Rush Hour
Lab rapport IT3105 - Module 1

Sigve Skaugvoll & Thomas Wold

September 2017

1 Introduction

The assignments was to solve rush hour puzzles using A*, where we had to implement everything from scratch. So part of the problem was to find a good representation of the problem state, find a good heuristics and create a simple GUI which shows the solution, how many nodes that are generated and number of moves from start to goal.

2 Rush hour

In this section, we will give answers to the following questions:

1. A clear, concise description (using mathematical expressions and text) of the heuristic function (h) used to solve the puzzle.
2. A thorough description of the procedure used to generate successor states when expanding a node.
3. A comparison of A*'s performance to that of both depth-first and breadth-first search when applied to each of the 4 puzzles, easy, medium, hard and expert.

Our implementation of the A* for this module, does not use different heuristics, or diffetent functions to generate successors of a node. Thus for all 4 puzzles handed out, will have the same answers, except for comparison when compared to Depth-first Search and Breadth-first Search.

2.1 Q1: Heuristic function

The heuristics implementation used, counts how many cells/ moves needed to drive in a straight line from the current position of the playing-piece (not obstacle piece) and to the goal cell. It also counts how many of the cells on the path to the goal has another car in it. This blocking cars needs at least one

move. The sum of these two variables are the estimated heuristics to reach goal state. Thus the heuristics will never overestimate, but try and give so close to exact estimation of minimum moves needed to reach goal state. To calculate the straight line, we use Manhattan distance. This because the playing piece has to be on the same row as the goal. We also count from where the car ends. So we take goal column - (car column + (size - 1)) to get number of moves if no ones blocks the "road".

The h(x) will thus be admissible, since it never over estimates, but close to the number of moves needed to reach goal state.

2.2 Q2: Generating successors

When generating successors, we pass in the current node, which represent the entire current game state. Then for each piece on the "board", we check if it's a horizontal or vertical piece. If it's horizontal, pass it in to a function that tries to move in it's allowed direction (left and right). The first thing the if allowed function does, is to check if the piece is on either of the board edges, if left edge and wants to go left, return false. Else, we just check if any of the other pieces around the piece we want to move, starts, expands or ends in the cell we want to move the piece. If no piece is placed around, then the move is legal, return true. Then this move represents a new search state.
A new search node object is created that represents the new search state found. The new node is then appended to a list of all legal successors of the current search state. When found all legal new search state / moves, we return the list with all generated successors. The next thing that happens is that for each new successor, we check if it has been generated / discovered previously and take action accordingly.

3 Comparison

Algorithm vs Boards				
Algorithms	Easy	Medium	Hard	Expert
A*	(73, 16)	(594, 24)	(856, 33)	(6082, 73)
BDF	(96, 16)	(752, 24)	(1706, 33)	(10144, 73)
DFS	(45, 42)	(591, 278)	(955, 795)	(2772,)

Each result in the table shows how many nodes that are visited and how many steps it took to get from start to goal for each algorithm and board. These numbers are represented on the format: (visited, steps).

4 Conclusion

Through this lap exercise we have implemented a generic Astar.py class, which can take be used to solve alot of different problems, such as Rush Hour, nonograms, etc. We created abstract classes for templates, which problems that want

to use our A*, can inherit from and implement the problem specific domains. Thus if the problem classes such as problemBFS, ProblemNode inhereits from our abstract classes. A star can solve all problems. The A* class it self can be inherent and thus a subclass of it can be modified.

The classes involved for solving this lab exercise is:

1. AstarClass.py - Generic A* implementation.
2. RushHourBFS.py - inherits from abstract BFSclass.py
3. RushHourNode.py - inherits from abstract NodeClass.py
4. RushHourGui.py - (optional) Creates a GUI which shows steps and solution found.

References

1. AI Intro (IT-3105) Homework Module #1 - Using the A* Algorithm to Solve Rush Hour Puzzles - Author: ? - Pub: August 4, 2017 - Accessed : September 20, 2018 - Online.
2. Essentials of the A* Algorithm - Author: Keith L. Downing - Pub: ? - Accessed : September 20, 2018 - Online.