CIS 303 Analysis of Algorithms
Current Semester Here

Trevor Butler
Assignment 7b

12/15/2020

Problem

Create a java implementation or both Prim and Kruskal's algorithm to create a minimum spanning tree. Create a method labeled `getMST` in both files to create said trees with the parameter `Graph g)`. These methods will be executed in `MST.java` which will result in a displayed total weight in the terminal as well as a corresponding `.pmst` or `.kmst` file for Prim's algorithm and Kruskal's algorithm respectively.

Hypothesis

Based on the Kruskal and Prim's algorithms I believe that Kruskal's algorithm will be faster since adjacent vertices do not have to be selected like they do in Prim's. This is because Kruskal's algorithm sorts by edges with the least weight instead of using an adjacency list from a selected set of vertices. This leads me to believe when sorting and creating the MST Kruskal's will have a slight edge compared to Prim's.

Methods

To test the hypothesis, times were recorded of each execution of Prim/Kruskal's `getMST` function for each text file of 100 or greater. This allowed for easy comparisons of the timings between each algorithm.

1. Your implementation (code).

   Kruskal's `getMST`:

```
public static Graph getMST(Graph g) throws IOException {
    Set<Set<Vertex>> hashSet = new HashSet<Set<Vertex>>();
    for (int i = 0; i < g.vertices.size(); i++) {
        Set<Vertex> kSet = new HashSet<Vertex>();
        kSet.add(g.vertices.get(i));
        hashSet.add(kSet);
    }
```

```
    MinHeap<Edge> heap = new MinHeap<Edge>(g.edges.size());
    for (int i = 0; i < g.edges.size(); i++) {
        heap.insert(g.edges.get(i));
    }
    ArrayList<Edge> edgesList = new ArrayList<Edge>();
    while (hashSet.size() > 1) {
        Edge edge = heap.removeMin();
        Set<Vertex> head = findSet(hashSet, edge.source);
        Set<Vertex> second = findSet(hashSet, edge.dest);
        Set<Vertex> span = new HashSet<Vertex>(head);
        span.retainAll(second);
        if (span.isEmpty()) {
            Set<Vertex> append = new HashSet<Vertex>(head);
            append.addAll(second);
            hashSet.remove(head);
            hashSet.remove(second);
            hashSet.add(append);
            edgesList.add(edge);
        }
    }
    return new Graph(g.vertices, edgesList);
}
```

Prim's `getMST`:

```
public static Graph getMST(Graph G) throws IOException {
        G.getVertex(0).visited = true;
        Set<Vertex> vertices = new TreeSet<Vertex>();
        vertices.add(G.getVertex(0));
        Set<Edge> edgesTSet = new TreeSet<Edge>();

        while (vertices.size() < G.vertices.size()) {
                ArrayList<Edge> outsideEdge = new ArrayList<Edge>();
                for (Vertex vertex : vertices) {
                        ArrayList<Edge> array = G.edges(vertex);
                        for (int i = 0; i < array.size(); i++) {
                                if (!array.get(i).dest.visited)
                                outsideEdge.add(array.get(i));
                        }
                }
                Edge[] temp = new Edge[outsideEdge.size()];
                outsideEdge.toArray(temp);
                MinHeap<Edge> heap = new MinHeap(temp, temp.length,
                                                        temp.length);
                Edge edge = heap.removeMin();
```

```
                Vertex dest = edge.dest;
                dest.visited = true;
                vertices.add(dest);
                edgesTSet.add(edge);
        }
        ArrayList<Vertex> Vertex = new ArrayList<Vertex>();
        ArrayList<Edge> Edge = new ArrayList<Edge>();
        for (Vertex vertex : vertices){
                Vertex.add(vertex);
        }
        for (Edge e : edgesTSet){
                Edge.add(e);
        }
        Graph set = new Graph(Vertex, Edge);
        return set;
}
```

2. General approach.
   The approach required an understanding and implementation of both Kruskal and Prim's algorithm. This was done by following each algorithm and recognizing key components. First using pseudo code to lay out a basis of what needed to be done then turning that into java code to complete the task of creating a minimum spanning tree. After some time two `getMST` methods were completed for each algorithm which would read a .txt file and turn the data into a type of MST.

3. Data structures.
   There were a few data structures used in order to complete as solution to the problem. The following dtat structures were used: ArrayList, HashSet, Set, TreeSet

4. Experiment details.

   (a) Size of N you used.
   7 Text Files: 100.txt, 900.txt, 1000.txt, 1500.txt, 1900.txt, 2000.txt, 10000.txt

   (b) Case(s) tested (best, worst, random, etc.).
   Two cases being Prim and Kruskal implementations

   (c) Number of iterations/copies of the experiment.
   14 total iterations of the experiment being 7 tested .txt files for both Prim and Kruskal implementations

   (d) Metrics.
   Time (ms) and MST Trees

Results/Discussion

The data collected was quite astounding as it showed a clear difference of time complexities between both algorithms, a difference that was much unexpected. The data at first did not show

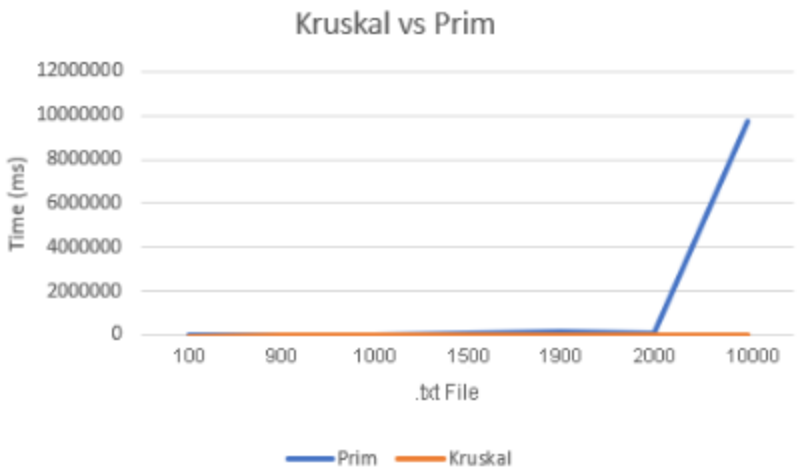

Figure 1: Graph of recorded numbers in time test between Prim and Kruskal's algorithm

much of a difference as both algorithms had a time variance of only one millisecond with the 100.txt, Prim holding that extra millisecond. But as more data was collected there was an obvious jump in time showing the efficiency of Kruskal's algorithm. Kruskal's results from 900-9000.txt files stayed between 260-804 milliseconds, which is stunning compared to the time results of Prim's algorithm. Prim's algorithm for these same text files had a time span of 3795-105920 milliseconds, which is appalling compared to Kruskal.

The biggest difference came when executing the 10000.txt file for both these algorithms. Kruskal blew Prim out of the water with 22136 milliseconds versus the 9760198 milliseconds Prim took to create a MST. That is the equivalent of about 22 seconds by Kruskal and a whopping 2.7 hours for Prim. This can be seen in Figure 1 where there is a blatant jump beginning at the 2000.txt to the 10000.txt illustrating the absurd efficiency of Kruskal's algorithm

Conclusion

The hypothesis stated was correct as Prim's algorithm took more time to complete than it did using Kruskal's algorithm. While running the 100.txt the numbers were very similar only having a difference of one millisecond but every text file after this (900.txt, 1000.txt, etc.) showed a large difference in time to complete the MST (Prim's 105920ms vs Kruskal's 803ms for 1900.txt). Assuming the reason stated in my hypothesis is correct, this is due to the fact Kruskal's algorithm does not use an adjacency list for a selected set of vertices. Rather it sorts by the weight of the edges first which results in faster times.