Introduction à la calculabilité et à la complexité

Cours de M1, premier semestre 2010–2011

Université Paris Diderot – Paris 7
Cours : Sylvain Perifel, TD : Christian Choffrut

1 Introduction

Dans ce cours :
- Formaliser la notion de calcul : qu'est-ce qu'une « méthode effective de calcul » ? Qu'est-ce qu'un *algorithme* ? (peut-on se contenter des automates finis — nos ordinateurs ont un nombre fini d'états — ou des automates à pile p.ex. ?) Un programme en C, en Caml (**est-ce équivalent ?**) ?
- Peut-on tout calculer ? Il y a des limites aux automates finis, aux automates à pile (lemme de l'étoile, etc.) : y en a-t-il à nos ordinateurs ? Existe-t-il un algorithme qui décide si un programme en C affiche toujours « Hello world ! » ?
- Enfin, que peut-on calculer *efficacement* ? Qu'est-ce que ça veut dire ? P. ex., existe-t-il un algorithme « efficace » pour décider si un graphe est 3-coloriable ?

2 Calculabilité

2.1 Algorithmes

« Processus de résolution d'un problème par le calcul », « méthode effective de calcul » — notion intuitive.

2.1.1 Histoire

- Premières traces chez les Babyloniens (actuel Irak), 2ème millénaire avant JC, pour le commerce et les impôts.
- Chez les Grecs : algorithme d'Euclide (3ème siècle avant JC) pour le calcul du pgcd.
- Étymologie : mathématicien perse (actuel Iran) Al Khuwarizmi au 9ème siècle après JC (a aussi donné *algèbre*). Étude systématique des algorithmes.
- Machines et automates des 17ème et 18ème siècles (Pascaline 1642, Leibniz 1673, etc.).
- Algorithmes pour construire des longueurs à la règle et au compas : quels nombres peut-on construire ? (plus petit corps contenant les rationnels et clos par racine carrée, Wantzel 1837)
- Fin 19ème-début 20ème : formalisation des mathématiques (axiomes, systèmes de preuve). Peut-on tout résoudre par algorithme ?
- Hilbert 1900 (10ème problème de Hilbert) : « Trouver un algorithme déterminant si une équation diophantienne a des solutions. »
- Hilbert 1928, Entscheidungsproblem : donner un algorithme capable de décider si un énoncé mathématique est vrai (notion intuitive d'algorithme).
- Années 1930 : Church (λ-calcul), Turing (machine de Turing), etc., proposent des formalisation de la notion d'algorithme. Ils montrent qu'un algorithme pour l'Entscheidungsproblem n'existe pas (contre toute attente).
- Matiyasevich 1970 : pas d'algorithme pour décider si une équation diophantienne a une solution.

2.1.2 Machine de Turing

- Une formalisation de la notion d'algorithme (Turing 1936). Modèle théorique. Sorte d'ancêtre des ordinateurs.
- *Thèse de Church-Turing* : tout « algorithme » au sens intuitif est équivalent à une machine de Turing.

Description informelle d'une machine de Turing (**+ dessin**) : un ruban contient des cases dans lesquelles on peut écrire une lettre (p. ex. 0 ou 1) ou qui peuvent rester vides (imaginer une bande magnétique, une cassette, ou un disque dur). Une tête de lecture/écriture se déplace sur le ruban, pour lire et éventuellement modifier les cases selon un nombre fini de règles. La tête de lecture possède un nombre fini d'états $q_0, \ldots, q_k$ (automate fini). Initialement, le ruban contient la donnée à traiter (écrite en binaire) et la tête de lecture est dans l'état initial q_0. Le contenu du ruban et l'état de la tête de lecture évoluent en fonction des cases lues ou écrites par la tête de lecture.

La tête de lecture correspond grosso-modo au processeur de nos ordinateurs, tandis que le ruban correspond à la mémoire. Mais le ruban d'une machine de Turing est infini.

Dessin + exemples : multiplication d'un nombre binaire par deux, ajouter un à un nombre binaire.

Définition : une machine de Turing est un octuplet $(Q, q_0, q_a, q_r, \Sigma, \Gamma, B, \delta)$ où :
- Q est l'ensemble des états de la tête de lecture : c'est un ensemble fini $\{q_0, q_1, \ldots, q_k\}$;
- $q_0 \in Q$ est l'état initial ;
- $q_a \in Q$ est l'état final d'acceptation, $q_r \in Q$ celui de rejet ;
- Σ est l'alphabet d'entrée : il sert à écrire l'entrée initiale sur le ruban ;
- Γ est l'alphabet de travail ($\Sigma \subset \Gamma$) : ce sont les symboles utilisés sur le ruban lors du calcul ;
- $B \in \Gamma \setminus \Sigma$ est le symbole "blanc", représentant une case vide ;
- $\delta : (Q \setminus \{q_a, q_f\}) \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ est la fonction de transition (le « programme » de la machine), c'est une fonction totale.

Initialement, le ruban contient le mot d'entrée (sur l'alphabet Σ, une lettre par case), toutes les autres cases (à gauche et à droite) contiennent la symbole B et la tête de lecture est positionnée sur la première lettre du mot (la plus à gauche). À chaque étape, la tête de lecture lit le contenu ($\in \Gamma$) de la case sur laquelle elle est et, en fonction de son état ($\in Q$), la fonction de transition δ donne le nouveau symbole ($\in \Gamma$) à écrire dans la case, la tête change d'état ($\in Q$) et se déplace d'une case à gauche, à droite ou reste sur place (L, R ou S). Le ruban étant infini vers la droite et vers la gauche, il n'y a pas de risque « d'aller trop loin ».

La machine s'arrête lorsqu'elle entre dans un état terminal q_a ou q_r. Le résultat du calcul est alors le mot écrit sur le ruban à ce moment-là. Le mot d'entrée est accepté si l'état terminal est q_a, rejeté sinon.

Configuration d'une machine : état, contenu du ruban, position de la tête.

Exemples précédents formels :
- Multiplication par 2 : il s'agit de se déplacer jusqu'à la fin du mot et d'écrire un 0 sur la première case vide (c.-à-d. contenant le symbole B). $Q = \{q_0, q_a\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$ et la fonction de transition : $\delta(q_0, x) = (q_0, x, R)$ pour $x \in \{0, 1\}$, $\delta(q_0, B) = (q_a, 0, R)$.
- Ajouter 1 : il s'agit de se déplacer jusqu'à la fin du mot, d'inverser la dernière lettre, et de revenir vers le début du mot pour propager la retenue. $Q = \{q_0, q_1, q_2, q_c, q_a\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$ et la fonction de transition : $\delta(q_0, x) = (q_0, x, R)$ pour $x \in \{0, 1\}$, $\delta(q_0, B) = (q_1, B, L)$, $\delta(q_1, 1) = (q_r, 0, L)$, $\delta(q_1, 0) = (q_2, 1, L)$, $\delta(q_2, x) = (q_2, x, L)$ pour $x \in \{0, 1\}$, $\delta(q_2, B) = (q_a, B, R)$, $\delta(q_c, 0) = (q_2, 1, L)$, $\delta(q_c, 1) = (q_c, 0, L)$, $\delta(q_c, B) = (q_a, 1, R)$, et n'importe quelle valeur pour le dernier cas $\delta(q_1, B)$ qui n'arrive pas.

Note historique : dans les années 1920, il y a eu d'autres tentatives de capturer la notion intuitive d'algorithme (fonctions récursives primitives) ; on s'est rendu compte par la suite qu'elles ne capturaient pas tout ce qu'on peut calculer par algorithme (p. ex. pas la fonction d'Ackermann).

Exemples sur le simulateur : égalité de deux mots, addition, crible d'Eratosthène, castors affairés (alphabet à 2 symboles dont B ; la meilleure machine à 2 états non terminaux s'arrête en 6 étapes, à 3 états en 21 étapes, à 4 en 107, à 5 en $\geq 47.10^6$, et à 6 en $\geq 2{,}5.10^{2879}$)

Vidéo : machine de Turing en Lego.

2.1.3 Variantes des machines de Turing

Alphabets restreints : on peut se restreindre à $\Sigma' = \{0,1\}$ et $\Gamma' = \{0,1,B\}$. Il suffit de coder les éléments de Γ sur $\{0,1\}$ en prenant $k = \lceil \log_2(|\Gamma|) \rceil$ cases pour chaque lettre et de modifier la fonction de transition en conséquence (lire k cases pour savoir quelle transition faire, éventuellement bouger de k cases). On augmente beaucoup le nombre d'états (pour décoder : un état par préfixe du codage d'une lettre, multiplié par le nombre d'états initial + k états pour se déplacer). On peut aussi se passer du symbole B avec un encodage approprié.

Ruban semi-infini : plutôt qu'un ruban bi-infini, la machine ne dispose que d'un ruban infini à droite. Cases indexées par $\mathbb{N}$ plutôt que $\mathbb{Z}$. Le mot d'entrée est écrit à partir de la première case (numérotée 0). Si la machine souhaite aller à gauche alors qu'elle est sur la première case, elle effectue son changement d'état mais reste sur place.

Ce modèle est aussi puissant que celui du ruban bi-infini : pour simuler une machine à ruban bi-infini par une machine à ruban semi-infini, il suffit de "plier" le ruban bi-infini pour avoir dans la nouvelle case i le contenu de la case i et celui de la case $-i$. Il faut augmenter l'alphabet de travail : pour chaque paire de lettres de départ on crée une nouvelle lettre. Il faut aussi un symbole spécial # pour voir qu'on est sur la case 0 (pour ne pas aller à gauche). On change la fonction de transition de manière appropriée. Enfin, il faut multiplier par deux l'ensemble des états pour savoir si on est à gauche ou à droite du 0.

a_0	a_1	a_2	$\dots$	
#	a_{-1}	a_{-2}	$\dots$	

On peut aussi écrire une case sur deux la partie gauche et une case sur deux la partie droite.

Deux rubans ou plus : plusieurs têtes de lecture. L'état change en fonction de ce que lisent toutes les têtes. C'est encore équivalent au modèle à un seul ruban. Pour simuler une machine à plusieurs rubans par une machine à un ruban : on agrandit l'alphabet pour pouvoir coder tous les rubans sur un seul. On ajoute un caractère spécial par tête pour mémoriser sa position. Grâce à des états spéciaux, on parcourt toutes les têtes (deux fois) pour pouvoir effectuer la transition.

Exemple : utiliser le ruban de travail pour gérer un compteur afin de déterminer la longueur du mot en entrée, ou trouver la lettre du milieu, etc.

2.2 Propriétés des machines de Turing

On peut **simuler** le fonctionnement d'une machine de Turing par une autre, c'est-à-dire réutiliser un bout de code existant. Attention, si la première ne s'arrête pas, la seconde non plus. On notera $M \equiv N$ si pour tout x, soit ni $M(x)$ ni $N(x)$ ne s'arrêtent, soit elles s'arrêtent toutes deux et $M(x) = N(x)$.

Composition : si M et M' sont deux machines de Turing, il existe une machine N qui calcule leur composition $M \circ M'$, c'est-à-dire qui, sur l'entrée x, renvoie la valeur $M(M'(x))$ si les deux machines s'arrêtent, et ne s'arrête pas sinon.

Manipulation des programmes (machines de Turing) eux-mêmes : par exemple, est-ce que ce programme affiche bien "Hello world !" ? → codage d'une machine de Turing

Il suffit de coder l'octuplet $(Q, q_0, q_a, q_r, \Sigma, \Gamma, B, \delta)$: pour l'ensemble des états Q, on peut considérer que c'est $\{0,1,\dots,|Q|-1\}$, donc il suffit de donner $|Q|$ (entier en binaire) ; on peut considérer que $q_0 = 0$, $q_a = |Q| - 2$ et $q_r = |Q| - 1$ donc on n'a pas besoin de les donner dans le codage ; on peut considérer que $\Sigma = \{0,1,\dots,|\Sigma| - 1\}$ donc il suffit de donner $|\Sigma|$ (entier en binaire) ; on peut considérer que $\Gamma = \{0,1,\dots,|\Gamma| - 1\}$ donc il suffit de donner $|\Gamma|$ (entier en binaire) ; on peut considérer que $B = |\Gamma| - 1$ donc on n'a pas besoin de le donner dans le codage ; enfin, il faut coder la fonction de transition : il suffit de coder l'ensemble des transitions séparées par un # (pour chaque transition : numéro de l'état de départ, lettre lue, numéro de l'état d'arrivée, lettre écrite, direction de déplacement).

Si M est une machine de Turing, on notera $\langle M \rangle$ son codage. Si x est un mauvais code de machine, on prendra la convention que x encode la machine qui rejette toujours. Ainsi, tout mot x correspond au code $\langle M \rangle$ d'une machine M. Pour tout mot x, on notera $\mathcal{M}_x$ la machine codée par x.

Machine universelle : il existe une machine de Turing $\mathcal{U}$ qui, sur l'entrée $\langle M, x \rangle$ composée d'un codage d'une MT M et d'un mot x, simule le fonctionnement de $M(x)$. En d'autres termes, si $M(x)$

s'arrête, $U(\langle M, x\rangle)$ s'arrête et renvoie la même valeur ; si $M(x)$ ne s'arrête pas, $U(\langle M, x\rangle)$ ne s'arrête pas non plus. Une telle machine $\mathcal{U}$ est appelée **machine universelle**.

Principe de fonctionnement de $\mathcal{U}$: pour simuler une MT M à un ruban, $\mathcal{U}$ a trois rubans, un ruban d'entrée, un de travail/sortie et un pour stocker l'état de M. Au départ, sur le ruban de travail on recopie x et on positionne la tête de lecture sur le début du mot, et sur le ruban d'état on écrit q_0. Pour chaque transition de M, on lit la lettre sous la tête de lecture sur le ruban de travail, on lit l'état sur le ruban d'état, on va chercher la transition correspondant dans le codage de M sur le ruban d'entrée, on effectue cette transition sur le ruban de travail et on change l'état sur le ruban d'état.

Vocabulaire : une fonction est dite *calculable* (ou *récursive*) si elle est totale et calculée par une machine de Turing. NB : selon certaines définition, une fonction calculable peut être partielle. Pour éviter la confusion, on précisera donc à chaque fois si la fonction est partielle ou totale.

Théorème s-m-n (ou théorème d'itération, Kleene 1943) : il existe une fonction calculable (totale) s qui prend en entrée un code de machine $\langle M\rangle$ et un mot x et telle que pour tout y, $\mathcal{M}_{s(\langle M\rangle,x)}(y) \equiv M(x, y)$.

Preuve. $s(\langle M\rangle, x)$ est le code de la machine N qui écrit d'abord $x\#$ sur son ruban de travail avant l'entrée y, et qui simule le fonctionnement de M. La fonction s est clairement calculable et totale.

Théorème de récursion (Kleene, 1938) : soit f une fonction calculable (totale). Alors il existe un code de machine m tel que $\mathcal{M}_{f(m)} \equiv \mathcal{M}_m$ (point fixe).

Preuve. Soit N la machine suivante : $N(x, y)$ simule $\mathcal{M}_{f(s(x,x))}(y)$. Pour cela, elle calcule $f(s(x, x))$ (composition de deux fonctions calculables) puis utilise la machine universelle pour exécuter sur l'entrée y le code obtenu. On appelle a le code de N (c.-à-d. $N \equiv \mathcal{M}_a$). Par définition de N, on a $N(a, y) \equiv \mathcal{M}_{f(s(a,a))}(y)$. D'autre part, par le théorème s-m-n, on a $N(x, y) \equiv \mathcal{M}_{s(a,x)}(y)$. Ainsi, $N(a, y) \equiv \mathcal{M}_{f(s(a,a))}(y) \equiv \mathcal{M}_{s(a,a)}(y)$ et notre point fixe m est $s(a, a)$.

Applications Calcul récursif (factorielle) : soit f la fonction qui prend en entrée le code $\langle N\rangle$ d'une machine N et qui renvoie le code de la machine N' suivante : $N'(0) = 1$ et $N'(n) = n.N(n - 1)$. Alors il existe un point fixe m (code d'une machine M), c'est-à-dire $\mathcal{M}_m \equiv \mathcal{M}_{f(m)}$: en d'autres termes, $M(0) = 1$ et $M(n) = n.M(n - 1)$, donc M calcule $n!$.

Quines : soit f la fonction qui prend $\langle M\rangle$ en entrée et qui renvoie le code d'une machine N qui écrit $\langle M\rangle$ sur son ruban. Par le théorème, il existe un point fixe m, c'est-à-dire une machine M telle que $M \equiv \mathcal{M}_{f(\langle M\rangle)}$. En d'autres termes, M écrit son propre code sur son ruban. En suivant la preuve du théorème de récursion, on peut écrire un Quine en C :

- $s(x, x)$ est le code de la machine $P()$ qui simule $\mathcal{M}_x(x)$ (on suppose que g est le nom de la fonction du programme x)
  ```
  #define A(y) y main(){g(#y);}
  A(x)
  ```
  (la commande `#y` du préprocesseur C signifie “mettre le contenu de `y` entre guillemets”)
- a est le code de la machine $N(.)$ qui sur l'entrée x simule $\mathcal{M}_{f(s(x,x))}$, c'est-à-dire qui écrit $s(x, x)$
  ```
  void g(char *x){printf("#define A(y) y main(){g(#y);}\nA(%s)\n",x);}
  ```
- le point fixe est $s(a, a)$:
  ```
  #define A(y) y main(){g(#y);}
  A(void g(char *x){printf("#define A(y) y main(){g(#y);}\nA(%s)\n",x);})
  ```
- en ajoutant l'entête (stdio.h) on obtient le programme complet suivant :
  ```
  #include<stdio.h>
  #define A(y) y main(){g(#y);}
  A(void g(char *x){printf("#include<stdio.h>\n#define A(y) y main(){g(#y);}\nA(%s)\n",x);})
  ```

2.3 Machines RAM

Vidéoprojeter les instructions et l'exemple

« Random Access Machine » (Minsky 1961, Melzak 1961, Cook and Reckhow 1973) : plus proche de nos ordinateurs, mais équivalente aux machines de Turing, elle manipule directement des entiers (chaque case contient un entier). Elle contient :

- un ruban d'entrée en lecture seule de la gauche vers la droite, muni d'une tête de lecture ;

- un ruban de sortie en écriture seule de la gauche vers la droite, muni d'une tête d'écriture ;
- un nombre arbitraire de *registres* pouvant contenir chacun un entier, appelés $r_0, r_1, \ldots, r_n, \ldots$;
- l'arithmétique s'effectue dans le registre r_0 appelé *accumulateur* ;
- le programme de la machine (une suite finie d'instructions) ;
- le *compteur ordinal*, contenant le numéro de l'instruction en cours et initialisé à 1 : il est incrémenté après chaque instruction, sauf s'il s'agit d'une rupture de séquence auquel cas il est modifié en conséquence.

Notations : $\mathsf{val}(r_i)$ désigne le contenu du registre r_i, CO désigne le compteur ordinal. Liste d'instructions possibles :

- `ChargerVal`(x) : l'accumulateur r_0 reçoit la valeur x ;
- `ChargerReg`(i) : r_0 reçoit $\mathsf{val}(r_i)$;
- `ChargerInd`(i) : r_0 reçoit $\mathsf{val}(r_{\mathsf{val}(r_i)})$;
- `RangerReg`(i) : r_i reçoit $\mathsf{val}(r_0)$;
- `RangerInd`(i) : $r_{\mathsf{val}(r_i)}$ reçoit $\mathsf{val}(r_0)$;
- `Incrémenter` : r_0 reçoit $\mathsf{val}(r_0) + 1$;
- `Décrémenter` : r_0 reçoit $\mathsf{val}(r_0) - 1$;
- `Ajouter`(i) : r_0 reçoit $\mathsf{val}(r_0) + \mathsf{val}(r_i)$;
- `Soustraire`(i) : r_0 reçoit $\mathsf{val}(r_0) - \mathsf{val}(r_i)$;
- `Multiplier`(i) : r_0 reçoit $\mathsf{val}(r_0) \times \mathsf{val}(r_i)$;
- `Diviser`(i) : r_0 reçoit $\mathsf{val}(r_0)/\mathsf{val}(r_i)$;
- `Saut`(n) : le compteur ordinal CO reçoit la valeur n ;
- `SautSiPositif`(n) : CO reçoit n si $\mathsf{val}(r_0) \geq 0$, CO+1 sinon ;
- `SautSiNul`(n) : CO reçoit n si $\mathsf{val}(r_0) = 0$, CO+1 sinon ;
- `Lire`(i) : r_i reçoit l'entier qui est dans la case sous la tête de lecture, qui se déplace d'un cran à droite ;
- `Écrire`(i) : la tête d'écriture écrit $\mathsf{val}(r_i)$ dans la case pointée et se déplace d'un cran à droite ;
- `Arrêt` : fin de l'exécution.

Exemple de programme pour $n!$ (l'entrée n est sur la première case du ruban d'entrée, r_1 stocke $n - i$, r_2 stocke $n(n-1)\ldots(n-i)$, r_0 fait les calculs) :

1. `Lire`(1) // r_1 contient n
2. `Soustraire`(1) // r_0 contient $-n$
3. `SautSiPositif`(15) // arrêt si $n \leq 0$
4. `ChargerVal`(0) // r_0 contient 0
5. `Ajouter`(1) // r_0 contient n (ainsi que r_1)
6. `RangerReg`(2) // début boucle, r_0 et r_2 contiennent $n(n-1)\ldots(n-i)$, r_1 contient $n-i$
7. `ChargerReg`(1) // r_0 contient $n-i$
8. `Décrémenter` // r_0 contient $n-i-1$
9. `SautSiNul`(14) // on a fini le calcul ($i = n$) : sortir de la boucle
10. `RangerReg`(1) // r_1 contient $n-i-1$
11. `ChargerReg`(2) // r_0 contient $n(n-1)\ldots(n-i)$
12. `Multiplier`(1) // r_0 contient $n(n-1)\ldots(n-i-1)$
13. `Saut`(6) // fin de la boucle
14. `Écrire`(2) // on écrit $n!$ (contenu dans r_2)
15. `Arrêt`

Restriction de l'ensemble d'instructions : on peut se passer de `Saut` (en utilisant `SautSiNul` après avoir mis 0 dans r_0) et de `SautSiPositif` (il s'agit de se ramener à `SautSiNul` en exécutant en parallèle : incrémentations successives, et décrémentations successives, on arrête quand l'un des deux devient nul). On peut se passer de `Diviser`(i) (en incrémentant x jusqu'à ce que $\mathsf{val}(r_i)x > \mathsf{val}(r_0)$), de `Multiplier`$(i)$ (en ajoutant $\mathsf{val}(r_i)$ fois $\mathsf{val}(r_0)$), de `Ajouter` (en incrémentant), de `Soustraire` (en

décrémentant). Il nous reste donc le jeu d'instructions suivant : `ChargerVal`, `ChargerReg`, `ChargerInd`, `RangerReg`, `RangerInd`, `Incrémenter`, `Décrémenter`, `SautSiNul`, `Lire`, `Écrire`, `Arrêt`.

Simulation d'une machine de Turing On simule une machine de Turing à un ruban semi-infini. La valeur de la case i est stockée dans le registre r_{i+2}. Le registre r_1 contient le numéro de la case sur laquelle est la tête de lecture. À chaque état de la machine de Turing et à chaque lettre lue, on associe un bout de programme qui permet de faire l'opération : par exemple, s'il s'agit d'écrire 0 et de se déplacer à droite, on aura `ChargerVal`(0), `RangerInd`(1), `ChargerReg`(1), `Incrémenter`, `RangerReg`(1) ; puis on aura une instruction de branchement pour aller au bon endroit du programme selon le nouvel état donné par la fonction de transition de la machine de Turing.

Simulation par une machine de Turing Un ruban d'entrée, un ruban de sortie, un ruban de travail et un ruban de compteur. On écrit dans l'ordre en binaire sur le ruban de travail de la machine de Turing les entiers stockés dans les registres $r_0, r_1, \ldots$ On utilise un symbole spécial # pour délimiter les registres. Pour exécuter chaque ligne du programme on a un état initial et un état final correspondant ; la fonction de transition suit l'ordre du programme. Simulation des instructions :

- `ChargerVal`(x) : on écrit x en binaire à la place de l'ancienne valeur de r_0.
- `ChargerReg`(i) : grâce à un compteur sur le ruban de compteur, on compte i symboles # pour aller chercher le dernier chiffre du registre r_i (on met un symbole spécial pour retenir où on en est) et on revient à r_0 pour écrire ce chiffre ; on fait pareil pour l'avant-dernier chiffre, etc.
- `ChargerInd`(i) : on va chercher la valeur du registre r_i et on exécute `ChargerReg`(val(r_i)).
- `RangerReg`(i) : idem que `ChargerReg`(i) mais dans l'autre sens ; mais il se peut qu'on doive tout décaler à droite pour insérer chaque lettre.
- `RangerInd`(i) : on va chercher la valeur du registre r_i et on appelle `RangerReg`(val(r_i)).
- `Incrémenter`/`Décrémenter` : il suffit d'ajouter ou de retrancher 1 à r_0.
- `SautSiNul`(n) : on teste si val(r_0) $= 0$ et on change d'état en conséquence.
- `Lire`/`Écrire` : il s'agit de recopier des symboles dans le bon registre.
- `Arrêt` : on nettoie le ruban (sauf la sortie) et on entre dans un état terminal.

2.3.1 Thèse de Church-Turing

Ainsi, toutes les variantes des machines de Turing ci-dessus, ainsi que toutes les variantes des machines RAM ci-dessus, sont équivalentes. Aucun autre modèle de calcul « réaliste » défini à ce jour ne permet de calculer plus de fonctions. Cela plaide en faveur de la **thèse de Church-Turing** :

Les problèmes résolus par une « procédure effective »
sont ceux résolus par une machine de Turing.

Ou encore : « Toute fonction mécaniquement calculable est Turing-calculable. »

Voici donc notre définition d'algorithme : les machines de Turing.

2.4 Langages

Notion intuitive de problèmes :

- entrée (instance)
- question
- réponse oui ou non (problème de décision)

Exemple 1 : décider si un entier est premier :

- entrée : un entier n
- question : n est-il premier ?

Exemple 2 : décider si un graphe est planaire :

- entrée : un graphe $G = (V, E)$
- question : G est-il planaire ?

Exemple 3 : décider si un polynôme a une racine entière :

- entrée : un polynôme p
- question : p a-t-il une racine entière ?

Formalisation : *langage* = ensemble de mots sur un alphabet fini.

Exemple : reconnaissance du langage $L = \{a^{2^n} \ : \ n \in \mathbb{N}\}$ par machine de Turing. Principe : on barre la moitié des a à chaque passage et on vérifie que ça tombe juste. Exercice à faire en cours : décrire complètement la machine.
Avec deux rubans, on peut aussi compter la longueur du mot d'entrée et vérifier qu'elle est de la forme 10^* en binaire.

Langage associé à un problème : ensemble des instances positives, selon un codage approprié.

Exemples de codages usuels : binaire pour les entiers (alphabet $\{0,1\}$), matrice d'adjacence pour les graphes, ensemble des coefficients en binaire pour les matrices à coefficients entiers, suite des coefficients en binaire pour les polynômes à coefficients entiers, etc. Exemple matrice creuse (ou polynôme creux) : ne représenter que les coefficients non nuls. Codage d'un uple $(a_1, \ldots, a_k)$, noté $\langle a_1, \ldots, a_k \rangle$, par exemple en séparant les a_i par des séparateurs #.

Exemples de codages inhabituels : unaire pour les entiers, ensemble de valeurs prises par un polynôme, etc.

Exemples de langages correspondant aux problèmes précédents :
- $\{n \ : \ n \text{ est premier}\}$ où n est codé en binaire ($n \in \{0,1\}^*$) ;
- $\{G \ : \ G \text{ est planaire}\}$ où G est codé par matrice d'adjacence ($\langle a_{i,j} \rangle_{1 \leq i,j \leq k}$) ;
- $\{p \ : \ p \text{ a une racine entière}\}$ où p est codé par la liste de ses coefficients en binaire ($\langle a_0, a_1, \ldots, a_k \rangle$) ;

Taille de l'entrée : taille de son codage. Exemples
- pour un entier n : taille $\lceil \log_2 n \rceil$
- pour un graphe G : taille $|V|^2 + |V| - 1$ (en comptant les séparateurs #)
- pour un polynôme p : taille = degré $\times$ taille max des coefficients (plus les séprateurs)

Énumération des mots finis : on peut numéroter les mots finis (injection de Σ^* dans $\mathbb{N}$) par longueur croissante, puis pour chaque longueur par ordre lexicographique : ainsi sur $\Sigma = \{a, b\}$, ϵ a le numéro 0, a le numéro 1, b le 2, aa le 3, ab le 4, ba le 5, bb le 6, aaa le 7, etc.

Vocabulaire : un langage L est dit *reconnu* par une MT M si M s'arrête sur toute entrée et accepte exactement les mots de L. On dira que L est *accepté* par M si M accepte exactement les mots de L mais ne s'arrête pas forcément sur toute entrée.

2.5 Langages récursifs

Définition : un langage L est décidable (ou récursif) s'il est reconnu par une MT (c.-à-d. qu'il est égal à l'ensemble des mots acceptés par une MT qui s'arrête sur toute entrée). En d'autres termes, il existe une MT qui accepte tout mot de L et qui rejette tout mot hors de L (elle s'arrête toujours).
→ notion intuitive d'algorithme pour un problème

Exemples :
- pour $L = \{1^p \ : \ p \text{ premier}\}$, la MT teste tous les entiers de 2 à $p-1$ (ou $\sqrt{p}$) : si l'un d'entre eux divise p alors elle rejette, sinon elle accepte ;
- pour $L = \{G \ : \ G \text{ est 3-coloriable}\}$, la MT énumère chaque possibilité de 3-colorations et vérifie qu'elle est compatible : si c'est le cas, elle accepte, sinon elle rejette.

Propriété : l'ensemble des langages récursifs est clos par intersection, union, complémentaire

Preuve : si on a deux langages L_1 et L_2 décidés par M_1 et M_2 : pour décider si $x \in L_1 \cap L_2$ (resp. $x \in L_1 \cup L_2$), on simule $M_1(x)$ et $M_2(x)$ et on accepte ssi les deux acceptent (resp. l'une des deux accepte) ; pour décider si $x \in {}^cL_1$, on simule $M_1(x)$ et on accepte ssi $M_1(x)$ rejette.

Définition : un ensemble E est dit dénombrable s'il est en bijection avec $\mathbb{N}$.

Proposition : il existe un langage non récursif.

Preuve : l'ensemble des langages est indénombrable (surjection sur $\mathbb{R}$: faire preuve Cantor ; suite infinie de 0 et 1 ; preuve par procédé diagonal si besoin = si énumération L_i, on construit $L = \{x_i \ : \ x_i \notin L_i\}$) tandis que l'ensemble des MT est dénombrable (injection dans les mots finis).

Problème de l'arrêt (halting problem) : $H = \{\langle M, x\rangle \ : \ M \text{ s'arrête sur } x\}$

Proposition : le problème de l'arrêt est indécidable.

Preuve : supposons que M_H décide H. Soit N la machine suivante : sur l'entrée $\langle M\rangle$, elle simule $M_H(\langle M, M\rangle)$ et si M_H rejette, elle rejette, si M_H accepte elle part dans une boucle infinie. Ainsi, si $N(\langle N\rangle)$ s'arrête, alors $M_H(\langle N, N\rangle)$ rejette donc $N(\langle N\rangle)$ ne s'arrête pas ; si $N(\langle N\rangle)$ ne s'arrête pas, alors $M_H(\langle N, N\rangle)$ accepte donc $N(\langle N\rangle)$ s'arrête. Contradiction.

Exercice : montrer que les variantes suivantes du problème de l'arrêt restent indécidables :
- avec deux entrées x, y : $H_1 = \{\langle M, x, y\rangle \ : \ M(x, y) \text{ s'arrête}\}$;
- sans entrée (entrée vide) : $H_2 = \{\langle M\rangle \ : \ M(\epsilon) \text{ s'arrête}\}$;
- sur M elle-même : $H_3 = \{\langle M\rangle \ : \ M(\langle M\rangle) \text{ s'arrête}\}$.

2.6 Langages récursivement énumérables

Définition : un langage L est récursivement énumérable (ou semi-décidable) s'il est accepté par une MT (c.-à-d. qu'il est égal à l'ensemble des mots acceptés par une MT). En d'autres termes, il existe une MT qui accepte tout mot de L et qui, sur $x \notin L$, soit rejette soit ne s'arrête pas.

Exemples :
- pour $L = \{p \text{ polynôme} \ : \ p \text{ a une racine entière}\}$, la MT énumère tous les entiers i et $-i$ (en partant de 0) et accepte si elle trouve une racine : si p n'a pas de racine entière, elle ne s'arrête pas (note : ici, p a une seule variable donc ce langage est quand même décidable car on a une borne sur les racines ; ce ne serait plus le cas si p avait plusieurs variables) ;
- tout langage récursif est récursivement énumérable ;
- pour $H = \{\langle M, x\rangle \ : \ M(x) \text{ s'arrête}\}$, la MT simule $M(x)$ et accepte si elle s'arrête (et ne s'arrête pas sinon). On en déduit :

Proposition : il existe des langages récursivement énumérables mais pas récursif (p. ex. H).

Propriété : l'ensemble des langages récursivement énumérables est clos par union et intersection.

Preuve : soient $L_1 = \{x \ : \ M_1(x) \text{ accepte}\}$ et $L_2 = \{x \ : \ M_2(x) \text{ accepte}\}$. Soit $M_\cap$ la machine suivante : sur l'entrée x, elle simule « en parallèle » $M_1(x)$ et $M_2(x)$ (c'est-à-dire qu'elle simule une fois sur deux une étape de l'une et une fois sur deux une étape de l'autre) et qui accepte ssi les deux calculs acceptent (sinon elle ne s'arrête pas). Alors $L_1 \cap L_2 = \{x \ : \ M_\cap(x) \text{ accepte}\}$. (note : pour l'intersection, on peut aussi simuler une machine puis l'autre)
De même, $M_\cup(x)$ accepte ssi l'un des deux calculs accepte (le premier qui termine). Alors $L_1 \cup L_2 = \{x \ : \ M_\cup(x) \text{ accepte}\}$.

Proposition : si un langage L et son complémentaire ${}^cL$ sont récursivement énumérables, alors L est récursif.

Preuve : soit M_1 une machine qui énumère L et M_2 qui énumère ${}^cL$. On définit la machine M suivante : sur l'entrée x, elle simule en parallèle $M_1(x)$ et $M_2(x)$. L'une d'entre elles doit accepter puisque x est soit dans L soit dans ${}^cL$. Si M_1 accepte, alors M accepte x, si M_2 accepte, alors M rejette x : la machine M s'arrête toujours et décide le langage L.

Proposition : un langage L est récursivement énumérable ssi il existe une machine M qui énumère exactement l'ensemble des mots de L (c'est-à-dire qu'elle écrit sur son ruban de sortie les mots de L les uns après les autres séparés par #, mais pas forcément dans l'ordre lexicographique).

Preuve : $\Leftarrow$ il suffit de construire une machine qui, sur l'entrée x, simule M tant que x n'apparaît pas sur le ruban.

$\Rightarrow$ soit N la machine qui accepte exactement les mots de L. À l'étape i, la machine M simule N sur l'ensemble des mots de taille $\leq i$ pendant i pas de calculs. Si N accepte x, on écrit x sur le ruban s'il n'avait pas déjà été écrit. Si un mot est accepté par N, alors il sera énuméré par M.

Proposition : il existe un langage non récursivement énumérable.

Preuve : même argument de cardinalité qu'avant.
Ou alors explicitement : soit $L = \{\langle M\rangle \ : \ M \text{ n'accepte pas } \langle M\rangle\}$ (c.-à-d. soit $M(\langle M\rangle)$ s'arrête et rejette, soit $M(\langle M\rangle)$ ne s'arrête pas). Si N semi-décide L, alors si $\langle N\rangle \in L$, N n'accepte pas $\langle N\rangle$ alors qu'elle devrait l'accepter, tandis que si $\langle N\rangle \notin L$, alors $N(\langle N\rangle)$ s'arrête et accepte alors qu'elle ne devrait pas l'accepter : contradiction.

2.7 Réductions

Pour montrer qu'un problème est indécidable, il suffit de montrer qu'il est aussi difficile que le problème de l'arrêt. On peut formaliser cela à l'aide de la notion de réduction.

Définition : un langage A se réduit à un langage B, noté $A \leq_m B$, s'il existe une fonction calculable (totale) $f : \Sigma^* \to \Sigma^*$ telle que $x \in A \iff f(x) \in B$. En d'autres termes, pour décider si $x \in A$, il suffit de calculer $f(x)$ et de décider si $f(x) \in B$. La fonction f s'appelle réduction de A à B.

Proposition : si B est décidable et $A \leq_m B$, alors A est décidable.

Preuve : si M décide B, pour décider si $x \in A$ il suffit de calculer $f(x)$ et de décider si $f(x) \in B$ grâce à M.

Corollaire : si A est indécidable et $A \leq_m B$, alors B est indécidable.

Problème de l'acceptation : soit $A = \{\langle M, x\rangle \ : \ M(x) \text{ accepte}\}$ (c'est-à-dire que $M(x)$ s'arrête et accepte). Alors A est indécidable.

Preuve : montrons que $H \leq_m A$. Soit f la réduction suivante : $f(\langle M, x\rangle) = \langle N, x\rangle$ où N est la machine qui simule M, et si M s'arrête N accepte. Ainsi, $\langle M, x\rangle \in H \iff \langle N, x\rangle \in A$, ce qui montre $H \leq_m A$ et donc que A est indécidable.

Problème de l'égalité : on note $L(M)$ l'ensemble des mots acceptés par une machine M. Soit $E = \{\langle M_1, M_2\rangle \ : \ L(M_1) = L(M_2)\}$. Alors E est indécidable.

Preuve : montrons que $H \leq_m E$. Il s'agit de construire à partir de $\langle M, x\rangle$ deux machines M_1, M_2 qui ont le même langage ssi $M(x)$ s'arrête. Soit M_1 la machine qui accepte tous les mots (donc $L(M_1) = \Sigma^*$). Soit $M_2(y)$ la machine qui simule $M(x)$ puis qui accepte si la simulation termine. On définit la réduction $f(\langle M, x\rangle) = \langle M_1, M_2\rangle$. Alors $L(M_1) = L(M_2)$ ssi $L(M_2) = \Sigma^*$ ssi $M(x)$ s'arrête, donc $\langle M, x\rangle \in H \iff f(\langle M, x\rangle) \in E$ ce qui montre que $H \leq_m E$ et donc que E est indécidable.

Remarque : pour montrer l'indécidabilité d'un problème, on ne part pas toujours du problème de l'arrêt : on peut choisir n'importe quel problème dont on a déjà montré qu'il était indécidable.

2.8 Théorème de Rice

Soit P une propriété quelconque sur les MT, par exemple « la machine M s'arrête sur ϵ » ou « la machine M calcule la factorielle ». Il s'agit d'une propriété sur le comportement des machines et pas sur leur code, donc si $M \equiv N$, on a $M \in P \iff N \in P$. On note L_P le langage $\{\langle M\rangle \ : \ M \text{ a la propriété } P\}$.

Théorème (Rice, 1951) : si P n'est pas triviale (c.-à-d. il existe $M \notin P$ et $N \in P$), alors L_P est indécidable.

Preuve : soit M_0 une machine qui ne s'arrête sur aucune entrée. Si $M_0 \in P$, on considère le complémentaire de P, ce qui ne change rien puisque L_P est décidable ssi ${}^cL_P$ l'est. On peut donc supposer que $M_0 \notin L_P$. Soit $M_1 \in L_P$ une machine quelconque. Montrons que $H \leq_m L_P$.

Soit $\langle M, x\rangle$ une instance de H. On définit la réduction $f(\langle M, x\rangle) = \langle N\rangle$ où N est la machine suivante : sur l'entrée y, elle simule $M(x)$ puis, si la première simulation termine, simule $M_1(y)$ et renvoie la même valeur si elle termine. La fonction f est calculable (totale).

Si $\langle M, x\rangle \in H$ alors $M(x)$ s'arrête donc $N \equiv M_1$ donc $f(\langle M, x\rangle) \in L_P$. Si $\langle M, x\rangle \notin H$ alors $M(x)$ ne s'arrête pas donc $N \equiv M_0$ donc $f(\langle M, x\rangle) \notin L_P$.

Ainsi, f est une réduction de H à L_P : $H \leq_m L_P$ donc L_P est indécidable.

Remarque : c'est bien sûr faux avec récursivement énumérable à la place de décidable puisque l'arrêt est une propriété non triviale mais le langage associé est récursivement énumérable.

2.9 Indécidabilité du problème de correspondance de Post

Jouons aux dominos – Post 1946.

Définition Problème de correspondance de Post (PCP) :

– Entrée : une liste de mots $u_1, \ldots, u_n$ et $v_1, \ldots, v_n$ dans Σ^*

– Question : existe-t-il une suite $i_1, i_2, \ldots, i_k \in [1, n]$ telle que $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$?

Exemple 1 : $u_1 = a$, $u_2 = ab$, $u_3 = bba$ et $v_1 = baa$, $v_2 = aa$, $v_3 = bb$

a	ab	bba
baa	aa	bb

Une solution : 3, 2, 3, 1 car $u_3u_2u_3u_1 = bbaabbbaa = v_3v_2v_3v_1$

bba	ab	bba	a
bb	aa	bb	baa

Exemple 2 : $u_1 = aa$, $u_2 = ab$ et $v_1 = ba$, $v_2 = a$ n'a pas de solution car on ne peut pas utiliser seulement u_1 et v_1 donc le mot du dessus sera toujours plus court que le mot du dessous.

Définition Problème de correspondance de Post modifié (PCPM) : on doit mettre le domino 1 en premier et ne pas le réutiliser ensuite.
- Entrée : une liste de mots $u_1, \ldots, u_n$ et $v_1, \ldots, v_n$ dans Σ^*
- Question : existe-t-il une suite $i_2, i_3, \ldots, i_k \in [2, n]$ telle que $u_1u_{i_2}u_{i_3}\cdots u_{i_k} = v_1v_{i_2}v_{i_3}\cdots v_{i_k}$?

Proposition : PCPM$\leq_m$PCP

Preuve : on transforme une instance de PCPM $\bar{u}, \bar{v}$ en une instance équivalente de PCP $\bar{u}', \bar{v}'$. Soit @ un nouveau symbole. Soient $\phi : \Sigma^* \to \Sigma^*$ et $\psi : \Sigma^* \to \Sigma^*$ les fonctions suivantes : $\phi(x_1 \ldots x_n) = @x_1@x_2@\ldots@x_n$ et $\psi(x_1 \ldots x_n) = x_1@x_2@\ldots@x_n@$. Ainsi, pour tous mots u, v, on a $\phi(u)@ = @\psi(v)$ ssi $u = v$.

On définit alors $u'_i = \phi(u_i)$ pour $i > 1$ et $u'_1 = \S\phi(u_1)$ où § est un nouveau symbole. On définit $v'_i = \psi(v_i)$. Puis on ajoute deux dominos $u'_0 = \S$, $v'_0 = \S\S@$ et $u'_{n+1} = @\#$ et $v'_{n+1} = \#$ où # est un nouveau symbole.

Si $1, i_2, \ldots, i_k$ est une solution de PCPM, alors pour $j \in [2, k]$, $i_j \neq 1$ donc $u'_{i_j} = \phi(u_{i_j})$ et $v'_{i_j} = \psi(v_{i_j})$, d'où $u'_0u'_1u'_{i_2}\cdots u'_{i_k}u'_{n+1} = \S\S\phi(u_1u_{i_2}\ldots u_{i_k})@\# = \S\S@\psi(v_1v_{i_2}\ldots v_{i_k})\# = v'_0v'_1v'_{i_2}\ldots v'_{i_k}v'_{n+1}$, donc $0, 1, i_2, \ldots, i_k, n+1$ est une solution de PCP.

Réciproquement, si PCP a une solution $u'_{i_1}\cdots u'_{i_k} = v'_{i_1}\cdots v'_{i_k}$, supposons-la la plus courte possible. Alors u'_{i_1} et v'_{i_1} doivent commencer par la même lettre donc $i_1 = 0$, puis u'_{i_2} doit commencer par §@ donc $i_2 = 1$; enfin, u'_{i_k} et v'_{i_k} doivent terminer par la même lettre donc $i_k = n+1$. Il faut maintenant montrer que $i_j \notin \{0, 1, n+1\}$ pour $j \in [3, k-1]$. Supposons $j \in [3, k-1]$ le plus petit possible tel que $i_j \in \{0, 1, n+1\}$. Alors on a $u'_0u'_1u'_{i_3}\cdots u'_{i_{j-1}}u'_{i_j} = \S\S\phi(u_1\ldots u_{i_{j-1}})u'_{i_j}$ d'une part, et $v'_0v'_1v'_{i_3}\cdots v'_{i_{j-1}}v'_{i_j} = \S\S@\psi(v_1\ldots v_{i_{j-1}})v'_{i_j}$ d'autre part. Si $i_j = 0$, alors u'_{i_j} et v'_{i_j} commencent par § (qui n'était pas apparu depuis les deux premiers), donc $\phi(u_1\ldots u_{i_{j-1}}) = @\psi(v_1\ldots v_{i_{j-1}})$, ce qui est impossible puisque le membre droit fini par @ et pas le membre gauche. Si $i_j = 1$, alors u'_{i_j} commence par § et de même on doit avoir égalité avant les symboles §, mais cette fois v'_{i_j} ne commence pas par §. Il faut donc attendre un domino 0 pour que v'_0 contienne §. Mais dans ce cas si $u_1 \neq \epsilon$ alors le mot du haut contient deux symboles § séparés par des lettres, tandis que le mot du bas contient deux symboles § consécutifs ; et si $u_1 = \epsilon$, il doit y avoir deux u'_1 consécutifs pour obtenir deux symboles § consécutifs, mais en bas ils sont précédés par @ ou # et pas en haut. Enfin, si $i_j = n+1$, alors les deux mots doivent être égaux avant # donc $u'_{i_1}\cdots u'_{i_j} = v'_{i_1}\cdots v'_{i_j}$, ce qui est une solution plus courte, contradiction.

Ainsi, pour tout $j \in [3, k-1]$, $i_j \notin \{0, 1, n+1\}$ et donc $u'_{i_j} = \phi(u_{i_j})$ et $v'_{i_j} = \psi(v_{i_j})$. On en déduit que $u'_0u'_1u'_{i_3}\cdots u'_{i_{k-1}}u'_{n+1} = \S\S\phi(u_1\ldots u_{i_{k-1}})@\#$ d'une part, et $v'_0v'_1v'_{i_3}\cdots v'_{i_{k-1}}v'_{n+1} = \S\S@\psi(v_1\ldots v_{i_{k-1}})\#$ d'autre part. Donc $\phi(u_1\ldots u_{i_{k-1}})@ = @\psi(v_1\ldots v_{i_{k-1}})$, puis $u_1\ldots u_{i_{k-1}} = v_1\ldots v_{i_{k-1}}$, donc PCPM a une solution.

Théorème : le problème PCPM est indécidable.

Preuve : réduction du problème de l'arrêt $H = \{\langle M\rangle \ : \ M \text{ s'arrête}\}$. Pour cela il faut simuler une MT M à un ruban, alphabet de travail Γ, états $q_0, \ldots, q_k$ où q_0 initial et q_k terminal, fonction de transition δ. Le mot constitué par les dominos décrira les étapes de calcul encodées comme suit : à chaque étape, on retient le contenu du ruban, l'état et la position de la tête sous la forme xq_iy si le mot avant la tête de lecture est x, la tête est positionnée sur la dernière lettre de x, l'état est q_i et y est le mot après la tête. Les symboles # séparent les différentes étapes. Par exemple, $abaq_1a\#abbaq_2$ signifie que le ruban contient $abaa$ au départ, que la tête est située sur le second a et est dans l'état q_1, puis elle écrit b (à la place de a) et se déplace vers la droite.

On encode les transitions sur les dominos ; le mot du haut sera en « retard » d'une étape sur le mot du bas. Pour commencer par l'état q_0, on définit le premier domino par $\boxed{\begin{array}{c}\S\\ \hline \S\#q_0B\end{array}}$ où § est un nouveau symbole. Pour séparer les étapes, on aura le domino $\boxed{\begin{array}{c}\#\\ \hline \#\end{array}}$. Pour effectuer une transition, on aura $\boxed{\begin{array}{c}xq_i\\ \hline q_jy\end{array}}$ si

$\delta(q_i, x) = (q_j, y, \leftarrow)$ (pour $x, y \in \Gamma$) et $\boxed{\begin{array}{|c|}\hline xq_ix' \\ \hline yx'q_j \\ \hline \end{array}}$ si $\delta(q_i, x) = (q_j, y, \rightarrow)$. Lorsqu'on est au bord gauche du mot : $\boxed{\begin{array}{|c|}\hline \#q_ix \\ \hline \#yxq_j \\ \hline \end{array}}$ si $\delta(q_i, B) = (q_j, y, \rightarrow)$ et $\boxed{\begin{array}{|c|}\hline \#q_i \\ \hline \#q_jy \\ \hline \end{array}}$ si $\delta(q_i, B) = (q_j, y, \leftarrow)$. Il faut aussi des dominos $\boxed{\begin{array}{|c|}\hline x \\ \hline x \\ \hline \end{array}}$ pour chaque $x \in \Gamma$ afin de recopier les symboles inchangés par la tête. Enfin, pour terminer le calcul, il faut supprimer les symboles sur le ruban pour que le mot du haut puisse rattraper son retard : on ajoute les dominos $\boxed{\begin{array}{|c|}\hline xq_k \\ \hline q_k \\ \hline \end{array}}$, $\boxed{\begin{array}{|c|}\hline q_kx \\ \hline q_k \\ \hline \end{array}}$ et $\boxed{\begin{array}{|c|}\hline q_k\# \\ \hline \\ \hline \end{array}}$.

Si M s'arrête, alors en suivant les étapes de calcul de M on obtient une solution à ce PCP. Si ce PCP a une solution, cela correspond au calcul de M donc M s'arrête.

Exemple : si on a le fonctionnement suivant de M sur l'alphabet $\{a, b\}$: sur l'état q_0 elle écrit a, va à droite et passe dans q_1, puis de q_1 elle écrit b, va à gauche et passe dans q_2, puis écrit b, va à gauche et passe dans q_3, puis va à droite et passe dans q_4 qui est terminal, on aura les dominos suivants :

$$\boxed{\begin{array}{|c|}\hline \S \\ \hline \S\#q_0B \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline \# \\ \hline \# \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline a \\ \hline a \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline b \\ \hline b \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline \#q_0B \\ \hline \#aBq_1 \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline Bq_1 \\ \hline q_2b \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline aq_2 \\ \hline q_3b \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline \#q_3b \\ \hline \#bq_4 \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline bq_4 \\ \hline q_4 \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline q_4b \\ \hline q_4 \\ \hline \end{array}}\ \boxed{\begin{array}{|c|}\hline q_4\# \\ \hline \\ \hline \end{array}}$$

La solution reflète le fonctionnement de la machine :

$\S$	$\#q_0B$	$\#$	a	Bq_1	$\#$	aq_2	b	$\#q_3b$	b	$\#$	bq_4	b	$\#$	q_4b	$\#$	$q_4\#$
$\S\#q_0B$	$\#aBq_1$	$\#$	a	q_2b	$\#$	q_3b	b	$\#bq_4$	b	$\#$	q_4	b	$\#$	q_4	$\#$	

Corollaire : PCP est indécidable.

Autres exemples de problèmes "naturels" indécidables : une équation diophantienne a-t-elle une solution ? Un énoncé de la logique du premier ordre est-il valide ? Problème du mot dans un groupe finiment présenté : on se donne la présentation et un mot u et on veut savoir si $u = 1 \ldots$

3 Complexité

- « Temps » mis par une MT : nombre d'étapes de calcul ; « espace » : nombre de cases visitées.
- Idée de mesurer le nombre d'étapes en fonction de la taille de l'entrée : Hartmanis et Stearns 1965.
- Classes P, NP
- Cook, Karp, Levin 1971 : NP-complétude, nombreux problèmes

3.1 Temps déterministe

Si on compte le temps à un facteur polynomial près, le modèle de calcul importe peu : simulations polynomiales entre MT (nombre de rubans, alphabet, etc.) et entre RAM et MT.

Proposition : si M est une machine à k rubans fonctionnant en temps $t(n)$, on peut simuler M par une machine M' à un ruban fonctionnant en temps $O(k^2t(n)^2)$.

Preuve : on stocke la case i du ruban j de la machine M sur la case $ik+j$ du ruban de M' (c'est-à-dire qu'on alterne les cases des différents rubans). On a un symbole spécial pour connaître l'emplacement des têtes de lecture sur les k rubans. Pour chaque transition de M', il faut pour tout $i \in [1, k]$ aller chercher l'emplacement de la tête du i-ème ruban et revenir au début, ce qui prend $O(kt(n))$ étapes à chaque fois (donc $O(k^2t(n))$ en tout), puis effectuer les transitions de la même façon, ce qui prend encore $O(kt(n))$ étapes pour chaque ruban. Ainsi, chaque étape de M est simulée par $O(k^2t(n))$ étapes de M', donc M' fonctionne en temps $O(k^2t(n)^2)$.

Pour la suite, on choisira les MT à trois rubans bi-infinis.

Définition : si $f : \mathbb{N} \to \mathbb{N}$ est une fonction, la classe $\mathsf{DTIME}(f(n))$ est l'ensemble des langages reconnus par une machine de Turing fonctionnant en temps $\leq f(n)$ sur les entrées de taille n.

Remarque : si $f < g$, on a bien sûr $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n))$.

Théorème (accélération linéaire) : pour toute fonction f et toute constante $\epsilon > 0$, on a $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(n + \epsilon f(n))$.

Preuve : soit M une machine fonctionnant en temps $f(n)$. On construit une machine M' simulant $m = 6/\epsilon$ étapes de M en six étapes. Pour cela il faut bien sûr agrandir l'alphabet (on prend les m-uplets de l'alphabet de départ) et le nombre d'états. Chaque case de M' contient m cases de M, dont une est marquée par un symbole spécial permettant de savoir sur quelle case est la tête de M.

Au départ, M' lit l'entrée x et la regroupe sur un autre ruban par paquets de m caractères pour avoir une entrée de taille n/m. Puis pour effectuer m transitions de M, la machine M' agit comme suit : elle visite la case de droite pour retenir les m lettres qu'elle contient, elle visite la case de gauche pour retenir les m lettres qu'elle contient : ainsi, elle connaît les m voisins à gauche et à droite de la case sur laquelle était M et peut effectuer m transitions de M d'un coup.

$\longrightarrow$ Ainsi, le temps n'est défini qu'à une constante près.

Définition : si $\mathcal{F}$ est un ensemble de fonctions, alors $\mathsf{DTIME}(\mathcal{F}) = \cup_{f\in\mathcal{F}}\mathsf{DTIME}(f(n))$. On définit alors $\mathsf{P} = \mathsf{DTIME}(n^{O(1)}) = \cup_{k>0}\mathsf{DTIME}(n^k)$ et $\mathsf{EXP} = \mathsf{DTIME}(2^{n^{O(1)}}) = \cup_{k>0}\mathsf{DTIME}(2^{n^k})$ (on a donc $\mathsf{P} \subseteq \mathsf{EXP}$).

Exemples :
- $\{(x, y, z) \ : \ z = xy\} \in \mathsf{P}$;
- $\{G \ : \ G \text{ est un graphe connexe}\} \in \mathsf{P}$;
- si $\phi(x_1, \ldots, x_n)$ formule booléenne (p.ex. $\phi(x_1, x_2, x_3) \equiv (\neg x_1 \vee x_2) \wedge (\neg x_2 \wedge x_3)$) :

$$\{\phi \ : \ \forall x, \phi(x) = 1\} \in \mathsf{EXP}$$

Propriété : la classe P est close par union, intersection et complémentaire.

Preuve : si $L_1, L_2 \in \mathsf{P}$ reconnus par M_1, M_2 en temps $p_1(n), p_2(n) \in n^{O(1)}$, pour reconnaître $L_1 \cup L_2$ (respectivement $L_1 \cap L_2$) on simule les deux machines et on accepte si l'une des deux accepte (resp. si les deux acceptent) ; le temps d'exécution est $O(p_1 + p_2)$. Pour ${}^cL_1$, on simule M_1 et on accepte ssi M_1 rejette ; le temps d'exécution est $O(p_1)$.

Propriété : si f et g sont deux fonctions calculables en temps polynomial, alors $f \circ g$ est calculable en temps polynomial.

Preuve : pour calculer $f \circ g(x)$, il suffit de calculer $g(x)$ en temps polynomial $p_1(|x|)$, puis $f(g(x))$ en temps $p_2(|g(x)|)$. Puisque $g(x)$ est calculable en temps $p_1(|x|)$, on a $|g(x)| \leq p_1(|x|)$ donc $p_2(|g(x)|) \leq p_2(p_1(|x|))$ ce qui est polynomial en $|x|$.

Proposition : soit M une machine à trois rubans fonctionnant en temps $t(n)$. Alors il existe une machine universelle U à deux rubans pouvant simuler M en temps $O(|M|t^2(n))$ (où $|M|$ est la taille du codage de M).

Preuve : sur chaque case du premier ruban de U, on stocke des paires pour pouvoir écrire (M, q) (où q est l'état de M) et x l'un sur l'autre. De même, sur le second ruban on représente les deux autres rubans de M. Pour chaque transition de M, on doit parcourir le code de M pour trouver la transition à effectuer, et effectuer cette transition. Pour cela, on doit parcourir le premier ruban et le modifier en conséquence (modifier q et x) et parcourir deux fois le second ruban pour modifier les deux rubans de travail de la machine. D'où le temps $|M|t(n)^2$.

Proposition (padding) : soit $A \in \mathsf{DTIME}(2^{O(n)})$. Alors $A' = \{(x, 0^{2^{|x|}}) \ : \ x \in A\} \in \mathsf{P}$.

Preuve : soit M une machine reconnaissant A en temps 2^{kn}. Soit M' la machine qui sur l'entrée $(x, 0^m)$ vérifie que $m = 2^{|x|}$ et si c'est le cas simule $M(x)$. Cette machine fonctionne en temps $O(2^{|x|} + 2^{k|x|})$, c'est-à-dire $O(n^k)$ si $n = |(x, 0^m)|$ est la taille de l'entrée.

Définition : une fonction $f : \mathbb{N} \to \mathbb{N}$ est constructible en temps s'il existe une MT à deux rubans qui, sur l'entrée 1^n, écrit $1^{f(n)}$ sur son second ruban en temps $O(f(n))$.

Exemples :
- $f(n) = n$ (recopier l'entrée) ;
- $f(n) = n^2$ (recopier n fois l'entrée) ;

– $f(n) = 2^n$ (écrire 2^n en binaire puis retrancher 1 à chaque étape : il faut propager la retenue donc chaque étape coûte en moyenne $\sum_{i=1}^{n} i2^{-i}$ qui converge) ;
– $f(n) = n\lfloor \log n \rfloor$ (on garde un symbole sur deux jusqu'à ce qu'il n'y ait plus qu'un symbole ; ça prend $O(n \log n)$ et ça calcule $k = \lfloor \log n \rfloor$. Puis on recopie k fois 1^n).

Remarque : si f est constructible en temps et non ultimement constante, alors $f(n) \in \Omega(n)$ puisqu'elle doit lire l'entrée en entier.

Théorème (de hiérarchie déterministe en temps, Hartmanis et Stearns 1965) : soit f une fonction constructible en temps. Alors $\mathsf{DTIME}(f(n))$ est strictement inclus dans $\mathsf{DTIME}(nf(n)^2)$.

Preuve : soit $A = \{\langle M \rangle \ : \ M(\langle M \rangle) \text{ rejette en temps } \leq f(n)\}$, où $n = |\langle M \rangle|$.

Montrons d'abord que $A \in \mathsf{DTIME}(nf(n)^2)$. Il existe une machine M_f qui calcule (en unaire) $f(n)$ en temps $O(f(n))$ grâce à la constructibilité en temps de f. On simule d'abord M_f en temps $O(f(n))$ de façon à avoir $f(n)$ en unaire sur le troisième ruban. Puis, pour décider A, il suffit de simuler $M(\langle M \rangle)$ pendant $\leq f(n)$ étapes sur les deux premiers rubans : par la proposition précédente, tout cela prend $O(|M|f(n)^2) = O(nf(n)^2)$ étapes. Donc $A \in \mathsf{DTIME}(nf(n)^2)$.

Montrons maintenant que $A \notin \mathsf{DTIME}(f(n))$. Supposons que ce soit le cas : il existe une machine M décidant A en temps $f(n)$. Si $\langle M \rangle \in A$ alors par définition de M, $M(\langle M \rangle)$ accepte, mais par définition de A, $M(\langle M \rangle)$ rejette. Si $\langle M \rangle \notin A$, alors par définition de M, $M(\langle M \rangle)$ rejette, mais par définition de A, $M(\langle M \rangle)$ ne rejette pas en temps $\leq f(n)$; puisque M fonctionne en temps $f(n)$, elle doit donc accepter. Dans les deux cas on obtient une contradiction.

Remarque : on a même $\mathsf{DTIME}(o(f(n)/\log f(n)))$ strictement inclus dans $\mathsf{DTIME}(f(n))$ si on fait plus attention à la simulation.

Corollaire : $\mathsf{P} \subsetneq \mathsf{EXP}$.

Preuve : $\mathsf{P} \subseteq \mathsf{DTIME}(n^{\log n})$ et par le théorème, $\mathsf{DTIME}(n^{\log n})$ est strictement inclus dans $\mathsf{DTIME}(2^n)$ donc dans EXP.

Définition : une énumération $M_1, M_2, \ldots$ des MT est une fonction calculable f telle que
– sur l'entrée i, f renvoie le code d'une MT M_i ;
– tous les codes de MT apparaissent dans l'énumération $M_1, M_2, \ldots$

Par exemple, $\langle M_i \rangle$ pourrait être le i-ème code de machine dans l'ordre lexicographique.

Théorème (de la lacune, « Gap theorem », Trakhtenbrot 1964) : pour toute fonction calculable $f : \mathbb{N} \to \mathbb{N}$ telle que $f(n) \geq n$ pour tout n, il existe une fonction calculable t telle que $t(n) \geq n$ et $\mathsf{DTIME}(f(t(n))) = \mathsf{DTIME}(t(n))$.

Preuve : on va construire t telle qu'aucune machine ne s'arrête sur une entrée de taille n en un nombre d'étapes compris entre $t(n)$ et $f(t(n))$. Soit $M_1, M_2, \ldots$ une énumération des MT. La propriété suivante sera satisfaite : aucune des machines $M_1, \ldots, M_i$ ne s'arrête sur une entrée de taille i en un temps compris entre $t(i)$ et $f(t(i))$.

Définissons $t(i)$ pour un i fixé. Soit $u_1 = i$ et $u_{n+1} = f(u_n) + 1$ (note : si on veut t croissante, on définit $u_1 = t(i-1)$). Pour $j \leq i$, considérons la machine M_j sur une entrée x de taille i : soit elle s'arrête en moins de i étapes, soit entre u_k et $f(u_k)$ étapes pour un certain k, soit elle ne s'arrête pas. Puisqu'il y a $i|\Sigma|^i$ couples machines/entrées, il existe un intervalle $[u_k, f(u_k)]$, pour $k \leq i|\Sigma|^i + 1$, tel qu'aucune machine M_j ne s'arrête dans cet intervalle sur une entrée de taille i. On définit alors $t(i) = u_k \geq i$.

S'il existait un langage $L \in \mathsf{DTIME}(f(t(n))) \setminus \mathsf{DTIME}(t(n))$ décidé par une machine M_i, alors pour une infinité d'entiers j, M_i s'arrêterait dans l'intervalle $[t(n), f(t(n))]$ sur des entrées de taille i. Or par définition de t, pour tout x de taille $n \geq i$, $M_i(x)$ ne s'arrête pas dans l'intervalle $[t(n), f(t(n))]$: une contradiction.

3.2 Machines non déterministes

Définition : une machine de Turing non déterministe (MTND) est une machine de Turing ayant plusieurs transitions possibles à chaque étape. Ainsi, plutôt que d'avoir une fonction de transition $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, on a une relation $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \rightarrow\})$: si $((q, x), (q', y, d)) \in \delta$, cela signifie qu'à partir de l'état q et en lisant la lettre x, on peut écrire y, se déplacer dans la direction d et passer dans l'état q'.

Définition : une suite de transitions valides d'une MTND M est appelée une exécution de M. On dit que M accepte un mot x s'il existe une exécution de M qui accepte x : dans l'arbre des exécutions possibles il doit y avoir une branche qui accepte.

Exemple : sur l'entrée z représentant un entier en binaire, trouver x et y tels que $xy = z$. Il s'agit d'abord de deviner x et y, puis de vérifier que leur multiplication vaut z. La multiplication se fait par une machine déterministe. Pour deviner x, on devine ses chiffres binaires un par un, jusqu'à avoir écrit un entier dont la taille est inférieure à celle de z. Pour cela, on a les deux relations $((q, B), (q, 0, \rightarrow))$ et $((q, B), (q, 1, \rightarrow))$. Idem pour y.

Proposition : il existe une MTND universelle qui simule en temps $O(|M|f(n)^2)$ une MTND M fonctionnant en temps $f(n)$.

Preuve : on fait de même que pour les machines déterministes, mais il faut simuler toutes les transitions possibles. Pour cela, à chaque étape on compte le nombre k de transitions possibles, on devine un entier i entre 1 et k et on effectue la i-ème transition.

Corollaire : toute MTND est équivalente à une MTND qui a au plus deux transitions possibles à chaque étape, et qui fonctionne avec ralentissement linéaire.

Preuve : même argument que ci-dessus : il s'agit de choisir la i-ème transition parmi k en devinant un entier $i \leq k$.

Note : dorénavant, on considèrera en général des MTND qui ont au plus deux transitions possibles à chaque étape.

Définition : un langage L est reconnu par une MTND M en temps $f(n)$ si M accepte exactement les mots de L et toutes les exécutions possibles de M sur un mot de taille n fonctionnent en temps $\leq f(n)$. La classe $\mathsf{NTIME}(f(n))$ est l'ensemble des langages reconnus par une MTND en temps $f(n)$. Si $\mathcal{F}$ est un ensemble de fonctions, alors $\mathsf{NTIME}(\mathcal{F}) = \cup_{f \in \mathcal{F}} \mathsf{NTIME}(f(n))$.

Remarque : si $f \leq g$, on a bien sûr $\mathsf{NTIME}(f(n)) \subseteq \mathsf{NTIME}(g(n))$. Et $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$ puisqu'une MTD est un cas particulier de MTND.

Proposition : $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$.

Preuve : à chaque étape d'une MTND M, il y a 2 transitions possibles. Donc en tout il y a $2^{f(n)}$ exécutions possibles : on peut reconnaître le langage en les simulant toutes une par une et en acceptant ssi l'une d'entre elles accepte.

Définition : on définit les classes $\mathsf{NP} = \mathsf{NTIME}(n^{O(1)}) = \cup_{k>0} \mathsf{NTIME}(n^k)$ et $\mathsf{NEXP} = \mathsf{NTIME}(2^{n^{O(1)}}) = \cup_{k>0} \mathsf{NTIME}(2^{n^k})$ (on a donc $\mathsf{NP} \subseteq \mathsf{NEXP}$).

Proposition : la classe NP est close par union et intersection.

Preuve : soient $L_1, L_2 \in \mathsf{NP}$ reconnus par M_1, M_2 en temps $p_1(n), p_2(n)$. Pour reconnaître $L_1 \cup L_2$, on simule M_1 ; si le chemin qu'on simule est acceptant, on accepte, sinon on simule M_2. Le temps mis est donc $p_1 + p_2$. Pour reconnaître $L_1 \cap L_2$, on simule M_1 ; si le chemin qu'on simule est rejetant, on rejette, sinon on simule M_2. Le temps mis est donc $p_1 + p_2$.

Remarque : on ne sait pas si NP est clos par complément (et ce n'est probablement pas le cas) car on ne sait pas construire une MTND M' qui fait l'opposé d'une MTND M en temps polynomial. On définit donc la classe coNP qui est l'ensemble des langages dont le complémentaire est dans NP. En d'autres termes, il s'agit des langages dont tous les chemins de la MTND rejettent, ou de manière équivalente dont tous les chemins acceptent.

Proposition : la classe NP est l'ensemble des problèmes qui ont des témoins de taille polynomiale qu'on peut vérifier en temps (déterministe) polynomial. Plus formellement, $A \in \mathsf{NP}$ ssi il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que

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

Preuve : si A vérifie cette propriété, alors une MTND pour A devine y et simule la machine déterministe M_B pour B pour décider si $(x, y) \in B$: si M_B fonctionne en temps $p_B(n)$, le temps total sera $p(n) + p_B(n + p(n))$ ce qui reste polynomial.

Réciproquement, si $A \in \mathsf{NP}$, soit M une MTND polynomiale pour A. Si on connaît l'ensemble des choix ND de M, on peut simuler la branche correspondante par une machine déterministe. Soit B le problème suivant : $(x, y) \in B$ ssi la branche décrite par y dans $M(x)$ accepte. Alors $B \in \mathsf{P}$ puisque M fonctionne en temps polynomial, et $x \in A$ ssi $\exists y \in \Sigma^{\leq p(|x|)}(x, y) \in B$, où $p(n)$ est la longueur de y, c'est-à-dire le temps d'exécution de M.

Remarque : de manière duale, $A \in \mathsf{coNP}$ ssi il existe $B \in \mathsf{P}$ et un polynôme $p(n)$ tels que $[x \in A \iff \forall y \in \Sigma^{\leq p(|x|)}(x, y) \in B]$.

Exemples de problème NP

- Circuit hamiltonien – Entrée : un graphe G. Problème : G admet-il un circuit qui passe exactement une fois par chaque sommet ? On devine le circuit et on vérifie qu'il s'agit bien d'un chemin qui passe une fois et une seule par chaque sommet.
- Clique – Entrée : un graphe G et un entier k. Problème : existe-t-il une clique de taille k dans G ? On devine les k sommets de la clique et on vérifie qu'ils sont tous reliés.
- SAT – Entrée : une formule booléenne $\phi(x_1, \ldots, x_n)$. Problème : ϕ est-elle satisfaisable (admet-elle une solution) ? On devine une instanciation $\bar{a}$ des variables et on vérifie que $\phi(\bar{a})$ est vraie. (Exemple de problème coNP : est-ce que toute instanciation est solution)
- Factorisation – Entrée : $x \in \mathbb{N}$. Problème : trouver une décomposition $x_1 x_2 = x$. Problème de décision : $\{(x, i, j) \ : \$ le i-ème bit de x_j est $1\}$. On devine x_1, x_2 et on vérifie que $x_1 x_2 = x$.
- Énormément d'autres problèmes importants en pratique.

D'où l'importance de la question « $\mathsf{P} = \mathsf{NP}$? » : existe-t-il des algorithmes performants pour résoudre ces problèmes ?

Théorème (hiérarchie non-déterministe en temps, Cook 1973 – ici preuve de Žàk 1983) : si f est constructible en temps, alors $\mathsf{NTIME}(f(n))$ est strictement inclus dans $\mathsf{NTIME}(nf(n+1)^2)$.

Remarque : la preuve du théorème de hiérarchie déterministe ne fonctionne pas car on simulait une machine et on en prenait la négation : on ne sait pas faire ça suffisamment efficacement avec des MTND.

Preuve : Soit L le langage accepté par la MTND M suivante sur l'entrée $(\langle N \rangle, 0^k)$: si $k > 2^{f(m)}$, pour $m = |(\langle N \rangle, \epsilon)|$, alors on simule *de manière déterministe* $N(\langle N \rangle, \epsilon)$ pendant $f(m)$ étapes (ce qui prend un temps $2^{f(m)} < k$) et on accepte ssi N rejette ; si $k \leq 2^{f(m)}$, alors on simule (de manière non-déterministe) $N(\langle N \rangle, 0^{k+1})$ pendant $f(n+1)$ étapes, où $n = |(\langle N \rangle, 0^k)|$ (ce qui prend un temps $(n+1)f(n+1)^2$ par la machine universelle). Ainsi, M fonctionne en temps $O(nf(n+1)^2)$ donc $L \in \mathsf{NTIME}(nf(n+1)^2)$.

Supposons maintenant que $L \in \mathsf{NTIME}(f(n))$: le langage L est donc reconnu par une machine N en temps $\leq f(n)$. Pour $k > 2^{f(m)}$, on a $N(\langle N \rangle, 0^k) = \neg N(\langle N \rangle, \epsilon)$ et pour $k \leq 2^{f(m)}$ on a $N(\langle N \rangle, 0^k) = N(\langle N \rangle, 0^{k+1})$. Ainsi, $N(\langle N \rangle, \epsilon) = N(\langle N \rangle, 0) = N(\langle N \rangle, 0^2) = \ldots = N(\langle N \rangle, 0^{k_0}) = \neg N(\langle N \rangle, \epsilon)$ pour $k_0 = 2^{f(m)} + 1$, une contradiction.

Remarque : on peut améliorer le résultat en $\mathsf{NTIME}(o(f(n))) \neq \mathsf{NTIME}(f(n+1))$.
Corollaire : $\mathsf{NP} \subsetneq \mathsf{NEXP}$
Preuve : $\mathsf{NP} \subseteq \mathsf{NTIME}(n^{\log n}) \neq \mathsf{NTIME}(2^n) \subseteq \mathsf{NEXP}$

Pour résumer : $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$ et $\mathsf{P} \neq \mathsf{EXP}$ et $\mathsf{NP} \neq \mathsf{NEXP}$

Proposition (padding) : $\mathsf{P} = \mathsf{NP}$ implique $\mathsf{EXP} = \mathsf{NEXP}$.

Preuve : soit $A \in \mathsf{NEXP}$ décidé par MTND en temps n^k. Soit $A' = \{(x, 0^{2^{|x|^k}}) \ : \ x \in A\}$: alors $A' \in \mathsf{NP}$ donc $A' \in \mathsf{P}$ par hypothèse, donc A' est reconnu par une MT déterministe en temps $2^{\alpha|x|^k}$, donc $A \in \mathsf{EXP}$.

3.3 Réductions polynomiales

Comme en calculabilité, pour comparer deux langages on a une définition formelle de la notion « être plus facile » en passant par les réductions.

Définition : soient A et B deux langages. On dit que A se réduit polynomialement à B, et on note $A \leq_m^p B$, s'il existe une fonction $f : \Sigma^* \to \Sigma^*$ calculable en temps polynomial telle que $x \in A \iff f(x) \in B$. La fonction f est appelée réduction de A à B.

Voyons un exemple.

Définition : un ensemble stable d'un graphe G est un ensemble de sommets n'ayant aucune arête entre eux. Le problème IS (Independent Set) est le suivant :
Entrée – un graphe G et un entier k.
Question – existe-t-il un ensemble stable de taille k ?

Exemple : IS $\leq_m^p$ Clique. En effet, soit $f(G,k) = ({}^cG, k)$, où ${}^cG$ est le complémentaire de G (i.e. x et y sont reliés dans G ssi ils ne sont pas reliés dans ${}^cG$). La fonction f est clairement calculable en temps polynomial. Si $x_1, \ldots, x_k$ est un stable de G, alors c'est une clique de ${}^cG$; réciproquement, si $x_1, \ldots, x_k$ est une clique de ${}^cG$, alors c'est un stable de G. Ainsi, $(G,k) \in \text{IS} \iff f(G,k) \in \text{Clique}$.

Propriété : la relation $\leq_m^p$ est transitive.

Preuve : soient A, B, C tels que $A \leq_m^p B$ et $B \leq_m^p C$: il existe f, g tels que $x \in A$ ssi $f(x) \in B$ et $y \in B$ ssi $g(y) \in C$. Ainsi, $x \in A$ ssi $g(f(x)) \in C$. Or $g \circ f$ est calculable en temps polynomial donc $A \leq_m^p C$.

Propriété : les classes P et NP sont closes par réduction $\leq_m^p$.

Preuve : si $A \in \mathsf{P}$ (respectivement $A \in \mathsf{NP}$) et $B \leq_m^p A$, alors $x \in B$ ssi $f(x) \in A$, donc pour décider B il suffit de calculer $f(x)$ et d'utiliser la machine déterministe (resp. non déterministe) polynomiale pour A pour décider si $f(x) \in A$. Cela nous donne un algorithme déterministe (resp. non déterministe) polynomial pour B, donc $B \in \mathsf{P}$ (resp. $B \in \mathsf{NP}$).

Définition : soit $\mathcal{C}$ une classe de complexité. On dit qu'un langage A est $\mathcal{C}$-complet (pour la réduction $\leq_m^p$) si :

1. $A \in \mathcal{C}$, et
2. pour tout langage $B \in \mathcal{C}$, $B \leq_m^p A$.

Si seule la condition 2 est satisfaite, on dit que A est $\mathcal{C}$-difficile.

Proposition : l'ensemble des langages P-complets est exactement $\mathsf{P} \setminus \{\emptyset, \Sigma^*\}$.

Preuve : $\emptyset$ et Σ^* ne peuvent pas être complets ; en effet : si $A \leq_m^p \emptyset$ alors $x \in A \iff f(x) \in \emptyset$ donc $A = \emptyset$; si $A \leq_m^p \Sigma^*$ alors $x \in A \iff f(x) \in \Sigma^*$ donc $A = \Sigma^*$. Soit maintenant $A \in \mathsf{P} \setminus \{\emptyset, \Sigma^*\}$ et montrons que A est P-complet. Soient $x_0 \in A$ et $x_1 \notin A$. Soit $B \in \mathsf{P}$ et $f(x) = x_0$ si $x \in B$ et $f(x) = x_1$ si $x \notin B$. Alors f est calculable en temps polynomial et est une réduction de B à A.

Proposition : si A est NP-complet et $A \in \mathsf{P}$ alors $\mathsf{P} = \mathsf{NP}$.

Preuve : soit $B \in \mathsf{NP}$. On a $B \leq_m^p A$ et $A \in \mathsf{P}$ donc $B \in \mathsf{P}$ puisque P est close par réduction $\leq_m^p$. Ainsi $\mathsf{NP} \subseteq \mathsf{P}$.

Proposition : soient $A, B \in \mathsf{NP}$. Si A est NP-complet et $A \leq_m^p B$ alors B est NP-complet.

Preuve : puisque $B \in \mathsf{NP}$, il suffit de montrer que tout problème $C \in \mathsf{NP}$ se réduit à B. On sait que $C \leq_m^p A$ et $A \leq_m^p B$ donc $C \leq_m^p B$ par transitivité de $\leq_m^p$.

Proposition : le langage $A = \{\langle M, x, 0^t\rangle \ : \ M(x) \text{ accepte en temps } \leq t\}$, où M est une MTND, est NP-complet.

Preuve : montrons d'abord que $A \in \mathsf{NP}$; il suffit de voir que notre machine universelle pour les MTND peut simuler t étapes de $M(x)$ en temps $O(|M|t^2)$, ce qui est polynomial en $|\langle M, x, 0^t\rangle|$.

Montrons maintenant que tout problème $B \in \mathsf{NP}$ se réduit à A. Soit M une MTND fonctionnant en temps n^k pour B. On définit la réduction f par $f(x) = \langle M, x, 0^{|x|^k}\rangle$. Alors f est calculable en temps polynomial et $x \in B$ ssi $M(x)$ accepte en temps $\leq |x|^k$ ssi $\langle M, x, 0^{|x|^k}\rangle \in A$ ssi $f(x) \in A$.

Définition : SAT est le langage $\{\phi \ : \ \phi \text{ satisfaisable}\}$, où $\phi = \phi(x_1, \ldots, x_k)$ est une formule booléenne. Exemple de codage pour ϕ : on représente x_i par i en binaire et $\neg, \wedge$ et $\vee$ ainsi que les parenthèses par des symboles spéciaux.

Proposition : le langage SAT est NP-complet.

Preuve : montrons d'abord que SAT est dans NP. La MTND devine une instanciation des variables et vérifie qu'il s'agit d'une solution : pour cela on peut évaluer de manière récursive la valeur de la formule, ce qui se fait en temps polynomial.

Montrons maintenant que tout problème $B \in \mathsf{NP}$ se réduit à SAT. Soit M une MTND fonctionnant en temps n^k pour B. On suppose que M a un état d'acceptation q_a et un état de rejet q_r, et que l'alphabet de travail est $\{0,1\}$. On décrit le fonctionnement de $M(x)$ par une formule booléenne, où $n = |x|$.

À l'étape i, on note $c_{i,a}$ le contenu de la case a du ruban (pour $-n^k \leq a \leq n^k$) ; pour chaque état q on note $r_{i,q} = 1$ si l'état est q ; enfin, pour chaque position $-n^k \leq a \leq n^k$, on note $p_{i,a} = 1$ si la tête est à la position a. Ainsi, on a $O(n^{2k})$ variables.

Pour tout i, on peut définir une formule $Trans_i$ qui dit si la transition de l'étape i à l'étape $i+1$ est correcte : selon la table de transition de M, il suffit de dire que si $p_{i,a} = 1$ et $r_{i,q} = 1$, alors $p_{i+1,a'} = 1$ et $r_{i+1,q'} = 1$ et $c_{i+1,a} = b$ où a' est la nouvelle position de la tête, q' le nouvel état et b le symbole écrit par la tête avant de bouger (attention plusieurs transitions possibles car non déterministe).

Pour le début du calcul (à l'étape 0), on a une formule $Debut$ qui dit que l'état est q_0, que $c_{0,a} = x_a$ (la a-ème lettre du mot x) et que la tête est en position 0. Enfin, on a une dernière formule Acc qui dit que l'on a atteint l'état d'acceptation q_a à la fin du calcul, c'est-à-dire $r_{n^k,q_a} = 1$.

Ainsi, $M(x)$ a un chemin acceptant ssi $\exists c_{i,a}, r_{i,q}, p_{i,a} \quad \psi_x$, où $\psi_x = Debut \wedge \bigwedge_{i \leq n^k} Trans_i \wedge Acc$. La réduction de B à SAT est alors la fonction $f(x) = \psi_x$: f est calculable en temps polynomial et $x \in B$ ssi $M(x)$ accepte en temps $\leq n^k$ ssi il existe une solution à ψ_x ssi $f(x) \in$ SAT.

Définition : une formule booléenne $\phi(x_1, \ldots, x_k)$ est dite sous forme normale conjonctive (CNF) si ϕ est une conjonction de disjonctions, c'est-à-dire de la forme $\bigwedge_i \bigvee_{j \in I_i} y_j$ où $y_j = x_j$ ou $\neg x_j$ (littéral). Chaque disjonction est appelée clause. Si chaque clause a exactement k littéraux, on dit que la formule est en k-CNF.

Définition : le langage 3-SAT est l'ensemble des formules booléennes satisfaisables en 3-CNF.

Corollaire : 3-SAT est NP-complet.

Preuve : il est clair de 3-SAT $\leq_m^p$ SAT donc que 3-SAT est dans NP. Il s'agit maintenant de montrer que SAT $\leq_m^p$ 3-SAT. Soit $\phi(x_1, \ldots, x_k)$ une instance de SAT : $\phi \in$ SAT ssi $\exists x \phi(x)$. On décrit une nouvelle formule ψ en 3-CNF. On voit ϕ comme un arbre. Si ϕ possède des négations en dehors des feuilles, on les fait descendre aux feuilles. On ajoute des nouvelles variables y_i pour chacun des sommets de l'arbre. Aux feuilles, on dit que y_i est égal au littéral a correspondant : $(y_i \vee \neg a) \wedge (\neg y_i \vee a)$. Pour une porte $\wedge$: y_i doit être égal à $y_j \wedge y_{j'}$, donc on a $(y_i \vee \neg y_j \vee \neg y_{j'}) \wedge (\neg y_i \vee y_j) \wedge (\neg y_i \vee y_{j'})$. Pour une porte $\vee$: y_i doit être égal à $y_j \vee y_{j'}$, donc on a $(\neg y_i \vee y_j \vee y_{j'}) \wedge (y_i \vee \neg y_j) \wedge (y_i \vee \neg y_{j'})$. Ainsi on obtient ψ avec ≤ 3 littéraux par clause. Il suffit d'ajouter des littéraux redondants pour obtenir ψ en 3-CNF. Enfin, $\exists x \phi(x)$ ssi $\exists x, y \psi(x, y)$: la fonction qui à ϕ associe ψ est une réduction de SAT à 3-SAT et est calculable en temps polynomial.

Remarque : attention, la satisfaisabilité des formules en DNF ($\bigvee_i \bigwedge_{j \in I_i} y_j$) est décidable en temps polynomial.

Proposition : IS (independent set) est NP-complet.

Preuve : on représente chaque clause par un triangle dans un graphe, dont les sommets sont les littéraux. On relie les littéraux x et $\neg x$ par une arête. Si ϕ a c clauses, on veut savoir s'il existe un stable de taille c. Si ϕ a une solution, alors en prenant dans chaque clause un littéral vrai, on obtient un stable de taille c. Réciproquement, si G a stable de taille c, alors forcément chaque triangle contient un sommet du stable (car un même triangle ne peut pas en contenir 2, et il y a c triangles) : mettre les littéraux du stable à Vrai fournit une solution à ϕ. Voir Papadimitriou page 189.

Par la réduction de IS à Clique vue plus haut, et puisque Clique est dans NP, on obtient le corollaire suivant.

Corollaire : Clique est NP-complet.

Définition : VC (Vertex Cover) est le problème suivant :
Entrée – un graphe G et un entier k.
Question – existe-t-il un ensemble de sommets de taille k couvrant toutes les arêtes de G ?

Corollaire : VC est NP-complet.

Preuve : réduction de IS. Il suffit de remarquer qu'il existe un stable de taille k ssi il existe un ensemble couvrant de taille $n - k$ (le complémentaire).

Définition : Subset Sum est le problème suivant :
Entrée – un ensemble d'entiers $\{a_1, \ldots, a_k\}$ et un entier N.
Question – existe-t-il un sous-ensemble $I \subseteq [1, k]$ tel que $\sum_{i \in I} a_i = N$?

Proposition : Subset Sum est NP-complet.

Preuve : clairement dans NP. Voir Cormen p. 1014.

Théorème (Ladner, 1975) : si P $\neq$ NP alors il existe un problème $A \in$ NP qui n'est ni dans P ni NP-complet.

Preuve : on va définir une fonction $f : 1^* \to \mathbb{N}$, calculable en temps polynomial, telle que $A = \{\phi : \phi \in \text{SAT et } f(1^{|\phi|}) \text{ pair}\}$. On aura donc $A \in$ NP. En d'autres termes, A consiste en une alternance d'intervalles "vides" et d'intervalles de SAT. Les intervalles "vides" permettent de s'assurer que A n'est pas NP-complet tandis que les intervalles de SAT permettent de s'assurer que $A \notin$ P.

Soit $M_1, M_2 \ldots$ une énumération des machines et soit $g_1, g_2 \ldots$ une énumération des fonctions (réductions). Lorsque $f(n) = 2k$, on s'assure que A n'est pas décidé par la machine M_k tandis que lorsque $f(n) = 2k + 1$, on s'assure que SAT ne se réduit pas à A via g_k.

On définit les valeurs de f successivement : si f est définie jusqu'à 1^n, on note $A_n = \{\phi : |\phi| \leq n \text{ et } \phi \in \text{SAT et } f(1^{|\phi|}) \text{ pair}\}$. On remarquera lors de la construction que $f(1^n) \leq n$. On définit $f(1^{n+1})$ comme suit :

1. si $f(1^n) = 2k$, alors on simule M_k pendant $(\log n)^{\log \log k}$ étapes sur toutes les formules ϕ de taille $\leq \log n$; s'il existe ϕ telle que $M_k(\phi) = 1$ tandis que $\phi \notin A_n$, ou $M_k(\phi) = 0$ tandis que $\phi \in A_n$, alors on définit $f(1^{n+1}) = 2k + 1$; sinon on définit $f(1^{n+1}) = 2k$.
2. si $f(1^n) = 2k + 1$, alors on simule g_k pendant $(\log n)^{\log \log k}$ étapes sur toutes les formules ϕ de taille $\leq \log n$; s'il existe ϕ telle que $|g_k(\phi)| \leq n$ et soit $[\phi \in \text{SAT et } g_k(\phi) \notin A_n]$, soit $[\phi \notin \text{SAT et } g_k(\phi) \in A_n]$, alors $f(1^{n+1}) = 2k + 2$; sinon $f(1^{n+1}) = 2k + 1$.

Il est clair que f est calculable en temps polynomial. Puisque SAT $\notin$ P par hypothèse, la machine M_k doit se tromper sur une infinité d'instances de SAT, donc une formule ϕ existera toujours dans le cas 1 pour n assez grand. Dans le cas 2, si n_0 est le premier n tel que $f(1^n) = 2k + 1$, alors $A_n = A_{n_0}$ est fini donc dans P ; ainsi, par hypothèse SAT ne se réduit pas à A_n, donc une formule ϕ existera toujours dans le cas 2 pour n assez grand. Ainsi, f tend vers l'infini et aucune machine M_k ne décide A en temps $n^{\log \log k}$ et aucune réduction g_k ne réduit SAT à A en temps $n^{\log \log k}$. D'où le résultat.

3.4 Complexité en espace

On considère des machines avec un ruban d'entrée en lecture seule, un ruban de sortie en écriture seule (déplacement uniquement vers la droite) et au moins un ruban de travail en lecture/écriture.

Définition : l'espace utilisé par une MT déterministe est le nombre de cases différentes que la machine visite pendant son calcul sur ses rubans de travail (on ne compte pas le ruban d'entrée ni le ruban de sortie). Pour une MTND, il s'agit du nombre maximal de cases visitées au cours d'une exécution du calcul. Un langage L est reconnu par une MTD (resp. MTND) M en espace $f(n)$ si M accepte exactement les mots de L et fonctionne en espace $\leq f(n)$. La classe DSPACE($f(n)$) (resp. NSPACE($f(n)$)) est l'ensemble des langages reconnus par une MTD (resp. MTND) en espace $f(n)$. Si $\mathcal{F}$ est un ensemble de fonctions, alors $\text{DSPACE}(\mathcal{F}) = \cup_{f \in \mathcal{F}} \text{DSPACE}(f(n))$ (resp. $\text{NSPACE}(\mathcal{F}) = \cup_{f \in \mathcal{F}} \text{NSPACE}(f(n))$).

Théorème (de hiérarchie en espace) : soit f une fonction constructible en espace. Alors DSPACE($o(f(n))$) est strictement inclus dans DSPACE($f(n)$). (idem pour NSPACE($f(n)$))

(preuve similaire aux théorèmes de hiérarchie en temps)

Définition :

$$\text{L} = \text{DSPACE}(\log n) \quad \text{et} \quad \text{NL} = \text{NSPACE}(\log n)$$
$$\text{PSPACE} = \text{DSPACE}(n^{O(1)}) \quad \text{et} \quad \text{NPSPACE} = \text{NSPACE}(n^{O(1)})$$

Exemples
– Additionner deux entiers binaires se fait en espace logarithmique par l'algorithme de l'école primaire (il faut juste une case pour la retenue et un compteur pour savoir quel chiffre on est en train de traiter).

– 3-SAT est dans PSPACE : il suffit d'énumérer toutes les instanciations en espace polynomial (on peut p.ex. passer de l'une à l'autre en ajoutant 1 en binaire). Ainsi, NP ⊆ PSPACE (car PSPACE est close par $\leq_m^p$).
– Reachability est dans NL. Entrée : un graphe orienté G donné par sa matrice d'adjacence, et deux sommets s et t. Question : existe-t-il un chemin entre s et t dans G ? Notre MTND part du sommet s. À chaque étape, elle devine le voisin suivant : il suffit de retenir le sommet en cours, ce qui prend un espace $\log n$. Elle accepte si elle arrive en t. Pour ne pas boucler infiniment, à chaque étape on incrémente un compteur et on rejette si le compteur dépasse n (le nombre de sommets).

Proposition : $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$ et $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$ pour $f(n) \geq \log n$.

Preuve : pour simuler une MTND M fonctionnant en temps $f(n)$ par une MTD fonctionnant en espace $f(n)$, il suffit d'énumérer en espace $f(n)$ tous les choix non déterministes et de simuler M pour ces choix.

Simulons maintenant une MTND M fonctionnant en espace $f(n)$ par une MTD fonctionnant en temps $2^{f(n)}$. Pour cela, on considère le graphe des configurations de M : il s'agit du graphe orienté dont les sommets sont toutes les configurations possibles de M et il y a un arc entre c_1 et c_2 s'il y a une transition de c_1 à c_2. Une configuration est donnée par le contenu du ruban, l'état de la machine et la position de la tête. Puisque la taille du mot sur le ruban est $\leq f(n)$, on a $2^{O(f(n))}$ configurations possibles. Décider s'il existe une exécution acceptante revient à décider s'il existe un chemin de la configuration initiale à une configuration acceptante dans le graphe. Pour cela, on peut simplement faire un parcours du graphe en partant de la configuration initiale, et pour connaître les voisins d'un sommet il suffit de consulter la fonction de transition de la machine. Cela se fait en temps polynomial en la taille du graphe, donc en temps $2^{O(f(n))}$.

Corollaire : L ⊆ NL ⊆ P ⊆ NP ⊆ PSPACE ⊆ NPSPACE ⊆ EXP ⊆ NEXP

Soit QBF le problème suivant :
– Entrée : une formule booléenne de la forme $\psi \equiv \exists x_1 \forall x_2 \exists x_3 \dots Q x_n \phi(x_1, \dots, x_n)$ où ϕ est sans quantificateur.
– Question : ψ est-elle vraie ?

Proposition : QBF est PSPACE-complet. (sans démonstration)

Théorème (Savitch, 1970) : Reachability $\in \mathsf{DSPACE}(\log^2 n)$.

Preuve : l'idée de l'algorithme est de tester récursivement s'il existe des chemins de taille $\leq 2^i$ entre deux sommets. Les sommets s et t sont reliés s'il existe un chemin de taille $\leq 2^{\log n}$ entre eux. Pour tester cela, on teste pour chaque sommet z s'il existe un chemin de taille $\leq 2^{\log n - 1}$ entre s et z et s'il existe un chemin de taille $\leq 2^{\log n - 1}$ entre z et t. Puis on continue récursivement.

```
Reach(G, s, t, d) // existe-t-il un chemin entre s et t de taille au plus 2^d ?
  Si s = t alors
    Renvoyer Vrai
  Si d = 0 alors
    Si s et t sont voisins alors
      Renvoyer Vrai
    Sinon Renvoyer Faux
  b := Faux
  Pour tout sommet z faire
    Si Reach(G, s, z, d-1) ET Reach(G, z, t, d-1) alors
      b := Vrai
  Renvoyer b
```

Évaluons la complexité en espace de l'algorithme : il y a $\log n$ appels récursifs. Pour chacun d'entre eux, on énumère les sommets z en réutilisant l'espace, ce qui prend un espace $\log n$. En tout, on utilise un espace $\log^2 n$.

Corollaire : $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}(f(n)^2)$ pour $f(n) \geq \log n$ constructible en espace.

Preuve : soit M une MTND fonctionnant en espace $f(n)$. Comme précédemment, pour décider si M accepte x, il suffit de savoir s'il existe un chemin dans le graphe des configurations de la configuration initiale à une configuration acceptante. Le graphe des configurations a une taille $2^{O(f(n))}$ donc par le théorème de Savitch, on peut tester l'accessibilité en espace $O(f(n)^2)$ (note : pour tester si deux sommets sont voisins il suffit de connaître la fonction de transition de M). On simule donc M dans $\mathsf{DSPACE}(f(n)^2)$.

Corollaire : $\mathsf{NPSPACE} = \mathsf{PSPACE}$. On a aussi $\mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2 n)$.

Question ouverte : $\mathsf{NL} = \mathsf{L}$?

Est-ce que $\mathsf{NL} = \mathsf{coNL}$? Première stratégie qui ne fonctionne pas : simuler toutes les exécutions possibles (i.e. lister toutes les configurations accessibles), mais il y en a trop (ça prend trop d'espace). Deuxième stratégie qui ne fonctionne pas : pour chaque configuration, tester si elle est accessible (on peut faire ça dans NL) ; le problème : si ce n'est pas le cas, on ne sait pas si on a emprunté la bonne branche de calcul. Il nous faut une astuce pour pouvoir "complémenter".

Théorème (Immerman-Szelepscényi, 1987-1988) : il existe une MTND fonctionnant en espace $\log n$ qui, sur l'entrée consistant d'un graphe orienté G et d'un sommet s, calcule le nombre de sommets de G accessibles depuis s (c'est-à-dire que pour tout chemin acceptant, ce nombre est inscrit sur le ruban à la fin du calcul).

Preuve : pour $0 \leq k \leq n$, soit S_k l'ensemble des sommets de G accessibles par un chemin de taille $\leq k$. On veut connaître $|S_n|$, pour cela on va calculer successivement $|S_0|, |S_1|, \ldots, |S_n|$. On a $|S_0| = 1$. Supposons la valeur de $|S_k|$ connue et calculons $|S_{k+1}|$.

Il nous faut d'abord une méthode pour tester si un sommet u est dans S_{k+1}. On va tester s'il existe un sommet de S_k voisin de u : on va lister tous les éléments de S_k. Pour cela on énumère à tour de rôle tous les sommets $u_1, \ldots, u_n$ de G. Pour le sommet u_i, on devine $k-1$ sommets $x_1, \ldots, x_{k-1}$ un par un (on a besoin de garder seulement les deux derniers en mémoire) ; au fur et à mesure, on vérifie que $x_i = x_{i+1}$ ou $E(x_i, x_{i+1})$. Enfin, on vérifie que $s = x_1$ ou $E(s, x_1)$ et que $x_{k-1} = u_i$ ou $E(x_{k-1}, u_i)$. Si ces conditions sont remplies, cela signifie que $u_i \in S_k$ et on incrémente un compteur.

Au cours de l'exécution, on retient aussi si l'un des sommets $u_i \in S_k$ est voisin de u. Mais, à cause des choix non déterministes, tous les chemins de calcul n'ont pas trouvé tous les sommets de S_k et on risque de sous-estimer $|S_{k+1}|$; pour être sûr que le chemin en cours est le bon, à la fin de la procédure on compare le compteur avec $|S_k|$: si c'est égal, on accepte si u est voisin de S_k, sinon on rejette.

La connaissance de $|S_k|$ nous a permis d'éliminer des chemins "incomplets".

Maintenant, pour calculer $|S_{k+1}|$, on énumère à tour de rôle tous les sommets $u_1, \ldots, u_n$ de G (on a besoin de garder seulement le dernier en mémoire). Pour le sommet u_i, on teste par la procédure précédente s'il appartient à S_{k+1} et on incrémente un compteur dans ce cas.

Puisqu'il nous suffit de conserver $|S_k|$ pour calculer $|S_{k+1}|$, l'espace requis par la procédure est logarithmique (à chaque étape, on doit conserver en mémoire un nombre constant de numéros de sommets et un nombre constant de compteurs binaires qui prennent des valeurs $\leq n$).

Corollaire : $\mathsf{NSPACE}(f(n)) = \mathsf{coNSPACE}(f(n))$ pour $f(n) \geq \log n$ constructible en espace.

Preuve : soit $L \in \mathsf{coNSPACE}(f(n))$ et soit M une MTND pour L. Ainsi, M fonctionne en espace $f(n)$ et $x \in L$ ssi tous les chemins de $M(x)$ sont acceptants. Soit G le graphe des configurations de M : on veut savoir (de manière non déterministe) si aucun chemin rejetant n'est atteint. Pour cela, on lance l'algo précédent pour connaître le nombre A de configurations atteintes, puis pour chaque configuration $c_1, c_2, \ldots$ à tour de rôle, on teste si elle est accessible (on devine n sommets un par un et on teste si x_i est relié à x_{i+1}) ; si le nombre de configurations accessibles est égal à A, on sait qu'on les a toutes parcourues et on rejette ssi l'une d'entre elles était rejetante. Espace requis : on doit stocker un nombre constant de configurations et un nombre constant de compteurs pour compter les configurations, soit $O(f(n))$.

Corollaire : $\mathsf{NL} = \mathsf{coNL}$