# Complexité TP2

Jeff ALLARD, Elsbeth MONRROY, Étienne MULLER, Axel POULAIN

Novembre 2022

## 1 Vérificateur déterministe pour SAT

L'objectif de cette partie était d'implémenter un vérificateur qui, étant donné une représentation d'une formule logique en CNF et d'une affectation des variables, vérifie la satisfaisabilité de cette formule vis-à-vis de l'affectation donnée.

### 1.1 Implémentation du vérificateur

Notre algorithme de vérification va d'abord itérer sur toutes les clauses. Pour chaque clause, on va itérer sur l'ensemble des littéraux composants celle-ci, et dès que l'on trouve un littéral affecté à vrai, la clause en question est satisfaite avec notre affectation et on continue d'itérer sur les clauses restantes. En revanche, si l'on atteint la fin de la clause sans avoir rencontré un littéral affecté à vrai, la clause n'est alors pas satisfaite avec notre affectation, et la formule ne l'est également pas — le vérificateur retourne donc faux. Dans le cas où toutes les clauses ont été visitées et que chacune est satisfaite par l'affectation donnée, alors le vérificateur retourne vrai.

### 1.2 Complexité du vérificateur

Comme expliqué dans la section précédente, le cas le plus long à vérifier est celui où la formule n'est pas satisfaisable ; on va alors vouloir vérifier toutes les clauses afin de s'assurer qu'aucune ne soit insatisfaisable — nous allons donc nous intéresser à ce cas-là.

Les affectations sont stockées dans un tableau — en l'occurrence, un Bitset stockant la valeur des différentes variables. Le nombre d'étapes pour accéder à une affectation de littéral est donc une valeur constante, que nous pouvons nommer $k$.
Lors de la vérification d'une clause, dans le pire des cas, le vérificateur doit itérer sur tous les littéraux présents afin de vérifier leur affectation ; lorsqu'une clause contient $n$ littéraux, on a alors $n \times k$ étapes de vérifications d'affectations. Enfin, le vérificateur parcourt toutes les clauses, donc avec $m$ clauses, nous avons $m \times n \times k$ étapes.

Étant donné que l'entrée de notre algorithme est la formule et que celle-ci est encodée en détaillant la composition en littéraux de chacune de ses clauses, celle-ci prend une taille de $m \times n$. Notre algorithme est donc linéaire par rapport à la taille de l'entrée.

### 1.3 Performance du vérificateur

Afin de vérifier les performances de notre vérificateur, nous avons implémenté deux générateurs de formules sous forme normale conjonctive. L'un génère une formule tautologique, et l'autre une formule contradictoire/insatisfaisable.
Dans les paragraphes suivants, $v_1, v_2, \ldots, v_n$ sont des variables booléennes.

Le générateur de formules tautologiques génère des formules de la forme suivante, avec $n$ la taille demandée pour la

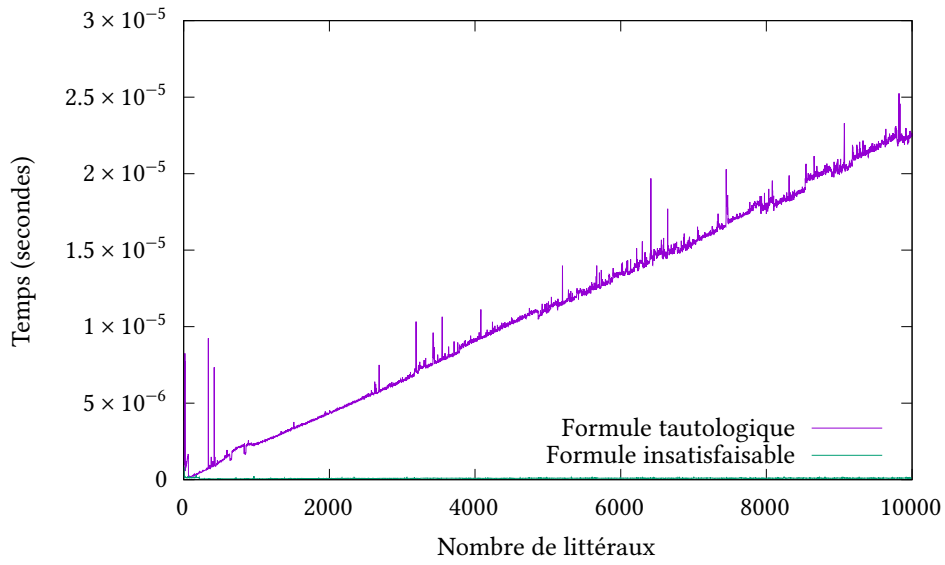


FIGURE 1 – Performances de la réduction et du solveur

formule correspondant au nombre de variables présentes dans la formule :

$$(v_1 \vee \neg v_1) \wedge (v_2 \vee \neg v_2) \wedge \cdots \wedge (v_n \vee \neg v_n)$$

De même, le générateur de formules insatisfaisables génère des formules de la forme suivante :

$$v_1 \wedge \neg v_1 \wedge v_2 \wedge \neg v_2 \wedge \cdots \wedge v_n \wedge \neg v_n$$

Étant donné que chaque littéral n'apparaît qu'une seule fois dans ces formules, la taille de celles-ci est équivalente au nombre de littéraux différents y apparaissant, et donc au nombre de variables multiplié par 2.

Enfin, deux générateurs d'affectations ont été implémentés, permettant d'obtenir une assignation de variables pour chaque formule générée ; l'un assigne une valeur vraie à toutes les variables, et l'autre assigne une valeur fausse à toutes les variables. Seul le premier sera utilisé dans nos tests, mais cela revient au même.

Afin de mesurer efficacement les performances de notre vérificateur, nous avons écrit une fonction par générateur de formule, permettant de tester celui-ci sur une taille spécifique de formule.
Au moment du test de performance, on sélectionne un intervalle de taille à tester, et ces fonctions de test sont appelées pour chaque taille de l'intervalle. Notons de plus que, pour la même taille de formule, on exécute plusieurs fois ces fonctions de test et l'on prend ensuite la moyenne de toutes ces valeurs afin d'éviter d'obtenir des valeurs trop bruitées.

Les résultats présentés sur la figure 1 ont été obtenus en effectuant des tests sur 5000 variables (et donc sur des formules possédant 10000 littéraux différents), et 200 mesures ont été prises pour chaque taille.
Les mesures concernant les formules tautologiques sont assez bruitées, particulièrement à partir des formules de taille 5000 ainsi qu'au tout début des mesures, mais malgré cela il apparaît clairement que le temps mis pour vérifier une formule semble évoluer linéairement avec la longueur de celle-ci, ce qui est le comportement attendu. Quant aux formules insatisfaisables, on ne voit presque pas leurs mesures sur le graphique tant elles sont proches de 0 ; il s'agit également du résultat attendu, étant donné que pour les formules générées la vérification des toutes premières clauses permet d'en déduire leur insatisfaisabilité, cela étant alors fait en temps constant.

# 2 Réduction de zone vide à SAT

L'objectif est ici de se servir d'une réduction vers le problème SAT afin de résoudre le problème de zone vide.

## 2.1 Passage d'un graphe à une formule

Afin de pouvoir exprimer la notion de graphe et de zone vide d'un graphe dans une formule, il va nous falloir tout d'abord définir ce que représentent les variables que nous prenons. Supposons que nous avons un graphe constitué de $n$ sommets. Pour le moment, nous allons créer $n$ variables $v_1, v_2, \ldots, v_n$, dont la valeur de vérité représentera le fait de prendre ou pas respectivement les sommets 1, 2, …, $n$ dans la zone vide.

Une zone vide impose à tous ses sommets de n'avoir aucun arc menant vers un autre sommet de la zone vide. Nous pouvons alors facilement traduire cette contrainte en clause.
En effet, soit un sommet $i$. S'il est contenu dans la zone vide du graphe, il est alors nécessaire que pour tout autre sommet $j$ voisin de $i$, $j$ ne soit pas contenu dans la zone vide. Avec nos variables, on a alors pour tout sommet $j$ voisin de $i$ :

$$v_i \implies \neg v_j \equiv \boxed{\neg v_i \vee \neg v_j}$$

On itère alors sur tous les sommets $i$, et on génère une clause pour chaque sommet $j$ voisin de $i$. Cela nous assure alors que toute affectation satisfaisant la formule sera une zone vide valide.

Seulement, dans l'état actuel, rien n'oblige le solveur à chercher une solution intéressante : en effet, il est tout à fait possible de satisfaire la formule en ne prenant aucun sommet dans la zone vide. Il faut alors trouver un moyen de faire correspondre l'affectation de nos variables à $k$, le nombre de sommets que l'on désire retrouver dans la zone vide.

Nous allons donc introduire de nouvelles variables. Pour un problème avec $n$ sommets et une zone vide à chercher de taille $k$, on va introduire $n \times k$ variables $v_{i,j}$, avec $i$ variant de 1 à $n$ et $j$ de 1 à $k$. On va considérer que les sommets contenus dans la zone vide sont ordonnés, c'est ce qui va nous permettre de réaliser notre réduction, bien que ce ne soit pas le cas en réalité.
Assigner à la variable $v_{i,j}$ une valeur vraie représente le fait de prendre le sommet $i$ à la $j$-ème place de la zone vide.

On peut visualiser toutes ces variables via une matrice :

$$\begin{pmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,k} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \cdots & v_{n,k} \end{pmatrix}$$

À partir de là, on peut en déduire de nouvelles clauses adaptées à ces nouvelles variables :

1. Il y a au moins une variable vraie par colonne, en effet, cela signifie que le $j$-ème sommet de la zone vide existe :
$$\bigwedge_{j=1}^{k} \bigvee_{i=1}^{n} v_{i,j}$$
2. Il y a au plus une variable vraie par colonne, cela signifie que le $j$-ème sommet de la zone vide est unique :
$$\bigwedge_{j=1}^{k} \bigwedge_{i_1=1}^{n} \bigwedge_{i_2=i_1+1}^{n} \neg v_{i_1,j} \vee \neg v_{i_2,j}$$
3. Au plus une variable vraie par ligne : cela signifie que le sommet $i$ peut ne pas être dans la zone vide (auquel cas aucune variable de la ligne $i$ n'est assignée à vrai), mais aussi qu'il ne peut pas prendre la place de plusieurs sommets dans le graphe vide :
$$\bigwedge_{i=1}^{n} \bigwedge_{j_1=1}^{k} \bigwedge_{j_2=j_1+1}^{k} \neg v_{i,j_1} \vee \neg v_{i,j_2}$$

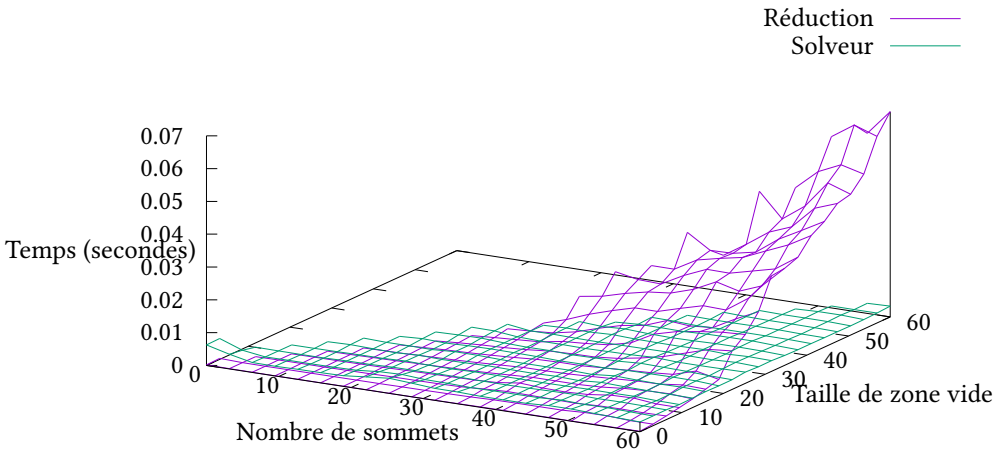


FIGURE 2 – Performances du vérificateur et solveur du problème Zone Vide

4. On adapte notre contrainte de voisinage à ces nouvelles variables : si je prends le sommet $i_1$ dans la zone vide, quelque soit son emplacement dans celle-ci, alors pour tout sommet $i_2$ voisin de $i_1$, je ne peux pas prendre ce premier dans la zone vide, encore une fois quelque soit son emplacement dans celle-ci. On peut décrire cela plus précisément, en partant d'un graphe $G = (S, A)$ :

$$\bigwedge_{i_1 \in S} \bigwedge_{j_1=1}^{k} \bigwedge_{i_2 \text{ voisin de } i_1} \bigwedge_{j_2=1}^{k} \neg v_{i_1,j_1} \vee \neg v_{i_2,j_2}$$

## 2.2 Performances

On va tester notre algorithme pour chaque paire {taille de graphe, taille de zone vide} possible. On se servira pour cela de graphes générés aléatoirement avec une densité approximative de $0,7$, en allant jusqu'à des graphes de taille 60. Pour chaque paire, on effectue 5 mesures différentes afin de réduire la quantité de bruit présente dans les résultats. On obtient alors un nuage de points en 3 dimensions. Afin de mieux le visualiser, on le simplifie en une grille 3D qui approxime le comportement de tous les points (obtenue avec la commande `dgrid3d` de gnuplot), visible sur la figure 2.

Remarque importante, le temps mis par le solveur est calculé en mesurant le temps d'exécution de celui-ci sur le problème donné… Mais malheureusement, ce temps comporte également le temps que le solveur met à analyser, ou "parser", les formules, ce qui risque d'avoir un effet sur les résultats.

On observe qu'à partir d'un certain nombre de sommets et une certaine taille de zone vide, la croissance du temps d'exécution de la réduction croit de manière importante. En revanche, le temps d'exécution du solveur semble quasi-constante ; cela peut-être dû aux formules qui seraient faciles à manipuler pour le solveur, ou potentiellement le temps de recherche de solution, très faible, qui se fait noyer par le temps d'analyse du fichier — il est possible que ce soit ce dernier cas, car selon les rapports du solveur sur certains exemples, le temps mis à analyser le fichier n'est pas négligeable.

### 2.3 Analyse de complexité

Reprenons les quatre types de contraintes. Considérons que rajouter une variable à une clause se fait en temps linéaire.
Soit une instance du problème sur un graphe de $n$ sommets, et avec une zone vide de taille $k$.

1. On rajoute $n$ variables $k$ fois, on a donc $n \times k$ étapes.
2. On rajoute $k$ fois deux variables sur toutes les paires possibles des sommets. Étant donné que le nombre de paires possibles sur $n$ éléments est du même ordre de grandeur que $n \times n$, alors le nombre d'étapes est du même ordre de grandeur que $2k \times n^2$.
3. Avec exactement le même procédé que précédemment, on montre que le nombre d'étapes est du même ordre de grandeur que $2n \times k^2$.
4. On rajoute encore une fois deux variables, mais cette fois sous certaines conditions particulières. En l'occurrence, on veut empêcher deux sommets voisins d'être dans la même zone vide. Dans le pire cas, nous sommes dans un graphe complet, et tous les sommets sont donc voisins entre eux. On itère tout d'abord sur toutes les variables (donc sur tout les sommets à chaque place de zone vide possible) ; cela nous fait donc $n \times k$ étapes. On itère ensuite sur chaque variable dont le sommet est voisin avec le sommet de la première variable, quelque soit son emplacement dans la zone vide. Dans le cas d'un graphe complet, cela nous fait encore $n \times k$ étapes. Au final, nous avons donc $2 \times (n \times k)^2$ dans le pire des cas.

Dans le pire des cas, le nombre d'étapes est donc du même ordre de grandeur que $n^2$. Le graphe étant donné à l'algorithme avec un codage de taille $n^2$ (plus le nombre $k$, de taille négligeable), cet algorithme s'exécute donc linéairement selon la taille de l'entrée.

## 3 Réduction de Sudoku à SAT

Pour réduire le problème du Sudoku à SAT il faut d'abord comprendre qu'est ce que le Sudoku.

Le sudoku est donc une grille avec $n^2$ lignes, $n^2$ colonnes, $n^2$ régions, $n^4$ cases et $n^2$ valeurs. Il n'y a que quelques règles à respecter lorsque l'on complète un Sudoku : les valeurs ne peuvent apparaître qu'une seul fois par ligne, par colonne et par région.(On notera qu'une case ne peut avoir qu'une seule et unique valeur). Donc pour réduire ce problème a SAT il faut déjà généraliser le problème et de bien choisir les valeurs pour SAT.

### 3.1 Généralisation du problème du Sudoku à SAT

Avant de commencer quoi que ce soit il faut choisir les valeurs pour SAT. Ici nous allons utiliser $V_{i,j,k}$ avec $i,j$ et $k$ allant de 1 à $n^2$. $i$ et $j$ représentent ses coordonnées et $k$ sa valeur dans le Sudoku. exemple : $V_{1,1,1}$ = Vrai veut dire qu'à la position (1,1) du Sudoku on a la valeur 1.

On peut donc maintenant commencer à créer les clauses respectant les règles du Sudoku.

1. Chaque case a au moins une valeur :
$$\bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigvee_{k=1}^{n^2} V_{i,j,k}$$
2. Chaque case ne peut pas avoir plus d'une valeur :
$$\bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigwedge_{k_1=1}^{n^2} \bigwedge_{k_2=k_1+1}^{n^2} \neg V_{i,j,k1} \vee \neg V_{i,j,k_2}$$
3. On ne peut pas avoir plusieurs fois la même valeur dans chaque colonne :
$$\bigwedge_{i_1=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigwedge_{k=0}^{n^2} \bigwedge_{i_2=i_1+1}^{n^2} \neg V_{i_1,j,k} \vee \neg V_{i_2,j,k}$$

4. On ne peut pas avoir plusieurs fois la même valeur dans chaque ligne :

$$\bigwedge_{i=1}^{n^2}\bigwedge_{j_1=1}^{n^2}\bigwedge_{k=0}^{n^2}\bigwedge_{j_2=j_1+1}^{n^2}\neg V_{i,j_1,k}\vee\neg V_{i,j_2,k}$$

5. On ne peut pas avoir plusieurs fois la même valeur dans chaque zone :

$$\bigwedge_{z=1}^{n}\bigwedge_{i_1=z*n}^{z*n+n}\bigwedge_{i_2=z*n,i_1\neq i_2}^{z*n+n}\bigwedge_{w=1}^{n}\bigwedge_{j_1=w*n}^{w*n+n}\bigwedge_{j_2=w*n,j1\neq j_2}^{w*n+n}\bigwedge_{k=1}^{n^2}\neg V_{i_1,j_1,k}\vee\neg V_{i_2,j_2,k}$$

Maintenant que le problème est généralisé il suffit de gérer le cas particulier.

## 3.2 Cas particulier du problème du Sudoku à SAT

Le cas particulier du problème du Sudoku est très simple. Il suffit simplement d'affecter les valeurs que l'on sait positif du Sudoku initial (SI). exemple : si dans le Sudoku initial il a la valeur 1 à la position (1,1) c'est à dire si SI(1,1) = 1, il suffit de créer la clause $V_{1,1,1}$. On applique donc cela pour toutes les valeurs initialement placées :

$$(6)\ :\ \bigwedge_{i=1}^{n^2}\bigwedge_{j=1}^{n^2}(SI(i,j)\neq\emptyset)\Rightarrow V_{i,j,k}$$

## 3.3 Méthode et complexité

On rappel que pour un Sudoku de taille $n$, on a $n^2$ ligne, $n^2$ colonnes, $n^2$ zones, $n^2$ valeurs et $n^4$ cases.

La méthode *toSAT()* permet de créer la CNF d'un Sudoku. On a donc 6 parties dans cette méthode qui représentent les 6 formules des 2 sous sections précédentes :

- (1) On doit parcourir toutes les cases et créer une clause obligeant la case à avoir au moins une valeur. La complexité est donc $\Theta(n^6)$.

-(2) On doit parcourir toutes les cases et créer une suite de clause interdisant d'avoir 2 valeurs sur cette même case. La complexité est donc $\Theta(n^8)$.

-(3) On doit parcourir chaque case de chaque colonne et créer une suite de clause interdisant la même valeur sur deux cases ou plus de cette même colonne. La complexité est donc $\Theta(n^8)$.

-(4) On doit parcourir chaque case de chaque ligne et créer une suite de clause interdisant la même valeur sur deux cases ou plus de cette même ligne. La complexité est donc $\Theta(n^8)$.

-(5) On doit parcourir chaque case de chaque zone et créer une suite de clause interdisant la même valeur sur deux cases ou plus de cette même zone. La complexité est donc $\Theta(n^8)$.

-(6) On doit parcourir chaque case du Sudoku initial et affecter les valeurs présentes. La complexité est donc $\Theta(n^4)$.

La complexité de toSAT est finalement $\Theta(n^8)$.

## 3.4 Analyse des données et performance

La figure 3 nous montre le temps de calul de la réduction et du SAT-Solver en fonction de la taille du Sudoku. On remarque que le temps de calcul croit exponentiellement par rapport a la taille de l'entrée alors que le Solveur n'évolue que très peu. La complexité pour résoudre ce problème est donc principalement du à la réduction dans notre cas.

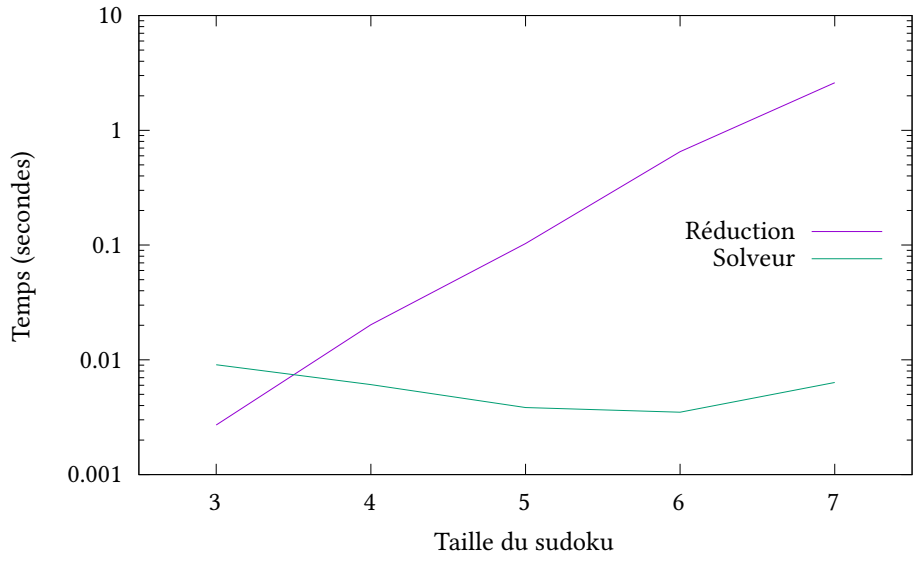


FIGURE 3 – Performances de la réduction et du solveur de Sudoku