RAPPORT DU PROJET DE PROGRAMMATION

"Solver of Logipix"

January 2022

ED-DIB Abdessalam
LABIAD Ismail

ÉCOLE POLYTECHNIQUE

IP PARIS

TABLE DES MATIÈRES

1
RAPPEL DU PROBLÈME

L'objectif de ce projet est de créer un solveur des puzzles **Logipix**, comme celui ci représenté dans l'image ci-dessous. Le but d'un puzzle **Logipix** est de découvrir une image cachée grâce à des indices numériques. Chaque puzzle consiste en une grille contenant des cellules vides ou des cellules avec des chiffres. Afin de résoudre un tel puzzle, on doit relier les paires de chiffres (à part les 1), par des chemins dont la longueur est égale à la valeur des indices des extrémités.

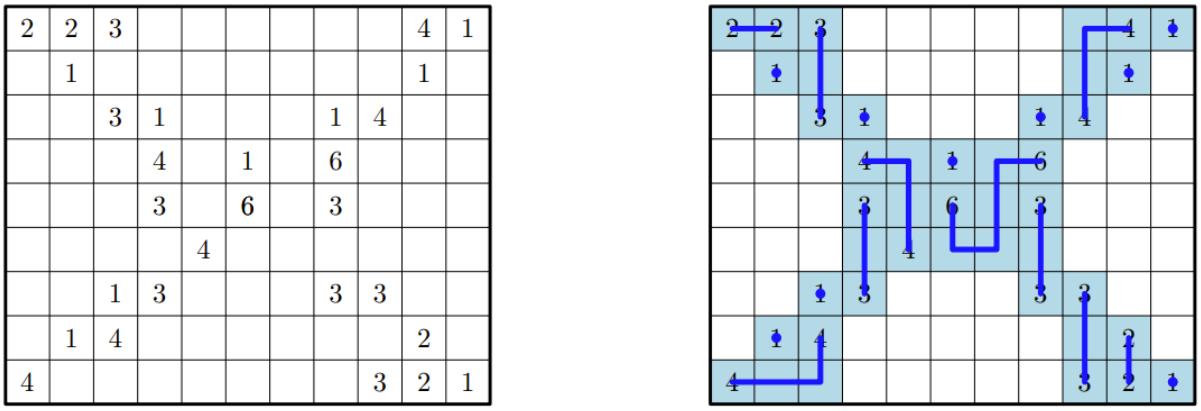

Figure 1 – Exemple d'un puzzle "logipix".

2
MODÉLISATION DU PROBLÈME

Pour modéliser notre problème, nous avons utilisé une grille composé de cellules, chacune de ces cellules comporte une valeur numérique correspondant à un indice. Pour ce faire, nous avons créé les classes suivantes :

2.1 Cell

Cette classe définit la structure d'une cellule et les actions qui vont avec. Chaque instance de cette classe est définie par ses coordonnées **"x"** (colonne) et **"y"** (ligne), sa valeur qui correspond à un indice **"value"**, une liste chainée contenant les **broken lines** allant de cette cellule **"listBrokenLines"**, un booléen précisant l'état de cette cellule **"occupied"** [1] et finalement la cellule à laquelle elle est connectée **"connectedCell"**.

On trouve également les méthodes suivantes :
- Un constructeur **Cell(int v, int x, int y)**
- Deux méthodes **connectToCell(Cell c)** et **disconnectFromCell(Cell c)** permettent de connecter et déconnecter deux cellules
- Deux méthodes **distance(Cell c)** et **distance(int x0, int y0)** permettant de calculer la distance d'une cellule à une autre

2.2 BrokenLine

Cette classe nous permet de définir la structure d'une "broken line". Chaque instance de celle-ci est définie par deux cellules **first** et **last** qui correspondent à à ses extrémités, et un tableau de cellules **line** qui correspond au chemin reliant **first** et **last**.

Cette classe nous fournit les méthodes ci-après :
- Un constructeur **BrokenLine(Cell f, int length, Cell l)**
- Une méthode **copyAndAddCell(Cell toAdd, int addIndex)** qui permet de générer une nouvelle broken line à partir de celle donnée en échangeant la cellule à l'indice **addIndex** par la cellule **toAdd**
- Une méthode **contains(Cell cell)** qui vérifie si la cellule **cell** fait partie de la broken line

1. Si $occupied == true$, cela signifie que la cellule peut être occupée, sinon elle correspond à une case vide.

2.3 Logipix

Celle-ci constitue la classe principale de notre programme. Elle nous permet de modéliser un puzzle logipix. Une instance de cette classe, et donc un puzzle logipix est défini par une grille **cellPuzzle** représentant le jeu, les dimensions de cette grilles **n_cols** et **n_line**, une table de hachage qui nous permet de grouper les cellules par leurs valeurs **mapCellValues** et finalement la liste des indices présent dans le puzzle **intToConsider**.

On retrouve les méthodes suivantes :

- Une méthode **loadFromTxt(String path)** qui nous permet d'initialiser un puzzle logipix
- Une méthode **addBrokenLines(Cell c)** qui permet de générer la liste des brokenlines possible allant de la cellule c
- Trois méthodes permettant de résoudre le puzzle en se basant sur la technique du **back-tracking** qu'on explicitera dans ce qui suit.

2.4 DisplayLogipix

Cette classe nous permet de visualiser un puzzle logipix en utilisant **java.awt** et **javax.swing**. Celle-ci est composé d'un constructeur **DisplayLogipix(Logipix logipix)**, une méthode pour afficher le puzzle **display()** et deux méthodes permettant de dessiner les brokenlines et peindre les cellules occupées.

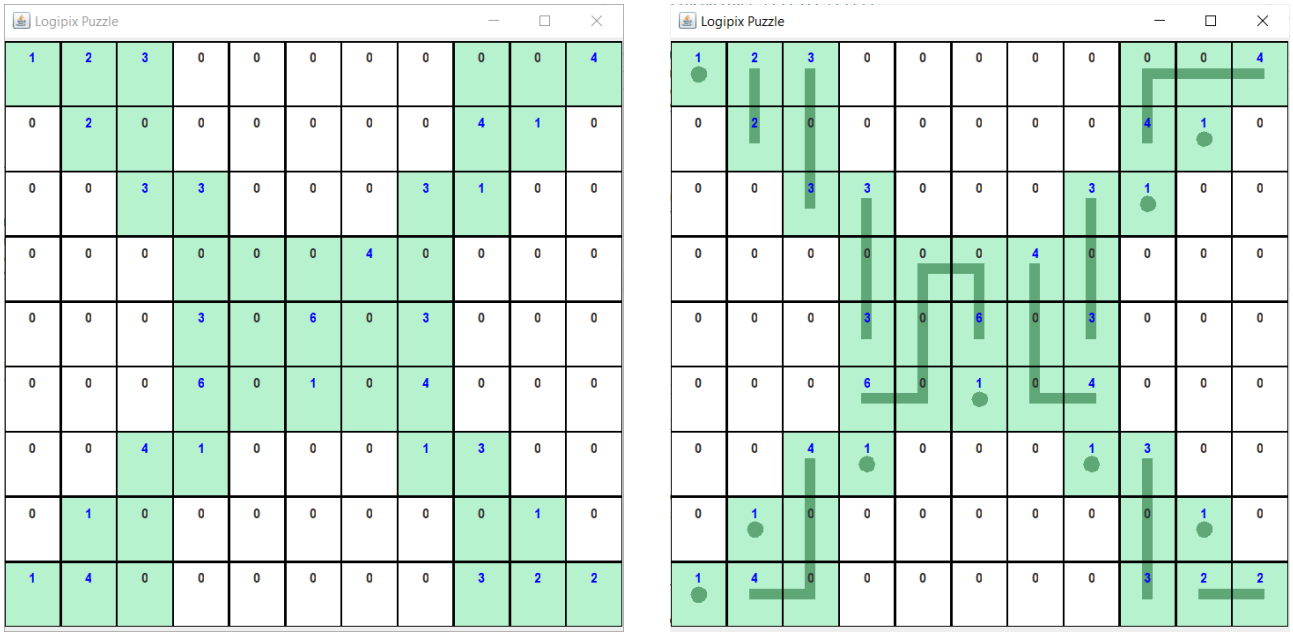

Figure 2 – Affichage d'une puzzle Logipix.

3
ALGORITHMES EMPLOYÉS

3.1 Recherche des "broken lines"

On commence tout d'abord par vérifier si la cellule correspond à un indice égale à 1, dans ce cas il n'y a rien à chercher. Sinon, on cherche les candidats possibles ,avec lesquels elle peut se connecter, dans la liste de hachage regroupant les cellules ayant les mêmes valeurs et ce en testant l'égalité entre la distance qui sépare chaque candidat de la cellule et la valeur de celle-ci. Ensuite, on parcourt tous les candidats possibles et on ajoute une broken line temporaire à la liste des broken lines correspondante à la cellule **"listBrokenLines"**. Partant d'une cellule d'une brokenLine temporaire, on regarde les cellules voisines, si c'est possible d'ajouter un seul voisin, on l'ajoute à cette broken line, si c'est possible d'ajouter plusieurs voisins, on recopie cette broken line et à chaque fois on lui ajoute le bon voisin, sinon si ce n'est pas possible d'ajouter un voisin, on supprime cette broken line de notre liste.

3.2 Back-tracking

3.2.1 • Première approche : Brute Force

La première méthode que nous allons utiliser pour résoudre un puzzle logipix repose sur du "brute force" : **back-tracking classique**. L'idée est la suivante : On prend les indices "clues" à parcourir dans l'ordre croissant. Pour chaque indice, nous générons l'ensemble de toutes les broken lines possibles. Nous commençons par le premier indice et choisissons une des possibilités. Puis nous passons au deuxième indice, faisons de même, tout en nous assurant que celle-ci est compatible avec le choix fait précédemment et ainsi de suite. Si, à un moment donné, nous n'avons plus de possibilités, nous changeons l'un des choix que nous avons fait auparavant, et on continue. Cela dit, si on arrive à parcourir tous les indices en trouvant les broken lines qui leurs correspondent, le puzzle a une solution et on renvoie true sinon on renvoie false.

3.2.2 • Deuxième approche : Combinaison Exclusion

Nous considérons maintenant des combinaisons de solutions partielles possibles. Nous commençons par générer toutes les "broken lines" possibles correspondant à un indice donné. Si l'intersection de toutes ces lignes est non vide, alors nous pouvons conclure que les cases appartenant à l'intersection appartiennent à la solution. Après avoir identifié avec succès certaines cellules faisant partie de la solution, certaines lignes, qui , avant, été des broken lines "possibles" ne sont plus compatibles avec le nouvel état actuel. De ce fait, on peut appliquer le principe d'exclusion et donc supprimer ces lignes avant d'effectuer le back-tracking.

Cela dit, en implémentant cette méthode, nous avions l'idée d'un algorithme reposant sur le même principe d'exclusion et donnant des résultats plus rapide en comparaison avec le programme implémentant la méthode

précédente. L'idée est la suivante, avant de commencer le back-tracking, on parcourt toutes les cellules puis on recherche celles ayant une seule broken line. Après les avoir trouvées on marque leur trajet pour les exclure lors des recherches effectuées lors du back-tracking. Concernant, le parcours des indices : on les parcourt ligne par ligne de gauche à droite.

3.2.3 • Troisième approche : Méthode probabiliste

Pour cette méthode, on a utilisé le même code du back-tracking qu'on a implémenté précédemment. Par contre, on a changé la méthode addBrokenLines, de telle façon, à ce que chaque brokenLine a une probabilité qui correspond à produit(distance de la case à la cellule / nombre de steps restants) afin de pouvoir classer les broken lines en commenceant par celles qui, le plus vraisemblable possible, feront partie de la solution. C'est en quelaue sorte, commme si on a utilisé une file de priorité dont la priorité est donnée par la quantité précédente à savoir : produit(distance de la case à la cellule / nombre de steps restants).

4
COMPLEXITÉ ET ÉVALUATION

4.1 Évaluation

On liste ci-après le temps d'exécution de chacun des algorithmes précédents :

Method	logipix1	Man	TeaCup	Immensite
BackTracking	3ms	147ms	225ms	58ms
BackTrackinOrdered	3ms	24ms	140ms	14ms
BackTrackingExclusion	2ms	23ms	28ms	9ms
Exclution & Prob	2ms	49ms	11ms	11ms

Table 1 – Temps d'exécution des algorithmes de résolution

4.2 Complexité

On donne ci-après des majorations des complexités des programmes utilisés :

— **addBrokenLines** $pC * v * 4 * 3^{(v-3)}$ On boucle sur le nombre de candidat possibles **pC**, puis on boucle sur le nombre de steps qui est égale à la valeur de la cellule **v** - 1, et pour chaque step, on avance tous les broken lines d'un pas qu'on peut majorer par $4 * 3^{(v-3)}$ approximativement. En général, le nombre de candidats possibles est égale à 1 ou 2, et le nombre des brokenLines qu'on fait avancer à chaque "step" est généralement petit de l'ordre de 3.

— **backTracking** Au pire des cas, tous les algorithmes du backtracking doivent essayer tous les cas possibles ce qui correspond aux produit de la longueur des broken lines de chaque valeur "clue", et donc le produit des complexités de addBrokenLines de chaque valeur.