**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**

For the course Programming Theory (Heuristics) at the University of Amsterdam team Spitsuur has chosen the case Rush-hour. The case Rush-hour seemed interesting to solve, because it was a well-known game played by team Spitsuur and it was exciting to find out how a board could be solved in as little as possible steps.

**1.1 The Case**

The Rush-hour puzzle exists of a six by six grid which has one red car; a number of other cars, depending on the difficulty the number of other cars vary; and an exit. The red car is always horizontally placed on the same level as the exit and it always occupies two grids. The length of the other cars can vary from two to three grids and can be placed horizontally or vertically on the board. The goal of this puzzle is to move the red car to the exit by moving the other cars, which are blocking the red car, out of the way. The puzzle has, however, three constraints. The first constraint is that a horizontally placed car can only move horizontally and a vertically places car can only move vertically. The second constraint is that a car cannot rotate and the third constraint is that the cars may not overlap while making a move.

This problem is considered a constrained optimization problem, because there can possibly be a number of solutions to solve a board, however, not every solution is equally optimal.

The goal of the case is to move the car to the exit with as little as possible steps. A step can be interpreted in two ways. These ways are explained with the help of the figure below. Moving the orange car opposite of the red dot to the red dot can be interpreted as one step or as four steps. In other words: the number of steps for moving a car as far as it possibly can, get can be interpreted as the number of grids it used to move or as one step. Team Spitsuur has decided to interpret the number of steps as the number of grids the car used to move. This results in a higher number of steps used to solve a puzzle than when interpreting the car its far as possible move as one.

The case has provided seven puzzles to solve. Three of these puzzles are six by six and vary in difficulty, another three are nine by nine and also vary in difficulty and one of these are twelve by twelve, which is the hardest one to solve. The goal team Spitsuur has set is to create a computer program that solves at least the three six by six board in as little as possible steps and in as little as possible time.



**1.2 The state-space of rush hour**

The state-space size of rush hour is considered infinite, because a car can always move from left to right and back again. Also the state-space of each rush hour puzzle can differ, because it depends on the dimensions of the puzzle and the number of cars. To calculate the state-space the following formula is formed:

1) S = State-space

2) d = puzzle dimension

3) c = number of cars

4) t = number of trucks

S = (d - 1)c x (d-2)t

This formula calculates an overestimation of the state-space, because the constraints of the game are not considered.

**2. Methods**

To solve this puzzle and to reach the goal set, team Spitsuur has written a computer program in Python 2.7, which is called the skeleton and a computer program with algorithms to solve the puzzle. The reason for choosing Python 2.7 is because team Spitsuur was already familiar with this programming language.

**2.1 The computer program - Skeleton**

The computer program, rush.py, exists of two classes and another program, rushvisua.py, to visualize how the boards are being solved.

The program rush.py has two classes, namely, the Board class and a Car class. The Board class gets a .txt file as input and parses this information into a dictionary with all the existing car’s ids as key and the car’s position, meaning which coordinates on the board the car occupies as values in a tuple. The Board class has the full knowledge of all the cars on the board and distinguishes between the red car and all the other cars. This class further has knowledge of the dimensions of the board, where the exit is and where the empty spots are i.e. grids that are not occupied by a car.

With this class it is therefore possible to check whether a car can be moved. If the case is that the car can be moved because there are empty spots near the car, it is possible to move the car up, down, left or right depending on their direction, namely horizontally or vertically. If a car is placed horizontally the car can only move to the left of right and if a car is placed vertically it can only move up or down. After moving a car a new Board object is created with in the dictionary the new positions on the board.

The Auto class has knowledge of all the cars its properties such as the length of the car, the color of the car and the direction of the car, horizontally or vertically.

The computer program rushvisua.py visualizes the generated Board objects by creating a board which has the number of grids belonging to the dimensions given in the .txt file and an exit. After creating this, the cars are generated and placed on the given positions. If the color red is specified behind a car in the .txt file, that car will be colored red and the other cars will have a random color. After this the dictionaries of each board object are read by this program and are being visualized, resulting in moving cars and the red car being moved to the exit.

To test if these programs were working properly team Spitsuur has generated a .txt file with small dimensions and few cars.

**2.2 The algorithms**

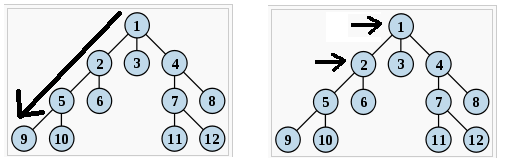
After the skeleton was build and was working properly the provided puzzles could be solved. To efficiently solve these puzzles, algorithms were implemented into the program.

**2.2.1 Depth-first search with pruning**

The first algorithm implemented was depth-first search. Depth-first search is an algorithm which searches each branch of a tree as far as possible, before going back to search in another branch. This algorithm was implemented recursively. Team Spitsuur decided to use pruning , this means when a solution is found for the puzzle at a certain depth the algorithm will only look for solutions (in other branches) before that depth until the algorithm is done searching the tree. To prevent infinite loops, because the algorithm can move a car for example one grid to the left and then again back and again to the left and again back, each Board object with its dictionary that was visited is being saved in a Python set.

**2.2.2 Breadth-first search**

The second algorithm implemented was breadth-first search. Breadth first search is also an algorithm to search trees, however this three starts at a node and checks all its children. Instead of searching in each branch in the depth, it searches in the width of the tree. The figure below can make this explanation more clear. First the algorithm checks node 1, in the rush-hour case this would be the first board. Then it inserts all the children, namely 2, 3 and 4, in a queue. In the rush-hour case this would be all the boards with the cars on different positions and adds these in to a Python queue. Then the first child is taken out of the queue and it again produces all the children of this child and inserts these to the back of the queue and so on until a solution is found. The advantage of this algorithm is that if a solution is found it is the optimal one, the disadvantage is that it takes a lot of memory to store all the boards in the queue. To prevent the algorithm from searching two branches that are the same, the visited boards are also saved in a set. Also the path to the solution is saved.



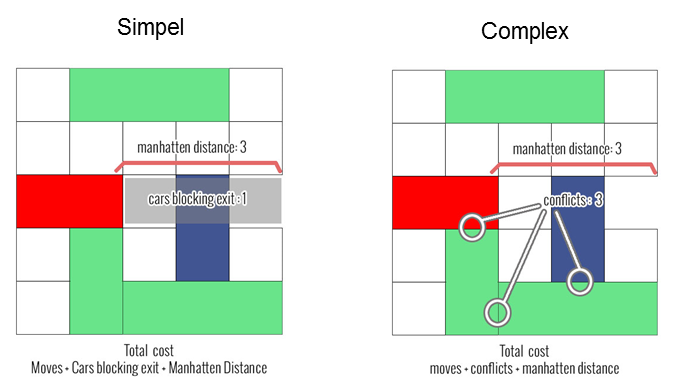
**2.2.3 A\* algorithm**

The third algorithm implemented is A\*. The A\* algorithm is also used to find an optimal path to a certain goal with the help of a total cost formula f(x). The formula f(x) exists of g(x) and h(x) in which g(x) is the distance from a node in the tree to the parent node and h(x) is a heuristic value, for example, the Manhattan distance (crow flies) to the goal.

The implementation of A\* for the rush-hour case, is a bit different than the traditional A\*. A\* is not considered fitted for a puzzle board game, therefore team Spitsuur has rewritten the A\* to fit the puzzle board game. The total cost function f(x) is the Manhattan distance, the distance from the exit of the red car, added to the number of steps already taken, added to a heuristic value.

Team Spitsuur has created two heuristics for the A\* algorithm. The first heuristic is considered a simple heuristic which adds one to the total cost function for each car that blocks the red car. This was considered, because the cars blocking the red car should be moved out of the way, before the red car can move closer to the exit.

The second heuristic is considered a more complex one, because this heuristic does not only validate the cars blocking the exit, however, also the cars that are blocking these cars (conflicting cars). This heuristic was considered because the conflicting car should be moved out of the way before the cars blocking the exit can move out of the way for the red car to move closer to the exit.



**3. Results**

The results of the used algorithms are set forth in this paragraph.

**3.1 Depth-first with pruning**

When testing the algorithm on a self-made four by four board with three cars the algorithm would find a solution, however when testing one of the provided six by six boards, the algorithm did not even find a solution within 2 minutes. Therefore team Spitsuur did not even try to test the nine by nine boards with this algorithm. The algorithm was found to be very slow.

**3.2 Breadth-first search**

When testing this algorithm on the same self-made board as depth-first search, the solution was found in less than a second. When testing the other boards it was found that the optimal solutions where found for puzzle 1, 2, 3 and 4. The runtime for finding the solutions are depicted in the table below. For puzzle 5 until the last puzzle 7 no solutions where found due to an memory error.

**3.3 A\* algorithm**

A\* simple and A\* complex where also tested on the puzzles. A\* simple and complex could find the optimal solutions for puzzle 1, 2, 3 and 4. On the other puzzles also a memory error occurred. The run time of the algorithm with both heuristics are also depicted in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Depth-first search | Breadth-first search | A\* simple | A\* complex |
| Puzzel 1 6x6 |  | 1.267s | 3.497s | 8.040s |
| Puzzel 2 6x6 |  | 0.533s | 3.566s | 2.010s |
| Puzzel 3 6x6 |  | 0.058s | 0.244s | 0.503s |
| Puzzel 4 9x9 |  | 128.994s | 308.665s | 253.000s |
| Puzzel 5 9x9 |  |  |  |  |
| Puzzel 6 9x9 |  |  |  |  |
| Puzzel 7 12x12 |  |  |  |  |

**4. Discussion**

Although the runtime of an algorithm is a good method to compare one algorithm with another, team Spitsuur does not think the runtime of only one algorithm is saying 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 finish quicker than running it on a computer with slow performance level and a slow compiler.

Also the way the skeleton of the program is build is important for the performance of the algorithms. If the skeleton contains a lot of iterative functions, and big hash functions the runtime of the algorithms will increase.

The A\* performed worse than breadth-first, because the heuristics where not found to be smart ones, namely distance of the red car to the exit. It is assumed that when using a smarter heuristic the A\* might perform better than breadth-first search. The downside of using the heuristic in the implementation of the rush-hour case is that it might search first through all the lowest costs, however, the solution might not be in that path. Therefore the algorithm will perform worse than breadth-first.

**5. Conclusion**

For the course Programming Theory (Heuristics) at the University of Amsterdam team Spitsuur has chosen the case Rush-hour to solve. The case’s goal is to move the car to the exit with as little as possible steps. A step can be interpreted in two ways. To solve this puzzle and to reach the goal set, team Spitsuur has written a computer program in Python 2.7, which is called the skeleton and a computer program with algorithms to solve the puzzle.

To efficiently solve these puzzles, the algorithms depth-first search with pruning and breadth-first search were implemented into the program. The depth-first search with pruning algorithm was found to be very slow. The breadth-first search algorithm was found to be faster than depth-first search. The A\* algorithm performed worse than the breadth-first algorithm on most boards, however better than the depth-first algorithm. A\* with the complex heuristic performed better than the simple heuristic on puzzle 2 and 4.

Although the runtime of an algorithm is a good method to compare one algorithm with another, team Spitsuur does not think the runtime of only one algorithm is saying much about the performance of the algorithm itself. Also the way the skeleton of the program is build is important for the performance of the algorithms. A\* might perform better if a smarter heuristic is used.