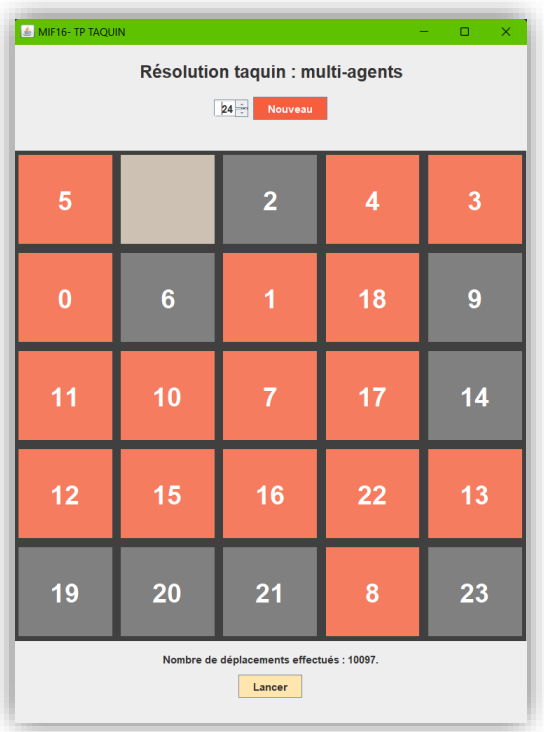
RAPPORT DE PROJET - TAQUIN

MIF16 – Techniques d'IA



Etudiants :

CHASSIN DE KERGOMMEAUX Loïc

LEFEBVRE Julien

MERCIER Loris

Responsable d'UE :

AKNINE Samir

Printemps 2023

Introduction

Le taquin est un jeu solitaire de logique consistant à déplacer des tuiles numérotées dans une grille afin de les placer dans un ordre spécifique. Dans sa version initiale, il se compose d'une grille 4x4 remplis de 15 tuiles (numérotés de 1 à 15) à replacer dans un ordre croissant. Bien que le taquin soit un jeu simple, il possède plusieurs propriétés mathématiques intéressantes dont la notion de grille solvable.

Dans le cadre de l'UE MIF16 – Techniques d'IA, nous avons été amenés à reconstruire le jeu du taquin avec une approche multi-agents de l'intelligence artificielle. Concrètement, cela signifie que le jeu ne se résout pas à travers un pilotage centralisé, mais plutôt via la mise en place d'agents autonomes communiquant entre eux afin de parvenir à leurs objectifs.

Nous reviendrons d'ailleurs dans ce rapport sur les caractéristiques et principaux résultats de notre projet avant d'évoquer la mise en place effective de notre système multi-agents, à savoir, les agents et leur communication.

Caractéristiques & Résultats

Caractéristiques techniques :

Langage : *Java*

Bibliothèque graphique : *Swing*

Lien du projet : *https://forge.univ-lyon1.fr/M1_LEFEBVRE_MERCIER/mif16_tia*

Classes

Main.java : Chargement du jeu.

Grille.java : Plateau de jeu du taquin.

Agent : Tuile de jeu du taquin fonctionnement en tant qu'agent autonome.

Message.java : Interface générale pour la gestion des messages de communication

MessageDemande.java : Message envoyé entre les agents

Messagerie.java : Boite de réception des messages pour les agents

Direction.java : Enumération listant les directions possibles d'un agent

Position.java : Classe gérant les opérations sur les positions spatiales des agents

Solveur.java : Classe permettant de savoir si une grille donnée est résoluble ou non.

vueTaquin.java : Affichage graphique de notre projet

Résultats :

- 3x3, 4x4 et 5x5 jusqu'à 1 trou : 100% de réussite
- Fonctionne aussi pour des grilles plus grandes (mais pas optimisé, cela peut être très long)
- Fonctionne aussi pour des grilles rectangulaires. (3x4, 4x5, etc…)

Nos agents

Représentation

Dans notre simulation, nos agents sont codés à travers une classe du même nom héritant de la classe *Thread.* Chaque agent représente une tuile numérotée du jeu tandis que les « trous » sont laissés comme case vide de la grille.

Etant dans une approche multi-agent, la question de la responsabilité attribuée à nos agents a été au cœur de nos préoccupations. Ainsi, chaque agent n'agit que sur sa position actuelle. Il peut se déplacer, déterminer ses voisins et envoyer des messages aux autres agents. Néanmoins, il ne possède pas de vision globale de la grille de jeu.

Fonctionnement

Tout le fonctionnement de nos agents se retrouve dans la fonction *run()* de la classe *Agent.* C'est ici que sont codées toutes les interactions et prises de décisions de chaque agent du jeu.

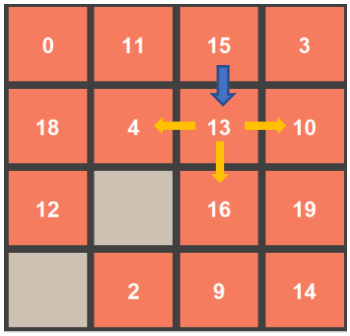
Plus concrètement, tant que l'agent est toujours actif (voir partie *terminaison d'un agent*) :

- Il lit sa boite de réception
- **S'il trouve un message valide** (=l'envoyeur n'a pas bougé entre temps)
  - **S'il peut se déplacer vers une case libre qui l'emmène vers sa destination** => il prend cette direction.
  - **Sinon**, il tire au sort une direction parmi les 3 restantes (toutes les directions sauf celle emmenant vers l'envoyeur du message, cela évite les confrontations directes). Ce tirage au sort est pondéré : 4 chances sur 5 d'aller sur une case différente de la direction de l'envoyeur. (Cela évite de rebloquer l'envoyeur au coup d'après, voir *Exemple*).

    —
  - **Si la direction mène vers une case occupée** : Il envoie un message demandant à l'agent bloquant de se décaler. Il garde en mémoire ce déplacement pour le prochain coup.
  - **Sinon**, il se déplace dans la direction choisie
- **Sinon** :
  - **S'il a un déplacement en mémoire a effectué**, l'agent essaye de le faire en envoyant des messages à d'autres agents si besoin. Si au bout de 1000 tentatives, il ne parvient pas à bouger, il oublie ce déplacement.
  - **Sinon,** si une case le rapprochant de sa destination est libre, il s'y déplace.
  - **Sinon,** il envoie un message à l'agent qui bloque sa future case en indiquant sa position et sa direction.

Exemple traitement gestion d'envoi/réception message :

Dans l'exemple ci-contre, on suppose que 15 veut aller à la position de 16. Il possède donc une direction objectif SUD. Il envoie alors un message à 13 l'invitant à se décaler. Lors de la réception du message, si 15 n'a pas bougé entre temps, 13 va se déplacer en tirant au sort parmi les directions OUEST, SUD, EST. Cependant, pour éviter de gêner 15 par la suite, les probabilités ne sont pas uniformes. 13 a plus de chance de tirer OUEST et EST (4 chances sur 5) que SUD. Une fois une direction tirée, 13 va devoir envoyer un message à l'agent correspondant à sa direction pour lui demander de se décaler, etc…



Terminaison d'un agent

Nos agents possèdent deux conditions d'arrêt.

1) L'agent s'arrête de travailler car toutes les cases sont bien placées => Le jeu est terminé. A noter que s'il s'arrêtait quand lui seul uniquement est bien placé, il pourrait bloquer d'autres cases pas encore bien positionnées.

2) L'agent peut aussi s'arrêter à une autre condition : S'il est sur une ligne du bord du taquin et que toute la ligne est bien placée. Alors, tous les agents de cette ligne peuvent finir leur exécution car ils ne gênent pas le reste de la grille. Le problème général est alors réduit à la résolution d'un taquin plus petit. Sur l'image ci-contre, la ligne supérieure est finie. Les agents de cette ligne ont fini leur exécution, ce qui réduit le problème à un taquin de taille 4x5.

Solveur de grille

Toutefois, la deuxième condition d'arrêt nécessite quelques vérifications supplémentaires. Il y a en effet un risque à prendre en compte : celui où le reste de la grille n'est pas solvable. Dans ce cas, un taquin résoluble pourrait passer dans un état non-résoluble. Notre optimisation d'algorithme bloquerait alors le jeu.

Pour pallier ce risque, nous avons implémenté un algorithme permettant de vérifier si une grille donnée est solvable ou non. Nous pouvons ainsi savoir si le jeu est possible ou non. A noter que ce solveur ne fonctionne qu'avec des grilles à une case vide.

Ce solveur est utilisé à 2 occasions :

1) A la création du jeu, nous ne générons que des taquins résolubles.
2) Lorsqu'une ligne du bord du taquin est bien placée, nous vérifions si la grille restant est résoluble ou non. Si oui, les agents formant la ligne du bord arrêtent leur exécution.

Communication

La communication entre agents s'effectue grâce aux classes *MessageDemande* et *Messagerie.*

Chaque message se compose de l'identifiant de l'agent envoyeur, de sa position, de sa destination ainsi que de la position du receveur.

Les positions de l'envoyeur et du receveur permettent de vérifier la validité d'un message. Si l'un des deux agents a bougé entre l'envoi et la lecture du message, alors ce-dernier n'a plus réellement d'intérêt. La direction de l'envoyeur permet de connaitre ses envies afin de se déplacer si possible sur une direction perpendiculaire à celle-ci.

Notre système de communication repose également sur la classe *Messagerie*. Cette classe gère la mémoire, le stockage, l'accès et la suppression des messages envoyés entre les agents. C'est une sorte de boite postale fonctionnement en mode FIFO. Les messages sont accessibles dans l'ordre dans lequel ils ont été reçus.

Interface graphique

En bonus de ce projet, nous avons pris le temps de développer une interface graphique de notre application. Cette interface est réalisée via la bibliothèque graphique *swing*.

Bien que notre projet ne repose pas sur un modèle MVC complet, cette interface utilise malgré tout ce principe à travers un pattern Observer. L'affichage se rafraichit alors à chaque déplacement des tuiles.

Simple mais efficace, cette fenêtre permet à l'utilisateur de choisir le nombre de tuile souhaitée, de générer autant de partie qu'il souhaite et de voir en temps réel les tuiles bien placées et celle en cours de placement.

Conclusion

L'objectif de résolution des grilles remplies à 80% fonctionne donc parfaitement quelle que soit la taille de la grille. Nous avons pour cela utilisé un principe de communication efficace et une approche probabiliste pour déterminer des directions optimales de déplacement. Notre solveur est également une vraie plus-value à notre projet nous permettant de réduire progressivement des problèmes de taille N à des problèmes de taille N-1.

En conclusion :

Ce qui fonctionne ->

- Résolution complète de taquin 3x3, 4x4, et 5x5
- Résolution de taquin rectangulaire 3x4, 4x5 ou de taquin carré plus grande
- Conception d'un solveur vérifiant si une grille est solvable
- Conception d'une interface graphique

Ce qui pourrait être amélioré ->

- Des messages entre agents plus élaborés
- Des gestions de cas particulier (agent dans un angle par exemple).