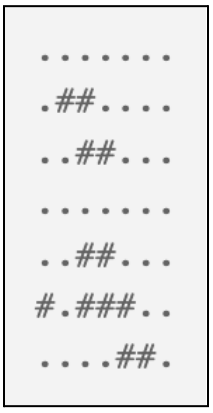
Lauren Kidman
COSC76: Artificial Intelligence 24F
18 October 2024

Report for PA2: AStar Search

Table of Contents:

The Assignment + *README

Here is a maze, drawn in the venerable tradition of ASCII art:



The periods represent open floor tiles (places that can be walked on or over), and the number signs represent walls, places in the maze where a simple robot cannot go. This assignment has two different applications of these mazes, both using a robot that can move North, South, East, West, and sometimes even stay in place, that depend on the A-Star search algorithm to find the most optimal paths depending on the assigned heuristic.

A-Star Search

---

1. Description

The premise of Astar search is to, starting from a specific point on a graph, find the path to the goal state that has the lowest cost–where cost is defined by the heuristic created for the problem–i.e. shortest path.

My implementation of the A-star search algorithm is pretty standard and straightforward–following the pseudocode from the class textbook, *Artificial Intelligence: A Modern Approach, 4th US ed*, and the in class lecture slides. The Astar search depends on Astar Nodes, which store information about the state, the total transition cost up until that state (g(n)), and the parent node.

2. Evaluation

The evaluation of my Astar function will be discussed more thoroughly when talking of the specific problem examples; however, generally the implementation worked very well.

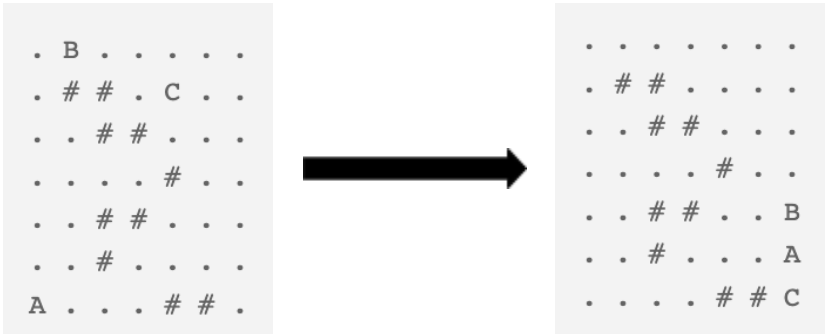
Multi-robot Coordination Problem

---

1. Description
    a. *Background pulled from the PA assignment:*

“k robots live in an n x n rectangular maze. The coordinates of each robot are (x_i, y_i), and each coordinate is an integer in the range 0...n-1. For example, maybe the maze looks like this:

That's three robots A, B, C in a 7x7 maze. You'd like to get the robots to another configuration.



There are some rules. The robots only move in four directions, north, south, east, and west. The robots cannot pass through each other, and may not occupy the same square. The robots move one at a time. First robot A moves, then robot B, then robot C, then D, E, and eventually, A gets another turn. Any robot may decide to give up its turn and not move. So there are five possible actions from any state.

Let's make the cost function the total fuel expended by the robots. A robot expends one unit of fuel if it moves, and no fuel if it waits a turn. Only one robot may occupy one square at a time. You are given a map ahead of time, and it will not change during the course of the game."

b. Implementation Description

My MazeworldProblem Class consists of five key functions: get_successors, get_cost, goal_test, manhattan_heuristic, and animate_path.

Get_successors is the meat of the class; its goal is to, given the current position of a robot, essentially determine what its next possible moves are given the layout of the maze (collision vs. floor) as well as the current positions of the other robots in the maze. The first index of states_tuple is used to indicate which robot's turn it is to move. From there, the robot then checks its position after all 5 possibilities (N,S,E,W,stay). If a position is legal, meaning there is no collisions with another robot or a wall, then it is counted as a successor.

Manhattan_heuristic is the other vital component of the class. It essentially finds the total Manhattan distances (think the grid-like structure of New York City; a robot cannot move diagonally or in some weird squiggle) of all the robots from their respective goal locations.

Get_cost can return one of two values, 0 or 1. The function checks if the robot's position has changed at all in order to see if it moved. If the position is unchanged, return 0, as the robot did not move; if the position is new, return 1.

Goal_test checks if the current position of the robot is its goal position.

Animate_path is a supplementary function given in the starter code that "animates" what the robots' paths look like using . and #.

I also for fun included a euclidean heuristic function, which obviously given the constraints of robot movement does not actually apply.

2. Evaluation

The class overall worked very well, and this is especially confirmed when comparing the results of the manhattan heuristic to the provided null heuristic. I made a variety of different mazes–ranging from large, such as a 40x40, to very small but barrier-heavy, and all mazes returned their results immediately without lag. With mazes 3 and 6, I compared the results, and in both cases a final path of the same cost as the null heuristic test was found, yet having explored much fewer nodes–a testament to the quality of the functions.

3. Discussion Questions
    a. **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?**
        i. The two main components that represent the state of this system are: location and turn. For robot location, you would need the numbers representing the coordinates of each robot, = 2k. For turn, you would need the indicator value like it is shown in the states_tuple. Therefore the system should be a tuple of size **2k+1.**
    b. **Give an upper bound on the number of states in the system, in terms of n and k.**
        i. In a given nxn maze (or something like nxm), a robot can be in n^2 possible locations. Furthermore, the more robots there are, the higher chance for collisions–a robot cannot share a space with another robot. So, as number of robots k increases, the number of possible states decreases, making the upper bound the sum of (n^2 - i), where i = k-1 (and k = num robots)
    c. **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?**
        i. No, I do not expect a straightforward BFS to be computationally feasible, especially based on how my code reacted to a 40x40 maze. With a maze as large as 100x100 and with so few barriers, that essentially means every space will have all five possibilities/valid successors, making time and space overly complex.

Blind Robot Problem (with Pacman Physics)

---

1. Description
    a. *A background from the PA:*

Assume that you now only have a single robot in mazeworld, but there's a catch. The robot is blind, and doesn't know where it starts! The robot does know the map of the maze. The robot has a sensor that can tell it what direction is North (so that it can still move in intended directions). However, the robot has no other sensors. No, it really can't tell when it hits a wall. If you execute the action "west" from a configuration where "west" is blocked, the robot simply doesn't move.

    b. My implementation:

For the sensorless robot program, my A-star search algorithm remained unchanged; Animate_path, get_cost, and goal_test are all identical. All changes between the sensorless robot and the multi robot program were between how the two search problems were implemented–because the robot is blind, get_successors no longer has a stay option, instead only being able to move up, down, left, and right. These movements are hypotheses, as again, the robot is blind, and can only guess where it can go. If a robot is running into a wall/obstacle of some kind, it simply becomes “more aware” of its location, and adjusts accordingly. For each movement, it adds its new location into a set (to avoid duplicates), which then becomes a tuple of all possible coordinates. The heuristic function for this problem has a more detailed check, making sure the goal path calculated is the absolute minimum cost, which I created in order to have variety of heuristics as well.

2. Evaluation

Since most of the implementation is the same as Mazeworld, the evaluation is very similar–the class performed very well. For the mazes I tested, such as maze 3, I confirmed that using the min_manhattan_heuristic accurately reduces the number of nodes visited in order to get a goal path of the same cost as the null_heuristic. Each maze variation I created for the problem printed out immediately without lag and without issue.