e-Yantra Robotics Competition (eYRC-2017)
Feeder Weeder
Task 1A - Description
Introduction to Mazes and Path Planning

Path planning is one of the important challenges in robotic technology. Generally in a path planning there is a starting point called **Start Node** and a finish point denoted as **Finish Node**. In this theme, robots will frequently need to find a path from Start Node to and End Node.

The aim of this task is to find a solution path for a given maze from Start Node to Finish Node.

This document is divided into three parts. Section 1, Section 2 and Section 3.

Section 1 deals with explanation of Grids and Mazes. It explains how mazes can be represented as 2D arrays of numbers and other concepts related to mazes.

Section 2 deals with Graphs. Graphs have been explained in brief in this section, along with the adjacency matrix representation of Graphs.

Section 3 will explain the problem statement to be solved in detail.

In order to attempt the task effectively, it is recommended that you read thoroughly through all three sections sequentially.

1. Understanding Grids and Mazes

In this section, we will first learn about what mazes are and how they are constructed.
We start with a **Grid**. Figure 1 shows a representation of a 5x5 grid where first number represents the number of rows and the second number represents the number of columns in the Grid.

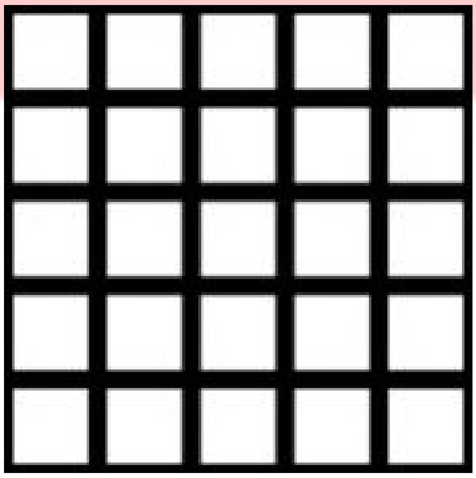


Figure 1: Grid

The smallest building element of a Grid is defined as a **Cell.** A Cell is defined as a square with 4 sides where a **Wall (** represented by bold black lines around the cell**)** may or may not exist at each of the sides. A Grid can be converted into a **Maze** by carving out passages through the Grid. An example maze carved out of Figure 1 is shown in Figure 2.

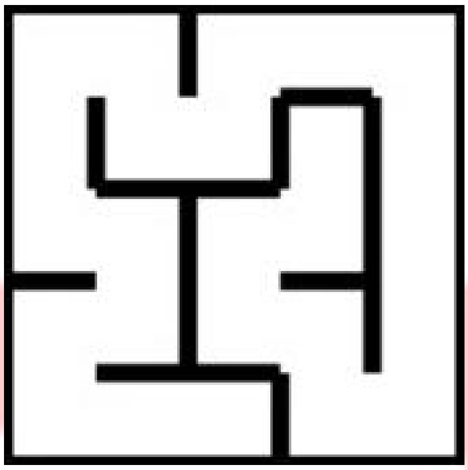


Figure 2: Example Maze

As we have converted the Grid of Figure 1 into a Maze, this maze also has dimensions of 5x5, hence there are 25 cells in each of them.

For example, consider the following cells in Figure 3.

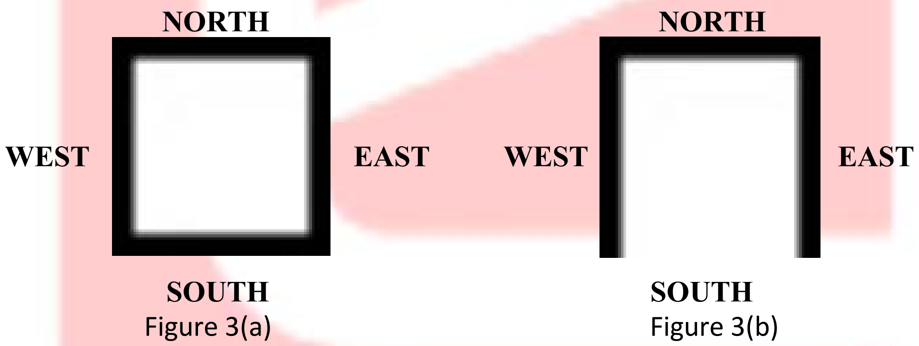
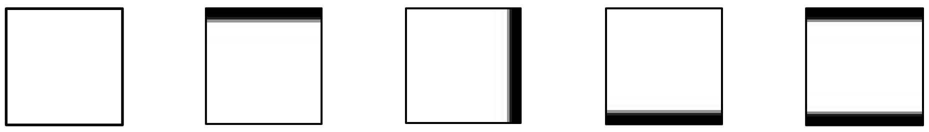


Figure 3: Two Variants of Cells

As shown in Figure 3, the dark bands indicate presence of walls. NORTH, SOUTH, EAST and WEST are called **directions** in the Cell. The first Cell in Figure 3(a) has no open passages in either of the 4 directions of the Cell while the second Cell given in Figure 3(b) has an open passage in SOUTH while the passages to the other three directions are blocked by walls. Since each Cell has four sides we can have a total of 2^4=16 different ways in which a cell may be represented. Each of the way a Cell is represented is called a **Type**. Hence, we have 16 Types of Cells. Two of them are shown in Figure 3 and rest of the 14 configurations are shown in Figure 4.



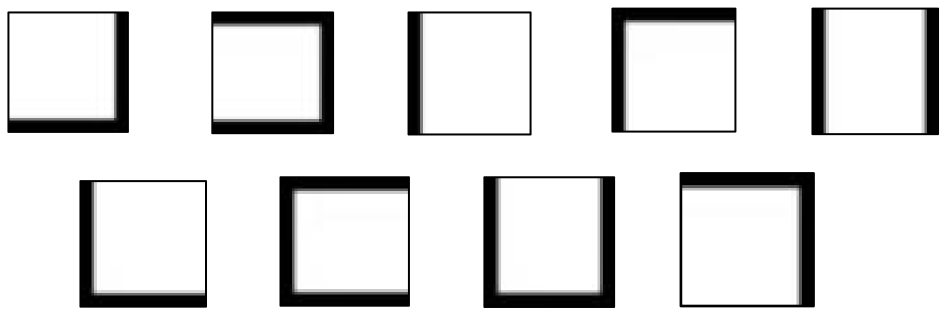


Figure 4: Different types of Cells

A maze as given in Figure 2, is nothing but a collection of different types of cells as given in Figure 4.

1.1 Unique Numbering to represent Cells.

As discussed above, there are a total of 16 different Types of cells. Each Type of Cell can be represented using a unique C**ell Number** between 0 to 15. That number is assigned on the basis of wall configuration of the Cell.

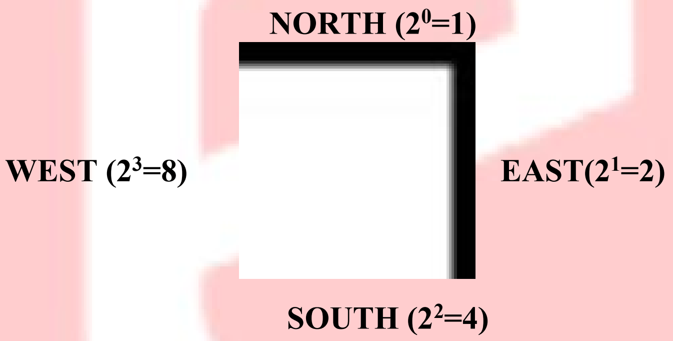
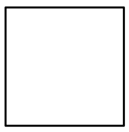


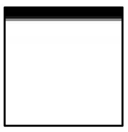
Figure 5: Cell Weights

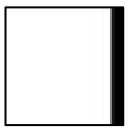
As shown in Figure 5, we have assigned a binary weight to each of the directions of the Cell. The Cell Number for each Type of Cell can be calculated by adding the weights for the directions in which a wall exists and ignoring the weights of the directions where wall doesn't exist.

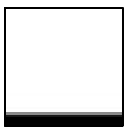
For example, the Cell number for Cell in Figure 5 is Weight of North wall(W_N)+Weight of East wall(W_E) = 1+ 2 = 3.

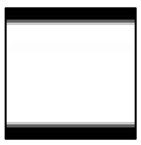
Similarly the Cell numbers for the rest of the Cell types can be calculated as given in Figure 6

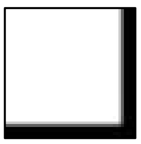
 0 + 0 + 0 + 0 = 0

 0 + 0 + 0 + 1 = 1

 0 + 0 + 2 + 0 = 2

 0 + 4 + 0 + 0 = 4

 0 + 4 + 0 + 1 = 5

 0 + 4 + 2 + 0 = 6

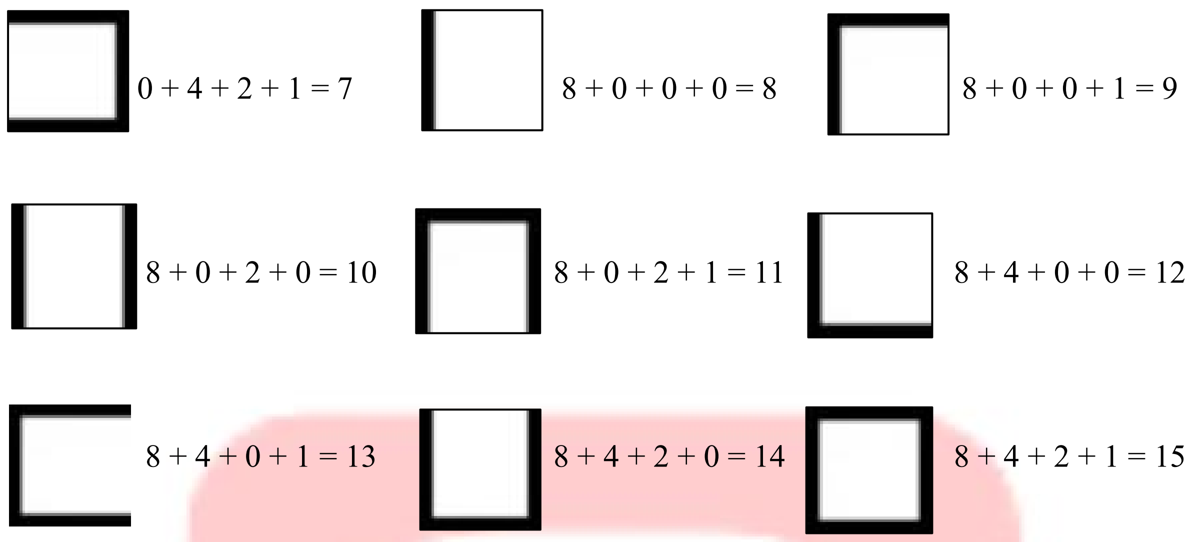


Figure 6: Cell Numbers

Note that Cell Numbers uniquely identify the different Types of Cells. Cell Numbers are significant because these allow us to represent a maze as a 2D 5x5 array of 25 elements in which each array element represents a Cell of the maze by its corresponding Cell Number.

Consider the maze in Figure 7

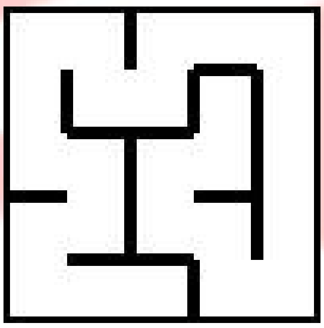


Figure 7: Example Maze

This maze can be represented using a 2D array using the Cell Numbers given in Table 1.

```
int maze[5][5] = {{9, 3, 9, 5, 3},{10, 12, 6, 11, 10},{12, 3, 9, 6,
10},{ 9, 6, 12, 3, 10},{12,  5,  7, 12,  6}}
```

Table 1: 2D Array representation of Example Maze

Henceforth, all mazes in Task 1A will be represented as 2D arrays.

1.2 Indexing of Cells in a Maze

We have seen that the complete maze can be represented using a 2-D array.Using this 2-D array representation, Cells are indexed in a maze as coordinates as shown in Figure 8.

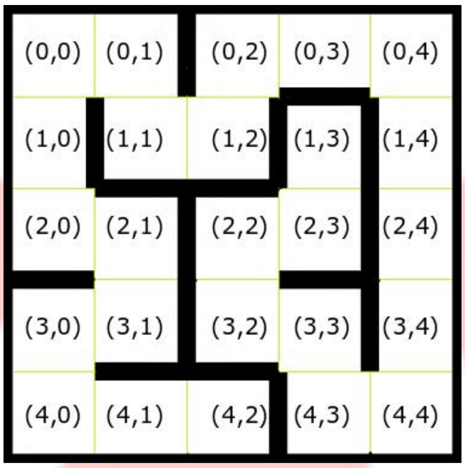


Figure 8: Coordinate Indexing System

Each of the Cells in the maze has a coordinate which can be used to access that Cell.
For example if we want to know the Cell Number of (1,0) we can access it from the array index maze[1][0] of the maze described in Table 1.

There is a second way of indexing the Cells in the maze. This indexing is called **sequential indexing** (shown in Figure 9). This indexing is particularly useful when we are constructing graphs (discussed later).

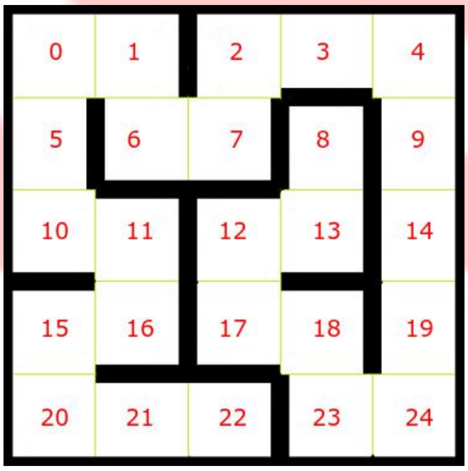


Figure 9: Sequential Indexing System

So far, we learnt all about the mazes and how to represent a maze in form of a 2D array. So lets move on to the next section in which we will deal with Graphs and Path Planning.

2 . Graphs and Path Planning

The aim of Task 1A is to find a path in a maze from a **Start Node** to **Finish Node.**

For finding the path from start node to finish node, the maze needs to be converted into a graph.

Mazes in general are represented as **undirected graphs** so as to depict all traversable neighbors.

An undirected graph is a set of objects (called vertices or nodes) that are connected together with the edges, where all the edges are bi-directional. The edges are line between pair of nodes. An undirected graph is sometimes called an undirected network and is illustrated in Figure 10.

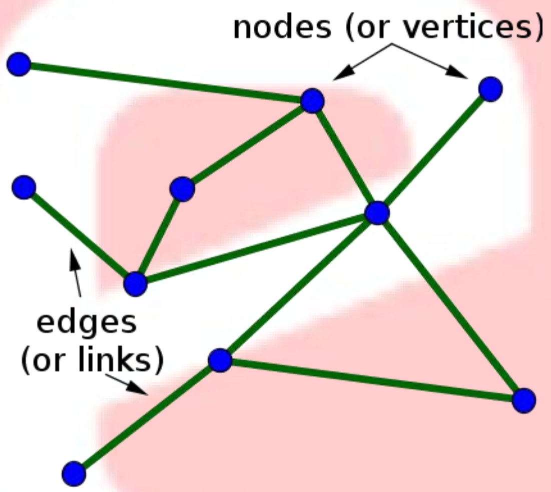


Figure 10: Undirected Graph or Network (Reference)

Undirected graph can be represented using Adjacency Matrix. This matrix representation will help us in developing algorithm for path planning.

2.1 Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array of adjacency matrix be represented by a 2-D array adj[V][V], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j and 0 represent absence of the edge between the nodes. Adjacency matrix for undirected graph is always symmetric i.e. transpose of the matrix is same as the original adjacency matrix. Consider the following example graph in Figure 11.

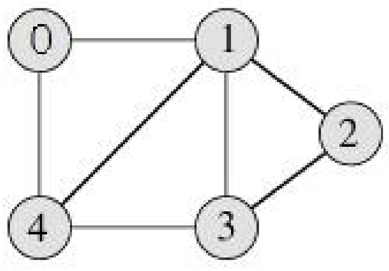
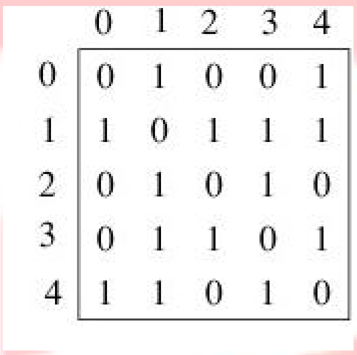


Figure 11: Example Graph (Reference)

The adjacency matrix for the example graph is shown in Figure 12:



$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

Figure 12: Adjacency Matrix (Reference)

You can read more about graphs here.

We have explained mazes in great detail in the previous sections. A maze can also be thought of as an undirected graph consisting of 25 nodes/ vertices where each node represents a cell in the maze. The cells have to be indexed using a sequential indexing system as explained in Section 1.2.

Consider the following example maze as shown in Figure 13.

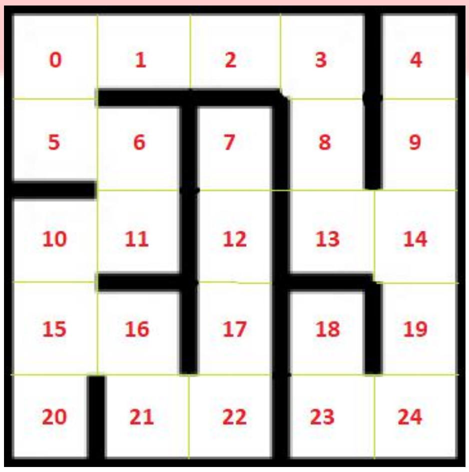


Figure 13:Example Maze

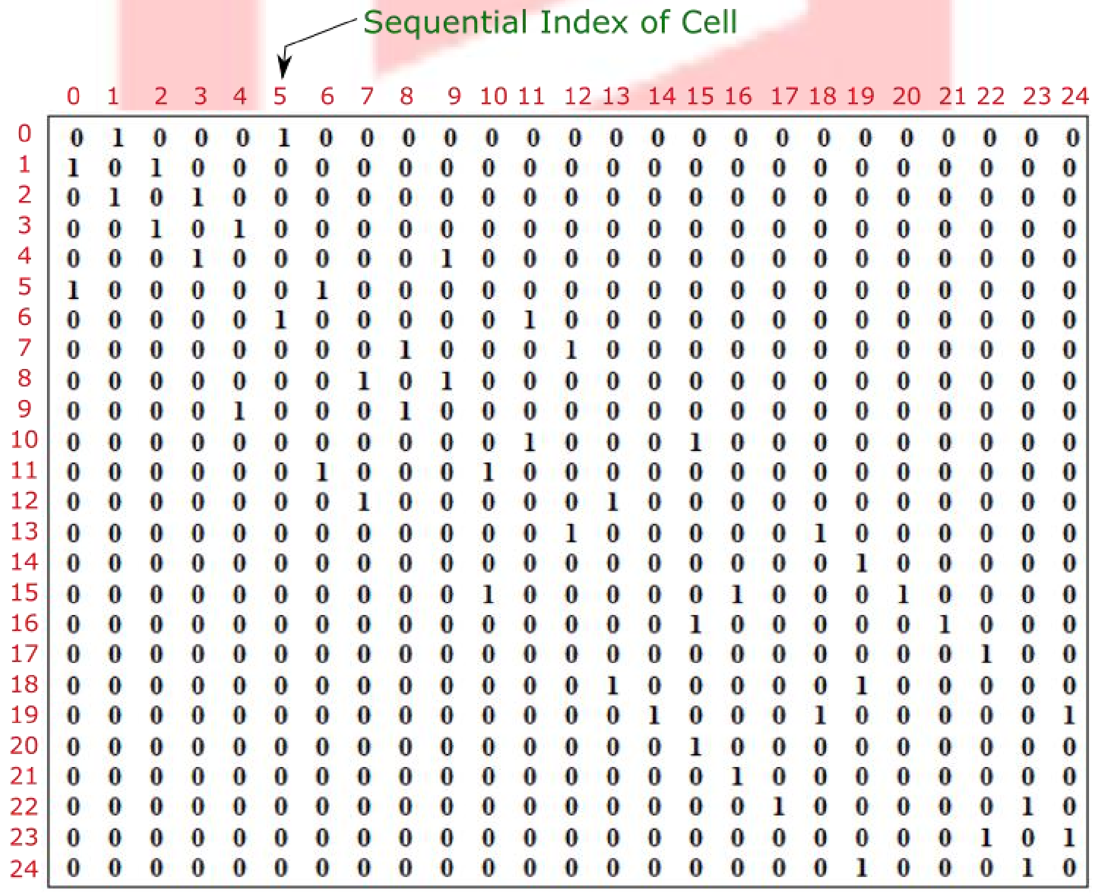
The maze can be represented as a 2D array as given in Table 2

```
int maze[5][5] = {{9, 5, 5, 3, 11},{12, 3, 11, 10, 10},{9, 6, 10, 12,
2},{8, 3, 10, 11, 10},{14, 12, 6, 12, 6}}
```

Table 2: Maze 2D Array

As shown in Table 2, Cell 0 has a Cell Number of 9 (1+8) and it means that thise Cell is open towards East and South i.e. towards Cell 1 and Cell 5.This will be represented in the adjacency matrix by placing a 1 in column 1 and 5 and 0 in the other columns of Row 0.

Considering another example, if we see Cell 11 it has a Cell Number of 6 (1+2) which means that this Cell is open towards North and West i.e. towards Cell 6 and Cell 10 and hence for row 11, there will be 1 in column 6 and 10 and 0 elsewhere. Similar observation can be made for the other row also. It is possible to construct an adjacency matrix for the maze as given in Figure 14.


Sequential Index of Cell

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	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
6	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0

Figure 14:Adjacency Matrix for Maze

Once we have created an adjacency matrix, it is possible to apply various path finding algorithms like Dijkstra, A*, BFS, DFS etc to the adjacency matrix to find the path from Start node to Finish node.

Now that you have grasped the concepts related to graphs, let us move on to the next section where you will be implementing graph construction and path planning in C programming language.

3 . Problem Statement

In this task you'll be required to construct two functions buildgraph() and findpath().

1. buildGraph() - buildGraph function will be used to convert a maze (in form of a 2D array) into a graph (in form of adjacency matrix).
2. findPath() - The graph constructed in buildGraph function will be passed as argument to the findPath() function which will then find a path from Start Node to Finish Node using that graph.

Before we move on to programming, there are 3 data structures that need to be explained. Navigate to the folder MazeSolve and open the project MazeSolve.cbp project in Code::Blocks IDE and then open the src.c file in the project.

The code contains the following structures as given in Code Section 1

```c
#define X 25
#define Y 5

/** Structure Name: Maze
 ** Purpose: Used to store maze information retrieved from file.
 **          The maze is represented in form of a 2D array wrapped in a structure.
 **/
struct Maze
{
    int maze_array[Y][Y];
};

/** Structure Name: Graph
 ** Purpose: Used to store the Graph constructed by the buildGraph() function.
 **          The graph is represented in form of a 2D array wrapped in a structure.
 **/
struct Graph
{
    _Bool graph_array[X][X];
};

/** Structure Name: Path_Array
 ** Purpose: Used to store the final path Array constructed by the findPath()
             function.
 **          The path is represented in form of a 1D array wrapped in a structure.
 **/
struct Path_Array
{
       int path[X];
};
```

Code Section 1: Data Structures

The first data structure we use is **struct Maze** which wraps a 2D 5x5 array into a structure. The 2D array is in the same form as earlier explained in Section 1 above.

The second data structure is **struct Graph** which wraps a 2D 25x25 adjacency matrix array into a structure.

The third data structure is **struct Path_Array.** It contains the solution path in a 1D array of 25 elements which is wrapped in a structure.

The two functions which you need to edit are given in Code Section 2 and 3 below:

```
/** Function Name: buildGraph()
 ** Input: Accepts a Maze structure as input
 ** Output: Returns a Graph structure as output
 ** Purpose: Constructs a adjacency matrix graph using the maze information passed
as argument
 ** Example Call: struct Graph matrix = buildGraph(maze);
 **/
struct Graph buildGraph(struct Maze maze)
{
    struct Graph adj_matrix;



    return adj_matrix;
}
```

The **buildGraph()** function takes a Maze structure as argument and returns Graph structure which should contain the adjacency matrix for the maze.
Suppose we have the following maze structure:

struct Maze maze = {{{9, 5, 5, 3, 11},{12, 3, 11, 10, 10},{9, 6, 10, 12, 2},{8, 3, 10, 11, 10},{14, 12, 6, 12, 6}}};

If we pass this maze structure as input to the buildGraph() function, the following should be the output:

```
{ { 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 },
  { 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 },
  { 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 },
  { 0 0 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 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 },
  { 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 0 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 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 },
  { 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 },
  { 0 0 0 0 1 0 0 0 0 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 1 0 0 0 1 0 0 0 0 0 0 0 0 0 },
  { 0 0 0 0 0 0 1 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 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 },
  { 0 0 0 0 0 0 0 0 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  1  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0 },
    { 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  1  0  0  0  0 },
    { 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0 },
    { 0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  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  0  0  0  1  0 },
    { 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1 },
    { 0  0  0  0  0  0  0  0  0  0  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  0  0  0  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  1  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  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  1  0  0  0  1  0 }    }
```

You need to write the code for this function so that it returns the required output as explained.

Code Section 2: buildGraph

Consider the following Example in Figure 15.

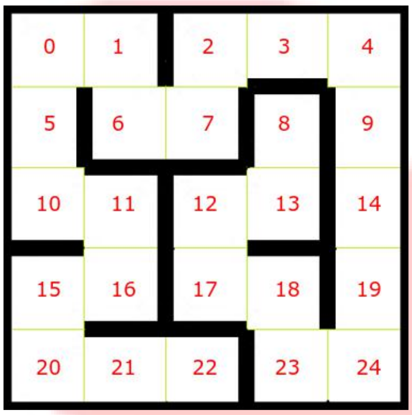


Figure 15: Example Maze

We want to traverse from cell index 0 to cell index 24.
The solution path for such traversal will be as follows: 0,1,6,7,2,3,4,9,14,19,24.

```
/** Function Name: findPath
 ** Input: graph - Graph structure
 **        src - start point of path
 **        dst - end point of path
 ** Output: Returns a Path_Array structure which contains the path in the maze from
           src to dst.
 ** Example Call: struct Path_Array path = findPath(graph, 0, 24)
 **/
struct Path_Array findPath(struct Graph graph, int start, int finish)
{
       struct Path_Array path = init_path();



      return path;
}
```

This function accepts a Graph structure and the Start and Finish node of the path and returns a Path_Array structure.
A Path_Array structure is created and all elements are initialized to -1 using the **init_path()** function. You can look up the **init_path()** function in the **src.c** code file.

The output array of this function will look like the following for the input maze in Figure 15
{ 0, 1, 6, 7, 2, 3, 4, 9, 14, 19, 24, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,}

You need to write the code for this function so that it returns the required output as explained.

Code Section 3: findPath()

3.1 Main Function

In order to understand how the path is calculated, we need to examine two more functions. Consider Code Section 4 given below:

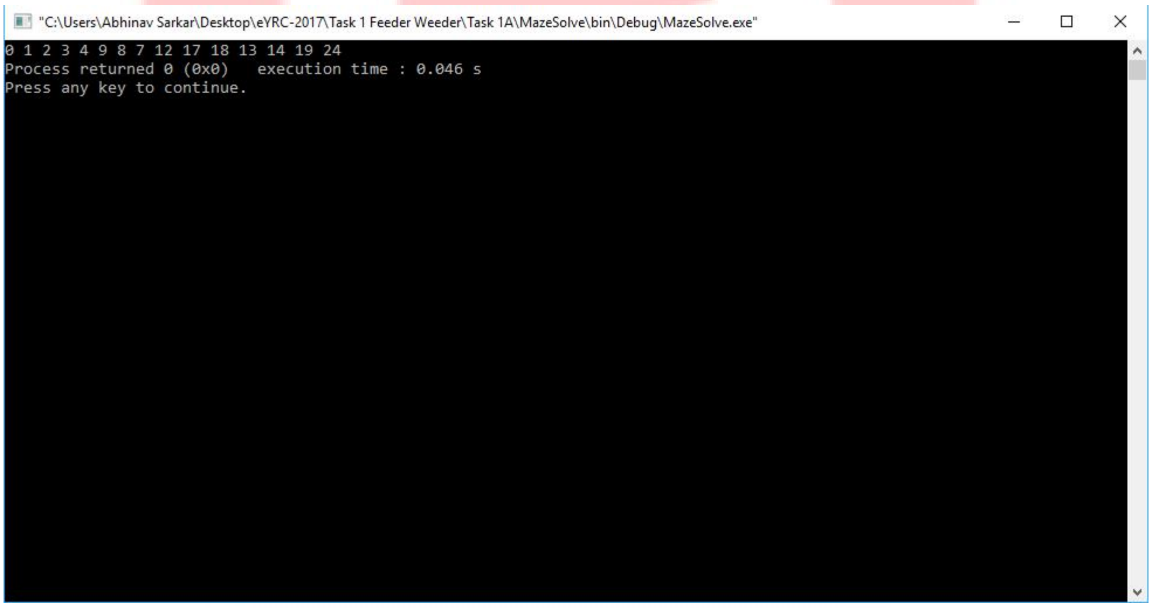
```
/** Function Name: main_function
 ** Input: maze - Maze structure
 **        start - start point of path
 **        finish - end point of path
 ** Output: Returns a Path_Array structure which contains the path in the maze from
start to finish.
 ** Logic: This function first constructs a graph from the maze passed as argument
and then finds out
 **            path from start to finish. The path is returned as a Path_Array
structure.
 ** Example Call: struct Path_Array path = main_function(maze,0,24);
 **/
struct Path_Array main_function(struct Maze maze, int start, int finish)
{
    struct Graph adjacency_matrix = buildGraph(maze);
    struct Path_Array path = findPath(adjacency_matrix, start, finish);
     return path;
}

/**Function Name: main
Input: None
Output: None
Logic: This function first reads the Test Case file and constructs Maze structure.
       Then it constructs Path_Array structure by calling main_function()
       Finally it prints the constructed path array.
Example Call: struct Path_Array path = main_function(maze,0,24);
**/

int main()
{
    struct Maze maze;
    int start = 0, finish = 24;
    maze = parseFile("..\\..\\Task 1A\\TestCases\\maze_0.txt");
    struct Path_Array path = main_function(maze, start, finish);
    for(int i = 0; i < X; i++)
    {
        if(path.path[i] == -1)
        {
            break;
        }
        printf("%d ", path.path[i]);
    }
    return 0;
}
```

When the program is executed, the main function is called first. It creates a Maze structure by reading the test case file specified as argument to the **ParseFile**() functio. (Refer **src.c** file to understand what the **parseFile()** function does). This maze structure is then passed as argument to the **main_function()** along with Start and Finish of the path.

The **main_function()** first constructs a graph adjacency matrix by calling the **buildGraph()** function and then passes the adjacency matrix as argument to the **findPath()** function.
Finally it will return the final Path_Array structure, which is then printed by the program.

Code Section 4: Main

Once you have completed the code for the buildGraph() and findPath() functions, run your program
The output should be similar to one given in Figure 16.


"C:\Users\Abhinav Sarkar\Desktop\eYRC-2017\Task 1 Feeder Weeder\Task 1A\MazeSolve\bin\Debug\MazeSolve.exe"

```
0 1 2 3 4 9 8 7 12 17 18 13 14 19 24
Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

Figure 16:Output

You are only allowed to make changes in the buildGraph() and findPath() functions. You are not allowed to make any changes in the rest of code in the src.c code file. However you are allowed to declare utility functions for your own use in the space provided in the src.c file. However these functions can only be called in buildGraph() and findPath() functions.

Once you have successfully run your program, you need to test if your solution works for all 5 test cases provided to you in the TestCases folder.

Further instructions to test the program are given in the ***Testing Your Solution.pdf.***