CSCI 311 S25 City Connections Project Team 3

Nikita Bityutskiy, Duy Le, Nhi Cao

April 2025

1 Prim's Algorithm

Runtime Analysis

Our implementation of Prim's algorithm uses a priority queue to efficiently select the next edge with the minimum weight. Let n be the number of nodes and m be the number of edges in the input graph. The algorithm begins by making an adjacency list, which takes $O(m)$ time, since each edge is processed exactly once.

The main loop of the algorithm executes as long as there are edges in the heap and the number of edges in the MST is less than $n-1$. In each iteration, the algorithm pops the minimum edge from the heap, which takes $O(\log m)$ time. For each accepted edge, it may push up to $O(\deg(v))$ new edges into the heap. Across all iterations, the total number of heap insertions is $O(m)$, and each insertion takes $O(\log m)$ time.

In total, the runtime complexity of the algorithm is $O(m \log m)$, which is efficient for sparse and moderately dense graphs. The algorithm also records a list of steps for visualization purposes, but these operations involve only constant-time bookkeeping per edge and thus do not impact the overall asymptotic runtime.

2 Kruscal's Algorithm

Runtime Analysis

The runtime of Kruskal's Algorithm in our implementation depends on three major components: sorting the edges, performing Union-Find operations, and iterating over the edges. Suppose the number of nodes is n and the number of edges is m.

First, the edges are sorted by their weights using Python's built-in sorting, which has a time complexity of $O(m \log m)$. This sorting step dominates the overall runtime if the number of edges is larger than the number of nodes.

Second, the Union-Find data structure supports efficient operations due to the use of path compression and union by rank. Each operation of find() and union() has an amortized complexity of $O(\alpha(n))$. This function grows extremely slowly, making the Union-Find operations practically constant.

Third, the algorithm processes each edge in a single pass through the sorted list. For each edge, it performs one find() on each endpoint and a union() operation if they belong to different components.

Therefore, the total time complexity is $O(m \log m + m \cdot \alpha(n))$. Since $\alpha(n)$ is nearly constant, the overall complexity simplifies to $O(m \log m)$, making our implementation of Kruskal's Algorithm efficient even for large graphs.

3 Comparison of Prim's and Kruskal's Algorithms

Algorithmic Approach

Prim's and Kruskal's algorithms are both greedy methods for finding a MST, but they differ fundamentally:

- **Prim's Algorithm** grows the MST from a starting node by repeatedly adding the smallest-weight edge that connects a visited node to an unvisited node. It relies on a priority queue to efficiently select the next edge to add.
- **Kruskal's Algorithm** begins with all nodes as separate components and considers edges in order of increasing weight. It uses a Union-Find data structure to determine whether adding an edge would create a cycle, in order to avoid such.

When to Use Which

- **Prim's Algorithm** is typically more efficient on dense graphs because it maintains adjacent edges and doesn't need to consider all edges globally.
- **Kruskal's Algorithm** is better suited for pretty sparse graphs where edge lists are already sorted/nearly sorted. It is simpler to implement when edges are provided explicitly and the graph is not stored as an adjacency list.