Assignment 1
Modeling and solving the Sokoban Puzzle

Sokoban (Pushbox) is a classic puzzle game, originating from Japan and dating back to the early 1980s. In this game, a warehouse keeper (the player) moves around a warehouse, represented by a grid of squares. These squares are of five types: PLAYER, WALL, EMPTY SPACE, BOX, TARGET SPACE (see Figure 1). The goal of the game is to place all the boxes on the target spaces.

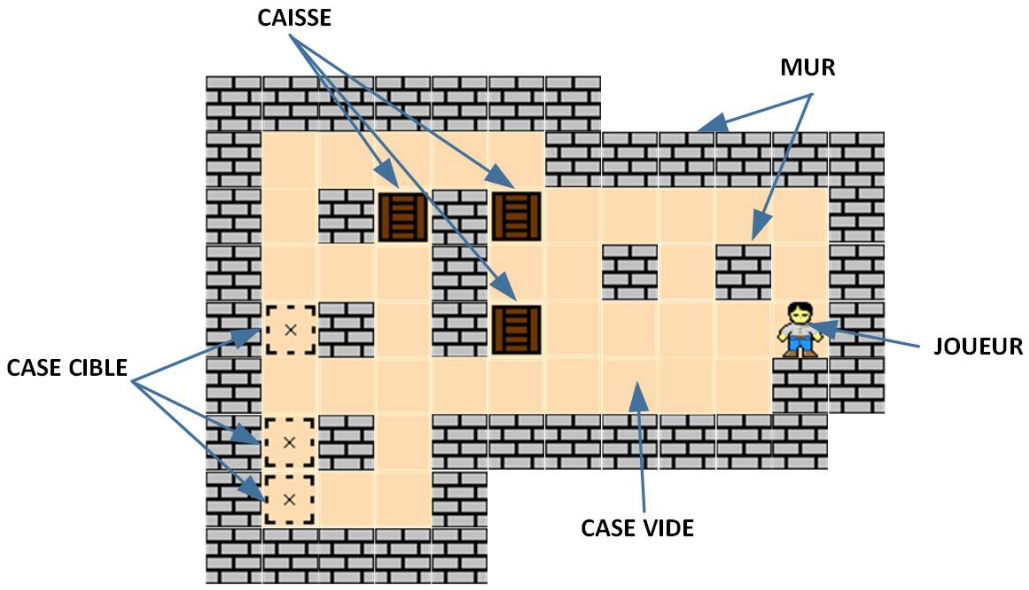

Figure 1 Example (1) of a Sokoban board

The rules of the game are very simple:

- The structure of the warehouse and the number of boxes vary from one level to another.
- The warehouse in which the boxes must be placed rarely has the shape of a simple rectangle; it is more of a maze, more or less complicated. There are walls on all sides of this maze.
- There can be one or more boxes, and there are always as many target spaces as there are boxes.
- The game has only one player, who can move in the four cardinal directions.
- The player is only allowed to move to an unoccupied space (one that is not a wall) or push a box and move onto the freed space. They are not allowed to pull a box or move over it.
- To push a box, the space adjacent to the box in the direction of the push must be free. The player cannot push two boxes at the same time.

- The player wins when all the boxes are on the target spaces, in any order. There are no boxes assigned to specific target points.

The objective of the project is first to model the game, then implement a solver based on various search algorithms, and finally create a GUI for this game.

A. Problem formulation:

The first step in this assignment is to propose a formulation for the Sokoban game, and implement it in Python.

1. Modeling the state:

To model a state, the following instructions must be followed:

- The game is presented in the form of a two-dimensional grid. We need to be able to represent the following elements: walls, empty spaces, and target spaces, which are static elements, as well as the player and the boxes, which are dynamic elements. In this grid, we can have squares with the following configurations (see Figure 2):

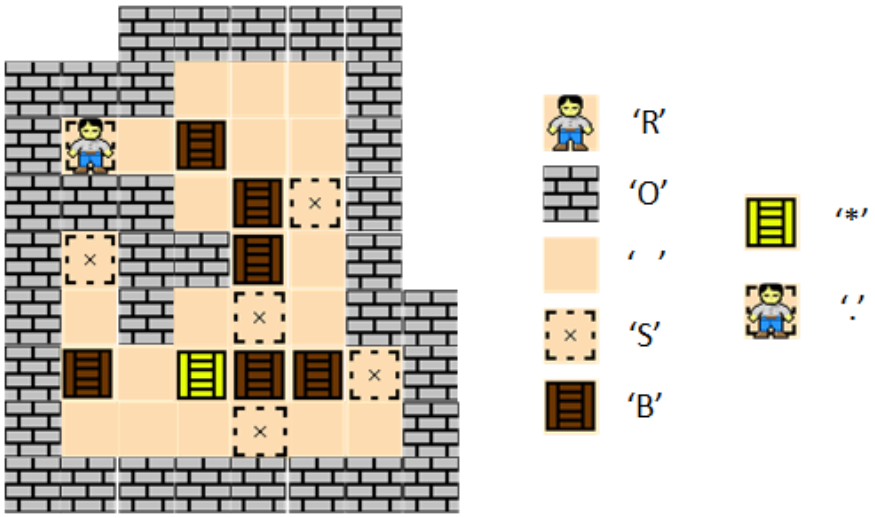

Figure 2 Configuration of a Sokoban board

- Player (Robot): represented by 'R'
- Wall (Obstacle): represented by 'O'
- Empty space: represented by ' '
- Target space (Storage): represented by 'S'
- Box (Block): represented by 'B'
- Player on a target space: '.'
- Box on a target space: '*'

- To move from one state to another, the player must move within the grid. Four movements are allowed:

- UP: (-1, 0)
- DOWN: (1, 0)
- LEFT: (0, -1)
- RIGHT: (0, 1)

- For a move to be valid, it must follow the game's rules (see the game description). The goal state is reached when all the boxes are on the target spaces.

To model a state in the Sokoban game, we define the class **SokobanPuzzle** with the following functions:

- ✧ *init()*: This function creates a state.
- ✧ *isGoal( )*: This function checks if a state is a goal.
- ✧ *successorFunction( )*: This function generates pairs of (*action*, *successor*) representing all possible moves that can be applied on the board.

2. Modeling the node:

The **Node** class should include the following components:

- ✧ *state*: An instance of the **SokobanPuzzle** class representing the state;
- ✧ *parent*: A reference to the parent node;
- ✧ *action*: The action applied to generate the current node from the parent node;
- ✧ *g*: The path cost, where the step cost *c* is always equal to 1;
- ✧ *f*: The fitness function used in the A* algorithm;

In the **Node** class, you should also provide definitions for the following functions:

- ✧ *getPath ( )*: This function returns a list or sequence of states representing the path from the initial state to the goal state of the solution;
- ✧ *getSolution ( )*: This function returns a list or sequence of actions applied to the initial state to reach the goal state;
- ✧ *setF( )*: This function calculates the fitness function *f* based on a heuristic.

B. Searching for a solution:

The next step in this assignment is to solve the Sokoban puzzle using two search algorithms: Breadth-First Search (BFS) and A* search. In the BFS algorithm, we'll aim to find a solution straightforwardly, while in the A* algorithm, we'll explore various heuristics to find the most efficient one.

1. **Implement the BFS algorithm:** Utilize the provided pseudocode from the course to create an implementation of the BFS algorithm.
2. **Implement the A* algorithm:** Develop an implementation of the A* algorithm using the following two heuristics:

h1: The number of boxes that the player has not yet placed in the target spaces (Equation 1).

$$h1(n) = nb\_left\_blocks(n) \quad (1)$$

h2: In this heuristic, we will add to heuristic h1 an estimate of the number of pushes required to move each box to the nearest target.
To estimate the number of pushes needed to move a box b to a target s, we will calculate the Manhattan distance between the box b and the target s. This distance is given by the following equation (Equation 2):

$$Manhattan\_distance_{b,s} = |x_b - x_s| + |y_b - y_s| \quad (2)$$

Where (x_b, y_b) represents the position of the box b and (x_s, y_s) represents the position of the target space s. The Manhattan distance between a box and a target is illustrated in Figure 3 (with the two red lines).

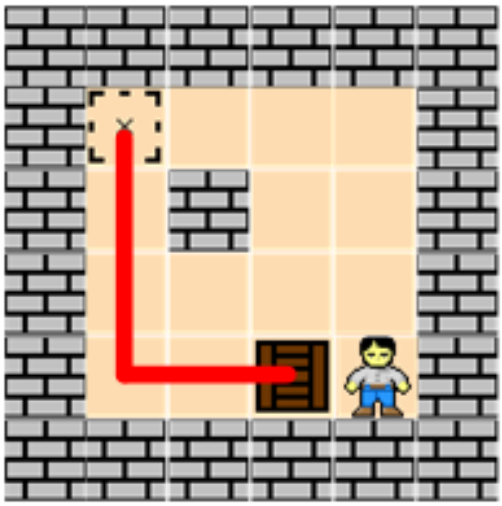

Figure 3 Illustration of the Manhattan distance between a box and a target

Thus, heuristic 2 is given by the following equation (Equation 3):

$$h2(n) = 2 * nb\_left\_blocks(n) + \sum_{b \in B} \min_{s \in S} Manhattan\_distance_{b,s}(n) \quad (4)$$

Here, we have multiplied the number of boxes that the player has not yet placed in the target spaces by 2 to give it more weight in the heuristic.

3. **Propose a third heuristic for A*:** Suggest a third heuristic that can be used in the A* algorithm to improve its performance
4. **Test both algorithms:** Evaluate the performance of both algorithms using the examples provided in Figure 4. Assess the efficiency of the search algorithms by determining how many steps it takes for each algorithm to find a solution to the puzzle.

C. Game interface:

Create an interface in Pygame that displays a simulation of the solution found, its cost, and the number of steps taken by the different search algorithms.

This interface will provide a visual representation of the solution process, allowing users to see the simulation of how the puzzle is solved, along with other relevant information.

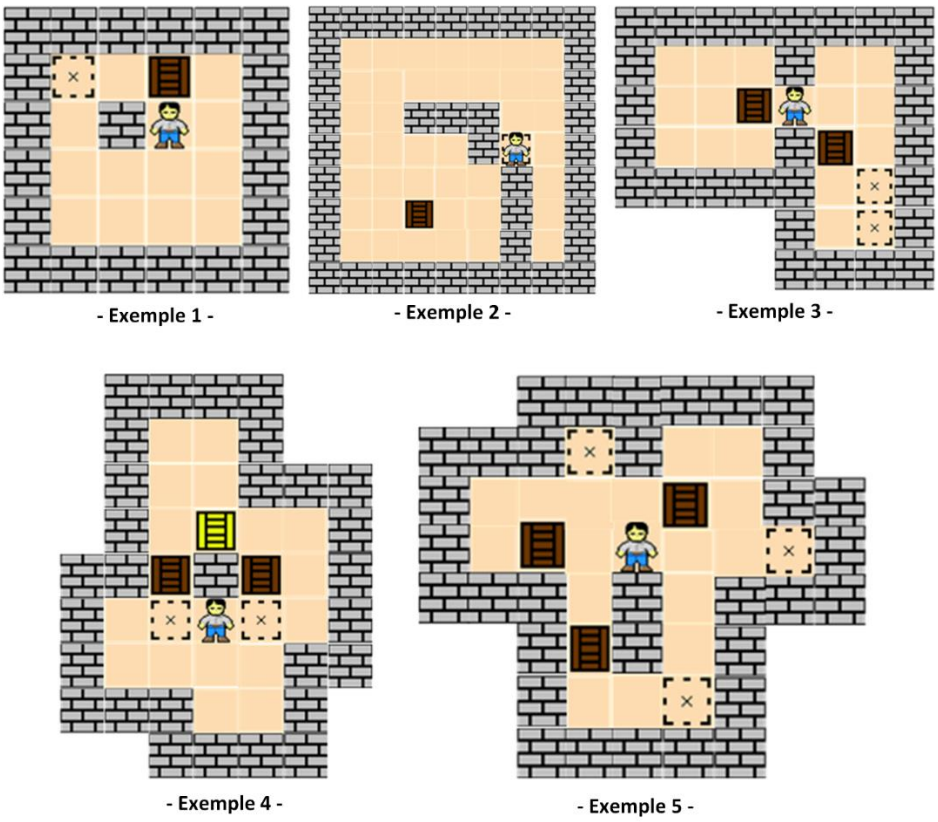

Figure 4 Test examples

D. Sokoban's deadlock

Sokoban is a game in which we can encounter what we call deadlocks. A deadlock is a state in the search tree from which no solution state can be found. In general, a deadlock is caused by a situation where it is impossible to push all the boxes onto their storages.

If we traverse the search tree while ignoring these deadlocks, it is highly likely that we will encounter a large number of subtrees doomed never to reach a solution. The size of the tree will then increase significantly. This is why deadlock detection is an essential step in the game of Sokoban.

- **Corner deadlock:** A corner deadlock occurs when a box is stuck on a tile that is not a storage and is surrounded by two walls, forming a corner. Examples of a corner deadlock are shown in Figure 5.

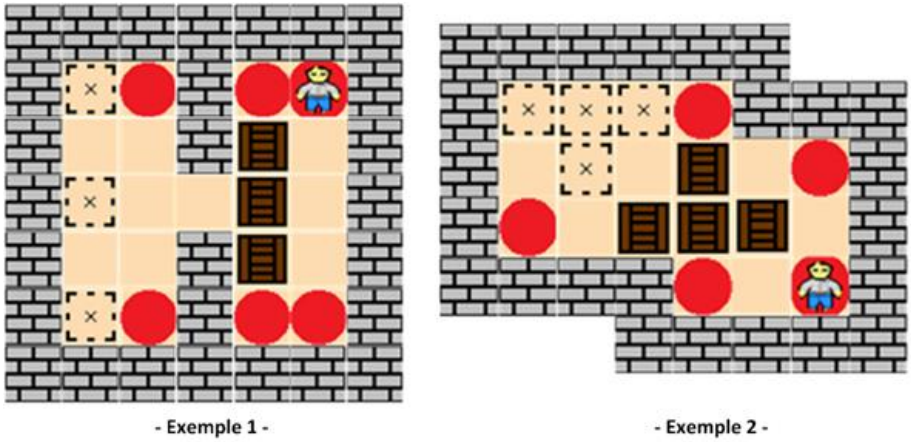

Figure 5 The positions in red cause a corner deadlock if a crate is located there.

- **Line deadlock:** A line deadlock occurs when a crate is positioned against a wall and is unable to move away toward a target. Examples of a line deadlock are shown in Figure 6. Line deadlocks are caused by walls that lack an escape route, meaning there is no passage through which the player can push the crate in the opposite direction. The method used to detect problematic lines involves using corner deadlock positions. By connecting one of these positions to another via a straight line, if one side of this line is entirely composed of walls, we can consider all the positions along that line as deadlocks.

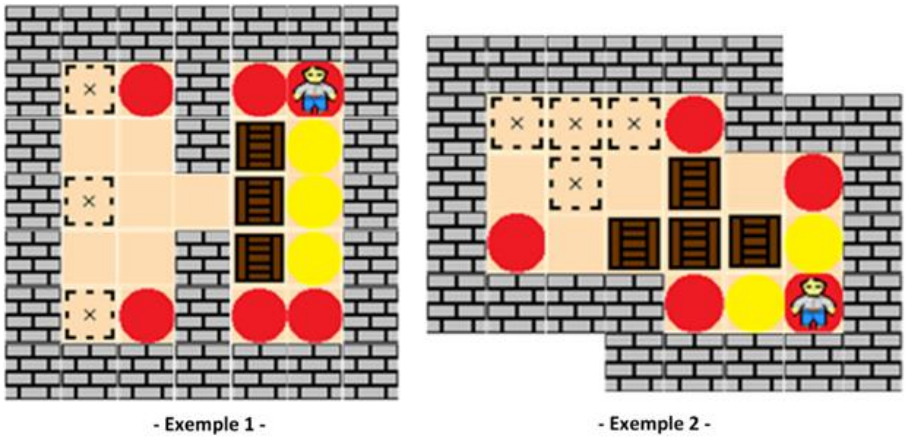

Figure 6 The positions in yellow cause a line deadlock if a crate is located there.

1. Write a function that determines, for a given Sokoban state, if it contains boxes on deadlock positions (corner or line deadlocks).
2. Use the previous function to improve the A* search algorithm. This function will help the A* algorithm avoid developing subtrees that are doomed to never reach a solution.
3. Perform tests of the improved A* algorithm on the examples in Figure 7 using heuristic 3. Provide the solution returned for each example, as well as the number of iterations required to reach this solution. Compare the results obtained with previous search algorithms.

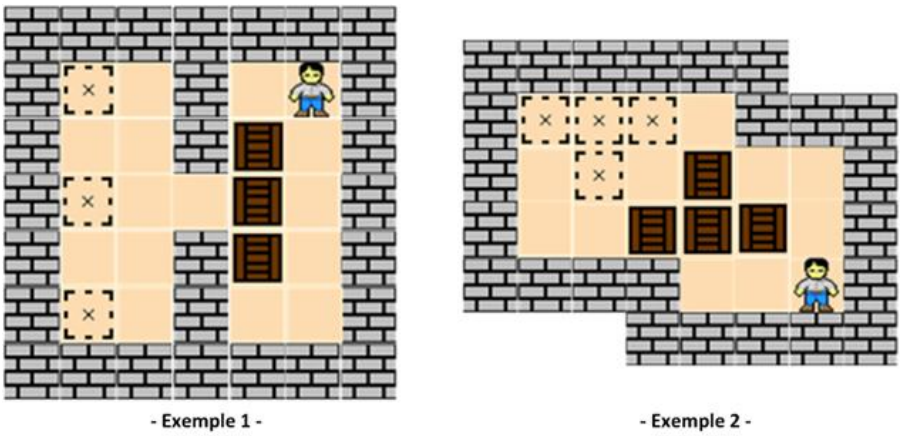

Figure 7 Test examples