# CS 5002 Final Project Report: Graph Theory behind The Logics of Randomized Maze Generation

# Introduction

Mazes are a common feature of video games and movies. The generation of randomized mazes has become a subject of interest in computer science, with the aim of producing mazes that are visually appealing and provide a satisfying experience for users. In recent years, a variety of algorithms and techniques have been developed to generate mazes, each with its own advantages and disadvantages.

After learning Chapter 9  in the 5002 class, I wonder if is there any graph theory behind the generation of randomized mazes. During our research, I found that Prim's algorithm is a widely-used and efficient approach that uses a greedy strategy to add edges to the maze until a complete structure is formed. During the research, I have been exploring the theory of Prim's algorithm, the process behind maze generation, and writing Python code to generate randomized mazes.

In the conclusion part, I conclude both advantages and disadvantages of Prim’s algorithm, the complexity is also taken into account. Besides, we also collect currently popular maze generation algorithms and compare them by time complexity, space complexity, and feasibility. Aside from the Prim’s algorithm, Randomized Kruskal’s Algorithm and Wilson’s Algorithm are also famous algorithms in the application of randomized maze generation, which can also easily generate mazes with relativity low complexity.

# Analysis

## Application of  Prim's algorithm to generate minimum spanning trees

At first, we found that Prim's algorithm for generating minimum spanning trees(MST) is a popular topic. The algorithm provides an efficient way to construct a tree that connects all the vertices in a graph with minimum total weight. In order to better understand the terminology and theories, I've collected some basic definitions of the algorithm itself and MST.

### Definition:

Prim's algorithm:
"It is a well-known algorithm in computer science used to generate minimum spanning trees(MST) of a graph. It starts with a single node and iteratively adds edges to the graph, connecting it to the closest available nodes until all nodes are connected."

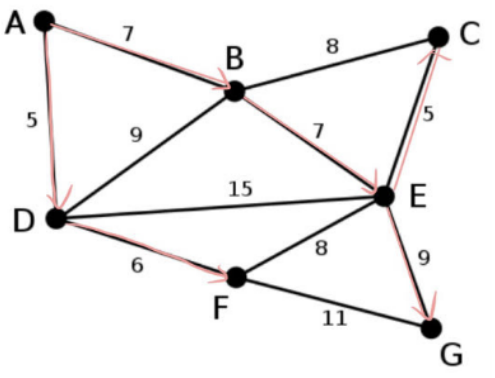
MST:
"MST stands for Minimum Spanning Tree, which is a tree that connects all the vertices in an undirected graph with the minimum total weight. In other words, it is a subset of the edges in the graph that forms a tree and has the minimum possible sum of edge weights. "

After understanding the theory, I also took time to learn how to use this algorithm to get a minimum spanning tree from a connected weighted graph. I concluded the possible steps:

### Steps:

- The graph is connected and weighted. All vertices in set V, all edges in set E.
- initiation: choose a start vertice, suppose to be A. Vnew = {A}, Enew = {}
- repeat the following steps until V = Vnew:
- choose the edge(u, v) which has the lowest weight, where u is already in Vnew and v is not in Vnew.
- put v into Vnew and put(u, v) into Enew, and update the sum cost as well.
- We can get the minimum spanning tree using Vnew and Enew.

Here is an example of the calculation:




The final set Vnew and Enew will look like this:

*Vnew = {A, D, F, B, E, C, G}*

*Enew = {AD, DF, AB, BE, EC, EG}*

*Minimum cost = 5 + 6 + 7 + 7 + 5 + 9 = 39*

By using the prim's algorithm, we will finally get a MST with a total cost of 39.

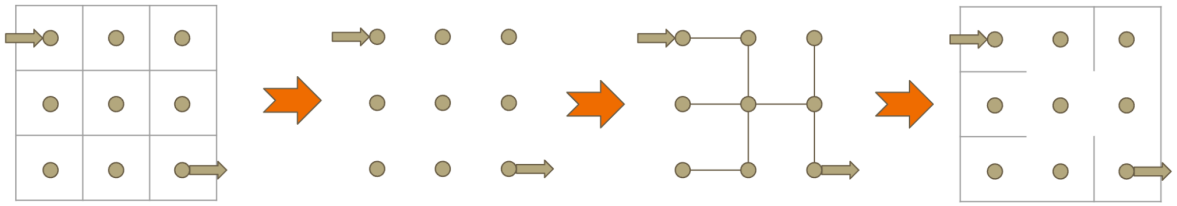
## abstraction of maze

After learning Prim's algorithm, we wonder how to use this algorithm to generate randomized 2D mazes. Therefore, we have done the abstraction process to show the connection between the MST and maze generation. Firstly, let's give the definition of a 2D perfect maze:

- A 'perfect' maze has no loops or closed circuits and is without any inaccessible areas.
- The overall outline of the maze is a two-dimensional rectangle.
- Cells inside the maze are squares, each adjacent to another cell on the top, bottom, left, or right side (except for edges and corners).
- There is only one path from the starting point to the endpoint.

Then we find that there are common features between MST and perfect 2D mazes, they are both connected graphs with no loops or circuits and without any inaccessible areas. Aside from that, there also remain some differences. For example, MSTs are weighted and the edges of mazes have no weight, so the implementation process of the generation of the randomized tree will be different from the original definition.

We have made pictures to show the abstraction process of a randomized maze.



1 2 3 4

1. Each grid can be abstracted to a vertex on graph
2. The paths between connected grids can be abstracted to edges
3. One possible tree by randomly adding edges
4. Convert the tree into a maze. If an edge exists between two vertices, break the “wall”

Having gained a clear understanding of the theory, abstraction process, and possible implementation ideas of using Prim's algorithm to generate mazes, I have made the decision to proceed with writing the Python code. My intention is to apply the algorithm to generate a maze and visualize the process of its creation. By doing so, I aim to gain practical experience with the implementation of the algorithm and deepen my understanding of its workings.

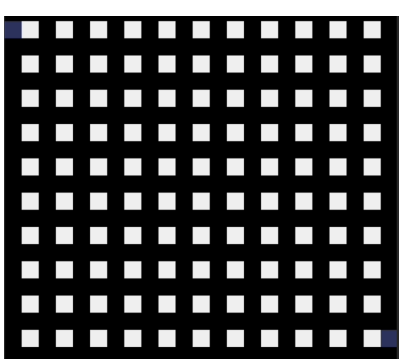
## implementation of maze generation by Python

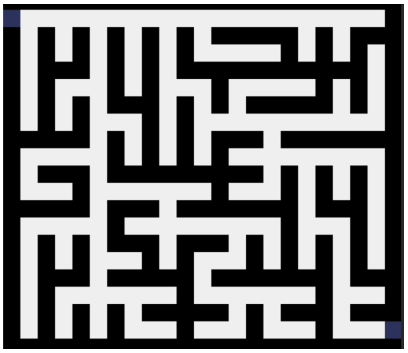
There are slight differences in the implementation of maze generation using Prim's algorithm as compared to its original definition. The most significant deviation lies in the process of adding edges. In the original definition, the algorithm selects one vertex from the set and another not in the set, ensuring that the edge between these two vertices has the smallest weight to obtain the minimum spanning tree (MST). However, in maze generation, the 'original graph' has no weights assigned to its edges, rendering the comparison process unnecessary. Instead, the system randomly selects a neighbor edge to generate the maze. Having considered the disparity, I have made modifications to the algorithm.

Furthermore, I have also given great consideration to the initialization of the original graph in order to make the generated mazes more realistic. To achieve this, I created a graph with all vertices and no edges. During program execution, the system will create a list of all the adjacent walls and randomly select one to break and form an edge. The wall is the block between two grids, when the wall is

broken, two vertices are connected and one edge is made. To avoid the addition of duplicate edges, a visit list has been implemented to keep track of visited walls.

I use the pixel package to visualize my program. At first, every white vertex exists with no edge. Black walls are also set. The Python code has been added in the Appendix.
The final project will look like this. The left picture is the original graph and the right one is a possible randomly generated maze using the Python program.





The visualization of the randomized Prim's algorithm has been achieved by visually representing the steps of breaking the "walls". As a result, every vertex in the graph is connected, leaving only one path between any two vertices. This guarantees that regardless of the starting and ending points, there will always be only one path connecting them, which perfectly matches the logic of the maze game.

## The complexity of Prim’s algorithm

The time complexity of maze generation using Prim's algorithm depends on the implementation of the data structure. Generally, the algorithm uses a priority queue to trace the minimum-weight edges to add to the minimum spanning tree.

If we use an adjacency list to represent the graph, the time complexity for adding an edge to the priority queue is O(log V), where V is the number of vertices in the graph. Since we need to add E edges to the priority queue, the total time complexity of the algorithm is O(E log V).

Otherwise, if we use an adjacency matrix to represent the graph, the time complexity for adding an edge to the priority queue is O(V), meaning the total time complexity of the algorithm becomes O(V^2).

In terms of space complexity, both adjacency list and adjacency matrix implementations require O(V^2) space to store the graph, as well as additional space for the priority queue and the visited list.

# Conclusion

We have done research on the popularly used maze generation algorithms to compare their advantages and conclude them in the following table:

| Algorithm | Advantages | Disadvantages | Time Complexity |
|---|---|---|---|
| **Randomized Kruskal's** (minimum spanning tree) | is better in typical situations and easier to implement as it uses disjoint sets and simpler data structures | biased towards many short dead ends | O(n log n) |
| **Prim's** (minimum spanning tree) | low space complexity<br>create mazes with different shapes and styles easily | biased towards many short dead ends | O(n log n) |
| **Wilson's** (uniform spanning tree) | high complexity<br>create mazes with unbiased, uniform trees | inefficient<br>high memory usage | $O(n^3)$ |
| **Randomized DFS** (graph traversal) | simple to implement<br>low memory usage | biased towards long corridors | O(n) |
| **Aldous-Broder** (uniform spanning tree) | fast for small mazes | inefficient<br>biased towards long corridors<br>high memory usage | $O(n^3)$ |

During my research on the application of Prim's algorithm, I gained a deeper understanding of the theory and the implementation process. While the basic principle of the algorithm is easy to comprehend, its practical use in generating randomized mazes is impressive. The algorithm has proven to be powerful and effective in generating intricate and challenging mazes.

By visualizing the maze generation process, I was able to create an image that closely resembles a real-life maze. The result was ideal and diligent, as the maze generated by Prim's algorithm contains all the necessary components.

However, the complexity of maze generation using Prim's algorithm must be analyzed. While it is efficient for small to medium-sized mazes, generating larger mazes may become impractical due to space requirements. It is important to consider this limitation when implementing the algorithm in real-life applications.

This report primarily focuses on the analysis of the theory and implementation of Prim's algorithm for maze generation. However, we have not considered any modified versions of the algorithm or specific

implementation details. As previously noted, the efficiency of the algorithm may decrease when generating larger mazes, highlighting the importance of exploring optimization strategies.

In addition to gaining a deeper understanding of the theoretical concepts, this project also provided me with the opportunity to enhance my programming skills. Moreover, I learned the value of teamwork and thoroughly enjoyed collaborating with my teammates on this project. I look forward to future opportunities to work together on fun and practical research projects.

# Reference


1. Tarjan, Robert Endre (1983), "Chapter 6. Minimum spanning trees. 6.2. Three classical algorithms", *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 44, Society for Industrial and Applied Mathematics, pp. 72–77.
2. Pettie, Seth; Ramachandran, Vijaya (January 2002), "An optimal minimum spanning tree algorithm" (PDF), *Journal of the ACM*, **49** (1): 16–34, CiteSeerX 10.1.1.110.7670, doi:10.1145/505241.505243, MR 2148431, S2CID 5362916.
3. Cheriton, David; Tarjan, Robert Endre (1976), "Finding minimum spanning trees", *SIAM Journal on Computing*, **5** (4): 724–742, doi:10.1137/0205051, MR 0446458.

# Appendix

```python
import random
import pyxel

class Maze:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.map = [[0 if x % 2 == 1 and y % 2 == 1\
                     else 1 for x in range(width)] for y in range(height)]
        self.map[1][0] = 0  # entry
        self.map[height - 2][width - 1] = 0  # exit
        self.visited = []
        # right up left down
        self.dx = [1, 0, -1, 0]
        self.dy = [0, -1, 0, 1]
        self.start = [1, 1]
        self.visited.append(self.start)
        self.wall_list = []
        self.get_neighbor_wall(self.start, self.wall_list)

    # generate the maze once, just for testing
    def generate(self):
        start = [1, 1]
        self.visited.append(start)
        wall_list = []
        self.get_neighbor_wall(start, wall_list)
        while wall_list:
            wall_position = random.choice(wall_list)
            neighbor_road = self.get_neighbor_road(wall_position)
            wall_list.remove(wall_position)
            # one of two roads have been visited or both are visited
            self.deal_with_not_visited(neighbor_road[0], wall_position, wall_list)
            self.deal_with_not_visited(neighbor_road[1], wall_position, wall_list)

    # only generate the next frame
    def next_frame(self):
        if not self.wall_list:
            return False

        while self.wall_list:
            wall_position = random.choice(self.wall_list)
            neighbor_road = self.get_neighbor_road(wall_position)
            self.wall_list.remove(wall_position)
            # one of two roads have been visited or both are visited
            vis1 = self.deal_with_not_visited(neighbor_road[0], wall_position, self.wall_list)
            vis2 = self.deal_with_not_visited(neighbor_road[1], wall_position, self.wall_list)
            if vis1 + vis2 == 1:
                break

        return True

    def get_neighbor_wall(self, start, wall_list):
        x, y = start[0], start[1]
        for i in range(4):
            x_new = x + self.dx[i]
            y_new = y + self.dy[i]
            if 0 < x_new < self.height - 1 and 0 < y_new < self.width - 1 and self.map[x_new][y_new] == 1:
                wall_list.append((x_new, y_new))
```

```
    def get_neighbor_road(self, wall_position):
        x, y = wall_position[0], wall_position[1]
        neighbor_road = []
        for i in range(4):
            x_new = x + self.dx[i]
            y_new = y + self.dy[i]
            if 0 < x_new < self.height - 1 and 0 < y_new < self.width - 1 and self.map[x_new][y_new] == 0:
                neighbor_road.append((x_new, y_new))
        return neighbor_road


    def print_map(self):
        for i in range(len(self.map)):
            for j in range(len(self.map[0])):
                print(self.map[i][j], end = ' ')
            print()


    def deal_with_not_visited(self, nr, wall_position, wall_list):
        if nr in self.visited:
            return 0
        self.visited.append(nr)
        x, y = wall_position[0], wall_position[1]
        self.map[x][y] = 0 #打通
        self.get_neighbor_wall(nr, wall_list)
        return 1
```

```
from maze_generation import Maze
import pyxel

class App:
    pixel = 6
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.maze = Maze(self.width, self.height)
        pyxel.init(width * App.pixel, height * App.pixel)
        pyxel.run(self.update, self.draw)

    def update(self):
        if pyxel.btn(pyxel.KEY_Q):
            pyxel.quit()
        if pyxel.btnp(pyxel.KEY_S) or pyxel.btnp(pyxel.KEY_D):
            self.maze.next_frame()
        if pyxel.btnp(pyxel.KEY_R):
            self.maze = Maze(self.width, self.height)

    def draw(self):
        # draw maze
        pyxel.cls(0)
        road_color = 7
        wall_color = 0
        start_point_color = 1
        end_point_color = 1
        for x in range(self.height):
            for y in range(self.width):
                color = road_color if self.maze.map[x][y] == 0 else wall_color
                pyxel.rect(y * App.pixel, x * App.pixel, App.pixel, App.pixel, color)
        pyxel.rect(0, App.pixel, App.pixel, App.pixel, start_point_color)
        pyxel.rect((self.width - 1) * App.pixel, (self.height - 2) * App.pixel, App.pixel, App.pixel, end_point_color)

App(23, 21)
```