EXPERIMENT ONE
FINDING SHORTEST PATH USING BREADTH FIRST SEARCH (BFS) AND DEPTH FIRST SEARCH (DFS)

OBJECTIVE:

- To understand and implement MATLAB built-in function for Breadth First Search (BFS) and Depth First Search (DFS).
- To implement BFS and DFS to find shortest path from source to destination on a grid.
- To do the comparative analysis between BFS and DFS based on various parameters, such as time, number of nodes, path length, number of turns, number of iteration, number of explored nodes, etc.

EQUIPMENT OR SETUP:

- A workable Personal Computer (PC) with operating system
- Installed MATLAB software

BACKGROUND:

Breadth First Search (BFS):

Tutorial link: https://www.mathworks.com/help/matlab/ref/graph.bfsearch.html

The Breadth-First search algorithm begins at the starting node, s, and inspects all of its neighboring nodes in order of their node index. Then for each of those neighbors, it visits their unvisited neighbors in order. The algorithm continues until all nodes that are reachable from the starting node have been visited.

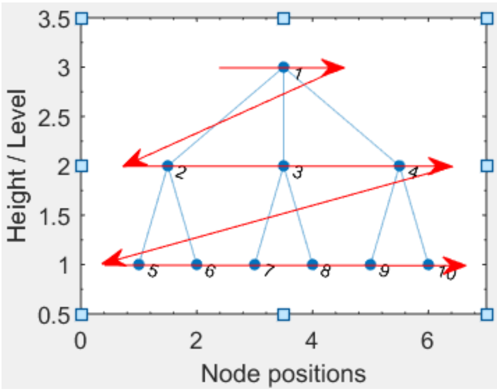


Figure Exp1.1
BFS Conceptual flow on a graph

Algorithm:

Event **startnode(S)**
Event **discovernode(S)**
NodeList = **{S}**

WHILE **NodeList** is not empty
 C = **NodeList{1}**
 Remove first element from **NodeList**
 FOR edge **E** from outgoing edges of node **C**, connecting to node **N**
  Event **edgetonew(C, E)**, **edgetodiscovered(C, E)** or **edgetofinished(C, E)**
  (depending on the state of node **N**)
  IF event was **edgetonew**
   Event **discovernode(N)**
   Append **N** to the end of **NodeList**
  END
 END

 Event **finishnode(C)**
END

bfsearch **Syntax for MATLAB:**

v = bfsearch(G, s)
T = bfsearch(G, s, events)
[T, E] = bfsearch(G, s, events)
[___] = bfsearch(___, 'Restart', tf)

v = bfsearch(G, s) applies breadth-first search to graph **G** starting at node **s**. The result is a vector of node

IDs in order of their discovery.

T = bfsearch(G, s, events) customizes the output of the breadth-first search by flagging one or more search events. For example, **T = bfsearch(G, s, 'allevents')** returns a table containing all flagged events, and **X = bfsearch(G, s, 'edgetonew')** returns a matrix or cell array of edges.

[T, E] = bfsearch(G, s, events) additionally returns a vector of edge indices E when events is set to **'edgetonew'**, **'edgetodiscovered'**, or **'edgetofinished'**. The edge indices are for unique identification of edges in a multigraph.

[__] = bfsearch(__, 'Restart', tf), where **tf** is true, restarts the search if no new nodes are reachable from the discovered nodes. You can use any of the input or output argument combinations in previous syntaxes. This option ensures that the breadth-first search reaches all nodes and edges in the graph, even if they are not reachable from the starting node, **s**.

Depth First Search (DFS):

Tutorial link: *https://www.mathworks.com/help/matlab/ref/graph.dfsearch.html*

The Depth-First search algorithm begins at the starting node, s, and inspects the neighbor of s that has the smallest node index. Then for that neighbor, it inspects the next undiscovered neighbor with the lowest index. This continues until the search encounters a node whose neighbors have all been visited. At that point, the search backtracks along the path to the nearest previously discovered node that has an undiscovered neighbor. This process continues until all nodes that are reachable from the starting node have been visited.

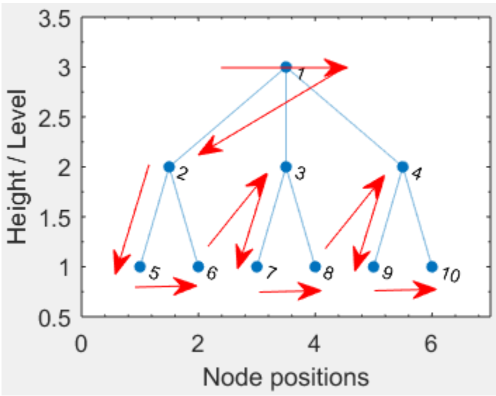


Figure Exp1.2
DFS Conceptual flow on a graph

Algorithm:

Event **startnode(S)**
Call **DFS(S)**

function **DFS(C)**
Event **discovernode(C)**

FOR edge **E** from outgoing edges of node **C**, connecting to node **N**
Event **edgetonew(C, E)**, **edgetodiscovered(C, E)** or **edgetofinished(C, E)**
(depending on the state of node **N**)
IF event was **edgetonew**
Call **DFS(N)**
END
END

Event **finishnode(C)**
END

dfsearch **Syntax for MATLAB:**

v = dfsearch(G, s)
T = dfsearch(G, s, events)
[T, E] = dfsearch(G, s, events)
[__] = dfsearch(__, 'Restart', tf)

v = dfsearch(G, s) applies depth-first search to graph **G** starting at node **s**. The result is a vector of node IDs in order of their discovery.

T = dfsearch(G,s,events) customizes the output of the depth-first search by flagging one or more search events. For example, **T = dfsearch(G, s, 'allevents')** returns a table containing all flagged events, and **X = dfsearch(G, s, 'edgetonew')** returns a matrix or cell array of edges.

[T, E] = dfsearch(G, s, events) additionally returns a vector of edge indices **E** when events is set to **'edgetonew'**, **'edgetodiscovered'**, or **'edgetofinished'**. The edge indices are for unique identification of edges in a multigraph.

[___] = dfsearch(___, 'Restart', tf), where **tf** is true, restarts the search if no new nodes are reachable from the discovered nodes. You can use any of the input or output argument combinations in previous syntaxes. This option ensures that the depth-first search reaches all nodes and edges in the graph, even if they are not reachable from the starting node, **s**.

- ***Note: dfsearch and bfsearch treat undirected graphs the same as directed graphs. An undirected edge between nodes s and t is treated like two directed edges, one from s to t and one from t to s.***
- ***Trees can be traverse in multiple ways in depth–first order or breadth–first order. The DFS for trees can be implemented using preorder, inorder, and postorder, while the DFS for trees can be implemented using level order traversal.***
- ***The time complexity of both DFS and BFS traversal is $O(V + E)$, where V and E are the total number of vertices and edges in the graph, respectively.***

PROCEDURE:

- Open MATLAB
- Open new M-file
- Type the program
- Save in current directory
- Compile and Run the program
- For the output see command window/Figure window
- Write/modify the code as instructed and determine the results

EXPERIMENTS (LAB PRACTICE):

1. bfsearch:

1.1 Perform Breadth-First Graph Search (Create and plot an undirected graph):

Write the following code in a **new m-file** and run the code. This will present and plot a graph on the figure window, as shown in Figure Exp1.3.

```
s = [1 1 1 1 2 2 2 2 2];
t = [3 5 4 2 6 10 7 9 8];
G = graph(s,t);
plot(G);
```

Now, perform a breadth-first search of the graph starting at node 2. The result indicates the order of node discovery.

```
v = bfsearch(G,2);
```

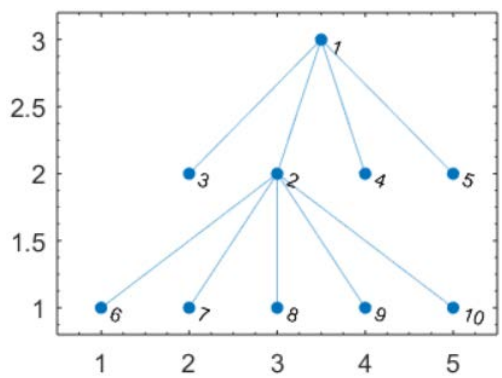


Figure Exp1.3
An undirected graph for BFS

Now type 'v' on the command window and observe the result shown below.

```
v =

     2
     1
     6
     7
     8
     9
    10
     3
     4
     5
```

1.2 Breadth-First Graph Search with All Events:

Create and plot a directed graph. Write the following code in a **new m-file** and run to observe the outputs. Relevant graph is shown in Figure Exp1.4.

```
s = [1 1 1 2 3 3 3 4 6];
t = [2 4 5 5 6 7 4 1 4];
G = digraph(s,t);
plot(G);

T = bfsearch(G,1,'allevents');
```

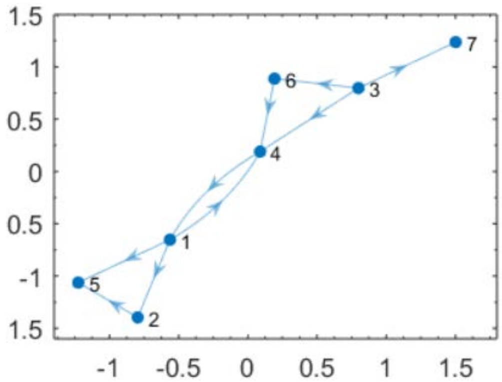


Figure Exp1.4
A directed graph for BFS

Now, type 'T' in the command window.

```
T =

         Event            Node      Edge
    ________________      ____    __________

    startnode               1     NaN    NaN
    discovernode            1     NaN    NaN
    edgetonew             NaN       1      2
    discovernode            2     NaN    NaN
    edgetonew             NaN       1      4
    discovernode            4     NaN    NaN
    edgetonew             NaN       1      5
    discovernode            5     NaN    NaN
    finishnode              1     NaN    NaN
    edgetodiscovered      NaN       2      5
    finishnode              2     NaN    NaN
    edgetofinished        NaN       4      1
    finishnode              4     NaN    NaN
    finishnode              5     NaN    NaN
```

1.3 Breadth-First Graph Search with Multiple Components:

Perform a breadth-first search of a graph with multiple components, and then highlight the graph nodes and edges based on the search results. Create and plot a directed graph in a **new m-file**. This graph has two weakly connected components.

```
s = [1 1 2 2 2 3 4 7 8 8 8 8];
t = [3 4 7 5 6 2 6 2 9 10 11 12];
G = digraph(s,t);
p = plot(G,'Layout','layered');
```

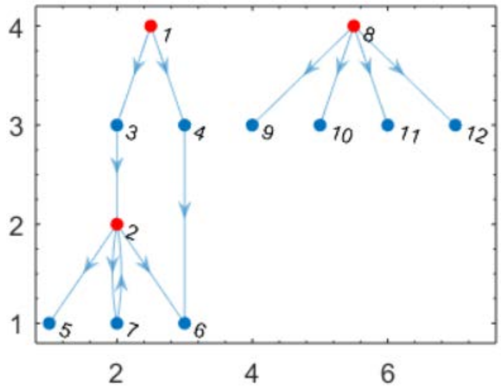


Figure Exp1.5
Directed graph with two weakly connected components

To check the connection components (weakly connected components), type the following code in the m-file and run.

```
c = conncomp(G,'Type','weak');
```

The 'c' contains the result as:

```
c =

     1     1     1     1     1     1     1     2     2     2     2     2
```

To run the bfsearch and see the events, write the following codes in the m-file and run.

```
events = {'edgetonew','edgetofinished','startnode'};
T = bfsearch(G,2,events,'Restart',true);

% highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetonew'), 'EdgeColor', 'g')
% highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetofinished'), 'EdgeColor', 'k')
highlight(p,T.Node(~isnan(T.Node)),'NodeColor','r')
```

The 'T' contains the result as:

```
T =

        Event            Node       Edge
    ________________     ____    __________

    startnode               1     NaN    NaN
    discovernode            1     NaN    NaN
    edgetonew             NaN       1      2
    discovernode            2     NaN    NaN
    edgetonew             NaN       1      4
    discovernode            4     NaN    NaN
    edgetonew             NaN       1      5
    discovernode            5     NaN    NaN
    finishnode              1     NaN    NaN
    edgetodiscovered      NaN       2      5
    finishnode              2     NaN    NaN
    edgetofinished        NaN       4      1
    finishnode              4     NaN    NaN
    finishnode              5     NaN    NaN
```

1.4 Determine if Graph is Bipartite:

Use BFS to determine that a graph is bipartite, and return the relevant partitions. A bipartite graph is a graph that has nodes you can divide into two sets, A and B, with each edge in the graph connecting a node in A to a node in B.

Create and plot a directed graph using a new m-file. The corresponding graph is presented in Figure Exp1.6. The partitioned graph is shown in Figure Exp1.7.

```
s = [1 1 1 1 2 2 4 5 6 7 8];
t = [2 3 6 8 5 10 6 6 10 3 10];
g = digraph(s,t);
plot(g);
```

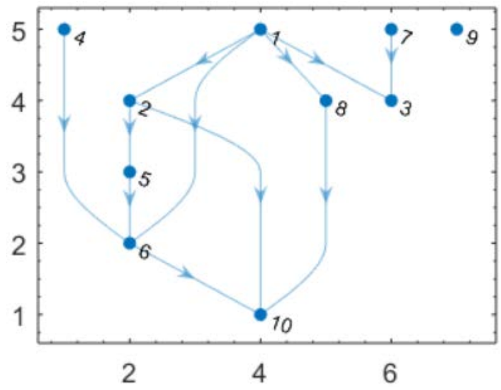


Figure Exp1.6
Directed bipartite graph for BFS

Use a BFS on the graph to determine if it is bipartite, and if so, return the relevant partitions. Just write the following code into the m-file.

```
events = {'edgetonew', 'edgetodiscovered', 'edgetofinished'};
T = bfsearch(g, 1, events, 'Restart', true);
partitions = false(1, numnodes(g));
is_bipart = true;
is_edgetonew = T.Event == 'edgetonew';
ed = T.Edge;

for ii=1:size(T, 1)
    if is_edgetonew(ii)
        partitions(ed(ii, 2)) = ~partitions(ed(ii, 1));
    else
        if partitions(ed(ii, 1)) == partitions(ed(ii, 2))
            is_bipart = false;
            break;
        end
    end
end
```

Results can be sown by writing **'is_bipart'** and **'partitions'** in the command window.

```
is_bipart =

    1
```

```
partitions =

     0     1     1     0     0     1     0     1     0     0
```

The graphical output of the bipartite graph can be visualized by executing the following code in the same m-file.

```
plot(g, 'Layout', 'layered', 'Source', find(partitions));
```

2. dfsearch:

2.1 Perform Depth-First Graph Search:

Write the following code in a new m-file and run the code. This will present and plot a graph on the figure window.

```
s = [1 1 1 1 2 2 2 2 2];
t = [3 5 4 2 6 10 7 9 8];
G = graph(s,t);
plot(G);

v = dfsearch(G,7);
```

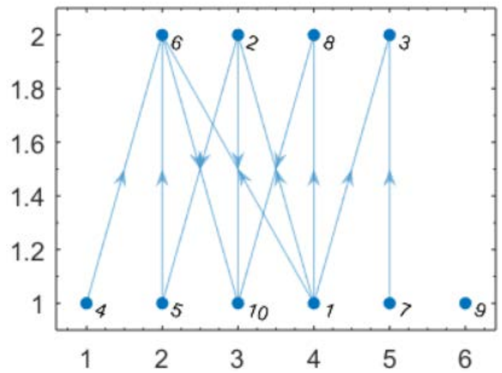


Figure Exp1.7
Directed bipartite graph

Draw the graph in your report. The result in 'v' is presented as:

```
v =

     7
     2
     1
     3
     4
     5
     6
     8
     9
    10
```

2.2 Depth-First Graph Search with All Events:

Take a new m-file and write the following codes to create and plot a directed graph. *Draw the graph in your report.*

```
A = [0 1 0 1 1 0 0;
     0 0 0 0 0 0 0;
     0 0 0 1 0 1 1;
     0 0 0 0 0 1 0;
     0 0 0 0 0 0 0;
     0 0 0 0 0 0 0;
     0 0 0 0 0 0 0];
G = digraph(A);
plot(G);
```

Perform a depth-first search on the graph starting at node 3. Specify 'allevents' to return a table containing all of the events in the algorithm.

```
T = dfsearch(G,3,'allevents');
```

The results in ‘T’ are presented below.

```
T =

        Event            Node        Edge
    _______________      ____    ____________

    startnode              3     NaN     NaN
    discovernode           3     NaN     NaN
    edgetonew            NaN       3       4
    discovernode           4     NaN     NaN
    edgetonew            NaN       4       6
    discovernode           6     NaN     NaN
    finishnode             6     NaN     NaN
    finishnode             4     NaN     NaN
    edgetofinished       NaN       3       6
    edgetonew            NaN       3       7
    discovernode           7     NaN     NaN
    finishnode             7     NaN     NaN
    finishnode             3     NaN     NaN
```

2.3 Depth-First Graph Search with Multiple Components:

Take a new m-file. Perform a depth-first search of a graph with multiple components, and then highlight the graph nodes and edges based on the search results. Create and plot a directed graph. This graph has two weakly connected components.

```
s = [1 1 2 2 2 3 4 7 8 8 8 8];
t = [3 4 7 5 6 2 6 2 9 10 11 12];
G = digraph(s,t);
p = plot(G,'Layout','layered');

events = {'edgetonew','edgetodiscovered','edgetofinished','startnode'};
T = dfsearch(G,4,events,'Restart',true);

% highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetonew'), 'EdgeColor', 'g')
% highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetofinished'), 'EdgeColor', 'k')
% highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetodiscovered'), 'EdgeColor', 'm')
highlight(p,T.Node(~isnan(T.Node)),'NodeColor','r')
```

Draw the graph and result in 'T' in your report.

2.4 Remove Cycles from Graph:

To make a directed graph acyclic by reversing some of its edges. Take a **new m-file** and create and plot a directed graph.

```
s = [1 2 3 3 3 3 4 5 6 7 8 9 9 9 10];
t = [7 6 1 5 6 8 2 4 4 3 7 1 6 8 2];
g = digraph(s,t);
plot(g, 'Layout', 'force')
```

Perform a depth-first search on the graph, flagging the 'edgetodiscovered' event. This event corresponds to edges that complete a cycle.

```
[e, edge_indices] = dfsearch(g, 1, 'edgetodiscovered', 'Restart', true);
% T = dfsearch(g, 1, 'edgetodiscovered', 'Restart', true);
```

Use ***flipedge*** to reverse the direction of the flagged edges, so that they no longer complete a cycle. This removes all cycles from the graph. Use isdag to confirm that the graph is acyclic.

```
gnew = flipedge(g, edge_indices);
isdag(gnew)
```

Plot the new graph and highlight the edges that were flipped.

```
p = plot(gnew, 'Layout', 'force');
highlight(p,'Edges',findedge(gnew,e(:,2),e(:,1)),'EdgeColor','r');
```

Now, draw the graph and results in your report.

3. Implementing BFS to find shortest path in a grid:

This code executes breadth first search. This simulation is meant to mimic a robot navigation problem in a grid and planning a path around obstacles. The user defines the obstacles, goal, and starting position. This simulation is set up for a 11 x 11 grid that ranges from (0, 0) to (10, 10).

Now, write the code in a new m-file. Study the code to understand, run the code, and observe the results as the behavior of the BFS algorithm.

Code:

```

clear all
%close all
clf
clc

% defining the grid size...
xx = 10; % 0 - 10 ...
yy = 10; % 0 - 10 ...
```

```
% Defining the obstacles...
obstacles(1,:)=[0,3];
obstacles(2,:)=[1,1];
obstacles(3,:)=[2,1];
obstacles(4,:)=[2,3];
obstacles(5,:)=[3,1];
obstacles(6,:)=[3,2];
```

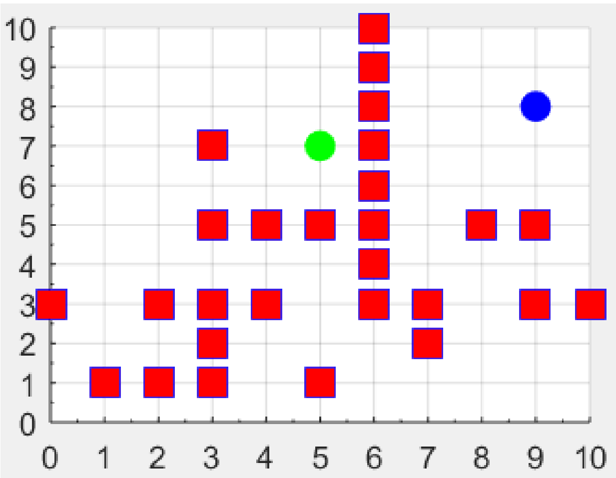
```
obstacles(7,:)=[3,3];
obstacles(8,:)=[3,5];
obstacles(9,:)=[3,7];
obstacles(10,:)=[4,3];
obstacles(11,:)=[4,5];
obstacles(12,:)=[5,1];
obstacles(13,:)=[5,5];
obstacles(14,:)=[6,3];
obstacles(15,:)=[6,4];
obstacles(16,:)=[6,5];
obstacles(17,:)=[6,6];
obstacles(18,:)=[6,7];
obstacles(19,:)=[6,8];
obstacles(20,:)=[6,9];
obstacles(21,:)=[6,10];
obstacles(22,:)=[7,2];
obstacles(23,:)=[7,3];
obstacles(24,:)=[8,5];
obstacles(25,:)=[9,3];
obstacles(26,:)=[9,5];
obstacles(27,:)=[10,3];

% Defining starting and goal locations...
startingPosition=[5,7];
goal=[9,8];
```

```
% Define the colors of nodes...
obstacleColor=[1,0,0];  %red
nodeColor=[0,1,0];      %green
expandColor=[0,0,0];    %black
goalColor=[0,0,1];      %blue
pathColor=[0,1,1];      %cyan

% Plotting the grid and obstacles...
s = scatter(obstacles(:,1), obstacles(:,2), 150, ...
    obstacleColor, 'filled', 's', 'MarkerEdgeColor', 'b');
grid on;
% grid monor;
% grid(gca,'minor')
set(gca, 'YMinorTick','on', 'XMinorTick','on')
axis([0 10 0 10]);
hold on;

% Plotting the goal position...
scatter(goal(1,1), goal(1,2), 100, goalColor, 'filled');

% Initializing variables...
pathCount=1;  % Keeps track of the current node in the bfs_queue set...
tempCount=1;    % Keeps track of the end of the bfs_queue set...

% Initialize the bfs_queue set as ...
% bfs_queue(xPosition, yPosition, parentNode)...
bfs_queue(pathCount,:)=[startingPosition, pathCount];
```

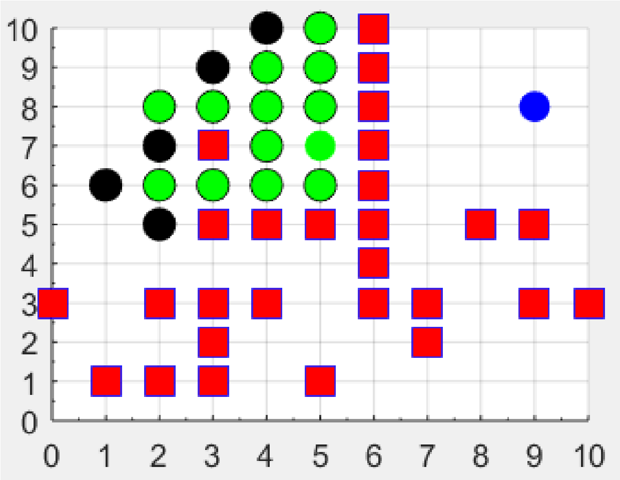
```
%This loop executes until the goal is found
while (~((bfs_queue(pathCount,1)==goal(1,1)) & ...
        (bfs_queue(pathCount,2)==goal(1,2))))

    % Plot the starting node (Source) ...
    scatter(bfs_queue(pathCount,1), bfs_queue(pathCount,2), ...
        100, nodeColor, 'filled');

    % Exploring the neighbour (left right top bottom)...
    for x=-1:1
        for y=-1:1
            % Ensuring the bfs_queue set does not expanded diagonally...
            if (x*y==0)

                % 'failsTest' is used to determine outside the grid,
                % on an obstacle,
                % or it has already been expanded...
                failsTest=0;
                % 'tempNode' is the current node that is trying to
                % be expanded...
                tempNode=[bfs_queue(pathCount,1)+x, ...
                    bfs_queue(pathCount,2)+y, pathCount];

                % Test if the node is outside grid...
                if ( (tempNode(1,1)<0) | (tempNode(1,2)<0) ) | ...
                        ( (tempNode(1,1)>xx) | (tempNode(1,2)>yy) )
                    failsTest=failsTest+1;
                end

                % Test to see if node is already in bfs_queue set...
                if (failsTest<1)
                    for i=1:size(bfs_queue,1)
                        if (tempNode(1,1)==bfs_queue(i,1)) & ...
                                (tempNode(1,2)==bfs_queue(i,2))
                            failsTest=failsTest+1;
                        end
                    end
                end

                % Test to see if node is an obstacle...
                if (failsTest < 1)
                    for i=1:size(obstacles,1)
                        if (tempNode(1,1)==obstacles(i,1)) & ...
                                (tempNode(1,2)==obstacles(i,2))
                            failsTest=failsTest+1;
                        end
                    end
                end
```

```
                % If not fail any tests, add to end of bfs_queue.
                % In BFS, nodes are removed from the end of the bfs_queue,
                % so to make things easy, add new nodes to the end.
                if (failsTest < 1)
                    bfs_queue(pathCount+tempCount,:) = tempNode;
                    scatter(tempNode(1,1), tempNode(1,2), 120, ...
                        expandColor, 'filled');
                    tempCount=tempCount+1;
                end
            end
        end
    end

    % Increment to the next node.
    % Also decrement tempCount as it is a position in the bfs_queue
    % Set relative to pathCount
    pathCount=pathCount+1;
    tempCount=tempCount-1;
    pause(.1);
end

%Initialize a counter
i=1;

%Trace back through the parent nodes to receover the path
while ~(pathCount==1)
    path(i,:)=[bfs_queue(pathCount,1),bfs_queue(pathCount,2)];
    pathCount=bfs_queue(pathCount,3);
    i=i+1;
end

%Add the start position to the path
path(i,:)=startingPosition;

%Plot the path
plot(path(:,1),path(:,2));
scatter(path(:,1), path(:,2), 60, pathColor, ...
    'filled', 'MarkerEdgeColor', 'b');
```

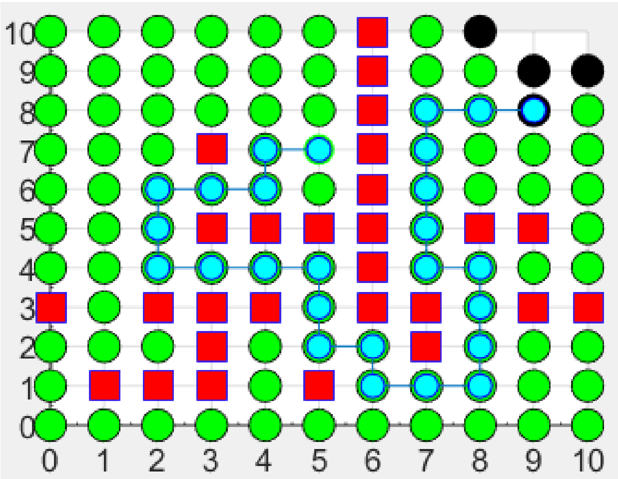
The corresponding graph and final results are presented below.



(11x11) grid with obstacles (red squires), Source node (Green dot), and goal node (Blue dot).



(11x11) grid with obstacles (red squires), Source node (Green dot), goal node (Blue dot), explored nodes (green dots with black border), and to be explored nodes (black dots).



(11x11) grid with obstacles (red squires), Source node (Green dot), and goal node (Blue dot), explored nodes (green dots with black border), to be explored nodes (black dots), and shortest final path (Cyan dots with double borders).

4. Implementing DFS to find shortest path in a grid:

This code executes depth first search. This simulation is meant to mimic a robot navigation problem in a grid and planning a path around obstacles. The user defines the obstacles, goal, and starting position. This simulation is set up for a 11 x 11 grid that ranges from (0, 0) to (10, 10).

Now, write the code in a new m-file. Study the code to understand, run the code, and observe the results as the behavior of the DFS algorithm.

Code:

```

clear all
% close all
clf
clc

% defining the grid size...
xx = 10; % 0 - 10 ...
yy = 10; % 0 - 10 ...
```

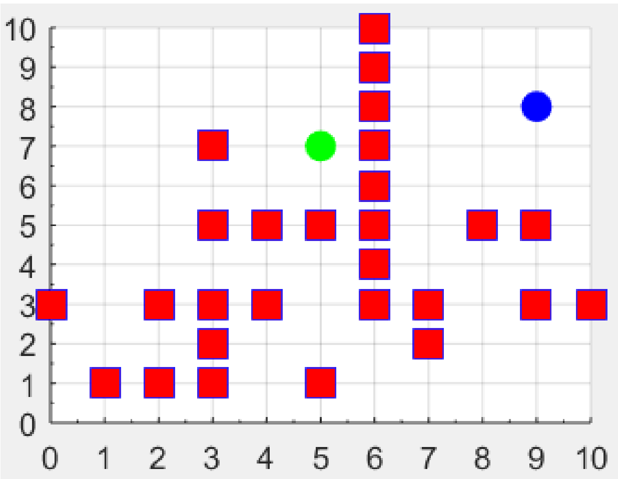
```matlab
% Defining the obstacles...
obstacles(1,:)=[0,3];
obstacles(2,:)=[1,1];
obstacles(3,:)=[2,1];
obstacles(4,:)=[2,3];
obstacles(5,:)=[3,1];
obstacles(6,:)=[3,2];
obstacles(7,:)=[3,3];
obstacles(8,:)=[3,5];
obstacles(9,:)=[3,7];
obstacles(10,:)=[4,3];
obstacles(11,:)=[4,5];
obstacles(12,:)=[5,1];
obstacles(13,:)=[5,5];
obstacles(14,:)=[6,3];
obstacles(15,:)=[6,4];
obstacles(16,:)=[6,5];
obstacles(17,:)=[6,6];
obstacles(18,:)=[6,7];
obstacles(19,:)=[6,8];
obstacles(20,:)=[6,9];
obstacles(21,:)=[6,10];
obstacles(22,:)=[7,2];
obstacles(23,:)=[7,3];
obstacles(24,:)=[8,5];
obstacles(25,:)=[9,3];
obstacles(26,:)=[9,5];
obstacles(27,:)=[10,3];
```

```matlab
% Defining starting and goal locations...
startingPosition=[5,7];
goal=[9,8];

%Define the colors for plotting the results
obstacleColor=[1,0,0];  %red
nodeColor=[0,1,0];      %green
expandColor=[0,0,0];    %black
goalColor=[0,0,1];      %blue
pathColor=[0,1,1];      %cyan

% Plotting the grid and obstacles...
s = scatter(obstacles(:,1), obstacles(:,2), 150, ...
    obstacleColor, 'filled', 's', 'MarkerEdgeColor', 'b');
grid on;
% grid monor;
% grid(gca,'minor')
set(gca, 'YMinorTick','on', 'XMinorTick','on')
axis([0 10 0 10]);
hold on;

% Plotting the goal position...
scatter(goal(1,1), goal(1,2), 100, goalColor, 'filled');

% Initializing variables...
pathCount=1;  % Keeps track of the current node in the bfs_queue set...

% Initialize the bfs_queue set as ...
% bfs_queue(xPosition, yPosition, parentNode)...
dfs_stack(pathCount,:)=[startingPosition, pathCount];
```

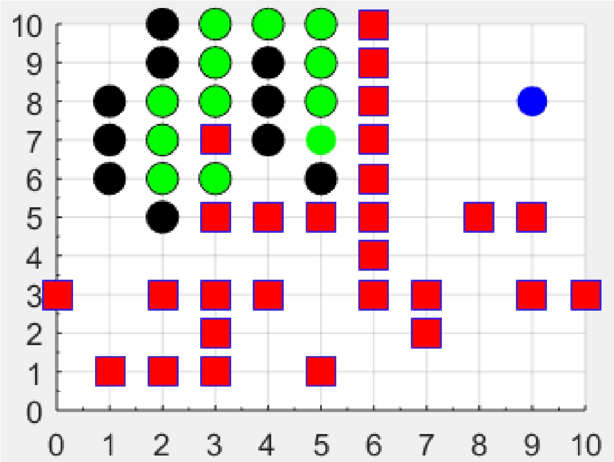
```
% This loop executes until the goal is found...
while (~((dfs_stack(pathCount, 1) == goal(1,1)) & ...
        (dfs_stack(pathCount, 2) == goal(1,2))))

    % Plot the starting node (Source) ...
    scatter(dfs_stack(pathCount, 1), dfs_stack(pathCount, 2), ...
        100, nodeColor, 'filled');

    % Exploring the neighbour (left right top bottom)...
    for x=-1:1
        for y=-1:1
            % Ensuring the bfs_queue set does not expanded diagonally...
            if (x*y==0)

                % 'failsTest' is used to determine outside the grid,
                % on an obstacle,
                % or it has already been expanded...
                failsTest=0;
                % 'tempNode' is the current node that is trying to
                % be expanded...
                tempNode=[dfs_stack(pathCount,1)+x, ...
                    dfs_stack(pathCount,2)+y, pathCount];

                % Test if the node is outside grid...
                if ( (tempNode(1,1)<0) | (tempNode(1,2)<0) ) | ...
                        ( (tempNode(1,1)>xx) | (tempNode(1,2)>yy) )
                    failsTest=failsTest+1;
                end

                % Test to see if node is already in dfs_stack set...
                if (failsTest<1)
                    for i=1:size(dfs_stack,1)
                        if (tempNode(1,1)==dfs_stack(i,1)) & ...
                                (tempNode(1,2)==dfs_stack(i,2))
                            failsTest=failsTest+1;
                        end
                    end
                end

                % Test to see if node is an obstacle...
                if (failsTest<1)
                    for i=1:size(obstacles,1)
                        if (tempNode(1,1)==obstacles(i,1)) & ...
                                (tempNode(1,2)==obstacles(i,2))
                            failsTest=failsTest+1;
                        end
                    end
                end

                % If no fail by any tests, add to beginning of dfs_stack
                % set. In DFS nodes are removed from the
                % beginning of the dfs_stack set, so to make things easy
                % new nodes are added to the beginning of the stack.
                if (failsTest<1)
                    if pathCount<size(dfs_stack,1)
                        dfs_stack(size(dfs_stack,1)+1,:)=[0,0,0];
                        prevNode=dfs_stack(pathCount+1,:);
```

```
                        for i=(pathCount+2):(size(dfs_stack,1))
                            nextNode = prevNode;
                            prevNode = dfs_stack(i,:);
                            dfs_stack(i,:) = nextNode;
                        end
                    end
                    dfs_stack(pathCount+1,:) = tempNode;
                    scatter(tempNode(1,1), tempNode(1,2), 120, ...
                        expandColor, 'filled');
                end
            end
        end
    end

    % Increment to the next node...
    % Decrement tempCount as it is a position in the dfs_stack...
    % Set relative to pathCount...
    pathCount = pathCount+1;
    pause(.1);
end

%Initialize a counter...
i = 1;

%Trace back through the parent nodes to receover the path...
while ~(pathCount==1)
    path(i,:) = [dfs_stack(pathCount,1), dfs_stack(pathCount,2)];
    pathCount = dfs_stack(pathCount,3);
    i = i+1;
end

%Add the start position to the path
path(i,:) = startingPosition;

%Plot the path
plot(path(:,1),path(:,2));
scatter(path(:,1), path(:,2), 60, pathColor,'filled', ...
    'MarkerEdgeColor', 'b');
```

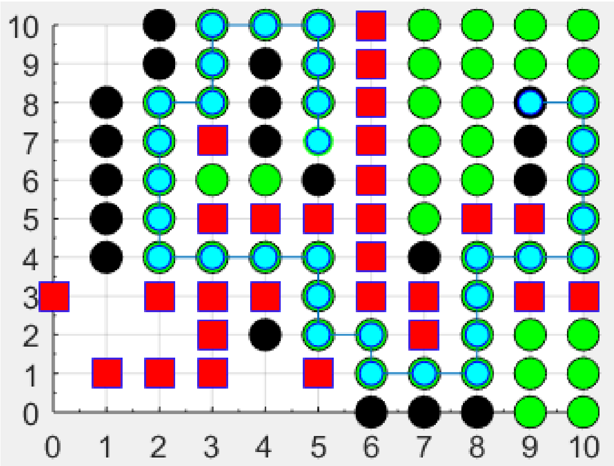
The corresponding graph and final results are presented below.



(11x11) grid with obstacles (red squires), Source node (Green dot), and goal node (Blue dot).



(11x11) grid with obstacles (red squires), Source node (Green dot), goal node (Blue dot), explored nodes (green dots with black border), and to be explored nodes (black dots).

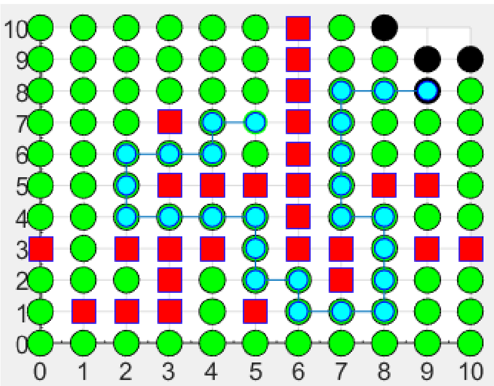


(11x11) grid with obstacles (red squires), Source node (Green dot), and goal node (Blue dot), explored nodes (green dots with black border), to be explored nodes (black dots), and shortest final path (Cyan dots with double borders).

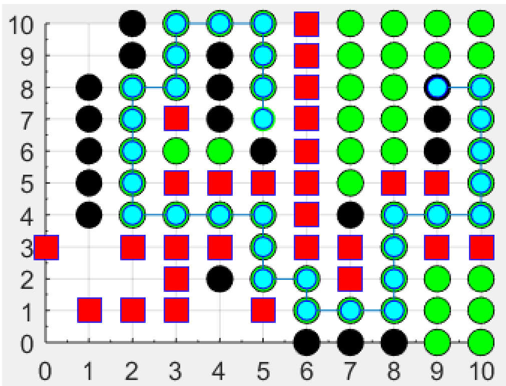
RESULT/COMMENTS:

Now comparing the BFS and DFS result side by side:

BFS result



DFS result



Run all the codes sequentially, observe all the outputs, and write comments in the report form.

Now modify the programs of the BFS and DFS for the same grid and obstacle positions and fill-up the following table with necessary comparative values/results. The result table is presented in Table Exp1.1.

Table Exp1.1: Comparative result table (will be completed by the students)

Comparison points	**BFS**	**DFS**
Execution time		
Number of nodes of the found path		
Path length		
Number of turns		
Number of iteration		
Number of explored nodes		
Max use of the Queue and Stack		
Number of nodes in the Queue or Stack at the end		
Other point 1 (if any)		
Other point 2 (if any)		
Other point 3 (if any)		

INSTRUCTIONS:

1. Run all the codes and show to invigilator/instructor.
2. Observe each of the results and provide your comments in the report form.
3. Fill up the report form properly and take signature from instructor.

EXPERIMENT NO.: 01
TITLE: INDING SHORTEST PATH USING BREADTH FIRST SEARCH (BFS) AND DEPTH FIRST SEARCH (DFS)

DATE OF EXPERIMENT & SUBMISSION:
Batch & Section:

Student ID: ________________________ Student Name: ___________________________________

The student was active/attentive during the experiment: (☐ X ☐ Y ☐ Z).

Marks from the instructor: (☐ X ☐ Y ☐ Z).

Instructor Name: ___________________________ *Instructor Signature & Date:* ____________

Objectives of the experiment:

Inputs / Outputs / Graphs with appropriate title:

Comments (overall understanding):