Vilde Schliekelmann Barth & Vilde Graven Stokke
IT3105 – Artificial Intelligence Programming

Module 1

Using the A* Algorithm to Solve Rush Hour Puzzles

Introduction

The purpose of Module 1 is to solve Rush Hour puzzles using best-first search with the A* algorithm. We have chosen to write the solver in python, and used pygame for visualization of the path from root to final solution.

We have created one superclass *Astar* which contains the highly general A*-algorithm. To specifically handle Rush Hour puzzles, we created a subclass of Astar, *RushHour(Astar)*. There we've placed the problem-specific heuristics and check for solution, *isSolution*, as well as calling the general A* function.

In order to speed up the algorithm, we've used hash functions to store the nodes in dictionaries. Each node/state is therefore identified with a tuple containing the positions of each vehicle in the current state, which is a unique value for each state.

Node

Each node represents a unique state of the board. The nodes have the following attributes:

- **parent**: node the current node was expanded from (which is the one with least cost - if the state is reached through various paths)
- **children**: all nodes created when the current node is expanded
- **h:** heuristic value for the node
- **g:** cost from the root node to the current node
- **f:** sum of h and g
- **state:** represents if the current node has been expanded or not. True if not expanded and still in the *opened* queue, false if expanded and in the *closed* list
- **positions**: positions of the vehicles in current state, unique for every node
- **board**: board representation used when expanding a node, to find empty cells and possible moves
- **prevBoard**: board of parent node. Used when expanding a node in order to not move back to previous board

Heuristic

The heuristic used to solve the Rush Hour Puzzle is based on the distance of the main car (car-0) from the goal and the number of cars blocking its way to the goal. We have used the same heuristic function for all 4 puzzles. Below you can see the code for our heuristic function, with the following attributes:

- **positions:** list of positions for each vehicle in a given node (state), unique for each state
- **constantPos:** position of each vehicle that is fixed, depends on orientation of vehicle
- **goal:** goal position of car-0
- **board:** board in a current state. If cell has value '.' , the cell is not occupied, if it is occupied it has the number of the vehicle occupying it
- **lengths:** lengths of the vehicles

Our heuristic is based on

$$f(node) = g(node) + h(node)$$

where g represents the cost so far and h the estimated, admissible heuristic value to reach the goal. The arc-cost is defined as being 1.

Our chosen heuristic (for car-0) is

$$h(node) = distance\ from\ goal + \sum cars\ blocking$$

Shown in code:

```
def heuristic(self, node, goal):
    h = 0
    h += goal - node.positions[0]
    for i in range(node.positions[0] + static.lengths[0], len(node.board)):
        if node.board[static.constantPos[0]][i] != '.':
            h += 1
    return h
```

Expand

When expanding a node, we create a new state (node) for each possible move from the current state. A for-loop iterates over all vehicles and checks whether they can move up or down/left or right, depending on their orientations. For each of these possible moves, we generate a child node, setting the current board as its parent. The child will always differ from the parent with a move of 1 for one of the vehicles. When moving one of the vehicles, we also update the position in the child node's position list for the given vehicle. Expand returns a list of the current node's generated children.

Comparison of A* to BFS and DFS

	Puzzle Variants			
Search Method	Easy-3	Medium-1	Hard-3	Expert-2
Breadth-First	(95, 16)	(751, 24)	(1705, 33)	(10143, 73)
Depth-First	(88, 86)	(659, 478)	(1749, 1536)	(3786, 3656)
Best-First (A*)	(68, 16)	(586, 24)	(825, 33)	(5892, 73)

The first value in the above tuples is the total number of nodes generated by the search tree, whereas the second value is the number of moves in the first solution that the method finds. We can easily see that for the simpler boards, the total number of nodes generated and length of solution path is quite similar for BFS and Best-1st search, with A* being slightly better.

When expanding further by using more challenging boards, A* gives a considerably better result. Number of nodes with A* is about halved, yielding a more optimal algorithm for the Rush Hour Puzzle.