Capstone Project

Sergey Kapustin
October 13th, 2016

Machine Learning Engineer Nanodegree
Project: Plot and Navigate a Virtual Maze

Definition

Project Overview

In this project a robot mouse explores a maze with the goal of finding the most optimal path to maze center. The project takes inspiration from Micromouse competitions.

A maze is *n* x *n* grid of squares where *n* is even number. The maze is fully enclosed by walls. Its interior has passageways and dead-ends. At least one of the passageways leads to maze center, which is a 2x2 square.

There are two stages that a virtual robot mouse operates in. In the first stage the robot is free to explore the maze and map out its interior so as to discover the best path to maze center. Main characteristic of the best path is the minimum number moves that is required to reach the center.
In the second stage, the robot will attempt to reach the center in shortest time possible using the knowledge it acquired during the first, exploratory stage. The number of moves for both stages is limited to one thousand.

To achieve best results, robot should learn to avoid dead-ends and loops. It should learn to take advantage of straight paths because on a straight path it can move in larger strides, up to three cells at a time, which would result in faster run time.

In this project, the specifications of robot and three mazes are provided.

Problem Statement

The main goal for this challenge is to program the robot so that it finds most optimal route to the maze center within a given time frame. The time frame is represented by a maximum number of 1000 steps. Both, the number of steps taken during exploration and final run to the center count towards the time limit.

The project can be broken down into several subtasks:

- Analyze the data and environment. Ideally, the result would provide visualizations for easier interpretation and generation of ideas

- Determine which techniques would be useful in achieving the project's goal. Adjust the approach as new information is discovered
- Implement the logic using predetermined techniques. In this project, I plan to use A-star algorithm in exploration and final run stages. The algorithm is suited well to solving the problems of short-path discovery
- Evaluate the results, rinse and repeat previous steps as necessary
- Document the findings and overall process

Metrics

Robot's performance is evaluated based on the number of moves it requires to reach the maze center plus one-thirties of the number of moves taken during the discovery stage.

Analysis

Data Exploration

The robot assumes the following environment and operation characteristic.

It will navigate a square maze of dimension *n x n.* A center of the maze, its target, is a square of four cells. Robot's initial position will be a cell with the location (*n* - 1, 0) where the first element is *y* coordinate, and the second is *x* coordinate in a two-dimensional Euclidian space.

Maze specification is defined in a text file. The first line of the file contains a single number which describes the number of cells along each dimension *n*. For example, number 12 specifies that we deal with a 12 x 12 maze.
There are *n* subsequent lines, with each line containing *n* comma-delimited list of numbers. Each number describes which edges of a single cell are open (passageway) and which are closed (wall). The number is represented by a four-bit value with each bit describing top, right, bottom and left side of the cell; registers 1, 2, 4 and 8 respectively. If a bit is set to 0 then that side is closed, if a bit is set to 1, the side is open for travel. For example, number 9 describes a cell where top side is open, right and bottom side is closed, and left side is open: 9 = 1 * 1 + 0 * 2 + 0 * 4 + 1 * 8.
The second line in the file corresponds with the leftmost column and its first number describes the starting cell. For example, in a 12x12 maze, the starting cell is located at position (11, 0).

At every step, a robot will be given an obstacle sensor. The sensor contains three distances expressed as a number of open-for-travel cells to robot's left, center and right direction. A sensor direction with zero cells represents a wall. Sensor with zeroes in all directions represents a dead-end.

The environment expects a robot to provide a move instruction in the form of a tuple, where the first element is a turn in degrees, the second is a number of steps to move. When deciding on the next move, the robot may choose to turn clockwise or counterclockwise to a maximum of ninety degrees. Zero-degree rotation (no rotation) is also allowed and means to keep the current heading. Any other value than negative ninety, zero, or positive ninety degrees is ignored by the environment.

The robot can choose to move in one of four directions, with the limit of three steps at a time. If its next move would result in hitting the wall, the robot will be left where it was before it attempted the move. It can move backwards by returning a negative number of steps to the environment. Every turn and movement is perfect as long as the constraints described above are not violated, i.e., walls, turns, maximum steps.

To end the run, the robot can provide reset instruction via a tuple (‘Reset’, ‘Reset’) instead of (degree, steps) instruction. When the robot returns resets, the environment will place it at the initial position (origin) during exploration stage, otherwise the environment will terminate the experiment.

Exploratory Visualization

Figure 1 shows an example of a 12x12 maze. The first row and column of numbers represent *x* and *y* coordinates. Red bars represent walls. The origin is at location (11,0) and marked with a green dot. Maze center is marked with blue dots.

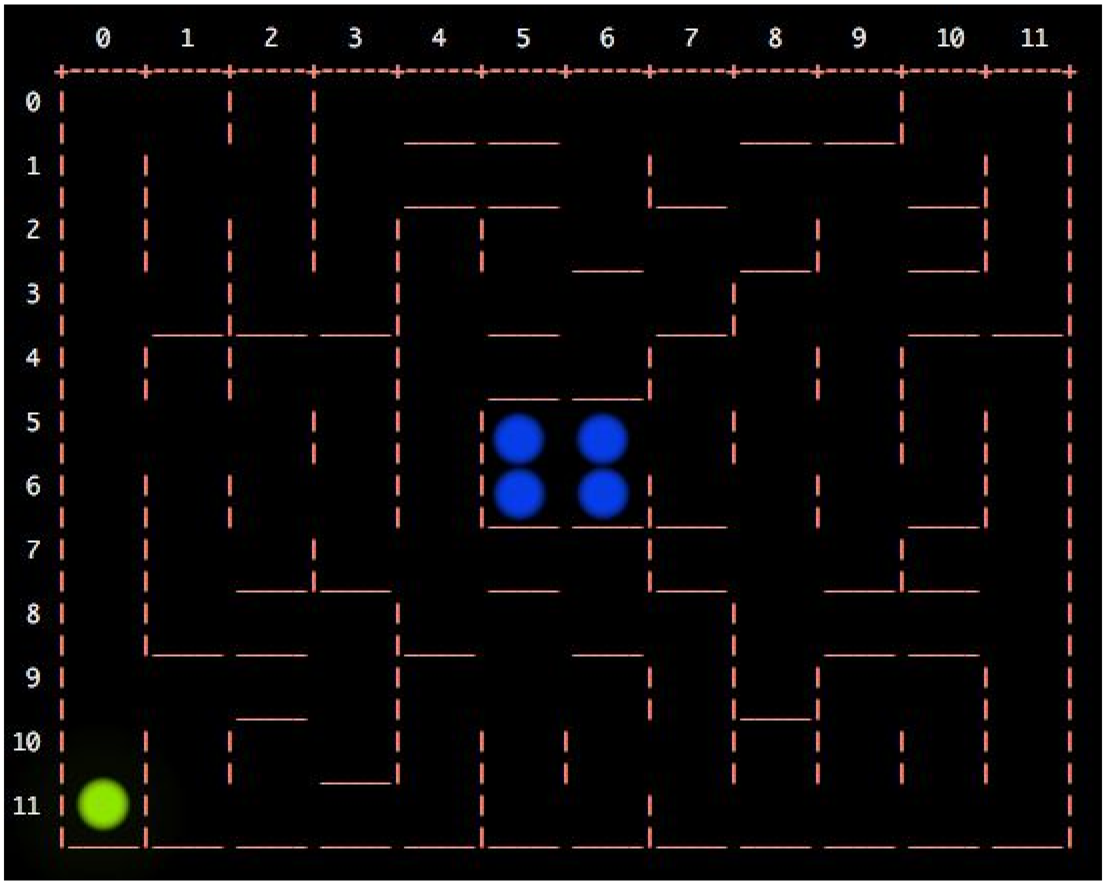


Figure 1. Maze, paths

Some of the challenging features of a maze that a robot has to deal with are dead-ends and loops. In Figure 2 shows some of the dead-ends and are circled with green ellipsis, the loops are circled with a cyan.

When faced with a dead-end, a robot has to plan a path back to a cell that allows to continue exploration of other locations that may lead towards the goal. Moreover, the robot should mark the complete path leading to a dead-end as a non-viable so that the path is eliminated when determining the optimal route.

Loops will cause the robot to wander in circles unless they are accounted for with some mechanism of detection. Once the robot determines the loop, it must find a nearby cell that will not only allow to break out, but will also be closer to the goal.

A favorable feature is a straight path. This will allow the robot to travel further using fewer number of steps up to a maximum of three at a time. In the figure, straight paths are circled with magenta.

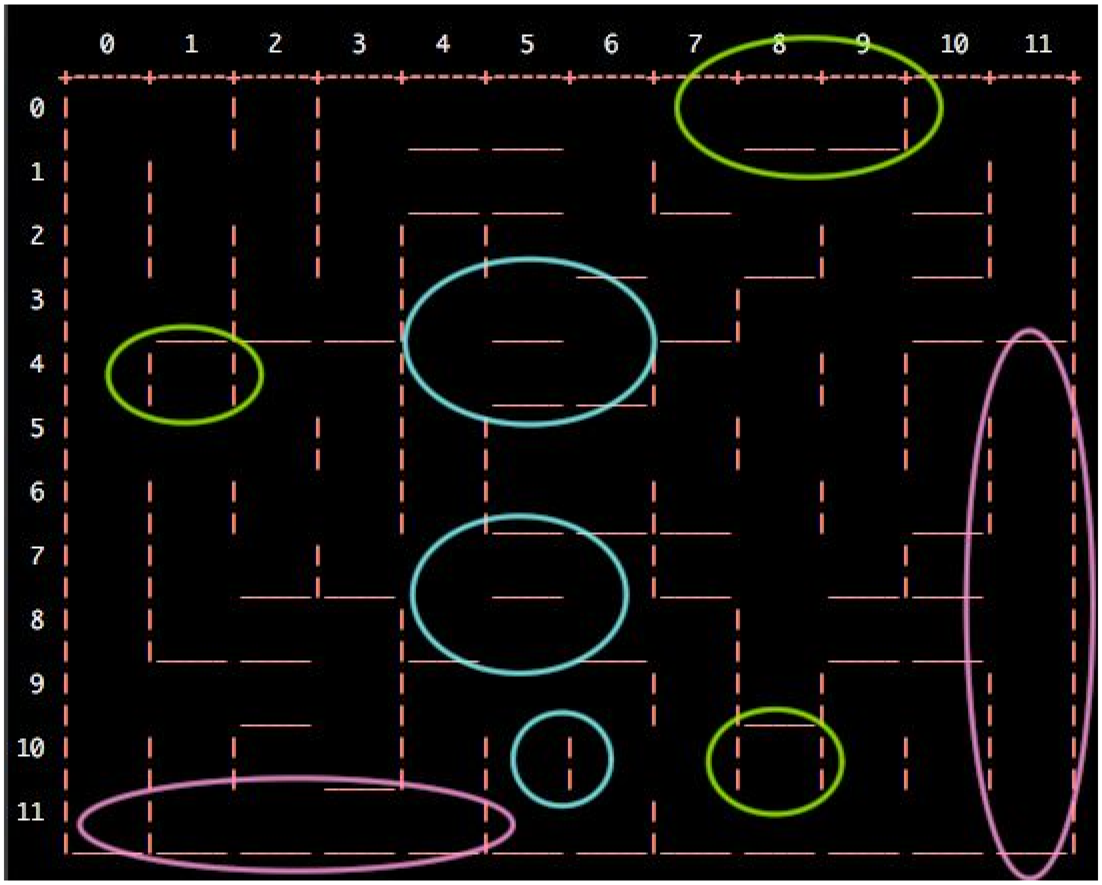


Figure 2. Maze challenges

Algorithms and Techniques

Since the robot is required to traverse a graph and find a shortest path to the center, an algorithm that aligns well with these requirements is A*.

With A*, each cell of the maze will describe its cost on the path from origin to center using parameters: g-cost, h-cost and f-cost.

- g-cost is the number of steps that robot took from origin to the current cell
- h-cost is algorithm's heuristic. In this project, Manhattan distance is used, which represents the sum of differences between *x* and *y* coordinates from a current cell to the goal
- f-cost is a sum of g- and h- costs. It represents the total cost incurred if the robot uses this cell on the way from start to finish. When estimating optimal path, the robot selects cells that have the minimal f-cost(s)

To avoid loops, the robot will use a cell's parameter that contains a count of visits. If the number of visits is greater than 0, the robot will select other unvisited cell from a list of candidates.

The chosen cell will have the smallest value, which is defined as sum of

A. Distance between current and some unvisited cell in the number of steps
B. The unvisited cell's h-cost, which is Manhattan distance to the goal

In order to calculate the distance, the robot will traverse a tree of visited cells and select the shortest path based on the number of steps required to reach an unvisited cell. This would account for walls and dead-ends on the path between the cells as opposed to Manhattan distance, which just uses right angles for calculation.

For example, assume that the robot ended up in cell (3,4), marked as a red dot in Figure 3 below. This cell is part of the loop comprised of cells: (4,4), (4,5), (4,6), (3,6), (3,5) and (3,4). The unvisited cells, marked with a green dot, are (2,5), (3,7), (7,5) and (8,4). The robot will pick cell (2,5) to visit next because the combined distance and h-cost are lower than any other combination:

- Cost of (2,5): distance + h-cost = 2 + 3 = 5
- Cost of (3,7): distance + h-cost = 3 + 3 = 6
- Cost of (7,5): distance + h-cost = 5 + 1 = 6
- Cost of (8,4): distance + h-cost = 5 + 3 = 8

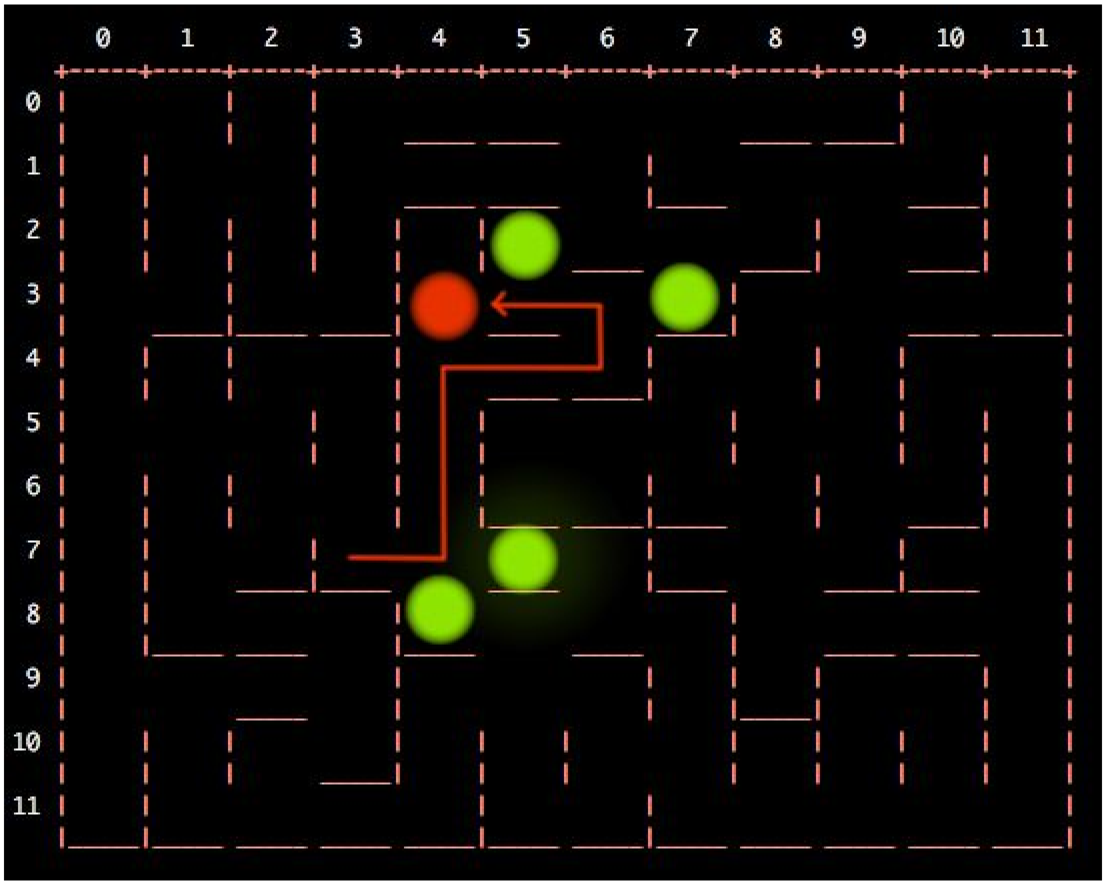


Figure 3. Loops and next unvisited cells

During discovery stage, upon reaching maze center, the robot will reset the goal to the origin, and will continue exploring. Once the robot reaches the origin, two paths are compared and the one with shorter distance is chosen.

Benchmark

As a baseline, this project will use the minimal number of moves that is required to reach the maze center given all of the maze is known. One move is between one and three steps. The goal of this project is to enable the robot to perform not worse than 50% of the baseline.

Methodology

Data Preprocessing

Since the sensor specification and maze designs are provided as part of this project, no data preprocessing is needed.

Implementation

The implementation includes several classes: Defs, Cell, Grid, Robot. The classes are spread between two files: robot.py and core_lib.py. In addition, the package will include script analysis.py that is used to calculate stats, graphically represent the maze, and other useful information.

When executed, a log file “output.log” is created in current working directory. The log file is used for analysis and debugging.

Class Defs (core_lib.py)

The class encapsulates various constants used by other classes. The constants include:

- *NORTH, EAST, SOUTH, WEST* - polar headings
- *LEFT, CENTER, RIGHT* - sensor direction
- *START_CENTER_PHASE* - the flag indicating that the current exploration direction is from origin to maze center
- *CENTER_START_PHASE* - the flag indicating that the current exploration direction is from maze center to origin
- *HEADINGS* - a map used by the robot to determine the next rotation and movement
- *MOVES* - a list containing move instructions depending on current robot’s heading
- *HEADINGSS* - a list of one-letter headings used for logging

Class Cell (core_lib.py)

The class represents a cell in a maze and contains the following data members:

- *loc* - tuple of (*y*, *x*) coordinate within maze

- *parent* - represents the parent cell that discovered the current cell. The parent provides the shortest path to this cell on the way to the goal
- *g_cost* - the number of steps required to reach this cell from the start position
- *f_cost* - the cost of traveling through this cell on the way to goal. It is a combination of *g_cost* and Manhattan distance to the goal
- *viable* - a list of open cells from the perspective of the current cell in four polar directions: north, east, south, west
- *visits* - the number of times this cell was visited
- *deadend* - the number representing a rank within a dead-end path. A cell that has only one way in or out has a rank of maze dimension squared. A parent cell leading to this dead-end cell would have that rank minus 1, and so on until a parent that has at least 3 viable ways in/out. The rank is used to determine if a path starting with a given cell leads to a deadend

Cell class also provides a few functions:

- *__repr__* - used to output cell definition to a log for analysis. In this example, the parameters defined above are separated with commas:
  `Cell((3, 4), (4, 4), 16, 19, [1, 3, 5, 0], 0, 0)`
- *__eq__, __ne__* - used to compare two cells for equality using location tuple
- *is_path_defined* - boolean function that returns false if at least one of the walls is undetermined to have a viable path
- *set_cost* - used to update g_cost and f_cost data members

Class Grid (core_lib.py)

The class represents a maze as a grid of cells. It provides several utility functions that the robot uses to discover and evaluate paths, determine deadend condition, get unvisited cells, calculate distances, and others.

The class defines the following data members:

- *dim* - grid dimension
- *cells* - a mapping of a grid location to a cell instance
- *goals* - the current goal(s). It would contain 4 cells of the center square when the robot travels towards it. The goal is switched to origin when the robot continues discovery in the opposite direction from the center
- *unvisited* - a map containing discovered but unvisited cells

Grid class provides the following functions:

- *__setitem__, __getitem__* - used to access cells using subscript notation
- *__coord__* - convenience function to output a location tuple given Cell object
- *set_goals* - used to set the robot's goals

- *reset* - used to reset the grid state for when a robot wants to keep exploring toward the origin after reaching the center cell. This function clears the list of unvisited cell, and reset parameters *Cell.visits, Cell.parent, Cell.g_cost, Cell.f_cost* of each cell to initial state. This helps a robot to discover new, potentially shorter path
- *distance* - calculates Manhattan distance between two cells
- *distance_to_goal* - uses *distance* function to calculate the closest distance to the goal. Since maze's center is comprised of a 4-cell square, this function provides a convenient way to get the closest one
- *get_unvisited* - returns all unvisited, non-deadend cells
- *set_cost* - calculates g_cost from a starting cell to each cell in the list. If g_cost is less than the one a cell currently has, the function updates the cell with the new value, calculates f_cost and set the cell's parent to the current cell. This mechanism is used by A* algorithm to determine the shortest path to a goal
- *on_visit* - called by a robot every time it visits a cell. This function invokes supporting functions to create/update new neighbouring cells, update cell's viable status, remove the current cell from unvisited map, and add the discovered neighbours to unvisited map
- *neighbours* - returns a list of neighbour cells from all polar headings. The function doesn't return dead-end cells. Depending on function parameters, it may or may not return already visited cells and update neighbour costs. When a robot gets into a loop, it asks for visited cells so as to plan a path to the other unvisited cells which are more than one step away. In this case, it doesn't require to update cell costs, but only wants to escape the loop
- *get_cell* - is called by *neighbours* function and is used to return the existing or create a new cell. The function would update cell's viable status based on the current cell the robot is in. This is required because the sensors would contain only 3 directions relative to robot's heading, left/center/right, but not rear. This would set neighbour's viable cell count in the opposite-to-heading direction to a count of current cell's opposite direction plus 1. The same applies to heading direction, but the count is reduced by 1.
- *build_tree* - recursively builds a path to each unvisited cell. Since the unvisited cells may be more than 1 step away, the function create a sequence of cell locations required to travel from current to the target unvisited cell. The robot then evaluates each path, and selects the most cost-efficient (more details provided in Robot class). The function contains several conditions to terminate the search early, such as whether the target cell can be reached via a shorter path, if it is viable (all walls are defined).
- *build_path_on_deadend* - determines if a cell is in deadend, if so, builds an escape path using a chain of parent cells, viable status and a deadend score of each parent

Class Robot

The class contains various state parameters of the robot and functionality required to navigate the maze. The functions include:

- *select_next* - determines the next path to take depending on whether the robot is in a deadend, or navigates to next unvisited cell. It invokes the grid to build navigation trees (see *Grid.build_tree*). When selecting the most cost-efficient path, it uses a combination of path length to the target cell and h-cost from the target to the goal (Manhattan distance). The past with the least cost is selected
- *next_move* - called by the environment when robot moves to the next location. The environment provides sensor information as part of parameters. If a robot predetermined a path before, the function creates a move instruction according to the next cell to navigate to, and returns the instruction to the environment. If no predetermined path exists, the robot determines new one by calling *select_next()*
- *move_instr* - the function builds an move instruction depending on the next cell to visit. The instruction consists of a rotation degree, and number and direction of steps to take
- *on_goal_reached* - robot calls the function when the goal is reached. The function updates robot's state based on the current execution phase, and indicates whether to continue discovery or prepare for final run. When the discovery is complete, this function selects the shortest path created during two discovery phases (from origin to center and from center to origin)
- *save_path* - called upon reaching the goal. It saves the discovered path for further evaluation
- *reset* - resets robot's state after reaching maze center and before exploring back to the starting cell
- *optimize_path* - called at the end of discovery stage to select the shortest path; it invokes other functions described below
- *select_short_legs* - given two paths, the function picks shorter leg between two intersecting cells. Example given below
- *merge_steps* - merges single steps into multi-step moves on straight legs in of the maze

analysis.py

The script contain utilities to display information about robot's performance and to visually represent the maze. It can include such indicators as optimal path, coordinates, and distance to center.
The script is run standalone and accepts the following command-line arguments:

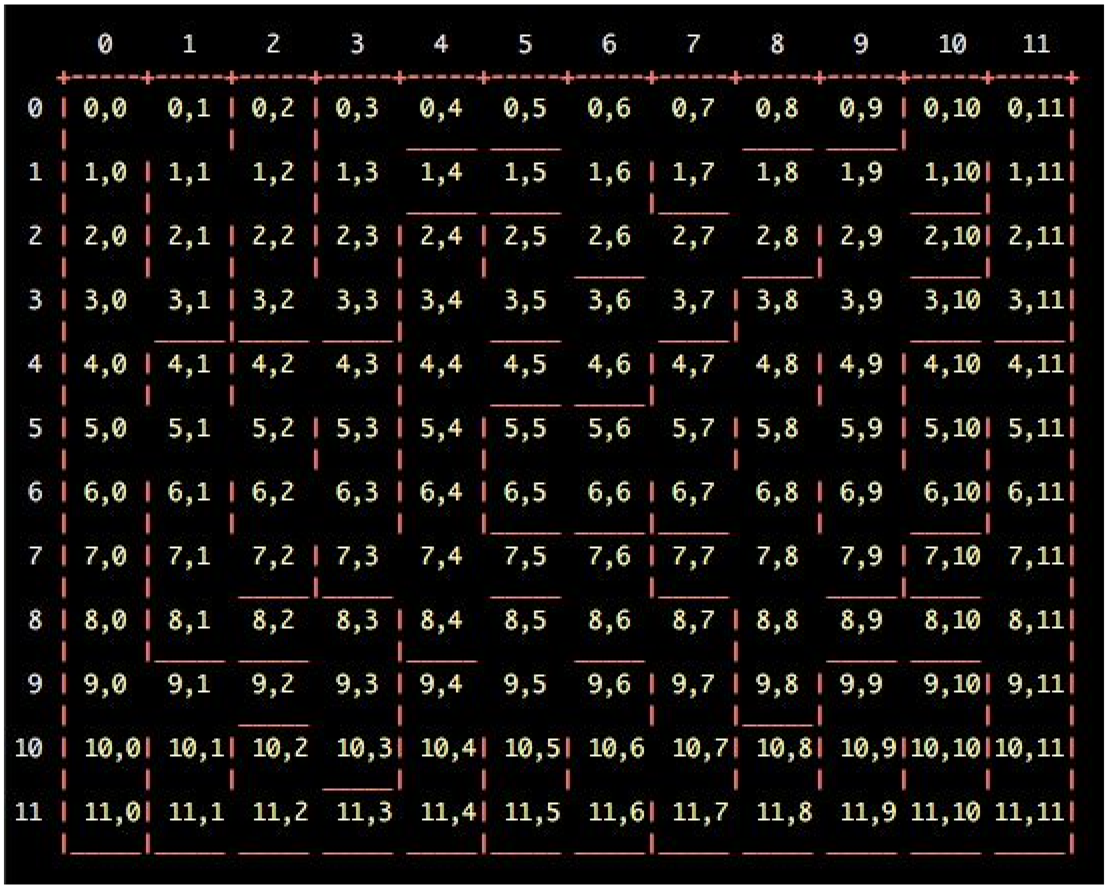
```
analysis.py -m <maze_spec> [ -l <log> -r (OhP | ShP | ScP | CsP) | -d | -c ) ]

   -m - maze specification file
   -l - log file
   -r - show marks (requires log parameter). Options:
         OhP - optimal path
         ShP - non-shorted path
         ScP - start-to-center path
         CsP - center-to-start path
   -d - show distance to centre
   -c - show coordinates
```

```
    -h - this help
```

For example to show each cell's coordinates, run:

```
      python analysis.py -m test_maze_01.txt -c
```

Output:



```
       0      1      2      3      4      5      6      7      8      9     10     11
    +------+------+------+------+------+------+------+------+------+------+------+------+
  0 | 0,0    0,1  | 0,2  | 0,3    0,4    0,5    0,6    0,7    0,8    0,9  | 0,10   0,11 |
    |             |      |      ______ ______               ______ ______|             |
  1 | 1,0  | 1,1    1,2  | 1,3    1,4    1,5    1,6  | 1,7    1,8    1,9    1,10 | 1,11 |
    |      |             |      ______ ______        |______               ______|      |
  2 | 2,0  | 2,1  | 2,2  | 2,3  | 2,4  | 2,5    2,6    2,7    2,8  | 2,9    2,10 | 2,11 |
    |      |      |      |      |      |       ______        ______|       ______|      |
  3 | 3,0    3,1  | 3,2    3,3  | 3,4    3,5    3,6    3,7  | 3,8    3,9    3,10   3,11 |
    |       ______|______ ______|       ______        ______|               ______ ______|
  4 | 4,0  | 4,1  | 4,2    4,3  | 4,4    4,5    4,6  | 4,7    4,8  | 4,9  | 4,10   4,11 |
    |      |      |             |       ______ ______|             |      |             |
  5 | 5,0    5,1    5,2  | 5,3  | 5,4  | 5,5    5,6    5,7  | 5,8    5,9  | 5,10 | 5,11 |
    |                    |      |      |                    |             |      |      |
  6 | 6,0  | 6,1  | 6,2    6,3  | 6,4  | 6,5    6,6  | 6,7    6,8  | 6,9    6,10 | 6,11 |
    |      |      |             |      |______ ______|______        |       ______|      |
  7 | 7,0  | 7,1    7,2  | 7,3    7,4    7,5    7,6  | 7,7    7,8    7,9  | 7,10   7,11 |
    |      |       ______|______        ______       |______        ______|______        |
  8 | 8,0  | 8,1    8,2    8,3  | 8,4    8,5    8,6    8,7  | 8,8    8,9    8,10   8,11 |
    |      |______ ______       |______        ______       |       ______ ______        |
  9 | 9,0    9,1    9,2    9,3  | 9,4    9,5    9,6  | 9,7  | 9,8  | 9,9    9,10 | 9,11 |
    |              ______       |                    |      |______|             |      |
 10 | 10,0 | 10,1 | 10,2   10,3 | 10,4 | 10,5 | 10,6   10,7 | 10,8 | 10,9 |10,10 |10,11 |
    |      |      |       ______|      |      |             |      |      |      |      |
 11 | 11,0 | 11,1   11,2   11,3   11,4 | 11,5   11,6 | 11,7   11,8   11,9  11,10  11,11 |
    |______|______ ______ ______ ______|______ ______|______ ______ ______ ______ ______|
```

Figure 4. Maze coordinates

Solution Challenges

One of the more challenging parts in this solution is selecting the next cell to visit after the robot finds itself in a loop (see Figure 3). The problem involves a trade-off decision between whether to travel to nearest unvisited cell or to a cell which is closer to the goal (maze center or origin).

On one hand, the robot would use fewer steps by visiting the nearest unvisited cell and thus will be immediately rewarded by that saving. On the other hand, that cell could be farther away from the goal, and the long-term cost would be negatively affected by that decision.

To resolve that dilemma, the robot has to make more intelligent decision and balance out its chances. In this solution, I decided to combine the two options to introduce a decision factor.

From the implementation point of view, the solution resulted in introducing a recursive search *core_lib.build_tree* function and subsequent evaluation of the most optimal move which is implemented in *robot.select_next.*
When building a tree representing the shortest path to unvisited cells, various criteria has to be taken into account such as the search should:

- Include only visited cells where all passageways are defined. Otherwise, it is unclear how to build a path to the target from the undefined cell
- Terminate if some branch leading to the target would result in longer path
- Continue searching through all branches to find the least-costly

Subsequent evaluation of which unvisited cell and the path leading to it is better required the use of h-cost of that cell. But the implementation here required selecting the path with shorter length and lowest h-cost using a sorting algorithm.

The recursive search and evaluation is thus resulted in one of the harder-to-understand logic in the code.

Refinement

This section describes refinements done to initial implementation.

1. When a robot selects next path between paths provided by the grid (described under *Robot.select_next*), most-efficient path is the one whose cost is a sum of path length and h-cost. When this sum is biased to give more weight to h-cost (x1.5), the total number of steps to reach the goal is reduced:

Maze	Score Before	Score After
test_maze_01	37.5	35.9
test_maze_02	60.4	58.8
test_maze_03	68.0	58.1

Table 1. The score after adding bias to h-cost

2. Two discovered paths ((1) on the way to maze center (2) on the way to the origin) may intersect at some points. The length of legs between two intersections may differ. More optimal path should contain a shorter leg. The robot was modified to merge best legs from either path when evaluating the overall optimal path. Note that this should be taken care of by A* algorithm when thorough, one-way, discovery process is performed. But since the approach used here uses two-way discovery (center and origin cells), this

optimization is helpful. Result for maze_01 is improved:

Maze	Score Before	Score After
test_maze_01	35.9	33.9
test_maze_02	58.8	58.8
test_maze_03	58.1	58.1

Table 2. The score after selecting shorter legs

3. The robot is further improved so as to take into account that it can travel 3 cells at a time on a straight row of cells. The results:

Maze	Score Before	Score After
test_maze_01	33.9	20.9
test_maze_02	58.8	42.8
test_maze_03	58.1	37.1

Table 3. The score after utilizing straight paths

Results

Model Evaluation and Validation

Each maze has the optimal path expressed by the number of moves to reach maze center from the origin, and is given in the table below. Information in this table will be used to compare the results against.

Maze	Benchmark Moves
01	17
02	23
03	25

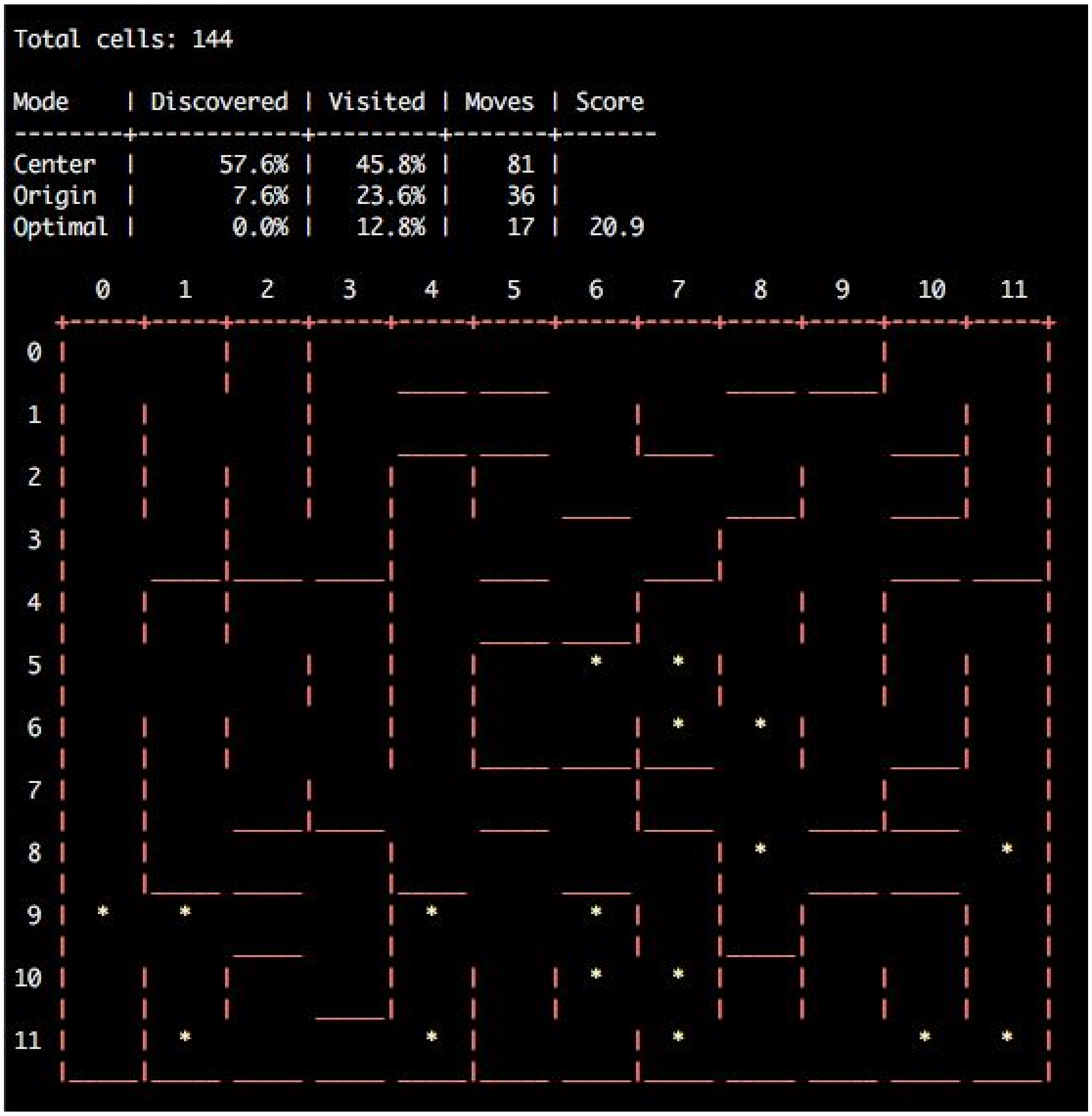
Table 4. Benchmark

Test Maze 01

Figure 5 shows robot's performance against test_maze_01. The star symbols plot the moves that robot makes when following its optimal path.

The first row of the stats shows that when traveling towards maze center, the robot discovered 57.6% and visited 45.8% of the total number of cells. It made 81 moves when it reached the center. The robot kept exploring towards origin and discovered additional 7.6% of cells, and visited 23.6% more. It took 36 steps to reach the origin.

The robot discovered a path that takes 17 moves to reach maze center. A final score comes down to 20.9. The number of moves is on par with the benchmark value.



```
Total cells: 144

Mode    | Discovered | Visited | Moves | Score
--------+------------+---------+-------+-------
Center  |      57.6% |   45.8% |    81 |
Origin  |       7.6% |   23.6% |    36 |
Optimal |       0.0% |   12.8% |    17 |  20.9
```

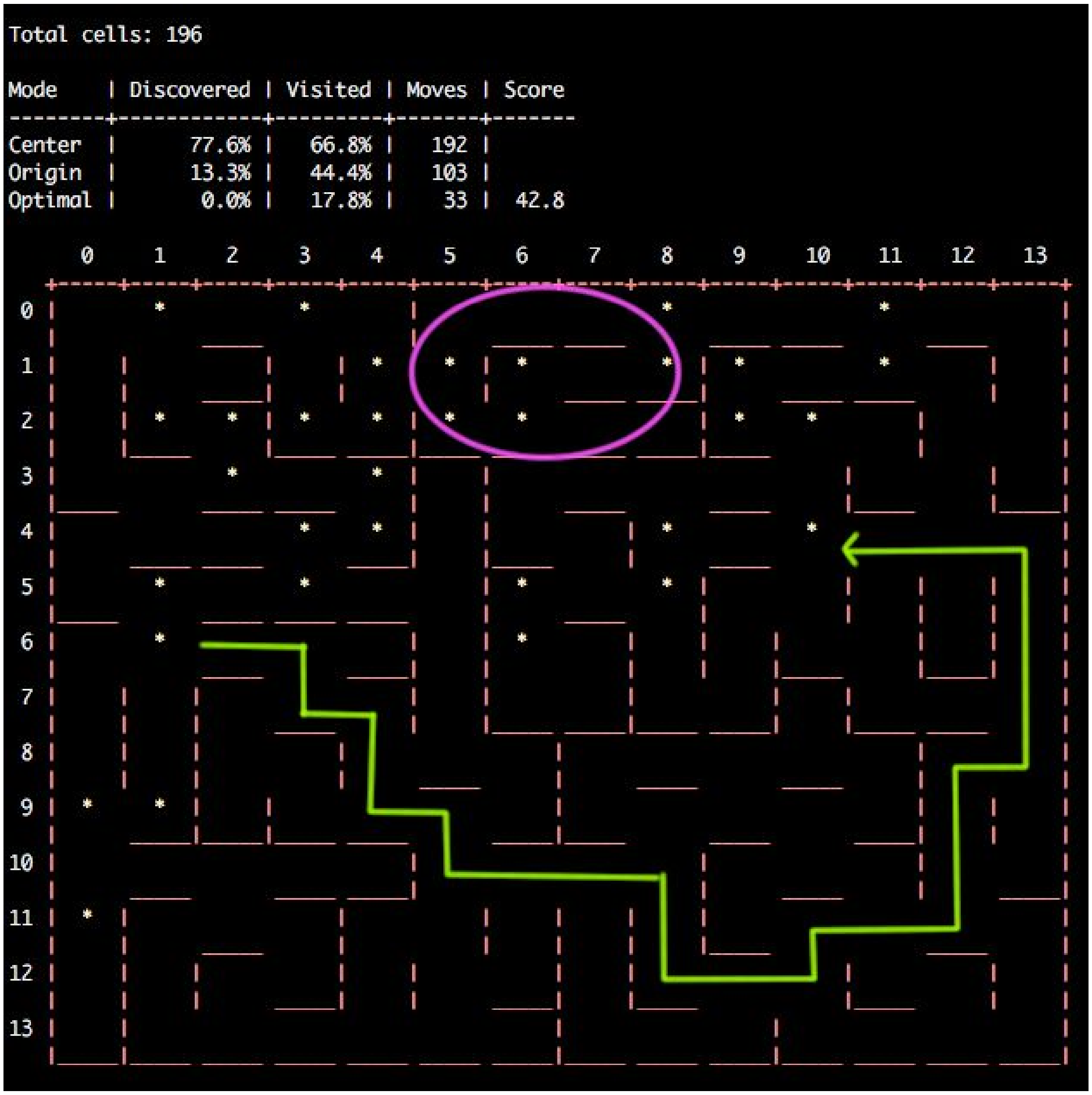
Figure 5. Optimal path for 12x12 maze

Test Maze 02

In maze 02, the discovered path takes 33 moves to reach the center; shown with yellow stars in Figure 6. Although, more optimal path exists as indicated by green line. This path takes 23 moves.

On some path legs, circled with magenta, the robot didn't discover more optimal moves and thus incurred extra cost.

Performance on this maze is 44% worse when compared to the most optimal path.



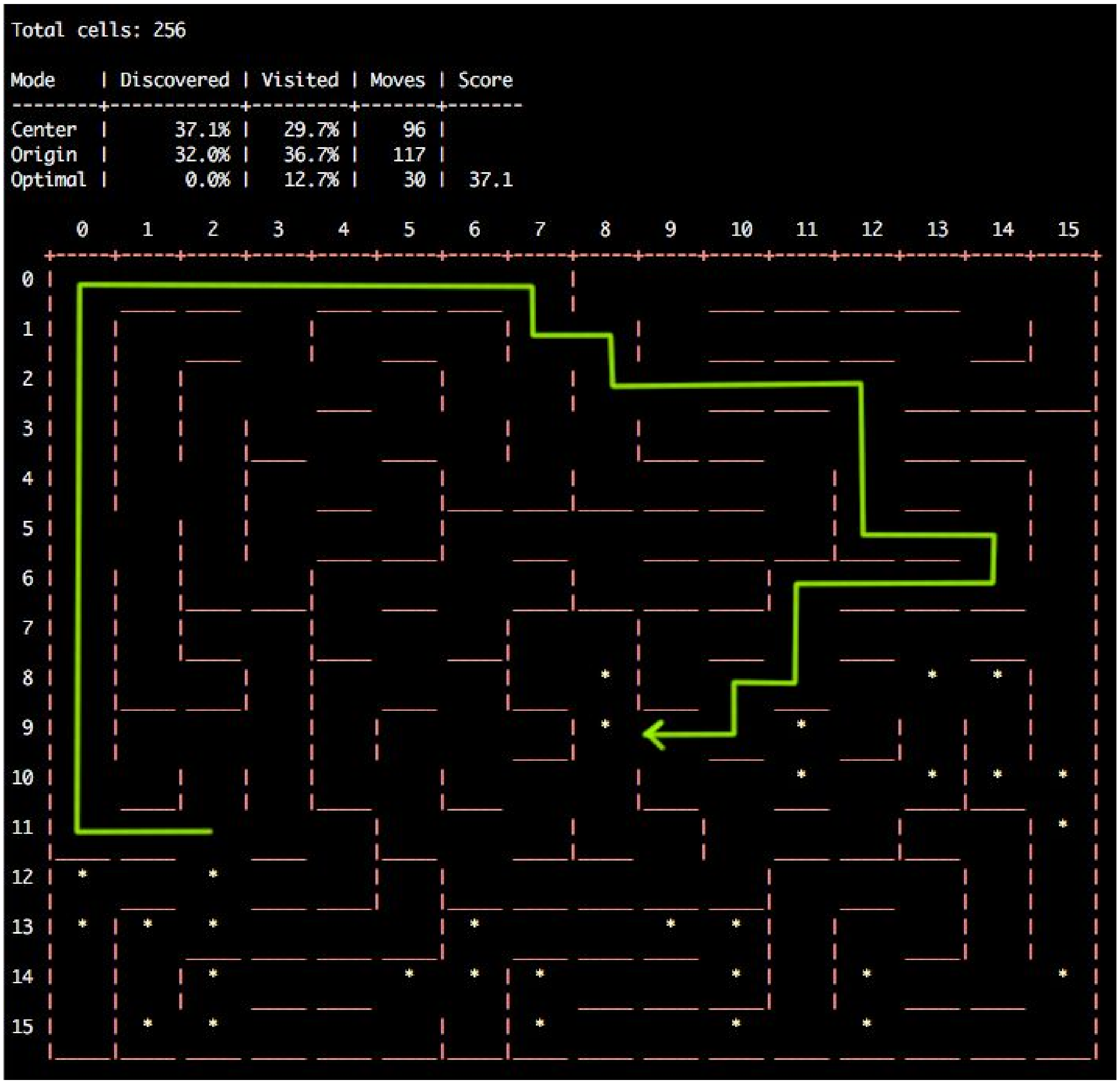
```
Total cells: 196

Mode     | Discovered | Visited | Moves | Score
--------+------------+---------+-------+-------
Center  |      77.6% |   66.8% |   192 |
Origin  |      13.3% |   44.4% |   103 |
Optimal |       0.0% |   17.8% |    33 |  42.8
```

Figure 6. Optimal path for 14x14 maze

Test Maze 03

In maze 03, robot-discovered optimal path takes 30 moves (see Figure 7). In comparison, the best optimal path (green line) takes 25 moves.

The performance on this maze is better than on maze 02. It is about 20% worse than the most optimal solution.



```
Total cells: 256

Mode    | Discovered | Visited | Moves | Score
--------+------------+---------+-------+-------
Center  |      37.1% |   29.7% |    96 |
Origin  |      32.0% |   36.7% |   117 |
Optimal |       0.0% |   12.7% |    30 |  37.1
```

Figure 7. Optimal path for 16x16 maze

Justification

With the performance comparison given in Table 5, the project objective is achieved: performance results are within 50% of the best possible.

Maze	Benchmark Moves	Robot Moves	Performance
01	17	17	100%
02	23	33	56%
03	25	30	80%

Table 5. Results

Conclusion

Free-Form Visualization

When exploring a maze, the robot prefers proximity to the target (center or origin). It uses path straightness when planning the final route. Therefore, the performance should be affected with the presence of straight paths which the robot didn't discover.

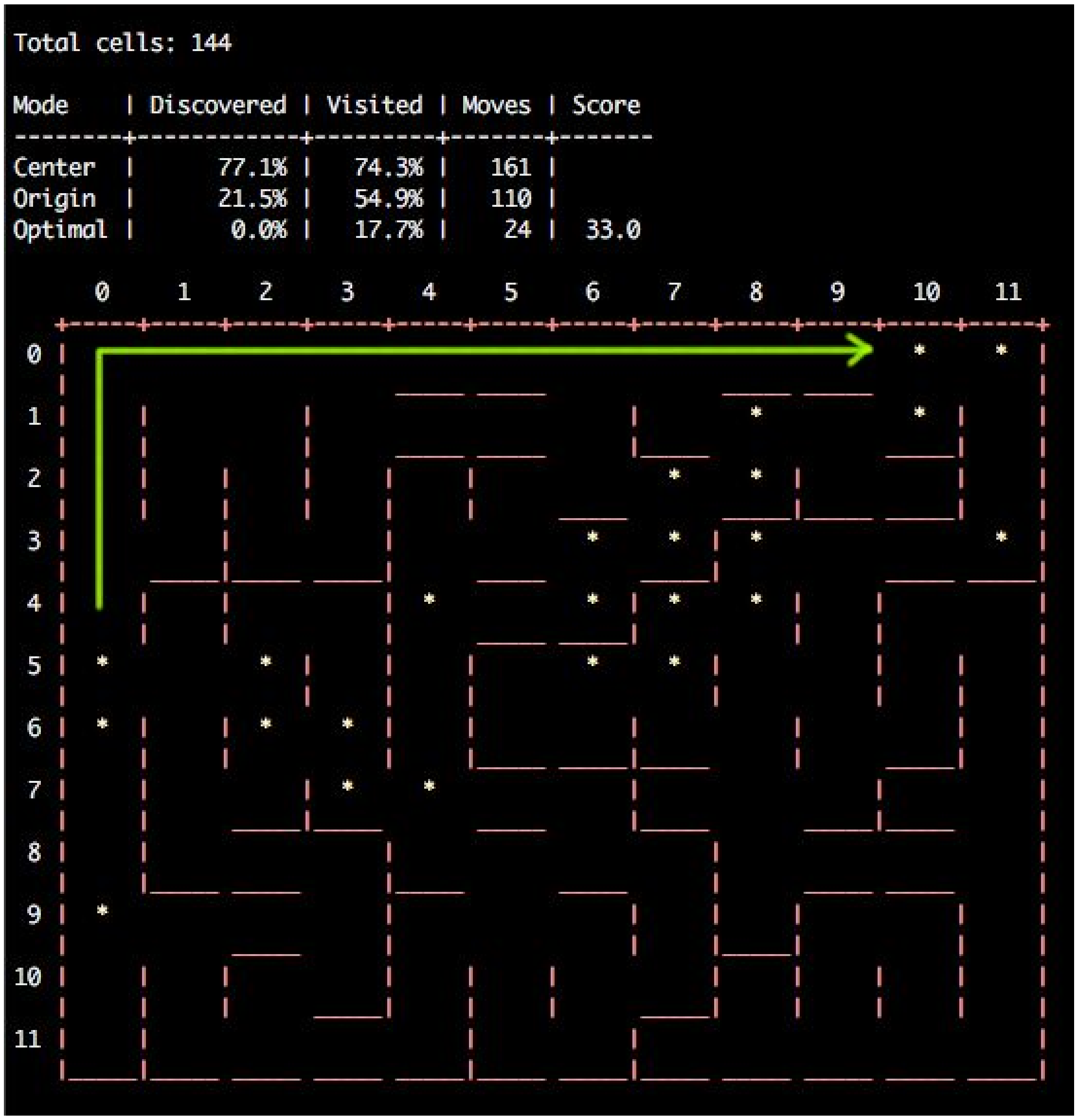
Figure 8 demonstrates the shortcoming in robot's algorithm. Maze 01 specification was modified to prove that fact as so:

- Removed vertical walls along the top row
- Added top wall to cell (11, 7)
- Added top wall to cell (3, 9)

The results show that the robot discovered 98.6% of the maze, and selected an optimal path of 24 moves. However, significantly better path exists which requires only 14 moves (shown with green line). The performance is at low 28%.

Maze	Benchmark Moves	Robot Moves	Performance
04	14	24	28%

Table 6. Results for modified 12x12 maze



```
Total cells: 144

Mode    | Discovered | Visited | Moves | Score
--------+------------+---------+-------+-------
Center  |      77.1% |   74.3% |   161 |
Origin  |      21.5% |   54.9% |   110 |
Optimal |       0.0% |   17.7% |    24 |  33.0
```

Figure 8. Optimal path for modified 12x12 maze

Reflection

As the first step while working on this project, I wanted to get a better visualization of the mazes. The functionality would allow me to focus on high-level problem without a lot of time spent on mental calculations. The display has to be programmatic and automated so as to reduce manual input or use of other tools. Initial implementation would show cell locations and Manhattan distances to center.

To help with the debugging and analysis further, I added logging capability. The maze display would optionally parse the log, and show robot's path in single and multi-cell navigation. For final analysis, I added statistical information such as the percentage of discovered and visited cells, the number of moves and the score of optimal path.

Algorithm choice was a challenging process. After some research, A* mostly fit the bill for the main navigation strategy. Although, some questions needed to be addresses on top of that including loop and dead-end detection. For example, after getting into the loop, the robot had to pick next unvisited cell that is more than one step away. This required an evaluation of cost-effectiveness for visiting each of those cells. A common approach in A* of visiting a first cell from a priority queue could incur a significant cost.

Since A* in current implementation would process neighbouring cells in certain order, the discovered path may not likely be so optimal. This is one reason I decided to switch the goal to the origin and make the robot continue exploring towards it after reaching the maze center. This decision had to account for the fact that f-score(s) deduced for each cell on the way to the center now are invalid with the new goal. Overall, I'm satisfied with the strategy, but as usual there is always a room for improvement.

To help with verifying the changes, I added unit tests in the form of doctest(s).

Improvement

Further improvements could definitely be made. One problem that needs to be addressed is described in section Free-Form Visualization. The robot must have a more robust algorithm to account for straight paths during and/or after exploration.

In general, A* as implemented in this solution should be augmented further to improve the performance in discovering shorter paths. Alternatively, it would be interesting to see how an algorithm such as Q-learning could be used in such a problem.

In environments where real hardware is used, significant considerations has to be given to robot's precision in sensor readings and motion. The information from sensors will likely have noise and vary depending on a robot's proximity and position to walls and passageways.

Propulsion mechanisms are also imperfect so that the robot cannot be 100% certain that when it issues a command to move to point *x* that it will be at that point. Issues with motion precision can come from over/under supplying power to propulsion mechanism, timing of applied power or imprecise feedback from the mechanism.

Popular approaches to mitigate these problems is to use probabilistic techniques such as simultaneous localization and mapping (SLAM). To help with motion precision or smoothness of navigation, PID control is the way to go.

Important factors also include hardware selection and budget. Sensors such as laser scanners are very accurate in measuring distance and shape, but are costly. Infrared distance sensors are much less expensive, but may not perform well in brightly lit environments. Rotary encoders

will provide a good motion feedback, but what if the robot operates on a slippery surface and the wheels slip? The distance travelled as reported by the encoder will not be representative of the actual distance. Therefore, various alternatives must be evaluated to meet the needs.
As an example of a creative approach is when one of the competing robots in Micromouse challenge used a telescoping pole with a camera on it to get a snapshot of the maze from above so as to "explore" it without actually moving through the maze. The time saved is the time gained.