TP2: Trois mini Projets

Groupe numéro 24

Auteurs :

* JLIDI Moetez
* KARTOUT Ahmed
* KHACHIA ERRAHMAN El Hassan
* KASMI Badreddine
* KOUHHIZ Imane

Contents

1 Exercice 1: Le tri deux-tiers

1.1 Explication du fonctionnement de l'algorithme

Le tri deux-tiers est un algorithme récursif structuré en trois étapes :

- **Étape (a) :** Trier les premiers 2/3 du tableau.
- **Étape (b) :** Trier les derniers 2/3 du tableau.
- **Étape (c) :** Trier de nouveau les premiers 2/3 du tableau.

Pour les tableaux de taille inférieure ou égale à 2, le tri est trivial, car on peut directement comparer et ordonner les éléments.

1.2 Pourquoi l'algorithme trie correctement

L'idée clé du tri deux-tiers repose sur le fait que chaque partie du tableau est triée plusieurs fois, ce qui conduit finalement à un tableau entièrement trié. Bien que seules les premières et dernières 2/3 parties du tableau soient triées à plusieurs reprises, le chevauchement entre ces sections garantit que tous les éléments sont finalement ordonnés correctement. En triant récursivement les sections qui se chevauchent, les éléments mal placés sont déplacés dans les bonnes positions.

1.3 Explication du code en Java

L'algorithme de tri deux-tiers est implémenté en Java comme suit :

```
import java.util.Arrays;
import java.util.Random;

public class TwoThirdSort {

    public static void twoThirdSort(int[] arr, int start, int end, int depth) {
        if (end - start <= 0) {
            return;
        } else if (end - start == 1) {
            if (arr[start] > arr[start + 1]) {
                int temp = arr[start];
                arr[start] = arr[start + 1];
                arr[start + 1] = temp;
            }
            return;
        }
        int twoThird = (int) Math.ceil((2 * ((double) end - (double) start + 1)) / 3);

        twoThirdSort(arr, start, start + twoThird - 1, depth + 1);
        twoThirdSort(arr, end - twoThird + 1, end, depth + 1);
        twoThirdSort(arr, start, start + twoThird - 1, depth + 1);
    }

    public static int[] generateRandomArray(int size) {
```

```
        Random rand = new Random();
        int[] arr = new int[size];
        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(100);
        }
        return arr;
    }

    public static void main(String[] args) {
        int[] arr = generateRandomArray(10000);
        System.out.println("Avant le tri: " + Arrays.toString(arr));
        twoThirdSort(arr, 0, arr.length - 1, 0);
        System.out.println("Après le tri: " + Arrays.toString(arr));
    }
}
```

L'algorithme de tri deux-tiers peut être décomposé comme suit :

- **Fonction de tri** : La méthode principale `twoThirdSort` prend en entrée un tableau `arr`, des indices de début et de fin, et un paramètre de profondeur (pour le suivi de la récursion). Elle divise et trie récursivement les sections de tableau correspondant aux deux-tiers.
- **Cas de base** : Lorsque la sous-partie à trier contient 1 ou 0 éléments, la fonction retourne sans effectuer d'opérations supplémentaires. Si elle contient deux éléments, ils sont simplement comparés et éventuellement échangés.
- **Étapes du tri** :
  - Calculer 2/3 de la portion du tableau à trier.
  - Appliquer le tri sur les premiers 2/3, les derniers 2/3, puis encore une fois sur les premiers 2/3.
- **Tableaux aléatoires** : La méthode `generateRandomArray` génère un tableau de taille donnée, rempli de valeurs aléatoires entre 0 et 100.

1.4 Analyse de la complexité dans le pire cas à l'aide du théorème maître

Nous analysons maintenant la complexité temporelle du tri deux-tiers dans le pire cas en utilisant le théorème maître.

L'algorithme divise le tableau en deux sous-problèmes de taille $\frac{2}{3}n$, et chaque appel récursif fait trois appels de cette taille. Le coût de la division et de la combinaison est linéaire, c'est-à-dire $O(n)$. Cela nous donne une relation de récurrence de la forme :

$$T(n) = 3T\left(\frac{2}{3}n\right) + O(n)$$

Appliquons maintenant le théorème maître à cette relation de récurrence :

- Ici, $a = 3$, $b = \frac{3}{2}$, et $d = 1$.
- On calcule $\log_b a = \log_{\frac{3}{2}} 3 \approx 1.71$.

En comparant $d = 1$ et $\log_b a \approx 1.71$, nous observons que $\log_b a > d$, ce qui signifie que le terme dominant est $T(n) = O(n^{\log_b a})$. Par conséquent, la complexité dans le pire cas est :

$$T(n) = O(n^{\log_{\frac{3}{2}} 3}) \approx O(n^{1.71})$$

1.5 Expérimentation et graphique

Pour tester l'algorithme, nous avons généré des tableaux remplis aléatoirement. Ensuite, nous avons mesuré le temps d'exécution pour différentes tailles de tableaux et tracé un graphique montrant la relation entre la taille du tableau et le temps d'exécution.

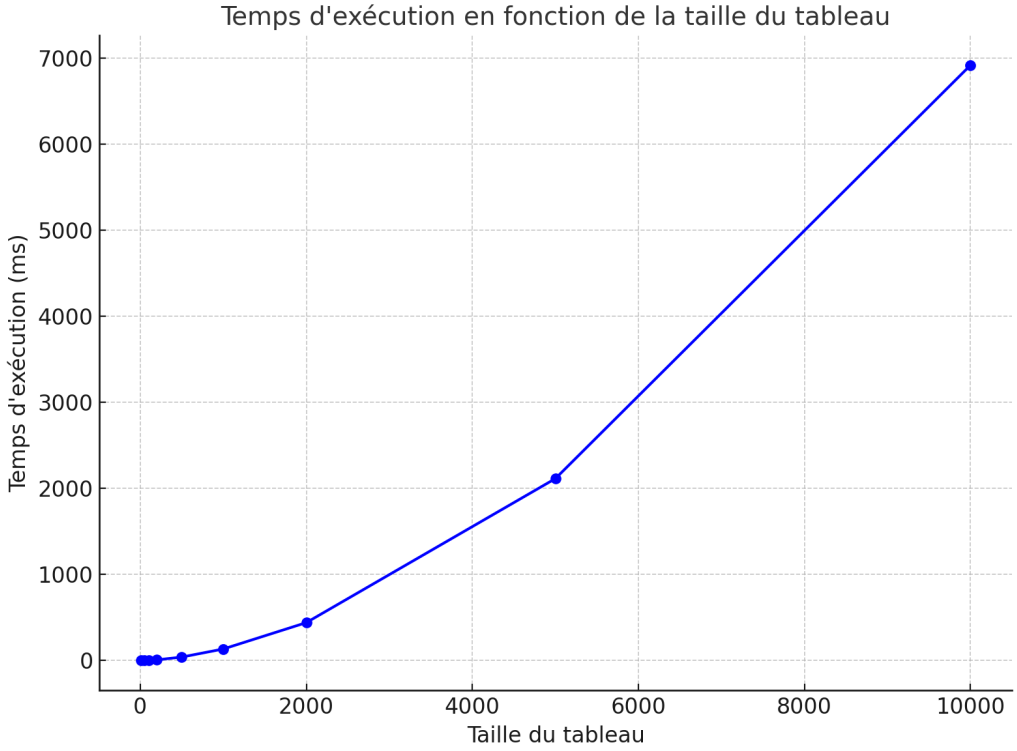

Figure 1: Temps d'exécution en fonction de la taille du tableau

Le graphique obtenu montre une croissance exponentielle correspondant à la complexité $O(n^{1.71})$, confirmant l'analyse théorique.

2 Exercice 2: Sudoku

Le Sudoku est un jeu de placement des chiffres de 1 à 9 dans une grille de 9 lignes et 9 colonnes, également découpée en 9 zones de taille 3×3. Les chiffres de 1 à 9 doivent être placés de façon à ce que chaque ligne, chaque colonne et chaque zone contienne tous les chiffres de 1 à 9.

2.1 Exemple d'instance négative du problème de décision du Sudoku

Une instance négative du problème de décision du Sudoku correspond à une grille initialement partiellement remplie qui ne peut pas être complétée selon les règles du jeu.

Par exemple, la grille suivante ne peut pas être complétée correctement:

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

Si l'on essaie d'ajouter un **5** dans la case (8,8), il y aura un conflit dans la colonne 9, car le chiffre 5 sera présent deux fois dans cette colonne. Cela constitue un exemple d'instance négative du problème, car il n'existe aucune solution valide pour compléter cette grille.

2.2 Complexité asymptotique de la résolution d'un Sudoku 9 × 9

La résolution du Sudoku est un problème de satisfaction de contraintes. Le backtracking, qui explore toutes les configurations possibles, est une des méthodes classiques pour résoudre ce problème.

Dans le pire des cas, chaque cellule vide de la grille (81 au total) peut prendre l'une des 9 valeurs possibles, menant à un espace de recherche de taille 9^{81}, ce qui donne une complexité exponentielle.

Ainsi, la complexité asymptotique d'un algorithme de résolution du Sudoku dans le pire des cas est de:

$$O(9^{81})$$

Bien que cette complexité soit exponentielle, des techniques d'optimisation comme la propagation de contraintes peuvent être utilisées pour réduire le temps de calcul en pratique.

2.3 Modélisation d'un Sudoku de taille $n^2 \times n^2$

Pour modéliser un Sudoku généralisé de taille $n^2 \times n^2$ en tant que formule propositionnelle, nous utilisons des variables binaires $X_{i,j,k}$, où :

$$X_{i,j,k} = 1 \text{ si le symbole } k \text{ est placé dans la case } (i,j),$$

avec $1 \leq i, j \leq n^2$ et $1 \leq k \leq n^2$.

Contraintes

1. Chaque case contient exactement un symbole:

$$\bigvee_{k=1}^{n^2} X_{i,j,k}$$

$$\forall k \neq l, \quad \neg X_{i,j,k} \vee \neg X_{i,j,l}$$

2. Chaque ligne contient chaque symbole exactement une fois:

$$\bigvee_{j=1}^{n^2} X_{i,j,k}$$

$$\forall j \neq l, \quad \neg X_{i,j,k} \vee \neg X_{i,l,k}$$

3. Chaque colonne contient chaque symbole exactement une fois:

$$\bigvee_{i=1}^{n^2} X_{i,j,k}$$

$$\forall i \neq l, \quad \neg X_{i,j,k} \vee \neg X_{l,j,k}$$

4. Chaque région $n \times n$ contient chaque symbole exactement une fois:

$$\bigvee_{(i,j)\in \text{zone}} X_{i,j,k}$$

$$\forall (i_1, j_1) \neq (i_2, j_2), \quad \neg X_{i_1,j_1,k} \vee \neg X_{i_2,j_2,k}$$

Le nombre de variables est n^6 et le nombre de clauses croît en fonction de n.

2.4 Question 4: Introduction

Cette partie décrit un programme qui traduit une grille de Sudoku en un ensemble de clauses SAT (satisfaisabilité propositionnelle) et écrit ces clauses dans un fichier au format CNF (DIMACS). Le solveur SAT peut ensuite être utilisé pour résoudre la grille.

2.4.1 Structure du Programme

Le programme se structure autour de trois étapes principales :

1. **Lecture et parsing** d'une chaîne de caractères représentant un Sudoku.
2. **Génération des clauses SAT** représentant les règles du Sudoku.
3. **Écriture du fichier CNF** contenant les clauses SAT.

2.4.2 Conversion de la Grille Sudoku

La fonction `parseSudokuString` prend une chaîne de 81 caractères où chaque chiffre (1-9) représente un nombre dans la grille, et chaque point (`.`) représente une case vide. Elle convertit cette chaîne en un tableau 2D 9×9.

2.4.3 Initialisation des Littéraux

Chaque cellule (i, j) de la grille et chaque valeur $k \in \{1, 2, ..., 9\}$ est associée à un littéral unique via la fonction `initValeurCorrespondant`. Un littéral représente une variable booléenne dans le système SAT, indiquant si une valeur est présente dans une cellule donnée.

2.4.4 Génération des Clauses

Les règles du Sudoku sont traduites en contraintes logiques, à l'aide des fonctions suivantes :

- `manipulationUneParLigne` : garantit qu'un chiffre apparaît au moins une fois par ligne.
- `manipulationUneParColonne` : assure qu'un chiffre apparaît au moins une fois par colonne.
- `manipulationNumeroExistant` : fixe les chiffres déjà présents dans la grille.

- `manipulationAuMoinUnNuméro` : assure qu'au moins un chiffre est assigné à chaque cellule.
- `manipulationAuMaxUnNuméro` : interdit d'avoir plusieurs chiffres dans une même cellule.
- `manipulationDesRegions` : garantit qu'un chiffre n'apparaît qu'une seule fois dans chaque sous-région 3×3.

2.4.5 Clauses et Littéraux

Chaque contrainte est exprimée sous forme de clauses SAT à l'aide de littéraux positifs et négatifs :

- Un **littéral positif** indique qu'un chiffre est présent dans une cellule donnée.
- Un **littéral négatif** indique qu'un chiffre est absent.

La méthode `getLitteral` renvoie le littéral associé à une position (i, j) et un chiffre k.

2.4.6 Écriture du Fichier CNF

Les clauses générées sont écrites dans un fichier au format DIMACS par la méthode `ecriture`. Ce format standard est compris par la majorité des solveurs SAT.

2.4.7 Résolution par MiniSat

Après la génération du fichier CNF, le solveur **MiniSat** est appelé via la méthode `appelerMiniSat`. Celle-ci exécute MiniSat sur le fichier CNF généré et interprète les résultats :

- Si la sortie est `SAT`, le solveur a trouvé une solution et les littéraux assignés sont lus pour reconstruire la grille résolue.
- Si la sortie est `UNSAT`, il n'existe pas de solution valide pour la grille donnée.

2.4.8 Mesure de Performance

La méthode `mesurerPerformance` permet de résoudre plusieurs grilles Sudoku à partir de fichiers de benchmarks. Elle limite le temps d'exécution à 1 seconde pour évaluer combien de grilles peuvent être résolues dans cette période. Les résultats sont affichés pour chaque fichier de benchmark testé.

- Mesure de performance pour `benchmark_sudoku_1.txt` :
  - Nombre de grilles résolues en 1 seconde : 14
- Mesure de performance pour `benchmark_sudoku_2.txt` :
  - Nombre de grilles résolues en 1 seconde : 12

2.4.9 Exécution Principale

La fonction `main` orchestre le processus complet :

- Conversion de la chaîne de caractères en tableau 2D.
- Affichage de la grille initiale.
- Génération des clauses SAT.
- Écriture des clauses dans un fichier CNF.
- Appel du solveur SAT (MiniSat) pour résoudre le Sudoku.
- Affichage de la grille résolue ou d'un message indiquant qu'il n'y a pas de solution.
- Résultats des mesures de performances pour les benchmark

2.4.10 Conclusion

Le programme traduit un Sudoku en un problème SAT en respectant les contraintes logiques du jeu. Le fichier CNF généré peut ensuite être résolu par un solveur SAT pour obtenir une solution valide. La performance de ce processus est mesurée en fonction du nombre de grilles résolues par le solveur dans un temps limité. Ce processus permet ainsi de valider l'efficacité de la méthode sur des exemples pratiques.

3 Algorithmes de Quasi-Stabilité

3.1 RandomGraphGenerator

La classe `RandomGraphGenerator` est utilisée pour générer des graphes aléatoires, avec ou sans sous-ensemble quasi-stable. Elle propose deux méthodes principales pour la génération de graphes.

- **Méthode** `generateRandomGraph(int n, double p)` **:**
  - Cette méthode génère un graphe aléatoire avec n sommets.
  - La probabilité p détermine la création d'une arête entre chaque paire de sommets. Si un nombre aléatoire généré est inférieur à p, une arête est ajoutée.
- **Méthode** `generateRandomGraphWithQuasiStable(int n, double p, int k)` **:**
  - Cette méthode génère un graphe aléatoire similaire à la méthode précédente, mais avec un sous-ensemble de k sommets quasi-stables.
  - Un ensemble quasi-stable est un sous-ensemble de k sommets dans lequel il y a au plus une arête.
  - La méthode sélectionne aléatoirement k sommets, puis vérifie qu'au plus une arête existe entre eux. Si une arête existe, elle est supprimée pour garantir la quasi-stabilité.

3.2 QuasiStableChecker

Fonctionnement : La classe QuasiStableChecker contient une méthode statique appelée `isQuasiStable` qui vérifie si un sous-ensemble de sommets X est un ensemble quasi-stable dans un graphe G. Un sous-ensemble quasi-stable est un ensemble de sommets où il y a au plus une arête entre chaque paire de sommets.

Voici le déroulement du fonctionnement de cette méthode :

Paramètres d'entrée :

- **Graph** G : Le graphe non orienté dans lequel la vérification sera effectuée.
- **Set<Integer>** X : Le sous-ensemble de sommets (X) à vérifier.

Vérification des arêtes :
Pour chaque paire de sommets distincts (u, v) appartenant à l'ensemble (X), la méthode vérifie si une arête existe entre eux dans le graphe.
Si une arête existe et qu'il y a déjà eu une autre arête rencontrée entre les sommets dans l'ensemble, la méthode retourne `false` car la contrainte de quasi-stabilité (au plus une arête entre les sommets) est violée.
Si la condition est respectée (au plus une arête est trouvée), la méthode continue de vérifier les autres sommets.

Retour de la méthode :
Si, après avoir vérifié toutes les paires de sommets, aucune violation de la contrainte de quasi-stabilité n'est trouvée, la méthode retourne `true`.

Analyse de la complexité :

- Complexité temporelle : La méthode vérifie toutes les paires de sommets dans l'ensemble X. Le nombre de paires est de l'ordre de $O(|X|^2)$, où $(|X|)$ est la taille du sous-ensemble.
- Pour chaque paire, l'existence d'une arête est vérifiée, ce qui se fait en temps constant dans un graphe représenté par des listes d'adjacence.
- La complexité globale de cette méthode est donc $O(|X|^2)$, avec $(|X| \leq n)$, où n est le nombre total de sommets dans le graphe.

Cette méthode est utilisée dans divers algorithmes (comme MaximalQuasiStable ou QuasiStableCompromise) pour vérifier si un ensemble donné de sommets est quasi-stable. Elle permet de garantir que la structure de l'ensemble respecte bien la contrainte de quasi-stabilité dans le graphe.
Conclusion : La méthode 'isQuasiStable' est essentielle pour valider que les sous-ensembles de sommets respectent la contrainte de quasi-stabilité dans les algorithmes qui cherchent à maximiser ou améliorer ces ensembles dans un graphe.

3.3 MaximalQuasiStable

Fonctionnement : L'algorithme MaximalQuasiStable construit un ensemble quasi-stable maximal. Il parcourt tous les sommets du graphe, tente de les ajouter à l'ensemble quasi-stable en cours, puis vérifie si cela viole la contrainte de quasi-stabilité (au plus une arête entre chaque

paire de sommets). Si l'ajout d'un sommet viole la contrainte, il est retiré de l'ensemble quasi-stable.

Analyse de Complexité :

- Boucle externe : L'algorithme parcourt chaque sommet du graphe n fois.
- Pour chaque sommet, il vérifie si l'ensemble quasi-stable actuel est valide, ce qui prend $O(|X|^2)$, où $|X|$ est la taille de l'ensemble quasi-stable.
- La complexité globale est donc $O(n^3)$ dans le pire des cas, où n est le nombre de sommets.

Étude expérimentale :
En utilisant le générateur de graphe aléatoire (G, n, p), nous pouvons générer des graphes avec différents nombres de sommets et probabilités d'arêtes pour tester l'efficacité de MaximalQuasiStable.
Le graphe quasi-stable aléatoire permet de tester la robustesse de l'algorithme lorsqu'un quasi-stable est déjà présent dans le graphe.

3.4 MaximumQuasiStable

Fonctionnement : MaximumQuasiStable est une méthode exhaustive qui explore toutes les combinaisons possibles de sommets pour trouver l'ensemble quasi-stable le plus grand. Il utilise une approche de type backtracking pour construire des ensembles, vérifier leur quasi-stabilité, et sauvegarder l'ensemble de plus grande taille.

Analyse de Complexité :

- Cet algorithme explore toutes les combinaisons possibles de sommets, soit 2^n ensembles.
- Chaque ensemble est vérifié pour voir s'il est quasi-stable, ce qui prend $O(n^2)$.
- La complexité globale est donc $O(2^n \cdot n^2)$, ce qui est exponentiel et devient impraticable pour des graphes de grande taille.

Étude expérimentale :
Pour des petits graphes générés aléatoirement $G(n, p)$ ou avec un quasi-stable aléatoire $G(n, p, k)$, cet algorithme peut être testé pour confirmer qu'il trouve bien l'ensemble quasi-stable maximum, mais son coût computationnel élevé le rend peu efficace pour des graphes avec un grand nombre de sommets.

3.5 QuasiStableCompromise

Fonctionnement : L'algorithme QuasiStableCompromise combine la rapidité de l'algorithme MaximalQuasiStable avec une amélioration locale. Il commence par trouver un ensemble quasi-stable maximal, puis tente d'améliorer cet ensemble en échangeant des sommets (en retirant un sommet de l'ensemble et en ajoutant un nouveau sommet). Si l'ensemble amélioré est quasi-stable et plus grand, il est sauvegardé.

Analyse de Complexité :

- La première phase, MaximalQuasiStable, a une complexité de $O(n^3)$.

- La deuxième phase (amélioration locale) peut parcourir tous les sommets, en effectuant des vérifications similaires à chaque échange de sommet.
- La complexité totale reste dans un cadre polynomial $O(n^4)$ dans le pire des cas, mais est souvent plus rapide en pratique.

Étude expérimentale :
Sur des graphes aléatoires, QuasiStableCompromise peut être testé pour vérifier si son ensemble quasi-stable est plus grand que celui trouvé par MaximalQuasiStable.
Le graphe quasi-stable aléatoire permet de tester la capacité de cet algorithme à améliorer des solutions sous-optimales. Il peut trouver des ensembles quasi-stables de plus grande taille tout en restant plus rapide que MaximumQuasiStable.

Conclusion des expériences :

- **MaximalQuasiStable** est rapide et trouve une solution raisonnable, mais pas forcément optimale.
- **MaximumQuasiStable** garantit de trouver l'ensemble quasi-stable maximum, mais il est inapplicable pour des graphes de grande taille en raison de sa complexité exponentielle.
- **QuasiStableCompromise** est un compromis intéressant, car il peut trouver des solutions de meilleure qualité que MaximalQuasiStable tout en restant dans un cadre de complexité polynomial, ce qui en fait une solution viable pour des graphes de taille moyenne à grande.

3.6 Réduction SAT pour la Quasi-Stabilité

Pour formuler un problème de quasi-stabilité en termes de logique propositionnelle, nous devons transformer les contraintes du graphe G et de la taille k en une formule booléenne, de manière à ce que cette formule soit satisfiable si et seulement si le graphe G possède un ensemble quasi-stable de taille k.

Étapes pour construire la formule $\phi_{G,k}$:

Variables propositionnelles :

Variables de sélection des sommets :

- Soit x_i une variable propositionnelle qui vaut vrai si le sommet i appartient à l'ensemble quasi-stable, et faux sinon.
- On aura donc n variables propositionnelles, une pour chaque sommet du graphe G, où n est le nombre total de sommets dans G.

Contraintes de la formule :

Contrainte de taille k (exactement k sommets dans l'ensemble quasi-stable) :

- On impose qu'il y ait exactement k sommets dans l'ensemble quasi-stable. Cette contrainte peut être exprimée en forçant que la somme des variables x_i (qui représentent les sommets) soit exactement égale à k.
- Cette contrainte se formule par des clauses où l'on impose que précisément k sommets soient sélectionnés.

Contrainte de quasi-stabilité (au plus une arête entre les sommets de l'ensemble) :

- Pour chaque paire de sommets (i, j), s'il existe une arête entre eux (i.e., $\{i, j\} \in E$), alors au plus un des deux sommets peut être dans l'ensemble quasi-stable. Autrement dit, pour chaque arête $\{i, j\} \in E$, on impose que x_i et x_j ne peuvent pas être simultanément vrais.
- Cette contrainte est représentée par la clause $\neg x_i \vee \neg x_j$, ce qui signifie que x_i et x_j ne peuvent pas être tous les deux vrais s'ils sont connectés par une arête.

Formule finale $\phi_{G,k}$:

La formule $\phi_{G,k}$ est donc la conjonction des deux contraintes suivantes :

1. Somme des variables égale à k : Exactement k sommets sont dans l'ensemble quasi-stable.
    - Cela peut être exprimé sous forme de clauses qui imposent la sélection de k sommets parmi n.
2. Quasi-stabilité : Pour chaque arête $\{i, j\} \in E$, la clause $\neg x_i \vee \neg x_j$ garantit qu'il n'y ait pas plus d'une arête entre les sommets sélectionnés.

Ainsi, **Formule finale** $\phi_{G,k}$ est :

$$\phi_{G,k} = \left(\sum_{i=1}^{n} x_i = k \right) \wedge \bigwedge_{\{i,j\} \in E} (\neg x_i \vee \neg x_j) \tag{1}$$

Explication :

1. **Première partie :** Nous avons besoin d'imposer que exactement k sommets appartiennent à l'ensemble quasi-stable. Cela peut être fait en combinant des clauses qui sélectionnent exactement k sommets.
2. **Deuxième partie :** Pour garantir la quasi-stabilité, nous interdisons la présence de plus d'une arête dans l'ensemble en imposant que pour chaque paire de sommets connectés par une arête, au moins un des sommets ne soit pas dans l'ensemble.

3.7 Description de la classe SatReduction

La classe `SatReduction` a pour but de générer des clauses de logique propositionnelle (SAT) à partir d'un graphe donné et d'un nombre spécifique de sommets k. Ces clauses représentent les contraintes du problème de quasi-stabilité dans un format que MiniSat peut résoudre. Le but est de formuler les contraintes du problème sous forme de clauses SAT, qui seront ensuite résolues par un solveur SAT.

3.7.1 Méthodes de la classe :

1. `generateSatClauses(Graph G, int k)`

Fonction : Cette méthode génère une liste de clauses SAT pour un graphe G avec la contrainte que l'ensemble quasi-stable contiendra exactement k sommets.

Étapes principales :

- **Contrainte 1 (Sélection de k sommets) :** Cette contrainte s'assure que l'ensemble quasi-stable final contient exactement k sommets. Elle est générée par la méthode `generateExactlyKVertices`.
- **Contrainte 2 (Quasi-stabilité) :** Cette contrainte garantit que l'ensemble des sommets sélectionnés respecte la propriété de quasi-stabilité. Pour chaque paire de sommets connectés par une arête dans le graphe, elle ajoute une clause qui empêche les deux sommets d'être présents simultanément dans l'ensemble quasi-stable.

Retour : Elle renvoie une liste de chaînes de caractères, où chaque chaîne est une clause SAT exprimée en termes de variables booléennes représentant les sommets.

2. `generateExactlyKVertices(Graph G, int k)`

Fonction : Cette méthode génère la contrainte SAT pour s'assurer que l'ensemble quasi-stable contiendra exactement k sommets. Chaque sommet du graphe est représenté par une variable propositionnelle, et cette méthode impose que la somme de ces variables soit égale à k.

Détail de la contrainte :

- Chaque sommet est représenté par une variable de la forme x_i, où i représente l'indice du sommet.
- La méthode crée une clause qui exprime que la somme des variables associées aux sommets est égale à k, ce qui signifie que précisément k sommets doivent être choisis dans l'ensemble quasi-stable.

Exemple de sortie : Pour un graphe avec 5 sommets et $k = 3$, la clause générée pourrait ressembler à :

$$x_0 + x_1 + x_2 + x_3 + x_4 = 3$$

Cela signifie que l'ensemble des variables x_0 à x_4 doit totaliser exactement 3 sommets sélectionnés.

3. `generateQuasiStabilityClause(int u, int v)`

Fonction : Cette méthode génère une clause SAT pour exprimer la contrainte de quasi-stabilité pour une paire de sommets u et v. Si une arête existe entre ces deux sommets, alors la clause impose que les deux sommets ne peuvent pas être sélectionnés simultanément dans l'ensemble quasi-stable.

Clause générée :

- La clause est exprimée sous la forme $\neg x_u \vee \neg x_v$, ce qui signifie que si le sommet u est dans l'ensemble quasi-stable (c'est-à-dire x_u est vrai), alors v ne peut pas l'être (c'est-à-dire x_v doit être faux), et inversement.

Exemple de sortie : Si une arête existe entre les sommets 1 et 2, la clause générée serait:

$$\neg x_1 \vee \neg x_2$$

Cela signifie que les sommets 1 et 2 ne peuvent pas être simultanément sélectionnés dans l'ensemble quasi-stable.

Conclusion :

La classe `SatReduction` convertit le problème de la quasi-stabilité dans un graphe en un ensemble de clauses SAT, qui peut ensuite être résolu par un solveur SAT comme MiniSat. Elle génère deux types de contraintes :

1. La contrainte d'exactement k sommets dans l'ensemble quasi-stable.
2. La contrainte de quasi-stabilité, qui s'assure qu'il n'y ait pas plus d'une arête entre les sommets sélectionnés.

Cela permet de résoudre le problème de quasi-stabilité via la réduction SAT en utilisant un solveur de satisfaction booléenne (ici MiniSat).