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 connect maze solving and more complex tasks but the methodology and algorithms used in maze solving provide a good, simpler representation of more interesting problems. An agent often needs to keep track of what it knows and make decisions to optimize for a goal.

A real world example is path finding. A robot car could represent the roads and intersections as actions and tiles respectively. The model would be a little more complicated since the roads would have more than a unit length, but our techniques here are a strong basis for an initial solution to a robot exploring a city or any general unknown environment. It doesn't have to be just on the roads, it could be offroad in challenging territory. The robot could map out a path through a battlefield that is safe for transport, or in some kind of urban search and rescue operation where the goal is to find survivors in a collapsed building, and then relay the directions to the survivors to other robots or a human team.

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.

To explore the maze, the robot will create a graph to record its observations, while at the same time making decisions to most rapidly grow the graph until it can be used to find a path to the center. The robot will use a breadth first search, using its current location as a root node to find the closest tile that it has seen and knows how to get to, but doesn't know where else the tile could lead. The robot sets the tile as a waypoint, and then determines the direction to it. Since the breadth first search on the next move may return a more attractive tile to explore, the robot may abandon its previous waypoint.

Once the robot has found the goal area, it will use a shortest path algorithm to find the shortest path through the graph to the center.

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.

This metric makes sense in two different ways. The first is that the second run is much more important than the first. This emphasizes the idea "Take the time you need to learn, but when test day comes, you have to be able to perform"

which is useful in practice. For instance, a facial recognition algorithm should spend a good time learning as it needs, so that when test day comes, it determines if the face is a match to unlock the iphone X or not. Scoring a robot critically on performance runs ensures the agent will perform reliably.

The second reason this makes sense is that the metric does take the train time into account. An agent needs a reasonable amount of time, but due to finite resources cannot have infinite. There needs to be some kind of trade off. An Iphone X user doesn't want to stare at his phone for two hours while it tries to determine the exact 3d model of his face down to the molecule. (Oddly enough, the phone does determine a 3d model to map the face preventing people from tricking it using photos). Aside from jokes, scoring a robot on its train time prevents it from assuming it has unlimited resources – making it viable in a given application apart from purely theoretical.

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

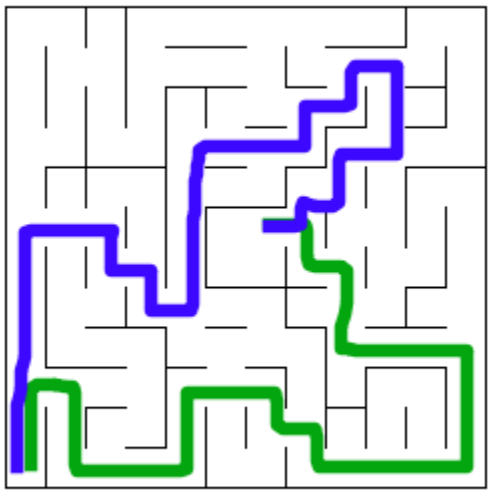
- The maze is of the dimensions 12x12, 14x14 or 16x16
- The maze has a goal section in the center 4 squares.
- The maze contains only unit distance walls.
- The maze is bounded on all sides
- The maze is solvable.

The structure of the 12x12 maze was the motivation for the state at which the algorithm for solving the maze is currently in. Initially the algorithm didn't both searching the entire maze, focussing on discovering around 75% of the tiles, maybe 85% before feeling that it was good enough to begin the second run. Also, the algorithm had a way of deciding which tiles to explore – the ones near the center. But the 12x12 maze had a tile furthest from the center that was integral to finding the shortest path. This structure and the decently small size of the mazes themselves made it possible to ask the robot to explore the entire maze before proceeding to the second run.

It is important to point out that the maze did have more than one solution, and that my robot would take the solution where it went over the goal area, looping around the mini staircase at the top right to get into the goal area. This maneuver around the staircase involved lots of single step actions which was bad for the robot score. Going the route near the bottom was better bcause although the path length was around 30, the actual needed number of steps was only 17.

Exploratory Visualization

Here is a picture of the 12x12 graph with two potential solutions drawn onto it. The green path is the recommended one, since it takes just 17 actions, whereas the robot takes 20 actions to reach the goal on the blue path. This graphic highlights the importance of taking the extra effort to search all the squares. The square in the bottom right corner is on the shortest path, and the robot would have to visit it to know this. (Or at the very least visit the tiles that can see it)



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.

For exploration, the Breadth First Search algorithm was used. The search would pick the robot's current location as root of the tree and then generate a shortest path tree. As it processed a tile, it would create an entry in a local dictionary with the tile as the key and the parent as the value. That way, when the first tile was found that was not fully discovered, the algo could simply use the locally generalted dictionary to determine the path of tiles to its next objectives. This saved computation time and turned the process of finding a waypoint and directions to it into one single operation. Breadth first search was also useful because it terminated as soon as it found the first undiscovered node. It didn't have to calculate the entire shortest path tree for every next_move() call, it would only calculate what was necessary.

The above are the pros of the algorithm: efficient and thorough but there are cons as well. This method doesn't value any tiles as better than others – when some tiles do deserve preferential treatment. Going by heuristics alone, it is not likely that the tile at the opposite corner of the start location will have to be searched most of the time. It seems that only a contrived maze would force a robot to use that square for a shortest path. That opposite tile isn't as important as exploring the tiles closer to the center. The algo does not take this into

account. Another issue is that the algorithm doesn't search for the robot doesn't even search for the shortest path to the center until it has explored the entire maze. There is likely some logic that would allow the robot to determine on a particular move if it is even possible to find a shorter path than the one it has. In this particular problem case, it doesn't seem like that much extra effort to explore the whole maze – but this option is rarely present in real life. Google maps will not calculate the entire map of San Fransisco to help you find a quick route to a coffee shop. And at a certain point, this method becomes unweildy. The number of squares in a maze grows in quadratic fashion as the width of the maze grows by one. At some point, this algorithm will not be as efficient as smarter decision making agents.

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.

Since our bench mark is based on the robot taking double the number of tiles in the maze as on the first run, and near a perfect run on the second, we can say that the base scores will be:

12x12 score = 144 * 2 * (1/30) + 20 = 29.600
14x14 score = 192 * 2 * (1/30) + 25 = 38.066
16x16 score = 256 * 2 * (1/30) + 30 = 47.066

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.

This project saw many refinements materialize as development of the solution progressed. Each time a refinement came up, there was a specific problem it was addressing.

1. Giving tiles the references to not just tiles that are only a unit distance away, but up to three unit distances away. This is useful because in the initial version of our program, it was so simple it wouldn't make full use of the robot's suite of possible actions. Allowing the robot to take more than a step at a time helped drop down the end score by around half. Also, the algorithms would technically be finding the wrong paths to tiles – if two paths to tile T are found, (A and B) and the number of single steps in B is higher than the number of single steps in A, the algorithm would choose path A. However, the robot could feasibly reach path B in less steps using moves that cover more than one tile in a step.

2. Removing the heuristic of how much of the maze the robot should learn. It is hard to decide for any given maze if the robot has seen enough to know that there are no better solutions for the second run. There is also the problem of overfitting this value to mazes that we have especially since we have such a small sample set.

3. Switching priorities from chasing tiles in the center to prioritizing immediate information gain. Initially the robot would hunt tiles closest to the center. As I realized it would make lots of zig zag movements across the map to explore tiles that were marginally closer in lieu of exploring tiles that were right next to it, I programmed in the new priority. This helped the run time by quite a bit. The robot took a lot less time to explore and delivered better results. It's not folly to think that tiles near the center are more important, but there needs to be levels to it. Those tiles aren't that much more important that we abandon what we have. This refinement improved run times by a fair amount.

4. Recoding the entire project to provide stronger modularity and readability to the code. The first iteration of the robot was fraught with bad coding practices. Loops themselves looked like someone had done some copypasta with C code and replaced the curly braces with colons. Nothing was pythonic and the class capabilites of python were not really taken

advantage of. There was also high coupling – it was not easily distinguishable where the robot class ended and where the algo took over – if there even was an algo class. The new implementation improved on all of these. Old algorithms could be subclasses of an Algo class and then easily swapped in the robot's constructor. The code is much more readable and the algos themselves are provided with a larger but better graph wherin the graph better represents the robot's environment.

5. Switching from Dijkstras to Breadth first Search to minimize computation. Building a tree of the entire graph at every step is costly – and since each edge has a distance of 1 thanks to the robot's newer, more sophisticated graph updating protocol, the Breadth First search as a feasible alternate implementation makes sense. In fact Dijkstras wouldn't even work if the graph didn't support all of the robot's actions from each tile. The BFS builds the tree up to its needs, which is what makes it faster.

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.

After some consideration, it was decided that the best way to determine that our algorithms really were correct and could generalize to other mazes would be to create the other mazes themselves. Writing a program to randomly generate solvable mazes is not a particularly trivial task – from this time of writing. The script to create the maze would first use a random number generator to decide wether a wall would exist at a position. The grid would be filled in with hard coded values for certain tiles (edges, start tile, goal area, etc) and then the values of the tiles themselves would be calculated as per the showmaze.py script requirements. The tile values would be printed and then the showmaze script would be used to see a visual of the maze.

The randomly generated mazes themselves were quite interesting – many of them would block the robot or various sections of the maze completely. To make the maze solvable, I'd have to manually remove some walls to connect areas together. To minimize my manual effort, the maze sizes were restricted to 12x12 mazes. It was fun being adversarial – knowing that my robot would map out another 50 squares because I'd eliminated one wall and that none of those squares would lead to the goal. Such is the nature of the maze itself, but in creating all these mazes, the faults of my algorithms became ever more pronounced. In some of the mazes it was glaringly obvious that a robot would be wasting time searching in a corner when it could be calculated that any path that went through the robot's current location could not be shorter than one it has already found. In any case, 10 mazes were generated as our dataset. The maze files are included with the submission.

The psuedo-random mazes were used to show that our algorithms do improve in performance across the board with refinements and reliably solve mazes. It also indicates that we haven't overtrained on any of the three mazes, since the performance of the robot doesn't waver wildy on our new test set. A particular algo would be run across the 10 mazes and its average and range of scores would be calculated.

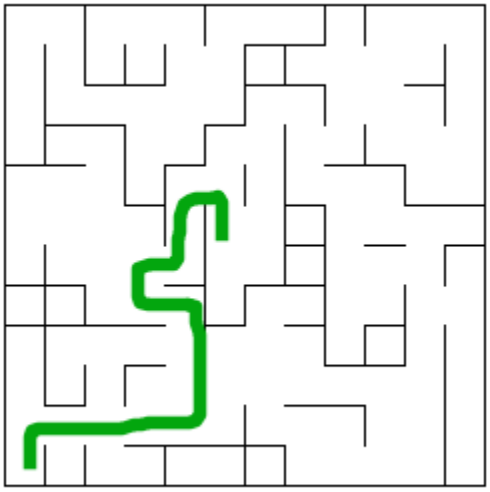
Algorithm	Average Score	Worst Score	Best Score	Range
SingleStepCenterChaser	29.400	41.133	21.433	19.700
CenterChaser	26.303	39.233	18.567	20.666
Dijkstras	20.950	30.857	13.133	17.724
Bread First Search	21.187	31.467	13.133	18.334

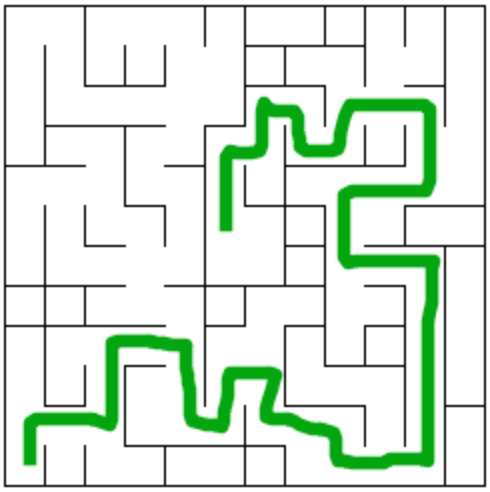
A glance at this table shows that the both the average, best and worst case improve with each row up to row 3, where the differences are more or less minute. The range is included to show that our results are a little thrown off by the length of the shortest paths in the maze. Most of the results were closer together rather than being a whole 20 points apart. All this means is that there

was one maze whose shortest path was very long, and another maze's shortest path was – short. If we exclude the exclude these two extremities, the range drops to close to 6-7 across the board.

On a final note, a future extention would be to implement an algorithm that would decide to cut short investigation of a maze before discovering every node and see how it fares against our current algos. Another avenue of exploration would be in trying to generate mazes that are larger, and also include the logic in wall placement to ensure solvability. The justification in the next section outlines how each of these algorithms work.

Two of the generated mazes are shown below, the first is the one in which the robot would have very low scores, and the second very high scores. It is easy to see why – the second maze requires the robot to undergo many twists and turns to get to its destination while the first maze has a very straightforward path.





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.

The main reason that our model is very robust is because it is very deterministic. The current code demands a complete run of the maze. This ensures the robot can calculate the shortest route. However, it is important to back this with statistical evidence that our robot does indeed solve mazes quickly. As such, here is some data on how the robot performed with earlier algorithms as compared to now.

Algorithm Used	12x12	14x14	16x16
Single Step CenterHunter	33.267	40.067	53.467
CenterHunter	28.133	36.667	45.233
GreedyWalk – BFS	21.133	27.833	32.667
GreedyWalk - Dijkstra	21.133	27.767	32.833

From the above data, we can see how the robot improves across the board. The Single Step Center Hunter spent too much time running around the same tiles, and only ever took one step at a time. The CenterHunter reduced the times by allowing the robot to move up to 3 tiles at a time, assuming that the movement was possible. This reduced the traversal time by quite a bit. Since it's expected that they all found the shortest path, we're seeing score differences of around 4 to 8 – that's saving around 120 to 240 moves! The next big improvement is changing from being a CenterHunter to looking for the closest undiscovered tiles – which improve the exploration times by around 200 to 300 moves.

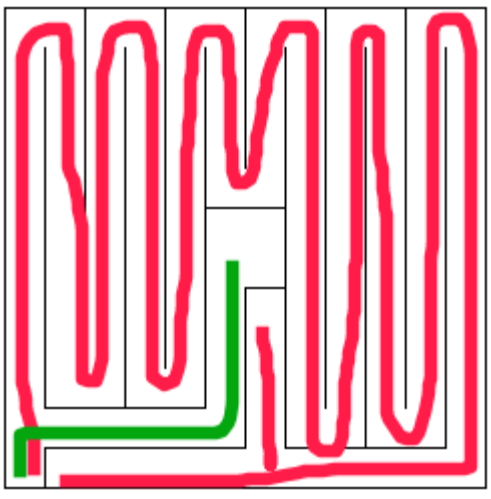
Finally, we should compare our results to the benchmark made a while ago. Our final algorithms – BFS and Dijkstra perform well under the benchmark, as do their averages for the randomly generated 12x12 mazes. In fact, the only algorithm that seems to do worse than the benchmarks is the Single Step Center Hunter.

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, which highlighted a subtle bug. 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.

This highlights the robust nature of our algorithm. It is complete in that it will find the shortest path, even if it makes the wrong decision at the beginning. It is worth noting that it is unclear how an algorithm might realize that taking the right turn at the tile (0, 1) is the correct move. Perhaps at 50% completion of the maze, it would backtrack to explore tiles closer to the starting square.



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.