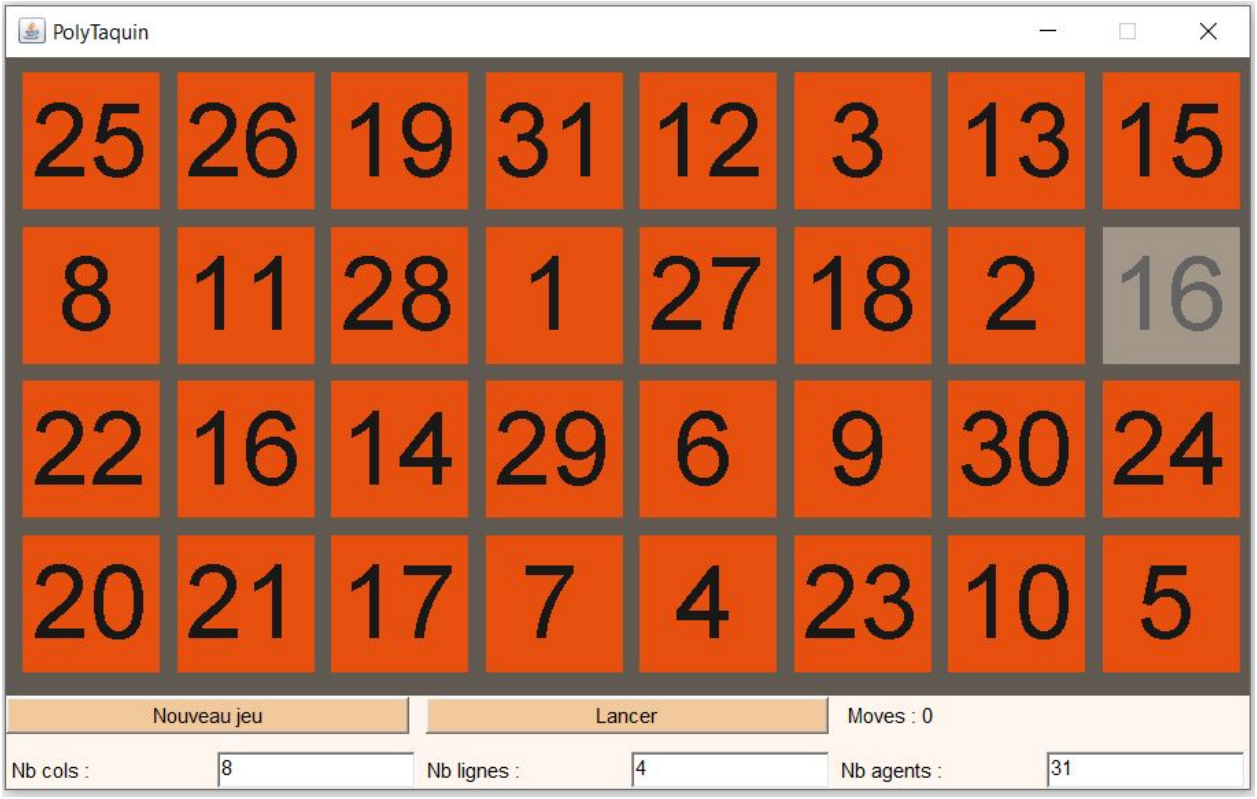
Ewald Janin

Systèmes Multi-Agents

Rapport du TP Taquin



Introduction

Dans le cadre de nos cours de Systèmes Multi-Agents, nous avons eu à programmer un projet modélisant un jeu de taquin. Relativement facile de compréhension, ce jeu, dont le but est de ranger les tuiles qui le composent dans le bon ordre, est bien plus *ardu de résolution* que ce que son *principe assez simple* laisse entendre.

Mon repository Github est disponible avec ce lien, il contient toutes les sources du projet, avec le code relativement documenté. Un jar exécutable est également en ligne sur mon repository. Il permet de lancer des exécutions du Taquin, mais pas d'avoir un affichage complet des actions comme en console. Je

recommande de l'afficher sur un écran dont la **fréquence de rafraîchissement est d'au moins 144Hz**, pour une expérience visuelle optimale. Des artefacts visuels risquent d'apparaître si la fréquence de rafraîchissement est trop basse, en ce cas, il vaudra mieux lancer le projet avec un IDE Java pour voir l'affichage de la grille dans la console.

De plus, le langage que j'ai utilisé est un **Java Development Kit 11 update 06**, pour profiter des *streams parallèles*, des *lambdas*, et d'autres fonctionnalités facilitant le **traitement multi-threading**, pour une *rapidité d'exécution optimale*.

Implémentation

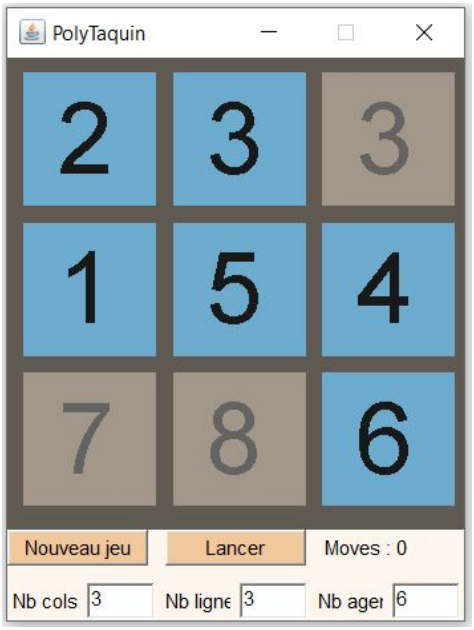
Mon code respecte du mieux que j'ai pu le *design pattern* **MVC**, avec l'utilisation du modèle **Observer/Observable**. J'ai donc une classe qui représente mon *modèle*, que j'ai appelée ***Environment***. Elle utilise les autres classes du package ***model*** (*AgentTile*, *Cell*, *Direction*, *Position*, *Message*, et *MailBox*) pour représenter le jeu du Taquin et résoudre ce dernier. Seule la classe ***Environment*** utilise les classes du package ***model***, les autres classes n'ont pas le droit de les utiliser.

Mon *controller* est la classe ***FrameTaquin***, qui utilise les classes du package ***view.utils*** (*LaunchGameListener*, *NewGameListener*, et *FrameTaquinWindowListener*) pour récupérer les actions de l'utilisateur sur les boutons. Elle possède le *modèle* ***Environment*** en attribut de classe, pour manipuler le jeu, l'instancier, en utilisant les paramètres de l'utilisateur.

Ma *vue* est la classe ***CanvasTaquin***, qui est affichée sur la fenêtre ***FrameTaquin***. Elle utilise le *modèle* ***Environment*** possédé par le *controller* ***FrameTaquin*** pour afficher le jeu actuel, en ne faisant que requêter les informations nécessaires *via* les *getters* du *modèle*.

Dans la mesure du possible j'ai utilisé la méthode de programmation *fluent interface*, pour faciliter la lecture et la compréhension de mon code, en diminuer le volume, et le rendre plus clair. C'est notamment le cas pour la classe ***Message***, ainsi que pour la classe ***Environment***, ce qui permet d'initialiser les objets plus facilement et de réduire le nombre de lignes de codes utilisées, puisque je peux chaîner les méthodes appelées en cascade.

Contrôleur et View



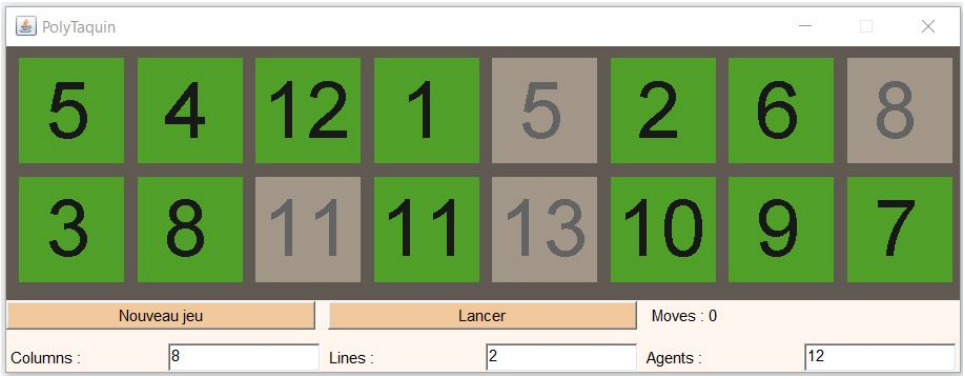
FrameTaquin

Cette classe représente le *Controller*, associée à des *EventListener*. C'est cette classe qui, à l'aide des *EventListener*, capte les actions de l'utilisateur et manipule le modèle pour créer un jeu de Taquin avec les paramètres choisis par l'utilisateur. L'interface comporte deux boutons, un pour créer ou réinitialiser le taquin (bouton ***Nouveau jeu***), tandis que l'autre sert à lancer l'exécution du taquin (bouton ***Lancer***). Les trois entrées clavier servent à définir la taille du Taquin (nombre de lignes et nombre de colonnes), ainsi que le nombre d'***AgentTile*** à placer sur la grille.

FrameTaquin implémente l'interface *Observer*, et son instance utilisée dans le programme est ajoutée aux *observers* du modèle ***Environment***. Ainsi, à chaque modification du Taquin, la ***FrameTaquin*** est notifiée et peut notifier à la *View* de se mettre à jour.

CanvasTaquin

Cette classe représente la *View*. Affichée sur la ***FrameTaquin***, elle est notifiée par cette dernière à chaque fois qu'elle doit s'actualiser et utilise les *getters* de l'***Environment*** pour dessiner l'état actuel du Taquin. Les cases vides sont dessinées en gris, et j'ai affiché le numéro des cases vides afin de connaître les emplacements sur lesquels doivent se trouver les tuiles pour que le Taquin soit complété. À chaque lancement du programme, une couleur aléatoire est choisie pour afficher les tuiles correspondant aux ***AgentTile*** pour rendre le projet un peu plus fun, par exemple dans la capture d'écran ci-dessus, ils sont en bleu, sur la capture au début du rapport, ils sont en orange, et sur la capture ci-dessous ils sont en vert.



Modèle

Environment

Représentant le jeu du Taquin, cette classe sert également de lien avec le *controller* et la *view*. Cette classe est créée avec les dimensions qui correspondront à la grille du jeu, ainsi que le nombre de tuiles sur la grille (au maximum une de moins que le nombre d'emplacements disponibles). Cette classe possède de nombreuses méthodes utilisées par les ***AgentTile***, que je ne détaillerai pas car leur nom est trivial et il y a peu d'intérêt à les expliquer, ainsi que quelques méthodes utilisées par les ***Position*** (pour vérifier la validité des coordonnées dans la grille). Les méthodes importantes sont :

- **init() :** cette méthode doit être appelée après la création d'une instance de l'environnement, et avant l'appel à la méthode ***run()***. Elle se charge d'initialiser (ou réinitialiser) l'***Environment***, c'est à dire de créer les ***Cell*** et les placer sur la grille du taquin, créer les ***AgentTile***, leur définir leur objectif de position, les placer aléatoirement sur la grille, et leur créer chacun leur ***MailBox***, ainsi que de mettre à zéro le nombre de mouvements effectués et donner leurs valeurs par défaut aux diverses variables. L'intérêt de ne pas faire cette initialisation dans le constructeur est de pouvoir relancer une initialisation aléatoire sans recréer un objet.
- **run() :** pour chaque ***AgentTile*** placé sur la grille de jeu, cette méthode crée un nouveau ***Thread***, et le lance. Si la méthode ***init()*** n'a pas été appelée avant celle-ci, une erreur est renvoyée, car le jeu n'est pas en état d'être lancé
- **update() :** cette méthode sert à notifier les *Observers* du modèle (donc la ***FrameTaquin***) d'une mise à jour du modèle, par exemple après un mouvement d'un agent, permettant d'actualiser l'affichage. Elle fait également un affichage console de l'état actuel du jeu
- **killAgents() :** cette méthode est appelée quand un utilisateur ferme la fenêtre sans que le jeu soit terminé. Ainsi, les ***Thread*** dans lesquels se trouvent chacun des agents sont tués, ce qui permet de fermer le programme sans fuite de mémoire ni exécution en arrière-plan, surtout si l'utilisateur a lancé un grande grille avec beaucoup d'agents, ce qui peut prendre très longtemps à résoudre

- **takeActionMutex() :** j'ai introduit un *mutex* dans l'environnement, qu'un ***AgentTile*** récupère quand il veut effectuer son tour de jeu. Cela assure que l'***Environment*** ne change pas entre la perception et les actions de l'***AgentTile***, et notamment les cases libres autour de lui. Cette méthode est donc appelée par les ***AgentTile*** pour tenter de récupérer le *mutex*, s'il est disponible.
- **releaseActionMutex() :** cette méthode est utilisée par un ***AgentTile*** quand il a fini de réaliser son tour de jeu, pour relâcher le *mutex* et permettre aux autres ***AgentTile*** d'effectuer leur propre tour de jeu.

AgentTile

Cette classe représente une tuile du jeu de Taquin et un agent à la fois. Chaque tuile du jeu est un agent indépendant, qui perçoit son environnement et fait ses actions en fonction de sa perception et des messages qu'il échange avec d'autres agents. Chaque agent a pour objectif de rejoindre l'emplacement qui porte le même numéro que lui. Les ***AgentTile*** découlent de la classe ***Thread***, ce qui permet de les lancer dans des processus indépendants, respectant donc au mieux le principe d'un système multi-agents. Cette classe comporte quelques méthodes essentielles que je vais détailler ci-après :

- **run() :** cette méthode est appelée au moment où les ***AgentTile*** sont lancés dans des ***Thread*** dans la méthode ***run()*** d'***Environment***. Elle s'exécute en continu jusqu'à ce que le jeu de Taquin soit terminé. Au début de son tour de jeu, l'***AgentTile*** essaie de récupérer le *mutex* de l'***Environment***. S'il échoue, il se met en pause pendant un nombre aléatoire de millisecondes, compris entre 10 et 100, puis répète cette séquence jusqu'à obtenir le *mutex*. Une fois le *mutex* obtenu, l'agent va effectuer sa ***perception***. Il va ensuite faire les appels à ***checkMessagesReceived*** puis ***checkMessagesSent***. S'il n'a toujours pas bougé à ce point et qu'il peut bouger, l'***AgentTile*** va faire appel à ***move***, puis vérifier s'il est sur son objectif. Si l'agent n'est pas satisfait, n'a pas bougé, et n'a pas déjà envoyé un ***Message*** non répondu pour demander à un autre ***AgentTile*** de bouger, il fait appel à ***sendMoveMessage***. Enfin, l'***AgentTile*** relâche le *mutex*, et se met en sommeil pendant un temps aléatoire compris entre 30 et 300 millisecondes, pour ne pas faire de famine pour les autres agents.
- **perception() :** cette méthode récupère les informations nécessaires à la prise de décision de l'agent, comme les ***Cell*** disponibles, les ***AgentTile*** à proximité…

- **checkMessagesReceived() :** cette méthode lit un par un les ***Message*** présents dans la ***Mailbox*** de l'***AgentTile***, qui lui ont été envoyés par les autres ***AgentTile***. Le seul type de ***Message*** actuellement implémenté est une demande de déplacement, c'est-à-dire quand un ***AgentTile*** demande à un autre ***AgentTile*** de libérer la ***Cell*** sur laquelle il se trouve actuellement. Si l'***AgentTile*** ne peut pas réaliser la demande, il appelle ***forwardMessage*** pour tenter de transférer la demande aux ***AgentTile*** à proximité, et si la requête a pu être transférée à au moins un ***AgentTile***, alors il répond au ***Message*** en spécifiant que la demande a été transférée. Si il a la possibilité d'accéder à une requête, il essaie de le faire en appelant la méthode ***move***, et s'il arrive à bouger, il répond positivement au ***Message***. S'il n'a pas réussi à bouger, alors il répond négativement au ***Message***, pour que l'***AgentTile*** à l'origine du ***Message*** soit notifié que sa demande a été refusée. La probabilité de continuer à lire les messages décroît avec le nombre de ***Message*** lus, et elle décroit également fortement à partir du moment où l'***AgentTile*** a répondu positivement à au moins une demande (transfert de demande ou déplacement)
- **checkMessagesSent() :** cette méthode vérifie les ***Message*** que l'***AgentTile*** a envoyés aux autres ***AgentTile*** durant ses précédents tours de jeu. Cela permet de savoir si des messages ont expiré, donc si l'***AgentTile*** est encore en train de pousser un autre ***AgentTile***, de savoir si des ***Message*** ont été transférés, ou encore si un ***Message*** a été satisfait, la méthode arrête de regarder les messages envoyés et renvoie ce ***Message*** satisfait, de manière à ce que si la ***Position*** qui était visée par l'***AgentTile*** au moment de l'envoi dudit ***Message*** est toujours disponible, il tente de se déplacer dessus en priorité. Les ***Message*** expirés, satisfaits ou transférés sont supprimés de la liste des ***Message*** envoyés.
- **forwardMessage() :** cette méthode tente de transférer un ***Message*** aux ***AgentTile*** à proximité. Plus un ***AgentTile*** est proche de l'objectif de l'***AgentTile*** actuel, plus la probabilité d'arriver à lui transférer la demande est élevée, de manière à ce que le transfert de ***Message*** aide à la fois l'expéditeur original du ***Message*** mais également les ***AgentTile*** qui le relaient, pour avoir un effet de synergie. La méthode renvoie un booléen *True* si le ***Message*** a été transféré à au moins un ***AgentTile***, *False* sinon
- **move() :** cette méthode tente de faire un déplacement sur une ***Cell*** libre du voisinage de l'***AgentTile***. De nombreux facteurs sont pris en compte dans le calcul de la probabilité de se déplacer, comme le fait d'être actuellement sur la case

objectif ou non, le fait d'avoir un des ***Message*** envoyés par que l'***AgentTile*** actuel est satisfait, transféré, non lu, expiré, ou non satisfait, le fait qu'un autre ***AgentTile*** veut que l'***AgentTile*** actuel se déplace, la différence entre la distance à l'objectif de la ***Cell*** actuelle et de la ***Cell*** visée… Si l'***AgentTile*** effectue un déplacement, il appelle les méthodes ***checkSatisfiedReceivedMessages*** et ***expireSentMessages***. La méthode renvoie un booléen *True* si un déplacement a été réalisé, *False* sinon

- **checkSatisfiedReceivedMessages() :** cette méthode est appelée si un déplacement a été réalisé (à l'aide de la méthode ***move***). Elle vide la ***MailBox*** de tous les ***Message*** obsolètes qu'elle contient, puis vérifie si le déplacement qui vient d'être fait a satisfait certains des ***Message*** reçus, auquel cas ces ***Message*** sont marqués comme satisfaits.
- **expireSentMessages() :** cette méthode est appelée si un déplacement a été réalisé (à l'aide de la méthode ***move***). Elle permet de marquer tous les ***Message*** qui ont été envoyés précédemment comme expirés, puisque l'***AgentTile*** ne se situe plus dans la même ***Cell*** que quand il a envoyé les ***Message***. Ainsi, les autres ***AgentTile*** ne prendront pas en compte ces ***Message*** expirés durant leur tour de jeu.
- **sendMoveMessage() :** cette méthode est appelée si l'***AgentTile*** ne s'est pas déplacé durant son tour de jeu, s'il n'est pas actuellement en train de pousser au moins un autre ***AgentTile***, et s'il ne se trouve pas actuellement sur sa ***Cell*** objectif. Si des ***AgentTile*** se situent à proximité de l'***AgentTile*** actuel, alors ce dernier va envoyer un ***Message*** à l'***AgentTile*** le plus proche de l'objectif de l'***AgentTile*** actuel pour lui demander de se déplacer.

MailBox

Cette classe représente la boîte aux lettres d'un ***AgentTile***. Elle contient tous les ***Message*** qui lui ont été envoyés par les autres ***AgentTile***. Elle possède des méthodes pour supprimer tous les ***Message*** obsolètes avant de lire les ***Message***, elle permet de supprimer des ***Message***, et de lire les ***Message*** les uns après les autres, à la manière d'un *Itérator*.

Message

Cette classe représente le moyen de communication entre des ***AgentTile***. Il y a un système de statut des ***Message*** (satisfait, non satisfait, transféré, non lu, expiré), qui permet aux ***AgentTile*** d'avoir un retour d'informations sur les ***Message*** qu'ils ont envoyés. Un ***Message***

comporte deux ***AgentTile*** (expéditeur et destinataire), leurs ***Position*** respectives au moment de l'envoi du ***Message*** (pour savoir si un ***Message*** est devenu obsolète ou s'il est toujours valide), ainsi que le statut et le type du ***Message*** (pour l'instant uniquement demande de déplacement). La classe ***Message*** utilise la programmation *fluent interface* pour faciliter les instanciations, par exemple avec le code :

```
Message newMsg = new Message()
.sender(agentA).senderPos(agentA.pos())
.recipient(agentB).recipientPos(agentB.pos())
.request(Message.RequestType.MOVE).answer(Message.AnswerType.NOT_READ)
;
```

Cell

La classe ***Cell*** représente une case de la grille du Taquin. Elle possède une ***Position*** pour indiquer où elle se situe sur la grille, a un numéro pour que les ***AgentTile*** puissent identifier leur objectif, et peut contenir un ***AgentTile***. C'est principalement une classe utilitaire, permettant de simplifier la représentation du Taquin, et de la rendre encore plus orientée objet.

Position

La classe ***Position*** permet de traduire des coordonnées *X* et *Y* en un emplacement sur la grille du Taquin, de comparer facilement les localisations entre deux ***Cell***, de calculer la distance (Euclidienne ou de Manhattan) entre deux ***Cell***, et de valider des coordonnées dans la grille, par exemple pour connaître toutes les ***Position*** valides à partir d'une ***Cell***, quand un ***AgentTile*** veut connaître toutes les ***Cell*** libres autour de lui. C'est une classe utilitaire que j'ai déjà utilisé dans d'autres projets et que je réadapte quand j'en ai besoin.

Direction

La classe ***Direction*** sert à représenter les 4 directions cardinales (Nord, Sud, Est, Ouest) et est utilisée avec la classe ***Position*** notamment lorsqu'un ***AgentTile*** veut connaître les ***Cell*** autour de lui.

Conclusion

Ce projet a été intéressant et m'a permis de mettre en pratique les principes du cours de systèmes multi-agents.

J'ai encore des axes d'amélioration dans ce projet, comme par exemple améliorer le système de ***Message***, en introduisant des intentions en plus des demandes de déplacements, ou passer plus de temps à régler les coefficients des probabilités, notamment de mouvement, de transfert, etc.

Il faut également que j'introduise une stratégie générale, pour que les ***AgentTile*** devant se placer en bord de grille soient prioritaires sur les autres ***AgentTile***, c'est une astuce qui facilite la résolution.

Cependant, je trouve que le Taquin n'est pas forcément adapté à un système multi-agents, et qu'une stratégie globale qui ferait toujours le déplacement optimal en fonction de l'état de la grille serait nettement plus efficace, car parfois les ***AgentTile*** s'inter-bloquent et mettent longtemps à trouver une solution. J'ai notamment trouvé le TP de tri collectif plus adapté aux systèmes multi-agents.