

Homework 1 Part 2

Solving Sequence Mazes using Informed Search Algorithms

 $\begin{array}{c} {\rm Mason~Edmison} \\ {\rm University~of~Wisconsin\text{-}Milwaukee} \\ 10/07/2019 \end{array}$

1 Changes to code

Since 2-step move functionality was added, changes were made to the SequenceMaze and Node classes. The Node class now has a mid_val attribute. This attribute has a default value of None and is only set when the action to get to self is a 2 step move - when the action is a 2-step move it holds the state of the middle value. We check for a two step move using theSequenceMaze class attribute middle_map.keys(). middle_map is a dictionary which actually serves two purposes; to check for actions which are 2-step moves and it maps a two-step action to the action needed to get the middle value.

This attribute is used so that we do not have to generate a Node for the middle tile in two-move actions; we only verify that the move sequence is valid, and if so, we generate the the starting node and ending node of a 2-step move, e.g. if a 2-step move sequence is a->b->c, we only generate nodes with character values of a and c where c has parent node of character value a and c has mid_val attribute of <index, b>

Other miscellaneous changes

- The mid_val attribute is also used within Node.solution() where the middle values are annotated with TRANSITION NODE.
- The Node Class' __repr__ method returns a string with the count_star_as value to make the solution easier to follow when values are present,
 e.g "<Node {}, star counts as: {}>".format(self.state, self.count_star_as)

2 Results and Findings

We implemented A* Search, Greedy First Search, and Hill Climbing Search. Each method takes a heuristic function as an argument where Manhattan distance (Table ??) and Euclidean distance are currently implemented (Table ??). Given the addition of two-step moves, shorter paths were able to be found - largely due to the fact that paths can now double-back, e.g. a two-step move and the path can double back to the transition node (or middle node).

The addition of heuristics greatly improved performance and optimization from the uninformed search algorithms in the last assignment. We are now finding shorter solutions but it is interesting to note that the algorithms are exploring more states - this can be accounted for by the fact that there are now 16 possible actions vs. only 4 in the last assignment. See the **tables** below for a overview of results. For a more detailed look and to see the specific solution sequence, see the seq_maze_res.txt file in this file's directory.

Observations regarding the Hill climbing algorithm implementation: I used a random-restart to generate a random *initial-state*. Each time the algorithm kicked out - via break statement - a check was made to see if the current heuristic value equaled that of our goal, if it did not then a new *initial* state created and the Hill_Climbing algorithm was called recursively. I found that this algorithm did not report any meaningful solutions; it would only cycle until the *goal* state was generated in which case it would report a solution of 0

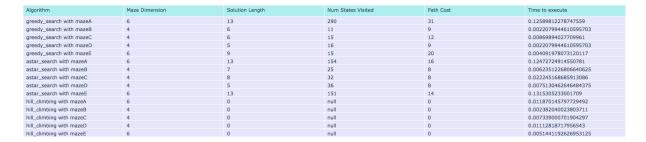


Table 1: Euclidean distance as distance measure

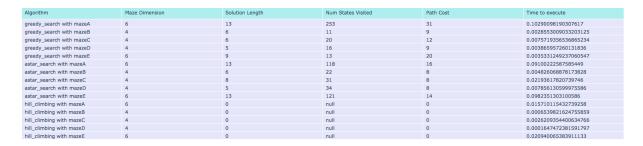


Table 2: Manhattan distance as distance measure

3 Conclusions

Given the informed search algorithms explored in this assignment, I would argue that the A^* Search algorithm using the Euclidean distance heuristic is the best, most optimal way to solve sequence mazes.

4 Files of interest in this directory

- informed.py_search.py informed search algorithms and heuristic functions
- maze.py SequenceMaze and Node classes
- solve.py solver that iterates through algorithms and creates table of results
- mazes/mazes_to_test.py mazes used within solve.py
- seq_maze_res.txt detailed log of results including solution path