Lecture 13:
Minimum Spanning Trees

Steven Skiena

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

http://www.cs.stonybrook.edu/~skiena

Topic: Problem of the Day

Problem of the Day

Your job is to arrange n rambunctious children in a straight line, facing front. You are given a list of m statements of the form "i hates j". If i hates j, then you do not want put i somewhere behind j, because then i is capable of throwing something at j.

1. Give an algorithm that orders the line, (or says that it is not possible) in $O(m+n)$ time.

2. Suppose instead you want to arrange the children in rows, such that if i hates j then i must be in a lower numbered row than j. Give an efficient algorithm to find the minimum number of rows needed, if it is possible.

Questions?

Topic: Minimum Spanning Trees

Weighted Graph Algorithms

Beyond DFS/BFS exists an alternate universe of algorithms for *edge-weighted graphs*.

Our adjacency list representation quietly supported these graphs:

```
typedef struct edgenode {
    int y;                  /* adjacency info */
    int weight;             /* edge weight, if any */
    struct edgenode *next;  /* next edge in list */
} edgenode;


typedef struct {
    edgenode *edges[MAXV+1];  /* adjacency info */
    int degree[MAXV+1];       /* outdegree of each vertex */
    int nvertices;            /* number of vertices in the graph */
    int nedges;               /* number of edges in the graph */
    int directed;             /* is the graph directed? */
} graph;
```

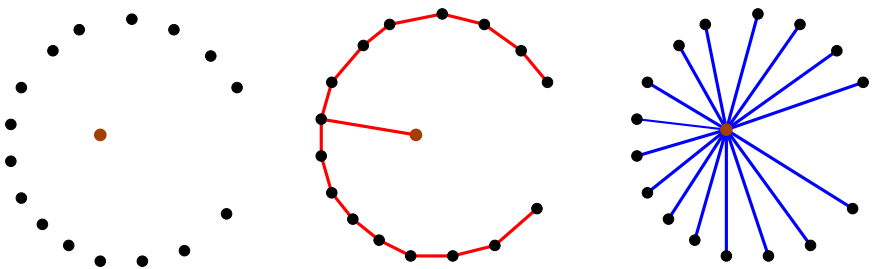
Minimum Spanning Trees

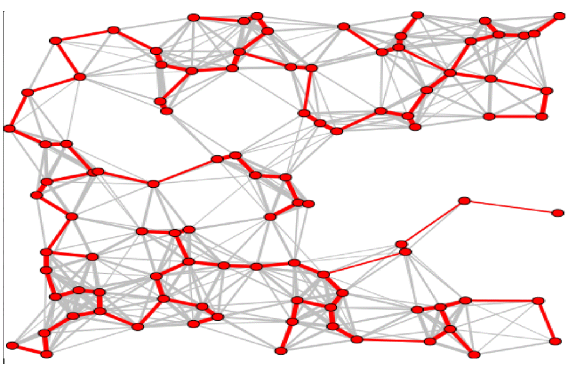
A tree is a connected graph with no cycles. A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.

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.

Find the Minimum Spanning Tree





Why Minimum Spanning Trees?

The minimum spanning tree problem has a long history – the first algorithm dates back to 1926!
MST is taught in algorithm courses because:

- It arises in many graph applications.
- It is problem where the *greedy* algorithm always gives the optimal answer.
- Clever data structures are necessary to make it work.

Greedy algorithms make the decision of what next to do by selecting the best local option from all available choices.

Applications of Minimum Spanning Trees

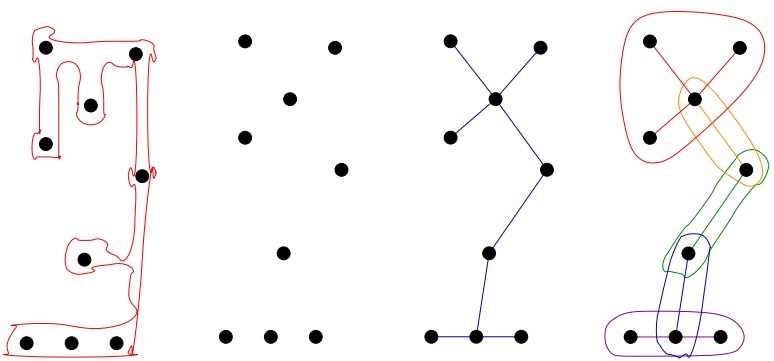
Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.

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

What are natural clusters in the friendship graph?

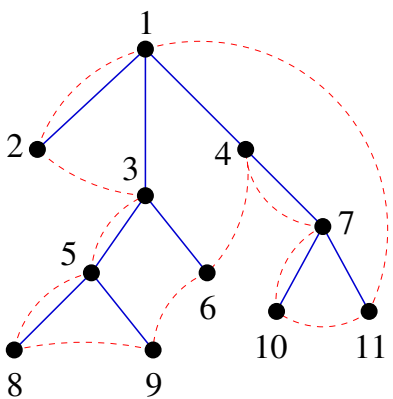
Minimum Spanning Trees and Net Partitioning

One of the war stories in the text describes how to partition a graph into compact subgraphs by deleting large edges from the minimum spanning tree.



Minimum Spanning Trees and TSP

For points in the Euclidean plane, MST yield a good heuristic for the traveling salesman problem:



The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

Questions?

Topic: Prim's Algorithm

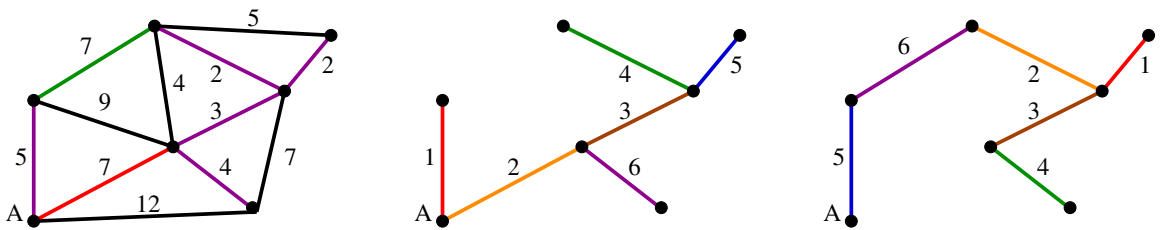
Prim's Algorithm

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.

Prim's Algorithm in Action



https://upload.wikimedia.org/wikipedia/en/3/33/Prim-algorithm-animation-2.gif

Prim's Algorithm (Pseudocode)

During execution each vertex v is either in the tree, *fringe* (meaning there exists an edge from a tree vertex to v) or *unseen* (meaning v is more than one edge away).

Prim-MST(G)

Select an arbitrary vertex s to start the tree from.

While (there are still non-tree vertices)

Pick min cost edge between tree/non-tree vertices

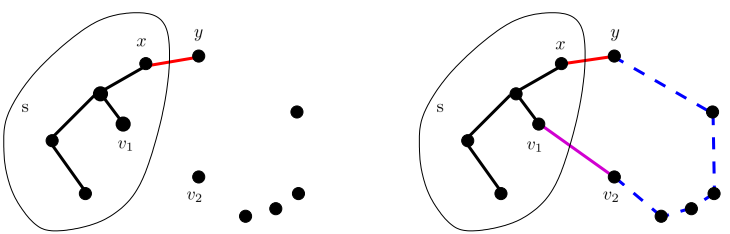
Add the selected edge and vertex to the tree T_{prim}.

This creates a spanning tree, since no cycle can be introduced.

But is it minimum?

Why is Prim Correct? (Proof by Contradiction)

- If Prim's algorithm is not correct, these must be some graph G where it does not give the minimum cost spanning tree.
- If so, there must be a first edge (x, y) Prim adds, such that the partial tree V' cannot be extended into a MST.



(a) (b)

The Contradiction

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y, because the tree is connected.
- Let (v_1, v_2) be the other edge on this path with one end in V'.
- Replacing (v_1, v_2) with (x, y) we get a spanning tree. with smaller weight, since $W(v, w) > W(x, y)$. Thus you did not have the MST!!
- If $W(v, w) = W(x, y)$, then the tree is the same weight, but we couldn't have made a fatal mistake picking (x, y).

Thus Prim's algorithm is correct!

How Fast is Prim's Algorithm?

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.

Prim's Implementation

To do it faster, we must identify fringe vertices and the minimum cost edge associated with it fast.

```
int prim(graph *g, int start) {
    int i;                  /* counter */
    edgenode *p;            /* temporary pointer */
    bool intree[MAXV+1];    /* is the vertex in the tree yet? */
    int distance[MAXV+1];   /* cost of adding to tree */
    int v;                  /* current vertex to process */
    int w;                  /* candidate next vertex */
    int dist;               /* cheapest cost to enlarge tree */
    int weight = 0;         /* tree weight */

    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;

    while (!intree[v]) {
```

```
        intree[v] = true;
        if (v != start) {
            printf("edge (%d,%d) in tree \n",parent[v],v);
            weight = weight + dist;
        }
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            if ((distance[w] > p->weight) && (!intree[w])) {
                distance[w] = p->weight;
                parent[w] = v;
            }
            p = p->next;
        }

        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++) {
            if ((!intree[i]) && (dist > distance[i])) {
                dist = distance[i];
                v = i;
            }
        }
    }

    return(weight);
}
```

Prim's Analysis

Finding the minimum weight fringe-edge takes $O(n)$ time, because we iterate through the *distance* array to find the minimum

After adding a vertex v to the tree, by running through its adjacency list in O(n) time we check whether it provides a cheaper way to connect its neighbors to the tree. If so, update the *distance* value.

The total time is $n \times O(n) = O(n^2)$.

Questions?

Topic: Kruskal's Algorithm

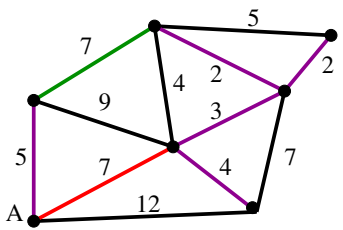
Kruskal's Algorithm

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 on dense graphs.

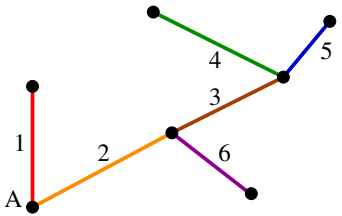
The complexity of Prim's algorithm is independent of the number of edges. Kruskal's algorithm is faster on sparse graphs

Kruskal's algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.

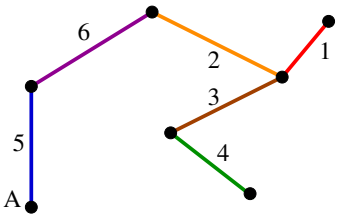
Kruskal's Algorithm in Action



G



Prim(G,A)



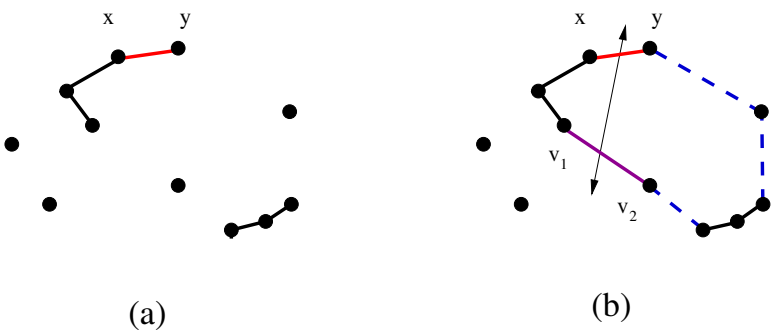
Kruskal(G)

Kruskal is Correct (Proof by Contradiction)

- If Kruskal's algorithm is not correct, these must be some graph G where it does not give the minimum cost spanning tree.
- If so, there must be a first edge (x, y) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.
- When we added (x, y) there no path between x and y, or it would have created a cycle. Thus adding (x, y) to the optimal tree it must create a cycle.
- But at least one edge in this cycle must have been added after (x, y), so it must have heavier.

The Contradiction

Deleting this heavy edge leaves a better MST than the optimal tree, yielding a contradiction!



(a) (b)

Thus Kruskal's algorithm is correct!

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?

Fast Component Tests Give Fast MST

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!

- *Same component*(v_1, v_2) – Do vertices v_1 and v_2 lie in the same connected component of the current graph?
- *Merge components*(C_1, C_2) – Merge the given pair of connected components into one component.

Fast Kruskal Implementation

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 $O(m \log m)$?

Questions?

Topic: The Union-Find Data Structure

Union-Find Programs

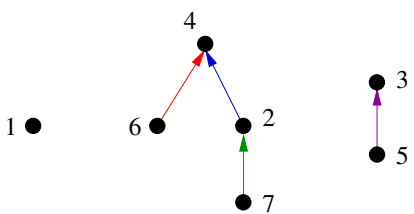
We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by *union* and *find* operations, where

- *Find(i)* – Return the label of the root of tree containing element i, by walking up the parent pointers until there is no where to go.
- *Union(i,j)* – Link the root of one of the trees (say containing i) to the root of the tree containing the other (say j) so $find(i)$ now equals $find(j)$.

Union-Find "Trees"

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:



(l)

1	2	3	4	5	6	7
1	4	3	4	3	4	2

(r)

Union-Find Data Structure

```
typedef struct {
    int p[SET_SIZE+1];      /* parent element */
    int size[SET_SIZE+1];   /* number of elements in subtree i */
    int n;                  /* number of elements in set */
} union_find;


void union_find_init(union_find *s, int n) {
    int i;    /* counter */

    for (i = 1; i <= n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }
    s->n = n;
}
```

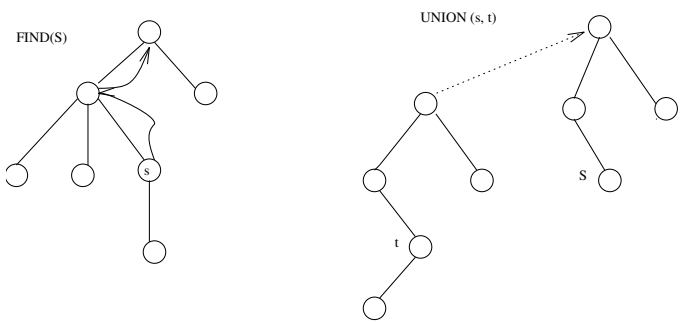
Worst Case for Union Find

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)

Who's The Daddy?

We want the limit the height of our trees which are affected by *union*'s.
When we union, we can make the tree with fewer nodes the child.



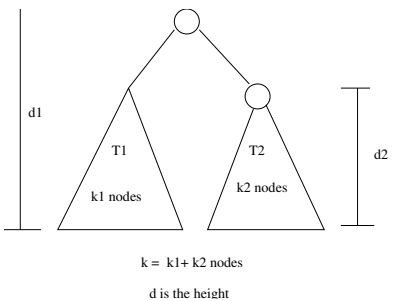
Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal

height!

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

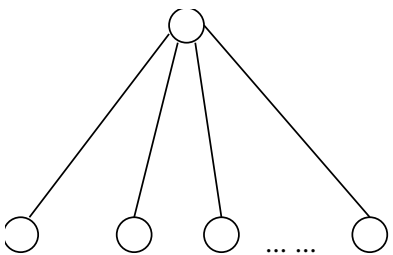
By induction on the number of nodes k, $k = 1$ has height 0.
Let d_i be the height of the tree t_i



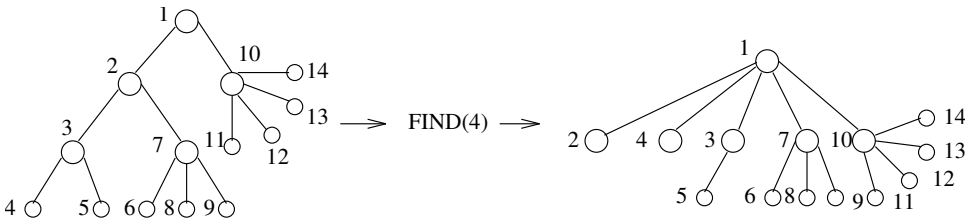
If $(d_1 > d_2)$ then $d = d_1 \leq \lfloor \log k_1 \rfloor \leq \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \log k \rfloor$
If $(d_1 \leq d_2)$, then $k_1 \geq k_2$.
$d = d_2 + 1 \leq \lfloor \log k_2 \rfloor + 1 = \lfloor \log 2k_2 \rfloor \leq \lfloor \log(k_1 + k_2) \rfloor = \log k$ ∎

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:



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 α(number of atoms in the universe)$= 5$.

Same Component Test

```
bool same_component(union_find *s, int s1, int s2) {
    return (find(s, s1) == find(s, s2));
}


int find(union_find *s, int x) {
    if (s->p[x] == x) {
        return(x);
    }
    return(find(s, s->p[x]));
}
```

Merge Components Operation

```
void union_sets(union_find *s, int s1, int s2) {
    int r1, r2;    /* roots of sets */

    r1 = find(s, s1);
    r2 = find(s, s2);

    if (r1 == r2) {
        return;    /* already in same set */
    }

    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
        s->p[r2] = r1;
    } else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[r1] = r2;
    }
}
```

Questions?