Aishwary Gupta

MLND

Date: December 3, 2017

Project Overview

This project is inspired by the Micromouse contests wherein a robot is put into a maze and is tasked with finding a way to the center in the shortest amount of time. Since mice were requent subjects of study in maze solving capabilities, and the robots in the contest attempt to do the same task, the contest name fits well. It can be difficult to the connection between maze solving and more complex tasks but it is a good representation of more interesting problems. An agent often needs to keep track of what it knows and make decisions to optimize for a goal.

Problem Statement

A robot works within a perfect maze in two different runs. In the first run, the robot explores the maze to determine its structure. In the second run, it aims to reach the center as fast as possible using the information it collected in the first run.

Metrics

The metric to judge the robot is calculated by taking the sum of the number of actions in the second run plus one thirtieth of the actions in the first run. The number of actions in both runs is limited to 1000.

Data Exploration:

The specifications for the robot are as follows:

- The robot always starts in the bottom left corner, facing up.
- The robot has 3 sensors, facing left, right and up.
- The sensors can see infinitely far, and return values corresponding to the distance of closest wall in the given direction.
- The robot can turn 90 degrees, -90 degrees or not rotate in an action.
- The robot can make an integral move in the range [-3, 3].

The specification for the maze is as follows

- The maze is of the dimensions 12x12, 14x14 or 16x16
- The maze has a goal section in the center 4 squares.
- The maze contains only unit distance walls.
- The maze is bounded on all sides
- The maze is solvable.

Algorithms and Techniques

There are several approaches to exploring the maze on the first run. The robot can choose to hunt down tiles that are closer to the center in an effort to discover a route to the center of the maze faster. Alternatively, the robot could instead decide to explore every node and try to find a way to cover the entire maze. Additionally the robot could decide to finish exploration early deeming itself to have enough knowledge of the maze to do the second run, or it could explore more with the expectation that doing so will yeild a better score in the second run.

The final algorithm that worked for me was to require the robot to explore the entire maze and build a graph where the vertices were the tiles, and two tiles would share an edge if the robot required only one step to move from one to the other. In the second run, we would then run Dijkstra's shortest path algorithm and then take it to reach the goal area.

Benchmark

Different exploration algorithms provide different quality data. We can use the provided benchmarking score to compare them. The score itself doesn't tell us if the robot found the shortest path, but its values across the board compared to other algorithms does.

Since the robot should explore the tiles and spend the minimum amount of time revisiting them because doing so yields no new information, we should expect early run times of perhaps around double the number of tiles in the maze.

Data Preprocessing

Since the given data is perfect in that the robot sees perfectly, there is no need for it. Internally, there was some processing, for instance, combining the robot's direction and sensor data into a normalized list to update vertices, but this was the main extent of the preprocessing.

Implementation and Refinement

The maze is stored as a graph. The vertices are the tiles, and two tiles are connected if the robot can move from one to the other. For instance, if (3, 4) and (3,5) are connected, then there is no wall between them. The tile object for (3, 4) will have a dictionary in which the key "N1" will have the reference to (3, 5) as its value. It can also be thought of in the way that for a tile, if there is a wall in a direction that prevents connection to the next tile, then the corresponding cardinal direction that is the key in the direction will have a value of None. If the key isn't even in the dictionary, then the tile has not been explored. The goal is to fill the dictionary for every tile until the coverage of each tile is 12. (We know what exists three spaces in all four directions for that tile)

Creating the graph is an involved process. For instance, to minimize computation we realized that after the first move, the front mounted sensor is completely useless. No matter what, since the robot will have to turn into a direction to move forward in it, the robot would already have seen as far as the turn goes. Hence, the forward facing sensor would not yeild any new information once the turn is already made. The robot then has to link together the tiles it does see with the correct edges. This is fairly simple as an idea, but tricky and

slightly unweildy to implement. The robot also keeps track of its heading and location.

The robot then provides the algorithm with the heading and location. The algorithm already has a reference to the maze itself. The algorithm decides the next tile to move to, figures out a way to get to it, and returns the first action in a series of actions needed to reach the chosen destination. Between different calls to next_move the robot can change its long term destination if a closer more useful option is found.

There were several complications as this was coded. The main one was attempting to determine if the robot really needed to explore the whole maze, or just a percentage of it. Another was trying to figure out the logic behind whether the robot should do two 90 degree turns or move backwards when in a tight situation. It was determined that the latter was the better option since the first option would yield no new information and would simply take up an extra move. A final complication was trying to determine the heuristic that the robot should use. Should it always move to the tile closest to the center? Or is it better to treat all tiles equally? At the end it was determined that the latter is a better heuristic since the former resulted in the robot wasting too much time criss-crossing the maze, and the added effort wasn't giving data of higher quality.

Of course, there was always the slew of bugs that come with coding something decently large. These could sometimes be difficult to diagnose, especially since they were "silent" errors and wouldn't cause crashes. The main reason for why the bugs existed was because the graph was somewhat large. Instead of stepping through the code, the main method of looking for bugs was noticing systematic problems in the program output or runtimes. For instance, in the dijkstra algorithm, it would take a very long time, but would complete, and give a good score. However, there really was something wrong with an algorithm, and it was only when a contrived maze was created to test the robot that dijkstra infinite looped – meriting a third look at which point the bug was finally resolved and the robot completed the runs through the maze much more quickly.

Model Evaluation and Validation

There is not much to validate in this model. The graph is perfect, in that it completely describes the maze. The algorithm has also been proven correct to find the shortest path. The only piece of software left that needs validation is likely the algorithm that searches for the next waypoint. This is fairly robust as well, since it is a breadth first search of the graph from the robot's position to find the closest tile that is not fully discovered.

I would be remiss if I didn't remark here that because of my method of implementation there is little machine learning happening. There aren't a lot of unknowns, and no real complex decisions to make. As such, it may be better to improve on the problem a bit by introducing more complexity that cannot be modeled completely by a relatively small graph.

Justification

The algorithm that uses Breadth First search to find the tile closest to the robots current location gives us 21.133 for the first run, 27.833 for the second run and 32.667 for the third run. These are reasonably fast times. Using full dijkstras on every single move provides more information but ultimately gives very marginally different scores, simply because of how python might store its keys. Ie, the algo will pick one of two different tiles that are both the same distance from the current location which results in different scores.

Free Form Visualization

A fourth test maze was created with the intent to highlight the robot's shortcomings. Doing so yielded unexpected results. The idea as to show that the robot would insist on exploring the entire maze, thereby taking up many moves, when a much shorter path is available to it right at the start. And to explore the new path, the robot would have to go all the way back to the beginning and finish up the maze from there.

In reality, the algorithm looped forever. Thankfully, the algorithm was fixed, and while the robot did explore the entire maze as was its programming, it also realized that a very short path existed. Since the number of steps taken in the first run is not as important as the number of steps in the second run by a factor of 30, the algorithm gave us a score of 11.067 on test_maze_04.

Reflection

The problem in the beginning was finding a way to keep track of what the robot knew as it explored the maze. Q-Learning was considered as were some small state machines that took the position of the tile and distance from the center into account but a graph arose as a natural choice once it was seen how easily it describes the maze itself. Given that there are so many graph algorithms that could be used in general, it also served as a good launchpad to developing efficient path-finding algorithms.

Over complicating the problem was another issue that had to be addressed. Assigning new data to each of the squares was unnecessary. Since the robot has to explore the whole maze anyways, it will simply always search for the closest square and always be moving towards it, or perhaps instead calculate something interesting like cluster viability – whether a certain section of the graph is relatively unexplored and is likely to yield the most information in the shortest number of moves. As a possible extension, I would try the cluster chasing method to gain more information faster as opposed to hunting squares closest to the current location.

The solution arrived upon at the end was to completely explore all reachable nodes and then start the second run.

Improvement

Extending this to the continuous domain requires a large amount of software that interfaces with the robot's physical surroundings. This would include getting accurate readings from sensors, executing turns and moves that ensure the robot is close to the center of each tile at the beginning and end of each move, etc. For instance, if my robot moves to the center of the next tile, but

is a milometer off, and is a milometer off on every move, eventually the sensors will give values that the robot can't decide between – why is the distance in between two integers? The sensors would have to give floating point data so that the robot can do automatic position correction.

There would have to be a fair amount of error detection or adjustment. Sensors would have to be correctly interpreted for the robot to learn that it is moving a little too far every time, or is turning 1 less degree than it should, etc. This process involves more machine learning than the algorithm to solve the maze itself. Once the robot has discovered its misalignment, how would it fix itself?

If the walls are still unit length, then I don't see why the robot could not solve the physical maze. Unless the maze walls became curved, the robot can still use the graph model as described in the current algorithm – the issue would be keeping track of the robot's current location. This again should not be too difficult since the robot knows how much a unit distance and wall length is.