Calculabilité et complexité

Support de cours

Exemple d'école	**Classe de langage**	**Reconnu par**	**Engendré par**
a^*b^*	langages rationnels	automates à états finis	grammaire régulière
$\{a^nb^n \mid n \geq 0\}$	langages algébriques	automates à pile	grammaire algébrique
$\{a^nb^nc^n \mid n \geq 0\}$	langages récursifs	machine de Turing	grammaire (générale)

Réf. : Elements of the Theory of Computation
Harry R. Lewis, Christos H. Papadimitriou
éd. Prentice-Hall

Introduction à la calculabilité
Pierre Wolper
éd. Dunod

Introduction to the Theory of Computation
Michael Sipser, MIT
éd. Thomson Course Technology

Introduction to Theory of Computation
Anil Maheshwari, Michiel Smid, School of Computer Science, Carleton University
free textbook

Roman : Gödel Escher Bach, les Brins d'une Guirlande Eternelle
Douglas Hofstadter
éd. Dunod

BD : Logicomix
Apóstolos K. Doxiàdis, Christos Papadimitriou, Alecos Papadatos, Annie Di Donna
éd. Vuibert

Chapitre 4

MACHINES DE TURING

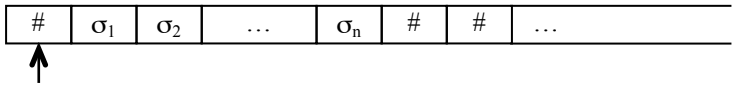
I. Définitions

Une machine de Turing est constituée :
- d'un contrôle (ensemble fini d'états et de transitions),
- d'un ruban infini à droite,
- d'une tête sur le ruban qui peut lire et écrire, et qui peut se déplacer dans les deux directions d'un caractère à la fois.

A chaque étape, en fonction de l'état courant et du symbole courant, la machine :
- change d'état,
- écrit un symbole à l'emplacement courant,
- déplace la tête d'une position, à droite ou à gauche.

Initialement la machine est dans un état initial :
- le mot $w = \sigma_1\sigma_2\ldots\sigma_n$ est dans le ruban, cadré à gauche, avec un blanc devant et une suite infinie de blancs derrière,
- la tête de lecture / écriture pointe sur l'extrémité gauche du ruban,
- le contrôle sur l'état initial.

L'état initial de la machine peut être représenté par le schéma suivant (le symbole # désigne un blanc) :



La machine s'arrête quand elle ne peut plus appliquer de nouvelles transitions.
Si la machine tente de se déplacer trop à gauche (au-delà de l'extrémité gauche du ruban)
→ le traitement se termine anormalement.

Définition
Une machine de Turing standard est un quintuplet $M = (K, \sum, \Gamma, \delta, q_0)$ où :
- K est un ensemble fini d'états,
- $\sum$ est l'alphabet d'entrée,
- Γ est l'alphabet des symboles du ruban,
- δ est la fonction de transition : fonction partielle de $K \times \Gamma$ dans $K \times \Gamma \times \{G, D\}$,
  (les symboles G et D désignent un déplacement élémentaire à gauche ou à droite)
- $q_0 \in K$ est l'état initial.

Remarque
Le symbole qui désigne le blanc (#) n'est pas dans $\sum$, mais appartient à Γ.
$\sum \subset \Gamma$ et Γ peut contenir des symboles utilisés pour écrire sur le ruban.

Soit la transition $\delta(q_i, a) = (q_j, b, G)$.
Cette transition s'applique lorsque :
- la machine est dans l'état courant q_i,
- le symbole courant sur le ruban est a.

Après l'application de cette transition :
- la machine est dans l'état q_j,
- le symbole b est écrit sur le ruban à la place de a,
- la tête de lecture est déplacée d'une position vers la gauche.

Définition
Une configuration associée à une machine de Turing $M = (K, \sum, \Gamma, \delta, q_0)$ est un élément de :
$K \times \Gamma^* \times \Gamma \times (\Gamma^* (\Gamma - \{\#\}) \cup e)$.

Dans une configuration quelconque (q, w_1, a, u_1) :
- la machine est dans l'état courant q,
- w_1 est la partie à gauche de la tête,
- a est le symbole courant,
- u_1 est la partie à droite de la tête jusqu'au premier # (exclu) de la suite infinie de blancs à droite.

Pour simplifier l'écriture des configurations, on introduit une notation abréviée sous la forme :
(état courant, contenu du ruban où le symbole courant est souligné).
Avec cette notation :
- la configuration (q, e, a, bcdf) s'écrit (q, abcdf),
- la configuration (q, ab, #, #f) s'écrit (a, ab##f).

Définition
Soit une machine de Turing $M = (K, \sum, \Gamma, \delta, q_0)$ et deux configurations (q_1, w_1, a_1, u_1) et (q_2, w_2, a_2, u_2).
On dit que (q_1, w_1, a_1, u_1) conduit à (q_2, w_2, a_2, u_2) en une étape ssi :

soit $\delta(q_1, a_1) = (q_2, b, D)$ et $w_2 = w_1 b$ et
- $a_2 = \#$ et $u_2 = e$ si $u_1 = e$
- ou
- $a_2 u_2 = u_1$ si $u_1 \neq e$

soit $\delta(q_1, a_1) = (q_2, b, G)$ et $w_2 a_2 = w_1$ et
- $u_2 = b u_1$ si $b \neq \#$ ou $u_1 \neq e$
- ou
- $u_2 = e$ si $b = \#$ et $u_1 = e$

On note cette relation $(q_1, w_1, a_1, u_1) \vdash_M (q_2, w_2, a_2, u_2)$.

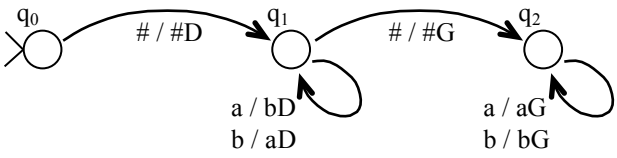
Définition
La relation $\vdash_M^*$ est la fermeture réflexive transitive de la relation $\vdash_M$.

Exemple
Soit la machine de Turing $M = (K, \sum, \Gamma, \delta, q_0)$ où :
- $K = \{q_0, q_1, q_2\}$,
- $\sum = \{a, b\}$,
- $\Gamma = \{a, b, \#\}$

δ	#	a	b
q_0	$(q_1, \#, D)$		
q_1	$(q_2, \#, G)$	(q_1, b, D)	(q_1, a, D)
q_2		(q_2, a, G)	(q_2, b, G)

(δ : fonction partielle)

Représentation graphique de M :



(pas d'état final !)

Les machines de Turing peuvent être utilisées :
- soit pour reconnaître (ou accepter) un langage,
- soit pour calculer une fonction.

II. Les machines de Turing pour reconnaître un langage

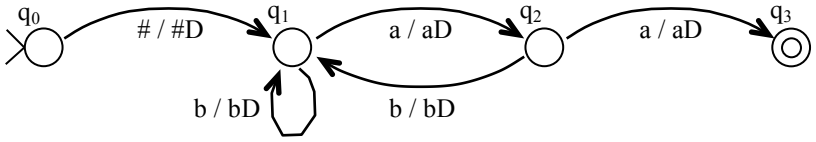
Dans ce contexte, il faut modifier le concept de machine de Turing standard introduit au paragraphe précédent :
→ ajouter la notion d'état final.

Une machine de Turing augmentée avec des états finaux est le sextuplet $M = (K, \sum, \Gamma, \delta, q_0, F)$ où :
$F \subseteq K$ est l'ensemble des états finaux.

Définition
Soit M = (K, ∑, Γ, δ, q_0, F) une machine de Turing.
Une chaîne w ∈ ∑* est acceptée par M ssi :
$(q_0, \underline{\#}w) \vdash_M^* (q_f, w'\underline{a}w'')$ où : - $q_f \in F$, | - $a \in \Gamma$,
- $w', w'' \in \Gamma^*$, | - $\delta(q_f, a)$ n'est pas défini.

Définition
Le langage accepté par M, noté L(M), est l'ensemble de toutes les chaînes acceptées par M.

Exemple
Soit la machine de Turing décrite par le graphe de transitions suivant :



La suite de configurations associée au mot aabb est :
$(q_0, \underline{\#}aabb) \vdash_M (q_1, \#\underline{a}abb) \vdash_M (q_2, \#a\underline{a}bb) \vdash_M (q_3, \#aa\underline{b}b)$
Comme l'état q_3 est final, et qu'il n'y a pas de transition depuis q_3, le mot w = aabb est accepté.
Pour tout mot w ne contenant par aa, le calcul s'arrête sur le premier # à droite de w sur le ruban dans un état non final.

Définition
Le langage accepté par une machine de Turing est dit Turing-acceptable ou récursivement énumérable.

Si la machine de Turing s'arrête sur toutes les entrées possibles (c-à-d pour tous les mots w, w ∈ L ou w ∉ L), alors le langage est dit Turing-décidable ou récursif.
On dit que M semi-décide L, ou encore M accepte L.

On a alors :
∀ w ∈ L, $(q_0, \underline{\#}w) \vdash_M^* (q_f, \underline{\#}Y)$ } $q_f \in F$ → YES (accepté)
∀ w ∉ L, $(q_0, \underline{\#}w) \vdash_M^* (q_f, \underline{\#}N)$ } $q_f \in F$ → NO (rejeté)

III. Les machines de Turing pour calculer des fonctions

L'idée est d'utiliser les machines de Turing pour calculer des fonctions de chaînes vers chaînes.

Définition
Soient $\sum_0$ et $\sum_1$ deux alphabets ne contenant pas le symbole blanc (#).
Soit f une fonction de $\sum_0^*$ vers $\sum_1^*$.
Une machine de Turing M = (K, $\sum_0$, Γ, δ, q_0, F) calcule la fonction f ssi :
∀ w ∈ $\sum_0^*$ tel que f(w) = u, on a :
$(q_0, \underline{\#}w) \vdash_M^* (q_f, \underline{\#}u)$ où : - $q_f \in F$,
- $\delta(q_f, \#)$ n'est pas défini.

Lorsqu'une telle machine de Turing existe, la fonction est dite Turing-calculable.

La notion de Turing-calculable n'est pas restreinte aux fonctions de chaînes vers chaînes.
→ elle peut être étendue de plusieurs façons :

Extension 1 pour des fonctions avec un nombre quelconque d'arguments de la forme f : $(\sum_0^*)^k \rightarrow \sum_1^*$:

Définition
Une machine de Turing M = (K, $\sum_0$, Γ, δ, q_0, F) calcule la fonction f ssi :
∀ $\sigma_1, \sigma_2, \ldots, \sigma_k \in \sum_0^*$ tels que $f(\sigma_1, \sigma_2, \ldots, \sigma_k) = u$, on a :
$(q_0, \underline{\#}\sigma_1\#\sigma_2\#...\#\sigma_k) \vdash_M^* (q_f, \underline{\#}u)$ où : - $q_f \in F$,
- $\delta(q_f, \#)$ n'est pas défini.

Extension 2 pour des fonctions de N dans N :

Notons I un symbole fixé différent de #.
Tout entier naturel n peut être représenté par la chaîne I^n en notation unaire
(dans ce cas l'entier zéro est représenté par la chaîne vide)

Définition
Une fonction f : N → N est calculée par une machine de Turing M,
si M calcule la fonction f' : {I}* → {I}* telle que $f'(I^n) = I^{f(n)}$ pour tout n ∈ N.

Exemple
La fonction successeur définie par succ(n) = n + 1, ∀ n ≥ 0, est calculée par la machine de Turing
M = (K, ∑, Γ, δ, q_0, F) où :
- K = {q_0, q_1, q_2},
- ∑ = {I},
- Γ = {I, #},
- F = {q_2}

δ	#	I
q_0	(q_1, #, D)	
q_1	(q_2, I, G)	(q_1, I, D)
q_2		(q_2, I, G)

IV. Combinaison des machines de Turing

On présente ici une méthode pour combiner des machines de Turing simples → machines plus complexes
⇒ machine de Turing = module ou sous-routine pour faciliter la conception.

Deux types de machines de base :

1. Les machines qui écrivent un symbole

Une machine pour chaque symbole de l'alphabet Γ.
Une telle machine :
- écrit le symbole spécifié sur le symbole courant (dont le contenu est ignoré),
- et s'arrête sans bouger la tête.

Elle est simplement appelée *a* si *a* est le symbole spécifié.

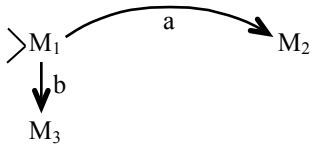
2. Les machines qui déplacent la tête d'une position

Il existe deux machines de ce type :
- G = ({q_0, q_f}, ∑, ∑ ∪ {#}, δ_G, q_0, {q_f}) avec $\delta_G(q_0, \sigma) = (q_f, \sigma, G)$, ∀ σ ∈ Γ,
- D = ({q_0, q_f}, ∑, ∑ ∪ {#}, δ_D, q_0, {q_f}) avec $\delta_D(q_0, \sigma) = (q_f, \sigma, D)$, ∀ σ ∈ Γ.

Ces deux types de machines peuvent être connectées entre elles en utilisant des règles de combinaisons.

Exemple
Soient M_1, M_2, M_3 des machines de Turing quelconques.

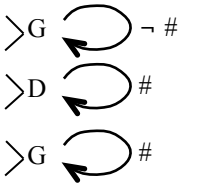
M_1 travaille à partir de son état initial.
Quand M_1 s'arrête, M_2 commence à fonctionner à partir de son état initial, quel que soit le caractère courant.



Lorsque M_1 s'arrête, si le symbole courant est a, alors M_2 s'exécute, si c'est b alors c'est M_3 qui s'exécute.
Si le symbole courant est différent de a et b, alors la machine globale s'arrête.

Dans la suite on utilisera les notations suivantes :

noté $D_{\#}$ déplace la tête à droite du symbole courant jusqu'au premier blanc.

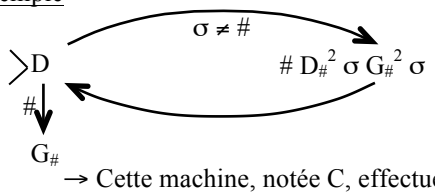


noté $G_{\#}$ déplace la tête à gauche du symbole courant jusqu'au premier blanc.

noté $D\neg_{\#}$ déplace la tête à droite du symbole courant jusqu'au premier non blanc.

noté $G\neg_{\#}$ déplace la tête à gauche du symbole courant jusqu'au premier non blanc.

Exemple



→ Cette machine, notée C, effectue une copie : $(q_0, \underline{\#}\sigma_1 \ldots \sigma_n) \vdash_C^* (q_f, \underline{\#}\sigma_1 \ldots \sigma_n \# \sigma_1 \ldots \sigma_n)$

V. Extension des machines de Turing

Est-ce qu'en étendant le modèle des machines de Turing, la classe des langages acceptés et / ou des fonctions calculées est étendue ?

Pour cela nous considérons successivement plusieurs machines de Turing et nous montrons que dans chaque cas la classe des langages Turing-acceptables et des fonctions Turing-calculables est inchangée.

Les extensions considérées sont :
(a) un ruban infini dans les deux directions,
(b) plusieurs rubans,
(c) plusieurs têtes sur le ruban,
(d) un ruban bidimensionnel,
(e) le non-déterminisme,
(f) l'accès aléatoire.

Pour chaque type d'extension, nous montrons que l'opération de la machine étendue peut être simulée par une machine de Turing normale.
La démonstration consiste dans chaque cas :
(1) à montrer comment construire une machine normale à partir de la machine étendue considérée,
(2) à prouver que la machine normale construite simule correctement le comportement de la machine de départ.

A. Machine à ruban infini dans les deux sens

Soit une machine de Turing $M = (K, \sum, \Gamma, \delta, q_0, F)$ dont le ruban n'a pas de borne à gauche :
- la chaîne d'entrée peut se trouver n'importe où sur le ruban,
- la tête pointe sur le premier blanc à gauche de la chaîne.

Dans cette machine, une configuration est de la forme : $(q, w\underline{a}u)$ avec $q \in K$, $w, u \in \Gamma^*$, $a \in \Gamma$, où :
- w ne commence pas par un blanc
- u ne finit pas par un blanc.

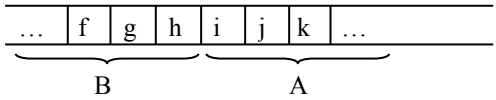
On étend la relation entre configurations pour prendre en compte les déplacements à gauche :
si $\delta(q, a) = (p, b, G)$ alors $(q, \underline{a}u) \vdash_M (p, \underline{\#}bu)$.

Montrons qu'une machine M avec ruban infini dans les deux sens n'est pas plus puissante qu'une machine normale (dans le sens qu'elle ne permet pas de reconnaître plus de langages, ou calculer plus de fonctions).
Pour cela montrons comment construire une machine $M' = (K', \sum, \Gamma', \delta', q_0', F')$, à partir de M et qui simule M :
- si M s'arrête sur un mot w, alors M' s'arrête sur ce même mot w,
- si M ne s'arrête pas sur un mot w, alors M' ne s'arrête pas non plus sur ce même mot w.

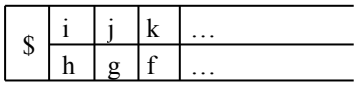
Pour simuler le ruban doublement infini de M dans celui de M', on définit pour M' un ruban à 2 pistes :
Ce ruban est obtenu en coupant en 2 celui de M de façon arbitraire.

Exemple

Ruban de M :



Ruban de M' :



Principe :
- quand M travaille sur A, M' travaille sur la piste du haut,
- quand M travaille sur B, M' travaille sur la piste du bas.

La présence du marqueur $ permet de savoir s'il faut changer de piste.

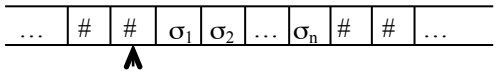
Initialement, M' reçoit sur son ruban normal la chaîne à traiter (infinie dans les deux sens).
→ M' découpe alors son ruban en deux pistes pour simuler le comportement de M.
A la fin (fin de la simulation de M) :
→ M' doit restaurer un ruban normal.

D'où $\Gamma' = \{\$\} \cup \Gamma \cup (\Gamma \times \Gamma) \cup (\neg\Gamma \times \Gamma) \cup (\Gamma \times \neg\Gamma)$ où $\neg\Gamma = \{\neg\alpha \mid \alpha \in \Gamma\}$
(Les symboles $\neg\alpha$ servent à restaurer le ruban de M'.)

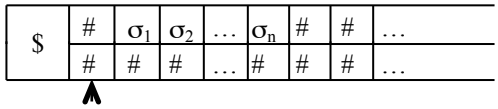
M' simule M de la façon suivante :

(1) M' commence par diviser le ruban en deux pistes et copie la chaîne de départ sur la piste du haut en plaçant un marqueur $ à gauche de la façon suivante :

Ruban de M :



Ruban de M' :



Cette partie d'initialisation du fonctionnement de M' est réalisée par une machine de Turing normale M_i.

(2) M' simule M sur le ruban divisé. Cette partie du traitement est réalisée par la machine M_1' (détail plus loin).

(3) Quand M_1' s'arrête de simuler M (car M se serait arrêtée), M' restaure le ruban sous forme normale.
Cette partie de terminaison du fonctionnement de M' est réalisée par M_f.

La machine M' est donc définie par $M_i\ M_1'\ M_f$.

Description de M_1'

Pour chaque état q de M, il y a deux copies dans M_1' : <q, 1> et <q, 2> :
- <q, 1> signifie que M_1' travaille sur la piste du haut,
- <q, 2> signifie que M_1' travaille sur la piste du bas.

Quand M_1' s'arrête, la machine M_f doit savoir sur quelle piste M_1' travaillait
→ Nécessité de distinguer le symbole pointé (celui de la piste du haut ou celui de la piste du bas ?)
⇒ On le marque d'une barre : α devient $\neg\alpha$.

On a alors $M_1' = (K_1', \sum, \Gamma', \delta_1', F_1')$ où :
- $K_1' = (K \cup \{q_f\}) \times \{1, 2\}$ avec $q_f \notin K$
- $F_1' = \{q_f\} \times \{1, 2\}$ → piste du haut, → piste du bas
- δ_1' est définie de la façons suivante, avec $q \in K$ et $(\alpha_1, \alpha_2) \in \Gamma \times \Gamma$:

(a) Simulation de M sur la piste du haut.
- Si $\delta(q, \alpha_1) = (p, b, \mathbf{D})$ alors $\delta_1'(<q, 1>, (\alpha_1, \alpha_2)) = (<p, 1>, (b, \alpha_2), \mathbf{D})$.
- Si $\delta(q, \alpha_1) = (p, b, \mathbf{G})$ alors $\delta_1'(<q, 1>, (\alpha_1, \alpha_2)) = (<p, 1>, (b, \alpha_2), \mathbf{G})$.

Dans ce cas, seule la piste du haut est modifiée.

(b) Simulation de M sur la piste du bas.

- Si $\delta(q, \alpha_2) = (p, b, \mathbf{D})$ alors $\delta_1'(<q, 2>, (\alpha_1, \alpha_2)) = (<p, 2>, (\alpha_1, b), \mathbf{G})$.
- Si $\delta(q, \alpha_2) = (p, b, \mathbf{G})$ alors $\delta_1'(<q, 2>, (\alpha_1, \alpha_2)) = (<p, 2>, (\alpha_1, b), \mathbf{D})$.

Dans ce cas, en raison du repliement du ruban de M, le déplacement dans M sur la partie B du ruban se traduit par un déplacement dans la direction opposée sur le ruban de M_1'.

(c) Changement de piste.
Pour tout $q \in K$ on a :

- $\delta_1'(<q, 1>, \$) = (<q, 2>, \$, D)$
- $\delta_1'(<q, 2>, \$) = (<q, 1>, \$, D)$

(d) Extension du ruban à droite.
Pour tout $q \in K$ on a :

- $\delta_1'(<q, 1>, \#) = (<q, 1>, (\#, \#), S)$,
- $\delta_1'(<q, 2>, \#) = (<q, 2>, (\#, \#), S)$,

où S (pour *Stay*) signifie que la tête ne se déplace pas.
Cela exprime le fait que lorsque M pointe sur un blanc ordinaire (#)
→ il faut le transformer dans M_1' en un couple (#, #).

(e) Mémorisation de la position de la tête quand M_1' s'arrête.
Pour tout $q \in \mathbf{F}$ on a :

- $\delta_1'(<q, 1>, (\alpha_1, \alpha_2)) = (<q_f, 1>, (\neg\alpha_1, \alpha_2), S)$,
- $\delta_1'(<q, 2>, (\alpha_1, \alpha_2)) = (<q_f, 2>, (\alpha_1, \neg\alpha_2), S)$.

Quand la machine définie par M_i M_1' s'arrête, le ruban est de la forme :

\$	σ_{k+1}	…	σ_{2k-1}	σ_{2k}	#	#	…
	σ_k	…	σ_2	σ_1			

avec $\sigma_1\sigma_2\ldots\sigma_{2k} = \#...\#\ \alpha_1\ldots\alpha_{i-1}\ \neg\alpha_i\ \alpha_{i+1}\ldots\alpha_n\ \#...\#$

La machine de terminaison M_f commence alors à travailler en transformant d'abord le ruban sous forme :

\$	α_1	…	$\neg\alpha_i$	…	α_n	#	…
	#	…	#	…	#		

Puis sous la forme :

#	α_1	…	α_i	…	α_n	#	…
			▲				

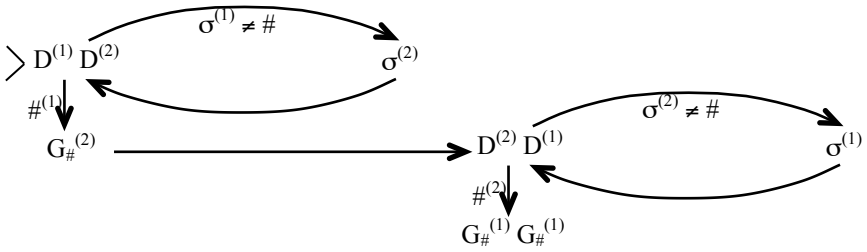
Ces explications décrivent comment construire une Machine de Turing normale M' à partir d'une machine M à ruban infini dans les deux sens.

B. Machine à plusieurs rubans

Une machine de Turing étendue peut être caractérisée par k rubans, chaque ruban étant munie d'une tête autonome.
Une telle machine respecte les conventions :

- La chaîne d'entrée est initialement placée sur le premier ruban, cadrée à gauche et précédée d'un blanc, avec la tête sur ce blanc.
- Les autres rubans sont remplis de blancs avec la tête à l'extrême gauche.
- Lorsque la machine s'arrête, la chaîne de sortie se trouve sur le premier ruban, les autres rubans sont ignorées.

Exemple
La machine à copier C peut être décrite par une machine à deux rubans :



L'exposant (1) ou (2) indique qu'on se trouve sur le premier ou le deuxième ruban.

C. Machines à plusieurs têtes

Machine étendue avec plusieurs têtes sur le même ruban.
→ Simplifier la construction de certaines machines
⇒ il est possible d'implanter une machine à copier avec 2 têtes.

D. Machines de Turing multi dimensionnelles

Pour une machine bidimensionnelle, par exemple, on n'a pas de ruban, mais un plan (discret).
Il faut donc tenir compte des mouvements : D, G, H, B mais aussi les déplacements en diagonale.

E. Machines de Turing à mémoire à accès direct (*Random Access*)

(Description succincte)

Une telle machine comporte :
- un ruban à accès direct (= mémoire) : on peut accéder à chaque case en une étape, contrairement au ruban d'une machine de Turing qui est à accès séquentiel,
- un ensemble de registres,
- un compteur de programme (qui est un registre particulier).

On définit des instructions. Par exemple :
- read j : placer dans le 1er registre le contenu de la $R_j^{ème}$ case, R_j étant la valeur du $j^{ème}$ registre,
- store j : placer le contenu du 1er registre dans le $j^{ème}$ registre,
- etc. (en tout une quinzaine d'instructions de base).

Ainsi une machine de Turing à accès direct est un coupe M = (k, ∏) où :
- k > 0 est le nombre de registres,
- ∏ est une suite finie d'instructions (le programme).

Une configuration d'une machine M = (k, ∏) est un (k+2)-uplet (m, R_0, …, R_{k-1}, T) où :
- m est le compteur de programme,
- R_j ($0 \le j < k$) est le contenu du $j^{ème}$ registre
- T est un ensemble de couples d'entiers : $(i,j) \in T$ signifie que la $i^{ème}$ case du ruban contient la valeur j.

Une telle machine peut être simulée par une machine de Turing à plusieurs rubans :
- un ruban pour la mémoire,
- un ruban pour le programme
- un ruban pour chaque registre.

Le contenu de la mémoire est représenté par des paires de mots (adresse, contenu).

La simulation pourrait être la répétition du cycle suivant :

- parcourir le ruban du programme jusqu'à l'instruction correspondant à la valeur trouvée dans le compteur de programme,
- lire et décoder l'instruction,
- exécuter l'instruction → modifications éventuelles des rubans correspondant à la mémoire et / ou aux registres,
- incrémenter le compteur de programme.

F. Machines de Turing universelles

Existe-t-il une machine de Turing qui peut simuler n'importe quelle machine de Turing ?
Le but est de construire une machine de Turing M à laquelle on fournit :

- la description d'une machine de Turing quelconque M'
- un mot w

et qui simule l'exécution de M sur w.
En clair construire une machine de Turing qui serait un interpréteur de machines de Turing…

Ces machines de Turing existent et sont appelées machines de Turing universelles.

G. Machines de Turing non déterministes

Une machine de Turing non déterministe a la caractéristique suivante :
Pour un état et un symbole courant, il peut y avoir plusieurs choix de comportements possibles.
Une telle machine est décrite par $M = (K, \sum, \Gamma, \Delta, q_0, F)$ où :

Δ est un sous-ensemble de $K \times \Gamma \times K \times \Gamma \times \{G \cup D\}$.

(Machine de Turing classique : δ est une fonction partielle de $K \times \Gamma$ dans $K \times \Gamma \times \{G \cup D\}$.)

Ainsi une machine de Turing non déterministe peut produire deux sorties différentes pour une même entrée.
→ Une machine non déterministe est donc un accepteur dont le seul résultat qui nous intéresse est de savoir si la machine s'arrête ou non, sans considérer le contenu du ruban.

Le non déterminisme n'apporte aucune puissance supplémentaire. En effet, pour toute machine de Turing non déterministe M, on peut construire une machine normale M' telle que pour toute chaîne $w \in \sum^*$ on a:

- si M s'arrête avec w en entrée, alors M' s'arrête sur w,
- si M ne s'arrête pas sur l'entrée w, alors M' ne s'arrête pas non plus sur w.

Théorème

> Tout langage accepté par une machine de Turing non déterministe est accepté par une machine de Turing déterministe

VI. Grammaires (générales)

Définition

Une grammaire générale est un quadruplet $G = (V, \sum, R, S)$ où :

- V : symboles non terminaux
- $\sum$: symboles terminaux ($V \cap \sum = \varnothing$)
- $S \in V$: symbole de départ
- R est l'ensemble de règles : sous ensemble fini de $\underbrace{(V \cup \sum)^* V (V \cup \sum)^*}_{\text{au moins un non-terminal}} \times (V \cup \sum)^*$

(Dans une grammaire algébrique, $R \subset V \times (V \cup \sum)^*$.)

Exemple

G = (V, ∑, R, S) où :
- V = { S, A, B, C, T_a, T_b, T_c }
- ∑ = { a, b, c }
- R = { S → ABCS,
  S → T_c,
  CA → AC,
  BA → AB,
  CB → BC,
  CT_c → T_cc,
  CT_c → T_bc,
  BT_b → T_bb,
  BT_b → T_ab,
  AT_a → T_aa,
  T_a → e }

Théorème

> Un langage L est engendré par une grammaire générale ssi il est récursivement énumérable.
> (c-à-d accepté par une machine de Turing)

Définition (calculabilité grammaticale)

• Soit G = (V, ∑, R, S) une grammaire et f : ∑* → ∑* une fonction.
On dit que G calcule f si ∀ w, v ∈ ∑* on a :
SwS $\Rightarrow_G^*$ v ssi v = f(w)
(c-à-d toute dérivation par G de SwS donne v)
• Une fonction f : ∑* → ∑* est grammaticalement calculable ssi il existe une grammaire la calculant.

En adaptant le théorème précédent, on a :

Théorème

> Une fonction f : ∑* → ∑* est récursive (Turing-calculable) ssi elle est grammaticalement calculable

VII. Fonctions numériques

Définition (fonctions primitives récursives)

Fonctions de base
(k ≥ 0)
(a) $zéro_k$: N^k → N, définie par
∀ $n_1, \ldots, n_k \in N$, $zéro_k(n_1, \ldots, n_k) = 0$.
(b) $j^{ème}$ k-projecteur ($j^{ème}$ k-identité) : $id_{k,j}$: N^k → N
∀ $n_1, \ldots, n_k \in N$, $id_{k,j}(n_1, \ldots, n_k) = n_j$ (pour $1 \le j \le k$)
(c) successeur : ∀ n ∈ N, succ(n) = n+1

Opérations sur les fonctions
(1) composition :
g : N^k → N
$h_1, \ldots, h_k$: N^p → N
Fonction f : composée de g avec $h_1, \ldots, h_k$
f : N^p → N
$f(n_1, \ldots, n_p) = g(h_1(n_1, \ldots, n_p), \ldots, h_k(n_1, \ldots, n_p))$

(2) récursivité :

$g : N^k \to N$

$h : N^{k+2} \to N$

Fonction f : définie récursivement par g et h :

$f : N^{k+1} \to N$

$f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$

$f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))$

Fonctions primitives récursives

Ensemble de fonctions de $N^k \to N$ (pour tout $k \in N$) pouvant être définies :

- à partir des fonctions de base,
- à l'aide des opérateurs composition et récursivité.

Exemple

plus : $N^2 \to N$

$(n,m) \mapsto n + m$

$plus(n, 0) = n = \overbrace{id_{1,1}}^{g}(n)$

$plus(n, m+1) = succ(plus(n, m))$

$= \underbrace{succ(id_{3,3}}_{h}(n, m, plus(n, m)))$

Exemple de prédicat primitif récursif

$iszéro(0) = 1$

$iszéro(n+1) = 0$ (1 signifie vrai, 0 signifie faux)

Définition par cas :

$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n) \text{ si } p(x_1, \ldots, x_n) \\ g_2(x_1, \ldots, x_n) \text{ sinon} \end{cases}$

Définition (minimisation d'une fonction)

Soit g une fonction (k+1)-aire, pour un certain $k \geq 0$.

La minimisation de g est la fonction $f : N^k \to N$ définie par

$f(n_1, \ldots, n_k) = \begin{cases} \text{le plus petit m (s'il existe) tel que } g(n_1, \ldots, n_k, m) = 1 \\ 0 \text{ sinon} \end{cases}$

(= VRAI si g est une fonction booléenne)

Cette minimisation n'est pas toujours une fonction calculable :

L'algorithme

m := 0

tant que $g(n_1, \ldots, n_k, m) \neq 1$ faire m := m+1 fait

retourner m

peut ne pas se terminer.

On dit qu'une fonction g est minimisable si sa minimisation est calculable par l'algorithme précédent, c-à-d ssi :

$\forall\, n_1, \ldots, n_k \in N, \exists\, m \in N$ tel que $g(n_1, \ldots, n_k, m) = 1$

On note alors $\mu\, m[g(n_1, \ldots, n_k, m)] = \begin{cases} \text{le plus petit m (s'il existe) tel que } g(n_1, \ldots, n_k, m) = 1 \\ 0 \text{ sinon} \end{cases}$

Définition

Les fonctions μ-récursives sont les fonctions obtenues à partir :

- des fonctions de base
- des opérations de composition et de récursivité
- de la minimisation pour les fonctions minimisables

Exemple

Si $Log(m, n) = \lceil \log_{m+2}(n+1) \rceil$ Alors on a $Log(m, n) = \mu\, p[(m+2)\uparrow p \geq n+1]$

Théorème (équivalence μ-récursive et récursive)

Une fonction $f : N^k \to N$ est μ-récursive ssi elle est récursive (= Turing-calculable).

Chapitre 5

INDECIDABILITE

I. Thèse de Church-Turing

Thèse de Church-Turing

Formulation Lewis – Papadimitriou

> Un algorithme est une machine de Turing qui s'arrête pour toutes ses entrées.
> *We therefore propose to adapt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm".*

Formulation Wolper

> Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing.

Ca n'est pas un théorème : la notion d'algorithme / de procédure effective est intuitive *a priori* et formalisée par cette thèse. Adopter cette thèse revient à choisir une modélisation du concept de procédure effective.

Comme ça n'est pas un théorème, on ne peut donc pas le prouver. Mais on pourrait éventuellement le contredire, en exhibant une notion d'algorithme plus générale. Jusqu'à présent toutes les notions d'algorithmes produites par l'homme sont équivalentes à celle-ci. Personne ne pense que c'est possible de contredire cette thèse.

II. Machines de Turing universelles

Une machine de Turing universelle est une machine de Turing capable de simuler :
- le comportement de n'importe quelle machine de Turing,
- avec n'importe quelle entrée.

U("M" "w") = "M(w)"
" " : codage de la machine M dans le langage de la machine U
$\sum, K$

Principe du codage :
Δ_U : interne, vu plus bas
$\sum_U$: on doit coder une machine normale dans U → $\sum_U$ = {#, a, q, 0, 1, (,), ,}

M = (K, $\sum$, Γ, δ, s, H)
$\sum$ → $\#_M$, D_M, G_M : a…… (nombre en binaire)

$\#_M$: a00…000
D_M : a00…001
G_M : a00…010 ($|\sum| + 2$)

K → q_1 : q00…000
q_2 : q00…001
… } même principe

Une machine de Turing est alors représentée par sa table de transitions :
"M" : suite de chaînes ("q","a","p","b","s") rangées dans l'ordre lexicographique croissant

Exemple

Soit la machine de Turing M = (K, $\sum$, Γ, δ, q_0) où :
- K = {q_0, q_1, q_2},
- $\sum$ = {a, b},
- Γ = {a, b, #}

δ	#	a	b
q_0	(q_1, #, D)		
q_1	(q_2, #, G)	(q_1, b, D)	(q_1, a, D)
q_2		(q_2, a, G)	(q_2, b, G)

Codage : q_0 : q000
q_1 : q001

q_2 : q010
: a000 par exemple, la chaîne #aba est représentée par :
a : a001 a000a001a010a001
b : a010
D : a011
G : a100

"M" est représentée par la chaîne :
"M" = (q000, a000, q001, a000, a011),
(q001, a000, q010, a010, a100), (q001, a001, q001, a010, a011), (q001, a010, q001, a001, a011),
(q010, a001, q010, a001, a100), (q010, a010, q010, a010, a100) } δ

Fonctionnement de U

U : machine à trois rubans :

#	codage du ruban de M
#	codage de M
#	codage de l'état courant de M

Début : "M" et "w" sur le premier ruban.

1ère étape : "w" sur le premier ruban
"M" sur le second
"s" sur le troisième

2ème étape :
Recherche sur le 2ème ruban de la transition de M
(état sur la 3ème)
(lettre courante : position de la tête sur le 1er ruban)

Si il existe une transition
Exécution de la transition :
- manipulation du 1er ruban
→ changer le caractère courant
→ déplacer la tête vers la droite ou vers la gauche
- écriture du nouvel état sur le 3ème ruban

Sinon arrêt.

III. Problème de l'arrêt

Problème des énoncés "autoréférants" (principe de la diagonale)

Exemple

Le barbier rase tous les hommes qui ne se rasent pas eux-mêmes et uniquement ceux-là.
Qui rase le barbier ?
- lui-même ? non parce qu'il ne rase que les hommes ne pouvant pas se raser,
- un autre ? non parce qu'alors il ne peut pas se raser et il rase tous les hommes qui ne peuvent pas se raser.

On pose H = {"M" "w" | M est une MT, w est un mot de $\sum^*$, et M s'arrête avec w en entrée}

Propriété

H est récursivement énumérable : c'est le langage accepté par une MT universelle U
(qui boucle lorsque l'entrée n'est pas valide)

Théorème

- Le langage H n'est pas récursif.
- Il existe des langages récursivement énumérables qui ne sont pas récursifs.
- La classe des langages récursivement énumérables n'est pas stable par complément.
  (Sinon récursivement énumérable serait équivalent à récursif, c'est qui n'est pas le cas.)

IV. Problèmes indécidables à propos des machines de Turing

Indécidabilité d'un problème

≡ il n'existe pas d'algorithme résolvant ce problème de manière générale
≡ il n'existe pas de machine de Turing qui prend en entrée les données (langage)
et qui s'arrête toujours en donnant une solution.

Exemple
Le problème de l'arrêt pour les machines de Turing est indécidable.

Définition (réduction)
Soient L_1 et $L_2 \subset \Sigma^*$ deux langages.
Une réduction de L_1 à L_2 est une fonction récursive $\rho : \Sigma^* \rightarrow \Sigma^*$ telle que :
$w \in L_1$ ssi $\rho(w) \in L_2$.

Théorème

Si L_1 n'est pas récursif et s'il existe une réduction de L_1 à L_2, alors L_2 n'est pas récursif non plus.

Autrement dit, pour montrer qu'un langage L est non récursif,
il suffit de trouver L' non récursif tel que L' se réduit à L (dans ce sens).

Théorème (problème indécidables avec les machines de Turing)

Les problèmes suivants sont indécidables :
(a) Soient M une MT et $w \in \sum^*$. Est-ce que M s'arrête sur w ?
(b) Soit M une MT. Est-ce que M s'arrête sur le mot vide en entrée ?
(c) Soit M une MT. Y a-t-il des mots pour lesquels M S'arrête ?
(d) Soit M une MT. Est-ce que M s'arrête pour toutes les entrées possibles ?
(e) Soient M_1 et M_2 deux MT. Est-ce que M_1 et M_2 s'arrêtent pour les mêmes entrées ?
(f) Soit M une MT, qui semi-décide (= accepte) L. Est-ce que L est rationnel ? algébrique ? récursif ?
(g) Il existe une machine M pour laquelle le problème suivant est indécidable :
Soit $w \in \sum^*$. Est-ce que M s'arrête sur w ?

V. Problèmes indécidables à propos des grammaires

Théorème (grammaires générales)

Les problèmes suivants sont indécidables :
(a) Soient G une grammaire (générale) et $w \in \sum^*$. Est-ce que $w \in L(G)$?
(b) Soit G une grammaire. Est-ce que $e \in L(G)$?
(c) Soient G_1 et G_2 deux grammaires. Est-ce que $L(G_1) = L(G_2)$?
(d) Soit G une grammaire. Est-ce que $L(G) = \emptyset$?
(e) Il existe une grammaire G_0 pour laquelle le problème suivant est indécidable :
Soit $w \in \sum^*$. Est-ce que $w \in L(G_0)$?
(G_0 : grammaire universelle.)

Théorème

> Les problèmes suivants sont indécidables :
> (a) Soit G une grammaire algébrique. Est-ce que $L(G) = \sum^*$?
> (b) Soient G_1 et G_2 deux grammaires algébriques. Est-ce que $L(G_1) = L(G_2)$?
> (b') Soient M_1 et M_2 deux automates à pile. Est-ce que $L(M_1) = L(M_2)$?
> (c) Soit M un automate à pile. Trouver un automate à pile équivalent minimal en nombre d'états.

VI. Propriétés des langages récursifs

Rappel : Récursif ⇒ récursivement énumérable
(Réciproque fausse)

← Récursivement énumérable = accepté par une MT
Récursif = s'arrête pour toutes les entrées

Théorème

> Un langage L est récursif ssi L et ¬L sont récursivement énumérables.

Alternative à récursivement énumérable :

Définition (énumération)

On dit qu'une machine de Turing M énumère le langage L ssi pour un certain état fixé q, on a :
$L = \{ w : (s, \underline{\#}) \vdash_M^* (q, \underline{\#}w) \}$ (q est appelé l'état d'affichage)

Un langage est dit Turing-énumérable ssi il existe une MT qui l'énumère.

Théorème

> Un langage est récursivement énumérable ssi il est Turing-énumérable.

Définition (énumération lexicographique)

Soit M une MT qui énumère un langage L (avec l'état d'affichage q).
On dit que M énumère lexicographiquement L si la relation
$(q, \underline{\#}w) \vdash_M^+ (q, \underline{\#}w')$
entraîne que w' vient après w dans l'ordre lexicographique.

Un langage est lexicographiquement-énumérable ssi il existe une MT qui l'énumère lexicographiquement.

Théorème

> Un langage est récursif ssi il est lexicographiquement-énumérable.

Théorème de Rice

> Soit C une partie propre non vide de la classe des langages récursivement énumérables.
> Le problème suivant est indécidable :
> Soit M une MT. Est-ce que $L(M) \in C$?

Chapitre 6

COMPLEXITE

Différence entre calculabilité (ou décidabilité) et effectivité…

I. La classe P

Exemple : voyageur de commerce
Visite de *n* villes en faisant le moins de km possibles.
Algorithme ?

- produire toutes les permutations de villes possibles (sauf la première qui est toujours la même),
- pour chaque permutation, calculer le trajet.

→ $(n - 1)$! permutations possibles.

Définition

Une machine de Turing $M = (K, \sum, \Gamma, \delta, q_0, F)$ est dite polynomialement bornée s'il existe un polynôme p tel que pour toute entrée w, il n'y a pas de configuration C telle que

$$(q_0, \#w) \vdash_M^{p(|w|+1)} C$$

c'est-à-dire M s'arrête toujours, et ce, en au plus $p(|w|+1)$ étapes.

Un langage est dit polynomialement décidable ssi il existe une machine de Turing polynomialement bornée qui le décide.

La classe des langages polynomialement décidables est notée P.

La thèse de Church-Turing est raffinée en :
Les machines de Turing polynomialement bornées et la classe P correspondent aux notions

- d'algorithmes pratiquement exécutables,
- et de problèmes réellement solvables.

Théorème

> La classe P est stable par complément.

Théorème

> Il existe des langages récursifs non polynomialement décidables.

II. Exemples de problèmes et de complexité

Exemple : existence d'un chemin

Soient un graphe orienté $G \subset V \times V$ ($V = \{v_1, \ldots, n_n\}$) et deux sommets v_i et $v_j \in V$.
Existe-t-il un chemin entre v_i et v_j ?

⇒ Il existe un algorithme en $O(n^3)$: calcul de la fermeture réflexive – transitive.

Exemple : graphes Eulériens

Soit un graphe G.
Existe-t-il un chemin fermé (cycle) dans G qui utilise chaque arc une fois et une seule ?
Un graphe qui contient un tel cycle est dit Eulérien ou unicursal.

Le problème du cycle Eulérien ∈ P.

Exemple : graphes Hamiltoniens

Soit un graphe G.

Existe-t-il un cycle passant par chaque sommet une fois et une seule ?
Un graphe qui contient un tel cycle est dit Hamiltonien.

Algorithme : - examiner toutes les permutations de nœuds possibles (→ exponentiel),
- regarder si le parcours correspondant existe dans G.

Le problème des cycles Hamiltoniens n'est pas connu comme étant dans P.

III. Classe NP

Les problèmes classiques dont on ne sait pas s'ils sont P ou non peuvent être résolus par des machines de Turing non déterministes bornées polynomialement au sens suivant :

Définition

Une machine de Turing non déterministe $M = (K, \sum, \Gamma, \Delta, q_0, F)$ est dite polynomialement bornée s'il existe un polynôme p tel que :

pour toute entrée w, il n'y a pas de configuration C telle que

$$(q_0, \#w) \vdash_M^{p(|w|+1)} C$$

c'est-à-dire M s'arrête toujours, et ce, en au plus p(|w|+1) étapes.

NP est la classe des langages décidés par une machine de Turing non déterministe polynomialement bornée.
(NP pour *Non déterministe polynomialement*)

Proposition

> Le problème SAT est dans NP.
> (Est-ce qu'il existe une assignation booléenne satisfaisant une forme normale conjonctive ?)

Rappel : Logique d'ordre 0 – calcul des propositions :

- Variables booléennes
- Connecteurs (opérations) : ∧, ∨, et ¬
- Un littéral est une expression de la forme p ou ¬p pour une variable propositionnelle p.
- Une clause est une disjonction de variables propositionnelles ou de leur négation :
  $E = x_1 \vee x_2 \vee \dots \vee x_n$ (chaque x_i est un littéral)
- Forme normale conjonctive
  $F = E_1 \wedge E_2 \wedge \dots \wedge E_n$ (chaque E_i est une clause)
  $= \wedge_{i=1}^{n} (\vee_{ji=1}^{Pi} x_{ji})$ avec $x_{ji} = a_{ji}$ ou $\neg a_{ji}$ (atomes, littéraux positifs ou négatifs = clauses)
- Assignation booléenne ou interprétation :
  fonction : $X \rightarrow (T,\perp)$ X : ensemble des variables
  T : vrai, ⊥ : faux
- Satisfaisabilité : ∃ (au moins) une interprétation rendant la formule vraie.

Ennoncé du problème SAT

Soit une forme normale conjonctive F.
F est-elle satisfaisable ?
Algorithme brutal

Faire une table de vérité → exponentiel en fonction du nombre de variables.
Pour chaque assignation, tester la FNC (linéaire)
→ exponentiel sauf dans le cas 2-SAT ((a∨b) ∧ (c∨d) ∧ …) où les clauses ont au plus 2 littéraux : le problème 2-SAT est dans P (dans le sens on connaît un algorithme P) mais la table de vérité, elle, reste de taille exponentielle
Le problème 3-SAT ((a∨b∨c) ∧ (d∨e∨f) ∧ …) est NP (pas d'algorithme P connu).
Le problème SAT (FNC avec clauses avec nombre quelconque de littéraux) est NP.

Proposition

Le *problème* du voyageur de commerce (faible) est dans NP.

Enoncé du problème du voyageur de commerce
Soient un entier $n \geq 2$, une matrice de distance d_{ij} et un entier $B \geq 0$.
(B est le budget du voyageur de commerce.)
Le problème consiste à trouver une permutation π sur $\{1, 2, \ldots, n\}$ telle que $C(\pi) \leq B$, où :

$$C(\pi) = d_{\pi(1)\,\pi(2)} + d_{\pi(2)\,\pi(3)} + \ldots + d_{\pi(n-1)\,\pi(n)} + d_{\pi(n)\,\pi(1)}$$

De manière semblable, des algorithmes de type :
- générer de manière non déterministe une situation,
- tester de manière déterministe cette situation,

peuvent résoudre des problèmes précédents avec des machines de Turing ND polynomialement bornées,
- si le test de la situation se fait avec une machine de Turing (déterministe) polynomialement bornée,
- et si la taille de la situation est bornée polynomialement en fonction des entrées.

Il est clair que $P \subseteq NP$. ⟶ 2-SAT $\in$ P ; 3-SAT $\in$ NP $\Rightarrow$ SAT $\in$ NP } ce qui ne veut pas dire que SAT $\notin$ P…
A-t-on $NP \subseteq P$?
Cela voudrait dire :
1) qu'il existe une MT polynomialement bornée équivalente à une MT ND polynomialement bornée
(Se souvenir du théorème d'équivalence entre MT ND et MT…)
a priori le passage MT ND → MT est exponentiel.
Avec notre construction : MT ND $O(p(k))$ → MT $O(e^{p(k)})$.
2) que les problèmes
- SAT,
- voyageur de commerce,
- cycle de Hamilton,

seraient dans P.

En fait, on ne sait pas si $NP \subseteq P$. (Et donc P = NP.)

Voici une autre classe de langages (moins connue) (en revenant à la notion d'algorithme).
(Rappelons que les MT ND ne correspondent pas à un algorithme.)

Définition (borne exponentielle)

Une MT $M = (K, \sum, \Gamma, \delta, q_0, F)$ est dite exponentiellement bornée s'il existe un polynôme p tel que :
pour toute entrée w, il n'y a pas de configuration C telle que

$$(q_0, \#w) \vdash_M^{2^{p(|w|)}+1} C$$

Cette machine s'arrête toujours, et ce en au plus $2^{p(|w|)}+1$ étapes.

On note EXP la classe des langages qui peuvent être décidés par une MT exponentiellement bornée.

Théorème

Si $L \in NP$, alors $L \in EXP$.
Autrement dit : $NP \subseteq EXP$.

Certificat
Nous avons vu comment fonctionnaient les MT ND polynomialement bornées qui résolvaient les problèmes classiques d'existence.
- production d'une situation
- test → si réussite : on a résolu le problème d'existence,
  → si échec à tous : on a répondu par la négative.

Un mot synthétisant une solution et qui passe le test avec succès est appelé certificat ou témoin.
Pour qu'ils soient d'un intérêt pratique,
- ces certificats doivent avoir une longueur polynomiale vis-à-vis des entrées,
- et de plus on doit pouvoir tester ces certificats en temps polynomial.

Formalisons ce principe et montrons que les langages NP, et seulement ceux-ci, ont des certificats.

Définition
Soient $\sum$ un alphabet et ";" un symbole $\notin \sum$.
Soit L' un langage tel que $L' \subseteq \sum^* ; \sum^*$.
On dit que L' est polynomialement équilibré s'il existe un polynôme p tel que :
si $x;y \in L'$ alors $|y| \leq p(|x|)$.

Théorème

> Soit $L \subseteq \sum^*$ un langage et $\sum$ un alphabet pour lequel ; $\notin \sum$ et $|\sum| \geq 2$.
> Alors $L \in NP$ ssi il existe un langage polynomialement équilibré $L' \subseteq \sum^* ; \sum^*$ tel que :
> $L' \in P$ et $L = \{x \mid \exists\, y \in \sum^* : x ; y \in L'\}$.

IV. NP-complétude

Similarité de comportement entre
- récursif – non récursif
- P – NP

Problème non récursif → problème de l'arrêt.
Y a-t-il ainsi des problèmes qui jouent le rôle d'"étalon" ou de référence dans le domaine de la complexité ?
Oui : les problèmes NP-complets :
→ problèmes intrinsèquement compliqués (les plus compliqués de la classe NP).

Les problèmes NP-complets $\in$ NP.
On a vu que les problèmes de la classe $P \in NP$.
Il a de plus été démontré qu'il existe des problèmes dans NP qui ne sont ni dans P, ni NP-complets.

Théorème

> S'il existe un problème NP-complet décidé par un algorithme polynomial,
> alors tous les problèmes de NP sont décidables en temps polynomial (c-à-d P = NP)

Autrement dit, si les problèmes NP-complets $\in$ P, alors P = NP.

L'analogie avec la décidabilité et les problèmes de référence peut se poursuivre avec la notion de réduction polynomiale (comme on l'avait fait à propos de l'indécidabilité).

Définition
Une fonction $\Sigma^* \rightarrow \Sigma^*$ est dite calculable en temps polynomial ssi il existe une MT bornée polynomialement qui la calcule.
Soient L_1 et $L_2 \subseteq \Sigma^*$. Une réduction polynomiale (ou transformation polynomiale) de L_1 à L_2 est une fonction τ : $\Sigma^* \rightarrow \Sigma^*$, calculable en temps polynomial, telle que :
$x \in L_1$ ssi $\tau(x) \in L_2$

Définition
Un langage L est NP-complet si
- $L \in NP$,
- Pour tout langage $L' \in NP$, il existe une réduction polynomiale de L' dans L.

Le premier point est facile à établir (avec un algorithme non déterministe polynomial).
Le second point est plus délicat.
Mais si on connaît un langage NP-complet L",
il suffit de démontrer qu'il existe une réduction polynomiale de L" dans L.

Exemple

Soient les trois problèmes suivants :

(1) Planification de 2 machines (*2-machine scheduling*) :
n tâches de durées respectives $a_1, a_2, \ldots, a_n$ à répartir sur 2 machines de sorte qu'un deadline D soit respecté.
(deadline = délai de retour)

(2) Problème du sac-à-dos (*Knapsack*) :
Soit un ensemble d'objets avec des poids $S = \{a_1, a_2, \ldots, a_n\}$, et un entier K, le tout donné en binaire.
Trouver un sous-ensemble $P \subseteq S$ tel que $\Sigma_{ai \in P} a_i = K$.

(3) Problème de la partition :
Soit un ensemble de n entiers positifs $S = \{a_1, a_2, \ldots, a_n\}$ représentés en binaire.
Existe-t-il un sous-ensemble $P \subseteq \{1, 2, \ldots, n\}$ tel que $\Sigma_{ai \in P} a_i = \Sigma_{ai \notin P} a_i$?

Ces trois problèmes sont équivalents du point de vue de la complexité polynomiale :
On peut réduire polynomialement chacun d'eux dans les autres.

Lemme

> Si τ_1 est une réduction polynomiale de L_1 vers L_2,
> et τ_2 est une réduction polynomiale de L_2 vers L_3,
> alors $\tau_1 \circ \tau_2$ est une réduction polynomiale de L_1 vers L_3.

On en déduit :

V. Théorème de Cook

Théorème

> Le problème SAT est NP-complet.

Rappel : problème SAT
Soit F une FNC. Existe-t-il une fonction d'interprétation qui rend F vraie ?

Le problème SAT est le premier problème NP-complet prouvé.
La preuve est due à Stephen A. Cook, Canada, en 1971.

On dispose à présent d'un problème NP-complet (dans le sens on l'a démontré).
On va pouvoir s'en servir pour démontrer que d'autres problèmes sont NP-complets par réduction polynomiale.

Théorème

> 3-SAT est NP-complet.

Rappel

3-SAT : satisfaisabilité de FNC comportant exactement 3 littéraux par clause.
(= restriction de SAT)

Preuve

(a) 3-SAT $\in$ NP parce qu'il est une restriction de SAT.

(b) On peut réduire SAT à 3-SAT.
Soit F une FNC avec un nombre quelconque de littéraux dans ses clauses. F est une instance de SAT.
On substitue les clauses de F ne comportant pas 3 littéraux par une conjonction de clauses comportant 3 littéraux.
Cette substitution est une réduction polynomiale de SAT vers 3-SAT.

(1) Une clause $F_1 = (x_1 \vee x_2)$ (2 littéraux) est remplacée par :

$F_1' = (x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$ (y est une nouvelle variable propositionnelle)

Pour toute fonction d'interprétation telle que $F_1 = T$, on a également $F_1' = T$.
Inversement, si par exemple $y = T$, pour que $F_1' = T$, il faut $x_1 = T$ ou $x_2 = T$
(parce que dans la clause de droite, $\neg y = \perp$).

Une fonction d'interprétation telle que $F_1' = T$ comporte donc forcément $x_1 = T$ ou $x_2 = T$,
et donc avec cette fonction, on a aussi $F_1 = T$.

Donc F_1' est satisfaisable ssi F_1 est satisfaisable.

(2) Une clause $F_1 = x$ (1 littéral) est remplacée par :

$F_1' = (x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee \neg y_2) \wedge (x \vee \neg y_1 \vee y_2) \wedge (x \vee \neg y_1 \vee \neg y_2)$
(y_1 et y_2 sont deux nouvelles variables propositionnelles)

De manière similaire à ci-dessus, on peut montrer que F_1' est satisfaisable ssi F_1 est satisfaisable.

(3) Une clause $F_1 = x_1 \vee x_2 \vee \ldots \vee x_n$ ($n > 3$ littéraux) est remplacée par :

$F_1' = (x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$
($y_1 \ldots y_{n-3}$ sont n-3 nouvelles variables propositionnelles)

S'il existe une assignation τ satisfaisant F_1, alors on peut l'étendre pour qu'elle satisfasse F_1' :
on met les y_i à T jusqu'à avoir un x_i à T, on met alors les y_i suivants à $\perp$.

Inversement, si $\tau(F_1) = \perp$ (ce qui veut dire que tous les x_i valent $\perp$),
alors toute interprétation qui étend τ aux y_i rend F_1' fausse.
En effet, pour que F_1' soit vraie avec les y_i, il faudrait que toutes les clauses de F_1' valent T :

$x_1 \vee x_2 \vee y_1 : T$ donc $y_1 = T$ parce que $x_1 = \perp$ et $x_2 = \perp$
$\neg y_1 \vee x_3 \vee y_2 : T$ donc $y_2 = T$ parce que $\neg y_1 = \perp$ et $x_3 = \perp$
…
$\neg y_{n-3} \vee x_{n-1} \vee x_n : T$ $\neg y_{n-3} = \perp$ et $x_{n-1} = \perp$ et $x_n = \perp$
$\Rightarrow$ Contradiction : une des clauses vaut $\perp$ (la dernière ici)

Théorème

MAX-SAT est NP-complet.

Définition

Etant donné un ensemble de clauses et un entier K, existe-t-il une interprétation satisfaisant au moins K clauses ?

Considérations sur la NP-complétude

On a un problème, on cherche un algorithme pour le résoudre, mais ce problème est établi NP-complet.
Comme P ≠ NP (hypothèse), ce problème n'a pas de solution polynomiale.
Faut-il pour autant renoncer à résoudre ce problème ? pas forcément.

La mesure de la complexité est pour le pire des cas :
Pas d'algo polynomial → pas d'algo pour toutes les instances du problème.
Mais il peut très bien exister un algo polynomial pour certains cas, voire presque tous.

Il n'est obligé d'explorer tous les cas (nombre exponentiel de cas) :
On peut utiliser des heuristiques pour limiter le nombre de cas à explorer.
→ critères approximatifs pour découvrir rapidement la solution recherchée.
(Efficace des fois.)

Plutôt que de chercher la solution optimale, on peut chercher une solution s'en approchant.

On peut résoudre un ou plusieurs cas particuliers d'un problème NP-complet, en utilsant des algorithmes polynomiaux.