**Interaction Multi-Agents**

Le but de ce projet est de développer un système Multi-Agents capable de résoudre un Puzzle (du jeu Taquin) selon une approche cognitive. Pour cela, nous représenterons chacune des cases de notre Puzzle par un agent en les numérotant de 1 à 15 et une case « nulle » qui prendra la valeur 0 (On cherchera en priorité à résoudre ce problème de Taquin sur une grille de taille 4\*4).

Nous allons maintenant expliquer l’approche que nous avons adopté afin de développer notre Intelligence Artificielle.

Nous avons utilisé l’algorithme A\* pour essayer de trouver la solution optimale au problème du Taquin. Pour se faire, nous avons créé une classe State qui représente un état de la grille et permet d’enregistrer son heuristique, son « coût » et son parent. La classe TaquinSolver qui permet de résoudre le Taquin en implémentant l’algorithme A\*.

L'algorithme A\* est un algorithme de recherche de chemin dans un graphe entre un [nœud](https://fr.wikipedia.org/wiki/N%C5%93ud_(math%C3%A9matiques)) initial et un nœud final. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. Ici le principe général est basé sur une fonction heuristique qui estime le coût restant pour atteindre l'état final à partir de chaque état actuel. La fonction heuristique estime le nombre de tuiles mal placées par rapport à l'état final.

La grille est générée de façon aléatoire (c’est à nous de spécifier sa taille lorsque l’on exécute le code) et elle sera affichée à chaque coup jusqu’à obtenir la solution.

Après la première implémentation de notre Intelligence Artificielle, les performances n’étaient pas assez satisfaisantes, il fallait plusieurs secondes pour trouver une solution pour une grille de taille 3\*3 et plusieurs dizaines de minutes pour une grille 4\*4. De plus, lorsque la grille n’a pas de solution, notre algorithme explore toutes les possibilités jusqu’à conclure qu’il n’y a pas de solution, ce qui rendait notre algorithme interminable, pour une grille 4\*4 il finissait par s’arrêter, faute de mémoire. Voici les différentes modifications que nous avons apporté à notre implémentation :

• Utilisation d’une HashMap (accès plus facile, recherche plus efficace des éléments) pour stocker les états parcourus. Ainsi on garantit de ne pas explorer plusieurs fois le même état, et ce de manière plus efficace donc gain de performance.

• Nous avons choisis d’utiliser une structure de données permettant d'ordonner les éléments en fonction de leur priorité. Dans notre cas, la priorité sera le coût total de chaque état du taquin.

• Nous avons ensuite changé le fonctionnement de notre heuristique, en choisissant de calculer à chaque état la distance de Manhattan. La distance de Manhattan est la somme des distances horizontales et verticales entre les positions actuelles des tuiles et leurs positions finales attendues. Cette heuristique est généralement plus précise et permet à l'algorithme de trouver plus rapidement des solutions.

L’algorithme cherche une solution en se basant sur l’algorithme de A\* en calculant à chaque état la distance de Manhattan. Lorsque l’on exécute le code, on nous demande de rentrer la taille de la grille, l’état initial généré automatiquement et aléatoirement est affiché puis l’algorithme cherche comment arriver à la solution.  
  
Si aucune solution n’est possible (une grille de Taquin peut ne pas avoir de solution), on nous renvoie : « Aucune solution trouvée » (Voir la vidéo ‘*3\*3PasSolution.mov’* associée).

Sinon, le retour sera le nombre de coups nécessaires pour arriver à la solution, ainsi que tous les coups effectués pour arriver à cette solution (Voir la vidéo ‘*3\*3Solution.mov’* associée).

Lorsque l’on choisit une grille de taille 4\*4, il faudra seulement quelques petites secondes (en général) à l’algorithme pour nous renvoyer la solution (Voir la vidéo ‘*4\*4Solution.mov’* associée).

Cependant, lorsque l’algorithme ne (semble pas) trouver de solution, le temps d’exécution est beaucoup trop long avec l’implémentation de cet algorithme, ainsi si l’on n’a pas la solution au bout de quelques secondes, on sait que le Taquin qui a été généré n’a pas de solution.

Une solution que nous proposons est donc d’ajouter un chronomètre lors de l’exécution de notre méthode solve qui, après un certain délai prédéfini dépassé (admettons 10 secondes), arrête le processus et admet que le jeu de Taquin n’a pas de solution (Voir ‘*4\*4PasSolution.mov’*).