CS350 HW 8

Nate Callon, Travis Block, Keaton Spiller

November 2021

Abstract

Like many problems, there are often multiple ways to come to a solution, and attempting the find a minimum spanning tree of a weighted graph is no different. Often the fastest solution to solving our problems is desired, so examining options of solving the problem is crucial towards finding an efficient solution. Two ways of finding the minimum spanning tree examined: Prim's Algorithm using an adjacency matrix and Kruskal's Algorithm. Given a weighted graph where the vertices represent cities, and their edges the distance between both algorithms will be used to identify the minimum spanning tree if one exists, and the time it took to find a solution. In this case the findings in this paper confirms that Kruskal's algorithm is more effective at finding the minimum spanning tree the shortest amount of time. Furthermore, Kruskal's is better for finding the minimum spanning tree for sparse graphs because it uses simple data structures. While Prim's algorithm is better for finding the minimum spanning tree of dense graphs with many more edges than vertices. Although we didn't get the correct answer for out final result of city-pairs.txt, we learned a substantial about the interworkings of how to use a dictionary, lists, and

Introduction

Two efficient methods for finding a minimum spanning tree of a weighted, undirected graph involve the use of Prim's algorithm and Kruskal's algorithm. Both of these achieve the same goal through different methods, and in this write-up will measure the efficiency difference between them, and why.

Prim's algorithm involves keeping track of a set of "included" vertices, and starts out by picking the vertex with the minimum key value in graph G and "including" it in this new set, then looking at all the edges and including the next least weighted vertex (in respect to the vertex we chose previously). As this "branches" out, it will compare and choose accordingly to the least weighted vertex in connection to the already chosen vertices, and continue to add to the set of included vertices/connections until all vertices are included. The end result will be a minimum spanning tree of the graph.

Kruskal's algorithm begins by sorting all the edges in non-decreasing order of their weight, and starts out by picking the least weighted edge. It will then

go through the list (increasingly in weight values) and check if adding an edge would make a cycle, and if not, adding that edge and taking it off the list. This will then repeat until there are $V - 1$ edges and the final result is the minimum spanning tree of the graph.

Background

Prim's Algorithm was initially developed in 1930 by Czech mathematician Vojtěch Jarník. Robert Clay Prim independently discovered it in 1957, and in 1959 it was rediscovered once more by Edsger Wybe Dijkstra. Kruskal's algorithm first appeared in Proceedings of the American Mathematical Society, pp. 48–50 in 1956, and was written by Joseph Kruskal. Both of these are valid methods for finding the minimum spanning tree of an unweighted directed graph.

The time complexity for Kruskal's algorithm is $O(ElogV)$, where V is the number of vertices and E the number of edges. Prim's algorithm has a time complexity of $O(V^2)$, where again, V is the number of vertices and can be improved up to $O(ElogV)$ using Fibonacci heaps. We implemented Prim's algorithm the original way without the use of Fibonacci heaps.

Data Analysis Process or Procedure

0.1 Procedure

The website used to compare the two algorithms in C++ and Python was Replit. This allowed for multiple user to simultaneously work on the algorithms implementations and output results to console for everyone to compare. Only one terminal could run at a time so we had to take turns to display the results for each algorithm. In order to quickly test the code we used visual studio Code to individually run the results and pasted our updates to the algorithms back into Replit to compare.

0.2 Data Sets Used

For the comparison between the two algorithms, Prim and Kruskal, the first set is a graph of cities in Oregon and the distances between each city. This provided list is not initially given in any particular order. Next, the second set of data is a random set of a random set of vertices and their weighted edges.

Link to C++ test page: Longest Common Subsequence JavaScript

Link to python test page: Longest Common Subsequence C++

0.3 Runtime Results in C++

When using city-pairs.txt, when we took out all the print statements in prim's function and kruskal's function, while using high resolution clock from the

chrono namespace to measure the duration in microseconds, these were our results:

city-pairs.txt

```
0   0   0   0   0   0   0   11  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   24  0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   64  0   0   12  0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   66  0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   44  0   0   0   0   72  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   16  0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   130 0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   48  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
11  0   0   0   0   0   0   0   40  0   0   0   0   0   0   0   0   0   53  0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   40  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   4   0   0   0
0   0   0   0   0   0   48  0   0   0   0   0   0   0   0   0   0   0   50  0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   23  0   0   0   0   0   52  0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   29  0   0   0   0   0   0   68  0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   14  0   0   0   73  0   0
0   64  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   44  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   52  0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   14  0   0   0   0   0   0   0   0   0   0   0
0   12  0   0   0   0   0   0   0   0   0   29  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   14  0   0   0   0   0   23  0   0   0   0   0   0   19
0   0   0   0   0   0   0   53  0   50  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   72  0   130 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   52  0   0   0   0   0   0   0   0   0   0   0   125 0   0
0   0   0   0   0   0   0   0   0   0   23  0   14  0   0   0   0   23  0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   16  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   114 0   0
0   0   0   0   0   0   0   0   0   0   0   68  0   0   0   0   0   0   0   0   0   0   0   0   0   0   68  0   0   0
24  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   17
0   0   0   0   0   0   0   0   4   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   68  0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   73  0   0   0   0   0   0   0   0   125 0   114 0   0   0   0   0   0
0   0   66  0   0   0   0   0   0   0   52  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   19  0   0   0   0   0   0   17  0   0   0   0   0
```

```
Kruskal C++ duration: 1812 microseconds
Prim C++ duration: 47009 microseconds
Assuming our algorithm for C++ is correct, our minimum spanning tree weight would be 1325
The way we coded the C++ algorithm it would not output a minimum spanning weight
```

10.txt

```
0   0   170 0   0   0   0   0   270 0
0   0   166 0   0   184 0   0   0   0
170 166 0   0   0   0   0   0   0   0
0   0   0   0   0   0   285 0   0   0
0   0   0   0   0   0   0   0   0   130
0   184 0   0   0   0   0   0   0   0
0   0   0   285 0   0   0   321 89  0
0   0   0   0   0   0   321 0   0   94
270 0   0   0   0   0   89  0   0   0
0   0   0   0   130 0   0   94  0   0
```

```
Kruskal C++ duration: 149 microseconds
Prim C++ duration: 471 microseconds
```

100.txt This test is located in the Test Cases folder

```
Kruskal C++ duration: 87976 microseconds
Prim C++ duration: 7147454 microseconds
```

With this (relatively) large amount of data, Kruskal's algorithm was significantly faster than Prim's, which makes sense due to the time complexity of Prim's with an adjacency matrix being $O(V^2)$ and Kruskal's being $O(ElogV)$, the high number of vertices is definitely going to be faster with $logV$ in the equation rather than V^2.

These tests were ran before knowing about the printed requirements and expectations of the assignment. Hoping that just the timing would satisfy, we learned otherwise. So, We began designing the algorithms in Python to provide a better user experience to search given .txt graphs and clearer results of the spanning tree and its value, along with the timing. While we completed the algorithms in c++, we were unfortunately unable to provide stable versions in python. Many small test cases worked for the algorithms, but as the tests got larger, some inconsistencies arose and circuts created, thus not providing a minimum spanning tree. Despite this problematic outcome, the small test cases below show to some extend the algorithms effectiveness. Even with the errors Kruskal's algorithm can still be clearly seen as a speedier approach to finding the minimum spanning tree as mentioned from our previous tests in c++.

0.4 Runtime Results in Python

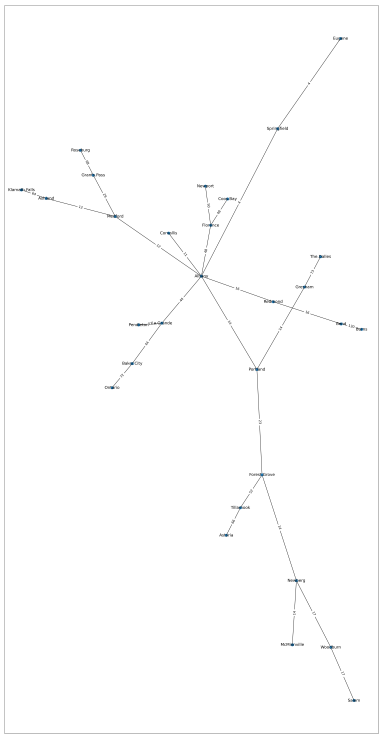
```
t.txt
Time to complete Kruskal's Algorithm: 0.009927996870828792 milliseconds.
Time to complete Prims's Algorithm: 0.13464899893733673 milliseconds.
B C 2
C A 3
MST size: 5

----
t1.txt
Time to complete Prims's Algorithm: 0.13464899893733673 milliseconds.
Time to complete Kruskal's Algorithm: 0.015855999663472176 milliseconds.
A B 2
B C 1
D A 4
MST size: 7

----

The 10.txt, 100.txt, and 1000.txt files had errors in the python version of the prim's algorithm

----
city-pairs.txt
Time to complete Prims's Algorithm: 410.36237200023606 milliseconds.
Time to complete Kruskal's Algorithm: 1.2155990007158834 milliseconds
The minimum spanning tree for the jupyter prims algorithm gave 1028,
although this was an incorrect weight it had the correct number of edges, and nodes.
```



Above is graph created using Networkx for city-pairs.txt prim's algorithm
The nodes that had errors were

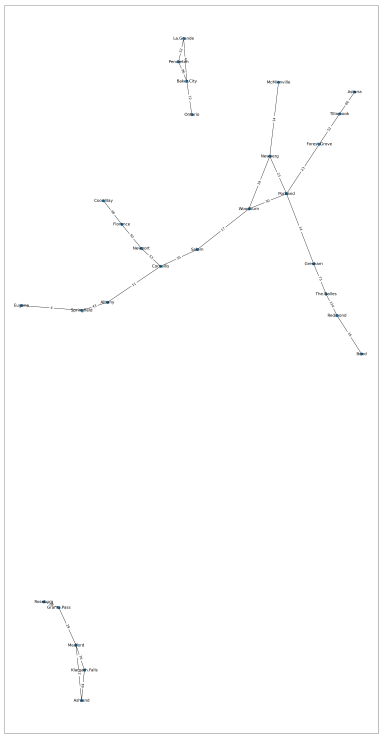
('Florence', 'Albany'): 48 , ('Medford', 'Albany'): 12,

('Newberg', 'Forest.Grove'): 14,('Portland', 'Albany'): 14,

('Redmond', 'Albany'): 16, ('Springfield', 'Albany'): 4,

('Woodburn', 'Newberg'): 17

However this still has the right number of edges and nodes.



Above is graph created using Networkx for city-pairs.txt Kruskals's algorithm

This algorithm still has problems with dealing with cycles and disconnects the wrong edges as you can see from the 3 separate trees.

0.5 Pitfalls with Kruskal's Algorithm in Python

With the variety of minds our group had we divided a lot of the work up to individual tasks. Approaches to solving the problems were varied and with that variation being a bane on the project overall problem to i

0.6 Suggestions and Recommendations

Because both algorithms have their own advantages and disadvantages, in order to select between the two you should start with identifying what the most common graph attributes are. Is the graph dense, sparse, sorted, or unsorted? These will help determine what algorithm will be the best in your given scenario.

Both implementations of C++ Prim and Kruskal's algorithms used adjacency lists to identify the minimum spanning tree. One implementation of our Prim's algorithm in python used an adjacency list, while Kruskal's in Python used a brute-force naive approach in an attempt to solve the minimum span. However, other types of adjacency lists, recursion, heaps (binary or Fibonacci) can be used to solve the problem faster. For instance, Prim's algorithm could be incorporated with a Fibonacci heap to solve the problem $O(|E| + |V| \log |V|)$.

Regarding the trials and tribulations these algorithms put our group through we learned from our test cases breaking. Taking thorough time to implement and test each algorithms to truly understand them and their outputs was difficult. There were many times where results of the algorithms produced false positives to the problem. So giving yourself more time to make thoughtful test cases is crucial.

We ended up making over 12 different test cases, and throughout testing found fixing one test could break the other cases.

Additionally, having some sort of graphing extension can greatly help verify your outputs. Without a doubt the graphs get more and more complicated as they grow in size to the point that extensive time is dedicated to manually reviewing the outputs and drawing graphs by hand.

Luckily we took the time to incorporate NetworkX into the python algorithm, which substantially helped see which edges/nodes were creating cycles, and if there were any extra trees.