ROB-UY 3303 Project 3

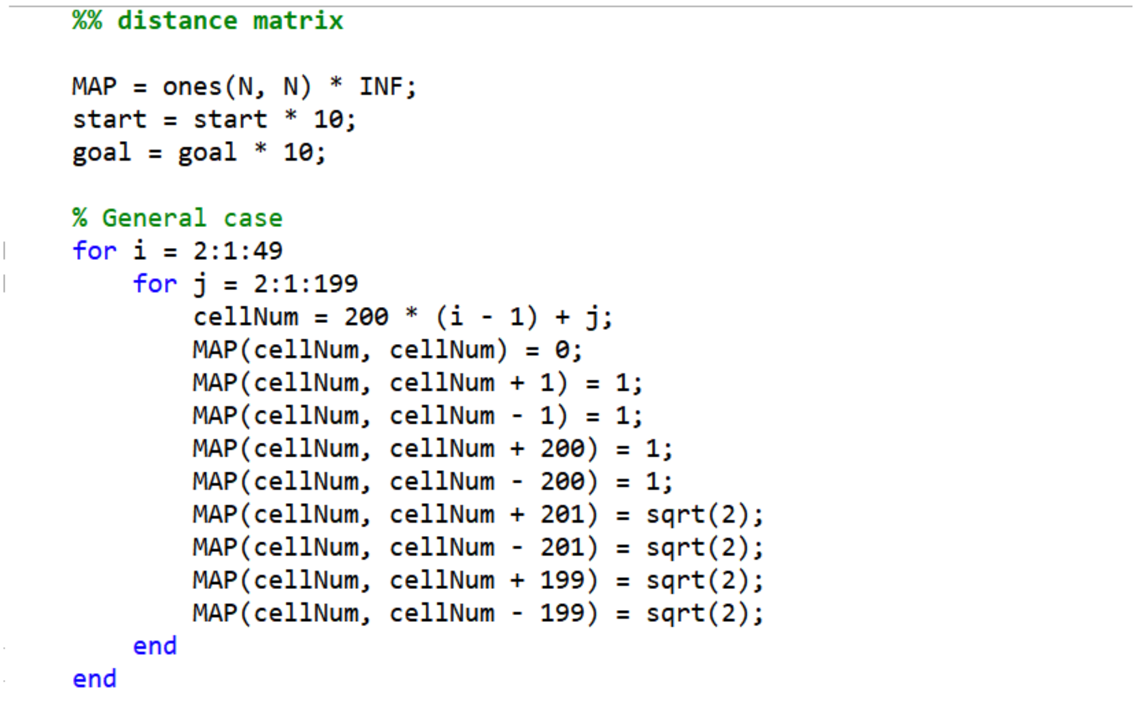
Seoyoon Kim / NetID: sk11361

1. Distance Matrix

To implement Dijkstra algorithm, $N \times N$ matrix that contains distance information between all nodes and their adjacent nodes is required. I assumed the environments as 8-connected graph. Therefore, the directly adjacent cells have a distance of 1 unit from each other, while cells located diagonally from one another correspond to a distance of $\sqrt{2}$ unit.

In my code, MAP is a matrix that contains all the distance information. It is initialized with INF value. Since the resolution of the grid is 0.1, I multiplied the coordinate values by 10 to make them compatible with cell numbers. For the cell at row i and column j, the cell number is calculated as $200 \times (i - 1) + j$.

In the general cases, which means it is not at a corner or an edge, there are 8 adjacent cells. The distance to itself is set as 0. It was utilized that there's a difference of 200 in cell numbers when there's a difference of one row, and a difference of 1 when there's a difference of one column.



```
%% distance matrix

MAP = ones(N, N) * INF;
start = start * 10;
goal = goal * 10;

% General case
for i = 2:1:49
    for j = 2:1:199
        cellNum = 200 * (i - 1) + j;
        MAP(cellNum, cellNum) = 0;
        MAP(cellNum, cellNum + 1) = 1;
        MAP(cellNum, cellNum - 1) = 1;
        MAP(cellNum, cellNum + 200) = 1;
        MAP(cellNum, cellNum - 200) = 1;
        MAP(cellNum, cellNum + 201) = sqrt(2);
        MAP(cellNum, cellNum - 201) = sqrt(2);
        MAP(cellNum, cellNum + 199) = sqrt(2);
        MAP(cellNum, cellNum - 199) = sqrt(2);
    end
end
```

For edge cases, there are 5 adjacent cells.

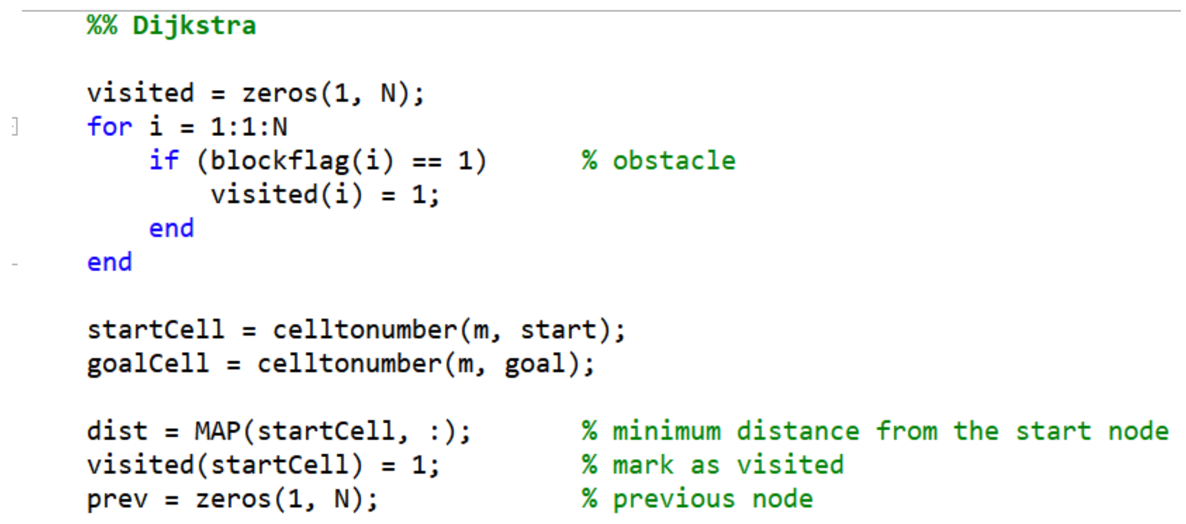
```
% i = 1
for j = 2:1:199
    cellNum = 200 * (1 - 1) + j;
    MAP(cellNum, cellNum) = 0;
    MAP(cellNum, cellNum + 1) = 1;
    MAP(cellNum, cellNum - 1) = 1;
    MAP(cellNum, cellNum + 199) = sqrt(2);
    MAP(cellNum, cellNum + 200) = 1;
    MAP(cellNum, cellNum + 201) = sqrt(2);
end

% i = 50
for j = 2:1:199
    cellNum = 200 * (50 - 1) + j;
    MAP(cellNum, cellNum) = 0;
    MAP(cellNum, cellNum + 1) = 1;
    MAP(cellNum, cellNum - 1) = 1;
    MAP(cellNum, cellNum - 199) = sqrt(2);
    MAP(cellNum, cellNum - 200) = 1;
    MAP(cellNum, cellNum - 201) = sqrt(2);
end

% j = 1
for i = 2:1:49
    cellNum = 200 * (i - 1) + 1;
    MAP(cellNum, cellNum) = 0;
    MAP(cellNum, cellNum + 200) = 1;
    MAP(cellNum, cellNum - 200) = 1;
    MAP(cellNum, cellNum + 1) = 1;
    MAP(cellNum, cellNum + 201) = sqrt(2);
    MAP(cellNum, cellNum - 199) = sqrt(2);
end

    % j = 50
for i = 2:1:49
    cellNum = 200 * (i - 1) + 50;
    MAP(cellNum, cellNum) = 0;
    MAP(cellNum, cellNum + 200) = 1;
    MAP(cellNum, cellNum - 200) = 1;
    MAP(cellNum, cellNum - 1) = 1;
    MAP(cellNum, cellNum - 201) = sqrt(2);
    MAP(cellNum, cellNum + 199) = sqrt(2);
end
```

For corner cases, there are 3 adjacent cells.

```
% Corner case (i = 1, j = 1)
cellNum = 200 * (1 - 1) + 1;
MAP(cellNum, cellNum + 200) = 1;
MAP(cellNum, cellNum + 1) = 1;
MAP(cellNum, cellNum + 201) = sqrt(2);

% Corner case (i = 50, j = 1)
cellNum = 200 * (50 - 1) + 1;
MAP(cellNum, cellNum - 200) = 1;
MAP(cellNum, cellNum + 1) = 1;
MAP(cellNum, cellNum - 199) = sqrt(2);

% Corner case (i = 1, j = 200)
cellNum = 200 * (1 - 1) + 200;
MAP(cellNum, cellNum + 200) = 1;
MAP(cellNum, cellNum - 1) = 1;
MAP(cellNum, cellNum + 199) = sqrt(2);

% Corner case (i = 50, j = 200)
cellNum = 200 * (50 - 1) + 200;
MAP(cellNum, cellNum - 200) = 1;
MAP(cellNum, cellNum - 1) = 1;
MAP(cellNum, cellNum - 201) = sqrt(2);
```

2. Dijkstra Algorithm

Matrix 'visited' stores whether each node has been visited before. The key is to consider nodes representing obstacles as visited from the start to prevent them from being explored.



```
%% Dijkstra

visited = zeros(1, N);
for i = 1:1:N
    if (blockflag(i) == 1)      % obstacle
        visited(i) = 1;
    end
end

startCell = celltonumber(m, start);
goalCell = celltonumber(m, goal);

dist = MAP(startCell, :);       % minimum distance from the start node
visited(startCell) = 1;         % mark as visited
prev = zeros(1, N);             % previous node
```

The 'celltonumber' function converts cell world coordinates to the cell index. The 'dist' matrix stores the minimum distance from the starting node to each node. The 'prev' matrix saves the cell number of the previous node that leads to the current node. The start cell is considered visited.

The next destination is selected as the closest node among those not yet visited. If the index is -1, then it implies all the node have been visited.

```
for i = 1:1:N-1

    % Find Smallest Node
    minDist = INF;
    minIndex = -1;

    for j = 1:1:N
        if (visited(j) == 1)
            continue;
        end
        if (dist(j) < minDist)
            minDist = dist(j);
            minIndex = j;
        end
    end

    newNode = minIndex;

    if (newNode == -1)
        break;
    end

    visited(newNode) = 1;
```

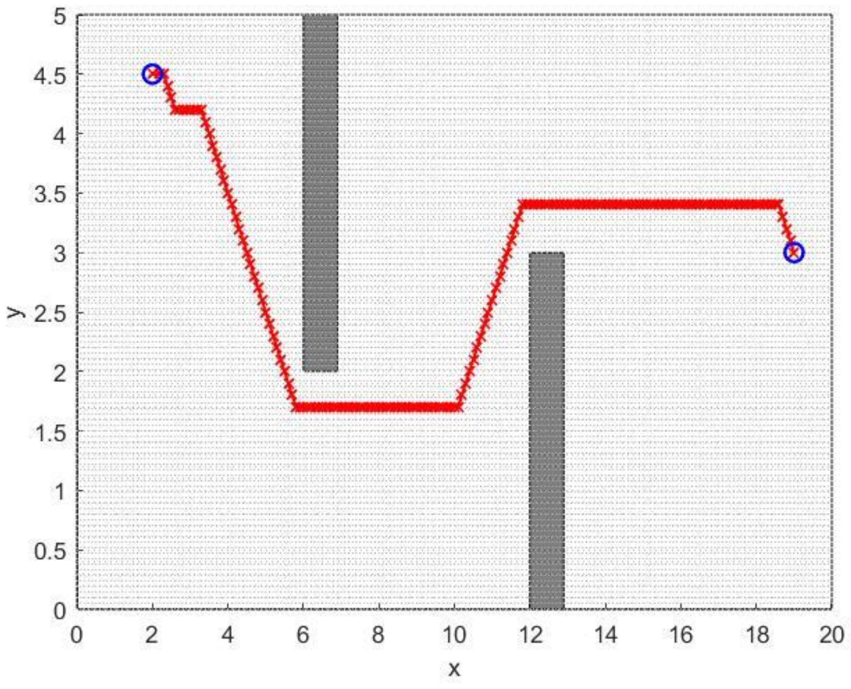
When a new node is selected, check whether the distance becomes shorter when passing through that node to reach other nodes. If it does, the shortest distance to that node becomes the sum of the distance from the starting point to the new node and the distance from the new node to that point. Therefore, the previous node becomes the new node.

```
% Update distance
for k = 1:1:N
    if (visited(k) == 1)
        continue;
    end
    if (dist(k) > (dist(newNode) + MAP(newNode, k)))
        dist(k) = dist(newNode) + MAP(newNode, k);
        prev(k) = newNode;
        % disp(newNode);
    end
end
```

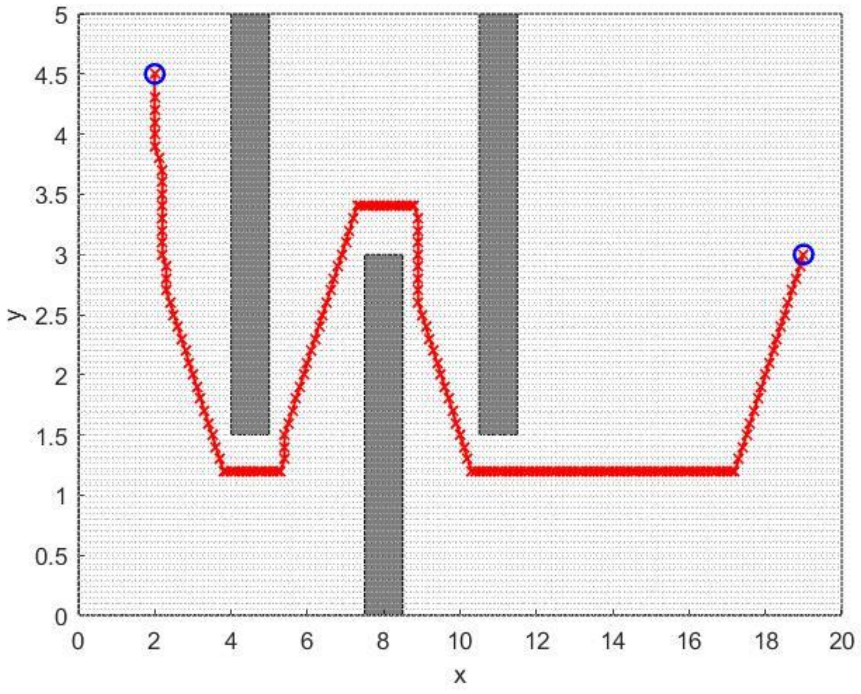
Since it's traversed backward by finding the previous node after reaching the final node, the path needs to be flipped once the reverse path is determined.

```
% Generate a Path
path(1, :) = goal;
curIndex = 1;
curCell = goalCell;

while (curCell ~= startCell)
    curIndex = curIndex+1;
    prevCell = prev(curCell);
    path(curIndex, :) = numbertocell(map, curCell);
    curCell = prevCell;
    if (curCell == 0)
        break;
    end
end

path(curIndex, :) = start;
path = path(1:curIndex, :);
path = flip(path / 10);
disp('Calculation Completed!');
```

3. Results



Map 1



Map 2

Considering distance of margin, I think my algorithm gives pretty efficient paths for both maps.