Analysis of Kruskal's Algorithm Runtime When Using Different Disjoint-Set Implementations

[redacted]

Implementations of Kruskal's algorithm require the use of a disjoint-set data structure, the efficiency of which heavily effects the efficiency of the algorithm implementation. In this paper, I analyze the differences between the trivial disjoint-set implementation and a more robust implementation of my own design, as well as how use of these disjoint-sets affects the runtime of my own Kruskal's algorithm implementation. The trivial implementation uses a vector of lists and results in quadratic runtime of Kruskal's algorithms. My own disjoint-set implementation utilizes two hash tables linking in and out of lists and results in linear runtime of Kruskal's algorithm. This difference can most likely be attributed to the trivial implementation finding the subset of an element in linear time, whereas my own implementation finds the subset of an element in constant time.

I. Introduction

Kruskal's algorithm generates the minimum spanning tree for a given graph. It does this by adding edges of successively larger weight from the graph until every vertex can be reached within the tree [1]. The algorithm starts by treating each vertex of the graph as its own tree in a forest, which is procedurally connected into a single tree, if possible. A disjoint-set is required to keep track of this forest and maintain a record of what vertices are or aren't connected. The efficiency of this disjoint-set data structure is integral to the efficiency of the algorithm as a whole, as the majority of the algorithm's runtime is spent interacting with it.

I have written a function which uses Kruskal's algorithm to find a graph's minimum spanning tree and created two implementations of a disjoint-set data structure. The algorithm uses each disjoint-set implementation in turn on the same set of graphs. The program running the algorithm records the runtime of the algorithm as it generates each minimum spanning tree.

II. Methods

My function implementing Kruskal's algorithm starts by initializing and empty vector of tuples, representing the minimum spanning tree. Depending on the Boolean passed to it, the function then initializes one of the disjoint-set implementations and populates it with a subset for every vertex of the graph. An instance of the C++ STL multimap is used as a priority queue, which the edges of the graph are sorted into.

The trivial disjoint-set implementation uses a C++ STL vector container to hold a sequence of subsets. Each subset is represented by a C++ STL list of elements, as required by the assignment handout. Presumably, the reason why a linked list is the trivial implementation is because two lists can be spliced together in constant time. The findSet function returns the subset ID of a given element by iterating through each subset. The setUnion function combines the subsets containing two given elements by finding their respective subset lists and splicing them together.

The optimized implementation also uses the C++ STL list to represent a subset, but these subsets are instead contained in a hash table which maps subset IDs to the subsets. Another hash table maps every element within the disjoint-set to subset ID it belongs to. The findSet function passes its argument to the latter hash table to find the element's subset ID. The setUnion function finds the subset IDs of the two passed elements using the latter hash table, then finds the subsets associated to the IDs using the former hash table. These subsets are then spliced together, and the subset IDs of the affected elements are updated in the former hash table.

The fifteen graphs used for testing were generated using the Watts-Strogatz model with rewiring probability p=1. The number of vertices double from 1 to 16384 with each successive graph. The graphs have a linear edge density of ten to simulate real-world graphs [2], except for those which have too few vertices to support ten times as many edges, in which case those graphs contain as many edges as possible. Testing was performed in trials. For each trial, the algorithm was timed while using the trivial implementation then the optimized implementation for all fifteen graphs.

III. Results

Across 21 trials, after throwing out outliers, averages shown in Figure 1 indicate using the trivial disjoint-set

implementation caused the algorithm to run in quadratic time. Figure 1 also shows using the optimized disjoint-set implementation allowed the algorithm to run in approximately linear time. For the purpose of keeping this paper within the two-page limit, the full data from the trials can be found in the project repository rather than the appendix.

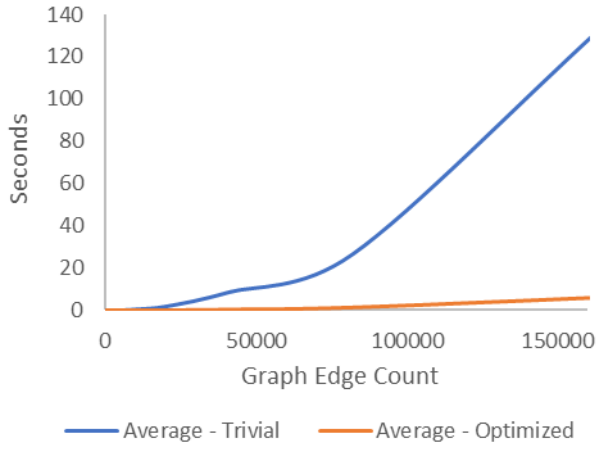

Fig. 1. Algorithm runtime as a function of the number of edges in graph

For graphs with 28 or fewer edges, there is no appreciable difference in runtime when using the two disjoint-set implementations, but difference in growth rates quickly become evident in graphs with more edges.

IV. Discussion

When performing a big-O analysis on my function for finding minimum spanning trees, time complexity should theoretically be quadratic when using either of my disjoint set implementations. Initializing data structures occurs in constant time, populating the disjoint-set with vertices occurs in linear time, and sorting edges into a priority queue occurs in log-linear time. The quadratic time complexity arises when iterating through the priority queue (linear complexity) and performing unions based on each edge encountered (linear complexity). The trivial implementation performs set unions in linear time because finding the subsets elements belong to requires linear time. The optimized implementation performs set unions in learn time because updating subset IDs for each affected element after combining sets requires linear time.

However, despite the loop body of the algorithm being linear in both cases, when the algorithm uses the optimized disjoint-set it is performing fewer linear time complexity operations. The find operations used to acquire subset IDs are linear when using the trivial implementation, but constant when using the robust implementation.

Another instance of less time used in practice comes when the algorithm calls the set union function. The trivial implementation must iterate through all elements stored in the disjoint-set twice to find the correct two subsets to perform a union on. The optimized implementation retrieves the appropriate subsets in constant time by referencing a hash table. After splicing the lists together, the disjoint-set must change the subset IDs of the elements to all be the same. On average, half of the elements will retain the same subset ID and the other half must be iterated through to change their subset ID. This set union operation requires much fewer iterations over elements of the disjoint-set.

V. Conclusion

While my optimized implementation of a disjoint-set performs set unions in linear time like the trivial implementation, it is keeping operations of linear complexity to a minimum, utilizing the constant access time nature of hash tables where possible to avoid iterating. Additionally, because the optimized disjoint-set implementation can find the subset ID of an element in constant time, Kruskal's algorithm can check much more quickly whether two vertices belong to the same subset. This allows the algorithm to iterate through edges of the graphs much more quickly when generating minimum spanning trees.

References

[1] J. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.* 7 (1956), 48–50.

[2] Guy Melançon. "Just how dense are dense graphs in the real world? A methodological note." *BE-LIV 2006: Beyond time and errors: novel evaluation methods for Information Visualization (AVIWorkshop)*, May 2006, Venice, Italy, pp.75-81. lirmm-00091354