Proving the Correctness of Prim's Algorithm for Computing Minimum Spanning Trees

Siddharth Mahendraker
siddharth.mahen@gmail.com

February 3, 2014

Contents

1 Motivation

Consider the following scenario. A group of 11th grade IB students are big basketball fans. Today, their favourite team is playing their arch rivals. The students watch the game during lunch, but unfortunately, the end of the game overlaps with their next class.

Itching to know how the game ends, the students decide to designate one student as a broadcaster, who will discretely read the game's twitter feed from his phone and tell everyone else what's going on. The students decide this broadcaster will tell his neighbours the news, and they will in turn tell their neighbours and so forth, until everyone in the class has been updated. The students agree this method will arouse the least amount of suspicion from their strict teacher, who already separates their desks to keep talking to a minimum.

At first, their strategy works well, however, they notice that some of their methods of passing messages are less risky than others. Particularly, they notice that it is sometimes less risky for a student to update only a few of his neighbours directly, and let the information reach his other neighbours indirectly via other people.

Interested in repeating this strategy for other basketball games in other classes, the students ask themselves: How can we pass messages around the classroom such that our total risk is minimized, but everyone receives updates about the game?

2 Minimum Spanning Trees

The students' dilemma can be modelled using graph theory. Recall that a graph is a mathematical object which captures information about the relationships between discrete objects.

Definition 2.1. A *graph* is an ordered pair $G = (V, E)$, where V is a finite set of nodes, or vertices, and $E \subseteq V \times V$ is a set of ordered pairs of nodes, representing edges.

We denote the vertex set and the edge set of a graph G by $V(G)$ and $E(G)$ respectively. For convenience, we also denote an edge (u, v) in $E(G)$ by uv.

Let us now introduce some basic graph theoretic vocabulary. We say a graph $G = (V, E, w)$ is *weighted* if it is equipped with a function $w : E \to \mathbb{R}^+$, which associates each edge to a positive real number. Further, we say a graph G is *simple* if, for every element (u, v) in $E(G)$, the element (v, u) is also in $E(G)$, and for all (u, v) in $E(G)$, u is different from v. Informally, this means that the graph is undirected, and does not contain loops. We say a graph G is *connected* if there exists a sequence of adjacent edges, i.e. a path,

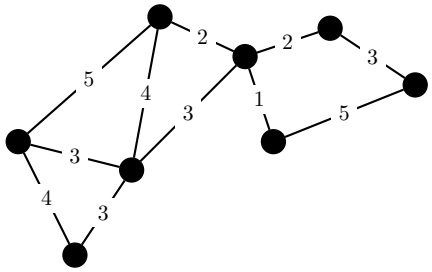

Figure 1: A graph representing a possible arrangement of students in the classroom. Vertices represent students and edges represent opportunities to pass messages, where the weight of an edge represents the risk that the conversation across the edge is compromised.

between every vertex u and v in $V(G)$. Finally, we say a graph is *cyclic* if it contains a non-empty path which starts and ends at the same vertex. A graph which is not cyclic is called *acyclic*.

For the remainder of this paper, all graphs are assumed to be simple.

Terminology aside, let us return to the students' dilemma. As we mentioned earlier, we can model their problem using graph theory. Let $G = (V, E, w)$ be a connected, weighted graph whose vertices correspond to students and whose edges correspond to opportunities to pass messages, where the weight $w(uv)$ of an edge $uv \in E$ corresponds to the risk that the conversation between the students u, v is compromised (see Figure 1). Call G the students' classroom graph.

An ideal method of passing messages would require that the students find a set of opportunities to pass messages, $E' \subset E(G)$, such that the total risk the students run is as low as possible, and every student receives updates about the game. This corresponds to some connected subgraph $T = (V, E', w)$ of the classroom graph G, where $E' \subset E(G)$ and $V(T) = V(G)$.

Immediately, we see that any connected cyclic subgraph T of the classroom graph can never be considered a low risk method of passing messages. This is a consequence of trying to minimize the total risk the students run. Suppose the students come up with some connected cyclic subgraph $T = (V, E', w)$ which they want to use to pass messages. We can create a new subgraph T^* of T by removing an edge from each of the cycles in T. This new subgraph has a lower risk of getting the students caught as it contains strictly fewer edges than T. Moreover, because removing single edges from cycles preserves the connectedness in T, this subgraph can still be used to pass messages to everyone. Therefore, since we can always construct a lower risk subgraph given a cyclic subgraph, we won't bother considering cyclic subgraphs to pass messages.

Definition 2.2. Let G be a connected graph. Then a *spanning tree* T of G is defined as a connected acyclic subgraph $T = (V, E')$, where $E' \subset E(G)$

and $V(T) = V(G)$.

Reformulating the students' problem, we say the students are looking for a spanning tree T of the classroom graph G where the sum of the weights of the edges in the spanning tree are as low as possible (or at least as low as that of any other spanning tree of G).

Definition 2.3. Let $T = (V, E', w)$ be a spanning tree of a weighted graph $G = (V, E, w)$. Define the cost of the spanning tree, $c(T)$ as the sum of the edge weights in E',

$$c(T) = \sum_{e \in E'} w(e).$$

If a spanning tree T of a graph G has a cost at least as low as the cost of every other possible spanning tree of G, we call T a minimum spanning tree of G.

Definition 2.4. Let T^* be a spanning tree of a graph G. We call T^* a *minimum spanning tree* (MST) if

$$c(T^*) \leq c(T)$$

for all alternate spanning trees T of G.

Given a minimum spanning tree, it is evident the students could achieve their goal of communicating updates to everyone while minimizing the total risk they run. The broadcaster would simply send his messages to students adjacent to him in the minimum spanning tree, and these students would simply propagate their messages to their neighbours on the spanning tree and so forth. Because the minimum spanning tree represents a way of propagating messages with the lowest total risk, and every student is part of the minimum spanning tree, it constitutes an optimal solution.

However, there still remains the problem of actually computing a minimum spanning tree for the classroom graph. Assuming the students have an accurate idea of the risk involved in speaking with any of their neighbours, how can the students compute a minimum spanning tree for their classroom graph? Furthermore, how can they be sure the spanning tree they compute is truly a *minimum* spanning tree?

3 Prim's Algorithm for Computing MSTs

The following algorithm, due to Robert C. Prim, can be used to compute the minimum spanning tree T^* of a weighted graph G [1, pp. 634–636].

Definition 3.1 (Prim's Algorithm). Let $G = (V, E, w)$ be a weighted graph. Then a minimum spanning tree T^* of G can be computed as follows:

1. Begin with an empty set $T = \emptyset$ and a set $X = \{v\}$, where $v \in V$ is arbitrarily selected.

2. Until $X = V$ repeat the step below.

3. Let Y be the set of edges with one vertex in X and the other vertex in $E - X$. Choose the edge $e \in Y$ with the smallest weight $w(e)$, breaking ties arbitrarily. Add e to T and add the vertex at the other end of e to X.

4. Output a minimum spanning tree $T^* = (V, T, w)$.

See Figure 2 for an example of how Prim's algorithm works.

Intuitively, this algorithm corresponds to the following strategy for the students:

1. Begin with an arbitrarily selected broadcaster.

2. Have the broadcaster pick a neighbour who, when spoken to, is least likely to compromise the pair. Pass messages to this neighbour.

3. Now, with respect to the current group of people who know the news, pick a neighbour who, when spoken to, is least likely to compromise the group.

4. Repeat the previous step until everyone in the class knows the latest information.

Apart from their utility in colluding against teachers, minimum spanning trees have a myriad of other important applications. They provide a solution to a vast array of real world optimization problems, in fields as disparate as computer networking, machine learning and infrastructure construction. From the fairly trivial scenario used above, we see that computing a minimum spanning tree is an important part of broadcasting information efficiently among a set of interconnected parties. Therefore, it is important to know whether algorithms such as Prim's algorithm really do work correctly to output minimum spanning trees.

My personal interest in analyzing minimum spanning tree algorithms grew out of my interest in their asymptotic running time analyses. Asymptotic analysis gives strong bounds on the number of operations a computer will need to perform when running an algorithm. I am particularly interested in the analysis of randomized algorithms which compute minimum spanning trees of arbitrary graphs very efficiently [3]. Moreover minimum spanning tree algorithms have introduced me to key concepts in graph theory and combinatorics, which I have thoroughly enjoyed studying.

The remainder of this paper is devoted to proving the correctness of Prim's algorithm.

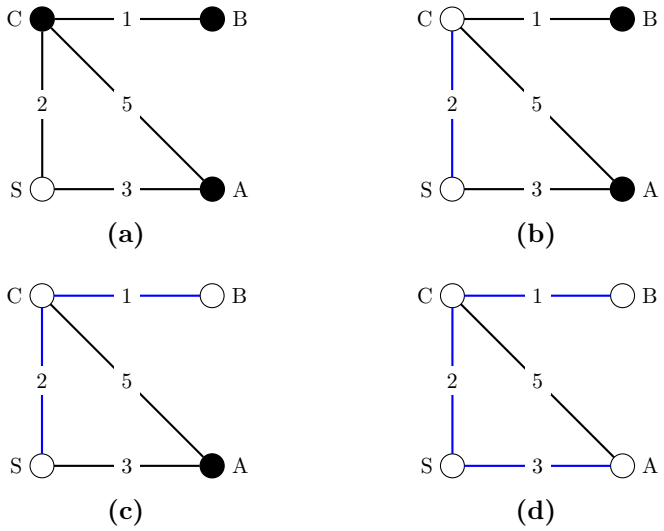

Figure 2: The execution of Prim's algorithm on a weighted graph G with 4 vertices. White vertices represent elements of X, while black vertices represent elements of $V - X$. Blue edges represent edges added to the output spanning tree T^*. In step (a), the inital element $S \in X$ is chosen arbitrarily. We see that in step (d), Prim's algorithm halts as $X = V$ and T^* is indeed a minimum spanning tree of G.

The proof of correctness will proceed in three parts. First, we show that the algorithm terminates. Then we show that it outputs a spanning tree. Finally, we show that the output spanning tree is minimal.

4 Connectedness and the Cut Property

Before we begin, let us establish an important property of connected graphs, called the cut property. This property will allow us to more easily navigate the proofs ahead.

Definition 4.1. Let G be a connected graph. A *cut* (A, B) of G is a partition of the vertex set $V(G)$ into two sets A and B, such that $A \cup B = V(G)$ and $A \cap B = \emptyset$.

If (A, B) is a cut of G, and $e \in E(G)$ is an edge with one endpoint in A and the other in B, we say the edge e *crosses the cut* (A, B).

Definition 4.2. Let G be a connected graph, and let A be a subset of $V(G)$. Then an *induced subgraph* $G[A]$ is a subgraph of G whose vertex set is A and whose edge set is all edges in $E(G)$ with both endpoints in A.

The motivation behind the cut property is that it allows us to clearly link what Prim's algorithm does in every step to the connectedness of its

input graph. In every step, the algorithm adds an edge e to T with one endpoint in X and another endpoint in $V - X$. If we can show that there always exists an edge crossing the cut $(X, V - X)$, then we can easily deduce many other properties of the algorithm.

The notion of an induced subgraph is required to formally prove the cut property.

Lemma 4.3 (The Cut Property). *Let $G = (V, E, w)$ be a weighted graph. Then G is connected if and only if for any cut (A, B) of G, there exists at least one edge which crosses the cut.*

Proof. Assume G is connected. Then there exists a path between every vertex u and v in $V(G)$. Let (A, B) be a cut of G, and suppose $a \in A$ and $b \in B$ are two vertices on different sides of the cut. Then by the connectedness of G, there exists a sequence of edges between a and b. Because a and b are on different sides of the cut, there must be some edge $uv \in E(G)$ with one endpoint u in A and the other endpoint v in B. Hence, there exists an edge which crosses the cut (A, B). Since the cut (A, B) was chosen arbitrarily, we have shown connectedness implies the cut property.

Now, assume that for all cuts (A, B) of G, there exists at least one edge $e \in E(G)$ which crosses the cut (A, B). First, we show that for any choice of A and B, the induced subgraphs $G[A]$ and $G[B]$ are connected.

Suppose $G[A]$ is not connected. Then there exist vertices $x_1, x_2, \ldots, x_k$ in $G[A]$ which cannot be connected by a path. Let $X \subset A$ denote the set of vertices which can be reached from x_1 by a path. Assume, without loss of generality, that $G[B]$ is connected, and the edge across (A, B) has an endpoint in X. Then there do not exist any edges which cross the cut $(X \cup B, A - (X \cup B))$ in G, a contradiction. Hence, $G[A]$ must be connected. Similarly, $G[B]$ must be connected.

By our assumption, there exists at least one edge $e \in E(G)$ across the cut (A, B). Since, $G[A]$ and $G[B]$ are connected, and the edge e joins these two partitions of G, the entire graph G is connected. Therefore, we've shown the cut property implies connectedness. □

5 Proof of Correctness of Prim's Algorithm

Using the cut property, proving the correctness of Prim's algorithm becomes much simpler.

Recall that proving the correctness of Prim's algorithm requires proving the algorithm terminates, that it outputs a spanning tree of the input graph, and that this spanning tree is minimal.

Theorem 5.1. *Let $G = (V, E, w)$ be a connected, weighted graph. Then the execution of Prim's algorithm for the input G eventually terminates.*

Proof. In each step of Prim's algorithm, an edge uv is added to T, with one endpoint $u \in X$ and another endpoint $v \in V - X$. The endpoint v is then added to X. By the cut property, we know that such an edge exists for all cuts $(X, V - X)$. Therefore, after every step of the algorithm, an endpoint $v \in V - X$ of an edge uv is added to X, and hence the cardinality of X increases by 1. Since the cardinality of X begins at 1, and the cardinality of V is finite, there must exist some step after which $X = V$. Hence, Prim's algorithm must terminate. □

Theorem 5.2. *Let $G = (V, E, w)$ be a connected, weighted graph. Then the output $T^* = (V, T, w)$ of the execution of Prim's algorithm for the input G constitutes a spanning tree of G.*

Proof. First, we show that the output of Prim's algorithm T^* is always connected, using induction on the number of steps of the algorithm.

In the base case, let $X = \{v\}$. By the cut property, we know there exists an edge $vw \in E(G)$ across the cut $(X, V - X)$. Adding the vertex w to X and the edge vw to T clearly leaves T^* connected as there is a path between all the vertices in T^*. Thus, T^* is connected after the first step.

Now assume T^* is connected at after some step k. We show that T^* is still connected after step $k + 1$. By the cut property, we know that there exists an edge uv crossing the cut $(X, V - X)$, with endpoint $v \in V - X$. Assuming T^* is connected, adding the vertex v to X and the edge uv to T leaves T^* connected at the end of step $k + 1$. By induction T^* is always connected.

Next, we show that T^* never contains any cycles. In any step of the algorithm, the only way to create a cycle in T^* is to add an edge uv to T with both endpoints u and v in X. This would create a cycle as T^* is connected. By definition, however, the algorithm only adds edges to T with one endpoint in X and the other in $V - X$. Therefore, $T^* = (X, T, w)$ is acylic after every step of the algorithm.

Since Prim's algorithm terminates when $X = V$, and the output $T^* = (X, T, w)$ is always connected and acyclic, the output of Prim's algorithm constitutes a spanning tree of the original graph G. □

Theorem 5.3. *Let $G = (V, E, w)$ be a connected, weighted graph. Then the output $T^* = (V, T, w)$ of the execution of Prim's algorithm for the input G constitutes a minimum spanning tree of G.*

Proof. Let $S^* = (V, S, w)$ be a minimum spanning tree of G. From the previous theorem, we know that T^* is a spanning tree of G. We show that the weight of every edge in T^* is at least as small as the weight of every edge in S^*, and thus that T^* is also a minimum spanning tree of G.

Consider the execution of Prim's algorithm in some step k. Let e_k be the edge across the cut $(X, V - X)$ which was added to T, and let $e^* \in S$ be the

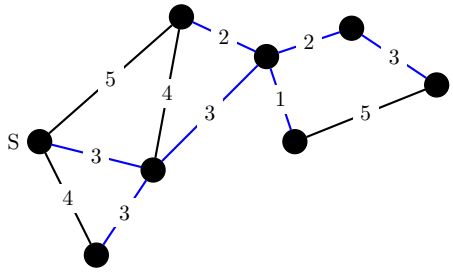

Figure 3: The output of Prim's algorithm given the example classroom graph from Figure 1 as input. Blue edges represent edges in the output spanning tree. The initial vertex S was chosen arbitrarily. It is not difficult to see that the output spanning tree would be the same had the algorithm selected any other initial vertex.

edge across $(X, V - X)$ in S^*. We know S^* must have an edge across the cut $(X, V - X)$ as S^* is connected. Now consider the weights of the edges e_k and e^*. By definition, Prim's algorithm chooses an edge across the cut $(X, V - X)$ with the lowest weight, and thus $w(e_k) \leq w(e^*)$. Since this holds in any arbitrary step k, we conclude that

$$\sum_{e \in T} w(e) = c(T^*) \leq c(S^*) = \sum_{e \in S} w(e).$$

If $c(T^*) < c(S^*)$, then S^* is not a minimum spanning tree, a contradiction. Thus $c(T^*) = c(S^*)$. Therefore, the output of Prim's algorithm, T^*, constitutes a minimum spanning tree of the input graph G. □

Note that no step in the three precedent proofs was dependent on the choice of the initial vertex, and thus this element can be selected arbitrarily without consequence.

6 Conclusion

We conclude that Prim's algorithm is indeed correct; given a connected weighted graph G as input, it outputs a truly *minimum* spanning tree of G. Great! This means the students can always compute a lowest risk method of passing messages to one another in their classroom graph (see Figure 3).

Although we have shown that Prim's algorithm is correct, there are many other aspects of minimum spanning tree computation that we could potentially explore. For example, if we change the weights of m edges in the input graph, how could we recompute an MST of the input graph efficiently? This question comes up very often when MSTs are used in relation to communication networks, where the strengths of communications

channels can vary over time. See [2, §8] for a thorough solution to this problem.

Moving further away from practical applications of MST computation, we could also ask whether MSTs and algorithms such as Prim's algorithm provide good approximate solutions to other mathematical problems. For example, consider using MST computation to approximate solutions to the Travelling Salesman Problem [1, pp. 1096–1097]. We could also ask whether Prim's algorithm could be modified to handle negative edge weights.

Prim's algorithm and MSTs are a rich and interesting area of mathematics with important applications to the real world. See the references below for a very thorough overview other MST computation techniques, as well as more information about MSTs in general.

All of the MSTs in this paper were found using an original implementation of Prim's algorithm, written in Python. See the Appendix for more information.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 55 Hayward Street, Cambridge, MA 02142-1315 USA, third edition, 2009.

[2] Jason Eisner. State-of-the-Art Algorithms for Minimum Spanning Trees. Technical report, University of Pennsylvania, 1997.

[3] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *Journal for the Association of Computing Machinery*, 2:321–328, 1995.

Appendix

An Implementation of Prim's Algorithm in Python

The following program can be run using Python 2.7.X. The program takes as input a text file describing a graph, and outputs a list of edges which constitute the minimum spanning tree of the graph described.

The input file must be formatted as follows. The first line must contain the number of vertices and the number of edges in the graph. For example the line `4 2` indicates the graph has 4 vertices and 2 edges. All subsequent lines must be integer triples, each describing a weighted edge of the graph. The first two integers describe the endpoints of the edge, and the final integer its weight. For example, the line `0 3 5` indicates the graph has an edge with endpoints at verticies 0 and 3, and this edge has weight 5.

To run this code with a given input file, execute `python prim.py graph.txt` in your favourite command line interface, where `prim.py` contains the code below and `graph.txt` describes the input graph as defined above.

See Listing 1 and Listing 2 for more examples.

```
from sys import argv
import re

# open the file and get read to read data
file = open(argv[1], "r");
p = re.compile("\d+");

# initialize the graph
vertices, edges = map(int, p.findall(file.readline()))
graph = [[0]*vertices for _ in range(vertices)]

# populate the graph
for i in range(edges):
    u, v, weight = map(int, p.findall(file.readline()))
    graph[u][v] = weight
    graph[v][u] = weight

# initialize the MST and the set X
T = set();
X = set();

# select an arbitrary vertex to begin with
X.add(0);

```

```
25 while len(X) != vertices:
26     crossing = set();
27     # for each element x in X, add the edge (x, k) to crossing if
28     # k is not in X
29     for x in X:
30         for k in range(vertices):
31             if k not in X and graph[x][k] != 0:
32                 crossing.add((x, k))
33     # find the edge with the smallest weight in crossing
34     edge = sorted(crossing, key=lambda e:graph[e[0]][e[1]])[0];
35     # add this edge to T
36     T.add(edge)
37     # add the new vertex to X
38     X.add(edge[1])
39 
40 # print the edges of the MST
41 for edge in T:
42     print edge
43 
44 # NOTE: This is a highly inefficient implementation of this
45 #       algorithm. However, I have chosen to use this
46 #       implementation because it most closely matches the
47 #       algorithm given in Definition 3.1.
48 #
49 #       A much more efficient implementation can be obtained
50 #       by using slightly more advanced data structures to
51 #       represent both the graph and the edges. For more
52 #       information see [1, pp. 634 -- 636].
```

Input Graphs for Examples

The following text files were input into the implementation of Prim's algorithm above to generate the MSTs in Figure 2 and Figure 3, respectively.

```
4 4
0 1 3
0 3 2
1 3 5
2 3 1
```

Listing 1: This file was used to generate the MST in Figure 2. Here, 0 corresponds to the initial vertex S. Vertices A through C correspond to numbers 1 through 3, respectively.

```
8 11
0 1 4
0 2 3
0 3 5
1 2 3
2 3 4
2 4 3
3 4 2
4 5 2
4 6 1
5 7 3
6 7 5
```

Listing 2: This file was used to generate the MST in Figure 3. Again, 0 corresponds to the initial vertex S.