Problem Statement:

A maze is a path or collection of paths, typically from an entrance to a goal. The word is used to refer both to branching tour puzzles through which the solver must find a route, and to simpler non-branching ("unicursal") patterns that lead unambiguously through a convoluted layout to a goal. Generally, a Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e.,maze [0][0] and destination block is lower rightmost block i.e.,maze[N-1][N-1]. Possible move is forward and down. Gray blocks are dead ends. Now design a suitable algorithm to find the path from source to destination. You have to use a backtracking approach to find the path. Also keep in mind that, Brute Force solution is not allowed.

System Requirements:

Processor: Inter Core i5 Quad Core and above (Recommended)

RAM: 6GB

Operating System: Windows 10 or Windows 11

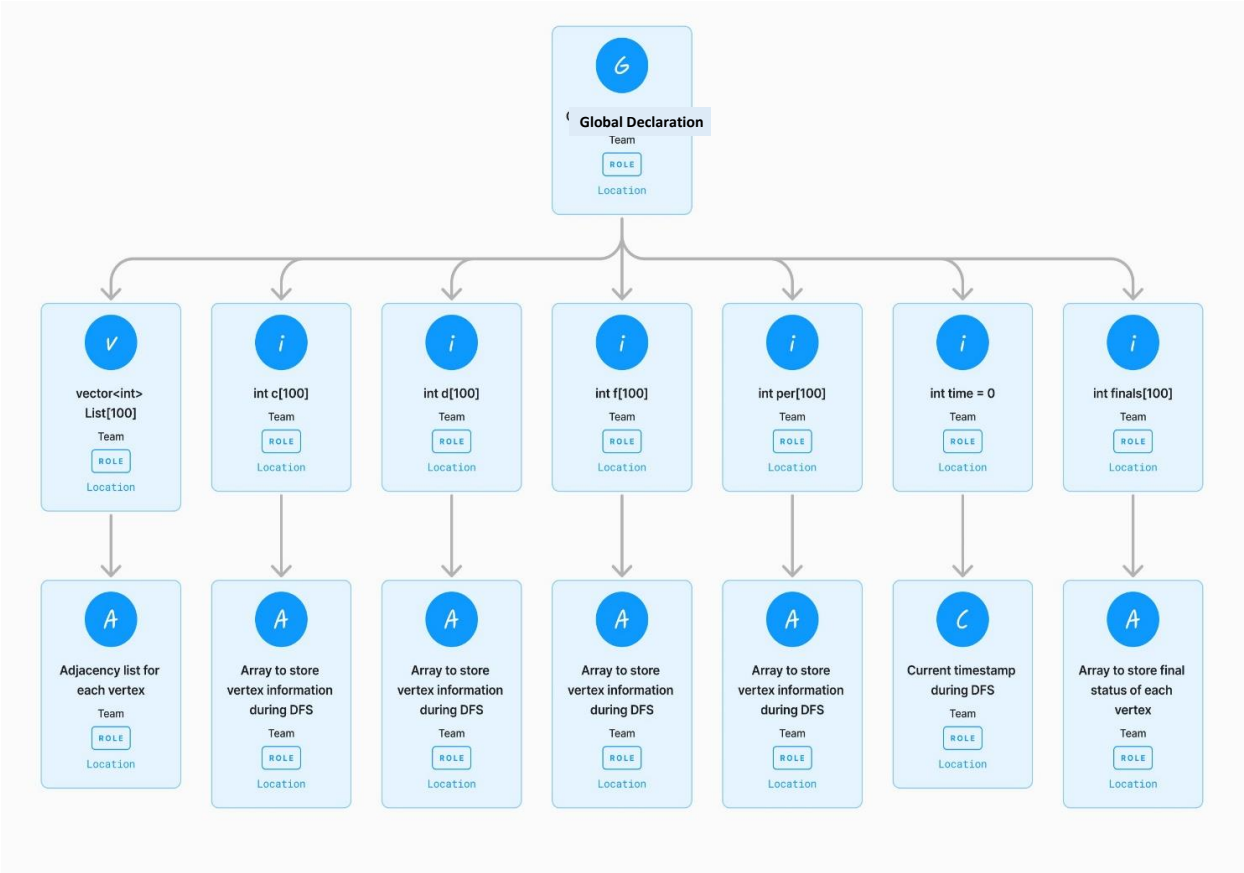
IDE: CodeBlocks (GNU G++ 19)

System Design:

Global Declarations:

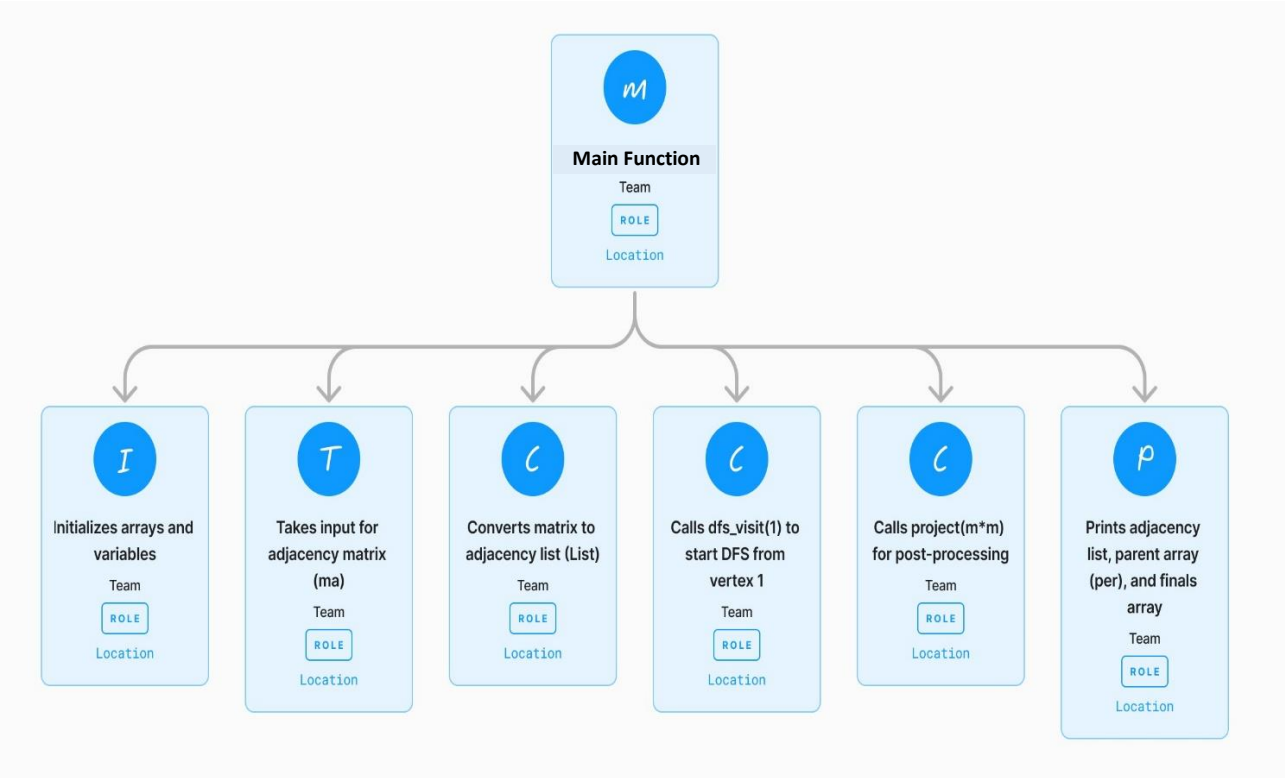
We first globally declare a vector array which will hold the adjacency list for each vertex. Then following that we declare 5 more normal arrays where c is for color, d is for discovery time, f is for finishing time, p is for parent tracking and finally the last array, A will have the final path of the maze. Also, we initialize the starting time, t as 0.

```
vector<int>List[100];
int c[100];
int d[100];
int f[100];
int p[100];
int t=0;
int A[100];
```



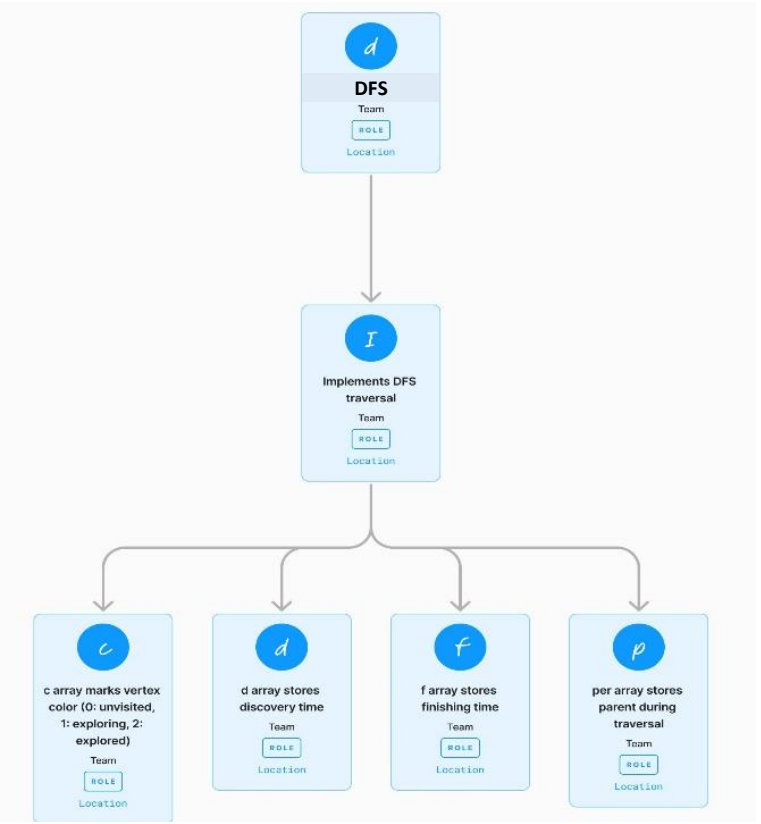
Main Function:

- Arrays A, c, d, f, and p are initialized to specific values. The program prompts the user to input the size of the maze (n), and the size is read from the standard input.
- The program takes input for an n x n matrix representing the maze. The matrix contains binary values (0 for a blocked path, 1 for an open path). The starting point is assumed to be the top-left corner (0,0), and the destination is assumed to be the bottom-right corner (n-1, n-1).
- The matrix is converted into an adjacency list, represented by the array "List". This is done by creating connections between open paths in the matrix. The connections are determined by adjacent open cells in the matrix.
- The DFS function is called with the starting node index (1) and traverses the graph created from the matrix using Depth-First Search and explores all the vertices.
- The BackTrack function is then called which is responsible for backtracking to find the solution path of the maze.
- The program prints the solution path of the maze, which is stored in the array A which is displayed as a matrix.



DFS Function:

The DFS function iterates through each vertex u in the graph and if the color of u is WHITE, initiate a DFS traversal from vertex u by calling DFS_Visit(u). The DFS_Visit(u) marks vertex u as GREY to indicate its discovery. Incrementing the time counter (time) while setting d[u] to the current time, the function iterates through each vertex v adjacent to u (denoted as Adj[u]). If v is WHITE, prev[v] is updated to u and DFS_Visit(v) is called recursively. After exploring all adjacent vertices, the function marks u as BLACK to indicate its finish and thus incrementing the time counter (time) again while setting f[u] to the current time.

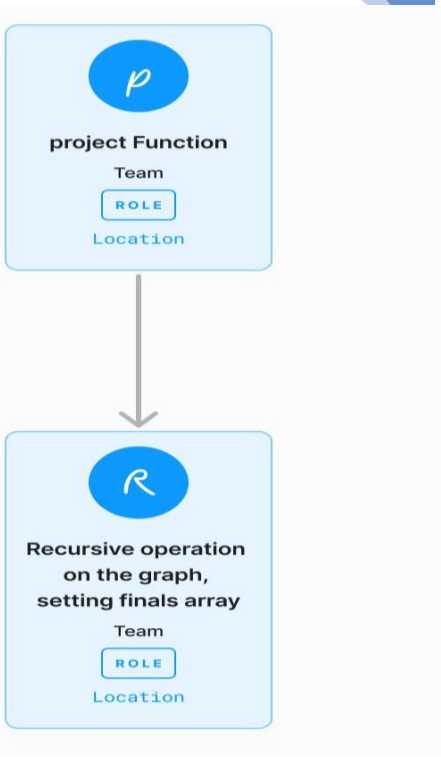


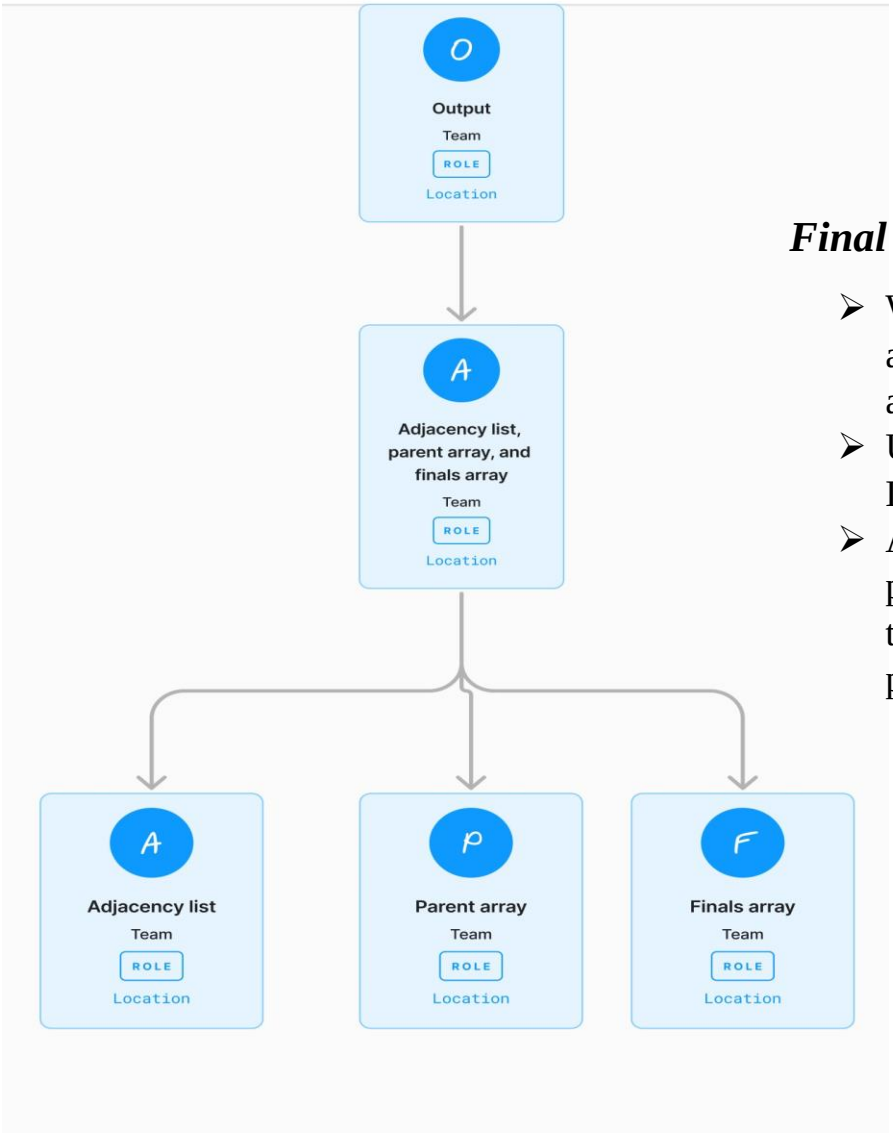

```
DFS(G) // where prog starts
{
for each vertex u ∈ V
{
color[u] = WHITE;
prev[u]=NIL;
f[u]=inf; d[u]=inf;
}
time = 0;
for each vertex u ∈ V
if (color[u] == WHITE)
DFS_Visit(u);
}
DFS_Visit(u)
{
color[u] = GREY;
time = time+1;
d[u] = time;
for each v ∈ Adj[u]
{
if(color[v] == WHITE){
prev[v]=u;
DFS_Visit(v);}
}
color[u] = BLACK;
time = time+1;
f[u] = time;
}
```

BackTrack Function:

This backtracking procedure begins with the parameter m as an input. If m is 0, the recursion is terminated. Otherwise, it gets a value q from the array p at index m, sets the corresponding element in array A to 1, and calls itself recursively with q. This process is repeated until the base case is reached, allowing for retracing a series of decisions or steps. The context and values stored in arrays A and p determine the specific application and purpose of this backtracking algorithm.

```
if (m == 0)
    return 0;
int q = p[m];
    A[m] = 1;
BackTrack(q);
```





Final Output:

- We get 3 separate arrays, adjacency list, parent array and the final path array.
- Using Dfs on the Adjacency List we got the parent array.
- And then backtracking the parent array we finally got the Final array which is the path way of the input maze.

Implementation:

Source Code:

```
#include<iostream>
#include<vector>
using namespace std;
vector<int>List[100];
int c[100];
int d[100];
int f[100];
int p[100];
int t=0;
int A[100];
```

//Backtracks the parent array to give final result using recursion

```
int BackTrack(int m)
{
   if(m==0)return 0;
   int q=p[m];
   A[m]=1;
   BackTrack(q);
}
```

The function begins with a base case check: **"if(m==0)" return 0;** if the input parameter m is equal to 0, the function returns 0 immediately. This serves as the base case for the recursive calls and aids in the recursion's termination. If the base case is not met, **"int q=p[m];"** retrieves the value from the array 'p' at index m and stores it in variable 'q'. **"A[m]=1;"** sets '1' at index m in the final array A. **"BackTrack(q);"** recursively calls 'BackTrack' function with value 'q'.

// DFS traversal starting from vertex m

```
void DFS(int m)
{
   int u=m;
   c[u]=1;
   ++t;
   d[u]=t;
   for(auto x:List[u])
   {
      if(c[x]==0)
      {
         p[x]= u;
         DFS(x);
      }
   }
   f[u]= ++t;
   c[u]=2;
}
```

With **c[u]=1;** the vertex u is marked as currently being explored. Then with **++t; d[u]=t;** we incremented a timestamp counter 't' to record the discovery time of vertex 'u' in the array 'd'. **"for(auto x:List[u])"** iterates through each vertex 'x' in the adjacency list of vertex 'u' where auto takes any type of variable and converts it into the tree. If the color of vertex 'x' (**c[x]**) is 0 (**WHITE**), indicating that it has not been visited, **"p[x]= u;"** sets the predecessor of vertex 'x' to be 'u' and also **"DFS(x);"** is recursively called to explore the unvisited vertex 'x'. At **f[u]= ++t;** incrementation of timestamp counter is done the same way to record the finish time of vertex 'u' in array 'f'. Finally, **c[u]=2;** marks the vertex 'u' as fully explored.

```
int main()
{

// Initializing the path array as zero

   for(int i=0; i<100; i++)
   {
      A[i]=0;
   }

   for(int i=0; i<100; i++)
   {
      c[i]=0;
      d[i]=9999;
      f[i]=9999;
      p[i]=0;
   }

   int m,n;

   cout<<" Enter maze size (should be given n*n size) ";

   cin>>n;

   int adj[n][n];

   cout<<endl<<" 0 is for block and 1 is for Open way "<<endl;

   cout<<" 1st value is the starting point and destination is the last value "<<endl<<endl;

// taking the input matrix

   for(int i=0; i<n; i++)
   {
      for(int j=0; j<n; j++)
      {
         cin>>m;
         adj[i][j]= m;
      }
   }

// introduced a counter k

   int k=1;
```

This function is responsible for the input of n*n matrix. **"for(int i=0; i<n; i++)"** iterates over the rows of the matrix where 'i' represents the current row. **"for(int j=0; j<n; j++)"** iterates over the columns of the matrix as it is a nested loop where 'j' represents the current column. **"cin>>m;"** takes the integer input and **"adj[i][j]= m;"** Assigns the value of m to the element at the i-th row and j-th column in the adjacency matrix.

```
// converting the matrix to tree

   for(int i=0; i<=n; i++)
   {
      for(int j=0; j<n; j++)
      {

         if(adj[i][j]==1)
         {
            if(adj[i][j+1]==1&&j!=n-1)
            {
               List[k].push_back(k+1);
            }
            if(adj[i+1][j]==1)
            {
               List[k].push_back(k+n);
            }
         }
         k++;
      }
   }
```

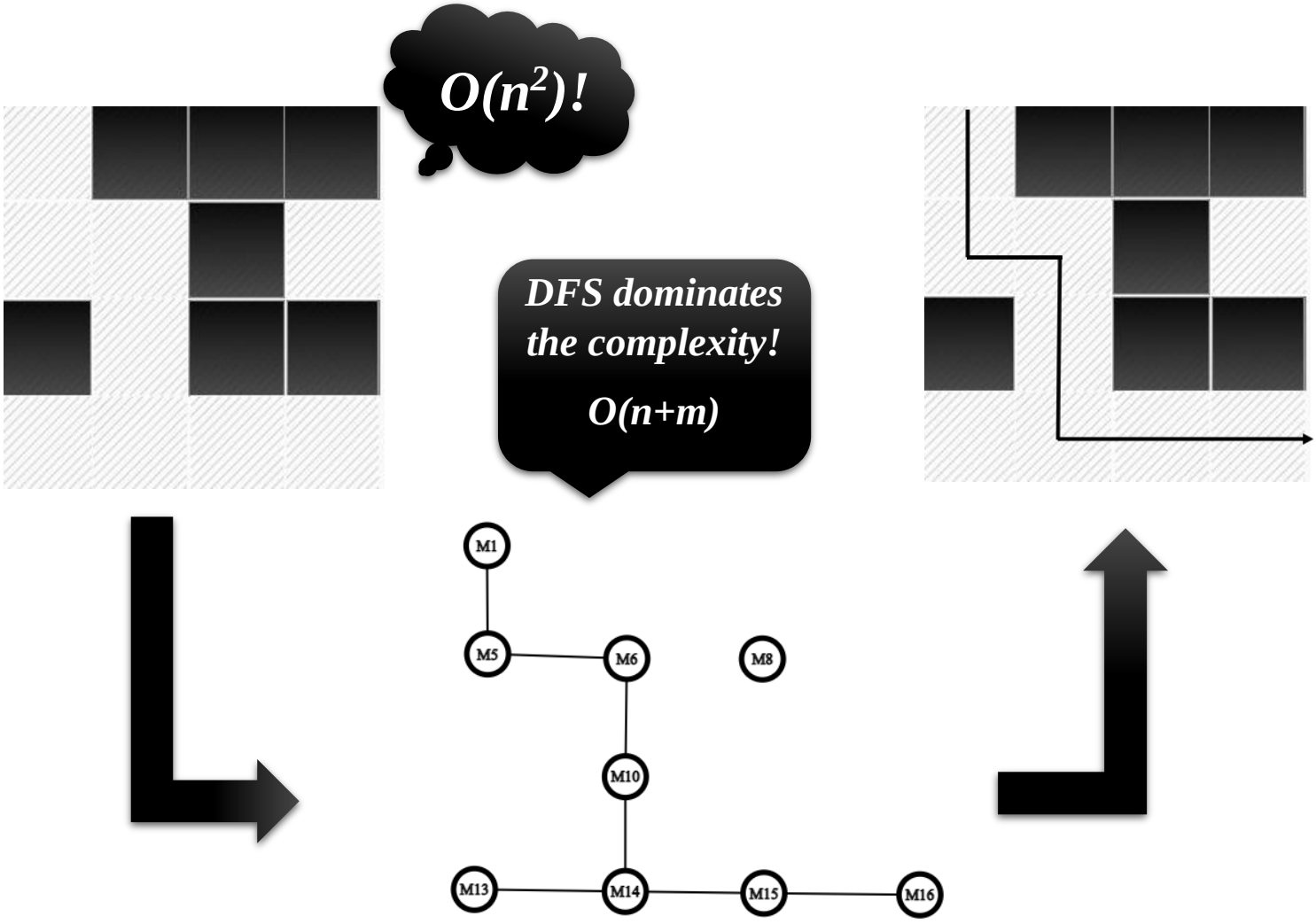
Here the adjacency matrix is converted to adjacency list. **"for(int i=0; i<n; i++)"** iterates over the rows of the matrix where 'i' represents the current row. **"for(int j=0; j<n; j++)"** iterates over the columns of the matrix as it is a nested loop where 'j' represents the current column. **"if(adj[i][j]==1)"** checks the presence of an edge between the vertices 'i' and 'j', 1 indicates presence of edge. **"if(adj[i][j+1]==1&&j!=n-1)"** checks the presence of an edge between the vertices 'j' and 'j+1' and if condition is met, '(k,k+1)' edge is added to the adjacency list 'List[k]'. **"if(adj[i+1][j]==1)"** checks the presence of edge between vertices 'i' and 'i+1' and if condition is met, '(k,k+n)' edge is added to the adjacency list 'List[k]'.

```
//Recursive call of dfs and backtrack

   DFS(1);

   BackTrack(n*n);

   cout<<endl;
   cout<<"-------------------------------------"<<endl;
   cout<<"-------------------------------------"<<endl;
   cout<<" The Solution of this maze "<<endl;

   cout<<endl;

//printing the final path after backtrack

   for(int i=1; i<=n*n; i++)
   {
      cout<<A[i]<<" ";
      if(i%n==0)
      {
         cout<<endl;
      }
   }

}
```

Here we have called 2 recursive functions.
"DFS(1);" which takes the adjacency list and explores all the vertices.
"BackTrack(n*n);" which back tracks the parent array to give the final path of the maze.

Complexity Analysis:

- Let n be the size of the maze, representing the number of vertices in the graph. The code initializes arrays and matrices, which takes O(n^2) time.
- The DFS function then iterates through each vertex in the graph, visiting all reachable vertices from a given starting point.
- In the worst case, where the graph is connected, DFS explores all n vertices, and for each vertex, it examines its adjacency list once.
- Consequently, the overall time complexity of the DFS function is O(n+m), where m is the number of edges in the graph.
- The code also includes a Backtrack function, which backtracks the parent array to construct the final path.
- Since the backtracking process follows the parent pointers from the destination to the source, it takes O(n) time in the worst case.
- Therefore, the overall time complexity of the code is dominated by the DFS traversal, resulting in O(n+m+n), which simplifies to O(n+m) for a connected graph.



Testing Results:

Input 1:

For the input, we first enter the size of the N*N matrix and then we enter binary values for the blocks and paths where 0 represents blocked and 1 represents open path.

```
 Enter maze size (should be given n*n size) 4

 0 is for block and 1 is for Open way
 1st value is the starting point and destination is the last value

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

Output 1:

In the output we get the pathway from the upper left most index (0,0) up to the lower right most index (5,5), replacing all the separated or singled out 1s by 0s as they are not included in the path.

```
------------------------------------
------------------------------------
 The Solution of this maze

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

Input 2:

```
 Enter maze size (should be given n*n size) 8

 0 is for block and 1 is for Open way
 1st value is the starting point and destination is the last value

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

Output 2:

```
------------------------------------
------------------------------------
 The Solution of this maze

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

Future Scope:

Throughout the project we tried to keep the complexity as minimum as possible but there are still limitations to our project such as,

- The whole project only works for a N*N matrix, that is only if the maze has a N*N space, we will be able to find a path out of it using this code.
- The project assumes that the starting point of the maze will always be the upper left most index which is (0,0) and the ending point of the maze will always be at the lower right most index which is (N,N)
- If the input does not have a proper path or is wrong at any single index, the output for the code will give a random or garbage value in the matrix.

Conclusion:

Overall the project was informative and we got to learn new ways of implementing dfs and also learned many more details regarding C++.