UNIVERSITY OF PADOVA

DEPARTMENT OF MATHEMATICS

ADVANCED ALGORITHMS

COMPARISON OF PRIM, NAIVE KRUSKAL AND EFFECTIVE KRUSKAL'S ALGORITHMS

MOHAMMADMAHDI GHAHRAMANI 2041608

BILGE ZERRE 2041585

MARGARITA KOSAREVA 2041604

Academic Year

2021 - 2022

Abstract

Research Focus: Comparison of Prim, Naive Kruskal's and Efficient Kruskal algorithms for Minimum Spanning Tree(MST) problems.

This report has scrutinized the implementation and analysis of three Minimum Spanning Tree finders which are Prim algorithm implemented with Heap, Naive Kruskal algorithm with O(m.n) complexity, and Efficient Kruskal algorithm based on Union-Find. The data set consists of 68 randomly generated graphs, ranging in the size of vertices from 10 to 100,000.

The reader of the paper can expect to answer following questions:

- What are the execution times, the measured times with the asymptotic complexity, and the weight of the MST obtained by these three algorithms?
- How do the algorithms behave with respect to the various instances?
- Is there any algorithm that is always better than the others?
- Which is the most efficient algorithm over the three of them?

The final inferences to answer previously stated questions have been made by implementing and testing Prim, Naive Kruskal, and Efficient Kruskal algorithms on the same dataset by using Python as a programming language and using Google Colab as an environment.

Keywords: Prim, Naive Kruskal, Efficient Kruskal, Minimum Spanning Tree, Heap, Union-Find

Table of Contents

CHAPTER 1

Introduction

A minimum spanning tree or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. In order to find such a tree, several algorithms have been introduced. However, this study implements and analyzes in detail three of them. Prim's algorithm implemented using Heap, Naive Kruskal's algorithm and Efficient Kruskal's algorithm implemeted using Union-Find are these algorithms. Given a connected and edge-weighted undirected graph as input, both Prim and Kruskal's algorithms find the subset of the edges of that graph which have the following characteristics:

- Form a tree that includes every vertex in the graph.
- Have the minimum sum of weights among all the possible trees that can be formed from the graph.

There is an algorithm called *GENERIC-MST* that is the core part of Prim and Kruskal's algorithms. This algorithm uses a concept called *safe edge* to find the MST. It claims that if we repeatedly find the safe edge and get it united with the current subset of MST, it finally produces the MST. Prim and Kruskal's algorithms only differ in selecting this safe edge.

Before going through explaining these three algorithms, this study defines the concepts of a ***cut***, ***crossing***, ***cut respectiveness***, a ***light edge*** and a ***safe edge***. A cut in a graph is nothing but a partition of set of vertices V. In other words, a cut can be considered as a set $(S, V\text{-}S)$ where S is an arbitrary subset of nodes. It is said that an edge (u, v) crosses a cut $(S, V\text{-}S)$ if $u \in S$ and $v \in V\text{-}S$. It is also said that a cut respects a set A of edges if no edge of A crosses the cut. Given a cut, the minimum-weight edge that crosses the cut is called light edge.

There is a theorem saying that if we let $G=(V, E)$ be a weighted, connected, undirected graph and let A be a subset of E of some MST of G and let $(S, V\text{-}S)$ a cut that respects A and let (u, v) be a light edge for $(S, V\text{-}S)$, then (u, v) is a safe edge for A.

1. Prim's Algorithm

Similar to the Kruskal's algorithm, Prim has two version of implementation. The first one is the naive one where the algorithm just follows the mentioned theorem and in each iteration computes the light edge and adds it to the current subset of MST. The drawback associated with this naive approach

is being time consuming when repeatedly calculating the result of minimum operation. It causes the model to have a complexity of *O(mn)*, where *n* is the number of nodes and *m* is the number of edges. In the second version, the algorithm takes advantage of a new data structure, named Heap, to carry out the calculation of minimum operation. In this way the complexity of the algorithm reduces to *O(mlogn)*. This research does not go through the first naive implementation and whenever it refers to Prim's algorithm, it means the second version which uses Heap data structure. It is obvious that this algorithm needs a source node to start constructing the MST.

Heap by considering a method, named *extraction* and defining *key* for each object, removes the object with the smallest key in each iteration. It is the first part of the Prim's algorithm which has the complexity of *O(logn)*. Since this operation is repeated *n* times in a while loop, all in all the complexity of *O(nlogn)* could be imagined for the first part of the algorithm. In the second part on the other hand, the algorithm performs delete and insert operations which requires the complexity of *O(logn)* and is repeated proportional to the summation of vertices' degree(*2m*), which result in the complexity of *O(mlogn)* for this part. The complexity of *O(nlogn)* coming from the first part along with the complexity of *O(mlogn)* coming from the second part, the complexity of *O(mlogn)* is considered for the entire algorithm(we suppose m is bigger than n which is a true assumption for most of real-world's graphs).

2. Naive Kruskal's Algorithm and Efficient Kruskal's Algorithm

In Kruskal's algorithm, the idea is to expand a subset of MST whose current version is a forest this time, instead of tree. Unlike the Prim's algorithm, Kruskal does not start from a given source node and in each iteration it selects an edge which satisfies a specific condition. Hence, the current subset of MST is not necessarily a tree, but a forest.

In the Naive Kruskal's algorithm, edges are first sorted according to their weights. Then the algorithm picks an edge in nondecreasing order of weights which does not create a cycle and continues this until getting the MST. Since the current version of MST is a forest and we need to check cyclicity on this forest, our implementation should visit all connected part of this forest and check cyclicity on that part. If none of them has cycle, then the algorithm selects the edge and get it united with the current forest until constructing the MST. The drawback relating to this approach is that the process of cycle checking is time demanding and it is used repeatedly in the algorithm. It results in the complexity of *O(mn)* for the algorithm.

Aiming at speeding up the cycle checking, in the Efficient Kruskal's algorithm, we use *Union-Find* data structure. There are three supported operations in this data structure which are *Initialize*, *Find* and *Union*. Calling initialize operation, it creates a Union-Find data structure with each object chosen from an array *X* of objects and its own set. This operation requires the complexity of *O(n)*

since it must be done for each node. Find operation returns the name of the set that contains x, for a given an object x. This operation requires the complexity of *O(mlogn)*. Union operation receives two objects x, y and merge the sets that contain x and y in a single set. This operation requires the complexity of *O(nlogn)*. There are also two more operations in the body of this algorithm which are sorting and updating. The complexity of them are *O(mlogn)* and *O(m)* respectively. So, all in all the complexity of *O(mlogn)* is considered for the entire algorithm. Note that, we can provide the complexity of *O(mlogn)* for Find operation only if the Union part is implemented smart.

3. Implementation Choices

This study uses Python programming language and Google Colab environment to implement these algorithms. It does not use any Python library when coding algorithms. It only uses four libraries including ***time***, ***os***, ***numpy***, ***pandas*** and ***matplotlib*** for loading the data, measuring execution time and result visualization.

This paper considers two Python classes, named ***DFSCyclicity*** and ***GraphAlgorithms***. The second class includes the implementation of our three algorithms and inherits from the first class to check cyclicity in the Naive Kruskal's algorithm.

The first class takes advantage of Depth First Search(DFS) algorithm to label the graph's edges as a *Discovery* or a *Back* edge. Then by considering an attribute named *ancestor*, it checks if a graph is cyclic. As previously discussed, the input for cyclicity checking task is a forest which might be not connected and contain several connected parts. That is why this class firstly defines a private and recurssive method, named ***cycleCheckingOnConnectedGraph***, whose responsibility is to label each edge as *Discovery* or *Back* in a connected part of a graph and then uses another method, named ***cycleChecking***, to traverse all connected parts of a graph and call the previous method repeatedly to label all edges in the graph. This method at the end uses the *ancestor* attribute to check graph cyclicity.

The graph to be processed by this class is defined as a list of edges, each edge be a two-element list consisting of nodes on which the edge is incident. For example, if the graph has only three nodes and they create a cycle, the representation of this graph is [[1, 2], [1, 3], [2, 3]]. Since, most of the class's attributes, such as *lebel* or *ancestor*, are defined for edges and not for nodes, we considered *Dictionary* data type in Python to update edges' properties. The only attribute which is defined on nodes, is *visit* attribute which specifies if a node has already been visited. We considered again a *Dictionary* data type for this attribute(and not a list), because the graph G=[[1, 2], [2, 3]] is the same exact graph as the graph G'=[[14, 15], [15, 16]] and the code should not be sensitive about the way that nodes are represented. In a Python list, indexes start from 0 and go on in order, but in our case

the set containing the nodes does not necessarily start from an specific number and go on in order. So, using Python lists is not a good idea and we are better to use Python dictionaries.

The second class which contains all three algorithms firstly defines the required variables to carry out algorithms and by means of several private methods constructs the body of the algorithms. It defines ***extractMin*** and ***find_adjacent*** private methods to be used in ***PRIM*** method. It also defines ***findParent*** and ***Union*** private methods to be used in ***KruskalUF*** method. Regarding ***NaiveKruskal*** method, the only requirement is to inheriting from the first class to check cyclicity which has already been done.

The graph to be processed by this class is defined as a list of edges, each edge be a three-element list consisting of nodes on which the edge is incident and the weight corresponding to that edge. For example, if the graph has only three nodes and they create a cycle and the weight corresponding to each edge is the same value of 17, the representation of this graph is [[1, 2, 17], [1, 3, 17], [2, 3, 17]]. This class also uses *Dictionary* data types to define most of the attributes and variables, which are more flexible than lists in a sense of indexes(keys).

CHAPTER 2

Analysis

In this section, we will analyze the algorithms by answering to two questions. Moreover, we try to provide clues on the correctness of the algorithms.

1. Question 1

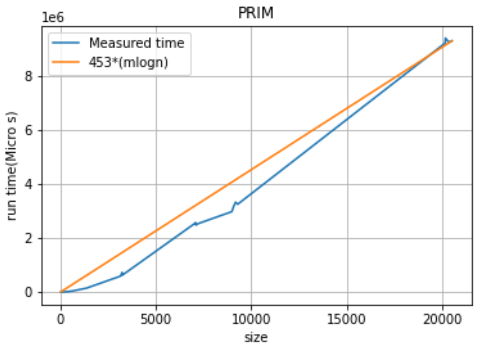
In this question, we are required to measure the execution times of the three algorithms and create a graph showing the increase of execution times as the number of vertices in the graph increases. Then we compare the measured times with the asymptotic complexity of the algorithms and finally we report the weight of the minimum spanning tree obtained the code.

1.1. Execution Times. In order to compute the execution time for each problem instance, we utilized the *time* library in python. Since measuring the execution time according to a single call is not accurate, the code defines an extra variable named ***numberOfRunningPerInput*** which specifies the number of times that an algorithm's execution time should be measured on each problem instance. It finally uses the average of these execution times for its corresponding problem instance. This variable is set to 30, 10 and 4 for the Efficient Kruskal, Prim and Naive Kruskal respectively. After computing these times, we need to plot them. Since, we already know the complexity associated with each algorithm, we calculate the horizontal axis of our plots according to these complexities. In other words, if we want to plot the executing times relating to the Prim, Naive Kruskal and Efficient kruskal's algorithm, we first compute the value of *mlogn*, *mn* and *mlogn* for each problem instance and plot the executing times with respect to these values by putting them on the horizontal axis.

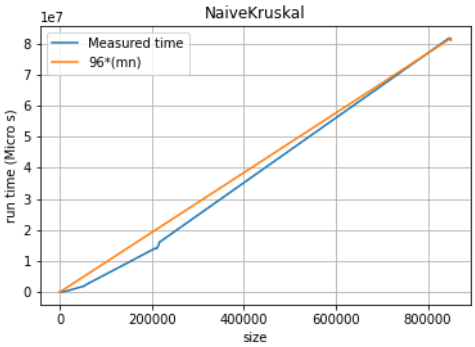
Since, the next section compares the measured time to the asymptotic complexity of algorithms and provide a more informative plots, this section postpones showing the graphs to the next section.

1.2. Measured Time Versus Asymptotic Complexity. Here, we first find a constant factor for each algorithm's complexity and plot the result to verify if our code is doing according to our expectation. To find the constant factors, the code defines a variable named ***c_estimates*** and selects the very last value(as the graph size grows) as the constant factor. Figure1.A shows that the Prim's algorithm is from a complexity of *O(mlogn)* with the constant factor of 453. Figure1.B shows that the Naive Kruskal's algorithm is from a complexity of *O(mn)* with the constant factor of 96. Figure1.C shows that the Efficient Kruskal's algorithm is from a complexity of *O(mlogn)* with the constant factor of 0.433. Note that the horizontal axis, size, consists of values computed according to the complexity of

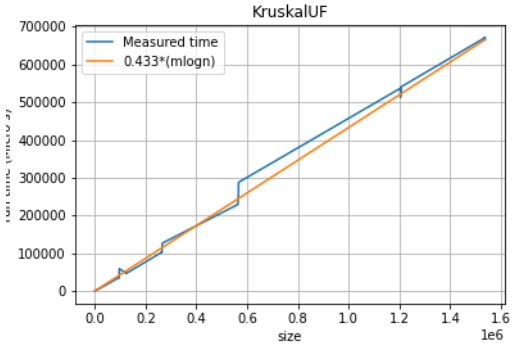
each algorithm and we already talked about that. Another important note is that, the constant factors have been computed according to the time unit of micro second.



(A) Prim



(B) Naive Kruskal



(C) Efficient Kruskal

FIGURE 1. **Algorithms' complexity analysis**

1.3. MSTs' Weight Obtained by the Code. In this section we run our code on each problem instance and report the total weight computed by the code. Here, we just report the results and do not review the code correctness. We will discuss correctness in the next section, right before the ***Question 2*** part. Figure2 reports the total weight associated with the MST found for each problem instance. We used a dataset of size 68 as problem instances.

	Problem_instance	MST_weight
1	input_random_01_10.txt	29316
2	input_random_02_10.txt	16940
3	input_random_03_10.txt	-44448
4	input_random_04_10.txt	25217
5	input_random_05_20.txt	-32021
6	input_random_06_20.txt	25130
7	input_random_07_20.txt	-41693
8	input_random_08_20.txt	-37205
9	input_random_09_40.txt	-114203
10	input_random_10_40.txt	-31929

(A)

	Problem_instance	MST_weight
11	input_random_11_40.txt	-79570
12	input_random_12_40.txt	-79741
13	input_random_13_80.txt	-139926
14	input_random_14_80.txt	-198094
15	input_random_15_80.txt	-110571
16	input_random_16_80.txt	-233320
17	input_random_17_100.txt	-141960
18	input_random_18_100.txt	-271743
19	input_random_19_100.txt	-288906
20	input_random_20_100.txt	-229506

(B)

	Problem_instance	MST_weight
21	input_random_21_200.txt	-510185
22	input_random_22_200.txt	-515136
23	input_random_23_200.txt	-444357
24	input_random_24_200.txt	-393278
25	input_random_25_400.txt	-1119906
26	input_random_26_400.txt	-788168
27	input_random_27_400.txt	-895704
28	input_random_28_400.txt	-733645
29	input_random_29_800.txt	-1541291
30	input_random_30_800.txt	-1578294

(C)

	Problem_instance	MST_weight
31	input_random_31_800.txt	-1664316
32	input_random_32_800.txt	-1652119
33	input_random_33_1000.txt	-2089013
34	input_random_34_1000.txt	-1934208
35	input_random_35_1000.txt	-2229428
36	input_random_36_1000.txt	-2356163
37	input_random_37_2000.txt	-4811598
38	input_random_38_2000.txt	-4739387
39	input_random_39_2000.txt	-4717250
40	input_random_40_2000.txt	-4537267

(D)

	Problem_instance	MST_weight
41	input_random_41_4000.txt	-8722212
42	input_random_42_4000.txt	-9314968
43	input_random_43_4000.txt	-9845767
44	input_random_44_4000.txt	-8681447
45	input_random_45_8000.txt	-17844628
46	input_random_46_8000.txt	-18798446
47	input_random_47_8000.txt	-18741474
48	input_random_48_8000.txt	-18178610
49	input_random_49_10000.txt	-22079522
50	input_random_50_10000.txt	-22338561

(E)

	Problem_instance	MST_weight
51	input_random_51_10000.txt	-22581384
52	input_random_52_10000.txt	-22606313
53	input_random_53_20000.txt	-45962292
54	input_random_54_20000.txt	-45195405
55	input_random_55_20000.txt	-47854708
56	input_random_56_20000.txt	-46418161
57	input_random_57_40000.txt	-92003321
58	input_random_58_40000.txt	-94397064
59	input_random_59_40000.txt	-88771991
60	input_random_60_40000.txt	-93017025

(F)

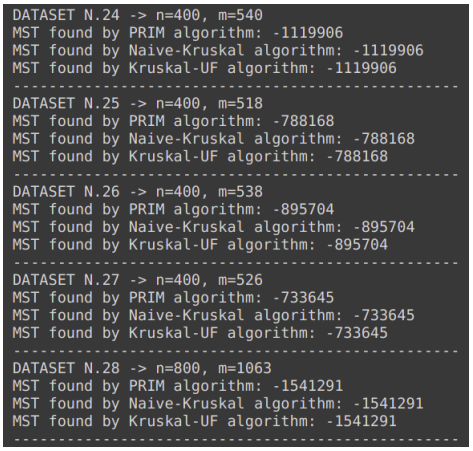
	Problem_instance	MST_weight
61	input_random_61_80000.txt	-186834082
62	input_random_62_80000.txt	-185997521
63	input_random_63_80000.txt	-182065015
64	input_random_64_80000.txt	-180793224
65	input_random_65_100000.txt	-230698391
66	input_random_66_100000.txt	-230168572
67	input_random_67_100000.txt	-231393935
68	input_random_68_100000.txt	-231011693

(G)

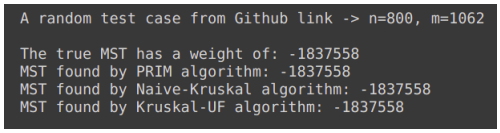
FIGURE 2. **MSTs' weight for problem instances**

2. Correctness of the Code

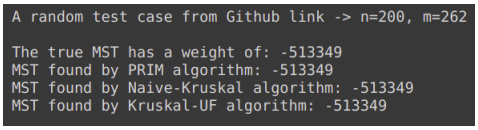
This study uses two simple steps to check correctness. The first one is to run all three algorithms on problem instances and compare the results. All algorithms should produce the same result. To some extent, this step can make us sure that algorithms are correct, because there is a slight probability that all algorithms produce the exact same result while the result is wrong. However, as the second step, this study uses several randomly selected graphs from the Github link of *https://github.com/beaunus/stanfordalgs/testCases/assignment1SchedulingAndMST/question3* and compare the result obtained by the code to the ground truth provided by this link. Our algorithms successfully passed these steps and we can claim that all three implemented algorithms are correctly-coded. Figure3.A shows a part of the results of the first step and Figure3.(B,C) reports a part of the results of the second step.



(A) Step 1



(B) Step 2



(C) Step 2

FIGURE 3. **Code correctness**

3. Question 2

In this part we comment on the results we have obtained. we first review how the algorithms behave with respect to the various instances. Then we discuss the efficiency of algorithms.

From the previous parts, it is obvious that as the size of problem instances grows, the execution time increases as well. By size, we mean the number of nodes and edges, since both of them affect the algorithms complexity. In Prim's algorithm, this time increasement is proportional to *mlogn* with the constant factor of 453. This indicator is *mn* with the constant factor of 96 for Naive Kruskal's algorithm and *mlogn* with the constant factor of 0.433 for Efficient Kruskal's algorithm.

Since the code of all algorithms are correct and produce the true total weight for MSTs, the criteria to be "better" than other algorithms is complexity. The less algorithm's complexity, the more efficient that algorithm is. If we want to compare these algorithm, it is obvious that the Naive Kruskal's algorithm is the worst one. Regarding Prim and Efficient Kruskal's algorithms, Prim's algorithm runs faster in dense graphs while Kruskal's algorithm runs faster in sparse graphs.

The most dense graph constructed by a set of nodes is a completed graph which has $\binom{n}{2}$ edges, where n is the number of nodes. If we define a dense graph as a graph with at least $\theta * \binom{n}{2}$ edges, for some $0 \leq \theta \leq 1$, and set θ even to a small number of $\theta = 0.3$, none of the 68 problem instances is dense. So, we can claim that the Efficient Kruskal's algorithm always performs the best. Prim's algorithm is in the second ranking and Naive Kruskal's algorithm is the worst. The constant factor of 453 for Prim's algorithm is very bigger than the constant factor of 0.433 for Efficient Kruskal's algorithm and it is an evidence to our claim.

CHAPTER 3

Conclusion

In this research, we implemented and reviewed three algorithms to solve the MST problem. These algorithms are Prim, Naive Kruskal and Efficient Kruskal. We tested the correctness of these algorithms and then measured the complexity of them. The algorithms are all correct and the measured times are according to the expected complexity of the algorithms. Given a dataset of 68 graphs, we ran all three algorithms and concluded that the Efficient Kruskal's algorithm is always the best algorithm. However, it does not mean that this algorithm is always better than Prim's algorithm. Prim's algorithm runs faster on dense graphs while the Efficient Kruskal's algorithm runs faster on sparse graphs. These 68 graphs are all sparse and so the Efficient Kruskal's performs the best. This paper also calculated the constant factor for each algorithm.

We believe that there are still too many graphs in the real-world's problems in which the number of nodes and edges is so high that even Prim and Efficient Kruskal's algorithms, implemented by this study, will run slowly. There might be several measurements to improve these algorithms and further research can be conducted to do so. One approach could be to get detailed in algorithms' code and consider some improvement to reduce the constant factor. Another approach could be to utilize common techniques in big data processing, such as using map reduce to handle the data. But, it requires high level consideration since in similar techniques as map reduce, the data is processed parallel and there is no connection between different parts of data. It does not make sense in our problem, since finding a MST for a graph requires to have knowledge about whole parts of data and not only a part of graph. However, it does not mean these techniques are totally useless. We can use them to introduce approximation algorithms which approximate the optimized solution. This is exactly like what we do in the clustering problem in which several approximation algorithms are introduced and coded in the content of map reduce. As the final note, we believe that if we can increase the number of times that an algorithm runs on a single problem instance and then compute the average execution time, we will have a better estimation on constant factor.