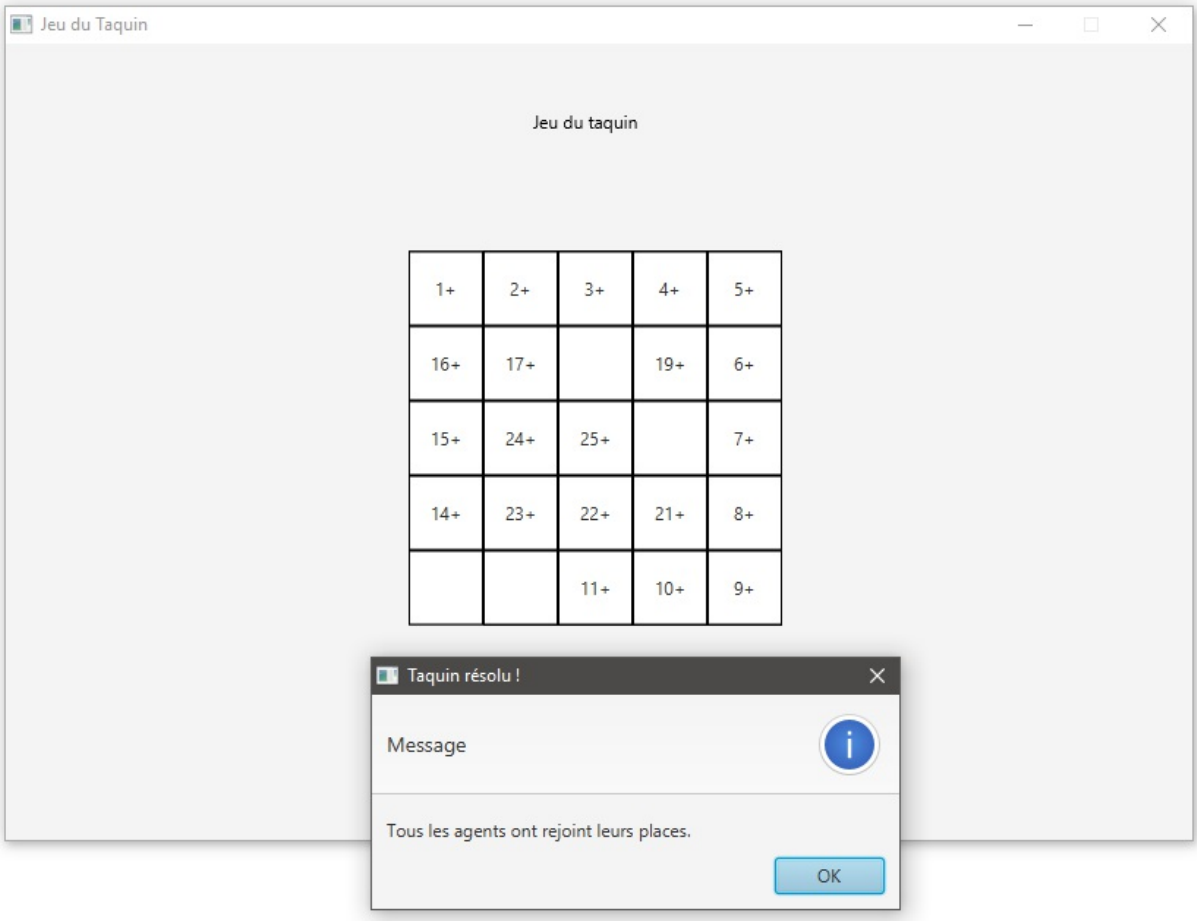
Taquin



Informations

Étudiants

- JACOUD Bastien
- REMOND Victor

Le projet

- Langage : Java
- Bibliothèque graphique : Java FX
- Projet Git

Introduction

Le but de ce projet était de mettre en place un système multi-agents permettant de résoudre un jeu de taquin.

I - Agents

Nos agents sont tous des classes filles de la classe `Thread` et chaque pièce du jeu est un agent.

Pour savoir dans quelle direction un agent doit se déplacer, nous avons décidé d'appliquer un algorithme A* (cf. notre classe `model.path.Graph`).

Afin de pouvoir gérer un plateau relativement rempli, nous avons donné à chaque agent une priorité dépendant de la position de son but sur le plateau. En effet, nous avons cherché "la plus grosse concentration de cases vides attendues", c'est à dire la zone sur laquelle, une fois le taquin terminée, il y aurait le plus de cases vides adjacentes. On essaye ainsi de garder cette zone pour la fin, car c'est celle dans laquelle il sera le plus facile de déplacer les derniers pions.

On commence par placer les pions les plus prioritaires (distribution en spirale dans le sens horaire en partant de la case en haut à gauche), les moins prioritaires sont à l'écoute des messages qui leur sont envoyés, afin de faciliter le déplacement des pions prioritaires.

Afin d'éviter un blocage général du système, on vérifie qu'un agent n'est pas déjà occupé avant de lui envoyer une requête de déplacement.

Lorsqu'un agent reçoit une requête, il la traite (en bougeant ou en envoyant une requête à un autre agent) puis renvoi une réponse à l'emetteur de la requête reçue. Si un agent reçoit une requête demandant la libération d'une case qu'il a déjà quittée, il repond tout de suite positivement.

Lorsqu'un agent emet une requête, il attend la réponse.

II - Communication

Pour gérer les communications entre les agents, nous avons créé une classe `model.communication.Messages`, regroupant la totalité des messages (représentés individuellement par la classe `model.communication.Message`). Les messages sont triés par expéditeur (id de l'agent l'ayant envoyé) dans des `HashMap<Integer, HashMap<Message.performs, Queue<Message>>>`, par type (grâce à l'énumération `Message.performs`) dans des `HashMap<Message.performs, Queue<Message>>`, puis par priorité (grâce à la priorité de l'agent expéditeur) dans la `Queue<Message>`.

Lorsqu'un agent demande à récupérer un message (en précisant s'il veut un message de type *request* ou *response*), on récupère celui qui se trouve en première position dans la queue correspondante et on le retourne.

III - Plateau

Génération

Pour générer un plateau de jeu, nous n'avons besoin que d'une taille de plateau et d'un nombre d'agents à placer.

Les agents sont placés aléatoirement, leur destination est aussi choisie aléatoirement. Une sécurité nous assure que deux agents ne seront jamais placés sur la même case, de même que deux agents ne peuvent pas avoir la même déstination.

Optimisation

Afin d'optimiser la résolution du jeu, nous avons mis en place un système de priorité pour les agents, selon la case sur laquelle ceux-ci doivent se rendre.

Cependant, nous avons remarqué que lorsque le nombre d'agents est important, il est nécessaire (ou du moins fortement souhaitable) que la case au centre soit libre, afin de pouvoir faire une résolution en spirale. Afin de s'en assurer, nous avons créé la classe `model.BoardSimplifier`. Cet classe permet de trouver la case libre la plus proche du centre et modifie la cible de certain agents pour centrer la case vide et simplifier le casse tête. Une fois le casse tête simplifié résolu, il ne reste plus qu'à replacer les agents ayant servis à la simplification.

IV - Interface Graphique

Afin de mieux visualiser notre plateau du Taquin ainsi que les différentes pièces qui le compose, nous avons décidé de mettre en place une interface graphique. Cette dernière a été réalisée à l'aide de la librairie graphique Javafx, ainsi qu'à l'aide de vues fxml.

Dans un premier temps, nous avons créé nos vues fxml qui vont composer l'application, c'est à dire : **root.fxml**, qui va être le support de notre application, **settings.fxml**, qui va être notre première vue appelée, nous permettant de sélectionner tous les paramètres avant de lancer le jeu, et enfin **game.fxml** qui va être notre composant principal contenant notre grille de taquin.

Nous avons alors créé deux controlleurs pour pouvoir gérer les deux vues principales : **SettingsController** et **GameController**. Nos deux controlleurs héritent d'une classe commune possédant une référence vers le **Main** ainsi qu'une référence vers notre classe **Board**, contenant tous les objets nécessaires pour notre taquin. Chaque classe redéfinit également la fonction `setMain(Board board, Main main)` permettant de définir le main à utiliser ainsi que le plateau de jeu qui va être utilisé.

En pratique, l'application se lance sur la fenêtre de settings, et lors d'un appui sur le bouton `launch`, l'objet **Board** est alors créé et la partie se lance. Grâce à notre main, nous changeons de fenêtre, pour visualiser notre fenêre de jeu. En fonction des paramètres de notre plateau, la grille est automatiquement créée et centrée dans la fenêtre de jeu.

Afin de mettre à jour la fenêtre graphique en temps réel, nous avons fait le choix d'utiliser le pattern Observable/Observer. Pour cela, notre objet **GameController** implements l'interface **Observer** de *java.util* et notre objet **Board** hérite de la classe **Observable** de *java.util*. Au lancement de notre application, dans la fonction `setMain()`, nous ajoutons à notre plateau l'observer de la manière suivante : `super.board.addObserver(this);`.

Nous avons alors redéfinit la méthode `public void update(Observable observable, Object o)`

dans notre controlleur afin qu'elle mette à jour la grille lors du mouvement d'un agent. Pour cela, nous passons en paramètre un tableau de **Position** composé de l'ancienne et de la nouvelle position de l'objet qui vient de se déplacer. Dans notre classe **Agent**, lorsque l'agent en question effectue un mouvement, nous appelons deux fonctions : `_board.setChanged();` pour indiquer qu'il y a eu un changement sur le plateau, et `_board.notifyObservers(new Position[]{oldPos, new Position(position)});` pour appeler la fonction *Update()* de notre controlleur, avec l'ancienne et la nouvelle position.

De cette manière, notre grille est mise à jour en temps réel lorsqu'un agent bouge sur le plateau, c'est à dire lorsque notre **Board** est modifié.

V - Conclusion

Ce qui fonctionne

La résolution de plateau remplis à 80% fonctionne presque tout le temps, certains plateaux plus remplis peuvent être resolus mais le succès n'est pas assuré.

Ce qui peut être amélioré

Traitement spécifique pour faire rentrer un agent dans un angle.

Notes

- Une part d'aléatoire a été ajoutée dans les déplacements ainsi que dans le temps écoulé entre les mouvement des agents, ainsi une situation qui a l'air de boucler peut se résoudre d'elle même au bout de plusieurs coups (cela peut être long).
- Les agents correctement placés ont leur numéro suivi d'un **+** dans la grille afin de mieux les repérer lors de l'exécution.