Dauphine | PSL UNIVERSITÉ PARIS | MIDO MATHEMATIQUES ET INFORMATIQUE DE LA DECISION ET DES ORGANISATIONS

Rapport Projet Phine Loops

Projet Java avancé & Intelligence Artificielle

M1 Miage classique 2019/2020

Version 2.1 – 29/12/2019
Git groupe : DLK
Groupe : Duc-Chinh PHAM – Léa ONG – Taoufiq KOUNAIDI

Responsable de Java : Benjamin NEGREVERGNE

Responsable d'Intelligence Artificielle : Julien LESCA

Sommaire

1. Description du projet

Le projet est basé autour du jeu de puzzle *Infinity Loop*.

Ci-dessous les différents programmes attendus du projet et l'état de nos avancements :

Générateur de niveau	COMPLET
Générateur de niveau avec le nombre de composantes connexes	NON EFFECTUÉ
Vérificateur de solution	COMPLET
Solveur de niveau	COMPLET (Méthode exhaustive et CSP)
Solveur de niveau avec multithreading	NON EFFECTUÉ
Interface graphique	PARTIEL (Possibilité de jouer)

2. Implémentation du projet

2.1. Architecture Modèle – Vue – Contrôleur

Nous avons structuré notre projet selon l'architecture MVC afin de séparer et faciliter la partie back-end (*model*) de la partie front-end (*view* et *controller*).

- Partie Model:

Dans cette partie nous avons défini un niveau comme étant une grille (*class Grid*) constituée de différentes pièces *(class Piece*). Les pièces pouvant être de différents types et d'orientations, nous avons donc utilisé l'héritage afin que chaque pièce hérite de la classe mère Piece. Une pièce est aussi définie par ses côtés connectables ou non. Pour faciliter le générateur de niveaux et le vérificateur, nous avons défini une liste d'orientations (North, South, West et East) modifiable selon l'orientation de la pièce. La grille aura les principales méthodes du générateur, du vérificateur et du solveur de niveaux.

- Partie View:

Afin de visualiser un niveau ou une solution nous avons donc réalisé la partie View du projet. Pour représenter un niveau et notamment les différentes pièces, nous avons créé des classes qui serviront uniquement à représenter les pièces (*class [nompiece]Drawer*) dans le même principe que la partie Model (héritage des pièces...).

- Partie Controller:

Afin que l'utilisateur puisse modifier le modèle : rotation des pièces en temps réel et résolution automatique du niveau, nous avons finalement ajouté la partie Controller avec la classe *MyListener* permettant de gérer les pièces et une *classe MyActionListener* permettant de gérer l'action du solveur.

Le diagramme de classes UML du projet résume les relations entre nos différentes classes :

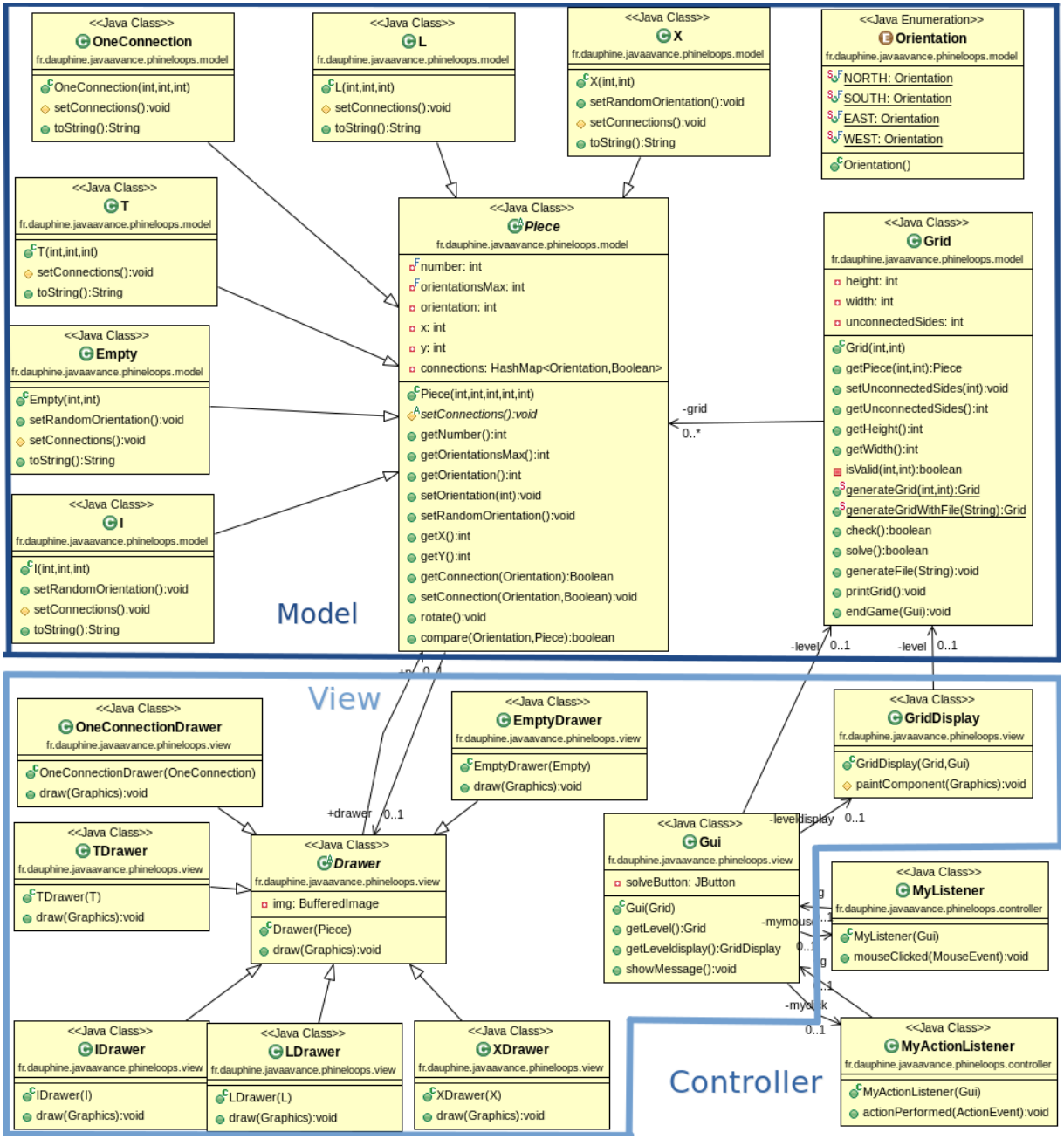

Figure 1 : Diagramme de classes - Architecture MVC

2.2. Organisation et répartition du travail

L'objectif de notre organisation était de répartir les tâches de la manière la plus pertinente et équitable possible, dont voici la répartition de départ :

- Duc : Générateur de niveaux, vérificateur de solutions
- Léa : Solveur de niveaux
- Taoufiq : Interface graphique

Mais au fur et à mesure du projet, on s'est rendu compte que chacun de nous avait son propre rythme d'avancement sur les tâches confiées, c'est pour cette raison qu'on a décidé d'adopter une organisation basé sur le travail collaboratif où nous avons pu mettre en avant nos compétences en commun, favorisant l'échange et la communication au sein du trinôme.

2.3. Générateur de niveaux

Le générateur de niveaux est basé sur la méthode proposée dans l'énoncé du projet, qui consiste à construire une solution puis à mélanger les positions initiales. On part ainsi du coin en haut à gauche de la grille que l'on parcourra de gauche à droite et de haut en bas, puis on définit une pièce au hasard, tout en prenant en compte les bords et les voisins de gauche et du haut dont les conditions seront vérifiées grâce à la méthode *isValid()*. Lorsque la solution a été générée, la grille sera de nouveau parcourue pour changer leur orientation de manière aléatoire.

Malheureusement, notre générateur ne prend pas en compte le nombre de composantes connexes, aucune solution viable n'a été trouvée pour ce problème (quelle(s) pièce(s) favoriser, une possible implantation d'un arbre ou d'une liste afin d'obtenir plusieurs graphes connexes, etc.).

2.4. Vérificateur de solution

Le vérificateur de niveaux reprend la même démarche que le générateur de niveaux: on parcourt la grille et on vérifie si chaque pièce est compatible avec les bords et les voisins de gauche et du haut grâce à la méthode *isValid()*. Si une pièce est invalide, le parcours de la grille est stoppé, le niveau n'est donc pas résolu et le vérificateur retourne *false*.

2.5. Partie Intelligence Artificielle : solveur de niveau

Méthode de résolution quasi exhaustive par la méthode de résolution CSP

Nous avons résolu la grille grâce à l'implémentation de la méthode solve() dans la classe Grid selon une méthode vue en cours d'Intelligence Artificielle. Cette méthode consiste à résoudre un problème de satisfaction de contraintes (CSP). Notre modèle est défini par la grille contenant des pièces. Chaque pièce aura un numéro d'orientation : une variable qui aura une valeur satisfaisant toutes les contraintes de la pièce, c'est-à-dire que chaque côté de la pièce soit connectable ou non avec ses pièces voisines selon son orientation et des orientations des pièces voisines.

C'est une manière quasi exhaustive et simple permettant de tester toutes les positions de chaque pièce et vérifier s'il y a une solution. Afin d'implémenter la structure d'un CSP nous avons utilisé la librairie "Choco Solver" vue en TP de IA. Nous avons donc défini la structure de cette façon :

- **L'ensemble de variables :** Les pièces.
- **L'ensemble des domaines pour chaque variable :** Pour chaque pièce, l'orientation de la pièce orientationP[i][j] prenant une valeur entre 0 et le nombre de rotations possibles selon le type de la pièce.
- **L'ensemble des contraintes qui précisent les combinaisons admissibles pour la valeur de la variable :** Selon l'orientation de la pièce, ses points cardinaux doivent être admissible avec ceux de ses voisins.

  Exemple : si la pièce est de type une connexion nord, son voisin du nord doit avoir une connexion sud reliable et les autres voisins ne doivent pas avoir de cardinal reliable.

  Il est nécessaire que l'affectation des valeurs soit cohérente pour ne pas violer toute autre contrainte. Nous avons donc défini une contrainte pour que chaque contrainte des points cardinaux de la pièce puisse avoir la même valeur assignée, c'est-à-dire la même orientation.

Le Choco Solver résout lui-même le problème selon le modèle établi par la méthode *getSolver().solve()* renvoyant un *boolean* si la résolution est possible ou non.

Afin de mieux visualiser et tester le programme, nous avons utilisé l'interface graphique ou des caractères Unicode dans la console (en utilisant la méthode *printGrid()* dans la classe *Grid*). Nous avons fait les tests avec les fichiers tests disponibles dans le répertoire *instances*, lorsque *true* était renvoyée avec les fichiers 'messup.true', nous savions qu'une/des erreur(s) étai(en)t présente(s) dans le code. Nous pouvions également tester avec notre générateur de niveaux.

Cependant cette implémentation a été longue à coder car il fallait vérifier tous les cas. La première version avait des erreurs car on s'y perdait avec les différents cas et il fallait se familiariser avec la librairie. Mais au fur et à mesure nous avons remarqué que certains cas se ressemblaient et nous avons donc implémenté au mieux afin d'éviter les répétitions et les duplications de code.

De plus cette méthode semble jouer grandement sur le temps de l'algorithme car elle va vérifier pour chaque orientation de chaque pièce. Les tests benchmarks nous montre aussi un temps de calcul long pour résoudre un niveau. En effet notre modèle repose sur le test pour chaque orientation donc même lorsqu'une grille est pratiquement résolue, le modèle recalcule depuis le début, ce qui peut prendre beaucoup de temps notamment avec une longue grille.

Au niveau de l'amélioration algorithmique nous avons éliminé le test des pièces vides et certains tests des pièces en bord de plateau afin d'éviter de faire des vérifications inutiles. L'amélioration via multi-threading n'as pas pu être implémentée car nous ne savons pas comment l'implémenter avec le Choco Solver.

2.6. Interface graphique

La conception de notre interface finale a dû passer par 3 améliorations :

- **Amélioration 1 :** Visualisation statique d'un niveau - (voir figure 2)

Pour une meilleure visualisation de notre grille, nous avons utilisé des images pour mieux illustrer nos pièces. Chaque image est affectée à chaque pièce selon son type, pour au final former un niveau statique non résolu . La première image ci-dessous représente une grille générée.

- **Amélioration 2 :** Faire participer l'utilisateur à trouver la solution

Une fois le résultat précédent obtenu, nous avons décidé d'améliorer notre interface en donnant la possibilité à l'utilisateur de jouer en cliquant sur les pièces afin de les faire tourner (rotation de 90°), jusqu'à trouver la bonne solution. Une fenêtre apparaît lorsque la grille est résolue par l'utilisateur (cf: figure 3).

- **Amélioration 3 :** Résoudre le niveau à l'aide d'un bouton SOLVE - (voir figure 4)

De plus, nous avons pensé à donner le choix à l'utilisateur de résoudre la grille automatiquement, en cliquant sur le bouton "SOLVE". Un message apparaîtra alors pour d'indiquer si la grille est solvable ou non.

- **Amélioration 4** (non implémentée) **:** Visualiser les étapes du solveur

Nous n'avons pas implémenté la visualisation du solveur en train de travailler; c'est à dire pouvoir voir les rotations des pièces en direct. Pour le faire, il était un peu plus compliqué de récupérer les étapes du solveur du Choco Solver dont la librairie était longue à assimiler. Mais sans l'utilisation de celui-ci, nous aurions certainement pu le faire.

Pendant notre implémentation de l'interface, l'une des principales difficultés rencontrées consistait à afficher l'interface pendant l'exécution du projet en ligne de commande. Pour le résoudre, nous avons évalué la structure du package 'view', ensuite modifié la classe *Main* fournie par le professeur en ajoutant une nouvelle option de visualisation avec interface.

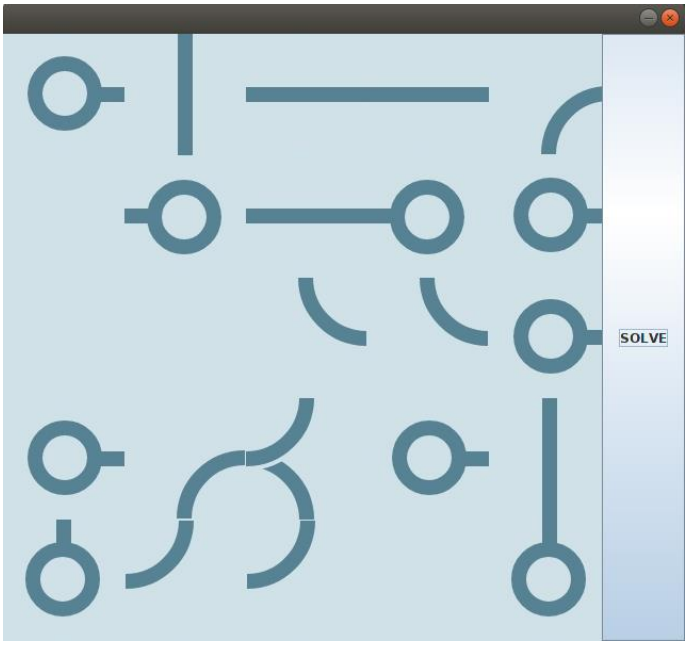

Figure 2 : Visualisation Statique d'un niveau – Bouton 'solve' du niveau

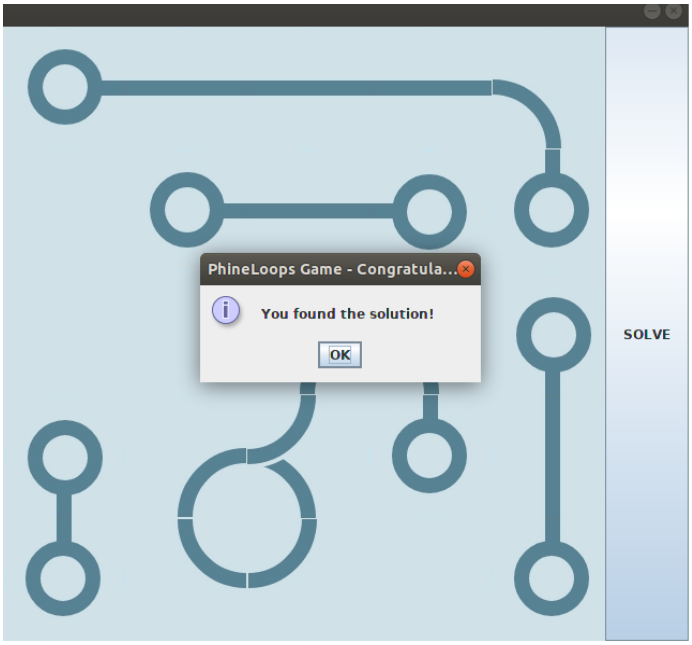

Figure 3 : Niveau résolu par l'utilisateur et affichage du message après résolution

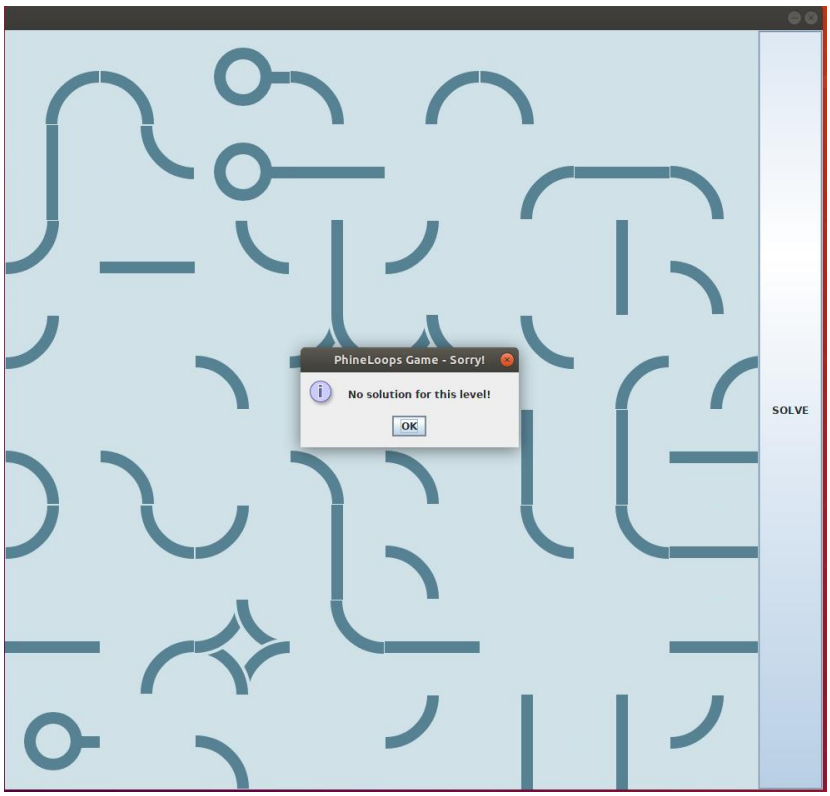

Figure 4 : Résoudre en utilisant le bouton SOLVE

Message "Pas de solution pour ce niveau"

3. Conclusion

Ce projet a été très intéressant car cela a été pour chaque membre du groupe une nouvelle expérience acquise dans le domaine du développement logiciel. Nous avons dû nous organiser plus distinctement et malgré cela nous avons rencontré quelques problèmes tout au long de l'élaboration du jeu.

Tous les objectifs principaux de notre programme ont été atteints, nous disposons d'un générateur, solveur et d'un vérificateur - tous fonctionnels - ainsi qu'une interface graphique pour l'affichage de la grille. Cependant, il reste possible d'ajouter diverses fonctionnalités telles que la visualisation du solveur étape par étape, ou encore la possibilité de générer des niveaux avec un nombre de composantes connexes défini.

Enfin, nous pensons avoir réussi à donner une implémentation au projet conforme au minimum requis, bien que la difficulté de ce dernier fût plus relevée que ceux que nous avons rencontré tout au long de notre cursus universitaire.