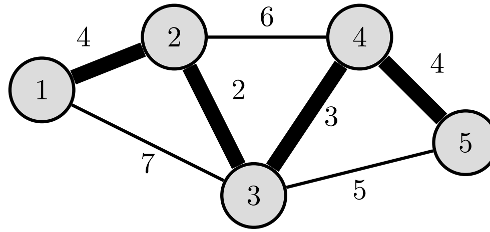
Minimum Spanning Trees

Made by Larry Wang and Charles Zhao, Edited and Delivered by Jongwan Kim

February 22nd, 2019

1 Introduction

A *spanning tree* of a connected, undirected, and weighted graph G is a subgraph that is a tree including all the vertices of G. The Minimum Spanning Tree (MST) problem is the problem of finding a spanning tree of G with minimal total edge weight. Note that if not all edge weights are distinct, then there may be multiple MSTs for a given graph.



2 Kruskal's Algorithm

Kruskal's Algorithm finds the MST by greedily adding edges; for all edges not yet in the MST, we can repeatedly add the edge of minimum weight to the MST except when adding said edge forms a cycle (which violates the tree structure). This can be done by sorting the edges in order of non-decreasing weight. Furthermore, we can easily determine whether adding an edge will create a cycle in (for all practical purposes) constant time using Union Find. Note that since the most expensive operation is sorting the edges, the computational complexity of Kruskal's Algorithm is $O(E \log E)$.

To see why this will always work, assume that we are trying to add edge e, which connects vertices u and v, to the MST. If the only path between u and v is through e, then adding e cannot form a cycle, and Kruskal will add e to the MST. However, assume that another path from u to v exists. This path consists of a sequence of edges. By Kruskal, any of these edges with weight less than e are already in the MST. If all of these edges have weight less than the weight of e, then we skip over e since adding it would create a cycle. Otherwise, note that e has a lower weight than any of the edges in this path that are not yet in the MST, so adding e is optimal.

Algorithm 1 Kruskal's Algorithm

```
function KRUSKAL(v, e)
    e ← SORT(e)
    UnionFind uf ← UNIONFIND(v)
    for each edge ∈ e do
        if not SAMESET(edge.first, edge.second) then
            UNION(edge.first, edge.second)
    return uf[0]
```

3 Prim's Algorithm

Rather than greedily adding edges, Prim's algorithm greedily adds vertices; on each iteration, we add the vertex that is closest to the current MST until all vertices have been added. The process of finding the closest vertex to the MST can be done efficiently using a priority queue in $O(\log N)$. After removing a vertex, we add all of its neighbors that are not yet in the MST to the priority queue and repeat. To begin the algorithm, we simply add any vertex to the priority queue. Note that Prim's algorithm has complexity $O(E \log V)$ since in the worst case every edge will be checked and its corresponding vertex will be added to the priority queue.

Alternatively, we may linearly search for the closest vertex instead of using a priority queue. Each linear pass runs in time $O(V)$, and this must be repeated V times. Thus, this version of Prim's algorithm has complexity $O(V^2)$. Note that this complexity is preferable for dense graphs (in which $E \approx V^2$).

To see why Prim's algorithm works, consider a cut of the graph partitioning the graph into two sets of vertices A and B. Now consider the set of edges E connecting a vertex in A to a vertex in B. Note that at least one edge in E must be in the MST. This means that the edge in E with minimum weight must be in the MST. To prove Prim's Algorithm, make A the set of vertices currently in the MST and B the set of all other vertices. Adding the vertex closest to the current MST is equivalent to adding the edge of minimum weight between A and B. Prim's Algorithm follows by repeating this process.[1]

Algorithm 2 Prim

```
for all vertices v do
    dist(v) ← ∞
    visited(v) ← 0
    prev(v) ← −1
dist(src) ← 0
while ∃v s.t. visited(v) = 0 do
    v ≡ v s.t. visited(v) = 0 with min dist(v)
    visited(v) ← 1
    for all neighbors u of v do
        if visited(u) = 0 then
            if weight(v, u) < dist(u) then
                dist(u) ← weight(v, u)
                prev(u) ← v
```

[1]pseudocode taken from Sam Hsiang's *Crash Course Coding Companion*

4 Problem Variants

4.1 Steiner Tree (NP-Hard)

Given a subset of vertices called terminal vertices, find a minimum spanning tree of those vertices. The spanning tree may contain vertices that are not in the set called Steiner vertices.

Minimum spanning tree: terminal vertices include all vertices
Shortest path: there are only two terminal vertices

4.2 Spanning Arborescence

This variant of minimum spanning tree is quite unique in a sense that it literally violates the definition of a "minimum spanning tree". As I mentioned before, a minimum spanning tree is a set of undirected edges that connect all vertices with minimum cost. In this variant, we will try to find a "minimum spanning tree" for a directed graph. Arborescence is a graph in which there is exactly one directed path from node u to node v. To be more technically correct, we are trying to find an arborescence that spans through all vertices.

4.2.1 Edmonds/Chu-Liu Algorithm

This algorithm is based on repeatedly contracting a digraph until an arborescence is found. Here are the steps of the algorithm:

1. Find cheapest edge entering each vertex and replace all weights with reduced costs
2. Find a 0-cost cycle and contract it into one supervertex.
3. Repeat step 1 and 2.
4. If there is no 0-cost cycle, then we have found the arborescence. Now, we have to uncontract the supervertices.
5. When uncontracting, make sure to take out one edge at each uncontraction that violates the arborescence.
6. When there are no more to uncontract, we are done.

The runtime of this algorithm is $O(EV)$.

5 Problems

1. USACO February 2015 Contest, Silver Problem 3. Superbull
2. USACO February 2016 Contest, Platinum Problem 2. Fenced In