

COURS DE MATHÉMATIQUES

TOME X
INFORMATIQUE

Mathématiques générales

France ~ 2025

Écrit et réalisé par Louis Lascaud

Table des matières

1	Analyse numérique	5
1.1	Résolution de systèmes linéaires	5
2	Théorie de la complexité	7
2.1	Problèmes en informatique théorique	7
2.1.1	Fonctions booléennes	7
2.1.2	Problèmes de décision et problèmes de recherche	8
2.2	Machines de Turing	8
2.2.1	Définition et premières propriétés	8
2.2.2	Description	10
2.2.3	Lecture seule, lecture unique, écriture seule	10
2.2.4	Graphe des configurations	11
2.2.5	Temps de calcul	12
2.2.6	Turing-complétude	12
2.2.7	Machine de Turing non déterministe	12
2.2.8	Machines prenant conseil	13
2.2.9	Automates	13
2.2.10	Machine de Turing universelle	14
2.2.11	Réalisation d'une machine de Turing	14
2.3	Circuits booléens	14
2.3.1	Définitions	14
2.3.2	Applications booléennes et circuits	15
2.3.3	Formules et circuits	15
2.4	Classes de complexité usuelles	17
2.4.1	Classes en temps	17
2.4.1.1	La classe P	17
2.4.1.2	La classe NP	18
2.4.1.3	La classe de complexité co-NP. Classes complémentaires	19
2.4.1.4	La classe PH	20
2.4.1.5	Les classes EXPTIME et NEXPTIME	21
2.4.1.6	La classe P/poly	21

2.4.2	Classes en espace	22
2.4.2.1	La classe PSPACE	23
2.4.2.2	La classe L	23
2.4.2.3	La classe NL	24
2.4.2.4	La classe EXPSPACE	25
2.4.3	Classes énoncées en termes de circuits	25
2.4.3.1	La classe AC^i	25
2.4.3.2	La classe NC^i	25
2.4.3.2.1	Uniformisation des classes AC^i et NC^i	26
2.4.4	Classes aléatoires	26
2.4.4.1	La classe aléatoire RP	26
2.4.4.2	La classe aléatoire BPP	27
2.5	Relations entre classes de complexité : une première approche	27
2.5.1	Relations simples entre classes de complexité	27
2.5.1.1	Relations entre classes de complexité sous conditions	27
2.5.1.2	Des inclusions absolues moins triviales entre classes spatiales et temporelles	28
2.5.1.3	Inclusions faisant intervenir des classes aléatoires	29
2.5.2	Relations moins évidentes entre classes de complexité	30
2.5.2.1	Effondrement sous condition $NP \subseteq P/poly$	30
2.5.2.2	Problème $P = NP$ en espace	31
2.5.2.3	Rôle du problème STCONN et égalité $NL = co-NL$	32
2.6	Complétude	33
2.6.1	Introduction : problèmes NP-complets	33
2.6.2	Définition générale de la complétude	35
2.6.3	Un problème PSPACE-complet	35
2.6.4	Relation $NL = co-NL$	36
2.6.5	Applications P-complètes	37
2.7	Relations entre classes de complexité données grâce à la complétude	38
2.7.1	Résolution de la chaîne classique d'inclusions	38
2.7.2	Existence de problèmes non polynomiaux non complets	40
2.7.3	$P = NP$ ne peut être déterminé par un argument diagonal	41
2.8	Complexité par les formules	42
2.8.1	Temps de calcul d'une fonction booléenne par une formule	42
2.8.2	Programmes branchés	43
2.8.3	Application du théorème de Barrington	45
2.9	Théorie de la décision	45

Chapitre 1

Analyse numérique

Résumé

L'analyse numérique étudie les méthodes programmables en algorithmique pour résoudre des problèmes mathématiques. C'est donc une discipline bel et bien mathématique à l'interface avec l'informatique théorique. Parmi les problèmes les plus étudiés, citons :

- ★ la réduction des systèmes linéaires ;
- ★ la convergence des méthodes de calcul d'une valeur d'une fonction en un point ;
- ★ l'interpolation polynomiale ;
- ★ les méthodes d'approximation géométriques ;
- ★ le calcul intégral informatique ;
- ★ la résolution d'équations différentielles par discrétisation ;
- ★ l'optimisation, linéaire ou non linéaire.

1.1 Résolution de systèmes linéaires

Théorème. (*Caractérisation de la solvabilité par l'orthogonal*)

Soit A une matrice de taille $m \times n$. L'équation $Ax = b$ admet au moins une solution si et seulement si b est orthogonal à $\text{Ker}(A^*)$.

Chapitre 2

Théorie de la complexité

Résumé

Dans un premier temps, on définit les concepts de base du traitement algorithmique informatique : machines de Turing, automates, etc. Dans un second temps, on introduit l'outil des circuits booléens qui facilite nombre de preuves de la section suivante. Dans un troisième temps, on introduit plusieurs classes de complexité et l'on explicite les liens entre elles, dont certains énoncés constituent des questions encore ouvertes à ce jour, parmi lesquels l'un des plus fameux problèmes du millénaire.

2.1 Problèmes en informatique théorique

2.1.1 Fonctions booléennes

→ *Notation.* On note $\{0,1\}^* = \bigcup_{n=1}^{\infty} \{0,1\}^n$ l'ensemble des suites finies de 0 et de 1. Si $x \in \{0,1\}^*$, on note $|x|$ la *longueur* de x qui est l'unique $n \in \mathbb{N}^*$ tel que $x \in \{0,1\}^n$.

Définition. (*Fonction booléenne*)

Une *fonction booléenne* est une fonction (non nécessairement définie partout, sinon on dit *application booléenne*) $\phi : \{0,1\}^* \rightarrow \{0,1\}$. Dans notre contexte, une fonction booléenne représente un *problème*, d'où l'identification terminologique parfois.

Définition. (*Fonction pseudo-booléenne*)

Une *fonction pseudo-booléenne* est une fonction (non nécessairement définie partout, sinon on dit *application booléenne*) $\phi : \{0,1\}^* \rightarrow \{0,1\}^*$.

→ *Convention.* On appelle parfois *fonction booléenne* une application de $\{0,1\}^n$ dans $\{0,1\}^m$.

2.1.2 Problèmes de décision et problèmes de recherche

Définition. (*Problème de décision*)

Soient p un polynôme et ψ une application de deux variables $x \in \{0,1\}^*$ et $y \in \{0,1\}^{p(|x|)}$, i.e. une application booléenne vue d'une certaine manière. Soit $x \in \{0,1\}^*$. La question : « existe-t-il y tel que $\psi(x,y) = 1$ » est un *problème de décision*. Un tel y est dit *certificat*.

Définition. (*Problème de recherche*)

Soient p un polynôme et ψ une application de deux variables $x \in \{0,1\}^*$ et $y \in \{0,1\}^{p(|x|)}$, i.e. une application booléenne vue d'une certaine manière. Soit $x \in \{0,1\}^*$. La question : « trouver y tel que $\psi(x,y) = 1$ » est un *problème de recherche*. Un tel y est encore dit *certificat*.

2.2 Machines de Turing

2.2.1 Définition et premières propriétés

En informatique théorique, une machine de Turing est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur. Ce modèle a été imaginé par Alan TURING en 1936, en vue de donner une définition précise au concept d'algorithme ou de « procédure mécanique ». Il est toujours largement utilisé en informatique théorique, en particulier dans les domaines de la complexité algorithmique et de la calculabilité.

À l'origine, le concept de machine de Turing, inventé avant l'ordinateur, était censé représenter une personne virtuelle exécutant une procédure bien définie, en changeant le contenu des cases d'un ruban infini, en choisissant ce contenu parmi un ensemble fini de symboles. D'autre part, à chaque étape de la procédure, la personne doit se placer dans un état particulier parmi un ensemble fini d'états. La procédure est formulée en termes d'étapes élémentaires du type : « si vous êtes dans l'état 42 et que le symbole contenu sur la case que vous regardez est « 0 », alors remplacer ce symbole par un « 1 », passer dans l'état 17, et regarder maintenant la case adjacente à droite ».

Définition. (*Machine de Turing*)

Une *machine de Turing* ou *machine de Turing déterministe* ou *MT* est

Reformulation pratique. (*Machine de Turing*)

Une machine de Turing est un concept abstrait composé des éléments suivants :

- (i) un *ruban infini* dans les deux sens ou *bande*, éventuellement plusieurs rubans dont l'un choisi par défaut est dit *ruban de travail*, divisé en *cases* ;

- (ii) un alphabet \mathcal{A} , dit *alphabet de travail*, souvent $\{0,1\}$, de sorte que les cases sont remplies des symboles de l'alphabet, à raison d'une lettre par case ; on suppose que $\mathcal{A} \ni B$ est non vide où B fixé est appelé *symbole blanc*, aussi souvent noté 0 ou # ;
- (iii) une *tête de lecture*, qui est dans un état (on dit, mais cela n'a pas ici de sens précis, qu'il existe un *registre d'état* qui mémorise l'état courant de la machine) ;
- (iv) un ensemble fini Σ d'états.

À chaque *étape* d'un *calcul*, la tête est devant une case, lit son contenu et décide de trois choses : le contenu de ladite case à la prochaine étape, son propre état pour la prochaine étape, et si elle se déplace à droite ou à gauche. Autrement dit, la tête, ou *fonction de transition*, est une application

$$\delta : A \times \Sigma \rightarrow A \times \Sigma \times \{G, D\}.$$

On note aussi $G = \leftarrow$ et $D = \rightarrow$. Si l'on n'identifie pas la tête à δ , on l'appelle *table d'actions* ou *partie de contrôle* qui est une entité non réalisée.

En particulier, le choix de l'état suivant par la tête est indépendant de sa position, mais dépend a priori et du symbole de la case lue et de son propre état actuel. On suppose qu'il existe un état $q_0 \in \Sigma$ dit *état initial* dans lequel la machine se trouve au départ, *i.e.* à la première étape ; cet état n'est pas forcément attaché à la définition de la machine et peut dépendre d'une mise en situation. De plus, on suppose qu'il existe un état $h \in \Sigma$ dit *halte* de sorte que si la machine arrive à cet état, elle s'arrête. La suite des étapes peut être finie ou infinie.

Dans certains livres, δ est une fonction et non une application et l'image point de $(a, s) \in A \times \Sigma$ hors du domaine de définition est définie dans la définition précédente par (a, h, G) où G est fixé tout à fait arbitrairement.

On peut également fixer $\Sigma \subseteq \mathcal{A}$ un *alphabet d'entrée* constitué des symboles dans les cases à la première étape ne contenant pas nécessairement B , et un ensemble $F \subseteq \Sigma$ d'*états acceptants* ou *finals* ou *finaux* constitué des états qui peuvent précéder une halte selon δ .

→ *Convention.* Les cases du ruban dont on ne précise pas quel symbole de l'alphabet elles contiennent, contiennent par défaut le symbole blanc. En particulier toutes les cases sont remplies.

VOC L'*entrée* de la machine est la case où elle *commence*, c'est-à-dire où elle se trouve à la première étape, ou, selon une convention annexe, le ruban à la première étape : dans ce cas, c'est que le ruban n'a qu'un nombre fini de symbole non blancs et à la première étape, la tête de la machine se trouve sur le premier symbole non blanc. Parfois, on impose qu'une case contenant B ne peut être une entrée, mais c'est bénin et dépend d'un changement mineur dans δ . L'entrée n'est pas intrinsèque à la machine. La *sortie* de la machine est le symbole devant la tête quand la machine s'arrête, ou, selon une convention annexe, le ruban à la halte. Elle dépend uniquement

de l'entrée pour une machine de Turing déterministe.

→ *Notation.* δ peut facilement être représentée sous la forme d'un tableau à deux entrées quitte à colorier d'une couleur ou d'une autre les cases selon qu'elles mènent à droite ou à gauche.

Remarque. Lorsque le contexte le permet, ce qui est souvent le cas, on se permet quelques abus quant à l'argument ou à la valeur de δ en considérant des projections, par exemple.

Définition. (*Configuration d'une machine de Turing*)

La *configuration* (d'une machine de Turing à une étape du calcul) consiste en les symboles dans les cases du ruban à cette étape, l'état de la tête à cette étape et la position de la tête à cette étape.

Heuristique

La notion de machine de Turing est trop générale pour être facilement manipulable.

2.2.2 Description

Une machine de Turing peut se voir de nos jours comme l'abstraction d'un ordinateur. Dans une machine de Turing,

- ★ la partie de contrôle représente un microprocesseur ;
- ★ la bande représente la mémoire interne de l'ordinateur ;
- ★ la tête de lecture représente la bus (*et l'on comprend pourquoi différencier abstraitement tête de lecture et partie de contrôle*).

2.2.3 Lecture seule, lecture unique, écriture seule

Définition. (*Lecture seule*)

Un ruban d'une machine de Turing est dit *en lecture seule* si la machine ne peut pas changer le contenu des cases, autrement dit avec les notations de la définition $\pi_A \circ \delta = \pi_A$.

Définition. (*Lecture unique*)

Un ruban d'une machine de Turing est dit *en lecture unique* ou *en écriture unique* ou *une seule passe* si après avoir lu une case du ruban, la machine doit se déplacer à droite, autrement dit avec les notations de la définition si $\text{Im}(\pi_{\{G,D\}} \circ \delta) = \{D\}$.

Définition. (*Écriture seule*)

Un ruban d'une machine de Turing déterministe dit *à écriture seule* est une façon de voir les choses où dans une machine, deux têtes non autonomes sont présentes, l'une

lisant le ruban, l'autre écrivain seulement, et c'est celle-ci que l'on appelle à écriture seule. Concrètement, cela revient à considérer la complexité de la machine de Turing sans les contraintes d'écriture.

Heuristique

La machinerie de Turing est un modèle robuste : on peut varier beaucoup d'hypothèses, sur le nombre de ruban, fixer certaines composantes et leur rôle, etc., cela ne changera pas le paradigme déterministe ou non.

2.2.4 Graphe des configurations

Définition. (*Graphe des configurations*)

Le *graphe de(s) configurations* d'une machine de Turing T , déterministe ((voir le sens ci-après)) ou non, a pour sommets les configurations possibles de T , et deux configurations C et C' sont reliées par une arête orientée de C à C' si et seulement si $C' = \delta(C)$ pour δ une/la fonction de transition de C .

Remarque. En général, le graphe de Turing est très gros, *i.e.* il a beaucoup de sommets.

Définition. (C_{init}, C_{accept})

Soit une machine de Turing explicitement sur l'alphabet $\{0,1\}$. On note C_{init} l'ensemble des *configurations initiales* possibles d'une machine de Turing. C'est un singleton si l'on prend la convention

On note $C_{accepte}$ l'ensemble des *configurations acceptées*, *i.e.* dont l'état est la halte et la sortie est 1.

Définition. (*Configuration acceptante*)

Une configuration d'une machine de Turing, en particulier une configuration initiale, est *acceptante* s'il existe un chemin orienté sur le graphe de configurations partant d'elle et vers une configuration acceptée.

Principe. (*Ajout d'un état ACCEPTÉ*)

À un graphe des configurations, on peut toujours ajouter un sommet dit *état* ou *configuration* ACCEPTÉ relié au reste du graphe seulement par un chemin orienté de chaque configuration acceptée vers lui.

Alors une configuration initiale est acceptante si et seulement si on peut trouver un chemin orienté partant d'elle vers ACCEPTÉ.

2.2.5 Temps de calcul

Dans toute la suite, on considérera des machines de Turing sur l'alphabet booléen $\{0,1\}$.

Définition. (Calcul d'une fonction par une machine)

Soit ϕ une fonction booléenne. Soit T une machine de Turing. On dit que T calcule ϕ si pour chaque $x \in \{0,1\}^*$, si x est l'entrée de T , $\phi(x)$ est la sortie.

VOC On dit aussi qu'on calcule $\phi(x)$ si la machine à entrée x a pour sortie $\phi(x)$. Plus généralement encore, une machine de Turing peut calculer une fonction ϕ sur une partie de $\{0,1\}^*$.

Remarque. On pourrait définir la calculabilité d'une fonction pseudo-booléenne sans vrai problème, par restriction de la définition précédente.

Définition. (Temps de calcul)

Soit ϕ une fonction booléenne. Soit T une machine de Turing calculant ϕ . Soit $x \in \{0,1\}^*$. Le temps de calcul de $\phi(x)$ par T est le nombre d'étapes de T avant que T ne s'arrête.



Le temps de calcul peut être infini.

Remarque importante. Souvent, on considérera des temps de calcul en ordres de grandeur. Il ne faut pas s'étonner à partir de là de rencontrer des expressions du type : telle machine calcule tel problème en temps polynomial, cela signifie par exemple : le temps de calcul est $t(x) = \mathcal{O}(p(|x|))$ où p est un polynôme fixé, de même avec toute autre unité de mesure d'odg.

2.2.6 Turing-complétude

2.2.7 Machine de Turing non déterministe

Définition. (Machine de Turing non déterministe)

Une machine de Turing non déterministe ou non déterminée ou MTND est une machine de Turing avec deux fonctions de transitions, par exemple δ_0 et δ_1 .

Définition. (Calcul d'une fonction par une machine non déterministe)

Soit ϕ une fonction booléenne. Soit T une machine de Turing non déterministe de transitions δ_0, δ_1 . On dit que T calcule ϕ si pour chaque $x \in \{0,1\}^*$, alors

- (i) $\phi(x) = 1$ si et seulement si l'on peut trouver une suite de configurations $C_0, \dots, C_m, m \in \mathbb{N}$, où C_0 est la configuration initiale avec entrée x , de sorte que l'état

de T dans C_m est sa halte, sa sortie est 1, et pour tout $i \in \llbracket 1, m \rrbracket$, $C_i = \delta_0(C_{i-1})$ ou $\delta_1(C_{i-1})$;

- (ii) $\phi(x) = 0$ si et seulement si l'on peut trouver une suite de configurations C_0, \dots, C_m , $m \in \mathbb{N}$, où C_0 est la configuration initiale avec entrée x , de sorte que l'état de T dans C_m est sa halte, sa sortie est 0, et pour tout $i \in \llbracket 1, m \rrbracket$, $C_i = \delta_0(C_{i-1})$ ou $\delta_1(C_{i-1})$.

Remarque. Les imprécisions dans la définition s'appliquent de la même manière que pour le temps de calcul d'une machine déterministe. De même, pour la définition du temps de calcul à suivre.

Cette dernière définition est attendue mais plus complexe que dans le cas déterministe.

Définition. (*Temps de calcul par une machine non déterministe*)

Soit ϕ une fonction booléenne. Soit T une machine de Turing non déterministe calculant ϕ . Soit $x \in \{0,1\}^*$. Le *temps de calcul* de $\phi(x)$ par T est la longueur minimale d'une suite de configurations du point (i) de la définition précédente si $\phi(x) = 1$ et du point (ii) si $\phi(x) = 0$.

Remarque. Cette définition est cohérente avec la définition du cas déterministe : si $\delta_0 = \delta_1$, on retrouve le calcul d'une fonction booléenne par une machine de Turing.

2.2.8 Machines prenant conseil

2.2.9 Automates

Définition. (*Automate*)

Un *automate* est une machine de Turing à lecture seule et unique.

2.2.10 Machine de Turing universelle

2.2.11 Réalisation d'une machine de Turing

2.3 Circuits booléens

2.3.1 Définitions

Définition. (*Circuit booléen*)

Un *circuit booléen* est un graphe orienté acyclique fini, dont les sommets sont étiquetés par \wedge (une *porte ET*), \vee (une *porte OU*) ou \neg (une *porte NON*), sauf les sommets de degré d'entrée nul au sens des graphes orientés, qui sont les *entrées*. Les sommets de degré de sortie nul sont les *sorties*. On impose de plus qu'un sommet étiqueté par \neg a toujours un unique fils.

La *taille* du circuit booléen est le nombre de sommets, soit de portes et d'entrées réunies. La *profondeur* du circuit est la longueur du chemin orienté le plus long dans le graphe orienté sous-jacent, reliant donc une entrée à une sortie.

→ *Notation.* Si x est l'entrée du circuit C , on note $C(x)$ sa sortie.

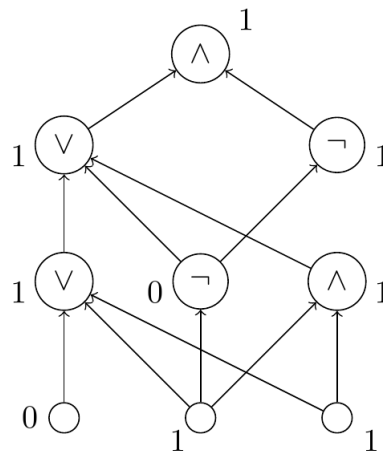


FIGURE 2.3.1 : *Exemple de circuit booléen.* —

On a de plus étiqueté les sommets pour associer au circuit une application booléenne selon la méthode décrite ci-dessous.

Remarque. Par simplification, on peut toujours supprimer les entrées constantes d'un circuit (s'en convaincre).

2.3.2 Applications booléennes et circuits

Méthode. (*Application pseudo booléenne associée à un circuit booléen*)

À un circuit booléen à n entrées rangées dans un certain ordre et m sorties, on peut associer une application $\phi : \{0,1\}^n \rightarrow \{0,1\}^m$ définie récursivement : si les entrées sont v_1, \dots, v_n , pour calculer un certain $\phi(x)$ où $x = (x_1, \dots, x_n)$, on donne la valeur x_i en v_i , puis on respecte les règles suivantes.

1. La valeur d'une porte *ET* est 1 si et seulement si tous ses fils^a ont la valeur 1.
2. La valeur d'une porte *OU* est 1 si et seulement s'il a un moins un fils de valeur 1.
3. Une porte *NON* a un seul enfant, et sa valeur est l'opposée de la valeur de l'enfant.

$\phi(x)$ est alors le m -uplet des valeurs obtenues pour les sorties.

En particulier, à un circuit booléen à une seule sortie, on peut associer une fonction booléenne.

Réciproquement, si ϕ est booléenne, on dit qu'un circuit booléen *calcule* ϕ si ϕ est l'application booléenne associée à ce circuit.

^a On rappelle que dans un graphe orienté $s_1 \rightarrow s_2$, s_1 est le fils de s_2 .

Lemme

Chaque application $\phi : \{0,1\}^n \rightarrow \{0,1\}$ peut être calculée par un circuit booléen de taille inférieure à $2^{n+1} + n + 1$.

▷ Soit $X = \{x \in \{0,1\}^n \mid \phi(x) = 1\}$. Pour chaque $x \in X$, il existe un circuit dont l'unique sortie est étiquetée par 1 si et seulement si l'entrée est x . Si les sommets d'entrée sont v_1, \dots, v_n , si $x_i = 0$, on attache une porte *NON* à v_i , puis l'on attache une porte *ET* à toutes ses portes *NON* et aux sommets v_i tel que $x_i = 1$. On fait cela pour chaque $x \in X$, et l'on attache enfin une porte *OU* à toutes ces portes *ET* et c'est terminé. ■

2.3.3 Formules et circuits

Reformulation pratique. (*Définition récursive des formules binaires*)

La classe des formules binaires sur un alphabet de variables booléennes $(x_i)_{i \in I}$ dit *littéraux*, ainsi que la *taille* et la *profondeur* des formules, est définie par récurrence de la manière suivante :

- (i) x_i est une formule de taille 1 et de profondeur 0 ;
- (ii) si ϕ est une formule de taille m et profondeur d , alors $\neg\phi$ est une formule de taille m et de profondeur d ;
- (iii) si ϕ_i est une formule de taille m_i et de profondeur d_i pour $i = 1, 2$, alors $\phi_1 \wedge \phi_2$ et $\phi_1 \vee \phi_2$ sont des formules et de taille $m_1 + m_2$ et de profondeur $\max(d_1, d_2) + 1$.

Ainsi, la taille d'une formule est le nombre de littéraux dans la formule, avec répétitions.

Méthode. (Circuit booléen associé à une formule binaire)

À chaque formule binaire, on peut associer récursivement un circuit de degré d'entrée ≤ 2 en rajoutant la porte correspondant en une racine à une ou deux formules.

Fait

Le graphe d'un circuit associé à une formule binaire est une arbre.

Exercice 1

Montrer que tout circuit booléen dont le graphe est un arbre est associé à une formule binaire.

Principe

En utilisant en pensée les lois de de Morgan, qui ne s'appliquent pas dans la définition des formules binaires, tout circuit associé à une formule binaire peut être remplacé par un circuit également associé à cette formule et dont toutes les portes NON sont au fond, *i.e.* aux entrées du circuit et à la racine des profondeurs de la formule.

Exemples. (Formules binaires)

1. La taille de $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ est de 4 et sa profondeur est 2.
2. La taille de $(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_2)$ est de 6 et sa profondeur est 2.

Lemme

Toute formule de taille $m > 1$ possède une sous-formule de taille entre $\frac{m}{3}$ et $\frac{2m}{3}$.

▷ En effet, mettons toutes les portes ET au fond. À chaque génération, on sépare la formule initiale de taille m en 2, de sorte qu'il y a au moins une formule de taille $\geq \frac{m}{2}$. Jusqu'à la profondeur 1, on peut donc trouver un chemin dans l'arbre qui à chaque étape divise la taille de la formule par 2. Il est impossible par un tel procédé qu'à aucune étape la taille de la formule ne soit dans $[\frac{m}{3}, \frac{2m}{3}]$. En considérant ce point du chemin construit, on obtient un sous-arbre de l'arbre initial, donc bien une formule binaire déduite de l'initiale. ■

Définition. (Formule binaire calculant une fonction booléenne)

On dit qu'une formule binaire *calcule* une fonction booléenne f si le circuit booléen associé calcule f .

→ *Notation.* Écrivons $\sigma(f)$ la taille minimale d'une formule qui calcule f fonction booléenne $\{0,1\}^* \rightarrow \{0,1\}$.

2.4 Classes de complexité usuelles

2.4.1 Classes en temps

Principe. (*Classes de complexité (temporelle)*)

Une *classe de complexité* est un sous-ensemble de l'ensemble des fonctions booléennes caractérisé par l'existence de machines de Turing (dépendant de la fonction) qui la calculent en un temps de calcul majoré (dépendant de la fonction et de l'entrée, indépendant de la machine). Cette majoration doit être en ordre de grandeur indépendante de l'entrée.

Définition. (*Classe de complexité DTIME*)

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ une application croissante. Alors $\text{DTIME}(f(n))$ est la classe d'applications qu'on peut calculer en temps $\mathcal{O}(f(n))$ avec une machine de Turing.

Définition. (*Classe de complexité NTIME*)

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ une application croissante. Alors $\text{NTIME}(f(n))$ est la classe d'applications qu'on peut calculer en temps $\mathcal{O}(f(n))$ avec une machine de Turing non déterministe.

Exemples. (*Classe DTIME, NTIME*)

1. La classe $\text{DTIME}(n)$ est celle des problèmes que l'on peut résoudre avec une quantité spatiale linéaire, de même $\text{NTIME}(n)$, avec une quantité spatiale linéaire dans le cas non déterministe.
2. La classe $\text{DTIME}(n^2)$ est celle des problèmes que l'on peut résoudre avec une quantité spatiale quadratique, de même $\text{NTIME}(n^2)$, avec une quantité spatiale quadratique dans le cas non déterministe.
3. La classe $\text{DTIME}(\log(n))$ est celle des problèmes que l'on peut résoudre avec une quantité spatiale sous-linéaire, de même $\text{NTIME}(\log(n))$, avec une quantité spatiale sous-linéaire dans le cas non déterministe.

2.4.1.1 La classe P

Définition. (*Classe de complexité P*)

La *classe de complexité P* est l'ensemble des fonctions booléennes ϕ telles qu'il existe un polynôme en une variable p tel que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing T qui calcule ϕ avec un temps de calcul de $\phi(x)$ d'au plus $p(|x|)$. Autrement dit, toute entrée de ϕ se calcule par une TM en temps polynomial indépendant de la machine et de l'entrée.

Remarque. Autrement dit, $P = \bigcup_{k \in \mathbb{N}} \text{DTIME}(n^k)$.

Exemples. (Problèmes de classe P)

1. Étant donné un graphe fini et deux sommets de ce graphe, pour $k \in \mathbb{N}$, le problème d'existence d'un chemin de longueur k sur ce graphe entre ces deux sommets, est encodable par une machine de Turing et se calcule en temps polynomial.
On appelle généralement STCONN le problème de déterminer s'il y a un chemin orienté de s à t dans un graphe orienté.
2. Ce n'est qu'en 2002 qu'il a été montré que le problème de détermination si un nombre est premier ou non est dans P !

2.4.1.2 La classe NP**Définition. (Classe de complexité NP)**

La *classe de complexité* NP est l'ensemble des fonctions booléennes ϕ telles qu'il existe un polynôme en une variable p tel que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing non nécessairement déterministe T qui calcule ϕ avec un temps de calcul de $\phi(x)$ d'au plus $p(|x|)$. Autrement dit, toute entrée de ϕ se calcule par une TM ou une TMND en temps polynomial indépendant de la machine et de l'entrée.

Remarque. Autrement dit, $P = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$.

Exemples. (Problèmes de classe NP)

1. Le problème d'existence d'un cycle hamiltonien dans un graphe est de classe NP.

Fait. (Inclusion $P \subseteq NP$)

$P \subseteq NP$.

Simplement car déterministe \implies non déterministe.

Heuristique

Heuristiquement, un problème de classe P est un problème dont la solution est « facile à trouver », tandis qu'un problème de classe NP est un problème dont la solution est « facile à vérifier ». Cette inclusion assure donc que s'il est facile de trouver une solution à un problème, il est également facile de vérifier qu'une solution est valable, ce qui énoncé comme cela n'est plus une énorme trivialité.

Conjecture**?**

$P = NP$.

Inclusions strictes entre classes de complexité

On croit que la hiérarchie polynomiale est une vraie hiérarchie, mais en général, il est très difficile de montrer qu'une classe de complexité est strictement incluse dans une autre. Au-delà du problème très célèbre de savoir si oui ou non $P = NP$, on ignore encore si l'égalité est infirmée pour des classes de complexité beaucoup plus faibles que nous allons introduire dans la suite.

Théorème. (Définition équivalente de la classe NP)

Une fonction booléenne $\phi \in NP$ si et seulement s'il existe un polynôme $p \in \mathbb{R}[X]$ et une application $\psi \in P$ telle que $\forall x \in \{0,1\}^* \quad \phi(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)} \quad \psi(x,y) = 1$.

▷ Si ϕ peut être calculée en temps polynomial par une machine de Turing non déterministe, pour chaque x on peut choisir une suite $\varepsilon_1, \dots, \varepsilon_m$ où $m \in \mathbb{N}, m \leq p(|x|)$ telle que si l'on choisit les fonctions de transition $\delta_{\varepsilon_1}, \dots, \delta_{\varepsilon_m}$, on calcule $\phi(x)$. Alors, soit ψ l'application qui calcule $\phi(x)$ étant donnés x et $\varepsilon_1, \dots, \varepsilon_m$. Il est clair que $\psi \in P$.

Réciproquement, si une telle ψ existe, on peut utiliser une machine de Turing non déterministe pour écrire y et calculer $\psi(x,y)$. ■

La quantification introduite dans cette caractérisation rend intuitive la description de plusieurs autres classes de complexité, en premier lieu la

2.4.1.3 La classe de complexité co-NP. Classes complémentaires**Définition. (Classe de complexité co-NP)**

Une fonction booléenne $\phi \in \text{co-NP}$ si et seulement s'il existe un polynôme $p \in \mathbb{R}[X]$ et une application $\psi \in P$ telle que $\forall x \in \{0,1\}^* \quad \phi(x) = 1 \iff \forall y \in \{0,1\}^{p(|x|)} \quad \psi(x,y) = 1$.

Reformulation pratique. (Calcul de co-NP)

$\text{co-NP} = 1 - NP = \{1 - \phi, \phi \in NP\}$.



Cette autre définition n'est donc pas du tout symétrique à la première !

Définition. (Complémentaire d'une classe de complexité)

On appelle *complémentaire* d'une classe de complexité C , ou *coclasse* de C , la classe définie en remplaçant les quantifications existentielles par des quantifications universelles et vice versa.

Fait. (Coclasse de complexité)

Soit C une classe de complexité. Alors $\text{co}C = 1 - C$.

Ceci a bien un sens puisqu'une fonction booléenne est à valeur dans $\mathbb{Z}/2\mathbb{Z}$.

Principe. (Inclusion des coclasses existentielles dans les classes)

Si C est une classe de complexité définie par une quantification existentielle, alors $\text{co}C \subseteq C$.

▷ Puisque la quantification universelle implique l'existentielle. ■

Proposition. (Involutivité du complémentaire de classes de complexité)

Soit C une classe de complexité. Alors $\text{co}(\text{co}C) = C$.

Proposition. (Croissance des coclasses)

Si deux classes de complexité $C \subseteq C'$, alors $\text{co}C \subseteq \text{co}C'$.

Conjecture**?**

$\text{NP} = \text{co-NP}$.

Fait. (Symétrie de la classe P)

$\text{coP} = \text{P}$.

2.4.1.4 La classe PH

On souhaite généraliser les trois constructions précédentes P, NP et co-NP. Pour définir la classe PH, on a besoin de sous-classes définies de manière imbriquées, qui sont les classes $(\Sigma_n)_{n \in \mathbb{N}}$ et $(\Pi_n)_{n \in \mathbb{N}}$.

On définit par récurrence :

- ★ $\Sigma_0 = \Pi_0 = \text{P}$;
- ★ $\Sigma_1 = \text{NP}$, $\Pi_1 = \text{co-NP}$;
- ★ soit $i \in \mathbb{N}$. On caractérise : $\phi \in \Sigma_{i+1}$ s'il existe $\psi \in \Pi_i$ telle qu'il existe p polynomiale telle que $\forall x \quad \phi(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)} \quad \psi(x,y) = 1$; de même, $\phi \in \Pi_{i+1}$ s'il existe $\psi \in \Sigma_i$ telle qu'il existe p polynomiale telle que $\forall x \quad \phi(x) = 1 \iff \forall y \in \{0,1\}^{p(|x|)} \quad \psi(x,y) = 1$.

Définition. (Classe de complexité PH)

On définit $\text{PH} = \bigcup_{n=0}^{\infty} (\Sigma_n \cup \Pi_n)$.

Fait. (*La classe PH est volumineuse*)

$$P \subseteq PH, NP \subseteq PH \text{ et } co-NP \subseteq PH.$$

2.4.1.5 Les classes EXPTIME et NEXPTIME

On peut généraliser les classes P, NP avec de plus grands ordres de grandeur.

Définition. (*Classe de complexité EXPTIME*)

La *classe de complexité* EXPTIME est l'ensemble des fonctions booléennes ϕ telles qu'il existe un polynôme en une variable p tel que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing T qui calcule ϕ avec un temps de calcul de $\phi(x)$ d'au plus $\exp(p(|x|))$.

Définition. (*Classe de complexité NEXPTIME*)

La *classe de complexité* NEXPTIME est l'ensemble des fonctions booléennes ϕ telles qu'il existe un polynôme en une variable p tel que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing non nécessairement déterministe T qui calcule ϕ avec un temps de calcul de $\phi(x)$ d'au plus $\exp(p(|x|))$.

Propriétés. (*Inclusions P avec EXPTIME et analogues non déterministes*)

1. $P \subseteq EXPTIME$.
2. $NP \subseteq NEXPTIME$.

2.4.1.6 La classe P/poly

Définition. (*Classe de complexité P/poly*)

Une fonction booléenne $\phi \in P/poly$ si et seulement s'il existe un polynôme $p \in \mathbb{R}[X]$ et pour chaque $n \in \mathbb{N}$, une suite finie $(y_i) \in \{0,1\}^{p(n)}$ et une application $\psi_n \in P$ telle que $\forall x \in \{0,1\}^* \quad \phi(x) = \psi_n(x, y_n)$. On appelle une telle $(y_i)_{i \in \llbracket 1, p(n) \rrbracket}$ une *suite de conseil*.

Théorème. (*Caractérisation de la classe P/poly*)

Soit ϕ une application booléenne. Alors $\phi \in P/poly$ si et seulement si pour tout $n \in \mathbb{N}$, on peut calculer $\phi(x)$ pour tout $x \in \{0,1\}^n$ grâce à un circuit booléen de taille polynomiale, suite de circuits dite *modèle non uniforme de calcul*.

▷ Le sens réciproque est facile : il suffit de prendre pour suite de conseil un encodage du circuit. Montrons le sens direct. Montrons que si $\phi \in P$, il existe une suite $(C_n)_{n \in \mathbb{N}}$ de circuits de taille polynomiale telle que pour tout $n \in \mathbb{N}$, pour tout $x \in \{0,1\}^n$, $\phi(x) = C_n(x)$. Soit par hypothèse T une machine de Turing qui calcule ϕ en temps polynomiale. Si T calcule $\phi(x)$ en temps m , alors la position de la tête est toujours entre $-m$ et m . Alors on peut coder la configuration de T à l'étape

t en utilisant des variables $x_{-m}^t, x_{(m-1)}^t, \dots, x_0^t, \dots, x_{m-1}^t, x_m^t$ où x_i^t est le contenu de la case i pour tout $i \in \llbracket -m, m \rrbracket$ et, en notant Σ l'ensemble d'états de la tête de T , on définit $(y_{ij}^t)_{i \in \llbracket -m, m \rrbracket, j \in \Sigma}$ par 1 si la tête est dans la position i et l'état j à l'étape t et 0 sinon. Alors ce nombre de variables est $(2m+1) + (2m+1) \times \text{taille}(\Sigma)$ qui est polynomial en n . Or on peut voir assez facilement que x_i^t et y_{ij}^t ne dépendent que d'un nombre borné des variables x_i^{t-1} et y_{ij}^{t-1} . Mais toute application pseudo-booléenne d'un nombre borné k d'entrées peut être calculée par un circuit de taille bornée... On peut donc réunir tous ces circuits ensemble pour créer un circuit de taille $\mathcal{O}_{|\Sigma|}(m^2)$.

Par conséquent, si $\phi \in \text{P/poly}$, il existe une suite $(C_n)_n$ de circuits de taille polynomiale qui calculent ϕ . En effet, soit $n \in \mathbb{N}$ et soit alors (y_n) une suite de conseil et soit $\psi \in \text{P}$ une application booléenne telle que $\phi(x) = \psi(x, y_n)$ pour tout $x \in \{0,1\}^n$. Par ce qui précède, il existe C'_n de taille polynomiale qui calcule ψ pour les entrées de taille $n + p(n)$ où $p(n) = |y_n|$. (De plus, après simplification, on peut rempalcer C'_n par un circuit C_n dont l'entrée est x .) ■

Exemple. (*Problème de classe P/poly*)

Calculer le chiffre des unités dans l'écriture du carré d'un entier si le nombre de symboles dans l'écriture du nombre est un nombre premier et 0 sinon, nécessite un modèle non uniforme de calcul. Heuristiquement, un seul algorithme ne suffit pas.

On définira plus tard d'autres classes de complexité uniquement en termes de circuits booléens.

2.4.2 Classes en espace

Définition. (*Classe de complexité DSPACE*)

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ une application croissante. Alors $\text{DSPACE}(f(n))$ est la classe d'applications qu'on peut calculer en espace $\mathcal{O}(f(n))$ avec une machine de Turing.

Définition. (*Classe de complexité NSPACE*)

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ une application croissante. Alors $\text{NSPACE}(f(n))$ est la classe d'applications qu'on peut calculer en espace $\mathcal{O}(f(n))$ avec une machine de Turing non déterministe.

Exemples. (*Classe DSPACE, NSPACE*)

1. La classe $\text{DSPACE}(n)$ est celle des problèmes que l'on peut résoudre avec une quantité spatiale linéaire, de même $\text{NSPACE}(n)$, avec une quantité spatiale linéaire dans le cas non déterministe.
2. La classe $\text{DSPACE}(n^2)$ est celle des problèmes que l'on peut résoudre avec une quantité spatiale quadratique, de même $\text{NSPACE}(n^2)$, avec une quantité spatiale quadratique dans le cas non déterministe.
3. La classe $\text{DSPACE}(\log(n))$ est celle des problèmes que l'on peut résoudre avec une

quantité spatiale sous-linéaire, de même $\text{NSPACE}(\log(n))$, avec une quantité spatiale sous-linéaire dans le cas non déterministe.

2.4.2.1 La classe PSPACE

Définition. (*Classe de complexité PSPACE*)

La *classe de complexité PSPACE* est l'ensemble des fonctions booléennes ϕ telles qu'il existe un polynôme en une variable p et une machine de Turing T qui calcule ϕ tels que pour tout $x \in \{0,1\}^*$, la position de T pendant le calcul est comprise entre $-p(|x|)$ et $p(|x|)$ à translation près du ruban, ou ce qui est équivalent (pourquoi?), dans un intervalle de cases de longueur $p(|x|)$.

Remarque. Autrement dit, $\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}(n^k)$.

Exemple fondamental. (*Problème de classe PSPACE*)

Le problème de détermination du gagnant au jeu de go, si les joueurs utilisent des stratégies optimales, n'est pas PH mais est PSPACE.

2.4.2.2 La classe L

On veut définir la classe des problèmes n'utilisant que des ressources « en quantité logarithmique ». Problème, une telle restriction telle quelle est évidemment trop forte : simplement l'écriture de l'entrée dans un ruban n'est pas un problème résolu en temps logarithmique, mais linéaire ! Il faut donc y aller plus subtilement.

Définition. (*Classe de complexité L*)

La *classe de complexité L* est l'ensemble des fonctions booléennes ϕ telles que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing T à deux rubans dont le premier est à lecture seule, contient pour entrée x , et le deuxième est le ruban de travail, de sorte que pour chaque x , la machine ne regarde que $\mathcal{O}(\log(|x|))$ de cases du ruban de travail et calcule $\phi(x)$.

Remarque. Autrement dit, $L = \text{DSPACE}(\log(n))$.

Exercice 2

On dit qu'une fonction booléenne $\phi \in \text{LP}$ si pour tout $x \in \{0,1\}^*$, il existe une machine de Turing à deux rubans dont le premier est à lecture seule, contient pour entrée x , et le deuxième est de travail et tel que pour chaque x , la machine ne regarde que $\mathcal{O}(\log(p(|x|)))$

de cases du ruban de travail et calcule $\phi(x)$. Montrer que $LP = L$.

Heuristique

C'est la complexité minimale raisonnable.

L'inclusion extrêmement grossière suivante est peut-être une égalité, car on ne sait pas si

Conjecture

?

$L \subsetneq PH$.

Mnémonik : on ne peut pas prouver que les miracles n'existent pas...

2.4.2.3 La classe NL

Définition. (Classe de complexité NL)

La *classe de complexité* NL est l'ensemble des fonctions booléennes telles que pour tout $x \in \{0,1\}^*$, il existe une machine de Turing non déterministe T à deux rubans dont le premier est à lecture seule, contient pour entrée x , et le deuxième est le ruban de travail, de sorte que pour chaque x , la machine ne regarde que $\mathcal{O}(\log(|x|))$ de cases du ruban de travail et calcule $\phi(x)$.



On voudrait dire : *une fonction booléenne $\phi \in NL$ si et seulement s'il existe un polynôme $p \in \mathbb{R}[X]$ et une application $\psi \in L$ telle que $\phi(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)} \quad \psi(x,y) = 1$* , mais ce n'est pas le cas.

Exercice 3

Démontrer que l'affirmation précédente est fausse par un contre-exemple hors de NL

On peut bel et bien caractériser, de même qu'avec P et NP, la classe NL à partir de L, mais il faut être plus fin.

Théorème. (Définition équivalente de la classe NL)

Une fonction booléenne $\phi \in NL$ si et seulement si pour tout $x \in \{0,1\}^*$, il existe une machine de Turing à trois rubans dont le premier est à lecture seule et contient l'entrée x , le deuxième est à lecture unique et le troisième est un ruban de travail, dont la machine n'utilise qu'une quantité logarithmique de cases, avec : $\phi(x) = 1$ si et seulement s'il existe une variable y et ψ calculée par une machine de Turing telle que si x et dans le premier ruban et y est dans le deuxième, alors la sortie de T est $\phi(x)$.

⊗ (*Idée de la preuve.*) C'est un peu plus subtil et laissé aux lecteurs plus aiguisés. ■

2.4.2.4 La classe EXPSPACE

Définition. (*Classe de complexité* EXPSPACE)

$$\text{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}(2^{n^k}).$$

2.4.3 Classes énoncées en termes de circuits

2.4.3.1 La classe AC^i

On peut alors définir d'autres classes de complexité en termes de circuits.

Définition. (*Classe de complexité* AC^i)

La *classe de complexité* AC^i est l'ensemble des fonctions booléennes ϕ telles qu'il existe pour tout $n \in \mathbb{N}$ un circuit booléen C_n de taille polynomiale et de profondeur $\mathcal{O}(\log(n)^i)$ qui calcule $\phi(x)$ pour tout $x \in \{0,1\}^n$.

En particulier, pour AC^0 , on peut utiliser une profondeur de circuit bornée et pour AC^1 , on peut utiliser une profondeur logarithmique.

Remarque. Pour tout $i \in \mathbb{N}$, on a $\text{AC}^i \subseteq \text{AC}^{i+1}$.

Définition. (*Classe de complexité* AC)

$$\text{AC} := \bigcup_{i=0}^{\infty} \text{AC}^i.$$

2.4.3.2 La classe NC^i

Définition. (*Classe de complexité* NC^i)

La *classe de complexité* NC^i avec N pour Nick PIPPIER est l'ensemble des fonctions booléennes ϕ telles qu'il existe pour tout $n \in \mathbb{N}$ un circuit booléen C_n de taille polynomiale, de profondeur $\mathcal{O}(\log(n)^i)$ et de degré d'entrée maximal dans le circuit au plus 2, qui calcule $\phi(x)$ pour tout $x \in \{0,1\}^n$.

En particulier, de même, pour NC^0 , on peut utiliser une profondeur de circuit bornée et pour NC^1 , on peut utiliser une profondeur logarithmique.

Remarque. Pour tout $i \in \mathbb{N}$, on a $\text{NC}^i \subseteq \text{NC}^{i+1}$.

Exercice 4

Montrer que l'on peut remplacer « au plus 2 » par « borné » dans la définition des NC^i .

Remarque. L'intérêt des NC^i commence à partir de $i = 1$. En effet, dans NC^0 , si l'on s'impose un nombre borné d'entrées et une profondeur bornée, alors la sortie dépend d'un nombre borné des variables.

Définition. (Classe de complexité NC)

$$\text{NC} := \bigcup_{i=0}^{\infty} \text{NC}^i.$$

2.4.3.2.1 Uniformisation des classes AC^i et NC^i **Définition. (Suite de circuits uniforme logarithmique)**

On dit qu'une suite $(C_n)_{n \in \mathbb{N}}$ de circuits booléens est *uniforme en espace logarithmique* s'il existe un machine de Turing qui peut calculer chaque C_n en temps polynomial avec un ruban de travail de taille $O(\log(n))$ et un autre ruban à écriture seule avec la définition connue.

Définition. (Classe de complexité $\text{NC}_{\text{unif}}^i$)

La classe de complexité $\text{NC}_{\text{unif}}^i$ est la version uniforme de la classe NC^i avec la définition de suites de circuits uniformes précédente.

Remarque. En particulier :

- ★ $\text{NC}_{\text{unif}}^i \subseteq \text{NC}^i$ pour tout $i \in \mathbb{N}$,
- ★ et donc $\text{NC}_{\text{unif}} \subseteq \text{NC}$.

2.4.4 Classes aléatoires**2.4.4.1 La classe aléatoire RP****Définition. (Classe de complexité RP)**

La classe de complexité RP est l'ensemble des fonctions booléennes ϕ telles qu'il existe $\psi \in \text{P}$ et un polynôme p tels que si $\phi(x) = 0$, alors $\psi(x, y) = 0$ pour tout $y \in \{0, 1\}^{p(|x|)}$ et si $\phi(x) = 1$, alors $\mathbb{P}[\psi(x, y) = 1] \geq \frac{1}{2}$ quand y est choisi au hasard de $\{0, 1\}^{p(|x|)}$, où \mathbb{P} est la probabilité uniforme =.

Autrement dit, pour essayer de calculer $\phi(x)$, on choisit un y au hasard, on calcule $\psi(x, y)$. S'il est égal à 1, on sait que $\phi(x) = 1$ et sinon, on n'est pas sûr, mais quand même au moins à moitié sûr.

Remarque. On peut choisir y_1, \dots, y_k au hasard et alors $\mathbb{P}(\exists i \ \psi(x, y_i) = 1) =$

$$\begin{cases} 0 & \text{si } \phi(x) = 0 \\ \geq 1 - 2^{-j} & \text{sinon.} \end{cases}$$

2.4.4.2 La classe aléatoire BPP

Définition. (*Classe de complexité BPP*)

La *classe de complexité BPP* est l'ensemble des fonctions booléennes ϕ telles qu'il existe $\psi \in P$ et un polynôme p tels que $\mathbb{P}[\phi(x) = \psi(x, y)] \geq \frac{2}{3}$ quand y est pris au hasard dans $\{0, 1\}^{p(|x|)}$.

Exercice 5

Montrer que dans la définition de BPP, si l'on choisit y_1, \dots, y_k au hasard, alors $\mathbb{P}[\phi(x) = \psi(x, y_i) \text{ pour } > \frac{k}{2} \text{ des } i] \geq 1 - \exp(-ck)$ pour une constante absolue $c > 0$.

2.5 Relations entre classes de complexité : une première approche

2.5.1 Relations simples entre classes de complexité

2.5.1.1 Relations entre classes de complexité sous conditions

Lemme

Si $P = NP$, alors $P = \text{co-NP}$.

▷ Parce que P est symétrique. ■

Proposition. (*Conséquence fondamentale de $P = NP$*)

Si $P = NP$, alors toute la hiérarchie polynomiale s'effondre, au sens que $P = PH$.

▷ On raisonne par récurrence. Par le lemme, on a $P = \Sigma_1 = \Pi_1$. Supposons maintenant que pour un $k \in \mathbb{N}^*$, $P = \Sigma_k = \Pi_k$ et soit $\phi \in \Sigma_{k+1}$. Alors il existe $\psi \in \Pi_k$ et un polynôme p tels que $\phi(x) = 1 \iff \exists y \in \psi(x, y) = 1$, mais $\Pi_k = P$, et donc $\phi \in NP \subseteq P$. Donc $\Sigma_{k+1} \subseteq P$, et par symétrie, $\Pi_{k+1} \subseteq P$, d'où le résultat. ■

Heuristique

Pour certains, c'est le signe que $P \neq NP$. Cependant, selon Timothy GOWERS, ce n'est pas une raison convaincante : les conséquences de $P = NP$ sont déjà assez spectaculaires pour qu'on puisse interpréter $P = PH$ comme un simple corollaire.

Mais encore, plus finement :

Proposition

Soit $k \in \mathbb{N}^*$. Si $\Sigma_k = \Pi_k$, alors PH s'effondre jusqu'au niveau k de la hiérarchie polynomiale, c'est-à-dire $PH = \Sigma_k = \Pi_k$.

▷ On raisonne encore par récurrence. Supposons que Σ_r et Π_r soient dans Σ_k et soit $\phi \in \Sigma_{r+1}$. Alors il existe $\psi \in \Pi_r \subseteq \Sigma_k$ tel que $\phi(x) = 1 \iff \exists y \psi(x,y) = 1$. Mais puisque $\psi \in \Sigma_k$, il existe $\omega \in \Pi_{k-1}$ et q tels que $\psi(x,y) = 1 \iff \exists z \omega(x,y,z) = 1$. Alors $\phi(x) = 1 \iff \exists y,z \omega(x,y,z) = 1$, mais $\omega \in \Pi_{k-1}$, et donc $\phi \in \Sigma_k$. Symétriquement, $\Pi_{r+1} \subseteq \Sigma_k$. ■

2.5.1.2 Des inclusions absolues moins triviales entre classes spatiales et temporelles**Proposition**

$NP \subseteq PSPACE$.

▷ Soit $\phi \in NP$ et $\psi \in P$, p un polynôme tels que $\phi(x) = 1 \iff \exists y \psi(x,y) = 1$. Pour un algorithme donné, on peut essayer tous les y dans l'ordre lexicographique, qui est d'ailleurs l'ordre numérique si on les considère comme des nombres en binaire. Pour chaque calcul de $\psi(x,y)$, on a besoin d'une quantité polynomiale de mémoire, et on peut réutiliser la mémoire pour chaque calcul. ■

Proposition

$PSPACE \subseteq EXPTIME$. Ainsi, un calcul utilisant une mémoire polynomiale peut se faire en temps exponentiel relativement à une taille polynomiale sur l'entrée.

▷ Si $\phi \in PSPACE$ et T est une machine de Turing qui calcule ϕ en utilisant une quantité polynomiale de mémoire, alors le nombre de configurations possibles de T pendant le calcul est borné par $2^{p(|x|)}$ pour un polynôme p . Par le principe des tiroirs, la machine ne peut être deux fois dans la même configuration, parce qu'elle doit s'arrêter et ne pas entrer dans un cycle, par hypothèse. Donc, le calcul se termine en temps $\leq 2^{p(|x|)}$. ■

Remarque. On a montré en fait quelque chose de plus fort : le même algorithme utilisé dans PSPACE se termine en temps exponentiel.

Dans le même esprit :

Proposition

$NL \subseteq P$.

▷ Soit $\phi \in NL$ et soit T une machine de Turing non déterministe qui calcule ϕ avec des fonctions de transition δ_0 et δ_1 . Le nombre de configurations de T est au plus $2^{\mathcal{O}(\log(n))} \times n \times |\Sigma| \times \mathcal{O}(\log(n)) \leq p(n)$ pour un polynôme p . Autrement dit, le nombre de sommets du graphe des configurations de T est au plus polynomial. De plus $\phi(x) = 1$ si et seulement s'il existe un chemin orienté dans le graphe des configurations qui commence à la configuration initiale et qui termine à une configuration acceptée. Or on sait que que l'on peut ajouter une configuration C_* au graphe relié à toutes les configurations acceptées et elles seulement. Le problème de savoir si $\phi(x) = 1$ revient alors à trouver un chemin dans un graphe orienté entre deux sommets C_0 et C_* , mais il est clair que c'est un problème qui est dans P . En outre, le graphe des configurations peut être déterminé en temps polynomial, d'où le résultat en joignant les deux temps polynomiaux. ■

Proposition

Pour tout $i \in \mathbb{N}$, $AC^i \subseteq NC^{i+1}$. Ainsi :

$$NC^0 \subseteq AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq \dots$$

▷ Une porte ET ou OU de degré m peut être réalisée par un nombre de porte de degré d'entrée 2 de profondeur $\lceil \log_2(m) \rceil$, donc si C est un circuit de profondeur k et de taille m qui calcule ϕ , on peut le remplacer par un circuit de taille $\leq m^2$ et de profondeur $\leq k \log(m)$. ■

2.5.1.3 Inclusions faisant intervenir des classes aléatoires**Proposition**

$BPP \subseteq P/\text{poly}$.

▷ Soit $\phi \in BPP$. On a vu qu'il existe $\psi \in P$ et un polynôme p tels que $\mathbb{P}[\phi(x) = \psi(x, y)] \geq 1 - 2^{-n}$ (en fait, on avait vu $1 - \exp(-cn)$, mais quitte à prendre k assez grand linéaire en n , on peut se ramener à une puissance de 2) si on choisit y au hasard de $\{0,1\}^{p(n)}$ où $n = |x|$. Il s'ensuit qu'il existe y_n tel que $\phi(x) = \psi(x, y_n)$ pour tout x . On peut donc utiliser (y_n) comme suite de conseil. ■

On peut se demander si BPP , non uniforme, est inclus dans une classe uniforme. La réponse est oui, mais il faut aller un peu plus loin dans la hiérarchie polynomiale.

Théorème. (Sipser-Lautemann, 1983)

$BPP \subseteq \Sigma_2 \subseteq \Pi_2$.

▷ Par symétrie, il suffit de montrer que $\text{BPP} \subseteq \Sigma_2$. Si $\phi \in \text{BPP}$, on peut choisir d'abord une ψ et un polynôme p tels que $\mathbb{P}[\phi(x) = \psi(x,y)] \geq 1 - 2^{-n}$ pour les mêmes arguments que précédemment, $y \in \{0,1\}^{p(n)}$ choisi au hasard, $n = |x|$. Pour tout x , soit $A_x = \{y \mid \psi(x,y) = 1\}$. Maintenant, si $\phi(x) = 1$, alors $\mathbb{P}[y \notin A_x] \leq 2^{-n}$. On peut choisir z_1, \dots, z_n avec m au plus polynomial en n tel que $\{0,1\}^{p(n)} = \mathbb{F}_2^{p(n)} = \bigcup_{i=1}^m A_x \oplus z_i$ où \oplus est l'addition modulo 2. Concrètement, $y \in A_x \oplus z_i \iff y \oplus z_i \in A_x$.

Si on choisit les z_i au hasard et indépendamment, alors pour tout y , la probabilité $\mathbb{P}[y \notin \bigcup_{i=1}^m A_x \oplus z_i] \leq 2^{-mn}$. Si $mn > (n)$, alors la probabilité que z_1, \dots, z_m marche est strictement positive, et donc il existe un tel choix de z_1, \dots, z_m . Par suite, $\phi(x) = 1 \implies \exists z_1, \dots, z_m \forall y \in \{0,1\}^{p(n)} \exists i \in [1, m] \quad \psi(x, y \oplus z_i) = 1$ et $\exists i \in [1, m] \quad \psi(x, y \oplus z_i) = 1$ se calcule en temps polynomial. D'autre part si $\phi(x) = 0$, la densité de A_x est au plus 2^{-n} et donc il n'est pas possible que $\bigcup_{i=1}^m A_x \oplus z_i = \{0,1\}^{p(n)}$ et donc on a l'implication réciproque, d'où $\phi \in \Sigma_2$. ■

2.5.2 Relations moins évidentes entre classes de complexité

2.5.2.1 Effondrement sous condition $\text{NP} \subseteq \text{P/poly}$

Lemme

Si $\text{P} = \text{NP}$, alors pour chaque problème dans NP , on peut résoudre le problème de recherche en temps polynomial.

▷ Soit $\psi \in \text{P}$ une application de deux variables, ce qui encore une fois n'est qu'un artifice de pensée : c'est toujours une application définie sur $\{0,1\}^*$. S'il existe y tel que $\psi(x,y) = 1$, on peut en trouver un en utilisant l'algorithme suivant : d'abord, considérer le problème de décision : existe-t-il y tel que $|y| = p(|x|)$, $y_1 = 1$ et $\psi(x,y) = 1$? Si $\text{P} = \text{NP}$, on peut par caractérisation résoudre ce problème en temps polynomial. Si la réponse est oui, soit $\epsilon_1 = 1$ et sinon, soit $\epsilon_1 = 0$. Ensuite, on considère le problème : existe-t-il y tel que $|y| = p(|x|)$, $y_1 = \epsilon_1$, $y_2 = 1$, $\psi(x,y) = 1$? Si oui, alors soit $\epsilon_2 = 1$ et sinon $\epsilon_2 = 0$. On continue jusqu'à construire une suite $\epsilon_1, \epsilon_2, \dots, \epsilon_{p(|x|)}$ et l'on a une suite ϵ telle que $\psi(x, \epsilon) = 1$. ■

Ce lemme se transpose pour les circuits :

Lemme

Si $\text{NP} \subseteq \text{P/poly}$, alors pour tout $\psi \in \text{P}$ de deux variables $x \in \{0,1\}^*$, $y \in \{0,1\}^{p(|x|)}$, il existe pour tout n un circuit C_n de n entrées et $p(n)$ sorties tel que pour tout x , s'il existe y tel que $\psi(x,y) = 1$, alors $C_n(x)$ est un tel y , i.e. $\psi(x, C_n(x)) = 1$.

▷ L'idée de la preuve est la même mais les détails sont légèrement différents. Commençons par prendre un circuit C^1 de taille polynomiale tel que $C^1(x) = 1$ si et seulement si $\exists y \in \{0,1\}^{p(n)} \quad y_1 = 1, \psi(x,y) = 1$ par hypothèse. Ensuite, on prend un circuit C^2 de taille polynomiale tel que $C^2(x) = 1$ si et seulement si $\exists y \in \{0,1\}^{p(n)} \quad y_1 = C^1(x), y_2 = 1, \psi(x,y) = 1$. On continue jusqu'à $C^{p(n)}$. En regroupant

tous ces circuits, on crée un circuit C_n de n entrées tel que la sortie de C_n est $(C^1(x), C^2(x), \dots, C^{p(n)}(x))$. S'il existe y tel que $\psi(x, y) = 1$, alors $(C^1(x), \dots, C^{p(n)}(x))$ est un tel y . ■

On peut alors démontrer :

Théorème. (Karp-Lipton, 1980)

Si $\text{NP} \subseteq \text{P/poly}$, alors $\Sigma_2 = \Pi_2$ et donc il y a effondrement polynomial jusqu'au niveau 2, soit $\Sigma_2 = \Pi_2 = \text{PH}$.

▷ Par symétrie, il suffit de montrer que $\Sigma_2 \subseteq \Pi_2$. Soit $\phi \in \Sigma_2$. Soient p, q deux polynômes et soit $\omega \in \text{P}$ une application booléenne telle que $\phi(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)} \exists z \in \{0,1\}^{q(|x|)} \omega(x, y, z) = 1$. À partir de là, on veut échanger l'ordre des quantificateurs. Utilisons le lemme. Pour chaque x comme précédemment, soit $\psi_x : \{0,1\}^{p(|x|)} \rightarrow \{0,1\}$ la fonction booléenne définie par $\psi_x(y) = 1 \iff \exists z \omega(x, y, z) = 1$. Pour tout x , $\psi_x \in \text{NP}$ et donc par l'hypothèse du théorème, il existe un circuit C_x de taille polynomiale tel que si $\psi_x(y) = 1$, alors $\omega(x, y, C_x(y)) = 1$ par le lemme. Donc si $\phi(x) = 1$, et alors $\forall y \psi_x(y) = 1$, il existe C_x tel que pour tout $y \in \{0,1\}^{p(|x|)}$, $\omega(x, y, C_x(y)) = 1$. Si $\phi(x) = 0$, alors $\exists y \forall z \omega(x, y, z) = 0$ et donc il n'existe aucun circuit C_x tel que $\forall y \omega(x, y, C_x(y)) = 1$. Donc, on a montré que $\phi(x) = 1 \iff \exists C_x \forall y \omega(x, y, C_x(y)) = 1$. Alors $\phi \in \Pi_2$. ■

2.5.2.2 Problème $\text{P} = \text{NP}$ en espace

Théorème. (Théorème de Savitch, 1970)

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$. Si $f(n) \geq c \log(n)$ pour une constante réelle c pour tout $n \in \mathbb{N}$, alors $\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f(n)^2)$.

▷ Soit $\phi \in \text{NSPACE}(f(n))$. Soit T une machine de Turing non déterministe qui calcule ϕ en espace $\mathcal{O}(f(n))$ et soit G le graphe de configurations de T quand l'entrée est x , $|x| = n$. Ajoutons comme de coutume un sommet ACCEPTE lié à toutes les configurations qui acceptent x de sorte que $\phi(x) = 1$ si et seulement si on peut trouver un chemin orienté de la configuration initiale à ACCEPTE. Le graphe G peut être calculé en espace $\mathcal{O}(f(n))$, c'est-à-dire, pour deux configurations C et C' , on peut déterminer en espace $\mathcal{O}(f(n))$ s'il existe une arête de C à C' , car $f(n)$ la taille du ruban de travail est plus grande que $c \log(n)$ où $\log(n)$ est le nombre de bits qu'il faut pour spécifier la position de la tête. On rappelle que STCONN est le problème de déterminer s'il y a un chemin orienté de s à t dans un graphe orienté. Pour nous, s est bien sûr la configuration initiale, t la configuration ACCEPTE et G est le graphe des configurations de T . Le nombre de sommets de G est $2^{\mathcal{O}(f(n))}$, car $f(n)$ est au moins logarithmique. Soit $c_k(x, y) = 1$ s'il existe un chemin orienté de x à y de longueur $\leq 2^k$. Pour calculer $c_0(x, y)$, on a vu qu'il suffit d'un espace en $\mathcal{O}(f(n))$. Maintenant, soit $\theta(k)$ la quantité d'espace qu'il faut pour calculer $c_k(x, y)$. Alors $c_{k+1}(x, y) = 1 \iff \exists u \ c_k(x, u) = 1 \text{ ET } c_k(u, y) = 1$. Pour faire ce calcul, il faut $1 + \theta(k)$ cases pour vérifier un u donné, parce qu'on peut réutiliser l'espace pour calculer $c_k(x, u)$ pour calculer $c_k(u, y)$, et pour garder la trace des u qu'on a vérifié, il faut $\log_2(2^{\mathcal{O}(f(n))}) = \mathcal{O}(f(n))$

cases. Donc $\theta(k+1) = \theta(k) + \mathcal{O}(f(n))$. Alors $\theta(k) = \mathcal{O}((k+1)f(n))$. Quand $2^k > 2^{cf(n)}$, $c_k(x,y) = 1$ si et seulement si on peut trouver un chemin de x à y dans G . Donc on peut résoudre le problème STCONN pour G en espace $\mathcal{O}(f(n)^2)$. ■

Corollaire. (*Problème P = NP spatial*)

$\text{PSPACE} = \text{NPSpace}$.

▷ En effet, si $\phi \in \text{NPSpace}$, alors il existe un polynôme p tel que $\text{NPSpace}(p(n)) \subseteq \text{DSPACE}(p(n)^2) \subseteq \text{PSPACE}$. ■

2.5.2.3 Rôle du problème STCONN et égalité $\text{NL} = \text{co-NL}$

Théorème. (*Immerman-Szelepcsényi, 1987*)

$\text{STCONN} \in \text{co-NL}$.

▷ D'abord, remarquons que s'il existe un chemin orienté de s à t dans un graphe, on peut le certifier en donnant une liste $s = s_0, s_1, \dots, s_m = t$ et pour vérifier que la liste nous donne bien un chemin de s à t , on peut utiliser un espace logarithmique en vérifiant qu'il existe une arête entre chaque $s_i \rightarrow s_{i+1}$, ce qui se ramène à lire une matrice d'adjacence. Mais ce n'est pas ce que l'on veut, on a là montré que $\text{STCONN} \in \text{NL}$.

Imaginons que l'on a un certificat que le nombre de sommets qu'on peut atteindre de s par un chemin de longueur $\leq k$ est m . Soit A_k l'ensemble de ces sommets. On peut maintenant créer un certificat en mettant tous les éléments de A_k en ordre lexicographique et en donnant pour chaque $y \in A_k$ un chemin de longueur $\leq k$ de s à y . On peut utiliser ce certificat pour certifier que $z \notin A_k$ si c'est le cas : pour ce faire, on vérifie que les sommets y forment une suite croissante pour l'ordre lexicographique et que pour chaque y , le certificat nous donne un chemin de s à y de longueur $\leq k$ et que le nombre de sommets y est m . Si z n'est pas l'un des y , alors $z \notin A_k$ et cette vérification peut se faire en espace logarithmique. On peut également certifier qu'il n'y a pas un chemin de longueur $\leq k+1$ de s à z .

L'argument s'applique presque partout, mais cette fois, pour chaque y dans le certificat, à la place de vérifier que $y \neq z$, on vérifie qu'il n'y a pas d'arête de y à z . On peut donc utiliser le certificat modifié pour certifier le nombre de z qu'on peut atteindre par un chemin de longueur $\leq k+1$, c'est-à-dire, $|A_{k+1}|$, la modification étant la suivante : c'est le certificat la liste de chemin répétée n fois. Pour chaque sommet z en ordre, on utilise la liste des chemins de s aux sommets dans A_k pour certifier soit que $z \in A_{k+1}$ soit que $z \notin A_{k+1}$ et on compte le nombre de $z \in A_{k+1}$. C'est nécessaire de répéter la liste, car on doit avoir un ruban à lecture seule. Ceci montre que si l'on a un certificat de $|A_k|$, on peut créer un certificat de $|A_{k+1}|$. Par récurrence, on peut certifier $|A_n|$, et en utilisant ce certificat, on peut certifier que $t \notin A_n$ et donc qu'il n'y a aucun chemin de s à t . ■

Remarque. Le calcul de ce certificat se fait en $n^3 \log(n)$.

Pour établir d'autres relations entre des classes de complexité et en particulier l'exemple fondamental de symétrie de NL, on aura besoin de la notion de complétude.

2.6 Complétude

La complétude est le phénomène bizarre que, dans une classe de complexité, si un problème est résoluble en temps raisonnable, alors tous les autres de la classe le sont. Après avoir introduit les problèmes NP-complets, qui sont les plus célèbres, on généralise à d'autres classes de complexité.

2.6.1 Introduction : problèmes NP-complets

Définition. (*Réduction d'applications booléennes*)

Soient ϕ, ψ deux fonctions booléennes. Soit $\theta : \{0,1\}^* \rightarrow \{0,1\}^*$ une *application booléenne à plusieurs sorties*. On dit que θ est une *réduction* de ψ à ϕ si $\psi = \phi \circ \theta$.

Si θ est calculable selon une certaine classe de complexité, la réduction est supposée de cette complexité.

VOC Si par exemple θ est calculable en temps polynomial, on dit que θ est *calculable en temps polynomial*.

Définition. (*NP-complétude*)

Un élément $\phi \in \text{NP}$ est dite *NP-complet* si pour tout $\psi \in \text{NP}$, il existe une réduction polynomiale de ψ à ϕ .

On essaie maintenant de voir qu'il existe de tels problèmes.

Exemple fondamental. (*CIRCUITSAT*)

CIRCUITSAT est le problème où l'entrée est un circuit de n entrées et la sortie est 1 si et seulement s'il existe un choix d'entrées du circuit tel que la sortie du circuit égale 1.

Proposition

CIRCUITSAT est NP-complet.

▷ Soit $\psi \in \text{NP}$ et soient $\omega \in \text{P}$ et p un polynôme, tels que $\psi(x) = 1 \iff \exists y \in \{0,1\}^{p(|x|)} \omega(x,y) = 1$. D'après un lemme déjà vu, pour tout n , on peut construire en temps polynomial un circuit C_n qui calcule ω pour les entrées de taille $n + p(n)$. Pour chaque x , on peut donc construire en temps polynomial un circuit C_x qui est la simplification de C_n où les n premières entrées sont fixées à x_1, \dots, x_n . Alors $C_x(y) = 1 \iff \omega(x,y) = 1$. Donc $\psi(x) = 1 \iff \text{CIRCUITSAT}(C_x) = 1$. ■

Définissons un autre problème très pratique pour démontrer que beaucoup de problèmes sont NP-complets. Pour cela, on redéfinit la notion de formule avec un vocabulaire spécifique aux problèmes complets.

Définition. (Littéraux, clauses, formules, satisfaisabilité)

Soient x_1, \dots, x_n des variables booléennes, *i.e.* des entités qui peuvent prendre la valeur 0 ou 1 librement indépendamment. Un *littéral* est une expression x_i ou $\neg x_i$.

Une *clause* est une disjonction finie de littéraux. La *taille* de la clause est le nombre de littéraux minimal, *i.e.* sans redondance ou vrais, apparaissant.

Une *formule (en forme normale conjonctive)* est une conjonction de clauses. La *taille* d'une formule est la somme des tailles des clauses la constituant.

Une formule est *satisfaisable* tout simplement s'il existe un choix de valeurs des variables telle que la valeur de la formule égale 1.

Exercice 6 (Exemple d'une satisfaisabilité uniquement réalisée)

Montrer que la formule $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ est satisfaisable pour $x_1 = x_2 = x_3 = 1$ et que c'est la seule solution.

Définition. (nSAT)

Soit $n \in \mathbb{N}$. *nSAT* est le problème de déterminer si une formule dont toutes les clauses ont une taille inférieure à n est satisfaisable.

Proposition

3SAT est NP-complet.

▷ Il suffit d'exhiber une réduction polynomiale de CIRCUITSAT à 3SAT. Soit C un circuit booléen de taille n et de degré d'entrée ≤ 2 . Pour chaque sommet, on prend une variable donnée. Pour chaque porte, on trouve une formule de 3SAT qui dit que la valeur du sommet est la valeur dans le circuit :

- ★ Si x_i correspond à une porte *NON* et x_j à son entrée, on prend la formule $(x_i \vee x_j) \wedge (\neg x_j \vee \neg x_i)$.
- ★ Si dans le circuit $x_i = x_j \wedge x_k$ ce qui correspond à une porte *ET*, on veut que $x_i \implies x_j \wedge x_k$ et $x_j \wedge x_k \implies x_i$. On fait correspondre $\neg x_i \vee (x_j \wedge x_k) \rightsquigarrow (\neg x_i \vee x_j) \wedge (\neg x_i \vee x_k)$ et $(\neg x_j \vee \neg x_k \vee x_i)$.
- ★ Les portes *OU* sont traitées de manière similaire.

On a donc construit en temps polynomial une formule qui imite le circuit. Si la variable x_r correspond à la sortie du circuit, on ajoute la clause x_r . Cette formule est satisfaisable si et seulement si le circuit est satisfaisable. ■

2.6.2 Définition générale de la complétude

Définition. (*Classe-complétude*)

Soit C une classe de complexité. Un élément $\phi \in C$ est dite C -complet si pour tout $\psi \in C$, il existe une réduction polynomiale de ψ à ϕ .



Pour une classe C en espace, une réduction polynomiale est par définition encore calculée en temps polynomial dans la définition précédente. Heuristiquement, les réductions ont besoin d'être moins puissantes que les classes de complexité considérées pour n'être pas triviales.

2.6.3 Un problème PSPACE-complet

On commence par donner une définition alternative de formule.

Définition. (*Formule quantifiée*)

Une *formule booléenne quantifiée* est une expression de type

$$Q_1 x_1 \dots Q_n x_n \quad \phi(x_1, \dots, x_n)$$

où chaque Q_i représente au choix \forall ou \exists , et ϕ est une formule booléenne. Une formule quantifiée peut encore être vraie ou fausse.

Définition. (FBQV)

Le problème FBQV pour « formule booléenne quantifiée vraie » est le problème de déterminer si une formule quantifiée est vraie.

Lemme

FBQV \in PSPACE.

▷ Soit f une formule quantifiée $Q_1 x_1 \dots Q_n x_n \quad g(x_1, \dots, x_n)$ où g est une formule booléenne de m variables. Soit $e(m, n)$ l'espace qu'il faut pour déterminer si une telle formule est vraie. Pour déterminer si f est vraie, il suffit de déterminer si elle est vraie quand $x_1 = 0$ et si elle est vraie quand $x_1 = 1$. On peut déterminer cela en espace

$$\underbrace{1}_{\text{pour enregistrer le résultat quand } x_1 = 0} + \underbrace{e(m, n-1)}_{\text{déterminer si la formule simplifiée est vraie}} + \underbrace{\mathcal{O}(m)}_{\text{calculer la formule simplifiée}}$$

Par suite, $e(m, n) = \mathcal{O}(mn)$. ■

Proposition

FBQV est PSPACE-complet.

▷ Montrons que pour toute $\phi \in \text{PSPACE}$, il existe une réduction en espace polynomiale de ϕ à FBQV. La preuve ressemble à celle du théorème de Savitch. Soit donc $\phi \in \text{PSPACE}$. Soit T une machine de Turing calculant ϕ en espace polynomial. Pour chaque x , soit G_x le graphe des configurations de T quand l'entrée est x . On a $\phi(x) = 1$ si et seulement si il existe un chemin orienté de C_{init} à C_{accepte} . Il existe une formule f de taille polynomiale telle que pour chaque couple (C, C') de configurations, $f(C, C') = 1$ si et seulement si $\delta_T(C) = C'$. Ainsi $\theta_k(C, C') = 1$ si et seulement si il existe un chemin de C à C' de longueur inférieure à 2^k . Soit e_k la taille d'une formule quantifiée nécessaire pour calculer θ_k . Alors $\theta_{k+1}(C, C')$ est vraie si et seulement si $\exists C'' \quad \theta_k(C, C'') \wedge \theta_k(C'', C')$.

On ne peut pas utiliser cette formule, car elle est trop grande, mais on peut utiliser la formule équivalente suivante : $\exists C'' \forall D_1, D_2 \quad (D_1 = C \wedge D_2 = C'') \vee (D_1 = C'' \wedge D_2 = C') \implies \theta_k(D_1, D_2)$. Chaque égalité peut être écrite en forme d'une formule booléenne de taille $p(n)$, donc $e_{k+1} \leq e_k + p(n)$. La taille de G est $2^{\mathcal{O}(p(n))}$, donc quand $k \geq p(n)$, $\theta_{C_{\text{init}}, C_{\text{accepte}}} = 1$ si et seulement si $\phi(x) = 1$ et il y a une formule booléenne quantifiée pour θ_k de taille polynomiale. ■

On peut en déduire

2.6.4 Relation NL = co-NL**Proposition**

STCONN \in co-NL.

Pour montrer cela, on a besoin de la notion suivante.

Définition. (Calculabilité logarithmique implicite)

Une application pseudo-booléenne $\phi : \{0,1\}^* \rightarrow \{0,1\}^*$ est *implicitement calculable en espace logarithmique (ICEL)* si les deux applications

$$(x, i) \mapsto \phi(x)_i$$

et

$$(x, i) \mapsto \mathbb{1}_{\{i \leq |\phi(x)|\}}$$

sont dans L.

Lemme. (Stabilité de la calculabilité log implicite par composition)

Si ϕ et ψ sont deux applications ICEL, alors $\psi \circ \phi$ est ICEL.

▷ Pour claculer $\psi(\phi(x))$ étant données des machines de Turing convenant pour ϕ et ψ respectivement T_ϕ et T_ψ , on commence par suivre les opérations de T_ψ , et chaque fois que T_ψ veut lire

une des $\phi(x)_i$, on utilise ϕ pour calculer $\phi(x)_i$. ■

Lemme. (*Stabilité de la calculabilité log implicite par réduction*)

Si $\phi \in L$, respectivement NL, et θ est ICEL, alors $\phi \circ \theta \in L$, respectivement NL.

▷ Si $\phi \in L$ et ϕ est ICEL, alors $\phi \circ \theta$ est ICEL donc trivialement $\phi \circ \theta \in L$ puisqu'elle n'a qu'une sortie. Si $\phi \in NL$, on peut imiter la preuve du lemme précédent pour obtenir une machine de Turing non déterministe qui calcule $\phi \circ \theta$. ■

Définition-propriété. (*Définition équivalente de la NL-complétude*)

Un problème ϕ est NL-*complet* si pour tout $\psi \in NL$, il existe une application θ ICEL telle que $\psi = \phi \circ \theta$.

Proposition

STCONN est NL-complet.

▷ Utilisons la caractérisation précédente. Soit $\psi \in NL$. Pour chaque x , soit G_x le graphe des configurations pour une machine de Turing non déterministe T qui calcule ψ en espace logarithmique quand l'entrée est x . G_x a un nombre polynomial de sommets et G_x est ICEL. De plus, $\phi(x) = 1$ si et seulement s'il existe un chemin L de C_{init} à $C_{accepte}$ dans G_x . Donc $\psi(x) = 1$ si et seulement si $STCONN(G_x, C_{init}, C_{accepte}) = 1$. ■

Corollaire

NL = co-NL.

▷ Soit $\psi \in NL$. Alors il existe θ ICEL telle que $\psi = STCONN \circ \theta$, d'où $1 - \psi = (1 - STCONN) \circ \theta$. Par le théorème d'Immerman-Szelepcsényi, $STCONN \in co-NL$ et donc $1 - STCONN \in L$, d'où $(1 - STCONN) \circ \theta \in NL$ d'où $1 - \psi \in cNL$, d'où $\psi \in co-NL$.

Si $\psi \in co-NL$, alors $1 - \psi \in NL$ d'où $1 - \psi \in co-NL$ d'où $\psi \in NL$. ■

2.6.5 Applications P-complètes

Les applications P-complètes permettraient sous réserve d'existence dans une certaine classe de montrer des égalités non triviales entre classes.

Définition. (*Application P-complète*)

Une application ϕ est P-complète si toute application ψ dans P a une réduction à ϕ en espace logarithmique.

Proposition

Si ϕ est P-complet et $\phi \in L$, alors $P =_c L$.

Proposition

Si ϕ est P-complet et $\phi \in NC_{\text{unif}}$, alors $P = NC$.

Exemple. (Une application P-complète)

Si C est un circuit de n entrées et $x = \{0,1\}^n$, alors $CIRCUITEVAL(C,x)$ est la sortie de C lorsque l'entrée est x .

2.7 Relations entre classes de complexité données grâce à la complétude

2.7.1 Résolution de la chaîne classique d'inclusions

Lemme

Il existe une machine de Turing U universelle et une façon de coder les machines de Turing telle que si τ est un codage de T , alors $U(\tau,x) = T(x)$ pour tout $x \in \{0,1\}^*$ et un polynôme p tel que si t est le temps de calcul de $T(x)$, alors le temps de calcul de $U(\tau,x)$ est au plus $C_T p(t)$.

Définition. (Bonne application arithmétique)

Une application $f : \mathbb{N} \rightarrow \mathbb{N}$ est dite *bonne* si

- (i) f est croissante;
- (ii) $f(n) \geq n$ pour tout n ,
- (iii) on peut calculer f en temps polynomial dans $f(n)$.

Fait

Si f est bonne, $\phi \in DTIME(f(n))$ si l'on peut calculer ϕ en temps $\mathcal{O}(f(n))$.

Théorème

Il existe un polynôme q tel que pour toute bonne application $f : \mathbb{N} \rightarrow \mathbb{N}$, si pour tout $n \in \mathbb{N}$, $g(n) > q(f(n))$, alors $DTIME(f(n)) \subsetneq DTIME(g(n))$.

▷ On utilise un argument diagonal. Il existe une façon de coder les machines de Turing telle que tout x code un T et si x code T , alors $(x, \underbrace{0, \dots, 0}_{n \text{ fois}})$ code T . Définissons ϕ par

$$\phi(\tau) = \begin{cases} 1 - T(\tau) & \text{si } T \text{ est codée par } \tau \text{ et } U \text{ calcule } U(\tau, \tau) \text{ en temps } q(f(\tau)) \\ 0 & \text{sinon} \end{cases} \quad \text{où } p \text{ est le polynôme}$$

donné par le lemme et $p(n) = o(q(n))$. Donc $\phi \in \text{DTIME}(g(n))$. Supposons que T calcule ϕ en temps $\mathcal{O}(f(n))$. Alors, si $|\tau|$ est suffisamment grand, U peut calculer $U(\tau, \tau) = T(\tau)$ en temps $Cp(f(|\tau|)) < q(f(|\tau|)) = g(|\tau|)$, égalité venant de $p = o(q)$. Donc $\phi(\tau) = 1 - T(\tau)$, contradiction. ■

Corollaire

$P \subsetneq \text{EXPTIME}$.

▷ $P \subseteq \text{DTIME}(n^{\log(n)}) \subsetneq \text{DTIME}(q(n^{\log(n)})) \subseteq \text{EXPTIME}$. ■

Corollaire



L'une des inclusions de la chaîne

$$P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

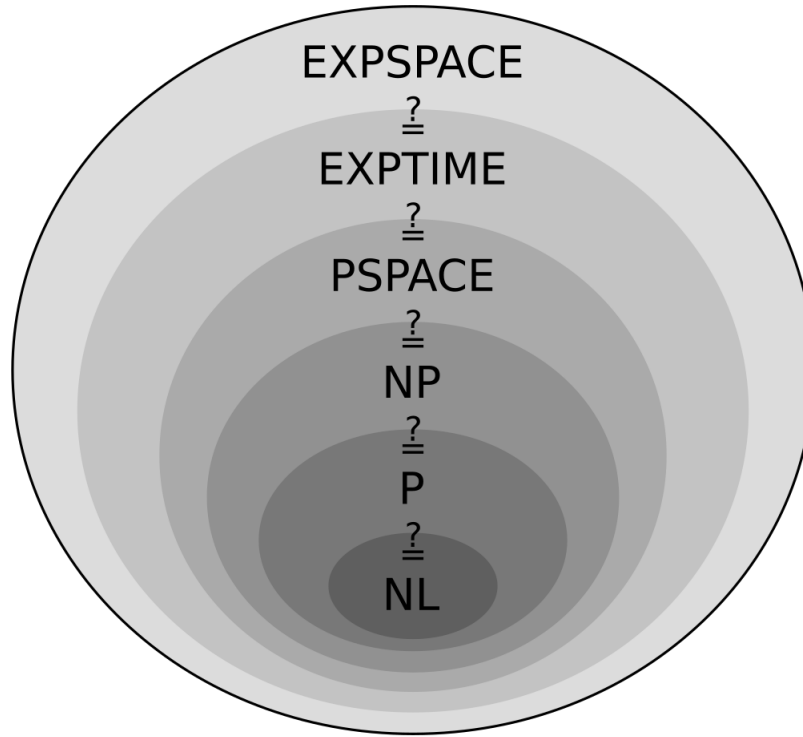
est stricte.

En particulier, $P = NP$ ou $\text{PSPACE} = \text{EXPTIME}$ résout toute la chaîne d'inclusion.

▷ En effet, on a vu que $P = NP \iff P = \text{PSPACE}$. ■

Heuristique

Aujourd'hui, on ne sait pas encore laquelle.

FIGURE 2.7.1 : *Espaces de complexité imbriqués.* —

2.7.2 Existence de problèmes non polynomiaux non complets

Exemples

1. Le problème de trouver un facteur $a \mid n$ avec $1 \leq a \leq m$, est dans $\text{NP} \cap \text{co-NP}$; il serait curieux qu'il soit NP-complet.
2. BABAI a démontré que le problème d'isomorphisme de graphes se résout en temps quasi-polynomial $\exp(\log(n)^c)$.

→ **Notations.** On pose, pour $m \in \mathbb{N}^*$, $O^m = \underbrace{(0, \dots, 0)}_{m \text{ fois}}$ et $I^m = \underbrace{(1, \dots, 1)}_{m \text{ fois}}$.

Théorème. (Ladner)

Si $P \neq \text{NP}$, alors il existe $\phi \in \text{NP} \setminus P$ tel que ϕ n'est pas NP-complet.

▷ On utilise un argument de rembourrage. Nous allons définir une application $f : \mathbb{N} \rightarrow \mathbb{N}$ et ensuite $\varphi(u) = [u = (x, 0, I^{f(|x|-|x|-1)})]$ et x est le codage d'une formule satisfaisable de 3SAT].

Soit T_1, T_2, T_3, \dots une liste de machines de Turing tel que T_i halte en temps $i n^i$ et chaque machine de Turing qui halte en temps polynomial est dans la liste. Pour construire une telle liste, on peut commencer par une liste de toutes les machines de Turing telles que chaque T est dans la liste un nombre infini de fois, puis on fait en sorte que T_i halte en temps $i n^i$. On pose, un peu artificiellement, $f(n) = n^{i(n)}$, $i(1) = 1$ et $i(n+1)$ est donné pour tout entier naturel n par $i(n) + 1$ si $i(n) \leq \log(\log(n))$ ET $\exists u \quad |u| \leq$

$\log(n)$, $u = (x, 0, I^{f(|x|-|x|-1)})$ ET x est le codage d'une forme, $T_{i(n)}(u) \neq 3SAT(x)$, $i(n)$ sinon.

Montrons que $\varphi \in NP$. Pour calculer $\phi(u)$, il faut identifier x , calculer $3SAT(x)$ et vérifier que la longueur du remboursement est correcte. Pour ce dernier, il faut calculer $f(|x|)$. Supposons que nous avons calculé $f(1), \dots, f(n), i(1), \dots, i(n)$. Pour calculer $i(n+1)$, le nombre de n qu'il faut vérifier est $\leq 2^{\log(n)}$, et pour chaque vérification le temps nécessaire est le temps de calcul de $T_{i(n)}(x) \leq (\log(n))^{\log(\log(n))} < n$. Par récurrence, on peut calculer $f(|x|)$ en temps polynomiale dans $|x|$. Donc, le temps de calcul de $i(n+1)$ étant donnés $f(1), \dots, f(n), i(1), \dots, i(n)$ est polynomial, ce qui implique que f est calculable en temps polynomial.

Montrons maintenant que $\phi \notin P$. Si $\phi \in P$, il existe i tel que T_i calcule ϕ . Mais ceci implique $i(n) \leq i$ pour tout n , car si $i(n)$ arrive à i , il reste à i , si $i(n)$ est borné, alors le remboursement est polynomiale et donc si $\phi \in P$, il s'ensuit de $3SAT \in P$, ce qui n'est pas le cas.

Enfin, montrons que ϕ n'est pas NP-complet. Si ϕ était NP-complet, on pourrait réduire $3SAT$ à ϕ en temps $\mathcal{O}(n^k)$ pour un certain $k \in \mathbb{N}$. Mais $i(n)$ n'est pas borné, donc pour n suffisamment grand, $f(n) \geq n^{k+1}$. Si la réduction u d'une formule de taille n est calculée en temps t , alors $|n| \leq t$, et si $u = (x, 0, I^{f(|x|)-|x|-1})$, $|x| \leq t^{\frac{1}{k+1}} < |u|$ quand n est suffisamment grand. Pour calculer $\phi(u)$, il suffit de calculer $3SAT(x)$, donc en temps polynomial. On a une instance plus petite de $3SAT$. Donc $3SAT \in P$, ce qui n'est pas le cas, d'où le résultat. ■

On se demande naturellement si l'on peut utiliser aussi la technique de diagonalisation pour démontrer que $P = NP$. Le théorème suivant permet de se convaincre que non.

2.7.3 $P = NP$ ne peut être déterminé par un argument diagonal

Définition. (*Oracle*)

Un *oracle* A est une application booléenne.

Définition. (*Machine de Turing à oracle*)

Une *machine de Turing à oracle* A est une machine de Turing ayant un ruban supplémentaire auquel elle peut *poser des questions*, c'est-à-dire que si le contenu de ce ruban est z à une certaine étape, alors à la prochaine étape, on pourra calculer $A(z)$.

Définition. (*Classe de complexité à oracle*)

Si C est une classe de complexité définie en termes de machines de Turing et A est un oracle, alors C^A est la classe C définie en remplaçant les machines de Turing par des machines de Turing à oracles A .

Théorème

Il existe un oracle A tel que $P^A = NP^A = EXPTIME$.

▷ Définissons $A(\tau, x) = 1$ si τ est le codage d'une machine de Turing T et $T(x) = 1$ et le temps de calcul de $T(x)$ est $\leq 2^{|\tau|}$. (Le codage est tel que si τ est un codage de T , alors (τ, O^m) est également un codage de T pour m entier naturel.)

Montrons que $\text{EXPTIME} \subseteq \text{P}^A$. Si $\phi \in \text{EXPTIME}$, alors il y a une machine de Turing T qui calcule $\phi(x)$ en temps $2^{p(|x|)}$ pour un polynôme p . Donc on peut rembourrer x par $y = (x, 1, O^m)$ où $m > p(|x|)$ et il existe une machine de Turing T' telle que $T'(y) = 1 \iff T(x) = 1$ et le temps de calcul de T' est $\leq 2^{|y|}$. Alors, pour calculer $\phi(x)$, on rembourre x pour créer y et pose la question (τ', y) à l'oracle A .

Montrons que $\text{NP}^A \subseteq \text{EXPTIME}$. Si $\phi \in \text{NP}^A$, il existe un polynôme q et $\psi \in \text{P}^A$ tel que $\phi(x) = 1 \iff \exists y \in \{0,1\}^{q(|x|)} \psi(x, y) = 1$. Le nombre de possibilités pour y est $\leq 2^{q(|x|)}$ et pendant chaque vérification, si on fait un appel (τ, z) à A , on peut calculer la réponse en temps $\mathcal{O}(2^{|z|})$, et donc le temps de calcul de ϕ est $2^{r(|x|)}$ pour un polynôme r . ■

Théorème. (Baker, Gill et Solovay)

Il existe un oracle B tel que $\text{P}^B \neq \text{NP}^B$.

▷ On utilise un argument diagonal 😊. On définit d'abord une application $\phi \in \text{NP}^B \setminus \text{P}^B$ par $\phi(x) = 1 \iff \exists y \quad |x| = |y| \text{ ET } B(y) = 1$. Clairement, $\phi \in \text{NP}^B$. Étant donné un certificat y , on vérifie que $|y| = |x|$ et en temps 1 que $B(y) = 1$. Maintenant, soit T_1, T_2, \dots une liste de machines de Turing à oracle B telles que le temps de calcul de T_i est $\leq in^i$ et toute machine de Turing dont le temps de calcul est polynomiale est dans la liste.

Supposons que l'on ait choisi un nombre fini de valeurs de B en sorte qu'on peut prouver que T_1, \dots, T_n ne calculent pas ϕ . Soit N plus grand que la taille de tous les y dont on a choisi les valeurs de $B(y)$. Soit $x \in \{0,1\}^N$. Maintenant, on va calculer $T_{n+1}(x)$ en choisissant d'autres valeurs de B si c'est nécessaire. Chaque fois que T_{n+1} fait un appel z à l'oracle B , si on n'a pas déjà choisi $B(z)$, soit $B(z) = 0$. À la fin du calcul, le nombre de choix de $B(z)$ pour $z \in \{0,1\}^N$ est au plus $(n+1)N^{n+1} < 2^N$. Donc il existe $z \in \{0,1\}^N$ pour lequel $B(z)$ n'est pas choisi si $T_{n+1}(x) = 1$, soit $B(z) = 0$ pour tout $z \in \{0,1\}^N$. Si $T_{n+1}(x) = 0$, on choisit z tel que $B(z)$ n'est pas encore choisi et soit $B(z) = 1$. Alors, $\phi(x) \neq T_{n+1}(x)$. Donc aucun des T_n ne calcule ϕ , donc $\phi \notin \text{P}^B$. ■

2.8 Complexité par les formules

2.8.1 Temps de calcul d'une fonction booléenne par une formule

Proposition

Soit f une application booléenne. Les trois propriétés suivantes sont équivalentes :

- (i) $\sigma(f)$ est polynomial ;
- (ii) f peut être calculée par une formule de profondeur logarithmique ;
- (iii) $f \in \text{NC}^1$.

▷ (i) \implies (ii) : soit ϕ une formule qui calcule f de taille m . Pour chaque m , soit $\delta(m)$ la profondeur nécessaire pour les formules de taille m . On a $\delta(1) = 0$. De plus, chaque formule de taille $m > 1$ a une sous-formule de taille entre $\frac{m}{3}$ et $\frac{2m}{3}$. Soit ψ une telle sous-formule. Pour $i = 0, 1$, soit $\phi^{(i)}$ la formule qu'on obtient en remplaçant ψ par i et simplifiant la formule qui résulte. Notez que $\phi(x) = 1$ si et seulement si $\psi(x) = 0$ et $\phi^{(0)}(x) = 1$, ou $\psi(x) = 1$ et $\phi^{(1)}(x) = 1$. Alors $\phi = (\neg\psi \wedge \phi^{(0)}) \vee (\psi \wedge \phi^{(1)})$. Les tailles de $\neg\psi$, $\phi^{(0)}$ et $\phi^{(1)}$ sont toutes entre $\frac{m}{3}$ et $\frac{2m}{3}$ et donc $\delta(m) \leq \delta(\frac{2m}{3}) + 2$. Il s'ensuit que $\delta(m) \leq 4 \log_2(m)$. Pour $m = 1$, cette formule est encore valable. De plus, $4 \log_2(\frac{2m}{3}) + 2 = 4 \log_2(m) + 4 - 4 \log_2(3) + 2 < 4 \log_2(m)$, car $\log_2(3) > \frac{1}{2}$.

(ii) \implies (iii) : s'il existe un circuit polynomial, en particulier logarithmique, il existe une formule logarithmique.

(iii) \implies (i) : soit $f \in \text{NC}^1$. Alors il existe un circuit de degré d'entrée plus petit que 2 et de profondeur d qui est logarithmique et qui calcule f . On peut *déplier* ce circuit pour obtenir un arbre de profondeur $\leq d$, et donc de taille $\leq 2^d$. Puisque d est logarithmique, 2^d est polynomial. ■

2.8.2 Programmes branchés

Définition. (*Programme branché*)

Soit $n \in \mathbb{N}$. Un *programme branché* de longueur m et de largeur k est une suite de triplets $(i_t, p_t, q_t)_{i \in \llbracket 1, m \rrbracket}$ où chaque $i_t \in \llbracket 1, n \rrbracket$ et chaque p_t et q_t est une application de $\llbracket 1, k \rrbracket$ dans lui-même.

Définition. (*Calcul par un programme*)

Soit (i, p, q) un programme branché. Si $x \in \{0, 1\}^n$, alors la sortie du programme branché (i, p, q) est $r_m \circ r_{m-1} \circ \dots \circ r_2 \circ r_1$ où $r_t = \begin{cases} p_i & \text{si } x_{i_t} = 0 \\ q_t & \text{sinon.} \end{cases}$ Si $F \subseteq \llbracket 1, k \rrbracket^{\llbracket 1, k \rrbracket}$, on dit que

$\phi : \{0, 1\}^n \rightarrow \{0, 1\}$ est (F, m) -calculable s'il existe un programme branché de taille m tel que la sortie est $\in F$ si et seulement si $\phi(x) = 1$.

Si $\alpha : \llbracket 1, k \rrbracket \rightarrow \llbracket 1, k \rrbracket$, on dit que ϕ est (α, m) -calculable s'il y a un problème de taille m tel que la sortie est α si $\phi(x) = 1$ et id si $\phi(x) = 0$.

On montre maintenant un théorème étonnant.

Théorème. (*Théorème de Barrington*)

Soit ϕ une application (pseudo-)booléenne. Si $\sigma(\phi)$ est polynomial, alors il y a un programme branché de taille polynomiale et de largeur 5 qui calcule ϕ .

Lemme

Si ϕ est (α, m) -calculable et $\beta = \gamma\alpha\gamma^{-1}$, alors ϕ est (β, m) -calculable.

▷ Si $(i_t, p_t, q_t)_{i \in \llbracket 1, m \rrbracket}$ un programme branché qui α -calcule ϕ , on remplace p_t, q_t par $p_t \gamma^{-1}, q_t \gamma^{-1}$ et p_m, q_m par $\gamma p_m, \gamma q_m$. Alors la sortie est $\gamma \alpha \gamma^{-1}$ si $\phi(x) = 1$ et $\gamma \gamma^{-1} = id$ si $\phi(x) = 0$. ■

Lemme

Soit α un 5-cycle. Si ϕ est (α, m) -calculable, alors $\neg \phi$ est (α, m) -calculable.

▷ Soit $(i_t, p_t, q_t)_{i \in \llbracket 1, m \rrbracket}$ un programme branché qui α -calcule ϕ . Remplaçons p_1, q_1 par $p_1 \alpha^{-1}, q_1 \alpha^{-1}$. Alors la sortie est id si $\phi(x) = 1$ et α^{-1} si $\phi(x) = 0$. Donc $\neg \phi$ est (α^{-1}, m) -calculable. Mais α, α^{-1} sont des 5-cycles, dont ils sont conjugués, donc par le lemme précédent, ϕ est (α, m) -calculable. ■

Lemme

Il existe deux 5-cycles α, β tels que $\alpha \beta \alpha^{-1} \beta^{-1}$ est également un 5-cycle.

▷ $\alpha = (1, 2, 3, 4, 5)$ et $\beta = (1, 3, 5, 4, 2)$ conviennent. ■

Exercice 7

Expliquer comment retrouver ce résultat.

Lemme

Soient α, β des 5-cycles tels que $\alpha \beta \alpha^{-1} \beta^{-1}$ est un 5-cycle. Alors si ϕ_1 et ϕ_2 sont (α, m) -calculables, alors $\phi_1 \wedge \phi_2$ est $(\alpha, 4m)$ -calculable.

▷ Soit $(i_t, p_t, q_t)_{i \in \llbracket 1, m \rrbracket}$ un programme branché qui α -calcule ϕ_1 et $(j_t, r_t, s_t)_{i \in \llbracket 1, m \rrbracket}$ un programme branché qui β -calcule ϕ_2 en utilisant le lemme précédent. Pour calculer $\phi_1 \wedge \phi_2$, commencer avec l'inverse du programme branché qui calcule ϕ_2 , puis l'inverse de celui qui calcule ϕ_1 , puis le programme qui calcule ϕ_2 , puis celui qui calcule ϕ_1 . La sortie est alors donnée par

$$\begin{cases} \alpha \beta \alpha^{-1} \beta^{-1} & \text{si } \phi_1(x) = \phi_2(x) = 1 \\ \alpha id \alpha^{-1} id = id & \text{si } \phi_1(x) = 1, \phi_2(x) = 0 \\ id \beta id \beta^{-1} = id & \text{si } \phi_1(x) = 0, \phi_2(x) = 1 \\ id id id^{-1} id^{-1} = id & \text{si } \phi_1(x) = 0, \phi_2(x) = 0. \end{cases}$$

Donc $\phi_1 \wedge \phi_2$ est $(\alpha \beta \alpha^{-1} \beta^{-1}, 4m)$ -calculable et puisque $\alpha \beta \alpha^{-1} \beta^{-1}$ est un 5-cycle, $\phi_1 \wedge \phi_2$ est $(\alpha, 4m)$ -calculable. Notez que $\phi_1 \vee \phi_2 = \neg(\neg \phi_1 \wedge \neg \phi_2)$ donc on a aussi que si ϕ_1 et ϕ_2 sont (α, m) -calculables, et alors $\phi_1 \vee \phi_2$ est $(\alpha, 4m)$ -calculable. ■

Preuve.

▷ Par la proposition, ϕ peut être calculée par une formule de profondeur d logarithmique, et donc par les lemmes, il existe un 5-cycle α tel que ϕ est $(\alpha, 4^d)$ -calculable. Puisque d est logarithmique, 4^d est polynomial. ■

2.8.3 Application du théorème de Barrington

Proposition

$$\text{NL} \subseteq \text{AC}_{unif}^1.$$

▷ Soit $\phi \in \text{NL}$ et soit T une machine de Turing non déterministe qui calcule ϕ en espace logarithmique. Pour chaque entrée x , on peut calculer un graphe de configuration quand l'entrée de T est x . Ce graphe G_x est assez clairement ICEL, donc $\phi(x) = 1$ si et seulement si on peut trouver un chemin de V_{init} à $V_{accepte}$ dans G_x . Il suffit donc de montrer que $\text{STCONN} \in \text{AC}_{unif}^1$.

Définissons par $\theta_r(u, v) = 1$ si et seulement si on peut trouver un chemin de u à v de longueur $\leq r$. On voit que $\theta_{2r}(u, v) = \bigvee_w \theta_r(u, w) \wedge \theta_r(w, v)$. Alors si l'on a un circuit de profondeur d qui calcule chaque $\theta_r(u, v)$, on peut créer un circuit de profondeur $d + 2$ qui calcule chaque $\theta_{2r}(u, v)$. La taille est polynomiale. Il existe un chemin de u à v si et seulement si $\theta_m(u, v) = 1$ où m est le nombre de sommets de G_x , qui est polynomial. Donc $\text{STCONN} \in \text{AC}^1$. Le circuit est ICEL, et donc $\text{STCONN} \in \text{AC}_{unif}^1$. ■

On sait que si ϕ est P -complet et $\phi \in L$, alors $P \subseteq L$. Montrons maintenant :

Proposition

$$\text{Si } \phi \text{ est } P\text{-complet et } \phi \in \text{NC}_{unif}^i, \text{ alors } P \subseteq \text{NC}_{unif}^{\max(i, 2)}.$$

▷ Soit $\psi \in P$, alors $\psi = \phi \circ \gamma$ où γ est ICEL. Chaque sortie de γ est dans $L \subseteq \text{NL} \subseteq \text{AC}_{unif}^1 \subseteq \text{NC}_{unif}^2$ et donc peut être calculée par un circuit L -uniforme de profondeur $\mathcal{O}(\log(n)^i)$ et donc $\psi \in \text{NC}_{unif}^{\max(i, 2)}$. ■

2.9 Théorie de la décision

Chapitre 3

Exercices

Difficulté des exercices :

- Question de cours, application directe, exercice purement calculatoire sans réelle difficulté technique
- Exercice faisable, soit intuitivement, soit en employant des moyens rudimentaires ou des techniques déjà vues
- Exercice relativement difficile et dont la résolution appelle à une réflexion plus importante à cause d'obstacles techniques ou conceptuels, qui cependant devraient être à la portée de la plupart des étudiants bien entraînés
- Exercice très exigeant, destiné aux élèves prétendant aux concours les plus difficiles, exercice « classique ».
- La résolution de l'exercice requiert un raisonnement et des connaissances extrêmement avancés, dépassant les attentes du prérequis. Il est presque impossible de le mener à terme sans indication. Bien qu'exigibles à très peu d'endroits, ces exercices sont très intéressants et présentent souvent des résultats forts.

Appendice

Bibliographie

[1] *Titre du livre*, Auteur du livre, date, maison d'édition

Table des figures

2.3.1 <i>Exemple de circuit booléen.</i> —	14
2.7.1 <i>Espaces de complexité imbriqués.</i> —	40

Liste des tableaux