PHINE LOOP

Rapport de projet de Java Avancé et d'Intelligence Artificielle

Par **Lucas DEDIEU**, **Alban TIACOH** et **Thierno Moussa BAH**

Première année de Master MIAGE

Professeurs référents : **BENJAMIN NEGREVERGNE** et **JULIEN LESCA**

1. Introduction

1.1 Rappel du but du projet

Le but de ce projet est d'implémenter le jeu Infinité Loop en Java. Infinity Loop est un jeu de casse-tête. Le joueur dispose d'une grille de pièces avec plus ou moins de connection. Le but du jeu de tourner chaque pièce de telle sorte à ce qu'elle soit correctement connectée avec ses voisins horizontaux et verticaux.

En plus d'implémenter le jeu original, nous devions à minima coder un vérificateur de solution, un générateur aléatoire de grilles, un résolveur de grille et la possibilité de lire et écrire des fichiers texte représentant des grilles.

Nous avions également la possibilité d'implémenter des fonctionnalités optionnelles. Pour notre cas nous avons fait :

- La possibilité de générer un niveau avec un nombre maximum de composantes connexes.
- Une interface graphique en Swing permettant de jouer à un niveau, générer un niveau et visualiser la résolution progressive d'un niveau.
- Trois types de multi-threading pour notre solveur, dont un maître-esclave.
- Quatre algorithmes de résolution de niveau.
- Des tests unitaires.

2. Guide utilisateur

2.1 Utilisation de l'application

En plus de celles décrites dans l'énoncé du sujet, nous avons ajouté des options de lancement via ligne de commande :

- -G *inputfile* (--d *hxw*): lance l'application avec l'interface graphique. On affiche la grille écrite dans le fichier *inputfile.* Si aucun fichier n'est fourni alors il faut ajouter l'option --d et l'argument *hxw* représentant la taille de la grille à générer.
  Dans le cas où l'on précise les deux, seul le fichier est pris en compte.
  Dans le cas où le fichier n'est pas un fichier valide, alors on prend en compte l'argument --d.

- --m <arg> (à utiliser avec –solve) : permet de choisir le type de résolveur pour résoudre la grille. Pour arg on peut avoir les valeurs :
  « line » : lance le résolveur ligne par ligne.
  « csp » : lance le résolveur csp.
  « choco » : lance le résolveur ChocoSolver.
  « snail » : lance le résolveur escargot.

2.2 Utilisation de l'interface utilisateur

L'interface utilisateur permet de :

- Jouer à un niveau : pour cela il suffit de cliquer sur une pièce pour la tourner. Une fois que l'utilisateur positionne bien toute les pièces et que la grille est résolue, une boite de dialogue apparait afin de féliciter le joueur.
- Générer un niveau : pour cela il suffit de cliquer sur « Generate » dans la top-bar. On peut aussi utiliser le raccourci clavier ALT+8.
  Un boite de dialogue apparait pour demander à l'utilisateur les dimensions de la grille qu'il veut générer.
  Il faut impérativement écrire les dimensions comme cela : « wxh » avec w la largeur et h la hauteur.
  Exemple : 5x5 pour générer une grille de 5 pièces par 5.
- Résoudre un niveau : le menu déroulant propose les différents résolveurs. Le résolveur escargot (snail dans l'interface) permet de visualiser en temps réel la résolution du niveau.
  Les raccourcis clavier sont :
  ALT+1 : Solveur ligne par ligne
  ALT+2 : Solveur escargot

3. Description du modèle

3.1 Implémentation du jeu original

Pour implémenter ce jeu, nous avons pris chaque entité du jeu original et en avons fait une classe.

3.1.1 La classe Shape et ses sous-classes

La classe abstraite Shape représente une pièce. Elle possède six sous-classe qui représente chacune une pièce différente :

- EmptyShape : la pièce vide ne possédant aucune connexion
- XShape : la pièce en croix possédant quatre connexions
- TShape : la pièce en forme de « T » possédant trois connexions
- LShape : la pièce en forme de « L » possédant deux connexions
- IShape : la pièce en forme de « I » possédant deux connexions
- QShape : la pièce ne possédant qu'une seule connexion. Son nom vient de son image dans le jeu mobile Infinity Loop (un cercle et un trait).

Les principaux attributs de la classe Shape sont :

- `boolean[] connections` : représente ses connections (Nord, Est, Sud, Ouest).
  Par exemple, `connection[1] == true` correspond à une connexion vers l'Est.
- `int orientation` : représente son orientation courante.
- `int i` et `int j` : représente ses coordonnées dans la grille.

Sa principale méthode est `rotate()` qui la fait tourner de 90° vers la droite.

3.1.2 La classe Game

La classe Game représente un niveau de jeu.
Les principaux attributs de la classe Game sont :

- `Shape[][] board` : qui représente la grille de pièce.
- `int height` et `int width` : représentent les dimensions de la grille.

Cette classe sert principalement à faire des opérations sur la grille de pièce. En effet seule cette classe a accès à l'intégralité des pièces.
Le vérificateur de solution et les résolveurs vont solliciter cette classe notamment pour vérifier si une pièce est bien positionnée par rapport à ses voisins ou la bordure.
Toutes ces méthodes commencent par `isShape…()`.

3.2 Diagrammes de classe

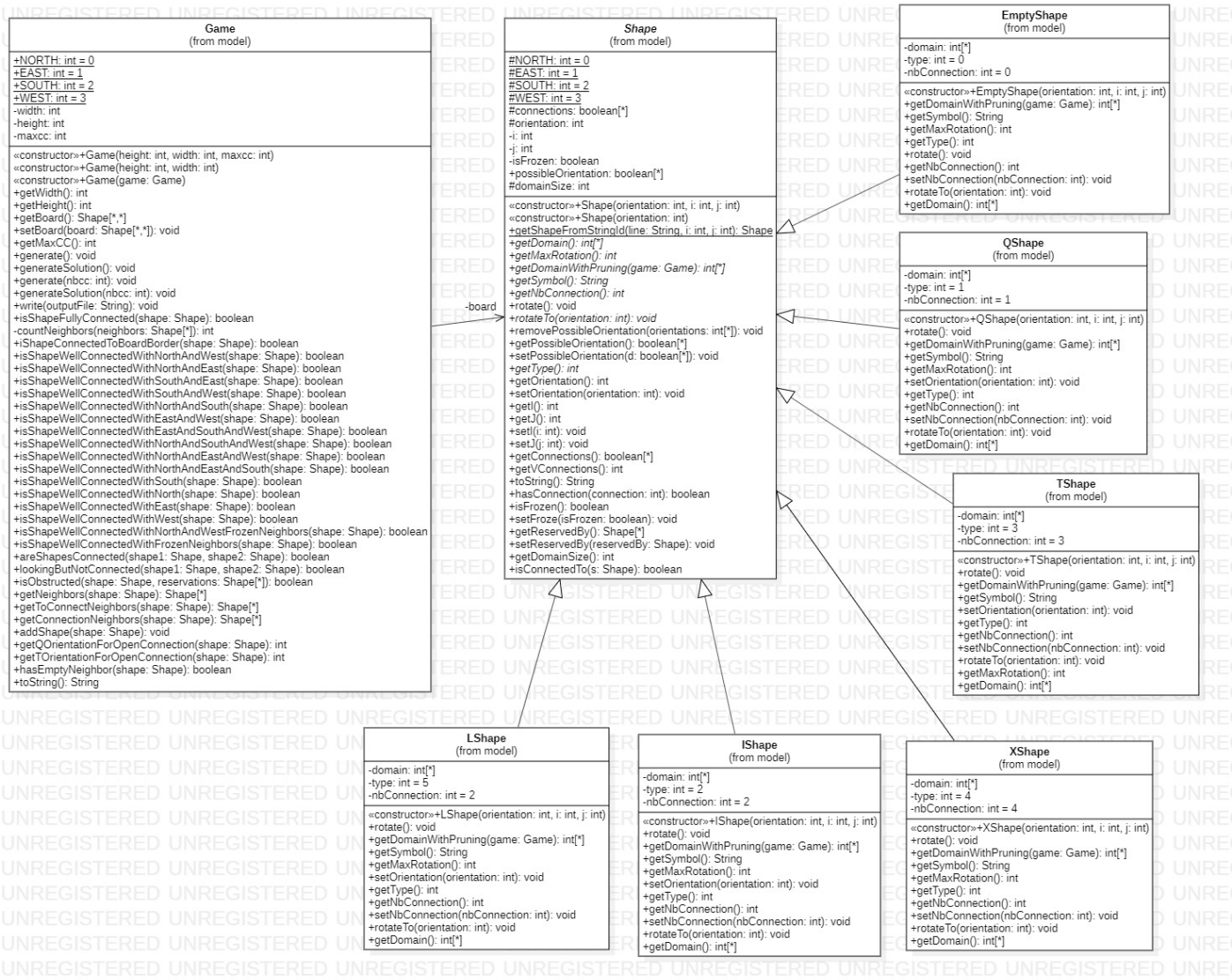

Figure 1 : Diagramme de classe du modèle

4. Lecture/écriture des niveaux et vérificateur de solution

4.1 Lecture d'une grille

Afin de récupérer les fichiers des grilles de niveau, nous avons mis en place un système de lecture de fichier. C'est la méthode Main.loadFile() qui se charge de cette mission.
Tout d'abord la méthode lit les deux premières lignes. Par convention, ces lignes représentent la hauteur et largeur de la grille. La méthode vérifie donc si les deux lignes sont bien des nombres et s'ils sont supérieurs à 0. Si oui, on les utilisent pour instancier un objet Game.
Ensuite, on itère sur la hauteur et la largeur récupérées et on lit le fichier ligne par ligne. Si cette ligne est un identifiant valide de pièce (ces identifiants se trouvent dans l'énoncé) alors on instancie une des sous-classes de Shape à partir de cet identifiant puis on ajoute la pièce à la grille. Si une des lignes n'est pas un identifiant valide alors on quitte la méthode car le fichier ne représente pas un vrai niveau.
De même, on quitte la méthode si le nombre de pièce créé est inférieur à largeur*hauteur de la grille.
Si tout se passe bien la méthode renvoie un objet Game avec sa grille bien remplie.

4.2 Ecriture d'une grille

Dans l'optique de conserver les niveaux générés par le générateur et les solutions trouvées par le résolveur il nous faut retranscrire un objet Game dans un fichier texte. Pour cela on utilise la classe PrintStream. On écrit ensuite la hauteur puis la largeur de la grille. Puis on itère sur la grille pour écrire une à une les pièces.
Il n'y a pas de difficultés particulières ici contrairement à la lecture où il faut traiter les cas de faux fichiers

4.3 Vérificateur

Pour vérifier si une grille est une solution, la classe Checker parcourt la grille et vérifie si chaque pièce est bien connectée avec ses quatre voisins. Dès qu'une pièce ne l'est pas, le vérificateur arrête et retourne que la grille n'est pas résolue.
Pour savoir si une pièce est bien connecté il faut qu'elle soit bien connectée avec son voisin au Nord, à l'Est, au Sud et à l'Ouest. Pour chaque voisin, le vérificateur regarde s'il possède une connexion vers la pièce. Si oui, alors elle doit elle-même avoir une connexion vers son voisin. Si non elle ne doit pas non plus avoir de connexion avec lui.
Dans le cas de la bordure, on regarde simplement si la pièce à une connexion vers la bordure ou non.

5. Générateur de niveaux

5.1 Générateur aléatoire de niveau

Le générateur aléatoire de niveau reprend la méthode proposée dans le descriptif du projet, à savoir, placer les pièces en fonction d'à la fois leur position dans la grille (coin et bordure) et des pièces précédemment placées. Pour cela, on pioche lors du parcours dans un ensemble de pièces qui est différent selon l'endroit dans lequel on se trouve dans la grille et on s'assure alors que la pièce choisis soit connectée avec ses voisins.

5.2 Générateur de niveau avec maximum de composantes connexes

Afin de générer un niveau avec un nombre maximal de composantes connexes, nous avons décidé d'implémenter d'abord un générateur qui en crée un nombre exact spécifié en paramètre et de choisir ensuite aléatoirement un nombre inférieur à celui-ci.
La bonne approche pour réaliser ce générateur ne nous est pas venu directement, nous avons dans un premier temps essayé de trouver une caractérisation du nombre maximal de composante connexe dans un graphe, par rapport à son nombre d'arêtes.
Nous avons trouvé cette formule :

Un graphe à n sommets avec k composantes connexes possède au moins $n - k$ arêtes.

L'idée était de diviser aléatoirement la grille en k sous-grilles. Puis dans chacune d'entre elles, construire une et une seule composante connexe. Pour chaque pièce (or pièce vide) on ajouterait 1 sommet et ½*le nombre de connexions arêtes.
Mais après plusieurs essais et schémas, cette formule ne peux pas être utilisé lors de la construction du graphe. Il peut seulement valider si un graphe déjà construit à k composantes connexes.

Nous avons alors ensuite finalement trouvé une façon plus simple de réaliser ce générateur en plaçant à tour de rôle chaque composante connexe.
L'objectif premier lors de la réalisation du générateur fut de conserver la nature aléatoire de la génération de niveau en faisant intervenir des choix le plus de fois possibles; ainsi, le choix de la case de départ de la première composante connexe est aléatoire, le choix de la première pièce à positionner est aléatoire, le choix de la pièce du voisinage à connecter (emplacements vides adjacents à la composante connexe et vers lesquelles l'une de ses variables est orienté) et la nature de la pièce placée est aléatoire, et enfin, la case de début de la prochaine composante connexe à placer est également aléatoire.
Une des difficultés majeures de la réalisation de ce générateur fut le débogage, en effet, à cause de la nature aléatoire de la génération de la composante connexe, plusieurs grilles valides pouvaient apparaître avant d'en avoir finalement une présentant le bug à corriger rendant ainsi l'action du débuggeur difficile et chronophage.
La redondance de code a été aussi difficile à éviter du fait des spécificités des positions des pièces dans la grille.

6. Résolveurs de niveaux

Notre projet comporte 4 solveur de niveaux :

1. Le ligne par ligne
2. Le CSP
3. Celui basé sur ChocoSolver
4. « L'escargot »

6.1 Résolveur ligne par ligne

6.1.1 Description

Pour le premier résolveur, nous avons suivi les indications fournies dans le sujet du projet en implémentant la méthode de résolution quasi-exhaustive. Ce résolveur, comme son nom l'indique va résoudre la grille en positionnant correctement les pièces ligne par ligne. Il commence en bas à droite de la grille. Il fait tourner la pièce jusqu'à trouver une position dans laquelle la pièce est bien connectée avec son voisin au Sud et son voisin à l'Est. Puis, il traite le voisin à l'Ouest. S'il n'a pas de voisin à l'Ouest (cas bordure), il monte d'une ligne et recommence en partant du bord Est de la grille.

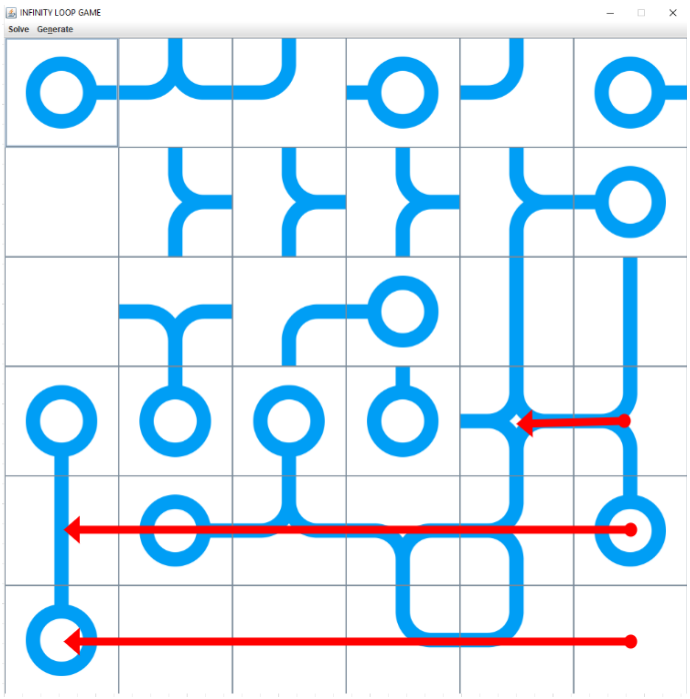

Figure 2 : Chemin de résolution dans le cas ligne par ligne

6.1.2 Première implémentation

L'avantage de ce résolveur est qu'il est assez simple à implémenter. Cependant, son plus gros défaut est qu'il n'est pas très performant puisqu'il explore tout l'arbre des possibilités de mouvement. Pour implémenter ce résolveur, nous utilisons la méthode du backtracking. Comme nous utilisons un algorithme récursif pour implémenter cette méthode, pour éviter de surcharger la pile de la JVM nous avons utilisé un objet de type Stack afin de gérer la récursivité.
Nous avons donc créé une classe Solver représentant l'algorithme de résolution et une classe State, représentant une itération de cet algorithme.
Dans cette itération, on stock une copie du jeu, la position i et j dans la grille et le nombre de rotations effectuées pour cette itération (nommé « r » par la suite).

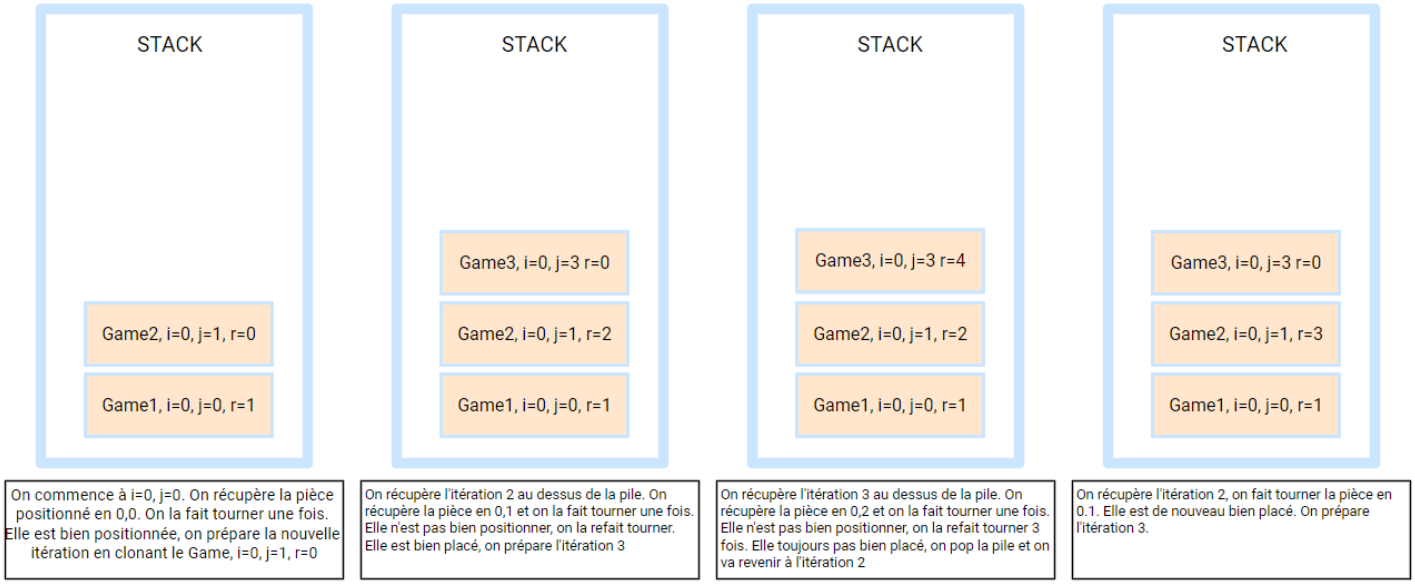

Figure 3 : Exemple montrant l'évolution l'état de la pile pour quatre itérations

6.1.3 Premières optimisations

Après la première implémentation, le solveur n'était vraiment pas performant. La première optimisation fut d'enlever l'objet Game de l'itération et de le remplacer par l'objet Shape sur laquelle l'itération interagit. Cela a amélioré les performance (on résout des niveau 15x15) mais cela reste très faible. On a ensuite enlevé la pièce de l'itération pour n'y laisser que i, j et r. Cela a permis de monter résoudre des grilles un peu plus grandes.

Ensuite, on a enlevé la vérification complète de la grille à chaque itération. En effet cela n'est pas nécessaire car par construction, le bas de la grille est forcément bien placé. (On place chaque pièce de façon qu'elle soit bien connectée avec Sud et Est). On a donc décider d'appeler le vérificateur de la pièce courante jusqu'au haut de la grille. Puis finalement nous l'avons complétement enlevé. En effet, dans le cas de grille mélangé aléatoirement il est très rare que le jeu soit résolu avant d'itérer sur toute la grille. La nouvelle condition d'arrêt de la boucle principale fut donc de savoir si la dernière pièce (0,0) est correctement placée. Après ces améliorations, le résolveur a pu résoudre des grilles de 45x45.

Ultérieurement, nous avons décidé de traiter différemment les EmptyShape et les XShape des autres pièces car elles n'ont qu'une seule orientation possible. Soit la pièce est bien placée par rapport à ses voisins du Sud et de l'Est et dans ce cas on prépare la prochaine itération, soit-elle ne l'est pas et dans ce cas on pop la pile pour effectuer un backtrack. On évite ainsi des rotations inutiles.

Après avoir implémenter ces différentes améliorations sur l'algorithme, nous chercher à optimiser son implémentation. Pour cela, nous avons utilisé le profiler Oracle Java Mission Control (abrégé en JMC par la suite). Grâce à ce dernier nous avons pu identifier les méthodes les plus consommatrices. Lors de la première utilisation il en est ressorti que la vérification `canRotate(Shape)` prenait 30% du temps de traitement (voir Capture 1).

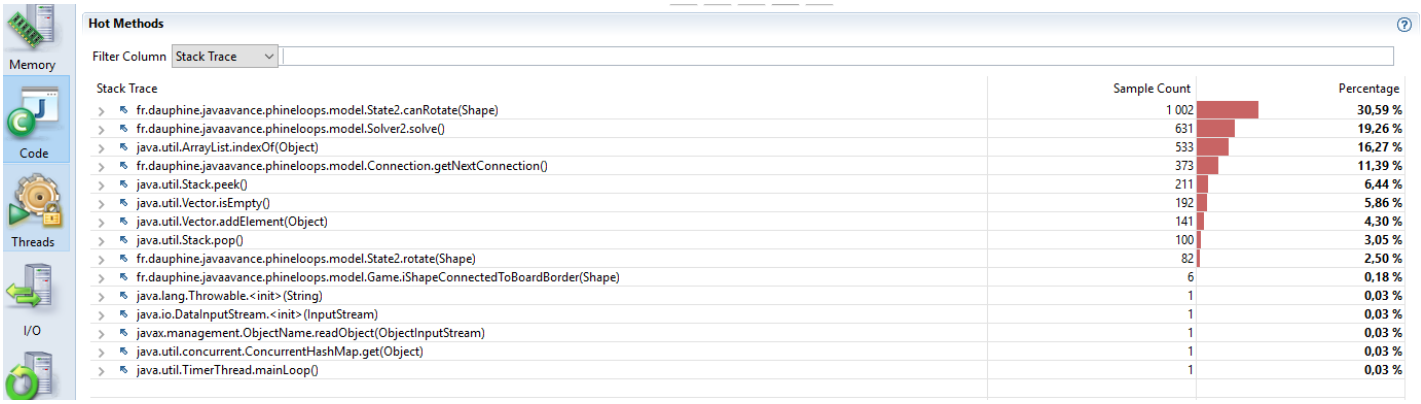

Figure 4 : capture d'écran de JMC

Ce chiffre nous paraissait trop important pour une simple vérification et nous nous sommes rendu compte que cette méthode vérifiait si la pièce passée en paramètre était une EmptyShape ou une XShape. Cette vérification se faisait en comparant par le nom de la classe de l'attribut. Nous avons supprimé cette vérification et ajouté un cas dans la boucle while de l'algorithme : Avant de faire quoi que ce soit, on regarde si la pièce est une EmptyShape ou une XShape et on la traite directement pour éviter de faire cette différenciation avec les autres pièces dans la suite de l'algorithme.

Les deux autres méthodes qui prennent beaucoup de ressource sont ArrayList.indexOf() Connection.getNextConnection(). Cela est lié à l'attribut List<Connection> connections de la classe Shape. Cet attribut représentait les connections de la pièce (N,E,S,O). Mais l'utilisation d'une énumération et d'une liste coutait beaucoup trop de ressource et nous avons remplacé cela par un tableau de quatre booléen représentant les 4 connexions d'une pièce (Nord, Est, Sud et Ouest).

Une fois cela fait, JMC a confirmé l'impact de nos modifications :

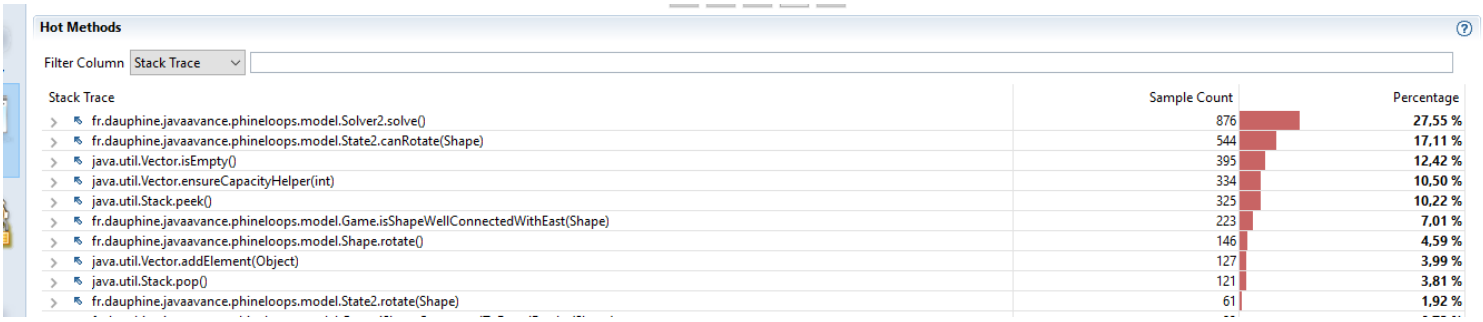

Figure 5 :capture d'écran de JMC

À la suite de cette nouvelle mesure, nous avons pu voir que l'utilisation d'un objet Stack est très couteuse. En effet les méthodes Vector.isEmpty(), Vector.ensureCapacityHelper(), Stack.peek(), Vector.addElement(), Stack.pop() représentent plus de 40% de la mobilisation du solveur.
Nous avons fait des recherches pour trouver une classe Java de gestion de pile plus performante. Nous avons ainsi découvert la classe Deque du JDK qui est beaucoup plus performante.
Après cela et quelques autres optimisations, JMC a validé ces optimisations :

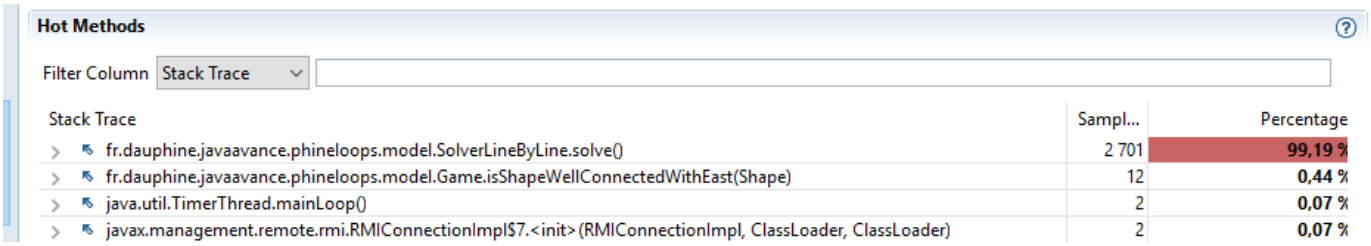

Figure 6 : Capture d'écran de JMC

La bonne nouvelle est que désormais 99% du temps du solver sert à faire de nouvelles itérations et explorer l'arbre des possibilités. A ce stade notre solver passe les grilles aléatoires de 64x64 en environ 5 secondes. N'ayant plus de piste d'optimisation de l'implémentation, nous nous sommes penchés sur la réduction du nombre de pièce à analyser.

6.1.4 Ajout du gel des pièces au début de l'algorithme

Pour réduire le nombre de pièce à traiter il fallait éliminer de l'algorithme celle n'ayant qu'une seule position possible. Une fois ces pièces identifiées nous les « gelons » dans leur seule bonne position.
Nous avons d'abord pensé à geler les EmptyShape, les XShape et celles de la bordure (les IShape, les TShape et les LShape dans les angles). De la même manière dont on traite les EmptyShape et XShape, l'itération de l'algorithme regarde maintenant si une pièce est gelée. Si c'est le cas et que cette pièce

est bien connectée au Sud, à l'Est et ses possibles voisins gelés alors on passe à l'itération suivante. Sinon c'est qu'on est sur une mauvaise branche de l'arbre des possibilités et on effectue un backtrack.

Nous est ensuite venu l'idée de geler tant que possible les voisins des pièces préalablement gelées. En effet, par exemple, une QShape voisine à une XShape gelée n'a qu'une orientation possible (celle où elle pointe la XShape).
Avec cette approche, cela permet de geler très rapidement environ un quart des pièces de la grille avant même de commencer l'algorithme d'exploration. Avec cette optimisation, le solveur traite désormais des grilles de 80x80 en quelques secondes.

Pour fixer encore plus de pièce, nous avons ajouté un attribut domaine sur la classe Shape. Cette attribut, un tableau de booléen, représente si la pièce peut être dans son orientation 0, 1, 2, ou 3.
A chaque appel de la méthode freeze() (qui gèle les pièces comme précédemment décrit), on réduit maintenant le domaine des voisins des pièces gelées. Par exemple, sur une QShape voisine d'une EmptyShape (forcément gelée) on peut éliminer l'orientation où elle la pointe.
Par addition de ces nouvelles contraintes, certaines pièces se retrouve avec plus qu'une seule orientation possible. On gèle alors la pièce et on recommence jusqu'à ce qu'aucune autre pièce ne puisse être gelée.
Après cet algorithme de préparation de grille, on réussit à geler plus de la moitié des pièces d'un niveau. Et pour certaines petite grilles (<20x20) toutes les pièces sont gelées à la fin de la préparation. Avec cette optimisation, le solveur peut maintenant résoudre des grilles de 150x150 en une seconde.

6.1.5 Clustering de la grille

Lors de la mise au point des différentes optimisations, notamment pour le gel des pièces, nous avons cherché à colorer dans la console d'Eclipse les pièces (représenté avec des symboles Unicode), selon la taille de leur domaine. Pour cela nous avons utilisé le plugin Eclipse ANSI Escape Console qui permet d'écrire dans console avec différentes couleurs.

En visualisant en couleur les niveaux préparés, nous nous sommes rendu compte que les pièces à traiter formaient des petits ilots dans la grille. Ces ilots étant la plupart du temps très petit par rapport à la grille totale, nous avons eu l'idée de les localiser et de les résoudre séparément.

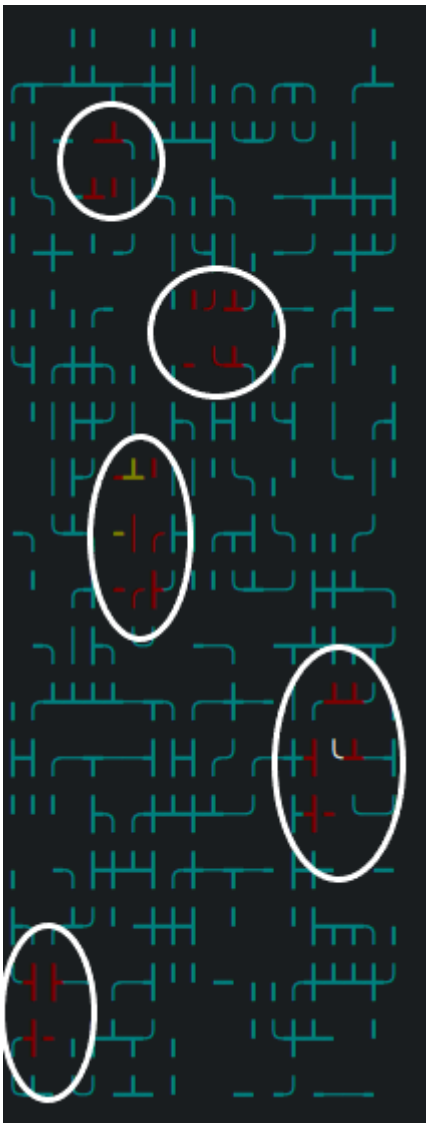

Figure 7 : Localisation des ilots de pièce non gelées

Les pièces de couleur turquoise sont gelées. Les rouges ont deux positions possibles, les vertes trois et les blanches quatre.

La problématique est de délimiter ces ilots composés de pièces connexes à traiter. Puis de les extraire en grille rectangulaire en prenant en compte leurs bordures pour en faire des sous-grilles.

Le principe de résolution est de donner cet ensemble de sous-grilles au solver afin qu'il les résolve un par un. Cette méthode devrait améliorer les performances car avec la précédente méthode dans le cas où deux clusters sont sur la même ligne, un backtrack sur un des deux entraine un backtrack sur les deux. Avec cette méthode ils sont tous indépendant. De plus dès que l'analyse d'un cluster n'a pas abouti à une solution, alors on sait que le grille n'a pas de solution. Chose qui était impossible à déterminer au préalable car cela revenait à explorer tout l'arbre des possibilités de mouvement.

Ce qui est beaucoup trop long.
Voici un exemple de ce que donne le clustering d'une grille (la grille est coupé en haut pour ne pas prendre trop de place) :

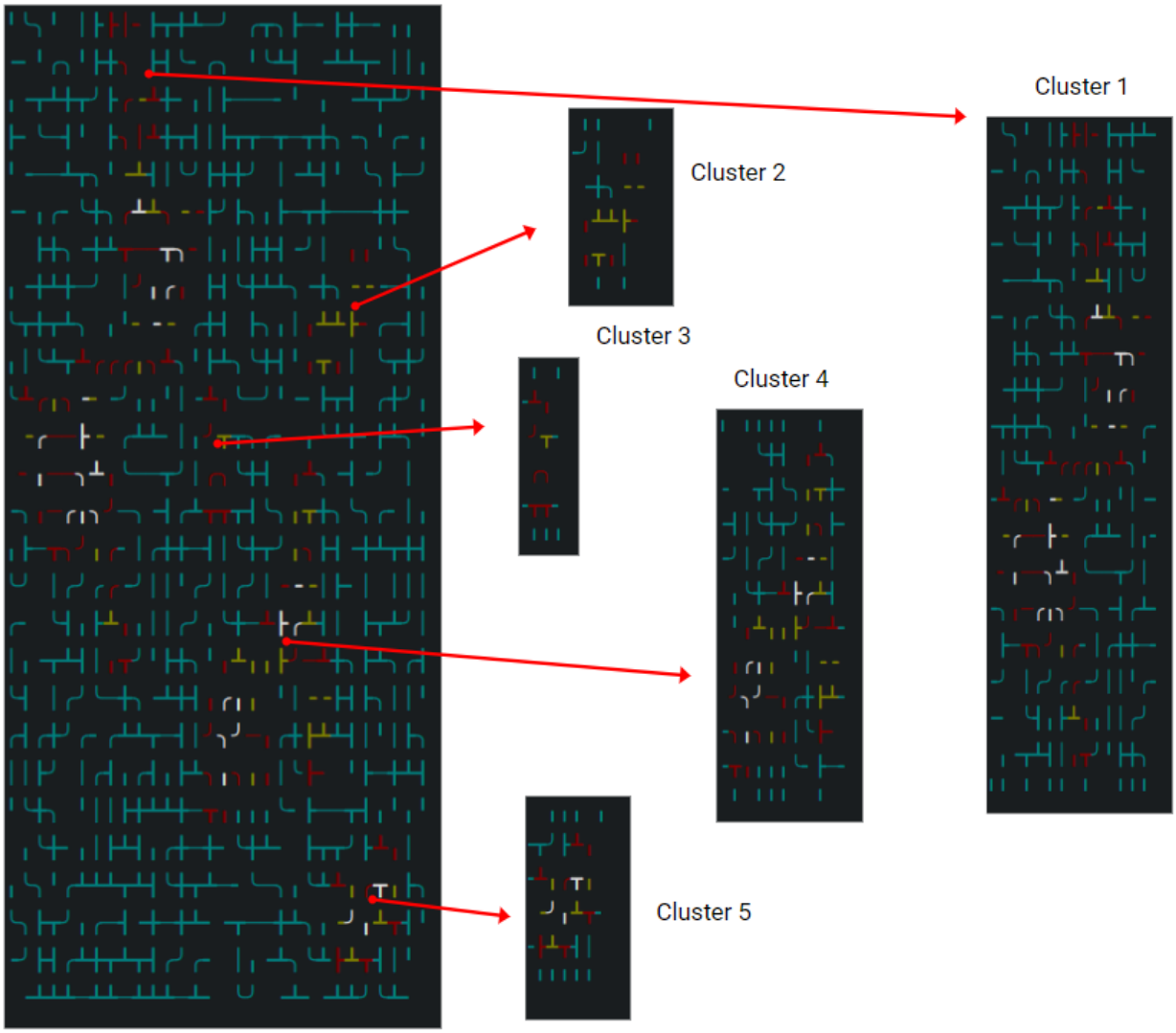

Figure 8 : Extraction des cluster d'une grille

6.1.5.1 Trouver les clusters

Pour trouver les clusters dans la grille on part d'un Set de Cluster vide, clusterSet. On parcourt la grille de haut en bas. Pour chaque pièce, on vérifie si elle est connexe à un ou plusieurs clusters du clusterSet. Grâce aux coordonnées de la pièce, on peut savoir si elle touche l'une des pièces de chacun des cluster.
Si la pièce n'est connexe à aucun cluster, on crée un nouveau cluster dans lequel on ajoute cette pièce.
Sinon on identifie tous les clusters qui acceptent cette pièce. On l'ajout au premier. S'il y en a d'autres, alors on déclenche une fusion de tous ces clusters car cela signifie que la pièce est à la frontière de plusieurs clusters.

Les clusters étant par nature indépendants les uns des autres, chaque pièce à traiter n'appartient qu'à un seul cluster. Aussi, afin de ne pas avoir à dupliquer les pièces de la grille originale dans chaque cluster puis à les réinjecter du cluster dans la grille, chaque cluster opère directement sur les pièces de la grille originale.

6.1.5.2 Construire une sous-grille à partir d'un cluster

Après avoir localisé les clusters, il faut préparer une grille afin de construire un objet Game que l'on pourra donner au solver. Pour ce faire on va extraire de la grille principale le rectangle qui délimite le cluster (Voir illustration). Cependant plusieurs paramètres sont à prendre en compte :

- Le cas où plusieurs rectangles de deux clusters rentrent en conflit. Par exemple, dans l'illustration, cela est le cas avec le cluster 3 et 4. Sur la grille de base leurs rectangles rentrent en conflit. Il faut donc remplacer cette pièce par une autre pour ne pas bloquer la résolution de l'autre cluster. On la remplace dans cette sous-grille par une EmptyShape gelée.
  Il se peut très bien que cette EmptyShape rende la sous-grille irrésolvable. Mais cela n'a pas d'impact car dans l'algorithme de résolution, nous traiteront que les pièces non gelées. De plus on sait par construction que EmptyShape ajoutée ne peut pas être voisine d'une pièce non gelée du cluster car cela signifierait que les deux clusters se touchent. Ce qui est impossible puisqu'ils auraient été fusionnés.

- Rajouter une nouvelle bordure autour des pièces non fixés du cluster afin de retranscrire les contraintes de la grille originales. Si un voisin de la pièce pointait la pièce on met une QShape, sinon EmptyShape.

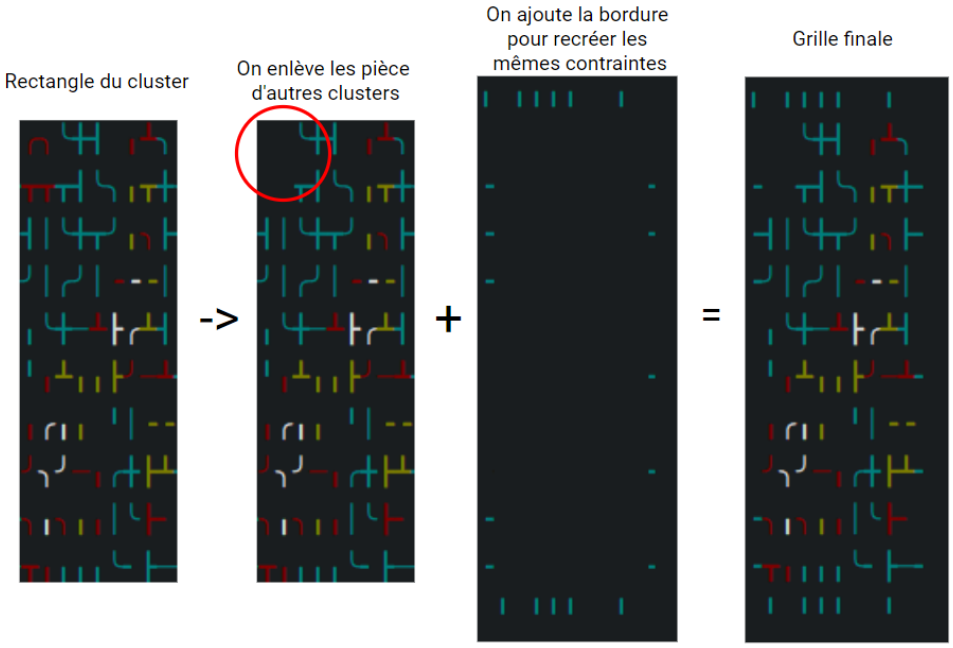

Figure 9 : Conversion d'un cluster en sous-grille

Après réflexion, on n'aurait pas forcément eu besoin de faire cette nouvelle bordure. En effet, étant donné nous traitons que les pièces non gelées, nous aurions pu simplement prendre la bordure originale du cluster.

La construction des clusters est prise en charge par la classe ClusterManager. Elle est alimentée par les différentes pièce de la grilles originale et retourne l'ensemble des clusters identifiés. Un cluster est représenté par la classe Cluster. Elle contient l'ensemble des pièces du cluster et des méthodes pour déterminer les coordonnées du cluster.

Une fois la sous-grille bien construite, on peut créer un objet Game. Le ClusterManager construit un `Set<Game>` à partir du `Set<Cluster>`. Enfin on itère sur ce Set en fournissant chaque Game au solveur. Une fois toutes les sous-grilles résolues, la grille originale l'est aussi (puisqu'elles partagent les mêmes objets).

6.1.6 Ajout de différents types de multi-threading

Au cours de l'implémentation de ce solveur, nous avons plusieurs fois utilisé le multi-threading.
La première fut de lancer quatre threads en commençant la résolution par les quatre coins de la grille. En effet, nous avons constaté que partir d'un coin peut être aléatoirement plus avantageux que partir d'un autre. Par exemple, pour un niveau, du haut vers le bas prend 5 secondes de résolution et du bas vers le haut 1 seconde. Mais pour d'autres grilles, cela ne changeait rien. Par ailleurs, cette méthode ne permet pas d'aller au-delà de quatre threads. Elle n'est donc pas vraiment efficace.

La deuxième consiste comme la première, à jouer sur l'aspect aléatoire de la grille de départ. Cette approche permet de lancer un nombre non limité de thread. Chaque thread démarre avec un mélange aléatoire du jeu de base. Mais les résultats ne furent pas satisfaisants car très aléatoires. Cela permet d'avoir potentiellement le meilleur temps possible pour une grille donnée. Mais si le solveur n'aboutit pas sur des grilles de grande tailles, cette méthode n'apporte aucune garantie que l'une des thread aboutira dans le temps imparti.

Pour mettre en place ces deux multi-threading il faut que dès qu'une thread fille trouve une solution elle arrête les autres et redonne la main à la thread principale. Pour cela nous avons utilisé un CountDownLatch, initialisé à un compteur de 1. Une fois les threads filles créées, la thread principale se met en pause jusqu'à qu'une des threads filles décrémente le latch (lorsqu'elle trouve une solution). Une fois réveillée, la thread principale récupère la grille résolue dans le singleton ThreadController où la thread fille victorieuse aura préalablement déposé la grille résolue.

6.1.7 Ajout du modèle maître-esclave

Etant donnée que le clustering de la grille originale renvoi un `Set<Game>` à résoudre, le modèle maitre-esclave est bien adapté pour traiter ces grilles. En effet le maitre dispose d'un ensemble de grilles qu'il distribue aux esclaves. Chaque esclave résout une grille. Puis lorsqu'il l'a

résolu, il prend une prochaine grille à traiter.
Pour matérialiser la distribution des grilles aux esclaves, nous avons utilisé une `BlockingQueue<Game>`. Comme pour les multi-threading vu précédemment nous avons utilisé un CountDownLatch et le singleton ThreadController. Le compteur du latch est cette fois initialisé au nombre de grille à résoudre. Il sert comme précédemment à redonner la main à la thread principale (le maitre) lorsque toutes les threads esclaves ont terminées. A chaque sous-grille résolue, la thread fille décrémente le latch. Mais si elle trouve une sous-grille sans solution, alors elle baisse le compteur du latch à 0 et indique aux autres threads, via le ThreadController, qu'il n'y a plus de grille à traiter.

6.1.8 Dernière optimisation et limite du solveur

Avec toute ces améliorations, le solveur résout des grilles aléatoire de 256x256 en environ 3 secondes.

```
Running phineloops solver.
Nb freeze : 36186
Freeze time :739 ms
Find Cluster time :1985 ms
Delimit games times :20 ms
Solving time :21 ms
Total time : 2797 ms
SOLVED: true
```

Figure 10 : Résultat du solveur pour une grille 256x256 aléatoirement générée

Ce qu'on peut remarquer est que le temps de résolution des sous-grilles est quasi instantané (21ms dans l'exemple). Cela signifie qu'une fois les sous-grilles construite, notre modèle maitre-esclave est performant. La majorité du temps est prise à geler les pièces lors de la préparation de la grille originale, puis ensuite à localiser les clusters dans la grille. Pour voir comment optimiser cela, nous avons réutilisé JMC :

Stack Trace	Sample Count	Percentage
fr.dauphine.javaavance.phineloops.model.Cluster.add(Shape)	1 000	55,90 %
fr.dauphine.javaavance.phineloops.controller.ClusterManager.addShape(Shape)	582	32,53 %
java.util.HashMap$HashIterator.<init>(HashMap)	135	7,55 %
fr.dauphine.javaavance.phineloops.controller.ClusterManager.mergeClusters(Set)	39	2,18 %
fr.dauphine.javaavance.phineloops.model.Cluster.accept(Shape)	29	1,62 %
java.util.HashMap.resize()	2	0,11 %

Figure 11 : Capture d'écran de JMC

L'analyse a montré que la méthode `add(Shape)` de la classe Cluster consomme plus de 55% du temps de traitement. A ce stade nous étions dans une impasse car cette méthode ne fait qu'appeler la méthode add() de la classe HashSet du JDK. Les autres classes de Set du JDK (TreeSet ou LinkedHashSet) n'offrent pas de meilleures performances lors d'ajout. Nous avons donc recherché

des implémentations Java plus performantes de Set.
Nous avons trouvé ces études :

- http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/

- https://www.researchgate.net/publication/313820944_Empirical_Study_of_Usage_and_Performance_of_Java_Collections

Nous avons donc essayé les différentes librairies décrites dans ces études et fait des benchmark pour notre cas. Voici les résultats :

Taille de grille	Trove	Guava	FastUtils	HPPC	Jdk HashMap
64	85	94	41	77	52
128	313	326	188	155	207
256	3311	4582	2007	2127	2900
300	6657	8924	3283	3968	6326
400	19945	31280	12464	13127	20955
512	54799	96870	35656	38831	61883

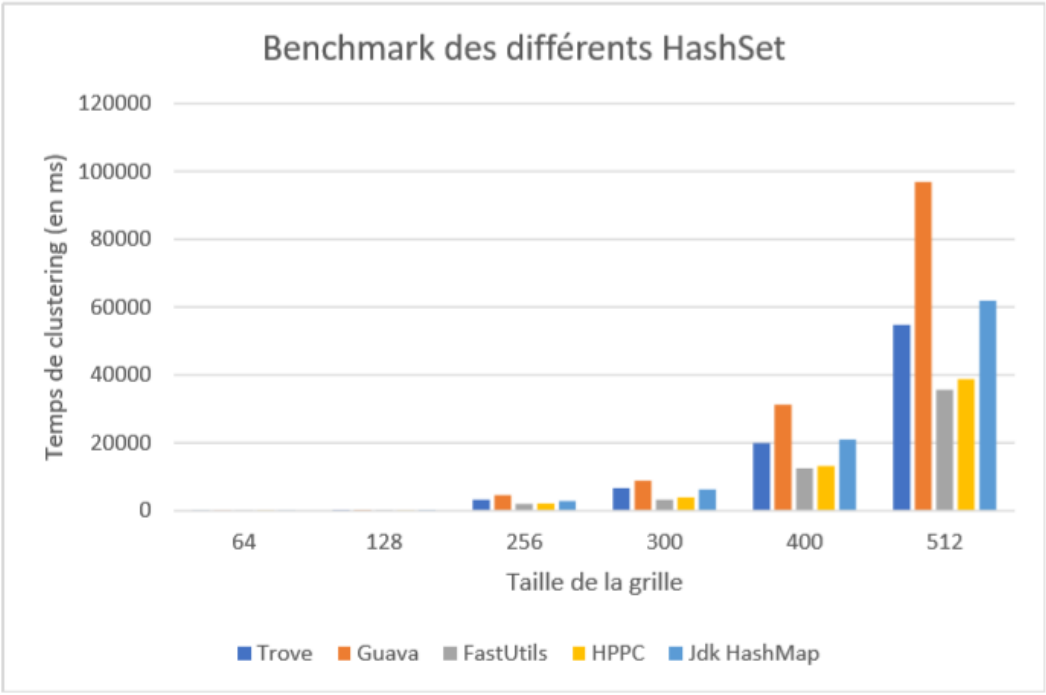

Figure 12 : Benchmark des différents HashSet

Grâce à ces benchmarks nous avons pu orienter notre choix sur la classe ObjectOpenHashSet de la librairie FastUtils. Nous avons fait une nouvelle mesure avec JMC :

Stack Trace	Sample Count	Percentage
fr.dauphine.javaavance.phineloops.model.Cluster.add(Shape)	1 949	72,35 %
it.unimi.dsi.fastutil.objects.ObjectOpenHashSet$SetIterator.next()	719	26,69 %
it.unimi.dsi.fastutil.objects.ObjectOpenHashSet$SetIterator.<init>(ObjectOpenHashSet)	17	0,63 %
fr.dauphine.javaavance.phineloops.controller.ClusterManager.findClusters(Game)	2	0,07 %

Figure 13 : Capture d'écran de JMC

Bien que l'on constate un gain de performance d'environ 50% par rapport à l'utilisation de la classe HashSet du JDK (Pour une grille 512x512, on passe de 60s à 30s), le problème reste le même et nous n'avons plus de pistes pour optimiser l'extraction des clusters.

Le principal défaut de notre résolveur est donc qu'il est très dépendant du clustering pour être efficace. En effet, comme il utilise un algorithme de résolution simple, il doit compter sur toutes les améliorations décrites précédemment pour être performant. Cette baisse de performances se voit notamment sur les grilles où l'on ne peut geler que très peu de pièces durant la phase préparatoire. Dans ces cas-là, les clusters sont de taille très importantes et on retombe sur les limites intrinsèques de ce solveur.
Nous avons constaté ce phénomène sur les grilles notées dist1 dans les instances fournies dans le repository.

Pour ces instances un algorithme de CSP semble plus adapté. Ce n'est malheureusement qu'après l'implémentation du solveur ligne par ligne et donc tard dans le projet que nous avons réalisé cela. Notre solveur basé sur ChocoSolver n'étant pas assez performant, nous avons donc dû implémenter au plus vite un nouveau solveur CSP.

6.2 Résolveur ChocoSolver

6.2.1 Description et premier essai

En parallèle de l'implémentation du solver ligne par ligne, nous voulions implémenter un solveur CSP car plus adapté à ce type de problème. Ayant utilisé en cours d'IA la librairie ChocoSolver (pour faire un solveur de Sudoku et solveur d'Hidato), nous avons directement décider de partir sur cette librairie. Ce que nous n'avions pas réalisé est le fait que dans le cas du Sudoku et de l'Hidato, les contraintes opèrent sur des nombres et sont disponible nativement dans ChocoSolver. Pour notre projet, les contraintes opèrent sur notre classe Shape. Il n'y a donc pas de contraintes nativement à

notre disposition.
Après quelques jours passés à lire la documentation de ChocoSolver et quelque essais non-concluant pour créer nos propres contraintes, nous avons abandonné ChocoSolver pour revenir optimiser notre solveur ligne par ligne par crainte de prendre du retard.

6.2.2 Reprise

Plus tard dans le projet, et après l'implémentation du générateur, nous avons décidé de reprendre l'implémentation de ce solveur. Nous avons décidé de choisir comme variables du modèle, les cases du tableau avec une pièce déjà fixée. Le domaine de chaque variable s'est alors résumé à son ensemble d'orientations possibles qui sont modélisées en tant qu'entier, ceci nous a alors permis d'utiliser des « IntVar » pour le domaine de chaque variable qui représente la modélisation d'un domaine d'entier avec laquelle ChocoSolver peut travailler.

6.2.3 Choix et limite du solveur

Pour résoudre une grille donnée, nous avons dans un premier temps tenté d'éliminer toutes les valeurs inconsistantes des variables du domaines dès le premier parcours du tableau afin de n'utiliser que les méthodes de contrainte natives de la librairie ChocoSolver. Cependant, nous avons très vite fait face à l'incapacité d'éliminer toutes les mauvaises orientations du domaine de chaque variable pour la plupart des instances en un seul parcours.
Nous avons alors décidé de créer notre propre contrainte d'interconnections pour chaque pièce et celles de son voisinage (contrainte de connexion avec au moins une), nous avons réalisé cela en créant un objet propagateur dont le rôle est de filtrer les valeurs du domaine. Une fois notre contrainte de connexion créée et envoyé au modèle pour chaque variable, l'algorithme n'a finalement plus besoin d'être « guidé » et ChocoSolver se charge de tester les positions automatiquement.
Malheureusement, le nombre de retour sur trace est combinatoire. De ce fait, le solveur peut très vite réaliser jusqu'à 5 000 000 de retour sur traces pour des instances avec simplement des grilles 14x14.
.

```
- Model[Phineloops model] features:
        Variables : 196
        Constraints : 196
        Building time : 0,035s
        User-defined search strategy : yes
        Complementary search strategy : no
- Complete search - 1 solution found.
        Model[Phineloops model]
        Solutions: 1
        Building time : 0,035s
        Resolution time : 41,298s
        Nodes: 2 600 377 (62 966,2 n/s)
        Backtracks: 5 200 605
        Backjumps: 0
        Fails: 2 600 326
        Restarts: 0
```

Figure 13: Capture d'écran des statistiques du solveur sur une grille 14x14

Face à cette complexité bien trop importante, nous avons pensé que le propagateur n'éliminait finalement pas assez de valeur des domaines, obligeant ainsi le solveur à tester trop de positions. Nous avons alors tenter d'« aider » le propagateur en utilisant des méthodes du solveur CSP que nous implémentions en parallèle, et en gelant (i.e. en réduisant le domaine d'une variable à une seul valeur) la plupart des variables avant de les passer au modèle. Cependant, malgré le fait que la plupart des variables soit instanciées, et que le solveur réalise donc beaucoup moins de retours sur trace, il s'est avéré que ses performances étaient moins bonnes que lorsque les domaines de bases des variables étaient plus grands. Au vu des performances de l'algorithme pour des grilles de grandes instances, et le temps de construction du modèle déjà supérieur aux temps de résolution des autres solveurs même pour des petites grilles, nous avons préféré continuer sur le SolverCSP nous permettant finalement d'être moins « contraint » qu'avec le ChocoSolveur.

6.3 Résolveur « escargot »

6.3.1 Description

Peu après la première implémentation du solveur ligne par ligne, une autre idée de parcours de grille nous est venue : la parcourir en spirale.

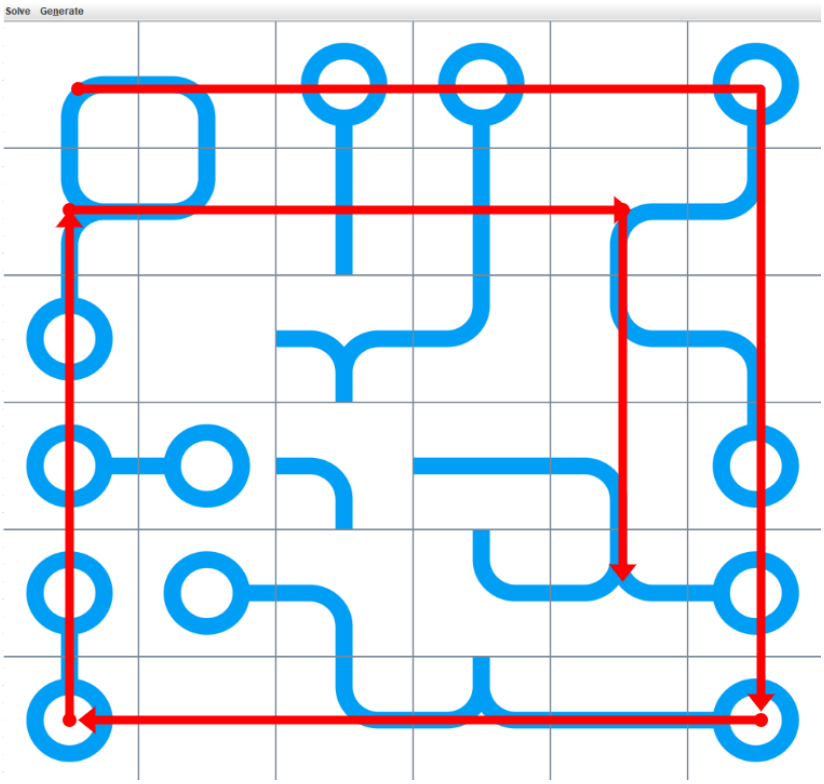

Figure 14 : Chemin de résolution dans le cas de spirale

L'idée initiale était basée sur le fait que traiter les pièces de la bordure en premier pourrait être plus performant.

6.3.2 Implémentation

Pour implémenter ce résolveur, nous avons récupéré le code du solveur ligne par ligne et modifié la façon de préparer la prochaine itération. Nous avons rajouté à l'itération la direction dans laquelle elle se trouve. En fonction de cela et des coordonnées i et j, on peut savoir s'il faut descendre, monter, aller à droite ou à gauche.
L'autre différence est qu'à chaque itération, on ne place plus la pièce en respectant les contraintes de son voisin du Sud et de l'Est mais cela est différent en fonction de la direction. Par exemple lorsque l'on descend, on vérifie le voisin du Nord et de l'Est.
Dans le cas du changement de direction, il faut aussi vérifier les contrainte du voisin de devant. Par exemple dans le cas où la direction est l'Est et que l'on change vers le Sud il faut vérifier Ouest, Nord et Est et non pas seulement Ouest et Nord.

6.3.3 Limite du solveur

Après avoir implémenter ce solveur, les résultats ne furent pas mieux que le solveur ligne par ligne. Nous sommes donc restés sur le solveur ligne par ligne et abandonné celui-ci.

6.4 Résolveur CSP

6.4.1 Description

Après avoir fini le solveur ligne par ligne avec clustering et le modèle maitre-esclave, nous en sommes venus à la conclusion que pour certaines grilles (les dist1 par exemple) il nous fallait prendre une approche basé sur un algorithme de CSP. Le ChocoSolver n'étant pas au point, nous avons décidé de faire notre propre solveur CSP, sans librairie externe.
Notre solveur utilise la méthode du Minimum Remaining Values (abrégé en MVR par la suite). Le principe est que l'on traite en premier la pièce ayant le moins de positions possibles restantes.

6.4.2 Implémentation

La plus grande différence par rapport aux autres résolveurs, c'est que dans ce cas-là, on doit déterminer la prochaine pièce à traiter par rapport à son domaine et non juste prendre le voisin.
Nous avons donc écrit la méthode pickShape() qui renvoie la prochaine pièce à traiter.
Pour le reste voici l'algorithme que nous avons mis en place :

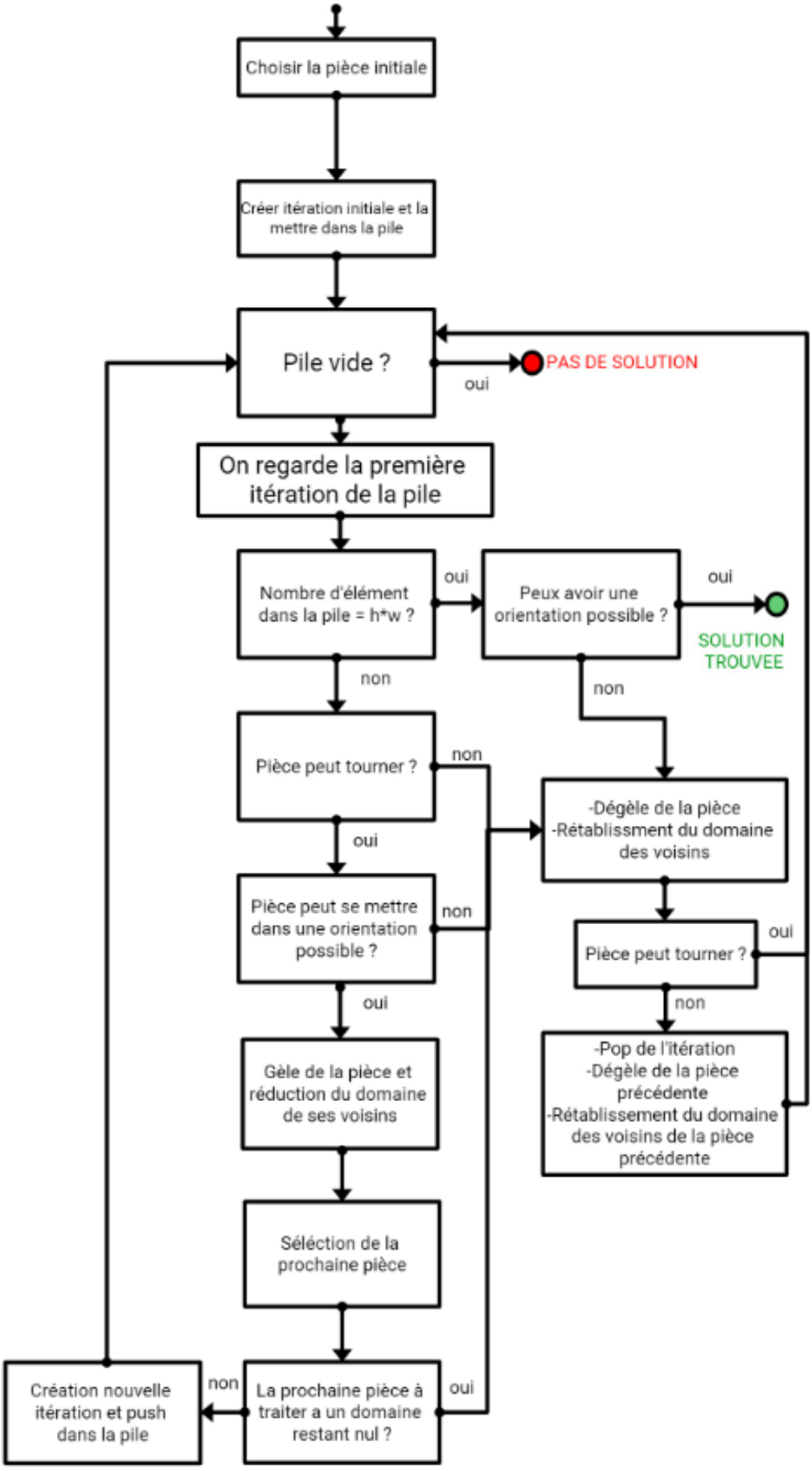

Figure 14 : Algorithme du résolveur CSP

6.4.3 Limites du résolveur

La plus grande limite de ce résolveur est qu'il se trompe pour certaine grille de taille supérieur à 40x40. Nous avons dû oublier de traiter un cas qui fait que dans certains cas le backtracking est erroné. (Sans doute un problème de domaine des voisins mal rétabli) Si nous avions directement commencé le projet par cette implémentation nous aurions pu probablement en faire un très bon solveur. Mais comme nous avons démarré son implémentation tard dans le projet, nous n'avons pas eu le temps de le finaliser.

6.5 Benchmark des différents résolveurs

On peut voir (Figure 16) que pour les grilles de 8 à 32, le CSP fait beaucoup mieux que le ligne par ligne. Cela peut confirmer l'hypothèse que si nous avions fini le résolveur CSP, il aurait était plus performant que le ligne par ligne.

Taille de grille	LineByLine	CSP	ChocoSolver	Snail
8	62	3	275	3
16	62	6	TO	31
32	60	33	TO	7081
64	93	KO	TO	TO
128	480	KO	TO	TO
256	4526	KO	TO	TO
512	33255	KO	TO	TO
TO = <60000	KO = retourne qu'il n'y a pas de solution			

Figure 16 : Benchmark des différents résolveurs (temps en ms)

Pour bénéficier de la performance du résolveur CSP sur les grilles de petites tailles mais difficiles à résoudre (que le ligne par ligne ne trouve pas), nous avons mis en place à la fin du projet un résolveur MultiSolver. Ce dernier va combiner le résolveur CSP et le résolveur ligne par ligne.

Si l'on veut résoudre une grille et que l'utilisateur accorde n threads (n>2), on lance alors dans une thread le résolveur CSP et dans une autre le résolveur ligne par ligne avec n-2 threads pour son modèle maître esclave. Pour arrêter le programme dès qu'un des deux solveur a trouvé une bonne solution, nous avons réutilisé la même méthode (celle d'utilisation d'un CountDownLatch) que pour nos multi-threading précédents.

Etant donné que le CSP renvoie de fausses solutions à partir d'une certaine taille de grille, il ne faut tenir compte de sa réponse qui s'il trouve une solution et que cette solution en est vraiment une (on le sait grâce au vérificateur). Dans ce cas on décrémente le latch pour redonner main à la thread principale.
Dans le cas où il ne trouve pas, où renvoie une fausse solution, on ne décrémente pas le latch et seul le résolveur ligne par ligne pourra alors trouver une solution.

7. Interface graphique

7.1 Débogage via Unicode

Tout au long de ce projet nous avons utilisé les caractère Unicode pour représenter un jeu dans la console. Pour cela nous avons surcharger la méthode `toString()`.De plus, comme vu dans la partie sur le clustering des grilles, nous avons utilisé de la couleur pour pouvoir voir l'état du domaine des pièces. Cela fut d'une grande aide pour l'implémentation du solveur CSP.

7.2 Ajout de l'interface graphique Swing

Pour créer notre interface graphique nous avons utilisé Swing et des sprites faits par nous-même. Notre interface se compose d'une JFrame, d'un GridLayout (représentant la grille), de ShapeButton et d'une JMenuBar.

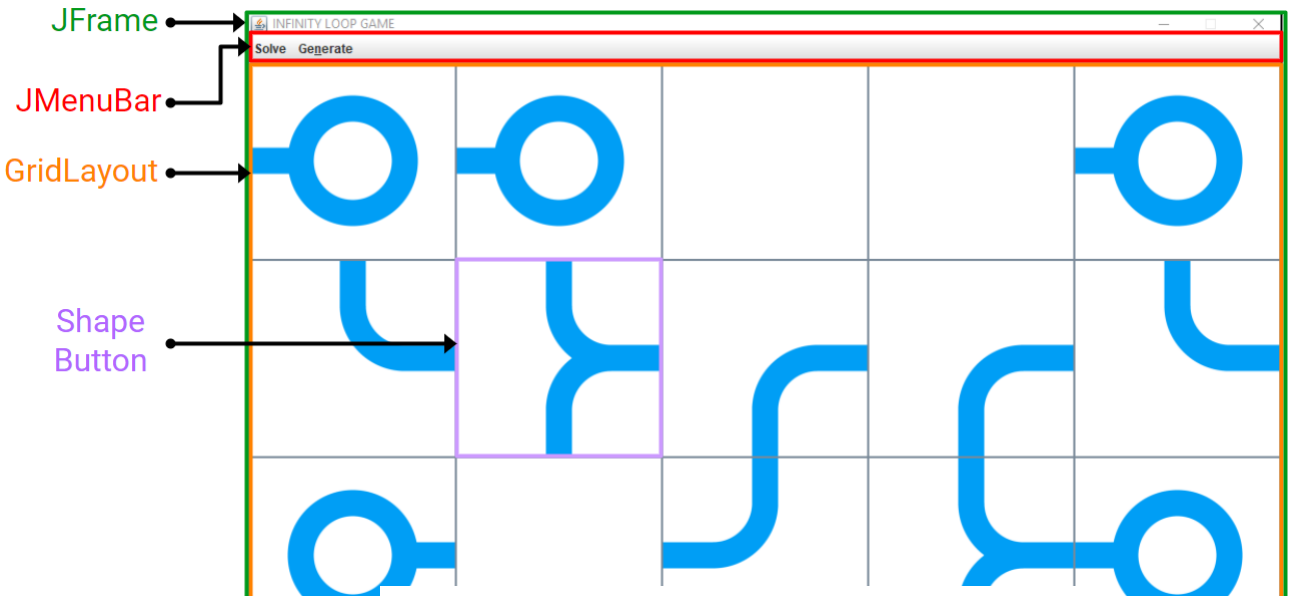

Figure 17 : Les composants de l'interface graphique

Afin de visualiser le résolveur résoudre la grille en direct, nous avons dû mettre en place une communication entre le modèle et l'interface graphique. Nous avons alors créer une classe RenderManager qui sert à faire cette liaison.
A chaque appel de `rotate()`, l'itération du résolveur va indiquer au RenderManager que la pièce est à actualiser sur l'interface graphique. Le RenderManager va alors transmettre l'ordre au GameVizualizer qui va actualiser l'image du bouton lié à cette pièce.

8. Tests Unitaires

Au cours de ce projet nous avons fait des test unitaires. Nous en avons surtout fait au début du projet pour les classes du modèle, Shape et Game.
En effet en les faisant dès le début du développement, cela nous a permis à chaque amélioration ou optimisation de faire des tests de non-régression afin de déceler au plus vite les bugs.
Par la suite nous n'en avons plus trop écrit par manque de temps.

Vers la fin du projet, pour réaliser l'implémentation du clustering de grille, nous avons fait des tests unitaires sur le clustering d'une matrice de String avant de l'appliquer à une matrice de Shape.

9. Glossaire

Profiler : Un profiler est programme qui mesure la complexité spatiale (mémoire utilisé) ou temporelle d'un programme, l'utilisation d'instructions particulières, ou la fréquence et la durée des appels de fonction.

Singleton : Design pattern dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet.