PA 2 - Mazeworld

Julien Blanchet | CS 276 (Artificial Intelligence) | Prof. Li.

October 2, 2020

Introduction

This package provides a solution to our second programming assignment, and includes an implementation of A* search, compatible search-problem models for multi-robot and blind-pacman robot maze navigation scenarios, and test code for the these search problems. Each components along with the relevant discussion questions is discussed in its own section.

A* Search

My implementation of A* search is equivilant to canonical descriptions of the algorithm in the textbook and online (in Wikipedia). In developing this implementation, I referenced and adapted an earlier implementation of A* that I had written for COSC 181 (in the Winter 2020 class). Like these, mine is a best-first search algorithm which uses $path - cost + heuristic$ as the criteria for determining the next-best node to expand. I made a few design decisions that differ from the direction the assignment seemed to be suggesting:

1. I elected to use native python dictionaries to store search information rather than node objects. This felt like a more straightforward approach than introducing extra object-orientation, and incurs no additional runtime or space complexity, as adding to / removing from the dictionary are constant-time operations and the information stored in the dictionary would also be stored in the object-oriented approach.
    - As a consequence of this decision, state objects returned by search problems must be hashable for my solution to work. As discussed in later sections, I designed the multi-robot coordination and sensorless robot problems to have hashable states.
    - Instead of the object-oriented approach of implementing a comapre method on the node object, my code inserts tuples into the frontier priority queue in the form of `(priority, node_age, state)`. Comparisons with python tuples work by comparing element by position and returning the result of the first comparison that isn't equal. The result in my case is that elements will be compared by priority and in case of a tie the older node will be expanded first.
2. Rather than using the `heapq` library directly, I elected to use the `PriorityQueue` class. This provides the same performance characteristics with a cleaner interface (in fact, `PriorityQueue` using `heapq` under the hood).

Multi-Robot Coordination

The multi-robot coordination problem tasks us with finding a way to get k robots in an $n \times n$ maze to target locations while avoiding walls and not running into each other.

Discussion Questions 1. I defined a model for the state of the system in the `MazeworldProblemState` class. Included in this state are the `x` and `y` locations of each of the robots and an indicator for which robots turn it was to move. I also put logic in this class to return a successor state based on a specific move, and implemented a hash function to allow the state to be used with my A* search implementation. The s 2. Given k robots and a $n \times n$ matrix, there are n^2 possible locations for each of the k robots, and for each of these states it could be any of the k robots' turn, giving us an upper bound of k^2n^2 possible states. 3. If $n \gg k$, than the number of robot-robot collisions will be small, as the the number of possible locations grows quadratically with n. Therefore, we can ignore robot-robot collisions and only focus on robot-wall collisions. The percentage of occupied (wall) cells on the maze can be calculated as $\frac{w}{n^2}$. From there, we can estimate the number of collisions as $k \cdot \frac{w}{n^2}$. 4. Given a problem with a large n, significant k, and small w, the state space will be enormous, as it grows quadratically with both of these inputs and a small w indicates few collisions / invalid states. Since BFS expands nodes breath-wise from the start, I expect BFS to be computationally infeasible in cases where the goal locations are far from the start locations. 5. I used the manhattan distance as a heuristic, specifically the sum of manhattan distances between each robot's current and goal positions. This heuristic is *consistent* (monotonic) because for every node n, there are 5 possible successors n' - the robot whose turn it is can move in each of the four cardinal directions or stay still. If the robot stays still, the path cost is zero and the heuristic is unchanged, so the consistency requirement of $h(n) \leq h(n') + c(n, a, n')$ is met. If the robot moves, than the heuristic is at most changed by one, and the path cost is one, so the requirement is also met. 6. My implementation of the multi robot problem is coded in `MazeworldProblem.py`. Validation of the sucessor function and state implementation is tested in `test_mazeworldproblem.py`, and the program is tested with several mazes in `test_mazeworld.py`. I used an ASCII maze generator website to generate some of my larger mazes. Some of the interesting test cases were:

Case Title	Description
Maze 1	Basic validation. 5×5 maze, singlerobot must go in a spiral. No turns available.
Maze 2	More complex 8×5 matrix, simple robotmust navigate to the other corner.

Case Title	Description
Maze 3	5×5 maze, 2-robot test case - one robotmust wait for the other.
Maze 4	7×5, 3-robot, starting in oppositecorners they must cross in the middle and reach far cornerwithout colliding. *See below for sample path.*
Mazes 40,41	21×31, single robot tests
Maze 50	16×11, two robot test
Impossible Corridor	5×1, two robot test whereno path should be found because the robots have to crosseach other and there's no space to do so.
Maze Disconnected	6×6, single robot, goal is ina disconnected part of the maze to robot. No solutionsohuld be found.

Test Case 4

Capitals represent robot locations, lowercase represents goals.

```
#######     #######  #######  #######  #######
.A.#.Bc     ...#..c  ...#..c  ...#..c  ...#..c
##...## ... ##A.B##  ##A.B##  ##.AB##  ##.A.##
##...##     ##...##  ##C..##  ##C..##  ##C.B##
bC.#..a     b.C#..a  b..#..a  b..#..a  b..#..a
  (0)         (6)      (7)      (8)      (9)

#######    #######  #######  #######   #######
...#..c    ...#..c  ...#..c  ...#..c   ...#..c
##CA.##    ##C.A##  ##C.A##  ##.CA##   ##.C.##
##..B##    ##..B##  ##.B.##  ##.B.##   ##.BA##
b..#..a    b..#..a  b..#..a  b..#..a   b..#..a
  (10)       (11)     (12)     (13)      (14)

#######  #######  #######  #######     #######
...#..c  ...#..c  ...#..c  ...#..c     ...#.Cc
##.C.##  ##..C##  ##..C##  ##..C## ... ##...##
##B.A##  ##B.A##  ##B..##  ##...##     ##...##
b..#..a  b..#..a  b..#A.a  b.B#A.a     b..#..a
  (15)     (16)     (17)     (18)        (24)
```

7. The 8 puzzle problem can be viewed as a special case of the multi-robot problem on a 3×3 maze with no walls and 8 robots arranged appropriately. My heuristic function - summed manhattan distance - should be fine for

this problem, but better heuristics are surely possible which perhaps better take into account how "ordered" the robots are.

8. To find the disjoint sets in the state space of this problem, I'd perform a BFS on any given state to see what is reachable - thereby detecting the first set. Then, I can try to find a state that is not reachable by the first state and perform a BFS on *that* in order to detect the second disjoint set.

Blind Robot with Pacman Physics

State Representation The second search problem we were tasked with implementing consisted of: given a maze and a blind robot that knows the structure of the maze but not its location within the maze and cannot detect when it has successfully moved of not, find a sequence of actions the robot can take such that it will end up in the goal location no matter where it started in the maze.

I represented the state of this problem as a set of tuples, each representing a different potential location of the robot. In the initial state, this is initialized to contain tuples with all the non-wall squares in the maze. The successor function is fairly brute force - it simple computes where the robot would be next for each of the potential locations given 4 possible moves (one in each of the cardinal directions). In order to make the state hashable and comparable, I sort the possible states by the x and y locations.

Heuristics I decided to use a composite heuristic based off two sub-metrics - conversion distance and goal distance. This is inspired by the obsevation that the robot's hypothesized locations must both converge and be equal to the goal location in order to achieve the goal state. The specific calculations are justified as follows: * *Goal Distance*: if the robot has a hypothesized location with manhattan distance d from the goal location, it will take at least d moves before that hypothesized location can reach the goal location. * *Convergence Distance*: if the robot has two hypothesized locations with c manhattan distance between them, it will take at least c moves before these two beliefs can converge, which must happen before the goal state can be achieved. * *Composite Heuristic*: The maximum of *Goal Distance* and *Convergence Distance* - given that these are both admissible, consistent heuristics, the higher the estimation, the closer to the actual path cost. * ***Bonus:*** *Nonadmissible Weighted Heuristic*: I ran into performance limitations when testing with my composite heuristic on larger mazes, so I tried another option. This heuristic calculates a weighted sum of the goal distance d_g and convergence distance $d_c$metrics as follows:

$$h(n) = C_h \cdot (C_c \cdot d_c + d_g)$$

C_h represents the heuristic weight and C_c represents the convergence factor. The result can be larger than the remaining path cost and as such is non optimistic and does not guarantee optimality.

Testing Here's the outputted plan for a basic 4×3 maze: (note: `*` represent beliefs, `0` represents the goal position, and `0` represents the goal postition with a belief at that position)

```
##*#     ##*#     ##*#     ##.#     ##.#     ##.#
#***  ↑  #*.*  ←  #.**  ↓  #.*.  →  #..*  ↓  #...
#*#0     #.#0     #.#0     #.#0     #.#0     #.#0
  1        2        3        4        5        6
```

I also tested the search on a larger maze and quickly ran into performance limitations with the composite heuristic. One of the larger ones that completed in a reasonable time is shown here:

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

I also ran the weighted heuristic with a $C_h = 1.5$ and $C_c = 1.5$ for comparison. The results (shown below) indicate a huge performance benefit for the weighted heuristic at a slight optimality cost.

Heuristic	Visited Nodes	Cost
Composite	5672	17
Weighted	24	18

To find the ideal weights for the weighted heuristic, I ran several trials with various weights on a larger, 10×7 maze. The bolded row is the one I ultimately selected.

C_h	C_h	Visited Nodes	Cost
2	2	284	35
1.5	2	297	32
1.25	2	473	32
1	2	981	28
0.5	2	41919	23
1.5	2	297	32
1.5	**1.5**	**719**	**26**
1.5	1	1440	26
1.5	0.5	6412	25

Additional Comments

I made some changes to the provided code to assist with testing and to better align with how I wanted to represent the state space to this problem. Specifically, instead of storing robot locations as a one dimensional list `[x1, y1, x2, y2, ...]`, I chose to store it as a list of tuples `[(x1, y1), (x2, y2), ...]`. This made the indexing identical between robot locations and goal locations and allowed for easier comparison between positions. Additionally, I extended the maze class to support loading mazes directly from strings, and adding `str` methods for displaying goal locations and potential robot positions.