Rush Hour

AI Final Project

Itzik Oskovski, Oded Valtzer, Oren Ben Meir

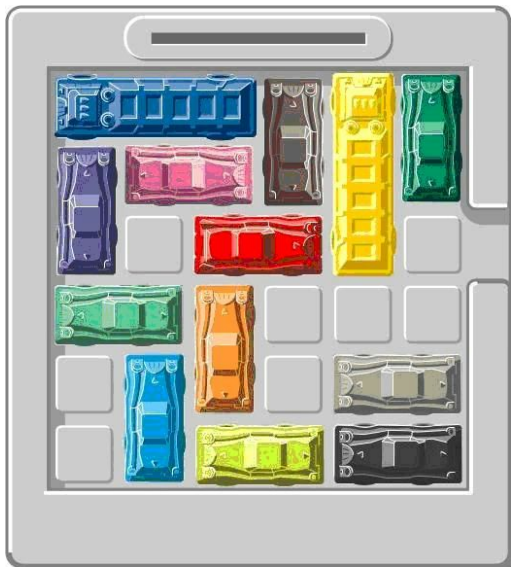
The Hebrew University of Jerusalem
Jerusalem, Israel

***Abstract*—This is an AI final project report in which we talk about our solution to the Rush Hour board game puzzle, using search algorithms such as BFS, A* and IDA*, and utilizing some heuristics.**

I. Introduction

Rush Hour is a sliding block puzzle invented by Nob Yoshigahara in the 1970s. The board is a 6x6 grid with grooves in the tiles to allow cars to slide, and an exit hole which according to the puzzle cards, only the red car can escape. The game comes with 12 cars and 4 trucks, each colored differently. The cars take up 2 squares each; and the trucks take up 3. Each puzzle card shows which colored cars get placed on the board and also where they should be placed. Each card has a different level number. The higher the level number, the more difficult the puzzle. Cars and trucks can only be moved within a straight line along the grid. They cannot be rotated.

The goal of the game is to get the red car out of a six-by-six grid full of automobiles by moving the other vehicles out of its way. However, the cars and trucks (set up before play according to a puzzle card) obstruct the path which makes the puzzle harder.



.**Fig.1** The game board with all the vehicles placed on it. It's easy to see the complexity of the Rush Hour game in this picture

II. Our Goal

In this project our goal is to solve the Rush Hour puzzle with several search algorithm, such as Breadth-First-Search, A* and Iterative-Deepening A* search. In the BFS algorithm we'll simply solve the puzzle, while in the A* & IDA* algorithms we'll test some heuristics to find the best one. Furthermore, we'll test the algorithm with a in-admissible heuristic to see how the agent acts. We'll test our results by checking how many nodes were expanded and in how many steps the algorithm found a solution to the puzzle.

III. Our Approach

Given a game board, the Rush Hour game requires us to choose a minimal series of vehicle movements in order to get the red car to the exit. The natural approach to this kind of problem, as learned in class, is to use search algorithms - where the state space is the set of all possible boards (meaning, all the different placements of cars & trucks on the board) and the transitions are valid vehicle movements that transform one board to another.

As mentioned above, we will solve the puzzle with search agents, using three search algorithms:

- BFS
- A*
- IDA*

As the focus of the course is not on programming or design, we found a Rush Hour puzzle solver in GitHub (Link below) from where we took the code of building the game (holds the construction of the board, cars and trucks) and we focused on writing the search algorithms and on constructing efficient and fast heuristics.

We've constructed both simple and complex heuristics and both admissible and inadmissible heuristics. Let's describe each one and discuss its admissibility:

Admissible heuristics

- The distance of the red car from the exit
    - This heuristic is admissible as the number of actions we need to do in order to get to a goal state is at least the distance of the red car from the exit (it could be more as we might need to move some vehicles). So this heuristic doesn't overestimate the actual number of actions and thus is admissible.
- The number of cars blocking the way to the exit
    - This heuristic is admissible as the number of blocking cars is lower or equal to the distance of the red car from the exit, and thus lower or equal to the number of actions we need to do.
- An improved heuristic of the above where a blocking car which is blocked too, gains two points to the state
    - In this heuristic, the only improvement to the one above is that blocked vehicles gain another point. A blocked car means we need to make (at least) another move, so the improvement doesn't make this heuristic overestimate the actual cost function, and thus is admissible
- The distance of the red car from the exit + the number of blocked blocking cars (the improved heuristic above combined with the distance heuristic)
    - This heuristic is clearly admissible too, as the actual number of steps to the goal is at least the distance of the red car from the exit (in order to drive it to the exit) + the number of blocked blocking cars as every such car adds a movement of the car that's blocking it and a movement of the car itself.

Inadmissible heuristics

- Vertical vehicles to the right of the red car
    - Since there cannot be horizontal cars on the red car's path, the only blocking cars will be vertical. Thus, this heuristic emphasizes vertical vehicles. To improve the heuristic, each such car gains also its length (we would like to give "better" heuristic value to cars opposed to trucks).
    - This heuristic is in-admissible, as can seen in the following figure:

.**Fig.2** In this state it is clear that 4 moves are needed in order to get the red car to the exit, while the heuristic for this state will be 6 moves. So the heuristic overestimates the actual cost and thus is in-admissible.

- Manhattan Distance to Goal
    - In this heuristic we first run BFS on the problem and then, as we have the final position of each vehicle, we sum the distances of each car in the present state to its position in the solution we found in BFS. This heuristic is not admissible, as can be seen in the following scenario:

Let's say this is the initial state of the problem:

It’s clear that if we run BFS on this problem, the solution will be:

with 4 moves. Now let’s look at a possible state of this problem:

For this state, our heuristic will give a value of 7, as the red car has 4 moves to get to its final position and the three lower cars have 1 move each to make. However, it’s obvious that 4 moves will get to a solution, so the heuristic overestimates the actual cost and thus is in-admissible.

IV. How to use the code

The code is written in python 2.7.10. We implemented the interesting part which is the heuristics and the search algorithms. We also implemented an interface of running the program through the command line, and a GUI interface for display of the progress in the game (it prints out the states and the movement of vehicles). In each frame of the GUI interface we have the current state board and a button to move to the next frame (except for the goal state)

Main Files

We have 4 files:

- vehicle.py - this file defines the vehicle object and all its methods.
- util.py - this files is a utility function that we use (similar to what we used in the course projects).
- rushhour.py - this is the game object and the main file. the Rushhour object contains a list of vehicles which represent the state. All the heuristics and algorithms are defined within this file.
- rhGUI.py - this file has the GUI interface. it receives a list of boards and prints out the boards, step-by-step

Each vehicle has 4 properties: id, column position ,row position and orientation (the order of properties is important).
We have 2 types of vehicles - Car (occupies 2 spots on the map) & Truck (occupies 3 spots on the map).
The id of a vehicle represents also its type:

- Cars ids: X, A, B, C, D, E, F, G, H, I, J, K
- Trucks ids: O, P, Q, R

The positions are 0-5 x 0-5 - 6x6 board.
The orientations can be H (for Horizontal) or V (for Vertical),
For example, a full definition of a vehicle is: A23V. A is the id and type (Car type), 2 & 3 is the coordinates and V means vertical orientation.
The 'RED' car (the car that needs to reach the exit of the board) is represented by id 'X' and should always be vertical, in row number 2, and of course should always appear in the definition of a game.

Problem Creation
In order to create an initial state, which is an input to the game, the user needs to create a file with a vehicle definition in each line as described above. for example, a file with the following lines (the order of the lines is not important, though definitions are case sensitive):

A00V
B11H
C31V
D22V
E33H
F24V
G34H
X02H
O30H
Q35H
R52V

will produce the following initial state:

A goal state might look as follows (will be printed by the program):

Notes:

- A state is represented by a list of vehicle definitions as seen above, and thus each successor in the game is a list of vehicles as well, which we translate to the board position and etc.
- Also, note that the goal state is when we have a state which contains the following vehicle description:X42H - which means that the red car is in the exit and the problem is solved.

Input & Run

Once you have a file ready with vehicle definitions (or use our problems), you can start running the program. We use a command line interface to start the game.

- In order to output all the available search algorithms (BFS, A* and IDA*).
  - *python rushhour.py -alg*
- In order to output all the available heuristics, which can be used with the algorithms above.
  - *python rushhour.py -he*
- Running the game - general usage:
  - *python rushhour.py -r <initial state filepath> <algorithm name> <heuristic name>*

Examples:

- *python rushhour.py -r problems/prob1 bfs*
  - This will run BFS algorithm on prob1 in the problems folder (relative to the rushhour.py file).
- *python rushhour.py -r problems/prob1 idastar distanceHeuristic*
  - This will run IDA* on prob1 with distance heuristic
- *python rushhour.py -r problems/prob1 astar distanceHeuristic*
  - This will run A* on prob1 with distance heuristic

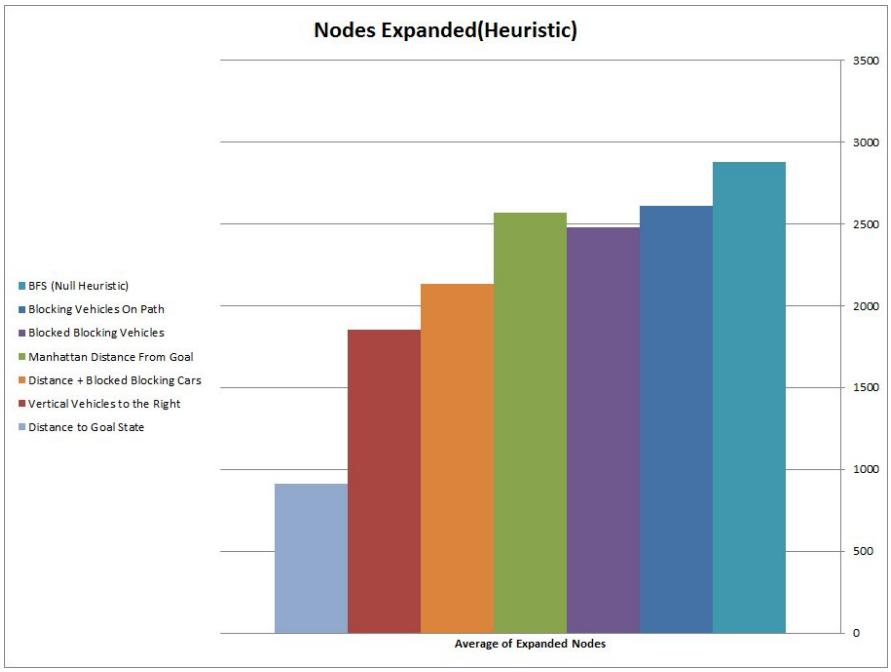
Please note - as this is a project for the university we didn't handle user input errors, so we assume that the input is valid.

Output

The solution printout, step by step.
Solution length
Explored nodes

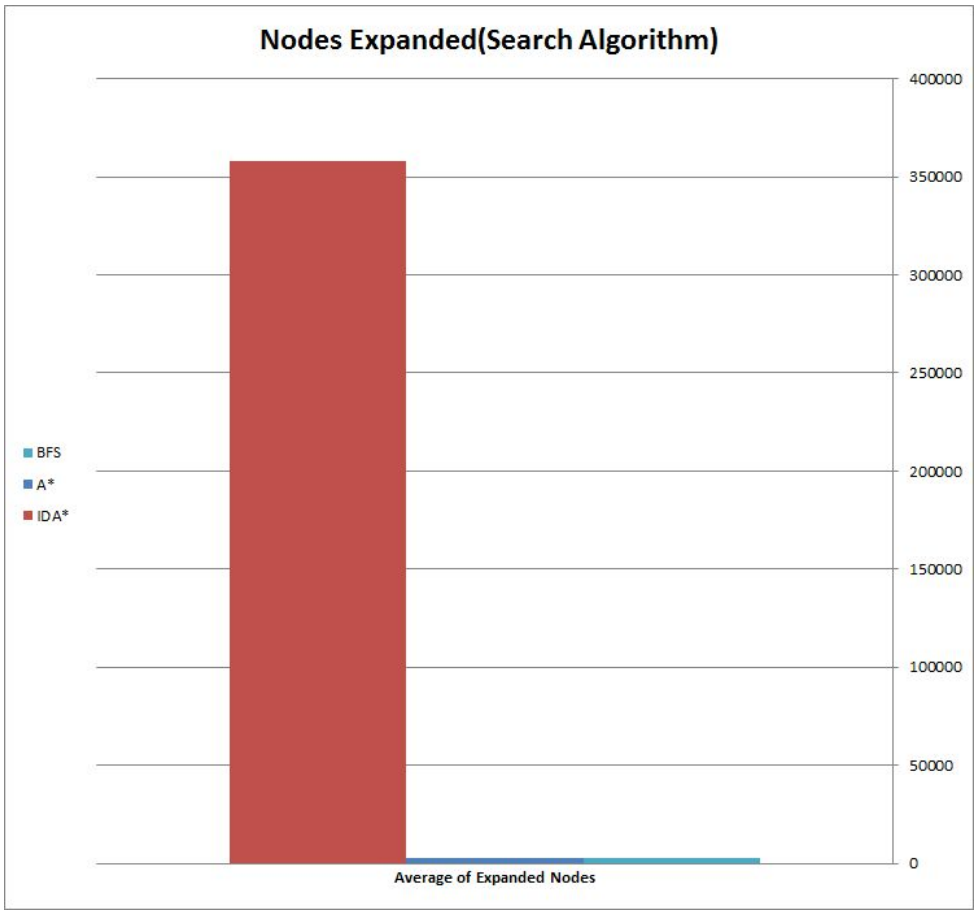
V. RESULTS

As mentioned above, we wanted to test how different search algorithms affected the number of expanded nodes, and how different heuristics affected A*'s number of expanded nodes. First, lets examine the number of expanded nodes as a function of the heuristic function:



In this graph, the two left heuristics are inadmissible, the orange one to their right is a "super heuristic" which is a combination of an improvement to the blocking cars heuristic and the distance heuristic (note that this one is admissible, as explained above). We can see that the inadmissible heuristics are the best ones, after them comes the complex admissible heuristic and the worst one is the null heuristic (which is actually BFS).

Now we would like to examine the difference in the number of expanded nodes as a function of the search algorithm. We'll see it clear in the following graph:



We can see that BFS and A* expanded nearly the same number of nodes (BFS with ~2900 and A* with ~2600) while IDA* expanded, in average, ~360,000 - almost 170 times more!

VI. Conclusions

Examining the results, we can conclude:

- In terms of expanded nodes, A* is better than BFS and both are better than IDA*. The latter usually took us a very long time to run as it expanded lots of nodes.
- Regarding heuristics, we can conclude that combining and improving heuristics gives us better results. This can be seen in the case of "blocking vehicles" heuristic - in which we expanded ~2500 nodes - compared to "distance + blocked blocking vehicles" heuristic (which improves the "blocking vehicles" heuristic and is combined with the "distance" heuristic) - in which we expanded ~2000 nodes. Interestingly enough, the null heuristic (implemented by BFS) expanded around ~2900 nodes so the improvement of the dominating heuristic is very big.
- Another important conclusion regarding heuristics is the impact of the inadmissibility of the heuristic functions. Although for inadmissible heuristics the solution isn't always optimal (but for most of the problems they gave optimal solutions), they are much faster as t hey expand less nodes. Our best inadmissible heuristic "Manhattan Distance to Goal State" expanded ~900 nodes while, as mentioned before, the best admissible heuristic "Blocked Blocking + Distance" expanded ~2000 nodes and the null heuristic expanded ~2500. So it's easy to see that the improvement is huge, and we can conclude that inadmissible heuristic are better than admissible ones, in terms of expanded nodes.

VII. References and additional links

[1] Rush Hour puzzle solver found on GitHub - https://github.com/ryanwilsonperkin/rushhour

[2] Rush Hour Wikipedia page - https://en.wikipedia.org/wiki/Rush_Hour_(board_game)

[3] Wikipedia pages of BFS, A* & IDA* - in the text above (section III)

[4] J. Rosenschein: Introduction to Artificial Intelligence - Course 67842 course notes, Hebrew University of Jerusalem.