Capstone Project

Plot and Navigate a Virtual Maze

Definition

Project Overview

In this section, look to provide a high-level overview of the project in layman's terms. Questions to ask yourself when writing this section:

- ***Has an overview of the project been provided, such as the problem domain, project origin, and related datasets or input data?***
- ***Has enough background information been given so that an uninformed reader would understand the problem domain and following problem statement?***

The project has been defined for you in this document. Write a paragraph or two using your own words as to what the project is about, and summarize the project description.

The project is to have a virtual robot plot a course from a corner of a virtual maze (the start point) to its center (the end point or goal). The robot is allowed two runs in the maze. In the first run, the aim is for the robot to map out or explore the maze as much as possible and more importantly find the center.

The aim of the second run is to reach the center (goal) in the quickest possible time ie. in the least amount of moves, using the information learn't on the first run.

Both runs have to complete successfully within a time limit of 1000 time steps in total.

Problem Statement

In this section, you will want to clearly define the problem that you are trying to solve, including the strategy (outline of tasks) you will use to achieve the desired solution. You should also thoroughly discuss what the intended solution will be for this problem. Questions to ask yourself when writing this section:

- ***Is the problem statement clearly defined? Will the reader understand what you are expecting to solve?***
- ***Have you thoroughly discussed how you will attempt to solve the problem?***
- ***Is an anticipated solution clearly defined? Will the reader understand what results you are looking for?***

The problem which needs to be solved has been defined for you in this document. Discuss what

that problem is, and the motivation (outline) you will use to solve this problem, including what solutions are expected.

The problem that this project seeks to address is how to guide a virtual robot through a selection of virtual mazes, from a known start point to an unknown end point and achieve this in the fastest possible time ie. find the shortest possible route.

The Mazes

The mazes are grids of **n x n** squares, where **n** is an even number, that can take one of the following values { **12, 14, 16** }.

Around the perimeter of the grid are walls that block all movement, as do the internal walls of the mazes.

The starting square is always the bottom-left hand corner of the grid, co-ordinates [0, 0]. In the center of the grid is the goal room consisting of a 2 x 2 square.

The robot has a total of 7 mazes to solve:

- Three mazes that are supplied, a 12x12, a 14x14 and a 16x16
- Three unknown mazes that will be used in the project evaluation.
- One maze that I will create

Solution Strategy

As the problem is really a combination of two problem, ie.

1. Find a solution to the maze within the allotted time limit (maze run 1)
2. Find the optimal solution to the maze within the allotted time limit (maze run 2)

I intend to approach each of the above problems with a different strategy.

Strategy 1: Explore and Solve

The approach to the first run of each maze will be to use a reinforcement learning variant to explore the maze and ultimately find the goal location without expending too much of the 1000 time-step limit. As much as possible of the maze should be explored inorder to find as many routes to the goal as possible. The more of the maze that can be explored gives the greatest possible chance of an optimal route being discovered.

Not withstanding how much of the maze gets explored, the absolute minimum requirement of the first run is that the goal location must be discovered within the time limit and leave sufficient time to complete run 2.

Strategy 2: Find the Optimal Route to the Goal

The approach to the second run of each maze will be to use the information gained in the first run to find the shortest path to the goal from the start point. The second run must be completed within the time limit remaining after the first run.

If the exploration run has not discovered all possible routes to the goal there is a possibility that the shortest path might not be the absolute shortest path to the goal.

The route will be computed after the first run has completed and before the second run starts. The route computed will be used to direct the robot to the goal during the second run.

Metrics

In this section, you will need to clearly define the metrics or calculations you will use to measure performance of a model or result in your project. These calculations and metrics should be justified based on the characteristics of the problem and problem domain. Questions to ask yourself when writing this section:

- ***Are the metrics you've chosen to measure the performance of your models clearly discussed and defined?***
- ***Have you provided reasonable justification for the metrics chosen based on the problem and solution?***

A performance metric has been defined for you in this document. Report what this metric is and how it will be calculated in your implementation.

The metric to be used in this project to evaluate its performance is the score.

For each maze the robot must complete two runs within the 1000 time step limit. The robot's score for each will be calculated as follows:

Score = 1/30 * (Number of time steps used in Run 1) + (Number of Time steps used in Run 2)

If the robot fails to complete either run within the time limit no score is achieved.

Analysis

Data Exploration

In this section, you will be expected to analyze the data you are using for the problem. This data can either be in the form of a dataset (or datasets), input data (or input files), or even an environment. The type of data should be thoroughly described and, if possible, have basic statistics and information presented (such as discussion of input features or defining characteristics about the input or environment). Any abnormalities or interesting qualities about the data that may need to be addressed have been identified (such as features that need to be transformed or the possibility of outliers). Questions to ask yourself when writing this section:

- ***If a dataset is present for this problem, have you thoroughly discussed certain features about the dataset? Has a data sample been provided to the reader?***
- ***If a dataset is present for this problem, are statistics about the dataset calculated and reported? Have any relevant results from this calculation been discussed?***
- ***If a dataset is not present for this problem, has discussion been made about the input space or input data for your problem?***
- ***Are there any abnormalities or characteristics about the input space or dataset that need to be addressed? (categorical variables, missing values, outliers, etc.)***

Data Exploration: Use the robot specifications section to discuss how the robot will interpret and explore its environment. Additionally, one of the three mazes provided should be discussed in some detail, such as some interesting structural observations and one possible solution you have found to the goal (in number of steps). Try to aim for an optimal path, if possible!

Due to the nature of the problem there is no input dataset in the traditional sense, the maze itself is the input space and as the robot moves about that space, its sensors provide data about the space. The robot has three obstacle sensors, mounted on the front, left and right sides. The sensors indicate the number of adjacent open grid squares and their direction.

At the start of each time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, center and right sensors.

For example: A sensor reading of [1, 5, 0] indicates

- there is 1 open grid square to the robot's left.
- there are 5 open grid squares directly in front of the robot.
- no open grid squares to the robot's right. No open grid squares indicate the presence of a wall.

The robot is able to move a maximum of 3 grid squares either forward or backward, assuming sufficient open grid squares, in any one time step.

For the purposes of my project I have decided the following:

- To limit the robot's movement to only 1 grid square per time step during the first run. This will help maximize the exploration in that openings are less likely to be passed by and thereby have a greater chance of being explored.
- The robot will never hit any wall.
- The robot will only move forwards, never backwards. This was done for two reasons:
  1. The way I have coded my **robot.py** does not require the robot to move backwards. Moving backwards is just a waste of time-steps.
  2. The way **tester.py** is coded, if I want to move the robot backwards and rotate after getting into a dead-end, the rotate happens first, followed by the move backwards. This results in the robot backing into the wall. To me it would be more logical to get out of a dead-end by first backing out and then perform the rotation.

The 12x12 Maze

As most of my initial testing was done using the supplied 12x12 maze, **see test_maze_01.txt**, I will use that one for an in-depth discussion. See the diagram below **"12 x 12 Maze – A possible Solution"**.

The maze is a 12x12 grid, I number the rows and columns 0-11, the starting position in the maze is the bottom left hand corner, which corresponds to grid position [0, 0]. The goal of the maze is at the center of the grid, at grid position [5, 6].

I will use compass directions, north, south, east and west to track the orientation of the robot, north is to the top of the maze, south is to the bottom, east is to the right hand side and west is to the left hand side.

At the start of both runs, the robot is positioned at grid square [0, 0] and is facing north.

Maze Details

There are 4 possible openings from column 0 that allow entry into the heart of the maze. These openings give rise to three distinct regions in the maze (refer to diagram below):

1. Top section of the maze – rows 8, 9, 10 and 11
2. Middle section of the maze – rows 3, 4, 5, 6 and 7
3. Lower section of the maze – rows 0, 1 and 2

The middle section leads directly to the lower section, and vice-versa, the middle section also gives access to the top section once in column 4.

Key Locations

In the maze there are key locations that the robot must find during exploration if it has any hope in finding the goal. If the robot finds itself in the lower section it has to find and pass grid square [11, 0] while going east then north to have a chance of locating the goal. Likewise the robot has to locate grid square [9, 10] moving either east or west when approaching the goal from the top section.

There are two other key grid squares, either of which must be located if the robot is to locate the goal. They are grid squares [8, 5] and [8, 7], each provide the only access from the lower and top sections to the final few grid squares where the goal is located.

In-order to solve the maze, not only does the robot have to locate these key locations, it has to make the appropriate turns to enable it to progress.

Dead-Ends

Other features of note in the maze are dead-ends. The robot finds itself in a dead-end when a wall directly in front prevents forward movement and walls to both its left and right prevent movement to the left and right. The 12x12 maze has dead-ends at the following grid squares: [1, 7], [8, 1], [10, 4], [7, 4], [9, 11] and [10, 9].

Maze Solution

One possible solution to the maze is marked on the diagram below with a red line, this route takes 31 time-steps to reach the goal assuming the robot moves one grid square per time-step. If the robot would move at most three grid squares per time-step along the same route it would take 17 time-steps to reach the goal.

Exploratory Visualization

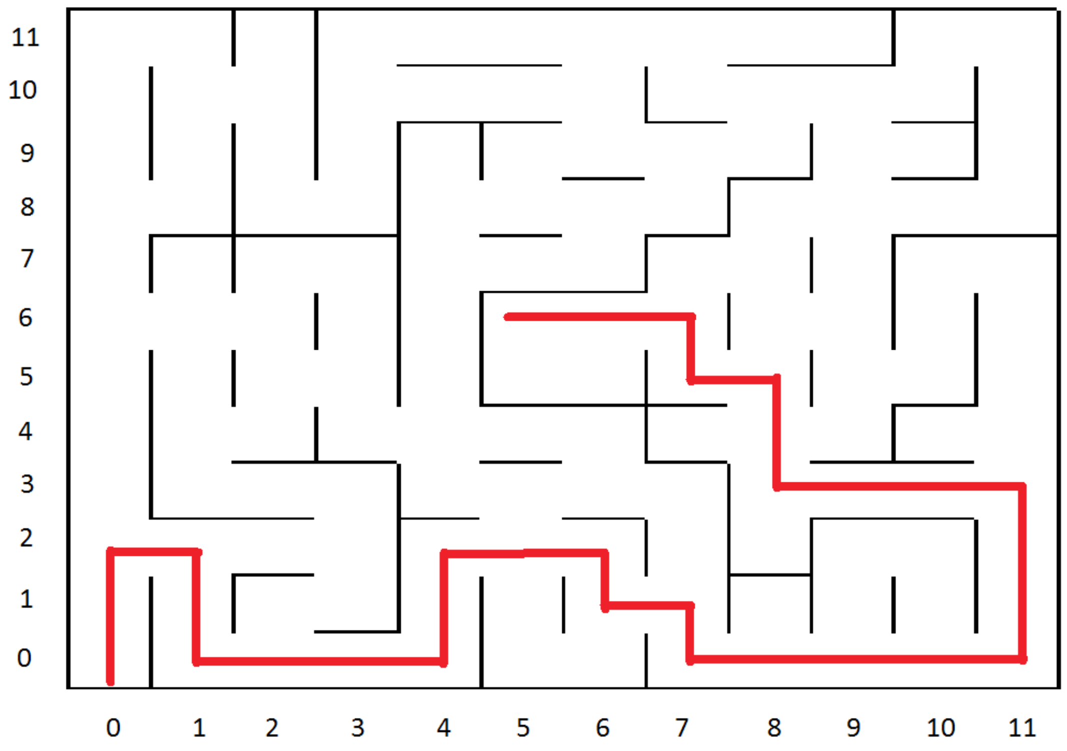
In this section, you will need to provide some form of visualization that summarizes or extracts a relevant characteristic or feature about the data. The visualization should adequately support the data being used. Discuss why this visualization was chosen and how it is relevant. Questions to ask yourself when writing this section:

- ***Have you visualized a relevant characteristic or feature about the dataset or input data?***
- ***Is the visualization thoroughly analyzed and discussed?***
- ***If a plot is provided, are the axes, title, and datum clearly defined?***

Exploratory Visualization: This section should correlate with the section above, in that you should provide a visualization of one of the three example mazes using the `showmaze.py` file. Your explanation in Data Exploration should coincide with the visual cues from this maze.

The diagram below depicts the 12x12 Maze from file **test_maze_01.txt,** the rows and columns are numbered 0-11. The red line depicts one possible solution to the maze. The Data Exploration section above provides a detailed discussion of the maze and its features.

12 x 12 Maze - A Possible Solution



Algorithms and Techniques

In this section, you will need to discuss the algorithms and techniques you intend to use for solving the problem. You should justify the use of each one based on the characteristics of the problem and the problem domain. Questions to ask yourself when writing this section:

> ***•Are the algorithms you will use, including any default variables/parameters in the project clearly defined?***
> ***•Are the techniques to be used thoroughly discussed and justified?***
> ***•Is it made clear how the input data or datasets will be handled by the algorithms and techniques chosen?***

Algorithms and techniques used in the project are thoroughly discussed and properly justified based on the characteristics of the problem.

As stated above in the “Problem Statement” section, I will be using a different approach in each of the two runs for each maze.

Maze Run 1

In the first run of each maze, the algorithm that will be used to explore and solve the maze is a DYNA based algorithm called Prioritized Sweeping.

DYNA Architecture

In Reinforcement Learning there is a trade-off between spending time acting in an environment and spending time planning what actions are best. In model-free methods of Reinforcement Learning eg. Q-Learning, the agent updates only the state most recently visited. At the opposite end of the spectrum, the dynamic programming methods eg. model-based certainty equivalence MDP's, re-evaluate the utility of every state in the environment after each step. Sutton's DYNA architecture has strategies that are both more effective than model-free learning and more computationally efficient than the certainty equivalence approaches.

DYNA operates in a loop of interaction with the environment. It simultaneously uses experience to build a model, uses experience to adjust the policy and uses the model to adjust the policy. The model is defined as anything the agent can use to predict how the environment will respond to its actions.

For the purposes of this project, the model is comprised of two tables:

1. model_s – which records **state**, **action** pairs and their corresponding next state.
2. model_r – which records **state**, **action** pairs and the reward received for taking action **action** when in state **state**.

DYNA given an experience tuple <s, a, s', r> behaves as follows:

- Update the model, incrementing statistics for the transition from **s** to **s'** on action **a** and for receiving reward **r** for taking action **a** in state **s**. The updated models are **T_hat** and **R_hat**.
- Update the policy at state **s** based on the newly updated model using the rule
    - Q(s, a) = R_hat(s, a) + gamma Σ s' T_hat(s, a, s') maxQ a' (s', a')
- Perform k additional updates: choose k state-action pairs at random and update them according to the same rule as before:
    - Q(sk, ak) = R_hat(sk, ak) + gamma Σ s' T_hat(sk, ak, s') maxQ a' (s', a')
- Choose an action **a'** to perform in state **s'**, based on the Q values but perhaps modified by an exploration strategy.

The DYNA algorithm requires about k-times the computation of Q-Learning per instance, but requires an order of magnitude fewer steps of experience than does Q-Learning to arrive at an optimal policy. However, DYNA does require about 6x times more computational effort.

There are many variants of the DYNA architecture, as mentioned above, for the purposes of this project I chose to implement “Prioritized Sweeping”.

Prioritized Sweeping

The DYNA algorithm uses state-action pairs to update its Q-values during the time interval between interactions with the real world. However, all state-action pairs are not equal in their importance, as not all will eventually lead to the goal.

The Q-value indicates the degree of the effect of a state-action pair. Also, the preceding state-action pairs to those that have had a large effect in the past are also more likely to have a large effect in the future. So it is natural to prioritize the updates according to a measure of their urgency or influence, this is the idea behind the prioritized sweeping algorithm. During execution the agent retrieves these influential state-action pairs and updates their corresponding Q-values to speed up the policy learning.

The Prioritized Sweeping Algorithm

```
Initialise Q(s, a), Model(s, a) for all s, a, and PQueue to empty.
Do Forever:
    (a) s ← current (non-terminal) state
    (b) a ← policy(s, Q)
    (c) execute action a; observe resultant state s', and reward r
    (d) Model(s, a) ← s', r
    (e) p ← | r + gamma max a' Q(s', a') – Q(s, a) |
    (f) if p > theta, then insert s, a into PQueue with priority p
    (g) Repeat N times, while PQueue is NOT empty:
        s, a ← first(PQueue)
        s', r ← Model(s, a)
        Q(s, a) ← Q(s, a) + alpha[r + gamma max a' Q(s', a') – Q(s, a)]
        Repeat for all $\bar{s}$, $\bar{a}$ predicted to lead to s:
            $\bar{r}$ ← predicted reward
            p ← | $\bar{r}$ + gamma max a Q(s, a) – Q($\bar{s}$, $\bar{a}$) |
            if p > theta then insert $\bar{s}$, $\bar{a}$ into PQueue with priority p
```

In model-based Reinforcement Learning, as in the basic model-free approach, the primary goal of learning is the improvement of a behavioral policy in order to maximize a numerical reward. However, the approach differs from model-free Reinforcement Learning in that simultaneously, the agent attempts to learn a model of the environment.

Having a model, allows the agent to predict the consequences of actions before they are taken, allowing the agent to generate virtual experience as well as perform mental search through a problem space to locate an efficient solution. Thus model-based Reinforcement Learning integrates both learning on the basis of past experience and planning future actions. Planning is any computational process that uses a model to create or improve a policy.

Prioritized sweeping is a model-based Reinforcement Learning method that attempts to achieve a good estimate of the value of environmental states. Prioritized sweeping falls between the model-free methods and dynamic programming methods in that only the most “important” states are updated. The so-called important states are determined by a priority metric that attempts to measure the anticipated size of the update for each state. This metric or heuristic seeks to focus on the states that are likely to have the largest errors and hence the largest change on the value function.

Prioritized sweeping balances performing actions in the environment with propagating the value of states. After updating the value of state s (see step (g) above), all states from which the agent might reach state **s** in one step are examined, a priority based upon the expected size of their change in value is then assigned. The largest change in value results in the highest priority in the queue for a particular experience or state.

Maze Run 2

Once the maze has been explored and solved in Run 1, the information gained will be used as input to the algorithm to be used in Run 2. The purpose of Run 2 is to find the optimal solution to the maze that allows the robot to reach the goal in the least number of steps. The A* (A Star) search algorithm will be used to find the optimal route using the information gained from Run 1.

A* Search

The A* search algorithm is a graph search algorithm, in that it takes as input a “graph”, a graph being a set of locations or “nodes” and the connections or “edges” between them. Thus a maze problem is an ideal candidate to be solved using the A* search algorithm as a maze is just a series of interconnected grid squares or nodes.

The path found by A* search is made of graph nodes and edges. Specifically in this project the path returned is a list of grid squares and the correct action to take from each, that will ultimately lead to the goal.

The A* search algorithm is a combination of Dijkstra's algorithm and the Best-First algorithm. Both Dijkstra's and the Best-First algorithm's when searching use a frontier that expands in all directions. This is a reasonable choice if trying to find a path to all locations or to many locations. Dijkstra's algorithm works well to find the shortest path, but it wastes time exploring in directions that are not promising. Whereas the Best-First algorithm explores in promising directions but it may not find the shortest path.

For this project we are trying to find a path to a single location. We need to make the frontier expand towards the goal more than it expands in other directions. A* search uses a heuristic function that tells how close we are to the goal. A heuristic can be defined as something that is not a definite series of steps to a solution, but it helps determine our answers in a rough way.

When searching, A* search picks a node that looks the most promising, a node is judged the most promising if it has the least projected cost.

The Cost Function

The cost of a node, denoted as **f**, is given by the following formula:

$$\mathbf{f = g + h}$$

where:
g: is the cost it took to get to the node, most likely the number of grid squares we passed by from the start.

h: is our heuristic, and is a guess as to how much it will cost to reach the goal from that node.

Once a possibility is generated and its cost calculated, it stays in a list of possibilities until all the better nodes have been searched before it. The output of the cost function, **f**, is used to re-order the nodes so that its more likely that the goal node will be encountered sooner.

A* search will find the best path in a very short time provided the **h** is perfect. We can't determine **h** perfectly without using some additional path finding so we just use an approximation.

Heuristics

On a grid or a maze there are well known heuristic functions to use. As the mazes in this project allow only 4 directions of movement (North, South, East and West) the Manhattan distance is used, this calculates the distance in steps between the current grid square and the goal grid squares. The distance is then multiplied by **D**, the minimum cost for a step or movement.

The Manhattan distance is defined by the following formula:

$$\mathbf{h = D * abs(a.x – b.x) + abs(a.y – b.y)}$$

where:
a – co-ordinates of the goal grid square
b – any location or grid square in the maze ie. current location of the robot
D – minimum cost for a step or movement

When picking a value for **D**, we use a scale that matches the cost function. It is usually set by the lowest cost between adjacent grid squares. Moving one step closer to the goal should increase **g** by **D** and decrease **h** by **D**. When you add the two, **f (g+h)**, will stay the same. That's an indicator that the heuristic and cost function scales match.

Breaking Ties

In some grid maps/mazes there can be more than one path with the same length. A tie breaker can be used to make the **f** value differ, this can be done by scaling **h** upwards slightly:

$$h \mathrel{*}= (1.0 + p)$$

p should be chosen so that:

p < (minimum cost of taking one step) / (expected minimum path length)

Assuming that the paths won't be more than 1000 steps long, I chose **p** = 1/1000.

A* Search Algorithm

The A* search algorithm is quite straight forward, see table below for pseudo-code:

```
OPEN = priority queue containing the START grid square or node
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
    current = remove lowest rank item from OPEN
    add current to CLOSED
    for neighbors of current:
        cost = g(current) + movement cost(current, neighbor)
        if neighbor in OPEN and cost less than g(neighbor):
            remove neighbor from OPEN, because new path is better
        if neighbor in CLOSED and cost less than g(neighbor):
            remove neighbor from CLOSED
        if neighbor not in OPEN and neighbor not in CLOSED:
            set g(neighbor) to cost
            add neighbor to OPEN
            set priority queue rank to g(neighbor) + h(neighbor)
            set neighbors parent to current
reconstruct reverse path from goal to start by following parent pointers
```

The OPEN set contains those nodes that are candidates for examining, initially it just contains one node, the starting grid square. For the mazes used in this project that is grid square [0,0].

The CLOSED set contains those nodes that have already been examined. Initially the CLOSED set is empty.

Each node also keeps a pointer to its parent or predecessor node, so that we can determine how it was found.

The main loop repeatedly pulls out the best node, **n** from OPEN (ie. the node with the lowest **f** value) and examines it. If **n** is the goal we have finished. A* search terminates only when a goal state is popped from the priority queue, OPEN.

Otherwise, node **n**, is removed from OPEN and added to CLOSED. Then its neighbors **n'** are examined. Neighbors are defined as those grid squares or nodes that are immediately adjacent and accessible from node **n**.

A neighbor that is in CLOSED has already been seen, so we don't need to look at it.

A neighbor that is in OPEN is scheduled to be looked at, so we don't need to look at it now.

Otherwise we add it to OPEN with priority **f**, with its parent set to **n**. The path cost to **n'** will be updated, ie. **g(n) + movement cost(n, n')**.

Once the goal is found, the reverse path is reconstructed from goal to start by following the parent pointers. Each grid square in the path is added to a python deque (a list like container that supports fast appends and pops from either end).

During Run 2, with each iteration, a grid square is popped off the deque, the grid square contains the appropriate action (one of “left”, “forward” or “right”) for the current location. Once the deque is empty the robot will be at the goal grid square.

Benchmark

In this section, you will need to provide a clearly defined benchmark result or threshold for comparing across performances obtained by your solution. The reasoning behind the benchmark (in the case where it is not an established result) should be discussed. Questions to ask yourself when writing this section:

- ***Has some result or value been provided that acts as a benchmark for measuring performance?***
- ***Is it clear how this result or value was obtained (whether by data or by hypothesis)?***

Benchmark: You will need to decide what you feel is a reasonable benchmark score that you can compare your robot’s results on. Consider the visualization and data exploration above: If you are allowed one thousand time steps for exploring (run 1) and testing (run 2), and given

the metric defined in the project, what is a reasonable score you might expect? There is no right or wrong answer here, but this will help with your discussion of solutions later on in the project.

A benchmark score will be specified for each maze in-order to assess the performance of the robot in solving each maze. As mentioned in the Metrics section above, the score is the sum of the number of time-steps used in Run-2 plus one-thirtieth of the number of time-steps used in Run-1.

As a benchmark figure for the number of time-steps used in Run-1 I have chosen a figure of 900 time-steps, one-thirtieth of that figure is 30, this will be used for all mazes. This allows the robot plenty of time to explore the mazes in Run-1 and also leaves sufficient time to complete Run-2. The Run-2 benchmark figures will be unique for each maze and will be calculated by manually finding the optimal number of time-steps needed to solve the maze.

The benchmark figures for each run of each maze can be summarized as follows:

Maze	Run-1 Benchmark	Run-2 Benchmark	Total Benchmark
12x12	30	31	61
14x14	30	43	73
16x16	30	51	81
My Custom 12x12	30	31	61

Thus the success or failure of the robot can be assessed by comparing the actual score obtained from each maze run to the appropriate benchmark score. If the benchmark score is equaled or bettered then the run is considered successful, otherwise the run is considered unsuccessful.

Using the optimum number of steps as the benchmark for Run 2 is somewhat optimistic as its unlikely that the robot will find the optimum route each and every Run 1, however it is a target to aim for.

Methodology

Data Preprocessing

In this section, all of your preprocessing steps will need to be clearly documented, if any were necessary. From the previous section, any of the abnormalities or characteristics that you identified about the dataset will be addressed and corrected here. Questions to ask yourself when writing this section:

- ***If the algorithms chosen require preprocessing steps like feature selection or feature transformations, have they been properly documented?***
- ***Based on the Data Exploration section, if there were abnormalities or characteristics that needed to be addressed, have they been properly corrected?***
- ***If no preprocessing is needed, has it been made clear why?***

Data Preprocessing: Because there is no data preprocessing needed in this project (the sensor specification and environment designs are provided to you), be sure to mention that no data preprocessing was necessary and why this is true.

As data is generated in real-time via sensor inputs to the robot as it moves about maze, there is actually no data to pre-process.

The maze itself is pre-defined and static once the robot starts moving about. The only information available about the maze at the beginning of Run 1 are the dimensions of the maze and the location of the starting grid square.

Implementation

In this section, the process for which metrics, algorithms, and techniques that you implemented for the given data will need to be clearly documented. It should be abundantly clear how the implementation was carried out, and discussion should be made regarding any complications that occurred during this process. Questions to ask yourself when writing this section:

- ***Is it made clear how the algorithms and techniques were implemented with the given datasets or input data?***
- ***Were there any complications with the original metrics or techniques that required changing prior to acquiring a solution?***
- ***Was there any part of the coding process (e.g., writing complicated functions) that should be documented?***

The process for which metrics, algorithms and techniques were implemented has been thoroughly documented. Complications that occurred during the coding process are discussed in some detail.

In addition, student's robot code consistently completes mazes (one learning run and one fast run) within a one thousand time step limit. This includes the three sample mazes provided in the starter code, the three mazes provided by evaluators.

The initial phase of implementation was solely directed at exploring and solving the maze during the first run. At this stage nothing had been implemented to perform the second run. If the maze could be solved in a timely fashion in Run 1 then the algorithm selected to perform Run 2, A* search, would always complete Run 2 within the remaining time.

Implementing Prioritized Sweeping

In implementing the final solution for this project, I went through several variants of Q-Learner before finally settling on Prioritized sweeping. Initially I started with the Reinforcement Learning class I developed for project **P4: Train a Smartcab to Drive** and did my initial testing using Q-Learning. The only code changes required were to implement a new function to generate the state. This of course meant settling on a state space, using the experience gained from P4, I quickly settled on a state string incorporating the following states:

- 99 – a static constant
- xx – the robot's x position in the maze, counting from zero.
- yy – the robot's y position in the maze, counting from zero.
- n – the compass heading of the robot, north, south, east or west represented by 1, 2, 3 or 4 respectively.

These states were then combined into a simple numeric "state string" of the form "99xxyyn". This would then serve as the key in the Q-Matrix, thus allowing a simple single value for the key rather than a long list of words and numbers.

My **create_state** function (see **ReinforcementLearning.py**) could then be updated easily by commenting/un-commenting various combinations of states in-order to form new "state strings" to allow testing of various combinations. Other states considered for the state string included:

- individual data points from the sensor
- distance to goal from current location

In all cases when I included more states in the state string, compared to the final values ie. 99xxyyn, always lead to the robot rarely finding the goal, this indicated that the state space was too large relative to the time allotted to explore and solve the maze. The smaller state string allowed many of the states to be explored without exceeding the time limit.

I have three action selection algorithm's from project P4, these being:

- **Epsilon Greedy** (function **chooseActionUsingEpsilonGreedy** in **ReinforcementLearning.py**)
- **SoftMax** (function **chooseActionUsingSoftMax** in **ReinforcementLearning.py**)
- **Maximum Q-value** (function **chooseActionUsingMaximumQValue** in **ReinforcementLearning.py**)

For initial testing these remained unchanged and were each tested to see which gave the best results. Overall testing with Q-Learning resulted in seeing the robot solve the maze within the time limit about 50% of the time.
With the limited success of Q-Learning, I then implemented the Q-Q Learning algorithm or Double-Q Learning algorithm. Here two Q matrices are maintained, on each iteration only one is updated, which one gets updated is chosen at random. This algorithm bought a noticeable improvement with about 80%-90% of runs successfully finding the goal within the time limit. However, this was still far from ideal as we are requiring all first runs to complete within the time limit, with sufficient time left over to allow the successful completion of Run 2.

The DYNA-Q algorithm was next to be implemented and tested, as before with the other algorithms tested, DYNA-Q was tried with each of the action selection methods. Methods **EpsilonGreedy** and **SoftMax** produced the most consistent results. Overall the DYNA-Q algorithm tended to consistently locate the goal within the time limit approximately 90-95% of the time. This lead to my final choice of algorithm, that being Prioritized Sweeping, see function **prioritizedSweeping** in **ReinforcementLearning.py**.

This algorithm was able to consistently explore the mazes and find the goal within the time limit while leaving sufficient time to successfully complete Run 2.

Implementing A* Search

Once the robot was able to successfully solve the maze in Run 1 and leave sufficient time for Run 2, it was time to implement the A* search algorithm.

The algorithm itself was straight forward to implement, but some additional consideration had to be given, specifically, what would be:

1. **The input to A* search ?** In a lot of search problems A* search would be given the entire environment to search, but in this case, the environment ie. the maze had already been explored so there was no need for A* search to re-explore the whole maze. The result of the exploration performed by prioritized sweeping is available in the Model that is produced. The model contains an entry for each state visited ie. each grid square, each entry also has the next state the robot moves to from that state. This information would be the input to A* search.

2. **What would be the output of A* search ?** Once Run 1 was completed and before Run 2 starts, A* search would have to be called to analyze the model and calculate the best path to the goal and provide the result in a structure that could then be passed to the **robot.py** function **next_move** in the form of "**action, movement**" ie. what action to take, one of left, forward or right and the number of steps to take.

Maze Visualizer

During the testing of the algorithms used to explore and solve Run 1 it became apparent that having some sort of graphic visualization of the robot's movement during the maze run would be very beneficial.

To facilitate this a small class **TheMaze** (in **MazeVisualizer.py**) was developed, this runs on a thread and displays the maze and the robot's progress during each of the runs.

In **robot.py** whenever a new location is calculated, the location [x, y] is added to a python FIFO queue. The python queue allows the sharing of data between threads, in this case, the function **next_move()** in **robot.py** places the current location of the robot in the queue on each iteration. In function **DisplayRobotsProgress** (in **MazeVisualizer.py**) while the queue is not empty the next location is popped from the queue and the maze diagram is updated with the current location of the robot.

If this is the robot's first visit to a grid square, a yellow circle is displayed to indicate the robot's location, a count of the number of visits to that grid square is also incremented. Subsequent visits to the same grid square result in a different color of circle being displayed. The visit count to the grid square is added to the color object's R/G/B code to create different shades of blue-green, the darker the color equates to the more visits to a grid square.

At the end of the first run, **robot.py** adds a dummy location [9999, 9999] to the queue, this indicates to the **MazeVisualizer** that Run 1 has completed and the color of the circles used to indicate the robots path is to be changed. This allows the robots movements in Run 2 to be easily contrasted with those of Run 1.

Pressing the escape key when the **MazeVisualizer** window is in focus causes the thread to terminate, this has no affect on the running of **robot.py**.

Metrics

The scoring metric is implemented in the supplied code (**tester.py**), my code does however keep track of the number of iterations made during each run.

State Visits

For the purposes of monitoring robot behaviour and debugging, the Visit object in the **ReinforcementLearning** class is used to count the number of times the robot visits each [state, action] pair. At the end of the first run, any [state, action] pair that has had 3 or more visits is displayed.

Generally during testing if the robot was visiting a [state, action] pair 3 or more times indicated a potential problem ie. the robot may have gotten itself into a loop, visiting the same few states repeatedly or the robot was not exploring enough unvisited states. The **MazeVisualizer** was able to provide additional evidence as to what the robot was doing. Either way some parameter tuning was required.

Function: calculate_reward

The reward function is to provide feedback to the robot for successful or unsuccessful actions in the form of a numeric reward, see **calculate_reward** in **utilities.py**.

Dead-Ends

If the robot has reaches a dead-end, ie. its sensors register [0, 0, 0], the robot receives a negative reward of -25.0.

The Goal

If the robot has reached the goal, then a reward of +100.0 is given.

General Movement

Compute the Manhattan distance between the robots current location and the goal location, then express that distance as a proportion of the distance between the start location and the goal location. If the action to be taken will lead to the robot moving then award a positive reward. The reward is calculated using the exponential of the distance ratio ie. as the robot gets closer to the goal the reward increases exponentially, likewise as it moves away from the goal the reward decreases exponentially. If the action to be taken will result in the robot being unable to move then a negative reward of -50.0 is awarded .

Function: determine_movement_rotation

This function sets the default movement amount to one step at a time. If the reward (from calculate_reward) was positive set the rotation angle based on the action selected eg. Rotation is set to -90 if action is "left". If the reward indicates a dead-end, then set movement to zero, ie. the robot cannot move "left", "forward" or "right", the rotation angle is then set randomly, ie. either -90 or 90 and set action to the correct corresponding value ie. either "left" or "right". Otherwise, the robot will stay put, both movement and rotation are set to zero.

Function: future_sight

During Run 1 when the robot arrives at a grid square, function **future_sight** is called. This function takes each non-zero sensor reading and uses the information to project the robot to possible future states. Using the co-ordinates of a possible future location the distance to the goal is calculated. If the future location is within a threshold of the goal, then future_sight will influence the selection of the next action to be selected. If the future location places the robot outside the threshold then future_sight has no influence on the next action selected for this iteration.

The purpose of this function is to direct the robot to the goal when it gets very close to the goal. This functionality was added quite late during development to help alleviate the frustration of observing the robot passing by the goal opening and not turning in.

In the final project deliverable I have left this function enabled, however I am not sure this is a valid thing to do. The main impact on the final solution if this function is disabled is that the robot will tend to perform additional exploration steps.

Refinement

In this section, you will need to discuss the process of improvement you made upon the algorithms and techniques you used in your implementation. For example, adjusting parameters for certain models to acquire improved solutions would fall under the refinement category. Your initial and final solutions should be reported, as well as any significant intermediate results as necessary. Questions to ask yourself when writing this section:

•Has an initial solution been found and clearly reported?
•Is the process of improvement clearly documented, such as what techniques were used?
•Are intermediate and final solutions clearly reported as the process is improved?

The process of improving upon the algorithms and techniques used is clearly documented. Both the initial and final solutions are reported, along with intermediate solutions, if necessary.

Q-Learning, Double Q-Learning, Prioritized Sweeping

As discussed previously in the “Implementation” section, I started off using the basic Q-Learning algorithm for my initial testing by having the robot explore and solve the maze in Run 1. During testing it became obvious that no amount of parameter tuning would provide the level of success required given the constraint of the time limit.

Changing the learning algorithm to Double Q-Learning made a significant improvement but was still not as reliable as one would of liked. It wasn't until I switched to the Prioritized Sweeping algorithm that I became more confident that I had found an algorithm that could consistently explore the maze and find the goal.

Using the maze visualizer highlighted some opportunities to further refine the solution via parameter tuning, especially after viewing the robots progress on each of the 4 mazes. Generally parameter tuning consisted of changing the alpha parameter, changes to which seemed to make the most difference (compared to other parameter changes), and running trials with each of the 4 mazes. The conclusion from these resulted in the slightly different values for the alpha parameter in the final solution used by each of the different maze sizes. I tried hard to keep the remaining parameters the same as I want the learner to generalize well so unknown mazes can be solved without the need of further parameter tuning.

A* Search

The A* search algorithm is straight forward to implement and doesn't offer much opportunity for refinement as it has a very specific function. However during the course of my initial testing I found that when A* search was running prior to Run 2 it was taking an extremely long time to run, tens of minutes, compared to a few seconds now.

I found that the piece of code that finds adjacent nodes was searching the entire state space for each node that was examined. By refining this process by pre-building a structure that holds all adjacent nodes for each known state prior to starting the A* search process made a significant performance improvement.

Results

Model Evaluation and Validation

In this section, the final model and any supporting qualities should be evaluated in detail. It should be clear how the final model was derived and why this model was chosen. In addition, some type of analysis should be used to validate the robustness of this model and its solution, such as manipulating the input data or environment to see how the model's solution is affected (this is called *sensitivity analysis*). Questions to ask yourself when writing this section:

- ***Is the final model reasonable and aligning with solution expectations? Are the final parameters of the model appropriate?***
- ***Has the final model been tested with various inputs to evaluate whether the model generalizes well to unseen data?***
- ***Is the model robust enough for the problem? Do small perturbations (changes) in training data or the input space greatly affect the results?***
- ***Can results found from the model be trusted?***

The final model's qualities — such as parameters — are evaluated in detail. Some type of analysis is used to validate the robustness of the model's solution.

Final Model Qualities

The final model or solution is the combination of the two algorithm's documented above, specifically Prioritized Sweeping and A* search. The prioritized sweeping algorithm uses the following parameters that could be changed or tuned during the course of testing, they are specified in **robot.py**:

alpha

Alpha, the learning rate, set between 0.0 and 1.0. A setting of 0.0 means that Q-values are never updated hence nothing will be learned. Setting a high value > 0.9 means that learning can occur quickly.

gamma

Gamma, the discount factor, set between 0.0 and 1.0. This models the fact that future rewards are worth less than immediate rewards. If equal 1.0, the agent will value future rewards just as much as current rewards. A value of 0.0 will cause the agent to only value immediate rewards.

epsilon

Epsilon, set between 0.0 and 1.0, is used in the function **chooseActionUsingEpsilonGreedy** (see **ReinforcementLearning.py**). A setting of 0.0 means that actions will be chosen using the **chooseActionUsingMaximumQValue** function (see **ReinforcementLearning.py**). A setting of 1.0 means that all actions will be selected at random from the list of all possible actions. For values other than 0.0 and 1.0 if a random number between 0.0 and 1.0 is less than the value of Epsilon, then the action will be chosen at random otherwise it will be chosen using the **chooseActionUsingMaximumQValue** function.

temperature

The temperature, set between 0.0 and 1.0, is used in function **chooseActionUsingSoftMax** (see **ReinforcementLearning.py**). The temperature (or T) specifies how random values should be chosen. When T is high, the actions are chosen in almost equal amounts. As the temperature is reduced, the higher value actions are more likely to be chosen, and in the limit as T → 0, the best action is always chosen.

theta

Theta, set between 0.0 and 1.0, is used in the function **prioritizedSweeping** (see **ReinforcementLearning.py**). It is used to control which state-action pairs are added to the prioritized queue, if the state-action pairs priority is greater than theta then the state-action pair is added to the prioritized queue.

decay

The decay parameter can take a value of either True or False. This determines whether or not the learning rate (alpha) will be decayed at each iteration or not. If set True, the learning rate will be decayed using the following formula:

1.0 / 1.0 + counter

where, the **counter** is the number of iterations in the trial so far.

planningSteps

The planningSteps parameter, a number greater than 0, is used in function **prioritizedSweeping** (see **ReinforcementLearning.py**) to specify how many planning steps to perform.

A* Search

Strictly speaking there are no specific parameters that are passed to the A* search algorithm. There is however some chance of affecting the behaviour of the algorithm, specifically this is done by manipulating component costs of the cost function, f = g + h. Namely in the choices for:

- The minimum cost for the robot to move one node from an adjacent node. This is set by the **minimumStepCost** variable in class **Astar** (see **AstarSearch.py**), it is currently set to 20.0.
- The choice of the heuristic, h and how it is calculated. Please refer to the "Algorithms and Techniques" section for details of the chosen heuristic.

Final Model Parameters

The table below details the values for the parameters chosen in the final solution.

Parameter	12x12 Maze	14x14 Maze	16x16 Maze
alpha	0.98	0.90	0.65
gamma	0.25	0.25	0.25
decay	False	False	False
epsilon	0.20	0.10	0.20
temperature	0.15	0.15	0.15
theta	0.01	0.01	0.01
PlanningSteps	50	50	50

I used the chosen values for the alpha and gamma parameters to have the Q-Learner learn quickly and place more value on current rewards.

Justification

In this section, your model's final solution and its results should be compared to the benchmark you established earlier in the project using some type of statistical analysis. You should also justify whether these results and the solution are significant enough to have solved the problem posed in the project. Questions to ask yourself when writing this section:

•Are the final results found stronger than the benchmark result reported earlier?
•Have you thoroughly analyzed and discussed the final solution?
•Is the final solution significant enough to have solved the problem?

The final results are compared to the benchmark result or threshold with some type of statistical

analysis. Justification is made as to whether the final model and solution is significant enough to have adequately solved the problem.

Analysis

The following tables summarize the results of 20x runs against the 12x12, 14x14 and 16x16 supplied mazes and my 12x12 maze. They show the following:

- Number Steps Run 1 – the number of iterations the robot took to reach the maze goal in Run 1
- Number Steps Run 2 – the number of iterations the robot took to reach the maze goal in Run 2
- Score – the Overall Score achieved

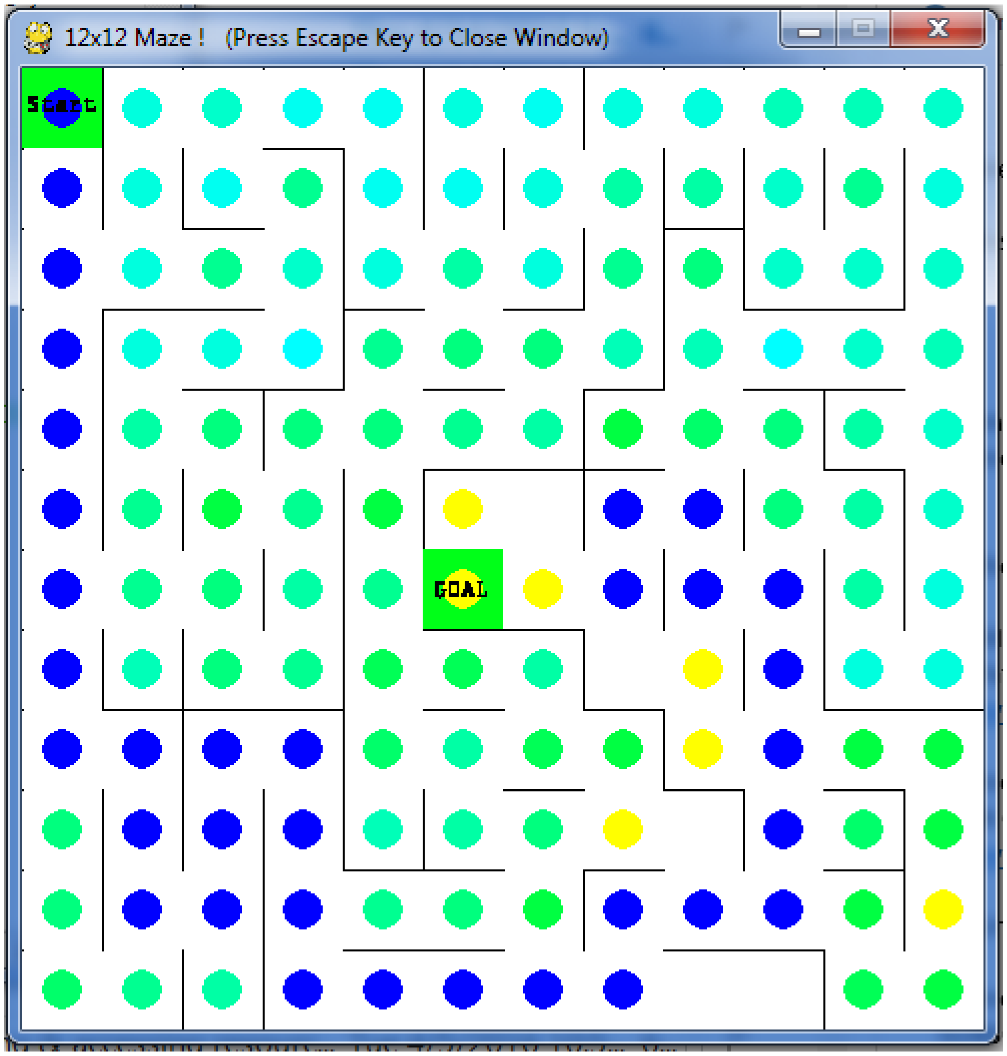
12x12 Maze

Number Steps Run 1	Number Steps Run 2	Score
269	39.0	47.967
808	39.0	65.933
392	52.0	65.067
244	67.0	75.133
392	38.0	51.067
293	68.0	77.767
100	36.0	39.333
86	45.0	47.867
252	34.0	42.400
355	59.0	70.833
119	39.0	42.967
101	34.0	37.367
827	30.0	57.567
431	59.0	73.367
285	56.0	65.500
148	62.0	66.933
626	41.0	61.867
343	48.0	59.433
209	44.0	50.967
593	50.0	69.767

Mean of 20x values	Mean of 20x values	Mean of 20x values
343.65 / 30 = 11.455	47	58.455

Solved 12x12 Maze

The following figure shows an example of a solved 12x12 maze.

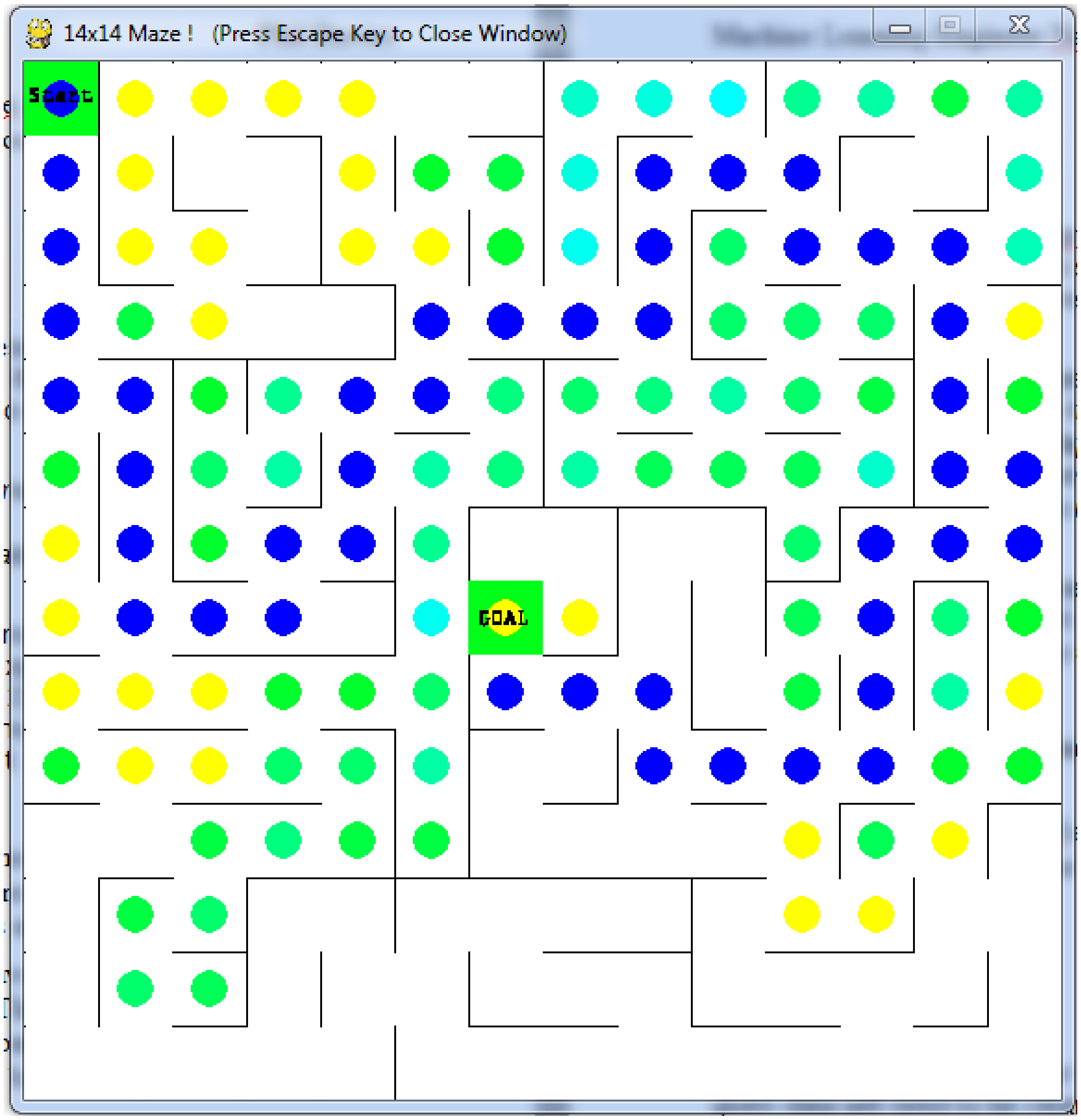


14x14 Maze

Number Steps Run 1	Number Steps Run 2	Score
797	77.0	103.567
318	62.0	72.600
521	49.0	66.367
875	66.0	95.167
884	56.0	85.467
684	63.0	85.800
472	73.0	88.733
563	51.0	69.767
922	54.0	84.733
549	77.0	95.300
170	60.0	65.667
240	61.0	69.000
415	81.0	94.833
666	60.0	82.200
365	65.0	77.167
753	58.0	83.100
843	66.0	94.100
370	71.0	83.333
302	49.0	59.067
754	53.0	78.133
Mean of 20x values	**Mean of 20x values**	**Mean of 20x values**
573.15 / 30 = 19.105	62.6	81.705

Solved 14x14 Maze

The following figure shows an example of a solved 14x14 maze.

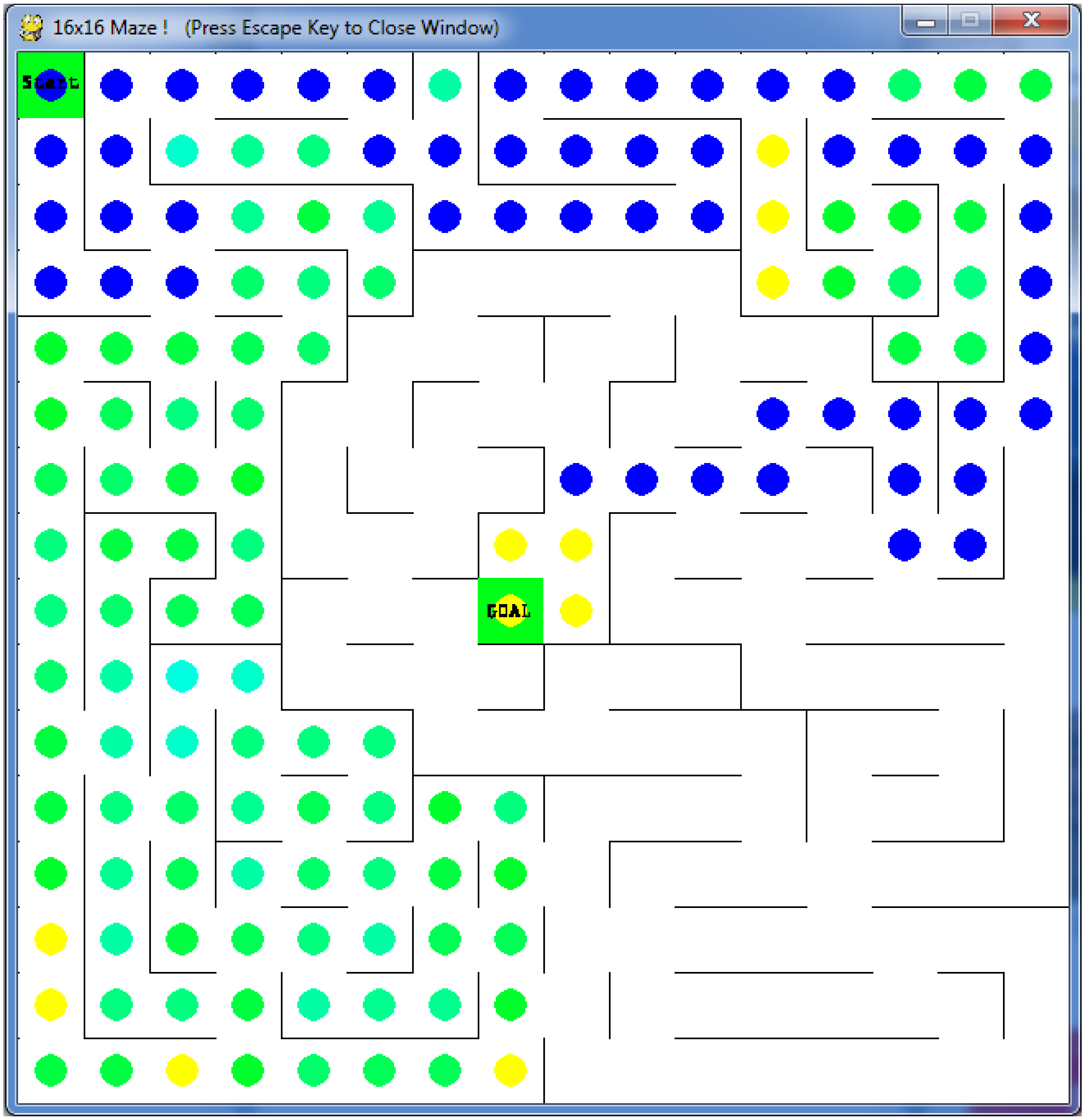


16x16 Maze

Number Steps Run 1	Number Steps Run 2	Score
486	59	75.200
443	69	83.767
181	54	60.033
151	55	60.333
614	57	77.467
554	65	83.467
649	59	80.633
349	63	74.633
667	59	81.233
411	63	76.700
506	57	73.867
478	67	82.933
465	59	74.500
704	61	84.467
496	53	69.533
456	69	84.200
482	69	85.067
518	63	80.267
319	57	67.633
356	67	78.867
Mean of 20x values	**Mean of 20x values**	**Mean of 20x values**
464.25 / 30 = 15.475	61.25	76.725

Solved 16x16 Maze

The following figure shows an example of a solved 16x16 maze.



My 12x12 Maze

Number Steps Run 1	Number Steps Run 2	Score
293	50.0	59.767
394	30.0	43.133
328	74.0	84.933
328	49.0	59.933
215	37.0	44.167
139	56.0	60.633
147	44.0	48.900
80	37.0	39.667
311	53.0	63.367
232	55.0	62.733
495	54.0	70.500
464	36.0	51.467
469	46.0	61.633
456	79.0	94.200
429	82.0	96.300
278	42.0	51.267
164	69.0	74.467
715	32.0	55.833
693	42.0	65.100
778	32.0	57.933
Mean of 20x values	**Mean of 20x values**	**Mean of 20x values**
370.40 / 30 = 12.347	50.95	63.297

My 12x12 Maze Solved

For a diagram of a possible solution to my 12x12 maze please see the Conclusion section below.

Overall Summary

The following table provides a summary for each of the four mazes and a comparison to the benchmark.

Maze	Run 1 Score	Run 1 Benchmark	Run 2 Score	Run 2 Benchmark	Total Score	Total Score Benchmark
12x12	11.455	30	47.00	31	58.455	61
14x14	19.105	30	62.60	43	81.705	73
16x16	15.475	30	61.25	51	76.725	81
My 12x12	12.347	30	50.95	31	63.297	61

From looking at the table, the most obvious observation is that the “Run 1 Benchmark” values were overstated in that the achieved Run 1 scores are substantially less than the benchmark. This was not entirely unexpected as the benchmark values assumed that the robot took 900 steps each time during Run 1. This was never going to be the case as I let the robot complete Run 1 once the goal was reached, rather than keep exploring until 900 steps/iterations had been made. My results show that on average, the goal was located in much less than half of the benchmark, the exception being the 14x14 maze which took approximately 64% (19.105/30) of the benchmark time.

Looking at “Run 2 Score” values we can see that all run 2 scores exceeded the “Run 2 Benchmark” scores and by significant margins. This was due to the fact that the robot was locating the goal relatively quickly (see Run 1 discussion previous) and thus a lot of states remain unexplored, some of those unexplored states may well have lead to a lower run 2 score.

Overall in two of the mazes, the 12x12 and 16x16, the average score for each was better than the respective benchmark.

In the case of my 12x12 maze, the average score achieved was only slightly worse than the benchmark, by less than 4%. This was expected as I had purposely tried to make my 12x12 maze more difficult than the supplied 12x12. A higher level of difficulty was achieved for my 12x12, as the average number of steps taken in both runs 1 and 2 compared to those of the supplied 12x12 maze were larger, specifically for Run 1, 370.40 steps and for Run 2, 50.95 steps for my maze, compared to 343.65 steps and 47.0 steps for Runs 1 and 2 for the supplied 12x12 maze.

When looking at the 14x14 maze, the average score achieved exceeded the benchmark by 11.9% (81.705 compared to the 73 benchmark). Although disappointing it was not unexpected as throughout my testing I felt that the 14x14 maze was the most difficult of the mazes to solve as there are so few entrances into the final section of the maze that contains the goal.

Summing up, looking at my overall results the final model and solution have solved the problem of having a robot explore a maze, locate the goal and then find an optimum route. However there is still plenty of room for improvement.

Conclusion

Free-Form Visualization

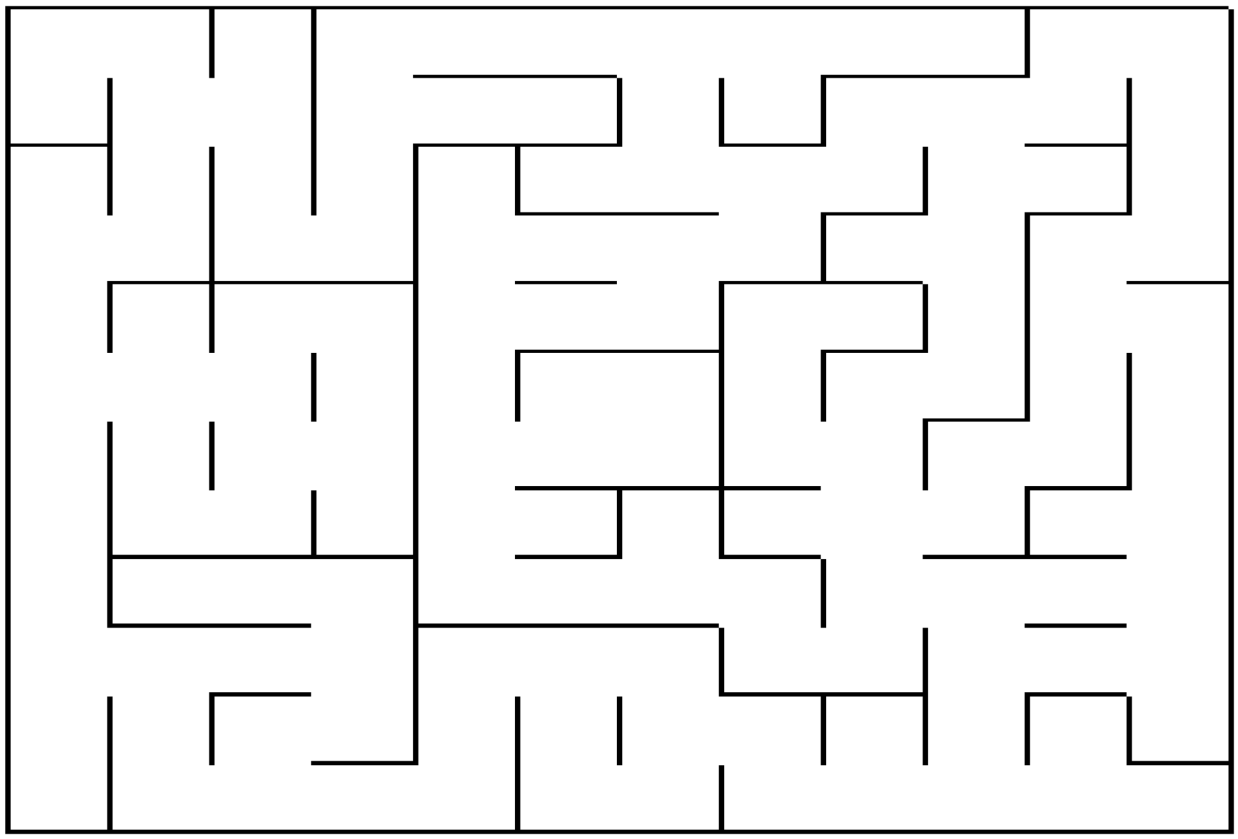
In this section, you will need to provide some form of visualization that emphasizes an important quality about the project. It is much more free-form, but should reasonably support a significant result characteristic about the problem that you want to discuss. Questions to ask yourself when writing this section:

- ***Have you visualized a relevant or important quality about the problem, dataset, input data, or results?***
- ***Is the visualization thoroughly analyzed and discussed?***
- ***If a plot is provided, are the axes, title, and datum clearly defined?***

Free-Form Visualization: Use this section to come up with your own maze. Your maze should have the same dimensions (12x12, 14x14, or 16x16) and have the goal and starting positions in the same locations as the three example mazes (you can use `test_maze_01.txt` as a template). Try to make a design that you feel may either reflect the robustness of your robot's algorithm, or amplify a potential issue with the approach you used in your robot implementation. Provide a small discussion of the maze as well.

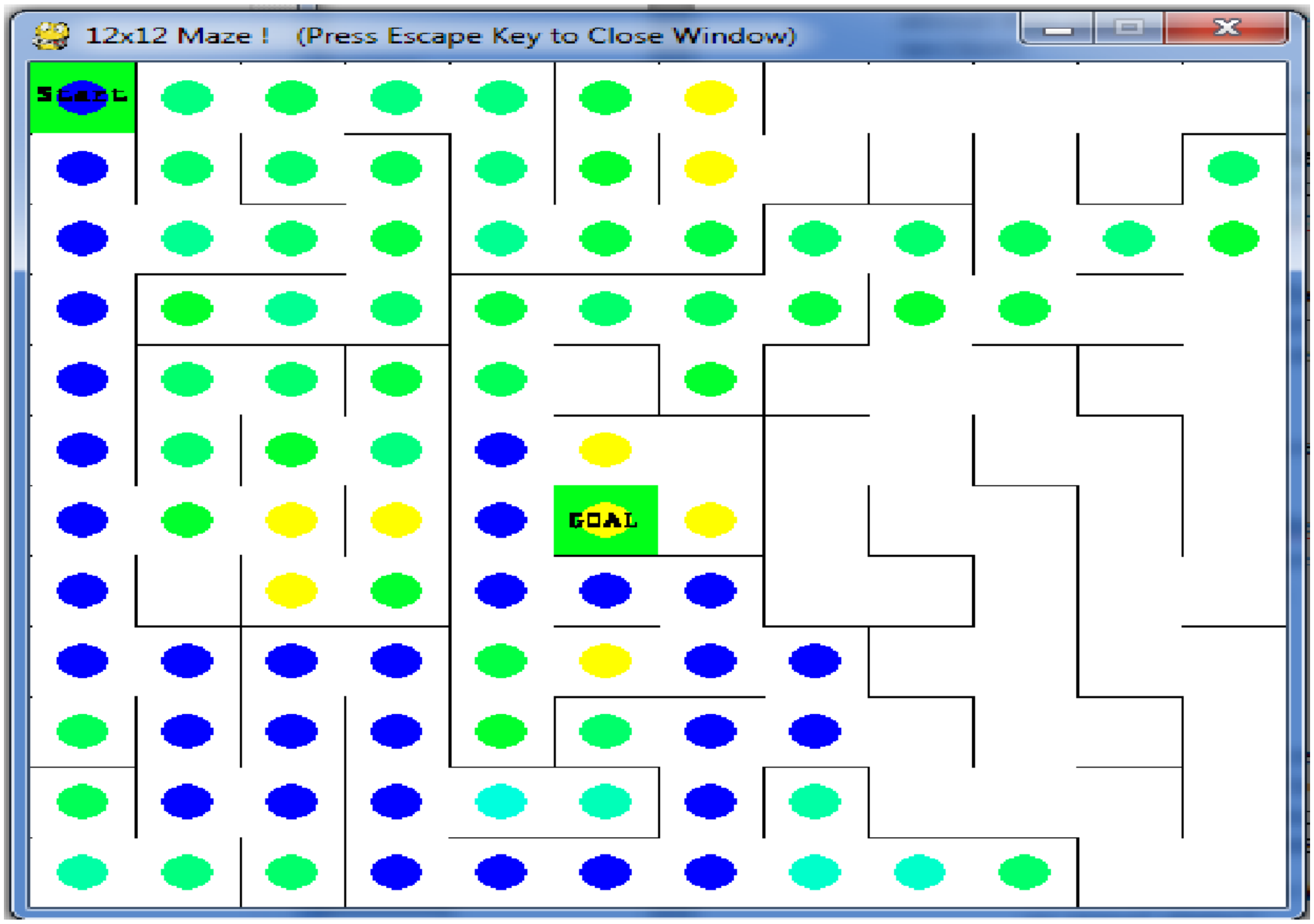
12x12 Custom Maze

Using the supplied maze definition file **test_maze_01.txt** (12x12) I created maze definition file **test_maze_04.txt** (12x12). My plan was to make the 12x12 maze much more difficult by having more areas that allowed movement but ultimately lead no where, more turns required to get to the goal, more dead ends and less paths that lead to the goal. This maze is pictured below:


12x12 Custom Maze

In order for the robot to reach the goal, it must approach the right hand side of the maze from either the top or bottom of the maze. It must then try and find a path back into the center where the entrance way to the goal is located.

Below is a picture showing once possible solution that the robot has discovered.



This solution happens to be the second most optimal, where the robot finds the goal via the top of the maze. Note that the MazeVisualizer displays the maze with the start location [0, 0] at the top left hand corner.

Reflection

In this section, you will summarize the entire end-to-end problem solution and discuss one or two particular aspects of the project you found interesting or difficult. You are expected to reflect on the project as a whole to show that you have a firm understanding of the entire process employed in your work. Questions to ask yourself when writing this section:

- *Have you thoroughly summarized the entire process you used for this project?*
- *Were there any interesting aspects of the project?*
- *Were there any difficult aspects of the project?*
- *Does the final model and solution fit your expectations for the problem, and should it be used in a general setting to solve these types of problems?*

Student adequately summarizes the end-to-end problem solution and discusses one or two particular aspects of the project they found interesting or difficult.

To recap, the original problem was to have a virtual robot plot a course from the start of the maze to the goal. The robot is allowed two runs, the first, an exploratory run and the second, a run to move from the start location to the goal in the least amount of moves. Together both runs must be completed within the 1000 step time limit.

The solution I chose was to use a Q-Learner that exploits the DYNA architecture, namely Prioritized Sweeping, to facilitate the exploration and location of the goal required in Run 1, while still leaving enough time for the second run. For Run 2, A* search was used to determine the optimal route to the maze goal using the information learned during Run 1.

The most challenging aspect of this project was to tune the Q-Learner for each of the mazes so that the goal could be found in a reasonable amount of time, while ensuring the model was not over-tuned as the unseen mazes will have to be solved with the same parameters. Trying to keep the Q-Learner as generalized as possible also had the side affect of the robot not always exploring as much as it could before finding the goal. The knock-on affect is seen in Run 2, where A* search can only find the optimal path from the states that have been explored during Run 1. If the robot had not explored the states that make up the true optimal path, then A* search will not be able to find the true optimal path. The found path will only be optimal in terms of the explored states.

Another difficulty or frustration I encountered during the course of the project was not really knowing how far I could go in the action selection functions in terms of trying to influence what action is ultimately selected (see my discussion on function **future_sight** above). Obviously the algorithm chosen should make the choice, but its hard to sit back and watch it make bad choices sometimes, especially when the robot approaches the entrance to the goal box and turns away rather than going forward for example.

One aspect of the project that was particularly interesting was being able to observe the differences in the robots behaviour after parameter and reward tuning. At times a small change in a parameter can make a striking difference in the robots behaviour, for example, going from getting into a tight loop to finding the goal.

Another aspect of the project that I found both interesting and challenging was interpreting algorithms found in the literature and from that interpretation writing runnable code that did what it was meant to.

Improvement

In this section, you will need to provide discussion as to how one aspect of the implementation you designed could be improved. As an example, consider ways your implementation can be made more general, and what would need to be modified. You do not need to make this improvement, but the potential solutions resulting from these changes are considered and compared/contrasted to your current solution. Questions to ask yourself when writing this section:

•Are there further improvements that could be made on the algorithms or techniques you used in this project?
•Were there algorithms or techniques you researched that you did not know how to implement, but would consider using if you knew how?
•If you used your final solution as the new benchmark, do you think an even better solution exists?

Improvement: Consider if the scenario took place in a continuous domain. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units. What modifications might be necessary to your robot's code to handle the added complexity? Are there types of mazes in the continuous domain that could not be solved in the discrete domain? If you have ideas for other extensions to the current project, describe and discuss them here.

Improving Existing Implementation

The most significant improvement that I think can be made to my implementation is to either improve the existing algorithm used in run 1 or find an alternative algorithm.

The improvement to the algorithm used in run 1, ie. prioritized sweeping, that is required is to have the robot perform a more efficient exploration. This would involve reducing the amount of re-visits to states already visited and placing more emphasis on visiting as yet unexplored states. Failure to explore enough is a hazard of DYNA-Q architecture, in that it can be overly aggressive in exploiting the observations made in planning, which in turn results in DYNA-Q forfeiting the opportunity to learn the optimal policy.

The improvements required all center around how the next action is selected and therefore what next action is selected, some ideas on improvements include:

- Update the action selection function to use additional information such as visit counts and sensor information to augment the decision coming from the Q-Matrix.
- Implement an "exploration bonus" similar to that used in DYNA-Q+ where an extra reward is added to the reward for transitions caused by state-action pairs related to how long ago they were tried. The longer a state-action pair remains unvisited the greater the reward for visiting.
- Another idea is to replace the Q-Matrix with a supervised learner, currently the data samples obtained through agent-system interaction are used once to update the value function and then are discarded. Using a supervised learner ie. a neural network, can overcome this problem by using the same data multiple times during training. This should lead to a more robust action selection.

All these improvements should lead to more of the state space being explored and thus increase the likelihood that the optimal path to the goal will be traversed during exploration. This will then give A* search every chance to find the true optimal path.

Continuous Domain Scenario

If this maze and virtual robot scenario were shifted to a continuous domain, a number of code changes to my solution would be required.

For instance the continuous data coming from the sensors would have to discretized in some manner inorder to continue using the Q-Learning algorithm implemented. Alternatively an algorithm better suited to handling a continuous domain could be used instead. One such alternative is the algorithm Abstraction Selection.

In the continuous world I make the assumption that the virtual robot may no longer be able to move one grid square per iteration, it might require multiple iterations to move one square to the next. If this is the case then additional logic would be required in-order to keep an accurate location of the robot.

Are there types of mazes in the continuous domain that could not be solved in the discrete domain?

In theory all continuous maze problems can be solved in the discrete domain, as the continuous maze state space can be discretized to a level of resolution that would allow it to be solved. However, in practice, for highly dimensional complex continuous domains this may not be practical due to the large state-action space required and the resulting storage, time and computational resources needed to compute and maintain the optimal policy.

Ideas for Extending the Current Project

The idea I had to extend my project was to capture the image of the virtual robot moving about in the maze visualizer window and train a Deep Q-Learner from the images, this would be a training phase. From this the Deep Q-Learner will learn how to control/direct the robot during run 1. Furthermore, it would be interesting to see if a Deep Q-Learner could learn the A* search algorithm by having it observe the second run performed by A* search.

Files Included

The following files are included:

- robot.py (modified supplied file)
- ReinforcementLearning.py
- MazeVisualizer.py
- AStarSearch.py
- utilities.py
- maze.py (Udacity supplied file)
- showmaze.py (Udacity supplied file)

- tester.py (Udacity supplied file)
- test_maze_01.txt (Udacity supplied file)
- test_maze_02.txt (Udacity supplied file)
- test_maze_03.txt (Udacity supplied file)
- test_maze_04.txt (my maze)
- README.txt
- P5 Capstone Project.pdf (this document)

Literature Cited

The following are the books and papers read during the course of this project.

- R.S. Sutton, A.G. Barto: Reinforcement Learning: An Introduction MIT Press, 1998.
- M. Stolle: Finding and Transferring Policies Using Stored Behaviors, 2008.
- Hybrid adaptive heuristic critic architectures for learning in mazes with continuous search spaces. PPSN 111, pp 482-491.