

# Complexité : $P$ v.s. $NP^*$

## Un problème qui peut rapporter des millions

PAR OLIVIER RUATTA

Université de Limoges - XLIM

*Email* : olivier.ruatta@unilim.fr

### Résumé

Ce cours est la deuxième partie du module « Calculabilité, complexité et évaluation de performances » de la première année du Master Cryptis. Dans la première partie du cours, on a répondu aux questions « Qu'est-ce que calculer avec une machine de calcul ? » et « Peut-on tout calculer ? ». On ne s'intéresse plus maintenant qu'à des problèmes qu'on sait décidable. On se demande si tout les problèmes présentent le même ordre de difficulté calculatoire.

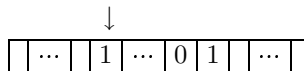
## 1 Introduction

Dans ce document, j'utiliserai librement les notions vues dans la première partie du cours, comme celles relatives aux machines de Turing. Une bonne partie de ce qui sera vue dans cette partie est tiré du livre « Algorithmes et complexité » de H. S. Wilf (publié chez Masson). On décrira quelques problèmes, qui sont souvent des problèmes de décision et des algorithmes.

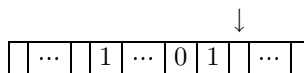
Pour commencer à pouvoir parler de complexité, nous aurons besoin de savoir donner une estimation de celle-ci. Que mesure la complexité ? Ici, nous ne nous intéresserons qu'à la complexité en temps, i.e. aux estimations du nombre d'opérations élémentaires effectuées pour sortir d'un algorithme ou pour faire tourner une machine. Il faut encore dire par rapport à quoi on mesure ! En générale, on compte en fonction de la taille de l'entrée (ce qui est une mesure de la complexité en espace que nous ne traiterons pas en détail dans ce cours). Nous commencerons par donner un exemple avant de donner des définitions plus précises.

### 1.1 Un exemple : la multiplication par 2 dans une machine de Turing

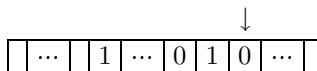
On considère une machine de Turing avec comme alphabet  $\{0, 1\}$ , un entier est donné comme entrée sous la forme de son développement en base 2. On rappelle que par convention, la tête de lecture se situe à sur la première lettre de l'entrée.



On se déplace vers la droite jusqu'à ce qu'on dépasse le mot :



On écrit alors 0 dans la case courante :



---

\*. This document has been written using the GNU  $\text{TeX}_{\text{MACS}}$  text editor (see [www.texmacs.org](http://www.texmacs.org)); Ce texte est concu pour servir de support pédagogique pour le cours de Master 1 cryptis sur la calculabilite, la complexité et l'évaluation de performances.

Puis on ramène la tête de lecture au début du mot. On va maintenant détailler, grossièrement les opérations faites au cours de ce calcul : soit  $n$  l'entier contenu sur la bande, il y a  $\log_2(n)$  case contenant de l'information au début du calcul. On se déplace de  $\log_2(n)$  case vers la droite (et si vous rappelez bien, une écriture à chaque fois), puis on fait une écriture de plus (le zéro de la fin) et on revient au début du mot, ce qui représente  $\log_2(n) + 2$  déplacement-écriture. En comptant une **unité constante de temps** élémentaire pour un déplacement-écriture, ce calcul demande  $2 * \log_2(n) + 3$  unités élémentaire de temps. On voit donc, que pour multiplier un entier par 2 dans ce modèle, il faut deux fois plus d'unité de temps (on dira d'opérations binaire) que la taille de l'entrée.

C'est le modèle le plus élémentaire et le plus réaliste, celui des machines de Turing sur  $\mathbb{F}_2$ , qui donne la complexité binaire d'un algorithme. On verra qu'on ne fait pas toujours les raisonnements sur des machines aussi « bas niveau » et que souvent on prend des alphabets plus compliqués (comme  $\mathbb{N}$ ) muni de lois de composition (dans  $\mathbb{N}$  c'est l'addition et la multiplication par exemple) et qu'on compte le même temps unitaire (ce qui est loin d'être réaliste dans le modèle binaire) pour chacune de ces lois de composition. Ces modèles sont dit « algébrique » et ils permettent d'étudier la « complexité algébrique ».

## 1.2 Un principe : comparaison asymptotique

En général, on ne s'intéresse beaucoup à comparer différents algorithmes. Surtout, on s'intéresse à la variation du temps de calcul en fonction de la taille de l'entrée. La première chose consiste à définir des notations pour capturer des propriétés asymptotiques de fonctions.

**Définition 1.** Soient  $f$  et  $g$  deux fonctions positive d'une variable réelle, s'il existe un réel  $x_0$  et une constante  $C \in \mathbb{R}_+$  tels que pour tout  $x > x_0$ , on ait  $f(x) \leq C * g(x)$ , on dit que  $f$  est un « grand  $O$  » de  $g$ . L'ensemble des fonctions qui sont des « grand  $O$  » de  $g$  est noté  $\mathcal{O}(g)$  et on note  $f \in \mathcal{O}(g)$  et parfois (souvent même) par abus de notations  $f = \mathcal{O}(g)$ .

L'ensemble  $\mathcal{O}(g)$  est « moralement » l'ensemble des fonctions majorées (à multiplication par une constante) par  $g$  quand  $x$  devient suffisamment grand.

**Définition 2.** Soient  $f$  et  $g$  deux fonctions positives d'une variable réelle, s'il existe un réel  $x_0$  et une constante  $C \in \mathbb{R}_+$  tels que pour tout  $x > x_0$ , on ait  $f(x) \geq C * g(x)$ , on dit que  $f$  est un « grand oméga » de  $g$ . L'ensemble des fonctions qui sont des « grand oméga » de  $g$  est noté  $\Omega(g)$  et on note  $f \in \Omega(g)$  et souvent par abus de notations  $f = \Omega(g)$ .

Deux types de croissances vont particulièrement nous intéresser, les fonctions  $f$  tels qu'il existe un entier  $k$  pour lequel  $f \in \mathcal{O}(x^k)$  qui sont dites à croissance polynômiale, et les fonctions qui sont à croissance exponentielle, i.e.  $f \in \mathcal{O}(e^{k*x})$  qui sont « au-dessus » de toute fonction polynômiale, i.e. pour tout  $k \in \mathbb{N}$ , on a  $f \in \Omega(x^k)$ . Il est clair que  $\mathcal{O}(1) \subsetneq \mathcal{O}(x) \subsetneq \mathcal{O}(x^2) \subsetneq \dots \subsetneq \mathcal{O}(x^k) \subsetneq \dots \subsetneq \mathcal{O}(e^{l*x})$ . On comprend aussi qu'il y a un vrai décalage entre les fonctions à croissance polynômiale et les fonctions à croissance exponentielle non polynômiale puisque une fonction de cette dernière classe majore toutes les fonctions à croissance polynômiale !

**Paradigme :** Les algorithmes **efficaces** sont les algorithmes dont la complexité est à croissance polynômiale, i.e. la fonction de complexité est dans un  $\mathcal{O}(x^k)$  pour un certain entier  $k$ .

## 1.3 Complexité intrinsèque

Ce point de vu sur la complexité veut ne considérer que le problème et pas les algorithmes qui cherche à résoudre le problème (même si on ne s'intéresse ici qu'à des problèmes pour lesquels on a un algorithme de résolution). L'objet de ce sujet est de trouver une minoration de la complexité de toute algorithme résolvant ce problème. Ce peut être le cas avec la taille de la sortie ! Il faut bien écrire la sortie. Par exemple pour le produit de deux entiers  $n$  et  $m$  sur une machine de Turing binaire, la taille de la sortie est  $\log_2(n * m) = \log_2(n) + \log_2(m)$  et, par conséquent, une machine de Turing binaire calculant le produit de deux entiers à au moins une complexité linéaire en l'entrée (puisque'il faut bien écrire la sortie et lire l'entrée !).

Il y a des tas de méthodes et bien peu de résultats dans ce domaine. C'est sans doute l'un des plus difficile de l'informatique théorique. S'il n'est pas très difficile de majorer la complexité d'un problème, il est autrement difficile de la minorer.

## 1.4 Ce qui va suivre

Dans ce qui va suivre, on va voir des problèmes pour lesquels on ne connaît pas d'algorithme qui les résolve en temps polynômial ! On verra qu'il a des problèmes ainsi qui on une certaine « propriété universelle ». On étudiera des outils assez systématiques pour étudier la fonction de complexité d'un algorithme dans une prochaine partie du cours.

## 2 La classe $P$

La classe  $P$  est la classe des problèmes qui « se résolvent facilement ».

**Définition 3.** *Un problème est dans la classe  $P$  si il existe un algorithme  $A$  et une constante entière  $c$  tels que pour toute instance  $I$  représentée par  $B$  bits, l'algorithme résoud cette instance en  $\mathcal{O}(B^c)$  opérations binaires.*

En fait on peut facilement prendre des opération sur les entiers comme l'addition, la multiplication car ces opérations sont polynômiales et qu'on ne changera pas la nature de la complexité, on aura juste faussé l'exposant.

**Exemple 4.** Décider si un entier est pair est un problème qui est dans  $P$ .

**Exemple 5.** Soient  $n$  et  $m \in \mathbb{N}$ , décider si  $n \mid m$  est un problème qui est dans  $P$ .

## 3 La classe $NP$

### 3.1 Définition de la classe $NP$

Cette classe est plus subtile. On a déjà vu ce qu'est le langage associé à une machine de Turing (l'ensemble des mots pour lesquels la machine de Turing s'arrête dans un état acceptant). Un problème de décision revient à décider si un mot fait partie d'un langage donné. On désigne souvent le problème de savoir si un mot appartient à un langage  $Q$  par la même lettre  $Q$ . Soit  $Q$  un problème de décision, il est dans la classe  $NP$  s'il existe un algorithme  $A$  tel que :

- a) à chaque mot du langage  $Q$  (i.e. à chaque instance  $I$  tel que  $I \in Q$ , i.e. les mots  $I$  pour lesquels la réponse est « oui ») un certificat  $C(I)$  est associé tel que si la pair  $(I, C(I))$  est donné en entrée de  $A$ , celui-ci reconnaît que  $I \in Q$  ;
- b) si  $I$  est un mot qui n'appartient pas à  $Q$ , alors il n'existe aucun certificat tel que  $A$  reconnaisse  $I$  ;

c) l'algorithme  $A$ , s'arrête en temps polynômial.

Pour être clair, la classe  $NP$  est la classe des problèmes de décision pour lesquels il est facile de vérifier qu'une solution est correcte. Ici, on ne parle pas de trouver une solution, mais uniquement de la vérifier.

On a clairement,  $P \subset NP$  puisque pour vérifier pour un problème dans  $P$ , qu'une entrée est solution, on résout le problème en temps polynômiale et on vérifie que l'entrée figure dans les solutions. L'inclusion dans l'autre sens est un problème ouvert (qui peut rapporter des millions). La communauté des informaticiens semblent penser que  $P \subsetneq NP$ , mais aucune preuve n'existe.

Pour illustrer voici un exemple : on considère le problème de savoir si un entier  $M$  est le produit de deux autres entiers. Il s'agit du problème de factorisation qui est réputé difficile, il est facile si je vous donne deux entiers  $n$  et  $m$  en vous disant que  $M = n * m$ , de vérifier le résultat en calculant le produit dont on a vu que ça se faisait en temps polynômial.

### 3.2 Quelques exemples de problèmes dans $NP$

On considère un graphe  $G = (S, A)$ , un  $K$ -coloriage (en générale c'est un entier qui représente le nombre de couleurs avec lesquelles on peut colorier, généralement on numérote les couleur de 1 à  $K$  est on utilise les indices des couleurs pour colorier) admissible de  $G$  est une application  $\varphi: S \rightarrow K$  tel que si  $s_1$  et  $s_2 \in S$  sont tels que  $(s_1, s_2) \in A$  alors  $\varphi(s_1) \neq \varphi(s_2)$ . Autrement, c'est l'attributions de couleurs aux sommets (pas plus de  $K$  couleurs) tels que deux sommets adjacents n'ont pas même couleur. Le problème de trouver, pour tout entier  $K$  et tout graphe  $G$ , un  $K$ -coloriage de  $G$  est un problème  $NP$ . En effet, si je vous donne une application  $\varphi$ , pour vérifier que c'est un coloriage admissible il vous suffit de vérifier que pour tout  $(s, t) \in A$ ,  $\varphi(s) \neq \varphi(t)$ . Pour ce problème-ci, on ne connaît pas d'algorithme calculant une application  $\varphi$  étant donné  $G$  et  $K$  en temps polynômiale.

## 4 $NP$ -complétude

Les problème dit  $NP$ -complet sont des problèmes  $NP$  qui ont une forme de propriété universel : si on sait résoudre un problème  $NP$ -complet, alors on sait tous les résoudre avec essentiellement la même complexité.

### 4.1 Réduction de problèmes

**Définition 6.** Soient  $Q'$  et  $Q$  deux problèmes de décision, on dit que  $Q'$  se réduit polynômialement en  $Q$  s'il existe un algorithme  $A$  tel que si  $I'$  est une instance de  $Q'$ , l'algorithme transforme cette instance en temps polynômiale en une instance  $I$  du problème  $Q$ , ces instances étant telles que  $I' \in Q' \iff I \in Q$ .

### 4.2 Problèmes $NP$ -complets

**Définition 7.** Un problème  $Q$  est dit  **$NP$ -complet** s'il est dans  $NP$  et si tout problème  $Q'$  dans  $NP$  se réduit polynômialement dans  $Q$ .

Donc si on sait résoudre un problème  $NP$ -complet en temps polynômiale, on sait résoudre tous les problèmes de la classe  $NP$  en temps polynômiale et on a montré que  $P = NP$ . Par contre si on montre qu'un problème  $NP$  n'admet pas d'algorithme de résolution en temps polynômiale, alors on a montré que  $P \neq NP$  et qu'aucun problème  $NP$ -complet n'admet d'algorithme de résolution en temps polnômiale.

Dans la sous-section suivante, nous montrons qu'il existe un problème (en fait il y en a des tas) qui est *NP*-complet.

### 4.3 Le théorème de Cook

En 1971, Cook a montré que le problème de satisfaisabilité est *NP*-complet. Dans cette sous-section, nous allons définir le problème de satisfaisabilité et nous montrerons qu'il est *NP*-complet.

On considère des variables sur  $\mathbb{F}_2 : x_1, \dots, x_n$ . On appelle **coefficient littéral** une de ces variables  $x_i$  ou sa négation  $\bar{x}_i$ . Si on a  $n$  variables, on a  $2 * n$  coefficients littéraux. En générale on associe à 1 la valeur « vraie » booléenne et 0 la valeur « fausse » booléenne. Ainsi, si  $x_i = 1$ ,  $\bar{x}_i = 0$  et de façon plus générale  $\bar{x}_i = 1 + x_i$ .

Une **clause** est un ensemble de coefficients littéraux. A chaque affectation de valeurs aux variables, les clauses héritent également d'une valeur déterminée de la façon suivante : une clause prend la valeur 1 si au moins un de ses coefficients littéraux prend la valeur 1 et 0 sinon. Une clause est donc le « ou » de tous les littéraux qui la forme.

**Définition 8.** *Un ensemble de clauses est satisfaisable s'il existe une affectation des variables qui rendent toutes les clauses de l'ensemble vraies.*

Pour qu'un ensemble de clauses soit satisfaisable il suffit de le « et » de toutes les clauses le soit !

**Problème de satisfaisabilité (SAT) :** Soit un ensemble de clause. Existe-t-il une affectation pour chaque variable tel que toute les clauses soient vraies ?

**Remarque 9.** Pour qu'une clause soit satisfaisable, il faut qu'elle ne contienne pas un littéral et sa négation. Enfin, un littéral n'apparait qu'une fois par clause.

On ne connaît pas d'algorithme plus performant que de tester successivement toutes les affectations possibles pour les variables (enfin, toutes les améliorations proposées ont des cas où elles ne sont pas meilleures). Ce qui donne  $2^n$  cas à traiter, i.e.  $e^{\log(2)*n}$  ce qui est bien à croissance exponentielle. Néanmoins, ce problème est *NP*. En effet, si on vous donne une affectation des variable, il est facile de voir (en temps polynômiale) si toutes les clauses sont satisfaites.

**Théorème 10. [ de Cook ] :** *Le problème SAT est NP-complet.*

**Démonstration.** On veut montrer que toute instance d'un problème *NP* se réduit en temps polynômiale en une instance du problème SAT. Soit  $Q$  un problème de la classe *NP* et soit  $I$  une de ces instances. Alors il existe une machine de Turing reconnaissant des instances codées de  $Q$ , si elles sont accompagnées d'un certificat, en temps polynômial. Soit  $\mathcal{T}_Q$  une telle machine et  $P(n)$  le polynôme tel que  $\mathcal{T}_Q$  reconnaisse  $(I, C(I))$ , où  $I$  est de longueur  $n$ , en un temps inférieur à  $P(n)$ .

Pour chaque instance  $I$  de  $Q$  on va construire une instance  $f(I)$  de SAT telle que toutes les classes de  $f(I)$  sont satisfaites si et seulement  $I$  est un mot du langage  $Q$ .

L'idée est de construire une une instance de SAT telle que l'ensemble des clauses expriment le fait qu'il existe un certificat qui fait faire à  $\mathcal{T}_Q$  un calcul affirmatif. Par conséquent, il suffira de vérifier que l'ensemble des clauses est satisfaisable.

Pour construire une instance de SAT, nous allons définir des variables, des coefficients littéraux et des clauses, de manière à ce que les clauses soient satisfaisable si et seulement si  $I \in Q$ , i.e. si la machine  $\mathcal{T}_Q$  reconnaît  $I$  et son certificat.

Un calcul affirmatif de  $\mathcal{T}_Q$  doivent être interprété comme la vérification simultanée d'un certain nombre de clauses.

Commençons par décrire les variables :  $Q_{i,k}$  est vraie si, après le pas  $i$  de la vérification,  $\mathcal{T}_Q$  est dans l'état  $q_k$  et fausse sinon ;  $s_{i,j,k}$  est vraie si, après le pas  $i$  de la vérification, le symbole  $k$  est dans la  $j$ -ième case de la bande et fausse sinon ;  $T_{i,j}$  est vraie si, après le pas  $i$  de la vérification, la tête de lecture est sur la case  $j$  et fausse sinon.

Comptons les variables introduites : puisque  $\mathcal{T}_Q$  effectue la vérification en un temps inférieur à  $P(n)$ , la tête ne peut pas aller sur plus de  $P(n)$  cases différentes, par conséquent  $j$  ne peut prendre plus de  $\mathcal{O}(P(n))$  valeurs. L'indice  $i$  décrit les différentes étapes du calcul et il ne peut prendre plus de  $\mathcal{O}(P(n))$  valeurs distinctes. Enfin, l'indice  $k$  indique les états de la machine qui est un nombre fini fixé  $K$ . Finalement, il y a  $\mathcal{O}(P(n)^2)$  variables, ce qui est polynômial.

Toutes les affectations ne correspondent pas à un calcul affirmatif de  $(x, C(x))$ . Beaucoup, comme ceux conduisant à un déplacement sur la bande de plus de 1 case à la fois correspondent à des arrêts indéterminés de la machine.

On doit maintenant préciser comment l'ensemble des valeurs affectées aux variables permet un calcul affirmatif possible pour  $(x, C(x))$ . Nous saurons alors que pour tout ensemble de valeurs satisfaisantes des variables choisies pour résoudre SAT, celles-ci donneront un calcul affirmatif sur la machine  $\mathcal{T}_Q$ . Chaque clause exprimant une condition **nécessaire** devra être satisfaite. Voici les conditions nécessaires et les clauses correspondantes :

- A chaque pas, la machine est dans au moins un état, comme il y a  $K$  états possibles il faut que pour un  $i$  au moins et pour un  $l \in \{1, \dots, K\}$ , le coefficient littéral  $Q_{i,l}$  soit vraie, ce qui nous donne les clauses suivantes :  $\forall i, \{Q_{i,1}, \dots, Q_{i,K}\}$ . Il y a  $\mathcal{O}(P(n))$  telles clauses puisqu'il y a  $\mathcal{O}(P(n))$  valeurs possibles pour  $i$ .
- A chaque pas la machine est dans au plus un état : ce qui nous donne les clauses suivantes pour tous les  $l$  et  $l' \in \{1, \dots, K\}$ ,  $l \neq l'$ , et pour chaque  $i$  on a la clause  $\{\overline{Q_{i,l}}, \overline{Q_{i,l'}}\}$  doit être vraie. Il y a  $\mathcal{O}(K^2 * P(n)) = \mathcal{O}(P(n))$  telles clauses.
- A chaque pas, chaque case de la bande contient au plus un symbole : ce qui nous donne deux listes de clauses comme pour les états :  $\forall i$  et  $j$ ,  $\{s_{i,j,1}, s_{i,j,2}, \dots, s_{i,j,r}\}$  où  $r$  est le cardinal de l'alphabet (qui ne dépend pas de la taille de l'entrée) et pour  $l \neq l' \in \{1, \dots, r\}$ ,  $\{\overline{s_{i,j,l}}, \overline{s_{i,j,l'}}\}$ . Ce qui nous donne  $\mathcal{O}(P(n)^2)$  telles clauses.
- A chaque pas la tête est positionnée sur une case et une seule :  $\forall i, \{T_{i,1}, \dots, T_{i,L}\}$  (où  $L < C * P(n)$ ) pour une certaine constante entière  $C$  et pour  $l \neq l', \{\overline{T_{i,l}}, \overline{T_{i,l'}}\}$ , ce qui nous donne  $\mathcal{O}(P(n)^2)$  telles clauses.
- Initialement la machine est dans l'état  $q_0$ , la tête sera par convention sur la case 1, le  $x$  d'entrée est dans les cases numérotées de 1 à  $n$  et  $C(x)$  à la suite de  $n+2$  à  $n+2+P(n)$ . L'expression des clauses est laissée en exercice.
- A pas  $P(n)$ , la machine est dans l'état  $q_Y$ .
- A chaque pas, la machine prend sa nouvelle configuration (état, symbole, position de la tête) en accord avec l'application de son unité centrale et de sa configuration précédente (état, symbole, position de la tête) : Pour trouver les clauses, commençons par le symbole dans la case  $j$  ne change pas pendant le  $i$ -ième pas du calcul si la tête n'est pas positionnée dessus qui donne les clauses  $\{T_{i,j}, \overline{S_{i,j,k}}, S_{i,j,k}\}$  pour chaque  $(i, j, k) = (\text{étape}, \text{position}, \text{symbole})$ . Il y a  $\mathcal{O}(P(n)^2)$  telles clauses.

Il reste à exprimer le fait que les transitions d'une configuration à une autre résultent de l'unité centrale :  $\{\overline{T_{i,j}}, \overline{Q_{i,k}}, \overline{s_{i,j,l}}, T_{i+1,j+\text{inc}}\}$ ,  $\{\overline{T_{i,j}}, \overline{Q_{i,k}}, \overline{s_{i,j,l}}, Q_{i+1,k'}\}$ ,  $\{\overline{T_{i,j}}, \overline{Q_{i,k}}, \overline{s_{i,j,l}}, s_{i+1,j,l'}\}$ . Ces clauses expriment des conditions de la forme « si la tête n'est pas positionnée sur la case  $j$  à l'étape  $i$  soit l'état n'est pas  $q_k$ , soit le symbole lu n'est pas  $l$ , mais si c'est le cas alors ... ». Il existe une telle clause pour chaque case de la table de transition de  $\mathcal{T}_Q$ . Il y en a un nombre polynômial ( $\mathcal{O}(P(n)^2)$ ).

Nous avons construit un ensemble de clauses qui ont la propriété que si on exécute le calcul de reconnaissance de  $x$  et son certificat, en un nombre d'étape au plus  $P(n)$  la valeur de chaque variable ci-dessus est déterminée et donc que les clauses sont satisfaites si et seulement si  $(x, C(x))$  est reconnu par  $\mathcal{T}_Q$ .

Ainsi, tout problème de  $NP$  peut-être polynômialement réduit dans SAT et SAT est  $NP$ -complet.  $\square$

Donc nous connaissons maintenant un problème  $NP$ -complet et pour montrer qu'un autre problème est  $NP$ -complet, on montrera maintenant qu'on sait réduire polynômialement SAT dans notre problème. Nous donnons dans la suite un exemple de réduction polynômiale explicite et puis nous donnerons quelques exemples de problèmes  $NP$ -complets (on en connaît des centaines). Enfin, on parlera de deux problèmes d'intérêt cryptographique qui sont  $NP$  et dont on sait pas s'ils sont  $NP$ -complet.

## 5 Quelques exemples et illustrations

Dans cette section, nous allons donner des exemples de problèmes  $NP$ -complet, mais aussi illustrer quelques pièges classiques associés à cette notion.

### 5.1 Le problème 3SAT

Le problème de « 3-satisfaisabilité », appelé 3SAT est un cas particulier de SAT qui a la particularité d'être  $NP$ -complet. Comme pour SAT, les instances de 3SAT sont formés de clauses, mais dans 3SAT chaque clause contient au plus trois littéraux. La question est, comme pour SAT, de savoir s'il y a une affectation des variables telle que toutes les clauses de l'instance soient satisfaites.

**Théorème 11.** *Le problème 3SAT est  $NP$ -complet.*

**Démonstration.** Soit une instance de SAT, nous allons montrer comment réduire en temps polynômial cette instance en une instance de 3SAT telle que cette dernière soit satisfaisable si et seulement la première l'est également.

En fait, on va se contenter de montrer comment remplacer une clause contenant plus de trois coefficients littéraux en clauses contenant au plus trois coefficients littéraux. Supposons que l'instance de SAT contienne une clause  $\{x_1, x_2, \dots, x_k\}$ , avec  $k \geq 4$ . On remplace cette clause par les  $k - 2$  clauses suivantes  $\{x_1, x_2, x_3\}, \{x_3, \bar{z}_1, z_2\}, \{x_4, \bar{z}_2, z_3\}, \dots, \{x_{k-1}, x_k, z_{k-3}\}$ .

On va maintenant montrer que s'il existe une affectation des  $x_i$  rendant la première clause satisfaite, alors il existe une affectation des  $x_i$  et des  $z_j$  rendant l'ensemble des 3-clauses satisfaites. Pour que la première clause soit satisfaite, il faut qu'un au moins des  $x_1$  ait la valeur 1. Disons que  $x_r$  a la valeur 1. On peut alors satisfaire toutes les 3-clauses de la façon suivante : on pose  $z_k \leftarrow 1$  pour  $k \leq r - 2$  et  $z_k \leftarrow 0$  pour  $k > r - 2$ . La vérification est directe.

Réciproquement, montrons que si toutes les 3-clauses sont satisfaites, alors la première clause l'est. Il suffit de montrer qu'un au moins des  $x_i$  a 1 comme affectation. Supposons qu'il ait une affectation des  $x_i$  et  $z_j$  satisfaisant toutes les 3-clauses et telle que tous les  $x_i$  soient affectés à 0. Alors on peut oublier les  $x_i$  dans les 3-clauses et on a toutes les clauses suivantes qui sont satisfaites :  $\{z_1\}, \{\bar{z}_1, z_2\}, \{\bar{z}_2, z_3\}, \dots, \{\bar{z}_{k-4}, z_{k-3}\}, \{\bar{z}_{k-3}\}$ . Comme toutes ces clauses doivent être satisfaites, il faut que  $z_1 = 1$ , ce qui implique que  $z_2 = 1, \dots$ , jusqu'à  $z_{k-3} = 1$ , mais la dernière clause impose que  $z_{k-3} = 0$ , ce qui nous conduit à une contradiction. Ainsi, les 3-clauses sont toutes satisfaites implique qu'un au moins des  $x_i$  soit affecté à 1 et donc ceci montre que l'ensemble des 3-clauses est « équivalent » à la clause de départ.

Comme on multiplie chaque clause par au plus nombre de littéraux moins deux, on a un nombre polynômial de 3-clauses. Ainsi SAT se réduit polynômialement en 3-SAT et 3-SAT est  $NP$ -complet.  $\square$

**Remarque 12.** On pourrait croire que 3SAT est un ensemble d'instance réduit de SAT, mais en fait ce sous-problème est aussi général que SAT au sens de la complexité. Ce qu'il faut comprendre c'est que le sous-problème 3SAT représente un sous-ensemble d'instance de SAT suffisamment gros pour qu'il capture toute la complexité du problème. Ce n'est pas toujours le cas. Par exemple, le problème 2SAT est dans la classe  $P$ .

## 5.2 Le problème de colorabilité des graphes

On a déjà rencontré ici le problème de colorabilité. Ce problème est équivalent à ce donner un graphe et un entier  $n$  et à ce demander si le graphe est  $n$ -partie (chaque partie de la partition correspondra à une couleur).

**Théorème 13.** *Le problème de coloriage des sommets d'un graphe est NP-complet.*

**Démonstration.** On va réduire en temps polynômial le problème 3SAT dans un problème de coloriage de graphe. Soit une instance de 3SAT, i.e. un ensemble de  $k$  clauses faisant intervenir  $n$  variables. On va construire un graphe qui sera  $n + 1$  coloriable si et seulement si l'instance de 3SAT peut être satisfaite. On peut considérer sans perte de généralité que  $n > 4$ .

Le graphe aura  $3n + k$  sommets :  $\{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n, y_1, \dots, y_n, C_1, \dots, C_k\}$ .

Pour les arêtes : chaque sommet  $x_i$  est relié à  $\bar{x}_i$ , chaque  $y_i$  est relié à  $y_j$  tels que  $i \neq j$ , à tous les  $x_j$  tel que  $i \neq j$  et tous les  $\bar{x}_j$  tels que  $j \neq i$ . Le sommet  $x_i$  est relié à  $C_j$  si  $x_i$  n'est pas un littéral de la  $j$ -ème clause. De même,  $\bar{x}_i$  est relié à  $C_j$  si  $\bar{x}_i$  n'est pas un littéral de la  $j$ -ème clause.

Montrons maintenant que le graphe que nous avons construit est  $n + 1$ -coloriable si et seulement si toutes les clauses peuvent être satisfaites. Il est évident que  $G$  ne peut être colorié en moins de  $n$  couleurs car les  $y_i$  sont tous reliés entre eux. Numérotions  $n + 1$  couleurs. On donne à  $y_i$  la couleur  $i$ .

Puisque  $y_i$  est relié à chaque  $x_j$  et  $\bar{x}_j$ ,  $j \neq i$ , on peut utiliser la couleur  $i$  pour  $x_i$  ou  $\bar{x}_i$ , mais pas les deux. On doit donc avoir au moins  $n + 1$  couleurs. La seule façon de continuer à colorier le graphe consiste à donner la couleur  $n + 1$  à un membre de chaque pair  $\{x_i, \bar{x}_i\}$ . Le sommet qui reçoit la couleur  $n + 1$  est appelé sommet *faux* et l'autre sommet *vrai*. Il reste à colorier les sommets  $C_1, \dots, C_k$ . Le graphe sera  $n + 1$  coloriable si et seulement si on peut le faire sans utiliser de nouvelles couleurs. Puisque chaque clause contient 3 littéraux et que  $n > 4$ , chaque  $C_i$  est relié à  $x_i$  et  $\bar{x}_i$  pour au moins une valeur de  $j$ . Par conséquent,  $C_i$  ne peut avoir la couleur  $n + 1$ . Donc aucun  $C_i$  n'a la couleur  $n + 1$ . Cette couleur doit être choisie parmi  $1, 2, \dots, n$ .

Puisque  $C_i$  est relié à chaque  $x_j$  ou  $\bar{x}_j$ , il ne peut être de la couleur de ceux-ci. Ainsi, la couleur de  $C_j$  doit être la même que celle d'un *vrai*  $x_i$  ou  $\bar{x}_i$ , correspondant à un littéral de la  $j$ -ième clause. Ainsi, le graphe est  $n + 1$ -coloriable si et seulement si il existe un sommet *vrai* pour chaque  $C_i$  et par conséquent si et seulement si chaque clause peut être satisfaite. Il n'est pas difficile de vérifier que cette réduction est polynômiale.  $\square$

## 5.3 Quelques autres exemples

**Clique maximum :** Soit  $G$  un graphe et  $K$  un entier. Existe-t-il un ensemble de  $K$  sommets de  $G$  qui soient tous reliés entre eux par des sommets de  $G$  ?

**Coloriage des arêtes :** Soit  $G$  un graphe et  $K$  un entier. Existe-t-il un coloriage des arêtes de  $G$ , avec  $K$  couleurs, de telle sorte que deux arêtes adjacentes soient de couleurs différentes ? (prendre le graphe dual de  $G$  et se demander s'il est  $K$ -coloriable).

## 6 Exercices

### 6.1 Manipulation des notations d'asymptotique

1. Soient  $f$ ,  $g$  et  $h$  des fonctions positives d'une variable réelle. Montrer que si  $f \in \mathcal{O}(h)$  et  $g \in \mathcal{O}(h)$ , alors  $f + g \in \mathcal{O}(h)$ . Montrer que si  $f \in \mathcal{O}(h)$  et  $g \in \mathcal{O}(h)$ , alors  $fg \in \mathcal{O}(h^2)$ . Montrer que les fonctions à croissance polynômiale forment une  $\mathbb{R}$ -algèbre pour les lois d'addition et de multiplication des fonctions.



2. Soient  $f, g$  et  $h$  des fonctions positives d'une variable réelle. Montrer que si  $f \in \mathcal{O}(g)$  et  $g \in \mathcal{O}(h)$ , alors  $f \in \mathcal{O}(h)$ .

## 6.2 Quelques calculs de complexité binaire

1. Soit  $N$  et  $M$  deux entiers de taille respective  $n = \log_2(N)$  et  $m = \log_2(M)$ , montrer que l'addition de  $M$  et  $N$  sur une machine binaire a une fonction de complexité dans  $\mathcal{O}(\max(n, m))$ .
2. Soit  $M > 0$  un entier. Décrivez une machine de Turing binaire calculant  $M - 1$ . Quelle est la complexité asymptotique de ce calcul.
3. Soit  $N$  et  $M$  deux entiers de taille respective  $n = \log_2(N)$  et  $m = \log_2(M)$ . Décrire sommairement une machine de Turing binaire à trois bandes calculant  $N * M$  en simulant l'algorithme suivant :

```

a ← M; b ← M; c ← N;
tant que c ≠ 0 faire
  a ← a+b;
  c ← c-1;
fin faire;

```

En déduire que la complexité binaire asymptotique du produit d'entier est  $\mathcal{O}(n * \max(n, m))$ .

En fait, la complexité binaire du produit d'entier est connu pour être en  $\mathcal{O}((n + m)\log(n + m))$ .

## 6.3 Complexité binaire v.s. complexité algébrique

1. Montrer que si les opérations de somme, de produit, ainsi que le teste de signe et le teste à zéro peuvent être calculés par des machines de Turing sur  $\{0, 1\}$  en temps polynômiale de la taille des entiers en entrée et que si  $\mathcal{T}$  est une machine de Turing sur  $\mathbb{Z}$  ne faisant que des opérations d'addition et de produit qui calcule en temps polynômiale de sont entrée alors  $\mathcal{T}$  est équivalente à une machine de Turing sur  $\{0, 1\}$  qui calcule en temps polynômial.
2. Montrer que la différence de deux entiers est calculable par une machine de Turing binaire en temps polynômial.
3. Montrer que la division euclidienne de deux entiers a une complexité binaire polynômiale.

## 6.4 Traduction algébrique de 3SAT

1. Exprimer le « ou inclusif » logique en fonction du « et » et du « ou exclusif ». Remarquez que le « et » correspond à la multiplication dans  $\mathbb{F}_2$  et que le « ou exclusif » correspond au produit. On note  $\oplus$  l'addition et  $\otimes$  la multiplication dans  $\mathbb{F}_2$ .
2. Montrer qu'une clause de la forme  $\{x_1, \bar{x}_2, x_3\}$  se traduit par le polynôme  $1 \oplus x_2 \oplus (x_1 \otimes x_2) \oplus (x_2 \otimes x_3) \oplus (x_1 \otimes x_2 \oplus x_3)$ .
3. Déduisez-en que toute instance du problème 3SAT se réduit polynômialement en un problème de résolution d'un système polynômial.
4. Montrer que la résolution de systèmes polynômiaux à coefficients dans  $\mathbb{F}_2$  est un problème  $NP$ -complet.

## 6.5 Systèmes algébriques

Soit  $\alpha = (\alpha_1, \dots, \alpha_k) \in \{0, 1\}^k$  tel que  $|\alpha| = \sum_{i=1}^k \alpha_i = 3$ . Soit  $p \in \mathbb{Z}$ , pour tout  $n \in \mathbb{Z}$ , on note  $\hat{n}$  le reste de la division euclidienne de  $n$  par  $p$ . Soit  $P(x_1, \dots, x_k) = \sum_{|\alpha|=3} a_\alpha x_1^{\alpha_1} \cdots x_k^{\alpha_k} \in \mathbb{Z}[x_1, \dots, x_k]$ .

Montrer que si  $(z_1, \dots, z_k) \in \mathbb{Z}^k$  est tel que  $P(z_1, \dots, z_k) = 0$ , alors  $(\hat{z}_1, \dots, \hat{z}_k) \in \mathbb{F}_p^k$  est un zéro de  $\hat{P}(x_1, \dots, x_k) = \sum_{|\alpha|=3} \hat{a}_\alpha x_1^{\alpha_1} \cdots x_k^{\alpha_k} \in \mathbb{F}_p[x_1, \dots, x_k]$ .

Montrer que le problème de l'existence de solution de systèmes de polynômes à coefficients entiers est  $NP$ -complet.

## 6.6 Quelques majorations utiles

1. Montrer que  $\sum_{i=1}^n i^m \in \mathcal{O}(i^{m+1})$ .
2. On considère la suite définie par  $T(n+1) = K + T(n)$ , où  $K \in \mathbb{N}$ . Donner un équivalent asymptotique de la fonction  $n \rightarrow T(n)$ .
3. Soit  $f$  une fonction positive strictement croissante. Montrer que  $\sum_{i=0}^n f(i) \in \mathcal{O}(n * f(n))$ .