THE UNIVERSITY OF WINNIPEG

APPLIED COMPUTER SCIENCE

Course Number GACS-7101

Course Name – Advanced Data Structure and Algorithm

PROJECT REPORT

MINIMUM SPANNING TREE USING PRIM ALOGITHM

Submitted by

Gunjan Basra

Student id – 3110114

Submitted to

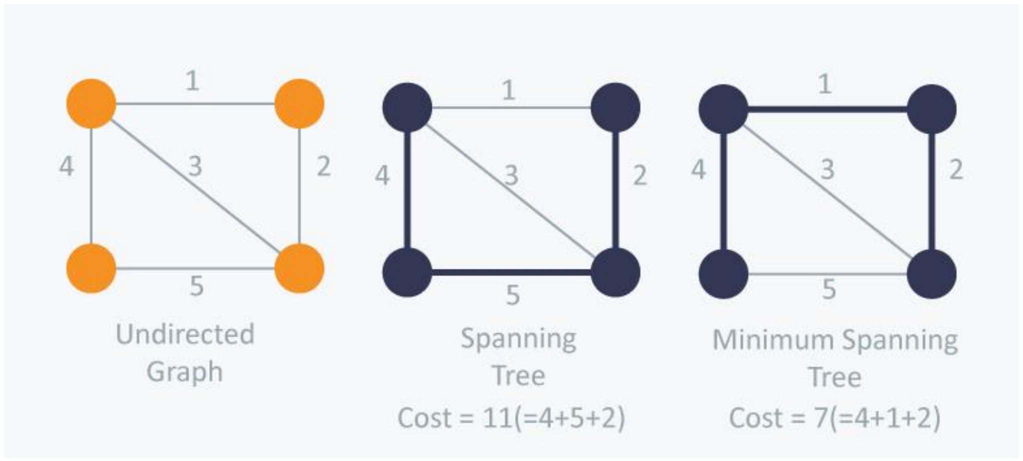
Dr. Mary Adedayo

Abstract

Finding **Minimum spanning tree (MST)** or **minimum weight spanning tree** in various types of networks is well studied algorithm in theory and practical way. A number of efficient algorithms have been already developed for this problem for example Prim's algorithm and Kruskal's algorithm. In this report I will discuss Prim's algorithm. Specifically, I present a algorithm that will find a minimum spanning tree of a graph with V vertices and E edges that runs in time *O*(ElogV) where the minimum number of edge-weight comparisons needed to determine the solution. The algorithm is simple and can be implemented on a pointer machine.
There are quite a few applications for minimum spanning trees. One example could be an telecommunication company trying to lay cable in different cities. If it is constrained to bury the cable only along certain route, then there would be a graph containing the points (e.g. telephone office in the city) connected by those paths. Some of the paths are less expensive, because they are shorter, or require the less cable to be buried; these paths would be represented by edges with lower weights. A spanning tree for that graph would be a subset of those paths that has no cycle i.e. acyclic graph but still connects every city; there possibility of having several spanning trees. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

1 Introduction

The spanning tree of a connected, undirected graph is a subgraph of the graph that is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with least sum of edge weights. The minimum spanning forest is a generalization of the minimum spanning tree for unconnected graphs. A minimum spanning forests consists of minimum spanning trees on each of the connected components of the graph.



(a) (b) (c)

Figure 1

Figure 1 shows the simple example for better understanding of minimum spanning tree.
Figure 1(a) is the given undirected graph in which we want to find the minimum spanning tree.

Figure1(b) demonstrate the spanning tree but the sum of weights of edges 4+5+2=11 is not minimum. But if we look at figure 1(c) the sum of weights is 4+1+2=7 is minimum.

Both Prim's and Kruskal's algorithm are greedy algorithm. But I implemented Prim's algorithm. A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy is not generally guaranteed to find globally optimal solutions to problems. For strategy advocates making the choice that is the best at the moment. Such a strategy is not generally guaranteed to find globally optimal solutions to problems. For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight. Prim is similar to Dijkstra's shortest-paths algorithm.

2 Related Works

2.1 Kruskal's Algorithm

Proposed by Kruskal in 1956, this algorithm follows directly from Corollary 1.1. Here are the main steps. To begin with the set A consists of only isolated vertices, and no edges (so, |V | "connected" components in all).

1. Sort the edges of E in non-decreasing order with respect to their cost.
2. Examine the edges in order; if the edge joins two components then add that edge (a safe edge) to A.

To implement Step 2, we do the following. Let e_i be the edge under consideration, implying that all edges with a lesser cost than $e_i = (a, b)$ have already been considered. We need to check whether the endpoints a and b are within the same component or whether they join two different components. If the endpoints are within the same component, then we discard the edge e_i . Otherwise, since it is the next lightest edge overall, it must be the lightest edge between some pair of connected components, and so we know from Corollary 4.1 that it is safe to add to A. We will need to merge these two components to form a bigger component.

To accomplish all of this, we will need some data structure which supports the following operations:

- Make-Set(v) - create a new set containing only the vertex v.
- Find(v) - Find the set which presently contains the vertex v.
- Merge(V_x, V_y) - Merge the two sets V_x and V_y together such that Find will work correctly for all vertices in merged set.

We can implement this data structure as follows. For each vertex we keep track of which component it lies in using a label associated with the vertex. Initially each vertex belongs to its own component, which is done with Make-Set. During the algorithm the components will be merged, and the labels of the vertices will be updated. Assume that we need to merge the two components Va and Vb corresponding to the end points a and b of the edge e_i = (a, b). We use Find(a) and Find(b) to get the sets Va and Vb respectively. We then call Merge which will relabel all of the vertices in one of the components to have the same labels as the vertices of the other. The component which we relabel will be the one which is smaller in size. Given such a data structure, we can implement Kruskal's algorithm as in Algorithm 1.2.

Let us analyze the complexity of Kruskal's algorithm. Sorting the edges takes O(|E| log |E|) time. The test for an edge, whether it joins two connected components or not, can be done in constant time. (In all O(E) time for all edges.) What remains is to analyze the complexity of merging the components which can be bounded by the total complexity of relabeling the vertices. Consider a particular vertex v, and let us estimate the maximum number of times this will be relabeled. Notice that the vertex gets relabeled only if it is in a smaller component and its component is merged with a larger one. Hence after merging, the size of the component containing v becomes at least double. Since the maximum size of a component is |V |, this implies that v can be relabeled at most log2 |V | times. Therefore, the total complexity of the Step 2 of the algorithm is O(|E| + |V | log |V |) time. These results are summarized in the following theorem.

Theorem 1.2 (Kruskal). A minimum (cost) spanning tree of an undirected connected graph G = (V, E) can be computed in O(|V | log |V | + |E| log |E|)=O(ElogV) time.

Kruskal's Algorithm

Input: Graph G = (V, E), cost function w
Output: A minimum spanning tree of G

```
1  A ← ø
2  for each vertex v ∈ V[G]
3    do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6    do if FIND-SET(u) ≠ FIND-SET(v)
7         then A ← A ∪ {(u, v)}
8              UNION(u, v)
9   returns A
```

Thus, the total time for Kruskal's algorithm is *O(V* lg *V* + *E* lg *V)* = *O(E* lg *V)*, which is asymptotically the same as for our implementation of Prim's algorithm.

2.2 Other related Algorithms

Minimum spanning tree of graph is found by various algorithm Boruvka's algorithm consists of running time of O(lg V), Chazelle MST algorithm running time of O(m .α(m, n)) etc. But my focus would be on Prim's algorithm and Kruskal's algorithm. Both of them have time complexity of *O(E* lg *V)* using ordinary binary heaps but if we use Fibonacci heaps, Prim's algorithm can be sped up to run in time *O(E + V* lg *V)*, which is an improvement if $|V|$ is much smaller than $|E|$.

3 Discussions

There is a telecommunication company trying to lay cable in different cities. They want expand their business to various cities in Canada. For this, they are trying to find minimum cost of cable required to connect to all the desired cities. If they use less cable it can save their money hence they trying to find the minimum distance between all cities to bury cable.

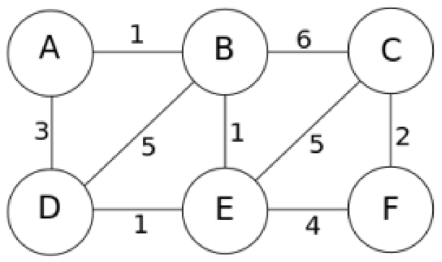
This problem can be solved by Minimum Spanning Tree. Assume the map as the graph where vertices represents the cities, edges represents the route from one city to another and weights represents the distance in kilometers / length of cable in kilometers. The graph will be undirected and connected. MST is implemented using Prim's algorithm Prim's algorithm is a special case of the generic minimum-spanning-tree algorithm. I used Binary Heap to implement Prim's algorithm.
Input: the number of cities and then cities name. After that which cities are connected and weight.
Output: the minimum spanning tree
In section 3.1 we will discuss about MST properties and in section 3.3 about how Prim's algorithm works.

3.1 Properties

3.1.1 Cut property

Theorem 3.1 Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V − S) be any cut of G that respects A, and let (u, v) be a light edge crossing (S, V − S). Then, edge (u, v) is safe for A.

Proof: Assume the contrary, i.e., in the figure at the bottom, make edge BC (weight 6) part of the MST T instead of edge e (weight 4). Adding e to T will produce a cycle, while replacing BC with e would produce MST of smaller weight. Thus, a tree containing BC is not a MST, a contradiction that violates our assumption.



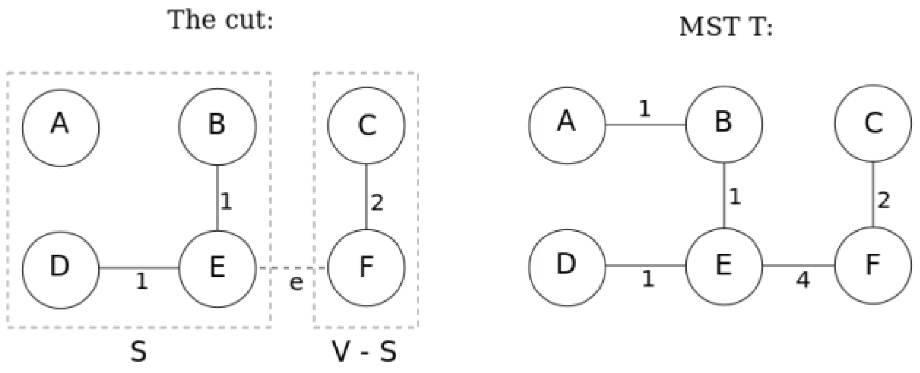


Figure 1: This figure shows the cut property of MST. T is the only MST of the given graph. If S = {A, B, D, E}, thus V − S = {C, F}, then there are 3 possibilities of the edge across the cut (S, V − S), they are edges BC, EC, EF of the original graph. Then, e is one of the minimum-weight-edge for the cut, therefore S ∪ {e} is part of the MST T.

3.1.2 Cycle property

Theorem 3.2 For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to a MST.

Proof Assume the contrary, i.e. that e belongs to an MST T1. Then deleting e will break T1 into two subtrees with the two ends of e in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge f of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree T2 with weight less than that of T1, because the weight of f is less than the weight of e.

3.2 Generic MST Algorithm

Assume we have undirected and connected graph G=(V,E) where V represents the vertices and E represents the edges of the graph G. The minimum weight function is defined as E -> R. As Prim's algorithm uses greedy strategy which grows the minimum spanning tree one edge at a time. The algorithm manages a set of edges *A*, maintaining the following loop invariant: A is a subset of some minimum spanning tree. At each step, we determine an edge *(u, v)* that can be added to *A* without isolating this invariant, in the sense that A∪{(u, v)} is also a subset of a minimum spanning tree. We call such an edge a **safe edge** for A, since it can be safely added to A while maintaining the invariant.

GENERIC-MST(G,w)

```
1 A ← ø
2 while A does not form a spanning tree
3 do find an edge (u, v) that is safe for A
4 A ← A ∪{(u, v)}
5 return A
```

We use the loop invariant as follows:
Initialization: After line 1, the set A trivially satisfies the loop invariant.
Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.
Termination: All edges added to A are in a minimum spanning tree, and so the set A is returned in line 5 must be a minimum spanning tree.

Intuitively this algorithm is straight-forward except for two pressing questions: What is a safe edge, and how do we find one? To answer these questions, we first need a few definitions.

Cut: A cut (S, V \ S) of G = (V, E) is a partition of vertices of V(see section 3.2).
Edge crossing a cut: An edge (u, v) ∈ E crosses the cut(S, V \ S) if one of its end point is in the set S and the other one in the set (V \ S).
Cut respecting A: A cut (S, V \ S) respects the set A if none of the edges of A crosses the cut.
Light edge: An edge which crosses the cut and which has the minimum cost of all such edges.

Theorem 3.2.1
Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V − S) be any cut of G that respects A, and let (u, v) be a light edge crossing (S, V − S). Then, edge (u, v) is safe for A.
Proof: assume T be a MST that includes A. there are two cases
Case 1: When edge(u,v) is in MST T. It does not contain the light edge (u, v), since if it does, we are done.
Case 2: When edge(u,v) is not in MST T. We construct another minimum spanning tree T' that includes A ∪ {(u, v)}, thereby showing that (u, v) is a safe edge for A.
If the edge (u, v) forms a cycle with the edges on the path p from u to v in T. Since u and v are on opposite sides of the cut (S, V −S), there is at least one edge in T on the path p that also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree T' = T − {(x, y)} ∪ {(u, v)}. We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing (S, V −S) and (x, y) also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T). \end{aligned}$$

But T is a minimum spanning tree, so that $w(T) \leq w(T')$; thus, T' must be a minimum spanning tree also. We have A □T', since A □ T and (x, y) not subset of A; thus, A ∪ {(u, v)} □ T'. Consequently, since T'is a minimum spanning tree, (u, v) is safe for A.

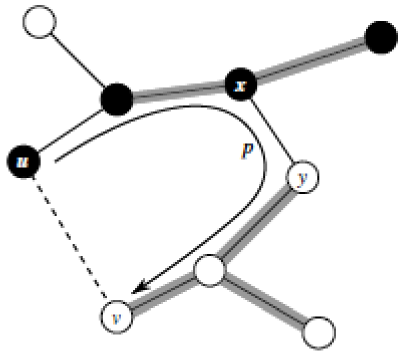


Figure 3.3 The proof of Theorem 23.1. The vertices in S are black, and the vertices in V − S are white. The edges in the minimum spanning tree T are shown, but the edges in the graph G are not. The edges in A are shaded, and (u, v) is a light edge crossing the cut (S, V − S). The edge (x, y) is an edge on the unique path p from u to v in T . A minimum spanning tree T _ that contains (u, v) is formed by removing the edge (x, y) from T and adding the edge (u, v).

The above theorem leads to the following corollary, where we fix a particular cut (i.e. the cut(C, V \ C)).

Corollary3.2.1
Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let C = (VC, EC) be a connected component (tree) in the forest GA = (V, A). If (u, v) is a light edge connecting C to some other component in GA, then (u, v) is safe for A.
Proof The cut (VC, V − VC) respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.

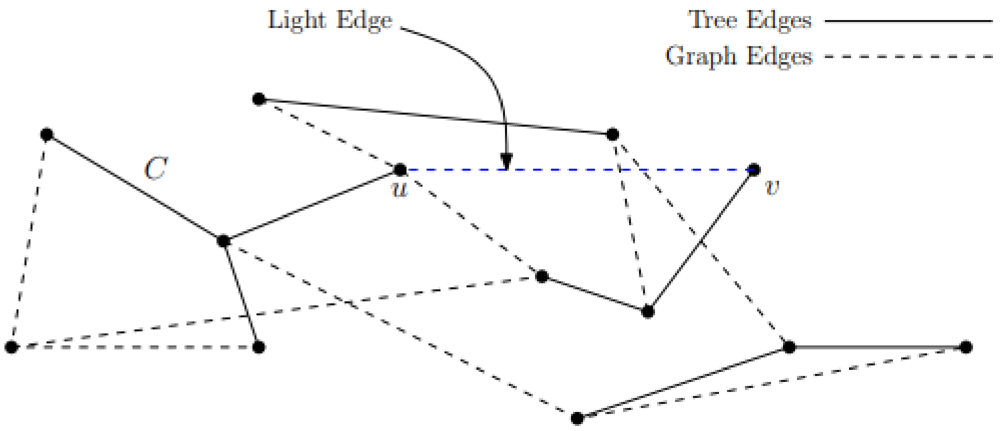


Figure 4.2: An example of Corollary 1.1. The edge (u, v) connects C to some other component of GA and is a light edge; it is therefore safe to add to the MST.

3.3 Prim’s Algorithm

Prim's (also known as Jarník's) algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The algorithm as developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm or Prim–Dijkstra algorithm.

Prim’s algorithm is very similar to Dijkstra’s single source shortest path algorithm, and, in fact, their complexity analysis will be the same. Here the set A at any stage of the algorithm forms a tree, rather than a forest of connected components as in Kruskal’s. Initially the set A consists of just one vertex. In each stage, a light edge is added to the tree connecting A to a vertex in V \ A.

The key to Prim’s algorithm is in selecting that next light edge efficiently at each iteration. For each v ∈ V \ A, we keep track of the least cost edge which connects v to A, and the cost of this edge is used as the “key” value of v. These key values are then used to build a priority queue Q. See Figure 1.2 for an example of these sort of light edges.

In each step of the algorithm, the vertex v with the least priority is extracted out of Q. Suppose that corresponds to the edge e = {u, v}, where u ∈ A, then observe that e is a safe edge since it is the light edge for cut(A, V \ A). We update A := A ∪ {e}. Finally, after extracting v out of Q, we need to update Q.

Prim’s Algorithm

Input: Graph G = (V, E), V: name of city and E: connection between the cities; cost function/cost of burring cable w, source city r
Output: A minimum spanning tree of G

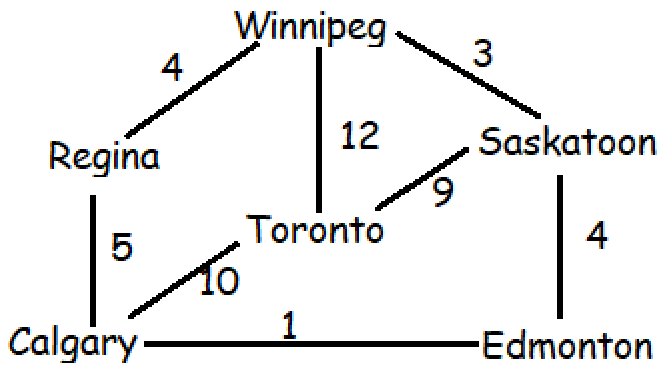
```
MST-PRIM(G,w, r)
for each u ∈ V[G]
   do key[u]←∞
      π[u]← NIL
key[r] ← 0
Q ← V[G]
while Q = ø
   do u ← EXTRACT-MIN(Q)
      for each v ∈ Adj[u]
          do if v ∈ Q and w(u, v) < key[v]
                then π[v]← u
                     key[v] ← w(u, v)
```

The vertices that are in the set A at any stage of the algorithm are the vertices in V \ Q, i.e., the ones that are not in Q. kev(v) is the weight of the light edge {v, π(v)} connecting v to some vertex in the MST A. Notice that the key value for any vertex starts at infinity, when it is not adjacent to A via any edge, and then keeps decreasing. Let us analyze the complexity of the algorithm. The main steps are the priority queue operations, namely decrease-key and extract-min. We perform |V | extract-min operations in all, one for each vertex. We also perform O(|E|) decrease-key operations, one for each edge. The following table shows the complexity of these operations depending on the type of priority queue you choose. These complexities are per operation, although the complexities of Fibonacci Heaps are amortized

	Binary Heap	Fibonacci Heap
Extract-Min	O(log n)	O(log n)
Decrease-key	O(log n)	O(1)

3.2.1 Execution

The execution of Prim's algorithm on the graph is shown below. The root vertex or starting city is Winnipeg. Red edges are in the tree being grown, and the vertices/cities in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree.



(a)

This shows the given graph G=(V,E) here V={Winnipeg, Regina, Saskatoon, Toronto, Calgary, Edmonton} and Edges={Winnipeg-Regina, Winnipeg-Saskatoon, Winnipeg-Toronto, Saskatoon –Edmonton, Saskatoon –Toronto, Regina-Calgary, Calgary-Edmonton, Toronto-Calgary }, root vertex is Winnipeg and weight w={ Winnipeg-Regina 4,

Winnipeg-Saskatoon 3,

Winnipeg-Toronto 12,

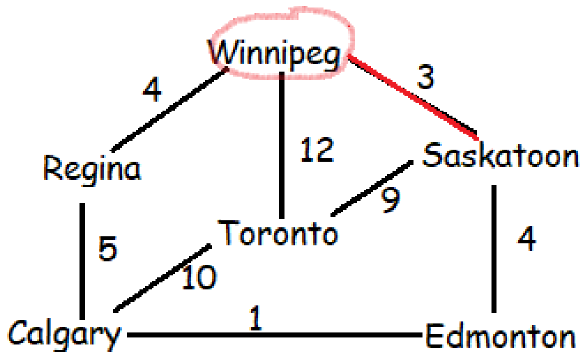
Saskatoon –Edmonton 4,

Saskatoon –Toronto 9,

Regina-Calgary 5,

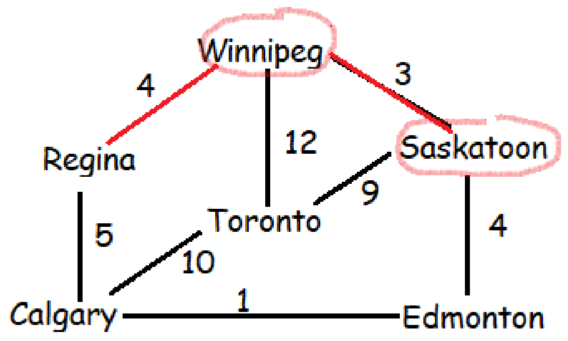
Calgary-Edmonton 1,

Toronto-Calgary 10}



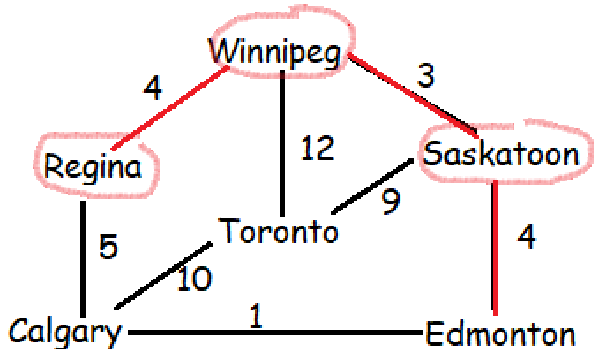
(b)

From Winnipeg we have Regina, Toronto and Saskatoon at 4, 12 and 3 weights respectively. As Prim's algorithm is greedy algorithm it will choose minimum weight i.e. 3. Add city to in priority queue Q= {Winnipeg, Saskatoon}



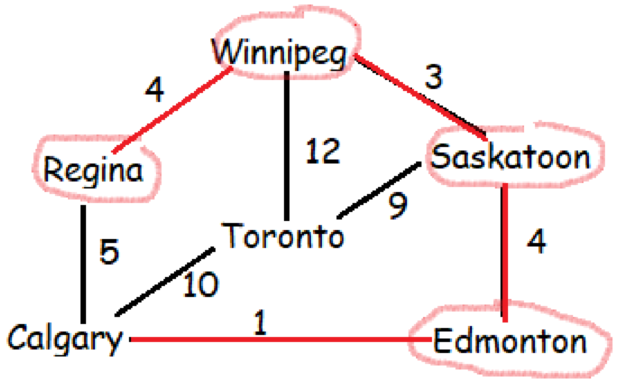
(c)

In figure c we have to find minimum weight from Winnipeg and Saskatoon which is Winnipeg to Regina with weight 4. Q={ Winnipeg, Saskatoon, Regina}



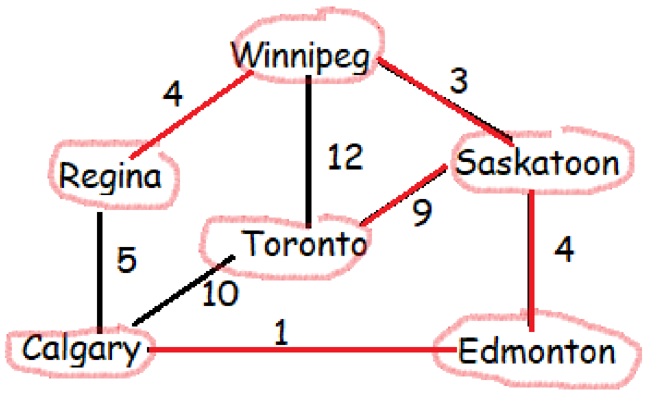
(d)

Now we have to minimum weight from Winnipeg, Regina and Saskatoon which is Saskatoon to Edmonton with weight 4. Q={ Winnipeg, Saskatoon, Regina, Edmonton}



(e)

Now we have to minimum weight from Winnipeg, Regina , Saskatoon and Edmonton which is Edmonton to Calgary with weight 1. Q={ Winnipeg, Saskatoon, Regina, Edmonton, Calgary}



(f)

Now we have to minimum weight from Winnipeg, Regina, Saskatoon, Edmonton and Calgary which is from Calgary to Regina with weight 5 but Regina is already in the priority queue so we will look at city which is not traversed yet and minimum weight to reach at that city which is from Saskatoon to Toronto with weight 9. Q= { Winnipeg, Saskatoon, Regina, Edmonton, Calgary, Toronto}

5 Experiments

The data for this experiment can be manually be inputted by the user in .txt files and certain formats. So, the user can check it manually for the result. It is important to install Netbeans IDE 8.2. To check the CPU and Memory Consumption, it checked using Windows Task Manager so, the actual needed foe processing can be seen and for this Running Time, it is checked by program execution time.

Tools for Experiment
This experiment is run on Asus A455L Notebook Series with Intel(R), Core ™ i5-5200u CPU @ 2.20 GHz (4 CPUs), Memory : 8192 Mb DDR3L RAM, Intel (R) HD Graphics 5500 Dedicated Graphic Card. The processor has a clock speed of 3.2 GHz with turbo-boost option up to 3.4 GHz clock-rate. The implementation of the serial version was done in Java programming language. The implementation was done using libraries of Java in NetBeans IDE 8.2. also need to install jdk 8u151 windows x64 and jre 8u231 windows x64.

Experiment Data
The sampling data is inputted by user through graph.txt file such as number of vertices, the vertex name like Winnipeg, the possible connection between vertex and the weight for every edge.

Data Sets
The sets of data used in experiment can be seen below in 4 tables

Table 1 Data Set 1 root= Winnipeg

Main city	**Connected city**	**Weight**
Winnipeg	Saskatoon	5
	Calgary	6
	Toronto	8
Saskatoon	Calgary	9
	Ottawa	1
Toronto	Calgary	8
Ottawa	Calgary	4

Output
Winnipeg ⟶ Saskatoon
Saskatoon ⟶ Ottawa
Winnipeg ⟶ Calgary
Winnipeg ⟶ Toronto

Table 2 Data Set2 root=Winnipeg

Main city	**Connected city**	**Weight**
Winnipeg	Regina	7
	Vancouver	9
	Ottawa	6
	Calgary	4
	Brandon	1
Regina	Vancouver	4
	Edmonton	8
Calgary	Vancouver	3
	Edmonton	9
Ottawa	Vancouver	11
	Edmonton	4

Output
Winnipeg ⟶ Brandon
Winnipeg ⟶ Calgary
Calgary ⟶ Edmonton
Calgary ⟶ Vancouver
Edmonton ⟶ Ottawa
Vancouver ⟶ Regina

Table 3 Data Set 3 root=Toronto

Main city	**Connected city**	**Weight**
Toronto	Ottawa	1
	Quebec City	6
	Winnipeg	10
	Saskatoon	1
	Vancouver	5
Ottawa	Quebec City	2
	Halifax	6
	Saskatoon	4
Winnipeg	Quebec city	3
	Halifax	4
	Vancouver	8
	Saskatoon	6
Quebec City	Halifax	3
Vancouver	Saskatoon	5
	Halifax	2

Output

Toronto $\longrightarrow$ Saskatoon
Toronto $\longrightarrow$ Ottawa
Ottawa $\longrightarrow$ Quebec City
Quebec City $\longrightarrow$ Halifax
Halifax $\longrightarrow$ Vancouver
Quebec City $\longrightarrow$ Winnipeg

Table 4 Data Set 4 root = Ottawa

Main city	Connected city	Weight
Ottawa	Waterloo	3
	Toronto	2
	Saskatoon	6
	Edmonton	9
	Halifax	10
Toronto	Waterloo	1
	Calgary	7
	Vancouver	12
	Regina	10
Winnipeg	Vancouver	10
	Saskatoon	6
	Regina	7
	Edmonton	4

Calgary	Saskatoon	2
	Waterloo	10
	Edmonton	3
Regina	Halifax	8
Edmonton	Halifax	10
	Vancouver	10
Vancouver	Saskatoon	4

Output

Ottawa ⟶Toronto
Toronto ⟶Waterloo
Ottawa ⟶ Saskatoon
Saskatoon ⟶Calgary
Calgary ⟶ Edmonton
Edmonton ⟶Winnipeg
Saskatoon ⟶Vancouver
Winnipeg ⟶ Regina
Regina ⟶ Halifax

The result for the total weight of every connected edge in MST Graph is shown in below table

Table 5 Total Weight of the edges from MST Graph using Prim's algorithm

Data Set	**Number of experiment**	**Total Weight**
1	1	17
	2	17
	3	17
	4	17
	5	17
2	1	18
	2	18
	3	18
	4	18
	5	18
3	1	12
	2	13
	3	12
	4	12
	5	11

	6	12
4	1	Error
	2	37
	3	38
	4	37
	5	35
	6	Error
	7	38
	8	37

Table 6: Time complexity for each experiment using Prim's algorithm whose running time complexity is O(ElogV)

Data Set	Number of Experiment	Running Time
1	1	1.22 seconds
	2	0.34 seconds
	3	0.12 seconds
	4	0 .45seconds
	5	0.12 seconds
2	1	2 seconds
	2	1.56 seconds
	3	1.45 seconds
	4	1.67 seconds
	5	1.33 seconds
3	1	5 seconds
	2	4.34 seconds
	3	3.67 seconds
	4	4.88 seconds
	5	3.32 seconds
	6	2.95 seconds
4	1	2.33 seconds
	2	5.67 seconds
	3	6.03 seconds
	4	5.09 seconds
	5	5.89 seconds
	6	2.63 seconds
	7	6.78 seconds

	8	5.89 seconds

Result Analysis

As a result of the experiment shown above for Prim's algorithm there is an error output for data set 4 experiment and also shows different weight results are because of the stability of the algorithm on this problem computer scientists are still discussing.
The result shows that if the graph is sparse (data set 1 & 2) the algorithm has better stability as compared with dense graph (data set 3 & 4).
The results from the experiments can be plotted on the graph. Where x-axis shows the no of experiment and y axis shows Running time.

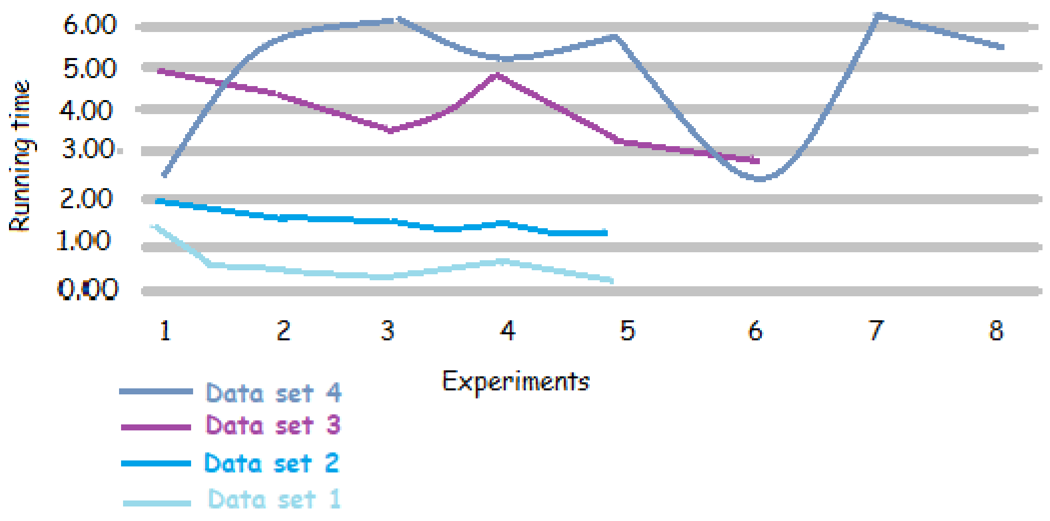


Figure .1 Prim's algorithm running time

github repository: https://github.com/Gunjan1995-ux/Project

6 Applications

Minimum spanning trees find applications in a range of fields. A few of them are listed below.

- A common application of MST is in construction of road or telephone networks. We would like to connect places/houses with the minimum length of road/wire possible. This is exactly the same as computing the minimum spanning tree.
- They find numerous applications in image processing. For example, if you have an image of cells on a slide, then you could use the minimum spanning tree of the graph formed by the nuclei to describe the arrangement of these cells.
- They are the basis for single-linkage clustering. Single-linkage clustering is a hierarchical clustering method. Each element is in its own cluster at the beginning. The clusters are then sequentially combined into larger clusters, until all elements end up being in the same cluster. At each step, the two clusters separated by the shortest distance are

combined. We will later see that this process basically mimics the Kruskal's algorithm for constructing the MST.

- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.

6 Future Works

The asymptotic running time of Prim's algorithm can be improved, however, by using Fibonacci heaps. We can perform an EXTRACT-MIN operation in $O(\lg V)$ amortized time and a DECREASE-KEY operation in $O(1)$amortized time. Therefore, if we use a Fibonacci heap to implement the priority queue Q, the running time of Prim's algorithm improves to $O(E+V \lg V)$.

There might be ways to further improve the design of the algorithms. For example, one might be able to think of a clever partitioning scheme for edge partitioning and vertex partitioning. Or, for parallel Prim's, one might be able to think of a way of caching the smallest edges leaving each connected component eliminating some duplication in work done

7 References

Swaroop Indra Ramaswamy & Rohit Patki, Distributed Minimum Spanning Trees, June 3, 2015.

https://www.statisticshowto.datasciencecentral.com/minimum-spanning-tree/

Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, 2nd Edition, 3rd Edition, The MIT Press, 2009.

Jeffrey D. Ullman, Alfred V. Aho, and John E. Hopcroft, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Com., London, 1969

https://ieeexplore.ieee.org/document/7543874/authors#authors

JONES, N. 1997. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, Mass.

KARGER, D. R., KLEIN, P. N., AND TARJAN, R. E. 1995. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM 42*, 321–328.

KING, V. 1997. A simpler minimum spanning tree verification algorithm. *Algorithmica 18*, 2, 263–270.

LARMORE, L. L. 1990. An optimal algorithm with unknown time complexity for convex matrix searching.*Infor. Process. Lett. 36*, 147–151.