Bridget Murphy

Tim Wu

Alexa Furnari

Blockmaze.py

Description:

The purpose of Blockmaze.py is to find the most optimal path from the starting position to the ending position in a maze. A* Search is advantageous in this situation because it is complete and it uses heuristics to guide the search.Therefore, it won't go down a path it knows will not get it closer to the goal. The game piece that moves through the maze is similar to a pair of adjacent dice. It can either stand up, where one dice is stacked on top of the other, or it can lay down horizontally. The orientation of the piece is stored in the node class under the attribute state as a pair of coordinates. The node class also defines f, g and h. H is the cost of our heuristic, g is the cost of making a move which in this case is always one and f is the cost of g + h. The f cost is how A* Search will determine which node to pull off the frontier next. Since it costs the least, this path should be more optimal than the other paths it could have chosen to explore.

In Blockmaze.py the user defines which maze to use in the command line. Blockmaze.py then reads in each line of the maze file as a list and searches for the characters 'S' and 'G'. The program saves the coordinates of 'S' and 'G' in the variables start and goal. Start is later passed into both the A* Search function and it is used to initialize the initial state of the first node. Goal is also passed into A* Search and is used for terminal testing. For all nodes generated after the first node, the node's state, also known as its location and orientation, is tested against the goal.

The legal_actions function takes in a node and tests for all the possible actions the piece can make from its current position. It returns those actions as a list of coordinates. It does this by first checking if the block is standing vertically or laying down horizontally. Then, once the piece's current location and orientation is known, the function checks the positions around it.

The A* Search Algorithm takes in the parameters start (starting position), goal (ending position), mazeLines (2D array), xMax (length of maze), yMax (width of maze) and a heuristic boolean. Two lists, frontier() and explored() are created. The frontier holds all the nodes that are generated, but have not yet been explored. The list is sorted so that the node with the smallest f cost is always the next node to be removed from the frontier. If a node is taken off the frontier and it does not match the goal state, then it is

added to the explored list of nodes and A* Search will move on to finding and adding the node's children to the frontier. A* Search will call legal_actions to find all the possible descendants of the parent node and if the child has yet to be seen in frontier or explored, it is added to the frontier list. The cycle then repeats and the node with the next lowest cost is removed from the frontier. This is where our scaled manhattan distance heuristics comes into play. See heuristic details below.

Heuristics:

The first heuristic used in blockmaze.py is when h(n) = 0. This is a trivial heuristic that supposes that the approximate cost from the current location of the block to the goal is 0. Since the number of moves that it takes to get from that start to the goal will always be greater than or equal to 0 no matter where they are located, this heuristic is admissible.

The second heuristic measures a scaled version of the Manhattan distance from the current location of the block to the goal. This means that h(n) = (the number of squares from the x coordinate of the current block to the x coordinate of the goal + the number of squares from the y coordinate of the current block to the y coordinate of the goal)/2. In order for the heuristic to be admissible, h(n) must be less than the total number of moves that it takes for the block to get from its current location to the goal. Since the block can move a maximum distance of 2 squares in a single move, this means that the normal Manhattan distance will not be admissible, as the block could reach the goal in fewer number of moves than the number of squares between them. To accommodate this constraint, the heuristic calculates the normal Manhattan distance and divides it by 2, ensuring that h(n) is always less than the true cost to get to the goal. Since the block may have two different locations if it is laying down, the heuristic measures the distance from the first block only.

Data:

Maze #	Heuristic	Number of nodes generated	Number of nodes visited
1	h(n) = 0	143	138
1	Scaled Manhattan distance	78	46
2	h(n) = 0	57	58
2	Scaled Manhattan distance	50	43

3	h(n) = 0	213	208
3	Scaled Manhattan distance	88	55
4	h(n) = 0	141	126
4	Scaled Manhattan distance	86	60
5	h(n) = 0	32	27
5	Scaled Manhattan distance	16	13

Results:

The fewer obstacles there were in the maze, the less effective the Manhattan Distance heuristic was compared to the h = 0 heuristic. With the exception of Maze 2, which only has 2 obstacles compared to the four or more obstacles in every other maze, the h = 0 heuristic nearly doubled the amount of nodes generated and nodes visited.

We predicted that the algorithm would run more optimally with the addition of the Manhattan Distance heuristic, however we were surprised that its efficiency was correlated to the number of obstacles in the maze.