Kruskal's Minimum Spanning Tree

COP4533 Final Project

Lydia Chung and Lauren Nunag

Presentation video link: https://youtu.be/lUIBXPtstyw
Github repository link: https://github.com/frooia/kruskals-project

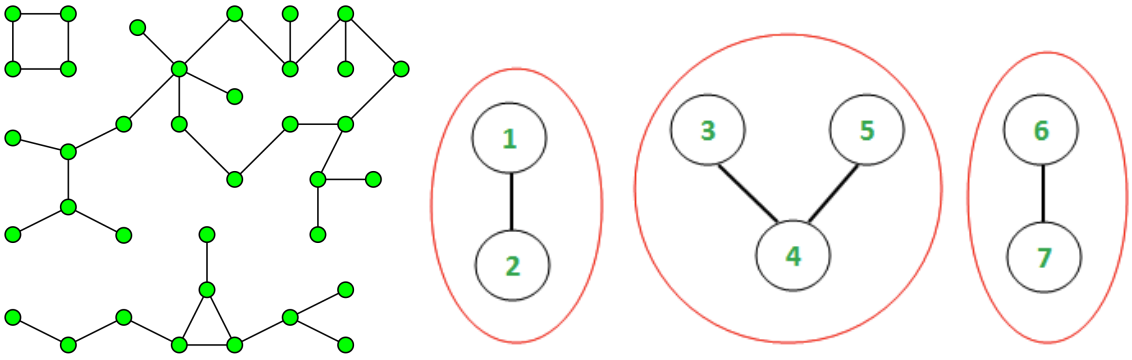
Algorithm Overview

Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph. An MST is defined as a subset of the graph's edges with the minimum possible total edge weight that connects all vertices without forming any cycles.

We implemented Kruskal's algorithm in C++ using a Disjoint Sets and Union (DSU) data structure to detect and avoid cycles. The algorithm processes edge-weighted graph data read from a file, builds the MST by selecting the lowest-cost edges, and outputs the sum of all edges used to construct the MST.

Disjoint Sets and Union

The DSU structure is a way to keep track of graph components, defined as one or more nodes connected together by some edges, which are separate from other components. A graph can have multiple components. For example, the graphs below have 3 components each.



A *connected graph* is defined as a graph that is composed of one single component. The input to Kruskal's algorithm is a connected graph, since the goal of an MST is to find a way to connect all vertices, so the premise must be that such a collection of edges exists. The graphs above are not valid inputs to Kruskal's algorithm, since they are not connected graphs.

DSU keeps track of graph components by two methods: *find* and *union*. The find method is called on a single vertex and it searches for its *parent*. A *parent* is defined as the node with the highest rank in a component. The *rank* of a node is used to keep track of the number of

components out of all vertices. The number of nodes with the highest rank is equal to the number of components in the graph.

The goal of Kruskal's algorithm is to use *union* to connect all vertices into one single component. In this final scenario, there will be a single node that is the parent of all other nodes in its component. This parent will have the highest rank out of all existing nodes. Therefore, every time we call *union*, we have to keep track of the current edge weight and number of nodes in the component we have connected so far.

Implementation Details

Our implementation of Kruskal's algorithm begins by sorting all edges in increasing order of their weights. This is a key component of this greedy approach, since we only want to consider the lowest weighted edges we need to connect all vertices. Edges with the highest weights will not be considered if they are not necessary for connecting new vertices. We create a new DSU structure with V vertices. Initially, all vertices are disconnected, with their ranks = 1 and parents = themselves.

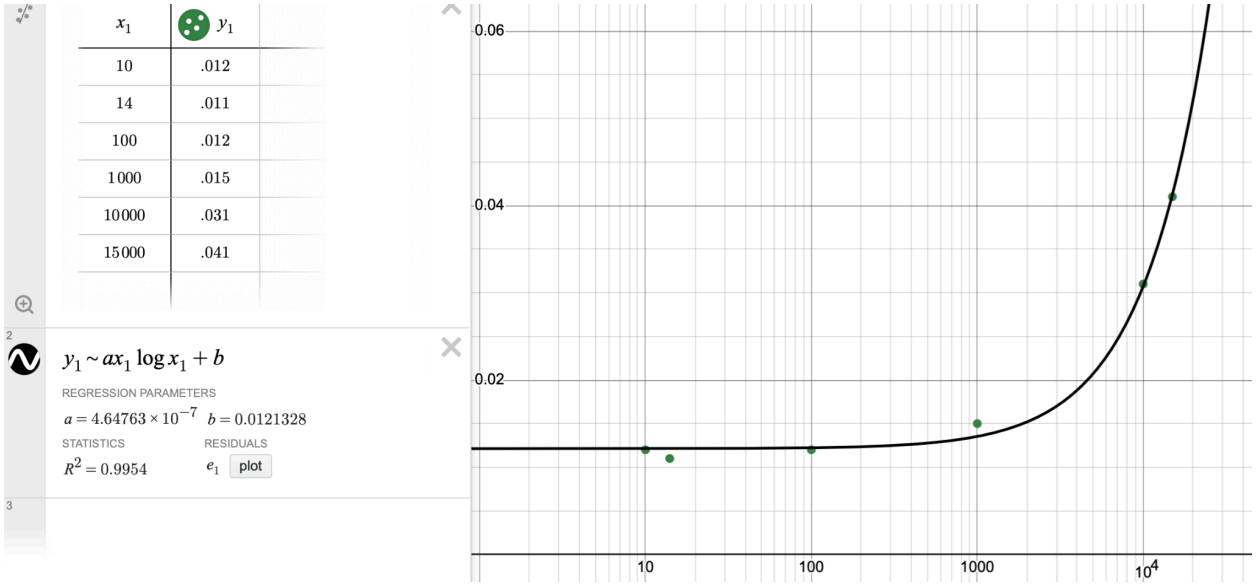
We iterate through all edges starting from the edge with lowest weight. First, we find the parents of the source and destination nodes of that edge using the *find* DSU method. If the nodes have the same parent, that means they are already in the same component and they've already been connected by a previous, cheaper edge. If the parents are different, that means the nodes are part of separate components that we need to *union* together. We call *union* on both nodes, which combines the components under the parent with higher rank. Finally, since we are including the edge, we update the total cost by its edge weight and increment the total number of edges we have unioned. When this count reaches V - 1, we have combined all V nodes into a single component, and we return the final cost.

Experimental Analysis

We wrote a test case generator script and used the *time* Unix utility to measure the runtime and memory usage of our Kruskal's algorithm implementation on different sized inputs.

Runtime Analysis

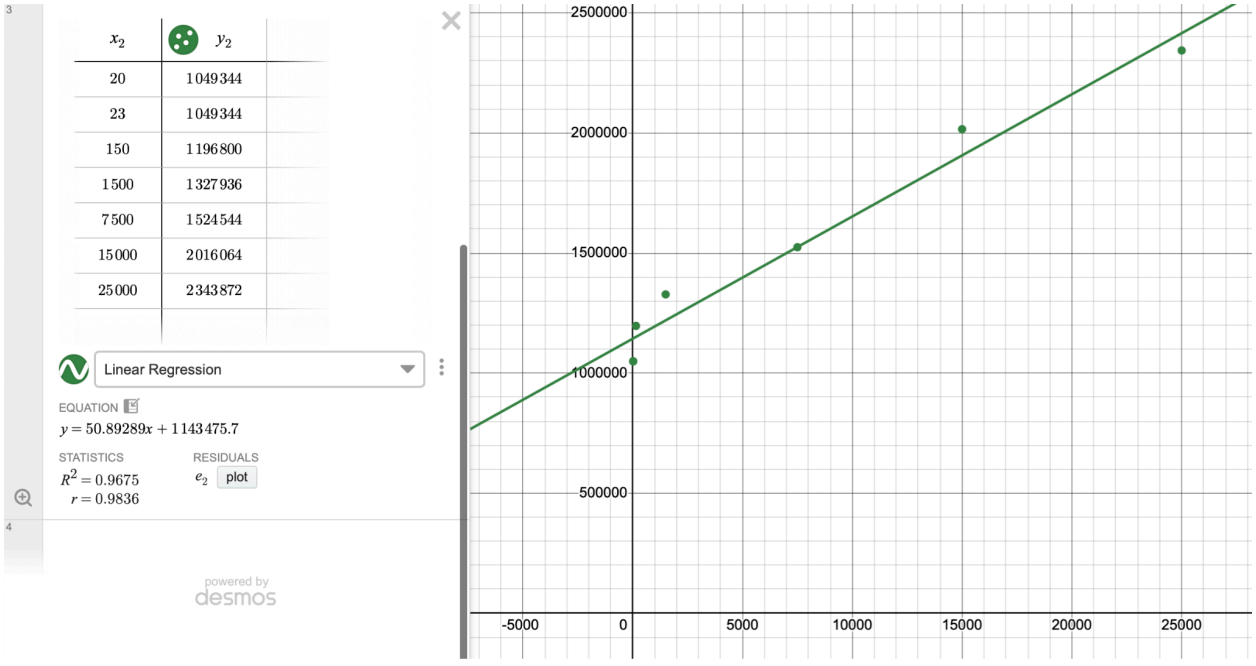
The theoretical time complexity is in the order of O(E log E). The runtime is dominated by the sorting of E edges, which runs in O(E log E). In the loop over the edges, the *find* and *union* DSU functions are called a constant number of times. Each of these functions run according to a function called the *inverse Ackermann function*, which achieves near constant time. Hence, this loop runs in O(E * a(V)), since the DSU structure is operating on V nodes and a() is the inverse Ackermann function. This is much lower than the sorting runtime of O(E log E).



The x axis shows the number of input edges in logarithmic scale, ranging from 10 to 15,000 for increasingly large graphs. The y axis measures runtime in seconds as measured by the time utility on Unix for each corresponding input size. The green dots show our actual measured values (for instance, y1 = 0.012 seconds for x1 = 10). The curve is the fit to the function x*log(x), modeling the theoretical O(E log E) time complexity. The fit is quite close with an R^2 value of 0.9954, showing a strong correlation.

Memory Usage Analysis

The theoretical space complexity is in the order of O(V + E). The DSU structure contains two arrays of size V to store the parent and rank of each vertex. Processing the input requires reading and iterating over an array of size E. No other extra storage is required for computation.



The x-axis shows the value of V + E as retrieved from our experimental inputs, and the y-axis shows the peak memory usage in bytes as measured by the time utility on Unix. V is the number of vertices and E is the number of edges in the input graph. This plot demonstrates a strong linear correlation between input size and memory usage, shown by the regression line with a high R^2 coefficient of 0.9675.

Key Takeaways

Kruskal's algorithm is generally considered quite efficient, since it runs in less than quadratic time. The runtime completely depends on the number of edges and not at all on the number of vertices, although we must have E be at least equal to V - 1 to have a connected graph. The primary runtime bottleneck is the initial sorting of edges based on weights. The space complexity is linear, which is generally considered the best possible, since inputs are usually read using linear space anyway. Future improvements to this project could compare Kruskal's MST implementation to Prim's MST in terms of runtime efficiency.