Classes de Complexité
&
Problèmes de Satisfaction de Contraintes

par Vladimir Reinharz

Classes de Complexité
&
Problèmes de Satisfaction de Contraintes

Vladimir Reinharz

23 avril 2009

Table des matières

Introduction

Les problèmes de satisfaction de contraintes se retrouvent dans tous les champs des sciences, que ce soit les mathématiques, la physique, la chimie, l'économie, etc... Intuitivement, nous avons un ensemble de variables reliées entre elles par un ensemble de contraintes. Nous recherchons à assigner à chacune des variable un élément d'un domaine fini afin que toutes les contraintes soient satisfaites. Afin de les résoudre, nous avons depuis moins d'une centaine d'années accès aux ordinateurs. Ceux-ci peuvent calculer des milliards d'opérations par seconde et se révèlent ainsi des alliés précieux. Malheureusement les ordinateurs ont eux aussi leurs limites, cela n'est d'aucune utilité si le domaine de recherche des solutions est démesurément grand. De plus , ils ne peuvent résoudre un problème que de la façon dont ils ont été programmés pour le faire. Si un programme prend 10^50 opérations pour trouver une solution, nous ne sommes pas plus avancés. Une des branches de la théorie informatique s'intéresse donc à classifier les problèmes selon la capacité que nous avons à les programmer, plus ou moins efficacement, sur des ordinateurs.

Dans tous les cas, il y a deux contraintes principales qui se posent, le temps et l'espace. Car, même si nous connaissons des algorithmes permettant de factoriser un nombre, ceux-ci nécessitent un temps dépassant de loin la durée de l'univers, pour des nombres d'une grandeur moyenne. Il est donc extrêmement intéressant de classer les problèmes selon des bornes de temps et d'espace nécessaire au calcul, ces bornes dépendant de la grandeur de l'entrée du problème. Il existe plusieurs méthodes pour faire cela, en particulier

pour les problèmes de satisfaction de contraintes, nous pourrions utiliser les systèmes logiques, la théorie des graphes, ou l'algèbre universelle. Dans le présent travail nous utiliserons cette dernière approche.Pour bien comprendre les difficultés qui se posent dans ce genre de contexte, nous devrons d'abord poser les bases de la théorie informatique. Cela se fera à travers les différents types de machines de Turing, qui sont nos modélisation d'ordinateurs. Ce sont ces machines qui nous permettrons de définir la complexité spatiale et temporelle. Puis nous verrons comment nous pouvons décrire des familles de problèmes de satisfaction de contraintes et les associés à des algèbres particulières. En particulier, le cas de l'algèbre ayant les fonctions de Mal'tsev sera présenté. Cet exemple est important car il permet de montrer que nous pouvons résoudre, en un temps considéré comme acceptable, tous les systèmes d'équations linéaires, pour lesquels les algorithmes classiques de résolution des problèmes de satisfaction de contraintes ne fonctionnent pas.

Chapitre 1

Classes de complexités

1.1 Machines de Turing

Il faut, afin de développer une théorie rigoureuse, commencer par formaliser ce que nous considérons comme un ordinateur, ce qui se fait à travers la notion de machines de Turing. Ces machines sont d'une simplicité désarmante, mais cette simplicité de fonctionnement permet pourtant de représenter tout ce que peut faire, théoriquement, un ordinateur. Cette machine est essentiellement composée de deux parties, une bande de longueur infinie, vide, et une tête de lecture\écriture. La machine reçoit en entrée une bande et par une suite d'opérations prédéterminés elle va accepter ou rejeter l'entrée. De manière générale le problème se pose comme cela : étant donné un langage L, existe-t-il une machine de Turing vérifiant si une entrée appartient à ce langage. Un problème sera donc programmable sur un ordinateur s'il existe une machine de Turing pouvant le représenter. Bien sûr, avec une telle définition, il est facile de trouver un problème représentable par un tel type de machine mais dont l'implantation sur un ordinateur nécessite un temps ou un espace dépassant les ressources physiques mises à notre disposition. Au début de la théorie informatique, dans les années 60, afin de faire une analyse rigoureuse de ce qui est réalistement présentable ou non sur un ordinateur, la notion de classes de complexité a été inventée afin de classer les

problèmes selon leur difficulté de modélisation, soit en terme de temps de calcul ou d'espace physique nécessaire. Ces notions découlent toutes de la définition de machines de Turing.

Mais premièrement, nous devons décrire ce que nous entendons par langage, et ce que nous voulons que la machine de Turing puisse prendre en entrée. Un langage L est un ensemble de chaînes de caractères provenant d'un alphabet Σ. Le problème est donc le suivant : Existe-t-il une machine de Turing M qui, prenant une chaîne de caractère en entrée, détermine si cette chaîne appartient au langage L ? Si oui, nous dirons que M reconnaît L.

Définition 1.1. Une **machine de Turing déterministe à une bande**, noté **TM** est un septuplet,
$(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, où Q, Σ et Γ sont des ensembles finis et :

1. Q est l'ensemble des états de la machine de Turing.
2. Σ est l'alphabet de la machine : toutes les bandes de données que nous demandons à la machine de lire doivent être constituées d'une suite d'éléments de Σ. Il est à noter que le caractère $\llcorner$, représentant une case vide, n'appartient jamais à Σ.
3. Γ est l'alphabet que nous pouvons retrouver sur la bande infinie de la machine de Turing. Notons que $\llcorner \in \Gamma$ et que $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L,R}\}$, est la fonction de transition de la tête de la machine. À chaque case de la bande d'entrée, étant donné l'état où se trouve la machine, la tête va modifier la case qu'elle est en train de lire, changer l'état de la machine, puis se déplacer à gauche ou à droite sur la bande. La dernière valeur du triplet indique le sens du déplacement, L pour gauche et R pour droite. Si la tête se retrouve à l'extrémité gauche de la bande et reçoit comme instruction de se déplacer encore vers la gauche, elle ne bouge pas. L'entrée est une suite de caractères $a_0, a_1, \ldots, a_n$, et donc la machine commence avec $\delta(q_0, a_0)$.
5. $q_0 \in Q$ est l'état initial de la machine.

6. $q_a \in Q$ est l'état dans lequel la machine accepte la bande d'entrée. Dès que la machine se retrouve dans cet état le processus s'arrête.
7. $q_r \in Q$ est l'état où la machine rejette la bande d'entrée. Dès que la machine se trouve dans cet état le processus s'arrête.

Il faut noter que la TM peut continuer à rouler indéfiniment s'il se créé une boucle infinie ne contenant ni l'état q_a ni q_r. Évidemment, même si un langage peut être reconnue par une TM, la description de celle-ci peut devenir incroyablement complexe. Nous définissons donc les machines de Turing à plusieurs bandes et les machines de Turing non-déterministes, qui permettent de décrire certaines situations avec plus de facilité. Ces deux types de machines sont équivalentes aux TM, c'est-à-dire qu'il existe une TM reconnaissant le même langage. Comme le dit son nom, le premier de ces types de machines est exactement comme une TM mais avec plusieurs bandes d'écriture/lecture et sur chacune d'entre elles une tête. Les machines non-déterministes ont une fonction de transition qui, à chaque point et état donné, donne une possibilité d'écriture, de changement d'état, et de déplacement pour la tête, appartenant à un ensemble fini. Donc, pour tout couple $(q, \sigma) \in Q \times \Sigma$, il existe un nombre de valeurs finis que peut prendre $\big(\delta(q, \sigma)\big) \in (Q \times \Sigma \times \{L, R\})$. La fonction δ n'est donc pas bien définie. Ce type de machine est primordial afin de caractériser aisément certaines classes de complexité.

Définition 1.2. Une **machine de Turing à plusieurs bandes**, noté **MTM**, est comme une TM, mais avec plusieurs bandes, et avec une fonction de transition un peu plus complexe. Au commencement, l'entrée apparaît sur la première bande, et toutes les autres bandes sont vide. La fonction de transition peut faire écrire, lire et bouger les têtes sur chacune des bandes en même temps et est, formellement, pour une machine ayant k bandes :

$$\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{\mathrm{L,R}\}^k$$

Théorème 1.1. *Toute machine de Turing à plusieurs bandes a une machine de Turing déterministe à une bande qui lui est équivalente.[7]*

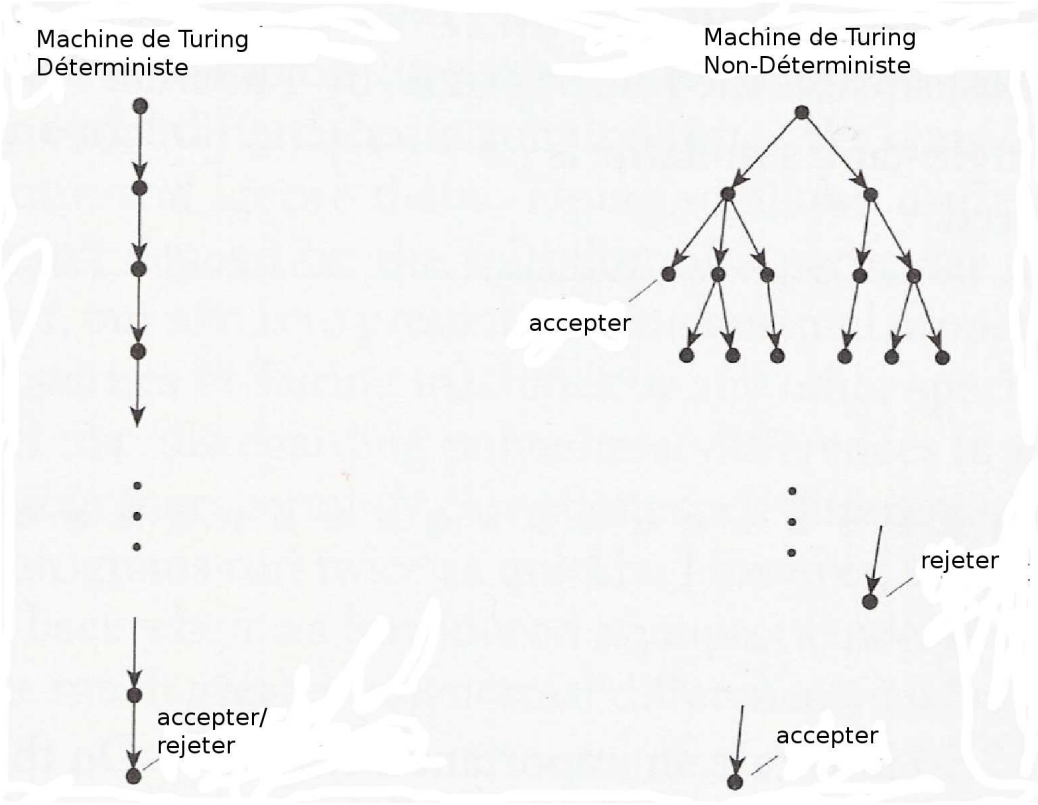

FIG. 1.1: Représentation schématique du calcul par une TM et une NTM d'une entrée quelconque

Il est clair que la réciproque est vrai, toute TM est une MTM à une bande. Définissons maintenant les machines de Turing non-déterministes.

Définition 1.3. Une **machine de Turing non-déterministe**, noté **NTM**, est un septuplet, $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, où $Q, \Sigma, \Gamma, q_0, q_a$ et q_r sont définis comme dans la définition 1.1, et la fonction de transition δ est :

$$\delta : Q \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{\text{L,R}\})$$

où $\mathcal{P}(Q \times \Sigma \times \{\text{L,R}\})$ choisi de manière aléatoire un des triplets sur lesquels peut être envoyé le couple (q, σ), avec $q \in Q, \sigma \in \Sigma$. Une NTM accepte une entrée w si une des branches du calcul de celle-ci contient l'état q_a.

Nous pouvons nous représenter une NTM comme un arbre où, à chaque

nœud, la machine décide aléatoirement quelle branche suivre, voir figure 1.1. Il est clair que chaque machine de Turing est en particulier une machine de Turing non-déterministe, où, à chaque opération, il n'y a qu'une seule possibilité. C'est dans l'autre sens que le résultat est intéressant.

Théorème 1.2. *Toute machine de Turing non-déterministe possède une machine de Turing déterministe à une bande qui lui est équivalente.*

Démonstration. Soit N, une NTM, et D une MTM à trois bandes. Par le théorème 1.1, si nous montrons que D est équivalente à N, alors c'est fini. Décrivons les bandes de D. La première bande contient la chaîne d'entrée et ne la modifie jamais. La deuxième bande contient une copie de la bande de N sur une de ses branches non-déterministe. Finalement, la troisième bande situe où se trouve D dans l'arbre de N.

Commençons par la troisième bande. Posons b comme le nombre maximal de possibilités que peut nous donner la fonction de transition de N. Soit $\Sigma_b := \{1, 2, \ldots, b\}$, un alphabet. Alors nous assignons à chaque nœud dans l'arbre une chaîne de caractères sur cet alphabet. Par exemple 223 est le nœud que nous rencontrons lorsque, en partant de l'origine nous prennons la deuxième bifurcation, au nœud suivant encore la deuxième puis au nœud d'après la troisième. Chaque symbole nous indique donc quel chemin prendre lorsque nous simulons une branche de N. Parfois, le symbole sera plus élevé que le nombre de possibilités, alors l'entrée est invalide et ne correspond à aucun nœud. La chaîne vide est l'adresse de la racine de l'arbre. Nous pouvons maintenant décrire le fonctionnement de D.

1. Au début, la bande 1 contient la chaîne de caractères d'entrée w, et les bandes 2 et 3 sont vides.
2. Copier la bande 1 sur la bande 2
3. Utiliser la bande 2 pour simuler une branche de N avec l'entrée w. Avant chaque étape, D regarde sur la bande 3 quel choix faire parmi ceux que permet la fonction de transition de N. S'il ne reste plus de symboles sur la bande 3, que le choix suivant est invalide, ou si une configuration de

rejet est rencontrée, aller à l'étape 4. Si une configuration d'acceptation est rencontrée, accepter w.

4. Remplacer la bande 3 par la chaîne de caractères suivante dans l'ordre lexicographique habituel. Simuler la branche suivante de N en allant à l'étape 2.

Donc pour tout w acceptée par N, D l'accepte aussi. Et par le théorème 1.1, le résultat est démontré.

□

1.2 Complexité temporelle

1.2.1 TM et la complexité temporelle

Une des premières interrogations lorsque nous avons un langage ainsi que sa machine de Turing associée, peu importe son type comme nous avons démontré qu'elles sont toutes équivalentes, est : combien d'opérations sont nécessaires à la TM avant de déterminer si l'entrée appartient bien au langage. Dans un cadre pratique, même si nous connaissons une TM permettant de vérifier l'appartenance d'une chaîne de caractères à un langage, mais que la machine demande une très grande quantité d'opérations pour une entrée courte, les algorithmes équivalents sur un ordinateurs seront également très long à exécuter. Il faut donc déterminer ce que nous pouvons considérer comme résoluble dans un temps réaliste, cela revient à classer les TM en fonction de la longueur de leur temps d'exécution. Pour cela, il faut d'abord définir la complexité temporelle d'une TM.

Définition 1.4. Soit M une machine de Turing qui accepte ou rejette toutes les entrées. La **complexité temporelle**, ou **temps d'exécution**, de M est la fonction $f : \mathbb{N} \to \mathbb{N}$, où $f(n)$ est le nombre maximum d'étapes que fait M avant de s'arrêter, pour toute entrée de longueur n. Si $f(n)$ est le temps d'exécution de M nous dirons que M s'exécute en temps $f(n)$.

Comme la fonction du temps d'exécution d'un algorithme en fonction de la taille de son entrée est souvent très compliquée, nous utiliserons la notation asymptotique grand «O» pour décrire la complexité temporelle des machines et pour définir les classes de complexité. De plus cela permet d'éviter une dépendance trop forte de l'évaluation du temps d'exécution par rapport au modèle utilisé.

Définition 1.5. Soit $t : \mathbb{N} \to \mathbb{N}$, une fonction. La **classe de complexité temporelle** $TIME(t(n))$ est :

$$TIME(t(n)) := \{L | L \text{ est un langage tel qu'il existe une TM le reconnaissant en un temps d'exécution } O(t(n))\}.$$

De façon équivalente nous définissons la notion de temps d'exécution pour les NTM. Par la suite nous montrerons que si une NTM permet, pour toute entrée, de vérifier si elle appartient ou non à un langage, une machine de Turing déterministe à une bande peut le faire également, mais en beaucoup plus de temps.

Définition 1.6. Soit N une NTM tel que chaque entrée est acceptée ou rejetée. Le temps d'exécution de N est la fonction $f : \mathbb{N} \to \mathbb{N}$, où $f(n)$ est le plus grand nombre d'opérations que N peut exécuter sur chacune de ses branches avec une entrée de longueur n.

Théorème 1.3. *Soit $f(n)$ une fonction, $f(n) \geq n$. Alors pour toute NTM ayant un temps d'exécution de $t(n)$, il existe une TM équivalente ayant un temps d'exécution de $2^{O(t(n))}$.*

Démonstration. Soit N une machine de Turing non-déterministe ayant un temps d'exécution de $t(n)$. Nous construisons une MTM D comme pour la démonstration du théorème 1.2. Analysons la situation.

Pour une chaîne de caractères w en entrée, le calcul de chaque branche de N prend un espace maximal de $t(n)$. Comme en chaque nœud il y a au plus b possibilités, où b est le nombre maximal de choix que peut donner la fonction

de transition de N, il y a donc au plus $b^{t(n)}$ feuilles à l'arbre de N. Comme le nombre maximal de nœuds sur un arbre est au maximum inférieur au double du nombre de feuilles, nous pouvons le borner par $O(b^{t(n)})$. Le temps nécessaire pour arriver à un certain nœud, en partant de la racine, est au plus $O(t(n))$. Le temps d'exécution de D est donc au plus $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. Il est facile de démontrer que le temps d'exécution sur une TM d'une MTM est au plus le carré du temps requis par la MTM. Donc, le temps d'exécution de N sur une TM est $(2^{O(t(n))})^2 = 2^{O(t(n))}$ □

1.2.2 P et NP

Maintenant que nous avons défini rigoureusement la notion de temps d'exécution, nous devons remarquer une chose essentielle. Lors de la démonstration du théorème 1.3, il y a une augmentation quadratique du temps d'exécution des MTM sur une TM et une augmentation exponentielle du temps pour l'exécution d'une NTM sur une TM. Alors que dans le premier cas il n'y a qu'une différence polynomiale, dans le deuxième, l'augmentation du temps de calcul est drastique. Les NTM ne pouvant pas être représentées sur un ordinateur, nous nous intéresserons au temps d'exécution sur les TM. De plus, à partir de maintenant, les aspects de la théorie ne feront pas la différence si nous sommes à un polynôme près. Non pas que la différence entre deux polynômes soit négligeable, si notre machine prend un temps d'exécution de $O(n^{1000})$, qui est un polynôme, il devient vite impossible de calculer la réponse en un temps raisonnable, mais cela permet de donner une perspective ayant plus de recul sur l'ensemble de ces problèmes, et de développer une théorie très riche. Cela nous amène à définir, afin de classifier les langages, les classes de complexité.

Définition 1.7. P est la classe des langages qui sont décidables en un temps polynomial sur une machine de Turing déterministe à une bande. Formellement :

$$P := \bigcup_{k \in \mathbb{N}} TIME(n^k)$$

La classe P joue un rôle primordial dans la théorie pour principalement deux raisons. Premièrement, P est invariante par rapport aux modèles de calcul équivalentes en un temps polynomial aux TM, entre-autre les MTM. Et deuxièmement car P correspond, grossièrement, aux problèmes que nous pouvons espérer résoudre sur un ordinateur. Nous allons présenter maintenant un problème classique, $PATH$, puis montrer que ce problème appartient à P.

Exemple 1.1. *$PATH$ est un problème sur les graphes dirigés qui consiste à vérifier s'il existe un chemin entre deux points du graphe. Formellement, soit G un graphe dirigé, s et t deux nœuds du graphe, alors :*
$PATH = \{\langle G, s, t\rangle | G \text{ est un graphe dirigé qui a un chemin directe de } s \text{ à } t\}$

Théorème 1.4. $PATH \in P$

Démonstration. Nous allons tout simplement donner un algorithme M qui vérifie $PATH$ puis l'analyser et montrer qu'il s'exécute en un temps polynomial.
Soit $M :=$ «avec l'entrée $\langle G, s, t\rangle$, où G est un graphe dirigé contenant les nœuds s et t :

1. Placer une marque sur le nœud s.
2. Répéter l'étape suivante jusqu'à ce qu'il n'y ait plus de nœuds additionels de marqués :

   Regarder toutes les arêtes de G. Si une arrête (a, b) est trouvé allant d'un nœud a, marqué, vers un nœud b, qui ne l'est pas, alors marquer b.
3. Si t est marqué, accepter, sinon rejeter.»

Il est évident que les étapes 1 et 3 ne sont executées qu'une fois. Posons $m := |G|$, alors l'étape 2 est répétée au plus m fois car à chaque itération nous marquons un point qui ne l'était pas précédemment. Le nombre d'opérations est donc au plus $1 + 1 + m$, soit un polynôme en fonction de la taille de G. Il est facile de voir que ces trois étapes peuvent être simplement implémentées

de manière polynomiale sur une machine de Turing déterministe. Donc M est un algorithme de temps polynomial pour $PATH$. □

Évidemment, il existe une énorme quantité de problèmes pour lesquels nous ne connaissons pas d'algorithmes polynômiaux permettant de les résoudre. Peut-être est-ce que nous ne les avons toujours pas trouvés, ou qu'ils n'existent tout simplement pas. Par contre, une des grandes découvertes dans ce champs théorique est que la complexité de beaucoup de problèmes sont liés, et donc si nous trouvons un algorithme pour un de ces problèmes, il y a toute une classe de problèmes qui sera résolue du même coup. Nous allons donc maintenant définir la notion de vérificateur d'un langage, ce qui va nous permettre d'introduire la prochaine classe très importante, NP.

Définition 1.8. Un **vérificateur** pour un langage A est un algorithme V tel que :
$A = \{w | V \text{ accepte } \langle w, c\rangle \text{ pour une certaine chaîne de caractères } c.\}$
Nous mesurons le temps d'exécution d'un vérificateur seulement en fonction de la longueur de w. Nous dirons qu'un langage A est vérifiable polynomialement s'il existe un vérificateur qui affirme ou infirme, pour toute chaîne de caractères w, si celle-ci appartient où pas à A, en un temps polynomial.
Un vérificateur utilise de l'information supplémentaire, représenté par c, afin de vérifier l'appartenant de w à A. Cette information est appelé le **certificat**, ou la **preuve**, d'appartenance à A. Notons que c doit avoir une taille polynomiale en fonction de la longueur de w, car sinon le vérificateur n'aurait pas le temps de le lire en entier, étant donné que son temps d'exécution est borné par une telle grandeur.

Définition 1.9. NP est la classe de tous les langages ayant un vérificateur s'exécutant en un temps polynomial.

L'intérêt de la classe NP est qu'il existe une quantité énorme de problèmes lui appartenant. Par exemple : $HAMPAHT = \{\langle G, s, t\rangle | G$ est un graphe dirigé tel qu'il existe un chemin Hamiltonien de s à $t\}$. La question mainte-

nant est : comment faire pour identifier si un problème se retrouve où pas dans NP. Le théorème suivant donne un moyen simple d'y répondre.

Théorème 1.5. *Un langage appartient à NP si et seulement si une machine de Turing non-détermiste le décide en un temps polynomial.*

La démonstration est assez simple et consiste tout simplement à convertir un vérificateur en une NTM qui choisie de manière non-déterministe le certificat. Pour prouver l'autre sens, il faut créer un vérificateur qui simule une NTM, en choisissant comme preuve le branche aboutissant sur la feuille «accepter» de celle-ci.

Nous pouvons maintenant définir la classe de complexité temporelle non-déterministe, $NTIME(t(n))$ de manière équivalente à $TIME(t(n))$.

Définition 1.10. $NTIME(t(n)) := \{L | L$ est un langage tel qu'il existe une NTM le reconnaissant en un temps d'exécution $O(t(n)).\}$

Corollaire 1.1. $\text{NP} = \bigcup_k NTIME(n^k)$.

Nous devons maintenant faire quelques remarques. Premièrement, il est clair que P$\subseteq$NP, et deuxièmement, la question capitale est , est-ce que P=NP ? La conjecture actuelle est que non. Tout le reste de cette théorie est basée là-dessus, car nous tentons principalement reconnaître les langages de NP qui ne sont pas dans P. Mais quelle est la différence majeure entre les problèmes de P et de NP ? Simplement, nous pouvons dire qu'étant donné un langage A et une entrée w, P est la classe des langages dont l'appartenance de w à A peut être rapidement décidée. Tandis que NP est la classe des langages dont l'appartenance de w à A est rapidement vérifiable. Nous allons maintenant présenter les notions de réduction en un temps polynomial et de complétude ainsi qu'un problème de NP utilisant ces concepts.

Définition 1.11. Une fonction $f := \Sigma^* \rightarrow \Sigma^*$ est une fonction **calculable en temps polynomial** s'il existe une TM M qui s'arrête avec uniquement $f(w)$ sur sa bande, lorsqu'elle reçoit w en argument.

Définition 1.12. Un langage A est **réductible en temps polynomial** vers un langage B, noté $A \leq_P B$, s'il existe une fonction f calculable en temps polynomial, telle que pour toute entrée w, $w \in A \iff f(w) \in B$. On appelle alors f la **réduction polynomiale** de A vers B.

Soit les langages A et B ainsi qu'une entrée w dont nous voulons tester l'appartenance à A, il suffit de vérifier si $f(w)$ appartient à B. Forcément, le langage B est alors au moins plus complexe que le langage A, à un polynôme près. Immédiatement nous obtenons que si $A \notin$ P alors $B \notin$ P. L'intérêt derrière l'introduction d'une telle notion se retrouve dans la définition et le théorème suivant.

Définition 1.13. Soit C une classe de complexité et soit un langage A. Nous dirons que A est **C-complet** si :

1. $A \in$ C
2. $\forall B \in$ C, il existe une réduction polynomiale de B vers A.

Notons qu'un langage C-complet est le plus difficile de sa classe étant donné que tous les autres problèmes s'y réduisent. De plus, il découle directement des définitions 1.12 et 1.13 que si un langage A est C-complet et qu'il existe un langage B tel que $B \in$ C et $A \leq_P B$, alors B est également C-complet.

Théorème 1.6. *Soit B un langage, $D \subseteq C$ des classes de complexité. Si B est C-complet et que $B \in D$, alors $C = D$.*

Ce théorème est une autre conséquence directe des définition 1.12 et 1.13. Son importance est immédiate lorsque nous pensons à la conjecture sur laquelle repose toute la théorie actuelle, soit que P$\neq$NP. Afin de montrer qu'elle est fausse, il suffit donc de trouver un problème NP-complet et de montrer qu'il appartient à P. Présentons à présent un langage très naturel en informatique ayant des propriétés intéressantes, SAT. Formellement, $SAT = \{\langle\phi\rangle | \phi$ est une formule booléenne satisfiable$\}$.

Théorème 1.7. *SAT est NP-complet [7]*

La démonstration de ce théorème n'est pas très compliquée mais technique. Elle se fait en deux parties et nous exposons ici une idée de preuve. Il faut d'abord montrer que SAT appartient à NP. Il suffit, par le théorème 1.5 de choisir de manière non-déterministe des valeurs pour les variables apparaissant dans la formule ϕ que nous voulons vérifier, puis de tester si cette assignation satisfait ϕ. Ensuite il faut montrer que tout langage appartenant à NP est réductible en temps polynomial vers SAT. C'est construction n'est pas très difficile mais afin d'être rigoureuse elle nécessite plusieurs détails techniques. En gros, nous construisons pour une entrée w une formule booléenne ϕ composée d'implications, chacune représentant une étape de la NTM. L'existance d'une branche de la NTM acceptant w devient alors un problème de satisfiabilité de ϕ.

En pratique, nous travaillons avec un problème équivalent à SAT, donc également NP-complet, mais un peu plus simple à manipuler, qui est $3-SAT$. Nous le définissons ainsi :
$SAT = \{\langle\phi\rangle | \phi$ est une formule booléenne satisfiable composée telle que :
$\phi = \bigvee_{1\leq i\leq k} (x_{1_i} \wedge x_{2_i} \wedge x_{3_i})$, où $k \in \mathbb{N},\ 1 \leq i \leq k$, sont des variables booléennes$\}$.

Il est évident que $3 - SAT \leq_P SAT$, et donc que $3 - SAT \in$ NP. Pour montrer qu'il est NP-complet, il suffit de modifier de manière astucieuse l'algorithme utilisé dans la démonstration du théorème 1.7 en ajoutant des variables supplémentaires. Nous retrouvons cette modification entre autre dans le livre de Sipser [7]. Ou plus simplement il est assez facile de démontrer que $SAT \leq_P 3 - SAT$.

1.3 Complexité spatiale

Dans la section précédente nous avons classé les langages selon leur vitesse de décidabilité, ou de vérification. Le temps est bien sur une contrainte essentielle lorsque nous regardons les applications possibles, mais ce n'est pas la seule. Logiquement, en plus d'avoir le temps nécessaire afin d'exécuter un certain nombre d'opérations sur un ordinateur, il faut également avoir

l'espace nécessaire. Commençons donc par définir la complexité spatiale puis introduisons les classes de complexité spatiale.

Définition 1.14. Soit M une TM qui accepte ou rejette toutes les entrées. La **complexité spatiale** de M est la fonction $f : \mathbb{N} \to \mathbb{N}$, où $f(n)$ est le nombre maximal de cases sur la bande de la machine de Turing qui sont utilisées lorsque l'entrée est de taille n. Si M est une NTM qui accepte ou rejette toutes les entrées sur chacune de ses branches, alors sa complexité spatiale est le maximum de cases utilisées sur chacune de ses branches. Nous utiliserons la notation asymptotique grand «O» afin d'estimer la complexité spatiale des machines de Turing.

Définition 1.15. Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction. Les **classes de complexité spatiale**, notées $SPACE(f(n))$ et $NSPACE(f(n))$ sont :

$$SPACE(f(n)) = \{L | L \text{ est un langage décidé par une TM}\}$$
$$NSPACE(f(n)) = \{L | L \text{ est un langage décidé par une TM}\}$$

Rappelons-nous du problème SAT que le théorème 1.7 classe dans NP-complet. Nous allons maintenant donner un simple algorithme montrant que SAT appartient également à la classe $SPACE(n)$. L'espace est donc quelque chose de beaucoup plus maniable que le temps, car nous pouvons le réutiliser.

Exemple 1.2. *Soit M = «à l'entrée $\langle\phi\rangle$ où ϕ est une formule Booléenne :*

- *Pour chaque assignement de valeur de vérité aux variables $x_1, \ldots, x_m$ de ϕ :*
- *Évaluer ϕ pour une assignation en particulier*
- *Si ϕ est évaluée au moins une fois comme 1, accepter, sinon rejeter.»*

La machine M prend un espace linéaire car à chaque itération elle réutilise la même partie de la bande. De plus, la machine ne doit enregistrer que la valeur de vérité de chacune des variables, ce qui peut être fait dans un espace de $O(m)$, mais comme il y a au plus n variables, la longueur de l'entrée, la machine roule sur un espace de $O(n)$.

Alors qu'il semble normal de penser que l'espace utilisé pour le calcul d'une NTM sur une machine déterministe prendra un espace considérable, étant donné que, par le théorème 1.3, le temps d'exécution est à priori exponentiel. Or, une des premières découvertes dans ce champ d'étude, due à Savitch, est qu'en faît les deux espaces sont équivalents. Voici ce théorème ainsi qu'une idée de la preuve.

Théorème 1.8. ***Théorème de Savitch*** *Pour toute fonction* $f : \mathbb{N} \to \mathbb{N}$, *où* $f(n) > n$, $NSPACE(f(n)) \subseteq SPACE(f^2(n))$.

Soit une NTM utilisant un espace $O(f(n))$. La première approche afin de la simuler sur une machine déterministe serait de regarder directement chacun des chemins que peut prendre une entrée w dans la NTM. Cette méthode est malheureusement trop simple et donne une borne largement supérieure à notre objectif de $O(f^2(n))$. Procédons donc différement. Recevons en entrée deux configurations que peut avoir la NTM, disons c_1 et c_2 ainsi qu'un nombre naturel t. Nous voulons alors tester si la machine peut se rendre de c_1 à c_2 en un maximum de t étapes. Si nous réussissons à résoudre ce problème alors nous pouvons déterminer si l'entrée est acceptée ou non en considérant c_1 comme l'étape de départ puis c_2 comme la configuration qui est acceptée. L'algorithme déterministe et récursif permettant de résoudre ce problème fonctionne ainsi. Nous commençons par chercher un état intermédiaire, c_m compris entre c_1 et c_2. Puis nous regardons si nous pouvons l'atteindre à partir de c_1 en moins de $t/2$ étapes et si nous pouvons nous rendre de c_m à c_2 en également moins de $t/2$ étapes. Bien sur, pour chacun des tests nous réutilisons le même espace.
Chaque niveau de récursion de cette algorithme utilise $O(f(n))$ cases pour conserver la configuration de chaque itération, étant donné que la machine fonctionne en un espace de $O(f(n))$. De plus, il est facile de montrer que la profondeur de chacune des branches est $\log(t)$, où t est le temps maximale que prend la NTM pour calculer une de ses branches. Sur une TM, nous avons par le théorème 1.3 que $t = 2^{O(f(n))}$. L'espace utilisé est alors celui pour le calcul d'une branche sur une TM, multiplié par celui nécessaire par

pour le calcul d'une itération, donc de $\log(t) \times O(f(n)) = O(f^2(n))$.

Définissons à présent une classe de complexité spatiale très importante qui nous permettra de faire un commentaire sur la hiérarchie des classes de complexité.

Définition 1.16. PSPACE est la classe des langages qui sont décidables par une machine de Turing déterministe fonctionnant en temps polynomial. Formellement :

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$$

Nous pourrions définir de manière analogue à NP une classe NPSPACE comme l'union des NSPACE(n^k), mais le théorème de Savitch montre directement que NPSPACE = PSPACE. De plus, l'exemple 1.2 montre que SAT, un problème NP-complet, est compris dans PSPACE. Donc, NP $\subseteq$ PSPACE et en résumé nous avons : P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE.

1.3.1 L et NL

Alors que pour la complexité temporelle nous ne pouvons avoir une borne inférieure à n, car la machine doit au moins lire l'entrée, nous pouvons en utilisant une définition différente de TM, considérer les machines utilisant un espace sous-linéaire. Donc, nous considérerons les machines de Turing, déterministes ou non, comme ayant 2 bandes. Une d'entrée que la machine ne peut que lire, et une deuxième bande sur laquelle la machine peut lire et écrire de l'information. Alors, uniquement l'espace de la deuxième bande est considéré. Introduisons donc tout de suite les deux classes de complexité sous-linéaire.

Définition 1.17. L est la classe des langages décidables en un espace logarithmique sur une TM. Formellement :

$$\text{L} = \text{SPACE}(\log(n))$$

Définition 1.18. NL est la classe des langages décidables en un espace logarithmique sur une NTM. Formellement :

$$\mathrm{L} = \mathrm{NSPACE}(\log(n))$$

L'intéret que nous portons aux espaces logarithmique est dû au fait que beaucoup de problèmes très intéressants se retrouvent dans ces classes. Montrons à présent que le problème $PATH$ présenté à l'exemple 1.1 se retrouve dans NL.

Exemple 1.3. *Soit G un graphe contenant m nœuds et soit M une machine de Turing non-déterministe vérifiant si G appartient à $PATH$. M n'enregistre sur sa bande d'écriture que le nœud sur lequel elle se trouve. Elle commence évidemment sur s puis, à chaque itération, choisi de manière non-déterministe un nœud parmis ceux sur lesquels la machine peut se déplacer. Elle fait cela jusqu'à ce qu'elle arrive au nœud t et accepte ou, si elle a fait m itérations, elle rejette l'entrée. Donc $PATH \in NL$.*

Tout comme pour les contraintes de temps, nous aimerions pouvoir définir des équivalences dans la complexité des problèmes. Bien sur, comme ici nous travaillons sur des espaces logarithmiques, cela n'aurait pas de sens de considérer des fonctions de transformations fonctionnant en un temps polynomial. Voici donc comment nous définissons le fait qu'un langage nécessitant un espace logarithmique est réductible vers un autre langage.

Définition 1.19. Un **transformateur d'espace logarithmique** est une machine de Turing à 3 bandes. La première pouvant uniquement être lue et contenant l'entrée. La machine peut lire et écrire sur la deuxième, la bande de travail. Et finalement, la troisième est la bande de sortie sur laquelle la machine ne peut qu'écrire. La bande de travail peut contenir au maximum $O(\log(n))$ caractères. Un transformateur d'espaces logarithmiques M calcul une fonction $f := \Sigma^* \to \Sigma^*$, où, étant donné une entrée w, $f(w)$ est la chaîne de caractères se retrouvant sur la bande de sortie. f est appelée une **fonction de calcul d'espace logarithmique**. Soit A et B des langages. Nous dirons

que A est **réductible en espace logarithmique** vers B, noté $A \leq_L B$, s'il existe une fonction de calcul d'espace logarithmique f telle que pour toute entrée w, $w \in A \iff f(w) \in B$.

À présent nous pouvons définir la NL-complétude.

Définition 1.20. Un langage A est NL-complet si :

1. $A \in NL$
2. Pour tout langage $B \in NL$, $B \leq_L A$

Retournons maintenant au problème $PATH$. L'exemple 1.3 montre qu'il est dans NL. Or, $PATH$ est en fait NL-complet. La preuve est plutôt simple et nous n'en exposerons que l'idée.

Théorème 1.9. *$PATH$ est NL-complet.*

Nous devons donc montrer que tout langage A dans NL est logarithmiquement réductible à $PATH$. Pour ce faire nous utiliserons la machine M fonctionnant ainsi. Pour toute entrée w, M construit un graphe où chaque nœud représente une des configuration de la NTM étant donnée l'entrée w. Un nœud est connecté au suivant si la NTM de A peut se rendre du premier au deuxième en une étape. La machine M accepte donc l'entrée s'il existe un chemin menant du nœud correspondant à la configuration de départ jusqu'au nœud représentant la configuration d'acceptation.

Cela nous emmène directement à l'inclusion suivante : NL $\subseteq$ P et donc
L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE.

1.4 Parallélisation

1.4.1 Circuits Booléens

En informatique,les problèmes dont les algorithmes sont les plus plus intéressant sont ceux parallélisables, c'est-à-dire que nous pouvons le calculer

sur plusieurs machines simultanément, sauvant ainsi beaucoup de temps. Il est donc important de vérifier si un problème ne serait pas parallélisable avant de tenter de trouver un algorithme pour le résoudre. Premièrement, nous considérons un modèle facile à simuler et réaliste par rapport à nos besoins en théorie de la parallélisation. Un modèle couramment utilisé est celui des machines à accès aléatoire en parallèle, PRAM. Pour la simplicité du travail, nous considérerons ici un cas particuler, soit les circuits Booléens qui de manière intuitive représentent bien le fonctionnement d'un ordinateur.

Définition 1.21. Un **cicuit Booléen** est une collection de **portes** et d'**entrées** connectées par des **fils**, les cycles ne sont pas permis. De plus, chaque porte a trois possibilités, soit la fonction AND, soit OR soit NOT, selon les définitions habituelles en algèbre Booléenne. Les variables d'entrées peuvent donc uniquement prendre les valeurs $\{0,1\}$. De plus, une des portes est désignée comme étant la **porte de sortie**. Le circuit accepte l'entrée w si la valeur de la porte de sortie est 1, si c'est 0 l'entrée est rejetée.

Exemple 1.4. *La fonction* $par_4 : \{0,1\}^4 \to \{0,1\}$ *prend 4 variables en entrée et détermine si il y a un nombre impair de* 1 *parmis les celles-ci. La figure 1.2 donne le schéma du circuit. Nous pouvons remarquer que chaque ligne ne considère que les lignes au dessus. Si chaque porte est un processeur, en partant du haut, tout ceux sur la même ligne peuvent s'exécuter en même temps.*

Il est aisé de voir qu'un circuit ne peut convenir qu'à une entrée d'une certaine longueur, ce qui n'est pas suffisant pour pouvoir vérifier un langage ayant des entrées de longueurs variables. Définissons donc les familles de circuits afin de pouvoir faire le lien entre les circuits et les langages.

Définition 1.22. Une **famille de circuits** C est une liste infinie de circuits, $\{C_0, C_1, \ldots\}$ où C_n à n variables d'entrées. Nous dirons que C décide un langage A sur le domaine $\{0,1\}$ si pour toute chaîne de caratères w de longueur n :

$$w \in A \iff C_n(w) = 1.$$

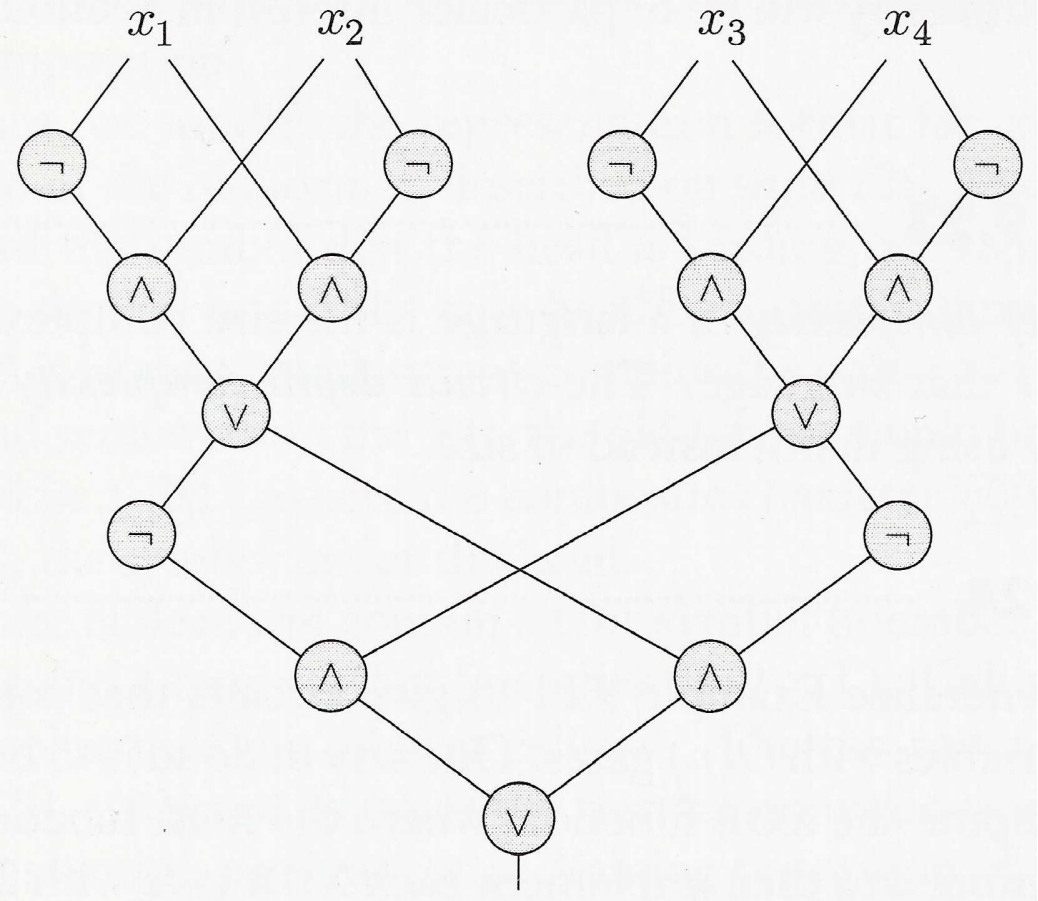

FIG. 1.2: Circuit Booléen pour vérifier la parité de 4 variables

La **taille** d'un circuit est le nombre de portes qu'il contient. La **profondeur** d'un circuit est la longueur, le nombre de fils maximal, entre une variable d'entrée et la porte de sortie.

Définition 1.23. 1. Deux circuits sont **équivalents** si, prenant la même entrée, la sortie est équivalente.

2. Soit $C := \{C_0, C_1, \ldots\}$ une famille de circuits, D_i le circuit de plus petite taille équivalent à C_i.

   La **complexité de la taille d'une famille de circuits**, ou simplement **la complexité d'un circuit**, est la fonction $f := \mathbb{N} \to \mathbb{N}$, où $f(n)$ est la taille de D_n.

   La **complexité de la profondeur d'une famille de circuits** est la fonction $f := \mathbb{N} \to \mathbb{N}$, où $f(n)$ est la profondeur de D_n.

La complexité des circuits d'un langage est intimement liée à la complexité temporelle de celui-ci, comme le montre le théorème suivant. Sa preuve étant très technique nous la laissons de côté dans le présent travail.

Théorème 1.10. *Soit $t := \mathbb{N} \to \mathbb{N}$, une fonction telle que $t(n) \geq n$.*
Si $A \in TIME(t(n))$ alors A est un circuit de complexité $O(t^2(n))$ [7].

Nous pouvons maintenant discuter de parallélisation. Si nous modélisons un programme fonctionnant en parallèle avec un circuit Booléen, nous considérons chaque porte comme étant un processeur et nous définissons donc la **complexité de processeur** d'un tel circuit comme étant son nombre de portes. De façon équivalente la **complexité de temps en parallèle** du circuit est la profondeur de celui-ci. Nous disons que l'algorithme fonctionne en parallèle car chacune des lignes du circuit peut-être calculée simultanément. Le temps nécessaire au calcul de l'algorithme est alors celui nécessaire pour calculer la plus longue branche du circuit. Comme chaque circuit a un nombre fixe de variables qu'il peut prendre en entrée, nous considérerons toujours les familles de circuits. De plus, afin de faire correspondre les circuits Booléens aux modèles de calcul qui sont parallélisables, nous devons introduire une restriction supplémentaire, soit que le circuit soit **uniforme**.

Définition 1.24. Une famille de circuits, $\{C_1, C_2, \ldots\}$ est **uniforme** s'il existe un transformateur d'espace logarithmique T tel que lorsqu'il reçoit 1^n en entrée, n fois le nombre 1, la sortie est le circuit C_n.

Cette condition est nécessaire lorsque nous voulons étudier la parallélisation car il faut s'assurer de pouvoir décrire efficacement la famille de circuits. Ici, nous nous retrouvons donc avec la classe des problèmes les mieux définis car nous étudions leur temps de calcul ansi que l'espace requis pour les calculer. Nous dirons donc que la **taille-profondeur simultanées** d'un langage est au plus $(f(n), g(n))$ s'il existe pour ce langage un circuit Booléen de complexité de taille $f(n)$ et de complexité de profondeur $g(n)$.

1.4.2 NC

Nous allons donc maintenant introduire une famille de classes de complexité utilisant la notion de taille-profondeur simultanée.

Définition 1.25. Soit $i \geq 1$, alors $\mathbf{NC}^i$ est la classe des langages reconnus par une famille uniforme de circuits de taille polynomiale et de profondeur en $O(\log^i(n))$. Alors **NC** est la classe des langages appartenant à NC^i pour un certain i.

Démontrons maintenant quelques inclusions des classes de complexité.

Théorème 1.11. *$NC^1 \subseteq L$.*

Démonstration. Voici un algorithme d'espace polynomial qui reconnait un langage A appartenant à NC^1.
Soit une entrée w de taille n, l'algorithme construit alors le n^{e} circuit de la famille de circuits uniformes de A. Par définition cela prend au maximum un espace logarithmique. Puis, l'algorithme évalue le circuit en cherchant un chemin vers la porte de sortie. Le seul espace nécessaire est celui pour conserver une trace de la position actuelle de l'algorithme ainsi que des résultats partiels obtenus. Comme le circuit a une profondeur logarithmique cela ne nécessite pas plus qu'un espace en $O(\log(n))$. □

Théorème 1.12. *$NC \subseteq P$.*

Démonstration. Étant donné une entrée w de taille n. Il est évident qu'un algorithme polynomial peut simuler le transformateur d'espace logarithmique afin de modéliser le circuit C_n puis l'évaluer. □

Chapitre 2

CSP et l'Algèbre Universelle

Il existe un lien fondamental entre les classes des problèmes de satisfaction de contraintes, et l'algèbre universelle. En effet, une construction très naturelle permet de créer une algèbre $\mathbb{A}_\Gamma$ à partir de n'importe quel problème de satisfaction de contraintes Γ, sur un certain domaine D. Pour cela il faudra développer la notion de clôture d'un problème de satisfaction de contraintes et de polymorphismes. Ceci nous permettra de démontrer le théorème fondamental du chapitre, nous indiquant que pour tout problème de satisfaction de contraintes sur un domaine D, il existe une bijection entre l'algèbre lui correspondant et la clôture de ce problème, sur le même domaine.

2.1 Problèmes de Satisfaction de Contraintes

Définition 2.1. Soit D un ensemble **fini**, soit $n \in \mathbb{N}$, alors l'ensemble de tous les n-tuples éléments de D est noté D^n. Les sous-ensembles de D^n sont des relations n-aires sur D. L'ensemble de toutes les relations n-aires sur un ensemble D est noté $\mathbf{R}_D$. Nous définissons alors **un langage de contrainte** sur un domaine D comme un sous-ensemble de $\mathbf{R}_D$. Un langage de contrainte est noté Γ, le domaine étant toujours sous-entendu dans le contexte.

Définition 2.2. Pour tout domaine D et pour tout langage de contrainte Γ sur ce domaine, le **problème de satisfaction de contraintes**, noté $CSP(\Gamma)$

est composé d'un triplet d'ensembles, et d'un problème de décision combinatoire.

1. Le triplet $\langle V, D, \mathcal{C} \rangle$, où :
   (a) V est un ensemble de variables
   (b) D est le domaine de définition du problème
   (c) $\mathcal{C}$ est un ensemble de contraintes, $C_1, \ldots, C_k$. Chaque $C_i \in \mathcal{C}$ est une paire $\langle s_i, R_i \rangle$, où :
   - s_i est un n_i-tuples de variables, appelé le **rayon de contrainte**.
   - R_i est une n_i-aires relation sur D telle que $R_i \in \Gamma$. On appelle R_i une **relation de contrainte**.
2. Existe-t-il alors une fonction, $\phi : V \to D$, telle que, pour toute contrainte $\langle s_i, R_i \rangle \in \mathcal{C}, s_i := \{v_1, \ldots, v_{n_i}\}$, alors $\{\phi(v_1), \ldots, \phi(v_{n_i})\} \in R_i$?

Définissons maintenant la clôture d'un langage de contraintes. Ce concepte est très important car il donne place à un cadre de résolution beaucoup plus général que l'étude de Γ. L'idée est de considérer toutes les nouvelles contraintes qui peuvent être dérivées à partir de combinaisons des contraintes de Γ.

Définition 2.3. La **clôture d'un langage de contrainte** Γ, noté $\langle \Gamma \rangle$, est définie comme étant l'ensemble des relations qui peuvent être exprimées en utilisant :

- Les relations de Γ
- La relation d'égalité sur le domaine de définition de Γ
- Les conjonctions de différentes relations de Γ ainsi que de la relation d'égalité.
- Des quantificateurs existentiels sur les relations obtenues.

Étudier $\langle \Gamma \rangle$ plutôt que le langage de contrainte lui-même peut sembler complexifier le problème, mais tout l'intéret est dans le fait suivant : il existe une bijection entre l'algèbre dérivée naturellement de Γ et la clôture de celui-ci. De plus le théorème suivant permet de comprendre que la complexité

des différents langages de contraintes, ayant la même clôture, est toujours polynomialement équivalente.

Théorème 2.1. *Pour tout langage de contrainte* Γ *et pour tout sous-ensemble fini* $\Gamma' \subseteq \langle\Gamma\rangle$*, il existe une réduction polynomiale de* $CSP(\Gamma')$ *vers* $CSP(\Gamma)$*.*

Démonstration. Soit Γ', un ensemble fini de relations sur le domaine fini D, où chaque contrainte peut-être représentée par une formule pouvant contenir des relations de Γ avec des conjonctions, des quantificateurs existentiels et la relation d'égalité sur D. Nous voulons maintenant montrer que chaque instance de $CSP(\Gamma')$ est équivalent à une instance de $CSP(\Gamma)$.
En particuler, soit une instance $\langle V, D, \mathcal{C}\rangle \in CSP(\Gamma')$. Alors pour chaque contrainte $\langle s, R\rangle \in \mathcal{C}$ où $s := \{v_1, \ldots, v_l\}$, R peut être représentée par une formule de la forme :

$$R(v_1, \ldots, v_l) = \exists u_1, \ldots, \exists u_m(r_1(w_1^1, \ldots, w_{l_1}^1), \ldots, r_n(w_1^n, \ldots, w_{l_n}^n)),$$

où les $w_i^j \in \{v_1, \ldots, v_l\}$ et les r_i sont soit des relations de Γ soit la relation d'égalité. Voici maintenant l'algorithme de transformation en 4 étapes pour une contrainte de $\mathcal{C}$

1. Ajouter les variables auxiliaires $u_1, \ldots, u_m$ à V en les renommant si nécessaire pour éviter les répétitions.
2. Pour tout $r_j(w_1^j, \ldots, w_{l_j}^j)$ étant la relation d'égalité, remplacer toutes les occurences des $w_i^j, 1 < i \leq j$, par w_1^j et enlever $r_j(w_1^j, \ldots, w_{l_j}^j)$ de $\mathcal{C}$.
3. Pour toutes les relations $r_j(w_1^j, \ldots, w_{l_j}^j)$ restantes après l'étape 2, ajouter $\langle(w_1^j, \ldots, w_{l_j}^j), r_j\rangle$ à $\mathcal{C}$.
4. Enlever $\langle s, R\rangle$ de $\mathcal{C}$.

La première étape permet d'enlever tous les quantificateurs existentiels. La deuxième étape va enlever les relations d'égalités ne laissant donc qu'une suite de conjonctions de contraintes de Γ. Les deux dernières étapes permettent de remplacer chaque contrainte de $\mathcal{C}$ par une relation de contrainte de Γ. Comme chaque représentation d'une relation de contrainte de Γ' est fixée,

toutes ces étapes peuvent se faire en un temps polynomial. Il est évident que l'instance restante appartient à $CSP(\Gamma)$. □

Dans un cadre pratique, un problème devient intéressant s'il existe un algorithme efficace permettant de le résoudre. En théorie informatique, nous définissons ce genre de problèmes comme ceux ayant un algorithme dont le temps d'exécution est borné par un polynôme en fonction de la taille du problème. Pour les CSP, les langages de contrainte, Γ, peuvent être infinis, et donc s'il existe des représentations finies de celles-ci, elles peuvent être de tailles très différentes. Par contre, si Γ est fini, il n'y a aucun problème. Nous allons donc définir la complexité des langages de contrainte Γ de manière à contourner ce problème en ne considérant que les sous-groupes finis de Γ.

Définition 2.4. Un langage de contrainte Γ est dit **maniable**[1] si, pour tout sous-ensemble fini $\Gamma' \subseteq \Gamma$, le problème $CSP(\Gamma')$ peut être résolu en un temps polynomial par rapport à la taille de l'entrée.

Définition 2.5. Un langage de contrainte Γ est dit **NP-complet** s'il existe un sous-ensemble fini $\Gamma' \subseteq \Gamma$ tel que $CSP(\Gamma')$ est NP-complet.

Alors qu'il existe une grande quantité de problèmes computationnels qui ne sont ni maniables, ni NP-complets, nous pouvons montrer que tous les CSP, définis sur des domaines de deux ou trois éléments, sont soit maniables, soit NP-complets, et la conjecture actuelle est que la même classification dychotomique prévaut pour tous les domaines finis, peut-importe leur cardinalité.

2.2 Des CSP à l'Algèbre Universelle

Alors que l'étude des CSP est très complexe, plusieurs articles ont montré que nous pouvions entièrement caractériser la complexité de ceux-ci en

[1] En anglais le mot «tractable»est utilisé, cette traduction provient de [6]

utilisant les propriétés algébriques des relations formant ces CSP. Nous allons montrer comment construire les algèbres correspondantes aux CSP en commençant par définir les polymorphismes.

Définition 2.6. Soif $f := D^m \to D$, une fonction. f **préserve** une k-aires relation $R \in \Gamma$ si $\forall t_1, \ldots, t_m \in R$, où $t_i = (t_i^1, \ldots, t_i^k) \in R$ on a :
$f(t_1, \ldots, t_m) = \big(f(t_1^1, \ldots, t_m^1), \ldots, f(t_1^k, \ldots, t_m^k)\big) \in R$.

Définition 2.7. Nous dirons que f est un **polymorphisme** de Γ si f préserve toutes les relations de celui-ci.

Définition 2.8. Une relation k-aire R est **invariante** sous une fonction $f := D^m \to D$ si pour tout ensemble appartenant à R, par exemple $\{t_1, \ldots, t_m\}$ où $t_i = (t_i^1, \ldots, t_i^k) \in R$, $f(t_1, \ldots, t_m) \in R$.

Définition 2.9.
1. **Pol**(Γ)= $\{f | f$ est un polymorphisme de $\Gamma.\}$
2. **Inv**(F)= $\{R | R$ est invariante sous toutes les opérations de $F.\}$

Le lien intrinsèque entre les opération Pol et Inv avec la clôture de Γ, présenté dans le prochain théorème, est ce qui va nous ammener à construire une algèbre ayant pour opération Pol(Γ) afin d'étudier la complexité de CSP(Γ).

Théorème 2.2. *Pour tout langage de contrainte* Γ *sur un ensemble fini* D, $\langle\Gamma\rangle$ =*Inv(Pol(*Γ*))*.

Afin de simplifier la preuve nous introduisons une définition supplémentaire.

Définition 2.10. Soit Γ un langage de contrainte fini, sur un domain fini D. Pour tout $k \in \mathbb{N}$, le **problème d'indicateur d'ordre** k pour Γ est l'instance de CSP suivante : $\mathcal{P} = \langle V, D, \mathcal{C}\rangle \in CSP(\Gamma)$ où :

- $V = D^k$, soit chaque variable est un k-tuples d'éléments du domaine.
- $\mathcal{C} = \{\langle s, R\rangle | R \in \Gamma$ et $s := \{v_1, \ldots, v_n\}$ est la liste de tous les k-tuples où n est l'arité de R et où pour tout $i \in \{1, \ldots, k\}$ le n-tuple $(v_1^i, \ldots, v_n^i)$ appartient à $R.\}$

Remarquons que les solutions de ce problème sont toutes les fonctions à k arguments appartenant à Pol(Γ).

Démonstration. Commençons par l'inclusion de gauche à droite. Si deux relations ont un même polymorphisme f, alors forcément leur conjonction a le même polymorphisme. De plus, si une relation a comme polymorphisme f, nous pouvons vérifier que les relations dérivées de celle-ci en ajoutant des quantificateurs existentiels ont également f comme polymorphisme. Évidemment, toutes les opérations sont un polymorphisme pour la relation d'égalité. Il en suit que pour toute relation $R \in \langle R \rangle$, $\mathrm{Pol}(R) \supseteq \mathrm{Pol}(\Gamma)$ et donc que $\langle \Gamma \rangle \subseteq \mathrm{Inv}(\mathrm{Pol}(\Gamma))$.
Pour l'autre direction, posons Γ un langage de contrainte sur un domaine fini D. Soit R une n-aire relation quelconque dans Inv(Pol(Γ)). Il faut montrer que nous pouvons exprimer R en utilisant les relations de contraintes de Γ. Soit k le nombre de n-tuples dans la relation R. Construisons le problème d'indicateur $\mathcal{P}$ d'ordre k pour Γ. Choisissons une liste de variables $t := \{v_1, \ldots, v_n\} \in \mathcal{P}$ telle que tout n-tuples $(v_1^i, \ldots, v_n^i)$, $i \in \{1, \ldots, k\}$, est un élément différent de notre relation R. Considérons maintenant la contrainte $R_t := \big\{\{f(v_1), \ldots, f(v_n)\} | f \text{est une solution de } \mathcal{P}.\big\}$ Par la remarque au bas de la définition 2.10, les f sont donc les k-aires polymorphismes de Γ. Dans ce cas, les k-projections, celles qui retournent la k^{e} composante en font parties, par définition de t. Et donc, pour tout choix de t, chaque projection donne un n-tuple différent inclus dans R_t et donc $R \subseteq R_t$. Inversement, par le choix de R nous avons que chaque polymorphisme de Γ préserve R, et donc que chaque élément de R_t appartient à R. Nous avons donc montré que pour tout $R \in \mathrm{Inv}(\mathrm{Pol}(\Gamma))$, R peut être exprimée par des contraintes de Γ et donc que $\mathrm{Inv}(\mathrm{Pol}(\Gamma)) \subseteq \langle \Gamma \rangle$. □

Nous pouvons maintenant définir l'algèbre associée au problème $CSP(\Gamma)$ comme suit :

Définition 2.11. Soit Γ un langage de contrainte et D un domaine fini.

L'algèbre associé au problème $CSP(\Gamma)$ est :

$$\mathbb{A}_\Gamma := \langle D, F \rangle.$$

Où D est son domaine et $F = \mathrm{Pol}(\Gamma)$.

2.3 Fonctions de Mal'tsev

Jusqu'à présent, nous avons montré qu'afin d'étudier la complexité d'un CSP, nous pouvions construire une algèbre puis regarder sa structure afin de déterminer la complexité du CSP. Dans les faits, nous procédons plutôt dans l'autre sens. Nous étudions une algèbre $\mathbb{A}_\Gamma := \langle D, F \rangle$, ayant des propriétés interessantes puis regardons l'ensemble $\mathrm{Inv}(\mathbb{A}_\Gamma)$. Un des exemples intéressants est celui de l'algèbre dont l'ensemble F est composé de fonctions de Mal'tsev, sur un domaine fini A quelconque.

Définition 2.12. Une fonction $\phi := A^3 \to A$ telle que pour tout $x, y \in A$, $\varphi(x, y, y) = \varphi(y, y, x) = x$ est appelée **une fonction de Mal'tsev**.

L'intérêt de telles fonction est qu'un grand nombre de CSP importants leurs sont associés, par exemple les systèmes d'équations linéaires. Soit R l'ensemble solution d'un sytème d'équations linéaires. Nous pouvons décrire R par : $R := \{x \mid A \cdot x = b\}$. Alors R est invariant par rapport à la fonction $\varphi(x, y, z) = x - y + z$, qui est une fonction de Mal'tsev puisque $\varphi(x, y, y) = x - y + y = x = y - y + x = \varphi(y, y, x)$. Pour montrer que φ est un polymorphisme de R il suffit de remarquer que pour tous $x, y, z \in R$: $A \cdot \varphi(x, y, z) = A \cdot (x - y + z) = A \cdot x - A \cdot y + A \cdot z = b - b + b = b$ $\Longrightarrow \varphi(x, y, z) \in R$.
Il existe une autre raison pour laquelle les fonctions de Mal'tsev sont si intéressantes. Il existe un type d'algorithme, dit de propagation locale, qui est utilisé afin de résoudre presque tous les CSP solubles en un temps polynomial. Ces algorithmes trouvent des solutions pour un certain nombre de contraintes des CSP puis les modifient en rajoutant les autres contraintes.

La majorité des CSP invariant par rapport à une fonction de Mal'tsev ne pouvent être résolus d'une telle manière. Les sous ensembles des systèmes d'équations linéaires, en particulier, possède automatiquement une infinité de réponses, en analyse réelle, s'il y a moins de contraintes que de variables. Nous pouvons montrer que touts les CSP invariants par rapport à une fonction de Mal'tsev sont maniables. Nous allons maintenant énoncer le théorème puis donner une idée de la preuve.

Théorème 2.3. *Soit φ, une fonction de Mal'tsev. Alors CSP(Inv(φ)) est résoluble en temps polynomial.*

Afin de prouver le théorème, nous devons montrer que nous pouvons déterminer pour toute instance $\mathcal{P}$ de CSP(Inv(φ)), $\mathcal{P} := (\{v_i, \ldots, v_n, A, \{C_1, \ldots, C_m\})$, si elle a une solution en un temps polynomial. Pour cela, pour tout $0 \leq l \leq m$, définissons $\mathcal{P}_l := (\{v_i, \ldots, v_n\}, A, \{C_1, \ldots, C_l\})$ et $R_l := \{\big(s(v_1), \ldots, s(v_n)\big) : s$ est une solution de $\mathcal{P}_l\}$. Il est évident que $R_0 = A^n$ car il est composé de tous les n-tuples sur l'ensemble A sans aucune contraintes. Notre algorithme va calculer une représentation compacte de R_0, noté R'_0, puis à partir de celle-ci et de la contrainte C_1, évaluer R'_1, une représentation compacte de R_1. Ainsi, de manière récursive notre algorithme va se rendre jusqu'à R'_m. Si ce dernier est vide, alors l'instance $\mathcal{P}$ n'a pas de solution, sinon oui. Appelons NEXT la procédure qui prend en entrée R'_l et la contrainte C_{l+1}. La complexité et la validité de l'algorithme ne dépendent que de cette procédure. Afin de la créér il a fallu une grande idée.

Celle-ci, est d'avoir réussi à créér une représentation équivalente à chaque R_i, R'_i, telle que $R'_i \subseteq R_i$ et ayant une même signature. La signature est un ensemble de triplets, $(i, a, b) \in \{1, \ldots, n\} \times A^2$, qui décrit un ensemble de couples de R_i. C'est cette représentation qui permet à l'algorithme de fonctionner en un temps polynomial. Étant donné une relation R_l, ça représentation compacte R'_l, ainsi que la contrainte C_{l+1}. NEXT va premièrement augmenter R'_l afin qu'elle soit invariante par rapport à φ. Cette transformation ne change pas l'inclusion $R'_l \subseteq R_l$ car chaque élément du premier appartient au second, et comme R_l est invariant par rapport à φ, pour tout

$t_1, t_2, t_3 \in R'_l$, $\varphi(t_1, t_2, t_3) \in R_l$. Puis, pour tout les triplets (i, a, b), elle va extraire tous les objets du nouvel ensemble vérifiant la contrainte supplémentaire. C'est cet ensemble qui est R'_{l+1}.

Conclusion

Les possibilités de recherche dans tous ces domaines sont vastes car il subsiste une grande quantité d'interrogations. Sur les classes de complexité, même si nous connaissons plusieurs théorèmes important. Rappelons nous en particulier l'inclusion des classes suivante :

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE}$$

et également le fait que $\mathrm{NC} \subseteq \mathrm{P}$. Alors qu'il semble y avoir une hiérarchie bien définie, la conjecture à toujours été que P était différent de NP. Par contre, comme $L \neq PSPACE$, nous savons qu'une des inclusions est stricte. Nous avons exposé plusieurs outils permettant de travailler sur de telles questions, surtout les notions de complétude et de réduction polynomiale et l'exemple SAT.

Par la suite, l'étude des CSP se faisait toujours dans un cadre général. Nous voulions des solutions pour des ensembles de problèmes et pour cela nous étudions l'algèbre associée à chaque ensemble. Une telle méthode est merveilleuse pour les résultats théoriques mais nous ne devons pas oublier qu'en pratique, lorsque nous devons trouver une solution, nous ne pouvons pas juste nous dire qu'un problème n'est pas maniable et donc le laisser de côté. Il est important de voir si nous ne sommes pas dans un cas particulier, un des cas du théorème de dichotomie de Schaefer si nous avons un problème de satisfiabilité booléen par exemple, ou si la taille de l'entrée est assez petite pour que, même si l'algorithme est inefficace, avoir la réponse aisément. Finalement, nous ne devons pas oublier que nous démontrons des théorèmes d'existence d'algorithmes. Dans le cas des systèmes d'équations linéaires, nous connaissions déjà l'algorithme de réduction de Gauss, qui prend un temps en $O(n^3)$.

En fait, cet algorithme est parallélisable et se retrouve donc dans NC. Pour faire suite à ce travail, une idée intéressante serait de modifier la démonstration du théorème 2.3 afin de montrer qu'en faite, CSP(Inv(φ)) est dans NC.

Bibliographie

[1] D. Cohen and P. Jeavons. *Handbook of Constrain Programming*. Elsevier, 2006.

[2] A. Bulatov & V. Dalmau. A simple algorithm for mal'tsev constraints. *SIAM J. Comput*, 36(1) :16–27, 2006.

[3] Peter Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200 :185–204, 1998.

[4] Peter Jeavons. Constraints algebra. *International Workshop on Mathematics of Constraint Satisfaction : Algebra, Logic and Graph Theory*, 2006.

[5] A. Bulatov & P. Jeavons & A. Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM J. Comput*, 34(3) :720–742, 2005.

[6] J. Mansion and Al. *Harrap's Standart French and English Dictionnary*. Harrap London and Clark Irwin and Compagnie Associated, 1962.

[7] Michael Sipser. *Introduction to the Theory of Computation*. PWS, 1997.