Machine Learning Engineer Nanodegree

Capstone Project - Robot motion planning - Micromouse Simulation

Rob Fitch

May 5th, 2017

I. Definition

Project Overview

My interest in robot motion planning stems from curiosity about the functioning of autonomous vehicles and systems. Due to resource constraints, I am currently unable to build and test a self-driving car. For now, I have instead turned my attention to the related, though more narrowly defined and decidedly less resource intensive area of micromouse competitions. A area strongly related to robot motion planning and novel environment exploration.

A micromouse is a type of robot used to compete in applied maze solving challenges. Formal micromouse competitions date back to the late 1970s with their modern rules codified in the early 1990s. For the competitions, fully autonomous robots must navigate an unknown maze from the southwest corner to the center. This is done in an exploratory phase followed by a timed phase. Together, these phases are limited to 10 minutes. During the exploratory phase, the robot attempts to find paths to the goal and create an internal representation of the maze. For the timed phase, they use those maze representations to race to the center.

With the requirement for total autonomy, the robots must be self-contained for the full competition duration. This makes hardware, battery and even chassis selection a trade-off in capability versus longevity. A balancing act made far more sensitive by the size restrictions of the competitions, no more than 25 mm in length and width.

Previous research in this domain presents a range of algorithm choices; Dead Reckoning, Wall Follower, Waterfall/Flood Fill, Modified Flood Fill and even Neural Networks. Of these, the flood fill variations appear to be the most competitive.

This project will simulate a micromouse using python with the goal of implementing an effective maze solving algorithm. To measure algorithm effectiveness, each algorithm will be tested on 3 mazes, measuring 12, 14 and 16 cells on a side, respectively.

Problem Statement

My implementation of the maze solving algorithms will rely on the assumptions made by the environment about the performance of the micromouse itself. First, it is limited to 90 degree turns. Second, within a single time step it can move no more than 3 cells, forward or reverse. The environment tests each cell of movement to verify is legal, stopping the movement if a wall is encountered. Third, the robot receives sensor data at the start of each time step. These sensors face to each side and forwards, measuring the distance to the nearest wall in the given direction. Fourth, during the exploratory phase, after the robot finds the goal, it may continue to explore or start the speed phase at any time. When the robot starts the speed phase, it will be instantly returned to the starting position at no penalty. These assumptions will allow direct comparison of algorithm performance by holding all hardware attributes constant. An algorithm that finds the goal more quickly, all else held equal, is superior at the task.

Metric

The measure of algorithmic performance used in this project will be the speed phase duration added to 1/30th of the exploratory phase duration, in time steps, with lower scores being preferred.

$$t_{explore}\frac{1}{30} + t_{speed} = t_{score}$$

This calculation, in seconds rather than time steps, is the metric used by micromouse competitions. If an algorithm takes 120 time steps to explore the maze and 25 time steps in the speed run, the final score for the algorithm is $100\frac{1}{30} + 25 = 28.333$ time steps.

II. Analysis

Data Exploration

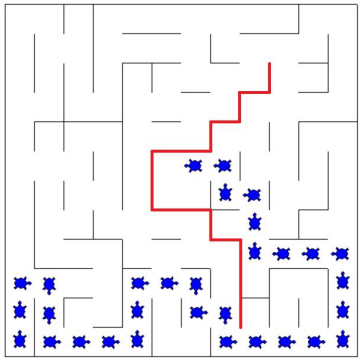
Original versions of the simulation code is available on the Udacity github repository under Robot Motion Planning. Both the Maze class definition and the test mazes are unchanged from their original form while the show maze utility was restructured into a class and is now accompanied by the show robot utility. These were used to generate live visualization of algorithm performance during testing, resulting in several of the images in this report.

A micromouse compliant maze has several distinguishing features. First, the starting cell is in the lower left corner of the maze, called the southwest corner. This cell always opens to the north. The goal of the maze is comprised of the 4 cells adjacent to the

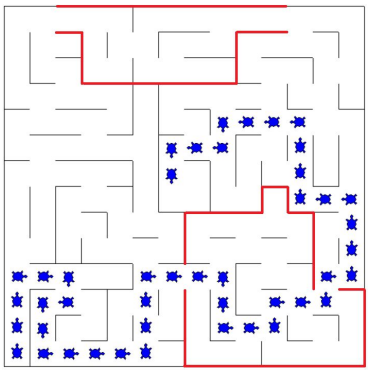
center post of the maze. This center post never has walls adjacent to it, this is the only post in the maze that has no adjacent walls. Mazes will include loops and dead ends and multiple paths with similar performance characteristics, increasing the exploration cost to the benefit of more complex algorithms.

The 3 test mazes comply with the above maze definitions. They are measure 12, 14 and 16 cells per side and are visualized with their optimal routes in the next section. The variety of available routes are effective at thwarting the efforts of Wall Follower algorithms and though similar in length have enough variation to reward algorithms that continue exploring after discovering a first valid route.

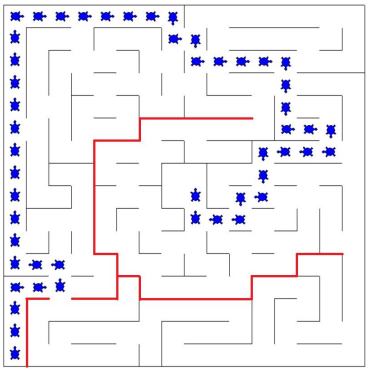
Exploratory Visualization



Test Maze 01 is marked by a vertical wall that separates the east quarter of the maze from the rest, highlighted in red. It limits valid paths, forcing them to pass through one of the bottlenecks at either end of the wall. This structure creates 3 primary routes with comparable characteristics, 2 to the north and one to the south. Each route starts from a different eastward turn from the initial northern run. The calculated optimal path runs along the southern edge of the maze before approaching the goal, shown with the line of blue icons.



Test Maze 02 has 2 primary routes, each passing through a large closed area, as shown in red to the right. These closed areas have only two openings for entry or exit, reducing the possible effective routes to one each. With this maze feature, agreement between the algorithms on the best path seems more likely. Again, the optimal path is highlighted in blue.

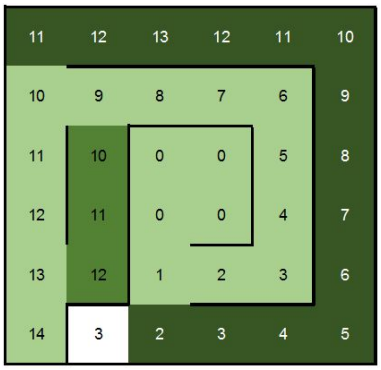


Test Maze 03 also has 2 primary routes, splitting at the first branch in the path from the start. The southern path enters a closed area immediately and stays within it until it is near the goal. Despite this advantage it is less effective than the optimal northern path. That optimal path has numerous loops and branches offering similar performance in the

northwest corner of the maze. It is notable that the goal sits in the center of a mostly complete circular wall. This again reduces the possible routes in the maze.

Algorithms and Techniques

Review of previous work shows that Dead Reckoning, an algorithm that makes a random choice at each intersection, has a history as a benchmark for this task. Setting the standard of performance for all other algorithms to be compared against. A more reliably performing option would be a wall follower, which chooses a wall and follows that wall through the maze. Unfortunately, it has issues with mazes that contain loops, a feature of micromouse mazes.



I intend to use a more memory and processor intensive algorithm known as Flood Fill or Waterfall. The diagram to the left demonstrates a Waterfall algorithm matrix on a 6 cell wide maze. The goal cells are marked with a value zero. Each blank cell accessible from a marked cell is marked with the next higher number, repeating until every cell is marked. The space next to the goal not blocked with a wall receives a value of 1. As the robot explores the maze, newly discovered terrain forces the Waterfall map to be recalculated. With sufficient exploration, this should always result in the best route through the maze. In the diagram, progressively darker greens represent less favorable paths. The white square is a dead end and does not contribute to any path.

My approach will utilize a 3 dimensional matrix to represent the robot's knowledge about the maze. The matrix will have the same width and length as the maze, containing a cell for each cell of the maze. In addition, it will have a number of layers based on the needs of the algorithm. For all of the mapping algorithms, the first layer of the maze contains a map of the discovered walls. Each cell contains an 8 bit integer representing the wall layout of the cell based on headings. One bit in the cell value is assigned to each heading, in order, North, East, South, West, making the wall values, 1 for north, 2 for east, 4 for south and 8 for west. This gives each possible configuration of walls a unique value and helps ensure the robot does not attempt to drive through walls. Additional layers will be added for visit tracking, flood fill or dead end marking depending on the needs of the algorithm.

Benchmark

My intent to follow the norm in maze solving algorithms and use an implementation of Dead Reckoning as my benchmark implementation, did not survive initial testing with

the algorithm. In the allotted 1000 time steps, it failed to solve either test maze 02 or test maze 03. This would have any algorithm that can find the goal exceeding the benchmark, I have chosen to use a Wall Follower instead. To ensure it always completes the maze, I added a visit counter to the algorithm. By setting the algorithm to prefer cells that have had fewer visits, I found that it would quickly escape from loops and explore novel territory. It performs adequately to the need and stands as the minimum performance benchmark.

After further review of the problem space, I concluded that there is a definite maximum performance as well. In the interests of finding and verifying this maximum, I built a model named Oracle Waterfall. The images with blue icons used throughout this project are the results of this model. It is initialized with the full maze layout, flood fills the maze, then uses a recursive search algorithm to find the best route based on the limitations of the robot. This model simulates a case where a robot accidentally stumbles upon the perfect route on the first try and uses only that route, or a case of cheating. This ensures any model outperforming this standard should be regarded with suspicion.

None of the models I use in the benchmarks or evaluation for this project contain a random element. As a result any given algorithm will receive the same score on every run through a given maze.

III. Methodology

Data Preprocessing

All data preprocessing is handled within the Maze class. It receives the contents of a test maze file and decodes those values into a virtual maze. This preprocessing function accompanies the test maze files in their repository. After review of the class definition, I did not feel it needed modification.

Implementation

The initial files provided space for the algorithm to be implemented directly in the Robot class. I chose to move it into a separate Algorithm class, using the Robot class as a shell and interface, allowing experimental Algorithms to be passed to the test robot as arguments. This gave me the opportunity to quickly test and replace as needed without changes to Robot.

Within robot, I added a function to use the current heading to translate the sensor inputs from the native 3 direction form into a oriented 4 directional form. Ensuring that North, East, South and West were represented by a value at every time step. The value -1 was used as a placeholder for missing values to indicate the blind spot of the robot. To allow

this sensor input translation, the robot needed to track which direction it is facing, its heading, at every time step. A dead reckoning implementation was added to Robot, as a template for the algorithm interface, and to allow for functional testing of the class without an external algorithm.

The Wall Follower benchmark has been designed as a general base class for algorithms. It creates the 3 dimensional representation of the maze with 2 layers and initializes the outer walls. Every time step, this algorithm notes the wall configuration of the current cell and increments the number of visits to that cell. This class also implements the functions required to translate a cell value into walls, mark additional walls as they are discovered and translate between changes in heading and rotation. Finally, it provides the means to convert a heading value into the coordinates required to move a cell in the direction of the heading. For example, moving north by adding (0, 1) to cell (4, 4) would reach cell (4, 5).

Waterfall is built on the Algorithm class, capturing the Waterfall specific algorithm features and implementing a functional basic waterfall algorithm. It implements the waterfall generator which is regenerated every time step. As a result, I chose not to store this between steps.

I ran into problems with my version of the Waterfall algorithm. I noticed that during the speed phase, it would use a path not previously explored, despite the best and second best paths having been already explored. This is an artifact of unexplored spaces in the waterfall representation having no walls. I tried to implement a visit check, similar to my wall follower as a solution to this issue. My attempts to combine distance to goal with visits resulted in an unpredictable and poorly performing algorithm. As an alternative I added a lap counter to the algorithm. This counter prevents the robot from ending the exploration phase until it had moved from start to goal or the reverse, a number of times based on the size of the maze. This reduced the errors made by the algorithm during the time critical portion of the simulation.

Lastly, the implementation of the Oracle waterfall adds a function for extracting all of the walls from a Maze object and placing them into the Algorithm representation of the maze. As it is assumed to know the maze, it spends no effort on exploration. This algorithm exists solely as to explore the upper limit of performance on this task and cannot function without the collusion of the tester. It is included here for completeness.

Refinement

Initially, my intent was to build a waterfall algorithm and tune it for performance. During testing I discovered that while the waterfall method is excellent for exploration, it is

prone to continue exploring during the speed phase. In trying to resolve this issue, I moved towards using a recursive search to find ideal routes. It was the implementation of this search that gave me the opportunity to build my Oracle benchmark and further clarify the search space.

The adjustments to the basic waterfall required to incorporate the recursive search resulted in my final experimental algorithm, Search Waterfall, which takes advantage of the fact that I am recalculating the waterfall representation at every time step. This class implements 2 optimizations, recursive search and moving the goals.

Recursive search works through the waterfall map seeking out all of the valid routes. These routes are then refined based on the known performance of the robot, resulting in instruction stacks with each instruction being optimized to those limits. The algorithm then selects the stack with the fewest instructions.

Moving the goals involves calculating the waterfall map from a list of targets. At the start, these targets are the goal cells. However, as the robot explores and starts planning the best routes, those routes will pass through unexplored cells. Those unexplored cells become the new targets, causing the robot to detour to explore them and encouraging it to explore areas it believes contain optimal routes. This continues until the robot settles on a fully explored route, which then becomes the instruction stack for the speed phase.

IV. Results

Model Evaluation and Validation

Each of the algorithms was evaluated on each of the mazes. For each maze, the Oracle outperforms by wide margin, an expected result for a model able to skip exploration. The Wall Follower also performs as expected, setting minimum level of effectiveness.

All of the Waterfall algorithms are inherently reliable, methodically moving through the space with the assumption that cells on the route to the goal will be closer to the center of the maze as well. Neither of these presented results outside an acceptable range.

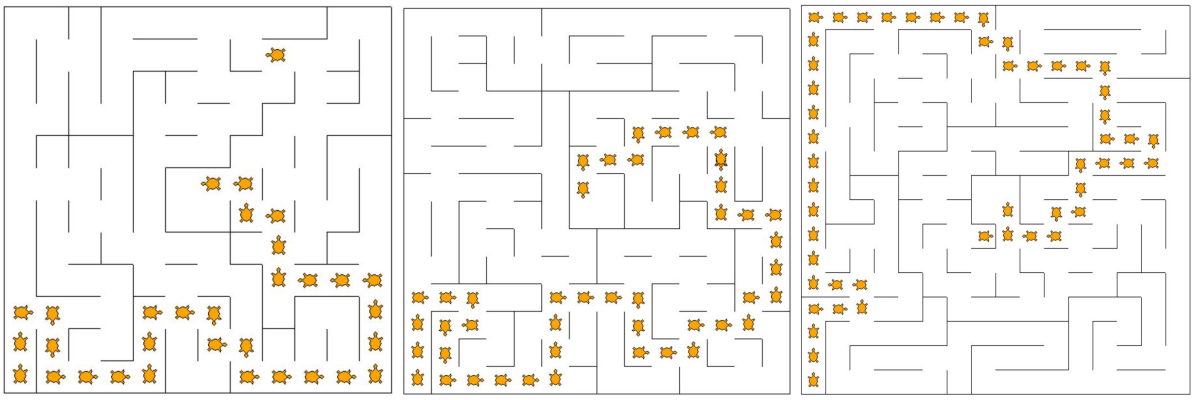
I found it surprising that, as shown in the chart below, Basic Waterfall outperforms Search Waterfall on Maze 02. It appears the limited exploration of the simpler model is able to take advantage of the restricted options within that maze. Maze 03 shows the

fault in this with the numerous options presented by the northwest area. Basic Waterfall's worst showing was on the only maze at competitive size (16 cells across).

Search Waterfall consistently finds the ideal route through the maze. Unfortunately, with the current implementation of the recursive search, the model is not efficient. Further refinement would be needed to attempt to use this model in a competitive environment. Below are the scores for each algorithm on each maze. The best non-benchmark score for each maze is highlighted.

Final test results	Maze 01	Maze 02	Maze 03
Wall Follower	100.633	272.733	137.933
Basic Waterfall	25.533	23.400	41.067
Search Waterfall	24.567	31.100	37.433
Oracle Waterfall	17.600	22.767	25.867

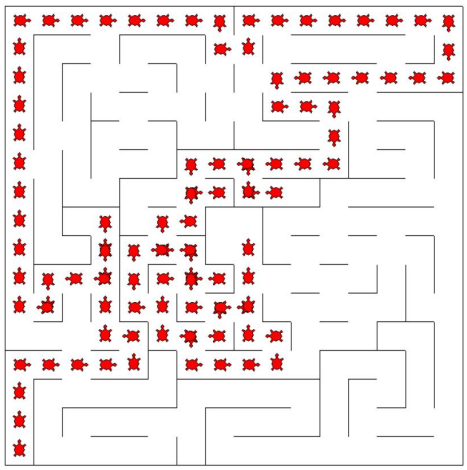
Justification



By moving the goals, Search Waterfall consistently finds the best path through the maze. This search method should result in significant response time improvements during the speed phase, shifting all of the decision making into the exploration phase. This design is intended to improve response time when it is most critical.

V. Conclusion

Free-Form Visualization



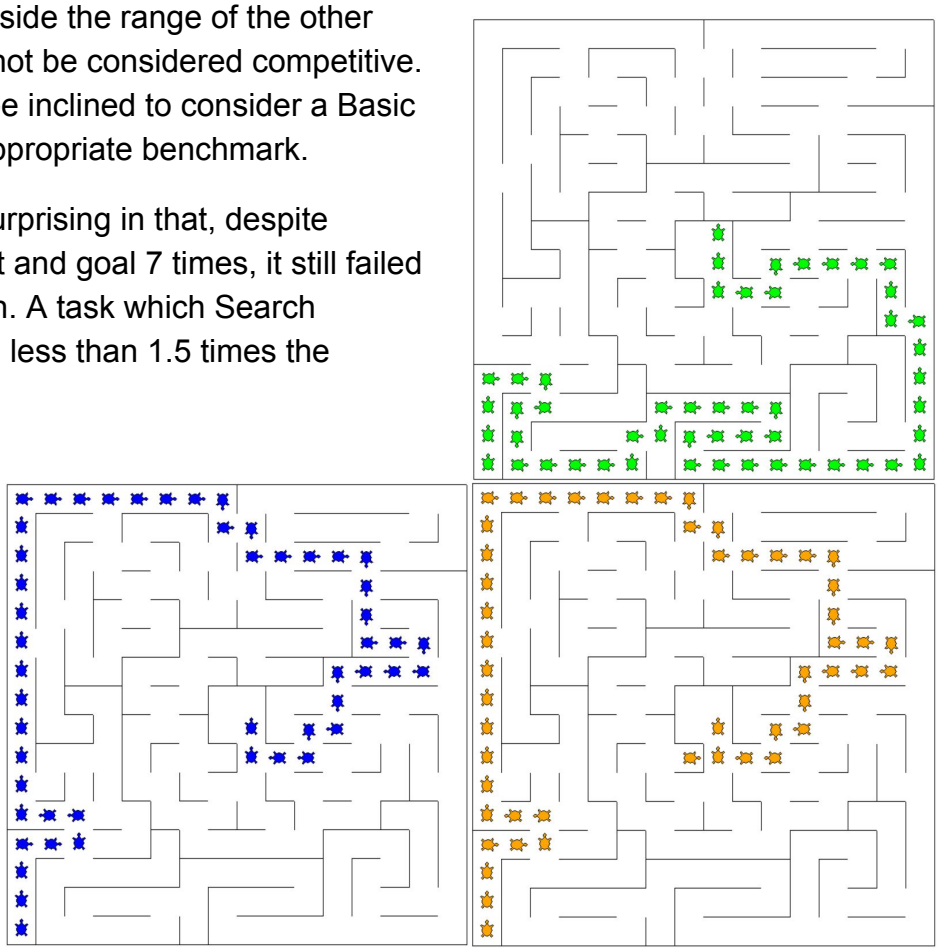
As the only maze at competition size, I consider performance on maze 03 to be the most indicative of algorithm performance. As a result, the speed phase for each of the algorithms on maze 03 is shown, Wall

Follower in red, Basic Waterfall in green, Search Waterfall in orange and Oracle Waterfall in blue.

In this maze format the Wall Follower performed terribly. It was well outside the range of the other algorithms and could not be considered competitive. In the future, I would be inclined to consider a Basic Waterfall as a more appropriate benchmark.

Basic Waterfall was surprising in that, despite crossing between start and goal 7 times, it still failed to find the optimal path. A task which Search Waterfall completed in less than 1.5 times the minimum steps.

I was interested to see that the best possible time, that seen on the Oracle, rose more slowly than the size of the maze did. This could be an indication of a trend in maze exploration.



Reflection

Initially, my approach was to find a way to use a depth first search to find a route to the goal before refining it in the hopes of locating an ideal path. Theoretically, a depth first search would allow the algorithm to find the goal then work back to the start, improving the route along the way. This ignored the bulk of the research in maze solving and presented a roadblock to my progress.

Once I started delving into published work and compiling lessons learned from those projects, I started to follow in their footsteps. My first foray was an ill-fated attempt at the Dead Reckoning algorithm. This implementation became a tool to unit test my robot implementation as I had little other use for it. As I explored the standard wall follower, it

quickly became apparent that some form of priority weighting would need to exist to direct the robots explorations. This priority weighting, it turns out, is known as Waterfall.

As stated before, my first Waterfall algorithm was is too fond of exploration. Attempting to combine the descending Waterfall metric with the ascending visits metric resulted in unpredictable, poor performance. To pursue this further, I will need to find a better way to combine these. My attempts to resolve this lead me back to planning and it was in reconsidering my approach to flood fill, and testing my ideas, that I found my way forward.

Experimenting with waterfall algorithms brought me back to using more generalized search algorithms, recursive tree search, in this case. With the addition of the recursive search function, I was able to demonstrate what a perfect performance would look like and used that as a base to try recreating that performance in a search pattern. The success was not stellar, but it points at a way forward.

To aid troubleshooting, I refitted the show maze function from the Udacity repository into a class with an accompanying show robot class. With these, I could watch every step the robot took and examine behaviors I found unwanted or unexpected.

These visualizations helped immensely in resolving my missteps. They assisted with simple issues like obviously invalid instruction stacks, single digits instead of paired instructions, caused by an incorrectly stated base case. Or in more complex issues where the robot drive into a wall, then attempted to reset the maze, caused by an off-by-one error in the route validation function.

They also helped when, while troubleshooting an issue in the Basic Waterfall, I broke the functionality of the Wall Follower. This caused me to re-examine my program flow and separate the robot function from the algorithms.

Improvement

Inefficiencies in the algorithms are 3-fold. The 8-bit maze representation uses only half of the memory reserved. Improvements here would reduce the memory footprint of the base internal map.

In addition, the waterfall representation is expensive to recalculate every time step. An algorithm called modified flood fill exists, though I chose to go a different route in my explorations. Reducing the recalculations in the waterfall representation would allow less demanding processors to stay responsive with waterfall.

Lastly, watching the Search Waterfall explore makes clear that the goal moving part of the algorithm is effective but inefficient. Reducing the exploration time the robot spends

in already explored cells should help significantly with this issue. I will focus my attention on this issue, as it is the most relevant to the simulated micromouse and may lead to successes in the other two areas.

I believe that greater efficiency could be found by grouping valid routes into main paths and encouraging the robot to explore the unknowns in each group together before moving to the next group. This should localize the explorations and eliminate the frantic crossing of the maze to check a single cell. In addition, prioritizing finding the goal over exploring routes should also improve the final algorithm performance.

I believe either the Basic Waterfall or Search Waterfall as implemented could be used as a benchmark for further exploration in this area. There is clearly still significant distance between the theoretically perfect route and the performance of these models.

References

Initial project files - Udacity
https://github.com/udacity/machine-learning/tree/master/projects/capstone/open_projects/robot_motion_planning

UK Micromouse Maze Solver Rules
https://www.bcu.ac.uk/Download/Asset/29ccf534-6a1e-4135-8109-4597abdfc2d7

USC Micromouse Project
http://robotics.usc.edu/~harsh/docs/micromouse.pdf

Invobot: Artifically intelligent
https://madan.wordpress.com/2006/07/24/micromouse-maze-solving-algorithm/

MINNI: Micromouse Incorporating Neural Network Intelligence by Jondarr Gibb and Len Hamey
http://web.science.mq.edu.au/~len/preprint/gibb.acsc97.pdf

Analysis of Micromouse Maze Solving Algorithms by David M. Willardson
http://web.cecs.pdx.edu/~edam/Reports/2001/DWillardson.pdf

Quantitative Comparison of Flood Fill and Modified Flood Fill Algorithms by George Law
http://ijcte.org/papers/738-T012.pdf