[]: Solving the game Rushhour in the least amount of steps

Oscar Keur

Universiteit van Amsterdam

O.keur@student.tudelft.nl

Kyra Kieskamp

Universiteit van Amsterdam

Kyra.kieskamp@student.auc.nl

# 1. Inleiding

Rushhour is a sliding block puzzle created in the seventies by Nob Yoshigahara, and is analyzed in our research. The initial form of the game is a 6x6 playing field in which various cars and trucks are situated. The cars have a length of 2 board places and the trucks have a length of three board places, and can be placed either horizontally or vertically on the board. The goal of the game is to get the red car to the exit of the board, which is the opening at the right hand side, see figure 1. This can be achieved by step-wise moving the cars and trucks on the playing field to a new position until the red car can exit the board. The movement restrictions are that cars and trucks can only be moved in their length direction, only sliding movements are allowed (so no picking up), and no movement through another car/truck is possible. The level of difficulty of the game is dependent on the initial configuration of the board and its cars and trucks. The amount of cars and/or trucks used can be changed for each game, as well as the initial position of the cars and trucks. Additionally, beside the 6x6 board size, alternative board sizes are possible.



Figure 1 Small-scale 6x6 version of Rushhour, with both cars as trucks. Both cars and trucks are only allowed to move length-wise is a , cannot be picked up, and cannot move through anohter car/truck. If an adjacent position in front or behind the car is free, the user can move the car to that spot. The game is solved if all the cars and trucks are moved in such a way that the red car can exit the board at the opening on the right side.

**Goal**

The goal for this research is to find the minimum amount of moves necessary for solving 7 different Rushhour boards which are presented in the appendix. These 7 boards vary in both board size, number of cars and trucks used, and the position of these cars and trucks. In the original Rushhour game, cars and trucks can move as many fields as possible in one round, yet the extra restriction in our game is that cars and truck can only move one field per round.

**State space**

The total state space of a Rushhour game with a certain initial board configuration is dependent on the size of the board, the amount of cars and trucks used, and the position of these cars and trucks. Yet the actual state space is difficult to determine. The maximum theoretical state space can however be determined and is formed by the total number of positions on a Rushhour board and the number of positions taken up by either cars or trucks. Therefore, the state space for every different board in the Rushhour game can be different. The formula for the maximum theoretical state space is presented below.

*Max. theoretical state space Rushhour board* =

*n is the number of positions on a Rushhour board (in a 6x6 that would be 36)*

*k is the number of positions that are taken (n – empty spaces on the Rushhour board)*

One of the factors that is not taken into account by calculating the maximum theoretical state space of a Rushhour board is that car/truck cannot potentially be positioned at all board positions. In some boards, some positions will never be filled by any car/truck, decreasing the value of n in the formula. The state space will therefore decrease in size.

For example, board number 1 is a 6x6 Rushhour board, with 21 filled positions (n = 36, k = 21). The max. theoretical state space of this board is 5,567,902,560. Yet, figure 2 shows that positions 2,2 and 5,2 will never be visited by either a car or truck, decreasing the number of possible positions on the board with 2, giving an n = 34 and k = 21. The maximum theoretical state space is then 927,983,760.

In order to tackle



Figure Rushhour board number 1 is presented. The red crosses present field positions on which no car or truck will ever visit. Not taking these positions into account in the formula for the theoretical state space, results in a lower value of the theoretical state space of this board.

# 2. Methods

For this research three different algorithms are used, namely a Random search, a Breadth-first search and a Depth-first search. Each of these algorithms is discussed below. The ‘Determining moves’ part of the algorithm is explained separately and referred to in each of the algorithms’ s explanations, as this part of the algorithm is the same for all three algorithms.

## 2.1 Random search

The Random algorithm explores new board states by randomly choosing a move, until a solution is found. Random search is implemented to compare the other algorithms.

The algorithm starts at the initial board state. For this board state all possible moves for this board state are determined according to ‘Determining Moves’. Using a random function, one move is chosen from all the possible moves. The move is made and the board state is updated with this new move. If this board state is the solution, namely the red car is positioned in front of the exit, the algorithm stops and a solution is found. If the new board state is not the solution the algorithm repeats itself with the new board state and goes back to the step where all new possible moves are created for the board state.

Extra features for random search include, running the algorithm a certain amount of times, each and remembering the amount of steps necessary for a solution. The next time the algoritm only runs as far as the amount of steps in memory. If a solution is found before this, this solution will be remembered.

## 2.2 Breadth-first search

The Breadth-first algorithm explores all possible board positions, depth layer after depth layer. The initial starting board position is at layer 0 and all the board states made with the moves possible from the starting board are in the next layer, and this process is repeated for all boards in each layer. The benefit of using Breadth-first search is that the first solution found is certainly the minimum amount of moves in which the initial Rushhour board can be solved. The downside of Breadth-first search is that the number of boards states to be searched increases more or less exponentially, which makes Breadth-first search a rather slow algorithm.

The algorithm starts at the initial board state, this board is at depth level 0, and the amount of board iterations is also 0. The initial board is in this case the parent board state. For the parent board state all possible moves are determined according to ‘Determining Moves’. The algorithm then iterates through all of the moves. Per move a board state is created, in which the move is made, and the rest of the board is exactly the same as the parent board state. This board state is a child of the parent board state and is given additionally given the value of the next depth layer. At this point, three possibilities exist. Firstly, if the board state is the solution, namely the red car is positioned in front of the exit, the algorithm stops and a solution is found. Secondly, if the new board state has not occurred before and is not the solution, the new board state will be remembered together with its depth layer, one depth deeper than the parent board state, and the algorithm continues. Thirdly, if the new board state has occurred already, the board is not remembered, and the algorithm continues. Once all moves have been iterated through, and all the children board states have been created and memorized for this initial board, the algorithm continues. From memory, the first board positioned in memory will be obtained and is the new parent board state. The amount of board iterations increases with 1. The algorithm then continues accordingly obtaining all moves for this new board.

Once a solution is found, the depth level of the board state added with 1 presents the minimum amount of steps in which the initial board state can be solved, and the number of board iterations represents how many different boards the algorithm has evaluated.

## 2.3 Depth-first search

The Depth-first algorithm searches the tree of all possible board positions by exploring all possibilities in one branch first. If at the end of the branch, the algorithm backtracks and starts to search the next branch. The benefit of using depth-first search is that specifically searches for solutions, while breadth-first search searches all the possibilities. Therefore, Depth-first search can be a fast algorithm, yet the disadvantage is that there is no assurance that the solution found is the minimal steps possible.

The algorithm starts at the initial board state, and all board states in this algorithm are remembered. For this board state all possible moves for this board state are determined according to ‘Determining Moves’. If there are moves the algorithm continues. Using a random function, one move is chosen from all the possible moves, and the board state is updated with this move. The move made is remembered. If this board is the solution, namely the red car is in front of the exit, the algorithm stops. If the new board is not the solution, the algorithm goes back to finding new moves for this new board state and continues accordingly.

If either a new board state has occurred before or if no moves can be made from a board state, the algorithm goes one step back in the branch by reversing the last move made from the board position (which was remembered). This last move made is then deleted from the memory. With the board the algorithm is at, the algorithm goes back to finding all possible moves, and continues accordingly.

**Determining moves**

In all our algorithms possible moves have to be determined for every single board state that occurs, and is done as follows. For every car and truck in the board state, both the position in front and the position behind that car is checked. If this position is free, so no other car or truck occupies this position and the car/truck is not at one of the sides of a playing field, a move is possible. This move is remembered. If the position is not free, no move is possible.

**Making moves**

The board state is updated with the given move, so this position is presented as occupied. The position on the board that was first occupied is now presented as free.

# 3. Results

Results have been found for all algorithms, yet not all encompassing. Even with a running time of 9 hours, the algorithm Breadth-first search could not present data. Therefore, a simple test board (6x6) was made as well, in order to validate accuracy of the Breadth-first search algoritm. This

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Board | Max State space | Number of solutions | | |  | Minimal Moves | | |  | Iterations  *(Random: # of executions)*  *(BSF/DSF: boards visited)* | | |
|  | **theoretical** | **Random** | **BFS** | **DFS** |  | **Random** | **BFS** | **DFS** |  | **Random** | **BFS** | **DFS** |
| (6x6)1 | 5567902560 | 10 | - | 1 |  | 51 | Not Found | 2043 |  | 1000000 | 7165 | 902 |
| (6x6)2 | 94143280 | 7 | - | 1 |  | 71 | Not Found | 192 |  | 1000000 | 3430 | 143 |
| (6x6)3 | 94143280 | 8 | - | 1 |  | 37 | Not Found | 667 |  | 1000000 | 2843 | 234 |
| (9x9)4 | 2.30628E+21 | 14 | - | 1 |  | 323 | Not Found | 7860 |  | 223264 | 1593 | 1372 |
| (9x9)5 | 2.30628E+21 | 15 | - | 1 |  | 276 | Not Found | 4987 |  | 430944 | 1321 | 672 |
| (9x9)6 | 1.36362E+19 | 18 | - | 1 |  | 95 | Not Found | 5306 |  | 1000000 | 826 | 641 |
| (12x12)7 | 6.60504E+35 | 16 | - | 1 |  | 818 | Not Found | 7384 |  | 90574 | 624 | 576 |
| (Test)8 | 30260340 | 9 | 1 | 1 |  | 6 | 6 | 12 |  | 1000000 | 98 | 9 |

board gave results for all algorithms.

Tabel This table shows the results per Rushhour board the theoretical maximum state space and per algorithm the number of solutions found, the minimal moves found and the iterations it took to find these minimal moves. The value at iterations for random moves presents the amount of times random moves is run, in order to obtain the value at minimal moves. For BFS and DFS the iterations represent the amount of boards visited to obtain the value at minimal moves. The minimal moves are the minimum amount of moves the algorithm could find. For the BFS this means that this is also the actual minimal moves to find a solution. The number of solutions at the Random algorithm presents the amount of times the algorithm found a solution at the number of iterations given.

Random

BFS

DFS

Comparing Random to BSF

Comparing Random to DFS

Comparing DFS to BSF

* BFS: no solutions found even after 6 hours. The results presented in the table are
* Random has a lower minimal move for

# 4. Conclusies

*Hier schrijf je je conclusies, eventuele overdenkingen (hoe zou het nog beter kunnen, is het algoritme ook in andere gebieden toepasbaar).*

The goal of this research was to find the minimum amount of moves necessary for solving 7 different Rushhour boards. Using all the algorithms created in this research, Random, Breadth-first search and Depth-first search, solutions have been found for the 7 Rushhour boards. However, the goal of finding the solution which expresses the minimum amount of moves has not been reached. This is mainly because our algorithm Breadth-first search is slow. This algorithm was supposed to provide with certainty the minimum amount of moves, yet this was only achieved for a simple test board. After Breadth-first search, the Random search comes close to giving an indication of the minimum moves, which is or alike or lower than the solution given by Random search. Depth-search is not very reliable for presenting a minimum amount of solutions, except perhaps if the user would run it as many times as Random search is run in the Random search algorithm.

So, basically the Random search algorithm is fast and has reasonably good results regarding to finding solutions with low amount of steps necessary. Breadth-first search algorithm is the best algorithms regarding our research goal. However, this algorithm is too slow and would need to be adapted or combined with another algorithm for better results. Depth-first search is not optimal, but is capable of finding a solution fast. Yet, this solution is not necessarily close to a minimal amount of moves.

Various possibilities exist to improve the algorithms and to find a solution with the lowest number of moves. Firstly, an A \* algorithm could be implemented. The cost in this algorithm could be the amount of cars that are in between the red car and the exist, and whether or not these cars are blocked by other cars. Secondly, pruning of the current algorithms could be improved, determining board states from which an optimum solution (minimum moves) will probably not be found. Finally, an algorithm we came up with ourselves could be implemented. This algorithm starts with the red car and checks if the position in front of it is occupied. If this is not the case, the (red) car moves to that position,and the evaluation if the position in front of the car is occupied start again. If the position is occupied, the focus goes to that car. For this car the position in front of it is checked, and the according steps are followed. The car is moved if empty, and the next car is evaluated if not empty. If the car has moves, the algorithm backtracks goes back to the car before and checks again if this car can move

## 6. Appendix



Board 1, 6x6.

Board 2, 6x6.

**

Board 4, 9x9, 6x6.

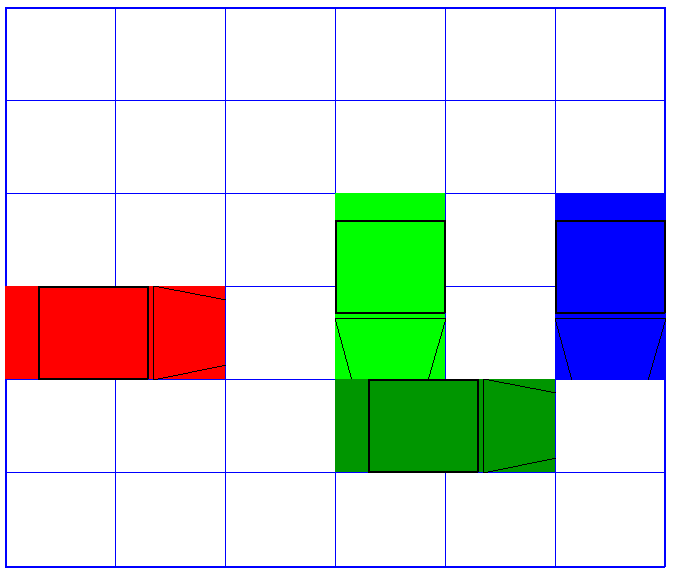
Board 3, 6x6, 6x6.

**

Board 6, 9x9, 6x6.

Board 5, 9x9, 6x6.

Board 7, 12x12, 6x6.



Board 8 (test board), 6x6, 6x6.