Rush Hour

Assignment 1
2802ICT Intelligent systems
Bennett Taylor - s5095512

Introduction:

The Rush Hour puzzle game, invented in the 1970's by Nob Yoshigahara, consists of solving a 6x6 grid by sliding vehicles in a particular order. The board has grooves shaped like tiles to allow the vehicles to slide back and forward. An exit hole is found on the right side of the board where only the red car can escape. The game comes with different coloured cars and trucks which are predefined. Each car takes up 2 tiles and each truck takes up 3 tiles. There are a range of puzzle cards that come with the game that display different board formations for the player to solve. Each card shows which coloured vehicles get placed on the board and where they get placed. The cards also range in difficulty level, where the higher the level number gives a more difficult puzzle. Each vehicle can only move in a straight line. Therefore if a vehicle is horizontal, it can only move up and down. If a vehicle is vertical, it can only move left and right.

The goal of the game is to make a clear path for the red car to escape out of the exit hole. This is done by moving the other vehicles within the 6 by 6 grid, in order to not obstruct the red cars path.

The Goal:

The aim of this project is to implement a range of different search algorithms to solve the Rush Hour puzzle and compare the differences between them. These search algorithms include Breadth-First-Search(BFS), Iterative-Deepening, A* search and Hill-Climbing. Some Heuristics will tested before using A* and Hill-Climbing in order to find the best approach to solving the puzzle. No heuristics will be used to aid BSF and iterative-Deepening. The informed search algorithms will be tested with in-admissible heuristics to see how the agent acts. The results will be tested by checking how many steps an algorithm took to find a solution and by how many nodes were expanded.

The Approach:

Given a game board, the Rush Hour puzzle requires the player to make as little steps as possible to get the red car to the exit hole. The first approach that comes to mind with this kind of problem is to create a state space (or sample space) that contains all possible board

combinations from the initial game board. This means all possible placements/movement of cars and trucks where each movement is classed as a new board (or state).

Admissible Heuristics:

- **H1: Red car's distance from the exit -** The number of movements needed to get to the goal state is at least the distance of the red car from the exit. This is admissible because it doesn't overestimate the actual number of actions.
- **H2: Number of cars blocking exit -** The number of blocking cars is lower than or equal to the number of actions we need to do.
- **H3: Check cars blocking cars from in H1 -** This heuristic is an improvement to H1. Each blocking car means another move needs to be made. Therefore, adding one for each blocking car. This does not overestimate the cost function.
- **H4: H2 + H3 -** A combination of H2 and H3 where the number of movements is the distance of the red car from the exit plus the number of blocked blocking cars.

Inadmissible heuristics:

- **Only vertical vehicles to the right of the red car (in the same row) -** There can not be any horizontal vehicles to the right of the red car in the same row. This is because it would be impossible to solve if there were. Therefore, the only vehicles blocking the red car from the exit are vertical.

Algorithms used:

Breadth First Search:

Breadth first search is an uninformed search algorithm for traversing tree/graph data structures. It uses a brute force approach to explore all of current states' neighbours. Each state on the depth level is explored prior to moving on to the states at the next depth level. Since each state in Rush Hour represents a different board configuration, states are in the tree by vehicle movement. For example, if the player was to move the A car to the right 3 places, it would be classified as a new state in the next depth level (a child state). This algorithm uses a Queue to keep track of all states in the tree. A state is popped when it is explored and a child state is added when it is discovered. The algorithm has a time complexity of O(|V| + |E|), where |V| is the number of vertices/states and |E| is the number of edges in the tree. It has this complexity because in the worst case every state and edge will be explored before a solution is found.

Iterative Deepening:

Iterative Deepening is an uninformed graph search algorithm which is a depth limited variation of Depth First Search. It visits the stats in the search in the same order as depth first search but limits the search to a specified depth. If the algorithm does not find a goal state, it runs repeatedly with increasing depth limits until the goal is found. The main data structure used is a stack and this is what makes its depth diving nature. When a state is popped from the top of the stack, all its children states are added to the top of the stack. This is then repeated until a goal state is found, depth limited has been reached or there are no more states in the stack. Iterative Deepening is optimal like Breadth First Search but uses

much less memory at each iteration. The time complexity is O(b^d) for a well balanced tree, where b is the branching factor and d is the depth of the goal state.

A*:

A* is an informed graph search algorithm which offers more completeness, optimality and optimal efficiency than Breadth First Search and Iterative deepening. Informed search algorithms should in theory provide better performance because they use heuristics to guide the search towards the right path. However, this can make the algorithm slower than most others if it is searching a big graph. This algorithm makes use of a priority queue ranked by each state's heuristic value.

Hill Climbing:

Hill Climbing is a local search algorithm designed for mathematical optimization. It takes use of heuristics to find a sufficiently good solution. It starts with an initial state, then attempts to find a better solution by making incremental changes to it. It utilises a priority queue, however the queue only contains the current state's children. These children are ranked by heuristic value.

Results:

As mentioned above, we want to test the advantages and disadvantages of each algorithm to see which one is best suited to solving this problem. First we can check the number of states expanded by each algorithm, and how each heuristic has affected A* and the Hill Climbing Algorithms.

As displayed above in Figure 1, we can clearly see that the inadmissible heuristics worked out the best. This is then followed by the more complex heuristics, then null heuristics (BFS).

Figure 2 displays the time taken by each algorithm to complete the problem. Both the A* and Hill Climbing lines represent a group function of those algorithms, displaying the average time that group took. This was to clearly see the results better instead of having 4 lines for hill climbing and 3 lines for A*.
The graph clearly displays the hill climbing algorithms as the fastest of them all. BFS and A*, for the most part, stay pretty close to each other with only slight variation between them. This might be a different story if we were comparing larger state graphs. Iterative Deepening, the slowest of them all, had recorded times so large i couldn't be put on the map. This is most likely due to the number of states the algorithm expands.

Conclusion:

As shown from the results, each algorithm has a wide range of advantages and disadvantages. In terms of expanded state, A* worked the best. Followed by BFS then Hill Climbing then Iterative Deepening. BFS and Hill Climbing provided some faster solutions than A* and Iterative Deepening, but do not always give an optimal solution. A*'s downfall is the amount of nodes it stores in memory and time taken on bigger problems. Hill Climbing seemed to always get stuck in a local maximum. Iterative deepening suffered in time because of the amount of nodes it was expanding. In regards to heuristics, we can see from the results that combining heuristics gives us better results and produces much less states being expanded.

The Code:

Each vehicle is defined as its own object and has four properties:

- Id - The unique letter identifier for each car
- Length - The length of the vehicle (car = 2, truck = 3)
- Orientation - The way the vehicle is facing (vertical or horizontal)
- X - The front x coordinate of vehicle on the map
- Y - The front y coordinate of vehicle on the map
- End_x - The back x coordinate of vehicle on the map
- End_y - The back y coordinate of vehicle on the map

The id of a vehicle represents the type:

- Cars = [X, A, B, C, D, E, f, G, H, I, J, K]
- Trucks = [O, P, R, Q]

The Red car is represented by vehicle id 'X'

Function Definitions:

Class Vehicle()
Responsible for holding unique information for vehicle objects, such as id, orientation, x, y, etc.
Class board()
Object for representing each state/node. It carries each state's unique map(board), list of vehicles, and path taken to lead to that state (moves taken from initial game map to get to this map).

Class Rush_Hour()
This class is responsible for setting up the game and housing all the search algorithm functions. Raw game data is passed into the object when it is created. It then calls create_game_map to create a matrix containing the raw game data. Create_vehicle_list gets called next to create a list of vehicle objects which relate to the matrix. With this information

the class then creates the initial state (start state/start node) for the search algorithms to use later.

Def Breadth_First_Search()

The theory of BSF is to search a state's neighbouring states, then their neighbouring states, then their neighbouring states, etc. This follows the same pattern of behaviour. Every time a vehicle moves, it is classed as a new state (this is because it could have just become a goal state). Therefore, this function checks all possible moves each vehicle can take in a current state. If a move creates a map that hasn't been made before, a new state is created and it is added to the BSF queue. After the function has checked all possible moves within a current state, it moves to the next state in the queue. This happens until a goal state has been found or there is no more state left in the queue.

Def iterative_deepening()

This function is responsible for completing an iterative deepening search. It continuously runs iterative deepening by calling the function iterative_deepening_DFS(). This function will return the goal state if one was found and return None if no goal state was found. If no goal state was found, the depth is increased and iterative_deepening_DFS() is run again. This will repeat until a goal state is found.