \in \mathsf{NP}$ par ce qui précède.

Nous prétendons que $\mathtt{SAT}_h$ n'est pas dans P et qu'il n'est pas NP-complet. Pour montrer cela, nous raisonnons par l'absurde.

Si $\mathtt{SAT}_h \in \mathsf{P}$ alors il est reconnu par une machine polynomiale M_k. Par définition de $\mathtt{SAT}_h$, pour tout φ si $h(2k) \leqslant |\varphi| < h(2k+1)$ alors $[\varphi \in \mathtt{SAT}_h$ ssi $\varphi \in \mathtt{SAT}]$. Or par définition de h, il existe φ de taille comprise entre $h(2k)$ et $h(2k+1)$ telle que $[M_k(\varphi) = 1$ ssi $\varphi \notin \mathtt{SAT}]$. Ainsi, M_k donne la mauvaise réponse pour φ, une contradiction.

Si $\texttt{SAT}_h$ est NP-complet, alors il existe une réduction polynomiale de SAT à $\texttt{SAT}_h$ calculée par une machine M_k, c'est-à-dire $\varphi \in \texttt{SAT} \iff M_k(\varphi) \in \texttt{SAT}_h$. Or par définition de h, il existe φ de taille comprise entre $h(2k-1)$ et $h(2k)$ telle que :

- soit $h(j) \leqslant |M_k(\varphi)| < h(j+1)$ pour j impair et $\varphi \in \texttt{SAT}$, dans ce cas $M_k(\varphi) \notin \texttt{SAT}_h$ et donc M_k n'est pas une réduction de SAT à $\texttt{SAT}_h$;
- soit $h(j) \leqslant |M_k(\varphi)| < h(j+1)$ pour j pair et $[M_k(\varphi) \in \texttt{SAT}$ ssi $\varphi \notin \texttt{SAT}]$. Or la taille de $M_k(\varphi)$ implique que $[M_k(\varphi) \in \texttt{SAT}$ ssi $M_k(\varphi) \in \texttt{SAT}_h]$, donc M_k n'est pas une réduction de SAT à $\texttt{SAT}_h$.

Ainsi, aucune machine polynomiale ne décide $\texttt{SAT}_h$ ni ne réduit SAT à $\texttt{SAT}_h$, ce qui conclut la preuve. □

3-AM **Remarques**

- Le résultat original de Ladner est en fait sensiblement plus fort. On peut en effet faire la même construction à partir d'un langage $A \notin \mathsf{P}$ quelconque, plutôt que de partir de SAT. On obtient donc l'énoncé suivant : si $A \notin \mathsf{P}$ alors il existe $B \notin \mathsf{P}$ tel que $B \leqslant^{\mathsf{p}}_{\mathsf{m}} A$ et $A \not\leqslant^{\mathsf{p}}_{\mathsf{m}} B$.
  En d'autres termes, sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$ il existe une infinité de langages $\texttt{SAT} >^{p}_{m} A_1 >^{p}_{m} A_2 >^{p}_{m} \dots$ tel que A_{i+1} se réduit à A_i mais A_i ne se réduit pas à A_{i+1}.
- On ne connaît pas de langage qui soit intermédiaire sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$ et qui soit plus « naturel » que la construction de Ladner ci-dessus (bien que le problème de la factorisation d'entiers et celui de l'isomorphisme de graphes que l'on verra au chapitre 10 soient des candidats).
- Il découle du théorème de Ladner que la NP-complétude *ne peut pas* être définie ainsi : « les langages $A \in \mathsf{NP}$ tels que $A \in \mathsf{P} \implies \mathsf{P} = \mathsf{NP}$ ». En effet, si $\mathsf{P} \neq \mathsf{NP}$ alors tout langage intermédiaire (un langage de NP qui ne soit ni complet ni dans P) serait complet (car « faux implique faux »), une absurdité.

✎ **3-AN** **Exercice**

Montrer le premier point de la remarque précédente.

✎ **3-AO** **Exercice**

Montrer qu'il existe un langage $A \notin \mathsf{EXP}$ qui ne soit pas EXP-difficile.

Indication : grâce à une stratégie de preuve similaire à celle pour le théorème de Ladner, transformer un problème $L \notin \mathsf{EXP}$ en un problème $A \notin \mathsf{EXP}$ qui ne soit pas EXP*-difficile.*

3.2.6 Théorème de Mahaney

En guise d'échauffement à cette partie, on pourra s'entraîner à résoudre l'exercice B-F.

Dans l'objectif de mieux comprendre les problèmes NP-complet, nous allons maintenant voir qu'un problème NP-difficile doit posséder beaucoup d'éléments. Plus précisément, si $\mathsf{P} \neq \mathsf{NP}$ alors il ne peut pas être « creux » au sens de la définition suivante.

3-AP Définition

Un langage A est dit *creux* s'il existe un polynôme $p(n)$ tel que pour tout n, le nombre de mots de A de taille n est au plus $p(n)$.

De manière équivalente, A est creux si le nombre de mots de A de taille $\leqslant n$ est majoré par un polynôme $q(n)$ (prendre $q(n) = \sum_{i=0}^{n} p(i)$).

On rappelle que $A^{=n}$ et $A^{\leqslant n}$ désignent respectivement l'ensemble des mots de A de taille n et de taille $\leqslant n$. On a donc $|A^{=n}| \leqslant p(n)$ et $|A^{\leqslant n}| \leqslant q(n)$ pour tout n.

Mahaney [Mah82] a en effet montré le résultat suivant.

3-AQ Théorème (Mahaney, 1982)

S'il existe un langage creux NP-difficile, alors $\mathsf{P} = \mathsf{NP}$.

Idée de la démonstration Sous l'hypothèse d'un langage creux NP-difficile, nous donnons un algorithme polynomial pour SAT. Celui-ci recherche la solution maximale (si elle existe) d'une formule φ. Si S est un langage creux NP-difficile et f une réduction de SAT à S, l'idée est de parcourir l'arbre des solutions possibles en se servant de f et du caractère « creux » de S pour élaguer des branches de l'arbre et n'explorer qu'un nombre polynomial de nœuds à chaque niveau.

Démonstration Supposons que S soit un langage creux NP-difficile possédant $\leqslant p(n)$ mots de taille $\leqslant n$, et que SAT se réduise à S via f (calculable en temps polynomial) : $[\varphi \in \mathtt{SAT}$ ssi $f(\varphi) \in S]$. Soit q un polynôme tel que $|f(\varphi)| \leqslant q(|\varphi|)$. Nous allons nous servir de f et du caractère « creux » de S pour donner un algorithme polynomial pour SAT. Sur l'entrée φ, celui-ci va rechercher la solution de φ maximale pour l'ordre lexicographique (si elle existe).

Si $\varphi(x)$ est une instance de SAT (où $x = x_1 \dots x_n$), pour $y = y_1 \dots y_n$ on note $\varphi_y(x)$ la formule $[(x \geqslant y) \wedge \varphi(x)]$, où $x \geqslant y$ signifie que x est après y dans l'ordre lexicographique et peut s'écrire simplement avec les symboles autorisés habituels en disant que soit $x = y$, soit il existe un préfixe identique de taille $k-1$ et que le k-ème bit de x est 1 tandis que celui de y est 0 :

$$\Big(\bigwedge_{i\in[1,n]} x_i \leftrightarrow y_i\Big) \vee \bigvee_{k\in[1,n]} \left(\left(\bigwedge_{i<k} x_i \leftrightarrow y_i\right) \wedge (x_k \wedge \neg y_k)\right).$$

Ainsi, $\varphi_y \in \texttt{SAT}$ ssi φ possède une solution x supérieure à y (pour l'ordre lexicographique). Le code de φ_y est bien sûr calculable en temps polynomial à partir de y et de φ. On notera m la taille d'une formule φ_y.
L'avantage de considérer les formules φ_y repose sur le fait que l'ensemble $\{y \mid \varphi_y \in \texttt{SAT}\}$ est clos vers la gauche, c'est-à-dire que si $\varphi_{y'} \in \texttt{SAT}$ et $y \leqslant y'$ alors $\varphi_y \in \texttt{SAT}$ (cf. illustration figure 3.9). Cela va faciliter la recherche de la solution maximale de φ.

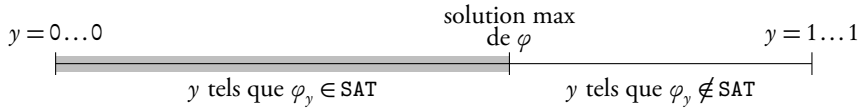


FIGURE 3.9 – Clôture vers la gauche des y tels que $\varphi_y \in \texttt{SAT}$.

Notre algorithme pour rechercher la solution maximale de φ va développer l'arbre binaire complet des φ_y pour $|y| = n$, dans lequel :

- les 2^i nœuds à la profondeur i sont tous les $\varphi_{z0^{n-i}}$ pour $|z| = i$;
- les fils de $\varphi_{z0^{n-i}}$ sont $\varphi_{z'0^{n-i-1}}$ et $\varphi_{z''0^{n-i-1}}$ où $z' = z0$ et $z'' = z1$.

La branche descendant d'un nœud $z0\ldots0$ est donc l'ensemble des y ayant z comme préfixe. Grâce à f et au caractère « creux » de S, nous pourrons garder seulement $p(q(n))$ nœuds à chaque niveau (cf. figure 3.10). Il faut toutefois maintenir l'invariant suivant :

à chaque niveau, l'une des branches restantes contient la solution maximale de φ
(de sorte que l'une des feuilles est égale à cette solution maximale).

Il y a deux raisons grâce auxquelles on peut supprimer des nœuds dans l'arbre tout en préservant l'invariant :

- si $f(\varphi_y) = f(\varphi_{y'})$ et $y < y'$, alors par définition de f soit φ_y et $\varphi_{y'}$ sont tous deux dans SAT, soit ils sont tous deux hors de SAT. Donc la solution x maximale de φ vérifie soit $x \geqslant y'$, soit $x < y$. Dans les deux cas, il n'est pas nécessaire d'explorer la branche de y. Cela permet de se restreindre aux nœuds ayant des images par f toutes différentes.
- Si après application de l'élagage précédent, un niveau contient encore un nombre $N > p(q(m))$ nœuds, on peut supprimer les $(N - p(q(m)))$ plus grands nœuds, car ils sont « en surplus ». En effet, les images des N nœuds sont toutes différentes (par le point précédent), donc au moins $N - p(q(m))$ nœuds ont leur image hors de S (puisque $|S^{\leqslant q(m)}| \leqslant p(q(m))$), c'est-à-dire qu'ils sont eux-mêmes hors de SAT. Et par clôture à gauche, seuls les $p(q(m))$ premiers peuvent éventuellement appartenir à SAT.

L'algorithme parcourt l'arbre niveau par niveau en élaguant les branches inutiles, de sorte qu'il ne conserve qu'un nombre polynomial de nœuds à chaque niveau (au plus

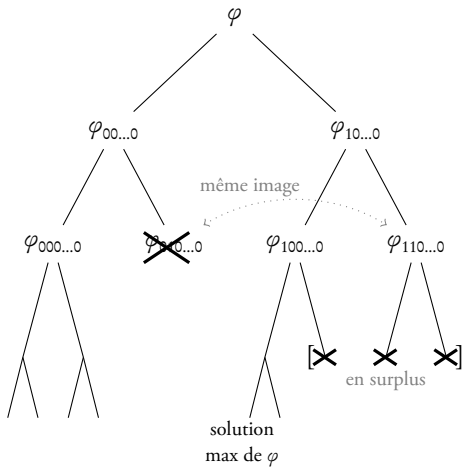


FIGURE 3.10 – Élagage de l'arbre pour la formule φ.

$p(q(m)))$. Au dernier niveau de l'arbre (le niveau n), il y a $\leqslant p(q(m))$ feuilles $y_1, \ldots, y_k$. Par l'invariant, l'une d'entre elles correspond à la solution maximale de φ si elle existe. Il suffit alors de vérifier si $\varphi(y_i)$ est vrai pour l'une des feuilles y_i. Puisqu'il y a n niveaux, l'algorithme est polynomial.

Plus précisément, on maintient l'ensemble Y_i des préfixes y indexant les nœuds du niveau i. Au départ, $Y_1 \leftarrow \{0,1\}$. Voici la procédure pour calculer Y_{i+1}, où U est un ensemble permettant de conserver les images par f des nœuds vus jusqu'à présent. On énumère les nœuds du niveau précédent par ordre décroissant afin de pouvoir supprimer le plus petit nœud lorsque deux nœuds ont la même image.

- $Y_{i+1} \leftarrow \emptyset$;
- $U \leftarrow \emptyset$;
- `Pour chaque` $y \in Y_i$ par ordre décroissant `faire`
  - $u \leftarrow f(\varphi_{y10\ldots0})$,
  - `si` $u \notin U$ `alors`
    - $Y_{i+1} \leftarrow Y_{i+1} \cup \{y1\}$
    - $U \leftarrow U \cup \{u\}$,
  - $u \leftarrow f(\varphi_{y00\ldots0})$,
  - `si` $u \notin U$ `alors`
    - $Y_{i+1} \leftarrow Y_{i+1} \cup \{y0\}$
    - $U \leftarrow U \cup \{u\}$;

– si Y_{i+1} contient $N > p(q(m))$ éléments, supprimer les $(N - p(q(m)))$ plus grands.

Une fois que Y_n est calculé, il suffit de tester si $\varphi(y)$ est vrai pour l'un des $y \in Y_n$. Si c'est le cas, alors $\varphi \in \texttt{SAT}$, sinon $\varphi \notin \texttt{SAT}$ puisque par l'invariant, Y_n contient la solution maximale de φ si elle existe. Cet algorithme pour SAT fonctionne en temps polynomial, donc P = NP. □

3.2.7 Algorithme polynomial pour SAT si P = NP

Pour compléter ce chapitre sur NP, nous allons voir que la question « P = NP ? » se ramène à l'analyse d'un unique algorithme. Il nous faut tout d'abord un lemme simple qui montre comment transformer un problème d'évaluation en un problème de décision.

3-AR Lemme

Si P = NP alors il existe une machine déterministe fonctionnant en temps polynomial qui, sur l'entrée φ (une instance de SAT), accepte et renvoie une affectation satisfaisant φ si φ est satisfaisable, ou rejette sinon.

Idée de la démonstration Il s'agit de réaliser une recherche préfixe de la plus petite solution de φ.

Démonstration Soit L le langage $L = \{(\varphi, a) \mid \exists b, \varphi(ab) = 1\}$, où a est le début d'une affectation (partielle) de φ et b est la fin de cette affectation. En d'autres termes, L permet de savoir si l'on peut compléter a de manière à obtenir une solution de φ. C'est bien sûr un langage de NP donc, par hypothèse, de P. Voici l'algorithme pour trouver la plus petite solution de $\varphi(x_1, \ldots, x_n)$:

– $a \leftarrow \epsilon$ (affectation vide) ;
– `pour` i de 1 à n `faire`
  ◦ si $(\varphi, a0) \in L$ alors $a \leftarrow a0$,
  ◦ sinon $a \leftarrow a1$;
– si $\varphi(a) = 1$ alors `accepter` et renvoyer a,
– sinon `rejeter`.

C'est ainsi une recherche préfixe de la plus petite solution de φ, qui se déroule en temps polynomial puisque le test $(\varphi, a0) \in L$ se fait en temps polynomial. □

Si P = NP alors SAT possède un algorithme polynomial, mais on ne sait pas lequel. Avec un peu d'astuce on peut tout de même donner un algorithme A fixé tel que A résout

« presque » SAT et fonctionne en temps polynomial si P = NP. Il s'agit d'une variante de l'algorithme optimal de recherche universelle de Levin [Lev73]. L'énoncé précis est donné à la proposition 3-AT.

Pour décrire l'algorithme, on aura besoin comme à la remarque 3-AL d'une énumération (M_i) des machines fonctionnant en temps polynomial. Ici, ces machines sont censées renvoyer une affectation des variables de la formule φ donnée en entrée (comme au lemme 3-AR).

Voici cet algorithme M sur une entrée φ de taille n (une formule booléenne, c'est-à-dire une instance de SAT).

- $i \leftarrow 1$;
- pour i de 1 à n faire
  - simuler $M_i(\psi)$ pour toute formule ψ de taille $\leqslant \log n$,
  - si pour tout ψ, M_i donne toujours une affectation valide lorsque $\psi \in$ SAT, sortir de la boucle ;
- simuler $M_i(\varphi)$, produisant un résultat a (une affectation des variables de φ) ;
- si $\varphi(a) = 1$ accepter, sinon rejeter.

3-AS **Remarque** Comme dans tout langage de programmation, la valeur de i après la boucle pour est égale soit à la valeur de i au moment de la sortie de boucle si on sort prématurément, soit à $n+1$ si la boucle s'est déroulée complètement.

3-AT **Proposition**

Si P = NP alors la machine M ci-dessus vérifie :

- M fonctionne en temps polynomial ;
- M décide SAT pour toute entrée suffisamment grande, c'est-à-dire qu'il existe m tel que pour tout φ, si $|\varphi| \geqslant m$ alors $[M(\varphi) = 1$ ssi $\varphi \in$ SAT$]$.

Idée de la démonstration M simule les machines M_i les unes après les autres. Si P = NP alors l'une d'entre elles, M_{i_0}, résout SAT. Pour toutes les entrées suffisamment grandes, on atteint M_{i_0} et notre algorithme donne alors la bonne réponse. Le temps de calcul est alors celui de M_{i_0}, qui est polynomial.

Démonstration Si P = NP, soit M_{i_0} la première machine de l'énumération qui renvoie une affectation valide pour toute formule $\varphi \in$ SAT. Toute machine M_i pour $i < i_0$ se

trompe donc sur au moins une formule (c'est-à-dire qu'il existe $\varphi \in \texttt{SAT}$ et $M_i(\varphi)$ n'est pas une affectation valide) : soit N la taille minimum telle que $\forall i < i_0$, M_i se trompe sur une formule de taille $< N$.

Alors pour $|\varphi| \geqslant \max\{i_0, 2^N\}$, la boucle `pour` trouve une erreur pour toutes les machines M_i telles que $i < i_0$, donc on sort de la boucle pour $i = i_0$. On simule alors la machine M_{i_0} qui donne une affectation valide si une telle affectation existe. Ainsi, $M(\varphi) = 1$ si $\varphi \in \texttt{SAT}$, et bien sûr, par conception de l'algorithme, pour tout $\varphi \notin \texttt{SAT}$, $M(\varphi) = 0$.

Pour le temps d'exécution, rappelons que M_i fonctionne en temps $O(n^i)$, donc on simule M_i en temps $O(n^{2i})$ par la machine universelle de la proposition 1-Q. On exécute au plus i_0 fois la boucle `pour`, toutes les machines simulées fonctionnent en temps $\leqslant n^{i_0}$ et sont simulées sur $2^{\log n} = n$ formules de taille $\log n$. Donc la boucle `pour` prend un temps $\leqslant i_0 n (\log n)^{2i_0}$. Puis la simulation de $M_{i_0}(\varphi)$ prend un temps $O(n^{2i_0})$: au total, M fonctionne donc en temps polynomial. □

3-AU **Remarques**

- Si $\mathsf{P} \neq \mathsf{NP}$ alors la machine M ci-dessus ne fonctionne pas en temps polynomial car elle va alors simuler des machines M_i pour i de plus en plus grand. Or M_i fonctionne en temps n^i, ce qui fait que M n'est pas polynomiale. De plus, il n'y a pas de raison qu'elle reconnaisse correctement `SAT`, même sur des instances suffisamment grandes.
- On ne sait pas construire une telle machine M qui fonctionnerait sur toute entrée φ (et non seulement sur des entrées suffisamment grandes).

Pour compléter les réflexions sur la NP-complétude, les exercices B-F et B-C en annexe abordent des thèmes traités dans ce chapitre.

4 Considérations de base sur l'espace

Jusqu'à présent, nous avons abordé la complexité algorithmique seulement à travers le prisme du temps de calcul. Pour qu'un calcul soit utilisable, il est en effet nécessaire qu'il s'exécute rapidement. Mais il existe une autre ressource critique : la mémoire. Par exemple, un algorithme effectuant n^4 opérations peut éventuellement encore être considéré comme raisonnablement efficace en termes de temps d'exécution puisque le temps nécessaire à une machine actuelle effectuant 10^{11} opérations par seconde pour exécuter l'algorithme sur une entrée de taille 10000 est d'environ un jour. Mais si à chaque étape il utilise une nouvelle case mémoire alors il lui faudra 10^{16} cases pour réaliser son calcul. C'est rédhibitoire actuellement, tout au moins si l'on se restreint à la mémoire vive afin d'éviter les lents accès disque.

De la même façon que nous l'avons fait pour le temps de calcul, nous allons donc définir et étudier des classes de complexité en espace. Cela nous amènera notamment à l'étude de classes plus petites que P, du rôle du non-déterminisme pour l'espace ainsi que de problèmes complets naturels.

4.1 Espace déterministe

Comme pour le temps, nous définirons la complexité en espace en fonction de la taille de l'entrée. L'espace utilisé par une machine est simplement le nombre de cases différentes visitées sur les rubans de travail.

4.1.1 Définitions

Nous rappelons ici la définition 1-F concernant le décompte de l'espace utilisé par une machine avant de définir des classes de complexité.

4-A Définition

– Si M est une machine de Turing déterministe et x une entrée, l'espace utilisé par M sur x est le nombre de cases différentes visitées par M sur ses *rubans de travail* au cours de son calcul.

– Si $s : \mathbb{N} \to \mathbb{N}$ est une fonction, on dit que M fonctionne en espace $O(s(n))$ si *elle s'arrête sur toute entrée* et s'il existe une constante $\alpha > 0$ telle que pour toute entrée x, $M(x)$ utilise un espace $\leqslant \alpha s(|x|)$.

– Si $s : \mathbb{N} \to \mathbb{N}$ est une fonction, la classe $\mathsf{DSPACE}(s(n))$ est l'ensemble des langages reconnus par une machine fonctionnant en espace $O(s(n))$.

4-B Remarques

– Une machine pourrait boucler infiniment et ne jamais s'arrêter tout en utilisant un espace $\leqslant s(n)$, c'est pourquoi nous devons imposer l'arrêt sur toute entrée des machines que nous considérons.

– Dans une définition alternative, équivalente mais moins élégante, on pourrait imposer que la tête du ruban de lecture ne dépasse de plus d'une case, ni sur la gauche ni sur la droite, les extrémités du mot d'entrée. Cela éviterait de pouvoir détourner le rôle de ce ruban en utilisant la position de la tête pour effectuer des calculs, et simplifierait ainsi la preuve du lemme 4-V. À l'exercice 4-Y on verra l'équivalence des deux définitions.

Ainsi, comme pour le temps, la complexité en espace n'est définie qu'à une constante multiplicative près. Cela vient du fait que, de la même manière que le théorème d'accélération linéaire 2-E, on peut diviser par une constante l'espace nécessaire en augmentant l'alphabet de travail. La démonstration est très similaire à celle du théorème 2-E mais on évite de recopier l'entrée puisqu'on peut la lire directement sur le ruban d'entrée sans pénaliser l'espace. On obtient alors le théorème suivant que nous donnons sans démonstration.

4-C Théorème (réduction linéaire de l'espace)

Soit M une machine fonctionnant en espace $s(n)$ et décidant un langage A. Pour toute constante $\epsilon > 0$, il existe une machine M' décidant A et fonctionnant en espace $\leqslant \epsilon s(n)$.

✎ **4-D Exercice**

Faire la démonstration du théorème précédent.

Afin de s'habituer aux algorithmes en espace restreint, il n'est pas inutile de voir dès à présent quelques exemples.

4.1.2 Espace logarithmique

Un peu de recul

Le fait de ne compter que les rubans de travail dans le calcul de la complexité en espace permet d'autoriser en un espace sous-linéaire à lire quand même l'entrée en entier, ce qui est nécessaire pour la plupart des problèmes non triviaux.

Comme nous les avons définies, nos machines de Turing ne peuvent pas « tricher » avec les rubans d'entrée et de sortie car ceux-ci sont respectivement en lecture seule et en écriture seule, ce qui ne leur permet pas vraiment de réaliser des calculs.

Cette remarque nous amène à la définition de la classe très usuelle des langages reconnus en espace logarithmique. Par abus de langage, on notera $\log n$ pour désigner la fonction sur les entiers $\lfloor \log n \rfloor$ afin de respecter la définition ci-dessus, bien que cela ne change pas grand-chose puisque l'espace est mesuré à une constante près.

4-E Définition (espace logarithmique)

La classe L est l'ensemble des langages reconnus en espace logarithmique $O(\log n)$, c'est-à-dire $\mathsf{L} = \mathsf{DSPACE}(\log n)$.

La restriction sur l'espace semble si forte qu'on peut se demander ce qu'il est possible de calculer en espace logarithmique. Voici quelques exemples.

4-F Exemple Machine qui ajoute 1 à un nombre binaire donné en entrée. Cette machine est en fait un automate fini puisqu'elle n'a pas besoin de ruban de travail : elle ne dispose que de son ruban d'entrée en lecture seule et de son ruban de sortie en écriture seule. Son fonctionnement est le suivant :

- elle se déplace sur la dernière lettre de son ruban d'entrée ;
- de droite à gauche, tant qu'elle lit un 1 sur son ruban d'entrée, elle écrit 0 sur son ruban de sortie (la retenue se propage) ;
- dès qu'elle lit 0 en entrée elle écrit 1 en sortie (la retenue) et continue son parcours de droite à gauche, en recopiant maintenant chaque symbole sur la sortie (il n'y a plus de retenue) ;
- si elle arrive au début de l'entrée sans avoir vu de 0, alors elle ajoute un 1 en tête (la retenue) sur le ruban de sortie.

Une décrémentation peut bien sûr se faire de manière similaire.

4-G Exemple Machine à un ruban de travail qui renvoie $\lfloor \log n \rfloor$ en unaire sur l'entrée 1^n (n en unaire) :

- un compteur binaire est initialisé à 0 sur le ruban de travail ;

- on parcourt l'entrée de gauche à droite tant que le symbole 1 est lu ;
- le compteur est incrémenté de 1 à chaque étape à la façon de l'exemple précédent ;
- enfin, on parcourt à partir de la deuxième case la valeur finale du compteur en écrivant un 1 sur le ruban de sortie à chaque symbole lu.

Puisque la longueur de l'écriture binaire de n est $1+\lfloor \log n \rfloor$, on obtient bien le résultat voulu. L'espace utilisé sur le ruban de travail ne dépasse jamais $\log n$.

4-H **Exemple** Machine à deux rubans de travail qui prend en entrée deux nombres x et y en binaire, et calcule leur somme $x+y$ en binaire. On va effectuer l'algorithme de l'école primaire pour l'addition. Mais puisqu'on ne peut pas écrire sur le ruban d'entrée pour se repérer dans les mots x et y, ni les recopier sur un ruban de travail (ce qui dépasserait l'espace logarithmique autorisé), l'idée est de maintenir un compteur binaire c sur le premier ruban de travail pour retenir la position en cours de traitement dans x et y.

- Initialiser le compteur c à 0 et retenir une retenue r égale à 0.
- Tant que x et y n'ont pas été lus complètement, faire
  - aller à la fin du mot x ;
  - se déplacer à gauche de c cases : pour cela, recopier c sur le second ruban de travail et le décrémenter jusqu'à tomber sur 0, en se déplaçant à gauche à chaque décrémentation ;
  - retenir le chiffre lu a ;
  - aller à la fin du mot y ;
  - se déplacer à gauche de c cases (même méthode) ;
  - retenir le chiffre lu b ;
  - si $a+b+r>1$ alors écrire $a+b+r-2$ sur le ruban de sortie, se déplacer à gauche et retenir $r=1$;
  - sinon écrire $a+b+r$ sur le ruban de sortie, se déplacer à gauche et retenir $r=0$,
  - incrémenter c de 1 comme à l'exemple 4-F.

L'espace utilisé sur les rubans de travail n'excède pas deux fois la longueur du compteur et est donc logarithmique.

On le voit, l'astuce pour utiliser seulement un espace logarithmique consiste souvent à utiliser des compteurs en binaire pour se repérer sur les rubans. En réalité, en espace logarithmique on peut calculer beaucoup de fonctions de bases (addition, multiplication, etc.), peut-être plus qu'on pourrait le croire a priori.

✎ **4-I Exercice**

Décrire une machine de Turing fonctionnant en espace logarithmique qui calcule le produit de deux nombres binaires x et y donnés en entrée.

4.1.3 Composition

Lorsque deux fonctions $f, g : \Sigma^\star \to \Sigma^\star$ sont calculables en espace $s(n)$, il n'est pas évident que leur composition $f \circ g$ le soit également : il se pourrait en effet que $g(x)$ soit de taille bien supérieure à s (c'est en effet un mot destiné seulement au ruban de sortie qui ne compte pas dans l'espace) et qu'on ne puisse donc pas l'écrire sur un ruban de travail pour pouvoir ensuite calculer f dessus. Pourtant on peut s'en sortir moyennant une astuce comme le montre la proposition suivante.

4-J Proposition

Si $f, g : \Sigma^\star \to \Sigma^\star$ sont deux fonctions calculables en espace $s(n)$, alors leur composition $f \circ g$, $x \mapsto f(g(x))$, est calculable en espace $O(s(|x|) + \log|g(x)|)$.

Nous verrons par la suite (proposition 4-W) que $|g(x)| \leqslant 2^{O(s(|x|))}$, donc en réalité $f \circ g$ est également calculable en espace $O(s(n))$.

Idée de la démonstration Pour calculer $f(g(x))$, il s'agit simplement d'exécuter la machine pour f et, dès qu'elle souhaite lire une nouvelle case de son entrée, de recalculer $g(x)$ à la volée (grâce à la machine pour g) sans stocker le résultat.

Démonstration Remarquons tout d'abord qu'en espace $s(|x|) + \log|g(x)|$, sur l'entrée (i, x) on peut calculer le i-ème symbole de $g(x)$. Voici comment procéder, où M_g désigne la machine calculant g en espace $s(n)$.

- Simuler $M_g(x)$ sans écrire sur le ruban de sortie, mais en mettant à jour sur un ruban de travail auxiliaire la position de la case la plus à gauche du ruban de sortie dans laquelle un symbole non vide est inscrit. Cela se fait en incrémentant un compteur dès que la tête dépasse la précédente position sur la gauche. L'espace utilisé est $s(|x|)$ pour la machine M_g elle-même, et $O(\log|g(x)|)$ pour maintenir le compteur. Nous connaissons donc maintenant le numéro de la case du premier symbole de $g(x)$ sur le ruban de sortie.
- Simuler une nouvelle fois $M_g(x)$ sans écrire sur le ruban de sortie, mais en mettant à jour sur un ruban de travail auxiliaire la position de la tête sur le ruban de sortie.
- Dès que la tête passe par la i-ème case à partir de la position la plus à gauche (c'est-à-dire dès qu'on passe par la case correspondant au i-ème symbole de $g(x)$), mettre à jour le nouveau symbole inscrit.

À la fin du calcul, nous avons bien le dernier symbole écrit sur la case correspondant au i-ème symbole de $g(x)$. Si M_g a k_g rubans de travail alors la machine que l'on vient de décrire en a $k_g + 2$.

Voici maintenant comment calculer $f(g(x))$ en espace $O(s(|x|)+\log|g(x)|)$, où M_f désigne la machine calculant f en espace $s(n)$. Si k_f désigne le nombre de rubans de travail de M_f, alors le nombre de rubans de travail de la machine suivante est k_f+k_g+2. Le ruban de sortie correspond à celui de M_f.

- $i \leftarrow 1$;
- `tant que` M_f n'atteint pas un état terminal `faire`
  - calculer le i-ème symbole de $g(x)$, noté a,
  - simuler une étape de calcul de M_f en remplaçant le résultat de la lecture du ruban d'entrée par le symbole a,
  - si la tête du ruban d'entrée de M_f se déplace vers la gauche, alors $i \leftarrow i-1$; si elle se déplace vers la droite, alors $i \leftarrow i+1$.

Puisqu'on réutilise à chaque fois l'espace pris par le calcul des symboles de $g(x)$, l'espace total est $O(s(|x|)+\log|g(x)|)$. □

4.1.4 Théorème de hiérarchie

Comme pour la complexité en temps, un théorème de hiérarchie dû à Stearns, Hartmanis et Lewis [SHL65] montre que disposer de plus d'espace permet de décider strictement plus de langages. Pour l'énoncer, nous avons besoin de la notion de fonction constructible en espace.

4-K Définition

Une fonction $s : \mathbb{N} \to \mathbb{N}$ est *constructible en espace* s'il existe une constante α et une machine de Turing M qui, sur l'entrée 1^n (l'entier n en unaire) renvoie $1^{s(n)}$ (l'entier $s(n)$ en unaire) en utilisant un espace $\leqslant \alpha s(n)$.

Là encore, les fonctions usuelles de complexité sont constructibles en espace, par exemple :

- $s(n) = \lfloor \log n \rfloor$ comme on l'a vu à l'exemple 4-G ;
- $s(n) = n$;
- si $s(n)$ et $s'(n)$ sont constructibles en espace, alors il en est de même de leur produit et de leur somme : ainsi, tous les $\lfloor \log n \rfloor^k$ et tous les polynômes sont constructibles en espace ;
- $s(n) = 2^n$ est constructible en espace.

✎ **4-L** **Exercice**

Montrer que les fonctions ci-dessus sont constructibles en espace.

✎ **4-M** **Exercice**

Montrer que si s est constructible en espace et $s = o(\log n)$ alors s est ultimement constante.

Indication : utiliser la proposition 4-W ci-après.

On montre le théorème de hiérarchie en espace suivant de la même façon que pour la complexité en temps (théorème 2-J). L'écart entre f et g est plus réduit que pour le temps car l'espace utilisé par la machine universelle de la proposition 1-Q n'est qu'un facteur constant plus grand que celui de la machine simulée.

4-N **Théorème (hiérarchie en espace déterministe)**

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions telles que $f(n) \neq 0$ (pour tout $n \in \mathbb{N}$), g est constructible en espace et $f = o(g)$. Alors $\mathsf{DSPACE}(f(n)) \subsetneq \mathsf{DSPACE}(g(n))$.

✎ **4-O** **Exercice**

Montrer ce théorème.

4.2 Espace non déterministe

De la même façon que pour la complexité en temps, on peut aussi considérer des machines de Turing non déterministes. Rappelons qu'un mot est accepté par une machine de Turing non déterministe si et seulement s'il existe un chemin acceptant.

4-P **Définition**

- Si $s : \mathbb{N} \to \mathbb{N}$ est une fonction, on dit qu'une machine non déterministe N fonctionne en espace $O(s(n))$ si *elle s'arrête sur toute entrée le long de toute exécution* et s'il existe une constante $\alpha > 0$ telle que pour toute entrée x, $N(x)$ utilise un espace $\leqslant \alpha s(n)$ *sur tout chemin de calcul.*
- Si $s : \mathbb{N} \to \mathbb{N}$ est une fonction, la classe $\mathsf{NSPACE}(s(n))$ est l'ensemble des langages reconnus par une machine de Turing non déterministe fonctionnant en espace $O(s(n))$.

Encore une fois, nous disposons d'un théorème de hiérarchie. Pour le montrer, on peut suivre la preuve du théorème 2-J de hiérarchie en temps *déterministe* mais avec la machine universelle de la remarque 2-AC, et utiliser le théorème 4-AZ d'Immerman-Szelepcsényi ci-dessous pour prendre le complémentaire. Si l'on se contente de $f(n+1) = o(g(n))$, alors on peut également suivre la preuve du théorème 2-AI de hiérarchie en temps non déterministe combinée avec le théorème 4-AS de Savitch ci-dessous pour simuler une machine non déterministe par une machine déterministe.

4-Q **Théorème (hiérarchie en espace non déterministe)**

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions telles que $f(n) \neq 0$ (pour tout $n \in \mathbb{N}$), g est constructible en espace et $f = o(g)$. Alors $\mathsf{NSPACE}(f(n)) \subsetneq \mathsf{NSPACE}(g(n))$.

✎ **4-R** **Exercice**

Rédiger les deux stratégies de preuve mentionnées ci-dessus.

Cependant, comme on le verra à la section 4.5, le non-déterminisme occupe une moins grande importance pour la complexité en espace, sauf pour l'espace logarithmique. À cette section nous étudierons deux exemples d'algorithmes non déterministes en espace restreint.

4.3 Comparaison avec les classes en temps

Si l'on connaît plus de résultats sur la complexité en espace (par rapport à celle en temps), c'est principalement grâce à une technique, consistant à étudier le graphe des configurations de la machine fonctionnant en espace $s(n)$. Nous en verrons plusieurs illustrations, ce qui justifie d'en donner d'abord le cadre général.

4-S **Définition (graphe des configurations)**

Soit N une machine de Turing (éventuellement non déterministe).

- On appelle *configuration de travail* de N la donnée de l'état de la machine, du contenu des rubans de travail et de la position des têtes des rubans d'entrée et de travail. Il s'agit donc d'une configuration « normale » comme au chapitre 1 mais où le contenu des rubans d'entrée et de sortie n'est pas pris en compte, ni la position de la tête sur le ruban de sortie.
- Le *graphe des configurations* de $N(x)$ est le graphe orienté dont les sommets sont les configurations de travail prises par N au cours de son calcul sur l'entrée x, et il y a un arc d'un sommet c_1 à un sommet c_2 si N peut aller en une étape de la configuration c_1 à la configuration c_2 (cf. figure 4.1).

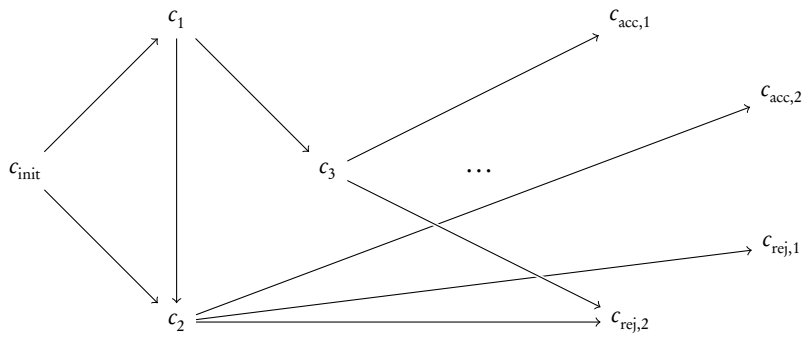


FIGURE 4.1 – Graphe des configurations d'une machine non déterministe N sur une entrée x, où c_{init} est la configuration initiale, $c_{\text{acc},1}$ et $c_{\text{acc},2}$ sont des configurations acceptantes et $c_{\text{rej},1}$ et $c_{\text{rej},2}$ rejetant.

4-T **Remarques**

– La raison pour laquelle on oublie le contenu des rubans d'entrée et de sortie est que le premier est en lecture seule et son contenu n'est donc jamais modifié, et le second est en écriture seule, ce qui fait que son contenu n'influence pas le calcul de la machine.

– Une configuration de travail contient toute l'information nécessaire pour déterminer le comportement futur d'une machine déterministe, puisque celui-ci ne dépend que de l'état de la machine et du contenu des cases lues sur les rubans de travail et d'entrée.

L'utilité de ce graphe dans le cadre de la complexité en espace vient du lemme suivant qui montre qu'il ne peut pas être trop grand.

4-U **Remarque** De nombreux textes utilisent, parfois de manière implicite, la définition alternative décrite à la remarque 4-B où la tête du ruban de lecture ne peut pas sortir du mot d'entrée. Dans ce cas, seule le début, beaucoup plus simple, de la preuve du lemme suivant est nécessaire. C'est la raison pour laquelle nous l'avons divisée en deux parties : en première lecture, on pourra sans crainte laisser de côté la seconde partie.

De même, selon cette définition alternative, il ne serait plus utile de différencier configurations de travail et configurations potentielles (cf. définition 4-AH) et il suffirait de dénombrer ces dernières.

4-V Lemme

Soit N une machine de Turing (éventuellement non déterministe) fonctionnant en espace $s(n) \geqslant \log n$. Alors il existe une constante α telle que pour toute entrée x, le graphe des configurations de $N(x)$ possède au plus $2^{\alpha s(|x|)}$ sommets.

En particulier, il existe une constante γ telle que la position de la tête du ruban de lecture reste dans l'intervalle $[-2^{\gamma s(|x|)}, 2^{\gamma s(|x|)}]$ au cours du calcul $N(x)$.

Idée de la démonstration La borne vient essentiellement du fait que seules $O(s(n))$ cases sont non vides sur chaque ruban de travail, donc le nombre de contenus différents est majoré par $|\Gamma|^{O(s(n))}$. Il faut en outre prendre en compte le nombre d'états et la position des têtes, ce qui ne change pas fondamentalement cette borne. La partie technique est due à la tête du ruban d'entrée qui n'est pas contrainte par la borne en espace (ce qui pourrait être évité en adoptant la définition alternative de la remarque 4-B) et on utilisera un argument de « pompage » pour résoudre ce problème.

Démonstration Soit k le nombre de rubans de N, et $n = |x|$.

Nous allons tout d'abord ignorer les positions possibles de la tête sur le ruban d'entrée, ce que nous appellerons de *configurations partielles*. La machine N peut être dans l'un des $|Q|$ états. Le nombre de cases visitées sur les rubans de travail est $\leqslant \beta s(n)$ pour une certaine constante $\beta > 0$, c'est-à-dire que la position de chaque tête prend au plus $\beta s(n)$ valeurs. Le contenu des cases non visitées reste inchangé (cases vides), donc le contenu de chaque ruban de travail prend au plus $|\Gamma|^{\beta s(n)}$ valeurs possibles. Au total, sans compter la position de la tête sur le ruban d'entrée, le nombre de possibilités est majoré par

$$|Q|(\beta s(n))^k |\Gamma|^{\beta k s(n)}.$$

En prenant en considération que β, $|Q|$, $|\Gamma|$ et k sont constants et que $s(n)^k \leqslant 2^{k s(n)}$, on peut majorer ce nombre de configurations partielles par $2^{\gamma s(n)}$ pour une certaine constante γ ne dépendant que de la machine N.

Afin de dénombrer les configurations de travail, il nous faut maintenant montrer qu'il n'y a pas trop de positions possibles pour la tête du ruban d'entrée (cette partie plus technique serait inutile si l'on adoptait la définition alternative de la remarque 4-B). C'est un argument de « pompage » (comme dans le lemme de l'étoile pour les automates finis) pour montrer que si la tête va trop loin sur le ruban de lecture, la machine boucle et le calcul ne termine pas.

Si N est non déterministe, on fixe une exécution de $N(x)$. Soit m la position de la case extrême visitée sur le ruban de lecture à droite du mot d'entrée x (donc $m > 0$). On note i_1 la première étape à laquelle on atteint la case à la position m.

Si $m > n + 1 + 2^{\gamma s(n)}$, alors pour arriver la première fois à cette position extrême, la machine a effectué au moins $1 + 2^{\gamma s(n)}$ étapes en ne visitant que des cases vides sur le ruban d'entrée. On note i_0 la dernière étape avant i_1 à laquelle la tête est en position $n + 1$: en particulier, $i_1 - i_0 \geqslant 1 + 2^{\gamma s(n)}$ et entre i_0 et i_1 seules des cases vides sont lues sur le ruban d'entrée. On définit maintenant comme suit une suite d'intervalles $[u_j, v_j]$ inclus dans $[i_0, i_1]$ (voir figure 4.2) :

- $u_0 = i_0$;

– v_j est tel que $[u_j, v_j]$ est le plus grand intervalle d'étapes pendant lequel N se déplace uniquement vers la droite ;

– u_{j+1} est la première étape après v_j à laquelle la position de la tête est à droite de celle à l'étape v_j.

Soit $I = \cup_j [u_j, v_j]$: si $a < b$ sont deux étapes de I, la position de la tête à l'étape b

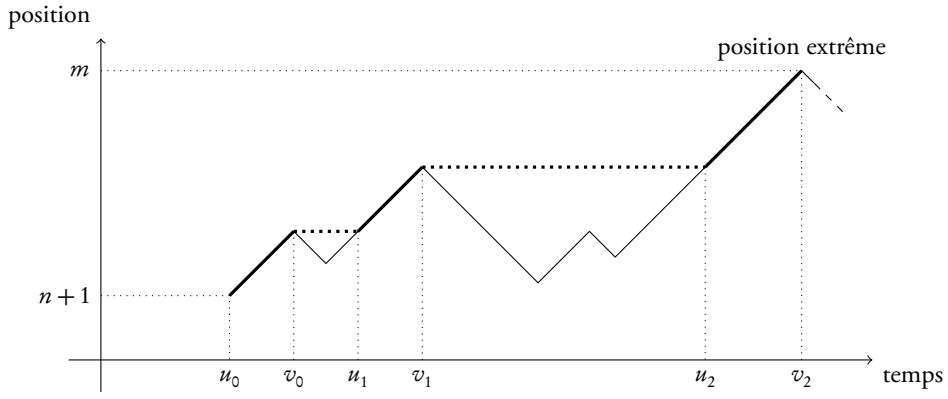


FIGURE 4.2 – Intervalles où N se déplace de manière « croissante » sur le ruban d'entrée.

est plus à droite que celle à l'étape a. La taille de I est au moins $1 + 2^{\gamma s(n)}$ puisqu'on doit atteindre la position m en partant de la position $n+1$. Puisque le nombre de configurations partielles est majoré par $2^{\gamma s(n)}$, il existe deux étapes $a < b$ dans I pour lesquelles N a la même configuration partielle, et où les positions de la tête sur le ruban d'entrée sont respectivement $p_a < p_b$. Entre ces deux étapes, la tête sur le ruban d'entrée ne visite que des cases vides (par définition de i_0) ; puisque la configuration partielle est la même, de l'étape b on peut répéter exactement le comportement de la machine pour aller de la case p_b à la case $p_b + (p_b - p_a)$ (en ne visitant que des cases vides), et ainsi de suite, c'est-à-dire qu'il existe une exécution allant à droite à l'infini, ce qui contredit l'hypothèse que toute exécution doit s'arrêter.

On en conclut que $m \leqslant n+1+2^{\gamma s(n)}$. Le même raisonnement vers la gauche montrerait que la position de la case extrême visitée à gauche vérifie $m' \geqslant -2^{\gamma s(n)}$. Donc le nombre de positions possibles pour la tête de lecture du ruban d'entrée est majoré par $n + 2 + 2^{1+\gamma s(n)}$. Puisque le nombre de configurations de travail est majoré par le nombre de configurations partielles multiplié par le nombre de positions possibles sur le ruban d'entrée, on en déduit le résultat. □

Notre première application de cette méthode est la démonstration de la proposition suivante comparant complexité en temps et en espace.

4-W **Proposition**

1. Pour toute fonction $t(n)$, $\mathsf{NTIME}(t(n)) \subseteq \mathsf{DSPACE}(t(n))$;
2. Pour toute fonction $s(n)$, $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DTIME}(2^{O(s(n))})$.

Idée de la démonstration Pour 1, en espace $t(n)$ on peut simplement énumérer tous les chemins possibles et simuler la machine fonctionnant en temps $t(n)$.

Pour 2, le nombre de configurations différentes d'une machine M fonctionnant en espace $s(n)$ ne dépasse pas $2^{O(s(n))}$ puisque les rubans de travail ne peuvent pas contenir des mots de taille supérieure à $s(n)$ (lemme 4-V). Si M prenait un temps $2^{\omega(s(n))}$, elle passerait alors deux fois par la même configuration et bouclerait infiniment.

Démonstration Montrons d'abord 1. On suppose pour simplifier que $t(n)$ est constructible en espace, mais on pourrait se passer de cette hypothèse (voir l'exercice 4-X).

Soit N une machine non déterministe fonctionnant en temps $\alpha t(n)$ (pour une certaine constante $\alpha > 0$) et décidant un langage A. Soit R un majorant du nombre de transitions possibles à chaque étape (R dépend seulement de la fonction de transition de N). Voici une machine M pour A fonctionnant en espace $O(t(n))$.

- Calculer $t(n)$;
- `pour chaque` $y \in \{0,\dots,R-1\}^{\alpha t(n)}$ `faire`
  - exécuter $N(x)$ le long du chemin y,
  - `accepter` si ce calcul accepte ;
- `rejeter`.

Puisque $t(n)$ est supposé constructible en espace, pour le calculer il suffit d'un espace $O(t(n))$. Pour énumérer tous les mots $y \in \{0,\dots,R-1\}^{\alpha t(n)}$, il suffit également d'un espace $O(t(n))$: on commence par $y = 0^{\alpha t(n)}$, puis à chaque itération on incrémente y *en place* en supposant qu'il représente un entier en base R, de la même façon qu'à l'exemple 4-F en base 2. Enfin, l'exécution de $N(x)$ le long du chemin y prend encore un espace $O(t(n))$ puisque le temps d'exécution est lui-même $O(t(n))$. Noter qu'à chaque nouvelle itération il faut réutiliser l'espace pris aux itérations précédentes et qui ne sert plus.

Pour montrer 2, on utilise la technique du graphe des configurations de la définition 4-S. Soit N une machine non déterministe fonctionnant en espace $O(s(n))$ pour décider un langage A. Par le lemme 4-V, il existe une constante α telle que le graphe des configurations G_x de $N(x)$ soit de taille majorée par $2^{\alpha s(|x|)}$. Afin de savoir si $x \in A$, il suffit de décider s'il existe un chemin dans G_x de la configuration initiale à une configuration acceptante. Pour cela, on peut utiliser un algorithme de parcours (par exemple en largeur) qui prend un temps polynomial en la taille du graphe. Chaque sommet (correspondant à une configuration) est représenté par l'état de la machine, le contenu des rubans de travail et la position des têtes sur les rubans d'entrée et de

travail. Pour implémenter un tel algorithme, il suffit d'être capable d'énumérer les voisins d'un sommet, ce qui se fait aisément en simulant une étape de N le long de toutes les transitions possibles.
Puisque le graphe est de taille $2^{O(s(n))}$ et que le parcours prend un temps polynomial en la taille du graphe, on obtient un algorithme déterministe fonctionnant en temps $2^{O(s(n))}$. □

✎ **4-X Exercice**

On a supposé dans la preuve du point 1 de la proposition précédente que la fonction $t(n)$ était constructible en temps. Montrer que cette hypothèse n'est pas nécessaire.

Indication : énumérer y bit par bit et simuler $N(x)$ le long de y au fur et à mesure, jusqu'à un état terminal, puis effectuer du backtracking.

✎ **4-Y Exercice**

Le lemme 4-V montre que la tête sur le ruban de lecture d'une machine fonctionnant en espace $s(n)$ ne peut pas visiter des cases hors de l'intervalle $[-2^{O(s(n))}, 2^{O(s(n))}]$.

En utilisant ce résultat, montrer que les deux définitions de complexité en espace données à la définition 4-A et à la remarque 4-B sont équivalentes.

Indication : maintenir un compteur pour connaître la position de la tête sur le ruban d'entrée.

Outre la classe L vue précédemment (définition 4-E), on définit également deux autres classes usuelles.

4-Z Définition

- La classe PSPACE est l'ensemble des langages reconnus par une machine de Turing fonctionnant en espace polynomial, c'est-à-dire $\mathsf{PSPACE} = \bigcup_{k>0} \mathsf{DSPACE}(n^k)$.
- La classe NL est l'ensemble des langages reconnus par une machine de Turing *non déterministe* fonctionnant en espace logarithmique, c'est-à-dire $\mathsf{NL} = \mathsf{NSPACE}(\log n)$.

On pourrait ajouter la classe NPSPACE des langages reconnus par une machine non déterministe fonctionnant en espace polynomial. Cependant, nous ne la hissons pas jusque dans notre définition à cause du corollaire 4-AU que nous verrons ci-dessous.

De la proposition 4-W nous déduisons les inclusions suivantes concernant les classes vues jusqu'à présent :

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}.$$

Par ailleurs, nous savons par les théorèmes de hiérarchie que PSPACE $\neq$ NL (grâce au corollaire 4-AU), EXP $\neq$ P et NEXP $\neq$ NP. Outre ces séparations, savoir si les autres inclusions sont strictes est une question ouverte, notamment « P $\neq$ NL ? », « PSPACE $\neq$ P ? » et « EXP $\neq$ PSPACE ? ».

4.4 Complétude

C'est notamment l'existence de problèmes naturels complets qui légitime la définition de classes de complexité. Comme pour NP, nous allons ainsi voir des problèmes PSPACE-complet et NL-complet. Cela nous permettra aussi de mieux comprendre la puissance de ces deux classes.

4.4.1 Espace polynomial

Le problème qui nous intéresse ici est une généralisation de SAT. Dans SAT, il s'agit de savoir si une formule avec un quantificateur existentiel est vraie. Dans le problème QBF que nous allons présenter, la formule contiendra à la fois des quantificateurs existentiels et universels.

4-AA Définition

– Une formule booléenne quantifiée est une formule de la forme

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \varphi(x_1, \dots, x_n)$$

où Q_i est un quantificateur $\exists$ ou $\forall$, et $\varphi(x_1, \dots, x_n)$ une formule booléenne sans quantificateurs (cf. définition 3-Q) sur les variables $x_1, \dots, x_n$. On notera qu'une telle formule n'a pas de variable libre puisqu'elles sont toutes liées par un quantificateur.

– Le langage QBF (pour *Quantified Boolean Formula*) est l'ensemble des formules booléennes quantifiées qui sont vraies.

4-AB Exemple $\psi = \forall x \forall y \exists z (\neg x \vee \neg y \vee z) \wedge ((x \wedge y) \vee \neg z)$ est une formule booléenne quantifiée qui est vraie, donc $\psi \in$ QBF.

Il s'avère que ce problème est PSPACE-complet comme l'ont montré dès 1973 Stockmeyer et Meyer [SM73].

4-AC Théorème

Le problème QBF est PSPACE-complet.

Idée de la démonstration QBF est dans PSPACE grâce à un algorithme qui teste toutes les possibilités, en réutilisant l'espace à chaque nouveau cas.

Pour la complétude, il s'agit d'exprimer par une formule booléenne quantifiée de taille polynomiale le calcul d'une machine fonctionnant en espace polynomial. Pour cela, nous exprimons le fait qu'il existe un chemin dans le graphe des configurations de la configuration initiale à une configuration acceptante. Pour éviter d'obtenir une formule de taille exponentielle, l'astuce consiste à employer un quantificateur universel afin qu'une même variable puisse jouer deux rôles.

Démonstration Voici un algorithme récursif pour QBF.

QBF-rec(ψ) :

- si ψ n'a pas de quantificateur, accepter ssi ψ est vraie ;
- si $\psi = \exists x \psi'(x)$, renvoyer QBF-rec($\psi'(0)$) $\vee$ QBF-rec($\psi'(1)$) ;
- si $\psi = \forall x \psi'(x)$, renvoyer QBF-rec($\psi'(0)$) $\wedge$ QBF-rec($\psi'(1)$).

Pour mettre en œuvre cet algorithme, il faut *réutiliser* l'espace lorsque celui-ci est libéré. Ainsi, une fois l'évaluation QBF-rec($\psi'(0)$) terminée, on efface l'espace qui a été utilisé et on utilise cet espace pour l'évaluation de QBF-rec($\psi'(1)$).

L'espace utilisé dépend du nombre n de variables (égal au nombre de quantificateurs). Si $n = 0$ alors ψ n'est composée que de constantes et son évaluation requiert simplement de parcourir son arbre, soit un espace linéaire.

Si $n > 0$, il y a des appels récursifs. On appelle $E(\psi)$ l'espace utilisé par l'algorithme sur l'entrée ψ. Lorsqu'on effectue un appel récursif, il faut pouvoir revenir à l'état précédant l'appel pour continuer le calcul (dans la terminologie habituelle, il faut « empiler le contexte »). Il faut donc garder en mémoire la formule traitée juste avant l'appel. Ainsi, en réutilisant l'espace pour les deux appels récursifs successifs (l'un sur $\psi'(0)$ et l'autre sur $\psi'(1)$), on obtient $E(\psi) \leqslant \alpha|\psi| + \max(E(\psi'(0)), E(\psi'(1)))$ pour une certaine constante α.

Au final, en déroulant les n appels récursifs, on obtient $E(\psi) = O(|\psi|^2)$, donc QBF $\in$ PSPACE.

Montrons maintenant que QBF est PSPACE-difficile. Soit $A \in$ PSPACE, nous devons donner une réduction $A \leqslant_m^p$ QBF. Soit M une machine pour A fonctionnant en espace polynomial, et x une instance de A de taille n. L'idée est d'utiliser le graphe des configurations de $M(x)$ (définition 4-S). En effet, $M(x)$ accepte si et seulement s'il existe un chemin de la configuration initiale c_0 à une configuration acceptante c_{acc} dans ce graphe. Il s'agit alors de donner une formule ψ exprimant l'existence d'un tel chemin.

Pour cela, nous définissons une famille de formules $\psi_i(u, v)$ exprimant le fait qu'il existe un chemin de longueur $\leqslant 2^i$ entre deux sommets u et v du graphe. Rappelons que u et v sont des configurations, qui doivent donc être encodées par un nombre polynomial de variables représentant le contenu des rubans et les positions des têtes.

La première de ces formules, $\psi_0(u, v)$, affirme que $u = v$ ou que v est voisin de u. En d'autres termes, soit le contenu des rubans et les positions des têtes sont identiques,

soit il existe une transition qui mène de u à v. On a vu à la preuve du théorème 3-V de Cook-Levin comment exprimer cela par une formule booléenne de taille polynomiale $q(n)$.
Pour définir $\psi_{i+1}(u,v)$, on pourrait simplement dire $\exists w(\psi_i(u,w)\wedge\psi_i(w,v))$. Cependant, la taille de la formule doublerait et serait trop grande au final. Il faut ruser : nous définissons

$$\psi_{i+1}(u,v)=\exists w\forall x,y\ \big((x=u\wedge y=w)\vee(x=w\wedge y=v)\big)\to\psi_i(x,y)$$

où, rappelons-le, u,v,w,x et y désignent des configurations. Ainsi, $|\psi_{i+1}|\leqslant p(n)+|\psi_i|$, où $p(n)$ est un polynôme tel que la première partie de la formule ψ_{i+1} (les quantificateurs et les égalités des configurations) prenne une taille $\leqslant p(n)$. Cela implique que $|\psi_i|\leqslant i\,p(n)+|\psi_0|\leqslant i\,p(n)+q(n)$.
Par le lemme 4-V, le graphe de configurations de $M(x)$ a une taille $\leqslant 2^{q(n)}$ pour un certain polynôme $q(n)$. Ainsi, s'il existe un chemin entre deux configurations u et v, alors il existe un chemin de taille $\leqslant 2^{q(n)}$. La réduction de A à QBF associe donc à x la formule $\psi^x=\exists c\ (c \text{ est acceptante})\wedge(\psi_{q(n)}(c_0,c))$ qui exprime le fait qu'il existe un chemin de taille $\leqslant 2^{q(n)}$ entre la configuration initiale c_0 et une configuration acceptante. Cette formule est calculable en temps polynomial et on a $x\in A$ ssi $\psi^x\in$ QBF, donc $A\leqslant^{\mathsf{P}}_{\mathsf{m}}$ QBF □

4-AD **Remarque** La réduction qui précède s'applique aussi bien aux machines non déterministes, ce qui fait que QBF est NPSPACE-difficile. Puisqu'il est dans PSPACE, cela implique que PSPACE = NPSPACE. Nous montrerons un résultat plus général au théorème 4-AS.

Nous n'en verrons pas plus mais, de la même manière que pour la classe NP, il existe de nombreux autres problèmes naturels PSPACE-complet. Plusieurs d'entre eux portent notamment sur la recherche de stratégies gagnantes dans des jeux à deux joueurs. En effet, dans ces jeux on retrouve l'alternance de quantificateurs de QBF : il existe une stratégie gagnante pour l'un des joueurs s'il existe un choix 1 (le premier coup) tel que pour tout choix 2 (la réponse de l'adversaire), il existe un choix 3 tel que pour tout choix 4… jusqu'à arriver au coup gagnant.

4.4.2 Espace non déterministe logarithmique

Réductions en espace logarithmique

Puisque NL ⊆ P, il n'est pas pertinent d'étudier les problèmes NL-complets pour les réductions polynomiales (cf. proposition 3-L). Il nous faut des réductions plus faibles. Pour comparer des problèmes en fonction de leur complexité en espace, il est naturel de définir des réductions many-one en espace logarithmique.

4-AE Définition

Une *réduction many-one* en espace logarithmique d'un problème B (sur l'alphabet Σ_B) à un problème A (sur l'alphabet Σ_A) est une fonction $f : \Sigma_B^\star \to \Sigma_A^\star$ calculable en espace logarithmique telle que :

$$\forall x \in \Sigma_B^\star, \quad x \in B \iff f(x) \in A.$$

Si une telle fonction f existe, on dira que B *se réduit à* A (via f) en espace logarithmique et on notera $B \leqslant_{\mathsf{m}}^{\mathsf{L}} A$.

De la même manière que précédemment (cf. définition 3-J), on a bien sûr aussi les notions de $\mathscr{C}$-difficulté et $\mathscr{C}$-complétude pour les réductions en espace logarithmique.

4-AF Remarque Si $B \leqslant_{\mathsf{m}}^{\mathsf{L}} A$ alors $B \leqslant_{\mathsf{m}}^{\mathsf{P}} A$.

✎ **4-AG Exercice**

Vérifier que tous les langages complets que nous avons vus jusqu'à présents restent complets pour les réductions en espace logarithmique. On peut par exemple vérifier que la réduction d'un problème NP à SAT donnée au théorème 3-V se calcule en espace logarithmique. De même pour la réduction de SAT à 3SAT (proposition 3-Z), etc.

Complétude pour NL

Pour préparer la preuve de complétude pour NL, nous devons tout d'abord définir une variante du graphe des configurations (définition 4-S), où l'on considère toutes les configurations potentielles, y compris celles qui ne sont pas accessibles depuis la configuration initiale.

4-AH Définition (graphe des configurations potentielles)

Soit N une machine non déterministe et x une entrée. On appelle *graphe des configurations potentielles de* $N(x)$ *en espace* $s(n)$ le graphe :

- dont les sommets sont toutes les configurations de travail possibles qui utilisent un espace $\leqslant s(|x|)$ et dont la position de la tête sur le ruban d'entrée est comprise entre $-2^{s(|x|)}$ et $2^{s(|x|)}$;
- il y a un arc de la configuration c à la configuration c' si N peut aller de c à c' en une étape.

4-AI Remarques

– La différence avec le graphe des configurations de la définition 4-S est que toutes les configurations sont prises en compte, et pas seulement celles accessibles depuis la configuration initiale.

– Par un dénombrement simple similaire à la première partie de celui de la preuve du lemme 4-V, le nombre de sommets de ce graphe est majoré par $2^{\alpha s(n)}$ pour une constante α.

L'intérêt de ce graphe vient du fait qu'il est simple à construire.

4-AJ Lemme

Soit $s(n)$ une fonction constructible en espace. Soit N une machine non déterministe fonctionnant en espace $\leqslant s(n)$ et α une constante. Alors il existe une constante γ telle que le graphe des configurations potentielles de $N(x)$ en espace $\alpha s(n)$ est constructible en espace déterministe $\leqslant \gamma s(n)$.

Démonstration Soit G_x le graphe des configurations potentielles de $N(x)$ en espace $\alpha s(n)$. Il s'agit de construire la matrice d'adjacence de G_x. Soit m le nombre des configurations potentielles en espace $\alpha s(n)$: il existe une constante β telle que $m \leqslant 2^{\beta s(n)}$ (cf. remarque 4-AI). Le codage de chaque configuration potentielle est simplement le contenu des rubans de travail, l'état ainsi que la position des têtes. Cela correspond à un numéro en binaire entre 0 et $m-1$.

On remplit la matrice d'adjacence de G_x coefficient par coefficient. Pour le coefficient (i,j), on se place à la bonne position sur le ruban de sortie, on simule une étape de calcul de N à partir de la configuration i pour toutes les transitions possibles. Si la configuration j est ainsi obtenue à partir de i alors on inscrit 1 sur le ruban de sortie, sinon on inscrit 0.

L'espace requis par ce procédé est majoré par $\gamma s(n)$ pour une certaine constante γ : en effet, d'abord on calcule $s(n)$ en espace $O(s(n))$; puis pour se déplacer sur le ruban de sortie on utilise un compteur en binaire dont la valeur n'excède pas $2^{\beta s(n)}$; l'écriture de i et j nécessite également $\beta s(n)$ bits, et enfin la simulation d'une étape de N à partir de i utilise un espace supplémentaire $\alpha s(n)$. □

Sans surprise après tout ce qui précède, le problème NL-complet que nous allons voir se rapporte à l'accessibilité dans un graphe orienté G pour lequel on note S l'ensemble des sommets et A l'ensemble des arcs. Nous l'avons déjà rencontré à la section 2.1.3.

4-AK Définition

Le problème ACCESSIBILITÉ est défini comme suit :

– *entrée* : un graphe orienté $G = (S, A)$ et deux sommets $s, t \in S$;
– *question* : existe-t-il un chemin de s à t dans G ?

4-AL Proposition

Le langage ACCESSIBILITÉ est NL-complet pour les réductions en espace logarithmique.

Idée de la démonstration Pour décider ACCESSIBILITÉ en espace non déterministe logarithmique, il suffit de se diriger dans le graphe en partant de s et en devinant à chaque étape le voisin suivant jusqu'à trouver t. Si t n'a pas été trouvé après $|S|$ étapes, on rejette.

La NL-difficulté vient du fait que décider si une machine non déterministe en espace logarithmique N accepte revient à trouver un chemin de la configuration initiale à une configuration acceptante dans le graphe des configurations de N.

Démonstration Voici un algorithme pour ACCESSIBILITÉ sur l'entrée (G, s, t) :

– $u \leftarrow s$;
– `Pour` i allant de 0 à $|S|-1$ `faire`
  ◦ si $u = t$ `accepter`,
  ◦ choisir de manière non déterministe un voisin sortant v de u,
  ◦ $u \leftarrow v$;
– `rejeter`.

S'il existe un chemin de s à t, alors il existe un chemin *simple* de s à t (c'est-à-dire sans boucle) et celui-ci est de taille $< |S|$ (le nombre de sommets). Donc il existera une suite de choix non déterministes dans l'algorithme arrivant à t en au plus $|S|-1$ étapes, c'est-à-dire que l'algorithme acceptera. Si par contre t n'est pas accessible depuis s, alors la boucle se terminera sans avoir trouvé t et tous les chemins rejetteront. Cet algorithme non déterministe décide donc ACCESSIBILITÉ.

Cet algorithme nécessite de conserver en mémoire le compteur i variant de 0 à $|S|-1$: en binaire, il prend $1+\log(|S|-1)$ cases. Il faut également écrire les deux sommets u et v : on les désigne par leur numéro en binaire, ce qui prend encore un espace $1+\log|S|$. Une fois v deviné, vérifier qu'il s'agit d'un voisin sortant de u se fait en parcourant l'entrée, de même que vérifier si $v = t$. Au final, cet algorithme non déterministe fonctionne en espace logarithmique, donc ACCESSIBILITÉ $\in$ NL.

Pour la NL-difficulté, nous allons réduire tout problème de NL à ACCESSIBILITÉ. Soit $A \in$ NL et N une machine non déterministe pour A fonctionnant en espace $\leqslant \alpha \log n$,

c'est-à-dire que $x \in A$ ssi $N(x)$ a un chemin acceptant. Pour une entrée x, on note G_x le graphe des configurations potentielles (définition 4-AH) de $N(x)$ en espace $\alpha\gamma \log n$, où γ est la constante du lemme 4-V concernant le déplacement de la tête sur le ruban d'entrée. À partir de G_x on définit un nouveau graphe G'_x en ajoutant un sommet t et en reliant toutes les configurations acceptantes de G_x à t. Pour s, on prend la configuration initiale. Ainsi, $N(x)$ accepte ssi t est accessible depuis s dans G'_x. Notre réduction f de A à ACCESSIBILITÉ associe (G'_x, s, t) à x. Par ce qui précède, $x \in A$ ssi $f(x) \in$ ACCESSIBILITÉ.

Le lemme 4-AJ montre que G_x est calculable en espace logarithmique. Pour en déduire G'_x, il suffit de relier toutes les configurations acceptantes à t, ce qui nécessite simplement de parcourir chaque sommet et se fait en espace logarithmique également. Ainsi, f est calculable en espace déterministe logarithmique, ce qui conclut. □

4-AM **Remarque** Quant à lui, le problème $\text{ACCESS}_{\text{n.o.}}$ de l'accessibilité dans un graphe *non orienté,* que nous rencontrerons au chapitre 12, n'est probablement pas NL-complet car ce problème est dans la classe L grâce un algorithme sophistiqué (cf. théorème 12-C).

4.5 Le rôle du non-déterminisme

On connaît beaucoup plus de liens entre les classes déterministes et non déterministes en espace que pour leurs analogues en temps. Cela provient toujours de la technique du graphe des configurations. Le non-déterminisme semble apporter moins de puissance aux classes en espace comme nous allons le voir. Nous commençons par donner une caractérisation de NL en termes de certificats comme pour NP.

4.5.1 Certificats unidirectionnels

On pourrait penser que la caractérisation de NP grâce aux certificats (proposition 2-AO) se transpose aisément à NL en imposant que la vérification soit faite en espace logarithmique. Il n'en est rien comme le montre la proposition suivante : si l'on n'y prend pas garde, on se retrouve avec toute la puissance de NP. Nous donnerons simplement l'idée de la démonstration.

4-AN **Proposition**

Soit $A \in$ NP. Il existe un langage $B \in$ L et un polynôme $p(n)$ tel que

$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)} (x, y) \in B.$$

Idée de la démonstration Puisque 3SAT est NP-complet pour les réductions en espace logarithmique (exercice 4-AG), il suffit de montrer qu'un certificat pour 3SAT peut être vérifié en espace logarithmique. Comme certificat pour une formule $\varphi(x_1,\dots,x_n)$, on donne la valuation $(a_1,\dots,a_n)$ des variables censées satisfaire φ. Pour vérifier qu'il s'agit bien d'une solution, il suffit d'évaluer $\varphi(a_1,\dots,a_n)$. Pour cela, on parcourt la formule φ et on vérifie que chaque clause est satisfaite, ce qui prend un espace logarithmique. □

Pour vraiment capturer la classe NL en termes de certificats, il faut imposer une restriction sur la vérification : le certificat ne peut être lu que de gauche à droite.

4-AO Définition

- Un *vérificateur unidirectionnel* est une machine de Turing V qui possède un ruban d'entrée supplémentaire sur lequel la tête ne peut pas se déplacer vers la gauche. En d'autres termes, les deux seuls mouvements possibles sont de rester sur place ou d'aller d'une case vers la droite.
- Le vérificateur V prend donc deux entrées x (lisible vers la gauche et vers la droite) et y (lisible seulement vers la droite). Le mot y sera appelé *certificat unidirectionnel*.
- On notera $V(x,y)$ le calcul de V sur l'entrée x et le certificat unidirectionnel y.

Cette notion nous permet maintenant de capturer NL.

4-AP Proposition

Un langage A est dans NL si et seulement s'il existe un vérificateur unidirectionnel V fonctionnant en *espace logarithmique* et un polynôme $p(n)$ tels que

$$x \in A \iff \exists y \in \{0,1\}^{p(|x|)}\ V(x,y) = 1.$$

Idée de la démonstration S'il existe un vérificateur unidirectionnel, alors dans NL il suffit de le simuler en devinant le nouveau bit du certificat à chaque déplacement à droite de la tête.

Dans l'autre sens, si $A \in \mathsf{NL}$ via une machine N, alors il suffit de donner comme certificat un chemin acceptant du calcul $N(x)$: pour simuler N le long de ce chemin, il suffit de lire le certificat une seule fois de gauche à droite.

Démonstration Commençons par l'implication de la droite vers la gauche. Si V est un vérificateur unidirectionnel pour A et $p(n)$ la taille de y, alors voici un algorithme non déterministe pour A, sur l'entrée x :

– simuler $V(x,.)$ en devinant le nouveau bit du certificat y à chaque déplacement à droite de la tête sur le second ruban d'entrée, tant que $|y| \leqslant p(|x|)$ (le bit précédent peut être oublié) ;
– `accepter` ssi V accepte.

Cet algorithme possède un chemin acceptant si et seulement s'il existe $y \in \{0,1\}^{p(|x|)}$ tel que $V(x,y)=1$, c'est-à-dire ssi $x \in A$. De plus, il fonctionne en espace logarithmique puisque V fonctionne en espace logarithmique. Ainsi, $A \in \mathsf{NL}$.

Pour l'implication de la gauche vers la droite, soit $A \in \mathsf{NL}$ reconnu par une machine non déterministe N fonctionnant en espace logarithmique. Un chemin de calcul de $N(x)$ est déterminé par les choix non déterministes effectués dans l'arbre, que l'on code par un mot $y \in \{0,1\}^*$. Puisque le temps de calcul de $N(x)$ est polynomial (proposition 4-W), y est de taille polynomiale $p(|x|)$. Le vérificateur V prend alors deux entrées x et y et simule $N(x)$ le long du chemin y. Pour ce faire, il n'est nécessaire de lire y que de la gauche vers la droite sans revenir sur ses pas. C'est un calcul déterministe qui prend un espace logarithmique, d'où le résultat. □

✎ **4-AQ** **Exercice**

Montrer que NP admet des vérificateurs unidirectionnels fonctionnant en temps polynomial.

Indication : répéter plusieurs fois le certificat usuel du langage NP.

4.5.2 Théorème de Savitch

Le théorème de Savitch [Sav70] permet de relier espace déterministe et non déterministe de manière beaucoup plus fine que pour le temps. Nous donnons d'abord l'algorithme déterministe pour `ACCESSIBILITÉ` (définition 4-AK) qui est au cœur du théorème de Savitch.

4-AR **Lemme**

`ACCESSIBILITÉ` $\in \mathsf{DSPACE}((\log n)^2)$.

Idée de la démonstration Si n est le nombre de sommets du graphe G, il suffit de décider s'il existe un chemin de taille n de s à t.

Pour décider s'il existe un chemin de taille $\leqslant 2^i$ d'un sommet u à un sommet v, on teste tous les sommets w et on vérifie récursivement s'il existe un chemin de taille $\leqslant 2^{i-1}$ de u à w puis de w à v. Cela nécessite i appels récursifs imbriqués, chacun prenant un espace $O(\log n)$, c'est-à-dire que l'algorithme déterministe utilise au total, pour $i = \log n$, un espace $O((\log n)^2)$.

Démonstration Soit Accès(G,u,v,i) l'algorithme récursif suivant afin de décider s'il existe un chemin de taille $\leqslant 2^i$ de u à v dans le graphe G.

> Accès(G,u,v,i) :
> – si $i=0$
> ◦ `accepter` si $u=v$ ou s'il y un arc de u à v,
> ◦ sinon `rejeter` ;
> – `pour tout` sommet w `faire`
> ◦ si Accès$(G,u,w,i-1)$ et Accès$(G,w,v,i-1)$ alors `accepter` ;
> – `rejeter`.

Il est immédiat de montrer par récurrence que Accès(G,u,v,i) accepte si et seulement s'il existe un chemin de u à v de taille $\leqslant 2^i$ dans G.

Par ailleurs, s'il existe un chemin de u à v dans G, alors il existe un chemin *simple* (sans boucle) de u à v, et la longueur de celui-ci n'excède pas le nombre de sommets n. On en déduit que l'appel Accès$(G,s,t,\lceil \log n \rceil)$ résout le problème d'accessibilité.

Il reste à montrer que cet algorithme fonctionne en espace $(\log n)^2$. Soit $C(i)$ l'espace utilisé par un appel Accès(G,u,v,i). Alors $C(0) \leqslant \alpha \log n$ (pour une certaine constante $\alpha > 0$) car il suffit d'aller lire le coefficient (u,v) de la matrice d'adjacence, ce qui se fait en comptant jusqu'à u et v avec des compteurs binaires.

Lors du calcul Accès$(G,u,v,i+1)$, on parcourt tous les sommets w (ce qui prend un espace $\leqslant \alpha \log n$) puis on appelle Accès(G,u,w,i) (ce qui prend $C(i)$), on efface l'espace de calcul et on appelle Accès(G,w,v,i) (qui utilise le même espace précédemment effacé). En tout on a donc $C(i+1) \leqslant C(i)+\alpha \log n$. Ainsi, $C(\log n) = O((\log n)^2)$, ce qui conclut. □

Nous en déduisons le théorème de Savitch.

4-AS **Théorème (Savitch, 1970)**

Pour toute fonction $s(n) \geqslant \log n$ constructible en espace,

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2).$$

Idée de la démonstration Si N est une machine pour $A \in \mathsf{NSPACE}(s(n))$, son graphe des configurations est de taille $2^{O(s(n))}$. Il suffit de décider s'il existe un chemin de taille $2^{O(s(n))}$ entre la configuration initiale et une configuration acceptante. On utilise pour cela l'algorithme du lemme 4-AR.

Démonstration Soit $A \in \mathsf{NSPACE}(s(n))$ décidé par une machine non déterministe N fonctionnant en espace $\leqslant \alpha s(n)$, et soit G_x le graphe des configurations potentielles

de $N(x)$ en espace $\alpha s(n)$ (définition 4-AH). À partir de G_x on définit un nouveau graphe G'_x en ajoutant un sommet t et en reliant toutes les configurations acceptantes de G_x à t.
Voici un algorithme déterministe pour A sur l'entrée x :

> – s'il existe un chemin de c_0 (configuration initiale) à t dans G'_x, `accepter` ;
> – sinon `rejeter`.

On utilise pour cela l'algorithme du lemme 4-AR. L'algorithme fonctionne en espace $O((\log|G'_x|)^2)$ et le graphe G'_x est de taille $2^{O(s(n))}$ et peut être calculé en espace $O(s(n))$ (lemme 4-AJ). Donc appliquer l'algorithme au graphe G'_x calculé à la volée prend un espace $O(s(n)^2)$ par la proposition 4-J, ce qui conclut la preuve. □

4-AT Corollaire

$\mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2 n)$.

Un autre corollaire de ce théorème est que le non-déterminisme n'apporte pas de puissance aux classes $\mathsf{DSPACE}(\mathscr{F})$ où $\mathscr{F}$ est un ensemble de fonctions clos par carré ($f \in \mathscr{F} \implies f^2 \in \mathscr{F}$). En particulier, nous retrouvons le résultat de la remarque 4-AD.

4-AU Corollaire

$\mathsf{NPSPACE} = \mathsf{PSPACE}$

La situation est donc très différente de la complexité en temps, où même pour de larges bornes en temps, on suppose que le non-déterminisme augmente la puissance de calcul (questions « $\mathsf{NP} \neq \mathsf{P}$? » et « $\mathsf{NEXP} \neq \mathsf{EXP}$? »).

4.5.3 Théorème d'Immerman-Szelepcsényi

Nous allons voir encore une fois une propriété du non-déterminisme pour l'espace qui semble contraire aux classes en temps : les classe NSPACE sont closes par complément. Après ce qu'on vient de voir, c'est évident pour NPSPACE bien sûr (puisque ce n'est autre que PSPACE), mais c'est beaucoup plus surprenant pour NL.

Le problème de l'inaccessibilité dans un graphe

La classe coNL des langages dont le complémentaire est dans NL (cf. définition 2-AX) peut également être caractérisée en termes de machines non déterministes comme coNP (proposition 2-AY).

4-AV **Proposition** (coNL)

La classe coNL est l'ensemble des langages A tels qu'il existe une machine de Turing non déterministe N fonctionnant en espace logarithmique et satisfaisant :

$$x \in A \text{ ssi tous les chemins de calcul de } N(x) \text{ sont acceptants.}$$

Le résultat de clôture par complément montré indépendamment par Immerman et Szelepcsényi découle de deux astucieux lemmes ci-dessous qui permettent de décider dans NL le complémentaire du problème NL-complet ACCESSIBILITÉ (proposition 4-AL). On notera donc coACCESSIBILITÉ le problème suivant :

- *entrée* : un graphe orienté G et deux sommets s et t ;
- *question* : t est-il inaccessible depuis s dans G ?

Par le même raisonnement qu'au lemme 3-AI, ce problème est coNL-complet.

Compter les sommets accessibles

L'idée est d'utiliser le nombre de sommets accessibles depuis s pour vérifier que nos calculs non déterministes ont atteints tous les sommets voulus. Le premier lemme et son corollaire montrent qu'on peut compter « dans NL » le nombre de sommets de G accessibles depuis un sommet s.

4-AW **Lemme**

Si G est un graphe orienté et s un sommet, on note S_i l'ensemble des sommets de G à distance $\leqslant i$ de s.

Il existe une machine de Turing non déterministe N fonctionnant en espace logarithmique sur l'entrée $(G, s, i, |S_i|)$, telle que :

- $N(G, s, i, |S_i|)$ possède au moins un chemin acceptant ;
- $|S_{i+1}|$ est inscrit en binaire sur le ruban de sortie à la fin de tout chemin acceptant.

On n'exige rien du résultat si l'entrée donne une valeur erronnée de $|S_i|$.

Idée de la démonstration On parcourt tous les sommets et pour chaque sommet on teste s'il appartient à S_{i+1}. Ce test est bien sûr la partie astucieuse. À la fin du processus, un compteur nous donnera la valeur de $|S_{i+1}|$. Voici comment tester si un sommet u appartient à S_{i+1}.

Il s'agit de savoir s'il est dans S_i ou s'il est voisin de S_i. On parcourt tous les sommets et pour chaque sommet v on teste s'il appartient à S_i en se déplaçant de manière non déterministe de s à v pendant i étapes. Il se peut bien sûr que v soit accessible mais que les choix non déterministes ne trouvent pas ce chemin : pour pouvoir repérer plus

tard cette éventualité (et ignorer le calcul correspondant), on incrémente un compteur c si on a atteint v. Pendant le processus, on a vérifié si u était dans S_i ou s'il était voisin de l'un des sommets de S_i.
Si tous les choix non déterministes ont été bons, on a trouvé tous les sommets de S_i, donc $c = |S_i|$. Si ce n'est pas le cas, on rejette, sinon on a bien testé si $u \in S_{i+1}$. L'espace est resté logarithmique.

Démonstration On aura besoin d'un test non déterministe en espace logarithmique qui admet un calcul acceptant ssi il existe un chemin de longueur $\leqslant i$ de s à un sommet u (c'est-à-dire ssi $u \in S_i$). Par manque d'espace, on ne peut pas deviner le chemin complet. Il nous faut progresser dans le graphe pas à pas en devinant un nouveau sommet à chaque étape et en oubliant les précédents. Voici le test pour $i \geqslant 1$.

Test1(s, u, i) :
- $w \leftarrow s$ (se souvenir du dernier sommet) ;
- `pour` d de 1 à i `faire`
  - `deviner` un sommet x,
  - `rejeter` si $x \neq w$ et x n'est pas voisin de w,
  - `accepter` si $x = u$,
  - $w \leftarrow x$;
- `rejeter`.

L'espace utilisé par ce test est celui nécessaire pour stocker d, w et x, c'est-à-dire logarithmique.
Lorsque $u \notin S_i$, le test précédent n'a que des calculs rejetant. Mais au contraire nous voulons un test qui possède toujours au moins un chemin acceptant, et que ceux-ci renvoient 1 lorsque $u \in S_i$ et 0 lorsque $u \notin S_i$. L'astuce pour réaliser un tel test est d'utiliser une information supplémentaire, la taille de S_i, avec un compteur pour vérifier que l'on a bien atteint tous les sommets de S_i. Voici donc le nouveau test.

Test2$(s, u, i, |S_i|)$:
- $c \leftarrow 0$ (compteur pour le nombre de sommets atteints) ;
- $b \leftarrow 0$ (booléen pour savoir si u a été atteint) ;
- `pour tout` sommet v `faire`
  - tester si $v \in S_i$ par l'algorithme Test1,
  - `si` le calcul est acceptant
    - $c \leftarrow c + 1$
    - `si` $v = u$ `alors` $b \leftarrow 1$;

- `si` $c = |S_i|$ `alors accepter` et `renvoyer` b ;
- `sinon rejeter`.

Il existe bien sûr des suites de choix non déterministes telles que pour tout $v \in S_i$, les tests « $v \in S_i$? » effectués par Test1 donnent tous un calcul acceptant. Le long de ces calculs, on incrémente c à chaque nouveau sommet $v \in S_i$, donc on accepte. Sur chacun de ces calculs acceptants, on renvoie la valeur de b, c'est-à-dire 1 si $u \in S_i$ et `0` sinon.
Les autres calculs ont raté au moins l'un des sommets $v \in S_i$, donc $c < |S_i|$ et ils rejettent. Ainsi, il existe des chemins acceptants, ils renvoient tous la même valeur et cette valeur est 1 ssi $u \in S_i$. L'espace utilisé est logarithmique car il suffit de maintenir un compteur, d'énumérer les sommets et d'utiliser le test précédent.
On peut enfin calculer $|S_{i+1}|$ en fonction de $|S_i|$ à l'aide d'un compteur k et de Test2.

- $k \leftarrow$ `0` (compteur pour le nombre de sommets atteints) ;
- `pour tout` sommet u `faire`
  - $b' \leftarrow$ `0` (booléen pour savoir si $u \in S_{i+1}$),
  - `pour tout` sommet v `faire`
    - tester si $v \in S_i$ par l'algorithme Test2
    - `si` le calcul de Test2 est acceptant et renvoie $b = 1$, et `si` u est égal à v ou est un voisin de v, `alors` $b' \leftarrow 1$,
  - $k \leftarrow k + b'$;
- `si` tous les calculs de Test2 ont accepté, `alors accepter` et renvoyer k ;
- `sinon rejeter`.

Si l'on a suivi un calcul acceptant pour tous les tests $v \in S_i$ par l'algorithme Test2, alors on a incrémenté k dès que u est un sommet de S_i ou un voisin, donc l'algorithme renvoie la bonne valeur $|S_{i+1}|$. L'espace utilisé est logarithmique puisqu'il suffit de maintenir le compteur k, d'énumérer les sommets u et v et d'utiliser Test2. □

4-AX Corollaire

Il existe une machine de Turing non déterministe N fonctionnant en espace logarithmique sur l'entrée (G, s), où G est un graphe orienté et s un sommet de G, telle que :

- $N(G, s)$ possède des chemins acceptants ;
- le nombre de sommets accessibles depuis s est inscrit en binaire sur le ruban de sortie à la fin de tout chemin acceptant.

Démonstration Il suffit de calculer successivement grâce à l'algorithme du lemme précédent, en partant de $|S_0| = 1$, les valeurs de $|S_1|$, …, $|S_n|$, où n est le nombre de sommets de G (car si un sommet est accessible, il l'est par un chemin de longueur $\leqslant n$). On a besoin de retenir seulement la valeur $|S_i|$ pour calculer $|S_{i+1}|$ donc l'espace utilisé reste logarithmique. □

Algorithme pour l'inaccessibilté

Ce comptage des sommets accessibles est un élément central dans l'algorithme du lemme suivant.

4-AY **Lemme**

$$\texttt{coACCESSIBILITÉ} \in \texttt{NL}.$$

Idée de la démonstration Pour décider si t n'est pas accessible depuis s dans G, on utilise l'algorithme du corollaire précédent pour compter le nombre k de sommets accessibles depuis s. Puis on énumère tous les sommets u et pour chacun on teste de manière non déterministe s'il est accessible par un chemin de taille $\leqslant n$. Pendant ce processus on a également compté le nombre de sommets accessibles en incrémentant un compteur c. À la fin du calcul, si $c = k$ alors on a bien visité tous les sommets accessibles donc on accepte ssi t n'a jamais été atteint depuis s.

Démonstration Pour résoudre coACCESSIBILITÉ, voici l'algorithme non déterministe que l'on utilise sur l'entrée (G, s, t) :

- $c \leftarrow 0$ (compteur pour le nombre de sommets atteints) ;
- $b \leftarrow 0$ (booléen pour savoir si t a été atteint) ;
- compter le nombre k de sommets accessibles depuis s grâce à l'algorithme du corollaire 4-AX (rejeter si le calcul n'est pas acceptant) ;
- `pour tout` sommet u `faire`
  - tester s'il existe un chemin de s à u de taille $\leqslant n$ en se déplaçant dans G de manière non déterministe comme dans la preuve du lemme 4-AW,
  - si le chemin a atteint u
    - $c \leftarrow c + 1$
    - si $u = t$ alors $b \leftarrow 1$,
- `si` $c = k$ et $b = 0$ `alors accepter` ;

```
  – sinon rejeter.
```

Un calcul ne peut être acceptant que si $c = k$, c'est-à-dire que tous les sommets accessibles depuis s ont été atteints. Il y a toujours au moins un tel calcul. Si t ne fait pas partie des sommets atteints, alors $b = 0$ et on accepte, ce qui correspond bien au cas où t n'est pas accessible depuis s. Si t a été atteint, alors il est accessible depuis s et $b = 1$ pour tous les calculs tels que $c = k$, donc tous les calculs rejettent. Au final, cet algorithme décide bien coACCESSIBILITÉ.

L'espace qu'il utilise est logarithmique puisqu'il suffit d'énumérer tous les sommets, de maintenir un compteur et d'exécuter des tests recquérant un espace logarithmique. Donc coACCESSIBILITÉ $\in$ NL. □

Nous obtenons en corollaire le théorème suivant dû à Szelepcsényi [Sze87] et indépendamment à Immerman [Imm88].

4-AZ **Théorème (Szelepcsényi 1987, Immerman 1988)**

Pour toute fonction $s(n) \geqslant \log n$ constructible en espace,

$$\mathsf{coNSPACE}(s(n)) = \mathsf{NSPACE}(s(n)).$$

En particulier, NL = coNL.

Démonstration Il s'agit simplement de faire du padding (cf. proposition 2-AU). Soit $A \in \mathsf{coNSPACE}(s(n))$ et A' sa version « paddée » :

$$A' = \{(x, 1^{2^{s(|x|)}}) \mid x \in A\}.$$

Si $n = |x|$, la taille de l'entrée de A' est maintenant $m = n + 2^{s(n)}$. Puisque $s(n) \geqslant \log n$, on a $2^{s(n)} \leqslant m \leqslant 2^{1+s(n)}$.

Pour décider A', on calcule $s(n)$ en unaire, on en déduit l'écriture binaire de $2^{s(n)}$, on vérifie que l'entrée est de la forme $(x, 1^{2^{s(n)}})$ (en parcourant les 1 et en décrémentant le compteur contenant $2^{s(n)}$), puis on lance l'algorithme coNSPACE pour A sur l'entrée x. Sur une entrée de taille m, cette procédure coNSPACE utilise un espace $O(s(n))$, c'est-à-dire $O(\log m)$.

Ainsi, $A' \in \mathsf{coNL}$. Par complétude de coACCESSIBILITÉ, $A' \leqslant^{\mathsf{L}}_{\mathsf{m}}$ coACCESSIBILITÉ. Donc par le lemme 4-AY, $A' \in \mathsf{NL}$, c'est-à-dire que A' est reconnu par une machine non déterministe N' fonctionnant en espace $O(\log m)$. On en déduit l'algorithme non déterministe suivant pour A sur l'entrée x. Simuler le comportement de $N'(x, 1^{2^{s(n)}})$ sans écrire $1^{2^{s(n)}}$ mais en maintenant sur un ruban de travail un compteur binaire c égal à la position de la tête du ruban de lecture de N'. Si $c \in [1, n]$ alors le symbole lu sur le ruban d'entrée est x_c, si $c \in [n+1, n+2^{s(n)}]$ alors le symbole lu est 1, sinon c'est le symbole blanc B. Cela nous permet donc de simuler N' sans avoir à écrire $1^{2^{s(n)}}$ qui prendrait trop d'espace. L'espace total utilisé par cet algorithme pour A est

la taille du compteur c plus l'espace utilisé par N', c'est-à-dire $O(\log m) = O(s(n))$. Donc $A \in \mathsf{NSPACE}(s(n))$. □

4.5.4 Les questions ouvertes

Grâce à la méthode du graphe des configurations, on vient de voir que l'on sait résoudre plus de questions concernant l'espace que le temps. Cependant, il reste bien sûr de nombreuses questions ouvertes. Nous en verrons deux importantes.

L'analogue de « $\mathsf{P} = \mathsf{NP}$? » pour l'espace est la question « $\mathsf{L} = \mathsf{NL}$? ». Bien que le non-déterminisme n'apporte pas de puissance lorsqu'on dispose d'un espace polynomial, nous n'en savons rien si l'on se restreint à un espace logarithmique. Par complétude du problème ACCESSIBILITÉ, montrer que $\mathsf{L} = \mathsf{NL}$ revient à donner un algorithme déterministe fonctionnant en espace logarithmique pour décider l'accessibilité dans un graphe orienté.

On a également du mal à comparer finement le temps et l'espace. Par exemple, la question « $\mathsf{P} = \mathsf{PSPACE}$? » est toujours ouverte. En d'autres termes, est-ce que l'espace polynomial est plus puissant que le temps polynomial ? Ou encore, y a-t-il un algorithme en temps polynomial pour QBF ? Puisque $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$, la séparation de P et NP implique la séparation de P et PSPACE. Si les deux séparations sont vraies comme le pensent la plupart des informaticiens, alors la question « $\mathsf{P} \neq \mathsf{PSPACE}$? » serait moins difficile que « $\mathsf{P} \neq \mathsf{NP}$? ».

Signalons enfin que les exercices B-A et B-B touchent aux classes en espace.

5 Uniformité et non-uniformité

Jusqu'à présent, pour résoudre un problème nous avons utilisé un unique algorithme permettant de traiter toutes les entrées. Mais pourquoi devrions-nous utiliser la même méthode pour additionner deux entiers de 57 bits que pour additionner deux entiers de 73 bits ? Ou encore, les circuits que contiennent nos processeurs peuvent par exemple être spécialisés dans l'addition de nombres de 64 bits, faut-il changer la conception du circuit lorsqu'on passe à des entiers de 128 bits ?

Lorsqu'on donne un unique algorithme pour toutes les entrées d'un problème, on parle « d'uniformité » : la même méthode convient pour toutes les tailles d'entrée. C'est une notion très pertinente pour résoudre les problèmes puisqu'une description finie permet de traiter une infinité de cas, et elle est de plus très naturelle. Cependant, il n'est pas moins naturel de considérer des circuits spécialisés traitant seulement une certaine taille d'entrée : par exemple un additionneur 64 bits. En quelque sorte, on donne alors une méthode spécifique pour chaque taille d'entrée. Dans ce cas, pour traiter toutes les entrées possibles, il faut donner une infinité de circuits (un circuit par taille d'entrée). C'est ce qu'on appelle la « non-uniformité ».

L'étude de l'uniformité et de la non-uniformité est un aspect très riche de la complexité. Nous verrons dans ce chapitre les bases de la non-uniformité qu'on utilisera par la suite, notamment au chapitre consacré aux bornes inférieures non uniformes.

Ce chapitre contient à nouveau des simulations de machines (notamment par des circuits). Nous serons moins précis qu'aux chapitres précédents et donnerons seulement l'idée des démonstrations.

5.1 Conseils

Les conseils sont un moyen commode et très général d'introduire de la non-uniformité dans nos classes de complexité.

5.1.1 Définition

La non-uniformité consiste à donner une information potentiellement différente pour chaque taille d'entrée. Une formalisation de ce concept repose sur la notion de « conseil » : on aide une machine à reconnaître un langage en lui donnant une information qui dépend de la taille de l'entrée. Cette définition vient de Karp et Lipton [KL82].

5-A Définition

Soit une fonction $f : \mathbb{N} \to \mathbb{N}$ et une classe de complexité $\mathscr{C}$. La classe $\mathscr{C}/f(n)$ est l'ensemble des langages A tels qu'il existe $B \in \mathscr{C}$ et une suite de mots $(a_n)_{n\in\mathbb{N}}$ satisfaisant :

- $a_n \in \{0,1\}^{f(n)}$;
- pour tout mot x, $x \in A$ ssi $(x, a_{|x|}) \in B$.

5-B Remarques

- Le conseil a_n utilisé dépend uniquement de la taille de x et doit donc être le même pour tous les mots de même taille.
- Le conseil fait partie de l'entrée du langage B donc sa taille est prise en compte pour la complexité. Ainsi, dans la classe $\mathsf{P}/2^{\epsilon n}$, l'algorithme doit être polynomial en $(n + 2^{\epsilon n})$, donc il dispose d'un temps $2^{O(n)}$. Ce qui peut être trompeur puisqu'on parle quand même de la classe P.

Un peu de recul

Il y a une différence fondamentale entre les classes uniformes et non uniformes : les premières sont dénombrables alors que les secondes ne le sont pas comme le montre la proposition suivante.

5-C Proposition

Pour toute fonction $f(n) > 0$, $\mathsf{L}/f(n)$ est non dénombrable.

Idée de la démonstration Il y a une quantité non dénombrable de suites de conseils possibles, ce qui permet de définir une quantité non dénombrable de langages.

Démonstration Soit $b = (b_n) \in \{0,1\}^{\mathbb{N}}$ une suite de bits quelconque. On définit le langage $A_b = \{x \in \{0,1\}^\star \mid b_{|x|} = 1\}$ des mots dont la longueur correspond à un bit 1 dans la suite b.

On définit la suite de conseils $a_n = b_n 0^{f(n)-1}$, c'est-à-dire le bit b_n suivi du nombre de zéros nécessaires pour donner une taille $f(n)$. Alors $A_b \in \mathsf{L}/f(n)$ car il suffit de donner la suite (a_n) comme conseil : en effet, si l'on définit $B = \{(x,y) \mid y_1 = 1\}$ (où y_1 désigne le premier bit de y), alors $x \in A_b$ ssi $(x, a_{|x|}) \in B$. Or B est clairement dans L.

Ainsi, $\mathsf{L}/f(n)$ contient tous les langages A_b possibles. Il y en a autant que de suites $(b_n) \in \{0,1\}^{\mathbb{N}}$, c'est-à-dire une quantité non dénombrable. □

5-D Corollaire

Pour toute fonction $f(n) > 0$ et toute classe $\mathscr{C}$ vérifiant $\mathsf{L} \subseteq \mathscr{C}$, la classe $\mathscr{C}/f(n)$ est non dénombrable.

Démonstration L'hypothèse implique que $\mathscr{C}/f(n) \supseteq \mathsf{L}/f(n)$ qui est non dénombrable par la proposition 5-C. □

5-E Remarque Puisque toutes les classes $\mathscr{C}$ que nous avons vues jusqu'à présent contiennent L, elles vérifient toutes que $\mathscr{C}/f(n)$ est non dénombrable. Ce résultat est vrai même pour des classes plus petites que L puisqu'il faut simplement être capable de décider si le premier bit du conseil est 1 ou non. Ainsi, toute classe non uniforme « raisonnable » est non dénombrable.

En revanche, les classes uniformes sont dénombrables : chaque langage est reconnu par une machine et il y a un nombre dénombrable de machines puisque leur code est un mot fini.

Ainsi, en particulier, lorsqu'une classe uniforme est incluse dans une classe non-uniforme, alors l'inclusion est stricte.

5-F Corollaire

Pour toute fonction $f(n) > 0$ et toute classe $\mathscr{C}$ contenant L, il y a un langage indécidable (c'est-à-dire qu'aucune machine de Turing ne le reconnaît) dans $\mathscr{C}/f(n)$.

5.1.2 Classes usuelles

Bien entendu, nous ne souhaitons pas toujours parler uniquement d'un conseil de taille $f(n)$ pour une fonction f fixée, mais plutôt de prendre la taille du conseil parmi un ensemble de fonctions : taille polynomiale, linéaire, logarithmique, etc.

5-G Définition

Soit $\mathscr{F}$ un ensemble de fonctions $f : \mathbb{N} \to \mathbb{N}$ et $\mathscr{C}$ une classe de complexité. Alors par définition,

$$\mathscr{C}/\mathscr{F} = \bigcup_{f \in \mathscr{F}} \mathscr{C}/f(n).$$

Certains ensembles de fonctions sont assez courants.

5-H Définition

On désigne par poly l'ensemble des fonctions polynomiales, c'est-à-dire

$$\mathsf{poly} = \{n \mapsto k + n^k \mid k \in \mathbb{N}\};$$

par lin l'ensemble des fonctions linéaires, c'est-à-dire

$$\mathsf{lin} = \{n \mapsto k(n+1) \mid k \in \mathbb{N}\};$$

et par log l'ensemble des fonctions logarithmiques, c'est-à-dire

$$\mathsf{log} = \{n \mapsto k\lfloor \log(n+2) \rfloor \mid k \in \mathbb{N}\}.$$

On considérera donc en particulier des classes de la forme $\mathscr{C}/\mathsf{poly}$, $\mathscr{C}/\mathsf{lin}$ et $\mathscr{C}/\mathsf{log}$.

5-I Remarque Les fonctions de poly, lin et log peuvent être aussi grandes que l'on veut en $n = 0$, c'est pour cela qu'on a utilisé $k + n^k$ plutôt que n^k, $k(n+1)$ plutôt que kn et $k\lfloor \log(n+2) \rfloor$ plutôt que $k\lfloor \log n \rfloor$.

Cela nous permet d'insister sur une classe centrale : P/poly.

5-J Définition

La classe P/poly est donc l'ensemble des langages reconnus en temps polynomial avec conseil de taille polynomiale.

5-K Remarque Cette notion de conseils n'est pas toujours adaptée et des versions différentes de non-uniformité ont été introduites, notamment pour des classes comme BPP que nous verrons au chapitre suivant.

Avec la définition ci-dessus, en effet, la machine probabiliste devrait avoir un comportement de type BPP quel que soit le conseil, alors que souvent seul le bon conseil nous intéresse.

5.1.3 Premiers résultats

Pour se familiariser avec le concept, nous commençons par quelques petits résultats.

Nous montrons tout d'abord qu'un conseil de taille exponentielle est beaucoup trop puissant.

5-L Proposition

Tout langage sur un alphabet Σ appartient à $\mathsf{P}/|\Sigma|^n$.

Idée de la démonstration Il suffit de donner en conseil les $|\Sigma|^n$ bits permettant de savoir si chacun des $|\Sigma|^n$ mots de taille n est dans le langage ou non.

Démonstration Soit A un langage quelconque sur un alphabet Σ. Pour n fixé, on ordonne les mots de Σ^n en les considérant comme des entiers en base $|\Sigma|$: $x_1 < x_2 < \cdots < x_{|\Sigma|^n}$. On définit alors le conseil a_n comme suit : $a_n \in \{0,1\}^{|\Sigma|^n}$ et le i-ème bit de a_n vaut 1 ssi $x_i \in A$.

Pour $x \in \Sigma^n$, on peut maintenant décider si $x \in A$ grâce au conseil a_n : il suffit d'aller lire le x-ème bit de a_n. Le langage B associé pour la définition 5-A est donc

$$\{(i,y) \mid y_i = 1\}, \text{ où } i \in \Sigma^n \text{ et } y \in \{0,1\}^{|\Sigma|^n} :$$

ce langage est clairement dans P (il faut remarquer que le temps de calcul est polynomial en $|i|+|y|$ qui est la taille de l'entrée de B, mais il est exponentiel en $|i|$ qui est la taille de l'entrée de A). Ainsi, $A \in \mathsf{P}/|\Sigma|^n$. □

Voici maintenant un résultat dans l'esprit du théorème d'Immerman-Szelepcsényi (4-AZ) mais beaucoup plus simple : disposer d'un conseil donnant le nombre d'éléments de taille n d'un langage permet de prendre le complémentaire d'un calcul non déterministe exponentiel.

5-M Proposition

$\mathsf{coNEXP} \subset \mathsf{NEXP}/\mathsf{lin}$.

Idée de la démonstration Pour décider un langage $A \in \mathsf{coNEXP}$ en temps non déterministe exponentiel, on donne comme conseil le nombre m de mots de ${}^cA$ de taille n. On peut alors deviner m mots $x_1, \ldots, x_m$, vérifier s'ils sont tous dans ${}^cA$, et accepter ssi x n'est pas dans la liste.

Démonstration Soit $A \in \mathsf{coNEXP}$ sur l'alphabet Σ et x un mot de taille n. Alors ${}^cA \in \mathsf{NEXP}$ donc il est reconnu par une machine non déterministe N_c fonctionnant en temps exponentiel. On définit le conseil $a_n \in \{0,1\}^\star$ comme le nombre $|({}^cA)^{=n}|$ codé en binaire (le nombre de mots de taille n de ${}^cA$) : puisque $|({}^cA)^{=n}| \leqslant |\Sigma|^n$, on a $|a_n| =$

$O(n)$. Voici un algorithme non déterministe pour A avec conseil a_n, sur l'entrée x de taille n :

- `pour` i de 1 à a_n `faire`
  - `deviner` $x_i \in \Sigma^n$,
  - `rejeter` si $x_i \in \{x_1, \ldots, x_{i-1}\}$ (les x_i doivent tous être différents),
  - simuler $N_c(x_i)$,
  - si le chemin simulé est rejetant, `rejeter` ;
- `accepter` ssi $x \notin \{x_1, \ldots, x_{a_n}\}$.

À la fin de la boucle pour, le calcul n'a pas rejeté seulement si on a deviné a_n mots distincts $x_1, \ldots, x_{a_n}$ et que l'on a suivi pour chacun un calcul acceptant dans N_c. En d'autres termes, $x_i \in {}^cA$ pour tout i. Puisque $a_n = |({}^cA)^{=n}|$, on a donc $({}^cA)^{=n} = \{x_1, \ldots, x_{a_n}\}$. Pour décider si $x \in A$, il suffit donc de tester si $x \notin \{x_1, \ldots, x_{a_n}\}$.

Tout ce calcul se fait en temps exponentiel : on effectue au plus $|\Sigma|^n$ itérations de boucle et à chaque itération on simule la machine N_c fonctionnant en temps exponentiel. Puisque le conseil est de taille linéaire, $A \in \mathsf{NEXP/lin}$. □

Nous avons donc vu qu'un conseil de taille exponentielle est bien trop puissant et qu'un conseil de taille linéaire permet déjà des choses non triviales (puisqu'il est probable que $\mathsf{NEXP} \neq \mathsf{coNEXP}$). Nous voyons maintenant qu'un conseil de taille logarithmique ne peut en revanche pas aider pour résoudre des problèmes NP en temps polynomial.

5-N Proposition

Si $\mathsf{NP} \subset \mathsf{P/log}$ alors $\mathsf{NP} = \mathsf{P}$.

Idée de la démonstration Si $A \in \mathsf{NP}$ et $\mathsf{NP} \subset \mathsf{P/log}$ alors le problème de trouver un certificat y pour un mot $x \in A$ est aussi dans $\mathsf{P/log}$. On énumère alors tous les conseils possibles de taille logarithmique et on teste si le certificat obtenu est correct pour x.

Démonstration Soit $A \in \mathsf{NP}$: pour tout mot x, $x \in A$ ssi $\exists y \in \{0,1\}^{p(|x|)} (x, y) \in B$, où $p(n)$ est un polynôme et $B \in \mathsf{P}$. On définit le langage

$$W = \{(x, y', 1^{p(|x|)-|y'|}) \mid \exists y \in \{0,1\}^{p(|x|)}, y \sqsupseteq y', (x, y) \in B\},$$

où y' est un préfixe d'un témoin y pour x. Nous allons nous servir de W pour effectuer une recherche préfixe d'un certificat y pour un mot x : les $p(|x|) - |y'|$ symboles 1 sont là pour être sûr d'utiliser toujours le même conseil lors de la recherche.

On a bien sûr $W \in \mathsf{NP}$, donc $W \in \mathsf{P/log}$ par hypothèse : il existe un langage $C \in \mathsf{P}$ et une famille (a_n) de conseils de taille $\alpha \log n$ tels que

$$(x, y', 1^{p(|x|)-|y'|}) \in W \iff ((x, y', 1^{p(|x|)-|y'|}), a_m) \in C,$$

où $m = |(x, y', 1^{p(|x|)-|y'|})|$. Nous en déduisons un algorithme avec conseil a_m pour trouver un certificat y pour x si un tel certificat existe, algorithme qu'on décrit avec un conseil a quelconque :

Cert(x, a) :
- $y' \leftarrow \epsilon$ (mot vide) ;
- `pour` i de 1 à $p(|x|)$ `faire`
  - si $((x, y'0, 1^{p(|x|)-|y'0|}), a) \in C$ alors $y' \leftarrow y'0$ sinon $y' \leftarrow y'1$;
- `renvoyer` y'.

Cet algorithme est polynomial puisque $C \in \mathsf{P}$ et que $p(n)$ est un polynôme. Si on lui fournit le bon conseil a_m, alors il effectue une recherche préfixe jusqu'à trouver le plus petit certificat y (s'il existe).
On en déduit un algorithme déterministe polynomial pour A, sur l'entrée x :

- `pour tout` $a \in \{0,1\}^{\alpha \log m}$ `faire`
  - soit $y = \text{Cert}(x, a)$,
  - `accepter` si $(x, y) \in B$;
- `rejeter`.

Cet algorithme fonctionne en temps polynomial puisqu'il y a m^α conseils de taille $\alpha \log m$ et que m est polynomial en $|x|$. Si $x \notin A$ alors il n'y a pas de certificat y et chaque test $(x, y) \in B$ est faux, donc on rejette. En revanche, si $x \in A$ alors il existe un certificat y. Celui-ci est trouvé par l'appel Cert(x, a) lorsque $a = a_m$ (le bon conseil) et dans ce cas le test $(x, y) \in B$ est vrai donc on accepte. Ainsi, $A \in \mathsf{P}$. □

5.2 Circuits booléens

L'une des raisons de l'importance des classes non uniformes, et notamment de P/poly, provient de leurs liens avec les circuits booléens.

5.2.1 Définition

Un circuit booléen se présente comme un circuit électronique avec des entrées, des portes d'opération et une ou plusieurs sorties. Il ne doit pas y avoir de boucle dans le circuit. On obtient ainsi la définition suivante.

5-O Définition

Un circuit booléen est un graphe orienté acyclique (et connexe) tel que :

- les sommets de degré entrant nul sont appelés *entrées* et sont étiquetés par le nom d'une variable x_i ou par la constante 0 ou 1 ;
- les sommets de degré entrant 1 sont appelés *portes de négation* et sont étiquetés par $\neg$;
- les autres sommets sont soit des *portes de conjonction*, soit des *portes de disjonction*, respectivement étiquetées par $\wedge$ et $\vee$;
- les sommets de degré sortant nul sont également appelés *sorties* ;
- les voisins entrants d'une porte sont appelés ses *arguments*.

Sauf mention contraire, nos circuits auront une *unique sortie* et les portes $\vee$ et $\wedge$ auront un *degré entrant égal à* 2.

Enfin, si C est un circuit, on appellera *sous-circuit* de C issu d'une porte γ le circuit dont la sortie est γ et dont les sommets sont les prédécesseurs de γ dans C.

5-P Remarque D'autres types de portes (par exemple des portes de majorité) pourront être ajoutés selon la classe que l'on souhaite obtenir.

On trouvera un exemple de circuit à la figure 5.1.

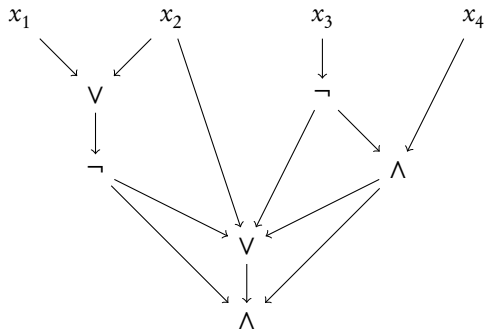


FIGURE 5.1 – Exemple de circuit de taille 10 et de profondeur 4, dont certaines portes sont de degré entrant 3 et 4.

Un circuit booléen calcule une fonction booléenne, définie de manière naturelle comme suit.

5-Q Définition

Soit C un circuit booléen sur les variables $x_1, \dots, x_n$. La fonction calculée par une porte γ de C est la fonction $f_\gamma : \{0,1\}^n \to \{0,1\}$ définie récursivement comme suit :

- si γ est une entrée étiquetée par une variable x_i, alors $f_\gamma(x_1, \dots, x_n) = x_i$;
- si γ est une entrée étiquetée par une constante $c = 0$ ou 1, alors $f_\gamma(x_1, \dots, x_n) = c$;
- si γ est une porte $\neg$ dont l'argument est la porte γ', alors $f_\gamma = \neg f_{\gamma'}$;
- si γ est une porte $\wedge$ (respectivement $\vee$) dont les arguments sont les portes $\gamma_1, \dots, \gamma_k$, alors $f_\gamma = \bigwedge_{i=1}^{k} f_{\gamma_i}$ (resp. $f_\gamma = \bigvee_{i=1}^{k} f_{\gamma_i}$).
  Si $k = 2$, on a donc $f_\gamma = f_{\gamma_1} \wedge f_{\gamma_2}$ (resp. $f_\gamma = f_{\gamma_1} \vee f_{\gamma_2}$).

Si C a k portes de sortie $s_1, \dots, s_k$, la fonction calculée par C est la fonction $f : \{0,1\}^n \to \{0,1\}^k$ dont la i-ème composante est la fonction calculée par s_i.
En particulier, dans le cas $k = 1$ où C a une unique porte de sortie γ, alors $f = f_\gamma$.

On doit maintenant définir deux caractéristiques importantes d'un circuit.

5-R Définition

Soit C un circuit booléen.

- La *taille* de C, notée $|C|$, est le nombre de sommets du graphe.
- La *profondeur* de C est la taille maximale d'un chemin d'une entrée à une sortie.

5-S Remarque On peut évidemment encoder un circuit sur l'alphabet $\{0,1\}$ en donnant par exemple la matrice d'adjacence du graphe et le type de chaque porte. Dans ce cas, si le circuit a n sommets, alors la taille de son codage sera $O(n^2)$. On notera C à la fois pour le circuit lui-même et pour son code, le contexte permettant de faire la distinction. Attention, la taille $|C|$ d'un circuit C correspond au nombre de ses portes, mais la taille de son code est $O(|C|^2)$.

On peut d'ores et déjà donner un lien simple entre taille et profondeur.

5-T Lemme

Soit C un circuit de profondeur p ayant une unique sortie. Si les portes ont un degré entrant majoré par k (pour un entier $k \geqslant 2$), alors $p < |C| \leqslant k^{p+1} - 1$.

Démonstration On a bien sûr $p < |C|$ puisqu'un chemin d'une entrée à une sortie

ne peut avoir une taille $\geqslant |C|$. On montre l'autre inégalité par récurrence sur p. Le résultat est clair pour $p = 0$ puisque C a une seule porte.

Pour $p > 0$: soit γ la porte de sortie, c'est-à-dire l'unique porte à la profondeur maximale p, et $C_1, \ldots, C_{k'}$ les sous-circuits issus de ses arguments, où $k' \leqslant k$. On a donc $|C| \leqslant 1 + \sum_{i=1}^{k'} |C_i| \leqslant 1 + k \max_i |C_i|$. Par hypothèse de récurrence (chaque C_i ayant une profondeur $\leqslant p-1$), $|C_i| \leqslant k^p - 1$ donc $|C| \leqslant 1 + k^{p+1} - k \leqslant 1 + k^{p+1}$. □

Nous pouvons également majorer le nombre de circuits d'une certaine taille.

5-U Lemme

Soit t et k des entiers. Si les portes ont un degré entrant majoré par k, alors il y a au plus $3^t\, t^{(k+1)t}$ circuits de taille $\leqslant t$.

Démonstration Un circuit C peut être décrit par la liste de ses portes avec pour chacune d'elles l'ensemble de ses arguments. Chaque porte ayant au plus k arguments, dans un circuit de taille $\leqslant t$ il y a donc au plus $\sum_{i=0}^{k} t^i \leqslant t^{k+1}$ possibilités pour l'ensemble des arguments de chaque porte (un circuit de taille $< t$ est vu comme un circuit de taille t où les portes superflues ne sont pas reliées au reste du circuit). Il y a également 3 façons de l'étiqueter : par $\vee$, $\wedge$ ou $\neg$. Ainsi, il y a au plus $3t^{k+1}$ possibilités pour chaque porte, soit au plus $3^t\, t^{(k+1)t}$ circuits différents de taille t au total. □

Nous allons utiliser les circuits pour reconnaître des langages. Un circuit a un nombre constant d'entrées, ce qui fait qu'il ne peut pas servir à lui seul pour tous les mots d'un langage : nous aurons donc besoin d'une *famille* $(C_n)_{n\in\mathbb{N}}$ de circuits, un pour chaque taille d'entrée (le circuit C_n reconnaissant des mots de taille n).

5-V Définition

- Si C est un circuit sur n variables, un mot $x \in \{0,1\}^n$ est *accepté* par C (noté $C(x) = 1$) si la fonction calculée par le circuit vaut 1 en x. Sinon, celle-ci vaut 0 en x et on dit que x est *rejeté* par C (noté $C(x) = 0$).
- Si $(C_n)_{n\in\mathbb{N}}$ est une famille de circuits où C_n est sur n variables, le *langage reconnu par* (C_n) est l'ensemble des mots acceptés par l'un des circuits de la suite (les mots de longueur n étant traités par le circuit C_n).
- On dit qu'une famille de circuits $(C_n)_{n\in\mathbb{N}}$ est de taille $\leqslant t(n)$ si pour tout n, $|C_n| \leqslant t(n)$. De même, une famille de circuits $(C_n)_{n\in\mathbb{N}}$ de profondeur $\leqslant p(n)$ est telle que la profondeur de C_n est $\leqslant p(n)$ pour tout n.

5-W **Remarque** Les circuits ne reconnaissent que des langages sur l'alphabet $\{0,1\}$: pour tout autre alphabet, il faut effectuer une traduction vers $\{0,1\}$ grâce à un morphisme.

5.2.2 Machines et circuits

Dans les deux propositions suivantes, nous allons voir le lien étroit en machines de Turing et familles de circuits. Tout d'abord, on peut évaluer efficacement un circuit.

5-X **Proposition**

Si C désigne un circuit booléen de taille t et de profondeur p, et x une entrée pour C, alors il existe une machine de Turing fonctionnant en temps $O(t^3)$ et en espace $O(p)$ qui sur l'entrée (C,x) calcule $C(x)$ (on remarquera que la taille de l'entrée (C,x) est $O(t^2)$ à cause du code de C).

Démonstration Nous pouvons évaluer $C(x)$ de manière récursive en suivant la définition de la valeur d'une porte du circuit.

> Eval(C,x) :
>
> – si C est réduit à une entrée étiquetée par une variable x_i alors renvoyer la valeur de x_i ;
>
> – si C est réduit à une entrée étiquetée par une constante c alors renvoyer la valeur de c ;
>
> – si la porte de sortie de C est une porte $\neg$ dont l'argument est le sous-circuit C', renvoyer $\neg$Eval(C',x) ;
>
> – si la porte de sortie de C est une porte $\vee$ dont les arguments sont les sous-circuits C_1 et C_2, renvoyer Eval$(C_1,x)\vee$Eval(C_2,x) ;
>
> – si la porte de sortie de C est une porte $\wedge$ dont les arguments sont les sous-circuits C_1 et C_2, renvoyer Eval$(C_1,x)\wedge$Eval(C_2,x).

Le temps d'exécution de cet algorithme vérifie

$$T(|C|) \leqslant O(|C|^2) + T(|C_1|) + T(|C_2|)$$

car on va chercher dans le code de C les sous-circuits C_1 et C_2 (ce qui prend $O(|C|^2)$ qui est la taille du codage de C) puis on évalue $C_1(x)$ et $C_2(x)$ (ce qui prend $T(|C_1|)+T(|C_2|)$). Puisque $|C_1|+|C_2| \leqslant |C|-1$, on a $|C_1|^3+|C_2|^3 \leqslant (|C|-1)^3$ et on montre aisément par récurrence que $T(t)=O(t^3)$.

Pour la borne en espace, si $S(p)$ désigne l'espace utilisé pour évaluer un circuit de profondeur p, on obtient

$$S(p) = \max\{O(\log t), S(p-1)+O(1)\}$$

puisque l'on va chercher dans le code de C le sous-circuit C_1 (ce qui prend un espace $O(\log|C|)$), en réutilisant cet espace on évalue C_1 (ce qui prend un espace $\leqslant S(p-1)$ puisque C_1 est de profondeur $\leqslant p-1$), on mémorise le résultat (espace $O(1)$) et on fait de même pour C_2 en réutilisant l'espace, puis on effectue l'opération sur les deux résultats. Au total, on trouve donc un espace $S(p) = O(p + \log t)$; puisque $t \leqslant 2^{p+1}$ (lemme 5-T), on obtient $O(p)$. □

Réciproquement, une famille de circuits peut simuler le fonctionnement d'une machine de Turing.

5-Y Proposition

- Si $A \in \mathsf{DTIME}(t(n))$ alors il existe une constante $\alpha > 0$ telle que A est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ de taille $\leqslant \alpha t(n)^2$.
- Si $A \in \mathsf{NSPACE}(s(n))$ alors il existe une constante $\alpha > 0$ telle que A est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ de profondeur $\leqslant \alpha s(n)^2$.

Idée de la démonstration Si A est reconnu par une machine M fonctionnant en temps $t(n)$, on considère le diagramme espace-temps de $M(x)$ comme à la figure 3.2. Dans ce diagramme, le contenu de la case i au temps $m+1$ dépend uniquement du contenu des cases $i-1$, i et $i+1$ au temps m, ainsi que de l'état au temps m. On construit un circuit ayant $t(n)$ « niveaux », chaque niveau correspondant à une étape dans le calcul $M(x)$ (voir la figure 5.2(a)). Chaque niveau contient $O(t(n))$ portes : un nombre constant de portes dont les valeurs coderont l'état de la machine à l'étape en cours, et pour chaque case un nombre constant de portes dont les valeurs coderont le contenu de la case à l'étape en cours, et un booléen pour savoir si la tête de lecture est sur cette case. Puisque le nombre d'états et la taille de l'alphabet de travail sont des constantes, un tel nombre de portes suffit.

Enfin, les cases du niveau $m+1$ prennent toutes leurs valeurs en fonction des cases du niveau m : il suffit d'un petit circuit, toujours le même, de taille constante, dépendant de la fonction de transition de M, calculant le contenu de la case i à l'étape $m+1$ en fonction du contenu des cases $i-1$, i et $i+1$ et de l'état au temps m (voir la figure 5.2(b)). De même, pour calculer l'état au temps $m+1$, chaque case du niveau m est affublée d'un circuit de taille constante calculant, en fonction de l'état au temps m, le nouvel état si la tête est sur cette case au temps m, et 0 sinon : il suffit ensuite de faire une grande disjonction bits à bits de tous ces résultats pour obtenir l'état au temps $m+1$ (voir la figure 5.2(c)). Tout cela donne un circuit de taille $O(t(n)^2)$.

Pour simuler une machine non déterministe fonctionnant en espace $s(n)$, on utilise à nouveau la méthode du graphe des configurations. Il s'agit de construire un circuit décidant s'il existe un chemin de taille $2^{O(s(n))}$ de la configuration initiale à une configuration acceptante. On suit pour cela l'algorithme de Savitch (théorème 4-AS) : pour i de 0 à $s(n)$ on construit un circuit C^i décidant s'il existe un chemin de taille $\leqslant 2^i$ d'une configuration u à une configuration v.

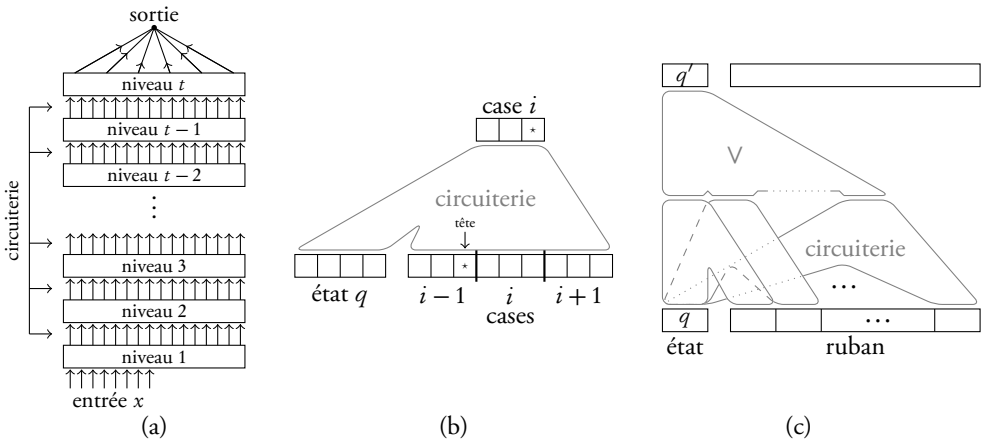


FIGURE 5.2 – (a) Vue d'ensemble du circuit simulant $M(x)$ selon le diagramme espace-temps de $M(x)$. (b) Calcul du contenu de la case i en fonction de l'état q et du contenu des cases voisines à l'étape précédente. (c) Calcul du nouvel état en fonction de l'ancien et de la case lue.

C^0 doit simplement décider si $u = v$ ou si v est accessible depuis u en une transition. Il s'agit de vérifier que les cases de u ne contenant pas la tête n'ont pas changé, que celle contenant la tête a changé en accord avec la relation de transition, et de même pour l'état. Pour cela, on compare pour chaque case son contenu dans u et dans v (profondeur constante) et on doit faire la conjonction de ces $s(n)$ comparaisons pour savoir si toutes son correctes : sous la forme d'un arbre, la profondeur est alors $O(\log s(n))$ (voir figure 5.3(a)).

Pour construire C^{i+1}, sur l'entrée (u, v) on teste pour toute configuration w si on a $C^i(u, w) = 1$ et $C^i(w, v) = 1$. On fait tous ces tests en parallèle (« en largeur », cf. figure 5.3(b)) si bien que la profondeur est celle de C^i à un près (et la « largeur » est $2^{O(s(n))}$). Puis il faut encore faire la disjonction de tous les résultats pour savoir s'il existe un tel w, ce qui prend une profondeur $\log(2^{O(s(n))}) = O(s(n))$ sous la forme d'un arbre. Ainsi, la profondeur de C^{i+1} est celle de C^i à laquelle on a ajouté $O(s(n))$. On en déduit que la profondeur de $C^{s(n)}$ est $O(s(n)^2)$. □

5-Z **Remarque** La preuve précédente est constructive et très uniforme : elle donne un algorithme déterministe fonctionnant en espace $O(\log t(n))$ qui, sur l'entrée n en unaire, construit le circuit simulant la machine $\mathsf{DTIME}(t(n))$ sur les entrées de taille n. Cette remarque nous servira notamment lorsqu'on abordera la question de l'uniformité des circuits.

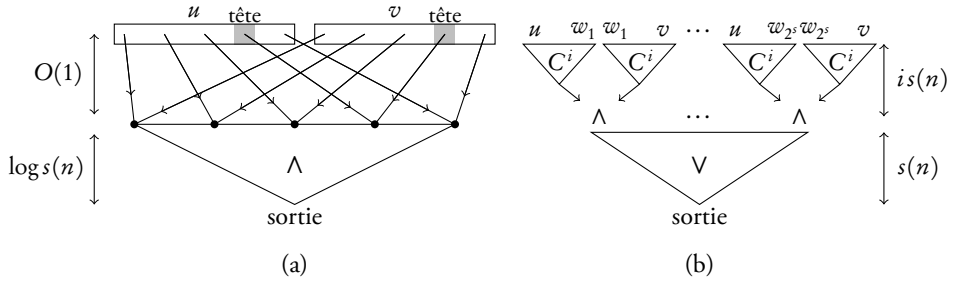


FIGURE 5.3 – (a) Le circuit C^0 pour tester si une configuration v est accessible depuis une configuration u en une étape. (b) Le circuit C^{i+1} construit à partir de C^i.

Un peu de recul

Ce qui précède montre que la taille d'un circuit correspond au temps de calcul, tandis que sa profondeur correspond à l'espace. Il est facile de voir que la profondeur correspond également au temps de calcul parallèle sur une machine disposant de suffisamment de processeurs pour évaluer en même temps toutes les portes d'un même niveau.

Ainsi, le temps parallèle et l'espace sont polynomialement liés et s'interprètent comme la profondeur d'un circuit.

5-AA Corollaire

- Tout langage de P a des circuits de taille polynomiale.
- Tout langage de PSPACE a des circuits de profondeur polynomiale.

On peut en outre déduire de ces deux propositions un résultat de complétude dû à Ladner [Lad75b]. Soit `VALEUR CIRCUIT` le problème suivant :

- *entrée* : un circuit C dont toutes les entrées sont des constantes `0` ou `1` ;
- *question* : la valeur calculée par le circuit C est-elle `1` ?

5-AB Proposition

Le problème `VALEUR CIRCUIT` est P-complet pour les réductions en espace logarithmique.

Idée de la démonstration La proposition 5-X permet d'évaluer en temps polynomial un circuit donné en entrée. Donc `VALEUR CIRCUIT` $\in$ P.

Pour la P-difficulté, soit $A \in \mathsf{P}$ reconnu par une machine fonctionnant en temps polynomial. Par la proposition 5-Y, A est reconnu par une famille (C_n) de circuits de taille polynomiale. Par ailleurs, la remarque 5-Z montre que ces circuits sont constructibles en espace $O(\log n)$. Si $a \in \{0,1\}^n$ est fixé, on note D_a le circuit C_n dans lequel l'entrée x_i est remplacée par la constante a_i (il s'agit donc de C_n évalué en a).

La réduction f de A à `VALEUR CIRCUIT` associe alors D_x à x : $x \in A$ ssi $C_{|x|}(x) = 1$ ssi D_x vaut 1 ssi $f(x) \in$ `VALEUR CIRCUIT`. Puisque f est calculable en espace logarithmique, on en déduit que `VALEUR CIRCUIT` est P-complet pour les réductions en espace logarithmique. □

Des problèmes EXP-complets et NEXP-complets concernant les circuits sont proposés à l'exercice B-C.

5.2.3 Circuits et conseils

On peut caractériser la classe P/poly en termes de circuits (c'est d'ailleurs souvent cette caractérisation qui est prise comme définition) grâce au résultat suivant montré par Karp et Lipton [KL82].

5-AC Proposition

P/poly est la classe des langages reconnus par une famille de circuits de taille polynomiale.

En d'autres termes, un langage A est dans P/poly ssi il existe une famille (C_n) de circuits et un polynôme $p(n)$ tels que :

- pour tout n, $|C_n| \leqslant p(n)$;
- pour tout $x \in \{0,1\}^*$, $C_{|x|}(x) = 1$ ssi $x \in A$.

Idée de la démonstration Si un langage est reconnu par une famille de circuits de taille polynomiale, alors il suffit de donner comme conseil le codage du circuit correspondant à la taille de l'entrée. La machine se contente d'évaluer le circuit sur l'entrée x, ce qui prend un temps polynomial.

Réciproquement, si $A \in \mathsf{P/poly}$ alors il est reconnu par la famille de circuit qui simule la machine avec le bon conseil. Puisque le conseil est le même pour toutes les entrées de même taille, on peut le coder « en dur » en entrée du circuit. Le circuit a une taille polynomiale puisque le temps de calcul est polynomial.

Démonstration Soit A un langage reconnu par une famille de circuits (C_n) de taille polynomiale. Par la proposition 5-X, il existe une machine de Turing M fonctionnant en temps polynomial qui, sur l'entrée $(x, C_{|x|})$, calcule la valeur $C_{|x|}(x)$. On appellera $B \in \mathsf{P}$ le langage reconnu par M. Pour reconnaître A, il suffit alors de donner comme conseil pour les entrées de taille n le code du circuit C_n, car $x \in A$ ssi $(x, C_n) \in B$. Puisque C_n est de taille polynomiale, cela montre que $A \in \mathsf{P/poly}$.

Réciproquement, si $A \in \mathsf{P/poly}$, alors A est reconnu par une machine M polynomiale avec conseils (a_n) de taille polynomiale : $x \in A$ ssi $M(x, a_{|x|}) = 1$. La proposition 5-Y donne un circuit $C_n(x,y)$ de taille polynomiale qui simule $M(x,y)$ pour $|x| = n$ et $|y| = |a_n|$. On remplace alors les entrées y par les constantes a_n, ce qui laisse la taille du circuit inchangée. Si l'on appelle D_n le nouveau circuit, on a $x \in A$ ssi $D_{|x|}(x) = 1$, ce qui conclut puisque D_n a une taille polynomiale. □

C'est en partie cette proposition qui donne son importance à la classe P/poly. Les autres classes non uniformes n'admettent pas de caractérisations si simples, à part peut-être L/poly qui correspond à l'ensemble des langages reconnus par des *branching programs* de taille polynomiale (nous ne définirons pas les branching programs car ils ne nous serviront pas par la suite).

5.3 Uniformité des circuits

La relation entre machines et circuits va encore plus loin que ce qu'on vient de voir. En effet, nous avons pour le moment étudié des familles où les circuits n'ont pas de lien entre eux : certains peuvent être compliqués, certains simples, mais la connaissance du circuit C_n ne nous renseigne pas sur celle de C_{n+1}. Or en imposant de pouvoir construire le code de C_n efficacement, on retrouve les classes uniformes que l'on a étudiées jusqu'à présent.

5-AD Définition

Soit $\mathscr{C}$ une classe de complexité et (C_n) une famille de circuits. On dit que (C_n) est *$\mathscr{C}$-uniforme* si

$$\{(1^n, i) \mid \text{le } i\text{-ème bit du code de } C_n \text{ est } 1\} \in \mathscr{C},$$

où i est donné en binaire.

En particulier, une famille de circuits (C_n) de taille polynomiale est P-uniforme (respectivement L-uniforme, PSPACE-uniforme) si on peut construire le code de C_n en temps polynomial en n (resp. espace logarithmique, polynomial en n).

Sans indication de la classe $\mathscr{C}$, (C_n) est dite uniforme si elle est P-uniforme.

Les familles uniformes de circuits de taille polynomiale reconnaissent une classe bien connue.

5-AE Proposition

La classe P est l'ensemble des langages reconnus par une famille *uniforme* de circuits de taille polynomiale.

Idée de la démonstration Si A est reconnu par une famille uniforme (C_n) de circuits de taille polynomiale, pour décider si $x \in A$ il suffit de :

– construire $C_{|x|}$ en temps polynomial en $|x|$ par uniformité ;

– évaluer $C_{|x|}(x)$ en temps polynomial par la proposition 5-X.

On décide ainsi A en temps polynomial.

Réciproquement, si $A \in \mathsf{P}$ alors la remarque 5-Z montre que A possède une famille de circuits de taille polynomiale dont le code est calculable en espace logarithmique : en particulier, cette famille est P-uniforme. □

✎ **5-AF** **Exercice**

Vérifier que la démonstration ci-dessus montre en réalité que $A \in \mathsf{P}$ ssi A est reconnu par des circuits L-uniformes de taille polynomiale. On en déduit que pour les circuits de taille polynomiale, la P-uniformité et la L-uniformité ont la même puissance.

De même, montrer qu'un langage A est dans PSPACE ssi A possède des circuits PSPACE-uniformes de profondeur polynomiale ssi A possède des circuits L-uniformes de profondeur polynomiale. Donc pour des circuits de profondeur polynomiale, les uniformités L, P et PSPACE ont même puissance.

On peut s'interroger sur la pertinence de définir un modèle non uniforme (les circuits) pour y adjoindre ensuite une contrainte d'uniformité grâce à des machines de Turing. C'est en effet peu utile pour les circuits de taille polynomiale puisqu'on retrouve la classe P. En revanche, cela prend tout son sens pour des classes plus petites que nous verrons à la section suivante (circuits de profondeur bornée par exemple), puisque ces classes n'ont pas nécessairement d'analogue défini par machine de Turing et que l'uniformité est nécessaire pour les comparer aux classes usuelles uniformes.

5-AG **Remarque** Pour de petites classes de circuits comme celles ci-dessous, d'autres notions d'uniformité plus restrictives ont été définies (par exemple l'uniformité DLOGTIME où la description de chaque bit du code du circuit C_n se fait en temps $O(\log n)$), mais nous ne les aborderons pas dans cet ouvrage.

5.4 Autres classes définies par circuits

Les circuits sont également utiles pour définir de « petites » classes, incluses dans P/poly. Nous ne les étudierons que marginalement dans ce livre, mais en donnons tout de même la définition. Nous commençons par une classe uniforme.

5-AH **Définition**

Pour $i \geqslant 1$, NC^i est la classe des langages reconnus par une famille L-uniforme de circuits de taille polynomiale et de *profondeur* $O((\log n)^i)$.

NC désigne l'union des NC^i, c'est-à-dire que la profondeur des circuits est polylogarithmique.

5-AI **Remarque** Le N dans NC ne signifie pas « non déterministe », mais NC désigne *Nick's Class* en l'honneur de Nick Pippenger qui a étudié ce genre de circuits.

Les propositions 5-X et 5-Y montrent le résultat suivant.

5-AJ **Proposition**

$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{NC} \subseteq \mathsf{P}$.

On peut encore restreindre la profondeur mais il faut alors autoriser les portes à avoir un degré entrant quelconque. Voici deux autres classes non uniformes importantes définies par des circuits de profondeur constante.

5-AK **Définition**

- AC^0 est la classe des langages reconnus par une famille de circuits de taille polynomiale et de *profondeur constante* (c'est-à-dire qu'il existe une constante α telle que $|C_n| \leqslant \alpha$ pour tout n) avec des portes $\vee$ et $\wedge$ de degré entrant arbitraire.
- Pour définir TC^0, on a besoin de portes de majorité (ou « portes à seuil »). Une porte de majorité a un degré entrant arbitraire et renvoie 1 si au moins la moitié de ses arguments valent 1, et 0 sinon.
- TC^0 est la classe des langages reconnus par une famille de circuits de taille polynomiale et de profondeur constante avec des portes $\vee$ et $\wedge$ de degré entrant arbitraire et des *portes de majorité*.

5-AL **Remarques**

- On pourrait définir AC^i et TC^i pour $i \geqslant 1$ de la même manière mais avec des circuits de profondeur $O((\log n)^i)$. Cependant, ces classes sont beaucoup moins étudiées.
- Les noms AC et TC viennent de *Alternation Circuits* et de *Threshold Circuits*.

5-AM Proposition

$\mathsf{AC}^0 \subseteq \mathsf{TC}^0 \subseteq \mathsf{L/poly}$.

Idée de la démonstration La première inclusion est évidente. Pour la seconde, si $A \in \mathsf{TC}^0$ alors on donne comme conseil pour les entrées de taille n la description du circuit C_n. Il s'agit alors de montrer qu'on peut l'évaluer en espace logarithmique. On procède de manière récursive comme à la proposition 5-X : pour évaluer une porte g d'arguments $g_1, \ldots, g_m$, on évalue les arguments dans l'ordre, en retenant où l'on en est par un compteur $i \in \{1, \ldots, m\}$, et selon le type de porte g, soit on attend d'obtenir un 1 (pour une porte $\vee$) ou un 0 (pour une porte $\wedge$), soit on compte le nombre de 1 (pour une porte majorité). L'espace utilisé à la profondeur p satisfait donc $S(p) = O(\log n) + S(p-1)$. Ainsi, $S(p) = O(p \log n)$, donc l'espace utilisé est $O(\log n)$ puisque la profondeur p est constante. □

5.5 Intérêt des circuits et bornes inférieures non uniformes

Outre leur définition naturelle, les circuits ont été étudiés notamment car leur manipulation est plus aisée que celle des machines de Turing. Nous verrons au chapitre 7 sur les limites de la diagonalisation que pour résoudre une question comme « $\mathsf{P} = \mathsf{NP}$? », il faut comprendre le fonctionnement interne des machines de Turing et non seulement les utiliser comme des boîtes noires. Or la définition des machines de Turing est trop compliquée pour comprendre précisément ce qui se passe, tandis qu'on peut manipuler plus aisément celle des circuits. Ainsi, pour montrer que $\mathsf{P} \neq \mathsf{NP}$, une stratégie est de montrer que NP n'a pas de circuits de taille polynomiale, comme le montre la proposition suivante.

5-AN Proposition

Si NP n'a pas de circuits de taille polynomiale, alors $\mathsf{P} \neq \mathsf{NP}$.

Démonstration Puisque $\mathsf{P} \subset \mathsf{P/poly}$, si $A \in \mathsf{NP} \setminus (\mathsf{P/poly})$ alors $A \in \mathsf{NP} \setminus \mathsf{P}$. □

Montrer qu'une classe de complexité n'admet pas de petits circuits est alors un problème central en complexité, mais c'est aussi un problème généralement difficile. Nous étudierons plus en détail ce genre de question au chapitre 11 sur les bornes inférieures non uniformes.

Notre connaissance est très limitée à ce sujet. Mentionnons par exemple que nous ne savons pas montrer qu'il existe des problèmes dans EXP qui n'ont pas de circuits de taille polynomiale : en symboles, « $\mathsf{EXP} \subset \mathsf{P/poly}$? » est une question ouverte. Comme nous l'avons déjà remarqué, ces deux classes sont différentes puisque $\mathsf{P/poly}$ est indénombrable, c'est donc bien l'inclusion qui nous importe. Notons que l'analogue uniforme de cette question est bien connu : séparer EXP de P se fait simplement par le théorème de hiérarchie en temps déterministe.

Il nous faudra plus d'outils pour aborder ces questions, c'est pourquoi nous le ferons seulement au chapitre 11.

5.6 Circuits arithmétiques

Outre les circuits booléens, nous aurons besoin par la suite de circuits arithmétiques qui calculent des polynômes. La définition est la même mais les circuits possèdent maintenant des portes d'addition et de multiplication plutôt que de disjonction et de conjonction. Pour parler de polynômes, on se place sur un corps K quelconque dans lequel vivront les coefficients du polynôme.

5-AO Définition

Un circuit arithmétique sur un corps K est un graphe orienté acyclique (et connexe) tel que :

- les sommets de degré entrant nul sont appelés *entrées* et sont étiquetés par le nom d'une variable x_i ou par une constante quelconque $\alpha \in K$;
- les autres sommets sont de degré entrant 2 et sont soit des *portes d'addition*, soit des *portes de multiplication*, respectivement étiquetées par $+$ et $\times$;
- un unique sommet est de degré sortant nul et est appelé *sortie* ;
- les voisins entrants d'une porte sont appelés ses *arguments*.

Nous dirons qu'un circuit est *sans constante* s'il n'utilise pas de constante arbitraire de K autre que -1 (pour pouvoir effectuer des soustractions).

La taille d'un circuit arithmétique est le nombre de portes de celui-ci.

5-AP Remarque Un circuit arithmétique sans constante, en tant que graphe étiqueté, peut bien sûr être encodé en binaire, par exemple par sa matrice d'adjacence et la liste des étiquettes des sommets. En revanche, si des constantes quelconques d'un corps infini sont autorisées alors celles-ci ne sont pas forcément encodable de manière finie.

Un circuit calcule un polynôme de manière naturelle.

5-AQ Définition

Si C est un circuit arithmétique sur K sur les variables $x_1, \ldots, x_n$, alors chaque porte γ de C calcule un polynôme $F_\gamma \in K[x_1, \ldots, x_n]$ défini comme suit :

- une porte γ étiquetée par une constante $\alpha \in K$ calcule le polynôme constant $f_\gamma(x_1, \ldots, x_n) = \alpha$;

– une porte γ étiquetée par une variable x_i calcule le polynôme $f_\gamma(x_1,\dots,x_n)=x_i$;
– une porte γ étiquetée par $+$ (respectivement $\times$) et dont les arguments sont γ_1 et γ_2 calcule le polynôme $f_\gamma = f_{\gamma_1}+f_{\gamma_2}$ (resp. $f_\gamma = f_{\gamma_1}f_{\gamma_2}$).

Le polynôme calculé par C est le polynôme calculé par sa porte de sortie.

5-AR **Remarque** Attention, on manipule ici vraiment des polynômes, et non des fonctions polynomiales.

On trouvera quelques exemples de circuits arithmétiques à la figure 5.4, illustrant notamment la technique des élévations au carré successives pour calculer l'entier 2^{2^n} en $n+1$ opérations.

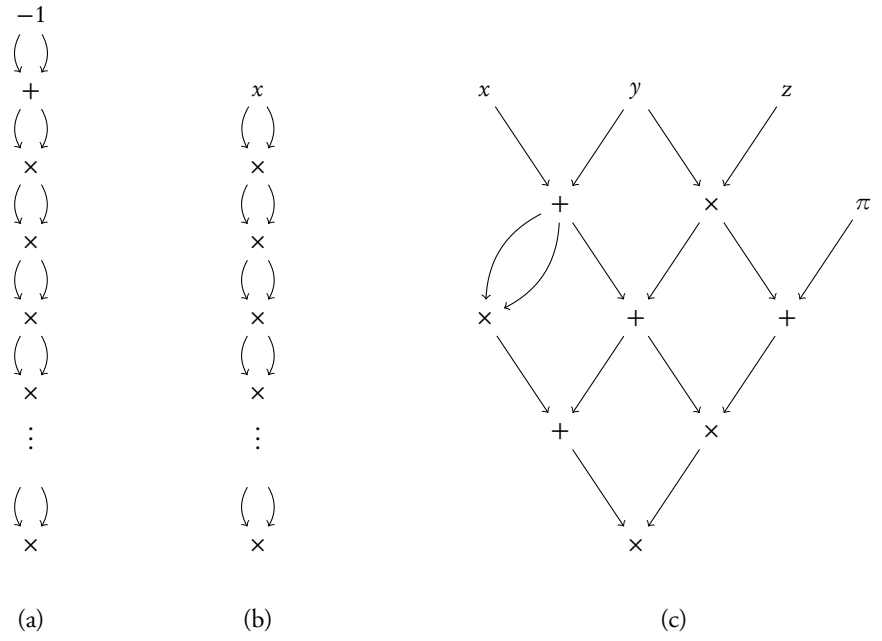


FIGURE 5.4 – (a) Circuit arithmétique sans constante de taille $n+2$ pour calculer 2^{2^n}. (b) Circuit arithmétique sans constante de taille $n+1$ pour calculer le polynôme x^{2^n}. (c) Circuit arithmétique sur $\mathbb{R}$ de taille 12 calculant le polynôme $((x+y)^2+x+y+yz)(x+y+yz)(\pi+yz)$.

Attardons-nous sur une remarque. Puisque le degré du polynôme calculé par une porte est au plus la somme de ceux de ses arguments, on déduit le lemme suivant.

5-AS **Lemme**

Si f est un polynôme calculé par un circuit de taille t, alors son degré est majoré par 2^t.

Dans le même esprit, 2^{2^n} est le plus grand entier calculable en $n+1$ opérations $+$ ou $\times$ à partir de la constante -1, comme le montre le lemme suivant. On peut aussi majorer les coefficients d'un polynôme calculé par un circuit sans constante de taille t, résultat dont nous aurons besoin plus tard.

5-AT **Lemme**

- Si $N \in \mathbb{Z}$ est calculé par un circuit arithmétique sans constante de taille t, alors $|N| \leqslant 2^{2^{t-1}}$.
- Si $p \in \mathbb{Z}[x_1, \ldots, x_n]$ est calculé par un circuit arithmétique sans constante de taille t, alors les coefficients de p sont strictement majorés en valeur absolue par $2^{2^{2t}}$.

Démonstration Pour le premier point, nous montrons que la meilleure stratégie pour calculer un grand entier est d'effectuer des élévations au carré successives. On raisonne par récurrence sur t. Pour $t = 1$, la valeur du circuit est $N = -1$ donc $|N| \leqslant 2^{2^{t-1}}$. Pour $t > 1$, si la porte de sortie du circuit est une addition, alors N est la somme de deux termes calculés par un circuit de taille au plus $t-1$, donc $|N| \leqslant 2 \times 2^{2^{t-2}} \leqslant 2^{2^{t-1}}$ par hypothèse de récurrence. Si c'est une multiplication, alors $|N| \leqslant (2^{2^{t-2}})^2 = 2^{2^{t-1}}$. Donc l'hypothèse est montrée au rang t.

Pour le second point, un polynôme en n variables calculé par un circuit de taille t a un degré $\leqslant 2^t$ (lemme 5-AS). Ainsi, il a au plus $(1+2^t)^n$ monômes (correspondant au choix d'un degré entre 0 et 2^t pour chacune des n variables).

On montre le résultat par récurrence sur t. Pour $t = 1$, le résultat est clair puisque le polynôme calculé est soit la constante -1 soit une variable x_i. Pour $t > 1$, soit C un circuit de taille t et $g = g_1 \circ g_2$ sa porte de sortie, où $\circ \in \{\times, +\}$. Par hypothèse de récurrence, les coefficients de g_1 et g_2 ont tous leur valeur absolue majorée par $\alpha_{t-1} = 2^{2^{2(t-1)}}$.

Si $g = g_1 + g_2$ alors les coefficients de g sont tous la somme d'un coefficient de g_1 et d'un coefficient de g_2 donc ils sont majorés en valeur absolue par $2\alpha_{t-1}$.

Si $g = g_1 g_2$ alors les coefficients de g sont la somme d'au plus $(1+2^t)^n$ produits d'un coefficient de g_1 et d'un coefficient de g_2 donc ils sont majorés en valeur absolue par $(1+2^t)^n \alpha_{t-1}^2$.

Puisque $n \leqslant t$ on a donc dans les deux cas :

$$\alpha_t \leqslant (1+2^t)^t \alpha_{t-1}^2 \leqslant 2^{t(t+1)} 2^{2 \cdot 2^{2(t-1)}} = 2^{t(t+1)+2^{2t-1}} < 2^{2^{2t}}.$$

□

Dans certaines constructions, nous aurons besoin de « tronquer » un polynôme. On appelle *composante homogène* de degré i d'un polynôme $f(x_1,\dots,x_n)$ la somme des monônes de f de degré i. Par exemple, la composante homogène de degré 2 de

$$1+2x+y+3x^2-xy+5y^2-2x^3+x^2y^4$$

est $3x^2-xy+5y^2$. Le polynôme f *tronqué* au degré d est la somme des composantes homogènes de f de degré allant de 0 à d. Dans l'exemple précédent, le polynôme tronqué au degré 2 est $1+2x+y+3x^2-xy+5y^2$. Si d n'est pas trop grand, il est facile de tronquer un polynôme calculé par un circuit.

5-AU Lemme

Si un circuit arithmétique C de taille t calcule un polynôme f, alors il existe un circuit arithmétique D de taille $\leqslant \alpha d^2 t$ calculant f tronqué au degré d, où α est une constante indépendante de C.

Démonstration Pour chaque porte g de C, on va ignorer les monônes de degré $\geqslant d+1$ et on va calculer les composantes homogènes de degré compris entre 0 et d. On notera g_i la composante homogène de degré i de g. Pour le calcul par D des composantes homogènes, on remplace chaque porte g de C par $d+1$ portes de D, la porte i calculant g_i.

Pour une constante $\gamma \in K$ en entrée, la porte de D correspondant à la composante homogène de degré 0 vaut γ, tandis que les autres valent 0. Pour une variable x, la porte de D correspondant à la composante homogène de degré 1 vaut x, tandis que les autres valent 0. Pour une porte $g = p+q$, pour tout i on a $g_i = p_i + q_i$, ce qui permet de calculer chaque g_i par une simple somme en fonction des composantes homogènes de p et q. Enfin, si $g = pq$, alors pour tout i on a $g_i = \sum_{j=0}^{i} p_j q_{i-j}$, ce qui permet de calculer chaque g_i par une circuiterie de taille $2i+1$ (soit une circuiterie totale de taille $\sum_{i=0}^{d}(2i+1) = (d+1)^2$ pour les $d+1$ composantes homogènes). La construction est illustrée à la figure 5.5.

La sortie du circuit D est alors la somme des $d+1$ composantes homogènes de la sortie de C, ce qui prend d portes d'addition supplémentaires. Au total, la taille de D est $O(d^2 t)$. □

On aura remarqué que nos circuits n'ont pas de porte de division puisqu'on souhaite rester dans l'anneau des polynômes. Néanmoins, les divisions semblent a priori pouvoir accélérer le calcul, comme par exemple pour calculer le polynôme

$$\sum_{i=0}^{d} x^i = \frac{x^{d+1}-1}{x-1}$$

pour lequel la division par $x-1$ réduit le nombre d'opérations à $O(\log d)$. En réalité, tant qu'on calcule un polynôme de petit degré celles-ci n'apportent pas grand-chose comme le montre « l'élimination des divisions » de Strassen [Str73b] ci-dessous. C'est donc une autre raison pour n'autoriser que les opérations $+$ et $\times$ dans nos circuits.

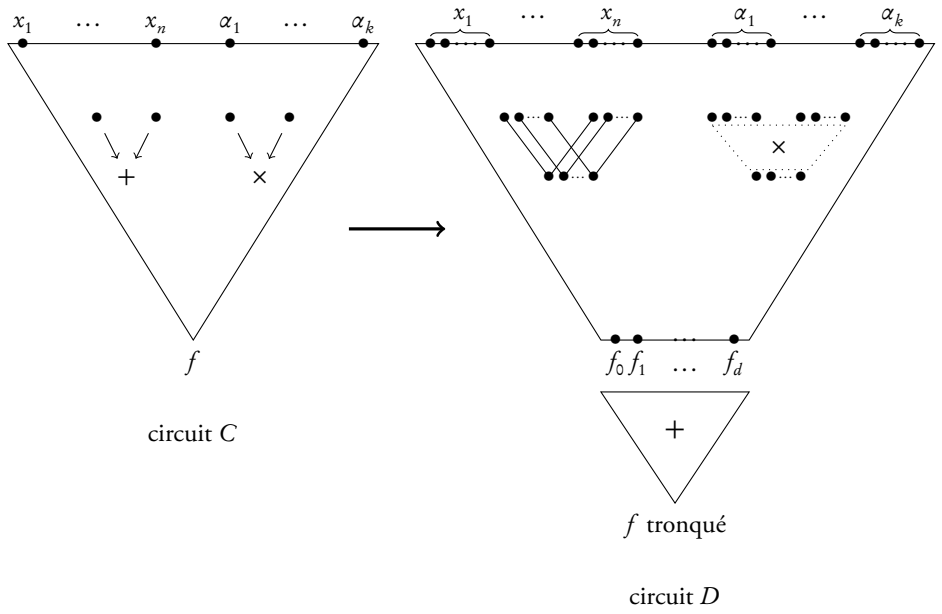


FIGURE 5.5 – Transformation d'un circuit C calculant f en un circuit D calculant f tronqué au degré d.

5-AV **Proposition (*Vermeidung von Divisionen*)**

Soit K un corps infini. Soit C un circuit arithmétique de taille t *avec divisions* calculant dans le corps des fractions rationnelles $K(x_1,\dots,x_n)$. Si C calcule un *polynôme* f de degré d, alors il existe un circuit arithmétique D *sans division* de taille $\leqslant \alpha(d^3+d^2t)$ calculant f, où α est une constante indépendante de C et d.

Idée de la démonstration On souhaiterait remplacer une division $f = p/(1-q)$ par son développement en série entière $p\sum_{i\geqslant 0} q^i$, ce qui ne fait plus intervenir de division, mais cette somme est infinie. En utilisant le fait que f est un polynôme et en tronquant le résultat comme au lemme précédent, on peut montrer que le début de cette somme suffit, c'est-à-dire qu'on remplacera la division par $p\sum_{i=0}^{d} q^i$ puis on tronquera le résultat au degré d.

Démonstration Tout d'abord, on remplace chaque porte g de C par deux portes, la première calculant le numérateur de g et la seconde son dénominateur. Puisque

$$(p/q)(p'/q') = (pp')/(qq') \quad \text{et} \quad (p/q)+(p'/q') = (pq'+p'q)/(qq'),$$

on peut aisément calculer le numérateur et dénominateur d'une porte d'addition ou de multiplication à partir des numérateurs et dénominateurs de ses arguments, ce qui permet de calculer les numérateurs et dénominateurs de toutes les portes de C en

multipliant la taille de C par une constante seulement. Dans ce nouveau circuit, il n'y a pas de division. Le polynôme f est alors le quotient du numérateur de la porte de sortie de C par son dénominateur, c'est-à-dire qu'on s'est ramené à une unique division $f = p/q$.

Soit $\alpha \in K^n$ un n-uple de constantes tel que $q(\alpha) = \beta \neq 0$, qui existe car K est infini et $q \neq 0$ en tant que dénominateur. Pour x un n-uple de variables, on note alors

$$p'(x) = \beta^{-1} p(\alpha + x) \quad \text{et} \quad q'(x) = 1 - \beta^{-1} q(\alpha + x),$$

de sorte que $q'(0) = 0$. On considère le circuit C' qui calcule p' et q', sa taille est linéaire en celle de C, et on note $f'(x) = f(\alpha + x)$, qui reste un polynôme de degré d et vérifie $f' = p'/(1 - q')$. On a alors $f' - f'q' = p'$. Donc par télescopage des termes q'^i,

$$p' \sum_{i=0}^{d} q'^i = f' \sum_{i=0}^{d} q'^i - f'q' \sum_{i=0}^{d} q'^i = f'(1 - q'^{d+1}).$$

Puisque $q'(0) = 0$, tout monôme de q'^{d+1} a un degré $\geqslant d + 1$, donc la troncature de $p' \sum_{i=0}^{d} q'^i$ au degré d vaut f'.

Ainsi, le calcul de f' s'effectue en calculant dans un premier temps $\sum_{i=0}^{d} q'^i$, qui requiert $O(d)$ portes supplémentaires une fois que q' est calculé, c'est-à-dire une taille $O(d + t)$ puisque C' a une taille $O(t)$. Puis on tronque le produit $p' \sum_{i=0}^{d} q'^i$ au degré d. Le lemme 5-AU indique qu'une telle opération multiplie la taille par $O(d^2)$. Ainsi, f' se calcule par un circuit de taille $O(d^3 + d^2 t)$.

Enfin, pour calculer f il suffit d'évaluer f' en $(x - \alpha)$ et on obtient donc un circuit sans division et de taille $O(d^3 + d^2 t)$ pour f. □

Précisons enfin pour les lecteurs désirant s'entraîner que les exercices B-B et B-G en annexe traitent également de non-uniformité.

6 Algorithmes probabilistes

Beaucoup d'algorithmes utilisent le hasard pour être plus efficaces, par exemple le classique tri rapide. Or les machines de Turing vues jusqu'à présent ne parlent aucunement de probabilités. Dans ce chapitre, nous allons donner le cadre théorique nécessaire pour l'étude des algorithmes probabilistes, définir les classes importantes de problèmes résolus par algorithmes probabilistes et enfin les comparer avec celles vues jusqu'à présent.

Nous commençons par l'exemple du tri rapide pour illustrer les nouvelles caractéristiques des machines de Turing probabilistes. Cependant, nous n'utiliserons pas ces machines par la suite car tout peut être défini en termes de machines de Turing déterministes.

6.1 Machines de Turing probabilistes

Nous avons vu que la classe NP peut être définie en termes de machines non déterministes ou en termes de machines déterministes avec un quantificateur existentiel (voir section 2.2.6). De même pour les classes probabilistes, on peut définir des machines probabilistes ou se contenter des machines déterministes. Nous privilégierons cette seconde approche, mais souhaitons tout de même définir les machines probabilistes pour mieux coller à la réalité des algorithmes probabilistes.

Il y a de nombreux exemples d'algorithmes probabilistes ; mais dans cette introduction nous illustrons notre propos par le classique tri rapide *(quicksort)*. Nous verrons un autre exemple intéressant plus loin dans ce chapitre.

6.1.1 Tri rapide

Nombre d'algorithmes existent pour trier par ordre croissant un tableau $T = (x_1, \ldots, x_n)$ de n nombres distincts. Parmi eux, le tri rapide fonctionne en choisissant un *pivot* x_i,

puis en triant récursivement les deux tableaux $T_1 = \{x_j \mid x_j < x_i\}$ et $T_2 = \{x_j \mid x_j > x_i\}$. Pour former T_1 et T_2, il suffit de parcourir T ce qui prend un temps linéaire ($\Theta(n)$).

La différence entre les versions déterministe et probabiliste vient du choix du pivot. Voici la version déterministe où on prend le premier élément x_1 comme pivot.

TriRapide_det(T) :

- si T a 0 ou 1 élément, `renvoyer` T ; sinon :
- $T_1 \leftarrow \{x_j \mid x_j < x_1\}$;
- $T_2 \leftarrow \{x_j \mid x_j > x_1\}$;
- `renvoyer` TriRapide_det(T_1),x_1,TriRapide_det(T_2).

Le problème de cette version est qu'elle fonctionne en $\Theta(n^2)$ étapes dans le pire cas. En effet, dans le cas particulier où T est déjà trié par ordre croissant, alors T_1 est toujours vide au cours de l'algorithme et T_2 possède $n-1$ éléments. Le nombre d'étapes $C(n)$ vérifie donc la relation $C(n) = \Theta(n) + C(n-1)$, ce qui donne $C(n) = \Theta(n^2)$.

Pour remédier à ce problème, la version probabiliste de l'algorithme choisit le pivot au hasard parmi $x_1, \ldots, x_n$.

TriRapide_proba(T) :

- si T a 0 ou 1 élément, `renvoyer` T ; sinon :
- choisir i au hasard parmi $\{1, \ldots, n\}$;
- $T_1 \leftarrow \{x_j \mid x_j < x_i\}$;
- $T_2 \leftarrow \{x_j \mid x_j > x_i\}$;
- `renvoyer` TriRapide_proba(T_1),x_i,TriRapide_proba(T_2).

Dans ce cas, le nombre d'étapes dépend du choix aléatoire des pivots. Pour tout $i \in \{1, \ldots, n\}$, avec probabilité $1/n$, T_1 a $i-1$ éléments et T_2 en a $n-i$. Donc le nombre moyen d'étapes vérifie

$$C(n) = \Theta(n) + \sum_{i=1}^{n} (C(i-1) + C(n-i))/n,$$

ce qui donne $C(n) = \Theta(n \log n)$, espérance du nombre d'étapes de l'algorithme probabiliste dans tous les cas.

Ainsi, l'algorithme probabiliste possède un temps moyen d'exécution de $\Theta(n \log n)$ *sur toute entrée*, alors que pour certaines entrées, l'algorithme déterministe a toujours un temps d'exécution de $\Theta(n^2)$.

Pour exécuter l'algorithme probabiliste, une machine doit être capable de tirer un nombre au hasard entre 1 et n. Plutôt que le nombre lui-même, on tire ses bits un à un, ce qui fait qu'il suffit de savoir tirer à pile ou face (0 ou 1). Cela donne naissance à la notion de machine de Turing probabiliste.

6.1.2 Machines de Turing probabilistes

Comme pour les machines non déterministes, plusieurs transitions seront possibles à chaque étape (sans perte de généralité, on se limitera à deux possibilités). La différence avec les machines non déterministes vient simplement de l'interprétation du calcul : ici on attribue des probabilités à chaque choix.

6-A Définition (machine de Turing probabiliste)

Une *machine de Turing probabiliste* M est une machine de Turing munie d'une relation de transition δ qui permet à chaque étape deux transitions possibles. Lors de l'exécution de M sur une entrée x, à chaque étape l'une ou l'autre des deux transitions est appliquée, avec probabilité $1/2$ chacune, indépendamment des choix précédents.

À la fin du calcul, M accepte x ($M(x) = 1$) ou rejette x ($M(x) = 0$). Le résultat de l'exécution est donc une variable aléatoire à valeur dans $\{0, 1\}$.

On dit que M fonctionne en temps $\leqslant t(n)$ si pour toute entrée x, le nombre d'étapes de l'exécution de $M(x)$ est inférieur à $t(|x|)$ *quels que soient les choix probabilistes.*

L'arbre d'exécution d'une machine de Turing probabiliste est donc un arbre binaire, et la probabilité d'une branche de longueur l est 2^{-l}.

6.2 Classes probabilistes

Soit M une machine probabiliste fonctionnant en temps $t(n)$. Si l'on donne une suite de choix probabilistes sous la forme d'un mot $r \in \{0,1\}^{t(n)}$, alors on peut simuler de manière déterministe le calcul de M le long de cette suite de choix. Il existe donc une machine déterministe M' qui, sur l'entrée (x, r), exécute le calcul de $M(x)$ le long des choix probabilistes donnés par r.

Ainsi, plutôt que de manipuler les machines probabilistes, nous utiliserons des machines déterministes qui prendront en entrée, en plus du mot x, le mot r des choix probabilistes (que l'on appellera *bits aléatoires*). Souvent on ne parlera même plus de machines mais seulement de langages (dans P, par exemple).

6.2.1 Définitions

Nous devons les définitions qui suivent à Gill [Gil77]. Une machine probabiliste M peut accepter un mot x le long de certains choix probabilistes et le rejeter le long d'autres choix. Cependant, pour reconnaître raisonnablement un langage A, on sent intuitivement que M ne doit pas faire trop d'erreurs : $M(x)$ doit accepter avec grande probabilité si $x \in A$, et réciproquement $M(x)$ doit rejeter avec grande probabilité si $x \notin A$. C'est ce qui donne lieu à la définition de la classe suivante.

6-B Définition (BPP)

La classe BPP (pour *Bounded error Probabilistic Polynomial time*) est l'ensemble des langages A tels qu'il existe un polynôme $p(n)$ et un langage $B \in \mathsf{P}$ satisfaisant pour tout mot x :

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 2/3\,; \\ x \notin A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \notin B) \geqslant 2/3. \end{cases}$$

On peut reformuler cette définition comme suit : il existe un machine de Turing probabiliste fonctionnant en temps polynomial qui reconnaît A avec erreur $\leqslant 1/3$.

Une autre classe intéressante est obtenue lorsqu'on impose à la machine de ne pas faire d'erreur pour les instances hors du langage.

6-C Définition (RP)

La classe RP (pour *Randomized Polynomial time*) est l'ensemble des langages A tels qu'il existe un polynôme $p(n)$ et un langage $B \in \mathsf{P}$ satisfaisant pour tout mot x :

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 2/3\,; \\ x \notin A \implies \forall r \in \{0,1\}^{p(|x|)}\ (x,r) \notin B. \end{cases}$$

On peut reformuler cette définition comme suit : il existe un machine de Turing probabiliste fonctionnant en temps polynomial qui reconnaît A avec erreur $\leqslant 1/3$ si $x \in A$ et sans erreur si $x \notin A$.

Dans le cas de RP, on dit que l'algorithme fait une erreur « d'un seul côté ». Il est évident que l'on a les inclusions $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{BPP}$.

6-D Remarque Souvent la définition de RP est donnée avec une probabilité d'acceptation $\geqslant 1/2$ plutôt que $\geqslant 2/3$ dans le cas où $x \in A$. Nous avons choisi $2/3$ pour la similitude avec BPP. Les deux définitions sont équivalentes (exercice 6-H) puisqu'on peut réduire la probabilité d'erreur arbitrairement comme nous le verrons à la proposition 6-E.

Un peu de recul

Ces deux classes sont appelées « sémantiques » (par opposition à « syntaxiques ») car leur définition dépend d'une condition sur le comportement des machines probabilistes que toutes ne satisfont pas : pour BPP par exemple, il faut en effet que sur toute entrée elles acceptent avec probabilité $\leqslant 1/3$ ou $\geqslant 2/3$. Cette propriété n'est même pas décidable.

6.2.2 Réduction d'erreur

Dans ces deux définitions, le choix du seuil d'erreur 1/3 est arbitraire (tant qu'il est strictement inférieur à 1/2 pour BPP), car on peut réduire la probabilité d'erreur comme le montrent les deux propositions suivantes.

6-E Proposition

La probabilité d'erreur pour les langages de RP peut être réduite à $2^{-n^{O(1)}}$.

Plus précisément, si $A \in \mathsf{RP}$ alors pour toute constante c il existe un langage $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que pour tout mot x,

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 1 - 2^{-|x|^c} ; \\ x \notin A \implies \forall r \in \{0,1\}^{p(|x|)}\ (x,r) \notin B. \end{cases}$$

Idée de la démonstration On répète $O(n^c)$ fois l'algorithme probabiliste : si l'une des exécutions accepte alors $x \in A$, sinon on décrète que $x \notin A$.

Démonstration Soit $A \in \mathsf{RP}$ reconnu par une machine $M(x,r)$, avec erreur $\leqslant 1/3$ quand $x \in A$. On considère l'algorithme probabiliste suivant pour A sur une entrée x de taille n :

- `pour` i de 1 à $N = n^c$ `faire`
  - choisir r au hasard et simuler $M(x,r)$;
- `accepter` si $M(x,r) = 1$ pour au moins une exécution ;
- sinon `rejeter`.

Si $x \notin A$, alors toutes les exécutions rejettent x donc l'algorithme rejette toujours. Si $x \in A$, alors l'algorithme rejette seulement si toutes les exécutions rejettent : puisqu'elles sont indépendantes et que chacune a une probabilité $\leqslant 1/3$ de rejeter, la probabilité qu'elles rejettent toutes est $\leqslant (1/3)^{n^c}$. Au final, la probabilité d'erreur est bien $\leqslant 2^{-n^c}$. □

La preuve pour BPP est légèrement plus subtile car l'erreur peut survenir des deux côtés.

6-F Proposition

La probabilité d'erreur pour les langages de BPP peut être réduite à $2^{-n^{O(1)}}$.

Plus précisément, si $A \in \mathsf{BPP}$ alors pour toute constante c il existe un langage $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que pour tout x,

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 1 - 2^{-|x|^c}\,; \\ x \notin A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \notin B) \geqslant 1 - 2^{-|x|^c}. \end{cases}$$

Idée de la démonstration Comme précédemment, on répète $O(n^c)$ fois l'algorithme probabiliste, mais cette fois on prend la réponse majoritaire.

Démonstration Soit $A \in \mathsf{BPP}$ reconnu par une machine $M(x,r)$ avec erreur $\leqslant 1/3$. Soit $\gamma \in \mathbb{N}$ tel que $(\frac{4}{3\sqrt{2}})^\gamma < 1/2$ (par exemple $\gamma = 12$). On considère l'algorithme probabiliste suivant pour A sur une entrée x de taille n :

- `pour` i de 1 à $N = \gamma n^c$ `faire`
  - choisir r au hasard et simuler $M(x,r)$;
- `accepter` si $M(x,r) = 1$ pour au moins la moitié des exécutions ;
- sinon `rejeter`.

On note X_i la variable aléatoire qui vaut 0 si la i-ème exécution $M(x,r)$ donne la bonne réponse $[x \in A]$ (probabilité $\geqslant 2/3$), et 1 sinon (probabilité $\leqslant 1/3$). Soit $X = \sum_{i=1}^{N} X_i$: l'algorithme ne fait donc pas d'erreur si $X < N/2$. Par les bornes de Chernoff (la version faible donnée à la proposition A-M suffit), on a

$$\Pr(X \geqslant N/2) \leqslant \left(\frac{1+1/3}{\sqrt{2}}\right)^N = \left(\frac{4}{3\sqrt{2}}\right)^{\gamma n^c} < 2^{-n^c}.$$

□

6-G Remarque La version faible des bornes de Chernoff (proposition A-M) permet de réduire la probabilité d'erreur seulement si celle-ci est initialement inférieure à $(\sqrt{2}-1)$. Cependant, en procédant de la même façon on pourrait partir de n'importe quelle probabilité d'erreur $p < 1/2$ en utilisant la version plus forte donnée à la proposition A-N.

✎ **6-H** **Exercice**

Montrer que la définition de RP est inchangée si l'on remplace la constante $2/3$ par toute constante $\alpha \in]0,1[$.

De même, montrer que la définition de BPP est inchangée si l'on remplace la constante $2/3$ par toute constante $\alpha \in]1/2,1[$.

6.2.3 Comparaison avec les classes uniformes

Un peu de recul

Puisqu'en pratique, on sait tirer efficacement des nombres « pseudo-aléatoires » qui semblent suffire pour les algorithmes connus, et étant donné la réduction d'erreur extrême que l'on parvient à obtenir (pour $n = 100$, une erreur 2^{-n} correspond à une probabilité 10^{-30} de se tromper), on considère en général que les problèmes de BPP, et a fortiori de RP, sont résolubles efficacement.

Cependant, dans notre cadre théorique on ne sait toujours pas simuler efficacement de manière déterministe les algorithmes probabilistes.

La meilleure inclusion connue de BPP dans une classe déterministe est l'inclusion évidente suivante.

6-I **Lemme**

$$\mathsf{BPP} \subseteq \mathsf{EXP}$$

Idée de la démonstration Il suffit de simuler la machine BPP le long de tous les chemins possibles et, en fonction du nombre de chemins acceptants, d'évaluer la probabilité que le mot soit accepté. □

Pour RP, accepter une erreur seulement du côté où $x \in A$ (plutôt que $x \notin A$) peut paraître arbitraire. Le choix de cette convention vient du résultat suivant.

6-J **Proposition**

$$\mathsf{RP} \subseteq \mathsf{NP}$$

Démonstration Si $A \in \mathsf{RP}$, en suivant la définition, soit $B \in \mathsf{P}$ et $p(n)$ un polynôme tels que pour tout x :

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 2/3\,; \\ x \notin A \implies \forall r \in \{0,1\}^{p(|x|)}\ (x,r) \notin B. \end{cases}$$

Alors $x \in A$ ssi $\exists r\ (x,r) \in B$. C'est exactement la caractérisation existentielle de NP. □

6-K Remarques

- La classe coRP (l'ensemble des langages dont le complémentaire est dans RP) autorise, quant à elle, l'erreur seulement si $x \notin A$, et on a bien sûr $\mathsf{coRP} \subseteq \mathsf{coNP}$.
- Puisque BPP est clos par complémentaire, on a aussi $\mathsf{coRP} \subseteq \mathsf{BPP}$.
- On ne sait pas si $\mathsf{BPP} \subseteq \mathsf{NP}$ ou si $\mathsf{NP} \subseteq \mathsf{BPP}$ (ou ni l'un ni l'autre), même si nous verrons au chapitre 12 quelques indices laissant penser que $\mathsf{BPP} = \mathsf{RP} = \mathsf{P}$.

✎ **6-L Exercice**

Montrer que $\mathsf{NP} \subseteq \mathsf{BPP}$ implique $\mathsf{NP} = \mathsf{RP}$.

Indication : si $L \in \mathsf{BPP}$, placer $L \in \mathsf{RP}$ en vérifiant si la solution trouvée en est effectivement une.

6.2.4 Théorème de hiérarchie

On ne connaît pas de théorème de hiérarchie très serré pour les classes probabilistes, ce qui vient du fait qu'on ne peut pas décider si une machine a un comportement de type BPP. On peut néanmoins montrer un résultat non trivial en faisant appel à une méthode simple.

De la même manière que BPP, pour une fonction $t : \mathbb{N} \to \mathbb{N}$ on définit $\mathsf{BPTIME}(t(n))$ comme l'ensemble des langages reconnus par une machine de Turing probabiliste fonctionnant en temps $O(t(n))$ avec probabilité d'erreur $\leqslant 1/3$. De plus, $t^{(k)}$ désigne la fonction t itérée k fois. Le meilleur théorème de hiérarchie que l'on connaisse pour les classes probabilistes est alors le résultat suivant dû à Karpinski et Verbeek [KV87].

6-M Théorème (hiérarchie probabiliste en temps)

Soit $k > 0$ un entier fixé. Si $t : \mathbb{N} \to \mathbb{N}$ est une fonction constructible en temps vérifiant $t^{(k)}(n) = 2^{\omega(n)}$, alors $\mathsf{BPTIME}(n) \subsetneq \mathsf{BPTIME}(t(n))$.

Et plus généralement, si $f : \mathbb{N} \to \mathbb{N}$ est constructible en temps, alors $\mathsf{BPTIME}(f(n)) \subsetneq \mathsf{BPTIME}(t(f(n)))$

Idée de la démonstration La preuve utilise la technique de *padding* que l'on a déjà vue (à la proposition 2-AU par exemple) pour multiplier l'écart k fois afin de pouvoir utiliser le théorème de hiérarchie déterministe.

Plus précisément, si $\mathsf{BPTIME}(t(n)) = \mathsf{BPTIME}(n)$ alors pour tout $1 \leqslant i \leqslant k-1$, $\mathsf{BPTIME}(t^{(i+1)}(n)) = \mathsf{BPTIME}(t^{(i)}(n))$ et donc $\mathsf{BPTIME}(t^{(k)}(n)) = \mathsf{BPTIME}(n)$ (cf. figure 6.1). Mais cela contredit le théorème de hiérarchie déterministe puisque $t^{(k)}(n) = 2^{\omega(n)}$ et $\mathsf{BPTIME}(n) \subseteq \mathsf{DTIME}(2^{O(n)})$.

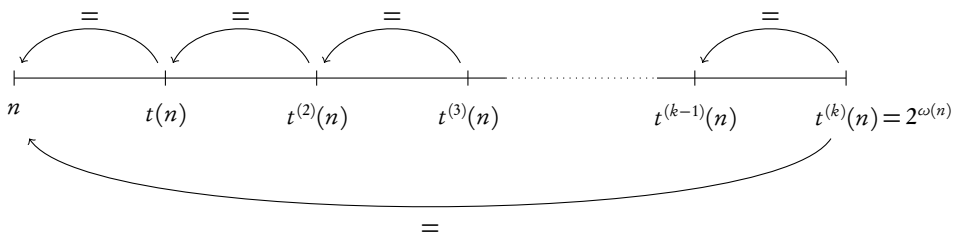


FIGURE 6.1 – Illustration de la preuve du théorème de hiérachie probabiliste en temps : effondrement sous l'hypothèse $\mathsf{BPTIME}(t(n)) = \mathsf{BPTIME}(n)$.

Démonstration Pour alléger les notations, nous ne donnons la démonstration que pour $\mathsf{BPTIME}(n)$ mais celle pour $\mathsf{BPTIME}(f(n))$ est identique (en remplaçant n par $f(n)$).

Supposons que $\mathsf{BPTIME}(t(n)) = \mathsf{BPTIME}(n)$ et montrons que

$$\text{pour tout } i \geqslant 0,\ \mathsf{BPTIME}(t^{(i+1)}(n)) = \mathsf{BPTIME}(t^{(i)}(n)).$$

Démonstration — Soit $A \in \mathsf{BPTIME}(t^{(i+1)}(n))$, on définit sa version *paddée* $A_i = \{(x, 1^{t^{(i)}(|x|)-|x|}) \mid x \in A\}$: la taille de l'entrée est alors $N_i = t^{(i)}(|x|)$.

Puisque t est constructible en temps, il en est de même pour $t^{(i)}$ et on peut donc reconnaître A_i par l'algorithme probabiliste suivant sur l'entrée (x, y) :

- calculer $N_i = t^{(i)}(|x|)$ et rejeter si $y \neq 1^{N_i}$;
- décider si $x \in A$.

Le premier calcul prend un temps linéaire en N_i, tandis que le second calcul prend un temps $t^{(i+1)}(|x|)$ puisque $A \in \mathsf{BPTIME}(t^{(i+1)}(n))$. Au total, l'algorithme fonctionne en temps $O(t(N_i))$ sur une entrée de taille N_i, donc $A_i \in \mathsf{BPTIME}(t(n))$. Ainsi par hypothèse, $A_i \in \mathsf{BPTIME}(n)$.

On en déduit qu'il existe un algorithme probabiliste fonctionnant en temps $O(N_i)$ pour A_i ; le même algorithme permet de décider A en temps $O(N_i) = O(t^{(i)}(n))$. Donc $A \in \mathsf{BPTIME}(t^{(i)}(n))$. ◇

De toutes ces égalités, on déduit donc que

$$\mathsf{BPTIME}(t^{(k)}(n)) = \mathsf{BPTIME}(n) \tag{6.1}$$

Puisque $t^{(k)}(n) = 2^{\omega(n)}$, soit $\alpha(n)$ la fonction tendant vers l'infini telle que $t^{(k)}(n) = 2^{\alpha(n)n}$ (remarque : $2^{\alpha(n)n} = t^{(k)}(n)$ est constructible en temps puisque $t(n)$ l'est). On sait que $\mathsf{BPTIME}(n) \subseteq \mathsf{DTIME}(2^{O(n)})$ en raisonnant comme au lemme 6-I (énumérer tous les choix probabilistes pour simuler de manière déterministe un problème de $\mathsf{BPTIME}(n)$), et donc $\mathsf{BPTIME}(n) \subseteq \mathsf{DTIME}(2^{n\alpha(n)/2})$.
Mais par le théorème de hiérarchie déterministe en temps 2-J,

$$\mathsf{BPTIME}(n) \subseteq \mathsf{DTIME}(2^{n\alpha(n)/2}) \subsetneq \mathsf{DTIME}(2^{n\alpha(n)}) = \mathsf{DTIME}(t^{(k)}(n)) \subseteq \mathsf{BPTIME}(t^{(k)}(n)).$$

On obtient donc $\mathsf{BPTIME}(t^{(k)}(n)) \neq \mathsf{BPTIME}(n)$ ce qui contredit l'égalité (6.1) ci-dessus. □

6-N Remarques

- Les plus petites fonctions t satisfaisant l'hypothèse du théorème $t^{(k)}(n) = 2^{\omega(n)}$ ne s'expriment pas grâce aux fonctions élémentaires habituelles. Il est facile de vérifier par exemple que des fonctions en $2^{n^{o(1)}}$ conviennent, mais que les fonctions $2^{(\log n)^{O(1)}}$ sont trop petites.
- Par ailleurs, ce théorème est aussi valide pour RP par la même preuve.

En réalité cette technique s'applique aussi aux classes déterministes. On obtient ainsi un resserrement du théorème de hiérarchie 2-J que l'on connaît, appliqué ici à la classe $\mathsf{DTIME}(n)$ pour plus de simplicité.

6-O Théorème

Soit $k > 0$ un entier fixé. Alors $\mathsf{DTIME}(n) \subsetneq \mathsf{DTIME}(n(\log n)^{1/k})$.

Plus généralement, si $f : \mathbb{N} \to \mathbb{N}$ est constructible en temps, alors $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(f(n)(\log f(n))^{1/k})$.

Idée de la démonstration La fonction $t : n \mapsto n(\log n)^{1/k}$ vérifie $t^{(k+1)}(n) = \omega(n \log n)$. Si $\mathsf{DTIME}(t(n)) = \mathsf{DTIME}(n)$ alors par *padding*, $\mathsf{DTIME}(t^{(i+1)}(n)) = \mathsf{DTIME}(t^{(i)}(n))$ pour tout i, et donc $\mathsf{DTIME}(t^{(k+1)}(n)) = \mathsf{DTIME}(n)$. Cela contredit le théorème de hiérarchie déterministe en temps 2-J. □

✎ **6-P Exercice**

Faire la démonstration complète de ce théorème.

6.2.5 Circuits et algorithmes probabilistes

Nous verrons au chapitre 12 des liens étroits entre bornes inférieures non uniformes et algorithmes probabilistes. Nous nous contentons ici de montrer que les circuits de taille po-

lynomiale peuvent simuler des algorithmes probabilistes polynomiaux, un résultat d'Adleman [Adl78].

6-Q Théorème (Adleman, 1978)

$$\mathsf{BPP} \subset \mathsf{P/poly}$$

Idée de la démonstration En réduisant suffisamment la probabilité d'erreur d'un algorithme probabiliste, on montre qu'il existe une même suite $r \in \{0,1\}^{p(n)}$ de bits aléatoires donnant la bonne réponse pour *toutes* les entrées x de taille n. Il suffit alors de donner cette suite r en conseil et de simuler de manière déterministe l'algorithme probabiliste le long de cette suite de bits aléatoires.

Démonstration Soit $A \in \mathsf{BPP}$ sur l'alphabet Σ, et on note α une constante telle que $2^\alpha \geqslant |\Sigma|$. En réduisant la probabilité d'erreur par la proposition 6-F, il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que pour toute entrée x de taille n :

$$\begin{cases} x \in A \implies \Pr_{r\in\{0,1\}^{p(n)}}((x,r) \notin B) < 2^{-\alpha n} \leqslant |\Sigma|^{-n}\,; \\ x \notin A \implies \Pr_{r\in\{0,1\}^{p(n)}}((x,r) \in B) < 2^{-\alpha n} \leqslant |\Sigma|^{-n}. \end{cases}$$

En d'autres termes, pour tout x de taille n on a $\Pr_r(x \in A \iff (x,r) \notin B) < |\Sigma|^{-n}$. Puisqu'il y a $|\Sigma|^n$ mots de taille n, on en déduit par l'inégalité de Boole (proposition A-B) que

$$\Pr_r(\exists x \in \Sigma^n \ (x \in A \iff (x,r) \notin B)) < |\Sigma|^n \cdot |\Sigma|^{-n} = 1.$$

Ainsi, $\Pr_r(\forall x \in \Sigma^n \ (x \in A \iff (x,r) \in B)) > 0$ donc il existe $r_0 \in \{0,1\}^{p(n)}$ tel que $\forall x \in \Sigma^n \ (x \in A \iff (x,r_0) \in B)$. C'est exactement la caractérisation d'un langage dans $\mathsf{P/poly}$ avec conseil r_0. □

6.3 Un exemple important

Outre l'exemple introductif de l'algorithme probabiliste de tri rapide, il nous faut décrire un autre problème important que nous retrouverons au chapitre 12.

6.3.1 Test de circuits arithmétiques

La question « $\mathsf{BPP} \neq \mathsf{P}$? », c'est-à-dire savoir si les probabilités accélèrent vraiment le calcul, est toujours ouverte. Mais la grande majorité des problèmes naturels de BPP que l'on connaît sont déjà dans P. À titre d'exemple, le problème PRIMALITÉ (décider si un entier donné en binaire est premier, cf. exemple 2-R) est longtemps resté un problème de BPP candidat à être hors de P ; mais Agrawal, Kayal et Saxena [AKS04] ont finalement montré qu'il peut être décidé en temps polynomial *déterministe*. Nous étudierons

cette question de trouver des algorithmes déterministes pour BPP au chapitre 12 sur la « dérandomisation ».

L'un des rares candidats naturels restant pour être dans BPP \ P concerne les circuits arithmétiques introduits au chapitre précédent (section 5.6).

Soit TIP (Test d'Identité de Polynômes, en anglais *Poynomial Identity Testing* ou PIT) le problème suivant :

- *entrée* : un circuit arithmétique sans constante donné par son code binaire (cf. remarque 5-AP), calculant un polynôme $p \in \mathbb{Z}[x_1, \ldots, x_n]$;
- *question* : le polynôme p est-il identiquement nul ($p \equiv 0$) ?

Ce problème portant sur les polynômes admet une variante plus simple, appelée TIE (Test d'Identité d'Entiers), restreinte aux circuits calculant des entiers :

- *entrée* : un circuit arithmétique sans variable et avec pour seule constante -1, calculant un entier $N \in \mathbb{Z}$;
- *question* : $N = 0$?

6.3.2 Équivalence des deux problèmes

En réalité, ces deux problèmes sont équivalents ; pour le montrer, nous avons besoin du lemme suivant.

6-R Lemme

Si $p \in \mathbb{Z}[x_1, \ldots, x_n]$ est un polynôme de degré d dont les coefficients sont majorés en valeur absolue par M, alors pour tout $\alpha_1, \ldots, \alpha_n \in \mathbb{N}$ vérifiant $\alpha_1 \geqslant M+1$ et $\alpha_{i+1} \geqslant 1 + M(d+1)^i \alpha_i^d$, on a : $p \equiv 0$ ssi $p(\alpha_1, \ldots, \alpha_n) = 0$.

Démonstration Par récurrence sur le nombre de variables n.

Pour $n = 1$: si $p \neq 0$ alors p s'écrit $p(x) = \sum_{i=0}^{d} a_i x^i$ avec $a_d \neq 0$. Alors

$$|p(\alpha_1)| \geqslant \alpha_1^d - |\sum_{i<d} a_i \alpha_1^i| ;$$

or

$$|\sum_{i<d} a_i \alpha_1^i| \leqslant M \sum_{i=0}^{d-1} \alpha_1^i = M(\alpha_1^d - 1)/(\alpha_1 - 1) < \alpha_1^d$$

car $\alpha_1 \geqslant M+1$: donc $p(\alpha_1) \neq 0$.

Pour $n > 1$: soit $p_0, \ldots, p_d \in \mathbb{Z}[x_1, \ldots, x_{n-1}]$ les polynômes tels que

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{d} p_i(x_1, \ldots, x_{n-1}) x_n^i ;$$

si $p \neq 0$ alors il existe i_0 tel que p_{i_0} est non nul. Puisque p_{i_0} est de degré $\leqslant d$ et a ses coefficients majorés en valeur absolue par M, par récurrence $p_{i_0}(\alpha_1,\ldots,\alpha_{n-1}) \neq 0$. Ainsi, $q(x_n) = p(\alpha_1,\ldots,\alpha_{n-1},x_n)$ est un polynôme non nul à une variable et ses coefficients sont les $p_i(\alpha_1,\ldots,\alpha_{n-1})$: ils sont donc majorés en valeur absolue par $M(d+1)^{n-1}\alpha_{n-1}^d$ puisque p_i a au plus $(d+1)^{n-1}$ monômes et ses coefficients sont majorés par M. Par le cas $n=1$,

$$\text{si } \alpha_n \geqslant 1 + M(d+1)^{n-1}\alpha_{n-1}^d, \text{ alors } q(\alpha_n) \neq 0.$$

Ainsi, $p(\alpha_1,\ldots,\alpha_n) \neq 0$. □

Nous pouvons maintenant montrer que TIP et TIE sont équivalents. Puisque TIE est une restriction de TIP, il est clair que TIE se réduit à TIP ; le lemme suivant montre la réciproque.

6-S Lemme

Le problème TIP se réduit au problème TIE pour les réductions many-one polynomiales :

$$\mathtt{TIP} \leqslant_{\mathsf{m}}^{\mathsf{p}} \mathtt{TIE}.$$

Idée de la démonstration On remplace les variables $x_1,\ldots,x_n$ d'une instance de TIP par des entiers croissant suffisamment vite : cela permet de séparer le rôle de chaque variable (tout se passe comme si les entiers étaient « indépendants »).

Démonstration Soit C une instance de TIP de taille t et $p \in \mathbb{Z}[x_1,\ldots,x_n]$ le polynôme calculé par C. Les coefficients de p sont strictement majorés en valeur absolue par $2^{2^{2t}}$ (lemme 5-AT) et son degré par 2^t (lemme 5-AS).

Par le lemme 6-R, $p = 0$ ssi $p(\alpha_1,\ldots,\alpha_n) = 0$ dès que

$$\alpha_1 \geqslant 2^{2^{2t}} \text{ et } \alpha_{i+1} > 2^{2^{2t}}(1+2^t)^i \alpha_i^{2^t}.$$

On peut prendre $\alpha_i = 2^{2^{2it}}$, qui est calculable par $2it$ élévations au carré successives. Ainsi, notre instance de TIE est le circuit arithmétique D calculant l'entier $C(\alpha_1,\ldots,\alpha_n)$ en appliquant C aux nombres α_i calculés par élévations au carré successives : on a alors $C \in \mathtt{TIP}$ ssi $D \in \mathtt{TIE}$ et le code de D est calculable en temps polynomial à partir de celui de C. □

6.3.3 Algorithme probabiliste

Nous donnons maintenant un algorithme probabiliste polynomial pour TIE (et donc pour TIP par la réduction que l'on vient de voir).

6-T **Remarque** Malgré la simplicité du problème TIE, on ne sait pas le résoudre par un algorithme déterministe polynomial. Une tentative naturelle est d'évaluer simplement chaque porte du circuit : cette approche échoue car les entiers calculés peuvent avoir une valeur doublement exponentielle (2^{2^n}), c'est-à-dire qu'ils s'écrivent avec un nombre exponentiel de bits (2^n). On ne peut donc pas les manipuler en temps polynomial.

C'est une question ouverte de savoir si $\mathtt{TIE} \in \mathsf{P}$ (et donc, bien sûr, de même pour TIP).

6-U **Lemme**

$\mathtt{TIE} \in \mathsf{coRP}$, c'est-à-dire qu'il existe un algorithme probabiliste polynomial tel que si un circuit arithmétique C calcule un entier nul, alors C est accepté avec probabilité 1, et sinon C est rejeté avec probabilité $\geqslant 2/3$.

Idée de la démonstration Pour éviter l'explosion de la taille des valeurs calculées, on évalue le circuit modulo un entier aléatoire m.

Démonstration Soit N l'entier calculé par une instance C de TIE de taille n. On ne peut pas calculer N en temps polynomial car il est potentiellement trop grand mais on peut l'évaluer modulo des entiers aléatoires : il suffit de l'évaluer porte par porte modulo l'entier choisi. Voici l'algorithme probabiliste :

- `répéter` $O(n^2)$ fois :
  - choisir $m \in \{2, \ldots, 2^{n^2}\}$ au hasard,
  - évaluer $N \mod m$;
- `rejeter` ssi au moins l'une des évaluations est non nulle.

Si $N = 0$ alors l'algorithme accepte C puisque $N \mod m$ est toujours nul.
Si $N \neq 0$: puisque $N \leqslant 2^{2^n}$ (lemme 5-AT), il a au plus 2^n diviseurs premiers. Or, par le théorème des nombres premiers (théorème A-O), il existe une constante $c > 0$ telle que le nombre de nombres premiers dans l'intervalle $[2, 2^{n^2}]$ est $\geqslant c2^{n^2}/n^2$. Ainsi, si m est choisi au hasard dans $\{2, \ldots, 2^{n^2}\}$,

$$\begin{aligned}\Pr(m \text{ ne divise pas } N) &\geqslant \Pr(m \text{ est premier et ne divise pas } N)\\ &= \Pr(m \text{ ne divise pas } N | m \text{ premier}) \Pr(m \text{ premier})\\ &\geqslant (1 - 2^n/(c2^{n^2}/n^2))c/n^2 \geqslant c/(2n^2).\end{aligned}$$

Par le lemme A-C, en choisissant $4n^2/c$ modulos m (c'est-à-dire en répétant $4n^2/c$ fois la boucle), on a donc une probabilité $\geqslant 2/3$ que l'un des m ne divise pas N : l'algorithme rejette donc C avec probabilité $\geqslant 2/3$.
Ainsi, $\mathtt{TIE} \in \mathsf{coRP}$. □

6.4 Questions ouvertes

Nous terminons ce chapitre par quelques questions ouvertes concernant les classes probabilistes.

6.4.1 Problèmes naturels de BPP

On l'a vu, TIE (ou TIP) est l'un des rares problèmes naturels de BPP pour lesquels on ne connaît pas d'algorithme déterministe polynomial. Ce problème est même dans coRP. Il est assez surprenant de constater qu'on ne dispose pas de réel problème naturel de BPP qu'on ne sache pas mettre dans RP ∪ coRP. On peut bien sûr concevoir des variantes de TIE, comme décider, parmi trois circuits arithmétiques, si deux calculent le même entier. Mais ce genre de problème n'est généralement pas considéré « naturel ». Il y a aussi un problème issu de la théorie des groupes, proposé par Babai, Beals et Seress [BBS09], pour lequel on dispose d'un algorithme BPP mais ni RP ni coRP, mais le peu d'attention qu'il a reçu en fait pour l'instant un candidat de faible valeur.

Comme on le verra au chapitre 12 sur la « dérandomisation », certains indices laissent penser que BPP = P. Si c'est le cas, alors bien sûr tous les problèmes de BPP sont aussi dans RP ∪ coRP et la question précédente est dénuée d'intérêt. Il n'en reste pas moins que le fait qu'une erreur d'un seul côté suffise pour « tous » les problèmes naturels est intriguant.

Un dernier point important est le fait qu'*on ne connaît pas de problème complet* pour BPP (ni pour RP). En effet, comme on l'a vu, ces classes sont *sémantiques* (définies par une condition que toutes les machines probabilistes ne vérifient pas) ce qui empêche de définir le problème complet usuel de simulation de machines : par exemple, le langage

$$\{(\langle M\rangle, x, 1^t) \mid \Pr_{r\in\{0,1\}^t}(M(x,r)=1) \geqslant 2/3, \text{ où } M(x,r) \text{ fonctionne en temps} \leqslant t\}$$

est certes BPP-difficile mais n'a aucune raison d'être dans BPP.

6.4.2 Hiérarchie

Le théorème de hiérarchie 6-M présenté à ce chapitre est le meilleur que l'on connaisse. Savoir si $\mathsf{BPTIME}(n^{\log n}) \neq \mathsf{BPP}$ par exemple est donc encore ouvert. Cela vient encore du fait que BPP est une classe sémantique et qu'on ne peut pas énumérer les machines qui ont un « comportement BPP ». Des théorèmes de hiérarchie meilleurs ont cependant été obtenus en ajoutant une dose de non-uniformité (voir par exemple [FS04] avec un bit de conseil).

6.4.3 Temps exponentiel

Bien qu'on soupçonne que P = BPP (ou du moins que $\mathsf{BPP} \subseteq \mathsf{DTIME}(2^{O(n)})$), ce qui impliquerait bien sûr que EXP ≠ BPP par le théorème de hiérarchie déterministe en

temps, en réalité on ne sait même pas montrer à ce jour (en 2013) que NEXP $\neq$ BPP… La classe NEXP étant considérée comme très grande (on combine la puissance du non-déterminisme avec celle d'un temps de calcul exponentiel !) et BPP petite (classe des problèmes qu'on peut résoudre efficacement), ce manque de résultats incite à une certaine humilité. À noter toutefois que NEXP $\neq$ RP puisque NEXP $\neq$ NP par le théorème de hiérarchie non-déterministe en temps, et que NP $\supseteq$ RP. Cependant, la question « EXP $\neq$ RP ? » reste ouverte elle aussi.

Pour s'exercer sur les classes probabilistes, on trouvera en annexe les exercices B-B, B-D et B-E sur des sujets abordés dans ce chapitre.

7 Oracles et limites de la diagonalisation

La seule technique vue jusqu'à présent pour séparer des classes de complexité (théorèmes de hiérarchie, par exemple) est la diagonalisation. En deux mots, pour montrer qu'une machine M ne peut pas reconnaître le langage L que l'on souhaite définir, il s'agit de simuler M sur une entrée x et de choisir si $x \in L$ de façon à ce que M commette une erreur ($x \in L$ ssi $M(x) = 0$). Si l'on fait cela sur toutes les machines M fonctionnant en temps $O(t(n))$ par exemple, on aura alors montré que $L \notin \mathsf{DTIME}(t(n))$. Dans un premier temps, nous revisiterons les preuves des théorèmes de hiérarchie pour montrer qu'elles entrent bien dans ce cadre. Cette nouvelle présentation des preuves permet de mieux comprendre l'essence de la diagonalisation et de montrer plus facilement des variantes de ces résultats.

En réalité on dispose de peu d'autres techniques de preuve de bornes inférieures. Cependant dans ce chapitre nous verrons une limitation à ce que l'on peut montrer par diagonalisation, grâce au concept *d'oracle*. En particulier, nous verrons que les principales questions ouvertes (comme « $\mathsf{P} = \mathsf{NP}$? ») ne peuvent être résolues au seul moyen d'une diagonalisation « simple » qui simulerait les machines comme des « boîtes noires », c'est-à-dire sans prendre en compte leur fonctionnement intérieur.

Il n'est toutefois pas évident de caractériser ce qu'on entend par « diagonalisation » et ce sera l'objet d'un résultat assez surprenant : pour une certaine définition de cette méthode, toute séparation peut être démontrée par diagonalisation…

Les oracles nous serviront également au chapitre suivant pour définir de nouvelles classes de complexité, à savoir les différents niveaux de la hiérarchie polynomiale.

7.1 Théorèmes de hiérarchie

Avant de pouvoir revisiter les théorèmes des hiérarchie que nous avons vus, il nous faut revenir sur le concept d'énumération de machines de Turing.

7.1.1 Énumération des machines

La preuve des théorèmes de hiérarchie que nous avons vus ne fait pas apparaître clairement le processus de diagonalisation. Pour cela, il est utile de travailler avec des énumérations de machines de Turing comme celle de la remarque 3-AL.

7-A Définition

Une *énumération de machines* pour une classe de complexité $\mathscr{C}$ est une suite $(M_i)_{i\in\mathbb{N}}$ de codes de machines de Turing telle que :

- pour tout $i \in \mathbb{N}$, le langage reconnu par M_i appartient à $\mathscr{C}$;
- pour tout $A \in \mathscr{C}$, il existe i tel que M_i reconnaît A.

De la même façon qu'à la remarque 3-AL, il existe des énumérations de machines pour $\mathsf{DTIME}(t(n))$, $\mathsf{NTIME}(t(n))$ et $\mathsf{DSPACE}(s(n))$ si $t(n)$ et $s(n)$ sont constructibles respectivement en temps et en espace. Pour cela, il suffit de prendre une énumération (M_i) des machines de Turing et d'y adjoindre un nouveau ruban comptant le nombre d'opérations ou l'espace utilisé : on arrête alors le calcul de la machine M_i dès que $it(n)$ étapes ont été réalisées, ou $is(n)$ cases visitées (on pourrait aussi prendre $(\log i)t(n)$ étapes ou $f(i)t(n)$ pour toute fonction f tendant vers l'infini). On obtient ainsi une nouvelle énumération (M'_i) pour la classe concernée.

✎ **7-B Exercice**

En supposant que le code de M_i est simplement i écrit en binaire (un code incorrect étant considéré comme la machine rejetant toujours), montrer que calculer le code de M'_i à partir de i donné en unaire se fait en temps $O(i)$ et en espace $O(\log i)$.

Indication : il s'agit de calculer i en binaire et d'ajouter au code ainsi obtenu le code du ruban comptant les étapes.

7-C Remarque Nous nous intéressons aux énumérations *efficaces* où le code de M_i est calculable facilement à partir de i comme à l'exercice précédent.

Néanmoins, toutes les classes ne possèdent pas d'énumération efficace : par exemple, il n'y a aucune énumération calculable de la classe probabiliste BPP que nous verrons au chapitre suivant, car décider si une machine a un comportement « de type BPP » est indécidable.

7.1.2 Hiérarchie déterministe

Contrairement aux preuves des théorèmes de hiérarchie données précédemment, la diagonalisation peut être complètement explicitée comme nous allons le voir. Nous allons

prouver deux variantes du théorème de hiérarchie en temps déterministe : une version « infiniment souvent » et une autre « presque partout ». La première n'est autre qu'une version plus faible du théorème 2-J puisque la différence entre f et g doit être plus grande ($(\log f(n))(\log n)$ au lieu de $\log f(n)$) [1].

7-D Théorème (hiérarchie en temps déterministe infiniment souvent)

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en temps telles que $f(n) \neq 0$ (pour tout n) et

$$f(n)(\log f(n))(\log n) = O(g(n)).$$

Alors $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.

Cette version est appelée « infiniment souvent » car le langage $A \in \mathsf{DTIME}(g(n))$ construit vérifie : pour tout $B \in \mathsf{DTIME}(f(n))$, pour *une infinité* de $n \in \mathbb{N}$, $B^{=n} \neq A^{=n}$ (l'infinité de n venant de ce que chaque machine M_i a une infinité de machines équivalentes dans l'énumération).

Idée de la démonstration On construit le langage A mot par mot, spécifiquement pour que chaque machine M_i d'une énumération des machines fonctionnant en temps $O(f(n))$ se trompe au moins sur le mot 0^i (cf. figure 7.1).

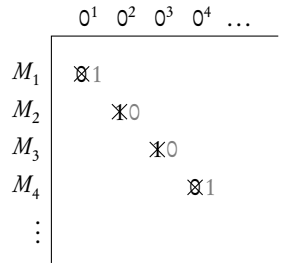


FIGURE 7.1 – Diagonalisation sur le résultat des calculs $M_i(0^i)$.

Démonstration On peut supposer sans perte de généralité que $f(n) = \Omega(n)$: sinon $f \leqslant \alpha$ pour une constante α et $g(n) = \Omega(n)$ car elles sont constructibles en temps (cf. exercice 2-I). Alors le langage $A = 0^\star$ (contenant exactement les mots 0^i pour $i \in \mathbb{N}$) est dans $\mathsf{DTIME}(g(n))$ mais pas dans $\mathsf{DTIME}(f(n))$.

Soit (M_i) une énumération des machines fonctionnant en temps $(\log i)f(n)$. Nous allons définir un langage $A \in \mathsf{DTIME}(g(n))$ sur l'alphabet $\{0,1\}$ qui ne soit reconnu par aucune des machines M_i. Contrairement à d'habitude, nous n'allons pas définir A par une simple expression ensembliste, mais nous allons plutôt indiquer pour chaque mot s'il appartient à A ou non.

1. Une autre différence est qu'ici les deux fonctions f et g sont constructibles en temps, et pas seulement la fonction g.

– Aucun mot x contenant le symbole 1 n'appartient à A (c'est-à-dire que $A \subseteq 0^\star$).
– Le mot 0^i appartient à A ssi $M_i(0^i) = 0$.

Le langage A ainsi défini ne contient que des mots de la forme 0^n.
Supposons que $A \in \mathsf{DTIME}(f(n))$: il est donc reconnu par une machine M_i de l'énumération. Mais alors $0^i \in A$ ssi $M_i(0^i) = 0$, une contradiction. Donc $A \notin \mathsf{DTIME}(f(n))$.
Il reste à montrer que $A \in \mathsf{DTIME}(g(n))$. Voici un algorithme pour A sur une entrée x de taille n :

– si $x \neq 0^n$, `rejeter` ;
– `accepter` ssi $M_n(0^n) = 0$.

Il s'agit donc de construire le code de M_n puis de simuler cette machine sur 0^n. Or construire le code de M_n prend un temps $O(n)$ comme on l'a vu à l'exercice 7-B (donc un temps $O(g(n))$ puisque $g(n) = \Omega(n)$) ; la machine M_n fonctionne en temps $(\log n)f(n)$ donc la simulation par la machine universelle du théorème 1-S prend un temps majoré par

$$\alpha(\log n)f(n)(\log\log n + \log f(n)) \leqslant 2\alpha f(n)(\log f(n))(\log n) = O(g(n)).$$

Au total, $A \in \mathsf{DTIME}(g(n))$. □

La version « presque partout » vérifie quant à elle que A et B diffèrent pour *presque tout* n comme on le voit dans l'énoncé suivant.

7-E Théorème (hiérarchie en temps déterministe presque partout)

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en temps telles que $f(n) \neq 0$ (pour tout n) et

$$f(n)(\log f(n))(\log n) = O(g(n)).$$

Alors il existe $A \in \mathsf{DTIME}(g(n))$ tel que pour tout $B \in \mathsf{DTIME}(f(n))$, il existe $m \in \mathbb{N}$ tel que $\forall n \geqslant m$, $B^{=n} \neq A^{=n}$.

Idée de la démonstration La démonstration ressemble à celle du théorème précédent mais on utilise davantage de mots (et non seulement 0^i) pour diagonaliser sur plus d'une machine par taille.

Démonstration De même qu'à la proposition précédente, on peut supposer $f(n) = \Omega(n)$ (sinon le langage $A = 0^\star$ convient).
Soit (M_i) une énumération des machines fonctionnant en temps $(\log i)f(n)$. Nous allons définir un langage $A \in \mathsf{DTIME}(g(n))$ sur l'alphabet $\{0,1\}$ telle que chaque machine M_i se trompe pour presque toute longueur de mot.
Pour une taille d'entrée n fixée, on note $x_0 < x_1 < \cdots < x_{2^n-1}$ les mots de $\{0,1\}^n$ ordonnés par ordre lexicographique. On définit alors A comme suit sur une entrée x de taille n :

– si $x = x_i$ pour $i \geqslant n$ alors $x \notin A$;
– si $x = x_i$ pour $0 \leqslant i < n$ alors $[x \in A$ ssi $M_i(x_i) = 0]$.

Le langage A ainsi défini contient au plus n mots de taille n.
Soit $B \in \mathsf{DTIME}(f(n))$, reconnu par une machine M_i de l'énumération. Alors pour toute taille d'entrée $n > i$, $x_i \in A$ ssi $M_i(x_i) = 0$ ssi $x_i \notin B$. Ainsi, pour tout $n > i$, $B^{=n} \neq A^{=n}$.
Il reste à montrer que $A \in \mathsf{DTIME}(g(n))$. Voici un algorithme pour A sur une entrée x de taille n :

– calculer i tel que $x = x_i$;
– si $i \geqslant n$, `rejeter` ;
– `accepter` ssi $M_i(x_i) = 0$.

La première étape revient simplement à supprimer les éventuels zéros de tête de l'entrée x (puisque i est simplement le rang en binaire de x). Le test $i \geqslant n$ est simple. Il reste donc à construire le code de M_i puis à simuler cette machine sur x. Construire le code de M_i prend un temps $O(i) = O(n)$ comme on l'a vu à l'exercice 7-B (donc un temps $O(g(n))$ puisque $g(n) = \Omega(n)$) ; la machine M_i fonctionne en temps $(\log i) f(n)$ donc la simulation par la machine universelle du théorème 1-S prend un temps majoré par

$$\alpha(\log i) f(n)(\log\log i + \log f(n)) \leqslant 2\alpha f(n)(\log f(n))(\log n) = O(g(n)).$$

Au total, $A \in \mathsf{DTIME}(g(n))$. □

7.2 Machines de Turing à oracle

Une notion qui nous vient de la calculabilité est celle d'oracle, qui est entre autre utile pour dépasser la limite de ce qui est calculable. Il s'agit d'un langage L que la machine de Turing peut interroger à tout moment sur un mot u : elle obtient ainsi en une étape la réponse à la question « $u \in L$? », quelle que soit la complexité de L (d'où le nom *oracle*). Plus précisément, on obtient la définition suivante.

7-F Définition (machine de Turing à oracle)

– Une *machine de Turing à oracle* est une machine de Turing M équipée d'un ruban supplémentaire spécial appelé *ruban d'oracle* et de trois états spéciaux $q_?$, q_{oui} et q_{non}.
– Lorsqu'on adjoint à M (qui travaille sur l'alphabet Γ) un langage $L \subseteq \Gamma^\star$ appelé *oracle*, ce que l'on note M^L, le fonctionnement de M^L sur une entrée x (noté $M^L(x)$) est le suivant :

- tant que M^L n'entre pas dans l'état $q_?$, son fonctionnement est celui d'une machine standard (en particulier, elle utilise le ruban d'oracle comme un ruban normal, de même les états q_{oui} et q_{non} sont utilisés normalement) ;
- lorsque M^L entre dans l'état $q_?$, le mot u inscrit à ce moment sur le ruban d'oracle est considéré :
  - si $u \in L$ alors M^L passe dans l'état q_{oui},
  - si $u \notin L$ alors M^L passe dans l'état q_{non},

  et les têtes de lecture ne bougent pas ni ne changent le contenu des rubans.

– De manière standard, le mot x est accepté si $M^L(x)$ arrive à un état acceptant. Le temps d'exécution est compté usuellement, le passage de $q_?$ à q_{oui} ou q_{non} prenant une seule étape.

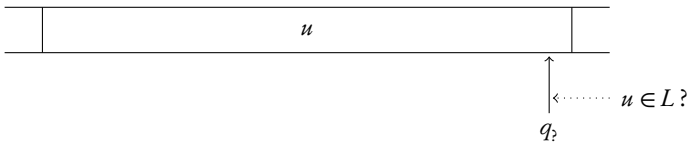


FIGURE 7.2 – Ruban d'oracle d'une machine de Turing avec oracle L.

7-G **Exemple** Voici une machine de Turing déterministe avec oracle SAT, fonctionnant en temps polynomial, qui calcule par recherche préfixe la plus petite affectation (si elle existe) satisfaisant une formule $\varphi(x_1, \ldots, x_n)$:

– $a \leftarrow \epsilon$;
– `tant que` $|a| < n$ `faire`
  - demander à l'oracle SAT si $\varphi(a, 0, x_{|a|+2}, \ldots, x_n)$ est satisfaisable :
    - si oui, $a \leftarrow a0$
    - sinon, $a \leftarrow a1$;

– `renvoyer` a.

Le temps mis par cette machine déterministe est linéaire en n. Si φ n'est pas satisfaisable, alors elle renvoie $a = 1 \ldots 1$.

On peut de même concevoir des machines non déterministes avec oracle comme dans l'exemple ci-dessous.

7-H **Exemple** Si L est un langage quelconque sur l'alphabet Σ, le code ci-dessous est une machine non déterministe avec oracle L, fonctionnant en temps polynomial, qui reconnaît le langage $A_L = \{1^n \mid L^{=n} \neq \emptyset\}$. Sur l'entrée x :

- si $x \neq 1^{|x|}$, `rejeter` ;
- `deviner` $y \in \Sigma^{|x|}$;
- demander à l'oracle si $y \in L$:
  - si oui, `accepter`,
  - sinon, `rejeter`.

Grâce à la définition de machine à oracle, on définit aussi des classes de complexité avec oracle.

7-I **Définition (relativisation)**

- Si A est un langage, la classe $\mathsf{DTIME}(t(n))^A$ (respectivement $\mathsf{NTIME}(t(n))^A$) est l'ensemble des langages reconnus en temps $O(t(n))$ par une machine de Turing déterministe (resp. non déterministe) avec oracle A.
- Si $\mathscr{C}$ est une classe de complexité,

$$\mathsf{DTIME}(t(n))^{\mathscr{C}} = \bigcup_{A\in\mathscr{C}} \mathsf{DTIME}(t(n))^A \quad \text{et}$$

$$\mathsf{NTIME}(t(n))^{\mathscr{C}} = \bigcup_{A\in\mathscr{C}} \mathsf{NTIME}(t(n))^A.$$

- On définit comme d'habitude

$$\mathsf{P}^A = \bigcup_k \mathsf{DTIME}(n^k)^A, \quad \mathsf{NP}^A = \bigcup_k \mathsf{NTIME}(n^k)^A \quad \text{et}$$

$$\mathsf{P}^{\mathscr{C}} = \bigcup_{A\in\mathscr{C}} \mathsf{P}^A, \quad \mathsf{NP}^{\mathscr{C}} = \bigcup_{A\in\mathscr{C}} \mathsf{NP}^A ;$$

  de même pour EXP et NEXP.
- Le fait d'adjoindre des oracles à des classes de complexité s'appelle *relativisation*.

7-J **Remarques**

- On trouve aussi parfois les notations $\mathsf{P}(A)$, $\mathsf{P}(\mathscr{C})$, etc.
- Nous nous contentons ici de définir la relativisation des classes en temps déterministe et non déterministe car il n'y a pas de définition standard pour une classe de complexité quelconque : en effet, la notion de relativisation dépend des machines définissant la classe.

Par exemple, pour l'espace, le concept varie selon que l'on compte les cases du ruban d'oracle dans l'espace ou non. On définira des relativisations spécifiques à d'autres classes lorsqu'on en aura besoin.

✎ **7-K** **Exercice**

Soit A un langage. Montrer que NP^A admet la caractérisation existentielle habituelle (proposition 2-AO) mais à partir d'un langage de P^A plutôt que de P.

7-L **Remarque** Nous prendrons comme convention que les classes définies à partir de P, comme BPP par exemple (cf. définition 6-B), relativisent avec un oracle A en changeant P par P^A dans leur définition.

Dans le cas de BPP, cela requiert en particulier que la machine probabiliste ait un comportement de type BPP seulement avec l'oracle A, mais pas forcément sans oracle.

7.3 Quelques résultats pour se faire la main

Pour s'habituer à la notion d'oracle, nous allons voir dans cette section quelques résultats simples qui nous resserviront par la suite. Le premier lemme montre qu'un oracle calculable en temps polynomial ne sert à rien pour les classes usuelles.

7-M **Lemme**

Soit $\mathscr{C}$ l'une des classes P, NP, EXP ou NEXP. Alors $\mathscr{C}^{\mathsf{P}} = \mathscr{C}$.

Démonstration Si l'oracle est $A \in \mathsf{P}$, on peut simuler le calcul de $\mathscr{C}^A$ sans oracle, puisque sur une question de la forme « $a \in A$? » il suffit de décider la réponse sans oracle, en temps polynomial en $|a|$.

Si $\mathscr{C} = \mathsf{P}$ ou NP, $|a|$ est polynomiale donc le temps de calcul de la réponse est polynomial et on reste dans la classe $\mathscr{C}$.

Si $\mathscr{C} = \mathsf{EXP}$ ou NEXP, $|a|$ est $2^{n^{O(1)}}$ donc le temps de calcul de la réponse est $2^{n^{O(1)}}$ et on reste dans la classe $\mathscr{C}$. □

Le résultat suivant illustre le fait qu'une machine EXP peut demander à son oracle une question de taille exponentielle, ce qui lui confère beaucoup de puissance. La preuve utilise un argument de *padding* comme on en a déjà vu à la proposition 2-AU par exemple.

7-N Lemme

$\mathsf{NEXP} \subseteq \mathsf{EXP}^{\mathsf{NP}}$.

Démonstration Soit $A \in \mathsf{NEXP}$ reconnu par une machine non déterministe en temps 2^{n^k}. Alors la version « délayée » $\tilde{A} = \{(x, 1^{2^{|x|^k}}) \mid x \in A\}$ est dans NP (padding). Voici une machine EXP avec oracle $\tilde{A}$ pour reconnaître le langage A sur l'entrée x :

- calculer $u = (x, 1^{2^{|x|^k}})$;
- `accepter` ssi $u \in \tilde{A}$.

Cette machine fonctionne en temps 2^{n^k} (le temps de calculer u) avec oracle NP, donc $A \in \mathsf{EXP}^{\mathsf{NP}}$. □

Lorsqu'une classe admet un langage complet, donner ce langage en oracle suffit pour avoir toute la puissance de la classe, comme on le voit grâce au lemme suivant.

7-O Lemme

Si A est un problème $\mathscr{C}$-complet pour les réductions many-one en temps polynomial, alors $\mathsf{P}^{\mathscr{C}} = \mathsf{P}^A$, $\mathsf{NP}^{\mathscr{C}} = \mathsf{NP}^A$, $\mathsf{EXP}^{\mathscr{C}} = \mathsf{EXP}^A$ et $\mathsf{NEXP}^{\mathscr{C}} = \mathsf{NEXP}^A$.

Démonstration Nous montrons le résultat seulement pour NP, les autres cas étant similaires. Soit $B \in \mathsf{NP}^{\mathscr{C}}$: par définition, $B \in \mathsf{NP}^C$ pour un langage $C \in \mathscr{C}$. On appelle N la machine non déterministe polynomiale avec oracle C pour B.

Par $\mathscr{C}$-complétude de A, il existe une réduction polynomiale f telle que

$$x \in C \iff f(x) \in A.$$

Voici donc le fonctionnement d'une machine NP avec oracle A pour B sur l'entrée x : simuler le fonctionnement de $N^C(x)$; remplacer les questions à l'oracle « $u \in C$? » par le calcul de $f(u)$ et la question « $f(u) \in A$? ». Puisque f est calculable en temps polynomial, on reste dans NP ; et la machine accepte bien B par définition de la réduction f. □

Le résultat suivant montre qu'un oracle PSPACE donne la même puissance à P et NP.

7-P Lemme

$$\mathsf{P}^{\mathsf{PSPACE}} = \mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE}$$

Démonstration Il est clair que $\mathsf{PSPACE} \subseteq \mathsf{P}^{\mathsf{PSPACE}} \subseteq \mathsf{NP}^{\mathsf{PSPACE}}$, donc il suffit de montrer que $\mathsf{NP}^{\mathsf{PSPACE}} \subseteq \mathsf{PSPACE}$.

Soit N une machine NP avec oracle $A \in \mathsf{PSPACE}$. Son fonctionnement peut être simulé dans PSPACE ainsi sur l'entrée x :

- énumérer tous les chemins de $N^A(x)$ et pour chacun d'eux :
  - dès que N demande à son oracle une question « $u \in A$? », décider si $u \in A$ et continuer le calcul en conséquence ;
- `accepter` ssi l'un des chemins a accepté.

Puisque N pose à son oracle A seulement des questions de taille polynomiale, décider si $u \in A$ se fait en espace polynomial. Chaque chemin est de taille polynomiale donc l'énumération se fait aussi en espace polynomial. Au total, la machine N^A est simulée en espace polynomial. □

Le dernier résultat est un peu plus subtil. Avant de l'aborder, il convient de comprendre pourquoi il n'est pas évident (et probablement faux) que $\mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}$. Le point crucial est que l'oracle « peut être utilisé à la fois positivement et négativement » : c'est-à-dire que ses réponses peuvent (bien sûr) être positives ou négatives. En termes de certificats, une réponse positive ne pose pas de problème puisqu'il existe un certificat, mais une réponse négative n'a pas de certificat et on ne peut donc vérifier qu'il s'agit de la bonne réponse, ce qui pose problème pour mettre le langage dans NP (cf. figure 7.3). C'est la même différence qu'entre NP et coNP (d'ailleurs, on a bien sûr $\mathsf{coNP} \subseteq \mathsf{NP}^{\mathsf{NP}}$). Cette remarque sera à la base du prochain chapitre sur la hiérarchie polynomiale.

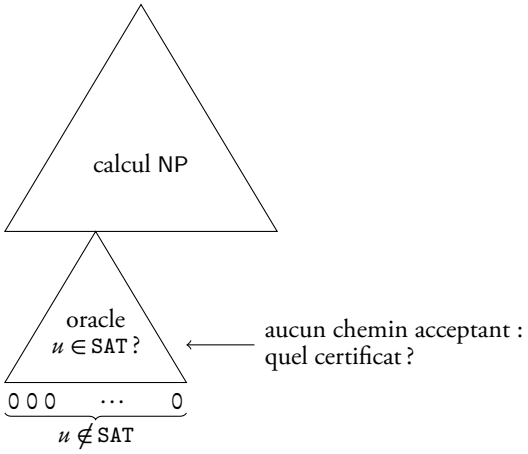


FIGURE 7.3 – Tentative de simulation d'une machine NP avec oracle NP.

Le lemme suivant montre que si NP est clos par complément, alors en effet $\mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}$.

7-Q Lemme

Si $\mathsf{NP} = \mathsf{coNP}$ alors $\mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}$.

Idée de la démonstration Afin de simuler dans NP le calcul de l'oracle, on utilisera le certificat issu de NP lorsque la réponse de l'oracle est positive, et celui issu de coNP lorsqu'elle est négative.

Démonstration Soit $L \in \mathsf{NP}^{\mathsf{NP}}$ décidé par une machine non déterministe polynomiale N avec oracle $A \in \mathsf{NP}$. Puisque $\mathsf{NP} = \mathsf{coNP}$, A possède les deux caractérisations suivantes, où $p(n)$ est un polynôme et B, B' deux langages de P :

- $x \in A$ ssi $\exists y \in \{0,1\}^{p(|x|)}\ (x,y) \in B$;
- $x \notin A$ ssi $\exists y' \in \{0,1\}^{p(|x|)}\ (x,y') \in B'$.

Lors du calcul $N^A(x)$, sur chaque chemin de calcul u un certain nombre de questions $a_1, \ldots, a_k$ sont posées à l'oracle A. Pour décider dans NP si $x \in B$, il s'agit de deviner un chemin acceptant de la machine N, de deviner les réponses de l'oracle le long de ce chemin, et pour chaque question a_i de deviner le certificat y ou y' selon si $a_i \in A$ ou $a_i \notin A$.
Le calcul $N^A(x)$ prend un temps $q(|x|)$ pour un certain polynôme $q(n)$, et il fait donc au plus $q(|x|)$ appels à son oracle. Les certificats pour B auront alors la forme

$$(u, b_1, y_1, y'_1, \ldots, b_{q(n)}, y_{q(n)}, y'_{q(n)}),$$

où u est un mot de taille $q(n)$ (un chemin de N), y_i et y'_i sont des mots de taille $p(q(n))$ (un certificat pour A ou ${}^cA$ pour un mot de taille $q(n)$), et b_i sont des bits.
Voici la machine NP pour L sur l'entrée x :

- deviner $(u, b_1, y_1, y'_1, \ldots, b_{q(n)}, y_{q(n)}, y'_{q(n)})$;
- simuler $N^A(x)$ le long du chemin u en répondant aux questions $a_1, \ldots, a_{q(n)}$ posées à l'oracle par $b_1, \ldots, b_{q(n)}$;
- pour tout $1 \leqslant i \leqslant q(n)$, vérifier que :
  - si $b_i = 1$ alors $(a_i, y_i) \in B$,
  - si $b_i = 0$ alors $(a_i, y'_i) \in B'$;
- `accepter` ssi toutes ces vérifications sont positives et le chemin simulé accepte.

Cette machine non déterministe fonctionne bien en temps polynomial.
Si $x \in L$ alors il existe un chemin u de $N^A(x)$ acceptant ; le long de ce chemin, des questions $a_1, \ldots, a_{q(n)}$ sont posées à A, donnant des réponses $b_1, \ldots, b_{q(n)}$; si $b_i = 1$ alors $a_i \in A$ donc il existe un certificat y_i tel que $(a_i, y_i) \in B$, et si $b_i = 0$ alors $a_i \notin A$ donc il existe un certificat y'_i tel que $(a_i, y'_i) \in B'$. Ainsi, la machine NP décrite ci-dessus a un chemin acceptant.

Réciproquement, s'il existe un chemin acceptant $(u, b_1, y_1, y'_1, \ldots, b_{q(n)}, y_{q(n)}, y'_{q(n)})$, cela signifie que toutes les réponses b_i devinées sont correctes car elles sont certifiées par les certificats y_i ou y'_i, et donc le chemin u simulé correspond à un vrai chemin acceptant de $N^A(x)$. Donc $x \in L$. □

✎ **7-R** **Exercice**

Soit A un langage. Montrer que $\mathsf{E}^A = \mathsf{E}$ ssi $A \in \mathsf{P}$.

Indication : padding…

✎ **7-S** **Exercice**

Montrer que $\mathsf{BPP}^{\mathsf{BPP}} = \mathsf{BPP}$.

Indication : penser à la réduction d'erreur.

7.4 Langages creux et réduction Turing

7.4.1 Langages creux

Voici un exemple intéressant de relativisation de la classe P dû à Berman et Hartmanis [BH77]. Nous rappelons la définition 3-AP d'un langage creux.

7-T **Définition**

Un langage A est dit *creux* s'il existe un polynôme $p(n)$ tel que pour tout $n \geqslant 0$, $|A^{=n}| \leqslant p(n)$. En d'autres termes, le nombre de mots de A de même taille est polynomialement borné.

7-U **Proposition**

Si Creux désigne l'ensemble des langages creux, alors

$$\mathsf{P/poly} = \mathsf{P}^{\mathsf{Creux}}.$$

Idée de la démonstration Si $L \in \mathsf{P}^S$ où S est creux, la machine avec oracle S demande des mots de S de taille majorée par un polynôme $p(n)$ car elle fonctionne en temps

polynomial. Il suffit alors de donner comme conseil la liste des mots de S de taille $\leqslant p(n)$ pour connaître les réponses de l'oracle.
Si $L \in \mathsf{P/poly}$ est reconnu grâce à une famille de conseils (a_n) de taille polynomiale $p(n)$, il suffit d'encoder a_n dans les $p(n)$ premiers mots de taille n d'un langage creux.

Démonstration Soit $L \in \mathsf{P}^S$ où S est un langage creux : il est reconnu par une machine déterministe M avec oracle S fonctionnant en temps polynomial $p(n)$. Cette machine ne peut demander à son oracle que des mots de taille $\leqslant p(n)$. Puisque S est creux, le nombre de mots de S de taille $\leqslant p(n)$ est polynomial : on peut en donner la liste dans un conseil a de taille polynomiale. On obtient alors la machine déterministe polynomiale suivante pour décider L avec conseil a sur l'entrée x :

- simuler $M^S(x)$;
- lorsque M fait appel à son oracle sur un mot u, parcourir la liste du conseil a et répondre 1 si u est dans la liste et 0 sinon ;
- accepter ssi la simulation accepte.

Cette machine fonctionne en temps polynomial et simule exactement $M^S(x)$ si le conseil contient la liste des mots de S de taille $\leqslant p(n)$. Ainsi, $L \in \mathsf{P/poly}$.
Réciproquement, si $L \in \mathsf{P/poly}$, il est reconnu par une machine M avec conseils $(a^{(n)})$ de taille polynomiale. Soit n fixé suffisamment grand pour que $|a^{(n)}| \leqslant 2^n$. Pour alléger les notations, on note $a = a_1 a_2 \dots a_{p(n)}$ le conseil $a^{(n)}$. Les mots de taille n sont ordonnés $x_1 < x_2 < \cdots < x_{2^n}$. On définit alors le langage S suivant pour les mots de taille n :

- pour $i \leqslant |a|$, $x_i \in S$ ssi $a_i = 1$ (le i-ème bit du conseil a) ;
- pour $i > |a|$, $x_i \notin S$.

On remarquera que $|a| \leqslant 2^n$ donc il y a assez de mots de taille n pour satisfaire cette définition. On notera n_0 un entier à partir duquel $|a^{(n)}| \leqslant 2^n$. Pour toutes les entrées de taille $< n_0$, leur appartenance au langage sera codée en dur dans la machine.
Puisque $(a^{(n)})$ est de taille polynomiale, le langage S ainsi défini est creux car le nombre de mots de taille n de S est au plus $|a^{(n)}|$. Grâce à S en oracle, on peut maintenant décider L sur l'entrée x de taille n avec une machine à oracle M'^S :

- si $n < n_0$, répondre directement grâce au code de la machine M' dans lequel est codé en dur la réponse pour tous les x de taille $< n_0$;
- sinon, demander à l'oracle S ses réponses sur les $p(n)$ premiers mots de taille n : on note $a_1, \dots, a_{p(n)}$ ces réponses ;
- simuler $M(x)$ avec conseil $a = a_1, \dots, a_{p(n)}$.

Le langage S nous a permis de retrouver le conseil a pour les mots de taille n et simuler la machine M avec le bon conseil. Cette machine déterministe avec l'oracle S reconnaît L en temps polynomial, donc $L \in \mathsf{P}^S$. □

✎ **7-V** **Exercice**

Un langage L est dit *unaire* si $L \subseteq 1^*$ (les mots de L ne sont composés que de symboles 1). On note Unaires l'ensemble des langages unaires.

Montrer que $\mathsf{P}^{\mathsf{Unaires}} = \mathsf{P}^{\mathsf{Creux}} = \mathsf{P/poly}$.

7.4.2 Réduction Turing polynomiale

Le résultat précédent peut être reformulé grâce à un nouveau type de réductions.

7-W **Définition (réduction Turing polynomiale)**

Si A et B sont deux langages, on dit que A se réduit à B par une *réduction Turing polynomiale,* et on note $A \leqslant_T^p B$, si $A \in \mathsf{P}^B$. En d'autres termes, il existe un algorithme polynomial qui permet de résoudre A s'il a accès à B comme oracle.

7-X **Proposition**

La réduction Turing est plus puissante que la réduction many-one, c'est-à-dire

$$A \leqslant_{\mathsf{m}}^{\mathsf{p}} B \implies A \leqslant_T^p B.$$

Démonstration Si $A \leqslant_{\mathsf{m}}^{\mathsf{p}} B$ alors il existe une fonction f calculable en temps polynomial telle que $x \in A \iff f(x) \in B$. On a alors $A \in \mathsf{P}^B$ via la machine déterministe polynomiale suivante avec oracle B, sur l'entrée x :

- calculer $f(x)$;
- demander à l'oracle si $f(x) \in B$;
- `accepter` ssi $f(x) \in B$.

□

On déduit de cette définition une reformulation de la proposition 7-U.

7-Y **Corollaire**

Un langage A est dans $\mathsf{P/poly}$ si et seulement s'il se réduit à un langage creux par une réduction Turing polynomiale.

7.5 Relativisation

7.5.1 Diagonalisation, oracles et la question « P = NP ? »

Certains résultats ou certaines preuves sont toujours valides si l'on adjoint à toutes les machines un même oracle A. On dit alors que la preuve ou le résultat est « relativisable » (ou encore qu'il *relativise*, ou qu'il passe à l'oracle). C'est le cas lorsque la preuve considère les machines comme des « boîtes noires », c'est-à-dire qu'elle ne se préoccupe pas du fonctionnement interne de la machine car une simple simulation pas à pas est effectuée sans chercher à comprendre son comportement : cette simulation reste valide si on adjoint aux machines un oracle A.

✎ **7-Z Exercice**

En remplaçant dans les preuves chaque machine M par une machine avec oracle M^A, vérifier que les théorèmes de hiérarchie relativisent : par exemple, pour tout oracle A, $\mathsf{EXP}^A \neq \mathsf{P}^A$.

Un peu de recul

Il est généralement considéré qu'une « preuve par diagonalisation » relativise toujours car le comportement interne de la machine n'est pas pris en compte : celle-ci est simplement simulée sans plus de subtilités. Cela implique qu'une question qui ne relativise pas comme « P = NP ? » (cf. ci-dessous) ne pourrait pas être résolue par une simple diagonalisation. Néanmoins, définir formellement la notion de « preuve par diagonalisation » n'est pas chose aisée comme nous le verrons à la fin du chapitre.

L'intérêt des circuits était justement de faciliter l'étude du comportement interne des machines car ce comportement se « voit » dans la structure du circuit. L'espoir était ainsi de pouvoir dépasser la barrière de la relativisation, mais pour l'instant celui-ci s'est révélé vain.

Le théorème suivant montrant que la question « P = NP ? » ne relativise pas est de Baker, Gill et Solovay [BGS75].

7-AA Théorème (Baker, Gill et Solovay, 1975)

Il existe deux oracles A et B tels que

$$\mathsf{P}^A = \mathsf{NP}^A \quad \text{et} \quad \mathsf{P}^B \neq \mathsf{NP}^B.$$

7-AB **Remarque** Lorsqu'on n'est pas habitué aux oracles, il semblerait que si $\mathsf{P} = \mathsf{NP}$ alors pour tout oracle B, $\mathsf{P}^B = \mathsf{NP}^B$, et on pourrait donc conclure du théorème que $\mathsf{P} \neq \mathsf{NP}$… Cette implication n'est bien sûr pas exacte car les modes d'utilisation de l'oracle par une machine déterministe et par une machine non déterministe sont différents : dans le second cas, la machine peut interroger son oracle indépendamment sur chaque chemin d'exécution et effectuer ainsi « virtuellement » un nombre exponentiel d'appels à l'oracle.

On peut mieux comprendre ce phénomène grâce à une image, la *métaphore des bûcherons.* Deux bûcherons travaillent dans une forêt, l'un s'appelle P et l'autre NP. Ils sont tous les deux aussi forts (c'est-à-dire que $\mathsf{P} = \mathsf{NP}$) et parviennent à abattre chacun dix arbres par jour avec leur hache. Certes le bûcheron NP sait lire alors que ce n'est pas le cas de P, mais cela ne l'aide pas à couper des arbres.

Un jour, on leur offre chacun la même tronçonneuse sophistiquée (l'oracle B), objet qu'ils ne savent bien sûr pas encore utiliser. Cette machine vient avec un livret d'instructions permettant d'apprendre à s'en servir. Le bûcheron NP, qui sait lire, apprend le soir à utiliser la tronçonneuse tandis que son collègue P ne peut rien en faire. Le lendemain, NP peut ainsi couper cent arbres grâce à sa tronçonneuse alors que P n'en coupe que dix avec sa hache. On a donc $\mathsf{NP}^B \neq \mathsf{P}^B$, alors que $\mathsf{P} = \mathsf{NP}$. Une capacité « orthogonale » à la force brute, la lecture, permet d'utiliser la tronçonneuse dans un cas mais pas dans l'autre : en d'autres termes, la machine NP parvient à utiliser son oracle pour augmenter sa puissance, mais pas la machine P.

FIGURE 7.4 – Bûcheron par Ferdinand Hodler.

Idée de la démonstration du théorème 7-AA Un langage assez puissant convient pour A : intuitivement il « écrase » les différences entre P et NP (ce serait dans la remarque

précédente une « tronçonneuse » puissante mais simple d'utilisation, que l'on pourrait utiliser sans livret d'instructions). Il est aisé de voir qu'un problème PSPACE-complet convient.

La construction de B est plus difficile et correspond à la « tronçonneuse » de la remarque précédente. Pour tout oracle B, on peut décider dans NP sur l'entrée 1^n si B contient un mot de taille n : il suffit de deviner ce mot et de tester son appartenance par un appel à l'oracle. Nous allons construire un oracle B tel qu'une machine déterministe ne peut pas répondre à cette question en temps polynomial.

Pour cela, on définit B de sorte qu'une machine déterministe polynomiale M ne demande à son oracle que des mots pour lesquels la réponse est négative, c'est-à-dire que B n'aide pas la machine M. Cela nous permettra de faire en sorte que M se trompe sur au moins une entrée (diagonalisation).

Démonstration du théorème 7-AA Les lemmes 7-O et 7-P montrent que le problème PSPACE-complet QBF convient comme oracle A puisque $\mathsf{P}^{\mathtt{QBF}} = \mathsf{NP}^{\mathtt{QBF}} = \mathsf{PSPACE}$.

La construction de B est un peu plus subtile. Nous allons diagonaliser sur les machines déterministes polynomiales tout en garantissant que notre langage reste dans NP. Quel que soit le langage D, on définit pour cela $L_D = \{1^n \mid D^{=n} \neq \emptyset\}$ l'ensemble des longueurs n en unaire telles que D possède au moins un mot de taille n. Pour tout langage D, $L_D \in \mathsf{NP}^D$ puisqu'il suffit de deviner un mot x de taille n et de vérifier qu'il est dans D, comme à l'exemple 7-H.

Il s'agit donc maintenant de construire un langage B tel que $L_B \notin \mathsf{P}^B$. On va faire en sorte que toutes les questions posées par une machine déterministe polynomiale aient une réponse négative. Soit (M_i) une énumération des machines de Turing à oracle fonctionnant en temps polynomial. On définit simultanément une suite d'ensembles finis B_i et une suite d'entiers u_i en commençant par $u_0 = 0$ et $B_0 = \emptyset$, puis pour $i \geqslant 1$:

- soit v la taille maximale des mots demandés à son oracle par $M_{i-1}^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_{i-1}})$: si $t(n)$ est le temps de calcul (polynomial) de $M_i^{B_1 \cup \cdots \cup B_{i-1}}$ sur une entrée de taille n, on définit alors u_i le premier entier $> \max(v, u_{i-1})$ tel que $t(u_i) < 2^{u_i}$ (un tel entier existe puisque $t(n)$ est polynomial) ;
- si $M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i}) = 1$ alors $B_i = \emptyset$ (et donc $M_i^{B_1 \cup \cdots \cup B_i}(1^{u_i}) = M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i})$) ;
- si $M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i}) = 0$ alors

$$B_i = \{a \mid |a| = u_i \text{ et } M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i}) \text{ ne demande pas } a \text{ à son oracle}\}.$$

  Remarquons dans ce cas que $B_i \neq \emptyset$ par le choix de u_i : en effet, puisque $t(u_i) < 2^{u_i}$, M_i ne peut pas demander à son oracle les 2^{u_i} mots de taille u_i. Remarquons aussi que $M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i}) = M_i^{B_1 \cup \cdots \cup B_i}(1^{u_i}) = 0$ puisque l'on ajoute à l'oracle seulement des mots qui ne sont pas demandés par $M_i(1^{u_i})$. Ce cas est illustré à la figure 7.5.

Intuitivement, u_{i+1} est défini suffisamment grand à partir de u_i de sorte que les calculs $M_j(1^{u_j})$ soient « indépendants » les uns des autres. On pose alors $B = \cup_{i \geqslant 0} B_i$.

Mots de taille n	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$	x_{2^n}
Questions à l'oracle	✓	✓	×	✓	×	×	$\cdots$	✓
Mot dans B ?	0	0	1	0	1	1	$\cdots$	0

FIGURE 7.5 – Construction de l'oracle B tel que $\mathsf{P}^B \neq \mathsf{NP}^B$: cas où $M_i(1^{u_i}) = 0$.

Considérons le fonctionnement de la machine M_i avec oracle B sur l'entrée 1^{u_i}. Par définition de B_i et u_{i+1}, $M_i^B(1^{u_i}) = M_i^{B_1 \cup \cdots \cup B_{i-1}}(1^{u_i})$ car M_i ne fait aucun appel à son oracle sur un mot de $B \setminus (B_1 \cup \cdots \cup B_{i-1})$. En effet, les mots de B_j pour $j > i$ sont trop grands pour être accessibles (de taille $\geqslant u_{i+1}$), et les mots de B_i ne sont pas demandés par définition de B_i.
Mais alors par définition de B_i, $M_i^B(1^{u_i}) = 1$ si $B^{=u_i} = \emptyset$ et $M_i^B(1^{u_i}) = 0$ si $B^{=u_i} \neq \emptyset$, donc M_i^B ne reconnaît pas L_B. Ainsi, aucune machine de l'énumération avec oracle B ne reconnaît L_B, donc $L_B \notin \mathsf{P}^B$. □

Ce résultat nous dit que pour résoudre la question « $\mathsf{P} = \mathsf{NP}$? », les techniques simples de diagonalisation que l'on a utilisées jusqu'à présent, dans lesquelles les machines étaient simulées comme des boîtes noires, ne suffiront pas. Il montre ainsi les limites de l'une des rares méthodes connues pour séparer des classes.

7-AC **Remarque** La construction de l'oracle B met en fait le langage L_B non seulement dans NP^B mais aussi dans RP^B car la plupart des mots a dans la définition de B_i ne sont pas demandés par la machine puisqu'elle fonctionne en temps polynomial : pour décider si $1^n \in L_B$ il suffit de choisir au hasard un mot x de taille n et de demander à l'oracle si $x \in B$. Si $1^n \notin L_B$ alors aucun x ne conviendra, sinon la plupart des x conviendront.
Ainsi, la même preuve montre que $\mathsf{RP}^B \neq \mathsf{P}^B$.

7.5.2 Relativisation de la question « EXP = NP ? »

Nous proposons un second résultat de non-relativisation d'une autre question ouverte importante, « $\mathsf{EXP} = \mathsf{NP}$? ». La difficulté ici vient du fait que la machine EXP peut demander à son oracle des questions de taille exponentielle, lui conférant ainsi une grande puissance par rapport à la machine NP. Ce résultat a été montré par Heller [Hel84].

7-AD **Théorème (Heller, 1984)**

Il existe deux oracles A et B tels que

$$\mathsf{EXP}^A = \mathsf{NP}^A \quad \text{et} \quad \mathsf{EXP}^B \neq \mathsf{NP}^B.$$

On en conclut à nouveau qu'on ne peut se contenter de techniques de diagonalisation simples comme celles vues jusqu'à présent pour résoudre la question « $\mathsf{EXP} = \mathsf{NP}$? ». Cette fois la partie difficile est la construction de A. La preuve de ce théorème est scindée en deux lemmes ci-dessous, en commençant par la construction plus facile de l'oracle B.

7-AE Lemme

Il existe un oracle B tel que $\mathsf{EXP}^B \neq \mathsf{NP}^B$.

Démonstration Si $\mathsf{NP} \neq \mathsf{EXP}$ alors $B = \emptyset$ convient.

Sinon, $\mathsf{NP} = \mathsf{EXP}$ donc NP est clos par complément ($\mathsf{NP} = \mathsf{coNP}$) et $\mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}$ par le lemme 7-Q.

Mais par ailleurs, $\mathsf{EXP}^{\mathsf{NP}} \supseteq \mathsf{NEXP}$ par le lemme 7-N donc par le théorème 2-AI de hiérarchie non déterministe en temps, $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{NP}$.

Ainsi, $B = \mathtt{SAT}$ convient d'après le lemme 7-O puisque $\mathsf{EXP}^{\mathtt{SAT}} = \mathsf{EXP}^{\mathsf{NP}} \supseteq \mathsf{NEXP} \supsetneq \mathsf{NP} = \mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}^{\mathtt{SAT}}$. □

Étrangement, la construction de l'oracle A qui rend EXP égal à NP ressemble en partie à celle de l'oracle séparant P et NP du théorème 7-AA.

7-AF Lemme

Il existe un oracle A tel que $\mathsf{EXP}^A = \mathsf{NP}^A = \mathsf{EXP}$.

Idée de la démonstration Le langage A va contenir l'ensemble des requêtes à son oracle effectuées par une machine universelle U simulant les machines déterministes exponentielles. Ainsi, cet oracle n'aide pas U puisque ses réponses seront toujours *oui*, donc $\mathsf{EXP}^A = \mathsf{EXP}$.

Par ailleurs, puisque U pose au plus un nombre exponentiel de questions, certains mots de taille polynomiale ne sont pas demandés à l'oracle et n'influencent donc pas le calcul de U^A. On encode alors dans ces mots un problème EXP-complet : en devinant l'un de ces mots, dans NP^A on pourra donc résoudre tout problème EXP.

Démonstration Pour un oracle B quelconque, soit C^B le problème EXP^B-complet canonique

$$C^B = \{(\langle M\rangle, x, 1^t) \mid M^B(x) = 1 \text{ en temps } \leqslant 2^t\}.$$

Ce problème est EXP^B-complet comme on peut le voir en adaptant la preuve de la proposition 3-M. On peut le résoudre en utilisant une machine universelle déterministe U telle que pour tout oracle B, $U^B(\langle M\rangle, x)$ simule $M^B(x)$ en temps $O(t^2)$ si $M^B(x)$ fonctionne en temps t (la construction de la proposition 1-Q convient) : le temps de simulation est alors $O(2^{2n})$ si n est la taille de l'entrée $(\langle M\rangle, x, 1^t)$. On notera M_C une machine telle que pour tout oracle B, M_C^B décide C^B en temps $O(2^{2n})$.

Nous pouvons maintenant décrire l'oracle A du résultat : il sera l'union d'ensembles finis A_i définis inductivement comme suit. Par commodité, on fixe $A_{-1} = \emptyset$ et pour

chaque i on notera $A'_i = \cup_{j=0}^{i} A_j \cup \Sigma^{\geqslant (i+1)^2}$ (où Σ désigne l'alphabet quelconque sur lequel on définit A, $|\Sigma| \geqslant 2$). On définit alors pour $i \geqslant 0$,

$$A_i = \{(a,b) \mid \begin{array}{l} |(a,b)| \geqslant i^2 \text{ et} \\ (M_C^{\emptyset}(a) = 0 \text{ ou} \\ \exists x \text{ de taille } i \text{ tel que } M_C^{A'_{i-1}}(x) \text{ demande } (a,b) \text{ à son oracle})\}. \end{array}$$

Intuitivement, la partie « $M_C^{\emptyset}(a) = 0$ » permettra à NP de décider le problème EXP-complet canonique $C^{\emptyset}$, tandis que le reste de la définition sert à s'assurer que l'oracle n'aidera pas la machine M_C car toutes ses réponses seront positives. Le mot b du couple (a,b) sert ici simplement de rembourrage pour s'assurer un nombre suffisant de mots dans la preuve de $\mathsf{EXP} = \mathsf{NP}^A$.

On définit alors $A = \cup_{i \geqslant 0} A_i$. Montrons que pour tout mot x de taille n, $M_C^A(x) = M_C^{A'_{n-1}}(x)$.

Démonstration — Considérons $r_1, \ldots, r_N$ les requêtes de $M_C^A(x)$ à son oracle et montrons par récurrence sur i que la réponse à r_i est la même que dans le calcul $M_C^{A'_{n-1}}(x)$. Pour $i = 1$, la question r_1 est la même dans $M_C^A(x)$ et dans $M_C^{A'_{n-1}}(x)$ puisqu'il s'agit de la première. Si $|r_1| < n^2$ alors $r_1 \in A \iff r_1 \in A'_{n-1}$ donc la réponse est la même. Si $|r_1| \geqslant n^2$, alors par définition de A_n le mot r_1 est dans A, et il est également dans A'_{n-1} par définition, donc la réponse est encore identique dans les deux cas.

Pour $i > 1$: toutes les questions et les réponses jusqu'au rang $i-1$ ont été les mêmes pour $M_C^A(x)$ et $M_C^{A'_{n-1}}(x)$ par hypothèse de récurrence. Ainsi, les deux calculs sont pour l'instant identiques et en particulier la i-ème question r_i est la même dans les deux cas. Si $|r_i| < n^2$ alors comme précédemment $r_i \in A \iff r_i \in A'_{n-1}$ donc la réponse est identique dans les deux cas. Si $|r_i| \geqslant n^2$ alors par définition de A_n le mot r_i est dans A (puisque les i premières questions sont identiques), et il est également dans A'_{n-1} par définition, donc la réponse est encore identique dans les deux cas.

Ainsi, toutes les questions et réponses sont les mêmes dans les deux cas, donc les calculs sont identiques. ◇

Pour la suite, nous aurons besoin d'énumérer des éléments de A : pour ce faire, nous montrons que $A \in \mathsf{EXP}$.

Démonstration — Soit (a,b) de taille n. Puisque A_i ne contient que des mots de taille $\geqslant i^2$, il s'agit de tester si $(a,b) \in \cup_{i=0}^{m} A_i$ où $m = \lfloor \sqrt{n} \rfloor$.

On teste d'abord si $M_C^{\emptyset}(a) = 0$, ce qui prend un temps $2^{O(|a|)}$, et on accepte si c'est le cas.

Sinon, nous construisons l'ensemble des éléments de $A_0, A_1, \ldots, A_m$. Pour construire A_i à partir de $A_0, \ldots, A_{i-1}$, il suffit d'exécuter $M_C^{A'_{i-1}}$ sur tous les mots de taille i, ce que l'on effectue en simulant les appels à l'oracle par un parcours des éléments de $\cup_{j=0}^{i-1} A_j$, ou en répondant 1 si le mot demandé

est de taille $\geqslant i^2$. Puisque M_C fonctionne en temps $2^{O(n)}$, lister les éléments de A_i à partir de ceux de $A_0,\ldots,A_{i-1}$ prend donc un temps $2^{O(i)}$. Au total, construire l'ensemble des éléments de $A_0,A_1,\ldots,A_m$ prend donc un temps $2^{O(m)}$.

Pour décider si $(a,b)\in A$, il suffit alors de tester si (a,b) fait partie de la liste d'éléments énumérés, ce qui se fait en temps $2^{O(n)}$. ◇

Nous pouvons maintenant montrer que $\mathsf{EXP}^A = \mathsf{EXP}$.

Démonstration — Soit $B\in\mathsf{EXP}^A$: par complétude de C^A, il existe une réduction f calculable en temps polynomial telle que $x\in B \iff f(x)\in C^A \iff M_C^A(f(x))=1$.

Fixons x et soit m la taille de $f(x)$. Par ce qui précède, on a $M_C^A(f(x)) = M_C^{A'_{m-1}}(f(x))$ donc $x\in B$ ssi $M_C^{A'_{m-1}}(f(x))=1$. Simuler ce calcul se fait en temps exponentiel sans oracle puisqu'il suffit d'énumérer les éléments de $\cup_{j=0}^{m-1}A_j$ (ce qui se fait en temps exponentiel en m comme on l'a vu ci-dessus) et de remplacer les réponses de l'oracle par 1 si $(a,b)\in\cup_{j=0}^{m-1}A_j$ ou si $|(a,b)|\geqslant m^2$, et par 0 sinon. ◇

Montrons maintenant que $\mathsf{NP}^A = \mathsf{EXP}$.

Démonstration — Soit $B\in\mathsf{EXP}$: il existe une réduction polynomiale f telle que $x\in B$ ssi $M_C^{\emptyset}(f(x))=1$. On note m la taille de $f(x)$. Voici une machine non déterministe avec oracle A pour résoudre B sur l'entrée x :

- calculer $f(x)$ (de taille m) ;
- deviner b de taille m^2 ;
- `accepter` ssi $(f(x),b)\notin A$.

Si $M_C^{\emptyset}(f(x))=0$ alors par définition de A, tous les b satisfont $(f(x),b)\in A$ et donc on rejette. En revanche si $M_C^{\emptyset}(f(x))=1$, il faut d'abord remarquer que M_C ne peut pas poser plus de $O(2^{2m})$ questions à son oracle car elle fonctionne en temps $O(2^{2m})$. Puisqu'il y a $2^{m^2}=\omega(2^m 2^{2m})$ mots de taille m^2, il existe un mot b_0 de taille m^2 tel que $(f(x),b_0)$ n'est demandé à l'oracle au cours du calcul de M_C^A sur aucune entrée de taille $\leqslant m$. Par définition de A, ce mot satisfait donc $(f(x),b_0)\notin A$ et notre machine accepte x le long du chemin devinant b_0.

Cette machine non déterministe décide B en temps polynomial grâce à l'oracle A, donc $B\in\mathsf{NP}^A$. ◇

Nous avons donc montré que $\mathsf{EXP}^A = \mathsf{NP}^A = \mathsf{EXP}$. □

7-AG **Remarque** Dans l'algorithme NP^A de la preuve ci-dessus, en réalité la majorité des mots b de taille m^2 fournissent la bonne réponse. La même preuve montre donc que $\mathsf{RP}^A = \mathsf{EXP}^A$.

7.6 De la difficulté de définir la bonne notion de diagonalisation

Les résultats précédents montrent qu'une méthode de preuve qui « passe à l'oracle » ne permet pas de résoudre les grandes questions ouvertes de complexité. Nous avons interprété ces résultats comme l'impossibilité d'utiliser une diagonalisation simple pour ces questions, car en général les méthodes de diagonalisation simulent les machines comme des boîtes noires et restent donc valides avec un oracle. Néanmoins, nous n'avons pas défini précisément ce qu'est une diagonalisation car il n'y a pas de définition formelle satisfaisante. À titre d'illustration, nous allons voir que pour une certaine définition naturelle mais large de la diagonalisation, « toute » séparation est diagonalisation… Nous donnons ce résultat pour contrebalancer l'interprétation que nous avons faite des théorèmes de non-relativisation, mais il ne remet pas en cause leur pertinence : il montre plutôt que la tentative ci-dessous de définition de la notion de diagonalisation est trop générale. Et il montre qu'il faut être prudent sur le terme « diagonalisation ».

Quels sont les ingrédients d'une preuve par diagonalisation ? Pour montrer qu'un langage L n'est pas dans une classe $\mathscr{C}$, on énumère les machines (M_i) reconnaissant les langages de $\mathscr{C}$ et on montre que chacune se trompe sur au moins un mot de L (c'est-à-dire que pour tout i il existe x tel que $M_i(x) = 0$ ssi $x \in L$).

On peut donc considérer qu'une preuve par diagonalisation de $L \notin \mathscr{C}$ est une manière d'énumérer les machines (M_i) de sorte qu'on parvienne à montrer que chaque machine se trompe sur au moins un mot x. Il s'agit donc de trouver un ordre d'énumération des machines nous permettant de montrer cette propriété. En d'autres termes, on cherche une fonction $h : \Sigma^* \to \mathbb{N}$ surjective (pour que chaque machine soit énumérée) telle que le langage L soit égal à

$$\mathrm{Diag}_h = \{x \mid M_{h(x)}(x) = 0\}.$$

En effet, si h est surjective alors chaque machine sera énumérée et chacune fera une erreur, ce qui montre que $\mathrm{Diag}_h \notin \mathscr{C}$ (argument diagonal).

Le résultat suivant, dû à Kozen [Koz80], montre que l'on peut toujours exprimer L de cette façon. Nous montrerons ce résultat dans le cas particulier où $\mathscr{C} = \mathsf{P}$ mais il se généralise aisément à toute classe $\mathscr{C}$ qui admet une énumération de machines (ce qui est le cas de P comme on l'a vu à la remarque 3-AL et à la section 7.1.1).

On note donc $(M_i)_{i\in\mathbb{N}}$ une énumération des machines déterministes fonctionnant en temps polynomial et, pour toute fonction $h : \Sigma^* \to \mathbb{N}$, $\mathrm{Diag}_h \subseteq \Sigma^*$ le langage $\mathrm{Diag}_h = \{x \mid M_{h(x)}(x) = 0\}$.

7-AH Proposition (Kozen, 1980)

Soit $L \subseteq \Sigma^*$ un langage hors de P. Alors il existe une fonction surjective $h : \Sigma^* \to \mathbb{N}$ telle que $L = \mathrm{Diag}_h$.

Idée de la démonstration Puisqu'aucune machine M_i ne reconnaît L, chacune doit se

tromper sur au moins un mot x (c'est-à-dire $x \in L$ ssi $M_i(x) = 0$). On peut donc associer un tel mot x_i à tout i, en prenant garde qu'ils soient tous différents. La fonction inverse h qui à x_i associe i vérifie donc $x_i \in L$ ssi $M_{h(x_i)}(x_i) = 0$, ou en d'autres termes $x_i \in L$ ssi $x_i \in \mathrm{Diag}_h$. Il est alors facile d'étendre h à tout $\Sigma^\star$ pour conclure.

Démonstration Nous allons définir h comme fonction inverse d'une fonction $f : \mathbb{N} \to \Sigma^\star$ strictement croissante pour l'ordre lexicographique (et donc injective) ; et si $x \notin f(\mathbb{N})$ alors $h(x)$ sera égal à une valeur convenue d'avance. Le fait que h soit l'inverse d'une fonction injective f assure que h est surjective. Donnons donc dans un premier temps la définition de $f : \mathbb{N} \to \Sigma^\star$.

Puisque $L \notin \mathsf{P}$ et que M_0 définit un langage de P, cette machine doit se tromper sur un mot de L, c'est-à-dire qu'il existe $x_0 \in \Sigma^\star$ tel que $M_0(x_0) = 0$ ssi $x_0 \in L$. On définit alors $f(0) = x_0$.

Pour $i > 0$, définissons maintenant $f(i) \in \Sigma^\star$ en supposant que $f(i-1)$ est défini. La machine M_i fonctionnant en temps polynomial et $L \notin \mathsf{P}$, elle doit se tromper sur *une infinité* de mots x (c'est-à-dire $x \in L$ ssi $M_i(x) = 0$), sinon L serait dans P par clôture de P par changements finis (lemme 2-T). Donc il existe un mot $x_i > f(i-1)$ (pour l'ordre lexicographique) tel que $x_i \in L$ ssi $M_i(x_i) = 0$. On définit $f(i) = x_i$.

La définition de $f : \mathbb{N} \to \Sigma^\star$ garantit que :

- f est strictement croissante, et
- pour tout $i \in \mathbb{N}$, $f(i) \in L$ ssi $M_i(f(i)) = 0$ (en d'autres termes, la machine M_i se trompe pour le mot $f(i)$).

Nous pouvons maintenant passer à la définition de la fonction $h : \Sigma^\star \to \mathbb{N}$. On notera i_0 l'indice d'une machine M_{i_0} qui rejette tous les mots ($\forall x \in \Sigma^\star$, $M_{i_0}(x) = 0$), et i_1 l'indice d'une machine M_{i_1} qui accepte tous les mots ($\forall x \in \Sigma^\star$, $M_{i_1}(x) = 1$). Soit $x \in \Sigma^\star$:

- si x a un antécédent i par f (c'est-à-dire que $f(i) = x$), alors on définit $h(x) = i$;
- sinon, on définit $h(x) = \begin{cases} i_0 \text{ si } x \in L \\ i_1 \text{ si } x \notin L \end{cases}$.

L'injectivité de f implique que h est bien définie et surjective. Par ailleurs, deux cas se présentent pour un mot $x \in \Sigma^\star$:

- si $x \in f(\mathbb{N})$, disons $f(i) = x$, alors $x \in L$ ssi $M_i(x) = 0$; mais puisque $h(x) = i$, on a donc $x \in L$ ssi $M_{h(x)}(x) = 0$ ssi $x \in \mathrm{Diag}_h$;
- sinon, si $x \in L$ alors $h(x) = i_0$, et si $x \notin L$ alors $h(x) = i_1$: puisque $M_{i_0}(x) = 0$ et $M_{i_1}(x) = 1$, dans ces deux cas $x \in L$ ssi $M_{h(x)}(x) = 0$ ssi $x \in \mathrm{Diag}_h$.

On en déduit que $L = \mathrm{Diag}_h$. □

7-AI **Remarque** Il est facile de voir dans la preuve ci-dessus que si L est décidable, alors h est calculable.

Nous pouvons maintenant appliquer ce qui précède à la question « $\mathsf{P} = \mathsf{NP}\,?$ » pour montrer que « si $\mathsf{P} \neq \mathsf{NP}$ alors il existe une preuve de type diagonalisation que $\mathtt{SAT}$ n'est pas dans P ».

7-AJ **Corollaire**

Si $\mathsf{NP} \neq \mathsf{P}$ alors il existe une fonction $h : \{0,1\}^\star \to \mathbb{N}$ surjective telle que $\mathrm{Diag}_h = \mathtt{SAT}$. On peut considérer qu'il s'agit d'une sorte de « preuve par diagonalisation » du fait que $\mathtt{SAT} \notin \mathsf{P}$ puisque Diag_h est clairement hors de P par un argument diagonal.

Gardons tout de même en tête que, malgré ce résultat, les grandes questions ouvertes en complexité ne peuvent être résolues par les méthodes connues de « diagonalisation », dans lesquelles les machines sont simulées comme des boîtes noires. En d'autres termes, de nouvelles techniques qui ne relativisent pas doivent être mises au point pour avoir un espoir de résoudre ces questions.

Mentionnons enfin que les exercices B-B et B-E en annexe permettent de s'entraîner sur les oracles.

8 La hiérarchie polynomiale

Nous avons déjà discuté du fait que la classe NP n'est probablement pas close par complémentaire. Nous avons donc été amenés à considérer la classe coNP et également $\mathsf{NP}^{\mathsf{NP}}$ au lemme 7-Q.

Nous allons voir comment cette remarque permet de construire une famille de classes de complexité, appelée *hiérarchie polynomiale*, se situant entre P et PSPACE. La définition sera donnée en termes d'oracles, mais à l'instar de NP, nous montrerons une caractérisation à l'aide de quantificateurs.

Nous nous attacherons ensuite à comparer la hiérarchie polynomiale aux classes vues jusqu'à présent. Puis le théorème 8-P de Karp et Lipton permettra de connecter la question des bornes inférieures non uniformes pour NP à celle de savoir si la hiérarchie polynomiale est stricte. Enfin, le genre de techniques développées dans ce chapitre nous permettra de montrer une borne inférieure sur le temps et l'espace simultanés nécessaires à la résolution de problèmes de type NP.

8.1 La hiérarchie polynomiale

Pourquoi NP ne semble pas être close par complément ? Que se passerait-il si elle l'était ? Comment une machine non déterministe peut-elle mettre à profit son oracle ? C'est notamment pour étudier ces questions que la hiérarchie polynomiale a été introduite par Meyer et Stockmeyer [MS72]. Les premières propriétés ci-dessous et la complétude des problèmes $\mathtt{SAT}_i$ ont également été montrées dans cet article.

8.1.1 Définition et premières propriétés

La hiérarchie polynomiale est définie en termes d'oracles.

8-A Définition (hiérarchie polynomiale)

La famille des classes de complexité Σ_i^{P} (pour $i \geqslant 0$) est définie inductivement comme suit :

- $\Sigma_0^{\mathsf{P}} = \mathsf{P}$;
- $\Sigma_{i+1}^{\mathsf{P}} = \mathsf{NP}^{\Sigma_i^{\mathsf{P}}}$.

La hiérarchie polynomiale est alors $\mathsf{PH} = \bigcup_{i \geqslant 0} \Sigma_i^{\mathsf{P}}$.

On notera également $\Pi_i^{\mathsf{P}} = \mathrm{co}\Sigma_i^{\mathsf{P}}$ l'ensemble des langages dont le complémentaire est dans Σ_i^{P}, et $\Delta_i^{\mathsf{P}} = \mathsf{P}^{\Sigma_{i-1}^{\mathsf{P}}}$ pour $i \geqslant 1$.

Ainsi, $\Sigma_1^{\mathsf{P}} = \mathsf{NP}$, $\Delta_2^{\mathsf{P}} = \mathsf{P}^{\mathsf{NP}}$, $\Sigma_2^{\mathsf{P}} = \mathsf{NP}^{\mathsf{NP}}$, etc. (cf. figure 8.1). La classe Σ_i^{P} est appelé le i-ème niveau de la hiérarchie polynomiale.

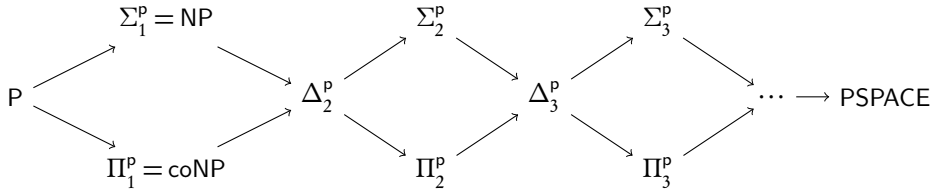


FIGURE 8.1 – Différents niveaux de la hiérarchie polynomiale. Une flèche $\mathscr{C} \to \mathscr{D}$ exprime l'inclusion $\mathscr{C} \subseteq \mathscr{D}$.

La hiérarchie polynomiale est incluse dans PSPACE comme le montre le résultat suivant.

8-B Lemme

$\mathsf{PH} \subseteq \mathsf{PSPACE}$.

Démonstration Il suffit de montrer que pour tout i, $\Sigma_i^{\mathsf{P}} \subseteq \mathsf{PSPACE}$. Nous procédons par récurrence.

Pour $i = 0$, le résultat est clair puisque $\Sigma_0^{\mathsf{P}} = \mathsf{P}$.

Pour $i > 0$, $\Sigma_i^{\mathsf{P}} = \mathsf{NP}^{\Sigma_{i-1}^{\mathsf{P}}}$. Par hypothèse de récurrence, on a donc $\Sigma_i^{\mathsf{P}} \subseteq \mathsf{NP}^{\mathsf{PSPACE}}$. Le lemme 7-P permet de conclure puisque $\mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE}$. □

Il est communément supposé que $\mathsf{P} \neq \mathsf{NP}$, c'est-à-dire que le non-déterminisme apporte de la puissance. Cette supposition peut être généralisée à la hiérarchie polynomiale : l'opinion dominante est que celle-ci est probablement stricte, c'est-à-dire que pour tout i, $\Sigma_{i+1}^{\mathsf{P}} \neq \Sigma_i^{\mathsf{P}}$. Le résultat suivant montre que l'égalité de deux niveaux successifs provoque un *effondrement* de toute la hiérarchie. Karp [Kar72] avait déjà fait cette remarque avant même la définition de la hiérarchie polynomiale par [MS72].

8-C Proposition

Si $\mathsf{P} = \mathsf{NP}$ alors $\mathsf{PH} = \mathsf{P}$.

Plus généralement, pour $i \geqslant 0$, si $\Sigma^{\mathsf{p}}_{i+1} = \Sigma^{\mathsf{p}}_i$ alors $\mathsf{PH} = \Sigma^{\mathsf{p}}_i$.

Démonstration Supposons $\Sigma^{\mathsf{p}}_{i+1} = \Sigma^{\mathsf{p}}_i$ et montrons par récurrence sur $j \geqslant i$ que $\Sigma^{\mathsf{p}}_j = \Sigma^{\mathsf{p}}_i$.

Pour $j = i$, le résultat est trivial.

Pour $j > i$: $\Sigma^{\mathsf{p}}_j = \mathsf{NP}^{\Sigma^{\mathsf{p}}_{j-1}} = \mathsf{NP}^{\Sigma^{\mathsf{p}}_i}$ par hypothèse de récurrence. Ainsi, $\Sigma^{\mathsf{p}}_j = \Sigma^{\mathsf{p}}_{i+1}$. Mais par hypothèse, $\Sigma^{\mathsf{p}}_{i+1} = \Sigma^{\mathsf{p}}_i$, donc $\Sigma^{\mathsf{p}}_j = \Sigma^{\mathsf{p}}_i$. □

La réciproque de ce résultat est évidente, il s'agit donc d'une équivalence.

Le lemme 7-Q nous donne également un autre résultat d'effondrement.

8-D Corollaire

Si $\mathsf{NP} = \mathsf{coNP}$ alors $\mathsf{PH} = \mathsf{NP}$.

Démonstration Par le lemme 7-Q, si $\mathsf{NP} = \mathsf{coNP}$ alors $\Sigma^{\mathsf{p}}_2 = \Sigma^{\mathsf{p}}_1$. La proposition 8-C implique que $\mathsf{PH} = \Sigma^{\mathsf{p}}_1 = \mathsf{NP}$. □

On généralisera ce résultat (cf. corollaire 8-H) lorsqu'on aura vu la caractérisation de PH en termes de quantificateurs.

Enfin, nous noterons que chaque niveau de la hiérarchie polynomiale est clos par réduction many-one.

8-E Lemme

Soit A et B deux langages. Si $A \in \Sigma^{\mathsf{p}}_i$ et $B \leqslant^{\mathsf{p}}_{\mathsf{m}} A$, alors $B \in \Sigma^{\mathsf{p}}_i$.

Démonstration Par définition de Σ^{p}_i, A est reconnue par une machine non déterministe polynomiale N avec un oracle $S \in \Sigma^{\mathsf{p}}_{i-1}$.

Soit f la réduction polynomiale de B à A, c'est-à-dire $x \in B \iff f(x) \in A$. Voici une machine non déterministe polynomiale N_B avec oracle S pour décider B sur l'entrée x :

- calculer $f(x)$;
- exécuter le calcul non déterministe $N^S(f(x))$.

Cette machine non déterministe N_B avec oracle S décide B en temps polynomial. Puisque $S \in \Sigma^{\mathsf{p}}_{i-1}$, on en déduit que $B \in \mathsf{NP}^{\Sigma^{\mathsf{p}}_{i-1}} = \Sigma^{\mathsf{p}}_i$. □

8.1.2 Caractérisation en termes de quantificateurs

La classe NP admet la caractérisation existentielle que nous connaissons (proposition 2-AO), coNP une caractérisation universelle. Que se passe-t-il si l'on associe les deux, si l'on alterne les quantificateurs ? Le résultat de PSPACE-complétude du langage QBF (théorème 4-AC) montre qu'un nombre arbitraire d'alternances de quantificateurs défini la classe PSPACE. Nous allons voir comment caractériser la hiérarchie polynomiale de cette manière. Dans les démonstrations, si $Q \in \{\exists, \forall\}$ est un quantificateur, on notera $\bar{Q}$ le quantificateur opposé (c'est-à-dire $\bar{Q} = \forall$ si $Q = \exists$, tandis que $\bar{Q} = \exists$ si $Q = \forall$).

8-F Proposition

Soit $i \geqslant 0$. Un langage A est dans Σ_i^p si et seulement s'il existe un langage $B \in P$ et un polynôme $p(n)$ tels que :

$$x \in A \iff$$
$$\exists y_1 \in \{0,1\}^{p(|x|)}\, \forall y_2 \in \{0,1\}^{p(|x|)}\, \exists y_3 \in \{0,1\}^{p(|x|)} \ldots Q y_i \in \{0,1\}^{p(|x|)}\, (x, y_1, \ldots, y_i) \in B,$$

où Q est le quantificateur $\exists$ si i est impair et $\forall$ sinon.

En d'autres termes, un problème est dans Σ_i^p si et seulement s'il s'exprime par une alternance de i quantificateurs commençant par $\exists$.

8-G Remarque Puisque $\Pi_i^p = \text{co}\Sigma_i^p$, la classe Π_i^p admet une caractérisation similaire mais en commençant par $\forall$ plutôt que par $\exists$.

Idée de la démonstration de la proposition 8-F Dans la définition de $\Sigma_i^p = \text{NP}^{\text{NP}^{\cdot^{\cdot^{\text{NP}}}}}$, chaque oracle NP ajoute des quantificateurs : une réponse positive de l'oracle NP donne un certificat et donc un quantificateur $\exists$, tandis qu'une réponse négative donne un quantificateur $\forall$. On pourrait donc croire qu'il faut une alternance de $2i$ quantificateurs (et non seulement i).

Mais par induction, si le dernier quantificateur ajouté était $\exists$, alors il prend déjà en compte le cas des réponses positives de l'oracle, donc on ajoute seulement une alternance avec le quantificateur $\forall$ pour les réponses négatives. De même si le dernier quantificateur ajouté était $\forall$, alors il prend déjà en compte le cas des réponses négatives de l'oracle, donc on ajoute seulement une alternance avec le quantificateur $\exists$ pour les réponses positives. Ainsi, chaque oracle ajoute exactement une alternance de quantificateurs.

Démonstration de la proposition 8-F Nous commençons par montrer le sens $\Leftarrow$. Supposons donc que l'appartenance à A s'exprime par une alternance de i quantificateurs comme dans l'énoncé. Nous montrons par récurrence sur i qu'alors $A \in \Sigma_i^p$.

Le résultat est clair pour $i = 0$.

Pour $i > 0$: si l'on oublie le premier quantificateur $\exists$, on obtient un langage A' dont les entrées sont les couples (x, y_1) vérifiant $\forall y_2 \exists y_3 \ldots Q y_i\ (x, y_1, y_2, \ldots, y_i) \in B$. Il s'agit

par hypothèse de récurrence d'un langage Π^{p}_{i-1}. Mais décider A revient simplement, sur l'entrée x, à décider si $\exists y_1\,(x,y_1)\in A'$: on en déduit que $A\in \mathsf{NP}^{A'}$ puisqu'il suffit de deviner y_1 et de tester grâce à l'oracle si $(x,y_1)\in A'$.
Or adjoindre A' comme oracle ou son complémentaire ${}^cA'$ ne change rien puisqu'il suffit d'inverser les réponses de l'oracle : en d'autres termes, $A\in \mathsf{NP}^{{}^cA'}$. Puisque ${}^cA'\in \Sigma^{\mathsf{p}}_{i-1}$, on en déduit que $A\in \mathsf{NP}^{\Sigma^{\mathsf{p}}_{i-1}}=\Sigma^{\mathsf{p}}_i$.

Pour la réciproque $\Rightarrow$, soit $A\in\Sigma^{\mathsf{p}}_i$ et montrons par récurrence qu'il s'exprime comme une alternance de i quantificateurs comme dans l'énoncé. La preuve ressemble à celle du lemme 7-Q.
Le résultat est clair pour $i=0$.
Pour $i>0$: par définition de Σ^{p}_i, A est reconnu par une machine non déterministe N fonctionnant en temps polynomial $p(n)$ avec un oracle $B\in\Sigma^{\mathsf{p}}_{i-1}$. Par hypothèse de récurrence, B s'exprime par une alternance de $i-1$ quantificateurs :

$$v\in B \iff \exists y_1\in\{0,1\}^{q(|v|)}\,\forall y_2\ldots\,\bar{\mathsf{Q}}y_{i-1}\,(v,y_1,\ldots,y_{i-1})\in C,$$

où $C\in\mathsf{P}$ et $q(n)$ est un polynôme.
Un mot x est dans A si et seulement s'il existe un chemin acceptant dans le calcul $N^B(x)$. Pour exprimer cette condition sous la forme voulue, on va deviner le chemin acceptant u, les questions $(a_1,\ldots,a_m)$ à l'oracle et ses réponses $(b_1,\ldots,b_m)$ (où $m\leqslant p(|x|)$, $|a_i|\leqslant p(|x|)$ et $b_i\in\{0,1\}$) :

$$\begin{aligned} x\in A \iff\ & \exists u\in\{0,1\}^{p(|x|)}\,\exists(a_1,\ldots,a_m)\exists(b_1,\ldots,b_m)\\ & \text{le chemin } u \text{ de } N(x) \text{ où les réponses de l'oracle sont remplacées par}\\ & (b_1,\ldots,b_m) \text{ est acceptant et pose les questions } (a_1,\ldots,a_m) \text{ à son oracle,}\\ & \text{et pour tout } 1\leqslant j\leqslant m,\ [b_j=1\iff a_j\in B]. \end{aligned}$$

En connaissant u et $(b_1,\ldots,b_m)$, on peut décider en temps polynomial si le chemin u est acceptant et pose les questions $(a_1,\ldots,a_m)$ à l'oracle, propriété que l'on appellera φ. Il reste à déterminer si les réponses $(b_1,\ldots,b_m)$ sont les bonnes, c'est-à-dire $[b_j=1\iff a_j\in B]$, que l'on peut réécrire en $[(b_j=1\wedge a_j\in B)\vee(b_j=0\wedge a_j\notin B)]$. Par la caractérisation de B, cela s'exprime sous la forme

$$\begin{aligned} (b_j=1\wedge\exists y_1\,\forall y_2\ldots\,\bar{\mathsf{Q}}y_{i-1}\,(a_j,y_1,\ldots,y_{i-1})\in C)&\\ \vee\,(b_j=0\wedge\forall y_1\,\exists y_2\ldots\,\mathsf{Q}y_{i-1}\,(a_j,y_1,\ldots,y_{i-1})\notin C).& \end{aligned}$$

Un nouveau jeu d'écriture nous permet d'obtenir la forme équivalente suivante :

$$\exists z_1\,\forall z_2\ldots\,\mathsf{Q}z_i\ [(b_j=1\wedge(a_j,z_1,\ldots,z_{i-1})\in C)\vee(b_j=0\wedge(a_j,z_2,\ldots,z_i)\notin C)].$$

La conjonction de toutes ces formules pour j allant de 1 à m s'exprime donc sous la forme :

$$\exists z_1^1 z_1^2\ldots z_1^m\ \forall z_2^1\ldots z_2^m\ldots\,\mathsf{Q}z_i^1\ldots z_i^m\ \psi$$

où

$$\psi \equiv \bigwedge_{j=1}^{m} [(b_j = 1 \wedge (a_j, z_1^j, \ldots, z_{i-1}^j) \in C) \vee (b_j = 0 \wedge (a_j, z_2^j, \ldots, z_i^j) \notin C)].$$

Rappelons que les variables z_k^j sont de taille polynomiale puisqu'elles correspondent à des certificats dans la caractérisation de B ; que les variables b_k sont des bits (les réponses de l'oracle) ; et que les variables a_k sont de taille polynomiale puisqu'il s'agit des questions à l'oracle.

On remarquera que la condition ψ est décidable en temps polynomial sur l'entrée $(a_1 \ldots a_m, b_1 \ldots b_m, z_1^1 \ldots z_i^m)$ puisque m est polynomial et $C \in \mathsf{P}$.

Au final, l'appartenance à A s'exprime alors comme suit :

$$\begin{aligned} x \in A \iff & \exists u \, \exists (a_1, \ldots, a_m) \exists (b_1, \ldots, b_m) \\ & \exists z_1^1 z_1^2 \ldots z_1^m \, \forall z_2^1 \ldots z_2^m \ldots \, \mathsf{Q} z_i^1 \ldots z_i^m \, [\varphi \wedge \psi]. \end{aligned}$$

Les quatre premiers quantificateurs $\exists$ peuvent bien sûr être regroupés au sein d'un seul. Puisque $\varphi \wedge \psi$ peut être décidé en temps déterministe polynomial, il s'agit bien de la caractérisation voulue avec une alternance de i quantificateurs. □

Cette caractérisation nous permettra de donner des problèmes complets pour les classes Σ_i^{p}. Mais avant cela, elle nous autorise à généraliser le corollaire 8-D.

8-H Corollaire

Si $\Sigma_i^{\mathsf{p}} = \Pi_i^{\mathsf{p}}$ alors $\mathsf{PH} = \Sigma_i^{\mathsf{p}}$.

Démonstration Par la proposition 8-C, il suffit de montrer que $\Sigma_{i+1}^{\mathsf{p}} = \Sigma_i^{\mathsf{p}}$. Soit $A \in \Sigma_{i+1}^{\mathsf{p}}$: il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que

$$x \in A \iff \exists y_1 \forall y_2 \ldots \mathsf{Q} y_{i+1} (x, y_1, \ldots, y_{i+1}) \in B, \tag{8.1}$$

où les y_j sont des mots de taille $p(|x|)$.

Sur l'entrée (x, y_1), décider si $\forall y_2 \exists y_3 \ldots \mathsf{Q} y_{i+1} (x, y_1, \ldots, y_{i+1}) \in B$ est un problème de Π_i^{p}. Mais par hypothèse, $\Pi_i^{\mathsf{p}} = \Sigma_i^{\mathsf{p}}$: on peut donc le réécrire sous la forme

$$\exists z_1 \forall z_2 \ldots \bar{\mathsf{Q}} z_i (x, y_1, z_1, \ldots, z_i) \in C, \tag{8.2}$$

où $C \in \mathsf{P}$ et les mots z_i sont de taille polynomiale.

En combinant les expressions (8.1) et (8.2), on obtient

$$x \in A \iff \exists y_1 \exists z_1 \forall z_2 \ldots \bar{\mathsf{Q}} z_i (x, y_1, z_1, \ldots, z_i) \in C,$$

qui est l'expression d'un langage dans Σ_i^{p}. □

Afin d'illustrer ces nouvelles classes, nous donnons deux exemples de problèmes naturels contenus dans la hiérarchie polynomiale à différents niveaux, ainsi qu'une variante de l'un d'eux en exercice ; à la section suivante nous verrons en outre pour chaque niveau un problème complet.

8-I **Exemple** 1. `CLIQUE EXACTE` :

- *entrée* : un graphe non orienté G et un entier k ;
- *question* : la clique de taille maximum de G est-elle de taille k ?

Ce problème peut être exprimé sous la forme

$$\exists s_1,\dots,s_k \in G \text{ distincts tels que } \{s_1,\dots,s_k\} \text{ est une clique}$$
$$\wedge\, \forall s'_1,\dots,s'_{k+1} \in G \text{ distincts, } \{s'_1,\dots,s'_{k+1}\} \text{ n'est pas une clique.}$$

Il est donc dans $\mathsf{P}^{\mathsf{NP}} = \Delta_2^{\mathsf{p}}$.

2. `CIRCUIT MIN` :

- *entrée* : un entier k et un circuit booléen $C(x_1,\dots,x_n)$ calculant une fonction $f : \{0,1\}^n \to \{0,1\}$;
- *question* : existe-t-il un circuit C' de taille $\leqslant k$ qui calcule f ?

Ce problème peut être exprimé sous la forme

$$\exists C' \forall x \in \{0,1\}^n \ (|C'| \leqslant k \wedge C(x) = C'(x))$$

et est donc dans Σ_2^{p}.

✎ **8-J** **Exercice**

Montrer que la variante du problème `CIRCUIT MIN` ci-dessus où l'on demande si le circuit minimum pour f est de taille *exactement* k, est dans Δ_3^{p}.

8.1.3 Problèmes complets

Tout d'abord, il faut garder à l'esprit que la hiérarchie polynomiale est une union de classes et de ce fait n'a probablement pas de problème complet, comme le montre le lemme suivant.

8-K **Lemme**

Si la hiérarchie polynomiale possède un problème complet, alors elle s'effondre ($\mathsf{PH} = \Sigma_i^{\mathsf{p}}$ pour un certain i).

Démonstration Supposons que $A \in \mathsf{PH}$ soit PH-complet. Par définition de PH, $A \in \Sigma_i^{\mathsf{p}}$ pour un certain i.
Soit $B \in \mathsf{PH}$: par complétude de A, il se réduit à $A \in \Sigma_i^{\mathsf{p}}$ donc par le lemme 8-E de clôture de Σ_i^{p} par réduction, $B \in \Sigma_i^{\mathsf{p}}$. On en déduit que $\mathsf{PH} = \Sigma_i^{\mathsf{p}}$. □

Une application simple de ce lemme est l'exercice suivant.

✎ **8-L Exercice**

Montrer que si $\mathsf{PH} = \mathsf{PSPACE}$ alors PH s'effondre ($\mathsf{PH} = \Sigma_i^{\mathsf{p}}$ pour un certain i).

La caractérisation de la section précédente nous permet néanmoins d'identifier des problèmes complets pour les classes Σ_i^{p}. Voici la définition du problème $\mathtt{SAT}_i$:

- *entrée :* une formule booléenne sans quantificateur $\varphi(y_1, \ldots, y_i)$, où les y_i sont des uples de variables ;
- *question :* est-il vrai que $\exists y_1 \, \forall y_2 \ldots \mathsf{Q} y_i \, \varphi(y_1, \ldots, y_i)$?

8-M Proposition

Pour $i \geqslant 1$, le problème $\mathtt{SAT}_i$ est Σ_i^{p} complet.

Idée de la démonstration Il s'agit d'utiliser la caractérisation en termes de quantificateurs de la proposition 8-F. Le dernier quantificateur ∃ ou ∀ s'interprète alors comme l'appartenance à un problème NP ou coNP. Grâce au résultat de NP-complétude de SAT (théorème de Cook et Levin 3-V), l'appartenance à ce langage NP ou coNP peut être remplacée par un quantificateur devant une formule booléenne. Au total, on obtient une alternance de i quantificateurs devant une formule booléenne, c'est-à-dire une instance de $\mathtt{SAT}_i$.

Démonstration Si l'on dispose des valeurs de $y_1, \ldots, y_i$ (où pour tout $j \leqslant i$, le uple $y_i \in \{0,1\}^\star$ est de taille au plus linéaire puisqu'il fait partie de l'entrée), décider si $\varphi(y_1, \ldots, y_i)$ est vrai se fait en temps polynomial, donc $\mathtt{SAT}_i \in \Sigma_i^{\mathsf{p}}$ par la caractérisation de la proposition 8-F.
Montrons maintenant que $\mathtt{SAT}_i$ est Σ_i^{p}-difficile. Soit $A \in \Sigma_i^{\mathsf{p}}$: sans perte de généralité, on suppose que A est sur l'alphabet $\{0,1\}$. Par la proposition 8-F, il s'exprime sous la forme

$$x \in A \iff \exists y_1 \in \{0,1\}^{p(|x|)} \, \forall y_2 \in \{0,1\}^{p(|x|)} \ldots \mathsf{Q} y_i \in \{0,1\}^{p(|x|)} \, (x, y_1, \ldots, y_i) \in B,$$

où $B \in \mathsf{P}$ et $p(n)$ est un polynôme.
Nous allons distinguer deux cas selon si Q est ∃ ou ∀ (c'est-à-dire selon si i est impair ou pair) mais les raisonnements seront quasi identiques.

Premier cas : $\mathtt{Q} = \exists$. Le problème de savoir sur l'entrée $(x, y_1, \ldots, y_{i-1})$ si

$$\exists y_i \, (x, y_1, \ldots, y_i) \in B$$

est dans NP. Donc par NP-complétude de SAT (théorème 3-V), il se réduit à SAT : $\exists y_i \, (x, y_1, \ldots, y_i) \in B$ ssi $\exists z \, \varphi_{(x,y_1,\ldots,y_{i-1})}(z)$, où $\varphi_{(x,y_1,\ldots,y_{i-1})}$ est une formule booléenne calculable en temps polynomial. En inspectant la démonstration du théorème 3-V, on voit que les entrées $(x, y_1, \ldots, y_{i-1})$ du problème ne servent dans la formule qu'à définir le contenu du ruban de lecture grâce à l'expression :

$$\bigwedge_{1 \leqslant i \leqslant n} c^1_{x_i,i,0}.$$

Puisqu'ici l'alphabet est $\{0, 1\}$, on peut réécrire cette expression en

$$\bigwedge_{1 \leqslant i \leqslant n} (x_i \wedge c^1_{1,i,0}) \vee (\neg x_i \wedge c^1_{0,i,0})$$

où maintenant les entrées apparaissent sous la forme de variables dans la formule. En d'autres termes, l'inspection de la preuve du théorème 3-V permet d'exprimer $\exists y_i \, (x, y_1, \ldots, y_i) \in B$ sous la forme $\exists z \, \varphi(x, y_1, \ldots, y_{i-1}, z)$, c'est-à-dire que φ ne dépend plus de $(x, y_1, \ldots, y_{i-1})$ mais celles-ci apparaissent maintenant en tant que variables de la formule.
Ainsi, l'appartenance à A s'exprime sous la forme

$$x \in A \iff \exists y_1 \, \forall y_2 \ldots \forall y_{i-1} \exists z \, \varphi(x, y_1, \ldots, y_{i-1}, z).$$

Cette instance de $\mathtt{SAT}_i$ est calculable en temps polynomial, ce qui donne une réduction de A à $\mathtt{SAT}_i$.
Second cas : $\mathtt{Q} = \forall$, on procède symétriquement avec coNP. Le problème de savoir sur l'entrée $(x, y_1, \ldots, y_{i-1})$ si $\forall y_i \, (x, y_1, \ldots, y_i) \in B$ est dans coNP. Donc il se réduit à coSAT : $\forall y_i \, (x, y_1, \ldots, y_i) \in B$ ssi $\forall z \, \varphi(x, y_1, \ldots, y_{i-1}, z)$, où φ est une formule booléenne calculable en temps polynomial. Ainsi, l'appartenance à A s'exprime sous la forme

$$x \in A \iff \exists y_1 \, \forall y_2 \ldots \exists y_{i-1} \forall z \, \varphi(x, y_1, \ldots, y_{i-1}, z).$$

Cette instance de $\mathtt{SAT}_i$ est calculable en temps polynomial, ce qui donne une réduction de A à $\mathtt{SAT}_i$. □

8.2 Comparaison avec les classes probabilistes

Nous avons vu que P ⊆ PH ⊆ PSPACE. Par ailleurs, NP est lui-même un niveau de la hiérarchie polynomiale. Pour avoir une vision plus complète, il reste à situer la classe BPP (définition 6-B) dans ce paysage. Sipser [Sip83] a d'abord montré que BPP ⊆ PH avant que ce résultat ne soit amélioré par Gács puis Lautemann [Lau83] pour donner le théorème suivant.

8-N **Théorème (Sipser, Gács, Lautemann, 1983)**

$$\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{p}} \cap \Pi_2^{\mathsf{p}}$$

Pour montrer ce théorème, nous aurons besoin d'un lemme technique. On désigne par $+$ l'addition des vecteurs de $\{0,1\}^n$ (addition modulo 2, ou encore « ou exclusif » composante par composante). Si $R \subseteq \{0,1\}^n$ et $u \in \{0,1\}^n$, on note $R+u = \{r+u \mid r \in R\}$. Ce lemme montre que si $R \subseteq \{0,1\}^n$ est suffisamment grand, alors un petit nombre de translations de R permet de recouvrir complètement l'ensemble $\{0,1\}^n$.

8-O **Lemme**

Soit $R \subseteq \{0,1\}^n$ tel que $|R| \geqslant (1-\epsilon)2^n$. Soit k tel que $\epsilon^k < 2^{-n}$. Alors il existe $u_1, \ldots, u_k \in \{0,1\}^n$ tels que $\bigcup_{i=1}^{k}(R+u_i) = \{0,1\}^n$.

Démonstration On choisit les vecteurs $u_1, \ldots, u_k \in \{0,1\}^n$ aléatoirement, indépendamment les uns des autres. Soit $r_0 \in \{0,1\}^n$ fixé. Remarquons que $r_0 \in (R+u_i)$ ssi $u_i \in (R+r_0)$. Ainsi, les événements « $r_0 \in (R+u_i)$ » (pour $1 \leqslant i \leqslant k$) sont indépendants dans leur ensemble.

Puisque $|(R+u_i)| = |R| \geqslant (1-\epsilon)2^n$, on a pour tout i que $\Pr_{u_1,\ldots,u_k}(r_0 \notin (R+u_i)) \leqslant \epsilon$. On en déduit par indépendance que $\Pr_{u_1,\ldots,u_k}(r_0 \notin \bigcup_i (R+u_i)) \leqslant \epsilon^k$. Par l'inégalité de Boole (proposition A-B), puisqu'il y a 2^n mots $r \in \{0,1\}^n$,

$$\Pr_{u_1,\ldots,u_k}(\exists r \in \{0,1\}^n,\ r \notin \bigcup_i (R+u_i)) \leqslant 2^n \epsilon^k.$$

Cette probabilité est < 1 puisque $\epsilon^k < 2^{-n}$, ce qui prouve l'existence de vecteurs $u_1, \ldots, u_k$ tels que pour tout $r \in \{0,1\}^n$, $r \in \bigcup_i (R+u_i)$. □

Idée de la démonstration du théorème 8-N Soit $A \in \mathsf{BPP}$: si $x \in A$ alors la plupart des choix probabilistes r mènent à un calcul acceptant. Un petit nombre de translations à partir de ces choix probabilistes permet alors de recouvrir tous les mots d'après le lemme précédent 8-O.

En revanche, si $x \notin A$ alors la plupart des choix probabilistes r mènent à un calcul rejetant et les choix acceptants ne suffisent pas à recouvrir tous les mots par un petit nombre de translations. Une illustration de ces deux cas est donnée à la figure 8.2.

On exprime donc l'appartenance à A avec deux quantificateurs : *il existe* des vecteurs de translations tels que *tout* mot est couvert, ce qui donne un problème Σ_2^{p}.

Démonstration du théorème 8-N Puisque BPP est clos par complément, il suffit de montrer que $\mathsf{BPP} \subseteq \Sigma_2^{\mathsf{p}}$ car cela implique $\mathsf{BPP} = \mathsf{coBPP} \subseteq \mathsf{co}\Sigma_2^{\mathsf{p}} = \Pi_2^{\mathsf{p}}$.

Soit $A \in \mathsf{BPP}$. Par réduction d'erreur (proposition 6-F), il existe un langage $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que pour tout mot x :

$$\begin{cases} x \in A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \geqslant 1 - 2^{-|x|}\,; \\ x \notin A \implies \Pr_{r \in \{0,1\}^{p(|x|)}}((x,r) \in B) \leqslant 2^{-|x|}, \end{cases}$$

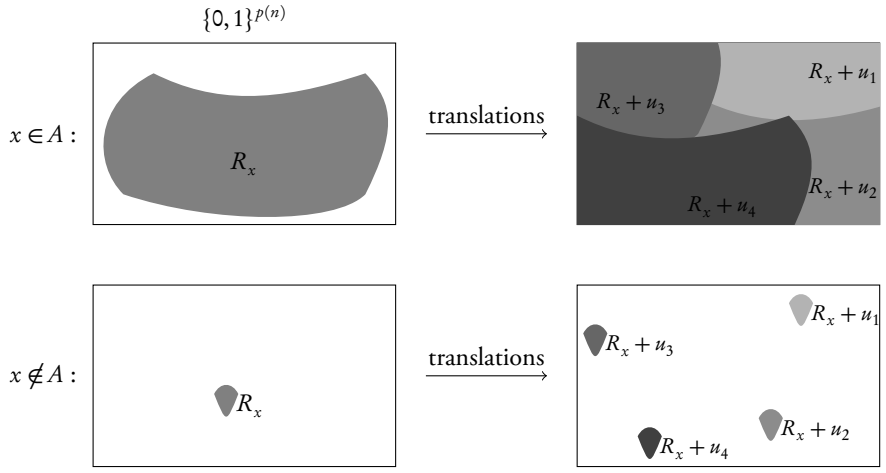


FIGURE 8.2 – Couverture de $\{0,1\}^{p(n)}$ par translation si $x \in A$, impossible si $x \notin A$.

c'est-à-dire un algorithme probabiliste avec erreur $\leqslant \epsilon = 2^{-n}$ utilisant des mots aléatoires r de taille $p(n)$ (où n désigne la taille de x). Soit $k = 1 + \lfloor p(n)/n \rfloor$ de sorte que $\epsilon^k < 2^{-p(n)}$ et $k\epsilon < 1$ (l'entier k étant polynomial et le réel ϵ exponentiellement petit). Pour une entrée x, on note $R_x = \{r \in \{0,1\}^{p(|x|)} \mid (x,r) \in B\}$ l'ensemble des choix probabilistes menant à un calcul acceptant. Soit $x \in \{0,1\}^n$.

– Si $x \in A$, alors $|R_x| \geqslant (1-\epsilon)2^{p(n)}$ donc par le lemme 8-O, il existe $u_1, \ldots, u_k \in \{0,1\}^{p(n)}$ tels que $\cup_{i=1}^k (R_x + u_i) = \{0,1\}^{p(n)}$. En d'autres termes,

$$\exists u_1, \ldots, u_k \in \{0,1\}^{p(n)} \; \forall r \in \{0,1\}^{p(n)} \quad r \in \cup_{i=1}^k (R_x + u_i).$$

– Si $x \notin A$, alors $|R_x| \leqslant \epsilon 2^{p(n)}$ donc pour tous vecteurs $u_1, \ldots, u_k \in \{0,1\}^{p(n)}$, $|\cup_{i=1}^k (R_x + u_i)| \leqslant k\epsilon 2^{p(n)} < 2^{p(n)}$: en particulier, $\cup_{i=1}^k (R_x + u_i) \neq \{0,1\}^{p(n)}$. En d'autres termes,

$$\forall u_1, \ldots, u_k \in \{0,1\}^{p(n)} \; \exists r \in \{0,1\}^{p(n)} \quad r \notin \cup_{i=1}^k (R_x + u_i).$$

Nous avons donc montré que

$$x \in A \iff \exists u_1, \ldots, u_k \in \{0,1\}^{p(n)} \; \forall r \in \{0,1\}^{p(n)} \quad r \in \cup_{i=1}^k (R_x + u_i).$$

Les mots u_i et r sont de taille polynomiale et leur nombre k est aussi polynomial. De plus, puisque $R_x = \{r \in \{0,1\}^{p(|x|)} \mid (x,r) \in B\}$, sur l'entrée $(x, u_1, \ldots, u_k, r)$ on peut tester en temps polynomial si $r \in \cup_{i=1}^k (R_x + u_i)$ comme suit, en utilisant la remarque que $r \in (R_x + u_i) \iff (r + u_i) \in R_x$:

- `pour` i de 1 à k `faire`
  - si $(x, r + u_i) \in B$ alors `accepter` ;
- `rejeter`.

La caractérisation de PH à la proposition 8-F permet de conclure que $A \in \Sigma_2^p$. Ainsi, $\mathsf{BPP} \subseteq \Sigma_2^p$. □

8.3 Liens avec les circuits

Les circuits booléens ont été introduits notamment dans l'espoir de résoudre des questions comme « $\mathsf{P} = \mathsf{NP}$? » grâce à ce modèle plus simple à manipuler. En effet, on sait bien sûr que si NP n'a pas de circuits de taille polynomiale, alors $\mathsf{P} \neq \mathsf{NP}$ (cf. proposition 5-AN).

Mais qu'en est-il de la réciproque : montrer $\mathsf{NP} \neq \mathsf{P}$ est-il aussi difficile que $\mathsf{NP} \not\subset \mathsf{P/poly}$? Nous ne connaissons pas la réponse, mais Karp et Lipton [KL82] ont fait un pas dans cette direction en montrant que la hiérarchie polynomiale s'effondre si NP a des circuits de taille polynomiale. Puisque la hiérarchie polynomiale est supposée être stricte, c'est un signe de plus que NP n'a probablement pas de circuits de taille polynomiale.

8.3.1 Théorème de Karp et Lipton

N'ayant d'abord obtenu qu'un effondrement sur Σ_3^p, dans leur article Karp et Lipton attribuent la version suivante de leur théorème à Sipser.

8-P Théorème (Karp et Lipton, Sipser, 1982)

Si $\mathsf{NP} \subset \mathsf{P/poly}$ alors $\Sigma_2^p = \Pi_2^p$ (et donc $\mathsf{PH} = \Sigma_2^p$).

Idée de la démonstration Il s'agit d'intervertir un quantificateur ∃ avec un quantificateur ∀ pour passer de Π_2^p à Σ_2^p.

Dans une expression Π_2^p de la forme $\forall y \exists z\, (x, y, z) \in B$, sur l'entrée (x, y) la partie $\exists z\, (x, y, z) \in B$ est un langage NP, donc elle se réduit à `SAT`. L'expression Π_2^p se réécrit alors $\forall y\, f(x, y) \in \mathtt{SAT}$, où f est une réduction polynomiale.

Par hypothèse, `SAT` a des circuits de taille polynomiale. L'idée est alors de deviner ces circuits avec un quantificateur ∃ en tête de formule. La partie $f(x, y) \in \mathtt{SAT}$ s'exprime alors par $C(f(x, y)) = 1$ et on obtient $\exists C\, \forall y\, C(f(x, y)) = 1$, mais il reste à vérifier que le circuit C deviné est bien un circuit pour `SAT`.

Pour cela on utilise l'autoréductibilité [1] de `SAT` en exigeant que pour toute formule

1. C'est-à-dire le fait pour un langage A que décider si $x \in A$ se réduit à décider si $y_1, \ldots, y_k \in A$ pour des instances y_i calculables en temps polynomial et de taille strictement inférieure à celle de x.

Attention, cela n'implique pas nécessairement l'existence d'algorithmes efficaces pour A puisqu'en procé-

$\varphi(x_1,\dots,x_k)$, on ait $C(\varphi(x_1,\dots,x_k)) = C(\varphi(0,x_2,\dots,x_k)) \vee C(\varphi(1,x_2,\dots,x_k))$ (condition que l'on écrira $(C,\varphi)\in D$). On a alors transformé l'expression Π_2^{p} de départ en :

$$\exists C\, \forall y\, \forall \varphi\ [(C,\varphi)\in D \wedge C(f(x,y)) = 1],$$

ce qui est une expression Σ_2^{p}.

Pour des raisons de simplicité quant à la taille des arguments, dans la preuve ci-dessous nous n'allons pas manipuler des circuits mais des conseils.

Démonstration La parenthèse de l'énoncé découle du corollaire 8-H.

Pour montrer que $\Sigma_2^{\mathsf{p}} = \Pi_2^{\mathsf{p}}$ il suffit de montrer que $\Pi_2^{\mathsf{p}} \subseteq \Sigma_2^{\mathsf{p}}$ car cela implique $\mathrm{co}\Pi_2^{\mathsf{p}} \subseteq \mathrm{co}\Sigma_2^{\mathsf{p}}$, soit $\Sigma_2^{\mathsf{p}} \subseteq \Pi_2^{\mathsf{p}}$.

Soit $A \in \Pi_2^{\mathsf{p}}$: l'appartenance à A s'exprime sous la forme

$$x\in A \iff \forall y \in \{0,1\}^{p(|x|)}\, \exists z \in \{0,1\}^{p(|x|)}\, (x,y,z)\in B,$$

où $B\in\mathsf{P}$ et $p(n)$ est un polynôme. Sur l'entrée (x,y), décider si

$$\exists z \in \{0,1\}^{p(|x|)}\, (x,y,z)\in B$$

est un problème de NP et se réduit donc à SAT par une réduction polynomiale f :

$$\exists z \in \{0,1\}^{p(|x|)}\, (x,y,z)\in B \iff f(x,y)\in \mathtt{SAT}.$$

Si $|x| = n$ et $|y| = p(n)$, on notera $q(n)$ la taille polynomiale de $f(x,y)$. Mais par hypothèse, $\mathtt{SAT}\in \mathsf{P/poly}$ donc il est reconnu par une machine déterministe polynomiale avec conseils (a_n) de taille polynomiale $r(n)$:

$$\varphi\in\mathtt{SAT} \iff (\varphi, a_{|\varphi|})\in C, \quad \text{où } C\in\mathsf{P}.$$

En combinant ces expressions, on obtient que

$$x\in A \iff \forall y\in\{0,1\}^{p(n)}\, (f(x,y), a_{q(n)})\in C. \tag{8.3}$$

Sans perte de généralité, on supposera que le conseil a_m convient pour toutes les entrées de taille $\leqslant m$, et non seulement celles de longueur exactement égale à m (il suffit pour cela d'adjoindre à a_m les conseils $a_0,\dots,a_{m-1}$).

Le problème est qu'on ne sait pas construire le conseil a_m. L'astuce est alors de le deviner et de vérifier qu'il s'agit d'un bon conseil.

Pour la suite, si $\varphi = \varphi(x_1,\dots,x_k)$ est une formule booléenne avec k variables libres, on notera $\varphi_0(x_2,\dots,x_k) = \varphi(0,x_2,\dots,x_k)$ et $\varphi_1(x_2,\dots,x_k) = \varphi(1,x_2,\dots,x_k)$ les formules booléennes à $k-1$ variables libres où x_1 est remplacé respectivement par 0 et 1. On remarquera que la taille du codage de φ_0 et de φ_1 est inférieure à celle de φ puisqu'une variable désignée par un numéro de $\log k$ bits est remplacée par 0 ou 1.

Que doit vérifier un conseil a_m de taille $r(m)$ pour être un bon conseil pour SAT ? Voici trois conditions nécessaires :

dant de manière récursive sur des entrées plus petites, le nombre d'instances à tester croît exponentiellement a priori.

1. si φ de taille $\leqslant m$ n'a pas de variables et est vraie, alors $(\varphi, a_m) \in C$;
2. de même si φ de taille $\leqslant m$ n'a pas de variables et est fausse, alors $(\varphi, a_m) \notin C$;
3. pour tout φ de taille $\leqslant m$, $(\varphi, a_m) \in C$ ssi $[(\varphi_0, a_m) \in C$ ou $(\varphi_1, a_m) \in C]$.

Ces trois conditions sont aussi suffisantes comme on le montre maintenant.

Démonstration — Soit a_m un conseil vérifiant les trois conditions ci-dessus. Montrons qu'il s'agit d'un bon conseil par récurrence sur le nombre de variables de φ.

Si φ n'a pas de variable, alors les conditions 1 et 2 assurent que le conseil permet d'obtenir la bonne réponse.

Si φ a $k > 0$ variables $x_1, \ldots, x_k$, alors φ est satisfaisable ssi φ_0 ou φ_1 l'est (en effet, il faut bien donner la valeur 0 ou 1 à la variable x_1). Par hypothèse de récurrence, a_m permet d'obtenir la bonne réponse sur φ_0 et φ_1 qui ont chacune $k-1$ variables. Ainsi, la condition 3 garantit que le conseil permet d'obtenir la bonne réponse sur φ. ◇

Ainsi, un conseil satisfaisant les trois conditions ci-dessus est un bon conseil pour SAT. Pour résoudre SAT, on peut donc deviner un conseil et vérifier qu'il satisfait ces trois conditions. Pour notre problème A, cela va nous permettre d'échanger un quantificateur $\exists$ par un $\forall$, quitte à ajouter un $\exists$ en tête…

Grâce à l'expression (8.3), l'appartenance à A s'écrit alors

$$x \in A \iff \exists a \in \{0,1\}^{r(n)} \forall y \in \{0,1\}^{p(n)} (\text{« } a \text{ est un bon conseil »} \wedge (f(x,y),a) \in C).$$

Vérifier que « a est un bon conseil » s'exprime ainsi :

$$\forall \varphi \in \{0,1\}^{\leqslant q(n)} (\varphi, a) \in D$$

où D est le langage défini selon les trois conditions ci-dessus :

$$\begin{aligned}(\varphi,a) \in D \iff & (\varphi \equiv 1 \to (\varphi,a) \in C) \\ & \wedge (\varphi \equiv 0 \to (\varphi,a) \notin C) \\ & \wedge ((\varphi,a) \in C \leftrightarrow [((\varphi_0,a) \in C) \vee ((\varphi_1,a) \in C)]).\end{aligned}$$

On a bien sûr $D \in \mathsf{P}$ puisqu'il s'agit simplement d'évaluer φ si elle n'a pas de variables et de tester des appartenances à $C \in \mathsf{P}$. On obtient donc :

$$x \in A \iff \exists a \in \{0,1\}^{r(n)} \forall y \in \{0,1\}^{p(n)} \forall \varphi \in \{0,1\}^{\leqslant q(n)} (\varphi,a) \in D \wedge (f(x,y),a) \in C.$$

C'est une expression Σ_2^{p}, donc $A \in \Sigma_2^{\mathsf{p}}$. □

8.3.2 Langages creux

On se rappelle du résultat du corollaire 7-Y : un langage A est dans P/poly si et seulement s'il se réduit à un langage creux par une réduction Turing polynomiale. Ainsi, $\mathsf{NP} \subset \mathsf{P/poly}$ ssi $\mathtt{SAT} \in \mathsf{P/poly}$ ssi $\mathtt{SAT} \leqslant_{\mathsf{T}}^{\mathsf{p}} C$ où C est un langage creux. Le langage creux C est donc NP-difficile pour les réductions Turing polynomiales.

On peut alors reformuler le théorème de Karp et Lipton en termes de réduction Turing : NP ⊂ P/poly devient « il existe un langage creux NP-difficile pour les réductions Turing polynomiales ». Ce n'est pas sans ressemblance avec le théorème de Mahaney 3-AQ. Nous pouvons ainsi comparer ces deux théorèmes.

8-Q **Corollaire (reformulation des théorèmes de Mahaney et de Karp-Lipton)**

1. S'il existe un langage creux NP-difficile pour les réductions many-one polynomiales, alors P = NP (Mahaney).
2. S'il existe un langage creux NP-difficile pour les réductions Turing polynomiales, alors $\Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$ (Karp et Lipton).

Ce sont donc deux résultats semblables, l'un portant sur les réductions many-one, l'autre sur les réductions Turing. Puisque tout langage NP-difficile pour les réductions many-one l'est aussi pour les réductions Turing, l'hypothèse du premier point est plus forte. Mais la conclusion l'est aussi.

8.4 Borne inférieure sur le temps et l'espace conjugués pour NTIME(n)

En général, il est très difficile d'obtenir des bornes inférieures sur le temps de calcul nécessaire à la résolution d'un problème. Par exemple, la meilleure borne inférieure connue à ce jour sur le temps déterministe nécessaire pour résoudre un problème de NTIME(n) est linéaire. Il est donc intéressant de constater que si l'on contraint également l'espace, alors nous pouvons faire mieux, et la démonstration utilise une alternance de quantificateurs comme dans la hiérarchie polynomiale. La technique trouve ses prémices dans Nepomnjaščiĭ [Nep70] et dans Kannan [Kan83] ; l'application à TISP a d'abord été faite par Fortnow [For97] avant que Fortnow et van Melkebeek [FM00] n'obtiennent le résultat suivant. Depuis, de meilleures bornes inférieures ont été montrées, par Williams [Wil07] par exemple.

8-R **Définition**

La classe TISP($t(n), s(n)$) (pour TIme-SPace) est l'ensemble des langages reconnus par une machine de Turing déterministe fonctionnant en temps $O(t(n))$ *et* utilisant un espace $O(s(n))$.

On peut alors montrer le résultat suivant.

8-S **Théorème (Fortnow et van Melkebeek, 2000)**

Soit $\alpha \geqslant 1$ et β deux constantes telles que $\alpha(\alpha+\beta)<2$. Alors $\mathsf{NTIME}(n) \not\subseteq \mathsf{TISP}(n^\alpha, n^\beta)$.
Par exemple, $\alpha = 5/4$ et $\beta = 3/10$ conviennent.

Idée de la démonstration À l'aide de deux quantificateurs (calcul de type Σ_2^{p}), on peut accélérer le calcul $M(x)$ d'une machine $\mathsf{TISP}(t(n), s(n))$ en devinant $O(\sqrt{t(n)/s(n)})$ configurations intermédiaires $C'_1, \ldots, C'_m$ régulièrement espacées et vérifier pour tout i que C'_i mène à C'_{i+1} en $\sqrt{t(n)s(n)}$ étapes : chaque configuration étant de taille $O(s(n))$, cela prend un temps $O(\sqrt{t(n)s(n)})$.
Mais si $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(n^\alpha, n^\beta)$ alors le second quantificateur peut être remplacé par un calcul déterministe prenant un temps n^α, ce qui permet d'accélérer un calcul $\mathsf{TISP}(t(n), s(n))$ grâce à une machine non déterministe (c'est-à-dire avec un seul quantificateur cette fois).
Pour un bon choix des paramètres α et β, l'hypothèse $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(n^\alpha, n^\beta)$ implique donc l'accélération d'un calcul non déterministe par une machine non déterministe, une contradiction avec le théorème de hiérarchie en temps non déterministe.

La démonstration de ce théorème repose sur le lemme suivant.

8-T **Lemme**

Soit $t(n) \geqslant n^2$ et $s(n)$ deux fonctions constructibles en temps. Alors

$$\mathsf{TISP}(t(n), s(n)) \subseteq \mathsf{NTIME}(\sqrt{t(n)s(n)})^{\mathsf{NTIME}(n)}.$$

Démonstration Soit L un langage de $\mathsf{TISP}(t(n), s(n))$ reconnu par une machine déterministe M fonctionnant en temps $O(t(n))$ et en espace $O(s(n))$. Sur une entrée x, le calcul $M(x)$ passe par $O(t(n))$ configurations $C_0, C_1, \ldots, C_k$, et chacune d'entre elles peut être décrite par un mot de taille $O(s(n))$ puisque l'espace utilisé par la machine est $O(s(n))$.
Voici maintenant comment reconnaître L dans $\mathsf{NTIME}(\sqrt{t(n)s(n)})^{\mathsf{NTIME}(n)}$ (se reporter à la figure 8.3) :

- deviner $O(\sqrt{t(n)/s(n)})$ configurations $C'_1, \ldots, C'_m$, et une configuration finale C'_f ;
- grâce à l'oracle $\mathsf{NTIME}(n)$, vérifier que pour tout i, le calcul déterministe $M(x)$ passe bien de C'_i à C'_{i+1} en $\sqrt{t(n)s(n)}$ étapes, et de même de C_0 à C'_1 et de C'_m à C'_f ;
- `accepter` ssi C'_f est une configuration acceptante.

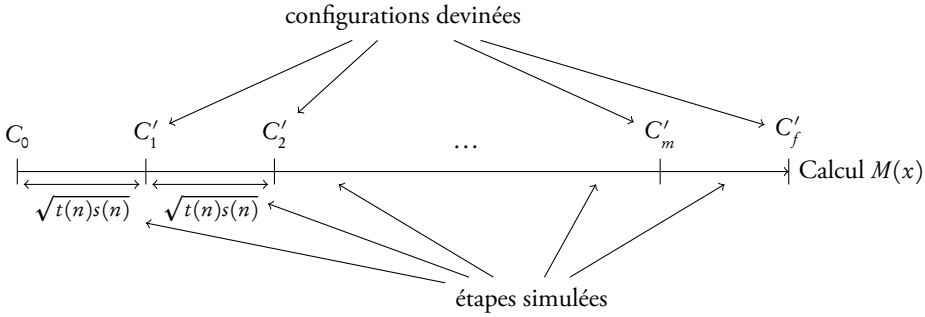


FIGURE 8.3 – Accélération du calcul en devinant $m = O(\sqrt{t(n)/s(n)})$ configurations C'_i régulièrement espacées dans le calcul $M(x)$.

L'oracle est ici utilisé comme un quantificateur universel permettant de tester la condition sur tous les i. L'existence d'un chemin acceptant dans cette procédure garantit que $M(x)$ passe par les configurations C'_i pour finalement accepter. Réciproquement, si $M(x)$ accepte alors cette procédure non déterministe possède un chemin acceptant puisqu'il suffit de deviner les bonnes configurations C'_i. Ainsi, cette procédure reconnaît le langage L.

Puisque chacune des $O(\sqrt{t(n)/s(n)})$ configurations devinées a une taille $O(s(n))$, la première étape prend un temps non déterministe $O(\sqrt{t(n)s(n)})$. Quant à la seconde étape, puisque $t(n) \geqslant n^2$ la question posée à l'oracle a une taille $O(\sqrt{t(n)s(n)})$ (on fournit x et la description de toutes les configurations C'_i) et pour chaque i fixé, il suffit de vérifier qu'un calcul de longueur $\sqrt{t(n)s(n)}$ va de C'_i à C'_{i+1}, ce qui prend $O(\sqrt{t(n)s(n)})$ étapes soit, en fonction de la taille de l'entrée, un temps linéaire. Le test universel « pour tout i » effectué par l'oracle se fait donc dans coNTIME(n), on en conclut que $L \in \mathsf{NTIME}(\sqrt{t(n)s(n)})^{\mathsf{NTIME}(n)}$. □

8-U **Remarque** Dans la preuve précédente, on remarquera que l'oracle utilise en fait seulement $O(\log(t(n)/s(n)))$ bits non déterministes puisqu'il suffit de vérifier une condition pour chaque i de 1 à $m = O(\sqrt{t(n)/s(n)})$. Mais il lui faut un temps de calcul (déterministe) linéaire pour simuler $\sqrt{t(n)s(n)}$ étapes de $M(x)$.

Nous pouvons maintenant montrer le théorème.

Démonstration du théorème 8-S Supposons que $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(n^\alpha, n^\beta)$. Soit $\gamma > 1$ une constante telle que $\gamma\alpha > 2$ et soit $L \in \mathsf{NTIME}(n^\gamma)$. Nous allons obtenir une contradiction en montrant que $L \in \mathsf{NTIME}(n^\delta)$ pour un certain $\delta < \gamma$, ce qui viole le théorème de hiérarchie en temps non déterministe 2-AI.

Par un padding similaire à la preuve du théorème 4-AZ, l'hypothèse $\mathsf{NTIME}(n) \subseteq \mathsf{TISP}(n^\alpha, n^\beta)$ implique

$$\mathsf{NTIME}(n^\gamma) \subseteq \mathsf{TISP}(n^{\alpha\gamma} \log n, n^{\beta\gamma}).$$

Démonstration — Pour un langage $A \in \mathsf{NTIME}(n^\gamma)$, on considère le langage « délayé » $\tilde{A} = \{(x, 1^{|x|^\gamma}) \mid x \in A\}$: alors $\tilde{A} \in \mathsf{NTIME}(n)$ donc par hypothèse il existe une machine déterministe $\tilde{M}$ fonctionnant en temps $O(n^\alpha)$ et en espace $O(n^\beta)$ pour $\tilde{A}$.
On construit alors une machine déterministe M pour A comme suit : $M(x)$ simule $\tilde{M}(x, 1^{|x|^\gamma})$ sans écrire $1^{|x|^\gamma}$ mais en maintenant un compteur binaire c pour connaître la position de la tête sur le ruban de lecture de $\tilde{M}$. Le compteur c permet ainsi de savoir quelle lettre est lue en entrée. La simulation d'une étape de $\tilde{M}$ nécessite de mettre à jour le compteur c, ce qui prend un temps $O(\log n)$, donc M fonctionne en temps $O(n^{\alpha\gamma} \log n)$; l'espace utilisé est quant à lui $O(\log n) + O(n^{\beta\gamma}) = O(n^{\beta\gamma})$. ◇

On en déduit que $L \in \mathsf{TISP}(n^{\alpha\gamma} \log n, n^{\beta\gamma})$.
Puisqu'on a trivialement l'inclusion $\mathsf{TISP}(n^\alpha, n^\beta) \subseteq \mathsf{DTIME}(n^\alpha)$, en particulier l'hypothèse implique que $\mathsf{NTIME}(n) \subseteq \mathsf{DTIME}(n^\alpha)$, donc par le lemme 8-T (que l'on peut appliquer car $\alpha\gamma > 2$)

$$\mathsf{TISP}(n^{\gamma\alpha} \log n, n^{\gamma\beta}) \subseteq \mathsf{NTIME}(n^{\gamma(\alpha+\beta)/2} \log n)^{\mathsf{DTIME}(n^\alpha)}.$$

Mais on peut directement simuler l'oracle $\mathsf{DTIME}(n^\alpha)$ lorsqu'on lui pose une question : puisque les questions sont de taille $O(n^{\gamma(\alpha+\beta)/2} \log n)$, on obtient alors que

$$\mathsf{NTIME}(n^{\gamma(\alpha+\beta)/2} \log n)^{\mathsf{DTIME}(n^\alpha)} \subseteq \mathsf{NTIME}((n^{\gamma(\alpha+\beta)/2} \log n)^\alpha).$$

On en déduit que

$$\mathsf{TISP}(n^{\gamma\alpha} \log n, n^{\gamma\beta}) \subseteq \mathsf{NTIME}(n^{\alpha\gamma(\alpha+\beta)/2} (\log n)^\alpha).$$

Ainsi, $L \in \mathsf{NTIME}(n^{\alpha\gamma(\alpha+\beta)/2} (\log n)^\alpha)$.
Puisque L était un langage quelconque de $\mathsf{NTIME}(n^\gamma)$, on en déduit que

$$\mathsf{NTIME}(n^\gamma) \subseteq \mathsf{NTIME}(n^{\alpha\gamma(\alpha+\beta)/2} (\log n)^\alpha),$$

une contradiction avec le théorème de hiérarchie non déterministe en temps 2-AI si $\alpha(\alpha+\beta)/2 < 1$. □

8-V **Remarque** Avec un peu plus de travail, cette borne inférieure s'applique au problème SAT (qui est un cas particulier de problème $\mathsf{NTIME}(n)$), c'est-à-dire $\mathtt{SAT} \notin \mathsf{TISP}(n^\alpha, n^\beta)$ si $\alpha(\alpha+\beta) < 2$. En effet, on peut montrer une version du théorème de Cook-Levin 3-V où la réduction à SAT est si facile à calculer que $\mathtt{SAT} \in \mathsf{TISP}(n^\alpha, n^\beta)$ implique $\mathsf{NTIME}(n) \in \mathsf{TISP}(n^{\alpha+\epsilon}, n^{\beta+\epsilon})$ pour tout ϵ.

✎ **8-W** **Exercice**

Montrer la remarque précédente.

9 Comptage

Jusqu'à présent, nous nous sommes surtout intéressés aux problèmes de décision, c'est-à-dire ceux qui appellent une réponse « oui ou non ». Mais on peut également vouloir calculer des fonctions $f : \Sigma^\star \to \Sigma^\star$. En particulier, plutôt que de décider comme dans NP s'il existe une solution à un problème, on peut vouloir *compter* le nombre de solutions. De là vient la définition de la classe de comptage ♯P (« dièse-p » en français, « sharp-pi » en anglais).

Cette classe et ses variantes ont plusieurs intérêts en complexité. D'abord ce sont des généralisations naturelles de classes vues jusqu'à présent, et notamment nous verrons que la définition de la classe PP (une de ses variantes) découle naturellement de celles des classes probabilistes. Par ailleurs, de nombreux problèmes intéressants sont complets pour ces classes, et c'est probablement la complétude du permanent pour ♯P qui assoit la légitimité de la définition : ce polynôme joue en effet un rôle central en complexité (booléenne et algébrique), en théorie des graphes, en physique théorique, etc.

Enfin, si à première vue compter le nombre de solutions d'un problème peut sembler de difficulté comparable à décider l'existence d'une solution, nous verrons qu'il n'en est rien : le comptage apporte une puissance surprenante et compter le nombre de solutions semble beaucoup plus difficile que décider s'il en existe une. Le théorème 9-AI de Toda nous donne en effet une idée de la puissance qu'apporte le comptage : elle dépasse celle de toute la hiérarchie polynomiale.

9.1 Définitions

Dans cette première partie nous verrons les définitions de base des classes de comptage. Le terme « comptage » laisse penser à des fonctions dont la valeur « compte » le nombre de solutions d'un problème. En effet, certaines des classes importantes sont des classes

de fonctions. Mais plus généralement, on considère aussi des classes de langages lorsque celles-ci partagent certaines caractéristiques avec la classe de fonctions ♯P que l'on va voir en premier.

9.1.1 Classes de fonctions

La classe ♯P a été introduite par Valiant [Val79b].

9-A Définition (♯P)

La classe ♯P est une classe de *fonctions à valeurs entières* $f : \Sigma^\star \to \mathbb{N}$, où Σ est un alphabet fini quelconque.

Une fonction $f : \Sigma^\star \to \mathbb{N}$ est dans ♯P s'il existe un langage $A \in \mathsf{P}$ et un polynôme $p(n)$ tels que pour tout $x \in \Sigma^\star$,

$$f(x) = |\{y \in \Sigma^{p(|x|)} \mid (x,y) \in A\}|.$$

En d'autres termes, $f(x)$ « compte » le nombre de mots y satisfaisant $(x,y) \in A$.

9-B Remarque Par définition, si $f \in \sharp\mathsf{P}$ et $x \in \Sigma^\star$, alors $0 \leqslant f(x) \leqslant |\Sigma|^{p(|x|)}$ pour le polynôme $p(n)$ de la définition. Ainsi, le nombre de chiffres de $f(x)$ est polynomial.

Une autre caractérisation (souvent donnée comme définition) est la suivante.

9-C Proposition

Une fonction f est dans ♯P si et seulement s'il existe une machine de Turing non déterministe N fonctionnant en temps polynomial telle que pour tout x, $f(x)$ est le nombre de chemins acceptants de $N(x)$.

Idée de la démonstration Le langage A de la définition correspond simplement aux couples (x,y) tels que y désigne un chemin acceptant dans le calcul $N(x)$. □

✎ **9-D Exercice**

Montrer la proposition précédente.

Il est facile de voir que l'ensemble des fonctions $f : \Sigma^\star \to \mathbb{N}$ calculables en temps polynomial (c'est-à-dire que le nombre $f(x)$ en binaire est calculable en temps polynomial à partir de x), noté FP, est contenu dans ♯P.

9-E Proposition

Si $f : \Sigma^\star \to \mathbb{N}$ est calculable en temps polynomial, alors $f \in \sharp\mathsf{P}$.

Démonstration Puisque f est calculable en temps polynomial, le nombre de chiffres de $f(x)$ est polynomial en $|x|$. Il existe donc un polynôme $p(n)$ tel que $f(x) < |\Sigma|^{p(|x|)}$. On définit $A = \{(x,y) \mid y < f(x)\}$ sur l'alphabet Σ, où y est considéré comme le code d'un entier en base $|\Sigma|$. Alors $A \in \mathsf{P}$ et $f(x) = |\{y \in \Sigma^{p(|x|)} \mid (x,y) \in A\}|$, ce qui montre que $f \in \sharp\mathsf{P}$. □

Les fonctions de $\sharp\mathsf{P}$ ne prennent que des valeurs positives. On définit alors une autre classe appelée GapP autorisant les valeurs négatives.

9-F Définition

La classe GapP est l'ensemble des fonctions $g : \Sigma^\star \to \mathbb{Z}$ telles qu'il existe deux fonctions $f, f' \in \sharp\mathsf{P}$ satisfaisant pour tout $x \in \Sigma^\star$,

$$g(x) = f(x) - f'(x).$$

En d'autres termes, GapP est la clôture de $\sharp\mathsf{P}$ par soustraction.

Une nouvelle fois, on peut caractériser GapP en termes de nombre de chemins d'un calcul non déterministe.

9-G Proposition

Une fonction $g : \Sigma^\star \to \mathbb{Z}$ est dans GapP si et seulement s'il existe une machine de Turing non déterministe N fonctionnant en temps polynomial telle que pour tout $x \in \Sigma^\star$,

$$g(x) = \sharp\mathrm{acc}(N(x)) - \sharp\mathrm{rej}(N(x)),$$

où $\sharp\mathrm{acc}(N(x))$ (respectivement $\sharp\mathrm{rej}(N(x))$) désigne le nombre de chemins acceptants (resp. rejetant) de $N(x)$.

Idée de la démonstration Pour toute machine non déterministe N, on note $\tilde{N}$ la machine fonctionnant comme N mais, lorsque N arrive dans un état rejetant, $\tilde{N}$ crée exactement un chemin acceptant et un chemin rejetant. Une telle machine vérifie

$$\sharp\mathrm{acc}(N(x)) = \sharp\mathrm{acc}(\tilde{N}(x)) - \sharp\mathrm{rej}(\tilde{N}(x)).$$

Soit $g = f - f' \in \mathsf{GapP}$ et soit N et N' les machines non déterministes dont f et f' comptent les chemins acceptants (proposition 9-C).

On construit la machine non déterministe N_g suivante sur l'entrée x :

– deviner $b \in \{0,1\}$;
– si $b = 0$, simuler $\tilde{N}(x)$;
– si $b = 1$, simuler $\tilde{N}'(x)$ en inversant les états acceptant et rejetant.

On a alors

$$\begin{aligned}\sharp\mathrm{acc}(N_g(x)) - \sharp\mathrm{rej}(N_g(x)) = \\ &\left(\sharp\mathrm{acc}(\tilde{N}(x)) + \sharp\mathrm{rej}(\tilde{N}'(x))\right) - \left(\sharp\mathrm{rej}(\tilde{N}(x)) + \sharp\mathrm{acc}(\tilde{N}'(x))\right) \\ &= \left(\sharp\mathrm{acc}(\tilde{N}(x)) - \sharp\mathrm{rej}(\tilde{N}(x))\right) - \left(\sharp\mathrm{acc}(\tilde{N}'(x)) - \sharp\mathrm{rej}(\tilde{N}'(x))\right) \\ &= \sharp\mathrm{acc}(N(x)) - \sharp\mathrm{acc}(N'(x)) = f(x) - f'(x) = g(x).\end{aligned}$$

Réciproquement, si $g(x) = \sharp\mathrm{acc}(N(x)) - \sharp\mathrm{rej}(N(x))$ alors il suffit de définir $f(x) = \sharp\mathrm{acc}(N(x))$ et $f'(x) = \sharp\mathrm{rej}(N(x))$, deux fonctions $\sharp\mathsf{P}$ (cf. proposition 9-C), et on a $g = f - f'$. □

✎ **9-H** **Exercice**

Donner les détails de la démonstration de la proposition précédente.

✎ **9-I** **Exercice**

Soit $g \in \mathsf{GapP}$. Montrer qu'il existe $f \in \sharp\mathsf{P}$ et $f' \in \mathsf{FP}$ (fonction calculable en temps polynomial) telles que $g = f - f'$.

9.1.2 Classes de langages

Les classes que nous allons voir maintenant contiennent des *problèmes de décision* et non plus des fonctions. Elles font tout de même partie de ce qu'on appelle les « classes de comptage » par analogie avec les classes vues ci-dessus.

Dans la première, on cherche en quelque sorte à savoir si le bit de poids fort d'une fonction $\sharp\mathsf{P}$ est 0 ou 1, ou encore si une majorité de certificats sont valides. Il serait donc légitime de l'appeler « Majorité-P » mais son nom PP, qui vient de *Probabilistic Polynomial time*, a été proposé avant celui des classes probabilistes du chapitre 6. Cette classe a été introduite par Gill [Gil77].

9-J Définition

La classe PP est l'ensemble des langages A tels qu'il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ satisfaisant pour tout $x \in \Sigma^\star$:

$$x \in A \iff |\{y \in \Sigma^{p(|x|)} \mid (x,y) \in B\}| \geqslant \frac{1}{2}|\Sigma|^{p(|x|)}.$$

En d'autres termes, la majorité des mots $y \in \Sigma^{p(|x|)}$ doit satisfaire $(x,y) \in B$. On peut donc réécrire la définition sous la forme

$$\Pr_{y \in \Sigma^{p(|x|)}}\big((x,y) \in B\big) \geqslant 1/2.$$

✎ **9-K Exercice**

Montrer qu'à nouveau, il existe une caractérisation de PP en termes de calculs acceptants d'une machine N non déterministe polynomiale : $x \in A$ ssi au moins la moitié des chemins de $N(x)$ sont acceptants.

On peut également faire le lien avec la classe GapP vue ci-dessus.

9-L Proposition

Un langage A est dans PP si et seulement s'il existe une fonction $g \in \mathsf{GapP}$ telle que pour tout $x \in \Sigma^\star$,

$$x \in A \iff g(x) \geqslant 0.$$

Démonstration Dans le sens $\Rightarrow$, soit $B \in \mathsf{P}$ le langage et $p(n)$ le polynôme pour A dans la définition de PP. Soit $f \in \sharp\mathsf{P}$ la fonction $f(x) = |\{y \in \Sigma^{p(|x|)} \mid (x,y) \in B\}|$, et soit $f'(x) = \lceil (1/2)|\Sigma|^{p(|x|)} \rceil$, fonction elle aussi dans $\sharp\mathsf{P}$ puisque

$$f'(x) = |\{y \in \Sigma^{p(|x|)} \mid y < \lceil (1/2)|\Sigma|^{p(|x|)} \rceil\}|$$

où y représente un nombre écrit en base $|\Sigma|$. Alors la fonction $g = f - f'$ est dans GapP et satisfait $x \in A$ ssi $g(x) \geqslant 0$.

Réciproquement, par la caractérisation de la proposition 9-G, $g(x) = \sharp\mathrm{acc}(N(x)) - \sharp\mathrm{rej}(N(x))$ pour une machine non déterministe N fonctionnant en temps polynomial $p(n)$. Sans perte de généralité, on suppose que tout $y \in \Sigma^{p(|x|)}$ désigne un chemin valide de $N(x)$, qui est soit acceptant soit rejetant. Soit

$$B = \{(x,y) \mid y \text{ est un chemin acceptant de } N(x)\} \in \mathsf{P}.$$

Alors $x \in A$ ssi $\sharp\mathrm{acc}(N(x)) \geqslant \sharp\mathrm{rej}(N(x))$ ssi $(x,y) \in B$ pour au moins la moitié des mots y, ce qui est la définition de $A \in \mathsf{PP}$. □

9-M **Remarque** En particulier, pour remplacer le seuil $(1/2)|\Sigma|^{p(n)}$ de la définition de PP, toute fonction $f(n)$ calculable en temps polynomial convient.

Pour l'exercice suivant et au théorème 9-AI, on utilisera $\sharp\mathsf{P}$ en oracle. Disposer d'une fonction $f : \Sigma^\star \to \mathbb{N}$ en oracle signifie qu'au moment d'une requête u, le contenu du ruban d'oracle passe en temps unitaire à $f(u)$ écrit en binaire. Et comme d'habitude, $\mathsf{P}^{\sharp\mathsf{P}} = \bigcup_{f\in\sharp\mathsf{P}} \mathsf{P}^f$.

✎ **9-N** **Exercice**

Montrer que $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\sharp\mathsf{P}} = \mathsf{P}^{\mathsf{GapP}}$.

Dans le même esprit que $\sharp\mathsf{P}$, on trouve également des classes de comptage « modulo », c'est-à-dire où le nombre de solutions est compté modulo un entier k.

9-O **Définition**

Soit $k \geqslant 2$ un entier. La classe $\mathsf{Mod}_k\mathsf{P}$ est l'ensemble des langages A tels qu'il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ vérifiant pour tout $x \in \Sigma^\star$:

$$x \in A \iff |\{y \in \Sigma^{p(|x|)} \mid (x,y) \in B\}| \not\equiv 0 \mod k.$$

Lorsque $k = 2$, la classe $\mathsf{Mod}_2\mathsf{P}$ est aussi appelée $\oplus\mathsf{P}$ (« parité-p »).

De manière équivalente, il existe une fonction $f \in \sharp\mathsf{P}$, $f : \Sigma^\star \to \mathbb{N}$, telle que

$$x \in A \iff f(x) \not\equiv 0 \mod k.$$

Comme on le verra ci-dessous (proposition 9-AD), si k est un nombre premier alors dans la condition on peut remplacer « $\not\equiv 0 \mod k$ » par « $\equiv 0 \mod k$ ».

9.2 Premiers résultats de complétude

L'un des résultats de complétude les plus importants et les plus intéressants pour les classes de comptage est la complétude du permanent pour $\sharp\mathsf{P}$ que nous verrons à la fin du chapitre. En attendant, nous allons voir des résultats plus simples. Mais nous devons d'abord définir une notion de réduction entre fonctions.

9.2.1 Réductions pour les classes de fonctions

Les réductions vues jusqu'à présent (many-one et Turing) n'ont de sens que pour les langages. Dans une réduction many-one e de A à B, on transforme l'entrée x de A afin d'obtenir la même réponse avec B, c'est-à-dire que la transformée $e(x)$ satisfait $x \in A \iff e(x) \in B$. Une manière naturelle d'étendre cette notion aux fonctions est donc de transformer l'entrée x afin d'obtenir le même résultat avec les deux fonctions, comme suit.

9-P Définition (réduction parcimonieuse)

Soit $f, g : \Sigma^\star \to \mathbb{Z}$ deux fonctions. On dit que f se réduit à g par une *réduction parcimonieuse*, et on note $f \leqslant^{\mathsf{P}}_{\text{parci}} g$, s'il existe une fonction $e : \Sigma^\star \to \Sigma^\star$ calculable en temps polynomial telle que

$$\text{pour tout } x \in \Sigma^\star, \quad f(x) = g(e(x)).$$

Bien qu'importante, cette réduction s'avère parfois trop faible et la notion plus générale de réduction de comptage autorise également un calcul sur la sortie de $g(e(x))$.

9-Q Définition (réduction de comptage entre fonctions)

Soit $f, g : \Sigma^\star \to \mathbb{Z}$ deux fonctions. On dit que f se réduit à g par une *réduction de comptage* en temps polynomial, et on note $f \leqslant^{\mathsf{P}}_{\mathsf{c}} g$, s'il existe deux fonctions $e, s : \Sigma^\star \to \Sigma^\star$ calculables en temps polynomial telles que

$$\text{pour tout } x \in \Sigma^\star, \quad f(x) = s(g(e(x))).$$

9-R Remarque La fonction de sortie s fait que cette réduction de comptage est l'analogue d'une réduction Turing où un seul appel à l'oracle serait permis.

On remarquera que cette notion de réduction reste bien sûr transitive. C'est cette réduction qui sert à définir la ♯P-complétude.

9-S Définition (♯P-complétude)

Une fonction $f : \Sigma^\star \to \mathbb{N}$ est ♯P-complète si

- $f \in \sharp\mathsf{P}$;
- pour tout $g \in \sharp\mathsf{P}$, $g \leqslant^{\mathsf{P}}_{\mathsf{c}} f$.

Malgré cette définition générale, il n'en reste pas moins que la réduction dans les résultats de ♯P-complétude est souvent parcimonieuse, avec le permanent faisant figure d'exception notoire.

9.2.2 Complétude pour ♯P

Soit ♯3SAT le problème de compter le nombre de solutions d'une formule en 3-CNF, c'est-à-dire la fonction $f : \{0,1\}^\star \to \mathbb{N}$ telle que pour toute formule $\varphi(x_1,\dots,x_n)$ en 3-CNF codée sur l'alphabet $\{0,1\}$, $f(\varphi)$ est le nombre d'affectations satisfaisant φ.

9-T Proposition

La fonction ♯3SAT est ♯P-complète.

Démonstration Tout d'abord, ♯3SAT $\in$ ♯P car il suffit de considérer le langage

$$A = \{(\varphi, y) \mid \varphi(y) = 1\},$$

où $y = (y_1,\dots,y_n) \in \{0,1\}^n$ encode une affectation des variables de la formule φ : on a alors $A \in \mathsf{P}$ et $\sharp\mathtt{3SAT}(\varphi) = |\{y \in \{0,1\}^n \mid (\varphi, y) \in A\}|$.
Pour la ♯P-difficulté, la proposition 3-Z de NP-complétude de 3SAT et le théorème de Cook-Levin 3-V montrent que toute machine non déterministe $N(x)$ fonctionnant en temps polynomial est simulée par une instance $\varphi_{(N,x)}$ de 3SAT dont le nombre de solutions correspond au nombre de chemins acceptants de $N(x)$. Cette réduction $e_N : x \mapsto \varphi_{(N,x)}$ donne alors une réduction parcimonieuse de toute fonction ♯P à ♯3SAT : en effet, si $f \in \sharp\mathsf{P}$ compte le nombre de chemins acceptants d'une machine N, alors pour tout $x \in \Sigma^\star$, $f(x) = \sharp\mathtt{3SAT}(e_N(x))$. □

Maintenant que l'on a notre premier problème ♯P-complet, pour montrer un autre résultat de ♯P-complétude, il suffit de montrer une réduction à partir de ♯3SAT.

On peut définir de la même façon la fonction de comptage associée au problème CLIQUE : le problème ♯CLIQUE consiste à compter sur l'entrée (G,k) le nombre de cliques de taille k dans le graphe G.

9-U Proposition

Le problème ♯CLIQUE est ♯P-complet.

Démonstration Tout d'abord, ♯CLIQUE $\in$ ♯P car il suffit de considérer le langage

$$A = \{(G,k,y) \mid y \text{ est une clique de taille } k\},$$

où y encode un ensemble de k sommets de G : on a alors $A \in \mathsf{P}$ et $\sharp\mathtt{CLIQUE}(G,k) = |\{y \mid (G,k,y) \in A\}|$.

Pour la ♯P-difficulté nous allons suivre la preuve de NP-complétude de CLIQUE. En composant la réduction de 3SAT à ENSEMBLE INDÉPENDANT (proposition 3-AE) et celle de ENSEMBLE INDÉPENDANT à CLIQUE (exemple 3-I), on obtient une réduction e qui à une instance φ de 3SAT associe un graphe et un entier, $e(\varphi) = (G, k)$: on vérifie aisément que le nombre de solutions de φ est égal au nombre de cliques de taille k de G. On obtient ainsi une réduction parcimonieuse e de ♯3SAT à ♯CLIQUE, c'est-à-dire que $\sharp\text{3SAT}(\varphi) = \sharp\text{CLIQUE}(e(\varphi))$. Cela montre que ♯CLIQUE est ♯P-difficile puisque ♯3SAT l'est. □

On pourrait définir de même les versions de comptage de nombreux problèmes NP-complets naturels et montrer qu'elles deviennent ♯P-complètes (la plupart du temps grâce à des réductions parcimonieuses issues des réductions de NP-complétude). Il est en effet rare que la réduction perde l'information du nombre de solutions.

✎ **9-V** **Exercice**

Soit ♯SOMME PARTIELLE le problème de comptage associé à SOMME PARTIELLE vu à l'exemple 2-AM, c'est-à-dire compter le nombre d'ensembles dont la somme donne la valeur cible.

Montrer que ♯SOMME PARTIELLE est ♯P-complet.

Indication : transformer la réduction de NP*-complétude de la proposition 3-AH en une réduction parcimonieuse.*

9.2.3 Complétude pour les autres classes

L'esprit de la classe PP est de décider si la majorité des candidats sont des solutions. Il n'est donc guère surprenant que le problème Maj3SAT défini ci-après soit PP-complet :

- *entrée* : une formule booléenne sans quantificateur $\varphi(x_1, \ldots, x_n)$ sous forme 3-CNF ;
- *question* : $\varphi(a_1, \ldots, a_n)$ est-elle vraie pour au moins la moitié des affectations des variables par toutes les valeurs possibles $(a_1, \ldots, a_n) \in \{0,1\}^n$?

9-W **Proposition**

Le problème Maj3SAT est PP-complet.

Idée de la démonstration Maj3SAT est dans PP puisqu'il suffit de décider si

$$|\{a \in \{0,1\}^n \mid \varphi(a) = 1\}| \geqslant 2^{n-1}.$$

Pour montrer la complétude, soit $A \in \text{PP}$: par définition, il existe une fonction ♯P $f : \Sigma^\star \to \mathbb{N}$ telle que $x \in A$ ssi $f(x) \geqslant (1/2)|\Sigma|^{p(|x|)}$ pour un certain polynôme $p(n)$. Le résultat de ♯P-complétude de ♯3SAT montre qu'on peut construire une formule

booléenne φ_x à m variables dont le nombre de solutions $\sharp\varphi_x$ est égal à $f(x)$. On cherche donc à modifier φ_x en une formule ψ_x de sorte que $\sharp\varphi_x \geqslant (1/2)|\Sigma|^{p(|x|)}$ ssi au moins la moitié des affectations satisfont ψ_x. On définit alors ψ_x comme suit, où $y = (y_1, \ldots, y_m)$ sont les variables de φ_x vues comme un entier codé en binaire variant entre 0 et $2^m - 1$, et z est une nouvelle variable :

$$\psi_x(y,z) \equiv (z \wedge \varphi_x(y)) \vee (\neg z \wedge y \geqslant \lceil (1/2)|\Sigma|^{p(|x|)} \rceil).$$

La condition $y \geqslant \lceil (1/2)|\Sigma|^{p(|x|)} \rceil$ s'exprime facilement en 3-CNF (exercice 9-X). Le nombre de solutions de ψ_x est $\sharp\varphi_x + (2^m - \lceil (1/2)|\Sigma|^{p(|x|)} \rceil)$ et la formule possède $m+1$ variables. Ainsi, $\psi_x \in \texttt{Maj3SAT}$ ssi

$$\sharp\varphi_x + (2^m - \lceil (1/2)|\Sigma|^{p(|x|)} \rceil) \geqslant 2^m, \quad \text{soit} \quad \sharp\varphi_x \geqslant \lceil (1/2)|\Sigma|^{p(|x|)} \rceil,$$

donc $x \in A$ ssi $\psi_x \in \texttt{Maj3SAT}$, ce qui prouve la réduction de A à $\texttt{Maj3SAT}$. □

✎ **9-X** **Exercice**

> Pour des entiers k et s, on désigne par y un mot sur $\{0,1\}$ de taille $m = \lceil k \log s \rceil$ représentant un entier de $[0, 2^m - 1]$. Montrer qu'on peut exprimer la condition $y \geqslant \lceil (1/2)s^k \rceil$ par une formule en 3-CNF de taille polynomiale en m.

De même, les variantes de comptage modulo k de 3SAT sont $\texttt{Mod}_k\texttt{P}$-complètes : on appelle $\texttt{Mod}_k\texttt{3SAT}$ le problème suivant

- *entrée :* une formule booléenne sans quantificateur $\varphi(x_1, \ldots, x_n)$ en 3-CNF ;
- *question :* le nombre m d'affectations $a_1, \ldots, a_n \in \{0,1\}$ rendant φ vraie vérifie-t-il $m \not\equiv 0 \mod k$?

9-Y **Proposition**

Pour tout entier $k \geqslant 2$, le problème $\texttt{Mod}_k\texttt{3SAT}$ est $\texttt{Mod}_k\texttt{P}$-complet.

✎ **9-Z** **Exercice**

> Montrer la proposition précédente.

On notera en particulier $\oplus\texttt{3SAT}$ le problème $\oplus\texttt{P}$-complet $\texttt{Mod}_2\texttt{3SAT}$.

9.3 Propriétés de clôture

Après des propriétés de clôture de base des classes de comptage, nous allons aussi voir que PP est clos par intersection et union (ce qui est moins facile).

9.3.1 Propriétés de base

9-AA Proposition

La classe ♯P est close par opération polynomiale à coefficients positifs.

Plus précisément, si $f \in \sharp\mathsf{P}$ et $p(n)$ est un polynôme à *coefficients entiers positifs,* alors $g : x \mapsto p(f(x))$ est une fonction de ♯P.

Démonstration Il suffit de montrer que la classe ♯P est close par addition et multiplication puisque $p(f(x))$ n'est autre que la composition d'un nombre constant d'additions et de multiplications en partant de $f(x)$ et de 1 (fonction constante égale à 1, qui est bien sûr dans ♯P).

Si $f(x) = |\{y \mid (x,y) \in A\}|$ et $g(x) = |\{y \mid (x,y) \in B\}|$ sont des fonctions ♯P (et $A, B \in \mathsf{P}$), alors

$$(f+g)(x) = |\{yb \mid ((x,y) \in A \wedge b = 0) \vee ((x,y) \in B \wedge b = 1)\}| \text{ où } b \text{ est un bit,}$$

et

$$(f\,g)(x) = |\{yy' \mid (x,y) \in A \wedge (x,y') \in B\}|.$$

Puisque ces conditions sont calculables en temps polynomial, il résulte que $(f+g) \in \sharp\mathsf{P}$ et $f\,g \in \sharp\mathsf{P}$. □

On en déduit un résultat similaire concernant GapP, et une propriété de clôture de $\mathsf{Mod}_k\mathsf{P}$ lorsque k est premier.

9-AB Corollaire

La classe GapP est close par opération polynomiale à coefficients entiers relatifs.

Plus précisément, si $f \in \mathsf{GapP}$ et $p(n)$ est un polynôme à coefficients dans $\mathbb{Z}$, alors $g : x \mapsto p(f(x))$ est une fonction de GapP.

✎ **9-AC Exercice**

Montrer le corollaire précédent.

9-AD Proposition

Si k est un nombre premier, alors $\mathsf{Mod}_k\mathsf{P}$ est clos par complémentaire.

Démonstration Soit $A \in \mathsf{Mod}_k\mathsf{P}$, c'est-à-dire que

$$x \in A \iff f(x) \not\equiv 0 \mod k,$$

où $f \in \sharp\mathsf{P}$. On a alors

$$x \in {}^cA \iff f(x) \equiv 0 \mod k,$$

et il s'agit donc de montrer qu'il existe $g \in \sharp\mathsf{P}$ telle que $f(x) \equiv 0 \mod k$ ssi $g(x) \not\equiv 0 \mod k$. On pourra ainsi définir ${}^cA$ par $g(x) \not\equiv 0 \mod k$, ce qui est une définition $\mathsf{Mod}_k\mathsf{P}$.
En utilisant une interpolation de Lagrange (extrêmement simple dans notre cas), nous allons définir un polynôme de degré $k-1$ sur $\mathbb{Z}/k\mathbb{Z}$ tel que

$$\begin{cases} p(0) = 1 \\ p(i) = 0 \quad \text{pour } i \neq 0 \end{cases} \quad (i \in \mathbb{Z}/k\mathbb{Z}).$$

Puisque k est premier, $\mathbb{Z}/k\mathbb{Z}$ est un corps donc tout élément non nul est inversible. Il suffit alors de définir

$$p(n) = \prod_{0<j<k} j^{-1}(n-j) :$$

ainsi, comme voulu, $p(0) = \prod_{j\neq 0} j^{-1} \prod_{j\neq 0} j = 1$, et $p(n) = 0$ si $n \neq 0$. En particulier, $p(f(x)) \equiv 1 \mod k$ si $f(x) \equiv 0 \mod k$ et $p(f(x)) \equiv 0 \mod k$ si $f(x) \not\equiv 0 \mod k$.
Si l'on voit $p(n)$ comme un polynôme à coefficients dans $\mathbb{N}$ (et même dans $\{0,\dots,k-1\}$), alors $g = p(f) : x \mapsto p(f(x))$ est une fonction $\sharp\mathsf{P}$ par la proposition 9-AA, qui vérifie

$$x \in {}^cA \iff f(x) \equiv 0 \mod k \iff g(x) \not\equiv 0 \mod k.$$

Donc ${}^cA \in \mathsf{Mod}_k\mathsf{P}$. □

Une méthode similaire à base de polynôme interpolateur permet de résoudre l'exercice suivant.

✎ **9-AE** **Exercice**

> Soit FewP la classe des langages reconnus par une machine non déterministe polynomiale N telle qu'il existe un polynôme p pour lequel, sur toute entrée x, $N(x)$ a au plus $p(|x|)$ chemins acceptants.
>
> Montrer que $\mathsf{FewP} \subseteq \oplus\mathsf{P}$.

Enfin, la clôture de PP par complémentaire est facile à voir.

9-AF **Proposition**

La classe PP est close par complémentaire.

Démonstration Soit $A \in \mathsf{PP}$ défini par

$$x \in A \iff g(x) \geqslant 0,$$

où $g \in \mathsf{GapP}$. Alors $h = -(g+1) \in \mathsf{GapP}$ et

$$x \in {}^{c}A \iff g(x) < 0 \iff h(x) \geqslant 0,$$

ce qui montre que ${}^{c}A \in \mathsf{PP}$ d'après la proposition 9-L. □

Nous allons maintenant voir la clôture de PP par union ou intersection (l'une découle de l'autre par la proposition précédente puisque $A \cap B = {}^{c}({}^{c}A \cup {}^{c}B)$), mais cela requiert un peu plus de travail.

9.3.2 PP est clos par union

Le résultat suivant est dû à Beigel, Reingold et Spielman [BRS95].

9-AG Proposition (Beigel, Reingold et Spielman, 1995)

La classe PP est close par union.

Idée de la démonstration L'appartenance à un problème PP s'exprime comme un test de positivité d'une fonction GapP : pour deux langages $A, B \in \mathsf{PP}$ il existe donc deux fonctions GapP a et b telles que $a(x) \geqslant 0$ ssi $x \in A$ (resp. $b(x) \geqslant 0$ ssi $x \in B$). Tout en restant dans GapP, pour décider l'appartenance à $A \cup B$ il s'agit donc de construire un test de positivité $h(a,b)$ vérifiant $h(a,b) \geqslant 0$ ssi $a \geqslant 0$ ou $b \geqslant 0$.

L'idée est alors de définir une fonction rationnelle $s(x)$ qui approxime le signe, c'est-à-dire qui est proche de 0 si $x < 0$ et qui est proche de 1 si $x > 0$. Alors $h(a,b) = s(a) + s(b) - 1/2$ est le test recherché puisqu'il est positif exactement lorsque a ou b est positif. Pour rester dans GapP, il suffit de le convertir en un polynôme en multipliant par un dénominateur commun.

Démonstration Soit A et B deux langages de PP définis par $x \in A \iff a(x) \geqslant 0$ et $x \in B \iff b(x) \geqslant 0$, où a et b sont deux fonctions GapP (cf. proposition 9-L). Afin d'éviter les cas d'annulation par la suite, on effectue le changement d'échelle $a \mapsto 8a+5$ et $b \mapsto 8b+5$, si bien que l'on suppose que $a(x)$ et $b(x)$ sont impairs et de valeur absolue $\geqslant 3$. Enfin, on note $m(n)$ un polynôme vérifiant $|a(x)|, |b(x)| \leqslant 2^{m(|x|)}$. Pour le raisonnement, on fixe $n = |x|$ et on notera plus simplement m à la place de $m(n)$.

Soit $P(t)$ le polynôme suivant défini sur $\mathbb{Z}$:

$$P(t) = \prod_{i=0}^{m} (2^i + t)^2.$$

Sur $\mathbb{Z}$, ce polynôme prend des valeurs entières positives et ne s'annule pas si t est impair et de valeur absolue $\geqslant 3$, ce que l'on supposera par la suite.

Nous allons montrer que

$$\frac{P(t)}{P(-t)} > 9 \text{ si } t \in [3, 2^m] \text{ est impair.}$$

Démonstration — Il s'agit de montrer que

$$\prod_{i=0}^{m}(2^i + t)^2 > 9\prod_{i=0}^{m}(2^i - t)^2$$

où $3 \leqslant t < 2^m$. Si j désigne l'entier tel que $2^j < t \leqslant 2^{j+1}$, alors $2t \leqslant 4.2^j$ et donc $2^j + t \geqslant 3(t - 2^j)$.
Puisque $(2^j + t)^2 \geqslant 9(2^j - t)^2$ et que les autres termes vérifient $(2^i + t)^2 > (2^i - t)^2$, l'inégalité est montrée. ◇

On en déduit également que $P(t)/P(-t) < 1/9$ si $t \in [-2^m, -3]$ est impair, puisque $P(-t)/P(t) > 9$.
Ces deux propriétés impliquent que

$$[a > 0 \vee b > 0] \iff \frac{P(a)}{P(-a)} + \frac{P(b)}{P(-b)} - 1 \geqslant 0,$$

puisque pour que l'expression soit positive, les deux fractions ne peuvent pas être simultanément $< 1/9$. En d'autres termes,

$$[a > 0 \vee b > 0] \iff P(a)P(-b) + P(-a)P(b) - P(-a)P(-b) \geqslant 0.$$

Si on note $f : \Sigma^\star \to \mathbb{Z}$ la fonction suivante :

$$f(x) = P(a(x))P(-b(x)) + P(-a(x))P(b(x)) - P(-a(x))P(-b(x)),$$

nous avons donc montré que $x \in A \cup B$ ssi $(a(x) \geqslant 0) \vee (b(x) \geqslant 0)$ ssi $f(x) \geqslant 0$.

Il reste à montrer que $f \in \mathsf{GapP}$. Pour calculer $f(x)$, il s'agit d'effectuer un nombre polynomial de produits et sommes à partir de $a(x)$ et $b(x)$.
Si $a(x) = \sharp\mathrm{acc}(N(x)) - \sharp\mathrm{rej}(N(x))$, où N est une machine non déterministe fonctionnant en temps polynomial, alors pour $0 \leqslant i \leqslant m$ on note N_i la machine suivante sur l'entrée x :

- deviner $b \in \{0, 1\}$;
- si $b = 0$, exécuter $N(x)$;
- si $b = 1$, créer 2^i chemins acceptants.

La machine N_i vérifie alors $\sharp\mathrm{acc}(N_i(x)) - \sharp\mathrm{rej}(N_i(x)) = a(x) + 2^i$.
On peut alors définir une machine N'_m dont la différence des nombres de chemins acceptant et rejetant sera $\prod_{i=0}^{m}(a(x) + 2^i)$: $N'_m(x)$ fonctionne comme suit

– Pour i de 0 à m faire
  ◦ exécuter $N_i(x)$;
– accepter ssi le nombre de machines N_i ayant rejeté est pair.

L'idée est que les chemins rejetant contribuent avec un signe moins au résultat, c'est pourquoi on s'occupe seulement de leur parité. On montre par récurrence sur m que

$$\sharp\mathrm{acc}(N'_m(x)) - \sharp\mathrm{rej}(N'_m(x)) = \prod_{i=0}^{m}(a(x)+2^i).$$

Démonstration — Pour $m = 0$ le résultat est clair puisque seule $N_0(x)$ est exécutée et on accepte ssi $N_0(x)$ n'a pas rejeté : on « calcule » donc $a(x)+2^0$.
Pour $m > 1$: par récurrence, N'_{m-1} « calcule » $\prod_{i=0}^{m-1}(a(x)+2^i)$. Or le fonctionnement de N'_m revient à exécuter N_m après N'_{m-1} et à accepter ssi ($N_m(x)$ et $N'_{m-1}(x)$ acceptent) ou ($N_m(x)$ et $N'_{m-1}(x)$ rejettent).
Le nombre de chemins acceptants de $N'_m(x)$ est donc

$$\sharp\mathrm{acc}(N_m(x))\sharp\mathrm{acc}(N'_{m-1}(x)) + \sharp\mathrm{rej}(N_m(x))\sharp\mathrm{rej}(N'_{m-1}(x))$$

et de même le nombre de chemins rejetant de $N'_m(x)$ est

$$\sharp\mathrm{acc}(N_m(x))\sharp\mathrm{rej}(N'_{m-1}(x)) + \sharp\mathrm{rej}(N_m(x))\sharp\mathrm{acc}(N'_{m-1}(x)).$$

La différence des deux vaut donc

$$(\sharp\mathrm{acc}(N_m(x)) - \sharp\mathrm{rej}(N_m(x)))(\sharp\mathrm{acc}(N'_{m-1}(x)) - \sharp\mathrm{rej}(N'_{m-1}(x)))$$

c'est-à-dire

$$(a(x)+2^m)\prod_{i=0}^{m-1}(a(x)+2^i)$$

par hypothèse de récurrence. ◇

On peut bien sûr faire de même pour calculer $\prod_{i=0}^{m}(a(x)-2^i)$, $\prod_{i=0}^{m}(b(x)+2^i)$ et $\prod_{i=0}^{m}(b(x)-2^i)$ dans GapP. Il suffit enfin pour conclure d'utiliser la clôture de GapP par un nombre constant d'opérations d'addition ou multiplication (corollaire 9-AB) pour calculer $f(x)$ dans GapP. □

Comme nous l'avons déjà mentionné, cela implique aussi la clôture par intersection puisque PP est clos par complémentaire.

9.3.3 ♯P est-il clos par soustraction ?

Pour clore cette partie, il est intéressant de mentionner un résultat de Ogiwara et Hemachandra [OH93] concernant la clôture éventuelle de ♯P par soustraction. Plus précisément, on dit que ♯P est clos par soustraction propre si

$$f, g \in \sharp\mathrm{P} \Rightarrow \max(f-g, 0) \in \sharp\mathrm{P}.$$

9-AH **Proposition**

Si $\sharp\mathsf{P}$ est clos par soustraction propre, alors $\mathsf{PH} = \mathsf{NP}$ (la hiérarchie polynomiale s'effondre sur NP).

Ce résultat, montrant que $\sharp\mathsf{P}$ n'est probablement pas clos par soustraction propre, est également montré dans le livre [HO02] écrit par les auteurs de ce résultat.

9.4 Théorème de Toda

En 1989, Toda [Tod89] montrait la surprenante puissance du comptage : le théorème de Toda dit essentiellement que la classe de comptage $\sharp\mathsf{P}$ est au moins aussi puissante que toute la hiérarchie polynomiale.

On ne peut évidemment pas comparer directement d'une part une classe de langages (PH) et d'autre part une classe de fonctions ($\sharp\mathsf{P}$) : pour contourner ce problème, on utilise $\sharp\mathsf{P}$ en oracle de la classe P. Cela signifie que la machine polynomiale peut écrire un mot u sur son ruban d'oracle et obtenir en temps unitaire la valeur $f(u)$, où $f \in \sharp\mathsf{P}$.

9-AI **Théorème (Toda, 1989)**

$$\mathsf{PH} \subseteq \mathsf{P}^{\sharp\mathsf{P}}$$

Avant de voir en détail la démonstration de ce théorème, nous en esquissons les grandes lignes. Pour tout i, il s'agit de résoudre dans $\mathsf{P}^{\sharp\mathsf{P}}$ le problème $\mathtt{SAT}_i$ qui est Σ_i^{P}-complet (cf. proposition 8-M), c'est-à-dire de décider si une formule φ ayant une alternance de i quantificateurs est vraie.

Dans un premier temps, on raisonne sur une fonction booléenne $\alpha : \{0,1\}^n \to \{0,1\}$ quelconque. En mettant des poids aléatoires sur les variables $x_1, \ldots, x_n$, on montre qu'avec probabilité $1/n^{O(1)}$ il existe une unique solution de poids minimal x telle que $\alpha(x) = 1$ (théorème 9-AK). On utilise pour cela le lemme 9-AJ de Valiant et Vazirani (qui a de nombreuses autres applications).

En exigeant non plus que la solution soit unique, mais que le nombre de solutions de poids minimum soit impair, on peut amplifier la probabilité $1/n^{O(1)}$ arbitrairement proche de 1 : essentiellement, il suffit de répéter suffisamment de fois le résultat précédent. Une telle amplification nous permet alors d'itérer le résultat : si φ s'exprime comme

$$\varphi = \exists x\ \varphi'(x),$$

par récurrence on peut construire à partir de φ' (où seuls $i-1$ quantificateurs alternent) une formule ψ' sans quantificateur qui, avec grande probabilité, aura un nombre impair

de solutions ssi φ' est vraie. On applique alors le résultat précédent pour obtenir une formule ψ sans quantificateur qui, avec grande probabilité, aura un nombre impair de solutions ssi φ est vraie. En d'autres termes, on a réduit de manière probabiliste PH à $\oplus$P (lemme 9-AM).

Pour conclure, il nous faut une réduction déterministe mais plutôt que $\oplus$P nous pouvons utiliser $\sharp$P. Une transformation, grâce au lemme 9-AN, de la formule obtenue permet de se ramener à des valeurs -1 ou 0 modulo 2^k pour un grand k (et non plus modulo 2). Pour k suffisamment grand, une somme sur l'ensemble des choix aléatoires donne alors des résultats disjoints modulo 2^k selon que φ est vraie ou non. Cette somme est évaluée grâce à un appel à l'oracle $\sharp$P.

Nous commençons donc par le lemme de Valiant et Vazirani issu de l'article [VV86].

9-AJ Lemme (Valiant et Vazirani, 1986)

Soit $E = \{x_1, \ldots, x_n\}$ un ensemble de n éléments et une famille $S_1, \ldots, S_N$ de sous-ensembles distincts de E. À chaque élément x_i on associe un poids $w_i \in \{1, 2, \ldots, 2n\}$ au hasard, indépendamment selon la distribution uniforme. Pour $S \subseteq E$, on appelle *poids* de S la somme des poids de ses éléments, c'est-à-dire

$$w(S) = \sum_{x_i \in S} w_i.$$

Alors la probabilité que, parmi la famille $S_1, \ldots, S_N$, il existe un unique ensemble S_i de poids minimal, est au moins $1/2$.

Démonstration Une fois que les poids sont fixés, on dit que $x \in E$ est *mauvais* s'il appartient à un ensemble S_i de poids minimal mais pas à tous. S'il existe un unique ensemble S_i de poids minimal, alors il est clair qu'aucun élément $x \in E$ n'est mauvais. Réciproquement, s'il existe au moins deux ensembles S_i, S_j de poids minimal, tout élément de $S_i \setminus S_j$ ou $S_j \setminus S_i$ est mauvais. On a donc l'équivalence : il existe un unique ensemble S_i de poids minimal ssi aucun élément $x \in E$ n'est mauvais.

Il suffit donc de montrer que la probabilité d'existence d'un élément mauvais est majorée par $1/2$. Pour cela, nous allons montrer que pour chaque x_i, $\Pr(x_i \text{ est mauvais}) \leqslant 1/(2n)$, ce qui permet de conclure puisque par l'inégalité de Boole

$$\Pr(\exists i, x_i \text{ est mauvais}) \leqslant \sum_{i=1}^{n} \Pr(x_i \text{ est mauvais}) \leqslant n \frac{1}{2n} = \frac{1}{2}.$$

Considérons donc un élément x_i fixé et on suppose que le poids de tous les autres éléments a été défini. On cherche à savoir quelles valeurs peut prendre le poids de x_i pour rendre x_i mauvais. On considère alors deux parties de la famille (S_i) :

– on note $M_1, \ldots, M_k$ les ensembles de la famille (S_i) de poids minimal *parmi ceux qui ne contiennent pas* x_i et on note leur poids m ;

– *parmi ceux qui contiennent* x_i, on note $M'_1, \dots, M'_l$ les ensembles de poids minimal *sans compter le poids de* x_i (non encore fixé). On note m' ce poids « partiel ».

Si $m \leqslant m'$ alors quel que soit le poids de x_i, les ensembles M_j seront de poids inférieur aux ensembles M'_j, donc x_i ne fait pas partie d'un ensemble de poids minimal et ne sera jamais mauvais.
Si $m > m'$, alors le seul choix de poids permettant de rendre x_i mauvais est $w_i = m - m'$: en effet, si $w_i < m - m'$ alors les ensembles M'_j sont les ensembles minimaux mais contiennent tous x_i ; si $w_i > m - m'$ alors les ensembles M_j sont les ensembles minimaux mais aucun ne contient x_i. Ainsi, la seule manière de rendre x_i mauvais est de choisir $w_i = m - m'$, ce qui arrive avec probabilité $1/(2n)$ puisque les poids prennent leurs valeurs dans $\{1, 2, \dots, 2n\}$.
Au total, x_i est mauvais avec probabilité $\leqslant 1/(2n)$, ce qui conclut la démonstration. □

On peut se servir de ce lemme afin de transformer avec une certaine probabilité une formule satisfaisable en une formule ayant exactement une solution. On obtient le résultat suivant issu du même article [VV86].

9-AK **Théorème (Valiant et Vazirani, 1986)**

On désigne par $W = (w, w^{(1)}, \dots, w^{(n)})$ un $(n+1)$-uplet d'entiers tels que $w^{(i)} \in \{1, \dots, 2n\}$ et $w \in \{1, \dots, 2n^2\}$.

Il existe un algorithme polynomial qui, sur l'entrée $(1^n, W)$, produit une formule booléenne $\gamma_W(x_1, \dots, x_n)$ vérifiant la propriété suivante : pour toute fonction booléenne $\alpha : \{0,1\}^n \to \{0,1\}$,

$$\exists x, \alpha(x) = 1 \implies \Pr_W(\text{il existe un unique } x \text{ tel que } (\alpha(x) = 1) \wedge \gamma_W(x)) \geqslant \frac{1}{4n^2}.$$

Bien sûr, si $\alpha(x) = 0$ pour tout x alors $(\alpha(x) = 1) \wedge \gamma_W$ n'a pas de solution non plus, quel que soit W. En d'autres termes, on transforme de manière uniforme une fonction booléenne α en une fonction équivalente pour la satisfaisabilité mais qui, avec une probabilité suffisante, a au plus une solution.

Idée de la démonstration On associe le poids $w^{(i)}$ à la variable x_i. La formule γ_W exprime le fait que la somme des poids des variables mises à vrai vaut w. Par le lemme de Valiant et Vazirani, avec probabilité $\geqslant 1/2$ sur $(w^{(1)}, \dots, w^{(n)})$ le poids minimal est unique. Il suffit alors de choisir w égal à ce poids minimal, ce qui se produit avec probabilité $1/(2n^2)$.

Démonstration Si l'on considère que les variables x_i valent 0 ou 1, alors la formule γ_W exprime le fait que

$$\sum_{i=1}^{n} x_i w^{(i)} = w.$$

Nous allons simplement esquisser la preuve du fait que l'on peut exprimer cette somme par une formule booléenne γ_W sans quantificateur et de taille polynomiale (en d'autres termes, que l'addition itérée est dans NC^1).

Démonstration — On dispose des bits des nombres $w^{(i)}$, il est donc facile de calculer les bits de $y^{(i)} = x_i w^{(i)}$ (entier qui vaut soit 0 soit $w^{(i)}$). Pour calculer la somme des $y^{(i)}$, on procède par récurrence sur n. Si y est un entier en binaire, on notera y_j son j-ème bit.

Pour $n = 2$, il s'agit simplement d'exprimer que le j-ème bit de w est celui que l'on souhaite. Pour cela, il y a une retenue lorsque $y_k^{(1)} = y_k^{(2)} = 1$ pour un certain $k < j$ et que pour tout $l \in \{k, \dots, j\}$, $y_l^{(1)} \vee y_l^{(2)} = 1$. On obtient donc la retenue au rang j par la formule suivante :

$$\bigvee_{k<j} (y_k^{(1)} \wedge y_k^{(2)} \wedge \bigwedge_{k<l<j} (y_l^{(1)} \vee y_l^{(2)})).$$

L'expression du j-ème bit de la somme suit alors aisément et il suffit de tester si tous les bits de w sont corrects.

Pour $n > 2$: on groupe d'abord les entiers $y^{(i)}$ par trois. L'astuce est de remplacer l'addition des trois entiers de m bits de chaque groupe par l'addition de deux entiers de $m+1$ bits, et ainsi se ramener à l'addition de $2n/3$ entiers.

Pour ajouter trois entiers y, z, t de m bits, on définit deux entiers u, v de $m+1$ bits par $y_j + z_j + t_j = v_{j+1} u_j$ (le nombre dont l'écriture binaire est la concaténation des deux bits v_{j+1} et u_j) : la somme des trois bits $y_j + z_j + t_j$ prend une valeur parmi $\{0, \dots, 3\}$ et se code donc sur deux bits. Le bit de poids faible est stocké par u à la j-ème position et le bit de poids fort par v à la position $(j+1)$. En complétant la définition de u et v par $u_{m+1} = 0$ et $v_1 = 0$, on a bien défini deux entiers u et v tels que $y + z + t = u + v$. Chaque bit de u et v s'exprime comme une formule de taille constante en fonction des bits de y, z et t.

On s'est donc ramené à l'addition de $2n/3$ nouveaux entiers, qui se fait par induction en remplaçant u et v par leur expression ci-dessus. Puisque l'expression de chaque bit de u et v est de taille constante, on ajoute un nombre linéaire de portes à la formule qui permet d'additionner ces $2n/3$ termes. Au final, puisque l'induction possède seulement $O(\log n)$ étapes, la formule totale est de taille polynomiale. ◇

On remarque que la description de γ_W ci-dessus permet de déduire un algorithme écrivant cette formule en temps polynomial à partir de W.

Revenons à notre fonction $\alpha(x_1, \dots, x_n)$. Si α est satisfaisable (c'est-à-dire $\exists x, \alpha(x) = 1$), soit $s_1, \dots, s_N$ la famille des solutions. Chaque s_i attribue des valeurs vrai ou faux aux variables : pour la suite, on interprète s_i comme l'ensemble des variables mises à vrai par la solution s_i. Par le lemme 9-AJ, en choisissant un poids aléatoire parmi $\{1, \dots, 2n\}$ pour chaque variable x_i, avec probabilité $\geqslant 1/2$ il existe une unique solution de poids minimum $w_{\min}$. On a alors une probabilité $1/(2n^2)$ de choisir $w = w_{\min}$, et dans ce cas

$\alpha \wedge \gamma_W$ a une unique solution. Au total, puisque ce sont des événements indépendants, la probabilité que $\alpha \wedge \gamma_W$ ait une unique solution est $\geqslant 1/(4n^2)$. □

On peut amplifier la probabilité dans l'énoncé précédent en autorisant non plus une unique solution, mais un nombre impair de solutions. On peut même généraliser à des formules ayant un nombre borné d'alternances de quantificateurs. Pour cela on a besoin d'une notation pratique sur le nombre de solutions d'une formule ψ.

9-AL Définition

- Pour toute formule $\psi(x_1,\dots,x_r)$ à r variables libres, on notera $\sharp\psi$ le nombre de ses solutions ($0 \leqslant \sharp\psi \leqslant 2^r$). S'il y a un doute sur les variables concernées, on les précisera en indice du $\sharp$: par exemple si a est fixé, alors $\sharp_y(\varphi(a,y))$ est le nombre d'affectations des variables y telles que $\varphi(a,y)$ est vraie.
- Pour toute formule $\psi(x_1,\dots,x_r)$ sur r variables, on notera $(1+\psi)$ la formule suivante sur $r+1$ variables :

$$(1+\psi)(x_1,\dots,x_r,y) = \big(y \wedge \psi(x_1,\dots,x_r)\big) \vee \big(\neg y \wedge \bigwedge_{i=1}^{r} x_i\big),$$

  où y est une nouvelle variable. Le nombre de solutions de $(1+\psi)$ est alors celui de ψ plus un, c'est-à-dire :

$$\sharp(1+\psi) = 1 + \sharp\psi.$$

- Plus généralement, si $\psi(x_1,\dots,x_r)$ et $\psi'(y_1,\dots,y_s)$ sont des formules sur r et s variables respectivement avec $r \geqslant s$, on note $(\psi+\psi')$ la formule suivante sur $r+1$ variables :

$$(\psi+\psi')(z,x_1,\dots,x_r) = \big(z \wedge \psi(x_1,\dots,x_r)\big) \vee \big(\neg z \wedge \psi'(x_1,\dots,x_s) \wedge \bigwedge_{i=s+1}^{r} x_i\big).$$

  Le nombre de solutions de $(\psi+\psi')$ est alors la somme de celui de ψ et de ψ', c'est-à-dire :

$$\sharp(\psi+\psi') = \sharp\psi + \sharp\psi'.$$

- De même, si $\psi(x_1,\dots,x_r)$ et $\psi'(y_1,\dots,y_s)$ sont des formules sur r et s variables respectivement, on note $(\psi\times\psi')$ la formule suivante sur $r+s$ variables :

$$(\psi\times\psi')(x_1,\dots,x_r,y_1,\dots,y_s) = \psi(x_1,\dots,x_r) \wedge \psi'(y_1,\dots,y_s).$$

  Le nombre de solutions de $(\psi\times\psi')$ est alors le produit de celui de ψ et de ψ', c'est-à-dire :

$$\sharp(\psi\times\psi') = \sharp\psi \times \sharp\psi'.$$

- Enfin, pour une formule ψ et un entier k, $k\psi$ désignera $\psi+(\psi+(\cdots+\psi))$ (k fois) et ψ^k désignera $\psi\times(\psi\times(\cdots\times\psi))$ (k fois).

9-AM Lemme

On suppose que $\varphi(x)$ désigne une formule de variables libres x et possédant q alternances de quantificateurs (où q est une constante).

Il existe un algorithme polynomial qui, sur l'entrée $(\varphi, 1^m, u)$ où u est un mot de taille polynomiale en $|\varphi|$ et m, produit une formule booléenne $\psi_u(x,y)$ sans quantificateur vérifiant la propriété suivante pour toute affectation a des variables x :

$$\begin{cases} \varphi(a) \text{ vraie} & \Longrightarrow \Pr_u(\psi_u(a,z) \text{ a un nombre impair de solutions } z) \geqslant 1-2^{-m}, \\ \varphi(a) \text{ fausse} & \Longrightarrow \Pr_u(\psi_u(a,z) \text{ a un nombre pair de solutions } z) \geqslant 1-2^{-m}. \end{cases}$$

En d'autres termes, si l'on voit $\varphi(a)$ comme une valeur 0 ou 1,

$$\Pr_u\left(\sharp_z\left(\psi_u(a,z)\right) \equiv \varphi(a) \mod 2\right) \geqslant 1-2^{-m}.$$

Idée de la démonstration En raisonnant par récurrence sur le nombre de quantificateurs, l'idée est d'amplifier la probabilité $1/(4n^2)$ du théorème précédent 9-AK en requérant seulement que le nombre de solutions soit impair (et non plus égal à 1). Pour cela, comme d'habitude on répète un nombre polynomial de fois le corollaire précédent et on encode le tout dans une formule booléenne.

Un peu plus précisément, l'étape d'induction revient grosso-modo à traiter le cas de base $\varphi(a) = \exists y \varphi'(a,y)$. Or par le théorème 9-AK, il existe une formule booléenne γ simple à construire telle que :

$$\begin{cases} \sharp_y(\varphi'(a,y) \wedge \gamma(y)) \text{ impair} \implies \exists y \varphi'(a,y), \\ \exists y \varphi'(a,y) \implies \sharp_y(\varphi'(a,y) \wedge \gamma(y)) \text{ impair, avec probabilité } 1/(4n^2). \end{cases}$$

Cette probabilité peut être amplifiée par répétition : en considérant k formules booléennes $\gamma_1, \ldots, \gamma_k$ on construit la formule (selon les notations de la définition 9-AL)

$$\psi(a,y) = 1 + \prod_{i=1}^{k} (1 + (\varphi'(a,y) \wedge \gamma_i(y))) :$$

on a alors $\sharp_y \psi$ impair ssi l'un des $\sharp_y(\varphi' \wedge \gamma_i)$ est impair. Donc, avec probabilité au moins $1-(1-1/(4n^2))^k$, on a $\sharp_y(\psi(a,y))$ impair ssi $\exists y \varphi'(a,y)$ ssi $\varphi(a) = 1$. Il suffit de choisir k polynomial suffisamment grand pour conclure.

Les détails ci-dessous sont malheureusement obscurcis par les notations dues à la gestion de la récurrence.

Démonstration On raisonne par récurrence sur q. Pour $q = 0$, la formule n'a pas de quantificateur donc il suffit de prendre $\psi_u = \varphi$ qui vérifie que $\psi_u(a)$ a 1 solution si $\varphi(a)$ est vraie, et 0 sinon.

Soit $q > 0$. Nous supposons tout d'abord que le premier quantificateur est $\exists$, c'est-à-dire $\varphi(x) = \exists y \varphi'(x,y)$ où y est un uple de variables. La formule $\varphi'(x,y)$ a $(q-1)$

alternances de quantificateurs, donc par hypothèse de récurrence, on peut construire une formule sans quantificateur $\psi'_{u'}(x,y,z)$, où u' est un mot de taille polynomiale en $|\varphi'|$ et m, vérifiant pour toute affectation (a,b) de (x,y) :

$$\Pr_{u'}\left(\sharp_z\left(\psi'_{u'}(a,b,z)\right) \equiv \varphi'(a,b) \mod 2\right) \geqslant 1-2^{-|y|-m-1}.$$

Puisqu'il y a $2^{|y|}$ valeurs possibles pour b, par l'inégalité de Boole on en déduit que pour tout a :

$$\Pr_{u'}\left(\forall b, \sharp_z\left(\psi'_{u'}(a,b,z)\right) \equiv \varphi'(a,b) \mod 2\right) \geqslant 1-2^{-m-1}.$$

Ainsi, avec probabilité $\geqslant 1-2^{-m-1}$ sur le choix de u', on a

$$\varphi(a) \text{ vrai} \iff \exists y\ \varphi'(a,y) \iff \exists y\ \sharp_z\left(\psi'_{u'}(a,y,z)\right) \text{ impair.}$$

Afin d'éliminer le quantificateur $\exists y$, on va bien sûr utiliser le théorème 9-AK. On note $\alpha_{u'}(x,y)$ la fonction booléenne suivante :

$$\alpha_{u'}(x,y) = \begin{cases} 1 & \text{si } \sharp_z\left(\psi'_{u'}(x,y,z)\right) \text{ impair} \\ 0 & \text{sinon.} \end{cases}$$

Par définition, on a donc pour tout (a,b), $\sharp_z\left(\psi'_{u'}(a,b,z)\right) \equiv \alpha_{u'}(a,b) \mod 2$ et l'équivalence précédente avec probabilité $\geqslant 1-2^{-m-1}$ sur le choix de u', se réécrit

$$\varphi(a) \text{ vrai} \iff \exists y\ \alpha_{u'}(a,y). \tag{9.1}$$

On peut maintenant débuter l'amplification de la probabilité de succès du théorème 9-AK en relâchant la contrainte d'unicité qui devient une contrainte de parité. Cela va se faire grâce à plusieurs applications indépendantes du théorème. Si n désigne le nombre de variables de φ, on choisit un entier k polynomial en n et m tel que $(1-1/(4n^2))^k \leqslant 2^{-m-1}$. On considère k uples $W^{(i)}$ aléatoires, chacun de taille $n+1$: le théorème 9-AK nous donne ainsi k formules $\gamma_{W^{(i)}}(x,y)$ qui vérifient chacune pour tout a :

$$\begin{cases} \exists y\ \alpha_{u'}(a,y) & \Longrightarrow \Pr_{W^{(i)}}\left(\sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)=1\right) \geqslant 1/(4n^2) \\ \forall y\ \neg\alpha_{u'}(a,y) & \Longrightarrow \forall W^{(i)},\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)=0. \end{cases}$$

Par l'équivalence 9.1, avec probabilité $\geqslant 1-2^{-m-1}$ sur le choix de u', on a donc

$$\begin{cases} \varphi(a) \text{ vrai} & \Longrightarrow \Pr_{W^{(i)}}\left(\sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)=1\right) \geqslant 1/(4n^2) \\ \varphi(a) \text{ faux} & \Longrightarrow \forall W^{(i)},\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)=0. \end{cases} \tag{9.2}$$

Nous souhaitons maintenant amplifier la probabilité $1/(4n^2)$ et remplacer $\alpha_{u'}$ par $\psi'_{u'}$. Remarquons que

$$\sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right) \equiv \sharp_{y,z}\left(\psi'_{u'}(a,y,z)\wedge\gamma_{W^{(i)}}(a,y)\right) \mod 2 : \tag{9.3}$$

en effet,

$$\sharp_{y,z}\left(\psi'_{u'}(a,y,z)\wedge\gamma_{W^{(i)}}(a,y)\right) = \sum_{y \mid \gamma_{W^{(i)}}(a,y)} \sharp_z\left(\psi'_{u'}(a,y,z)\right)$$
$$\equiv \sum_{y \mid \gamma_{W^{(i)}}(a,y)} \alpha_{u'}(a,y) \equiv \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right) \mod 2.$$

Si u désigne le uple $(u', W^{(1)},\dots,W^{(k)})$, on définit alors la formule $\psi_u(x,y,z)$ suivante grâce aux notations de la définition 9-AL :

$$\psi_u(x,y,z) = 1+\left(1+\psi'_{u'}(x,y,z)\wedge\gamma_{W^{(1)}}(x,y)\right)\times\ldots\times\left(1+\psi'_{u'}(x,y,z)\wedge\gamma_{W^{(k)}}(x,y)\right).$$

Puisque k est polynomial, cette formule a un nombre polynomial de variables, est de taille polynomiale et son code est calculable en temps polynomial. Par l'équation 9.3, on a pour tout a :

$$\sharp_{y,z}\left(\psi_u(a,y,z)\right) \equiv 1+\prod_i\left(1+\sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)\right) \mod 2$$

et donc

$$\sharp_{y,z}\left(\psi_u(a,y,z)\right) \text{ est impair} \iff \exists i\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right) \text{ est impair.}$$

Si $\varphi(a)$ est vrai, alors l'équation 9.2 et l'indépendance des $W^{(i)}$ nous apprennent qu'avec probabilité $\geqslant 1-2^{-m-1}$ sur le choix de u',

$$\Pr_{W^{(1)},\ldots,W^{(k)}}\left(\exists i\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)\ \text{impair}\right) =$$
$$1-\Pr_{W^{(1)},\ldots,W^{(k)}}\left(\forall i,\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)\ \text{pair}\right)$$
$$\geqslant 1-(1/(4n^2))^k \geqslant 1-2^{-m-1}.$$

En revanche si $\varphi(a)$ est faux, alors avec probabilité $\geqslant 1-2^{-m-1}$ sur le choix de u',

$$\Pr_{W^{(1)},\ldots,W^{(k)}}\left(\exists i\ \sharp_y\left(\alpha_{u'}(a,y)\wedge\gamma_{W^{(i)}}(a,y)\right)\ \text{impair}\right) = 0.$$

Au total, en prenant en compte les probabilités $1-2^{-m-1}$ sur le choix de u' et sur le choix des $W^{(i)}$ (on a donc une probabilité $\leqslant 2^{-m}$ de mal choisir l'un ou l'autre), on a donc montré qu'avec probabilité $\geqslant 1-2^{-m}$ sur le choix de u,

$$\varphi(a) \text{ vrai} \iff \sharp_{y,z}\left(\psi_u(a,y,z)\right)\ \text{impair},$$

ce qui conclut ce cas.

Il reste le cas où φ commence par le quantificateur $\forall$. Dans ce cas, on raisonne sur la formule $\neg\varphi = \exists x \neg\varphi'(x)$: par ce que l'on vient de montrer, cela nous donne une formule ψ_u. Il s'agit alors simplement d'inverser la parité du nombre de solutions, ce qui se fait en remplaçant ψ_u par $(1+\psi_u)$. □

On a ainsi réduit de manière probabiliste tout la hiérarchie polynomiale à $\oplus\mathsf{P}$. Pour la suite de la preuve, il nous faut une réduction déterministe, mais on a le droit à $\sharp\mathsf{P}$ (et non seulement $\oplus\mathsf{P}$). On va donc pouvoir raisonner avec un modulo plus élevé.

9-AN Lemme

Dans ce qui suit, l'entier k est une puissance de 2.

Il existe un algorithme polynomial A qui prend en entrée une formule booléenne $\varphi(x)$ sans quantificateur et un entier k en unaire, et qui renvoie une formule

$$\psi(y) = A(\varphi, 1^k)$$

vérifiant

$$\begin{cases} \sharp\varphi \equiv 0 \mod 2 \implies \sharp\psi \equiv 0 \mod 2^k \\ \sharp\varphi \equiv 1 \mod 2 \implies \sharp\psi \equiv -1 \mod 2^k. \end{cases}$$

Démonstration Selon les notations de la définition 9-AL, on définit

$$\begin{cases} \psi_0 = \varphi \\ \psi_{i+1} = 4(\psi_i)^3 + 3(\psi_i)^4 \quad \text{pour } i \geqslant 0, \end{cases}$$

et on pose

$$\psi = \psi_{\log k}.$$

Pour toutes formules ξ et ξ', la taille de $(\xi + \xi')$ et de $(\xi \times \xi')$ est une constante fois celle de ξ et ξ'. On en déduit que la taille de ψ_i est $2^{O(i)}|\varphi|$: pour $i = \log k$, la taille de ψ est donc polynomiale en la taille de l'entrée. De même, le calcul de ψ se fait en temps polynomial.

Par ailleurs, on vérifie aisément que pour tout entier x,

$$\begin{cases} x \equiv 0 \mod 2^i & \implies 4x^3 + 3x^4 \equiv 0 \mod 2^{i+1}; \\ x \equiv -1 \mod 2^i & \implies 4x^3 + 3x^4 \equiv -1 \mod 2^{i+1}. \end{cases}$$

On en déduit le résultat pour ψ. □

On peut enfin montrer le théorème de Toda.

Idée de la démonstration du théorème 9-AI On cherche à décider dans $\mathsf{P}^{\sharp\mathsf{P}}$ si une formule φ ayant une alternance de i quantificateurs est vraie. Le lemme précédent combiné au lemme 9-AM permet de calculer une formule ψ_u dont le nombre de solutions est, avec grande probabilité sur u, soit -1 soit 0 modulo 2^k selon que φ est vraie ou non.

Dans $\sharp\mathsf{P}$ on peut faire la somme sur u de ces nombres de solutions et on obtient alors des cas disjoints selon que φ est vraie ou non, ce qui permet de décider dans $\mathsf{P}^{\sharp\mathsf{P}}$ la véracité de φ.

Démonstration du théorème 9-AI Puisque $\mathsf{P}^{\sharp\mathsf{P}}$ est clos par réduction many-one, il suffit de montrer que le problème Σ_i^{P}-complet $\mathtt{SAT}_i$ (cf. proposition 8-M) est dans $\mathsf{P}^{\sharp\mathsf{P}}$ pour tout i.

Soit i fixé et φ une instance de $\mathtt{SAT}_i$, c'est-à-dire une formule booléenne de la forme

$$\exists x_1\, \forall x_2 \ldots \mathsf{Q} x_i\ \psi(x_1, \ldots, x_i).$$

Grâce au lemme 9-AM avec $m = 2$, on peut calculer en temps polynomial une formule sans quantificateur $\psi_u(z)$ telle que

$$\begin{cases} \varphi \text{ vraie} & \Longrightarrow \ \Pr_u(\psi_u(z) \text{ a un nombre impair de solutions } z) \geqslant 3/4, \\ \varphi \text{ fausse} & \Longrightarrow \ \Pr_u(\psi_u(z) \text{ a un nombre pair de solutions } z) \geqslant 3/4. \end{cases}$$

Soit k la puissance de 2 telle que $|u| + 1 \leqslant k \leqslant 2|u| + 1$. Le lemme 9-AN produit en temps polynomial une formule ψ'_u vérifiant :

- pour tout u, $\sharp\psi'_u$ est congru à 0 ou -1 modulo 2^k ;
- si φ est vraie alors $\Pr_u(\sharp\psi'_u \equiv -1 \mod 2^k) \geqslant 3/4$;
- si φ est fausse alors $\Pr_u(\sharp\psi'_u \equiv 0 \mod 2^k) \geqslant 3/4$.

En sommant sur tous les mots u possibles, on obtient donc :

- si φ est vraie alors $(\sum_u \sharp\psi'_u)$ est congru modulo 2^k à une valeur de l'intervalle $[-2^{|u|}, -(3/4)2^{|u|}]$;
- si φ est fausse alors $(\sum_u \sharp\psi'_u)$ est congru modulo 2^k à une valeur de l'intervalle $[-(1/4)2^{|u|}, 0]$.

Ces deux intervalles sont disjoints modulo 2^k puisque $k \geqslant |u| + 1$. Or

$$\sum_u \sharp\psi'_u = |\{(u, z) \mid \psi'_u(z) \text{ vrai}\}|$$

et tester sur l'entrée (u, z) si $\psi'_u(z)$ est vrai se fait en temps polynomial. En d'autres termes,

$$(f : \varphi \mapsto \sum_u \sharp\psi'_u) \in \sharp\mathsf{P}.$$

Pour décider si $\varphi \in \mathtt{SAT}_i$, il suffit alors de demander à l'oracle $\sharp\mathsf{P}$ la valeur de $f(\varphi)$ et de tester si elle appartient à l'intervalle $[-2^{|u|}, -(3/4)2^{|u|}]$ ou $[-(1/4)2^{|u|}, 0]$ modulo 2^k. □

9-AO Remarques

- Le résultat montré est un peu plus fort qu'annoncé car la construction ci-dessus ne fait qu'un seul appel à son oracle. On écrit donc en général le théorème de Toda sous la forme $\mathsf{PH} \subseteq \mathsf{P}^{\sharp\mathsf{P}[1]}$ pour signaler qu'un seul appel est effectué.
- L'exercice 9-N montrant que $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\sharp\mathsf{P}}$ implique que $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$ également.

9.5 Permanent

Chacun se rappelle de la fameuse formule pour le déterminant d'une matrice carrée $M = (x_{i,j})_{1\leqslant i,j\leqslant n}$ de taille n :

$$\det_n(x_{1,1},\dots,x_{1,n},x_{2,1},\dots,x_{n,n}) = \sum_{\sigma\in\mathscr{S}_n}(-1)^{\epsilon(\sigma)}\prod_{i=1}^{n}x_{i,\sigma(i)},$$

où la somme est prise sur l'ensemble $\mathscr{S}_n$ des permutations σ de $\{1,\dots,n\}$ (et où $\epsilon(\sigma)$ désigne la signature de σ). Le déterminant d'une matrice $n\times n$ est ainsi un polynôme à n^2 variables, qui correspondent aux coefficients de la matrice.

Lorsqu'on souhaite calculer le déterminant par une machine de Turing, il faut bien sûr coder la matrice : on supposera ses coefficients entiers et on les codera en binaire. En utilisant une élimination de Gauss, il est bien connu qu'on peut calculer le déterminant en temps polynomial.

Il existe un autre polynôme qui ressemble étrangement au déterminant, il s'agit du permanent. Le permanent d'une matrice carrée $M = (x_{i,j})_{1\leqslant i,j\leqslant n}$ de taille n est en effet défini ainsi :

$$\mathrm{per}_n(x_{1,1},\dots,x_{1,n},x_{2,1},\dots,x_{n,n}) = \sum_{\sigma\in\mathscr{S}_n}\prod_{i=1}^{n}x_{i,\sigma(i)}.$$

La seule différence avec le déterminant est l'absence du coefficient $(-1)^{\epsilon(\sigma)}$, c'est-à-dire que la signature de la permutation n'intervient pas. C'est ce « détail » qui donnait au déterminant son interprétation géométrique permettant de le calculer efficacement. Et son absence rend le calcul du permanent singulièrement plus compliqué. C'est ce que nous allons étudier dans cette partie.

Mais dans un premier temps, nous allons voir pourquoi le permanent est un polynôme intéressant et pourquoi il joue un rôle important en complexité.

9.5.1 Intérêts du permanent

Outre sa définition formelle pour une matrice quelconque, on peut également considérer le permanent de la matrice d'adjacence d'un graphe biparti. Rappelons qu'un graphe biparti est un graphe non orienté $G = (V,E)$ tel qu'on peut partitionner l'ensemble V des sommets en deux sous-ensembles V_1 et V_2 de sorte que toute arête de E soit entre un sommet de V_1 et un sommet de V_2. La matrice d'adjacence de G est alors la matrice de taille $|V_1|\times|V_2|$ dont les lignes représentent les sommets de V_1 et les colonnes ceux de V_2, et le coefficient à l'intersection de la ligne $u\in V_1$ et de la colonne $v\in V_2$ est

$$\alpha_{u,v} = \begin{cases} 1 & \text{si } (u,v)\in E \\ 0 & \text{sinon.} \end{cases}$$

Un exemple de graphe biparti et sa matrice d'adjacence est donné à la figure 9.1. Le cas qui nous intéressera par la suite est celui où $|V_1| = |V_2|$.

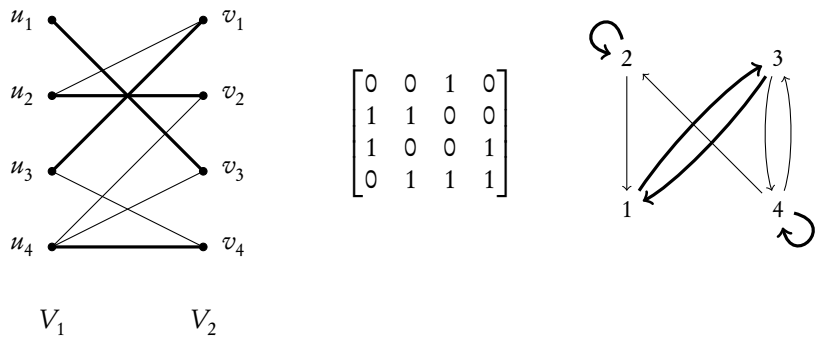


FIGURE 9.1 – Un graphe biparti (et un couplage parfait en gras), sa matrice d'adjacence et la couverture par cycles associée (en gras).

Dans un graphe biparti, généralement utilisé pour représenter les appariements possibles entre deux ensembles de n objets, une notion fondamentale est celle de *couplage parfait*, c'est-à-dire une manière de former des paires d'objets compatibles sans oublier d'objets. Plus formellement, si G est un graphe biparti tel que $|V_1| = |V_2| = n$ et $E \subseteq V_1 \times V_2$ alors un couplage parfait est simplement un sous-ensemble $F \subseteq E$ de n arêtes deux à deux non adjacentes (c'est-à-dire n paires disjointes de sommets reliés par une arête). On peut aussi voir ce couplage parfait comme une permutation σ de $\{1,\dots,n\}$, envoyant les sommets de V_1 sur ceux de V_2 le long des arêtes de G.

Il s'avère qu'il existe un algorithme polynomial pour décider l'existence d'un couplage parfait dans un graphe G [1]. Si en revanche on veut *compter* le nombre de couplages parfaits, cela revient à calculer le permanent de la matrice d'adjacence. En effet, le terme $\prod_i x_{i,\sigma(i)}$ vaut 1 si la permutation σ correspond à un couplage parfait, et 0 sinon. Donc la valeur du permanent

$$\sum_{\sigma \in \mathcal{S}_n} \prod_{i=1}^{n} x_{i,\sigma(i)}$$

est bien égale au nombre de couplages parfaits de G. Puisque toute matrice à coefficients $\{0,1\}$ est la matrice d'adjacence d'un graphe biparti, caractériser la complexité du calcul du permanent de matrices $\{0,1\}$ permet donc de connaître celle de compter les couplages parfaits dans un graphe biparti.

Une autre façon de représenter un graphe biparti G pour lequel $|V_1| = |V_2| = n$ est par un graphe orienté G' à n sommets, dans lequel il y a un arc de i à j ssi dans G il y a une arête entre u_i et v_j (cf. figure 9.1). Dans ce contexte, un couplage parfait de G, décrit par une permutation σ de $\{1,\dots,n\}$ (où u_i est relié à $v_{\sigma(i)}$), est donné par les cycles de la permutation et correspond donc à une *couverture* des sommets de G' par des cycles. Plus précisément, une *couverture par cycles* de G' est un ensemble de cycles de G' disjoints (à

1. Cet algorithme, dont le fonctionnement non trivial à base de « chemins augmentants » ne nous intéresse pas ici, est dû à Edmonds [Edm65].

la fois en termes d'arcs et de sommets) dont la réunion passe par tous les sommets de G'. Ainsi, le nombre de couplages parfaits de G est égal au nombre de couvertures par cycles de G'.

Nous voyons donc les nombreuses interprétations du permanent qui, outre les considérations de complexité que nous allons étudier, font tout son intérêt.

9.5.2 Complétude du permanent

Nous sommes maintenant prêts pour montrer que le permanent est difficile à calculer. En d'autres termes, compter le nombre de couplages parfaits d'un graphe biparti est difficile. C'est un exemple intéressant car la plupart des problèmes ♯P-complets proviennent de problèmes de *décision* eux-mêmes NP-complets, alors qu'ici le problème de décision associé (l'existence d'un couplage parfait) est dans P : décider l'existence d'un couplage parfait est facile tandis que compter le nombre de couplages parfaits est difficile puisque par le théorème de Toda c'est au moins aussi difficile que toute la hiérarchie polynomiale. Nous définissons tout d'abord formellement le problème $\texttt{PERMANENT}_{\{0,1\}}$:

- *entrée* : M une matrice $n \times n$ à coefficients dans $\{0,1\}$;
- *sortie* : le permanent de M (entier compris entre 0 et $n!$).

Nous rappelons qu'il s'agit ainsi d'une fonction $f : \{0,1\}^\star \to \mathbb{N}$. Plutôt qu'une matrice, on verra aussi l'entrée comme un graphe orienté dont on cherche à compter le nombre de couvertures par cycles. La complétude du permanent a été montrée par Valiant [Val79a].

9-AP Théorème (Valiant, 1979)

$\texttt{PERMANENT}_{\{0,1\}}$ est ♯P-complet pour les réductions de comptage $\leqslant_c^{\mathsf{P}}$.

Nous montrons d'abord que ce problème est dans ♯P avant de montrer sa difficulté.

9-AQ Lemme

$\texttt{PERMANENT}_{\{0,1\}}$ est dans ♯P.

Démonstration Soit A le langage suivant :

$$A = \{(M,\sigma) \mid \sigma \in \mathscr{S}_n \text{ et } \prod_{i=1}^{n} m_{i,\sigma(i)} = 1\},$$

où M désigne une matrice $n \times n$ à coefficients $m_{i,j} \in \{0,1\}$ et σ une permutation de $\{1,\dots,n\}$. Si l'on définit

$$f(M) = |\{\sigma \mid (M,\sigma) \in A\}|$$

alors $f(M) = \mathrm{per}(M)$. Puisque $A \in \mathsf{P}$, on en déduit que $\texttt{PERMANENT}_{\{0,1\}} \in ♯\mathsf{P}$. □

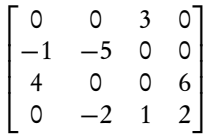
La preuve de difficulté du permanent se fait en deux étapes. Nous montrons d'abord que $\texttt{PERMANENT}_{\mathbb{Z}}$, le calcul du permanent de matrices à coefficients entiers (et non seulement $\{0,1\}$), est $\sharp\mathsf{P}$-difficile, en réduisant $\sharp\texttt{3SAT}$. Puis nous montrerons comment réduire $\texttt{PERMANENT}_{\mathbb{Z}}$ à $\texttt{PERMANENT}_{\{0,1\}}$.

9-AR **Remarque** Le problème $\texttt{PERMANENT}_{\mathbb{Z}}$ n'est pas dans $\sharp\mathsf{P}$ car il peut prendre des valeurs négatives (on peut néanmoins le mettre dans GapP), c'est pourquoi nous parlons seulement de $\sharp\mathsf{P}$-difficulté et non de $\sharp\mathsf{P}$-complétude.

Tout d'abord, nous avons besoin de définir une version pondérée des couvertures par cycles. La matrice d'adjacence contient maintenant des entiers relatifs quelconques. Un nombre non nul w signifie que l'arc correspondant dans le graphe porte le poids w. Le poids d'une couverture par cycles c est le produit des poids de ses arcs $\prod_{a\in c} w(a)$; le poids total des couvertures par cycles est la somme des poids de toutes les couvertures par cycles :

$$\text{poids total}_{\text{cc}} = \sum_{c}\prod_{a\in c} w(a).$$

Il s'agit là encore de la valeur du permanent de la matrice (cf. figure 9.2). Pour la clarté des figures, les poids égaux à 1 ne seront pas représentés sur les arcs des graphes : en d'autres termes, un arc sans étiquette a pour poids 1.



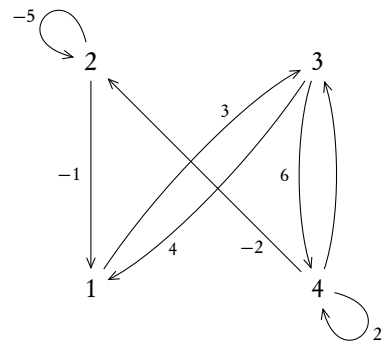


Figure 9.2 – La somme des poids des couvertures par cycles est égale au permanent de la matrice.

Dans une réduction de $\sharp\texttt{3SAT}$ à $\texttt{PERMANENT}_{\mathbb{Z}}$, le but sera donc de construire un graphe pondéré dont le poids total des couvertures par cycles sera en rapport avec le nombre de solutions d'une instance φ de $\sharp\texttt{3SAT}$.

9-AS Lemme

$\sharp$3SAT se réduit à $\texttt{PERMANENT}_{\mathbb{Z}}$ par une réduction de comptage $\leqslant_c^{\mathsf{p}}$.

Démonstration À partir d'une instance φ de $\sharp$3SAT (une formule booléenne en 3-CNF), nous allons construire une instance G_φ de $\texttt{PERMANENT}_{\mathbb{Z}}$, c'est-à-dire un graphe orienté dont les arcs sont pondérés par des entiers de $\mathbb{Z}$. Sans perte de généralité, on suppose que chacune des clauses de φ a exactement trois littéraux (et non seulement « au plus trois »). Si m est le nombre de clauses de φ, alors le poids total des couvertures par cycles de G_φ sera $4^{3m}\sharp\varphi$ (où $\sharp\varphi$ dénote, rappelons-le, le nombre d'affectations satisfaisant φ).

La construction de G_φ requiert la définition d'un gadget de littéral et d'un gadget de clause, donnés à la figure 9.3. On associe à chaque variable x de φ un gadget de littéral, dont on nommera par commodité les arcs extérieurs x et $\neg x$. On associe également à chaque clause C_i un gadget de clause. Si φ a n variables et m clauses, on obtient ainsi $n+m$ « blocs » indépendants (pour l'instant), avec en tout $2n+4m$ sommets.

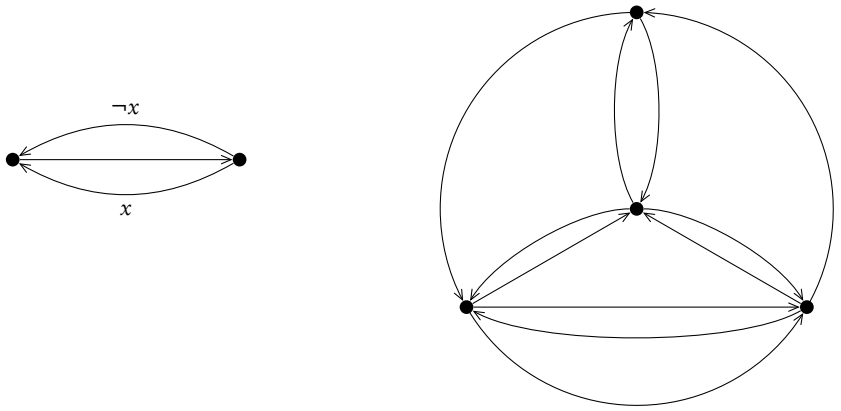
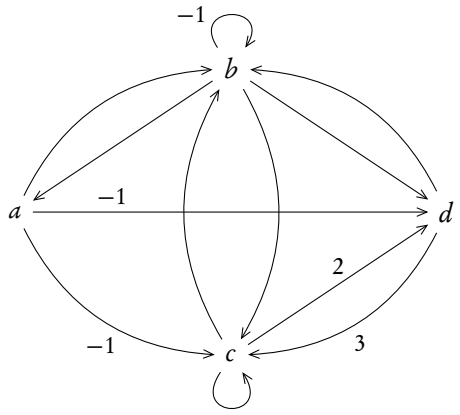


FIGURE 9.3 – Gadgets de littéral et de clause.

Afin de relier entre eux ces blocs indépendants, nous définissons un gadget de liaison que l'on appellera « ou-exclusif » (ou XOR), donné à la figure 9.4. Ce gadget permet de relier des paires d'arcs (s,s') et (t,t') du graphe que l'on est en train de définir. Attention, ce gadget *supprime* les deux arcs existants (s,s') et (t,t') et relie les sommets s,s',t et t' par la construction indiquée à la figure 9.4. Ainsi, certains arcs des gadgets de littéral et de clause disparaîtront au cours de la construction, au profit des arcs de nombreux gadgets ou-exclusif.

Chacun des trois arcs externes d'un gadget de clause représente un littéral de la clause correspondante. Il ne reste plus qu'à relier chaque arc externe des gadgets de clause au littéral qui correspond dans les gadgets de littéral, grâce au gadget de liaison XOR. La construction finale est illustrée à la figure 9.5.



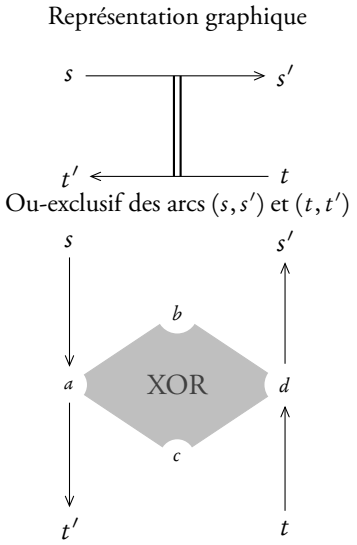


FIGURE 9.4 – Gadget ou-exclusif et liaison de deux arcs par le gadget. Attention, les arcs (s,s') et (t,t') de la représentation graphique disparaissent lorsqu'on les relie par le gadget.

Il est clair que le graphe G_φ ainsi défini peut se construire en temps polynomial à partir de φ. Il reste à montrer que le nombre de couvertures par cycles de G_φ est $4^{3m}\sharp\varphi$, c'est-à-dire que chacun des $3m$ gadgets XOR multiplie par 4 le nombre de solutions de φ.

Pour cela nous avons besoin de plusieurs observations sur le gadget de clause et le gadget XOR. Concernant le gadget de clause de la figure 9.3, on vérifiera aisément que :

- aucune couverture par cycles du gadget de clause ne passe par les trois arcs externes ;
- pour tout sous-ensemble strict des trois arcs externes (y compris l'ensemble vide), il existe une et une seule couverture par cycles empruntant les arcs externes du sous-ensemble et évitant les autres.

Pour comprendre l'appellation « ou-exclusif » du gadget de la figure 9.4, il faut imaginer un graphe H, auquel on adjoint un XOR entre deux arcs (s,s') et (t,t') (comme illustré à la figure 9.4) : on appelle H' le graphe obtenu. On considère alors une couverture par cycles C de H :

- si ni l'arc ss' ni tt' ne font partie de C, alors la seule façon « d'étendre » C au graphe H' est d'ajouter à C une couverture par cycles du gadget XOR (on rappelle que les arcs ss' et tt' n'existent plus dans H'). Mais on vérifie que la

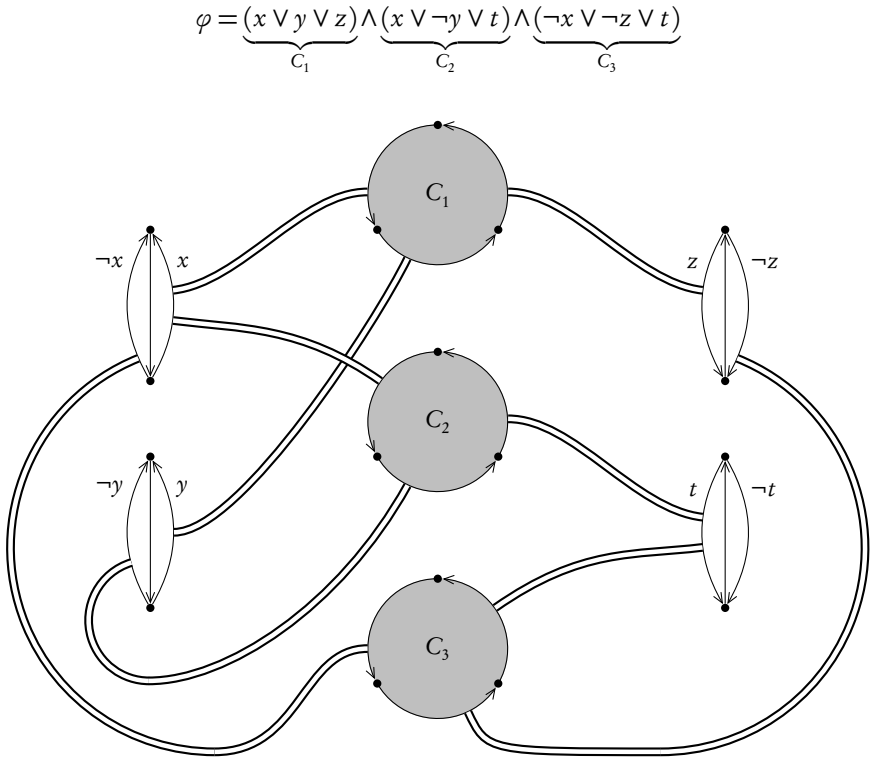


FIGURE 9.5 – Construction générale pour la formule $(x \vee y \vee z) \wedge (x \vee \neg y \vee t) \wedge (\neg x \vee \neg z \vee t)$.

somme des poids des couvertures par cycles du XOR est nulle : en effet, cela revient à évaluer le permanent de la matrice d'adjacence du gadget, à savoir

$$\begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix}.$$

Le permanent de cette matrice est nulle, confirmant que le poids total des couvertures par cycles du XOR est nul ;

– si les deux arcs ss' et tt' font partie de C, alors la seule façon « d'étendre » C au graphe H' est de remplacer l'arc ss' par sa et ds', et de remplacer l'arc tt' par td et at'. Pour obtenir une couverture par cycles, il faut compléter la couverture du gadget (les sommets b et c ne sont pas encore couverts) : il y a deux manières d'y parvenir, d'une part en prenant la boucle sur b et celle sur c (poids de la couverture -1), d'autre part en prenant le cycle bcb (poids 1). Le poids total de ces deux couvertures est donc nul ;

– si seul l'arc ss' fait partie de C et non tt', alors la seule façon « d'étendre » C au graphe H' est de remplacer l'arc ss' par sa et ds', puis de couvrir le reste du gadget en créant un chemin de a à d. Pour ce faire, on peut d'une part utiliser l'arc ad, puis soit boucler sur b et sur c (poids 1), soit utiliser le cycle bcb (poids -1) ; on peut d'autre part utiliser l'arc ab, puis soit bd en bouclant sur c (poids 1) soit bc et cd (poids 2) ; enfin, on peut aussi utiliser l'arc ac puis soit cd en bouclant sur b (poids 2) soit cb et bd (poids -1). Au total, la somme des poids des couvertures par cycles est 4 ;

– le dernier cas est celui où seul l'arc tt' fait partie de C et non ss'. Alors la seule façon « d'étendre » C au graphe H' est de remplacer l'arc tt' par td et at', puis de couvrir le reste du gadget en créant un chemin de d à a. Pour ce faire, on peut emprunter les arcs db et ba et boucler sur c (poids 1), ou emprunter les arcs dc, cb et ba (poids 3). Au total, la somme des poids des couvertures par cycles est 4.

On comprend mieux l'appellation XOR du gadget : si ss' et tt' sont tous deux dans la couverture C ou en dehors d'elle, alors les extensions possibles de C à H' donnent un poids nul, tandis que si $ss' \in C$ et $tt' \notin C$ ou inversement, alors les extensions possibles de C à H' donnent un poids non nul égal à 4.

Revenons à notre construction générale. Si σ est une affectation des variables satisfaisant φ, alors on construit une couverture par cycles en commençant par emprunter les arcs des gadgets de littéral correspondant à σ (c'est-à-dire en empruntant l'arc x si x est vrai et l'arc $\neg x$ si x est faux). Pour obtenir un poids non nul, les propriétés du gadget XOR ci-dessus imposent de choisir exactement les arcs externes des gadgets de clause qui *ne sont pas* reliés (par XOR) aux arcs de littéral choisis précédemment. Puisque σ satisfait φ, cela signifie qu'aucune clause n'a vu ses trois arcs externes choisis. Par la propriété du gadget de clause ci-dessus, on peut donc compléter d'une unique manière la couverture par cycles. Au total, puisqu'on traverse les $3m$ gadgets XOR qui comptent chacun pour un poids 4, on obtient un poids 4^{3m} correspondant à la solution σ.

Réciproquement, si un choix d'arcs dans les gadgets de littéral donne un poids non nul dans une couverture par cycles, cela signifie qu'aucun gadget XOR n'a un poids nul, donc que les arcs externes choisis dans les gadgets de clause sont exactement ceux qui ne sont pas reliés à un arc de littéral choisi. Aucune clause n'a ses trois arcs externes choisis car sinon le poids serait nul (cf. propriété du gadget de clause ci-dessus). Donc chaque clause est satisfaite par le choix des littéraux, ce qui signifie qu'il s'agit d'une solution de φ.

Au total, le poids des couvertures par cycles de G_φ est bien $4^{3m}\sharp\varphi$ comme annoncé. On a donc

$$\sharp\varphi = s(\texttt{PERMANENT}_{\mathbb{Z}}(G_\varphi))$$

où s est la fonction qui divise par 4^{3m}. Nous avons montré que $\sharp$3SAT se réduit à $\texttt{PERMANENT}_{\mathbb{Z}}$. □

9-AT **Lemme**

$\texttt{PERMANENT}_{\mathbb{Z}}$ se réduit à $\texttt{PERMANENT}_{\{0,1\}}$ par une réduction de comptage $\leqslant_c^{\mathsf{p}}$.

Démonstration Il s'agit de transformer une instance de $\texttt{PERMANENT}_{\mathbb{Z}}$, c'est-à-dire un graphe orienté G dont les arcs sont étiquetés par des entiers relatifs, en une instance de $\texttt{PERMANENT}_{\{0,1\}}$, c'est-à-dire un graphe orienté H dont les arcs ne portent pas de poids.

Les poids positifs de G ne posent pas trop de problème : comme illustré à la figure 9.6, un arc portant une puissance de 2 est remplacé par une succession de « triangles » permettant chacun de multiplier par deux le nombre de couvertures par cycles. Un poids positif quelconque se décompose en une somme de puissances de 2 et on remplace l'arc par un ensemble de blocs de puissances de 2 en « parallèle ». On remarque le rôle essentiel des boucles permettant de compléter les couvertures par cycles lorsqu'un des 2^{a_i} est « choisi ».

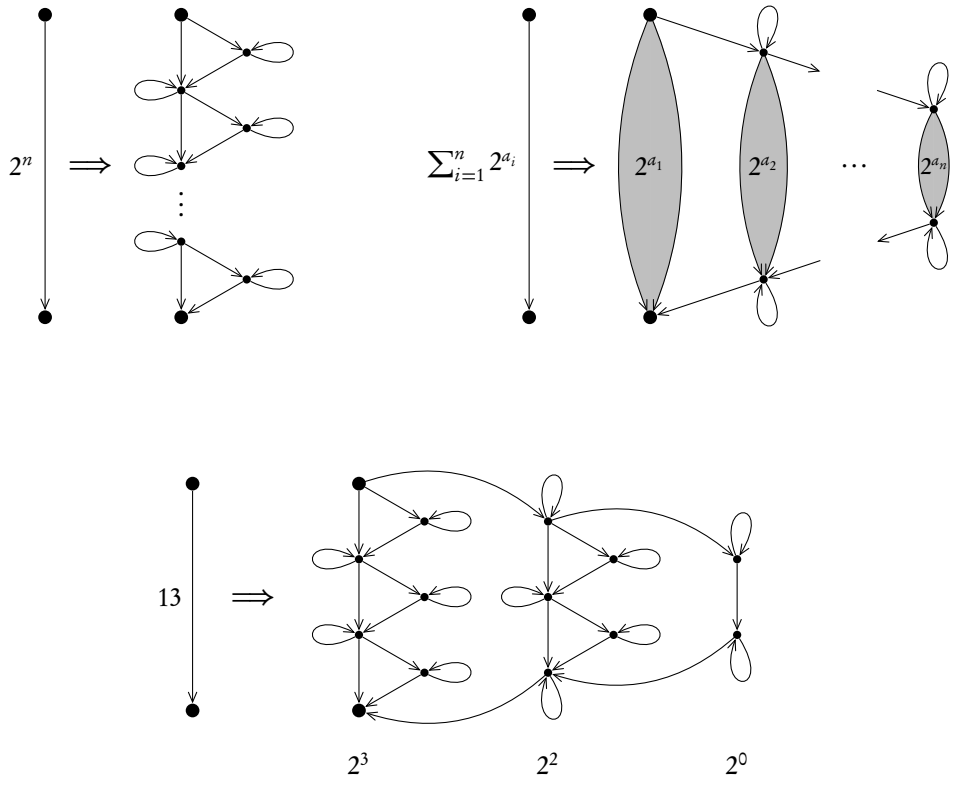


FIGURE 9.6 – Remplacement d'un arc pondéré par des arcs non pondérés : poids égal à une puissance de 2, poids quelconque et exemple avec le poids 13.

En revanche on ne peut pas remplacer directement les arcs de poids négatif. L'astuce consiste à travailler modulo. Soit N un entier suffisamment grand pour que l'on soit assuré que la valeur absolue du permanent de la matrice d'adjacence M_G de G (à coefficients dans $\mathbb{Z}$) soit strictement inférieure à $N/2$ (la raison pour laquelle on choisit $N/2$ et non N vient de ce que le permanent peut être négatif, cf. ci-dessous). On peut prendre par exemple $(1+2n!m^n)$, où n désigne la taille de la matrice et m la valeur absolue maximale des coefficients de la matrice. Cet entier N a un nombre de chiffres polynomial et est calculable en temps polynomial. On remplace alors un poids négatif $-w$ par $N-w$ (et on applique la transformation de la figure 9.6 pour revenir sur $\{0,1\}$). Au final, on a construit en temps polynomial un graphe non pondéré H. Modulo N, le permanent de sa matrice d'adjacence M_H est égal à celui de M_G, qu'on notera p_G. Soit $p_H \in [0, N-1]$ le permanent de M_H modulo N. Puisque N est strictement supérieur à $2|p_G|$, si $p_H \leqslant N/2$ alors $p_G = p_H$; tandis que si $p_H > N/2$ alors $p_G = p_H - N$ (valeur négative). La fonction s permettant de retrouver p_G à partir du permanent de M_H est donc calculable en temps polynomial : il s'agit de prendre le modulo N et de renvoyer p_H ou $p_H - N$ comme ci-dessus. Au final,

$$\texttt{PERMANENT}_{\mathbb{Z}}(M_G) = s(\texttt{PERMANENT}_{\{0,1\}}(M_H))$$

ce qui montre que $\texttt{PERMANENT}_{\mathbb{Z}}$ se réduit à $\texttt{PERMANENT}_{\{0,1\}}$. □

On déduit de ces trois lemmes la ♯P-complétude de $\texttt{PERMANENT}_{\{0,1\}}$, c'est-à-dire le théorème 9-AP

9-AU **Remarque** Si $\mathsf{P} \neq \mathsf{NP}$, aucune réduction de ♯3SAT à $\texttt{PERMANENT}_{\{0,1\}}$ ne peut être parcimonieuse. S'il existait une telle réduction parcimonieuse e, alors le nombre de solutions d'une formule φ serait égal au nombre de couplages parfaits de $e(\varphi)$. Dans ce cas, décider si $\varphi \in \texttt{3SAT}$ se ferait en temps polynomial puisqu'il suffirait de décider s'il existe un couplage parfait dans $e(\varphi)$, ce qui est polynomial.

10 Protocoles interactifs

La caractérisation de la classe NP en termes de certificat peut être réinterprétée par le mécanisme suivant : un « prouveur » P cherche à convaincre un « vérificateur » V que l'entrée x est dans le langage A. Pour cela, P envoie à V une prétendue « preuve » que $x \in A$, que V va ensuite vérifier. Si $x \in A$ alors il existe un certificat que P peut envoyer et dont V peut vérifier la correction en temps polynomial. En revanche si $x \notin A$ alors il n'y a pas de certificat et quel que soit le message envoyé par P, il sera rejeté par V. Dans ce scénario, la « preuve » doit avoir une taille polynomiale, le prouveur est tout-puissant et le vérificateur fonctionne en temps déterministe polynomial.

Cela correspond également plus ou moins à un mathématicien donnant dans un article sa preuve d'un résultat pour que celle-ci soit vérifiée par un relecteur : la preuve peut être très sophistiquée mais il suffit de la suivre pas à pas pour la vérifier.

Cette vision des choses appelle naturellement l'étude de variantes. Tout d'abord sur les capacités du vérificateur : on souhaite certes pouvoir vérifier la preuve efficacement, mais on a vu que les algorithmes probabilistes polynomiaux peuvent être considérés efficaces. Une première variante consiste donc à autoriser le vérificateur à être probabiliste en lui imposant une faible probabilité d'erreur.

Ensuite, si l'on reprend l'analogie du mathématicien désirant valider sa preuve, plutôt que d'écrire un article il peut aussi l'expliquer au tableau devant des étudiants ou des pairs. Ceux-ci vont lui poser des questions pour clarifier les choses et une *interaction* entre le prouveur et le vérificateur débute alors. Cette interaction consiste en plusieurs tours au cours desquels le vérificateur pose des questions et le prouveur fournit des réponses.

Ce cadre de réflexion s'est avéré particulièrement instructif en complexité. De manière surprenante, Shamir a montré en 1990 que la combinaison d'un vérificateur probabiliste et d'une interaction entre prouveur et vérificateur apporte une grande puissance puisqu'on capture alors la classe PSPACE (théorème 10-G). Pour illustrer cela, supposons que monsieur V, par essence limité, ne parvienne pas à distinguer un Romanée-Conti 2005 d'un vulgaire vin de table. Monsieur P, amateur de vin, souhaite quant à lui convaincre son

homologue que les deux vins sont réellement différents. Pour cela, ils mettent en place le protocole suivant. Au départ, un verre à gauche contient le précieux nectar tandis qu'à droite un autre verre contient la piquette. Puis on effectue dix tours comme suit : hors de la vue de P, V choisit au hasard de laisser les verres en place ou de les inverser ; P doit alors goûter les vins et dire lequel est le Romanée-Conti. Puisqu'il a lui-même inversé ou non les verres, V peut savoir si P dit vrai, et s'il dit vrai dix fois de suite, alors V est convaincu avec grande probabilité que les vins sont en effet différents et qu'il doit faire des progrès en œnologie. On note que ce protocole ne fonctionne que grâce à l'interaction et au hasard.

À ce chapitre nous allons étudier différentes variantes de ces protocoles interactifs, comparer les classes obtenues avec celles vues jusqu'à présent, voir le résultat de Shamir évoqué ci-dessus (intéressant également car il ne relativise pas) et montrer que le problème d'isomorphisme de graphe n'est probablement pas NP-complet. Nous mentionnerons également le théorème PCP qui caractérise la classe NP en termes de protocoles interactifs et permet de montrer des résultats d'inapproximabilité de problèmes NP-complets.

La notion de protocoles interactifs et les définitions de ce chapitre ont été introduites par Babai [Bab85] et par Goldwasser, Micali et Rackoff [GMR85].

10.1 Les classes IP

10.1.1 Définition

La première variante que nous allons étudier s'appelle simplement *interactive protocols* (IP). Ce sont en général des protocoles dont le nombre de tours dépend de la taille de l'entrée. Le vérificateur est une machine probabiliste dont les bits aléatoires sont privés, c'est-à-dire qu'ils ne sont pas dévoilés au prouveur (ils seront notés r dans la suite et considérés comme un mot donné une fois pour toute). Après une série de questions du vérificateur et de réponses du prouveur, un mot du langage doit être accepté avec grande probabilité par le vérificateur tandis qu'un mot hors du langage doit être rejeté avec grande probabilité. La définition formelle est la suivante.

10-A Définition

- Un prouveur est une fonction $P : \Sigma^\star \to \Sigma^\star$, sans contrainte de calculabilité. Le prouveur P produit ses réponses (des mots sur Σ) en fonction des questions du vérificateur (des mots sur Σ) et de l'entrée x. Ainsi, si $y_1, \ldots, y_k \in \Sigma^\star$ sont les k premières questions du vérificateur, alors $P(x, y_1, \ldots, y_k) = z_k \in \Sigma^\star$ est la k-ème réponse du prouveur.
- Si x désigne le mot dont on veut décider l'appartenance au langage, le vérificateur est une fonction $V : \Sigma^\star \to \Sigma^\star$ calculable en temps déterministe polynomial en $|x|$ qui produit les questions au prouveur en fonction de ses réponses précédentes, de l'entrée x et de bits aléatoires. Ainsi, si $z_1, \ldots, z_k \in \Sigma^\star$ sont les k premières réponses

du prouveur, alors $V(x,r,z_1,\dots,z_k) = y_{k+1} \in \Sigma^\star$ est la $(k+1)$-ème question du vérificateur selon le mot aléatoire r, où le temps de calcul de V est $|x|^{O(1)}$ quels que soient les autres arguments (cf. remarque 10-B).

Pour alléger le formalisme, on considérera la dernière question du vérificateur comme étant sa décision d'accepter ou non le mot d'entrée x : 1 si x est accepté, 0 s'il est rejeté.

– L'exécution du protocole entre un prouveur P et le vérificateur V donne donc lieu à l'échange des messages suivants :
  - $y_1 = V(x,r)$ (première question du vérificateur) ;
  - $z_1 = P(x,y_1)$ (première réponse du prouveur) ;
  - …
  - $y_i = V(x,r,z_1,\dots,z_{i-1})$ (i-ème question du vérificateur) ;
  - $z_i = P(x,y_1,\dots,y_i)$ (i-ème réponse du prouveur) ;
  - …
  - $V(x,r,z_1,\dots,z_s) \in \{0,1\}$: décision du vérificateur.

  On dira que le nombre de tours dans ce protocole est $2s$: c'est le nombre de messages échangés par P et V.

– Si $s : \mathbb{N} \to \mathbb{N}$ est une fonction, $\mathsf{IP}[s(n)]$ est l'ensemble des langages L possédant un protocole à $2\lfloor s(n)/2 \rfloor$ tours, c'est-à-dire tels qu'il existe un polynôme $p(n)$ et un vérificateur V fonctionnant en temps $p(|x|)$, satisfaisant :

$$\begin{cases} x \in L \implies & \text{il existe un prouveur } P \text{ tel que} \\ & \Pr_{r\in\{0,1\}^{p(|x|)}}(V(x,r,z_1,\dots,z_{\lfloor s(|x|)/2\rfloor}) = 1) \geqslant 2/3 \\ x \notin L \implies & \text{pour tout prouveur } P, \\ & \Pr_{r\in\{0,1\}^{p(|x|)}}(V(x,r,z_1,\dots,z_{\lfloor s(|x|)/2\rfloor}) = 1) \leqslant 1/3, \end{cases}$$

  où $z_{i+1} = P(x, V(x,r), V(x,r,z_1),\dots, V(x,r,z_1,\dots,z_i))$ est la $(i+1)$-ème réponse du prouveur P.

– La classe IP est définie par $\mathsf{IP} = \bigcup_{k\in\mathbb{N}} \mathsf{IP}[n^k]$, c'est-à-dire des protocoles ayant un nombre polynomial de tours.

10-B **Remarque** Dans cette définition il n'y a pas de limitation de la taille des messages échangés car celle-ci est contrainte par le temps de calcul de V, qui doit fonctionner en temps polynomial en $|x|$ (ne représentant qu'une partie de son entrée). Il ne peut donc ni produire ni lire des messages de taille plus que polynomiale en $|x|$. De même pour le nombre de bits aléatoires qui peut être supposé polynomial.

En d'autres termes, la contrainte sur le temps de calcul de V permet de supposer sans

perte de généralité que r, $(z_1,\dots,z_k)$ et y_{k+1} sont de taille polynomiale en $|x|$.

En répétant le protocole un nombre polynomial de fois et en prenant la réponse majoritaire (comme à la proposition 6-F), on obtient le lemme suivant.

10-C Lemme

La classe IP est inchangée si, au lieu d'autoriser une erreur 1/3 dans la définition 10-A, on limite l'erreur à 2^{-n^k} pour un k fixé.

C'est-à-dire que si $x \in L$ alors V accepte avec probabilité $\geqslant 1-2^{-n^k}$ et si $x \notin L$ alors V accepte avec probabilité $\leqslant 2^{-n^k}$

✎ **10-D Exercice**

Montrer le lemme précédent.

10.1.2 Isomorphisme de graphes

Un problème bien connu est celui de l'isomorphisme de graphes (ISO) :

- *entrée :* deux graphes non orientés $G_1=(V_1,E_1)$ et $G_2=(V_2,E_2)$;
- *question :* G_1 et G_2 sont-ils isomorphes ?

En d'autres termes et comme illustré à la figure 10.1, si $V_1=V_2=\{1,\dots,n\}$, existe-t-il une permutation σ des sommets $\{1,\dots,n\}$ de G_1 telle que $\forall(i,j)$,

$$(i,j)\in E_1 \iff (\sigma(i),\sigma(j))\in E_2\,?$$

Ce problème est bien sûr dans NP puisqu'il suffit de deviner la permutation σ et de vérifier qu'il s'agit d'un isomorphisme.

FIGURE 10.1 – Deux graphes isomorphes.

Le complémentaire de ce problème, c'est-à-dire non-isomorphisme de graphes (coISO) où il s'agit de décider si G_1 et G_2 ne sont pas isomorphes, est donc dans coNP. En revanche, on ne sait pas s'il est dans NP. On peut néanmoins donner un protocole IP pour le problème coISO dans le même esprit que le protocole de l'introduction pour repérer la bouteille de Romanée-Conti. Ce protocole est dû à Goldreich, Micali et Wigderson [GMW87].

Protocole pour coISO sur l'entrée (G_1, G_2) :

V Tirer au hasard $i \in \{1,2\}$. Appliquer une permutation aléatoire aux sommets de G_i pour obtenir un nouveau graphe H. Envoyer H à P.

P Identifier quel graphe G_j, pour $j \in \{1,2\}$, a été permuté pour obtenir H. Envoyer j à V.

V Accepter ssi $i = j$.

Analysons maintenant ce protocole.

- Si $(G_1, G_2) \in$ coISO, alors les deux graphes ne sont pas isomorphes et un prouveur est capable de déterminer lequel des deux graphes a été permuté pour obtenir H. Il peut donc donner à V la bonne valeur de j, à savoir i, et V accepte avec probabilité 1.
- Si $(G_1, G_2) \notin$ coISO, alors les deux graphes sont isomorphes et un prouveur est incapable de déterminer lequel des deux graphes a été permuté pour obtenir H, puisque les deux auraient pu faire l'affaire. Ainsi il donne la bonne réponse pour j avec probabilité 1/2 seulement.

Si l'on souhaite réduire la probabilité d'erreur sous 1/3, une simple modification suffit, puisqu'à l'instar des classes probabilistes du chapitre 6, on peut réduire la probabilité d'erreur en répétant le protocole :

Protocole avec erreur 1/4 pour coISO sur l'entrée (G_1, G_2) :

V Tirer au hasard $i, i' \in \{1,2\}$. Appliquer une permutation aléatoire aux sommets de G_i pour obtenir un nouveau graphe H, et de $G_{i'}$ pour obtenir H'. Envoyer H et H' à P.

P Identifier quels graphes G_j et $G_{j'}$, pour $j, j' \in \{1,2\}$, ont été permutés pour obtenir H et H' respectivement. Envoyer j et j' à V.

V Accepter ssi $i = j$ et $i' = j'$.

Nous avons donc un protocole IP à deux tours (une question de V/une réponse de P) pour coISO, ce qui place ce problème dans la classe IP.

10-E **Remarque** Dans le protocole pour coISO que nous venons de voir, le vérificateur ne fait pas d'erreur dans le cas où l'entrée est dans le langage. Il s'agit en réalité d'un résultat général comme nous le verrons après la preuve de IP = PSPACE : tout langage de IP a un protocole où le vérificateur accepte avec probabilité 1 lorsque $x \in L$.

✎ **10-F** **Exercice**

En utilisant n permutations aléatoires des graphes G_1 et G_2 plutôt que seulement deux, montrer que l'on peut diminuer la probabilité d'erreur à 2^{-n} dans le protocole précédent.

Nous reverrons ce problème coISO lorsque nous aborderons les protocoles à bits aléatoires publics.

10.1.3 IP = PSPACE

On pourrait a priori penser que le mécanisme de preuve de la classe IP n'ajoute guère de puissance par rapport à une simple preuve vérifiée de manière déterministe comme pour la classe NP, puisque des indices laissent penser qu'un calcul probabiliste peut être déterminisé (cf. chapitre 12). Mais en réalité la combinaison de l'aléatoire du vérificateur et de l'interaction entre prouveur et vérificateur (nombre polynomial de tours) apporte une grande puissance à IP, puisque IP = PSPACE. Ce résultat de Shamir [Sha90], outre son intérêt intrinsèque, est l'un des rares qui ne relativise pas, comme nous le verrons à la fin de cette section. La démonstration que nous donnons est due à Shen [She92].

10-G **Théorème (Shamir, 1990)**

IP = PSPACE

De manière naturelle, nous allons scinder la preuve en deux parties, l'inclusion la plus simple IP ⊆ PSPACE d'abord, puis l'inclusion PSPACE ⊆ IP où nous donnerons un protocole pour le problème PSPACE-complet QBF (cf. définition 4-AA et théorème 4-AC).

10-H **Lemme**

IP ⊆ PSPACE.

Pour montrer ce lemme, nous allons d'abord voir que l'on peut supposer que le prouveur est dans PSPACE.

10-I Lemme

Dans IP, le prouveur valide peut être supposé dans PSPACE sans perte de généralité.

Plus précisément, à l'image de IP, soit $\mathsf{IP}_{\mathsf{PSPACE}}$ la classe des langages L tels qu'il existe un prouveur P fonctionnant en espace polynomial en $|x|$ satisfaisant :

- si $x \in L$ alors V accepte x avec probabilité $\geqslant 2/3$ lors de son dialogue avec P ;
- si $x \notin L$ alors pour tout prouveur P' (sans restriction de calculabilité), V rejette x avec probabilité $\geqslant 2/3$ lors de son dialogue avec P'.

Alors $\mathsf{IP}_{\mathsf{PSPACE}} = \mathsf{IP}$.

Idée de la démonstration La difficulté provient du fait que le prouveur dans IP est tout-puissant et peut donc calculer des choses qui ne sont pas calculables en espace polynomial. Mais pour chaque suite de réponses possibles, en énumérant tous les bits aléatoires un prouveur peut calculer la probabilité que le vérificateur accepte, et ainsi trouver la suite de réponses qui maximise la probabilité d'acceptation. En particulier, si $x \in L$ alors cette probabilité est au moins $2/3$ et V accepte.

Puisque l'énumération porte sur des mots de taille polynomiale, le prouveur peut l'effectuer en espace polynomial.

Démonstration L'inclusion $\mathsf{IP}_{\mathsf{PSPACE}} \subseteq \mathsf{IP}$ est évidente : pour $L \in \mathsf{IP}_{\mathsf{PSPACE}}$, si $x \in L$ alors le prouveur P permet à V d'accepter, donc il existe un prouveur qui convainc V ; si $x \notin L$ alors aucun prouveur ne convainc V. Donc $L \in \mathsf{IP}$.

Nous montrons maintenant l'autre inclusion $\mathsf{IP} \subseteq \mathsf{IP}_{\mathsf{PSPACE}}$. Soit $L \in \mathsf{IP}$ avec un protocole à $2T = n^{O(1)}$ tours. Si $x \notin L$ alors aucun prouveur ne convainc V, ce qui correspond aussi à la définition de $\mathsf{IP}_{\mathsf{PSPACE}}$. Pour le cas contraire, on définit un prouveur P fonctionnant en espace polynomial qui convainc V que $x \in L$.

Nous avons d'abord besoin de définir deux probabilités :

- $p_{y_1,z_1,y_2,\ldots,y_k,z_k}$ est la probabilité que le meilleur prouveur convainque V à partir du moment où les messages $y_1, z_1, \ldots, y_k, z_k$ ont été échangés (k questions $y_1, y_2, \ldots, y_k$ et k réponses $z_1, z_2, \ldots, z_k$) ;
- $q_{y_1,z_1,y_2,\ldots,y_k}$ est la probabilité que le meilleur prouveur convainque V à partir du moment où les messages $y_1, z_1, \ldots, y_k$ ont été échangés (k questions $y_1, y_2, \ldots, y_k$ et $(k-1)$ réponses $z_1, z_2, \ldots, z_{k-1}$).

Par définition de IP, nous avons donc $x \in L \iff p_\epsilon \geqslant 2/3$ (où le mot vide ϵ désigne le fait qu'aucun échange n'a encore eu lieu). Il est maintenant facile de calculer ces

probabilités en partant de la fin du protocole grâce aux relations suivantes :

$$\begin{cases} p_{y_1,z_1,\ldots,y_T,z_T} & = \Pr_r(V(x,r,z_1,\ldots,z_T)=1 \mid y_1,z_1,\ldots,y_T,z_T) \\ q_{y_1,z_1,\ldots,y_k} & = \max_{z_k} p_{y_1,z_1,\ldots,y_k,z_k} \\ p_{y_1,z_1,\ldots,y_k,z_k} & = \sum_{y_{k+1}} \Pr_r(y_{k+1} \mid y_1,z_1,\ldots,y_k,z_k) q_{y_1,z_1,\ldots,y_{k+1}} \end{cases}$$

où $k \geqslant 0$ et la notation $\Pr_r(y_{k+1} \mid y_1,z_1,\ldots,y_k,z_k)$ désigne la probabilité que la $(k+1)$-ème question soit y_{k+1} sachant que les k premières questions et réponses ont été $y_1,z_1,\ldots,y_k,z_k$, c'est-à-dire qu'on se restreint aux bits aléatoires r compatibles avec cet échange (ces probabilités conditionnelles ne seront considérées que quand la condition est non vide). Détaillons ces trois relations :

- la première relation donne exactement la définition de $p_{y_1,z_1,\ldots,y_T,z_T}$: la probabilité que V accepte sachant que les questions/réponses ont été $y_1,z_1,\ldots,y_T,z_T$ (c'est-à-dire que la probabilité est prise sur l'ensemble des r qui ont pu donner cet échange) ;
- la deuxième relation signifie simplement que le prouveur maximise la probabilité de convaincre V, en d'autres termes il choisit la meilleure réponse z_k ;
- la dernière s'interprète ainsi : le meilleur prouveur doit attendre la question y_{k+1} avant de pouvoir répondre z_{k+1} qui maximise $q_{y_1,z_1,\ldots,y_{k+1}}$, et donc la probabilité qu'il convainque V est la moyenne des $q_{y_1,z_1,\ldots,y_{k+1}}$ sur l'ensemble des $(k+1)$-ème questions y_{k+1} possibles de V.

Le calcul de $p_{y_1,z_1,\ldots,y_T,z_T}$ se fait en énumérant tous les mots r possibles et pour chacun en simulant le protocole avec ce choix de r et avec les réponses $z_1,\ldots,z_T$: on compte le nombre de fois que V accepte et que les questions de V ont bien été $y_1,\ldots,y_T$.
Si l'on donne $y_1,z_1,\ldots,y_k$, calculer $q_{y_1,z_1,\ldots,y_k}$ se fait en évaluant récursivement les $p_{y_1,z_1,\ldots,y_k,z_k}$ pour tous les z_k possibles.
Si l'on donne $y_1,z_1,\ldots,y_k,z_k$, calculer $p_{y_1,z_1,\ldots,y_k,z_k}$ se fait en évaluant récursivement les $q_{y_1,z_1,\ldots,y_{k+1}}$ pour tous les y_{k+1} possibles.
L'espace total utilisé pour ces calculs est polynomial car cela revient au pire à énumérer toutes les possibilités de $r,y_1,z_1,\ldots,y_T,z_T$, c'est-à-dire un nombre polynomial de bits.
On remarque que dans le calcul de $q_{y_1,z_1,\ldots,y_k}$ (deuxième relation), outre la valeur du max, on peut évidemment calculer la réponse z_k qui maximise la probabilité. Notre prouveur agit alors comme ceci pour la k-ème réponse, lorsque l'échange a jusqu'à présent consisté en $y_1,z_1,\ldots,y_k$:

- calculer $q_{y_1,z_1,\ldots,y_k}$ et le z_k correspondant qui maximise la probabilité ;
- répondre z_k.

Ce prouveur fonctionne en espace polynomial. Puisque la probabilité d'acceptation de V est maximisée, un tel prouveur convainc V avec probabilité $\geqslant 2/3$ si $x \in L$. □

Nous obtenons le lemme 10-H en corollaire.

Démonstration du lemme 10-H Soit $L \in \mathsf{IP}$: par le lemme 10-I, $L \in \mathsf{IP}_{\mathsf{PSPACE}}$ et a donc un protocole où le prouveur fonctionne en espace polynomial. Pour décider si $x \in L$, il suffit alors de simuler ce protocole : cela se fait en espace polynomial, donc $L \in \mathsf{PSPACE}$. □

Nous en venons maintenant à la partie difficile du théorème.

10-J Lemme

$\mathsf{PSPACE} \subseteq \mathsf{IP}$.

Plus précisément, il existe un protocole interactif de type IP pour le langage PSPACE-complet QBF, dans lequel un prouveur convainc le vérificateur avec probabilité 1 lorsque l'instance est dans le langage (c'est-à-dire que V accepte x avec probabilité 1 si $x \in \mathtt{QBF}$).

Tout d'abord un lemme simple permettant de justifier la stratégie de preuve qu'on va suivre.

10-K Lemme

La classe IP est close par réductions many-one polynomiales.

✎ **10-L Exercice**

Montrer le lemme précédent.

Il suffit donc de donner un protocole IP pour QBF, problème PSPACE-complet pour les réductions many-one polynomiales (cf. théorème 4-AC), puisque par clôture cela implique que la classe PSPACE est entièrement contenue dans IP.

Nous aurons également besoin du lemme suivant qui permet de vérifier de manière probabiliste l'égalité de deux polynômes.

10-M Lemme

Soit p un nombre premier. $\mathbb{F}_p$ désigne le corps à p éléments (les entiers modulo p). Si $q(x)$ et $q'(x)$ sont deux polynômes distincts de degré $\leqslant d$ sur $\mathbb{F}_p$, alors

$$\Pr_{r \in \mathbb{F}_p} (q(r) = q'(r)) \leqslant d/p.$$

Démonstration Si $q \neq q'$ alors $q - q' \neq 0$ et $(q - q')$ est de degré au plus d, donc $q - q'$ a au plus d racines sur le corps $\mathbb{F}_p$. Il y a donc au plus d valeurs de r pour lesquelles $(q - q')(r) = 0$, c'est-à-dire $q(r) = q'(r)$. □

Idée de la démonstration du lemme 10-J Le protocole interactif pour le problème QBF utilise une technique appelée *l'arithmétisation,* c'est-à-dire la transformation des formules booléennes en polynômes. Cela permet de se placer sur un corps fini plus grand que $\{0,1\}$ et autorise plus de marge de manœuvre.

En transformant la formule booléenne en un polynôme, puis les quantificateurs $\exists$ en sommes et $\forall$ en produits, il suffit de tester si la valeur obtenue est non nulle. Pour cela, le prouveur aide le vérificateur en lui fournissant une évaluation variable par variable, c'est-à-dire en se ramenant à des polynômes à une variable.

Démonstration du lemme 10-J

Hypothèses et notations Sans perte de généralité, on suppose que les instances de QBF sont de la forme

$$\varphi = \exists a_1 \forall a_2 \exists a_3 \dots \forall a_n \psi(a_1, \dots, a_n),$$

où ψ est une formule sans quantificateur en 3-CNF, n est pair et où $a_i \in \{0,1\}$ (c'est-à-dire que a_i désigne une unique variable et non un uple de variables).

On associe d'abord à ψ un polynôme Q_ψ de la manière suivante :

- à une variable booléenne a_i on associe la variable x_i, et à $\neg a_i$ on associe $(1 - x_i)$;
- à une clause constituée de trois littéraux, on associe la somme : par exemple, à la clause $(a_1 \vee \neg a_2 \vee a_3)$ est associé le polynôme

$$x_1 + (1 - x_2) + x_3 \,;$$

- enfin, Q_ψ est le produit des polynômes associés aux clauses.

En particulier, Q_ψ est un polynôme à n variables de degré m si ψ a m clauses. De plus, sur $\mathbb{Z}$, pour tout $(a_1, \dots, a_n) \in \{0,1\}^n$, $Q_\psi(a_1, \dots, a_n) \geqslant 0$ et

$$Q_\psi(a_1, \dots, a_n) > 0 \text{ ssi } \psi(a_1, \dots, a_n) = 1.$$

En d'autres termes, Q_ψ est non nul ssi ψ est vraie.

On manipulera ce polynôme sous sa forme factorisée, c'est-à-dire sans développer le produit des clauses, grâce à quoi son encodage est de taille linéaire.

On définit ensuite l'entier Q_φ représentant φ, de telle sorte que $Q_\varphi \geqslant 0$ et [$Q_\varphi > 0$ ssi φ est vraie] :

$$Q_\varphi = \sum_{a_1 \in \{0,1\}} \prod_{a_2 \in \{0,1\}} \sum_{a_3 \in \{0,1\}} \cdots \prod_{a_n \in \{0,1\}} Q_\psi(a_1, \dots, a_n).$$

Sur $\mathbb{Z}$, on a $\varphi \in$ QBF ssi $Q_\varphi \neq 0$. Néanmoins, les produits utilisés ici pour remplacer les quantificateurs $\forall$ font exploser le degré des polynômes successifs qui devient exponentiel, ce qui ne convient pas pour le protocole que l'on souhaite mettre en œuvre.

Opérateurs de linéarisation, somme et produit Pour réduire ce degré, on remarque que l'on ne travaille que sur des valeurs $a_i \in \{0,1\}$, pour lesquelles $a_i^k = a_i$ pour tout

k. Ainsi, pour tout $i \leqslant n$ et tout polynôme $Q(x_1,\dots,x_n)$, on définit l'opérateur de « linéarisation » de la i-ème variable

$$L_i(Q)(x_1,\dots,x_n) = \\ x_i Q(x_1,\dots,x_{i-1},1,x_{i+1},\dots,x_n) + (1-x_i)Q(x_1,\dots,x_{i-1},0,x_{i+1},\dots,x_n).$$

On a alors

$$\forall (a_1,\dots,a_n) \in \{0,1\}^n,\ Q(a_1,\dots,a_n) = L_i(Q)(a_1,\dots,a_n)$$

et le degré en x_i de $L_i(Q)$ est 1.
Pour faciliter les notations, on définit également deux autres opérateurs qui représentent l'arithmétisation des quantificateurs $\forall x_i$ et $\exists x_i$:

$$\forall_i(Q)(x_1,\dots,x_{i-1},x_{i+1},\dots,x_n) = \\ Q(x_1,\dots,x_{i-1},0,x_{i+1},\dots,x_n) \times Q(x_1,\dots,x_{i-1},1,x_{i+1},\dots,x_n)$$

et

$$\exists_i(Q)(x_1,\dots,x_{i-1},x_{i+1},\dots,x_n) = \\ Q(x_1,\dots,x_{i-1},0,x_{i+1},\dots,x_n) + Q(x_1,\dots,x_{i-1},1,x_{i+1},\dots,x_n)$$

grâce auxquels la variable x_i est supprimée. Ainsi l'entier Q_φ défini précédemment peut être exprimé sous la forme

$$Q_\varphi = \exists_1 \forall_2 \exists_3 \dots \forall_n (Q_\psi).$$

L'idée est d'insérer des opérateurs de linéarisation entre les opérateurs $\forall_i$ et $\exists_i$ afin de ne pas subir d'explosion du degré. On définit alors (où les opérateurs s'appliquent de droite à gauche, d'abord L_n, L_{n-1}, …, L_1 puis $\forall_n$, etc.)

$$S = \exists_1 L_1\ \forall_2 L_1 L_2\ \exists_3 L_1 L_2 L_3\ \forall_4 L_1 \dots L_4 \exists_5 \cdots \exists_{n-1} L_1 \dots L_{n-1}\ \forall_n L_1 \dots L_n (Q_\psi)$$

Les linéarisations assurent qu'après chaque application d'un opérateur $\forall_i$ (qui augmente potentiellement le degré), le polynôme est linéarisé [1]. On notera que cette expression est au final un entier car toutes les variables ont été instanciées grâce aux opérateurs $\exists_i$ et $\forall_i$: en effet, après l'application de chaque $\exists_i$ ou $\forall_i$, le polynôme ne contient plus que les variables $x_1,\dots,x_{i-1}$. De plus, on a $S = Q_\varphi$.

Résultats de l'arithmétisation Par les considérations qui précèdent, $\varphi \in \mathtt{QBF}$ ssi $S \neq 0$ sur $\mathbb{Z}$. C'est ce que notre protocole va tester. On remarque que $S \in [0, 3^{m2^n}]$ où m est le nombre de clauses de ψ, puisque pour tout $a_1,\dots,a_n \in \{0,1\}$, $Q_\psi(a_1,\dots,a_n) \in [0,3^m]$ et que chaque opérateur $\forall_i$ élève au plus au carré la valeur maximale, chaque opérateur $\exists_i$ la double tandis que L_i la laisse inchangée. Si $S \neq 0$ alors il a au plus $\log_2(S)$ diviseurs

1. On aurait pu éviter les linéarisations après les opérateurs $\exists_i$ mais celles-ci facilitent la description du protocole en homogénéisant les notations.

premiers. Ainsi grâce au théorème des nombres premiers A-O, il existe une constante α telle que si $S \neq 0$ alors dans tout intervalle de taille $T = \alpha m^2 2^{2n}$ inclus dans $[0, 3^{m2^n}]$, il existe un entier premier p qui ne divise pas S, car la densité des nombres premiers dans un tel intervalle est $\Omega(1/(m2^n))$.
On notera les polynômes successifs dans l'expression de S, q_i après un opérateur $\forall$ ou $\exists$ ($0 \leqslant i \leqslant n$), et $\ell_{i,j}$ après un opérateur L ($1 \leqslant j \leqslant i \leqslant n$), c'est-à-dire :

- $q_n(x_1,\dots,x_n) = Q_\psi(x_1,\dots,x_n)$;
- $\ell_{i,i}(x_1,\dots,x_i) = L_i(q_i)(x_1,\dots,x_i)$;
- $\ell_{i,j}(x_1,\dots,x_i) = L_j(\ell_{i,j+1})(x_1,\dots,x_i)$ pour $j < i$;
- si $i < n$ est pair alors $q_i(x_1,\dots,x_i) = \exists_{i+1}(\ell_{i+1,1})(x_1,\dots,x_i)$;
- si $i < n$ est impair alors $q_i(x_1,\dots,x_i) = \forall_{i+1}(\ell_{i+1,1})(x_1,\dots,x_i)$.

On a donc $q_0 = S$; $\ell_{1,1}(x_1) = L_1\forall_2 L_1 L_2 \exists_3 \dots L_n(Q_\psi)$; $q_1(x_1) = \forall_2 L_1 L_2 \exists_3 \dots L_n(Q_\psi)$, etc. (cf. figure 10.2).

$$S = \underbrace{\exists_1}_{q_0} \underbrace{L_1}_{\ell_{1,1}} \underbrace{\forall_2}_{q_1} \underbrace{L_1}_{\ell_{2,1}} \underbrace{L_2}_{\ell_{2,2}} \underbrace{\exists_3}_{q_2} \cdots \underbrace{\forall_n}_{q_{n-1}} \underbrace{L_1}_{\ell_{n,1}} \cdots \underbrace{L_n}_{\ell_{n,n}} \underbrace{Q_\psi}_{q_n}$$

FIGURE 10.2 – Polynômes successifs dans l'arithmétisation de φ.

Pour se repérer, on remarquera que pour tout i, q_i et $L_{i,j}$ ont i variables $(x_1,\dots,x_i)$.

Protocole En suivant les définitions ci-dessus des q_i et $\ell_{i,j}$, voici le protocole pour tester si $S \neq 0$. Nous en donnons d'abord une description informelle. Le prouveur envoie successivement les coefficients des polynômes q_0, $\ell_{1,1}$, q_1, $\ell_{2,1}\dots$ dans lesquels toutes les variables, sauf une, sont remplacées par des valeurs aléatoires choisies par le vérificateur : ainsi, le prouveur ne manipule que des polynômes univariés de degré 1 (ce faible degré est obtenu grâce à l'opérateur de linéarisation), dont il peut aisément donner les deux coefficients. Remplacer les variables par des valeurs aléatoires permet toujours au vérificateur de tester l'égalité de polynômes grâce au lemme 10-M. Entre chaque envoi des coefficients d'un polynôme, le vérificateur vérifie que la définition de ce polynôme est respectée et renvoie pour la suite la nouvelle valeur aléatoire qu'il a attribuée à la variable non encore évaluée. Plus précisément, à chaque étape, une seule variable x_j n'a pas de valeur attribuée, toutes les autres $(x_1,\dots,x_{j-1},x_{j+1},\dots,x_i)$ possèdent une valeur $(r_1,\dots,r_{j-1},r_{j+1},\dots,r_i)$; voici alors ce qui se passe :

- lorsque le prouveur doit convaincre que

$$\ell_{i,i}(r_1,\dots,r_{j-1},x_j,r_{j+1},\dots,r_i) = L_i(q_i)(r_1,\dots,r_{j-1},x_j,r_{j+1},\dots,r_i),$$

  il envoie les deux coefficients du polynôme univarié

$$\ell_{i,i}(r_1,\dots,r_{j-1},x_j,r_{j+1},\dots,r_i) ;$$

le vérificateur, qui dispose déjà des coefficients de q_i, vérifie que

$$\ell_{i,i}(r_1,\ldots,r_{j-1},x_j,r_{j+1},\ldots,r_i) = \\ (1-x_j)q_i(r_1,\ldots,r_{j-1},0,r_{j+1},\ldots,r_i)+x_j q_i(r_1,\ldots,r_{j-1},1,r_{j+1},\ldots,r_i)$$

en attribuant une nouvelle valeur aléatoire r_j à la variable x_j et en utilisant le lemme 10-M, puis il envoie cette valeur r_j au prouveur ;

– ils procèdent de la même façon pour tous les autres cas en suivant les relations définissant les polynômes q_i et $\ell_{i,j}$;

– à la fin, le vérificateur vérifie que l'évaluation de q_n est bien égale à l'évaluation de Q_ψ sur les valeurs aléatoires choisies $(r_1,\ldots,r_n)$.

Néanmoins l'ordre d'évaluation des variables, complètement occulté dans cette discussion, complique singulièrement la description complète du protocole. Cette discussion informelle permet tout de même de mieux appréhender le fonctionnement du protocole suivant, dans lequel les polynômes échangés n'ont qu'une variable et sont de degré au plus 1, donc peuvent être codés par la liste de leurs (deux) coefficients. Le prouveur envoie d'abord un entier premier p et tous les calculs s'effectueront sur $\mathbb{F}_p$. Il faut que p soit de taille polynomiale, mais assez grand pour que le prouveur ne puisse pas convaincre avec grande probabilité le vérificateur d'accepter si $S=0$. Nous verrons qu'il suffit de prendre p dans l'intervalle $[D,D+T]$ où $D=3(n(n+3)/2+m)$ et où $T=\alpha m^2 2^{2n}$ est défini ci-dessus. Les éléments $r_1,\ldots,r_n\in\mathbb{F}_p$ sont des éléments aléatoires tirés au fur et à mesure comme indiqué dans le protocole, où un nouveau tirage d'un r_i existant remplace l'ancienne valeur.

– P envoie un entier premier $p\in[D,D+T]$ (où D et T sont définis ci-dessus), et un entier $q'_0\in\mathbb{F}_p$ censé être égal à q_0 ;
si p n'est pas premier ou si $q'_0=0$ (sur $\mathbb{F}_p$), alors V rejette.

– P envoie un polynôme $\ell'_{1,1}(x_1)$ censé être $\ell_{1,1}(x_1)$ (et vérifiant donc $\exists_1\ell_{1,1}=q_0$) ;
si $q'_0\neq\ell'_{1,1}(0)+\ell'_{1,1}(1)$, V rejette.

– P envoie un polynôme $q'_1(x_1)$ censé être $q_1(x_1)$;
V choisit $r_1\in\mathbb{F}_p$ au hasard et vérifie que

$$\ell'_{1,1}(r_1)=(1-r_1)q'_1(0)+r_1q'_1(1)\ :$$

V rejette si ce n'est pas le cas, sinon il envoie r_1.

Le protocole continue alors ainsi pour i de 2 à n :

– P envoie un polynôme $\ell'_{i,1}(x_i)$ censé être

$$\ell_{i,1}(r_1,\dots,r_{i-1},x_i)\,;$$

V vérifie que

$$q'_{i-1}(r_{i-1}) = \ell'_{i,1}(0)+\ell'_{i,1}(1) \text{ (si } i \text{ est impair) ou}$$
$$q'_{i-1}(r_{i-1}) = \ell'_{i,1}(0)\times\ell'_{i,1}(1) \text{ (si } i \text{ est pair) :}$$

V rejette si ce n'est pas le cas, sinon il envoie $r_i \in \mathbb{F}_p$ choisi au hasard.

– P envoie un polynôme $\ell'_{i,2}(x_1)$ censé être $\ell_{i,2}(x_1,r_2,\dots,r_i)$;
V vérifie que

$$\ell'_{i,1}(r_i) = (1-r_1)\ell'_{i,2}(0)+r_1\ell'_{i,2}(1)\ :$$

V rejette si ce n'est pas le cas, sinon il envoie $r_1 \in \mathbb{F}_p$ choisi au hasard.

Et pour j de 3 à i :

◦ P envoie un polynôme $\ell'_{i,j}(x_{j-1})$ censé être

$$\ell_{i,j}(r_1,\dots,r_{j-2},x_{j-1},r_j,\dots,r_i)\,;$$

V vérifie que

$$\ell'_{i,j-1}(r_{j-2}) = (1-r_{j-1})\ell'_{i,j}(0)+r_{j-1}\ell'_{i,j}(1)\ :$$

V rejette si ce n'est pas le cas, sinon il envoie $r_{j-1}\in\mathbb{F}_p$ choisi au hasard. *(fin de la boucle sur j)*

– P envoie un polynôme $q'_i(x_i)$ censé être $q_i(r_1,\dots,r_{i-1},x_i)$;
V vérifie que

$$\ell'_{i,i}(r_{i-1}) = (1-r_i)q'_i(0)+r_iq'_i(1)\ :$$

V rejette si ce n'est pas le cas, sinon il envoie $r_i \in \mathbb{F}_p$ choisi au hasard.

Le dernier test du protocole est alors le suivant :

– V accepte ssi

$$q'_n(r_n) = q_n(r_1,\dots,r_n) = Q_\psi(r_1,\dots,r_n).$$

Les messages échangés au début du protocole sont donc :

P	p, q'_0		$\ell'_{1,1}(x_1)$		$q'_1(x_1)$		$\ell'_{2,1}(x_2)$		$\ell'_{2,2}(x_1)$								
V						r_1		r_2		r_1							

P	$q'_2(x_2)$		$\ell'_{3,1}(x_3)$		$\ell'_{3,2}(x_1)$		$\ell'_{3,3}(x_2)$		$q'_3(x_3)$		$\ldots$						
V		r_2		r_3		r_1		r_2		r_3	$\ldots$						

Analyse du protocole Si $\varphi \in \texttt{QBF}$ alors il existe $p \in [D, D+T]$ premier qui ne divise pas S (c'est-à-dire $S \neq 0$ sur $\mathbb{F}_p$). Il suffit donc au prouveur d'envoyer un tel entier p et ensuite les « bons » polynômes $q'_i = q_i$ et $\ell'_{i,j} = \ell_{i,j}$ pour convaincre le vérificateur avec probabilité 1.

Si en revanche $\varphi \notin \texttt{QBF}$, nous allons montrer qu'aucun prouveur ne convainc le vérificateur avec probabilité $> 1/3$. Nous allons donc majorer la probabilité qu'un prouveur P convainque V. Au dernier tour du protocole, on doit avoir

$$q'_n(r_n) = Q_\psi(r_1, \ldots, r_n) = q_n(r_1, \ldots, r_n)$$

pour r_n aléatoire inconnu de P au moment où il a envoyé q'_n. Puisque $q'_n(x_n)$ et $q_n(r_1, \ldots, r_{n-1}, x_n)$ sont des polynômes à une variable de degré au plus m, cela signifie que $q'_n(x_n) = q_n(r_1, \ldots, r_{n-1}, x_n)$ avec probabilité $\geqslant 1 - m/p$ par le lemme 10-M.

Raisonnons maintenant par récurrence en suivant le protocole en sens inverse, pour i variant de n à 2.

- Si $q'_i(x_i) = q_i(r_1, \ldots, r_{i-1}, x_i)$ alors le dernier test de la boucle i garantit que $\ell'_{i,i}(r_{i-1}) = \ell_{i,i}(r_1, \ldots, r_i)$ par définition de $\ell_{i,i}$. Puisque r_{i-1} est choisi au hasard et inconnu de P au moment où il a envoyé $\ell'_{i,i}$, le lemme 10-M implique que $\ell'_{i,i}(x_{i-1}) = \ell_{i,i}(r_1, \ldots, r_{i-2}, x_{i-1}, r_i)$ avec probabilité $\geqslant 1 - 1/p$.
- Pour j variant de i à 3 : si $\ell'_{i,j}(x_{j-1}) = \ell_{i,j}(r_1, \ldots, r_{j-2}, x_{j-1}, r_j, \ldots, r_i)$, alors le test de V garantit que $\ell'_{i,j-1}(r_{j-2}) = \ell_{i,j-1}(r_1, \ldots, r_i)$ par définition de $\ell_{i,j-1}$. Puisque r_{j-2} est choisi au hasard et inconnu de P au moment où il a envoyé $\ell'_{i,j-1}$, le lemme 10-M implique que $\ell'_{i,j-1}(x_{j-2}) = \ell_{i,j-1}(r_1, \ldots, r_{j-3}, x_{j-2}, r_{j-1}, \ldots, r_i)$ avec probabilité $\geqslant 1 - 1/p$.
- Si $\ell'_{i,2}(x_1) = \ell_{i,2}(x_1, r_2, \ldots, r_i)$ alors le test précédant la boucle j assure que l'on a $\ell'_{i,1}(r_i) = \ell_{i,1}(r_1, \ldots, r_i)$ par définition de $\ell_{i,1}$. Puisque r_i est choisi au hasard et inconnu de P au moment où il a envoyé $\ell'_{i,1}$, le lemme 10-M implique que $\ell'_{i,1}(x_i) = \ell_{i,1}(r_1, \ldots, r_{i-1}, x_i)$ avec probabilité $\geqslant 1 - 1/p$.
- Si $\ell'_{i,1}(x_i) = \ell_{i,1}(r_1, \ldots, r_{i-1}, x_i)$ alors le premier test de la boucle i assure que $q'_{i-1}(r_1, \ldots, r_{i-1}) = q_{i-1}(r_1, \ldots, r_{i-1})$ par définition de q_{i-1}. Puisque r_{i-1} est choisi au hasard et inconnu de P au moment où il a envoyé q'_{i-1}, le lemme 10-M implique que $q'_{i-1}(x_{i-1}) = q_{i-1}(r_1, \ldots, r_{i-2}, x_{i-1})$ avec probabilité $\geqslant 1 - 1/p$.

– Enfin, si $q'_1(x_1) = q_1(x_1)$ alors le test précédant la boucle i assure que $\ell'_{1,1}(x_1) = \ell_{1,1}(x_1)$ avec probabilité $\geqslant 1 - 1/p$; puis si c'est le cas, le test précédent assure que $q'_0 = q_0$; et enfin le premier test garantit que $q_0 \neq 0$.

À chacune des étapes décrites ci-dessus, il y a une probabilité « d'erreur » d'au plus $1/p$, sauf à la première (correspondant au dernier test de V) où c'est m/p. Puisqu'il y a $n(n+3)/2$ étapes pouvant donner des erreurs, la probabilité que l'un des polynômes q'_i ou $\ell'_{i,j}$ ne soit pas celui qu'il est censé être est au plus

$$\frac{n(n+3)/2 + m}{p} \leqslant 1/3.$$

Dans le cas où tous les polynômes sont « les bons » alors $q'_0 = 0$ puisque $\varphi \notin \mathtt{QBF}$, donc V rejette. On en déduit que P convainc V avec probabilité au plus 1/3. □

10-N Corollaire

Tout langage de IP est reconnu par un protocole dans lequel le prouveur convainc le vérificateur avec probabilité 1 quand le mot est dans le langage.

Démonstration Soit $L \in \mathsf{IP} = \mathsf{PSPACE}$: $L \leqslant^{\mathsf{p}}_{\mathsf{m}} \mathtt{QBF}$ donc il existe f calculable en temps polynomial tel que $x \in L \iff f(x) \in \mathtt{QBF}$. Le protocole consiste donc à faire tourner le protocole pour QBF du lemme 10-J sur $f(x)$. □

✎ **10-O Exercice**

Montrer en revanche que si dans un protocole IP pour un langage L, tout prouveur convainc le vérificateur avec probabilité 0 lorsque $x \notin L$ (c'est-à-dire que V rejette avec probabilité 1), alors $L \in \mathsf{NP}$.

Puisque $\mathsf{PP} \subseteq \mathsf{PSPACE}$, il est évident que PP a des protocoles IP. Cependant, en procédant comme pour le lemme 10-J mais de manière plus simple car il n'est plus besoin de s'encombrer des opérateurs de linéarisation et de produit, on peut obtenir le résultat plus fort suivant sur le problème PP-complet Maj3SAT (cf. section 9.2.3), qui nous servira au chapitre suivant pour montrer une borne inférieure non uniforme sur la classe PP.

10-P Proposition

Le problème Maj3SAT possède un protocole IP pour lequel il existe un prouveur valide dans $\mathsf{FP}^{\mathsf{PP}}$ (la classe des fonctions calculables en temps polynomial grâce à un oracle PP).

✎ **10-Q** **Exercice**

Montrer la proposition précédente.

Non-relativisation

Le surprenant théorème 10-G a été d'autant mieux accueilli qu'il ne relativise pas comme le montre le résultat suivant de Fortnow et Sipser [FS88]. Dans un protocole IP, c'est évidemment le vérificateur qui porte l'oracle, car celui-ci ne servirait pas au prouveur qui est déjà tout-puissant.

10-R **Proposition (Fortnow et Sipser, 1988)**

Il existe un oracle A tel que $\mathsf{coNP}^A \not\subseteq \mathsf{IP}^A$, et donc $\mathsf{IP}^A \neq \mathsf{PSPACE}^A$.

Idée de la démonstration On construit par diagonalisation un oracle A tel qu'aucun protocole IP^A ne peut décider le langage $\{1^n \mid A \text{ contient tous les mots de longueur } n\}$, alors que ce langage est clairement dans coNP^A.

Nous utilisons pour cela le fait que le vérificateur est probabiliste et ne peut donc pas détecter un changement « négligeable », contrairement à une machine coNP : modifier un mot de A qui n'est demandé qu'avec faible probabilité ne change pas la réponse du vérificateur.

Démonstration Si A est un langage, on note

$$L(A) = \{1^n \mid A \text{ contient tous les mots de longueur } n\}.$$

Pour tout oracle A, on a donc $L(A) \in \mathsf{coNP}^A$ car il suffit, sur l'entrée 1^n de vérifier de manière non déterministe et grâce à l'oracle que pour tout mot x de taille n, $x \in A$. Nous allons construire A de sorte que $L(A) \notin \mathsf{IP}^A$.

On note $(V_i)_{i\geqslant 1}$ une énumération des machines de Turing déterministes (voir la définition 7-A et la remarque 3-AL) où V_i fonctionne en temps n^i lorsque n désigne la taille de l'entrée x. On considérera les protocoles à $2n^i$ tours dans lesquels V_i est le vérificateur et on notera r (mot de taille n^i) les bits aléatoires donnés à V_i.

Dans un protocole à $2n^i$ tours avec un prouveur P, sur l'entrée 1^n le vérificateur V_i fonctionnant en temps n^i ne peut demander à son oracle des mots de taille supérieure à n^i. On définit une suite $(u_i)_{i\geqslant 1}$ par $u_1 = 18$ et $u_{i+1} = 1+u_i^i$, de sorte que sur l'entrée 1^{u_i} le vérificateur V_i ne demande à son oracle que des mots de taille strictement inférieure à u_{i+1}. On a choisi la première valeur $u_1 = 18$ pour que $u_i^{2i}/2^{u_i} < 1/3$ pour tout $i \geqslant 1$ (ce choix deviendra clair ci-dessous), ce que l'on vérifie aisément par récurrence en passant au logarithme.

On définit le langage A par étapes en commençant par $A_1 = \{0,1\}^\star$ (au départ il contient tous les mots) et en retirant des mots au fur et à mesure. À l'étape $i \geqslant 1$, on considère les protocoles entre un prouveur P et V_i avec oracle A_i sur l'entrée 1^{u_i} :

– si aucun prouveur P ne convainc $V_i^{A_i}$ d'accepter 1^{u_i} avec probabilité $\geqslant 2/3$, alors $A_{i+1} = A_i$ (on ne retire aucun mot) ;

– en revanche, s'il existe un prouveur P qui convainc $V_i^{A_i}$ d'accepter 1^{u_i} avec probabilité $\geqslant 2/3$, alors au cours du protocole, pour des bits aléatoires r fixés, V_i demande à son oracle A_i au plus u_i^{2i} mots (car il y a u_i^i tours de V_i lors desquels son temps de calcul est au plus u_i^i). Il existe donc un mot x_0 de taille u_i dont la probabilité d'être demandé à son oracle par $V_i^{A_i}$ est au plus $u_i^{2i}/2^{u_i}$ (où la probabilité est prise sur tous les choix de r possibles). On définit alors $A_{i+1} = A_i \setminus \{x_0\}$ (c'est-à-dire qu'on retire le mot x_0).

Le langage A est enfin $A = \cap_{i \geqslant 1} A_i$. On remarque que la définition de la suite (u_i) implique que pour tout i, $V_i^{A_{i+1}}$ et V_i^A ont le même comportement sur 1^{u_i}.

Par définition, si aucun prouveur P ne convainc $V_i^{A_i}$ d'accepter 1^{u_i} avec probabilité $\geqslant 2/3$, alors $A_{i+1} = A_i$ donc aucun prouveur P ne convainc V_i^A d'accepter 1^{u_i}. Mais A contient tous les mots de taille u_i, donc $1^{u_i} \in L(A)$ et V_i^A ne décide correctement $L(A)$ dans aucun protocole à $2n^i$ tours.

De même, s'il existe un prouveur P qui convainc $V_i^{A_i}$ d'accepter 1^{u_i} avec probabilité $\geqslant 2/3$, alors par rapport à A_i, dans A_{i+1} seul le mot x_0 manque et celui-ci n'affecte qu'une proportion $u_i^{2i}/2^{u_i} < 1/3$ des choix aléatoires de $V_i^{A_i}$. Donc P convainc encore $V_i^{A_{i+1}}$ d'accepter 1^{u_i} avec probabilité $> 2/3 - 1/3 = 1/3$, alors que $1^{u_i} \notin L(A)$ puisque A ne contient plus x_0.

En conclusion, aucun vérificateur V_i polynomial avec oracle A ne reconnaît le langage $L(A)$ dans un protocole à $2n^i$ tours. Donc $L(A) \notin \mathsf{IP}^A$. □

Ce résultat a renforcé l'attention apportée aux protocoles interactifs qui permettent ainsi de dépasser la barrière de la relativisation. Cependant, une autre barrière, « l'arithmétisation » généralisant la relativisation, a depuis été trouvée par Aaronson et Wigderson [AW08] en remarquant que les protocoles comme celui pour QBF utilisent massivement l'arithmétisation (c'est-à-dire la transformation d'objets booléens en polynômes pour utiliser des outils algébriques) mais que cette technique ne suffit pas pour résoudre les grandes questions ouvertes comme « P = NP ? ».

10.2 Les classes Arthur-Merlin

Une sous-classe des protocoles IP que nous avons vus consiste à rendre publics les bits aléatoires du vérificateur. Le prouveur dans cette variante a reçu le nom de Merlin (magicien tout puissant) et le vérificateur celui d'Arthur (roi intelligent que Merlin cherche à convaincre).

10.2.1 Définitions

Contrairement à IP, le vérificateur ne peut pas cacher ses bits aléatoires au prouveur. Néanmoins, ils ne sont révélés qu'au fur et à mesure de leur utilisation et non tous d'un coup en début de protocole. Dans un protocole IP, seuls les bits aléatoires privés du vérificateur empêchaient au prouveur tout-puissant de calculer lui-même les questions du vérificateur. Dans un protocole Arthur-Merlin, le vérificateur, en révélant ses bits aléatoires, n'a donc pas besoin d'envoyer sa question au prouveur.

10-S Définition

– Un protocole Arthur-Merlin est un protocole IP dans lequel le vérificateur (appelé Arthur et noté A) envoie au prouveur (appelé Merlin et noté M) seulement des bits aléatoires à chaque tour, et effectue son calcul de manière déterministe à la fin du protocole en fonction de l'entrée et des messages échangés (c'est-à-dire les bits aléatoires qu'il a envoyés et les réponses de Merlin).
En d'autres termes, sur l'entrée x le protocole consiste en un échange

$$r_1, z_1, r_2, z_2, \ldots, r_k, z_k$$

(s'il y a $2k$ tours où A commence, tous les autres choix étant possibles) où les r_i sont des mots aléatoires de taille polynomiale (les « questions » d'Arthur) et où le mot $z_i = M(x, r_1, r_2, \ldots, r_{i-1})$ est la i-ème réponse de Merlin. La dernière étape est la vérification par A : l'entrée x est acceptée ssi $A(x, r_1, z_1, \ldots, r_k, z_k) = 1$. On rappelle que A est une machine déterministe fonctionnant en temps polynomial en $|x|$ et que M est une fonction sans contrainte de calculabilité.

– Pour $k \geqslant 2$, la classe $\mathsf{AM}[k]$ est l'ensemble des langages L possédant un protocole Arthur-Merlin à k tours où Arthur commence, c'est-à-dire tels qu'il existe un polynôme $p(n)$ et un vérificateur A fonctionnant en temps $p(|x|)$, satisfaisant :

$$\begin{cases} x \in L \implies & \text{il existe un prouveur } M \text{ tel que} \\ & \Pr_{r_i \in \{0,1\}^{p(|x|)}}(A(x, r_1, z_1, \ldots) = 1) \geqslant 2/3 \\ x \notin L \implies & \text{pour tout prouveur } M, \\ & \Pr_{r_i \in \{0,1\}^{p(|x|)}}(A(x, r_1, z_1, \ldots) = 1) \leqslant 1/3, \end{cases}$$

où $A(x, r_1, z_1, \ldots)$ signifie

$$A(x, r_1, z_1, \ldots, r_{k/2}, z_{k/2}) \text{ si } k \text{ est pair,}$$
$$\text{et } A(x, r_1, z_1, \ldots, r_{(k+1)/2}) \text{ sinon.}$$

– La classe $\mathsf{MA}[k]$ est définie de la même façon mais M débute le protocole : l'échange est donc $(z_1, r_1, z_2, \ldots, z_{k/2}, r_{k/2})$ si k est pair et $(z_1, r_1, z_2, \ldots, z_{(k+1)/2})$ sinon.

– Contrairement à IP où l'on s'intéressait aux protocoles ayant un nombre polynomial de tours, nous étudierons principalement les protocoles Arthur-Merlin avec un nombre constant de tours. On note alors $\mathsf{AM} = \mathsf{AM}[2]$ et $\mathsf{MA} = \mathsf{MA}[2]$ pour une raison qui sera claire avec le théorème 10-W.

10-T **Remarques**

– On trouve aussi la notation alternative AMA pour AM[3], MAM pour MA[3], AMAM pour AM[4], etc. Nous utiliserons la plus commode des deux notations selon le contexte.

– Lorsque le protocole se termine par un tour d'Arthur, comme par exemple dans MA ou AMA, etc., le dernier calcul d'Arthur est probabiliste puisqu'il tire des bits aléatoires au début de son tour (ce qui n'est pas le cas pour AM, MAM, etc., où le calcul final d'Arthur est déterministe). Par exemple dans MA, Merlin envoie une preuve et Arthur la vérifie de manière probabiliste : on peut donc voir MA comme une version probabiliste de NP.

Les deux classes $\mathsf{AM} = \mathsf{AM}[2]$ et $\mathsf{MA} = \mathsf{MA}[2]$ sont importantes (cf. théorème 10-W), c'est pourquoi nous en donnons une caractérisation spécifique qui découle directement de la définition.

10-U **Proposition**

– Un langage L est dans AM s'il existe un langage $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que

$$\begin{cases} x \in L \implies & \Pr_{r\in\{0,1\}^{p(|x|)}}(\exists y \in \{0,1\}^{p(|x|)}\ (x,r,y) \in B) \geqslant 2/3, \\ x \notin L \implies & \Pr_{r\in\{0,1\}^{p(|x|)}}(\exists y \in \{0,1\}^{p(|x|)}\ (x,r,y) \in B) \leqslant 1/3. \end{cases}$$

– Un langage L est dans MA s'il existe un langage $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que

$$\begin{cases} x \in L \implies & \exists y \in \{0,1\}^{p(|x|)} \Pr_{r\in\{0,1\}^{p(|x|)}}((x,y,r) \in B) \geqslant 2/3, \\ x \notin L \implies & \forall y \in \{0,1\}^{p(|x|)} \Pr_{r\in\{0,1\}^{p(|x|)}}((x,y,r) \in B) \leqslant 1/3. \end{cases}$$

Les preuves qui suivent concerneront souvent des inversions de « quantificateurs » Pr et $\exists$, il est donc utile de prendre le temps de réfléchir à leur signification et notamment aux dépendances entre y et r.

✎ **10-V** **Exercice**

Montrer que MA ⊆ PP.

10.2.2 Deux petits tours et puis s'en vont

De manière surprenante, on peut montrer que deux tours suffisent dans les protocoles Arthur-Merlin. Il s'agit d'un résultat de Babai et Moran [BM88].

10-W **Théorème (Babai et Moran, 1988)**

Pour toute constante $k \geqslant 2$, $\mathsf{AM}[k] = \mathsf{AM}[2]$ (c'est-à-dire AM).

Puisque évidemment $\mathsf{MA}[k] \subseteq \mathsf{AM}[k+1]$, on obtient le corollaire suivant.

10-X **Corollaire**

Pour un nombre constant de tours il n'y a que deux classes Arthur-Merlin, MA et AM, vérifiant MA ⊆ AM.

Idée de la démonstration du théorème 10-W Nous allons transformer tout protocole $\mathsf{AM}[2k]$ en un protocole équivalent $\mathsf{AM}[2k-2]$ en échangeant les tours $2k-2$ et $2k-1$, ce qui transforme AM…AMAM en AM…AAMM = AM…AM (puisque deux tours A successifs peuvent être réunis en un seul, et de même pour M). Pour cela, il s'agit tout d'abord de réduire exponentiellement la probabilité d'erreur sans toucher aux $2k-2$ premiers tours : en effectuant une « répétition parallèle » lors des deux derniers tours (Arthur envoie plusieurs questions au tour $2k-1$ et Merlin renvoie toutes les réponses au tour $2k$), on réduit l'erreur suffisamment pour qu'Arthur puisse envoyer r_k avant que Merlin n'envoie z_{k-1} et z_k.

Démonstration du théorème 10-W Pour $k \geqslant 2$, nous allons montrer que $\mathsf{AM}[2k] \subseteq \mathsf{AM}[2k-2]$. Soit $L \in \mathsf{AM}[2k]$ possédant un protocole à $2k$ tours où sur l'entrée x, les messages échangés sont $r_1, z_1, \ldots, r_k, z_k$ (r_i dénote les mots aléatoires envoyés par Arthur et z_i désigne les réponses de Merlin). On interprète z_i comme une fonction de $r_1, \ldots, r_i$ et on écrira $z_i(.)$ pour souligner le fait qu'il s'agit d'une fonction. Néanmoins, la dernière réponse z_k de Merlin vient après le choix de $r_1, \ldots, r_k$ et en la « rentrant » dans la probabilité, il suffit alors de quantifier sur un mot plutôt qu'une fonction :

$$\begin{cases} x \in L \implies & \exists z_1(.), \ldots, z_{k-1}(.) \Pr_{r_1,\ldots,r_k}\big(\exists z_k\, (x, r_1, z_1(r_1), r_2, z_2(r_1, r_2), \ldots, \\ & \qquad z_{k-1}(r_1, \ldots, r_{k-1}), r_k, z_k) \in B\big) \geqslant 2/3 \\ x \notin L \implies & \forall z_1(.), \ldots, z_{k-1}(.) \Pr_{r_1,\ldots,r_k}\big(\exists z_k\, (x, r_1, z_1(r_1), r_2, z_2(r_1, r_2), \ldots, \\ & \qquad z_{k-1}(r_1, \ldots, r_{k-1}), r_k, z_k) \in B\big) \leqslant 1/3, \end{cases}$$

où $B \in \mathsf{P}$. C'est à partir de cette nouvelle caractérisation de la classe $\mathsf{AM}[2k]$ que nous allons travailler.

Réduction de la probabilité d'erreur Nous allons réduire la probabilité d'erreur de ce protocole sans toucher aux $2k-2$ premières étapes : dans le nouveau protocole, les $2k-2$ premiers tours sont identiques. Au tour $2k-1$, au lieu d'envoyer seulement r_k, A envoie N mots aléatoires indépendants de taille $|r_k|$, que l'on notera $r_k^1, \ldots, r_k^N$. M répond alors par N mots de taille $|z_k|$ que l'on notera $z_k^1, \ldots, z_k^N$. A accepte ssi pour la majorité des $i \in [1,N]$, $(x, r_1, z_1, \ldots, z_{k-1}, r_k^i, z_k^i) \in B$. Pour simplifier les notations, on note $r_k' = (r_k^1, \ldots, r_k^N)$, $z_k' = (z_k^1, \ldots, z_k^N)$ et

$$C = \{(x, r_1, z_1, \ldots, z_{k-1}, r_k', z_k') \mid \text{ pour la majorité des } i, (x, r_1, z_1, \ldots, z_{k-1}, r_k^i, z_k^i) \in B\}$$

qui sera la vérification finale de notre nouveau protocole. En résumé, le protocole consiste donc en l'envoi de r_1 par A, z_1 par M, …, z_{k-1} par M, puis r_k' par A et enfin z_k' par M.
Majorons la probabilité d'erreur de ce nouveau protocole (l'analyse est la même que pour BPP à la proposition 6-F). Soit X la variable aléatoire égale au nombre de $i \in [1,N]$ tels que $\forall z_k^i\ (x, r_1, z_1, \ldots, z_{k-1}, r_k^i, z_k^i) \notin B$. Alors $X = \sum_{i=1}^N X_i$ où $X_i \in \{0,1\}$ vaut 0 ssi $\exists z_k^i\ (x, r_1, z_1, \ldots, z_{k-1}, r_k^i, z_k^i) \in B$. Les X_i sont indépendantes. Si $x \in L$, alors X_i vaut 0 avec probabilité $\geqslant 2/3$. Par les bornes de Chernoff (la version faible de la proposition A-M suffit), $\Pr_{r_1,\ldots,r_{k-1},r_k'}(X \geqslant N/2) \leqslant \alpha^N$ où $\alpha = 4/(3\sqrt{2}) < 1$. En choisissant $N = (|z_{k-1}|+2)/(-\log \alpha)$, la probabilité d'erreur du nouveau protocole est donc inférieure à $2^{-|z_{k-1}|-2}$. Le calcul est exactement le même dans le cas où $x \notin L$ et mène à la même probabilité d'erreur. En résumé, on a :

$$\begin{cases} x \in L \implies & \exists z_1(.), \ldots, z_{k-1}(.) \\ & \Pr_{r_1,\ldots,r_{k-1},r_k'}\left(\exists z_k'\ (x, r_1, z_1(r_1), \ldots, r_k', z_k') \in C\right) \geqslant 1 - 2^{-|z_{k-1}|-2} \\ x \notin L \implies & \forall z_1(.), \ldots, z_{k-1}(.) \\ & \Pr_{r_1,\ldots,r_{k-1},r_k'}\left(\exists z_k'\ (x, r_1, z_1(r_1), \ldots, r_k', z_k') \in C\right) \leqslant 2^{-|z_{k-1}|-2} \end{cases}$$

Inversion des deux dernières étapes Bien sûr, dans le premier cas ($x \in L$) on peut « rentrer » le z_{k-1} : en effet, il y a moins de contraintes sur z_{k-1} car on peut maintenant le choisir en fonction de r_k'. On en déduit que si $x \in L$ alors

$$\exists z_1(.), \ldots, z_{k-2}(.) \quad \Pr_{r_1,\ldots,r_{k-1},r_k'}\left(\exists z_{k-1}, z_k'\ (x, r_1, z_1(r_1), \ldots, r_{k-1}, z_{k-1}, r_k', z_k') \in C\right) \geqslant 1 - 2^{-|z_{k-1}|-2}.$$

D'un autre côté, si $x \notin L$ alors

$$\forall z_1(.), \ldots, z_{k-2}(.) \quad \Pr_{r_1,\ldots,r_{k-1},r_k'}\left(\exists z_{k-1}, z_k'\ (x, z_1(r_1), \ldots, r_{k-1}, z_{k-1}, r_k', z_k') \in C\right)$$
$$\leqslant \sum_{z_{k-1}} \Pr_{r_1,\ldots,r_{k-1},r_k'}\left(\exists z_k'\ (x, r_1, z_1(r_1), \ldots, r_k', z_k') \in C\right) \leqslant \sum_{z_{k-1}} 2^{-|z_{k-1}|-2} = 1/4.$$

D'où :

$$\begin{cases} x \in L \implies & \exists z_1(.), \ldots, z_{k-2}(.) \\ & \Pr_{r_1,\ldots,r_{k-1},r'_k}\left(\exists z_{k-1}, z'_k\,(x, r_1, z_1(r_1), \ldots, r_{k-1}, z_{k-1}, r'_k, z'_k) \in C\right) \geqslant 2/3 \\ x \notin L \implies & \forall z_1(.), \ldots, z_{k-2}(.) \\ & \Pr_{r_1,\ldots,r_{k-1},r'_k}\left(\exists z_{k-1}, z'_k\,(x, r_1, z_1(r_1), \ldots, r_{k-1}, z_{k-1}, r'_k, z'_k) \in C\right) \leqslant 1/3, \end{cases}$$

ce qui est la définition d'un protocole $\mathsf{AM}[2k-2]$ où A envoie r_1, M envoie z_1, …, M envoie z_{k-2}, A envoie r_{k-1}, r'_k et M envoie enfin z_{k-1}, z'_k. Donc $L \in \mathsf{AM}[2k-2]$. □

10-Y **Remarques**

– La technique de la « répétition parallèle » vue dans la preuve précédente permet de réduire la probabilité d'erreur dans tout protocole AM ou MA jusqu'à 2^{-n^k} pour toute constante $k > 0$.

– On remarque en examinant la preuve que si le protocole $\mathsf{AM}[k]$ de départ ne fait aucune erreur dans le cas où $x \in L$ (c'est-à-dire que A accepte avec probabilité 1 dans ce cas), alors il en est de même pour le protocole AM final.

– Plus généralement, on notera $\mathsf{AM}[t(n)]$ la classe des langages reconnus par un protocole Arthur-Merlin à $t(n)$ tours. Si $t(n)$ est un polynôme et $t(n) \geqslant 2$, alors le théorème précédent est étendu sans difficulté à l'énoncé suivant : pour toute constante $k \geqslant 0$, $\mathsf{AM}[t(n)+k] = \mathsf{AM}[t(n)]$.

✎ **10-Z** **Exercice**

Montrer les différents points de la remarque précédente.

Ainsi, lorsqu'on parle de protocoles Arthur-Merlin ayant un nombre constant de tours, on parle en réalité seulement des classes MA ou AM.

10.2.3 Erreur d'un seul côté

La définition de MA ou AM est symétrique dans le sens où l'erreur maximale commise lorsque $x \in L$ ou lorsque $x \notin L$ est la même, 1/3. On a vu à la remarque 10-Y que l'on peut réduire cette erreur jusqu'à 2^{-n^k}. Nous allons voir que nous pouvons même empêcher toute erreur dans le cas où $x \in L$.

10-AA **Proposition**

Tout langage $L \in \mathsf{MA}$ est reconnu par un protocole MA qui ne fait aucune erreur lorsque $x \in L$, c'est-à-dire qu'Arthur accepte avec probabilité 1 lorsque $x \in L$ et avec probabilité $\leqslant 1/3$ lorsque $x \notin L$.

De même, tout langage $L \in \mathsf{AM}$ est reconnu par un protocole AM qui ne fait aucune erreur lorsque $x \in L$, c'est-à-dire qu'Arthur accepte avec probabilité 1 lorsque $x \in L$ et avec probabilité $\leqslant 1/3$ lorsque $x \notin L$.

Idée de la démonstration Comme au théorème 8-N ($\mathsf{BPP} \subseteq \Sigma_2^{\mathrm{p}}$), lorsque $x \in L$ la probabilité d'accepter est grande donc on peut recouvrir tout l'espace des choix aléatoires par un nombre polynomial de translations de l'ensemble des mots aléatoires amenant à une acceptation.

Dans le cas MA, Merlin fournit ces translations : Arthur n'a plus qu'à vérifier qu'un mot aléatoire r appartient à l'une des translations et accepte avec probabilité 1 si $x \in L$. En revanche, lorsque $x \notin L$ les translations ne couvrent qu'une petite partie de l'espace et Arthur rejette avec grande probabilité.

Dans le cas AM, on conçoit d'abord un protocole MAM pour que Merlin puisse fournir les translations, puis le protocole est transformé en AM grâce au théorème 10-W et la remarque 10-Y.

Démonstration Nous commençons par MA. Soit $L \in \mathsf{MA}$ reconnu par un protocole avec erreur maximale 2^{-n} (où n désigne la taille de l'entrée x) :

$$\begin{cases} x \in L \implies & \exists y \Pr_r((x,y,r) \in B) \geqslant 1 - 2^{-n} \\ x \notin L \implies & \forall y \Pr_r((x,y,r) \in B) \leqslant 2^{-n}. \end{cases}$$

Nous allons utiliser une technique similaire à la preuve du théorème 8-N ($\mathsf{BPP} \subseteq \Sigma_2^{\mathrm{p}}$). On note $R_{x,y}$ l'ensemble des mots r tels que $(x,y,r) \in B$. On a donc : si $x \in L$ alors $\exists y, |R_{x,y}| \geqslant (1-2^{-n})2^{|r|}$, et si $x \notin L$ alors $\forall y, |R_{x,y}| \leqslant 2^{|r|-n}$. Par le lemme 8-O, si $x \in L$ alors il existe $N = 1 + \lfloor |r|/n \rfloor$ mots $u_1, \ldots, u_N$ de taille $|r|$ tels que $\cup_i (R_{x,y} + u_i) = \{0,1\}^{|r|}$. Et bien sûr, si $x \notin L$ alors pour tous mots $u_1, \ldots, u_N$ de taille $|r|$, on a $|\cup_i (R_{x,y} + u_i)| \leqslant N 2^{|r|-n} < 1/3 \cdot 2^{|r|}$ (pour n suffisamment grand). Voici donc le nouveau protocole MA pour L :

- Merlin envoie $y, u_1, \ldots, u_N$;
- Arthur choisit v de taille $|r|$ au hasard et accepte ssi $v \in \cup_i (R_{x,y} + u_i)$, c'est-à-dire si $\exists i, (x,y,v+u_i) \in B$.

Si $x \in L$ alors Merlin peut envoyer le bon y et les bons mots u_i pour qu'Arthur accepte avec probabilité 1. Si $x \notin L$, alors quels que soient $y, u_1, \ldots, u_N$, Arthur rejettera avec probabilité au moins 2/3.

Nous passons maintenant au cas de AM, en partant d'un protocole avec probabilité d'erreur au plus 2^{-n} :

$$\begin{cases} x \in L \implies & \Pr_r(\exists y, (x,y,r) \in B) \geqslant 1 - 2^{-n} \\ x \notin L \implies & \Pr_r(\exists y, (x,y,r) \in B) \leqslant 2^{-n}. \end{cases}$$

Nous utilisons les mêmes notations que précédemment, et l'idée est similaire. Si $x \in L$ alors $|\cup_y R_{x,y}| \geqslant (1-2^{-n})2^{|r|}$ donc il existe des mots $u_1, \dots, u_N$ tels que $\cup_{i,y}(R_{x,y}+u_i) = \{0,1\}^{|r|}$. En revanche, si $x \notin L$ alors $|\cup_y R_{x,y}| \leqslant 2^{|r|-n}$ et donc pour tous mots $u_1, \dots, u_N$, $|\cup_{i,y}(R_{x,y}+u_i)| \leqslant N2^{|r|-n} < (1/3)2^{|r|}$ (pour n suffisamment grand).
Nous transformons le protocole AM en un protocole MAM comme suit :

- Merlin envoie $u_1, \dots, u_N$;
- Arthur envoie v ;
- Merlin envoie y ;
- Arthur vérifie que $v \in \cup_i(R_{x,y}+u_i)$, c'est-à-dire que $\exists i, (x,y,v+u_i) \in B$.

Le dernier tour est déterministe, nous avons donc un protocole MAM. Si $x \in L$ alors il existe des mots u_i tels que $\cup_i((\cup_y R_{x,y})+u_i) = \{0,1\}^{|r|}$. Au premier tour, Merlin envoie de tels mots u_i. Alors le mot v renvoyé par Arthur appartient à $(R_{x,y}+u_i)$ pour un certain y et un certain i : Merlin peut envoyer le y correspondant et Arthur accepte avec probabilité 1. Si $x \notin L$, alors quels que soient les mots u_i, $|\cup_{i,y}(R_{x,y}+u_i)| < (1/3)2^{|r|}$ donc la probabilité que $v \in \cup_i(R_{x,y}+u_i)$ est au plus $1/3$.
Grâce à la remarque 10-Y, ce nouveau protocole MAM peut être transformé en un protocole AM sans perdre la propriété d'accepter avec probabilité 1 quand l'entrée est dans le langage. □

Cela nous permet de placer MA et AM dans la hiérarchie polynomiale.

10-AB Corollaire

$$\mathsf{MA} \subseteq \Sigma_2^{\mathsf{p}} \qquad \text{et} \qquad \mathsf{AM} \subseteq \Pi_2^{\mathsf{p}}.$$

Démonstration Grâce à la proposition 10-AA, l'appartenance à un langage $L \in \mathsf{MA}$ s'écrit :

$$x \in L \iff \exists y \forall r (x,y,r) \in B$$

où $B \in \mathsf{P}$, c'est-à-dire que $L \in \Sigma_2^{\mathsf{p}}$. De même, l'appartenance à un langage $L \in \mathsf{AM}$ s'écrit :

$$x \in L \iff \forall r \exists y (x,y,r) \in B$$

où $B \in \mathsf{P}$, c'est-à-dire que $L \in \Pi_2^{\mathsf{p}}$. □

10.2.4 Bits aléatoires publics ou privés

La différence entre les protocoles Arthur-Merlin et les protocoles de type IP est que dans ces derniers, les bits aléatoires utilisés par le vérificateur sont privés, c'est-à-dire inconnus du prouveur. Ce n'est pas le cas de Merlin qui connaît les bits aléatoires d'Arthur au moment où il les tire : on dit alors qu'ils sont publics. Garder ses bits privés donne au vérificateur plus de latitude, et le protocole pour le problème coISO donné plus haut (section 10.1.2) utilise en effet de manière cruciale le fait que les bits aléatoires sont inconnus du prouveur.

Néanmoins, et aussi surprenant que cela puisse paraître, Goldwasser et Sipser [GS86] ont montré que cacher les bits aléatoires au prouveur (c'est-à-dire utiliser des bits privés) n'apporte pas de puissance supplémentaire. Nous n'avons introduit les classe Arthur-Merlin que pour un nombre constant de tours, mais nous utiliserons la généralisation évidente à un nombre polynomial $t(n)$ de tours en notant $\mathsf{AM}[t(n)]$. Nous avons tout d'abord besoin d'un lemme technique permettant de concevoir un protocole pour calculer une approximation la taille d'un ensemble [2].

10-AC Lemme

Soit $E \subseteq \{0,1\}^n$ un ensemble de mots de taille n sur l'alphabet $\{0,1\}$, que l'on verra comme des vecteurs sur l'espace vectoriel $(\mathbb{F}_2)^n$, et soit $m < n$ et $k \geqslant m$ deux entiers (avec $k \geqslant 3$). Dans ce qui suit, $A_1, \ldots, A_k$ désigneront des matrices aléatoires sur $\mathbb{F}_2$ de taille $m \times n$ (c'est-à-dire des applications linéaires de $\{0,1\}^n$ vers $\{0,1\}^m$) et $y_1, \ldots, y_{8k^2}$ désigneront $8k^2$ mots aléatoires de $\{0,1\}^m$ formant un ensemble noté Y. On note $A(E) = \cup_j A_j(E) \subseteq \{0,1\}^m$.

1. Si $|E| \geqslant 2^{m-3}$ alors la probabilité que $Y \cap A(E) \neq \emptyset$ est supérieure à $1 - 2^{-k}$ (grande probabilité d'intersection de Y par les images de E).
2. Pour tout $d > 0$, si $|E| \leqslant 2^m/d$ alors la probabilité que $Y \cap A(E) \neq \emptyset$ est inférieure à $8k^3/d$ (faible probabilité d'intersection de Y par les images de E).

Idée de la démonstration Si l'ensemble E est grand alors ses images par les applications A_i couvrent une grande partie de $\{0,1\}^m$ et l'un des mots aléatoires y_i a une grande probabilité d'appartenir à l'une de ces images.

Si en revanche l'ensemble E est petit, alors ses images le sont également et un mot aléatoire y_i aura une faible chance d'être dans l'union des images.

Démonstration

1. On commence par le cas où E est de grande taille. Quitte à considérer un sous-ensemble de E, on peut supposer que $|E| \leqslant 2^{m-2}$.

   Montrons d'abord que $|A(E)| \geqslant |E|/k$ avec grande probabilité. Il suffit de montrer qu'avec grande probabilité il existe i tel que $|A_i(E)| \geqslant |E|/k$. On appelle

2. Ce genre de protocole est une brique importante pour concevoir des protocoles plus sophistiqués. C'est pourquoi d'autres protocoles existent pour évaluer la taille d'un ensemble ; ils varient par la probabilité d'erreur ou par le nombre d'éléments dont il faut tester l'appartenance à l'ensemble. On peut citer notamment le protocole de Sipser [Sip83].

classe par A_i d'un élément $x \in E$ l'ensemble des éléments $x' \in E$ qui ont la même image que x par A_i. Il suffit alors de montrer qu'avec grande probabilité, l'un des A_i possède au moins $|E|/k$ classes distinctes, et par le principe des tiroirs, il est suffisant que le nombre total cumulé de classes de A_1, de A_2, … et de A_k soit au moins $|E|$.
Pour cela, il suffit de montrer qu'avec grande probabilité,

$$\forall x \in E, \exists j, \forall x' \in E\ (x \neq x' \implies A_j x \neq A_j x') :$$

en effet, si tel est le cas alors tout point $x \in E$ est seul dans sa classe pour l'un des A_j, et donc le nombre total cumulé de classes est au moins $|E|$. Nous allons montrer que cette propriété est vraie avec probabilité au moins $1-2^{-k-2}$ (résultat intermédiaire qu'on notera $(\star)$), c'est-à-dire que son complémentaire est vrai avec probabilité au plus 2^{-k-2}.
Soit $x \in \{0,1\}^n$, $x \neq 0$, et $i \in [1,m]$. Pour exactement la moitié des matrices A de taille $m \times n$ sur $\mathbb{F}_2$, la i-ème coordonnée de Ax est 1 : en effet, si le j-ème bit de x est non nul, alors il suffit de changer la coordonnée (i,j) de A pour faire passer la i-ème coordonnée de Ax de 1 à 0. Par ailleurs, pour une matrice aléatoire A, les différentes coordonnées de Ax sont indépendantes les unes des autres puisque la i-ème dépend uniquement de la i-ème ligne de A. On en déduit que pour $x \neq x'$, $\Pr_A(Ax = Ax') = 2^{-m}$ puisque les m coordonnées de $A(x-x')$ doivent être nulles, et donc que pour tout j, $\Pr_{A_j}(\exists x' \in E\ (x \neq x' \wedge A_j x = A_j x')) \leqslant |E| 2^{-m}$ par l'inégalité de Boole (proposition A-B).
Puisque les matrices $A_1, \ldots, A_k$ sont indépendantes, on a donc

$$\Pr_{A_1,\ldots,A_k}\left(\forall j\ \exists x' \in E\ (x \neq x' \wedge A_j x = A_j x')\right) \leqslant |E|^k 2^{-mk}.$$

On déduit par l'inégalité de Boole que

$$\Pr_{A_1,\ldots,A_k}\left(\exists x \in E\ \forall j\ \exists x' \in E\ (x \neq x' \wedge A_j x = A_j x')\right) \leqslant |E|^{k+1} 2^{-mk}.$$

Ainsi, si $|E| \leqslant 2^{m-2}$ alors la probabilité précédente est majorée par

$$2^{(m-2)(k+1)} 2^{-mk} = 2^{-2k+m-2} \leqslant 2^{-k-2} \text{ puisque } k \geqslant m,$$

ce qui montre le résultat $(\star)$.
On a donc $|A(E)| \geqslant |E|/k$ avec probabilité au moins $1 - 2^{-k-2}$. Puisque $|E| \geqslant 2^{m-3}$, on en déduit que $|A(E)| \geqslant 2^m/(8k)$ avec probabilité $1 - 2^{-k-2}$. Si c'est le cas, alors en tirant $8k^2$ points $y_1, \ldots, y_{8k^2} \in \{0,1\}^m$, la probabilité que tous soient hors de l'image $A(E)$ est au plus $(1-1/(8k))^{8k^2} \leqslant 2^{-k-1}$ (pour $k \geqslant 3$, en utilisant l'inégalité $(1-1/n)^n < 1/e$). On en déduit que

$$\Pr_{A_1,\ldots,A_k,Y}(Y \cap A(E) \neq \emptyset) \geqslant 1 - 2^{-k-1} - 2^{-k-2} \geqslant 1 - 2^{-k}.$$

2. Puisque $A(E)$ est composé des images de E par k matrices, on a $|A(E)| \leqslant k|E|$. Donc $|A(E)| \leqslant k2^m/d$ par hypothèse sur $|E|$, c'est-à-dire $|A(E)|/2^m \leqslant k/d$. Pour

chacun des $8k^2$ mots y_j, la probabilité qu'il soit dans $A(E)$ est donc $\leqslant k/d$. Par l'inégalité de Boole (proposition A-B), la probabilité que l'un d'entre eux soit dans $A(E)$ est donc majorée par $8k^3/d$.

□

Ce lemme permet de concevoir un protocole pour évaluer la taille d'un ensemble, dans lequel Merlin n'a besoin d'envoyer qu'un seul mot de E. En effet, si Merlin veut convaincre Arthur qu'un ensemble E a une taille au moins 2^{m-3}, on obtient le protocole suivant où l'on suppose que tester l'appartenance à E se fait en temps polynomial :

- Merlin envoie m à Arthur ;
- Arthur envoie des matrices aléatoires $A_1,\dots,A_k$ et des mots aléatoires $y_1,\dots,y_{8k^2}$;
- Merlin répond par un mot $x \in E$;
- Arthur vérifie que $\{A_1(x),\dots,A_k(x)\} \cap \{y_1,\dots,y_{8k^2}\} \neq \emptyset$.

Si $|E| \geqslant 2^{m-3}$ alors Arthur accepte avec probabilité $\geqslant 1-2^{-k}$; si $|E| \leqslant 2^m/d$ alors Arthur accepte avec probabilité $\leqslant 8k^3/d$.

Nous aurons en outre besoin d'un lemme technique permettant de découper un grand ensemble en de « nombreuses grosses » parties.

10-AD Lemme

Soit $E_1,\dots,E_n$ des ensembles finis non vides tels que $|\cup_i E_i| = N$. Alors il existe un entier $N' \geqslant 1$ tel que le nombre de i satisfaisant $|E_i| \geqslant N'$ soit au moins $N/(N'(1+\ln n))$ (où le logarithme est en base e).

Pour $n \geqslant 10$, $1+\ln n \leqslant \log n$ (où log désigne le logarithme en base 2), donc on peut minorer le nombre de i ci-dessus par $N/(N'\log n)$.

Démonstration Pour $n = 1$, il suffit de prendre $N' = N = |E_1|$.

On considère donc maintenant $n > 1$. Remarquons d'abord que $\sum_{i=1}^{n} 1/i < 1 + \int_1^n dx/x = 1+\ln n$.

Sans perte de généralité on peut supposer que les E_i vérifient $|E_1| \geqslant |E_2| \geqslant \cdots \geqslant |E_n|$. Nous allons montrer qu'il existe i tel que $i|E_i| \geqslant N/(1+\ln n)$, ce qui impliquera le résultat puisque les i ensembles $E_1,\dots,E_i$ seront chacun de cardinal $\geqslant N/(i(1+\ln n))$. Si ce n'est pas le cas, on a alors $|E_i| \leqslant N/(i(1+\ln n))$ pour tout i, donc

$$|\cup_i E_i| \leqslant \sum_{i=1}^{n} N/(i(1+\ln n)) = (N/(1+\ln n)) \sum_{i=1}^{n} 1/i < N,$$

une contradiction. □

10-AE **Théorème (Goldwasser et Sipser, 1986)**

Pour tout polynôme $t(n) \geqslant 2$, $\mathsf{IP}[t(n)] = \mathsf{AM}[t(n)]$.

Idée de la démonstration L'inclusion intéressante est $\mathsf{IP}[t(n)] \subseteq \mathsf{AM}[t(n)]$ où nous souhaitons transformer un protocole à bits aléatoires privés (type IP) en protocole à bits aléatoires publics (type AM). Il s'agit d'évaluer par un protocole Arthur-Merlin le nombre de choix aléatoires (privés) qui font accepter le vérificateur, afin de déterminer si la probabilité d'acceptation est grande ou non.

Pour évaluer la taille de cet ensemble, on utilise le lemme 10-AC, mais Arthur ne peut pas tester directement l'appartenance à l'ensemble car il lui faut connaître les réponses du prouveur. Le protocole Arthur-Merlin simule alors chaque étape du protocole IP, et au cours de la simulation Merlin fournit à Arthur les questions « typiques » du vérificateur et les réponses associées du prouveur. Il lui faut montrer que les questions qu'il propose sont effectivement « typiques » (c'est-à-dire qu'elles reflètent le comportement le plus probable du vérificateur), ce qui se fait en évaluant le nombre de choix aléatoires aboutissant à ces questions, grâce aux lemmes 10-AC et 10-AD.

Démonstration L'inclusion $\mathsf{AM}[t(n)] \subseteq \mathsf{IP}[t(n)]$ est évidente puisque dans un protocole IP à bits privés, le vérificateur peut envoyer ses bits aléatoires au prouveur et simuler ainsi un protocole Arthur-Merlin.

Pour l'autre inclusion, soit $L \in \mathsf{IP}[t(n)]$ par un protocole de type IP entre un vérificateur V et un prouveur P. À partir du protocole de type $\mathsf{IP}[t(n)]$ entre V et P, nous allons construire un protocole de type $\mathsf{AM}[t(n)+4]$ entre A et M (et on se ramènera à $\mathsf{AM}[t(n)]$ grâce à la remarque 10-Y). Le but de Merlin est de convaincre Arthur que pour la plupart des choix aléatoires (privés) r du vérificateur V, inconnus de P, le prouveur P arrivait à convaincre V avec grande probabilité.

Hypothèses et notations Sans perte de généralité, on suppose que $t(n)$ est pair et pour simplifier on notera $t(n) = 2s$; que V et P échangent à chaque tour un message de $p(n)$ bits, où $p(n)$ est un polynôme qui vérifie $2^{p(n)} \geqslant 10$ pour tout n (c'est-à-dire $p(n) \geqslant \log 10$, cela pour simplifier l'application du lemme 10-AD) ; et que V dispose au total de $p(n)$ bits aléatoires formant un mot r. On note $y_1, z_1, y_2, \ldots, y_s, z_s$ les $2s$ messages envoyés alternativement par V puis P, qui sont donc tous de taille $p(n)$. On fixe par ailleurs $k = \lceil 2 + \log(s+1) \rceil$, de sorte que $(s+1)2^{-k} < 1/3$.

On va aussi supposer que la probabilité d'erreur est faible. Si l'on effectuait q répétitions parallèles du protocole, on passerait d'une probabilité d'erreur $1/3$ à α^q pour une constante $\alpha < 1$ tandis que la taille des messages échangés passerait à $q\,p(n)$: on pourrait choisir q polynomial en n tel que $\alpha^q < \max(q\,p(n), 6k^3)^{-2s-2}$, c'est-à-dire que la probabilité d'erreur serait inférieure à la taille des messages puissance $(-2s-2)$ et à $(6k^3)^{-2s-2}$. Ainsi, sans perte de généralité on supposera pour la suite que la probabilité d'erreur finale est η^2 où η vérifie à la fois $\eta < (1-\eta^2)p(n)^{-s}$ et $\eta < (6k^3)^{-s-1}$.

Pour $i \leqslant s$ on définit

$$E_{y_1,\ldots,y_i} = \{r \mid V(x,r) = y_1 \wedge V(x,r,z_1) = y_2 \wedge \cdots \wedge V(x,r,z_1,\ldots,z_{i-1}) = y_i \\ \wedge\ x \text{ est accepté dans le protocole entre } V \text{ et } P \text{ avec bits aléatoires } r\},$$

c'est-à-dire l'ensemble des bits aléatoires de V qui permettent d'obtenir la suite de messages $y_1,\dots,y_i$ sachant que les réponses de P sont $z_1,\dots,z_{i-1}$ (celles-ci ne dépendent que des $y_1,\dots,y_{i-1}$ et non de r puisque les bits aléatoires de V sont privés), dans une exécution du protocole qui accepte x. On remarque en particulier que

$$\text{pour tout } 0 < i \leqslant s, \quad E_{y_1,\dots,y_{i-1}} = \bigcup_{y_i} E_{y_1,\dots,y_i}.$$

Selon cette notation, E_ϵ (où ϵ est le mot vide) désigne l'ensemble

$$\{r \mid x \text{ est accepté dans le protocole entre } V \text{ et } P\}.$$

On a alors $x \in L$ ssi $|E_\epsilon| > \eta 2^{p(n)}$: en effet, dans ce cas V accepte pour une proportion strictement supérieure à η de ses choix aléatoires r de taille $p(n)$, donc x ne peut pas être hors de L par définition du protocole entre V et P qui a une probabilité d'erreur η^2. Ainsi, Merlin va chercher à prouver à Arthur que $|E_\epsilon| > \eta 2^{p(n)}$.

Discussion informelle Le problème est que l'appartenance d'un mot r à E_ϵ n'est a priori pas testable par Arthur sans connaître les réponses $z_1,\dots,z_s$ de P, donc on ne peut pas appliquer directement le protocole d'approximation de la taille d'un ensemble du lemme 10-AC. On peut en revanche l'utiliser indirectement. Rappelons que P n'a pas accès aux bits aléatoires de V, donc il ne peut travailler directement sur r : il manipule seulement les questions y_i de V et ses réponses z_i. L'idée pour prouver que E_ϵ est gros est de montrer qu'il existe de nombreuses questions y_1 pour lesquelles E_{y_1} est gros : cela implique que $E_\epsilon = \cup_{y_1} E_{y_1}$ est gros. De même, prouver que E_{y_1} est gros nécessite de montrer qu'il existe de nombreuses questions y_2 telles que E_{y_1,y_2} est gros, etc.

Donc Merlin souhaite montrer qu'il y a de nombreuses questions y_1 pour lesquelles E_{y_1} est gros. Pour cela Arthur lui envoie des matrices aléatoires selon le lemme 10-AC et Merlin lui donne un tel y_1 (censé certifier qu'il y a beaucoup de mots y_1 tels que E_{y_1} est gros), ainsi que la réponse z_1 de P. Merlin prétend alors qu'il y a de nombreux y_2 pour lesquels E_{y_1,y_2} est gros : Arthur lui envoie des matrices aléatoires selon le lemme 10-AC et Merlin lui donne un tel y_2, ainsi que la réponse z_2 de P, etc.

La notion de « beaucoup » et de « gros » dans la discussion ci-dessus se réfère au lemme 10-AD. Si $x \in L$ alors on sait que $|E_\epsilon| \geqslant (1-\eta^2)2^{p(n)}$ (mais il suffit de montrer à Arthur que $|E_\epsilon| > \eta 2^{p(n)}$). Par le lemme 10-AD, il existe N_1 et N_1' tels que pour au moins N_1 mots y_1, on ait

$$|E_{y_1}| \geqslant N_1', \quad \text{et} \quad N_1 N_1' \geqslant (1-\eta^2)2^{p(n)}/p(n).$$

Merlin envoie un tel y_1 et la réponse z_1 de P. En réalité, il suffit à Merlin de prouver à Arthur que $|E_{y_1}| \geqslant N_1'' = N_1'/p(n)^{s-1}$, puisque $N_1 N_1'' > \eta 2^{p(n)}$ par hypothèse sur η. Il envoie donc N_1 et N_1'' et prétend que $|E_{y_1}| \geqslant N_1''$.

Il agit de même pour la suite : on sait que $|E_{y_1}| \geqslant N_1'$ donc il existe N_2 et N_2' tels que $N_2 N_2' \geqslant N_1'/p(n)$ et pour au moins N_2 mots y_2 on ait $|E_{y_1,y_2}| \geqslant N_2'$. Merlin envoie un

tel y_2 ainsi que la réponse z_2 de P. Mais il suffit à Merlin de montrer que $|E_{y_1,y_2}| \geqslant N_2'' = N_2'/p(n)^{s-2}$. Il envoie donc N_2 et N_2'' et prétend que $|E_{y_1,y_2}| \geqslant N_2''$. Et ainsi de suite. À la dernière étape, il doit montrer que $|E_{y_1,\ldots,y_s}| \geqslant N_s''$, ce qui se fait grâce au lemme 10-AC en envoyant un élément de $E_{y_1,\ldots,y_s}$.

Si $x \in L$, alors $E_\epsilon \geqslant (1-\eta^2)2^{p(n)}$ et la stratégie précédente permet donc à Merlin de convaincre Arthur avec grande probabilité. En revanche, si $x \notin L$ alors les ensembles $E_{y_1,\ldots,y_i}$ sont trop petits pour que Merlin puisse convaincre Arthur avec grande probabilité. Nous formalisons cette intuition dans la suite.

Le protocole Arthur-Merlin Plus précisément, on obtient le protocole suivant entre Arthur et Merlin.

Tours 1 et 2 :

- Merlin envoie N_1, N_1'' et prétend que pour au moins N_1 mots y_1, on a $|E_{y_1}| \geqslant N_1''$.
- Arthur vérifie que $N_1 N_1'' > \eta 2^{p(n)}$ (sinon il rejette).
  Soit $m_1 = 3 + \lfloor \log N_1 \rfloor$. Arthur renvoie des matrices aléatoires $A_1^1, \ldots, A_k^1$ de taille $m_1 \times p(n)$ et des mots aléatoires $r_1^1, \ldots, r_{8k^2}^1 \in \{0,1\}^{m_1}$ formant un ensemble R^1 comme au lemme 10-AC.

Tours $2i-1$ et $2i$ $(2 \leqslant i \leqslant s)$:

- Merlin envoie $y_{i-1}, z_{i-1}, N_i, N_i''$ et prétend que pour au moins N_i mots y_i, on a $|E_{y_1,\ldots,y_i}| \geqslant N_i''$.
- Arthur vérifie que $A^{i-1}(y_{i-1}) \cap R^{i-1} \neq \emptyset$ (sinon il rejette).
  Arthur vérifie que $N_i N_i'' \geqslant N_{i-1}''$ (sinon il rejette).
  Soit $m_i = 3 + \lfloor \log N_i \rfloor$. Arthur renvoie des matrices aléatoires $A_1^i, \ldots, A_k^i$ de taille $m_i \times p(n)$ et des mots aléatoires $r_1^i, \ldots, r_{8k^2}^i \in \{0,1\}^{m_i}$ formant un ensemble R^i comme au lemme 10-AC.

Tours $2s+1$ à $2s+3$:

- Merlin envoie y_s, z_s et prétend que $|E_{y_1,\ldots,y_s}| \geqslant N_s''$.
- Arthur vérifie que $A^s(y_s) \cap R^s \neq \emptyset$ (sinon il rejette).
  Soit $m'' = 3 + \lfloor \log N_s'' \rfloor$. Arthur renvoie des matrices aléatoires $A_1'', \ldots, A_k''$ de taille $m'' \times p(n)$ et des mots aléatoires $r_1'', \ldots, r_{8k^2}'' \in \{0,1\}^{m''}$ formant un ensemble R'' comme au lemme 10-AC.
- Merlin envoie un mot r de taille $p(n)$.
- Arthur vérifie que $r \in E_{y_1,\ldots,y_s}$ et que $A''(r) \cap R'' \neq \emptyset$ (sinon il rejette). Si tous les tests ont réussi, Arthur accepte.

On remarquera que la dernière intervention d'Arthur est déterministe donc n'est pas un « tour » à proprement parler. Il y a $2s+3 = t(n)+3$ tours et Merlin commence, donc le protocole est du type $\mathsf{MA}[t(n)+3] \subseteq \mathsf{AM}[t(n)+4]$. La remarque 10-Y permet de se ramener à $\mathsf{AM}[t(n)]$.

Analyse lorsque $x \in L$ Si $x \in L$, alors Merlin peut convaincre Arthur de la façon esquissée ci-dessus. Plus précisément :

- aux tours 1 et 2, $|E_\epsilon| \geqslant (1-\eta^2)2^{p(n)}$ donc par le lemme 10-AD il existe N_1, N_1' tels que $N_1 N_1' \geqslant (1-\eta^2)2^{p(n)}/p(n)$ et au moins N_1 mots y_1 vérifient $|E_{y_1}| \geqslant N_1'$. On définit $N_1'' = N_1' p(n)^{-s+1}$ et Merlin envoie N_1 et N_1''. Par définition de η, $N_1 N_1'' > \eta 2^{p(n)}$ donc Arthur ne rejette pas et envoie les matrices et les mots aléatoires.
- Aux tours $2i-1$ et $2i$: le choix de N_{i-1} et N_{i-1}' assure qu'il existe au moins N_{i-1} mots y_{i-1} pour lesquels $|E_{y_1,\dots,y_{i-1}}| \geqslant N_{i-1}'$. Par le lemme 10-AC, il y a suffisamment de tels mots y_{i-1} pour que Merlin puisse répondre au défi d'Arthur d'en trouver un tel que $A^{i-1}(y_{i-1}) \cap R^{i-1} \neq \emptyset$ avec probabilité $1-2^{-k}$. En plus de ce mot y_{i-1}, Merlin envoie la réponse z_{i-1} de P correspondante.
  Puisque $|E_{y_{i-1}}| \geqslant N_{i-1}'$, le lemme 10-AD garantit qu'il existe N_i, N_i' tels que
  $$N_i N_i' \geqslant N_{i-1}'/p(n)$$
  et au moins N_i mots y_i vérifient $|E_{y_i}| \geqslant N_i'$. On définit $N_i'' = N_i' p(n)^{-s+i}$ et Merlin envoie N_i et N_i''.
  Les tests d'Arthur sont positifs puisque $A^{i-1}(y_{i-1}) \cap R^{i-1} \neq \emptyset$ et que $N_i N_i'' = N_i N_i' p(n)^{-s+i} \geqslant N_{i-1}' p(n)^{-s+i-1} = N_{i-1}''$.
- Aux tours $2s+1$ à $2s+3$: le choix de N_s et N_s' permet de s'assurer qu'il existe au moins N_s mots y_s pour lesquels $|E_{y_1,\dots,y_s}| \geqslant N_s'$. Par le lemme 10-AC, il y a suffisamment de tels mots y_s pour que Merlin puisse répondre au défi d'Arthur d'en trouver un tel que $A^s(y_s) \cap R^s \neq \emptyset$ avec probabilité $1-2^{-k}$. En plus de ce mot y_s, Merlin envoie la réponse z_s de P correspondante.
  Puisque $|E_{y_1,\dots,y_s}| \geqslant N_s' = N_s''$, Merlin peut répondre au dernier défi d'Arthur avec probabilité $1-2^{-k}$ et envoyer un mot r tel que $A''(r) \cap R'' \neq \emptyset$. Il achève ainsi de convaincre Arthur.

Lorsque $x \in L$, la probabilité que Merlin ne convainque pas Arthur est donc 2^{-k} à chacun des $s+1$ défis d'Arthur, soit au total sur l'ensemble du protocole au plus $(s+1)2^{-k}$. Le choix de k assure donc que Merlin convainc Arthur avec probabilité $\geqslant 2/3$.

Analyse lorsque $x \notin L$ Nous montrons maintenant que Merlin ne peut pas convaincre Arthur avec grande probabilité lorsque $x \notin L$. Si $x \notin L$, alors $|E_\epsilon| \leqslant \eta^2 2^{p(n)}$. Nous allons raisonner par l'absurde en partant de la fin du protocole et en supposant que Merlin convainque Arthur avec probabilité $> 1/3$.

– Au dernier tour, cela signifie que Merlin a pu trouver un mot r convenable avec probabilité $> 1/3$: par le lemme 10-AC, on a donc $|E_{y_1,\dots,y_s}| \geqslant 2^{m''}/(24k^3) \geqslant N''_s/(6k^3)$.

– De même au tour $2s+1$, pour convaincre Arthur avec probabilité $> 1/3$ il faut que le nombre de y_s tels que $|E_{y_1,\dots,y_s}| \geqslant N''_s/(6k^3)$ soit au moins $N_s/(6k^3)$: on en déduit que $|E_{y_1,\dots,y_{s-1}}| \geqslant N_s N''_s/(6k^3)^2$.

Pour les tours $2i-1$ et $2i$, où i varie de s à 3 :

– on doit avoir $N_i N''_i \geqslant N''_{i-1}$ pour tout $i \geqslant 2$ (en particulier, $N_s N''_s \geqslant N''_{s-1}$ donc $|E_{y_1,\dots,y_{s-1}}| \geqslant N''_{s-1}/(6k^3)^2$) ;

– le nombre de mots y_{i-1} tels que $|E_{y_1,\dots,y_{i-1}}| \geqslant N''_{i-1}/(6k^3)^{s-i+2}$ doit être au moins $N_{i-1}/(6k^3)$, ce qui implique

$$|E_{y_1,\dots,y_{i-2}}| \geqslant N_{i-1}N''_{i-1}/(6k^3)^{s-i+3} \geqslant N''_{i-2}/(6k^3)^{s-i+3}.$$

Aux tours 3 et 4, on doit avoir que le nombre de mots y_1 tels que $|E_{y_1}| \geqslant N''_1/(6k^3)^s$ soit au moins $N_1/(6k^3)$, donc $|E_\epsilon| \geqslant N_1 N''_1/(6k^3)^{s+1}$. La première vérification d'Arthur s'assure que $N_1 N''_1 > \eta 2^{p(n)}$, donc $|E_\epsilon| > \eta 2^{p(n)}/(6k^3)^{s+1} > \eta^2 2^{p(n)}$ par définition de η. C'est une contradiction avec le fait que $x \notin L$ et que la probabilité d'erreur dans le protocole entre V et P est η^2. □

10-AF **Remarque** Attention, le théorème précédent n'implique bien sûr pas que IP = AM, car pour des raisons historiques, comme nous l'avons vu, les classes Arthur-Merlin sont définies pour un nombre constant de tours alors que la classe IP l'est pour un nombre de tours polynomial…

10.2.5 Le problème de l'isomorphisme de graphes

Une application des protocoles interactifs concerne le problème de l'isomorphisme de graphes introduit à la section 10.1.2. On ne sait pas si le problème ISO est NP-complet ni s'il est dans P. Il s'agit de l'un des rares problèmes naturels dans ce cas, aux côtés de la factorisation d'entiers. Néanmoins Boppana, Håstad et Zachos [BHZ87] ont montré qu'il est peu probable qu'il soit NP-complet car cela amènerait à un effondrement de la hiérarchie polynomiale.

10-AG **Théorème (Boppana, Håstad et Zachos, 1987)**

Si coNP ⊆ AM alors $\Sigma_2^p = \Pi_2^p$ (et donc PH = Σ_2^p).

10-AH Corollaire

Si ISO est NP-complet, alors $\Sigma_2^p = \Pi_2^p$ (et donc $\mathsf{PH} = \Sigma_2^p$).

Démonstration Grâce au théorème 10-AE, le protocole $\mathsf{IP}[2]$ pour coISO vu à la section 10.1.2 peut être transformé en protocole AM, donc $\mathtt{coISO} \in \mathsf{AM}$.

Si ISO est NP-complet alors coISO est coNP-complet (cf. lemme 3-AI), donc $\mathsf{coNP} \subseteq \mathsf{AM}$ et le théorème 10-AG conclut. □

Démonstration du théorème 10-AG Il suffit de montrer que $\Sigma_2^p \subseteq \Pi_2^p$ puisque cela implique $\Pi_2^p = \mathrm{co}\Sigma_2^p \subseteq \mathrm{co}\Pi_2^p = \Sigma_2^p$. Soit $L \in \Sigma_2^p$: on a

$$x \in L \iff \exists y \forall z (x,y,z) \in B,$$

où y et z sont de taille polynomiale et $B \in \mathsf{P}$. Nous allons exprimer l'appartenance à L sous une forme Π_2^p.

Le langage $C = \{(x,y) \mid \forall z (x,y,z) \in B\}$ est dans coNP, donc dans AM par hypothèse. Par la proposition 10-AA et en réduisant la probabilité d'erreur, on a :

$$\begin{cases} (x,y) \in C \implies & \forall r \exists t (x,y,r,t) \in D \\ (x,y) \notin C \implies & \Pr_r(\exists t (x,y,r,t) \in D) < 2^{-|y|}, \end{cases}$$

où $D \in \mathsf{P}$ et r et t sont des mots de taille polynomiale. On en déduit que

$$x \in L \implies \exists y \forall r \exists t (x,y,r,t) \in D,$$

ce qui implique évidemment

$$x \in L \implies \forall r \exists y \exists t (x,y,r,t) \in D.$$

Lorsque $x \notin L$, pour tout mot y soit $R_y = \{r \mid \exists t (x,y,r,t) \in D\}$. Par ce qui précède, on a :

$$x \notin L \implies \forall y |R_y| < 2^{|r|-|y|},$$

ce qui implique $|\cup_y R_y| < 2^{|r|}$ donc $\exists r \notin \cup_y R_y$, c'est-à-dire :

$$x \notin L \implies \exists r \forall y \forall t (x,y,r,t) \notin D.$$

On en déduit que

$$x \in L \iff \forall r \exists y \exists t (x,y,r,t) \in D,$$

donc $L \in \Pi_2^p$. □

Nous verrons une autre application des protocoles interactifs au chapitre 11 sur les bornes inférieures non uniformes.

10.3 Le théorème PCP

Nous terminons ce chapitre par un bref aperçu d'un autre type de protocoles interactifs appelé *Probabilistically Checkable Proofs*. Le résultat principal (et difficile) connu sous le nom de « théorème PCP » a été une contribution majeure à la complexité et a permis de montrer des résultats d'inapproximabilité comme nous allons le voir. Nous ne donnerons pas la preuve de ce théorème et invitons le lecteur à consulter le livre d'Arora et Barak [AB09] pour les détails. Les premières versions du théorème sont dues à Arora et Safra [AS92] et à Arora, Lund, Motwani, Sudan et Szegedy [Aro+92]. Dinur [Din06] en a grandement simplifié la preuve. La version que nous donnons et l'application à 3SAT sont de Håstad [Hås97].

10.3.1 Probabilistically Checkable Proofs

Pour les « preuves vérifiables de manière probabiliste », le mécanisme d'interaction ressemble fortement à la classe NP mais avec une vérification probabiliste du certificat. Il existe plusieurs variantes dans la définition des classes PCP notamment sur la probabilité d'erreur, qui ne sont pas forcément équivalentes en termes de nombre de requêtes ; nous présentons ici la variante qui permet d'obtenir le meilleur résultat d'inapproximabilité.

10-AI Définition

– Soit M une machine de Turing déterministe et π un mot de taille quelconque. On dit que M a un *accès direct* à π, et on note M^π, si M peut demander de consulter le i-ème bit de π (elle écrit pour cela i en binaire sur un ruban spécial) et recevoir la réponse en temps unitaire.
En d'autres termes, M a accès à l'oracle $\Pi = \{i \mid \pi_i = 1\}$ où i est codé en binaire. Ainsi, accéder à tout bit de π demande à M seulement $O(\log|\pi|)$ étapes.

– Si $r(n)$ et $s(n)$ sont des fonctions et $\epsilon \in]0, 1/4[$ une constante, la classe

$$\mathsf{PCP}_\epsilon(r(n), s(n))$$

est l'ensemble des langages L tels qu'il existe une machine de Turing déterministe M fonctionnant en temps polynomial telle que :

$$\begin{cases} x \in L \implies & \exists \pi \Pr_{r \in \{0,1\}^{r(|x|)}}(M^\pi(x, r) = 1) \geqslant 1 - \epsilon \\ x \notin L \implies & \forall \pi \Pr_{r \in \{0,1\}^{r(|x|)}}(M^\pi(x, r) = 1) \leqslant 1/2 + \epsilon, \end{cases}$$

et M consulte au plus $s(|x|)$ bits de π de manière non adaptative (cf. remarque ci-dessous).
Ainsi, $r(n)$ est le nombre de bits aléatoires autorisés et $s(n)$ est le nombre maximal de bits de π que M consulte.

Le mécanisme de certificat habituel des classes non déterministes est donc généralisé à une vérification probabiliste de la preuve, mais où l'on contrôle finement le nombre de bits aléatoires et le nombre de bits lus dans la preuve.

10-AJ Remarques

- Aucune contrainte n'est imposée sur la taille du mot π ; cependant, en pratique celle-ci est limitée à cause du temps de fonctionnement de M puisqu'en temps polynomial $t(n)$ elle ne pourra pas demander des bits dont la position s'écrit avec plus de $t(n)$ bits. Donc sans perte de généralité, la taille de π peut être restreinte à $2^{n^{O(1)}}$.
- Le fait que les requêtes de M soient non adaptatives nous permet de reformuler l'accès à π par l'existence d'une fonction $f : \Sigma^\star \times \Sigma^\star \times \mathbb{N} \to \mathbb{N}$, calculable en temps déterministe polynomial et donnant les positions consultées dans la preuve :

$$\begin{cases} x \in L \implies & \exists \pi \Pr_{r \in \{0,1\}^{r(|x|)}}(M(x, r, \pi_{f(x,r,1)} \dots \pi_{f(x,r,s(|x|))}) = 1) \geqslant 1 - \epsilon \\ x \notin L \implies & \forall \pi \Pr_{r \in \{0,1\}^{r(|x|)}}(M(x, r, \pi_{f(x,r,1)} \dots \pi_{f(x,r,s(|x|))}) = 1) \leqslant 1/2 + \epsilon. \end{cases}$$

10.3.2 Résultats

De manière incroyable, dans ce contexte le théorème PCP montre qu'il suffit de lire 3 bits de la preuve pour capturer toute la puissance de la classe NP. C'est exactement comme si un enseignant mettait au point une façon pour les étudiants de rédiger leur copie (un « code » à suivre pour rédiger une copie) grâce auquel il n'a besoin de lire que trois lettres aléatoires de chaque copie pour mettre sa note [3]…

10-AK Théorème (Håstad, 1997)

Pour tout $\epsilon \in]0, 1/4[$, $\mathsf{NP} = \mathsf{PCP}_\epsilon(O(\log n), 3)$.

Encore une fois, cela témoigne de la puissance de l'aléatoire dans l'interaction prouveur/vérificateur.

Ce théorème a des conséquences profondes concernant l'approximation de nombreux problèmes NP-complets. Nous donnerons seulement un exemple mais beaucoup d'autres problèmes peuvent être traités ainsi. Soit Max3SAT le problème d'évaluation (c'est-à-dire une fonction $\mathtt{Max3SAT} : \Sigma^\star \to \mathbb{N}$) suivant :

- *entrée :* une formule booléenne $\varphi(x_1, \dots, x_n)$ en 3-CNF ;

3. En réalité, il doit lire seulement 3 bits (et non lettres), mais il ne peut répondre que par vrai ou faux et non mettre une note puisqu'il doit s'agir d'un problème de décision. Par dichotomie, pour mettre une note entière sur 20 il n'a qu'à répéter 5 fois la procédure, c'est-à-dire lire 15 bits au total. Évidemment, pour être tout à fait honnête il faudrait que l'enseignant diminue la probabilité d'erreur et donc répète toute la procédure plusieurs fois.

– *sortie :* le nombre maximal de clauses satisfaisables simultanément par une assignation de $(x_1, \ldots, x_n)$.

Avant de voir le résultat d'inapproximabilité, nous avons besoin d'un lemme technique.

10-AL Lemme

Toute fonction booléenne à trois variables peut être exprimée par une formule booléenne en 3-CNF ayant au plus 4 clauses.

Démonstration Soit $f(x,y,z)$ une fonction booléenne à 3 variables. Nous l'exprimons d'abord facilement grâce à 8 clauses au plus. Pour cela, on considère les valeurs possibles $(a,b,c) \in \{0,1\}^3$ affectées à (x,y,z) : à $a=0$ on associe le littéral x et à $a=1$ on associe $\neg x$, de même pour b avec y, et pour c avec z. Pour chaque triplet $(a,b,c) \in \{0,1\}^3$ tel que $f(a,b,c)=0$, on ajoute alors comme clause la disjonction des littéraux associés. Ainsi au triplet $(0,1,0)$ par exemple, on associe la clause $(x \vee \neg y \vee z)$, qui est fausse seulement en $(0,1,0)$. Au final on obtient la conjonction ξ d'au plus 8 clauses et cette conjonction est fausse pour toutes les valeurs telles que $f(a,b,c)=0$ et vraie ailleurs, donc elle exprime bien f.

Nous nous ramenons maintenant à au plus 4 clauses. En ignorant la variable z, il n'y a que quatre groupes possibles de clauses : $(x \vee y)$, $(x \vee \neg y)$, $(\neg x \vee y)$ et $(\neg x \vee \neg y)$. S'il y a deux clauses dans le même groupe, elles ont les mêmes littéraux en x et en y et des littéraux opposés en z, et peuvent donc être « factorisées » en supprimant z : par exemple, $(x \vee \neg y \vee z)$ et $(x \vee \neg y \vee \neg z)$ sont dans le même groupe donc on les remplacerait par la clause $(x \vee \neg y)$, ce qui est équivalent à la conjonction des deux clauses originales. □

Nous sommes prêts pour le résultat d'inapproximabilité de Max3SAT.

10-AM Corollaire

Soit $\epsilon > 0$ quelconque. Si $\mathsf{P} \neq \mathsf{NP}$ alors il n'existe pas d'algorithme déterministe polynomial approximant Max3SAT à $(7/8+\epsilon)$.

C'est-à-dire que si $\mathsf{P} \neq \mathsf{NP}$, alors pour toute fonction $f : \Sigma^\star \to \mathbb{N}$ calculable en temps polynomial telle que $f \leqslant \texttt{Max3SAT}$, il existe une instance φ telle que

$$f(\varphi) < (7/8+\epsilon)\texttt{Max3SAT}(\varphi).$$

Démonstration Sans perte de généralité, on suppose $\epsilon < 1/8$.

Par le théorème PCP, $\texttt{3SAT} \in \mathsf{PCP}_{\epsilon/2}(\alpha \log n, 3)$ pour une certaine constante $\alpha > 0$. Par contraposée, supposons qu'il existe une fonction f calculable en temps polynomial telle $(7/8+\epsilon)\texttt{Max3SAT} \leqslant f \leqslant \texttt{Max3SAT}$: pour une instance $\varphi(x)$ en 3-CNF, on souhaite décider en temps déterministe polynomial si $\varphi \in \texttt{3SAT}$.

Dans le protocole PCP pour 3SAT, pour chacun des n^α choix des $\alpha \log n$ bits aléatoires r utilisés par le vérificateur on a une fonction f_r calculable en temps polynomial qui prend en argument les 3 bits $(\pi_{i_1}, \pi_{i_2}, \pi_{i_3})$ lus de la preuve π et renvoie 1 ssi φ est satisfaisable. On peut exprimer le fait que $f_r(\pi_{i_1}, \pi_{i_2}, \pi_{i_3}) = 1$ par une formule φ_r en 3-CNF de taille constante : le lemme 10-AL montre qu'il suffit de 4 clauses.
En ajoutant une variable booléenne y_i pour chaque bit π_i de la preuve π lu au cours d'une des exécutions possibles du vérificateurs (il y en a donc $N \leqslant 3.2^{|r|} = n^{O(1)}$), on construit la formule

$$\psi(y_1, \ldots, y_N) = \bigwedge_r \varphi_r.$$

On note m le nombre de clauses de ψ, où $m \leqslant 4.2^{|r|}$ est polynomial.
Si φ est satisfaisable, alors il existe une preuve π qui satisfait φ_r pour plus d'une fraction $(1 - \epsilon/2)$ des mots r, donc le nombre de clauses satisfaisables simultanément dans ψ est au moins $(1 - \epsilon/2)m$. En revanche, si φ n'est pas satisfaisable, alors toute preuve π satisfait φ_r pour au plus une fraction $(1/2 + \epsilon/2)$ des mots r (et pour les autres, au moins une des 4 clauses de φ_r n'est pas satisfaite) donc le nombre de clauses satisfaisables simultanément dans ψ est au plus

$$((1/2 + \epsilon/2) + 3/4(1/2 - \epsilon/2))m = (7/8 + \epsilon/8)m.$$

Ainsi, s'il existe une approximation f de Max3SAT meilleure que $(7/8 + \epsilon)$, alors si φ est satisfaisable on a

$$f(\psi) \geqslant (7/8 + \epsilon)(1 - \epsilon/2)m > (7/8 + \epsilon/2)m,$$

tandis que si φ n'est pas satisfaisable on a $f(\psi) \leqslant (7/8 + \epsilon/8)m$. On peut donc distinguer les deux cas en temps polynomial et décider si $\varphi \in$ 3SAT. □

10-AN **Remarque** Ce résultat est optimal dans le sens suivant. En tirant une affectation aléatoire des variables d'une formule en 3-CNF où chacune des m clauses a vraiment trois variables différentes, l'espérance du nombre de clauses satisfaites est $(7/8)m$ puisque chaque clause est satisfaite avec probabilité 7/8. Cette observation peut être transformée en un algorithme déterministe polynomial. Pour les formules en 3-CNF n'ayant pas toujours exactement trois variables distinctes par clause (comme ce que produit la réduction utilisée dans la preuve ci-dessus), un algorithme déterministe polynomial dû à Karloff et Zwick [KZ97] permet lui aussi de satisfaire au moins 7/8 des clauses des formules satisfaisables.

Les autres résultats d'inapproximabilité utilisant le théorème PCP concernent les problèmes MaxCLIQUE, MinVertexCover, MaxCSP, etc., et procèdent généralement par réduction à partir d'un résultat d'inapproximabilité déjà montré. En réalité, plutôt qu'une simple implication, les auteurs de [Aro+92] ont montré qu'il existe une équivalence entre le théorème PCP et l'inapproximabilité de certains problèmes.

11 Bornes inférieures non uniformes

Pour classifier les problèmes en fonction de leur difficulté, en complexité on doit montrer des bornes inférieures sur les ressources nécessaires à la résolution de ces problèmes. Cela revient souvent à séparer des classes : par exemple, $\mathsf{EXP} \neq \mathsf{P}$ (corollaire 2-P) signifie que tout problème EXP-difficile ne peut être résolu en temps déterministe polynomial. De même, séparer NP de P revient à montrer une borne inférieure superpolynomiale sur le temps nécessaire pour résoudre le problème SAT. Alors qu'on sait séparer les classes déterministes entre elles (par le théorème de hiérarchie déterministe 2-J) ou les classes non déterministes entre elles (par le théorème de hiérarchie non déterministe 2-AI), on ne sait pas vraiment comparer de manière satisfaisante les classes déterministes et les classes non déterministes comme le rappelle la célèbre question « $\mathsf{P} = \mathsf{NP}$? ».

De la même manière qu'il reste ouvert de montrer la puissance du non-déterminisme, comprendre la puissance de la non-uniformité reste une quête largement balbutiante. Pour montrer que les calculs non uniformes n'apportent pas une puissance démesurée, il s'agit de montrer que certains problèmes ne peuvent pas être résolus par des circuits de petite taille. C'est l'objet de ce chapitre. Malheureusement, nous le verrons, les résultats sont encore assez limités.

Nous commencerons par des méthodes habituelles de diagonalisation afin de montrer que certaines classes n'ont pas de petits circuits. Puis nous verrons que certaines restrictions sur les circuits (profondeur bornée ou monotonicité notamment) empêchent de calculer des problèmes très simples. Enfin, nous étudierons le cas des circuits arithmétiques calculant des polynômes.

11.1 Circuits booléens sans restriction

Pour cette partie on notera $\mathsf{SIZE}(s(n))$ l'ensemble des langages pouvant être décidés par une famille de circuits booléens (C_n) de taille $O(s(n))$ et dont les portes ont un degré

entrant au plus 2.

11.1.1 Bornes inférieures en $\Omega(n^k)$

Nous nous proposons ici de montrer que certains langages n'ont pas de circuits de taille n^k pour une constante k fixée. Nous commençons par un résultat facile que nous améliorerons par la suite. Celui-ci apparaît déjà dans Kannan [Kan81].

11-A Proposition (Kannan, 1981)

Pour toute constante k, $\mathsf{PSPACE} \not\subset \mathsf{SIZE}(n^k)$.

Plus précisément, pour tout k il existe un langage $L_k \in \mathsf{PSPACE}$ et un entier N tels que $\forall n \geqslant N$, tout circuit de taille n^k pour décider $L_k^{=n}$ fait une erreur sur au moins une entrée de taille n.

Idée de la démonstration Il s'agit d'une simple diagonalisation : on construit un langage $L \in \mathsf{PSPACE}$ sur lequel tous les circuits de taille $O(n^k)$ se trompent. Pour chaque taille n, on considère successivement tous les mots de taille n. Pour déterminer si l'on doit mettre un mot x dans L, on évalue tous les circuits qui ne se sont pas trompés sur les mots précédents : on met x dans L ssi la réponse majoritaire est 0. Ainsi, à chaque étape la moitié des circuits qui restent se trompent, donc en un nombre polynomial d'étapes on parvient à éliminer tous les circuits.

Démonstration On note $\mathscr{C}_n$ l'ensemble de tous les circuits booléens de taille n^{k+1} à n entrées : on remarquera que $|\mathscr{C}_n| = 2^{n^{O(k)}}$ (cf. lemme 5-U). Pour n fixé, on note $x^1, \dots, x^{2^n}$ l'ensemble des mots de taille n sur l'alphabet $\{0,1\}$, dans l'ordre lexicographique.

Définissons $L^{=n}$, l'ensemble des mots de taille n du langage L que l'on souhaite construire, avec pour objectif d'avoir $L \in \mathsf{PSPACE} \setminus \mathsf{SIZE}(n^k)$. L'ensemble $L^{=n}$ sera défini mot après mot : on notera L_i les mots ajoutés lors des i premières étapes, avec au début $L_0 = \emptyset$. On notera R_i les circuits de $\mathscr{C}_n$ « restants » (non encore « éliminés ») après les i premières étapes du processus, avec bien sûr $R_0 = \mathscr{C}_n$. Pour i allant de 1 à $1 + \lfloor \log|\mathscr{C}_n| \rfloor$, on définit alors :

$$L_i = \begin{cases} L_{i-1} \cup \{x^i\} & \text{si pour au moins la moitié des circuits } C \in R_{i-1},\ C(x^i) = 0\,; \\ L_{i-1} & \text{sinon.} \end{cases}$$

On pose alors $R_i = \{C \in R_{i-1} \mid C(x^i) = 1\}$ dans le premier cas et $R_i = \{C \in R_{i-1} \mid C(x^i) = 0\}$ dans le second. On pose finalement $L^{=n} = \cup_i L_i = L_{1+\lfloor \log|\mathscr{C}_n| \rfloor}$.

Montrons dans un premier temps que $L \in \mathsf{PSPACE}$. Soit x^i un mot de taille n : si $i > 1 + \lfloor \log|\mathscr{C}_n| \rfloor$ alors $x^i \notin L$ et on peut rejeter. Sinon, on teste successivement si $x^1, \dots, x^{i-1} \in L$ pour pouvoir décider si $x^i \in L$. Pour j de 1 à i, afin de tester si $x^j \in L$ lorsqu'on connaît la réponse pour $x^1, \dots, x^{j-1}$, on procède de la façon suivante :

– compter le nombre N de circuits C de $\mathscr{C}_n$ qui décident correctement si $x^1 \in L$, $x^2 \in L$, …, $x^{j-1} \in L$ (on a donc $N = |R_{j-1}|$) ;
– parmi ces circuits, compter le nombre N' de ceux qui renvoient 0 sur x^j ;
– accepter x^j ssi $N' \geqslant N/2$.

Cette procédure requiert de se souvenir si $x^1 \in L$, …, $x^{j-1} \in L$, donc de mémoriser au plus $\lfloor \log |\mathscr{C}_n| \rfloor$ bits, et d'énumérer et simuler tous les circuits de $\mathscr{C}_n$ de taille n^{k+1}. Tout cela se fait en espace polynomial.

Montrons maintenant que $L \notin \mathsf{SIZE}(n^k)$. Soit (C_n) une famille de circuits de taille $O(n^k)$, et soit α une constante telle que $|C_n| \leqslant \alpha n^k$. Ainsi, pour tout $n \geqslant \alpha$, $n^{k+1} \geqslant |C_n|$.

Soit $n \geqslant \alpha$: on a donc $C_n \in \mathscr{C}_n$. Par construction de L, on a $|R_0| = |\mathscr{C}_n|$ et $|R_i| \leqslant |R_{i-1}|/2$, donc $R_{1+\lfloor \log |\mathscr{C}_n| \rfloor} = \emptyset$. Or R_i est l'ensemble des circuits de $\mathscr{C}_n$ qui donnent la bonne réponse sur $x^1, \ldots, x^i$. Ainsi, $C_n \notin R_{1+\lfloor \log |\mathscr{C}_n| \rfloor}$ doit se tromper sur l'un des mots $x^1, \ldots, x^{1+\lfloor \log |\mathscr{C}_n| \rfloor}$ et donc la famille (C_n) ne reconnaît pas L. □

Nous mentionnons une variante de cette proposition, non pas sur les circuits, mais sur les conseils cette fois. La stratégie est très similaire mais il faut simuler des machines de Turing fonctionnant en temps polynomial et diagonaliser contre tous les conseils possibles. Cette proposition améliore le corollaire 2-P ($\mathsf{EXP} \neq \mathsf{P}$) du théorème de hiérarchie déterministe.

11-B Proposition

Pour toute constante k, $\mathsf{EXP} \not\subseteq \mathsf{P}/n^k$.

✎ **11-C Exercice**

Montrer la proposition précédente.

En réalité nous pouvons montrer mieux que PSPACE dans la proposition 11-A : nous pouvons en effet descendre jusqu'au deuxième niveau de la hiérarchie polynomiale. Nous avons d'abord besoin d'un lemme intéressant en lui-même, qui n'est autre qu'un simple dénombrement et donne un « théorème de hiérarchie non uniforme ».

11-D Lemme (hiérarchie non uniforme)

Pour tout entier $k \geqslant 1$, il existe un langage $L \notin \mathsf{SIZE}(n^k)$ reconnu par une famille de circuits de taille $2n^{k+2}$.

En particulier, $\mathsf{SIZE}(n^k) \subsetneq \mathsf{SIZE}(n^{k+2})$.

Plus précisément, pour tout n suffisamment grand, il existe un circuit C_0 de taille $2n^{k+2}$ reconnaissant un ensemble $E \subseteq \{0,1\}^n$ qui n'est reconnu par aucun circuit de taille majorée par $n^k \log n$.

Démonstration À n fixé, on note $x^1, \ldots, x^{2^n}$ les mots de taille n dans l'ordre lexicographique sur l'alphabet $\{0,1\}$. Pour tout mot $a \in \{0,1\}^{n^{k+1}}$, on note $L_a \subseteq \{0,1\}^n$ l'ensemble vérifiant :

- si $i > n^{k+1}$ alors $x^i \notin L_a$;
- sinon, $x^i \in L_a$ ssi $a_i = 1$.

Clairement $L_a \neq L_b$ si $a \neq b$: pour tout n il y a donc exactement $2^{n^{k+1}}$ ensembles L_a distincts. Or chacun d'eux est reconnu par un circuit de taille $\leqslant 2n^{k+2}$ comme suit. Pour chaque $i \leqslant n^{k+1}$ tel que $a_i = 1$, on crée un sous-circuit vérifiant que l'entrée x est égale à x^i : après avoir calculé les littéraux $\neg x_j$ (ce qui ajoute n portes), c'est simplement une conjonction des littéraux x_j (si le j-ème bit de x^i est 1) ou $\neg x_j$ (si le j-ème bit de x^i est 0), ce qui se fait avec $(n-1)$ portes $\wedge$. On fait ensuite la disjonction de tous ces sous-circuits : il y en a au plus n^{k+1}, il faut donc $n^{k+1}-1$ portes $\vee$. En comptant les entrées, la taille totale du circuit est donc majorée par $2n+n^{k+1}(n-1)+(n^{k+1}-1) = n^{k+2}+2n-1 \leqslant 2n^{k+2}$.

Par ailleurs, d'après le lemme 5-U il y a au plus $3^t t^{3t}$ circuits de taille $\leqslant t$, donc le nombre de circuits de taille $\leqslant n^k \log n$ est majoré par $3^{n^k \log n}(n^k \log n)^{3n^k \log n}$, ce qui est inférieur à $2^{n^{k+1}}$ pour n suffisamment grand. On en déduit que, pour n suffisamment grand, il existe au moins un mot $a \in \{0,1\}^{n^{k+1}}$ tel que L_a n'est pas reconnu par un circuit de taille $\leqslant n^k \log n$. Pour chaque n on note a_n un tel mot $a \in \{0,1\}^{n^{k+1}}$ et on définit le langage $L = \cup_{n \in \mathbb{N}} L_{a_n}$. Ainsi, il existe une constante N telle que pour tout $m \geqslant N$ et pour toute famille de circuits (C_n) de taille $\leqslant n^k \log n$, le circuit C_m ne décide pas $L^{=m}$. Puisque pour toute constante α, on a $n^k \log n \geqslant \alpha n^k$ pour n suffisamment grand, on en déduit que $L \notin \mathsf{SIZE}(n^k)$. □

Dans le même article [Kan81] que précédemment, Kannan montre aussi le résultat suivant.

11-E Théorème (Kannan, 1981)

Pour toute constante k, $\Sigma_2^{\mathsf{p}} \not\subset \mathsf{SIZE}(n^k)$.

Idée de la démonstration Avec quatre quantificateurs, donc dans Σ_4^{p}, on peut exprimer qu'il existe un circuit C_0 de taille $2n^{k+2}$, qui soit le premier dans l'ordre lexicographique tel que pour tout circuit C de taille $O(n^k)$, il existe un mot y pour lequel $C(y) \neq C_0(y)$: ainsi, le langage reconnu par C_0 n'est pas dans $\mathsf{SIZE}(n^k)$.

Pour se ramener à Σ_2^{p} plutôt que Σ_4^{p}, l'astuce consiste à utiliser le théorème 8-P de Karp et Lipton : si $\mathsf{NP} \not\subset \mathsf{P/poly}$ alors le résultat suit ; sinon $\mathsf{PH} = \Sigma_2^{\mathsf{p}}$ donc en particulier $\Sigma_4^{\mathsf{p}} = \Sigma_2^{\mathsf{p}}$.

Démonstration Par le lemme 11-D, il existe des familles de circuits de taille $2n^{k+2}$ qui décident un langage hors de $\mathsf{SIZE}(n^k)$. Pour chaque n, nous allons trouver le premier circuit C_0 dans l'ordre lexicographique qui vérifie une telle propriété : il suffit d'exprimer qu'il existe un circuit C_0 de taille $2n^{k+2}$ tel que :

- pour tout circuit C de taille $n^k \log n$, il existe un mot $y \in \{0,1\}^n$ tel que $C(y) \neq C_0(y)$ (la fonction calculée par C_0 ne possède pas de circuit de taille $n^k \log n$) ;
- pour tout circuit C_0' de taille $2n^{k+2}$, soit $C_0' \geqslant_{\text{lex}} C_0$, soit il existe C' de taille $n^k \log n$ tel que pour tout mot $y' \in \{0,1\}^n$ on ait $C'(y') = C_0'(y')$ (le circuit C_0 est le premier dans l'ordre lexicographique).

Une fois C_0 isolé, sur l'entrée x il suffit de tester si $C_0(x) = 1$ et on décide ainsi l'ensemble reconnu par C_0, qui n'est reconnu par aucun circuit de taille $n^k \log n$.

En symboles, sur l'entrée x les considérations précédentes s'expriment donc comme suit :

$$\exists C_0 \quad \forall C, C_0' \quad \exists y, C' \quad \forall y' \\ \Big(C(y) \neq C_0(y) \quad \wedge \quad (C_0' \geqslant_{\text{lex}} C_0 \quad \vee \quad C'(y') = C_0'(y')) \quad \wedge \quad C_0(x) = 1 \Big),$$

ce qui définit un langage $L \in \Sigma_4^{\mathsf{p}}$. Par définition, pour tout n suffisamment grand, aucun circuit de taille $n^k \log n$ ne reconnaît $L^{=n}$, donc $L \notin \mathsf{SIZE}(n^k)$.

Il s'agit maintenant de se ramener à Σ_2^{p}. Si $\mathsf{NP} \not\subset \mathsf{P/poly}$ alors a fortiori $\Sigma_2^{\mathsf{p}} \not\subset \mathsf{SIZE}(n^k)$. Sinon, par le théorème 8-P de Karp et Lipton, $\mathsf{PH} = \Sigma_2^{\mathsf{p}}$ donc $L \in \Sigma_2^{\mathsf{p}}$. □

Dans le même genre mais plus difficile, on peut montrer que la classe PP n'admet pas non plus de circuits de taille n^k pour un k fixé. Avec le théorème 9-AI de Toda, à première vue le résultat peut sembler moins fort que l'énoncé précédent portant sur Σ_2^{p}, mais en réalité on ne sait pas comparer directement les deux classes Σ_2^{p} et PP. Le résultat de Vinodchandran [Vin05] qui suit combine de manière subtile plusieurs techniques, certaines issues notamment des protocoles interactifs. Nous nous appuyons fortement sur la proposition 11-L de la section suivante, qui montre que $\mathsf{PP} \subset \mathsf{P/poly} \implies \mathsf{PH} = \mathsf{MA}$ et qui concentre la difficulté technique du résultat.

11-F Théorème (Vinodchandran, 2005)

Pour toute constante k, $\mathsf{PP} \not\subset \mathsf{SIZE}(n^k)$.

Démonstration Si $\mathsf{PP} \not\subset \mathsf{P/poly}$ alors bien sûr $\mathsf{PP} \not\subset \mathsf{SIZE}(n^k)$.

Sinon, la proposition 11-L implique $\mathsf{PH} = \mathsf{MA}$, donc par le théorème 11-E de Kannan, $\mathsf{MA} \not\subset \mathsf{SIZE}(n^k)$. Puisque $\mathsf{MA} \subseteq \mathsf{PP}$ (cf. exercice 10-V), le résultat suit. □

11.1.2 Bornes inférieures en $n^{\omega(1)}$

À la section précédente nous avons vu des langages qui n'avaient pas de circuits de taille n^k pour une constante k fixée. Or si l'on veut séparer certaines classes étudiées précédemment (P et NP par exemple), les familles de circuits à considérer sont celles de taille polynomiale, c'est-à-dire la classe P/poly où la borne polynomiale n'est pas fixée. Dans cette direction malheureusement, on ne connaît pas de résultat très satisfaisant. En guise d'échauffement, nous proposons l'exercice suivant qui se résout d'une manière similaire à la proposition 11-A.

✎ **11-G** **Exercice**

On note $\mathsf{EXPSPACE} = \mathsf{DSPACE}(2^{n^{O(1)}})$. Montrer que $\mathsf{EXPSPACE} \not\subset \mathsf{P/poly}$.

Outre un théorème de Buhrman, Fortnow et Thierauf [BFT98] montrant que la version exponentielle $\mathsf{MA}_{\mathsf{EXP}}$ de MA n'est pas dans P/poly et qui s'appuie sur la proposition 11-J (cf. exercice 11-K), l'un des meilleurs résultats reste la mise à l'échelle ci-dessous du théorème 11-E de Kannan. En particulier, il est encore ouvert de savoir si $\mathsf{NEXP}^{\mathsf{RP}}$ a des circuits de taille polynomiale.

11-H **Proposition (Kannan, 1981)**

$$\mathsf{NEXP}^{\mathsf{NP}} \not\subset \mathsf{P/poly}$$

Idée de la démonstration Il suffit d'adapter la preuve du théorème 11-E au cas exponentiel, où l'on peut quantifier et diagonaliser sur les circuits de taille $n^{\log n}$.
Pour cela il faut également adapter la preuve du théorème 8-P de Karp et Lipton au cas de la « hiérarchie exponentielle » : dans notre cas, il suffit de montrer que $\mathsf{NP} \subset \mathsf{P/poly} \implies \mathsf{NEXP}^{\mathsf{NP}^{\mathsf{NP}}} = \mathsf{NEXP}^{\mathsf{NP}}$ (en effet, l'adaptation du théorème 11-E ne requiert que trois quantificateurs exponentiels, ceux sur les circuits, car les autres portant sur les mots peuvent être remplacés par une énumération des mots de taille n). □

✎ **11-I** **Exercice**

Donner les détails de la démonstration de la proposition précédente.

Nous nous tournons maintenant vers des bornes inférieures « conditionnelles » de la forme « si telle hypothèse probable est vraie, alors telle classe n'a pas de circuits de taille polynomiale ». Les quatre propositions suivantes montrent qu'il est improbable que PSPACE, PP et EXP aient des circuits de taille polynomiale car cela impliquerait l'égalité de classes

généralement considérées différentes. Nous nous servirons de l'un de ces résultats au chapitre suivant.

11-J Proposition

$$\mathsf{PSPACE} \subset \mathsf{P/poly} \implies \mathsf{PSPACE} = \mathsf{MA}$$

Idée de la démonstration On sait que $\mathsf{IP} = \mathsf{PSPACE}$ (théorème 10-G) et on peut supposer dans le protocole IP que le prouveur valide est dans PSPACE (lemme 10-I). Il possède donc d'après l'hypothèse des circuits de taille polynomiale. On en déduit le protocole AM suivant pour un langage de PSPACE : Merlin envoie le circuit pour le prouveur et Arthur simule le protocole IP en calculant les réponses du prouveur grâce au circuit.

Démonstration Soit $L \in \mathsf{PSPACE}$ et montrons que $L \in \mathsf{MA}$. Les lemmes 10-J et 10-I montrent qu'il existe un protocole IP pour L entre un vérificateur V et un prouveur, dans lequel le prouveur valide P fonctionne en espace polynomial. Par hypothèse, il existe donc une famille de circuits de taille polynomiale calculant la fonction du prouveur valide (ce sont des circuits à plusieurs sorties puisqu'ils calculent des fonctions, dont l'existence découle du fait que le calcul de chaque bit de la sortie est un langage de PSPACE).

Voici donc un protocole de type MA pour L :

- Merlin envoie un circuit C censé calculer les réponses de P (le prouveur valide du protocole IP pour L) ;
- Arthur simule le protocole IP en calculant les réponses de P grâce au circuit C. Arthur accepte ssi le vérificateur accepte.

Si $x \in L$ alors en donnant le bon circuit C, Merlin s'assure qu'Arthur accepte avec grande probabilité puisqu'il simule correctement le protocole IP avec le prouveur valide. Si $x \notin L$, alors tout prouveur ne convainc le vérificateur qu'avec faible probabilité : c'est en particulier le cas du prouveur simulé par le circuit C, donc Arthur rejette avec grande probabilité. On en déduit que $L \in \mathsf{MA}$. □

✎ **11-K Exercice**

On appelle $\mathsf{MA}_{\mathsf{EXP}}$ la classe des langages reconnus par un protocole de type MA où le vérificateur est une machine probabiliste fonctionnant en temps exponentiel $2^{n^{O(1)}}$ (en particulier, les messages échangés peuvent être de taille exponentielle).

Montrer que $\mathsf{MA}_{\mathsf{EXP}} \not\subset \mathsf{P/poly}$ (résultat de [BFT98]).

> *Indication : montrer par padding que* PSPACE = MA *implique* EXPSPACE = $\mathsf{MA}_{\mathsf{EXP}}$, *puis utiliser le résultat de l'exercice 11-G, et conclure grâce à la proposition 11-J.*

En combinant l'idée de la preuve précédente avec le théorème 9-AI de Toda, on obtient le résultat suivant dû à Babai, Fortnow et Lund [BFL91], et qui nous a servi à montrer que PP n'a pas de circuits de taille n^k (théorème 11-F).

11-L Proposition (Babai, Fortnow et Lund, 1991)

Si PP ⊂ P/poly alors PH = MA.

Idée de la démonstration Si PP ⊂ P/poly alors le prouveur du protocole IP de la proposition 10-P pour le langage `Maj3SAT` a des circuits de taille polynomiale (car il est dans P^{PP}). Pour décider `Maj3SAT` dans MA, il suffit alors que Merlin donne un tel circuit et qu'Arthur simule le protocole IP en calculant les réponses du prouveur grâce au circuit donné par Merlin.

Afin de décider un langage de PH, on combine ce protocole avec le théorème 9-AI de Toda qui donne une sorte de « réduction » d'un langage de PH à `Maj3SAT`.

Démonstration Puisque MA ⊆ PH (corollaire 10-AB), il suffit de montrer que PH ⊆ MA. Soit $L \in \mathsf{PH}$: la démonstration du théorème 9-AI de Toda donne une « réduction » f calculable en temps polynomial, qui à toute entrée x et tout mot r de taille polynomiale associe $f(x,r) \in \{0,1\}$, et des entiers N, k tels que $N < 2^k$ et :

$$\begin{cases} x \in L \implies \sum_r f(x,r) \in [0,N] \mod 2^k ; \\ x \notin L \implies \sum_r f(x,r) \in [N+1, 2^k-1] \mod 2^k. \end{cases}$$

On note

$$B_{\geqslant} = \{(x,a) \mid \sum_r f(x,r) \geqslant a2^k\},$$
$$B_{<} = \{(x,a) \mid \sum_r f(x,r) < (a+1)2^k\} \text{ et}$$
$$B = \{(x,a) \mid \sum_r f(x,r) \leqslant N + a2^k\} :$$

ces trois langages sont dans PP (cf. proposition 9-L) et si l'on connaît l'entier a tel que $(x,a) \in B_{\geqslant} \cap B_{<}$ alors il suffit de décider si $(x,a) \in B$ pour savoir si $x \in L$. Pour cela, dans un protocole MA Merlin va donner cet entier a, ainsi qu'un circuit pour les langages PP. On note $g_{\geqslant}$, $g_{<}$ et g les réductions respectives de $B_{\geqslant}$, $B_{<}$ et B à `Maj3SAT`. Si PP ⊂ P/poly alors le prouveur valide du protocole IP de la proposition 10-P pour le langage `Maj3SAT` a des circuits de taille polynomiale puisqu'il est dans $\mathsf{FP}^{\mathsf{PP}}$. Dans ce protocole, sur toute entrée de taille n on note $p(n)$ un polynôme majorant la taille des requêtes à l'oracle PP dans le fonctionnement du prouveur. On obtient alors le protocole MA suivant pour `Maj3SAT` sur l'entrée x :

– Merlin envoie un entier a et un circuit C censé décider `Maj3SAT` pour des entrées de taille au plus

$$\max(p(|g_{\geqslant}(x,a)|), p(|g_{<}(x,a)|), p(|g(x,a)|));$$

– Arthur simule trois fois le protocole IP pour `Maj3SAT`, en calculant les réponses du prouveur grâce au circuit C, afin de décider si $g_{\geqslant}(x,a), g_{<}(x,a), g(x,a) \in$ `Maj3SAT` : Arthur accepte ssi le vérificateur a accepté lors de ces trois protocoles.

Si $x \in L$ alors il existe a tel que $g_{\geqslant}(x,a), g_{<}(x,a), g(x,a) \in$ `Maj3SAT` : il suffit à Merlin d'envoyer un tel a et un circuit C pour `Maj3SAT` afin de convaincre Arthur avec grande probabilité puisque ce dernier simule alors correctement le prouveur valide qui parvient à convaincre le vérificateur. En revanche, si $x \notin L$ alors au moins l'une des trois instances $g_{\geqslant}(x,a), g_{<}(x,a), g(x,a)$ est hors de `Maj3SAT` : pour une telle instance de `Maj3SAT`, tout prouveur ne convainc le vérificateur qu'avec faible probabilité et c'est en particulier le cas du prouveur simulé grâce au circuit C, donc Arthur rejette avec grande probabilité.

On en déduit que $L \in \mathsf{MA}$ et donc que $\mathsf{PH} \subseteq \mathsf{MA}$. □

Nous passons maintenant à la classe EXP, avec un résultat préliminaire que nous allons ensuite améliorer par le corollaire qui le suit. Ce premier résultat est donné dans l'article de Karp et Lipton [KL82] et attribué à Meyer.

11-M **Proposition (Meyer, 1982)**

$$\mathsf{EXP} \subset \mathsf{P/poly} \implies \mathsf{EXP} = \Sigma_2^{\mathsf{p}}$$

Idée de la démonstration Si M est une machine pour un langage L de EXP, par l'hypothèse il existe des circuits de taille polynomiale calculant sur l'entrée t chaque bit de la configuration de $M(x)$ au temps t. Pour décider L dans PH, il suffit alors de deviner un tel circuit, de vérifier qu'il est cohérent avec la fonction de transition de M, et de tester si l'état final est acceptant. On se ramène ensuite à Σ_2^{p} grâce au théorème 8-P de Karp et Lipton.

Démonstration Soit $L \in \mathsf{EXP}$ et M une machine déterministe reconnaissant L en temps 2^{n^k}. En particulier, la fonction qui à (x,t,i) où t et i sont des entiers donnés en binaire, associe l'état de $M(x)$, la position des têtes de lecture et les symboles lus à l'étape t, et le contenu de la case i au temps t sur chaque ruban, est calculable en temps exponentiel. Puisque $\mathsf{EXP} \subset \mathsf{P/poly}$ par hypothèse, il existe une famille de circuits de taille polynomiale qui la calcule (ce sont ici des circuits à plusieurs sorties).

On peut alors décider L dans PH : il suffit de deviner le circuit correspondant à la taille de l'entrée, de vérifier qu'il calcule bien ce qu'il doit calculer, et enfin de tester

que l'état final est acceptant. Plus formellement, voici les différents points à vérifier concernant le circuit C :

- initialisation : au temps $t = 0$, $M(x)$ est dans l'état initial, les têtes en position 1 et les rubans sont vides sauf le ruban d'entrée qui contient x ;
- respect des transitions de M : à chaque temps t, les têtes écrivent, se déplacent et leur état évolue conformément à la fonction de transition δ_M de M ;
- cohérence des rubans : pour tout t, le contenu de chaque case au temps $t+1$ est le même qu'au temps t si la tête de lecture n'est pas sur la case au temps t ;
- acceptation : au temps $t = 2^{n^k}$, l'état est acceptant.

On obtient donc une formule pour décider L de la forme :

$$\exists C \quad (\forall i, C(x,0,i) \text{ renvoie le bon symbole pour la case } i) \wedge$$

$$(\forall t, C(x,t,0) \text{ et } C(x,t+1,0) \text{ décrit un comportement conforme à } \delta_M) \wedge$$

$$(\forall t, \forall i, C(x,t,i) \text{ et } C(x,t+1,i) \text{ donnent le même symbole à la case } i \text{ si la tête est ailleurs}) \wedge$$

$$(C(x,2^{n^k},0) \text{ renvoie un état acceptant}).$$

Tester ces conditions se fait dans PH. En ayant deviné le bon circuit, on décide bien L, donc $L \in \mathsf{PH}$.

Puisque $\mathsf{NP} \subseteq \mathsf{EXP} \subset \mathsf{P/poly}$, par le théorème 8-P de Karp et Lipton on a $\mathsf{PH} = \Sigma_2^{\mathsf{p}}$, ce qui conclut. □

Nous avons maintenant les outils pour renforcer la proposition précédente. Ce résultat vient de [BFL91].

11-N Corollaire

$$\mathsf{EXP} \subset \mathsf{P/poly} \implies \mathsf{EXP} = \mathsf{MA}$$

Démonstration Sous l'hypothèse $\mathsf{EXP} \subset \mathsf{P/poly}$, la proposition 11-M donne déjà l'égalité $\mathsf{EXP} = \Sigma_2^{\mathsf{p}}$, donc en particulier $\mathsf{EXP} = \mathsf{PSPACE}$. L'hypothèse implique en outre $\mathsf{PSPACE} \subset \mathsf{P/poly}$, donc par la proposition 11-J on a $\mathsf{PSPACE} = \mathsf{MA}$. □

11.2 Circuits restreints

Si on ne sait pas montrer de bornes inférieures satisfaisantes dans le cas des circuits booléens généraux, en revanche en imposant des restrictions naturelles sur les circuits on parvient à des résultats beaucoup plus forts.

11.2.1 Monotonie

La première restriction que nous mentionnons est l'absence de portes de négation dans nos circuits. En effet, pour calculer des fonctions booléennes monotones, on peut se restreindre à des circuits booléens monotones.

11-O Définition

– Une fonction $f : \{0,1\}^n \to \{0,1\}$ est *monotone* si pour tous $x, y \in \{0,1\}^n$,

$$(\forall i \in [1,n], x_i \leqslant y_i) \implies f(x) \leqslant f(y).$$

– Un circuit booléen est *monotone* s'il ne contient pas de porte de négation $\neg$.

11-P Remarques

– La notion introduite correspond plutôt à la notion habituelle de fonction *croissante* (et non seulement monotone), mais nous conservons ici la terminologie habituelle.

– On remarquera qu'un circuit monotone ne peut calculer que des fonctions monotones et que toute fonction monotone peut être calculée par un circuit monotone (de taille éventuellement exponentielle).

La fonction $\texttt{CLIQUE}_{n,k}$ désigne la fonction $\{0,1\}^{n(n-1)/2} \to \{0,1\}$ prenant en entrée la partie triangulaire supérieure de la matrice d'adjacence d'un graphe non orienté G à n sommets, et qui vaut 1 ssi G possède une clique de taille k. C'est évidemment une fonction monotone puisque ajouter une arête à G ne peut pas supprimer une clique existante. Sur l'entrée (n,k,G), calculer $\texttt{CLIQUE}_{n,k}(G)$ est un problème NP-complet (cf. corollaire 3-AG) ; cependant, on ne sait pas montrer de borne inférieure non linéaire sur la taille des circuits nécessaires. En revanche, en imposant que les circuits soient monotones, on peut faire beaucoup mieux : Razborov [Raz85] a montré la première borne inférieure pour le calcul de $\texttt{CLIQUE}_{n,k}$ par des circuits monotones, qui a ensuite été améliorée par Alon et Boppana [AB87] pour obtenir le résultat suivant. Pour une preuve de ce résultat, nous renvoyons au livre d'Arora et Barak [AB09].

11-Q Théorème (Razborov 1985, Alon et Boppana 1987)

Il existe une constante $\epsilon > 0$ telle que pour tout $k \leqslant n^{1/4}$, tout circuit monotone pour $\texttt{CLIQUE}_{n,k}$ a une taille $\geqslant 2^{\epsilon\sqrt{k}}$.

11.2.2 Profondeur constante

Une autre restriction concerne la profondeur des circuits. Nous allons voir que des familles de circuits de profondeur constante ne sont pas très puissantes, même lorsque les circuits sont munis de portes modulo.

11-R Définition

Soit $m \geqslant 2$ un entier.

- Dans un circuit booléen, une porte MOD_m a un degré entrant arbitraire, vaut 0 si le nombre de ses entrées valant 1 est congru à 0 modulo m, et vaut 1 sinon.
- La classe $\mathsf{ACC}^0(m)$ est l'ensemble des langages reconnus par des circuits booléens de type AC^0 (cf. définition 5-AK) munis de portes MOD_m.
  En d'autres termes, un langage L est dans $\mathsf{ACC}^0(m)$ s'il possède une famille de circuits de taille polynomiale et de profondeur constante, munis de portes MOD_m et dont les portes $\wedge$ et $\vee$ sont aussi de degré entrant arbitraire.
- Enfin, on appelle ACC^0 la classe $\mathsf{ACC}^0 = \bigcup_{m\geqslant 2}\mathsf{ACC}^0(m)$.

La notation MOD_m désignera également la fonction booléenne $\mathrm{MOD}_m : \{0,1\}^\star \to \{0,1\}$ valant 0 ssi la somme de ses arguments est congrue à 0 modulo m. On pourra aussi voir cette fonction booléenne comme un langage. Avant d'aborder le résultat principal, on a besoin d'un lemme technique facile.

11-S Lemme

Soit q un nombre premier, $\mathbb{F}_q$ désigne le corps à q éléments (les entiers modulo q). Soit $\alpha_1, \ldots, \alpha_m \in \{0,1\} \subseteq \mathbb{F}_q$. On désigne par S un sous-ensemble aléatoire de $[1,m]$, où chaque entier de $[1,m]$ est choisi indépendamment avec probabilité $1/2$.

S'il existe $i \in [1,m]$ tel que $\alpha_i = 1$, alors

$$\Pr_S\Big(\sum_{i\in S} \alpha_i \neq 0 \text{ sur } \mathbb{F}_q\Big) \geqslant 1/2.$$

Démonstration Soit $i_0 \in [1,m]$ tel que $\alpha_{i_0} = 1$. Puisque i_0 appartient à S avec probabilité $1/2$, on a :

$$\Pr\Big(\sum_{i\in S} \alpha_i \neq 0\Big) = (1/2)\Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq -1 \mid i_0 \in S\Big) + (1/2)\Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq 0 \mid i_0 \notin S\Big).$$

Or les événements « $\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq -1$ » et « $i_0 \in S$ » sont indépendants, donc

$$\Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq -1 \mid i_0 \in S\Big) = \Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq -1\Big).$$

De même,

$$\Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq 0 \mid i_0 \notin S\Big) = \Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq 0\Big).$$

On conclut la preuve en remarquant que

$$\Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq -1\Big) + \Pr\Big(\sum_{i\in S\setminus\{i_0\}} \alpha_i \neq 0\Big) \geqslant 1.$$

□

En s'inspirant de Razborov [Raz87] qui a montré une borne inférieure similaire pour la fonction « majorité » (et non MOD_p), Smolensky [Smo87] a obtenu le résultat suivant.

11-T Théorème (Razborov 1987, Smolensky 1987)

Soit p et q deux entiers premiers distincts. Alors $\mathrm{MOD}_p \notin \mathsf{ACC}^0(q)$.

Idée de la démonstration Pour d et ϵ bien choisis, nous allons montrer que pour tout circuit $\mathsf{ACC}^0(q)$ calculant une fonction booléenne $f : \{0,1\}^n \to \{0,1\}$, il existe un polynôme de degré d sur $\mathbb{F}_q$ (le corps à q éléments) qui coïncide avec f sur une fraction $(1-\epsilon)$ des entrées de $\{0,1\}^n$, alors qu'aucun polynôme de degré d sur $\mathbb{F}_q$ ne coïncide avec MOD_p sur $(1-\epsilon)$ des entrées de $\{0,1\}^n$.

La première partie se fait en « approximant » le circuit porte par porte par un polynôme, tandis que la seconde partie montre qu'un polynôme de degré n (ce qu'est MOD_p) ne peut pas être bien « approximé » par un polynôme de degré $\sqrt{n}/2$.

Démonstration Nous approximons d'abord tout circuit $\mathsf{ACC}^0(q)$ par un polynôme sur le corps à q éléments $\mathbb{F}_q$ (les entiers modulo q), avant de montrer que la fonction MOD_p ne peut être approximée ainsi.

Approximation d'un circuit par un polynôme Soit k un entier que l'on déterminera plus tard. Sur le corps $\mathbb{F}_q$, le polynôme $Q \in \mathbb{F}_q[x]$ défini par $Q(x) = x^{q-1}$ est de degré $(q-1)$ et vérifie :

$$Q(x) = \begin{cases} 0 & \text{si } x = 0, \\ 1 & \text{sinon.} \end{cases}$$

Soit C un circuit de type $\mathsf{ACC}^0(q)$ de taille t_C et de profondeur h_C, à n entrées. On suppose sans perte de généralité que C ne possède pas de porte $\wedge$ puisque celles-ci peuvent être remplacées par la négation d'une porte $\vee$ dont toutes les entrées sont niées ($y_1 \wedge \cdots \wedge y_m \equiv \neg(\neg y_1 \vee \cdots \vee \neg y_m)$), ce qui au pire triple la profondeur du circuit. Pour chaque porte α de C à la profondeur h, nous allons montrer qu'il existe un polynôme $P_\alpha \in \mathbb{F}_q[x_1, \ldots, x_n]$ de degré au plus $((q-1)k)^h$ tel que :

– pour tout $x \in \{0,1\}^n$, $P_\alpha(x) \in \{0,1\} \subseteq \mathbb{F}_q$;

– pour au moins $(1-t_C^h/2^k)2^n$ mots $x \in \{0,1\}^n$, on a $P_\alpha(x) = \alpha(x) \in \{0,1\}$ (où $\alpha(x)$ désigne la valeur de la porte α lorsque l'entrée du circuit C est x). (On dira que P_α fait une erreur sur x lorsque $P_\alpha(x) \neq \alpha(x)$.)

Démonstration — Nous procédons par induction sur h. Pour $h = 0$, la porte α est une entrée x_i donc il suffit de prendre $P_\alpha(x) = x_i$ et P_α ne fait aucune erreur. Pour $h > 0$, il y a plusieurs cas selon le type de la porte α.

– Porte $\alpha = \neg\alpha'$: le polynôme $P_\alpha = 1 - P_{\alpha'}$ convient puisqu'il vaut $\alpha(x)$ sur toutes les entrées où $P_{\alpha'}(x) = \alpha'(x)$.

– Porte $\alpha = \mathrm{MOD}_q(\alpha_1, \ldots, \alpha_m)$: on pose $P_\alpha = Q(\sum_{i=1}^m P_{\alpha_i})$. Puisque par hypothèse de récurrence, le degré de chaque P_{α_i} est majoré par $((q-1)k)^{h-1}$, le degré de P_α est majoré par $(q-1)((q-1)k)^{h-1} \leqslant ((q-1)k)^h$. Si sur l'entrée x, aucune des portes α_i ne fait d'erreur, alors $P_\alpha(x) = \alpha(x)$ par définition de Q.
Concernant les erreurs cette fois, bien qu'il puisse éventuellement y avoir plus de t_C entrées à la porte α (c'est-à-dire $m > t_C$), on remarquera qu'il y a au plus t_C portes distinctes en entrée et donc au plus t_C ensembles distincts de mots x sur lesquels les α_i font des erreurs, et ceux-ci sont de taille au plus $(t_C^{h-1}/2^k)2^n$ par hypothèse de récurrence. Ainsi, P_α fait une erreur sur une fraction au plus $t_C \cdot t_C^{h-1}/2^k = t_C^h/2^k$ des entrées x.

– Porte $\alpha = \bigvee_{i=1}^m \alpha_i$: la solution naïve de définir pour P_α le polynôme $1 - \prod_{i=1}^m (1 - P_{\alpha_i})$ ne convient pas car elle fait exploser le degré. Il faut donc être plus subtil.
Pour x fixé, en prenant k sous-ensembles aléatoires indépendants

$$S_1, \ldots, S_k \subseteq [1, m],$$

le lemme 11-S montre que

$$\Pr_{S_1,\ldots,S_k}\big(\alpha(x) \equiv \bigvee_{j=1}^k (\sum_{i\in S_j} \alpha_i(x) \neq 0)\big) \geqslant 1 - 2^{-k}.$$

Ainsi, pour chaque x la proportion des ensembles $S_1, \ldots, S_k$ « mauvais pour x » est majorée par 2^{-k}. En prenant l'union sur tous les mots x, chaque choix de $S_1, \ldots, S_k$ est mauvais en moyenne pour $\leqslant 2^{n-k}$ mots x. Il existe donc un choix de $S_1, \ldots, S_k$ fixé qui est mauvais pour au plus une proportion 2^{-k} des mots x :

$$\Pr_{x\in\{0,1\}^n}\big(\alpha(x) \equiv \bigvee_{j=1}^k (\sum_{i\in S_j} \alpha_i(x) \neq 0)\big) \geqslant 1 - 2^{-k}.$$

On pose alors

$$P_\alpha = 1 - \prod_{j=1}^k \big(1 - Q(\sum_{i\in S_j} P_{\alpha_i})\big),$$

c'est-à-dire le OU des valeurs « $\sum_{i\in S_j} P_{\alpha_i} \neq 0$ ». Puisque le degré des P_{α_i} est majoré par $((q-1)k)^{h-1}$, le degré de P_α est au plus

$$k(q-1)((q-1)k)^{h-1} = ((q-1)k)^h.$$

Par ce qui précède, lorsque les P_{α_j} ne se trompent pas sur x, alors P_α se trompe avec probabilité au plus 2^{-k}. On remarquera que $m \leqslant t_C - 1$ sans perte de généralité, et que par hypothèse de récurrence, la probabilité d'erreur des P_{α_j} est majorée par $t_C^{h-1}/2^k$. Au total, la probabilité d'erreur de P_α est donc majorée par $2^{-k} + (t_C - 1)t_C^{h-1}/2^k \leqslant t_C^h/2^k$.

La propriété est donc montrée au rang h. ◇

On en déduit qu'il existe un polynôme $P_C \in \mathbb{F}_q[x_1,\dots,x_n]$ de degré au plus $((q-1)k)^{h_C}$ tel que :

- pour tout $x \in \{0,1\}^n$, $P_C(x) \in \{0,1\}$;
- pour au moins $(1 - t_C^{h_C}/2^k)2^n$ mots $x \in \{0,1\}^n$, on a $P_C(x) = C(x)$.

Impossibilité d'approximer MOD_p [1] Si l'on dispose de la fonction MOD_p, il est facile de calculer par un circuit de profondeur constante les fonctions $\mathrm{MOD}_{i,p} : \{0,1\}^n \to \{0,1\}$, qui valent 0 ssi la somme de leurs entrées est congrue à i modulo p. En effet, pour calculer $\mathrm{MOD}_{i,p}(a_1,\dots,a_n)$ il suffit d'appliquer $\mathrm{MOD}_p(a'_1,\dots,a'_p,a_{p+1},\dots,a_n)$ où $(a'_1,\dots,a'_p)$ a été obtenu à partir de $(a_1,\dots,a_p)$ en passant i zéros en uns si $(a_1,\dots,a_p)$ contient au moins i zéros, ou sinon $p-i$ uns en zéros. Une telle opération se fait par un circuit de taille constante puisque p est une constante.
Il suffit donc de montrer qu'on ne peut pas calculer toutes les fonctions $\mathrm{MOD}_{i,p}$ par des circuits de type $\mathsf{ACC}^0(q)$, et pour cela de montrer qu'on ne peut pas toutes les approximer par des polynômes de petit degré.
L'idée va être d'utiliser de supposés polynômes de petit degré calculant les fonctions $\mathrm{MOD}_{i,p}$ sur un ensemble E, pour calculer grâce à un polynôme de degré $\leqslant \sqrt{n}/2$ une variante de la fonction $x \in \{0,1\}^n \mapsto (|x|_1 \mod p)$ (le nombre modulo p de composantes à 1 dans x). Or cette fonction correspond en réalité à un monôme de degré n, qu'on pourra donc remplacer par ce polynôme de degré $\sqrt{n}/2$ et ainsi abaisser le degré de tout polynôme : cela permettra de majorer de manière indirecte le cardinal de E.
Supposons donc que pour tout $i \in [0, p-1]$ il existe un polynôme $P_i \in \mathbb{F}_q[x_1,\dots,x_n]$ de degré $\sqrt{n}/2$ qui coïncide avec la fonction $\mathrm{MOD}_{i,p}$ sur un ensemble $E_i \subseteq \{0,1\}^n$: la suite de la preuve consiste à majorer $|E_i|$. On note $E = \cap_{i=0}^{p-1} E_i$ l'ensemble commun où aucun P_i ne fait « d'erreur ».
Il nous faut nous placer sur un corps de caractéristique q plus grand que $\mathbb{F}_q$. Soit $K = \mathbb{F}_{q^{p-1}}$ le corps à q^{p-1} éléments : son groupe multiplicatif $K^\star$ est cyclique et contient

1. Se restreindre au cas $p = 2$ et $q = 3$ permettrait de simplifier la preuve qui suit en allégeant notamment les considérations algébriques. Nous avons néanmoins pris le parti de traiter le cas général et renvoyons le lecteur au livre d'Arora et Barak s'il se satisfait du cas $p = 2$ et $q = 3$.

$q^{p-1}-1$ éléments. Ainsi, p divise $|K^*|$ car $q^{p-1} \equiv 1 \mod p$ (on se sert ici du fait que p est premier pour appliquer le petit théorème de Fermat) : K contient donc une racine primitive p-ème de l'unité, c'est-à-dire un élément $\omega \in K$ satisfaisant $\omega^p = 1$ et $\forall i \in [1, p-1], \omega^i \neq 1$. On définit un polynôme $P' \in K[x_1, \ldots, x_n]$ par

$$P'(x_1, \ldots, x_n) = \sum_{i=0}^{p-1} (1 - P_i(x_1, \ldots, x_n))\omega^i,$$

c'est-à-dire que pour tout $x \in E$, si le nombre de 1 dans x est congru à k modulo p alors $P'(x) = \omega^k$. Le degré de P' est majoré par celui des P_i, c'est-à-dire $\sqrt{n}/2$.
Enfin, on pose $y_i = (\omega - 1)x_i + 1$ de sorte que y_i vaut 1 si $x_i = 0$ et ω si $x_i = 1$, et on définit $R \in K[x_1, \ldots, x_n]$ par

$$R(x_1, \ldots, x_n) = \prod_{i=1}^{n} y_i :$$

ainsi, $R(x_1, \ldots, x_n)$ vaut ω^k si le nombre de composantes à 1 dans $x \in \{0,1\}^n$ est congru à k modulo p. Donc P' et R coïncident sur E, mais l'un est de degré $\sqrt{n}/2$ alors que l'autre est de degré n. On va se servir de cette égalité afin de diminuer le degré de polynômes exprimant des fonctions de E dans K.
On dit qu'un polynôme est *multilinéaire* si le degré de chacune de ses variables est au plus 1. Nous allons montrer que toute fonction $f : E \to K$ peut être exprimée par un polynôme multilinéaire de degré au plus $\lceil n/2 \rceil + \sqrt{n}/2 - 1$ (et puisqu'il y a « peu » de tels polynômes, cela impliquera que E est nécessairement « petit »).

Démonstration — Toute fonction $f : E \to K$ peut bien sûr être exprimée par un polynôme $F \in K[x_1, \ldots, x_n]$ en interpolant f sur tous les points de E. Puisque $E \subseteq \{0,1\}^n$, on peut supposer sans perte de généralité que le degré de chaque variable dans F est majoré par 1 car pour tout $k > 1$, $x_i^k = x_i$. En particulier, chaque monôme de F est de degré au plus n.
On exprime maintenant F en fonction des $y_i = (\omega - 1)x_i + 1$, c'est-à-dire qu'on remplace dans F chaque variable x_i par $(\omega - 1)^{-1}(y_i - 1)$. Puisque la substitution est linéaire, le degré de chaque y_i dans F est toujours au plus 1.
Soit m un monôme de F en $y_1, \ldots, y_n$ de degré $\geqslant n/2+1$, $m = \prod_{i \in I} y_i$ avec $I \subseteq [1, n]$ un ensemble de taille $|I| \geqslant n/2+1$. Alors pour $x \in E$, le polynôme suivant prend les mêmes valeurs que m, où ${}^cI$ désigne le complémentaire de I dans $[1, n]$:

$$\left(\prod_{i=1}^{n} y_i\right)\left(\prod_{i \in {}^cI} y_i^{p-1}\right),$$

car y_i^{p-1} vaut y_i^{-1}. Or sur E, le polynôme y_i^{p-1} exprimé en x_i, à l'origine de degré $(p-1)$ en x_i, peut être ramené à un polynôme en x_i de degré 1 (toujours car $\forall k > 1$, $x_i^k = x_i$ sur E), donc $\prod_{i \in {}^cI} y_i^{p-1}$ est ramené sur E à un polynôme en $x_1, \ldots, x_n$ de degré

$$|{}^cI| \leqslant n - (n/2+1) = n/2 - 1 \leqslant \lceil n/2 \rceil - 1.$$

De plus, le polynôme $\prod_{i=1}^{n} y_i$ vaut $R(x_1,\dots,x_n)$ qui est égal à $P'(x_1,\dots,x_n)$ sur E. On peut donc exprimer le monôme m par le produit du polynôme P' de degré $\sqrt{n}/2$ et d'un polynôme de degré $\leqslant \lceil n/2\rceil - 1$, c'est-à-dire au final par un polynôme en $x_1,\dots,x_n$ de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$.
Les monômes m qui n'ont pas été remplacés dans le processus sont ceux qui étaient déjà de degré $< n/2+1 \leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$, donc le polynôme complet est bien de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$. Puisqu'on travaille sur $E \subseteq \{0,1\}^n$, on peut supposer à nouveau que ce polynôme est multilinéaire. ◇

Or un polynôme multilinéaire de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$ est décrit par la liste des coefficients (éventuellement nuls) de chaque monôme multilinéaire de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$. Si N est le nombre de tels monômes, il y a donc $|K|^N$ polynômes multilinéaires de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$ dans $K[x_1,\dots,x_n]$. Nous allons montrer que $N \leqslant (19/20)2^n$.

Démonstration — Pour caractériser un monôme multilinéaire m en les variables $x_1,\dots,x_n$ et de degré $\leqslant \lceil n/2\rceil + \sqrt{n}/2 - 1$, il suffit de donner l'ensemble des variables qui apparaissent dans m. On a donc :

$$N = \sum_{i=0}^{\lceil n/2\rceil+\sqrt{n}/2-1} \binom{n}{i} \leqslant 2^{n-1} + \sum_{i=\lceil n/2\rceil}^{\lceil n/2\rceil+\sqrt{n}/2-1} \binom{n}{i} \leqslant 2^{n-1} + \frac{\sqrt{n}}{2}\binom{n}{\lceil n/2\rceil}$$

(où la seconde inégalité est obtenue en majorant chaque coefficient binomial de la somme par $\binom{n}{\lceil n/2\rceil}$). Or le corollaire A-Q de la formule A-P de Stirling donne :

$$\binom{n}{\lceil n/2\rceil} \leqslant \frac{e2^n}{\pi\sqrt{n}},$$

donc

$$N \leqslant 2^{n-1} + \frac{e}{2\pi}2^n = (\frac{1}{2} + \frac{e}{2\pi})2^n \leqslant (19/20)2^n.$$

◇

Puisque le nombre de fonctions $f : E \to K$ est égal à $|K|^{|E|}$, on déduit que $|K|^{|E|} \leqslant |K|^N$, donc $|E| \leqslant N \leqslant (19/20)2^n$. Enfin, puisque $E = \cap_{i=0}^{p-1} E_i$, il existe $i \in [0, p-1]$ tel que

$$|E_i| \leqslant (1 - 1/(20p))2^n.$$

Conclusion En choisissant $k = \log(40pt^h)$ dans la première partie, tout circuit C de type $\mathsf{ACC}^0(q)$ de profondeur h et de taille t coïncide avec un polynôme P_C de degré $\leqslant (q\log(40pt^h))^h$ sur une proportion au moins $(1-1/(40p))$ des entrées. Or pour une famille $\mathsf{ACC}^0(q)$, $t = t(n)$ est polynomial et $h = h(n)$ est constant, donc le degré de P_C est polylogarithmique en n et en particulier inférieur à $\sqrt{n}/2$ pour n suffisamment grand.
En revanche, grâce à la seconde partie l'une des fonctions $\mathrm{MOD}_{i,p}$ ne peut coïncider avec un polynôme de degré $\leqslant \sqrt{n}/2$ que sur une fraction au plus $(1 - 1/(20p))$ des entrées. On en déduit que la fonction $\mathrm{MOD}_{i,p}$ n'est pas dans $\mathsf{ACC}^0(q)$, et donc de même pour la fonction MOD_p. □

11-U Remarques

– La preuve donne en réalité une meilleure borne inférieure, puisqu'on peut choisir pour k toute valeur vérifiant $((q-1)k)^h \leqslant \sqrt{n}/2$ et $t^h/2^k < 1/(20p)$. Ainsi par exemple, des circuits de type $\mathsf{ACC}(q)$ de profondeur $O(\log\log n)$ et de taille polynomiale ne peuvent pas calculer MOD_p. De même pour une profondeur constante et une taille $2^{n^{o(1)}}$.

– En revanche, pour la classe $\mathsf{ACC}^0 = \bigcup_{m\geqslant 2}\mathsf{ACC}^0(m)$, on est loin d'une telle borne inférieure puisque le meilleur résultat connu est dû à Williams [Wil11] : $\mathsf{NEXP} \not\subset \mathsf{ACC}^0$.

✎ **11-V Exercice**

On rappelle que $\mathsf{ACC}^0 = \bigcup_{m\geqslant 2}\mathsf{ACC}^0(m)$ et que TC^0 est la classe des langages reconnus par circuits de taille polynomiale et de profondeur constante ayant des portes de majorité (cf. définition 5-AK). En particulier on a $\mathsf{TC}^0 \subseteq \mathsf{L/poly}$ par la proposition 5-AM.

Montrer que $\mathsf{ACC}^0 \subseteq \mathsf{TC}^0$.

Indication : on pourra simuler une porte MOD_m *par un nombre polynomial de portes de majorité « en parallèle » (au même niveau dans le circuit), auxquelles on ajoute un certain nombre d'entrées constantes* 0 *ou* 1.

11.3 Polynômes

Des techniques intéressantes ont également été développées pour montrer des bornes inférieures sur la taille de circuits arithmétiques pour calculer des polynômes (pour un rappel sur ces circuits, voir la section 5.6). Nous allons voir un résultat de Baur et Strassen [BS83] montrant que le polynôme $\sum_{i=1}^{n} x_i^n$ n'a pas de circuits arithmétiques de taille $o(n\log n)$, puis un résultat de Shnorr [Sch78] (améliorant des articles de Strassen [Str74] et de Lipton [Lip75]) montrant l'existence de polynômes à une variable, de degré polynomial et à coefficients dans $\{0,1\}$, qui n'ont pas de circuits de taille n^k.

Une des difficultés dans le cas des circuits arithmétiques provient du fait qu'il peuvent utiliser des constantes arbitraires du corps sous-jacent (ici $\mathbb{C}$), et donc que les méthodes de dénombrement ou de diagonalisation usuelles ne fonctionnent plus.

11.3.1 Baur et Strassen

Afin de monter le résultat de Baur et Strassen, nous allons d'abord voir la *degree bound* de Strassen [Str73a] par une démonstration de Schönage [Sch76]. Il s'agit de donner

une borne inférieure sur la taille d'un circuit calculant plusieurs polynômes. Puis nous verrons comment le calcul des dérivées partielles (par une démonstration de Morgenstern [Mor85]) nous permettra de revenir au calcul d'un seul polynôme.

Borne du degré

Dans cette section, K désigne un corps infini quelconque (par exemple $\mathbb{C}$). Avant de prouver le résultat principal, nous avons besoin de plusieurs lemmes sur l'existence de relations polynomiales entre des polynômes calculés par circuit. En guise d'échauffement, nous rappelons le nombre de monômes d'un certain degré.

11-W Lemme

Le nombre de monômes de degré $\leqslant d$ en $x_1,\dots,x_n$ est $\binom{n+d}{d}$.

Démonstration Un monôme en $x_1,\dots,x_n$ de degré d est vu comme un tableau de $n+d$ cases dans lesquelles sont placés n délimiteurs : le nombre de cases libres entre le délimiteur $i-1$ et le délimiteur i est le degré de x_i (on suppose qu'il y a un délimiteur 0 « virtuel » avant la première case). Pour dénombrer ces monômes, il s'agit donc de placer n délimiteurs dans un tableau de $(n+d)$ cases, c'est-à-dire qu'il y a $\binom{n+d}{n}$ possibilités. □

Le lemme suivant montre l'existence d'un polynôme annulateur, la partie difficile étant de faire en sorte que le degré en y_0 *ne dépende pas* de celui de p_0.

11-X Lemme

Si $p_0,p_1,\dots,p_m \in K[x_1,\dots,x_m]$ sont des polynômes de degré $\deg(p_i)=d_i$, alors il existe un polynôme non nul $H \in K[y_0,\dots,y_m]$ tel que $\deg_{y_0}(H) \leqslant D = d_1 \times \cdots \times d_m$ et $H(p_0,\dots,p_m)=0$.

Démonstration Soit d un entier suffisamment grand (qui sera une borne sur le degré total de $H(p_0,\dots,p_m)$). Écrivons un polynôme $H(y_0,\dots,y_m)=\sum_v c_v y_0^{v_0}\dots y_m^{v_m}$, où $v=(v_0,\dots,v_m)\in\mathbb{N}^{m+1}$ vérifient :

$$v_0 \leqslant D \qquad \text{et} \qquad v_0 d_0 + \cdots + v_m d_m \leqslant d. \tag{11.1}$$

Comptons le nombre de tels v.
Tout d'abord, dénombrons les solutions de

$$v_1 d_1 + \cdots + v_m d_m \leqslant q \tag{11.2}$$

pour un certain $q \leqslant d$. À une solution $u=(u_1,\dots,u_m)$ de l'équation

$$u_1 + \cdots + u_m \leqslant q \tag{11.3}$$

on associe la solution v de l'équation 11.2 définie par $v_i = \lfloor u_i/d_i \rfloor$. Tout v a au plus D antécédents (correspondant aux choix des restes $0 \leqslant r_i < d_i$ dans les divisions entières de u_i par d_i). Puisqu'il y a $\binom{q+m}{m}$ solutions à l'équation 11.3 par le lemme 11-W, il y en a au moins $\binom{q+m}{m}/D$ à l'équation 11.2.
Pour notre application, $q = d - v_0 d_0$. Or

$$\binom{m+(d-v_0d_0)}{m} \sim \frac{d^m}{m!}$$

quand d tend vers l'infini (ici, m, d_0, v_0 et D sont des constantes). Puisqu'on peut choisir v_0 entre 0 et D, on en déduit que le nombre de solutions de 11.1 est au moins

$$\sum_{v_0=0}^{D} \binom{m+(d-v_0d_0)}{m}/D \sim \frac{(D+1)d^m}{Dm!}.$$

Pour d grand, le polynôme H a donc plus de $(1+1/(2D))d^m/m!$ monômes.
Soit maintenant $G(x_1,\dots,x_m) = H(p_0,\dots,p_m)$: le degré de G est au plus d puisque $v_0d_0+\cdots+v_md_m \leqslant d$, donc G a au plus $\binom{d+m}{m} \sim d^m/m!$ coefficients par le lemme 11-W. Pour d grand, le nombre de coefficients du polynôme G est donc majoré par $(1+1/(2D))d^m/m!$.
Chaque coefficient de G est une combinaison linéaire des coefficients c_v. La contrainte $H(p_0,\dots,p_m) = 0$ se traduit donc en moins de $(1+1/(2D))d^m/m!$ équations linéaires en les coefficients c_v (le coefficient de chaque monôme en $x_1,\dots,x_m$ doit être nul). Ce système linéaire a une solution non nulle car le nombre d'inconnues est plus grand que le nombre d'équations. □

En appliquant le lemme précédent au cas de polynômes calculés par un circuit à plusieurs sorties, on peut maîtriser le degré du polynôme annulateur en fonction de la taille du circuit.

11-Y Lemme

Soit C un circuit arithmétique de taille t, calculant n polynômes

$$p_1,\dots,p_n \in K[x_1,\dots,x_n].$$

Alors pour tout polynôme $p_0 \in K[x_1,\dots,x_n]$, il existe un polynôme non nul $H \in K[y_0,\dots,y_n]$ tel que $\deg_{y_0}(H) \leqslant 2^t$ et $H(p_0,\dots,p_n) = 0$.

Démonstration On numérote les portes de C de 1 à t de sorte que les $n+k$ premières portes soient les variables $x_1,\dots,x_n$ et les constantes $\alpha_1,\dots,\alpha_k \in K$ du circuit, et on introduit t nouvelles variables $z_1,\dots,z_t$. Pour tout i entre 1 et $(t+n)$, on définit un polynôme $f_i \in K[x_1,\dots,x_n,z_1,\dots,z_t]$ (afin d'avoir « moralement » $f_i = z_i - p_i$ où p_i est le polynôme calculé à la porte i) de la manière suivante :

– pour $1 \leqslant i \leqslant n$ (portes correspondant aux variables $x_1,\dots,x_n$), $f_i = z_i - x_i$;

– pour $n+1 \leqslant i \leqslant n+k$ (portes correspondant aux constantes $\alpha_1,\dots,\alpha_k$), $f_i = z_i - \alpha_{i-n}$;
– pour $n+k+1 \leqslant i \leqslant t$, si la porte i est une opération $\circ \in \{+,\times\}$ d'arguments j et j', alors $f_i = z_i - (z_j \circ z_{j'})$;
– enfin, pour $i > t$, soit j_{i-t} la porte calculant p_{i-t} : on définit alors $f_i = z_{j_{i-t}}$.

Au plus t de ces polynômes ont un degré 2, les autres ayant un degré $\leqslant 1$. Par le lemme 11-X, il existe donc un polynôme non nul $G \in K[y_0,\dots,y_{t+n}]$ tel que

$$\deg_{y_0}(G) \leqslant 2^t \quad \text{et} \quad G(p_0,f_1,\dots,f_{t+n}) = 0.$$

On peut voir $G(p_0,f_1,\dots,f_{t+n})$ comme un polynôme de $(K[x_1,\dots,x_n])[z_1,\dots,z_t]$. Alors, par définition des polynômes f_i, remarquons que, si $g_i(x_1,\dots,x_n)$ est le polynôme calculé par la porte i, on a :

$$[G(p_0,f_1,\dots,f_t,f_{t+1},\dots,f_{t+n})](g_1,\dots,g_t) = G(p_0,0,\dots,0,p_1,\dots,p_n) = 0,$$

l'égalité portant sur des polynômes de $K[x_1,\dots,x_n]$ (puisque $g_i \in K[x_1,\dots,x_n]$). Ainsi, $H(y_0,\dots,y_n) = G(y_0,0,\dots,0,y_1,\dots,y_n)$ est le polynôme recherché. □

Afin de pouvoir énoncer le résultat suivant, nous avons besoin d'une définition.

11-Z Définition

On dit que *presque tout* point $x \in K^n$ vérifie une propriété s'il existe un polynôme non nul $f \in K[x_1,\dots,x_n]$ tel que x vérifie la propriété dès que $f(x) \neq 0$ (c'est-à-dire que la propriété est vraie partout sauf éventuellement sur une hypersurface).

Voici maintenant le résultat principal de cette sous-section : la *degree bound* de Strassen [Str73a].

11-AA Théorème (Strassen, 1973)

Soit des polynômes $p_1,\dots,p_n \in \mathbb{C}[x_1,\dots,x_n]$ et un entier $d \geqslant 1$ tels que pour presque tout point $v = (v_1,\dots,v_n) \in \mathbb{C}^n$, le système d'équations polynomiales d'inconnues $(x_1,\dots,x_n)$, $S_v = (p_i(x_1,\dots,x_n) = v_i)_{1\leqslant i\leqslant n}$, a au moins d solutions distinctes sur $\mathbb{C}^n$. Alors tout circuit calculant les n polynômes $p_1,\dots,p_n$ a une taille au moins $\log(d)$.

Démonstration Soit C un circuit de taille t calculant $p_1,\dots,p_n$. On introduit n nouvelles variables $z_1,\dots,z_n$ et on considère le calcul de C sur le corps $K = \mathbb{C}(z_1,\dots,z_n)$. Soit $p_0 = z_1x_1 + \cdots + z_nx_n \in K[x_1,\dots,x_n]$: le lemme 11-Y fournit un polynôme $H \in K[y_0,\dots,y_n]$ tel que $\deg_{y_0}(H) \leqslant 2^t$ et $H(p_0,\dots,p_n) = 0$, qu'on peut écrire

$$H(y_0,\dots,y_n) = \sum_{i=0}^{M} q_i(y_1,\dots,y_n)y_0^i,$$

où $M \leqslant 2^t$, $q_i \in K[y_1,\dots,y_n]$ et $q_M \neq 0$. Ainsi, le polynôme $P \in K[x_1,\dots,x_n]$ défini par $P(x) = H(p_0,\dots,p_n)(x)$ est nul ; en d'autres termes :

$$P(x) = \sum_{i=0}^{M} q_i(p_1(x),\dots,p_n(x))(\sum_{j=1}^{n} z_j x_j)^i = 0.$$

Soit $f \in \mathbb{C}[x_1,\dots,x_n]$ un polynôme tel le système S_v admette au moins d solutions pour tout point $v \in \mathbb{C}^n$ vérifiant $f(v) \neq 0$, et soit $\alpha = (\alpha_1,\dots,\alpha_n) \in \mathbb{C}^n$ tel que $f(\alpha) \neq 0$ et $q_M(\alpha) \neq 0$. Par hypothèse, le système d'équations $(p_i(x) = \alpha_i)_{i\in[1,n]}$ a au moins d solutions distinctes $a^{(1)},\dots,a^{(d)} \in \mathbb{C}^n$: pour tous $i \in [1,n]$ et $j \in [1,d]$, $p_i(a^{(j)}) = \alpha_i$. Soit $b^{(i)} = p_0(a^{(i)}) = \sum_j z_j a_j^{(i)} \in K$: on a, pour tout k,

$$P(a^{(k)}) = \sum_{i=0}^{M} q_i(p_1(a^{(k)}),\dots,p_n(a^{(k)}))(\sum_{j=1}^{n} z_j a_j^{(k)})^i = \sum_{i=0}^{M} q_i(\alpha_1,\dots,\alpha_n)(b^{(k)})^i = 0.$$

Ainsi, les $b^{(i)}$ sont d racines distinctes du polynôme

$$h(x) = \sum_{j=0}^{M} q_j(\alpha) x^j \in K[x]$$

qui est non nul car $q_M(\alpha) \neq 0$, donc $d \leqslant M \leqslant 2^t$, c'est-à-dire $t \geqslant \log d$. □

Dérivées partielles

Nous avons vu comment montrer une borne inférieure sur la taille d'un circuit arithmétique calculant plusieurs polynômes simultanément ; nous allons maintenant voir comment se ramener au calcul d'un seul polynôme, en utilisant ses dérivées partielles. Pour éviter toute confusion, on appellera *porte d'opération* toute porte qui n'est pas une entrée dans un circuit arithmétique (c'est-à-dire ni une variable ni une constante).

11-AB Lemme

Soit $p(x_1,\dots,x_n) \in K[x_1,\dots,x_n]$ un polynôme calculé par un circuit arithmétique ayant t portes d'opération. Alors il existe un circuit arithmétique à n sorties ayant $\leqslant 5t$ portes d'opération, les mêmes entrées (variables et constantes) et les constantes 0 et 1, calculant toutes les dérivées partielles $\frac{\partial p}{\partial x_i}(x_1,\dots,x_n)$ pour $i \in [1,n]$.

Démonstration Par récurrence sur t. Soit $C(x_1,\dots,x_n,\alpha_1,\dots,\alpha_k)$ un circuit arithmétique à t portes d'opérations calculant p, où les $\alpha_i \in K$ sont des constantes. Pour $t = 0$, soit $p = x_i$ soit $p = \alpha_i$ et dans les deux cas on calcule ses dérivées partielles par un circuit ayant 0 porte d'opération (puisqu'il s'agit des constantes 0 et 1).

Pour $t \geqslant 1$: soit g une porte dont les deux arguments sont des entrées (constantes ou variables). Sans perte de généralité, on peut considérer qu'au moins l'un des arguments

est une variable : en effet, si les deux arguments sont des constantes α, β, on peut remplacer la porte g par une nouvelle constante $g(\alpha, \beta)$ tout en faisant diminuant la taille du circuit.

On considère le circuit C' ayant $(t-1)$ portes d'opérations et dont les entrées sont $x_1, \ldots, x_n, \alpha_1, \ldots, \alpha_k, g$. Ce circuit calcule un polynôme $q(x_1, \ldots, x_n, y)$ (où y est la nouvelle variable créée par la nouvelle entrée g). On a : $p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n, g)$ et donc

$$\frac{\partial p}{\partial x_i}(x_1, \ldots, x_n) = \frac{\partial q}{\partial x_i}(x_1, \ldots, x_n, g) + \frac{\partial q}{\partial y}(x_1, \ldots, x_n, g)\frac{\partial g}{\partial x_i}.$$

Par induction, il existe un circuit D' à $(n+1)$ sorties, ayant $\leqslant 5t-5$ portes d'opérations, calculant $\frac{\partial q}{\partial x_i}$ (pour $1 \leqslant i \leqslant n$) et $\frac{\partial q}{\partial y}$. Nous allons maintenant considérer les différents cas possibles pour la porte g pour construire un circuit D calculant toutes les dérivées partielles de p.

1. Addition d'une constante et d'une variable, $g(x_a, \alpha_b) = \alpha_b + x_a$: pour tout $i \neq a$, $\frac{\partial g}{\partial x_i} = 0$ et donc $\frac{\partial p}{\partial x_i} = \frac{\partial q}{\partial x_i}$. Pour $i = a$: $\frac{\partial p}{\partial x_a} = \frac{\partial q}{\partial x_a} + \frac{\partial q}{\partial y}$. Pour calculer toutes les dérivées partielles de p, il suffit donc d'ajouter une porte à D' (une addition) ; par ailleurs, il faut également calculer g, ce qui ajoute une porte. La taille de D est donc $\leqslant 5t-3$.
2. Multiplication d'une constante et d'une variable, $g(x_a, \alpha_b) = \alpha_b x_a$: pour tout $i \neq a$, $\frac{\partial g}{\partial x_i} = 0$ et donc $\frac{\partial p}{\partial x_i} = \frac{\partial q}{\partial x_i}$. Pour $i = a$: $\frac{\partial p}{\partial x_a} = \frac{\partial q}{\partial x_a} + \alpha_b \frac{\partial q}{\partial y}$. Pour calculer toutes les dérivées partielles de p, il suffit donc d'ajouter 2 portes (une multiplication et une addition) ; par ailleurs, il faut également calculer g, ce qui ajoute une porte. La taille de D est donc $\leqslant 5t-2$.
3. Addition de deux variables, $g(x_a, x_b) = x_a + x_b$: pour tout $i \neq a, b$, $\frac{\partial g}{\partial x_i} = 0$ et donc $\frac{\partial p}{\partial x_i} = \frac{\partial q}{\partial x_i}$. Pour $i = a$: $\frac{\partial p}{\partial x_a} = \frac{\partial q}{\partial x_a} + \frac{\partial q}{\partial y}$, de même pour $i = b$. Pour calculer toutes les dérivées partielles de p, il suffit donc d'ajouter 2 portes (deux additions) ; par ailleurs, il faut également calculer g, ce qui ajoute une porte. La taille de D est donc $\leqslant 5t-2$.
4. Multiplication de deux variables, $g(x_a, x_b) = x_a x_b$: pour tout $i \neq a, b$, $\frac{\partial g}{\partial x_i} = 0$ et donc $\frac{\partial p}{\partial x_i} = \frac{\partial q}{\partial x_i}$. Pour $i = a$: $\frac{\partial p}{\partial x_a} = \frac{\partial q}{\partial x_a} + x_b \frac{\partial q}{\partial y}$, de même pour $i = b$. Pour calculer toutes les dérivées partielles de p, il suffit donc d'ajouter 4 portes (deux multiplications et deux additions) ; par ailleurs, il faut également calculer g, ce qui ajoute une porte. La taille de D est donc $\leqslant 5t$.

Dans tous les cas, il existe un circuit D ayant $\leqslant 5t$ portes d'opération, les mêmes entrées que C et les constantes 0 et 1, et calculant toutes les dérivées partielles de p. □

En comptant cette fois les entrées, on obtient le corollaire suivant.

11-AC **Corollaire**

Soit $p(x_1,\dots,x_n) \in K[x_1,\dots,x_n]$ un polynôme calculé par un circuit arithmétique de taille t. Alors il existe un circuit arithmétique de taille $\leqslant 5t$ à n sorties calculant toutes les dérivées partielles $\frac{\partial p}{\partial x_i}(x_1,\dots,x_n)$ pour $i \in [1,n]$.

Démonstration Soit C un circuit de taille t pour $p(x)$ et soit $m \geqslant 1$ le nombre d'entrées (variables et constantes). Alors le nombre de portes d'opérations est $t-m$, donc en comptant les constantes 0 et 1, le lemme 11-AB fournit un circuit de taille $\leqslant 2+m+5(t-m) = 5t-4m+2 \leqslant 5t$ pour les dérivées partielles. □

Le calcul des dérivées partielles combiné à la borne du degré permet de déduire des bornes inférieures sur le calcul d'un polynôme seul [BS83].

11-AD **Corollaire (Baur et Strassen, 1983)**

Tout circuit arithmétique sur $\mathbb{C}$ calculant le polynôme $p(x_1,\dots,x_n) = \sum_{i=1}^{n} x_i^d$ a une taille au moins $(n/5)\log(d-1)$.

Démonstration Soit t la taille d'un plus petit circuit pour p. Par le corollaire 11-AC, il existe un circuit de taille $\leqslant 5t$ calculant toutes les dérivées partielles de p, c'est-à-dire calculant $d x_i^{d-1}$ pour tout i.

Or $d x^{d-1} = \alpha \in \mathbb{C}$ a exactement $(d-1)$ racines distinctes, sauf pour $\alpha = 0$. Ainsi, pour presque tout $v \in \mathbb{C}^n$ (tous sauf les racines du polynôme $\prod_i x_i$), le système $(d x_i^{d-1} = v_i)_{i\in[1,n]}$ a $(d-1)^n$ solutions distinctes. Par le théorème 11-AA, $5t \geqslant \log((d-1)^n)$ donc $t \geqslant (n/5)\log(d-1)$. □

11.3.2 Borne inférieure en $\Omega(n^k)$

Nous présentons maintenant le résultat de Schnorr [Sch78] mentionné en début de section. Il parvient à montrer, comme dans le cas booléen, une borne inférieure en $\Omega(n^k)$ (pour k fixé) sur un polynôme univarié de degré $n^{O(1)}$. Certes $\Omega(n^k)$ est mieux que $\Omega(n\log n)$ (borne inférieure obtenue ci-dessus sur un polynôme de degré $n^{O(1)}$), mais contrairement au résultat 11-AD de Baur et Strassen, le polynôme est construit de manière ad-hoc et n'est pas « naturel ». Dans toute cette partie, les circuits ont des constantes arbitraires de $\mathbb{C}$: l'une des difficultés vient de la présence de ces constantes qui empêchent d'utiliser des arguments de dénombrement. Mais commençons par spécifier sur quel genre de polynôme doit porter la borne inférieure.

Bornes inférieures triviales

Certains polynômes n'ont pas de petits circuits pour des raisons évidentes et ils ne nous intéresseront pas ici :

- le polynôme $p(x) = x^{2^t}$ ne peut pas être calculé par des circuits de taille $< t$ car le degré maximal d'un polynôme calculé par un circuit de taille s est 2^s ;
- le polynôme $p(x_1, \ldots, x_t) = \sum_{i=1}^{t} x_i$ ne peut pas être calculé par des circuits de taille $< t$ car il a trop de variables ;
- si $\alpha_1, \ldots, \alpha_t \in \mathbb{C}$ sont algébriquement indépendants, alors le polynôme $p(x) = \sum_{i=1}^{t} \alpha_i x^i$ ne peut pas être calculé par des circuits de taille $< t$ car $\mathbb{Q}(p(1), \ldots, p(t))$ est de degré de transcendance t alors que ce degré est majoré par le nombre de constantes utilisées par le circuit.

Pour éviter ces raisons évidentes, nous voulons pour tout k une borne inférieure $\Omega(n^k)$ sur un polynôme p à une variable, de degré polynomial en n et à coefficients dans $\{0, 1\}$. Remarquons qu'un tel polynôme est toujours calculable par un circuit arithmétique de taille polynomiale.

Borne inférieure en n^k

Bien que la borne inférieure porte sur des polynômes univariés, nous utiliserons aussi pour les démonstrations des polynômes multivariés. Nous nous intéressons encore une fois à des relations polynomiales, mais cette fois entre les coefficients d'un polynôme calculé par un circuit.

11-AE Lemme

Pour tout $t \geqslant 1$, il existe une famille de polynômes $g_j \in \mathbb{Z}[y_1, \ldots, y_{t^2+2t-2}]$, $j \in \mathbb{N}$, $\deg(g_j) \leqslant 3tj + 1$, satisfaisant la propriété suivante.

Si un polynôme univarié $f(x)$ est calculé par un circuit C de taille au plus t alors il existe $\alpha_1, \ldots, \alpha_{t^2+2t-2} \in \mathbb{C}$ tels que pour tout j le coefficient de x^j dans $f(x)$ est $g_j(\alpha_1, \ldots, \alpha_{t^2+2t-2})$.

Démonstration On numérote à partir de 1 les portes non constantes de C de manière compatible avec leur profondeur (en particulier, la porte étiquetée par la variable x reçoit le numéro 1). Quitte à augmenter la taille de C, on suppose que t est le numéro de la porte de sortie de C (la dernière dans la numérotation). On désignera par $y = (y_{k,i})_{k\in[0,i-1],i\in[2,t]}$, $y' = (y'_{k,i})_{k\in[0,i-1],i\in[2,t]}$ et $z = (z_1, \ldots, z_t)$ des uples de nouvelles variables. Pour $i \geqslant 1$ et $j \geqslant 0$, on va définir des polynômes $h_i \in \mathbb{C}[x, y, y']$ pouvant simuler le calcul de la porte i, et $h_{i,j} \in \mathbb{C}[y, y', z]$ sera le coefficient de x^j dans h_i : on prendra alors $g_j = h_{t,j}$. Pour l'initialisation, on définit $h_0 = 1$. Afin de maîtriser le degré, on traite les coefficients constants à part : pour tout i on pose $h_{i,0} = z_i$ (z_i représente donc le coefficient constant de la porte i).
On pose $h_1 = x$, polynôme calculé par la porte 1 (qui n'est autre que la variable x), donc $h_{1,1} = 1$ et $h_{1,j} = 0$ pour $j > 1$. Puis par récurrence on définit pour $i > 1$

$$h_i = \Big(\sum_{k=0}^{i-1} y_{k,i} h_k\Big)\Big(\sum_{k=0}^{i-1} y'_{k,i} h_k\Big),$$

et donc pour $j > 0$:

$$h_{i,j} = \sum_{j_1+j_2=j} \sum_{k_1,k_2<i} y_{k_1,i} y'_{k_2,i} h_{k_1,j_1} h_{k_2,j_2}.$$

On a $h_i \in \mathbb{Z}[x,y,y']$ et $h_{i,j} \in \mathbb{Z}[y,y',z]$ (les variables z_k représentant les coefficients constants).
Il existe des valeurs $\alpha \in \mathbb{C}^{|y|}$ et $\alpha' \in \mathbb{C}^{|y'|}$ pour les y et y' telles que $h_i(x,\alpha,\alpha')$ soit le polynôme calculé par la porte i :

- si la porte i est une addition $h_{i_1} + h_{i_2}$, il suffit de prendre $\alpha_{i_1,i} = \alpha_{i_2,i} = 1$, $\alpha'_{0,i} = 1$ et le reste à 0 ;
- si la porte i est une multiplication $h_{i_1} h_{i_2}$, il suffit de prendre $\alpha_{i_1,i} = \alpha'_{i_2,i} = 1$ et le reste à 0 ;
- si la porte i est une multiplication par une constante, γh_{i_1}, il suffit de prendre $\alpha_{i_1,i} = \gamma$, $\alpha'_{0,i} = 1$ et le reste à 0 ;
- enfin, si la porte i est une addition avec une constante, $\gamma + h_{i_1}$, il suffit de prendre $\alpha_{0,i} = \gamma$, $\alpha_{i_1,i} = 1$, $\alpha'_{0,i} = 1$ et le reste à 0.

Si par ailleurs on donne aux variables z_i la valeur β_i du coefficient constant de la porte i, alors $h_{i,j}(\alpha,\alpha',\beta)$ est le coefficient de x^j de la porte i.
Les variables de $h_{t,j}$ sont $y_{i,k}$ et $y'_{i,k}$ pour $2 \leqslant i \leqslant t$ et $0 \leqslant k < i$, ainsi que z_i pour $1 \leqslant i \leqslant t$, donc $h_{t,j}$ a $(t-1)(t+2)+t = t^2+2t-2$ variables.
En outre, on montre par récurrence sur i que $\deg(h_{i,j}) \leqslant 3ij+1$: c'est vrai pour $i = 1$ et, $\forall i$, pour $j = 0$. Pour $i > 1$ et $j > 0$:

$$\begin{aligned}\deg(h_{i,j}) &\leqslant 2 + \max_{j_1+j_2=j;k_1,k_2<i} (\deg(h_{k_1,j_1}) + \deg(h_{k_2,j_2})) \\ &\leqslant 2 + 3(i-1)j + 2 = 3ij + 4 - 3j \leqslant 3ij + 1.\end{aligned}$$

Ainsi, $g_j = h_{t,j}$ vérifie la propriété demandée. □

Nous avons besoin d'un lemme qui n'est pas sans rappeler le lemme 11-X, mais nous devons ici avoir un contrôle sur le degré total du polynôme H.

11-AF Lemme

Soit $t \geqslant 14$ un entier. Si $g_1, \ldots, g_{t^3} \in \mathbb{Z}[x_1, \ldots, x_{t^2+2t-2}]$, $\deg(g_i) \leqslant 3t^4 + 1$, alors il existe un polynôme non nul $H \in \mathbb{Z}[y_1, \ldots, y_{t^3}]$, $\deg(H) \leqslant t^4$, tel que $H(g_1, \ldots, g_{t^3}) = 0$.

Démonstration On écrit le système que doivent vérifier les coefficients

$$\alpha = (\alpha_{i_1,\ldots,i_{t^3}})_{i_1+\cdots+i_{t^3} \leqslant t^4}$$

du polynôme

$$H = \sum_{i_1+\cdots+i_{t^3}\leqslant t^4} \alpha_{i_1,\ldots,i_{t^3}} y_1^{i_1}\ldots y_{t^3}^{i_{t^3}} :$$

on doit avoir

$$H(g_1,\ldots,g_{t^3})(x) = \sum_{i_1,\ldots,i_{t^3}} \alpha_{i_1,\ldots,i_{t^3}} g_1(x)^{i_1}\ldots g_{t^3}(x)^{i_{t^3}} = 0.$$

Le coefficient de chaque monôme en $x_1,\ldots,x_{t^2+2t-2}$ est une combinaison linéaire des α, donc on a un système linéaire en α. Le nombre d'inconnues est au moins t^{t^3} puisque chaque i_j peut prendre n'importe quelle valeur entre 0 et t.
Par ailleurs, le nombre d'équations est égal au nombre de monômes en $x_1,\ldots,x_{t^2+2t-2}$: puisqu'il y a t^2+2t-2 variables x_i et que le degré de chaque monôme est majoré par

$$\sum_{j=1}^{t^3} i_j \deg(g_j) \leqslant t^3 \times t^4 \times (3t^4+1) < t^{12},$$

il y a au plus $t^{12(t^2+2t-2)}$ monômes en $x_1,\ldots,x_{t^2+2t-2}$.
Il y a plus d'inconnues que d'équations, donc le système linéaire admet une solution non nulle. □

Les deux lemmes précédents nous donnent le résultat suivant.

11-AG Corollaire

Pour tout $t \geqslant 14$, il existe un polynôme non nul $H \in \mathbb{Z}[y_1,\ldots,y_{t^3}]$, $\deg(H) \leqslant t^4$, tel que pour tout $f(x) = \sum_{i=1}^{t^3} \gamma_i x^i$ calculé par un circuit de taille t, $H(\gamma_1,\ldots,\gamma_{t^3}) = 0$.

Démonstration Par le lemme 11-AE, il existe $\alpha \in \mathbb{C}^{t^2+2t-2}$ tel que $\gamma_i = g_i(\alpha)$, où les g_i sont des polynômes vérifiant $\deg(g_i) \leqslant 3t^4+1$ pour $1 \leqslant i \leqslant t^3$. Par le lemme 11-AF, il existe H tel que $H(g_1,\ldots,g_{t^3}) = 0$. En évaluant $H(g_1,\ldots,g_{t^3})$ en α, on obtient $H(\gamma_1,\ldots,\gamma_{t^3}) = 0$. □

Enfin, le lemme suivant montre qu'un polynôme non nul ne peut pas s'annuler en tous les points de petite valeur.

11-AH Lemme

Soit un polynôme non nul $H \in \mathbb{Z}[y_1,\ldots,y_{t^3}]$ tel que $\deg(H) \leqslant t^4$. Alors il existe $a_1,\ldots,a_{t^3} \in \{0,\ldots,t^4\}$ tel que $H(a_1,\ldots,a_{t^3}) \neq 0$.

Démonstration Il s'agit d'un résultat classique. Montrons par récurrence sur le nombre n de variables que si p est un polynôme non nul de degré d à n variables, alors il existe $a \in \{0,\ldots,d\}^n$ tel que $p(a) \neq 0$.

Pour $n = 1$, le nombre de racines ne peut être plus grand que le degré.
Pour $n > 1$: on peut écrire p sous la forme $p(x_1,\dots,x_n) = \sum_i p_i(x_2,\dots,x_n)x_1^i$ où les p_i sont des polynômes à $(n-1)$ variables de degré $\leqslant d-i$. Puisque p est non nul, il existe i_0 tel que p_{i_0} est non nul. Le degré de p_{i_0} est $\leqslant d$. Par hypothèse de récurrence, il existe $a_2,\dots,a_n \in \{0,\dots,d\}$ tel que $p_{i_0}(a_2,\dots,a_n) \neq 0$. Ainsi, le polynôme à une variable $q(x_1) = p(x_1,a_2,\dots,a_n)$ est non nul et de degré $\leqslant d$: il existe donc $a_1 \in \{0,\dots,d\}$ tel que $q(a_1) = p(a_1,\dots,a_n) \neq 0$. □

Nous sommes maintenant prêts pour la borne inférieure : en effet, du corollaire 11-AG et du lemme 11-AH on déduit le résultat suivant.

11-AI Corollaire

Pour tout $t \geqslant 14$, il existe $a_1,\dots,a_{t^3} \in \{0,\dots,t^4\}$ tel que le polynôme $\sum_{i=1}^{t^3} a_i x^i$ n'a pas de circuits de taille t.

Enfin, on peut ramener les coefficients dans $\{0,1\}$.

11-AJ Corollaire (Srassen 1974, Lipton 1975, Schnorr 1978)

Pour tout $t \geqslant 14$, il existe $b_1,\dots,b_{t^3} \in \{0,1\}$ tel que le polynôme $\sum_{i=1}^{t^3} b_i x^i$ n'a pas de circuits de taille $t/(4\log t)$.

En particulier, en choisissant $t = n^{k+1}$: pour tout $k \geqslant 1$ et $n \geqslant 4$, il existe $b_1,\dots,b_{n^{3(k+1)}} \in \{0,1\}$ tel que le polynôme $\sum_{i=1}^{n^{3(k+1)}} b_i x^i$ n'a pas de circuits de taille n^k.

Démonstration Soit $a_1,\dots,a_{t^3}$ les coefficients donnés par le corollaire 11-AI et $p(x) = \sum_{i=1}^{t^3} a_i x^i$. Soit

$$\sum_{j=0}^{m} a_{i,j} 2^j = a_i$$

la décomposition en base 2 des coefficients a_i, avec $m = \log(t^4) = 4\log t$ et $a_{i,j} \in \{0,1\}$. Alors

$$p(x) = \sum_{j=0}^{m} 2^j \sum_{i=1}^{t^3} a_{i,j} x^i$$

donc l'un des polynômes $\sum_{i=1}^{t^3} a_{i,j} x^i$ n'a pas de circuits de taille $t/(4\log t)$. □

✎ **11-AK Exercice**

Mener des calculs plus précis au lemme 11-AF pour annuler $O(t^2)$ polynômes g_i au lieu de t^3. En déduire que le corollaire 11-AJ est encore valide pour un polynôme $\sum_i b_i x^i$ de degré $O(t^2)$ (au lieu de t^3).

12 Dérandomisation et bornes inférieures

Nous avons vu dans les chapitres précédents que de nombreux types de questions restent ouverts en complexité. Deux d'entre eux vont particulièrement nous intéresser à ce chapitre. D'une part, l'utilisation de l'aléatoire permet-elle d'accélérer significativement les algorithmes ? Ce sont des questions du type « $\mathsf{EXP} = \mathsf{BPP}$? » : peut-on résoudre tout problème exponentiel en temps probabiliste polynomial ? Ou en d'autres termes, l'aléatoire peut-il accélérer exponentiellement le calcul ? Bien entendu, toute accélération superpolynomiale (et pas forcément exponentielle) serait déjà un apport décisif, mais ce genre de résultat est généralement considéré comme improbable. Vue sous l'angle opposé, la question revient à se demander si l'on peut transformer tout algorithme probabiliste en un algorithme déterministe sans perdre trop en temps de calcul : a-t-on par exemple $\mathsf{BPP} \subseteq \mathsf{DTIME}(n^{\log n})$ ou même $\mathsf{BPP} = \mathsf{P}$? C'est ce qu'on appelle la dérandomisation.

D'autre part, la non-uniformité peut-elle accélérer grandement le calcul ? C'est le domaine des bornes inférieures non uniformes étudié au chapitre précédent : montrer par exemple que $\mathsf{EXP} \subset \mathsf{P/poly}$, c'est montrer qu'en temps polynomial, la non-uniformité suffit à calculer tous les problèmes exponentiels. Là encore, on peut rechercher des accélérations plus modestes plutôt qu'une accélération exponentielle du temps de calcul : tout résultat de la forme $\mathsf{DTIME}(n^{f(n)}) \subset \mathsf{P/poly}$, pour une fonction f tendant vers l'infini, serait déjà un apport considérable, mais ce genre de résultat est habituellement considéré comme improbable car les bornes inférieures de type $\mathsf{EXP} \not\subset \mathsf{P/poly}$ sont généralement considérées comme vraies.

L'objet de ce chapitre est de montrer que ces deux questions, dérandomisation et bornes inférieures non uniformes, sont étroitement liées. Nous allons voir en effet qu'elles sont « presque équivalentes » : une borne inférieure de type $\mathsf{EXP} \not\subset \mathsf{P/poly}$ implique un résultat de dérandomisation de type $\mathsf{BPP} \subseteq \mathsf{DTIME}(2^{n^\epsilon})$ (pour toute constante $\epsilon > 0$), et renforcer l'hypothèse jusqu'à $\mathsf{E} \not\subset \mathsf{SIZE}(2^{\epsilon n})$ entraîne $\mathsf{BPP} = \mathsf{P}$ (cf. corollaire 12-AV). Réciproquement, dérandomiser un problème comme TIP (tester si un circuit arithmétique calcule un polynôme nul, cf. section 6.3.1) implique une borne inférieure, soit sur le permanent

soit sur NEXP (cf. théorème 12-BA).

Nous commençons par quelques considérations sur la dérandomisation avant de montrer que bornes inférieures non uniformes impliquent dérandomisation, ce qui remplira la majorité du chapitre et nous amènera à étudier les codes correcteurs d'erreurs. Nous terminerons par la démonstration que la dérandomisation de TIP implique un borne inférieure non uniforme.

12.1 Dérandomisation

Si jusqu'aux années 1980, les chercheurs pensaient que l'aléatoire pouvait significativement accélérer le calcul et que, par exemple, $\mathsf{BPP} \neq \mathsf{P}$, la vision générale a complètement changé dans les années 1990 lorsque sont apparus les résultats exposés dans ce chapitre. Il fallait en effet être cohérent, les bornes inférieures non uniformes de type $\mathsf{EXP} \not\subset \mathsf{P/poly}$ sont habituellement considérées comme vraies et celles-ci impliquent une dérandomisation…

L'avis général penche donc maintenant plutôt du côté de l'égalité entre BPP et P. De nombreuses dérandomisations de problèmes naturels sont venues étayer cet avis comme par exemple le problème PRIMALITÉ :

- *entrée :* un entier N donné en binaire ;
- *question :* N est-il premier ?

On connaît des algorithmes probabilistes efficaces pour ce problème depuis les années 1970 (notamment le test de Miller et Rabin [Mil76 ; Rab80]) mais il a fallu 25 ans avant de découvrir un algorithme déterministe polynomial [AKS04] :

12-A Théorème (Agrawal, Kayal et Saxena, 2004)

$$\texttt{PRIMALITÉ} \in \mathsf{P}$$

Un des rares problèmes naturels de BPP dont on ignore encore s'il est dans P est TIP (tester si un circuit arithmétique calcule un polynôme nul, cf. section 6.3.1).

Des résultats positifs sont également connus depuis longtemps dans le cas de la complexité en espace.

12-B Définition

La classe RPSPACE est l'ensemble des langages L tels qu'il existe une machine de Turing probabiliste M fonctionnant en espace polynomial et en temps exponentiel vérifiant pour tout $x \in \Sigma^\star$:

– si $x \in L$ alors $\Pr(M(x) \text{ accepte}) \geqslant 1/2$;
– si $x \notin L$ alors $\Pr(M(x) \text{ accepte}) = 0$.

Il est facile de voir que RPSPACE $\subseteq$ NPSPACE, et donc par le théorème 4-AS de Savitch, RPSPACE = PSPACE. Il s'agit donc d'une dérandomisation de la classe probabiliste RPSPACE.

En réalité, quelques techniques génériques existent pour dérandomiser des problèmes : réduire l'univers tout en étant capable de définir des variables aléatoires k à k indépendantes (et, si l'univers est suffisamment petit, le parcourir en entier pour résoudre le problème) ; la méthode des probabilités et des espérances conditionnelles (où il s'agit de calculer itérativement la probabilité de succès) ; les expandeurs, etc. Nous renvoyons le lecteur au livre d'Alon et Spencer [AS04] pour approfondir ce sujet.

Les expandeurs ont justement permis à Reingold [Rei05] de dérandomiser le problème d'accessibilité dans un graphe non orienté $\texttt{ACCESS}_{\text{n.o.}}$:

– *entrée :* un graphe non orienté G et deux sommets s et t ;
– *question :* s et t sont-ils reliés dans G ?

On connaissait depuis les années 1970 un algorithme probabiliste simple et fonctionnant en espace logarithmique pour ce problème : effectuer une marche aléatoire dans G à partir de s et accepter ssi on atteint t en moins de n^3 étapes (Aleliunas, Karp, Lipton, Lovász et Rackoff [Ale+79]). La dérandomisation suivante, connue sous le nom SL = L car SL désigne la classe des problèmes qui se réduisent à $\texttt{ACCESS}_{\text{n.o.}}$, a donc provoqué l'émoi de la communauté.

12-C Théorème (Reingold, 2005)

$$\texttt{ACCESS}_{\text{n.o.}} \in \mathsf{L}$$

Nous renvoyons au livre de Goldreich [Gol08] pour la preuve de ce théorème grâce aux expandeurs.

Malgré toutes ces avancées, aucune dérandomisation générique de la classe BPP, par exemple, ne semble à portée de main. Une des raisons est que cela impliquerait de montrer une borne inférieure non uniforme, une tâche réputée difficile. Nous exposerons ce résultat à la fin du chapitre.

Mais avant cela, nous montrons dans un premier temps que des bornes inférieures non uniformes impliquent une dérandomisation de BPP. Le chemin pour ce faire est tortueux.

Stratégie de démonstration Dans une première section, nous verrons qu'on peut dérandomiser BPP si l'on possède une fonction « imprédictible », c'est-à-dire dont aucun

circuit de petite taille ne peut prédire correctement les bits de sortie : pour ces circuits, une telle fonction est donc considérée comme « aléatoire » et on peut l'utiliser pour remplacer les bits aléatoires d'un algorithme probabiliste.

Puis nous verrons à la section suivante comment construire une fonction imprédictible si l'on possède une fonction f difficile *en moyenne*, c'est-à-dire qu'aucun circuit de petite taille ne calcule correctement f sur une proportion des entrées significativement plus grande que $1/2$. En évaluant f sur des sous-ensembles pseudo-aléatoires construits grâce au générateur de Nisan et Wigderson (définition 12-Q), avec grande probabilité chaque bit de la sortie est imprédictible.

Enfin, à la section suivante nous montrerons que l'existence d'une fonction g difficile *dans le pire cas* (c'est-à-dire une borne inférieure non uniforme comme nous le souhaitons) permet de construire une fonction difficile en moyenne. Il s'agit de la partie la plus technique et la plus longue car nous avons besoin de codes correcteurs d'erreurs très performants. Supposons en effet que l'on dispose d'un code E pour lequel on soit capable de retrouver le mot x codé lorsqu'on dispose d'un mot y qui coïncide avec $E(x)$ seulement sur la moitié des bits. En voyant une fonction comme un mot égal à sa table de vérité, on définit alors $g = E(f)$: si g' diffère de g sur moins de la moitié des entrées, alors à partir de g' on sait retrouver f. En d'autres termes, si f est difficile dans le pire cas, alors on ne peut pas calculer g sur plus de la moitié des entrées, donc g est difficile en moyenne.

Dans la suite, nous manipulerons donc de nombreuses fonctions. Par abus de notation mais pour aider la lisibilité, nous dirons qu'une fonction $f : \{0,1\}^\star \to \{0,1\}^\star$ est dans $\mathsf{DTIME}(t(n))$ si elle peut être calculée par une machine de Turing déterministe en temps $O(t(n))$, c'est-à-dire que nous étendons la définition de $\mathsf{DTIME}(t(n))$ aux fonctions. Nous noterons $f^{=n}$ la restriction de f à $\{0,1\}^n$.

12.2 Imprédictibilité implique dérandomisation

Informellement, une fonction est *imprédictible* si, pour deviner le bit $(i+1)$ de sa sortie lorsqu'on connaît les i premiers bits, on ne peut pas faire beaucoup mieux que de choisir au hasard.

12-D Définition (imprédictibilité)

Soit $m : \mathbb{N} \to \mathbb{N}$. Une fonction $f : \{0,1\}^\star \to \{0,1\}^\star$ est $m(n)$-*imprédictible* si

- pour tout $x \in \{0,1\}^\star$, $|f(x)| = m(|x|)$ et
- pour tout n, pour tout $i \in [0, m(n)-1]$ et pour tout circuit booléen C de taille $\leqslant m(n)^2$ à i entrées, on a :

$$\Pr_{z \in \{0,1\}^n : r = f(z)} \big(C(r_1, \ldots, r_i) = r_{i+1}\big) < 1/2 + 1/(6m(n)).$$

✎ **12-E** **Exercice**

Montrer que la fonction identité $\mathrm{id} : \{0,1\}^* \to \{0,1\}^*$ est n-imprédictible.

Puisqu'on ne sait prédire un bit de f en fonction des précédents, on peut utiliser f comme un générateur de bits « aléatoires », ce qui nous permet de montrer le théorème suivant de Yao [Yao82] (pour mémoire, la simulation déterministe évidente d'un algorithme probabiliste donne l'inclusion $\mathsf{BPTIME}\big(o(m(u(n)))\big) \subseteq \mathsf{DTIME}\big(2^{m(u(n))}\big)$).

12-F **Théorème (Yao, 1982)**

Pour toutes fonctions $m, t : \mathbb{N} \to \mathbb{N}$, s'il existe une fonction $m(n)$-imprédictible $f : \{0,1\}^* \to \{0,1\}^*$ calculable en temps $O(t(n))$, alors pour toute fonction $u : \mathbb{N} \to \mathbb{N}$ calculable en temps $O(2^{u(n)})$,

$$\mathsf{BPTIME}\big(o(m(u(n)))\big) \subseteq \mathsf{DTIME}\big(2^{u(n)} t(u(n))\big).$$

Plus précisément, si un langage de $\mathsf{BPTIME}\big(o(m(u(n)))\big)$ est reconnu par une machine de Turing probabiliste $M(x,r)$ où $|r| = m(u(|x|))$, alors pour tout mot x de taille suffisamment grande,

$$x \in L \iff \Pr_{z \in \{0,1\}^{u(|x|)}}(M(x, f(z)) = 1) \geqslant 1/2.$$

Idée de la démonstration Pour donner une intuition de la preuve, supposons que $|r| = |z| = |f(z)| = 2$ et que $M(x, f(z)) = 1$ avec probabilité $\geqslant 1/2$ alors que $M(x,r) = 1$ avec probabilité $1/4$. Il y a donc un unique mot r_0 tel que $M(x, r_0) = 1$, disons $r_0 = 00$, ce qui implique qu'il y a deux mots z_1, z_2 tels que $f(z_1) = f(z_2) = 00$, tandis que les mots z_3 et z_4 ont une image différente, disons $f(z_3) = 01$ et $f(z_4) = 11$. On remarque alors que le premier bit de $f(z)$ est nul avec probabilité $3/4$, donc « prédictible ». Mais que se passe-t-il si par exemple $f(z_3) = f(z_4) = 11$? Dans ce cas, le second bit de $f(z)$ est prédictible puisqu'il est toujours égal au premier. C'est cet exemple simpliste que nous allons généraliser.

Démonstration Pour alléger les notations, on note $N = u(n)$. Soit un langage $L \in \mathsf{BPTIME}\big(o(m(N))\big)$ et $M(x,r)$ une machine de Turing fonctionnant en temps $o(m(N))$ pour L, où le mot r désigne les bits aléatoires de M. L'idée est de remplacer les bits aléatoires $r \in \{0,1\}^{m(N)}$ de M par $f(z)$, où z parcourt tous les mots de taille N, et de prendre la réponse majoritaire. Cela revient à énumérer tous les $z \in \{0,1\}^N$ et pour chacun de calculer $f(z)$ et de simuler $M(x, f(z))$, ce qui donne un algorithme A déterministe fonctionnant en temps $O(2^N(t(N) + m(N))) = O(2^N t(N))$ (car $t \geqslant m$). Montrons que cet algorithme est correct. Par l'absurde, supposons que A fasse une erreur sur une certaine entrée $x_0 \in \{0,1\}^n$, c'est-à-dire que [1]

$$\Pr_{z \in \{0,1\}^N}(M(x_0, f(z)) = [x_0 \notin L]) \geqslant 1/2,$$

1. On rappelle que la notation $[x \in L]$ (respectivement $[x \notin L]$) désigne le booléen qui vaut 1 ssi $x \in L$ (resp. $x \notin L$).

alors que par définition de M, on a :

$$\Pr_{r\in\{0,1\}^{m(N)}}(M(x_0,r)=[x_0\notin L])<1/3.$$

Nous allons construire un circuit C de taille $\leqslant m(N)^2$ tel qu'il existe $i\in[0,m(N)-1]$ vérifiant

$$\Pr_{z\in\{0,1\}^N:r=f(z)}(C(r_1,\dots,r_i)=r_{i+1})>1/2+1/(6m(N)).$$

Nous décrivons d'abord un algorithme probabiliste B pour tenter de calculer r_{i+1}, dépendant de x_0 et de $[x_0\in L]$, avant d'en faire un circuit. Soit $z\in\{0,1\}^N$ et $r=f(z)\in\{0,1\}^{m(N)}$: sur l'entrée $r_1,\dots,r_i$, B choisit $r'_{i+1},\dots,r'_{m(N)}$ au hasard et renvoie le bit a_{i+1} défini par :

$$\begin{cases} a_{i+1}=r'_{i+1} & \text{si } M(x_0,r_1\dots r_i\,r'_{i+1}\dots r'_{m(N)})=[x_0\notin L]\\ a_{i+1}=1-r'_{i+1} & \text{sinon.}\end{cases}$$

L'intuition est que, puisque M fait une erreur sur x_0 lorsqu'elle utilise les bits $f(z)$, elle devrait avoir tendance à renvoyer la mauvaise réponse si $r'_{i+1}=r_{i+1}$. Si l'on connaît x_0 et $[x_0\in L]$, le temps de calcul de B est $o(m(N))$. Nous allons montrer qu'il existe i tel que $\Pr_{z,r'}(a_{i+1}=r_{i+1})>1/2+1/(6m(N))$. Soit

$$p_i=\Pr_{z,r':r=f(z)}(M(x_0,r_1\dots r_i\,r'_{i+1}\dots r'_{m(N)})=[x_0\notin L]):$$

alors $p_0<1/3$ et $p_{m(N)}\geqslant 1/2$ par hypothèse. On a :

$$\begin{aligned}\Pr_{z,r'}(a_{i+1}=r_{i+1})=&\\ &\Pr_{z,r':r=f(z)}\left(M(x_0,r_1\dots r_i\,r'_{i+1}\dots r'_{m(N)})=[x_0\notin L]\wedge r'_{i+1}=r_{i+1}\right)+\\ &\qquad\Pr_{z,r':r=f(z)}\left(M(x_0,r_1\dots r_i\,r'_{i+1}\dots r'_{m(N)})=[x_0\in L]\wedge r'_{i+1}\neq r_{i+1}\right)\end{aligned}$$

car $a_{i+1}=r'_{i+1}$ ssi M se trompe. On note $r=r_1\dots r_i$ et $r'=r'_{i+1}\dots r'_{m(N)}$; alors par définition des probabilités conditionnelles (cf. lemme A-D) :

$$\begin{aligned}\Pr_{z,r'}(a_{i+1}=r_{i+1})=\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}=r_{i+1}\right)\Pr(r'_{i+1}=r_{i+1})+&\\ \left(1-\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}\neq r_{i+1}\right)\right)\Pr(r'_{i+1}\neq r_{i+1})&\end{aligned}$$

où les probabilités sont prises sur z et r', avec $r=f(z)$. Mais $\Pr(r'_{i+1}=r_{i+1})=\Pr(r'_{i+1}\neq r_{i+1})=1/2$ et on a

$$\begin{aligned}\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}=r_{i+1}\right)=&\\ \Pr\left(M(x_0,r_1\dots r_{i+1}r'_{i+2}\dots r'_{m(N)})=[x_0\notin L]\right)=p_{i+1},&\end{aligned}$$

donc :

$$\Pr_{z,r'}(a_{i+1}=r_{i+1})=1/2\Big(1+p_{i+1}-\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}\neq r_{i+1}\right)\Big).$$

Ainsi,

$$\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}\neq r_{i+1}\right)=1+p_{i+1}-2\Pr(a_{i+1}=r_{i+1}).$$

On en déduit que

$$\begin{aligned}
p_i &= \Pr\left(M(x_0,r\,r')=[x_0\notin L]\right)\\
&= 1/2\Big(\Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}=r_{i+1}\right)+\\
&\quad \Pr\left(M(x_0,r\,r')=[x_0\notin L]\mid r'_{i+1}\neq r_{i+1}\right)\Big)\\
&= 1/2\big(p_{i+1}+1+p_{i+1}-2\Pr(a_{i+1}=r_{i+1})\big)\\
&= 1/2+p_{i+1}-\Pr(a_{i+1}=r_{i+1})
\end{aligned}$$

donc

$$\Pr(a_{i+1}=r_{i+1})=1/2+p_{i+1}-p_i.$$

Puisque $\sum_{i=0}^{m(N)-1}(p_{i+1}-p_i)=p_{m(N)}-p_0>1/6$, il existe i tel que

$$p_{i+1}-p_i>\frac{1}{6m(N)}$$

et donc tel que $\Pr_{z,r'}(a_{i+1}=r_{i+1})>1/2+1/\big(6m(N)\big)$.
Ainsi, il existe un choix r'_0 de r' tel que $\Pr_z(a_{i+1}=r_{i+1})>1/2+1/(6m(N))$. Maintenant, notre circuit C simule simplement $B(x_0,r'_0)$: sa taille est $o(m(N)^2)$ (puisque le temps de calcul de B est $o(m(N))$, cf. proposition 5-Y) et

$$\Pr_{z\in\{0,1\}^N:r=f(z)}\big(C(r_1,\ldots,r_i)=r_{i+1}\big)>1/2+1/\big(6m(N)\big),$$

une contradiction avec l'imprédictibilité de f. □

12-G Corollaire

- Si, pour une constante $\epsilon>0$, il existe une fonction $f:\{0,1\}^\star\to\{0,1\}^\star$ dans $\mathsf{DTIME}(2^{O(n)})$ qui est $2^{\epsilon n}$-imprédictible, alors $\mathsf{BPP}=\mathsf{P}$.
- Si, pour une constante $\epsilon>0$, il existe une fonction $f:\{0,1\}^\star\to\{0,1\}^\star$ dans $\mathsf{DTIME}(2^{O(n)})$ qui est 2^{n^ϵ}-imprédictible, alors $\mathsf{BPP}\subseteq\cup_k\mathsf{DTIME}(2^{(\log n)^k})$.
- Si pour tout $c>0$, il existe une fonction $f_c:\{0,1\}^\star\to\{0,1\}^\star$ dans $\mathsf{DTIME}(2^{O(n)})$ qui est n^c-imprédictible, alors $\mathsf{BPP}\subseteq\cap_\epsilon\mathsf{DTIME}(2^{n^\epsilon})$.

Démonstration – On a $m(n) = 2^{\epsilon n}$. Pour tout k, considérons $u(n) = (k/\epsilon)\log n$: alors $m(u(n)) = n^k$. Soit $\alpha > 0$ tel que $f \in \mathsf{DTIME}(2^{\alpha n})$. Par le théorème 12-F, $\mathsf{BPTIME}(o(n^k)) \subseteq \mathsf{DTIME}(n^{k/\epsilon} n^{\alpha k/\epsilon}) \subseteq \mathsf{P}$.

– On a $m(n) = 2^{n^\epsilon}$. Pour tout k, considérons $u(n) = (k \log n)^{1/\epsilon}$: alors $m(u(n)) = n^k$. Par le théorème 12-F, $\mathsf{BPTIME}(o(n^k)) \subseteq \mathsf{DTIME}(2^{(\log n)^{O(1)}})$.

– Soit ϵ et k fixés. On définit $u(n) = n^\epsilon$, $c = k/\epsilon$ et $m(n) = n^c$. Alors par le théorème 12-F, $\mathsf{BPTIME}(o(n^k)) \subseteq \mathsf{DTIME}(2^{O(n^\epsilon)})$.

□

12.3 Difficulté en moyenne implique imprédictibilité

Comme nous allons le voir à la définition suivante, une fonction est dite difficile en moyenne si aucun circuit de petite taille ne peut la calculer sur une proportion significativement plus grande que la moitié des entrées. Elle est dite difficile dans le pire cas si aucun circuit de petite taille ne peut la calculer sur toutes ses entrées. Dans ce qui suit on parlera de probabilités prises sur l'ensemble des entrées x : ce n'est rien d'autre qu'un moyen commode de parler d'une proportion d'entrées satisfaisant une propriété.

12-H Définition (difficultés en moyenne et dans le pire cas)

Soit $f : \{0,1\}^n \to \{0,1\}$ et $\rho \in [0,1]$.

– La ρ-difficulté en moyenne de f, notée $\mathsf{D}^\rho_{\mathsf{moy}}(f)$, est le plus grand entier s tel que pour tout circuit C de taille $\leqslant s$, $\Pr_{x\in\{0,1\}^n}(C(x) = f(x)) < \rho$.

– La *difficulté dans le pire cas* de f est $\mathsf{D}_{\mathsf{pire}}(f) = \mathsf{D}^1_{\mathsf{moy}}(f)$.

– La *difficulté en moyenne* de f est $\mathsf{D}_{\mathsf{moy}}(f) = \max\{s \mid \mathsf{D}^{1/2+1/s}_{\mathsf{moy}}(f) \geqslant s\}$, c'est-à-dire le plus grand s tel que pour tout circuit C de taille $\leqslant s$, $\Pr_{x\in\{0,1\}^n}(C(x) = f(x)) < 1/2 + 1/s$.

– Si $f : \{0,1\}^\star \to \{0,1\}$ alors $\mathsf{D}_{\mathsf{moy}}(f)$ et $\mathsf{D}_{\mathsf{pire}}(f)$ sont les fonctions $\mathbb{N} \to \mathbb{N}$ qui sur l'entrée n valent respectivement $\mathsf{D}_{\mathsf{moy}}(f^{=n})$ et $\mathsf{D}_{\mathsf{pire}}(f^{=n})$.

12-I Remarques

– Pour toute fonction $f : \{0,1\}^\star \to \{0,1\}$, $\mathsf{D}_{\mathsf{pire}}(f) \geqslant \mathsf{D}_{\mathsf{moy}}(f)$.

– Si s est la taille du plus petit circuit calculant f, alors $\mathsf{D}_{\mathsf{pire}}(f) = s - 1$.

✎ **12-J** **Exercice**

Montrer que la plupart des fonctions $f : \{0,1\}^\star \to \{0,1\}$ ont une difficulté en moyenne exponentielle (c'est-à-dire $\mathsf{D}_{\mathsf{moy}}(f) = 2^{\Omega(n)}$).

Indication : dénombrer les fonctions et les circuits en utilisant le lemme 5-U et le théorème A-P.

12-K **Remarques**

- $\mathsf{NP} \not\subset \mathsf{P/poly}$ ssi $\mathsf{D}_{\mathsf{pire}}(\mathrm{SAT}) = n^{\omega(1)}$.
- Puisque $\mathsf{BPP} \subset \mathsf{P/poly}$, si $f \in \mathsf{BPP}$ alors $\mathsf{D}_{\mathsf{pire}}(f) = n^{O(1)}$.

Il s'avère qu'on peut transformer une fonction difficile en moyenne en une fonction imprédictible comme le montre le théorème suivant de Nisan et Wigderson [NW88].

12-L **Théorème (Nisan et Wigderson, 1988)**

Soit $s(n)$ une fonction telle qu'il existe $d : \mathbb{N} \to \mathbb{N}$ dans $\mathsf{DTIME}(2^n)$ vérifiant $d(n) \leqslant n/48$ et $2^{2d(n)} < s(\lfloor \sqrt{nd(n)/12} \rfloor)$. Supposons qu'il existe $f : \{0,1\}^\star \to \{0,1\}$ dans $\mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{moy}}(f) \geqslant s(n)$. Alors il existe une fonction $g \in \mathsf{DTIME}(2^{O(n)})$ qui est $\lfloor 2^{d(n)/2} \rfloor$-imprédictible.

Plus précisément, à partir de toute fonction $f' : \{0,1\}^m \to \{0,1\}$ vérifiant $\mathsf{D}_{\mathsf{moy}}(f') \geqslant s(m)$, on construit une fonction $g' : \{0,1\}^{\alpha(m)} \to \{0,1\}^{\lfloor 2^{d(\alpha(m))/2} \rfloor}$, où $\alpha(m)$ vérifie $\alpha(m)d(\alpha(m)) = \Theta(m^2)$, telle que g' est $\lfloor 2^{d(\alpha(m))/2} \rfloor$-imprédictible. Le calcul de g' se fait en temps $2^{O(\alpha(m))}$ en effectuant des appels à la fonction f'.

Idée de la démonstration Pour $n > m$, on va définir $g^{=n}$ à partir de $f^{=m}$. Si l'on évalue f sur un sous-ensemble de taille m des entrées $x_1, \ldots, x_n$, alors un petit circuit ne pourra prédire le résultat sur beaucoup plus de la moitié des entrées puisque f est difficile en moyenne. Si maintenant f est évaluée sur N sous-ensembles disjoints de taille m des entrées $x_1, \ldots, x_n$, alors les N résultats de f sur ces sous-ensembles sont indépendants : on obtient ainsi N bits imprédictibles. Cela implique d'avoir Nm variables pour g, ce qui est trop : on va donc construire des sous-ensembles de $[1,n]$ « presque disjoints », qui peuvent être nombreux sans faire exploser n, et on évaluera f sur chacun de ces sous-ensembles.

Pour la démonstration nous avons besoin de lemmes techniques. Définissons d'abord ce qu'on entend par « sous-ensembles presque disjoints » : une famille de grands sous-ensembles de $[1,n]$ dont les intersections deux à deux sont petites.

12-M Définition (motif combinatoire)

Soit m, d et n des entiers. Une famille $(I_1,\dots,I_N)$ de sous-ensembles de $\{1,\dots,n\}$ est un (m,d)-*motif* si

- $m \geqslant 2d$ et $n \in [12m^2/d, 24m^2/d]$;
- pour tout i, $|I_i| = m$;
- $|I_i \cap I_j| \leqslant d$ pour $i \neq j$.

12-N Remarque Les paramètres m,n,d ci-dessus satisfont $n \geqslant 24m \geqslant 48d$ puisque $m \geqslant 2d$.

Nous nous attachons maintenant à construire de tels ensembles.

12-O Lemme

Soit $d \geqslant 2$, $m \geqslant 2d$ et $n \in [12m^2/d, 24m^2/d]$. Si $I_1,\dots,I_k$ sont des sous-ensembles arbitraires de $\{1,\dots,n\}$ de taille m, où $k < 2^{d/2}$, alors il existe $J \subset \{1,\dots,n\}$ tel que $|J| = m$ et pour tout i, $|J \cap I_i| \leqslant d$.

Démonstration Nous utilisons la méthode probabiliste consistant à montrer l'existence d'un tel ensemble J en le choisissant au hasard. On construit un sous-ensemble aléatoire J de $\{1,\dots,n\}$ comme suit : chaque élément $x \in \{1,\dots,n\}$ est inséré dans l'ensemble avec probabilité $2m/n \leqslant d/(6m)$. On note $X = |J|$: alors l'espérance et la variance de X sont

$$E(X) = 2m \quad \text{et} \quad \operatorname{Var}(X) = n(2m/n)(1-2m/n) < 2m$$

comme somme de n variables indépendantes suivant une loi de Bernoulli (voir le lemme A-K). Donc par l'inégalité de Tchebychev A-L :

$$\begin{aligned}\Pr(X \geqslant m) \geqslant \Pr(|X - E(X)| \leqslant m) &= 1 - \Pr(|X - E(X)| > m) \\ &> 1 - \operatorname{Var}(X)/m^2 > 1 - 2/m \geqslant 1/2\end{aligned}$$

car $m \geqslant 4$.
De plus pour tout i fixé, puisque la probabilité d'insertion d'un élément dans J est $2m/n \leqslant d/(6m)$, on a :

$$\Pr(|J \cap I_i| > d) \leqslant \binom{m}{d}(d/(6m))^d < (m^d/(d!))(d/m)^d/6^d \leqslant 1/2^d$$

car $d! > (d/3)^d$ par la formule A-P de Stirling. Ainsi

$$\Pr(\exists i, |J \cap I_i| > d) < 2^{d/2}/2^d \leqslant 1/2$$

donc $\Pr(\forall i, |J \cap I_i| \leqslant d) > 1/2$. Au final,

$$\Pr\left((\forall i, |J \cap I_i| \leqslant d) \wedge X \geqslant m\right) > 1 - (1/2) - (1/2) = 0,$$

donc il existe J tel que $\forall i, |J \cap I_i| \leqslant d$ et $|J| \geqslant m$. Afin d'obtenir $|J| = m$, il suffit d'enlever des éléments arbitraires de J puisque cela préserve la propriété. □

12-P Lemme

Pour tous $d \geqslant 2$, $m \geqslant 2d$, $n \in [12m^2/d, 24m^2/d]$ et $N \leqslant 2^{d/2}$, un (m,d)-motif $(I_1, \ldots, I_N)$ existe et peut être construit en temps $O(2^{2n})$.

Démonstration Un algorithme glouton va fonctionner. On choisit le premier ensemble I_1 de taille m arbitrairement, par exemple $I_1 = \{1, \ldots, m\}$. Puis à chacune des $N-1$ étapes suivantes, on trouve par recherche exhaustive dans $\{1, \ldots, n\}$ un sous-ensemble I_{i+1} satisfaisant $|I_{i+1}| = m$ et pour tout $j \leqslant i$, $|I_j \cap I_{i+1}| \leqslant d$. Un tel ensemble existe par le lemme 12-O. Le temps nécessaire pour en trouver un est $O(Nm\binom{n}{m})$ puisqu'il faut énumérer l'ensemble des possibilités et comparer à tous les sous-ensembles déjà construits. Or $\binom{n}{m} \leqslant 2^n$: l'algorithme complet tourne donc en un temps

$$O(2^{d/2} N m \binom{n}{m}) = O(2^{2n}).$$

□

Ces motifs nous permettent de définir le générateur de Nisan et Wigderson, dont une illustration est donnée à la figure 12.1.

12-Q Définition (générateur de Nisan et Wigderson)

Soit $f : \{0,1\}^m \to \{0,1\}$ une fonction et $I = (I_1, \ldots, I_N)$ un (m,d)-motif (où $I_1, \ldots, I_N$ sont des sous-ensembles de $\{1, \ldots, n\}$, avec $n \in [12m^2/d, 24m^2/d]$). La fonction $f_I^{\mathsf{NW}} : \{0,1\}^n \to \{0,1\}^N$ est définie par

$$f_I^{\mathsf{NW}}(x) = f(x|_{I_1}) f(x|_{I_2}) \ldots f(x|_{I_N}),$$

où $x|_{I_i}$ est $x_{i_1} x_{i_2} \ldots x_{i_m}$ si $I_i = \{i_1 < i_2 < \cdots < i_m\}$. Cette définition s'étend aux fonctions $f : \{0,1\}^\star \to \{0,1\}$ de manière naturelle si l'on dispose de (m,d)-motifs pour tout $m \in \mathbb{N}$.

Dans la suite, on supposera que le motif utilisé est celui donné par le lemme 12-P, donc on omettra la référence au motif pour alléger les notations : on notera simplement f^{NW} au lieu de f_I^{NW}.

Du lemme 12-P et de la définition précédente, on déduit le résultat suivant.

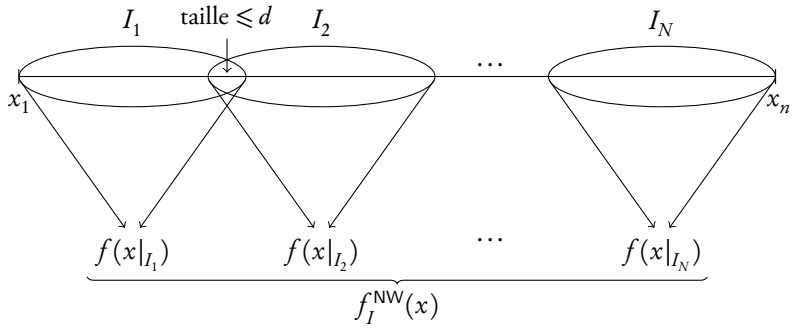


FIGURE 12.1 – Générateur de Nisan et Wigderson (contrairement à ce que suggère le dessin, les I_j sont des sous-ensembles quelconques, pas forcément des intervalles).

12-R Corollaire

Soit $m(n)$ et $d(n)$ des fonctions calculables en temps $O(2^n)$ telles que $m(n) \geqslant 2d(n)$ et $n \in [12m(n)^2/d(n), 24m(n)^2/d(n)]$. Si $f \in \mathsf{DTIME}(2^{\alpha n})$ (pour une constante $\alpha \geqslant 1$) alors la fonction $f^{\mathsf{NW}} : \{0,1\}^\star \to \{0,1\}^\star$ est calculable en temps $O(2^{(\alpha+1)n})$.

Démonstration On commence par calculer le $(m(n), d(n))$-motif en temps $O(2^{2n})$ par le lemme 12-P. Puis on évalue f sur tous les $N \leqslant 2^{d(n)/2}$ sous-ensembles, ce qui prend un temps $O(N2^{\alpha n}) = O(2^{(\alpha+1)n})$, donc le temps de calcul total est $O(2^{(\alpha+1)n})$. □

Nous sommes maintenant prêts pour la démonstration du théorème.

Démonstration du théorème 12-L Soit n fixé, on notera $d = d(n)$ pour alléger les notations. Définissons $m = \lfloor \sqrt{nd/12} \rfloor$ de sorte que $n \in [12m^2/d, 24m^2/d]$. Puisque $n \geqslant 48d$, on a

$$m > \sqrt{nd/12} - 1 \geqslant \sqrt{48d^2/12} - 1 = 2d - 1, \text{ donc } m \geqslant 2d.$$

Soit $N = \lfloor 2^{d/2} \rfloor$ et $(I_1, \ldots, I_N)$ un (m,d)-motif. Soit $g = f^{\mathsf{NW}}$, donc $g : \{0,1\}^n \to \{0,1\}^N$ et $g \in \mathsf{DTIME}(2^{O(n)})$ par le corollaire 12-R. On remarque donc que g sur les entrées de taille n est calculée à partir de $f^{=m}$; et si on appelle α la fonction telle que $\alpha(m) = n$, on a $\alpha(m)d(\alpha(m)) \in [12m^2, 24m^2]$.

Supposons par l'absurde que g n'est pas N-imprédictible, c'est-à-dire qu'il existe un circuit C de taille $\leqslant N^2$ et $i \in [0, N-1]$ tels que

$$\Pr_{z \in \{0,1\}^n : r = g(z)} (C(r_1, \ldots, r_i) = r_{i+1}) \geqslant 1/2 + 1/(6N).$$

Nous allons obtenir une contradiction avec la difficulté de f. Puisque $r_j = f(z|_{I_j})$, on a :

$$\Pr_{z \in \{0,1\}^n} \left(C(f(z|_{I_1}), \ldots, f(z|_{I_i})) = f(z|_{I_{i+1}})\right) \geqslant 1/2 + 1/(6N).$$

Appelons z' les m variables $z|_{I_{i+1}}$, z'' les $(n-m)$ variables $z|_{[1,n]\setminus I_{i+1}}$, et pour $j \neq i+1$, z'_j les $\leqslant d$ variables $z|_{I_j \cap I_{i+1}}$ et z''_j les $\geqslant m-d$ variables $z|_{I_j \setminus I_{i+1}}$. Alors

$$\Pr_{z' \in \{0,1\}^m, z'' \in \{0,1\}^{n-m}} \left(C(f(z'_1, z''_1), f(z'_2, z''_2), \ldots, f(z'_i, z''_i)) = f(z')\right) \geqslant 1/2 + 1/(6N).$$

Il existe donc un mot z'' fixé tel que

$$\Pr_{z' \in \{0,1\}^m} \left(C(f(z'_1, z''_1), f(z'_2, z''_2), \ldots, f(z'_i, z''_i)) = f(z')\right) \geqslant 1/2 + 1/(6N).$$

Maintenant, puisque z'' est fixé, $f(z'_j, z''_j)$ dépend seulement des $\leqslant d$ variables z'_j et a donc un circuit de taille $d2^d$ (le circuit qui calcule simplement sa table de vérité). Combiné avec C, cela donne un circuit D de taille $id2^d + 2^d \leqslant 2^{d/2} d2^d + 2^d$ tel que

$$\Pr_{z' \in \{0,1\}^m} (D(z') = f(z')) \geqslant 1/2 + 1/(6N).$$

C'est une contradiction avec la difficulté de f dès que $2^{d/2} d2^d + 2^d < s(m)$. Puisque $m = \lfloor \sqrt{nd/12} \rfloor$, il suffit que $2^{2d(n)} < s(\lfloor \sqrt{nd/12} \rfloor)$, ce qui est vrai par hypothèse. □

12-S Corollaire

- S'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{moy}}(f) \geqslant 2^{\epsilon n}$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} = \mathsf{P}$.
- S'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{moy}}(f) \geqslant 2^{n^\epsilon}$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} \subseteq \bigcup_k \mathsf{DTIME}(2^{(\log n)^k})$.
- Si pour tout $c > 0$ il existe une fonction $f_c \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{moy}}(f_c) \geqslant n^c$, alors $\mathsf{BPP} \subseteq \bigcap_{\epsilon > 0} \mathsf{DTIME}(2^{n^\epsilon})$.

Démonstration

– Sans perte de généralité, $\epsilon \leqslant 1$. Pour $d(n) = (\epsilon^2/50)n$, la fonction $s(n) = 2^{\epsilon n}$ satisfait les hypothèses du théorème 12-L :

$$s(\sqrt{nd/12}) = s((\epsilon/\sqrt{600})n) = 2^{(\epsilon^2/\sqrt{600})n} > 2^{(\epsilon^2/25)n} = 2^{2d(n)}.$$

Donc il existe une fonction $2^{\epsilon^2 n/100}$-imprédictible $g \in \mathsf{DTIME}(2^{O(n)})$. Par le corollaire 12-G, cela implique $\mathsf{BPP} = \mathsf{P}$.

– Sans perte de généralité $\epsilon < 2$. Pour $d(n) = n^{\epsilon/2}$, la fonction $s(n) = 2^{n^\epsilon}$ satisfait les hypothèses du théorème 12-L :

$$s(\sqrt{nd/12}) = s(12^{-1/2} n^{1/2 + \epsilon/4}) = 2^{12^{-\epsilon/2} n^{\epsilon/2 + \epsilon^2/4}} > 2^{2n^{\epsilon/2}} = 2^{2d(n)}.$$

Il existe donc une fonction $2^{(1/2)n^{\epsilon/2}}$-imprédictible $g \in \mathsf{DTIME}(2^{O(n)})$. Par le corollaire 12-G, cela implique $\mathsf{BPP} \subseteq \bigcup_k \mathsf{DTIME}(2^{(\log n)^k})$.

– Pour $d(n) = (c/4)\log n$, la fonction $s(n) = n^c$ satisfait les hypothèses du théorème 12-L :

$$s(\sqrt{nd/12}) = ((c/48)n\log n)^{c/2} > n^{c/2} = 2^{2d(n)}.$$

Il existe donc une fonction $n^{c/8}$-imprédictible $g \in \mathsf{DTIME}(2^{O(n)})$. Par le corollaire 12-G, cela implique $\mathsf{BPP} \subseteq \cap_\epsilon \mathsf{DTIME}(2^{n^\epsilon})$.

□

12.4 Difficulté dans le pire cas implique difficulté en moyenne

Nous allons maintenant construire une fonction difficile en moyenne à partir d'une fonction difficile dans le pire cas. Pour cela, nous passerons par des codes correcteurs d'erreurs. Informellement, un code correcteur d'erreurs est une fonction E qui code un mot x en un mot $y = E(x)$ de sorte qu'on peut retrouver x à partir d'une version erronée y' de son image qui diffère de y sur une proportion α des bits. Nous verrons une fonction $f : \{0,1\}^n \to \{0,1\}$ comme un mot de $\{0,1\}^{2^n}$ correspondant à sa table de vérité.

L'idée pour transformer une fonction f difficile dans le pire cas en une fonction g difficile en moyenne est la suivante. Supposons qu'on dispose d'un bon code correcteur d'erreurs E : on définit simplement $g = E(f)$. Ainsi, si on peut calculer g correctement sur une fraction $(1-\alpha)$ de ses entrées, alors on peut calculer (retrouver) f exactement grâce au code correcteur. Donc si f est difficile dans le pire cas, aucun petit circuit ne la calcule correctement sur toutes les entrées et on en déduit qu'aucun petit circuit ne calcule g sur une fraction $(1-\alpha)$ de ses entrées, et g est donc difficile en moyenne si α est proche de $1/2$.

La discussion qui précède montre qu'il nous faut un code correcteur d'erreurs performant, et c'est le sujet de la partie (technique) qui suit.

12.4.1 Codes correcteurs d'erreurs

Nous allons voir deux codes correcteurs, celui de Hadamard et celui de Reed-Muller. Nous montrerons qu'on peut décoder efficacement ces codes, mais cela ne suffit pas. Pour notre application, nous aurons besoin d'une combinaison des deux, appelée « concaténation ». Nous devrons réussir à décoder cette concaténation en temps polylogarithmique, c'est-à-dire sans lire toute l'image, tout en étant capable de retrouver chaque bit du mot original lorsque près de la moitié des bits du codage sont corrompus. Pour cela, nos algorithmes renverront non pas directement le mot x qui a été codé, mais une petite liste de candidats parmi lesquels figure x : nous pouvons ensuite retrouver x grâce à un conseil de taille logarithmique donnant le numéro de x dans cette liste. Le but de cette partie est de donner les algorithmes de décodage locaux avec conseil pour ces deux codes, et de montrer comment les combiner pour obtenir le décodage de la concaténation des codes.

Dans cette partie Σ désigne un ensemble fini (alphabet). On aura bientôt besoin que Σ soit un corps fini, mais pour l'instant cette hypothèse est inutile.

12-T **Définition (distance de Hamming fractionnaire)**

Pour $x, y \in \Sigma^m$, la *distance de Hamming fractionnaire* entre x et y est

$$\Delta(x,y) = \frac{|\{i \in [1,m] \mid x_i \neq y_i\}|}{m}.$$

12-U **Définition (code correcteur d'erreurs)**

Pour $\delta \in]0,1[$, un *code correcteur d'erreurs* de distance δ est une fonction $E : \Sigma^n \to \Sigma^m$ telle que

$$x \neq y \implies \Delta(E(x),E(y)) \geqslant \delta.$$

Cette définition se généralise aux fonctions $E : \Sigma^\star \to \Sigma^\star$ vérifiant

$$|x| = |y| \implies |E(x)| = |E(y)|.$$

L'entier m est appelé *taille* du code E.

12-V **Remarques**

- La définition implique que tout code correcteur est injectif.
- Sur $\{0,1\}$, des codes de taille polynomiale (c'est-à-dire que m est polynomial en n) existent pour toute distance $\delta < 1/2$ mais pas pour $\delta = 1/2$, distance pour laquelle seuls des codes de taille exponentielle existent. Enfin, aucun code, quelle que soit sa taille, n'existe pour $\delta > 1/2$ (attention, cela n'est vrai que si $|\Sigma| = 2$).

12-W **Remarque** Si $E : \Sigma^n \to \Sigma^m$ est un code de distance δ et $x \in \Sigma^n$, on peut retrouver x à partir de tout mot $z \in \Sigma^m$ vérifiant $\Delta(z,E(x)) < \delta/2$: $E(x)$ est en effet le mot $y \in \mathrm{Im}(E)$ le plus proche de z et à partir de $E(x)$ il suffit alors d'inverser E pour retrouver x.

On rappelle que pour $x, y \in \{0,1\}^n$, le produit scalaire de x, y est $x \cdot y = \sum_{i=1}^n x_i y_i \mod 2$.

12-X Définition (code de Hadamard)

Le *code de Hadamard* est la fonction $\mathsf{H} : \{0,1\}^* \to \{0,1\}^*$ définie sur $\{0,1\}^n$ par $\mathsf{H}(x) = z_0 z_1 \dots z_{2^n-1}$, où $z_j = x \cdot j$ (ici, j est interprété comme sa représentation binaire sur $\{0,1\}^n$). On a donc $|\mathsf{H}(x)| = 2^{|x|}$.

12-Y Lemme

La fonction H est un code de distance $1/2$.

Démonstration Soit $x, y \in \{0,1\}^n$ deux mots différant au i-ème bit. Pour tous $u \in \{0,1\}^{i-1}$ et $v \in \{0,1\}^{n-i}$, soit $x \cdot u0v \neq y \cdot u0v$ soit $x \cdot u1v \neq y \cdot u1v$, donc $x \cdot j \neq y \cdot j$ pour la moitié des entiers $j \in [0, 2^n - 1]$. □

Pour la définition des codes de Reed-Muller [Mul54], nous avons besoin d'un lemme. Pour un polynôme, on appellera « degrés individuels » les degrés de chaque variable.

À partir de maintenant, Σ **est un corps fini.**

12-Z Lemme

Soit k et d deux entiers tels que k divise d et $d/k < |\Sigma|$. Pour tout ensemble $S \subseteq \Sigma$ de taille $1 + d/k$, l'application φ définie par $\varphi(p) = f_p$, où $f_p : S^k \to \Sigma$ est la fonction définie par $f_p(x) = p(x)$, est une bijection entre l'ensemble des polynômes à k variables $p : \Sigma^k \to \Sigma$ de degrés individuels $\leqslant d/k$ (et donc de degré total $\leqslant d$) et l'ensemble des fonctions $f : S^k \to \Sigma$.

Démonstration Les polynômes à k variables sur Σ de degrés individuels $\leqslant d/k$ ont $(1 + d/k)^k$ monômes, chacun avec un coefficient dans Σ : il y a donc $|\Sigma|^{(1+d/k)^k}$ tels polynômes. C'est aussi le nombre de fonctions de S^k dans Σ puisque $|S| = (1 + d/k)$. Il suffit donc de montrer que la fonction φ est injective. Cela se fait en montrant que pour tout polynôme non nul p à k variables de degrés individuels $\leqslant m$ sur un corps K de taille $> m$ et pour tout sous-ensemble $T \subseteq K$ de cardinal $1 + m$, il existe $a \in T^k$ tel que $p(a) \neq 0$. Cette simple adaptation du lemme 11-AH est laissée en exercice. □

Soit $k \leqslant d$ deux entiers tels que $d < |\Sigma|$ et k divise d. Par le lemme 12-Z, un polynôme à k variables

$$p(x_1, \dots, x_k) = \sum_{d_1 \leqslant d/k, \dots, d_k \leqslant d/k} c_{d_1,\dots,d_k} x_1^{d_1} \dots x_k^{d_k}$$

de degrés individuels $\leqslant d/k$ (et donc de degré total $\leqslant d$) sur Σ est caractérisé par la liste de ses valeurs sur toutes les entrées de l'ensemble $\{\sigma_0, \dots, \sigma_{d/k}\}^k$, où les $\sigma_i \in \Sigma$ sont des éléments distincts et arbitraires de Σ. On prendra pour $\sigma_0, \dots, \sigma_{d/k}$ les $(1 + d/k)$ premiers

points dans l'ordre lexicographique dans la représentation binaire de Σ. Cette représentation d'un polynôme p sera l'entrée de notre code correcteur.

12-AA Définition (code de Reed-Muller)

Un polynôme $p \in \Sigma[x_1,\dots,x_k]$ de degrés individuels $\leqslant d/k$ est codé par la liste de ses valeurs sur les $(1+d/k)$ premiers points de Σ.

La fonction $\mathsf{RM} : \Sigma^{(1+d/k)^k} \to \Sigma^{|\Sigma|^k}$ est définie sur un tel codage d'un polynôme p, par la liste des valeurs que prend p sur Σ^k, c'est-à-dire

$$\mathsf{RM}(p) = (p(a_1,\dots,a_k))_{a_1,\dots,a_k\in\Sigma}.$$

12-AB Remarques

– Ce n'est pas la définition usuelle des codes de Reed-Muller, car ceux-ci prennent habituellement comme entrée la liste des coefficients de p. Notre version nous évite de faire de l'interpolation qui serait trop coûteuse pour le décodage, puisqu'il suffit ici de calculer $p(x)$ pour tout $x \in \{\sigma_0,\dots,\sigma_{d/k}\}$.

– Les entiers k et d dans le code de Reed-Muller seront fixés plus tard pour notre application.

Le lemme suivant est classique et connu sous le nom de Schwartz-Zippel car il apparaît explicitement dans les articles [Zip79 ; Sch80] (on a déjà vu des résultats similaires aux lemmes 10-M et 11-AH).

12-AC Lemme (Zippel 1979, Schwartz 1980)

Si r est un polynôme non nul à k variables de degré d sur le corps fini Σ, alors $\Pr_{a\in\Sigma^k}(r(a)\neq 0) \geqslant 1 - d/|\Sigma|$.

Démonstration Par récurrence sur k. Pour $k=1$, r a au plus d racines et le résultat suit (il s'agit du lemme 10-M). Pour $k>1$, on exprime r selon la variable x_1 :

$$r(x_1,\dots,x_k) = \sum_{i=0}^{d} r_i(x_2,\dots,x_k)x_1^i$$

où les r_i sont des polynômes à $(k-1)$ variables de degré $\leqslant d-i$. Puisque r est non nul, il existe i tel que r_i est non nul : soit i_0 le plus grand i tel que $r_i \neq 0$. Par hypothèse de récurrence,

$$\Pr_{a_2,\dots,a_k}(r_{i_0}(a_2,\dots,a_k)\neq 0) \geqslant 1-(d-i_0)/|\Sigma|,$$

et si $r_{i_0}(a_2,\dots,a_k) \neq 0$ alors $r(x_1,a_2,\dots,a_k)$ est un polynôme univarié non nul de degré i_0. Donc

$$\Pr_{a_1,\dots,a_k}(r(a_1,\dots,a_k) \neq 0) \geqslant (1-(d-i_0)/|\Sigma|)(1-i_0/|\Sigma|) \geqslant 1-d/|\Sigma|.$$

□

12-AD Lemme

La fonction RM est un code de distance $1-d/|\Sigma|$.

Démonstration Soit $p \neq q$ deux polynômes à k variables de degré $\leqslant d$ sur Σ : alors $p-q$ est un polynôme non nul à k variables de degré $\leqslant d$. Le lemme 12-AC implique que $p(x) \neq q(x)$ pour une fraction $\geqslant 1-d/|\Sigma|$ des points $x \in \Sigma^k$, ce qui conclut. □

On suppose dorénavant que $|\Sigma|$ **est une puissance de deux**. On a donc $\Sigma = \mathbb{F}_{2^q}$ (le corps fini à 2^q éléments) pour un certain entier q. La définition suivante introduit la concaténation $F \circ E$ de deux codes $E : \Sigma^\star \to \Sigma^\star$ et $F : \Sigma^\star \to \{0,1\}^\star$: il s'agit du code où l'on applique F à chaque symbole de $E(x)$ (illustration à la figure 12.2).

12-AE Définition (concaténation de codes)

- Un code $E : \Sigma^n \to \Sigma^m$ peut être vu comme une fonction $E : \{0,1\}^{n\log|\Sigma|} \to \Sigma^m$ en encodant chaque symbole de Σ par $\log|\Sigma|$ bits. De même, un code $F : \{0,1\}^{\log|\Sigma|} \to \{0,1\}^t$ peut être vu comme une fonction $F : \Sigma \to \{0,1\}^t$.
- Si $E : \Sigma^n \to \Sigma^m$ et $F : \{0,1\}^{\log|\Sigma|} \to \{0,1\}^t$ sont deux codes, leur *concaténation* est le code $F \circ E : \{0,1\}^{n\log|\Sigma|} \to \{0,1\}^{mt}$ défini par $(F \circ E)(x) = F(y_1)\dots F(y_m)$, où $y = E(x) \in \Sigma^m$.

12-AF Lemme

Si $E : \Sigma^n \to \Sigma^m$ a une distance δ_E et $F : \{0,1\}^{\log|\Sigma|} \to \{0,1\}^t$ a une distance δ_F, alors $F \circ E$ a une distance $\delta_E\delta_F$.

Démonstration Soit $x, y \in \Sigma^n$ deux entrées distinctes. Alors $E(x)$ et $E(y)$ diffèrent sur au moins $\delta_E m$ positions. Si i est une telle position (c'est-à-dire $E(x)_i \neq E(y)_i$), alors $F(E(x)_i)$ et $F(E(y)_i)$ diffèrent sur au moins $\delta_F t$ positions. Au total, $(F \circ E)(x)$ et $(F \circ E)(y)$ diffèrent donc sur au moins $\delta_E m \delta_F t$ positions. □

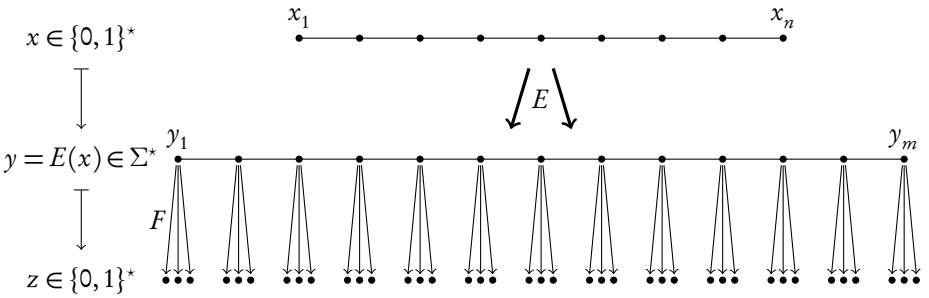


FIGURE 12.2 – Concaténation de deux codes $E : \{0,1\}^\star \to \Sigma^\star$ et $F : \Sigma \to \{0,1\}^\star$.

12-AG Lemme

La concaténationn $\mathsf{H} \circ \mathsf{RM} : \{0,1\}^{(1+d/k)^k \log|\Sigma|} \to \{0,1\}^{|\Sigma|^{k+1}}$ du code de Reed-Muller $\mathsf{RM} : \Sigma^{(1+d/k)^k} \to \Sigma^{|\Sigma|^k}$ et de celui de Hadamard $\mathsf{H} : \{0,1\}^{\log|\Sigma|} \to \{0,1\}^{|\Sigma|}$ est une fonction calculable en temps polynomial en $|\Sigma|^k$.

Démonstration Soit p un polynôme dont on connaît la liste de ses valeurs sur l'ensemble $\{\sigma_0, \ldots, \sigma_{d/k}\}^k$. On peut calculer les coefficients de p en résolvant le système linéaire correspondant, dans lequel les coefficients de p sont les inconnues. Ce système est de taille $(1+d/k)^k \leqslant |\Sigma|^k$ donc la résolution se fait en temps polynomial en $|\Sigma|^k$ par élimination de Gauss.

Une fois que l'on dispose des coefficients de p, il est alors facile en temps polynomial en $|\Sigma|^k$ d'évaluer p sur chaque point de Σ^k puis d'appliquer la fonction H sur chaque valeur obtenue. □

Afin de construire une fonction difficile en moyenne à partir d'une fonction difficile dans le pire cas, supposons que nous voulions appliquer l'idée décrite en début de partie. Puisqu'aucun code sur $\{0,1\}$ n'est de distance $> 1/2$, la remarque 12-W semble indiquer qu'on ne pourrait pas décoder correctement un mot ayant une proportion $\geqslant 1/4$ d'erreurs, ce qui n'est pas suffisant. Ce problème est contourné grâce au « décodage par liste » : plutôt que de décoder de manière unique un mot, on donne une petite liste de candidats possibles, ce qui permet d'augmenter la proportion admissible d'erreurs.

Nous avons besoin de quelques lemmes avant d'expliquer la procédure de décodage par liste du code de Reed-Muller. Le premier, dû à Sudan [Sud96], montre que la liste de candidats peut être calculée efficacement dans le cas univarié ($k = 1$).

12-AH Lemme (Sudan, 1996)

Il y a un algorithme fonctionnant en temps polynomial qui, sur l'entrée un entier $d < m/4$ et m couples $(a_1, b_1), \ldots, (a_m, b_m)$ d'éléments de Σ, renvoie la liste de tous les polynômes $g : \Sigma \to \Sigma$ de degré d tels que le nombre de positions i auxquelles $g(a_i) = b_i$ est $> 2\sqrt{d\,m}$.

Cette liste contient au plus $\sqrt{m/d}$ polynômes.

Démonstration Supposons que $Q(x,y)$ est un polynôme non nul à deux variables sur Σ tel que $Q(a_i,b_i)=0$ pour tout i, $\deg_x(Q)\leqslant\sqrt{md}$ et $\deg_y(Q)\leqslant\sqrt{m/d}$. Si g est dans la liste, alors :

- $Q(x,g(x))=0$ car le polynôme à une variable $Q(x,g(x))$ est de degré majoré par
$$\sqrt{md}+d\sqrt{m/d}=2\sqrt{md}$$
et s'annule sur $>2\sqrt{md}$ points ;
- $g(x)-y$ est un facteur de $Q(x,y)$: en effet, soit $Q'(y)=Q(x,y)\in(\Sigma[x])[y]$, alors $g(x)$ est une racine de $Q'(y)$, donc $y-g(x)$ divise $Q'(y)$, c'est-à-dire que
$$\text{il existe } R'\in(\Sigma[x])[y] \text{ tel que } Q'(y)=(y-g(x))R'(y) :$$
en d'autres termes, il existe $R\in\Sigma[x,y]$ tel que $Q(x,y)=(y-g(x))R(x,y)$. Le nombre de tels facteurs est au plus $\deg_y(Q)\leqslant\sqrt{m/d}$.

Il est donc suffisant de trouver un tel polynôme $Q(x,y)$ et de le factoriser en utilisant, par exemple, l'algorithme de Berlekamp [Ber67] qui permet de factoriser en temps polynomial un polynôme sur un corps fini. Afin de trouver Q, on écrit les m équations linéaires $(Q(a_i,b_i)=0)_{i\in[1,m]}$ que ses coefficients doivent vérifier. Il y a $(1+\sqrt{md})(1+\sqrt{m/d})>m$ coefficients, donc le système a une solution non nulle que l'on trouve en temps polynomial par élimination de Gauss. □

Les deux lemmes qui suivent montrent qu'à partir du codage d'un polynôme comportant des erreurs, on peut efficacement retrouver le polynôme initial. Le premier résultat, de Gemmell et Sudan [GS92] (améliorant un algorithme original de Berlekamp et Welch) concerne les polynômes univariés tandis que le second traite des polynômes à plusieurs variables. On remarquera que la borne polylogarithmique sur le temps d'exécution porte sur la taille de l'entrée qui est exponentielle en n, et il est donc nécessaire d'obtenir une si bonne borne pour obtenir un algorithme polynomial au final.

12-AI Lemme (Berlekamp et Welch 1986, Gemmell et Sudan 1992)

Il existe un algorithme fonctionnant en temps polynomial qui, sur l'entrée un entier $d<|\Sigma|$ et une liste $(a_1,b_1),\ldots,(a_m,b_m)$ de $m>d$ couples d'éléments de Σ tels qu'il existe un polynôme $p:\Sigma\to\Sigma$ de degré d satisfaisant $p(a_i)=b_i$ pour au moins $t>m/2+d/2$ nombres $i\in[1,m]$, renvoie la liste des coefficients de p.

On remarquera que p est unique car il est de degré $\leqslant d$ et spécifié sur $t>d$ points.

Démonstration L'algorithme fonctionne comme suit.

– Trouver un polynôme $A(x)$ de degré $m-t+d$ et un polynôme $B(x)$ non nul de degré $m-t$ tels que $A(a_i)=b_iB(a_i)$ pour tout $i\in[1,m]$. De tels polynômes existent car pour B on peut prendre un polynôme non nul de degré $\leqslant m-t$ qui s'annule sur les $\leqslant m-t$ points a_i pour lesquels $p(a_i)\neq b_i$, et ainsi $A=pB$ est de degré $\leqslant m-t+d$.
Pour trouver A et B, il suffit de résoudre un système linéaire à $2m-2t+d+2$ inconnues (les coefficients de A et B) et m équations (une pour chaque i) : cela prend un temps polynomial grâce à l'élimination de Gauss.

– Renvoyer Q, le quotient dans la division euclidienne de A par B. Puisque $A(x)-p(x)B(x)$ est de degré $\leqslant m-t+d<t$ et est nul sur $\geqslant t$ points, on a $A(x)=p(x)B(x)$, donc $Q(x)=p(x)$.

□

On rappelle qu'un algorithme A a *accès direct* à un mot y, noté A^y, s'il dispose d'un oracle qui, sur l'entrée i en binaire, renvoie le i-ème bit de y (cf. définition 10-AI). Nous pouvons maintenant décoder Reed-Muller sans lire toute l'entrée (d'où le nom « local »). Ce résultat est dû à Gemmell, Lipton, Rubinfeld, Sudan et Wigderson [Gem+91].

12-AJ Lemme (décodeur local pour Reed-Muller)

Il existe un algorithme probabiliste D fonctionnant en temps $(k|\Sigma|)^{O(1)}$ tel que, pour tout $d<(|\Sigma|-1)/3$, si $f:\Sigma^k\to\Sigma$ coïncide avec un polynôme P de degré d sur une fraction $17/18$ des entrées, alors pour tout $x\in\Sigma^k$, $\Pr(D^f(x,d)=P(x))\geqslant 5/6$.

Démonstration L'algorithme D fonctionne comme suit sur l'entrée (x,d).

– Choisir $z\in\Sigma^k$ au hasard et considérer la « ligne » $L_z=\{x+tz\mid t\in\Sigma\}$. Si $X\subset\Sigma^k$ est l'ensemble des entrées sur lesquelles f et P diffèrent, alors on définit $N=|L_z\setminus\{x\}\cap X|$: puisque pour tout $t\neq 0$, $x+tz$ est uniformément distribué, on a

$$E(N)=\sum_{t\in\Sigma\setminus\{0\}}\Pr_z(x+tz\in X)\leqslant(|\Sigma|-1)/18$$

donc, par l'inégalité de Markov (proposition A-I),

$$\Pr_z(N>\frac{|\Sigma|-1}{3})<\frac{(|\Sigma|-1)/18}{(|\Sigma|-1)/3}=1/6.$$

Ainsi, avec probabilité $>5/6$, le nombre de points de $L_z\setminus\{x\}$ sur lesquels f et P coïncident est au moins

$$2(|\Sigma|-1)/3>(|\Sigma|-1)/2+d/2.$$

– Demander la valeur de f sur tous les $|\Sigma|-1$ points de $L_z \setminus \{x\}$ afin d'obtenir l'ensemble

$$\{(t, f(x+tz)) \mid t \in \Sigma \setminus \{0\}\}.$$

– Le polynôme $Q(t) = P(x+tz)$ est univarié de degré $\leqslant d$. Exécuter la procédure de Berlekamp-Welch (lemme 12-AI) sur les points

$$\{(t, f(x+tz)) \mid t \in \Sigma \setminus \{0\}\}$$

afin de retrouver un polynôme Q' (supposé égal à Q).

– Renvoyer $Q'(0)$.

Avec probabilité $> 5/6$, $L_z \setminus \{x\}$ contient plus de $(|\Sigma|-1)/2 + d/2$ points sur lesquels f et P coïncident, donc la procédure de Berlekamp-Welch renvoie Q et l'algorithme renvoie $Q(0) = P(x)$. □

Les deux lemmes techniques qui suivent concernent des courbes aléatoires de degré 3 : quand nous parlerons de polynômes $q : \Sigma \to \Sigma^k$ (courbes paramétrées par $t \in \Sigma$), nous utiliserons le produit $tu = (tu_1, \ldots, tu_k)$ si $t \in \Sigma$ et $u = (u_1, \ldots, u_k) \in \Sigma^k$.

12-AK Lemme

Soit $S \subseteq \Sigma^k$ un ensemble fixé et $x \in \Sigma^k$ un point fixé. Soit $q(t) = x + tu + t^2 v + t^3 w$ un polynôme aléatoire de degré 3 où $t \in \Sigma$, et $u, v, w \in \Sigma^k$ sont choisis au hasard, définissant une courbe $Q = q(\Sigma)$. Alors pour tout $t \in \Sigma \setminus \{0\}$, $\Pr(q(t) \in S) = |S|/|\Sigma|^k$. En outre, pour tous $t \neq t' \in \Sigma$, les événements $q(t) \in S$ et $q(t') \in S$ sont indépendants.

Démonstration Supposons dans un premier temps que $t \neq 0$ et $t' \neq 0$. Nous montrons d'abord que pour $t \neq t'$, si $y, y' \in \Sigma^k$ sont fixés, alors les événements $q(t) = y$ et $q(t') = y'$ sont indépendants. On a

$$q(t) = y \iff u = (y - x - t^2 v - t^3 w)/t,$$

donc le choix de v et w est arbitraire mais alors u est fixé, donc $\Pr(q(t) = y) = 1/|\Sigma|^k$ et de même pour $q(t') = y'$. De plus,

$$(q(t) = y \wedge q(t') = y') \iff$$
$$(tu + t^2 v + t^3 w = y - x \quad \wedge \quad t'u + t'^2 v + t'^3 w = y' - x) \iff$$
$$u = (y - x - t^2 v + t^3 w)/t \quad \wedge \quad (t't^2 - tt'^2)v = t'(y-x) - t(y'-x) - (t't^3 - tt'^3)w :$$

si $t \neq t'$, on peut choisir arbitrairement w mais alors u et v sont fixés, donc

$$\Pr(q(t) = y \wedge q(t') = y') = 1/|\Sigma|^{2k} = \Pr(q(t) = y)\Pr(q(t') = y').$$

Par ailleurs, $\Pr(q(t) \in S) = \sum_{y \in S} \Pr(q(t) = y)$ car ces événements sont disjoints, donc $\Pr(q(t) \in S) = |S|/|\Sigma|^k$, et de même pour $\Pr(q(t') \in S)$. Mais pour $t \neq t'$,

$$\Pr(q(t) \in S \wedge q(t') \in S) = \sum_{y,y' \in S} \Pr(q(t) = y \wedge q(t') = y') = |S|^2/|\Sigma|^{2k},$$

ce qui montre l'indépendance.

Maintenant, si t ou t' est nul, disons $t = 0$, alors soit $x \in S$ et $\Pr(q(t) \in S) = 1$, soit $x \notin S$ et $\Pr(q(t) \in S) = 0$: dans tous les cas, les événements $q(t) \in S$ et $q(t') \in S$ sont indépendants. □

12-AL Lemme

Soit $S \subseteq \Sigma^k$ un ensemble fixé tel que $|S| \geqslant \epsilon|\Sigma|^k$, où $\epsilon < 1/2$, et soit $x \in \Sigma^k$ fixé. Soit $q(t) = x + tu + t^2v + t^3w$ un polynôme aléatoire de degré 3 où $t \in \Sigma$ et où $u, v, w \in \Sigma^k$ sont choisis au hasard, définissant une courbe $Q = q(\Sigma)$.

Alors $\Pr(|Q \cap S| \geqslant 2\epsilon|\Sigma|/15) \geqslant 1 - 25/(\epsilon^2|\Sigma|)$.

Démonstration Puisque q est de degré 3, chaque $y \in \Sigma^k$ n'a pas plus de 3 antécédents. Donc $|Q \cap S| \geqslant (\sum_{t \in \Sigma \setminus \{0\}} x_t)/3$ où $x_t = 1$ si $q(t) \in S$ et $x_t = 0$ sinon. Par le lemme 12-AK, c'est une somme de variables aléatoires deux à deux indépendantes, où $\Pr(x_t = 1) \geqslant \epsilon$. On a

$$E(|Q \cap S|) \geqslant \epsilon(|\Sigma| - 1)/3$$

donc

$$\Pr(|Q \cap S| \leqslant 2\epsilon|\Sigma|/15) \leqslant \Pr\Big(\big||Q \cap S| - E(|Q \cap S|)\big| \geqslant \epsilon(|\Sigma| - 1)/3 - 2\epsilon|\Sigma|/15\Big).$$

Mais puisque $|\Sigma| \geqslant 2$, on a

$$(|\Sigma| - 1)/3 - 2|\Sigma|/15 \geqslant |\Sigma|/30.$$

Par ailleurs,

$$\mathrm{Var}(|Q \cap S|) \leqslant (|\Sigma| - 1)/36$$

(cf. propriétés de la variance au lemme A-K) donc par l'inégalité de Tchebychev (théorème A-L) :

$$\Pr(|Q \cap S| \leqslant 2\epsilon|\Sigma|/15) \leqslant \Pr\Big(\big||Q \cap S| - E(|Q \cap S|)\big| \geqslant \epsilon|\Sigma|/30\Big) \leqslant (|\Sigma| - 1)/(36(\epsilon|\Sigma|/30)^2) \leqslant 25/(\epsilon^2|\Sigma|).$$

□

Pour notre application, nous avons besoin d'algorithmes de décodage extrêmement efficaces permettant de retrouver le mot codé lorsque seule une infime fraction du codage est correcte. Nous nous aiderons pour cela d'un conseil. Le conseil dont il sera question dans nos décodeurs peut être vu comme le numéro du mot x à retrouver dans une liste de candidats fournie par l'algorithme, c'est pourquoi ce genre de décodeur est aussi appelé « à liste » plutôt que « à conseil ». On rappelle que Δ désigne la distance de Hamming fractionnaire (définition 12-T).

12-AM Définition (décodeur local à conseil)

Soit $E : \Sigma^n \to \Gamma^m$ un code et soit $0 < \eta < 1$. Un algorithme probabiliste D (avec bit aléatoires r) est dit *décodeur local à conseil pour E corrigeant η erreurs en temps t avec conseil de taille s* s'il fonctionne en temps t et, pour tous $x \in \Sigma^n$ et $z \in \Gamma^m$ vérifiant $\Delta(E(x), z) \leqslant \eta$, il existe $a(r) \in \{0,1\}^s$ (un conseil dépendant de x, z et r) tel que pour tout $i \in [1, n]$,

$$\Pr_r(D^z(i, a(r), r) = x_i) \geqslant 2/3,$$

c'est-à-dire que D avec accès direct à z décode chaque bit x_i avec probabilité $2/3$, grâce à un conseil indépendant de i.

12-AN Remarques

- Dans la suite, nous considérerons des décodeurs locaux à conseil fonctionnant en temps polylogarithmique en m avec conseil de taille logarithmique en n.
- Si $\mathscr{F}$ est l'ensemble des fonctions $f : \Sigma^k \to \Sigma$, alors on peut considérer que le code de Reed-Muller prend ses valeurs dans $\mathscr{F}$. Dans la proposition suivante, nous voulons un décodeur local à conseil pour le code de Reed-Muller : le conseil a dépend de f mais doit être le même pour tous les mots $x \in \Sigma^k$.

Le décodage suivant de Reed-Muller est dû à Sudan [Sud96] mais notre présentation est plus proche de Sudan, Trevisan et Vadhan [STV99].

12-AO Proposition (décodeur local à conseil pour Reed-Muller)

Il existe une constante $\alpha > 0$ telle que pour tout entier d suffisamment grand, si $d^2 \leqslant |\Sigma|$ alors le code de Reed-Muller a un décodeur local à conseil corrigeant $1 - 30\sqrt{d/|\Sigma|}$ erreurs en temps $(k|\Sigma|)^\alpha$ avec conseil de taille $(k+1)\log|\Sigma|$.

Plus précisément, il existe un algorithme probabiliste D fonctionnant en temps $(k|\Sigma|)^\alpha$ tel que, si $f : \Sigma^k \to \Sigma$ coïncide avec un polynôme p de degré d sur une fraction $30\sqrt{d/|\Sigma|}$ de ses entrées, alors il existe $(a, b) \in \Sigma^k \times \Sigma$ tel que pour tout $x \in \Sigma^k$, $\Pr(D^f(x, a, b) = p(x)) \geqslant 2/3$.

Démonstration Soit $\epsilon = (2/\sqrt{d})^{1/2}$, et on prend d suffisamment grand pour que $\epsilon <$

1/18. Nous décrivons d'abord un algorithme probabiliste avec conseil $(a,b)\in\Sigma^k\times\Sigma$, travaillant sur $\geqslant 1-\epsilon$ des entrées $x\in\Sigma^k$, qui a une probabilité de succès $\geqslant 1-\epsilon$.

- Choisir au hasard $r\in\Sigma\setminus\{0\}$ et $v,w\in\Sigma^k$.
- On définit $u=(a-x-vr^2-wr^3)/r$ et $q(t)=x+tu+t^2v+t^3w$ (pour $t\in\Sigma$), de sorte que la courbe de degré 3, $Q=q(\Sigma)$, vérifie $q(0)=x$ et $q(r)=a$.
- Demander la valeur de f sur tous les points de Q afin d'obtenir les couples $(t,f(q(t)))$ pour tout $t\in\Sigma$.
- Exécuter l'algorithme de Sudan (lemme 12-AH) pour obtenir une liste $g_1,\dots,g_m$ de tous les polynômes de degré $3d$ tels que $g_i(t)=f(q(t))$ pour au moins $4\sqrt{d|\Sigma|}$ points $t\in\Sigma$.
  (On remarquera que l'algorithme de Sudan renvoie tous les polynômes qui coïncident avec plus de $2\sqrt{3d|\Sigma|}<4\sqrt{d|\Sigma|}$ points, mais on peut ignorer ceux qui coïncident sur $<4\sqrt{d|\Sigma|}$ points)
- S'il existe un unique i tel que $g_i(r)=b$ alors renvoyer $g_i(0)$. Sinon s'arrêter (sans rien renvoyer).

Afin de montrer qu'un couple correct (a,b) existe, nous allons utiliser un argument probabiliste. Soit $x\in\Sigma^k$ fixé, on choisit $a\in\Sigma^k$ au hasard et on pose $b=p(a)$. Les choix aléatoires sont donc a,r,v et w. Alors la première étape de l'algorithme revient à choisir une courbe aléatoire Q telle que $q(0)=x$, et la dernière étape à choisir un point aléatoire $r\in\Sigma$ et à vérifier que $g_i(r)=p(q(r))$. Nous prétendons que la probabilité que $g_i=p\circ q$ est élevée. En effet, si $S=\{y\in\Sigma^k\mid f(y)=p(y)\}$, alors par hypothèse $\Pr_y(y\in S)\geqslant 30\sqrt{d/|\Sigma|}$, donc par le lemme 12-AL,

$$\Pr_{a,r,v,w}(|Q\cap S|\geqslant 4\sqrt{d|\Sigma|})\geqslant 1-1/(36d).$$

Ainsi, $p\circ q$ est dans la liste $g_1,\dots,g_m$ avec probabilité $\geqslant 1-1/(36d)$. De plus, la probabilité qu'aucun autre polynôme g_i satisfasse $g_i(r)=p(q(r))$ est élevée : g_i et $p\circ q$, en tant que polynômes distincts de degré $\leqslant 3d$, coïncident seulement sur au plus $3d$ points, donc $\Pr_{a,r,v,w}(g_i(r)=p(q(r)))\leqslant 3d/|\Sigma|$. Au final, puisqu'il y a au plus $\sqrt{|\Sigma|/(3d)}$ polynômes g_i dans la liste, on a

$$\Pr_{a,r,v,w}(\exists\text{ au plus un }i\text{ tel que }g_i(r)=p(q(r)))\geqslant 1-\sqrt{|\Sigma|/(3d)}(3d/|\Sigma|)=1-\sqrt{3d/|\Sigma|}.$$

Donc avec probabilité au moins

$$1-1/(36d)-\sqrt{3d/|\Sigma|}\geqslant 1-2/\sqrt{d}=1-\epsilon^2$$

(pour d suffisamment grand) nous identifions $p\circ q$ de manière unique et renvoyons $p\circ q(0)=p(x)$. Ainsi, $\Pr_{a,r,v,w}(D^f(x,a,b)=p(x))\geqslant 1-\epsilon^2$.

Maintenant, soit A l'événement aléatoire « a est valide pour x » défini par

$$\Pr_{r,v,w}(D^f(x,a,b)=p(x))\geqslant 1-\epsilon\ .$$

Alors

$$\Pr_{a,r,v,w}(D^f(x,a,b)=p(x))=$$
$$\Pr_{a,r,v,w}(D^f(x,a,b)=p(x)\mid a\in A)\Pr(A)+\Pr_{a,r,v,w}(D^f(x,a,b)=p(x)\mid a\notin A)(1-\Pr(A))$$
$$\leqslant \Pr(A)+(1-\epsilon)(1-\Pr(A))=1-\epsilon+\epsilon\Pr(A).$$

Donc $1-\epsilon+\epsilon\Pr(A)\geqslant 1-\epsilon^2$, c'est-à-dire que $\Pr(A)\geqslant 1-\epsilon$. Ainsi, un a aléatoire est valide pour x avec probabilité $1-\epsilon$.
Puisque cet argument fonctionne pour un $x\in\Sigma^k$ arbitraire, on a :

$$1-\epsilon\leqslant \Pr_x\Pr_a(a \text{ est valide pour } x)=\Pr_a\Pr_x(a \text{ est valide pour } x).$$

Donc il existe un point $a\in\Sigma^k$ spécifique qui est valide pour une fraction $\geqslant 1-\epsilon$ des entrées $x\in\Sigma^k$.
Il reste à obtenir le résultat sur toutes les entrées $x\in\Sigma^k$: nous voulons appliquer le décodeur local de Reed-Muller (lemme 12-AJ) afin de retrouver le polynôme p qui, grâce à l'algorithme précédent, a été calculé correctement sur $\geqslant 1-\epsilon>17/18$ des entrées. En d'autres termes, nous voulons exécuter le décodeur et, à chacune de ses requêtes à une valeur de p, lancer l'algorithme précédent pour calculer cette valeur.
Remarquons tout d'abord que si on répète l'algorithme précédent (avec le même conseil correct a) $O(\log(k|\Sigma|))$ fois et qu'on prend la réponse majoritaire, alors la probabilité d'erreur devient $\leqslant(k|\Sigma|)^{-\Omega(1)}$. Cela signifie qu'avec probabilité $\geqslant 5/6$, pour toutes les entrées x sauf une fraction ϵ, toutes les $(k|\Sigma|)^{O(1)}$ requêtes du décodeur local à conseil de Reed-Muller recevront la bonne réponse de l'algorithme précédent.
Ainsi, avec probabilité $\geqslant 5/6$, le décodeur local à conseil de Reed-Muller fonctionne comme s'il avait accès à un oracle calculant le polynôme correct sur une fraction

$$\geqslant 1-\epsilon>17/18$$

des entrées. Au final, la probabilité de succès de la combinaison des deux algorithmes est au moins $1-(1/6+1/6)=2/3$. □

De même, le code de Hadamard possède un décodeur local à conseil. Avant de le décrire, nous avons besoin d'un lemme technique.

12-AP Lemme

Pour deux entiers m et n, soit $s^1,\dots,s^m\in\{0,1\}^n$ des mots aléatoires uniformes et indépendants. Pour $j\in[0,2^m-1]$ et $i\in[1,m]$ on note j_i le i-ème bit de j dans son écriture binaire et on définit

$$r^j=\sum_{\{i\in[1,m]\mid j_i=1\}} s^i.$$

Alors chaque r^j est uniforme sur $\{0,1\}^n$ et les r^j sont deux à deux indépendants.

Démonstration On montre par récurrence sur k qu'une somme finie de k vecteurs indépendants uniformes est encore uniforme. Pour $k=1$ c'est évident. Pour $k>1$: soit $x^1,\dots,x^k$ des vecteurs indépendants uniformes. Par hypothèse de récurrence, $\sum_{i=1}^{k-1} x^i$ est uniforme. On peut alors écrire pour tout $a\in\{0,1\}^n$:

$$\Pr(\sum_{i=1}^{k} x^i = a) = \sum_{b\in\{0,1\}^n} \Pr(\sum_{i=1}^{k-1} x^i = a-b \mid x^k = b)\Pr(x^k=b)$$
$$= \sum_b \Pr(\sum_{i=1}^{k-1} x^i = a-b)2^{-n} = 2^{-n}$$

par uniformité de $\sum_{i=1}^{k-1} x^i$ et par indépendance des x^i, ce qui montre l'hypothèse au rang k.

On montre maintenant que les r^j sont deux à deux indépendants. Soit $j\neq k$ deux entiers de $[0,2^m-1]$ et $a,b\in\{0,1\}^n$. On note $E=\{i\in[1,m]\mid j_i=k_i\}$, c'est-à-dire l'ensemble des indices des vecteurs s^i communs à r^j et r^k, et on note $S=\sum_{i\in E} s^i$. Ainsi, r^j-S et r^k-S sont indépendants (et uniformes). On a donc :

$$\Pr(r^j=a\wedge r^k=b)=\Pr(r^j-S=a-S\wedge r^k-S=b-S)=$$
$$\Pr(r^j-S=a-S)\Pr(r^k-S=b-S)=\Pr(r^j=a)\Pr(r^k=b),$$

ce qui montre l'indépendance de r^j et r^k. □

Le décodage suivant du code de Hadamard est dû à Goldreich et Levin [GL89].

12-AQ Proposition (décodeur local à conseil du code de Hadamard)

Pour tout $\epsilon>0$, le code de Hadamard $\mathsf{H}:\{0,1\}^n\to\{0,1\}^{2^n}$ a un décodeur local à conseil corrigeant $(1/2-\epsilon)$ erreurs en temps $(n/\epsilon)^{O(1)}$ avec conseil de taille $\lceil\log(2n^2/\epsilon^2)\rceil$. La probabilité d'erreur de ce décodeur est majorée par $1/(8n)$.

Démonstration On note $\oplus$ le ou exclusif bit à bit sur $\{0,1\}^n$.

On dispose d'un accès direct à un mot y qui coïncide avec $\mathsf{H}(x)$ sur une fraction $(1/2+\epsilon)$ des bits : c'est-à-dire qu'on peut demander une valeur de $r\cdot x$ pour tout $r\in\{0,1\}^n$ et que la réponse n'est correcte que pour une fraction $(1/2+\epsilon)$ des mots r. L'objectif est de concevoir un algorithme probabiliste qui fournit avec grande probabilité en temps $(n/\epsilon)^{O(1)}$ une liste de taille $\ell=(n/\epsilon)^{O(1)}$ de mots $x^1,\dots,x^\ell\in\{0,1\}^n$ parmi lesquels se trouve x : en effet, il suffira alors de donner comme conseil le numéro i tel que $x^i=x$, conseil dont la taille binaire est $O(\log(n/\epsilon))$.

Idée Si notre mot y était exactement $\mathsf{H}(x)$, il suffirait de demander pour tout $i\in[1,n]$ la valeur de $x\cdot e^i$ (où $e^i=0\dots010\dots0$ possède un unique 1 en position i) pour

obtenir tous les bits de x. Malheureusement, à cause des erreurs cette information est « diluée » dans y : l'idée est de demander $x \cdot (r \oplus e^i)$ pour un mot $r \in \{0,1\}^n$ aléatoire, car $(r \oplus e^i)$ est alors uniformément distribué. Puisque $x \cdot (r \oplus e^i) = (x \cdot r) \oplus (x \cdot e^i)$, en connaissant $x \cdot r$ on en déduit $x \cdot e^i$: il faut donc construire des mots « aléatoires » r pour lesquels on connaît $x \cdot r$. Mais pour réduire la taille du conseil, on va utiliser seulement un nombre logarithmique de « vrais » mots aléatoires $s^1, \ldots, s^m \in \{0,1\}^n$ et définir à partir d'eux 2^m mots $r^1, \ldots, r^{2^m}$ comme au lemme 12-AP. Il suffira alors de donner la liste de tous les mots x' correspondant à toutes les 2^m valeurs possibles de $x \cdot s^i$: cette liste contiendra x avec grande probabilité. Il nous reste à formaliser cette idée.

Formalisation Soit $m = \lceil \log(2n^2/\epsilon^2) \rceil$ et $s^1, \ldots, s^m \in \{0,1\}^n$ des mots aléatoires. Pour $j \in [0, 2^m - 1]$ on définit r^j comme au lemme 12-AP : chaque r^j est donc uniformément distribué sur $\{0,1\}^n$ et ils sont deux à deux indépendants. De plus, $x \cdot r^j = \sum_{\{i \mid j_i = 1\}} x \cdot s^i \pmod 2$ donc on peut calculer $x \cdot r^j$ dès que l'on connaît toutes les valeurs $x \cdot s^i$. On en déduit l'algorithme probabiliste A suivant avec accès direct au mot y qui retourne une liste de candidats pour x :

- choisir $s^1, \ldots, s^m \in \{0,1\}^n$ aléatoirement ;
- pour tout $v \in \{0,1\}^m$ (les 2^m valeurs possibles pour $x \cdot s^i$) faire :
  - pour tout $j \in [0, 2^m - 1]$, calculer $w_j = \sum_{\{i \mid j_i = 1\}} v_i \pmod 2$ (la valeur correspondante de $x \cdot r^j$),
  - pour tous $i \in [1,n]$ et $j \in [0, 2^m - 1]$, consulter y pour connaître la valeur supposée u_i^j de $x \cdot (r^j \oplus e^i)$,
  - pour tout $i \in [1,n]$, on pose $x_i^v =$ la valeur majoritaire (sur tous les j) de $w_j + u_i^j \pmod 2$;
- renvoyer la liste des mots $(x^v)_{v \in \{0,1\}^m}$.

Cet algorithme A renvoie une liste de mots de $\{0,1\}^n$ de taille 2^m, donc le conseil doit être de taille m. Nous allons montrer qu'avec grande probabilité, cette liste contient le mot x.

Dans cette analyse on s'intéresse uniquement à l'itération lors de laquelle la valeur de v vérifie $v_i = x \cdot s^i$ pour tout $i \in [1,n]$, de sorte que $w_j = x \cdot r^j$. Puisque r^j est uniforme par le lemme 12-AP, $(r^j \oplus e^i)$ est un mot uniforme de $\{0,1\}^n$, donc u_i^j est égal à $x \cdot (r^j \oplus e^i)$ avec probabilité $\geqslant (1/2 + \epsilon)$. Ainsi, avec probabilité $\geqslant (1/2 + \epsilon)$, on a

$$w_j + u_i^j = (x \cdot r^j) + (x \cdot (r^j \oplus e^i)) = x \cdot (r^j \oplus (r^j \oplus e^i)) = x \cdot e^i = x_i.$$

Nous montrons maintenant qu'en prenant la réponse majoritaire, alors $x_i^v = x_i$ avec grande probabilité : soit $X_{i,j} \in \{0,1\}$ la variable aléatoire qui vaut 1 ssi $w_j \oplus u_i^j = x_i$ (en particulier, $E(X_{i,j}) \geqslant 1/2 + \epsilon$ et $\mathrm{Var}(X_{i,j}) \leqslant 1/4$, cf. lemme A-K). Pour i fixé, on

pose $S_i = \sum_j X_{i,j}$ le nombre de fois qu'on obtient la bonne valeur pour x_i : on a $E(S) \geqslant (1/2+\epsilon)2^m$ et, les $X_{i,j}$ étant deux à deux indépendantes par le lemme 12-AP, $\mathrm{Var}(S_i) = \sum_j \mathrm{Var}(X_{i,j}) \leqslant 2^m/4$. Par l'inégalité de Tchebychev (théorème A-L), on obtient donc :

$$\Pr(S_i < 2^m/2) \leqslant \Pr(|S_i - E(S_i)| > \epsilon 2^m) \leqslant \frac{2^m/4}{(\epsilon 2^m)^2} = 2^{-m-2}/\epsilon^2.$$

La probabilité que tous les bits x_i soient corrects est donc minorée par $1 - n2^{-m-2}/\epsilon^2 > 1 - 1/(8n)$ par définition de m. Ainsi, x est dans la liste avec probabilité $> 1 - 1/(8n)$.

Il ne reste pour conclure qu'à remarquer que l'algorithme fonctionne en temps polynomial en n/ϵ. □

Nous montrons finalement que la concaténation de deux codes ayant chacun un décodeur local à conseil, possède elle aussi un décodeur local à conseil.

12-AR Lemme

Supposons que $E : \{0,1\}^n \to \Sigma^m$ et $F : \Sigma \to \{0,1\}^q$ possèdent des décodeurs locaux à conseil corrigeant respectivement $(1-\epsilon_E)$ et $(1/2-\epsilon_F)$ erreurs en temps t_E et t_F avec conseils de taille s_E et s_F, où $\epsilon_F < 2\epsilon_E 2^{s_F}$. Supposons en outre que la probabilité d'erreur du décodage de E soit $\leqslant 1/6$ et celle de F soit $\leqslant 1/(8\log|\Sigma|)$.

Alors $F \circ E : \{0,1\}^n \to \{0,1\}^{mq}$ possède un décodeur local à conseil corrigeant $(1/2-\epsilon)$ erreurs en temps $t_E t_F$ avec conseil de taille $s_E + s_F$, où $\epsilon = 4\epsilon_E 2^{s_F}$.

Démonstration Puisque le décodeur de F a une probabilité d'erreur $\leqslant 1/(8\log|\Sigma|)$, on décode correctement tous les $(\log|\Sigma|)$ bits d'un mot codé $y_i \in \Sigma$ avec probabilité $\geqslant 7/8$ en utilisant les mêmes choix de bits aléatoires et le même conseil : on peut ainsi supposer que l'algorithme de décodage de F décode d'un coup un mot complet $y_i \in \Sigma$ avec probabilité $7/8$ et conseil de taille s_F.

On pourra se reporter à la figure 12.2 (dont on conserve ici les notations) pour mieux suivre la démonstration. Pour décoder $F \circ E$ sur l'entrée $z \in \{0,1\}^{mq}$, on va utiliser un couple de conseils (a_E, a_F), où a_E est un conseil pour E et a_F pour F. Le mot z est constitué de m blocs de q bits, le bloc i étant une image avec erreurs par F d'un mot $y_i \in \Sigma$. Pour décoder chaque bit de x, on exécute le décodeur pour E avec conseil a_E, construisant son entrée « à la volée » : à chaque fois qu'il demande une valeur $y_i \in \Sigma$, on décode le i-ème bloc de z en utilisant F avec conseil a_F. La difficulté provient du fait que les conseils pour E et F dépendent du choix des bits aléatoires.

Pour analyser cet algorithme, on fixe l'ensemble des bits aléatoires r_F utilisés lors des différents décodages de F. Montrons qu'il existe un choix de (a_E, a_F) tel que la procédure décrite ci-dessus peut corriger $(1/2-\epsilon)$ erreurs. En effet, supposons que $\Delta(z, F \circ E(x)) < 1/2-\epsilon$: ils coïncident sur au moins $(1/2+\epsilon)qm$ bits. Donc le nombre b de blocs de q bits pour lesquels ils coïncident sur au moins $(1/2+\epsilon_F)q$ bits satisfait

$$bq + (m-b)(1/2+\epsilon_F)q \geqslant (1/2+\epsilon)qm,$$

c'est-à-dire

$$b \geqslant (\epsilon - \epsilon_F)/(1/2 - \epsilon_F)m > 2(\epsilon - \epsilon_F)m > \epsilon m.$$

Soit $N = 2^{s_F}$ le nombre total de conseils a_F possibles : alors $b > 4\epsilon_E N m$ par définition de ϵ, donc il existe un même conseil a_F convenant pour plus de $4\epsilon_E m$ blocs. Pour chaque choix aléatoires r_F fixé ci-dessus, on obtient donc un conseil $a_F(r_F)$ convenant pour plus de $4\epsilon_E m$ blocs.

Ainsi, en prenant r_F au hasard, si Y désigne le nombre de ces $4\epsilon_E m$ blocs qui sont mal décodés à cause de la probabilité d'erreur $1/8$ de l'algorithme pour F, l'espérance de Y est majorée par $4\epsilon_E m/8 = \epsilon_E m/2$. Donc par l'inégalité de Markov (proposition A-I), $\Pr(Y \geqslant 3\epsilon_E m) \leqslant 1/6$: c'est-à-dire qu'avec probabilité $5/6$, le mot $y' \in \Sigma^m$ décodé de z par F avec conseil $a_F(r_F)$ coïncide avec $E(x) \in \Sigma^m$ sur plus de $\epsilon_E m$ positions.

Maintenant, puisque l'erreur sur le mot $E(x)$ est moins de $(1 - \epsilon_E)$, le décodeur pour E avec le conseil a_E (correspondant aux bits aléatoires r_E du décodeur pour E) permet alors de décoder chaque bit de x avec probabilité $\geqslant 5/6$: au total, la probabilité d'erreur finale est majorée par $1/3$. □

12-AS Corollaire

Il existe une constante $c > 0$ telle que, pour $\epsilon = 30(d/|\Sigma|)^{1/6}\log^2|\Sigma|$, $\mathsf{H}\circ\mathsf{RM}$ a un décodeur local à conseil corrigeant $1/2 - \epsilon$ erreurs en temps $(k|\Sigma|)^c$ avec conseil de taille $ck\log|\Sigma|$.

Démonstration La proposition 12-AQ assure le décodage local de $F = \mathsf{H} : \{0,1\}^{\log|\Sigma|} \to \{0,1\}^{|\Sigma|}$ avec probabilité d'erreur majorée par $1/(8\log|\Sigma|)$, corrigeant $(1/2 - \epsilon_F)$ erreurs où $\epsilon_F = 4(d/|\Sigma|)^{1/6}$, en temps $t_F = (|\Sigma|/d)^{O(1)}$ avec conseil de taille

$$s_F = \lceil \log((2\log^2|\Sigma|)/\epsilon_F^2) \rceil.$$

La proposition 12-AO assure le décodage local de $E = \mathsf{RM} : \Sigma^{(1+k/d)^k} \to \Sigma^{|\Sigma|^k}$ corrigeant $(1 - \epsilon_E)$ erreurs où $\epsilon_E = 30\sqrt{d/|\Sigma|}$, en temps $t_E = (k|\Sigma|)^{O(1)}$ avec conseil de taille $(k+1)\log(|\Sigma|)$. En répétant un nombre constant de fois cette procédure, on atteint une probabilité d'erreur $1/6$ avec conseils de taille $s_E = O((k+1)\log|\Sigma|)$.

On a donc

$$2\epsilon_E 2^{s_F} > \epsilon_F \quad \text{et} \quad \epsilon \geqslant 4\epsilon_E 2^{S_F}.$$

Le lemme 12-AR implique que $F \circ E$ possède un décodeur local corrigeant $(1/2 - \epsilon)$ erreurs, fonctionnant en temps $(k|\Sigma|)^{O(1)}$ avec conseil de taille $O(k\log|\Sigma|)$. □

12.4.2 Application

Nous pouvons maintenant appliquer la théorie précédente des codes correcteurs à notre propos. Le résultat suivant d'Impagliazzo et Wigderson [IW97] montre qu'on peut transformer une fonction difficile dans le pire cas en une fonction difficile en moyenne. Avec les lemmes qui précèdent, nous avons suivi la preuve de Sudan, Trevisan et Vadhan [STV99].

12-AT Théorème (Impagliazzo et Wigderson, 1997)

Il existe une constante $a > 0$ telle que, pour toute fonction croissante $s(n) \geqslant n^a$ constructible en temps, s'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ vérifiant $\mathsf{D}_{\mathsf{pire}}(f) \geqslant s(n)$, alors il existe une fonction $g \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{moy}}(g) \geqslant s(n/5)^{4/(15a)}$.

Plus précisément, à partir de toute fonction $f' : \{0,1\}^m \to \{0,1\}$ vérifiant $\mathsf{D}_{\mathsf{pire}}(f') \geqslant s(m)$, on construit une fonction $g' : \{0,1\}^{\alpha(m)} \to \{0,1\}$ telle que $m < \alpha(m) \leqslant 5m$ et $\mathsf{D}_{\mathsf{moy}}(g') > s(m)^{4/(15a)}$. Le calcul de g' se fait en temps $2^{O(m)}$ en effectuant des appels à la fonction f'.

Démonstration

Paramètres Soit $a = 16(c+1)$, où c est la constante du corollaire 12-AS (décodeur local à conseil de $\mathsf{H} \circ \mathsf{RM}$) : donc en particulier $a \geqslant 16$. La restriction de f à $\{0,1\}^n$ est considérée comme un mot sur $\{0,1\}$ de taille 2^n. Soit un entier m et $\Sigma = \mathbb{F}_{2^m}$ le corps à 2^m éléments tels que $2^m = s(n)^{4/a}$. Puisque $s(n) \leqslant n2^n$ (toute fonction f possède des circuits de taille $n2^n$ calculant sa table de vérité), on a donc $|\Sigma| \leqslant (n2^n)^{1/4}$ et $\log|\Sigma| < n/3$. Soit $d = \sqrt{|\Sigma|}$ et k tels que $(1+d/k)^k \log|\Sigma| = 2^n$. En particulier, on remarque que $k \in [5, n]$ donc

$$\left(\sqrt{|\Sigma|}\right)^k \log|\Sigma| \geqslant 2^n, \quad \text{donc} \quad \frac{k}{2}\log|\Sigma| \geqslant n - \log(n/3) \quad \text{et} \quad (1+k)\log|\Sigma| > n.$$

En outre,

$$d = s(n)^{2/a} \geqslant n^2, \quad \text{donc} \quad k \leqslant \sqrt{d} \quad \text{et} \quad d/k \geqslant |\Sigma|^{1/4}$$

et ainsi

$$k\log(|\Sigma|^{1/4}) \leqslant k\log(d/k) \leqslant k\log(1+d/k) + \log\log|\Sigma| = n,$$

donc $k\log|\Sigma| \leqslant 4n$, c'est-à-dire $|\Sigma|^k \leqslant 2^{4n}$. On a donc au final :

$$n < (1+k)\log|\Sigma| \leqslant 5n.$$

Analyse Soit $g = \mathsf{H} \circ \mathsf{RM}(f) \in \{0,1\}^{|\Sigma|^{(k+1)}}$, c'est-à-dire $g : \{0,1\}^{(k+1)\log|\Sigma|} \to \{0,1\}$. La fonction g est calculable en temps $2^{O(n)}$ car $\mathsf{H} \circ \mathsf{RM}$ est calculable en temps polynomial en $|\Sigma|^k$ (cf. lemme 12-AG) : il suffit de calculer $f(x)$ pour tout $x \in \{0,1\}^n$ afin d'obtenir la chaîne caractéristique χ_f de f, puis de calculer $\mathsf{H} \circ \mathsf{RM}(\chi_f)$ en temps polynomial en $|\Sigma|^k \leqslant 2^{4n}$. Au final, $g \in \mathsf{DTIME}(2^{O(n)})$.
Soit $h = \mathsf{D}_{\mathsf{moy}}(g)$ et supposons $h < |\Sigma|^{1/15} = s(n)^{4/(15a)}$: g peut donc être calculée par un circuit C de taille $h+1$ sur une fraction $\geqslant 1/2 + 1/h$ de ses entrées. Soit $x \in \{0,1\}^n$: nous voulons calculer $f(x)$. Il suffit d'exécuter le décodeur local à conseil pour $\mathsf{H} \circ \mathsf{RM}$ du corollaire 12-AS : en effet,

$$30(d/|\Sigma|)^{1/6}\log^2|\Sigma| < 1/h \quad \text{puisque} \quad (d/|\Sigma|)^{1/6} = |\Sigma|^{-1/12}.$$

En répétant $O(n)$ fois ce décodeur, la probabilité d'erreur devient $< 2^{-n}$ donc il existe un choix de bits aléatoires fonctionnant pour tout $x \in \{0,1\}^n$. Pour ce choix, le conseil du décodeur est de taille $n \times (ck\log|\Sigma|)$ et le temps pour décoder est $n(k|\Sigma|)^c$. Un tel

décodeur peut être transformé en un circuit D pour f de taille $O(n^2(k|\Sigma|)^{2c})$ avec des appels à la fonction g (cf. proposition 5-Y) : au final, si les appels à g sont remplacés par le circuit C, on obtient un circuit pour f de taille $O(n^2(k|\Sigma|)^{2c}(h+1))$. C'est une contradiction puisque $n^2(k|\Sigma|)^{2c}(h+1) = o(s(n))$, car

$$n^2(k|\Sigma|)^{2c}h \leqslant n^{2c+2}s(n)^{8c/a+4/(15a)} \leqslant n^{a/8}s(n)^{9/15} = o(s(n)).$$

□

Avec le corollaire 12-S, on obtient donc directement le résultat suivant.

12-AU Corollaire

- S'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant 2^{\epsilon n}$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} = \mathsf{P}$.
- S'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant 2^{n^\epsilon}$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} \subseteq \bigcup_k \mathsf{DTIME}(2^{(\log n)^k})$.
- Si pour tout $c > 0$ il existe $f_c \in \mathsf{DTIME}(2^{O(n)})$ vérifiant $\mathsf{D}_{\mathsf{pire}}(f_c) \geqslant n^c$, alors $\mathsf{BPP} \subseteq \bigcap_{\epsilon>0} \mathsf{DTIME}(2^{n^\epsilon})$.

Nous pouvons reformuler cela en termes de classes de complexité. On rappelle qu'on désigne par $\mathsf{SIZE}(s(n))$ l'ensemble des langages calculés par une famille de circuits booléens de taille $O(s(n))$, que $\mathsf{EXP} = \mathsf{DTIME}(2^{n^{O(1)}})$ et que $\mathsf{E} = \mathsf{DTIME}(2^{O(n)})$.

12-AV Corollaire

- Si $\mathsf{E} \not\subset \mathsf{SIZE}(2^{\epsilon n})$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} = \mathsf{P}$.
- Si $\mathsf{EXP} \not\subset \mathsf{SIZE}(2^{n^\epsilon})$ pour un certain $\epsilon > 0$, alors $\mathsf{BPP} \subseteq \bigcup_k \mathsf{DTIME}(2^{(\log n)^k})$.
- Si $\mathsf{EXP} \not\subset \mathsf{P/poly}$ alors $\mathsf{BPP} \subseteq \bigcap_{\epsilon>0} \mathsf{DTIME}(2^{n^\epsilon})$.

Démonstration
- Si $\mathsf{E} \not\subset \mathsf{SIZE}(2^{\epsilon n})$ alors il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant \alpha 2^{\epsilon n}$ pour une constante $\alpha > 0$, donc $\mathsf{D}_{\mathsf{pire}}(f) \geqslant 2^{(\epsilon/2)n}$ pour n assez grand, ce qui permet d'appliquer le corollaire 12-AU.
- Si $\mathsf{EXP} \not\subset \mathsf{SIZE}(2^{n^\epsilon})$ alors il existe $f \in \mathsf{DTIME}(2^{n^k})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant \alpha 2^{n^\epsilon}$ pour une constante $\alpha > 0$; alors la fonction g définie par $g(x0^{n^k}) = f(x)$ est dans $\mathsf{DTIME}(2^{O(n)})$ et satisfait $\mathsf{D}_{\mathsf{pire}}(g) \geqslant \alpha 2^{n^{\epsilon/k}} \geqslant 2^{n^{\epsilon/(2k)}}$ pour n assez grand, ce qui permet d'appliquer le corollaire 12-AU.

– Si $\mathsf{EXP} \not\subseteq \mathsf{P/poly}$ alors il existe $f \in \mathsf{DTIME}(2^{n^k})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant n^c$ pour tout c ; alors la fonction g définie par $g(x0^{n^k}) = f(x)$ est dans $\mathsf{DTIME}(2^{O(n)})$ et satisfait $\mathsf{D}_{\mathsf{pire}}(g) \geqslant n^{c/k}$ pour tout c, ce qui permet d'appliquer le corollaire 12-AU. □

Plus généralement, la combinaison des théorèmes 12-F, 12-L et 12-AT donne le corollaire suivant.

12-AW Corollaire

Il existe une constante $a > 0$ telle que si l'on a accès à une fonction $f : \{0,1\}^n \to \{0,1\}$ vérifiant $\mathsf{D}_{\mathsf{pire}}(f) \geqslant n^k$ pour $k \geqslant a$, alors on peut calculer en temps $2^{O(\alpha(n))}$ une fonction $h : \{0,1\}^{\alpha(n)} \to \{0,1\}^{\alpha(n)^{k/(30a)}}$ qui est $\alpha(n)^{k/(30a)}$-imprédictible, où $\alpha(n) = \Theta(n^2/\log n)$.

En particulier, s'il existe une fonction $f \in \mathsf{DTIME}(2^{O(n)})$ telle que $\mathsf{D}_{\mathsf{pire}}(f) \geqslant n^k$ pour $k \geqslant a$, alors pour toute fonction $u : \mathbb{N} \to \mathbb{N}$ calculable en temps $O(2^{u(n)})$, on a

$$\mathsf{BPTIME}(o(u(n)^{k/(30a)})) \subseteq \mathsf{DTIME}(2^{O(u(n))}).$$

Démonstration Soit a la constante du théorème 12-AT. Le théorème 12-AT permet de calculer en temps $2^{O(n)}$ à partir de f une fonction $g : \{0,1\}^m \to \{0,1\}$ telle que $\mathsf{D}_{\mathsf{moy}}(g) \geqslant s(m) = (m/5)^{4k/(15a)}$, où $n < m \leqslant 5n$.

Pour $d(m) = (k/15a)\log m$, qui vérifie donc pour m suffisamment grand :

$$s\left(\sqrt{md(m)/12}\right) > 2^{2d(m)},$$

le théorème 12-L permet de calculer en temps $2^{O(\gamma(m))}$ à partir de g une fonction $h : \{0,1\}^{\gamma(m)} \to \{0,1\}^{\gamma(m)^{k/(30a)}}$ qui est $\gamma(m)^{k/(30a)}$-imprédictible, où

$$\gamma(m)d(\gamma(m)) = \Theta(m^2), \text{ c'est-à-dire } \gamma(m) = \Theta(m^2/\log m).$$

On pose $\alpha(n) = \gamma(m)$, donc $\alpha(n) = \Theta(n^2/\log n)$: on dispose donc d'une fonction $h : \{0,1\}^{\alpha(n)} \to \{0,1\}^{\alpha(n)^{k/(30a)}}$ qui est $\alpha(n)^{k/(30a)}$-imprédictible. De plus, $h \in \mathsf{DTIME}(2^{O(n)})$ si $f \in \mathsf{DTIME}(2^{O(n)})$.

Le théorème 12-F, pour $m(n) = n^{k/(30a)}$ et $t(n) = 2^{O(n)}$, montre que

$$\mathsf{BPTIME}(o(u(n)^{k/(30a)})) \subseteq \mathsf{DTIME}(2^{O(u(n))}).$$

□

12.5 Dérandomisation implique borne inférieure

Nous nous intéressons maintenant à la « réciproque » de ce qui précède. Nous avons montré que les bornes inférieures non uniformes impliquaient une dérandomisation, nous

montrons dorénavant qu'une dérandomisation implique une certaine borne inférieure non uniforme. Comme nous verrons, cette borne inférieure est « hybride » en ceci qu'elle porte soit sur des circuits booléens, soit sur des circuits arithmétiques. Il est ouvert de montrer, sous hypothèse de dérandomisation, une borne inférieure non déjà connue et qui soit purement booléenne.

Nous avons besoin d'un premier ingrédient ressemblant aux résultats montrés au chapitre précédent, dont la technique de preuve est intéressante en soi. Il est dû à Impagliazzo, Kabanets et Wigderson [IKW01].

12-AX Définition

Soit $\mathscr{C}$ et $\mathscr{C}'$ deux classes de complexité. On dit que $\mathscr{C} \subseteq_{\text{i.s.}} \mathscr{C}'$ (pour « inclus infiniment souvent ») si pour tout langage $L \in \mathscr{C}$, il existe un langage $L' \in \mathscr{C}'$ tel que pour une infinité d'entiers n, $L^{=n} = L'^{=n}$.

12-AY Théorème (Impagliazzo, Kabanets et Wigderson, 2001)

$\mathsf{NEXP} \subset \mathsf{P/poly} \implies \mathsf{NEXP} = \mathsf{EXP}$.

Idée de la démonstration Si $\mathsf{NEXP} \neq \mathsf{EXP}$ alors les certificats non déterministes pour un langage $L \in \mathsf{NEXP} \setminus \mathsf{EXP}$ ne sont pas calculables en temps déterministe exponentiel, donc ne possèdent pas de circuits de taille n^k (pour k fixé). On peut ainsi se servir d'un tel certificat pour dérandomiser un algorithme probabiliste grâce au corollaire 12-AW. Sous l'hypothèse $\mathsf{NEXP} \subset \mathsf{P/poly}$, le corollaire 11-N donne $\mathsf{EXP} = \mathsf{MA}$. Or tout protocole MA peut maintenant être dérandomisé comme suit : deviner le message envoyé par Merlin, deviner un certificat y pour L n'ayant pas de circuits de taille n^k, se servir de y pour dérandomiser l'algorithme d'Arthur. On a donc un algorithme non déterministe fonctionnant en temps 2^{n^c} pour c fixé pour tout langage de $\mathsf{MA} = \mathsf{EXP}$. Or tout algorithme de ce type a des circuits de taille n^d pour d fixé si $\mathsf{NEXP} \subset \mathsf{P/poly}$, donc $\mathsf{EXP} \subset \mathsf{SIZE}(n^d)$, une contradiction avec le proposition 11-A.

Démonstration Nous allons aboutir à une contradiction en supposant à la fois $\mathsf{NEXP} \neq \mathsf{EXP}$ et $\mathsf{NEXP} \subset \mathsf{P/poly}$. La preuve va se dérouler en deux étapes. Dans un premier temps, on montre en utilisant les théorèmes des sections précédentes que si $\mathsf{NEXP} \neq \mathsf{EXP}$ alors il existe une constante γ telle que pour tout $k \geqslant 1$, $\mathsf{BPTIME}(o(n^k)) \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{O(n^{\gamma/k})})/n^{1/k}$ (temps non déterministe $2^{O(n^{\gamma/k})}$ avec conseil de taille $n^{1/k}$). Puis en passant par les protocoles MA on montre que $\mathsf{EXP} \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{n^\gamma})/n$, ce qui implique, sous l'hypothèse $\mathsf{NEXP} \subset \mathsf{P/poly}$, que $\mathsf{EXP} \subseteq_{\text{i.s.}} \mathsf{SIZE}(n^\delta)$ pour une constante δ fixée, une contradiction avec la proposition 11-A.

Première étape Nous commençons donc par montrer, sous l'hypothèse $\mathsf{NEXP} \neq \mathsf{EXP}$, que tout langage $B \in \mathsf{BPTIME}(o(n^k))$ est reconnu pour une infinité de longueurs n

par un algorithme $\mathsf{NTIME}(2^{O(n^{\gamma/k})})/n^{1/k}$ (pour une certaine constante γ indépendante de k). Soit $L \in \mathsf{NEXP} \setminus \mathsf{EXP}$: il existe un langage $A \in \mathsf{P}$ et une constante γ tels que

$$x \in L \iff \exists y \in \{0,1\}^{2^{|x|^{\gamma/2}}} (x,y) \in A.$$

On note a la constante du corollaire 12-AW et $n = |x|$. Supposons d'abord, par l'absurde, que tous les certificats de L, sauf éventuellement pour un nombre fini d'entrées, puissent être calculés par des circuits de taille n^{30ak}, c'est-à-dire qu'on dispose, pour presque tout x de taille n, d'un circuit de taille n^{30ak} qui sur l'entrée $i \in [0, 2^{n^{\gamma/2}} - 1]$ en binaire renvoie le i-ème bit d'un mot y censé certifier que $x \in L$. Alors on peut décider $L \in \mathsf{EXP}$ de la manière suivante [2] : sur l'entrée x de taille n, on énumère tous les circuits C de taille n^{30ak}, on les simule sur tout $i \in [0, 2^{n^{\gamma/2}} - 1]$ pour connaître le mot y qu'ils calculent, et on vérifie si $(x,y) \in A$. C'est une contradiction avec le fait que $L \notin \mathsf{EXP}$, ce qui signifie que pour une infinité de mots x les certificats de L n'admettent pas de petits circuits. Donc il existe une suite infinie de mots $(x_j^k)_{j \in \mathbb{N}}$ tels que :

- pour tout j, $x_j^k \in L$;
- aucun $y \in \{0,1\}^{2^{|x_j^k|^{\gamma/2}}}$ tel que $(x_j^k, y) \in A$ n'est calculé par un circuit C de taille $|x_j^k|^{30ak}$.

Si on voit les mots y comme des fonctions $y : \{0,1\}^{\ell^{\gamma/2}} \to \{0,1\}$ pour $\ell = |x_j^k|$, le second point peut être reformulé en $\mathsf{D}_{\mathsf{pire}}(y) > \ell^{30ak}$.

Soit $B \in \mathsf{BPP}$ reconnu par une machine $M(x,r)$ fonctionnant en temps $o(n^k)$. Pour mieux comprendre ce qui suit, il faut garder en tête que nous comptons appliquer le théorème 12-F de Yao avec les paramètres $u(n) = n^{1/k}$ et $m(n) = n^{k^2}$ (de sorte que $m(u(n)) = n^k$). On fixe j arbitraire, on note $x_0 = x_j^{k^2}$, on pose $\ell = |x_0|$ et on va s'intéresser aux entrées x de taille $n = \ell^k$. Les considérations qui précèdent donnent pour B l'algorithme non déterministe suivant sur une entrée $x \in \{0,1\}^n$, avec conseil x_0 :

- deviner $y \in \{0,1\}^{2^{\ell^{\gamma/2}}}$;
- si $(x_0, y) \notin A$, rejeter ;
- sinon $\mathsf{D}_{\mathsf{pire}}(y) > \ell^{30ak^2}$ donc par le corollaire 12-AW, pour $\alpha(\ell) = \Theta(\ell^\gamma / \log \ell)$, on peut calculer en temps $2^{O(\alpha(\ell))}$ à partir de y une fonction $h : \{0,1\}^{\alpha(\ell)} \to \{0,1\}^{\alpha(\ell)^{k^2}}$ qui est $\alpha(\ell)^{k^2}$-imprédictible, c'est-à-dire au moins n^k-imprédictible ;

2. Les mots x en nombre fini dont les certificats n'ont pas de petits circuits sont codés en dur dans l'algorithme et traités à part.

– en énumérant tous les mots $z \in \{0,1\}^{\alpha(\ell)}$, calculer

$$q = \Pr_z(M(x, h(z)) = 1)$$

et accepter ssi $q \geqslant 1/2$.

Le théorème 12-F de Yao montre que cet algorithme non déterministe reconnaît le langage B sur les entrées de taille $n = \ell^k$, et il fonctionne en temps $2^{O(\ell^\gamma)}$ avec conseil de taille ℓ. En d'autres termes, $\mathsf{BPTIME}(o(n^k)) \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{O(n^{\gamma/k})})/n^{1/k}$. Cela conclut la première étape.

Seconde étape Nous montrons maintenant que sous l'hypothèse $\mathsf{NEXP} \subset \mathsf{P/poly}$, on a $\mathsf{EXP} \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{n^\gamma})/n$. On a bien sûr $\mathsf{EXP} \subset \mathsf{P/poly}$ donc par le corollaire 11-N, $\mathsf{EXP} = \mathsf{MA}$. Il suffit donc de montrer que $\mathsf{MA} \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{n^\gamma})/n$. Soit $L \in \mathsf{MA}$ reconnu par un protocole où Merlin envoie une preuve y de taille n^k et Arthur la vérifie en temps probabiliste polynomial : sur l'entrée (x, y), si $x \notin L$ ou si $x \in L$ et y est une preuve valide, alors Arthur a un comportement de type BPP. Sans perte de généralité, on peut supposer qu'il s'agit d'un comportement de type $\mathsf{BPTIME}(o(n^{k'}))$ pour $k' > k$. Par ce qui précède,

$$\mathsf{BPTIME}(o(n^{k'})) \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{O(n^{\gamma/k'})})/n^{1/k'}.$$

Par un algorithme non déterministe avec conseil, sur une infinité de tailles d'entrées on peut donc deviner la preuve y de Merlin et simuler Arthur dans $\mathsf{NTIME}(2^{O(n^{\gamma/k'})})/n^{1/k'}$ sur son entrée (x, y) de taille $n + n^k$. Puisque $k' > k$, l'algorithme fonctionne en temps $O(2^{|x|^\gamma})$ avec conseil de taille $\leqslant |x|$. Donc $\mathsf{MA} \subseteq_{\text{i.s.}} \mathsf{NTIME}(2^{n^\gamma})/n$, ce qui conclut la seconde partie.

Conclusion Or $\mathsf{NEXP} \subset \mathsf{P/poly}$ implique que $\mathsf{NTIME}(2^{n^\gamma})/n \subset \mathsf{SIZE}(n^\delta)$ pour une constante fixée δ : en effet, tout langage $L \in \mathsf{NTIME}(2^{n^\gamma})$ se réduit au langage

$$A = \{(N, x, t) \mid N(x) \text{ accepte en temps } \leqslant t\}$$

(où N est une machine non déterministe et t est codé en binaire) par une réduction f_L : $x \mapsto (N_L, x, t_L)$ où N_L est une machine pour L et $t_L = O(2^{|x|^\gamma})$ son temps d'exécution. La réduction est donc de taille $O(|x|^\gamma)$. Puisque $\mathsf{NEXP} \subset \mathsf{P/poly}$, il existe une certaine constante β telle que $A \in \mathsf{SIZE}(n^\beta)$, et on a donc $L \in \mathsf{SIZE}(n^\delta)$ où $\delta = \gamma\beta$; ajouter le conseil de taille n se fait en le codant en dur dans les circuits, donc $\mathsf{NTIME}(2^{n^\gamma})/n \subseteq \mathsf{SIZE}(n^\delta)$.
On en déduit que $\mathsf{MA} = \mathsf{EXP} \subseteq_{\text{i.s.}} \mathsf{SIZE}(n^\delta)$, ce qui contredit la proposition 11-A. □

Nous avons maintenant besoin d'un lemme sur des circuits arithmétiques pour le permanent. Pour un rappel sur les circuits arithmétiques, voir la section 5.6, sur le permanent voir la section 9.5, et sur le test d'identité de polynômes TIP voir la section 6.3.1.

12-AZ Lemme

Soit A le problème consistant à décider, étant donné un circuit arithmétique C à n^2 entrées, si C calcule per_n.

Si $\mathtt{TIP} \in \mathsf{P}$ alors $A \in \mathsf{P}$.

Démonstration On utilise le développement selon les lignes et colonnes du permanent : en effet, si $M = (m_{i,j})_{i,j\in[1,n]}$ est une matrice de taille $n \times n$ et que $M_{i,j}$ désigne la sous-matrice de taille $(n-1)\times(n-1)$ où l'on a enlevé la i-ème ligne et la j-ème colonne, alors

$$\mathrm{per}_n(M) = \sum_{i=1}^{n} m_{i,n}\mathrm{per}_{n-1}(M_{i,n}).$$

Pour vérifier que C calcule le permanent, il suffit donc de tester si cette égalité est vraie à tous les rangs de n à 1, puis de vérifier que pour les matrices de taille 1 on a $C(M) = m_{1,1}$.

Pour $m \in [1,n]$, on notera C^m le circuit à m^2 entrées $X = (x_{i,j})_{i,j\leqslant m}$ qui est égal à C calculé sur la matrice

$$\begin{pmatrix} X & 0 \\ 0 & I \end{pmatrix}$$

où I désigne la matrice identité de taille $n-m$. Si C calcule per_n, alors C^m calcule per_m. Il s'agit alors de vérifier pour tout $m \in [1,n]$, grâce au problème $\mathtt{TIP}$, que

$$C^m(X) = \sum_{i=1}^{m} x_{i,m} C^{m-1}(X_{i,m})$$

et que $C^1(x) = x$, ce qui se fait en temps polynomial si $\mathtt{TIP} \in \mathsf{P}$. □

Nous sommes maintenant prêts pour montrer la borne inférieure principale de cette section, due à Kabanets et Impagliazzo [KI03].

12-BA Théorème (Kabanets et Impagliazzo, 2003)

Si $\mathtt{TIP} \in \mathsf{P}$ alors $\mathsf{NEXP} \not\subset \mathsf{P/poly}$ ou le permanent n'a pas de circuits arithmétiques de taille polynomiale.

Démonstration Par l'absurde, supposons qu'on ait à la fois $\mathtt{TIP} \in \mathsf{P}$, $\mathsf{NEXP} \subset \mathsf{P/poly}$ et le permanent a des circuits arithmétiques de taille polynomiale. Alors le théorème 12-AY et le corollaire 11-N impliquent que $\mathsf{NEXP} = \mathsf{EXP} = \mathsf{MA}$. Or $\mathsf{MA} \subseteq \mathsf{PH}$ (corollaire 10-AB), $\mathsf{PH} \subseteq \mathsf{P}^{\sharp\mathsf{P}}$ (théorème 9-AI de Toda) et le problème $\mathtt{PERMANENT}_{\{0,1\}}$ est $\sharp\mathsf{P}$-complet (théorème 9-AP), donc

$$\mathsf{NEXP} \subseteq \mathsf{P}^{\mathtt{PERMANENT}_{\{0,1\}}}.$$

Or nous allons montrer, sous nos hypothèses, que tout langage $L \in \mathsf{P}^{\mathtt{PERMANENT}_{\{0,1\}}}$ appartient à NP, ce qui aboutit à une contradiction puisque $\mathsf{NEXP} \not\subseteq \mathsf{NP}$ (théorème 2-AI).

Sur l'entrée x, l'algorithme polynomial avec oracle $\texttt{PERMANENT}_{\{0,1\}}$ pour L fait des appels à son oracle de taille au plus $p(|x|)$ pour un certain polynôme p. On peut décider si $x \in L$ ainsi de manière non déterministe :

- deviner un circuit arithmétique C de taille polynomiale pour $\mathrm{per}_{p(|x|)}$;
- tester comme au lemme 12-AZ que C calcule le permanent ;
- simuler l'algorithme polynomial pour L en remplaçant les appels à l'oracle par une évaluation du circuit C.

Puisque par hypothèse le permanent a des circuits arithmétiques de taille polynomiale, et que tester si C calcule le permanent se fait en temps polynomial par le lemme 12-AZ car $\texttt{TIP} \in \mathsf{P}$, cet algorithme non déterministe décide le langage L et donc $L \in \mathsf{NP}$. □

A Probabilités et arithmétique

Nous donnons ici un simple aide-mémoire des notions minimales utiles dans ce livre en probabilités, ainsi que deux résultats d'arithmétique dont nous nous servons. Pour approfondir le domaine des probabilités, nous conseillons par exemple le livre de Jacod et Protter [JP02], en particulier les chapitres traitant des espaces finis ou dénombrables.

A.1 Probabilités

On s'intéressera ici uniquement au cas très simplifié des espaces probabilistes finis. Un *espace probabiliste fini* est un ensemble $\Omega = \{a_1, \ldots, a_n\}$ appelé *univers* auquel on associe des réels $p_1, \ldots, p_n \in [0,1]$ tels que $\sum_i p_i = 1$. Le réel p_i est la probabilité de a_i, que l'on écrira $\Pr(a_i)$. Si $S \subseteq \Omega$, alors la probabilité de l'ensemble S est $\Pr(S) = \sum_{a_i \in S} p_i$. Un tel ensemble S est appelé *événement*.

Nous commençons par un lemme évident qui récapitule quelques propriétés de base.

A-A Lemme

Soit A et B des événements sur un univers Ω.

- $\Pr({}^cA) = 1 - \Pr(A)$.
- Si A et B sont disjoints alors $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.
- Si A est un événement tel que $\Pr(A) > 0$, alors $A \neq \emptyset$.

Nous continuons par une inégalité très simple aussi connue sous son nom anglais de *union bound*.

A-B Proposition (inégalité de Boole)

Si $A_1,\dots,A_n$ sont des événements sur un univers Ω, alors

$$\Pr(\bigcup_{i=1}^{n} A_i) \leqslant \sum_{i=1}^{n} \Pr(A_i).$$

Démonstration Si l'on appelle $p_1,\dots,p_n$ les probabilités des éléments $a_1,\dots,a_n$ de l'univers Ω, alors $\Pr(\bigcup_{i=1}^{n} A_i) = \sum_{a_j \in \cup_i A_i} p_j$. Chaque élément a_j de la somme apparaît dans au moins un événement A_i, donc cette somme est majorée par

$$\sum_{i=1}^{n} (\sum_{a_j \in A_i} p_j) = \sum_{i=1}^{n} \Pr(A_i).$$

□

On dit que des événements $S_1,\dots,S_n$ sont *indépendants* si

$$\forall T \subseteq \{1,\dots,n\},\ \Pr(\cap_{i\in T} S_i) = \prod_{i\in T} \Pr(S_i).$$

Le résultat suivant nous servira pour évaluer le nombre de répétitions nécessaires afin d'obtenir un succès dans une expérience.

A-C Lemme

Soit $A_1,\dots,A_n$ des événements indépendants se produisant chacun avec probabilité $p > 0$. Alors

$$\Pr(\cup_{i=1}^{n} A_i) = 1 - (1-p)^n.$$

En particulier, si $n > 2/p$ alors $\Pr(\cup_{i=1}^{n} A_i) \geqslant 2/3$.

Démonstration $\Pr(\cup A_i) = 1 - \Pr(\cap({}^c A_i)) = 1 - (1-p)^n$ par indépendance.

La fonction $(1-1/x)^x$ est croissante sur $[1,+\infty[$ et tend vers $1/e$ quand x tend vers l'infini. Donc pour tout $x \geqslant 1$,

$$(1-1/x)^{2x} < 1/e^2 :$$

en d'autres termes, si $n > 2/p$ alors $(1-p)^n < 1/e^2$, donc

$$\Pr(\cup A_i) > 1 - 1/e^2 > 2/3.$$

□

Si A et B sont deux événements sur un univers Ω tels que $\Pr(B) \neq 0$, on définit la *probabilité conditionnelle* de A sachant B par

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}.$$

On a donc immédiatement le résultat suivant.

A-D Lemme

Si A et B sont deux événements tels que $\Pr(B) \neq 0$, alors $\Pr(A \cap B) = \Pr(A|B)\Pr(B)$.

Plus généralement, le *conditionnement par cas* désigne la formule suivante.

A-E Lemme

Sur un univers Ω, soit A un événement et $B_1, \ldots, B_n$ une partition de Ω. Alors

$$\Pr(A) = \sum_{i=1}^{n} \Pr(A|B_i)\Pr(B_i).$$

Démonstration Puisque $B_1, \ldots, B_n$ partitionne Ω, les ensembles $A \cap B_i$ sont disjoints et $A = \cup_i (A \cap B_i)$. Donc

$$\Pr(A) = \sum_{i=1}^{n} \Pr(A \cap B_i) = \sum_{i=1}^{n} \Pr(A|B_i)\Pr(B_i).$$

□

Une *variable aléatoire* est une fonction X de Ω dans $\mathbb{R}$, qui associe donc un réel x_i à chaque a_i. Si $R \subseteq \mathbb{R}$, la probabilité que X prenne une valeur dans R est $\Pr(X \in R) = \Pr(X^{-1}(R))$. On peut donc voir une variable aléatoire comme une variable prenant des valeurs réelles avec certaines probabilités. Un type de variables aléatoires nous sera particulièrement utile.

A-F Définition

On dit qu'une variable aléatoire X suit une loi de Bernoulli de paramètre p si X est à valeurs dans $\{0, 1\}$ et vérifie :

$$X = \begin{cases} 1 & \text{avec probabilité } p \\ 0 & \text{avec probabilité } 1 - p. \end{cases}$$

L'*espérance* de X est sa valeur moyenne, c'est-à-dire le réel $E(X) = \sum_{i=1}^{n} x_i \Pr(X = x_i)$. Il est bien connu que l'espérance est linéaire comme le montre le lemme suivant (en particulier, $E(\sum_{i=1}^{k} X_i) = \sum_{i=1}^{k} E(X_i)$).

A-G Lemme (linéarité de l'espérance)

Si α est une constante et X et Y des variables aléatoires, alors $E(\alpha X + Y) = \alpha E(X) + E(Y)$.

Démonstration Si les valeurs prises par X sont $x_1, \ldots, x_n$ et celles prises par Y sont $y_1, \ldots, y_m$, alors

$$\begin{aligned} E(\alpha X + Y) &= \sum_{i,j} (\alpha x_i + y_j) \Pr(X = x_i \wedge Y = y_j) \\ &= \alpha \sum_i x_i \sum_j \Pr(X = x_i \wedge Y = y_j) + \sum_j y_j \sum_i \Pr(X = x_i \wedge Y = y_j) \\ &= \alpha \sum_i x_i \Pr(X = x_i) + \sum_j y_j \Pr(Y = y_j) = \alpha E(X) + E(Y). \end{aligned}$$

□

Des variables aléatoires $X_1, \ldots, X_n$ sont dites *indépendantes* si pour tous sous-ensembles $R_1, \ldots, R_n$ de $\mathbb{R}$, les événements $X_i^{-1}(R_i)$ sont indépendants.

A-H Lemme

Si $X_1, \ldots, X_n$ sont des variables aléatoires indépendantes, alors $E(\prod_{i=1}^{n} X_i) = \prod_{i=1}^{n} E(X_i)$.

Démonstration Nous donnons la preuve pour deux variables aléatoires indépendantes X et Y, le cas général étant similaire. On suppose que X prend les valeurs $x_1, \ldots, x_n$ et Y les valeurs $y_1, \ldots, y_m$. On a alors

$$E(XY) = \sum_{i,j} \Pr(X = x_i \wedge Y = y_j) x_i y_j.$$

Par indépendance de X et Y,

$$\Pr(X = x_i \wedge Y = y_j) = \Pr(X = x_i) \Pr(Y = y_j),$$

donc

$$\begin{aligned} E(XY) = \sum_{i,j} \Pr(X = x_i) \Pr(Y = y_j) x_i y_j = \\ (\sum_i \Pr(X = x_i) x_i)(\sum_j \Pr(Y = y_j) y_j) = E(X) E(Y). \end{aligned}$$

□

A-I **Proposition (inégalité de Markov)**

Si X est une variable aléatoire *à valeurs positives*, alors pour tout $t > 0$,

$$\Pr(X \geqslant t) \leqslant E(X)/t.$$

Démonstration $E(X) = \sum_i p_i x_i = \sum_{x_i<t} p_i x_i + \sum_{x_i \geqslant t} p_i x_i \geqslant t \sum_{x_i \geqslant t} p_i = t \Pr(X \geqslant t)$. □

Lorsqu'on possède plus d'information sur la distribution de X, on peut généralement montrer une inégalité meilleure que celle de Markov. Par exemple, celle de Bienaymé-Tchebychev utilise la variance que nous définissons maintenant. La variance mesure la dispersion de X autour de sa moyenne.

A-J **Définition (variance)**

Soit X une variable aléatoire. La variance de X est le réel

$$\mathrm{Var}(X) = E(X^2) - E(X)^2.$$

A-K **Lemme**

- Si $a \in \mathbb{R}$ est une constante et X une variable aléatoire, alors $\mathrm{Var}(aX) = a^2 \mathrm{Var}(X)$.
- La variance d'une variable aléatoire X qui suit une loi de Bernoulli de paramètre p est $p(1-p)$, donc en particulier $\mathrm{Var}(X) \leqslant 1/4$.
- Si $X_1, \ldots, X_n$ sont des variables aléatoires deux à deux indépendantes, alors

$$\mathrm{Var}\Big(\sum_{i=1}^{n} X_i\Big) = \sum_{i=1}^{n} \mathrm{Var}(X_i).$$

Démonstration

– En utilisant la linéarité de l'espérance, on a :

$$\mathrm{Var}(aX) = E(a^2X^2) - E(aX)^2 = a^2E(X^2) - a^2E(X)^2 = a^2\mathrm{Var}(X).$$

– X^2 vaut 1 avec probabilité p et 0 sinon, donc $E(X^2) = p$. Puisque $E(X) = p$, on a donc

$$\mathrm{Var}(X) = p - p^2 = p(1-p).$$

Cette quantité est maximale quand $p = 1/2$ et vaut alors $1/4$.

– On a :

$$\mathrm{Var}(\sum_i X_i) = E((\sum_i X_i)^2) - E(\sum_i X_i)^2 = \sum_i E(X_i^2) + 2\sum_{i<j} E(X_i X_j) - \sum_i E(X_i)^2 - 2\sum_{i<j} E(X_i)E(X_j).$$

Or X_i et X_j sont indépendantes si $i < j$, donc le lemme A-H donne $E(X_iX_j) = E(X_i)E(X_j)$ donc

$$\mathrm{Var}(\sum_i X_i) = \sum_i E(X_i^2) - \sum_i E(X_i)^2 = \sum_i \mathrm{Var}(X_i).$$

□

A-L **Théorème (inégalité de Bienaymé-Tchebychev)**

Soit X une variable aléatoire. Alors pour tout réel $t > 0$, on a

$$\Pr(|X - E(X)| \geqslant t) \leqslant \frac{\mathrm{Var}(X)}{t^2}.$$

Démonstration Soit $Y = (X - E(X))^2$. En utilisant le fait que $E(X)$ est une constante (donc $E(E(X)) = E(X)$) et la linéarité de l'espérance, on a

$$E(Y) = E(X^2) - 2E(X)^2 + E(X)^2 = \mathrm{Var}(X).$$

L'inégalité de Markov donne donc :

$$\Pr(|X - E(X)| \geqslant t) = \Pr(Y \geqslant t^2) \leqslant \frac{E(Y)}{t^2} = \frac{\mathrm{Var}(X)}{t^2}.$$

□

Si on suppose plus de régularité dans nos variables aléatoires, on peut encore montrer de meilleures inégalités. Dans ce livre nous n'aurons besoin que d'une version faible des bornes de Chernoff dont la démonstration est facile, mais nous donnons tout de même aussi la version habituelle plus bas (sans démonstration).

A-M **Proposition (version faible des bornes de Chernoff)**

Soit $p \in [0,1]$ et $X_1, \ldots, X_n$ des variables aléatoires indépendantes telles que

$$X_i = \begin{cases} 1 & \text{avec probabilité } p \\ 0 & \text{avec probabilité } (1-p). \end{cases}$$

Soit $X = \sum_{i=1}^{n} X_i$. Alors

$$\Pr(X \geqslant n/2) \leqslant \left(\frac{1+p}{\sqrt{2}}\right)^n.$$

Démonstration On pose $Y = 2^X$: on cherche donc à majorer $\Pr(Y \geqslant 2^{n/2})$. Par l'inégalité de Markov,

$$\Pr(Y \geqslant 2^{n/2}) \leqslant E(Y)2^{-n/2}.$$

Or $Y = \prod_{i=1}^{n} 2^{X_i}$ et les variables aléatoires 2^{X_i} sont indépendantes puisque les X_i le sont. Donc par le lemme A-H, $E(Y) = \prod_{i=1}^{n} E(2^{X_i})$. Puisque 2^{X_i} vaut 1 avec probabilité $(1-p)$ et 2 avec probabilité p,

$$E(2^{X_i}) = (1-p) + 2p = 1 + p.$$

Ainsi, $E(Y) = (1+p)^n$. On en déduit que

$$\Pr(Y \geqslant 2^{n/2}) \leqslant E(Y)2^{-n/2} = (\frac{1+p}{\sqrt{2}})^n.$$

□

Avec un peu plus de travail on obtient la version plus générale suivante qui donne un résultat non trivial pour tout $p \in [0, 1/2[$ (contrairement à la version faible qui n'est plus intéressante dès que $p \geqslant \sqrt{2} - 1$), et dont l'énoncé a été adapté à notre propos.

A-N **Proposition (bornes de Chernoff)**

Soit $p \in [0, 1/2[$ et $X_1, \ldots, X_n$ des variables aléatoires indépendantes telles que

$$X_i = \begin{cases} 1 & \text{avec probabilité } p \\ 0 & \text{avec probabilité } (1-p). \end{cases}$$

Soit $X = \sum_{i=1}^{n} X_i$. Alors pour $\epsilon = 1/2 - p$ on a :

$$\Pr(X \geqslant n/2) \leqslant 2^{-(\epsilon^2/2)n}.$$

A.2 Arithmétique

Nous donnerons sans démonstration deux théorèmes d'arithmétique, dont le premier n'est pas facile à montrer. Il donne une estimation asymptotique de la densité des nombres premiers.

A-O **Théorème (des nombres premiers)**

Si x est un réel positif, on note $\pi(x)$ le nombre de nombres premiers $\leqslant x$. Alors

$$\pi(x) \sim_{x\to+\infty} \frac{x}{\ln(x)}.$$

Le second concerne la formule de Stirling, qui est bien connue dans sa version asymptotique ; ici nous donnons un encadrement de la factorielle. Cela nous permettra de majorer ci-dessous des coefficients binomiaux.

A-P **Théorème (formule de Stirling)**

Pour tout entier $n \geqslant 1$,

$$\left(\frac{n}{e}\right)^n \sqrt{2\pi n} \quad \leqslant \quad n! \quad \leqslant \quad \left(\frac{n}{e}\right)^n e\sqrt{n}.$$

Par simple calcul, on en déduit le corollaire suivant.

A-Q **Corollaire**

Pour tout entier pair $n \geqslant 2$,

$$\binom{n}{n/2} \leqslant \frac{e2^n}{\pi\sqrt{n}}.$$

Démonstration Le théorème A-P, donne la majoration suivante :

$$\binom{n}{n/2} = \frac{n!}{(n/2)!^2} \leqslant \frac{e\sqrt{n}(n/e)^n}{\pi n(n/2e)^n} = \frac{e2^n}{\pi\sqrt{n}}.$$

□

B Exercices supplémentaires

Outre les exercices contenus dans les différents chapitres, nous donnons ici quelques exercices et problèmes supplémentaires.

Padding et théorème de hiérarchie

✎ **B-A** **Exercice**

Soit $c > 0$ une constante.

1. Montrer que si $\mathsf{DSPACE}(n^c) \subseteq \mathsf{NP}$, alors $\mathsf{PSPACE} \subseteq \mathsf{NP}$.
2. En déduire que $\mathsf{DSPACE}(n^c) \neq \mathsf{NP}$.
3. Raisonner de même pour montrer que $\mathsf{DTIME}(2^{n^c}) \neq \mathsf{NP}$.

Circuits, oracles, padding

✎ **B-B** **Exercice**

On rappelle que $\mathsf{EXPSPACE} = \mathsf{DSPACE}(2^{n^{O(1)}})$.

1. Montrer que $\mathsf{EXP} = \mathsf{EXPSPACE} \iff \mathsf{PSPACE} \cap \mathsf{P/poly} = \mathsf{P}$.
   Indication : utiliser l'exercice 7-V avec la proposition 7-U pour se ramener à un langage unaire de PSPACE.
2. En déduire que $\mathsf{P} \neq \mathsf{RP} \implies \mathsf{EXP} \neq \mathsf{EXPSPACE}$.

Problèmes « succincts » et complétude pour le temps exponentiel

✎ **B-C** **Exercice**

Un circuit booléen C peut fournir le code d'un autre circuit booléen D comme suit : sur l'entrée i en binaire, $C(i)$ est égal au i-ème bit du code de D. On peut ainsi représenter des circuits D de taille exponentielle grâce à un circuit C de taille polynomiale. On dit alors que C représente succinctement le circuit D.

Soit $\texttt{VALEUR CIRCUIT}_{\text{succ.}}$ le problème suivant :

- *entrée :* un circuit C représentant succinctement un circuit D sans variables ;
- *question :* la valeur de D est-elle 1 ?

Montrer que $\texttt{VALEUR CIRCUIT}_{\text{succ.}}$ est EXP-complet.

Soit $\texttt{SAT-CIRCUIT}_{\text{succ.}}$ le problème suivant :

- *entrée :* un circuit C représentant succinctement un circuit $D(x_1,\dots,x_N)$;
- *question :* existe-t-il une affectation $(a_1,\dots,a_N) \in \{0,1\}^N$ des variables $(x_1,\dots,x_N)$ telle que $D(a_1,\dots,a_N)=1$?

Montrer que $\texttt{SAT-CIRCUIT}_{\text{succ.}}$ est NEXP-complet.

La classe ZPP

✎ **B-D** **Exercice**

La classe ZPP (pour *Zero-error Probabilistic Polynomial time*) est définie comme l'ensemble des langages A pour lesquels il existe un polynôme $p(n)$ et une machine déterministe polynomiale M à trois états terminaux q_a, q_r et $q_?$, vérifiant pour tout $x \in \Sigma^\star$:

- si $x \in A$ alors $\Pr_{r\in\{0,1\}^{p(|x|)}}(M(x,r) \to q_r) = 0$ et $\Pr_{r\in\{0,1\}^{p(|x|)}}(M(x,r) \to q_?) \leqslant 1/2$;
- si $x \notin A$ alors $\Pr_{r\in\{0,1\}^{p(|x|)}}(M(x,r) \to q_a) = 0$ et $\Pr_{r\in\{0,1\}^{p(|x|)}}(M(x,r) \to q_?) \leqslant 1/2$.

En d'autres termes, la machine M ne se trompe jamais mais peut ne pas répondre (état $q_?$) avec probabilité $\leqslant 1/2$.

1. Montrer que $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$
2. Montrer qu'un langage A est dans ZPP ssi il est reconnu par une machine probabiliste qui répond avec probabilité 1, ne se trompe jamais et dont le temps d'exécution est polynomial *en moyenne*.

Oracles aléatoires et RP

✎ **B-E** **Exercice**

Lorsqu'on met chaque mot de $\{0,1\}^*$ avec probabilité $1/2$ dans un langage A, on obtient un langage aléatoire dont on se servira en oracle.

Montrer qu'avec probabilité 1 sur le choix de l'oracle aléatoire A, $\mathsf{P}^A = \mathsf{RP}^A$.

Indication : soit M une machine probabiliste avec oracle A pour un langage RP^A. Dans P^A, interroger l'oracle sur des mots non demandés par $M(x)$ afin d'obtenir des « bits aléatoires » pour pouvoir simuler $M(x)$.

NP-complétude et langages unaires

✎ **B-F** **Exercice**

Le but de cet exercice est de montrer une version faible du théorème 3-AQ de Mahaney, due à Berman [Ber78] :

s'il existe un langage unaire A NP*-difficile, alors* $\mathsf{P} = \mathsf{NP}$.

On rappelle qu'un langage A est dit unaire si $A \subseteq 1^*$ (tous les mots de A ne contiennent que le symbole 1).

Supposons A unaire et NP-difficile : nous allons montrer que $\mathtt{SAT} \in \mathsf{P}$. Puisque A est NP-difficile, il existe une réduction polynomiale f de SAT à A.

L'idée est d'utiliser l'autoréductibilité de SAT et de maintenir grâce à f un ensemble de taille polynomiale de formules.

Soit $\varphi(x_1,\dots,x_n)$ une instance de SAT : nous devons décider en temps polynomial si $\varphi \in \mathtt{SAT}$.

1. Que dire de φ si $f(\varphi) \notin 1^*$?
2. Pour une formule $\psi(x_1,\dots,x_n)$, on désigne par ψ_0 la formule $\psi_0(x_2,\dots,x_n) = \psi(0,x_2,\dots,x_n)$ et par ψ_1 la formule $\psi_1(x_2,\dots,x_n) = \psi(1,x_2,\dots,x_n)$.
   Montrer que $\varphi \in \mathtt{SAT}$ ssi [$\varphi_0 \in \mathtt{SAT}$ ou $\varphi_1 \in \mathtt{SAT}$] (autoréductibilité).

   Pour simplifier, pour toute formule ψ on suppose que la taille $|\psi_0|$ (respectivement $|\psi_1|$) du codage de ψ_0 (resp. ψ_1) est égale à la taille $|\psi|$ du codage de ψ.
3. Tout au long de l'algorithme pour SAT qu'on cherche à décrire, on maintiendra un ensemble de formules booléennes. On appellera Φ_i cet ensemble à l'étape $i \leqslant n$. Au départ, on a $\Phi_0 = \{\varphi\}$.
   On définit alors Φ_{i+1} par la procédure suivante :

- $\Phi_{i+1} \leftarrow \emptyset$
- $I \leftarrow \emptyset$
- Pour tout $\psi \in \Phi_i$ faire
  - $u \leftarrow f(\psi_0)$
  - Si $u \in 1^\star$ et $u \notin I$ alors
    - $I \leftarrow I \cup \{u\}$
    - $\Phi_{i+1} \leftarrow \Phi_{i+1} \cup \{\psi_0\}$
  - $u \leftarrow f(\psi_1)$
  - Si $u \in 1^\star$ et $u \notin I$ alors
    - $I \leftarrow I \cup \{u\}$
    - $\Phi_{i+1} \leftarrow \Phi_{i+1} \cup \{\psi_1\}$
- Renvoyer Φ_{i+1}

Remarquer qu'on élimine une variable à chaque fois : l'ensemble Φ_i contient des formules à $n - i$ variables. Il y aura ainsi n étapes.
Pour tout i, montrer que $\varphi \in \mathtt{SAT}$ si et seulement s'il existe $\psi \in \Phi_i$ satisfaisable.

4. Montrer qu'il existe un polynôme $p(n)$ tel que pour tout i, $|\Phi_i| \leqslant p(|\varphi|)$ (ce polynôme dépend de la réduction f).
5. Conclure que $\mathtt{SAT} \in \mathsf{P}$.

Langages P-sélectifs

✎ **B-G** **Exercice**

Cet exercice est adapté du chapitre 3 du livre [HO02].

Un langage L est dit P-*sélectif* ($L \in \mathsf{P\text{-}sel}$) s'il existe une fonction $f : \{0,1\}^\star \times \{0,1\}^\star \to \{0,1\}^\star$ calculable en temps polynomial telle que $\forall x, y \in \{0,1\}^\star$:

- $f(x,y) \in \{x,y\}$, et
- si $x \in L$ ou $y \in L$ alors $f(x,y) \in L$.

On appelle f *fonction de sélection* pour L.

1. Montrer que $\mathsf{P} \subseteq \mathsf{P\text{-}sel}$.
2. Montrer que $\mathsf{P\text{-}sel}$ est clos par complémentaire.
3. Montrer que s'il existe un langage L NP-difficile dans $\mathsf{P\text{-}sel}$, alors $\mathsf{P} = \mathsf{NP}$.

Indication : en s'aidant de L, on pourra donner un algorithme polynomial pour SAT *qui utilise l'autoréductibilité de* SAT *:* $\varphi(x_1,\ldots,x_n) \in \mathtt{SAT} \iff [\varphi(0,x_2,\ldots,x_n) \in \mathtt{SAT}$ *ou* $\varphi(1,x_2\ldots,x_n) \in \mathtt{SAT}]$.

4. Soit $r \in [0,1]$ un réel. Soit L_r l'ensemble des mots $b = b_1 \ldots b_n \in \{0,1\}^*$ tels que le réel $0,b_1 \ldots b_n \in [0,1]$ soit majoré par r. Montrer que $L_r \in$ P-sel.
5. En déduire qu'il existe un langage non récursif dans P-sel.
6. On appelle *tournoi* tout graphe complet dont on a orienté les arêtes.
   Montrer le lemme suivant : si $G = (S,A)$ est un tournoi à k sommets, il existe un sous-ensemble des sommets $H \subseteq S$ de cardinal au plus $\lfloor \log(k+1) \rfloor$ tel que pour tout $s \in S \setminus H$ il existe $h \in H$ tel que $(s,h) \in A$.
7. Soit $L \in$ P-sel. Montrer qu'il existe une fonction de sélection f pour L telle que $\forall x, y, f(x,y) = f(y,x)$.
8. Soit $L \in$ P-sel et soit une fonction de sélection f pour L telle que pour tout x et y, $f(x,y) = f(y,x)$.
   Montrer qu'il existe un sous-ensemble H_n de $L^{=n}$ de cardinal au plus $\max(1,n)$ tel que s appartient à $L^{=n}$ si et seulement s'il existe un mot h de H_n tel que $f(s,h) = s$.
9. En déduire que P-sel $\subseteq$ P/n^2.
10. Montrer que si G est un tournoi alors il existe un sommet s tel que tout sommet soit accessible à partir de s par un chemin orienté de longueur au plus deux.
11. En déduire que P-sel $\subseteq$ NP/$(n+1)$.

C Classes de complexité rencontrées dans ce livre

Nous donnons ici la liste de toutes les classes de complexité apparaissant dans cet ouvrage, avec une brève description pour chacune.

En outre, les figures qui suivent décrivent les principales classes. La figure C.1 représente les classes déterministes et non déterministes. La figure C.2 les classes probabilistes, avec quelques autres points de repère en gris. De même pour la figure C.3 qui représente les classes non uniformes.

Liste des classes de complexité

Voici une liste de ces classes par ordre alphabétique et l'endroit où elles sont définies :

- ACC^0, langages reconnus par une famille de circuits de taille polynomiale et de profondeur constante avec portes modulo : définition 11-R.
- AC^0, langages reconnus par une famille de circuits de taille polynomiale et de profondeur constante : définition 5-AK.
- AM, langages reconnus par un protocole Arthur-Merlin : définition 10-S.
- BPP, langages reconnus en temps polynomial par un algorithme probabiliste avec faible probabilité d'erreur : définition 6-B.
- $\mathsf{BPTIME}(f(n))$, langages reconnus en temps $O(f(n))$ par un algorithme probabiliste avec faible probabilité d'erreur : cf. section 6.2.4.
- $\mathscr{C}/f(n)$, classe de complexité $\mathscr{C}$ avec conseil de taille $f(n)$: définition 5-A.
- coRP, complémentaire d'un langage RP : remarque 6-K.
- coNL, complémentaire d'un langage NL : proposition 4-AV.
- coNP, complémentaire d'un langage NP : proposition 2-AY.

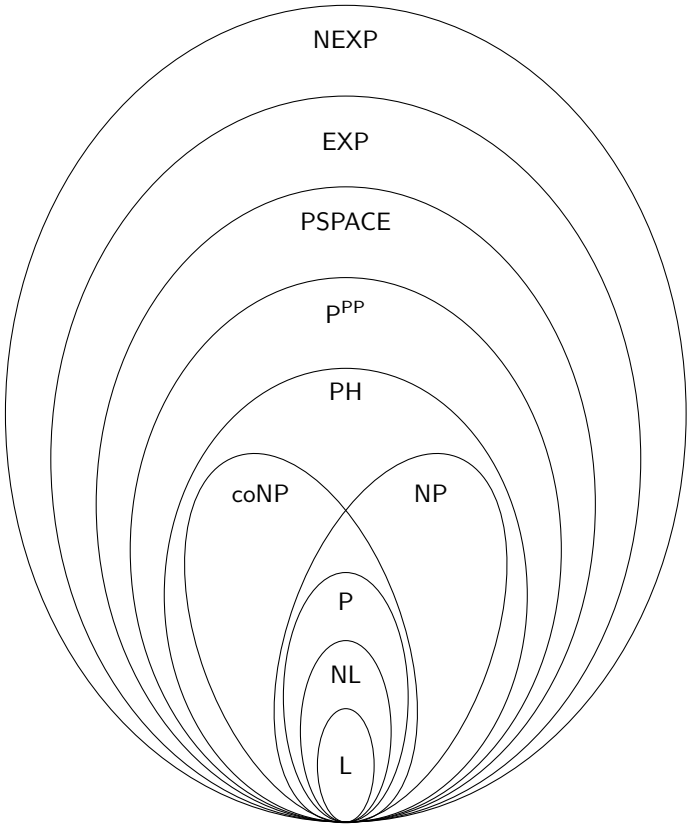


FIGURE C.1 – Principales classes de complexité déterministes et non déterministes rencontrées dans ce livre.

– ♯P, fonctions comptant le nombre de solutions d'un problème NP : définition 9-A.
– DSPACE($f(n)$), langages reconnus par un algorithme déterministe fonctionnant en espace $O(f(n))$: définition 4-A.
– DTIME($f(n)$), langages reconnus par un algorithme déterministe fonctionnant en temps $O(f(n))$: définition 2-A.
– E, langages reconnus par un algorithme déterministe fonctionnant en temps exponentiel $2^{O(n)}$: définition 2-O.
– EXP, langages reconnus par un algorithme déterministe fonctionnant en temps exponentiel $2^{n^{O(1)}}$: définition 2-O.
– EXPSPACE, langages reconnus par un algorithme déterministe fonctionnant en espace exponentiel $2^{n^{O(1)}}$: exercice 11-G.
– FewP, ensemble des langages reconnus par une machine non déterministe ayant au

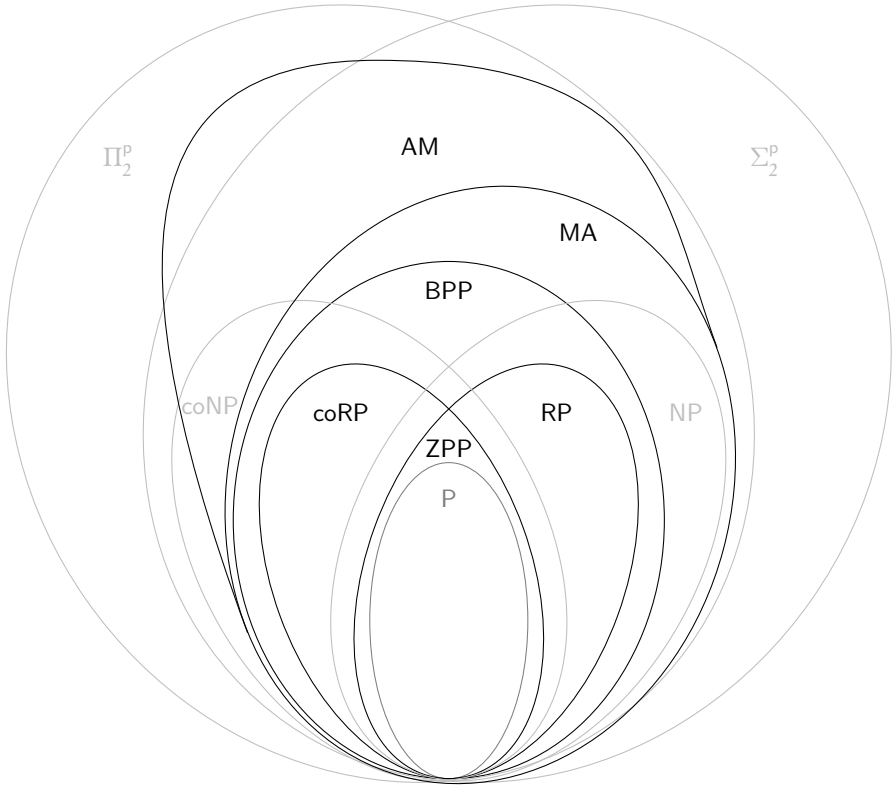


FIGURE C.2 – Principales classes de complexité probabilistes rencontrées dans ce livre.

plus un nombre polynomial de chemins acceptants : exercice 9-AE.

– GapP, clôture de ♯P par soustraction : définition 9-F.

– IP, langages reconnus par protocoles interactifs ayant un nombre polynomial de tours : définition 10-A.

– L, langages reconnus par un algorithme déterministe fonctionnant en espace logarithmique $O(\log n)$: définition 4-E.

– MA, langages reconnus par protocoles Merlin-Arthur : définition 10-S.

– $\mathsf{MA}_{\mathsf{EXP}}$, langages reconnus par protocoles de type Merlin-Arthur avec vérificateur exponentiel : exercice 11-K.

– $\mathsf{Mod}_k\mathsf{P}$, classe de comptage modulo k en temps polynomial : définition 9-O.

– NC^1, langages reconnus par circuits de taille polynomiale et profondeur logarith-

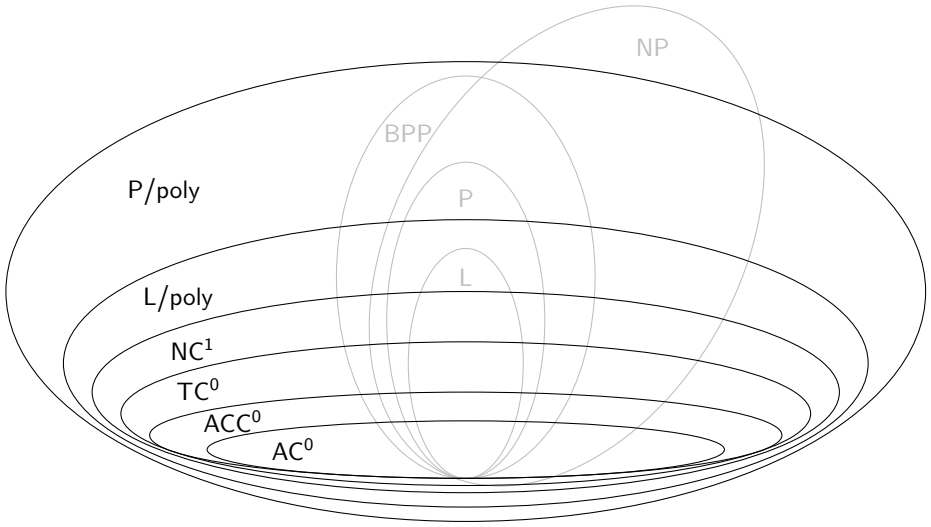


FIGURE C.3 – Principales classes de complexité non uniformes rencontrées dans ce livre.

mique $O(\log n)$: définition 5-AH.

– NEXP, langages reconnus par un algorithme non déterministe fonctionnant en temps exponentiel $2^{n^{O(1)}}$: définition 2-AK.

– NL, langages reconnus par un algorithme non déterministe fonctionnant en espace logarithmique $O(\log n)$: définition 4-Z.

– NP, langages reconnus par un algorithme non déterministe fonctionnant en temps polynomial $n^{O(1)}$: définition 2-AK.

– NSPACE$(f(n))$, langages reconnus par un algorithme non déterministe fonctionnant en espace $O(f(n))$: définition 4-P.

– NTIME$(f(n))$, langages reconnus par un algorithme non déterministe fonctionnant en temps $O(f(n))$: définition 2-AE.

– P, langages reconnus par un algorithme déterministe fonctionnant en temps polynomial $n^{O(1)}$: définition 2-O.

– $\oplus$P, égal à Mod_2P : définition 9-O.

– PCP$(f(n), g(n))$, langages reconnus par un protocole PCP utilisant $f(n)$ bits aléatoires et lisant $g(n)$ bits de la preuve : définition 10-AI.

– PH, hiérarchie polynomiale, union des Σ_k^{p} : définition 8-A.

– Π_k^{p}, k-ème niveau de la hiérarchie polynomiale, langages définis par une alternance de k quantificateurs commençant par $\forall$: définition 8-A.

– PP, langages correspondant aux instances d'un problème NP qui ont une majorité

de solutions : définition 9-J.

- P-sel, langages P-sélectifs : exercice B-G.
- PSPACE, langages reconnus par un algorithme déterministe fonctionnant en espace polynomial $n^{O(1)}$: définition 4-Z.
- P/poly, langages reconnus par des circuits de taille polynomiale : définition 5-J et proposition 5-AC.
- RP, langages reconnus en temps polynomial par un algorithme probabiliste avec erreur d'un seul côté : définition 6-C.
- Σ_k^{P}, k-ème niveau de la hiérarchie polynomiale, langages définis par une alternance de k quantificateurs commençant par $\exists$: définition 8-A.
- SIZE$(f(n))$, langages reconnus par des circuits de taille $O(f(n))$: cf. section 11.1.
- TC^0, langages reconnus par une famille de circuits de taille polynomiale et de profondeur constante avec portes de majorité : définition 5-AK.
- TISP$(f(n), g(n))$, langages reconnus par un algorithme déterministe fonctionnant en temps $O(f(n))$ et en espace $O(g(n))$: définition 8-R.
- ZPP, langages reconnus sans erreur par un algorithme probabiliste fonctionnant en temps moyen polynomial : exercice B-D.

Classes déterministes et non déterministes

Concernant les classes déterministes et non déterministes, voici un petit résumé des inclusions principales vues dans cet ouvrage :

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$$

et $\mathsf{P} \subseteq \mathsf{coNP} \subseteq \mathsf{PH}$. Par ailleurs, on sait séparer PSPACE de NL, EXP de P et NEXP de NP.

Classes probabilistes

Concernant les classes probabilistes, voici un petit résumé des inclusions que nous avons vues :

$$\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{MA} \subseteq \mathsf{AM} \subseteq \Pi_2^{\mathsf{P}}$$
$$\mathsf{P} \subseteq \mathsf{coRP} \subseteq \mathsf{BPP} \subseteq \Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$$

et $\mathsf{RP} \subseteq \mathsf{NP}$.

Classes non uniformes

Concernant les classes non uniformes, voici un petit résumé des inclusions que nous avons vues :

$$\mathsf{AC}^0 \subseteq \mathsf{ACC}^0 \subseteq \mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L/poly} \subseteq \mathsf{P/poly}$$

et $\mathsf{BPP} \subset \mathsf{P/poly}$. Le récent résultat de Williams [Wil11] montre que $\mathsf{NEXP} \not\subseteq \mathsf{ACC}^0$.

D Indications pour l'enseignant

Quelques commentaires sur le contenu de cet ouvrage pourront aider l'enseignant qui souhaiterait utiliser ce livre. Il couvre un large spectre de la complexité algorithmique et ne peut guère être traité en un seul cours car il faudrait au moins une centaine d'heures pour tout enseigner. Nous donnons donc des suggestions de cours ci-dessous.

Cours introductif

Les cours introductifs à la complexité, en L3 ou M1 par exemple, sont souvent couplés à une introduction à la calculabilité, ce qui permet de mettre en commun une bonne partie du chapitre 1 sur les machines de Turing. En évitant les détails trop formels pour se consacrer à l'intuition du modèle de calcul, ce chapitre peut être traité en 3 heures environ si l'on se contente de la machine universelle avec ralentissement quadratique (c'est-à-dire qu'on ne montre pas la machine universelle avec ralentissement logarithmique du théorème 1-S).

Il reste donc en général de 8 à 12 heures à consacrer à la complexité. Un déroulement logique est certainement de traiter le chapitre 2 (en laissant éventuellement le théorème de la lacune 2-L de côté) : si on doit réduire le temps consacré à ce chapitre, en plus du théorème de la lacune on supprimera la machine universelle non déterministe (proposition 2-AB) et le théorème 2-AI de hiérarchie non déterministe, mais il semble difficile d'y passer moins de 3 heures.

Puis le chapitre 3 semble incontournable pour des étudiants en informatique. On n'échappera pas au théorème 3-V de Cook-Levin de NP-complétude de SAT qui, selon les détails donnés, peut prendre de 30 minutes à une heure, et à quelques exemples de réductions. On pourra sans crainte supprimer le théorème 3-AQ de Mahaney. Mais là encore, même en sabrant le théorème 3-AK de Ladner, difficile de passer moins de 2 à 3 heures sur ce chapitre.

On aura ensuite le choix entre le chapitre 4 sur la complexité en espace, ou le chapitre 6 sur les algorithmes probabilistes. Le « canal historique » optera pour l'espace mais les algorithmes probabilistes sont de plus en plus importants et constituent donc également un choix tout à fait pertinent. L'idéal étant bien sûr de traiter les deux. Pour le chapitre 4, on pourra éviter le théorème 4-AZ d'Immerman-Szelepcsényi, et donner seulement les idées de la complétude de QBF (théorème 4-AC) : à ce prix, on peut réussir à boucler le chapitre en 2 à 3 heures. Le chapitre 6, quant à lui, est assez court, surtout sans traiter le lien avec les circuits booléens (qui n'auront pas été définis), et on peut envisager de l'enseigner en 2 heures. Enfin, une petite ouverture en guise de conclusion sur les sujets plus avancés est en général bienvenue.

Si on dispose d'un cours complet pour enseigner la complexité, soit environ 25 heures, nous recommandons de traiter les mêmes sujets mais plus en profondeur : plus de détails, éventuellement un ou deux théorèmes qui auraient été sautés, etc. Il restera certainement 5 ou 6 heures pour lesquelles on pourra choisir de traiter le chapitre 5 sur les circuits (sans les circuits arithmétiques, section 5.6) avec quelques bornes inférieures du chapitre 11, ou alors les chapitres 7 sur les oracles (sans la proposition 7-AH de Kozen et sans le théorème 7-AD sur la non-relativisation de $\mathsf{EXP} = \mathsf{NP}$) et 8 sur la hiérarchie polynomiale (sans le théorème 8-S sur TISP).

Cours avancé

Pour des étudiants de M1 (voire M2) d'informatique ou de mathématique ayant déjà « subi » un cours introductif, on pourra choisir d'enseigner le matériel des chapitres 1 à 8 qui n'aura pas été traité dans le cours introductif. Par exemple : théorème 2-AI de hiérarchie non déterministe en temps, théorèmes de Ladner 3-AK et de Mahaney 3-AQ, théorème 4-AZ d'Immerman-Szelepcsényi, non-uniformité (chapitre 5), algorithmes probabilistes si ce n'est déjà fait (chapitre 6) oracles et hiérarchie polynomiale (chapitres 7 et 8). Avec les rappels qui s'imposent, on approche ainsi les 20 heures selon le niveau de détails donnés. Dans le temps qui reste, on pourra par exemple faire la preuve de $\mathsf{IP} = \mathsf{PSPACE}$ (théorème 10-G), ou quelques bornes inférieures du chapitre 11.

Cours spécialisés

Les quatre derniers chapitres donnent matière à des cours spécialisés au niveau M2 par exemple, pour de bons étudiants en informatique théorique ou en mathématique. Plusieurs thèmes pourront être traités.

On pourra par exemple donner un cours cohérent sur le comptage et le calcul de polynômes : on traitera alors tout le chapitre 9 ($\sharp\mathsf{P}$, théorème 9-AI de Toda, théorème 9-AP de complétude du permanent notamment). Puis on pourra passer au cadre algébrique du calcul de polynômes (où le permanent est aussi complet pour la classe algébrique VNP) et enseigner la partie du chapitre 5 consacrée aux circuits arithmétiques (notamment l'élimination des divisions de Strassen, proposition 5-AV), le problème d'identité de polynômes (section 6.3), ainsi que les bornes inférieures sur les circuits arithmétiques données à la

section 11.3. Si le temps le permet, on pourra compléter le cours avec une introduction à VP et VNP grâce au livre de Bürgisser [Bür00].

Un autre thème possible concerne les protocoles interactifs, avec l'intégralité du chapitre 10 et les bornes inférieures du chapitre 11 qui utilisent ces notions (par exemple le théorème 11-F ou le corollaire 11-N). Avec un peu de travail, on pourra compléter l'exposé par une preuve du théorème PCP et d'autres applications à l'inapproximabilité de problèmes d'optimisation NP-complets (cf. section 10.3), grâce au livre d'Arora et Barak [AB09] par exemple (pour les idées de la preuve, on pourra aussi se référer à celui de Goldreich [Gol08]). Au total, il y a largement de quoi remplir un cours complet.

On peut encore se consacrer aux bornes inférieures sur les circuits, en traitant le chapitre 11 en entier et en le complétant avec la preuve du théorème 11-Q sur les circuits monotones pour CLIQUE, que l'on trouvera par exemple dans le livre d'Hemaspaandra et Ogihara [HO02] ou celui d'Arora et Barak [AB09]. Avec les rappels nécessaires sur les circuits (booléens et arithmétiques, chapitre 5), sur la hiérarchie polynomiale (chapitre 8) et sur les protocoles interactifs (chapitre 10), on y passera de l'ordre de 15 heures. Dans le temps qui reste, avec un peu de travail on pourra aborder des résultats plus récents comme la borne inférieure de Williams [Wil11] ($\mathsf{NEXP} \not\subset \mathsf{ACC}^0$), ou d'autres bornes inférieures sur les circuits arithmétiques (circuits monotones, circuits multilinéaires, etc., voir l'excellent survol de Shpilka et Yehudayoff [SY10]).

Enfin, on peut donner un cours sur la dérandomisation et ses liens avec les bornes inférieures non uniformes. Un contenu cohérent serait de faire quelques rappels sur les algorithmes probabilistes (chapitre 6) et sur le problème d'identité de polynômes (section 6.3), de donner quelques bornes inférieures non uniformes du chapitre 11, de donner quelques exemples de dérandomisation grâce au livre d'Alon et Spencer [AS04] (cf. section 12.1), et enfin de traiter le chapitre 12 en entier. Avec les rappels nécessaires sur les circuits et les algorithmes probabilistes notamment, il y a du matériel pour remplir un cours de 25 heures. Pour le théorème 12-BA de Kabanets et Impagliazzo, à défaut de plus de temps on sera malheureusement obligé d'admettre quelques résultats (le théorème 9-AI de Toda par exemple).

Bibliographie

[AB09] Sanjeev Arora et Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009, p. I–XXIV, 1–579 (cf. p. i, xiii, 283, 297, 375).

[AB87] Noga Alon et Ravi B. Boppana. « The monotone circuit complexity of Boolean functions ». In : *Combinatorica* 7.1 (1987), p. 1–22 (cf. p. 297).

[Adl78] Leonard M. Adleman. « Two Theorems on Random Polynomial Time ». In : *FOCS*. 1978, p. 75–83 (cf. p. 163).

[AKS04] Manindra Agrawal, Neeraj Kayal et Nitin Saxena. « PRIMES is in P ». In : *Annals of mathematics* 160.2 (2004), p. 781–793 (cf. p. 41, 163, 316).

[Ale+79] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász et Charles Rackoff. « Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems ». In : *FOCS*. 1979, p. 218–223 (cf. p. 317).

[Aro+92] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan et Mario Szegedy. « Proof Verification and Hardness of Approximation Problems ». In : *FOCS*. 1992, p. 14–23 (cf. p. 283, 286).

[AS04] Noga Alon et Joel H. Spencer. *The Probabilistic Method*. Wiley, 2004 (cf. p. 317, 375).

[AS92] Sanjeev Arora et Shmuel Safra. « Probabilistic Checking of Proofs ; A New Characterization of NP ». In : *FOCS*. 1992, p. 2–13 (cf. p. 283).

[AW08] Scott Aaronson et Avi Wigderson. « Algebrization : a new barrier in complexity theory ». In : *STOC*. 2008, p. 731–740 (cf. p. 266).

[Bab85] László Babai. « Trading Group Theory for Randomness ». In : *STOC*. 1985, p. 421–429 (cf. p. 250).

[BBS09] László Babai, Robert Beals et Ákos Seress. « Polynomial-time theory of matrix groups ». In : *STOC*. 2009, p. 55–64 (cf. p. 167).

[BDG88] José L. Balcázar, Josep Díaz et Joaquim Gabarró. *Structural complexity 1*. T. 11. Springer, 1988 (cf. p. xii).

[Ber67] Elwyn R. Berlekamp. « Factoring Polynomials Over Finite Fields ». In : *Bell System Technical Journal* 46 (1967), 1853–1859 (cf. p. 334).

[Ber78] Piotr Berman. « Relationship Between Density and Deterministic Complexity of NP-Complete Languages ». In : *ICALP*. T. 62. 1978, p. 63–71 (cf. p. 363).

[BFL91] László Babai, Lance Fortnow et Carsten Lund. « Non-Deterministic Exponential Time has Two-Prover Interactive Protocols ». In : *Computational Complexity* 1 (1991), p. 3–40 (cf. p. 294, 296).

[BFT98] Harry Buhrman, Lance Fortnow et Thomas Thierauf. « Nonrelativizing Separations ». In : *IEEE Conference on Computational Complexity*. 1998, p. 8–12 (cf. p. 292, 293).

[BGS75] Theodore P. Baker, John Gill et Robert Solovay. « Relativizatons of the P =?NP Question ». In : *SIAM J. Comput.* 4.4 (1975), p. 431–442 (cf. p. 183).

[BH77] Leonard Berman et Juris Hartmanis. « On Isomorphisms and Density of NP and Other Complete Sets ». In : *SIAM J. Comput.* 6.2 (1977), p. 305–322 (cf. p. 180).

[BHZ87] Ravi B. Boppana, Johan Håstad et Stathis Zachos. « Does coNP Have Short Interactive Proofs ? » In : *Inf. Process. Lett.* 25.2 (1987), p. 127–132 (cf. p. 281).

[BM88] László Babai et Shlomo Moran. « Arthur-Merlin Games : A Randomized Proof System, and a Hierarchy of Complexity Classes ». In : *J. Comput. Syst. Sci.* 36.2 (1988), p. 254–276 (cf. p. 269).

[Bor72] Allan Borodin. « Computational Complexity and the Existence of Complexity Gaps ». In : *J. ACM* 19.1 (1972), p. 158–174 (cf. p. 38).

[BRS95] Richard Beigel, Nick Reingold et Daniel A. Spielman. « PP Is Closed under Intersection ». In : *J. Comput. Syst. Sci.* 50.2 (1995), p. 191–202 (cf. p. 225).

[BS83] Walter Baur et Volker Strassen. « The Complexity of Partial Derivatives ». In : *Theor. Comput. Sci.* 22 (1983), p. 317–330 (cf. p. 304, 310).

[Bür00] Peter Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*. Springer, 2000 (cf. p. 375).

[Car08] Olivier Carton. *Langages formels, calculabilité et complexité*. 240 pages. Vuibert, 2008 (cf. p. xiii).

[Chu36] Alonzo Church. « A Note on the Entscheidungsproblem ». In : *J. Symb. Log.* 1.1 (1936), p. 40–41 (cf. p. x, 1).

[Cob65] Alan Cobham. « The Intrinsic Computational Difficulty of Functions ». In : *Logic, Methodology and Philosophy of Science, proceedings of the second International Congress, held in Jerusalem, 1964*. 1965 (cf. p. x, 39).

[Coo04] S. Barry Cooper. *Computability Theory*. Chapman & Hall/CRC Mathematics, 2004 (cf. p. xiii).

[Coo71] Stephen A. Cook. « The Complexity of Theorem-Proving Procedures ». In : *STOC*. 1971, p. 151–158 (cf. p. xi, 70).

[Coo72] Stephen A. Cook. « A Hierarchy for Nondeterministic Time Complexity ». In : *STOC*. 1972, p. 187–192 (cf. p. 52).

[Cor+10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. *Algorithmique*. 3ème édition. Dunod, 2010 (cf. p. 41).

[Deh93] Patrick Dehornoy. *Complexité et décidabilité*. Springer-Verlag, 1993 (cf. p. xiii).

[Din06] Irit Dinur. « The PCP theorem by gap amplification ». In : *STOC*. 2006, p. 241–250 (cf. p. 283).

[Edm65] Jack Edmonds. « Paths, Trees, and Flowers ». In : *Canad. J. Math.* 17 (1965), p. 449–467 (cf. p. x, 39–41, 239).

[FM00] Lance Fortnow et Dieter van Melkebeek. « Time-Space Tradeoffs for Nondeterministic Computation ». In : *IEEE Conference on Computational Complexity*. 2000, p. 2–13 (cf. p. 207).

[For97] Lance Fortnow. « Nondeterministic Polynomial Time versus Nondeterministic Logarithmic Space : Time-Space Tradeoffs for Satisfiability ». In : *IEEE Conference on Computational Complexity*. 1997, p. 52–60 (cf. p. 207).

[FS04] Lance Fortnow et Rahul Santhanam. « Hierarchy Theorems for Probabilistic Polynomial Time ». In : *FOCS*. 2004, p. 316–324 (cf. p. 167).

[FS88] Lance Fortnow et Michael Sipser. « Are There Interactive Protocols for coNP Languages ? » In : *Inf. Process. Lett.* 28.5 (1988), p. 249–251 (cf. p. 265).

[Für82] Martin Fürer. « The Tight Deterministic Time Hierarchy ». In : *STOC*. 1982, p. 8–16 (cf. p. 38).

[Gem+91] Peter Gemmell, Richard J. Lipton, Ronitt Rubinfeld, Madhu Sudan et Avi Wigderson. « Self-Testing/Correcting for Polynomials and for Approximate Functions ». In : *STOC*. 1991, p. 32–42 (cf. p. 335).

[Gil77] John T. Gill. « Computational Complexity of Probabilistic Turing Machines ». In : *SIAM J. Comput.* 6.4 (1977), p. 675–695 (cf. p. 156, 216).

[GJ79] Michael R. Garey et David S. Johnson. *Computers and Intractability : A Guide to the Theory of* NP*-Completeness*. W. H. Freeman & Co., 1979 (cf. p. 85).

[GL89] Oded Goldreich et Leonid A. Levin. « A Hard-Core Predicate for all One-Way Functions ». In : *STOC*. 1989, p. 25–32 (cf. p. 341).

[GMR85] Shafi Goldwasser, Silvio Micali et Charles Rackoff. « The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract) ». In : *STOC*. 1985, p. 291–304 (cf. p. 250).

[GMW87] Oded Goldreich, Silvio Micali et Avi Wigderson. « How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority ». In : *STOC*. 1987, p. 218–229 (cf. p. 253).

[Gol08] Oded Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008, p. I–XXIV, 1–606 (cf. p. xiii, 317, 375).

[GS86] Shafi Goldwasser et Michael Sipser. « Private Coins versus Public Coins in Interactive Proof Systems ». In : *STOC*. 1986, p. 59–68 (cf. p. 274).

[GS92] Peter Gemmell et Madhu Sudan. « Highly Resilient Correctors for Polynomials ». In : *Inf. Process. Lett.* 43.4 (1992), p. 169–174 (cf. p. 334).

[Hås97] Johan Håstad. « Some Optimal Inapproximability Results ». In : *STOC*. 1997, p. 1–10 (cf. p. 283).

[Hel84] Hans Heller. « On Relativized Polynomial and Exponential Computations ». In : *SIAM J. Comput.* 13.4 (1984), p. 717–725 (cf. p. 186).

[HO02] Lane A. Hemaspaandra et Mitsunori Ogihara. *The Complexity Theory Companion*. Springer, 2002 (cf. p. xii, 228, 364, 375).

[HS65] Juris Hartmanis et Richard E. Stearns. « On the computational complexity of algorithms ». In : *Transactions of the American Mathematical Society* 117 (1965), p. 285–306 (cf. p. x, 32, 33, 36).

[HS66] F. C. Hennie et Richard E. Stearns. « Two-Tape Simulation of Multitape Turing Machines ». In : *J. ACM* 13.4 (1966), p. 533–546 (cf. p. 21, 36).

[IKW01] Russell Impagliazzo, Valentine Kabanets et Avi Wigderson. « In Search of an Easy Witness : Exponential Time vs. Probabilistic Polynomial Time ». In : *IEEE Conference on Computational Complexity*. 2001, p. 2–12 (cf. p. 348).

[Imm88] Neil Immerman. « Nondeterministic Space is Closed Under Complementation ». In : *SIAM J. Comput.* 17.5 (1988), p. 935–938 (cf. p. 125).

[Imm99] Neil Immerman. *Descriptive Complexity*. Springer, 1999 (cf. p. xiii).

[IW97] Russell Impagliazzo et Avi Wigderson. « P = BPP if EXP Requires Exponential Circuits : Derandomizing the XOR Lemma ». In : *STOC*. 1997, p. 220–229 (cf. p. 344).

[JP02] Jean Jacod et Phillip Protter. *L'essentiel en théorie des probabilités*. Cassini, 2002 (cf. p. 353).

[Kan81] Ravi Kannan. « A Circuit-Size Lower Bound ». In : *FOCS*. 1981, p. 304–309 (cf. p. 288, 290).

[Kan83] Ravi Kannan. « Alternation and the Power of Nondeterminism ». In : *STOC*. 1983, p. 344–346 (cf. p. 207).

[Kar72] Richard M. Karp. « Reducibility Among Combinatorial Problems ». In : *Complexity of Computer Computations*. 1972, p. 85–103 (cf. p. xi, 70, 194).

[KI03] Valentine Kabanets et Russell Impagliazzo. « Derandomizing polynomial identity tests means proving circuit lower bounds ». In : *STOC*. 2003, p. 355–364 (cf. p. 351).

[KL82] Richard M. Karp et Richard Lipton. « Turing machines that take advice ». In : *Enseign. Math* 28.2 (1982), p. 191–209 (cf. p. 128, 141, 204, 295).

[Koz80] Dexter Kozen. « Indexings of Subrecursive Classes ». In : *Theor. Comput. Sci.* 11 (1980), p. 277–301 (cf. p. 190).

[KV87] Marek Karpinski et Rutger Verbeek. « On the Monte Carlo Space Constructible Functions and Seperation Results for Probabilistic Complexity Classes ». In : *Inf. Comput.* 75.2 (1987), p. 178–189 (cf. p. 160).

[KZ97] Howard J. Karloff et Uri Zwick. « A 7/8-Approximation Algorithm for MAX 3SAT ? » In : *FOCS*. 1997, p. 406–415 (cf. p. 286).

[Lad75a] Richard E. Ladner. « On the Structure of Polynomial Time Reducibility ». In : *J. ACM* 22.1 (1975), p. 155–171 (cf. p. 86).

[Lad75b] Richard E. Ladner. « The circuit value problem is log space complete for P ». In : *SIGACT News* 7.1 (1975), p. 18–20 (cf. p. 140).

[Lau83] Clemens Lautemann. « BPP and the Polynomial Hierarchy ». In : *Inf. Process. Lett.* 17.4 (1983), p. 215–217 (cf. p. 201).

[Lav08] Ivan Lavallé. *Complexité et algorithmique avancée : Une introduction*. Éditions Hermann, 2008 (cf. p. xiii).

[Lev73] Leonid Levin. « Universal sequential search problems ». In : *Problemy Peredachi Informatsii* 9.3 (1973), p. 115–116 (cf. p. xi, 70, 94).

[Lip75] Richard J. Lipton. « Polynomials with 0-1 Coefficients that Are Hard to Evaluate ». In : *FOCS*. 1975, p. 6–10 (cf. p. 304).

[LR96] Richard Lassaigne et Michel de Rougemont. *Logique et complexité*. Hermès, 1996 (cf. p. xiii).

[Mah82] Stephen R. Mahaney. « Sparse Complete Sets of NP : Solution of a Conjecture of Berman and Hartmanis ». In : *J. Comput. Syst. Sci.* 25.2 (1982), p. 130–143 (cf. p. 90).

[Mil76] Gary L. Miller. « Riemann's Hypothesis and Tests for Primality ». In : *J. Comput. Syst. Sci.* 13.3 (1976), p. 300–317 (cf. p. 316).

[Mor85] Jacques Morgenstern. « How to compute fast a function and all its derivatives : A variation on the theorem of Baur-Strassen ». In : *ACM SIGACT News* 16.4 (1985), p. 60–62 (cf. p. 305).

[MS72] Albert R. Meyer et Larry J. Stockmeyer. « The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space ». In : *SWAT (FOCS)*. 1972, p. 125–129 (cf. p. 193, 194).

[Mul54] David E. Muller. « Application of Boolean algebra to switching circuit design and to error detection ». In : *Electronic Computers, Transactions of the IRE Professional Group on* 3 (1954), p. 6–12 (cf. p. 330).

[Nep70] V. A. Nepomnjaščiĭ. « Rudimentary predicates and Turing computations ». In : *Dokl. Akad. Nauk SSSR* 195 (1970). En russe ; traduction anglaise dans *Soviet Mathematics Doklady*, Vol. 11, pp. 1642–1645, 1970., p. 282–284 (cf. p. 207).

[NW88] Noam Nisan et Avi Wigderson. « Hardness vs. Randomness (Extended Abstract) ». In : *FOCS*. 1988, p. 2–11 (cf. p. 323).

[Odi92] Piergiorgio Odifreddi. *Classical recursion theory, I and II*. Elsevier, 1992 (cf. p. xiii).

[OH93] Mitsunori Ogiwara et Lane A. Hemachandra. « A Complexity Theory for Feasible Closure Properties ». In : *J. Comput. Syst. Sci.* 46.3 (1993), p. 295–325 (cf. p. 227).

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994, p. I–XV, 1–523 (cf. p. xii).

[Rab80] Michael O. Rabin. « Probabilistic algorithm for testing primality ». In : *Journal of Number Theory* 12.1 (1980), p. 128–138 (cf. p. 316).

[Raz85] Alexander A. Razborov. « Lower bounds for the monotone complexity of some Boolean functions ». In : *Soviet Math. Dokl.* 31.2 (1985), p. 354–357 (cf. p. 297).

[Raz87] Alexander A. Razborov. « Lower bounds on the size of bounded depth circuits over a complete basis with logical addition ». In : *Math. notes of the Academy of Sciences of the USSR* 41.4 (1987), p. 333–338 (cf. p. 299).

[Rei05] Omer Reingold. « Undirected ST-connectivity in log-space ». In : *STOC*. 2005, p. 376–385 (cf. p. 317).

[Rey04] Jean-François Rey. *Calculabilité, complexité et approximation*. 363 pages. Vuibert, 2004 (cf. p. xiii).

[Sav70] Walter J. Savitch. « Relationships Between Nondeterministic and Deterministic Tape Complexities ». In : *J. Comput. Syst. Sci.* 4.2 (1970), p. 177–192 (cf. p. 118).

[Sch76] Arnold Schönhage. « An elementary proof for Strassen's degree bound ». In : *Theoretical Computer Science* 3.2 (1976), p. 267–272 (cf. p. 304).

[Sch78] Claus-Peter Schnorr. « Improved Lower Bounds on the Number of Multiplications/Divisions which are Necessary of Evaluate Polynomials ». In : *Theor. Comput. Sci.* 7 (1978), p. 251–261 (cf. p. 304, 310).

[Sch80] Jack T. Schwartz. « Fast Probabilistic Algorithms for Verification of Polynomial Identities ». In : *J. ACM* 27.4 (1980), p. 701–717 (cf. p. 331).

[SFM78] Joel I. Seiferas, Michael J. Fischer et Albert R. Meyer. « Separating Nondeterministic Time Complexity Classes ». In : *J. ACM* 25.1 (1978), p. 146–167 (cf. p. 52).

[Sha90] Adi Shamir. « IP = PSPACE ». In : *FOCS*. 1990, p. 11–15 (cf. p. 254).

[She92] Alexander Shen. « IP = PSPACE : Simplified Proof ». In : *J. ACM* 39.4 (1992), p. 878–880 (cf. p. 254).

[SHL65] Richard Edwin Stearns, Juris Hartmanis et Philip M. Lewis. « Hierarchies of memory limited computations ». In : *SWCT (FOCS)*. 1965, p. 179–190 (cf. p. 102).

[Sip83] Michael Sipser. « A Complexity Theoretic Approach to Randomness ». In : *STOC*. 1983, p. 330–335 (cf. p. 201, 274).

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997, p. I–XV, 1–396 (cf. p. xii).

[SM73] Larry J. Stockmeyer et Albert R. Meyer. « Word Problems Requiring Exponential Time : Preliminary Report ». In : *STOC*. 1973, p. 1–9 (cf. p. 110).

[Smo87] Roman Smolensky. « Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity ». In : *STOC*. 1987, p. 77–82 (cf. p. 299).

[Str73a] Volker Strassen. « Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten ». In : *Numerische Mathematik* 20.3 (1973), p. 238–251 (cf. p. 304, 307).

[Str73b] Volker Strassen. « Vermeidung von Divisionen ». In : *Journal für die reine und angewandte Mathematik* 264.2 (1973), p. 184–202 (cf. p. 149).

[Str74] Volker Strassen. « Polynomials with Rational Coefficients Which are Hard to Compute ». In : *SIAM J. Comput.* 3.2 (1974), p. 128–149 (cf. p. 304).

[STV99] Madhu Sudan, Luca Trevisan et Salil P. Vadhan. « Pseudorandom Generators Without the XOR Lemma (Extended Abstract) ». In : *STOC*. 1999, p. 537–546 (cf. p. 338, 344).

[Sud96] Madhu Sudan. « Maximum Likelihood Decoding of Reed Solomon Codes ». In : *FOCS*. 1996, p. 164–172 (cf. p. 333, 338).

[SY10] Amir Shpilka et Amir Yehudayoff. « Arithmetic Circuits : A survey of recent results and open questions ». In : *Foundations and Trends in Theoretical Computer Science* 5.3-4 (2010), p. 207–388 (cf. p. 375).

[Sze87] Róbert Szelepcsényi. « The method of focing for nondeterministic automata ». In : *Bulletin of the EATCS* 33 (1987), p. 96–99 (cf. p. 125).

[Tod89] Seinosuke Toda. « On the Computational Power of PP and $\oplus$P ». In : *FOCS*. 1989, p. 514–519 (cf. p. 228).

[Tra64] Boris Trakhtenbrot. « Turing Computations with Logarithmic Delay ». In : *Algebra i Logika* 3.4 (1964), p. 33–48 (cf. p. 38).

[Tur37] Alan M. Turing. « On Computable Numbers, with an Application to the Entscheidungsproblem ». In : *Proceedings of the London Mathematical Society* s2-42.1 (1937), p. 230–265 (cf. p. x, 1, 14, 43).

[Val79a] Leslie G. Valiant. « The Complexity of Computing the Permanent ». In : *Theor. Comput. Sci.* 8 (1979), p. 189–201 (cf. p. 240).

[Val79b] Leslie G. Valiant. « The Complexity of Enumeration and Reliability Problems ». In : *SIAM J. Comput.* 8.3 (1979), p. 410–421 (cf. p. 214).

[Vin05] N. V. Vinodchandran. « A note on the circuit complexity of PP ». In : *Theor. Comput. Sci.* 347.1-2 (2005), p. 415–418 (cf. p. 291).

[VV86] Leslie G. Valiant et Vijay V. Vazirani. « NP is as Easy as Detecting Unique Solutions ». In : *Theor. Comput. Sci.* 47.3 (1986), p. 85–93 (cf. p. 229, 230).

[Wil07] Ryan Williams. « Time-Space Tradeoffs for Counting NP Solutions Modulo Integers ». In : *IEEE Conference on Computational Complexity*. 2007, p. 70–82 (cf. p. 207).

[Wil11] Ryan Williams. « Non-uniform ACC Circuit Lower Bounds ». In : *IEEE Conference on Computational Complexity*. 2011, p. 115–125 (cf. p. 304, 371, 375).

[Wol06] Pierre Wolper. *Introduction à la calculabilité*. 3ème édition. Dunod, 2006 (cf. p. xiii, 29).

[WW86] K. Wagner et G. Wechsung. *Computational Complexity*. Springer, 1986 (cf. p. xii).

[Yao82] Andrew Chi-Chih Yao. « Theory and Applications of Trapdoor Functions (Extended Abstract) ». In : *FOCS*. 1982, p. 80–91 (cf. p. 319).

[Žák83] Stanislav Žák. « A Turing Machine Time Hierarchy ». In : *Theor. Comput. Sci.* 26 (1983), p. 327–333 (cf. p. 52).

[Zip79] Richard Zippel. « Probabilistic algorithms for sparse polynomials ». In : *EUROSAM*. T. 72. 1979, p. 216–226 (cf. p. 331).

Index