Université d'Aix-Marseille

UFR des Sciences
Département d'Informatique

Rapport de TP

Spécialité :

Master 1 Informatique

Thème :

Réductions.

Réalisé par :

M[lle] ZEMMOURI Yasmine.

Mr. YAHI Lotfi.

Mr. BELOUAHCHI Nomane.

M[lle] BEKHADDA Hadjira.

Année Universitaire : 2023-2024.

Mini-projet 1 : Vérificateur déterministe pour SAT.

Le problème SAT, ou "Boolean Satisfiability," constitue le cœur de la classe des problèmes NP-complets, qui regroupe des défis considérés parmi les plus difficiles à résoudre de manière générale.

Le problème se pose ainsi : étant donnée une expression booléenne, existe-t-il une assignation de valeurs aux variables qui rend l'expression vraie ? Cette question apparemment simple cache une complexité redoutable. Ce problème est NP-complet, ce qui signifie que s'il peut être résolu efficacement, alors tous les problèmes NP-complets peuvent également l'être.

L'importance de SAT réside dans sa capacité à être utilisé comme point de départ pour démontrer la NP-complétude d'autres problèmes. En effet, s'il est possible de transformer efficacement toute instance d'un problème NP-complet en une instance du problème SAT (via une réduction polynomiale), cela confirme que le problème initial est également NP-complet.

Son appartenance à la classe NP signifie qu'il existe un algorithme polynomial non déterministe pour déterminer si une formule en forme normale conjonctive (FNC) possède une affectation qui la satisfait. En d'autres termes, bien que la recherche de cette affectation puisse être difficile, si elle est fournie, elle peut être vérifiée en temps polynomial.

Plus précisément, cet algorithme non déterministe peut deviner l'affectation en temps polynomial et vérifier ensuite en temps polynomial si cette affectation satisfait la formule SAT. Cette caractéristique définit la classe NP : la non-déterminisme intervient dans le processus de recherche, tandis que la vérification peut être effectuée rapidement.

D'autre part, il existe également un vérificateur déterministe polynomial pour SAT. Ce vérificateur prend en entrée une formule en FNC et une affectation pour ses variables. Il peut rapidement (en temps polynomial) vérifier si l'affectation satisfaite la formule, offrant ainsi une méthode efficace pour confirmer la validité d'une solution proposée.

L'objectif de cette phase initiale du projet est ainsi de mettre en œuvre **un vérificateur déterministe** pour le problème SAT.

Définition formelle du problème SAT

Le problème SAT (Boolean Satisfiability Problem) peut être formellement défini comme suit : étant donnée une expression booléenne ϕ sur un ensemble de variables booléennes $X = \{x_1, x_2, \ldots, x_n\}$, représentée en forme normale conjonctive (FNC), c'est-à-dire comme une conjonction de clauses, l'objectif est de déterminer s'il existe une assignation de valeurs booléennes à chaque variable de X telle que ϕ soit évaluée à vrai. Formellement, cela revient à déterminer s'il existe une fonction d'assignation $f : X \to \{0, 1\}$ telle que l'évaluation de ϕ sous cette assignation donne le résultat $\phi(f(x_1), f(x_2), \ldots, f(x_n)) =$ vrai. Si une telle assignation existe, le problème SAT a une réponse positive ; sinon, la réponse est négative.

Considérons l'expression booléenne suivante en FNC :

$$\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_4)$$

Le problème SAT consiste à déterminer s'il existe une assignation de valeurs booléennes à x_1, x_2, x_3, et x_4 qui rend ϕ évaluée à vrai.

Par exemple, l'assignation $x_1 =$ Vrai, $x_2 =$ Faux, $x_3 =$ Faux, $x_4 =$ Vrai satisfait cette expression.

Format DIMACS CNF

Le format DIMACS CNF (Conjunctive Normal Form) est un standard de représentation de formules logiques en logique propositionnelle, souvent utilisé dans le contexte du problème SAT.

La première ligne d'un fichier DIMACS CNF commence par "p cnf" suivi du nombre de variables (`nbvar`) et du nombre de clauses (`nbclauses`) dans la formule logique. Les lignes suivantes représentent les clauses, chaque clause étant une disjonction de littéraux (variables ou leur négation). Chaque ligne se termine par le chiffre 0 pour indiquer la fin de la clause.

Par exemple, la ligne `p cnf 3 2` est suivie des clauses `1 -3 0` et `2 3 -1 0`, représentant la formule $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$ avec trois variables (x_1, x_2, x_3) et deux clauses.

Affectation aux Variables

L'affectation des variables dans le contexte du problème SAT se réfère à l'assignation de valeurs booléennes spécifiques (Vrai ou Faux) aux différentes variables de la formule logique. Cette affectation est représentée dans un fichier distinct et suit également un format standard, souvent utilisé en conjonction avec le format DIMACS CNF. Chaque ligne de ce fichier représente une affectation complète des variables, où chaque variable peut apparaître en forme positive ou négative, indiquant son état de Vrai ou Faux.

Par exemple, une ligne telle que "1 -2 -3" signifie que x_1 est assigné à Vrai, x_2 à Faux, et x_3 à Faux. Chaque variable apparaît une fois dans cette ligne, soit en forme positive, soit en

forme négative, et cette ligne représente donc une affectation complète et cohérente.

Ce fichier d'affectation est utilisé par des programmes de vérification pour déterminer si une assignation donnée satisfait la formule logique correspondante, en format DIMACS CNF, apportant ainsi une solution à la question centrale du problème SAT.

Solution proposée

Avant d'utiliser un vérificateur SAT, il est impératif de s'assurer que les données d'entrée sont correctement formatées. Il convient de suivre les étapes suivantes :

1. Vérifier la validité du format des fichiers en s'assurant que les informations nécessaires, telles que le nombre de variables et de clauses, sont correctement spécifiées.
2. Si les fichiers sont valides, extraire les clauses de la formule logique et les variables d'affectation du format DIMACS CNF.

Le pseudo-algorithme suivant décrit la vérification de la validité du fichier CNF ainsi que l'extraction de la formule :

Algorithm 1 Extraction de formule DIMACS CNF

```
1 : function EXTRAIRE_FORMULE(path)
2 :     nb_variables ← 0
3 :     nb_clauses ← 0
4 :     formule ← []
5 :     conforme ← True
6 :     with  open(path, 'r') as file :
7 :           lignes ← file.readlines()
8 :     if not lignes then
9 :         print"Le fichier DIMACS CNF est vide."
10 :        conforme ← False
11 :    end if
12 :    if not lignes[0].startswith('pcnf') then
13 :        print"La première ligne ne commence pas par 'p cnf'."
14 :        conforme ← False
15 :    end if
16 :    for ligne in lignes do
17 :        if ligne.startswith('p cnf') then
18 :                                          ▷ Extraire le nombre de variables et de clauses
19 :            _, _, nb_variables, nb_clauses ← ligne.split()
20 :            nb_variables, nb_clauses ← int(nb_variables), int(nb_clauses)
21 :            if len(lignes) − 1 ≠ nb_clauses then
22 :                print"Le nombre de clauses n'est pas respecté."
23 :                conforme ← False
24 :            end if
25 :        else if ligne.startswith('-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9') then
26 :                                          ▷ Convertir la ligne en une liste d'entiers
27 :            clause ← list(map(int, ligne.split()[: −1]))      ▷ Ignorer le dernier élément (0)
28 :            formule.append(clause)
29 :        end if
30 :    end for
31 :    for num_ligne, ligne in enumerate(lignes[1 :], start=2) do
32 :        if not ligne.endswith('0      n') then
33 :            print"La ligne", num_ligne, "ne se termine pas par 0."
34 :            conforme ← False
35 :        end if
36 :    end for
37 :    return nb_variables, nb_clauses, formule, conforme
38 : end function
```

Les pseudo-algorithmes suivants décrivent respectivement l'extraction de l'affectation ainsi que la validité du fichier :

Algorithm 2 Validation d'affectation

```
1 : function VALIDER_AFFECTATION(affectation, nb_variables)
2 :     if len(affectation) ≠ nb_variables then
3 :         print"Le nombre d'affectations ne correspond pas au nombre de variables dans le
    fichier DIMACS CNF." ,len(affectation), "!=", nb_variables
4 :         return False
5 :     end if
6 :     return True
7 : end function
```

Algorithm 3 Extraction d'affectation depuis un fichier

```
1 : function EXTRAIRE_AFFECTATION(path)
2 :     with open(path, 'r') as file :
3 :         ligne ← file.readline().strip()
4 :                                          ▷ Convertir la ligne en une liste d'entiers
5 :         affectation ← list(map(int, ligne.split()))
6 :     return affectation
7 : end function
```

La solution proposée est d'implémenter une fonction ayant pour objectif de déterminer si une affectation donnée de valeurs booléennes aux variables d'une formule logique en conjonctive normale (CNF) satisfait l'ensemble de la formule. Elle prend en entrée deux paramètres : l'affectation, représentée par une liste de valeurs booléennes associées aux variables, et la formule, constituée d'une liste de clauses.

La fonction parcourt chaque clause de la formule. Pour chaque clause, elle examine individuellement chaque littéral (variable ou négation de variable) qui la compose. Si au moins un littéral dans une clause est évalué à Vrai selon l'affectation donnée, la clause est considérée comme satisfaite, et la fonction passe à la clause suivante. Si, après avoir parcouru tous les littéraux d'une clause, aucun n'est satisfait, la clause est jugée non satisfaite.

Si au moins une clause n'est pas satisfaite, la fonction conclut que l'affectation ne satisfait pas la formule entière et retourne `Faux`. Dans le cas contraire, si toutes les clauses sont satisfaites, la fonction conclut que l'affectation satisfait la formule et retourne `Vrai`.

le pseudo-algorithme suivant décrit les étapes précedantes :

Algorithm 4 Vérification de la satisfaction de la formule

```
1 : function VERIFIER_SAT(affectation, formule)
2 :     for clause in formule do
3 :         clause_satisfaite ← False          ▷ Réinitialiser à False pour chaque nouvelle clause
4 :         for literal in clause do
5 :             if literal ∈ affectation then
6 :                 clause_satisfaite ← True
7 :                 break                      ▷ Sortir de la boucle dès qu'un littéral est satisfait
8 :             end if
9 :         end for
10 :        if not clause_satisfaite then
11 :            return False    ▷ Si la clause n'est pas satisfaite, la formule entière ne peut pas être satisfaite
12 :        end if
13 :    end for
14 :    return True
15 : end function
```

Analyse de la complexité :

Fonction d'extraction des clauses de la formule :

— La lecture de toutes les lignes du fichier a une complexité en temps de $\Theta(n)$, où n est le nombre total de lignes dans le fichier.

— L'extraction du nombre de variables et de clauses à partir de la ligne 'p cnf' a une complexité constante $\Theta(1)$.

— La boucle qui parcourt chaque ligne pour extraire les clauses a une complexité en temps de $\Theta(m)$, où m est le nombre total de littéraux dans toutes les clauses.

— La boucle finale qui vérifie la terminaison des lignes a une complexité en temps de $\Theta(n)$.

— Ainsi, la complexité totale au pire des cas est $\Theta(n + m)$.

Fonction d'extraction de l'affectation :

— La lecture d'une seule ligne dans le fichier a une complexité en temps de $\Theta(1)$.

— La conversion de la ligne en une liste d'entiers a une complexité linéaire $\Theta(m)$, où m est le nombre d'entiers dans la ligne, ainsi le nombre de littéraux.

— Ainsi, la complexité totale au pire des cas est $\Theta(k)$.

Fonction de validation de l'affectation :

— La comparaison du nombre d'affectations avec le nombre de variables a une complexité en temps de $\Theta(1)$.

Fonction de vérification de SAT :

— La fonction parcourt chaque clause et chaque littéral dans la formule. Dans le pire des cas, le dernier littéral de chaque clause est vrai.

La boucle parcours ainsi chaque littéral de chaque clause. La complexité en temps est $\Theta(m*n)$, où m est le nombre total de littéraux dans toutes les clauses, et n le nombre de clause.

La complexité est donc linéaire par rapport à la taille de l'entrée (au pire des cas

m littéraux pour chaque clause, donc taille entrée = m * n. Elle s'effectue ainsi en un temps polynomial.

Performance du programme :

Nous avons opté pour l'environnement *Python* afin d'effectuer nos tests. La figure suivante représente le temps d'exécution en fonction de la variation du nombre de clauses :

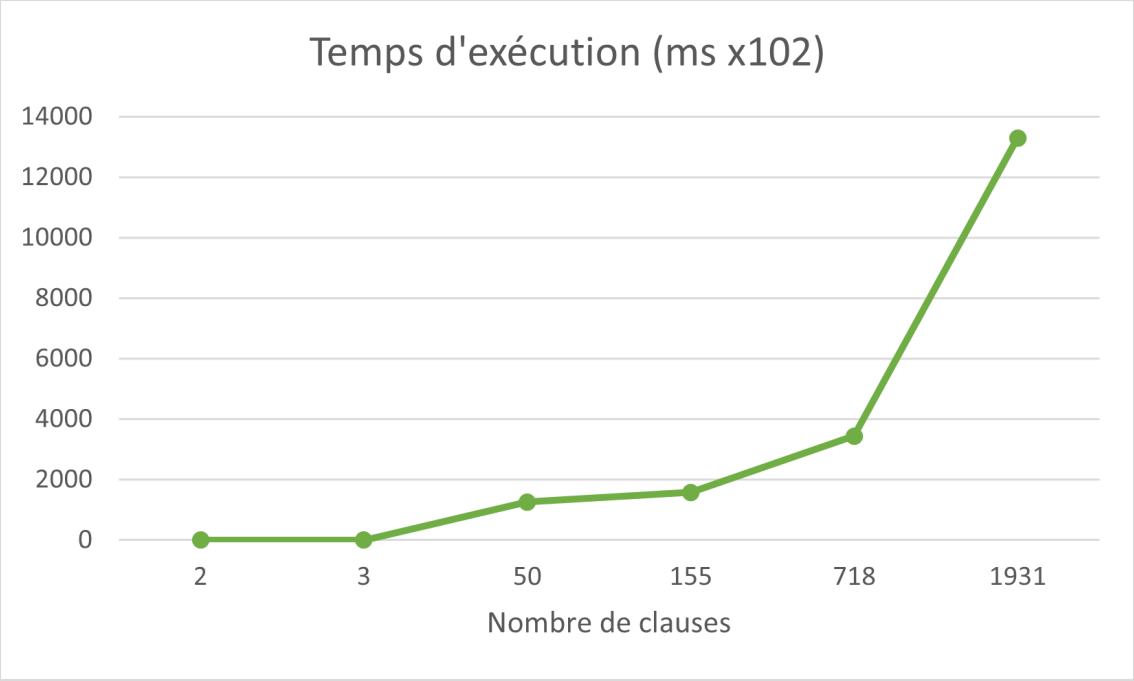

FIGURE 1 – Temps d'exécution en ms *100.

Les résultats montrent que le temps d'exécution est proportionelle au nombre de clauses : plus le nombre de clauses augmente, plus le programme prend du temps à s'exécuter. Cette observation confirme la complexité en temps linéaire de l'algorithme de vérification de SAT. C'est par conséquent un algorithme déterministe polynomial.

Mini-projet 2 : Réduction de Zone Dense à SAT.

Une réduction est une transformation d'un problème en un autre de telle sorte que la résolution du premier problème permette de résoudre le second. Plus formellement, une réduction polynomiale entre deux problèmes de décision A et B signifie qu'il existe une fonction calculable en temps polynomial qui transforme les instances de A en instances de B de manière à préserver la réponse.

Soient A et B deux problèmes de décision. Une réduction polynomiale de A à B, notée $A \leq_p B$, est définie par l'existence d'une fonction calculable en temps polynomial $f : \{0,1\}^* \to \{0,1\}^*$ telle que, pour toute instance x de A, la réponse à x est "oui" si et seulement si la réponse à $f(x)$ est "oui". En d'autres termes, la transformation f conserve la réponse du problème.

Cela signifie que si l'on dispose d'un algorithme efficace pour résoudre le problème B, on peut également résoudre le problème A de manière efficace en utilisant la fonction de réduction f. Ainsi, la difficulté de résoudre le problème A est réduite à celle de résoudre le problème B.

Dans le contexte du problème SAT, la propriété NP-complète de SAT signifie que tout problème dans la classe NP peut être réduit de manière polynomiale au problème SAT. Cela souligne l'importance des réductions dans l'étude de la complexité des algorithmes et fournit un cadre conceptuel pour comprendre les relations entre différents problèmes de décision.

L'objectif de cette deuxième partie du projet est de résoudre le problème de la zone dense dans un graphe non-orienté en utilisant une réduction vers le problème SAT. Plus précisément, il s'agit de déterminer si un graphe donné possède une zone dense de taille **k** ou plus. La réduction consiste à générer une formule en format **DIMACS CNF** à partir du graphe, qui est ensuite soumise au **solver MiniSat**. De plus, une approche "**force brute**" est implémentée pour comparer les performances des deux méthodes. L'algorithme force brute teste exhaustivement toutes les affectations possibles pour déterminer si l'une d'entre elles satisfait la formule résultant de la réduction.

Réduction de Zone Dense à SAT

Considérons la variante du problème de la zone dense, qui équivaut au problème bien connu CLIQUE. Une "zone dense" d'un graphe est simplement une clique du graphe.

Données : Un graphe non-orienté sans boucle $G = (S, A)$ et un entier positif $k \leq |S|$.

Question : Existe-t-il dans G une zone dense de taille k ou plus ?

Pour démontrer une réduction polynomiale vers le problème SAT, nous allons définir une fonction f qui transforme une instance du problème de la clique en une instance équivalente du problème SAT.

Soit $G = (S, A)$ une instance du problème de la clique avec k comme taille de la clique recherchée. Nous construisons une formule booléenne en forme normale conjonctive (FNC) en utilisant des variables booléennes pour représenter les sommets de G et des clauses pour représenter les arêtes de la clique.

Variables propositionelles :

Pour exprimer la relation X_{ij} de manière logique, où X_{ij} est vrai si le sommet i du graphe G est le j-ème sommet de la zone dense, et faux sinon, nous utilisons la notation logique suivante :

$$X_{ij} = \begin{cases} \text{Vrai}, & \text{si le sommet } i \text{ est le } j\text{-ème sommet de la zone dense,} \\ \text{Faux}, & \text{sinon.} \end{cases}$$

Cela peut être exprimé en termes de variables booléennes de la manière suivante :

$$X_{ij} \equiv (\text{le sommet } i \text{ est le } j\text{-ème sommet de la zone dense})$$

Pour le problème de la clique, cela signifie que X_{ij} est vrai si le sommet i est inclus à la position j dans la clique, et faux sinon.

Contraintes : Nous proposons la réduction suivante :

1. Pour chaque sommet i dans le graphe, et pour chaque paire j et j' où $j \neq j'$, le sommet i ne peut pas être à la fois le j-ème et le j'-ème sommet de la zone dense. Cela se traduit par les clauses :

   $$\neg X_{ij} \vee \neg X_{ij'}$$

   où X_{ij} est vrai si le sommet i est le j-ème sommet de la zone dense.

2. Pour chaque paire de sommets distincts i et i' dans le graphe, et pour chaque j de 1 à k, les sommets i et i' ne peuvent pas être simultanément les j-èmes sommets de la zone dense. Cela se traduit par les clauses :

   $$\neg X_{ij} \vee \neg X_{i'j}$$

3. Si i et i' ne sont pas reliés par une arête dans le graphe, alors ils ne peuvent pas être

tous deux dans la zone dense. Cela se traduit par les clauses :

$$\neg X_{ij} \vee \neg X_{i'j}$$

pour chaque paire de sommets i et i' sans arête entre eux, et pour chaque j de 1 à k.

4. Pour chaque j de 1 à k, il doit exister un sommet i tel que i est le j-ème sommet de la zone dense. Cela se traduit par la clause :

$$X_{i_1 j} \vee X_{i_2 j} \vee \ldots \vee X_{inj}$$

où $i_1, i_2, \ldots, in$ sont tous les sommets dans le graphe.

Cette construction définit la fonction de réduction, montrant ainsi que le problème de la zone dense se réduit polynomialement au problème SAT.

Solution proposée

Encodage des variables :

Nous avons préalablement défini une fonction d'encodage, pour exprimer nos variables du type :

$$X_{ij} \equiv \text{(le sommet } i \text{ est le } j\text{-ème sommet de la zone dense)}$$

C'est une fonction utilitaire qui prend deux indices i et j en entrée et renvoie un numéro de variable unique, est utilisée pour générer des variables booléennes correspondant aux sommets du graphe et à leur position dans la clique.

La formule utilisée est $(i-1) \times k + j$, où i représente l'indice du sommet dans le graphe et j représente la position du sommet dans la clique. Les indices commencent à partir de 1 pour être cohérents avec la numérotation des sommets et des positions dans le contexte du problème de la clique. Ainsi, la fonction attribue un identifiant unique à chaque variable booléenne, en tenant compte de la position de la clique et de l'index du sommet.

Le pseudo-algorithme suivant décrit la fonction implémentée :

Algorithm 5 Fonction `var(i, j)`

```
1 : function VAR(i, j)
2 :     return (i − 1) × k + j
3 : end function
```

Analyse de la complexité

La complexité de la fonction `var(i, j)` est constante, car le nombre d'opérations qu'elle effectue ne dépend pas de la taille des entrées i et j. Peu importe les valeurs de i et j, la fonction effectue un nombre fixe d'opérations arithmétiques (additions et multiplications) pour calculer le résultat.

Par conséquent, la complexité de cette fonction est $\Theta(1)$ au pire des cas, parce qu'il n'y a pas de variations dans le nombre d'opérations en fonction de la taille des entrées.

Réduction Zone Dense à SAT :

Cette fonction réalise une réduction du problème de recherche de clique dans un graphe vers le problème SAT (satisfiabilité booléenne). La réduction se fait en générant des clauses logiques qui expriment les contraintes nécessaires pour déterminer l'existence d'une clique de taille k dans le graphe fourni.

1. **Initialisation des variables** : La fonction commence par initialiser des variables telles que le nombre total de sommets (`nb_sommets`), le nombre total de variables (`nb_variables`), et le nombre total de clauses (`nb_clauses`). Une liste `clauses` est également initialisée pour stocker les clauses logiques.
2. **Contrainte 1 :** Chaque sommet ne peut être à la fois le j-ème et le j'-ème de la zone dense. Pour chaque sommet i, deux boucles (j et m) génèrent des clauses négatives empêchant un sommet d'occuper simultanément deux positions différentes dans la clique.
3. **Contrainte 2 :** Les sommets i et i' ne peuvent pas être les j-èmes de la zone dense. Deux boucles (i et m) génèrent des clauses négatives empêchant deux sommets différents d'occuper la même position dans la clique.
4. **Contrainte 3 :** Si deux sommets i et i' ne sont pas reliés par une arête, alors ils ne peuvent pas être simultanément présents dans la zone dense. Deux boucles (i et m) vérifient l'absence d'arête entre i et i', générant des clauses négatives en conséquence.
5. **Contrainte 4 :** Pour chaque position j, il doit exister un sommet i tel que i est le j-ème sommet de la zone dense. Une boucle sur j génère des clauses positives assurant qu'au moins un sommet occupe chaque position de la clique.

Complexité : La complexité de cet algorithme dépend principalement de la taille du graphe, exprimée par le nombre de sommets (`nb_sommets`). La triple boucle utilisée pour générer les clauses impose une complexité quadratique en $\Theta(nb\_sommets^2 \times k)$ dans le pire des cas.

Le pseudo-code suivant décrit la méthode réalisée :

Algorithm 6 Clique to SAT Reduction

```
1 : function CLIQUE_SAT(graph, k)
2 :     nb_sommets ← length of graph
3 :     nb_variables ← nb_sommets × k
4 :     nb_clauses ← 0
5 :     clauses ← []
                                                          ▷ Contrainte 1
6 :     for i ← 1 to nb_sommets do
7 :         for j ← 1 to k do
8 :             for m ← j + 1 to k do
9 :                 clauses.append([−var(i, j), −var(i, m)])
10 :                nb_clauses ← nb_clauses + 1
11 :            end for
12 :        end for
13 :    end for
                                                          ▷ Contrainte 2
14 :    for i ← 1 to nb_sommets do
15 :        for m ← i + 1 to nb_sommets do
16 :            for j ← 1 to k do
17 :                clauses.append([−var(i, j), −var(m, j)])
18 :                nb_clauses ← nb_clauses + 1
19 :            end for
20 :        end for
21 :    end for
                                                          ▷ Contrainte 3
22 :    for i ← 1 to nb_sommets do
23 :        for m ← i + 1 to nb_sommets do
24 :            if graph[i − 1][m − 1] == 0 and graph[m − 1][i − 1] == 0 then
25 :                for j ← 1 to k do
26 :                    clauses.append([−var(i, j), −var(m, j)])
27 :                    nb_clauses ← nb_clauses + 1
28 :                end for
29 :            end if
30 :        end for
31 :    end for
                                                          ▷ Contrainte 4
32 :    for j ← 1 to k do
33 :        clause ← [var(i, j) for i in range(1, nb_sommets + 1)]
34 :        clauses.append(clause)
35 :        nb_clauses ← nb_clauses + 1
36 :    end for
37 :    return nb_variables, nb_clauses, clauses
38 : end function
```

Algorithme brute :

L'algorithme de recherche brute-force proposé pour résoudre une instance du problème SAT génère toutes les combinaisons possibles de valeurs booléennes pour les variables de la formule SAT, puis teste chaque combinaison en appelant la fonction `verifier_SAT`, implémentée dans la première partie. Si une combinaison satisfait la formule, elle est affichée à la console, enregistrée dans un fichier, et la fonction retourne `True`. Si aucune solution n'est trouvée après avoir testé toutes les combinaisons, la fonction retourne `False`. Ce type d'algorithme explore exhaustivement l'espace des solutions, ce qui peut être inefficace pour des instances de problèmes de grande taille, mais garantit la découverte de solutions s'il en existe.

Le pseudo-algorithme suivant représente la fonction utilisée

Algorithm 7 brute_reduit(formule, nombre_variables)

```
valeurs_variables ← [[i, −i] pour i de 1 à nombre_variables]
for chaque combinaison dans produit_cartesien(∗valeurs_variables) do
    if verifier_SAT(combinaison, formule) then
        Afficher combinaison
        data ← convertir_en_chaine(combinaison).supprimer_caracteres('(',')','')
        Écrire data dans le fichier 'affectation_brute.txt'
        return Vrai
    end if
end for
return Faux
```

Analyse de la complexité : Dans le pire des cas, l'algorithme de recherche brute-force a une complexité exponentielle en fonction du nombre de variables dans la formule SAT. Plus précisément, si n est le nombre total de variables, la complexité est $O(2^n \cdot f)$, où f est la complexité de la fonction `verifier_SAT`. Cette complexité exponentielle résulte de la nécessité d'explorer toutes les 2^n combinaisons possibles de valeurs booléennes pour les variables. Ainsi, le temps d'exécution de l'algorithme augmentera de manière exponentielle à mesure que le nombre de variables dans la formule SAT augmente, rendant l'algorithme impraticable pour des instances de problèmes de grande taille.

Minisat :

MiniSat est un solveur SAT open-source utilisé pour résoudre des problèmes de satisfiabilité booléenne. Son objectif est de déterminer si une formule booléenne donnée est satisfaisable, ce qui signifie qu'il peut trouver une assignation de valeurs aux variables qui rend la formule vraie. Le problème SAT est NP-complet, mais MiniSat utilise des techniques avancées telles que le backtracking, la propagation de contraintes, et la clause learning pour résoudre efficacement des instances de taille pratique.

Comparaison des performances

Nous avons opté pour l'environnement *Python* afin d'effectuer nos tests. La figure suivante représente le temps d'exécution du solveur Minisat et de l'algorithme brut implémenté en fonction de la variation du nombre de sommets du graphe G :

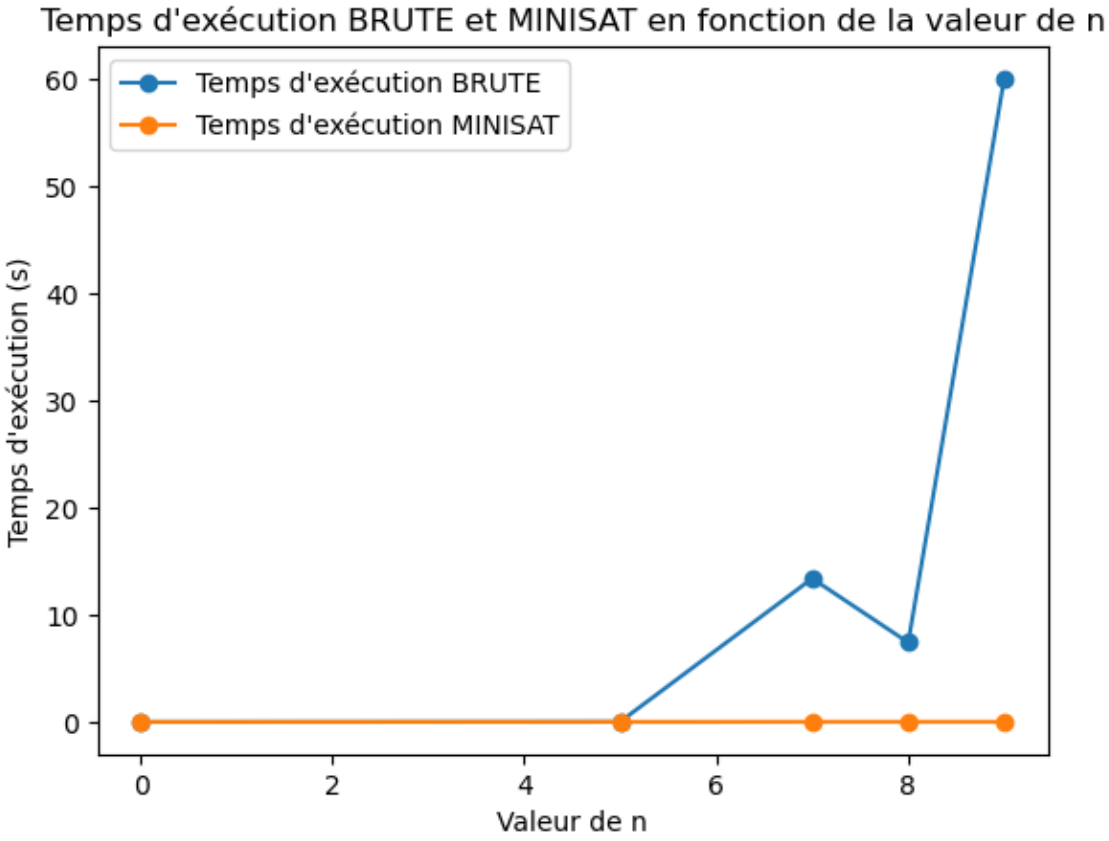

FIGURE 2 – Temps d'exécution en s du Minisat vs Brut.

Les résultats montrent clairement que l'algorithme brute devient rapidement inefficace en terme de temps d'exécution à mesure que n augmente en raison de sa complexité exponentielle.

MiniSat est plus performant grâce à l'utilisation d'heuristiques sophistiquées, telles que la règle de pureté et le clause learning, ainsi que des stratégies d'exploration efficaces. Ces techniques permettent à MiniSat de guider la recherche vers une solution plus rapidement, même pour des problèmes de grande taille, en évitant des branches inutiles de l'espace de recherche.

En revanche, la recherche brute, en explorant de manière exhaustive toutes les combinaisons possibles, peut devenir rapidement impraticable en raison de son coût exponentiel, surtout lorsque le nombre de variables ou les possibilités de combinaisons augmentent. Les performances de la recherche brute-force diminuent considérablement avec la complexité du problème, ce qui la rend moins adaptée à des instances de grande taille.

Mini-projet 3 : Réduction de Sudoku à SAT.

Le jeu Sudoku est un puzzle logique qui se joue sur une grille carrée de neuf cases de côté, subdivisée en neuf régions carrées égales. L'objectif du jeu est de remplir chaque case de la grille avec un chiffre de 1 à 9, de telle sorte que chaque ligne, chaque colonne et chaque région contienne chaque chiffre exactement une fois. La grille initiale comporte déjà certains chiffres, appelés les "indices" ou "pré-remplissages", qui servent de point de départ pour la résolution du puzzle.

Le but de cette dernière partie du projet est d'implémenter une réduction de Sudoku vers SAT.

La réduction du problème Sudoku vers SAT implique de transformer une instance du problème Sudoku en une formule booléenne dans le format CNF (Conjunctive Normal Form) pour qu'elle puisse être résolue par un solveur SAT tel que MiniSat.

Réduction de Sudoku à SAT

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

Considérons une version généralisée du jeu Sudoku où les données consistent en une grille de taille $n^2 \times n^2$ où chaque case peut contenir un entier dans l'intervalle $[1, n]$, ou bien elle peut être vide. La question associée est de savoir s'il existe une façon de remplir les cases vides de la grille avec des entiers dans l'intervalle $[1, n]$ de manière à ce que chacune des n^2 lignes, colonnes et régions de la grille ne contienne qu'une fois tous les entiers de l'intervalle. Un exemple d'instance de ce problème est une grille de taille $3^2 \times 3^2$ où chaque cellule peut contenir un entier dans l'intervalle $[1, 9]$, ou elle peut être vide. L'objectif est de remplir les

cases vides de manière à ce que chaque ligne, colonne et région de la grille ne contienne qu'une fois tous les entiers de l'intervalle [1, 9].

Variables propositionelles :

Nous avons exprimer les cases du Sudoku grace à une variable booléenne à chaque proposition possible dans la grille.

Par exemple, pour une grille de taille $n^2 \times n^2$, chaque cellule peut contenir un chiffre de 1 à n^2, soit n lignes, n colonnes et n régions.

Donc, il y aura n^3 variables booléennes au total. Chaque variable X_{ijk} représente la proposition que la cellule à la ligne i, colonne j contient le chiffre k. Les indices sont définis comme suit : $1 \leq i, j, k \leq n^2$.

Cela peut être exprimé en termes de variables booléennes de la manière suivante :

$$X_{ijk} \equiv (\text{la cellule à la ligne } i \text{ et la colonne } j \text{ contient le chiffre } k)$$

Ainsi, X_{ijk} est définie pour chaque combinaison possible de ligne, colonne et valeur de chiffre, permettant de modéliser de manière compacte toutes les propositions de remplissage de la grille de Sudoku.

Contraintes :

Nous proposons la réduction suivante :

1. Chaque cellule contient au moins un chiffre : Pour chaque cellule de la grille, une clause est générée, garantissant qu'au moins l'un des chiffres de 1 à n est présent dans cette cellule.

2. Chaque cellule contient au plus un chiffre : Pour chaque cellule de la grille, des clauses sont ajoutées, assurant qu'au plus un chiffre de 1 à n est présent dans cette cellule. Cela est réalisé en ajoutant des clauses avec des paires de littéraux négatifs pour chaque combinaison de deux chiffres différents.

3. Chaque chiffre apparaît au moins une fois dans chaque ligne et colonne : Pour chaque chiffre de 1 à n, une clause est générée pour chaque ligne et chaque colonne, assurant qu'au moins une cellule contient ce chiffre.

4. Chaque chiffre apparaît au plus une fois dans chaque ligne et colonne : Pour chaque chiffre de 1 à n, des clauses sont ajoutées pour chaque ligne et chaque colonne, garantissant qu'au plus une cellule contient ce chiffre. Cela est réalisé en ajoutant des clauses avec des paires de littéraux négatifs pour chaque combinaison de deux cellules différentes.

5. Chaque chiffre apparaît au plus une fois dans chaque région : Pour chaque chiffre de 1 à n, des clauses sont générées pour chaque région, assurant qu'au plus une cellule de chaque région contient ce chiffre. Cela est réalisé en ajoutant des clauses avec des paires de littéraux négatifs pour chaque combinaison de deux cellules différentes dans la même région.

6. Contraintes pour les chiffres préexistants dans la grille : Ajouter des clauses spécifiques pour chaque chiffre déjà présent dans la grille. Ces clauses assurent que les chiffres préexistants sont correctement pris en compte dans la solution SAT générée.

Cette construction définit la fonction de réduction, montrant ainsi que le problème du

sudoku se réduit au problème SAT.

Solution proposée

Encodage des variables :

Nous avons préalablement défini une fonction d'encodage, pour exprimer nos variables du type :

$$X_{ijk} \equiv \text{(la cellule à la ligne } i \text{ et la colonne } j \text{ contient le chiffre } k)$$

La fonction `variable(i, j, k)` est utilisée pour créer une variable booléenne associée à une proposition spécifique dans le contexte de la réduction de Sudoku vers SAT. La grille de Sudoku est représentée comme une grille de $n^2 \times n^2$ cellules, où chaque cellule peut contenir un chiffre de 1 à n. Les indices i, j, et k décrivent la position de la cellule dans la grille et le chiffre qu'elle contient.

La formule `return (i - 1) * n**2 + (j - 1) * n + k` crée une variable booléenne unique pour chaque combinaison de position de cellule et de chiffre. Voici comment cela fonctionne :

- La variable associée à la cellule à la ligne i, à la colonne j, et avec le chiffre k est calculée en utilisant une formule mathématique.
- L'indice de la ligne i et de la colonne j est converti en un indice unique pour la cellule en multipliant par n^2. Cela garantit que chaque ligne et chaque colonne contribuent de manière unique à l'indice de la variable.
- L'indice de la variable est ensuite ajusté en ajoutant k. Cela garantit que chaque variable associée à une cellule donnée a un indice unique, compte tenu du chiffre k qu'elle représente.

Supposons que $n = 3$ et que nous voulons représenter la variable pour la cellule située à la deuxième ligne, troisième colonne et contenant le chiffre 4.

En utilisant la formule, la variable associée serait $((2 - 1) \times 3^2) + ((3 - 1) \times 3) + 4 = 23$. Ainsi, la variable booléenne associée à cette cellule et à ce chiffre spécifiques serait la 23ème variable dans le contexte de la réduction SAT du Sudoku.

Réduction Sudoku à SAT :

Les pseudo-algorithmes suivants décrivent la méthode utilisée pour réduire Sudoku vers SAT :

Algorithm 8 Manipuler le nombre existant

variable variable
manipulationNumeroExistant manipulationNumeroExistant
k, i, j, n, clauses ▷ Il faut le prendre (ajouter une clause de taille 1, avec le littéral correspondant à "mettre le nombre k sur [i,j]")
$c \leftarrow [i, j, k]$ $clauses$.append(c)
▷ Il ne doit plus y avoir le numéro k sur la ligne i
for $m \leftarrow 1$ n **do**
  if $i \neq m$ **then** $c \leftarrow [-m, j, k]$ $clauses$.append(c)
  ▷ Il ne doit plus y avoir le numéro k sur la colonne j
  for $l \leftarrow 1$ n **do**
    if $j \neq l$ **then** $c \leftarrow [-i, l, k]$ $clauses$.append(c)
    ▷ Il ne doit plus y avoir de numéro autre que k sur le carré [i,j]
    for $p \leftarrow 1$ n **do**
      if $k \neq p$ **then** $c \leftarrow [-i, j, p]$ $clauses$.append(c)

La fonction `manipulationNumeroExistant` est conçue pour traiter les chiffres déjà présents dans une grille de Sudoku. Supposons que la grille a une taille $n \times n$, où n est la taille d'une région carrée. La fonction prend en entrée le chiffre k et les coordonnées (i, j) d'une cellule dans la grille. Elle utilise ces informations pour générer des clauses logiques, qui sont essentielles pour décrire les conditions nécessaires et suffisantes pour résoudre le Sudoku. Voici comment chaque partie de la fonction opère :

1. **Ajout de la clause pour le chiffre existant :** La première ligne de la fonction ajoute une clause avec une seule variable. Cette variable correspond à la présence du chiffre k dans la cellule (i, j) de la grille.
2. **Suppression des occurrences du chiffre dans la même ligne, colonne et carré :** Les boucles suivantes génèrent des clauses pour s'assurer que le chiffre k n'apparaît pas ailleurs dans la même ligne, colonne et carré.
   — La première boucle parcourt toutes les colonnes m de la même ligne i et ajoute une clause négative indiquant que k ne doit pas être dans la cellule (m, j).
   — Les deuxième et troisième boucles fonctionnent de manière similaire pour les colonnes et les lignes respectivement.
   — La dernière boucle parcourt les cellules du même carré que (i, j) et ajoute des clauses négatives pour garantir l'absence de k dans ces cellules.

Algorithm 9 Convertir Sudoku en SAT

sudokuToSatsudoku_to_sat
grille $n \leftarrow \text{len}(grille)$ $clauses \leftarrow []$

▷ Définir la fonction variable

variable variable i, j, k **Retourner** $(i-1) \times n^2 + (j-1) \times n + k$

▷ Chaque cellule contient au moins un chiffre

for $i \leftarrow 1$ n **do**

for $j \leftarrow 1$ n **do** $clauses$.append($[i, j, k$ pour $k \leftarrow 1$ à $n]$)

▷ Chaque cellule contient au plus un chiffre

for $i \leftarrow 1$ n **do**

for $j \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do**

for $l \leftarrow k+1$ n **do** $clauses$.append($[-i, j, k, -i, j, l]$)

▷ Chaque chiffre apparaît au moins une fois dans chaque ligne

for $i \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do** $clauses$.append($[i, j, k$ pour $j \leftarrow 1$ à $n]$)

▷ Chaque chiffre apparaît au moins une fois dans chaque colonne

for $j \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do** $clauses$.append($[i, j, k$ pour $i \leftarrow 1$ à $n]$)

▷ Chaque chiffre apparaît au plus une fois dans chaque ligne

for $i \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do**

for $j \leftarrow 1$ n **do**

for $l \leftarrow j+1$ n **do** $clauses$.append($[-i, j, k, -i, l, k]$)

▷ Chaque chiffre apparaît au plus une fois dans chaque colonne

for $j \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do**

for $i \leftarrow 1$ n **do**

for $l \leftarrow i+1$ n **do** $clauses$.append($[-i, j, k, -l, j, k]$)

▷ Chaque chiffre apparaît au plus une fois dans chaque région

for $r \leftarrow 1$ n **do**

for $k \leftarrow 1$ n **do**

for $i \leftarrow 1$ n **do**

for $j \leftarrow 1$ n **do**

for $l \leftarrow j+1$ n **do**

for $m \leftarrow 1$ n **do**
$clauses$.append($[-n \times (i-1) + j, n \times (j-1) + l, k, -n \times (i-1) + j, n \times (j-1) + m, k]$)

▷ Vérifier et manipuler les chiffres existants dans la grille

for $i \leftarrow 1$ n **do**

for $j \leftarrow 1$ n **do**

if grille$[i-1][j-1] \neq 0$ **then manipulationNumeroExistant**
manipulationNumeroExistant
manipulationNumeroExistant(grille$[i-1][j-1], i, j, n, clauses$)

Retourner $clauses$

La fonction `sudoku_to_sat` prend une grille de Sudoku en entrée et la convertit en un ensemble de clauses logiques en utilisant la méthode de réduction vers SAT (*satisfiability problem*). Voici une explication détaillée du fonctionnement de cette fonction :

Création des variables : La fonction définit une fonction interne `variable(i, j, k)` qui attribue un numéro unique à chaque combinaison de ligne, colonne et chiffre. Cela permet de créer des variables booléennes pour chaque cellule de la grille et chaque chiffre possible.

Contraintes de base pour les cellules :

— La première boucle imbriquée ajoute des clauses pour s'assurer qu'au moins un chiffre est présent dans chaque cellule de la grille.

— La deuxième boucle imbriquée ajoute des clauses pour garantir qu'au plus un chiffre est présent dans chaque cellule.

Contraintes pour chaque ligne, colonne et région :

— Les boucles suivantes ajoutent des clauses pour garantir que chaque chiffre apparaît au moins une fois dans chaque ligne, colonne et région.

— Les boucles subséquentes ajoutent des clauses pour garantir qu'au plus un chiffre est présent dans chaque ligne, colonne et région.

Contraintes pour les chiffres déjà présents : Appel de la fonction `manipulationNumeroExistant` pour chaque cellule déjà remplie dans la grille. Cette fonction ajoute des clauses pour respecter les chiffres déjà présents dans la grille.

Analyse de la complexité :

La complexité en θ au pire des cas de la fonction `sudoku_to_sat` dépend de plusieurs facteurs, notamment la taille de la grille de Sudoku ($n^2 \times n^2$). Analysons les composants principaux de la fonction :

Création des variables : Il y a $3n^3$ variables booléennes générées, où n est la taille de la grille. La création de ces variables a une complexité en $\theta(n^3)$.

Contraintes de base pour les cellules :

— Les clauses garantissant qu'au moins un chiffre est présent dans chaque cellule ont une complexité en $\theta(n^3)$.

— Les clauses garantissant qu'au plus un chiffre est présent dans chaque cellule ont une complexité en $\theta(n^4)$.

Contraintes pour chaque ligne, colonne et région :

— Les clauses pour chaque ligne, colonne et région, en termes de garantir la présence d'au moins un chiffre, ont une complexité en $\theta(n^4)$.

— Les clauses pour chaque ligne, colonne et région, en termes de garantir la présence d'au plus un chiffre, ont une complexité en $\theta(n^5)$.

Contraintes pour les chiffres déjà présents : La manipulation des chiffres déjà présents a une complexité en $\theta(n^3)$ puisqu'elle parcourt chaque cellule.

En conclusion, la complexité totale de la fonction `sudoku_to_sat` est dominée par les termes quadratiques et cubiques, ce qui donne une complexité en $\theta(n^5)$ au pire des cas. Cela signifie

que la complexité de l'algorithme croît de manière polynomiale en fonction de la taille de la grille de Sudoku.

Analyse des performances

Nous avons opté pour l'environnement *Python* afin d'effectuer nos tests. La figure suivante représente le temps d'exécution de la réduction vers SAT et du solveur Minisat en fonction de la variation de la taille n :

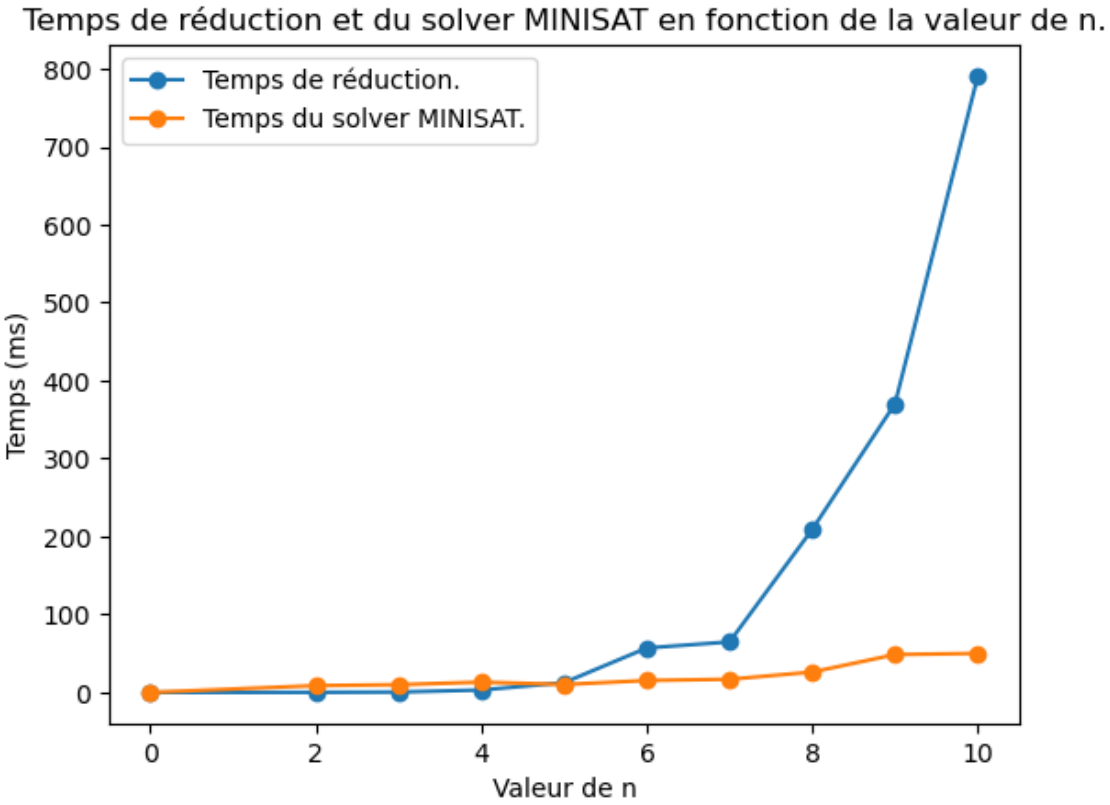

FIGURE 3 – Temps de réduction et du solveur Minisat en ms.

Les résultats nous montrent que effectivement, le temps d'exécution l'algorithme croît de manière polynomiale en fonction de la taille de la grille de Sudoku. Ce qui suggère que la résolution de données de grande taille peut être coûteuse en termes de temps.

Le Sudoku, en tant que problème NP-complet, est connu pour avoir une complexité algorithmique élevée, en particulier lorsque la grille devient plus grande. Pour les grilles de taille standard (9x9), les algorithmes de résolution sont généralement rapides, mais à mesure que la taille augmente, la résolution exacte peut devenir impraticable.