Minimum Spanning Trees

Michiel Smid*

May 15, 2006

1 Introduction

We are given a connected undirected graph $G = (V, E)$ with vertex set V and edge set E. Each edge $(u, v) \in E$ has a weight $wt(u, v)$ which is a positive real number. We denote the number of vertices by n and the number of edges by m. We want to compute a subset T of E such that

1. the graph (V, T) is connected, and
2. the weight of T, which is $\sum_{(u,v)\in T} wt(u, v)$, is minimum.

Observe that in order to satisfy 1. and 2., the graph (V, T) must be a tree[1]. Therefore, it is called a *minimum spanning tree* of G.

In these notes, we will present two algorithms to compute a minimum spanning tree of G. Both algorithms are based on the following lemma.

Lemma 1 *Let $V = V_1 \cup V_2$ be a partition of the vertex set V. (Hence, V_1 and V_2 are both non-empty and disjoint.) Let (u, v) be an edge of minimum weight that connects V_1 and V_2, i.e., $u \in V_1$, $v \in V_2$, $(u, v) \in E$, and*

$$wt(u, v) = \min\{wt(x, y) : x \in V_1, y \in V_2, (x, y) \in E\}.$$

Then there is a minimum spanning tree of G that contains the edge (u, v).

*School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6. E-mail: michiel@scs.carleton.ca.

[1]recall that a graph is a tree if and only if it is connected and does not contain any cycle.

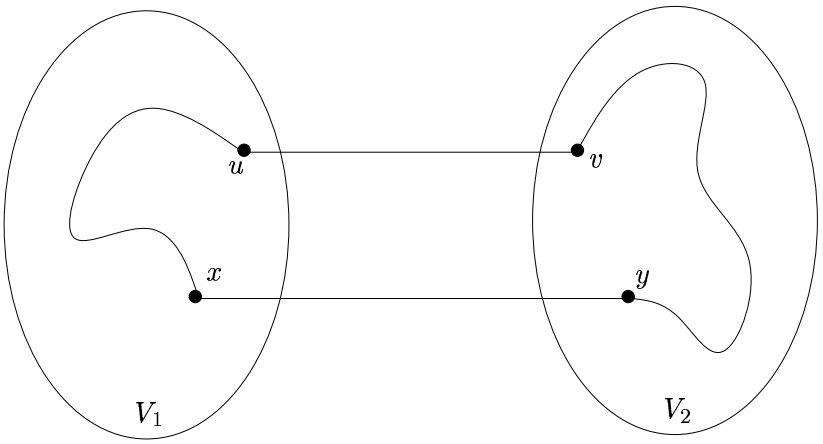

Figure 1: *Illustrating the proof of Lemma 1.*

Proof. Let MST be an arbitrary minimum spanning tree of G. If this tree contains the edge (u, v), then we are done. So assume that MST does not contain (u, v). We will construct another minimum spanning tree MST' that contains the edge (u, v).

Consider the graph G' obtained by adding the edge (u, v) to MST. (Refer to Figure 1.) In this graph, there is a cycle that contains (u, v). Since (u, v) is an edge between the sets V_1 and V_2, there must be another edge, say (x, y), on this cycle such that $x \in V_1$ and $y \in V_2$. Note that (x, y) is an edge of MST. It follows from our choice of the edge (u, v) that

$$wt(u, v) \leq wt(x, y).$$

Let MST' be the graph obtained from MST by replacing edge (x, y) by (u, v), i.e.,

$$MST' := (MST \setminus \{(x, y)\}) \cup \{(u, v)\}.$$

Then, MST' is a spanning tree of G, and

$$wt(MST') = wt(MST) - wt(x, y) + wt(u, v) \leq wt(MST).$$

On the other hand, since MST is a minimum spanning tree, we have

$$wt(MST) \leq wt(MST').$$

It follows that

$$wt(MST') = wt(MST),$$

i.e., MST' is also a minimum spanning tree of G. Hence, we have shown that there is a minimum spanning tree of G that contains the edge (u, v). ∎

2 Kruskal's algorithm

Our first minimum spanning tree algorithm is due to Kruskal (1956). This algorithm does the following. It maintains a *forest*, which is a collection of trees. In each step, an edge of minimum weight is added that does not create a cycle. More precisely:

1. The algorithm starts with a forest consisting of n trees, each consisting of a single vertex of V.

2. The algorithm combines two trees in the forest, using an edge of minimum weight, and repeats this, until the forest consists of one single tree. As we will see, this final tree is a minimum spanning tree of G.

The algorithm is given in Figure 2. It takes as input the connected undirected weighted graph $G = (V, E)$, and gives as output a minimum spanning tree of G. The graph G has n vertices and m edges. The vertices are denoted as $V = \{x_1, x_2, \ldots, x_n\}$.

We now prove that this algorithm indeed computes a minimum spanning tree of the graph $G = (V, E)$. The following lemma follows immediately from the algorithm.

Lemma 2 *Let k be an index with $1 \leq k \leq m$, and consider the k-th iteration of Kruskal's algorithm. At any moment after this iteration, the vertices u_k and v_k are connected by a path in the graph (V, T), where $T := \bigcup_\ell T_\ell$.*

Lemma 3 *During the main for-loop of Kruskal's algorithm, the following invariant is maintained:*

1. *The non-empty sets V_ℓ form a partition of the vertex set V.*

2. *There is a minimum spanning tree of G that contains the edge set $T := \bigcup_\ell T_\ell$.*

Kruskal's Algorithm
sort the m edges of E in non-decreasing order of their weights;
let $e_1, e_2, \ldots, e_m$ be the sorted sequence, i.e.,
$wt(e_1) \leq wt(e_2) \leq \ldots \leq wt(e_m)$;
for $i := 1$ **to** n
do $V_i := \{x_i\}$;
    $T_i := \emptyset$
endfor;
for $k := 1$ **to** m
do ($*$ do we include edge e_k? $*$)
    let u_k and v_k be the vertices of e_k;
    $i :=$ index such that $u_k \in V_i$;
    $j :=$ index such that $v_k \in V_j$;
    if $i \neq j$
    then ($*$ adding edge (u_k, v_k) does not introduce a cycle $*$)
            $V_i := V_i \cup V_j$;
            $V_j := \emptyset$;
            $T_i := T_i \cup T_j \cup \{(u_k, v_k)\}$;
            $T_j := \emptyset$
    endif
endfor

Figure 2: *Kruskal's minimum spanning tree algorithm.*

---

3. *For all i and i' with $i \neq i'$, no vertex of V_i is connected by a path (in the graph (V, T)) to any vertex of $V_{i'}$.*

Proof. The first and third claims follow immediately from the algorithm. So it remains to prove the second claim. Immediately before the first iteration of the for-loop, the edge set T is empty. Hence, the second claim is true at that moment.

Let $1 \leq k \leq m$, and consider the k-th iteration of the for-loop. Assume the second claim holds at the beginning of this iteration. Consider the indices i and j such that $u_k \in V_i$ and $v_k \in V_j$. If $i = j$, then the edge set T does not change during this iteration. Hence, in this case the second claim holds immediately after the k-th iteration. So assume that $i \neq j$. We first show

that

$$wt(u_k, v_k) = \min\{wt(x, y) : x \in V_i, y \in V \setminus V_i, (x, y) \in E\}. \qquad (1)$$

To prove (1), assume that there is an edge $(x, y) \in E$ such that $x \in V_i$, $y \in V \setminus V_i$, and

$$wt(x, y) < wt(u_k, v_k).$$

Let i' be the index such that—at the beginning of the k-th iteration—$y \in V_{i'}$. Since $x \in V_i$ and $y \in V \setminus V_i$, we have $i \neq i'$.

Let a be the integer such that the edge (x, y) was tested in the a-th iteration of the for-loop. Since we test the pairs of points in non-decreasing order of their distances, we have $1 \leq a < k$. Hence, by Lemma 2, at the beginning of the k-th iteration, the vertices x and y are connected by a path in the graph (V, T). But then the third claim implies that at that moment, the vertices x and y are contained in the same subset, i.e., $i = i'$. This is a contradiction.

So we know that (1) holds. Let MST be a minimum spanning tree of G that contains the edges of T. (Here, T is the edge set at the beginning of the k-th iteration.) If (u_k, v_k) is an edge of MST, then the second claim is true immediately after the k-th iteration. (Note that (u_k, v_k) is added to T during this iteration.) Assume that (u_k, v_k) is not an edge of MST. We replace, in MST, any edge e that joins a vertex of V_i with a vertex of $V \setminus V_i$, by edge (u_k, v_k). (Note that e is not contained in T.) As in the proof of Lemma 1, it follows that this gives another minimum spanning tree of G that contains the edge set $T \cup \{(u_k, v_k)\}$. Hence, also in this case, the second claim holds immediately after the k-th iteration. ∎

Lemma 4 *Let $T := \bigcup_\ell T_\ell$ be the edge set that is computed by Kruskal's algorithm. The graph (V, T) is a minimum spanning tree of the input graph G.*

Proof. It follows from Lemma 2, that (V, T) is connected. By Lemma 3, there is a minimum spanning tree that contains this graph (V, T). Since the minimum spanning tree is a connected graph of minimum weight, it follows that (V, T) is a minimum spanning tree of G. ∎

How do we implement Kruskal's algorithm? Sorting the m edges of E takes $O(m \log m)$ time. Since $m \leq \binom{n}{2}$, the sorting step takes $O(m \log n)$ time. The first for-loop takes $O(n)$ time.

Let us consider the main for-loop. We maintain each set T_i of edges in a linked list. Then, each of the assignments $T_i := T_i \cup T_j \cup \{(u_k, v_k)\}$ and $T_j := \emptyset$ takes $O(1)$ time. Since each of these assignments is carried out exactly $n-1$ times during the main for-loop (why?), the total time for maintaining the edge sets is $O(n)$.

The main problem is to maintain the non-empty sets V_i. We need a data structure that stores these sets, and that supports the following operations:

- Initialization: $V_i := \{x_i\}$. This operation has to be processed for n different values of i.
- Given a vertex $x \in V$, find the index i such that $x \in V_i$. This operation has to be processed exactly $2m$ times.
- Given two distinct indices i and j, assign $V_i := V_i \cup V_j$ and $V_j := \emptyset$. This operation has to be processed exactly $n-1$ times.

In Section 2.1, we will give a data structure having the following properties:

- Initializing any set V_i can be done in $O(1)$ time. So overall, we need $O(n)$ time for the entire initialization.
- For an arbitrary vertex $x \in V$, we can find in $O(\log n)$ time, the index i such that $x \in V_i$. Hence, the overall time for all these operations is $O(m \log n)$.
- The union of any two distinct sets V_i and V_j can be computed in $O(1)$ time. Hence, for these operations, we need $O(n)$ total time.

It follows that the total time for maintaining the non-empty sets V_i is $O(m \log n)$. This will prove the following theorem.

Theorem 1 *Given a connected undirected weighted graph with n vertices and m edges, Kruskal's algorithm computes a minimum spanning tree of G in $O(m \log n)$ time.*

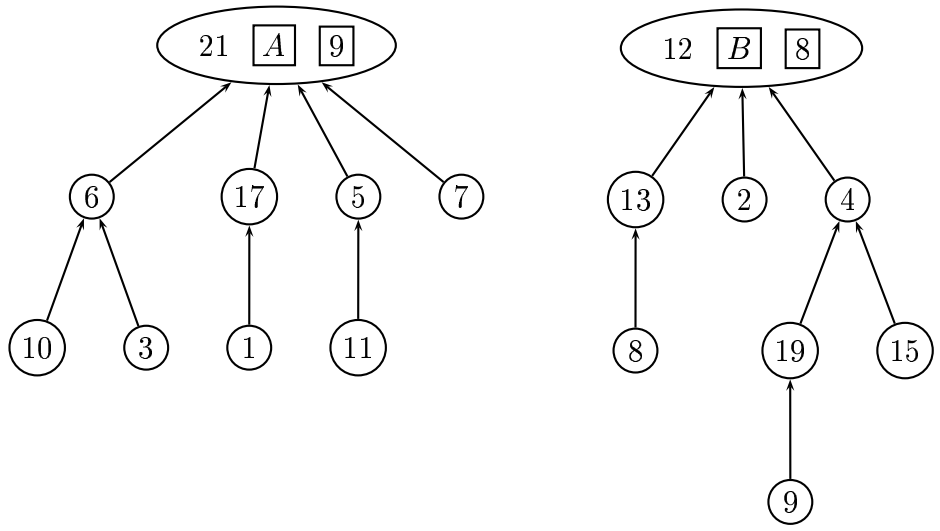

Figure 3: *Trees for the sets* $A = \{1, 3, 5, 6, 7, 10, 11, 17, 21\}$ *and* $B = \{2, 4, 8, 9, 12, 13, 15, 19\}$.

2.1 The Union-Find problem

We are given a collection of n disjoint sets $V_1, V_2, \ldots, V_n$, each containing a single element, and want to process a sequence of operations, where each operation is of one of the following two types:

- *Union*(A, B, C): Combine the two disjoint sets A and B into a new set named C. (Afterwards, the old sets A and B no longer exist.)
- *Find*(x): Compute the name of the (unique) set that contains x.

The data structure consists of a collection of trees. For each set A in the current collection of sets, there is one tree having $|A|$ nodes. Each node in this tree stores one element of A. Moreover, except for the root, each node contains a pointer to its parent. With the root, we store the name of the set and the number of its elements. See Figure 3.

Initialization: At the start of the sequence of operations, there are n trees. The i-th tree, $1 \leq i \leq n$, consists of one node that stores the only element of V_i, the name of this set and its size (which is one).

Initializing one tree clearly can be done in $O(1)$ time.

Union: To process the operation $Union(A, B, C)$, we are given pointers to the roots $r(A)$ and $r(B)$ of the trees that store the sets A and B, respectively. In these roots, we read the number of elements of A and B. We distinguish two cases.

1. If $|A| \leq |B|$, then we merge both trees by making $r(A)$ a child of $r(B)$: We give $r(A)$ a pointer to $r(B)$, and with $r(B)$, we store the name C of the new set and its size, which is $|A| + |B|$.

2. If $|B| < |A|$, then we merge both trees by making $r(B)$ a child of $r(A)$: We give $r(B)$ a pointer to $r(A)$, and with $r(A)$, we store the name C of the new set and its size, which is $|A| + |B|$.

See Figure 4 for an example. It is clear that one *Union* operation can be processed in $O(1)$ time.

Find: To process the operation $Find(x)$, we are given a pointer to the node u containing element x. Starting in this node u, we follow parent-pointers until we reach the root r of u's tree. In r, we read the name of the set that contains x.

The time for this *Find* operation is bounded by the *height* of the tree, which is the number of edges on a longest path from any leaf to the root. The following lemma implies an upper bound on the height of any tree in our data structure.

Lemma 5 *At any moment during the sequence of Union and Find operations, and for each set A, we have*

$$|A| \geq 2^{h(T_A)},$$

where T_A is the tree that stores the elements of A, and $h(T_A)$ is the height of this tree.

Proof. After the initialization, each set A has size one. The tree T_A storing A consists of a single node. Hence, $|A| = 1$ and $h(T_A) = 0$, which implies that $|A| \geq 2^{h(T_A)}$.

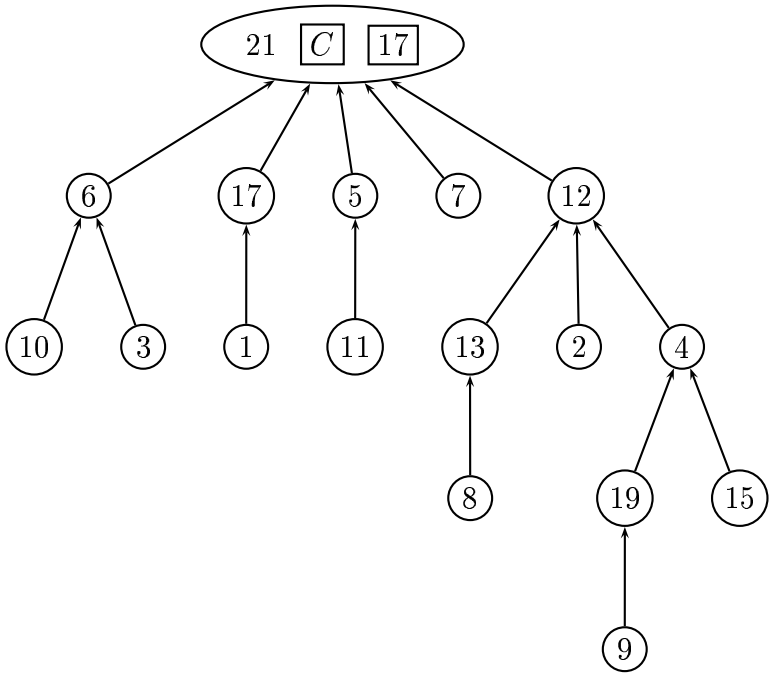

Figure 4: *The result of Union*(A, B, C) *on the two trees of Figure 3.*

Consider an operation *Union*(A, B, C), and assume that $|A| \geq 2^{h(T_A)}$ and $|B| \geq 2^{h(T_B)}$. We will show that $|C| \geq 2^{h(T_C)}$ after this operation. We may assume without loss of generality that $|A| \leq |B|$. Note that

$$h(T_C) = \max(1 + h(T_A), h(T_B)).$$

We distinguish two cases.

Case 1: $h(T_C) = h(T_B)$. Then

$$|C| = |A| + |B| \geq |B| \geq 2^{h(T_B)} = 2^{h(T_C)}.$$

Case 2: $h(T_C) = 1 + h(T_A)$. In this case, we have

$$|C| = |A| + |B| \geq 2 \cdot |A| \geq 2 \cdot 2^{h(T_A)} = 2^{1+h(T_A)} = 2^{h(T_C)}.$$

■

Consider any set A in our collection of disjoint sets. Then, by Lemma 5, $|A| \geq 2^{h(T_A)}$. Since $|A| \leq n$, it follows that

$$h(T_A) \leq \log |A| \leq \log n.$$

This proves that each *Find* operation can be processed in $O(\log n)$ time.

We have proved the following result:

Theorem 2 *For the Union-Find problem on n elements, there is a data structure such that*

1. *initializing any singleton set takes $O(1)$ time,*
2. *any Find operation can be processed in $O(\log n)$ time, and*
3. *any Union operation can be processed in $O(1)$ time.*

3 Prim's algorithm

We have seen that Kruskal's algorithm can be implemented such that its running time is $O(m \log n)$. In this section, we will give another algorithm having the same running time. This algorithm is commonly known as Prim's algorithm. It was discovered independently by Jarník (1930), Prim (1957), and Dijkstra (1959).

Consider again the connected undirected weighted graph $G = (V, E)$. Prim's algorithm does the following. It starts with a set A consisting of an arbitrary vertex of V, and an empty set T of edges. In each step, an edge of minimum weight joining a vertex of A with a vertex of $V \setminus A$ is added to T. In this step, the vertex of this edge that is in $V \setminus A$ "moves" to the set A. The algorithm terminates as soon as $A = V$. As we will see, at that moment, the graph (V, T) is a minimum spanning tree of G. A high-level description of this algorithm is given in Figure 5.

Lemma 6 *During the while-loop of Prim's algorithm, the following invariant is maintained:*

- *Each edge of T connects two vertices of A.*
- *There is a minimum spanning tree of G that contains the edge set T.*

Prim's Algorithm
$r :=$ arbitrary vertex of V;
$A := \{r\}$;
$T := \emptyset$;
while $A \neq V$
do find an edge $(u, v) \in E$ of minimum weight such that
    $u \in A$ and $v \in V \setminus A$;
    $A := A \cup \{v\}$;
    $T := T \cup \{(u, v)\}$
endwhile

Figure 5: *A high-level description of Prim's minimum spanning tree algorithm.*

Proof. It is clear that the first part of the invariant holds. So let us prove that the second part also holds.

Immediately before the while-loop starts, the set T is empty. Hence, the second part of the invariant holds at that moment. Consider one iteration of the while-loop, and assume that the invariant holds at the beginning of it. Consider the sets A and T at the beginning of this iteration. Let MST be a minimum spanning tree of G that contains all edges of T. Consider the edge (u, v) that is added to T during this iteration. If (u, v) is an edge of MST, then the invariant holds at the end of this iteration, i.e., for the set $T \cup \{(u, v)\}$.

Otherwise, if (u, v) is not an edge of MST, we replace in MST any edge e that joins a vertex of A with a vertex of $V \setminus A$, by edge (u, v). (Note that, by the first part of the invariant, e is not contained in T.) It follows in exactly the same way as in the proof of Lemma 1, that this results in another minimum spanning tree MST' of G that contains the edge set $T \cup \{(u, v)\}$. Hence, also in this case the invariant holds at the end of this iteration. ∎

Lemma 7 *Let T be the edge set that is computed by Prim's algorithm. The graph (V, T) is a minimum spanning tree of the input graph G.*

Proof. We make the following observations:

1. Initially, the set T is empty.

2. The while-loop makes exactly $n-1$ iterations.

3. In each iteration, one edge is added to T.

Hence, at the end of the algorithm, the set T contains exactly $n-1$ edges. By Lemma 6, there is a minimum spanning tree of G that contains this edge set T. Since a minimum spanning tree has $n-1$ edges, it follows that the graph (V, T) is a minimum spanning tree of G. ∎

How do we implement Prim's algorithm? The main problem is to find an edge in E of minimum weight that joins a vertex of A and a vertex of $V \setminus A$. Computing this edge by brute-force leads to a total running time of $\Theta(nm)$. (Why?)

In order to speed up the algorithm, we maintain the following information.

- For each vertex $y \in V \setminus A$,
  - a variable $\mathit{minweight}(y)$, whose value is

    $$\mathit{minweight}(y) = \min\{wt(x, y) : x \in A, (x, y) \in E\}.$$

    (In words, $\mathit{minweight}(y)$ is the minimum weight of any edge between y and a vertex of A.)
  - a variable $\mathit{closest}(y)$, which is a vertex $x \in A$ for which $(x, y) \in E$ and $wt(x, y) = \mathit{minweight}(y)$. (In words, $\mathit{closest}(y)$ is the vertex of A that is part of the edge of minimum weight between y and any vertex in A.)

Observation 1 *We have*

$$\min\{\mathit{minweight}(y) : y \in V \setminus A\} = \min\{wt(x, y) : x \in A, y \in V \setminus A, (x, y) \in E\}.$$

A version of Prim's algorithm that uses these variables is given in Figure 6. The value of the variable k is equal to the number of elements in the set A, whereas the set Q is equal to $V \setminus A$.

We maintain the set T in a list. Also, with each vertex, we store a bit indicating whether it belongs to A or to Q.

How do we find the vertex v of Q for which $\mathit{minweight}(v)$ is minimum? Here is the answer: We maintain the elements of Q in a *min-heap* with the keys being the *minweight*-values.

Prim's Algorithm
$r :=$ arbitrary vertex of V;
$A := \{r\}$;
$T := \emptyset$;
for each $y \in V \setminus \{r\}$
do $minweight(y) := \infty$;
    $closest(y) := nil$
endfor;
for each edge $(r, y) \in E$
do $minweight(y) := wt(r, y)$;
    $closest(y) := r$
endfor;
$Q := V \setminus \{r\}$;
$k := 1$;
while $k \neq n$
do $v :=$ vertex of Q for which $minweight(v)$ is minimum;
    $u := closest(v)$;
    $A := A \cup \{v\}$;
    $Q := Q \setminus \{v\}$;
    $T := T \cup \{(u, v)\}$;
    $k := k + 1$;
    for each edge $(v, y) \in E$
    do if $y \in Q$ and $wt(v, y) < minweight(y)$
        then $minweight(y) := wt(v, y)$;
            $closest(y) := v$
        endif
    endfor
endwhile

Figure 6: *An efficient version of Prim's minimum spanning tree algorithm.*

Immediately before the while-loop, we build the heap for the elements of $Q = V \setminus \{r\}$. This takes $O(n)$ time. So the part of the algorithm up to the while-loop takes $O(n)$ time.

Consider one iteration of the while-loop. Finding the vertex $v \in Q$ whose *minweight*-value is minimum, and deleting v from Q is an *extract_min* oper-

ation in the heap; it takes $O(\log n)$ time. In the for-loop, we consider each edge $(v, y) \in E$. If $y \in Q$ and $wt(v, y) < minweight(y)$, then we decrease $minweight(y)$ to $wt(v, y)$; in the heap, this corresponds to a *decrease_key* operation which takes $O(\log n)$ time. Therefore, the time for one iteration of the while-loop is

$$O(\log n + deg(v) \cdot \log n) = O(deg(v) \cdot \log n),$$

where $deg(v)$ is the degree of vertex v.

It follows that the entire while-loop takes time

$$O\left(\log n \sum_{v \in V} deg(v)\right).$$

Since $\sum_{v \in V} deg(v) = 2m$, the while-loop takes $O(m \log n)$ time. We have shown that the entire algorithm takes $O(n + m \log n) = O(m \log n)$ time.

Theorem 3 *Given a connected undirected weighted graph with n vertices and m edges, Prim's algorithm computes a minimum spanning tree of G in $O(m \log n)$ time.*