CS76 21F - PA-2 Mazeworld [Benjamin Cape]

Description

How do your implemented algorithms work? What design decisions did you make? How you laid out the problems?

A*

My algorithm works as follows:

```
Initialize an empty priority queue Q, add the start node to Q.

Initialize an empty associative array A. For x \in A, A[x] = min cost from root for x. Add t

While Q is not empty {
    c <- pop the lowest value off priority queue

    if c is marked: skip

    if c is goal: end and backchain solution

    for (cost, neighbor) of c {
        tot_cost <- A[c] + cost
        if neighbor has not been seen: add to queue
        if neighbor has been seen with higher cost, add to queue
    }
}
```

Regarding implementing this psuedocode, I use a heap to keep a priority queue. Each node needs to be compared, and it's comparison value is the total cost + the heuristic value.

MazeworldProblem

All of the algorithms that are apart of the MazeworldProblem are relatively simple. The only one that contains slight complexity is the legal moves. The legal moves returns, for the provided state, a set of moves that are legal from the position. It loops over all possible moves, and returns a generator of the ones that are legal. The check for legality makes sure that a robot is:

1. Moving into an open space (floor and not robot occupies) OR
2. Not moving at all

We then use this generator in the `get_successors()` to perform the actual transition.

I decided to convert the list of goals into a list of tuples, and turned the robot locations also into a list of tuples. We then match up the sizes of these two

tuples so that we require that A be in location 1 and B in 2 .... Using tuples to think of coordinates is much easier and requires less computation than making a what is a 2-d space into a 1-d array. I liked this, and made a few changes in the Maze.py file as well to accommodate this. The robot who's turn it is is specified by a variable in the state of the problem.

At each state, only one robot can move. The action enum described those moves, and how they change the robot's location coordinates.

I've added a new printing function for a maze to show the goal states as well.

My heuristic for the maze problem is the sum of the manhattan distances from each location to it's designated solution. The proof for why this works in below.

SensorlessProblem

This implementation is even simpler than the previous.

My heuristic for this problem is the maximum manhattan distance between any two options of any given state. This takes a LONG time to compute for larger graphs. $O(n^2)$ time. We could make it faster by doing clever computation when we generate the state though. I did not implement this faster one.

Evaluation

Do your implemented algorithms actually work? How well? If it doesn't work, can you tell why not? What partial successes did you have that deserve partial credit?

My algorithms work properly. When compared to BFS, which we know is correct and optimal I get the same answer. In fact, I think my algorithms work relatively well. I know it works because it visits fewer nodes than does BFS, or an A* search with the null heuristic (which I know are the same thing).

One algorithm I wrote that doesn't work so well is the randomized one, since sometimes there is no solution if a blind maze has an enclosed space, and if that is the case we have to explore EVERY node, which takes too long for my computer to handle.

I was experiencing some weird functionality with my A* implementation. One way I was getting a cost of 10 path with a length of 19 and only visiting ~160 nodes. Another way (submitted) I am getting a length of 16, and visiting ~250 nodes. I think the later is correct, because it finds both the shortest path and the cheapest path. But the former also gets a shortest path and does so MUCH faster, by visited WAY fewer nodes. It does so by considering the cost of each node 1 in terms of popping from the queue, but considers the cost in the actual path construction.

I have submitted both as `astar_search.py` and `astar_search2.py`.

I have concluded based on the following example, that the former is better, and sometimes I guess even explores fewer nodes! This is my source of confusion though...

```
First:  ----
Mazeworld problem:
.....2........#..#.
#...........#...#..
.##....#...#...#...
#....#........##...
............##..#.#
..........#........
.......#...........
...................
....##..#.....#...#
###.......A....#...
.B..#..#...#......#
.......#.....#.#...
.....#.#...........
.#.............0...
...#...........#...
.....#.............
.........#.#..#.##.
....#..............
...#.......#...#.#1
C...##.....#.......
..#.#..............

attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 17029
solution length: 88
cost: 58

 Other:  ----
Mazeworld problem:
.....2........#..#.
#...........#...#..
.##....#...#...#...
#....#........##...
............##..#.#
..........#........
.......#...........
...................
....##..#.....#...#
###.......A....#...
.B..#..#...#......#
```

```
.......#.....#.#...
.....#.#...........
.#............0...
...#..........#...
.....#............
.........#.#..#.##.
....#.............
...#.......#...#.#1
C...##.....#.......
..#.#.............
```

```
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 24343
solution length: 112
cost: 58
```

Discussion Questions:

1. If there are k robots, how would you represent the state of the system? Hint – how many numbers are needed to exactly reconstruct the locations of all the robots, if we somehow forgot where all of the robots were? Further hint. Do you need to know anything else to determine exactly what actions are available from this state?

Well, first we will need to know which robot's turn it is to move. This will take one value in the state. We can store each robots location in one value, but that is simply storing 2 log(n)-bit values in a log(nˆ2)-bit value. I've then represented the state of the system as such:

```
{
    robot: int,
    robot_locs: List[Tuple[int, int]]
}
```

2. Give an upper bound on the number of states in the system, in terms of n and k.

Assume we have an nxn board, with k robots, that means we have n^2 buckets to place k elements into. This is a simple

$$\binom{n^2}{k}$$

Essentially, counting the number of subsets of size k in a set of 2^k elements. That determines the k squares, then, within the k squares we can arrange the k robots in $k!$ different ways, since each one is distinct.

Therefore our total upper bound it:

$$O(k!\binom{n^2}{k})$$

3. Give a rough estimate on how many of these states represent collisions if the number of wall squares is w, and n is much larger than k.

BUT, we didn't consider the w walls. So lets count how many different scenarios there are where at least one robot is on a wall. We know that we will sum over all the number of robots that can be on a wall, so we will have a summation from $[1, k]$. Then, for each k we have a similar counting as above.

$$\sum_{i \in [1,k]} i!\binom{w}{i}\binom{n^2 - w}{k - i}$$

We tack on the other binomial coefficient because we also need to count the possible ways that we can permute the remaining robots on the rest of the board, hence $n^2 - w$ because that is the number of floor locations.

4. If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?

No, a breadth first search would not be feasible, because it is likely that our solution state is very very far from our start state. We assume that the branching factor for each state is $O(5 * k)$ (in our implementation it would only be k but consider the cycle of all k robots going one state for simplicity of calculations), and the depth could be as much as 200, that means our memory/time complexity is $O((5k)^{200})$, that's a huge number. This is incredibly unfeasible. Say we use the metrics from the question, we have time roughly $6.2x10^{339}$, that's more calculations clock-cycles then there are possible chess boards.

5. Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic. See the textbook for a formal definition of monotonic.

A good heuristic function, $h(x)$ for this search space would the be the sum of all the maximum manhattan distances from each robot to a goal state.

For this to be monotonic, we must show:

$$\forall x, y \text{ s.t. } x \rightarrow_a y; h(x) \leq c(x, a, y) + h(y)$$

Proof

Fix a robot, r, and consider two states x, y where we can reach y by performing the action a on x. The first scenario is that $a = NA$, meaning the cost is 0, in this case $h(x) = h(y)$, satisfying the

condition. (We only have to worry about one robot moving, because only one robot can move at each state)

In the other scenarios, $c(x, a, y) = 1$. It is very easy to show though that the move to y can only increase the manhattan distance by at most 2 (by extending the shorted path manhattan distance for x). Assume w.l.g that the shortest path leaves x to the (E). then in a moves to the (N, S, W) we've added 2 to our cost. If $a = E$ though, then $h(y) = h(x) - 1$, which obviously is OK.

$$h(x) \leq c(x, a, y) + h(y) = 1 + h(y) \leq 1 + h(x) + 2 = h(x) + 3$$

Making the inequality hold.

6. Describe why the 8-puzzle in the book is a special case of this problem. Is the heuristic function you chose a good one for the 8-puzzle?

The 8-puzzle problem is simple the maze problem when $k = n - 1$ The heuristic I chose is exactly the heuristic is commonly used for the 8-puzzle.

7. The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this. (You do not have to implement this.)

The state for our game can be described as two disjoint sets, being the set of robot locations, and the set of wall locations. That is enough information for us to accomplish our goal. We do not need to know where anything else is, or keep track of an entire map that is $n * m$ in size. We only need to keep track of the squares that are filled, and keep track of the size of the space (width/height). This allows us to efficiently determine if a certain coordinate is occupied by a wall or a robot, otherwise it is clear and we can move there.

This would save us some space. But since we are working on such small maps it probably won't make a big difference.

Blind Robot Problem

1. Describe what heuristic you used for the A* search. Is the heuristic optimistic? Are there other heuristics you might use? (An excellent might compare a few different heuristics for effectiveness and make an argument about optimality.)

I am using a heuristic that calculates the sum of the manhattan distances from all possible states that we current suspect. Consider that my solution only checks 10 nodes for a 5x6 grid, whose solution is a path of 10 nodes I imagine it is a good heuristic efficient. Also, this is the same heuristic proves above to be monotonic.

Using a similar heuristic, but instead of the sum using the maximum still is optimal, but expands more nodes 313 to be exact. This makes sense, given the hint in the assignment. We could in fact increase our maximum distance, but decrease the sum of the distances because we made more closer, or got rid of one along the way.

Testing

For my testing I have added the ability to create random maps. While this is nice to test working/not working on a variety of different solutions. For graphs of size <=40x40 my algorithm performs incredibly well. I have also test on graphs as large as 100x100 and the algorithm is able to complete find a solution, or determine no solution possible within a matter of seconds. The issue with my random graphs is that there will not always be a solution, because a robot could be trapped.

I have also provided test code for random graphs on sensorless graphs. This also complete in a timely manor and returns realistic data. Issue with these is that if there is no solution, we ultimately have to run an entire BFS order search. which is WAY to slow, so we don't end up getting a solution in decent time.

Here are a few graphs that were generate by the random generator and their return values:

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

```
#....####...#........#.........#..........##......
.........#.#......#.....###..#..#.#....#..........
.#............#.#...##..#......#.........####.....
.............#.#..............#........1.#...#.#0.
.#.#.#.............##......##.#.##..#...#....##...
.##.#.#........#..........##.........#....#.#...##
..#..##...#..#.......#..B.....#.#....#...#........
......#..##.........#.......#....#...#........#..#
```

```
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 3262 // Numbers are out of date since I've added the marking
solution length: 82
cost: 59
```

This is an example of a VERY easy one to solve. Since the heuristic of each consecutive square is monotonically decreasing so each best node is constantly popped off the stack.

```
Mazeworld problem:
```

```
.#........#...#..................#......#..#....#.#.#..##...
.#.#................#...#..#..##...#.....#...#....#........
.........................#......#.........................#.
....##...#...#......................#.............#......#
.##............................#........#....#..#....#.#....
............#......##.......#..#...#......#.#...#.#......#
...#...#.........#.......A.....#.....#....#....#..........
.........#.#.....................##.................#...#...
.........................#................#.#...........#....
..##.#............#..#..................#....#.#.#.......#
...#........#..................#.....#.#.##...#.#...#.....
............#......#...........#............##.....#......
..#.........................................#...#........
.#.#..........#.#.......#..............#..#.......#.......
...........#.....#...#.#..................#................
...........#.#.............##.......#..................#.#.
.......#......#.#....#..#.#......#.#.........#....#.#.#...
......##.....#..................#.....#..............#...##
.....................##.......................##..........##.
....#............#...#........#.#..#.....#.#..............
................#....##.....#.........#.....#..........#..#
........#.#.......#..##........##..#....#.....#.#.........
.....##............##.........#............#.#.#.....#.#..
....#...#..................#.......................#..#......
....#.#...........#.......#....#..#..##..................#.
.........................#......##...##..#..#..#.....#.#...#
.##.#............#......#....#.............................
........#..............#..............#..............#.
```

```
..##..#..#......#.......0....#..#.........#.......#....#..
.........................#.#.............#...........##..#.
#...#....#.##....##....#..#......#.........#..#...#....#..
..#.....................................#.............##.#.
...#..#.........#..#..#................#...................
..#.....#......#.#........#.#..#..#................#..#....
............#........##......#..................#..#.....#.
..##..................#.#.............................#.....
...........#........#......#......#..#...........#........
..........#...#.............................#...............
..#................#.....................#..#.....#........
...#...##...#.....B.......##...#...#..............#........
................#..#..#..#..#.##.....1#...............##...
..##...#..#.#.#.....#....#.....#...#.#...............#....
```

```
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 8906
solution length: 51
cost: 46
path: [Turn: A, locations: [(25, 35), (18, 2)],Turn: B, locations: [(25, 34), (18, 2)],Turn:
```

Here is an interesting case, and it helps us understand why we have a larger solution length then we do cost. one of the robots gets to it's goal before the other one, therefore it has to "forfeit" it's turn until the other one gets home.

For the Sensorless Problem: (Disregard the A, it doesn't mean anything here)

It is also interesting to note that when there is only one robot the cost of the solution will always be exactly one less than the length of the path. This is because nobody has to wait as described in another situation of this document.

```
Mazeworld problem:
.....................................#..#........#...#
.#....#.....................................#....#...
....#......#...#.....................................
.............................#............#............
.............................#........................
.................................................................
.....................................#......#.....#...
..................#................#............#...#..
................##.........#..........#..............
..............................#............#........
...##..........................#.....................
A..........................#.........................
.#..#.#................#0....#....#....##.....#.....
.....................................................##.....
...........#.....................................
.....................................#............
```

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

```
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 46
solution length: 26
cost: 25
path: [Turn: 0, locations: [(0, 10)],Turn: 0, locations: [(1, 10)],Turn: 0, locations: [(2,
```

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

```
attempted with search method Astar with heuristic distance
number of nodes visited: 883
solution length: 14
cost: 13
path: [Move: None, Options: {(4, 0), (3, 4), (4, 3), (3, 7), (4, 6), (0, 2), (0, 5), (2, 2),
```

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

```
attempted with search method Astar with heuristic distance
number of nodes visited: 24878
solution length: 21
cost: 20
```

This is a very difficult one to solve, because even if things are close, there are lot of steps to bring them together, i.e. the heuristic is not as great in this situation.

In this case, the options heuristic solves the problem much faster, but it does not in fact find an optimal solution:

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

attempted with search method Astar with heuristic options_heu
number of nodes visited: 3251
solution length: 24
cost: 23

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

There is no way for the robot to locate itself, because it cannot fold all locations into one (notice the hold, or other room made up to the right of the character A).

Extra Credit

I have implemented the synchronous movement as seen in `BONUS_MazeworldProblemSync`. Not much to tell here. We use a simple recursive call to find all the possible actions that we can get from a certain state.

Testing and Running

Run `make tests` to run all the tests on the random graphs and on the standard `maze3` provided. Sometimes the random graph is not solvable, if this is the case for the blind robot problem, then you'll need to re-run it because checking every state will take WAY to long.