Capstone Project

Ritwik Das

28th October 2016

Plot and Navigate a Virtual Maze

Definition

Project Overview

A robot mouse is tasked with plotting a path from a corner of the maze to its centre. Programs are developed to control the virtual robot to navigate a virtual maze.

Classical AI search algorithms are concerned with finding action sequences that reach goal states from start states in completely known state spaces. Often, however, state spaces are not known in advance, and path finding algorithms need to gather information in the world to locate a goal state and a path leading to it. Examples include (A) mobile delivery robots that operate in initially unknown buildings, and (B) software agents that have to find WWW pages of a given content by following links from their current page.

Problem Statement

In the first run, the robot mouse tries to map out the maze to not only find the centre, but also figure out the best paths to the centre. The robot must enter to the goal within the time limit but it is free to continue exploring the maze after finding the goal.
In the subsequent run, the robot mouse is brought back to the start location. It must attempt to reach the centre in the fastest time possible, using what it has previously learned.

Scoring Metric

On each maze, the robot must complete two runs. In the first run, the robot is allowed to freely roam the maze to build a map of the maze. The robot is then moved back to the starting position and orientation for its second run. The robot’s score for the maze is equal to the number of time steps required for the second run, plus one thirtieth the number of time steps required for the first run. The rationale behind the “1/30th is that the first run is primarily aimed towards exploration of the maze and maximization of the same so that shortest path can be found from start to finish in the second run. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

Analysis

Data Exploration and Visualization

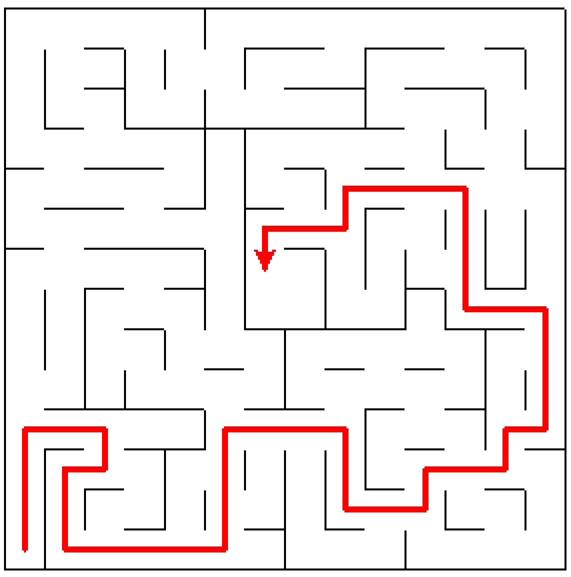
The shape of every test maze is a square with dimensions as 12, 14 or 16. The start location is always at the left bottom corner (0, 0), and the goal room always occupies 4 cells in the centre. The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor. It is assumed that the robot's turning and movement is perfect. It is also assumed that the sensors have no noise and the data provided is 100 % accurate.

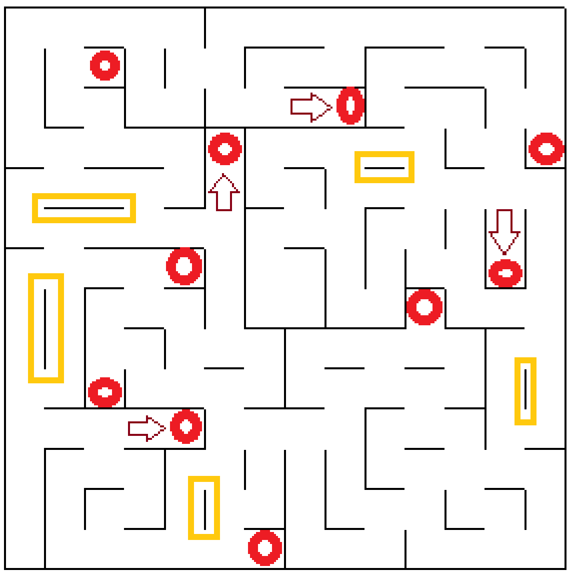
At the start location, both left and right sides have walls so that it can only move forward. The sensors will return the distance between the robot and walls in a tuple as in (left distance, forward distance, right distance).On each time step of the simulation, the robot may choose to rotate clockwise or counter clockwise 90 degrees or not rotate, then move forwards or backwards a distance of up to three units. If the robot tries to move into a wall, it stays where it is. After movement, one time-step has passed, and the sensors return readings for the open squares in the robot's new location to start the next time unit.

In a nutshell the following must be determined for each location of the robot:

- ***Rotation*** is expected to be an integer taking one of three values: -90, 90, or 0, indicating counter clockwise, clockwise, or no rotation, respectively.
- ***Movement*** follows rotation, and is expected to be an integer in the range [-3, 3] inclusive. The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

<u>Maze 02 (14X14)</u>





PROPERTIES OF THE MAZE

1. Not simply connected: - The above maze is *not simply connected* since it has loops indicated by orange rectangular boxes. It is also *not a perfect maze* since it clearly has more than one solution.

2. Number of dead-ends and blind alleys: - It has *10 dead ends* as indicated by the red circles and *4 blind alleys* indicated by the brown arrows.

3. Convolution: - It is evaluated by a metric *Length of Longest Path* which is defined as the ratio of cells in Shortest Path to the total number of cells. A ratio closer to 1 indicates a highly convoluted maze. **Convolution for this maze =22% [**100*43/196]

4. Distribution of valency: - Valency of a cell is defined as the number of possible directions to go from a particular cell. A dead-end has a valency of one. Crossroads have a valency of four. Unbranched corridors have a valency of two, and T-junctions have a valency of three. A maze with a high percentage of T-junctions and crossroads exposes the solver to lots of options. One with a high percentage of corridors (valence two cells), takes the user on long rides. The **average valency for this maze is 2.30** which means that is more biased towards taking the robot on long windy rides rather than expose it to many option.

5. Start to Goal Path: - One of the possible paths from start to goal is shown in red on the left maze.

Algorithms and Techniques

Good Neighbor Algorithm

First Run: The following algorithm is considered for the first run

1. Start-> Increment the **visited counter** of the start location by 1. Increment the **sensor detected counter** of the start location by 1
2. Update the graph of the maze using sensor readings of the current location. The graph must be updated in such a way that all possible moves of the robot from any location must be its neighbours. The robot will keep on adding edges (neighbours) as it keeps on gleaning information about the maze.
3. From the current location go to the next-most neighbour which has a **minimum visited counter**. Resolve ties by choosing **minimum heuristic distance** towards the goal. (The heuristic is defined here as the *Manhattan distance*.)
4. Update the **visited counter** of the current location by *1*.

5. Update the **sensor detected counter** of all the locations visible from the current location by *0.5*
6. Repeat from 2-5 till goal is reached. (Make sure to update both counters)
7. After goal is reached , calculate coverage of the maze by using the following method:
    a. Initialize total coverage to 0.
    b. For a particular location if the sensor detected counter is 0.5 increment total coverage by 0.5
    c. For a particular location if the sensor detected counter is >0.5 increment total coverage by 1
    d. Loop for all locations
    e. Calculate *Coverage=(100*total coverage)/(dimension of maze^2)*
8. If coverage is <*76* continue exploring: **Bad Neighbor Algorithm:**
    a. Continue exploring by the same algorithm used for reaching the goal except this time use **minimum sensor-detected counter** to choose next-most neighbour and **maximum heuristic distance to resolve ties**
    b. Update visited counter, sensor detected counter and add nodes to the graph G just like in steps 3, 4 and 5.
    c. Keep exploring until coverage<*76* is false
9. If coverage is >*76* Stop
10. Return 'Reset', 'Reset' for second run.

BackTracking Algorithm

First Run: The following algorithm is also considered for the first run

1. Initialize a matrix **vis2** to zero to all locations of the maze. This matrix will set 1 to locations reached once and will set -1 to locations from which the robot has turned back or visited twice.
2. Start-> set the **vis2** of the start location by 1. Increment the **sensor detected counter** of the start location by 1. Add the current location to a stack **s.**
3. Update the graph of the maze using sensor readings of the current location. The graph must be updated in such a way that all possible moves of the robot from any location must be its neighbours. The robot will keep on adding edges (neighbours) as it keeps on gleaning information about the maze.
4. Go to the next neighbor in the graph which has a vis2 value of 0. Use minimum distance to goal to resolve ties.
5. Add the new location to the stack and set its vis2 value to 1.
6. If from any particular location none of the neighbours have a vis2 value of 0, then backtrack. Pop the current location from the stack. Set its vis2 value to -1. Make the now current value in the stack as the new location.

7. Repeat until goal is reached. After goal is reached calculate coverage by the aforementioned formula.
8. If coverage < 76 continue exploring: **Bad Backtracker Algorithm:**
   a. Continue exploring by the same algorithm used for reaching the goal except this time use **maximum heuristic distance to resolve ties**
   b. Update the vis2 matrix, sensor detected counter and add edges to the graph in the same way.
   c. Keep exploring until coverage<*76* is false

   Location Backtracker Algorithm: Variation of the above algorithm is to use sensor detected counter to resolve ties among vis2==0 neighbours and use the maximum heuristic to further resolve ties.
9. If coverage is >*76* Stop
10. Return 'Reset', 'Reset' for second run.

Justification of the Algorithms:

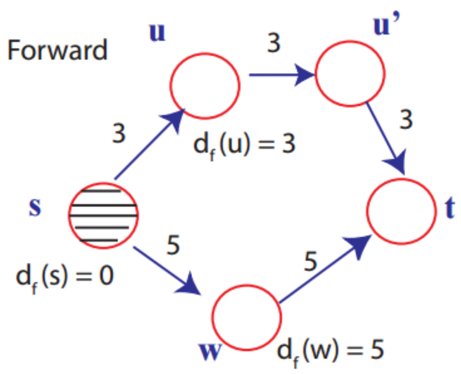
1. It doesn't keep looping around dead-ends and effectively escapes from them as will be further seen in the results. This is because it keeps a counter of the visited and located points.
2. It approaches towards the goal since it uses the heuristic distance to resolve ties.
3. 75% Coverage is a reasonable estimate to believe that in most of the cases 75 % maze coverage is a reasonable compromise between the number of time steps in the first and second run and would already have enough information for the second-run to find the shortest path or the close to shortest path
4. It uses the located points counter in the post-goal reach phase since it is more important to look at goals which have been not been located at all.
5. It is robust and is **guaranteed to find a path** from the start to end due to its goal-seeking greedy tendency coupled with inclination to visit nodes which have been less visited.

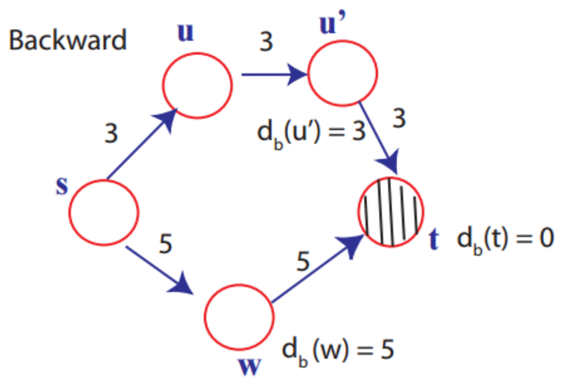
Second Run:

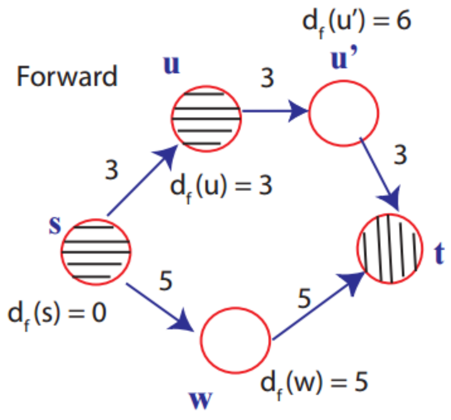
The Bi-Directional Dijkstra's algorithm is used on the second run. The algorithm is used with on the Graph G which has all the information of the maze in the form of nodes and edges. The Bi-Directional Dijkstra's algorithm is twice as fast as the ordinary Dijkstra's algorithm. This is because ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. The following steps are to be followed to implement the same.

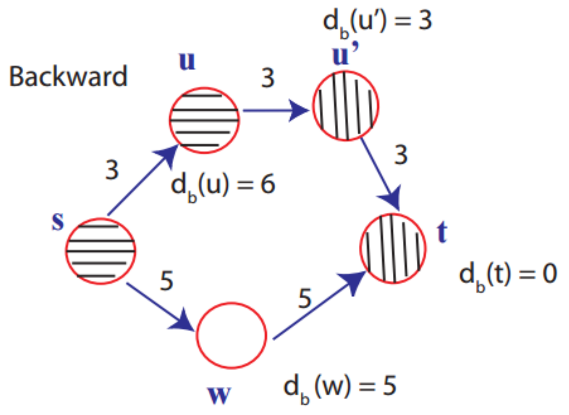
- Alternate forward search from start node(s), backward search from end node (t). (follow edges backward)
- $d_f(u)$ distances for forward search; $d_b(u)$ distances for backward search
- Algorithm terminates when some vertex w has been processed, i.e., deleted from the queue of both searches, Q_f and Q_b.

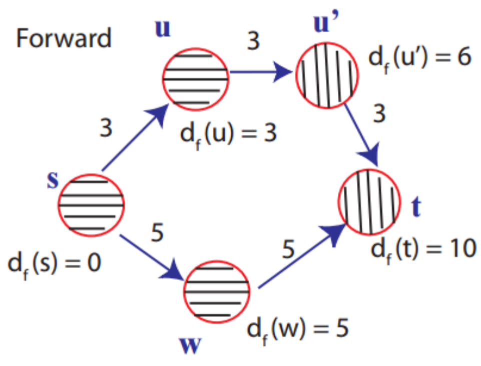
The following visual explanation will make it clearer.

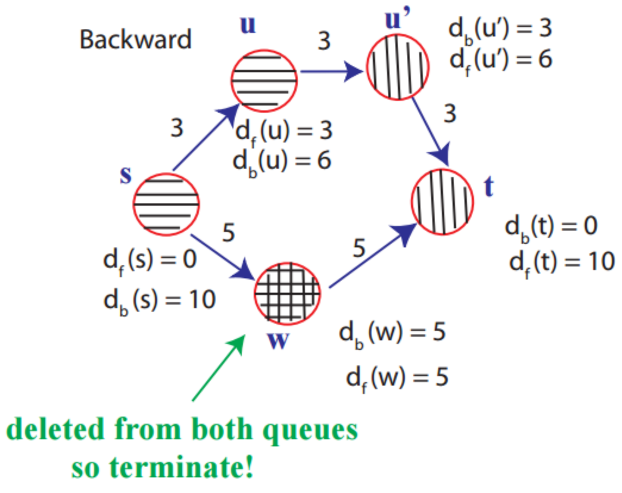












The Bi-directional search has many applications, including that of computing driving directions. We use the path that we obtained from the Bi-Directional Dijkstra algorithm to move the robot with the intended direction and rotation.

Benchmark

First Run:

Worst Case: For the first run the number of worst case steps would be of the order of $2N^{2.}$ Tremaux Algorithm would return to the start if every cell is visited twice. The recursive backtracking Algorithm if applied would also in the worst take about $2N^2$ steps. The random algorithm would perform even poorly in a very difficult maze but it won't be considered a usable, robust algorithm by any means if the maze is unknown before-hand. Hence it is not considered. Therefore in the worst case we will make around **$2N^2$ time steps** as well if the maze is convoluted enough to not use our jumping liberty of 3 steps at a time.

Average Case: The average case is subject to debate. As per my thinking and my take on the coverage metric of the maze defined earlier, on an average we should make around 75% coverage of the maze for the 1st run but we will have go more than once for some cells and also have *only detected but not visited* value of 0.5 for some cells. On an average we would take **$0.75*N^2$ time steps** keeping in mind our end goal of minimizing the number of time steps in the final run.

Second Run:

Worst Case: For the second run the worst case would be when the maze is convoluted enough that we not only require of the order of **N^2 moves** but around **N^2 time steps** as well.

Average Case: The average case for the second run highly depends on the type of maze and therefore would be a mere loose metric if used to compare with further results. However what could be a useful metric is if it was compared with the number of optimal moves for that particular maze. It is difficult to prove mathematically but the average case based on the coverage metric (3/4) defined earlier could be said to be at (1/ (3/4)) **1.33*No of optimal time steps.** 100% coverage would lead to shortest path and hence the above formula is mapped accordingly. In my opinion this average case is a bit conservative and therefore a good benchmark.

Total Run: <u>*N: = Maze Dimension*</u>

Benchmark Score [Worst Case] = $1.067N^2$ …… [N^2+ (2/30)*N^2]

Benchmark Score [Average Case] = 1.33*(Optimal Time Steps) + $0.025N^2$..[0.75 *(1/30)*N^2]

Methodology

Data Pre-processing

The sensor specification and environment designs are already provided. The sensor data and the maze specification is 100 % accurate. Therefore no data pre-processing is required.

Implementation

The first thing which is done for the first run for every time step is that at every location, the graph of the maze is updated by using **sensor data, current location and heading** and of-course the graph till the previous time step. Each node of the graph contains coordinates of the 2D array element. For instance [5] [6] = 6*N+5. This is the method used to store nodes in the graph.

E.g. In the position [0] [0] for the 12X12 Maze. The graph would have edges [(0, 1), (1, 2), (2, 3)… (10, 11)]

```
def addtograph(G, location, heading, sensors):
    loc = location[1] * dim + location[0]

    if heading == 'up':
        if sensors[0] > 0:
            for i in range(loc - dim, loc - dim * sensors[0] - 1, -dim):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[1] > 0:
            for i in range(loc + 1, loc + sensors[1] + 1, 1):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[2] > 0:
            for i in range(loc + dim, loc + dim * sensors[2] + 1, dim):
                G.add_edge(i, loc)
                loc = i
    elif heading == 'down':
        if sensors[0] > 0:
            for i in range(loc + dim, loc + dim * sensors[0] + 1, dim):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[1] > 0:
            for i in range(loc - 1, loc - sensors[1] - 1, -1):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[2] > 0:
            for i in range(loc - dim, loc - dim * sensors[2] - 1, -dim):
                G.add_edge(i, loc)
                loc = i
    elif heading == 'right':
        if sensors[0] > 0:
            for i in range(loc + 1, loc + sensors[0] + 1, 1):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[1] > 0:
            for i in range(loc + dim, loc + dim * sensors[1] + 1, dim):
                G.add_edge(i, loc)
                loc = i
```

```python
        loc = location[1] * dim + location[0]
        if sensors[2] > 0:
            for i in range(loc - 1, loc - sensors[2] - 1, -1):
                G.add_edge(i, loc)
                loc = i
    else:
        if sensors[0] > 0:
            for i in range(loc - 1, loc - sensors[0] - 1, -1):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[1] > 0:
            for i in range(loc - dim, loc - dim * sensors[1] - 1, -dim):
                G.add_edge(i, loc)
                loc = i
        loc = location[1] * dim + location[0]
        if sensors[2] > 0:
            for i in range(loc + 1, loc + sensors[2] + 1, 1):
                G.add_edge(i, loc)
                loc = i
```

After that the **located array** is updated. This array contains counter of visited locations as well as sensor detected locations.

```python
def addtolocated(located, location, heading, sensors, metric):
    loc = location[1] * dim + location[0]

    if heading == 'up':
        if sensors[0] > 0:
            for i in range(loc - dim, loc - dim * sensors[0] - 1, -dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[1] > 0:
            for i in range(loc + 1, loc + sensors[1] + 1, 1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[2] > 0:
            for i in range(loc + dim, loc + dim * sensors[2] + 1, dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
    elif heading == 'down':
        if sensors[0] > 0:
            for i in range(loc + dim, loc + dim * sensors[0] + 1, dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[1] > 0:
            for i in range(loc - 1, loc - sensors[1] - 1, -1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[2] > 0:
            for i in range(loc - dim, loc - dim * sensors[2] - 1, -dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
    elif heading == 'right':
        if sensors[0] > 0:
            for i in range(loc + 1, loc + sensors[0] + 1, 1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[1] > 0:
            for i in range(loc + dim, loc + dim * sensors[1] + 1, dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[2] > 0:
            for i in range(loc - 1, loc - sensors[2] - 1, -1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
    else:
        if sensors[0] > 0:
            for i in range(loc - 1, loc - sensors[0] - 1, -1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[1] > 0:
            for i in range(loc - dim, loc - dim * sensors[1] - 1, -dim):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
        if sensors[2] > 0:
            for i in range(loc + 1, loc + sensors[2] + 1, 1):
                x, y = getcoord(i)
                if located[x][y] == 0:
                    located[x][y] += metric
```

: Pre-Goal Reach Algorithms:

The following “goodneighbor” function was used to find the next best location given the current location of the robot, the till-now formed graph of the maze and the located counter matrix. This is if the ***Good Neighbor Algorithm (as discussed earlier)*** is applied.

```
def goodneighbor(G, visited, location):
    currentloc = location[1] * dim + location[0]
    minimum = inf
    for neighbor in G.neighbors(currentloc):
        if minimum == visited[neighbor % dim][neighbor / dim]:
            if heurist_val > hcost([neighbor % dim, neighbor / dim], dim):
                minimum = visited[neighbor % dim][neighbor / dim]
                heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
                optimumloc = neighbor
        if minimum > visited[neighbor % dim][neighbor / dim]:
            minimum = visited[neighbor % dim][neighbor / dim]
            heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
            optimumloc = neighbor
    return getcoord(optimumloc)
```

In a nutshell, it returns the coordinates of the location which has been visited minimum times and minimum heuristic distance is used to resolve ties.

The following backtracker function was used to determine the next best location of the robot **if the *Backtracking Algorithm*** is applied.

```
def backtracker(s, heading, vis2, G):
    currentloc = s.peek()
    vis2[currentloc] = 1
    heurist_val = inf
    flag = 0
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 0:
            flag = 1
            if heurist_val > hcost([neighbor % dim, neighbor / dim], dim):
                desiredloc = neighbor
                heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 1 and flag == 0:
            flag = 1
            s.pop()
            desiredloc = s.peek()
            vis2[currentloc] = -1
    print vis2
    return travel(currentloc, desiredloc, heading)
```

: Post Goal Reach Algorithms:

The following ***badneighbor*** function is used to explore the maze post goal reach till coverage is atleast 75 %. The difference between the goodneighbor and badneighbor is that the latter **moves away from the goal** to resolve ties and uses the **located matrix** which contains information about visited as well as sensor detected cells.

```
def badneighbor(G, located, location):
    currentloc = location[1] * dim + location[0]
    minimum = inf
    for neighbor in G.neighbors(currentloc):
        if minimum == located[neighbor % dim][neighbor / dim]:
            if heurist_val < hcost([neighbor % dim, neighbor / dim], dim):
                minimum = located[neighbor % dim][neighbor / dim]
                heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
                optimumloc = neighbor
        if minimum > located[neighbor % dim][neighbor / dim]:
            minimum = located[neighbor % dim][neighbor / dim]
            heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
            optimumloc = neighbor
    return getcoord(optimumloc)
```

The following **badbacktracker** function is used to explore the maze post goal reach till coverage is atleast 75 %. The difference between the backtracker and the bad backtracker is that the latter moves away from the goal to resolve ties.

```
def badbacktracker(s, heading, vis2, G):
    currentloc = s.peek()
    vis2[currentloc] = 1
    heurist_val = -inf
    flag = 0
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 0:
            flag = 1
            if heurist_val < hcost([neighbor % dim, neighbor / dim], dim):
                desiredloc = neighbor
                heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 1 and flag == 0:
            flag = 1
            s.pop()
            desiredloc = s.peek()
            vis2[currentloc] = -1
    print vis2
    return travel(currentloc, desiredloc, heading)
```

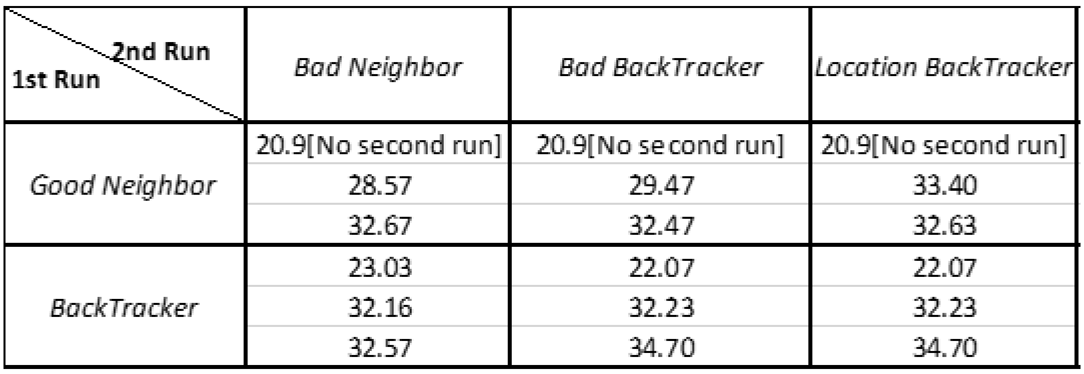
The following **locbacktracker** function is used to explore the maze post goal reach till coverage is at least 75 %. The difference between the backtracker and the locbacktracker is that the latter incorporates the located matrix to move to the desired location and moves away from the goal to resolve ties between them.

```
def locbacktracker(s, heading, vis2, G, located):
    currentloc = s.peek()
    vis2[currentloc] = 1
    heurist_val = -inf
    flag = 0
    minimum = inf
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 0:
            flag = 1
            if minimum == located[neighbor % dim][neighbor / dim]:
                if heurist_val < hcost([neighbor % dim, neighbor / dim], dim):
                    minimum = located[neighbor % dim][neighbor / dim]
                    heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
                    desiredloc = neighbor
            if minimum > located[neighbor % dim][neighbor / dim]:
                minimum = located[neighbor % dim][neighbor / dim]
                heurist_val = hcost([neighbor % dim, neighbor / dim], dim)
                desiredloc = neighbor
    for neighbor in G.neighbors(currentloc):
        if vis2[neighbor] == 1 and flag == 0:
            flag = 1
            s.pop()
            desiredloc = s.peek()
            vis2[currentloc] = -1
    print vis2
    return travel(currentloc, desiredloc, heading)
```
The minute details of the code and the implementation are properly discussed in the development file of robot.py which is submitted along with the report.

Refinement

A mix and match combination of the algorithms discussed above is used to create a table of results. For e.g. The Good Neighbour Algorithm is combined with the Bad Neighbour Algorithm , in another instance it is combined with the Bad Backtracker and so on. The tabular results of the scores are presented below.

1st Run: = Pre Goal Reach Stint
2nd Run: = Post Goal Reach Stint



2nd Run / **1st Run**	*Bad Neighbor*	*Bad BackTracker*	*Location BackTracker*
Good Neighbor	20.9[No second run]	20.9[No second run]	20.9[No second run]
	28.57	29.47	33.40
	32.67	32.47	32.63
BackTracker	23.03	22.07	22.07
	32.16	32.23	32.23
	32.57	34.70	34.70

Based on the above result it is seen that the Backtracker Algorithm performs poorly as compared to the Neighbor Algorithm. While looking at the GUI which was made using turtle during the entire simulation, it was seen that the backtracker was not using the optimal path.

Therefore, in the coming results in the second run the required coverage was increased to 85 %, and the following results were obtained.

2nd Run / 1st Run	Bad Neighbor	Bad BackTracker	Location BackTracker
Good Neighbor	N/A	21.53	21.63
		32.67	30.13
		35.33	35.40
BackTracker	23.03	24.47	24.47
	30.60	30.10	30.10
	35.37	36.20	36.20

Now, the results for some situations have improved while others have gone down due to over-coverage of the maze. It was observed that the backtracker was wasting many time-steps going backward since it makes only one unit movement per time step. In the next set of results, the all the backtrackers are boosted i.e. the neighbour of a cell is not only the adjacent cell but also all the cells within 3 unit distances away from it or less depending upon sensor values. To compensate for it the threshold coverage for the backtracker is kept at 66% while for the Good Neighbor is kept at 75%.

2nd Run / 1st Run	Bad Neighbor	Bad BackTracker	Location BackTracker
Good Neighbor	N/A	20.9[No Second Run]	20.9[No Second Run]
		30.23	28.67
		31.73	31.2
BackTracker	21.53	21.30	21.3
	28.83	30.87	30.87
	31.83	32.03	32.03

It is now observed that the score of all the backtrackers have reduced significantly. Therefore as a final run, even the Neighbor algorithms were boosted. The following results employ the boosted Neighbor and Backtracking Algorithms with threshold coverage at 75%.

2nd Run / 1st Run	Bad Neighbor	Bad BackTracker	Location BackTracker
Good Neighbor	23.60	22.53	22.2
	28.67	28.23	28.17
	32.37	31.37	30.97
BackTracker	21.53	21.63	21.63
	29.27	31.67	31.67
	32.37	32.83	32.83

The green cells are the best results overall so far. It is concluded that boosting (jumping around cells) doesn’t help much in smaller mazes (12X12) while is of a great help in larger mazes (14X14 and 16X16) .

Results

Model Evaluation and Validation

As confirmed by the refinement section above, the final model uses the following:

A. First Run:-

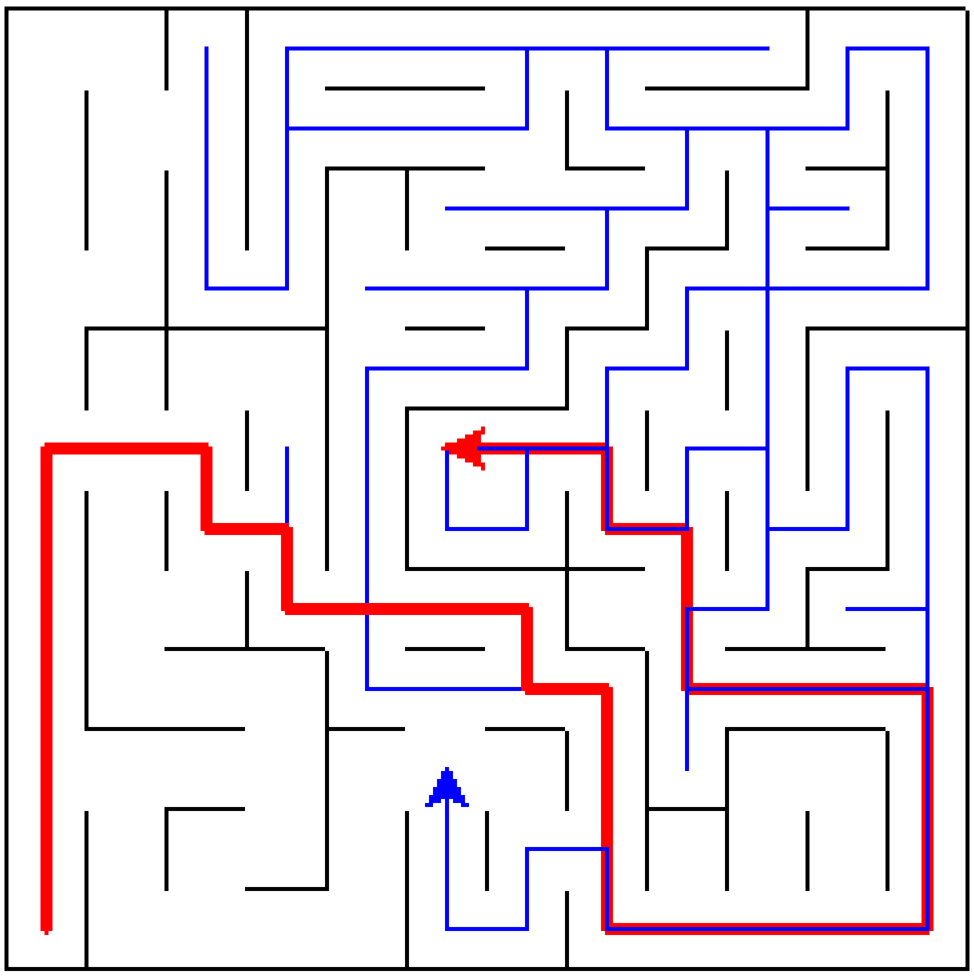
1. **Boosted Good Neighbor Algorithm** for pre-goal stint.
2. **Boosted Location BackTracker Algorithm** for post-goal stint.

B. Second Run:-

Bi-directional Dijkstra algorithm for second run.

In the following discussion, the path of the robot will be shown for both the runs. The blue lines show the movement of the robot in the first run. The red lines show the movement in the second run. [It overlaps over the blue line in some places making it invisible in places where the robot travelled only once]

Test Maze 01 Result



SCORE: *22.2*

Coverage Threshold for Second Run: 75 %

No. of Time Steps in the Second Run: 19

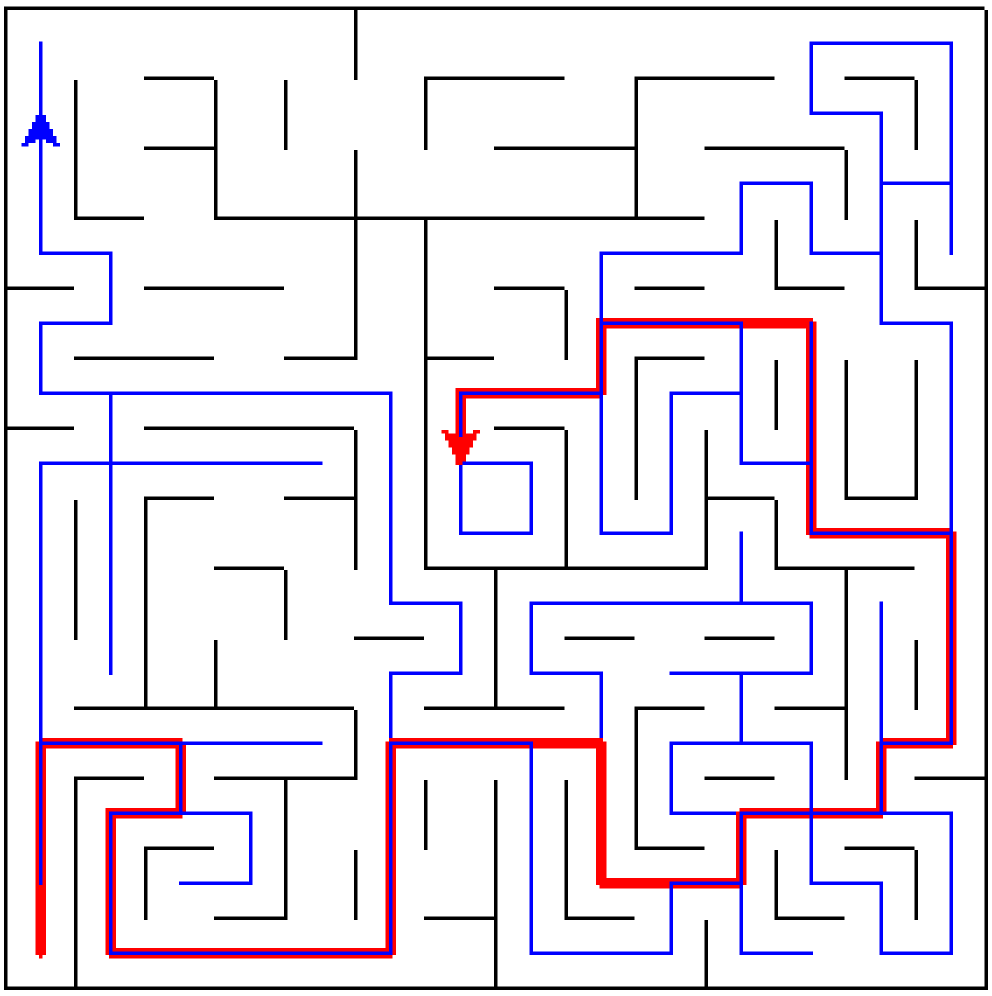
No. of Optimal Time Steps: 17

BENCHMARK SCORE: 26.17

BEST SCORE: 20.9[Good Neighbor & Bad Neighbor Algorithm]

No. of Time Steps in the Second Run: 17

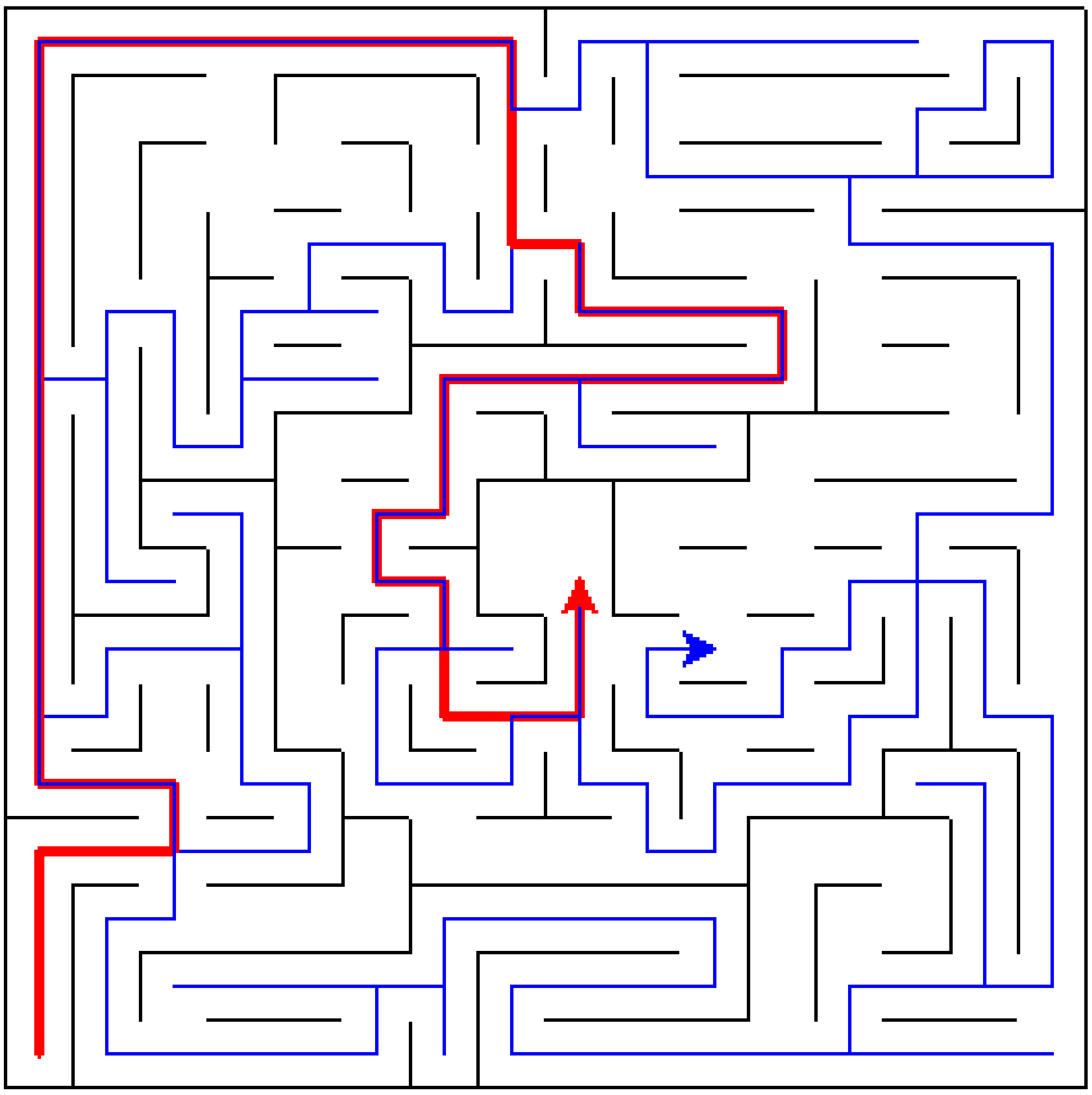
Test Maze 02 Result



SCORE: *28.17*
BENCHMARK SCORE: 35.44
Coverage Threshold for Second Run: 75 %
No. of Time Steps in the Second Run: 23
No. of Optimal Time Steps: 23

These are very good results. It is quite evident from the picture above how the robot covers a significant part of the maze in the first run without harming the score and in the second run is able to find the shortest path that exists.

Test Maze 03 Result



SCORE: *30.97*
BENCHMARK SCORE: 39.59
Coverage Threshold for Second Run: 75 %
No. of Time Steps in the Second Run: 26
No. of Optimal Time Steps: 25

This is again a good score. The optimal time steps is only one lesser than that taken by the robot. But again a compromise has to be made between the first run exploration and the second run path.

Taking all of the three cases into account it can be testified that the algorithmic approach used are robust and will reach the goal in the second with a minimal score. It also covers 75 % of the maze in about **0.5*N^2** time steps which is a validation for a good algorithm.

Justification

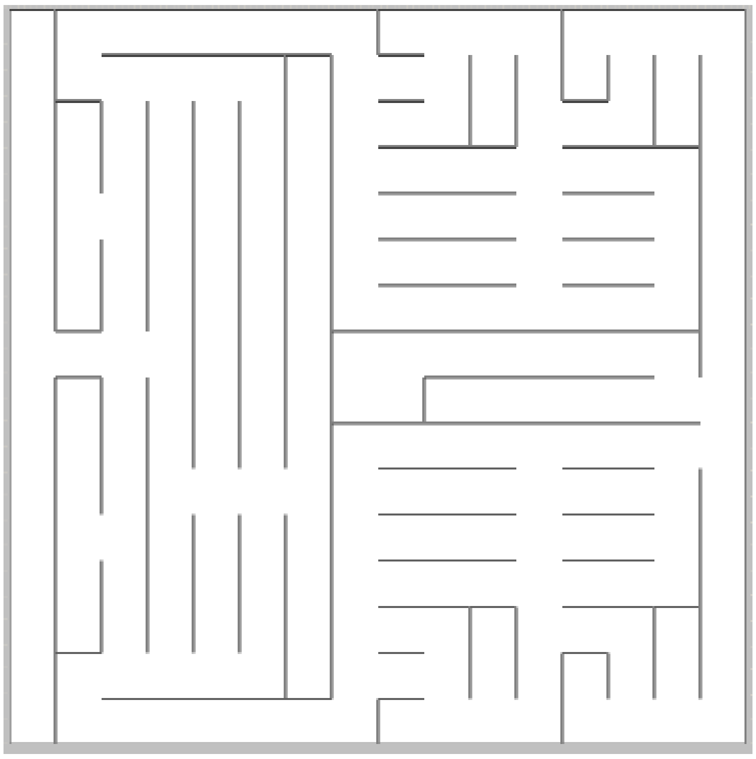
The algorithm performs much better than the benchmark score set in all of the test cases. Hence the solution proposed is significant enough to have adequately solved the problem. The following is a tabular result justifying the performance of the algorithm chosen.

Metric / Maze No	Score	Benchmark Score	Actual Moves	Optimal Moves
First	22.2	26.17	19	17
Second	28.17	35.44	23	23
Third	30.97	39.59	26	25

Conclusion

Free Form Visualization

Maze 04(16X16)



The above maze is specifically targeted to work against a heuristic/ goal seeking preference. The maze has a strong vertical bias in the left hand side, strong horizontal bias on the right hand side. It also has many dead ends and loops.

The strong point of the algorithm discussed till now becomes a weakness. In the first run, when the robot has no idea of the maze, it will keep going towards the goal only to wind up in the long vertical loops. It will loop each of them and then finally escape the left half. After that it will also loop the horizontal loops and then the upper dead-ends. After that only will it reach towards the goal but not before meeting the dead-end right before the goal one more time, coming out of it and FINALLY reaching the goal.

Other more advanced algorithms like VECA (Variable Edge Cost Algorithm) should be used. VECA provides a better performance guarantee than the heuristic-driven exploitation algorithms, and misleading heuristic knowledge can never completely deteriorate its performance. A parameter of VECA determines when it starts to restrict the Exploitation algorithm. This allows one to trade-off stronger performance guarantees (in case the heuristic knowledge is misleading) and more freedom of the exploitation algorithm (in case the quality of the heuristic knowledge is good).

Reflection

Summarizing,

1. The algorithm used has a greedy goal seeking tendency and gives preference to places less visited in the pre-goal reach stint and has a goal repulsive tendency in the post-goal reach stint for the 1st Run.
2. The 2nd Run uses Bi-Directional Dijkstra algorithm to find the shortest path.

I found deducing a strong algorithm for the 1st run to be a very good brain exercise. I started with a random controller having a weak heuristic tendency and ended up with a score of ~700 for the 3rd Maze. From there I improved to a score of 30.97. I used Turtle Graphics to continuously visualize the working of my algorithm. I could see physically what optimal path the robot should have taken by slowing down the speed of the turtle at necessary points. I finally landed up with a benchmark score, a coverage metric for the first run and maximization of the same for the first run. I used *networkx* library for the graph implementation which reduced the amount of code. The 2nd run was relatively simpler and used a simple implementation of the Bi-Directional Dijkstra algorithm on the graph generated in the first run.

Improvement

There is a significant amount of research done in goal-directed exploration techniques. Advanced algorithms like VECA (as discussed earlier) must be implemented.

A continuous domain will hold much more scope for improvement. The sensor values will have noise. The current location of the robot will have some noise and various filters like EKF may need to be used depending on the problem. SLAM algorithm must be implemented for the same. In addition to that appropriate control algorithms must be used to control the motors. The weight of the robot, its agility, choice of motor all will play a role in minimizing the time required to reach the goal. Hence, the project holds a scope of improvement not only in the virtual domain but also in the physical domain.