Oliver Tonnesen
V00885732

Project 1

MATH 423
September 26, 2019

1 Introduction

1.1 Background

The minimum-weight spanning tree problem is one of the oldest studied problems in graph theory. Given a connected, edge-weighted graph, a minimum-weight spanning tree is a subgraph that is connected, acyclic, and whose edge weights sum to the minimum possible value.

The first published solution to the minimum-weight spanning tree problem is due to Borůvka in 1926 [**?**]. In his paper, Borůvka devotes a section to the application of his method to the design of electricity power networks. The example he uses is his home region of Moravia.

The single-source shortest path problem is a more recent problem. Given a connected, edge-weighted graph, and a vertex s in it, a correct solution is a subgraph that is connected, acyclic, and in which any path from s to any other vertex has minimal weight.

This paper aims to briefly survey some of the classical methods of solving the above problems. Specifically, we investigate Kruskal and Prim's algorithms, two methods of solving the minimum-weight spanning tree problem, and Dijkstra's algorithm, a method of solving a special case of the single-source shortest path problem where the given graph has only nonnegative edge weights.

1.2 Applications

There are many useful applications of minimum-weight spanning trees. Some of their most obvious applications are those for which they were originally studied: physical networks, such as electrical networks, transportation systems, water networks, computer and telecommunications networks, among others. Some less obvious applications include optimally broadcasting information throughout all computers in a connected network, circuit design, or finding paths in a system with maximum bottleneck.

Some applications of solving the single-source shortest path problem are finding optimal paths for GPS navigation, routing in telecommunications networks, or updating computers in a network from a central server.

1.3 Definitions

We define some terms we will use in this paper:

MST: Given a connected, undirected graph $G = (V(G), E(G))$, with weight function $w : E(G) \to \mathbb{R}$, a MST of G is a connected, acyclic subgraph $T = (V(G), E(T))$ such that $\Sigma_{e \in E(T)} w(e)$ is minimal.

Cut: Given a graph $G = (V(G), E(G))$ and a subset $S \subseteq V(G)$, a cut $(S, V(G) \backslash S)$ of G is a partition of V. An edge $e \in E(G)$ **crosses** $(S, V(G) \backslash S)$ if e has one endpoint in either partition of the cut. A cut **respects** a set A of edges if no edge in A crosses the cut.

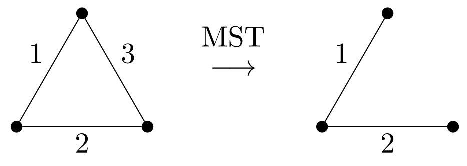
Light edge: An edge is called light for a cut $(S, V(G) \setminus S)$ if it is a minimal weight edge crossing it.

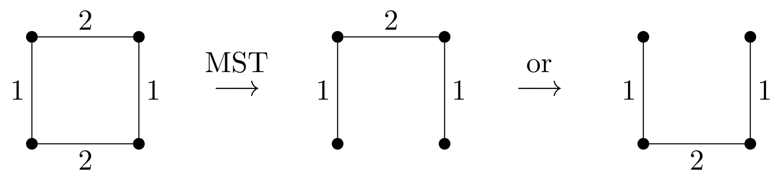
Safe edge: Let $A \subseteq E$ be a subset of a MST. An edge $e \in E$ is called safe for A if $A \cup \{e\}$ is still a subset of a MST.

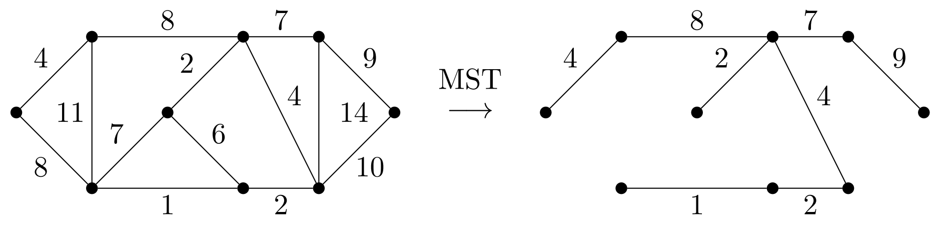
Path weight: If P is a path with edges $e_1, \dots, e_k$, then define the weight of P, $w(P)$ as $\sum_{i=1}^{k} w(e_i)$.

Minimal path: Let $G = (V(G), E(G))$ and $u, v \in V(G)$. Then define $\delta(u, v) = \min\{w(P) \mid P \text{ is a } u, v\text{-path in } G\}$.

1.4 Examples







2 Algorithms for finding minimum spanning trees

2.1 A generic MST algorithm

The two algorithms we consider in this paper both share very similar structure. The following generic MST algorithm will form the basis for both Kruskal and Prim's algorithms:
Given $G = (V(G), E(G))$. Start with the $T = \emptyset$. This will form a subset of $E(G)$ that will induce a MST of G. Apply the following step until $V(G[T]) = V(G)$: pick an edge e that is safe for T, and let $T = T \cup \{e\}$. [**?**]

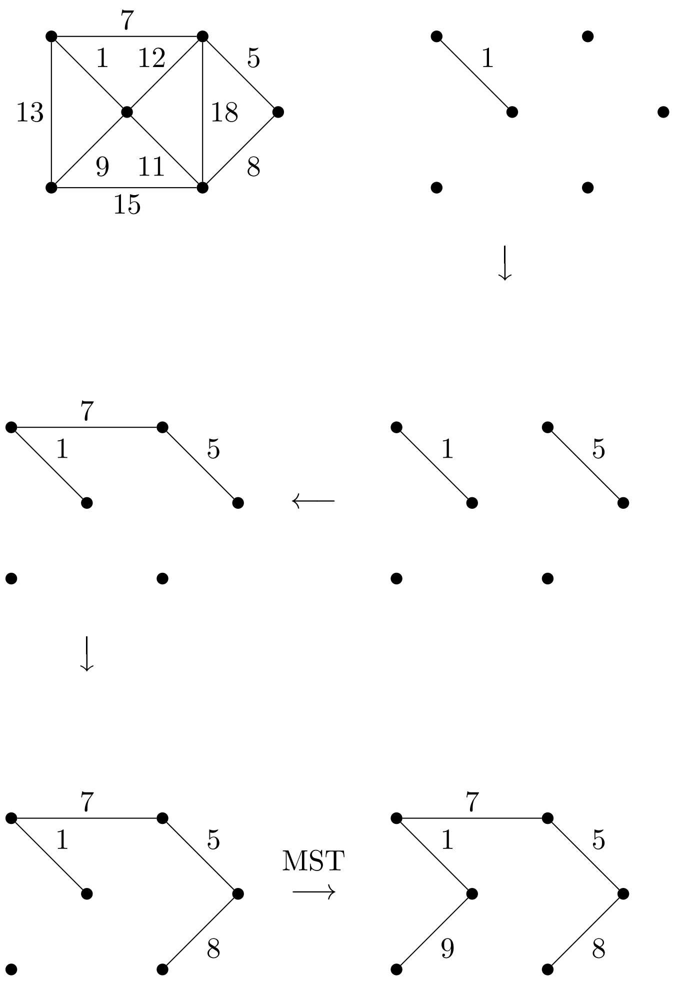
We first prove the correctness of this algorithm, then use its correctness to more easily prove the correctness of Kruskal and Prim's algorithms.

Proof. We wish to show that at the termination of this algorithm, $G[T]$ is a MST. Every edge added to T is safe, so whenever an edge is added to T, $G[T]$ remains a subgraph of some minimum-weight spanning tree. Thus after adding $n(G) - 1$ edges to T, $G[T]$ is still a subgraph of a MST. But any tree has only $n(G) - 1$ edges, so $G[T]$ is a MST. $\square$

2.2 Kruskal's algorithm

Given $G = (V(G), E(G))$. First the edges are sorted into a list in order of nondecreasing weight. Start with $T = \emptyset$. Apply the following step until $n(G) - 1$ edges have been chosen: if the lowest weight edge e has both endpoints in the same component of $G[T]$, ignore it. Otherwise, let $T = T \cup \{e\}$. Remove e from the list. [**?**]

2.3 Examples



2.4 Correctness of Kruskal's algorithm

We claim that by the end of this algorithm's execution, $G[T]$ is a MST. As mentioned, we will use the fact that the above generic MST algorithm is correct to simplify the proof that Kruskal's algorithm is correct. Notice that Kruskal's algorithm is nearly identical to the generic algorithm, and differs only in how it selects an edge to add to T. So it suffices to show that each edge Kruskal's algorithm selects to add to T is safe for T. First we refer to Theorem 23.1 from [**?**]:

Theorem: Let G be a connected, undirected graph with a real-valued weight function w defined on $E(G)$. Let A be a subset of $E(G)$ that is included in some MST for G, let $(S, V(G) \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V(G) \setminus S)$. Then edge (u, v) is safe for A. □

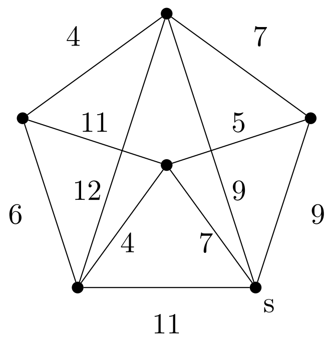
We now continue with the proof of the correctness of Kruskal's algorithm:

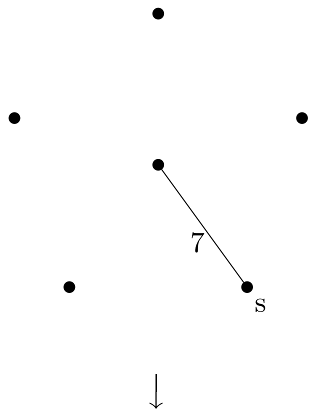
Proof. Define $C(v)$ to be the component of $G[T]$ containing vertex v. Then we have the cut $(C(v), V(G) \setminus C(v))$. (u, v) crosses $(C(v), V(G) \setminus C(v))$, since if both endpoints were in $C(v)$ it would not have been selected to be added to T. (u, v) has minimal weight for the same reason. So (u, v) is a light edge crossing $(C(v), V(G) \setminus C(v))$. $(C(v), V(G) \setminus C(v))$ respects T since one of it's partitions is defined to be an entire component in $G[T]$. So we can apply the above theorem, and thus (u, v) is safe for T. So Kruskal's algorithm is correct. [**?**] □

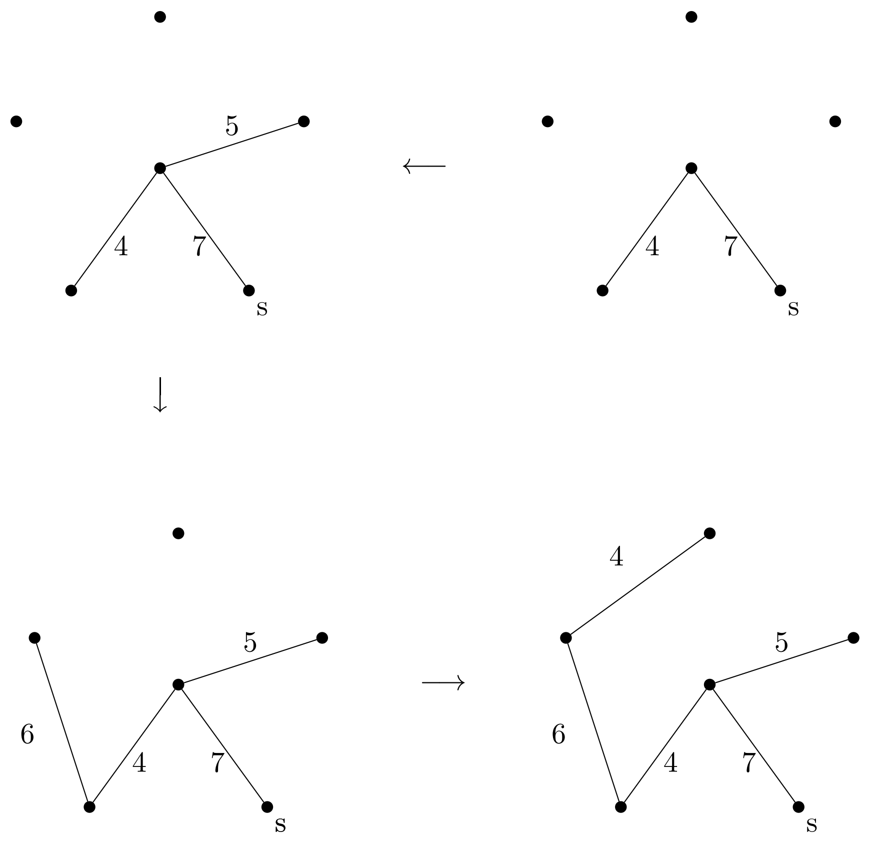
2.5 Prim's algorithm

Given a graph G, choose an arbitrary vertex $s \in V(G)$ and mark it as seen. Start with $T = \emptyset$, and apply the following until all vertices have been marked: find the lowest weight edge connecting a marked vertex to an unmarked vertex, and add it to T. [**?**]

2.6 Examples







2.7 Correctness of Prim's algorithm

We claim that by the end of this algorithm's execution, $G[T]$ is a MST. Again, this algorithm shares much structure with our generic MST algorithm, so it will suffice to show that each edge it chooses to add to T is safe for T.

Proof. Define the cut $(V(G[T]), V(G) \setminus V(G[T]))$. The edge chosen to be added to T is that which has minimum weight and exactly one endpoint in T. Thus it is by definition a light edge for $(V(G[T]), V(G) \setminus V(G[T]))$. This cut clearly respects T. Once again using the above theorem, we have that the chosen edge is safe for T, and so Prim's algorithm is correct. [**?**] $\square$

2.8 Comparing Kruskal and Prim's algorithms

As mentioned before, both Kruskal and Prim's algorithms share almost identical structure, and differ only in their method of edge choice.

Let $n = n(G)$, and $m = e(G)$.

Kruskal's algorithm is much faster than Prim's if we are provided with an already sorted list of edges by weight, in which case it runs in time proportional to $O(m)$. Under normal circumstances, however, the sorting of the edges bottlenecks the algorithm, and it runs in time proportional to $O(m \log m)$. $m \leq \frac{n^2-n}{2} < n^2$, so $O(\log n) \in O(\log m)$, so we can equivalently write Kruskal's algorithm's runtime as $O(m \log n)$. [**?**]

Prim's algorithm can be implemented using a priority queue which allows retrieving or updating the minimum weight edge in time $O(\log n)$ by only storing vertices and their minimum weight edges crossing into T. This operation occurs at most m times, so we have that Prim's algorithm also runs in time $O(m \log n)$. [**?**]

A much more precise analysis is possible, but roughly speaking Kruskal's and Prim's algorithms are about equivalent in terms of efficiency in the worst case.

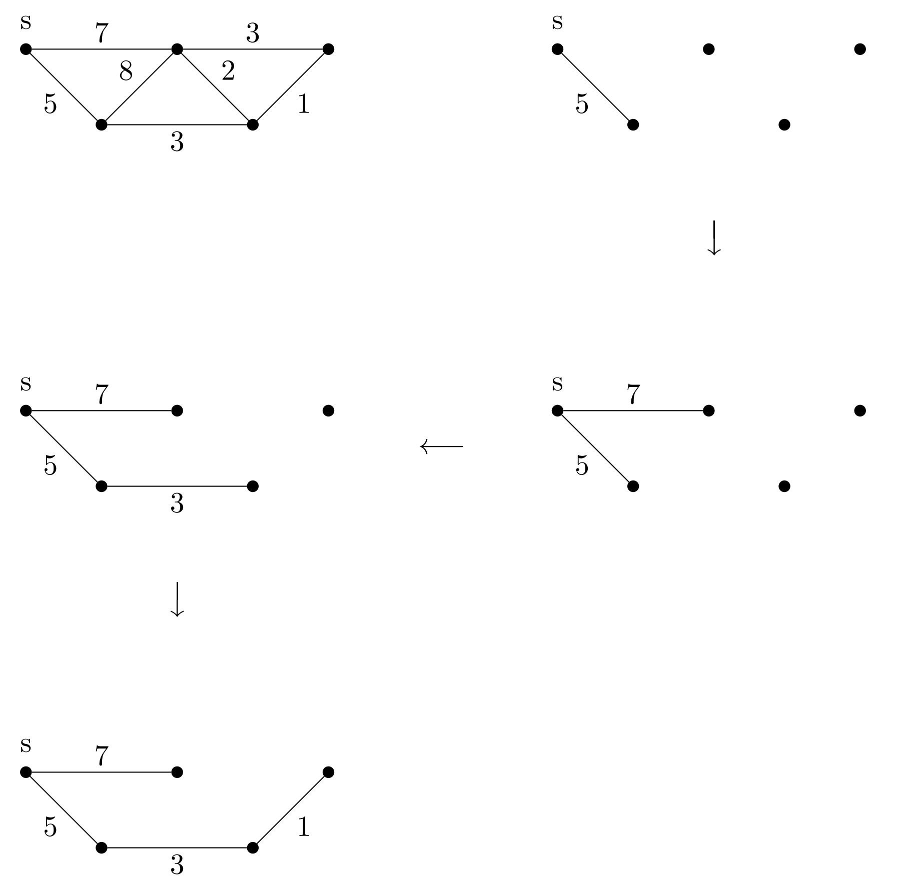
3 Algorithms for finding shortest paths

We keep a list $d_{v_1}, d_{v_2}, \ldots, d_{v_{n(G)}}$ of weight of the shortest we've seen from s to each vertex so far, and another list $\pi_{v_1}, \pi_{v_1}, \ldots, \pi_{v_{n(G)}}$ of each vertex's immediate parent in that path. When we "update" a vertex v from another vertex u whose minimum distance to s is known, we set $d_v = \min\{d_v, w((u,v)) + d_u\}$. If we change d_v to $w((u,v)) + d_u$, then we set π_v to u, otherwise we leave it. Before beginning the algorithm, we set $d_s = 0$ and $d_v = \infty$, $v \neq s$.

3.1 Dijkstra's algorithm

Given a graph G, start with $T = \emptyset$, and apply the following until all vertices have been marked: choose $v \in V(G)$ such that d_v is no more than d_u for any $u \in V(G)$, and mark it as seen. Update all unmarked vertices adjacent to v, and add (π_v, v) to T. [**?**]

3.2 Examples



3.3 Correctness of Dijkstra's algorithm

We claim that in $G[T]$, the (unique) s, v-path is minimal for any $v \in V(G)$. That is, if d_v is the minimum weight of an s, v-path possible in $G[T]$, we claim that when v is marked, $d_v = \delta(s, v)$ for all $v \in V(G)$.

Proof. We prove this fact by induction on the number of marked vertices, k. When $k = 1$, we have that $d_v = 0 = \delta(s, s)$. Let v be the vertex chosen to be marked next. Then assume for our induction hypothesis that for all $x \in V(G[T])$, $d_x = \delta(s, x)$. Suppose for a contradiction that there exists an s, v-path P in G such that $w(P) < d_v$. P starts in on a marked vertex and ends on an unmarked vertex (since v is unmarked, and P ends on v), so let (x, y) be the first edge in P joining a marked and an unmarked vertex. $x \in V(G[T])$, so by the induction hypothesis $d_x = \delta(s, x)$. $y \in N(x)$, so since x has already been marked, y has been updated, so $d_y = \delta(s, y)$. So we have

$$\begin{aligned} d_y &= \delta(s, y) \\ &\leq \delta(s, v) && \text{Since } P \text{ is a shortest path} \\ &\leq d_v. \end{aligned}$$

Both y, and v were unmarked, but v was chosen. Thus $d_v \leq d_y$. This implies that the above chain of inequalities are actually equalities, so we have that $d_y = \delta(s, y) = \delta(s, v) = d_v$. This means that $d_v = \delta(s, v)$, a contradiction, since $\delta(s, v) = w(P) < d_v = \delta(s, v)$, or $\delta(s, v) < \delta(s, v)$. Thus there is no such path P, and $d_v = \delta(s, v)$. So by induction, whenever we mark v, its path to s has minimal weight, and thus when the algorithm terminates, we are left with a tree T such that any s, v-path in T has weight $\delta(s, v)$, as desired. [?] [?] □

3.4 Relations between Dijkstra's algorithm and minimum spanning tree algorithms

Dijkstra's algorithm and Prim's algorithm are almost identical: when choosing a new vertex to mark, Dijkstra's checks the distance to s (that is, shortest distance to a marked vertex, then distance from that vertex to s) and Prim's only checks distance to a marked vertex. Although their implementations are nearly the same, their outputs are not. While Dijkstra's algorithm *can* output a MST, in general it does not, and its output does not generally provide any insight into finding a MST, or vice versa.

4 Concluding remarks

This has been a short survey of some classical solutions to the minimum-weight spanning tree problem and the single-source shortest path problem. We examined three algorithms: Kruskal's algorithm, Prim's algorithm, and Dijkstra's algorithm, considering their methods, proving their correctness, and comparing them to each other.