Lecture 17: Minimum Spanning Trees (1997)

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Show that you can topologically sort in $O(n+m)$ by repeatedly deleting vertices of degree 0.

The correctness of this algorithm follows since in a DAG there must always be a vertex of indegree 0, and such a vertex can be first in topological sort. Suppose each vertex is initialized with its indegree (do DFS on G to get this). Deleting a vertex takes O(degree v). Reduce the indegree of each efficient vertex - and keep a list of degree-0 vertices to delete next.

Time: $\Sigma_{i=1}^{n} O(deg(v_i)) = O(n+m)$

Minimum Spanning Trees

A tree is a connected graph with no cycles. A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.

A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

There can be more than one minimum spanning tree in a graph $\rightarrow$ consider a graph with identical weight edges.

The minimum spanning tree problem has a long history – the first algorithm dates back at least to 1926!.

Minimum spanning tree is always taught in algorithm courses since (1) it arises in many applications, (2) it is an important example where *greedy* algorithms always give the optimal answer, and (3) Clever data structures are necessary to make it work.

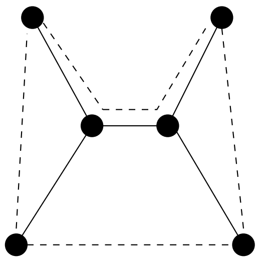
In greedy algorithms, we make the decision of what next to do by selecting the best local option from all available choices – without regard to the global structure.

Applications of Minimum Spanning Trees

Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire. Much of the work on minimum spanning (and related Steiner) trees has been conducted by the phone company.

Minimum spanning trees provide a reasonable way for *clustering* points in space into natural groups.

When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.



The Option Traveling System tour is at most twice the length of the minimum spanning tree.

Note: There can be more than one minimum spanning tree considered as a group with identical weight edges.

Prim’s Algorithm

If G is connected, every vertex will appear in the minimum spanning tree. If not, we can talk about a minimum spanning forest.

Prim’s algorithm starts from one vertex and grows the rest of the tree an edge at a time.

As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.

During execution we will label each vertex as either in the tree, *fringe* - meaning there exists an edge from a tree vertex, or *unseen* - meaning the vertex is more than one edge away.

Select an arbitrary vertex to start.

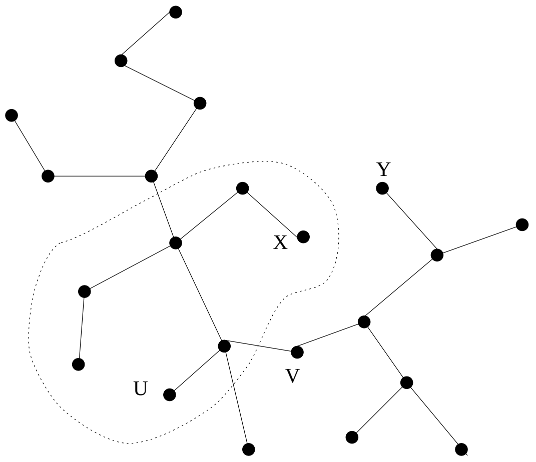
```
While (there are fringe vertices)
        select minimum weight edge between tree and fringe
        add the selected edge and vertex to the tree
```

Clearly this creates a spanning tree, since no cycle can be introduced via edges between tree and fringe vertices, but is it minimum?

Why is Prim's algorithm correct?

Don't be scared by the proof – the reason is really quite basic:
Theorem: Let G be a connected, weighted graph and let $E' \subset E$ be a subset of the edges in a MST $T = (V, E_T)$. Let V' be the vertices incident with edges in E'. If (x, y) is an edge of minimum weight such that $x \in V'$ and y is not in V', then $E' \cup \{x, y\}$ is a subset of a minimum spanning tree.
Proof: If the edge is in T, this is trivial.
Suppose (x, y) is not in T Then there must be a path in T from x to y since T is connected. If (v, w) is the first edge on this path with one edge in V', if we delete it and replace it with (x, y) we get a spanning tree.
This tree must have smaller weight than T, since $W(v, w) >$

$W(x, y)$. Thus T could not have been the MST.



Thus we cannot go wrong with the greedy strategy the way we could with the traveling salesman.

PRIM's Algorithm is correct!

Prim's Algorithm is correct!
Thus we cannot go wrong with the greedy strategy the way we could with the traveling salesman problem.

But how fast is Prim's?

That depends on what data structures are used. In the simplest implementation, we can simply mark each vertex as tree and non-tree and search always from scratch:

```
Select an arbitrary vertex to start.
While (there are non-tree vertices)
        select minimum weight edge between tree and fringe
        add the selected edge and vertex to the tree
```

This can be done in $O(nm)$ time, by doing a DFS or BFS to loop through all edges, with a constant time test per edge, and a total of n iterations.
Can we do faster? If so, we need to be able to identify fringe vertices and the minimum cost edge associated with it, fast.

We will augment an adjacency list with fields maintaining fringe information.
Vertex:

fringelink pointer to next vertex in fringe list.

fringe weight cheapest edge linking v to l.

parent other vertex with v having fringeweight.

status intree, fringe, unseen.

adjacency list the list of edges.

Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list.
After adding a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices

(there may be a cheaper way through the new vertex) can be done in $O(n)$ time.
Total time is $O(n^2)$.

Kruskal's Algorithm

Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim's algorithm is optimal in the worst case. Is that all she wrote?

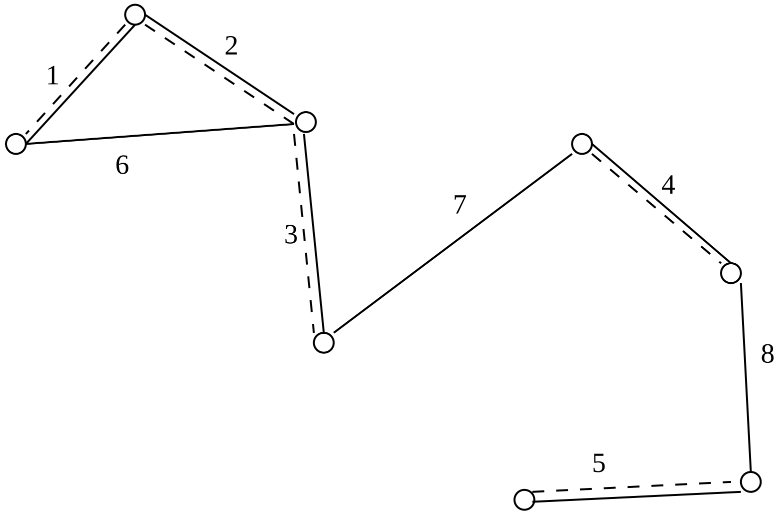
The complexity of Prim's algorithm is independent of the number of edges. Can we do better with sparse graphs? Yes!

Kruskal's algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle. Obviously, this gives a spanning tree, but is it minimal?

Why is Kruskal's algorithm correct?

Theorem: Let G be a weighted graph and let $E' \subset E$. If E' is contained in a MST T and e is the smallest edge in $E - E'$ which does not create a cycle, $E' \cup e \subseteq T$.

Proof: As before, suppose e is not in T. Adding e to T makes a cycle. Deleting another edge from this cycle leaves a connected graph, and if it is one from $E - E'$ the cost of this tree goes down. Since such an edge exists, T could not be a MST.


1
2
6
3
7
4
8
5

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle the forest we have thus far built – if so discard, else add to forest. With a BFS/DFS, this can be done in $O(n)$ time (since the tree has at most n edges).

The total time is $O(mn)$, but can we do better?
Kruskal's algorithm builds up connected components. Any edge where both vertices are in the same connected component create a cycle. Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!

```
Put the edges in a heap
$count = 0$
while $(count < n - 1)$ do
        get next edge $(v, w)$
        if (component (v) $\neq$ component(w))
                add to T
                component (v)=component(w)
```

If we can test components in $O(\log n)$, we can find the MST in $O(m \log m)$!
Question: Is $O(m \log n)$ better than $O(m \log m)$?

Union-Find Programs

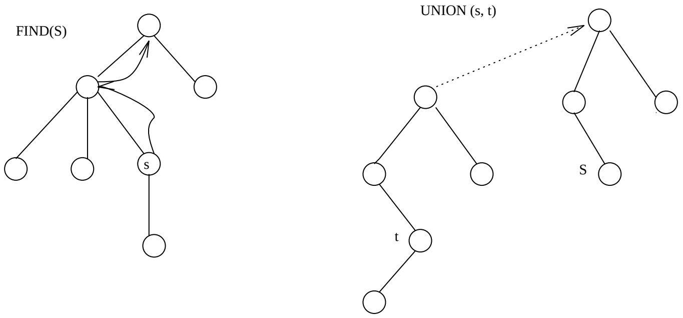
Our analysis that Kruskal's MST algorithm is $O(m \log m)$ requires a fast way to test whether an edge links two vertices in the same connected component.

Thus we need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by *UNION* and *FIND* operations:

```
Is $s_i \equiv s_j$
        $t = \text{Find}(s_i)$
        $u = \text{Find}(s_j)$
        Return (Is $t = u$?)
```

```
Make $s_i \equiv s_j$
    $t = d(s_i)$
    $u = d(s_j)$
    Union$(t, u)$
```

Find returns the name of the set and *Union* sets the members of t to have the same name as u.
We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.
A simple implementation is to represent each set as a tree, with pointers from a node to its parent. Each element is contained in a node, and the *name* of the set is the key at the root:

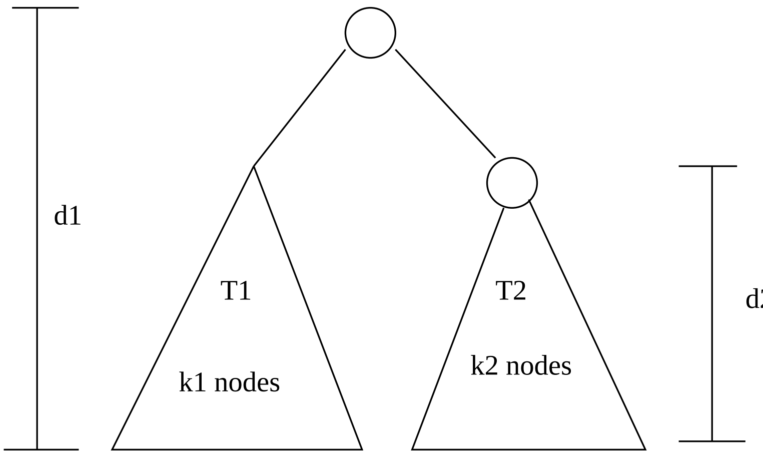


In the worst case, these structures can be very unbalanced:

```
For $i = 1$ to $n/2$ do
        UNION(i,i+1)
For $i = 1$ to $n/2$ do
        FIND(1)
```

We want the limit the height of our trees which are effected by

UNIONs. When we union, we can make the tree with fewer nodes the child.
Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal height!
Lemma: If $Union(t, v)$ attaches the root of v as a subtree of t iff the number of nodes in t is greater than or equal to the number in v, after any sequence of unions, any tree with $h/4$ nodes has height at most $\lfloor \lg h \rfloor$.
Proof: By induction on the number of nodes k, $k = 1$ has height 0.
Assume true to $k - 1$ nodes. Let d_i be the height of the tree t_i



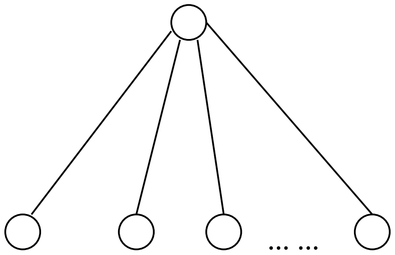
k = k1+ k2 nodes

d is the height

If $(d_1 > d_2)$ then $d = d_1 \leq \lfloor \log k_1 \rfloor \leq \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \log k \rfloor$
If $(d_1 \leq d_2)$, then $k_1 \geq k_2$.
$d = d_2 + 1 \leq \lfloor \log k_2 \rfloor + 1 = \lfloor \log 2k_2 \rfloor \leq \lfloor \log(k_1 + k_2) \rfloor = \log k$

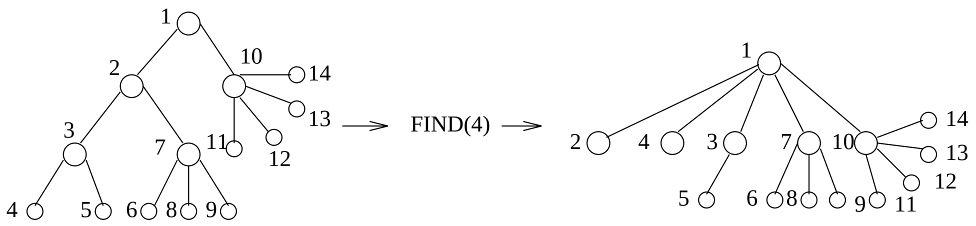
Can we do better?

We can do *unions* and *finds* in $O(\log n)$, good enough for Kruskal's algorithm. But can we do better?
The ideal *Union-Find* tree has depth 1:



N-1 leaves

On a find, if we are going down a path anyway, why not change the pointers to point to the root?



This path compression will let us do better than $O(n \log n)$ for n union-finds.
$O(n)$? Not quite …Difficult analysis shows that it takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and $\alpha($number of atoms in the universe$) = 5$.