CSCI311-F22: City Connection Project Team 3

Nishant Shrestha, Anushka Desai, Malcolm Sturgis, Mikey Ferguson
ns037@bucknell.edu, aad018@bucknell.edu, mms044@bucknell.edu, mtf009@bucknell.edu

November 2022

1 Underlying Data Structure

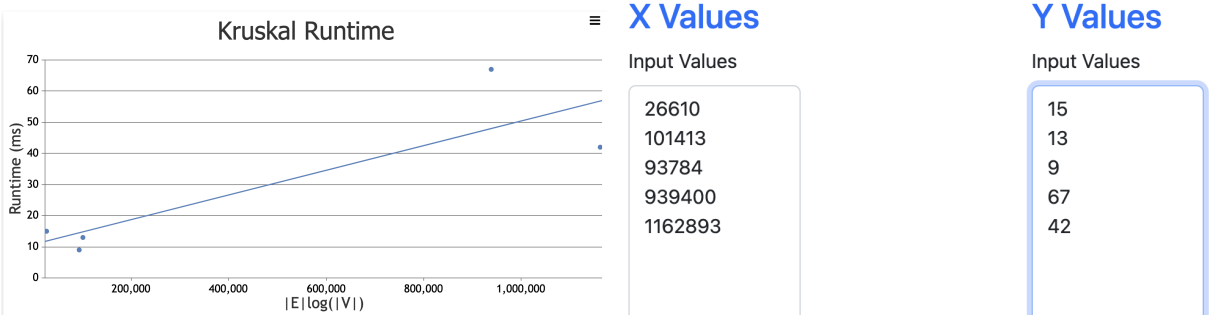
While we do have a matrix graph as an implementation option, which is a hash-map of hash-maps to avoid consecutive memory but still mimic the instant accessing of a matrix, the larger files, with hundreds of thousands of nodes, require over a billion doubles for all the potential edges and their lengths, even though most will be zero. We cannot store this much, and as such our two implementations which are of interest are our ConnectivityListGraph and DictionaryGraph. These only store edges which exist, accounting for less than one million edges per graph.
The ConnectivityListGraph is an array of Node objects, each of which stores a hash set (for instant searching and linear iteration) of connections (each of which communicates which node is connected and by what distance).
The DictionaryGraph is exactly like the above matrix graph, except instead of each inner hash-map containing all other nodes (most of which would have a corresponding null edge length of 0), only the connected nodes are included as keys in the inner hash-map, with values corresponding to the edge lengths between each inner node ID key and the outer node ID key.

2 Expected Versus Resulting Runtime For ConnectivityListGraph

Kruskal's Algorithm works by taking sorting all edges ($\Theta(|E|\log|E|)$), and then continually picking the smallest edge available until $|V|-1$ edges have been picked. With each new edge (and there are $|V|-1$ in total), the creation of cycles is avoided by utilizing disjoint sets on the nodes ($\Theta(\log^*|E|)$). This accounts for a total runtime of:

$$\begin{aligned} T(|E|,|V|) &= \Theta(|E|\log|E| + |E|\log^*|V|) \\ &= \Theta(|E|\log(|V|^2) + |E|\log^*|V|) \\ &= \Theta(|E|(2\log|V| + \log^*|V|)) \\ &= \Theta(|E|\log|V|) \end{aligned}$$

The observed runtimes for Kruskal's Algorithm are as follows:



Our runtime roughly adheres to the expected linear relationship between the quantity $|E|\log|V|$ and runtime. Our pseudo-code is as follows:
Kruskal-MST$(G=(E,V))$:

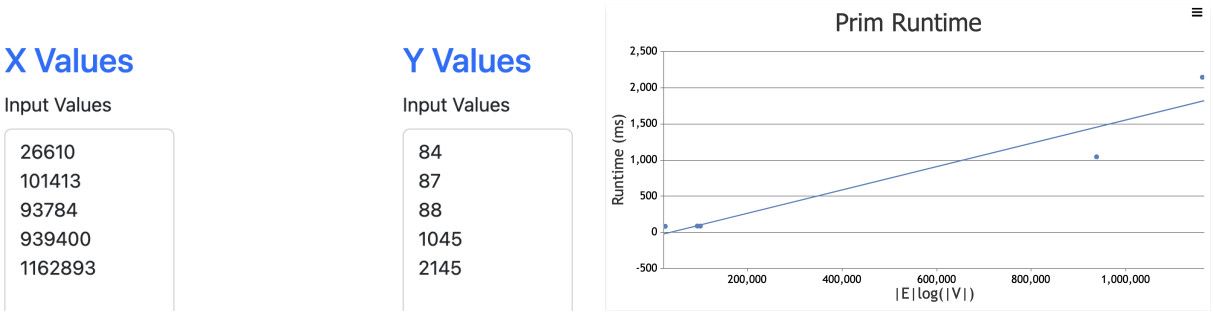
```
1: sort E
2: numEdges← 0
3:
4: for e in E do
5:     if numEdges= |V| − 1 then
6:         break
```

```
    end if

    id1← e.getFirstNodeID
    id2← e.getSecondNodeID

    if not sameSet(id1,id2) then
        makeSet(id1)
        makeSet(id2)
        // If these a set already exists, makeSet does nothing
        union(id1,id2)
        Add e to the minimum spanning tree
        numEdges=numEdges+1
    end if
end for
```

Prim's Algorithm works by starting at an arbitrary node, looking at all edges this node is a part of, picking the shortest one which does not induce a cycle, and hence yielding another node. From that node, you have some new potential edges to consider adding, and this process is repeated until $|V| - 1$ edges have been added to create the minimum spanning tree. In our implementation, we start at Node 0, add all of its corresponding edges to a tree set (heap) of edges, and pick the shortest one ($\Theta \log(|E|)$). That yields a new node, and we continually add those edges to the tree set, and then remove those edges which create a cycle. Ultimately, every edge is added to and removed from the tree set, accounting for a runtime of $\Theta(|E| \log(|E|)) = \Theta(|E| \log(|V|))$. Our observed runtimes for Prim's Algorithm are as follows:



Once again, our runtime roughly adheres to the expected linear relationship between the quantity $|E| \log |V|$ and runtime. Our Prim's Algorithm implementation likely takes longer than our Kruskal's Algorithm implementation because every edge must be added *and* removed from a tree-set. Our pseudo-code follows:

Prim-MST$(G = (E, V))$:

```
visited← new hash-set
potentialNextEdges← new tree-set
// potentialNextEdges sorts edges based on size

visited.add(0)
for edge e connected to Node 0 do
    Add e to potentialNextEdges
end for

while not all nodes are in visited do
    Iterate through potentialNextEdges until the an edge which does not induce a cycle is found
    // Is the second ID of the edge in the visited hash-set? That's how you know in constant time if an edge creates a cycle.
    id2← e.getSecondNodeID
    Add e to the minimum spanning tree
    visited.add(id2)
    Iterate through potentialNextEdges and remove any edges which would induce a cycle
    // We know that the first node ID's of each edge in potentialNextEdges are in visited. If the second node ID of an edge is in
end while
```