Introduction

In this project, we implement an AI solver for the classic board game "Rush Hour" with A* algorithm. Our goal is to explore different strategies to solve this game, so we came up with some heuristic functions and tried out different combinations of them. We did not refer to any code of others because the difficulty of programming is rather easy. We focus on designing and comparing the heuristics.

Rush Hour

The goal of the game is to guide the target car out of a traffic jam and arrive at the exit by moving cars on the puzzle board. Typically, the size of the board is 6*6 and a car is 1*2 or 1*3. And there may be some obstacles (walls) on the board.
Check out the online version for demonstration (by Michael Fogleman)

Rules

1. Cars cannot go out of the board, bump into each other or hit a wall
2. Cars lying vertically can move only upwards or downwards
3. Cars lying horizontally can move only leftwards or rightwards

Implementation

The implementation is composed of 2 classes - **Car** and **Board**.

Car

A car is represented by the following attributes and function

car_num: id of the car
isvertical: True if the car lies on the board vertically, false otherwise
size: The size of the car is 1*size
x: The position on the board (row)
y: The position on the bard (column)
car_move: Function that can move the car in different directions (by 1 cell)

Board

A car is represented by the following attributes and functions

size: The width and height of the board
exit: Exit is on the right side of board[2][5], so we set this to (2,5)
emptySymbol: It represents an empty cell on the board - “o”
wallToken: It represents a cell occupied by a wall on the board - “x”
board: A 2D array that describes the board
cars: A list that stores the cars (**Car** objects) on the board
stateAfterMove: Function that returns a new **Board** object after a given move
expand: Function that returns all valid moves on the current board (**Board** objects)

A	B ~ Z	o	x
target car	other cars	empty cell	wall

```
--------
|oBBBox|
|DDDEEK|
|ooIAAK|
|HoIooo|
|HoIJFF|
|HGGJoo|
--------
```

Algorithm Detail

A* search algorithm

Since Rush Hour puzzle is PSPACE-complete problem, we use A* search algorithm with different heuristics to solve instances of this game.

We defined a class **StateNode** to keep the information of each possible move.

f: lowest estimated total cost = heuristic + depth
depth: current move cost (+1 in every step)
state: current game board
path: steps to reach the current state

Moving a car in any valid direction by 1 cell leads to a cost of 1.

```
def Astar(start_board, heuristic):
  initial visited as a list
  initial node_list as a PriorityQueue

  depth = 0
  f = heuristic(start_board) + depth

  push StateNode(f, depth, start_board, path) to node_list
  push start_board to visited

  while node_list is not empty:
    N = node_list.pop() # having smallest f

    if N is solved:
      return (N.path)

    for B in (every car, every valid direction):
      if B not in visited:
        g = N.depth + 1
        f = heuristic(B) + g
        push StateNode(f, g, B, new_path) to node_list
        push B to visited
```

Heuristic Functions

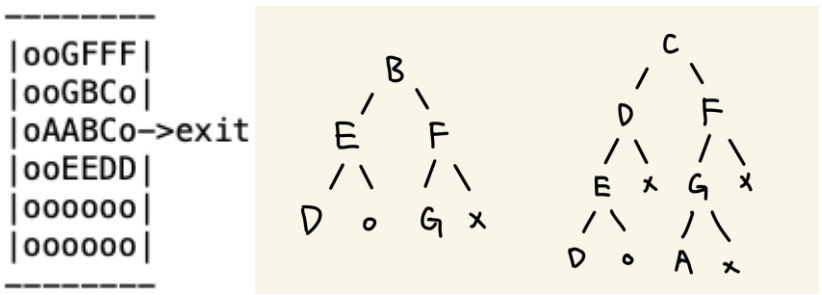
- **Manhattan**
  The distance between the target car and exit.

- **h2**
  The number of cars that block the path between the target car and exit.

- **h3**
  This heuristic takes a step further from **h2**. If a car (B~Z) blocks the target car (A) directly from the exit and this car (B~Z) cannot move immediately, we add 1 to cnt. The result is cnt + the number of blocking cars

- **h4**
  This is an upgraded version of **h3**, because adding 1 may not be close enough to the actual cost. We want to know how terrible the traffic jam is, so another function **blockDepth** (see below for more details) is defined. Instead of adding 1, we add the block depth of the car (the returned value of **blockDepth**).

  [blockDepth]
  Take the board below as example, we can see A is blocked by B and C, B is blocked by E and F, C is block by D and F, and so on. Our goal is to find the blockDepth of B and C. Using binary tree to describe this situation, we define the the blockDepth of the root node is the shortest path from the root to ‘o’.



  So, in this case, the blockDepth of B and C are 2 and 3 respectively. And **h4** will return *# cars blocking A + blockDepth(B) + blockDepth(C) = 2 + 2 + 3 = 7*

- **h5**
  This heuristic finds cars that block the target car and calculate their distance to the closer border. Then return the sum of these distance + the number of blocking cars.
  Note that if the blocking car is vertical and its size is 3, we will calculate the distance to bottom border. Because if this car doesn't go down, the target car can't pass it

Consistency of Heuristic Functions

For all the proofs below:
n denotes the current state,
n' denotes an arbitrary successor of **n**
a denotes the action it takes to go from **n** to **n'**

- **Manhattan**

  Depend on the target car movement (right, left, stall)
  h(n') = h(n) – 1 or **h(n) + 1** or **h(h)**
  cost(n, a, n') + **h(n')** **= [1 + h(n) - 1]** or **[1 + h(n) + 1]** or **[h(n)]** ≥ **h(n)**
  Thus, this heuristic is consistent, it can find the optimal solution for the game.

- **h2 & h3**

  They are all consistent, which can be proved by:
  h(n') = h(n) + 1 (a new car blocks the target car)
  or = **h(n) - 1** (a car that blocks the target car can move away or already moved)
  or = **h(n)** (no car moves between target car and exit)
  cost(n, a, n') + **h(n')** **= [1 + h(n) + 1]** or **[1 + h(n) – 1]** or **[h(n)]** ≥ **h(n)**
  Thus, these heuristics are consistent, they can find the optimal solution for the game.

  Besides, (**Manhattan + h2**) and (**Manhattan + h3**) are also consistent.
  Because in each step, only one car can move, Manhattan focus on the target car while h2, h3 focus on the other cars, they don't change simultaneously in the same step.

- **h4**

  This heuristic is not consistent because some situations will violate the triangle inequality. See the example below: (moving car E leftward by 1)

  Let **n** be the board on the left, **n'** be the board on the right.
  cost(n, a, n') + **h4(n')**
  = 1 + **h4(n')**
  = 1 + 2 + **blockDepth(n', B)** + **blockDepth(n', C)**
  = 1 + 2 + 1 + 2 = 6
  < h(n) = 7

```
--------      --------
|ooGFFF|      |ooGFFF|
|ooGBCo|      |ooGBCo|
|oAABCo->exit |oAABCo->exit
|ooEEDD|      |oEEoDD|
|oooooo|      |oooooo|
|oooooo|      |oooooo|
--------      --------
```

  As it is not consistent, when we implement A* with **h4, it** might end up with a suboptimal solution (possible not the minimal steps to solve the game)

- **h5**

  This heuristic is not consistent because some situations will violate the triangle inequality. See the example below: (moving car **B** downward by 1)

  Let **n** be the board on the left, **n'** be the board on the right.
  cost(n, a, n') + h5(n')
  = 1 + **h5(n')**
  = 1 + 3
  < h(n) = 5

```
________        ________
|ooGFFF|        |ooGFFF|
|ooGoCo|        |ooGoCo|
|oAABCo->exit   |oAAoCo->exit
|oEEBDD|        |oEEBDD|
|oooooo|        |oooBoo|
|oooooo|        |oooooo|
--------        --------
```

  As it is not consistent, when we implement A* with **h5, it** might end up with a suboptimal solution (possible not the minimal steps to solve the game)

Result

We prepare several problems in demo.txt for testing, including both easy and hard tasks. As for the results, we provide two versions source code to demonstrate the game solving process.

1. **rush_hour**
In this version, we first print out the unsolved game board in 2D matrix format, and then directly output every step of the solving process.

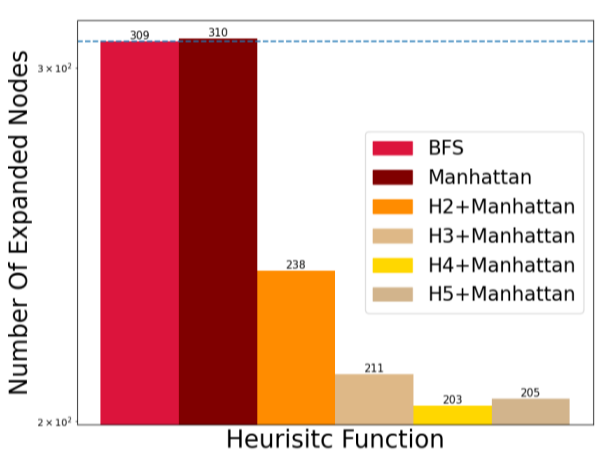
2. **rush_hour_gui (GUI version)**
In this version, we use **pygame** to visualize every step of the solving process on the game board. The process doesn't terminate until the red car arrives at the exit.

Please install pygame before running this code

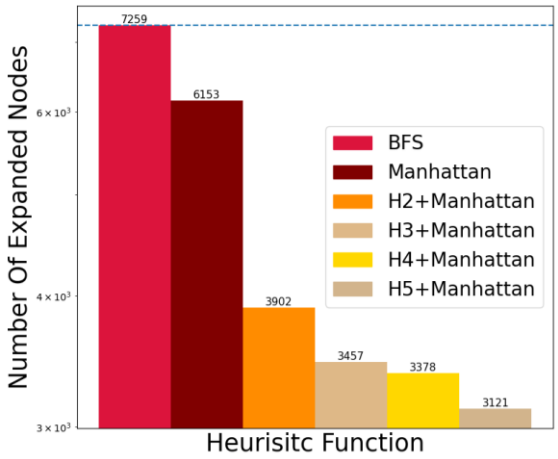
```
pip install pygame
```

- For more details, please refer to our ***GitHub repo***

Experiment Result



(Easy)



(Hard)

In this project, we implement A* algorithm with Manhattan, H2+Manhattan, H3+Manhattan, H4+Manhattan, H5+Manhattan and compare these heuristic functions with BFS. We judged the performance by the number of nodes they expanded.

Then we classify the difficulty of the game into two types, depending on the number of all the possible state in the game. **Easy** (less than 1000 possible state) and **Hard** (10000 ~ 100000 possible state), both have 100 test cases. The result is the average performance of these 100 problems. Note that a difficult problem doesn't mean that it needs more steps to figure out the solution, it means that the variety of this game board is high, the algorithm may need more time to explore the answer.

The experiment result turns out that when the problem gets harder, a good heuristic function can reduce a great proportion of expanded nodes comparing to BFS. We also find that although inconsistent heuristic may lead to suboptimal solution, in other word, not the minimal steps to solve the game, they can be more efficient to find a possible answer.

In the graph below, we analyze the heuristics from a different perspective. Regardless of the number of possible states, we randomly select 100 puzzles whose optimal solutions roughly take the same steps (31~40).