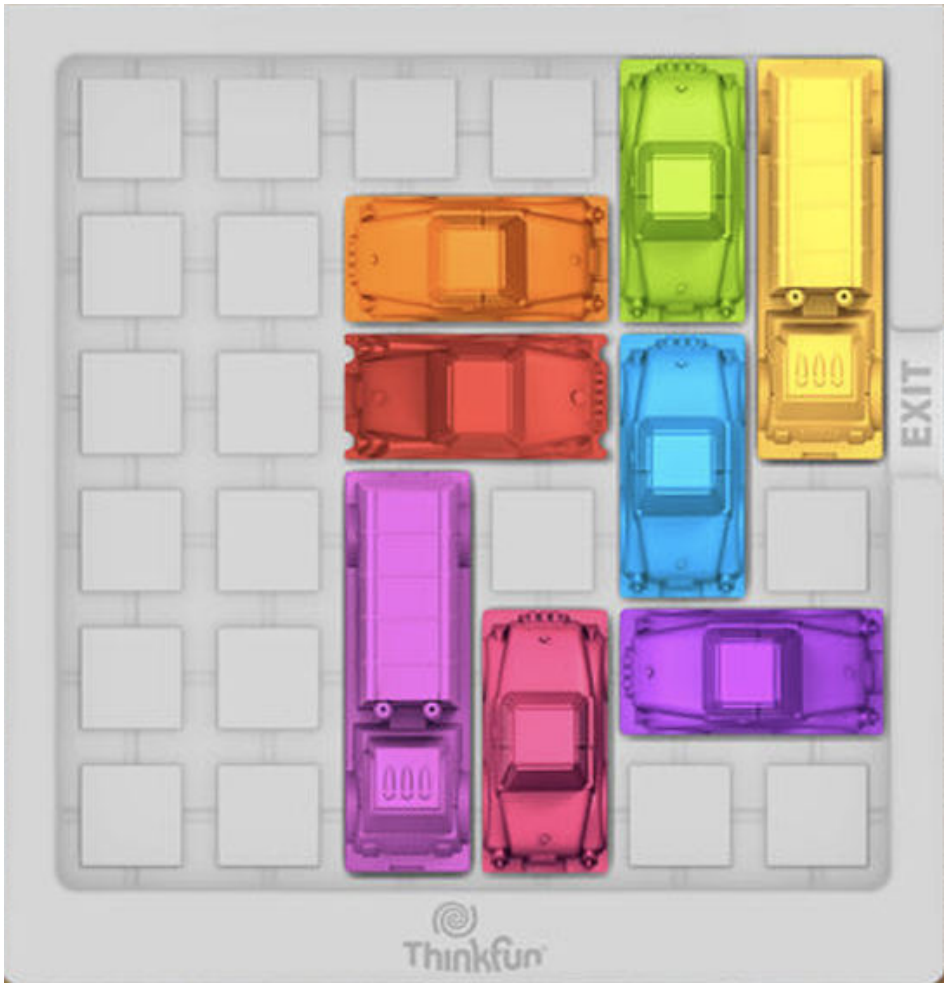
Rush Hour - Intro2AI (67842) Final Project

Michal Kellner (340978758), Gilad Ticher (318770039), Shir Uziel (316536580)

September 14, 2024



Contents

INTRODUCTION

The game of Rush Hour presents an intriguing challenge in the field of artificial intelligence, particularly in the domains of search and planning. The player must navigate a red car through a gridlocked parking lot filled with other vehicles, with the goal of reaching the exit. While the premise looks simple, the problems get complex pretty quickly.
In this project we explore and compare multiple approaches to solving the Rush Hour puzzles. Specifically, we will implement and analyze the performance of Breadth-First Search (BFS), Depth-First Search (DFS), and A* search algorithms with multiple heuristics, alongside a graph planning approach.
Our motivation for tackling this problem stems from its relevance to several key areas in AI:

1. State space exploration: Rush Hour's gameplay naturally translates into a state space search problem, where each configuration of the board represents a unique state.

2. Heuristic development: The game provides an excellent opportunity to design and evaluate heuristics for informed search algorithms like A*. While we explored heuristic development during the rest of the course, this gives us a chance to do an even deeper dive into the subject.

3. Planning and action sequencing: Solving Rush Hour efficiently requires careful planning and consideration of action sequences, making it an ideal candidate for graph planning techniques.

4. Computational complexity: The problem's NP-hardness offers insights into algorithm performance on computationally challenging tasks.

By implementing and comparing these diverse approaches, we aim to gain a deeper understanding of their strengths, weaknesses, and suitability for solving puzzles like Rush Hour. This project will not only demonstrate the practical application of core AI concepts but also provide valuable insights into algorithm selection and optimization for similar state-space exploration problems.
In the following sections, we will discuss previous work in this area, detail our methodology for each approach, present our results, and analyze the comparative performance of the implemented solutions.

Problem Description

The Rush Hour problem is a sliding block puzzle where the objective is to maneuver a specific car (typically represented as the "red car" or "X") out of a 6x6 square grid parking lot filled with other vehicles. In the parking lot, cars and trucks occupy multiple squares in either horizontal or vertical orientations, car size is 2 squares and truck size is 3 squares. Each vehicle can only move forward or backward along its orientation, and cannot change direction. The challenge lies in strategically moving the other vehicles to clear a path for the target car to exit through

a designated space, typically on the right edge of the board. Players must carefully plan their moves, considering the limited space and the interlocking positions of the vehicles, to solve the puzzle in the minimum number of moves possible. In addition to the stated above, we also explored the option for a two player Rush Hour game. We went through many iterations in trying to implement this challenge, as you will see in the coming sections of this report.

PREVIOUS WORK

In the first stages of this project we were interested in exploring previous work that attempted to solve a similar problem. We were able to find some examples online of other attempts at solving the Rush Hour puzzles, mostly using BFS, A*, and IDA approach. We took the basic structure for our project from code [1] we found that used BFS to solve a Rush Hour puzzle. We greatly changed much of the code and implementation but it is built on the foundation from this code.
In order to understand and interact with the data, we wanted to implement a GUI (Graphical User Interface) for our solver. This provided us with visual representations of the puzzle, making it easier for us to explore and analyze the data. In an effort to spend our time on the aspects of this project that are relevant to AI, we decided to integrate our solver with an already existing GUI for the Rush Hour game [2]. We didn't use the code, mostly we took inspiration for matplotlib and the animation.
As we progressed in the project we encountered extremely long runtimes. We found Michael Fogelman's project [3] about the game and used some of his approach to make our project more time efficient. Originally we had written all of our code in Python but after reviewing his project and understanding that we could reach much better results with C++, we rewrote our code in C++ and found that we were actually able to run large, complicated boards and they could be solved in a "normal" amount of time.
The common assumptions that we found were that boards larger than 6x6 were exponentially harder to solve, so most people didn't attempt it. We also found that most previous projects said that A* with the right heuristic is the best solution to this problem. We will try to see if GraphPlan is any better, as we didn't see any previous works that used this approach.
We did not find any previous work that explored a two player game of Rush Hour, so we didn't have other code and results to compare with our own.

METHODOLOGY

Classic Game

Search For the regular rush hour game we used a few different approaches. First, we ran a random algorithm that randomly picks a vehicle and a legal move for that vehicle for each move. This is the benchmark against which we will compare our algorithms.

BFS:

This is another basic benchmark against which we can test the rest of our methods. The BFS algorithm checks all options for actions on a breadth-first level, meaning in a tree of all possible actions and states, the algorithm opens all nodes on each level before descending down to the next level of the tree. While this approach is time consuming and requires opening perhaps more nodes than necessary, it also is guaranteed to find a solution if there is one.

DFS:

The natural progression from BFS is DFS - as the approach is extremely similar but instead of checking all nodes on each level of the tree, it instead goes deeper and deeper into the tree (depth first!) until it finds a solution (unless there is no solution) and then backtracks. We assume that the DFS algorithm will have an extremely high runtime because there are always more "forward" moves to make, meaning there is never a need for backtracking.

A* search:

We use the A* algorithm to find the shortest path from the start node (the start board) to the goal node (the red car gets to the exit). We use a cost function f(n) = g(n) + h(n) where g(n) is the cost to get to the current state (or in our case current board configuration) and h(n) is the heuristic estimate of the cost to get from the current node n to the goal node. With the right heuristic we expect the A* function to give us better results than BFS. We will explore a few different heuristics - some admissible and others not.

- Heuristic 1: Blockers Count
  We count the number of cars blocking the exit. This is admissible because each car blocking the exit needs to be moved at least once, meaning that the minimum number of moves until reaching the goal state is greater than or equal to the number of cars blocking the exit.
- Heuristic 2: Distance to Exit
  We count the distance between the end of the red car to the exit. This is admissible because in order to reach the goal state, the red car needs to move at least that many moves, possibly more if there are other cars blocking its way.
- Heuristic 3: singlePlayer
  If we combine the blocking and distance heuristics, we get an even more accurate heuristic (not perfect - we don't account for cars that are blocking the cars blocking the exit).
- Heuristic 4: Blockers Total Size
  This heuristic combines the lengths of all vehicles blocking the main vehicle from the exit. While this is a good estimator of the number of steps until winning, it can sometimes overestimate (and is not admissible) because for example the vehicle might be of length 4 but only requires one move to clear the path to the exit.

- Heuristic 5: Steps to Clear Path
  This heuristic is similar to Blockers Total Size, except it is admissible. Instead of summing up the length of each vehicle blocking the exit, we sum up the minimum number of moves it takes to move each vehicle blocking the exit (if its a 3 length vehicle, but two moves would be enough to clear the path, we only add 2 to the total sum).

Planning

We used a planning approach using a graph plan, somewhat similar to the way we solved the hanoi problem in an earlier exercise in this course. By using action layers and proposition layers, we were able to build a STRIP language that could solve Rush Hour puzzles. For each possible move we included the prerequisite states, and states to be added or deleted after the move. The parser we created takes a board of 6x6, where each empty space is represented by '.' and each vehicle space is represented by a different letter. The main vehicle is represented by 'X'. The parser takes this board and creates lists of the vehicles and their locations and then creates the domain and problem based on that information.

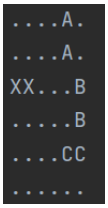


Figure 1: Board Representation

Adversarial Game

In order to make this challenge more interesting, we introduced a new version of Rush Hour - an adversarial game. There are two players (X and Y) and each is trying to reach its own exit before the other. One is going horizontally while the other is going vertically. Each player takes a turn moving one vehicle. We use multiple heuristic approaches to see what performed best. The trick to performing best will be finding the right balance between advancing the player's own vehicle towards the exit and blocking the opponent's vehicle.

Alpha-Beta Pruning

The Alpha-Beta Pruning algorithm is a more efficient version of the Minimax algorithm. It prunes the branches of the tree that are unnecessary to explore. In the two player version of Rush Hour, each node in the tree represents a different state and each edge represents a move. As this is adversarial search, every other layer represents a move by the player and then by the opponent. The player wants to maximize its value on the player layers and wants to minimize the values on the opponent layers. There are also two values, alpha, the best value that the

player can get at that level and the beta which is the lowest value that the opponent can get. If at any point the alpha value is greater than the beta value, we stop exploring that branch of the tree and "prune" it. In this way we are able to keep the algorithm more efficient. We run the AlphaBeta algorithm with different heuristics (the same as mentioned above) and see what gives the best results for the player.

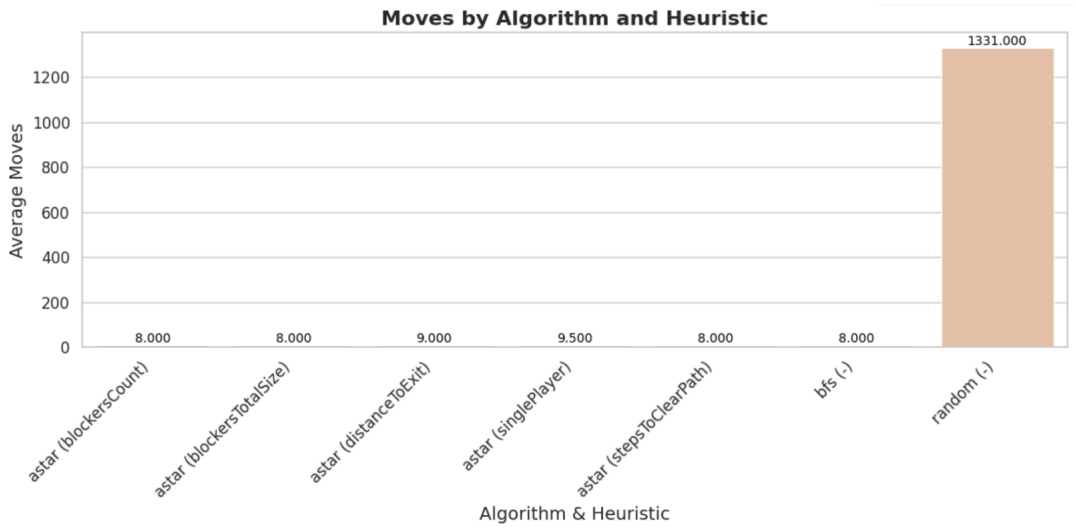
Monte Carlo tree search (MCTS)

The core idea of MCTS is to incrementally and asymmetrically build a search tree, prioritizing the most promising moves based on random sampling of possible states. In game scenarios, MCTS works by performing numerous playouts, where each playout runs the game to completion by selecting random moves. The result of each playout is then used to update the nodes in the tree, so that stronger moves are favored in subsequent simulations. In class, we discussed that the MCTS algorithm operates in four main stages: selection, expansion, simulation, and backpropagation. We selected MCTS because of its proven ability to handle complex decision-making problems with large state spaces (AlphaZero as example), making it highly effective and widely adopted in various types of games.

RESULTS

Single Player

After much trial and error, we were able to zero in on the heuristics and algorithms that gave us the best results. Using the random algorithm as a benchmark, we can see according to the graph below that all of our algorithms performed better than it. It is also important to note that the random algorithm was only able to solve the most simple boards. It was unable to solve more complex puzzles as it had no way of advancing in a "smart" way down the tree.



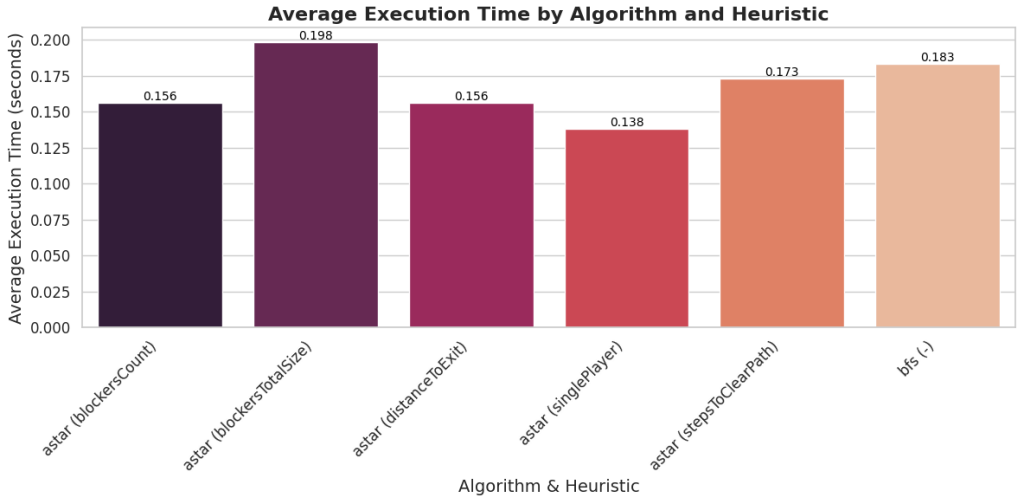
Remark 1. A Note on Running Time
We originally wrote our code in Python, but after having issues with extremely lengthy running times, we decided to convert our code into C++. This improved our runtimes significantly so we used C++ for all of our search problem code.

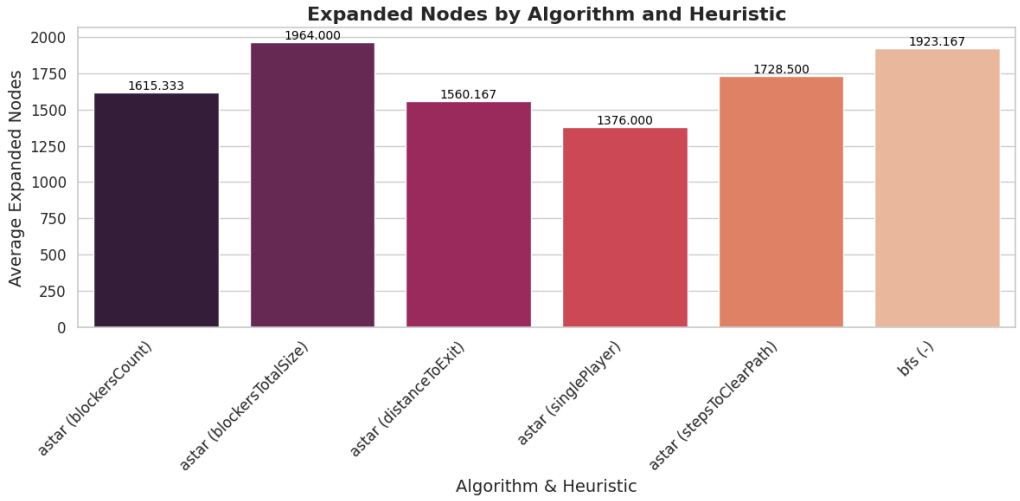
BFS/DFS

As we added more cars to the puzzles, (not necessarily made the puzzles harder, they could be cars that didn't interfere with the main car), the run time got exponentially harder, because the trees grew so quickly. We thought that DFS would take a lot longer than BFS because it never really has to "backtrack" as each move (even if it is the opposite of a move already played) is considered to be a new move. There is nothing that stops us from looping over the same moves over and over again. Therefore, we ultimately didn't include DFS in our project because we knew it wouldn't be efficient.
The puzzles we used are from the official Rush Hour game. These puzzles come with official solutions of a minimal number of steps. We found that the number of steps that the BFS algorithm used to solve each puzzle matched the number of steps in the official solution.

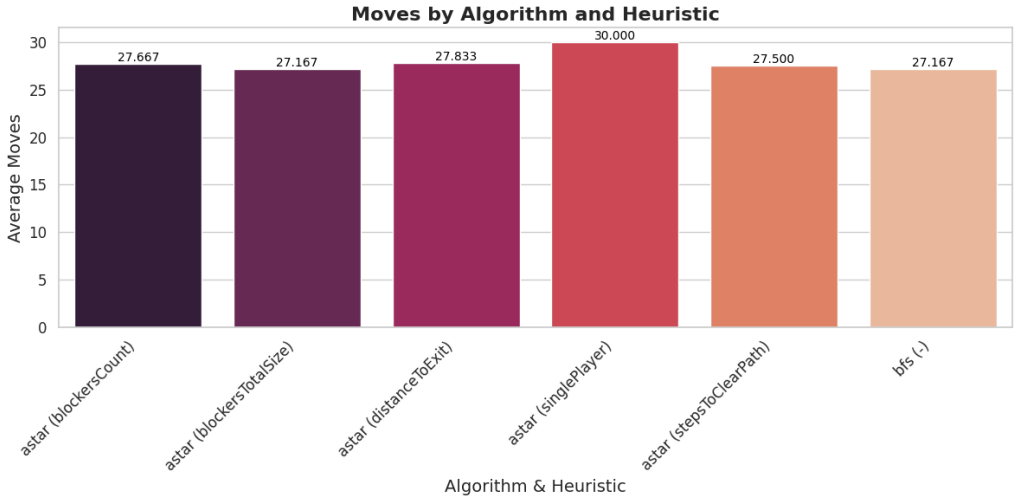
A* Algorithm

We expected the A* algorithm to have significantly better results than the BFS algorithm, but we found that it was highly dependent on the heuristic used. We can see in the below graphs that most of the heuristics we used improved the runtime and number of nodes expanded, but the Blockers Total Size heuristic was actually much worse than BFS. Interestingly, this was also the only inadmissible heuristic that we used. We concluded that because it is inadmissible, the heuristic fails to accurately predict what the best next move is and thus takes longer to reach a solution.



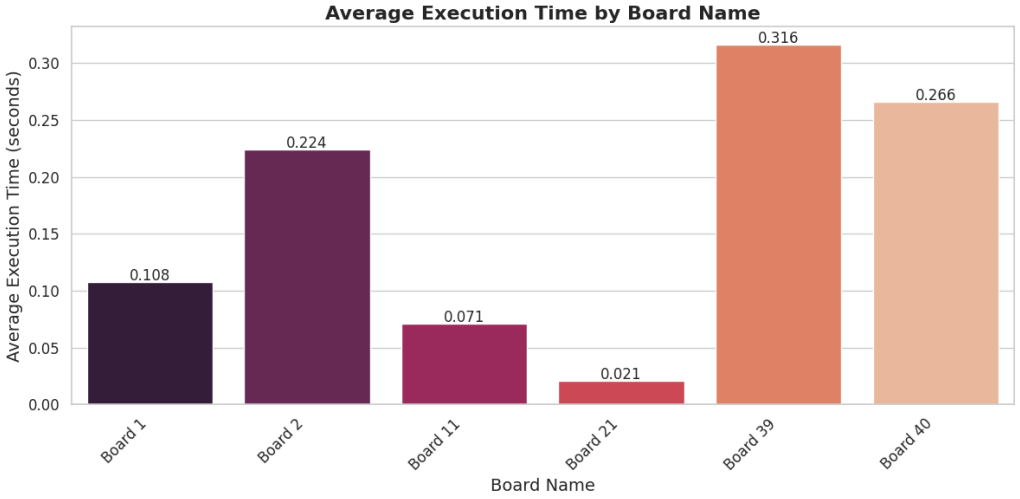


When we look at the average moves broken down by heuristic - we see that all of the algorithms and heuristics average between 27-30 moves (on average for all of the boards that we tested). Despite the difference in the number of nodes expanded, they all found solutions in about the same number of steps. We see that more expanded nodes does not mean that the algorithm is "more thorough", as the end result is more or less the same.



Correlation between board difficulty and execution The Rush Hour game comes with 40 puzzles, of varying difficulty. A puzzle is considered more difficult if it needs more moves to be solved.

Difficulty	Beginner	Intermediate	Advanced	Expert
Puzzle #	1-10	11-20	21-30	31-40



We expected the runtime to increase as the difficulty increased, but we see above that this isn't the case. Board 2, one of the easiest boards in the game, takes a lot longer for the algorithms to solve than Board 21, an advanced puzzle. Puzzle 2 requires 8 moves, while Puzzle 21 requires 21 moves.

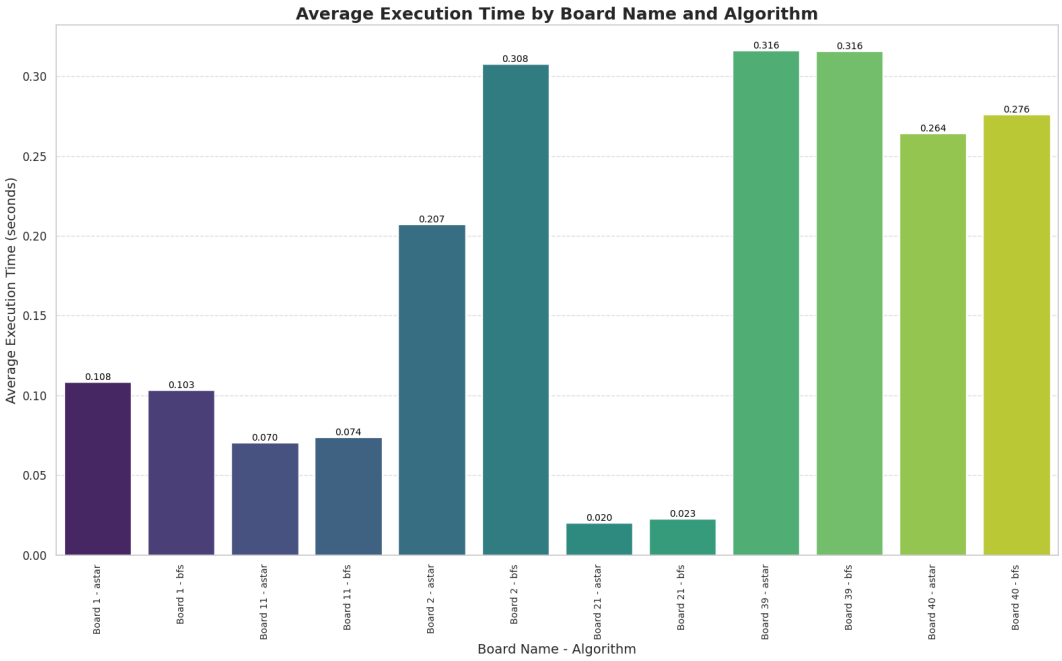
Upon further investigation, we saw that puzzle 2 had many more cars on the board than 21. Despite the puzzle being easier in terms of number of moves, it was actually much harder for

our algorithms to process, as the tree of game states was so much larger in correlation to the number of cars. The state space for puzzle 21 is much smaller than that of puzzle 2. From this we conclude that humans and algorithms catalog puzzles as more or less challenging on different scales. In this case the human scale of difficulty is based on the number of moves required to solve the puzzle, while the scale of difficulty for these algorithms is actually based on how large the state space is.

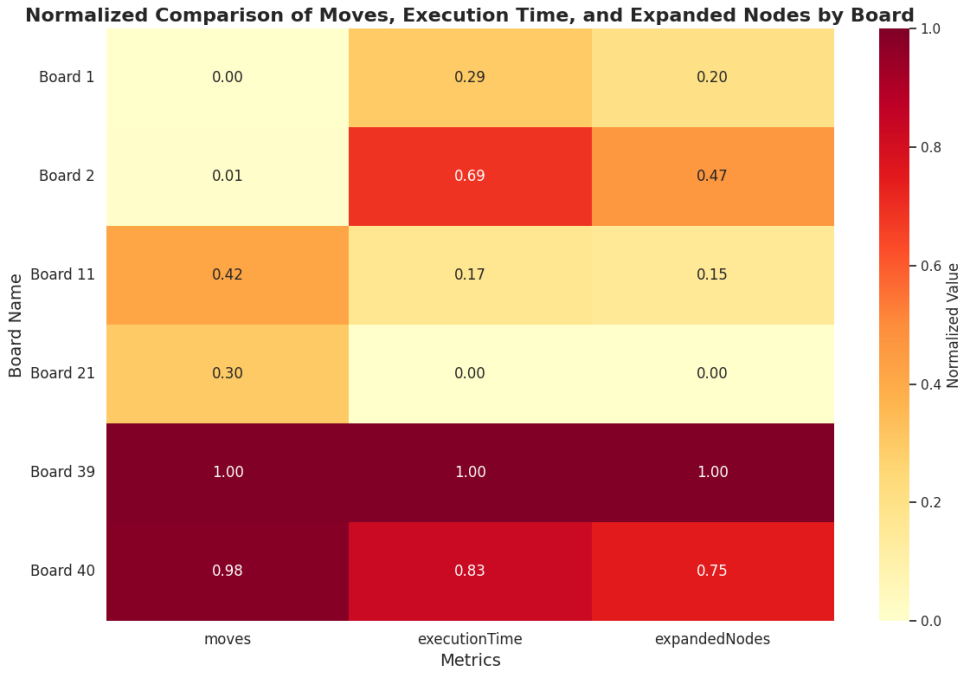
Figure 2: Puzzle 2

Figure 3: Puzzle 21

When we look at the average execution time for each board by A* algorithms vs BFS, we see that for most of the boards, A* and BFS perform more or less the same. However, when we look at puzzle 2 again, we see a big discrepancy between the execution time for A* and BFS. Again, we see that the state space is large so we understand why the BFS execution time is similar to that of Board 39, which also has a large state space. Interestingly, A* is able to cut down the execution time because it does not have to explore each layer of the tree in its entirety.



Below is another graph that supports our claims from above, that there is not a direct correlation between number of moves and execution time/expanded nodes.



GraphPlan

We were able to create a GraphPlan that solved boards quickly and easily, but only for very small, simple boards with just three or four moves. As we added vehicles or made the solution more complicated, the GraphPlan took too long to solve the puzzles because there were exponential numbers of possibilities and moves it had to think through. We discovered that the use of a STRIPS language with a GraphPlan is not so efficient on larger puzzles of this type. While we expected the GraphPlan to give us more interesting results, we ultimately found that it was better to approach the Rush Hour puzzle as a search problem.

Two Player

We ran experiments that included changing the algorithm both for the player and for the opponent. In creating our heuristics, we tried many different things - do we include the score of the other player (negative or positive?)? How much weight do we put towards the opponent's score? How much weight do we put towards blocking the opponent as opposed to distance to exit? As we played around with the numbers we were able to find what we found to be the best algorithm to solve the two player game.

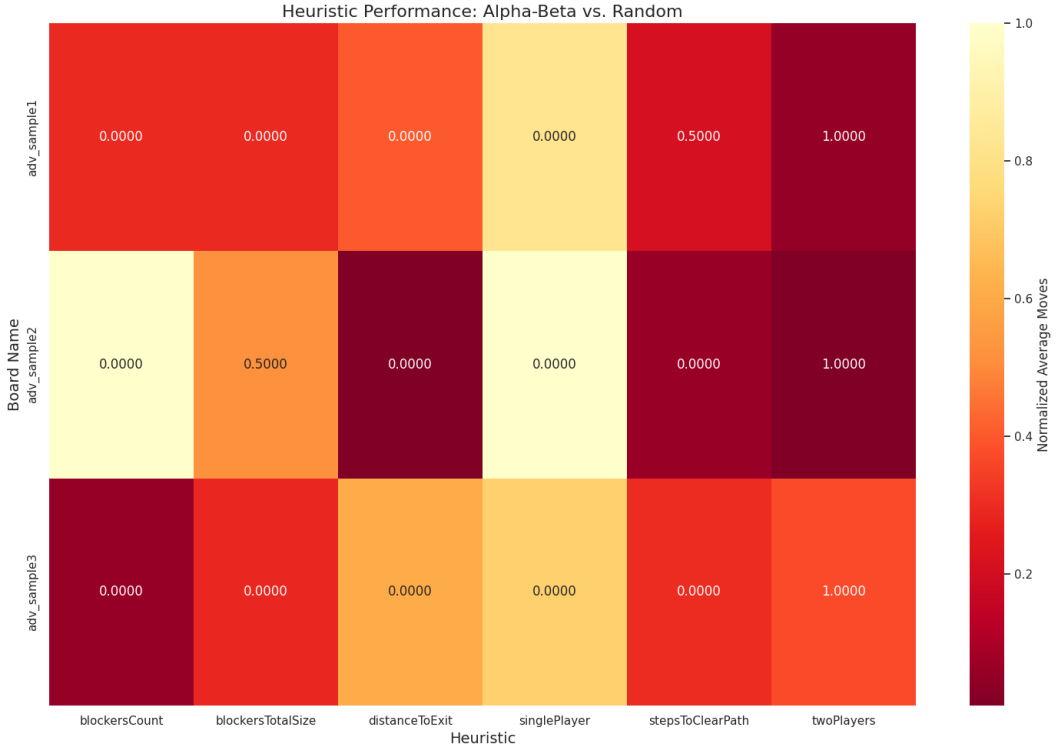
AlphaBeta

In its original form, the AlphaBeta algorithm often got stuck in a loop as each player would move the same piece back and forth each time. In order to get rid of this loop, we added a random mechanism that chooses randomly between the actions with the highest heuristic value (instead of always choosing the first one). Additionally, we added the option for a player to advance a single vehicle more than one spot forward (as long as it was just one vehicle per turn).
In the AlphaBeta algorithm we use all of the heuristics from the SinglePlayer game, but we also introduce a new heuristic called TwoPlayer that takes into account both the player and the opponent states.

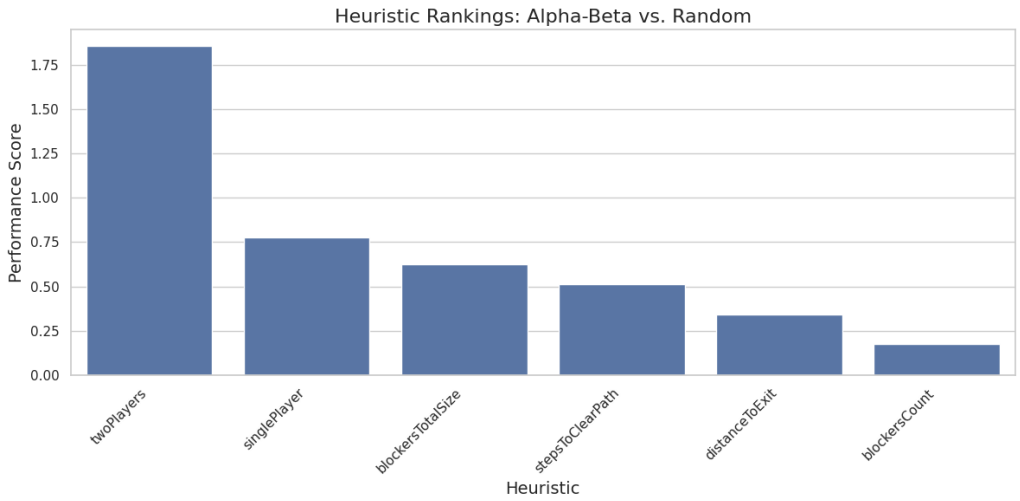
```
TwoPlayer(X):
  score = 2 * (Opponent Distance to Exit - Player Distance to Exit)
          + (Player Blockers - Opponent Blockers)
  If (Player Distance to Exit == 0)
    score += 20
  If (Opponent Distance to Exit == 0)
    score -= 20
  return score
```

We ran each of the heuristics against a random opponent and then we ran the TwoPlayer heuristic against an opponent running AlphaBeta with all of the different heuristics.

AlphaBeta against Random



The above graph shows a heat map of the average number of moves in a game between the AlphaBeta algorithm with the given heuristic and a random agent. The darker the box, the lower the number of moves in the game. The lighter the box, the higher the number of average moves. The number written in the box represents the win rate of the AlphaBeta algorithm with the given heuristic. We see that the Two Player heuristic always wins against the random agent and has a decently low amount of moves. Therefore it is the approach we would pick as the best way to solve the Two Player game. Additionally, the Blockers Total Size and Steps to Clear Path algorithms sometimes beat the random agent, while the rest of the heuristics were always beat by the random. We expected the heuristics to perform better, but we also knew that the Two Player heuristic was the only one to take into account the advancements of the opponent. This is supported by the Heuristic Ranking graph below.

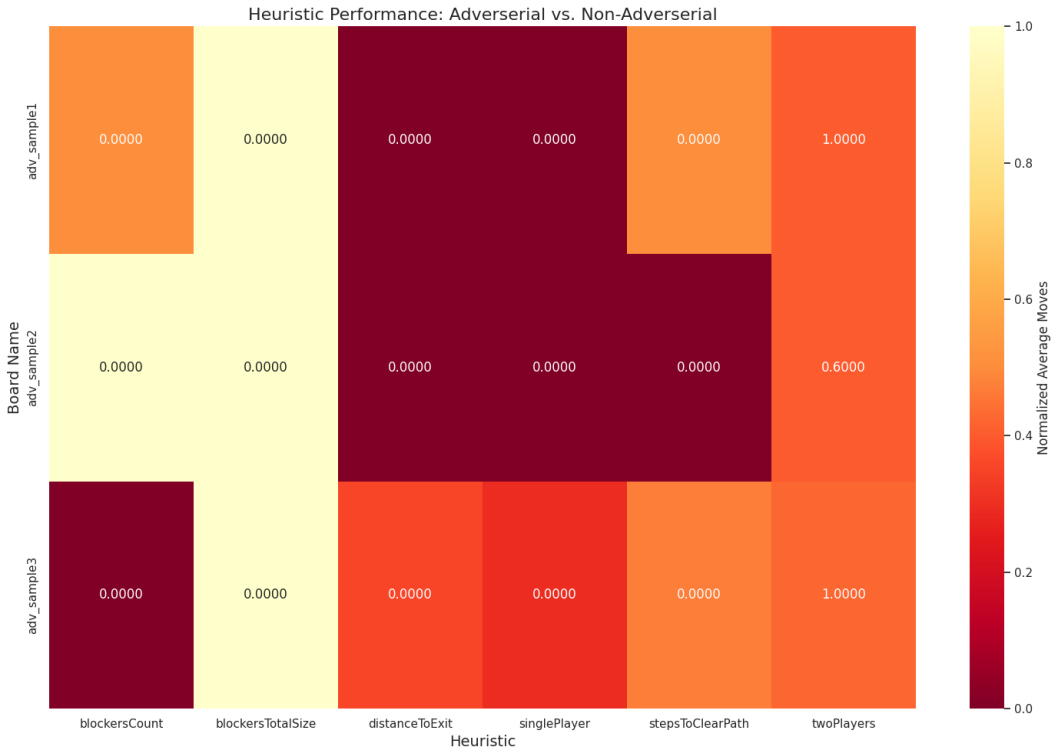


Score Interpretation:

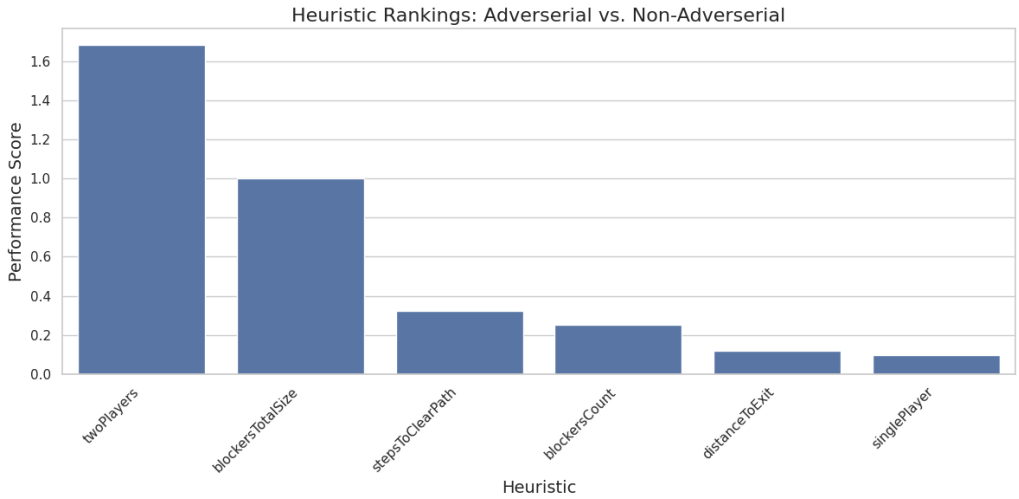
- The score now combines these three factors:
  - win_rate: Directly contributes to the score (higher is better)
  - (1 - avg_moves_when_won) * win_rate: Rewards low move counts when winning
  - avg_moves_when_lost * (1 - win_rate): Rewards high move counts when losing
- A perfect score would be 2 (win rate of 1, always winning immediately, and theoretically always losing with the maximum number of moves)
- Realistically, scores will be between 0 and 2

AlphaBeta: Adversarial Heuristic vs. Non-Adversarial Heuristics

In the below graph we see that none of the Non-Adversarial Heuristics ever win against the Two Player heuristic. The Two Player always wins or at least gets to a tie (no winner after a certain number of moves).



In terms of Heuristic rankings, we see again that the Two Player heuristic is by far superior to the rest of the heuristics, but amongst the non-adversarial ones, the Blockers Total Size is the best heuristic in the AlphaBeta algorithm against another player.

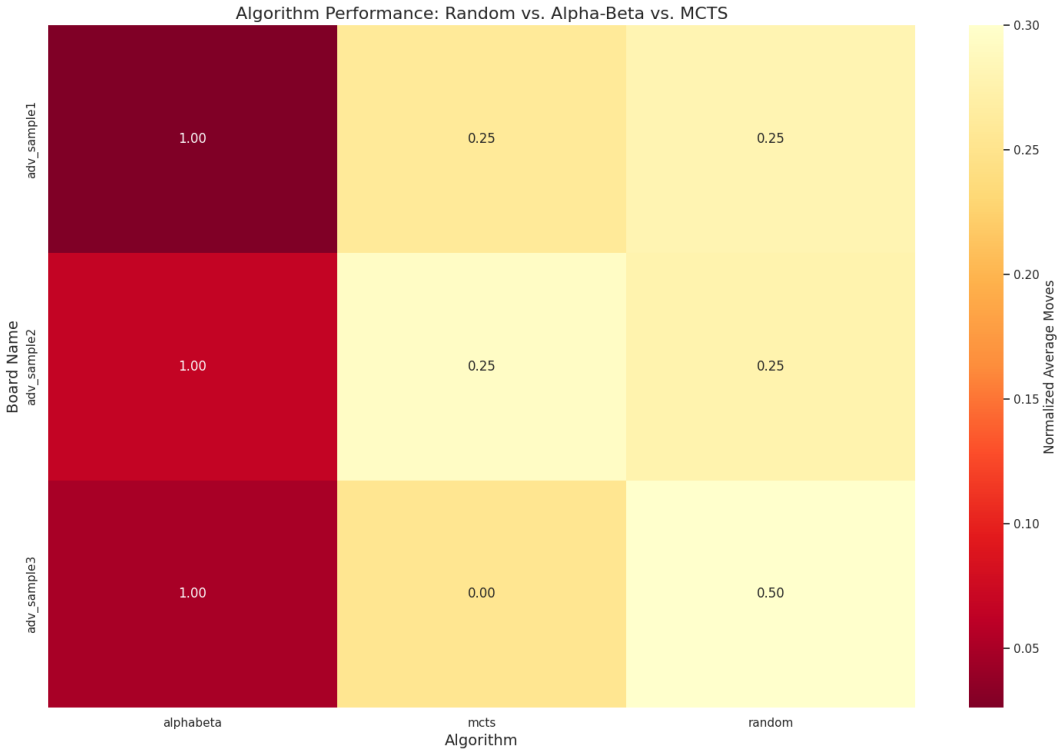


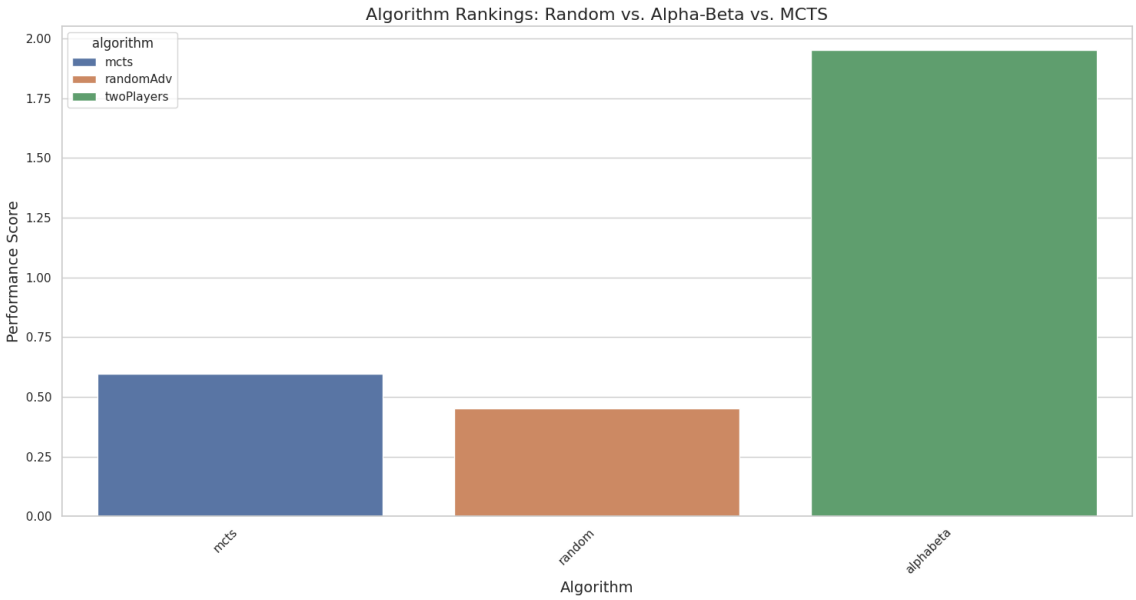
MCTS

The MCTS algorithm works based on simulations, meaning the more simulations run, the better the algorithm will be at predicting what moves are "good moves". Interestingly, we found that the MCTS algorithm (whether it had 200 or 1000 simulations) only won against the random agent if it made the first move. If the random agent made the first move, then the MCTS agent always lost.

So which algorithm is the best?

In the graphs below we see that the AlphaBeta Two Players algorithm is by far the best approach to solving the two player game. The MCTS is marginally better than the random algorithm but not by much. We expected the MCTS to have better results but in its current form we would not recommend using it in the Rush Hour game.





SUMMARY

In this project we tried to solve the problem of the Rush Hour game, on multiple boards of varying difficulties. The models that we included in our project were BFS, A* with varying heuristics and Graph Plan. Additionally, we modeled a two player Rush Hour game with Alpha Beta and Monte Carlo algorithms.

Ultimately we concluded that the best results on the single player game came from our Single Player heuristic which combines the number of vehicles blocking the exit with the distance of the red car from the exit. This is as we expected, as it is the closest heuristic to the actual cost of moving the red car to the exit. However, it is still a relaxed version of the actual cost so it does not give perfect results.

While we had high hopes for the Graph Plan approach, it turned out to have really quick results on simple boards and extremely lengthy results that never finished on more complex boards. We would not recommend using the Graph Plan to solve Rush Hour games.

For the two player game, we concluded that the best way to solve the problem, both against another heuristic and against a random player is to use the AlphaBeta algorithm with the TwoPlayer heuristic. This was the only heuristic that took into consideration the score of the other player.

What could we have done differently or better?

In regards to the GraphPlan - perhaps if we converted it to C++ it would have run faster and could handle slightly more complicated puzzles but we think that in general it wasn't the best

approach to this puzzle.
In regards to the A* algorithms - we could create more heuristics with more complicated calculations, like counting the number of cars that are anywhere on the side of the board between the red car and the exit (not just those blocking the path, as the others might be blocking those blocking the path) etc.
In other board games like Chess or AlphaGo, the MCTS algorithm was found to be very effective. We were unable to get the MCTS algorithm to work well on the Rush Hour puzzle but perhaps we should have spent more of our time on improving our MCTS algorithm and making it more efficient rather than developing more heuristics for the A* and AlphaBeta Algorithms.

PROJECT CODE

Here you can find our code → git repository

References

[1] CirXe0N (2017). Rush Hour Solver. GitHub repository. Retrieved from: https://github.com/CirXe0N/RushHourSolver

[2] Liao, W.C. (2021). Rush Hour Puzzle AI. GitHub repository. Retrieved from: https://github.com/LiaoWC/rush_hour_puzzle_AI

[3] Fogleman, M. (2013). Solving Rush Hour. Personal website. Retrieved from: https://www.michaelfogleman.com/rush/