Joshua Pfefferkorn

Programming Assignment #2

COSC 76, Fall 2021

Description

Problem Modeling and Design Choices

For the MazeworldProblem, I used a tuple to represent the state of the system. The first index in the tuple held the number of the robot whose turn it was. Subsequent pairs of numbers held x- and y-coordinates of robot locations in the maze, beginning with robot 0. For the Sensorless Problem, the state was a set of tuples. Each tuple within the state held the coordinates of one location where the robot could theoretically be. Since there was only one robot, I did not need to include the number of whose turn it was.

A* Search

A* is a search algorithm that calculates the value of each successor by adding the cost to that successor so far with a heuristic, an estimate of how close that node is to the goal. The node with the lowest value is then visited, which guides the search towards the goal (unliked a uniform-cost search, for example, which expands out in all directions, not necessarily towards the goal). My implementation used a priority queue to sort the nodes by their value, ensuring that the next node visited was the one with the lowest transition cost plus heuristic.

Evaluation

Results

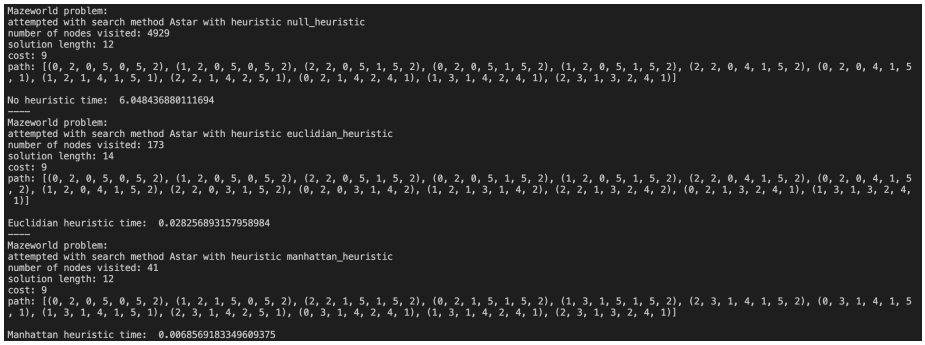

Figure 1: Results

Judging from my test cases, my algorithm worked as intended. Viewing the number of nodes visited and path length using each heuristic confirmed that

my heuristics were working, and animate_path provided visual evidence that in MazeworldProblem robots navigated successfully to their goals and in SensorlessProblem a good localization plan was generated. I tested MazeworldProblem with maps as big as 40x40, and I tested SensorlessProblem with maps up to 10x10.

In Figure 1, path length, solution cost, and runtime speed is shown with each of three heuristics (0, Euclidean, and Manhattan) for three robots in a 6 x 10 maze I created. As expected, the null heuristic (0), a simple uniform cost search, performed far worse than the heuristics, visiting nearly 5000 nodes in over 6 seconds. Between the two heuristics, total Manhattan distance performed better (41 nodes in 0.007 seconds) than total Euclidean distance (173 nodes in 0.03 seconds). This can be explained by the dominance of the Manhattan heuristic, elaborated on later in the discussion section of this report.

```
##L##  ##.##  ##.##  ##.##  ##.##  ##.##  ##.##  ##.##  ##.##
#GAD#  #.A.#  #.B.#  #F..#  #D..#  #...#  #...#  #...#  #...#
#M#H#  #I#F#  #.#.#  #.#.#  #.#.#  #D#.#  #.#.#  #.#.#  #.#.#
#BIK#  #B.H#  #C.G#  #AC.#  #C..#  #C..#  #C..#  #...#  #...#
#FCE#  #ECD#  #FDE#  #EB.#  #B..#  #B..#  #B..#  #B..#  #...#
#J###  #G###  #A###  #D###  #A###  #A###  #A###  #A###  #A###
```

The SensorlessProblem, too, seemed to generate accurate and consistent results. Above, the algorithm found a 9-step solution for a 5 x 6 maze with walls, sequentially eliminating belief states with each move until localization was achieved. I found that once the maze got bigger, though, the runtime of A* with SensorlessProblem increased dramatically. This could be because of the computationally-expensive get_successors method, which has to run about over a dozen checks each time it's called. The largest maze I tried was 10x10.maz, for which the alogrithm found a 30-step solution after visiting almost 10,000 nodes.

Test Cases for MazeworldProblem

I made a couple of interesting mazes as test cases for MazeworldProblem.

- The first, `wall.maz`, had a robot try to get from one side of a barrier of walls to the other. The algorithm, as expected, quickly ran out of successors to search and returned no solution.
- A second maze, `hallway.maz`, had two robots on opposite ends of a one-unit-wide corridor; their goal locations corresponded with the other robot's starting location. Like the first maze, a solution to this could not be found.
- A third maze, `inaccessible.maz`, had a single robot attempting to reach a goal that happened to be on the same square as a wall, not a floor. This maze, could not find a solution.
- The next maze, `blocked.maz`, gets interesting. One robot begins in a one-unit-wide corridor, its goal near the exit of the corridor. The second robot begins in the bottom left corner of the map; its goal is inside the

corridor, between the first robot and its goal. To solve this, the algorithm has the first robot exit the corridor (passing its goal on the way), allowing the second robot to enter before returning.

- Finally, in `two_path.maz`, both robots begin in each other's goals in one-unit-wide, circular pathway. The second robot most travel all the way around the path to reach its goal, even though both robots began only one space away from their goals.

Discussion Questions

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?

To represent the state of the system with `k` robots, `2*k+1` numbers are needed: an x and a y coordinate for each robot, plus a number representing which robot's turn it is.

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

For each tile in the maze, there can be any of the `k` robots (or no robots) on that spot. Therefore, an upper bound on the number of states is `(n^2)^k+1`.

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.

For each wall tile, any of the `k` robots on it would result in a collision. Thus, the number of states that represent collisions is around `w^k`. This does not include robot-robot collisions. However, if the size of the maze is much larger than the number of robots, robot-robot collisions represent a very small percentage of total collisions and are therefore computationally insignificant for the upper bound.

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?

Unless start and goal locations are very close for each of the 10 robots, a breadth-first search will not be computationally feasible for such a large maze and number of robots. For each state, BFS will explore every successcor state before moving to the next layer. Thus, the size of the queue will grow exponentially as the seach continues.

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 useful, monotonic heuristic function would be total Manhattan distance to the goal.

```
....G
...N.
..S..
```

Monotonicity requires that h(n) <= c(n, a, n') + h(n'); that is, the heuristic of a node is not greater than the transition cost to the next node plus the heuristic of that node. In the above example, the Manhattan distance of N to G is 2, making N's heuristic 2. The heuristic of S is 4. The transition cost from S to N is 2. Since 2 + 2 <= 4, consistency applies here. For all scenarios in the A* search, the monotonicity of the heuristic will hold.

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 is essentially a 3x3 maze in which 8 robots need to get to their goal locations. Here, a good heuristic function would be the total Manhattan distance for all the "robots" (numbers) to the goal locations because, as explained above, it will always underestimate the distance to the goal. Here, instead of walls, other "robots" may block this perfectly direct path.

The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this.

There are two sets in the state space: a set of states that can lead to a solution and set of states that have no solution (i.e., no amount of moves could being them to the goal). A* will always return a path if one exists. If A* returns an empty path, it can be determined that the starting state was a member of the set of states with no solution.

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 used total Manhattan distance as a heuristic for the A* search. Sometimes, these direct paths wil not possible (e.g., if there is a wall in between a robot and its goal); however, the heuristic will be admissible because it will always equal or underestimate the distance to the goal.

```
....G
.###.
.#S#.
.#.#.
.....
```

In the above example, the Manhattan distance from S to G is 4. The actual distance is 8, since the robot will have to go around all the walls. There is never a case in which Manhattan distance (the most direct path) will overestimate the distance to the goal, making it optimistic.

```
....G
.###.
.###.
.###.
S....
```

In the above example, the Euclidean distance from S to G is 4*sqrt(2), roughly 5.6. The actual distance is 8. Like the Manhattan distance, the Euclidean distance is admissible. However, since the Manhattan distance >= the Euclidean distance for all states, the Manhattan heuristic is said to dominate the Euclidean heuristic and will therefore guide the search better.