CSCE 221
Assignment Coverpage

Sources

Please list all sources consulted regarding this assignment. This includes, but is not limited to: Other CSCE 221 Students, other people, printed material, web material, etc.

1.
2.
3.
4.
5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalty to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct - it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

Assignment Description

Assignment:	
Culture Assignment Info (Speaker/Seminar/Date, or Data Structure):	

Honor Statement

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Date

Printed Name

Signature

CSCE 221 — 200
Programming Assignment 4
Final Report

James Corder Guy
Nathan Powell

May 3, 2016

1 Introduction

For this project, we implemented a fully-functional graph data structure, along with breadth-first search and Kruskal's minimum spanning tree algrorithms.

2 Implementation Details

2.1 Graph

The graph was implemented using a map of vertices, a master map of edges, and an adjacency edge map for each vertex. The vertex map keys the vertices by their identifiers, and each element contains a pointer to its corresponding vertex. Each vertex is assigned a unique identifier number corresponding to the number at which it was inputted, and also a list of edges that connect to it. Both the master and the adjacency edge maps key the edges by the vertex identifier pairs that they connect, and each element contains a pointer to its corresponding edge. Each edge contains the descriptors of the two verticies it connects, and its weight. All edges are directed; in order to insert an undirected edge, two edges going in opposite directions between the same two vertices are inserted.

2.2 Input/Output

The graph pulls its initial vertices and edges from a file, reading in first the number of vertices, then the number of edges. It then reads in the next n lines, where n is the number of vertices, and initializes them as vertices in the vertex container. The graph then reads in the remaining lines as edges, with each line having three values, in the order of source vertex, destination vertex, and weight.

2.3 Breadth-First Search

We used a standard implementation of breadth-first search in our project. To begin, each edge and vertex in the graph is labelled as unexplored. Then, starting at the first unexplored vertex, a queue is formed, initally with the starting vertex as its only member. Then, the algorithm checks each unexplored edge connected to the starting vertex (since this is the first iteration of the algorithm, all adjacent edges are checked, since they're all unexplored). The unexplored vertices (again, all of the adjacent ones on the first iteration) are then labelled as visited, and the edges used to visit them are labelled as discovery edges. To end this iteration, each newly-visited vertex is added to the back of the queue.

On the following iterations, the same procedure is repeated on the vertex currently at the front of the queue. The vertex is removed from the front, and the edges and connected vertices are checked. If an edge is found to go to an already visited vertex, it is labelled as a cross edge, and if it returns to the immediatly previous vertex (i.e., the current vertex's parent), it is labelled as a back edge. Any newly-visited vertices are then added to the queue.

As each vertex is removed from the queue and processed, the algorithm updates a map, keeping track of the parent of each vertex. This allows the tree traversed by the algorithm to be constructed once the algorithm finishes running.

Once all the vertices that can be reached have been processed through the queue, the queue is empty, and the algorithm checks to see if there are still any unexplored vertices in the graph, and if there are, adds the next one to the queue to begin the next iterations. These unlabelled vertices are the result of the graph either having multiple connected components, or of having a vertex that has directed nodes going only away from it. Regardless, running the algorithm again starting on each remaining unexplored vertex ensures the entire graph is labelled and added to the tree map.

2.4 Kruskal's Algorithm

Kruskal's Algorithm was used to find the minimum spanning tree of the input graph by dividing the vertices into clusters, and then merging them back together based upon the smallest edge weights between them. These edges would make up the minimum spanning tree, and were copied into a parent map for analysis. First, The algorithm then read all the edges in the graph into a multimap, keying the edges by weight.

Next, the first and last keys in the multimap are checked; if they are the same, then every edge in the map must have the same weight; therefore, every spanning tree is a minimum spanning tree. If not, the algorithm continues and sets up containers known as clusters; these represent the vertices linked together by Krusr ekal's algorithm that will form the minimum spanning tree.

Another container that holds pointers to each cluster is initialized for constant access to each cluster. Every vertex is then inserted into its own cluster, and the algorithm begins to iterate through every edge in the multimap, comparing the vertices it connects; if they are not in the same cluster, that edge is added to the parent map, and the smaller cluster is merged into the larger cluster, though the now empty smaller cluster is not deleted; doing so would reindex the remaining clusters, invalidating all of the pointers to the clusters. This process repeats until there are n-1 edges in the parent map, with n being the number of vertices in the graph.

3 Theoretical Analysis

3.1 Graph

Inserting an edge or vertex takes $\mathcal{O}(1)$ time, because the elements are simply added to their respective maps. Erasing an edge also takes $\mathcal{O}(1)$ time, because each edge contains the descriptors to its two vertices that can immediately access the vetices' adjacency lists. Removing a vertex takes $\mathcal{O}(m + deg(v))$ time, because every edge connected to that vertex must be deleted, as well as deleting the edge from the master edge list.

3.2 Input/Output

The input and output both will run in $\mathcal{O}(n + m)$ time, because every vertex and every edge will have to be inserted into their respective containers, and if each individual process takes $\mathcal{O}(1)$ time, if there are n vertices and m edges then the total time will be $\mathcal{O}(n + m)$.

3.3 Breadth-First Search

Breadth-First Search runs in $\mathcal{O}(n+m)$ time, because each vertex and edge is visited exactly twice: once to set it to UNEXPLORED, and once to set it a vertex to VISITED and an edge to either DISCOVERY or CROSS. If there are n vertices and m edges, then the total time would be $\mathcal{O}(2(n + m))$, or just $\mathcal{O}(n + m)$.

3.4 Kruskal's Algorithm

Kruskal's Algorithm should run in $\mathcal{O}((n + m) \log n)$ time, because each vertex will be merged at most $\log n$ times, resulting in $\mathcal{O}(n \log n)$ time, because smaller clusters are always merged into larger clusters, and there will be at most m removals from the edge multimap (because there are m edges in total), resulting in $\mathcal{O}(m \log n)$ time, because each union operation on the edge's vertices takes $\mathcal{O}(\log n)$ time. Therefore, the total running time of Kruskal's Algorithm is $\mathcal{O}((n + m) \log n)$ time.

4 Experimental Analysis

4.1 Testing Hardware

Specifications of Corder's system, used for testing:

CPU:	Intel i5-4210U @ 2.7GHz
RAM:	8GB
OS:	Arch Linux
Kernel:	4.5.1-1

4.2 Testing Procedure

Testing was performed with the provided timing file. There were three rounds of testing, and for each round, a different type of graph was considered: a fully-connected graph, a mesh or grid graph, and a random graph. Each round started with a small graph (2^2 or 2^4, in the case of the mesh graph), and doubled in size for each consecutive test, up until the tests took a prohibitively long time to run on our hardware. Each size graph was tested 20 times, and the results were averaged.

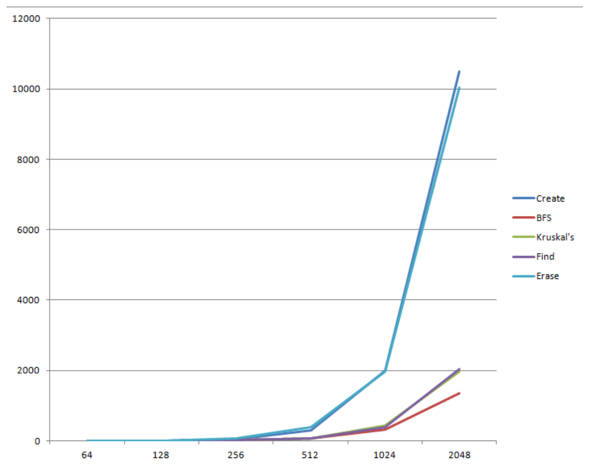
Max Input Size:

Complete:	2^{11}
Mesh:	2^{13}
Random:	2^{12}

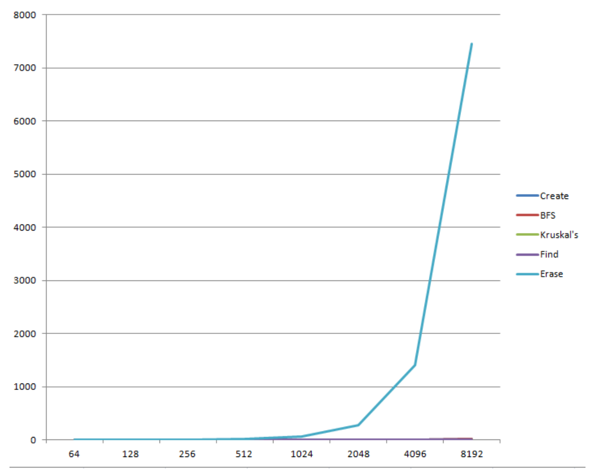
5 Results

In each figure, the x axis is $\log_2$ (input size), and the y axis is time in ms.

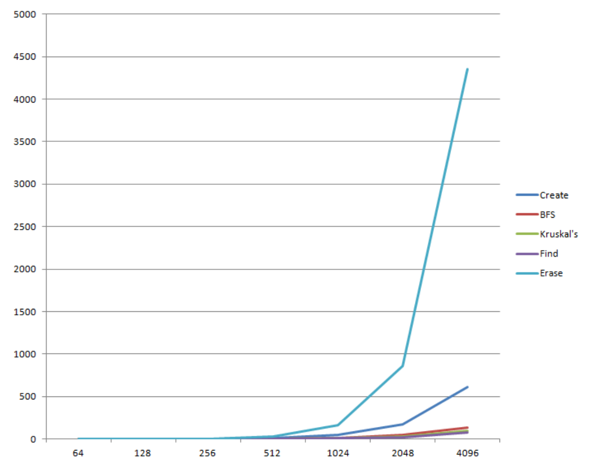
5.1 Complete Graph



5.2 Mesh Graph



5.3 Random Graph



6 Team Contributions

6.1 Graph

Nathan:	iterators, containers, inserters, erasers
Corder:	accessors, mutators

6.1.1 Vertex

Nathan:	adjacency list
Corder:	iterators, accessors, data members

6.1.2 Edge

Corder:	iterators, accessors, data members

6.1.3 I/O

Nathan:	input
Corder:	output

6.2 Algorithms

Nathan:	Kruskal's
Corder:	BFS

7 Conclusion

Looking at the figures, we see results consistent with our theoretical analysis. Our algorithms take much longer on the complete graph, because their time complexities are dependent on the number of edges, which is $\mathcal{O}(n^2)$ in a fully-connected graph. Also, BFS runs very quickly on a mesh graph, as each iteration expands to another column and row of the grid.