

# **Plot and Navigate a Virtual Maze**

## **Capstone Project**

**Machine Learning Engineer Nanodegree**

Srikanth Pagadala

Nov 3<sup>rd</sup>, 2016

Rev1: Nov 6<sup>th</sup>, 2016

# Table of Contents

## [Definition](#)

- [Project Overview](#)
- [Problem Statement](#)
- [Metrics](#)

## [Analysis](#)

- [Data Exploration](#)
- [Exploratory Visualization](#)
- [Algorithms and Techniques](#)
  - [Random Turn](#)
  - [Follow Wall](#)
  - [Block Deadend](#)
  - [Depth First Search](#)
  - [Breadth First Search](#)
  - [Uniform Cost Search](#)
  - [A\\* Search](#)
  - [Value Iteration](#)
  - [Q Learning](#)
  - [Flood Fill](#)
- [Benchmark](#)

## [Methodology](#)

- [Data Preprocessing](#)
- [Implementation](#)
  - [Project Structure](#)
  - [Files Description](#)
  - [Class Diagram](#)
  - [Sequence Diagram](#)
  - [Commands](#)
- [Refinement](#)

## [Results](#)

- [Algorithm Evaluation and Validation](#)
- [Justification](#)

## [Conclusion](#)

- [Free-Form Visualization](#)
- [Reflection](#)
- [Improvement](#)

# Definition

## Project Overview

This project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its center. The robot mouse may make multiple runs in a given maze. In the first run, the robot mouse tries to map out the maze to not only find the center, but also figure out the best paths to the center. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned. This youtube [video](#) is an example of a [Micromouse](#) competition.

## Problem Statement

In this project, we are not going to build a toy robot as seen in the video. We will rather create software functions to control a virtual robot to navigate a virtual maze. A simplified model of the world is provided along with specifications for the maze and robot; our goal is to obtain the fastest times possible in a series of test mazes.

On each maze, the robot completes two runs. In the first run, the robot freely roam the maze to build a map of the maze. It enters the goal room at some point during its exploration, but is free to continue exploring the maze after finding the goal. After entering the goal room, the robot may choose to end its exploration at any time. The robot is then moved back to the starting position and orientation for its second run. Its objective now is to go from the start position to the goal room in the fastest time possible.

Our strategy is to explore various AI techniques available for “Maze Solving” problem. There are quite a few applicable algorithms out there, but we are keeping focus on the most important ones. Each technique will be considered for its strengths and weaknesses, consistency, reliability and speed.

Various AI techniques that we have chosen to examine in this project are as follows

1. EZ
  - a. Random turn
  - b. Follow wall
  - c. Block deadend
2. Graph Search
  - a. Depth first search
  - b. Breadth first search

- c. Uniform cost search
  - d. A\* search
- 3. Reinforcement Learning
  - a. Value Iteration
  - b. Q Learning
- 4. Other
  - a. Flood fill

## Metrics

It is expected that all the chosen techniques will solve the given maze; albeit by varying degree. At the end of the project, we intend to find the best technique that can solve any unseen maze in the fastest time. For this, we need to define a single Score Metric that will help us pick the winner.

**Robot Score:** The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. It is the indicator of the cost of achieving the goal. Hence lower scores are better here. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

$$\text{Robot Score} = \text{No of steps in run-2} + (1/30) * \text{No of steps in run-1}$$

Additionally, as we will have mazes with different dimension and the steps taken depends on the maze structure or the shortest path from start to destination, we need another metric that will help us compare across varied mazes.

**Normalized Robot Score:** Normalized Robot Score is equal to 'robot score' divided by 'best possible robot score' for the given maze. If the Normalized Score is 1, we know that our solution is the best possible for that maze. Normalized score can be used to compare any other score from any other maze. While Robot Score is calculated by above formula, Best Possible Robot Score is calculated by hand for each maze.

$$\text{Normalized Robot Score} = \text{Robot Score} / \text{Best Possible Robot Score}$$

We will make use of both these metrics in our analysis to pick the winning algorithm in the Results section of this report.

# Analysis

## Data Exploration

The maze exists on an  $n \times n$  grid of squares,  $n$  even. The minimum value of  $n$  is twelve, the maximum sixteen. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are walls that block all movement. The robot starts in the square in the bottom-left corner of the grid, facing upwards. The starting cell always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the grid is the goal room consisting of a  $2 \times 2$  square; the robot must make it here from its starting cell in order to register a successful run of the maze.

Mazes are provided to the system via text file, as shown in Fig 1 below:

```
12
1,5,7,5,5,5,7,5,7,5,5,6
3,5,14,3,7,5,15,4,9,5,7,12
11,6,10,10,9,7,13,6,3,5,13,4
10,9,13,12,3,13,5,12,9,5,7,6
9,5,6,3,15,5,5,7,7,4,10,10
3,5,15,14,10,3,6,10,11,6,10,10
9,7,12,11,12,9,14,9,14,11,13,14
3,13,5,12,2,3,13,6,9,14,3,14
11,4,1,7,15,13,7,13,6,9,14,10
11,5,6,10,9,7,13,5,15,7,14,8
11,5,12,10,2,9,5,6,10,8,9,6
9,5,5,13,13,5,5,12,9,5,5,12
```

Fig 1: Maze Specification - test\_maze\_01.txt

On the first line of the text file is a number describing the number of squares on each dimension of the maze  $n$ . On the following  $n$  lines, there are  $n$  comma delimited numbers describing which edges of the square are open to movement. Each number represents a four bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom ( $0*1 + 1*2 + 0*4 + 1*8 = 10$ ). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom left) corner of the maze. Fig 2 below is showing various cells with corresponding wall specification.

2	12	7	14
6	15	9	5
1	3	10	11

Fig 2: Wall specification values of cells

'showmaze.py' script can be used to create a visual demonstration of what a maze looks like, using command 'python showmaze.py [test\_maze\_01.txt]'. The script uses the python turtle module to visualize the maze. It reads the maze\_specification.txt file line by line, splitting each line by commas and building an internal array of cell specifications. Then it loops through the array, figures out which side of the cell is open by bitwise ANDing the cell value with 1, 2, 4 or 8 for up, right, down and left directions respectively. Lastly, it graphically renders each cell using turtle pen. Showmaze.py can be seen in action in one of the videos in the following sections of the report.

The robot can be considered to rest in the center of the square it is currently located in, and points in one of the cardinal directions of the maze. The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robot's left and right sensors will state that there are no open squares in those directions and at least one square towards its front. On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robot's turning and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.

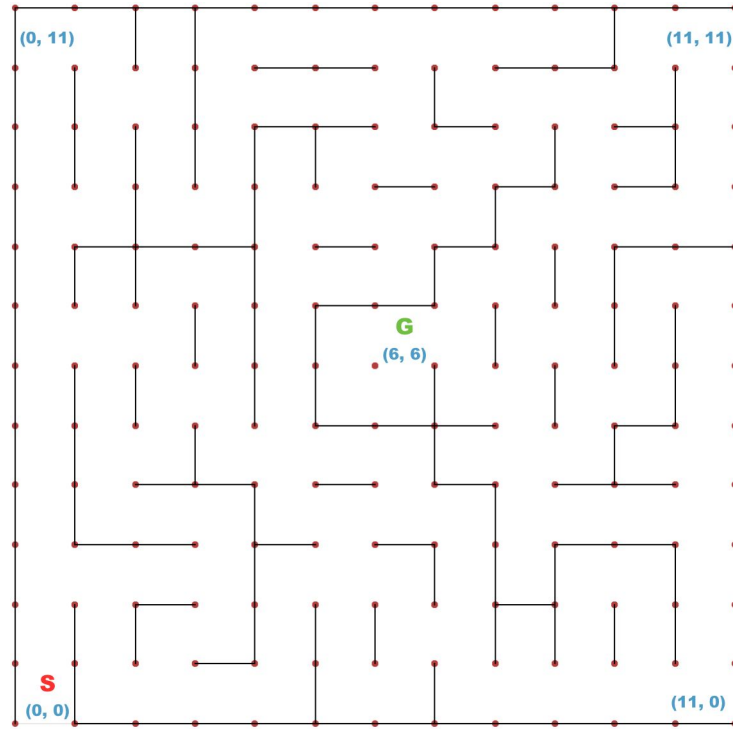


Fig 3: Maze Structure

Fig 3 above, is an example maze (test\_maze\_01). Robot's start location is at the bottom-left and goal location is in the center of the maze. Cell numbering follows cartesian coordinate system, with origin cell (0, 0) on bottom-left. Cell's x-coordinate increase from left to right; y-coordinate increase from bottom to top.

Mazes containing no loops are known as "simply connected", or "perfect" mazes. Mazes with some loops are called "braid" maze. Our maze is a braid maze, as it contains loop near (2, 1) and at other places. Braid maze are not easy to solve. As we will learn soon, some of the simple AI techniques do not play well with such maze. Braid maze usually has more than one path to the goal cell. Our objective is to find the shortest path in the least number of steps.

### Exploratory Visualization

Maze exploration can be described in following 4 phases.

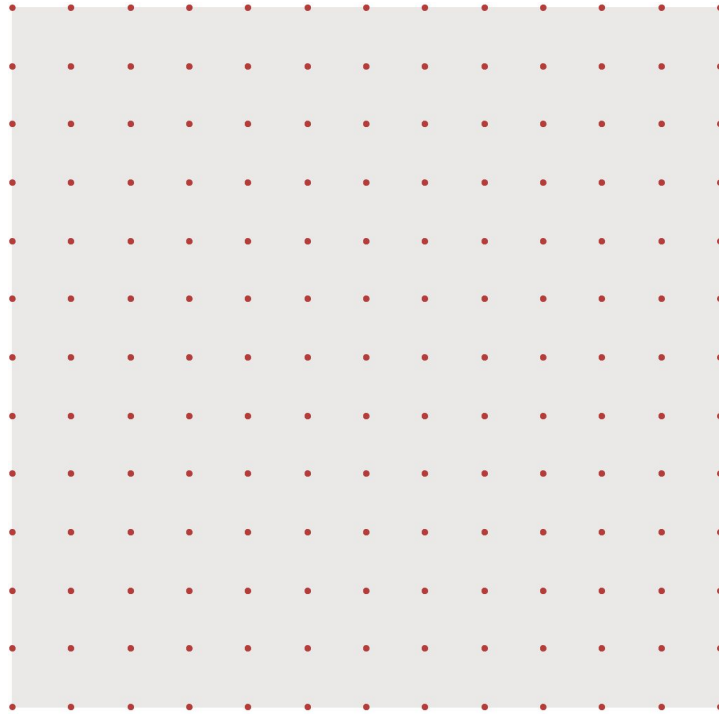


Fig 4: Maze exploration, phase 1. No walls

As shown in Fig 4, in phase 1 or start phase, robot has no knowledge of the maze. It does not know where the walls are. It just knows its start location is (0, 0) and heading Upwards.

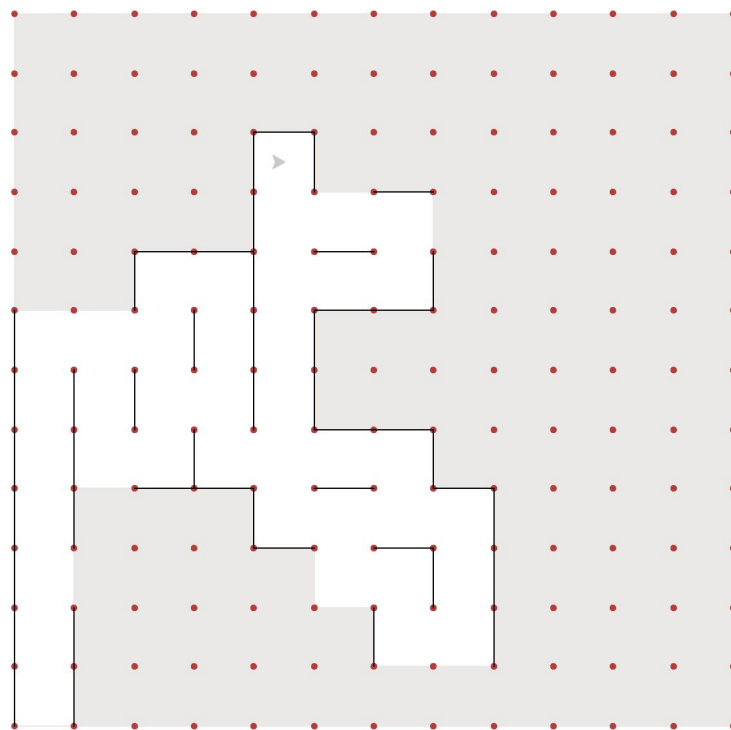


Fig 5: Maze exploration, phase 2. Discovering walls



As shown in Fig 5, in phase 2, robot starts moving around. As it take steps and sense around with its distance sensors, it discovers portions of maze walls. Thus increasing its knowledge of the maze. Often times these discovery movements are random, but can also be based on Heuristics, as we will see in coming sections.

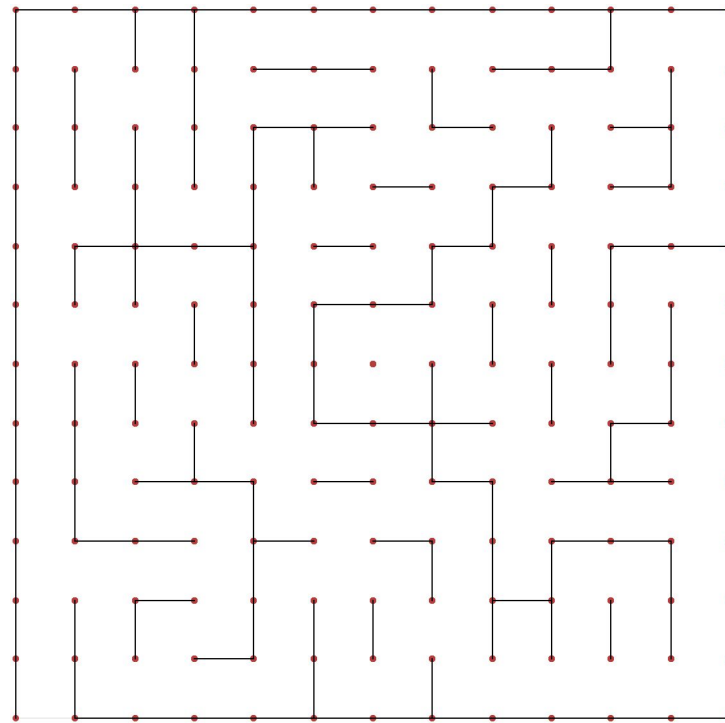


Fig 6: Maze exploration, phase 3. Maze explored

As shown in Fig 6, by phase 3, robot has explored significant portion of the maze and reached goal cell atleast once. In ideal case, robot would explore 100% of the maze, but that rarely happens and is unnecessary too. Since there is a race against time, each AI technique, based on its properties decides when it has sufficient information to solve the maze and hence stop the exploration. Phase 3, is also called end of run=1. By this time robot is expected to have gathered all the information that it needs to solve the maze. Robot sends reset signal to the environment indicating that it is ready for run=2 and solve the maze.

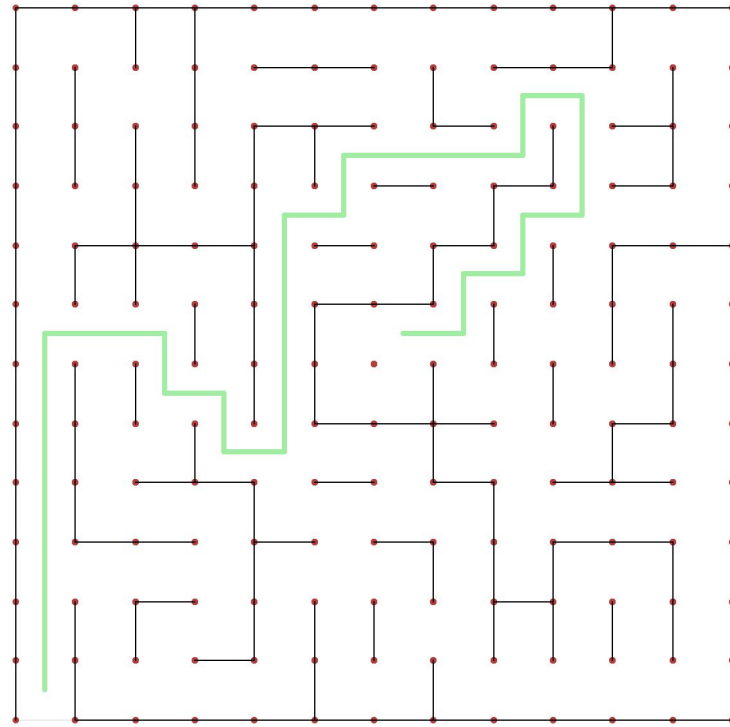


Fig 7: Maze exploitation, phase 4.

As shown in Fig 7, phase 4 is also called run=2. In this phase, robot is placed back in the start cell. From there it tries to solve the maze in minimum number of steps. Above figure is showing the optimal path to the goal cell. Under ideal conditions, robot will find that path, and it does ... :)

## Algorithms and Techniques

Various AI techniques that we are examining in this project are as follows

5. EZ
  - a. Random turn
  - b. Follow wall
  - c. Block deadend
6. Graph Search
  - a. Depth first search
  - b. Breadth first search
  - c. Uniform cost search
  - d. A\* search
7. Reinforcement Learning
  - a. Value Iteration
  - b. Q Learning
8. Other
  - a. Flood fill

Note: Below definitions of algorithms have been adapted from wikipedia. Graph illustrations have been borrowed from Berkeley AI course [CS188x](#).

## Random Turn

This is a trivial method that can be implemented by a very unintelligent robot. At each step of the way, robot randomly picks a direction to proceed, with slight bias toward 'unexplored' cells. At a dead end, it turns around. Although such a method would always eventually find a solution, this algorithm can be extremely slow. Let us see it in action.



As seen in the video, Random Turn approach is very wasteful. Robot tends to revisit already explored cells repeatedly. Occasionally, robot misses to find the goal cell in the allotted time. Robot Score is unacceptably pretty high as well. It is a very unscientific method. This algorithm is used mainly as a control, for determining if the other algorithms are better based on robot score.

## Follow Wall

The wall follower, the best-known rule for traversing mazes, is also known as either the left-hand rule or the right-hand rule. If the maze is simply connected, that is, all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze the solver is guaranteed not to get lost and will reach a different exit if there is one; otherwise, he or she will return to the entrance having traversed every corridor next to that connected section of walls at least once.



After watching the video, it is now quite obvious that Follow Wall technique will not work for our braid mazes. It works only for maze that have exit or goal cell on the outer boundary wall as shown below in Fig 8.

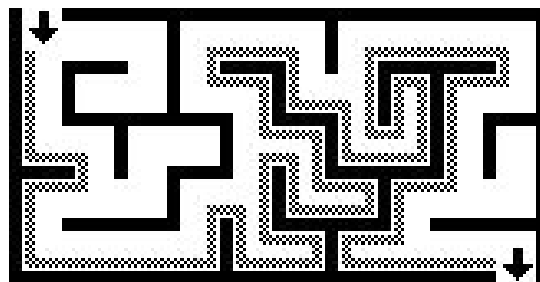


Fig 8: Maze with exit on the outer boundary wall

### Block Deadend

This algorithm is similar to Random Turn, except that any dead ends are remembered and a virtual wall is placed at the opening so that the robot does not re-enter. The big idea here is that once we start blocking dead ends, eventually only the correct ways from start to finish will remain unblocked. This design is illustrated in the following maze in Fig 9.

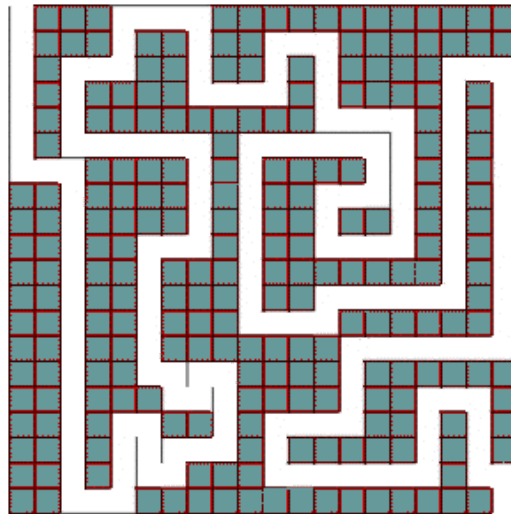


Fig 9: Maze with all dead ends blocked leaving only the correct path open

It has the advantage of simplicity, but it is anything but optimal. Furthermore, this algorithm cannot guarantee that the quickest route will be found if the maze has loops. Depending on how the algorithm is implemented, sometimes it will cause the robot to go around loops several times before actually finishing.



That brings us to the end of the what I call EZ algorithms. These were all non scientific and common sense based algorithms. They took us only so far in that results are not guaranteed. Even when robot makes it to the goal cell by chance, it comes at a steep cost.

### Depth First Search

Depth first search (DFS) is an uninformed algorithm for traversing or searching tree or graph data structures, where one starts at the root and explores as far as possible along each branch before backtracking. It uses stack to remember the next vertex to start a search.

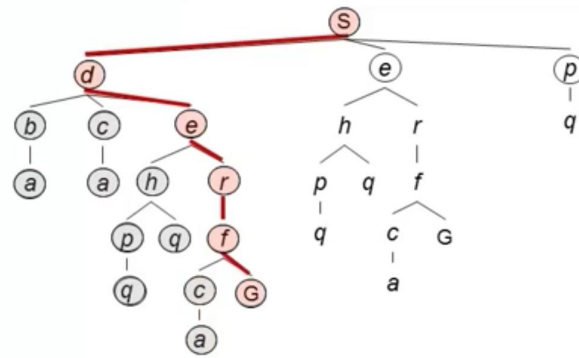


Fig 10: Depth First Search tree

This idea is illustrated in the above search tree in Fig 10. It expands nodes from left to right, digging deep in each branch. Disadvantage is that this approach could lead to processing the whole tree. If tree is 'm' tiers deep, and 'b' nodes per tier, then algorithm can potentially expand  $O(b^m)$  nodes. As far as memory used to store fringe nodes is concerned, it stores only siblings on path to root, so  $O(bm)$ . If m is infinite, then algorithm is not guaranteed to find the solution. Lastly, DFS is not guaranteed to find optimal solution either. It just finds "leftmost" solution regardless of cost or depth. Let us now see how this algorithm flares for our maze.



As seen in the video, algorithm started digging deep as soon as it found a branch. In maze world, it is not necessarily good or bad approach. It is what it is. It does find the goal consistently though.

## Breadth First Search

Breadth first search (BFS) is an uninformed algorithm for traversing or searching tree or graph data structures, where it starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors. It uses FIFO queue to remember fringe nodes.

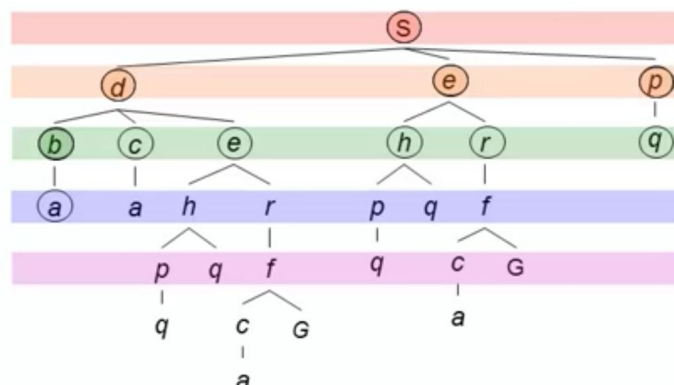


Fig 11: Breadth First Search tree

This idea is illustrated in the above search tree in Fig 11. If first shallowest solution is 's' tiers deep ( $s=5$  above), then BFS processes all nodes above shallowest solution. Search takes time of  $O(b^s)$ , where 'b' is nodes per tier. Fringe roughly have entire last tier, so memory usage is  $O(b^s)$ . BFS is guaranteed to find the solution if one exists. Usually found solution is optimal also, provided search costs are all 1. With that, let us see how BFS does for our maze.



As seen in the video, BFS expanded nodes horizontally and reached goal in the shallowest path.

## Uniform Cost Search

Uniform Cost Search is like BFS. The difference is rather than going evenly across the layers, prioritizing them by their depth, instead we prioritize them by their cost. So that cheap things get done first, even if they are multiple steps into the tree.

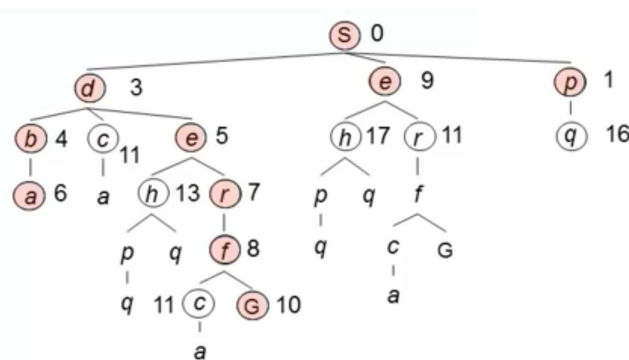


Fig 12: Uniform Cost Search tree

This idea is illustrated in the above search tree in Fig 12. So the difference is although you are moving from the top to the bottom, you are doing it in a ragged way that follows the cost contours. At any given time, we have expanded all paths up to a certain cost.

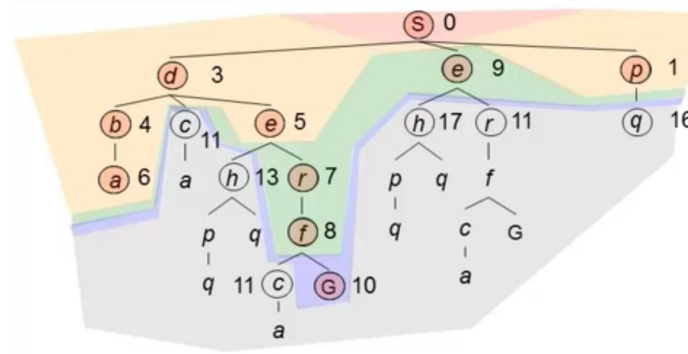


Fig 13: Uniform Cost Search contour lines

UCS processes all nodes with cost less than cheapest solution. If the solution costs  $C^*$  and arcs cost at least  $\epsilon$ , then the “effective depth” is roughly  $C^*/\epsilon$ . In worst case, time it takes to reach solution is  $O(b^{(C^*/\epsilon)})$ . Fringe roughly takes the space for nodes in the last tier, so  $O(b^{(C^*/\epsilon)})$ . Assuming best solution has a finite cost and minimum arc cost is positive, UCS is guaranteed to find the solution and an optimal one. Let us see it in action.



## A\* Search

A\* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

At each iteration of its main loop, A\* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A\* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where  $n$  is the last node on the path,  $g(n)$  is the cost of the path from the start node to  $n$ , and  $h(n)$  is a heuristic that estimates the cost of the cheapest path from  $n$  to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node. Typical implementations of A\* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. Let us watch A\* in action now.



Here we used manhattanHeuristic to find the path. euclideanHeuristic is also implemented for comparison.

## Value Iteration

Value iteration is a method of computing an optimal MDP policy and its value. Value iteration starts at the "end" and then works backward, refining an estimate of either  $Q^*$  or  $V^*$ . There is really no end, so it uses an arbitrary end point. Let  $V_k$  be the value function assuming there are  $k$  stages to go, and let  $Q_k$  be the  $Q$ -function assuming there are  $k$  stages to go. These can be defined recursively. Value iteration starts with an arbitrary function  $V_0$  and uses the following equations to get the functions for  $k+1$  stages to go from the functions for  $k$  stages to go:

$$\begin{aligned} Q_{k+1}(s,a) &= \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma V_k(s')) \text{ for } k \geq 0 \\ V_{k+1}(s) &= \max_a Q_{k+1}(s,a) \text{ for } k \geq 0 \end{aligned}$$

It can either save the  $V[S]$  array or the  $Q[S,A]$  array. Saving the  $V$  array results in less storage, but it is more difficult to determine an optimal action, and one more iteration is needed to determine which action results in the greatest value. Let us look at Value Iteration here



Here Living Reward is 0.0, Robot Noise is 0.0 and Goal State Reward is 1.0. Discount (gamma) is set to 0.9, and num of iterations ( $k$ ) is set to 100. The transition probability for a state (cell location) is based on the number of wall openings in that state.

## Q Learning

In Q-learning and related algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment. A history of an agent is a sequence of state-action-rewards:

$$\langle s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, \dots \rangle,$$

which means that the agent was in state  $s_0$  and did action  $a_0$ , which resulted in it receiving reward  $r_1$  and being in state  $s_1$ ; then it did action  $a_1$ , received reward  $r_2$ , and ended up in state  $s_2$ ; then it did action  $a_2$ , received reward  $r_3$ , and ended up in state  $s_3$ ; and so on.  $Q^*(s,a)$ , where  $a$  is an action and  $s$  is a state, is the expected value (cumulative discounted reward) of doing  $a$  in state  $s$  and then following the optimal policy. Q-learning uses temporal differences to estimate the value of  $Q^*(s,a)$ . In Q-learning, the agent maintains a table of



$Q[S,A]$ , where  $S$  is the set of states and  $A$  is the set of actions.  $Q[s,a]$  represents its current estimate of  $Q^*(s,a)$ . An experience  $\langle s,a,r,s' \rangle$  provides one data point for the value of  $Q(s,a)$ . The data point is that the agent received the future value of  $r + \gamma V(s')$ , where  $V(s') = \max_a' Q(s',a')$ ; this is the actual current reward plus the discounted estimated future value. This new data point is called a return. The agent can use the temporal difference equation to update its estimate for  $Q(s,a)$ :

$$Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$$

or, equivalently,

$$Q[s,a] \leftarrow (1-\alpha) Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'])$$



Q-learning learns an optimal policy no matter which policy the agent is actually following (i.e., which action  $a$  it selects for any state  $s$ ) as long as there is no bound on the number of times it tries an action in any state (i.e., it does not always do the same subset of actions in a state). Because it learns an optimal policy no matter which policy it is carrying out, it is called an off-policy method.

Here total num of learning episodes is set to 100. Learning rate,  $\alpha=0.5$ , Discount,  $\gamma=0.9$  and Exploration,  $\epsilon=0.3$ . Both state transition function and reward function are missing. Agent learns both by interacting with the environment during its learning episodes.

### Flood Fill

This algorithm is much more complex than the previous algorithms. It involves an initial assumption that there are no walls in the maze. A number is assigned to each cell. The number corresponds to the distance to the goal. This algorithm has the advantage of always finding the shortest path between start and finish. However, the shortest path is only in terms of distance; depending on the number of turns and the associated time to turn, the shortest path may not be the quickest. This concept is illustrated in below video



The big idea here is that we imagine maze has a convex plane, with goal at the bottom of plane and start cell at the top. Numbers on each cell is in a way indicating how far higher each cell is compared to the goal cell. We now imagine flooding the convex plane with some fluid and follow the path of the liquid downstream to the goal cell. Following the algorithm strictly will improve the average time to finish in any maze. This algorithm always works, and is not random – it is systematic and predictable.

## Benchmark

As enumerated in the previous section, there are multiple algorithms, each with different properties and unique solutions. We want to pick the best algorithm that can solve any unseen maze in the fastest time. For this, we need a single number Score that will help us pick the winner.

**Objective Criteria:** Solving maze involves two runs. Exploration run and Exploitation run. Exploration run, often involves lot of random movements. It is kind of unavoidable. Without sufficient exploration robot might miss out on the optimal path. Because of the randomness in the movements, many times robot ends up in the already explored cell, thus wasting some of its precious total allotted steps. In order to compensate of this, the weightage of run=1 in the final score must be factored down. However in Exploitation run (run=2), robot is expected to go from the start position to the goal position in the fastest time. Therefore, the weightage of run=2 in the final score must be factored 100%.

With these arguments in place, the **Robot Score** for the maze can be defined as - equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. Score is the indicator of the cost of achieving the goal. Hence lower scores are better here. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

$$\text{Robot Score} = \text{No of steps in run-2} + (1/30) * \text{No of steps in run-1}$$

Additionally, we have varied mazes, each with different dimension and structure. We need another metric that will help us compare results across mazes. **Normalized Robot Score** is equal to 'robot score' divided by 'best possible robot score' for the given maze. If the Normalized Score is 1, we know that our solution is the best possible for that maze. Normalized score can be used to compare any other score from any other maze.

$$\text{Normalized Robot Score} = \text{Robot Score} / \text{Best Possible Robot Score}$$

**Selection Threshold:** We have chosen the threshold for an acceptable Final Normalized Robot Score to be 1.200. It has built in safety buffer of 20% to account for any possible variations in the test environment and quirks in the algorithm implementation. Any score above this threshold means that algorithm is taking too many steps, potentially wasteful cell revisits, to complete the maze as compared to the optimal solution and hence must be rejected. The best possible Final Score is 1.000. Scores greater than 1 and near 1 are good scores. Scores above 1.2 are bad.

**Subjective Criteria:** Although, a mathematical score is nice - it often does not tell the full story. Due to randomness in the process, some algorithm can get lucky and score high. That is why we have to also consider subjective criteria, such as robustness, reliability, consistency and strengths of the algorithm in our final decision. Robot Score is simply a number that helps in deciding the winning algorithm. Winning algorithm will consistently complete any maze. In other words, it has to have not only a good score but also be a robust technique.

## Methodology

### Data Preprocessing

In this project, there is no data preprocessing needed. Robot gathers the data that it needs online by interacting with environment and sensing walls with its sensors. Once robot has accumulated sufficient data for its algorithm to work, it starts using it to solve the maze. Therefore there is no data preprocessing in this project.

### Implementation

All the algorithms and corresponding GUI have been implemented in python 2.7. Code is neatly commented and effectively explains complex implementations. It is best to go through the codebase, if interested in the technical implementation details. However, I would recommend watching videos is the bestest way to learn about my solutions. I talk about all the necessary and relevant details of each solution in my videos.

### Project Structure

Following fig 14 is the project structure of my implementation. When you unpack the sources bundle it will create folder "MLND Capstone" with following files.

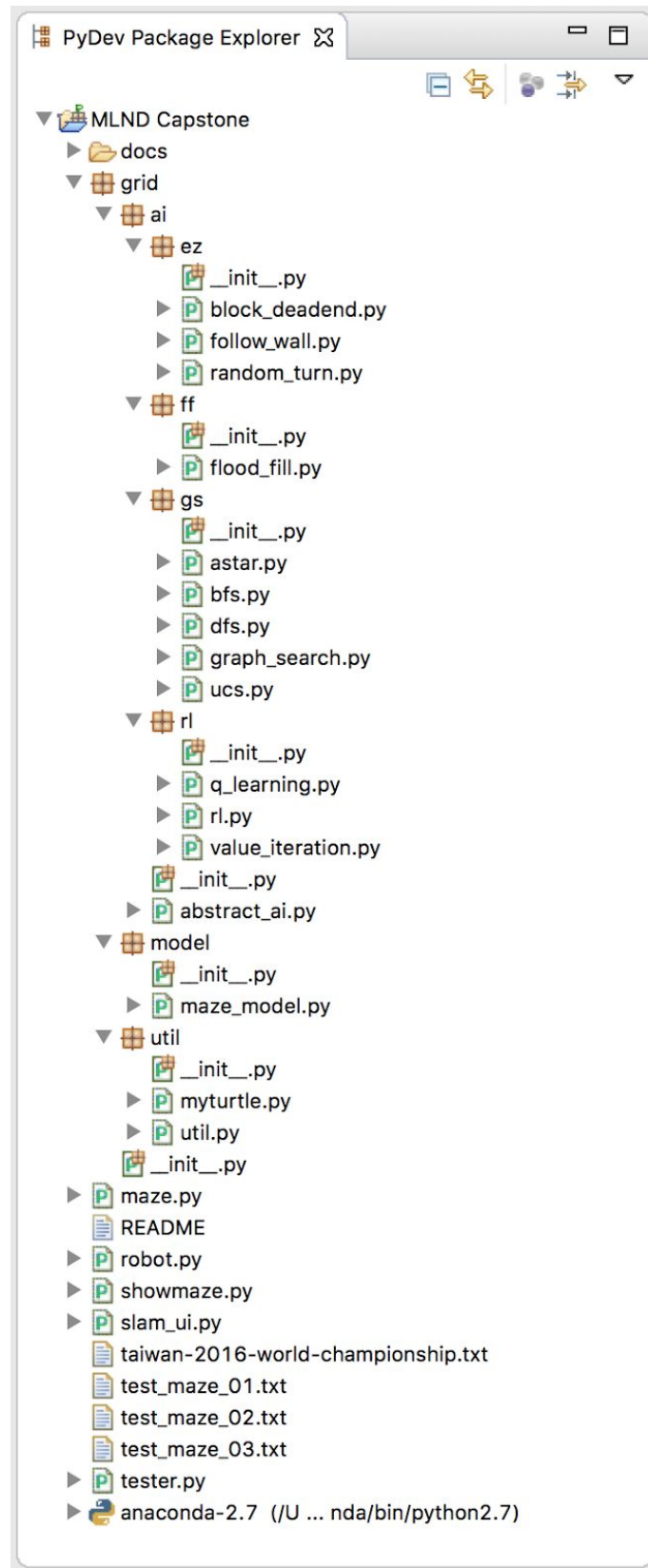


Fig 14: Project structure

## Files Description

Filename	Comments
taiwan*.txt, test_maze*.txt	These are the maze specification files. I recreated taiwan-2016-world-championship maze from <a href="#">here</a> .
tester.py, showmaze.py, maze.py	Were provided by <a href="#">AI startercode.zip</a>
robot.py	Is my implementation of awesome Virtual Robot - called "Ranger" ;)
slam_ui.py	Does all the hard work of bringing boring AI algorithms to life and make them look good. Implemented using python turtle library.
Package <u>util</u> : myturtle.py	I ran into lot of performance issues with standard turtle library. So I ended up patching it. Next time please remind me to try <a href="#">pygame</a> .
Package <u>model</u> : maze_model.py	Stores the "state" of the maze (walls, explored cell etc). Is used by all the AI routines.
Package <u>ai</u> : abstract_ai.py	Is the parent class of all the AI implementations. Carries the bulk of the logic and taps into each AI implementation for AI specific interpretation at relevant points in the flow.
Package <u>ez</u>	Contains implementation of EZ (common sense) techniques, such as, Random, Follow Wall and Block Deadend
Package <u>gs</u>	Contains implementation of Graph Search AI techniques - DFS, BFS, UCS and A*
Package <u>rl</u>	Contains implementation of Reinforcement Learning AI techniques - Value Iteration and Q Learning
Package <u>ff</u>	Contains implementation of Flood Fill technique.

Table 1: Files Description

## Class Diagram

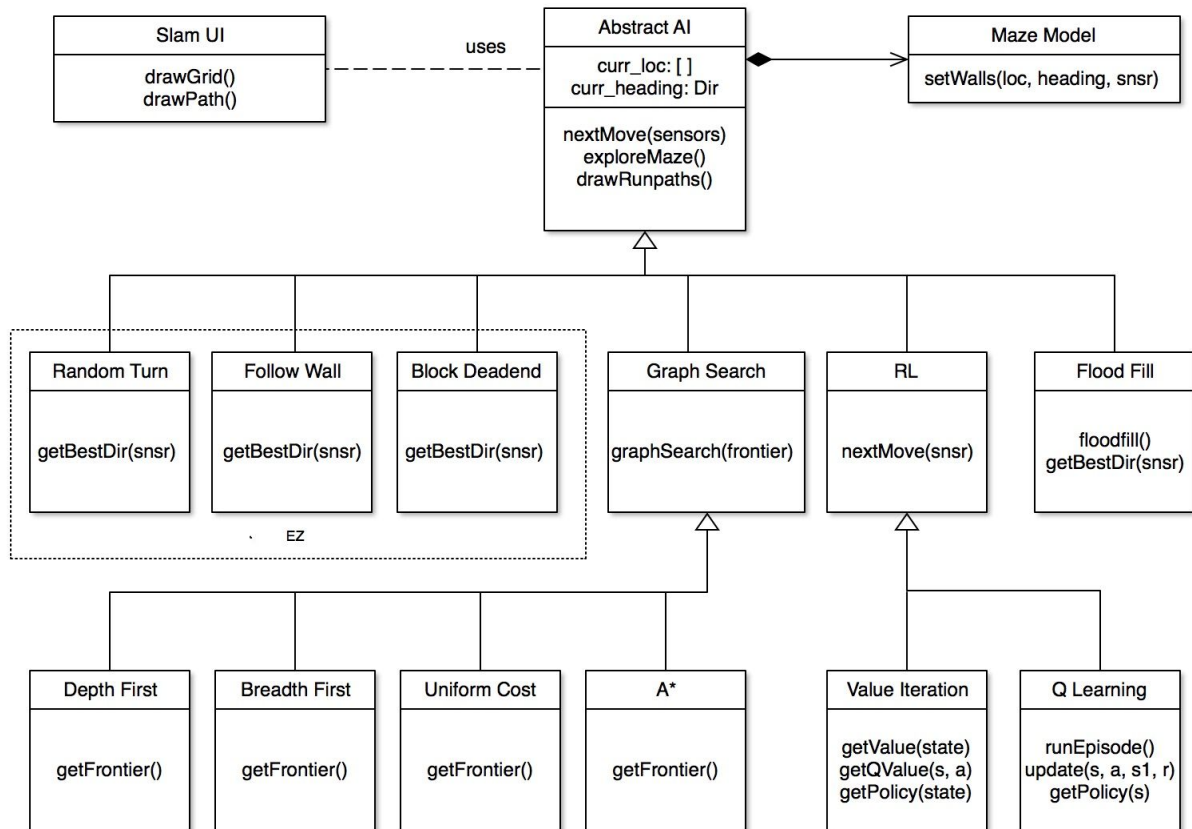


Fig 15: Class Diagram

'Abstract AI' is the most important class of the project. It is the parent class of all AI implementations. It carries the bulk of the logic and taps into each AI implementation for AI specific interpretation at relevant points in the flow. It uses 'Maze Model' to build and store the state of the maze. It also uses 'Slam UI' for bringing to life the smart moves of AI.

All the important class attributes and methods have been noted above.

For intricacies of each AI implementation, it is best to go through the source code and checkout inline comments.

## Sequence Diagram

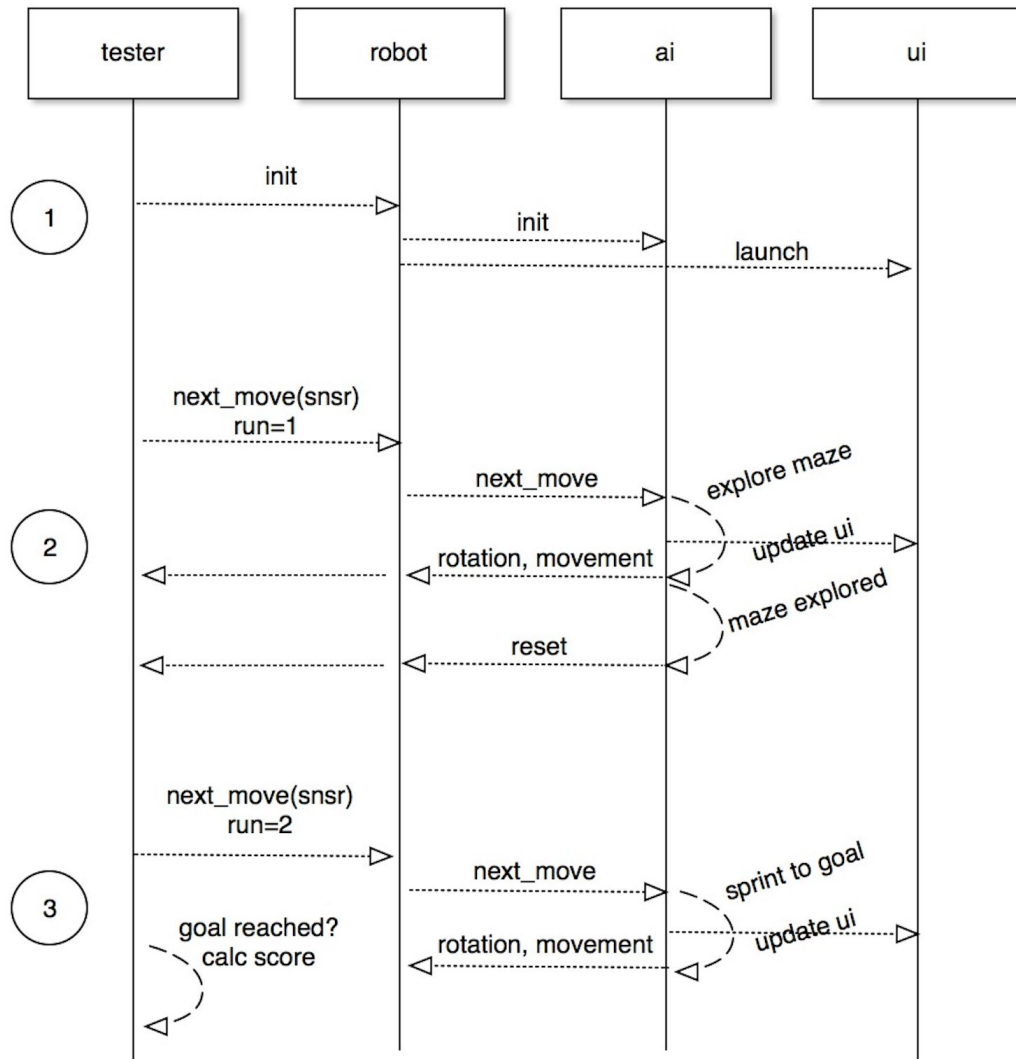


Fig 16: Sequence Diagram

The sequence of events in 'solving maze' can be broadly divided up in 3 stages:

1. **Init:** All components are initialized during this stage. `Tester.py` (main) instantiates `Robot`, which in turn initializes chosen `AI` module and launches `UI`
2. **Run=1:** This stage is also called 'maze exploration' stage. `Tester` in a loop keeps asking `Robot` to provide its next location and orientation. `Robot` in turn consults the chosen `AI` module to decide its next move based on the supplied sensor data. In these sequence of movements, `Robot` slowly discovers the maze. Once maze has been sufficiently explored, `Robot` 'reset' `run=1`.

3. Run=2: In this stage, Robot zooms from start to finish for every next\_move request from the Tester. Robot does this by exploiting the information it has gathered about the maze in the previous stage. Once the goal is reached, Tester calculates and publishes Robot Score. Lower scores are better.

## Commands

To run the tester

```
python tester.py test_maze_01.txt
```

To visualize a maze

```
python showmaze.py test_maze_01.txt
```

Other valid options are:

test\_maze\_01.txt, test\_maze\_02.txt, test\_maze\_03.txt, taiwan-2016-world-championship.txt

## Refinement

Of all the AI techniques under review in this project, Flood Fill algorithm is most promising, because neither its exploration nor its exploitation phase uses Random turns. Its every moment is precisely guided by heuristics known as depths. Depths steer robot straight towards the goal position without any unnecessary turns along the way. This approach is wonderful and works great if entire maze is known beforehand, where computed depths are optimal and the traced path from it is optimal as well.

However in our project, we are not given entire maze beforehand. Algorithms are supposed to discover maze as much as possible in exploration phase. For this, textbook Flood Fill algorithm prescribes that we compute initial depths of the maze by assuming that there are no walls in the maze. As the robot starts navigating towards the goal position following the initial depths and runs into any wall, we recompute depths for the entire maze with known knowledge of walls. New depths will steer the robot around the walls and towards the goal. This process is repeated until the robot finally reaches the goal position in the center of the maze. There is no guarantee that the path discovered in this process is optimal, as probably most of the maze is still unexplored. Hence we need an improvement over the textbook approach.

Improved Flood Fill approach has two objectives: explore maze more until the optimal path to the goal is found. For this, we refined (or augmented) the textbook FF algorithm — Once the robot reaches the center goal, we change its mission to find its way back to the start position. Since the knowledge of walls is increased, the computed reverse depths are more accurate and closer to optimal depths. New depths will steer robot back to the start position



on a new path, exploring new areas of maze along the way. We keep a count of number of steps taken to reach center and to reach start position. Once the robot reaches the start cell, we reverse robot's mission again to find center. As we keep repeating this back n forth process, new area of maze is explored and computed depths approach optimality. At certain point, we notice that number of steps taken to reach center is equal to number of steps taken to reach start cell. We stop our exploration phase at that point.

Thus refined FF algorithm approach finds THE optimal path to the goal position in the next exploitation run. For comparison, we present the Robot Scores for FF algorithm before and after refinement in the results section below.

## Results

### Algorithm Evaluation and Validation

Following are the robot scores recorded for various AI algorithms for each of the 3 mazes across 5 executions. 2nd column in the table is the individual Robot Score from each execution. 3rd column is the Average Normalized Robot Score per maze. Lastly, the 4th column is the Final Score of the algorithm calculated as average of Normalized Scores.

If an execution failed to complete within allotted time, it is marked with an X. If an AI algorithm has atleast one failed execution, it is considered not robust solution.

\* X = Allotted time exceeded, Lower Scores are better, Best Final Score is 1.000

Maze	Robot Score in 5 Executions (R)					Avg. Normalized Robot Score (N = R / Best R Score)	Final Avg Score (F = Avg N)
Random Turn							
test_maze_01	266.767	150.600	66.600	X	460.200	6.455	6.060 Not Robust
test_maze_02	X	X	X	X	X	X	
test_maze_03	X	273.267	X	524.767	231.967	5.666	
Follow Wall							
test_maze_01	X	X	X	X	X	X	X Not Robust
test_maze_02	X	X	X	X	X	X	
test_maze_03	X	X	X	X	X	X	
Block Deadend							
test_maze_01	744.833	143.700	77.367	496.867	264.900	9.449	7.163 Not Robust
test_maze_02	468.300	310.067	241.667	X	207.233	5.572	
test_maze_03	X	X	576.767	207.200	X	6.468	

Depth First Search							
test_maze_01	44.067	42.900	47.267	44.867	43.333	1.217	1.457 Not Robust
test_maze_02	X	111.533	81.133	77.067	X	1.633	
test_maze_03	88.133	120.800	63.033	86.633	102.233	1.521	
Breadth First Search							
test_maze_01	42.567	37.800	35.867	41.800	35.933	1.061	1.067 Not Robust
test_maze_02	69.000	53.267	55.633	X	59.900	1.080	
test_maze_03	65.467	63.333	62.433	62.233	67.433	1.059	
Uniform Cost Search							
test_maze_01	36.167	38.733	39.100	41.100	42.567	1.081	1.062 Not Robust
test_maze_02	56.933	X	X	54.167	52.633	0.991	
test_maze_03	65.233	61.967	78.433	X	64.700	1.115	
A* Search							
test_maze_01	44.333	41.633	41.867	41.000	41.833	1.152	1.201 Not Robust
test_maze_02	70.167	67.000	76.100	63.000	55.767	1.206	
test_maze_03	67.367	79.933	X	67.967	86.467	1.245	
Value Iteration							
test_maze_01	38.000	35.767	39.633	36.067	46.233	1.070	1.065 Not Robust
test_maze_02	62.167	51.667	X	51.033	57.167	1.008	
test_maze_03	63.233	X	62.400	77.400	X	1.117	
Q Learning							
test_maze_01	37.500	38.800	39.967	38.700	37.467	1.053	1.066 Not Robust
test_maze_02	61.300	56.700	52.733	66.767	56.967	1.069	
test_maze_03	64.833	X	62.033	60.133	74.000	.0771	
Flood Fill (textbook)							
test_maze_01	37.146	37.146	37.146	37.146	37.146	1.016	1.039 Robust
test_maze_02	56.981	56.981	56.981	56.981	56.981	1.035	
test_maze_03	64.678	64.678	64.678	64.678	64.678	1.067	
Flood Fill (refined)							
test_maze_01	36.567	36.567	36.567	36.567	36.567	1.000	1.000 Robust
test_maze_02	55.067	55.067	55.067	55.067	55.067	1.000	
test_maze_03	60.600	60.600	60.600	60.600	60.600	1.000	

**Selection Criteria:** The threshold for an acceptable Final Robot Score is 1.200. Any score above this threshold means that algorithm is taking too many steps, potentially wasteful cell revisits, to complete the maze as compared to the optimal solution. The best possible Final Score is 1.000. Scores greater than 1 and near 1 are good scores. Scores above 1.2 are bad.

Having an acceptable Final Robot Score is not enough. A winning algorithm must consistently complete any maze as well. In other words it has to be robust.

Following is the summary of results of evaluating various AI algorithms:

Note: Lower Robot Scores are better, Best Final Score is 1.000. Acceptable Final Score is between 1.0 and 1.2

Algorithm	Final Avg Robot Score	Comments
Random Turn	6.060	Way above the acceptable Robot Score threshold of 1.2. Not robust. Hence rejected.
Follow Wall	Allotted time exceeded	Failed
Block Deadend	7.163	Way above the acceptable Robot Score threshold of 1.2. Not robust. Hence rejected.
Depth First Search	1.457	Worst among graph search algos. Robot Score above acceptable threshold of 1.2. Not robust. Hence rejected.
Breadth First Search	1.067	Robot Score within acceptable range. However AI technique is not robust because of unpredictable (random) nature of exploration run. Hence rejected.
Uniform Cost Search	1.062	"
A* Search	1.201	"
Value Iteration	1.065	Values do not converge. Luckily policy converges much early, which is what we care about. Robot Score within acceptable range. However AI technique is not robust because of unpredictable (random) nature of exploration run. Hence rejected.
Q Learning	1.066	Very powerful algo. It can find optimal


		path just by trial and errors. However, takes too many trials. Robot Score within acceptable range. However AI technique is not robust because of unpredictable (random) nature of exploration run. Hence rejected.
Flood Fill (textbook)	1.039	Robot Score within acceptable range. Robust solution. However not the best Robot Score.
Flood Fill (refined)	1.000	Robot Score within acceptable range. Robust solution. Found the optimal path to the goal. Both exploration and exploitation runs are Systematic and Predictable. Best Robot Score too. 

Table 2: Algorithm Evaluation and Results

Our winning algorithm is Flood Fill (refined) algorithm, because its robot score is not only within acceptable range but also the best possible score. It has consistently found the optimal path to the goal. It never failed to find a solution in any execution for any test maze. In short, it is a Robust algorithm. Additionally, as you will see later in Conclusion section of this report that this implementation of algorithm went on to win 2016 micromouse world championship maze, reasserting its accuracy and robustness.

On the other hand, every other algorithm, namely Random Turns, Follow Wall, Block Deadend, DFS, BFS, UCS, A\*, Value Iteration and Q Learning all have atleast one failed execution. In other words, these algorithms are not robust. Hence we have no choice but to reject them.

## Justification

As noted previously in the Benchmark section of this report — Winning algorithm will consistently complete any maze. In other words, it has to have not only a good score but also be a robust technique.

We have chosen the threshold for an acceptable Final Normalized Robot Score to be 1.200. It has built in safety buffer of 20% to account for any possible variations in the test environment and quirks in the algorithm implementation. Any score above this threshold

means that algorithm is taking too many steps to complete the maze as compared to the optimal solution and hence must be rejected. The best possible Final Score is 1.000.

Our implementation of Flood Fill algorithm has scored 1.000, which is not only within the defined acceptable Robot Score threshold, but also the best possible score. Moreover, it never failed to find a solution in any execution for any of the test mazes. Hence it has met both the objective and subjective criteria to be the winning algorithm.

Flood Fill is the only algorithm that does not have any element of randomness in its process. It systematically approaches the problem and methodically finds the shortest path to the goal cell. It is the only algorithm that is predictable too. For a given maze, it always produces the same results consistently.

In the next section we will try it with an unseen complex maze from one of the micromouse world competition to see if our implementation is general enough to tackle new and complex mazes.

## Conclusion

### Free-Form Visualization

I recreated taiwan-2016-world-championship maze from [here](#). It is a 16x16 maze with a goal room in the center of the maze. It is a braid maze containing loops and dead ends. Maze has been specifically designed to discourage simple AI techniques and brute force approaches.

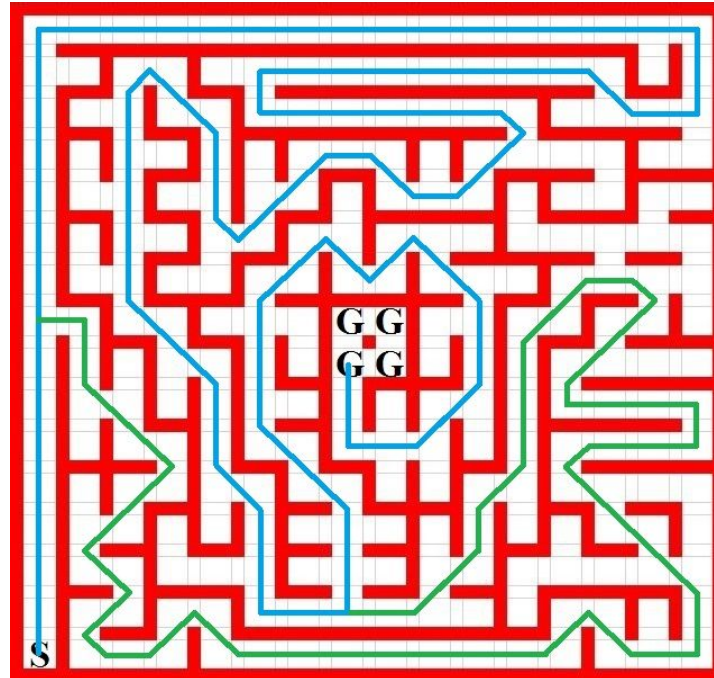


Fig 17: 2016 Micromouse World Championship Maze, Taiwan

There are 2 typical paths for this maze design, where the Green path was chosen by 2nd place and the blue path was picked by 1st place. Let us now find out, which path my implementation chooses, if any. Godspeed "Ranger" :)



Wow! Did you see that? Rather could you see that? It was all so fast. Let us watch it again :). We just won the 2nd place in world micromouse competition. Our Robot, Ranger, chose Green path all the way to the goal. Frankly, we actually came first. Green path is the shortest path, which is what we are aiming for in this project. Blue path is the fastest path (less number of turns for toy robot). Congratulations Ranger!

## Reflection

Oh boy! From watching [Johnny Sokko & His Giant Robot](#) on black & white TV as a kid in India.. to presenting a paper on Robotics in my first year of engineering and then building a shoebox on wheels with 8085 microprocessor and assembly language as my final year project... It has been an incredible journey. Like many kids, Robots always captivated my imagination. But that's not a very interesting story in itself. That is about 18 years old story.

My entire work life in Silicon Valley, I don't know how the Robot in me died. 10 years ago when Stanley crossed the finish line, it reignited a spark in me. I knew then, that in few years

I'll be doing what I'm doing right now (writing a report on robots) and starting self-driving nanodegree in few weeks. Thank you prof. Thrun. Thank you Udacity. You have started a chain reaction that will revolutionize human life in a short period.

As part of the Machine Learning Nanodegree project suggestions, when I saw "Robot Motion Planning" - I decided to work on it without a second thought. I had no clue what it would take to finish the project. I just wanted to Do It. And I'm so glad that I decided so. I ended up taking "Artificial Intelligence for Robotics (cs373)" and got some taste of "real robotics". It has been an exhilarating experience; more than Machine Learning. It is no exaggeration when I call it a "life changing" event.

My wife and I started solving mazes on paper. The best we could do together were those EZ (or common sense) solutions. I knew Reinforcement Learning techniques, that we learnt in MLND, could somehow be applied here. So I tried it next. It took me a while to correctly apply these techniques, and it worked. But I was not happy with the Robot Score and unpredictable nature of my solution. CS373 taught A\* and other graph search techniques and SLAM. These techniques worked as well, but once again unpredictable nature of the solution kept me from declaring success. Finally, in my quest to find a better solution, I started reading up more and more about Micromouse competitions. That is when I discovered "Flood Fill" algorithm. It is like A\* with unique twist in Heuristics. When I implemented it and saw Robot reach the goal for the first time, I stood in awe for few mins. It worked and it worked again and again and again consistently with accurate results. That's my Dragon! (it is a baby book, btw). Well done.

From a TV show to Stanley's news to discovering Flood Fill - the journey is on!

## **Improvement**

Without doubt the current scope of the project is awesome. It fulfils its objective of introducing Robot Motion Planning and then some. That been said, here are few areas of improvement that can bring our robot much closer to real life implementation:

- **Robot Motion Noise:** In real life, robot motion is hardly 100%. There is always some probability that robot ends up in a location different from the intended destination. Although the robot will make it to the final goal, it may take a path different from optimal path.
- **Sensor Noise:** Real sensors are also rarely 100% accurate. Their output can be drawn from a probability distribution. For practical motion planning we should account for it as well.
- **Generating Smooth Path:** Real robots cannot take 90 degree turns. Slow, smooth and curvy paths, especially around the corners are much preferred. We can interpolate the

discrete path coordinates obtained from A\* or Flood Fill and come up with new path coordinates that will allow robots to slowly negotiate the corners, thus producing much smoother ride for us :)

- **Keeping safe distance:** It is not enough for us to plan shortest path to the goal. Doing so may bring robot dangerously close to the obstacles or walls. We should always account for some safety margin and interpolate robot path coordinates to keep it at a safe distance.
- **PID Control:** Keeping a robot on top of planned continuous smooth path is not easy. Robot's wheel might be misaligned or trying to turn steering towards the intended path too much may cause overshoot. A poorly controlled robot will oscillate around its planned path. In a nutshell, it is no easy feat to keep the robot on a continuous path. We can apply techniques like Twiddle for PID control to fix this.
- **Diagonal Movement:** If a maze in continuous domain is such that it requires robot to go diagonally across the cells, then discrete robot might not follow that path. It will look for bit longer path, but with 90 degree turns. If a such a 90 degree path does not exists, then discrete robot will not be able to solve the continuous maze.

**Other ideas for improvement:** I think in order to bring our robot problem much closer to real life, we can incorporate following additional complexities:

- **Sensor Fusion:** Instead of simply depending on distance sensors, we could somehow include multiple sensors like Lidar, Radar, GPS and Camera into our problem definition. This will give students additional opportunity to understand how to work with multiple sensors.
- **Neural Nets:** I'm sure a real life robot will not work without a NN component. We should look for a place in the project to include the use of NN.
- **Computer Vision:** Similarly, we totally missed out on Computer Vision aspect in robot motion planning. It would be really nice if we could incorporate it along with NN in our toy project.

Given an opportunity, I'd love to come up with a toy project, that would incorporate all these aspects as part of its scope.