CSE 5311 Project 1:
A Study and Comparison of Minimum Spanning Tree Algorithms

Manish Munikar
(UTA ID: 1001826846)

April 21, 2020

Abstract

Finding the minimum spanning tree (MST) of a graph or a network is essential for efficient design of networks such as computer networks, water supply, electrical grid, transportation networks and so on. *Kruskal's algorithm* and *Prim's algorithm* are the most popular algorithms to find the MST of a graph. This report contains a brief study, comparison, and implementation details of these two MST algorithms.

1 Introduction

Minimum spanning tree (MST) of a weighted, connected and undirected graph is the subgraph that is still connected and has the minimum possible total edge weight. Figure 1 shows a connected, weighted, and undirected graph with the MST highlighted.

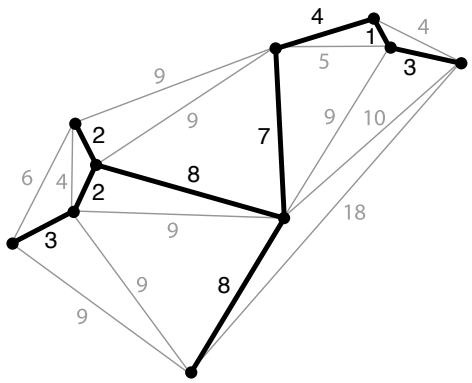

Figure 1: Example of a minimum spanning tree of a graph. (Source: [9])

Minimum spanning trees have many useful applications. It is most helpful in designing networks with minimum cost; network can be anything from telecommunication cable networks, electrical grid networks, water supply networks, computer networks, transportation networks, etc.

The problem of finding the minimum spanning tree of a connected graph has been studied throughout much of the 20th century and many algorithms have been proposed.[2] However, as of now, the most popular MST algorithms are the *Kruskal's algorithm* and the *Prim's Algorithm*.

In this project, I implemented the aforementioned minimum spanning tree algorithms and evaluated their performance on varying input sizes. The following sections describe these algorithms and their implementation details, and present the result of our evaluation.

2 Background

A *graph* is a structure containing a set of items where pairs of items may have some connection or relation. Items in graph are also known as *nodes* or *vertices*, and each connection or relation between a pair of vertices is called an *edge*. Edges can be *directed* or *undirected* depending on whether the relation is one-way or two-way respectively. A graph can be visualized as points denoting vertices and lines denoting edges, as in Figure 1.

Mathematically, a graph $G = \langle V, E \rangle$ where V is the set of vertices, and $E \subseteq \{\langle u, v \rangle \mid u, v \in V\}$ is the set of edges.

A *path* between two vertices in a graph is the sequence of edges between the two vertices. A *cycle* is a path that starts and ends at the same vertex. A graph is *weighted* if every edge as a weight (or cost) associated with it. A graph is *connected* if there exists a path between every pair of vertices in the graph.

3 Data Structures

To implement the minimum spanning tree algorithms, we need various data structures such as graph, heap, disjoint set, etc. In this section, I briefly describe the important data structures used in this project.

3.1 Graph

The mathematical definition of graph is given in Section 2. Graphs can be represented in computers in mainly two ways:

- **Adjacency list**: Vertices are stored in an array, and each vertex points to a linked list of vertices connected to it.
- **Adjacency matrix**: The graph is represented by a 2-dimensional matrix, in which rows are source vertices and columns are destination vertices. The values in the matrix is the weight of the edge from the corresponding source and destination vertices.

In this project, I implemented the graph data structure as adjacency list.
The graph data structure should provide the following basic methods:

- ADD-VERTEX(G, v): Adds a vertex v to the graph G.
- ADD-EDGE(G, u, v): Adds an edge between vertices u and v.
- REMOVE-VERTEX(G, v): Removes the vertex v from the graph G.
- REMOVE-EDGE(G, u, v): Removes the edge between the vertices u and v.
- ADJACENT(G, u, v): Tests if the graph G has an edge between the vertices u and v.
- NEIGHBORS(G, v): Lists all the vertices that v is adjacent to.

In computer science, many real world systems can be represented by a graph. Therefore, graph data structures are used to implement many graph and network algorithms such as minimum spanning tree, shortest distance, minimum cut, etc.

In this project, the graph data structure and the associated functions are implemented in `src/ds/graph.py`.

3.2 Heap

A *heap* a special type of balanced tree data structure that satisfies the heap property between parent and child nodes. There are two variants of heap: *min-heap* and *max-heap*. In a min-heap, every node is smaller than or equal to its children. Similarly, in a max-heap, every node is greater than or equal to its children. Figure 2 shows an example of a min-heap.

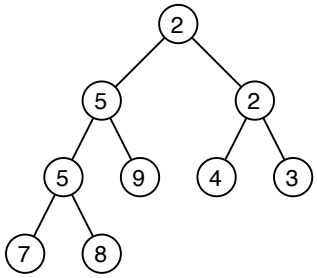

Figure 2: An example of a min-heap.

Since the root of a heap is either the maximum (or the minimum element), heaps are used as efficient priority queues.

Min-heaps provide the following basic operations:

- INSERT(H, x): Inserts an item x into the heap H.
- FIND-MIN(H, x): Finds the minimum element (root) of the heap.
- EXTRACT-MIN(H, x): Removes and returns the minimum element (root) of the heap. It also restructures the heap to maintain the heap property.

In some cases, the *key* (or *priority*) of a node in the heap may be specified separately. In such case, the heap may also support additional advanced operations such as:

- SET-PRIORITY(H, x, p): Sets (or updates) the priority of item x as p in heap H.

In this project, the heap data structure and the associated functions are implemented in `src/ds/heap.py`.

3.3 Disjoint Set

A *disjoint set* is a data structure that keeps track of a set of distinct elements partitioned into a number of disjoint (non-overlapping) subsets. This data structure is interesting because it provides very efficient (near constant time) operations to add new elements, merge two subsets, and to determine whether two elements belong to the same subset.

Each element in the disjoint set has two fundamental properties: a *key* and a *parent* pointer. Initially, all the elements' *parent* points to themselves. As elements are merged, one becomes the parent of the other. So we can easily find the *root* of the set by following the *parent* pointer.

Disjoint set data structure is a very useful to detect cycles in a graph, which is an important step in Kruskal's algorithm.

Disjoint sets provide efficient implementation of the following operations:

- MAKE-SET(x): Adds a new element with a unique key x and its *parent* pointing to itself.
- FIND(x): Returns the root element of the set x belongs to (by recursively following the *parent* pointer).
- UNION(x, y): Merges the roots of x and y if they are not already merged.

There are couple of optimization techniques to make all the above operations run in near-constant time. The first is *path compression*, which makes the *parent* point to root directly. The UNION(x, y) can be optimized by making sure that the shorter tree is attached to the root of the longer tree. This idea is known as *union-by-rank*.

In this project, the disjoint set data structure and the associated functions are implemented in `src/ds/disjointset.py`. I implemented both optimizations as well: path compression and union-by-rank.

4 Algorithms

In this project, I implemented and evaluated the two most popular algorithms for finding the minimum spanning tree of a weighted undirected graph.

In this section, I describe these algorithms in detail.

4.1 Kruskal's Algorithm

Kruskal's algorithm is a *greedy* algorithm to find the minimum spanning tree of a connected graph. By greedy, it means that this algorithm makes the locally optimal decision at every step based on heuristics.

This algorithm was first published by Joseph Kruskal in 1956 in *Proceedings of the American Mathematical Society*, pp. 48–50.[6]

The high-level operation of this algorithm is very simple:

1. Initialize an empty spanning tree.
2. Sort all the edges in ascending order.
3. Pick the smallest unseen edge. If this edge forms a *cycle* with the spanning tree formed so far, discard this edge. If not, add it to the spanning tree.
4. Repeat Step 3 until there are $(|V| - 1)$ edges in the spanning tree, where $|V|$ is the number of vertices in the graph.

A more detailed pseudocode of the algorithm is given in Algorithm 1.

In this project, Kruskal's algorithm is implemented in `src/alg/kruskal.py`.

4.1.1 Time Complexity

In this algorithm, the most expensive operation is the sorting of edges, which can be done in $\mathcal{O}(E \log E)$ time. All the other operations have lower time complexity. Moreover, since $E \in \mathcal{O}(V^2)$, $\mathcal{O}(\log E) = \mathcal{O}(\log V)$. Therefore, the time complexity of Kruskal's algorithm can be shown to be $\mathcal{O}(E \log V)$.

Algorithm 1: Pseudocode of Kruskal's algorithm

```
1 function KRUSKAL(G):
2     A ← ∅
3     foreach vertex v in G.V do
4         MAKE-SET(v)
5     foreach edge (u, v) in SORTED(G.E) do
6         if FIND(u) ≠ FIND(v) then
7             A ← A ∪ {(u, v)}
8             UNION(u,v)
9     return A
```

4.1.2 Space Complexity

Kruskal's algorithm needs space to store the disjoint set of vertices (which is $\mathcal{O}(V)$), the resulting minimum spanning tree (which is also $\mathcal{O}(V)$ because a spanning tree only contains $(V-1)$ edges), and the sorted list of edges (which will be $\mathcal{O}(E)$). So the overall space complexity of Kruskal's algorithm is $\mathcal{O}(V+E)$.

4.2 Prim's Algorithm

This is another very popular algorithm to find the minimum spanning tree of a graph. It was first developed by a Czech mathematician Vojtěch Jarník in 1930, and rediscovered and republished by Robert C. Prim in 1957 and again by Edsger W. Dijkstra in 1959.[5, 7, 1] This is also a *greedy* algorithm.

The high-level logic of this algorithm is as follows:

1. Initialize a spanning with a random vertex
2. From all the edges that connect the tree to new vertices, choose the one with minimum weight and add it to the spanning tree.
3. Repeat Step 3 until the spanning tree contains all vertices.

A more detailed pseudocode of the Prim's algorithm is given in Algorithm 2. Here, G is the graph and s is the starting vertex.

In this project, Prim's algorithm is implemented in `src/alg/prim.py`.

4.2.1 Time Complexity

This algorithm makes use of a minimum priority queue data structure which can be implemented in many ways: with array, linked list, heap or Fibonacci heap. Many priority queue operations are performed in this algorithm such as ENQUEUE, DEQUEUE, and changing an elements priority dynamically. These are the most critical operations of Prim's algorithm and the overall time complexity depends on how the priority queue is implemented.

Algorithm 2: Pseudocode of Prim's algorithm

```
1  function PRIM(G,s):
2      Q ← empty priority queue
3      foreach vertex v in G.V do
4          v.key ← ∞
5          v.p ← NIL
6          ENQUEUE(Q,v)
7      s.key ← 0
8      A ← ∅
9      while not IS-EMPTY(Q) do
10         v ← DEQUEUE(Q)
11         if v.p ≠ NIL then
12             A ← A ∪ {(v.p, v)}
13         foreach vertex u in NEIGHBORS(G,v) do
14             if u is in Q and WEIGHT(u,v) < w.key then
15                 w.key ← WEIGHT(u,v)
16                 w.p ← v
17     return A
```

Priority Queue Implementation	**Time Complexity**
Linear search	$\mathcal{O}(V^2)$
Binary heap	$\mathcal{O}((V+E)\log V) = \mathcal{O}(E \log V)$
Fibonacci heap	$\mathcal{O}(E + V \log V)$

Table 1: Time complexity of Prim's algorithm depending on the internal implementation of priority queue.

In this project, I implemented the priority queue using binary heap.

4.2.2 Space Complexity

Prim's algorithm needs to store a priority queue of size $\mathcal{O}(V)$ and the set of minimum spanning tree edges which is also of size $\mathcal{O}(V)$. Therefore, the overall space complexity of Prim's algorithm is $\mathcal{O}(V)$.

5 Evaluations

In this project, I implemented and evaluated Kruskal's and Prim's algorithms on graphs of varying sizes. I created random connected graphs with varying number of nodes (10 to 8000). I then ran these algorithms on each of these test graphs 10 times each and calculated the average running time. Figure 5 shows some samples of the test graphs with their minimum spanning tree highlighted.

The graph of run-times of both Kruskal's and Prim's algorithm on graphs of various sizes is shown in Figure 3. We can clearly see that the runtime of Kruskal's algorithm grows much faster than that of Prim's. When the graph size is small, both have similar runtimes, but as the number of vertices increases, Prim's algorithm runs much faster than Kruskal's.

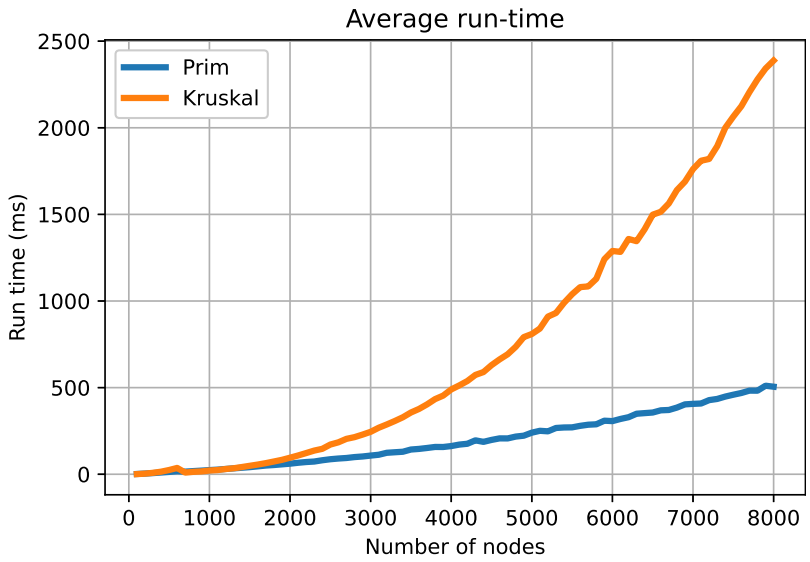

Figure 3: The runtime comparison of Kruskal's and Prim's algorithms.

When we look at the the plot in Figure 3, it seems that Prim's algorithm is always better than Kruskal's. But it's not the case. It actually depends on how *dense* the graph is, i.e., the number of edges per vertex. To see how Kruskal's and Prim's algorithm perform on varying degree of density, I fixed the number of vertices to 1000 and measured the run-time of both algorithms on graphs with varying number of edges. The result is shown in Figure 4. We can see from the graph that for *sparse* graphs, Kruskal's algorithm runs faster than Prim's. But as the graph get denser, the runtime of Kruskal's algorithm increases faster than Prim's. We can therefore conclude from this analysis that there is not a clear winner between Kruskal's and Prim's algorithm. For sparse graphs, Kruskal's algorithm is better whereas Prim's algorithm is more suited for dense graphs.

6 Implementation Details

I chose Python[8] (version 3) as the programming language to implement our project partly because of our existing familiarity with the language, but mostly because of the ease of programming it enables with its dynamic typing, clean and easy syntax, and support for interactive development.

I mostly implemented everything from scratch by using the default data structures that the Python programming language provides, such as list, set, dictionary, and tuple. However, I used some third-party libraries for easier visualization of the graphs since visualization is not a part of the algorithm implementation. Specifically, I used the `matplotlib`[4] and `networkx`[3] Python libraries for easier visualization and plotting.

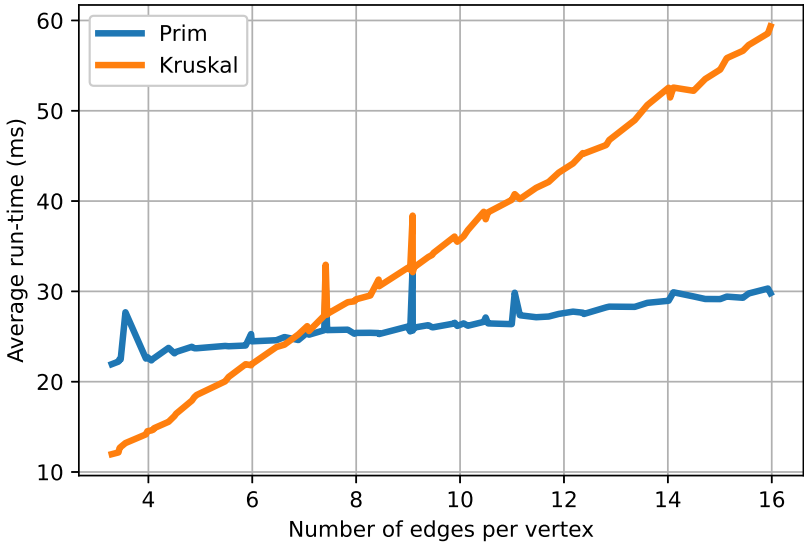

Figure 4: Performance of Kruskal's and Prim's algorithm based on graph density. In this experiment, the number of vertices is fixed to 1000.

I internally represented the graph data structure as an adjacency list, mostly to save some space because I don't want to allocate memory for non-existent edges. Also, finding neighbors is very straightforward with adjacency lists representation. In our implementation, a graph can be saved as (and read from) a text file. The file contains the list of edges (one per line). Each line contains 3 items separated by a space. The three items are source vertex, destination vertex, and edge weight.

When implementing heaps, I first implemented the in-place version. Then I realized that Prim's algorithm requires a priority queue where the priority of an element can change dynamically, which is very expensive with the default in-place implementation of heaps. So I implemented another out-of-place heap implementation which supports changing the priority of an element more efficiently by keeping track of the index of all the elements.

7 Conclusion

In this project, I implemented, analyzed, evaluated, and compared two very popular algorithms for finding the minimum spanning tree of a connected graph: Kruskal's algorithm and Prim's algorithm. Both of these are greedy search algorithms. I implemented these algorithms and all the required data structures in Python programming language. Actually, I chose this project because this project requires dealing with many interesting data structures like heaps, disjoint sets and graphs.

I wanted to see which of these two algorithm is better in terms of runtime, and I found out that it depends on how dense a graph is. If the graph is very sparse, Kruskal's algorithm runs faster. For everything else, Prim's algorithm is better.

By doing this project, I learned a great deal about how to efficiently implement various data structures and algorithms and how to convert pseudocode into working code.

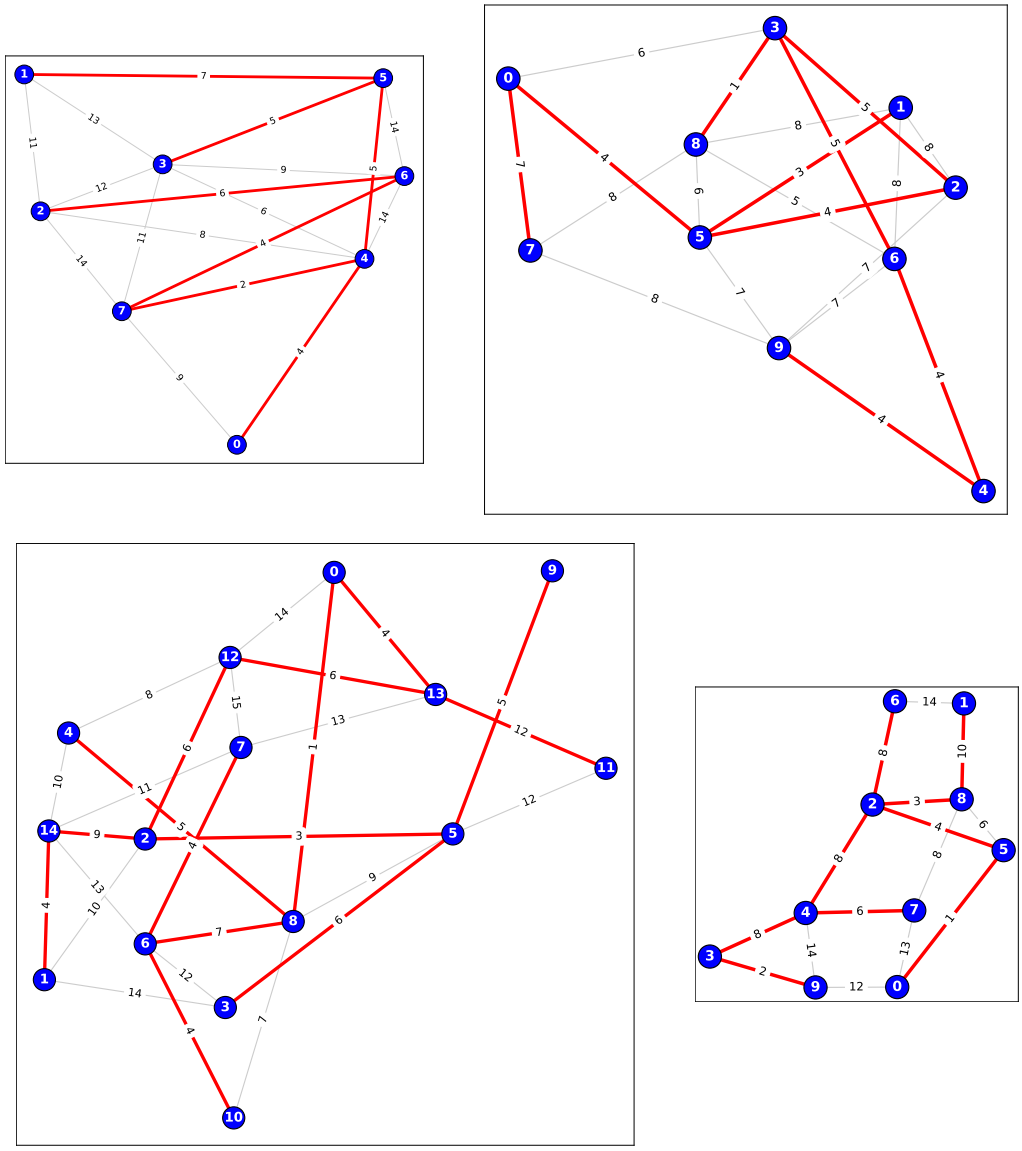

Figure 5: Some examples of connected graphs with their minimum spanning tree highlighted.

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[2] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[3] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*

(SciPy2008), pages 11–15, Pasadena, CA, USA, 8 2008.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[5] V. Jarník. O jistém problému minimálním [on a certain problem of minimization]. *Práce Moravské Přírodovědecké Společnosti*, 6(4):57–63, 1930.

[6] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[7] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 11 1957.

[8] Python Core Team. *Python: A dynamic open-source programming language*. Python Software Foundation, Vienna, Austria, 2015.

[9] Wikipedia. Minimum spanning tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Minimum_spanning_tree, 2020. [Online; accessed 2020-04-19].