Christian Heinrich

Lab2

Part1

Kruskal’s Minimum Spanning Tree algorithm both with and without the Union find data structure. Kruskal’s works by taking a sorted set of edges and repeatedly adding the minimum weight edge which won’t create a cycle, until the creation of the minimum spanning tree. In my program I opted to store the various nodes of the graph in a priorityQueue with the weight determining the priority, as this avoided the need to actually sort the graph. The Union find data structure works by creating a series of sets of nodes, one for each vertex, and as edges are added, the sets for the beginning and ending of the edge are merged into a single set, by setting their parent reference equal to the same node at the root of the current set. My own method for performing Kruskals was similar to this, but far less efficient without the usage of the union find data structure. I performed Kruskals by creating a list object called vertex set, initially creating one for each node, and then merging them together by moving all the nodes from on to the other. This resulted in a similar operation but with a significantly lower run time than the union -find operation. In order to test my program I used the provided graph generation, and the actual graph operations are done simply by passing an adjacency matrix into either the KruskalsUnionFind or KruskalsNoUnionFind functions, which will then perform the operations and print out a list of the edges. I chose to use Java to write my program, which was fine for writing Kruskal’s Algorithm for part 1, but ended up being an issue for part 2. Union find operates in O(log n), due to its efficient design, however my method takes at least O(n) as it has to iterate over the VertexSets in order to check if a set contains a given value.

Part2

Part 2 was implementing Huffman encoding for a given file, and it is here where I ran into some unfortunate issues. Specifically the goal of the encoding was to compress the size of the file, however I was unable to do so, and instead my compression program actually increases the size of the file once encoded, by a factor of almost four times, due to the fact that the characters being written to the file still hold the same encoding as regular string characters, a problem which I attempted to solve multiple ways, but was ultimately unable to in large part due to my relative unfamiliarity with Java, as I know how to solve the problem using C++, but didn’t end up having enough time to completely rewrite the program. Furthermore, trying to fix this problem consumed enough time that I wasn’t able to complete the decoding from a given encoded file. Ultimately there is no real excuse for these issues, and will be solved in the future.

Part 1 data

Set 1: connectivity <10

Set 2: connectivity > N/2

Set 3: random graph

For 40, 60 and 100 node ararys

|  |  |  |  |
| --- | --- | --- | --- |
| UF | 40 | 60 | 100 |
| Set 1 | 1 | 3 | 3 |
| Set 2 | 2 | 2 | 6 |
| Set 3 | 3 | 3 | 5 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| No UF | 40 | 60 | 100 |
| Set 1 | 2 | 6 | 14 |
| Set 2 | 4 | 6 | 18 |
| Set 3 | 4 | 11 | 15 |