Timothy Young

Title: *Greedy Algorithm Analysis*

Prim vs Kruskal, CS-350

Table of Contents

1 Intro

1.1 Introduction

“Greedy” programming is a design technique typically applied to optimization problems. A popular problem in the realm of engineering is the traveling salesman, in which each path(graph edge) is associated with a unit of length. Each destination(node) will be weighted with some dollar amount, with a goal of finding the shortest path possible given the maximum amount of revenue to be made, ensuring that each node is visited only once and no cycles in the graph are formed.

This problem may be solved using a variety of techniques, for example, exhaustive search. If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles:

1. The number of spanning trees grows exponentially with the graph size.
2. Generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph.

For problems involved with optimization, a “greedy” programming approach may find an approximate solution faster than techniques relying on brute force alone. Two popular approaches that follow “greedy” programming and can be used to approximate such solutions are known as Prim’s and Kruskal's algorithms.

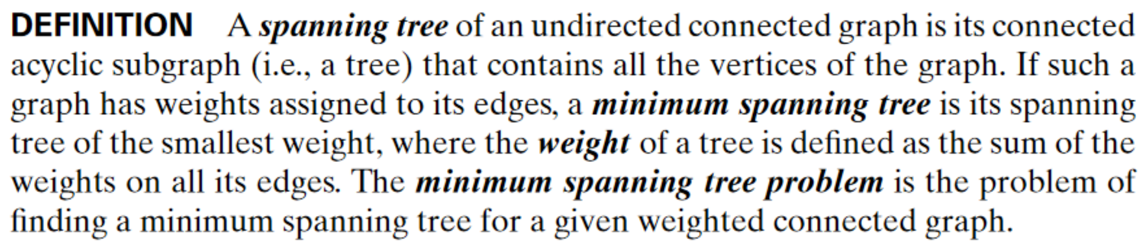
When solving problems for optimality, not every greedy technique always yields the exact optimal solution depending on the technique and constraints of the problem. The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. There are three general requirements that must me met for each step taken:

1. *feasible*: it has to satisfy the problem’s constraints.

2. *locally optimal*: it has to be the best local choice among all feasible choices available on that step.
3. *irrevocable*: once made, it cannot be changed on subsequent steps of the algorithm.

1.2 Background

We can represent the points given by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights. Then the question can be posed as the minimum spanning tree problem.



DEFINITION A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Two such algorithms always evaluate to an optimal solution given the minimum spanning tree problem:

1.2.1 Prim's Algorithm:

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. After picking an arbitrary vertex V, the algorithm picks the nearest path and stores this path and node into a priority que. The selection then moves onto the next node, picking the shortest path from that node that has not been previously visited. If a subtree is already in the priority que, then we know the node was already visited and we must check the paths of the most previously visited nodes to avoid creating a cyclic graph. Prim's algorithm always yields a minimum spanning tree, which can be proven via induction. The algorithm is listed as the following:

```
ALGORITHM  $Prim(G)$
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$    //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

1.3.1.1 Unsorted Array w/ Matrix

The efficiency of Prim's algorithm depends on the data structures implemented for the priority que and the graph itself. If the priority que is represented as an unsorted array, the runtime is said to be:

$$c(v) \rightarrow \Theta(|V^2|)$$

For each |v| -1 iteration, the priority que must be traversed to find, delete, and update the minimum.

1.2.1.2 Min-Heap w/ Adjacency List

The second option is to implement the priority que as a min-heap. A min-heap can be seen as a mirror image of a heap. One such way to implement one is to construct a heap and then to negate all of the key values given. A better representation of a min-heap is viewing it as a complete binary tree in which every element is less than or equal to its children. For binary trees, the largest element is typically the root of the tree(heap). A min-heap can give us all of the properties of heap, but the minimum value is now always at the root of the tree. Deletion and insertion for the smallest element are considered to be O(log(n)) operations, making this a faster representation with deleting the smallest element as a priority.

The graph must be represented as an adjacency list, with a priority que represented as a min-heap. The resulting run-time will be that of:

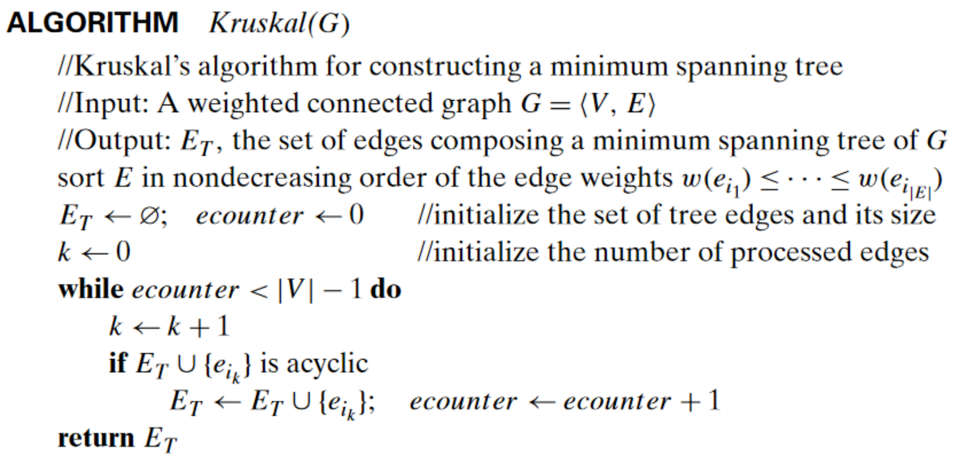
$$c(V) \rightarrow \ O(E\ |\log|(|V|))$$

The algorithm performs |V|-1 deletions of the smallest element and makes |E| verifications. The proof follows that if the deletion time is already in O(log(v)), then:

$$c(V) \rightarrow (|V| - 1 + E) O(\log(|V|)) = \ O(E\ |\log|(|V|))$$

1.2.2 Kruskal's Algorithm:

Kruskal's Algorithm is fairly similar to Prim's by growing a minimum spanning tree to solve a solution for the traveling salesman problem. In this implementation however, the subtrees are not automatically connected after entering the priority que. The vertexes entering the priority que are always acyclic, but they are not necessarily connected during the intermediate stages of the algorithm.



```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$      //initialize the set of tree edges and its size
    $k \leftarrow 0$                              //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;   $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

During the beginning stages, the weighted edges may be generated into disjoint subsets. There are two popular approaches to implementing disjoint subset data structures, Quick-Find & Quick-Union. After generating all of the potential disjoint subsets in sored order, we start connecting at the minimum(least) edge. This operation degrades into checking whether a node is already within its binary tree, allowing us to optimize the runtime further.

1.2.2.1 Quick-Find

For union by size, the runtime efficiency for n-1 unions and m finds is that of:

$$\mathrm{c}(\mathrm{n},\mathrm{m}) \rightarrow \mathrm{O}(n\ log(n+m))$$

Union-by-size: The size of the tree is measured by the number of nodes.

1.2.2.2 Quick-Union

For union by rank, the runtime efficiency for n-1 unions and m finds is:

$$\mathrm{c}(\mathrm{n},\mathrm{m}) \rightarrow \mathrm{O}(n + m\ log(n))$$

Union-by-rank: The size of the tree is measured by its height.

2 Test Plan

2.1 Data Analysis Process or Procedure

This test process will consider time analysis on both implantations of Prim & Krushkal's algorithm in Python 3. Prim has been implemented using a dictionary with a list of tuples as edge connections, representing an adjacency list as the graph. The priority que was implemented using the heapq library, which is essentially a minheap unless specified. Therefore, the runtime should follow, where E is the number of edges and V is the number of vertices:

$$c(V) \rightarrow O(E\ |\log|(|V|))$$

Krushkal's Algorithm has been implemented using a UnionFind data structure that follows Union by Rank. Therefore, the theoretical time complexity should follow:

$$\mathrm{c(n,m)} \rightarrow O(n + m\ log(n))$$

There are two test files for Krushkal's in this analysis. The first is the city-pairs file that was supplied. The second is an extremely similar file with different city names, numbers and is a third of the size of the "city-pairs" text file. The number of cities (n=7).

2.2 Analysis Results

Timing:

File/Function():	Prim()	Kruskal()	%diff
city-pairs.txt	.00108	.00319	33.33%
test1.txt		.00032	NA

Correctness:

File/Function():	Prim()	Kruskal()
city-pairs.txt	1675	1325
test1.txt		307

3 Conclusion

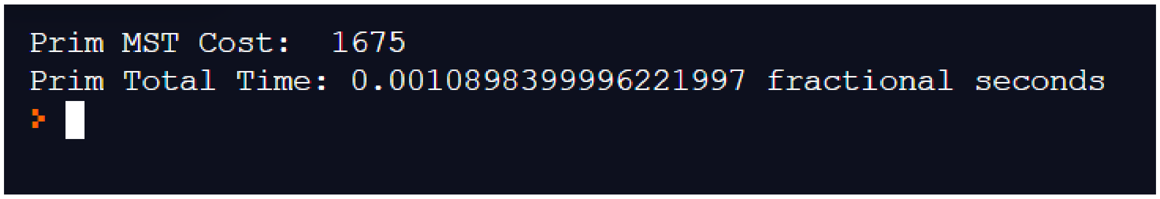
Given the theoretical runtime complexities for the two implementations, the timing values recorded using perf_counter() are not accurate and are fairly approximated. This is due to the runtime environment of the Replit IDE, and perf_counter() is a poor choice of timer. I spent a lot of time getting this implementation to work, and unfortunately ran out of time before the due date of this report to get the pseudo-critical edges working for Prim. The Kruskal implementation does output the optimal minimum spanning tree, but the Prim needs further development before we can consider running another test case against it. Prim should always output the optimal MST.

3.1 Recommendations

In the future I would like to implement the pseudo-critical edge functions for Prim() to output the optimal minimum spanning tree rather than just one from a given node. It would also be interesting to implement graphs in python to show the path of the MST for better visualization of these techniques. I attempted this during the first versions of the project, but constraints within the IDE as well as portable graph libraries made it hard to do so with the time given. Originally I made data structures in python without using libraries or built-in methods and parsed the text file into an actual adjacenyList structure. This was foolish and did not take full advantage of the built-in methods that Python 3.9 has to offer. Now that I have a ‘somewhat’ working implementation for Prim, I am excited to be able to go back and finish the critical edges, as well as improve my graph mapping skills.

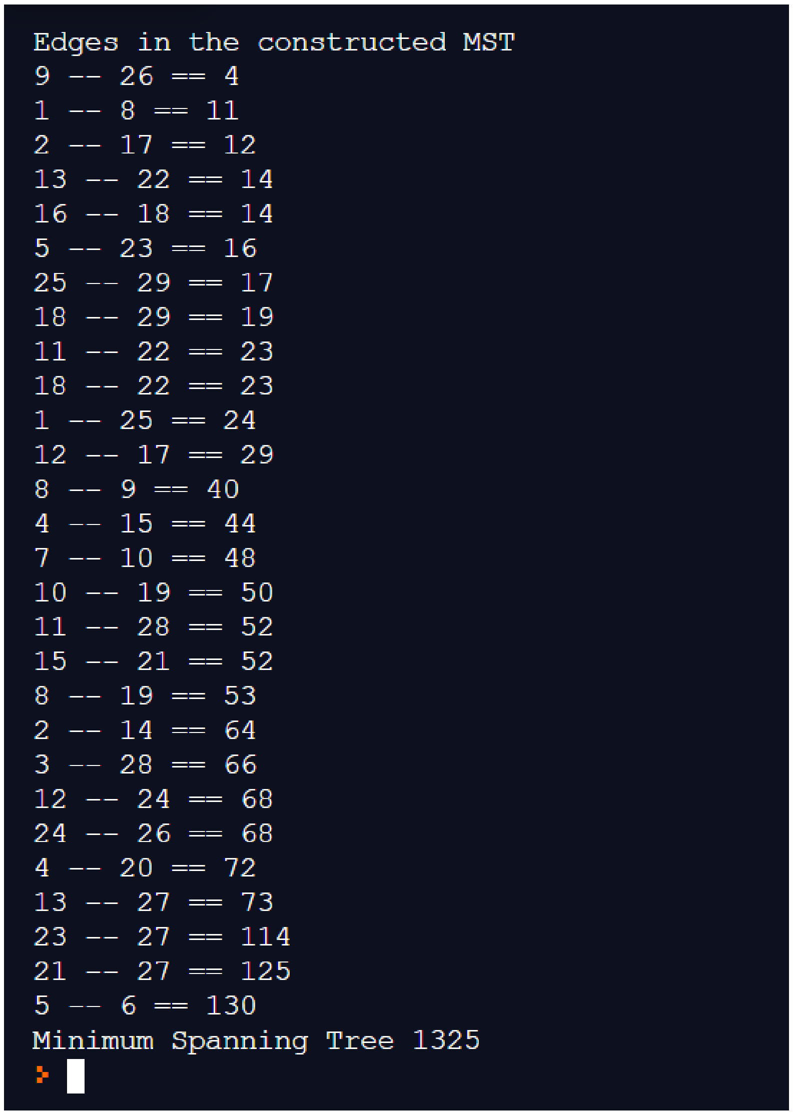
4 Figures

4.1 Prim Appendix

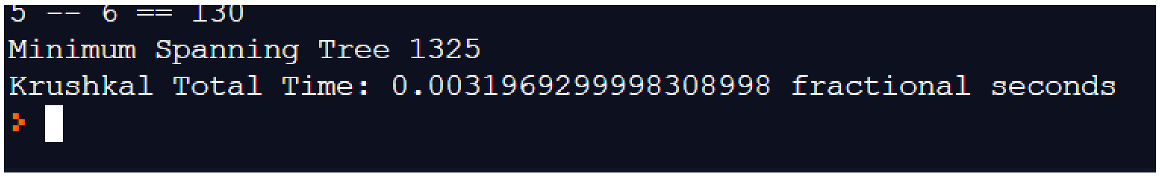


```
Prim MST Cost:  1675
Prim Total Time: 0.0010898399996221997 fractional seconds
```

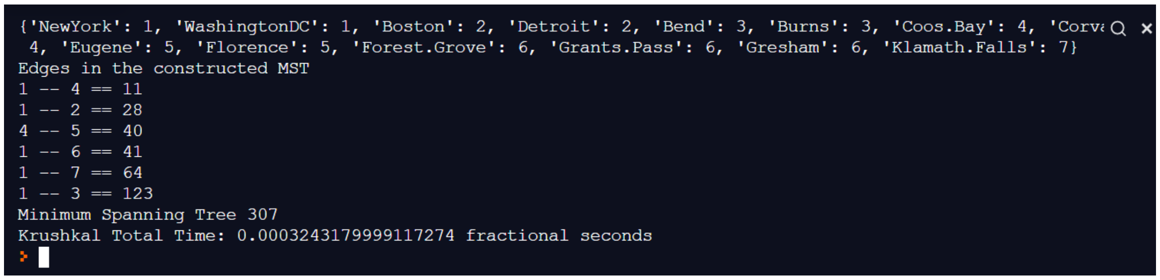
4.2 Krushkal Appendix



```
Edges in the constructed MST
9 -- 26 == 4
1 -- 8 == 11
2 -- 17 == 12
13 -- 22 == 14
16 -- 18 == 14
5 -- 23 == 16
25 -- 29 == 17
18 -- 29 == 19
11 -- 22 == 23
18 -- 22 == 23
1 -- 25 == 24
12 -- 17 == 29
8 -- 9 == 40
4 -- 15 == 44
7 -- 10 == 48
10 -- 19 == 50
11 -- 28 == 52
15 -- 21 == 52
8 -- 19 == 53
2 -- 14 == 64
3 -- 28 == 66
12 -- 24 == 68
24 -- 26 == 68
4 -- 20 == 72
13 -- 27 == 73
23 -- 27 == 114
21 -- 27 == 125
5 -- 6 == 130
Minimum Spanning Tree 1325
```



```
5 -- 6 == 130
Minimum Spanning Tree 1325
Krushkal Total Time: 0.0031969299998308998 fractional seconds
```



```
{'NewYork': 1, 'WashingtonDC': 1, 'Boston': 2, 'Detroit': 2, 'Bend': 3, 'Burns': 3, 'Coos.Bay': 4, 'Corv
 4, 'Eugene': 5, 'Florence': 5, 'Forest.Grove': 6, 'Grants.Pass': 6, 'Gresham': 6, 'Klamath.Falls': 7}
Edges in the constructed MST
1 -- 4 == 11
1 -- 2 == 28
4 -- 5 == 40
1 -- 6 == 41
1 -- 7 == 64
1 -- 3 == 123
Minimum Spanning Tree 307
Krushkal Total Time: 0.0003243179999117274 fractional seconds
```