Calculabilité et Complexité

Rédigé par S. Ali Tokmen, http://ali.tokmen.com

Partie calculabilité: 16 – 20 Mars 2006
Partie complexité: 06 – 08 Mai 2006

Calculabilité

Machines de Turing

Composants

La machine de Turing est un modèle d'éxécution d'un algorithme particulier. C'est une machine à état fini qui a trois composants:

- Un ruban, infini des deux cotés, qui contient des symboles (S)
- Une tête de lecture, qui peut lire, écrire et se deplacer sur le ruban
- Une logique interne, qui stocke l'état (q) et une table de transitions

La table des transitions est formé d'entrées de l'une des trois formes suivantes:

q_1 s_1 s_2 q_2

Ceci veut dire que si la machine est à l'état q_1 et que la tête de lecture lit le symbole s_1, il faut que la tête écrive sur le ruban le symbole s_2 et que la machine aille à l'état q_2.

q_1 s_1 R q_2

Ceci veut dire que si la machine est à l'état q_1 et que la tête de lecture lit le symbole s_1, il faut que la tête aille vers la droite (Right) et la machine aille à l'état q_2.

q_1 s_1 L q_2

Ceci veut dire que si la machine est à l'état q_1 et que la tête de lecture lit le symbole s_1, il faut que la tête aille vers la gauche (Left) et la machine aille à l'état q_2.

Il est important de savoir que la table est consistante: il n'y a donc aucune entrée avec les deux couples (q s) qui se répètent. En d'autres termes, la machine n'a jamais le choix entre deux états, elle est donc déterministe.

Le calcul

Configuration

On apelle une configuration la forme γ q s δ , où:

- γ dénote le mot d'avant
- q dénote l'état courant
- s dénote le symbole observé
- δ dénote le mot d'après

Par exemple: $1\ 2\ 3\ q_7\ 7\ 6\ 9\ 8\ 7\ 9$; où $\gamma = 1\ 2\ 3$

$q = q_7$

$s = 7$

$\delta = 6\ 9\ 8\ 7\ 9$

Le pas de calcul

Soit α la configuration $\gamma\ q_i\ s_j\ \delta$ et β la configuration $\eta\ q_k\ s_l\ \theta$. On dit que α dérive vers β dans la machine m ($\alpha \vdash_m \beta$) dans un des trois cas suivants:

1. Si l'entrée $q_i\ s_j\ s_k\ q_l$ est dans la table de transitions de m et que $\gamma = \eta$ et $\delta = \theta$
2. Si une entrée $q_i\ s_j\ R\ q_k$ tel que $\eta = \gamma \mid s_j$ et $\delta = s_l \mid \theta$ est dans la table de transitions de m.
3. Si une entrée $q_i\ s_j\ L\ q_k$ tel que $\gamma = \eta \mid s_j$ et $\theta = s_l \mid \delta$ est dans la table de transitions de m.

Le calcul

On dit qu'un calcul est une suite finie de pas de calcul. Un calcul:

- Commence par un état initial, généralement noté q_0
- Au bout de plusieurs pas, il se peut que la machine de Turing soit dans un état q_i dans lequel sa tête observe un symbole s_j et que dans la table de transition il n'existe aucune entrée de la forme $q_i\ s_j\ s_x\ q_y$ ... Donc la machine ne peut plus changer d'état, elle va s'arrêter: ceci est un état terminal.

Une machine de Turing peut très bien ne jamais atteindre un état terminal (boucler).

Conventions de codage

Dans ce cours et pendant l'examen, on prendra comme conventions:

- Les symboles sont B (comme “Blanc”) et 1. Le ruban est par défaut rempli de B.
- En entrée:
  - L'entier x est représenté par une suite de x + 1 fois le symbole “1”
  - Le vecteur (x_1, ... , x_n) est représenté par les x_1, ... , x_n séparés par le symbole “B”.

    Donc, l'entreé **(3 , 0 , 2)** est représentée par **B 1 1 1 1 B 1 B 1 1 1 B**
- La machine de Turing retourne un entier, et cet entier est le nombre de fois où le symbole “1” apparait sur le ruban. Les “1” ne sont pas obligés de se suivre, il peuvent très bien être dispersés sur le ruban d'une façon quelconque.

On notera le calcul de la machine m en partant de l'état q_0 sur l'entrée (x_1, ... , x_n) comme:

$$Res_m(q_0(x_1, \dots , x_n))$$

Variantes de la machine de Turing

On verra dans cette partie des variantes de la machine de Turing, et montrera que ces variantes ne nous donnent pas une machine qui calcule plus de choses.

La compléxité en temps est défini comme: $\frac{(Nombre\ de\ pas\ de\ calcul)}{(Longueur\ du\ mot)}$

Machine de Turing à quintuplets

La Machine de Turing à quintuplets contient une table de transitions avec des entrées de la forme:

$q_1 \quad s_1 \quad L / S / R \quad s_2 \quad q_2$

La machine peut donc écrire un symbole est changer de position de la tête en même temps, la commande de déplacement de tête “S” (Stay) désigne le fait que la tête doit rester à la même place.

Il est évident que toute entrée de cette forme peut être remplacée par deux quadruplets. Cette nouvelle machine est donc jusqu'à deux fois plus rapide que l'initial, mais n'offre pas plus de possibilités.

Machine de Turing avec un ruban infini à un seul sens

Pour simuler, avec cette machine, une machine de Turing avec un ruban infini dans les deux sens, on va faire le suivant:

- Les cases avec un indice positif seront codées sur les cases à numéro pair
- Les cases avec un indice négatif seront codées sur les cases à numéro impair.

Ce qui aura donc comme effet:

À la position 0:	L devient R
	R devient R ; R
À la position 1:	L devient R ; R
	R devient L
À une position pair:	R devient R ; R
	L devient L ; L
À une position impair:	R devient L ; L
	L devient R ; R

Cette machine a donc une compléxité temporelle jusqu'à deux fois plus grande.

Machine de Turing avec plusieurs rubans

Pour une machine à n rubans, la table de transition contient des entrées de la forme:

q_1 s_{1_1} ... s_{1_n} L / S / R s_{2_1} ... s_{2_n} q2

n fois

Théorème: la machine de Turing à plusieurs rubans est équivalent à la machine de Turing à un seul ruban, et pour une machine à k rubans une entrée de longueur n est calculée avec une compléxité temporelle jusqu'à $(4*n+2*k)*n$ fois plus petite (donc la machine à plusieurs rubans est plus rapide que la machine à un seul ruban).

Machine de Turing avec Oracle

Une machine de Turing avec Oracle est une machine de Turing “classique”, mais qui peut posséder des entrées de la forme:

q s q_y q_n

dans sa table de transition. Ceci veut dire que si la machine est à l'état q et sa tête lit le symbole s, il fera appel à son oracle. Son oracle va regarder le ruban, compter le nombre de 1, et changer l'état de la machine à q_y si ce nombre appartient à un ensemble A et à q_n sinon.

On apelle T^A-calculable les calculs effectuables par une machine de Turing avec Oracle fonctionnant sur le domaine A.

Fonctions intuitivement calculables

Dans cette partie, notre but est d'identifier ce qu'est une fonction intuitivement (algorithmiquement) calculable.

Système d'équations

Une première idée est d'utiliser les systèmes d'équations récursifs.

On va donc essayer de définir la fonction " factoriel " :

$f(0) = 1$	$f(0) = 1$
$f(n + 1) = (n + 1) * f(n)$	$f(n) = f(n + 1) / (n + 1)$

La définition de gauche converge alors que celle de droite diverge... Or, elles représentent toutes les deux la même chose et sont équivalentes en système!

Un système d'équations définit une relation et pas une fonction.

Récursion primitive: la classe RP

Définition

La classe RP est formée de:

- Une base:
  - La fonction successeur, qui à x fait correspondre x + 1
  - La fonction constante d'arité n, noté $m^{(n)}$

    Par exemple, la fonction $5^{(n)}(x_1 , ... , x_n) = 5$ pour tout $x_1 , ... , x_n$

    PS: seul la constante $0^{(n)}$ est suffisante pour définir le reste.
  - La fonction de selection d'argument, $a_i^{(n)}(x_1, ... , x_n) = x_i$
- Deux fonctions:
  - L'application de fonction, σ: $\sigma[f^{(k)}, g_1^{(n)}, ... ,g_k^{(n)}] = f^{(k)}(g_1^{(n)}, ... , g_k^{(n)})$
  - La récursion primitive, ρ: si on a $h = \rho[g^{(n)} , f^{(n + 2)}]$

$$h(x_1, ... , x_{n+1}) = \begin{cases} g^{(n)} (x_2, ... , x_{n+1}) \text{ si } x_1 = 0 \\ f^{(n + 2)} (x_1 - 1, h(x_1 - 1, x_2, ... , x_{n+1}), x_2, ... , x_{n+1}) \text{ sinon} \end{cases}$$

Par exemple, on peut définir la fonction $Add(x, y) = x + y$ par:

$$Add = \rho[a_1^{(1)}, \sigma[Successeur, a_2^{(3)}]$$

Éléments

Cette définition de la classe RP suffit pour définir de nombreuses fonctions... On apelle ceci **la fermeture de RP** (donc les fonctions définissables en utilisant RP).

Arithmétique

On a auparavant vu comment définir l'addition. Il est simple de voir que la multiplication ainsi que l'exponentiel peuvent être définis d'une façon symetrique.

Comme la fonction ρ nous informe sur l'élément d'avant, on a la possibilité de définir la fonction “prédecesseur” comme suit:

$$\text{Pred} = \rho[\, 0^{(0)}, a_1^{(2)} \,] \quad \text{donc} \quad \text{Pred}(x) = \begin{cases} x - 1 \text{ si } x > 0 \\ 0 \text{ sinon} \end{cases}$$

En utilisant cette fonction, on peut donc aussi créer la fonction de différence propre:

$$\text{Diff} = \rho[\, 0^{(0)}, \sigma[\, \text{Pred}, a_2^{(3)} \,] \,] \quad \text{donc} \quad \text{Diff}(x, y) = \begin{cases} x - y \text{ si } x - y > 0 \\ 0 \text{ sinon} \end{cases}$$

On a par conséquent les fonctions usuelles de l'arithmétique.

Somme et Produit

$\sum$ et $\prod$ sont dans RP. Ceci est une conséquence immédiate du fait que ces deux fonctions ne sont que des simples boucles “pour”.

Prédicats

On définir deux fonctions: Signe et Inverse.

- Signe(x) vaut 1 si x est positif, 0 sinon: $\text{Signe} = \rho[\, 0^{(0)}, 1^{(2)} \,]$
- Inverse(x) est l'inverse de signe: $\text{Inverse} = \rho[\, 1^{(0)}, 0^{(2)} \,]$

Si on prend comme convention que la valeur “vrai” est codé par 1 et la valeur “faux” par 0, il est évident que les prédicats logiques de base qui sont $\wedge$ et $\vee$ sont des combinaisons des fonctions Add et Multiplication avec Signe et Inverse. **Les prédicats logiques sont donc aussi dans RP**.

Si – Alors – Sinon

La conditionelle

SI cond ALORS $code_1$ SINON $code_2$

vaut $code_1$ si cond est “vrai” et $code_2$ sinon; ce qui est la même chose que:

$$\text{If}_{code_1, code_2}(\text{cond}) = (\text{Signe}(\text{cond}) * code_1) + (\text{Inverse}(\text{cond}) * code_2)$$

Minimum borné

L'opérateur de minimum borné s'écrit comme $\mathfrak{M}\,[\,P^{(n+1)}\,]$ ou encore $\mu{\leq}\,[\,P^{(n+1)}\,]$.

$$\mathfrak{M}\,[\,P^{(n+1)}\,](\,z,\,\vec{x}\,) = \begin{cases} \text{Le plus petit } y \leq z \text{ s'il existe tel que } P(\,y,\,\vec{x}\,) \text{ vaut } 1 \\ 0 \text{ s'il n'existe aucun } y \leq z \text{ tel que } P(\,y,\,\vec{x}\,) \text{ vaut } 1 \end{cases}$$

Théorème: si $P^{(n+1)}$ est un prédicat dans RP, alors le minimum borné correspondant, $\mu{\leq}\,[\,P^{(n+1)}\,]$, est aussi dans RP.

Minimum non borné: la classe P

On peut généraliser l'opérateur du minimum borné par un opérateur de minimum non borné:

$$\mu\,[\,P^{(n+1)}\,](\,\vec{x}\,) = \begin{cases} \text{Le plus petit } y \text{ s'il existe tel que } P(\,y,\,\vec{x}\,) \text{ vaut } 1 \\ \text{Indéfini (diverge) sinon} \end{cases}$$

Or, cet operateur peut très bien diverger et donc définit une fonction partielle: **il ne peut appartenir à la classe RP**! Ceci définit donc la classe P, qui contient la classe RP.

En d'autres termes, la classe RP définissait les programmes avec des boucles “pour” alors que la classe P définit les programmes avec des boucles “tant que”.

Plus formellement, la classe P^A est formée de:

- Une base:
  - La fonction successeur
  - La fonction constante d'arité n, noté $m^{(n)}$
  - La fonction de selection d'argument, $a_i^{(n)}$
  - La fonction caractéristique de A, noté 1_A :

$$1_A(\,x\,) = \begin{cases} 1 \text{ si } x \in A \\ 0 \text{ sinon} \end{cases}$$

- Trois fonctions:
  - L'application de fonction, σ
  - La récursion primitive, ρ
  - La minimisation non-bornée, μ:

$$\mu\,[\,P^{(n+1)}\,](\,\vec{x}\,) = \begin{cases} \text{Le plus petit } y \text{ s'il existe tel que } P(\,y,\,\vec{x}\,) \text{ vaut } 1 \\ \text{Indéfini (diverge) sinon} \end{cases}$$

Equivalence des deux modèles

Pour montrer que les deux modèles sont équivalentes, on va faire deux choses:

1. Montrer que tout ce qui est calculable par une fonction Récursive Partielle l'est aussi avec une Machine de Turing
2. Montrer que tout ce qui est calculable par une Machine de Turing l'est aussi avec une fonction Récursive Partielle

La simulation des opérateurs de P par les Machines de Turing

Une Machine de Turing est dite “régulière” si elle satisfait ces deux conditions:

1. Quand la machine s'arrête, elle s'arrête dans une configuration $q_f\ B\ 1^{f(x)}$
2. Elle ne fait aucune modification sur la partie gauche du ruban à partir la position initiale.

Théorème: toute machine de Turing est transformable en une Machine de Turing Régulière.

Calcul des éléments de la base

La fonction “constante” est évidente à faire.

Comme l'entrée est formée de x + 1 fois le symbole “1”, cette machine a juste besoin d'aller une fois à gauche (pour satisfaire le configuration “régulière”).

La fonction de sélection d'argument:

1. Efface tous les “1” des valeurs à gauche de la valeur à conserver
2. Saute la valeur à conserver
3. Efface tous les “1” des valeurs à droite de la valeur à conserver
4. Va au début de la valeur à conserver et efface le premier “1”

Pour le calcul de la fonction caractéristique, on peut tout simplement utiliser l'Oracle. Un théorème indique que toute Machine de Turing avec Oracle peut être transformée en une Machine de Turing “classique”. Donc, on est bon.

On vient de montrer, sans rentrer dans les détails “techniques”, **que la base de P** (et donc aussi de RP) **est simulable par une Machine de Turing**. Restent donc les trois fonctions: composition, récursion primitive et minimum non-bornée.

Fonction de composition, récursion primitive et minimum non-bornée

Pour la fonction de composition:

1. On fait une copie de l'argument sur un deuxième ruban
2. On lance la première fonction sur le premier ruban
3. Quand cette dernière termine on se déplace à la fin de la sortie (toujours du premier ruban donc), ajoute un "1" (par convention d'entrée – sortie), écrit une copie de l'argument (que l'on peut retrouver sur le deuxième ruban) et lance l'autre fonction interne
4. On itère pour chaque fonction interne
5. On lance chaque fonction externe sur le bon résultat

Pour la récursion primitive:

1. On connait le nombre d'itérations, on écrit donc les divers x , x – 1 , ... , 0 en forme de vecteur sur le ruban
2. On va sur le 0 et lance la machine de calcul de f
3. On ajoute un "1" (par convention), recule pour que la machine prenne en compte le x suivant et la sortie du calcul d'avant, puis relance la machine de calcul de f
4. On itère tant que l'on n'a pas atteint le début

Pour le calcul du minimum non-bornée, on peut prendre une Machine de Turing à 3 rubans:

- Un pour coder l'entrée (r_1)
- Un pour coder le y courant (r_2), qui vaut 0 au début
- Un pour exécuter la machine calculant le prédicat (r_3)

Puis, on lance la machine qui va faire la chose suivante:

1. Copier y suivi de l'entrée sur r_3: donc copier les rubans r_2 et r_1 sur r_3
2. Lancer la machine du calcul du prédicat sur r_3
3. Quand ce dernier termine, regarder le résultat:
   - Si le résultat est 1, alors le minimum est sur le ruban r_2, à un "1" près (par convention)
   - Sinon, incrémenter y de 1 (donc ajouter un "1" à r_2) est relancer à la première étape

Comme la fonction de minimum non-bornée, la machine est aussi capable de diverger.

On vient de montrer, sans rentrer dans les détails "techniques", **que les trois fonctions de P** (et donc aussi de RP) **sont aussi simulables par une Machine de Turing**.

$$\mathbf{P \subseteq TM}$$

Propriété: toute fonction calculable par une Machine de Turing l'est aussi par une fonction Récursive Partielle (P)

Comme on a vu avant, la récursion partielle (même primitive) peut modéliser l'arithmétique. L'idée pour modéliser une Machine de Turing sera donc de passer cette dernière dans le monde de l'arithmétique.

Numéros de Gödel

La Gödelisation fait correspondre à une suite d'entiers un nombre unique, défini en tant que:

$$Gödel(\vec{x}) = \prod_{i=1}^{card(\vec{x})} Pr(i)^{(x_i)}$$ où **Pr(i)** est le i^ième^ nombre premier, **Pr(1) = 2**

Par exemple: Gödel(4, 8, 2) = $2^4 * 3^8 * 5^2$ = 2624400

Pour retrouver le i^ième^ composant du vecteur, il suffit donc de retrouver la puissance du i^ième^ nombre premier dans la décomposition de son nombre de Gödel en facteurs premiers.

Il est évident que à toute séquence correspond une et une seule nombre de Gödel et vice-versa.

Arithmétisation de MT

Numéro d'un mot

Considérons une Machine de Turing avec un alphabet de codage à n éléments sur son ruban que l'on notera $\mathbf{a_1, \ldots, a_n}$ (notre Machine de Turing initial avait un alphabet minimal à 2 éléments: “B” et “1”).

On prendra comme convention que chaque lettre de cet alphabet a comme numéro son indice.

Un mot étant une suite de lettres, on peut aussi la noter avec une suite de numéros.

Le nnuméro de Gödel correspondant à un mot sera donc la fonction Gödel appliquée à la suite de numéros des lettres de ce mot.

Numéro d'une instruction

Une Machine de Turing à quintuplets a des éléments de la forme

q_i s_j L / S / R q_k s_l

dans sa table de transition. On va donner comme numéro 1 à “L”, 2 à “S” et 3 à “R”. La séquence d'entiers représentant ce quintuplet sera:

(i, j, numéro(L / S / R), k, l)

Numéro d'une Machine de Turing

Une Machine de Turing est défini par une table de transition d'éléments et on vient de voir comment numéroter chaque élément.

Comme la table de transition est une table unidimensionelle donc une liste, on peut tout simplement l'écrire comme une suite d'éléments.

Le numéro que Gödel d'une Machine de Turing sera la fonction Gödel appliquée à cette suite.

PS: on notera que changer l'ordre de deux éléments du tableau, une opération qui de toute évidence n'a aucun effet sur le fonctionnament de la machine, changera le numéro de Gödel de cette machine! En effet, comme on peut toujours ajouter des éléments "qui ne servent à rien" dans la table de transitions, on peut potentiellement avoir une infinité de numéros de Gödel différents pour des machines qui, en fin de compte, font la même chose. **Par contre, il ne faut pas oublier qu'à chaque numéro de Gödel correspond une et une seule machine.**

Fonctions intermédiaires

On définit cinq fonction récursives primitives:

- **Init(m, x)** donne le numéro de la configuration initiale de la machine numéro m pour l'entrée x.
- **Suivant(m, c)** donne le numéro de la configuration de la machine numéro m après une transition à partir de la configuration numéro c.
- **Conf(m, x, t)** donne le numéro de la configuration de la machine numéro m après t transitions à partir de l'entrée x. On peut prouver que Conf(m, x, t) est récursive primitive:

  $Conf(m, x, 0) = Init(m, x)$

  $Conf(m, x, t+1) = Suivant(m, Conf(m, x, t))$
- **Stop(m, c)** donne **true** si la configuration numéro c est une configuration de l'arrêt de la machine numéro m.
- **Sortie(c)** est le numéro du mot sur le ruban dans la configuration numéro c.

Et, une fonction récursive partielle (qui peut diverger –tout comme une Machine de Turing):

$$\textbf{TempsDeCalcul(m, x)} = \mu_t [\, Stop(m, Conf(m, x, t)) \,]$$

Application et conclusion

En appliquant tout ceci, on peut dire que l'éxécution de la Machine de Turing m sur l'entrée x (x peut très bien être d'arité supérieur à 1) peut s'écrire comme:

$$\textbf{Execution(m, x)} = \textbf{Sortie(Configuration(m, x, TempsDeCalcul(m, x)))}$$

D'où:

$$\mathbf{TM \subseteq P}$$

Complexité

Définitions et rappels

- **Complexité**: la compléxité d'une Machine de Turing est une fonction qui, à une taille d'entrée fait associer un nombre maximal de pas de calcul que la machine va faire avant de s'arreter.
- **Grand O**: $g(n) \in O(f(n))$ s'il existe k_1 et n_1 tel que pour tout $n > n_1$ on ait

  $|g(n)| \leq k_1 * f(n)$, donc si g est majoré par f à partir d'un certain rang.
- **O mega**: $g(n) \in \Omega(f(n))$ s'il existe k_2 et n_2 tel que pour tout $n > n_2$ on ait

  $|g(n)| \geq k_2 * f(n)$, en d'autres termes si g majore f à partir d'un certain rang.
- **Theta**: Finalement, $g(n) \in \theta(f(n))$ si $g(n) \in O(f(n))$ et $g(n) \in \Omega(f(n))$

Notre but sera de trouver des relations de type θ ou O entre les complexité de divers algorithmes. On peut prendre de divers **fonctions de référence**:

$$\log(n) \quad ; \quad n \quad ; \quad n * \log(n) \quad ; \quad n^k \quad ; \quad k^n \quad ; \quad n! \quad ; \quad n^n$$

Quelques modèles

Modèle RAM

Composants

Une machine RAM est “physiquement” composée de:

- Un ruban d'entrée
- Un ruban de sortie
- Des registres
- Des instructions RAM, qui sont numérotés (pour pouvoir faire des sauts d'instruction)

Les rubans sont de longueur infinis et il y a autant de registres que l'on veut. Les registres sont numérotés comme $\mathbf{r_0}$, $\mathbf{r_1}$, $\mathbf{r_2}$, ...

Les registres stockent chacun des entiers naturels (aussi grand que l'on veut). Ils peuvent tous être lus et écrits une infinité de fois. Les rubans se décallent vers la fin à chaque lecture, et on n'a pas le droit de revenir sur les lectures (donc de “rembobiner” les rubans).

Execution

Les instructions RAM peuvent être:

LOAD =i	Met dans le registre $\mathbf{r_0}$ l'entier **i**.
LOAD i	Met dans le registre $\mathbf{r_0}$ l'entier stocké dans le registre $\mathbf{r_i}$.
LOAD *i	Met dans le registre $\mathbf{r_0}$ l'entier stocké dans le registre $\mathbf{r_j}$, **j** étant l'entier stocké dans le registre $\mathbf{r_i}$.
STORE i	Met dans le registre $\mathbf{r_i}$ l'entier stocké dans le registre $\mathbf{r_0}$.
STORE *i	Met dans le registre $\mathbf{r_j}$ l'entier stocké dans le registre $\mathbf{r_0}$, **j** étant l'entier stocké dans le registre $\mathbf{r_i}$.
ADD =i	Ajoute **i** à l'entier stocké dans le registre $\mathbf{r_0}$ et stocke le résultat dans le registre $\mathbf{r_0}$.
ADD i	Ajoute l'entier stocké dans le registre $\mathbf{r_i}$ à l'entier stocké dans le registre $\mathbf{r_0}$ et stocke le résultat dans le registre $\mathbf{r_0}$.
ADD *i	Ajoute l'entier stocké dans le registre $\mathbf{r_j}$ à l'entier stocké dans le registre $\mathbf{r_0}$ et stocke le résultat dans le registre $\mathbf{r_0}$, **j** étant l'entier contenu dans le registre $\mathbf{r_i}$.

Similairement à ADD, on a aussi SUBS (la soustraction). On notera que la machine fonctionne sur les naturels, donc les résultats négatifs sont tronqués à 0.

READ i	Lit l'entier courant sur le ruban d'entrée, le stocke dans le registre $\mathbf{r_i}$ et fait décaller d'une entrée le ruban.
READ *i	Lit l'entier courant sur le ruban d'entrée, le stocke dans le registre $\mathbf{r_j}$ (**j** étant l'entier stocké dans le registre $\mathbf{r_i}$) et fait décaller d'une entrée le ruban.
WRITE =i	Ecrit sur le ruban de sortie l'entier **i** et fait décaller d'une entrée le ruban.
WRITE i	Ecrit sur le ruban de sortie l'entier stocké dans le registre $\mathbf{r_i}$ et fait décaller d'une entrée le ruban.
WRITE *i	Ecrit sur le ruban de sortie l'entier stocké dans le registre $\mathbf{r_j}$ (**j** étant l'entier stocké dans le registre $\mathbf{r_i}$) et fait décaller d'une entrée le ruban.
JUMP n	Continue l'execution à l'instruction RAM numéro **n**.
JGTZ n	Continue l'execution à l'instruction RAM numéro **n** si l'entier stocké dans le registre $\mathbf{r_0}$ est strictement supérieur à **0**.
JGT0 n	Continue l'execution à l'instruction RAM numéro **n** si l'entier stocké dans le registre $\mathbf{r_0}$ est **0**.
HALT	Arrête la machine.

Complexité

La complexité d'une machine RAM est définie de façon standard, et la taille d'un entier est défini comme étant la longueur de sa représentation en binaire.

Equivalence polynomial RAM – Machine de Turing

La simulation d'une Machine de Turing en utilisant une machine RAM est évidente:

L'alphabet de de cette Machine de Turing formé des symboles s_1, ... , s_n. On considère aussi que les cases du ruban (infini d'un seul coté, mais dans tous les cas on avait démontré que toute Machine de Turing peut se ramener en une Machine de Turing avec un ruban infini d'un seul coté avec une complexité polynomiale) de cette Machine de Turing sont numérotées. Pour décrire le comportement en utilisant une machine RAM:

- Le registre $\mathbf{r_{2+i}}$ contient l'entier **j** si sur le ruban de la Machine de Turing le symbole $\mathbf{s_j}$ est à la position **i**.
- Le registre $\mathbf{r_1}$ contient le numéro de case (donc aussi de registre à une addition près) vers lequel la tête de la Machine de Turing pointe.

En utilisant ces informations, la machine RAM simulant une Machine de Turing est évidente à faire. On observe qu'à chaque instruction de la Machine de Turing correspond au maximum 7 instructions RAM, donc la complexité reste de même ordre (on notera que **O(f(n)) = O(7 * f(n))**)

Pour le faire dans le sens inverse, on peut utiliser une Machine de Turing avec 8 rubans:

- Le ruban 1 contient l'entrée de la machine RAM
- Le ruban 2 contient les registres de la machine RAM. Le codage que l'on va utiliser est:

$$\mathbf{[\ B\ i : r_i\ B\ \ldots\ B\ j : r_j\ B\ \ldots\ B\ k : r_k\ B\ ]}$$

En d'autres termes:

- On commence avec le symbole **[**
- Pour chaque registre on met un **B**, le numéro de registre, le symbole **:**, le contenu du registre et un autre **B**
- On termine avec le symbole **]**

On notera que cette liste de registres n'est pas obligatoirement ordonnée, et plusieurs B peuvent séparer les divers contenus. Les numéros sont écrits en binaire.

- Le ruban 3 contient le numéro de l'instruction RAM courant.
- Le ruban 4 contient l'adresse du registre de travail.
- Les rubans 5, 6 et 7 sont utilisées pour les opérations arithmétiques.
- Le ruban 8 est pour la sortie.

On note aussi deux choses:

1. On code les entiers en binaire et l'opération le plus “lourd” qui est autorisé est l'addition. L'addition le plus grand que l'on puisse faire est une multiplication par 2, ce qui correspond à un décallage à droite en binaire... Après l'execution de la $j^{ème}$ instruction, le contenu de chaque registre a donc au maximum **t** + | **I** | + | **B** |, où:
   - **I** est le plus grand entier apparaissant comme opérande dans les instructions de la machine RAM et
   - **B** est le plus grand entier apparaissant dans l'entrée de la machine RAM.
2. Pendant la simulation, la chose la plus dure à faire est la localisation et déplacement de registre (donc le ruban 2). La simulation d'une instruction RAM va donc couter en **O(n^2)**.

   Donc, la simulation entière d'une machine RAM par Machine de Turing va couter en **O(n^3)**.

Donc, les modèles RAM et Machine de Turing sont polynomialement équivalents en complexité.

Langage Pseudo-Pascal

Le Pseudo-Pascal est un langage de programmation “haut niveau” à base d'affectation habituel. Il a donc les instructions habituels:

- **variable <- expression**
- **if condition then statement fi**
- **if condition then statement else statement fi**
- **while condition do statement od**
- **repeat statement until condition od**
- **for variable start end step do statement od**
- **begin statement end**
- **read variable**
- **write variable**

La complexité est toujours définie de façon analogue, et la taille est toujours la taille du codage en binaire.

Théorème: les modèles Pseudo-Pascal et RAM sont polynomialement équivalents.
(PS: on ne va pas s'amuser à le démontrer)

Machine de Turing non-deterministe

Une Machine de Turing non-déterministe est une Machine de Turing qui peut avoir plusieurs actions possibles pour un même symbole lu dans un certain état. La machine peut donc avoir le choix entre plusieurs actions. Même si la machine “fait un choix” dans un certain pas, ce pas prend 1 unité de temps.

La complexité de cette machine est définie comme la plus courte execution possible parmi toutes les executions pour une même entrée. Comme une execution est définie comme étant la plus longue execution possible pour une même taille d'entrée, on pourrait informellement écrire:

min(max(complexités pour entrée de longueur n))

Il est évident de voir qu'une Machine de Turing déterministe est aussi une Machine de Turing non-déterministe (mais qui n'a jamais de choix à faire).

Les classes P et NP

La classe P

La classe P (*Polynomial*) est l'ensemble des problèmes qui peuvent être résolus en un temps polynomial par une Machine de Turing déterministe.

Un problème est donc dans la classe P s'il existe au moins un algorithme déterministe qui le résout et cet algorithme a une complexité en $\mathbf{O(n^k)}$.

Réduction polynomiale de problèmes de décision

Définitions

Un problème de décision est un problème qui va d'un ensemble vers un booléen. On peut donner comme exemple si un mot donné est un palindrome (“se lit de la même façon de gauche à droite que de droite à gauche”), qui va donc de l'ensemble des mots vers un booléen.

Soient B_1 et B_2 deux problèmes de décision. Une réduction de B_1 vers B_2 est une fonction f qui va des instances de B_1 vers les instances de B_2 et a la propriété suivante:

$$\mathbf{B_1(x) \Leftrightarrow B_2(f(x))}$$

Une réduction est dite polynomiale si sa fonction de réduction est de complexité polynomiale. On note alors:

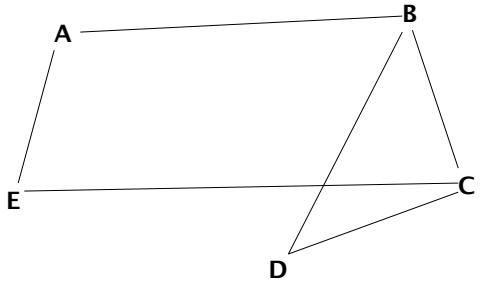
$$\mathbf{B_1 \leq_P B_2}$$

Si $\mathbf{B_1 \leq_P B_2}$ et $\mathbf{B_2 \leq_P B_1}$ alors on dit que ces deux problèmes sont polynomialement équivalents, noté $\mathbf{B_1 \equiv_P B_2}$.

Exemple

Cycle Hamiltonien

Soit un graphe G formé d'un ensemble de points et d'un ensemble de connexions entre ces points. Un cycle est dit “Hamiltonien” si le cycle suit l'ensemble des connexions entre les points et passe par tous les points du graphe une et une seule fois. On notera **CH** le problème le problème du cycle Hamiltonien.



Sur ce graphe, on a deux cycles Hamiltoniens pour deux sous graphes: **A-B-C-E** et **B-C-D**

Voyageur de commerce

Dans le problème de voyageur de commerce, on se donne une liste de villes et une matrice des distances entre chaque ville. Le but est bien sûr de trouver la liste des routes à suivre pour passer par toutes les villes le plus rapidement possible.

Décrite ainsi, le voyageur de commerce n'est pas dans NP: son résultat n'est pas un booléen.

Donc on va faire une variante, **VC-D**: VC-D est le problème du voyageur de commerce, posé ainsi: “pour ces villes, existe-t-il un parcours de longueur n?”

Réduction polynomiale de CH vers VC-D

Pour mapper CH vers VC-D, on doit crée l'entrée pour VC-D à partir de l'entrée de CH:

- Quelle est la liste des villes?
- Quelle est la matrice des distances?
- Quelle est la longueur à demander?

On la crée:

- Comme liste de villes, on prend la liste des points du graphe.
- La matrice: si les deux points sont connectés alors la distance est 1, sinon 2.
- La longueur de parcours à demander: le nombre d'éléments de la liste de points.

On peut facilement vérifier que ce mappage est correct et de complexité polynomial par rapport à la taille de l'entrée (donc le nombre de points sur le graphe).

Propriétés

La réduction polynomiale a trois propriétés importantes qui nous intéressent:

Si $B_1 \leq_P B_2$ alors:

1. Si B_2 est dans P alors B_1 l'est aussi.
2. Si B_1 n'est pas dans P, alors B_2 ne l'est pas non plus.
3. Si $B_1 \leq_P B_2$ et $B_2 \leq_P B_3$ alors $B_1 \leq_P B_3$.

La classe NP

La classe NP (*Non-determinist Polynomial*) est l'ensemble des problèmes qui peuvent être résolus en un temps polynomial par une Machine de Turing non-déterministe.

Elle contient donc des problèmes pour lesquelles trouver une solution est (potentiellement) difficile, par contre vérifier que la solution est une solution est très simple (polynomial).

Relations entre P et NP

Simulation d'une Machine de Turing non-déterministe par une Machine de Turing déterministe

Pour simuler une Machine de Turing non-déterministe en utilisant une Machine de Turing déterministe, on peut tout simplement faire, pas à pas, les divers chemins d'execution (donc executer chaque choix possible à chaque fois qu'il y a un choix à faire). Soit **r** le nombre maximum de quadruplets de la Machine de Turing non-déterministe commençant par le même couple (donc pour lesquels il y a un choix à faire). Si cette machine non-déterministe avait une complexité polynomiale (notons **q(n)**), alors la Machine de Turing déterministe va faire au maximum:

$$r * 1 + r^2 * 2 + \ldots + r^{q(n)} * q(n) = O(r^{q(n)} * q(n))$$

pas pour la simuler. Cette somme est inférieur à $2^{\log(r * q^2(n) * n)}$.

Donc, une Machine de Turing non-déterministe qui a une complexité polynomiale peut être simulée par une Machine de Turing déterministe avec une complexité en $O(2^{p(n)})$, avec p un polynome.

Problèmes NP-complet

Définition

Un problème B est NP-complet si:

1. B est dans NP et
2. Pour tout autre problème B' dans NP, $B' \leq_P B$.

Utilité

Les problèmes NP-complets représentent donc les problèmes “les plus durs” de la classe NP.

Ils sont importants pour nous car pour le moment (08 Mai 2006, 12h22) non seulement personne n'a pu trouver un algorithme déterministe de complexité polynomial pour les résoudre mais aussi personne n'a pu prouver qu'un tel algorithme ne peut exister.

Il faut savoir que beaucoup de problèmes sont NP-complets: le problème du voyageur de commerce que l'on vient de présenter, beaucoup de problèmes sur les graphes ou encore le problème de la satisfabilité booléenne (que l'on va présenter juste après) sont NP-complets.

Certains de ces problèmes ont aussi une signification pratique importante: par exemple, le voyageur de commerce est important non seulement à cause du nombre de voyageurs de commerce mais aussi parce que le déplacement de l'information sur des réseaux (du genre internet) est un problème proche du problème de voyageur de commerce!

Vous pouvez aussi voir: http://en.wikipedia.org/wiki/List_of_NP-complete_problems pour une liste de problèmes NP complets dans de divers domaines.

Cook: “la satisfaisabilité booléenne est NP-complet”

Le problème de la satisfaisabilité booléenne est le suivant: étant donné une formule booléenne, existe-t-il une combinaison d'affectation aux variables booléennes qui rende l'expression vraie?

Pour montrer que tout problème dans NP est polynomialement équivalent au problème de la satisfaisabilité booléenne, on va exprimer les propriétés de la Machine de Turing non-déterministe à quintuplets et un ruban infini dans les deux sens (l'entrée, la table de transitions, la tête de lecture / écriture, le pas de calcul, le calcul, ...) en utilisant des formules booléennes. Si on est capable de faire ceci en un temps polynomial, alors on aura gagné.

La machine en question avait une complexité polynomiale. Soit **p(n)** le polynome qui majore cette complexité. On définit:

- **Q(i, k)** pour **i** entre **0** et **p(n)**: au pas **i** la machine est à l'état $\mathbf{q_k}$.
- **H(i, j)** pour **i** entre **0** et **p(n)** et **j** entre **−(p(n) + 1)** et **p(n) + 1**: au pas **i** la tête de la machine pointe vers la case numéro **j** du ruban.
- **S(i, j, k)**: pour **i** entre **0** et **p(n)** et **j** entre **−(p(n) + 1)** et **p(n) + 1**: au pas **i** la case numéro **j** du ruban contient le symbole $\mathbf{s_k}$.

On va donc créer 6 formules booléennes (**r** dénote le nombre total d'états, et on va supposer que la machine démarre à l'état $\mathbf{q_0}$ et termine à l'état $\mathbf{q_{r-1}}$ si la réponse est “oui”, $\mathbf{q_r}$ sinon):

1. À chaque pas, la machine est à exactement un état:

$$\mathbf{(Q(i, 0) \vee Q(i, 1) \vee \ \dots \vee Q(i, r)) \wedge}$$
$$\mathbf{(Q(i, 0) \Rightarrow \neg Q(i, 1) \wedge \ \dots \wedge \neg Q(i, r)) \wedge}$$
$$\mathbf{(Q(i, 1) \Rightarrow \neg Q(i, 0) \wedge \ \neg Q(i, 2) \wedge \ \dots \wedge \neg Q(i, r)) \wedge}$$
$$\dots$$
$$\mathbf{(Q(i, r) \Rightarrow \neg Q(i, 0) \wedge \ \dots \wedge \neg Q(i, r - 1))}$$

2. À chaque pas, la tête de lecture pointe vers exactement une case:

$$(H(i, -(p(n) + 1)) \vee H(i, -p(n)) \vee \ ... \vee H(i, p(n) + 1)) \wedge$$
$$(H(i, -(p(n) + 1)) \Rightarrow \neg H(i, -p(n)) \wedge \ ... \wedge \neg H(i, p(n) + 1)) \wedge$$
$$(H(i, -p(n)) \Rightarrow \neg H(i, -(p(n) + 1)) \wedge \ ... \wedge \neg H(i, p(n) + 1)) \wedge$$
$$...$$
$$(H(i, p(n) + 1) \Rightarrow \neg H(i, -(p(n) + 1)) \wedge \ ... \wedge \neg H(i, p(n)))$$

3. À chaque pas, chaque case du ruban contient exactement un symbole:

(similaire aux deux d'avant, mais nettement plus long à écrire!)

4. Au premier pas, la machine est à l'état q_0 et le ruban contient l'entrée:

$$Q(0, 0) \wedge H(0, 0) \wedge S(0, 1, k_1) \wedge ... \wedge S(0, n, k_n)$$

5. Au bout de **p(n)** pas maximum, la machine s'arrête (ou s'est déjà arrétée) à l'état “oui” (et pas à l'état “non”, vu qu'elle a accepté le mot):

$$Q(p(n), r - 1) \wedge \neg Q(p(n), r)$$

6. À chaque instant, le pas est obtenu en regardant l'état courant, le symbole du ruban pointé par la tête de lecture / écriture et la table de transition:

    1. Au pas de transition, si la tête ne pointe pas vers une case alors elle ne va pas changer son contenu:

$$(S(i, j, k) \wedge \neg H(i, j)) \Rightarrow S(i+1, j, k)$$

    2. Si elle y pointe, alors la symbole va être remplacé et la tête va bouger en fonction de la table de transition **T**:

$$(S(i, j, k) \wedge H(i, j) \wedge Q(i, e) \wedge T(k, e, m, e', k')) \Rightarrow$$
$$(S(i + 1, j, k') \wedge H(i + 1, j + m) \wedge Q(i + 1, e'))$$

Faire un “et” entre ces formules, qui ont de toute évidence toutes une longueur polynomiale par rapport au nombre d'états et le nombre de pas de calcul, va définir le calcul. Vu comme on a formulé l'étape 5, la formule vaut “vrai” si la machine termine à l'état “oui”, et elle vaut “faux” sinon.

Donc, tout problème de décision se réduit polynomialement au problème de satisfaisabilité booléenne.