29/12/2019

Projet Java Avancé

Amrouche-Khaldi-Tran
GROUPE : KBY

Contents

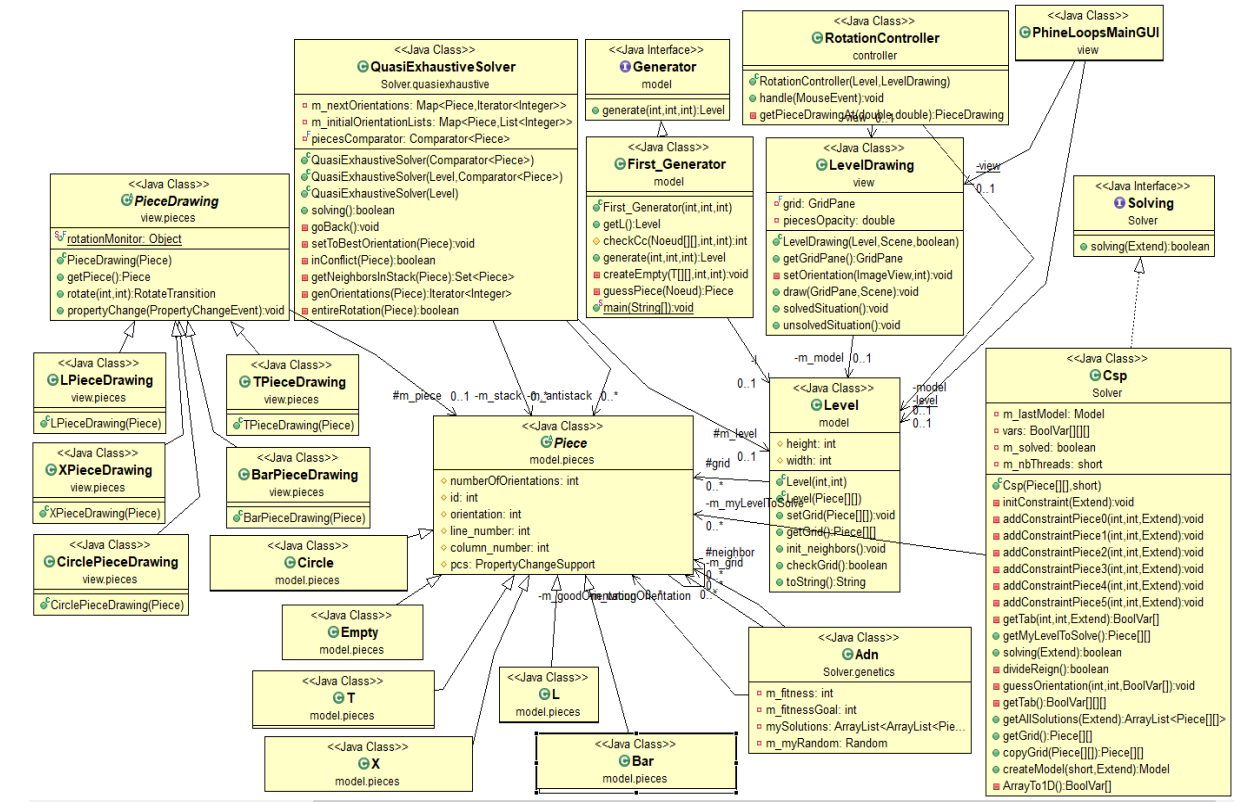
I. Introduction

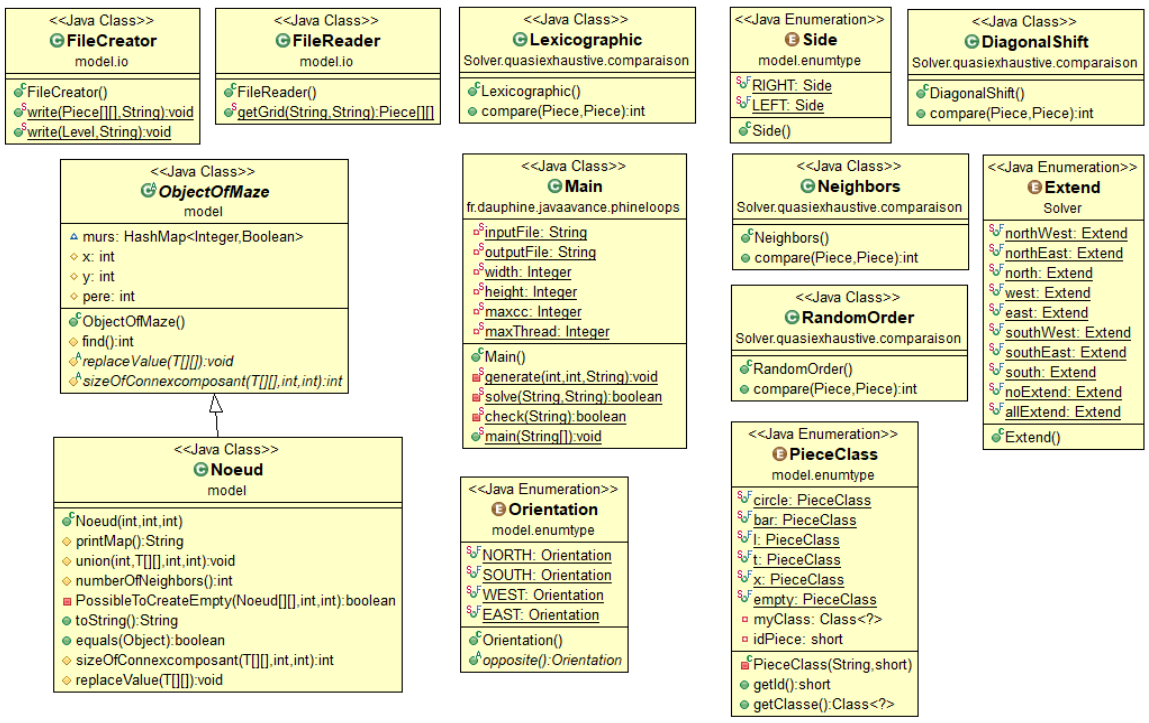
Dans le cadre du projet, il nous est demandé d'implémenter le jeu InfinityLoop ainsi qu'un solveur de grilles intelligent et des moyens de générer et vérifier des instances de niveaux. Ces problèmes sont intéressants à relever car il n'existe probablement pas de méthode directe et rapide pour résoudre un niveau. Un moyen naïf mais coûteux en temps d'exécution serait parcourir toutes les solutions possibles à l'aide d'un arbre de recherche et de vérifier chacune d'entre elles afin de trouver une solution correcte. Il est donc nécessaire d'avoir un vérificateur d'instances de niveaux. Des améliorations algorithmiques permettent d'économiser du temps de calcul en rusant sur certaines situations pour permettent de conclure plus directement qu'une solution correcte est ou n'est pas dans cette direction. Mais il reste tout de même un nombre important de possibilités à explorer. De plus, des grilles peuvent ne pas avoir de solutions possibles. Ces niveaux insolubles doivent pouvoir être détecter par nos algorithmes. Afin de développer de tels solveur, nous avons donc besoin d'instances pour tester nos méthodes de résolution. De ce fait, nous avons implémenté un générateur de niveaux dont les paramètres sont personnalisables. Par ailleurs, afin de visualiser un niveau, une interface graphique peut être utile pour mieux détecter les anomalies du niveau et/ou du solveur.

Dans ce rapport, nous montrons une manière de parvenir à résoudre n'importe quelle grille donnée par l'intermédiaire d'une recherche quasi-exhaustif dans un arbre de recherche et nous démontrerons qu'il est possible d'avoir un algorithme plus efficace et plus puissant en terme de temps de résolution via une modélisation pour vers un CSP. Nous détaillerons l'architecture de notre projet ainsi que les méthodes pour générer et vérifier efficacement un niveau du jeu.

II. Architecture

1) Diagramme UML :





2) Model :

L'implémentation du jeu est composée de trois étapes. Nous retrouvons en premier lieu le niveau (*Level*) qui contient tous les éléments (classe mère générale). Cette classe contient l'ensemble des pièces sous la forme d'un tableau à double entrée. De ce fait, on assure l'accès aux éléments en O(1). Une pièce peut être de différentes sorte dans le jeu (en forme de L, T, X...). Pour modéliser cette situation, chaque forme de pièce a sa propre classe qui étend la classe *Piece.* Chaque pièce a un attribut qui définit son orientation dans la grille. Chacune d'elles porte également les pièces situées dans son voisinage, c'est-à-dire les pièces directement au nord, au sud, à l'est et à l'ouest si elles existent (les pièces ne sont pas nécessairement connectées). Ainsi, il est possible de tirer des informations des voisins facilement pour faire les meilleurs choix dans l'exploration des solutions d'un niveau. Ces voisins sont mis dans une *Map* associant la direction (*NORTH, SOUTH, EAST, WEST* d'une classe énumérée) à la pièce voisine. Afin de faciliter les choses, une classe représentant une pièce vide est créée. Cela permet d'éviter l'apparitions des erreurs sur les pointeurs nuls.

3) Vue :

Le but ici est de fournir une représentation graphique du jeu, c'est-à-dire de chaque élément qui compose le modèle. Nous utilisons ici la librairie de JavaFX qui propose des outils et fonctionnalités destinées à l'animation et affichage graphique. JavaFX contient des outils puissants ce qui permet d'éviter une certaine lourdeur dans le code et contribue à sa meilleure lisibilité. À titre d'exemple, nous pouvons citer les classes fournissant des animations tels que RotationAnimation permettant d'afficher dans un thread à part l'animation d'une rotation de pièce, la mise en jour dynamique de la dimension des pièces après redimensionnement de la fenêtre par les méthodes *bind, widthProperty* et *heightProperty.*

Dans ce paquet, nous cherchons à visualiser le niveau (la grille) et les pièces où le tout est affiché dans un fenêtre graphiquement.
La classe qui lance toute affichage graphique s'appelle *PhineLoopMainGUI*, responsable des caractéristiques de la fenêtre (dimensions, titre...) et créée modèle et contrôleur. Elle contient principalement une grille (object GridPane de JavaFX) qui permet d'afficher des éléments de manières structurée en se repérant sur les coordonnées de cette grille. Ceci concorde parfaitement avec notre jeu où l'ensemble des pièces disposées forment une grille. Étant donnée que la grille de jeu est le seul élément du niveau à afficher, une classe *LevelDrawing* dont le rôle est d'afficher graphiquement l'instance du niveau, charge la grille. Pour cela, elle instancie chaque représentation des pièces. En effet, pour chaque pièce, un sous-type de *PieceDrawing*

est créé, représentant une pièce dans la grille de jeu. Chaque type de pièce (forme de L, T...) à une sous-classe de *PieceDrawing* qui lui correspond. Cela permet personnaliser les l'affichage de chaque type de pièce si besoin. Les classes qui peuvent être ajoutées à une GridPane sont celles qui étendent les classes *Node* de JavaFX. C'est pourquoi la classe *PieceDrawing* étend *ImageView* de JavaFX (qui est une sous-classe de *Node)*. En effet, chaque pièce affiche l'image qui correspond.

L'animation du solveur exhaustif en train de résoudre le niveau permet de visualiser et de prendre conscience de manière claire et concise la façon dont un solveur s'y prend trouver une solution acceptable. L'implémentation de cette fonctionnalité repose principalement sur une mise en place du pattern Observateur/Observé (de java.beans) ainsi que de la gestion des threads. En effet, il faut s'assurer que le solveur ne commence pas à dérouler l'algorithme d'exploration avant que le niveau soit affiché. De plus, le solveur doit également continuer à chercher sans incident lorsque l'utilisateur ferme la vue. Par dessus tout, il faut surveiller le fait que l'animation d'une rotation de pièce soit terminée avant que le solveur puisse continuer à explorer d'autres orientations. Pour cela, nous employons les méthodes wait/notify afin de synchroniser l'animation et le solveur. La difficulté vient du fait qu'une animation lancée par JavaFX se fait sur un thread indépendant et irrécupérable (la méthode *join* ne peut donc pas être utilisées).
Il est possible que les pièces ne soient plus synchronisées avec le modèle lorsque le curseur quitte la fenêtre. Il est conseillé de garder le curseur sur la fenêtre pour garder cette synchronisation.

4) Contrôleur:

Au sein de ce projet, la seule action sur la vue qui peut nous intéresser est la rotation d'une pièce. Pour cela, une classe *RotationController* est réalisée à cet effet et réagit au simple clique de la souris. Elle met à jour le modèle et vérifie si le niveau est résolu.

Pour gérer la communication entre la vue et le modèle, nous voulions nous baser sur les interfaces observateur/observé afin de mettre en marche les rotations lorsque l'utilisateur joue et lorsqu'un solveur quasi-exhaustif explore les différentes orientations des pièces tout en gardant le code le plus organisé possible. Cependant, elles ont été mis en *deprecated* depuis la version 9 de Java. Nous avons donc trouver un équivalent dans l'API java.beans. Il s'agit de *PropertyChangeListener* qui joue le rôle de l'observateur et *PropertyChangeSupport* qui assure le rôle de l'observé. Cette implémentation fournit plus d'information que la simple conception observateur/observé car il est possible de savoir quelles valeurs ont été changées. L'observateur est notifié lorsque l'orientation d'une pièce change (setter et méthode translation).

5) Gestion des entrées/sorties:

Afin de gérer les entrées/sorties, une classe spécifique à l'écriture d'un niveau dans un fichier ainsi qu'une classe dédiée à lecture d'un niveau dans un fichier sont créées. Ces classes recensent principalement une méthode pour écrire et lire un fichier. Toutes activités liées aux entrées/sorties sont déléguées à ces classes.

Pour la lecture d'un fichier, nous nous sommes inspiré de la méthode *Factory pattern* dans laquelle une classe est dédiée à la génération de la bonne sous de pièce selon l'identifiant récupéré dans le fichier. Cette classe est la classe énumérée *PieceClass*. Elle agit comme une usine de pièces et a la responsabilité de renvoyer les bonnes instances.

6) Générateur de niveaux

Afin de générer un niveau, on considère au tout début de l'algorithme une grille où chaque élément de celle-ci est une pièce et composante connexe unique. Suite à cela on fusionne les éléments avec leurs voisins de manière aléatoire de sorte à créer un labyrinthe dans la matrice. On s'assure également que le niveau généré ne contient pas de pièces non reliées aux autres, en procédant dès le début à une fusion pour chaque élément de la matrice.
Les fusions s'effectuent et l'algorithme, au fure et à mesure vérifie le nombre de composantes connexes contenues dans le niveau en cours de génération.
Celle ci s'arrête dès que le nombre de composantes connexes est atteint.
Pour finir une fois le labyrinthe créé, on fait appel à la méthode "guessPiece".
Pour chaque élément de la matrice, en fonction du nombre de ses voisins et de leurs orientation, elle crée la pièce relative à ce cas.
La fonction "generate" renvoie au final un objet de type Level, ayant pour attribut une matrice de Pièces.

7) Vérificateur de niveaux

Le vérificateur de niveau est conçu de manière assez générique de sorte à pouvoir générer tout élément de la classe "Object of Maze", en utilisant plusieurs fonctions, il vérifie que toutes les connexions de chaque pièce du labyrinthe sont reliées à d'autres pièces voisines.

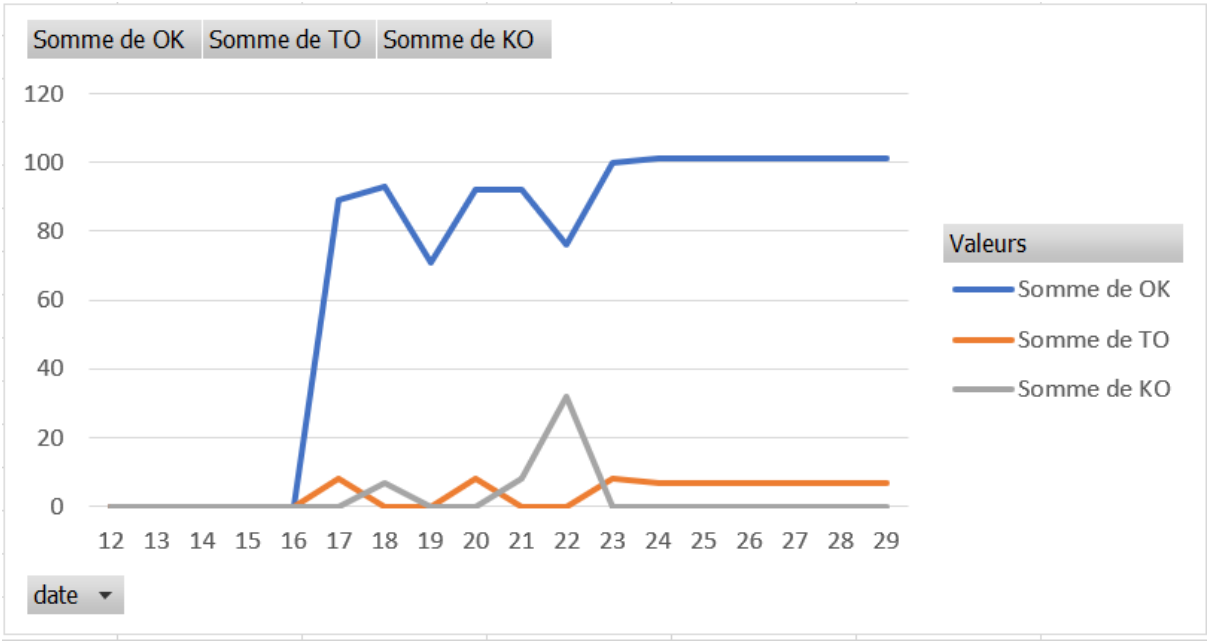
III. Solveur de niveaux

8) Solveur CSP

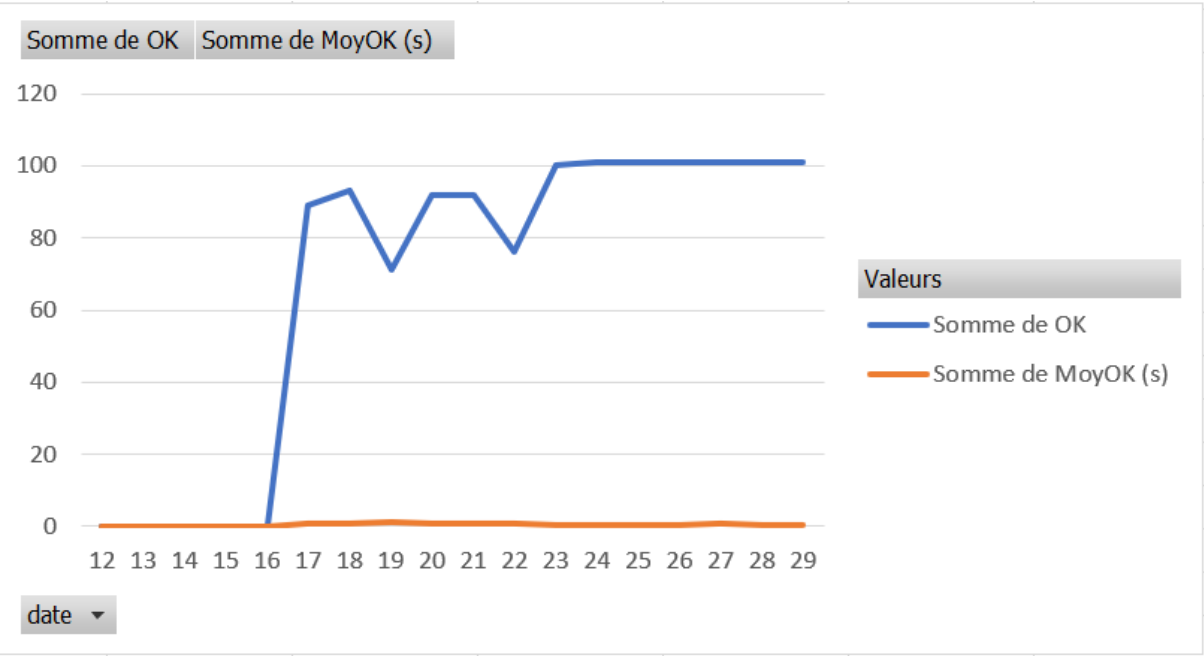
Pour l'implémentation de la solution à l'aide d'une modélisation sous contraintes nous nous sommes inspirés de la recherche de chemin dans un labyrinthe c'est-à-dire que chaque pièce de la grille représente un mouvement dans le labyrinthe et une solution admissible pour notre solveur serait de trouver un chemin sans cul de sac ou alors se terminant par la piece à une connexion.

Chaque pièce est associée à ses quatre murs, un mur est associé à une variable boolean true ou false c'est-à-dire si c'est true la pièce à un voisin de ce côté.

Et pour chaque pièce on vérifie qu'il y ait bien un nombre correct de mur ouvert par exemple le L il faut que deux murs soit ouverts et pas le nord et sud par exemple pour toutes les pièces on effectue toutes ces vérifications voire images ci-dessous.



Au début nous avions dans notre CSP pour chaque pièces quatre variables booléenne qui représente les différents mur de la pièces en question. Il est évident que certains mur sont partagée par les pièces . Donc nous avons modifié l'algorithme de définitions des contraintes en faisant en sorte qu'il n'y plus aucune variable en double cela nous as permit de réduire considérablement le nombre de variables pris en compte par le CSP . Celà d'améliorer les temps de résolutions des grilles et par la même occasion le nombre de OK des instances de test.



Afin d'améliorer cette solution qui nous permettait de résoudre seulement 101 grilles nous avion commencée à implémenter un algorithme génétique avec une population uniquement de deux individu (père et fils). Avec un procédé de Hill Climbing d'amélioration par pallier. Le principe étant de diviser la grille principale en une multitude de sous grilles afin de réduire la complexité du problème. Nous avons eu le temps d'énumérer toutes les solutions possibles pour une sous grilles. Mais nous n'avons pas eu le temps de finir l'algorithme de HillClimbing qui nous permettrait de trouver le bon agencement des sous grilles entre elle afin de résoudre la grille principale.

Pour l'amélioration via multi threading nous utilisons différentes heuristiques de recherche par thread via un portfolio disponible dans la librairie de choco solver .

9) Solveur quasi-exhaustif

Le solveur quasi-exhaustif se base sur un arbre de recherche comme décrit dans le sujet (chaque orientation de pièce à tester constitue un nœud de l'arbre et une solution complète est feuille). L'exploration de l'arbre dépend de l'ordre des pièces à considérer. Nous présentons l'exploration puis l'ordre des pièces.

L'exploration de l'arbre de fait naturellement en hauteur. Elle est gérer par deux piles. La première pile contient l'ordre des pièces à considérer. Les orientations possibles de la pièce au sommet sont testées et lorsque toutes les orientations sont épuisées et ne conviennent pas, les pièces qui sont dans cette situation sont dépilées et garder en mémoire dans la deuxième pile (anti-stack). Lorsqu'une pièce dont ses orientations ne sont pas toutes testées est atteintes, on teste sa prochaine orientation et on ajoute les pièces en mémoire de la deuxième pile vers la première.
Une orientation peut être directement rejetée si elle entre en conflit avec l'orientation des pièces déjà fixée (c'est-à-dire, celles qui sont dans la première pile). Il y a un conflit dans ces trois cas :

- la pièce testée est orientée vers une case vide ou vers le bord du plateau,
- la pièce testée est orientée vers une pièce de la première pile mais cette pièce n'est pas orientée vers elle,
- la pièce testé n'est pas orienté vers une pièce de première pile mais un de ces pièces l'est.

L'ajout de ces règles permet un gain de temps considérable face à un solveur purement exhaustif où toutes les solutions sont testées et permet de résoudre les grilles 8x8 en un temps très réduit là où le solveur exhaustif prenait un temps inconcevable.

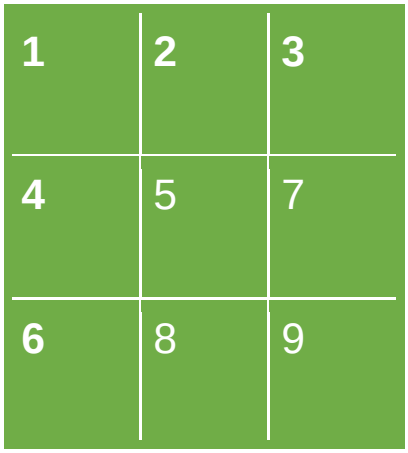
La séquence des orientations d'une pièce à tester est fournit par un objet *Iterator*. Cela permet de faciliter la lecture du code (demander la prochaine orientation et savoir s'il reste des orientations à tester) et de pouvoir se rendre compte qu'une pièce à épuisée toutes ces orientations possibles. De plus, l'ordre des orientations à tester est également modifiable. Il n'est pas nécessaire de suivre le même ordre. On peut commencer par tester la position actuelle de la pièce par exemple.

Les séquences des pièces sont stockées dans une *Map<Piece, Iterator<Integer>>.* Les méthodes utilisées sont l'ajout, la suppression et l'accès qui, implémentées (dans notre cas) avec une *HashMap*, se fait en O(1) ; le parcours de cette Map n'est pas réalisé.

Les piles sont implémentées avec *ArrayDeque* de java.util. Cette classe est, d'après sa javadoc, plus rapide que les autres classes de la même API utilisée dans le cadre d'une pile.

Pour gérer l'ordre des pièces à tester, nous mettons en place une stratégie dans laquelle un comparateur sur des pièces est demandé. Nous pouvons ainsi implémenter plusieurs comparateurs de pièces différents sans changer la classe du solveur. Le comparateur est défini lors de l'appel du constructeur. Par défaut, un comparateur lexicographique est utilisé.

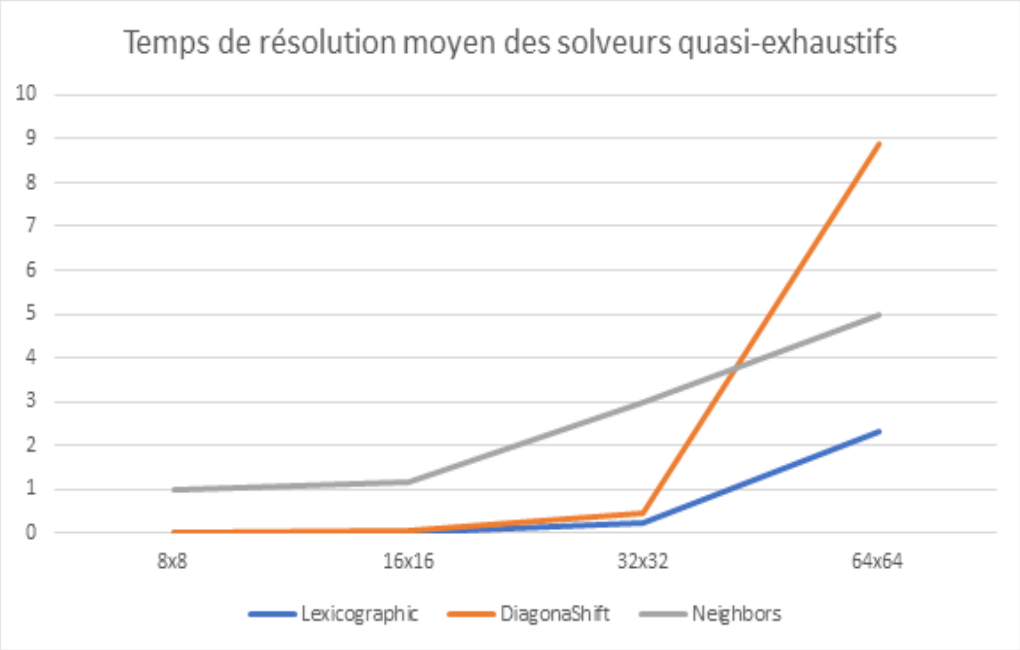
Afin d'ordonner les pièces, chacune d'entre elles est ajoutée dans une liste (ArrayList qui a une meilleure borne asymptotique supérieure pour l'ajout) puis est triée par la méthode *Collections.sort(Collection, Comparator<Piece>)*. Une autre alternative qui a été abandonnée consistait à utiliser une *PriorityQueue* avec un comparateur de pièces à la place d'une simple liste. Cette méthode a une complexité équivalente. Néanmoins, l'ordre final n'est pas stable et le comportement n'est pas toujours prévisible (la documentation de cette classe le mentionne). À titre d'exemple, le comparateur donné par *DiagonalShit* fournit un ordre différent pour les deux méthodes citées ci-dessus. Avec *Collections.sort*, l'ordre est bien celle qu'on attend (à droite) alors qu'avec une *PriorityQueue*, l'ordre diffère et nous nous retrouvons avec l'ordre ci-dessous.



Les différents comparateurs pour ce solveur sont résumé dans le tableau suivant.

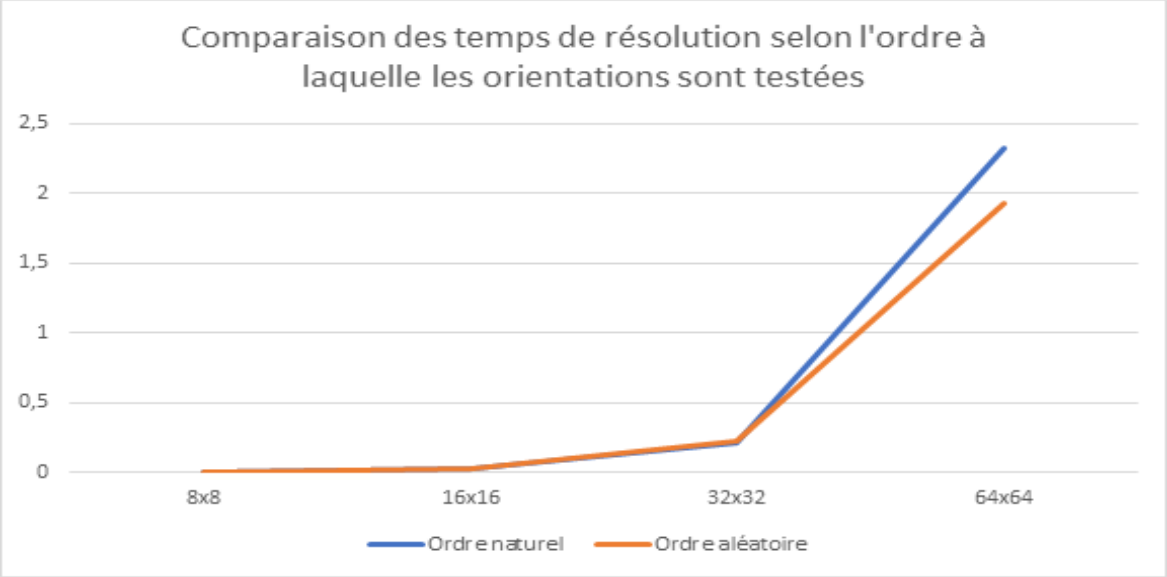
Comparateur	Description
Lexicographic	Ordre lexicographique
DiagonalShift	Ordre par rangée antidiagonale
RandomOrder	Pas d'ordre. Aléatoire.
Neighbors	Ordre en fonction du nombre de voisins que possèdent un pièce

Nous montrons ici les temps moyens du solveur quasi-exhaustif avec différents comparateurs sur le même ensemble de grilles (tirées des instances données). Nous voyons que le comparateur lexicographique réalise une meilleure performance que les autres. Un classement par nombre de voisins est assez maladroit puisque, étant donné que la détection des conflits permet d'accélérer le processus d'exploration de l'arbre, ce comparateur tend à prendre des pièces qui sont dispersées dans la grille ce qui diminue le nombre de conflits détectés dès le départ. Un comparateur "glouton" (lexicographique, diagonal) réalise de meilleures performances. Les temps sont



donnés en secondes.

Si nous changeons l'ordre des séquences d'orientations à tester pour chaque pièces, nous distinguons une différence. Celle qui ne suit pas l'ordre naturelle est moins performante. En effet, une grille dont chaque pièce est mise à son orientation 0 ne peut certainement pas être résolue. Néanmoins, si la première orientation à tester change pour chaque pièce, nous avons une chance des pièces bien orientées dès le départ. Dans le graphique ci-dessous, le ordonnanceur aléatoire réalise une meilleure performance que précédemment pour un solveur avec comparateur lexicographique.



Il est donc naturel de penser que s'il existe une heuristique pour s'approcher du meilleur ordre de priorité des orientations d'une pièce à tester, cela devrait améliorer les performances du solveur. Ainsi, une potentielle amélioration serait de calculer un score à attribuer pour chaque orientation d'une pièce donnée selon sa situation (voisinage, position dans la grille, etc.) et de trier les orientations en fonction de ce score. Cependant, le temps nécessaire pour calculer un score et trier les résultats obtenus est trop important pour permettre un quelconque gain de performance.

IV. Conclusion :

Ce projet nous a permis d'exploiter au maximum les spécificités liées à Java et vues en cours de Java Avancé ainsi que l'utilisation de Maven. Il nous a permis de mieux appréhender les concepts abordés et de savoir comment utiliser telle librairie ou quelle classe de Java utilisée dans quelle contexte afin d'optimiser le code au maximum. Dans ce projet, nous avons été confrontés à beaucoup de problèmes challengeant notamment au niveau de la gestion des threads pour l'animation ou encore pour l'optimisation du temps de calcul en testant et sélectionnant les meilleures classes de la libraire de Java.