SOLVING RUSH-HOUR WITH ALGORITHMS

Y . Kada
University of Amsterdam
y.kada@live.nl

R . F . G. Wagid Hosain
University of Amsterdam
rfg.wagidhosain@gmail.com

1. Introduction

Rush-hour is a well-known sliding block puzzle. The concept is that of a parking lot with cars blocking each other's paths. The objective is to get the red car out of the parking lot by moving cars in such a way that the red car gets to the exit. The rules of the game are very intuitive.

Traditionally, the size of the board is six by six blocks. All cars have a width of one block and a length of either two or three blocks, except for the red car which always has a length of two blocks. The cars can be positioned either vertically or horizontally. The red car is always positioned horizontally in line with the exit. A possible move of a car is dependent on its direction, which means that a car placed horizontally cannot move vertically and vice versa. Cars can only move to a block adjacent to their position when it is not occupied by another car. Jumps and rotation of a car are not permitted. In total the game has three constraints: cars may not overlap, jump over other cars or rotate.

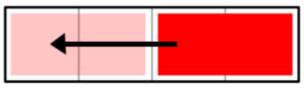

Figure 1 : This move could have a cost of one or a cost of two depending on the chosen definition of a move.

In this case, seven puzzles varying in board size and difficulty are to be solved using algorithms. Two choices have been made to define a satisfying solution. First, the choice had been made that a solution should always be optimal. An optimal solution includes necessary moves only. Therefore a solution should always be found in the least amount of moves possible. There might be multiple optimal solutions for this game. Finding an optimal solution is the objective, other solutions are not accepted. This makes the problem a constrained optimization problem. Second, it was needed to define what is seen as a move (figure 1). Each time a car position is changed is considered a move. The choice has been made to count the amount of blocks a car has travelled and consider. It is possible to pay no heed to the distance a car has travelled. This way every move has a cost of one, no matter the distance. Different understanding of a move could lead to a different optimal solution.

1.1 State-Space

The state-space size of rush hour is considered infinite when focused on move sequences. A car can be moved left and back for an infinite amount of time, never leading to a solution. These eternal loops in moves should be eliminated and only unique moves should be taken into account. This automatically results in a state-space consisting of only unique board configurations, from here on referred to as game states. The state-space is dependent on the size of the board, number of cars and the movability of each car in each of the states. Therefore the state-space can differ significantly from puzzle to puzzle.

It is assumed that it is impossible to calculate the exact size of the state-space with a formula, because of the variability caused by a move on the movability of other cars. Therefore only an upper bound (overestimation) can be calculated. The exact size of the state-space can be found by running a brute force algorithm which tries all possible moves and counts all possible unique states.

To calculate the upper bound of the state-space, all possible configurations are calculated by multiplying all possible positions a car can take, not taking into account configurations with cars overlapping or cars jumping over one another.

$$State\ space\ size = (d-1)^c * (d-2)^t$$

Formula 1 : Calculating the upper bound of the state-space size, with d being either the length or width of the board (dimension), c being the number of cars of two blocks (cars) long and t being the number of cars of three blocks long (trucks).

In table 1 the state-space size of each puzzle is stated. There seems to be an exponential increase in the state-space size when there is an increase in cars and board size. This is to be expected, since the number of possible positions a car can take will increase when the board is larger.

Table 1: Upper bound of the state-space size of each puzzle.

	Puzzle 1	Puzzle 2	Puzzle 3	Puzzle 4	Puzzle 5	Puzzle 6	Puzzle 7
Board dimensions	6x6	6x6	6x6	9x9	9x9	9x9	12x12
State-space size	$1*10^{6}$	$1*10^{9}$	$1.9*10^{19}$	$1.9*10^{19}$	$3.3*10^{19}$	$7{,}2*10^{23}$	$1{,}4*10^{38}$
Number of cars	6	12	12	12	16	18	28
Number of trucks	3	1	1	10	6	9	16
Total number of vehicles	9	13	13	22	22	27	44

2. Methods

Our aspiration was to write an algorithm that could automatically find an optimal solution for any Rush-hour game with a maximum board dimension of 12x12. Puzzle specific heuristics were not favored because of this ideal. Therefore more general approaches needed to be found. A total of three algorithms have been written and compared.

2.1 Depth-first search with pruning

Depth-first search is implemented as a state-space search algorithm. The algorithm is usually programmed recursively and searches each branch (sub-tree) of the search-tree for a solution, before trying the next branch to find a solution. It is not advised to use this algorithm without a depth-limit or pruning when the depth of a branch in the search-tree could be infinite. Pruning the search tree will decrease the total time needed to find an optimal solution. The search-tree is pruned during the search. Each time a solution is found higher in the branch, the search in next branches will only reach as far as that depth. Visiting a game state multiple times in the same game will not lead to an optimal solution and can cause the algorithm to get stuck in an infinite loop. Therefore only unique game states in a path will be explored further.

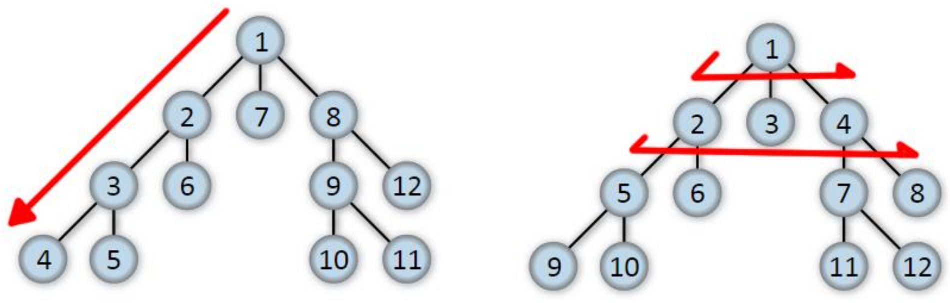

Figure 2: The order in which the algorithm explores game states in the state-space, illustrated using a simplified abstraction of a search-tree. Left: Depth-first search, right: Breadth-first search.

2.2 Breadth-first search

Like depth-first this algorithm is also implemented as a state-space search algorithm. Starting from the first board, every possible game state that is one move removed from the first board is stored in a queue. The first board is removed from the queue and stored in a list. This queue will be explored in order and to this will be added all new possible game states. In figure 2 the difference in search order is illustrated. The advantage of this algorithm is that if a solution is found it is an optimal one. The disadvantage is that it takes a lot of memory to store all the boards in the queue when the optimal solution is located deep in the search tree. Earlier explored game states found deeper in the search tree will not lead to an optimal solution and will not be explored further.

2.3 A* algorithms

The A* algorithm is an uninformed search used to find an optimal path to a goal with the help of a total cost formula f(x). A* is mostly used to find the best path from node A to a node B in a graph. This is done by evaluating the total cost of every node and exploring the nodes that have the lowest cost first. If A* is used to find the shortest path from A to B the total cost is calculated by adding the travelled distance and an underestimation of the total distance, usually the Manhattan distance (crow flies) to the goal. The total cost uses a heuristic value, such as the Manhattan distance. A* will always find the best path when the heuristic value is an underestimation.

The A* algorithm is breadth-first extended with a priority queue. Every game state the algorithm finds will be evaluated using a total cost formula. Every game state will be evaluated by the total cost and will be inserted in a priority queue. The traditional A* needs to be extended or changed before it can be applied to the Rush-hour game. Instead of the travelled distance, the number of moves will be used. It is expected that using just the Manhattan distance as heuristic to calculate the total cost will not lead to much difference in the priority of the game states.

For the Rush-hour game two different heuristics are added to the total cost. The first heuristic is considered a simple heuristic and will increase the total cost by one for every car that is blocking the path to the exit for the red car. This was considered, because all cars blocking the red car should be moved out of the way for the red car to arrive at the exit.

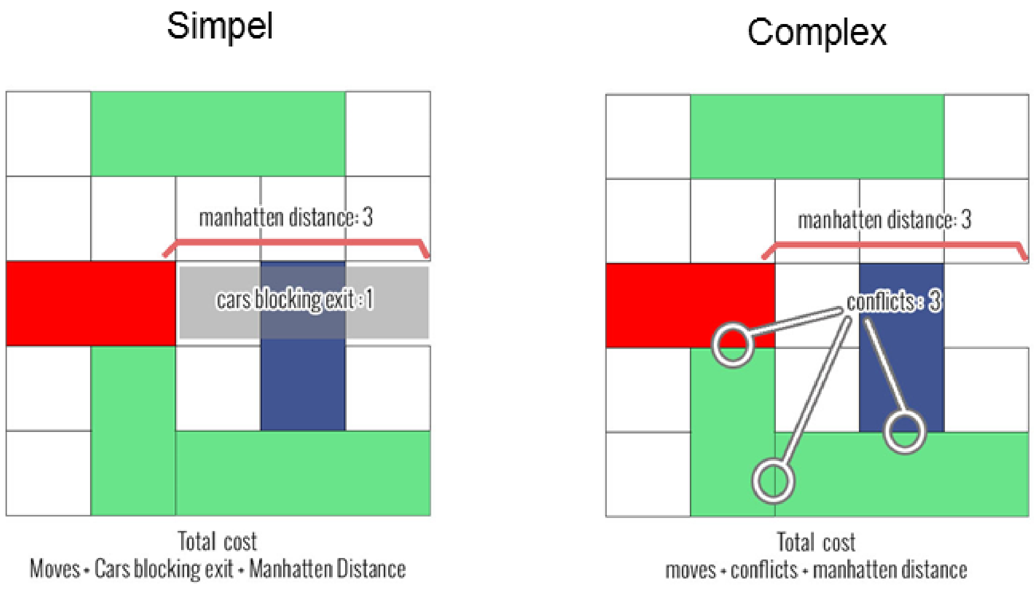

Figure 3: Heuristics used to calculate the total cost of a game state.

The second heuristic is considered more complex. Instead of focusing on the cars blocking the exit, conflicts that the cars blocking the exit have are taken into account. A conflict occurs when the car is blocked by another car, from at least one side. If car A is blocked by car B and car B is blocked by car C there are a total of two conflicts. In figure 3 the conflicts of a game state are illustrated. Only direct blocks are considered a conflict, therefore the truck positioned at the top is not considered in conflict with the car. This heuristic was considered because the conflicting car should be moved before the cars blocking the exit can move and clear the path for the red car.

These A* algorithms are naïve algorithms, meaning that there is no knowing if the heuristics would always evaluate a good game state as a good game state and bad game states as bad. Figuring out a heuristic that would always do so for any Rush-hour game, is very difficult and it is not certain if such a heuristic exist.

3. Results

All algorithms have been run 20 times to compare their performance and results. The number of moves needed for an optimal solution is shown in table 2.

Table 2: Number of moves of an optimal solution found with the algorithms.

	Puzzle 1	Puzzle 2	Puzzle 3	Puzzle 4	Puzzle 5	Puzzle 6	Puzzle7
Moves	83	29	35	51	-	-	-
State-space size	$1*10^6$	$1*10^9$	$1.9*10^{19}$	$1.9*10^{19}$	$3.3*10^{19}$	$7,2*10^{23}$	$1,4*10^{38}$

3.1 Depth-first with pruning

When testing the algorithm on an easy test game with three cars the algorithm finds a solution within seconds. However, when running it to find a solution for any of the 6x6 boards provided by the case, the algorithm does not find a solution within 15 minutes. This algorithm was deemed too slow and not efficient enough.

3.2 Breadth-first

When testing this algorithm on the same testing board as depth-first search, the solution was found in less than a microsecond. All 6x6 games were solved rather quickly. Only one 9x9 game was solved and the 12x12 was not solved due to lack of memory. The runtime for finding the solutions are depicted figure 4.

3.3 A* algorithms

A* simple and complex could find the optimal solutions for puzzles one to three. Unfortunately, the heuristics did not help with solving the remaining puzzles 5 to 7. The runtime of the algorithm with both heuristics are shown in figure 4 as well.

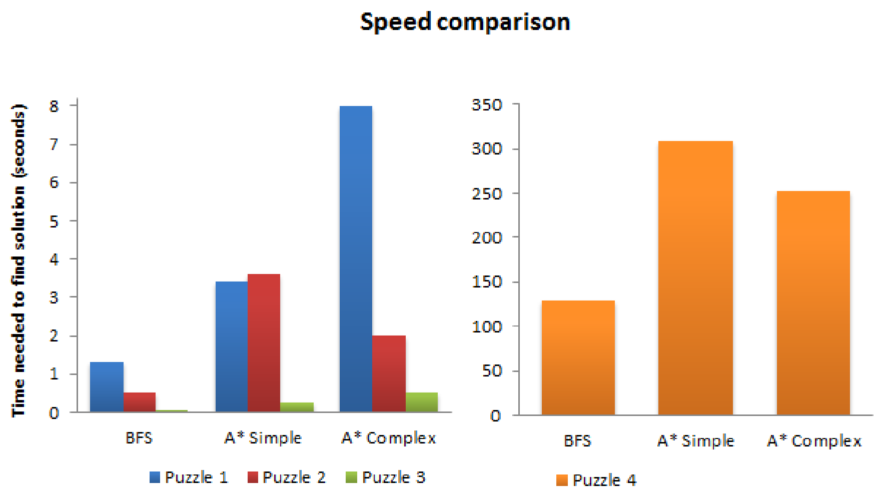

Figure 4: Column chart comparing algorithm the time needed to find an optimal solution. The values are the mean of 20 runs for each algorithm for every puzzle

4. Conclusion

The objective was to solve all seven puzzles and to find the optimal solution for these puzzles. Unfortunately, the objective to solve all seven puzzles was not met using the algorithms as they are described. However, four out of seven puzzles were solved and their corresponding optimal solutions where found. The results are quite interesting.

The depth-first search with pruning algorithm was found to be very slow and was left out of the comparison. The breadth-first search algorithm was significantly faster than any A* algorithm implemented. This is very interesting, because the A* algorithm was implemented with the thought of getting better results by exploring the best options first. It was to be expected that in some cases the A* algorithm would be slower, because of the extra calculations needed to evaluate the game states. However, the A* algorithms performed far worse than the breadth-first algorithm, which probably indicates that the heuristics used were not efficient in choosing better game states. A* with the complex heuristic performed better than the simple heuristic on puzzle 2 and 4.

Although the A* algorithms were implemented naïvely, nevertheless it was in our interest to take a risk and implement these to analyze their performance. The heuristics turned out to be not sufficient in deciding the priority of the game states.

All algorithms found an optimal solution. The slowness of the program might be caused by the heaviness of the Rush-hour program, which handles move operations and stores information about the game. The programming of this Rush-hour core was not memory efficient and needed a lot of memory for each generated game state. The algorithms would perform much better if the Rush-hour core would be memory efficient.

Three puzzles remained unsolved. When looking back at the upper bound of the state-space size, it is noticeable that the size of the state-space for the last three puzzles were significantly larger than the first three puzzles. This value does not indicate the difficulty of the game, but could explain why the algorithms ran out of memory before finding a solution.

Although the runtime of an algorithm is a good method to compare one algorithm with another, the runtime does not say much about the performance of the algorithm itself. For example, running the breadth -first algorithm on a computer with high performance level and with a faster compiler will find a solution in a shorter amount of time.