République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Akli Mohand Oulhadj - Bouira -

Faculté des Sciences et des Sciences Appliquées
Département de Mathématiques

Mémoire de Master

Filière : Mathématiques

Spécialité : Recherche Opérationnelle

Thème

Étude sur la théorie de la décidabilité et complexité algorithmique

Présenté par :

- **Boussada Hafid**
- **Hadiouche Slimane**

Devant le jury composé de :

Président	M^r**Birouche Madjid**	**MAA**	**U. A/M/O Bouira.**
Encadreur	M^r **Hamid Karim**	**MAA**	**U. A/M/O Bouira.**
Examinatreurs	M^r **Demmouche Nacer**	**MCB**	**U. A/M/O Bouira.**
	M^r **Hamdouni Omar**	**MAA**	**U. A/M/O Bouira.**

2020/2021

Théorie de la décidabilité et complexité algorithmique

Résumé
On s'intéresse dans ce mémoire à l'étude de la théorie de la décidabilité et à la classification des problèmes de décision en fonction de la complexité algorithmique. Des classes de problèmes **P**, **NP**, **NP-complet**, etc. sont dégagées. La démarche consiste à partir de la notion théorique de la machine de Turing pour définir d'une manière rigoureuse la notion de décidabilité et les différentes classes de complexité et souligner le lien entre elles.

Mots clés : Problème de décision, machine de Turing, algorithme, complexité, classe de problèmes.

Decidability theory and algorithmic complexity

Abstract
In this thesis, we are interested in the study of the theory of decidability and the classification of decision problems according to algorithmic complexity. Classes of problems **P**, **NP**, **NP-complet**, etc. are cleared. The approach consists of starting from the theoretical notion of the Turing machine to define in a rigorous way the notion of decidability and the different classes of complexity and to underline the link between them.

Key words : decision problem, Turing machine, algorithm, complexity, problem class.

Remerciements

Nous tenons à remercier :

Allah *de nous avoir donné la patience et la volonté pour accomplir ce travail.*

Nos remerciements s'adressent également à :

Notre promoteur ***Mr K.Hamid*** *pour ses conseils, ses orientations pour nous avoir transmis les renseignements nécessaires à la réalisation de ce travail, et son aide durant l'encadrement.*

Nous remercions également :

Les membres de jury, Président : ***Mr M.Birouche*** *et les Examinateurs :* ***Mr N.Demmouche*** *et* ***Mr O.Hamdouni*** *pour l'honneur qu'ils nous font en acceptant de juger, de lire et d'évaluer ce mémoire.*

Nous tenons également à remercier

Tous les enseignants de notre département qui nous ont accompagnés au cours de notre formation et à toute l'équipe pédagogique de la Faculté des Sciences et Sciences Appliquées .

Enfin, nous remercions toute personne ayant contribué de près ou de loin à la réalisation de ce travail.

Table des matières

Historique

Nous commençons notre travail par un bref exposé historique qui permet de comprendre l'évolution de la discipline de la théorie de la décidabilité et de la complexité algorithmique et le long chemin qui a permis la formalisation des notions liées à notre thème. L'essentiel de ce rappel historique est dû à Sylvain Perifel [1].

Les premières traces d'algorithmes ont été retrouvées chez les Babyloniens (l'actuel Irak) au deuxième millénaire avant notre ère et étaient principalement des méthodes de calcul pour le commerce et les impôts. Il faut attendre le troisième siècle avant J.-C. en Grèce pour l'apparition du fameux algorithme d'Euclide pour le calcul du pgcd : on peut considérer qu'il s'agit du premier algorithme « moderne » et il est tout à fait remarquable qu'il soit toujours utilisé de nos jours. Mille ans plus tard, au IX^{e} siècle ap. J.-C., Al Khuwarizmi, un mathématicien perse (actuel Iran), publie un ouvrage consacré aux algorithmes : l'étymologie du terme « algorithme » vient du nom de ce mathématicien. On commence en effet à étudier les algorithmes en tant que tels, mais il faudra encore 800 ans avant que l'Occident continue cette étude. Aux XVII^{e} et au XVIII^{e} siècles, des savants comme Pascal ou Leibniz construisent des machines à calculer mécaniques (la Pascaline en 1642) ou des automates, ouvrant ainsi la voie à l'automatisation du calcul et à la recherche d'algorithmes efficaces. En 1837 , Wantzel résout le problème de savoir quelles longueurs il est possible de construire à la règle et au compas : il s'agit de connaître la capacité des algorithmes dont les opérations de bases sont celles de la règle et du compas. Ce sont les prémices de la calculabilité.

La volonté de formalisation des mathématiques à la fin du XIX^{e} siècle et au début du XX^{e} amène la notion d'axiomes et de systèmes de preuve, ce qui permet d'envisager une « automatisation » des mathématiques. Dans ce cadre, il est alors naturel de se demander si l'on peut « tout » résoudre par algorithme.

Hilbert pose en 1900 dans son dixième problème la question de trouver un algorithme déterminant si une équation diophantienne a une solution. Malgré des voix plus sceptiques, il est alors clair pour Hilbert qu'un tel algorithme doit exister. Il va plus loin en 1928 avec son (« problème de décision ») : il demande un algorithme capable de décider si un énoncé mathématique est vrai.

En l'absence d'une solution positive, de nombreux mathématiciens tentent de formaliser ce qu'on entend par " algorithme " afin de mieux cerner la question. Plusieurs formalismes voient le jour dans les années 1930 et tous sont prouvés équivalents, notamment le $\lambda-$ calcul de Church et la machine de Turing. Ce dernier modèle, par sa simplicité et son évidence, emporte l'adhésion

et les algorithmes ont maintenant une définition formelle. Or c'est un choc, ou au moins une surprise, qui attend Hilbert : Church en 1936 avec son modèle, et Turing en 1937 avec le sien, montrent qu'il n'existe pas d'algorithme pour son "problème de décision" : c'est le début de la calculabilité, qui s'attache comprendre ce que sont capables de réaliser les algorithmes. Des techniques de plus en plus sophistiquées permettent d'obtenir d'autres résultats d'indécidabilité (le fait pour un problème de ne pas avoir d'algorithme), comme par exemple Matiyasevich en 1970 qui pose la dernière pierre de la preuve qu'il n'existe pas d'algorithme pour décider si une équation diophantienne a une solution (dixième problème de Hilbert). Mais bien plus que l'indécidabilité, on généralise le modèle de calcul, on compare les différentes classes, on étudie l'aléatoire, etc. Ces considérations, bien que très théoriques, ont contribué à la construction des premiers ordinateurs au début des années 1940 , qui sont finalement très proches de la machine de Turing. Et c'est en quelque sorte le développement de ces ordinateurs qui donne naissance à la théorie de la complexité.

La calculabilité s'attache à connaître ce qu'on peut résoudre par algorithme quel que soit le temps d'exécution. Or dans les années 1960, si les gros ordinateurs se sont diffusés, il n'en reste pas moins qu'ils sont très lents et ne disposent pas de beaucoup de mémoire. Les chercheurs s'intéressent donc naturellement à l'efficacité des algorithmes : quel algorithme puis-je faire tourner sur cette machine sans que le résultat mette un an à arriver ? Quel algorithme puis-je faire tourner sans dépasser la capacité de la mémoire de la machine ? On trouve des réflexions théoriques très profondes de ce genre dans une lettre de Gödel à von Neumann en 1956, qui demande s'il existe un algorithme quadratique pour le problème SAT : tous les ingrédients de la question « P = NP? » sont déjà présents dans son esprit. Malheureusement, von Neumann mourant n'a pas pu prendre cette lettre en considération et elle n'a été retrouvée que trente ans plus tard. Si l'on exclut cette lettre, la première référence explicite aux algorithmes fonctionnant en temps polynomial comme définition d'algorithmes " efficaces " se trouve dans les articles de 1965 de Cobham et d'Edmonds . Puis le papier de Hartmanis et Stearns lance réellement le domaine de la complexité en montrant que certains problèmes ne peuvent pas être résolus en un temps donné (théorèmes de hiérarchie).

Quant à la question « $P = \mathrm{NP}$? », elle prend de l'importance dès le début des années 1970 avec l'article de Cook en 1971 (ou, de l'autre côté du rideau de fer, l'article de Levin en 1973) montrant la NP-complétude de SAT, et avec d'autres résultats de NP-complétude en 1972 par Karp pour de nombreux problèmes ayant une réelle importance.

Introduction

Either mathematics is too big for the human mind,
or the human mind is more than a machine.
Kurt Gödel

Le thème de *décidabilité et complexité algorithmique* est un sujet qui est à la frontière entre les mathématiques et l'informatique. Il s'agit d'un sujet central en informatique théorique ou encore en mathématiques appliquées à l'informatique.

L'objet de notre travail porte sur la présentation des questions liées à la notion de *décidabilité*, qui donne un cadre pour déterminer si un problème mathématique peut être résulu au moyen d'un algorithme (autrement dit : par un ordinateur). Puis, dans le cas échéant, on s'intéresse à la qualité de l'algorithme dans le cadre de la théorie de la complexité. Cette dernière, nous permet de déterminer si le problème en question peut être résolu par un algorithme *efficace*, c'est-à-dire un algorithme s'exécutant en un temps raisonnable et utilisant une quantité raisonnable de mémoire.

Lorsque l'on dispose d'algorithmes pour résoudre un problème, ceux-ci peuvent se révéler inutilisables dans la pratique parce qu'ils requièrent trop de temps même pour les exemples simples. Un objectif est aussi de classifier les problèmes selon leur complexité en temps, mais il est davantage question de distinguer les complexités polynomiales. On ne considère naturellement que les problèmes décidables. L'objectif de notre travail est de faire une étude de la complexité et des classes des problèmes via la notion de *machine de Turing* qu'on va supposer implicitement finies, et en s'aidant de *l'algèbre de Boole* qui sera un outil précieux pour la compréhension de certaines notions.

Notre travail comprend est divisé en quatre chapitres, précédés par la présente introduction, et après la conclusion y est mise une bibliographie des ouvrages et références qui nous ont permis de réaliser ce mémoire.

Nous commençons le premier chapitre par la pésentation de la théorie de la décidabilité en donnant un sens précis à un problème décidable [1]. Pour ceci, nous allons nous appuyer sur la notion incontournable de machine de Turing. C'est grâce à cette notion qu'on parviendra à des définitions claires et à un formalisme rigoureux. L'importance théorique de la thèse de Church y est soulignée [3].

Le deuxième chapitre est consacré à la mesure de la complexité des algorithmes. L'impact pratique entre une complexité polynomiale et une complexité exponentielle est illustré par des exemples, et de là se dégage le besoin permanent d'améliorer la qualité des algorithmes des

différents problèmes qu'on rencontre en mathématiques. Ainsi, nous parviendrons à distinguer deux grandes classes de problèmes [6]. La classe des problèmes **P** et la classe des problèmes **EXPTIME**.

Dans le troisième chapitre, nous allons continuer dans la classification des problèmes de décision d'après la complexité de meilleurs algorithmes permettant de les résoudre. Ainsi, nous obtenons la classe **NP** et discuter l'une des plus importantes questions non encore résolue, à savoir est ce que Les classes **P** et **NP** sont confondues ou non, et les conséquences éventuelles sur le monde dans le cas d'une réponse positive [9]. Nous allons également intrduire le problème de satisfaisabilité, connu dans la littérature par **SAT**.

Dans le quatrième et dernier chapitre de ce mémoire, nous allons parler d'une classe importante de problèmes, à savoir la classe des problèmes **NP-complets**, qui sont considérés comme les problèmes les plus difficiles des problèmes **NP**. Le premier problème appartenant à cette classe, établi par Cook est le problème **SAT** [6]. La réduction polynomiale nous permet de d'élargir cette classe. Pour illustrer l'importance de l'application de réductions de problèmes, nous allons présenter la réduction du problème **SAT** à **3-SAT**, et la réduction du problème de 3-colorations de graphe, qu'on notera **COLORING**, au problème **3-SAT**.

Enfin, dans la conclusion, nous synthétisons les principaux résultats de notre travail, et mentionnons les nombreux problèmes non résolus, qui pourraient constituer des perspectives de recherches intéressantes.

Chapitre 1

Décidabilité et Machine de Turing

L'objectif de ce chapitre est de définir la notion de décidabilité via la notion de de machine de Turing. La théorie de la décidabilité donne un cadre qui permet de savoir si un problème mathématique peut-être résolu par un ordinateur, autrement dit est-ce qu'il existe un algorithme par sa résolution.

Afin d'approfondir les travaux sur les modèles de calculs formels et leur compréhension, Alan Turing introduit en 1936 la machine qui porte son nom. Cette machine s'inscrit dans la suite des travaux sur la théorie de décision. Cependant, Alan Turing réussit à pendre un nouveau point de vue sur le problème en s'abstrayant du point de vue uniquement fonctionnel et en introduisant les notions mathématiques.

Les machines de Turing viennent à leur tour conforter la théorie de la décidabilité via la thèse de Church. Elles sont aujourd'hui considérées comme le modèle abstrait des ordinateurs actuels. Elles englobent l'idée de procédure effective et donnent un moyen pratique de la mettre en place grâce à une machine. Permettent de classer un problème de décision et répondent à la question : **Quelles sont les ressources nécessaires pour la décidabilité ?**, ce que les précédents modèles n'arrivaient pas à faire.

1.1 Problème de décision

Notons qu'un problème de décision peut être exprimé comme l'appartenance d'un mot à un langage (un langage est un ensemble des mots). En effet chaque entrée du problème se représente par un mot (une suite des symboles) et l'ensemble des entrées pour lesquelles la solution au problème est oui.

Définition 1.1. Un **problème de décision** est un problème possédant des entrées (les instances) et une question concernant l'entrée dont la réponse est Oui ou Non en fonction de l'entrée.

Exemple 1.1.

a) **PRIME**

« Déterminer si un entier est premier » un problème est donc composé de deux éléments : une entrée et une question ou une tâche à réaliser :

- Entrée : un entier N ;
- Question : N est-il premier ?

b) Le problème de la chaîne hamiltonienne.

- Données : un graphe non orienté $G = (S.A)$, deux sommets s et t de G.
- Question : est ce qu'il existe une chaîne d'extrémités s et t qui passe une fois et une seule fois par chaque sommet du graphe G.

1.1.1 Instance de problème

On commence par définir un problème qui fixe un ensemble de questions appelées instances du problème ainsi que le sous ensemble de questions ayant une réponse positive.

Définition 1.2. On appelle **instance** du problème donné les objets mathématiques qui porte sur le problème.

Exemple 1.2.

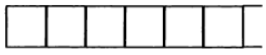
- Pour le problème PRIME, une instance est un nombre entier N.
- Pour le problème de chaîne Hamiltonien, une instance est un graphe G.

1.2 Machine de Turing

Description

Une machine de Turing[1] déterministe (deterministic Turing machine) est composée des éléments suivants.

a) Une mémoire infinie sous forme de ruban divisé en cases. Chaque case du ruban peut contenir un symbole d'un alphabet de ruban (tape alphabet).



b) Une tête de lecture (read head ou tape head) se déplaçant sur le ruban.

c) Un ensemble fini d'états parmi lesquels on distingue un état initial et un ensemble d'états accepteurs.

d) Une fonction de transition qui pour chaque état de la machine et symbole se trouvant sous la tête de lecture précise :

1. **Alan Mathison Turing** né le 23 juin 1912 à Londres et mort le 7 juin 1954 à Wilmslow, est un mathématicien et cryptologue britannique, auteur de travaux qui fondent scientifiquement l'informatique
Renommé pour : Problème de l'arrêt Machine de Turing Cryptanalyse d'Enigma ACE

- l'état suivant.
- un caractère qui sera écrit sur le ruban à la place du caractère se trouvant sous la tête de lecture.
- un sens de déplacement de la tête de lecture.

L'exécution d'une machine de Turing peut se décrire comme suit.

e) Initialement, le mot d'entrée se trouve au début du ruban. Les autres cases du ruban contiennent un symbole spécial appelé symbole blanc (blank symbol). La tête de lecture est sur la première case du ruban et la machine se trouve dans son état initial.

f) À chaque étape de l'exécution, la machine :

- lit le symbole se trouvant sous sa tête de lecture.
- remplace ce symbole par celui précisé par la fonction de transition.
- déplace sa tête de lecture d'une case vers la gauche ou vers la droite suivant le sens précisé par la fonction de transition.
- change d'état comme indiqué par la fonction de transition.

g) Un mot est accepté par la machine lorsque l'exécution atteint un état accepteur.

Définition 1.3.
Une machine de Turing est formellement décrite par un octuplet M=$(\Sigma, \Gamma, B, Q, q_0, q_a, q_{r,}, \delta)$ où :

* Σ est un ensemble fini non vide appelé alphabet d'entrée (ce sont les symboles utilisés pour écrire le mot d'entrée).
* Γ est un ensemble fini appelé alphabet de travail, tel que $\Sigma \subset \Gamma$ (ce sont les symboles utilisés dans les cases au cours du calcul).
* B est un symbole spécial blanc (représentant une case vide) tel que $B \in \Gamma \setminus \Sigma$.
* Q est un ensemble fini appelé ensemble des états (les états que peuvent prendre les têtes de lecture/écriture).
* $q_0 \in Q$ est un état spécial appelé état initial (indiquant l'état dans lequel les têtes commencent le calcul).

- $q_a \in Q$ et $q_r \in Q$ sont des états spéciaux appelé états terminaux (indiquant la fin du calcul) : q_a est l'état d'acceptation et q_r l'état de rejet .
- $\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{G, S, D\}$ est la fonction de transition décrivant le comportement des têtes.

Exemple 1.3. Machine de Turing déterminant si un nombre est pair.

- Entrée : le nombre à tester, sous forme binaire
- Sortie : 1 si le nombre est pair, 0 sinon

La figure (1.1) représente cette machine (le symbole ε représente le symbole blanc).

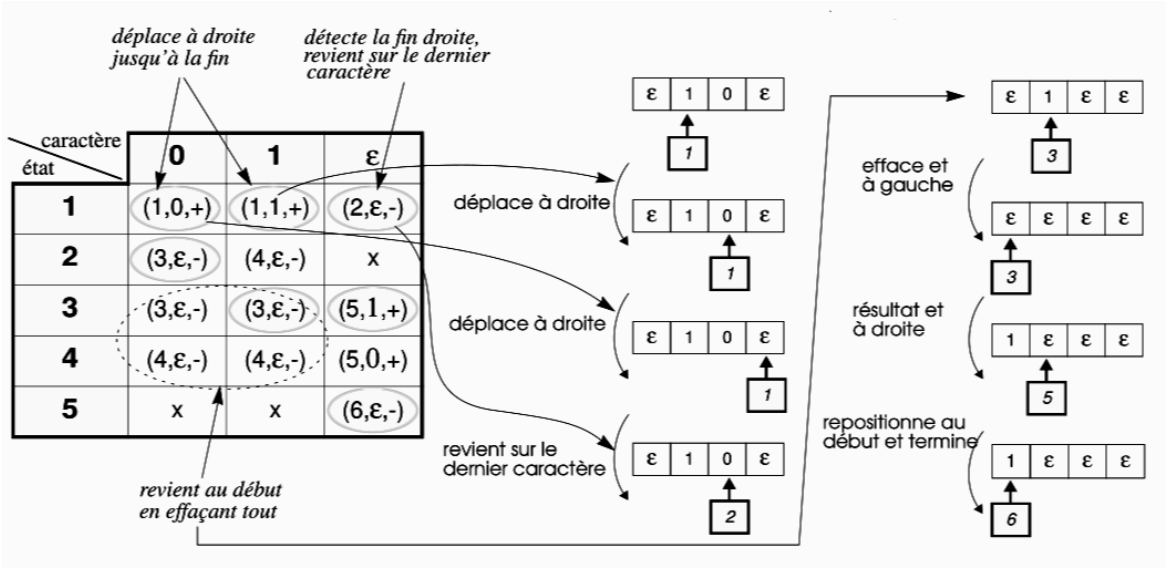


FIGURE 1.1 – Test de parité

1.2.1 Machines de Turing déterministes à plusieurs rubans

Une machine de Turing déterministe à plusieurs rubans ($k \geq 2$) est définie comme une machine de Turing déterministe comportant plusieurs rubans avec plusieurs têtes de lectures qui agissent simultanément mais indépendamment sur chacun des rubans. La fonction de transition prend en compte toutes les informations sur l'ensemble des rubans pour faire son choix et agit (en écriture ou en lecture)sur tous les rubans.

Soit k le nombre de rubans de la machine, alors la fonction de transition se définit comme suit :
$\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{G, S, D\}^k$

Théorème 1.1. [7] Les machines de Turing déterministes et les machines de Turing déterministes à plusieurs rubans sont équivalentes.

Preuve. Les machines de Turing déterministes et les machines de Turing déterministes à plusieurs rubans sont équivalentes. Comme, une machine de Turing déterministe est une machine de Turing déterministe à plusieurs rubans, on montre la réciproque.

On construit alors une machine de Turing déterministe simulant une machine de Turing déterministe à k rubans. Cette simulation consiste à définir comment utiliser l'unique ruban de notre machine de Turing déterministe. Sur ce dernier, on considère que les cases à la position i modulo k sur le ruban de la machine de Turing déterministe représentent les cases du ruban i de la machine de Turing que l'on simule.

La figure (1.2) présente un exemple de cette simulation pour une machine de Turing déterministe à deux rubans.

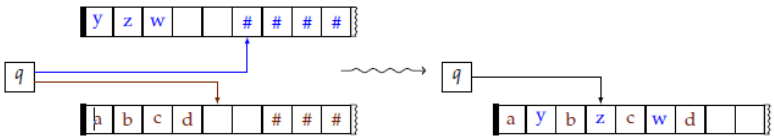


FIGURE 1.2 – Exemple d'une simulation d'une machine de Turing déterministe à deux rubans par une machine de Turing déterministe.

1.3 Thèse de Turing-Church

Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing. C'est la thèse de Turing-Church [2].

La notion physique de calculabilité, définie comme tout traitement systématique réalisable par un processus physique ou mécanique, peut être exprime par une machine de Turing déterministe.
Autrement tout ce qui est « calculable par un système physique » est calculable par machine de Turing[1].

Définition 1.4. Soit $\mathcal{T}$ une machine de Turing déterministe. **Le mot d'entrée** x est représente par le mot écrit sur le ruban au départ de l'exécution de la machine.

Si la machine atteint un état final après un nombre fini d'étapes, on dit que le mot x est **accepté** et le mot inscrit sur le ruban après l'arrêt s'appelle **mot de sortie** est noté $\mathcal{T}(x)$ si la machine s'arrête sans atteindre un état final ou si la machine ne s'arrête jamais, alors x est **rejeté**.

1.4 Machines de Turing non déterministe

Une machine de Turing (général) se décompose des mêmes éléments qu'une machine Turing déterministe sauf que la fonction de transition est remplacée par un ensemble de la relation

2. **Alonzo Church :** Mathématicien américain qui étudia le concept de calculabilité à partir de celui de fonction.

transition c'est-à-dire une combinaison état-case du ruban peut conduire à plusieurs action distinctes pour une machine de Turing non- déterministe, le même mot d'entrer x peut conduire à des calculs acceptant(c'est-à-dire qui atteignent un état final) et d'autres non acceptant. On dit que x est accepté si au moins un des calculs à partir de x est acceptant.

Théorème 1.2. [8] Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Considérons une machine de Turing à q état et à p lettre notons les états $(q_1, q_2, ..., q_p)$ et les lettre (c'est-à-dire les symboles du ruban) sont $(\emptyset, r_1, r_2, ..., r_q)$ il ya un nombre fini de transition possible (même si la machine n'pas déterministe), donc le nombre de machine de Turing à q état et p lettre est fini.

Remarque 1.1. L'ensemble des machine de Turing est dénombrable[3].

1.5 Machine de Turing Universelle

Une machine de Turing universelle peut simuler n'importe quelle machine de Turing. Une machine de Turing universelle peut être vue comme un interpréteur de machines de Turing.
En effet, comme les interpréteurs, elle prend en paramètre une autre machine de Turing que l'on applique un mot d'entrée x.
La machine universelle donne alors le résultat de la machine qu'elle simule sur son entrée x.

1.6 Décidabilité

Dans la théorie de la décidabilité, on cherche a savoir si il existe une procédure algorithmique qui permet de calculer pour chaque instance si la réponses est oui ou non.

Définition 1.5. Un problème de décision est décidable s'il existe une machine de Turing déterministe qui s'arête sur tout les entrées et qui atteint un état final si et seulement si la réponse est oui (ou vrai).

Exemple 1.4.

- le problème PRIME est décidable
- le problème de circuit hamiltonien est décidable

Pour le moment on a présenté seulement les problèmes décidables par la machine de Turing, mais la majorité des problèmes sont indécidable. Nous donnerons alors quelques éléments de preuve d'indécidabilité.

3. Rappelons qu'un ensemble U est dit dénombrable s'il est en bijection avec $\mathbb{N}$ ou une partie de $\mathbb{N}$. Il est fini s'il est en bijection avec une partie finie de $\mathbb{N}$, et infini dénombrable s'il est en bijection avec $\mathbb{N}$ tout entier. Dire qu'un ensemble est dénombrable, c'est donc dire qu'on peut numéroter ses éléments.

1.6.1 Problème indécidable

On dira d'un problème qu'il est indécidable si aucun algorithme n'existe pour le résoudre.

Exemple 1.5. On considère une fonction, qui pour tout algorithme et valeur d'entrée pour cet algorithme, indique si l'algorithme s'arrête ou pas. Ce qui revient à dire que le problème de l'arrêt est indécidable.
En d'autres termes, il n'existe pas de programme qui prenne en argument n'importe quel programme avec une valeur d'entrée et qui, en temps fini, renvoie « oui » si l'exécution du programme reçu en argument finit par s'arrêter avec l'entrée spécifiée et « non » s'il ne finit pas.

Preuve. On s'intéresse plus précisément aux programme (machine de Turing noté $\mathcal{T}$) qui prennent deux paramètres du problème P et la variable d'entre x, On considère la fonction suivante $C(x)$:

if $\mathcal{T}(x, x)$:
while True :
print("Le code boucle")
else :
print("Le code s'arrête")

Que se passe-t-il quand on lance $C(C)$?
Ou bien cet appel s'arrête : dans ce cas, $\mathcal{T}(C, C)$ renvoie True et on entre dans une boucle infinie. C'est absurde !
Ou bien cet appel ne s'arrête pas : $\mathcal{T}(C, C)$ renvoie False, puis le programme s'arrête !

Dans les deux cas, on aboutit à une contradiction. On en déduit que le programme $\mathcal{T}$ ne peut pas exister.

Le dixième problème de Hilbert (Matiassevitch 1970)

Le grand mathématicien allemand David Hilbert (1862–1943) a présenté en août 1900, à Paris, une liste de 23 problèmes à ses collègues. Ces problèmes interrogent en particulier les fondements des mathématiques et de la logique, et seule une moitié d'entre eux ont été résolus depuis. Le dixième problème de Hilbert consiste en la décidabilité d'un problème de décision. Il s'agit de déterminer s'il existe un algorithme qui décide si un polynôme à plusieurs variables dont tous les coefficients sont entiers admet ou non des solutions entières.
Par exemple, l'équation $3x^2 - 2xy + 4y^2z = 31$ admet comme solution $x = 1$, $y = 2$ et $z = 2$, alors que l'équation $3x^2 + 6xy + y^2 + z^2 + 1 = 0$ n'admet aucune solution entière.
Il a fallu attendre 1970 pour que le mathématicien Vladimirovitch Matiassevitch démontre, que ce problème est indécidable.

Dès que l'équation polynomiale considérée possède deux variables, tout se complique. Par exemple il est vrai que l'équation $x^2 - 991y^2 - 1 = 0$ a des solutions entières, mais elles ne sont pas faciles à trouver : la plus petite d'entre elles est $x = 379516400906811930638014896080$ et $y = 12055735790331359447442538767$.

On sait démontrer que le problème de savoir si une équation polynomiale à coefficients entiers à p inconnues a des solutions entières est décidable pour $p = 1$, on sait aussi qu'il est

indécidable pour $p \geq 9$. Mais on ne sait pas si le problème est décidable pour $p \in [2, 8]$. Il y a des sous-problèmes décidables : si on se restreint à des équations où le degré total est inférieur ou égal à 2, le problème est décidable ; le problème devient indécidable pour des degrés inférieurs ou égaux à 4.

En revanche, on ne sait pas dire si le dixième problème de Hilbert restreint aux polynômes de degré 3 est décidable ou non décidable !

Chapitre 2

Complexité algorithmique

Si on dispose d'un problème décidable, il est théoriquement possible de le résoudre avec un algorithme. Mais en pratique on se pose la question de savoir est-ce qu'on peut le résoudre dans un "temps raisonnable" et en utilisant " un espace mémoire raisonnable "(complexité en mémoire). Imaginons que pour résoudre un problème décidable, on fait recours à une procédure algorithmique qui nécessite 10^{80} bits de mémoire. Mission impossible, car aucun support ne peut contenir ce chiffre astronomique qui dépasse le nombre d'atome de l'univers visible.

Il est établi que le complexité en mémoire est inférieure à la complexité en temps (accéder à une case mémoire nécessite au moins une unité de temps), ainsi, dans la suite on va se contenter de considérer la complexité en temps. C'est la démarche adoptée par la plupart des études sur ce sujet.

2.1 Mesure de la complexité

Nous nous intéressons à la complexité en temps des algorithmes, à savoir au temps nécessaire à leur exécution. Le temps d'exécution d'un algorithme donné dépend principalement de deux facteurs :

a) **La machine utilisée :** La performance de micro-processeur et des autres composants du la machine (l'ordinateur) et des logiciel utilisés (software). Il est clair qu'un simple ordinateur du bureau ne peut égaler un méga-serveur d'une grande entreprise.

b) **Les donnée initiales :** Il est évident qu'un algorithme prend plus de temps dans le cas où les donnés d'entrée sont d'une grande taille que de petite taille.

La mesure de la complexité d'un algorithme sera donc une fonction de la taille des données initiales. Autrement, c'est une fonction de temps $f(n)$ où n est la taille des données.

Exemple 2.1 (**Multiplication standard de deux nombres**). :
On considérons un nombre x à p chiffres est un nombre y à q chiffres. La multiplication xy nécessite kpq opération élémentaires, k étant le nombre d'opérations élémentaires requis pour multiplier deux chiffres.
La taille de donnée est : $n = p + q$.

Remarque 2.1. Dans l'exemple de la multiplication de deux nombres, pour n fixé le temps d'exécution dépend fortement de p et q. Prenons, par exemple $n = 1000$. Le temps d'exécution dans le cas où $p = 999$ et $q = 1$ est largement plus petite que le cas où $p = 500$ et $q = 500$, alors que la taille des données initiales dans les deux cas est la même n.

Pour ne pas tomber dans ce genre de situation dans la mesure de la complexité d'un algorithme, on utilise la mesure appelée le **pire cas**, c'est à dire qu'on considère le cas où la taille des données est maximum.

Pour revenir à l'exemple de la multiplication standard de deux nombre, le nombre d'opérations est maximum lorsque $p = q = \frac{1}{2}n$.
Donc la complexité de la multiplication est $kpq = k\frac{1}{4}n^2$.

La complexité en temps d'un algorithme est le nombre d'opération élémentaires qu'il effectue. Nous donnons maintenant la définition formelle de la complexité en temps.

Définition 2.1. Soit $\mathcal{T}$ une machine de Turing déterministe sans calcul infini. Pour l'entrée x, notons $D_{\mathcal{T}}(x)$ le nombre d'étapes dans l'exécution de $\mathcal{T}$. La complexité de $\mathcal{T}$ en temps est la fonction $C_{\mathcal{T}}(n)$définie pour :

$$C_{\mathcal{T}}(n) = \max\{\ m/\text{il existe un mot } x \text{ de taille } n \text{ tel que } D_{\mathcal{T}}(x) = m\}$$

Remarque 2.2. Dans la mesure de la complexité de la multiplication standard, on a trouvé que le résultat est de $\frac{k}{4}n^2$.

Dans la pratique le facteur k peut changer en fonction de la puissance de la machine (c'est-à-dire en changeant de processeur), or l'objectif est que la mesure ne dépende pas du support utilisé. C'est pour cette raison qu'on ne tiendra pas compte de la constante k, et on dira que la complexité de la multiplication standard est $\mathcal{O}(n^2)$. Autrement dit, on s'intéresse à l'ordre de grandeur.

2.2 Ordre de grandeur asymptotique

La notation $\mathcal{O}$ simplifie l'expression des fonctions de complexité. De plus, elle n'exprime qu'une borne supérieurs sur les fonctions et permet donc de donner une indication sur la complexité d'un algorithme, d'ou la notation $\mathcal{O}$ ignore le comportement des fonction pour les petites valeurs et nous permet souvent de simplifier l'expression de la complexité d'un algorithme, elle est souvent dominée par des opérations d'initialisation qui deviennent négligeable pour des données assez grandes.

Définition 2.2. Si $f(n)$ et $g(n)$ sont deux suites numériques, on dit que $f(n)$ est "grand $\mathcal{O}$" de $g(n)$ et on note $f(n) = \mathcal{O}g(n)$ si et seulement si :

$$\exists M > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, |f(n)| \leq M|g(n)|$$

Propriété 2.1.

- $c\mathcal{O}(f) = \mathcal{O}(f)$;
- $\mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f)$ *et que* $\mathcal{O}(f_1) + \mathcal{O}(f_2) = \mathcal{O}(max(f_1, f_2))$;
- $\mathcal{O}(\mathcal{O}(f)) = \mathcal{O}(f)$.

Exemple 2.2. La fonction cn^2 est $\mathcal{O}(n^2)$. Il en va de même de $c_1n^2 + c_2n$ puisque pour $n \geq 1$, $c_1n^2 + c_2n \leq (c_1 + c_2)n^2$. On dira donc qu'un algorithme dont le temps de calcul est donné par $c_1n^2 + c_2n$ a une fonction de complexité $\mathcal{O}(n^2)$

Définition 2.3. Si f et g deux fonctions $\mathbb{N}$ dans $\mathbb{R}_+^*$ alors, on dit que f et g sont équivalentes et on écrit $f \sim g$ si et seulement si : $f = \mathcal{O}_g$ et $g = \mathcal{O}_f$.
autrement, $\exists c > 0, \exists d > 0, \exists \varepsilon_0 \in \mathbb{N}$ tel que :

$$\forall \varepsilon \geq \varepsilon_0, dg(\varepsilon) \leq f(\varepsilon) \leq cg(\varepsilon).$$

Exemple 2.3.

$$2n \sim n$$

.

$$2n \nsim n^2$$

.

2.2.1 Classification des algorithmes

Le tableau suivant illustre la classification par des explications.
Les algorithmes habituellement rencontrés peuvent être classés dans les catégories suivantes :

Complexité	**Description**
Complexité $\mathcal{O}(1)$ Complexité constante ou temps constant.	L'exécution ne dépend pas du nombre d'éléments en entrée mais s'effectue toujours en un nombre constant d'opérations.
Complexité $\mathcal{O}(\log(n))$ Complexité logarithmique.	La durée d'exécution croît légèrement avec n. Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.
Complexité $\mathcal{O}(n)$ Complexité linéaire.	C'est typiquement le cas d'un programme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n.
Complexité $\mathcal{O}(n^2)$ Complexité quadratique.	Typiquement c'est le cas d'algorithmes avec deux boucles imbriquées chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.
Complexité $\mathcal{O}(n^3)$ Complexité cubique.	Idem quadratique mais avec ici par exemple trois boucles imbriquées, exemple : algorithme de Gauss pour échelonner une matrice de taille $n \times m$: $\mathcal{O}(n^3)$ complexité polynomial.
Complexité $\mathcal{O}(n^p)$ Complexité polynomiale.	Algorithme dont la complexité est de la forme $\mathcal{O}(n^p)$pour un certain p. Toutes les complexités précédentes sont incluses dans celle-ci.
Complexité $\mathcal{O}(2^n)$ complexité exponentielle.	Les algorithmes de ce genre sont dit " naïfs " car ils sont inefficaces et inutilisables des que n dépasse 50.
complexité $\mathcal{O}(n\,!)$complexité factorielle.	Résolution par recherche exhaustive du problème du voyageur de commerce.

On peut "ranger" les fonctions équivalentes dans une même classe. la figure (2.1) montre quelques classes de complexité usuelles (par ordre croissant en termes de $\mathcal{O}(.)$).

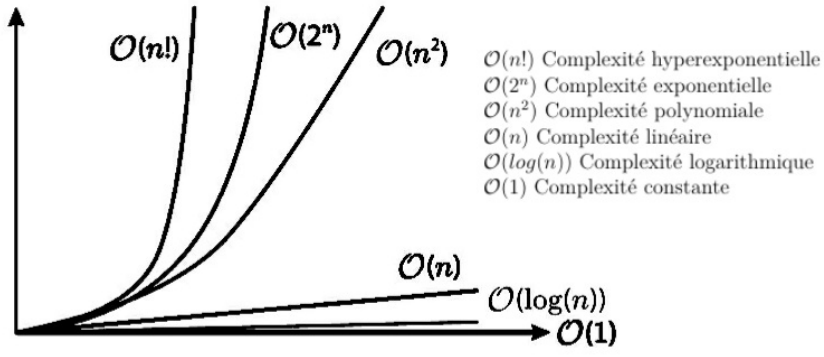


FIGURE 2.1 – Différentes classe de complexité

Le Tableau (2.1) montre la croissance du temps d'exécution en fonction de la taille des données

sur base d'unités concrètes de temps et permet de mieux se rendre compte des impacts pratiques. On considère un ordinateur effectuant 1000000 d'opérations par seconde.

n \ $f(n)$	$log(n)$	n	$n * log(n)$	n^2	n^3	2^n
10^2	$6.6\mu s$	$0.1ms$	$0.6ms$	$10ms$	$1s$	$4 * 10^6 a$
10^3	$9.9\mu s$	$1ms$	$9.9ms$	$1s$	$16.6mn$	$> 10^{100} a$
10^4	$13.3\mu s$	$10ms$	$0.1s$	$100s$	$11.5j$	$> 10^{100} a$
10^5	$16.6\mu s$	$0.1s$	$1.6s$	2.7	31.7	$> 10^{100} a$
10^6	$19.9\mu s$	$1s$	$19.9s$	$11.5j$	$31.7 * 10^3 a$	$> 10^{100} a$

- μs : micro-secondes, ms : milli-secondes, s :secondes.
- mn : minutes, h : heure, j : jour, a : année.

TABLE 2.1 – Comparaison en pratique du temps de calcul qu'impliquent les différentes complexités
.

Exemple 2.4.

- algorithme de Gauss pour échelonner une matrice de taille $n \times m$: $\mathcal{O}(n^3)$ complexité polynomiale
- calcul PGCD par l'algorithme d'Euclide : $\mathcal{O}(n)$, (n la taille du plus petit des nombre a et b) complexité linéaire.
- algorithme naïf de primalité de a en testant la divisibilité par tous les nombres inférieurs à $\sqrt{a}$: $\mathcal{O}(10^{\frac{n}{2}})$: complexité exponentielle.
- algorithme naïf pour le circuit hamiltonien d'un graphe à n sommets : $\mathcal{O}(n!)$: complexité factorielle.

on peut facilement se convaincre qu'un algorithme de complexité $\mathcal{O}(2^n)$ n'est pas recommandé (exemple : n=100, 2^n=4×10^{16}ans).

Définition 2.4. On considère généralement qu'un algorithme est **plus efficace** qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.

Par convention, un algorithme est dit **efficace** si sa complexité est polynomiale, c'est-a-dire de la forme $\mathcal{O}(n^k)$, avec $k \in \mathbb{N}$.

Exemple 2.5.
L'algorithme du Karatsuba fait la multiplication en $\mathcal{O}(n^{ln(3)/ln(2)}) \simeq \mathcal{O}(n^{1.58}) < \mathcal{O}(n^2)$, donc il est **plus efficace** que la multiplication standard.

Remarque 2.3. On peut argumenter qu'un algorithme de complexité $\mathcal{O}(n^{2021})$ n'est pas très raisonnable. Certes, mais il faut bien fixer une convention.
On considère que c'est raisonnable en théorie de la complexité.

2.3 Classe P et classe EXPTIME

Nous allons maintenant passer progressivement au classement des problèmes de décision en fonction de la complexité des algorithmes permettant de les résoudre.

Définition 2.5. L'ensemble des problèmes qui peuvent être décidés par une machine de Turing déterministe polynomiale est noté P.

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

L'ensemble des problèmes qui peuvent être décidés par une machine de Turing déterministe exponentielle (de complexité $\mathcal{O}(e^{nk})$), avec $k \geq 1$) est noté EXPTIME.

$$\mathrm{EXPTIME} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(e^{nk})$$

Remarque 2.4. On a $\mathrm{P} \subset \mathrm{EXPTIME}$ (car $f(n) = \mathcal{O}(n^k) \Rightarrow f(n) = \mathcal{O}(e^{nk})$).

Chapitre 3

Classe NP

Il s'avère toute fois qu'il y a toute une classe de problème pour lesquelles à ce jour on n'arrive pas à construire d'algorithme polynomiale, mais sans qu'on arrive à prouver formellement que cela ne soit pas possible. C'est historiquement ce qui a mené à considère la classe de problèmes que l'on appelle NP.

3.1 Présentation des problèmes de la classe NP

Avant de parler des problèmes de la classe NP, nous allons commencé par définir d'une maniéré rigoureuse la complexité d'une machine de Turing.

Définition 3.1. Soit $\mathcal{T}$ une machine de Turing sans calcul infini. Pour un mot initial x, le temps du calcul de $\mathcal{T}$ est défini par :
$D_{\mathcal{T}}(x) = \min \{m \mid$ il existe une exécution de $\mathcal{T}$ sur x qui s'arrête après m étapes$\}$. La complexité de $\mathcal{T}$ en temps est la fonction $C_{\mathcal{T}}(n)$ définie par :
$C_{\mathcal{T}}(n) = \max\{m \mid$ il existe un mot x de taille n tel que $D_{\mathcal{T}}(x) = m \}$.

Définition 3.2. La classe **NP** l'ensemble des problèmes qui peuvent être décidés par une machine de Turing polynomiale.

Lemme 3.1. La classe **NP** contient la classe **P ($\mathbf{P \subseteq NP}$).**

La démonstration est immédiate puisqu'une machine de Turing déterministe polynomiale est un cas particuliers de machine de Turing non déterministe polynomiale.

Remarque 3.1.

i. À la différence des problèmes de la classe **P** qui sont des problèmes qui peuvent être décidés par une machine de Turing déterministe polynomiale, les problèmes de la classe **NP** sont des problèmes qui peuvent être décidés par une machine déterministe ou non déterministe d'une manière polynomiale.

ii. On a d'une manière claire que $\mathbf{P} \subset \mathbf{NP}$. L'inclusion inverse, à savoir est-ce que $\mathbf{NP} \subset \mathbf{P}$, est un problème non encore résolu et fait partie des problèmes majeurs des mathématiques connus sous le nom des **problèmes du millénaires**.

Exemple 3.1.

i. Problème du circuit hamiltonien est un problème **NP** :

Si on dispose d'un ordre des sommets formant un circuit hamiltonien, on vérifie en un temps polynomial que c'est bien le cas.

ii. Problème du sodoku est un problème **NP** :

La vérification qu'une solution est acceptable se fait "rapidement".

Remarque 3.2.
La définition du temps de calcul d'une machine de Turing $D_{\mathcal{T}}(x)$ qui est le temps minimum de toutes les possibilités d'exécution sur x, et qui nous a permis de définir la classe des problèmes **NP**, nous fournit un outil intéressant du point de vue théorique, même si cette notion n'est pas très réaliste du point de vue pratique. Dit d'une façon courte, ce qui nous permet de dire qu'un problème est de classe **NP** est la vérification d'une solution (se fait en un temps polynomiale).

Définition 3.3 (**Autre définition de la classe NP**)**.**

NP est l'ensemble des problèmes de décision tel que pour toutes les instances qui sont acceptées, une preuve on dit aussi **(certificat)** que cette réponse est oui peut-être vérifiée en temps polynomiale par une machine de Turing déterministe.

3.2 Problème SAT

Nous allons exposer dans cette section un problème qui joue un rôle fondamental dans la théorie de la complexité. Il s'agit du problème qu'on note SAT.

Avant de se lancer dans la définition du problème SAT, quelque définitions relatives à la logique mathématique sont nécessaires pour la compréhension et seront utilisées ultérieurement.

3.2.1 Notions de base

Les variables booléennes, ou logiques, appelées aussi littéraux, sont susceptibles de prendre deux valeurs et deux seulement, à savoir $\{vrai$ ou $faux\}$ ou encore $\{0, 1\}$ avec 1 pour $vrai$ et 0 pour $faux$. Ces éléments 0 ; 1 ou $vrai$; $faux$ sont les éléments universels.

Dans l'algèbre de Boole[1], il existe trois opérateurs de base qui sont appelés **connecteurs** :

- $\vee$ qui correspond au *ou* logique
- $\wedge$ qui correspond au *et* logique
- $\neg x$ qui est la négation qu'on note aussi $\bar{x}$.

1. **George Boole :** né le 2 novembre 1815 à Lincoln et mort le 8 décembre 1864 à Ballin temple, est un logicien, mathématicien et philosophe britannique. Il est le créateur de la logique moderne, fondée sur une structure algébrique et sémantique, que l'on appelle algèbre de Boole en son honneur.

Les deux premiers connecteurs sont des opérateurs binaires, le troisième est un opérateur unaire, la **négation**.

Définition 3.4. Une **fonction logique** (booléenne) f à n variables est une application :

$$f : \{0,1\}^n \longrightarrow \{0,1\}$$

ou encore ;

$$f : \{vrai, faux\}^n \longrightarrow \{vrai, faux\}$$

Définition 3.5. Une **proposition** (ou assertion) est une phrase à laquelle on peut attribuer une valeur de vérité (vrai ou faux).

Définition 3.6. L'affectation est un opérateur qui affecte à une variable booléenne une valeur de l'ensemble $\{0,1\}$ ou, respectivement, de l'ensemble $\{vrai, faux\}$. Cet opérateur est également appelé assignement.

Définition 3.7. On appelle **clause** une formule propositionnelle qui n'utilise que des littéraux (variable propositionnelle) et la disjonction. La clause peut toujours s'écrire de la forme : $x_1 \vee x_2 \vee ... \vee x_n$.

Définition 3.8. On appelle **Forme normale conjonctive** ou **FNC** une expression booléenne formée de conjonctions de clauses.

Exemple 3.2. L'expression $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$ est une forme normale conjonctive. $(x_1 \vee \neg x_3)$ et $(x_2 \vee x_3)$ en sont ses clauses.

Définition 3.9. Une forme normale conjonctive définie sur l'ensemble des variables $\{0,1\}$ est **satisfaisable** si et seulement si il existe une affectation pour toutes les variables qui lui confère la valeur vraie.

Exemple 3.3. La formule $(\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_4)$ est satisfiable (satisfaisable) car elle admet au moins la solution $x_1 \leftarrow 0$, $x_3 \leftarrow 1$ quelles que soient les valeurs de x_2 et x_4.

3.2.2 Décidabilité du calcul propositionnel

Une logique est décidable s'il existe une machine de Turing qui permet de savoir pour chaque formule si elle est une tautologie ou pas. Une **tautologie** est une formule logique qui prend toujours la valeur vraie quelques soient les valeurs de vérité des variables qui la compose.

Théorème 3.1. [4] Le calcul propositionnel est décidable.

Preuve.

Soit φ une formule propositionnelle.

Méthode des tables de vérité : calculer la table de vérité prenant en argument les symboles propositionnels de φ et calculer pour chaque valuation possible la valeur de φ.

Coût : $\mathcal{O}(2^n)$ avec n la taille de φ (nombre de propositions).

Définition 3.10 (**Le problème SAT**). **Le problème SAT** s'énonce comme suit : Étant donnée une FNC, existe-t-il une valeur de vérité satisfaisante pour cette FNC, ou, dit autrement, existe-t-il une affectation de valeurs 0, 1 aux variables constituant cette FNC et lui conférant la valeur 1.

SAT est un problème de décision central en théorie de la complexité. SAT pour **satisfaisabilté**.

Définition 3.11. Un problème k-SAT, est un problème SAT où l'arité des clauses est égale à k. L'arité des clauses est le nombre maximum de littéraux contenus dans une clause.

Le problème $4-$SAT est un problèmes SAT où toutes les clauses sont d'arité 4.

Théorème 3.2. [3] Le problème SAT est décidable.

Preuve.
Étant donné une formule φ ayant n variables propositionnelles. Calculer les $2n$ valuations possibles. Pour chacune d'entre-elles, calculer la valeur de vérité de φ. Si au moins une est vraie, alors φ est satisfaisable.

Il existe des algorithmes plus performants, mais ces améliorations ne changent pas fondamentalement la difficulté du problème. On est devant la situation suivante. Étant donnée une formule φ, on se demande si elle admet un modèle ou non, c'est-à-dire, s'il existe des valeurs de vérité attribuables aux variables propositionnelles qui satisferaient φ :

- Une recherche exhaustive comme dans l'algorithme précédent peut demander jusqu'à $2n$ vérifications si φ possède n variables propositionnelles. Cette démarche est dite déterministe, mais son temps de calcul est exponentiel.
- D'un autre côté, si φ est satisfiable, il suffit d'une vérification à faire, à savoir tester précisément la configuration qui satisfait φ. Cette vérification demande un simple calcul booléen, qui se fait en temps polynomial (essentiellement linéaire en fait). Le temps de calcul cesse donc d'être exponentiel, à condition de savoir quelle configuration tester. Celle-ci pourrait par exemple être donnée par un oracle auquel on ne ferait pas totalement confiance. Une telle démarche est dite non déterministe.

3.3 Sous-classes de SAT

3.3.1 2-SAT

Le problème 2-SAT est celui de la satisfaisabilité d'une formule sous forme clausale dont les clauses sont d'ordre 2 (c'est-à-dire, chaque clause est une disjonction d'au plus deux littéraux).

Exemple 3.4. Problème **2-SAT**

- **Entrée** : un ensemble des clauses C d'ordre 2
- **Question** : $vrai$ si C est satisfaisable ou $faux$ sinon

Ce problème est résolvable en temps polynomial.

Démonstration.
Algorithme $2-SAT$: Une clause d'ordre deux $l_1 \vee l_2$ est équivalente à $(\neg l_1 \Rightarrow l_2) \wedge (\neg l_2 \Rightarrow l_1)$. Pour résoudre SAT pour une formule φ d'ordre 2, on construit le graphe orienté $G(\varphi) = (S, A)$ dual (appelé graphe $2-SAT$) selon les deux règles suivantes :

- l'ensemble des sommets est $S = \{\neg p_1 \mid p \in Prop(\varphi)\} \cup \in Prop(\varphi)$(où $Prop(\varphi)$ est l'ensemble des variables propositionnelles de la formule (φ). C'est l'ensemble des littéraux sur $Prop(\varphi)$.

- l'ensemble des arcs est $A = \{(l_1, l_2) \mid \varphi$ contient une clause équivalente à $l_1 \Rightarrow l_2\}$ Chaque clause($l_1 \vee l_2$) est donc associée à deux arcs $(\neg l_1, l_2)$ et $(l_1, \neg l_2)$. Alors, la formule φ est insatisfaisable ssi il existe une variable p telle qu'il existe dans $G(\varphi)$ un chemin allant de p à $\neg p$ et un chemin allant de $\neg p$ à p. En effet, cela signifie alors que $(\neg p \Rightarrow q) \wedge (\neg q \Rightarrow p)$ ce qui constitue une contradiction.

  Pour chaque variable propositionnelle p, les sommets p et $\neg p$ du graphe $2-SAT$ sont dans deux composantes fortement connexes distinctes. (une composante fortement connexe d'un graphe orienté G est un sous-graphe maximal de G tel que pour toute paire de sommets u et v dans ce sous-graphe, il existe un chemin de u à v et un chemin de v à u).

  L'algorithme de Tarjan permet de calculer les composantes fortement connexes d'un graphe orienté en $\mathcal{O}(|S| + |A|)$, donc $2-$ SAT est bien polynomial.

Exemple 3.5. $\varphi = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$
On a :

- $(p_1 \vee p_2) \equiv (\neg p_1 \Rightarrow p_2) \wedge (\neg p_2 \Rightarrow p_1)$
- $(\neg p_1 \vee p_3) \equiv (p_1 \Rightarrow p_3) \wedge (\neg p_3 \Rightarrow \neg p_1)$
- $(\neg p_2 \vee p_1) \equiv (p_2 \Rightarrow p_1) \wedge (\neg p_1 \Rightarrow \neg p_2)$
- $(\neg p_2 \vee p3) \equiv (p_2 \Rightarrow p_3) \wedge (\neg p_3 \Rightarrow \neg p_2)$
- $(\neg p_1 \vee \neg p_3) \equiv (p_1 \Rightarrow \neg p_3) \wedge (p_3 \Rightarrow \neg p_1)$

On construit le graphe $G(\varphi) = (S, A)$:

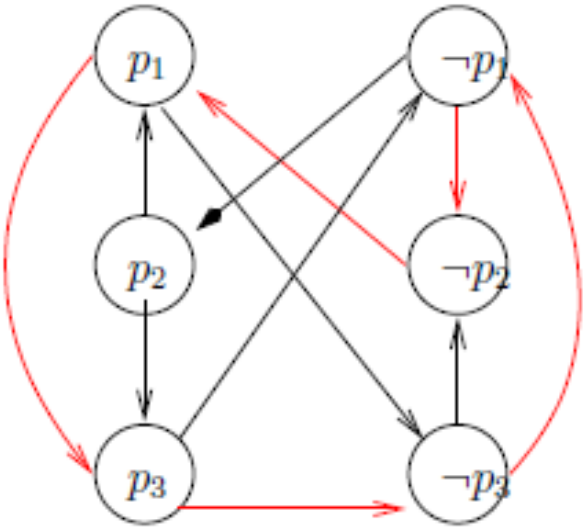


FIGURE 3.1 – Graphe correspondant à φ

On remarque dans le graphe (3.1) qu'il existe un cycle passant par p_1 et $\neg p_1$, donc la formule est insatisfaisable. Si on considère maintenant la formule :

$$\varphi' = (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$$

Le graphe est alors le suivant :

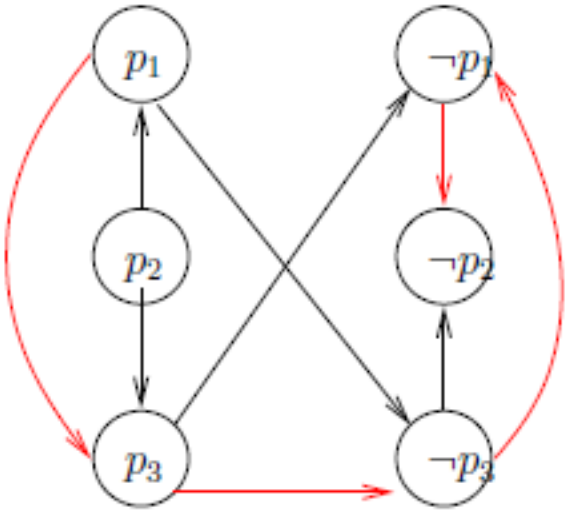


FIGURE 3.2 – Graphe correspondant à φ'

On voit sur le graphe (3.2) que $p_1 \Rightarrow \neg p_1, p_2 \Rightarrow \neg p_2$ et $p_3 \Rightarrow \neg p_3$ sont des tautologies. Il n'y a donc qu'un seul modèle : $v(p_1) = v(p_2) = v(p_3) = 0$.

3.3.2 Horn-SAT

Une clause de **Horn** est une clause comportant au plus un littéral positif. C'est donc une disjonction de la forme $\neg p_1 \vee ... \vee \neg p_n \vee p$, où les p_i et p sont des variables propositionnelles selon qu'elles comportent ou non un littéral positif (resp. négatif).
Le problème **Horn-SAT** s'énonce de la façon suivante :

Exemple 3.6. problème de **Horn-SAT**

- **Entrée** : un ensemble $\mathcal{C}$ de clauses de **Horn**.
- **Question** : $\mathcal{C}$ est-il satisfiable ?

Ce problème est résolvable en temps polynomial.

3.3.3 Probleme 3-SAT

Le problème de **3-satisfaisabilitè** appelé $3-\text{SAT}$ est un cas particulier de SAT qui a la particularité d'être **NP-complet**. Comme pour SAT, les instances $3-\text{SAT}$ sont formés de clauses, mais dans $3-\text{SAT}$ chaque clause contient au plus trois 3 litteraux. Il s'énonce de la façon suivante :

Exemple 3.7. Problème $3-\text{SAT}$

- **Entrée** : un ensemble des clauses C d'ordre 3
- **Question** : *vrai* si C est satisfaisable ou *faux* sinon

3.4 La classe Co-NP

Définition 3.12. La classe $\text{Co}-\text{NP}$ est l'ensemble des problèmes de décision tel que pour toutes les données initiales (instance) qui sont **rejetées**, une preuve que cette réponse est non peut être vérifiée en un temps polynomial par une machine de Turing déterministe.

Remarque 3.3.

i. Á la différence des problèmes **NP**, pour les problèmes $\text{Co}-\text{NP}$, c'est la preuve qu'une réponse est non qui peut être vérifiée en un temps polynomial et non pas en certificat pour une réponse positive.

ii. Il est évident que $\text{P} \subset \text{Co}-\text{NP}$

Exemple 3.8.

- Considérons le problème suivant appelé **VALID**
  - **Entré** : Une proposition logique avec n variable
  - **Question** : Est-ce que cette proposition est vraie pour toute assignation des variables.

VALID $\in$ Co $-$ NP car il suffit de donner une assignation qui rend la proposition fausse pour vérifier en un temps raisonnable que la réponse et non.

- On sait que le problème du circuit hamiltonien est dans NP. Mais étant donné une réponse non pour le problème du circuit hamiltonien, on ne sait pas si il existe une preuve vérifiable en un temps polynomial.

Remarque 3.4.

∗ La question qui consiste à savoir est ce que

$$\mathbf{Co - NP = NP}$$

et

$$\mathbf{Co - NP \bigcap NP = P}$$

restent des questions ouvertes.

∗ A ce stade, on peut représenter les différentes classse des problèmes par les deux schémas suivant :

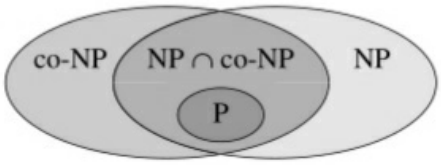


FIGURE 3.3 – Cas où : P $\neq$ NP $\cap$ Co $-$ NP

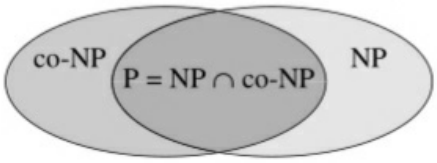


FIGURE 3.4 – Cas où : P = NP $\cap$ Co $-$ NP

Chapitre 4

NP-Complétude

Le principe de l'étude de la complexité est de classifier les problèmes par rapport au critère de temps d'exécution sur une machine de Turing. Les deux classes P et NP que nous avons définies déjà ne semblent pas assez fines pour discriminer la difficultése des classes de problèmes. Nous aimerions introduire la réduction polynomiale qui nous permettra de dire qu'un problème est plus difficile qu'un autre.

Définition 4.1 (**Réduction polynomiale**)**.** Soient deux problèmes Π_1 et Π_2, et un algorithme f qui prend en entrée des instances de Π_1 et qui retourne des instances de Π_2. f est appelée **réduction fonctionnelle polynomiale** de Π_1 vers Π_2 si et seulement si :

- f est un algorithme en temps polynomial.
- x est une instance positive de Π_1 si et seulement si $f(x)$ est une instance positive de Π_2 .

Remarque 4.1.

Si Π_1 se réduit à Π_2, tout algorithme qui permet de résoudre Π_1 permet de résoudre Π_2 en restant dans la même classe de complexité. Ceci provient du fait que l'application f est calculable en temps polynomiale :

$$\Pi_1 \in \mathrm{P} \Rightarrow \Pi_2 \in \mathrm{P}$$
$$\Pi_1 \in \mathrm{NP} \Rightarrow \Pi_2 \in \mathrm{NP}$$

Proposition. [6] Le problème SAT se réduit à $3-\mathrm{SAT}$

Preuve.

Il suffit de montrer qu'à toute proposition de SAT, on peut associer une proposition de $3-\mathrm{SAT}$ qui soit satisfaisable si et seulement si la première l'est .

Considérons une clause d'une proposition de SAT

$$a_1 \vee a_2 \vee ... \vee a_p \qquad (avec \;\; p \geq 3) \tag{4.1}$$

on remplace cette clause par la proposition suivante :

$$(a_1 \vee a_2 \vee u_1) \wedge (a_3 \vee \neg u_1 \vee u_2) \wedge (a_4 \vee \neg u_2 \vee u_3) \wedge ....(a_{p-2} \vee \neg u_{p-4} \vee u_{p-4}) \wedge (a_{p-1} \vee a_p \vee \neg u_{p-3}) \tag{4.2}$$

Les propositions (4.1) et (4.2) ont même valeur de vérité par construction. Ainsi on peut remplacer chaque clause du problème SAT par une formule (4.2) et on obtient un problème

$3-$SAT (chaque clause possède au plus trois variables). Remarquons qu'en effectuant la transformation de (4.1) vers (4.2), on a ajouté $(p-3)$ variables $(u_1, u_2, ..., u_{p-3})$, donc c'est une réduction polynomiale.

Remarque 4.2.
La proposition précédente signifie que tout algorithme qui permet de résoudre le problème SAT permet de résoudre le problème $3-$SAT. Donc $3-$SAT est aussi "difficile" que SAT.

4.1 Problème de coloration de graphes

Nous allons présenter une réduction de deux problèmes qui sont en apparence de nature différente, mais que grâce à cette réduction, si on dispose d'un algorithme qui résout l'un, on peut résoudre l'autre avec la même complexité.

4.1.1 Problème COLORING

Le problème **COLORING** est le problème qui consiste à savoir, étant donné un graphe G à n sommets, est-ce qu'on peut colorier ses sommets à l'aide de 3 couleurs de sorte que deux sommets reliés ne sont jamais de même couleur.

Théorème 4.1. [4] Le problème $3-$COLORING *est* NP $-$ complet.

Démonstration : $3-$COLORING est dans NP, car la donnée des sommets colorés par chacune des 3 couleurs constitue un certificat vérifiable en temps polynomial.

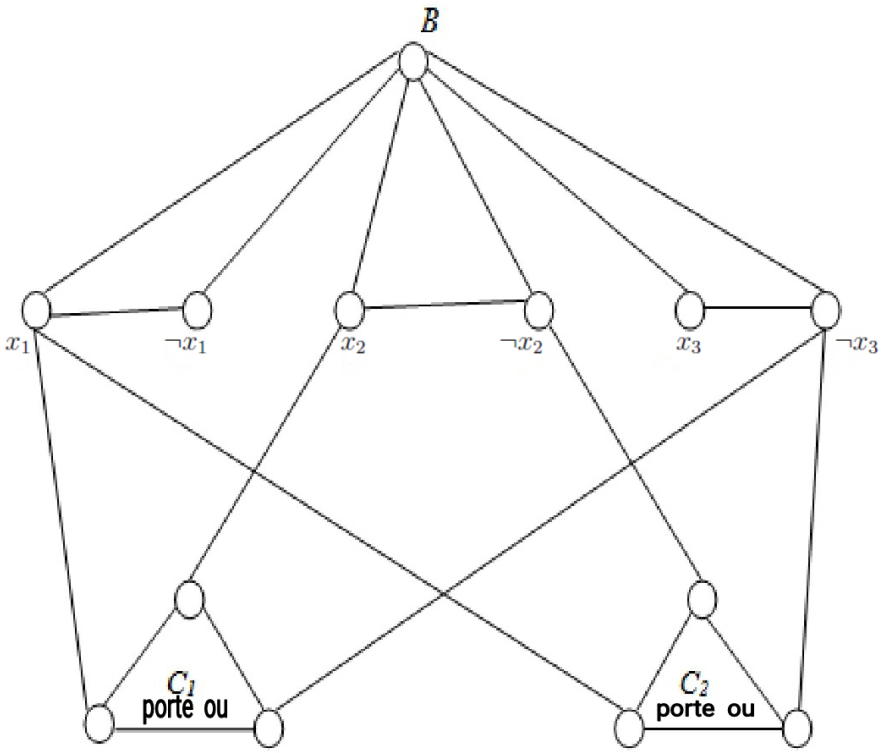
On va réduire $3-$SAT à $3-$COLORING. On se donne donc une conjonction de clauses à 3 littéraux, et il nous faut à partir de là construire un graphe. Comme dans les autres réductions de $3-$SAT, il faut parvenir à traduire deux contraintes : une variable peut prendre la valeur 0 ou 1 d'une part, et les règles d'évaluation d'une clause d'autre part.

On construit un graphe ayant $(3+2n+5m)$ sommets et $(3n+6m)$ arêtes et transforme la disjonction à des portes ou ; les trois premiers sommets sont notés $VRAI, FAUX, NSP$. Ces trois sommets sont reliés deux à deux en triangle de sorte qu'ils doivent être tous trois de couleurs différentes. On appellera les couleurs correspondantes $VRAI, FAUX, NSP$.

On associe un sommet à chaque variable et au complémentaire de chaque variable. Pour assurer qu'une variable prenne la valeur $VRAI$ ou $FAUX$, pour chaque variable x_i on construit un triangle dont les sommets sont x_i, $\neg x_i$, et NSP. Cela impose que soit couleur$(x_i) = VRAI$ et couleur $(\neg x_i) = FAUX$, ou couleur $(x_i) = FAUX$ et couleur $(\neg x_i) = VRAI$.

Il nous reste donc à encoder les règles d'évaluation d'une clause. Pour ce faire, on introduit le graphe suivant, qui correspond à deux clauses :

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$



L'existence d'une $3-$coloration du graphe est donc équivalente à la satisfaisabilité de la formule initiale.

La réduction est manifestement polynomiale ; on a donc bien prouvé que $3-$SAT se réduisait à $3-$COLORING ; ce dernier est donc bien NP $-$ complet.

4.2 Réduction du Sudoku à SAT

Une grille de Sudoku est composée de $n = \ell^2$ cases, et cette grille est elle-même divisée en ℓ sous-grilles. Généralement, on prend $n = 9$ et $\ell = 3$. Au départ certaines cases sont remplies par des chiffres et d'autres sont vides. Le but du jeu est de remplir les cases vides en respectant les règles suivantes :

- On ne doit pas avoir deux chiffres identiques sur une même ligne.
- On ne doit pas avoir deux chiffres identiques sur une même colonne.
- Il ne doit pas y avoir deux chiffres identiques dans l'une des sous-grilles de taille $\ell * \ell$.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Pour être un Sudoku, la grille doit accepter une et une seule solution.

4.2.1 Formalisation

Pour représenter le problème sous forme de problème SAT, nous avons besoin de beaucoup de variables propositionnelles.

En effet il faut n^3 variables V_{xyz} avec x, y et $z \in [1, n]$. La variable V_{xyz} sera vraie si et seulement si la case (x, y) contient la valeur z. Par exemple, si V_{135} est vraie, alors la case $(1, 3)$ contient $5.Si$ on prend $n = 4$, il faudra 64 variables, pour $n = 9$ il en faut 729. Voici la formulation des règles de remplissage d'un Sudoku.

« Chaque case contient un et un seul chiffre » : pour tout $i \in [1, n]$, pour tout $j \in [1, n]$, il existe $k \in [1, n]$ tel que la case (i, j) contient k et pour tout $k' \neq k$ dans $[1, k]$: la case (i, j) ne contient pas k'.

$$A = \bigwedge_{i\in[1,n]} \bigwedge_{j\in[1,n]} \bigvee_{k\in[1,n]} \left(V_{ijk} \bigwedge_{\substack{k'\in[1,k]\\k'\neq k}} \neg V_{i,j,k'} \right)$$

« On ne doit pas avoir deux chiffres identiques sur une même colonne » : pour toute colonne $i \in [1, n]$, pour toutes lignes $j \neq j'$ dans $[1, n]$, pour toute valeur $k \in [1, n]$: si la case (i, j) contient k, alors la case (i, j') ne contient pas k.

$$B = \bigwedge_{i\in[1,n]} \bigwedge_{j\in[1,n]} \bigwedge_{\substack{j'\in[1,n]\\j'\neq j}} \bigwedge_{k\in[1,n]} (V_{i,j,k} \Rightarrow \neg V_{i,j',k})$$

« On ne doit pas avoir deux chiffres identiques sur une même ligne » : pour toute ligne $j \in [1, n]$, pour toutes colonnes $i \neq i'$ dans $[1, n]$, pour toute valeur $k \in [1, n]$: si la case (i, j) contient k, alors la case (i', j) ne contient pas k.

$$C = \bigwedge_{j\in[1,n]} \bigwedge_{i\in[1,n]} \bigwedge_{\substack{i'\in[1,n]\\i'\neq i}} \bigwedge_{k\in[1,n]} (V_{i,j,k} \Rightarrow \neg V_{i',j,k})$$

« On ne doit pas y avoir deux chiffres identiques dans l'une des sous-grilles de taille $\ell * \ell^{\smile}$: pour

tous $x, y \in [1, \ell]$,

$$D = \bigwedge_{x,y \in [0,\ell-1]} \bigwedge \bigwedge_{\substack{i,i' \in [1+\ell x, \ell+\ell x] \\ i \neq i'}} \bigwedge_{j,j' \in [1+\ell y, \ell+\ell y]} \bigwedge_{k \neq j'} \bigwedge_{k \in [1,n]} (V_{i,j,k} \Rightarrow \neg V_{i',j',k})$$

Pour assurer qu'une grille donnée est un Sudoku, il faut indiquer les chiffres déjà inscrits et vérifier qu'il y a une et une seule solution au problème. Par exemple, pour la grille donnée en exemple, on crée la formule :

$$E = V_{1,9,5} \wedge V_{2,9,3} \wedge V_{5,9,7} \wedge \dots$$

On cherche alors les modèles de la formule $A \wedge B \wedge C \wedge D \wedge E$. Si il n'y a qu'un seul modèle, alors la grille est un Sudoku dont la solution est donnée par ce modèle.

4.3 Problème NP-complets

Définition 4.2. Un problème Π est dit **NP-difficile** si tous les problèmes de NP peuvent se réduire à lui.
Si de plus, Π est dans NP, on dit qu'il est **NP-complet**.

Ainsi, les problèmes NP-Complets sont les problèmes les plus difficile dans la classe NP.

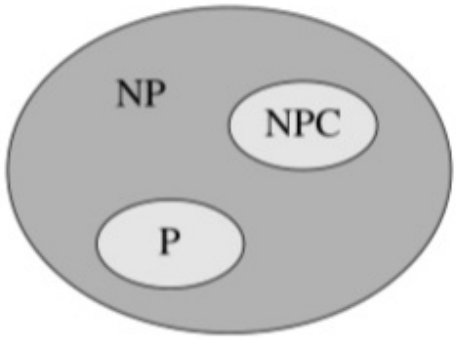


FIGURE 4.1 –

Remarque 4.3. Soit $\Pi \in \text{NP}-Complet$, si on a $\Pi \in \text{P}$ alors on aurait : P = NP. Autrement dit ; si on arrive à résoudre un problème NP-complet en temps polynomial, alors tous les problèmes NP sont résolubles en temps polynomial.

Théorème 4.2. [10] [**Théorème de Cook 1971**] SAT est NP-complet

Nous allons maintenant donner l'idée de la preuve du théorème de Cook.
Première partie : il faut montrer que SAT fait partie de NP.

Il est facile de vérifier que la donnée d'une fonction d'interprétation pour les variables booléennes satisfait ou non la formule logique, donc : SAT $\in$ NP.
Seconde partie : est beaucoup plus longue et difficile, Sa démonstration repose sur une description du déroulement du programme d'une machine de Turing à l'aide de propositions et de clauses de façon à ce que la donnée de l'ensemble des états du ruban soit équivalente à la donnée d'un ensemble de valeurs satisfaisant ces clauses.
Autrement dit en codant l'exécution de tout algorithme sur une machine de Turing par une expression en forme normale conjonctive.

Exemple 4.1 (**Problème NP-complet**)**.**
Pour montrer qu'un problème Π donné est NP-complet, on montre qu'il existe un problème NP-complet qui se réduit à Π.

La classe NP-complet bénéficie d'un suivi permanent par des chercheurs, et cette classe ne cesse de s'agrandir. On a recensé plus de 3000 problèmes NP-complet en 2019.

On peut citer :

- $3 - \text{SAT}$
- COLORING
- Problème du cycle hamiltonien
- Problème du voyageur du commerce
- Sodoku
- Problème de Sac-à-dos

Conclusion

If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.
John von Neumann

Pour la résolution d'un problème mathématique, on est amené à utiliser des algorithmes adéquats. La complexité d'un algorithme mesure sa qualité, c'est-à-dire les ressources en temps et en espace mémoire que nécessite son exécution. Certain algorithmes, trop boulimiques, sont inutilisables dans la pratiques. C'est pourquoi, il est primordial de trouver des algorithmes de plus en plus performants et de pouvoir classifier les problèmes en fonctions de la complexité des algorithmes les résolvant.

Nous nous sommes intéressés à l'intérieur de ce mémoire, à l'étude des problèmes de décision et à la notion de décidabilité. La théorie de la décidabilité est capitale à la fois en mathématiques et en informatiques. l'introduction du concept de la machine de Turing nous permet de formaliser la décidabilité et permet une étude théorique de la notion de complexité algorithmique. L' étude mathématique des différentes notions y afférentes confère à cette discipline un caractère clair et rigoureux et permet de développer plusieurs pistes de recherche. Nous avons présenté pour illustration des problèmes non décidables, à l'exemple du célèbre problème d'arrêt.

L'étude des différentes classes de problèmes nous permet de constater l'ampleur et l'importance des questions qui restent sans réponses à nos jours à l'instar de la fameuse question : **P=NP** ? Le problème **SAT** qui joue un rôle central, et qui est à la base un problème de décision en logique mathématique, par sa simplicité nous permet de décrire plusieurs autres problèmes grâce à la réduction polynomiale. Ceci nous amène à la **NP-complétude**, classe des problèmes les plus difficiles de la classe **NP** où il est établi que l'existence d'un algorithme efficace pour un quelconque problème de cette classe permet de résoudre efficacement tous les autres problèmes appartenant à cette classe et ainsi établir que **P=NP**. Même si cela parait impossible de l'avis de la plupart des chercheurs dans sans domaine.

Des voies de recherches sont énormes dans ce domaine, certaines ouvre la porte à un très grand nombre d'applications. À l'exemple de la recherche de meilleurs algorithmes (solveurs SAT,...).

Bibliographie

[1] B. Jack Copeland
The Church-Turing thesis. In Edward N. Zalta, editor, The Stanford Encyclopedia of Philosophy. Stanford University, Fall 2002

[2] C. Berge
Theorie des graphes et ses application, Dunod 1967

[3] C. H Papadimitriou
Computational Complexity,Pearson Édition 1993.

[4] Fratani, avec addenda par Luigi Santocanale
Cours Logique et Calculabilité, Version du 28 janvier 2014.

[5] J. A. Bondy et U.S.R. Murty
Théorie des Graphes, Édition 2008

[6] M. R. Garay, D.S.Johnson
Compteurs and intractability, Édition 1979

[7] O. Carton
Langages formels, Calculabilité et Complexité, Édition 2007/2008

[8] P. Wolper
introduction à la calculabilité, 3ème édition 2006.

[9] S. Perifel
Complexité algorithmique, édition 2014.

[10] T. Cormen, C.Leiserson, R.Rivest, C.Stein
Introduction à L'algorithmique, 2e édition

Chapitre A

Annexes

Dans cette annexe nous présentation les principales définitions de notion de la théorie des graphes évoquée dans ce mémoire.

Pour un exposé plus détaillé sur la théorie des graphes, le lecteur peut se référer à [2, 5]

Graphe

Un graphe G est constitué d'un ensemble de sommets reliés entre eux par des liens appelés : Arêtes dans le cas des graphes non-orientés ou arcs dans le cas des graphes orientés.

Chaîne et Chaîne simple

Suite finie de sommets reliés entre eux par une arête et chaîne simple qui n'utilise pas deux fois la même arête.

Chaîne hamiltonienne

Chaîne simple passant par tous les sommets d'un graphe une et une seule fois.

Chemin

Suite de sommets reliés par des arcs dans un graphe orienté.

Cycle

Chaîne qui revient à son point de départ.

Cycle hamiltonien

Cycle simple passant par tous les sommets d'un graphe une et une seule fois.

Graphe hamiltonien

Graphe qui possède un cycle hamiltonien.

Coloration d'un graphe

Nombre minimal de couleurs permettant de colorier les sommets d'un graphe, de telle sorte que deux sommets adjacents n'aient pas la même couleur.

Connexité et forte connexité

Un graphe non-orienté est connexe si pour tout couple de sommets x et y, il existe une chaîne reliant x à y.
Un graphe orienté est fortement connexe si pour tout couple de sommets x et y, il existe une chemin reliant x à y.

Les composantes connexes

On appelle composante connexe un ensemble de sommets, qui on deux à deux la relation de connexité, de plus tout sommet en dehors de la composante n'a pas de relation de connexité avec les sommets de cette composante.