Intelligence Artificielle : TP 2 (Algorithme A*)

Contexte :

Le but de ce TP est de programmer une intelligence artificielle permettant de résoudre le Taquin en utilisant l'algorithme A^*.

Le taquin est un jeu solitaire en forme de damier. Il est composé de 15 petits carreaux numérotés de 1 à 15 qui glissent dans un cadre prévu pour 16. Il consiste à remettre dans l'ordre les 15 carreaux à partir d'une configuration initiale quelconque.

Dans la figure 1 nous illustrons une version du jeu avec 8 cases. A gauche une situation de départ quelconque, et à droite la situation but désirée.

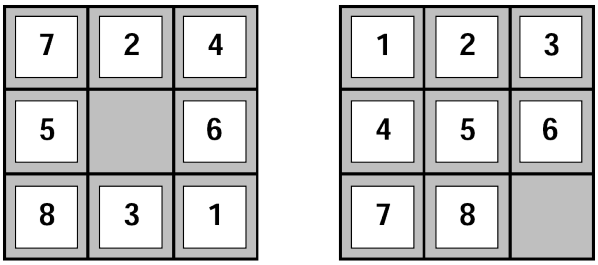


FIGURE 1 – Exemple du jeu du taquin en version 8.

Un mouvement du taquin consiste à faire glisser une case numérotée vers la case vide ; La case vide prend alors la place de la case déplacée. Pour rappel, l'évaluation $f(n)$ d'un noeud n est égale à $f(n) = g(n) + h(n)$ avec $g(n)$ le coût (nombre de mouvements ayant mené à l'état courant) et $h(n)$ l'estimation du coût restant au but.

Dans ce TP on va tester deux heuristiques différentes :

— $h1(n)$ = nombre de cases mal placées,
— $h2(n)$ = la somme des distances des cases par rapport à leurs positions cibles.

Par exemple, pour la grille initiale de la figure 1, $h1 = 1+0+1+1+1+0+1+1 = 6$ et $h2 = 4+0+3+3+1+0+2+1 = 14$

Implémentation

Votre projet sera constitué de 3 principales classes :

La classe Grille

Cette classe définit une grille du jeu. Elle est composée d'attributs et de méthodes suivantes :

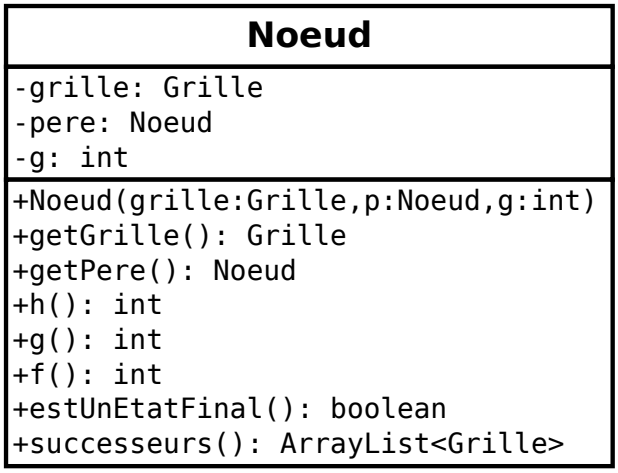
Grille
-grille: int[][] -taille: int -ligne0: int -colonne0: int
+Grille(g:int[][]) +getGrille(): int[][] +getTaille(): int +getLigne0(): int +getColonne0(): int +getValeur(i:int,j:int): int +copier(): int[][] +equals(y:Object): boolean +toString(): String

- **Attributs :**
  - *grille :* correspond à la grille du puzzle. Elle est implémentée avec une matrice carrée d'entiers (nombre de lignes = nombre de colonnes).
  - *taille :* représente la taille du puzzle. Par exemple, le puzzle de la figure 1 a une taille égale à 3.
  - *ligne0 :* représente l'indice de la ligne ou se trouve la case vide (la case 0).
  - *colonne0 :* représente l'indice de la colonne ou se trouve la case vide (la case 0). Dans la figure 1, la case vide se trouve initialement aux coordonnées $ligne0 = 1$ et $colonne0 = 1$
- **Méthodes :**
  - Le constructeur $Grille(g : int[][])$ initialise la grille de l'instance avec la grille g passée en paramètre.
  - Les méthodes $getGrille$, $getTaille$, $getLigne0$, $getColonne0$ sont de simples *getters* qui permettent de retourner les valeurs des attributs.
  - La méthode $getValeur$ retourne la valeur de la case qui se trouve à la ligne i et la colonne j.

— la méthode *copier*() retourne une copie de la grille de l'instance.
— Les méthodes *equals*(*Objecty*) et *toString*() doivent être implémentées.

La classe Noeud

Elle représente un noeud de l'arbre de recherche. Une instance *Noeud* est définie par une grille du jeu, une référence vers le noeud père et le nombre de mouvements g effectués depuis l'état initial (le noeud racine).



— **Attributs :**
  — *grille* représente la grille du jeu.
  — *pere* est la référence vers le noeud père dans l'arbre de recherche.
  — g est un champs qui sauvegarde le nombre de mouvements effectués depuis la racine de l'arbre.
— **Méthodes :**
  — Le constructeur de la classe initialise le noeud avec les valeurs passées en paramètres
  — La méthode *getGrille* retourne la grille du puzzle.
  — La méthode *getPere* retourne la référence du noeud pere.
  — La fonction $h()$ calcule et retourne l'estimation du nombre de mouvements restant pour arriver à l'état final. Pour un premier test, $h = h1$ c-a-d le nombre de cases mal placées.
  — La méthode $g()$ retourne simplement la valeur de l'attribut g.
  — La méthode $f()$ retourne l'évaluation du noeud qui sera utilisée par l'algorithme A^* : $f = g() + h()$

— La méthode $estUnEtatFinal()$ retourne vrai si le noeud contient la configuration finale (noeud but).
— La méthode $successeurs()$ retourne une liste de tous les noeuds successeurs obtenus en déplaçant la case vide vers le haut, le bas, la gauche et la droite. Le nombre de successeurs varie entre 2 et 4 selon la position de la case vide.

La classe Solveur

Cette classe implémente l'algorithme A^* pour résoudre une grille quelconque du Taquin. L'algorithme A^* utilise deux listes pour explorer les noeuds de l'arbre de recherche : $liste\_ouverte$ et $liste\_fermee$

Solveur
-liste_noeuds_ouverts: ArrayList<Noeud> -liste_noeuds_fermes: ArrayList<Noeud>
+chargerFichier(nomFichier:String): Grille +algoAstar(initial:Grille): Noeud +cheminComplet(noeudFinal:Noeud): ArrayList<Grille> +static main(args:String[])

— **Attributs :**
  — La liste $liste\_ouverte$
  — La liste $liste\_fermee$
— **Méthodes :**
  — La méthode $chargerFichier()$ permet de lire un puzzle à partir d'un fichier passé en paramètre et retourne la grille initial du jeu. Penser à utiliser les puzzles fournis dans le dossier **"puzzles"**. Vous pouvez débugger votre algorithme en utilisant la grille : **"puzzle04.txt"**.
  — La méthode $algoAstar()$ implémente l'algorithme A^* pour résoudre la grille passée en paramètre et retourne le noeud correspondant à la grille résolue sinon elle retourne *null* (certaines grilles dans le dossier n'ont pas de solution).
  — La méthode $cheminComplet()$ retourne toutes les étapes (grilles intermédiaires) de la solution. Elle doit afficher également le nombre de mouvements de la solution trouvée.
  — La méthode *main* sera consacrée au test de votre algorithme sur les différents puzzles fournis.

Description détaillée des étapes d'implémentation de l'algorithme A^* :

La première liste *liste_ouverte* contient tous les noeuds étudiés. Dès que l'algorithme va se pencher sur un noeud du graphe, il passera dans la liste ouverte (sauf s'il y est déjà). La seconde liste $liste\_fermee$, contiendra tous les noeuds qui ont été considérés comme faisant partie du chemin de la solution. Avant de passer dans la liste fermée, un noeud doit d'abord passer dans la liste ouverte. En effet, il doit d'abord être étudié avant d'être jugé comme bon. Ci dessous les étapes pour implémenter l'algorithme A^*. Vous pouvez ajouter à la classe Solveur toute autre méthode nécessaire à l'implémentation.

1. On initialise un noeud avec la grille initiale et on le met dans la liste ouverte *liste_ouverte*.
2. On cherche le meilleur noeud de la liste ouverte. Il s'agit du noeud qui a la valeur $f = g + h$ minimale. On le note $noeudCourant$.
3. On ajoute $noeudCourant$ à la liste fermée et on le retire de la liste ouverte.
4. On génère tous les noeuds successeurs du $noeudCourant$ en déplaçant la case vide.
5. Pour chaque noeud successeur s **non présent dans la liste fermée.**
   - Si s n'est pas dans la liste ouverte, on le rajoute à cette liste.
   - Sinon, s'il existe dans la liste ouverte un noeud n avec une grille identique à celle de s, alors on remplace n par s dans la liste ouverte si et seulement si l'évaluation du noeud s est meilleure que celle de n : $s.f() < n.f()$.
6. On réitère à partir de l'étape 2 jusqu'à ce qu'on trouve le noeud but (la grille finale) ou bien la liste ouverte est vide (pas de solution).