1 Overview

In this lab, you'll implement **Boruvka's algorithm**, a parallel-friendly approach to finding minimum spanning trees. Unlike Kruskal's (which processes edges globally) and Prim's (which grows a single tree), Boruvka's grows multiple trees simultaneously in discrete phases.

2 Learning Objectives

- Understand how Boruvka's algorithm combines ideas from both Kruskal's and Prim's algorithms
- Implement a phase-based MST algorithm
- Analyze why the algorithm terminates in $O(\log V)$ phases
- Appreciate the algorithm's potential for parallelization

3 Background: How Boruvka's Algorithm Works

Boruvka's algorithm builds the MST in **phases**:

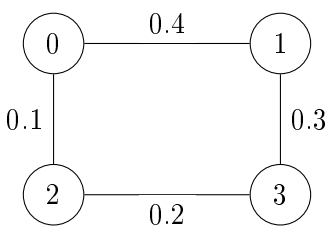
1. **Initially**: Each vertex is its own tree (component)
2. **Each phase**:
   - For each tree, find the minimum-weight edge connecting it to a *different* tree
   - Add all such edges to the MST simultaneously
   - These edges merge trees together
3. **Termination**: Stop when only one tree remains

3.1 Key Insight

The number of components decreases by at least a factor of 2 each phase, guaranteeing $O(\log V)$ phases maximum.
Why? Each component finds an edge to *another* component. In the worst case, components pair up, halving the total count.

3.2 Example Trace

Consider this graph:



Phase 1: Each vertex is a component

- Component {0}: cheapest edge to different component → (0,2) weight 0.1
- Component {1}: cheapest edge → (1,3) weight 0.3
- Component {2}: cheapest edge → (2,3) weight 0.2
- Component {3}: cheapest edge → (2,3) weight 0.2

Add edges: (0,2), (1,3), (2,3) → Components now: {0,2,3}, {1}
Phase 2:

- Component {0,2,3}: cheapest edge → (0,1) weight 0.4
- Component {1}: cheapest edge → (0,1) weight 0.4

Add edge: (0,1) → One component. **Done!**

4 Implementation Task

Implement the `BoruvkaMST` class following this structure:

```
#pragma once

#include <vector>
#include <algorithm>
#include <limits>
#include "weighted-graph.hpp"
#include "union-find.hpp"

class BoruvkaMST {
    std::vector<Edge> mst;
    double totalWeight;

public:
    BoruvkaMST(const EdgeWeightedGraph& G) : totalWeight(0.0) {
        // TODO: Implement Boruvka's algorithm
        // Hint structure:
        // 1. Initialize union-find for V vertices
        // 2. While number of components > 1:
        //    a. Find cheapest edge for each component
        //    b. Add all valid edges to MST
        //    c. Update union-find
    }

    const std::vector<Edge>& edges() const {
        return mst;
    }

    double weight() const {
        return totalWeight;
    }

    bool isValid(const EdgeWeightedGraph& G) const {
        return mst.size() == G.V() - 1;
    }
};
```

4.1 Implementation Hints

1. **Track cheapest edges per component**: Use a vertex-indexed array `std::vector<Edge*>` to store the minimum-weight edge for each component. Initialize with `nullptr` at the start of each phase.
2. **Finding cheapest edges**: For each edge (v, w):
   - Find components of v and w using union-find
   - If in different components, check if this edge is cheaper than the current best for each component
3. **Adding edges**: After examining all edges:
   - Iterate through the cheapest edge array
   - For each valid edge, check if it still connects different components (avoid duplicates)
   - Add to MST and unite components
4. **Termination**: Count components or check if any edges were added in the phase

4.2 Suggested Algorithm Structure

```
UnionFind uf(G.V());
int numComponents = G.V();

while (numComponents > 1) {
    // Array to store cheapest edge for each component
    std::vector<Edge*> cheapest(G.V(), nullptr);

    // Phase 1: Find cheapest edge for each component
    for (const Edge& e : G.edges()) {
        int v = e.either();
        int w = e.other(v);
        int compV = uf.find(v);
        int compW = uf.find(w);

        if (compV != compW) {
            // Update cheapest edge for component of v
            // Update cheapest edge for component of w
        }
    }

    // Phase 2: Add all cheapest edges
    for (int v = 0; v < G.V(); v++) {
        if (cheapest[v] != nullptr) {
            // Check if edge still connects different components
            // Add to MST if valid
            // Unite components
        }
    }
}
```

5 Testing Your Implementation

Create a test file `test-boruvka.cpp`:

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include "weighted-graph.hpp"
#include "boruvka-mst.hpp"
#include "kruskal-mst.hpp"
#include "lazy-prim-mst.hpp"

int main() {
    std::ifstream file("tinyEWG.txt");
    EdgeWeightedGraph G(file);

    std::cout << "Graph:\n" << G.to_string() << "\n";

    BoruvkaMST boruvka(G);
    KruskalMST kruskal(G);
    LazyPrimMST prim(G);

    std::cout << std::fixed << std::setprecision(5);
    std::cout << "Boruvka MST weight: " << boruvka.weight() << "\n";
    std::cout << "Kruskal MST weight: " << kruskal.weight() << "\n";
    std::cout << "Prim MST weight:    " << prim.weight() << "\n";

    std::cout << "\nBoruvka MST edges:\n";
    for (const Edge& e : boruvka.edges()) {
        int v = e.either();
        std::cout << v << "-" << e.other(v) << " "
                  << e.weight() << "\n";
    }

    return 0;
}
```

5.1 Sample Test Graph (tinyEWG.txt)

```
8 16
4 5 0.35
4 7 0.37
5 7 0.28
0 7 0.16
1 5 0.32
0 4 0.38
2 3 0.17
1 7 0.19
0 2 0.26
1 2 0.36
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93
```

Expected MST weight: **1.81000**

6 Questions to Consider

1. **Phase Analysis**: Add instrumentation to count the number of phases. Does it match the $O(\log V)$ bound?
2. **Edge Comparisons**: How many times is each edge examined? Compare to Kruskal's algorithm.
3. **Parallel Potential**: Which parts of Boruvka's algorithm could be parallelized? How does this compare to Kruskal's and Prim's?
4. **Duplicate Handling**: Why do we need to check `compV != compW` again when adding edges in phase 2?
5. **Space Complexity**: What is the space complexity of your implementation? Can it be improved?

7 Extensions (Optional)

1. **Phase Logging**: Modify your implementation to print which edges are added in each phase.
2. **Component Tracking**: Keep track of how many components exist after each phase and verify the halving property.
3. **Performance Comparison**: Time all three MST algorithms on large graphs. When might Boruvka's be preferred?
4. **Handle Equal Weights**: The current specification assumes distinct edge weights. How would you modify the algorithm to handle equal weights without creating cycles?

8 Submission

Submit your completed `boruvka-mst.hpp` file along with answers to questions 1–3.