Initiation à la théorie de la complexité

(Version 3.0 du 14 février 2025)

Jean-Pierre Zanotti

Table des matières

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 complexité [1]

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 17ème 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 19ème siècle.

Le mathématicien allemand David Hilbert présente au début du 20ème 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 Russel, 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 exclu [2] 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

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

sont 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 calcul* 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, connu sous le nom de *thèse de Church-Turing* a largement été validé depuis. Un modèle de calcul B qui a un pouvoir d'expression au moins aussi important qu'un modèle A, i.e. B est capable de simuler A, est dit *A-complet*.

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. Une instance d'un problème qui est traitée par un algorithme est *codée* par un mot d'un langage Σ^* et le résultat du calcul est lui aussi *codé* par un mot d'un langage Γ^* (souvent identique à Σ^*). Ces processus de traduction et d'interprétation sont qualifiés respectivement de *schéma d'encodage* et de *schéma de décodage*.

Exemple 1. (1) Le triplet $(255, 255, 0)$ de $[\![0, 255]\!]^3$ 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 $[\![0, 255]\!]^4$ 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 donc interprété en le *décodant*.

Figure 1. Schéma fonctionnel d'un modèle algorithme.

La similitude de ce schéma avec la description symbolique d'une *fonction* mathématique d'ensemble de départ X et d'arrivée Y

$$f : X \longrightarrow Y$$
$$x \longmapsto y$$

où x est l'entrée et y la sortie saute aux yeux. 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, i.e. $\Sigma^* = \Gamma^* = \mathbb{N}$.

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 nœud numéro 24 est associé au mot *bac* et le nœud numéro 10 est associé au mot *ca*. □

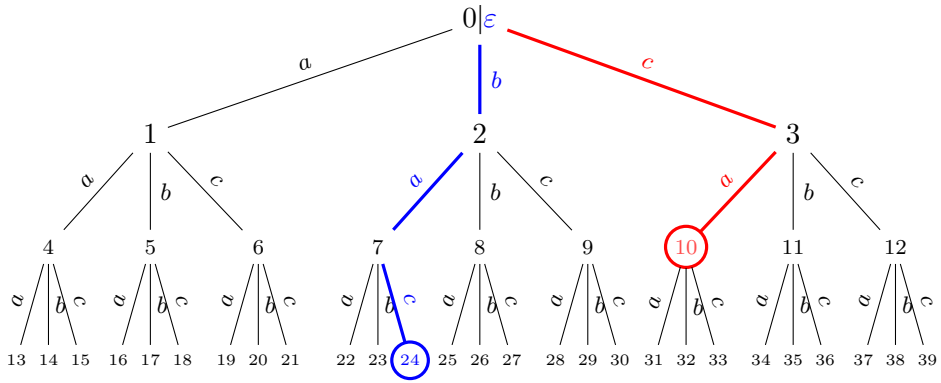


Figure 2. Les 4 premiers niveaux de l'arbre lexicographique du langage $\{a, b, c\}^*$ et l'indexation des mots.

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 $f : \mathbb{N} \to \mathbb{N}$ 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    ; ACC ← k
1 | WRITE      ; S[j++] ← k
2 | STOP       ; ARRET
```

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       ; ACC ← E[i++]
1 | MOD #2     ; ACC ← ACC % 2
2 | JUMZ 5     ; SI (ACC = 0) SAUTER A INSTRUCTION #5
3 | LOAD #0    ; ACC ← 0
4 | JUMP 6     ; SAUTER A INSTRUCTION #6
5 | LOAD #1    ; ACC ← 1
6 | WRITE      ; S[j++] ← ACC
7 | STOP       ; ARRET
```

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 $X \approx [\![1, n]\!]$. 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 $x : [\![1, n]\!] \to X$ 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 $x : \mathbb{N} \to X$ et on a $X = \{x_i \mid i \in \mathbb{N}\}$.

Théorème 2. *Il existe des fonctions $f : \mathbb{N} \to \mathbb{N}$ 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.
(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 $f : \mathbb{N} \to \mathbb{N}$ dont les images diffèrent des valeurs présentes sur la diagonale de cette table :

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

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 :

	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	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	⋯	$f_3(k)$	⋯	$f_3(n)$	⋯
⋮	⋯	⋯	⋯	⋯	⋱	⋯	⋯	⋯	⋯
f_k	$f_k(0)$	$f_k(1)$	$f_k(2)$	$f_k(3)$	⋯	$f_k(k)$	⋯	$f_k(n)$	⋯
⋮	⋯	⋯	⋯	⋯	⋯	⋯	⋱	⋯	⋯
⋮	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋱	⋯
⋮	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋱

Table 1. Images des applications de $\mathbb{N} \to \mathbb{N}$.

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

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. □

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 l'un des modèles les plus simples que l'on connaisse et qui satisfait aux critères informels mais universels qui caractérisent un algorithme (déterminisme, discrétion, finitude, généralité, reproductibilité, etc.) Il existe d'autres modèles encore plus simple avec le même pouvoir d'expression, comme le modèle de *la règle* 110 ou encore le modèle du *système d'étiquettage cyclique*. Ce dernier a été proposé en 2004 par Matthew Cook, précisément pour démontrer la *conjecture de Wolfram* affirmant que le modèle de la règle 110 est *Turing-complet*.

Remarque. Le modèle des *automates finis déterministes* étudié en théorie des langages est moins versatile que le modèle de la machine de Turing, et ne permet pas, entre autres, de reconnaître les mots binaires de la forme 0^n1^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 reconversion 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).

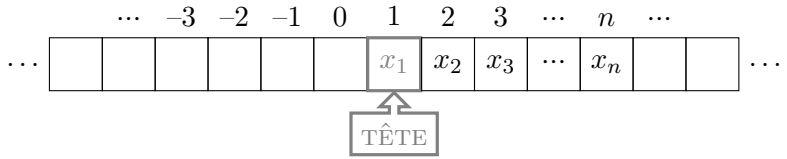


Figure 3. Schématisation d'une machine de Turing.

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 $x = x_1x_2\dots x_n \in (\Sigma \cup \{\square\})^*$ 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 $1\,\square\,10$ 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 $x_1 x_2 \dots x_n$ 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

$$\delta(p, s) = (e, d, q). \tag{2}$$

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, on retrouve le modèle des automates finis déterministes.

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 ?

Définition 6. *Soit x et y deux mots de longueur n et m définis sur deux alphabets finis Σ_1 et Σ_2 respectivement. On dit qu'une machine de Turing calcule y pour x si en initialisant la bande avec x :*

$$B[i] = \begin{cases} x_i & \text{si } i \in [\![1,n]\!], \\ \square & \text{sinon.} \end{cases}$$

l'exécution du programme s'arrête avec la bande qui contient y :

$$\exists k \in \mathbb{N}\ B[k+i] = \begin{cases} y_i & \text{si } i \in [\![1,m]\!], \\ \square & \text{sinon.} \end{cases}$$

Définition 7. *Soit Σ un alphabet fini. Une fonction $f : \Sigma_1^* \to \Sigma_2^*$ est dite Turing-calculable s'il existe une machine de Turing T qui calcule $f(x)$ pour x et ceci pour tout $x \in \mathscr{D}_f$.*

Exercice 6 Montrez que la fonction $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ définie par $n \mapsto k$ est Turing-calculable. On supposera que $\Sigma := \{0,1\}$.

Exercice † 8 Démontrez que l'addition de $\mathbb{N}\times\mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{Z}/4\mathbb{Z}$ définie par $n \mapsto n \pmod 4$ est Turing-calculable. On supposera que $\Sigma := \{0,1\}$.

Exercice 10 Montrez que si $f : \Sigma_1^* \to \Sigma_2^*$ et $g : \Sigma_2^* \to \Sigma_3^*$ sont deux fonctions Turing-calculables alors la fonction $g \circ f$ est Turing-calculable.

Exercice † 11 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 8. *On appelle fonction de complexité en temps d'un algorithme de la machine de Turing, la fonction $T : \mathbb{N} \to \mathbb{N}$ définie par*

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

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

Exercice 12 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 13 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 $O(T(n).\log(T(n))$ sur la machine RAM (resp. $O(T(n)^3)$) sur la machine de Turing).*

Comme on peut le constater, l'inflation peut être conséquente, par exemple, la simulation d'une machine RAM par une machine de Turing peut augmenter le coût d'un algorithme de manière cubique. Cependant, dans les deux cas, la complexité de l'algorithme reste polynomiale sur le simulateur si sa fonction de complexité T était polynomiale dans le modèle simulé. 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 9. *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 10. *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 $w : V \to \mathbb{N}$ (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

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

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 14 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 15 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.

Une instance I d'un problème de décision Π est donc encodée par le schéma d'encodage e en un mot $x = x_1 x_2 \dots x_n$ de Σ^* dont les symboles sont rangés sur la bande d'entrée (cf. figure 3). On associe alors au problème Π le *langage*

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

Définition 11. *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 12. *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

$$e(35) := \underline{1}10001.$$

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) :

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

Nous allons montrer que 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

$$\square^\infty x_1x_2 \dots x_n \square^\infty.$$

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 vérifier la case d'indice n.*
- $q_1, 1 : 1, \rightarrow, q_N$ // *Si $B[2] = 1$, alors rejet.*
- $q_2, 0 : 0, \rightarrow, q_2$ // *Tant que $B[i] \neq \square$*
- $q_2, 1 : 1, \rightarrow, q_2$ // $\quad i \leftarrow i + 1$
- $q_2, \square : \square, \leftarrow, q_3$ // *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 d'indice 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 opérandes 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 **N**on-déterministe **P**olynomial *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*. Il existe plusieurs manières équivalentes de définir une machine de Turing non-déterministe, celle que nous avons choisie est censée être plus facile à interpréter.

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, 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 $-i$ qui doit permettre de vérifier que x est l'encodage d'une instance positive du problème le cas échéant. Une case vierge en position 0 sépare donc les deux mots (cf. figure 4).

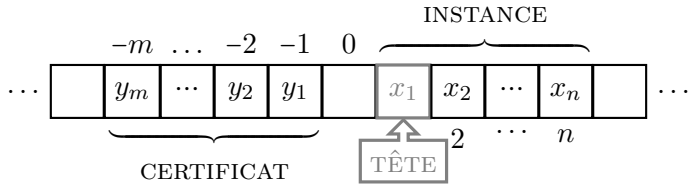


Figure 4. Machine de Turing non-déterministe.

Le programme écrit pour une telle machine peut alors se contenter de *vérifier* que x est une instance positive en utilisant le certificat y fourni par l'oracle. Plus précisément, si la preuve y que le mot x appartient au langage L est valide, alors la machine s'arrête dans l'état q_Y et reconnaît le mot x. Si le mot x n'appartient pas au langage, aucun certificat y ne permet de le reconnaître.

Définition 13. *Un langage L est dit* non-déterministe polynomial *s'il existe une machine de Turing non-déterministe qui reconnaît $x \in \Sigma^*$ en temps polynomial en $n := |x|$. La* classe NP *est l'ensemble des langages non-déterministes polynomiaux.*

Définition 14. *On dit qu'un problème* Π *est* non-déterministe polynomial *ou dans la classe NP si et seulement si $L[\Pi, e] \in NP$ pour un schéma d'encodage e.*

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. L'idée qui sous-tendait cette présentation, était que cette machine modélisait le fonctionnement (supposé) non-déterministe du cerveau 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, ce modèle n'a pas vocation à être utile *en pratique* car aucun oracle ne viendra nous souffler une preuve.

De manière plus prosaïque, si le mot $x \in L$, alors il doit exister au moins un certificat y tel que la machine reconnaitra x en temps polynomial. La difficulté pour montrer qu'un problème est dans la classe NP, ne consiste pourtant pas à chercher un certificat — c'est *précisément* l'obstacle que l'on n'arrive pas à surmonter pour répondre à un problème de décision en temps polynomial — mais à déterminer sa *nature*, sa *forme*. L'exemple suivant va permettre de comprendre ce que l'on entend par "nature".

Un entier N est dit *décomposable* 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 DÉCOMPOSABLE [entier décomposable]
Instance : Un entier naturel N.
Question : Peut-on factoriser N ?

Théorème 5. *Le problème DÉCOMPOSABLE appartient à la classe NP.*

Démonstration. Notons $e(N) := x_1x_2\dots x_n$ l'encodage binaire de l'entier naturel

$$N = \sum_{i=1}^{n} x_i 2^{i-1}$$

et soit $y := e(A,B)$ l'encodage d'un couple (A,B) d'entiers naturels, i.e. leurs écritures binaires séparées par un blanc $\square$, 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 $O(\log A \log B)$ puisque le nombre de chiffres d'un entier N dans sa représentation en base b est égal à $\lfloor \log_b N \rfloor + 1$. Il ne reste qu'à comparer les chiffres de ce produit avec ceux de N, ce qui se fait en $O(\min(\log A + \log B, \log N))$. La complexité de l'algorithme est donc polynomiale. □

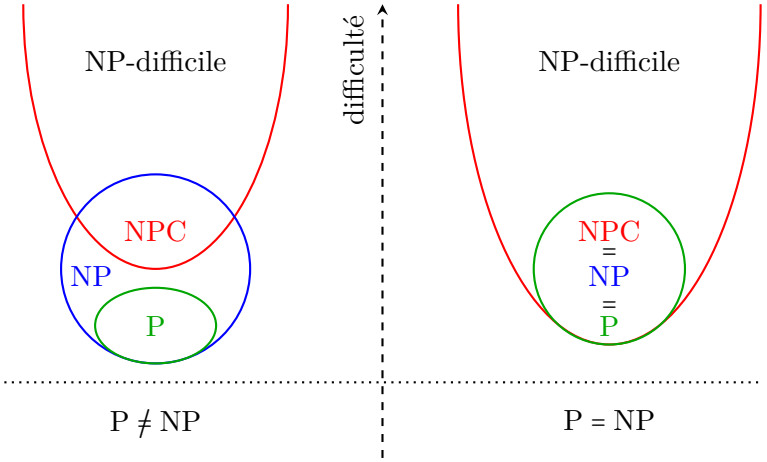


Figure 5. Classes de complexité.

Le certificat utilisé dans cette preuve est donc un couple d'entiers (A,B) et nous avons pu prouver qu'un tel certificat peut être vérifié en temps polynomial sans pour autant être capable de trouver les valeurs de A et B.

Théorème 6. *$P \subseteq NP$.*

Démonstration. Si $L \in \mathrm{P}$, 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 le certificat y pour accepter x, puisque l'on est capable de prouver le résultat grâce à l'algorithme déterministe. □

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

Conjecture. *$P \neq NP$.*

8. La classe NP-complet

Pour mesurer la difficulté d'un problème de décision, on va définir une relation binaire $\propto$ sur l'ensemble $\mathscr{L}$ 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 $\tau : \Sigma_1^* \to \Sigma_2^*$ Turing-calculable telle que*

$$x \in L_1 \Leftrightarrow \tau(x) \in L_2. \tag{7}$$

Si la fonction τ est calculable en temps polynomial, la transformation est dite polynomiale.

On définit 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. Si cette transformation est polynomiale, on note alors $L_1 \propto_P L_2$. Si Π_1 et Π_2 sont deux problèmes de décision, on note $\Pi_1 \propto \Pi_2$ (resp. $\Pi_1 \propto_P \Pi_2$) si et seulement si $L[\Pi_1, e_1] \propto L[\Pi_2, e_2]$ (resp. $L[\Pi_1, e_1] \propto_P L[\Pi_2, e_2]$) 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. Ce même programme résout indirectement le problème Π_1,

il suffit de traduire les mots de L_1 en mots de L_2 au préalable, autrement dit *qui peut le plus, peut le moins.*

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. □

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 $\Pi_1 \neq \Pi_2$. □

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

$$(\Pi_1 \propto_P \Pi_2) \ \wedge \ (\Pi_2 \propto_P \Pi_1) \tag{8}$$

ce qui définit une relation binaire ∞_P sur les problèmes de décision.

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

Exercice 16 Démontrez la proposition 2.

Si $\mathscr{D}$ désigne l'ensemble des problèmes de décision, les différents représentants d'une classe d'équivalence de l'ensemble quotient $\mathscr{D}/\infty_P$ sont des problèmes de même difficulté. On peut alors équiper l'ensemble quotient $\mathscr{D}/\infty_P$ de la relation $\propto_P$ 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*

$$\forall L' \in NP \quad L' \propto_P L. \tag{9}$$

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/∞_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 $\overline{\Pi}$ dont les instances sont celles de Π et dont la question est la négation de celle du problème Π.*

Exemple 6. 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 DÉCOMPOSABLE.

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

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

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

9. Le théorème de Cook

On trouvera en annexe (cf. section 13), 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 $I : U \to \{\mathscr{V}, \mathscr{F}\}$ des variables de U qui satisfait

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

Exemple 7. On considère l'ensemble des variables $U = \{u_1, u_2, u_3\}$ et l'ensemble des clauses $C = \{\{\overline{u}_1, u_2, u_3\}, \{u_1, \overline{u}_3\}\}$. Il y a trois variables et un ensemble de deux clauses à trois et deux littéraux respectivement. L'interprétation $u_1 \leftarrow u_2 \leftarrow \mathscr{V}$ et $u_3 \leftarrow \mathscr{V}$ (entre autres) montre que la formule

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

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 $L[\mathsf{SAT}, e]$ appartient à la classe des langages NP puis que tous les langages $L \in \mathrm{NP}$ se transforment polynomialement en $L[\mathsf{SAT}, e]$. 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

$$\forall \Pi \in \mathrm{NP} \quad \Pi \propto_P \mathsf{SAT}.$$

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 $\Pi \in \mathrm{NP}$ 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 $\tau : \Sigma^* \to \mathscr{C}$ de l'ensemble des mots sur l'alphabet Σ qui codent les instances du problème Π vers l'ensemble $\mathscr{C}$ des formules propositionnelles qui sont des conjonctions de clauses telle que

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

Nous conclurons 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 8. Une grille de Sudoku est une matrice 9×9 d'entiers de l'intervalle $[\![1,9]\!]$ 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 $[\![1,n^2]\!]$. 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 $[\![0,n^2]\!]$ (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 $(l,c,k)\in[\![1,9]\!]^3$ 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

$$\{S[1,1,1],S[1,1,2],\ldots,S[1,1,9]\} \tag{11}$$

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 $(i,j)\in[\![1,9]\!]^2,\ i<j$ (il y en a 36 au total) la clause

$$\{\overline{S[1,1,i]},\overline{S[1,1,j]}\} \tag{12}$$

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 $\mathfrak{S}_9$. Vérifiez que sans les clauses qui codent les indices, il y a 11 745 clauses.

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 $\Sigma=\{s_0,s_1,\ldots,s_\nu\}$ 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 $Q=\{q_0,q_1,\ldots,q_r\}$ 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 $\chi_t := (q, i, B) \in Q \times \mathbb{Z} \times \Sigma_\square^{\mathbb{Z}}$ où q désigne l'état de la machine, i la position de la tête de lecture/écriture et B le contenu de la bande, après t transitions.*

Exemple 9. La configuration initiale (i.e. à l'instant $t = 0$) de la machine non-déterministe polynomiale qui reconnaît $L[\Pi, e]$ est

$$\chi_0 = (q_0, 1, B)$$

où le contenu de la bande est défini par l'application :

$$B[i] := \begin{cases} x_i & \text{si } i \in [\![1, n]\!], \\ y_{-i} & \text{si } i \in [\![-m, -1]\!], \\ \square & \text{sinon.} \end{cases}$$

et $x = x_1 x_2 \dots x_n$ est l'encodage $e(I)$ d'une instance I du problème de décision Π et le mot $y = y_1 y_2 \dots y_m$ est le certificat.

La machine T étant de complexité 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 $[\![-p(n), p(n)+1]\!]$ de $\mathbb{Z}$ (cf. figure 7).

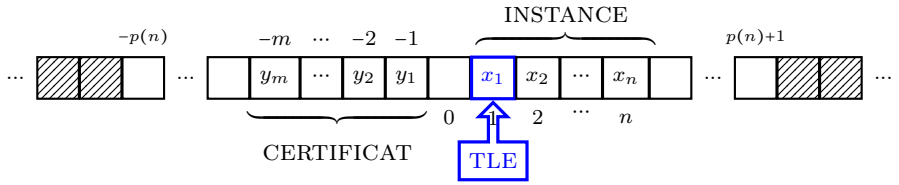


Figure 7. Zone utile de la bande.

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

$$I_T := [\![0, p(n)]\!] \text{ (instants)} \qquad I_Q := [\![0, r]\!] \text{ (états)}$$
$$I_P := [\![-p(n), p(n)+1]\!] \text{ (positions)} \qquad I_S := [\![0, \nu]\!] \text{ (symboles)}$$

Les variables sont partitionnées en 4 classes avec les interprétations :

(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, $B[i] = s_j$.

Nous supposerons que si la machine s'arrête à un instant $t \leq p(n)$, 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 ($\mathscr{V}$) ou éteintes ($\mathscr{F}$), 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 est active ou non.

Comme pour le Sudoku, la satisfaisabilité de l'ensemble C des clauses que nous allons construire doit assurer que ce modèle décrit fidèlement le fonctionnement d'une machine de Turing non-déterministe en général, et spécifiquement celui de l'exécution de la machine T qui reconnaît le langage $L[\Pi, e]$.

Nous allons répartir les clauses de C en 6 groupes. Les trois premiers assurent la cohérence du modèle par rapport au fonctionnement d'une machine de Turing non-déterminste quelconque, les trois derniers spécifiquement celui de la machine T 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 : le changement de configuration χ_t en χ_{t+1} entre deux instants t et $t+1$ de la machine respecte la fonction de transition δ.

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

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

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

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

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

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

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 $s_{j'}$ à l'instant t.

Clauses du groupe G_4. Initialement la machine est dans l'état q_0, sa tête de lecture/écriture est en position 1 et la bande contient $x = s_{k_1} s_{k_2} \ldots s_{k_n}$ dans la partie positive et est vierge ensuite :

$$\{Q[0,0]\},\ \{P[0,1]\},\ \{S[0,1,k_1]\},$$
$$\{S[0,2,k_2]\},\ \ldots,\ \{S[0,n,k_n]\},$$
$$\{S[0,n+1,0]\},\ \ldots,\ \{S[0,p(n)+1,0]\}.$$

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, aucune des autres cases de la bande que celle où se situe la tête de lecture/écriture n'aura changé de valeur à l'instant suivant $t+1$. Autrement dit si la tête de lecture/écriture est dans une autre position que i et que cette case contient le symbole s_j alors elle contient encore ce symbole à l'instant suivant $t+1$:

$$\overline{P[t,i]} \wedge S[t,i,j] \Rightarrow S[t+1,i,j].$$

Cette formule est logiquement équivalente à la clause

$$\left\{P[t,i], \overline{S[t,i,j]}, S[t+1,i,j]\right\} \tag{13}$$

On génère les clauses de la forme (13) pour tous les triplets (t,i,j) de l'ensemble $I_T \times I_P \times I_S$.

Le second sous-groupe de clauses assure qu'entre l'instant t et $t+1$, la configuration χ_t de la machine est passée à la configuration χ_{t+1} conformément à la fonction de transition δ, à savoir :

(1) le changement d'état est correct,
(2) la tête de lecture/écriture est à la bonne position,
(3) le symbole remplacé sur la bande est correct.

Pour simplifier les écritures qui vont suivre, on recode les déplacements $\leftarrow$ et $\rightarrow$ de la tête de lecture/écriture par -1 et $+1$ respectivement. 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. $k \notin \{1,2\}$, alors les entiers k', j' et $\Delta \in \{-1,0,1\}$ satisfont

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

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 $k = k'$, autrement dit la suite des configurations $(\chi_t)_{t\in\mathbb{N}}$ est constante à partir d'un certain rang majoré par $p(n)$).

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 et contient le symbole s_j, alors à l'instant $t+1$:

(1) son nouvel état est $q_{k'}$:

$$Q[t,k] \wedge P[t,i] \wedge S[t,i,j] \Rightarrow Q[t+1,k'].$$

formule logiquement équivalente à la clause

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

(2) la tête de lecture/écriture est à la position $i+\Delta$:

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

(3) la case i contient le symbole $s_{j'}$:

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

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

$$U := \bigcup_{t\in I_T}\left[\left(\bigcup_{i\in I_Q}\{Q[t,i]\}\right)\cup\left(\bigcup_{i\in I_P}\{P[t,i]\}\right)\cup\left(\bigcup_{\substack{i\in I_P\\ j\in I_S}}\{S[t,i,j]\}\right)\right].$$

$$C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6.$$

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 Π.

Problème † 22 [Algorithme DPLL[3]] 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 $\overline{\ell}$ de toutes les clauses qui contiennent $\overline{\ell}$. Par exemple, l'ensemble C ci-dessous contient la clause unitaire $\{\overline{u_2}\}$:

$$C := \big\{\{\overline{u_1}, u_2, u_3, \overline{u_5}, \overline{u_4}\}, \{\overline{u_2}\},$$
$$\{u_1, u_2, u_4\}, \{\overline{u_2}, u_3\}, \{u_2, u_5\}, \{u_3, \overline{u_4}\}\big\}.$$

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

$$C \leftarrow \big\{\{\overline{u_1}, u_3, \overline{u_5}, \overline{u_4}\}, \{u_1, u_4\}, \{u_5\}, \{u_3, \overline{u_4}\}\big\}.$$

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

$$C \leftarrow \big\{\{\overline{u_1}, u_3, \overline{u_4}\}, \{u_1, u_4\}, \{u_3, \overline{u_4}\}\big\}. \tag{14}$$

(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 (14) est pur, ce qui nous donne après réduction :

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

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

$$C \leftarrow \varnothing.$$

Ces réductions ont successivement interprété les variables

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

ce qui suffit à satisfaire C.

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

3. Davis-Putnam-Logemann-Loveland

(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 $\varnothing$ 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 interprétées $\mathscr{F}$?

(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 $R \cup \{\overline{u}\}$. Ceci revient à tester les deux hypothèses possibles pour l'interprétation de la variable u. Si l'ensemble des clauses est satisfaisable, les interprétations des variables effectuées lors des appels récursifs définissent une interprétation prouvant la satisfaisabilité.

```
5 2
1 -3 4 5 0
-2 3 5 0
```

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'interprétation de la variable correspondante :

$$C := \{\{u_1, \overline{u_2}, u_3\},\ \{\overline{u_1}, u_4\},\ \{\overline{u_1}, \overline{u_4}\},\ \{u_2, u_5\},\ \{u_2, \overline{u_5}\},\ \{\overline{u_3}\}\}.$$

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 sont codées dans un fichier texte contenant une ligne par clause. Pour faciliter la lecture des données, la première ligne du fichier contient 2 entiers, le premier désigne le nombre de variables, le second le nombre de clauses. Chaque clause est 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'agit du littéral positif, sinon du littéral négatif. L'instance $\big\{\{u_1, \overline{u_3}, u_4, u_5\},\ \{\overline{u_2}, u_3, u_5\}\big\}$ 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 suppose que les n variables sont numérotées de 1 à n, qu'aucune clause ne contient plusieurs fois le même littéral ni deux littéraux opposés (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 doit être justifié, les algorithmes doivent être analysés et leurs complexités estimées, et les fonctions doivent 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 $[\![0, n^2]\!]$ 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 dédoubler 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 $I : U \to \{\mathscr{V}, \mathscr{F}\}$ des variables de U qui satisfait

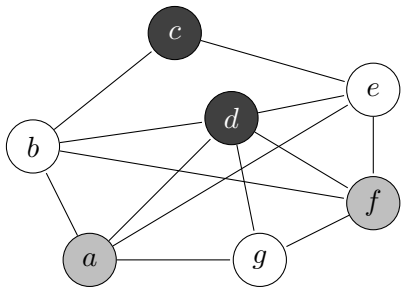
$$\forall c \in C \ I(c) = \mathscr{V}.$$

Exemple 10. Soit $U = \{u_1, u_2, u_3, u_4\}$ et $C = \{\{\overline{u}_1, u_2, u_3\}, \{u_1, \overline{u}_3, u_4\}\}$. 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 $c : X \to [\![1, k]\!]$ telle que

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

Exemple 11. 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 12. Soit $q = 3$ et on considère les ensembles

$$\begin{aligned} X &:= \{\text{Bob}, \text{Al}, \text{Cob}\} \\ Y &:= \{\text{Léa}, \text{Flo}, \text{May}\} \\ Z &:= \{1, 2, 3\} \\ T &:= \{(\text{Bob}, \text{Léa}, 2), (\text{Bob}, \text{Flo}, 3), (\text{Al}, \text{May}, 2), \\ &\qquad (\text{Cob}, \text{Flo}, 1), (\text{Cob}, \text{Léa}, 1)\} \end{aligned}$$

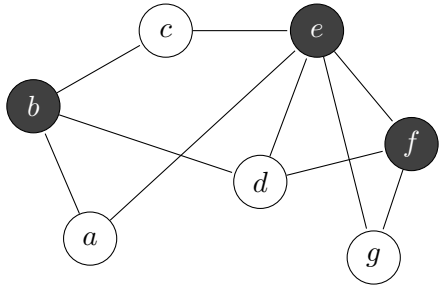
Cette instance du problème admet une solution :

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

Problème VC [couverture d'un graphe (par les 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

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

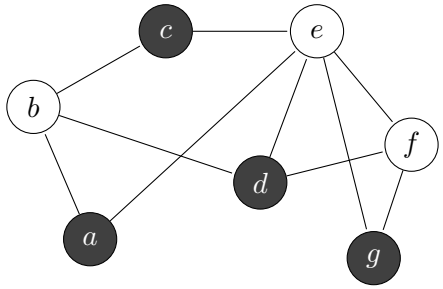
Exemple 13. Le graphe ci-dessous avec la borne $K = 3$, admet la couverture $Y = \{b, e, f\}$ (les sommets sont en noir dans le graphe). Pour $K = 2$ il n'y en a pas.



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

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

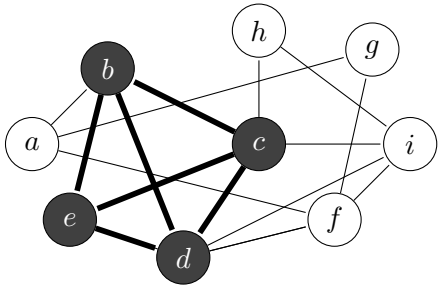
Exemple 14. Pour le graphe ci-dessous et pour $K = 4$, le problème admet une solution $Y = \{a, c, d, g\}$. Pour $K = 5$, il n'y a pas de solution.



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

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

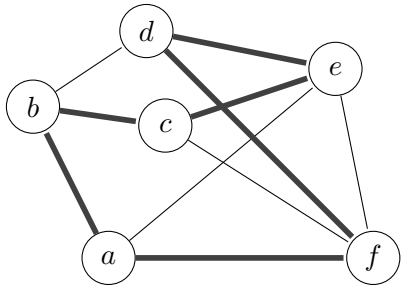
Exemple 15. Pour le graphe ci-dessous et $K = 4$, le problème a une solution $Y = \{b, c, d, e\}$ (ces sommets sont en noir dans le graphe). Pour $K = 5$, il n'y a pas de solution.



Problème HC [circuit hamiltonien]
Instance : Un graphe $G = (X, V)$, $X = \{x_1, x_2, \ldots, x_n\}$.
Question : Existe-t-il un circuit hamiltonien dans le graphe ? Autrement dit, eiste-t-il une permutation $\pi \in S_n$ telle que

$$(\forall i \in [\![1, n-1]\!] \ (x_{\pi(i)}, x_{\pi(i+1)}) \in V) \wedge ((x_{\pi(n)}, x_{\pi(1)}) \in V).$$

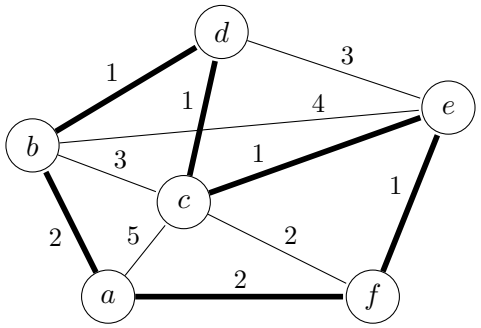
Exemple 16. Pour le graphe ci-dessous, il existe (au moins) un circuit hamiltonien (a, b, c, e, d, f, a) (matérialisé en gras dans le graphe).



Problème TSP [Traveling Salesman Problem]
Instance : Un graphe non-orienté $G = (X, V)$ avec $X = \{x_1, \ldots, x_n\}$ (villes), une pondération $w : V \to \mathbb{N}$ (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

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

Exemple 17. 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 $S \subseteq \mathscr{P}(E)$.
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

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

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 18. 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 $w : X \to \mathbb{N}$ (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 19. 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 une couverture par les sommets de G.
(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.

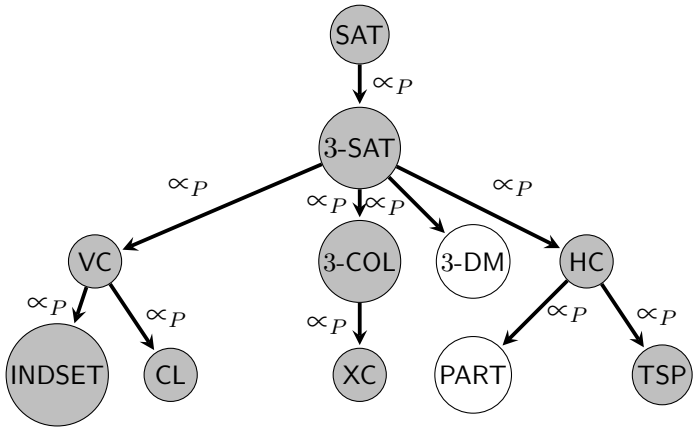


Figure 8. Ordre des preuves de NP-complétude. En gris les problèmes dont une preuve de NP-complétude est proposée dans ce document.

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 SAT $\in$ NP, on en déduit que 3-SAT $\in$ NP. Il nous reste à montrer

que SAT $\propto_P$ 3-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 $i \in [\![1, m]\!]$ et notons la clause $C_i := \{\ell_{i_1}, \ell_{i_2}, \dots, \ell_{i_k}\}$. 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

$$C'_i := \left\{ \{\ell_{i_1}, a_i, b_i\},\ \{\ell_{i_1}, a_i, \overline{b_i}\},\ \{\ell_{i_1}, \overline{a_i}, b_i\},\ \{\ell_{i_1}, \overline{a_i}, \overline{b_i}\} \right\}. \tag{15}$$

Si la clause $C_i = \{\ell_{i_1}\}$ 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

$$\{\ell_{i_1}, \ell_{i_2}, a_i\},\ \{\ell_{i_1}, \ell_{i_2}, \overline{a_i}\}. \tag{16}$$

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 $U'_i := \varnothing$, $C'_i := \{C_i\}$ et il n'y rien à prouver.

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

$$C'_i := \{\{\ell_{i_1}, \ell_{i_2}, a_{i_1}\}\} \cup \bigcup_{j=1}^{k-4} \left\{ \{\overline{a_{i_j}}, \ell_{i_{j+2}}, a_{i_{j+1}}\} \right\} \cup \{\{\overline{a_{i_k}}, \ell_{j_{k-1}}, \ell_{j_{k-2}}\}\}. \tag{17}$$

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 $\mathscr{V}$ et si $i \in \{k-1, k\}$, elles sont fixées à $\mathscr{F}$. Sinon on fixe a_{i_j} à $\mathscr{V}$ pour tous les indices $j \in [\![1, r-2]\!]$ et à $\mathscr{F}$ pour tous les indices $j \in [\![r-1, k-3]\!]$. 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. □

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

11. Le problème 2-SAT

Théorème 10. *Le problème 2-SAT de satisfaisabilité des clauses à 2 littéraux appartient à la classe P.*

Démonstration. La satisfaisabilité d'une instance de 2-SAT va être décidée par l'existence ou non d'un circuit dans un graphe orienté qui va coder cette instance. Quelques lemmes intermédiaires seront nécessaires à la preuve.

Une clause à deux littéraux $\{\ell, \ell'\}$ code la formule $(\ell \vee \ell')$ qui est logiquement équivalente aux implications $(\overline{\ell} \Rightarrow \ell')$ et $(\overline{\ell'} \Rightarrow \ell)$. Elles décrivent les conditions à satisfaire pour que la clause soit satisfaisable : si ℓ est fausse, alors ℓ' doit être vraie et si ℓ' est fausse alors ℓ doit être vraie.

Chaque implication $(u \Rightarrow v)$ est codée par un arc (u, v) dans un graphe orienté $G = (X, V)$ appelé *graphe d'implication*. Si $U = \{u_1, \dots, u_n\}$ désigne l'ensemble des variables booléennes et $C = \{C_1, \dots, C_m\}$ l'ensemble des clauses $C_j := \{\ell_j, \ell'_j\}$ de l'instance de 2-SAT, alors l'ensemble des $2n$ sommets X et l'ensemble des $2m$ arcs V du graphe d'implication G sont définis par

$$X := \bigcup_{i=1}^{n} \{u_i, \overline{u}_i\} \tag{18}$$

$$V := \bigcup_{j=1}^{m} \{(\overline{\ell}_j, \ell'_j), (\overline{\ell'}_j, \ell_j)\} \tag{19}$$

Les arcs $(\overline{\ell}_j, \ell'_j)$ et $(\overline{\ell'}_j, \ell_j)$ sont dits *contraposés*.

Par exemple, pour les clauses $C_1 := \{u_1, \overline{u}_2\}$ et $C_2 := \{u_2, u_3\}$, on a les implications :

$$(\overline{u}_1 \overset{(a)}{\Rightarrow} \overline{u}_2) \wedge (u_2 \overset{(b)}{\Rightarrow} u_1) \quad \text{et} \quad (\overline{u}_2 \overset{(c)}{\Rightarrow} u_3) \wedge (\overline{u}_3 \overset{(d)}{\Rightarrow} u_2). \tag{20}$$

et le graphe d'implication de cette instance est représenté en figure 9 (remarquer la symétrie dans le graphe).

Par transitivité de l'implication, on déduit des formules (a) et (c) en (20) que $\overline{u}_1 \Rightarrow u_3$ et des formules (d) et (b) que $\overline{u}_3 \Rightarrow u_1$ codant (en traitillés gris dans la figure) la nouvelle clause à deux littéraux $\{u_1, u_3\}$.

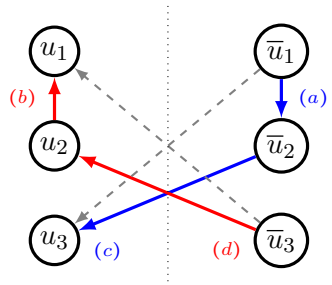


Figure 9. Graphe d'implication de (20) et sa fermeture transitive.

Plus généralement, pour une instance donnée de 2-SAT, quelles sont les clauses que l'on peut engendrer par transitivité ?

Lemme 1. *Soit $G = (X, V)$ le graphe d'implication d'une instance de 2-SAT. Si $\ell_1\ell_2\dots\ell_{k-1}\ell_k$ est un chemin de G alors $\overline{\ell}_k\overline{\ell}_{k-1}\dots\overline{\ell}_2\overline{\ell}_1$ est un chemin de G. On l'appelle chemin contraposé.*

Démonstration. Le chemin $\ell_1\ell_2\dots\ell_{k-1}\ell_k$ code la forme normale conjonctive

$$(\ell_1 \Rightarrow \ell_2) \wedge (\ell_2 \Rightarrow \ell_3) \wedge \dots \wedge (\ell_{k-1} \Rightarrow \ell_k). \tag{21}$$

En contraposant chaque implication et par commutativité de la conjonction, on obtient la forme normale conjonctive logiquement équivalente

$$(\overline{\ell}_k \Rightarrow \overline{\ell}_{k-1}) \wedge (\overline{\ell}_{k-1} \Rightarrow \overline{\ell}_{k-2}) \wedge \dots \wedge (\overline{\ell}_2 \Rightarrow \overline{\ell}_1). \tag{22}$$

codée par le chemin $\overline{\ell}_k\overline{\ell}_{k-1}\dots\overline{\ell}_2\overline{\ell}_1$ du graphe d'implication, puisque par construction, s'il contient un arc, il contient également l'arc contraposé. □

En appliquant le lemme 1 à l'exemple (20), le chemin contraposé du chemin $\overline{u}_1\overline{u}_2u_3$ est le chemin $\overline{u}_3u_2u_1$ (cf. figure 9).

Lemme 2. *L'ensemble des clauses engendrées par une instance de 2-SAT est l'ensemble des arcs de la fermeture transitive de son graphe d'implication.*

Démonstration. C'est évident puisque la transitivité dans le graphe code la transitivité de l'implication. □

On vérifie aisément que l'instance de 4 clauses

$$\{\{u_1, u_2\}, \{u_1, \overline{u}_2\}, \{\overline{u}_1, u_2\}, \{\overline{u}_1, \overline{u}_2\}\} \tag{23}$$

définie sur l'ensemble des variables $U := \{u_1, u_2\}$ dont le graphe d'implication est présenté à la figure 10 n'est pas satisfaisable.

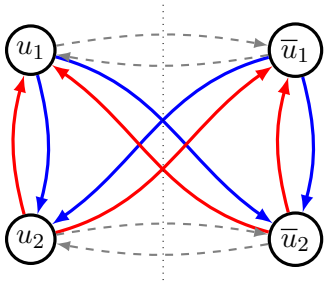


Figure 10. Graphe d'implication de (23) et sa fermeture transitive.

Pour cette instance particulière, on déduit par transitivité les deux implications $(u_1 \Rightarrow \overline{u}_1) \equiv (\overline{u}_1 \vee \overline{u}_1)$ et $(\overline{u}_1 \Rightarrow u_1) \equiv (u_1 \vee u_1)$ et par conséquent que $(\overline{u}_1 \wedge u_1)$, ce qui est contradictoire. Aucune interprétation de u_1 ne permet de satisfaire cette instance (même contradiction avec u_2).

Lemme 3. *Soit G le graphe d'implication d'une instance de 2-SAT, ℓ et ℓ' deux littéraux et I une interprétation des variables propopsitionnelles.*

(1) Si $\ell \dashrightarrow \ell'$, alors $I(\ell) = \mathscr{V} \Rightarrow I(\ell') = \mathscr{V}$ et $I(\ell') = \mathscr{F} \Rightarrow I(\ell) = \mathscr{F}$.

(2) Si $\ell \dashleftarrow\dashrightarrow \overline{\ell}$, alors l'instance n'est pas satisfaisable.

Démonstration. (1) Montrons que $(I(\ell) = \mathscr{V}) \Rightarrow (I(\ell') = \mathscr{V})$, le raisonnement est le même pour l'autre implication. Si $\ell \dashrightarrow \ell'$, alors $\ell \Rightarrow \ell'$ par transitivité de l'implication. Donc si $I(\ell) = \mathscr{V}$ et $\ell \dashrightarrow \ell'$, la règle du Modus Ponens permet de conclure que $I(\ell') = \mathscr{V}$.

(2) En effet, comme $\ell \dashrightarrow \overline{\ell}$ et $\overline{\ell} \dashrightarrow \ell$, on en déduit par transitivité que $\ell \Rightarrow \overline{\ell}$ et $\overline{\ell} \Rightarrow \ell$ et par conséquent que $\overline{\ell} \wedge \ell$, ce qui est contradictoire. □

Lemme 4. *Soit $G = (X, V)$ le graphe d'implication d'une instance de 2-SAT et Y l'une de ses composantes fortement connexes.*

(1) *Le sous-graphe induit par les littéraux opposés de Y est une composante fortement connexe appelée contraposée et notée $\overline{Y}$, dont les arcs sont les arcs opposés de ceux de Y.*

(2) *Si $\overline{Y} = Y$, alors l'instance n'est pas satisfaisable.*

Démonstration. Pour (1), c'est la conséquence du lemme 1. Pour (2), c'est la conséquence directe du lemme 3(2). □

Le graphe d'implication en figure 10 est fortement connexe et ne contient donc qu'une seule composante $X = \{u_1, \overline{u}_1, u_2, \overline{u}_2\}$ qui est sa propre contraposée $X = \overline{X}$. L'instance définie en (23) n'est donc pas satisfaisable.

On désigne par $\widetilde{G} = (\widetilde{X}, \widetilde{V})$ le *graphe quotient* de G pour la relation de connexité forte. Ses sommets sont les composantes fortement connexes de G que l'on connecte à chaque fois qu'il existe un arc qui les relie :

$$\widetilde{V} := \{(Y, Y') \in \widetilde{X} \mid \exists (\ell, \ell') \in Y \times Y' \ (\ell, \ell') \in V\}.$$

Lemme 5. *Le graphe quotient $\widetilde{G}$ du graphe d'implication d'une instance de 2-SAT est* acyclique.

Démonstration. Par l'absurde, soit $Y \neq Y'$ deux composantes fortement connexes dans un cycle du graphe quotient. Soit $\ell \in Y$ et $\ell' \in Y'$, alors nécessairement $\ell \leftrightarrow \ell'$ et par conséquent $Y = Y'$. □

Corollaire. *Les arcs (Y, Y') de la fermeture transitive du graphe quotient définissent une relation d'ordre partiel appelée ordre topologique entre composantes fortement connexes, notée $Y \to Y'$.*

Démonstration. La relation est réflexive, évidemment transitive et antisymétrique puisque le graphe est acyclique. □

Lemme 6. *Soit I l'interprétation d'une instance de 2-SAT. Tous les littéraux d'une même composante fortement connexe Y de son graphe d'implication ont même valeur de vérité et tous les littéraux de la composante contraposée $\overline{Y}$ ont la valeur de vérité opposée.*

Démonstration. Si un littéral de Y est vrai, d'après le lemme 3, tous les littéraux de Y sont vrais et tous ceux de $\overline{Y}$ sont donc faux. Si un littéral $\ell \in Y$ est faux, le littéral opposé $\overline{\ell} \in \overline{Y}$ est vrai, par conséquent tous ceux de $\overline{Y}$ également d'après le lemme 3 et ceux de Y sont alors faux. □

Lemme 7. *Une instance de 2-SAT est satisfaisable si et seulement si aucune des composantes fortement connexe de son graphe d'implication ne contient deux littéraux opposés.*

Démonstration. (1) Montrons que la condition est nécessaire. Si une composante fortement connexe Y contient deux littéraux opposés, sa contraposée $\overline{Y}$ également et donc $Y \cap \overline{Y} \neq \varnothing$. Comme les composantes fortement connexes forment une partition de X, nécessairement $Y = \overline{Y}$ et le lemme 4-(2) permet de conclure.

(2) Montrons que la condition est suffisante, c'est-à-dire que si aucune composante fortement connexe du graphe d'implication ne contient de littéraux opposés, l'instance est satisfaisable. Avec cette hypothèse, chaque composante fortement connexe est distincte de sa contraposée, il y en a donc un nombre pair $r = 2k$, notons les $Y_1, \ldots, Y_r$. Un *tri topologique* de ces composantes fortement connexes les range de manière à respecter l'ordre topologique et la liste $[Y_{\sigma(1)}, Y_{\sigma(2)}, \ldots, Y_{\sigma(r)}]$ obtenue satisfait la condition [4] :

$$\forall (i, j) \in [\![1, r]\!] \quad Y_{\sigma(i)} \to Y_{\sigma(j)} \Rightarrow i \leq j. \tag{24}$$

qui exprime que si Y précède Y' dans un chemin qui les relie dans le graphe quotient, Y est rangée *avant* Y' dans la liste, ce que réalise l'*algorithme de Tarjan* (cf. cours d'algorithmique de 3ème année).

Si $[Y_{\sigma(1)}, Y_{\sigma(2)}, \ldots, Y_{\sigma(r)}]$ respecte l'ordre topologique (24), alors les composantes $Y_{\sigma(1)}, \ldots, Y_{\sigma(k)}$ de la première moitié sont les contraposées des composantes $Y_{\sigma(k+1)}, \ldots, Y_{\sigma(r)}$ de la seconde moitié. En effet, compte tenu de la symétrie du graphe quotient (cf. figure 12), le graphe transposé $\widetilde{G}^T$ de $\widetilde{G}$ obtenu en inversant le sens des flèches codant l'ordre inverse, montre que si une composante fortement connexe Y est dans l'une des deux moitiés, sa contraposée est dans l'autre. Par conséquent aucune des deux moitiés ne peut contenir une composante fortement connexe Y et sa contraposée $\overline{Y}$, sans quoi l'autre moitié aussi, créant alors un cycle entre Y et $\overline{Y}$ qui seraient nécessairement confondues, ce qui est absurde.

4. Il est tentant d'écrire $i \leq j \Rightarrow Y_{\sigma(i)} \to Y_{\sigma(j)}$, mais deux composantes $Y_{\sigma(i)}$ et $Y_{\sigma(j)}$ ne sont pas nécessairement *comparables* (par exemple Y_7 et Y_2 dans la figure 12).

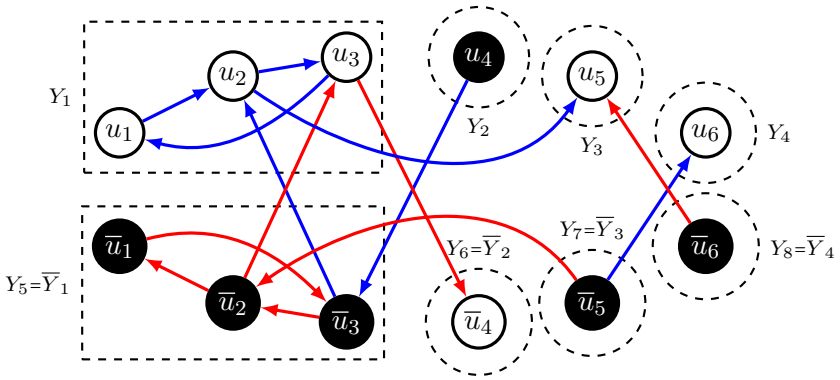


Figure 11. Graphe d'implication de (25) et ses 8 composantes fortement connexes.

On fixe alors la valeur de vérité des littéraux des composantes de la seconde moitié à $\mathscr{V}$, fixant celle des littéraux opposés dans leurs contraposée à $\mathscr{F}$ dans la première moitié. En procédant de la sorte, on est certain que l'on ne peut pas avoir d'arc $\mathscr{V} \to \mathscr{F}$.

Illustrons la construction avec l'instance de 2-SAT suivante :

$$C := \big\{\{\overline{u}_1, u_2\}, \{u_3, u_2\}, \{\overline{u}_2, u_3\}, \{\overline{u}_3, u_1\}, \{\overline{u}_3, \overline{u}_4\}, \{\overline{u}_2, u_5\}, \{u_5, u_6\}\big\}. \qquad (25)$$

Son graphe d'implication et ses composantes fortement connexes sont représentés en figure 11, et son graphe quotient en figure 12. Les boucles et les arcs que l'on déduit par transitivité ne sont pas représentés.

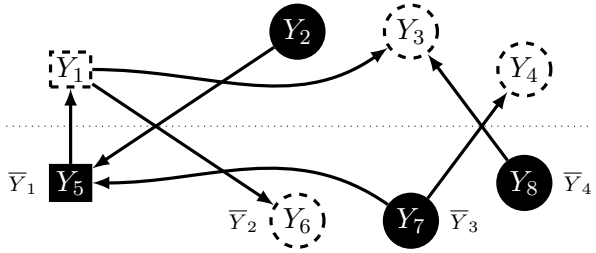


Figure 12. Graphe quotient du graphe de la figure 11 et interprétation des variables. Les littéraux sur fond blanc (resp. noir) sont $\mathscr{V}$ (resp. $\mathscr{F}$).

Un rangement possible des composantes fortement connexes Y_i respectant l'ordre topologique est

$$[Y_8, Y_2, Y_7, Y_5, Y_4, Y_1, Y_3, Y_6].$$

Autrement dit, en identifiant les composantes et leurs contraposées :

$$[\underbrace{\overline{Y}_4, Y_2, \overline{Y}_3, \overline{Y}_1}_{\mathscr{F}}, \underbrace{Y_4, Y_1, Y_3, \overline{Y}_2}_{\mathscr{V}}].$$

L'interprétation I des variables propositionnelles qui en découle est :

$$I(u_1) = I(u_2) = I(u_3) = I(u_5) = I(u_6) = \mathscr{V} \text{ et } I(u_4) = \mathscr{F}.$$

□

La construction du graphe d'implication se fait en temps linéaire en le nombre de variables n et le nombre de clauses m. Il reste à évaluer le coût du calcul des composantes fortement connexes du graphe puis du tri topologique. À l'aide d'un parcours en profondeur du graphe d'implication, l'algorithme de Tarjan calcule les composantes fortement connexes en instanciant la valeur de vérité des sommets et il le fait en temps linéaire en le nombre de sommets $2n$ (cf. cours d'algorithmique de 3ème année). C'est précisément quand l'algorithme aboutit à une contradiction en instanciant la valeur de vérité des sommets, qu'il a trouvé un cycle dans le graphe et peut conclure que l'instance n'est pas satisfaisable.

Le problème 2-SAT est donc polynomial. □

TP 3 Soit $U = \{u_1, u_2, \ldots, u_n\}$ l'ensemble des variables propositionnelles d'une instance de 2-SAT. Chaque ligne d'un fichier texte code une clause par les deux indices des variables associées à ces littéraux et dont le signe correspondant à la polarité du littéral.

(1) Construisez le graphe d'implication de l'instance avec une structure de listes d'adjacences.

(2) Utilisez l'algorithme de Tarjan pour décider si l'instance est satisfaisable ou non. Le cas échéant, donnez l'interprétation I obtenue par l'algorithme.

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

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

Nous allons montrer que 3-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

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

On note $n := |U|$ et $m := |C|$. Le graphe G est constitué, entre autres, de n *chaînettes* notées ∞_i, $i \in [\![1, n]\!]$ comprenant chacune une alternance de m sommets "gauches" $G_{i,j}$ et "droits" $D_{i,j}$ pour $j \in [\![1, m]\!]$ (cf. figure 13 pour la chaînette ∞_1 de notre exemple) :

Figure 13. La chaînette ∞_1 associée à la variable u_1.

La chaînette ∞_i est définie par les arcs

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

Un circuit hamiltonien ne peut traverser une chaînette ∞_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. En s'assurant que chaque chaînette puisse être parcourue dans les deux sens, il y aura 2^n façons de les traverser, soit autant que d'interprétations possibles des variables booléennes.

Dans cette optique, on connecte chaque chaînette ∞_i à la suivante ∞_{i+1} pour $i \in [\![1, n-1]\!]$ par les 2 extrémités, à l'aide des 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 chaînette ∞_i traversée (dans un sens ou l'autre), on peut passer à la chaînette suivante ∞_{i+1} de deux façons différentes, par la gauche ou la droite (cf. figure 14). On rajoute un sommet "source" S et un sommet "puit" P connectés respectivement aux deux extrémités de la première et de la dernière chaînette et reliés entre eux par l'arc $P \to S$.

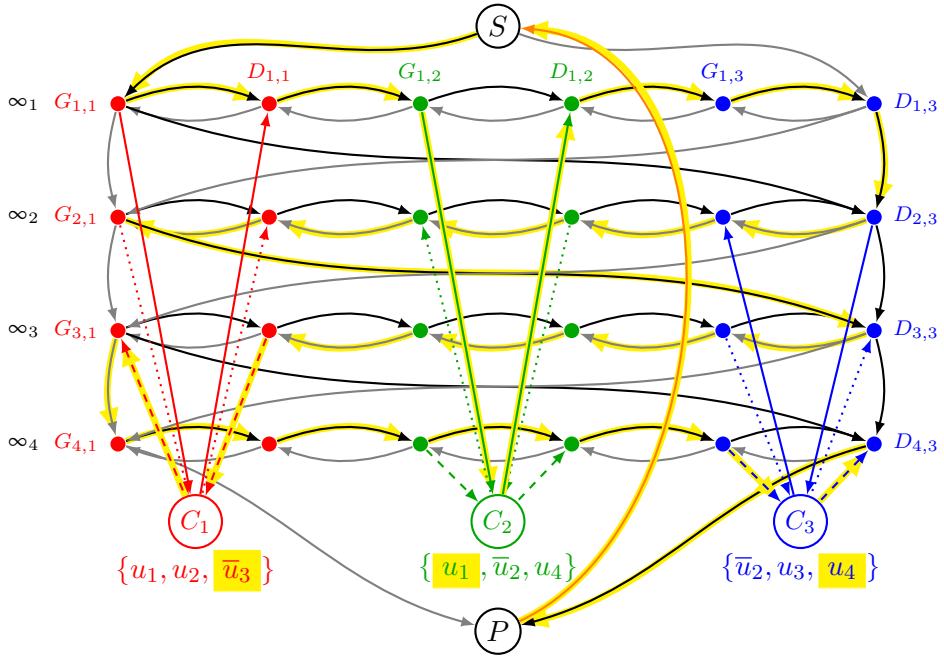


Figure 14. Circuit hamiltonien ⟶ associé à l'instance positive C de 3-SAT pour l'interprétation $\overline{u}_3 = u_1 = u_4 = \mathscr{V}$.

On achève la construction du graphe en créant un sommet C_j pour chacune des m clauses C_j et si u_i (resp. $\overline{u_i}$) est un littéral de C_j, on crée une dérivation dans la chaînette ∞_i par le sommet C_j grâce au chemin $G_{i,j} \to C_i \to D_{i,j}$ (resp. $D_{i,j} \to C_i \to G_{i,j}$). 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

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

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 $\overline{u}_3$ 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 $\overline{u}_3$ 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 $P \to S$. 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 14).

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 $I(u_i) := \mathscr{V}$ de gauche à droite et $I(u_i) := \mathscr{F}$ 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

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

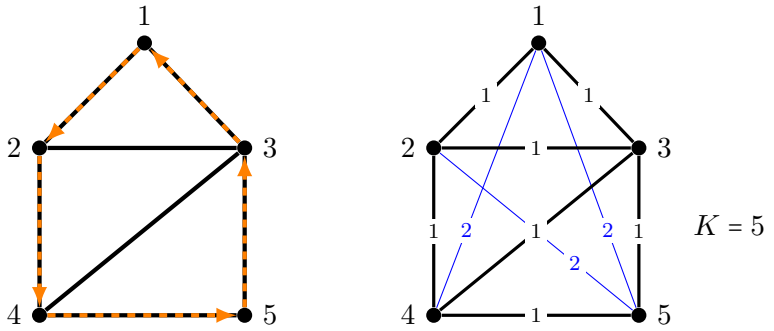


Figure 15. À gauche : instance du problème HC et circuit hamiltonien. À droite : instance associée du problème TSP.

On transforme l'instance $G = (X, V)$ du problème HC en une instance $G' = (X, V')$, K et $w : V' \to \mathbb{N}$ 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 15.

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. □

Proposition 7. *Le problème VC de la couverture d'un graphe est NP-complet.*

Démonstration. On a VC $\in$ NP. En effet, il suffit de fournir la couverture 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 $m := |V|$.

Nous allons montrer que 3-SAT $\propto_P$ VC en construisant à partir d'une instance de 3-SAT, un graphe $G = (X, V)$ qui admet une couverture Y de taille inférieure à K à déterminer, si et seulement si cette instance est satisfaisable. On note $U := \{u_1, \ldots, u_n\}$ l'ensemble des variables booléennes et $C := \{C_1, \ldots, C_m\}$ l'ensemble des clauses de l'instance de 3-SAT. On note également $\{\ell_j^a, \ell_j^b, \ell_j^c\}$ les trois littéraux de la clause C_j.

On crée deux groupes de graphes complets, les *duos* associés aux n variables propositionnelles, et les *trios* associés au m clauses. Pour chaque variable u_i on crée un graphe duo de sommets $\{u_i, \overline{u_i}\}$ et d'arcs D_i. Pour chaque clause C_j on crée un graphe trio de sommets $\{a_j, b_j, c_j\}$ et d'arcs T_j. Les arcs des duos sont en noir et ceux des trios sont en gris dans les figures (cf. figure 16).

Figure 16. Duo associé à u_i et trio associé à C_j.

Pour contrôler l'arc d'un duo D_i, une couverture Y doit nécessairement contenir l'un des deux sommets u_i ou $\overline{u_i}$ et deux des trois sommets a_j, b_j et c_j pour contrôler les trois arcs d'un trio T_j. Une couverture 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 en fonction du contenu de chaque clause C_j en reliant respectivement les sommets a_j, b_j et c_j aux trois littéraux de C_j dans l'ordre. Ce graphe est illustré en figure 17 pour les ensembles de variables U et de clauses C suivants :

$$U := \{u_1, u_2, u_3, u_4\}$$
$$C := \big\{ \underbrace{\{u_1, u_2, \overline{u}_3\}}_{C_1}, \underbrace{\{u_1, \overline{u}_2, u_4\}}_{C_2}, \underbrace{\{\overline{u}_2, u_3, u_4\}}_{C_3} \big\}.$$

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

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

et pour ensemble des arcs l'ensemble

$$V := \left(\bigcup_{j=1}^{m} D_i \right) \cup \left(\bigcup_{j=1}^{m} T_j \right) \cup \left(\bigcup_{j=1}^{m} \{(a_j, \ell_j^a), (a_j, \ell_j^b), (a_j, \ell_j^c)\} \right)$$

(1) Dans un premier temps, montrons que si C est satisfaisable alors le graphe $G := (X, V)$ admet une couverture Y telle que $|Y| \leq n + 2m$. Si C est satisfaisable, il existe une interprétation I des variables telle que toutes les clauses C_j sont satisfaites. Par exemple ici $I(u_1) = \mathscr{V}$, $I(u_2) = \mathscr{F}$,

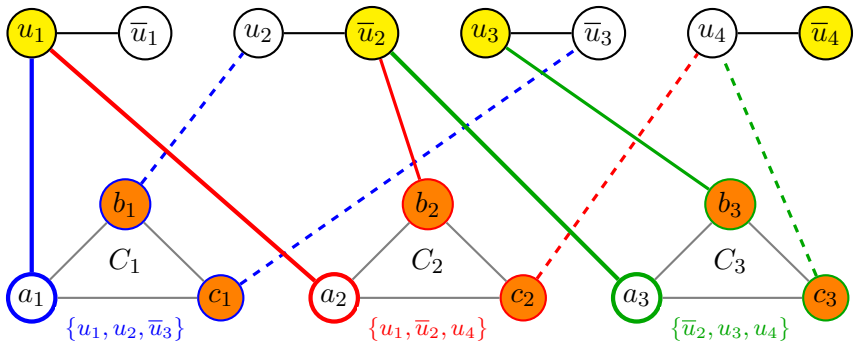


FIGURE 17. La couverture $Y := Y_U \cup Y_C$ déterminée par l'interprétation $I(u_1) = I(\overline{u}_2) = I(u_3) = I(\overline{u}_4) = \mathscr{V}$.

$I(u_3) = \mathscr{V}$ et $I(u_4) = \mathscr{F}$. Nous allons construire les couvertures Y_U et Y_C des duos et des trios respectivement, composant la couverture $Y := Y_U \cup Y_C$ de ce graphe. On pose :

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

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

Pour chacune des m clauses C_j il faut fixer la paire Y_j de sommets du trio correspondant qui feront partie de la couverture Y_C. Chaque littéral de Y_U contrôle tous les arcs qui le relient aux trios/clauses et il y en a *au moins* un pour chaque clause puisqu'elles sont toutes satisfaisables (ils sont en traits pleins dans la figure, les autres sont en traitillés.) On écarte de chaque trio $\{a_j, b_j, c_j\}$ le premier sommet qui appartient à l'un de ces arcs (dans l'exemple de la figure 17, ce sont les arcs $(\underline{a_1}, u_1)$, $(\underline{a_2}, u_1)$ et $(\underline{a_3}, \overline{u}_2)$ en traits gras.) :

$$Y_C := \bigcup_{j=1}^{m} Y_j \quad \text{où} \quad Y_j := \begin{cases} \{b_j, c_j\} & \text{si } I(\ell_j^a) = \mathscr{V}. \\ \{a_j, c_j\} & \text{si } I(\ell_j^a) = \mathscr{F} \text{ et } I(\ell_j^b) = \mathscr{V}. \\ \{a_j, b_j\} & \text{si } I(\ell_j^a) = I(\ell_j^b) = \mathscr{F}. \end{cases}$$

On a trivialement $|Y|_U = n$ et $|Y|_C = 2m$ et l'ensemble $Y = Y_U \cup Y_C$ est bien une couverture du graphe de taille égale à $n + 2m$.

(2) Réciproquement, on suppose que l'on dispose d'une couverture Y de taille égale à $n + 2m$ (on ne peut pas faire moins), montrons que la clause C est satisfaisable. Nous savons déjà que cette couverture contient nécessairement un sommet de chaque duo et deux sommets de chaque trio. Considérons la couverture suivante (cf. figure 18) :

$$Y = \underbrace{\{\overline{u}_1, \overline{u}_2, \overline{u}_3, u_4\}}_{Y_U} \cup \underbrace{\{\overbrace{a_1, b_1}^{Y_1}, \overbrace{a_2, c_2}^{Y_2}, \overbrace{b_3, c_3}^{Y_3}\}}_{Y_C}.$$

On définit l'interprétation I des variables booléennes par :

$$\forall u \in U \quad I(u) := \begin{cases} \mathscr{V} & \text{si } u \in Y_U. \\ \mathscr{F} & \text{si } \overline{u} \in Y_U. \end{cases}$$

Il reste à s'assurer que chaque clause C_j contient au moins un littéral de Y_U, autrement dit qu'un sommet au moins des trois sommets a_j, b_j ou b_j est connecté à un sommet de Y_U. Par construction, chaque clause/trio C_j contient un sommets qui n'appartient pas à Y_i (les sommets c_1, b_2 et a_3 la figure 18) et l'arc qui le relie à un duo (en gras) est donc nécessairement couvert par le littéral du duo qui appartient à Y_U (par $\overline{u}_2$ pour b_2 et a_3 et par $\overline{u}_3$ pour c_1).

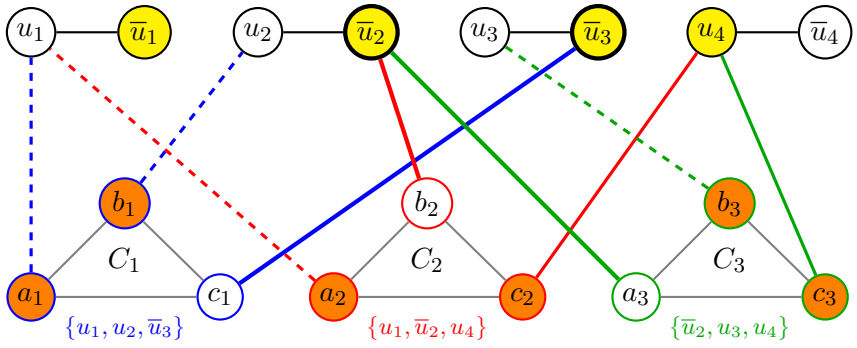


FIGURE 18. L'interprétation $I(\overline{u}_1) = I(\overline{u}_2) = I(\overline{u}_3) = I(u_4) = \mathscr{V}$ est fixée par la couverture $Y = Y_U \cup Y_C$.

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 $O(n+m)$. □

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

(1) Y est une couverture 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) ⇒ (2) ⇒ (3) ⇒ (1), les équivalences sont obtenues par transitivité de l'implication logique. Pour illustrer la preuve, on considère le graphe de la figure 19 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 une couverture 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

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

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

(3) ⇒ (1) On sait que $\overline{Y}$ constitue une clique dans le graphe complémentaire $\overline{G} = (X,\overline{V})$. Montrons que Y est une couverture 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. □

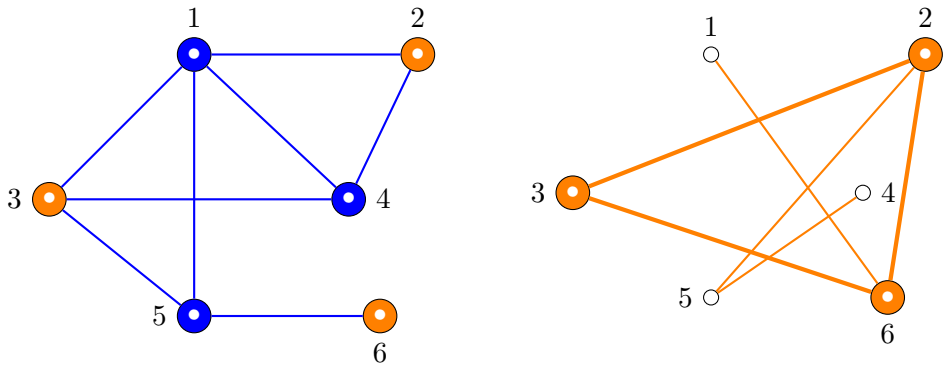


Figure 19. À gauche : un graphe $G = (X,V)$ et une couverture Y. À droite : le graphe complémentaire $\overline{G} = (X,\overline{V})$, un ensemble de sommets indépendants $\overline{Y}$ du graphe G et une clique du graphe $\overline{G}$.

Corollaire. *On a l'équivalence suivante : VC $\propto_P$ INDSET $\propto_P$ CL. Ces trois problèmes sont donc NP-complets.*

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

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-SAT $\propto_P$ 3-COL. On considère (U, C) une instance de 3-SAT où $U = \{u_1, \ldots, u_n\}$ est l'ensemble des n variables booléennes et $C = \{C_1, \ldots, C_m\}$ 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 $U_i := \{u_i, \overline{u_i}\}$ de sommets u_i et $\overline{u_i}$ reliés par l'arc $(u_i, \overline{u_i})$ (cf. graphe à gauche dans la figure 20).

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 a_j, b_j et c_j de la clause C_j (qui sont donc trois des $2n$ sommets associés aux littéraux u_i pour créer les n duos précédent) — 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 20.

Les sommets k_j, l_j, m_j, n_j et o_j deviendront rapidement accessoires et pour simplifier la représentation du graphe, on résumera la grappe X_j en la codant à l'aide du schéma triangulaire à droite de la figure 20.

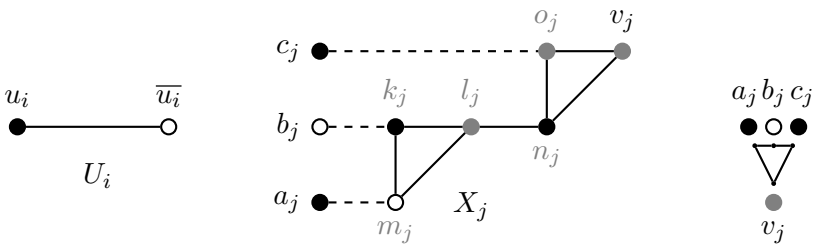


FIGURE 20. À gauche : un duo U_i. Au centre : une grappe X_j. À droite : notation condensée de la grappe.

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 $\overline{u_i}$ du groupe 1 et le sommet "puit" 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 :

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

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

Ce graphe est représenté en figure 21. Dans cette figure et pour faciliter la compréhension, on a volontairement distingué les sommets/littéraux u_i ou $\overline{u_i}$ des sommets/littéraux a_j, b_j et c_j alors qu'ils sont confondus. Tous les sommets reliés par des pointillés de même couleur ne font qu'un. Par exemple, les sommets c_1, c_2 et u_n sont confondus pour la clause $C_1 = \{u_1, \overline{u}_2, u_n\}$.

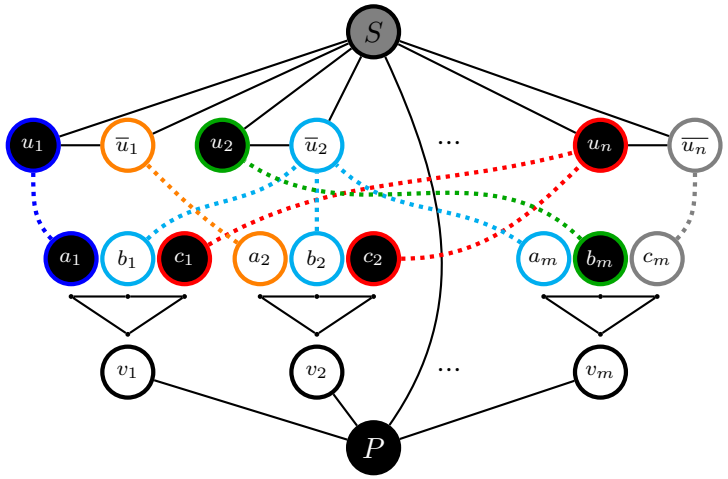


FIGURE 21. Instance du problème 3-COL.

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.

Lemme 8. *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 20, les autres sont laissées en exercice. □

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. □

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 XC $\in$ 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 (couverture exacte) 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 une couverture exacte, 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 $E = \varnothing$. Cet algorithme est en $O(|S|.|E|)$.

Nous allons montrer que 3-COL $\propto_P$ 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

$$\mathscr{V}(x) := \{y \in X \mid \{x, y\} \in V\} \tag{26}$$

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

$$S_x^R := \{x_R\} \cup \bigcup_{y \in \mathscr{V}(x)} \{y_x^R\},$$
$$S_x^V := \{x_V\} \cup \bigcup_{y \in \mathscr{V}(x)} \{y_x^V\},$$
$$S_x^B := \{x_B\} \cup \bigcup_{y \in \mathscr{V}(x)} \{y_x^B\}.$$

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 22 pour le sommet $x = a$ de voisins $\mathscr{V}(a) = \{b, c\}$. L'ensemble E réunit tous les éléments de X ainsi que tous les éléments des S_x^c :

$$E := X \sqcup \left(\bigsqcup_{(x,c) \in X \times C} S_x^c \right) \tag{27}$$

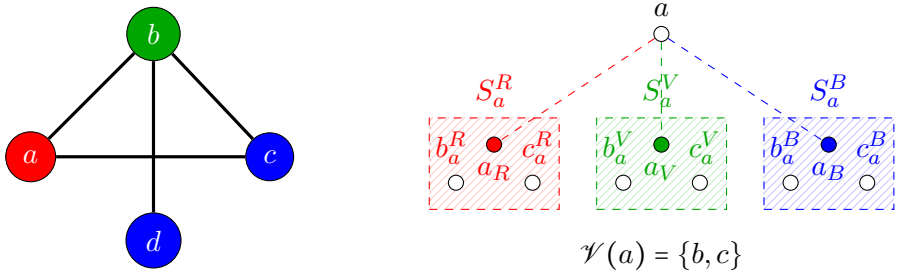


Figure 22. À gauche : une instance de 3-COL. À droite : les parties S_a^R, S_a^V, S_a^B, $\{a, a_R\}$, $\{a, a_V\}$ et $\{a, a_B\}$ de E dans S.

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

$$S := \bigcup_{(x,c) \in X \times C} \left(S_x^c \cup \underbrace{\{x, x_c\}}_{\in \text{gr. 1}} \cup \Big(\bigcup_{\substack{y \in \mathscr{V}(x) \\ c' \in C \setminus \{c\}}} \underbrace{\{y_x^c, x_y^{c'}\}}_{\in \text{gr. 2}} \Big) \right). \tag{28}$$

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 23 montre l'instance de XC associée à l'instance de 3-COL considérée dans la figure 22.

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 $y \in \mathscr{V}(x)$.

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

(1) Les deux parties S_x^c de couleurs *différentes* de $\nu(x)$, sur fond uni dans la figure 23 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 23.

(3) Il reste donc à recouvrir dans $S_x^{\nu(x)}$ les $|\mathscr{V}(x)|$ voisins $y_x^{\nu(x)}$ où $y \in \mathscr{V}(x)$, ce qui est réalisé par les paires $\{y_x^{\nu(x)}, x_y^{\nu(y)}\}$ en traits pleins noirs dans la figure 23 reconstituant ainsi le graphe G.

Ces parties couvrent 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 couvert 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 $\nu(x) \neq \nu(y)$.

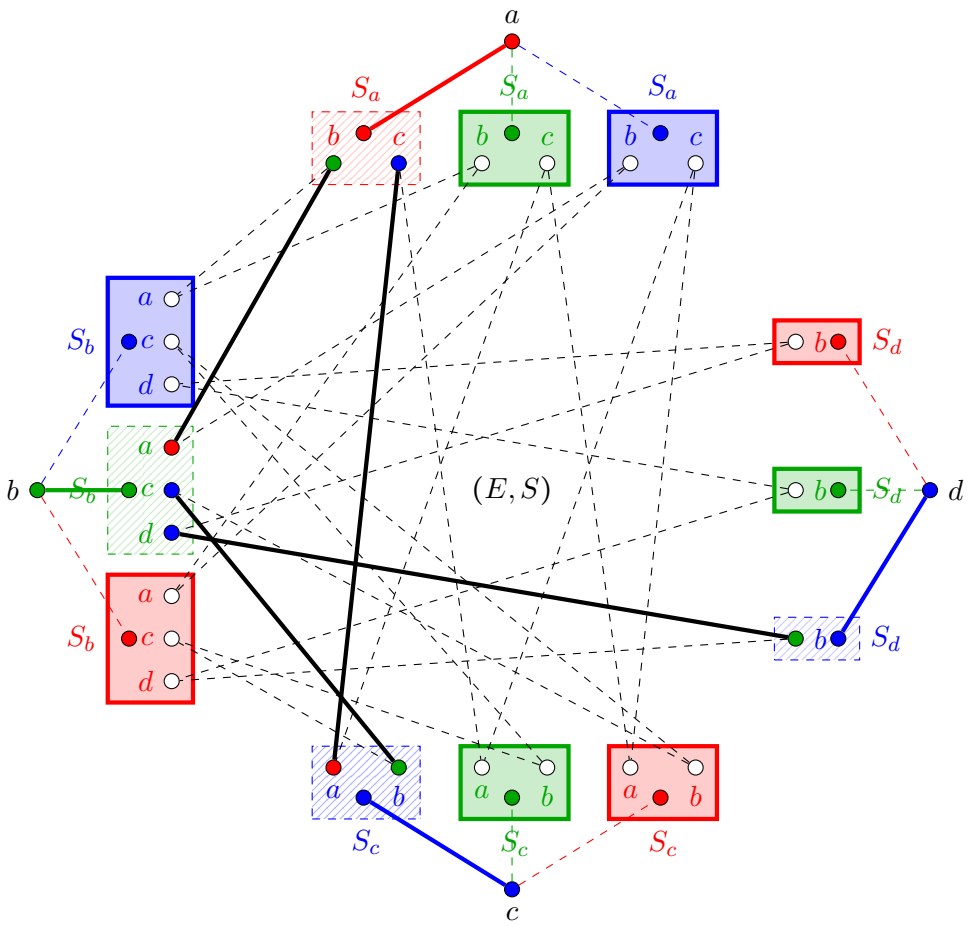


Figure 23. L'instance (E, S) associée au graphe $G = (X, V)$ et la **partition** associée au 3-coloriage de la figure 22 .

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 couverts que par les parties $S_x^{c'}$, $c' \neq \nu(x)$ qui restent les seules à les contenir et couvrent du même coup les éléments $y_x^{c'}$, $c' \neq \nu(x)$ pour $y \in \mathscr{V}(x)$. 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. $d(x) := |\mathscr{V}(x)|$ où $\mathscr{V}$ est défini en (26). Vérifiez que la somme des degrés des sommets est égale à $2|V|$, i.e.

$$\sum_{x \in X} d(x) = 2\,|V|. \tag{29}$$

Montrons que cette transformation est polynomiale. On a

$$\forall (x, c) \in X \times C \quad |S_x^c| = 1 + d(x). \tag{30}$$

Les ensembles X et S_x^c, $(x, c) \in X \times C$ formant une partition de E (cf. (27)), la formule de sommation nous donne

$$\begin{aligned} |E| &= |X| + \sum_{(x,c) \in X \times C} |S_x^c| \\ &= n + \sum_{x \in X} \sum_{c \in C} |S_x^c| \\ &= n + \sum_{x \in X} 3(1 + d(x)) \quad \text{d'après (30)} \\ &= 4n + 6m \quad \text{d'après (29)} \end{aligned}$$

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

$$\begin{aligned} |S| &= 3n + \sum_{(x,c) \in X \times C} d(x) \\ &= 3n + \sum_{x \in X} \sum_{c \in C} d(x) \\ &= 3n + 3 \sum_{x \in X} d(x) \\ &= 3n + 6|V| \quad \text{d'après (29)} \\ &= 3n + 6m. \end{aligned}$$

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

12. Annexe : la machine RAM

La machine RAM, acronyme de *Register Addressable Memory*[5], est un modèle abstrait de calcul introduit dans les années 1960. 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, ces modèles ayant été élaborés à une époque où les ordinateurs tels que nous les connaissons 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 plus familiers des informaticiens.

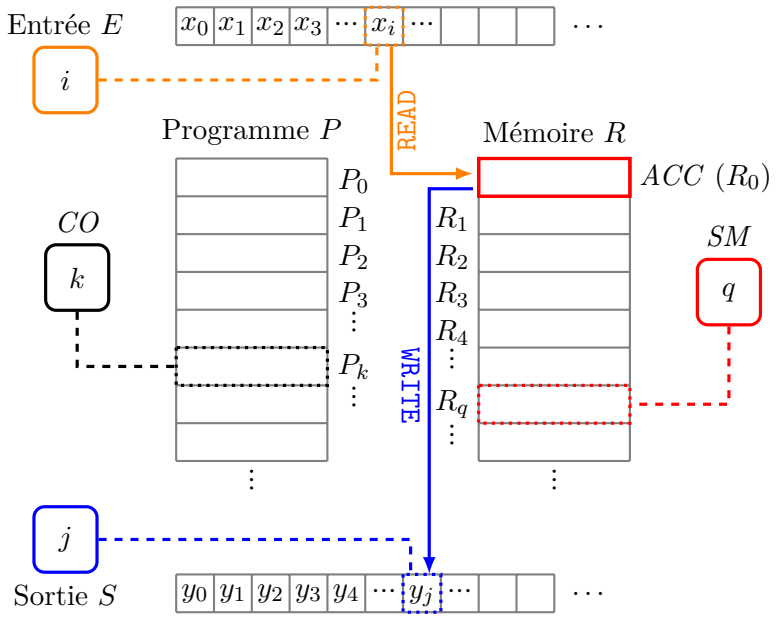


Figure 24. Vision schématisée de la machine RAM

Cette machine est constituée de 4 éléments principaux :

5. plus parlant que *Random Access Memory* pour des francophones

(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, ... 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 d'instructions codées par des couples (code opération, adresse) et indexée par le *compteur ordinal* (CO).

On charge (virtuellement) les instructions du programme dans le bloc programme et on saisit (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 P_k du programme sont décodées séquentiellement. Pour cela, le compteur ordinal contient l'adresse de la prochaine instruction à décoder, initialement 0. Après qu'une instruction a été décodée, le compteur ordinal est incrémenté pour passer à l'instruction suivante, sauf si l'instruction est une *rupture de séquence*, auquel cas son rôle est précisément de modifier la valeur du compteur ordinal. La machine s'arrête après l'instruction `STOP`.

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 l'accumulateur (R_0) et les résultats des calculs sont renvoyés séquentiellement sur la bande de sortie à l'aide de l'instruction d'écriture `WRITE`.

On dispose de l'ensemble des registres R_i de la mémoire pour y stocker des résultats (le contenu du registre R_i est noté $R[i]$). Le registre R_0 a un statut particulier, c'est l'*accumulateur* (ACC). Une opération arithmétique remplace le contenu de l'accumulateur par le résultat de l'opération entre l'accumulateur et le contenu d'un registre. La plupart des ruptures de séquence dépendent du contenu de ce registre.

Type	Instruction	Signification
Entrées/Sorties	READ	ACC $\leftarrow e_i$, $i \leftarrow i+1$
	WRITE	$s_j \leftarrow$ ACC, $j \leftarrow j+1$
Affectations	LOAD #n	ACC $\leftarrow n$
	LOAD n	ACC $\leftarrow R[n]$
	LOAD @n	ACC $\leftarrow R[R[n]]$
	STORE n	$R[n] \leftarrow$ ACC
	STORE @n	$R[R[n]] \leftarrow$ ACC
Arithmétiques	ADD n	ACC $\leftarrow$ ACC $+ R[n]$
	SUB n	ACC $\leftarrow$ ACC $- R[n]$
	MUL n	ACC $\leftarrow$ ACC $\times R[n]$
	DIV n	ACC $\leftarrow$ ACC $\div R[n]$
	MOD n	ACC $\leftarrow$ ACC $\%\ R[n]$
	INC n	$R[n] \leftarrow R[n]+1$
	DEC n	$R[n] \leftarrow R[n]-1$
Ruptures de séquence	JUMP n	CO $\leftarrow n$
	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 du programme
	NOP	No Operation

Table 3. Instructions de la machine RAM

À l'exception des instructions READ, WRITE, STOP et NOP, une instruction est un couple CODOP adr constitué d'un *code opération* et d'une *adresse*. 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.

L'instruction NOP ne fait "rien". Elle peut être insérée entre des instructions du programme en prévention pour "recaler" des sauts en cas de modification du code.

Le tableau 3 regroupe les différentes instructions possibles. Les opérations arithmétiques se déclinent en adressage absolu et en adressage relatif avec #n et @n pour travailler respectivement avec la valeur n et la valeur $R[R[n]]$. Les adresses pour les ruptures de séquence peuvent également être relatives, par ex. l'instruction JUMZ @3 fera sauter le programme à l'instruction dont le numéro est contenu dans le registre R_3.

Exemples. L'algorithme ci-dessous calcule la partie entière de la moyenne arithmétique des valeurs saisies sur la bande d'entrée. Par convention la valeur nulle sert de marqueur pour la fin des données à lire sur la bande d'entrée. Le registre R_1 contient le nombre de valeurs non-nulles lues et le registre R_2 la somme des valeurs non-nulles lues.

```
 0 | LOAD #0      ; ACC ← 0
 1 | STORE 1      ; R[1] ← ACC
 2 | STORE 2      ; R[2] ← ACC
 3 | READ         ; ACC ← ENTREE[I++]
 4 | JUMZ 9       ; SI ACC = 0 SAUTER A INSTRUCTION #9
 5 | ADD 2        ; ACC ← ACC + R[2]
 6 | STORE 2      ; R[2] ← ACC
 7 | INC 1        ; R[1] ← R[1] + 1
 8 | JUMP 3       ; SAUTER A INSTRUCTION #3
 9 | LOAD 2       ; ACC ← R[2]
10 | DIV 1        ; ACC ← ACC / R[1]
11 | WRITE        ; SORTIE[J++] ← ACC
12 | STOP         ; ARRET
```

L'algorithme suivant cherche la plus grande valeur parmi celles qui sont fournies en entrée. Par convention la valeur nulle sert de marqueur pour la fin des données à lire sur la bande d'entrée. Le registre R_1 sert à mémoriser la plus grande valeur courante et le registre R_2 permet de conserver la dernière valeur lue. On suppose que la liste contient au moins une valeur, il y a donc au moins deux cellules de la bande d'entrée qui sont utilisées.

```
 0 | READ         ; ACC ← ENTREE[I++]
 1 | STORE 1      ; R[1] ← ACC
 2 | READ         ; ACC ← ENTREE[I++]
 3 | JUMZ 11      ; SI ACC = 0 SAUTER A INSTRUCTION #11
 4 | STORE 2      ; R[2] ← ACC
 5 | LOAD 1       ; ACC ← R[1]
 6 | SUB 2        ; ACC ← ACC - R[2]
 7 | JUMG 2       ; SI (R[2] - R[1] > 0) SAUTER A INSTRUCTION #2
 8 | LOAD 2       ; ACC ← R[2]
 9 | STORE 1      ; R[1] ← ACC
10 | JUMP 2       ; SAUTER A INSTRUCTION #2
11 | LOAD 1       ; ACC ← R[1]
```

```
12 | WRITE       ; SORTIE[J++] ← ACC
13 | STOP        ; ARRET
```

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

13. Annexe : quelques rappels de logique propositionnelle

On se donne un ensemble de *variables propositionnelles*, i.e. des éléments de l'ensemble $\{\mathscr{V}, \mathscr{F}\}$ 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 $u \in \{\mathscr{V}, \mathscr{F}\}$, on note sa négation $\neg u$ ou $\overline{u}$ pour plus de lisibilité des formules.

(2) Le connecteur binaire de *conjonction* $\wedge$: si $u, v \in \{\mathscr{V}, \mathscr{F}\}$, 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 $u, v \in \{\mathscr{V}, \mathscr{F}\}$, 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	$\overline{u}$
$\mathscr{V}$	$\mathscr{F}$
$\mathscr{F}$	$\mathscr{V}$

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

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

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) $formule := \mathscr{V}$

(3) $formule := \mathscr{F}$

(4) $formule := \neg formule$

(5) $formule := (formule \wedge formule)$

(6) $formule := (formule \vee formule)$

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 20. Si u_1 est une variable propositionnelle, u_1 ou $\overline{u}_1$ 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 $U \to \{\mathscr{V}, \mathscr{F}\}$.*

Une interprétation I d'un ensemble de variables propositionnelle 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 21. Soit F et G les formules :

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

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 :

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

Proposition 9 (*Lois de De Morgan*). *Soit u et v deux variables propositionnelles. Alors*

$$\overline{(u \vee v)} \equiv \overline{u} \wedge \overline{v} \tag{31}$$

$$\overline{(u \wedge v)} \equiv \overline{u} \vee \overline{v} \tag{32}$$

Proposition 10 (*Règle du Modus Ponens*). *Soit P et Q deux formules propositionnelles et I une interprétation. Alors*

$$\big((I(P) = \mathscr{V}) \wedge (P \Rightarrow Q)\big) \;\Rightarrow\; (I(Q) = \mathscr{V}). \tag{33}$$

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 $\mathscr{V}$.*

Exemple 22. La formule propositionnelle F de l'exemple 21 est satisfaisable, par exemple avec l'interprétation $I(u_1) := \mathscr{V}$, $I(u_2) := \mathscr{F}$ et $I(u_3) := \mathscr{F}$.

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 $u \vee \overline{u} \equiv \mathscr{V}$, 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 23. La formule $u_1 \vee \overline{u}_2 \vee u_3$ est une clause. La formule $(u_1 \vee \overline{u}_2 \vee u_3) \wedge (u_1 \vee \overline{u}_3)$ 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 $\{u_1, \overline{u}_2, u_3\}$. 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.

14. 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 *prédecesseur* de y et y le *successeur* de x. L'ensemble des arcs définit implicitement une *correspondance* $\Gamma : X \to X$ 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 $\mathscr{R}$ définie sur X par $x \mathscr{R} y$ si et seulement $(x, y) \in V$, réflexivité, symétrie, transitivité, etc. Si l'orientation n'a pas d'importance, l'ensemble V est définit comme un sous-ensemble de paires $\{x, y\}$ et de singletons $\{x\}$ de X, donc un sous-ensemble de $\mathscr{P}_2(X) \cup \mathscr{P}_1(X)$. 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 $\overline{G} := (X, \overline{V})$ où $\overline{V}$ est le complémentaire de V dans $X \times X$ (ses arcs sont exactement ceux qui ne sont pas dans G). On appelle *graphe complet* ou *clique*, le graphe $G = (X, X \times X)$.

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*). La *longueur* d'un chemin (resp. d'une chaîne) est le nombre de ses arcs (resp. de ses arêtes).

Remarque. La terminologie des graphes non-orientés peut s'appliquer aux graphes orientés si l'on ne tient plus compte de l'orientation des arcs.

Soit $G := (X, V)$ un graphe. Le *graphe transposé* de G est le graphe $G^T = (X, V^T)$ défini en inversant le sens des arcs de V, autrement dit, $V^T := \{(y, x) \mid (x, y) \in V\}$.

Soit $G := (X, V)$ un graphe. Un *sous-graphe* de G est un graphe $H = (Y, W)$ tel que

$$(Y \subseteq X) \ \wedge \ \big(\forall (x, y) \in Y^2 \ (x, y) \in W \Rightarrow (x, y) \in V\big).$$

Soit $Y \subseteq X$, on appelle *sous-graphe de G induit par Y* le sous-graphe $H = (Y, W)$ tel que

$$\forall (x, y) \in Y^2 \ \ (x, y) \in W \Leftrightarrow (x, y) \in V.$$

On dit que deux sommets x et y d'un graphe non-orienté sont *connectés*, ce que l'on note x -- y, s'il existe une chaîne qui les relie. Cela définit trivialement une relation d'équivalence sur l'ensemble des sommets appelée *relation de connexité*. Les sous-graphes induits par les classes d'équivalence sont appelées *composantes connexes* du graphe et s'il n'en contient qu'une, il est dit *connexe*. Si le graphe est orienté, avec les même définitions, on parlera respectivement de sommets *faiblement connectés*, de *relation de connexité faible*, de *composantes faiblement connexes* et de graphe *faiblement connexe*.

On dit que deux sommets x et y d'un graphe orienté sont *connectés*, ce que l'on note $x \dashrightarrow y$, s'il existe un chemin qui relie x à y. Cette relation est encore réflexive et transitive mais plus nécessairement symétrique. On définit alors une relation de *connexité forte* entre sommets par $x \dashleftarrow\!\dashrightarrow y$ si et seulement si $(x \dashrightarrow y) \wedge (y \dashrightarrow x)$ qui est une relation d'équivalence. Les sous-graphes induits par ces classes sont appelées *composantes fortement connexe*. Un graphe qui ne contient qu'une seule composante fortement connexe est dit *fortement connexe*.

La *fermeture transitive* du graphe $G = (X, V)$ est le graphe $G_\triangle := (X, V_\triangle)$ dont les arcs sont ceux de V auxquels on a ajouté tous ceux que l'on peut déduire par transitivité, ce qui équivaut à dire qu'un arc (x, y) fait partie de la fermeture transitive si $x \dashrightarrow y$ en appliquant la transitivité de sommet en sommet sur ce chemin :

$$V_\triangle := \{(x, y) \in X^2 \mid x \dashrightarrow y\}.$$

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