Lab: Perfect Maze Generation Using Kruskal's Algorithm

Overview

In this lab, you will generate a *perfect maze*—a maze with exactly one simple path between any two cells—by constructing a **minimum spanning tree (MST)** of a 2D grid graph using **Kruskal's algorithm**.

Your goal is to:

- Model a rectangular grid as a graph
- Assign random weights to edges
- Use union–find to avoid cycles
- Use Kruskal's algorithm to build a spanning tree
- Render the resulting maze as ASCII art

This lab focuses strongly on **algorithmic thinking**, especially around graphs, disjoint-set data structures, and spanning trees.

What is a Perfect Maze?

A *perfect maze* satisfies two properties:

1. **Every cell is reachable from every other cell**
   → The maze is **connected**

2. **There is exactly one simple path between any two cells**
   → There are **no cycles**

In graph-theoretic terms, a perfect maze is exactly a **spanning tree** of the grid graph.

Using **Kruskal's MST algorithm**, you will generate a spanning tree where:

- Grid edges have random weights
- The MST edges become corridors
- All non-MST edges remain walls

The Grid as a Graph

We index grid cells using:

```
id = r * C + c
```

where:

- `r` is the row index
- `c` is the column index
- `C` is the number of columns

Each cell has up to two edges:

- Right neighbor: `(r, c+1)`
- Down neighbor: `(r+1, c)`

Each edge will be stored as:

```
struct Edge {
    int a, b;      // cell ids
    int weight;    // random priority
};
```

You will generate all valid edges in the grid and assign each one a random weight.

Union–Find (Disjoint Set Union)

You will implement the union–find structure used by Kruskal's algorithm.

Required operations:

Find(x)

Returns the representative of the set containing `x`.
You should implement **path compression**.

Union(a, b)

Merges the two sets containing `a` and `b`.
You should use **union by rank** (or size).

Union–find prevents cycles when selecting edges.

Kruskal's Algorithm

Kruskal's algorithm builds a minimum spanning tree using the following logic:

1. Sort all edges by weight
2. Iterate through them from smallest to largest
3. For each edge `(a, b)`:
    - If `Find(a) != Find(b)`:
        - Add the edge to the MST
        - `Union(a, b)`
4. Stop when you have selected `R*C - 1` edges

You must store newly selected edges in an `unordered_set<long long>` called `used`.
`encodeEdge(a, b)` converts an (a,b) edge into a hashable integer.

ASCII Maze Output

A provided function `printMaze(...)` draws the maze using only:

```
# = wall
. = open passage
```

Corridors are created by checking whether an edge is present in `used`.
Students **do not edit** this printing code.

Example output for a small maze:

```
###########
#.#...#.#.#
#.#.###.#.#
#.........#
#.#.###.###
#.#.#...#.#
#.###.#.#.#
#...#.#.#.#
#####.#.#.#
#.....#...#
###########
```

Example valid path (top-left to bottom-right):

```
Down, Right, Right, Right, Right,
Down, Down, Down,
Right, Right
```

This path uses only up/down/left/right moves and demonstrates that the maze is fully connected.

Your Tasks

You must complete the following TODO sections in `maze_student.cpp`:

1. `Find(int x)`
2. `Union(int a, int b)`
3. `buildEdges(int R, int C, vector<Edge>& edges)`
4. `runKruskal(int R, int C, vector<Edge>& edges, unordered_set<long long>& used)`

Do NOT modify:

- Any provided scaffolding
- The ASCII printing function
- The function signatures
- The random seeding or grid dimensions

Goal of the Lab

By the end of this lab, you will understand:

- How to model grids as graphs
- How to create and store edges
- How union–find detects cycles
- Why Kruskal's algorithm generates perfect mazes
- How spanning trees relate to maze structure

This lab reinforces key concepts in:

- Graph algorithms
- Data structures
- Randomized algorithms
- Software modularity and testing

What to Submit

Submit **only your modified `maze_student.cpp`**.

Your solution must:

- Compile successfully
- Produce a valid, connected perfect maze
- Pass all union-find, edge-construction, and Kruskal unit tests