# La complexité du problème *TETRIS*

Texte tiré des *Notes de la huitième Brussels Summer School of Mathematics*

**Juillet 2016**

Keno Merckx *

**Résumé**

Le but de ce texte est de formaliser le concept de la difficulté "mathématique" d'un problème et de donner une introduction non formelle à la théorie de la complexité. A cette fin, nous utiliserons le célèbre jeu vidéo Tetris pour illustrer les notions de "langage", de "problème", de "classe de complexité" et de "réduction polynomiale". Nous aborderons également le fameux problème $\mathcal{P} = \mathcal{NP}$. Ce travail sera divisé en plusieurs sections. Premièrement, nous discuterons de ce qu'est un problème d'un point de vue mathématique. Deuxièmement, nous observerons le jeu Tetris pour définir le problème *TETRIS*. Ensuite, nous établirons une notion consistante de complexité théorique que nous pourrons utiliser pour analyser le problème *TETRIS*.



*Université Libre de Bruxelles

# 1 Introduction

Nous allons ici aborder différentes définitions et notions propres à la théorie de la complexité. Notre but n'est pas de rédiger un document encyclopédique contenant l'ensemble des choses connues dans ce domaine. L'idée est plutôt de découvrir et d'entrapercevoir cette théorie en faisant un compromis entre formalisme et accessibilité. Cet aperçu sera centré autour de la question suivante : est-ce que le jeu Tetris est difficile ? La ligne directrice de ce travail suit en partie le raisonnement mis en place par Demaine, Hohenberger et Liben-Nowell [8] dans leur article sur la difficulté du Tetris. Le lecteur intéressé pourra aisément approfondir les concepts énoncés ci-dessous dans l'excellent livre de Sipser [11]. Se trouvent également dans ce livre les preuves des théorèmes généraux de complexité que nous ne démontrons pas dans ce travail.

## La difficulté

Si l'on nous donne un problème, il n'est pas toujours aisé d'estimer sa difficulté avant de le résoudre. De plus, selon les aptitudes de chacun, il est évident que la notion de difficulté est très subjective. Par exemple, voici une liste de tâches à réaliser :

- corriger des copies d'examens ;
- trier des dossiers par ordre alphabétique ;
- factoriser des nombres ;
- préparer un sac à dos avant un départ en vacances ;
- gagner une partie d'échec contre une intelligence artificielle ;
- réaliser une thèse de doctorat.

Pour affirmer qu'une de ces tâches est "plus dure" qu'une autre il nous faut formaliser un certain nombre d'idées, notamment le concept de difficulté. L'accent sera mis ici sur l'aspect algorithmique de la résolution d'un problème donné. Nous allons entrevoir ce que l'on nomme la *théorie de la complexité*, c'est un domaine à cheval entre les mathématiques et l'informatique théorique qui vise à quantifier les ressources nécessaires à la résolution d'un problème. Cette théorie a de nombreux débouchés théoriques et pratiques. De manière générale, l'étude de la complexité d'un problème apporte souvent une nouvelle vue sur celui-ci. Notons que dans les faits, la théorie est souvent en avance sur la pratique (voir le cas de la cryptographie post-quantique). Précisons également que les questions de théorie de la complexité ont aussi un intérêt en dehors de l'informatique comme expliqué dans l'article de Aaronson [1].

## Quelques définitions

Nous allons devoir commencer avec quelques définitions standards. Soit $\Sigma^*$ un *alphabet*, c'est-à-dire un ensemble d'éléments appelés *symboles*. Appelons un *mot*, la concaténation d'un nombre fini de symboles. Un *langage* $\Sigma$ sera un ensemble de mots. Par exemple, nous avons l'alphabet composé des chiffres : $\Sigma^* = \{0, 1, 2, 3, 4, 5, 6, 7, 9\}$, nous définissons le langage suivant :

$$\Sigma = PREMIER = \{x : x \text{ est un nombre premier}\}.$$

Une fois que nous avons cette notion de langage, nous pouvons parler formellement de problème. En particulier, nous allons ici observer des *problèmes*

*de décision*, c'est-à-dire des problèmes dont la réponse est soit "oui", soit "non". Ceux-ci seront dans ce travail du type : soient $x$ un mot et $\Sigma$ un langage, est-ce que $x \in \Sigma$ ? Pour fixer les idées, observons le problème suivant, que nous nommerons problème *PREMIER*.

**Problème 1.** Étant donné un nombre $x$ donné comme la concaténation d'un nombre fini d'éléments de l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 9\}$,

est-ce que $x \in PREMIER$ ?

Les notions de langage et de problème seront donc ici très fortement liées. Nous parlerons également plus loin d'*instance* d'un problème de décision comme étant la question obtenue en spécifiant des valeurs précises pour tous les paramètres du problème. Ci-dessous un exemple d'une instance du problème de décision *PREMIER* vu plus haut.

**Exemple 1.** Soient $\Sigma^* = \{0, 1, 2, 3, 4, 5, 6, 7, 9\}$, et $\Sigma = PREMIER$.

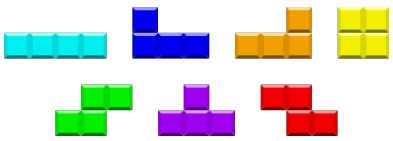
Est-ce que $2147483647 \in PREMIER$ ?

Le nombre 2147483647 est une instance du problème *PREMIER*.

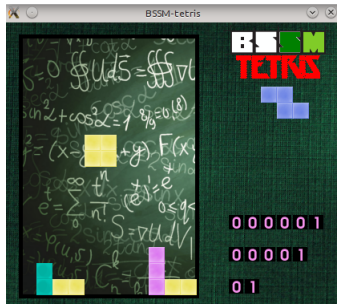
Nous parlerons d'instance *positive* si la réponse au problème de décision est "oui". Inversement, nous parlerons d'instance *négative* si la réponse est "non". Dans l'exemple précédent l'instance est positive car $2147483647 = 2^{31} - 1$ est effectivement premier.

## 2 Le Tetris

Le Tetris [12] est un jeu vidéo développé par Alexey Pajitnov en 1985 à Moscou. Le jeu se compose des Tétrominos suivant :



Le joueur doit stratégiquement translater, effectuer des rotations et descendre les Tétrominos qui tombent dans une matrice rectangulaire (appelée le *plateau*) à une certaine vitesse. Le joueur tente de vider autant de lignes que possible en complétant une ligne de la matrice sans case vide. Quand une telle ligne se crée, elle disparaît du plateau et les morceaux de Tétrominos au dessus d'elle tombent par un effet de gravité. Mais si une colonne de Tétrominos dépasse la hauteur de la matrice, le joueur a perdu. Ci-dessous un exemple d'une partie de Tetris en cours.




### Le problème TETRIS

Nous allons maintenant présenter le jeu Tetris comme un problème de décision. Bien entendu, il faudra modifier quelques règles, en particulier nous considérerons que le Tétris se joue *hors-ligne*, c'est à dire que nous voyons la séquence finie de Tétrominos qui se présentera à nous. Celle-ci sera notée $s$. Nous considérerons également que le Tétris est à un stade *avancé*, c'est-à-dire que le plateau est partiellement rempli, celui-ci est donné sous forme d'une matrice binaire notée $p$.

Avec ces éléments, nous sommes en mesure de définir un langage que nous nommerons *TETRIS*. Les mots que nous observerons seront des couples composés d'un plateau partiellement rempli $p$ et d'une séquence de Tétrominos $s$.

$$TETRIS = \{(p,s) : \text{ il est possible de jouer } s \text{ pour vider entièrement } p\}$$

Une fois ce langage défini, il nous est facile de considérer le problème *TETRIS*.

**Problème 2.** Étant donné un plateau $p$ et une séquence de Tétrominos $s$, est-il possible de vider $p$ à l'aide de $s$ ? Autrement dit :

$$\text{Soit } (p,s), \text{ est-ce que } (p,s) \in TETRIS\,?$$

**Exemple 2.** Pour un avoir une idée concrète, observons l'instance suivante du problème *TETRIS*.

Est-ce que  $\in TETRIS$ ?

Notons que l'instance ci-dessus est positive, car la séquence de Tétrominos permet de vider de tableau de jeu donné.

## 3 La complexité

La théorie de la complexité des algorithmes étudie formellement la difficulté intrinsèque des problèmes algorithmiques. Elle définit plusieurs "classes de complexité" permettant de classer les algorithmes selon leurs caractéristiques.

## Notion de complexité

Ici, nous évaluerons la difficulté d'un problème via la dépendance entre la "quantité de ressource" que demande la résolution de ce problème en fonction de la "taille" de l'instance. Avant d'arriver à des définitions, prenons une approche non formelle et observons quelques problèmes. La ressource considérée sera le temps.

**Problème 3.** Étant donné $L$ une liste de $n$ nombres, est-ce que le premier nombre de cette liste est positif ?

**Problème 4.** Étant donné $L$ une liste de $n$ nombres, est-ce que cette liste de nombres est triée de façon croissante ?

**Problème 5.** Étant donné $L$ une liste de $n$ nombres, est-ce qu'il existe une sous-liste $L'$ de $L$, telle que la somme des éléments dans $L'$ soit nulle ?

Le Problème 3 peut être rapidement résolu, il suffit de regarder le premier élément de la liste. Même si la liste $L$ possède plusieurs milliers d'éléments, il nous faudra une "étape" pour déterminer si une liste appartient bien à l'ensemble des listes dont le premier élément est positif.

Le Problème 4 sera plus compliqué, l'intuition nous indique qu'il faudra vérifier l'ensemble de la liste pour pouvoir certifier qu'elle est bien triée. Algorithmiquement, si $L = (l_1, \dots l_n)$, il faut vérifier que $l_i \leq l_{i-1}$ pour tout $i$ compris entre 1 et $n-1$. Ceci nous oblige à faire $n-1$ "étapes" pour déterminer si une liste appartient bien à l'ensemble des listes triées de façon croissante.

Le Problème 5 sera encore plus compliqué, dans le sens où nous n'avons pas vraiment d'algorithme efficace pour vérifier qu'il existe une sous-liste $L'$ de $L$, telle que la somme des éléments dans $L'$ soit nulle. A peu de chose près, il nous faudra essayer toutes les sous-liste possibles, et donc $2^n$ "étapes". Le temps dépensé pour la résolution du problème, augmente ici exponentiellement avec la taille des données.

## Formalisation de la complexité

Formellement, nous voulons mesurer la consommation en ressources d'un algorithme. Spécifions d'abord qu'en informatique théorique les deux ressources principalement considérées sont le temps et l'espace mémoire. Dans ce travail, nous allons surtout aborder le coté temporel de la complexité. Ensuite, nous allons utiliser une approche pessimiste, c'est-à-dire que pour un algorithme donné, nous considérerons la consommation de ressources dans le pire des cas en fonction de la taille des entrées fournies.

Formellement, prenons un problème de décision du type : "est-ce que $p \in L$ ?", et un algorithme $\mathcal{A}$ qui décide si $p \in L$. Nous allons décrire la complexité de $\mathcal{A}$ en deux étapes :

— lancer $\mathcal{A}$ sur toutes les instances de taille $n$ à l'aide d'une machine ;

— étudier (asymptotiquement) la fonction de $n$ par rapport à la ressource consommées par $\mathcal{A}$ sur "la pire" instance de taille $n$.

Le terme asymptotiquement désigne ici le fait que nous étudions notre fonction quand $n$ tend vers l'infini (c'est-à-dire quand $n$ est de plus en plus grand). Au plus les ressources consommées grandirons vite par rapport à $n$, au plus le problème sera *complexe*.

Nous définirons la *complexité d'un problème* comme étant la plus petite (d'un point de vue asymptotique) complexité parmi toutes les complexités d'algorithmes connus qui décident le problème.

**Exemple 3.** De façon un peu plus technique, si nous utilisons la comparaison asymptotique et que nous regardons la complexité temporelle dans le pire des cas, le Problème 3 a une complexité en $O(1)$, le Problème 4 a une complexité en $O(n)$ et le Problème 3 a une complexité en $O(2^n)$.

Il est important de remarquer que la complexité d'un problème dépend de notre capacité à trouver un algorithme "malin" pour résoudre le problème en question. Pour reprendre le Problème 5, peut-être qu'une méthode pour résoudre le problème en un nombre polynomial d'étapes verra le jour d'ici peu. Mais pour l'instant il nous est impossible de savoir si un tel algorithme existe. Il est donc possible que la complexité d'un problème évolue en fonction de nos nouvelles techniques algorithmiques.

## Classes de complexité

Une *classe de complexité* est simplement un ensemble de langages qui ont la propriété d'avoir tous la même complexité selon un certain critère. Voici quelques classes bien connues.

— la classe $\mathcal{P}$ contient les langages $L$ tels que toutes les questions $x \in L$ peuvent être décidées en un temps polynomial en la taille des données.

— la classe $\mathcal{P}$-SPACE contient les langages $L$ tels que toutes les questions $x \in L$ peuvent être décidées en un espace mémoire polynomial en la taille des données.

— la classe $\mathcal{EXP}$-TIME contient les langages $L$ tels que toutes les questions $x \in L$ peuvent être décidées en un temps exponentiel en la taille des données.

Il existe une multitude d'autres classes de complexité, beaucoup sont listées sur le site *Complexity Zoo* [5]. De plus, nous pouvons voir qu'il existe une hiérarchie entre ces ensembles comme le montre le théorème ci-dessous.

**Théorème 4.** *Nous avons les relations suivantes :*

$$\mathcal{P} \subseteq \mathcal{P}\text{-SPACE} \subseteq \mathcal{EXP}\text{-TIME},$$

*mais* $\mathcal{P} \neq \mathcal{EXP}$*-TIME.*

## Analyse du voyageur de commerce

Nous allons aborder le célèbre problème du voyageur de commerce (appelé *TSP* pour *Travelling Salesman Problem*). Observons la situation suivante : je veux me rendre dans les capitales des 28 états membres de l'Union européenne. Chaque déplacement en avion entre deux capitales me coûte un prix donné. Je ne veux jamais passer deux fois par la même ville, mais je veux revenir à ma ville de départ en fin de voyage. Existe-t-il un ordre de visite dont le prix est inférieur à 1000 euros ? Une approche naïve pour résoudre cette question :

— ordonner les villes de toutes les façons possibles ;

— pour chaque ordre, calculer le prix total du voyage ;

— vérifier si ce prix est inférieur à 1000 euros.

Nous avons ici un algorithme pour résoudre la question de notre voyage dans le vieux continent. Malheureusement, si jamais l'Union européenne s'agrandit, la première étape va prendre beaucoup plus du temps. A titre d'indication, pour 63 villes, il existe plus d'ordres de visite que le nombre estimé d'atomes dans l'univers observable. Formellement, le problème du voyageur de commerce se décrit de la manière suivante.

**Problème 6.** Étant donné un ensemble $V$ de $n$ villes, des nombres réels positifs $c_{x,y}$ pour tout $x, y \in V$, $x \neq y$ tel que $c_{x,y} = c_{y,x}$ et un nombre $M$, existe-t-il un ordonnancement des villes $(v_1, \ldots, v_n)$ tel que

$$\sum_{i=1}^{n-1} c_{v_i,v_{i+1}} + c_{n,1} \leq M ?$$

Nous avons vu une approche naïve pour résoudre ce problème, posons-nous la question de savoir si nous pouvons faire mieux, c'est-à-dire trouver un algorithme plus rapide pour trouver un ordonnancement des villes peu coûteux. Malheureusement personne n'a (encore) trouvé une solution beaucoup plus efficace pour résoudre le problème du *TSP*. Mais personne n'a prouvé qu'il faille obligatoirement un temps exponentiel en fonction des données (ici les villes et les prix des trajets) pour résoudre le problème. En fait, le problème du *TSP* est dans la classe de complexité $\mathcal{EXP}$-TIME. Ceci dit, il est possible que dans le futur, un algorithme efficace pour le problème soit trouvé et donc que *TSP* soit dans $\mathcal{P}$.

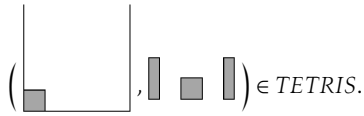
Une dernière remarque sur le problème du *TSP* dans l'Union européenne, si jamais nous recevons un ordre particulier des villes, il est "rapide" (polynomial) de vérifier si le prix du voyage est inférieur à 1000 euros. Cette constatation va permettre d'aborder la classe $\mathcal{NP}$.

## La classe $\mathcal{NP}$

Une autre façon de mesurer la complexité d'un langage est la notion de "certificat". Supposons que pour un langage $L$ et un mot $x$ nous voulions montrer que $x \in L$ à l'aide d'un argument supplémentaire. A quel point est-il difficile de vérifier si cet argument est correct ? Reprenons le Problème 5 et supposons que nous recevons comme argument supplémentaire une sous-liste $L'$ telle que la somme des éléments de $L'$ est nulle. Nous pouvons alors vérifier très vite (grossièrement en $|L'|$ "étapes") que effectivement notre liste $L$ de base faisait bien partie de notre langage (c'est-à-dire l'ensemble des listes telles qu'il existe une sous-liste dont la somme des éléments est nulle). L'argument utilisé est appelé *certificat*.

Pour reprendre le problème *TETRIS*, si nous prenons un plateau $p$ et une séquence de Tétrominos $s$, et qu'additionnellement nous recevons une liste de mouvements $m$ à effectuer pour vider le plateau $p$, il est facile de vérifier si les mouvements de $m$ vident le plateau $p$ à l'aide des pièces $s$. Ici facile veut dire qu'en un temps polynomial en la taille de $m$ nous pouvons vérifier que $(p, s)$ appartient bien à *TETRIS*.

**Exemple 5.** J'affirme que



$\in$ *TETRIS*.

Certificat : rotation gauche de la pièce 1 placée aux colonnes 5,6,7,8. Placer la pièce 2 aux colonnes 3,4. Rotation gauche de la pièce 3 placée aux colonnes 5,6,7,8.

Nous dirons qu'un langage $L$ est dans la classe de complexité $\mathcal{NP}$ s'il existe pour chaque $x$ tel que $x \in L$, un certificat $C$ de longueur polynomiale en la taille de $x$, tel que la vérification que $x$ soit bien dans $L$ à l'aide de $C$ se réalise en temps polynomial.

Il est important de remarquer que tous les langages ne sont pas dans $\mathcal{NP}$. L'exemple le plus connu est celui du jeu d'échec (voir l'article de Fraenkel et Lichtenstein [9]). Nous construisons (de façon analogue au Tetris) un langage en se basant sur le jeu d'échec, c'est-à-dire avec un plateau partiellement rempli nous voulons savoir si les pièces blanches peuvent gagner la partie. Dans ce cas, la séquence de mouvements à jouer pour les blancs (qui constituerait éventuellement un certificat) peut être trop longue à vérifier en temps polynomial en la taille des données.

En résumé, les langages dans $\mathcal{P}$ peuvent être décidés rapidement alors que les langages dans $\mathcal{NP}$ peuvent être vérifiés rapidement. Au niveau de la hiérarchie avec les autres classes, nous avons le résultat suivant.

**Théorème 6.** *Nous avons les relations suivantes :*

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}\text{-}SPACE \subseteq \mathcal{EXP}\text{-}TIME,$$

*et au moins une de ces inclusions est stricte.*

Comme nous l'avons vu avec les échecs et le Tetris, beaucoup de jeux peuvent être traduit en termes de langages et problèmes de façon naturelle. Une fois ceci fait, nous pouvons les classer dans les différentes classes que nous venons de voir. C'est ce qu'illustre la Figure 0.1, les différentes preuves de cette classification peuvent être trouvées dans l'article de Demaine [7].

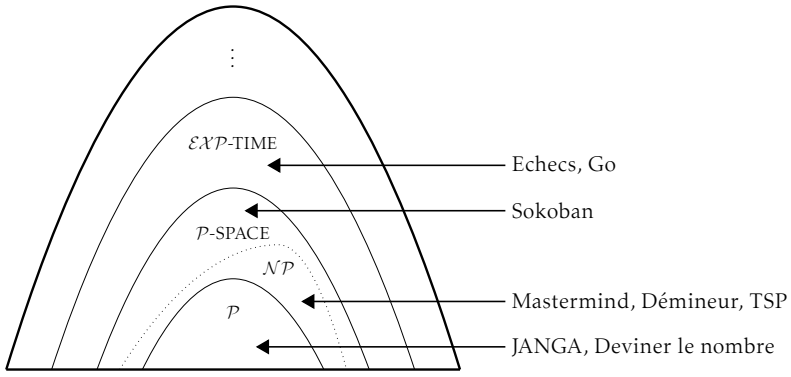



Figure 0.1 — Hiérarchie entre classes et exemples.

## $\mathcal{P}$ versus $\mathcal{NP}$

Une constatation qui découle du théorème précédent est que nous ne savons pas si $\mathcal{P} = \mathcal{NP}$. Cette égalité est l'un des plus grands problèmes irrésolus de l'informatique théorique et des mathématiques. Cette question peut être simplifié par : est-il plus facile de vérifier qu'une solution est correcte pour un problème que de résoudre le problème ? Notons que l'institut Clay [4] offre 1.000.000 $ pour une solution à ce problème qualifié comme étant un des *Millennium Problems*. La solution à ce problème aurait potentiellement de nombreuses applications en cryptographie, problèmes d'ordonnancements, confections d'horaires, bioinformatique, ...

La première avancée majeure sur la question de $\mathcal{P}$ versus $\mathcal{NP}$ est probablement le théorème de Cook–Levin [6] que nous énoncerons plus bas. Définissons d'abord (de façon peu formelle) qu'un problème de décision est $\mathcal{NP}$*-complet* si le problème est dans $\mathcal{NP}$, et qu'il est aussi difficile que tout autre problème dans $\mathcal{NP}$. Nous définirons "aussi difficile" de façon plus précise à l'aide de réductions polynomiales par la suite.

La notion de problème $\mathcal{NP}$-complet nous apporte les deux faits suivants. Premièrement, si nous pouvons décider d'un seul problème $\mathcal{NP}$-complet en temps polynomial, alors nous pouvons décider de tous les problèmes de $\mathcal{NP}$ en temps polynomial, et donc $\mathcal{P} = \mathcal{NP}$. Deuxièmement, s'il existe un problème qui est dans $\mathcal{NP}$ mais pas dans $\mathcal{P}$, alors l'ensemble des problèmes $\mathcal{NP}$-complets sont extérieurs à $\mathcal{P}$.

Pour accéder au théorème de Cook–Levin il nous faut d'abord définir le problème *SAT*. Nous rappelons qu'une expression logique est sous *forme normale conjonctive* si elles est une conjonction de disjonction de littéraux. Cela implique qu'une expression sous forme normale conjonctive ne possède comme opérateurs que le "et" logique, le "ou" logique et la négation. Nous dirons qu'une formule logique $\phi$ donnée sous forme normale conjonctive est *satisfaisable* s'il existe une assignation des variables de $\phi$ telle que $\phi$ est vrai.

$$SAT = \{\phi : \phi \text{ est satisfaisable}\}$$

**Exemple 7.** La formule

$$\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$$

appartient à *SAT*, car si $x_1$ est vrai et $x_2$ est faux alors $\phi_1$ sera satisfaite. Par contre la formule

$$\phi_2 = x_1 \wedge x_2 \wedge \neg x_1$$

n'appartient pas *SAT*, car elle est fausse pour toute assignation des variables.

Nous pouvons maintenant énoncer le théorème de Cook–Levin qui, pour la première fois, a exhibé un problème $\mathcal{NP}$-complet.

**Théorème 8.** *SAT est $\mathcal{NP}$-complet.*

La démonstration, bien que compréhensible, requiert tout de même un formalisme sur les machines de Turing que nous ne développons pas ici.

Une fois ce théorème établi, il existe une technique appelée "réduction polynomiale" permettant de montrer que certains problèmes sont également $\mathcal{NP}$-complets. Une réduction peut être vue comme une transformation $f$ d'un problème (ou langage) $A$ à un autre problème $B$ comme le montre la Figure 0.2 où

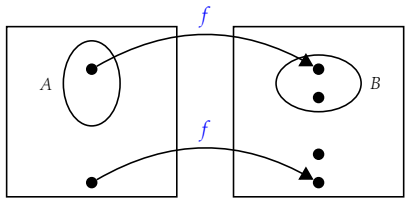



FIGURE 0.2 — Illustration de la notion de réduction

nous avons que $x \in A$ si et seulement si $f(x) \in B$. Autrement dit, le problème $A$ peut être réduit à $B$.

Ici, nous allons utiliser des réductions dites efficaces c'est-à-dire que la fonction $f$ de transformation peut être calculée en un temps polynomial.

Ceci nous amène aux remarques importantes qui suivent. S'il existe un algorithme rapide pour $B$, il en existe un également pour $A$. Par contre, s'il n'existe pas d'algorithme rapide pour $A$, il n'en existe pas pour $B$. Donc nous pouvons affirmer que le problème $B$ est au moins aussi difficile que $A$. Ces remarques nous amènent vers le théorème de Karp.

**Théorème 9.** *Soient un problème $\mathcal{NP}$-complet $A$ et un autre problème $B$ qui se trouve dans $\mathcal{NP}$. S'il existe une réduction efficace de $A$ vers $B$, alors $B$ est $\mathcal{NP}$-complet.*

Bien évidemment, pour appliquer le théorème de Karp en pratique, il faut disposer d'un problème que nous savons $\mathcal{NP}$-complet. Heureusement, nous pouvons utiliser le théorème de Cook–Levin.

## 4 La complexité du problème *TETRIS*

Le lecteur peut ici légitimement se demander si le jeu Tetris refera une apparition dans ce texte.

### Revenons au Tetris

Un des objectifs de ce travail est d'esquisser une idée de preuve pour le théorème suivant dû à Demaine, Hohenberger et Liben-Nowell [8].

**Théorème 10.** *TETRIS est $\mathcal{NP}$-complet.*

Pour nous convaincre de ce résultat, utilisons le théorème Karp. Il nous faut donc réaliser les étapes suivantes :

1. définir une correspondance $f$ entre les instances d'un problème $\mathcal{NP}$-complet connu $L$ et *TETRIS* ;
2. montrer que si $x \in L$ alors $f(x) \in$ *TETRIS* ;
3. montrer que si $x \notin L$ alors $f(x) \notin$ *TETRIS*.

Ces étapes seront réalisées dans les trois sous-sections suivantes.

## Correspondance entre un problème $\mathcal{NP}$-complet et *TETRIS*

Le problème $\mathcal{NP}$-complet en question que nous allons utiliser pour montrer le Théorème 10 est appelé 3-*PARTITION*. Avant de le définir nous avons besoin de la notion suivante. Un multiensemble $S$ de nombres est dit *3-partitionable* si $S$ peut être partitionné en sous-multiensembles de taille 3, tels que la somme des éléments dans chaque sous-multiensemble soit la même. Nous avons maintenant la définition du langage suivant :

$$3\text{-}PARTITION = \{S : S \text{ est 3-partitionable}\}$$

En 1975, Garey et Johnson [10] ont montré le théorème suivant.

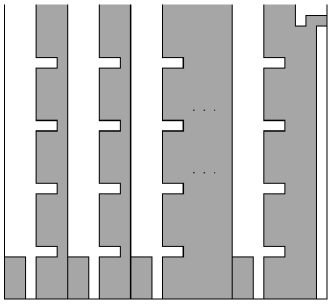
**Théorème 11.** *3-PARTITION est $\mathcal{NP}$-complet.*

Notons que le problème ci-dessus reste $\mathcal{NP}$-complet même si nous connaissons la somme des éléments commune aux sous-multiensembles que nous tentons d'atteindre. Avant de continuer, observons un exemple d'instance pour le problème de 3-*PARTITION*.

**Exemple 12.** Le multiensemble $S = \{26, 29, 33, 33, 33, 34, 35, 36, 41\}$ appartient à 3-*PARTITION*, car la partition suivante existe :

$$\{26, 33, 41\}, \{29, 35, 36\} \text{ et } \{33, 33, 34\}.$$

Nous avons maintenant les outils nécessaires pour esquisser l'idée d'une réduction efficace de 3-*PARTITION* à *TETRIS*. Supposons que nous disposons d'une instance de 3-*PARTITION* constituée du multiensemble $S = \{a_1, a_2, \ldots, a_{3s}\}$ et du réel $T$ qui est la somme commune aux $s$ multiensembles que nous tentons d'atteindre. Nous construisons une instance de *TETRIS* à l'aide du plateau décrit ci-dessous.

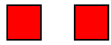


Ce plateau possède les dimensions suivantes : $6s + 3$ cases de largeur et $6T + 22$ cases de hauteur.

Il nous faut encore associer une séquence de Tétrominos pour que notre instance du problème soit complète. Celle-ci sera définie par les valeurs $a_1, a_2, \ldots a_{3s}$. Nous procédons de la manière suivante : pour $i$ compris entre 1 et $3s$, nous assignons à $a_i$ la sous-séquence composée de :

— initiation :  ;

— remplissage :  ;

— clôture :  .

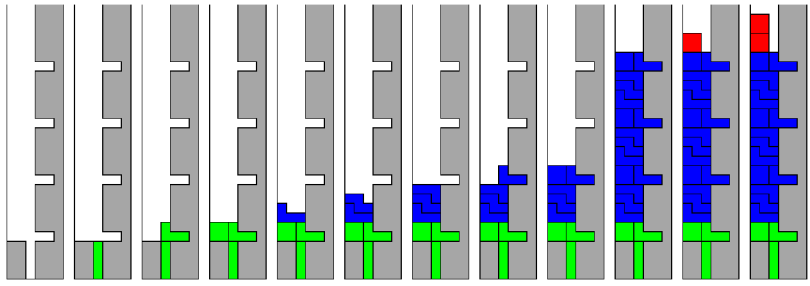
Après avoir créé ces $3s$ parties de séquence, nous terminons avec :

— fin de la séquence :  $s$ fois  1 fois  $(\frac{3T}{2} + 5)$ fois.

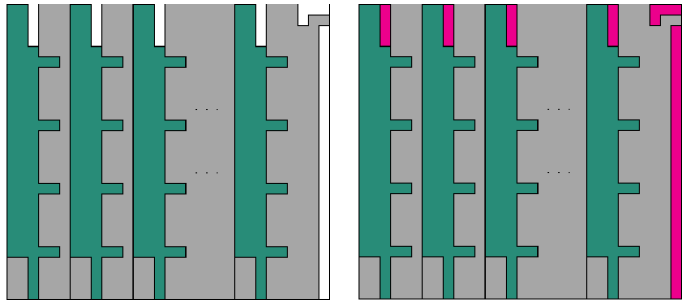
Et donc nous obtenons bien une séquence de pièces définie par $a_1, a_2, \ldots a_{3s}$. Avec ceci, nous voyons comment le problème *TETRIS* va être perçu comme un cas particulier de 3-*PARTITION* dans le sens où, pour n'importe quelle instance de 3-*PARTITION*, nous aurons une instance particulière de *TETRIS*. Autrement dit nous avons obtenu une réduction du problème. Bien entendu, il faudrait encore vérifier que cette réduction est polynomiale.

### Réduction : oui veut dire oui

Tentons de nous convaincre à présent que si une instance de 3-*PARTITION* est positive alors l'instance de *TETRIS* correspondante est également positive. Grâce à notre réduction ci-dessus, nous empilons les Tétrominos en fonction de la partition comme indiqué ci-dessous. Nous commençons par remplir presque entièrement chaque colonne vide du tableau :

 .

Une fois que toutes les colonnes sont presque remplies, nous utilisons la fin de la séquence pour vider le tableau :

 .

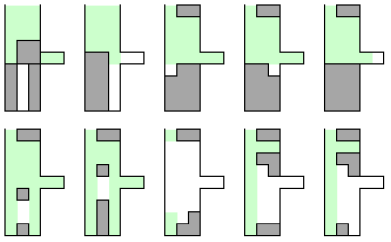
Au final, nous avons que tous les sous-multiensembles ont la même somme, donc les "piles" dans le tableau ont la même hauteur. C'est cela qui va permettre que

chaque "trou" dans le plateau soit rempli jusqu'en haut et que les pièces restantes vident le tableau. Et donc les instances positives donnent bien des instances positives.

### Réduction : non veut dire non

Tentons de nous convaincre à présent que si une instance de 3-*PARTITION* est négative alors l'instance de *TETRIS* correspondante est également négative. La construction particulière de notre tableau nous permet de tirer les remarques qui suivent. Premièrement, aucune ligne ne peut être effacée avant l'arrivée de la pièce en "L" en fin de séquence. Deuxièmement, les trous doivent complètement être remplis sinon le tableau ne pourra jamais être vidé. Et troisièmement, les alcôves peuvent très vite faire perdre le jeu.

Grâce à cela, si les $a_i$ ne forment pas un élément de 3-*PARTITION*, alors le tableau ne sera jamais vidé, et ce pour la raison suivante. Tout mouvement différent du remplissage précédent conduit à une des configurations suivantes :



Et celles-ci conduisent inévitablement à l'échec. Et donc les instances négatives donnent bien des instances négatives.

Nous avons à présent une idée plus précise de pourquoi *TETRIS* est $\mathcal{NP}$-complet.

## 5 Résultats annexes

Dans cette section, nous observerons des résultats liés plus ou moins aux concepts abordés ci-dessus.

### Résultats sur le vrai jeu Tetris

En ce qui concerne le jeux Tetris de base (et non le problème *TETRIS*), le résultat suivant a été établi par Brzustowski [3] en 1992.

**Théorème 13.** *Une série suffisamment longue de pièces "S" et "Z" cause une défaite.*

Si on part du principe que les pièces données au joueur sont "bien" aléatoires, le résultat ci-dessus implique qu'un joueur de Tetris (même très doué) finira par perdre avec probabilité 1.

### Résultats de complexité sur d'autres jeux vidéos

Pour terminer cette section, notons que le principe de traduire certains jeux informatiques en problèmes et d'en analyser la complexité ne se limite pas à Tétris.

Voici à titre d'exemple quelques résultats dont les preuves et réductions peuvent être trouvées dans les travaux de Aloupis, Demaine, Guo et Viglietta [2] et [13]. Dans la liste ci-dessous, un problème est dit $\mathcal{NP}$-difficile (respectivement $\mathcal{P}$-SPACE-difficile) si il est au moins aussi difficile que n'importe quel problème dans $\mathcal{NP}$ (respectivement dans $\mathcal{P}$-SPACE).

— *Super Mario Bros* est $\mathcal{NP}$-difficile ;

— *Donkey Kong Country* est $\mathcal{NP}$-difficile ;

— *Super Metroid* est $\mathcal{NP}$-difficile ;

— *Pokemon (Red/Blue)* est $\mathcal{NP}$-difficile ;

— *Lemmings* est $\mathcal{NP}$-difficile ;

— *Pac-man* est $\mathcal{NP}$-difficile ;

— *Starcraft* est $\mathcal{NP}$-difficile ;

— *Legend of Zelda : Ocarina of Time* est $\mathcal{P}$-SPACE-difficile ;

— *Doom* est $\mathcal{P}$-SPACE-difficile.

## 6 Conclusion

Nous avons ici, au travers du jeu Tetris, touché à certains concepts fondamentaux en théorie de la complexité. Sans rentrer dans un formalisme extrême (mais nécessaire dans un cadre plus rigoureux), nous avons tenté de transmettre un bon aperçu de la discipline. Nous avons vu quelques questions et théorèmes fondamentaux de l'informatique théorique.

Le lecteur intéressé peut approfondir les notions vues dans le texte avec les références données à la fin de ce travail. Nous encourageons l'approfondissement des concepts vus à l'aide des "machines de Turing" qui apporte la rigueur supplémentaire nécessaire. Nous nous permettons, pour terminer, de laisser un moyen facile de gagner un million de dollars cette semaine : montrer que $TETRIS \in \mathcal{P}$ et communiquer votre résultat à l'institut Clay.

## 7 Bibliographie

[1] S. Aaronson. Why philosophers should care about computational complexity. *Computability : Gödel, Turing, Church, and beyond*, MIT Press, 261–329 2013.

[2] G. Aloupis, E. D. Demaine, A. Guo, et G. Viglietta. Classic Nintendo games are (computationally) hard. *Fun with algorithms : 7th international conference*, Springer International Publishing, 40–51, 2014.

[3] J. Brzustowski. Can you win at TETRIS ? *PhD thesis*, The University of British Colomblia, 1992.

[4] Clay Mathematics Institute. `http://www.claymath.org`

[5] Complexity Zoo. `https://complexityzoo.uwaterloo.ca`

[6] S. A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM Symposium on theory of computing*, STOC '71, 151–158, 1971.

[7] E. D. Demaine. Playing games with algorithms : algorithmic combinatorial game theory. *Mathematical foundations of computer science*, Lecture Notes in Comput. Sci., **2136**, Springer, 18 – 32, 2001.

[8] E. D. Demaine, S. Hohenberger, et D. Liben-Nowell. Tetris is hard, even to approximate. *Computing and combinatorics*, Lecture Notes in Comput. Sci., **2697**, Springer, 351 – 363, 2003.

[9] A. S. Fraenkel et D. Lichtenstein Computing a perfect strategy for $n \times n$ chess requires time exponential in $n$. *Journal of Combinatorial Theory (Series A)*, 31, **2**, 199 – 214, 1981.

[10] M. R. Garey et D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM journal on computing*, 4, **4**, 397–411, 1975.

[11] M. Sipser. *Introduction to the theory of computation*, International Thomson Publishing, 1996.

[12] Tetris Inc. `http://www.tetris.com`

[13] G. Viglietta. Gaming is a hard job, but someone has to do it ! *Theory of computing systems*, 54, **4**, 595–621, 2013.