NTNU – Trondheim
Norwegian University of
Science and Technology

Artificial Intelligence programming IT3105

Report from Homework Module #1:
“Using the A* Algorithm to Solve Rush Hour Puzzles”

By Jakob Svennevik Notland
Fall 2017

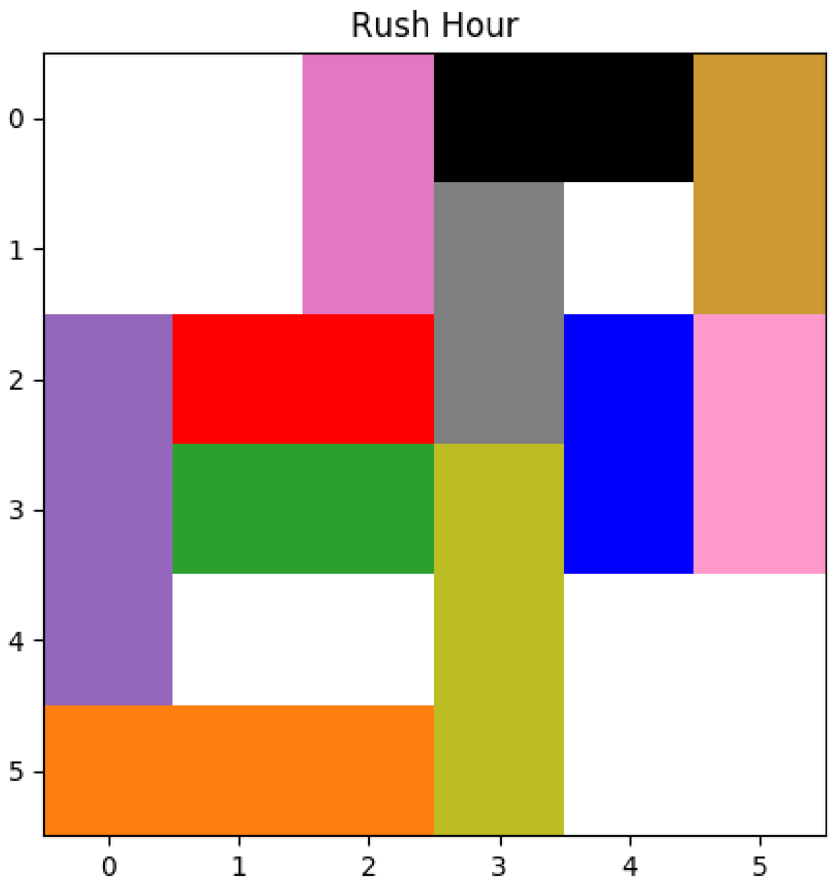

Figure 1: Sample of the view from the rush hour implementation

This is a report for the assignment where one shall create an implementation of the A-star algorithm to solve the "Rush-Hour" problem. One shall also implement breadth-first-search and depth-first-search for comparison. This report will contain a description of the heuristic function used with A-star. Also a description of the procedure to generate successor states when expanding a node. And in the end a comparison of the three algorithms.

On the front page there is a sample screenshot from my graphical view. Where the red car always is the vehicle in focus, and the goal is to get out at square [5, 2]. To run the program one must enter command line arguments. The syntax is as follows:

```
$ python3 astar_rush_hour.py <algorithm> <board> <display_solution> <display_agenda>
<display_time>
```

Example:

```
$ python3 astar_rush_hour.py AStar easy-3.txt true false 1
```

Where the <algorithm> can be "Astar", "DFS" or "BFS". The <board> is a text file containing the board. <display_solution> and <display_agenda> are optional parameters which takes a boolean value. And has their default values as false. The last parameter <display_time> is a float where one can specify how many seconds to display each frame in display mode. Also the program will always print console output with the current node expanding, total nodes expanded and path length for the solution.

1. Heuristic function

The heuristic function used for this solution is quite simple. It can be located in the Board class in the function named "calculate_heuristic". The board object keeps track of the heuristic value "h", which is calculated every time a node is discovered. First off the function adds 1 to "h" for every space between the target car and the goal. Which in figure 1 would be 3. Then the function adds 1 to "h" for every car blocking the road between the target car and the goal. Which in this case also is 3. This makes the total heuristic for this state 3 + 3 = 6. The calculation can be written is pseudo code:

```
def calculate_heuristic:
    h = goal_state.x – vehicle.x - vehicle.size + 1
    for car in cars blocking the road:
        h += 1
    return h
```

This heuristic will be an underestimate because the vehicle must at least move itself to the goal, and remove conflicting vehicles. Which can take several moves. Never the less, this makes the heuristic admissible. To improve the calculations I could be adding more complicated heuristics. E. g by trying to estimate how many moves the conflicting vehicles need to get out of the road.

2. Generate successors

The process of generating successor states happens every time a node is popped from the open list in the "best_first_search" function in the RushHour class. The main function used to expand a node is named "expand_node", and is part of the Board class. The function first uses another function "get_legal_moves" to get a list of tuples containing pairs of vehicles and legal moves. Then “expand_node” calls for “expand_move” for every legal move. The “expand_move” function then makes a copy of the current state, performs the move and returns the new state. In the end, a list of children is returned to the “best_first_search” function. Following is some pseudo code to show the process of the main function.

```
def expand_node:
    legalMoves = get_legal_moves()
    children = []
    for move in legalMoves:
        children.append(expand_move(move))
    return children
```

3. Performance comparison

	Puzzle variants			
Search method	Easy-3	Medium-1	Hard-3	Expert-2
Breadth-1st	(111, 16)	(804, 24)	(2040, 33)	(10941, 73)
Depth-1st	(103, 45)	(1497, 281)	(1824, 473)	(10463, 2486)
Best-1st (A*)	(83, 16)	(630, 24)	(897, 33)	(6059, 73)

Figure 2: Comparison of A*, depth-first and breadth-first search on 4 rush-hour puzzles. Pairs in parentheses are the node count and number of moves (from start to goal), respectively

The performance comparison in the table above shows the differences between running a search with A Star, BFS or DFS. As expected given that my heuristics are an underestimate, the A Star algorithm is superior at finding the best path. Also one can see that the procedure requires much fewer nodes expanded than BFS, which also finds an optimal solution, but much slower. The depth-first-search might get lucky and find the solution quite fast. However, because DFS is a very unintelligent algorithm, it mostly finds bad solutions (long paths). And also if DFS starts on the “wrong” side of the search tree, it can take very long time to finish. Another algorithm which is not intelligent is BFS. Because usually, the larger the search three becomes, the longer time it takes for the algorithm to find the solution. Because it is creating so many successor states.

Something I have noticed is that my results differs from the results presented in the assignment paper (except from the optimal solutions). That is probably a result of me implementing the algorithms slightly different such that it chooses another child in general when using DFS or BFS and when the heuristics are equal in A Star.