République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri - Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

Année : 2006-2007
N°d'ordre :
Série :

Mémoire de Magistère

En vue de l'obtention du diplôme de Magistère en Informatique
Option : Génie logiciel et intelligence artificielle

Présenté par:

Mokdad AROUS

Thème

Etude de la complexité de modèles de spécification et de méthodes de vérification des systèmes complexes

Soutenu le ... / ... / 2007 devant le jury composé de :

Pr. Mohammed BENMOHAMMED	Président	Prof	Université Mentouri de Constantine
Dr. Djamel-Eddine SAIDOUNI	Rapporteur	M. C	Université Mentouri de Constantine
Dr. Allaoua CHAOUI	Examinateur	M. C	Université Mentouri de Constantine
Dr. Salim CHIKHI	Examinateur	M. C	Université Mentouri de Constantine

Mémoire préparé au Laboratoire LIRE, Université Mentouri, 25000 Constantine, Algérie

Etude de la complexité de modèles de spécification et de méthodes de vérification des systèmes complexes

Mokdad AROUS

Laboratoire LIRE, Université Mentouri, 25000 Constantine

À mes parents...

À toute la famille AROUS…

Remerciements

Mes louanges et mes gratitudes intarissables vont en premier lieu à Dieu, le tout puissant qui ma prodigué le courage, la volonté et la patience afin d'accomplir ce présent travail.

Ma gratitude, mon profond respect et mes remerciements à tous les membres du jury : M. Mohammed BENMOHAMMED, qui a accepté la lourde tâche d'être président du jury ; M. Allaoua CHAOUI et M. Salim CHIKHI, pour le temps qu'ils ont accordé à l'examen de ce travail.

Plus particulièrement, je tiens à remercier M. Djamel-Eddine SAIDOUNI pour son encadrement continu, pour les remarques constructives qu'il m'a fourni ainsi que pour ses précieux conseils durant tout ce travail. Je suis heureux de pouvoir lui exprimer mes plus vifs remerciements et ma très sincère reconnaissance.

Je tiens également à adresser mes remerciements aux membres du Laboratoire LIRE, plus particulièrement à M. Farid Arfi, M. Adel BENAMIRA et M. Nabil BELALA pour l'aide qu'ils m'ont apporté.

J'adresse mes remerciements à toute ma famille et à tous mes amis qui, de près comme de loin, m'ont aidé et encouragé au moment opportun.

Résumé

La théorie de la complexité computationnelle est un outil important pour diriger l'algorithmique, elle nous aide à développer notre compréhension des difficultés inhérentes au calcul, en connaissant les coûts (le temps d'exécution et l'espace mémoire) des calculs, et en identifiant les différentes sources de complexité. En effet, l'utilisation des différents modèles et méthodes à bon escient doit passer impérativement par l'étude de leur complexité computationnelle. Il s'avère donc important de pouvoir quantifier les besoins en ressources des algorithmes, en particulier en temps et en espace mémoire.

Ce mémoire présente une étude de la complexité asymptotique computationnelle de la génération des STEMs (*Systemes de Transitions Etiquetées Maximales*) réduits, à savoir les STEMs α-réduits et les graphes de pas maximaux GPM, proposés dans le cadre de la vérification formelle des systèmes complexes.

Nous étudions, en premier lieu, la base formelle de la théorie de la complexité, les différentes techniques utilisées et les résultats majeurs trouvés.

Par la suite, nous évaluons la complexité asymptotique computationnelle (temporelle et spatiale) des méthodes de réduction proposées. Nous démontrons que la génération des deux types de STEMs réduits reste gourmande en temps d'exécution. Cependant, les GPMs sont d'une efficacité considérable en termes d'espace mémoire.

Table des matières

Table des figures

Chapitre 1

Introduction

Un objectif principal de l'informatique théorique est de comprendre la quantité de ressources (temps, mémoire, communication… etc.) exigées pour résoudre les problèmes informatiques qui nous intéressent. Etant donnée une description d'un problème particulier, un certain nombre de questions surgissent :

1. Est-il possible de concevoir un algorithme qui résout le problème ?
2. Est-il possible de concevoir un algorithme efficace qui résout le problème ?
3. Comment peut-on indiquer si un algorithme donné est "meilleur" par rapport aux autres algorithmes ? …

À partir de telles questions, deux questions principales peuvent être formulées:

- Qu'est-ce qui est calculable (par une procédure automatique), et qu'est-ce qui ne l'est pas ?
- Qu'est-ce qui est calculable efficacement, et qu'est-ce qui ne l'est pas ?

La théorie de la calculabilité (*computability theory*) et la théorie de la complexité computationnelle [Sim90] [Tra05] (*computational complexity theory*) sont les champs de l'informatique concernés par les questions soulevées plutôt. La théorie de la calculabilité est concernée par l'identification d'une classe particulière des problèmes indécidables (intraitables) : c'est-à-dire des problèmes qu'aucun algorithme effectif (réel) sera capable de résoudre, le but de cette théorie est donc de décider si un problème donné peut être résolu algorithmiquement (au moins en principe) ou non. En fait, de tels problèmes existent, et ils sont même très nombreux, le tout premier d'entre eux est connu sous le nom de "problème de l'arrêt" (*halting problem "HP"*) : il consiste à décider si une machine de Turing s'arrêtera avec une entrée donnée ou non [HC91] [Sto05].

Le but de la théorie de la complexité est d'identifier les problèmes résolubles qui sont intraitables dans le sens qu'aucun algorithme efficace qui les résout n'existe. En fait, il y a plusieurs problèmes informatiques qui sont résolubles (en principe), mais le temps d'exécution de n'importe quel algorithme (connu) qui les résout paraît très long (des

milliards d'années !!), il y a aussi des algorithmes qui utilisent d'énormes quantités de mémoire. Dans la pratique, nous ne pouvons pas résoudre de tels problèmes, sauf pour des cas limités.

Ainsi, la théorie de la complexité est concernée par l'étude des ressources requises durant le calcul pour résoudre un problème donné, dont le but de faire une classification scientifique des ressources exigées pour résoudre les problèmes, et, par conséquent, une classification des problèmes eux mêmes. Les ressources les plus communes sont le temps et l'espace mémoire, d'autres ressources peuvent être aussi considérées comme le nombre de processeurs parallèles dans un système parallèle, le nombre de messages de communication… etc.

L'analyse empirique de la complexité consiste à étudier les quantités de ressources pour un algorithme de manière expérimentale. À cet effet, l'algorithme doit être implémenté et testé sur un ensemble d'entrées de tailles et de compositions différentes. Toutefois cette méthode ne permet de comparer des algorithmes qu'à la condition de disposer d'implémentations réalisées dans un même environnement informatique. D'autre part, les résultats obtenus ne sont pas nécessairement représentatifs de toutes les entrées.

L'analyse théorique consiste à évaluer un pseudo-code de l'algorithme. On utilisera alors la complexité théorique qui permet de considérer une mesure qui rend compte de la qualité intrinsèque des algorithmes indépendamment des détails d'implantation qui interviennent dans la complexité pratique.

On mesure le temps de fonctionnement d'un algorithme comme le nombre de pas pris sur des entrées de longueur *n*, c'est-à-dire, si *A* est un algorithme qui résout un problème, alors le temps d'exécution de *A* est une fonction T : $\aleph$ **à** $\aleph$, de sorte que T(*n*) soit le nombre de pas -dans le pire des cas- pris sur des entrées de longueur *n*. Ainsi nous disons qu'un problème a la complexité en temps T(*n*) s'il y a un algorithme *A* pour ce problème tel que le nombre de pas pris par *A* sur des entrées de longueur *n* est toujours inférieur ou égal à T(*n*).

La complexité en espace mémoire d'un problème est la quantité de mémoire employée par le meilleur algorithme pour ce problème. Ainsi nous disons qu'un problème a la complexité en espace S(*n*) s'il y a un algorithme *A* qui résout ce problème tel que le nombre d'unités de mémoire employées par *A* sur des entrées de longueur *n* est toujours inférieur ou égal à S(*n*).

Ces notions nous permettent de classifier les problèmes selon leurs complexités en temps et en espace mémoire. Une classe de complexité est une collection de problèmes d'une certaine caractéristique de difficulté. Dans une classe, quelques problèmes peuvent être plus faciles que d'autres, mais tous les problèmes peuvent être résolus dans les limites de ressources liées à cette classe [GHR95].

Un autre but principal de l'étude de la complexité est l'arrangement des relations entre les différents problèmes informatiques, et entre les différents modes de calcul. Dans ce stade, la théorie de la complexité se base sur la thèse suivante:

***Thèse 1.1 (Invariance thesis)*:** Les machines "raisonnables" peuvent simuler l'une l'autre avec frais bornés polynomialement en temps, et avec frais de facteur constant en espace.

C'est-à-dire, si M et $M¢$ sont des machines raisonnables, $T_M(x)$ et $S_M(x)$ sont respectivement les quantités de temps et d'espace mémoire nécessaires pour que la machine M traite une entrée x, alors il existe un constant k tel que :

$$T_{M'}(x) \leq T_M(x)^k$$

$$S_{M'}(x) \leq k.S_M(x)$$

Par exemple, La simulation de tout algorithme par un programme de machine de Turing introduit au plus un ralentissement quadratique, et un facteur constant sur l'espace [Bou03]. En fait, depuis 1936, l'année où la machine de Turing a lieu, de nombreux autres modèles de calcul ont été proposés pour caractériser la notion d'algorithme (comme le lambda-calcul). Tous se sont révélés avoir une puissance de calcul équivalente à celle de machine de Turing, qui tient donc toujours lieu de modèle de référence.

En se basant sur les fondements de la théorie de la complexité, notre travail dans ce mémoire consiste à étudier la complexité en termes de temps de calcul et d'espace mémoire de méthodes proposées pour la génération des graphes d'états réduits dans le contexte général de la vérification formelle des systèmes complexes.

Problématique

Dans le cadre des travaux de notre équipe, différents modèles de spécifications et méthodes de vérification ont été proposés et implémentés, l'utilisation de ces modèles et méthodes à bon escient doit passer impérativement par l'étude de leur complexité (computationnelle). En fait, la théorie de la complexité est un outil important pour diriger l'algorithmique, elle nous aide à développer notre compréhension des difficultés inhérentes au calcul, en connaissant le coût (le temps d'exécution et l'espace mémoire) et en identifiant les différentes sources de complexité.

Contribution

Le présent mémoire présente une étude de la complexité computationnelle asympthotique des algorithmes de réduction des STEMs (*Systemes de Transitions Etiquetées Maximales*) proposés au niveau de notre équipe. Le travail se divise en deux parties :

- La maîtrise de la théorie de la complexité, en étudiant la base formelle de ce champs de l'informatique théorique.
- L'application des principes de cette théorie pour l'étude de la complexité temporelle et spatiale des méthodes et algorithmes développés au sein de notre équipe. Nous nous intéressons ici aux algorithmes proposés pour la génération des STEMs réduits (le STEM α-réduit et le graphe de pas maximaux GPM) en vue de la résolution de problème de l'explosion combinatoire du graphe d'états dans la vérification formelle des systèmes complexes.

Plan du document

Le mémoire est organisé de la manière suivante :

- Le chapitre 2 présente la base théorique formelle de la théorie de la complexité, nous introduisons les concepts nécessaires à la compréhension de ce document, en définissant les différents concepts clefs, qui se basent sur deux notions : modèle de machine et computation sur modèle de machine.
- Le chapitre 3 fournit un catalogue des classes de complexité les plus populaires : leurs définitions, propriétés et les types de problèmes qu'elles contiennent.
- Le chapitre 4 présente la démarche générale pour calculer la complexité computationnelle (temporelle et spatiale) d'un algorithme, pour cela, quelques hypothèses sur le modèle de machine considéré et sur la taille des données manipulées sont précisées.
- Le chapitre 5 présente l'étude de la complexité asymptotique temporelle et spatiale des algorithmes proposés pour la génération des STEMs réduits.
- Le chapitre 6 donne quelques conclusions et perspectives du travail présenté dans ce mémoire.

Chapitre 2

Théorie de la complexité

Le but de ce chapitre est de présenter la base théorique formelle de plusieurs concepts clefs dans la théorie de la complexité, en commençant par les notions de modèle de machine, computation, complexité, classe de complexité, réduction et complétude. En suite, quelques modèles de machine séquentielle et parallèle seront exposés.

2.1. Complexité et Simulation

Avant de parler de la difficulté de résoudre un problème, de fait que les mesures de temps et d'espace doivent être fondues sur un modèle de calcul, on doit premièrement définir les modèles de machine convenables dans lesquels les computations sont décrites.

2.1.1. Machine et computation

2.1.1.1. Modèle de machine

Un modèle de machine est une classe de dispositifs M_i ($i \geq 0$) structurés similairement, appelés "machines", qui peuvent être décrites comme des objets mathématiques sous forme des n-uplets d'ensembles finis. Dans les définitions des modèles de machine, deux notions sont communes : la présence d'un objet fini appelé "programme" (ou "contrôle fini") qui opère sur une structure appelée "mémoire".

Le contrôle fini réfère à certaine notion de "processeur" pour l'exécution du programme, à tout moment le contrôle fini est dans un "état" particulier, et le nombre de différents états possibles est fini. Le programme est écrit en un certain type de "langage de programmation", typiquement il y a des instructions pour amener ou enregistrer des informations dans la mémoire, modifier les informations dans des cellules accessibles en mémoire, tester des conditions, exécuter des branchements conditionnels ou inconditionnels dans le programme, et pour exécuter des entrées/sorties.

La mémoire est simplement un dépôt de morceaux finis d'information composée de symboles d'un ensemble fini appelé "alphabet", cet alphabet est spécifié dans le n-uplet qui définit la machine. La mémoire est toujours modélisée comme une structure régulière de "cellules" où l'information est enregistrée, cette structure est généralement linéaire. La mémoire d'un modèle de machine est, en principe, infinie, seulement une partie finie de la

mémoire est utilisée en toute computation réaliste [Emd90]. Deux sections spéciales de mémoire sont distinguées : la section d'*entrée* à partir de laquelle les informations sont lues, et la section de *sortie* à travers de laquelle les informations sont communiquées au monde extérieur.

2.1.1.2. Notion de configuration

Pour formaliser la notion de "computation", il est nécessaire d'introduire la notion de "configuration". Une configuration est une description complète de l'état de la machine, typiquement elle se compose d'un pointeur sur l'instruction courante dans le programme, le contenu de la partie finie de la mémoire qui a participée aux calculs, et l'état des sections d'entrée et de sortie de la mémoire.

- ***Relation de transition* :**

On définit la relation de transition entre deux configurations de la machine comme suit : une configuration C_2 est obtenue par transition d'une configuration C_1 si l'instruction du programme qui est imminente en C_1 prend la machine de C_1 à C_2, c'est-à-dire que la configuration C_2 est le successeur direct de la configuration C_1 par le programme de la machine. Cette transition est notée : $C_1 \vdash C_2$, et sa fermeture réflexive et transitive est notée $\vdash^*$.

Nous pouvons maintenant faire la distinction entre machines "déterministes" et machines "non déterministes" : Dans une machine déterministe, pour chaque configuration C_1 il existe au plus une configuration C_2 telle que $C_1 \vdash C_2$. Tandis que dans une machine non déterministe il peut exister plusieurs configurations C_2.

2.1.1.3. Notion de computation

Une computation réfère principalement aux séquences de configurations connectées par la relation de transition $\vdash$, on a introduit pour ces séquences les notions de configuration "initiale", configuration "accessible", configuration "finale", configuration "acceptante" et configuration "rejetante".

- *Configuration initiale* : elle consiste en un état initial dans le contrôle fini (indiqué dans le n-uplet décrivant la machine), une mémoire vide (sauf possiblement la section d'entrée) et une section de sortie vide. Une configuration initiale d'une machine M_i qui est déterminée complètement par une entrée x est notée $C_i(x)$.
- *Configuration accessible* : une configuration accessible est une configuration qui peut être obtenue par certaine computation à partir d'une configuration initiale.
- *Configuration finale* : une configuration C est dite finale s'il n'existe aucune configuration C' telle que $C \vdash C'$. Si l'état dans le contrôle fini dans la configuration finale égale un état acceptant spécifique (décrit dans le n-uplet décrivant la machine), alors la configuration finale est dite *acceptante* (*accepting configuration*), sinon elle est dite *rejetante* (*rejecting configuration*).

Une computation complète est une computation qui commence par une configuration initiale et qui se termine par une configuration finale [Emd90]. Donc, une computation peut être infinie, dans ce cas elle est appelée une computation "divergente" (*divergent computation*), ou elle peut se terminer en une configuration finale, cette computation

terminante (*terminating computation*) est dite "acceptante" ou "rejetante" selon sa configuration finale soit acceptante ou rejetante respectivement.

Les calculs machines peuvent être utilisés pour plusieurs intentions : accepter un langage, reconnaître un langage, ou calculer une relation.

- Le langage $D(M_i)$, accepté par une machine M_i, se compose des entrées x pour lesquelles il existe une computation terminante commençant par $C_i(x)$.
- Le langage $L(M_i)$, reconnu par une machine M_i, se compose des entrées x pour lesquelles une computation acceptante commençant par $C_i(x)$ peut être construite, de plus, tous les calculs de la machine M_i doivent être terminants.
- La relation $R(M_i)$, calculée par la machine M_i, se compose des paires $<x,y>$, d'entrées x et de sorties y, tel qu'il existe une computation acceptante qui commence par la configuration $C_i(x)$ et qui se termine par une configuration où y dénote le contenu de la section de sortie.

Dans cette formalisation, il existe un seul contrôle fini (ou programme) qui est connecté à la mémoire par possiblement plus d'un canal, par conséquent la machine exécutera une seule instruction en même temps, ce type de machine est appelé "modèle de machine séquentielle". Le "modèle de machine parallèle" est obtenu si on remplace le contrôle fini par un ensemble de programmes ou processeurs. Le nombre de processeurs n'est pas borné, mais à toute configuration accessible, seulement un nombre fini de processeurs est activé dans la computation [Emd90]. Dans certains modèles, une quantité de mémoire (mémoire partagée) est accessible à tous les processeurs en plus de mémoire locale de chaque processeur. Les processeurs peuvent communiquer entre eux directement à travers certain réseau d'interconnexion, ou indirectement à travers la mémoire partagée (ou par les deux manières).

2.1.2. Simulation

Il est difficile de donner pour la simulation une définition mathématique et suffisamment générale en même temps. Intuitivement, une simulation d'un modèle de machine M par un modèle M' est une certaine construction qui montre que tout ce que peut être fait sur une entrée x par une machine $M_i \in M$, peut être aussi bien fait par certaine machine $M'_i \in M'$ sur la même entrée.

On peut introduire une relation entre les configurations C de M_i et C' de M'_i pour exprimer que la configuration C' représente ou simule la configuration C, on note telle relation par $C \text{»} C'$. Pour la simulation, on peut maintenant exiger la condition suivante [Emd90] :

$$\forall\, C_1, C_1', C_2 \left((C_1 \vdash C_2 \wedge C_1 \text{»} C_1') \Rightarrow \exists\, C_2' \left[(C_1' \vdash C_2' \wedge C_2 \text{»} C_2') \right] \right) \qquad (2.1)$$

Ou la condition plus légère :

$$\forall\, C_1, C_1', C_2 \left((C_1 \vdash C_2 \wedge C_1 \text{»} C_1') \Rightarrow \exists\, C_2' \left[(C_1' \vdash^* C_2' \wedge C_2 \text{»} C_2') \right] \right) \qquad (2.2)$$

Mais ces conditions sont trop restrictives : la première condition implique que le temps d'exécution d'une computation est préservé par la simulation, cependant la deuxième condition exige que chaque configuration de M_i a son analogue dans la computation de M_i' et que l'ordre général de la computation de M_i est préservé par la simulation.

2.1.3. Mesure de la complexité

Etant donnée la description intuitive d'une computation, il est facile de définir la mesure de temps pour une computation terminante: simplement prendre le nombre de configurations dans la séquence qui décrit la computation, en attribuant une unité de temps à chaque transition. Cette mesure est appelée : "temps uniforme", et elle est appliquée dans les deux modèles de machine: séquentielle et parallèle, mais cette mesure peut être peu réaliste dans certains modèles [Emd90]. On peut utiliser une mesure raffinée, dans laquelle chaque transition est pesée suivant la quantité d'information qu'elle manipule.

Pour la mesure de l'espace consommé par une computation, une mesure brute consiste en nombre de cellules mémoires qui ont été affectées par la computation, mais aussi une mesure plus raffinée prend en compte combien d'informations enregistrées dans chaque cellule.

Après avoir défini la notion du temps et de l'espace pour chaque computation individuelle, on peut définir ces mesures pour la machine en totalité, on ne s'intéresse pas aux besoins en temps et en espace pour une entrée spécifique, mais à la relation globale entre la mesure du temps et de l'espace d'une computation et la longueur d'une entrée x, notée $|x|$. La longueur $|x|$ de l'entrée dépend du codage de x pour une machine particulière. De plus, la façon dont ces mesures sont définies dépend de ce que la machine a l'intention de faire avec l'entrée x : reconnaître, accepter ou calculer [Emd90].

Soit $f(n)$ une fonction de l'ensemble des entiers non négatifs $\aleph^+$ à lui-même :

Définition 2.1

- Une machine M_i est dite terminante en temps (resp. espace) $f(n)$ si toute computation sur une entrée x de longueur n consomme un temps (resp. espace) $\leq f(n)$.
- Une machine déterministe M_i est dite acceptante en temps (resp. espace) $f(n)$ si toute computation terminante sur une entrée de longueur n consomme un temps (resp. espace) $\leq f(n)$.
- Une machine non déterministe M_i est acceptante en temps (resp. espace) $f(n)$ si pour toute entrée acceptée de longueur n existe une computation terminante qui consomme un temps (resp. espace) $\leq f(n)$.

Définition 2.2

- Un langage $L=L(M_i)$ est reconnu en temps (resp. espace) $f(n)$ par une machine M_i si M_i est terminante en temps (resp. espace) $f(n)$.
- Un langage $L=D(M_i)$ est accepté en temps (resp. espace) $f(n)$ par une machine M_i si M_i est acceptante en temps (resp. espace) $f(n)$.

2.1.4. Classes de complexité

Une classe de complexité est une collection de problèmes (langages) d'une certaine caractéristique de difficulté, ces problèmes sont décidables donc pour une quantité de ressources donnée. Typiquement, une classe de complexité est définie par :

i) un modèle de calcul,

ii) une ressource (ou une collection de ressources), et

iii) une fonction connue comme la limite de la quantité de ressource.

Dans une classe, quelques problèmes peuvent être plus faciles que d'autres, mais tous les problèmes peuvent être résolus dans les limites de ressources liées à la classe [GHR95]. Nous pouvons maintenant définir les classes de complexité de la manière suivante :

Définition 2.3

- L'ensemble des langages reconnus par un modèle de machine M en temps (resp. espace) $f(n)$ se compose de tous les langages L qui sont reconnus en temps (resp. espace) $f(n)$ par certain membre M_i de M, on note cette classe par : M-$TIME(f(n))$ (resp. M-$SPACE$ $(f(n))$).
- La classe des langages reconnus simultanément en temps $f(n)$ et en espace $g(n)$ par un modèle de machines M est notée par : M-$TIME$ & $SPACE$ $(f(n), g(n))$.
- L'intersection des classes M-$TIME$ $(f(n))$ et M-$SPACE$ $(g(n))$ se compose des langages reconnus en temps $f(n)$ et en espace $g(n)$, et est notée par: M-$TIME,SPACE$ $(f(n), g(n))$.

Nous nous intéressons particulièrement aux classes F des fonctions de limite de ressources suivantes :

$\text{Log} = \{k.\ \log\ n / k \in \aleph^+\}$

$\text{Lin} = \{k.\ n\ / k \in \aleph^+\}$

$\text{Poly} = \{k.\ n^k / k \in \aleph^+\}$

$\text{Expl} = \{k.\ 2^{k.n} / k \in \aleph^+\}$

$\text{Exp} = \{k\ .\ 2^{n^k} / k \in \aleph^+\}$

Un ensemble L est reconnu en temps (resp. espace) F s'il est reconnu en temps (resp. espace) f, pour certaine fonction $f \in F$.

Pour n'importe quel modèle de machine M, on définit les classes de complexité de base suivantes :

Définition 2.4

M-LOGSPACE = $\{L$ /L est reconnu par $M_i \in M$ en espace $Log\}$.

M-PTIME = $\{L$ /L est reconnu par $M_i \in M$ en temps $Poly\}$.

M-PSPACE = $\{L$ /L est reconnu par $M_i \in M$ en espace $Poly\}$.

M-ETIME = $\{L$ /L est reconnu par $M_i \in M$ en temps $Expl\}$.

M-EXPTIME = $\{L$ /L est reconnu par $M_i \in M$ en temps $Exp\}$.

M-ESPACE= $\{L$ /L est reconnu par $M_i \in M$ en espace $Expl\}$.

M-EXPSPACE= $\{L$ /L est reconnu par $M_i \in M$ en espace $Exp\}$.

Par exemple, on dit que la classe M-PTIME est la classe des problèmes pouvant être résolus en temps polynomial par une machine M_i du modèle M, c'est-à-dire les problèmes

pour lesquels le temps d'exécution est $O(n^k)$, pour $k \in \aleph^+$, (la notation *O* sera expliquée dans la section ***4.3.2.1***).

En dénotant les machines de Turing déterministes par *T* et les machines de Turing non déterministes par *NT*, on obtient les définitions suivantes des classes de complexité :

Définition 2.5

LOGSPACE = *T*-LOGSPACE	NLOGSPACE=*T*-NLOGSPACE
P=*T*-PTIME	NP=*T*-NPTIME
PSPACE=*T*-PSPACE	NPSPACE=*T*-NPSPACE
EXPTIME=*T*-EXPTIME	NEXPTIME=*T*-NEXPTIME
EXPSPACE=*T*-EXPSPACE	NEXPSPACE=*T*-NEXPSPACE

La notation où la ressource est préfixée par *N* plutôt que le modèle de machine lui-même est de [Emd90], par exemple *T*-NPTIME est la classe dénotée en principe par *NT*-PTIME.

2.1.5. Coûts de simulation et classes de machine

Comme on a dit précédemment, on n'a pas une réponse précise de ce quoi une simulation, mais on peut maintenant au moins quantifier les frais généraux de simulations comme suit [Emd90]:

Définition 2.6 Pour des modèles de machine *M* et *M´*, on dit que *M´* simule *M* avec coût de temps (resp. espace) $f(n)$ s'il existe une fonction *s* telle que : pour toute machine $M_i \in M$ la machine $M'_{s(i)} \in M'$ simule M_i ; et pour toute entrée *x* de *M*, si $c(x)$ est l'entrée encodée qui représente *x* pour $M'_{s(i)}$, et si $t(x)$ est la limite de temps (resp. espace) exigé par M_i pour traiter *x*, alors le temps (resp. espace) requis par $M'_{s(i)}$ pour traiter $c(x)$ est borné par $f(t(x))$.

Dans cette définition, le mot "traiter" veut dire : reconnaître, accepter, rejeter une entrée, ou évaluer certaine fonction de l'entrée.

La simulation de *M* par *M´* avec frais de temps ou d'espace $f(n)$ sera notée par $M \leq M'(time\ f(n))$ ou $M \leq M'(space\ f(n))$ respectivement.

Si une seule simulation atteint les deux limites de temps $f(n)$ et d'espace $g(n)$, alors elle est notée: $M \leq M'(time\ f(n)\ \&\ space\ g(n))$.

Si les deux limites peuvent être atteintes mais pas nécessairement dans la même simulation, alors on note : $M \leq M'(time\ f(n), space\ g(n))$.

Si la simulation existe dans les deux directions, accomplissant des frais généraux semblables, alors on remplace le symbole $\leq$ par le symbole d'équivalence $\approx$ [Emd90], c'est à dire que :

$$(M \leq M'\ (time\ f(n)) \wedge M' \leq M\ (time\ f(n))\ ) \Rightarrow\ M \approx M'\ (time\ f(n))$$

$$(M \leq M'\ (space\ f(n)) \wedge M' \leq M\ (space\ f(n))\ ) \Rightarrow M \approx M'\ (space\ f(n))$$

Pour les classes des fonctions *F*, on dit que *M´* simule *M* avec frais de temps ou d'espace *F* si la simulation se maintient pour certaine fonction $f \in F$, ceci sera noté par $M \leq M'(time\ F)$ ou $M \leq M'(space\ F)$ respectivement. On peut maintenant définir "les classes de simulation" suivantes :

1) $M \leq M'$ (*time Poly*) simulation en temps polynomial,

2) $M \leq M'$ (*time Lin*) simulation en temps linéaire,

3) $M \leq M'$ (*space Lin*) simulation de facteur constant en espace.

Deux cas spéciaux de la simulation en temps linéaire sont celles appelées : simulation en "temps réel" et simulation en "délai constant". Chacune de ces deux simulations se comporte selon l'une des deux conditions (2.1) et (2.2) de la section ***2.1.2*** : la simulation en délai constant correspond à la deuxième condition où le nombre de configurations de M' dans la simulation de deux configurations successives de M est borné par un constant, pour la simulation en temps réel ce constant est égal à 1 comme il est indiqué dans la première condition, pour cette dernière simulation on utilise la notation : $M \leq M'$(*real-time*).

On peut maintenant donner la définition de la première et de la deuxième classe de machine [Emd90] :

Définition 2.7 Soient T le modèle de machine de Turing et M un modèle de machine arbitraire.

- M est un modèle de la première classe de machine si: $T \gg M$ (*time Poly & space Lin*).
- M est un modèle de la deuxième classe de machine si : PSPACE = M-PTIME = M-NPTIME.

La première classe de machine représente la classe de modèles séquentiels raisonnables satisfont la thèse de l'invariance (thèse 1.1) avec respect au modèle de machine de Turing traditionnelle. Alors que les machines de la deuxième classe sont celles qui satisfont la thèse du calcul parallèle (thèse 2.1 de la section 2.3) avec respect au modèle de machine de Turing séquentielle traditionnelle et pour les deux versions déterministe et non déterministe : Tout ce que peut être résolu en espace borné polynomialement sur une machine de Turing traditionnelle, peut être résolu en temps borné polynomialement sur une machine de la deuxième classe, et vice versa. Dans le cas où un modèle de machine n'a pas une version non déterministe, la dernière condition (concernant M-NPTIME) est omise.

2.1.6. Réduction et complétude

2.1.6.1. Réduction

La notion de réduction est une notion cruciale dans la théorie de la complexité : Supposant que nous avons deux langages L et L', nous voulons déterminer si un certain mot x est un membre de L, supposant aussi que nous pouvons transformer x en un mot y, en utilisant une fonction f, de manière que y soit un membre de L' exactement si x est un membre de L, c'est-à-dire que $x \in L$ si et seulement si $f(x) \in L'$. On dit que nous avons réduit L à L' dans le sens que si nous savons examiner l'adhésion à L', alors nous pouvons examiner l'adhésion à L. La fonction f s'appelle "réduction". Donc le langage L est "réductible" au langage L', en écrivant $L \leq_m L'$, s'il y a une fonction f telle que $x \in L$ si et seulement si $f(x) \in L'$, cette réduction est appelée "*many-one réduction*". Cette réduction est appelée *many-one* parce que beaucoup d'exemples distincts du problème original L peuvent être transformés en un exemple simple du nouveau problème L' [GHR95]. Si nous savons réellement décider l'adhésion à L', alors la complexité d'examiner l'adhésion de x à L

devient la somme de la complexité computationnelle de $f(x)$ et de la complexité de tester si $f(x) \in L'$ [GHR95], où la complexité de calculer la fonction f est appelée "la complexité de réduction" [Lok03].

Dans la théorie de la complexité, il est ordinairement requis que f puisse être évaluée en temps polynomial, dans ce cas la réduction est appelée "réduction en temps polynomial" (*polynomial-time reduction*) ou *P-reduction*, et sera notée par : $L \leq_m^p L'$. Si f peut être évaluée en espace logarithmique, alors on parle de "la réduction en espace logarithmique" (*logspace reduction*).

Dans la théorie de la complexité, une réduction d'un problème A à un problème B ($A \leq_m^p B$) établit le fait que A est facile si B est facile, par conséquent si B est difficile alors A est aussi difficile. L'idée de la réduction polynomiale est que : si $A \leq_m^p B$, alors n'importe quelle solution à B peut être convertie rapidement en solution à A (figure 2.1), on peut donc penser à cette signification : "A n'est pas plus difficile que B" [Hir04].

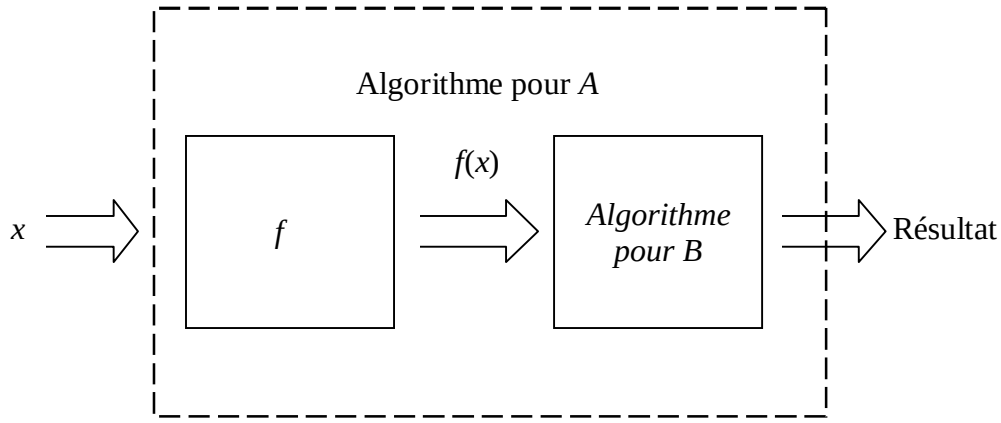


Figure 2. 1 : Réduction de A à B

***Exemple de P-reduction* :**

Considérons le problème *TSDP* (*Travelling Salesman Decision Problem*) (qui sera défini dans la section ***3.2.1***) : c'est une sorte du problème célèbre de "voyageur de commerce" *TSP*, où l'entrée consiste en un graphe G étiqueté avec des entiers non négatifs et un entier non négatif d, on veut décider s'il existe un circuit de G de poids total $\leq d$ (au lieu de trouver le plus court chemin dans le problème *TSP*).

Considérons aussi le problème *HCP* (*Hamiltonian Circuit Problem*), dans lequel on veut décider si un graphe G contient un circuit "Hamiltonien" : ceci est une séquence de nœuds distincts $n_0, n_1, \ldots n_k$ de G, tel que :

i) chaque nœud de G est dans la séquence, et

ii) chaque paire de (n_0, n_1), (n_1, n_2),… (n_{k-1}, n_k) et (n_k, n_0) est un arc de G.

On peut montrer que $HCP \leq_m^p TSDP$, de la manière suivante [Hir04]:

Prendre le graphe *G*, construire un graphe étiqueté *G'* avec les mêmes nœuds de *G*, si (*m*,*n*) est un arc de *G* alors *G'*(*m*,*n*) = 0, si (*m*,*n*) n'est pas un arc de *G* alors *G'*(*m*,*n*) = 1, si on sait comment décider si (*G'*, *d*= 0) est une instance positive (instance avec la réponse *oui*) pour le *TSDP*, alors on peut immédiatement décider que *G* contient un circuit Hamiltonien.

En effet, si *G* est instance positive pour le *HCP* alors il y a un circuit hamiltonien γ de *G*, γ devient un circuit de $G¢$ avec un poids total 0, d'où ($G¢$, 0) est une instance positive pour le *TSDP*. Par contre, si *G* est instance négative pour le *HCP*, alors toute séquence de nœuds distincts γ qui couvre tous les nœuds de *G* doit inclure une paire de nœuds consécutifs comme arc erroné, par conséquent tout circuit de $G¢$ inclut au moins un arc de poids 1, et donc le poids total est ≥1. Ainsi ($G¢$, 0) n'est pas une instance positive pour *TSDP*.

Nous pouvons donc déduire qu'un algorithme qui résout le *TSDP* peut être converti rapidement en un algorithme pour résoudre le *HCP*, et s'il n'y a pas un algorithme pour le *HCP* alors il n'y a pas un algorithme pour le *TSDP*, par conséquent, on dit que le *HCP* n'est pas plus difficile que le *TSDP*.

2.1.6.2. Complétude

Les réductions permettent de comparer des problèmes entre eux. En plus, elles peuvent être employées pour rapporter un problème à une classe de complexité entière, en introduisant les notions de "difficulté" (*hardness*) et "complétude" (*completeness*) [Emd90].

Définition 2.8 Soit *B* un sous ensemble de Σ^*, et *X* une famille d'ensembles.

- *B* est dit $\leq_m^p$*-difficile* ($\leq_m^p$*-Hard*) pour *X* (ou simplement *X-difficile* (*X-Hard*)) si $A \leq_m^p B$, pour tout $A \in X$.

- *B* est dit $\leq_m^p$*-Complet* pour *X* (ou *X-Complet*) si *B* est *X-difficile* et $B \in X$.

C'est-à-dire que : $(\forall A \in X : A \leq_m^p B) \Rightarrow$ *B* est *X-difficile*.

$$(B \text{ est } X\text{-difficile et } B \in X) \Rightarrow B \text{ est } X\text{-Complet}.$$

Quand nous disons qu'un problème est *X-difficile*, nous indiquons qu'il est aussi difficile à résoudre que n'importe quel autre problème dans *X*, le fait qu'il est *X-complet* ajoute l'information additionnelle que le problème est en fait dans *X*. La résolution donc d'un tel problème permet la résolution de n'importe quel autre problème dans *X*.

Exemples **:** Le problème paradigmatique de la classe *NP* est le problème *SAT* (*Satisfiability*) : il s'agit de savoir si une formule propositionnelle donnée est satisfiable ou non. Ce problème, selon le théorème de *Cook*, est *NP-Complet*, il est montré que tout problème dans la classe *NP* est réductible au problème *SAT* en espace logarithmique en la taille de la donnée [Jou].

2.2. Modèles de machine séquentielle

On définit la complexité des langages relativement à un modèle de calcul particulier, en effet, la mesure de la complexité est fortement relative au modèle de computation, mais quel modèle de computation est approprié à la théorie de la complexité ?

Peut-être il n'y a pas uniquement un modèle de computation approprié pour la théorie de la complexité [Yap98]. Ceci suggère qu'on examine différents modèles et découvrit leurs interdépendances. Mais, on s'aperçoit vite que tous les modèles de la calculabilité connus donnent plus ou moins les mêmes notions de complexité à un facteur polynomial près [Jou]: ce qui est linéaire dans un modèle peut très bien devenir quadratique dans un autre !

2.2.1. Machine de Turing (*Turing machine*)

Les machines de Turing ont été introduites en 1936 par le mathématicien Britannique *Alen TURING*. Elles ont établies elles-mêmes comme le modèle de machine standard dans la théorie de la complexité contemporaine.

Dans cette section nous allons décrire la version de base de machine de Turing. Des caractéristiques supplémentaires peuvent être considérées, comme la structure, la dimension et le nombre de bandes, le nombre de têtes sur la bande et des restrictions sur l'utilisation des bandes. Chaque sélection de certain ensemble particulier de telles caractéristiques aboutit à une autre version du modèle de machine de Turing.

2.2.1.1. Description du modèle de machine de Turing

Conformément à la figure 2.2, les machines de Turing considérées sont dotées :

- d'une bande de lecture bi-infinie, sur laquelle la donnée, mot sur le vocabulaire Σ, sera écrite, entourée à l'infini de caractères blancs, considéré comme notre caractère spécial, noté *b* (*blank*), n'appartenant pas à Σ,
- d'un nombre fini de bandes de travail qui peuvent stocker des mots entourés à l'infini de blancs sur les vocabulaires respectifs $V_1, V_2, \ldots V_n$ ne contenant pas le caractère blanc,
- d'un contrôle matérialisé par un ensemble fini d'états : *K*,
- d'un état initial q_0,
- d'un ensemble d'états acceptants,
- et d'une fonction de transition, qui est une application de $K \times \Sigma \times V_1 \times \ldots V_n$ dans :
  - $K \times \{L,O,R\} \times \{L,O,R\} \times \ldots \times \{L,O,R\}$: si la machine est déterministe,
  - $P(K \times \{L,O,R\} \times \{L,O,R\} \times \ldots \times \{L,O,R\})$: si la machine est non déterministe.

Dans le modèle de machine de Turing, la mémoire se compose d'une collection finie de bandes divisées en cellules, chaque cellule est capable d'enregistrer un symbole unique de l'alphabet de la bande, pour chaque bande il existe un ou plusieurs canaux (têtes de lecture/écriture) connectant le contrôle fini à la bande, chaque tête est positionnée à certaine cellule de la bande, et il est possible que plusieurs têtes soient situées dans la même cellule.

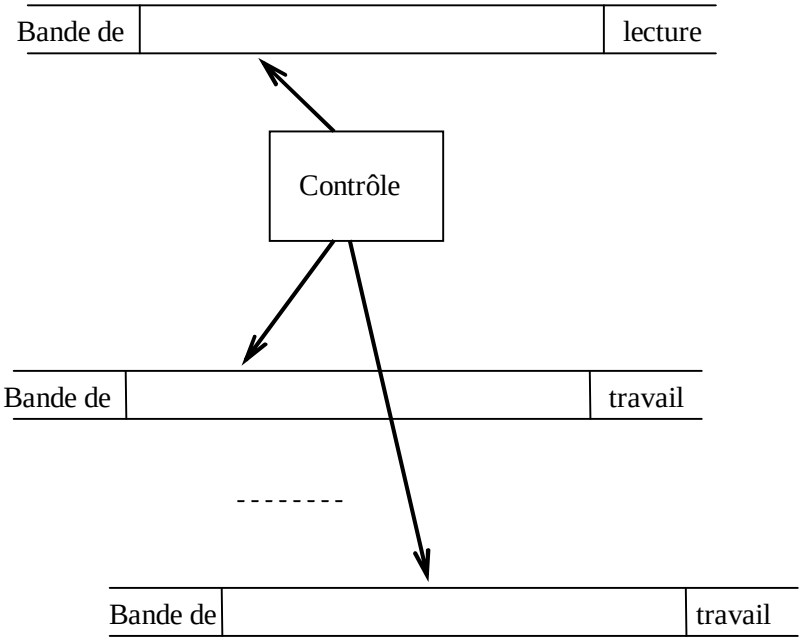


Figure 2. 2 : Machine de Turing

Dans le programme de machine de Turing, plusieurs types d'instructions ont été intégrées en une seule instruction : la machine lit l'ensemble ordonné des symboles à partir des bandes qui sont actuellement scannées par les têtes. Selon ces symboles et l'état interne du contrôle fini, l'instruction appropriée causera la machine à réécrire certains ou tous les symboles scannés par des nouveaux symboles, déplacer certaines ou toutes les têtes à une cellule adjacente dans la bande correspondante et passer le contrôle fini à un nouvel état.

Les machines de Turing sont une abstraction des ordinateurs [Car06] : La partie de contrôle représente le microprocesseur. Un élément essentiel est que le nombre d'états est fini, ceci prend en compte que les microprocesseurs possèdent un nombre déterminé de registres d'une taille fixe et que le nombre de configurations possibles est fini. La bande représente la mémoire de l'ordinateur, ceci comprend la mémoire centrale ainsi que les mémoires externes comme les disques durs. Contrairement à un ordinateur, la mémoire d'une machine de Turing est infinie, ceci prend en compte qu'on peut ajouter des disques durs à un ordinateur de façon (presque) illimitée. Une autre différence entre une machine de Turing et un ordinateur est que l'ordinateur peut accéder à la mémoire de manière directe (appelée aussi aléatoire), alors que la tête de lecture/écriture de la machine de Turing se déplace d'une seule position à chaque opération.

Nous donnons maintenant la définition formelle de la version la plus simple de la machine de Turing : c'est la machine de Turing avec une seule bande (*single-tape Turing machine*).

Définition 2.9 Une machine de Turing M est un 7-uplet $M=<K, \Sigma, P, q_0, q_r, b, D>$, tel que:

- K : est un ensemble fini d'états internes,

- Σ : est l'alphabet de la bande, c'est un ensemble fini de symboles qui ne contient pas le symbole blanc *b*.
- *P* : est le programme enregistré dans le contrôle fini, il se compose d'un ensemble fini de quintuples $<q, s, q', s', m> \in K\times\Sigma\times K\times\Sigma\times D$; avec $D = \{L, O, R\}$ est l'ensemble de mouvements possibles de la tête: *Left*, *No move* ou *Right*. Ce quintuple a pour dire que si le contrôle fini est en état *q* et la tête scanne le symbole *s*, alors écrire le symbole *s´*, déplacer la tête suivant *m* et mettre le contrôle fini à l'état *q´*,
- q_0 et q_r : sont deux éléments spéciaux dans *K* dénotant l'état initial et l'état final respectivement,
- *b* : est le symbole spécial *blank* ($\notin \Sigma$).

Une configuration de la machine de Turing est donnée par la description du ruban, la position de la tête de lecture/écriture, et l'état interne. Elle peut être notée sous la forme *$*Σ*K*Σ**$* [Emd90], où le symbole d'état est écrit devant le symbole actuellement scanné.

Les composants d'une instruction peuvent être toujours subdivisés en une partie d'observation et une partie d'action : dans les quintuples du programme *P*, le premier état et symbole représentent l'observation, et le reste représente l'action. Si une seule observation peut mener à au plus une action, donc la machine est dite déterministe (notée DTM pour "*Deterministic Turing Machine*"), sinon elle est non déterministe (notée NDTM pour "*Non Deterministic Turing Machine*"). Le non déterminisme de la machine de Turing est borné par définition, dû que l'ensemble des actions possibles est borné par la taille du programme qui est fini.

Il existe plusieurs types restreints de bandes qui ont obtenues des noms spéciaux, ce qui nous donne à chaque fois une variante différente de machine de Turing, cependant les différentes variantes conduisent toutes à la même notion de calculabilité [Car06].

- Ø ***Pile*** (*stack*) : Une pile est une bande semi-infinie avec la propriété que, à chaque fois que la tête fait un déplacement à droite, le contenu précédent de la cellule est effacé (dépilement de la pile). Après une instruction d'écriture, la tête peut se déplacer à gauche (empilement d'un symbole dans la pile). La tête peut se positionner directement sur le début et dans ce cas la pile devient vide (*the bottom*).
- Ø ***Queue*** (file d'attente) (*queue*) : Une file d'attente est une bande semi-infinie avec deux têtes de mouvement uniquement à droite, la première tête est seulement à écriture, tandis que la deuxième est à lecture seule, la deuxième tête peut lire tous ce que la première tête a écrit auparavant, de plus une queue peut être testée si elle est vide ou non.
- Ø ***Compteur*** (*counter*) : Un compteur est une pile avec un alphabet d'une seule lettre, son but est de compter un nombre *n* par enregistrement de *n* copies du symbole de l'alphabet de la bande, l'empilement et le dépilement correspondent respectivement à incrémenter et à décrémenter l'entier représenté sur la bande.

Pour les machines de Turing multi-bandes, il y a plusieurs choix possibles pour l'ensemble de mouvements des têtes. Nous décrivons ici une variante importante pour le modèle de

machine de Turing qui serait utile dans le troisième chapitre de ce mémoire : c'est "la machine de Turing avec Oracle".

- ***OTM* (*Oracle Turing Machine*)** : Une machine de Turing avec Oracle *OTM* est une machine de Turing à plusieurs rubans, où l'on peut faire appel à une fonction arbitraire calculée par un élément appelé "oracle" [LR93]. Une *OTM* a un ruban spécial, dit *ruban oracle*, et des états spéciaux : *query*, *yes*, et *no*, qui peut poser des questions sur l'adhésion à certain ensemble. Le comportement d'une *OTM M*, avec une bande oracle pour un langage *A*, est le même que celui d'une machine de Turing ordinaire, sauf que lorsque *M* entre à l'état *query* avec un mot *w* dans la bande oracle; *M* entre, en un seul pas, soit à l'état *yes* si $w \in A$, soit à l'état *no* si $w \notin A$. De plus, durant ce pas, la bande oracle est effacée, ainsi que le temps de traitement de chaque question est compté séparément. Telle machine est notée dans certaines références par M^A.

2.2.2. Machine à registre (*Register machine*)

La Machine de Turing a été inventée pour travailler sur la calculabilité, elle s'agit d'une machine théorique qui ne correspond pas exactement par son fonctionnement à un ordinateur de *Von Neumann*. Les machines à registre sont devenues le modèle de machine standard pour l'analyse des algorithmes concrets, d'une part les machines à registre peuvent être reconnues comme le modèle des ordinateurs réels réduits à leur répertoire d'instructions minimales essentielles, d'autre part les machines à registres n'ont pas certaines caractéristiques essentielles de tous les ordinateurs réels : elles n'ont ni une longueur de mots fini, ni un espace d'adressage fini.

2.2.2.1. Description du modèle de machine à registre

Les machines à accès aléatoires *RAM* (*Random Access Machine*) sont des systèmes de calcul beaucoup plus proches du fonctionnement réel des ordinateurs. Ce modèle de base est constitué d'un contrôle fini où un programme (formé d'une suite d'instructions) est enregistré, un ou plusieurs registres accumulateurs, un compteur d'instructions et une collection finie de registres mémoires $R[0]$, $R[1]$, …$R[n]$. Les registres et l'accumulateur contiennent les données manipulées par la machine, ils ont une longueur de mot non bornée, mais à toute configuration accessible seulement une valeur finie sera enregistrée dans chaque registre.

Les instructions des machines *RAM* se divisent en quatre catégories :

1- Instructions qui influent sur le flux de contrôle : elles incluent les branchements conditionnels et inconditionnels.

2- Instructions d'entrées/sorties.

3- Instructions de transfert des données entre l'accumulateur et la mémoire (les registres).

4- Instructions pour exécuter des calculs arithmétiques.

Le programme de la machine *RAM* est une séquence étiquetée d'instructions du répertoire d'instructions. Dans toutes les instructions qui ne sont pas des branchements le pointeur d'instructions sera simplement mis à l'instruction suivante de l'instruction qui est déjà complétée dans le programme. Le non déterminisme dans le programme du modèle *RAM*

est obtenu si chaque ligne du programme contient deux instructions, de lesquelles la machine doit choisir une à chaque exécution de cette ligne, ou dans l'utilisation multiple des étiquettes : c'est-à-dire lorsque la machine exécute un branchement à une étiquette qui se trouve deux fois dans le programme, alors une des occurrences de l'étiquette sera choisie comme l'adresse à laquelle l'exécution du programme se continue [Emd90].

Dépendamment du modèle précis choisi, le transfert des données est effectué soit directement entre les registres, soit en utilisant le registre accumulateur comme intermédiaire. De plus, le modèle *RAM* se caractérise par l'utilisation de l'adressage indirect. Ainsi, il y a trois types d'instructions de chargement et deux types d'instructions de rangement, définis comme suit :

LOADDi	(*Acc*:=*i*),		
LOADi	(*Acc*:=*R*[*i*]),	*STOREi*	(*R*[*i*]:=*Acc*),
LOADIi	(*Acc*:=*R*[*R*[*i*]]),	*STOREi*	(*R*[*R*[*i*]]:=*Acc*).

Pour les machines *RAM*, on peut obtenir plusieurs modèles, selon des restrictions sur les instructions :

- Le modèle qui a uniquement l'instruction de successeur (instruction d'incrémentation de l'accumulateur par un : *Acc* :=*Acc+1*), ce modèle est appelé *Successor RAM*, et est noté par *SRAM*.
- Le modèle standard avec les deux instructions d'addition et de soustraction, noté par *RAM*.
- Le modèle qui a des instructions d'addition, soustraction, multiplication et division, ce modèle est plus puissant, et est noté par *MRAM*.
- Si les opérations logiques sont disponibles, on ajoute *B* dans le nom du modèle (par exemple *MBRAM*).
- Si le modèle est non déterministe, alors on ajoute *N* dans le nom du modèle.

Pour chacun des différents modèles, il y a au moins deux méthodes différentes pour mesurer le temps et l'espace mémoire consommés par une computation : La mesure "uniforme" dans laquelle chaque instruction exécutée est comptée comme un seul pas, sans se soucier de la valeur sur laquelle se fait l'opération, et la mesure "logarithmique" dans laquelle chaque instruction exécutée a un coût égal à la somme des logarithmes de toutes les quantités d'informations impliquées dans l'instruction, même les calculs d'adresses dans l'adressage direct ou indirect.

L'idée de *Von Neumann* d'enregistrer un programme dans la mémoire (au lieu du contrôle fini) nous amène au modèle *RASP* (*Random Access Stored Programme machine*), ici la mémoire est divisée en registres, il est apparu que pour la théorie de la complexité les modèles *RAM* et *RASP* sont totalement équivalents : ils simulent l'un l'autre en temps réel avec un coût de facteur constant en espace [Emd90]. Les machines *RAM* sont aussi équivalentes aux machines de Turing, ceci signifie que toute fonction calculée par une machine *RAM* peut aussi l'être par une machine de Turing et qu'inversement toute fonction calculée par une machine de Turing peut aussi l'être par une machine *RAM*.

2.2.2.2. Mesures du temps pour les RAMs

Nous avons défini précédemment deux manières de mesurer le temps pour une computation : la mesure uniforme dans laquelle chaque instruction prend une unité de temps, et la mesure logarithmique dans laquelle chaque instruction est chargée de la somme des longueurs de toutes les données qu'elle manipule, pour cette dernière mesure on utilise la fonction *size* qui est définie comme suit [Emd90] :

$$Size\ (n) = \textbf{\textit{if}}\ n \leq 1\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ [\log n] + 1\ \textbf{\textit{fi}}$$

Par exemple, le coût de l'instruction *ADDI j* (c'est-à-dire : *Acc* := *Acc* + *R*[*R*[*j*]]) est égal à: *size*(*x*)+*size*(*j*)+*size*(*R*[*j*])+ *size*(*R*[*R*[*j*]]), où *x* dénote la valeur courante de l'accumulateur *Acc*.

Dans la suite, l'utilisation de la mesure uniforme ou logarithmique sera indiqué par préfixation des mots *TIME* et *SPACE* dans les notations des classes de complexité par *U* et *L* respectivement. Il est claire que le coût d'une computation est moins dans la mesure de temps uniforme que dans la mesure logarithmique.

L'utilisation des opérations logiques peut être simulé dans les modèles qui n'ont pas telles instructions en temps polynomial en la longueur des opérandes, comme il est indiqué par le théorème suivant [Emd90] :

Théorème 2.1 Les coûts de simulations reliées aux opérations logiques sont donnés par :

(1) *SBRAM-UTIME* ≤ *SRAM-UTIME* (*time n.log n*)

(2) *SBRAM-LTIME* ≤ *SRAM-LTIME* (*time n.log n*)

(3) *BRAM-UTIME* ≤ *RAM-UTIME* (*time Poly*)

(4) *BRAM-LTIME* ≤ *RAM-LTIME* (*time Poly*)

(5) *MBRAM-UTIME* ≤ *MRAM-UTIME* (*time Poly*)

(6) *MBRAM-LTIME* ≤ *MRAM-LTIME* (*time Poly*).

Ces résultats se maintiennent aussi pour les modèles non déterministes.

Les frais de simulations du modèle de machine de Turing et les différents modèles *RAM* sont exprimés par les deux théorèmes suivants [Emd90]:

Théorème 2.2 Les différents modèles *RAM* peuvent simuler le modèle de machine de Turing en temps polynomial comme suit :

(1) *T* ≤ *SRAM-UTIME* (*time real-time*),

(2) *T* ≤ *SRAM-LTIME* (*time k . n . log n*),

(3) *T* ≤ *RAM-UTIME* (*time k . n / log n*),

(4) *T* ≤ *RAM-LTIME* (*time n . log log n*).

Théorème 2.3 Le modèle de machine de Turing peut simuler les différents modèles *RAM* avec les frais suivants :

(1) *SRAM-UTIME* ≤ *T* (*time* n^2*. log n*),

(2) *RAM-UTIME* ≤ *T* (*time* n^3),

(3) *SRAM-LTIME* $\leq T$ (*time* n^2),

(4) *RAM-LTIME* $\leq T$ (*time* n^2),

(5) *MRAM-LTIME* $\leq T$ (*time Poly*).

Ces résultats montrent qu'un problème décidable sur une machine de Turing en temps polynomial est aussi décidable en temps polynomial sur une machine *RAM* [RT03].

Si des bandes multidimensionnelles sont utilisées, les frais sont réduits à l'effet que le facteur n^2 dans le coût de la simulation pour une bande linéaire peut être remplacé par $n^{1+1/d}$, où *d* dénote la dimension de la bande utilisée [Emd90].

2.2.2.3. Mesures de l'espace mémoire pour les RAMs

Dans la théorie de modèles de machine, tout registre de *RAM* est chargé d'un coût correspondant à la taille de ses contenus, mais il y a plusieurs manières de procéder : une mesure brute consiste à charger chaque registre utilisé avec la taille de la plus grande valeur produite durant la computation entière. Une méthode plus raffinée consiste à charger chaque registre avec la plus grande valeur enregistrée dans ce registre durant la computation, celle-ci peut être exprimée par :

$$space = \sum_{j=0}^{maxaddr} size(j, max(j)) \qquad (2.3)$$

Où *maxaddr* est l'index de l'adresse la plus élevée accédée durant la computation, et *max*(*j*) est la valeur la plus grande enregistrée dans *R*[*j*] durant la computation, le paramètre *j* est ajouté dans la fonction *size* pour permettre de considérer des fonctions de taille qui dépendent de l'adresse de registre.

Une méthode plus raffinée consiste à considérer les configurations individuelles du *RAM* : charger chaque registre dans chaque configuration avec la taille de la valeur courante enregistrée là, et calculer la somme de ces coûts de registres qui sont en cours d'utilisation, la valeur maximale de ces sommes dans toutes les configurations accessibles devient la mesure de l'espace. Il y a d'autre méthode pour définir la mesure de l'espace pour le modèle *RAM* [Emd90].

2.2.3. Le modèle de circuit (*Circuit model*)

Un troisième modèle de machine séquentielle est le modèle de circuit, ce modèle est simple à décrire et facile à analyser mathématiquement. Les circuits sont technologie de base, consistant en des portes logiques très simples connectées par des fils de transfert des bits, ils n'ont aucune mémoire et aucune notion d'état. Les circuits évitent presque tous les problèmes d'organisation de machine et de répertoire d'instructions. Leurs composants de calcul correspondent directement aux dispositifs qu'on peut fabriquer actuellement.

2.2.3.1. Description du modèle de circuit

Un circuit est simplement un modèle formel de circuit logique combinatoire. Un "circuit booléen" (ou un "réseau logique") est un graphe fini, étiqueté, orienté et acyclique. Les nœuds d'entrée sont les nœuds sans ancêtre dans le graphe, et les nœuds de sortie sont les nœuds sans successeurs. Les nœuds d'entrée sont étiquetés avec les noms des variables d'entrée x_1, x_2,…x_n. Les nœuds internes dans le graphe sont étiquetés avec des fonctions d'une collection finie appelée "la base" du circuit, ici il est imposé que le nombre d'entrées

d'un nœud interne est égal au nombre d'arguments de son étiquette, de plus, un étiquetage convenable des arrêtes établit une correspondance 1:1 entre les ancêtres et les arguments de l'étiquette.

Par définition, on définit pour chaque nœud dans le circuit une fonction représentée par ce nœud : pour une variable d'entrée étiquetée par x_j ceci est la fonction de projection $(x_1, x_2, \ldots x_n) \rightarrow x_j$. Un nœud interne avec l'étiquette $g(y_1, \ldots y_k)$, pour lequel les k ancêtres représentent les fonctions $h_1(x_1, \ldots x_n)$, … $h_k(x_1, \ldots x_n)$, représente la fonction $g(h_1(x_1, \ldots x_n) \ldots h_k(x_1, \ldots x_n))$. Une fonction $f(x_1, \ldots x_n)$ est calculée par le circuit si elle est représentée par un nœud de sortie. La figure suivante représente un exemple de circuit booléen, avec les nœuds d'entrée sont : x_1, x_2 et x_3, et les nœuds de sortie sont : $y_1=(x_1 \Rightarrow x_2) \wedge x_2$, $y_2 = \neg(x_2 \vee x_2)$ et $y_3= x_3$.

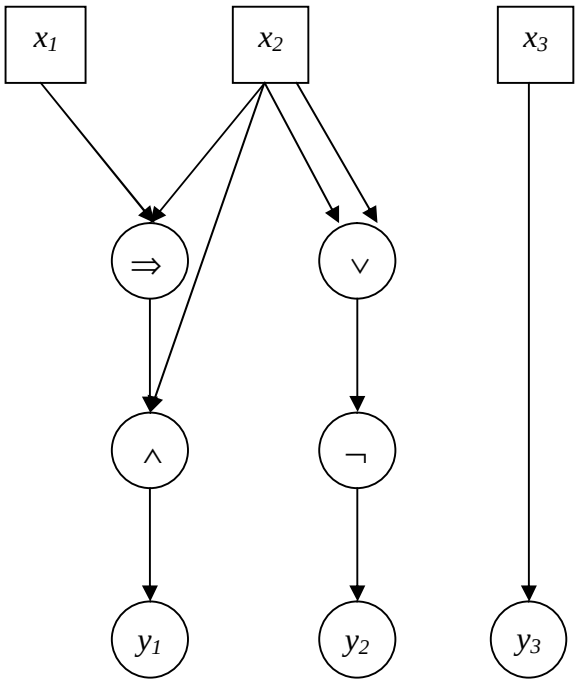


Figure 2. 3 : Exemple de circuit booléen

Les fonctions de mesure de la taille pour le modèle de circuit sont [Emd90] :

- *Depth* (profondeur) : c'est la longueur du chemin le plus long dans le graphe à partir d'un nœud d'entrée jusqu'à un nœud de sortie (Si un nœud x est un nœud d'entrée, sa profondeur est zéro. Autrement, la profondeur du nœud est déterminée par celle du nœud prédécesseur à laquelle on ajoute un).
- *Size* (taille) : c'est le nombre total de nœuds dans le graphe.

En général, nous nous restreignons aux circuits sur les bases des fonctions logiques monadiques (de lesquelles seulement la *not* est importante) et binaires (telles que *and*, *or*, *xor*…etc.). Il est bien connu qu'il existe des fonctions binaires simples, comme la fonction *nand*, qui forment une base complète par elles-mêmes. La base de circuit la plus commune

consiste en trois fonctions: *and*, *or* et *not*. Si la *not* est omise alors le circuit sera appelé "circuit monotone". Les réseaux avec des portes *and* et *or* d'entrance (*fan-in*) non borné sont aussi considérés. Si le nombre de successeurs de chaque nœud est limité à 1, alors le circuit devient un arbre et on parle de "formule" au lieu de "circuit".

Les mesures de la complexité introduites au-dessous dépendent de la base logique, et si on s'occupe de circuits ou de formules [Emd90]. Dans ce dernier cas, on parle de "la complexité de formules" (*formula complexity*) au lieu de "la complexité de circuit" (*circuit complexity*). Il n'est pas difficile de voire que, pour la complexité de circuit, l'effet du choix de la base est au plus d'un facteur constant en *depth* et en *size*.

Définition 2.10

- Pour une collection finie de fonctions logiques $f_1(x_1, \ldots x_n) \ldots f_k(x_1, \ldots x_n)$, les complexités de circuit $size(f_1,\ldots f_k)$ et $depth(f_1,\ldots. f_k)$ sont la taille et la profondeur minimale d'un circuit, tel que l'ensemble entier de fonctions est calculé par ce circuit.
- Si F_n, pour $n \in \aleph^+$, est une famille de fonctions de n-arguments, on définit $size(F_n)$ et $depth(F_n)$ comme une fonction de n.

2.2.3.2. Relation avec la complexité pour la machine de Turing

Le contrôle fini d'une machine de Turing peut être considéré comme un automate fini qui peut être implémenté en utilisant un circuit booléen, la taille de ce circuit nous dit quelque chose sur la complexité du programme de la machine, tandis que la profondeur (*depth*) indique avec quelle vitesse la machine peut exécuter un seul pas (phase) de sa computation.

Le théorème suivant [Emd90] définit la relation entre la machine de Turing et la taille du circuit qui la simule :

Théorème 2.4 $size \leq P.\ \mathrm{T}(n)\ .\mathrm{S}(n)$

Où T(n) et S(n) dénote la complexité en temps et en espace de la machine de Turing et P dénote sa taille.

Dans la simulation en dessus, une grosse quantité de circuiterie est gaspillée pour les calculs exigés pour copier le contenu des cellules non visitées de la bande d'une configuration à la suivante.

2.3. Deuxième classe de machine

Dans cette section nous explorons quelques modèles de la deuxième classe de machine, cette dernière, conformément à la définition 2.7, se compose de modèles de machine qui satisfont la thèse suivante [Emd90], qui définit la relation entre l'espace mémoire dans les modèles séquentiels et le temps dans les modèles parallèles.

***Thèse 2.1* (*parallel computation thesis*) :** Tout ce que peut être résolu en espace borné polynomialement sur un modèle de machine séquentielle raisonnable, peut être résolu en temps borné polynomialement sur une machine parallèle raisonnable, et vice versa.

Il y a une grande variété de modèles obéissant à cette thèse, cependant ils n'utilisent pas tous la notion de parallélisme intrinsèque. En fait, certains modèles de la deuxième classe

de machine, comme le modèle de machine de Turing alternante, obéissent à la thèse du calcul parallèle sans qu'ils soient des modèles parallèles. Par contre, certains modèles de machine parallèle ne font pas partie ni de la première ni de la deuxième classe de machine [Emd90] : ils sont plus faibles ou plus puissants que les membres de ces deux classes de machine.

2.3.1. Le modèle d'alternance

La notion du non-déterminisme utilise le mode "existentiel" dans la computation : si n'importe quel choix conduit à un état acceptant, alors la computation entière est acceptante, d'autres définitions utilisent le mode "universel" : si tous les choix conduisent à un état acceptant, alors la computation entière est acceptante. Nous allons voir ici un modèle qui alterne entre ces deux modes, c'est le modèle de la machine de Turing alternante (*alternating Turing machine* : *ATM*).

2.3.1.1. Concept de l'alternance

Le concept de l'alternance (*alternation*) conduit aux modèles de machine qui obéissent à la thèse du calcul parallèle sans aucun parallélisme intrinsèque [Emd90]. Comme un dispositif de calcul, une machine de Turing alternante est très similaire à la machine de Turing séquentielle standard, seulement la définition d'acceptation pour les entrées est différente. Simplement, une machine de Turing alternante est une machine de Turing non-déterministe dont les états sont divisés en deux ensembles : états "existentiels" et états "universels". Un état existentiel est acceptant si certaines transitions mènent à un état acceptant, un état universel est acceptant si toute transition mène à un état acceptant, la machine entière est acceptante si l'état initial est acceptant.

Depuis que la machine est non déterministe, la computation peut être représentée comme un arbre de computation dans lequel les branches représentent les transitions possibles. Les feuilles (les configurations où la machine s'arrête) sont acceptantes ou rejetantes. Pour une machine non déterministe standard, un arbre de computation représente une computation acceptante aussitôt qu'une seule feuille acceptante peut être trouvée, mais pour la machine alternante la notion d'acceptation est un peu plus compliquée.

L'idée principale est d'équiper les états dans le programme de la machine de Turing avec étiquettes "existentielles" et "universelles", une configuration hérite l'étiquette de son état. Puis assigner une qualité *Accept*, *Reject* ou *Undef* à chaque nœud dans l'arbre de computation selon les règles suivantes :

1- la qualité d'une feuille acceptante (resp. rejetante) égale *Accept* (resp. *Reject*),

2- la qualité d'un nœud interne représentant une configuration existentielle est *Accept* si l'une de ses configurations successeurs a la qualité *Accept*,

3- la qualité d'un nœud interne représentant une configuration existentielle est *Reject* si toutes ses configurations successeurs ont la qualité *Reject*,

4- la qualité d'un nœud interne représentant une configuration universelle est *Reject* si l'une de ses configurations successeurs a la qualité *Reject*,

5- la qualité d'un nœud interne représentant une configuration universelle est *Accept* si toutes ses configurations successeurs ont la qualité *Accept*,

6- La qualité de tout nœud qui n'est pas déterminée par application des règles précédentes est *Undef*.

La qualité *Undef* survient seulement si l'arbre de computation contient des branches infinies. Par définition, une machine alternante accepte son entrée si le nœud racine de l'arbre de computation, représentant la configuration initiale sur l'entrée, obtient la qualité *Accept*.

Le temps (resp. espace) consommé par une computation alternante est mesuré comme la consommation du temps (resp. espace) maximale le long de toute branche dans l'arbre de computation. Une autre mesure de la complexité pour la machine de Turing alternante est la taille minimale d'un arbre de computation acceptant [Emd90]. Un arbre acceptant est un sous-arbre de l'arbre de computation complet qui donne un certificat que l'entrée est acceptée, il inclut tous les successeurs d'un nœud universel. Mais, pour un nœud existentiel, uniquement les successeurs avec la qualité *Accept* doivent être inclus dans l'arbre de computation acceptante.

2.3.1.2. Relation entre machine de Turing alternante et modèles séquentiels

La machine alternante, qui est justement une machine standard en déguisement, hérite les résultats de simulation pour les modèles de la première classe de machine. Par conséquent il existe une hiérarchie indépendante de la machine (*machine-independent*) pour les classes alternantes :

$$\text{ALOGSPACE} \subseteq \text{APTIME} \subseteq \text{APSPACE} \subseteq \text{AEXPTIME}$$

Le comportement d'une machine alternante comme un modèle de machine parallèle est exprimé par le théorème suivant [Emd90] :

Théorème 2.5 Le modèle de machine de Turing alternante est relié à la hiérarchie de complexité séquentielle par l'égalité APTIME = PSPACE, les autres classes de complexité sont aussi des versions décalées dans la hiérarchie séquentielle :

$$\text{ALOGSPACE} = \text{P}, \text{APSPACE} = \text{EXPTIME}, \text{AEXPTIME} = \text{EXPSPACE}.$$

2.3.2. Le modèle EDITRAM

Le modèle de machine *EDITRAM*, proposé comme un modèle d'un éditeur de texte, est obtenu à partir du modèle *RAM* par adoption d'une nouvelle direction : en mettant l'accent sur les instructions de manipulation de symboles, et en baissant la puissance arithmétique.

Dans le modèle *EDITRAM*, nous étendons le modèle *RAM* avec un ensemble fini fixe de fichiers textes. Des registres standards peuvent être utilisés comme curseurs dans les fichiers textes. À côté des instructions standards sur les registres arithmétiques, le modèle *EDITRAM* a des instructions pour :

(1) la lecture d'un symbole à partir d'un fichier via un curseur,

(2) l'écriture d'un symbole dans un fichier via un curseur,

(3) le positionnement d'un curseur à la fin d'un fichier (et donc calcul de sa longueur),

(4) le positionnement d'un curseur dans un fichier par chargement d'une valeur arithmétique dans le curseur,

(5) le remplacement systématique de *string1* par *string2* dans un fichier texte,

(6) la concaténation des fichiers textes,

(7) la copie de segments des fichiers textes indiqués par positions des curseurs,

(8) la suppression de segments des fichiers textes indiqués par positions des curseurs,

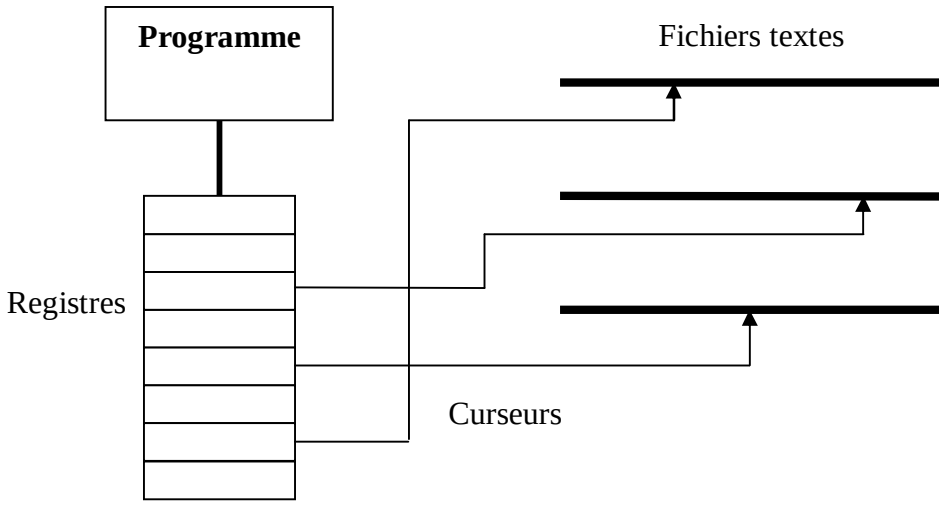


Figure 2. 4 : Machine *EDITRAM*

La complexité en temps pour le modèle *EDITRAM* est définie en utilisant la mesure de temps uniforme pour les registres arithmétiques, dans ce cas toute instruction d'édition est chargée par une unité de temps. Une alternative raisonnable est d'utiliser la mesure de temps logarithmique par rapport aux registres arithmétiques, alors le coût d'une instruction d'édition est le logarithme de la valeur de curseur impliqué, qui peut être proportionnelle au logarithme de la longueur de la portion affectée dans le fichier texte.

Le modèle *EDITRAM* est un modèle de la deuxième classe de machine, il obéit à la thèse du calcul parallèle, avec : EDITRAM-NPTIME $\subseteq$ PSPACE et PSPACE $\subseteq$ EDITRAM-PTIME.

2.3.3. Les machines de vrai parallélisme

Dans cette section nous considérons les modèles qui fournissent un parallélisme observable par possession des processeurs multiples qui opèrent sur données partagées et/ou canaux partagés. Certains modèles, qui appartiennent à la deuxième classe de machine, deviennent plus puissants lorsque des caractéristiques secondaires de leur définition sont considérées.

Dans ces modèles, une computation peut procéder soit synchroniquement, où tous les processeurs exécutent une étape de la computation en même temps selon une horloge locale, soit asynchroniquement, où chaque processeur calcule selon sa vitesse.

Il existe plusieurs stratégies pour résoudre les conflits en écriture qui survient lorsque plusieurs processeurs essayent d'écrire dans la même location de la mémoire partagée:

exclusive-write, *common-write*, *arbitrary-write* et *priority-write strategy*, cette dernière a établi elle-même comme la plus puissante stratégie.

Il existe plusieurs méthodes pour contrôler la création des processeurs parallèles. Certains modèles ont une grande collection des processeurs identiques qui opèrent en parallèle. Dans d'autres modèles, les processeurs peuvent créer, par leurs propres actions, un nombre fini de nouveaux processeurs qui fonctionnent en parallèle.

2.3.3.1 Le modèle SIMDAG

Dans le modèle *SIMDAG* (*Single Instruction, Multiple Data AGregate*), il existe un seul processeur global qui peut diffuser les instructions à une séquence potentiellement infinie de processeurs locaux, d'une manière que seulement un nombre fini de processeurs soit activé. Le mécanisme pour maintenir le nombre de processeurs activés (en un seul pas) fini, consiste à utiliser la "signature" des processeurs locaux : Chaque processeur local a un registre à lecture seule, appelé signature, contenant un nombre qui identifie uniquement ce processeur local. Dans la diffusion d'une instruction, le processeur global inclut une valeur de seuil, et tout processeur local avec signature inférieure que la valeur de seuil transmise exécute l'instruction, tandis que les autres restent inactifs.

Les processeurs locaux opèrent sur des mémoires locales et sur la mémoire partagée du processeur global, où les conflits en écriture sont résolus par priorité : le processeur local ayant le plus bas index devient le vainqueur en cas de conflit en écriture.

Dans le modèle *SIMDAG* le répertoire d'instructions pour le processeur global et les processeurs locaux contient des opérations arithmétiques additives et logiques parallèles, combinées avec l'opération de décalage restreint (*restricted shifting*) (division par 2). Pour un dispositif comme *SIMDAG*, il est nécessaire de restreindre la puissance des instructions arithmétiques considérées, sinon la machine peut devenir plus puissante [Emd90] que les membres de la deuxième classe de machine.

Théorème 2.6 Le modèle *SIMDAG* est un membre de la deuxième classe de machine.

2.3.3.2. La machine de traitement vectoriel

La machine de traitement vectoriel (*APM* : *Array Processing Machine*) est inspirée par les superordinateurs vectoriels contemporains, ces machines ont la structure de mémoire du modèle *RAM* ordinaire, mais elles ont un accumulateur vectoriel qui se compose d'un tableau linéaire, potentiellement non borné, des accumulateurs standards.

La machine de traitement vectoriel combine l'ensemble des instructions du *RAM* standard avec un nouveau répertoire d'instructions vectorielles qui opèrent sur l'accumulateur vectoriel. Ces instructions permettent la lecture, l'écriture, le transfert des données et les opérations arithmétiques sur vecteurs de taille assortie.

Un contrôle conditionnel sur une opération vectorielle est possible par l'utilisation de masque qui consiste en un tableau de valeurs booléennes de la même taille que les opérandes vectorielles, l'instruction vectorielle sera exécutée uniquement sur les locations correspondants aux occurrences de 1 dans le masque. Donc une adresse complète pour une opération vectorielle peut consister de quatre entiers : les limites inférieures et supérieures de l'argument vectoriel et du masque respectivement.

La puissance de parallélisme du modèle est pourvue par la mesure de temps utilisée : temps uniforme ou temps logarithmique (où toute instruction vectorielle est chargée selon son composant scalaire le plus coûteux). Ainsi dans une opération de chargement de vecteur (*vector-LOAD*) la complexité en temps logarithmique est proportionnelle au logarithme de la limite supérieure de l'opérande et/ou masque plus le logarithme de la plus grande valeur chargée dans l'accumulateur vectoriel.

La machine de traitement vectoriel obéit à la thèse du calcul parallèle, elle est un membre de la deuxième classe de machine, avec : PSPACE $\subseteq$ *APM*-PTIME et *APM*-NPTIME $\subseteq$ PSPACE [Emd90].

2.3.3.3. Les modèles avec parallélisme récursif

Comme un autre type de machines parallèles, nous avons la machine de Turing récursive. Dans ce modèle, toute copie du dispositif peut engendrer des nouvelles copies qui commencent à calculer dans leur propre environnement de bandes de travail.

Un modèle similaire, basé sur le modèle *RAM*, est le *k-PRAM*. Dans ce modèle un dispositif comme le *RAM* peut créer jusqu'à *k* copies de lui-même, qui commencent à calculer dans leur environnement local, pendant que leur créateur continue sa computation. Ces copies peuvent elles-mêmes créer aussi des nouveaux fils. Les données sont transmises du parent au fils au moment de génération, par chargement des paramètres dans les registres du fils. À la fin, un fils peut renvoyer un résultat à son parent en écrivant une valeur dans un registre spécial du parent qui est à lecture seule pour ce dernier. Le parent peut inspecter le registre pour voir si le fils a déjà l'assigné une valeur, s'il lit le registre avant que le fils a écrit une valeur, la computation du parent est suspendue. Utilisant ces caractéristiques, un parent peut activer un nombre de fils et consomme la première valeur qui est retournée, en abandonnant les computations des fils restants qui n'ont pas encore terminés.

Les modèles basés sur tels canaux de communication locale sont beaucoup plus lents en diffusion de l'information à une collection de sous-processeurs et en rassemblement des résultats. La validité de la thèse du calcul parallèle n'est pas perturbée par ce délai, dû que le ralentissement est -dans le pire des cas- polynomial [Emd90].

La différence entre le modèle *SIMDAG* et les modèles récursifs est que dans le modèle *SIMDAG* les processeurs locaux, s'ils sont tous activés à certain moment, exécutent la même instruction sur des données pouvant être différentes. Dans les modèles récursifs chaque processeur, une fois devient active, exécute son propre programme sauf pour l'impact de communication avec son parent ou ses fils.

2.4. Conclusion

La théorie de la complexité fournit une diversité de concepts et de résultats. Le but de ce chapitre est de présenter la base théorique pour l'étude de ce champ de l'informatique théorique. Ainsi, dans ce chapitre, nous avons présenté les principaux concepts et notions dans la théorie de la complexité, en suivant deux axes.

Le premier axe concerne les notions relatives à la complexité et les classes de complexité, avec présentation d'une relation très importante qui peut être établie entre les différents

problèmes, c'est la relation de réduction, celle-ci permet de comparer les complexités des problèmes et aide à leur résolution.

Le deuxième axe concerne les modèles de machine, sur lesquelles sont effectués les calculs, et par rapport auxquels sont définies les classes de complexité. Nous avons vu quelques modèles de la première et de la deuxième classe de machine, et les résultats majeurs sur leurs puissances de calcul, en utilisant la relation de simulation entre les modèles de machine, cette relation permet d'exprimer les frais nécessaires pour les simulations de ces modèles en termes de temps et d'espace mémoire.

Chapitre 3

Classes de complexité

Un des buts de la théorie de la complexité est de classifier les problèmes selon leurs difficultés de calcul intrinsèques. Jusqu'aux nos jours, on n'a pas une réponse précise pour la question sur les ressources et les puissances exigées pour résoudre un problème donné, mais on a fait une grande progression dans la classification des problèmes en classes de complexité, qui caractérisent quelque chose de leurs difficultés inhérentes. Dans ce chapitre nous étudierons les classes de complexité les plus populaires. La section 1 contient les définitions et les concepts préliminaires exigés pour définir les classes de complexité. Les sections 2, 3 et 4 peuvent être vues comme un catalogue des classes de complexité les plus populaires.

3.1. Préliminaires

3.1.1. Problèmes et instances

Énoncer un problème demande simplement la description de ses entrées, et du résultat attendu. Il s'agit donc de définir une fonction sur un ensemble de départ, les instances du problème, en associant à chaque instance le résultat attendu. Résoudre un problème, c'est être capable de fournir pour chaque instance le résultat correspondant.

Définition 3.1 Un problème est un ensemble X de paires ordonnées (I, A) de mots dans $\{0,1\}^*$, où I est appelé "instance" (*instance*), A est appelé "réponse" (*Answer*) pour l'instance, et tout mot dans $\{0, 1\}^*$ apparaît comme le premier composant d'au moins une paire.

Cette définition abstraite est exigée car les classes de complexité qu'on en discute sont toutes définies en termes de modèles de machines, qui sont équipés uniquement pour manipuler des entrées et des sorties qui sont des chaînes de caractères, plutôt que les objets combinatoires qui forment les sujets des problèmes, tels que graphes, équations ou expressions logiques,…etc. Cependant, ces objets combinatoires ne sont pas tous différents de chaînes de caractères, au moins dans la manière d'utilisation, par exemple un problème sur des graphes consiste en fait en une question sur une représentation symbolique d'objets combinatoires appelés graphes, plutôt que les graphes eux-mêmes [Joh90].

Typiquement, nous présentons ces représentations symboliques en une façon linéaire, c'est à dire par chaînes de caractères. Ainsi, nous devons avoir certaine manière pour représenter les instances et les réponses comme chaînes de caractères si on veut les représenter dans la mémoire de l'ordinateur, qui peut être elle même vue comme une chaîne de caractères. D'autre part, un objet combinatoire peut avoir plusieurs représentations (un graphe peut être représenté par une matrice d'incidences ou liste contiguë…), heureusement, ces représentations "raisonnables" sont interchangeables : étant donnée une forme de représentation, on peut rapidement la traduire à n'importe quelle autre forme.

Une relation de chaînes de caractères doit être "totale" pour être considérée comme un problème [Joh90], c'est à dire que la définition du problème exige qu'il soit complètement spécifié, même pour les entrées dépourvues de sens. Notez, de plus, que notre définition de problème n'exige pas que la relation soit une fonction, c'est-à-dire qu'il y a -peut être- plusieurs paires dans *X* qui ont le même premier composant.

La définition précédente du problème est suffisamment générale pour satisfaire d'autres besoins. Pour mettre l'accent sur cette généralité, les relations de chaînes de caractères sont parfois rapportées aux "problèmes de recherche" généraux, en opposition aux cas spéciaux importants des : fonctions, problèmes de décision et problèmes de calcul, définis ci-dessous :

Définition 3.2 Une fonction (*function*) est une relation de chaînes dans laquelle chaque mot $x \in \{0,1\}^*$ est le premier composant de précisément une paire.

Définition 3.3 Un problème de comptage (*counting problem*) est une fonction dans laquelle toutes les réponses sont des entiers non négatifs.

Définition 3.4 Un problème de décision (*decision problem*) est une fonction dans laquelle les réponses possibles sont "*Yes*" et "*No*".

Dans un problème de décision, une instance pour laquelle la réponse est "*Yes*" est appelée "instance positive", sinon elle est dite "négative" [RT03].

Les problèmes de décision représentent un cas important particulier, la plupart des classes de complexité sont restreintes à ce type de problèmes ou, plus précisément, aux langages qui y sont dérivés.

Définition 3.5 Un langage est tout sous ensemble de {*0,1*}*.

Il y a une correspondance naturelle entre langages et problèmes de décision, exprimée par les définitions suivantes, où les problèmes de décision sont modélisés par la notion du langage, en déterminant si un mot appartient à un langage ou non.

Définition 3.6 Si *L* est un langage, alors le problème de décision R_L correspondant à *L* est: $R_L=\{(x, Yes) : x \in L\} \cup \{(x, No) : x \ddot{I} L\}$.

Définition 3.7 Etant donné un problème de décision *R*, le langage *L*(*R*) correspondant à ce problème est : $L(R) = \{x \in \{0,1\}^* : (x,Yes) \in R\}$.

Notez qu'il y a une asymétrie entre "*Yes*" et "*No*" dans cette définition (c'est à dire que "*No*" n'apparaît pas tant que "*Yes*" apparaît). Interchanger "*Yes*" et "*No*" produit un langage différent et "complémentaire" défini comme suit :

Définition 3.8 Si *L* est un langage, alors son langage complémentaire est:

$$co\text{-}L = \{0,1\}^* - L.$$

Cette asymétrie dans la définition de $L(R)$ est pour une importance théorique considérable, les langages $L(R)$ et $co\text{-}L(R)$ n'appartiennent pas toujours à la même classe de complexité [Joh90]. En fait, une question est posée sur le fait qu'une classe *C* est "fermée sous complément", c'est à dire : pour tout langage $L \in C$, est-ce qu'on peut dire que $co\text{-}L \in C$?

3.1.2. Comment résoudre un problème

Dans la théorie de la complexité, un problème *X* n'est pas considéré résolu à moins qu'il y a une méthode (algorithme) générale qui fonctionne pour n'importe quelle instance de ce problème (supposant qu'assez de temps, mémoire et autres ressources sont assurées). En pratique, étant données des limites physiques de ressources disponibles, telles méthodes peuvent être utilisables pour seulement un petit ensemble fini d'instances. La question-clé qui se pose est : comment les exigences en ressources s'augmentent avec la taille de l'instance ?

Définition 3.9 La taille d'une instance *I*, noté $|I|$, est sa longueur, c'est à dire c'est le nombre de symboles qu'elle contient.

Définition 3.10 Si *M* est une méthode pour résoudre un problème *X*, et *R* est une ressource utilisée par cette méthode, Alors $R_M : \Re^+ \rightarrow \Re^+$ est la fonction définie par location de $R_M(n)$ comme le maximal, sur tous les mots *x* de longueur *n*, de la quantité de ressource *R* utilisée lorsque *M* est appliquée à l'entrée *x*.

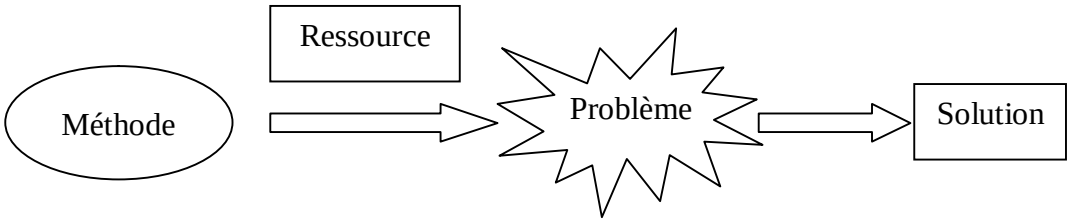


Figure 3. 1 : Résolution d'un problème

Dans cette définition, R_M est la mesure dans le pire des cas de la quantité de ressource *R* utilisée. Pour une instance particulière *x*, avec $|x|=n$, *M* peut utiliser moins que $R_M(n)$ de ressource *R*, cependant, $R_M(n)$ fournit la meilleure garantie générale la plus possible, et elle fournit ainsi une certitude qu'une mesure de moyen cas ne peut pas la donner [Joh90].

Dans le but de définir les classes de complexité, on parle des besoins en ressources pour les problèmes eux-mêmes, plutôt que juste les besoins pour une méthode particulière utilisée pour les résoudre. Cependant, il est difficile de définir directement cette notion de "besoin en ressource", nous nous contenterons d'une définition indirecte en termes de bornes supérieures et inférieures. Supposant que *X* est un problème, *R* est une ressource et *C* est une classe de méthodes.

Définition 3.11 Les besoins en ressources R de problème X sous méthodes de C sont bornés de supérieur par T(n) si et seulement s'il y a une méthode $M \in C$ pour résoudre X en utilisant une quantité de ressource $R_M(n) = O(\mathrm{T}(n))$.

Définition 3.12 Les besoins en ressources R de problème X sous méthodes de C sont bornés de l'inférieur par T(n) si et seulement si toutes les méthodes $M \in C$ pour résoudre X ont $R_M(n) = \Omega(\mathrm{T}(n))$.

Dans ces deux définitions, on a utilisé le terme "classe de méthodes", le but de n'importe quelle méthode qui résout un problème X est, étant donnée une instance x, de produire une réponse a tel que $(x,a) \in X$. Cependant, la manière dont a est produite peut varier considérablement. Notre choix de "classe de méthodes" peut produire des notions différentes de besoins en ressource pour un problème. Ce choix, et les limites de ressources correspondantes utilisées, sont ceux qui définissent une classe de complexité [Joh90]. Les classes de méthodes sont plus facilement décrites en termes de modèles de calcul (modèles de machine). Dans la littérature, il y a une grande variété de modèles de machine, avec plusieurs variantes (modes de calculs) pour chaque modèle (déterministe, non déterministe, parallèle, probabiliste,… etc.). Ainsi, une classe de complexité pour un problème donné est définie par détermination d'une variante de modèle de machine et la limite de ressource (s) utilisée (s) par celle-ci pour la résolution du problème considéré.

3.1.3. Limites de ressources

Il y a une diversité de ressources avec plusieurs types de limites, nous nous restreignons à la liste des types suivants :

- Constant : il existe un constant k tel que : $R_M(n) \leq k$, pour tout $n \geqslant 0$.
- Logarithmique : $R_M(n) = O(\log n)$.
- Poly-logarithmique : il existe un constant k tel que : $R_M(n) = O(\log^k n)$.
- Linéaire : $R_M(n) = O(n)$.
- Polynomial : il existe un constant k tel que : $R_M(n) = O(n^k)$.
- Exponentiel : il existe un constant k tel que : $R_M(n) = O\left(2^{n^k}\right)$.

3.1.4. Un premier exemple **:** *la classe P* **(***et la classe FP***)**

Comme un premier exemple, considérons le plus célèbre –peut être– de toutes les classes de complexité: la classe P des problèmes de décision solubles (ou langages reconnaissables) par DTMs (algorithme déterministe) obéissant à une limite polynomiale (en la taille de l'instance) sur le temps d'exécution. Informellement, on utilise souvent "P" pour indiquer la classe de tous les problèmes de recherche solubles en temps polynomial par un algorithme déterministe, mais nous utiliserons la notion "FP" pour dénoter cette classe plus générale [Joh90]. Nous avons donc la définition suivante :

Définition 3.13 La classe P (resp. FP) est l'ensemble de tous les problèmes de décision (resp. problèmes de recherche) solubles par DTMs en temps polynomial.

Les classes P et FP se composent donc de problèmes solubles par des algorithmes de complexité polynomiale, de même ces algorithmes sont dits polynomiaux.

L'importance de P (et FP) réside dans le fait que la solubilité en temps polynomial est équivalente théoriquement à la notion informelle de "soluble efficacement", autrement dit les algorithmes non polynomiaux sont certainement inefficaces.

De plus, la notion de temps polynomial est "robuste", c'est-à-dire qu'elle est indépendante de la technologie. En fait, la définition du temps de calcul dépend du modèle de calcul (le nombre d'instructions effectuées par un processeur), mais le temps de calcul est défini comme le nombre de transitions effectuées par une machine de Turing avant de s'arrêter, et les preuves d'équivalence entre tous les modèles connus montrent que la complexité se transforme de manière polynomiale.

Une autre caractéristique pour la classe P (et FP) est qu'elle forme une classe "stable" : la composition de deux algorithmes polynomiaux reste polynomiale, et l'algorithme qui est construit polynomialement à partir des appels des procédures de complexité polynomiale reste aussi polynomial.

Enfin, depuis que les classes de complexité déterministe sont fermées sous complément [GHR95], si $X \in P$, alors le problème complément de X est aussi dans P [HC91], ce qui implique que co-P=P. Ainsi, différemment de la plupart des autres classes, on ne parle pas de la classe co-P.

Une grande variété de problèmes sont dans la classe P (et FP), nous prenons comme un exemple le problème suivant :

- *CHEMIN*
  - *Instance* : Un graphe orienté *G*, et deux sommets *s* et *t*.
  - *Réponse* : "*Yes*" s'il existe un chemin dans *G* reliant le sommet *s* au sommet *t*.

CHEMIN appartient à P, en d'autres termes, il existe un algorithme *chemin*, de temps polynomial, tel que :

$$CHEMIN\ (G, s, t) = \begin{cases} \text{"}Yes\text{" s'il existe, dans } G, \text{ un chemin } s \text{ à } t, \\ \text{"}No\text{" sinon.} \end{cases}$$

Dans ce cas, un algorithme naïf, qui examine successivement tous les chemins de source *s* jusqu'à trouver un d'extrémité *t*, n'est pas en général de complexité polynomiale, car le nombre de chemins dans un graphe est en général exponentiel en fonction de la taille du graphe : pendant la construction d'un chemin, nous avons, à chaque sommet, un choix entre les différents arcs issus de ce sommet.

Au lieu de parcourir tous les chemins, nous pouvons nous contenter de marquer les sommets *i* tels qu'il existe un arc $s \to i$, puis de marquer les sommets *j* que nous pouvons atteindre à partir de ceux déjà marqués, et ainsi de suite, jusqu'à marquer *t*, ou jusqu'à stabilisation de l'ensemble des sommets marqués. Cet algorithme est de complexité polynomiale : le nombre de sommets marqués augmente au moins de 1 entre deux étapes consécutives de l'algorithme (sinon celui-ci se termine), donc le nombre d'étapes est borné par le nombre *n* de sommets du graphe, et chaque étape (d'examiner les arcs issus des sommets déjà marqués) est clairement polynomiale.

3.1.5. Réduction et complétude

Pour montrer qu'un problème particulier est dans une classe de complexité donnée, nous avons besoin d'exposer une "méthode" pour résoudre le problème, qui satisfait les exigences (le modèle de machine et les limites de ressources) de la définition de cette classe. Les preuves de la non adhésion sont plus indirectes. Cependant, une technique clé est appliquée aux deux sortes de preuve : c'est la réduction.

Un truc de programmation qui peut être incorporé dans la plupart de nos modèles de calcul est la "sous-routine": Un algorithme pour résoudre un problème *X* peut procéder par résolution d'un problème *Y*, c'est à dire par l'utilisation d'un algorithme pour la résolution du problème *Y*. Si nous considérons le modèle de machine de Turing, le concept de sous-routine peut être formalisé en termes d'oracles (figure 3.2): Supposant par exemple que nous avons une *OTM* déterministe de temps polynomial qui résout un problème *X* sous la supposition que l'oracle résout un problème *Y*, dans ce cas, l'oracle agit juste comme une sous-routine pour résolution de *Y*. Tout simplement, on peut dire qu'un problème *X* se réduit à un problème *Y* si, étant donné un moyen efficace pour résoudre *Y*, alors on peut trouver un algorithme efficace pour résoudre *X*.

Définition 3.14 Si *X* et *Y* sont des problèmes, une réduction de Turing de *X* à *Y* ($X \leq_T Y$) est toute *OTM* qui résout *X* étant donné un oracle pour *Y*.

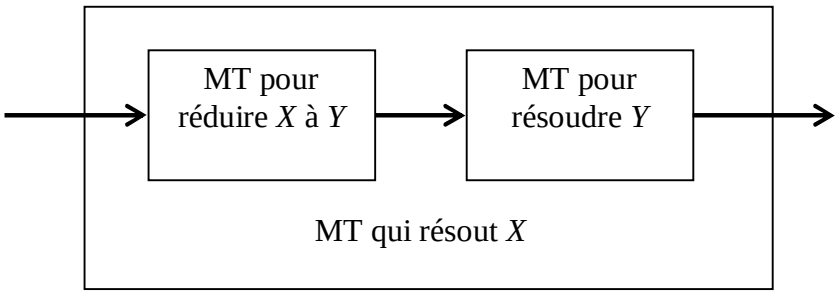


Figure 3. 2 : Réduction de Turing de *X* à *Y*

Par exemple, considérant le problème *TSDP* (*Traveling Salesman Decision Problem*) (défini dans la section ***3.2.1***) dans lequel on cherche s'il existe un tour avec longueur inférieure à une borne donnée *B*, et le problème *TSP* (*Traveling Salesman Problem*) dans lequel on cherche de trouver le plus court chemin, il est facile de construire une machine de Turing avec oracle *M* qui, étant donné un oracle pour le problème *TSDP*, résout le problème *TSP* : la machine *M* pose deux questions à l'oracle concernant respectivement des bornes *B* et *B-1*, notre machine *M* acceptera si et seulement si la première réponse de l'oracle est "*Yes*" et la deuxième réponse est "*No*", clairement cette machine de Turing avec oracle réduit le problème *TSP* au problème *TSDP* et fonctionne en temps polynomial, la réduction s'appelle ainsi : *Polynomial-Time Turing Reduction*.

Lorsque nous considérons seulement les problèmes de décision, la définition restreinte suivante suffit [Joh90].

Définition 3.15 Si *X* et *Y* sont des problèmes de décision, une "transformation" de *X* à *Y* est une fonction (calculable par *DTM*) $f : \{0,1\}^* \to \{0,1\}^*$, tel que *x* a une réponse "*Yes*" dans *X* si et seulement si $f(x)$ a une réponse "*Yes*" dans *Y*.

Notez que ceci est la même chose qu'une réduction de machine de Turing dans laquelle la *OTM* peut poser seulement une question à son oracle, et doit donner comme sa propre réponse la réponse qu'elle reçoit de l'oracle [Joh90], (c'est la réduction appelée "*many-one*" ($\leq_m$) vue dans le chapitre précédent). La relation suivante est donc vérifiée : si $X \leq_m Y$ alors $X \leq_T Y$ [GHR95], en d'autres termes, la réduction de Turing est une généralisation de la réduction "many-one", au lieu de faire une transformation sur l'entrée suivie par demande de la question unique "est-ce que $f(x) \in Y$", on permet des questions multiples, avec l'utilisation des nouvelles questions qui peuvent être des fonctions sur les réponses des questions précédentes. Ainsi, pour résoudre le problème *X* on fait plusieurs appels à une sous-routine pour la résolution du problème *Y*.

Les réductions deviennent un outil utile dans l'étude de la complexité lorsqu'elles-mêmes obéissent à des limites de ressources. Si *R* est une classe restreinte de réductions, on écrit " $X \leq_R Y$ " pour signifier qu'il y a une réduction dans *R* de *X* à *Y*.

Définition 3.16 Si *C* est une classe de complexité et *R* est une classe de réductions (limitées en ressources), on dit que *R* est compatible avec *C* si pour tous les problèmes *X* et *Y*, $X \leq_R Y$ et $Y \in C$ implique que $X \in C$.

Pour la classe P, la classe de réductions compatible la plus générale se compose de réductions de Turing qui obéissent à des limites de temps polynomial (*Polunomial-Time Turing reductions*) notées par "$\leq_T$ *reductions*". Deux autres classes de réductions compatibles avec P sont : la classe des transformations polynomiales (transformations *f* calculables en temps polynomial), notées par "$\leq_p$ *reductions*", et la classe des transformations calculables en espace logarithmique : $O(\log n)$, notées par "$\leq_{\log-space}$ *reductions*" [Joh90].

En plus de la compatibilité avec P, ces classes de réductions ont toutes une propriété plus importantes : c'est la transitivité.

Définition 3.17 Une classe de réductions *R* est transitive si pour tous les problèmes *X*, *Y* et *Z* : $X \leq_R Y$ et $Y \leq_R Z$ implique que $X \leq_R Z$.

La démonstration de l'adhésion (ou non) à une classe donnée *C* peut être effectuée à l'aide des deux théorèmes suivants:

Théorème 3.1 *$X \leq Y$ et $Y \in C$, implique que $X \in C$.*

Corollaire 3.1 *$X \leq Y$ et $X \notin C$, implique que $Y \notin C$.*

Par exemple, étant donné que $L_1 \leq L_2$, si $L_2 \in$ P alors $L_1 \in$ P, par contre si $L_1 \notin$ P alors $L_2 \notin$ P. les mêmes résultats sont vrais si on remplace P par les classes de complexité L, NL, NP, PSPACE, … [Bou03].

Dans la littérature, il y a d'autres formes de réductions plus complexes que celles présentées ici, telles que : "γ-*reduction*", "NC-*Turing reduction*", "NC^k-*Turing reduction*", "*Nondeterministic-reductions*" et des réductions de différents types… [Yap98].

Les réductions peuvent clairement simplifier le travail de prouver l'adhésion d'un problème à une classe de complexité. Plus important encore, elles peuvent être utilisées pour prouver la non-adhésion. Supposant que nous savons qu'un problème *X* n'est pas

dans une classe de complexité *C*, alors si nous pouvons exposer une réduction (compatible avec *C*) de *X* à *Y*, nous déduisons que *Y* ne peut pas être aussi dans *C* (par définition de "compatible"). La tâche difficile est de chercher premièrement un problème *X* qui n'est pas dans *C*, la réduction peut aussi aider ici.

Définition 3.18 Soit un problème *X*, *C* est une classe de complexité et *R* est une classe de réductions. Si $Y \leq_R X$ pour tout $Y \in C$, alors on dit que *X* est "difficile" (*hard*) pour *C* (sous *R-reductions*), ou simplement "*R-difficile*" (*R-hard*) pour *C*. Si aussi $X \in C$ alors on dit que *X* est "complet" pour *C* (sous *R-reductions*), ou "*R-complet*" (*R-complete*) pour *C*.

La relation entre ces types de problèmes est schématisée par la figure 3.3. Si *X* est *R-complet* pour une classe *C*, alors il peut être considéré comme problème typique des problèmes les plus difficiles dans *C*. Mais comment montrer qu'un problème *X* est difficile pour une classe sous un type de réduction donné? Étant donné que les classes en intérêt sont toutes de taille infinie, nous ne pouvons pas clairement exposer des réductions distinctes de chaque membre dans la classe à *X*. Cependant, pour beaucoup de nos classes de complexité, il y a un problème "générique" qui aide à la démonstration de la complétude d'autres problèmes.

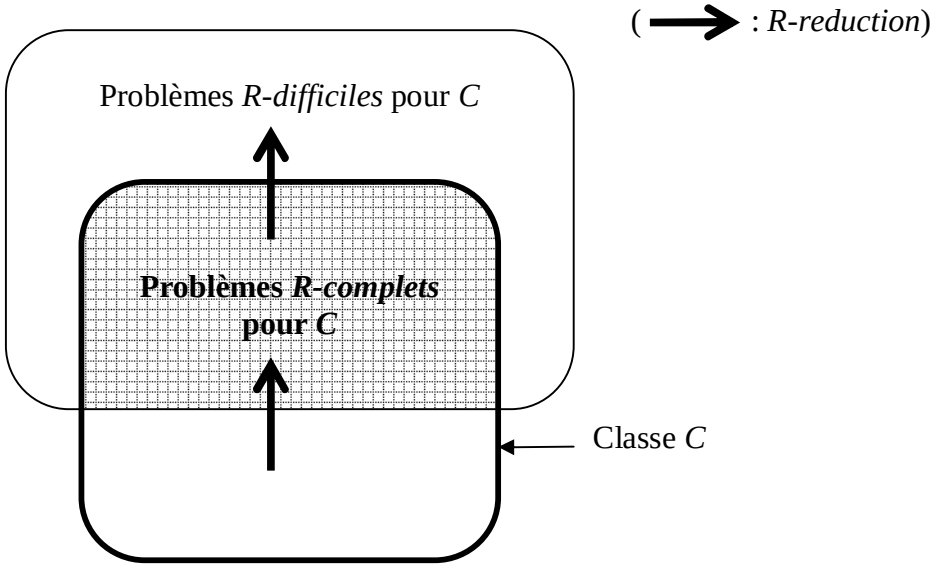


Figure 3. 3 : Notion de complétude

Une fois nous avons notre premier problème *R-complet Y* pour une classe *C*, l'obtention d'autres problèmes *R-complets* est conceptuellement beaucoup plus simple et directe : étant donné que notre classe de réduction est transitive, tout ce que nous avons besoin de faire pour montrer qu'un problème *X* est *R-complet* est de prouver que $Y \leq_R X$, la transitivité fait le reste.

3.2. Problèmes vraisemblablement intraitables

3.2.1. Les classes NP et NP-complets

La classe NP possède une définition moins naturelle que celle de P, et son nom est trompeur : il ne signifie pas "non polynomial", mais "polynomial non-déterministe". La

classe NP est donc une extension de la classe P, en autorisant des choix non déterministes pendant l'exécution de l'algorithme. Si le modèle de calcul est celui de machine de Turing, on autorise plusieurs transitions à partir d'un état et d'un symbole lu sur la bande.

La chasse NP est définie comme l'ensemble de tous les problèmes de décision solubles (ou langages reconnaissables) par NDTMs en temps polynomial. Dans ces machines non déterministes, il y a plusieurs choix de transitions possibles, si on fait tous ces choix, on obtient un arbre de calculs. Ces machines sont polynomiales si chaque branche est bornée par une fonction polynomiale en la longueur des données. Evidemment, cette classe contient P, depuis que DTMs sont des cas spéciaux de NDTMs. De plus, elle contient beaucoup plus de problèmes de fait que le non-déterminisme ajoute une puissance considérable aux calculs bornés en temps. En fait, telles machines n'existent pas, et à ce jour, aucun algorithme déterministe ne peut simuler ce type de machine en temps polynomial. L'algorithme le plus performant pour une telle tâche est de temps exponentiel. Le passage d'une machine non-déterministe à une machine déterministe est donc en temps exponentiel. Un problème NP peut alors être résolu en temps exponentiel par une machine déterministe.

Il est facile de considérer une version plus restreinte (mais pas moins puissante) de *NDTM* de temps polynomial : c'est la procédure «*guess-and-check*» de temps polynomial pour vérifie l'adhésion au langage (*yes-answers*). Sur une entrée *x*, telle procédure estime premièrement, de façon non déterministe, un mot *y* telle que $|y| \leq p(|x|)$, pour certain polynôme fixe *p* (dans certaines références ce mot *y* est appelé le "certificat" [Att01] [CP05] [Wil94]…), puis on exécute un algorithme de temps polynomial sur l'entrée (*x*,*y*). La réponse est "*Yes*" (*x* est dans *L*) si et seulement s'il y a un mot *y* tel que l'algorithme répondra "*Yes*". Un problème de décision (un langage) est dans la classe NP si et seulement si une telle procédure existe [Joh90].

NP est la classe des problèmes de décision pour lesquels il est facile de vérifier que les réponses sont correctes. On ne cherche pas un moyen de trouver une solution, mais de vérifier qu'une solution donnée soit correcte. On dit aussi que les problèmes de la classe P ont des solutions faciles à trouver, alors que ceux de la classe NP ont des solutions faciles à vérifier même si elles sont difficiles à trouver [Att01]. Donc, Un problème de décision *X* est dans la classe NP s'il existe une propriété π de P et un polynôme *p* tel que : pour toute entrée *w* du problème *X*, la réponse est "*Yes*" si et seulement s'il existe un mot *v* de longueur au plus $p(|w|)$ tel que (w, v) satisfasse la propriété π.

De plus, il est facile de voir qu'une grande variété de problèmes apparemment intraitables sont sensibles au procédure «*guess-and-check*» de temps polynomial. Considérant par exemple le problème suivant :

- *SATISFIABILITE* (*SAT*)
  - *Instance* : Liste de littéraux $U = (u_1, \overline{u_1}, u_2, \overline{u_2}, .. u_n, \overline{u_n})$, ensemble de clauses $C=(c_1,c_2,\ldots c_m)$, où chaque clause c_i est un sous ensemble de *U*.
  - *Réponse* : "*Yes*" s'il existe une fonction de vérité sur les variables u_1, u_2, … u_n telles que toutes les clauses dans *C* soient vraies.

Exemple : $U = \{u_1, u_2\}$, $C_1 = (\{\overline{u_1}, u_2\}, \{u_1, \overline{u_2}\})$ est satisfait par u_1 = Vraie, u_2 = Vraie, mais $C_2 = (\{u_1, u_2\}, \{\overline{u_1}\}, \{u_1, \overline{u_2}\})$ n'est pas satisfiable. (L'ensemble de clauses est sous forme normale conjonctive (FNC), c'est à dire que $C_1 = (\overline{u_1} \vee u_2) \wedge (u_1 \vee \overline{u_2})$ et $C_2 = (u_1 \vee u_2) \wedge (\overline{u_1}) \wedge (u_1 \vee \overline{u_2}))$

Il y a 2^n affectations possibles, et actuellement on ne connaît aucune méthode infaillible pour déterminer en temps sous exponentiel s'il existe une affectation qui satisfait toutes les clauses. Cependant, un algorithme «*guess-and-check*» a besoin seulement d'estimer une affectation de vérité (le certificat), vérifier que toutes les clauses sont satisfaites est alors facile.

Lemme *SAT est dans* NP.

Preuve

Pour la preuve, on suit le principe de la procédure «*guess-and-check*», il suffit de construire, de manière non-déterministe, des valeurs de vérité pour toutes les variables (qui constituent le certificat), puis d'évaluer la formule, ce qui se fait en temps linéaire en la taille de la formule.

Begin {input: *f* }

guess *t* in the set of assignments of values to the *n* variables of *f*;

if *t* satisfies *f* **then** accept;

else reject;

end.

D'une manière générale, les preuves d'appartenance à la classe NP se décomposent en deux phases [Jou] :

1. La première consiste à construire un arbre de choix de hauteur polynomiale, qui figure l'ensemble de tous les cas devant être examinés, en temps (non-déterministe) proportionnel à sa hauteur en utilisant le non-déterminisme de machine de Turing. Dans notre exemple, il s'agit de construire toutes les possibilités d'affecter des valeurs de vérité pour les variables.
2. Pour chaque choix, un calcul polynomial avec une machine déterministe. Lors de cette seconde phase, il s'agit de vérifier une propriété. Dans notre exemple, il s'agit de vérifier que les valeurs de vérité affectées aux variables rendent la formule vraie. C'est là la caractéristique essentielle des problèmes NP mentionnée précédemment.

Il suffit pour terminer de collecter les résultats aux feuilles afin de vérifier s'il existe un choix d'affectation de valeurs de vérité aux variables qui rend la formule vraie, ce qui est fait par la définition de l'acceptation pour les machines non-déterministes.

Les deux exemples suivants ont aussi la même propriété.

- *CLIQUE*
  - *Instance* : Graphe $G = (V, E)$, et un entier positif k.

- *Réponse* : "*Yes*" si *G* contient un sous graphe complet de taille *k*, c'est à dire s'il y a un sous ensemble $V' \subseteq V$ tel que $|V'| = k$, et pour tout $u, v \in V'$, $u \neq v$, la paire $\{u, v\}$ est une arête dans *E*.

- *TRAVELING SALESMAN DECISION PROBLEM* (*TSDP*)

  - *Instance* : Liste de villes $c_1, c_2, \ldots c_n$, une distance entière positive $d(c_i, c_j) = d(c_j, c_i)$ pour chaque paire (c_i, c_j) de villes distinctes, et un entier *B*.

  - *Réponse* : "*Yes*" s'il y a un tour de longueur totale $\leq B$, qui traverse toutes les villes, c'est-à-dire une permutation π d'indices $1, \ldots n$ telle que:

$$\sum_{i=1}^{n-1} d\left(c_{p(i)}, c_{p(i+1)}\right) + d\left(c_{p(n)}, c_{p(1)}\right) \leq B$$

De même, pour ce problème *TSDP* il y a (*n*!) tours possibles, et aucune méthode connue ne peut trouver la réponse de façon déterministe en temps sous exponentiel. Cependant, un algorithme «*guess-and-check*» a besoin seulement d'estimer le tour désiré, puis vérifier sa longueur.

Ces exemples permettent de conclure que P est un sous ensemble strict de NP. En fait, il existe beaucoup de problèmes dans NP pour lesquels aucun algorithme déterministe polynomial n'est connu, ce qui semble indiquer, avec une grande vraisemblance, que la classe NP est plus large que la classe P. La réponse de la question : P =? NP est donc très vraisemblablement négative, mais, aucune démonstration de ce résultat n'a été trouvée en plus de 30 ans de recherches très actives, d'où la célébrité de ce problème, qui intrigue tous les théoriciens de l'informatique.

Nous avons listé précédemment trois types de réductions qui sont compatibles avec P : les réductions de Turing en temps polynomial, les transformations polynomiales et les transformations en espace logarithmique. Par tradition, l'adjectif "NP-complet" est seulement réservé pour un seul type de ces réductions [Joh90].

Définition 3.19 Un problème (langage) est NP-complet s'il est complet pour NP sous transformations polynomiales.

Ces problèmes NP-complets ont certaines propriétés : ils sont tous très difficiles à résoudre et aucun algorithme de temps polynomial n'a été trouvé pour aucun d'entre eux, en contre partie, l'impossibilité de trouver des algorithmes polynomiaux pour ces problèmes n'a pas été prouvée [Att01]. Cependant, si un algorithme rapide pouvait être trouvé pour un seul problème NP-complet, il y aura des algorithmes rapides pour tous les autres problèmes. En revanche, s'il pouvait être prouvé qu'aucun algorithme n'existe pour un des problèmes NP-complets, il ne pourrait y en avoir pour aucun autre [Wil94] [Att01], ainsi on dit qu'ils sont équivalents. Enfin, l'existence ou non d'algorithmes polynomiaux pour les problèmes NP-complets est probablement considéré comme un des principaux problèmes encore irrésolus en informatique.

Le premier problème NP-complet qui a été identifié est le problème de "*SATISFIABILITE*", le résultat qui l'identifie est appelé "*théorème de Cook*".

Théorème 3.2 (Steven COOK) *SATISFIABILITE est NP-complet.*

Steven Cook prouva aussi que le problème "*CLIQUE*" est NP-complet. Peu de temps après, la classe des problèmes NP-complets était étendue pour inclure le *TSDP* et une grande variété de problèmes.

Une dernière observation intéressante est qu'une petite modification peut changer un problème polynomial à un problème NP-complet, comme il est montré par les exemples du tableau suivant [Joh90].

Problèmes dans P	Problèmes NP-complets
2-SAT - *Instance* : une instance de *SAT* dans laquelle aucune clause ne contient plus de 2 littéraux. - *Réponse* : "*Yes*" si toutes les clauses sont satisfiables.	*3-SAT* - *Instance* : une instance de *SAT* dans laquelle aucune clause ne contient plus de 3 littéraux. - *Réponse* : "*Yes*" si toutes les clauses sont satisfiables.
EDGE COVER - *Instance* : un graphe $G=(V,E)$, un entier k. - *Réponse* : "*Yes*" s'il y a un sous ensemble $E' \in E$ avec $\|E'\| \leq k$ tel que tout sommet est le point final d'une arête dans E.	*VERTES COVER* - *Instance* : un graphe $G=(V,E)$, un entier k. - *Réponse* : "*Yes*" s'il y a un sous ensemble $V' \in V$ avec $\|V'\| \leq k$ tel que toute arête a un point final dans V.
EULER CYCLE - *Instance* : un graphe $G=(V,E)$, avec $\|E\|=m$. - *Réponse* : "*Yes*" s'il y a un ordonnancement $e_1, e_2, \ldots e_m$ tel que e_1 et e_m partagent un point final, et même pour toutes les paires $\{e_i,e_{i+1}\}$, $1 \leq i < n$.	*HAMILTONIAN CYCLE* - *Instance* : un graphe $G=(V,E)$, avec $\|V\|=n$. - *Réponse* : "*Yes*" s'il y a un ordonnancement $v_1, v_2, \ldots v_n$ tel que $\{v_1, v_n\}$ forme une arête, et même pour toutes les paires $\{v_i,v_{i+1}\}$, $1 \leq i < n$.

Tableau 3. 1 : Exemples de problèmes P et NP-complets

3.2.2. Les classes co-NP et NP Ç co-NP

Considérant maintenant le problème complémentaire au *SAT*, dans lequel nous cherchons s'il est le cas où toute affectation échoue à satisfaire au moins une clause, ceci est simplement un problème de satisfiabilité avec les réponses "*Yes*" et "*No*" inversées, et d'ici il est équivalent en termes de calcul au problème de satisfiabilité. Cependant, ce n'est pas du tout clair que ce problème "co-*SATISFIABILITE*" est NP-complet ou même dans NP! [Joh90]

co-NP dénote l'ensemble de tous les langages $\{0,1\}^*-L$, où $L \in$ NP, et nous avons la question sur l'égalité NP=?co-NP. Notez que NP doit être égale à co-NP si n'importe quel problème NP-complet, comme *SAT*, devrait être dans co-NP, et que si NP≠co-NP, alors aucun des deux ensembles ne peut contenir l'autre.

Une conséquence importante de l'inégalité supposée de NP et co-NP est qu'elle nous fournit une autre classe potentiellement intéressante : c'est la classe NP∩co-NP, cette classe serait une sous classe des deux autres classes et elle contiendrait P, la question qui se pose donc est : est ce que P = NP∩co-NP ?

3.2.3. Les problèmes NP-difficiles et NP-faciles

La classe NP est limitée aux problèmes de décision, alors qu'en pratique les problèmes qui nous intéressent sont souvent des problèmes de calcul de fonctions ou plus généralement des problèmes de recherche. Par exemple, le *TSDP* défini précédemment n'est pas réellement le problème qu'on veut résoudre. En pratique, on veut trouver un tour qui a la longueur la plus courte possible (c'est le problème appelé *TSP* : *Traveling Salesman Problem*), non simplement de dire s'il existe un tour qui obéit à une borne de longueur donnée. On remarque que le problème d'optimisation n'est pas -peut être- plus facile que le problème *TSDP* (trouver un tour optimal nous permettrait de dire s'il existe un tour dont la longueur obéit à la borne donnée ou non). Dans ce contexte, on a introduit le terme "*NP-difficile*".

Définition 3.20 Un problème de recherche *X* est NP-difficile (NP-*hard*) si pour certain problème NP-complet *Y* il existe une réduction de Turing en temps polynomial de *Y* à *X*.

D'une manière générale, le problème d'optimisation associé à un problème de décision NP-complet est NP-difficile [Joh90]. La version *TSP* du problème *TSDP* est clairement NP-difficile, comme tous les problèmes de recherche qui contiennent des problèmes NP-complets comme des cas spéciaux.

Les problèmes NP-difficiles sont au moins aussi difficiles que les problèmes NP-complets. En face de ces problèmes, nous trouvons la notion des problèmes NP-faciles, qui sont définis comme suit :

Définition 3.21 Un problème de recherche *X* est NP-facile (NP-*easy*) si pour certain problème *Y* dans NP il y a une réduction de Turing en temps polynomial de *X* à *Y*.

Les problèmes NP-easy et les problèmes NP-hard sont solubles en temps polynomial si P=NP [Joh90].

3.2.3. La classe D^P

Définition 3.22 La classe D^P se compose de tous les langages qui peuvent être exprimés de la forme $X \cap Y$, où $X \in$ NP et $Y \in$ co-NP.

Cette classe est différente de la classe NP∩co-NP, elle contient en fait tous les problèmes de NP∪co-NP, et elle n'égale pas cette union à moins que NP=co-NP [Joh90].

Une variété de types de problèmes intéressants paraît dans D^P, parmi lesquels, on a le type de problèmes de "criticité" (*criticality*). Considérant, par exemple, le problème de "*SATISFIABILITE CRITIQUE*" : étant donnée une instance de problème *SAT*, est-il le cas où l'ensemble de clauses est non satisfiable, mais la suppression de n'importe quelle clause est suffisante de produire un sous ensemble qui est satisfiable? Ici la restriction de "non-satisfiabilité" détermine un langage co-NP, et les *m* restrictions de satisfiabilité (tel que *m* est le nombre de clauses) peuvent être combinées en une seule instance de "satisfiabilité" qui est dans NP, ce problème est donc D^P-complet.

3.2.4. PSPACE et ses sous-classes

La classe PSPACE est l'ensemble de tous les langages reconnaissables par machines de Turing (déterministes) en espace polynomial, cette classe est égale à la classe APTIME [Joh90], qui se compose de tous les langages acceptés par ATMs en temps polynomial, cette égalité est une conséquence d'un résultat plus général prouvé par *Chandra*, *Kozen* et *stockmeyer*, avant de présenter ce théorème, nous présentons d'abord certaines notions qui seront utilisées ensuite.

Définition 3.23 Si T(n) est une fonction de l'ensemble des entiers positifs à lui-même, alors DTIME[T(n)] (resp. NTIME[T(n)], ATIME[T(n)]) est l'ensemble de tous les langages reconnus par DTMs (resp. NDTMs, ATMs) en temps borné par T(n), où n est la taille de l'entrée.

Définition 3.24 Si S(n) est une fonction de l'ensemble des entiers positifs à lui-même, alors DSPACE[S(n)] (resp. NSPACE[S(n)], ASPACE[S(n)]) est l'ensemble de tous les langages reconnus par DTMs (resp. NDTMs, ATMs) en espace borné par S(n), où n est la taille de l'entrée.

Par exemple : P = DTIME[$n^{O(1)}$], NP = NTIME[$n^{O(1)}$] et PSPACE = DSPACE[$n^{O(1)}$].

Théorème 3.3 (Chandra, Kozen et Stockmeyer) [CKS81] *Pour toute fonction* T(n) ≥ n :

$$\text{ATIME}[T(n)] \subseteq \text{DSPACE}[T(n)] \subseteq \cup_{c>0} \text{ATIME}[c.T(n)^2]$$

En prenant une union sur tous les polynômes, On peut obtenir le résultat suivant [Joh90]:

$$\text{APTIME} = \cup_{k>0} \text{ATIME}\,[n^k] = \cup_{k>0} \text{DSPACE}\,[n^k] = \text{PSPACE}.$$

Avec PSPACE nous atteignons notre première classe de complexité contenant P qui n'est pas connue à s'écrouler à P si P = NP.

Ainsi PSPACE = APTIME est une nouvelle classe distincte, Le problème PSPACE-complet paradigmatique de cette classe est un problème de la logique, *QBF* pour "*Quantified Boolean Formula*", pour lequel il s'agit de décider si une formule propositionnelle quantifiée est satisfiable ou non. Le langage est donc bâti à partir d'un nombre fini de variables propositionnelles, des connecteurs logiques habituels, et des quantificateurs universel et existentiel (∀, ∃). Ce problème est défini comme suit :

- *QUANTIFIED BOOLEAN FORMULA* (*QBF*) :
  - *Instance* : Une sentence de la forme $S = (Q_1x_1)(\ Q_2x_2)\ldots(\ Q_jx_j)[F(x_1, x_{2,\ldots}\ x_j)]$ où chaque Q_i, $1 \le i \le j$, est un quantificateur (∃ ou ∀) et F est une expression booléenne dans les variables données.
  - *Réponse* : "*Yes*" si S est une sentence vraie.

Pour ce problème on dit qu'une variable est libre dans une formule si elle n'est pas dans la portée d'un quantificateur. Ce problème est dans PSPACE, et de plus il est PSPACE-complet.

Lemme *QBF* est dans PSPACE.

La preuve utilise l'élimination des quantificateurs sur un ensemble fini, ici l'ensemble des valeurs de vérité. Nous avons en fait les équivalences logiques suivantes:

$\forall x\ \phi(x) \equiv \phi(0) \wedge \phi(1)$.

$\exists x\ \phi(x) \equiv \phi(0) \vee \phi(1)$.

En définissant une procédure d'évaluation récursive *EVAL* qui utilise un espace de travail polynomial dans la pile d'exécution pour évaluer ϕ :

1. ϕ= *true* : *EVAL* retourne la valeur *true*,

2. $\phi = \phi_1 \wedge \phi_2$: *EVAL* retourne la conjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2,

3. $\phi = \phi_1 \vee \phi_2$: *EVAL* retourne la disjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2,

4. $\phi = \neg\ \psi$: *EVAL* retourne la négation du résultat obtenu récursivement pour ψ,

5. $\phi = \exists x\ \psi$: *EVAL* construit les formules ψ_0 et ψ_1 en remplaçant respectivement la variable *x* par 0 et 1 dans ψ, puis retourne la disjonction des résultats obtenus récursivement pour ψ_0 et ψ_1,

6. $\phi = \forall x\ \psi$: *EVAL* construit les formules ψ_0 et ψ_1 comme précédemment, puis retourne la conjonction des résultats obtenus récursivement pour ψ_0 et ψ_1.

Comme le nombre de connecteurs ou quantificateurs est au plus égal à la taille *n* de la formule, la profondeur de récursion est au plus *n*. La taille des données à stocker dans la pile d'exécution étant linéaire en *n*, on en déduit que l'espace occupé dans la pile est quadratique. Donc *QBF* est dans PSPACE. De plus, le problème *QBF* est dans la classe PSPACE-complet.

Théorème 3.4 *QBF* est SPACE-complet.

Commençant avec *QBF*, les chercheurs ont identifié une grande variété d'autres problèmes PSPACE-complets.

Avec PSPACE, nous atteignons le sommet de la tour des classes pour lesquelles la question de confinement dans P reste ouverte. La classe évidente suivante, NPSPACE pour un espace polynomial non déterministe, a plutôt une caractéristique sérieuse : comme APTIME, elle est égale à PSACE lui-même, ceci est conséquence du théorème de *Savitch* prouvé en 1970.

Théorème 3.5 (Savitch)[Sav70] *Pour toute fonction* $T(n) \geq log_2\ n$:

$$NSPACE[T(n)] \subseteq PSPACE[T(n)^2].$$

Ainsi, toute tentative d'une hiérarchie alternative au-dessus de PSPACE s'écroule à PSPACE elle-même [Joh90].

3.3. Problèmes prouvés intraitables

Aucune classe de complexité parmi les classes considérées dans les sections précédentes ne contient un problème prouvé intraitable. Si P = PSPACE, alors tous les problèmes dans les classes discutées seront solubles en temps polynomial. Dans cette section, nous considérons des classes de complexité qui contiennent des problèmes intraitables, nous nous intéresserons aux classes avec bornes de ressources exponentielles.

3.3.1. La classe EXPTIME et ses variantes:

Il y a actuellement deux notions distinctes de "temps exponentiel", dans ce qui suit, nous utiliserons une distinction terminologique entre ces deux notions (mais la terminologie utilisée n'est pas complètement standard). La première est la notion la plus naturelle du temps exponentiel, permettant de définir la classe suivante :

Définition 3.25 La classe EXPTIME est l'ensemble de tous les problèmes de décision solubles en temps borné par $2^{p(n)}$, où p est un polynôme, par DTMs, c'est-à-dire : EXPTIME = $\cup_{k>0}$ DTIME [2^{n^k}].

La classe EXPTIME est égale à la classe APSPACE, classe de tous les langages reconnaissables par ATMs en utilisant un espace borné polynomialement, cette équivalence est conséquence d'un théorème de *Chandra*, *Kozen* et *Stockmeyer*.

Théorème 3.6(Chandra, Kozen et Stockmeyer)[CKS81] *Pour toute fonction* S(n) ≥ *log n* :

$$\text{ASPACE}[S(n)] = \cup_{c>0} \text{DTIME}\,[2^{c.S(n)}]$$

La classe EXPTIME contient les deux classes : PSPACE et NP. EXPTIME est pour un intérêt principale car elle est la classe du temps déterministe connue la plus large qui contient NP [ALR99].

La deuxième notion du temps exponentiel se restreint aux exposants linéaires, et permet de définir la classe suivante :

Définition 3.26 La classe ETIME est égale à $\cup_{c>0}$ DTIME [$2^{c.n}$].

Notez que ETIME est proprement contenue dans EXPTIME. Ces deux classes sont indépendantes de modèles de machines [Joh90].

Les analogues non déterministes de EXPTIME et ETIME sont les classes suivantes :

Définition 3.27 La classe NEXPTIME est égale à $\cup_{k>0}$ NTIME [2^{n^k}].

Définition 3.28 La classe NETIME est égale à $\cup_{c>0}$ NTIME [$2^{c.n}$].

3.3.2. Au-delà de NEXPTIME

Dans cette section nous présentons quelques autres classes intéressantes de problèmes intraitables, la première est la suivante :

Définition 3.29 La classe EXPSPACE est égale à $\cup_{k>0}$ DSPACE [2^{n^k}].

Les classes suivantes dans l'échelle sont celles notées par "2-EXPTIME" et "2-NEXPTIME" : les classes des problèmes de décision solubles respectivement par DTMs et NDTMs en temps borné par $2^{2^{p(n)}}$, pour certain polynôme p. Continuant de remonter dans l'échelle, nous avons maintenant 2-EXPSPACE, puis k-EXPTIME, k-NEXPTIME et k-NEXPSPACE, avec $k \geq 3$, où k correspond au nombre de niveaux de puissance dans la fonction de limite de ressource approprie. Cependant, la classe d'intérêt réel suivante est l'union de cette hiérarchie, c'est la classe ELEMENTARY de problèmes de décision (*"Elementary" decision problems*) [Joh90].

Définition 3.30 La classe ELEMENTARY = $\cup_{k>1}$ *k*-EXPTIME.

3.4. À l'intérieur de P

Dans les sections précédentes, nous avons présenté des classes de complexité contenant la classe P. Cette section a pour objectif de présenter les classes des problèmes qui sont en divers sens "plus faciles" que les problèmes de la classe P (ou au moins incomparables), nous traversons ces classes de plus grandes aux plus petites.

3.4.1. Les classes POLYLOGSPACE, L, NL et SC

Aujourd'hui, beaucoup de calculs se font sur ordinateurs personnels qui ont une mémoire centrale limitée, par conséquent, la quantité de mémoire exigée par un algorithme peut être souvent plus critique que son temps d'exécution. Ainsi on s'intéresse aux algorithmes qui utilisent de façon significative moins de bandes de travail que la taille de son entrée, ce cas idéal est capturé par l'introduction des classes définies par des bornes d'espace sous linéaire, essentiellement les classes "LOGSPACE" et "POLYLOGSPACE".

Définition 3.31 La classe L (ou LOGSPACE) se compose de tous les problèmes de décision solubles par DTMs avec bande de travail bornée par $O(log\ |x|)$ sur une entrée *x*.

Définition 3.32 La classe POLYLOGSPACE se compose de tous les problèmes de décision solubles avec bande de travail bornée par $O(log^k\ |x|)$, pour certain *k* fixe et une entrée *x*.

D'autre manière, en utilisant la terminologie de la section ***3.2.5***, on dit que : L=DSPACE[$O(log\ n)$] et POLYLOGSPACE=$\cup_{k>1}$DSPACE[$log^k\ n$]. On peut voir immédiatement que L $\subseteq$ POLYLOGSPACE. Ce qui est plus intéressant est les relations entre ces deux classes et la classe P.

L $\subseteq$ P, depuis qu'aucune DTM d'espace logarithmique ne peut avoir plus qu'un nombre polynomial d'états de mémoire distincts [Joh90].

Pour POLYLOGSPACE et P la situation est plus compliquée, il est montré que POLYLOGSPACE $\neq$ P, et on ne connaît pas aussi si les deux classes sont incomparables ou si une classe contient totalement l'autre, la plupart des chercheurs préfèrent la non comparabilité.

Supposant que le temps polynomial et l'espace poly-logarithmique sont des propriétés importantes pour un algorithme à avoir, mais qu'aucun ne garantit l'autre, une classe naturelle à considérer est la classe SC, qui se compose de tous les problèmes de décision solubles par algorithmes qui obéissent simultanément aux deux types des bornes. Pour définir cette classe formellement, on introduit la notion suivante de TS[$t(n)$, $s(n)$] pour capturer la notion de bornes de temps et d'espace simultanées :

Définition 3.33 La classe TS[$t(n)$, $s(n)$] est l'ensemble de tous les problèmes solubles par DTMs qui utilisent au plus un temps $t(n)$ et un espace $s(n)$ sur entrées de taille *n*.

Définition 3.34 La classe SC = TS[$n^{O(1)}$, $log^{O(1)}n$].

C'est à dire que SC est la classe de tous les problèmes de décision solubles par DTMs qui utilisent simultanément un temps polynomial et un espace poly-logarithmique.

Le nom SC veut dire "*Steven's Class*", à l'honneur de "*Steven COOK*", qui prouva le premier résultat profond pour cette classe. Notez que SC $\subseteq$ P$\cap$POLYLOGSPACE, et elle peut ne pas égaler l'intersection : c'est le cas où un problème *X* peut être résolu par un algorithme en temps polynomial et par un deuxième algorithme qui utilise un espace polylog, mais par aucun algorithme qui obéit simultanément aux deux bornes des ressources.

Pour conclure cette section, regardons brièvement aux analogues non déterministes des classes que nous avons vu dans cette section, la plus importante est la classe suivante :

Définition 3.35 La classe NL se compose de tous les problèmes de décision solubles par NDTMs qui utilisent, étant donnée une entrée *x*, un espace borné par $O(log\ |x|)$.

La classe NPOLYLOGSPACE, définie de manière analogue à NL, est simplement égale à POLYLOGSPACE, et la classe co-NL est égale à NL, et on a comme relation générale L $\subseteq$ NL $\subseteq$ POLYLOGSPACE [Joh90], on a aussi NL $\subseteq$ P [Tre04] [Joh90], et comme pour la classe P, la classe NL est fermée sous complément : NL = co-NL [Tre04].

3.4.2. Calcul parallèle et la classe NC

Dans la section précédente nous avons considéré le cas d'ordinateurs personnels, un phénomène de calcul relativement récent. Dans cette section nous nous tournerons vers un autre type d'ordinateurs : l'ordinateur massivement parallèle. Malgré nos arguments dans la section précédente de l'importance de l'espace mémoire par rapport au temps pour des calculs pratiques, il y a des situations où le temps polynomial, même linéaire, peut être trop grand, et les praticiens acceptent volontiers d'appliquer plus de processeurs à une tache, cette démarche réduit considérablement le temps d'exécution.

En se basant sur ce principe et sur le modèle PRAM (*Parallel Random Access Machine*), on peut définir des classes en termes de temps parallèle et de nombre de processeurs.

Définition 3.36 La classe TP[$t(n)$, $p(n)$] est l'ensemble de tous les problèmes solubles par PRAMs qui consomme au plus, sur des entrées de taille *n*, un temps $t(n)$ avec $p(n)$ processeurs.

Définition 3.37 La classe NC = TP[$log^{O(1)}n$, $n^{O(1)}$].

C'est-à-dire, la classe NC se compose des problèmes qui sont solubles sur PRAM qui obéit simultanément à une borne poly-logarithmique sur le temps d'exécution et à une borne polynomiale sur le nombre de processeurs utilisés. Plus informellement, on peut dire que NC se compose des problèmes solubles avec une quantité bornée polynomialement de hardware dans un temps poly-logarithmique. Comme les classes de complexité pour les calculs séquentiels vues dans les sections précédentes, cette classe est considérablement "*model-independent*" [Joh90].

Cette classe reste dans P, c'est à dire que NC $\subseteq$ P [GHR95], depuis qu'une computation par *RAM* avec n^k processeurs en temps $log^j\ n$ peut être simulée par un seul processeur en temps $O(n^k.log^j\ n)$ [Joh90].

La classe de complexité analogue de NC pour les problèmes de recherche est la classe FNC définie comme suit :

Définition 3.38 La classe FNC se compose de tous les problèmes de recherche solubles par PRAMs en temps polylog avec un nombre de processeurs borné polynomialement.

Notez que NC $\subseteq$ FNC par définition. Parmi les problèmes de recherche dans FNC nous trouvons "*Matrix Multiplication*", et "*Euler Tours*", et une grande variété de problèmes algébriques [Joh90].

3.5. Conclusion

Dans ce chapitre, nous avons présenté quelques classes de complexité, les relations entre elles, et des exemples de problèmes qu'elles contiennent. Mais ce n'est pas tout ce qu'il existe, en fait, dans la littérature il y a une grande variété de classes de complexité, dont certaines peuvent être souvent définies en termes de plusieurs modèles de machines différents, toujours avec présentations de nouveaux concepts, notions et théorèmes… On trouve dans la "*Complexity Zoo*" [AK05] plus de 440 classes de complexité… Nous nous somme concentrés sur les classes de complexité les plus importantes : les classes définies par des limites logarithmiques, polynomiales et exponentielles sur le temps d'exécution et l'espace mémoire pour des modèles déterministes et des modèles non déterministes. Ainsi, nous avons considéré seulement les classes de complexité appelées "*canonical complexity classes*" [ALR99] [ALR04] ou "*Popular complexity classes*" [Wat05] qui sont présentées avec leurs interrelations dans la figure 3.4.

La plupart des relations d'inclusion entre ces classes de complexité ne sont pas prouvées strictes. Il est claire que, pour toutes les fonctions de temps $f(n)$ et d'espace $s(n)$, DTIME[$t(n)$] $\subseteq$ NTIME[$t(n)$] et DSPACE[$s(n)$] $\subseteq$ NSPACE[$s(n)$], parce qu'une machine déterministe est un cas spécial de machine non déterministe, de plus DTIME[$t(n)$] $\subseteq$ DSPACE[$t(n)$] et NTIME[$t(n)$] $\subseteq$ NSPACE[$t(n)$], parce qu'on peut utiliser au plus une nouvelle case du ruban à chaque étape de calcul, et donc en temps $t(n)$ on peut utiliser au plus un espace $t(n)$ [Bou03]. Les autres relations d'inclusion importantes sont présentées dans la figure suivante.

Plusieurs questions sont rencontrées lors de l'étude des classes de complexité, les plus intéressantes sont celles connues sous le nom "*big questions*" : est-ce que P = NP? est-ce que NP = co-NP ?… Ces questions sont ouvertes depuis plus de 30 ans! Elles sont certainement les questions ouvertes les plus importantes en informatique théorique et peut être en mathématique.

Deux classes de complexité importantes sont la classe P, l'ensemble de problèmes qui peuvent être résolus en temps polynomial par des algorithmes déterministes (dits "efficaces"), et la classe NP, l'ensemble de problèmes dont les solutions peuvent être vérifiées en temps polynomial. La première classe est la classe de problèmes dits "solubles efficacement", tandis que les meilleurs algorithmes connus actuellement pour les problèmes de la deuxième classe ont un coût (en temps d'exécution) exponentiel, et personne n'a encore réussi à démontrer que ces problèmes n'appartiennent pas à la classe P.

La plupart des problèmes computationnels communs rentrent dans un petit nombre de classes de complexité, en particulier les classes P et NP. Cependant, déterminer la classe de complexité d'un problème donné est une tâche très compliquée… En fait, il a fallu de nombreuses années (2000 ans) aux meilleurs chercheurs pour prouver que le problème de savoir si un nombre est premier ou non est dans la classe P [Har02] [HC91], il n'a été démontré que tout récemment (en août 2002 par *Agrawal*, *Keyal* et *Saxena*).

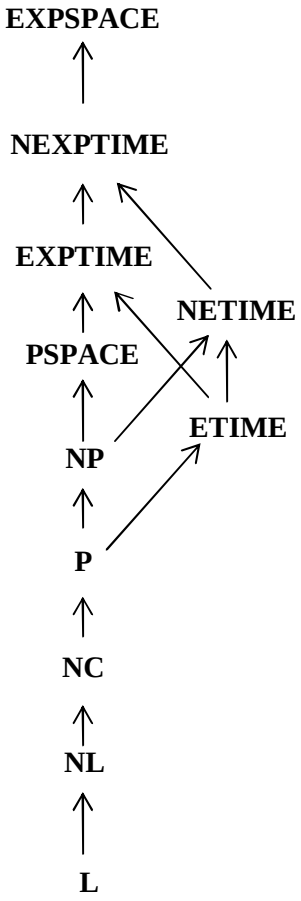


Figure 3. 4 : Les classes de complexité canoniques.

Chapitre 4

Calcul de la complexité computationnelle

L'étude de la quantité de l'effort computationnel exigé pour exécuter certain type de calculs est l'étude de "la complexité computationnelle" (appelée aussi "la complexité algorithmique"). Ainsi, la notion de la complexité des algorithmes est utilisée pour préciser l'idée selon laquelle un algorithme est caractérisé par un coût.

Il y a aujourd'hui un intérêt très marqué parmi les mathématiciens et les informaticiens pour la complexité computationnelle [Sim90] évaluée par le coût maximal (ou attendu) des ressources nécessaires pour résoudre les problèmes informatiques. La théorie de la computation et les techniques d'analyse de l'efficacité de l'algorithme définissent la complexité du problème en termes du temps et de l'espace mémoire requis par l'algorithme qui résout ce problème pour l'exécution des calculs. Elle est une mesure de l'efficacité des algorithmes utilisés pour implémenter un programme. Une mesure de la complexité computationnelle est donc une valeur qui indique le niveau de ressources requises pour résoudre un problème.

Ainsi, le présent chapitre se fixe pour objectif de comprendre, étant donné un algorithme pour la résolution d'un problème, la manière de décider la complexité computationnelle du problème donné. Pour cela, il nous est apparu nécessaire de clarifier la notion d'algorithme et de complexité en précisant quelques hypothèses sur le modèle de machine considéré, les notions d'instructions élémentaires, instructions fondamentales, taille des données…etc. Ensuite, nous présenterons la démarche générale pour calculer la complexité computationnelle asymptotique d'un algorithme.

Comme nous avons vu dans les chapitres précédents, la mesure de la complexité (et par conséquent la définition de la classe de complexité) d'un problème est basée sur la détermination d'une ressource (qui est souvent le temps ou l'espace mémoire) et d'un modèle de calcul qui est souvent le modèle de machine de Turing. Mais dans la pratique on n'utilise pas cette machine de Turing parce qu'elle est rarement disponible, d'autre part, conformément à la thèse de l'invariance, tous les modèles de la calculabilité connus se sont révélés avoir une puissance de calcul équivalente à celle des machines de Turing, et ils donnent plus ou moins les mêmes notions de complexité à un facteur polynomial près. Nos mesures de la complexité seront donc basées directement sur l'algorithme lui-même, en se basant sur le modèle *RAM*.

4.1. Notion d'algorithme

L'algorithmique est une science apparue, il y a très longtemps, bien avant l'idée même d'ordinateur. Citons, vers 1800 avant J-C, les babyloniens de l'époque de "*HAMMURABI*" formulaient des règles précises pour la résolution de certains types d'équations. Plus tard, au 3ème siècle avant J-C, les grecs connaissaient un grand nombre de procédés de calcul, dont le célèbre «*algorithme d'Euclide*». Au 9ème siècle après J-C, on trouve l'origine du mot «*algorithme*», qui provient du nom de "*Abu Ja'far Mohammed Ibn Mûsâ Al-Khowârismi*", qui écrivait un ouvrage d'arithmétique utilisant les règles de calcul sur la représentation décimale des nombres. Il eut une influence capitale pendant plusieurs siècles. Au fil du temps, la signification du mot s'élargit et finalement vient à désigner tout procédé de calcul systématique, voire automatique, mais sans référence nécessaire à une machine. Du côté de l'informatique, on peut dire qu'un algorithme est, tout simplement, un « mode d'emploi pour résoudre un problème ».

La notion d'un algorithme n'était pas en fait formalisée jusqu'aux années 30, par des mathématiciens comme *Alan TURING* à l'Angleterre et (indépendamment) *Alonzo CHURCH* à l'Amérique. La formalisation de Turing était par l'ordinateur primitif appelé aujourd'hui "machine de Turing". La machine de Turing a été apparue pour la première fois en 1936, quelques dizaines d'années après les ordinateurs réels ont été inventés. La formalisation de Turing de la notion d'un algorithme est : "un algorithme est ce qu'une machine de Turing implémente" [HC91]. On peut dire d'autre manière, selon la thèse de *CHURCH*, que, étant donné un algorithme, il y a certaine machine de Turing qui l'implémente [HC91]. C'est-à-dire que les machines de Turing fournissent une représentation possible pour les algorithmes de résolution des tâches pouvant être traitées algorithmiquement, et que tout algorithme peut être représenté sous la forme d'une table de transitions d'une machine de Turing. On admettra pour la théorie de la complexité cette thèse, c'est-à-dire l'équivalence entre machine de Turing et algorithme [RT03]

Pour nos besoins, nous pouvons considérer la définition suivante [Fro05] :

Définition 4.1 Un algorithme est un procédé de calcul automatique composé d'un ensemble fini d'étapes qui permettent de résoudre un problème en donnant la sortie requise, chaque étape est formée d'un nombre fini d'étapes élémentaires. Chaque étape élémentaire est :

- définie de façon rigoureuse et non ambiguë,
- effective, c'est-à-dire pouvant être réalisée par une machine.

De plus, l'algorithme doit toujours terminer après un nombre fini d'opérations et fournir un résultat, quelle que soit la donnée en entrée.

À tout algorithme écrit en langage naturel ou pseudo-code on peut associer un programme qui l'implémente : programme écrit avec des règles syntaxiques rigoureuses et destiné à être compris par une machine. C'est-à-dire qu'un algorithme est indépendant du langage de programmation dans lequel on va l'exprimer et de l'ordinateur utilisé pour le faire tourner, c'est une description abstraite des étapes conduisant à la solution d'un problème. Donc l'algorithme est la partie conceptuelle du programme (indépendante du langage et de la machine), et le programme est l'implémentation de l'algorithme dans un langage de programmation et sur un système particulier. Se pose alors le problème de trouver le bon

degré de précision nécessaire pour définir l'algorithme : il doit être suffisant pour permettre d'écrire un programme lui correspondant et pour calculer sa complexité.

4.2. Complexité d'algorithme

On se rend intuitivement compte que tous les problèmes pouvant être résolus algorithmiquement (par une machine de Turing) ne sont pas de même complexité, mais comment mesurer cette complexité ?

La complexité d'un problème peut être mesurée par le temps d'exécution (ou encore la quantité de mémoire) requis pour la mise en oeuvre d'un algorithme qui le résout sur un certain jeu de données (variables).

Cependant, les notions de temps de traitement et d'espace mémoire sont dépendantes de la machine physique utilisée pour implémenter l'algorithme, nous avons donc besoin d'une mesure absolue, c'est-à-dire indépendante de toute implémentation.

La notion de machine de Turing va nous permettre de définir une telle mesure absolue de la difficulté d'un problème indépendante de la machine physique utilisée pour implémenter l'algorithme [Cha02], elle sera appelée la complexité (algorithmique) du problème.

Considérons un problème *P* formalisable et décidable, et une machine de Turing *T* résolvant *P*. Rappelons que :

- La complexité temporelle de *P* est le nombre de déplacements de la tête de lecture/écriture que devra effectuer la machine *T* pour résoudre une instance de *P*.
- La complexité spatiale de *P* est la longueur de bande de la machine *T* nécessaire pour résoudre une instance de *P*.

Ainsi, dans l'approche théorique, la complexité temporelle et spatiale d'un problème est mesurée par rapport à la machine de Turing représentant l'algorithme qui résout ce problème. Mais, dans la pratique, cette machine de Turing est rarement disponible... [Cha02] et on mesure la complexité d'un problème par le nombre d'instructions "élémentaires" nécessaires à l'exécution de l'algorithme qui le résout.

En fait, les algorithmes sont la partie la plus importante, durable et originale en informatique, parce qu'ils peuvent être étudiés dans une manière indépendante de machine et de langage. Ça veut dire que nous avons besoin des techniques qui nous permettent de comparer des algorithmes sans les implémenter [Ski97] : Nos deux importants outils - comme nous verrons par la suite- sont :

i) le modèle *RAM*, et

ii) l'analyse asymptotique de la complexité dans le pire des cas.

Par "instruction élémentaire", nous entendrons ici toute instruction qui peut être réalisée par une machine de Turing en un nombre de déplacements de la tête de lecture/écriture borné par une constante connue a priori [Cha02]. Par exemple, écrire un caractère à l'écran, lire un caractère dans un fichier, affecter une valeur atomique (un caractère, un

entier, un réel, ...) à une variable, réaliser une opération arithmétique sur des valeurs atomiques… sont des instructions élémentaires.

S'intéresser donc à la complexité d'un algorithme, c'est chercher à évaluer les ressources utilisées par l'algorithme. Plus précisément, on cherche à exprimer le coût de l'algorithme - la quantité des ressources utilisées- en fonction de la taille des données, nous nous intéresserons la plupart du temps à la complexité temporelle et à la complexité spatiale. Avant de chercher à estimer la complexité, il faut donc préciser la taille des données.

4.2.1. Taille des données

Formellement, la taille d'une donnée est la longueur de la chaîne de caractères (en binaire) qui la code. Cependant, dans la pratique, le codage binaire donne une chaîne dont la longueur est proportionnelle à un paramètre plus simple, qui est souvent facilement reconnaissable et unanimement accepté. Par abus de langage, c'est ce dernier paramètre qui représentera la taille de la donnée.

Pour une entrée constituée de n données, la taille à prendre dans l'approche théorique est la taille de la séquence binaire permettant de coder ces n données. Cette taille, dans la plupart des cas, est de la forme $a.n + b$, où a et b sont des constantes, nous pourrons utiliser n comme mesure de la taille des entrées, et donc exprimer la complexité par rapport à n (en fait: $a.n + b = O(n)$).

Ainsi, la taille des données, utilisée pour exprimer la complexité, peut être le nombre d'éléments pour une liste, le nombre de sommets pour un graphe, les dimensions pour une matrice… etc.

4.2.2. Modèle de calcul RAM

La conception d'algorithmes indépendants de machine dépend d'un ordinateur hypothétique appelé *RAM* (*Random Access Machine*). Sous ce modèle de calcul, nous somme en face d'un ordinateur où :

- Chaque opération simple (par exemple +, *, -, test, …) prend exactement un pas de temps.
- Les boucles et les sous routines ne sont pas considérées comme opérations simples, elles sont la composition de plusieurs opérations simples. Le temps exigé pour passer à travers une boucle ou pour exécuter une sous routine dépend du nombre d'itérations de la boucle ou de la nature spécifique de la sous routine.
- Nous avons autant de mémoire que nous avons besoin [Ski97].

Sous le modèle *RAM*, nous mesurons le temps d'exécution d'un algorithme par calcul du nombre de pas qu'il prend pour une instance donnée de problème. Par supposition que notre *RAM* exécute un nombre donné de pas par seconde, le nombre d'opérations peut être converti facilement au temps d'exécution réel.

Le modèle *RAM* est un modèle simple de fonctionnement des ordinateurs. Une plainte commune est qu'il est trop simple de manière que les suppositions font les analyses et les conclusions trop grossières à croire en pratique. Par exemple, sur la plupart des processeurs, la multiplication de deux nombres prend plus de temps que leur addition, ce qui viole la première supposition du modèle… Malgré tout, le *RAM* est un modèle excellent pour comprendre comment un algorithme fonctionnera sur un ordinateur réel

[Ski97]. Nous utilisons le modèle *RAM* parce qu'il est utile en pratique, et leur robustesse nous permet d'analyser des algorithmes en manière indépendante de machine [Ski97].

De plus, nous supposons que nous nous plaçons dans le cas de modèle "séquentiel". Nous supposons aussi qu'une instruction élémentaire a en général un coût uniforme, c'est-à-dire indépendant de la taille de ses opérandes (par opposition au coût logarithmique où on tient compte de la taille des données). Nous supposons en particulier que les opérations sont effectuées les unes après les autres en mémoire centrale, et les transferts des données sont considérés comme ayant un coût négligeable par rapport aux opérations.

Les algorithmes que nous considérons sont "déterministes" **:** étant donné un algorithme, toute exécution de cet algorithme sur les mêmes données donne toujours le même résultat.

Enfin, nous nous limitons en général à compter certaines instructions dites "fondamentales", par exemples : les comparaisons ou les permutations pour un tri, les opérations arithmétiques de base pour des produits de matrices… etc.

Le coût d'un algorithme se mesure essentiellement par rapport à l'utilisation des deux ressources qui sont le temps et l'espace mémoire. On parle alors de la complexité temporelle d'un algorithme pour indiquer la mesure de la durée d'exécution, et de la complexité spatiale d'un algorithme pour indiquer le volume d'espace mémoire qu'il exige. La complexité d'un algorithme est donc la double mesure des temps et espace mémoire qu'il exige pour la réalisation de ses calculs.

4.3. Complexité pratique et complexité théorique

Il est important de distinguer la complexité pratique, qui est une mesure précise des temps de calcul et des tailles de mémoire pour un modèle de machine bien défini, de la complexité théorique, qui donne des ordres de grandeur des coûts, de façon plus indépendante des conditions pratiques d'exécution.

La complexité pratique est calculée par rapport à une machine donnée, et dépendante des temps d'exécutions des instructions de base. Par conséquent, la comparaison des complexités théoriques est beaucoup plus intéressante.

4.3.1. Temps d'exécution pratique et théorique

Etant donné un algorithme *A* conçu pour résoudre un problème *P* donné, on peut construire un programme *Pg* comme étant une représentation de *A* dans le modèle de calcul induit par le langage de programmation utilisé. Le programme *Pg* est exécutable sur un ordinateur en fournissant les données nécessaires à *P*. Ce n'est pas le cas de l'algorithme, qui n'est pas exécutable. Cette exécution va prendre du temps qui dépend des facteurs telles que :

- les données de *P* spécialement fournies pour l'exécution,
- la qualité du code objet engendré par le compilateur du langage utilisé pour le programme,
- la nature et la rapidité des instructions disponibles dans le répertoire de l'ordinateur,
- l'efficacité de l'algorithme *A*,

- la qualité du programme écrit par le programmeur.
- …etc.

Remarquons que dans cette suite de facteurs, il y en a beaucoup qui ont un caractère subjectif (données spécifiques, machine particulière, habileté du programmeur…).

Une comparaison basée sur ces facteurs serait donc biaisée. On est donc tenté de considérer non plus le programme mais l'algorithme à partir duquel le programme est construit.

De cette manière, on peut comparer objectivement deux algorithmes et trouver un algorithme optimal dans la classe de ceux qui résolvent un même problème. On utilisera alors la complexité théorique qui permet de considérer une mesure qui rend compte de la qualité intrinsèque des algorithmes, indépendamment des détails d'implantation qui interviennent dans la complexité pratique.

Nous sommes amenés dans la suite à parler du temps d'exécution en termes de coûts théorique.

4.3.2. Temps d'exécution en fonction de la taille des données

Soit un algorithme *A*, il existe presque toujours un paramètre, soit *n*, caractérisant la taille des données auxquelles *A* s'applique dans chaque cas considéré, si *A* est par exemple un algorithme de tri, alors *n* est le nombre de données à trier.

Ce qui nous intéresse est de savoir comment la complexité de l'algorithme évolue en fonction de la taille des données en entrée, autrement dit, étant donnée une procédure *p*, on s'intéresse à la variation de la durée du calcul de $p(x)$ en fonction de la taille de l'argument *x* de la procédure.

Pour cela, on effectue des comparaisons sur les ordres de grandeur asymptotiques (quand la taille des données en entrée tend vers l'infini), c'est-à-dire les taux de croissance de la complexité pour des calculs de grandes dimensions.

4.3.2.1. Ordres de grandeur

Dans le contexte d'algorithmes, nous avons besoin d'un bon langage dans le but de comparer les vitesses avec lesquelles des algorithmes différents font le même travail, les quantités de mémoire qu'ils l'utilisent, ou toute autre mesure de complexité des algorithmes. Nous rappelons ici les définitions et les propriétés essentielles des notations : O, o, Ω, Θ, $\sim$ introduites pour ce langage.

Soient f et g deux fonctions de $\aleph^+$ dans $\Re^+$ (l'ensemble des nombres réels positifs), les cinq symboles suivants sont introduits pour comparer les taux de croissance de f et g. On dit que :

i) $f(x) = O(g(x))$ s'il y a des constants positifs c et x_0 telle que $|f(x)| \leq c.|g(x)|$ pour tout $x \geq x_0$.

La notation O n'implique pas l'équivalence asymptotique entre deux fonctions, elle signifie simplement qu'une fonction est asymptotiquement majorée (dominée) par une autre fonction. Par exemple, on peut écrire $f = O(g)$ (on prononce "grand O de g") pour $f(n) = n$ et $g(n) = 2^n$. Quand on dit que $f(x) = O(g(x))$ on veut dire, informellement, que f

n'augmente pas certainement d'un rythme plus vite que *g*, elle peut augmenter avec le même rythme ou plus lentement.

Exemples :

- $x^3 + 5x^2 + 71\cos x = O(x^5)$,
- $1/(1 + x^2) = O(1)$,
- $log(n) = O(n)$
- $f(n) = O(1)$ veut dire que $f(n)$ est majorée par certain constant,
- $f(n) = n^{O(1)}$ veut dire que f est majorée par certain polynôme, $n^{O(1)}$ est l'ensemble de fonctions bornées polynomialement,

ii) $f(x) = o(g(x))$ si $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$.

On prononce "*f* égale petit *o* de *g*". $o(g(n))$ est l'ensemble de fonctions qui croissent plut lentement que $g(n)$ lorsque x est plus grand, on dit que f est asymptotiquement négligeable devant g. Voici quelques exemples.

Exemples :

- $x^2 = o(x^5)$,
- $sin\ x = o(x)$,
- $14.709\ \sqrt{x} = o(x/2 + 7\cos x)$,
- $1/x = o(1)$.

"*o*" donne une information plus précise que "*O*", mais "*O*" est suffisant et plus facile à calculer.

iii) $f(x) = \Omega(g(x))$ si $f(x) \neq o(g(x))$.

On peut considérer une autre définition [GHR95] comme suit : $f(x) = \Omega(g(x))$ si et seulement s'il existe deux constants c et x_0 tel que $f(x) \geq c.g(x)$, pour tout $x \geq x_0$. On dit ainsi que $\Omega(g(x))$ est l'ensemble de fonctions dont le taux de croissance est au moins d'ordre $g(x)$.

Dans l'étude des algorithmes, Ω est utilisé lorsque on veut exprimer l'idée que certain calcul prend au moins tel et tel temps, espace…etc. Par exemple, on peut multiplier deux matrices ($n\times n$) ensemble en temps $O(n^3)$, mais on ne peut jamais faire la multiplication des deux matrice en moins de n^2 étapes de calcul, parce que tout programme qu'on écrit devra regarder les données en entrée, et il y a $2n^2$ valeurs dans les matrices. Ainsi, un temps de calcul de $c.n^2$ est certainement une borne inférieure sur la rapidité de n'importe quel programme séquentiel de multiplication possible. Par conséquent, on peut dire que le problème de multiplication de deux matrices ($n\times n$) exige un temps $\Omega(n^2)$.

iv) $f(x) = \Theta(g(x))$ si $f(x) = O(g(x))$ et $f(x) = \Omega(g(x))$.

C'est à dire que $f(x) = \Theta(g(x))$ s'il y a des constants $c_1>0$, $c_2>0$ et x_0 tel que, pour tout $x \geq x_0$, $c_1.g(x) \leq f(x) \leq c_2.g(x)$. On peut dire donc que f est asymptotiquement de même ordre de grandeur que g. Voici quelques exemples:

Exemples :

- $(x+1)^2 = \Theta(3x^2)$,
- $(x^2 + 5x + 7) / (5x^3 + 7x + 2) = \Theta(1/x)$,
- $(1 + 3/x)^x = \Theta(1)$.
- $\sqrt{3+\sqrt{2x}} = \Theta\left(x^{\frac{1}{4}}\right)$,

Θ est beaucoup plus précis que O ou o. Si par exemple $f(x) = \Theta(x^2)$, alors nous connaissons que $f(x)/x^2$ reste entre deux constants différents de zéro pour toutes les valeurs de x suffisamment grandes, et le taux de croissance de f est établi par l'expression "f augmente d'une manière quadratique avec x".

v) $f(x) = \sim(g(x))$ si $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$.

C'est le symbole le plus précis, il nous dit que non seulement f et g augmentent avec le même taux, mais que, en fait, f / g s'approche de 1 lorsque x tend vers ∞, et on dit donc que f est asymptotiquement équivalente à g.

Exemples :

- $(x^2 + x) = \sim(x^2)$,
- $(3x + 1)^4 = \sim(81x^4)$,
- $\sin 1/x = \sim(1/x)$,
- $(2x^3 + 5x + 7)/(x^2 + 4) = \sim(2x)$,
- $(2^x + 7 \log x + \cos x) = \sim(2^x)$.

∨ *Remarque*

Il faut garder à l'esprit que, étant donnée une fonction g, les notations $o(g)$, $O(g)$, $\Omega(g)$, $\Theta(g)$ et $\sim(g)$ représentent respectivement les classes des fonctions suivantes :

$o(g) = \{f \ / \ \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$ (f est asymptotiquement négligeable devant g)$\}$,

$O(g) = \{f \ / \exists\, c, x_0 \ \forall\, x \geq x_0 : |f(x)| \leq c.|g(x)|$ (f est asymptotiquement majorée par g)$\}$,

$\Omega(g) = \{f \ / \ f \neq o(g)\}$,

$\Theta(g) = \{f \ / \ f = O(g)$ et $f = \Omega(g)\}$,

$\sim(g) = \{f \ / \ \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$ (f est asymptotiquement équivalente à g)$\}$.

Dans ce cadre, l'écriture, par exemple, $f = O(g)$ signifie en fait que $f \in O(g)$. (Sinon on aurait des absurdités telle que : $n^2+n = O(n^3)$ et $2n^2 = O(n^3) \Rightarrow n^2+n = 2n^2$!!).

- ***Echelle de comparaison***

L'ordre de grandeur permet de faire ressortir l'essentiel de la fonction de complexité *f*. Cette fonction est typiquement de forme :

- *constante* : si $f = O(1)$,
- *logarithmique* : si $f = O(\log n)$,
- *linéaire* : si $f = O(n)$,
- *quadratique* : si $f = O(n^2)$,
- *polynomiale* : si $f = O(n^k)$, pour certain entier k,
- *exponentielle* : s'il existe un constant $k > 0$ telle que $f = O(k^n)$.

Par exemple, un algorithme pourra être de complexité théorique $O(n^2)$, $O(2^n)$, $O(n \log n)$ ...etc. Si une opération s'effectue en un temps fixe ne dépendant pas de la taille du problème, on dira qu'elle est de complexité $O(1)$ (constante).

De plus, pour mesurer la complexité d'un algorithme on utilise évidemment le plus petit des "grands *O*" possibles, par exemples : n est $O(n^2)$ mais il est aussi $O(n)$; 12 est $O(n^2)$, $O(n)$, mais surtout $O(1)$…

La théorie de la complexité ne s'intéresse qu'à l'ordre de grandeur de la complexité, sans se préoccuper des constantes intervenant dans son expression, elle néglige les constantes de proportionnalité pour ne retenir que les comportements asymptotiques. Par exemple les fonctions (n^2), $(10^{75}n^2)$, $(1000n^2+ 10^{75}n+10^{50})$ et $(n^2/10^{75} + \log n)$ seront toutes considérées comme étant $O(n^2)$. Avec cette démarche on ne perd pas aucune information essentielle sur la complexité inhérente [Lok03], ceci est justifié par le théorème suivant, dit "théorème d'accélération" (*Speedup Theorem*), qui énonce qu'on peut toujours accélérer une machine d'un facteur constant à condition que le temps de calcul est au moins proportionnel à la taille de l'entrée ($n = O(T_M(n))$) [Car06] : Soit $k > 0$ un entier et soit M une machine de Turing, il existe une machine de Turing M ', équivalente à M telle que $T_{M'}(n) \leq T_M(n)/k$.

Théorème 4.1 (Speedup Theorem) [Lok03] Pour tout $\varepsilon > 0$, si $L \in$ TIME(T(n)), alors $L \in$ TIME(εT(n) + n + c), où c est un constant absolu.

Ainsi, une machine de Turing peut être transformée en une autre machine qui fonctionne plus rapidement par un facteur constant, L'analogue de ce théorème pour l'espace est appelé "théorème de compression" (*Tape Compression Theorem*).

Théorème 4.2 (Tape Compression Theorem) [Lok03] Pour tout $\varepsilon > 0$, si $L \in$ SPACE(S(n)), alors $L \in$ SPACE($\varepsilon S(n)+c$) pour certain constant absolu c.

Le théorème d'accélération est la raison de pourquoi la théorie de la complexité ignore les constantes multiplicatives. Ainsi, les complexités des algorithmes sont exprimées en utilisant la notation d'ordre de grandeur.

4.3.2.2. Envisager le pire

Dans la pratique, le temps d'exécution d'un algorithme dépend non seulement de la taille de l'ensemble de données, mais aussi de leurs valeurs précises (la configuration des données). En fait, pour deux données différentes de même taille, l'exécution d'un algorithme peut utiliser des quantités de ressources différentes. Par exemple, pour la recherche séquentielle d'un élément dans une liste de n éléments, si l'élément se trouve en première position, on a tout de suite fini, s'il n'est pas présent dans la liste, il faut parcourir toute la liste pour s'en apercevoir.

Pour tenir compte de ce fait, tout en conservant un moyen d'analyser les algorithmes indépendamment de leurs données, on utilise une approche expérimentale où on distingue trois manières de mesurer la complexité : la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas:

- La complexité "minimale" T_{min}, ou complexité du cas le plus favorable, c'est le cas où on choisit les données entraînant l'exécution la plus courte.
- La complexité "moyenne" qui est celle de la fonction T_{moy}, temps moyen d'exécution de l'algorithme appliqué à n données quelconques. Il convient de noter que $T_{moy}(n)$ n'a de sens que si l'on peut faire des hypothèses probabilistes sur la répartition des données, ce qui est loin d'être toujours le cas. Par exemple, on suppose souvent que toutes les données de même taille sont équiprobables, ce qui est peu réaliste.
- La complexité "maximale", ou complexité du cas le plus défavorable, qui est la complexité de la fonction T_{max}, où $T_{max}(n)$ est le temps d'exécution de l'algorithme lorsque l'on choisit les n données entraînant l'exécution la plus longue.

Par exemple, pour la recherche séquentielle dans une table de n éléments, on fait une seule comparaison au minimum, $n/2$ comparaisons en moyenne et n comparaisons dans le pire des cas.

Ces notions s'étendent sans difficulté à la mesure du coût d'un algorithme en espace mémoire: on parle de la complexité spatiale minimale, maximale ou moyenne [Att01].

Dans ce qui suit, nous nous intéressons seulement à la troisième définition : la complexité temporelle $T(n)$ et la complexité spatiale $S(n)$ d'un algorithme sont respectivement le temps de calcul et l'espace mémoire maximaux pour une donnée de taille n. La complexité dans le meilleur des cas n'est pas très utilisée. La complexité en moyenne est d'une certaine façon celle qui révèle le mieux le comportement "réel" de l'algorithme à condition de disposer d'un modèle de répartition des données, mais, bien sûr, elle ne garantit rien sur le pire des cas, et elle est souvent dure à calculer, même de façon approximative !

4.4. Calcul de la complexité temporelle

Considérons de nouveau (indépendamment du modèle de calcul choisi) une fonction $T_A(n)$ qui exprime, en fonction de la taille des données n, le maximal d'un certain coût pour un algorithme A donné. Ce coût dépend des opérations effectuées dans l'algorithme.

4.4.1 Opérations fondamentales

C'est le concept de classe d'algorithmes (pour résoudre un problème) qui met en évidence les opérations: chaque classe est caractérisée par un ensemble d'opérations dites "fondamentales" qui y sont utilisées, ces sont les opérations dont dépend le coût. Ainsi, le temps d'exécution d'un algorithme est toujours proportionnel dans le nombre de ses opérations fondamentales [Fro05]. On détermine la complexité intrinsèque de l'algorithme en comptant le nombre d'opérations fondamentales. Ces opérations sont toujours choisies parmi les opérations élémentaires utilisées par l'algorithme pour effectuer les calculs. En effet, leur nature peut être différente en fonction du problème traité.

Par exemple, pour la recherche d'un élément dans une liste, l'opération fondamentale est la comparaison entre cet élément et les éléments de la liste, dans le cas de parcours d'un arbre binaire, l'opération fondamentale est la visite d'un nœud, …etc. Il est également possible de choisir plusieurs opérations fondamentales et de les pondérer, par exemple pour la multiplication de deux matrices carrées les opérations fondamentales sont la multiplication et l'addition.

4.4.2 Hypothèse du coût uniforme

Avec l'hypothèse du "coût uniforme", toute opération élémentaire prend un temps constant (de l'ordre de $O(1)$). Dans la pratique, l'hypothèse de coût uniforme n'est acceptable que si toutes les données rencontrées au cours de calcul peuvent être codées sur un même nombre de bits, donc bornés [Att01].

En considérant ces modalités pour la taille, les opérations et leur coût, la complexité d'un algorithme A est donc proportionnelle au nombre maximal d'opérations fondamentales effectuées au cours de calcul.

4.4.3 Règles principales de comptabilisation des opérations fondamentales

Après la détermination des opérations fondamentales pour l'algorithme considéré, nous pouvons calculer sa complexité en termes de nombre de ces opérations. Cependant, il n'existe pas de méthodologie systématique permettant pour un algorithme quelconque de compter ses opérations fondamentales. Toutefois, des règles usuelles sont communément admises par la communauté des informaticiens qui les utilisent pour évaluer la complexité temporelle.

Nous ne considérons ici que le cas des algorithmes déterministes séquentiels. Un algorithme séquentiel a une structure qui met en évidence une méthode inductive de dénombrement des opérations effectuées. Cette structure dépend des structures de contrôle utilisées.

- ***La séquence***

Le coût d'une séquence d'instructions est la somme des coûts de chacune d'elles. Etant donnée une séquence *S* de *n* actions : a_1, a_2, …, a_n, alors le coût en temps d'exécution de la séquence *S* est :

$$coût_S = \sum_{i=1}^{n} T_{a_i}$$

où T_{a_i} représente le temps d'exécution de l'action a_i.

- ***La conditionnelle***

Dans les instructions conditionnelles, étant donné qu'il n'est pas possible, d'une manière générale, de déterminer systématiquement quelle partie de l'instruction est exécutée (le *alors* ou le *sinon*), on prend donc un majorant. Soit la conditionnelle :

```
si C
     alors A
     sinon B
finsi
```

Le temps d'exécution de la conditionnelle, avec *B* est éventuellement vide, est :

$$coût_{Si} = coût\ (C) + Max\ \{coût\ (A), coût\ (B)\}$$

- ***L'itération***

Soit l'itération (finie) suivante :

```
tantque C faire
     A
finfaire
```

Le temps d'exécution de telle boucle se calcule de la façon suivante :

$$coût_{Tantque} = m*coût\ (C) + m*coût\ (A) + coût\ (non\ C)$$

Où *m* est le nombre de passages dans le corps de la boucle *tantque*. Une des difficultés principales (à part dans les cas triviaux) est de trouver combien de fois s'exécute la boucle.

Pour les boucles *répéter* et *pour*, le temps d'exécution est déduit similairement.

- ***Les procédures et les fonctions***

Pour les appels des procédures ou des fonctions, s'il n'y a pas de récursivité, ni directe ni croisée, alors on évalue la complexité en utilisant les règles ci-dessus avec l'évaluation préalable de la complexité des procédures internes. On évalue d'abord la complexité des fonctions et des procédures qui ne contiennent pas d'appels à d'autres fonctions ou procédures, puis celles qui contiennent des appels aux précédentes, et ainsi de suite.

En cas de récursivité, il faut recourir à des procédés mathématiques plus sophistiqués, du type résolutions d'équations de récurrence.

- ***Exemple*** : Considérons le cas de calcul de *n*!, pour lequel nous avons la fonction récursive suivante ($n \geq 0$) :

```
Fonction fact(n : entier) : entier
Debut
   si(n=0) alors
        return(1)
   else
        return(n*fact(n-1))
Fin
```

Si nous nous intéressons au temps d'exécution, nous obtenons l'équation de récurrence suivante (où *a* et *b* sont des constantes, et T(*i*), $0 \leq i \leq n$, est le coût de la fonction appelée avec l'argument *i*) :

$T(0) = a$,

$T(n) = T(n\text{-}1) + b$.

Nous pouvons, dans ce cas, utiliser simplement des substitutions successives pour résoudre cette équation:

{1} $T(n) = \mathbf{T(n\text{-}1)} + b$

{2} $\mathbf{T(n\text{-}1)} = \mathbf{T(n\text{-}2)} + b$

{3} $\mathbf{T(n\text{-}2)} = \mathbf{T(n\text{-}3)} + b$

……

{*k*} $\mathbf{T(n\text{-}k+1)} = \mathbf{T(n\text{-}k)} + b$

……

{*n*-1} $\mathbf{T(2)} = \mathbf{T(1)} + b$

{*n*} $\mathbf{T(1)} = a + b$

Nous obtenons, par des substitutions successives des termes en gras, le résultat suivant (qui peut être prouvé par récurrence) :

$$T(n) = a + n.b,$$

Donc : $$T(n) = \Theta(n).$$

Le cas où il y a plusieurs appels est souvent plus difficile, c'est-à-dire les équations de récurrence de la forme : $T(n) = a_1T(n\text{-}1) + a_2T(n\text{-}2) + \ldots + a_kT(n\text{-}k) + b$, où a_i ($1 \leq i \leq k$) et *b* sont des constants. La résolution de telles équations nécessite des procédés mathématiques plus sophistiqués, nous discutons ici le cas d'équations de récurrence homogènes.

Soit l'équation de récurrence linéaire homogène (d'ordre k) suivante, avec des coefficients constants :

$$a_0.T_n + a_1.T_{n-1} + ... + a_k.T_{n-k} = 0$$

On appelle "équation caractéristique" l'équation suivante :

$$a_0.x^k + a_1.\ x^{k-1} + ... + a_k.\ x^0 = 0$$

- Si l'équation caractéristique a k racines distinctes : r_1, r_2,… r_k, alors les solutions uniques de la récurrence sont :

$$T_n = c_1.r_1^{\ n} + c_2.r_2^{\ n} + \ldots + c_k.r_k^{\ n}$$

- ***Exemple*** : soit l'équation de récurrence suivante : $T_n = 5.T_{n-1} - 6.T_{n-2}$, avec $T_0 = 0$ et $T_1=1$.

L'équation caractéristique est : ($x^2 - 5x + 6 = 0$), dont les racines sont : $r_1=3$ et $r_2=2$.

La solution générale de l'équation de récurrence est de la forme :

$$T_n = c_1.3^n + c_2.2^{\ n}$$

Nous utilisons les conditions initiales (T_0 et T_1) pour déterminer les valeurs des constants c_1 et c_2, en trouvant : $T_n=3^n - 2^{\ n}$

- Si r est une racine multiple (de multiplicité m) pour l'équation caractéristique, alors : $T_n=r^n$, $T_n = n.r^n$, $T_n = n^2.r^n$, … $T_n = n^{m-1}.r^n$, sont des solutions possibles pour l'équation de récurrence. Par conséquent, un terme de chacune d'elles doit être inclus dans la solution générale.

- ***Exemple*** : Soit l'équation de récurrence suivante : $T_n=7T_{n-1} -15T_{n-2} + 9T_{n-3}$, avec $T_0=0$, $T_1=1$ et $T_2=2$.

L'équation caractéristique est donnée par : $x^3 - 7x^2 + 15x - 9 = 0$, qui peut être écrite :

$(x-1)\ (x-3)^2=0$

Donc les racines sont : $r_1=1$ et $r_2= r_3=3$. La solution générale est ainsi de la forme :

$$T_n = c_1.1^n + c_2.3^{\ n} + c_3.n.3^{\ n}$$

et les conditions initiales (T_0, T_1 et T_2) nous donnent la solution suivante :

$$T_n = -1 + 3^n - (1/3).n.3^{\ n}$$

- Le cas des équations non homogènes est un peu plus compliqué, mais elles peuvent être souvent converties à des équations homogènes. Considérons par exemple l'algorithme précédent pour le calcul du factoriel dont l'équation de récurrence est la suivante :

$T(0) = a$,

$T(n) = T(n-1) + b$,

alors $T(n-1) = T(n-2) + b$.

Par soustraction de ces deux dernières équations nous obtenons : $T(n) = 2T(n-1) - T(n-2)$.

L'équation homogène associée est : $T(n) - 2T(n-1) + T(n-2) = 0$.

L'équation caractéristique est donc : $x^2 - 2x + 1 = 0. \Leftrightarrow (x-1)(x-1) = 0$, qui a une racine multiple: $r = 1$.

La solution générale est donc : $T(n) = c_1.r^n + c_2.n.r^n$, avec :

$T(0) = c_1 = a$,

$T(1) = c_1.1^n + c_2.1.1^n = a + b, \Rightarrow c_2 = b$.

Et nous obtenons enfin le même résultat obtenu intuitivement :

$$T(n) = a + b.n = \Theta(n).$$

- ***Remarque*** :

Lorsqu'on fait un appel, il faut en toute rigueur compter l'appel lui-même comme une opération particulière, mais aussi compter les opérations correspondant au passage des arguments. Plus précisément, considérons les deux types de passage des arguments: lors d'un passage par valeur, pour chaque argument passé il faut évaluer l'argument, et affecter sa valeur à une nouvelle variable (locale à la fonction). Cependant, si l'argument passé est par exemple un tableau de taille *n*, alors l'affectation correspond à *n* affectations élémentaires, ce coût n'est plus négligeable, et doit être pris en considération lors de calcul de la complexité temporelle. C'est en particulier une des raisons pour lesquelles on évite souvent de passer par valeur un tableau. Par contre, un passage par adresse ne correspond pas à *n* affectations élémentaires puisque seule l'adresse en mémoire du tableau est fournie à la procédure lors de l'appel. La même remarque est aussi vraie pour le calcul de la complexité spatiale. De plus, la situation devient plus compliquée dans le cas des méthodes récursives, soit pour la complexité temporelle ou pour la complexité spatiale.

4.5. Calcul de la complexité spatiale

Il est quelquefois nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire (par exemple tableau auxiliaire de même taille que le tableau donné en entrée). L'analyse de la complexité en place mémoire revient à évaluer, en fonction de la taille des données, la place mémoire nécessaire pour l'exécution de l'algorithme, en dehors de l'espace occupé par les instructions du programme et les données. La complexité en mémoire est donc la fonction $S(n)$ qui mesure l'espace mémoire exigée par l'algorithme pour effectuer les calculs sur des données de taille *n*.

La complexité spatiale dépend de la manière dont les entrées sont codées... Or, il peut y avoir des codages plus ou moins efficaces! Lorsqu'un algorithme est décrit de façon moins précise, il faut souvent préciser séparément comment sont codées les données.

Le cas compliqué, pour le calcul de la complexité spatiale, est le cas d'utilisation de fonctions récursives. Un algorithme qui fait appel à une fonction récursive demande beaucoup de mémoire. Souvenons-nous que chaque appel de fonction demande de créer une zone dans la pile et d'y enregistrer différentes valeurs. Donc, à chaque appel de la fonction on alloue un espace mémoire pour les arguments et pour les variables locales. En effet, pour calculer la complexité spatiale, dans le cas des fonctions récursives, nous avons besoin de connaître le nombre maximal d'appels de la fonction et l'espace mémoire nécessaire pour chaque appel.

4.6. Conclusion

Comme la résolution algorithmique elle-même, L'analyse de la complexité d'un algorithme donné n'est pas toujours simple. Nous suivons une approche générale pour ce faire.

La complexité d'un algorithme (déterministe séquentiel) est souvent mesurée sous la forme des taux de croissance du temps et de la quantité de mémoire requis pour réaliser des calculs de dimension de plus en plus grande. La dimension d'un calcul se représente par un entier *n*, en général, elle est reliée à la taille des données traitées. La complexité temporelle T(*n*) d'un algorithme est le nombre maximal d'étapes (opérations de base de durée fixe) impliqués par l'algorithme, on définit aussi la complexité spatiale S(*n*) comme le nombre maximal de mots de mémoire utilisés dans les calculs. Il s'agit donc d'obtenir le meilleur compromis entre l'espace et le temps.

La limite de la complexité (temporelle ou spatiale) lorsque le calcul croît en dimension est la complexité asymptotique (on néglige alors la phase de distribution des données au démarrage et de collecte des résultats à la fin, ce qui n'est pas toujours réaliste dans le cas du calcul parallèle). La complexité asymptotique peut servir à comparer deux algorithmes différents pour un même problème et à distinguer les algorithmes efficaces des algorithmes inefficaces. Pour ce faire, on utilise en général l'ordre de grandeur des complexités (leurs taux de croissance pour des calculs de grandes dimensions).

La complexité computationnelle avec la notion "*O*" mesure le nombre d'opérations élémentaires nécessaires dans le pire des cas pour manipuler une entrée de taille *n*. La complexité avec la notion "*O*" tient compte des étapes logiques de l'algorithme du programme plutôt que du nombre exact d'opérations dans le programme. Elle peut être donc calculée dès la phase de conception de l'algorithme, elle est aussi indépendante de la machine et du langage de programmation. Ce coût théorique permet de donner la complexité d'un algorithme indépendamment de tout critère matériel. En contrepartie, le coût pratique est lié aux caractéristiques d'une machine donnée, ce coût ne peut être comparé qu'avec d'autres coûts effectués dans les mêmes conditions.

D'autre part, déterminer la complexité d'un problème apparaît beaucoup plus difficile, 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 sur tous ceux que nous n'imaginons même pas…

Bien que les deux groupes de problèmes faciles et de problèmes difficiles soient distincts, il n'est pas toujours simple de rattacher un problème donné à l'un des deux groupes (par exemple le cas du problème des nombres premiers). En effet, il faut exhiber un algorithme de coût polynomial ou démontrer l'inexistence d'un tel algorithme.

Chapitre 5

Complexité computationnelle d'algorithmes de construction de STEMs réduits

Ce chapitre présente nos résultats de l'étude de la complexité computationnelle asymptotique des algorithmes proposés dans [Ben06] en vue de la résolution de problème de l'explosion combinatoire du graphe d'états dans la vérification formelle des systèmes complexes.

Avant de présenter le problème considéré et les méthodes et algorithmes proposés, nous rappelons les trois points essentiels pour juger la qualité d'une technique de réduction donnée, qui sont :

- *Taux de réduction fort* : le rapport taille de graphe complet et taille de graphe réduit doit être le plus grand possible.
- *Capacité d'analyse forte* : signifie que l'ensemble de propriétés satisfaites dans le graphe complet et non vérifiables dans le graphe réduit est le plus petit possible.
- *Simplicité de réalisation* : deux critères doivent être pris en compte : le temps et l'espace mémoire nécessaires à la construction du graphe réduit par rapport à ceux nécessaires à la construction du graphe complet.

Les deux premiers points sont jugés expérimentalement pour les méthodes proposées dans [Ben06] sous la forme de comparaisons avec d'autres méthodes du même intérêt, mais leur complexité, en termes de temps d'exécution et espace mémoire, nécessite une étude plus détaillée.

Le but du présent chapitre est d'évaluer la complexité des algorithmes proposés dans [Ben06]. Dans un premier temps, nous allons présenter brièvement le problème de l'explosion combinatoire du graphe d'états dans le domaine de la vérification formelle, et les approches de résolution courantes. Par la suite nous allons présenter (informellement) les méthodes et les algorithmes proposés pour avoir une idée sur le raisonnement et le principe suivis. Les définitions formelles détaillées pour les différentes notions utilisées ici sont expliquées dans [Ben06]. Nous allons ensuite évaluer les

complexités des différentes fonctions utilisées dans les algorithmes pour déduire enfin la complexité des algorithmes généraux.

5.1. Problème de l'explosion combinatoire et réduction du graphe d'états

Durant la dernière décennie, les méthodes formelles sont devenues une activité indispensable intégrée au processus de conception des systèmes complexes à caractère critique, tels que les protocoles de télécommunication, les systèmes répartis et les architectures multiprocesseurs. En effet, la complexité de ces systèmes rend leur analyse classique (prototypage et test) extrêmement ardue, leur fiabilité ne sera garantie autrement qu'en employant des méthodes de spécification et de vérification formelle, assistées par des outils informatiques performants. Une approche de vérification offrant un bon compromis coût performances est la vérification basée sur les modèles (model-checking).

Cette approche consiste à traduire le système, préalablement écrit dans un langage approprié (description du système), en un modèle (sémantique), généralement un graphe d'états, sur lequel les propriétés de correction attendues (spécification) sont vérifiées au moyen d'algorithmes spécifiques (figure 5.1). Bien que limitée aux applications ayant un nombre fini d'états, la présente approche est particulièrement utile dans les premières phases du processus de conception, permettant une détection rapide et économique des erreurs.

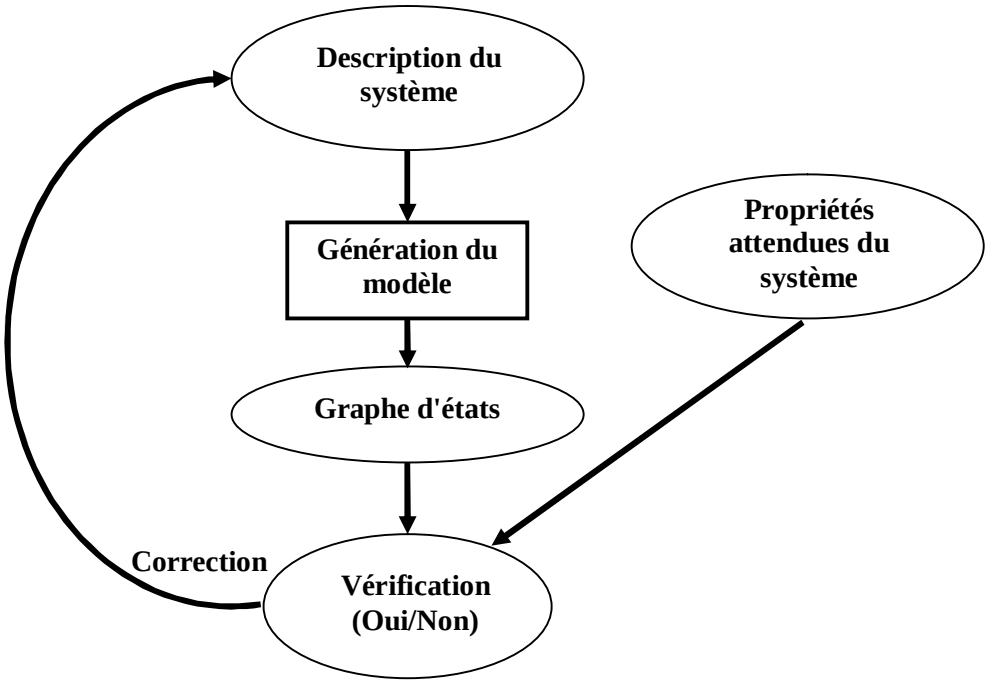


Figure 5. 1 : Vérification formelle

La vérification formelle est fondée précisément sur les quatre éléments suivants :

- *Un langage de description formelle du système* : C'est un formalisme de comportement des systèmes de haut niveau, il possède à la fois une syntaxe et une

sémantique bien définie (dite formelle). Parmi ces langages, nous pouvons citer les réseaux de Petri (RdP), les algèbres de processus (ACP, CCS, CSP, …) et les techniques de description formelle (ESTELLE, LOTOS, SDL).

- *Un modèle sémantique du parallélisme* (modèle de bas niveau) : Comme son nom l'indique, ce modèle est utilisé pour exprimer la sémantique du parallélisme des langages de description formelle. Nous distinguons deux grandes familles : les modèles d'entrelacement (Arbres de synchronisation, STE,…) et les modèles de non entrelacement (dits de vrai parallélisme) (RdP, SEP, STA,…). Leur principale différence réside dans l'adoption ou non de l'hypothèse de l'atomicité temporelle et structurelle des actions (les actions sont de durée nulle et indivisibles). Les modèles de vrai parallélisme nous permettent d'une part d'éviter les difficultés liées aux modèles entrelacés, et d'une autre part ils nous permettent d'introduire une méthode de conception formelle par raffinement successif (remplacer une action par un processus).
- *Un langage de spécification* : C'est un formalisme dédié à la description des propriétés attendues du système. Dans la littérature, nous trouvons deux types de langages de spécification : spécification logique et spécification comportementale. La spécification logique (HML, CTL, CTL*, …) est généralement fondée sur des formules d'une logique temporelle modale qui sont interprétées sur le modèle sémantique sous-jacent. Cependant les propriétés du deuxième type sont exprimées dans le même modèle sémantique de celui de la description du système.
- *La vérification* : C'est une relation de satisfaction qui définit la comparaison entre la description du système et sa spécification. Selon le formalisme de la spécification employé, il existe généralement deux types de vérification : le premier type (dit vérification logique) consiste à vérifier les formules logiques sur le graphe d'états du système, et le deuxième type (dit vérification comportementale) consiste en une comparaison entre les modèles sémantiques, elle s'effectue au moyen d'une relation d'équivalence ou de préordre.

Utiliser un modèle exhaustif pour exprimer le comportement du système entraîne deux conséquences majeures sur les techniques de vérification formelle, la première est considérée comme un avantage : avoir une technique de vérification complètement automatisable est devenue possible. Cependant, la deuxième conséquence est une forte limitation d'ordre opérationnel, effectivement, dans la majorité des cas le modèle est infini ou même trop gros pour être manipulé, il s'accroît très rapidement (accroissement exponentiel) avec la taille du système. Ce phénomène est connu sous le nom de "l'explosion combinatoire" du graphe d'états. Quand l'espace des états du système est fini, il est théoriquement possible de l'explorer en totalité dans le but de vérifier des propriétés sur le système. Mais dans la pratique, ce n'est souvent pas le cas, car l'espace des états est fréquemment beaucoup trop grand pour être exploré exhaustivement. Donc, si l'on veut que l'approche par model-checking soit possible, il faut trouver une méthode permettant de vérifier un système tout en ne construisant qu'un sous-ensemble de son espace d'états.

De multiples travaux sont en cours en vue de maîtriser cette explosion, il existe en général deux stratégies complémentaires. La première consiste à contourner la difficulté de traiter les systèmes: "gérer l'explosion", et la deuxième consiste à réduire la complexité du système : "combattre l'explosion".

La gestion de l'explosion cherche sur les effets du phénomène où les limites physiques de la mémoire sont vite atteintes. Parmi les approches introduites dans cet axe de recherche, la compression du graphe de comportement est la technique la plus connue, elle est basée sur la notion de BDD (*Binary Decision Diagram*), le taux de compression dépend de certains paramètres, et il peut être très faible pour certains graphes. Sur le même axe, on remarque une approche efficace dans le cas où la propriété à vérifier n'est pas satisfaite par le système, la vérification des propriétés ici se fait au cours de la construction du modèle, dite "vérification à la volée", et la non satisfaction des propriétés conduit directement à l'arrêt de la construction du modèle.

D'autre part les variables du système, les composants symétriques et la sémantique d'entrelacement sont jugés comme les racines du problème. Pour combattre ce phénomène, il faut éliminer les effets de chaque cause, toutes les techniques proposées dans cette approche sont basées sur la construction incomplète du modèle dont la partie construite doit être pertinente au vu des propriétés à vérifier.

Le travail de [Ben06] s'inscrit dans le cadre de la résolution du problème de l'explosion combinatoire du graphe d'états, en s'intéressant plus particulièrement à l'explosion due à la représentation du parallélisme par l'entrelacement d'actions concurrentes, ce qui génère plusieurs séquences d'exécution commençant d'un même état et finissant dans un autre état, où l'ordre d'exécution est arbitraire. Les techniques d'ordre partiel cherchent à éliminer les entrelacements superflus en se basant sur les relations d'indépendance calculées directement à partir de la spécification formelle du système à analyser. On distingue en général deux stratégies : élimination de l'entrelacement et pas couvrants.

Les différentes techniques de la première stratégie tentent à obtenir un sous-graphe du graphe de comportement, contenant le moins de séquences équivalentes possibles, leur principal inconvénient est l'indéterminisme du résultat obtenu où on peut avoir plusieurs sous-graphes pour un même graphe de comportement (figure 5.2.(b)). La deuxième alternative consiste à regrouper les événements indépendants dans un seul pas (figure 5.2.(c)), le graphe construit est le graphe de pas couvrants, qui est un graphe complet, où la préservation des états de blocage et des propriétés de vivacité est assurée.

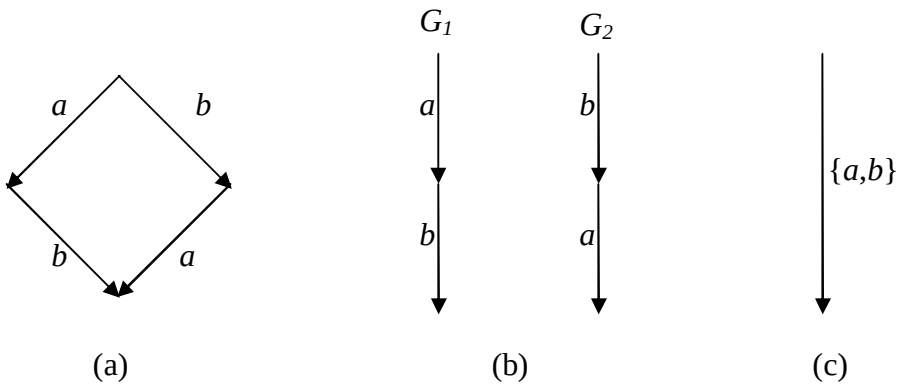


Figure 5. 2 : Le comportement de l'expression $a|||b$

5.2. Présentation de méthodes et algorithmes proposés

La contribution de [Ben06] consiste à proposer deux méthodes de réduction : la réduction basée sur l'approche d'ordre partiel et la réduction basée sur une relation d'équivalence (la α-équivalence), en donnant l'algorithme (et l'implémentation) de construction à la volée de graphe réduit pour chacune de ces deux méthodes. Mais avant d'exposer ces deux méthodes, nous présenterons brièvement le modèle sur lequel elles s'appliquent : c'est le *Système de Transitions Etiquetées Maximales* (STEM) [Sai96].

Le modèle des STEMs peut être utilisé comme une représentation sémantique de comportement du système étudié, d'où la possibilité d'utiliser différents modèles de spécification; pour cela, il suffit de définir la sémantique en termes de STEMs pour chacun de ces modèles. Prenons par exemple le STEM de la figure 5.3(c) qui représente le comportement de réseau de Petri de la figure 5.3(a). Dans l'état initial, aucune action n'a encore commencée son exécution. La transition t_1 (resp. t_2) représente le début d'exécution, identifié par l'événement *x* (resp. *y*) de l'action *a* (resp. *b*). À l'état 1 (resp. 2) l'action *a* (resp. *b*) est potentiellement en cours d'exécution, ceci est représenté respectivement par les événements (dits maximaux) *x* et *y*. Dans l'état 2, l'occurrence de *c* est conditionnée par la terminaison de *b*, ce qui est traduit par la présence de l'événement *y* au niveau de la transition t_5, donc *z* est le seul événement maximal dans l'état 4. Dans l'état 3, les événements *x* et *y* sont maximaux, c'est-à-dire que les actions correspondantes (*a* et *b*) peuvent s'exécuter simultanément (en parallèle).

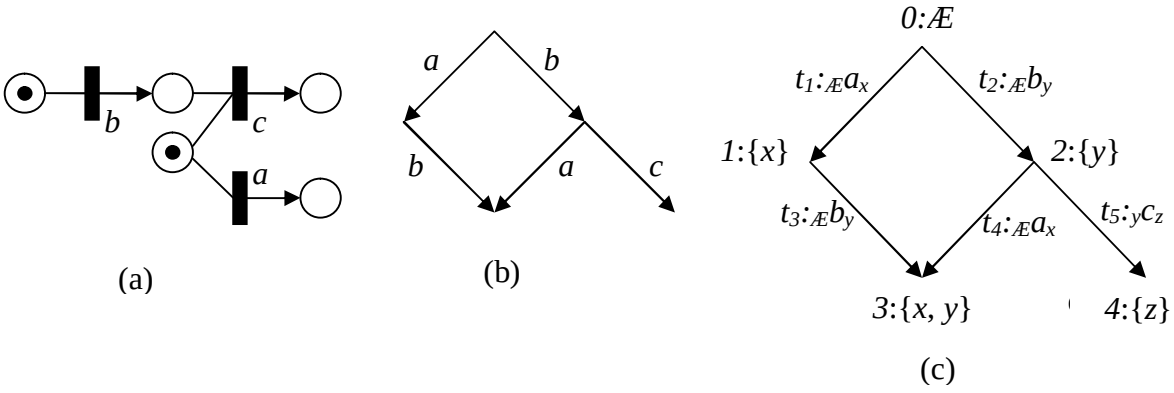


Figure 5. 3 : Exemple de STEM (Le conflit différé)

5.2.1. Le STEM a –réduit et la méthode de réduction basée sur la a –équivalence

Cette méthode consiste à éliminer les redondances modulo une relation d'équivalence spécifique au modèle des STEMs, connue sous le nom de la α-équivalence [Ben06]. Cette relation permet de regrouper des STEMs qui décrivent le même comportement dont la seule différence réside dans le choix des noms des événements. Par exemple, les deux STEMs de la figure 5.4 décrivent le même comportement (l'exécution parallèle de *a* et *b*), nous pouvons obtenir le STEM de la figure 5.4(a) à partir de celui de la figure 5.4(b) (modulo l'isomorphisme ≅) en substituant les noms des événements *e* (resp. *z*) par *x* (resp. *y*). On dit que ces deux STEMs sont α-équivalents.

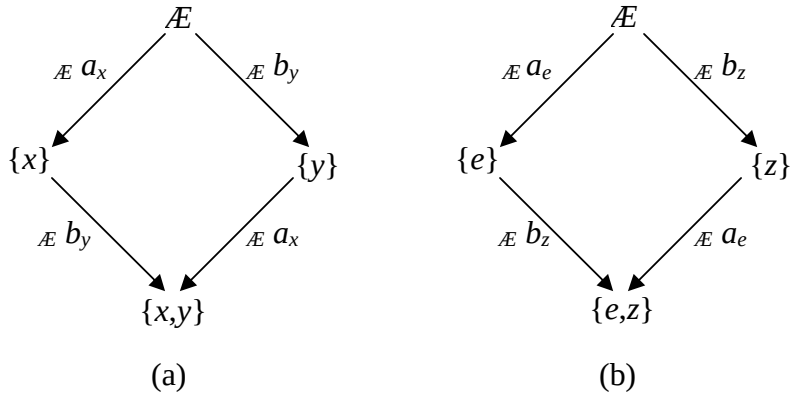


Figure 5. 4 : Deux STEMs α-équivalents

- ***Idée de base***

La réduction consiste à éliminer les redondances modulo certaine relation en préservant les propriétés à vérifier. La relation utilisée dans cette méthode comme critère de redondance est la α-équivalence. À titre d'illustration, le STEM de la figure 5.5(a) représente le comportement de l'expression "*a; d; stop* |[*d*]| *b; d; stop*" (composition parallèle avec synchronisation sur l'action *d*), il a été généré par l'application directe des règles de la sémantique opérationnelle de maximalité [Sai96]. Les deux sous-STEMs S_1 et S_2 de la figure 5.5(a) sont α-équivalents. En fait, il existe deux fonctions de substitution : $\sigma_1 = \{x/x, y/y, z/z\}$ et $\sigma_2=\{x/v, y/u, z/e\}$ tel que $S_1\sigma_1 \cong S_2\sigma_2$, ($\cong$ est l'isomorphisme des graphes). Pour supprimer une telle redondance, on doit, en premier lieu, substituer le STEM de la figure 5.5(a) par $\sigma_1 \cup \sigma_2$, et supprimer, en suite, $S_1\sigma_1$ ou $S_2\sigma_2$. Le STEM de 5.5(b) est le STEM α-réduit ($STEM_{/\approx\alpha}$) construit.

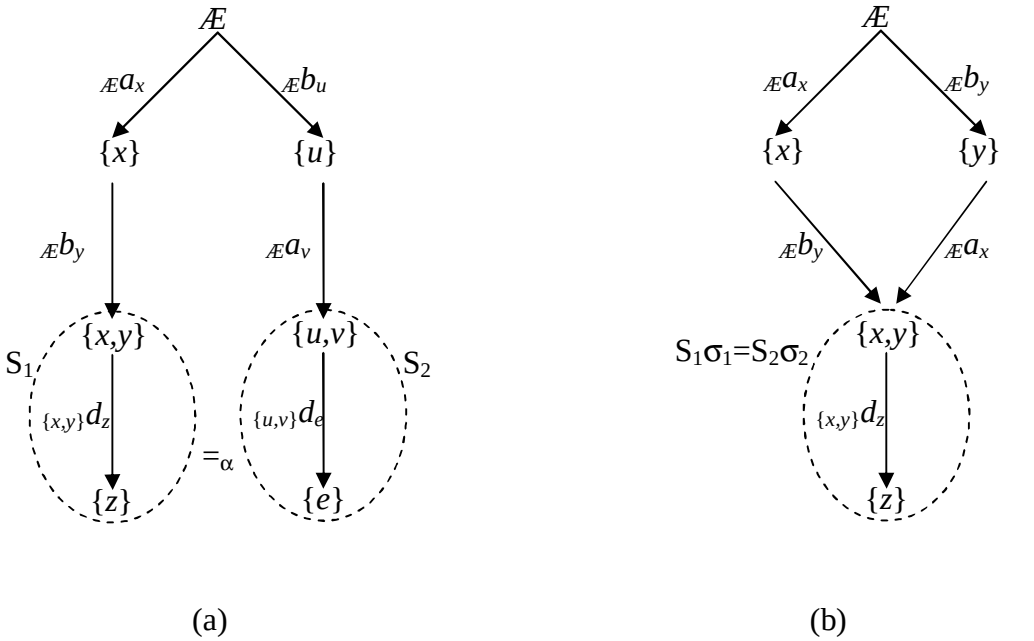


Figure 5. 5 : Réduction modulo α-équivalence

Cet exemple est appliqué à un STEM déjà généré, cependant, le but est la génération à la volée des STEMs α-réduits. L'alternative consiste à vérifier pour chaque configuration générée si elle est α-équivalente ou non avec une autre configuration générée auparavant, si c'est le cas, on supprime l'une des deux en substituant le STEM en construction.

- ***Algorithme de construction à la volée d'un STEM$_{/\approx\alpha}$***

L'Algorithme 01 nous permet de générer, à partir d'une expression de comportement LOTOS donnée, un STEM α-réduit. Le calcul est basé sur les configurations, où pour chaque nouvelle configuration, nous tirons de nouvelles transitions et de nouvelles configurations, en utilisant les règles de la sémantique opérationnelle de maximalité [Sai96].

Une configuration est dite nouvelle si elle n'est α-équivalente à aucune autre configuration générée précédemment.

Algorithme 01 *"Algorithme de construction à la volée de STEM$_{/\approx\alpha}$"*

Données *: Spécification LOTOS ;*

Résultats *: STEM α-Réduit ;*

Variables *:*

Liste_Confs : liste de configurations non traitées ;

Liste_Confs_traitées : liste de configurations déjà traitées ;

Sub : liste de listes des couples (action, événement) représentant un même événement (la fonction σ)

Début

Construire la configuration initiale ;

Initialiser la liste de configurations Liste_Confs par la configuration initiale ;

Tantque *Liste_Confs non vide* ***Faire***

Sub ← ∅ ;

Sélectionner et supprimer un élément Conf de Liste_Confs ;

Traiter_Configuration *Conf ;*

Ajouter Conf à la liste Liste_Confs_traitées ;

Ajouter les nouvelles configurations résultantes à Liste_Confs ;

Ajouter les transitions résultantes à STEM ;

Substituer *STEM, Liste_Confs et Liste_Confs_traitées en utilisant Sub ;*

FinTant

FinAlgo.

- **Description** :

Tant qu'il y a une configuration *s* à développer, on construit l'ensemble de transitions maximales sensibilisées dans *s* (la fonction *Traiter_Configuration*), et pour chaque transition (s, ${}_M a_x$, s_i) de cet ensemble :

- on ajoute cette transition au *STEM*, et
- la configuration s_i est intégrée dans *Liste_Confs* si et seulement si elle est un nouvel état (*i.e.* elle n'est α-équivalente à aucune autre configuration générée précédemment).

La liste *Sub* reste vide jusqu'à la détection d'une configuration s_j existante, traitée ou non, qui est α-équivalente avec la configuration s_i (*i.e.* $\exists s_j \in$ *Liste_Confs* $\cup$ *Liste_Confs_traitées*: $s_j \approx_\alpha s_i$ (s_j et s_i sont α-équivalentes)), dans ce cas, le *STEM*, la *Liste_Confs* et la *Liste_Confs_traitées* doivent être substitués par *Sub*, ce qu'est effectivement réalisé au niveau de la fonction *Substituer*.

- **Raffinement de l'algorithme** :

La génération d'un $STEM_{/\approx\alpha}$ est basée principalement sur deux fonctions, à savoir la fonction *Traiter_Configuration* et la fonction *Substituer*.

La fonction *Traiter_Configuration* (fonction 01) nous permet de construire l'ensemble de transitions sensibilisées dans *Conf*, où on vérifie pour chacune de ces transitions si sa configuration cible *Conf '* est α-équivalente avec une autre configuration existante *Conf "* ou non, si oui :

- § on met leurs événements maximaux en relation, ceci se traduit par la liste de substitutions *Sub*, et
- § on remplace l'état cible de la transition par l'état attribué à *Conf "*.

Si ce n'est pas le cas, la configuration *Conf '* sera ajoutée à la liste de nouvelles configurations *Liste_Confs*. La transition sera ajoutée enfin à la liste de transitions dans le *STEM*.

Fonction 01 *"Traiter_Configuration"*

Données *: Une configuration Conf, Liste_Confs et Liste_Confs_Traitées ;*

Résultats *: Sub, liste de transitions, liste de nouvelles configurations ;*

Début

À partir de Conf tirer des transitions en utilisant la sémantique opérationnelle de maximalité ;

PourChaque *transition t résultante* ***Faire***

// Soit Conf ' la configuration cible de t :

Vérifier, s'il existe une Conf " appartenant à Liste_Confs ou à Liste_Confs_Traitées, telles que Conf ' et Conf " sont α-équivalentes;

Si *Conf " existe* ***Alors***

Construire la liste de substitutions Sub ;

Remplacer l'état cible de t par l'état attribué à Conf " ;

Sinon

Ajouter Conf ' à la liste de nouvelles configurations ;

Finsi

Ajouter t à la liste de transitions ;

FinPour

FinAlgo

La substitution (fonction 02) cherche à remplacer, selon la liste de substitutions construite, l'occurrence d'un événement par un autre événement donné, cette modification touche le *STEM*, la *Liste_Confs* et la *Liste_Confs_traitées*. Avec : $\pi(\beta_i) = a$, donne le nom de l'action représentée par le bloc β_i (liste de couples (action, événement)).

Fonction 02 *"Substituer"*

Données *: Une liste de substitutions Sub, un STEM, Liste_Confs et Liste_Confs_traitées ;*

Résultats *: Un STEM ;*

Début

Attribuer à chaque bloc $\beta_i \in$ Sub un couple ($\pi(\beta_i)$, get) "identificateur" ;

PourChaque *$\beta_i \in$ Sub* ***Faire***

PourChaque *(a, é) $\in$ β_i* ***Faire***

// La substitution du STEM :

Remplacer les occurrences de (a, é) dans le STEM par son identificateur ;

// La substitution de Liste_Confs et Liste_Confs_traitées :

Remplacer les occurrences de (a, é) dans Liste_Confs et Liste_Confs_traitées par son identificateur ;

FinPour

FinPour

FinAlgo

5.2.2. Le GPM et la méthode de réduction basée sur l'approche d'ordre partiel

C'est une méthode qui combine l'utilisation du modèle des STEMs [Sai96] et la méthode de pas couvrants [VAM96] [RV02]. Ainsi, le graphe obtenu est "*le graphe de pas maximaux*" GPM [Ben06].

En utilisant la notion d'événements maximaux, qui offre la possibilité de prendre en considération le parallélisme dynamique et d'exploiter au maximum les relations d'indépendance entre les actions, le STEM de la figure 5.3(c) peut être réduit en la structure de la figure 5.6.

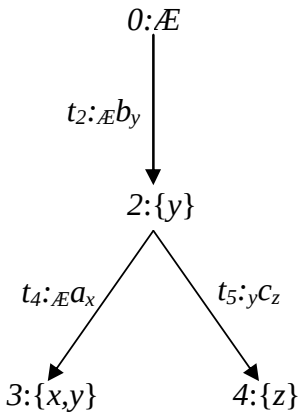


Figure 5. 6 : Graphe réduit

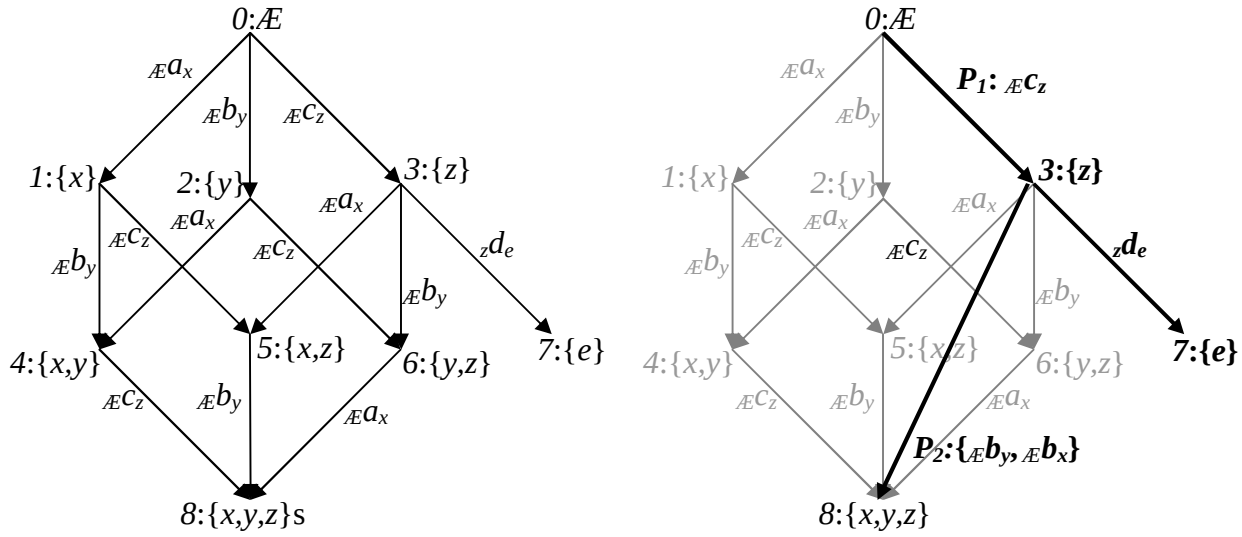
- ***Idée de base***

S'inspirant de la technique de pas couvrants [VAM96], on ne considère pas tous les entrelacements possibles. Par contre, on construit, sous certaines conditions, un pas permettant d'atteindre directement l'état final qui aurait été atteint par chacune des séquences entrelacées. La figure 5.7 montre le bénéfice obtenu dans le cas de la dérivation de trois actions parallèles (*a*, *b* et *c*) en présence de conflit différé. Le graphe de la figure 5.7(b) est le graphe de pas du STEM de la figure 5.7(a) dans lequel tous les entrelacements ont été convertis en deux pas (p_1 et p_2) : le premier pas exprime le début d'exécution de *c*, et l'autre exprime celui de l'exécution parallèle de *a* et *b*.

Le graphe de pas construit couvre le STEM initial modulo l'équivalence de "traces de *Mazurckiewicz*" [Maz86]. Il préserve également les états de blocage et la propriété de la vivacité, de plus, sa génération à la volée est possible [Ben06].

- ***Algorithme de construction à la volée du graphe de pas maximaux***

La construction à la volée du graphe de pas maximaux à partir d'une spécification LOTOS donnée est effectuée en se servant de l'extension directe (algorithme 02) de l'algorithme de génération à la volée du STEM α-réduit. La différence réside dans la ligne # (La construction des pas), qui consiste à vérifier, pour chaque transition développée, si elle peut faire partie d'un pas maximal ou elle est elle-même un pas.



(a) (b)

Figure 5. 7 : Un STEM et son graphe de pas maximaux

Algorithme 02 *"Algorithme de construction à la volée de GPM"*

Données *: Spécification LOTOS ;*

Résultats *: GPM ;*

Variables *:*

Liste_Confs : liste de configurations non traitées ;

Liste_Confs_traitéés : liste de configurations déjà traitées ;

T : liste des transitions ;

Sub : liste de listes des couples (action, événement) représentant un même événement (la fonction σ);

Début

Construire la configuration initiale ;

Initialiser la liste de configurations Liste_Confs par la configuration initiale ;

Tantque *Liste_Confs non vide* ***Faire***

Sub ← ∅ ;

Sélectionner et supprimer un élément Conf de Liste_Confs ;

Traiter_Configuration *Conf ;*

Ajouter Conf à la liste Liste_Confs_traitées ;

Ajouter les nouvelles configurations résultantes à Liste_Confs ;

Ajouter les transitions résultantes T à STEM ;

***Substituer** STEM, Liste_Confs et Liste_Confs_traitées en utilisant Sub ;*

***Construire_ pas** ; . . . . #*

FinTant

***FinAlgo**.*

- **La construction des pas** :

Afin d'assurer la préservation des chemins maximaux, le calcul (la fonction *Construire_ pas*) est effectué en cherchant pour chaque état *s'*, antécédent de l'état en cours de dérivation, à transformer ses petits chemins maximaux en pas. Avec C_s est l'ensemble de chemins maximaux associé à l'état *s*, et $Min(C_s)$ son ensemble de petits chemins.

***Fonction 03** "Construire_ pas"*

***Données** : s : l'état en cours de dérivation, GPM en cours de génération ;*

***Résultats** : Un GPM ;*

***Variables** :*

S : liste des antécédents de l'état s ;

Début

PourChaque** $s' \in S : s' \xrightarrow{p}_p s$ **Faire

Construire $C_{s'}$ et $Min(C_{s'})$;

Transformer $Min(C_{s'})$ en pas ;

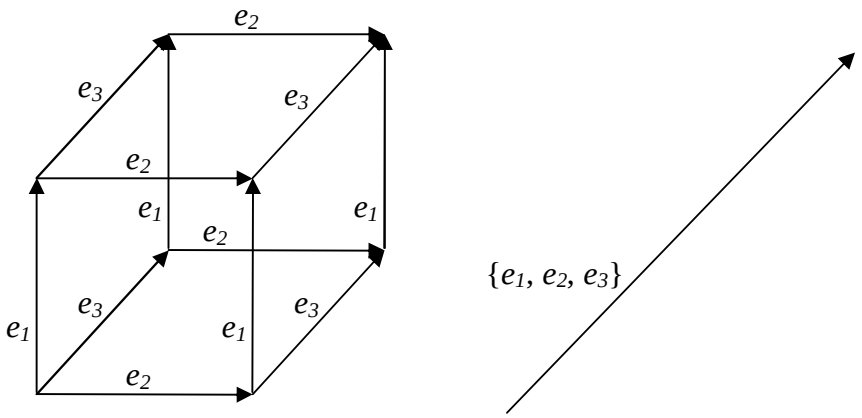
FinPour

FinAlgo

5.3. Etude de la complexité

Dans cette section, nous allons étudier la complexité computationnelle (algorithmique) dans le pire des cas des algorithmes présentés précédemment. La première vue de ces algorithmes donne directement l'impacte que leur complexité dépend du nombre de configurations du système, ainsi le nombre d'itérations nécessaire pour générer un des deux graphes réduits précédents est comparable à celles nécessaires pour générer le graphe d'états classique (par un algorithme exhaustif), le plus est introduit par les fonctions "*Traiter_Configuration*", "*Substituer*" et "*Construire_Pas*" introduites dans les calculs. Mais on doit faire attention aux nombres de configurations de ces deux graphes obtenus: comme un exemple simple, la figure suivante montre que, pour *n* événements parallèles, l'exploration selon le premier algorithme donne lieu à un hypercube d'ordre *n* : on obtient

un nombre exponentiel d'arcs et d'états. L'approche par "pas" utilisée dans le deuxième algorithme fournit un résultat optimal puisque le nombre d'états et d'arcs obtenus est indépendant du nombre d'évènements (parallèles).



	Approche par α-réduction	Approche par pas
Etats	2^n	2
Arcs	$n * 2^{n-1}$	1

Figure 5. 8 : Dérivation de *n* événements indépendants

5.3.1. L'algorithme de construction à la volée du STEM α-réduit

Le premier algorithme proposé permet de générer le STEM α-réduit pour une expression de comportement LOTOS donnée, il se compose essentiellement d'une boucle *Tantque* qui sert à parcourir l'ensemble de configurations générées, où pour chaque nouvelle configuration on construit la liste de transitions maximales sensibilisées, la liste de nouvelles configurations et la liste de couples de substitutions *Sub* (ceci est au niveau de la fonction *Traiter_Configuration*). Les transitions résultantes sont ajoutées au *STEM*, et les nouvelles configurations, qui ne sont pas α-équivalentes à aucune configuration générée précédemment, sont ajoutées à la liste *Liste_Confs*. La détection de configurations α-équivalentes aboutit à la construction de la liste de substitutions *Sub*, et ensuite à la substitution du *STEM*, de la *Liste_Confs* et de la *Liste_Confs_Traitées* en utilisant les blocs de substitutions (construits au niveau de la fonction *Substituer*).

Nous étudierons la complexité de cet algorithme en commençant par l'étude de la complexité des fonctions *Traiter_Configuration* et *Substituer*, puis nous conclurons la complexité globale de l'algorithme, mais avant ça, nous devons préciser la taille des données sur lesquelles on travaille.

a) Taille des données

En général, comme le montre le chapitre précédent, les résultats de complexité s'expriment en fonction d'une variable englobant toutes celles qui interviennent dans la description de l'algorithme. Nous avons choisi pour notre part de distinguer chacune des variables intervenant dans le calcul. Ainsi, la complexité de ce premier algorithme s'exprimera en fonction du nombre d'actions dans l'expression LOTOS en entrée, soit *N*, du degré de

parallélisme des actions, soit *P*, et du nombre de configurations (états) du STEM généré, soit *C*.

Pour ces paramètres, nous supposerons que *N* et *P* restent très inférieurs au nombre total d'états traités *C*, comme le montre le premier graphe de la figure 5.8, où le nombre de configurations est généralement exponentiel en fonction du nombre d'actions (parallèles) dans le système à vérifier.

b) La fonction "*Traiter_Configuration*"

Cette fonction génère pour la configuration en cours de traitement : la liste de transitions à ajouter au *STEM*, la liste de nouvelles configurations à ajouter à *Liste_Confs*, et la liste de substitutions à effectuer au niveau de la fonction *Substituer*. On construit en premier temps l'ensemble de transitions sensibilisées (majoré par *P*) dans la configuration en cours de traitement, ensuite on vérifie pour chacune de ces transitions si sa configuration cible *Conf '* est α-équivalente avec une configuration générée précédemment (dans *Liste_Confs* ou *Liste_Confs_Traitées*), soit *Conf"*, où on effectue au plus *C* comparaisons entre des configurations de $O(N)$ événements chacune. Si c'est le cas, alors on construit la liste *Sub* (dont la taille est majorée par $O(N)$) et on remplace l'état cible de la transition courante par l'état attribué à la configuration *Conf"*. Sinon, la configuration *Conf* ' est une nouvelle configuration, et elle doit être ajoutée à *Liste_Confs* pour être traitée ultérieurement. Finalement la transition traitée est ajoutée à la liste de transitions. Ces dernières opérations sont considérées comme opérations élémentaires.

Par conséquent, et selon sa structure, la fonction *Traiter_Configuration* est de complexité temporelle :

$$T_{Traiter\_Configuration} = O(P + P*(C*N + N)) = O(P*C*N).$$

c) La fonction "*Substituer*"

Dans cette fonction on cherche à remplacer l'occurrence d'un événement par un autre événement donné en utilisant la liste de substitutions *Sub* construite au niveau de la fonction *Traiter_Configuration*. On commence par l'attribution d'un identificateur à chaque bloc dans la liste *Sub* (majoré par $O(N)$: Chaque bloc de substitution correspond à une action, et on a $O(N)$ actions, donc on a $O(N)$ blocs). La modification, par la suite, va toucher le *STEM*, la *Liste_Confs* et la *Liste_Confs_Traitées* à travers deux boucles imbriquées, dans la première on parcourt l'ensemble de blocs de substitutions (majoré par $O(N)$), et pour chacun de ces blocs on parcourt les couples (généralement deux couples) pour faire la substitution du *STEM* ($O(C*N)$ substitutions), et des deux listes : *Liste_Confs* et *Liste_Confs_Traitées* ($O(C*N)$ substitutions).

Ainsi, la complexité temporelle de cette fonction est donnée par :

$$T_{Substitution} = O(N + N*2*(C*N)) = O(C*N^2).$$

d) L'algorithme

L'algorithme de construction à la volée de STEM α-réduit pour une spécification de comportement LOTOS consiste essentiellement à appliquer les deux fonctions précédentes pour chaque nouvelle configuration générée, ainsi on fait appels à ces deux fonctions pour toute nouvelle configuration, c'est-à-dire *C* appels en total (en fait, on ne connaît pas auparavant la valeur exacte de *C* en fonction de la taille des données en entrée (c'est-à-dire

en fonction de *N* et *P*), nous procédons donc comme dans [ACC+03] et nous introduisons le nombre d'états générés lors de la construction du graphe dans les expressions d'évaluation de la complexité).

Selon les résultats précédents et la structure générale de l'algorithme, la complexité temporelle pour construire le STEM α-réduit pour une expression LOTOS est donc donnée par :

$$T_{STEM_{/\approx\alpha}} = O(C * (P * C * N + C * N^2))$$

$$T_{STEM_{/\approx\alpha}} = O(C^2 * N * (P + N))$$

5.3.2. L'algorithme de construction du graphe de pas maximaux

L'algorithme (02) de construction à la volée du graphe de pas maximaux à partir d'une spécification LOTOS donnée a la même structure et presque les mêmes fonctions que l'algorithme(01) de construction de STEM α-réduit, dont la seule différence est l'utilisation -en plus des fonctions "*Traiter_Configuration*" et "*Substitution*"- de la fonction "*Construire_Pas*". Nous commençons donc par l'étude de la complexité de cette dernière fonction, et nous conclurons ensuite la complexité de l'algorithme général en utilisant les résultats concernant la complexité des fonctions "*Traiter_Configuration*" et "*Substitution*".

a) Taille des données

Tout comme pour l'algorithme de construction du STEM α-réduit, la complexité de l'algorithme de construction du graphe de pas maximaux sera exprimée en fonction du nombre d'actions dans l'expression LOTOS : *N*, du degré de parallélisme *P*, et du nombre de configurations traitées : *C* (*C* est le nombre d'états dans le STEM α-réduit. En fait, le nombre de configurations traitées dans cet algorithme n'est pas le nombre de configurations dans le graphe obtenu, mais c'est le nombre de configurations dans le STEM α-réduit correspondant, parce qu'on doit traiter toutes les configurations intermédiaires dans les différents pas pour construire à la fin le GPM). De plus, nous retiendrons la même hypothèse sur les trois paramètres *N*, *P* et *C*.

b) La fonction "*Construire_Pas*"

Cette fonction consiste à vérifier, pour chaque transition développée, si elle peut faire partie d'un pas maximal ou elle est elle-même un pas : On cherche pour chaque antécédent de l'état en cours de dérivation (on a au plus *C* antécédents) à transformer ses petits chemins maximaux en pas. En premier lieu, on construit :

- L'ensemble $C_{s'}$ de chemins maximaux associés à l'états *s'*, où on fait un test sur l'intersection des deux ensembles : l'ensemble des événements maximaux dans l'état courant et l'ensemble des événements qui conditionnent l'occurrence de la transition sensibilisée (au plus $O(P)$ transitions), ce qui nécessite $O(N^2)$ comparaisons pour chaque transition, donc $O(P*N^2)$ comparaisons en total.
- L'ensemble $Min(C_{s'})$ de petits chemins dans l'ensemble $C_{s'}$, où on cherche un chemin *c* dans $C_{s'}$ (avec $| C_{s'} | = O(P)$) tel qu'il n'existe pas un autre chemin *c'* (aussi dans $C_{s'}$) telle que la trace générée par *c'* soit incluse dans celle générée par *c*, donc on effectue $O(P^2)$ comparaisons sur des ensembles de $O(N)$ événements, ce qui nous donne $O(N^2*P^2)$ comparaisons en total.

On transforme par la suite les petits chemins dans $Min(C_{s'})$ (majoré par $O(P)$) en pas dans le graphe (où chaque chemin est de $O(N)$ éléments).

En conclusion, la complexité de la fonction de construction de pas maximaux est donnée par :

$$T_{Construire\_Pas} = O(C*(P*N^2 + P^2*N^2 + P*N)) = O(C*P^2*N^2)$$

c) L'algorithme général

L'algorithme de construction à la volée du graphe de pas maximaux à partir d'une spécification LOTOS donnée est de même structure que l'algorithme de construction à la volée du STEM α-réduit : parcours de l'ensemble de configurations, appels pour chaque configuration de la fonction "*Traiter_Configuration*", de la fonction "*Substitution*" et, dans ce cas, de la fonction "*Construire_Pas*". La complexité temporelle de cet algorithme est donc donnée par :

$$T_{GPM} = O(C*(\ P*C*N + C*N^2 + C*P^2*N^2))$$

$$T_{GPM} = O(C^2 * P^2 * N^2)$$

5.3.3. La complexité spatiale

Chacun des deux algorithmes a besoin d'un espace mémoire conforme à la taille du graphe généré, dans lequel on essaye de réduire, d'une manière ou d'une autre, le nombre d'états obtenus, mais la réduction diffère dans les deux algorithmes : Dans le premier on regroupe ensemble les configurations jugées α-équivalentes, mais on obtient en général un graphe de taille exponentielle en le nombre d'actions (parallèles). Dans le deuxième algorithme, la taille du STEM obtenu (nombre de configurations et de transitions) ne reste pas exponentielle -comme le montre la figure 5.8-, et nous aurons un rapport polynomial entre la taille du STEM obtenu et la taille de l'expression LOTOS en entrée. Donc, les deux algorithmes sont de complexités spatiales données par:

$$S_{STEM_{/\approx\alpha}} = O(C)$$

$$S_{GPM} = O(p(N, P))$$

Avec p désigne un polynôme.

5.4. Conclusion

Dans ce chapitre, nous avons exposé deux algorithmes proposés pour la construction à la volée des STEMs réduits (le STEM α-réduit et le GPM), nous nous sommes intéressés à leurs complexités asymptotiques temporelles et spatiales.

Nous avons montré que la complexité en temps, pour les deux algorithmes, est dépendante du nombre total de configurations du système, et par conséquent elle est exponentielle par rapport à la taille des données en entrée (précisément le nombre d'actions parallèles dans l'expression en entrée). En fait, il est évident que le nombre de configurations traitées dépend largement de l'expression LOTOS donnée (le nombre d'actions et le degré de parallélisme), en général ce nombre est exponentiel, et on reste toujours obligé de traiter

toutes les configurations du système. Par conséquent, les algorithmes proposés restent gourmands en ce qui concerne le temps d'exécution.

Cependant, les deux algorithmes n'ont pas la même complexité spatiale : pendant que le premier a une complexité spatiale de degré exponentiel en le nombre d'actions parallèles dans l'expression LOTOS en entrée, le deuxième algorithme a une efficacité considérable en termes d'espace mémoire, ce-ci dépend polynomialement de la taille de l'expression LOTOS en entrée.

Chapitre 6

Conclusion et perspectives

Le travail présenté dans ce mémoire représente un premier pas pour la maîtrise de la théorie de la complexité, ainsi nous avons étudié la complexité computationnelle asymptotique de quelques algorithmes développés au niveau de notre équipe, à savoir : l'algorithme de génération à la volée des STEMs α-réduits et l'algorithme de génération à la volée des graphes de pas maximaux GPM, nous avons constaté que ces deux algorithmes sont tous les deux de complexité temporelle exponentielle en la taille de l'expression LOTOS spécifiant le système à vérifier. Par contre, de point de vue complexité spatiale, le premier algorithme est de complexité spatiale exponentielle en le nombre d'actions parallèles dans le système à analyser, cependant, le deuxième algorithme a une complexité spatiale polynomiale, il offre donc un gain considérable dans la réduction des STEMs.

Après avoir donné une brève introduction à la théorie de la calculabilité et la théorie de la complexité, nous avons exploré, dans le chapitre 2, le monde de la complexité, plusieurs notions de base ont été éclairées. En fait, la théorie de la complexité fournit une diversité de concepts et de théorèmes liés à l'utilisation de ressources par une méthode (algorithme) sur un modèle de machine pour résoudre un problème. Ces modèles de machines (raisonnables) sont supposés être équivalents selon la thèse de l'invariance.

Dans le chapitre 3, nous avons parcouru les classes de complexité les plus populaires : les classes définies par des limites logarithmiques, polynomiales et exponentielles sur le temps et l'espace pour des modèles déterministes et non déterministes, nous avons considéré en particulier les classes PSPACE, P, NP, EXPTIME. Nous avons présenté aussi les relations entre ces classes et les types des problèmes qu'elles contiennent. Pour montrer qu'un problème particulier est dans une classe de complexité donnée, nous avons besoin d'exposer seulement une "méthode" pour résoudre le problème, qui satisfait les exigences (le modèle de machine et les limites de ressources) de la définition de cette classe.

Dans le chapitre 4, nous avons présenté la démarche générale pour l’analyse de la complexité des algorithmes. Etant donné un algorithme (séquentiel déterministe) pour la résolution d'un problème donné, sa complexité est souvent mesurée par le taux de croissance du temps et de la quantité de mémoire requise pour réaliser des calculs de

dimension de plus en plus grande. La complexité temporelle T(*n*) d'un algorithme est le nombre maximal d'étapes utilisées dans l'élaboration des calculs. On définit aussi la complexité spatiale S(*n*) qui est le nombre maximal de mots de mémoire utilisés pour effectuer les calculs. La limite de la complexité (temporelle ou spatiale) lorsque le calcul croît en dimension est la complexité asymptotique. On utilise pour noter la mesure de cette complexité la notation "*O*".

Dans le chapitre 5, Nous nous sommes intéressés à deux algorithmes pour la génération de graphes d'états réduits, à savoir les STEMs α-réduits et les GPMs, nous avons en outre évalué leur complexité computationnelle asymptotique (temporelle et spatiale), en donnant des bornes supérieures sur la complexité dans le pire des cas de chacun de ces algorithmes. La génération des STEMs α-réduits nécessite un temps et un espace exponentiel en la taille de l'expression LOTOS donnée, la génération des graphes de pas maximaux reste aussi gourmande en temps d'exécution, mais elle est pour une efficacité considérable en termes d'espace mémoire.

Perspectives

Ce travail n'est qu'une première exploration du monde de la théorie de la complexité. Effectivement, la maîtrise de la théorie de la complexité ne s'arrête pas dans l'évaluation de la complexité computationnelle temporelle et spatiale d'un algorithme ou d'une méthode pour la résolution d'un problème donné, l'objectif principal est de comprendre la complexité intrinsèque du problème lui-même, il nous reste la tâche difficile de déterminer la classe de complexité du problème de réduction du graphe d'états.

D'autre part, le travail présenté dans ce document peut être suivi dans le sens d'amélioration des algorithmes proposés, il paraît nécessaire de réduire la complexité temporelle de génération des STEMs réduits (plus particulièrement les GPMs), pour obtenir un bon compromis temps - espace.

Bibliographie

[ACC+03] C. ARNOULD, D. CRESTANI, C. COVES, A. JEAN-MARIE, F. PRUNET "*Complexité algorithmique asymptotique temporelle d'algorithmes de construction de graphe de couverture*." MOSIM'03, pp.82-87, Toulouse, 23-25 Avril 2003.

[AK05] Scott AARONSON and Greg KUPERBERG. *The Complexity Zoo*. http://qwiki.caltech.edu/wiki/Complexity Zoo

[ALR04] Eric ALLENDER, Michael C. LOUI and Kenneth W. REGAN "*Complexity Theory*". May 25, 2004.

[ALR99] Eric ALLENDER, Michael C. LOUI and Kenneth W. REGAN "*Complexity Classes*". In *Algorithms and Theory of Computation Handbook*, ed. M. J. Atallah, CRC Press, Boca Raton, Fla. 1999.

[Att01] Christian ATTIOGBE "*Eléments de Complexité et de Calculabilité*" Notes de cours - Licence. Faculté des sciences et des techniques, Université de Nantes. Juillet 1997, mars2001.

[Ben06] A. BENAMIRA "*Vérification des équivalences de comportements des systèmes concurrents*" Mémoire de Magistère. Laboratoire LIRE, Université Mentouri de Constantine 2006.

[Bou03] Olivier BOURNEZ "*Complexité de problèmes*", Cours DEA 2003. http://www.loria.fr/~bournez/cours/indexdeacomp.html

[Car06] Olivier CARTON, "*Langages formels, Calculabilité et Complexité*", Formation prédoctorale, école Normale Supérieure, Version du 28 septembre 2006.

[Cha02] Jean-Cédric CHAPPELIER "*Cours d'introduction à l'informatique et à la programmation* : *Bases d'algorithmique*" Laboratoire d'Intelligence Artificielle - Faculté I&C. Ecole Polytechnique Fédérale de Lausanne © EPFL 2002-2006.

[CKS81] CHANDRA, A.K., D.C.KOZEN and L.J.STOCKMEYER, "*Alternation*", J.ASSOC. Comput. Mach.28 (1981)

[CP05] P. CHRETIENNE et C. PICOULEAU "*Complexité des problèmes combinatoires*". Master d'informatique - Spécialité IAD.2005

http://www-master.ufr-info-p6.jussieu.fr/2005/IMG/pdf/courscomplexite.pdf

[EG95] Thomas EITER and Georg GOTTLOB "*The Complexity of Logic-Based Abduction*"-Technische Universität Wien, Wen, Austria. *Journal of the Association for Computing Machmwy*, Vol.42, No.1, pp.3-42, January1995.

[Emd90] Peter Van EMDE BOAS "*Machine models and simulations*". In "*Handbook of theoretical computer science*", Volume A: "*Algorithms and complexity*". Edited by J.Van LEEUWEN. Elsevier Science Publishers B.V., 1990.

[Fro05] Christine FROIDEVAUX "*Algorithmique et Complexité*" UE de S5 - Licence d'Informatique - Université de Paris Sud. 2005-2006.

[GHR95] Raymond GREENLAW, H. James HOOVER and Walter L. RUZZO "*Limits to Parallel Computation*: *P-Completeness Theory*". Oxford University Press. 1995.

[Har02] Jamila Sam HAROUD "*Cours d'introduction à l'informatique et à la programmation objet*: *Complexité des problèmes*" Laboratoire d'Intelligence Artificielle - Faculté I&C-EPFL. 2002-2006.

[HC91] Ian HODKINSON and Margaret CUNNINGHAMAS "*Computability, Algorithms, and Complexity*". Second-year undergraduate course in the Department of Computing at Imperial College, London, UK, since 1991.

[Hir04] Robin HIRSCH "*Computability and complexity Part II, Complexity*". November 10.2004.

[Joh90] David S. JOHNSON "*A Catalog Of Complexity Classes*". In "*Handbook of theoretical computer science*", Volume A: "*Algorithms and complexity*". Edited by J.Van LEEUWEN. Elsevier Science Publishers B.V., 1990.

[Jou] Jean-Pierre JOUANNAUD "*Vérification des Systèmes Réactifs Temps-Réel*". Université Paris-Sud - École Polytechnique.

http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/articles/cours-verification.pdf

[Lok03] Satyanarayana V. LOKAM, "*Computational Complexity Theory*", Lecture Notes. EECS 574, Fall 2003.
http://www.eecs.umich.edu/courses/eecs574/notes.html

[LR93] Richard LASSAIGNE et Michel de ROUGEMONT, "*Logique et fondements de l'informatique : Logique du 1er ordre, calculabilité et l-calcul*". Editions HERMES, Paris. 1993.

[Maz86] A. MAZURCKIEWICZ "*Trace theory*" in "*Petri nets: Applications and relationships to other model of concurrency*". "*Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*". Springer Verlag, LNCS 255 (1986).

[RT03] Christophe RAPINE et Denis TRYSTRAM "*Théorie de la complexité*", Notes de cours, DEA Recherche Opérationnelle, Combinatoire et Optimisation. Octobre 2003.

[RV02] P. O. RIBET, F. VERNADAT "*Graphes De Pas Couvrants Vers Une Relation De Conflit Dynamique*" Rapport LAAS 02065, Mars 2002.

[Sai96] D. E. Saidouni. "*Sémantique de Maximalité*: *Application Au Raffinement D'actions Dans LOTOS*". PhD thesis, LAAS, Université Paul Sabatier, Toulouse (Mars 1996).

[Sav70] SAVITCH, W.J., "*Relationship between nondeterministic and deterministic tape classes*", J. Comput. System Sci. 4 (1970).

[Sim90] Herbert A. SIMON "*Sur La Complexité des Systèmes Complexes*". Université Carnegie-Mellon, Pittsburgh. Dans "*Les Introuvables en langue française de H. A. Simon.* (*Document n° 6*)", traduction fut initialement publiée dans ' la Revue Internationale de Systémique' Vol. 4, N° 2, 1990.

[Ski97] Steven S. SKIENA "*The Algorithm Design Manual*" Department of Computer Science - State University of New York. Stony Brook, NY 11794-4400. Copyright © 1997 by Springer-Verlag, New York.

[Sto05] Viggo STOLTENBERG-HANSEN "*Computability and Complexity over Discrete Structures*" Notes for an FMB course - Second Draft. Department of Mathematics, Uppsala University - Sweden, 5 April 2005.

[Tra05] DE TRAN-CAO "*Mesure de la Complexité Fonctionnelle des Logiciels*" thèse présentée comme exigence partielle du doctorat en informatique cognitive. Université du Québec à Montréal. JUIN 2005.

[Tre04] Luca TREVISAN "*Lecture Notes on Computational Complexity*". Notes written in Fall 2002, Revised May 2004.

[VAM96] F. VERNADAT, P. AZEMA, and F. MICHEL. "*Covering step graph*". In "*Proceedings of Application and Theory of Petri Nets 96*", volume LNCS 1091. Springer Verlag 1996.

[Wat05] Osamu WATANABE "*Computational Complexity Theory*" Course Note. Dept. Mathematical Computing Sciences, Tokyo Institute of Technology. 2005.

[Wil94] Herbert S. WILF "*Algorithms and Complexity*" University of Pennsylvania - Philadelphia, PA 19104-6395. Internet Edition, 1994. http://www.math.upenn.edu/%7Ewilf/AlgoComp.pdf

[Yap98] Chee Keng YAP "*Theory of Complexity Classes*" VOLUME 1. Courant Institute of Mathematical Sciences - New York University. September 29, 1998.

Etude de la Complexité de Modèles de Spécification et de Méthodes de Vérification des Systèmes Complexes

Mokdad AROUS

Résumé

La théorie de la complexité est un outil important pour diriger l'algorithmique, elle nous aide à développer notre compréhension des difficultés inhérentes au calcul, en connaissant les coûts (le temps d'exécution et l'espace mémoire) des calculs, et en identifiant les différentes sources de complexité. En effet, L'utilisation des différents modèles et méthodes à bon escient doit passer impérativement par l'étude de leur complexité (computationnelle). Il s'avère donc important de pouvoir quantifier les besoins en ressources des algorithmes, en particulier en temps et en espace mémoire.

Ce mémoire présente une étude de la complexité asymptotique computationnelle de la génération des STEMs (*Systemes de Transitions Etiquetées Maximales*) réduits, à savoir les STEMs α-réduits et les graphes de pas maximaux GPM, proposés dans le cadre de la vérification formelle des systèmes complexes.

Nous étudions, en premier lieu, la base formelle de la théorie de la complexité, les différentes techniques utilisées et les résultats majeurs trouvés.

Par la suite, nous évaluons la complexité asymptotique computationnelle (temporelle et spatiale) des méthodes de réduction proposées. Nous démontrons que la génération des deux types de STEMs réduits reste gourmande en temps d'exécution. Cependant, les GPMs sont d'une efficacité considérable en termes d'espace mémoire.

Mots-clés : Théorie de la complexité computationnelle, Modèles de machine, Classes de complexité, STEM α-réduit, Graphe de Pas Maximaux.

Complexity Study of Specification Models and Verification Methods of Complex Systems

Abstract

The complexity theory is an important tool to direct the algorithmic, it helps us to develop our understanding of the inherent difficulties of computation, by knowing the costs (execution time and memory space) of the computations, and identifying the different sources of complexity. Indeed, the use of different models and methods advisedly must pass imperatively by the survey of their (computational) complexity. It proves to be therefore important to be able to quantify the resources requirements of algorithms, in particular in time and in memory space.

This memoir presents a survey of the asymptotic computational complexity of the generation of reduced MLTSs (*Maximality-based Labeled Transitions Systems*), namely the α-reduced MLTSs and the *Maximal Steps Graphs* (MSG), proposed in the context of the formal verification of complex systems.

Firstly, we study the formal basis of the complexity theory, the different used techniques and the major found results.

Afterwards, we value the asymptotic computational (temporal and spatial) complexity of the proposed reduction methods, we demonstrate that the generation of the both types of MLTSs remains gourmand in execution time, however, the MSGs are of a considerable efficiency in term of memory space.

Keywords : Computational complexity theory, Machine models, Complexity classes, α-reduced MLTS, Maximal Steps Graph.