Éléments de Calculabilité

I

1 Motivation et Introduction

La théorie de la complexité cherche à qualifier la difficulté intrinsèque des problèmes : pourquoi certains problèmes nous semblent moins faciles à résoudre que d'autres ? Une réponse possible, évidemment, est que cette difficulté est très relative, toute entière contenue dans le *nous* (nous n'avons pas pensé à la bonne méthode de résolution). La réponse que nous allons rechercher, plus satisfaisante pour notre ego, est que indépendemment de notre habileté certains problèmes requièrent une puissance de calcul supérieure aux autres pour leur résolution. La complexité d'un problème sera ainsi mesurée à l'aune du temps minimum nécessaire à un algorithme pour résoudre le problème.

Déterminer la complexité *d'un algorithme* donné, c'est-à-dire son temps d'exécution sur les entrées du problème, n'est pas toujours si simple. Déterminer la complexité *d'un problème* apparaît dès lors rien moins qu'ambitieux, puisqu'il s'agit de déterminer quelle est la plus faible complexité d'un algorithme de résolution, parmi tous les algorithmes que nous pouvons imaginer – et bien sûr tous ceux que nous n'imaginons même pas... La théorie de la complexité est présentée dans la deuxième partie de ce cours ; nous nous attachons dans cette première partie à définir les trois notions intervenant dans la définition de la complexité : qu'est-ce qu'un problème ?, qu'est-ce que résoudre un problème ?, comment modéliser l'exécution d'un algorithme ? Cette dernière question appelle à la définition d'un *modèle de calcul* : un modèle de machine "universelle" pour l'exécution des algorithmes.

Certains auteurs partent directement d'une description d'un modèle d'exécution de haut niveau, tel la RAM. Cependant il nous semble plus facile à établir les liens entre reconnaissance de langages et résolution de problèmes avec une description plus élémentaire. Ainsi, nous commencerons par discuter de la notion de problème et d'algorithme de résolution, puis nous introduirons les automates d'états finis avec quelques variantes et enfin, la machine de Türing. Nous ouvrirons à la fin de ce chapitre une discussion sur la *calculabilité* et les problèmes indécidables. Si la calculabilité, qui s'occupe de déterminer si une fonction est calculable sur tel ou tel type de machine n'est pas le centre de ce cours, il est nécessaire de se demander si le modèle de calcul retenu est suffisamment puissant pour modéliser l'exécution de tout algorithme. Une seconde interrogation, qui pourrait être le préambule à cette introduction, est de savoir si, pour tout problème, il existe un algorithme pouvant le résoudre...

2 Résolution de Problèmes et Reconnaissance de Langages

2.1 Quelques problèmes

A travers quelques problèmes, nous allons introduire simplement la formalisation de ce qu'est un problème et un algorithme qui le résout.

1. Savoir si un entier n est un nombre premier ou non ? Ce problème est un *problème de décision* : la réponse au problème est *vrai* si le nombre n'admet pas d'autre diviseur que 1 et lui-même, *faux* sinon. Il paraît simple de répondre à la question en essayant tous les diviseurs entre 2 et $\sqrt{n}$.
2. L'existence d'un chemin entre deux sommets dans un graphe. Ce problème est également un *problème de décision* : la réponse au problème est *vrai* si il existe un chemin, *faux* sinon. Il n'est pas "difficile" de trouver la réponse en visitant de proche en proche les sommets. La complexité de cet algorithme de recherche est linéaire en le nombre d'arêtes.
3. Considérons maintenant un problème plus compliqué : déterminer un flot maximum dans un réseau. Ce problème est un *problème d'optimisation* dont une résolution possible (Ford & Fulkerson) fait appel au problème précédent en saturant une série de chemins entre la source et le puits dans le réseau résiduel.
4. La recherche d'un sous-ensemble de sommets connectés deux à deux (une *clique*) de cardinalité maximale dans un graphe. Ce problème est difficile et on ne connaît pas de solution "satisfaisante" à ce jour. Par satisfaisant, nous entendons une solution autre que l'énumération exhaustive de tous les $\mathcal{O}(2^n)$ sous-graphes possibles.
5. L'existence d'un pavage du plan par un ensemble de formes géométriques. La question est de décider si il est possible en n'utilisant que les formes qui nous sont données et sans avoir de recouvrement entre elles, de paver entièrement le plan. Ce problème a été démontré *indécidable*.

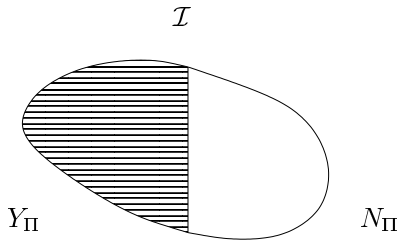
Énoncer un problème (calculatoire) demande simplement la description de ses entrées, et du résultat attendu. Il s'agit donc, ni plus, ni moins, de définir une fonction f sur un ensemble de départ, les instances du problèmes, associant à chaque instance le résultat attendu. Nous nous intéressons à des problèmes combinatoires, définissant un ensemble infini d'instances. Résoudre le problème, c'est être capable pour chaque instance x de fournir le résultat $f(x)$. Donner la fonction f en extension, par la liste des couples $(x, f(x))$ n'est évidemment pas une réponse satisfaisante au problème! Résoudre le problème consiste de fait à *calculer* la fonction f, à donner une *procédure effective* capable de construire sur chaque instance x son résultat $f(x)$. Résoudre un problème, c'est donner un algorithme de résolution.

2.2 Problèmes de décision

Un problème de décision concerne l'énoncé d'une question générique où la réponse pour chaque instance est soit *oui*, soit *non*, le *vrai* ou le *faux*. Une instance d'un problème de décision pour laquelle la réponse est *oui* est appelé une instance *positive*, sinon elle est dite *négative*. Cette notion de problème est syntaxique, indépendante de tout modèle de calcul : la lecture de l'énoncé du problème suffit à déterminer si nous avons affaire à un problème de décison. En terme fonctionnel, un problème de décision correspond à la description d'une fonction à valeur booléenne. Le problème REACHABILITY, le deuxième exemple de la section précédente, décidant de l'existence d'un chemin entre 2 sommets d'un graphe s'énonce formellement :

REACHABILITY
INSTANCE : Un graphe G=(V,E) et deux sommets x et $y \in V$.
QUESTION : Existe-t-il un chemin de x à y ?

Un problème de décision Π correspond ainsi à la description d'un ensemble d'instances $\mathcal{I}$, partitionné en l'ensemble Y_Π des instances positives (*yes* instances) et N_Π des instances négatives (*no* instances). Résoudre Π correpond à décider pour chaque instance $x \in \mathcal{I}$ si $x \in Y_\Pi$.



$x \in \mathcal{I},\ x \in ?\ Y_\Pi$

Les problèmes de décision sont le centre de la théorie de la Complexité. Aussi allons-nous pour l'instant limiter notre discussion à cette classe de problèmes.

2.3 Codage des instances

Pour écrire et exécuter en machine un algorithme de résolution d'un problème, il nous faut choisir une représentation des instances, un *codage* des entrées du problème rendant ces données accessibles à notre procédure effective.

Ainsi pour le problème REACHABILITY, nous pouvons choisir de représenter un graphe par exemple par sa matrice d'adjacence, ou encore par des listes d'adjacence. Le choix de la représentation des instances n'est évidemment pas

anodin ; l'art de l'algorithmique réside précisément dans le choix des bonnes structures de données. Les impacts sur la complexité de l'algorithme portent :

- sur le temps d'exécution. Pour REACHABILITY, un algorithme de recherche s'exécutera en temps $\mathcal{O}(m)$ pour une représentation par liste d'adjacence, mais en temps $\mathcal{O}(n^2)$ pour une représentation par matrice d'adjacence.
- sur la taille du codage. Si nous comptons le nombre de bits nécessaires à la représentation mémoire d'un graphe, la représentation par matrice d'adjacence demande $\mathcal{O}(n^2)$ bits, tandis que la représentation par liste en demande $\mathcal{O}(m \log n)$.

Or nous évaluerons la complexité d'un algorithme comme son temps d'exécution (nombre de pas de calcul de la machine) en fonction de la taille de l'instance (espace mémoire nécessaire au codage sur la machine). Il nous faudrait donc en toute rigueur définir la complexité d'un problème comme la plus petite complexité d'un algorithme sur le meilleur codage de ses instances. Nous allons faire abstraction du codage en admettant que tous les codages raisonnables d'un problème demandent la même taille mémoire et conduisent aux même temps d'exécution pour un algorithme, à un polynôme près. Nous considérerons donc implicitement pour chaque problème un *codage naturel* de ces instances, polynomialement liés à tous les codages raisonnables du problème. Pour prix de cette abstraction, nous ne pourrons préciser la complexité d'un problème qu'à un polynôme près.

2.4 Reconnaissance de Langage

Un codage des instances d'un problème conduit à une représentation des données en mémoire. Si nous pensons à un modèle d'exécution sur un ordinateur, au niveau élémentaire le codage d'une instance est une suite de bits à 0 ou à 1. Cette suite de bits est un *mot* sur l'alphabet $\Sigma = \{0, 1\}$. L'ensemble des mots codants les instances du problème défini un *langage*.

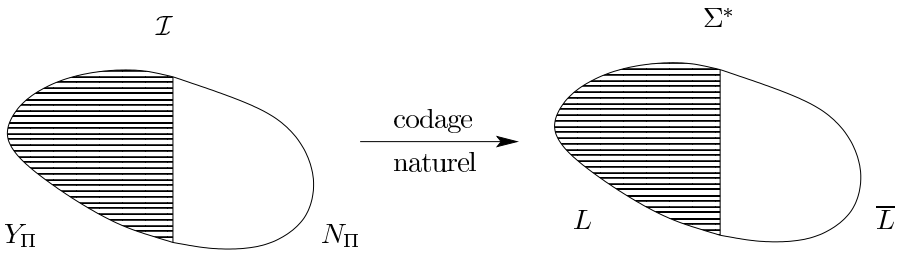
Bien que les langages soient des structures très pauvres, la théorie des langages a de multiples applications, en analyse syntaxique, en compilation,... Un alphabet Σ est un ensemble fini de symboles, par exemple $\Sigma = \{0, 1\}$. Un *mot* w est une suite finie de symboles de Σ, par exemple $w = 0$, $w = 10001$, $w = 000000011$. L'ensemble des mots sur Σ est noté Σ^*. Un langage L est simplement une partie de Σ^*, c'est-à-dire un ensemble de mots.

On peut décrire un langage de différentes façons. En extension, comme l'ensemble de ses mots, s'il est fini (les langages finis n'ayant que fort peu d'intérêt pour ce qui nous concerne). On peut également décrire un langage par une grammaire, ou encore par des règles de composition à partir d'autres langages. Parmi les compositions possibles :

- les opérations ensemblistes classiques : l'union, l'intersection, la différence symétrique...En particulier pour un langage L, nous noterons $\overline{L}$ son complémentaire dans Σ^*, c'est-à-dire l'ensemble des mots n'appartenant pas à L.
- la concaténation $L_1.L_2 = \{w \mid w = w_1 w_2 \text{ avec } w_1 \in L_1,\ w_2 \in L_2\}$,

– la fermeture itérative $L^* = \{w \mid w = w_1 \dots w_n, w_i \in L\}$. La fermeture itérative de L (ou fermeture de Kleene) est l'ensemble des mots formés par une concaténation finie de mots de L.

Le problème de la reconnaissance d'un langage L consiste à décider pour tout mot $w \in \Sigma^*$ si w appartient à L. La reconnaissance de langages est l'abstraction par rapport aux codages de la résolution d'un problème de décision. Pour un problème de décision Π, un codage représente chacune des instances comme un mot sur un alphabet Σ de symboles. Nous pouvons ainsi associer à Π le langage L_Π constitué de tous les mots codants, à travers un codage naturel de Π, les instances positives. Décider pour une instance x si $x \in Y_\Pi$ est alors équivalent à décider pour le mot w encodant x si $w \in L_\Pi$.



Problème de décision Π
$x \in \mathcal{I},\ x \in ?\ Y_\Pi$

Reconnaissance de L
$w \in \Sigma^*,\ w \in ?\ L$

En particulier la taille d'une instance sera définie par la taille $|w|$ (nombre de symboles) du mot codant l'instance dans un codage naturel.

3 Modèles de calcul

Nous avons modélisé la résolution d'un problème de décision comme la reconnaissance d'un langage, à un codage naturel près. Il nous reste pour définir formellement la complexité à modéliser l'exécution d'un algorithme de reconnaissance de langage sur une machine. Ceci revient à nous donner un *modèle de calcul* pour juger du temps d'exécution et de l'espace mémoire requis par un algorithme.

Les *automates* vont correspondre à ce modèle de calcul. On représente les automates à partir d'un ruban et d'un mécanisme (tête de lecture). Les états de l'automates et ses transitions représenteront les différentes étapes d'un algorithme. Le ruban correspondra à la mémoire de notre machine où stocker le mot d'entrée.

3.1 Automate d'états finis

Nous commençons par présenter une classe simple d'automates qui nous aidera à préciser ce que peut être un support d'exécution.

Le ruban de notre automate est constitué de cases élémentaires sur lesquelles sont inscrits des symboles appartenant à un alphabet Σ. Une tête de lecture est capable de lire le contenu de la case sur laquelle elle est placée. Notons que le ruban est en lecture seule, c'est-à-dire que la tête ne peut écrire sur le ruban. Au début de l'exécution, on est dans l'état initial de l'automate et un mot (l'entrée) est inscrit sur le ruban. La tête de lecture pointe sur le premier caractère de ce mot. A chaque étape (transition), on lit le symbole de la case repérée par la tête de lecture et on la déplace d'une case vers la droite en calculant un nouvel état pour l'automate. Formellement :

Définition 1 *Un automate fini (déterministe) est un quintuplet* $M = (Q, \Sigma, \delta, q_0, F)$
- Q *est un ensemble fini d'états,*
- Σ *est un alphabet,*
- $\delta : Q \times \Sigma \to Q$ *la fonction de transition,*
- $q_0 \in Q$ *l'état initial,*
- $F \subseteq Q$ *l'ensemble des états accepteurs.*

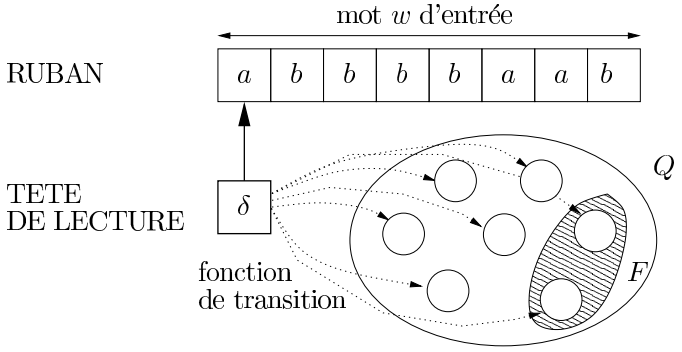


FIG. 1 – Composants d'un automate fini : ruban, états, tête de lecture et fonction de transition.

Le couple $(q, w) \in Q \times \Sigma^*$ est une *configuration* de l'automate. La configuration (q', w') est dérivable en une étape de la configuration (q, w) par l'automate fini déterministe M (noté $(q, w) \vdash (q', w')$) si, par définition, $\exists \sigma \in \Sigma$, tel que $w = \sigma w'$ et $q' = \delta(q, \sigma)$. Plus généralement, la configuration (q', w') est dérivable de la configuration (q, w) par l'automate fini déterministe M (noté $(q, w) \vdash^* (q', w')$) si, par définition, il existe $k \geq 0$ et des configurations $(q_i, w_i), 0 \leq i \leq k$ telles que :

- $(q, w) = (q_0, w_0)$
- $(q', w') = (q_k, w_k)$
- $\forall i, 0 \leq i \leq k-1$, $(q_i, w_i) \vdash (q_{i+1}, w_{i+1})$

L'exécution d'un automate M sur un mot w est la suite des configurations $(s, w) \vdash (q_1, w_1) \vdash \ldots \vdash (q_n, \varepsilon)$, où ε dénote le mot vide, c'est-à-dire que nous sommes arrivés dans la configuration où tous les symboles de w ont été lus. Un mot w est accepté par un automate fini déterministe M si $(s, w) \vdash^* (q, \varepsilon)$ avec $q \in F$. Le langage accepté par un automate fini déterministe M est l'ensemble $L(M)$ des mots acceptés par M.

Nous détaillons ci-dessous un exemple d'automate fini déterministe. On les représente souvent par un graphe orienté où les sommets sont les états et les arcs sont les transitions comme dans la figure 2. On représente par un double cercle les sommets dont l'état est accepteur et une flèche sans étiquette représente l'état initial :

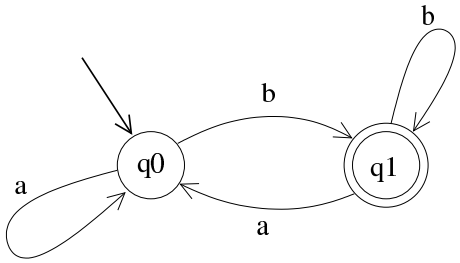


FIG. 2 – Automate déterministe reconnaissant $\{a, b\}^*.\{b\}$

L'automate fini déterministe M avec $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $F = \{q_1\}$, et δ défini par :

q	q_0	q_0	q_1	q_1
σ	a	b	a	b
$\delta(q, \sigma)$	q_0	q_1	q_0	q_1

Le langage accepté par M est $L(M) = \{w | w$ se termine par un $b\}$. On peut vérifier facilement que l'exécution à partir du mot d'entrée *aaab* passe bien par trois états q_0, puis s'arrête dans l'état q_1 qui est accepteur. Il appartient bien au langage $L(M)$. On vérifiera facilement que cet exemple fournit un automate correspondant à la résolution du problème de décider si un entier est pair pour un codage binaire.

Puissance des automates d'états finis

Nous pouvons nous demander quelle est la puissance d'expression des automates d'états finis : peuvent-ils reconnaître tous les langages ?

On peut montrer que les langages reconnus par les automates finis sont exactement les langages *réguliers* : tout langage reconnu par un automate fini est régulier, et tout langage régulier peut être reconnu par un automate fini. Les langages réguliers peuvent se définir de la manière suivante (entre autres !) :

Définition 2 *Un langage L sur un alphabet Σ est régulier ssi il vérifie l'une des conditions suivantes :*

- *L est un singleton (ne contient qu'un seul mot),*
- *Il existe deux langages réguliers L_1 et L_2 tels que $L = L_1 \cup L_2$, ou $L = L_1.L_2$ ou encore $L = L_1^*$.*

Par exemple, le langage L sur l'alphabet $\{a, b\}$ constitué des mots qui se terminent par un b est régulier, on peut l'écrire $L = (\{a\} \cup \{b\})^*.\{b\}$.

Pour montrer qu'il n'y a pas que des langages réguliers, il suffit de remarquer qu'il n'y a pas assez de façons de les construire à partir des singletons pour représenter tous les langages. La preuve est simple par des arguments combinatoires de mise en correspondance - bijection - d'ensembles infinis. L'ensemble des langages réguliers est dénombrable, alors qu'il y a un nombre non dénombrable de langages.

Un exemple simple de langage non régulier est celui constitué par des palindromes à partir d'un alphabet initial de deux lettres. De même le langage composé des mots de la forme $a^n b^n$ sur l'alphabet à deux lettres a et b ne peut être reconnu par un automate fini.

La démonstration utilise la technique dite du *lemme de l'étoile* qui vaut la peine qu'on la détaille : supposons par l'absurde qu'un automate fini A reconnaît le langage $L = \{a^n b^n\}$. Comme L est infini, il contient un mot w de taille supérieure au nombre d'états m de A. Lors de la reconnaissance de w, A passe donc forcément deux fois par un même état. Le mot w peut donc se décomposer sous la forme $w = xuy$, où le sous-mot u correspond aux symboles lus entre les deux occurrences du même état. Le mot $w' = xu^2y$ est donc nécessairement reconnu par A, or on peut montrer par une simple étude de cas sur u que w' n'est pas dans L (c'est à dire de la forme $a^n b^n$).

3.2 Automate d'états finis non déterministe

Pour augmenter la puissance des automates finis et reconnaître ainsi plus de langages, nous pouvons essayer de considérer leur version non déterministe. Les automates finis non déterministes sont des automates finis où l'on permet plusieurs transitions correspondant à la même lettre au départ de chaque état. On peut s'autoriser des transitions sur le symbole vide ε dans les automates finis non déterministes (sans avancer la tête de lecture). De plus, on peut avoir des transitions sur plusieurs symboles, c'est-à-dire directement sur des mots de Σ^*.

Définition 3 *Un automate d'états finis non déterministe M est un quintuplet $M = (Q, \Sigma, \Delta, q_0, F)$ tel que :*

- Q *est un ensemble fini d'états,*
- Σ *est un alphabet,*
- $\Delta \subseteq (Q \times \Sigma^* \times Q)$ *la relation de transition (Δ n'est pas une fonction : elle peut associer plusieurs états à la lecture d'un même symbole.),*
- $q_0 \in Q$ *l'état initial,*
- $F \subseteq Q$ *l'ensemble des états accepteurs.*

Nous donnons ci-dessous un exemple d'automate fini non déterministe, représenté figure 3. Cet automate M se définit par $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$, et la relation Δ :

q	q_0	q_0	q_0	q_1	q_1
σ	a	a	b	a	b
$\Delta(q, \sigma)$	q_0	q_1	q_0	q_2	q_2

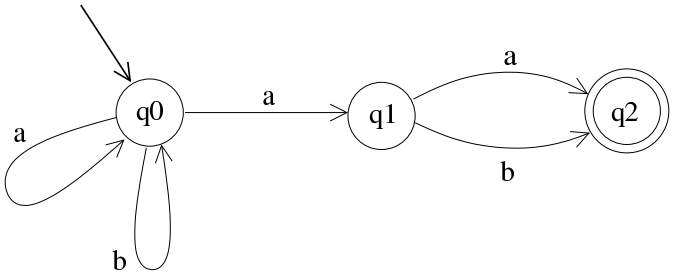


FIG. 3 – Un automate fini non déterministe

Plusieurs exécutions sont en général possibles sur un même mot. Pour que le mot soit accepté, *il suffit que l'une d'entre elles conduise à un état accepteur* (en ayant lu le mot d'entrée en entier). Le lecteur vérifiera facilement que le langage associé à l'exemple est : $L(M) = \{w | w$ possède un a en avant-dernière position $\}$.

Le non déterminisme semble rendre nos automates finis capables de reconnaître plus de langages, du fait qu'une seule exécution conduisant à un état accepteur suffit pour accepter un mot. Il n'en est rien : pour tout automate fini non déterministe, on peut construire un automate fini déterministe reconnaissant le même langage. La démonstration n'est pas difficile, elle consiste à expliciter systématiquement les transformations de l'un à l'autre (création d'un nouvel état pour les transitions multiples, sérialisation des transitions sur plusieurs caractères, etc.). Les automates finis, déterministes ou non déterministes, reconnaissent donc exactement les langages réguliers.

Il existe plusieurs plusieurs façons pour enrichir la puissance des automates d'états finis. On peut par exemple leur adjoindre une pile infinie. Ceci est suffisant pour permettre par exemple de reconnaître le langage $a^n b^n$ (on empile les

a et on dépile tant qu'il reste un b sur le ruban). Ceci fait l'objet de l'exercice **??** à la fin du chapitre. Ce cours n'étant pas centré sur la théorie des langages, passons donc directement aux machines de Türing.

3.3 Machine de Türing

Pour étendre la puissance des automates finis, nous allons ajouter la possibilité, en plus de lire sur le ruban, de pouvoir *écrire* sur le ruban et de controler le déplacement de la tête de lecture/écriture. Comme pour les automates d'états finis, le mot d'entrée est initialement inscrit sur le ruban, mais nous avons maintenant potentiellement accès à une mémoire infinie en lecture/écriture.

Nous allons présenter dans cette section la machine de Türing de base. C'est un modèle de calcul qui date de 1936, avant même l'existence de l'Informatique et des ordinateurs. La machine de Türing (TM en abrégé) est un automate constitué d'une mémoire qui est un ruban infini dans les deux sens divisé en cases élémentaires, contenant chacune un symbole d'un alphabet Γ ; d'une tête de lecture-écriture qui pointe sur une case du ruban et en lit le contenu ou y écrit un symbole, et qui peut se déplacer sur le ruban suivant 2 mouvements élémentaires (d'une case vers la gauche ou vers la droite) ; d'une unité de contrôle finie définie par un ensemble d'états et une fonction de transition. Cette fonction de transition est plus complexe que pour un automates d'états finis. Elle donne pour chaque état de la machine et chaque symbole lu par la tête de lecture, l'état suivant, un nouveau symbole à écrire et le mouvement de la tête de lecture.

Définition 4 *Une TM est un septuplet* $(Q, \Gamma, \Sigma, \delta, \square, q_0, F)$ *:*
- Q *est l'ensemble des états*
- Γ *est l'ensemble des symboles que l'on peut utiliser sur le ruban*
- Σ *est l'alphabet d'entrée (*$\Sigma \subset \Gamma$*)*
- δ *est la fonction de transition de* $Q \times \Gamma$ *dans* $Q \times \Gamma \times \{L, R, S\}$ *où* L *et* R *sont les déplacements respectivement vers la gauche et vers la droite,* S *est la position stationnaire*
- $\square$ *est le symbole blanc (*$\square \in \Gamma - \Sigma$*)*
- q_0 *est l'état initial et* F *est l'ensemble des états accepteurs.*

Initialement, la tête de lecture pointe sur le premier caractère du mot d'entrée. Toutes les autres cases contiennent le caractère $\square$ (blanc). La machine se trouve dans l'*état initial* q_0. L'exécution d'une procédure effective sur une TM consiste successivement à lire le symbole sous la tête de lecture-écriture, à remplacer ce caractère par le symbole indiqué par la fonction de transition, puis à déplacer la tête et à changer d'état conformément à la fonction de transition.

A titre d'exemple nous avons représenté figure **??** une TM pour la reconnaissance du langage $L = \{a^n b^n | n \geq 1\}$ sur l'alphabet $\Sigma = \{a, b\}$. La fonction de transition δ est représentée par les arcs de l'automate, en annotant chaque arc du caractère lu provoquant la transition, et du couple (σ, D) précisant le

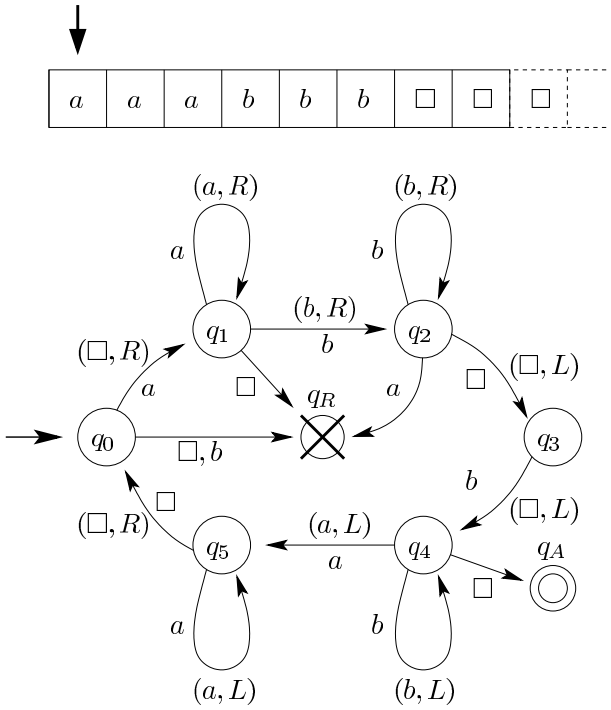


FIG. 4 – Une machine de Türing reconnaissant le langage $a^n b^n$

caractère à écrire et le déplacement à effectuer. Cette machine remplace successivement le premier a et le dernier b du mot du ruban par un blanc $\square$. Un mot w conduit à l'état accepteur q_A si l'exécution conduit à un ruban ne contenant que des blancs $\square$. Nous avons ajouté un état rejeteur q_R atteint pour les mots $w \notin L$.

La machine du Türing de l'exemple précédent n'est pas la seule pouvant reconnaître le langage L. En particulier, nous aurions pu plus naturellement *compter* le nombre de a et de b du mot. La différence fondamentale des machines de Türing par rapport aux automates d'états finis est, grâce à l'accès en écriture du ruban, de pouvoir manipuler l'*arithmétique*. Ainsi une machine de Türing peut non seulement reconnaître un langage, mais peut également *calculer* un résultat, en écrivant sur sa bande.

Par exemple, la TM suivante (figure **??**) calcule l'incrément d'un entier codé en binaire. Cette machine est donnée par $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\square\}$, $Q = \{q_0, q_1, q_2, q_f\}$, et la fonction de transition δ :

état	symbole lu	état	symbole écrit	déplacement
q_0	0	q_0	0	R
q_0	1	q_0	1	R
q_0	$\square$	q_1	$\square$	L
q_1	0	q_2	1	L
q_1	1	q_1	0	L
q_1	$\square$	q_2	1	L
q_2	0	q_2	0	L
q_2	1	q_2	1	L
q_2	$\square$	q_f	$\square$	R

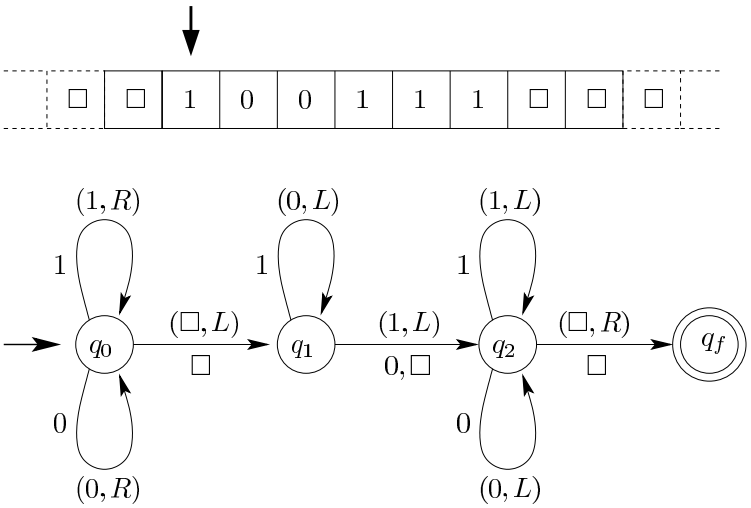


FIG. 5 – Une machine de Türing calculant l'incrément d'un binaire

L'entier est codé en binaire sur la bande par un mot sur l'alphabet $\{0, 1\}$, avec le bit de poids faible à droite et celui de poids fort à gauche. A l'initialisation, la tête de lecture se situe au début du mot, sur le bit de poids fort du nombre (nous supposons pour simplifier que le mot est précédé sur la bande d'un blanc). Dans l'état q_0, la machine parcourt alors le mot pour atteindre le bit de poids faible, en cherchant l'ocurrence du premier blanc $\square$. L'état q_1 sert à remonter le mot en propageant la retenue ; l'état q_2 permet de repositionner la tête de lecture/écriture au début du mot.

Le temps de calcul et l'espace mémoire requis pour l'exécution d'un algorithme sur une entrée w sont définis comme le nombre de transitions nécessaires à la machine de Türing pour aboutir à un état accepteur et comme le nombre de cases utilisées sur le ruban. Cette description est donc très bas niveau, puisqu'elle

revient à compter les instructions machines exécutées et les bits mémoires. Rappelons cependant que nous n'évaluerons le temps et la mémoire qu'à un polynôme près, ce qui nous évitera des analyses fastidieuses.

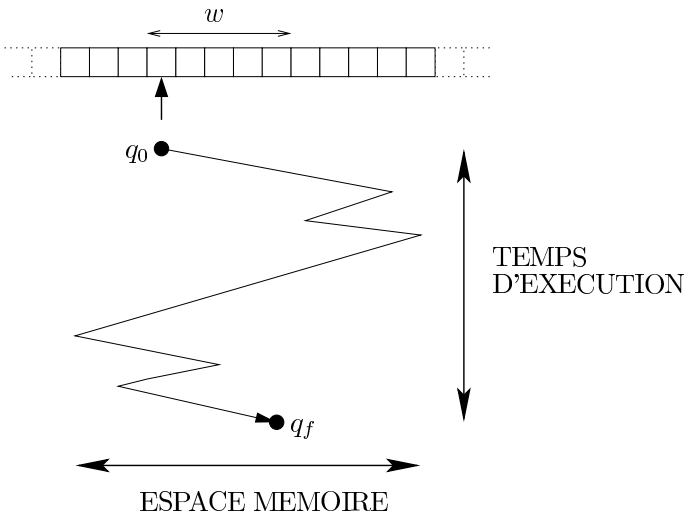


FIG. 6 – Le temps d'exécution et l'espace mémoire sur une machine de Türing

Puissance des machines de Türing

La machine de Türing est le modèle de calcul que nous adopterons pour modéliser l'exécution d'un algorithme. Nous pouvons nous demander si ce modèle est suffisamment puissant, c'est à dire si nous avons simplement le "droit" de faire cela !

La réponse n'a rien d'évident en soi. Les machines de Türing, si elles reconnaissent un ensemble de langages beaucoup plus vastes que les seuls langages réguliers reconnus par les automates d'états finis, ne reconnaissent pas tous les langages. Mais tout langage peut-il être décidé par un algorithme ? La réponse est non. La thèse de Church-Türing stipule que tout modèle "raisonnable" de calcul est équivalent au modèle de la machine de Türing. C'est une thèse au sens premier du terme, qui peut également être énoncée comme : les langages pouvant être reconnus par une procédure effective sont exactement ceux pouvant être décidés par une machine de Türing. Nous admettrons cette thèse pour la théorie de la complexité , c'est à dire l'équivalence entre machine de Türing et algorithmes (procédures effectives).

Pourquoi la machine de Türing s'appelle-t-elle ...une machine ? Si une certaine "machinerie" intervient (ruban, tête de lecture,...) nous considérons une

machine de Türing comme le modèle de l'exécution d'un algorithme, en quelque sorte son cablage en dur à travers les états et les transitions de l'automate. Et nous continuerons à faire ainsi dans la suite de ce document. Cependant nous pourrions aussi avoir une vision différente, similaire au modèle de Von Neumann pour les ordinateurs, dans laquelle l'automate, correspondant au modèle d'exécution, est une unité de contrôle "universelle" qui cherche sur le ruban à la fois les données et les instructions à exécuter ; un algorithme est alors directement codé par un mot sur le ruban, représentant les états et les transitions de sa machine de Türing.

Variantes de la machine de Türing

On distingue plusieurs variantes de la machine de Türing élémentaire : rubans multiples, rubans infinis d'un côté seulement, transitions non déterministes, etc..

On peut montrer tout d'abord que les machines à une ou plusieurs bandes sont équivalentes. Notre modèle de machine multi-bandes est constitué de k têtes de lecture, à transition *synchrone*. On suppose que chaque bande a le même ensemble de symboles, ceci n'est pas obligatoire mais simplifie la présentation.

Plus précisément, on cherche à établir que si un langage L est accepté par une machine de Türing à k bandes alors, il existe une machine à une bande qui reconnait L.

La preuve est constructive, elle est basée sur la notion de *piste*. Considérons une TM à k bandes et dérivons une machine à une bande TM' avec $2k$ pistes qui simule TM, chaque bande correspondant à 2 pistes : la piste $2i-1$ a des blancs partout sauf dans la position repérée par la tête de lecture. La piste $2i$ contient les informations de la bande i. En d'autres termes, les k bandes sont remplacées par une seule bande contenant $2k$ cases, dont une moitié correspond aux informations-symboles et l'autre moitié sert à retrouver les transitions. TM' a un alphabet plus grand pour pouvoir coder chaque 2-uplet en un seul symbole. Par exemple, la TM simulant une TM à deux bandes sur l'alphabet $\{0,1\}$ aura un alphabet sur $\{00,01,10,11\}$. TM' fonctionne en visitant à chaque étape les k cases repérées dans TM. Cette opération est délicate et suppose que les états contiennent aussi l'indication sur l'endroit où se trouve les têtes sur chaque bande (gauche ou droite par exemple). On peut imaginer des procédures plus sophistiquées et coûteuses, comme un déplacement à droite, 2 à gauche, 4 à droite, et ainsi de suite. On revisite ensuite les cases marquées et on effectue la transition correspondante.

On peut montrer de même l'équivalence d'une TM élémentaire à bandes infinies avec une TM dont le ruban est limité d'un coté (par exemple à gauche). L'exercice **??** détaille la construction d'une TM infini d'un seul côté à partir d'une TM à ruban infini des 2 cotés.

Une variante fondamentale : le non déterminisme

La dernière variante à considérer est la TM non déterministe. Cette caractéristique est importante, elle est à l'origine des classes de complexité.

Le déterminisme est défini par l'unicité du choix que l'on peut obtenir à chaque transition. On peut au contraire envisager plusieurs transitions possibles à chaque étape : on a alors potentiellement un grand nombre d'exécutions possibles (certaines pouvant être infinies) à partir d'un seul mot d'entrée, et on dira que le mot est accepté s'il est accepté dans *au moins* une de ces exécutions. On peut alors montrer, comme pour les automates, que cet ajout ne change pas la puissance d'expression des machines de Türing : en introduisant un nombre suffisant de nouveaux états, on peut transformer toute machine non-déterministe en une machine déterministe qui reconnaît le même langage (mais en un temps potentiellement beaucoup plus long).

Il est évident qu'un langage accepté par une TM déterministe peut l'être par une TM non déterministe. Nous allons démontrer la réciproque :

La preuve est une fois encore constructive. Nous montrons qu'un langage accepté par une TM non déterministe à une bande peut l'être par une TM déterministe à trois bandes. Intuitivement, on va explorer toutes les exécutions possibles non pas les unes après les autres (car on risque de tomber sur une exécution infinie), mais en fixant à l'avance un nombre n de transitions maximum et en considérant toutes les exécutions de longueur n, et si aucune ne mène à un état accepteur, en envisageant $n + 1$ transitions. Cela revient à parcourir l'arbre de toutes les exécutions possibles en largeur et non en profondeur.

Notons r le nombre maximum de choix possibles pour chaque transition. Sur le premier ruban, on va coder l'historique des choix que l'on a envisagés, en les générant par exemple dans l'ordre lexicographique (on utilise pour cela un alphabet de r symboles). Sur le second on conserve le mot initial pour pouvoir recommencer une nouvelle exécution avec un autre choix. Et le troisième ruban sert à effectuer le calcul proprement dit. Dès que la machine du troisième ruban détecte un état accepteur, le mot d'entrée est reconnu.

En conclusion, augmenter les fonctionnalités de la machine de Türing ne sert à rien : la machine simple acceptera les mêmes langages que dans les variantes les plus sophistiquées, ce qui est conforme à la thèse de Church-Türing. Ici, on ne se pose pas (encore) la question de l'efficacité et de la performance d'une machine...

4 Indécidabilité

Les machines que nous avons construites permettent de répondre "oui" pour tous les mots d'un langage et "non" pour tous les mots n'appartenant pas à ce langage. On dit que la machine accepte ou n'accepte pas ces mots. Il se peut cependant que la machine ne s'arrête pas pour un mot n'appartenant pas au

langage, qu'elle ne puisse pas décider si un mot appartient à ce langage, mais comment discerner si une exécution est infinie, ou simplement très longue ?

Les modèles de calcul ont intéressé les logiciens et mathématiciens depuis le début du siècle dernier, avant même l'apparition des machines. Depuis Hilbert en 1890, jusqu'à Türing, Church, Gödel, Post dans les années trente... Avant même l'apparition des ordinateurs, Gödel montrait l'existence d'énoncés impossibles à démontrer mathématiquement, comme le problème du pavage du plan par des polyminos quelconques. Cette notion (l'indécidabilité) a été largement utilisée en Informatique depuis. Établir qu'un problème est indécidable est plus fort que dire simplement qu'on ne sait pas le résoudre. Il est en effet alors inutile d'en chercher une solution algorithmique, on peut mieux cerner les problèmes à résoudre. Si un problème est indécidable, on cherchera à le relaxer pour pouvoir résoudre un sous-problème. Nous allons développer un exemple simple qui utilise un argument direct (par contradiction). Plus généralement, il existe deux types de preuves. La principale technique est la technique de diagonalisation qui consiste à partir d'un ensemble infini à construire un nouvel élément qui n'est pas dans cet ensemble. Ceci a permi en particulier d'établir que l'ensemble des nombres réels n'est pas dénombrable. La seconde technique est celle de la réduction dont nous reparlerons largement dans le chapitre suivant et qui sert à prouver l'intractabilité des problèmes.

HALTING PROBLEM
INSTANCE : Un programme écrit dans un langage donné et une entrée
QUESTION : Le programme s'arrête t-il sur cette entrée ?

Le *Halting problem* ou problème de l'arrêt n'est pas une procédure effective, il ne connaît en effet pas de solution calculable, il est indécidable.

Nous détaillons maintenant un exemple de démonstration d'indécidabilité sur ce problème. Nous partons de la thèse de Church-Türing qui stipule que toute machine programmable peut être simulée par une machine de Türing et ainsi, montrer le résultat à partir d'une machine de Türing.

Supposons par contradiction que *Halting problem* soit décidable. Dans ce cas, il existe une machine de Türing (que nous appellerons TM-A) qui s'arrête sur "oui" ou "non" si l'on écrit sur son ruban les instructions d'un programme quelconque et ses données. Cette machine TM-A, étant donné un programme P en entrée et un mot w, peut-être considérée comme une fonction booléenne :

TM-A(P, w) (s'arrête et) retourne la valeur VRAI si le programme P s'arrête pour l'instance w

(s'arrête et) retourne FAUX si P ne s'arrête pas pour l'instance w.

Supposons que TM-A soit constructible, alors on peut créer une autre machine TM-B qui rentrera dans une boucle infinie si et seulement si TM-A accepte son entrée :

TM-B(P) : si TM-A(P, P) alors boucle infinie sinon stop

Cette machine, si on l'applique à elle même, rentre dans une boucle infinie si

et seulement si elle s'arrête lorsqu'on l'applique à elle-même. Cette contradiction permet d'affirmer que TM-A(P, w) n'existe pas.

La démonstration d'indécidabilité précédente était directe. Il existe un autre type de démonstration où l'on déduit l'indécidabilité à partir d'un autre problème dont on sait qu'il est indécidable. C'est par exemple le cas du célèbre problème de Post, qui est assez simple pour nous permettre d'illustrer cette méthode de résolution. On montre l'indécidabilité du langage L_2 connaissant celle du langage L_1. S'il existe une procédure effective qui décide de L_2 alors il en existe une qui décide de L_1. On construit une procédure qui décide de L_1 en se servant de L_1 comme sous-procédure. On conclut que L_2 est indécidable car s'il l'était, L_1 le serait aussi. [1]

POST

INSTANCE : Deux listes de mots $(m_1, m_2, ..., m_p)$ et $(m'_1, m'_2, ..., m'_p)$ construits sur le même alphabet Σ^*.

QUESTION : Peut-on juxtaposer des mots de la première liste de telle sorte que le mot obtenu soit le même qu'en juxtaposant de la même façon les mots de la seconde liste ?

Par exemple, sur l'alphabet $\{0, 1\}$, considérant les deux listes $\{0, 11, 101\}$ et $\{01, 1, 01\}$, on peut faire correspondre les deux listes en juxtaposant les mots en positions $1, 2, 1$ et 3 (on obtient alors 0110101).

Ce problème est indécidable : aucun algorithme ne peut indiquer s'il existe une mise en correspondance quelles que soient les listes de départ. Montrons le théorème de Rice à partir de ce problème. Ce résultat connu stipule qu'il est impossible de décider *a priori* de l'utilité d'une certaine partie d'un programme. Le principe repose une fois encore sur un argument par contradiction : supposons que ce problème soit décidable et montrons qu'alors POST l'est aussi.

Supposons donc qu'il existe un algorithme (noté (A)) qui permette d'indiquer en temps fini que tout partie de code de tout programme soit utile ou non. On considère alors la sous-famille de programmes de la forme suivante :

- Programmer P dépendant des listes $(m_1, m_2, \ldots, m_p)$ et $(m'_1, m'_2, \ldots, m'_p)$.
- **pour** n=1 **jusqu'à** ∞
- rechercher parmi les p^n tentatives possibles s'il existe une correspondance de Post utilisant n mots
- si la réponse est positive alors répondre "oui"

On applique l'algorithme (A) à P et à la partie finale. Ceci indique si POST associé aux paramètres de P admet ou non une solution.

(f) est utile si et seulement si pour tous paramètres de P, POST admet une solution. Disposer de (A) permettrait donc de résoudre le problème de

[1] Remarquons ici que la réduction n'est pas dans le même "sens" que la réduction polynômiale pour prouver qu'un problème est NP-complet. Ici, on englobe le problème dont on sait qu'il est indécidable par le problème dont on veut montrer qu'il l'est.

POST. Comme celui-ci est indécidable, l'hypothèse sur (A) est donc fausse et le problème de Rice est indécidable.

Pour vérifier que cette technique a bien été comprise, nous proposons de prouver que le problème de l'arrêt sur mot vide est indécidable en utilisant l'indécidabilité du Halting problem (voir exercice **??** à la fin).

Pour terminer cette section, signalons que la notion d'indécidabilité est fortement liée aux langages récursivement énumérables. En effet, les procédures effectives reconnues sur une TM correspondent exactement à cette classe de langages.

Exercices du Chapitre I

4.1 Automates à Pile

Pour étendre la puissance des automates d'états finis, on ajoute à un automate (non déterministe) une mémoire infinie (par exemple une pile). L'idée ici est de faire uniquement des manipulations sur le haut de la pile.
Une *pile* est un ruban semi-infini muni d'une tête de lecture avec une contrainte particulière pour le déplacement de la tête : lorsque celle-ci se déplace vers l'origine (à gauche), le contenu du ruban à droite de la tête est effacé. On dispose de plus d'une fonction qui permet de tester si la pile est vide (par exemple en utilisant un symbole spécial de fond de pile).

Définition 5 *Un automate à pile M est un tuple $M = (Q, \Sigma, \Gamma, \Delta, Z, q_0, F)$ où :*

- *Q est un ensemble fini d'états,*
- *Σ est un alphabet d'entrée,*
- *Γ est un alphabet de pile,*
- *$\Delta \subseteq ((Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*))$ la relation de transition (ensemble fini),*
- *$Z \in \Gamma$ est le symbole initial de la pile,*
- *$q_0 \in Q$ l'état initial,*
- *$F \in Q$ l'ensemble des états accepteurs.*

Le triplet $(q, u, \beta) \in Q \times \Sigma^* \times \Gamma^*$ est une *configuration*. La configuration (q', w', α') est *dérivable en une étape* de la configuration (q, w, α) par l'automate à pile M (noté $(q, w, \alpha) \vdash (q', w', \alpha')$) si

$$\exists u \in \Sigma^*, \beta, \delta, \gamma \in \Gamma^*, \quad w = uw', \alpha = \beta\delta, \alpha' = \gamma\delta \text{ et } ((q, u, \beta), (q', \gamma)) \in \Delta$$

.

Une configuration C' est *dérivable en plusieurs étapes* de la configuration C par l'automate à pile M ($C \vdash^* C'$) si $\exists k \geq 0$ et $C_0, \ldots, C_k$ des configurations telles que :

- $C = C_0$,
- $C' = C_k$,
- $C_i \vdash C_{i+1}, \forall i, 0 \leq i < k$.

L'*exécution* d'un automate à pile M sur un mot w est la suite des configurations $(s, w, Z) \vdash (q_1, w_1, \alpha_1) \vdash \ldots \vdash (q_n, \varepsilon, \alpha_n)$. Un mot est *accepté* par un automate à pile M si $(s, w, Z) \vdash^*_M (q, \varepsilon, \gamma)$ avec $q \in F$. Le *langage accepté* par un automate à pile M est l'ensemble $L(M)$ des mots acceptés par M.

Question 1

Construire un automate à pile qui accepte exactement le langage $\{a^n b^n \mid n \in N\}$.

Cependant, on peut montrer que les mots de la forme $a^n b^n c^n$ ne peuvent être reconnus, ce qui motive la recherche d'un modèle encore plus puissant.

Question 2 : Extension

Montrer que l'on peut simuler n'importe quelle machine de Türing à l'aide d'un automate à deux piles.

Indication

Développons l'exemple suivant : soit l'automate $P_1 = (\{s, p, q\}, \{a, b\}, \{A, Z\}, \Delta, Z, s, \{q\})$
avec Δ : { $((s, a, \varepsilon), (s, A))$
$((s, \varepsilon, Z), (q, \varepsilon))$
$((s, b, A), (p, \varepsilon))$
$((p, b, A), (p, \varepsilon))$
$((p, \varepsilon, Z), (q, \varepsilon))\}$
P_1 est un automate à pile.

Sur cet automate, on a la dérivation $(p, baba, AAZ) \vdash (p, aba, AZ)$ par :

- $p = q$, $p = q'$, $w = baba$, $w' = aba$, $\alpha = AAZ$, $\alpha' = AZ$,
- $u = b$, $\beta = A$, $\gamma = \varepsilon$, $\delta = AZ$,
- $((p, b, A), (p, \varepsilon) \in \Delta$.

Le mot $aabb$ est accepté par P_1 par $(s, aabb, Z) \vdash^*_{P_1} (q, \varepsilon, Z)$:

configurations	transitions
$(s, aabb, Z)$	$((s, a, \varepsilon), (s, A))$
(s, abb, AZ)	$((s, a, \varepsilon), (s, A))$
(s, bb, AAZ)	$((s, b, A), (p, \varepsilon))$
(p, b, AZ)	$((p, b, A), (p, \varepsilon))$
(p, ε, Z)	$((p, \varepsilon, Z), (q, \varepsilon))$
$(q, \varepsilon, \varepsilon)$	

4.2 Machine de Turing à ruban infini d'un seul coté

La preuve de ce résultat s'obtient par un principe analogue à précédemment en simulant une TM à ruban infini des deux cotés par une TM à ruban fini d'un coté possédant trois pistes. Deux des pistes sont symétriques à partir d'une case donnée, elles contiennent les informations de la TM de référence. On utilise une bande supplémentaire pour indiquer sur laquelle des deux pistes se trouve l'information courante.

4.3 RAM

Machines à accès aléatoires(ou RAM) ou machines à registres.

4.3.1 Description

Le modèle de calcul RAM a été introduit par Shepherdson et Sturgis en 1963 dans le cadre du calcul sur des entiers, puis étendu aux nombres réels en 1989 par Blum, Shub et Smale. On en trouve des descriptions simples dans les ouvrages de Aho, Hopcroft et Ullman et plus récemment de Wolper ou De Rougemont avec quelques variantes...

C'est un modèle qui se rapproche des ordinateurs que nous manipulons tous les jours. Informellement, une RAM est une machine composées d'un nombre fini de registres R_i (plus un compteur d'instructions CO et un accumulateur ACC) pouvant contenir des entiers de longueur non bornée. [2] Un programme tournant sur une RAM est constitué d'une suite finie d'instructions. Les instructions typiques sont les mouvements en mémoire (READ, STORE), les accès aux registres (LOAD), les opérations de base (ADD), les branchements (JUMP), etc.. On ajoute deux instructions $STOP_A$ et $STOP_R$ qui terminent un programme dans un état accepteur ou non. On dit qu'un programme accepte l'entrée x s'il atteint l'instruction $STOP_A$.

L'entrée est décrite par les contenus des registres R_i (pour $0 \leq i \leq n$, ACC étant considéré comme le registre R_0). Une description instantanée de l'état d'un programme est un couple (j, v) où j est la valeur du compteur d'instructions et v est le vecteur des $n+1$ valeurs des registres.

- L'état initial est $(0, (0, x(1), \ldots, x(n))$.
- Une transition de j à j' est une relation qui donne la nouvelle valeur du compteur d'instructions et des registres après l'exécution de la jème instruction du programme.

On distingue deux modèles de coût pour la mesure du temps d'exécution d'un programme sur une RAM : le modèle uniforme où chaque instruction est de coût unitaire et le modèle logarithmique où il dépend logarithmiquement de la taille des opérandes. La mesure en espace représente le nombre de registres utilisés dans le programme et leurs longueurs, indépendamment de leurs entrées-sorties.

4.3.2 Question 1. Liens entre RAM et TM

Montrer l'équivalence entre une RAM et une TM (c'est-à-dire qu'un problème est décidable sur une TM en temps polynomial si et seulement si il est décidable en temps polynomial sur une RAM).

[2] Notons ici que Wolper introduit des registres finis avec une mémoire infinie... C'est sans doute un peu plus simple.

4.3.3 Indications

Considèrons une TM qui accepte ou rejette une entrée x. On peut définir un programme sur une RAM qui simule cette machine de Türing en associant un registre R_1 à la tête de lecture (plus précisément, il contient la valeur en binaire de sa position) et telle que le code du symbole $x(i)$ du mot d'entrée soit placé dans le registre R_{i+1}.

Chaque état de la TM correspond à une étiquette du compteur d'instructions ($CO_{q,a}$). A cette étiquette, le programme vérifie si le symbole dont l'adresse se trouve dans le registre R_1 est bien a. Si ce n'est pas le cas, le programme s'arrête et rejette x ($STOP_R$). On explicitera la transition $\delta(q, a) = (q', \#, D)$.

4.4 Variante du Halting Problem

On considère ici une variante du problème de l'arrêt d'une machine de Türing où un programme s'arrête ou non sur le mot vide.

Montrer que ce problème est indécidable. On utilisera une réduction à partir du Halting problem.