Equipe Alt-Loop :

KORMANN Liam
LECLERC Thomas
SCHWITTHAL Alexandre

Rapport de Projet : Java et IA

Projet Phineloops

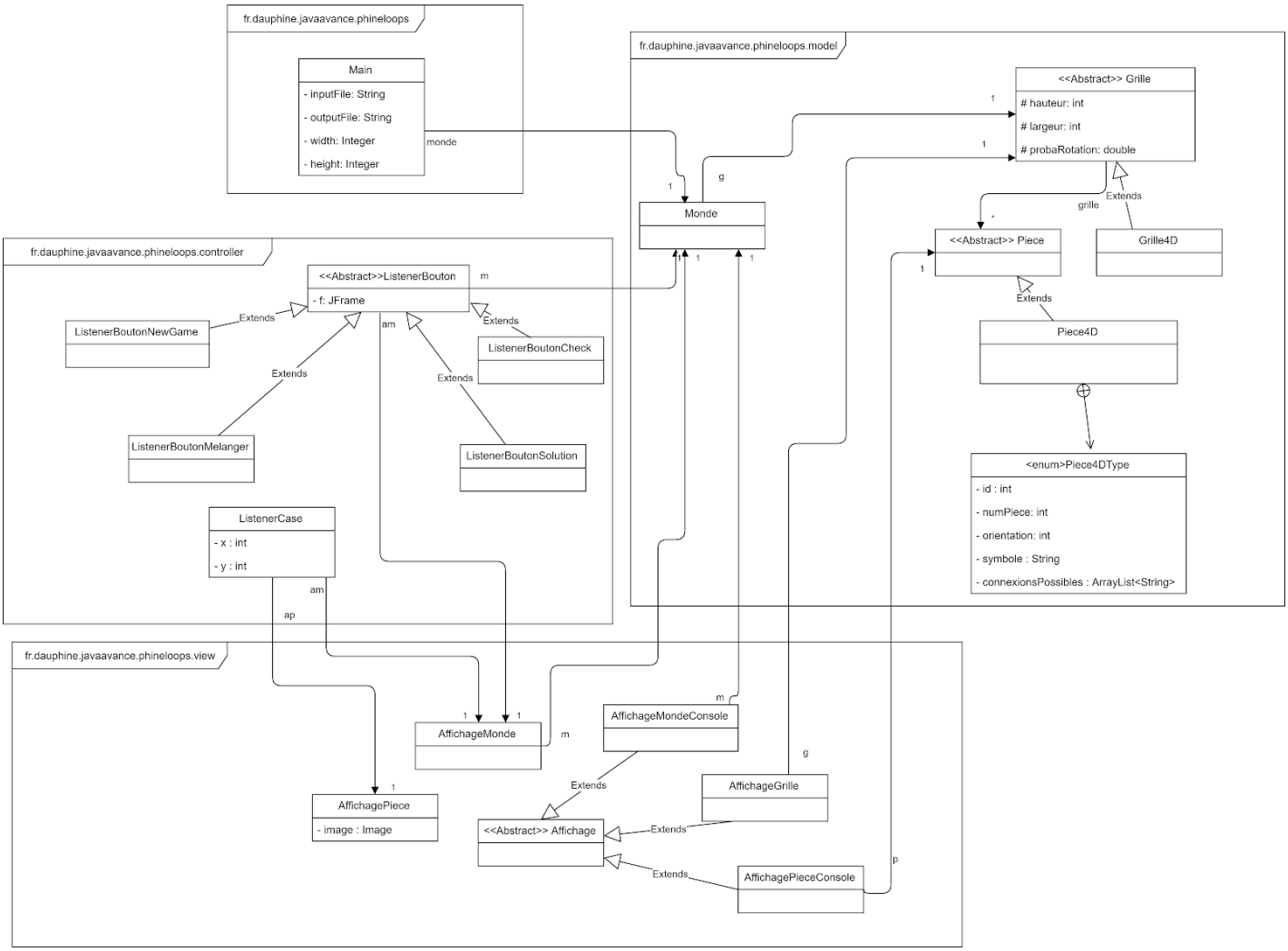
SOMMAIRE

I. Description de l'implémentation

Au début de nos recherches autour du projet, notre priorité a été de trouver une implémentation adéquate pour les pièces d'une grille. Ainsi, notre version de PhineLoops s'est initialement articulée autour d'une classe Grille et d'un type énuméré Piece, ce dernier disposant des informations relatives aux pièces décrites pour une grille établie sur les quatre points cardinaux.

Toutefois, cette première piste d'implémentation a engendré divers problèmes puisque le principe de généricité n'était pas respecté et que la manipulation d'un type énuméré n'était pas toujours pratique. Pour régler ces premiers problèmes, nous avons créé des classes abstraites Grille et Piece devant uniformiser le comportement de tous types de grilles ou de pièces que l'on voudrait éventuellement manipuler. Les anciennes implémentations pour la grille et les pièces ont été transférées respectivement dans la classe Grille4D et la classe Piece4D, cette dernière contenant un type énuméré interne contenant les informations des pièces.

Pour faciliter la compréhension de notre architecture, voici le diagramme de classes de notre projet :



Une fois nos premières classes implémentées, nous avons choisi d'établir une architecture MVC (Modèle-Vue-Contrôleur) au sein de notre projet afin d'une part, de mieux comprendre

les rôles propres de nos classes et, d'autre part, de fixer des périmètres stricts à chacune d'elles. Ainsi, les classes du package Model ne dépendent aucunement des classes des autres packages et se contentent de leur transmettre des informations utiles pour leurs traitements respectifs.

Nous avons implémenté une classe Monde, pouvant manipuler tout type hypothétique de grille dans ce package, notamment dans la classe Main. Notre classe Monde y est utilisée pour donner des informations sur sa grille en attribut et pour importer une grille à partir d'un fichier texte, ce qui l'affecte en attribut de l'instance de Monde.

Le package View regroupe les classes jouant un rôle dans l'affichage d'informations, que ce soit via notre interface graphique ou bien via la console. Dans le premier cas, on utilise les classes héritant de JPanel, nous avons AffichagePiece qui permet d'afficher une image représentant la pièce. Cette classe est ensuite utilisée dans AffichageMonde, qui affiche l'ensemble du plateau en cours. De la même façon, nous utilisons pour l'affichage console les classes AffichageMondeConsole et AffichagePieceConsole, héritant toutes les deux de la classe abstraite Affichage afin d'uniformiser l'affichage.

Le package Controller, quant à lui contient les classes permettant à l'utilisateur d'interagir avec l'interface graphique, nous avons créé une classe abstraite ListenerBouton qui est héritée par les classes ListenerBoutonNewGame, qui permet de créer une nouvelle grille, ListenerBoutonMelanger qui permet de mélanger la grille actuelle, ListenerBoutonCheck qui permet de vérifier si la grille actuelle est résolue ou non en faisant apparaître un message à l'écran et la classe ListenerBoutonSolution qui lance le solver pour essayer de corriger la grille. Enfin, pour permettre à l'utilisateur de « jouer » grâce à l'interface graphique, la classe ListenerCase gère la rotation d'une pièce lorsque l'on clique dessus.

Nous avons rencontré quelques difficultés au cours de ce projet, par exemple l'implémentation des pièces qui ont trouvé leur implémentation actuelle alors que le projet s'articulait déjà efficacement autour d'un seul type énuméré, mais vu que cette solution s'est révélée limitée pour la suite du projet, il nous a fallu reconsidérer notre stratégie. Plutôt que des difficultés, nous avons réfléchi longuement à l'implémentation de notre solver et de notre interface graphique pour éviter des rétropédalages qui nous auraient beaucoup ralenti.

Nous avons travaillé en équipe pour les décisions importantes et nous avons codé individuellement pour des fonctionnalités mineures. Mais globalement nous pouvons dire que nous avons suivi la répartition suivante : Liam a travaillé sur les fonctions utiles pour la classe Main, comme l'import/export de fichier, Thomas a réalisé l'interface graphique, aussi bien sur son côté visuel que sur son interactivité et Alexandre a principalement travaillé sur le solver du projet.

II. Explications et expérimentations des performances du solver

A. Choix de la modélisation

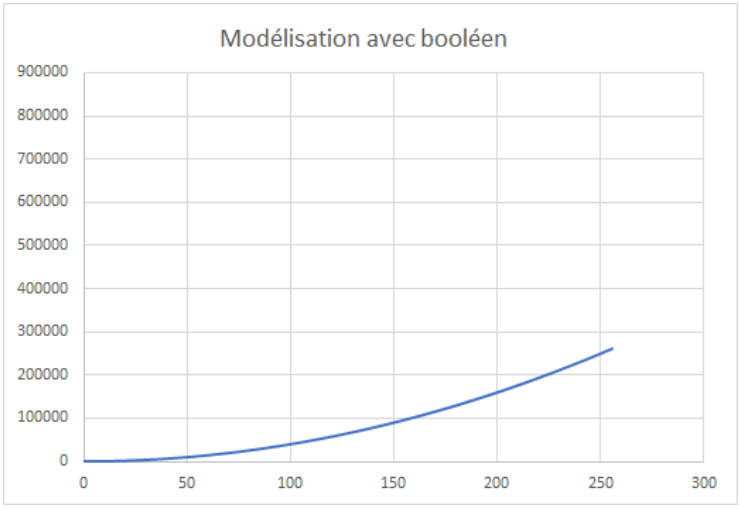
La solution que nous avons choisie pour réaliser la résolution de ce projet fût l'utilisation de Choco Solver, une library dédiée la résolution de CSP (Constraint Satisfaction Problem). L'objectif étant ainsi de décrire les différentes contraintes du problème, sous une certaine forme, propre à Choco Solver.

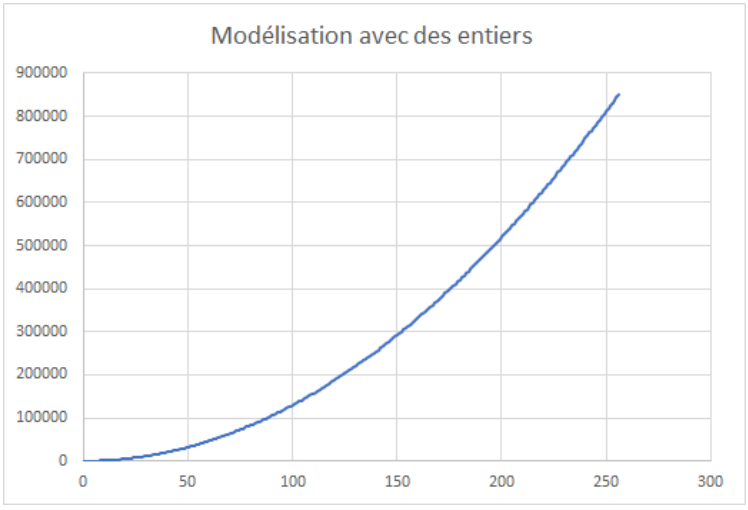
La principale difficulté était ainsi de trouver une modélisation permettant de résoudre le jeu, et de façon efficace.

Au début, nous sommes partis sur une modélisation basée sur des entiers, mais cette modélisation était bien trop lente, notamment car le nombre de possibilités pour des grilles relativement grandes étaient immense, le programme n'arrivait pas à trouver de solution sur des grilles de taille égale ou supérieure à 20x20.
Pour une grille de taille HxW, le nombre total de variables était égal à : (H * W * 6) + (H * W * 6) + (H * W).

Puis, au vu des faibles performances de cette modélisation, nous sommes alors partis sur une modélisation à base de booléens qui a été la modélisation finale utilisée pour le solver. Pour cette modélisation, une grille de taille HxW a un nombre total de variables égal à : H * W * 4

Ce qui, comparativement à la modélisation précédente, réduit grandement le nombre de variables (voir graphique ci-dessous)





Pour ces deux graphiques, l'axe des ordonnées correspond aux nombres de variables requises et celui des abscisses, à la taille d'une grille de taille HxW où H = W. L'on peut ainsi voir un gain relativement important au niveau des variables.

B. Fonctionnement globale de la modélisation

La modélisation s'appuie ainsi sur un tableau en 3 dimensions, de taille H * W * 4, où les deux premières valeurs correspondent tout simplement à la taille de la grille à résoudre, la dernière valeur, étant fixe et de 4, correspond aux 4 côtés d'une pièce, il s'agit de valeurs booléennes dont la valeur est true si la pièce pointe dans cette direction, sinon false.
Ainsi pour une Barre verticale, les valeurs HAUT et BAS sont vraies, mais les valeurs DROITE et GAUCHE sont fausses. Pour une pièce vide, toutes les valeurs sont à false et ainsi de suite.

Chaque pièce dispose ainsi de contraintes précises. De plus, vu que le but est de faire tourner les pièces pour résoudre le problème, les contraintes ne doivent pas être fixes.
Si l'on reprend l'exemple de la barre, elle peut avoir 2 positions possibles, verticale ou horizontale. Ainsi, les contraintes correspondant à cette pièce sont :

- les valeurs des booléens HAUT et BAS sont égales
- les valeurs des booléens DROITE et GAUCHE sont égales
- le nombre de booléens à True est toujours de 2.

L'on pourrait ajouter d'autres contraintes, mais cela créerait de la redondance (car ces contraintes sont obtenues implicitement via transitivité grâce aux 3 ci-dessus) ce qui alourdirait le solveur et le rendrait moins performant.

Les contraintes des bordures sont aussi à gérer, ainsi, si une pièce se trouve sur la toute première ligne, la valeur de son booléen HAUT est fixé à false, car une pièce ne doit pas pointer vers l'extérieur.

Puis évidemment, la cohérence des pièces entre-elles, si une pièce pointe vers la droite, alors la pièce à sa droite doit pointer vers la gauche par exemple.

C. Optimisation des contraintes

En ayant réalisé des tests de performances à l'aide de fonctionnalités temporaires dédiées (générer 100 grilles, les résoudre, puis afficher le temps moyen de résolution par exemple). Nous avons pu remarquer qu'avoir trop de contraintes augmentait le temps nécessaire à la résolution des grilles.
Ainsi, en optimisant les contraintes, nous avons pu passer d'un temps de résolution de 11 secondes en moyenne pour une grille de 256x256 à 8-9 secondes, ce qui est un gain de temps non négligeable.

Certaines contraintes, comme la gestion de cohérences des pièces entres elles ont grandement amélioré les performances du solveur.
En effet, il n'est pas nécessaire pour chaque pièce de vérifier chaque direction, seules 2 directions suffisent : une verticale (HAUT ou BAS) et une horizontale (DROITE ou GAUCHE).

Ici, nous avons arbitrairement choisi de vérifier pour la pièce courante, son HAUT et sa GAUCHE.
Si cette vérification est effectuée sur chaque pièce, et qu'en même temps une vérification est effectuée sur les bordures, alors la vérification BAS et DROITE ne sont pas nécessaires, car si le HAUT d'une pièce A est bien relié avec le BAS d'une pièce B, alors la vérification sur le BAS de la pièce B est effectuée à ce moment-là.

D'autres points, avec l'utilisation de nombreux tests ont permis d'améliorer légèrement l'efficacité du solver. Il est ainsi possible, avec Choco solver, de sommer des booléens (1 si vrai, 0 si faux) et cette fonctionnalité a permis une optimisation des contraintes de certaines pièces, comme la pièce vide et la pièce allant dans les 4 directions. Il est en effet préférable de dire pour la pièce vide que la somme des 4 booléens de direction est égale à 0 plutôt que de dire spécifiquement que les 4 booléens sont faux.

D. Ajout du multithreading

L'un des derniers points pour optimiser le CSP était l'utilisation du Multithreading, qui a permis la résolution de grille 512x512 , qui était jusque-là, impossible avec le timeout de 60 secondes.
Choco Solver dispose de fonctionnalités incluses pour l'utilisation de threads, grâce à un objet Portfolio qui permet de stocker plusieurs Model (chaque Model étant attribué à un thread). Puis, dès que l'un des threads arrivent à résoudre le modèle, alors ils s'arrêtent tous et l'on renvoie le Model résolu.
Les tests pour le Multithreading ont été réalisés via des grilles fixes générées, il est relativement difficile de donner les gains de temps, car cela dépendait des ordinateurs sur lesquels ces tests ont été effectués.
Sur certains, le Multithreading sur une grille 512x512 entraînait une erreur mémoire, sur d'autres en revanche, il y avait un gain de temps considérable (17-18 secondes pour résoudre une 512x512, pour un temps de plus de 60 secondes en monothread).

III. Fonctionnalités apportées et non apportées

A. Générateur de niveau

Nous avons réussi à programmer un générateur de niveaux prenant en paramètres les dimensions (largeur et hauteur) et le nom du fichier de sortie.

Le générateur de niveaux génère une grille déjà valide puis nous avons une méthode pour mélanger aléatoirement la grille ensuite. Les niveaux générés comportent parfois plusieurs composantes connexes.

À la fin de l'exécution, le générateur crée un fichier texte dans le répertoire courant selon le nom spécifié en paramètre de la commande et qui contient une description de la grille. Le fichier texte répond aux demandes d'entrées et sorties du sujet.
Cependant, l'option permettant de générer un niveau en spécifiant le nombre de composantes connexes de la solution n'est pas prise en charge par notre programme.

B. Vérificateur de solution

Nous avons développé un vérificateur de solution qui prend en paramètre un fichier comportant un niveau à tester. Ensuite, le programme retourne sur la sortie standard « SOLVED : true » si la grille est résolue, et « SOLVED : false » si la grille n'est pas résolue.

C. Solveur de niveau

Nous avons programmé un solver de niveaux avec la bibliothèque Java Choco Solver.
Nous avons également ajouté la prise en charge du multithreading pour améliorer la performance de notre programme.

D. Visualisation des niveaux

Nous proposons deux affichages graphiques différents :

- Un affichage en mode console à l'aide de codes Unicode
- Une interface graphique à l'aide de Swing

L'interface graphique permet de visualiser un niveau de manière statique : la grille est affichée, lors du lancement, résolue.

Mais elle offre également la possibilité à l'utilisateur de jouer au jeu. En effet, lorsque le joueur clique sur une pièce, celle-ci effectue une rotation à 90°. De plus, nous avons ajouté quatre boutons :

- « Mélanger » : permet de mélanger la grille pour commencer à jouer
- « Solution » : permet de lancer le solver sur la grille
- « Check » : permet de vérifier si la grille est résolue ou non
- « New Game » : permet de générer une nouvelle grille

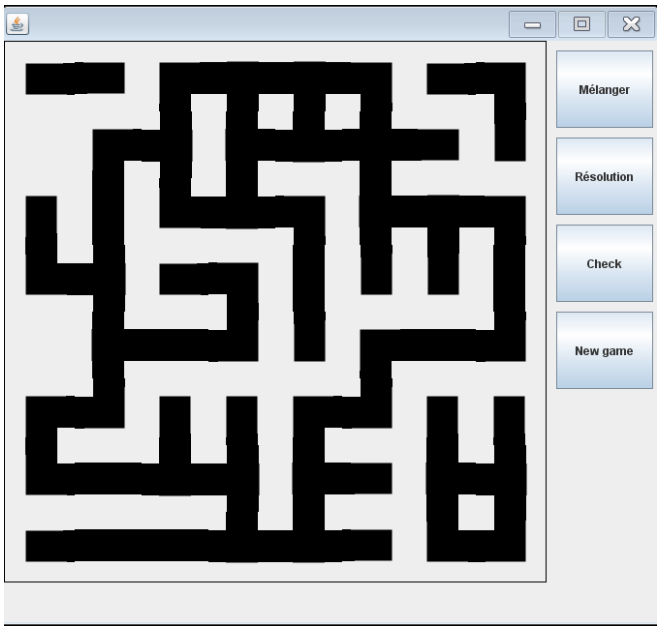


Figure 1 : Affichage lors du lancement de l'interface graphique

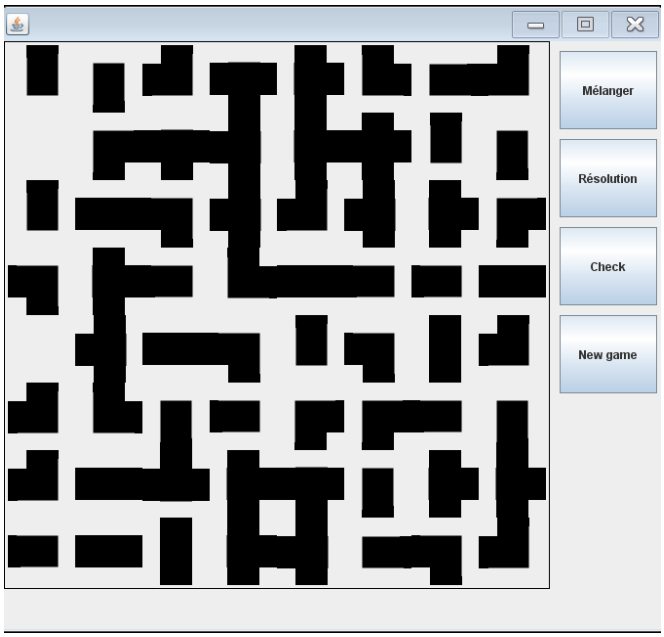
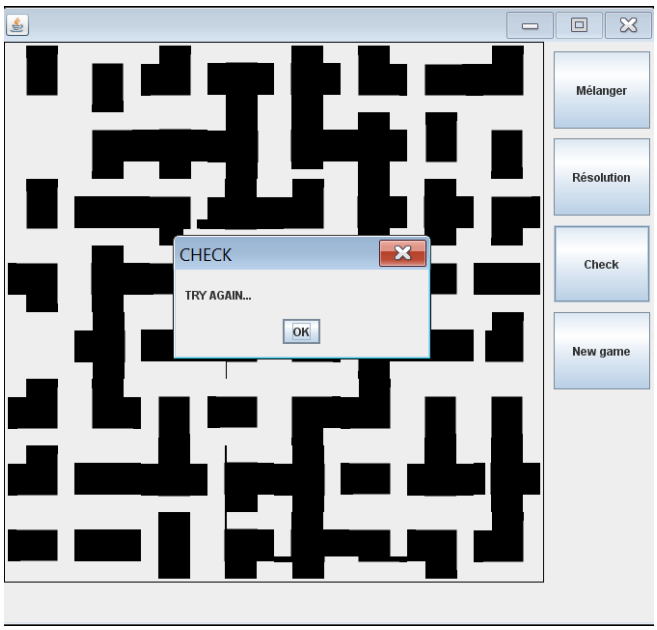


Figure 2 : Affichage suite à l'appui sur le bouton « Mélanger »



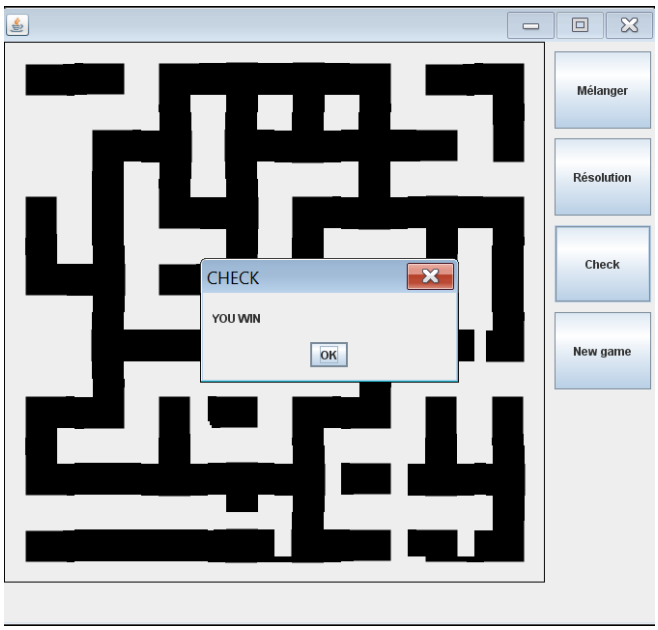


Figure 3 et 4 : Affichage suite à l'appui sur le bouton « Check »

Cependant, nous ne proposons pas l'option qui consiste à pouvoir visualiser le solver travailler, c'est-à-dire voir les rotations des pièces en direct et ainsi voir l'état du niveau par le solver, car celui-ci ne nous permet pas d'extraire les étapes de résolution.

A. La généricité

Nous avons pris en compte la généricité dans notre programme. Par exemple, nous avons utilisé une classe mère « Piece » afin de, si on le souhaite, créer une classe fille « Piece6D » afin de programmer la version HEX du jeu.
Cependant, nous n'avons pas programmé la version HEX du jeu où chaque pièce comporte au maximum 6 connexions au lieu de 4. De plus, nous n'avons pas développé le jeu avec le problème opposé où chaque pièce ne possède aucune connexion avec ses voisins. Enfin nous n'avons pas développé l'option permettant de définir le nombre de composantes connexes souhaité à la génération.