ULB UNIVERSITÉ LIBRE DE BRUXELLES

# Projet Algorithmique 2 (INFO-F203) : Fiat Lux

Hac Le

May 6, 2024

ULB UNIVERSITÉ LIBRE DE BRUXELLES

# 1 Introduction

Les solutions documentées dans ce rapport sont dédiées à résoudre le problème de l'éclairage des ampoules. Dans ce problème nous sommes donnés une instance d'ampoules et d'interupteurs interconnectés. Chaque interrupteur peut être soit ouverte soit fermée, et chaque ampoule ne s'allume seulement si les constraintes sur les 2 interrupteurs sont satisfaites. La phase 1 consiste à répondre si toutes les ampoules peuvent être allumées simultanément, tandis que la phase 2 consiste à chercher le nombre maximum d'ampoules qu'on peut allumer pour une instance donnée.

Nous commencerons par l'idée principale, à ce que le problème se traduit, et la conception et l'implémentation en Java des solutions proposées à partir de sources trouvées sur le Web (voir sources et citations).

Toutes les algorithmes utilisés sont implémentés avec la librairie Java JGraphT.

# 2 Idée principale

La première et seconde phase sont analogues à un 2-satisfiability problem (2-SAT) et un maximum-2-satisfiability (MAX-2-SAT). 2-SAT est un *"problème informatique consistant à attribuer des valeurs à des variables, dont chacune a deux valeurs possibles, afin de satisfaire un système de contraintes sur des paires de variables.""Les instances du problème de la 2-satisfiabilité sont généralement exprimées sous forme de formules booléennes d'un type particulier, appelé forme normale conjonctive. (2-CNF)."* Wikipedia

En effet, les instances de notre problème sont fournies sous la forme d'un fichier .txt, chaque ligne représentant les contraintes d'une ampoule pour pouvoir être allumée. L'idée est de traduire ces contraintes en implications pour ensuite construire un graphe d'implication sur lequel nous pouvons appliquer nos algorithmes. Ce graphe d'implication exprime *"les variables d'une instance et leurs négations comme des sommets d'un graphe, et les contraintes sur les paires de variables comme des arcs dirigées. "* Wikipedia Nous en prennerons avantage pour déterminer si toutes les ampoules peuvent être simultanément allumées et, si ce n'est pas le cas, pour la deuxième phase, le nombre maximum d'ampoules qui peuvent être allumées.

# 3 Traduction de contraintes et graphe d'implication

Les formules 2-CNF sont *"une conjonction (une opération booléenne AND) de clauses, où chaque clause est une disjonction (une opération booléenne OR) ."* Wikipedia

Il existe 16 combinaisons différentes pour les 4 bits indiquant les contraintes d'une ampoule. ($2^4$= 16). Chaque ampoule n'est affectée que par 2 interrupteurs et peut donc être convertie en clauses à 2 variables puis en implications.

Après avoir discuté avec d'autres étudiants, nous avons pu trouver toutes les implications suivantes:

| | | | |
|---|---|---|---|
| 1 0 0 0 | $\neg X \rightarrow X,\ \neg Y \rightarrow Y$ | 1 1 1 0 | $\neg X \rightarrow Y,\ \neg Y \rightarrow X$ |
| 0 1 0 0 | $\neg X \rightarrow X,\ Y \rightarrow \neg Y$ | 1 1 0 1 | $\neg X \rightarrow \neg Y,\ Y \rightarrow X$ |
| 0 0 1 0 | $X \rightarrow \neg X,\ \neg Y \rightarrow Y$ | 1 0 1 1 | $\neg Y \rightarrow \neg X,\ X \rightarrow Y$ |
| 0 0 0 1 | $X \rightarrow \neg X,\ Y \rightarrow \neg Y$ | 0 1 1 1 | $X \rightarrow \neg Y,\ Y \rightarrow \neg X$ |
| 1 1 0 0 | $X \rightarrow X$ | 0 1 1 0 | $\neg X \rightarrow Y,\ \neg Y \rightarrow X$ |
| 0 0 1 1 | $\neg X \rightarrow X$ | 1 0 0 1 | $\neg Y \rightarrow \neg X,\ X \rightarrow Y$ |
| 1 0 1 0 | $Y \rightarrow Y$ | | |
| 0 1 0 1 | $\neg Y \rightarrow Y$ | | |

Les variables X et Y sont respectivement des interrupteurs de ligne et des interrupteurs de colonne.

Nous pouvons maintenant créer notre graphe d'implication, appelons-le G.

### 3.1 Implémentation

Voici les fichiers nécessaires à la création du graphe d'implication :

- Constraint.java
- FileArrayProvider.java
- ImplicationGraph.java
- Lightbulb.java

La classe Constraint stocke tous les différents types de contraintes (combinaison de bits) sous forme de Strings.

FileArrayProvider est une classe copié de ce lien qui lit le fichier .txt dans un tableau de Strings.

Dans la classe ImplicationGraph, les Strings du tableau sont utilisées pour ajouter des sommets et des arcs au graphe. Chaque interrupteur possède deux sommets représentant ses deux états possibles, ouvert et fermé, true et false (par exemple, X et ∼X). Pour chaque ligne représentant une ampoule, en fonction du type de contrainte, des arcs dirigées seront ajoutées aux sommets correspondants conformément aux implications mentionnées précédemment.

La classe Lightbulb possède 2 attributs représentant les interrupteurs qui l'affectent et un attribut indiquant la contrainte qu'elle possède.

## 4 Phase 1

### 4.1 Idée

Maintenant que nos contraintes sont exprimées sous la forme d'un graphe d'implication, nous pouvons utiliser le travail existant d'Aspvall, Plass et Tarjan (lien) à notre avantage.

La phase 1 est un problème 2-SAT dont la réponse est simplement true ou false à la question de savoir si nous pouvons ou non allumer toutes les ampoules en même temps. Pour trouver la réponse, nous allons utiliser la notion de composantes fortement connectées (SCC) et l'algorithme de Kosaraju.

Condition : *"Comme l'ont montré Aspvall et al., il s'agit d'une condition nécessaire et suffisante : une formule 2-CNF est satisfaisable si et seulement si aucune variable n'appartient à la même composante fortement connectée que sa négation."* Wikipedia

Ainsi, en appliquant cette condition, il suffit d'exécuter l'algorithme de Kosaraju pour trouver tous les SCCs dans notre graphe d'implication, puis de vérifier pour chaque variable qu'elle ne réside pas dans le même SCC que sa négation. Si c'est le cas, nous pouvons arrêter la recherche et retourner false (l'instance n'est pas satisfaisable, ce qui signifie que toutes les ampoules ne peuvent pas être allumées en même temps). Si ce n'est pas le cas, l'instance est satisfaisable.

Pour ce faire, nous allons utiliser les structures de données et méthodes suivantes :

- Graphe d'Implication (avec JGraphT)
- HashMap
- Stack
- Depth-First Search (pour Kosaraju)

## 4.2 Implémentation

J'aimerais souligner que les problèmes 2-SAT sont très populaires et que de nombreuses implémentations de cette solution sont facilement accessibles en ligne. En fait, le code final de notre solution est un amalgame de plusieurs solveurs de 2-SAT tirés directement de sources trouvées en ligne ici. Le défi a été de découvrir en premier lieu qu'il s'agissait d'un 2-SAT, ainsi que d'adapter ces solveurs existants à notre problème spécifique (notamment à cause de la façon dont les contraintes sont encodées, les dépendances, puisque toutes les combinaisons de conditions ne permettent pas de fixer un état défini pour un interrupteur).

Pour résoudre la phase 1, SolverPhase1.java et les fichiers susmentionnés sont utilisés.

L'implémentation de l'algorithme de Kosaraju et la recherche de la présence d'une variable et de sa négation dans les SCCs sont copiées d'ici et d'ici, et est ensuite adapté pour fonctionner avec nos structures de données choisies (classe Graph de JGraphT, hashmaps) et optimisé.

### 4.2.1 Trouver tous les SCCs

L'algorithme de Kosaraju consiste à exécuter deux Depth-First Searches l'un après l'autre.

Le premier DFS stocke les sommets de G dans un Stack dans un postordre inverse. Le second DFS traverse le graphe transposé de G, $G^T$, pour trouver tous les SCCs. Cela se fait en simplement utiliser la méthode *predecessorListOf()* de JGraphT. 2 hashmaps différents sont utilisés pour les DFS pour stocker les somemts déjà visités dans les traversées normale et inversée. Un troisième hashmaps stocke quels sommets appartiennent à quelle CSC.

### 4.2.2 Recherche dans chaque SCC

The méthode *solve()* initialise les hashmaps, exécute le premier DFS, et, tant que le Stack remplie par le premier DFS n'est pas vide, exécute le second DFS sur tous les éléments du Stack, tout en incrémentant le numéro de la SCC courante. Enfin, pour chaque sommet (variable) de G, voir s'il est présent dans la même SCC que sa négation et renvoyer la réponse conformément.

## 4.3 Complexité

Soit V le nombre de sommets et E le nombre d'arcs dirigés de G.

*"L'algorithme de Kosaraju effectue deux traversées complètes du graphe en temps theta(V + E)."* Wikipedia, vu qu'il a 2 DFS (O(V + E)). Vérifier le SCC de chaque sommet se fait en O(V). Le solveur de la phase 1 a donc une complexité temporelle de O(V + E) (linéaire en le nombre de sommets et d'arcs).

# 5 Phase 2

## 5.1 Idée

La phase 2 est un problème MAX-2-SAT dont l'objectif est de trouver le nombre maximum d'ampoules qui peuvent être allumées. Ce problème est analogue au nombre maximum de clauses qui peuvent être satisfaites dans notre formule booléenne 2-CNF.

Pour ce faire, nous avons eu recours à une solution simple de backtracking. Par pure force brute, notre algorithme teste toutes les configurations possibles pour trouver celle qui présente le maximum de contraintes satisfaisantes. La vitesse est sacrifiée pour la certitude absolue que la solution trouvée est la meilleure.

## 5.2 Implémentation

Pour la Phase 2, SolverPhase2.java et les fichiers précédents sont utilisés.

La méthode *find()* exécute un backtracking sur un hashmap qui stocke l'état de chaque interrupteur (true ou false) et cela pour chaque configuration possible.

La méthode *countCurrentMaxLights()* compte le nombre maximum d'ampoules de la configuration courante. Pour ce faire, un loop itère sur chaque ampoule de la classe Lightbulb susmentionnée, vérifie si ses contraintes sont satisfaites et incrémente un compteur si c'est le cas. Le nombre max est retourné à la fin du backtrack.

### 5.3 Complexité

La complexité temporelle du backtracking est de $O(2^V)$ où V est le nombre de sommets de G. Lorsque le backtracking a trouvé une configuration possible, on doit itérer sur chaque ampoule afin de vérifier lesquelles peuvent être allumées. Ceci se fait en O(E). Donc notre algorithme est en $O(E * 2^V)$.

### 5.4 Idées d'optimisations possibles

Suite à une discussion avec d'autres étudiants, une optimisation possible de la phase 2 serait de prendre avantage du fait que chaque ampoule n'est affectée seulement par 2 interrupteurs. Et chaque interrupteur n'affecte qu'un nombre limité d'ampoules. L'idée serait de traiter les ampoules en groupes séparés (les ampoules dont les contraintes affectent les uns les autres, ce qui se manifeste peut être par la connexité dans le graphe d'implication) et d'additionner le nombre max. d'ampoules pour chaque groupe d'ampoules.

## 6 Sources et citations

Toutes les citations de Wikipedia sont traduites de l'anglais vers le français.

2-SAT : https://en.wikipedia.org/wiki/2-satisfiability

FileArrayProvider class : https://stackoverflow.com/questions/285712/java-reading-a-file-into-an-array

2-SAT solver : https://github.com/xiaoyuetang/AlgorithmTwoSAT/tree/master

Kosaraju's algorithm : https://www.topcoder.com/thrive/articles/kosarajus-algorithm-for-strongly-connected-components