MASTER 1 INFORMATIQUE UE COMPLEXITÉ

RENDU TP2 GROUPE 4

RAVET Alexis, MOUZNI Lamara, TINTORRI Floren, SMAIL Aghilas, ILLA mohamed abdallahi.

1. Sommaire

2. Mini-projet 1. Vérificateur déterministe pour SAT

2.1. Introduction

Nous devons implémenter un vérificateur déterministe pour SAT qui travaille sur des fichiers en format DIMACS CNF ainsi qu'un fichier contenant une affectation possible. Dans ce mini projet nous allons lire deux fichiers, les transformer en objet java afin de tester l'affectation proposée si celle-ci satisfait ou pas notre fichier CNF donné.
On vérifiera également la validité des documents données (construction cohérente).

2.2. Execution & sortie attendu

Le programme s'exécute via main en renseignant les champs :

```
static final String nameFile = "v155"; // à changer
static final String sufix = "model"; //true, false, model
```

Nous avons plusieurs sortie possible attendu pour les cas d'erreurs diverses avec comme sortie attendu :

[duree ms] construction CNF : 12
Les litéraux donnés sont une solution du CNF donné
[duree ms] validation Litéraux : 17

2.3. Déroulement du programme et complexité

On commence par l'initialisation de plusieurs variables utiles $\theta(1)$
Le main début par l'appel à la méthode

```
private static void resetClock()
```

Qui ne fait que réinitialiser la variable de la durée en la fixant à l'heure courant. $\theta(1)$

On crée et initialise un objet CNF : O(Cnf)
println() et resetClock() : $\theta(1)$
On crée et initialise un Literaux Validator() : O(LV)
Condition if avec test de validity : $\theta(1)$
2 ou 3 println() avec calcule : $\theta(1)$

O(Cnf) : θ(nbClause * (nb_literal ^2))
Définition de variable θ(1)
Lecture d'un fichier θ(n * nombre_char_lu)
Initialisation des variables nb* : θ(1)

Boucle de construction de clause (0 à nbClauses) θ(nbClause * (nb_literal ^2))
    If avec test sur la bonne terminaison de clause θ(1)
    Création d'un tableau de littéral représentant une clause: θ(1)
    Boucle pour initialiser la clause pour chaque littéral lu θ(nb_literal ^2)
        Affectation θ(1)
        Vérification du non dépassement du nombre de littéraux θ(1)
    ajout du littérale θ(1)
ajout de la clause θ(1)
Fermeture de lecture du fichier θ(n)

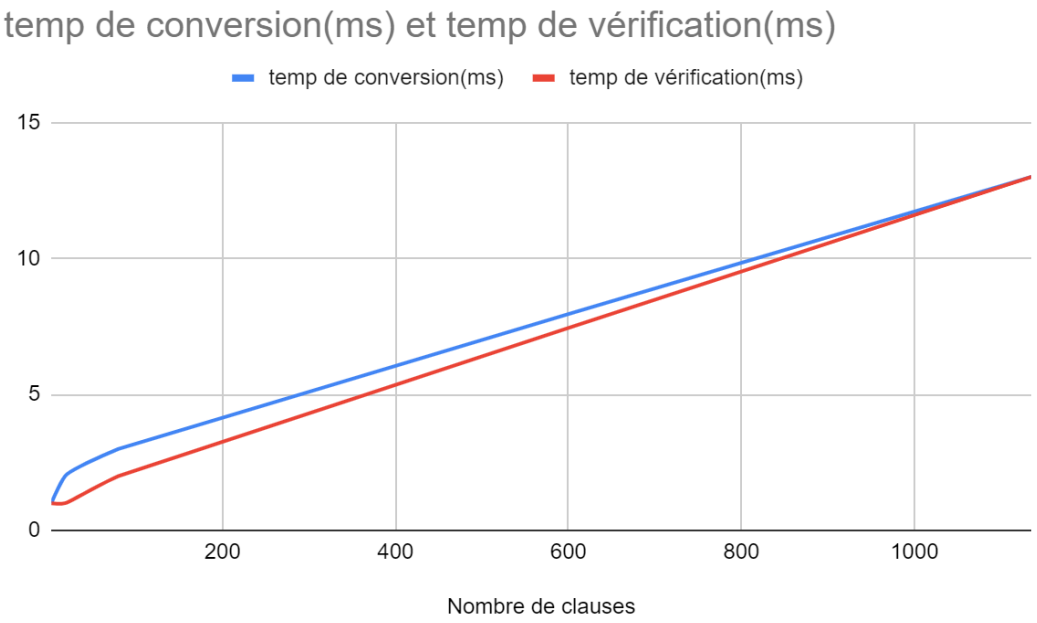
O(LV) : θ(nbClause * (nb_literal ^2))
Définition de variable : θ(1)
Lecture d'un fichier : θ(n * nombre_char_lu)
Test de ligne vide : θ(1)
Split de ligne θ(1) (temps constant)

Boucle vérifiant le nombre de littéraux θ(nbDataLu)
    test θ(4)

Boucle de vérification pour chaque clause θ(nb_literal * nb_clause)
    Boucle de vérification pour chaque litéral θ(nb_literal)
        test littéral valide θ(3)
Conclusion :
Complexité de FNCValidator :n θ(|nbOfClauses|*|nbOfLiteral|^2)

2.4. Schéma du temps d'exécution en fonction des entrées

Nombre de variables	Nombre de clauses	temp de conversion(ms)	temp de vérification(ms)
3	2	1	1
16	18	2	1
50	80	3	2
155	1135	13	13


temp de conversion(ms) et temp de vérification(ms)
temp de conversion(ms)
temp de vérification(ms)
15
10
5
0
200
400
600
800
1000
Nombre de clauses

3. Mini-projet 2. Réduction de Zone Vide à SAT

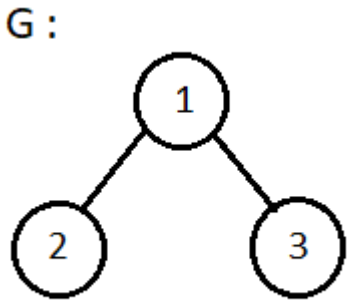
3.1. Introduction

Dans cette partie, nous devons implémenter un algorithme qui permet de réduire le problème zone vide vers le problème SAT. Cet algorithme créera en sortie un fichier sous le format DIMACS CNF qu'on donnera en entrée au solveur SAT Kissat-MAB.

3.2. La réduction

L'idée de la réduction est de parcourir toutes les arêtes (i, j) du graphe G donné en entrée, et pour chacune d'entre elles, on dit que le sommet source i et le sommet destination j ne peuvent pas être tous les deux dans la zone vide (soit l'un soit l'autre soit aucun des deux). Donc on aura autant de variables du problème SAT que de sommets dans le graph G, et autant de clauses que de nombre d'arêtes. L'exemple ci-dessous illustre cet algorithme.

Exemple :



1 - On a autant de variables que de sommets : donc les variables de notre problème sat seront 1, 2 et 3.
2 - Pour chaque arête, soit le sommet source est dans la zone vide soit le sommet destination soit aucun mais pa les deux en même temps:

- pour l'arête (1, 2) on aura la clause (-1 ou -2)
- pour l'arête (1, 2) on aura la clause (-1 ou -3)

Donc la on aura la formule : f = (-1 ou -2) et (-1 ou -3)

La table de vérité :

1	2	3	(-1 ou -2)	(-1 ou -3)	f
0	0	0	1	1	1
0	0	1	1	1	1

0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	0	1	0
1	1	1	0	0	0

On voit que la formule f est satisfaite lorsque les variables associées au sommets formant une zone vide sont à vrai.

3.3. Déroulement du programme et complexité

Le graphe est représenté sous forme d'une matrice de valeurs 0 ou 1, tels que si la case (i, j) est à 1, il existe une arête entre le sommet i et le sommet j, et sinon la case est à 0.
Les clauses sont représentées par une classe Clause qui contient une liste d'entiers qui représente les littéraux.

La fonction **stableToSAT** prend en paramètre une matrice **graph** qui représente un graphe, et retourne une liste de clauses. Et pour faire cela, elle suit l'algorithme suivant:

Entrées :
graph : le graphe à réduire
L'algorithme :
Initialiser une liste **clauses** à une liste vide.
pour i de 0 à la taille du graphe **graph** :
pour j de i à la taille du graphe **graphe** :
si (graph[i][j] = 1) alors ajouter une clause (-i, -j) à la liste **clauses**.
fin si
fin pour
fin pour
Retourne : clauses

La Complexité : La complexité de l'algorithme est par rapport à la taille du graph données en entrées n est : n+(n-1)+(n-2)+…+2+1 = (n^2*(n-1))/2
donc la complexité de l'algorithme est : θ(n^2)

Donc la réduction se fait en temps polynomial.

3.4. Schéma du temps d'exécution en fonction des entrées

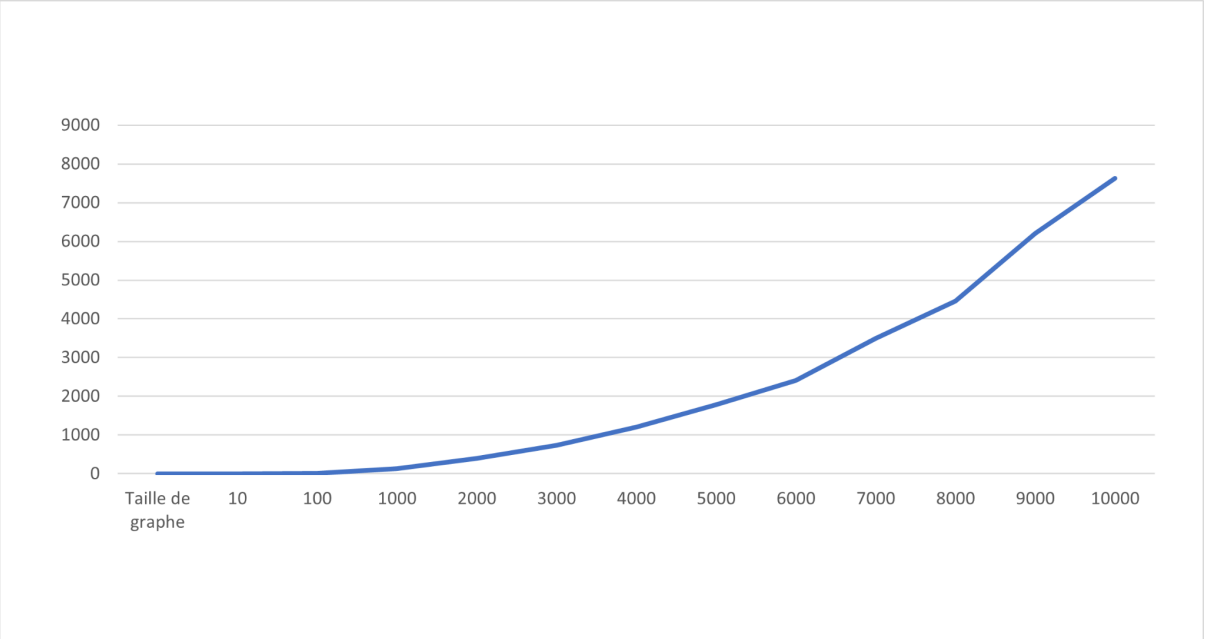


Figure : Temps de réduction * la taille du graphe.

4. Mini-projet 3. Réduction de Sudoku à SAT

4.1. Introduction

Ce projet à pour but d'implémenter une réduction de Sudoku vers SAT . Il donnera en format DIMACS CNF . Il est demandé aussi de vérifier le temps d'exécution de la réduction plus le temps d'exécution du SAT solver sur des instances de taille croissante.

4.2. Execution & sortie attendu

Pour éxécute la fonction , il faut depuis le terminal utiliser la commande : javac Sudoku2SAT.java <nom du fichier> Mais aussi on peut bien utilise cet commande : javac Sudoky2SAT.java <nom du fichier> * qui va permettre afficher les litéraux en mode ligne,colonne et chiffre et donc savoir les valeurs émises.Il sortira un fichier en format DIMACS CNF

4.3. Déroulement du programme et complexité

Pour récupérer les données du sudoku , on utilise la fonction `private static void lecture(String file)`
La fonction a une complexité de θ(n) ou n est le nombre de caractère du fichier.Il sera ensuite stocké dans la variable grâce a cette ligne :`matrice[compteur_ligne][i]= Integer.parseInt(séparationLigne[i])`
Convertir le caractère contenant le nombre en chiffre et le stocké est de θ((n^2)*|l|). `int[][] lecture(String file)`
est la même avec un ajout constant d'une ligne au début et de 0 à la fin de chaque ligne suivants . Ces complexités sont équivalentes.
Ensuite on utilise la méthode `String convertion`
qui va appelé toute les méthodes
:`manipulationUneParLigne()` ajoute une clause qui dira le nombre doit être dans cette ligne c'est a dire au moins au moins chaque numéro est présent dans une des colonnes de cette ligne
`manipulationUneParColonne()`: ajoute une clause qui dira le nombre doit être dans cette ligne c'est à dire au moins chaque numéro est présent dans une des colonnes de cette colonne
`manipulationNumeroExistant`
:si le numéro est déjà présent dans le sudoku, on applique des axiomes comme ajouter une clause de taille 1 ,avec le littéral correspondant à 'mettre le nombre k sur [i,j]
`manipulationAuMaxUnNuméro`: qui regarde un maximum d'un nombre présent et donc ajoute une clause qui dira qu'au plus un numéro doit être pressé.

`manipulationAuMoinUnNuméro`: qui regarde un minimum d'un nombre présent et donc ajoute une clause qui dira qu'au plus un numéro doit être pressé.
`manipulationDesRegions`: qui regarde si un nombre k existe dans une région alors il ne doit pas exister dans les autres cellules de la même région et donc ajoute alors une clause

Sa complexité algorithmique est de MAX[θ(n*(n^3 + n^2)), θ(n^2)] donc de θ(n*(n^3 + n^2)). En simplifiant nous arrivons à θ(n^4). La taille en entrée étant une matrice n^2, nous en concluant que notre programme de réduction du problème de sudoku vers un problème SAT est quadratique sur la taille de l'entrée: θ((|taille entrée|)^2).

4.4. Schéma du temps d'exécution en fonction des entrées

taille de la grille du sudoku	temp de conversion en objet Java(ms)	temp de réduction vers sat(ms)
9	1	42
16	1	102
25	2	272