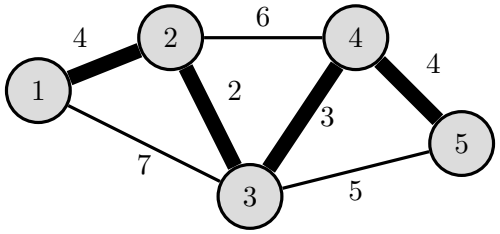
Minimum Spanning Trees

Larry Wang and Charles Zhao

October 3, 2016

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 $O(E \log E)$ since in the worst case every edge will be checked and its corresponding vertex will be added to the priority queue. However, this can be improved to $O(E \log V)$ if we were to update the distances of vertices in the priority queue rather than re-add them. This keeps the maximum size of the priority queue bounded at V. Note that this optimization usually isn't necessary for competitive programming.

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]

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

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
```

4 Problem Variants

4.1 Minimum Spanning Subgraph

What do you do if some of the edges are already fixed? It doesn't matter if the fixed edges form a cycle, simply continue running Kruskal's algorithm on the remaining edges.

4.2 Minimax and Maximin

The minimax problem is the problem of finding a path between two vertices such that the maximum edge weight is minimal. Since this problem wants a path with low individual edge weights regardless of the length (i.e. number of edges) of the path, using an MST is a good idea. The solution to the minimax problem is the maximum edge weight along the path between the two vertices in the MST.

Analogously, the maximin problem is to find a path between two vertices such that the minimum edge weight is maximal. The solution to the maximin problem is very similar, except we use a maximum spanning tree instead. The maximum spanning tree can be found by finding the MST of the graph where all of the edge weights are negated.

5 Problems

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