[]: Een snel algoritme voor het spel Rushhour

Oscar Keur\*

Universiteit van Amsterdam

emailadres@email.uva.nl

Kyra Kieskamp\*

Universiteit van Amsterdam

Kyra.kieskamp@student.auc.nl

*Hier schrijf je een inleiding die in elk geval kort, bondig en compleet de hele vraagstelling bevat. Ook moet er een inschatting komen van de toestandsruimtegrootte (belangrijk, daar deden we het voor), en eventuele restricties op transities in beschreven worden, maar nog niet de methodes die je gebruikt om de toestandsruimte te doorzoeken.*

# 1. Inleiding

Rushhour is a sliding block puzzle created in the seventies by Nob Yoshigahara. (more explanation)

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 boardplaces 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 untill 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 dependend 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.



Figuur 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 is to find the minimum amount of moves necessary for solving the 7 differerent rushhour boards (see appendix). These 7 board vary in both boardsize , 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. The 7 different boards are attached in the appendix.

**Statespace**

The total statespace of a rushhour game with a certain initial board configuration is dependent on the size of the board, and the amount of cars and trucks used, and the position of these cars and trucks. Basically, the statepace is formed by the amount of possible steps , so the amount of empty spaces in the board and the possibility of cars/trucks to move to these spaces. Therefore, the statespace for every different board in the rushhour game can be different. The formula for the maximum theoretical statespace, is presented below.

*Max. theroetical statespace 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 statespace is that car/truck cannot potentially be positioned at all board positions. This will decrease the statespace. In some boards, some positions will never be filled by any car/truck, decreasing the value of n in the formula. The statespace 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 statespace 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 statespace is then 927,983,760.

# 2. Methodes

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

**Determining moves**

In all our algorithms possible moves have to be determined for every single board state that occurs, this 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**

## 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 This is a rather (explain why using random search)

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 itsself 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, remembering the minimum possible of steps found by the algorithm after running the algorithm x amount of time, and determine a maximum amount of iterations the algorithm is allowed to make.

## 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 rush-hour 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 occured 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 occured already, the board is not remembered, and the algorithm continues. Once all moves have been iterated through, and all children boards 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 bord 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 Depth-first search algorithms works as follows. 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 conintues. Using a random function, one move is chosen from all the possible moves, and the board state is update 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 occured 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.

# 3. Resultaten

*Hier bespreek je heel droog je resultaten. Als je statistieken hebt: toevoegen. Als je vergelijkingen hebt met randomposities: toevoegen. Alles is woord en getal, alle details en het liefst ook het één en ander in grafieken, plaatjes of anderzins.*

Fig. 2. Aantal misfits per algoritme, gemiddeld over 5 trials. Ondanks dat alle methodes de puzzel relatief snel oplosten, is het BIDIBENCH algoritme zowel qua eindresultaat als qua algehele trajectperfomance te prefereren over beide andere methodes. *Een goede grafiek is, net als een goed plaatje, in één oogopslag duidelijk en vereist weinig arbeid van de lezer. Voorzie de assen van labels, de grafiek van een titel en kies je kleuren zo dat ze maximaal contrasteren en dus gemakkelijk leesbaar zijn. De legenda rechts van deze grafiek had rechtsbovenin het plaatje gemogen, dat had ruimte bespaard, had de grafiek iets groter kunnen zijn en dus beter leesbaar.*

**(2 uur)**

# 4. Conclusies

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

**(1 uur) – conclusie + eventuele extra algoritme**

# 5. Referenties

*Als je literatuur hebt gebruikt, hier toevoegen. Als je eraan refereert in de tekst, zet je op die plek alleen [1], zodat mensen achterin de details kunnen vinden. Als je geen literatuur gebruikt, weglaten.*

*Wat ook nog kan is een dankwoord, bijvoorbeeld voor mensen die wel geholpen hebben maar geen auteur zijn, mensen die je een inzicht hebben gegeven, of administrators die je even hun supercomputer hebben laten gebruiken. Altijd naam en bedrijf noemen en zorgen dat de bedankte persoon zich er goed over voelt.*

*Als je zowel een dankwoord als een referentiesectie hebt: de referentiesectie is \*altijd\* het laatste onderdeel van je verslag.*

[1] Artificial Intelligence, a modern approach, Russel & Norvig,3rd Edition, Addison-Wesly, pg 287-387.