MicroMouse Maze Exploration

Brian Johnson
February 5th, 2017

This project is a variation of the pre-curated robot motion planning "Plot and Navigate a Virtual Maze" project. In this project I'm extending the pre-curated project in a number of ways:

1. Using mazes from actual MicroMouse competitions
2. Incorporating more advanced robot mouse sensors, similar to the sensors used by actual competitive MicroMouse entries
3. Incorporating more advanced robot mouse movement strategies, more like the movements used by actual competitive MicroMouse entries
4. Using physical parameters of typical entries to estimate the time required for each movement and then using those times as the costs associated with those movements

The project includes the following activities:

1. Definition of a maze representation for Python implementation
2. Creation of a maze editing tool with to simplify capture of actual mazes
3. Implementation of maze, search and path visualization in Python with Tkinter
4. Implementation of a state representation to support various movement strategies
5. Calculation of the cost of each move for each of the movement strategies
6. Research, selection and implementation of path planning algorithms
7. Comparison of results between different movement strategies, algorithms and mazes
8. Comparison of the simulated results with actual MicroMouse competition results

Domain Background

The MicroMouse competition is a contest where the goal is for an autonomous robotic mouse to reach the goal of a maze as quickly as possible. IEEE held the first MicroMouse competition in NYC in 1977. Today there are 100's of MicroMouse competitions around the world every year.

This problem is especially interesting to me since I signed up for the first competition in 1977. I designed and constructed and entry but was unable to complete the coding before the date of the competition. I did go to the competition in NYC and saw that

only a handful of the thousands of people that signed up had a completed entry. Only 2 of those early entries actually completed the maze.

The Competition

The mouse starts in one corner of the maze with very limited a priori information about the maze which is described below. It must explore the maze by moving around autonomously and record the information it discovers about the maze and eventually reach the goal. Once it's found the goal it then returns back to its starting location where it can try another run. I've limited the number of runs to 3 but some competitions allow 4 or 5 or even unlimited runs within a finite time limit.

The competition typically involves more than just how fast the mouse can get from the start to the goal. Each time the mouse makes a run it's score is penalized by one thirtieth of total amount of time the mouse has been in the maze prior to that run. In addition there is a 2 or 3 second penalty/bonus if the mouse has been touched/not touched by its handler since the last run. In the scoring for the 2016 Taiwan competition there is a 3 second bonus if the mouse was not touched. In the scoring for the 2016 APEC competition there is a 2 second penalty if the mouse was touched.

For example, in the case of the 2016 Taiwan competition, if the mouse runs the maze in 60 seconds on its first run, it had spent no prior time in the maze so that run would have a score of 60 seconds. Let's assume that it returns to the start on its own (there is no 'touch') in 30 seconds. On the second run it makes it in 20 seconds. The score for the second run would be 20 seconds = 20 + (60+30)/30 – 3.

Multiple runs are typically made, unless the mouse is damaged in a run, and the best score (lowest time) is used as that entrant's score. The mouse with the best score wins.

The Maze

The mazes consists of a rectangular grid of 16 x 16 cells. Posts are located at each of the corners of every cell and each post will have at least 1 wall connected to it. There is a wall around the outer edge of the maze.

The starting cell is at an outer corner of the maze with 3 walls. The goal is any cell in the group of 2 x 2 cells located at the center of the maze. The goal has no interior walls and typically only one entrance.

The maze is designed so that a right/left hand wall following algorithm won't be able to find the goal.

Finding the Fastest Path

In addition to designing and building an autonomous mouse that can explore the maze on its own, it must be fast and smart enough to find and take the fastest path through the maze. The actual time through the maze is a function of both the physical capabilities of the mouse (maximum acceleration, maximum velocity, etc.) and finding the path that makes the best use of those capabilities. For example some mice might move only North, South, East and West and have to come to a complete stop for each change of direction. Other mice might be able to take corners without stopping while still others may be able to travel straight down diagonals at very high speeds. The optimal path for each of those mice would likely be different than the optimal paths for the others.

Datasets and Inputs

For this project there are not datasets per se, I'm using mazes from actual MicroMouse competitions. A lot of this data is publicly available online at various MicroMouse organizer and enthusiast web sites.

In particular, Micromouse USA has a lot of good information and links.

Project Details

Overview

The project will be implemented in Python using Tkinter for visualization of the mazes and paths. I used Excel to create a maze editing tool to facilitate entry and visual verification of mazes from actual competitions.

The Python implementation allows the selection of one of a number of mazes, selection of a movement strategy and selection of a search algorithm. It will then simulate the mouse running autonomously in the maze for three runs using the selected maze, motion strategy and search algorithm.

1. Maze representation

I actually defined two different representations, one to represent the maze and one to represent the mouse's state space within the maze. The maze representation is efficient to represent the maze, while the state space representation is more suitable for the movement strategies, sensor methods and algorithms. In addition the state space representation must be efficiently implemented for large state spaces as I expect to see up to 24,000 nodes in the state space for more complex movement strategies. The

underlying maze representation is hidden by routine calls so that the mouse will only know what it is capable of sensing given its current state.

The maze is represented as by a simple rectangular array where each element represents an open cell, a wall or a post. You can see an illustration of this in the Maze editing tool section below.

The mouse's state space is represented by a cell location (cy, cx), a position within the cell, a direction and a speed. For simpler movement strategies some of these state space values are always zero. In this project report I've used the term 'state' and 'node' interchangeably. When thinking of the maze and possible mouse states it seems natural to consider it as a state space. When running the search algorithms it seems appropriate to consider the states as nodes in a tree.

2. Maze editing tool

In order to speed up the creation of several actual MicroMouse mazes I implemented a tool in Excel to enable capture and visualization of the mazes. In addition it creates formatted maze data in a form for ready inclusion into the Python code.

Figure 1 below shows the maze for the 2016 APEC MicroMouse competition. In the upper left corner I've entered the maze data. The upper right corner has the data formatted as lists of lists for inclusion into Python. The lower left is an image of the actual maze that I pasted in to help with data entry and for visual verification. The bottom right corner is just empty cells, except for the start and goal cells that are conditionally formatted by Excel to better illustrate the data I entered and aid in visual verification.

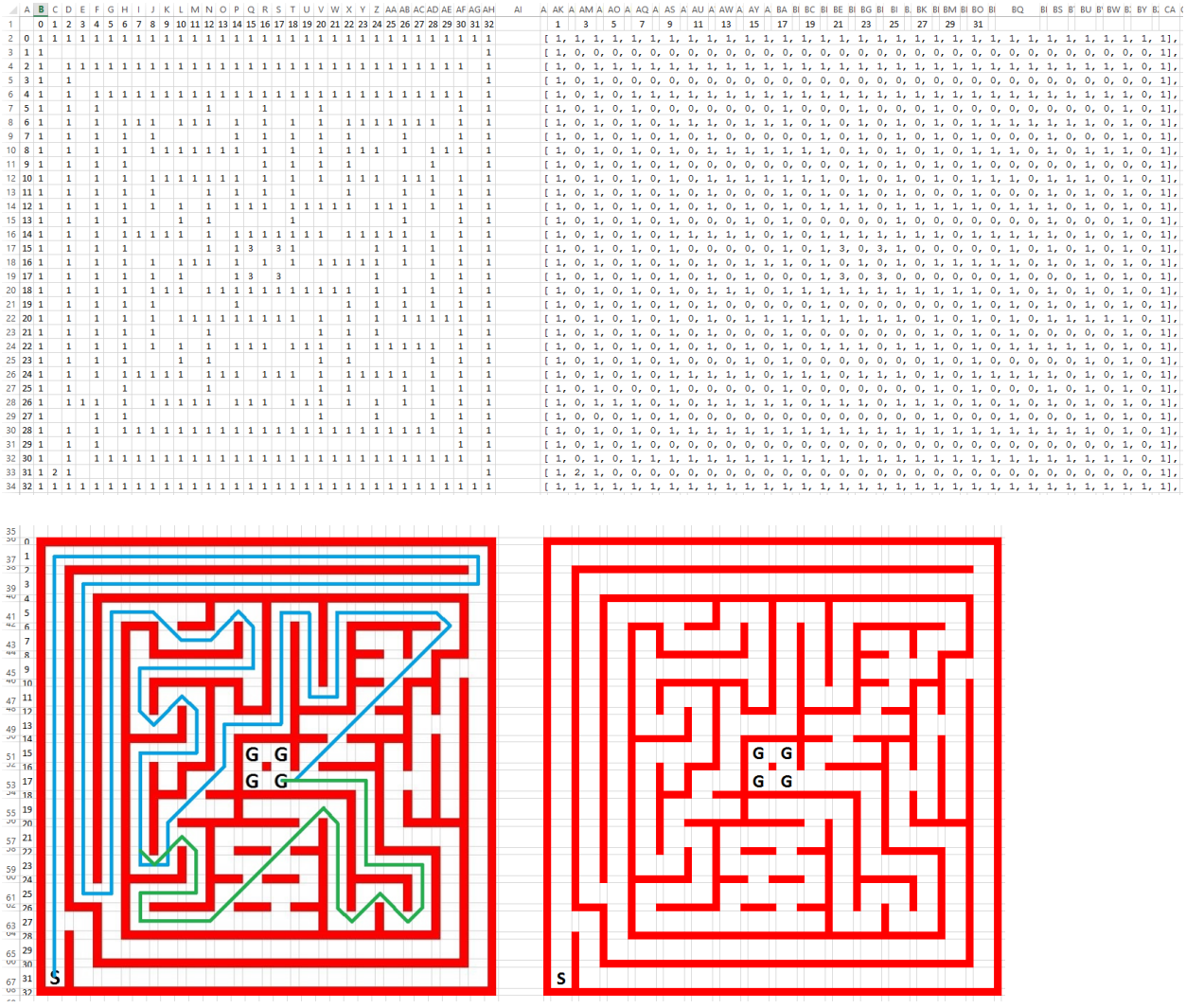

Figure 1 2016 APEC MicroMouse Maze in Excel

3. Maze, search and path visualization

Once the mazes have been entered it's desirable to visualize them to insure their correctness as well as to provide a background for the path visualization. The screen capture below shows the full maze on the left and the mouse's maze understanding on the right. The mouse has just begun exploring and is not aware of most of the walls, only the ones that its sensors have detected so far. The mouse is shown in green on both mazes and the nodes that the mouse has visited are shown as green arrows. There are also small red arrows that show the potential mouse nodes as the search algorithm is looking for the most efficient path from the mouse's current state to one of the goal states.

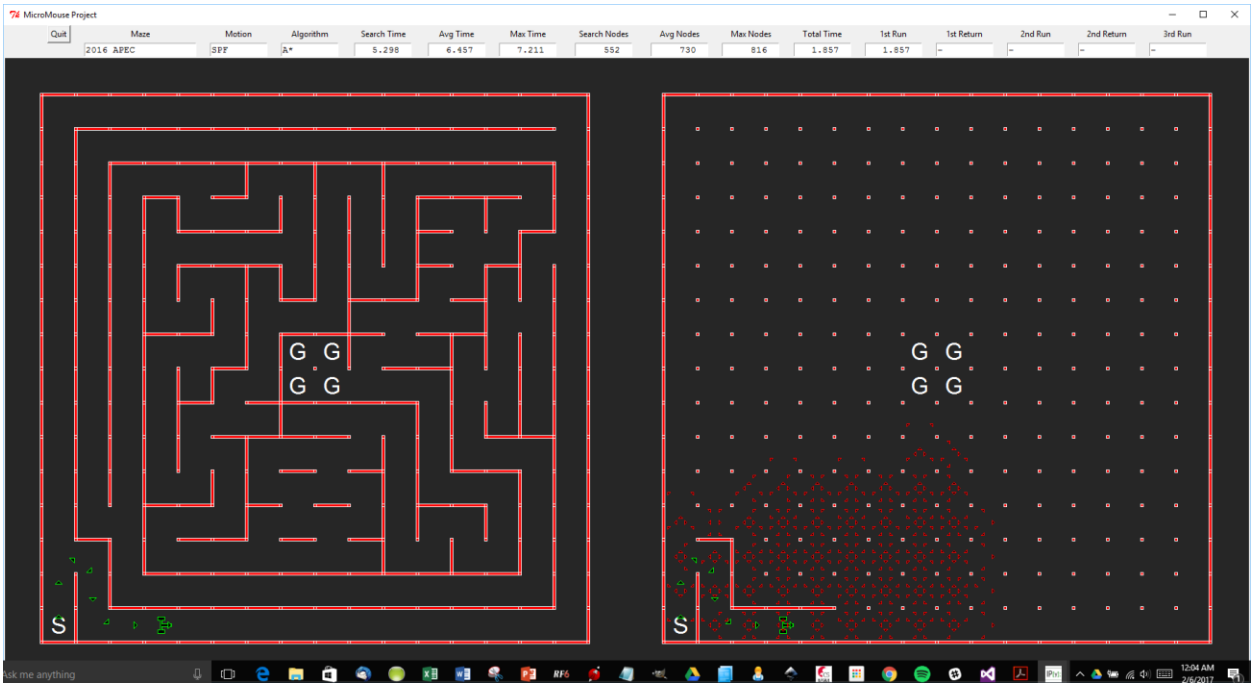

Figure 2. Screen capture of mouse exploring the maze

4. Representations for movement strategies

I investigated 4 different movement strategies:

1. Manhattan movement with Rotation and Fixed speed, MRF
2. Manhattan movement with Rotation and Variable speed, MRV
3. Smooth Paths with rotation and Fixed speed, SPF
4. Smooth Paths with rotation and Variable speeds, SPV

Manhattan movement with Rotation and Fixed speed, MRF

In this case the mouse can only move forward and must rotate toward the desired direction before moving. The possible moves are Move forward, Rotate +90, Rotate -90 and Rotate 180. Each of these actions have different costs.

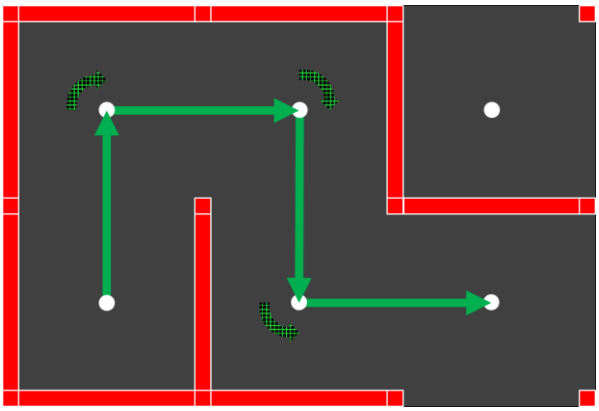

Figure 3 Manhattan movement with rotation

The state (node) representation consists of the Cell coordinates (up to 256 for a 16x16 maze) and the orientation of the mouse (4). In terms of complexity this movement strategy has 1,024 nodes and 4 actions (move forward, rotate 90, rotate 180, rotate -90).

Some considerations of this movement strategy are that it requires the ability to rotate in place and that it starts and stops for each movement and is therefore slow through the maze.

Manhattan movement with Rotation and Variable speed, MRV

This is similar to the Manhattan movement with Rotation and Fixed speed describe above, but now we allow the mouse to move continuously without stopping at each cell. This allows the mouse to move more quickly along straight lines through the maze. At each node the mouse may accelerate, continue at its current speed, decelerate or rotate if it is already stopped.

The state (node) representation consists of the Cell coordinates (up to 256 for a 16x16 maze), the orientation of the mouse (4) and a number of speeds. I limited the number of speeds to 3 so in terms of complexity this movement strategy has 3,072 states and 6 actions (accelerate forward, move forward with the same speed, decelerate while moving forward, rotate 90, rotate 180, rotate -90). Not all actions are possible at every state. For example the mouse cannot rotate while its speed is greater than zero.

Smooth Paths with Fixed speed, SPF

In order to improve our time over Manhattan movements, consider smooth movements where we can turn as part of our movement without requiring a separate rotation action while stopped. One approach is to define start and end points at different positions and orientations within each cell to allow us to predefine a finite number of paths from cell to cell that can be traveled without stopping when we change direction.

Select the positions to allow:

1. Smooth, continuous velocity paths from one cell to the next
2. Straight paths along diagonally connected cells

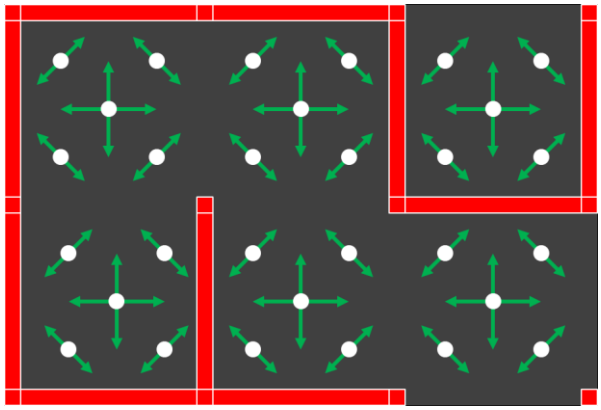

Figure 4 Start/end positions for smooth motion

Actions include: Move forward, Bear left and Bear right. Note: additional actions required for exploring to avoid getting stuck in a dead end: Rotate +90, Rotate -90, Rotate 180. Each of these have different costs.

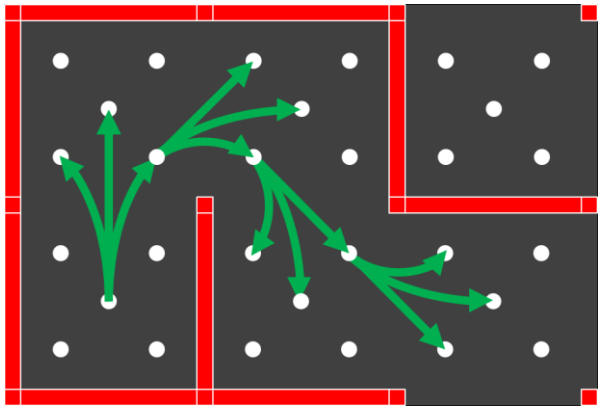

Figure 5 Smooth movements with predefined paths

In this movement strategy we still start and stop in each cell but usually don't have to take an extra rotation action to turn.

The state (node) representation consists of the Cell coordinates (up to 256 for a 16x16 maze), a position within the cell (5) and a direction (8). Since not all combinations of position and direction are possible there are only 12 combinations of position and direction (1*4 + 4*2). In terms of complexity this movement strategy has 3,072 nodes and 6 actions (move bearing left, move in middle, move bearing right, rotate 90, rotate 180, rotate -90). Again, not all actions are possible for all states.

This is becoming a competitive solution!

Smooth Paths with Variable speeds, SPV

A major limitation of the smooth paths above is that we don't get to take advantage of long straights in the maze where we could go much faster without stopping. With this movement strategy we'll keep the same smooth paths but allow them to be run a different speeds without stopping in every cell.

The state (node) representation consists of the Cell coordinates (up to 256 for a 16x16 maze), the position and direction as before (12) and the speed (3). In terms of complexity this movement strategy has 9,216 potential states (nodes) and 12 actions. It has a very complex adjacency model if we used typical tree structures.

This movement strategy has the following actions:

Move forward: decelerate, same start/end speed, accelerate
Bear left: decelerate, same start/end speed, accelerate
Bear right: decelerate, same start/end speed, accelerate
Rotate +90, Rotate -90, Rotate 180

Each of these actions will have different costs.

This is a very competitive solution!

5. Cost of moves

Each of the movement strategies outlined above have multiple possible actions (moves) between states. Each of these moves will typically have a different cost (time) associated with it which also depends on the state. I've calculated the time required for each move of each movement strategy using maximum accelerations and velocities based on what I've found about actual winning mice.

Some of the fastest mice have accelerations of over 1g! Most high performance cars are unable to reach 1 g acceleration due to limited traction. These amazing mice achieve this acceleration by running a small fan to suck the air out from under the mouse so that the atmospheric pressure above creates a significant down force that allows the mouse to get much better traction. I've limited the acceleration of the simulated mouse to 0.5g.

6. Research, select and implement search algorithms

Research

There are a large number of search algorithms that may be appropriate for finding the fastest path through an environment or maze. Mathematically the environment or maze is often represented by tree structure. The search algorithms that I investigated for inclusion in this project include:

1. Dijkstra
2. A*
3. LPA*
4. D* Lite
5. Reinforcement Learning

Dijkstra is a breadth first search of a tree with positively weighted edge costs. This is proven to find the most efficient path from a given initial node to a goal node. In this project I implement a variation that stops once it finds the minimum cost path to the goal. Some implementations expand to cover the entire state space.

A* is an enhancement to Dijkstra's algorithm in that it uses a heuristic estimate of the cost from a node to the goal node. This heuristic causes A* to initially pursue nodes that are more likely to reach the goal with the lowest cost. If the heuristic is chosen so that it is always less than the actual cost, it is also guaranteed to find the most efficient path.

Lifelong Planning A*, or LPA*, improves on A* where it addresses the fact that the maze or tree may change and allows incremental replanning without starting from scratch as A* would have to.

D* Lite, Dynamic A* Lite, is actually more closely related to LPA* than to D*. It builds on the LPA*'s ability to efficiently recalculate optimal paths as the environment changes.

Reinforcement Learning is an algorithm to learn the optimal action for any given state. This is directly applicable to solving the maze although it is more often used to model environments where there is a probabilistic outcome associated with a given action from a given state.

Selection

I was only able to implement Dijkstra and A* in Python for this project due to time limitations. Of the other search algorithms I was most interested to try D* Lite as I believe it would be critical to have an efficient replanning algorithm as we detect incremental changes in the environment (discovering previously unknown walls).

I don't think Reinforcement Learning would be very efficient for this task as I think it would take many thousands of iterations for it to learn the best path through a state space with thousands of nodes. I would then have to relearn every few moves as unknown walls are discovered. I would be interested in seeing the results but I don't think it's a good solution and I don't have the time to implement it.

Implementation

Both Dijkstra and A* were implemented as Python routines using a priority queue to store a prioritized list of states to be expanded (frontier). The prioritization is used to expand states from the frontier that have the lowest cost. In addition there is a list of the cost of each state so we can determine if we found a lower cost path to that state and a list of the parent of each state so we can trace the optimal path back to the start once we've discovered the goal.

I added search visualization and some statistics to the search algorithms to get a better understanding of how and how well they are working. The visualization adds a marker for each state the algorithm is expanding so we can watch the algorithm search. The statistics track how many nodes have been expanded and how much real time has elapsed during the search. At the end of each search I update the number of searches, the total number of expanded nodes, the maximum expanded nodes, the total search time and the maximum search time of the simulation. The real time and nodes are updated continuously during the search and the average expanded nodes, maximum expanded nodes, average time and maximum time are updated at the end of each search.

7. Results of different movement strategies, algorithms and mazes

Movement strategies

I was only able to implement three of the proposed movement strategies described above: Manhattan with Rotation Fixed speed (MRF), Manhattan with Rotation Variable speed (MRV) and Smooth Path with Fixed speed (SPF).

Since MRF stops at each node and has to take time for each rotation when required it was the slowest of the three. The MRV movement strategy improved the time by allowing the mouse to move from cell to cell without stopping when if didn't have to

stop to make a turn. The SPF strategy was the fastest even though it stopped at each cell since it typically didn't have to take time to do a separate rotation action to make a turn.

I would expect that the Smooth Path with Variable speed (SPV) would be the fastest but I was unable to implement it in the time available.

Algorithms

I was only able to implement Dijkstra and A* in the time available. One of the things I discovered was that although A* typically results in a speed up of the search over Dijkstra it didn't make much difference in the maze environment especially after most of the walls were known. I believe this is because mazes are intentionally designed to be complex and the simple heuristic of Manhattan distance to the goal in A* was not a very accurate estimate of the remaining cost to reach the goal. In other more open search environments I believe it could substantially speed up the search.

One of the results that surprised me was how long it took to run the algorithms. I know I'm not an experienced Python programmer but the algorithm implementation looked pretty clean. I used Python's priority queue to speed up ordering the list of 'frontier' nodes to expand the one with the lowest cost. I put a timer around the search function to see how long it would take and was surprised to see that it took about 7 to 9 microseconds to expand each node during a search. With the simpler move strategy there weren't as many nodes to expand so the search was quicker. With the most complex strategy I implemented, SPF, there could be over 2000 nodes expanded during a search so a single search took as much as 19 seconds in some cases. This is when running on a decent PC. I hate to imagine running those searches on a smaller, much less powerful processor in a mouse!

In my first implementation of the mouse I ran the search after each action to see what the best path to the goal (or start) was in case the path changed due to the discovery of an unknown wall. As this was so slow, I optimized the implementation by saving the previous path and only running a new search if a wall was discovered. This happened too frequently so it wasn't much of an improvement. I then modified it to run only when a newly discovered wall blocked my previously planned path. This helps speed up my simulation quite a bit by not searching again if I don't need to but it is not optimal for a real time implementation. In a real time implementation the worst case computing time is even longer since I am now doing an additional check each move and may still have to do the complete search.

A solution to this would be to use an algorithm that maintains information created during the search and only updates what is required due to a change in the

environment. There are several algorithms that support this, the most popular one seems to be D* Lite. It would be interesting to implement this to see how it improves the search time after the first initial search. In this case the search should start at the goal and search toward the mouse so that if the mouse discovers a new wall nearby it only impacts the previous information at the leaves of the tree nearest the mouse.

Another improvement would be to see if there is only one possible action and take that action. This would eliminate searching all the way to the goal when there's only one possible action. I wish I had thought of that sooner and implemented it!

Search algorithm performance

The table below shows the average search time, maximum search time, average nodes expanded and maximum nodes expanded. The number of nodes in the state space and the maximum number of actions are included for reference. The average time / average nodes and the maximum time / maximum nodes are also included for reference. The times are actual time that it took my laptop to run the searches.

It is interesting to note that the average time divided by the average nodes expanded is always in the range of 6.5 to 8.2 milliseconds per node expanded.

The average time for a search is always better with A* than Dijkstra and this makes since as A* uses a heuristic to direct its search more efficiently. In half the cases though, the maximum time for A* is worse than for Dijkstra. I believe this is because the heuristic of Manhattan distance which I used is a poor estimate of the remaining time to the goal for complex mazes like these.

Similarly the average nodes expanded for a search is always better with A* than Dijkstra. Surprisingly though, A* also did better in than Dijkstra in every cases for the maximum nodes expanded.

One interesting result is that the 2016 Taiwan maze was substantially more complex for the SPF strategy than the other two strategies. This is surprising given that the SPF strategy has the same size state space and number of moves as the MRF strategy. Perhaps it is because on the average more of the actions were possible in this maze.

Maze	Strategy	Algorithm	Avg time (sec)	Max time (sec)	Avg nodes	Max nodes	Nodes in state space	Max Actions	Avg time / Avg nodes (msec)	Max time / Max nodes (msec)
2016 APEC	MRF	Dijkstra	2.502	8.885	386	1,021	1,024	4	6.482	8.702
2016 APEC	MRF	A*	2.315	8.052	345	1,019	1,024	4	6.710	7.902
2016 APEC	MRV	Dijkstra	4.754	15.446	658	1,872	3,072	6	7.225	8.251
2016 APEC	MRV	A*	3.488	16.481	482	1,864	3,072	6	7.237	8.842
2016 APEC	SPF	Dijkstra	5.770	15.588	827	2,081	3,072	6	6.977	7.491
2016 APEC	SPF	A*	4.675	15.292	702	2,063	3,072	6	6.660	7.413
2016 Taiwan	MRF	Dijkstra	2.559	8.333	382	1,021	1,024	4	6.699	8.162
2016 Taiwan	MRF	A*	2.265	8.729	341	1,021	1,024	4	6.642	8.549
2016 Taiwan	MRV	Dijkstra	4.378	13.947	591	1,809	3,072	6	7.408	7.710
2016 Taiwan	MRV	A*	3.612	15.835	440	1,805	3,072	6	8.209	8.773
2016 Taiwan	SPF	Dijkstra	9.042	19.857	1,272	2,375	3,072	6	7.108	8.361
2016 Taiwan	SPF	A*	7.760	19.130	1,098	2,311	3,072	6	7.067	8.278

Figure 6. Search algorithm performance

Mazes

I focused on the two mazes that had the most information available, the 2016 APEC maze and 2016 Taiwan maze. There wasn't too much of a difference between the two mazes. The 2016 APEC maze was apparently simpler than the 2016 Taiwan maze and every version of the movement strategies performed better in the 2016 APEC maze.

In each case MRF was the slowest and MRV was faster. It was interesting to see that SPF was the fastest in the 2016 Taiwan maze but virtually tied with MRV in the 2016 APEC maze. I think this is because the 2016 APEC maze had more long straightaways that the MRV's ability to move in a straight line without stopping could take advantage of and fewer turns that penalized it.

8. Results compared to actual MicroMouse competition results

2016 APEC Micromouse competition

Here is the 2016 APEC maze. It's interesting to note that there are only two walls that the mouse didn't detect during its runs. Those were not along an optimal path so the mouse never explored near enough to discover them.

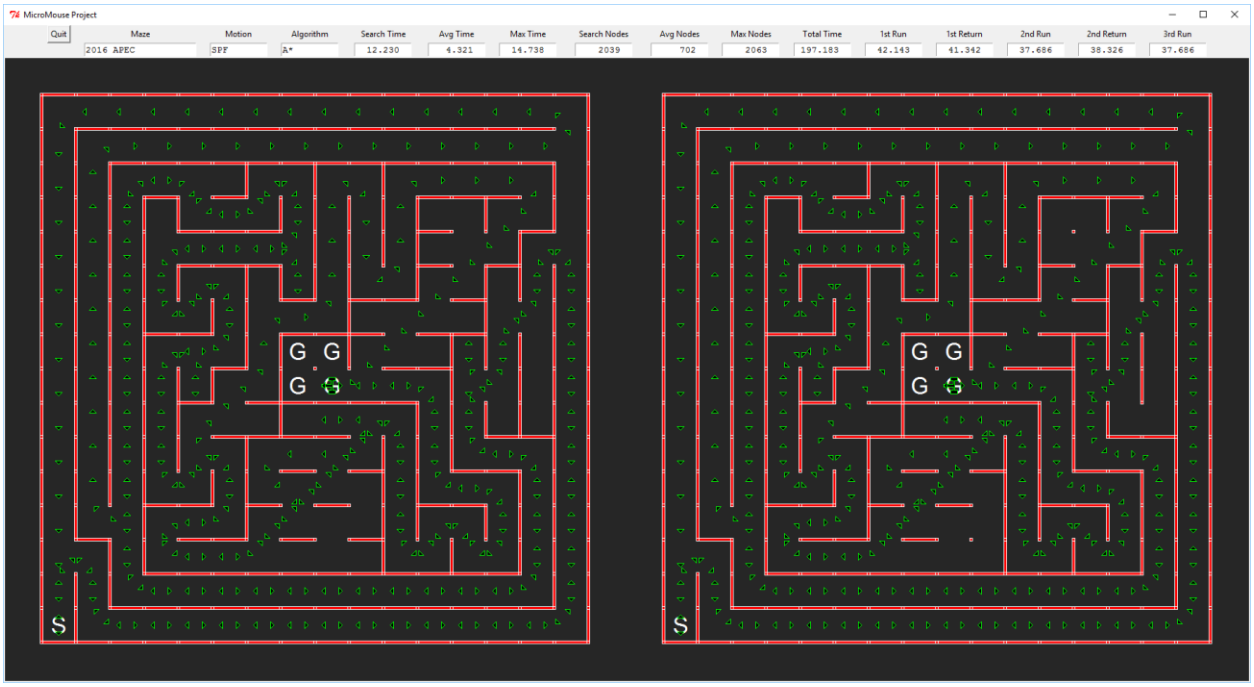

Figure 7. 2016 APEC maze with SPF paths

Below is a summary of the actual 2016 APEC competition results. The last 6 entries are mouse runs added from the simulation with variations of the movement strategies and search algorithms. In addition to adding a penalty of one thirtieth of the prior maze time the 2016 APC competition has a 2 second penalty if touched.

The first column is the entry name, next the runtime of the 1st run, then the maze time prior to the first run, which is always zero and the score of the 1st run. The first run never has any penalty. The next columns are the 2nd run time, the maze time prior to the 2nd run, whether the mouse was touched, typically to return it to the start and the score of the 2nd run. The next four columns are for 3rd run then there is the best score for that entry from all of its runs. The next column shows the rank of each entry assuming six versions of simulated mice were in the competition. The final column is the actual entry's actual ranking.

As you can see the movement strategies between the simulated mice makes a big difference in their performance but the search algorithms don't. This is to be expected

because both Dijkstra and A* should find the optimal path as long as we use an appropriate heuristic for A*.

The best score of the MRF mouse was 62.337 seconds from its second run. Typically the first run is poor, 87.761 seconds, as the mouse is learning the maze. The second run is often the best as it has a good understanding of the maze at this point. The third run is worse, 66.382 seconds, as now the score is penalized by more prior maze time.

With the MRV movement strategy the best score improved to 39.031 seconds on its second run. Again, the first and third runs were not as good, 58.621 and 40.575 seconds respectively.

With the SPF strategy the best score improved to 37.469 seconds on its second run. Again the Again, the first and third runs were not as good, 42.143 and 40.003 seconds respectively.

APEC 2016	Runtime 1	Mazetime 1	Score 1	Runtime 2	Mazetime 2	Touch	Score 2	Runtime 3	Mazetime 3	Touch	Score 3	Best Score	Rank	Act Rank
Zeetah VI	53.102	-	53.102	x	x	Y	x	x	x	Y	x	53.102	14	10
Hippo C	48.125	-	48.125	9.346	81.574	N	12.065	8.541	149.680	Y	15.530	12.065	2	2
PicOne Turbo	73.501	-	73.501	39.006	110.916	Y	44.703	x	x	Y	x	44.703	13	9
Long V2.0	40.016	-	40.016	x	x	Y	x	x	x	Y	x	40.016	12	8
Anonomouse	x	x	x	x	x	Y	x	x	x	Y	x	x	x	x
ElfMouse	x	x	x	x	x	Y	x	x	x	Y	x	x	x	x
Fab 1	x	x	x	43.948	94.157	Y	49.087	22.041	174.964	Y	29.873	29.873	7	7
Lightning McQueen V4.0	35.172	-	35.172	16.984	56.402	N	18.864	16.364	128.629	N	20.652	18.864	6	6
Green Giant 5.1V	53.394	-	53.394	9.442	86.782	N	12.335	x	x	Y	x	12.335	3	3
Excel-9	49.316	-	49.316	x	x	Y	x	11.652	136.611	Y	18.206	18.206	5	5
Decimus 4E	41.483	-	41.483	x	x	Y	x	11.487	115.422	Y	17.334	17.334	4	4
Que	60.809	-	60.809	x	x	Y	x	x	x	Y	x	60.809	15	11
Diu-Gow 4	29.402	-	29.402	9.556	60.835	N	11.584	8.902	110.830	Y	14.596	11.584	1	1
Udacity MRF Dijkstra	87.761	-	87.761	60.343	149.830	N	62.337	60.343	271.156	N	66.382	62.337	17	
Udacity MRF A*	87.716	-	87.716	60.343	149.785	N	62.336	60.343	271.111	N	66.380	62.336	16	
Udacity MRV Dijkstra	58.845	-	58.845	38.702	100.105	N	39.039	37.638	178.345	N	40.583	39.039	11	
Udacity MRV A*	58.621	-	58.621	38.702	99.881	N	39.031	37.638	178.121	N	40.575	39.031	10	
Udacity SPF Dijkstra	43.772	-	43.772	37.686	83.558	N	37.471	37.686	159.570	N	40.005	37.471	9	
Udacity SPF A*	42.143	-	42.143	37.686	83.485	N	37.469	37.686	159.497	N	40.003	37.469	8	

Figure 8. 2016 APEC competition results

These simulated mice don't win but they do a respectable job even though they are modeled with only 0.5 g maximum acceleration unlike the state of the art physical implementations of the actual winning mice.

I wish I had time to implement the Smooth Path with Variable speed movement strategy, I think that would have been a very competitive solution.

2016 Taiwan Micromouse competition

Here is the 2016 Taiwan maze. In this case there were five walls that the mouse didn't detect during its runs. Again, those we not along an optimal path so the mouse never explored near enough to discover them.

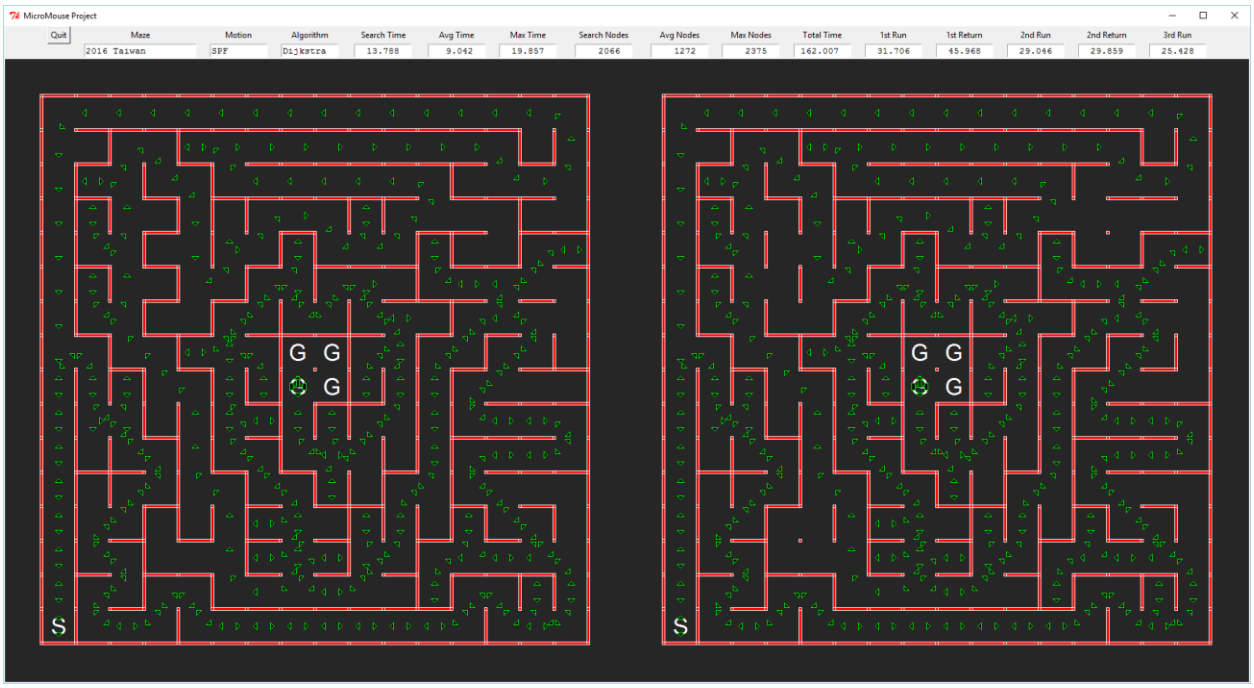

Figure 9. 2016 Taiwan Micromouse maze with SPF paths

Below is a summary of the actual 2016 Taiwan competition results. Again the last 6 entries are mouse runs added from the simulation with variations of the movement strategies and search algorithms. In addition to adding a penalty of one thirtieth of the prior maze time to the score, the 2016 Taiwan competition has a 3 second bonus if the mouse was not touched.

Overall the simulated mice had better scores in the 2016 Taiwan competition than they did in the 2016 APEC competition. Again, the movement strategies between the simulated mice makes a big difference in their performance but the search algorithms don't.

The best score of the MRF mouse was 58.596 seconds from its second run. This was slightly better than the 62.336 seconds it did in the 2016 APEC competition. With the MRV movement strategy the best score improved to 40.328 seconds on its second run which was close but not quite as good at the 39.031 second score it got in the 2016 APEC competition.

With the SPF strategy the best score improved to 26.957 seconds on its second run. This is significantly better than the 37.469 seconds it scored in the 2016 APEC competition. I think that

2016 Taiwan	Runtime 1	Mazetime 1	Score 1	Runtime 2	Mazetime 2	Touch	Score 2	Runtime 3	Mazetime 3	Touch	Score 3	Best Score	Rank	Act Rank
Wa-Zai	35.431	-	35.431	x	x	Y	x	x	x	Y	x	35.431	12	10
Lightning McQueen V4.0	68.912	-	68.912	x	x	Y	x	68.977	248.213	Y	77.251	68.912	24	18
Excel-9	33.274	-	33.274	33.141	65.005	Y	35.308	x	x	Y	x	33.274	11	9
Ki Siao	35.115	-	35.115	11.299	129.211	Y	15.606	x	x	Y	x	15.606	6	6
Hippo C2	36.084	-	36.084	x	x	Y	x	x	x	Y	x	36.084	13	11
P-Kojimouse	60.276	-	60.276	28.564	234.093	Y	36.367	x	x	Y	x	36.367	14	12
Yukikaze-5.5	38.164	-	38.164	x	x	Y	x	x	x	Y	x	38.164	17	15
Fairy	69.682	-	69.682	x	x	Y	x	x	x	Y	x	69.682	25	19
Min 7.1	36.648	-	36.648	36.603	123.020	Y	40.704	x	x	Y	x	36.648	16	14
Siden Kai	34.006	-	34.006	7.891	102.226	N	8.299	7.735	153.094	N	9.838	8.299	1	1
Ants	x	x	x	x	x	Y	x	x	x	Y	x	x	x	x
Red-Comet	31.661	-	31.661	x	x	Y	x	x	x	Y	x	31.661	10	8
MIN 7.2	38.434	-	38.434	10.726	111.115	N	11.430	9.926	163.162	N	12.365	11.430	4	4
dbHu	45.056	-	45.056	x	x	Y	x	x	x	Y	x	45.056	20	16
Greenfield++	47.349	-	47.349	47.373	96.162	Y	50.578	x	x	Y	x	47.349	21	17
Excel-8	29.916	-	29.916	10.282	86.023	N	10.149	10.051	146.147	N	11.923	10.149	3	3
Barracuda	76.745	-	76.745	76.702	142.168	Y	81.441	x	x	Y	x	76.745	26	20
Megatron	37.347	-	37.347	35.524	116.124	N	36.395	32.608	216.084	N	36.811	36.395	15	13
Decimus 4D	36.978	-	36.978	11.362	157.095	N	13.599	x	x	N	x	13.599	5	5
Zeetah VI	43.591	-	43.591	16.160	158.042	N	18.428	13.199	234.059	N	18.001	18.001	7	7
Diu-Gow 4	34.125	-	34.125	8.302	94.157	N	8.441	x	x	N	x	8.441	2	2
Udacity MRF Dijkstra	58.596	-	58.596	60.600	117.858	N	61.529	53.844	232.942	N	58.609	58.596	22	
Udacity MRF A*	58.596	-	58.596	60.600	117.858	N	61.529	53.844	232.942	N	58.609	58.596	22	
Udacity MRV Dijkstra	47.089	-	47.089	42.730	97.130	N	42.968	37.371	178.711	N	40.328	40.328	18	
Udacity MRV A*	47.089	-	47.089	42.730	97.130	N	42.968	37.371	178.711	N	40.328	40.328	18	
Udacity SPF Dijkstra	31.706	-	31.706	29.046	77.674	N	28.635	25.428	136.579	N	26.981	26.981	9	
Udacity SPF A*	31.727	-	31.727	29.046	76.975	N	28.612	25.428	135.880	N	26.957	26.957	8	

Figure 10. 2016 Taiwan competition results

Again, these simulated mice don't win the competition but they do a respectable job even though they are limited by 0.5 g maximum acceleration.

README

The project consists of 6 files:

MicroMouse.py	the main program, contains most setup and main loop
maze.py	defines a maze class object and methods
mouse.py	defines a mouse object and methods
common.py	defines a state object and methods defines a vector object and methods defines get_sensor_vectors() defines some common helper functions
graph.py	defines a graph object and methods
display.py	defines a display object and methods sets up Tkinter defines a number of display related functions

It requires Tkinter, math, numpy, time, random, copy and Queue.

The following variables may be changed in MicroMouse.py:

maze_index	selects which maze will be used, line 19
move_strategy_index	selects which move strategy will be used, line 33
algorithm_index	selects which algorithm will be used, line 39

The following variables may be changed in mouse.py:

with_markers	enables display of search markers, line 63