

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

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

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

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.

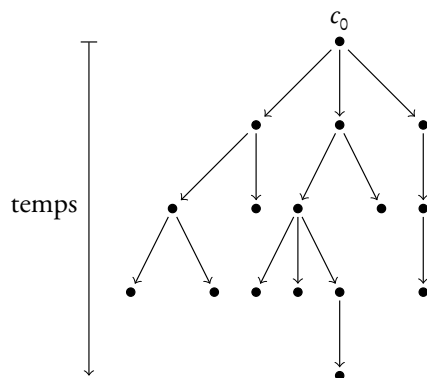


FIGURE 2.2 – Arbre de calcul d'une machine non déterministe.

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 re-

viendrons 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 abrègerons la boucle précédente par une instruction deviner étendue aux mots comme dans l'exemple suivant.

2-AA Exemple On réécritra 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 $U_{\text{non-det}}$ à 6 rubans, d'alphabet d'entrée $\Sigma_U = \{0, 1\}$ et d'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 $\varphi_N : \Sigma_N^* \rightarrow \Sigma_U^*$ tel que pour tout mot $x \in \Sigma_N^*$, $U_{\text{non-det}}(\langle N, \varphi_N(x) \rangle)$ accepte ssi $N(x)$ accepte ;
- il existe une constante α_N telle que pour tout $x \in \Sigma_N^*$, si $N(x)$ s'arrête en temps t , alors $U_{\text{non-det}}(\langle N, \varphi_N(x) \rangle)$ s'arrête en temps $\leq \alpha_N t$ (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' .

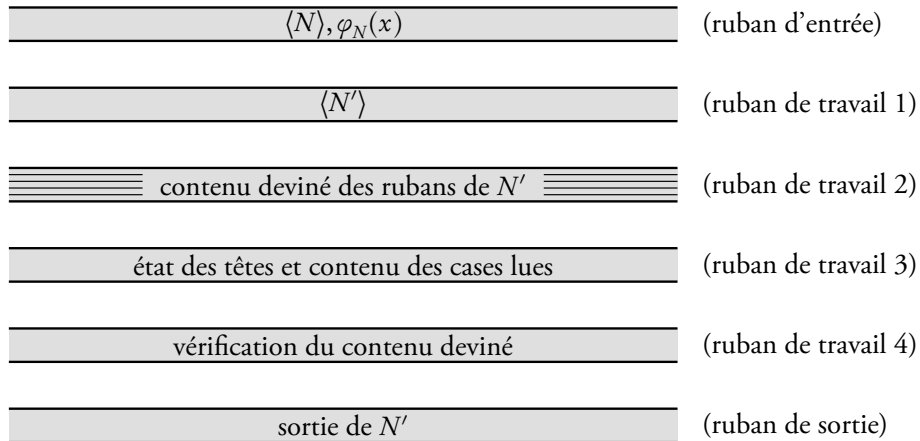


FIGURE 2.3 – Machine universelle non déterministe à 6 rubans simulant $N(x)$.

La fonction des 6 rubans de $U_{\text{non-det}}$ 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 $U_{\text{non-det}}$. 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, $U_{\text{non-det}}$ possède un quatrième ruban de travail qui servira à vérifier que les valeurs devinées étaient correctes.

Voici comment $U_{\text{non-det}}$ simule une étape de N' :

- pour chaque tête d'un ruban de travail de N' , $U_{\text{non-det}}$ 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 $1 \leq i \leq k-2$) :

- 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. \square

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 $t : \mathbb{N} \rightarrow \mathbb{N}$, la classe $\text{NTIME}(t(n))$ 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 $\leq \alpha t(|x|)$ (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 \mathcal{T} est un ensemble de fonctions, alors $\text{NTIME}(\mathcal{T})$ désigne $\cup_{t \in \mathcal{T}} \text{NTIME}(t(n))$.

Les classes NTIME sont closes par union et intersection.

2-AF Proposition

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

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.

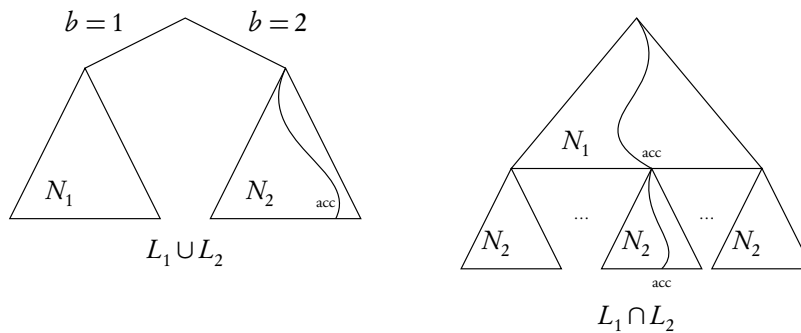


FIGURE 2.4 – Union et intersection pour des machines non déterministes.

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.

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 $O(1) + \max(\alpha_1 t(n), \alpha_2 t(n))$, on en déduit que $L_1 \cup L_2 \in \text{NTIME}(t(n))$.

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 $L_1 \cap L_2 \in \text{NTIME}(t(n))$. \square

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 **faudrait** 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 $t : \mathbb{N} \rightarrow \mathbb{N}$, $\text{DTIME}(t(n)) \subseteq \text{NTIME}(t(n)) \subseteq \text{DTIME}(2^{O(t(n))})$.

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 γ de taille $\alpha t(n)$ faire
 - simuler $N(x)$ sur le chemin γ ,
 - accepter si la simulation accepte ;
- rejeter (ici, tous les chemins ont été rejétés).

Pour γ fixé, le corps de boucle prend un temps déterministe $O(t(n))$. Il reste à compter le nombre de chemins γ à tester. Chaque transition de la machine non déterministe a

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.

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 $R^{xt(n)}$ chemins γ possibles, soit $2^{xt(n)\log R}$. 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 $2^{O(t(n))}$. \square

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 γ 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 γ .

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 $f, g : \mathbb{N} \rightarrow \mathbb{N}$ 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 $\text{NTIME}(f(n)) \subsetneq \text{NTIME}(g(n))$.

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 $L \in \text{NTIME}(g(n)) \setminus \text{NTIME}(f(n))$, 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 :

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

Remarquons tout d'abord que pour n assez grand, $2^{g(n)} \geq 2^{g(n-1)}$ 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 $U_{\text{non-det}}$ (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 $i \geq 2^{g(m)}$ 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 $U_{\text{non-det}}(\langle N \rangle, (\langle N \rangle, 1^{i+1}, x))$ pendant $g(n)$ étapes ($U_{\text{non-det}}$ 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 $i \geq 2^{g(m)}$ 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 cas², V prend un temps $O(g(n))$ donc $L \in \text{NTIME}(g(n))$.

Supposons maintenant que $L \in \text{NTIME}(f(n))$ pour une certaine fonction f vérifiant $f(n+1) = o(g(n))$: 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 $U_{\text{non-det}}$ de la proposition 2-AB. Ainsi, exécuter $U_{\text{non-det}}$ 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,

$$N(\langle N \rangle, \epsilon, x) = N(\langle N \rangle, 1, x)$$

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.

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, lorsque $i = 2^{g(m)}$, 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 :

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

une contradiction. □

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

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

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

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

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 :

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

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 $\leq k$ 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 instantiation 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 instantiation 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 $B \in P$ 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 $B \in P$ 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 $y \in \{0, 1\}^*$ 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 décide 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 $B \in P$. 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 $B \in P$.

\Leftarrow Soit $B \in P$ 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 $A \in NP$. 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 $A \in \text{NP}$.

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) . \square

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

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

Montrer que $\mathcal{C} = \text{DTIME}(2^{2^{O(n)}})$ (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 \mathcal{C} l'ensemble des langages A tels qu'il existe un polynôme $p(n)$ et un langage $B \in \text{EXP}$ vérifiant

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

Montrer que $\mathcal{C} = \text{EXP}$.

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


$$P \subseteq NP \subseteq EXP \subseteq NEXP$$

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

2-AS Corollaire

$$NP \subsetneq NEXP$$

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

 **2-AT Exercice**

| Séparer de même NE et NP.

Pour résumer, nous savons que

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

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

2.2.7 Le problème « $P = NP?$ »

En particulier, la question de savoir si $P = NP$ est ouverte et centrale en complexité. Pour reprendre la caractérisation précédente, cela revient à savoir si trouver une solution est aussi simple³ 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 $P \neq NP$. 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.

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

2-AU Proposition

$$P = NP \implies EXP = NEXP$$

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 $P = NP$. Soit $L \in NEXP$. Il est reconnu par une machine non déterministe N fonctionnant en temps 2^{n^k} pour un certain k . On définit le langage $\tilde{L} = \{(x, 1^{2^{|x|^k}}) \mid x \in L\}$, 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 $m \geq 2^{|x|^k}$.

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 $N(x)$.

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

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, $\tilde{L} \in \text{NP}$: par hypothèse, on en déduit que $\tilde{L} \in \text{P}$. 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 $L \in \text{EXP}$. \square

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 $n \in \mathbb{N}$, $|L^{=n}| \leq p(n)$. Il est dit *unaire* si $L \subseteq 1^*$ (tout mot de L n'est composé que de symboles 1).

Montrer que les trois assertions suivantes sont équivalentes :

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

Indication : montrer $1 \implies 2 \implies 3 \implies 1$. Pour $3 \implies 1$, « compresser » un langage unaire $L \in \text{NP} \setminus \text{P}$ pour obtenir un langage \tilde{L} de NE : si $\text{NE} = \text{E}$ alors $\tilde{L} \in \text{E}$ et on peut s'en servir pour placer L dans P .

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

$$\text{co}\mathcal{C} = \{^c A \mid A \in \mathcal{C}\}.$$

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 « $\text{NP} = \text{coNP}$? » 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 Σ^* 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 $\{(G, k) \mid G \text{ n'a pas de clique de taille } k\}$: on garde ainsi les mêmes entrées et on inverse simplement la question. En effet, on parle du complémentaire dans l'ensemble $\{(G, k) \mid G \text{ est un graphe et } k \text{ un entier}\}$ (l'ensemble des entrées valides pour ce problème) et non dans l'ensemble Σ^* .

Par rapport à prendre le complémentaire dans Σ^* , 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 $B \in P$ tels que

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

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

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

On en déduit que

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

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

On déduit en particulier de cette proposition que $P \subseteq \text{coNP}$. 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.