Algorithms Fall 2019

Lecture : MST - Prim's Algorithm

Imdad Ullah Khan

Contents

1 Introduction

Problem: Find the cheapest network connecting n nodes. nodes could be towns (intercity highway connectivity), computers (LANs), servers (Fiber optic network). The network should connect all n nodes (there must be a path between all pairs of nodes). It should be cheapest least total length of cable, fiber or highways depending on application.

Immediate observation: If all lengths are non-negative, then the output network should not contain any cycle, because removing an edge from cycle doesn't disconnect the graph.

Proof. Let G' be a minimum cost solution to the network design problem connecting the node set V, then G' must be a tree. By definition, G' must be connected; we show that it also will contain no cycles. Indeed, suppose it contained a cycle C, and let e be any edge on C. We claim that $G \setminus \{e\}$ is still connected, since any path that previously used the edge e can now go. "the long way" around the remainder of the cycle C instead. It follows that G' is also a valid solution to the problem, and it is cheaper–a contradiction (because we removed a non-negative quantity). So output is a tree, it should be a spanning tree (meaning containing all nodes in V), it is called a minimum spanning tree. $\square$

Recall the following basic facts about trees.

Fact 1. *A tree on n vertices has $n-1$ edges.*

Fact 2. *A connected graph on n vertices must have at least $n-1$ edges.*

Fact 3. *A connected graph on n vertices and $n-1$ edges must be a tree.*

Fact 4. *In a tree every pair of vertices has a unique path between them.*

Fact 5. *Removing any edge from a graph disconnects the tree, (a tree is a minimally connected graph).*

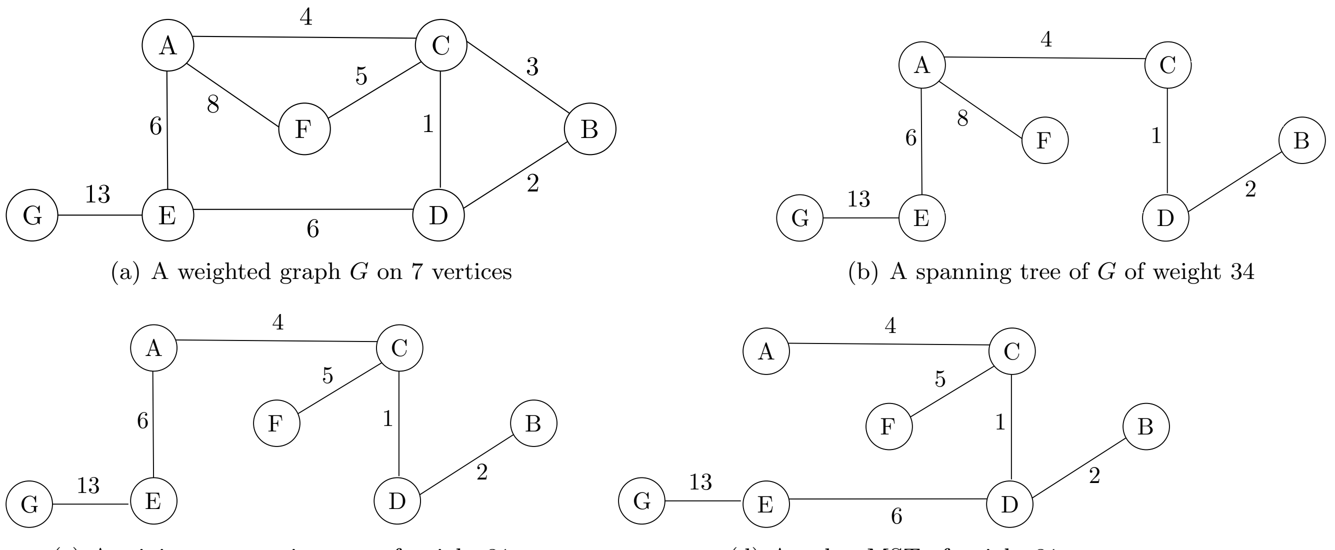
Fact 6. *Adding any edge to a tree create a cycle, (a tree is a maximally acyclic connected graph).*

Review the lecture notes for Dijkstra algorithm for single source shortest path problem to recap definitions of weighted graphs, weights of paths. Given a tree $T = (V, E')$, we define weight of T to be $wt(T) = \sum_{e \in E'} w(e)$.

2 The minimum spanning tree problem

Input: An undirected weighted graph $G = (V, E), w$, where $w : E \to \mathbb{R}$. Note that unlike Dijkstra weights don't have to be non-negative.
Output: A spanning tree $T = (V, E')$, with $E' \subseteq E$ such that $w(T)$ is minimum among all spanning trees of G.

Note that a graph can have exponentially many spanning trees. For example Figure 2 shows a graph and three of its spanning trees, two of them are minimum spanning trees. In general a graph could have actually more than one spanning trees too.



(a) A weighted graph G on 7 vertices

(b) A spanning tree of G of weight 34

(c) A minimum spanning tree of weight 31

(d) Another MST of weight 31

Figure 1: A graph and 3 of its spanning trees

We may assume that G is connected, otherwise the problem is not well defined, in this case we could talk about minimum spanning forests. Graph connectivity can be determined (and its components can be found) easily by using a pre-processing BFS or DFS. If we find a minimum spanning tree of each component, their union will be a minimum spanning forest.

Note that Edges could be negative as well. To make it easier We assume all weights are distinct, algorithms stays the same with arbitrary tie-breaking but analysis is just a little complicated.

As G could have exponentially many spanning trees, the brute force search for the minimum one is not a feasible strategy. It turns out that greedy strategy works here too.

2.1 Prim's Algorithm

Greedy Strategy :

1. We start with a vertex say s and start growing a tree from s.
2. We add one edge at each step and span one more vertex. So we attach a node to the already grown tree (like the conquered region R in the Dijkstra algorithm).

The greedy part is that every step we have a partial tree on vertices in R, we select a vertex $v \notin R$ that is attached to R by the cheapest crossing edge, i.e. v is a vertex that minimizes $argmin_{e=(u,v),u\in R,v\notin R} w(e)$.

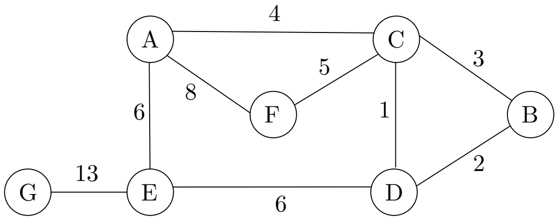
A highlevel pseudocode, without any data structure and implementation details, of the first algorithm, discovered by Prim is given below.

Algorithm 1 : Pseudocode of Prim's algorithm without any data structure

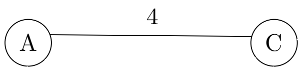
```
$R \leftarrow s$
$T \leftarrow \emptyset$                                              ▷ Begin with an empty tree
while $R \neq V$ do
    Select $e = (u, v) \in E$ such that $w(e) = \min_{u\in R, v\notin R} w(uv)$    ▷ Select the minimum weight edge
crossing the cut $R$
    $T \leftarrow T \cup \{e\}$
    $R \leftarrow R \cup \{v\}$
```

2.2 An example run of the Prim's algorithm

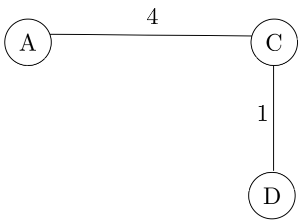
We run the above pseudocode on our example given above. Note that in our example edges weights are not unique. As remarked earlier, the algorithm is exactly the same, its analysis is slightly more complicated, for the analysis we will assume that all weights are unique. See problem set to fill in the details of the analysis when weights are actually not distinct.



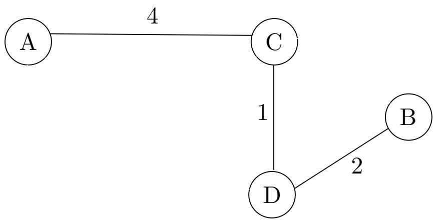
(a) The input graph G



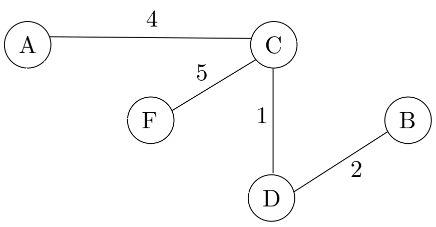
(b) Add C to R



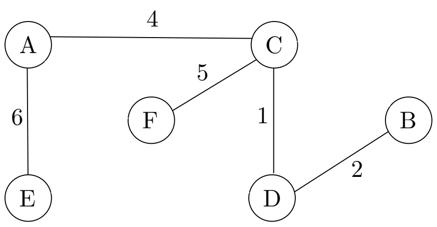
(c) Add D to R



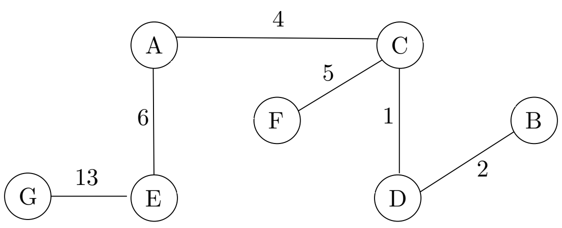
(d) Add B to R



(e) Add F to R



(f) Add E to R



(g) Add G to R

Figure 2: An test run of the pseudocode on G

3 Cuts in graphs

In order to prove the correctness and optimality of Prim's algorithm, we first review a few basic facts about cuts in a graph. Given a graph G a cut is a subset $S \subset V$, usually denoted by $[S, \bar{S}]$. In order to avoid triviality, we always assume that $\emptyset \neq S \neq V$. An edge (u, v) is said to be crossing the cut $[S, \bar{S}]$ if $u \in S$ and $v \in \bar{S}$. Since a cut is just defined to be just a subset, it is easy to see that there are 2^{n-1} different cuts in a graph of order n. We will study the following property of cuts.

3.1 Properties of Cuts

Fact 7 (Empty cut Property:). *A graph G is not connected iff there is a cut in G with no crossing edges, (empty cut).*

This actually gives us new way to characterize the graph connectivity. We usually define a graph to be connected if there is a path between any pair of vertices.

Proof. If part of empty cut lemma is easy, if there is any empty cut $[S, \bar{S}]$, then there is no path between any two vertices $x \in S$ and $y \in \bar{S}$.

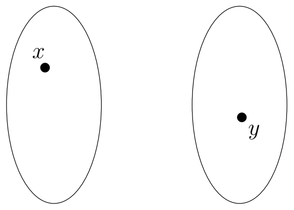


Figure 3: An empty cut, the left (right) side is the connected component containing x (y)

To see the only if part; if G is disconnected, means there are two vertices x and y with no path between them, so get the component of x and the component of y (meaning all vertices reachable from x and those reachable from y) there can't be any edge between these two components, hence we found an empty cut. □

Fact 8 (Double Crossing Property:)**.** *If a cycle crosses a cut, then it has to cross it at least twice, (actually it must cross it an even number times)*

Proof. Let $x_1, x_2, \dots, x_k, x_1$ be a cycle. Suppose there is an edge on this cycle (say (x_i, x_{i+1})) that crosses the cut $[S, \bar{S}]$. WLOG suppose that $x_1 \in S$ and that $x_i \in S$ and $x_{i+1} \in \bar{S}$ suppose this is the first edge crossing the cut $[S, \bar{S}]$, meaning $x_1, x_2, \dots, x_i \in S$. Consider the vertices $x_{i+1}, x_{i+2}, \dots, x_k, x_1$ at least one of these vertices is in S (we know for sure that $x_1 \in S$). Suppose the first vertex in S is x_j, this means that $x_{j-1} \in \bar{S}$ hence the edge (x_{j-1}, x_j) also crosses the cut. □

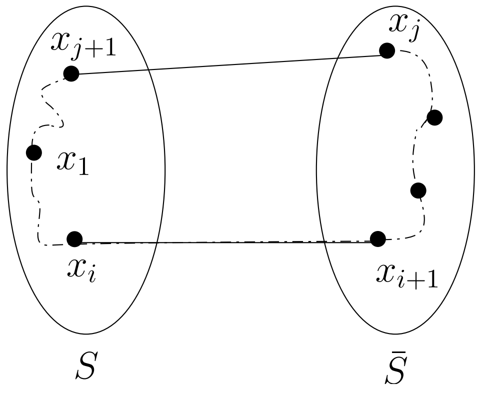


Figure 4: A cycle starting in S once reaches $\bar{S}$ must have another edge to come back to S

Fact 9 (Lonely cut corollary:)**.** *If e is the only edge crossing some cut $[S, \bar{S}]$, then e is not in any cycle*

Proof. Because if e was part of some cycle, then by the double crossing lemma this must have a dual (another edge crossing the cut $[S, \bar{S}]$), which will make it not the lonely edge crossing the cut. □

Finally, we use the following critical property about cuts, which is essential for proving optimality of Prim's algorithm.

Fact 10 (The cut property (blue rule))**.** *For any cut $[S, \bar{S}]$ in a graph G, if e is a minimum weight edge crossing the cut $[S, \bar{S}]$, then e belongs to a MST of the graph.*

Usually we will assume that all weights in G are unique, this implies that the minimum spanning tree of the graph is unique (see problem set). In this case the statement of the cut property would be as follows.

Fact 11 (The cut property (unique weights))**.** *For any cut $[S, \bar{S}]$ in a graph G, if e is the minimum weight edge crossing the cut $[S, \bar{S}]$, then e belongs to the MST of G.*

Let's take the cut property for granted, and use it to prove optimality of Prim's algorithm, we will give it's proof later.

3.2 Correctness and optimality of Prim's algorithm

Let T be the output of Prim's algorithm. We need to prove that T is a minimum spanning tree.

Lemma 12. *T is a spanning tree of G*

To prove this we will prove that T is connected, T has no cycle and that T is a spanning tree. Indeed we prove the following loop invariant for the Prim's algorithm.

Lemma 13. *After any iteration of the Prim's algorithm, T is a spanning subgraph of vertices in R.*

Proof. We prove it by induction on $|R|$ (or loop counter i). The base case is quite easy, $|R| = |\{s\}| = 1$, and $T = \emptyset$, T trivially is a spanning tree of s (or the subgraph including the vertex s).

If it is true for $|R| = k$, then in the next iteration, since we add one vertex to R with an edge connecting it to vertices already in R, hence T spans the vertices in R. □

Next we need to prove that in the end T is a spanning subgraph of V, in other words we prove that at some point R will become equal to V. Since in every iteration we add a vertex to R the maximum number of iteration is $n - 1$, so if there always exists an edge between the current R and $\bar{R}$, then we will always add one vertex to R. This is easy to see, because if during some iteration there is no edge between R and $\bar{R}$, this means that the cut $[R, \bar{R}]$ is an empty cut, hence the graph is disconnected by the empty cut property. A contradiction.

Now, we show that T doesn't contain any cycle. This follows from the fact at any iteration, we added one edge from the cut $[R, \bar{R}]$, and by the lonely cut property one edge cannot create a cycle.

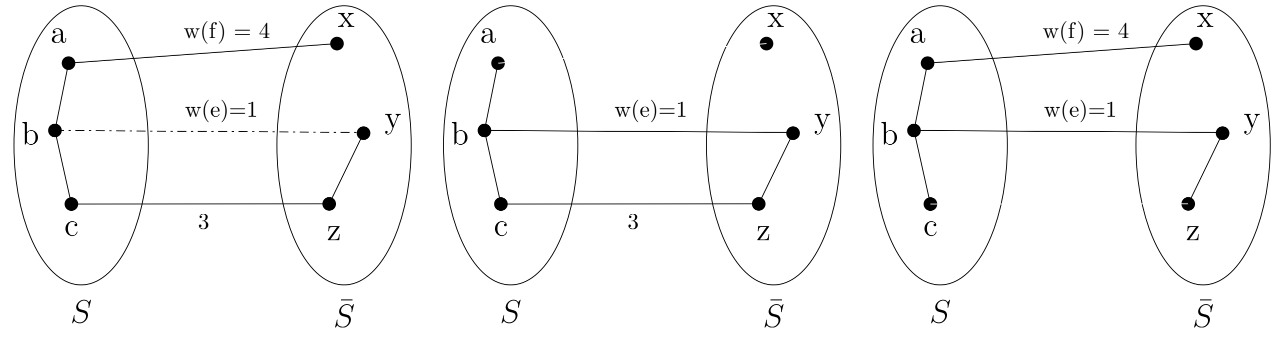
Lemma 14. *T is a minimum spanning tree*

Proof. We essentially have to prove that the greedy choice of selecting the minimum crossing edge at each iteration was an optimal choice.

This follows directly from the cut property. Recall, the cut property states an edge is guaranteed to be part of the MST if there is a cut across which it is the lightest. In each iteration, the algorithm added an edge which was the lightest edge across the cut $[R, \bar{R}]$, hence by the blue rule this must be part of the MST. So the algorithm just utilizes the cut property at each iteration. □

3.3 Proof of the cut property

Proof. We will use the so-called exchange argument to prove the cut property. Suppose T^* is the MST of G and $e = (u, v)$ is the lightest edge across some cut $[S, \bar{S}]$ ($u \in S, v \in \bar{S}$). Assume for the sake of a contradiction that T^* does not contain e. Note that T^* must have an edge from the cut $[S, \bar{S}]$, because otherwise by the empty cut property T^* is not connected (hence not a spanning tree), suppose $f = (u', v') \in T^*$. Can we exchange f with e, obviously $T^* \setminus \{f\} \cup \{e\}$ has smaller weight as $w(e) < w(f)$, but careful! $T^* \setminus \{f\} \cup \{e\}$ might not be a spanning tree (see e.g. Figure 5)



(a) edge $e = (b, y)$ is lightest crossing edge (b) Replacing f by e creates a cycle (c) Replacing (c, z) by e gives a lighter tree

Figure 5: Replacing an arbitrary heavier crossing edge with the lightest crossing of a cut. The resulting subgraph is not a spanning tree.

For the correct argument we will find such a removable f. Consider $T^* \cup \{e\}$, there must be a cycle C in this graph (because a tree has this property, that adding any edge to a tree creates a cycle, it is an edge maximal acyclic connected graph). Let $f = (u', v')$ be the heaviest edge on this cycle C, and consider $T^* \setminus \{f\} \cup \{e\}$. This graph is connected as removing an edge from a cycle doesn't disconnect any pair of vertices (there is always an alternate path; instead of taking the (u', v') edge, take the other part of the cycle to reach from u' to v'). $w(T^* \setminus \{f\} \cup \{e\}) = w(T^*) - w(f) + w(e) < w(T^*)$, as e is lighter than f (f is the heaviest on the cycle also containing e).

A contradiction to the assumption that T^* is optimal.

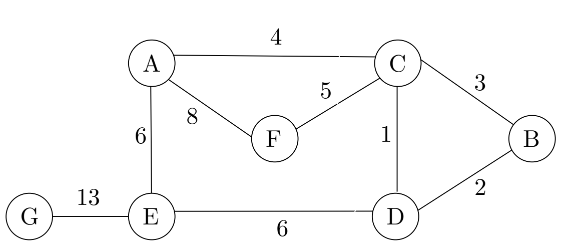
□

4 Implementation and Runtime

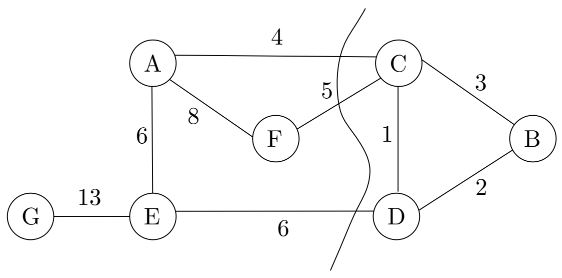
4.1 Naive Implementation

In the naive implementation of the pseudocode above does $n - 1$ iterations, as in each iteration it adds a vertex to R and stops when $R = V$. In each iteration, it finds an edge which is the lightest edge across the current cut $[R, \bar{R}]$. A cut could have almost all of the edges of the graph. For instance consider a complete graph when $|R| \sim n/2$, there are about $n^2/4$ edges crossing the cut. So in the worst case, we would have to do a find min over almost all the edges which takes $O(m)$ time. So the runtime of the naive implementation of the above pseudocode $O(nm)$.

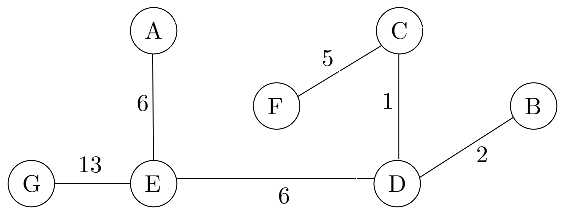
The trouble is $O(m)$ find-min, we need to do fast minimum computation. Notice that in each iteration R and $\bar{R}$ changes just by one vertex each. In other words suppose we added the vertex u into R in the



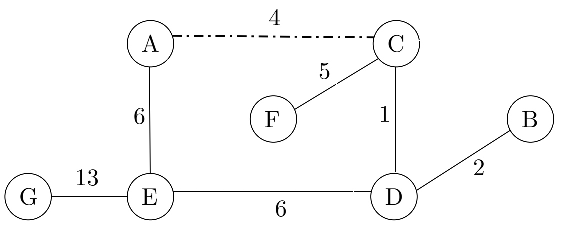
(a) The graph from our earlier example



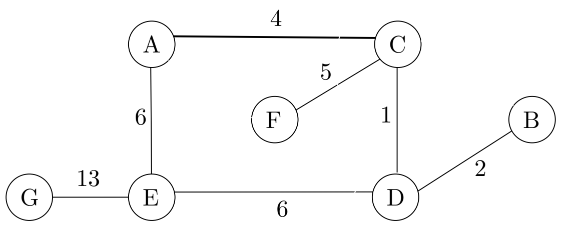
(b) $S = \{B, C, D\}$ lightest crossing edge is (A, C)



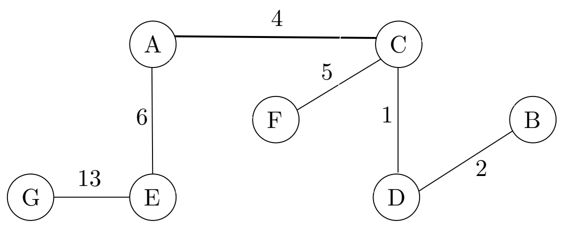
(c) A spanning tree of weight 33



(d) Adding (A, C) creates cycle A, C, D, E



(e) (E, D) is heaviest edge on this cycle



(f) A spanning tree of weight 31

Figure 6: An example of the exchange argument

current iteration (and added the edge (v, u) to T in the current iteration), i.e. (v, u) was the lightest edge crossing the cut $[R, \bar{R}]$. Now in the next iteration we have to find the lightest edge across the cut $[R \cup \{u\}, \bar{R} \setminus \{u\}]$. Instead of doing a complete find-min on the new cut, notice that only edges from u to other vertices $\bar{R}$ are the new crossing edges and edges going from vertices in R to u are no more crossing edges.

We reduce the cost of the repeated find-min operations by using a more suitable data structure. Recall the minimum binary heap implementation of priority queue ADT with the following operations:

- Insert(x, k) : $O(\log n)$
- DecreaseKey(x, k) : $O(\log n)$
- ExtractMin() : $O(\log n)$

4.2 Implementation using Min-Heap

For implementation of Prim's algorithm using priority queue, we use the vertices in $\bar{R}$ as elements of the heap. The key for each vertex is the smallest crossing edge incident to the vertex. In each iteration to select the node to be added to R we do an ExtractMin() operation. Suppose the vertex u is added to R then for the next iteration, for a neighbor x of u that is in $\bar{R}$, the edge (u, x) might be a lighter crossing edge. So we decrease the key of x to $w(ux)$ using DecreaseKey$(x, w(ux))$ operation.

Algorithm 2 : Prim's algorithm

```
$R \leftarrow s$
$T \leftarrow \emptyset$                                        ▷ Begin with an empty tree
$H \leftarrow \emptyset$                                        ▷ Initialize an empty min-heap

for $v \in V$ do
    $v.key \leftarrow \infty$                                   ▷ Initialize keys of all vertices to $\infty$
    $H$.Insert$(v, v.key)$
    $prev(v) \leftarrow null$        ▷ $prev(v)$ keeps the other end of min crossing edge incident on $v$
$H$.DecreaseKey$(s, 0)$
while $R \neq V$ do
    $v = H$.ExtractMin()
    for $(v, u) \in E(G)$ do                                    ▷ for each neighbor of $v$
        if $u.key > w(vu)$ then
            $H$.DecreaseKey$(u, w(vu))$
            $prev(u) \leftarrow v$
    $T \leftarrow T \cup \{(v, prev(v))\}$
    $R \leftarrow R \cup \{v\}$
```

By examining this pseudocode we get the runtime of this algorithm in terms of heap operations as

- n Insert(x, k) operations
- n ExtractMin() operations
- For each vertex v, $deg(v)$ DecreaseKey(x, k) operations. A given edge (v, u) is considered only once (when u or v is added to R first). So in total the number of DecreaseKey(x, k) operations are $\sum\limits_{v \in V} deg(v) = O(E)$.

Hence in total the time complexity of Prim's algorithm is $\underbrace{n \log n}_{insert/ExtractMin} + \underbrace{m \log n}_{decreasekey} = O(m \log n)$