

INITIATION À LA THÉORIE DE LA COMPLEXITÉ

Master DID — Outils et algorithmes pour la protection de l'information.
(Version 2.5 du 4 mai 2023)

Jean-Pierre ZANOTTI

TABLE DES MATIÈRES

1. Avertissement	1
2. Introduction	1
3. Une brève histoire de la calculabilité	2
4. La machine de Turing	6
5. Complexité et modèles algorithmiques	8
6. La classe P	10
7. La classe NP	11
8. La classe NP-complet	13
9. Le théorème de Cook	14
10. Quelques problèmes NP-complets	20
11. Annexe : la machine RAM	33
12. Annexe : Quelques rappels de logique propositionnelle	35
13. Annexe : Mini lexique sur les graphes	36
Bibliographie	37
Index	38

1. AVERTISSEMENT

Ce polycopié a été rédigé initialement *dans l'urgence* dans le cadre du plan de continuité pédagogique imposé pour assurer des enseignements durant le confinement lors de la première vague de l'épidémie du COVID19. Il est incomplet et contient des imprécisions et certainement des erreurs et des approximations. Il n'a pas vocation à remplacer un cours dans l'espace physique, mais simplement à donner une base de travail pour les étudiants de master. Les corrections éventuelles et/ou les compléments seront répercutés sur la page web où il est hébergé.

Toutes les questions, commentaires, suggestions et corrections sont les bienvenu-e-s : zanotti@univ-tln.fr.

2. INTRODUCTION

L'évolution perpétuelle des techniques et les progrès fulgurants réalisés dans le domaine de l'informatique aboutissent à une dématérialisation croissante de pans entiers de notre univers quotidien et à une virtualisation de tâches de plus en plus variées. Ces mutations ont fait ressurgir depuis quelques décennies des questions autour de la notion de calcul qui semblaient jusque là réservées aux logiciens et aux philosophes du début du XX-ème siècle. La cryptographie, l'analyse numérique, l'intelligence artificielle, la théorie de la démonstration, entre autres, ont stimulé les recherches sur la notion de calcul qui n'ont jamais été aussi vivaces qu'aujourd'hui en informatique.

On cherche à répondre formellement à trois questions initiées par les mathématiques et dont les prolongements s'avèrent fondamentaux en informatique. Pour y répondre, trois théories ont vu le jour. Dans l'ordre chronologique et de manière informelle :

- (1) Que peut-on calculer ? Question étudiée dans la *théorie de la calculabilité* apparue au début du siècle dernier.
- (2) Que peut-on calculer efficacement ? Question abordée dans la *théorie de la complexité*, qui fait l'objet de ce cours de master et qui s'est développée dans les années 1960.
- (3) Que peut-on approximer efficacement ? C'est l'objet de la *théorie de l'approximation polynomiale* qui s'est développée dans la foulée de la théorie de la complexité¹

Ce cours de première année de master se focalise sur la deuxième question, il a pour objectif de familiariser les étudiants en informatique avec la théorie de la complexité qui est le domaine de l'informatique théorique qui évalue les quantités de ressources *en temps* et *en espace* nécessaires à un algorithme pour résoudre un problème. Ces questions ont été abordées dès la première année en estimant le nombre d'opérations effectuées par

1. bien que l'approximation polynomiale soit un sujet d'étude des mathématiciens depuis bien plus longtemps.

les algorithmes, puis formalisées et approfondies en deuxième et en troisième année, notamment en introduisant les notations asymptotiques. Il s'agit cette année d'aller un peu plus loin et de s'intéresser à la classification des problèmes et particulièrement ceux pour lesquels on ne connaît pas encore d'*algorithme* efficace pour les résoudre, notions que nous serons amenés à expliciter. La théorie de la complexité est un outil fondamental dans l'élaboration des protocoles de chiffrement et leur cryptanalyse.

Pour résumer très grossièrement ce qui va suivre, la théorie de la complexité a principalement pour objectif de séparer les problèmes "faciles" des problèmes "probablement difficiles".

3. UNE BRÈVE HISTOIRE DE LA CALCULABILITÉ

Avant d'aborder la théorie de la complexité, nous allons brièvement évoquer quelques éléments de la théorie de la calculabilité/décidabilité. La théorie de la complexité ne pouvait en effet éclore qu'après avoir répondu à la question « Que peut-on calculer ? » et ceci indépendamment des ressources en temps et en espace dont on dispose, donc *intrinsèquement*. Tout ceci suppose que l'on définisse au préalable ce qu'est un *calcul*, un *algorithme*.

L'idée intuitive que l'on a d'un calcul/algorithme est celle d'un procédé "mécanisé" qui transforme une matière première (l'entrée) en produit fini (la sortie) en un temps fini et en appliquant des règles strictes et prédéfinies. Le procédé doit être déterministe, reproductible, général, etc. Mais pour pouvoir raisonner sur le concept de calcul/algorithme, il est indispensable de s'appuyer sur un modèle abstrait qui calque au mieux l'idée intuitive que l'on s'en fait.

L'*algorithmique* est une science très ancienne, elle a débuté bien avant la naissance des ordinateurs et avant même d'être baptisée ainsi en hommage au mathématicien perse Abu Abdullah Muhammad ibn Musa *al-Khwarizmi* né en 780. L'algorithme du PGCD d'Euclide ou celui du crible d'Eratosthène datent d'environ 300 ans avant JC. Depuis, le calcul n'a cessé d'être une préoccupation centrale des mathématiciens. Blaise Pascal invente une machine à calculer dès le milieu du 17^{ème} siècle, suivi de près par Leibnitz. En 1801 Joseph Jacquard conçoit le métier à tisser programmé par des cartes perforées et Ada Lovelace travaille avec Charles Babbage sur sa machine différentielle et analytique (un embryon d'ordinateur) au milieu du 19^{ème} siècle.

Le mathématicien allemand David Hilbert présente au début du 20^{ème} siècle un programme de recherche qui a pour objectif la refondation des mathématiques. Ce programme a été motivé par l'apparition de paradoxes engendrés par certaines constructions ensemblistes, et en particulier la manipulation d'ensembles infinis. L'un des paradoxes les plus célèbres, mis en évidence par le philosophe et mathématicien anglais Bertrand Russell, s'appuyait sur le *principe d'abstraction* de Cantor réfuté depuis dans la théorie moderne des ensembles de Zermelo-Fraenkel. Dans le langage de la théorie ZF, le principe d'abstraction de Cantor affirme que tout prédicat $P(x)$ est *collectivisant* en x , c'est-à-dire qu'il existe toujours un ensemble X dont les éléments sont précisément ceux qui satisfont le prédicat $P(x)$:

$$X := \{x \mid P(x)\}.$$

Russel considère alors le prédicat $P(x)$ suivant :

$$x \notin x.$$

D'après le principe d'abstraction, l'ensemble $X = \{x \mid x \notin x\}$ existe et le tiers exclu² impose que $X \in X$ ou $X \notin X$. Dans le premier cas, si $X \in X$ alors X ne satisfait pas le prédicat $P(x)$ et dans ce cas $X \notin X$ ce qui est contradictoire. Dans le second cas, si $X \notin X$ alors X satisfait le prédicat $P(x)$ et donc $X \in X$ ce qui est à nouveau contradictoire.

On comprend bien que tout l'édifice mathématique est en grave péril s'il est possible de démontrer des théorèmes de la forme $P \wedge \neg P$, la refondation des mathématiques apparaît alors impérieuse. David Hilbert est également convaincu que l'on doit toujours être en mesure de prouver ou de réfuter *toute* proposition mathématique, on doit pouvoir *démontrer* qu'elle est vraie ou fausse. Il complète quelques années plus tard, en 1928, son programme de refondation en posant le célèbre *problème de la décidabilité* exposé ici informellement : peut-on automatiser l'analyse d'une proposition et déterminer sa valeur de vérité? Autrement dit, existe-t-il un algorithme qui permet d'analyser une proposition pour *décider* si celle-ci est vraie ou fausse?

Le logicien autrichien Kurt Gödel ruine les espoirs de Hilbert en 1931 avec ses deux *théorèmes d'incomplétude*. Il montre, d'une part que toute théorie mathématique suffisamment riche pour coder l'arithmétique, contient des propositions dont on ne peut prouver qu'elles sont vraies ni qu'elles sont

2. Toute proposition P est vraie ou fausse, i.e. la formule $(P \vee \neg P)$ est vraie.

fausses (elles sont *indécidables*) et d'autre part que la cohérence d'une théorie ne peut se démontrer à l'aide de cette même théorie. Quelques années plus tard, en 1936, Alan Turing, grâce à un modèle de machine qui porte aujourd'hui son nom, et Alonzo Church grâce au λ -calcul, démontrent en substance qu'une théorie mathématique capable de formaliser le langage de l'arithmétique est algorithmiquement indécidable.

D'autres modèles abstraits ont été proposés à la même époque, en particulier le modèle des fonctions récursives partielles, mais Steven Kleene démontre que tous ces modèles sont équivalents, ce que l'on peut calculer avec l'un peut l'être avec un autre et réciproquement. Un modèle plus récent et beaucoup plus proche des ordinateurs tels qu'on les connaît à présent, a vu le jour dans les années 1960-1970, il s'agit du modèle de la machine RAM que nous avons étudié en cours d'algorithmique de deuxième année et qui est rappelé en annexe. Là encore, on a pu montrer que ce modèle était équivalent aux autres. Il est aujourd'hui admis que tout nouveau modèle abstrait de calcul respectant les caractéristiques informelles d'un algorithme (si l'on excepte le calcul quantique qui définit un nouveau paradigme) sera équivalent aux autres modèles. Ce résultat est connu sous le nom de *thèse de Church-Turing* et a largement été validé.

Avant de donner une définition abstraite d'un algorithme, formalisons l'articulation entre un algorithme et son entrée/sortie. On rappelle que si Σ est un alphabet fini, on désigne par Σ^* l'ensemble infini des *mots* définis sur cet alphabet. Les instances d'un problème qui doit être résolu par un algorithme sont *codées* par les mots d'un langage Σ^* . Ce processus de traduction est qualifié de *schéma d'encodage*.

Exemple 1. (1) Le triplet $(255, 255, 0)$ de $\llbracket 0, 255 \rrbracket^3$ code la couleur **jaune** pour le schéma d'encodage (R, V, B) des couleurs sur un écran en synthèse additive. (2) Le quadruplet $192.168.1.1$ de $\llbracket 0, 255 \rrbracket^4$ code l'adresse d'une machine dans le protocole de communication IP.

L'articulation entre un algorithme et les instances du problème se schématise naturellement par le diagramme de la figure 1 dans lequel x est l'encodage d'une instance du problème considéré (l'*entrée*) et y est le résultat du calcul exprimé dans un langage Γ^* (la *sortie*). Ce résultat est *décodé* pour être interprété dans le monde réel.

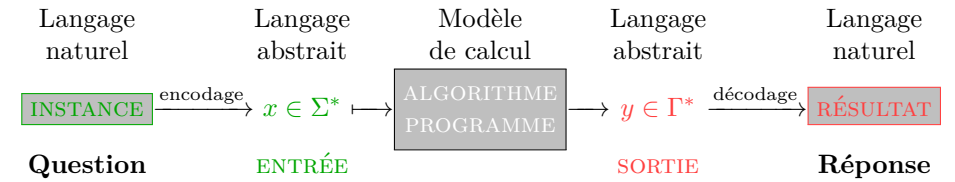


FIGURE 1. Schéma fonctionnel d'un modèle algorithmique.

La similitude de ce schéma avec la description symbolique d'une *fonction*

$$f : \begin{array}{l} X \rightarrow Y \\ x \mapsto y \end{array}$$

est manifeste, avec l'entrée x prise dans l'ensemble de départ X et la sortie y dans l'ensemble d'arrivée Y . Nous allons pousser plus loin cette analogie en éliminant le superflu, en particulier en montrant que l'on peut se contenter du langage des entiers naturels pour décrire les entrées et les sorties.

Définition 1. Deux ensembles X et Y sont dits *équipotents* s'il existe une bijection entre X et Y . Dans ce cas, on écrit $X \approx Y$.

Théorème 1. Soit Σ un alphabet fini. Alors $\Sigma^* \approx \mathbb{N}$.

Démonstration. Pour cela, on s'appuie sur l'*arbre lexicographique* associé au langage Σ^* . Il s'agit d'un *graphe connexe* sans circuit et enraciné (il existe une seule racine, i.e. un seul sommet qui n'a aucun arc entrant) construit de la manière suivante : on part de la racine de l'arbre à laquelle on attache q branches étiquetées de la gauche vers la droite par les q symboles de Σ suivant l'ordre alphabétique (on peut toujours supposer que l'alphabet Σ est totalement ordonné). On recommence indéfiniment la même construction pour chacun des q nouveaux nœuds ainsi créés.

On associe ensuite à chaque nœud de l'arbre un mot de Σ^* obtenu en concaténant les étiquettes le long du chemin qui relie la racine de l'arbre à ce nœud (on peut montrer qu'un tel chemin existe toujours et est unique). La numérotation des nœuds de l'arbre dans l'ordre de lecture en partant de 0 pour la racine associée au mot vide ε établit une bijection évidente entre Σ^* et \mathbb{N} (la preuve formelle est laissée en exercice 1). Par exemple, dans l'arbre lexicographique de la figure 2 pour l'alphabet $\Sigma := \{a, b, c\}$, le

noeud numéro 24 est associé au mot *bac* et le noeud numéro 10 est associé au mot *ca*. \square

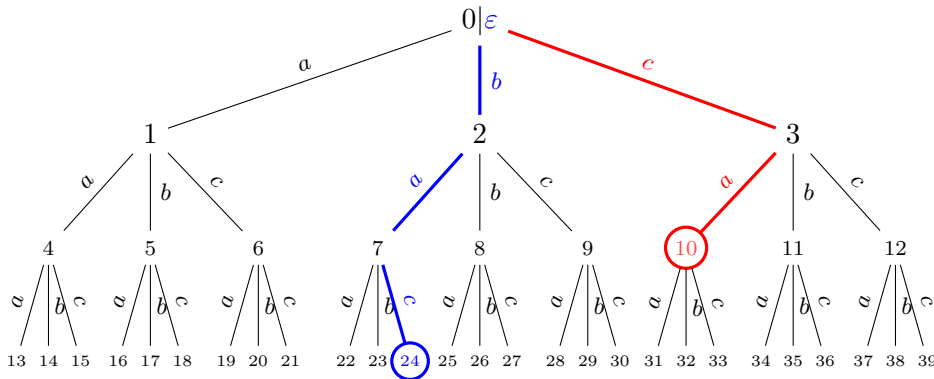


FIGURE 2. Les 4 premiers niveaux de l'arbre lexicographique du langage $\{a, b, c\}^*$ et l'indexation des mots.

On peut donc remplacer les langages Σ^* et Γ^* dans le schéma de la figure 1 par l'ensemble des entiers naturels \mathbb{N} , sans perdre en généralité, autrement dit se contenter d'étudier des algorithmes qui transforment des entiers naturels en entiers naturels. Notons que le fonctionnement des programmes informatiques développés sur des modèles de machines bien réelles est parfaitement conforme à cette vision en apparence "réductrice" de l'algorithmique.

Le problème de la calculabilité se résume à présent à la question centrale :

Une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ donnée est-elle *calculable*?

C'est-à-dire, existe-t-il un algorithme dans le modèle de calcul considéré qui prend en entrée un entier naturel n quelconque et qui renvoie en sortie la valeur $f(n)$ image de n par f ?

Exemple 2. La fonction constante définie par $n \mapsto k$ où $k \in \mathbb{N}$ est calculable, le programme suivant dans le modèle de la machine RAM en est la preuve :

0. LOAD #k
1. WRITE
2. STOP

Un problème de décidabilité est un problème auquel on répond par l'affirmative ou la négative. Par exemple, un nombre entier est-il premier? On encode les instances d'un tel problème avec des entiers naturels, ce qui définit implicitement une fonction de \mathbb{N} dans $\{0, 1\}$. Si cette fonction est calculable alors on dit que le problème associé est *décidable*.

Exemple 3. Le problème de la parité d'un nombre entier est un problème décidable puisque la fonction définie par

$$f : n \mapsto \begin{cases} 1 & \text{si } n \text{ est pair} \\ 0 & \text{sinon} \end{cases}$$

est calculable. Voici en effet un algorithme de la machine RAM qui la calcule :

- | | |
|------------|------------|
| 0. READ | 4. JUMP 6 |
| 1. MOD #2 | 5. LOAD #1 |
| 2. JUMZ 5 | 6. WRITE |
| 3. LOAD #0 | 7. STOP |

Rappelons qu'il n'est même pas nécessaire de faire référence à un modèle de calcul en particulier puisqu'ils sont tous *équivalents*, cependant on dit aussi que les deux fonctions que nous venons d'étudier sont RAM-calculables quand on souhaite préciser le modèle de calcul qui a été utilisé.

Aussi surprenant que cela puisse paraître, il existe des fonctions que l'on ne peut pas calculer, au sens où il n'existe pas d'algorithme pour le faire. Il est important de noter qu'il ne s'agit pas là de "monstres" parfois tapis dans quelques recoins de théories mathématiques et qu'il faut débusquer, comme par exemple la fonction de Weierstrass qui est continue partout mais dérivable nulle part. Ici, ces fonctions sont associées à des problèmes bien concrets. Pour ne pas déborder du cadre contraint de ce cours, nous nous contenterons d'en donner une preuve existentielle.

Définition 2. Un ensemble X est dit *fini* s'il existe un entier naturel n appelé *cardinal* de X tel que $X \approx \llbracket 1, n \rrbracket$. Dans le cas contraire l'ensemble

X est dit **infini**. On dit que X est **dénombrable** si $X \approx \mathbb{N}$, et au plus **dénombrable** s'il est fini ou dénombrable.

Ce qu'il faut retenir ici, c'est que l'on peut *indexer* les éléments d'un ensemble X au plus dénombrable. S'il est fini, on dispose d'une bijection $x : \llbracket 1, n \rrbracket \rightarrow X$ et on note x_i au lieu de $x(i)$ et donc $X = \{x_1, x_2, \dots, x_n\}$. S'il est dénombrable, on dispose cette fois d'une bijection $x : \mathbb{N} \rightarrow X$ et on a $X = \{x_i \mid i \in \mathbb{N}\}$.

Théorème 2. *Il existe des fonctions $f : \mathbb{N} \rightarrow \mathbb{N}$ non-calculables.*

Démonstration. Comme convenu, nous ne donnerons qu'une preuve existentielle de ce théorème. Pour cela, nous allons montrer que :

- (1) L'ensemble des algorithmes est *dénombrable*
- (2) L'ensemble $\mathbb{N}^{\mathbb{N}}$ des applications de \mathbb{N} dans \mathbb{N} n'est pas dénombrable.

(1) Le premier point est laissé en exercice.

	0	1	2	3	...	k	...	n	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$...	$f_0(k)$...	$f_0(n)$...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$...	$f_1(k)$...	$f_1(n)$...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$...	$f_2(k)$...	$f_2(n)$...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$...	$f_3(k)$...	$f_3(n)$...
\vdots
f_k	$f_k(0)$	$f_k(1)$	$f_k(2)$	$f_k(3)$...	$f_k(k)$...	$f_k(n)$...
\vdots
\vdots
\vdots

TABLE 1. Images des applications de $\mathbb{N} \rightarrow \mathbb{N}$.

(2) Pour le second, nous allons le prouver par l'absurde à l'aide d'un argument dit *diagonal*. Supposons que l'ensemble des applications de \mathbb{N} dans \mathbb{N} soit dénombrable. Dans ce cas, on peut les indexer et les ranger dans la première colonne de la table 3 en énumérant à leur droite les images des

entiers naturels. On définit alors une application $f : \mathbb{N} \rightarrow \mathbb{N}$ dont les images diffèrent des valeurs présentes sur la diagonale de cette table :

$$\forall n \in \mathbb{N} \quad f(n) := \begin{cases} 1 & \text{si } f_n(n) = 0. \\ f_n(n) - 1 & \text{sinon.} \end{cases} \quad (1)$$

Notons que toute autre construction de f telle que $f(n) \neq f_n(n)$ conviendrait. Puisque la colonne à gauche de la table contient *toutes* les applications de \mathbb{N} dans \mathbb{N} , il existe un entier $k \in \mathbb{N}$ tel que $f = f_k$. Par définition, deux applications sont égales si et seulement si elles ont même ensemble de départ, même ensemble d'arrivée et même graphe. Les applications f et f_k ont toutes deux pour ensemble de départ et d'arrivée \mathbb{N} , l'égalité des graphes se traduit par la proposition suivante :

$$\forall n \in \mathbb{N} \quad f(n) = f_k(n).$$

Et la contradiction apparaît pour $n := k$. En effet, $f(k) \neq f_k(k)$ par construction d'après (1), l'ensemble $\mathbb{N}^{\mathbb{N}}$ n'est donc pas dénombrable. \square

Nous concluons cette section en citant un résultat que nous ne démontrerons pas ici mais qui sert d'amorce à de nombreuses preuves de non-calculabilité. Il s'agit du problème de l'*arrêt d'un programme*, qui consiste à déterminer si un programme s'arrête ou non. C'est un problème indécidable. Une première conséquence concrète de ce résultat est qu'il n'existe pas de compilateur capable de décider s'il y a une boucle infinie dans un code informatique.

Exercice 1 Montrez que l'ensemble des algorithmes est dénombrable. Indication : considérez un algorithme comme un mot défini sur un alphabet particulier et utiliser l'arbre lexicographique.

Exercice 2 (1) Écrivez une fonction C appelée `Int2Alpha(A, n)` qui a pour paramètres un alphabet A donné (un sous-ensemble de l'UTF-8) et un nombre entier n et qui renvoie la chaîne de caractère associée à cet entier dans l'arbre lexicographique.

(2) Écrivez une fonction C appelée `Alpha2Int(A, s)` qui renvoie l'index de la chaîne de caractères s sur l'alphabet A dans l'arbre lexicographique.

(3) Reprenez les précédentes questions en écrivant l'équivalent mathématique de ces deux fonctions. Achevez la preuve du théorème 1.

Exercice 3 Une fois que vous aurez étudié la machine de Turing à la section suivante, trouvez une numérotation effective des algorithmes sur ce modèle, c'est-à-dire une application injective de l'ensemble des algorithmes dans l'ensemble des entiers naturels \mathbb{N} .

4. LA MACHINE DE TURING

La machine de Turing est largement utilisée en théorie de la programmation. La raison principale tient à son extrême simplicité qui a permis d'établir des résultats sans nul doute beaucoup plus difficiles à démontrer avec des modèles moins rudimentaires. Cette machine est en fait le modèle le plus *simple* que l'on puisse concevoir et qui satisfait aux critères informels mais universels qui caractérisent un algorithme (déterminisme, discrétion, finitude, généralité, reproductibilité, etc.)

Remarque. Le modèle des automates finis étudié en théorie des langages est plus simple que la machine de Turing, mais ne permet pas, entre autres, de reconnaître les mots binaires de la forme $0^n 1^n$ ce qui limite sensiblement sa portée en tant que modèle de calcul.

Il faut dès à présent garder à l'esprit que la machine de Turing est un modèle universel de calcul et qu'elle peut calculer tout ce que n'importe quel ordinateur physique peut calculer. Inversement, ce qu'elle ne peut pas calculer ne peut l'être non plus par un ordinateur. Elle résume donc de manière saisissante le concept d'*ordinateur* et constitue un support idéal pour raisonner autour de la notion d'*algorithme* de calcul ou de *démonstration*.

Pour reprendre l'idée même d'Alan Turing, sa machine n'est qu'une re-conversion minimaliste d'une machine à écrire que l'on pourrait contrôler à l'aide d'un programme. Une machine à écrire peut écrire ou effacer un symbole sur la feuille (même si l'effacement mécanisé avec un ruban correcteur n'était pas présent sur les premiers modèles), se déplacer sur le caractère à gauche ou à droite du dernier caractère écrit. Au lieu d'opérer en deux dimensions sur une feuille de papier, la machine de Turing se contente d'une *bande* potentiellement infinie à gauche et à droite et segmentée en *cases* pouvant contenir chacune un *symbole*. Une *tête de lecture/écriture* joue le

rôle des barres à caractères et agit sur la bande en suivant les règles d'un *programme*. Par commodité, on indexe souvent les cases de cette bande avec l'ensemble des entiers relatifs \mathbb{Z} , la tête de lecture/écriture étant initialement placée sur la case d'indice 1 (cf. figure 3).

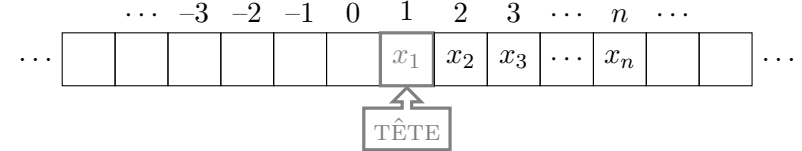


FIGURE 3. Schématisation d'une machine de Turing.

La bande joue à la fois le rôle de périphérique d'entrée et de sortie. Initialement vierge, on y inscrit (virtuellement) les données à traiter par le programme (entrée), codées par un mot $x = x_1 x_2 \dots x_n \in (\Sigma \cup \{\square\})^*$ où Σ est un alphabet fini et \square est un symbole particulier appelé *blanc*. Par convention le premier symbole x_1 du mot x est placé dans la case d'indice 1 où se trouve la tête de lecture/écriture avant l'exécution du programme. On désignera parfois par $B[i]$ le contenu de la case d'indice i de la bande, autrement dit, on voit la bande comme une application de \mathbb{Z} dans $\Sigma \cup \{\square\}$. Par exemple, le couple d'entiers $(1, 2)$ pourrait être codé par le mot $1 \square 1 0$ sur l'alphabet binaire $\Sigma = \{0, 1\}$, occupant ainsi 4 cases de la bande, le symbole blanc \square indiquant que la case d'indice 2 reste vierge, permettant ainsi de séparer les deux termes du couple.

Pendant l'exécution du programme, la tête de lecture/écriture peut se déplacer à gauche ou à droite de la case qu'elle visite, lire son contenu, y écrire un symbole de Σ ou le symbole \square pour l'effacer. À la fin de l'exécution du programme, le mot sur la bande constitue le résultat du calcul (sortie).

Le fonctionnement est discret et ce que réalise la machine à chaque étape ne dépend que de deux paramètres : l'état de la machine et le symbole lu, c'est-à-dire celui présent dans la case sur laquelle est placée la tête de lecture/écriture (le symbole x_1 dans la figure 3). Pour écrire ses programmes et contrôler la machine, le programmeur dispose d'un ensemble fini d'états Q de son choix et fixe l'état initial dans lequel la machine se trouve au début de l'exécution. C'est le programme qui va déterminer ce que la machine va faire en fonction du couple (p, s) constitué par l'état courant p

de la machine et le symbole lu s . Si la case est vide, on considère que le symbole lu est \square .

Définition 3. Soit Σ un alphabet fini, Q un ensemble fini d'états, \square le symbole blanc et $\Sigma_{\square} := \Sigma \cup \{\square\}$. On appelle règle l'expression $p, s : e, d, q$ où le couple (p, s) est une condition :

- $p \in Q$ est l'état courant de la machine.
 - $s \in \Sigma_{\square}$ est le symbole lu, i.e. celui sous la tête de lecture/écriture.
- et le triplet (e, d, q) l'opération à réaliser :
- e est le symbole d'écriture si $e \in \Sigma$ ou un effacement si $e = \square$.
 - $d \in \{\leftarrow, \rightarrow\}$ est la direction où déplacer la tête de lecture/écriture.
 - $q \in Q$ est le nouvel état dans lequel se trouvera la machine.

Notons ρ l'état dans lequel se trouve la machine et σ le symbole sous la tête de lecture/écriture à un instant t de l'exécution de l'algorithme. Chaque règle s'interprète de la manière suivante :

SI $((\rho = p)$ **ET** $(\sigma = s))$ **ALORS**

- On écrit le symbole e en écrasant le symbole s
- On déplace la tête de lecture/écriture dans la direction d .
- On passe de l'état p à l'état q .

FINSI

Si $e = \square$, on dit plutôt qu'on efface la case. Si aucune règle du programme ne contient le couple condition (p, s) où p est l'état de la machine et s le symbole lu, alors la machine s'arrête.

Définition 4. Un algorithme ou programme est un ensemble fini de règles d'une machine de Turing.

Exemple 4. On considère l'alphabet binaire $\Sigma := \{0, 1\}$ et l'ensemble des états $Q := \{q_0, q_1\}$ où q_0 est l'état initial. Le programme ci-dessous remplace un mot binaire par sa négation, i.e. les 0 sont remplacés par des 1 et réciproquement. Par convention, les bits d'un mot binaire $x_1x_2 \dots x_n$ de

longueur n sont inscrits dans les cases 1 à n respectivement.

- $q_0, 0 : 1, \rightarrow, q_0$ Si le symbole lu est 0, on écrit 1 et on va à droite.
- $q_0, 1 : 0, \rightarrow, q_0$ Si le symbole lu est 1, on écrit 0 et on va à droite.
- $q_0, \square : \square, \leftarrow, q_1$ Si la case est vide, on va à gauche et on passe à l'état q_1 .

Les deux premières règles réalisent la boucle, "tant qu'on lit un symbole binaire on le remplace par sa négation et on va à droite" et on ne change pas d'état. Une fois que la machine a lu et remplacé le dernier symbole binaire x_n par sa négation, la tête de lecture/écriture se retrouve sur la première case vide à droite de la séquence d'entrée et comme il n'y a aucune règle dont la première projection du couple condition est égale à q_1 , la machine s'arrête avec la tête de lecture/écriture sur le dernier bit de la séquence et dans l'état q_1 .

Un simulateur est accessible à l'adresse <http://zanotti.univ-tln.fr/turing>. Nous pouvons à présent donner une définition formelle de cette machine de Turing.

Définition 5. Une machine de Turing est un quintuplet $(Q, q_0, \Sigma, \square, \delta)$ où

- (1) Q est l'ensemble des états (fini) ;
- (2) $q_0 \in Q$ est l'état initial ;
- (3) Σ est l'alphabet (fini) ;
- (4) $\square \notin \Sigma$ est le symbole blanc et $\Sigma_{\square} := \Sigma \cup \{\square\}$.
- (5) $\delta : (Q \times \Sigma_{\square}) \rightarrow (\Sigma_{\square} \times \{\leftarrow, \rightarrow\} \times Q)$ est la fonction de transition.

Les règles du programme d'une machine de Turing telles que nous venons de les présenter ne sont rien d'autre qu'une réécriture plus commode des couples du graphe de la fonction de transition. On écrit simplement $p, s : e, d, q$ au lieu de

$$\delta(p, s) = (e, d, q). \quad (2)$$

Exercice 4 Comment définir l'arrêt de la machine de Turing à partir de sa définition formelle ? Indication : traduire l'absence de règle à suivre avec la fonction de transition.

Exercice 5 L'ordre dans lequel les règles sont écrites dans le programme a-t-il de l'importance ? Pourquoi ?

Exercice 6 Montrez que la fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ définie par $n \mapsto n + 1$ est RAM-calculable et Turing-calculable (écrivez un algorithme pour chacun des deux modèles). On supposera que $\Sigma := \{0, 1\}$ pour la machine de Turing.

Exercice 7 Soit $k \in \mathbb{N}$. Démontrez que la fonction constante $f : \mathbb{N} \rightarrow \mathbb{N}$ définie par $n \mapsto k$ est Turing-calculable. On supposera que $\Sigma := \{0, 1\}$.

Exercice † 8 Démontrez que l'addition de $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ définie par $(n, m) \mapsto n + m$ est Turing-calculable. On supposera que $\Sigma := \{0, 1\}$ et que les entiers n et m sont séparés par une case vide.

Exercice 9 Démontrez que la fonction $f : \mathbb{N} \rightarrow \mathbb{Z}/4\mathbb{Z}$ définie par $n \mapsto n \pmod{4}$ est Turing-calculable. On supposera que $\Sigma := \{0, 1\}$.

Exercice † 10 Modifiez la définition 5 pour modéliser une machine de Turing qui écrit *ou* efface *ou* déplace la tête de lecture/écriture. Démontrez que les deux modèles sont équivalents en simulant une machine par l'autre.

5. COMPLEXITÉ ET MODÈLES ALGORITHMIQUES

Nous savons d'après la thèse de Church-Turing que le modèle algorithmique choisi n'a pas d'incidence en terme de calculabilité/décidabilité. Toute fonction calculable avec un modèle A est calculable avec un modèle B et réciproquement. La calculabilité d'une fonction est totalement indépendante des ressources nécessaires en espace et en temps pour le faire. *A contrario*, elles sont absolument centrales dans la théorie de la complexité.

La définition des ressources nécessaires à l'exécution d'un programme est simple, que ce soit pour la machine RAM ou la machine de Turing. Pour la mesure de l'espace, il s'agit du nombre de registres utilisés dans la mémoire de la machine RAM ou du nombre de cases de la bande d'entrée/sortie de la machine de Turing. Pour la mesure du temps, on compte le nombre d'instructions décodées par la machine RAM ou le nombre de transitions réalisées par la machine de Turing. Notons $\eta(x)$ le nombre de transitions réalisées par la machine de Turing pour traiter l'entrée x .

Définition 6. On appelle **fonction de complexité en temps** d'un algorithme de la machine de Turing, la fonction $T : \mathbb{N} \rightarrow \mathbb{N}$ définie par

$$T(n) := \max_{|x|=n} \eta(x). \quad (3)$$

où $|x|$ désigne la longueur du mot x .

Exercice 11 En vous inspirant de cette définition, donnez une définition de la fonction de complexité en espace S d'une machine de Turing. Reprenez ces deux définitions pour la machine RAM cette fois.

Les fonctions de complexité en temps T et en espace ont donc pour paramètre n , la *taille* des données à traiter, c'est-à-dire le nombre d'entiers utilisés sur la bande d'entrée dans le cas de la machine RAM et le nombre de cases utilisées pour ranger x sur la bande d'entrée. Ces deux fonctions mesurent donc le *pire des cas*.

Exercice 12 Calculez les complexités en temps et en espace des deux algorithmes de l'exercice 6. Que constatez-vous ?

À la lumière de cet exercice, il est évident que le modèle utilisé a une influence majeure sur les fonctions de complexité des algorithmes. Si l'on définissait l'efficacité d'un algorithme, en affirmant par exemple que sa fonction de complexité en temps doit être au plus quadratique, cette notion dépendrait alors du modèle algorithmique considéré et la théorie de la complexité serait elle aussi dépendante du modèle, ce qui n'est pas satisfaisant. Les informaticiens A. Aho et J. Ullman ont pu simuler une machine de Turing avec une machine RAM et réciproquement tout en majorant le coût de cette simulation :

Théorème 3 (Aho-Ullman). *Un algorithme de complexité $T(n)$ sur la machine de Turing (resp. sur la machine RAM), peut être simulé par un algorithme de complexité $O(T(n) \cdot \log(T(n)))$ sur la machine RAM (resp. $O(T(n))^3$) sur la machine de Turing.*

Comme on peut le constater, l'inflation peut être conséquente. La simulation d'une machine RAM par une machine de Turing peut augmenter le coût d'un algorithme de manière cubique, mais celui-ci reste malgré tout polynomial si sa fonction de complexité T était polynomiale dans son modèle originel. Cette étude justifie la définition suivante, même si en pratique

un algorithme dont la fonction de complexité est polynomiale de degré 12 est inexploitable.

Définition 7. Un algorithme est dit *efficace* si sa fonction de complexité est majorée par une fonction polynomiale.

On dit également d'une machine de Turing est polynomiale si sa fonction de complexité est majorée par une fonction polynomiale en la taille de l'entrée.

Définissons sommairement la notion de *problème*. Un problème est une question d'ordre général qui attend une réponse, il dépend de *paramètres* ou *données* variables dont les valeurs ne sont pas spécifiées. Le problème est décrit en fournissant :

- (1) Une description générale de ses *paramètres*.
- (2) Un énoncé des propriétés que la *solution* doit satisfaire.

Chaque jeu de valeurs particulières des paramètres définit une *instance* du problème.

Définition 8. Un problème est dit *faisable* s'il existe un algorithme efficace pour le résoudre, *infaisable* dans le cas contraire.

NB. Le terme *infaisable* est utilisé pour traduire le terme anglo-saxon *intractable* qui reste largement employé, même dans les ouvrages francophones.

On déduit de cette étude que la théorie de la complexité partitionne en première approche les algorithmes en deux classes : ceux dont la fonction de complexité est majorée par un polynôme et les autres. On se libère ainsi du modèle algorithmique choisi, en effet s'il existe un algorithme polynomial pour résoudre un problème dans un modèle algorithmique, il existe un algorithme polynomial dans l'autre modèle.

Nous connaissons de nombreux algorithmes de complexité polynomiale, pour trier des données, pour calculer une exponentielle, pour diviser des nombres, etc. On peut donc affirmer que ces problèmes sont faisables au sens de la définition précédente. Malheureusement, il existe également des problèmes pour lesquels on ne connaît aucun algorithme de résolution efficace à l'heure actuelle, ce qui ne signifie pas nécessairement qu'il n'en

existe pas. Il est rare que l'on puisse établir des résultats de non-existence d'algorithmes efficaces, les fonctions non-calculables mises à part, puisque dans ce cas il n'existe pas d'algorithme *du tout*. Nous avons par exemple démontré en licence que la complexité dans le pire des cas d'un algorithme de tri comparatif est en $\Omega(n \log n)$ si n désigne le nombre de valeurs à trier.

Dans ces conditions, comment hiérarchiser la difficulté des problèmes algorithmiques en l'absence d'algorithmes qui les résolvent efficacement ? Nous allons voir comment franchir cet obstacle. La théorie se limite ici aux *problèmes de décision*, c'est-à-dire aux problèmes dont le résultat attendu est la réponse *Oui* ou *Non*. Par exemple, le problème de la primalité qui consiste à savoir si un entier naturel N donné est premier, est un problème de décision.

Le lecteur objectera que les problèmes que l'on cherche à résoudre ne sont pas toujours des problèmes de décision. En effet, le problème du *voyageur de commerce*, qui consiste à déterminer le circuit le plus court pour livrer toutes les villes d'une tournée donnée est un problème d'*optimisation*. On peut cependant le transformer en problème de décision en fixant une borne K et en demandant s'il existe un circuit dont la longueur est inférieure à K . Il est clair que si l'on peut établir que la version décisionnelle est difficile (dans un sens que nous serons amenés à préciser), le problème d'optimisation associé le sera au moins autant. On peut montrer, mais nous n'aborderons pas ces questions dans ce cours, que dans la plupart des cas, la version décisionnelle d'un problème d'optimisation n'est pas plus facile à résoudre.

Les problèmes de décision seront présentés comme le problème du voyageur de commerce ci-dessous et nous conserverons la terminologie anglo-saxonne pour éviter les confusions avec certains acronymes en français.

Problème TSP [TRAVELING SALESMAN PROBLEM]

Instance : Un graphe non-orienté $G = (X, V)$ avec $X = \{x_1, \dots, x_n\}$ (villes), une pondération $w : V \rightarrow \mathbb{N}$ (distances) et une distance maximale $K \geq 0$.

Question : Existe-t-il un circuit hamiltonien de longueur inférieure à K ? Autrement dit un circuit $\pi \in S_n$ qui satisfait

$$\left(\sum_{i=1}^{n-1} w(x_{\sigma(i)}, x_{\sigma(i+1)}) \right) + w(x_{\sigma(n)}, x_{\sigma(1)}) \leq K. \quad (4)$$

Ce problème est clairement décidable. Un algorithme naïf consiste à générer toutes les permutations $\sigma \in \mathfrak{S}_{n-1}$ (la première ville est fixée puisque les décalages circulaires d'un circuit hamiltonien définissent n circuits équivalents) et à calculer pour chacune d'entre-elles la **somme** des distances puis à la comparer à la borne K (inégalité (4)). Malheureusement il y a $(n-1)!$ permutations et nous savons que $\exp(n) = o(n!)$, autrement dit que la fonction factorielle croît strictement plus rapidement que la fonction exponentielle. Le problème est-il pour autant infaisable ? Nous verrons que la réponse à cette question particulière semble être affirmative.

Exercice 13 En supposant que l'on dispose d'une machine capable de réaliser 5 milliards d'additions sur des entiers de taille arbitraire, quelle est le nombre maximal de villes que l'algorithme naïf est capable de traiter en 24h ? On supposera que le coût pour générer une permutation ou pour comparer deux entiers est le même que celui d'une addition.

Exercice 14 Démontrez qu'un algorithme de tri comparatif ne peut avoir une complexité en temps meilleure que $\Omega(n \log n)$. Indication : considérez l'arbre binaire de décision associé à chaque comparaison réalisée par l'algorithme et montrez que pour trier n valeurs cet arbre doit contenir au moins $n!$ feuilles. Calculez la hauteur minimale d'un tel arbre binaire et concluez.

6. LA CLASSE P

Dans la suite, nous noterons Π un problème de décision, I une instance de Π et e un schéma d'encodage qui transforme l'instance I du problème en un mot $x \in \Sigma^*$ où Σ est un alphabet fini. Nous noterons également Y_Π le sous-ensemble des *instances positives*, i.e. celles pour lesquelles la réponse à la question est *Oui* et N_Π le sous-ensemble des *instances négatives*, i.e. celles pour lesquelles la réponse à la question est *Non*.

Nous allons utiliser la machine de Turing à la manière d'un automate pour reconnaître un langage associé à un problème de décision. Pour cela, l'ensemble des états contient deux états particuliers, *l'état d'acceptation* q_Y et *l'état de refus* q_N dans lesquels la machine doit s'arrêter pour indiquer respectivement si un mot a été reconnu ou non.

Remarque. Si l'on restreint la fonction de transition d'une machine de Turing à des transitions de la forme $\delta(p, \alpha) = (\alpha, d, q)$, autrement dit qui ne

modifient pas le symbole courant, alors on retrouve le modèle des *automates finis déterministes*.

Une instance I d'un problème de décision Π est donc encodée par le schéma d'encodage e en un mot $x = x_1x_2\dots x_n$ de Σ^* dont les symboles sont rangés sur la bande d'entrée (cf. figure 3). On associe alors au problème Π le langage

$$L[\Pi, e] := \{e(I) \mid I \in Y_\Pi\}. \quad (5)$$

Définition 9. Un langage est dit *polynomial* s'il est reconnu par une machine de Turing déterministe polynomiale, c'est-à-dire en temps majoré par $p(n)$ où p est un polynôme et n la longueur du mot $x \in \Sigma^*$ en entrée. La **classe P** est l'ensemble des langages polynomiaux.

Définition 10. Un problème de décision Π est dit *polynomial* ou dans la classe P si et seulement si $L[\Pi, e] \in P$ pour un schéma d'encodage e .

Exemple 5. Considérons le problème de décision suivant :

Problème DIV4 [DIVISIBILITÉ PAR 4]

Instance : Un entier naturel N .

Question : Le nombre N est-il un multiple de 4 ?

Considérons l'alphabet $\Sigma = \{0, 1\}$ et le schéma d'encodage e qui consiste à coder un entier naturel N par son écriture binaire avec le chiffre le *moins* significatif à gauche, par exemple

$$e(35) := \underline{110001}.$$

Les instances positives, i.e. les multiples de 4 sont encodés par un mot binaire qui commence par 00 si $N > 0$ ou le mot 0 si $N = 0$. Le langage associé à ce problème s'exprime sous la forme d'une expression régulière (cf. cours de théorie des langages) :

$$L[\text{DIV4}, e] = \{0 + 00(0 + 1)^*1\}. \quad (6)$$

Nous allons montrer que $\text{DIV4} \in P$ en exhibant un programme qui reconnaît le langage (6) en temps polynomial sur la taille du mot binaire à traiter.

Puisque N est un multiple de 4 si sa représentation binaire est exactement le mot 0 ou un mot qui commence par deux 0 et se termine par un 1, il suffit de consulter les cases d'indice 1, 2 et n de la bande d'entrée/sortie pour s'en assurer. On suppose par convention que la bande contient un mot binaire $x = x_1x_2 \dots x_n$ dont les n bits x_i sont placés dans les cases d'indice i respectivement et que la bande est vierge ailleurs (ne pas perdre de vue que nous avons supposé que les bits de poids faibles étaient à gauche). Autrement dit, sur l'alphabet $\{0, 1, \square\}$, la bande contient le mot

$$\square^\infty x_1 x_2 \dots x_n \square^\infty.$$

Le programme ci-dessous reconnaît le langage $L[\text{DIV4}, e]$:

- $q_0, 0 : 0, \rightarrow, q_1$ // Si $B[1] = 0$, alors on va à droite.
- $q_0, 1 : 1, \rightarrow, q_N$ // Si $B[1] = 1$, alors N est impair, rejet.
- $q_1, \square : \square, \rightarrow, q_Y$ // Si $B[2] = \square$, alors $N = 0$, acceptation.
- $q_1, 0 : 0, \rightarrow, q_2$ // Si $B[2] = 0$, alors il faut vérifier la case $\#n$.
- $q_1, 1 : 1, \rightarrow, q_N$ // Si $B[2] = 1$, alors rejet.
- $q_2, 0 : 0, \rightarrow, q_2$ // Tant que $B[i] \neq \square$
- $q_2, 1 : 1, \rightarrow, q_2$ // $i \leftarrow i + 1$
- $q_2, \square : \square, \leftarrow, q_3$ // Si $B[n + 1] = \square$, alors on va à gauche.
- $q_3, 1 : 1, \rightarrow, q_Y$ // Si $B[n] = 1$, alors acceptation.
- $q_3, 0 : 0, \rightarrow, q_N$ // Si $B[n] = 0$, alors rejet.

Notons que dans la deuxième instruction, on rejete aussi les mots de Σ^* qui commencent par 1 et qui ne sont pas des encodages valides, (rejetés également par la dernière instruction.)

Dans le pire des cas, il faut consulter le contenu de la case $\#n$ pour s'assurer qu'elle contient le chiffre 1 pour pouvoir décider qu'un entier N est un multiple de 4 et qu'on n'a pas affaire à un mot qui n'est pas un encodage conforme d'un entier. Ceci nécessite $O(n)$ transitions et prouve que la reconnaissance des entiers multiples de 4 est polynomial en le nombre n de chiffres binaires de sa représentation.

Nous venons de prouver le théorème suivant :

Théorème 4. *Le problème DIV4 appartient à la classe P.*

Un problème de décision est donc dans la classe P si l'on est capable de *prouver* le résultat en temps polynomial. Comme nous l'avons déjà évoqué, ne pas trouver d'algorithme polynomial pour répondre à une question ne signifie pas qu'il n'en existe pas. La piste que l'on va suivre à présent pour franchir cette barrière se base sur l'idée suivante : peut-on répondre à la question en temps polynomial si l'on se contente de *vérifier* une preuve et que l'on n'a pas à la trouver ?

Par exemple, dans le cadre du problème du voyageur de commerce, si l'on dispose d'une permutation σ qui réalise un circuit parcourant toutes les villes, il suffit de sommer les n distances de ce circuit et de comparer cette somme à la borne K pour conclure. Ceci peut se calculer en temps polynomial, l'algorithme de l'addition est linéaire en le nombre de chiffres des deux opérands et la comparaison entre deux nombres a un coût qui est lui aussi linéaire en le nombre de chiffres du plus petit des deux.

7. LA CLASSE NP

La terminologie employée pour décrire cette nouvelle classe n'est pas très heureuse, NP résumant Non-déterministe et Polynomial et pas Non Polynomial comme on pourrait le penser, mais elle est historique. Là encore, il s'agit de collecter des langages/problèmes polynomiaux, mais cette fois les mots x ne sont pas reconnus directement, mais à l'aide d'une preuve/certificat qu'il faut vérifier en temps polynomial. Pour cela, il nous faut introduire un nouveau modèle de machine de Turing, le modèle *non-déterministe*.

Une machine de Turing non-déterministe se distingue uniquement d'une machine déterministe *avant* l'exécution du programme. Après avoir inscrit un mot $x = x_1x_2 \dots x_n$ sur la partie positive de la bande à partir de la case d'indice 1 et juste avant l'exécution du programme, on suppose qu'un *oracle* inscrit un mot $y_1y_2 \dots y_m$ appelé *certificat* ou *preuve* sur la partie négative de la bande d'entrée/sortie avec y_i en case d'indice $-i$. Une case vierge en position 0 sépare donc les deux mots (cf. figure 4).

Le programme consiste alors à *vérifier* une preuve/certificat $y \in \Sigma^*$ fourni par l'oracle. Plus précisément, si la preuve que le mot x appartient au langage $L[\Pi, e]$ — ou similairement que I une instance positive du problème de décision — est valide alors la machine s'arrête dans l'état q_Y et reconnaît le mot x .

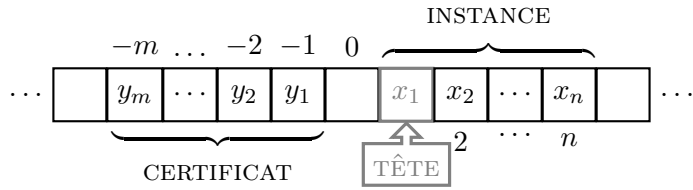


FIGURE 4. Machine de Turing non-déterministe.

Si l'on considère le couple (x, y) constitué de l'instance x et d'un certificat y comme une donnée, cette machine est parfaitement déterministe. Historiquement, le certificat était créé *ex-nihilo* par l'oracle, l'idée sous-jacente était que cette machine modélisait le fonctionnement (supposé) non-déterministe de l'être humain, à la manière d'un joueur d'échecs qui trouve le "meilleur" coup sans pour autant avoir évalué toutes les branches de l'arbre de décision. L'oracle fournit la preuve que la réponse à la question est positive et la machine n'a plus qu'à la vérifier. Bien entendu, et contrairement au modèle déterministe, le modèle ainsi présenté n'a pas vocation à être utile *en pratique* car aucun oracle ne viendra nous souffler une preuve.

Définition 11. Un langage est dit *non-déterministe polynomial* s'il est reconnu par une machine de Turing non-déterministe polynomiale, c'est-à-dire en temps majoré par $p(n)$ où p est un polynôme et n la longueur du mot $x \in \Sigma^*$ en entrée. La **classe NP** est l'ensemble des langages non-déterministes polynomiaux.

Définition 12. On dit qu'un problème Π est *non-déterministe polynomial* ou dans la classe NP si et seulement si $L[\Pi, e] \in NP$ où e désigne un schéma d'encodage raisonnable³.

Exemple 6. Un entier N est dit *factorisable* si on peut l'écrire comme un produit $A.B$ non-trivial, i.e. $A \neq 1$ et $B \neq 1$. Considérons le problème de décision suivant :

Problème FACTORISABLE [ENTIER FACTORISABLE]
Instance : Un entier naturel N .
Question : Peut-on factoriser N ?

Théorème 5. Le problème FACTORISABLE appartient à la classe NP.

Démonstration. Soit $y := e(A, B)$ l'encodage d'un couple (A, B) , par exemple les écritures binaires de A et de B séparées par un \square , codant le certificat. L'algorithme de la multiplication de deux entiers étudié à l'école primaire permet de calculer le produit AB en un temps $O(\log A \log B)$, en effet le nombre de chiffres d'un entier dans sa représentation est proportionnel à son logarithme, et il ne reste qu'à comparer les chiffres de ce produit avec ceux de $x = x_1x_2 \dots x_n$, ce qui se fait en $O(\min(\log A + \log B, \log N))$. \square

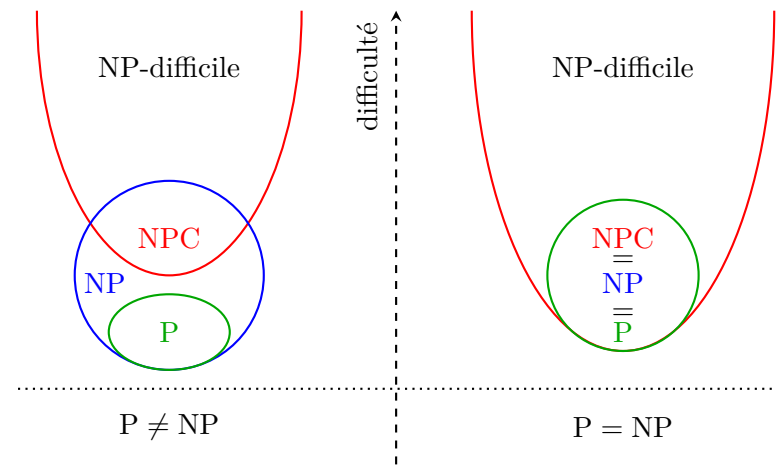


FIGURE 5. Classes de complexité.

Théorème 6. $P \subseteq NP$.

Démonstration. Si $L \in P$, par définition on dispose d'un programme sur une machine de Turing déterministe qui reconnaît les mots x de L en temps polynomial. Il suffit d'utiliser le même programme sur une machine non-déterministe. En d'autres termes, on n'a pas besoin de vérifier la preuve y fournie par l'oracle pour accepter x puisque l'on est capable de prouver le résultat grâce à l'algorithme déterministe. \square

3. au sens où la taille du mot codant l'instance n'est manifestement pas démesurée.

La conjecture suivante (à un million de dollars) exprime formellement dans cette théorie qu'il est certainement plus difficile de trouver une preuve que d'en vérifier une. Elle est illustrée en figure 5.

Conjecture 1. $P \neq NP$.

Définition 13. Soit Σ un alphabet fini. On dit qu'une machine de Turing calcule le mot $y_1y_2 \dots y_m$ pour le mot $x_1x_2 \dots x_n$ si la bande contient initialement $\square^\infty \overline{x_1} x_2 \dots x_m \square^\infty$ avec la tête de lecture/écriture placée sur la case d'indice 1 et la machine s'arrête sur la bande contenant le mot $\square^\infty y_1y_2 \dots y_m \square^\infty$.

Définition 14. Soit Σ un alphabet fini. Une fonction $f : \Sigma^* \rightarrow \Gamma^*$ est dite Turing-calculable s'il existe une machine de Turing T qui calcule $y = f(x)$ pour tout $x \in \mathcal{D}_f$.

Exercice 15 Soit f, g deux fonctions de Σ^* dans Σ^* . Démontrez que si f et g sont Turing-Calculables, alors $g \circ f$ est Turing-Calculable.

On note \mathcal{T} l'ensemble des machines de Turing et \mathcal{T}_P l'ensemble des machines de Turing polynomiales, i.e. qui s'arrêtent en temps $O(p(n))$ où $p(n)$ est un polynôme en la longueur n du mot x contenu initialement sur la bande d'entrée/sortie.

8. LA CLASSE NP-COMPLET

Pour mesurer la difficulté d'un problème de décision, on définit une relation binaire \propto sur l'ensemble \mathcal{L} des langages qui permet de *comparer* la difficulté de deux problèmes Π_1 et Π_2 à travers leurs langages $L[\Pi_1, e_1]$ et $L[\Pi_2, e_2]$.

Définition 15. On appelle **transformation** d'un langage L_1 en un langage L_2 d'alphabets respectifs Σ_1 et Σ_2 toute fonction $\tau : \Sigma_1^* \rightarrow \Sigma_2^*$ Turing-calculable telle que

$$x \in L_1 \Leftrightarrow \tau(x) \in L_2. \quad (7)$$

On définit alors une relation binaire \propto sur les langages par $L_1 \propto L_2$ si et seulement s'il existe une transformation de L_1 en L_2 . On dit qu'il s'agit d'une **transformation polynomiale** et l'on note alors $L_1 \propto_P L_2$, si la complexité de la transformation est polynomiale.

Si Π_1 et Π_2 sont deux problèmes de décision, on note $\Pi_1 \propto \Pi_2$ si et seulement si $L[\Pi_1, e_1] \propto L[\Pi_2, e_2]$ pour des schémas d'encodages e_1 et e_2 . Pour comprendre que $\Pi_1 \propto \Pi_2$ formalise l'idée que le problème Π_2 est *au moins aussi difficile* que le problème Π_1 , supposons que l'on dispose d'un programme qui reconnaît les mots du langage L_2 , autrement dit qui résout le problème Π_2 . On dispose alors d'un programme qui résout le problème Π_1 puisqu'il suffit de traduire les mots de L_1 en mots de L_2 au préalable (donc indirectement les instances positives du problème Π_1 en instances positives de Π_2).

Théorème 7. Soit Π_1 et Π_2 deux problèmes de décision. Si $\Pi_1 \propto_P \Pi_2$ et $\Pi_2 \in P$ alors $\Pi_1 \in P$.

Démonstration. Soit T_τ la machine de Turing qui réalise la transformation polynomiale de $L[\Pi_1, e_1]$ en $L[\Pi_2, e_2]$ et T_2 la machine de Turing polynomiale qui reconnaît les mots du langage $L[\Pi_2, e_2]$, alors la composition $T_1 := T_2 \circ T_\tau$ des machines de Turing T_2 et T_τ reconnaît les mots de $L[\Pi_1, e_1]$ en temps polynomial. \square

Proposition 1. La relation binaire \propto_P définie sur les langages est réflexive et transitive mais pas antisymétrique.

Démonstration. Pour la réflexivité et la transitivité, la preuve est laissée en exercice. Pour l'antisymétrie, il suffit de considérer les deux problèmes de décision Π_1 et Π_2 suivants : dans les deux cas l'instance est un entier naturel N , et la question pour Π_1 (resp. Π_2) est N est-il pair (resp. impair) ? Une traduction polynomiale possible de Π_1 en Π_2 est la fonction définie par $N \mapsto N + 1$ qui est Turing-Calculable en temps polynomial. La même fonction permet de transformer polynomialement Π_2 en Π_1 , on a donc $\Pi_1 \propto_P \Pi_2$ et $\Pi_2 \propto_P \Pi_1$ et pourtant $\Pi_1 \neq \Pi_2$. \square

Définition 16. Deux problèmes de décision Π_1 et Π_2 sont dits **polynomialement équivalents** si et seulement si

$$(\Pi_1 \propto_P \Pi_2) \wedge (\Pi_2 \propto_P \Pi_1) \quad (8)$$

ce qui définit une relation binaire \propto_P sur les problèmes de décision.

Proposition 2. *La relation binaire α_P est une relation d'équivalence sur les problèmes de décision/langages.*

Exercice 16 Démontrez la proposition 2.

Si \mathcal{D} désigne l'ensemble des problèmes de décision, les différents représentants d'une classe d'équivalence de l'ensemble quotient \mathcal{D}/α_P sont des problèmes de même difficulté. On peut alors équiper l'ensemble quotient \mathcal{D}/α_P de la relation α_P induite qui est cette fois une relation d'ordre (exercice).

Cette relation d'ordre est-elle totale, partielle ? Se pose la question de l'existence d'éléments maximaux, i.e. parmi les problèmes que l'on peut comparer, quels sont ceux qui sont les plus difficiles ? Existe-t-il un plus grand élément, c'est-à-dire un problème plus difficile que tous les autres ?

Définition 17. *Un langage L est dit **NP-difficile** si et seulement si*

$$\forall L' \in NP \quad L' \alpha_P L. \quad (9)$$

Un problème de décision Π est dit NP-difficile si $L[\Pi, e]$ est NP-difficile pour un schéma d'encodage e .

Définition 18. *Un langage L est dit **NP-complet** (en abrégé NPC) si et seulement si*

- (1) $L \in NP$,
- (2) L est NP-difficile.

Un problème Π est dit NP-complet si $L[\Pi, e]$ est NP-complet pour un schéma d'encodage e .

Remarque importante. Aucune de ces deux définitions n'est encore légitimée. Elles supposent qu'il existe un plus grand élément dans le quotient NP/α_P , or il n'existe pas toujours de plus grand élément dans un ensemble ordonné (l'ensemble des entiers naturels n'admet pas de plus grand élément pour l'ordre naturel \leq par exemple). C'est le théorème de Cook que nous étudierons après le chapitre suivant qui va en donner la justification.

Définition 19. *Soit Π un problème de décision. On appelle **problème dual** le problème noté $\bar{\Pi}$ dont les instances sont celles de Π et dont la question est la négation de celle du problème Π .*

Exemple 7. Le problème de la primalité dont l'instance est un entier naturel N et la question est *N est-il premier ?* est le problème dual du problème FACTORISABLE.

Exercice 17 Démontrez que si $\Pi \in P$, alors $\bar{\Pi} \in P$.

Exercice 18 Quel est le problème dual \overline{TSP} du problème du voyageur de commerce ? Quels arguments (informels) militent pour affirmer que le problème \overline{TSP} n'appartient pas à NP ?

Définition 20. *La classe **co-NP** est la classe des problèmes de décision Π tels que $\bar{\Pi} \in NP$.*

9. LE THÉORÈME DE COOK

On trouvera en annexe (cf. section 12), quelques rappels sur la logique propositionnelle. Considérons le problème de décision suivant :

Problème SAT [SATISFAISABILITÉ DES CLAUSES]

Instance : Un ensemble de variables booléennes U et un ensemble de clauses C définies sur U .

Question : L'ensemble des clauses C est-il satisfaisable ? Autrement dit, existe-t-il une interprétation $I : U \rightarrow \{\mathcal{V}, \mathcal{F}\}$ des variables de U qui satisfait

$$\forall c \in C \quad I(c) = \mathcal{V}.$$

Exemple 8. On considère l'ensemble des variables $U = \{u_1, u_2, u_3\}$ et l'ensemble des clauses $C = \{\{\bar{u}_1, u_2, u_3\}, \{u_1, \bar{u}_3\}\}$. Il y a trois variables et un ensemble de deux clauses à trois et deux littéraux respectivement. L'interprétation $u_1 \leftarrow u_2 \leftarrow \mathcal{V}$ et $u_3 \leftarrow \mathcal{V}$ (entre autres) montre que la formule

$$(\bar{u}_1 \vee u_2 \vee u_3) \wedge (u_1 \vee \bar{u}_3)$$

est satisfaisable.

La satisfaisabilité d'une instance I du problème SAT à n variables est équivalent à l'existence d'un n -uplet binaire x en entrée d'une fonction booléenne à n variables tel que $f(x) = 1$. Il suffit donc de tester les 2^n entrées binaires possibles pour s'en assurer, mais cet algorithme est de complexité exponentielle.

Théorème 8 (Cook). *Le problème SAT est NP-complet.*

Parmi les problèmes de la classe NP, il en existe donc au moins un qui est plus difficile que tous les autres. La suite de la section est consacrée à la démonstration de ce théorème.

Il faut montrer que le langage $L[\text{SAT}, e]$ appartient à la classe des langages NP puis que tous les langages $L \in \text{NP}$ se transforment polynomialement en $L[\text{SAT}, e]$. La première partie est simple, il suffit de considérer comme certificat une interprétation des variables propositionnelles. On sait évaluer la valeur d'une formule propositionnelle pour une interprétation donnée des variables. En exprimant le problème SAT dans le langage de la logique booléenne, il s'agit simplement d'évaluer une fonction booléenne à n variables et on connaît un algorithme linéaire en le nombre de littéraux de la formule (cf. évaluation d'une expression arithmétique ou logique en algorithmique). Si la formule est vraie pour le certificat donné, la formule est donc satisfaisable.

La seconde partie de la preuve est longue et un peu technique, mais n'est pas très difficile. Nous allons procéder en plusieurs étapes. Il nous faut montrer que

$$\forall \Pi \in \text{NP} \quad \Pi \leq_P \text{SAT}.$$

Il faut donc exhiber une transformation polynomiale pour chaque problème Π de la classe NP sans pour autant faire une preuve pour chacun.

On se donne un problème $\Pi \in \text{NP}$ et nous allons construire une transformation τ des encodages $x := e(I)$ des instances de ce problème en instances du problème SAT directement plutôt qu'en mots qui sont des encodages d'instances de SAT. Ceci ne limite pas la portée de la preuve, cela nous évite simplement de considérer un schéma d'encodage pour représenter les instances du problème SAT.

Nous allons construire cette transformation $\tau : \Sigma^* \rightarrow \mathcal{C}$ de l'ensemble des mots sur l'alphabet Σ qui codent les instances du problème Π vers l'ensemble \mathcal{C} des formules propositionnelles qui sont des conjonctions de clauses telle que

$$x \in L[\Pi, e] \Leftrightarrow \tau(x) \text{ est satisfaisable.} \quad (10)$$

Nous concluerons en prouvant que cette transformation est polynomiale.

La seule information dont nous disposons sur le langage $L[\Pi, e]$ est qu'il est dans NP, il existe donc une machine de Turing non-déterministe polynomiale T qui le reconnaît. L'idée de la preuve est de modéliser le fonctionnement de cette machine en logique propositionnelle, de la même manière que l'on peut modéliser l'énigme d'Einstein, le problème des 8 reines ou le problème du Sudoku que nous allons aborder en guise d'illustration.

Exemple 9. Une grille de Sudoku est une matrice 9×9 d'entiers de l'intervalle $\llbracket 1, 9 \rrbracket$ découpée en 9 régions de taille 3×3 . Un *bloc* désigne une ligne, une colonne ou une région de cette matrice, il y en a donc $3 \times 9 = 27$ et chacun contient 9 valeurs. Une grille de Sudoku est *valide* si et seulement si elle satisfait les trois conditions suivantes :

- (1) chaque ligne contient exactement chacune des 9 valeurs.
- (2) chaque colonne contient exactement chacune des 9 valeurs.
- (3) chaque région contient exactement chacune des 9 valeurs.

On pouvait exprimer la validité plus simplement en disant que tout bloc est une permutation de \mathfrak{S}_9 . La grille à droite dans la figure 6 est une grille valide. Le jeu consiste à ne fournir qu'une *grille indice* qui ne contient que certaines valeurs de la grille (par exemple la grille à gauche dans la figure 6), charge au joueur de trouver celles qui manquent. Notons que dans un cadre plus général, il peut exister plusieurs solutions à une grille indice donnée ou éventuellement aucune (grille invalide).

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

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

FIGURE 6. Une grille de Sudoku valide et sa solution.

Le problème du Sudoku se généralise en dimension n à une grille $n^2 \times n^2$ dont les valeurs appartiennent à l'intervalle $\llbracket 1, n^2 \rrbracket$. La version décisionnelle du problème du Sudoku est la suivante :

Problème SUDOKU [Jeu du SUDOKU]

Instance : Une grille de Sudoku, i.e. une matrice S carrée $n^2 \times n^2$ de valeurs dans $\llbracket 0, n^2 \rrbracket$ (la valeur 0 code une case vide).

Question : Existe-t-il une solution à cette grille, i.e. un triplet de n^2 permutations de \mathfrak{S}_{n^2} qui coïncident avec celles de la grille indice ?

Exercice 19 Démontrez que le problème SUDOKU est dans la classe NP.

On se propose à présent de transformer une instance du Sudoku classique ($n = 3$) en instance du problème SAT. On définit 729 variables propositionnelles $S[l, c, k]$ avec $(l, c, k) \in \llbracket 1, 9 \rrbracket^3$ que l'on interprète " $S[l, c, k]$ est vraie si et seulement si la case à la ligne l et à la colonne c contient la valeur k ". Ainsi la clause

$$\{S[1, 1, 1], S[1, 1, 2], \dots, S[1, 1, 9]\} \quad (11)$$

ne peut être satisfaite que si la case de coordonnées $(1, 1)$ dans l'angle supérieur gauche contient (au moins) une valeur entre 1 et 9. Pour exprimer que cette case ne peut pas contenir deux valeurs distinctes, on construit pour chaque couple $(i, j) \in \llbracket 1, 9 \rrbracket^2$, $i < j$ (il y en a 36 au total) la clause

$$\{\overline{S[1, 1, i]}, \overline{S[1, 1, j]}\} \quad (12)$$

qui ne peut être satisfaite que si la case $(1, 1)$ ne contient qu'une seule valeur en remarquant que $\overline{S[1, 1, i]} \vee \overline{S[1, 1, j]} \equiv \overline{S[1, 1, i] \wedge S[1, 1, j]}$ qui exprime que l'on ne veut pas que la case $(1, 1)$ contiennent la valeur i et la valeur j . Il faut bien sûr définir le même jeu de clauses pour chacune des 80 cases restantes. La grille indice est modélisée par les clauses

$$\{S[l, c, v]\}$$

pour toutes les cases (l, c) pour lesquelles la valeur v est connue.

Exercice † 20 Avec les notations introduites pour la formalisation du problème du Sudoku pour $n = 3$, écrivez les clauses qui modélisent la règle du jeu, à savoir que chacun des 27 blocs contient une permutation de \mathfrak{S}_9 .

C'est exactement la même démarche qui va être utilisée pour modéliser la reconnaissance d'un mot $x \in L[\Pi, e]$ par la machine polynomiale non-déterministe T .

Notons $\Sigma = \{s_0, s_1, \dots, s_\nu\}$ l'alphabet fini de cardinal $\nu + 1$ de la machine T avec pour convention que le premier symbole $s_0 = \square$ est le symbole blanc. Nous notons $Q = \{q_0, q_1, \dots, q_r\}$ les $r + 1$ états de cette machine avec pour convention que q_0 est l'état initial, $q_1 = q_Y$ est l'état d'acceptation et $q_2 = q_N$ est l'état de rejet. La définition suivante nous sera utile plus loin dans la preuve.

Définition 21. Soit $t \in \mathbb{N}$. On appelle **configuration** d'une machine de Turing T à l'instant t le triplet $\chi_t := (q, i, B) \in Q \times \mathbb{Z} \times \Sigma_{\square}^{\mathbb{Z}}$ dont les trois projections sont : l'état q de la machine, la position i de la tête de lecture/écriture et le contenu B de la bande après t transitions.

Exemple 10. La configuration initiale (i.e. à l'instant $t = 0$) de la machine non-déterministe polynomiale qui reconnaît $L[\Pi, e]$ est

$$\chi_0 = (q_0, 1, \square^\infty y_m \dots y_2 y_1 \square x_1 x_2 \dots x_n \square^\infty)$$

où le contenu de la bande est codé par un mot de Σ_{\square}^* où $x = x_1 x_2 \dots x_n$ est l'encodage d'une instance du problème Π et $y = y_1 y_2 \dots y_m$ le certificat fourni par l'oracle.

La machine T étant polynomiale, il existe un polynôme $p(n)$ qui borne le nombre de transitions effectuées avant que la machine s'arrête sur l'état q_Y si $x \in L[\Pi, e]$. La zone utile de la bande d'entrée/sortie est donc bornée puisque chaque il y a un déplacement par transition. La zone utile est circonscrite dans l'intervalle $\llbracket -p(n), p(n) + 1 \rrbracket$ de \mathbb{Z} (cf. figure 7).

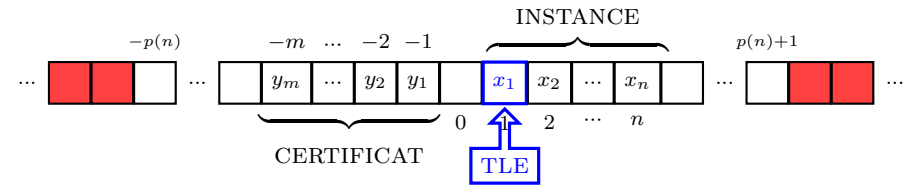


FIGURE 7. Zone utile de la bande (cases blanches).

On peut à présent définir l'ensemble U des variables propositionnelles de l'instance de SAT que nous allons construire. Pour simplifier les écritures, on définit les intervalles

$$\begin{aligned} I_T &:= \llbracket 0, p(n) \rrbracket \text{ (instants)} & I_Q &:= \llbracket 0, r \rrbracket \text{ (états)} \\ I_P &:= \llbracket -p(n), p(n) + 1 \rrbracket \text{ (positions)} & I_S &:= \llbracket 0, \nu \rrbracket \text{ (symboles)} \end{aligned}$$

Les variables sont partitionnées en trois groupes avec les interprétations suivantes :

- (1) $\{Q[t, i] \mid (t, i) \in I_T \times I_Q\}$. À l'instant t , la machine est dans l'état q_i .
- (2) $\{P[t, i] \mid (t, i) \in I_T \times I_P\}$. À l'instant t , la tête est en position i .
- (3) $\{S[t, i, j] \mid (t, i, j) \in I_T \times I_P \times I_S\}$. À l'instant t , la case i contient s_j .

Nous supposons que si la machine s'arrête à un instant $t < p(n)$, toutes les variables aux instants $t' > t$ conservent les valeurs logiques qu'elles avaient à l'arrêt de la machine.

L'exécution de la machine T crée *de facto* une interprétation I de l'ensemble U des variables propositionnelles. Si on voyait toutes ces variables comme autant de diodes allumées (\mathcal{V}) ou éteintes (\mathcal{F}), elles ne seraient qu'une autre façon de décrire le fonctionnement de T , à la manière d'un couple de diodes qui indiquent si une porte est verrouillée ou non et si l'alarme dans la pièce derrière cette porte est active ou non.

Comme pour le Sudoku, la satisfaisabilité des clauses que nous allons construire assurera que ce modèle descriptif calque d'une part le fonctionnement d'une machine de Turing de manière cohérente, et d'autre part qu'il décrit spécifiquement l'exécution de la machine T qui reconnaît le langage $L[\Pi, e]$.

Nous allons répartir les clauses de C en 6 groupes, avec un rôle spécifique pour chacun d'entre eux. Les trois premiers assurent la cohérence du modèle par rapport au fonctionnement d'une machine de Turing quelconque, les trois autres celui de la machine qui reconnaît le langage $L[\Pi, e]$.

- (1) G_1 : à chaque instant, la machine est dans un état et un seul.
- (2) G_2 : à chaque instant, la tête de lecture/écriture est sur une unique case de la bande.
- (3) G_3 : à chaque instant, chaque case de la zone utile de la bande contient un unique symbole.

- (4) G_4 : à l'instant initial, la machine est dans l'état q_0 et la bande contient les symboles du mot x à partir de la case 1 sur laquelle est placée la tête de lecture/écriture.
- (5) G_5 : la machine s'arrête dans l'état q_Y avant l'instant $p(n)$ et a reconnu le mot x .
- (6) G_6 : entre deux instants t et $t + 1$, les configurations χ_t et χ_{t+1} de la machine T sont conformes à sa fonction de transition δ .

Clauses du groupe G_1 . Pour chaque instant $t \in I_T$, on définit la clause

$$\{Q[t, 0], Q[t, 1], \dots, Q[t, r]\}$$

qui n'est satisfaite que si la machine est au moins dans un état q_k , $k \in I_Q$ à l'instant t . Pour chaque instant $t \in I_T$ et chaque couple $(i, j) \in I_Q^2$ tel que $i < j$, on définit la clause

$$\{\overline{Q[t, i]}, \overline{Q[t, j]}\} \equiv \overline{(Q[t, i] \wedge Q[t, j])}$$

qui n'est satisfaite que si la machine n'est pas dans deux états différents q_i et q_j au même instant t .

Clauses du groupe G_2 . Pour chaque instant $t \in I_T$, on définit la clause

$$\{P[t, -p(n)], \dots, P[t, p(n)], P[t, p(n) + 1]\}$$

qui n'est satisfaite que si la tête de lecture/écriture est au moins sur une case i à l'instant t . Pour chaque instant $t \in I_T$ et pour chaque couple $(i, j) \in I_Q^2$ tel que $i < j$, on définit la clause

$$\{\overline{P[t, i]}, \overline{P[t, j]}\} \equiv \overline{(P[t, i] \wedge P[t, j])}$$

qui n'est satisfaite que si la tête de lecture/écriture n'est pas sur deux cases différentes i et j au même instant t .

Clauses du groupe G_3 . Pour chaque instant $t \in I_T$ et chaque case $i \in I_P$ on définit la clause

$$\{S[t, i, 0], \dots, S[t, i, 1], S[t, i, \nu]\}$$

qui n'est satisfaite que si chaque case i contient au moins un symbole $s_j \in \Sigma$. Pour chaque instant $t \in I_T$, chaque case $i \in I_P$ et tout couple $(i, j) \in I_S^2$ tel que $i < j$, on définit la clause

$$\{\overline{S[t, i, j]}, \overline{S[t, i, j']}\} \equiv \overline{(S[t, i, j] \wedge S[t, i, j'])}$$

qui n'est satisfaite que si la case i ne contient pas simultanément les deux symboles s_j et $s_{j'}$ au même instant t .

Clauses du groupe G_4 . Initialement l'état de la machine est q_0 , la tête de lecture est sur la case 1 et la bande contient le mot $x = s_{k_1}s_{k_2}\dots s_{k_n}$ dans la partie positive et est vierge ensuite :

$$\begin{aligned} &\{Q[0, 0]\}, \{P[0, 1]\}, \{S[0, 1, k_1]\}, \\ &\{S[0, 2, k_2]\}, \dots, \{S[0, n, k_n]\}, \\ &\{S[0, n+1, 0]\}, \dots, \{S[0, p(n)+1, 0]\}. \end{aligned}$$

Clauses du groupe G_5 . La machine reconnaît x en temps $p(n)$, i.e. s'arrête dans l'état $q_Y = q_1$:

$$\{Q[p(n), 1]\}.$$

Clauses du groupe G_6 . Un premier sous-groupe de clauses assure qu'à tout instant t , si la tête de lecture/écriture *n'est pas en position i* alors le symbole s_j contenu dans cette case *n'a pas changé* à l'instant $t+1$:

$$\{P[t, i], \overline{S[t, i, j]}, S[t+1, i, j]\} \equiv \overline{(P[t, i] \wedge S[t, i, j] \wedge S[t+1, i, j])}$$

Ceci pour tout $(t, i, j) \in I_T \times I_P \times I_S$. Le second sous-groupe de clauses assure, qu'entre l'instant t et $t+1$:

- (1) La machine a correctement changé d'état.
- (2) La tête de lecture/écriture est à la bonne position.
- (3) Le changement de valeur dans la case courante est conforme.

Pour décrire le changement de configuration de la machine entre les instants t et $t+1$, nous supposons que si q_k est l'état de la machine et qu'elle n'a pas encore accepté ou rejeté x , i.e. $q_k \neq q_Y$ et $q_k \neq q_N$, alors les entiers k' , j' et $\Delta \in \{-1, 0, 1\}$ satisfont

$$\delta(q_k, s_j) = (q_{k'}, s_{j'}, \Delta).$$

En revanche si le mot a été accepté ou rejeté, i.e. $q_k \in \{q_Y, q_N\}$, dans ce cas $\Delta = 0$, $i = i'$ et $k = k'$ (la configuration ne change plus).

Exprimons les clauses de ce second groupe. Nous ne détaillons que la construction de la clause qui assure (1), i.e. que la machine a correctement changé d'état, les deux autres étant similaires.

On ne veut pas que si à l'instant t , la machine *est dans l'état q_k* et que *la*

tête de lecture/écriture est sur la case i qui *contient le symbole s_j* , alors à l'instant $t+1$:

- (1) son nouvel état *ne soit pas $q_{k'}$* :

$$\overline{\{Q[t, k], \overline{P[t, i]}, \overline{S[t, i, j]}, Q[t+1, k']\}} \equiv \overline{(Q[t, k] \wedge P[t, i] \wedge S[t, i, j] \wedge Q[t+1, k'])}.$$

- (2) la tête de lecture/écriture *ne soit pas* à la position $i + \Delta$:

$$\{Q[t, k], \overline{P[t, i]}, \overline{S[t, i, j]}, P[t+1, i + \Delta]\}.$$

- (3) la case i *ne contienne pas* le symbole $s_{j'}$:

$$\{Q[t, k], \overline{P[t, i]}, \overline{S[t, i, j]}, S[t+1, i, j']\}.$$

Résumons. L'instance (U, C) du problème SAT est définie par

$$\begin{aligned} U &:= \bigcup_{t \in I_T} \left[\left(\bigcup_{i \in I_Q} \{Q[t, i]\} \right) \cup \left(\bigcup_{i \in I_P} \{P[t, i]\} \right) \cup \left(\bigcup_{\substack{i \in I_P \\ j \in I_S}} \{S[t, i, j]\} \right) \right]. \\ C &= G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6. \end{aligned}$$

Nous venons de montrer que l'instance (U, C) du problème SAT est satisfaisable si et seulement si le mot $x = x_1x_2\dots x_n$ est reconnu par la machine de Turing non-déterministe polynomiale T . Il ne reste qu'à prouver que cette transformation est polynomiale. Un encodage raisonnable d'une instance du problème SAT est borné par un polynôme en la taille de l'ensemble U des variables propositionnelles. L'exercice ci-dessous permet de conclure.

Exercice 21 Dénombrez le nombre de variables propositionnelles de U , et le nombre de clauses de C en fonction de $p(n)$, r et ν et montrez qu'un encodage raisonnable de l'instance (U, C) est polynomial en n la taille de l'instance du problème II.

Problème † 22 [ALGORITHME DPLL⁴] On considère un ensemble de clauses $C = \{C_1, \dots, C_m\}$. On appelle *clause unitaire* toute clause qui ne contient qu'un seul littéral, et *littéral pur* tout littéral qui apparaît exclusivement sous forme positive (ou négative) dans les clauses de C .

(a) **Réduction par clauses unitaires.** On réduit C inductivement en éliminant pour chaque clause unitaire $\{\ell\}$, toutes les clauses qui contiennent le littéral ℓ , puis en éliminant le littéral opposé $\bar{\ell}$ de toutes les clauses qui contiennent $\bar{\ell}$. Par exemple, l'ensemble C ci-dessous contient la clause unitaire $\{\bar{u}_2\}$:

$$C := \{\{\bar{u}_1, u_2, u_3, \bar{u}_5, \bar{u}_4\}, \{\bar{u}_2\}, \{u_1, u_2, u_4\}, \{\bar{u}_2, u_3\}, \{u_2, u_5\}, \{u_3, \bar{u}_4\}\}.$$

On élimine les clauses $\{\bar{u}_2\}$ et $\{\bar{u}_2, u_3\}$ puis le littéral u_2 des autres clauses :

$$C \leftarrow \{\{\bar{u}_1, u_3, \bar{u}_5, \bar{u}_4\}, \{u_1, u_4\}, \{u_5\}, \{u_3, \bar{u}_4\}\}.$$

On a fait apparaître une nouvelle clause unitaire $\{u_5\}$ qui nous permet de réduire à nouveau :

$$C \leftarrow \{\{\bar{u}_1, u_3, \bar{u}_4\}, \{u_1, u_4\}, \{u_3, \bar{u}_4\}\}. \quad (13)$$

(b) **Réduction par littéraux purs.** On réduit C inductivement à nouveau en éliminant toutes les clauses qui contiennent un littéral pur. Le littéral u_3 dans (13) est pur, ce qui nous donne après réduction :

$$C \leftarrow \{\{u_1, u_4\}\}.$$

et conséquemment u_1 et u_4 sont des littéraux purs et une dernière réduction pour u_1 par exemple nous donne

$$C \leftarrow \emptyset.$$

Ces réductions ont instancié successivement les variables

$$u_2 \leftarrow \mathcal{F} \quad u_5 \leftarrow \mathcal{V} \quad u_3 \leftarrow \mathcal{V} \quad u_1 \leftarrow \mathcal{V}$$

ce qui suffit à satisfaire C .

On se place à présent dans le cas général d'un ensemble de clauses C quelconque.

(1) Démontrez que C est satisfaisable si et seulement si sa réduction par clauses unitaires est satisfaisable.

(2) Démontrez que C est satisfaisable si et seulement si sa réduction par littéraux purs est satisfaisable.

(3) Démontrez que si l'ensemble des clauses C est réduit à \emptyset après l'application des deux règles de réduction, alors C est satisfaisable.

(4) Que peut-on affirmer si en fixant la valeur de vérité d'une des variables, tous les littéraux d'une clause sont instanciés à la valeur \mathcal{F} ?

(5) Déduisez un algorithme récursif de résolution du problème SAT qui utilise ces deux règles de réduction. Indication : répétez les deux règles jusqu'à ce qu'il n'y ait plus de réduction. Si l'ensemble des clauses résiduelles R est vide alors C est satisfaisable. Sinon choisir arbitrairement une des variables résiduelles $\{u\}$ et relancer l'algorithme d'une part sur l'ensemble des clauses $R \cup \{u\}$ et d'autre part sur l'ensemble des clauses $R \cup \{\bar{u}\}$, ce qui revient à tester les deux hypothèses possibles pour la variable u . Si l'ensemble des clauses est satisfaisable, les instanciations des variables effectuées lors des appels récursifs définissent une interprétation prouvant la satisfaisabilité.

$$\begin{array}{cccccc} 5 & 2 & & & & \\ 1 & -3 & 4 & 5 & 0 & \\ -2 & 3 & 5 & 0 & & \end{array}$$

TABLE 2. Encodage d'une instance du problème SAT.

(6) Appliquer l'algorithme "à la main" sur l'instance suivante de SAT, en indiquant sur chaque branche de l'arbre binaire, l'instanciation de la variable correspondante :

$$C := \{\{u_1, \bar{u}_2, u_3\}, \{\bar{u}_1, u_4\}, \{\bar{u}_1, \bar{u}_4\}, \{u_2, u_5\}, \{u_2, \bar{u}_5\}, \{\bar{u}_3\}\}.$$

TP 1 Écrivez un programme dans le langage de votre choix qui décide si un ensemble de clauses est satisfaisable ou non à l'aide de l'algorithme DPLL. NB. Lors des appels récursifs, vous pourrez utiliser la valeur de retour de la fonction après l'appel pour la branche gauche afin de ne pas évaluer la branche droite au cas où la clause a été satisfaite.

Les clauses seront codées dans un fichier texte contenant une ligne par clause. Pour faciliter la lecture des données, la première ligne du fichier contiendra 2 entiers, le premier désignera le nombre de variables, le second le nombre de clauses. Chaque clause sera codée par une séquence d'entiers

relatifs non-nuls séparés par un espace et se terminant par la valeur 0. Si la valeur est positive, il s'agira du littéral positif, sinon du littéral négatif. L'instance $\{\{u_1, \bar{u}_3, u_4, u_5\}, \{\bar{u}_2, u_3, u_5\}\}$ est encodée dans le fichier en table 2.

Le nom du fichier contenant l'ensemble des clauses sera passé en paramètre sur la ligne de commande. On supposera que les n variables sont numérotées de 1 à n , qu'aucune clause ne contient plusieurs fois le même littéral ni un littéral et son opposé (le nombre de littéraux d'une clause est donc majoré par le nombre de variables) et qu'il n'y a jamais deux clauses identiques.

Le choix des structures de données devra être justifié et les algorithmes devront être analysés et leurs complexités estimées. Les fonctions devront impérativement être commentées.

Vous trouverez [ici](#) (si le site est toujours accessible) de nombreuses instances du problème SAT et [là](#) un petit script *Python* pour engendrer une instance de SAT au hasard avec pour paramètres le nombre de variables, le nombre de clauses ainsi que le nombre minimum et maximum de littéraux par clause.

TP 2 Écrivez un programme dans le langage de votre choix qui transforme une instance du Sudoku en instance SAT encodée conformément à la table 2. On encodera une instance du Sudoku dans un fichier texte contenant n^2 lignes de n^2 valeurs de $\llbracket 0, n^2 \rrbracket$ où 0 code la case vide. Résolvez une grille de Sudoku à l'aide de votre solveur DPLL.

10. QUELQUES PROBLÈMES NP-COMPLETS

Nous disposons à présent d'un premier problème NP-complet et pour prouver que d'autres problèmes de décision sont NP-complets, le résultat suivant nous suggère un raccourci :

Proposition 3. Soit Π un problème NP-complet et Π' un problème de décision. Si Π' satisfait les deux propositions

- (1) $\Pi' \in NP$,
- (2) $\Pi \propto_P \Pi'$.

Alors Π' est NP-complet.

Exercice 23 Démontrez cette proposition.

Pour démontrer qu'un problème Π' est NP-complet, il faut donc démontrer qu'il appartient à la classe NP, chercher un problème NP-complet Π qui lui ressemble et enfin trouver une transformation polynomiale de Π en Π' . À ce stade, les candidats sont peu nombreux, nous n'en connaissons qu'un ! Dans la suite nous présentons 10 nouveaux problèmes NP-complets.

Remarque. Les graphes constituent un modèle privilégié pour les problèmes de décision et nous emploierons souvent le vocabulaire des graphes orientés pour des graphes qui ne le sont pas. Nous parlerons d'un arc (x, y) quand il s'agit d'une paire $\{x, y\}$ par exemple. Ces abus de langage éviteront de doubler les problèmes dans leurs versions orientées/non orientées quand cela est sans conséquence.

Problème 3-SAT [SATISFAISABILITÉ DES CLAUSES À 3 LITTÉRAUX]

Instance : Un ensemble de variables booléennes U et un ensemble de clauses C à trois littéraux définies sur U .

Question : L'ensemble des clauses de C est-il satisfaisable ? Autrement dit, existe-t-il une interprétation $I : U \rightarrow \{\mathcal{V}, \mathcal{F}\}$ des variables de U qui satisfait

$$\forall c \in C \ I(c) = \mathcal{V}.$$

Exemple 11. Soit $U = \{u_1, u_2, u_3, u_4\}$ et $C = \{\{\bar{u}_1, u_2, u_3\}, \{u_1, \bar{u}_3, u_4\}\}$. Toutes les clauses de C (il y en a deux) ont exactement trois littéraux.

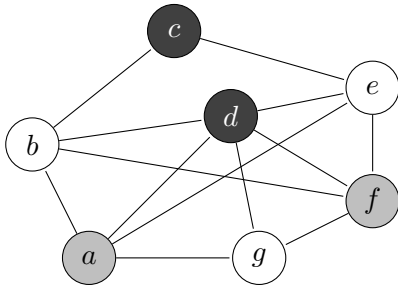
Problème k -COL [COLORIAGE D'UN GRAPHE AVEC k COULEURS]

Instance : Un graphe $G = (X, V)$.

Question : Peut-on colorier chaque sommet du graphe de manière à ce que deux sommets adjacents ne soient pas de la même couleur ? Autrement dit, existe-t-il une fonction $c : X \rightarrow \llbracket 1, k \rrbracket$ telle que

$$\forall (x, y) \in V, \ c(x) \neq c(y).$$

Exemple 12. Le graphe ci-dessous est 3-coloriable mais pas 2-coloriable (les couleurs utilisées sont le blanc, le gris et le noir).



Exercice 24 Démontrez que le problème du 2-coloriage est polynomial.

Problème 3-DM [MARIAGE TRI-DIMENSIONNEL]

Instance : Trois ensembles X , Y et Z deux-à-deux disjoints de même cardinal q et $T \subseteq X \times Y \times Z$ (ménages à trois).

Question : Existe-t-il une partie de M de T telle qu'aucun élément d'un ménage à trois ne soit déjà marié? Autrement dit, existe-t-il une partie $M \subseteq T$ telle que

$$(|M| = q) \wedge (\forall((x, y, z), (x', y', z')) \in M^2 (x \neq x') \wedge (y \neq y') \wedge (z \neq z')).$$

Exemple 13. Soit $q = 3$ et on considère les ensembles

$$X := \{\text{Bob, Al, Cob}\}$$

$$Y := \{\text{Léa, Flo, May}\}$$

$$Z := \{1, 2, 3\}$$

$$T := \{(\text{Bob, Léa, 2}), (\text{Bob, Flo, 3}), (\text{Al, May, 2}), \\ (\text{Cob, Flo, 1}), (\text{Cob, Léa, 1})\}$$

Cette instance du problème admet une solution :

$$M = \{(\text{Bob, Flo, 3}), (\text{Al, May, 2}), (\text{Cob, Léa, 1})\}.$$

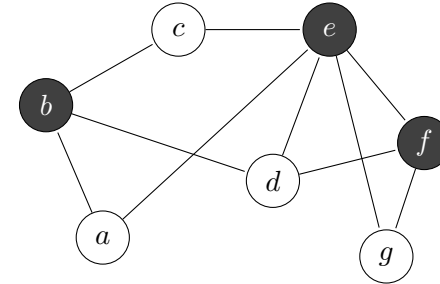
Problème VC [RECOUVREMENT PAR SOMMETS]

Instance : Un graphe $G = (X, V)$ et un entier positif $K \leq |X|$.

Question : Existe-t-il un sous-ensemble d'au plus K sommets contenant au moins une extrémité de chaque arc? Autrement dit, existe-t-il une partie $Y \subseteq X$ telle que

$$(|Y| \leq K) \wedge (\forall(x, y) \in V (x \in Y) \vee (y \in Y)).$$

Exemple 14. Pour le graphe ci-dessous et pour la borne $K = 3$, le problème admet une solution $Y = \{b, e, f\}$ (les sommets sont en noir dans le graphe). Pour $K = 2$ il n'y a pas de solution.



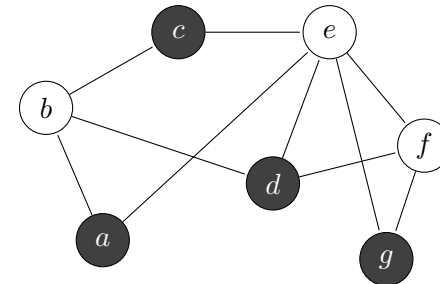
Problème INDSET [ENSEMBLE DE SOMMETS INDÉPENDANTS]

Instance : Un graphe $G = (X, V)$ et un entier positif $K \leq |X|$.

Question : Existe-t-il un sous-ensemble d'au moins K sommets sans aucun arc qui les relie? Autrement dit, existe-t-il une partie $Y \subseteq X$ telle que

$$(|Y| \geq K) \wedge (\forall(x, y) \in Y^2 (x, y) \notin V).$$

Exemple 15. Pour le graphe ci-dessous et pour $K = 4$, le problème admet une solution $Y = \{a, c, d, g\}$. Pour $K = 5$, il n'y a pas de solution.



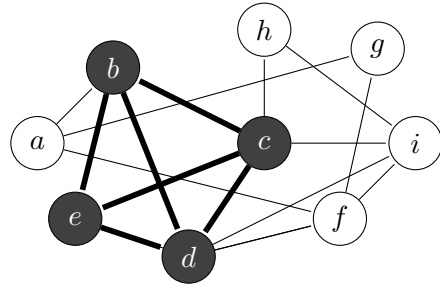
Problème CL [CLIQUE]

Instance : Un graphe $G = (X, V)$ et un entier positif $K \leq |X|$.

Question : Existe-t-il un sous-ensemble d'au moins K sommets tous adjacents? Autrement dit, existe-t-il une partie $Y \subseteq X$ telle que

$$(|Y| \geq K) \wedge (\forall(x, y) \in Y^2 (x, y) \in V).$$

Exemple 16. Pour le graphe ci-dessous et $K = 4$, le problème a une solution $Y = \{b, c, d, e\}$ (ces sommets sont en noir dans le graphe). Pour $K = 5$, il n'y a pas de solution.



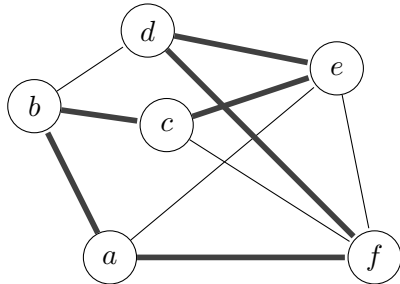
Problème HC [CIRCUIT HAMILTONIEN]

Instance : Un graphe $G = (X, V)$, $X = \{x_1, x_2, \dots, x_n\}$.

Question : Existe-t-il un circuit hamiltonien dans le graphe? Autrement dit, existe-t-il une permutation $\pi \in S_n$ telle que

$$(\forall i \in \llbracket 1, n-1 \rrbracket (x_{\pi(i)}, x_{\pi(i+1)}) \in V) \wedge ((x_{\pi(n)}, x_{\pi(1)}) \in V).$$

Exemple 17. Pour le graphe ci-dessous, il existe (au moins) un circuit hamiltonien (a, b, c, e, d, f, a) (matérialisé en gras dans le graphe).



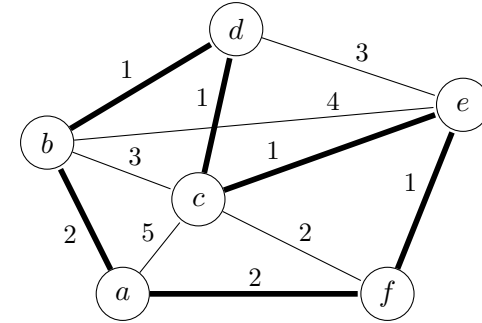
Problème TSP [TRAVELING SALESMAN PROBLEM]

Instance : Un graphe non-orienté $G = (X, V)$ avec $X = \{x_1, \dots, x_n\}$ (villes), une pondération $w : V \rightarrow \mathbb{N}$ (distances) et une distance maximale $K \geq 0$.

Question : Existe-t-il un circuit hamiltonien de distance totale au plus K ? Autrement dit, existe-t-il un circuit hamiltonien $\pi \in S_n$ tel que

$$\left(\sum_{i=1}^{n-1} w(x_{\pi(i)}, x_{\pi(i+1)}) \right) + w(x_{\pi(n)}, x_{\pi(1)}) \leq K.$$

Exemple 18. Pour le graphe pondéré ci-dessous, le problème admet une solution pour $K = 9$, en effet le circuit (a, b, d, c, e, f, a) est de longueur 8 (matérialisé en gras dans le graphe).



Problème XC [COUVERTURE EXACTE]

Instance : Un ensemble E et $S \subseteq \mathcal{P}(E)$.

Question : Existe-t-il une partie de S qui soit une partition (couverture exacte) de E ? Autrement dit, existe-t-il $P \subseteq S$ telle que

$$(\forall A \in P \ A \neq \emptyset) \wedge (\forall (A, B) \in P^2 \ A \cap B = \emptyset) \wedge \left(\bigsqcup_{A \in P} A = E \right).$$

Une couverture exacte désigne une partition en français, le mot couverture faisant référence au mot recouvrement, une partition est en effet un recouvrement parfait, au sens où il n'y a pas de chevauchement des pièces. Le mot partition en anglais désigne une césure, une séparation et il est employé pour qualifier le prochain problème NP-complet.

Exemple 19. Soit $E = \{1, 2, 3, 4, 5\}$ et $S = \{\{1, 2\}, \{3\}, \{4\}, \{1, 5\}, \{5\}\}$. La réponse est oui avec par exemple $P := \{\{1, 2\}, \{3\}, \{4\}, \{5\}\}$.

Problème PART [PARTITION]

Instance : Un ensemble fini X et une fonction $w : X \rightarrow \mathbb{N}$ (poids).

Question : Existe-t-il une partie de X de même poids que son complémentaire? Autrement dit, existe-t-il une partie $Y \subseteq X$ telle que

$$\sum_{x \in Y} w(x) = \sum_{x \in X \setminus Y} w(x).$$

Exemple 20. Considérons $X = \{a, b, c, d, e, f\}$ avec la fonction poids w définie par

x	a	b	c	d	e	f
$w(x)$	2	3	1	5	2	3

L'ensemble $Y := \{a, b, f\}$ permet de répondre positivement à cette question pour cette instance, en effet $w(a) + w(b) + w(f) = w(c) + w(d) + w(e) = 8$.

Théorème 9. Soit $G = (X, V)$ un graphe et Y un sous-ensemble de l'ensemble des sommets X . Les trois assertions suivantes sont équivalentes :

- (1) Y est un recouvrement par sommets de X .
- (2) $X \setminus Y$ est un ensemble de sommets indépendants.
- (3) $X \setminus Y$ est une clique du graphe complémentaire $\overline{G} = (X, \overline{V})$.

Les preuves de NP-complétude peuvent se faire dans l'ordre indiqué dans l'arbre enraciné en SAT qui est présenté en figure 8.

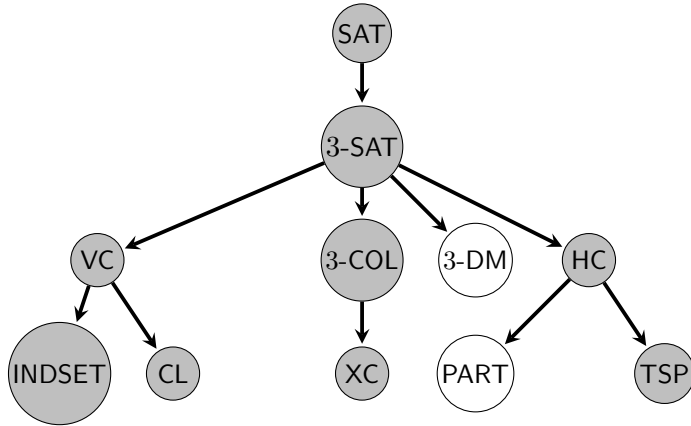


FIGURE 8. Ordre des preuves de NP-complétude. En gris les problèmes dont une preuve de NP-complétude est proposée dans ce document.

Proposition 4. Le problème 3-SAT de satisfaisabilité des clauses à 3 littéraux est NP-complet.

Démonstration. Les instances de 3-SAT étant des cas particuliers des instances de $\text{SAT} \in \text{NP}$, on en déduit que $3\text{-SAT} \in \text{NP}$. Il nous reste à montrer

que $\text{SAT} \leq_P 3\text{-SAT}$. On note $U := \{u_1, u_2, \dots, u_n\}$ l'ensemble des variables propositionnelles et $C = \{C_1, C_2, \dots, C_m\}$ l'ensemble des clauses du problème SAT. On note U' et C' les homologues du problème 3-SAT avec :

$$U' := U \cup \left(\bigcup_{i=1}^m U'_i \right) \quad \text{et} \quad C' := \bigcup_{i=1}^m C'_i$$

Soit $i \in \llbracket 1, m \rrbracket$ et notons la clause $C_i := \{l_{i_1}, l_{i_2}, \dots, l_{i_k}\}$. Nous allons expliciter la construction des ensembles U'_i et les C'_i qui ne dépendent que du nombre k de littéraux de la clause C_i :

($k = 1$) Dans ce cas $U'_i := \{a_i, b_i\}$ et

$$C'_i := \left\{ \{l_{i_1}, a_i, b_i\}, \{l_{i_1}, a_i, \overline{b_i}\}, \{l_{i_1}, \overline{a_i}, b_i\}, \{l_{i_1}, \overline{a_i}, \overline{b_i}\} \right\}. \quad (14)$$

Si la clause $C_i = \{l_{i_1}\}$ est satisfaite, C'_i est satisfaite puisque le littéral l_{i_1} est présent dans chacune des clauses de C'_i . Réciproquement C'_i contient les 4 disjonctions possibles des variables a_i et b_i avec l_{i_1} , ainsi, quelle que soit l'interprétation de a_i et b_i , l'une de ces disjonctions est fausse, ce qui impose que l_{i_1} doit être vrai pour satisfaire C'_i satisfaisant du même coup la clause C_i .

($k = 2$) Dans ce cas $U'_i := \{a_i\}$ et

$$\{l_{i_1}, l_{i_2}, a_i\}, \{l_{i_1}, l_{i_2}, \overline{a_i}\}. \quad (15)$$

et on fait le même raisonnement que pour $k = 1$ mais sur l'interprétation de la seule variable a_i .

($k = 3$) Dans ce cas $U'_i := \emptyset$, $C'_i := \{C_i\}$ et il n'y a rien à prouver.

($k \geq 4$) Dans ce cas $U'_i := \{a_{i_j} \mid j \in \llbracket 1, k-3 \rrbracket\}$ et

$$C'_i := \left\{ \{l_{i_1}, l_{i_2}, a_{i_1}\} \right\} \cup \bigcup_{j=1}^{k-4} \left\{ \{\overline{a_{i_j}}, l_{i_{j+2}}, a_{i_{j+1}}\} \right\} \cup \left\{ \{\overline{a_{i_k}}, l_{i_{k-1}}, l_{i_k}\} \right\}. \quad (16)$$

Si C_i est satisfaisable alors il existe un plus petit indice r tel qu'un littéral l_{i_r} est vrai. Si $r \in \{1, 2\}$, alors on fixe toutes les variables a_{i_j} à \mathcal{V} et si $i \in \{k-1, k\}$, elles sont fixées à \mathcal{F} . Sinon on fixe a_{i_j} à \mathcal{V} pour tous les indices $j \in \llbracket 1, r-2 \rrbracket$ et à \mathcal{F} pour tous les indices $j \in \llbracket r-1, k-3 \rrbracket$. On vérifie aisément que dans ce cas C'_i est satisfaite et réciproquement que si C'_i est satisfaisable, alors il existe nécessairement un littéral l_{i_r} qui est vrai. Le nombre de clauses à trois variables de C' est borné par nm et la

construction est une double itération sur l'ensemble des clauses de C et chaque littéral de ces clauses, la transformation est bien polynomiale. \square

La question qui vient à l'esprit immédiatement est de savoir si en limitant la taille des clauses à deux variables le problème reste toujours aussi difficile.

Théorème 10. *Le problème 2-SAT de satisfaisabilité des clauses à 2 littéraux est polynomial.*

Démonstration. Preuve en rédaction, cf. algorithme de [Tarjan](#). \square

Proposition 5. *Le problème HC du circuit hamiltonien est NP-complet.*

Démonstration. On a $HC \in NP$. En effet, si l'on se donne comme certificat une permutation des $|X|$ sommets du graphe $G := (X, V)$, il suffit de vérifier qu'il s'agit bien d'un circuit de taille $|X|$ ce qui se fait en temps linéaire $O(|X|)$.

Nous allons montrer que $3\text{-SAT} \propto_P HC$. Nous illustrerons cette construction avec l'ensemble des variables booléennes $U := \{u_1, u_2, u_3, u_4\}$ et l'ensemble des clauses

$$C := \left\{ \underbrace{\{u_1, u_2, \bar{u}_3\}}_{C_1}, \underbrace{\{u_1, \bar{u}_2, u_4\}}_{C_2}, \underbrace{\{\bar{u}_2, u_3, u_4\}}_{C_3} \right\}.$$

On note $n := |U|$ et $m := |C|$. Le graphe G est constitué, entre autres, de n chaînes notées ∞_i , $i \in \llbracket 1, n \rrbracket$ comprenant chacune une alternance de m sommets "gauches" $G_{i,j}$ et "droits" $D_{i,j}$ pour $j \in \llbracket 1, m \rrbracket$ (cf. figure 9 pour la chaîne ∞_1 de notre exemple) :



FIGURE 9. La chaîne ∞_1 associée à la variable u_1 .

La chaîne ∞_i est définie par les arcs

$$\begin{aligned} \infty_i := & \left(\bigcup_{j=1}^{m-1} \{(G_{i,j}, D_{i,j}), (D_{i,j}, G_{i,j+1})\} \right) \cup \{(G_{i,m}, D_{i,m})\} \\ & \cup \left(\bigcup_{j=m}^2 \{(D_{i,j}, G_{i,j}), (G_{i,j}, D_{i,j-1})\} \right) \cup \{(D_{i,1}, G_{i,1})\}. \end{aligned}$$

Un circuit hamiltonien ne peut traverser une chaîne ∞_i que de gauche à droite en entrant en $G_{i,1}$ et en sortant en $D_{i,m}$ ou réciproquement puisque chaque sommet doit être visité exactement une fois. L'idée qui sous-tend la suite de la construction du graphe G autour des n chaînes est qu'il doit fournir autant de circuits hamiltoniens qu'il y a d'interprétations des variables booléennes possibles, i.e. 2^n .

Dans cette optique, on connecte chaque chaîne ∞_i à la suivante ∞_{i+1} pour $i \in \llbracket 1, n-1 \rrbracket$ par les 2 extrémités, soit 4 arcs :

$$E_i := \{(G_{i,1}, G_{i+1,1}), (G_{i,1}, D_{i+1,m}), (D_{i,m}, G_{i+1,1}), (D_{i,m}, D_{i+1,m})\}.$$

Ainsi, une fois la chaîne ∞_i traversée (dans un sens ou l'autre), on peut passer à la chaîne suivante ∞_{i+1} de deux façons différentes, par la gauche ou la droite (cf. figure 10). On rajoute un sommet "source" S et un sommet "puits" P connectés respectivement aux deux extrémités de la première et de la dernière chaîne et reliés entre eux par l'arc $P \rightarrow S$.

Pour achever la construction du graphe, on crée un sommet C_j pour chacune des m clauses C_j et si u_i (resp. \bar{u}_i) est un littéral de C_j , on crée une dérivation dans la chaîne ∞_i par le sommet C_j grâce au chemin $G_{i,j} \rightarrow C_j \rightarrow D_{i,j}$ (resp. $D_{i,j} \rightarrow C_j \rightarrow G_{i,j}$). On a donc pour ensemble de sommets

$$X := \left(\bigcup_{i=1}^n \bigcup_{j=1}^m \{G_{i,j}, D_{i,j}\} \right) \cup \left(\bigcup_{j=1}^m \{C_j\} \right) \cup \{S, P\}.$$

et pour ensemble d'arcs

$$\begin{aligned} V := & \left(\bigcup_{i=1}^n \infty_i \right) \cup \left(\bigcup_{i=1}^{n-1} E_i \right) \cup \left(\bigcup_{j=1}^m \bigcup_{k=1}^3 \{([C_j, G_{j,k,j}], [D_{i,j_k}, C_j])\} \right) \\ & \cup \{(S, G_{1,1}), (S, D_{1,m}), (G_{n,1}, P), (D_{n,m}, P), (P, S)\}. \end{aligned}$$

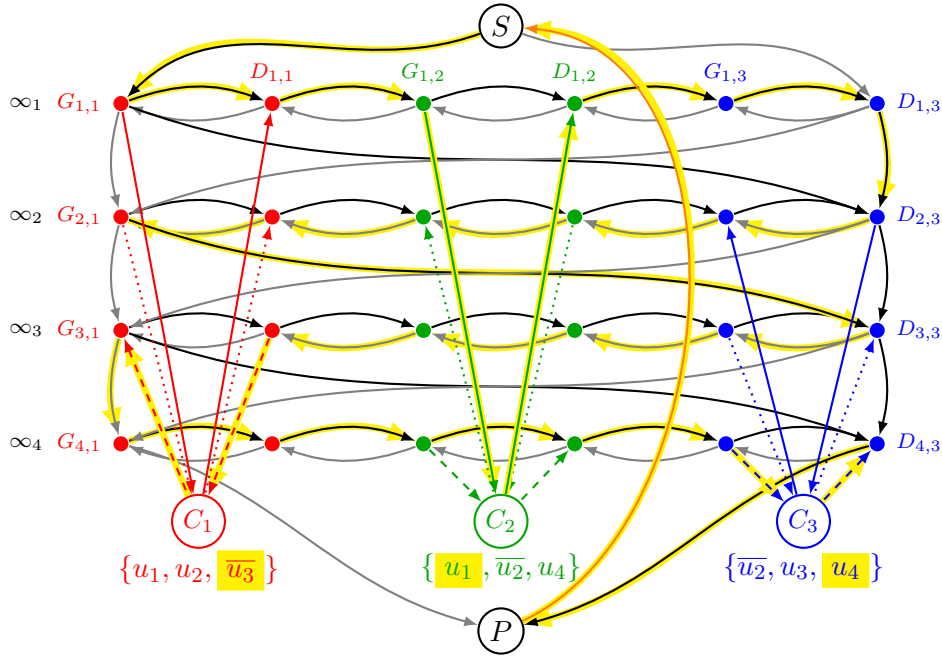


FIGURE 10. Circuit hamiltonien \longrightarrow associé à l'instance C du problème 3-SAT si les littéraux \bar{u}_3 , u_1 et u_4 sont vrais.

où j_k pour $k \in \{1, 2, 3\}$ désigne le numéro de la k -ème variable booléenne impliquée dans la clause C_j et $[C_j, G_{j_k, j}]$ désigne le couple $(C_j, G_{j_k, j})$ si le j_k -ème littéral de C_j est positif et le couple $(G_{j_k, j}, C_j)$ sinon.

Il faut à présent démontrer que l'instance de 3-SAT est satisfaisable si et seulement s'il existe un circuit hamiltonien dans ce graphe. Supposons que C soit satisfaisable. Dans ce cas, il existe (au moins) un littéral vrai dans chacune des clauses C_j , par exemple \bar{u}_3 dans la clause C_1 , u_1 dans la clause C_2 et u_4 dans la clause C_3 .

On traverse alors les 3 chaînettes ∞_3 , ∞_1 et ∞_4 en faisant un détour par les sommets respectifs C_1 , C_2 et C_3 . Cette traversée se fait de la droite vers la gauche pour la chaînette ∞_3 car le littéral \bar{u}_3 est négatif, et de la gauche vers la droite pour les deux autres car les littéraux u_1 et u_4 sont positifs. Les chaînettes sont connectées entre elles par la gauche ou

la droite selon le sens des traversées qui auront été fixées par les littéraux correspondants. La source S et le puit P sont connectés à la première et la dernière chaîne respectivement et le circuit passe nécessairement par l'arc $P \rightarrow S$. Si plusieurs littéraux sont vrais dans une même clause, une seule chaînette est détournée vers cette clause. Ce circuit passe exactement une fois par chacun des sommets, on a donc construit un circuit hamiltonien (voir ce circuit en jaune fluo dans la figure 10).

Réciproquement, supposons que l'on dispose d'un circuit hamiltonien. Il faut trouver une interprétation I des variables de U telle que les clauses C_j soient toutes satisfaites. Le circuit étant dérivé vers chacun des sommets C_j , le sens de la dérivation par une chaînette ∞_i détermine l'interprétation du littéral u_i $I(u_i) := \mathcal{V}$ de gauche à droite et $I(u_i) := \mathcal{F}$ sinon. On satisfait ainsi C .

Cette transformation est polynomiale en le nombre n de variables booléennes et m de clauses. En effet, on a $2m$ sommets par variable, soit au total $2mn$ sommets, plus les m sommets clauses ainsi que la source et le puit :

$$|X| = 2mn + m + 2.$$

On dénombre $2m + 2(m - 1)$ arcs par chaînette, soit $n(4m - 2)$ au total, $4(n - 1)$ arcs pour relier ces chaînettes entre elles, $6m$ arcs qui relient les chaînettes ∞_i aux clauses C_j , et 5 arcs pour relier les chaînettes à la source S et au puit P et enfin le puit à la source, soit

$$\begin{aligned} |V| &= n(4m - 2) + 4(n - 1) + 6m + 5 \\ &= 4nm + 2n + 6m + 1. \end{aligned}$$

□

Proposition 6. *Le problème TSP du voyageur de commerce est NP-complet.*

Démonstration. Nous savons déjà que $TSP \in NP$ et nous allons prouver que $HC \propto_P TSP$. La transformation est quasiment directe, les deux problèmes étant essentiellement les mêmes. Il faut simplement définir la fonction de pondération sur les arcs et la borne K sur la distance totale du circuit hamiltonien pour l'instance de TSP.

Soit $G = (X, V)$ une instance du problème HC avec $n := |X|$ et $m := |V|$. Nous illustrerons la preuve avec les ensembles

$$\begin{aligned} X &:= \{1, 2, 3, 4, 5\}, \\ V &:= \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5)\}. \end{aligned}$$

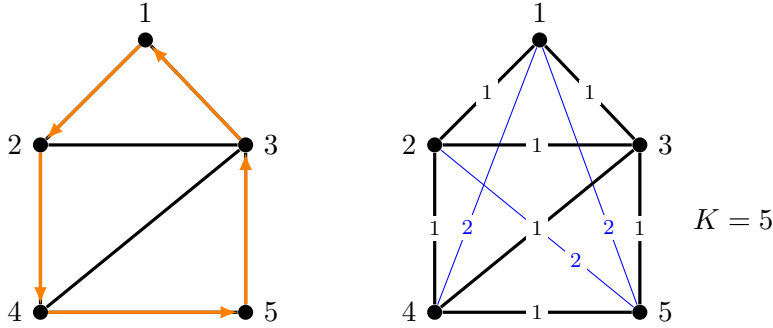


FIGURE 11. À gauche : Instance du problème HC et **circuit hamiltonien**. À droite : instance associée du problème TSP.

On transforme l'instance $G = (X, V)$ du problème HC en une instance $G' = (X, V')$, K et $w : V' \rightarrow \mathbb{N}$ du problème TSP comme suit : G' est le **graphe complet** d'ensemble de sommets X i.e. $V' := X \times X$, la borne $K := n$ est fixée au nombre de sommets de $|X|$ et la pondération w est définie en tout arc $(x, y) \in X \times X$ par

$$w(x, y) := \begin{cases} 1 & \text{si } (x, y) \in V, \\ 2 & \text{si } (x, y) \notin V. \end{cases}$$

L'instance de HC et l'instance de TSP associée sont représentées respectivement à gauche et à droite de la figure 11.

Si l'on dispose d'un circuit hamiltonien, il le reste pour l'instance du voyageur de commerce et la distance totale de ce circuit de n arcs de pondération 1 est donc égale à K . L'instance de TSP ainsi construite est donc positive. Réciproquement, s'il existe un circuit hamiltonien de distance totale inférieure à $K = n$, alors ce circuit ne peut traverser que des arcs de V sans quoi la distance totale excéderait n puisque les **autres arcs** sont pondérés

par la distance 2. Ce circuit est donc une solution du problème HC et cette transformation est évidemment polynomiale. \square

Proposition 7. *Le problème VC du recouvrement par sommets est NP-complet.*

Démonstration. On a $VC \in NP$. En effet, il suffit de fournir le recouvrement Y en guise de certificat et de vérifier, d'une part que $|Y| \leq K$, ce qui se fait en $O(n)$ où $n := |X|$, et d'autre part, que pour chaque arc $(x, y) \in V$ on a $(x \in Y) \vee (y \in Y)$, ce qui se fait en $O(m)$ où $m := |V|$.

Montrons que $3\text{-SAT} \leq_P VC$. Il faut construire un graphe $G = (X, V)$ admettant un recouvrement Y de taille $\leq K$ (à fixer) si et seulement si l'instance de 3-SAT à laquelle il est associé est satisfaisable. On note $U := \{u_1, \dots, u_n\}$ l'ensemble des variables booléennes et $C := \{C_1, \dots, C_m\}$ l'ensemble des clauses. On crée deux groupes de graphes, l'un associé aux variables, l'autre aux clauses. Le premier est constitué de n *duos* de sommets u_i et \bar{u}_i reliés par l'arc (u_i, \bar{u}_i) notés $[u_i, \bar{u}_i] := \{(u_i, \bar{u}_i)\}$ pour $i \in \llbracket 1, n \rrbracket$. Le second est constitué de m *trios* de sommets a_j , b_j et c_j reliés entre eux par les trois arcs (a_j, b_j) , (a_j, c_j) et (b_j, c_j) notés $[a_j, b_j, c_j] := \{(a_j, b_j), (a_j, c_j), (b_j, c_j)\}$ pour $j \in \llbracket 1, m \rrbracket$ (cf. figure 12).



FIGURE 12. Duo $[u_i, \bar{u}_i]$ associé à la variable booléenne u_i et trio $[a_j, b_j, c_j]$ associé à la clause C_j .

Pour contrôler l'arc d'un duo, un recouvrement Y doit contenir l'un des deux sommets u_i ou \bar{u}_i et deux des trois sommets a_j , b_j et c_j pour contrôler les trois arcs d'un trio $[a_j, b_j, c_j]$. Un recouvrement Y contient donc **au moins** $n + 2m$ sommets. C'est précisément la valeur que l'on fixe à la borne K , soit le nombre minimal de sommets pour contrôler tous les arcs des duos et des trios.

On connecte maintenant les duos avec les trios suivant le contenu de chaque clause C_j , $j \in \llbracket 1, m \rrbracket$: on relie respectivement les sommets a_j , b_j et c_j

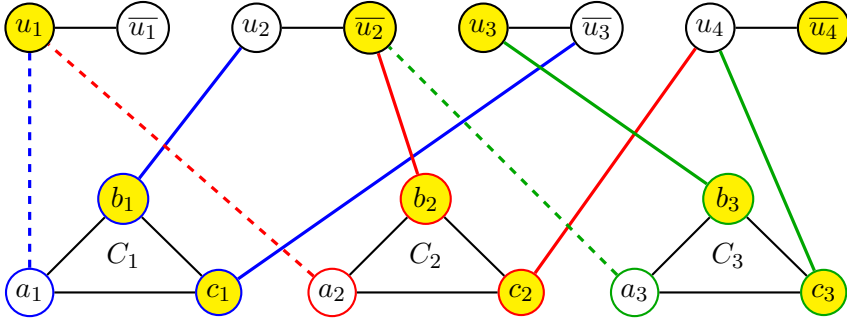


FIGURE 13. Duos $[u_i, \bar{u}_i]$ et trios $[a_j, b_j, c_j]$ et recouvrement par sommets \mathbf{Y} des arcs du graphe.

aux trois littéraux de C_j dans l'ordre d'apparition. Ce graphe est illustré en figure 13 pour l'ensemble des variables U et l'ensemble des clauses C suivants :

$$U := \{u_1, u_2, u_3, u_4\}$$

$$C := \left\{ \underbrace{\{u_1, u_2, \bar{u}_3\}}_{C_1}, \underbrace{\{u_1, \bar{u}_2, u_4\}}_{C_2}, \underbrace{\{\bar{u}_2, u_3, u_4\}}_{C_3} \right\}.$$

On a donc pour ensemble de sommets du graphe G l'ensemble

$$X := \left(\bigcup_{i=1}^n \{u_i, \bar{u}_i\} \right) \cup \left(\bigcup_{j=1}^m \{a_j, b_j, c_j\} \right).$$

et pour ensemble des arcs l'ensemble

$$V := \left(\bigcup_{i=1}^n [u_i, \bar{u}_i] \right) \cup \left(\bigcup_{j=1}^m [a_j, b_j, c_j] \right) \cup \left(\bigcup_{j=1}^m \{(a_j, l_{j_1}), (a_j, l_{j_2}), (a_j, l_{j_3})\} \right)$$

où l_{j_k} désigne le k -ème littéral de la clause C_j pour $k \in \{1, 2, 3\}$.

Il faut montrer que C est satisfaisable si et seulement si le graphe $G := (X, V)$ admet un recouvrement Y des arcs tel que $|Y| \leq n + 2m$. On pose $Y := Y_U \cup Y_C$ où Y_U et Y_C sont des recouvrements des arcs des duos des trios respectivement et que nous allons expliciter. Si C est satisfaisable, alors il existe une interprétation I des variables telle que toutes les clauses

C_j sont satisfaites, par exemple ici $I(u_1) = \mathcal{V}$, $I(u_2) = \mathcal{F}$, $I(u_3) = \mathcal{V}$ et $I(u_4) = \mathcal{F}$. Dans ce cas Y_U est fixé à l'ensemble de ces n littéraux :

$$Y_U := \bigcup_{i=1}^n \{\tilde{u}_i\} \quad \text{où} \quad \tilde{u}_i := \begin{cases} u_i & \text{si } I(u_i) = \mathcal{V} \\ \bar{u}_i & \text{si } I(u_i) = \mathcal{F} \end{cases}.$$

soit $Y_U := \{u_1, \bar{u}_2, u_3, \bar{u}_4\}$ pour notre exemple.

Pour chacune des m clauses C_j il faut déterminer quels sont les deux sommets du trio $[a_j, b_j, c_j]$ qui feront partie du recouvrement Y_C . Parmi les trois arcs qui relient un trio C_j à ses 3 littéraux, l'un d'entre eux au moins est contrôlé par un de ces littéraux, on considère le premier d'entre eux dans l'ordre a, b, c . Dans notre exemple, il s'agit des arcs (a_1, u_1) , (a_2, u_1) et (a_3, \bar{u}_2) en traitillés dans la figure 13. On rappelle que l_{j_k} désigne le littéral numéro $k \in \{1, 2, 3\}$ de la clause C_j , alors on pose

$$Y_C := \bigcup_{j=1}^m Y_j \quad \text{où} \quad Y_j := \begin{cases} \{a_j, b_j\} & \text{si } I(l_{j_3}) = \mathcal{V}, I(l_{j_1}) = \mathcal{F}, I(l_{j_2}) = \mathcal{F} \\ \{a_j, c_j\} & \text{si } I(l_{j_2}) = \mathcal{V} \text{ et } I(l_{j_1}) = \mathcal{F} \\ \{b_j, c_j\} & \text{si } I(l_{j_1}) = \mathcal{V} \end{cases}.$$

On a trivialement $|Y|_U = n$ et $|Y|_C = 2m$ et l'ensemble $Y = Y_U \sqcup Y_C$ est bien un recouvrement des arcs de taille égale à $n + 2m$.

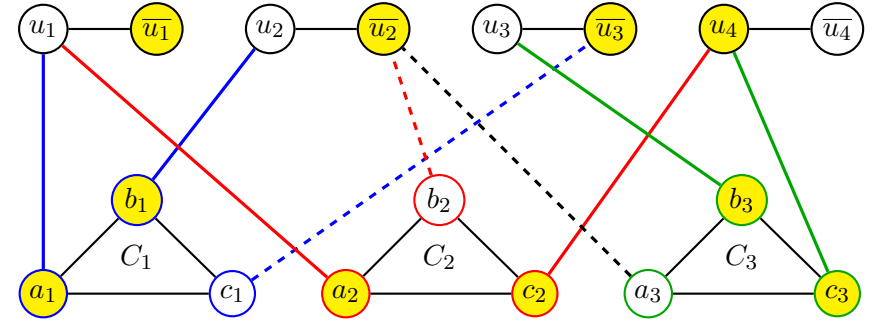


FIGURE 14. Recouvrement \mathbf{Y} des arcs du graphe.

Réciproquement, on suppose que l'on dispose d'un recouvrement par les sommets Y de taille égale à $n + 2m$ (on ne peut pas faire moins), montrons que la clause C est satisfaisable. Nous savons déjà que ce recouvrement contient nécessairement un sommet de chaque duo et deux sommets de

chaque trio. Dans l'exemple suivant (cf. figure 14) qui reprend le graphe de la figure 13, on a le recouvrement suivant :

$$Y = \underbrace{\{\overline{u_1}, \overline{u_2}, \overline{u_3}, u_4\}}_{Y_U} \cup \underbrace{\{a_1, b_1, a_2, c_2, b_3, c_3\}}_{Y_C}.$$

Il faut construire une interprétation I des variables booléennes sur la base de ce recouvrement Y qui satisfait C . Pour tous les littéraux $l \in Y_U$ on pose $I(l) := \mathcal{V}$ et pour les autres $I(l) := \mathcal{F}$, il reste à s'assurer que chaque clause C_j contient un littéral de Y_U . Le recouvrement Y_C ne contient que deux des trois sommets du trio $[a_j, b_j, c_j]$ et l'arc qui relie celui qui n'appartient pas à Y_C à un sommet d'un duo $[u_i, \overline{u_i}]$ est donc nécessairement couvert par un littéral de Y_U ce qui achève de prouver que C est satisfaisable avec cette interprétation.

Montrons pour finir que cette transformation est polynomiale. On dénombre aisément le nombre de sommets et d'arcs du graphe : $|X| = 2n + 3m$ et $|V| = n + 6m$ et il est évident que sa construction se fait en temps $O(n + m)$. \square

Théorème 11. Soit $G = (X, V)$ un graphe et $Y \subseteq X$. Les trois assertions suivantes sont équivalentes :

- (1) Y est un recouvrement par les sommets.
- (2) $X \setminus Y$ est un ensemble de sommets indépendants.
- (3) $X \setminus Y$ est une clique pour le graphe $\overline{G} = (X, \overline{V})$ où $\overline{V} := (X^2 \setminus V)$.

Démonstration. Nous allons montrer que (1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1), les équivalences sont obtenues par transitivité de l'implication logique. Pour illustrer la preuve, on considère le graphe de la figure 15 avec les ensembles de sommets et les arcs suivants

$$X := \{1, 2, 3, 4, 5, 6\}$$

$$V = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (1, 5), (3, 5), (5, 6)\}$$

On vérifie aisément que $Y := \{1, 4, 5\}$ est un recouvrement des arcs de G et que $\overline{Y} = X \setminus Y = \{2, 3, 6\}$ est un ensemble de sommets indépendants de G et une clique de \overline{G} . D'autre part, on a

$$\overline{V} = \{(1, 6), (2, 3), (2, 5), (2, 6), (3, 6), (4, 5), (4, 6)\}.$$

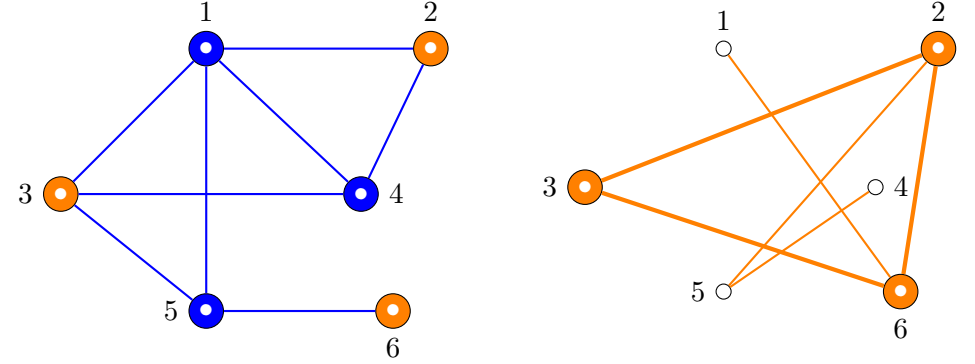


FIGURE 15. À gauche : un graphe $G = (X, V)$ et un recouvrement Y . À droite : le graphe complémentaire $\overline{G} = (X, \overline{V})$, un ensemble de sommets indépendants \overline{Y} du graphe G et une clique du graphe \overline{G} .

(1) \Rightarrow (2) Par hypothèse, Y est un recouvrement des arcs de G . Soit x et y deux sommets du complémentaire \overline{Y} . Si ces deux sommets étaient connectés, i.e. si $(x, y) \in V$ alors on aurait $(x \in Y) \vee (y \in Y)$ ce qui est absurde puisque $x \notin Y$ et $y \notin Y$.

(2) \Rightarrow (3) Puisqu'aucun couple d'éléments de \overline{Y} n'est connecté, par définition, ils le sont tous dans le graphe complémentaire, autrement dit, ils forment une clique.

(3) \Rightarrow (1) On sait que \overline{Y} constitue une clique dans le graphe complémentaire $\overline{G} = (X, \overline{V})$. Montrons que Y est un recouvrement des arcs du graphe $G = (X, V)$. Soit $(x, y) \in V$, il faut montrer que $(x \in Y) \vee (y \in Y)$. Par l'absurde, on suppose que $(x \in \overline{Y}) \wedge (y \in \overline{Y})$ autrement dit x et y appartiennent à la clique du graphe complémentaire, i.e. $(x, y) \in \overline{V} \Leftrightarrow (x, y) \notin V$, ce qui est absurde. \square

Corollaire 1. On a l'équivalence suivante : $VC_{\infty P} \text{ INDSET}_{\infty P} \text{ CL}$. Ces trois problèmes sont donc NP-complets.

Démonstration. Un recouvrement Y de taille $\leq K$ du graphe $G = (X, V)$ se transforme en un ensemble de sommets indépendants \bar{Y} de taille $\geq n - K$ en $O(n)$ où $n = |X|$, donc $VC \propto_P \text{INDSET}$. Ce même ensemble Y constitue une clique de taille $\geq n - K$ pour le graphe complémentaire \bar{G} qui se construit en temps $O(n)$ également, d'où $\text{INDSET} \propto_P \text{CL}$. On construit le graphe G à partir du graphe \bar{G} en temps linéaire en $O(n)$ et on transforme une clique \bar{Y} de \bar{G} en un recouvrement Y de G toujours en $O(n)$, soit $\text{CL} \propto VC$. Le problème VC étant NP-complet, on peut conclure. \square

Théorème 12. *Le problème 3-COL du 3-coloriage est NP-complet.*

Démonstration. Le problème est dans la classe NP. En effet, si l'on fournit un 3-coloriage en guise de certificat, il suffit de parcourir l'ensemble des arcs (x, y) du graphe et de vérifier que x et y ont des couleurs différentes tout en comptabilisant le nombre de couleurs distinctes utilisées pour colorier les différents sommets. Le nombre d'arcs m est majoré par n^2 où $n = |X|$ est le nombre de sommets, l'algorithme est linéaire en m donc polynomial en n et m .

Nous allons prouver que $3\text{-SAT} \propto_P 3\text{-COL}$. On considère (U, C) une instance de 3-SAT où $U = \{u_1, \dots, u_n\}$ est l'ensemble des n variables booléennes et $C = \{C_1, \dots, C_m\}$ est l'ensemble des m clauses. On note $G = (X, V)$ l'instance de 3-COL à construire à partir de (U, C) . Pour réaliser un 3-coloriage nous utiliserons les 3 couleurs noir, gris et blanc.

On crée deux groupes de sommets, le premier est associé aux variables booléennes u_i , le second aux clauses C_j de l'instance de 3-SAT, nous verrons plus loin comment les sommets de ces deux groupes sont connectés entre eux.

Groupe 1 : À chaque variable $u_i \in U$, on associe un duo $U_i := \{u_i, \bar{u}_i\}$ de sommets u_i et \bar{u}_i reliés par l'arc (u_i, \bar{u}_i) (cf. graphe à gauche dans la figure 16)..

Groupe 2 : À chaque clause $C_j = \{a_j, b_j, c_j\}$, on associe une grappe $X_j := \{k_j, l_j, m_j, n_j, o_j, v_j\}$ de 6 sommets reliés par les arcs $T_j := \{(o_j, v_j), (k_j, l_j), (l_j, n_j), (o_j, n_j), (v_j, n_j), (k_j, m_j), (l_j, m_j)\}$. On connecte les trois sommets/littéraux a_j, b_j et c_j de la clause C_j — ils font partie des $2n$ duos précédemment créés — à cette grappe, via les arcs $(c_j, o_j), (b_j, k_j)$ et (a_j, m_j) matérialisés en traitillés dans le graphe au centre de la figure 16.

Les sommets k_j, l_j, m_j, n_j et o_j deviendront rapidement accessoires et pour simplifier la représentation du graphe, on codera la grappe X_j à l'aide du schéma à droite de la figure 16.

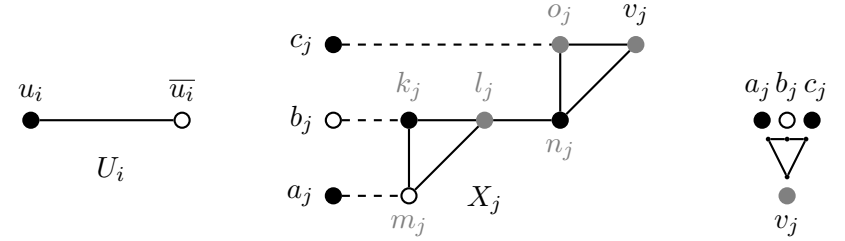


FIGURE 16. À gauche : un duo U_i . Au centre : une grappe X_j . À droite : notation condensée de la grappe.

On achève la construction en rajoutant deux derniers sommets S et P reliés par un arc : le sommet “source” S est également connecté à chacun des $2n$ sommets “littéraux” u_i et \bar{u}_i du groupe 1 et le sommet “puits” P est connecté à toutes les “sorties” v_j du groupe 2. On a donc construit le graphe $G = (X, V)$ avec les sommets et les arcs suivants :

$$X := \left(\bigcup_{i=1}^n U_i \right) \cup \left(\bigcup_{j=1}^m X_j \right) \cup \{S, P\}$$

$$V := \left(\bigcup_{i=1}^n \{(u_i, \bar{u}_i), (S, u_i), (S, \bar{u}_i)\} \right) \cup \left(\bigcup_{j=1}^m (T_j \cup \{(v_j, P)\}) \right) \cup \{(P, S)\}$$

Ce graphe est représenté en figure 17. Dans cette figure et pour faciliter la compréhension, on a volontairement distingué les sommets/littéraux u_i ou \bar{u}_i des sommets/littéraux a_j, b_j et c_j alors qu'ils sont confondus. Tous les sommets connectés entre eux par des arcs en pointillés bleus ne font qu'un. Par exemple, les sommets c_1, c_2 et u_n sont confondus pour la clause $C_1 = \{u_1, \bar{u}_2, u_n\}$.

Nous allons à présent établir un résultat important qui sera utilisé pour prouver que la transformation respecte bien l'équivalence entre la satisfaisabilité de la clause C_j et l'existence d'un 3-coloriage de la grappe associée.

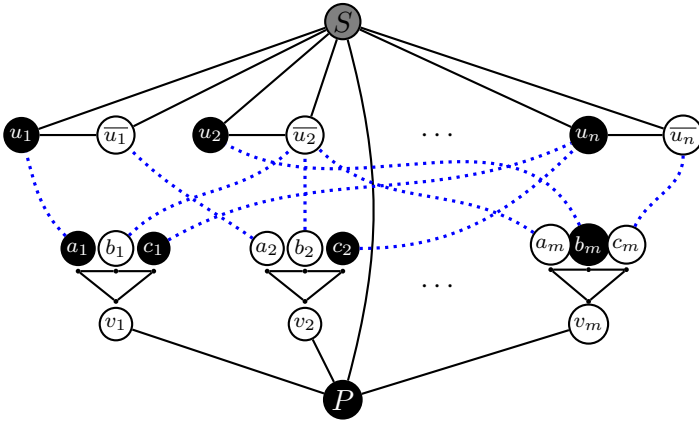


FIGURE 17. Instance de 3-COL.

Lemme 1. *Considérons la grappe liée à la clause C_j avec des sommets a_j , b_j et c_j noirs ou blancs.*

- (1) *Si les trois entrées sont noires, il existe un 3-coloriage de la grappe uniquement si la sortie v est noire.*
- (2) *Si les trois entrées ne sont pas identiquement noires, il existe un 3-coloriage de la grappe tel que la sortie v soit noire.*

Démonstration. (1) Supposons que les trois sommets a_j , b_j et c_j soient noirs. Les sommets k_j et m_j ne peuvent être que gris et blanc (ou réciproquement) car ils sont adjacents, et le sommet l_j est donc nécessairement noir. Les sommets z_j et l_j étant noirs, le même raisonnement prouve que o_j et n_j sont gris et blanc (ou réciproquement) et finalement v_j ne peut-être que noir.

(2) Il reste $2^3 - 1 = 7$ combinaisons de couleurs possibles pour les sommets a_j , b_j et c_j s'ils ne sont pas tous noirs. L'une de ces combinaisons a été traitée dans la figure 16, les autres sont laissées en exercice. \square

Montrons que si l'instance de 3-SAT est satisfaisable alors le graphe admet un 3 coloriage. Si l'instance de 3-SAT est satisfaisable, le sommet u_i est blanc et le sommet \bar{u}_i est noir ou réciproquement et ceci impose au sommet source d'être gris. D'autre part, chaque clause à trois littéraux étant satisfaite, l'un au moins des trois littéraux doit être vrai, i.e. l'un des trois sommets a_i

b_i et c_i est blanc. D'après le lemme, il est possible de colorier les sommets internes de la grappe afin que le sommet v_i soit blanc. Ceci impose au puit P d'être gris ou noir, mais comme il est connecté à la source S en gris, il ne reste plus que P noir.

Réciproquement, si l'on dispose d'un trois coloriage on peut supposer que P est noir et S est gris. En effet, il est évident que n'importe quelle permutation des trois couleurs d'un 3-coloriage est encore un 3-coloriage. Chaque littéral u et sa négation \bar{u} étant reliés avec S , l'un des deux est blanc et l'autre noir. Dans ce cas les sommets v_i sont gris ou blancs et ceci impose d'après la question précédente que l'un au moins des littéraux a_j , b_j et c_j est blanc (sans quoi v_j serait obligatoirement noir), autrement dit la clause associée est satisfaite et la table de vérité est fixée par les couleurs des littéraux.

Montrons pour finir que cette transformation est polynomiale. Comptons le nombre de sommets. Il y a n duos de littéraux, soit $2n$ sommets, 6 sommets pour chacune des m grappes, soit $6m$ sommets et la source S et le puit P . On a donc

$$|X| = 2n + 6m + 2.$$

Pour les arcs, on a $3n$ arcs pour connecter les n duos U_i et la source S à chaque littéral. Il faut $11m$ arcs pour connecter les sommets des m grappes et relier ces grappes aux littéraux des duos et au puit P et un dernier arc pour le relier à la source S . On a donc

$$|V| = 3n + 11m + 1$$

Avec un schéma d'encodage raisonnable, la construction d'un élément du graphe pour une variable booléenne ou une clause se fait en temps constant. La transformation globale est donc linéaire en n et m . \square

Exercice 25 Démontrez le point (2) du lemme pour les 6 combinaisons restantes de coloriage (en noir et blanc) des 3 sommets a_j , b_j et c_j .

Proposition 8. *Le problème XC de la couverture exacte est NP-complet.*

Démonstration. Montrons que $\text{XC} \in \text{NP}$. Soit (E, S) une instance de XC et soit $P \subseteq S$ le certificat. Il est aisé de vérifier que P est une partition (recouvrement exact) de l'ensemble E . On suppose, par exemple, que S est codée par une liste de listes d'éléments de E et que $P \subseteq S$ est codée

par la liste de tous les indices i tels que $S[i] \in P$. Pour chaque partie $S[i] \in P$, on parcourt ses éléments $x \in S[i]$ en s'assurant tout d'abord que $x \in E$, sinon P n'est pas un recouvrement exact, puis en éliminant x de E , i.e. $E \leftarrow E \setminus \{x\}$. La vérification s'achève une fois la dernière partie $S[i]$ parcourue et que $E = \emptyset$. Cet algorithme est en $O(|S| \cdot |E|)$.

Nous allons montrer que $3\text{-COL} \propto_P \text{XC}$. Notons $G = (X, V)$ une instance de 3-COL avec $n := |X|$ sommets et $m := |V|$ arcs, et (E, S) l'instance de XC que nous allons construire à partir de G de manière à ce que E admette une partition (couverture exacte) si et seulement si G admet un 3-coloriage. L'ensemble des couleurs est $C := \{R, V, B\}$ codant Rouge, Vert et Bleu.

Pour tout sommet $x \in X$, on désigne par

$$\mathcal{V}(x) := \{y \in X \mid \{x, y\} \in V\} \quad (17)$$

l'ensemble des sommets *voisins* de x . Notons que le graphe n'étant pas orienté, $y \in \mathcal{V}(x) \Leftrightarrow x \in \mathcal{V}(y)$. Pour chaque élément $x \in X$ et chacune des trois couleurs $c \in C$, on définit un ensemble S_x^c contenant $1 + |\mathcal{V}(x)|$ éléments :

$$S_x^R := \{x_R\} \cup \bigcup_{y \in \mathcal{V}(x)} \{y_x^R\},$$

$$S_x^V := \{x_V\} \cup \bigcup_{y \in \mathcal{V}(x)} \{y_x^V\},$$

$$S_x^B := \{x_B\} \cup \bigcup_{y \in \mathcal{V}(x)} \{y_x^B\}.$$

Par construction, X et les $3n$ ensembles S_x^c , $(x, c) \in X \times C$ sont deux-à-deux disjoints. Les trois ensembles S_x^c , $c \in C$ sont représentés dans la figure 18 pour le sommet $x = a$ de voisins $\mathcal{V}(a) = \{b, c\}$. L'ensemble E réunit tous les éléments de X ainsi que tous les éléments des S_x^c :

$$E := X \sqcup \left(\bigsqcup_{(x,c) \in X \times C} S_x^c \right) \quad (18)$$

L'ensemble S contient toutes les parties S_x^c ainsi que deux groupes de paires d'éléments de E . Le premier groupe est constitué de n sous-groupes de trois paires $\{x, x_R\}$, $\{x, x_V\}$, $\{x, x_B\}$ pour $x \in X$ matérialisées par les arcs en traitillés dans la figure 18 pour le sommet $x = a$. Le deuxième groupe contient les paires reliant l'élément y_x^c de la partie S_x^c à l'élément $x_{c'}^c$ de la

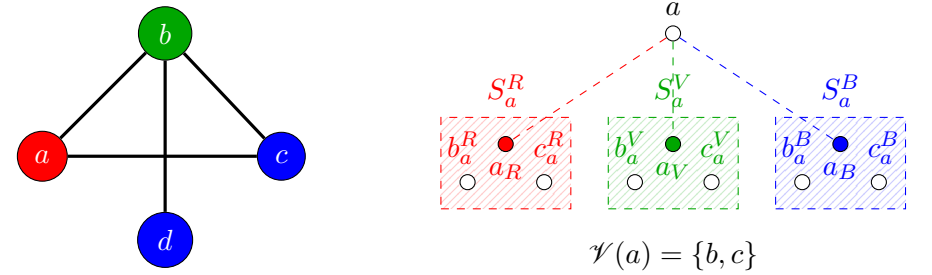


FIGURE 18. À gauche : une instance de 3-COL. À droite : les parties S_a^R , S_a^V , S_a^B , $\{a, a_R\}$, $\{a, a_V\}$ et $\{a, a_B\}$ de E dans S .

partie $S_y^{c'}$ pour tout $x \in X$ et tout couple de couleurs *distinctes* c et c' . On a donc

$$S := \bigcup_{(x,c) \in X \times C} \left(S_x^c \cup \underbrace{\{x, x_c\}}_{\text{groupe 1}} \cup \left(\bigcup_{(y,c') \in \mathcal{V}(x) \times C \setminus \{c\}} \underbrace{\{y_x^c, x_y^{c'}\}}_{\text{groupe 2}} \right) \right). \quad (19)$$

Pour alléger le graphique et les écritures, les éléments x_c sont matérialisés par un simple point de couleur c et les éléments y_x^c sont simplement notés y dans la couleur c , la référence à l'élément x étant évidente. Le schéma de la figure 19 montre l'instance de XC associée à l'instance de 3-COL considérée dans la figure 18.

Avant de prouver que l'instance E de XC admet une partition $P \subseteq S$ si et seulement si l'instance G de 3-COL admet un 3-coloriage, remarquons que :

- (1) Les ensembles S_x^c sont deux-à-deux disjoints.
- (2) Pour chaque $x \in X$, une paire $\{x, x_R\}$ ou $\{x, x_V\}$ ou $\{x, x_B\}$ exactement doit appartenir à la partition P .
- (3) Si $S_x^c \in S$ n'appartient pas à la partition P alors P doit contenir une seule des deux paires $\{y_x^c, x_y^{c'}\}$ avec $c' \in C \setminus \{c\}$ pour chaque $y \in \mathcal{V}(x)$.

Supposons que le graphe $G = (X, V)$ admette un 3-coloriage $\nu : X \rightarrow C$. Pour tout $x \in X$, on inclut dans la partition P :

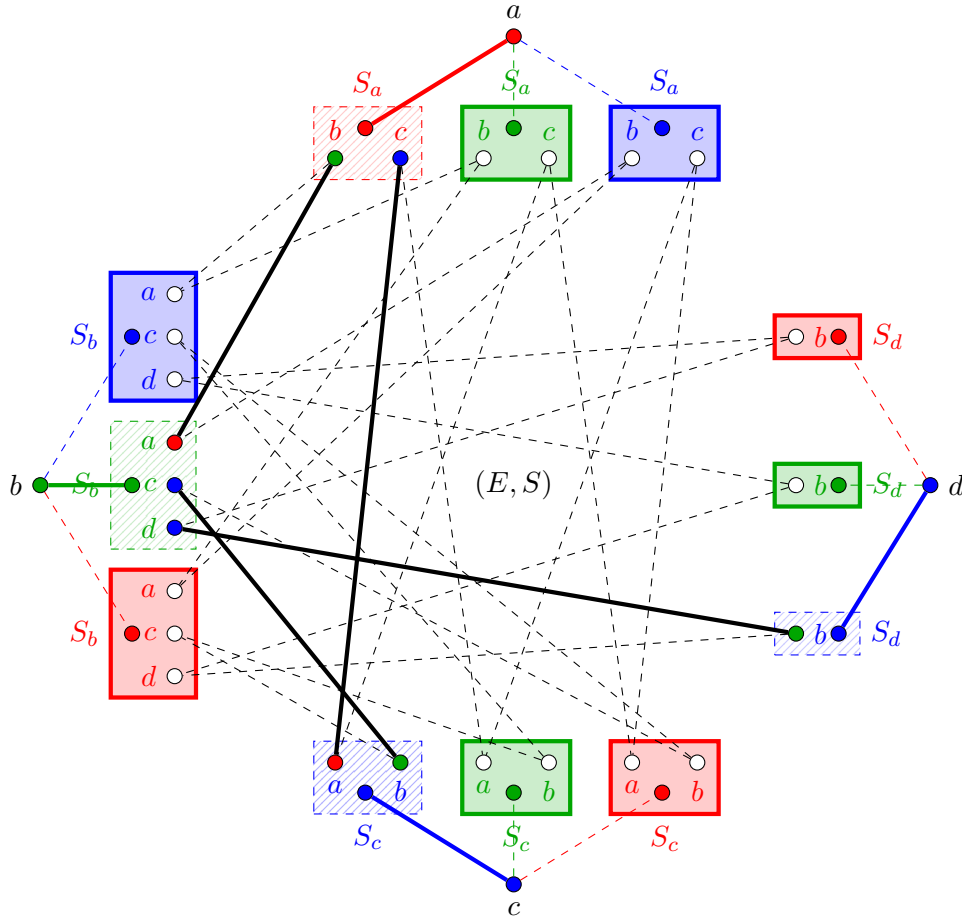


FIGURE 19. L'instance (E, S) associée au graphe $G = (X, V)$ et la **partition** associée au 3-coloriage de la figure 18.

- (1) Les deux parties S_x^c de couleurs *différentes* de $\nu(x)$, sur fond uni dans la figure 19 alors que celle de couleur $\nu(x)$ est sur fond hachuré.
- (2) La paire $\{x, x_{\nu(x)}\}$ recouvrant x et $x_{\nu(x)} \in S_x^{\nu(x)}$, en trait plein de couleur $\chi(x)$ dans la figure 19.

- (3) Il reste donc à recouvrir dans $S_x^{\nu(x)}$ les $|\mathcal{V}(x)|$ voisins $y_x^{\nu(x)}$ où $y \in \mathcal{V}(x)$, ce qui est réalisé par les paires $\{y_x^{\nu(x)}, x_y^{\nu(y)}\}$ en traits pleins noirs dans la figure 19 reconstituant ainsi le graphe G .

Ces parties recouvrent bien tous les éléments de E et sont deux-à-deux disjointes formant ainsi une partition de E .

Réciproquement, donnons nous une partition P de E et construisons un 3-coloriage de G . Chaque élément $x \in X$ est nécessairement recouvert par une seule des 3 paires $\{x, x_c\} \in P$ recouvrant par la même occasion l'élément x_c de la partie S_x^c . On fixe alors la couleur $\nu(x) := c$. Montrons que ν définit un 3-coloriage de G , i.e. que pour tout $(x, y) \in V$ alors $\nu(x) \neq \nu(y)$. Pour tout $x \in X$, les deux autres paires $\{x, x_{c'}\}$, $c' \neq \nu(x)$ ne pouvant plus être utilisées, les deux éléments $x_{c'}$ ne peuvent plus être recouverts que par les parties $S_x^{c'}$, $c' \neq \nu(x)$ qui restent les seules à les contenir et recouvrent du même coup les éléments $y_x^{c'}$, $c' \neq \nu(x)$ pour $y \in \mathcal{V}(x)$. Aucune paire connectée aux éléments de ces deux parties ne peut donc plus faire partie de P et les paires connectées à $S_x^{\nu(x)}$ ont des éléments de couleurs différentes par construction.

Exercice 26 On se donne un graphe $G = (X, V)$ non-orienté et sans boucle. On rappelle que le *degré* $d(x)$ d'un sommet $x \in X$ est le nombre de ses voisins, i.e. $d(x) := |\mathcal{V}(x)|$ où \mathcal{V} est défini en (17). Vérifiez que la somme des degrés des sommets est égale à $2|V|$, i.e.

$$\sum_{x \in X} d(x) = 2|V|. \quad (20)$$

Montrons que cette transformation est polynomiale. On a

$$\forall (x, c) \in X \times C \quad |S_x^c| = 1 + d(x). \quad (21)$$

Les ensembles X et S_x^c , $(x, c) \in X \times C$ formant une partition de E (cf. (18)), la formule de sommation nous donne

$$\begin{aligned} |E| &= |X| + \sum_{(x,c) \in X \times C} |S_x^c| \\ &= n + \sum_{x \in X} \sum_{c \in C} |S_x^c| \\ &= n + \sum_{x \in X} 3(1 + d(x)) \quad \text{d'après (21)} \\ &= 4n + 6m \quad \text{d'après (20)} \end{aligned}$$

Dans un premier temps, on dénombre $3n$ paires $\{x, x_c\}$ pour $(x, c) \in X \times C$. Dans un deuxième temps, on dénombre $2d(x)$ arcs issus de la partie S_x^c pour tout $(x, c) \in X \times C$, soit $d(x)$ paires, sachant que chaque arc est compté deux fois. On a donc au total

$$\begin{aligned} |S| &= 3n + \sum_{(x,c) \in X \times C} d(x) \\ &= 3n + \sum_{x \in X} \sum_{c \in C} d(x) \\ &= 3n + 3 \sum_{x \in X} d(x) \\ &= 3n + 6|V| \quad \text{d'après (20)} \\ &= 3n + 6m. \end{aligned}$$

La transformation est bien polynomiale en $O(n + m)$. \square

11. ANNEXE : LA MACHINE RAM

La machine RAM, acronyme de *Register Addressable Memory*, est un modèle abstrait de calcul introduit dans les années 1960-1970. Il avait pour objectif de fournir un outil d'étude des algorithmes plus proche des ordinateurs que ne l'étaient les modèles abstraits déjà proposés comme les fonctions récursives, le λ -calcul ou encore la machine de Turing élaborés à une époque où les ordinateurs n'existaient pas encore. Même si l'écriture d'algorithmes sur la machine RAM reste fastidieuse, son "langage" de programmation s'apparente aux langages d'assemblage beaucoup plus familiers des informaticiens.

Cette machine est constituée de 4 éléments principaux :

La machine RAM est constituée de 4 éléments :

- (1) Une *bande d'entrée* (E) segmentée en cases qui contiennent des entiers relatifs, accessible uniquement en *lecture* et de manière séquentielle.
- (2) Une *bande de sortie* (S) segmentée en cases qui contiennent des entiers relatifs, accessible uniquement en *écriture* et de manière séquentielle.
- (3) Une *mémoire* (R) indexée segmentée en *registres* R_0, R_1, \dots qui contiennent des entiers relatifs. Chaque registre est accessible directement via son *adresse* spécifiée dans le *registre de sélection mémoire* (SM).
- (4) Un *programme* (P), i.e. une séquence instructions codées par des couples (code opération, adresse) et indexée par le *compteur ordinal* (CO).

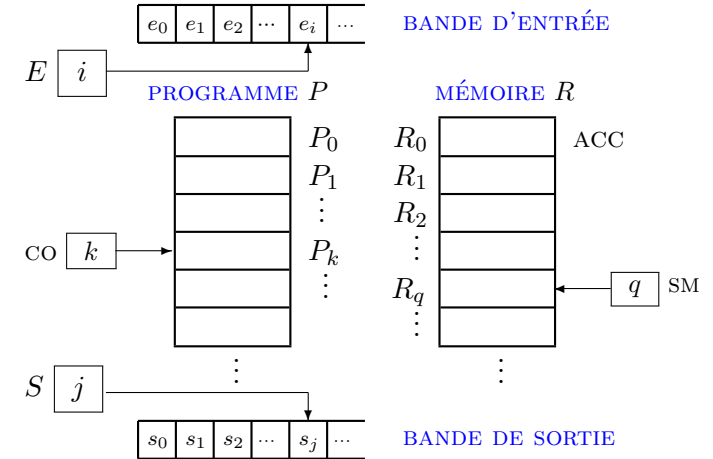


FIGURE 20. Description physique de la machine RAM

Le fonctionnement de cette machine est rudimentaire. On charge au préalable les instructions du programme dans le bloc programme (ce chargement est bien entendu virtuel, il s'agit d'un modèle abstrait) et on saisit (toujours virtuellement) les données à traiter sur la bande d'entrée. Ces

données sont encodées sous formes d'entiers suivant un schéma d'encodage arbitraire. Les instructions I_k du programme sont décodées séquentiellement. Pour cela, le compteur ordinal contient l'adresse du registre contenant l'instruction à décoder, initialement 0. Après qu'une instruction a été décodée, le compteur ordinal est incrémenté d'une unité pour passer à l'instruction suivante du programme, sauf si l'instruction est une *rupture de séquence*, auquel cas son rôle est justement de modifier la valeur du compteur ordinal. La machine arrête le cycle de décodage dès que l'instruction **STOP** a été exécutée.

Les données à traiter sont chargées dans la mémoire à l'aide de l'instruction de lecture **READ** qui copie le contenu de la cellule courante de la bande d'entrée dans le registre R_0 et les résultats des calculs sont renvoyés à l'utilisateur sur la bande de sortie à l'aide de l'instruction d'écriture **WRITE**.

Pour pouvoir réaliser des opérations complexes, on dispose de l'ensemble des registres R_i de la mémoire pour y stocker des résultats. Le registre R_0 a un statut particulier, c'est l'*accumulateur*. L'exécution d'une opération arithmétique remplace le contenu de l'accumulateur par le résultat de cette opération et le comportement de certaines instructions de rupture de séquence dépend du contenu de ce registre.

À l'exception des instructions **READ**, **WRITE** et **STOP**, toutes les instructions sont décomposées en deux blocs, le *code opération* et l'*adresse* : **CODOP adr**. Le code opération indique la nature de l'opération réalisée et l'adresse est soit celle d'une cellule de la mémoire, soit celle d'une cellule du programme dans le cas d'une rupture de séquence.

La table 3 regroupe les différentes instructions possibles et $R[n]$ représente la valeur contenue dans le registre R_n d'adresse n . Par souci de compacité, nous n'avons décliné que l'instruction d'addition **ADD** avec les préfixes **#n** et **@n** pour l'adressage absolu et indirect et que l'instruction de saut **JUMP** pour l'adressage indirect. On peut également utiliser l'adressage indirect pour l'incrément et le décrétement.

Exemple 21. Le programme ci-dessous calcule la factorielle du nombre entier inscrit sur la première cellule de la bande d'entrée. Le résultat des opérations arithmétiques se faisant toujours dans l'accumulateur, il est nécessaire de conserver les résultats intermédiaires (ici les produits successifs) dans

Type	Instruction	Signification
Entrées/Sorties	READ	$ACC \leftarrow e_i, i \leftarrow i + 1, CO \leftarrow CO+1$
	WRITE	$s_j \leftarrow ACC, j \leftarrow j + 1, CO \leftarrow CO+1$
Affectations	LOAD #n	$ACC \leftarrow n, CO \leftarrow CO+1, CO \leftarrow CO+1$
	LOAD n	$ACC \leftarrow R[n], CO \leftarrow CO+1$
	LOAD @n	$ACC \leftarrow R[R[n]], CO \leftarrow CO+1$
	STORE n	$R[n] \leftarrow ACC, CO \leftarrow CO+1$
	STORE @n	$R[R[n]] \leftarrow ACC, CO \leftarrow CO+1$
Arithmétiques	ADD #n	$ACC \leftarrow ACC + n, CO \leftarrow CO+1$
	ADD n	$ACC \leftarrow ACC + R[n], CO \leftarrow CO+1$
	ADD @n	$ACC \leftarrow ACC + R[R[n]], CO \leftarrow CO+1$
	SUB n	$ACC \leftarrow ACC - R[n], CO \leftarrow CO+1$
	MUL n	$ACC \leftarrow ACC \times R[n], CO \leftarrow CO+1$
	DIV n	$ACC \leftarrow ACC \div R[n], CO \leftarrow CO+1$
	MOD n	$ACC \leftarrow ACC \% R[n], CO \leftarrow CO+1$
	INC n	$R[n] \leftarrow R[n] + 1, CO \leftarrow CO+1$
	DEC n	$R[n] \leftarrow R[n] - 1, CO \leftarrow CO+1$
Ruptures de séquence	JUMP n	$CO \leftarrow n$
	JUMP @n	$CO \leftarrow R[n]$
	JUMZ n	$CO \leftarrow n$ si $ACC = 0$
	JUML n	$CO \leftarrow n$ si $ACC < 0$
	JUMG n	$CO \leftarrow n$ si $ACC > 0$
	STOP	Arrêt
Autres	NOP	$CO \leftarrow CO+1$

TABLE 3. Instructions de la *machine RAM*

un deuxième registre R_2 . Le registre R_1 contient quant à lui les valeurs successives $n, n - 1$ etc. Il faut remarquer que l'on fait toujours le produit sans se préoccuper de la valeur du registre R_1 après la décrémentation. On teste alors si le produit est nul. Si c'est le cas, on affiche le contenu du registre 2, sinon on range la valeur obtenue dans le registre R_2 .

0. READ	4. DEC 1	8. LOAD #1	12. STOP
1. JUMZ 8	5. MUL 1	9. STORE 2	
2. STORE 1	6. JUMZ 10	10. LOAD 2	
3. STORE 2	7. JUMP 3	11. WRITE	

Exercice 27 En vous inspirant de la définition 5 de la machine de Turing, proposez une définition de la machine RAM.

12. ANNEXE : QUELQUES RAPPELS DE LOGIQUE PROPOSITIONNELLE

On se donne un ensemble de *variables propositionnelles*, i.e. des éléments de l'ensemble $\{\mathcal{V}, \mathcal{F}\}$ et les trois connecteurs logiques qui connectent une ou deux variables propositionnelles et dont la valeur logique dépend des valeurs de vérité de(s) variable(s) connectée(s) :

- (1) Le connecteur unaire \neg de *négation* : si $u \in \{\mathcal{V}, \mathcal{F}\}$, on note sa négation $\neg u$ ou \bar{u} pour plus de lisibilité des formules.
- (2) Le connecteur binaire de *conjonction* \wedge : si $u, v \in \{\mathcal{V}, \mathcal{F}\}$, on note $u \wedge v$ la conjonction de u et v et on lit u *et* v .
- (3) Le connecteur binaire de *disjonction* \vee : si $u, v \in \{\mathcal{V}, \mathcal{F}\}$, on note $u \vee v$ la disjonction de u et v et on lit u *ou* v .

Leurs valeurs de vérité sont fixées par leurs *tables de vérité* :

u	\bar{u}
\mathcal{V}	\mathcal{F}
\mathcal{F}	\mathcal{V}

u	v	$u \wedge v$
\mathcal{F}	\mathcal{F}	\mathcal{F}
\mathcal{F}	\mathcal{V}	\mathcal{F}
\mathcal{V}	\mathcal{F}	\mathcal{F}
\mathcal{V}	\mathcal{V}	\mathcal{V}

u	v	$u \vee v$
\mathcal{F}	\mathcal{F}	\mathcal{F}
\mathcal{F}	\mathcal{V}	\mathcal{V}
\mathcal{V}	\mathcal{F}	\mathcal{V}
\mathcal{V}	\mathcal{V}	\mathcal{V}

TABLE 4. Tables de vérité des connecteurs logiques.

On peut combiner plusieurs variables propositionnelles à l'aide de ces connecteurs pour composer des expressions logiques :

Définition 22. Une *formule propositionnelle* ou *expression bien formée* est définie inductivement par l'une des règles de construction suivantes :

- (1) *formule* $:= x$ où x est l'une des variables propositionnelles
- (2) *formule* $:= \mathcal{V}$
- (3) *formule* $:= \mathcal{F}$
- (4) *formule* $:= \neg$ *formule*

$$(5) \text{ formule} := (\text{formule} \wedge \text{formule})$$

$$(6) \text{ formule} := (\text{formule} \vee \text{formule})$$

Définition 23. On appelle *formule atomique* ou *atome* toute formule réduite à une variable propositionnelle. On appelle *littéral* tout atome (*littéral positif*) ou *négation d'un atome* (*littéral négatif*).

Exemple 22. Si u_1 est une variable propositionnelle, u_1 ou \bar{u}_1 sont des littéraux, le premier est positif et le second négatif.

Définition 24. Si U est un ensemble de variables propositionnelles, on appelle *interprétation* toute application de $U \rightarrow \{\mathcal{V}, \mathcal{F}\}$.

Une interprétation I d'un ensemble de variables propositionnelles fixe *de facto* la valeur de vérité d'une formule propositionnelle P de ces variables que l'on notera $I(P)$ par abus de langage.

Définition 25. Deux formules F et G définies sur un ensemble de variables propositionnelles sont dites *logiquement équivalente*, ce que l'on note $F \equiv G$ si et seulement si elles ont même valeur de vérité pour toute interprétation I des variables propositionnelles.

Exemple 23. Soit F et G les formules :

$$F := \bar{u}_2 \wedge ((u_1 \vee u_3) \wedge (u_1 \vee \bar{u}_3))$$

$$G := u_1 \wedge \bar{u}_2$$

On a $F \equiv G$. Il suffit de faire la table de vérité des deux formules F et G pour les variables $\{u_1, u_2, u_3\}$ pour constater qu'elles ont la même valeur de vérité pour toutes les interprétations de ces variables. On peut également le montrer par le calcul :

$$\begin{aligned} \bar{u}_2 \wedge ((u_1 \wedge u_3) \vee (u_1 \wedge \bar{u}_3)) &\equiv \bar{u}_2 \wedge (u_1 \wedge (u_3 \vee \bar{u}_3)) \quad \text{distr. de } \wedge \text{ sur } \vee \\ &\equiv \bar{u}_2 \wedge (u_1 \wedge \mathcal{V}) \quad \text{tiers exclu} \\ &\equiv \bar{u}_2 \wedge u_1 \quad \mathcal{V} \text{ neutre de } \wedge \\ &\equiv u_1 \wedge \bar{u}_2 \quad \text{commutativité de } \wedge \end{aligned}$$

Proposition 9. Soit u et v deux variables propositionnelles. Alors

$$\overline{(u \vee v)} \equiv \bar{u} \wedge \bar{v} \quad (22)$$

$$\overline{(u \wedge v)} \equiv \bar{u} \vee \bar{v} \quad (23)$$

Ce sont les lois de De Morgan.

Définition 26. Une formule de la logique propositionnelle est dite **satisfaisable** s'il existe une interprétation I de ses variables propositionnelles telle que sa valeur de vérité soit \mathcal{V} .

Exemple 24. La formule propositionnelle précédente F est satisfaisable, par exemple avec l'interprétation $I(u_1) := \mathcal{V}$, $I(u_2) := \mathcal{F}$ et $I(u_3) := \mathcal{F}$.

Définition 27. On appelle **clause** toute formule propositionnelle qui est une disjonction de littéraux.

On supposera dans toute la suite et sans perdre en généralité, qu'une clause ne contient jamais plus de littéraux qu'il y a de variables propositionnelles. En effet, d'une part $(u \vee u) \equiv u$, ce qui permet de ne considérer que des clauses ne faisant jamais apparaître plusieurs fois le même littéral, et d'autre part $u \vee \bar{u} \equiv \mathcal{V}$, ce qui permet de ne considérer que des clauses qui ne contiennent pas un littéral et son opposé, sans quoi la clause est une tautologie et ne définit aucune contrainte.

Définition 28. On appelle **forme normale conjonctive** (resp. **forme normale disjonctive**) toute formule propositionnelle qui est une conjonction (resp. disjonction) de clauses.

Exemple 25. La formule $u_1 \vee \bar{u}_2 \vee u_3$ est une clause. La formule $(u_1 \vee \bar{u}_2 \vee u_3) \wedge (u_1 \vee \bar{u}_3)$ est sous forme normale conjonctive.

Par convention une clause est décrite simplement par l'ensemble de ses littéraux, la clause de l'exemple ci-dessus s'écrirait donc $\{u_1, \bar{u}_2, u_3\}$. On peut démontrer que toute formule propositionnelle est logiquement équivalente à une forme normale conjonctive. Il existe même une forme normale conjonctive dans laquelle toute clause fait apparaître chacune des variables booléennes à travers son littéral positif ou son littéral négatif, c'est la **forme normale conjonctive canonique**.

Remarque. La conversion d'une formule arbitraire en une forme normale conjonctive engendre parfois une croissance exponentielle du nombre de littéraux. On dispose néanmoins d'un algorithme linéaire, la **transformation de Tseitin**, qui fournit une forme normale conjonctive qui est satisfaisable si et seulement si la formule initiale est satisfaisable, mais au prix de variables supplémentaires.

13. ANNEXE : MINI LEXIQUE SUR LES GRAPHES

Définition 29. Un **graphe orienté** est un couple $G = (X, V)$ où X est un ensemble de **sommets** et $V \subseteq X \times X$ est l'ensemble des **arcs**.

Les sommets x et y sont appelés les **extrémités** de l'arc (x, y) . Le sommet x est appelé le **prédécesseur** de y et y le **successeur** de x . L'ensemble des arcs définit implicitement une **correspondance** $\Gamma : X \rightarrow X$ parfois notée Γ^+ et la correspondance réciproque Γ^- au lieu de Γ^{-1} . Un arc (x, x) est appelé une **boucle**.

Un graphe orienté $G = (X, V)$ hérite des propriétés de la relation binaire \mathcal{R} définie sur X par $x \mathcal{R} y$ si et seulement $(x, y) \in V$ (reflexivité, symétrie, transitivité, etc.). Si l'orientation n'a pas d'importance, l'ensemble V est défini comme un sous-ensemble de paires $\{x, y\}$ et de singletons $\{x\}$ de X , donc un sous-ensemble de $\mathcal{P}_2(X) \cup \mathcal{P}_1(X)$. On parle alors d'**arête** au lieu d'arc et le graphe est dit **non orienté**. On appelle **complémentaire** d'un graphe $G = (X, V)$ le graphe $\bar{G} := (X, \bar{V})$ où \bar{V} est le complémentaire ensembliste de V dans $X \times X$ (ses arcs sont exactement ceux qui ne sont pas dans G). On appelle **graphe complet** ou **clique**, tout graphe $G = (X, V)$ qui contient tous les arcs, i.e. tel que $V = X \times X$.

Dans un graphe orienté (resp. non orienté), une séquence de sommets telle que chaque couple de sommets consécutifs constitue un arc (resp. une arête) du graphe est appelée un **chemin** (resp. une **chaîne**); et si la séquence commence et se termine par le même sommet, on parle de **circuit** (resp. de **cycle**). Un chemin (resp. une chaîne) qui passe exactement une fois par chaque arc (resp. une arête) est appelé **chemin eulérien** (resp. une **chaîne eulérienne**) et un chemin (resp. une chaîne) qui passe exactement une fois par chaque sommet est appelé **chemin hamiltonien** (resp. **chaîne hamiltonienne**).

Remarque. Les termes chaîne et cycle peuvent s'appliquer à une séquence de sommets d'un graphe orienté si l'on ne tient pas compte de l'orientation des arcs.

Un graphe est dit *connexe* si pour tout couple de sommets du graphe, il existe au moins une chaîne qui les relie. La *longueur* d'un chemin (resp. d'une chaîne) est le nombre de ses arcs (resp. de ses arêtes). Un sous-graphe d'un graphe $G = (X, V)$ (sa restriction à un sous-ensemble Y de X) est appelé *composante connexe* de G s'il est connexe et maximal : aucune chaîne ne relie un sommet de $X \setminus Y$ à un sommet de Y . Un sous-graphe est appelé *composante fortement connexe* si pour tout couple de sommet (x, y) il existe un chemin qui relie x à y . Un graphe non-orienté connexe est donc fortement connexe.

BIBLIOGRAPHIE

- (1) *Computers and Intractability. A guide to the Theory of NP-Completeness*. Michael. R. Garey, David S. Johnson. WH. Freeman and Company, 1979.
- (2) *Computational complexity. A modern approach*. S. Arora, B. Barak, Cambridge University Press, 2009.
- (3) *Introduction to the Theory of Computation*. M. Spiser, Langage Learning, 2018.
- (4) *Calculabilité et décidabilité, une introduction*. J. Autebert, Masson, 1992.
- (5) *Computability : Computable Functions Logic and the Foundations of Mathematics*, R.L. Carnielli, W.A. Epstein, Chapman and Hall, 1990.

INDEX

- algorithme, 2, 7
 - de Tarjan, 23
 - DPLL, 18
 - entrée, 3
 - sortie, 3
- algorithmique, 2
- alphabet, 7
- arbre
 - lexicographique, 3
- atome, 35
- automates finis déterministes, 10

- bande, 6

- calcul, 2
- calculable, 4
- calculer, 13
- cardinal, 4
- cases d'une bande, 6
- certificat, 11
- chaîne
 - eulérienne, 36
 - hamiltonienne, 36
- chaînette, 24
- chemin
 - eulérien, 36
 - hamiltonien, 36
- Church-Turing
 - thèse, 3
- classe
 - NP, 12
 - NP-complet, 14
 - NP-difficile, 14
 - P, 10
- clause, 35
 - unitaire, 18
- compteur ordinal, 33
- configuration, 16
- connecteur
 - de conjonction, 34
 - de disjonction, 34
 - de négation, 34

- et, 34
- ou, 34
- correspondance, 36

- degré, 32
- diagonal, 5
- direction, 7

- effacement, 7
- efficace, 9
- ensemble
 - au plus dénombrable, 5
 - de sommets indépendants, 21
 - dénombrable, 5
 - fini, 4
 - infini, 5
- ensembles
 - équipotents, 3
- expression bien formée, 35
- extrémités
 - d'un arc, 36

- factorisable, 12
- fonction, 3
 - de complexité en temps, 8
 - de transition, 7
- forme normale
 - conjonctive, 35
 - canonique, 36
 - disjonctive, 35
- formule
 - atomique, 35
 - propositionnelle, 35
 - satisfaisable, 35

- graphe
 - arc, 36
 - arête, 36
 - boucle, 36
 - chaîne, 36
 - chemin, 36
 - circuit, 36
 - clique, 36

- complet, 25, 36
- complémentaire, 36
- composante
 - connexe, 36
 - fortement connexe, 36
- connexe, 3, 36
- cycle, 36
- non orienté, 36
- orienté, 36
- sommet, 36
- instance
 - négative, 10
 - positive, 10
- instruction
 - adresse, 33
 - code opération, 33
- langage
 - NP, 12
 - NP-complet, 14
 - NP-difficile, 14
 - P, 10
 - transformation, 13
 - polynomiale, 13
- littéral, 35
 - négatif, 35
 - positif, 35
 - pur, 18
- lois de De Morgan, 35
- longueur
 - d'un chemin, 36
- machine RAM
 - accumulateur, 33
 - adresse, 33
 - bande d'entrée, 33
 - bande de sortie, 33
 - mémoire, 33
 - programme, 33
 - registres, 33
- machine de Turing, 7
- mots, 3
- oracle, 11
- preuve, 11
- principe d'abstraction, 2
- problème, 9
 - 3-SAT, 20
 - de décision, 9
 - de l'arrêt d'un programme, 5
 - de l'ensemble de sommets indépendants, 21
 - de la clique, 21
 - de la couverture exacte, 22
 - de la décidabilité, 2
 - de la factorisation, 12
 - de la partition, 22
 - de la satisfaisabilité des clauses, 14
 - à 3 littéraux, 20
 - données, 9
 - du k -coloriage, 20
 - du circuit hamiltonien, 21
 - du mariage tri-dimensionnel, 20
 - du recouvrement par sommets, 21
 - du Sudoku, 16
 - du voyageur de commerce, 9, 22
 - dual, 14
 - décidable, 4
 - faisable, 9
 - infaisable, 9
 - instance, 9
 - intractable, 9
 - non-déterministe polynomial, 12
 - NP-complet, 14
 - NP-difficile, 14
 - P, 10
 - paramètres, 9
 - polynomial, 10
 - SAT, 14
 - solution, 9
- programme, 6, 7
- prédicat, 2
 - collectivisant, 2
- Register Addressable Memory, 33
- Registre de sélection mémoire, 33
- rupture de séquence, 33
- règle, 7

- schéma d'encodage, 3
- sommet
 - prédécesseur, 36
 - successeur, 36
 - voisin, 30
- symbole, 6
 - blanc, 7
 - d'écriture, 7
 - lu, 7
- table de vérité, 34
- théorie
 - de l'approximation polynomiale, 1
 - de la calculabilité, 1
 - de la complexité, 1
- théorèmes d'incomplétude de Gödel, 2
- transformation
 - de Tseitin, 36
 - polynomiale, 13
- tête de lecture/écriture, 6
- variable
 - interprétation, 35
 - propositionnelle, 34
- équivalence
 - logique, 35
 - polynomiale, 13
- état
 - courant, 7
 - d'acceptation, 10
 - d'une machine de Turing, 7
 - de refus, 10
 - initial, 7