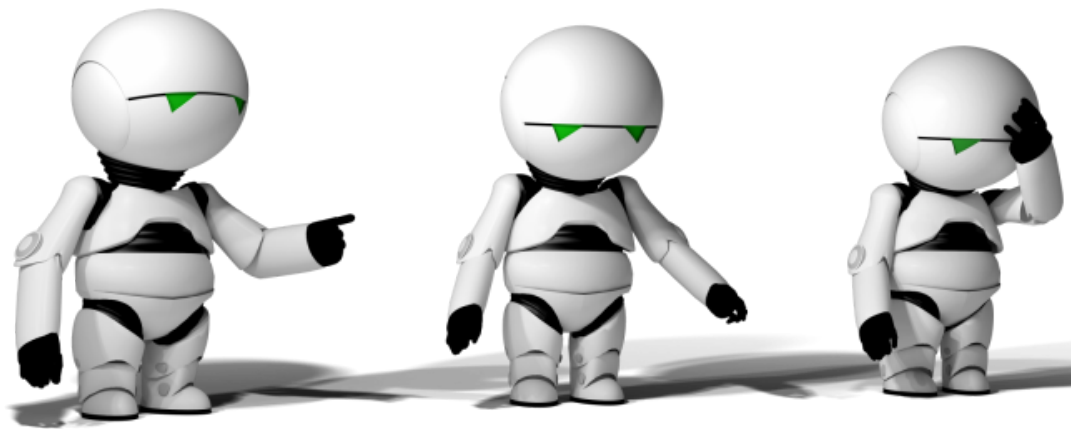


DEGUT Julian

POLYTECH 4A INFO

GUYOT DE LA POMMERAYE Benjamin



MULTI-AGENTS

INTELLIGENCE ARTIFICIELLE



[Lien du Github](#)

@jDegut

@LiakisGuyot

Table des matières

Introduction	4
Structure du projet	4
Approche naïve	7
Présentation	7
Pseudo code de l'agent.....	7
Tests	7
Limites	7
Approche cognitive	9
Présentation	9
Pseudo code de l'agent.....	9
Tests	9
Difficultés rencontrées.....	10

Introduction

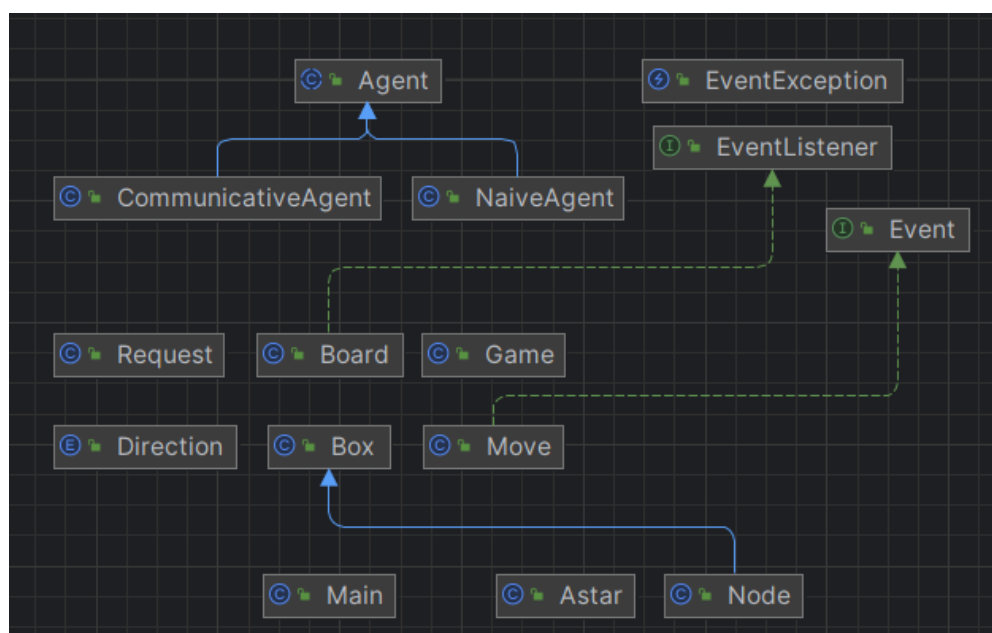
Notre objectif durant ce projet est d'utiliser un système multi-agent pour résoudre un puzzle via une approche cognitive. Nous avons choisi le jeu du Taquin comme puzzle à résoudre. Notre objectif final est d'être capable de résoudre un jeu de Taquin de taille $n \times n$. Pour faciliter le début de notre projet, nous supposons au début que $n = 5$.

Nous numérotons de la façon suivante les différentes pièces de notre grille

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Structure du projet

Nous avons structuré notre code de la façon suivante :

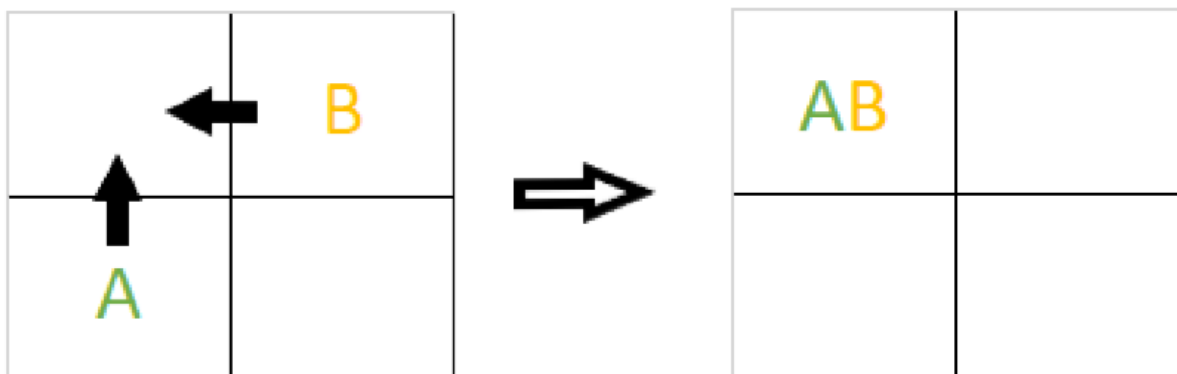


Nous réalisons un plateau de puzzle *Board* composé de $n \times n$ *Box*. Notre *Agent* se déplace via la classe *Move* dans la *Direction* souhaitée (haut, bas, gauche, droite). Notre *Agent* peut aussi choisir de ne pas se déplacer. Chaque *Agent* est associé à une *Box* dans une *HashMap* afin d'optimiser la recherche de ces derniers : c'est le plateau.

Notre projet est structuré selon le patron « écouteur d'évènements » afin de permettre de faire communiquer les agents avec le plateau. Chaque agent envoie des requêtes au plateau (« Je veux me déplacer vers la droite ») et le plateau met à jour la map à la réception de ces requêtes (« Ok ! Je mets à jour »). Cela permet d'accumuler les requêtes (s'effectuant de manière séquentielle puisque les mises à jour sont synchronisées) et de notifier le plateau des mises à jour.

Nous associons à chaque *Agent* un thread afin de permettre aux différents agents de s'exécuter en même temps. De la sorte nous pouvons simuler de "réelles" interactions entre nos agents et non faire jouer à tour de rôle les agents avec une instance globale qui gère les agents.

Cependant puisque chaque *Agent* dispose de son propre thread, nous devons gérer l'aspect concurrentiel de notre programme. Nous pourrions nous retrouver dans le cas où deux agents décident en même temps de se rendre sur la même case. En effet, s'exécutant en concurrentiel, il est possible que nos deux agents décident de se rendre sur la même case. Puisque rien ne contrôle leur exécution, ces dernières peuvent avoir pris la décision au même moment et aucun des deux ne sont au courant des intentions de l'autre *Agent*. Nos deux *Agents* se chevauchent donc, ce qui n'est pas un cas réalisable.



Pour pallier ce problème, nous utiliserons une *ConcurrentHashMap* (classe *Board*) qui associe à chaque agent la *Boxe* correspondant à sa position actuelle. Dès qu'un *Agent* veut se déplacer, il doit demander sa position à la *ConcurrentHashMap* qui gère de façon autonome l'aspect concurrentiel de notre programme. Nous n'avons donc pas à nous soucier des conflits entre *Agents* pour l'écriture dans notre *HashMap*.

Pour obtenir le chemin que chaque agent doit parcourir pour aller à sa position finale nous utiliserons l'algorithme A*. Cet algorithme permet la recherche d'un chemin optimal dans le graph des possibilités des mouvements.

Approche naïve

Présentation

Dans une première partie nous allons résoudre le problème sans approche cognitive. C'est-à-dire que nous ne faisons pas communiquer les agents entre eux. De la sorte, nous faisons simplement avancer à tour de rôle chaque agent vers sa position finale en utilisant le plus court chemin de l'algorithme A*.

Pseudo code de l'agent

```
Tant que l'agent n'est pas à sa position final
    Réaliser A* à partir de la position actuelle de l'agent
    Attribuer à l'agent la première direction indiqué par A*
    Si la direction est nulle
        Se déplacer vers la case vide la plus proche du point final
    Faire se déplacer l'agent dans la direction attribuée
```

Tests

Nous atteignons, lorsque nous ajoutons trop d'agents, la limite de l'approche autonome de chaque agent. Pour pallier ce problème il faudrait en effet que les agents communiquent pour que l'un libère sa place pour l'autre. Ils pourront alors débloquer la situation.

Nous avons réalisé des enregistrements vidéo afin de montrer le fonctionnement de notre programme.

Dans un premier temps nous avons pris un exemple simple de pions éparpillés sur tout le plateau de jeu.

Vidéo : [Approche naïve - test 1](#)

Maintenant réalisons un cas plus complexe illustrant les limites de notre approche. En effet nous réaliserons cercle de pions devant échanger de positions.

Vidéo : [Approche naïve - test 2](#)

Nous pouvons observer que le dernier pion, pion E, doit réaliser le tour du plateau de jeu pour rejoindre sa place définitive ce qui est chronophage. Dans une approche avec communication, ce dernier aurait pu communiquer avec ses voisins afin d'échanger de position et se rendre plus rapidement à sa position finale

Limites

Nous rencontrons des difficultés lorsque deux agents se bloquent mutuellement. Supposons l'exemple ci-dessous :

Nous avons ici deux agents *A* et *B*. Ces deux agents occupent respectivement la place finale de l'autre agent. Chacun des deux agents veut rejoindre sa position finale. Cependant cette position finale n'est pas libre car l'autre agent s'y trouve. Chaque agent estime alors qu'il ne peut pas se déplacer car la case où il veut se rendre n'est pas libre.

Pour que cette situation se débloque, il faudrait que l'un des agents se déplace et laisse sa place à l'autre agent. Cependant sans communication il n'est pas possible qu'un agent sache qu'il bloque un autre agent et se déplace. Nous rencontrons donc ici les limites de l'approche sans communication

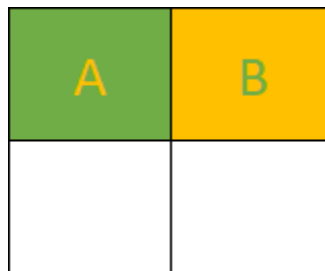


Schéma de deux agents bloquants respectivement la position finale de l'autre agent

Approche cognitive

Présentation

Pour pallier les limites rencontrées dans l'approche classique, nous décidons d'implémenter un système de communication.

Ce dernier se repose sur des boîtes aux lettres propres à chaque agent. De la sorte un agent peut vérifier s'il des agents l'ont contacté pour interagir avec, en regardant sa boîte aux lettres. Il peut aussi regarder s'il a obtenu des réponses à ses requêtes. Un agent peut bien-sûr choisir d'accepter ou non une requête d'un autre agent.

Pour ce faire nous implémentons une nouvelle classe d'agents, *CommunicativeAgent*. Cette classe reprend le fonctionnement de l'agent *NaiveAgent*. Cependant nous ajoutons en amont du déplacement naïf notre système de communication.

Pseudo code de l'agent

Tant que l'agent n'est pas à sa position finale

 L'agent regarde dans la boîte aux lettres s'il a du courrier

 Il a du courrier, il doit bouger, il lance donc un pile ou face ($p=0.7$)

 Si pile, l'agent bouge vers une case vide aléatoire

 Sinon

 L'agent prévient de son refus d'obtempérer

 Sinon

 L'agent regarde le dernier Agent qu'il a contacté pour savoir s'il a été refusé précédemment

 S'il n'a pas été accepté : c'est lui qui bouge vers une case vide aléatoire

 Fin du courrier

 L'agent regarde la meilleure direction vers laquelle il peut aller

 Si cette direction est entravée par un agent

 Il le contacte via la boîte aux lettres

 Sinon

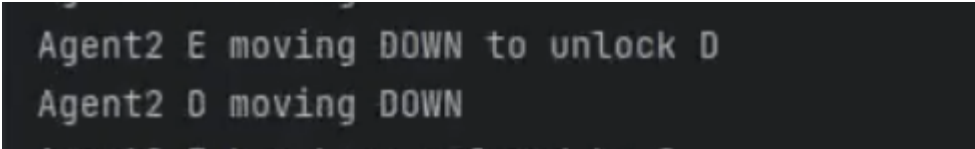
 Il bouge

Tests

Regardons en action le fonctionnement de notre nouvelle approche.

Vidéo : [Approche cognitive - test 1](#)

Dans cette vidéo nous pouvons voir des agents en train de coopérer. Plus précisément, nous pouvons voir l'agent E se déplacer pour débloquer le chemin de l'agent D. Nous retrouvons la preuve de cet échange en regardant le terminal de l'IDE.



```
Agent2 E moving DOWN to unlock D  
Agent2 D moving DOWN
```

Capture d'écran de l'échange entre l'agent E et l'agent D

Nous avons réalisé le même test complexe que dans l'approche sans communication. Regardons comment cette fois-ci les agents résolvent le problème.

[Approche cognitive - test 2](#)

Le terminal nous montre un grand nombre d'échanges entre les agents. Ces échanges s'observent car les agents réalisent moins de tours de plateau et échangent plus de positions. Nous pouvons voir que la case centrale sert de case d'échange pour les agents.

Difficultés rencontrées

Lors de la conception de cette partie du projet, nous avons eu du mal à imaginer comment chaque Agent pouvait communiquer avec un autre de manière indépendant au processus du jeu. Nous avons donc opté pour une « boîte aux lettres » générale et non pas une communication directe comme initialement pensée.

Toutefois, cette boîte aux lettres nous impose un bug d'affichage. En effet, l'affichage montre parfois le conflit entre deux agents sur une même case, alors que ce n'est jamais le cas d'après le code (vérification effectuée dans la méthode « `updateMap()` » de la classe *Board*).