RAPPORT DE PROJET

Résolution du Jeu Sokoban avec l'Algorithme A*

Réalisé par :

Ibrahim Zaryouh

Encadré par :

M'hamed Ait Kbir

Cycle Ingénieur – Logistique et Système Intelligent

INFORMATIONS GÉNÉRALES

Objectif : Implémenter un résolveur de puzzle Sokoban utilisant l'algorithme A* (Best-First Search)

Contexte : Sokoban est un jeu de puzzle inventé au Japon en 1982 où le joueur incarne un manutentionnaire (gardien d'entrepôt) devant pousser des caisses vers des emplacements de stockage.

Méthode : Adaptation de l'algorithme A* vu en classe au contexte spécifique du jeu Sokoban avec heuristique personnalisée et détection de deadlock.

DESCRIPTION DU JEU

Symboles Utilisés

— `@` : Personnage (Sokoban)

— `$` : Caisse

— `|` : $Mur(obstacle infranchissable)$

- `? : Espace vide`
- `T : Cible (Target/Emplacement de stockage)`
- `* : Caisse sur cible (succès partiel)`
- `+ : Personnage sur cible`

Règles de Base

— **Déplacement :** Le personnage se déplace dans les 4 directions (Haut, Bas, Gauche, Droite)

— **Poussée :** Une caisse peut être poussée si la case derrière est libre (ou T)

— **Interdictions :**
  — Impossible de tirer une caisse
  — Impossible de pousser deux caisses simultanément
  — Blocage contre un mur

— **Condition de Victoire :** Toutes les caisses sont sur des cibles (caractère $*$)

DÉMARCHE D'IMPLÉMENTATION

1. Architecture Globale

Le programme est structuré en **2 fichiers Java** :

SokobanSolver.java (Classe principale)

Contient :

— **Classe interne** `State` **:** Représente un état du puzzle avec :
  — `grid[][]` : Configuration actuelle de la grille
  — `g` : Coût réel (nombre de poussées effectuées)
  — `f` : Coût estimé total = `g + h(n)`
  — `parent` : État précédent (pour reconstruction du chemin)
  — `playerX, playerY` : Position du joueur
  — `List<boxes>` : Liste des positions des caisses
— **Classe** `SokobanSolver` **:** Moteur de recherche A* avec :
  — Algorithme A* complet (Open set, Closed set)
  — Génération des successeurs (mouvements valides)
  — Gestion d'états et détection des doublons

SokobanAStarSearch.java (Point d'entrée)

Contient :

— Deux grilles de test (10x10)
— Méthode main orchestrant les deux résolutions
— Affichage des résultats

2. Fonction Heuristique (Partie Critique)

L'heuristique `h(n)` est composée de **3 éléments** :

$$h(n) = \text{Distance(caisses} \rightarrow \text{cibles)} + \text{Distance(joueur} \rightarrow \text{caisse la plus proche)}$$

a) Détection de Deadlock

```
private boolean isCornerDeadlock(int x, int y) {
    boolean wL = isWall(x - 1, y);
    boolean wR = isWall(x + 1, y);
    boolean wU = isWall(x, y - 1);
    boolean wD = isWall(x, y + 1);
}
```

b) Assignation Optimale Caisses $\rightarrow$ Cibles

$$\text{Distance totale} = \min_{\text{permutations}} \sum \text{distance\_caisse\_i à cible\_j}$$

c) Distance Joueur $\rightarrow$ Caisse Proximale

Distance Manhattan du joueur à la caisse la plus proche (heuristique admissible).

ALGORITHME A* IMPLÉMENTÉ

```
1. Initialize: Open = { tat  initial}, Closed = {}
2. WHILE Open        :
   a. current = Pop(Open)
   b. IF current     Closed THEN continue
   c. Closed.add(current)
   d. IF current.isGoal() THEN return chemin
   e. FOR EACH neighbor:
      - IF neighbor     Closed THEN
        - Calcule h(neighbor)
        - f = g + h
        - Add neighbor    Open
```

Optimisation : Seules les poussées augmentent le coût g.

RÉSULTATS EXPÉRIMENTAUX

GRILLE 1 :

```
--- Grille 1 ---
Grille initiale:
##########
#????????#
#?##?###??#
#?$?T?$??#
#?#?@?#??#
#?$?T?$??#
#?##?###??#
#??T??T??#
#????????#
##########
But trouvé.
Nombre de poussées: 15
Chemin optimal: [U, U, U, L, L, L, D, D, R, R, D, R, D, D, D, R, R, R, U, U, L, L, U,
 U, R, L, D, D, R, R, U, R, U, U, L, D, R, D, D, L, L, L, U, L, D, D, U, U, R, U, R,
R, D, D, D, R, D, L, D, L, L, U, L, D, L, L, U, L, U, U, R, R]
Temps: 1232 ms
N?uds explorés: 61648
```

GRILLE 2 :