PA 2 - Mazeworld

CS 76 AI, Fall 2021

Melissa Valencia

Overall Design

`astar_search.py` contains the a* search algorithm used in both the Mazeworld Problem and the Sensorless Problem. `priority_queue.py` contains the functions necessary for the implementation of a priority queue in the a* search algorithm. `SearchSolution.py` outlines the necessary data for the valid solutions returned from any of the search algorithms. `Mazeworld.py` and `Sensorless.py` represent key information about the problems, respectively, including the start state, a get successors method stating the possible moves and checking the validity of states, a function to calculate total costs used for setting the priority, a hueristic function, and a goal test. `test_mazeworld.py` and `test_sensorless.py` is where the a* search algorithm is tested on the Mazeworld Problem and the Sensorless Problem using the provided mazes.

A* Search

Implementation

In order to solve these problems, this implementation of a* search followed a graph-searching approach. The data structures used were a queue and a dictionary implemented through the priority queue. This priority queue, also known as the frontier, kept track of the visited states, ensuring that the cheapest costs were updated in the dictionary of state costs and keeping the priority based on these costs. While the frontier was not empty, keep track of the current node using pop_task() and check if this node's state is the goal state. If we have found our goal state, we backchain through this node, update our transition cost, and return our solution. Until we find our goal state, we iterate through our current node's state's successors and check if this successor's state is the goal state. If it is, we backchain, update the transition cost, and return our solution. If not, we add this successor node to our priority queue, the frontier, which will determine whether this node has already been visited or not.

Test Cases

To test this a* search algorithm, I first tested the given mazes in `test_mazeworld.py`, then I tested my own mazes on both the Mazeworld Problem and the Sensorless problem, `test_sensorless.py`.

Mazeworld Problem

Discussion Questions

1. If there are k robots, the state of the system can be represented as a list of integers where the first value is the robot that is moving next and the following values are the coordinates of the locations of the robot.
2. An upper bound on the number of states in the system can be represented as ((n!)/k!(n-k)!)*k where k is the number of robots and n represents the size of the maze, assuming it is of size n by n.
3. A rough estimate of how many of these states represent collisions if the number of wall squares is w, and n is much larger than k can be ((n!)/(k!(n-k)!)-(w!)/(k!(w-k)!))*k.
4. If there are not many walls, n is large (say 100x100), and several robots (say 10), I do not expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs because BFS would continue getting exponentially worse as it would go through each possible location (N,S,E,W and not moving) which is 5^10 for all the robots moving.
5. The heuristic I implemented in this problem is the manhattan heuristic. In this heuristic, I add up the absolute values of subtracting the x and y coordinates of the goal locations and the states for each robot, respectively. This heuristic is monotonic, also known as consistent, as it satsifies the condition of `h(n) <= c(n,a,n') + h(n')` which states that the heuristic is monotonic if it is no greater than the step cost of getting to the successor of n plus the estimated cost of reaching the goal from the successor of n. This ensures that the manhattan heuristic calculated is never greater than the actual cost to the goal location.
6. The 8-puzzle in the book is a special case of this problem as the puzzle has more movement constraints given that only one square is empty thus only one tile can move at a time, so the movement of the rest of the tiles is restricted. I think the manhattan heuristic I chose is a good one for the 8-puzzle because it is still admissible, providing a lower bound.
7. The two disjoint sets of the state space of the 8-puzzle is composed of a set of reachable states and a set of non-reachable states. I would modify my program to determine what states are reachable vs which ones are not.

Implementing and Testing

Aside from testing a* search on the mazes provided for the Mazeworld Problem, I created my own mazes to continue ensuring completeness and optimality with the implementation of this a* search algorithm. The first maze I made, `maze4.maz` was more of a continuity of generally ensuring my a* search algorithm was performing as expected. With `maze5.maz`, I was following the order in which the robots moved. The following maze,`maze6.maz`, was testing the time complexity, analyzing how long it would take to switch the locations between two robots. The last two mazes, `maze7.maz` and `maze8.maz`, were simply testing much larger mazes than the other mazes.

Output

One of the outputs from my test mazes is shown below. This is `maze5.maz`. Note that the animation of the paths may take some time. The animation can be commented out in `test_mazeworld.py`.

```
____
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 210
solution length: 25
cost: 15
```

```
path: [(0, 0, 2, 1, 2, 2, 2), (1, 0, 2, 1, 2, 2, 2), (2, 0, 2, 1, 2, 2,
2), (0, 0, 2, 1, 2, 3, 2), (1, 0, 2, 1, 2, 3, 2), (2, 0, 2, 2, 2, 3, 2),
(0, 0, 2, 2, 2, 4, 2), (1, 0, 2, 2, 2, 4, 2), (2, 0, 2, 3, 2, 4, 2), (0,
0, 2, 3, 2, 5, 2), (1, 0, 2, 3, 2, 5, 2), (2, 0, 2, 4, 2, 5, 2), (0, 0, 2,
4, 2, 6, 2), (1, 0, 2, 4, 2, 6, 2), (2, 0, 2, 5, 2, 6, 2), (0, 0, 2, 5, 2,
6, 1), (1, 0, 2, 5, 2, 6, 1), (2, 0, 2, 6, 2, 6, 1), (0, 0, 2, 6, 2, 6,
1), (1, 1, 2, 6, 2, 6, 1), (2, 1, 2, 7, 2, 6, 1), (0, 1, 2, 7, 2, 6, 2),
(1, 2, 2, 7, 2, 6, 2), (2, 2, 2, 7, 2, 6, 2), (0, 2, 2, 7, 2, 5, 2)]

Mazeworld problem:
#########
ABC......
######.##
#########

Mazeworld problem:
#########
ABC......
######.##
#########

Mazeworld problem:
#########
ABC......
######.##
#########

Mazeworld problem:
#########
AB.C.....
######.##
#########

Mazeworld problem:
#########
AB.C.....
######.##
#########

Mazeworld problem:
#########
A.BC.....
######.##
#########

Mazeworld problem:
#########
A.B.C....
######.##
#########

Mazeworld problem:
#########
A.B.C....
```

```
######.##
#########

Mazeworld problem:
#########
A..BC....
######.##
#########

Mazeworld problem:
#########
A..B.C...
######.##
#########

Mazeworld problem:
#########
A..B.C...
######.##
#########

Mazeworld problem:
#########
A...BC...
######.##
#########

Mazeworld problem:
#########
A...B.C..
######.##
#########

Mazeworld problem:
#########
A...B.C..
######.##
#########

Mazeworld problem:
#########
A....BC..
######.##
#########

Mazeworld problem:
#########
A....B...
######C##
#########

Mazeworld problem:
#########
A....B...
```

```
######C##
#########

Mazeworld problem:
#########
A.....B..
######C##
#########

Mazeworld problem:
#########
A.....B..
######C##
#########

Mazeworld problem:
#########
.A....B..
######C##
#########

Mazeworld problem:
#########
.A.....B.
######C##
#########

Mazeworld problem:
#########
.A....CB.
######.##
#########

Mazeworld problem:
#########
..A...CB.
######.##
#########

Mazeworld problem:
#########
..A...CB.
######.##
#########

Mazeworld problem:
#########
..A..C.B.
######.##
#########
```

Sensorless Problem

Implementation and Testing

I tested my implementation of this a* search algorithm on the mazes provided and mazes I created as stated above. Here, I tested the Sensorless Problem, checking whether the goal state had been reach, that is that the robot had been located in a coordinate.

Discussion Questions

1. The heuristic I used for the a* search implementation in the Sensorless Problem is what I call a grid heuristic. This heuristic is based on the size of the maze, considering its width and height. Within this heuristic, we calculate based on a subset of possible locations. This heuristic is optimistic, because it is always calculating a value that is less than or equal to the actual cost which is ensured by the `-1`. Aside from this heuristic function, I could have used a heuristic that implemented the `log` of the length of the state. However, I believe that the grid heuristic I implemented is much more optimal and efficient as it calculates a heuristic that is more dominant, as it is closer to the actual cost of the path.

Output

One of the outputs from my test mazes is shown below. This is `maze4.maz`. Note that the animation of the path may take some time. The animation can be commented out in `test_sensorless.py`.

```
____
Blind robot problem:
attempted with search method Astar with heuristic grid_heuristic
number of nodes visited: 431
solution length: 12
cost: 11
path: [(1, 1, 1, 2, 1, 3, 1, 4, 2, 0, 2, 2, 2, 3, 2, 5, 3, 0, 3, 1, 3, 2,
3, 3, 3, 4, 3, 5, 4, 1, 4, 2, 4, 4), (4, 4, 1, 1, 1, 4, 3, 0, 2, 3, 4, 2,
3, 3, 2, 2, 3, 2, 4, 1, 3, 5), (4, 4, 1, 2, 3, 4, 3, 1, 1, 4, 4, 2, 2, 3,
3, 3, 3, 5), (1, 2, 3, 4, 3, 1, 1, 4, 2, 3, 3, 2, 2, 5, 1, 3), (1, 4, 2,
3, 3, 3, 3, 2, 2, 5, 1, 3, 3, 5), (1, 4, 2, 3, 2, 2, 2, 5, 1, 3), (1, 2,
2, 5, 1, 3, 2, 2), (2, 3, 3, 2, 3, 5, 2, 2), (2, 3, 3, 3, 3, 5), (3, 3, 3,
5), (3, 4, 3, 5), (3, 5)]

Blind robot problem:
##HN###
#D#MQ##
#CGL###
#BFKP##
#A#JO##
##EI###

Blind robot problem:
##.K###
#C#.A##
#.EG###
#.HIF##
#B#.J##
```

```
##.D###

Blind robot problem:
##.I###
#E#CA##
#.GH###
#B..F##
#.#D.##
##..###

Blind robot problem:
##G.###
#D#B.##
#HE.###
#A.F.##
#.#C.##
##..###

Blind robot problem:
##EG###
#A#..##
#FBC###
#..D.##
#.#..##
##..###

Blind robot problem:
##D.###
#A#..##
#EB.###
#.C..##
#.#..##
##..###

Blind robot problem:
##B.###
#.#..##
#C..###
#AD..##
#.#..##
##..###

Blind robot problem:
##.C###
#.#..##
#.A.###
#.DB.##
#.#..##
##..###

Blind robot problem:
##.C###
#.#..##
#.AB###
```

```
#....##
#.#..##
##..###

Blind robot problem:
##.B###
#.#..##
#..A###
#....##
#.#..##
##..###

Blind robot problem:
##.B###
#.#A.##
#...###
#....##
#.#..##
##..###

Blind robot problem:
##.A###
#.#..##
#...###
#....##
#.#..##
##..###
```