TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

MAZEGENERATOR

A PROJECT SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DATA STRUCTURES AND ALGORITHM

Submitted by:
Ankit Shahi (PUL076BEI006)
Bhupendra Chaulagain (PUL076BEI011)
Asmin Silwal (PUL076BEI040)

Submitted to:
Department of Electronics and Computer Engineering
Pulchowk Campus, Institute of Engineering,
Tribhuvan University
Lalitpur, Nepal

March, 2022

ABSTRACT

*This project focuses on generating a **nxn** sized maze using Depth-First Search(DFS) Algorithm. DFS is a algorithm(both recursive and iterative method) for searching all the vertices of a graph or tree data structure. The underlying data structure is graph, which means that we can apply graph theory theorems to analyze maze generation and path finding in a maze. In this project we have generated a maze by applying a randomized DFS to the grid. Use of DFS algorithm provides certainty of every cells being connected to the maze. The algorithm starts at the root cell (selecting some arbitrary cell as the root cell in the case of a grid) and explores as far as possible along each cell before backtracking.*

*Key Words: **Maze Generator,Depth-First Search(DFS) Algorithm,Iterative,Back Tracking***

Contents

1 Introduction

Our course project titled ”MazeGenerator” is about generation of a maze using Depth-First Search(DFS) Traversal Algorithm. We used ”olcPixelGameEngine” library by `https://community.onelonecoder.com/olcpixelgameengine/` for rendering the maze. We used the concepts of graph traversal and stack that were taught to us as part of our course for implementing this project.

2 Objectives

- To learn about Graph Theory and Graph Traversal.
- To learn about Depth-First Search Algorithm.
- To generate a maze by applying Depth-First Search Algorithm.
- To analyze time complexity of DFS algorithm.

3 Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices to be visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path. There are two graph traversal techniques which are as follows:

- Depth First Search
- Breadth First Search

4 Depth First Search (DFS) Traversal Algorithm

DFS Traversal Algorithm is an algorithm which traverses or searches graph or tree data structure starting at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Here, the word backtracking means that when you are moving forward and there are no more nodes to be visited along the current path, you move backwards on the same path to find unvisited nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

5 Algorithm

First Let's discuss how Depth-First Search works in traversal of data in graph. DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal.

5.1 Algorithm For Generalized DFS

```
Step 1 - Define a Stack of size total number of vertices in the graph.
Step 2 - Select any vertex as starting point for traversal. Visit that
         vertex and push it on to the Stack.
Step 3 - Visit any one of the non-visited adjacent vertices of a vertex
         which is at the top of stack and push it on to the stack.
Step 4 - Repeat step 3 until there is no new vertex to be visited from
         the vertex which is at the top of the stack.
Step 5 - When there is no new vertex to visit then use back tracking
         and pop one vertex from the stack.
Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7 - When stack becomes Empty, then produce final spanning tree
         by removing unused edges from the graph
```

5.2 Algorithm For MazeGenerator Using DFS

Although maze can be generated by both recursive and iterative method using DFS algorithm.In caseof recursion method there is disadvantage in a large depth of recursion – in the worst case, the routine may need to recur on every cell of the area being processed, which may exceed the maximum recursion stack depth in many environments. As a solution, the same backtracking method can be implemented with an explicit stack, which is usually allowed to grow much bigger with no harm.

The algorithm used for generation of Maze in our program is:

```
step 1: START
step 2: Create Window
step 3: Define Grid Size(height*width) of Grid and Draw Grid(cells)
step 4: Initialize Stack
        Choose Random Cell and Push it into Stack
        Mark that cell as visited.
        visitedCells=1
step 5: is visitedCells<Grid Size(height*width)
        YES:
        step 1: Create set of unvisited Neighbours using vector
                (Adjacency List)
        step 2: Are there any unvisited Neighbours?
            YES:
            step 1: visit any cell
                    push it to the stack
                    createPath
                    visitedCells++
            step 2: Go to step 5
            NO:
            step 1: pop the stack
            step 2: Go to step 5
        NO:
        step 1: END
```

6 FlowChart

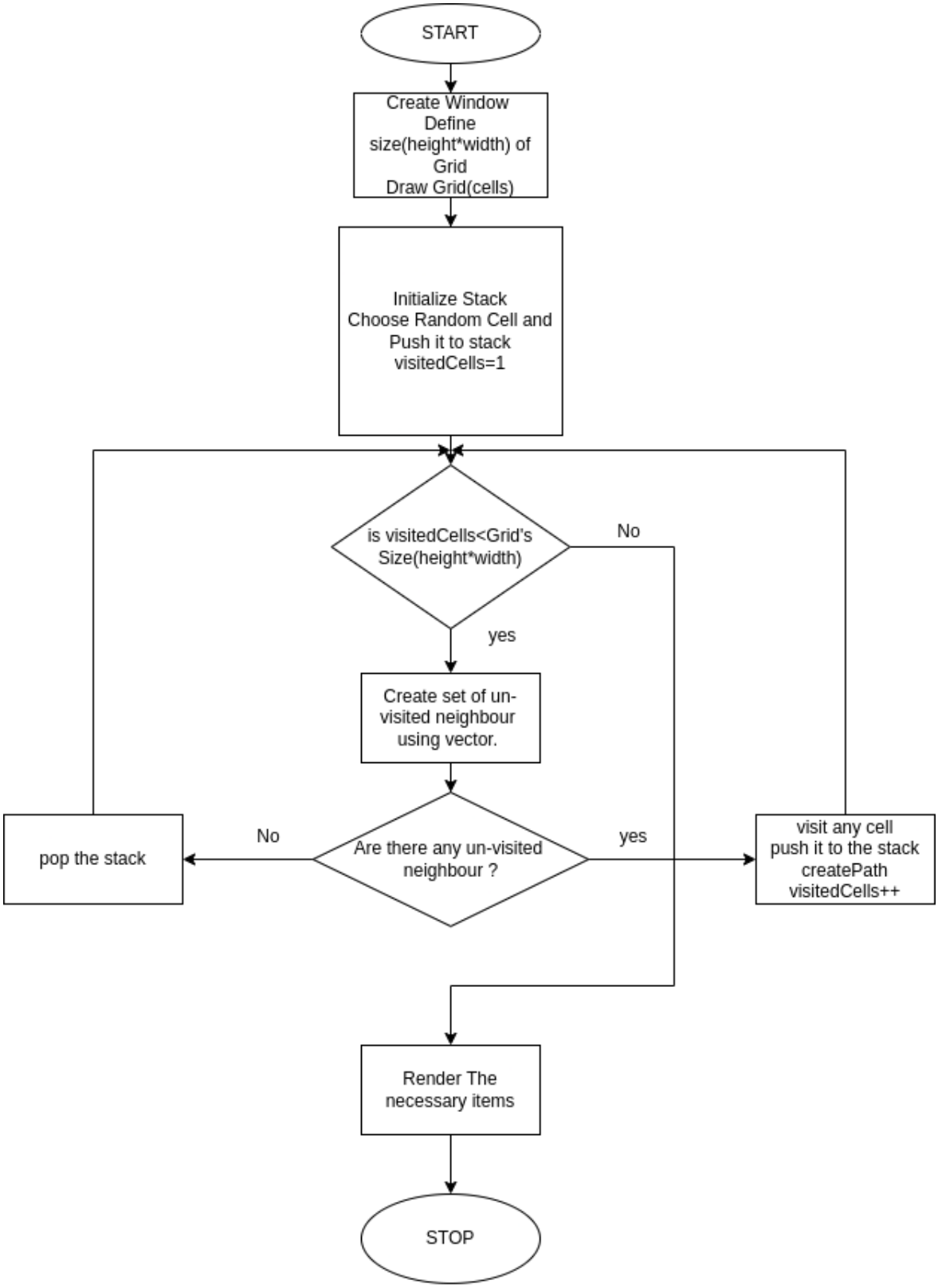

Figure 1: FlowChart of the Process

7 Time Complexity Analysis

In this project, we have traversed entire graph(maze) using DFS algorithm which takes time $O(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges.This is the worst case time complexity for DFS algorithm in our case. We have used adjacency list which stores the neighbours of every node(cell) in the form of vector. In DFS traversal, we explore each node and its neighbours exactly once.

The sum of all edges in the graph is E and each edge in the lists represents only one direction so for two vertices, A and B in an undirected graph; B will be in A's adjacency list and A in B's. The total number of vertices is V. Checking if the vertex has been visited can be done in $O(1)$ time via a set.

There is some constant time work $O(1)$ done irregardless of the number of vertices or edges. i.e work like initializing stack, initializing visited set etc. In the worst case, all vertices have to be visited. Now for the each vertex, the following amount of work is done:
$O(1) + degree(V_j) * O(1)$

where $degree(V_j)$ is the degree of vertex j i.e the number of edges going out of j; Constant work is done on each vertex such as popping the top of the stack hence the $O(1)$ and for each edge, constant work is also done such as checking if edge points to an already visited vertex.

Now putting all together; we have the initial constant work done plus the work done on each vertex described above:

$O(1) + \sum O(1) + degree(V_j) * O(1)$;the summation is on all vertices.
Simplifying the summation, we have:

$O(1) + O(V) + O(deg(V_1)) + O(deg(V_2)) + ...O(deg(V_n))$;the summation on $O(1)$ becomes $O(V)$ because summation is done V times, once for each vertex.

The term $O(deg(V_1)) + O(deg(V_2)) + ...O(deg(V_n))$ is the summation of the degree of each vertex which equals to 2E i.e. $O(E)$. So we have:
$O(1) + O(V) + O(E) = O(V) + O(E)$

8 Source Code

```
#include <stack>
#define OLC_PGE_APPLICATION
#include "olcPixelGameEngine.h"
using namespace std;
#define delay 50ms //ms

class MazeGen : public olc::PixelGameEngine
{
public:
    MazeGen()
    {
        sAppName="MazeGen";
    }
private:
    int *maze;
    enum
    {
        CELL_PATH_N = 0x01,
        CELL_PATH_E = 0x02,
        CELL_PATH_S = 0x04,
        CELL_PATH_W = 0x08,
        CELL_VISITED = 0x10,
    };
    // Algorithm variables
    int  visitedCells;
    stack<pair<int, int>> mStack;
    int  pathWidth;

protected:
    virtual bool OnUserCreate()
    {
        // Maze Parameters
        mazeWidth = 25;
        mazeHeight = 25;
        maze = new int[mazeWidth * mazeHeight];
        // Allocating the memory base on size
        memset(maze, 0x00, mazeWidth * mazeHeight * sizeof(int));
```

```
        pathWidth = 3;

        // A random cell is used to start the maze.
        int x = rand() % mazeWidth;
        int y = rand() % mazeHeight;
        mStack.push(make_pair(x, y));
        maze[y * mazeWidth + x] = CELL_VISITED;
        visitedCells = 1;
        return true;
    }
    virtual bool OnUserUpdate(float elapsedTime)
    {
        this_thread::sleep_for(delay);
        auto offset = [&](int x, int y)
        {
            return (mStack.top().second + y) * mazeWidth +
            (mStack.top().first + x);
        };
        // Maze Algorithm
        if (visitedCells < mazeWidth * mazeHeight)
        {   // Create a set of unvisted neighbours
            vector<int> neighbours;
            // North neighbour
            if (mStack.top().second > 0 && (maze[offset(0, -1)] &
            CELL_VISITED) == 0)
                neighbours.push_back(0);
            // East neighbour
            if (mStack.top().first < mazeWidth - 1 && (maze[offset(1, 0)] &
            CELL_VISITED) == 0)
                neighbours.push_back(1);
            // South neighbour
            if (mStack.top().second < mazeHeight - 1 && (maze[offset(0, 1)] &
            CELL_VISITED) == 0)
                neighbours.push_back(2);
            // West neighbour
            if (mStack.top().first > 0 && (maze[offset(-1, 0)] &
            CELL_VISITED) == 0)
                neighbours.push_back(3);
            if (!neighbours.empty())
```

```
        {
            int next_cell_dir = neighbours[rand() % neighbours.size()];
            switch (next_cell_dir)
            {
                case 0: // North
                    maze[offset(0, -1)] |= CELL_VISITED | CELL_PATH_S;
                    maze[offset(0,  0)] |= CELL_PATH_N;
                    mStack.push(make_pair((mStack.top().first + 0),
                    (mStack.top().second - 1)));
                    break;

                case 1: // East
                    maze[offset(+1, 0)] |= CELL_VISITED | CELL_PATH_W;
                    maze[offset( 0, 0)] |= CELL_PATH_E;
                    mStack.push(make_pair((mStack.top().first + 1),
                    (mStack.top().second + 0)));
                    break;

                case 2: // South
                    maze[offset(0, +1)] |= CELL_VISITED | CELL_PATH_N;
                    maze[offset(0,  0)] |= CELL_PATH_S;
                    mStack.push(make_pair((mStack.top().first + 0),
                    (mStack.top().second + 1)));
                    break;

                case 3: // West
                    maze[offset(-1, 0)] |= CELL_VISITED | CELL_PATH_E;
                    maze[offset( 0, 0)] |= CELL_PATH_W;
                    mStack.push(make_pair((mStack.top().first - 1),
                    (mStack.top().second + 0)));
                    break;
            }
            visitedCells++;
        }
        else
        {
            mStack.pop();
        }
    }
```

```
        FillRect(0, 0, ScreenWidth(), ScreenHeight(), olc::BLACK);

        for (int x = 0; x < mazeWidth; x++)
        {
            for (int y = 0; y < mazeHeight; y++)
            {
                for (int py = 0; py < pathWidth; py++)
                    for (int px = 0; px < pathWidth; px++)
                    {
                        if (maze[y * mazeWidth + x] & CELL_VISITED)
                            Draw(x * (pathWidth + 1) + px, y * (pathWidth + 1) +
                            py, olc::WHITE); // Draw Cell
                        else
                            Draw(x * (pathWidth + 1) + px, y * (pathWidth + 1) +
                            py, olc::RED); // Draw Cell
                    }
                // Draw passageways between cells
                for (int p = 0; p < pathWidth; p++)
                {
                    if (maze[y * mazeWidth + x] & CELL_PATH_S)
                        Draw(x * (pathWidth + 1) + p, y * (pathWidth + 1) +
                        pathWidth); // Draw South Passage

                    if (maze[y * mazeWidth + x] & CELL_PATH_E)
                        Draw(x * (pathWidth + 1) + pathWidth, y *
                        (pathWidth + 1) + p); // Draw East Passage
                }
            }
        }
        for (int py = 0; py < pathWidth; py++)
            for (int px = 0; px < pathWidth; px++)
                Draw(mStack.top().first * (pathWidth + 1) + px,
                mStack.top().second * (pathWidth + 1) + py,olc::GREEN);
        return true;
    }
};
int main()
{
    srand(clock());
```

```
    MazeGen maze;
    maze.Construct(160, 100, 5, 5);
    maze.Start();
    return 0;
}
```

9 Snippets of Maze Generator

Few Screenshots of Maze Generator

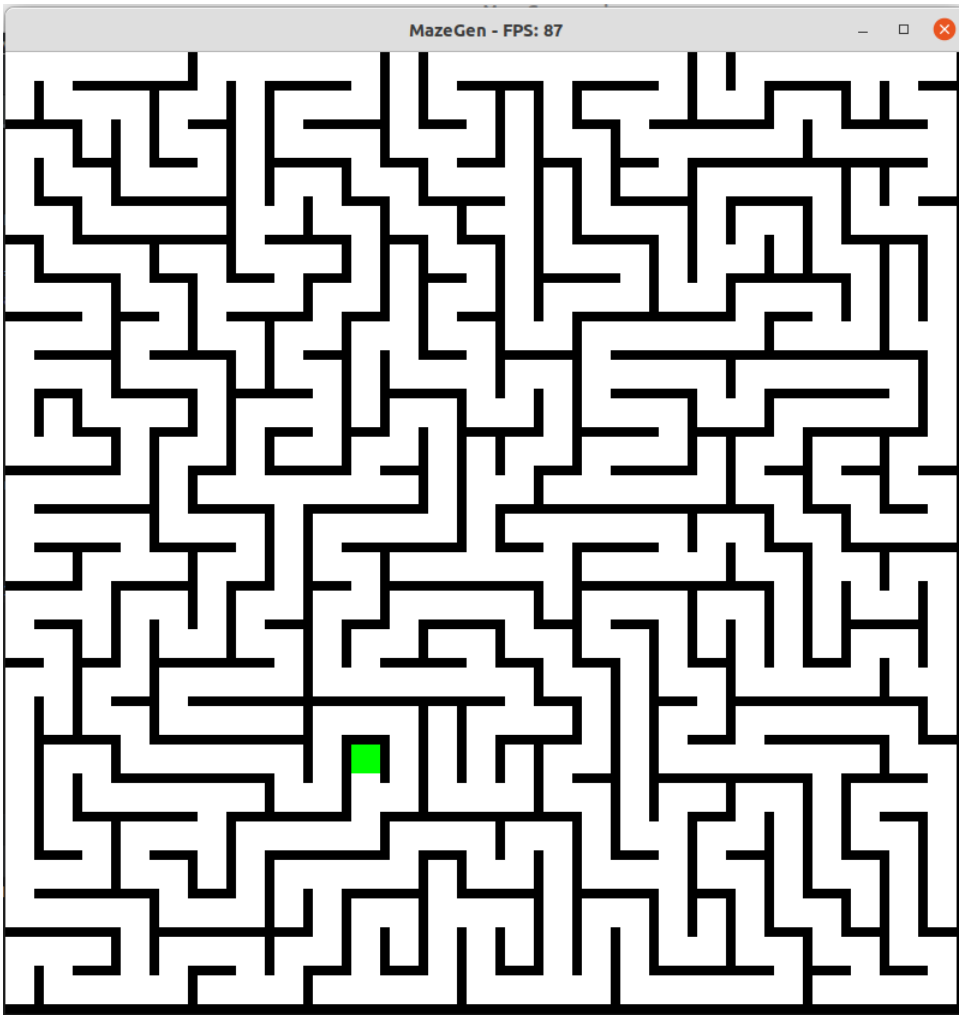

Figure 2: Fully Constructed Maze

Figure 3: Maze being Generated