CS 710 - Artificial Intelligence

Homework 1 Part 2

Solving Sequence Mazes using Informed Search Algorithms

Mason Edmison
University of Wisconsin-Milwaukee
10/07/2019

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 the`SequenceMaze` 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 soluton of `0`

Algorithm	Maze Dimension	Solution Length	Num States Visited	Path Cost	Time to execute
greedy_search with mazeA	6	13	290	31	0.12589812278747559
greedy_search with mazeB	4	6	11	9	0.0022079944610595703
greedy_search with mazeC	4	6	15	12	0.00869894027709961
greedy_search with mazeD	4	5	16	9	0.0022079944610595703
greedy_search with mazeE	6	9	15	20	0.004091978073120117
astar_search with mazeA	6	13	154	16	0.12472724914550781
astar_search with mazeB	4	7	25	8	0.0062351226806640625
astar_search with mazeC	4	8	32	8	0.022245168685913086
astar_search with mazeD	4	5	36	8	0.0075130462646484375
astar_search with mazeE	6	13	151	14	0.1315305233001709
hill_climbing with mazeA	6	0	null	0	0.011870145797729492
hill_climbing with mazeB	4	0	null	0	0.002382040023803711
hill_climbing with mazeC	4	0	null	0	0.007339000701904297
hill_climbing with mazeD	4	0	null	0	0.01112818717956543
hill_climbing with mazeE	6	0	null	0	0.0051441192626953125

Table 1: Euclidean distance as distance measure

Algorithm	Maze Dimension	Solution Length	Num States Visited	Path Cost	Time to execute
greedy_search with mazeA	6	13	253	31	0.10290098190307617
greedy_search with mazeB	4	6	11	9	0.0028553009033203125
greedy_search with mazeC	4	6	20	12	0.0075719356536865234
greedy_search with mazeD	4	5	16	9	0.003865957260131836
greedy_search with mazeE	6	9	13	20	0.0035331249237060547
astar_search with mazeA	6	13	118	16	0.09100222587585449
astar_search with mazeB	4	6	22	8	0.004826068878173828
astar_search with mazeC	4	8	31	8	0.02193617820739746
astar_search with mazeD	4	5	34	8	0.007856130599975586
astar_search with mazeE	6	13	121	14	0.0982351303100586
hill_climbing with mazeA	6	0	null	0	0.015710115432739258
hill_climbing with mazeB	4	0	null	0	0.0006539821624755859
hill_climbing with mazeC	4	0	null	0	0.0026209354400634766
hill_climbing with mazeD	4	0	null	0	0.0001647472381591797
hill_climbing with mazeE	6	0	null	0	0.020940065383911133

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