CSE 5311: Design and Analysis of Algorithms

Minimum Spanning Trees

Alex Dillhoff

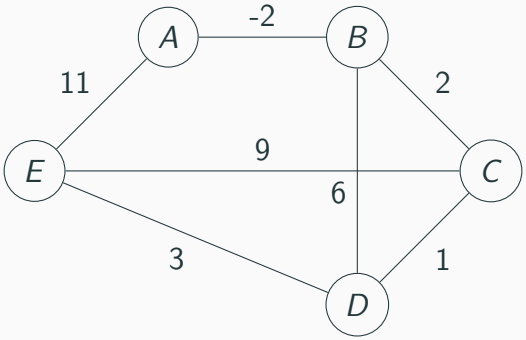
University of Texas at Arlington

Minimum Spanning Trees

Minimum spanning trees are undirected graphs that connect all of the vertices such that there are no redundant edges and the total weight is minimized.

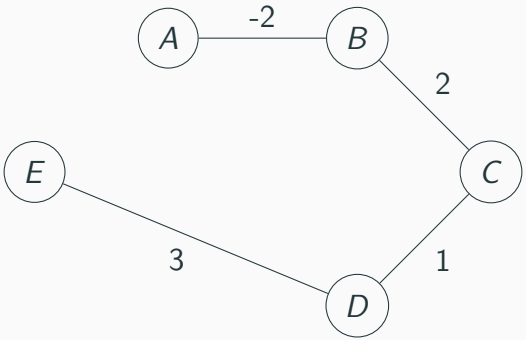
They are useful for finding the shortest path between two points in a graph.

Minimum Spanning Trees



An example graph with edge weights listed.

Minimum Sanning Trees



The minimum spanning tree

Minimum Spanning Trees

Other useful application of MSTs include

- **network design**: it is useful to know the least expensive path with respect to either latency or resource cost for telecommunications networks, transportation networks, or electrical grids.
- **approximation algorithms**: MSTs can be used to approximate the solution to the traveling salesman problem.
- **clustering**: MSTs can be used to cluster data points in a graph.
- **image segmentation**: MSTs can be used to segment images into smaller regions.

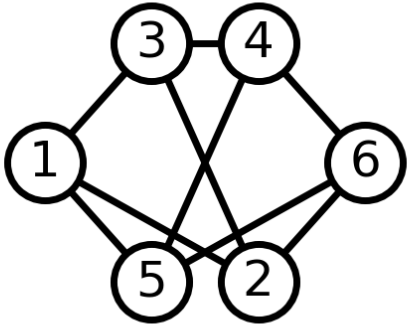
What is a Minimum Spanning Tree

Definition

Let G be a connected, undirected graph with edges E, vertices V, and edge weights w.

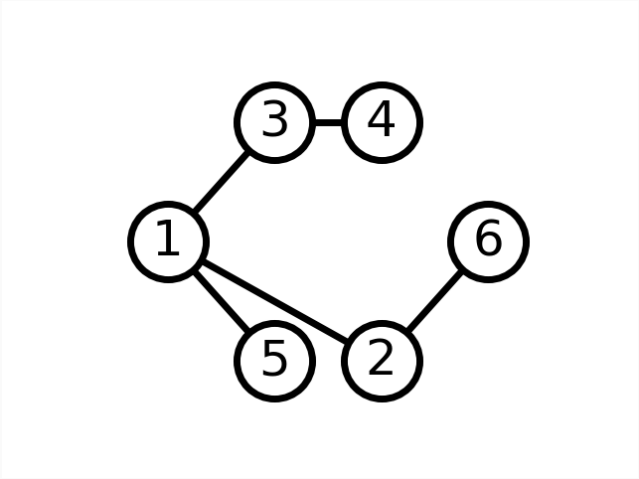
A **minimum spanning tree** is a subset $T \subseteq E$ that connects all of the vertices such that the total weight is minimized.

Definition



An undirected graph G.

Definition



A minimum spanning tree of G.

Finding the MST

Finding the MST

There are two greedy algorithms for finding the minimum spanning tree of a graph.

1. Kruskal's algorithm
2. Prim's algorithm

Finding the MST

The general algorithm for finding the minimum spanning tree of a graph grows a set of edges T from an empty set.

At each step, the algorithm adds the edge with the smallest weight that does not create a cycle.

The algorithm terminates when T is a complete tree.

Finding the MST

```
T = {}
while T is not a spaning tree
    find the edge e with the smallest weight that does not create a cycle
    T = T union {e}
```

Finding the MST

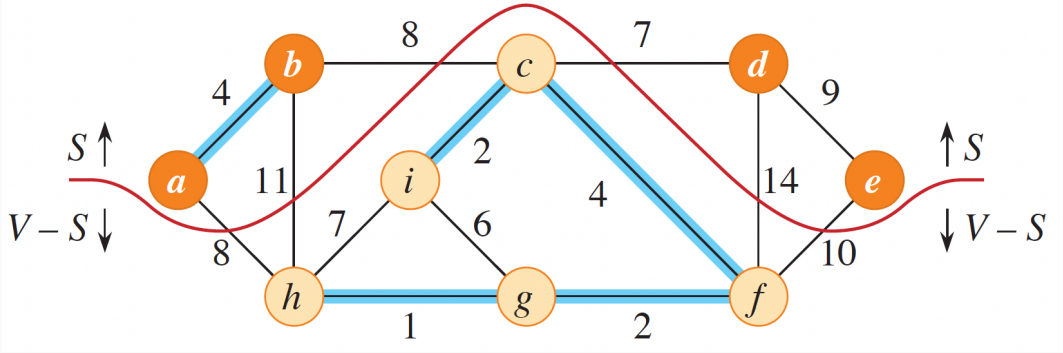
Each edge e that is added must result in a tree that is a subset of the minimum spanning tree.

The challenge of this algorithm is actually finding such an edge.

How would we know such an edge if we saw it?

We first need to define a few properties which will shine light on this.

Finding the MST



An example of a cut in a graph (Cormen et al.).

Finding the MST

A **cut** of a graph G is a partition of the vertices V into two disjoint sets S and $V - S$.

An edge e **crosses** the cut if one of its endpoints is in S and the other is in $V - S$.

If no edge in a given set E crosses the cut, then that cut **respects** E.

Finding the MST

An edge that is the minimum weight edge that crosses a cut is called a **light edge**.

With these definitions, we can now formally define how to find a **safe edge**, which is an edge that can be added to the current set of edges T without creating a cycle.

Finding the MST

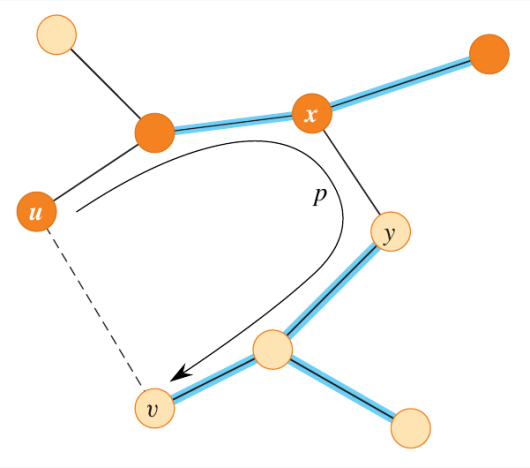
Theorem 21.1 from Cormen et al.

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 e be a light edge crossing $(S, V - S)$.

Then, edge e is safe for A.

Finding the MST



Visualization of proof of Theorem 21.1 from Cormen et al.

Finding the MST

The two sets in the figure above represent vertices in S (orange) and vertices in $V - S$ (tan).

T is the original MST depicted in the figure.

The dotted line is the new edge (u, v) to consider.

A is a subset of edges in T represented by the blue lines.

If the safe edge (u, v) is already in the original MST T, then we are done.

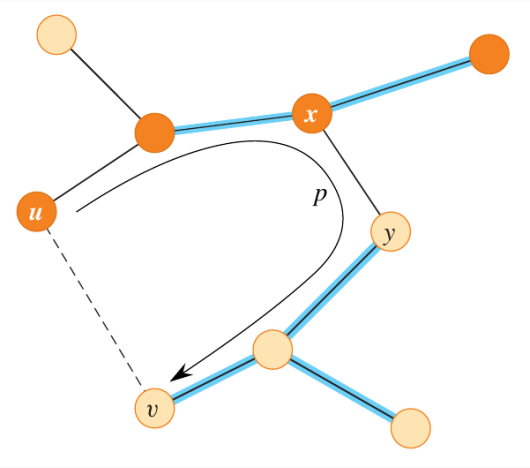
Finding the MST

The vertices u and v lie on opposite sides of the cut.

The edge (u, v) would introduce a cycle since there is already a path from u to v in T that crosses the cut via (x, y).

Since both (u, v) and (x, y) are light edges that cross the cut, then it must be that $w(u, v) \leq w(x, y)$.

Finding the MST



Visualization of proof of Theorem 21.1 from Cormen et al.

Finding the MST

Let T' be the minimum spanning tree with (x, y) replaced by (u, v). That is $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Since T is a minimum spanning tree, then $w(T) \leq w(T')$.

Then T' is also a minimum spanning tree.

Therefore, (u, v) is safe for A.

Finding the MST

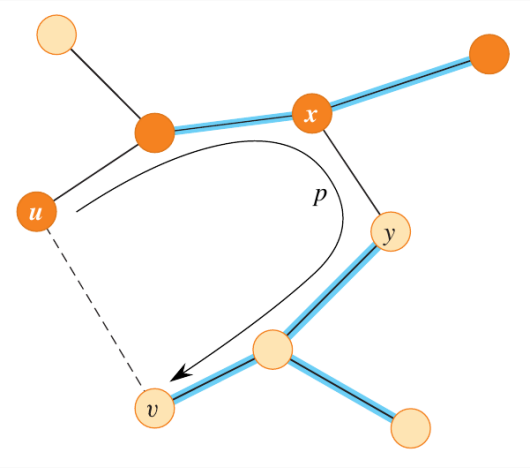


Figure 1: Visualize proof of Theorem 21.1 from Cormen et al.

Finding the MST

Corollary 21.2 from Cormen et al.

We can also view this in terms of **connected components**, which are subsets of vertices that are connected by a path.

If C and C' are two connected components in T and (u, v) is a light edge connecting C and C', then (u, v) is safe for T.

Finding the MST

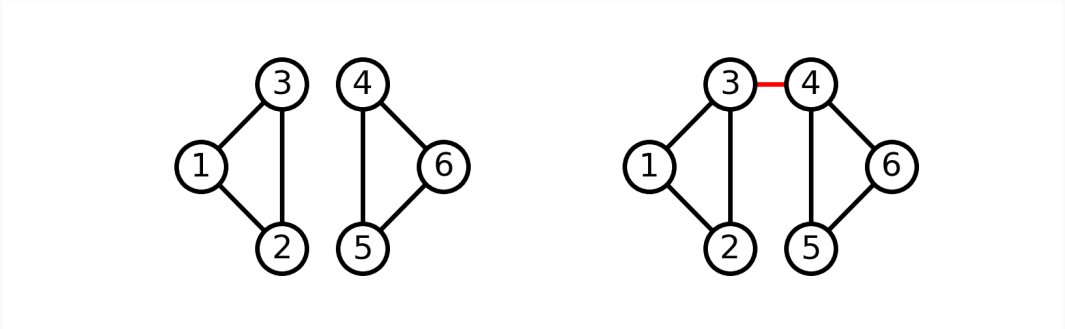


Figure 2: Connected components of a graph. The red line in the right graph is a **safe edge**.

Kruskal's Algorithm

Kruskal's Algorithm

The first solution to the minimum spanning tree that we will study is called **Kruskal's algorithm**.

This algorithm grows a forest of trees from an empty set.

At each step, the algorithm adds the lightest edge that does not create a cycle.

The algorithm terminates when the forest is a single tree.

Kruskal's Algorithm

This can be viewed as an **agglomerative clustering** algorithm.

The algorithm starts with each vertex in its own cluster.

At each step, the algorithm merges the two clusters that are closest together.

The algorithm terminates when there is only one cluster.

Kruskal's Algorithm

```
A = {}
for each vertex v in G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) in G.E, taken in nondecreasing order by weight
    if FIND-SET(u) != FIND-SET(v)
        A = A union {(u, v)}
        UNION(u, v)
return A
```

Kruskal's Algorithm

A step-by-step example of an implementation in Python is available here.

Analysis

The running time is dependent on how the disjoint-set of vertices is implemented.

In the best known case, a *disjoint-set-forest* implementation should be used.

Creating a list of edges takes $O(E)$ time.

Sorting the edges takes $O(E \log E)$ time.

Analysis

The `for` loop iterates over each edge, which is $O(E)$.

All disjoint-set operations take $O((V + E)\alpha(V))$ time.

Since the graph is connected, $E \geq V - 1$, so the total running time is $O(E \log E + E + E\alpha(V)) = O(E \log E + E\alpha(V)) = O(E \log V)$.

Prim's Algorithm

Prim's Algorithm

The second solution starts at an arbitrary vertex in a set A and adds a new vertex to A in a greedy fashion.

To efficiently select a new edge to add, Prim's algorithm uses a priority queue to keep track of the lightest edge that crosses the cut.

The algorithm terminates when A is a complete tree.

Prim's Algorithm

```
A = {}
for each vertex v in G.V
    key[v] = infinity
    pi[v] = NIL
key[r] = 0
Q = G.V
while Q is not empty
    u = EXTRACT-MIN(Q)
    A = A union {u}
    for each vertex v in G.Adj[u]
        if v in Q and w(u, v) < key[v]
            pi[v] = u
            key[v] = w(u, v)
            DECREASE-KEY(Q, v, key[v])
```

Prim's Algorithm

Prim's algorithm implicitly maintains the set $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$, where r is the root.

When the `while` loop terminates, $A = \{(v, v.\pi) : v \in V - \{r\}\}$, since the queue is empty.

Prim's Algorithm

The critical part of this is to understand how the algorithm changes the key values.

For all vertices $v \in Q$, if $v.\pi \neq NIL$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ that connects v to a vertex in A.

Prim's Algorithm

A step-by-step example of an implementation in Python is available here.

Prim's Algorithm

Prim's algorithm uses a priority queue to keep track of the lightest edge that crosses the cut.

If the priority queue is implemented as a min-heap, which has a worst-case running time of $O(\log V)$ for both `EXTRACT-MIN` and `DECREASE-KEY`.

Prim's Algorithm

The algorithm calls `EXTRACT-MIN` once for each vertex, which is $O(V \log V)$.

The algorithm calls `DECREASE-KEY` once for each edge, which is $O(E \log V)$.

The total running time is $O(V \log V + E \log V) = O(E \log V)$.