

Project 1

Modeling and Solving the Rush Hour puzzle

The Rush Hour puzzle is a sliding block game where the objective is to maneuver vehicles on a game board to free the red ‘X’ car from a traffic jam. The goal is to guide the red car out of the parking bay through the exit on the right-hand side of the board. To achieve this, the player has to slide the other vehicles out of its way, moving them either vertically or horizontally, depending on their orientation. The initial setup of the game features various vehicles, arranged according to a given configuration (See Figure 1).

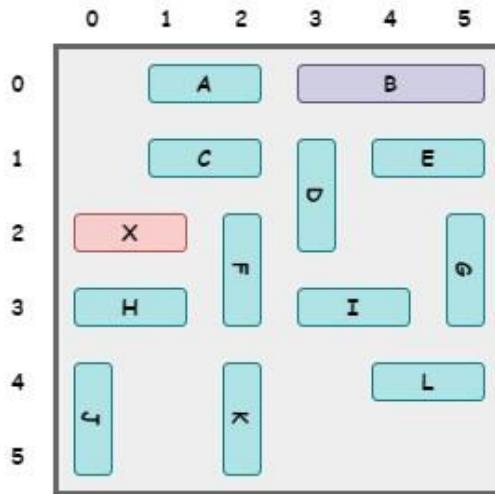


Figure 1 Example (1) of a Rush Hour board

The initial configuration of the game is determined by a set of vehicles, with each vehicle is characterized by: an *id*, a *position*, an *orientation* (horizontal or vertical), and a *length* (where cars occupy 2 squares and trucks occupy 3 squares). The configuration shown in Figure 1 is represented as follows: {‘X02H2’, ‘A10H2’, ‘B30H3’, ‘C11H2’, ‘D31V2’, ‘E41H2’, ‘F22V2’, ‘G52V2’, ‘H03H2’, ‘I33H2’, ‘J04V2’, ‘K24V2’, ‘L44H2’}.

Rules of the game:

- ❖ Each vehicle can only move in the direction it is facing, either horizontally or vertically, and can move one square at a time, either forward or backward.
- ❖ To move a vehicle, the target position must be unoccupied, free of any other vehicles or walls.

The solution to any Rush Hour puzzle is the sequence of actions represented as pairs (*vehicle id, move direction*) that are applied to achieve the goal: moving the red ‘X’ car to the exit, as shown in Figure 2:

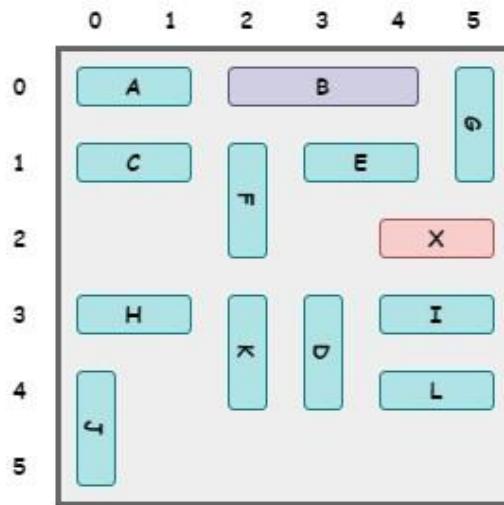


Figure 2 Goal of example (1)

A. Problem formulation:

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

1. Modeling the state:

To model a state in the Rush Hour game, we define the class **RushHourPuzzle** with the following attributes:

- ✧ *board_height, board_width*: Represent the height and width of the game board.
- ✧ *vehicles*: A list of vehicles in the game. Each vehicle is defined by five attributes (*id, x, y, orientation, length*)
- ✧ *walls*: Represents a list of walls if they exist in the puzzle;
- ✧ *board*: A matrix of characters, where an empty square is represented by a blank character. Vehicles are modeled as letters, with 'X' reserved for the red car.

In the state class, you should also define the following functions:

- ✧ *setVehicles()*: This function generates the list of vehicles from the CSV file. It also sets the values of *board_height* and *board_width* (which are given in the first line of the CSV file), and creates the list of walls if they exist. In the CSV file, the character '#' represents a wall, followed by its position.
- ✧ *setBoard()*: This function generates the game board from the list of vehicles and walls.
- ✧ *isGoal()*: This function checks if a state is a goal, meaning the red car is in the position (*board_width-2, board_height/2-1*).
- ✧ *successorFunction()*: This function generates pairs of (*action, successor*) representing all possible moves that can be applied on the different vehicles.

2. Modeling the node:

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

- ✧ *state*: An instance of the **RushHourPuzzle** 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 function h .

B. Searching for a solution:

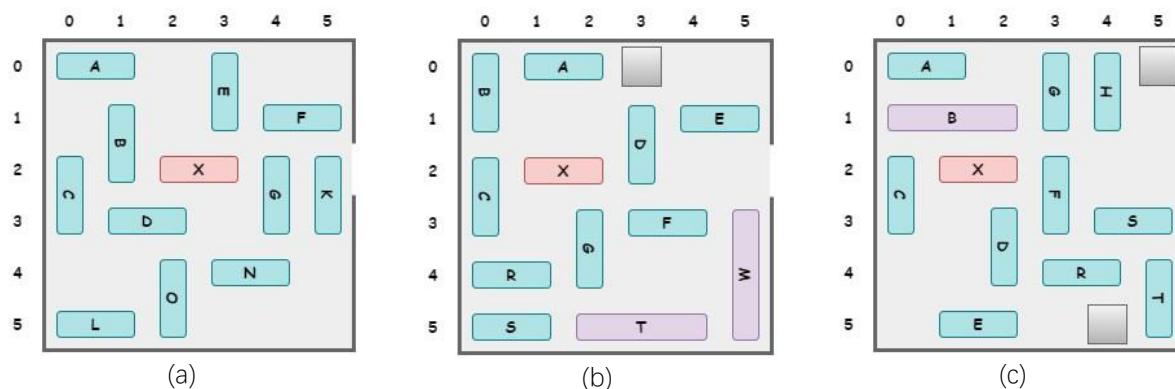
The next step in this assignment is to solve the Rush Hour puzzle using two search algorithms: Breadth-First Search (BFS) and A* search. In the BFS algorithm, we'll aim to find a solution straightforwardly, while in the A* algorithm, we'll explore various heuristics to find the most efficient one.

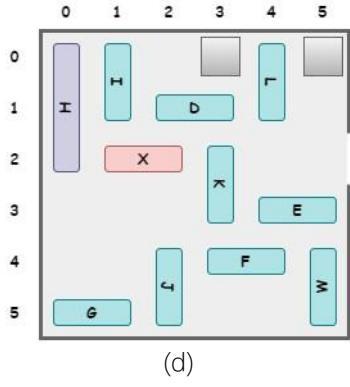
1. **Implement the BFS algorithm:** Utilize the provided pseudocode from the course to create an implementation of the BFS algorithm.
2. **Implement the A* algorithm:** Develop an implementation of the A* algorithm using the following two heuristics:
 - $h1$: The distance from the target vehicle (the red car) to the exit.
 - $h2$: $h1 +$ the count of vehicles blocking the path to the exit.
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 3. 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:

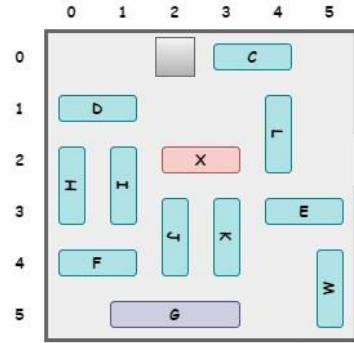
Create an interface in Pygame that displays a simulation of the solution found, its cost, and the number of steps taken by the different search algorithms.

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.

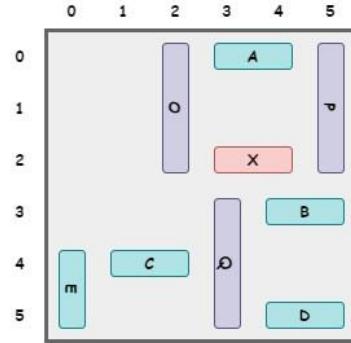




(d)



(e)



(f)

Figure 3 Test examples