ICS 311 #17: Minimum Spanning Trees

Outline

Readings and Screencasts

- Required: CLRS 3rd Ed. Chapter 23, entire chapter.
- See also: Sedgewick (1984) Chapter 31 for light conceptual introduction (in Laulima), or Sedgewick & Wayne (2001) Algorithms Chapter 4 for code and application examples.
- Screencasts 17 A Intro, 17 B Kruskal, 17 C Prim.

Minimum Spanning Trees

Spanning Trees

A **spanning tree** T for a connected graph G is a tree that includes all the vertices of G: it *spans* the graph.

Without calling them such, we have already encountered two kinds of spanning trees in the introduction to graphs (Topic 14): those generated by breadth-first search and depth-first search. We saw that *breadth-first trees* are one way of finding shortest paths in a graph, and *depth-first forests* (a collection of spanning trees, one for each connected component) are good for uncovering the structure of a graph such as topological sort and connectivity. These were defined on unweighted graphs.

Minimum Spanning Trees

Many application areas (e.g., in communications, electronics, and transportation) require finding the lowest cost way to connect a set of objects or locations. For example, the cost may be measured in terms of currency or distance. We can model such situations with ***weighted graphs***, introduced in Topic 14 as graphs where a real-valued number is associated with each edge. Then we want to find a spanning tree of minimum cost.

More formally, we can pose this as a problem on a graph representation $G = (V, E)$:

- The objects or locations are vertices V and the available connections are edges E.
- A weight function $w(u,v)$ gives the weight on each edge $(u,v) \in E$.
- We seek $T \subseteq E$ such that
  - T connects all the vertices V of G.
  - The sum of weights $w(T) = \Sigma_{(u,v) \in T}\, w(u,v)$ is minimized.

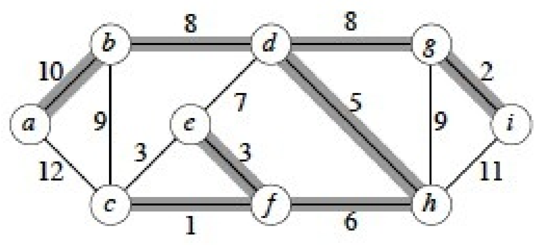
A few facts can be noted:

- G must be connected (consist of a single connected component) in order for T to be possible.

- However, if G is not connected we can generalize the problem to one of finding $T_1 \dots T_c$ for each of c connected components of G.
- A subgraph of G that connects its vertices V at minimal cost will always be a tree. *Why?*

Therefore we call this the **minimum spanning tree (MST)** problem (and the generalized version the minimum spanning forest problem).

Here is an example of a minimum spanning tree (the shaded edges represent T):



Are minimum spanning trees unique?

Look at edges (e,f) and (c,e).

Generic Algorithm and Safe Edge Theorem

We specify a generic greedy algorithm for solving the MST problem. The algorithm will be "greedy" in terms of always choosing a lowest cost edge. This algorithm is instantiated into two versions, Kruskal's and Prim's algorithms, which differ in how they define from what set of edges the lowest cost edge is chosen.

Let's start by noting some properties that MSTs of $G = (V, E)$ have

- A MST for G has $|V| - 1$ edges. (See properties of trees, Topic 8.)
- Any tree (and hence any MST) has no cycles. It has only one path between any two vertices.
- There might be more than one MST for G.

Building a Solution

- We will build a set of edges A.
- Initially A has no edges.
- As we add edges to A, we maintain a loop invariant: A is a subset of *some* MST for G.

Define an edge (u,v) to be **safe** for A iff $A \cup \{(u,v)\}$ is also a subset of some MST for G.

(BTW, "iff" is not a spelling error: it is shorthand for "if and only if" commonly used in proofs.

If we only add safe edges to A, once $|V| - 1$ edges have been added we have a MST for G. This motivates the following algorithm.

Generic MST Algorithm

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Loop Invariant: A is a subset of some MST for G

- *Initialization:* The initially empty set trivially satisfies the loop invariant.
- *Maintenance:* Since we add only safe edges, A remains a subset of some MST.
- *Termination:* We stop when A is a spanning tree ($|A| = |V| - 1$), and it is a subset of itself.

OK, great, but how do we find safe edges?

Finding Safe Edges

Each time we add an edge we are connecting two sets of vertices that were not previously connected by A. (Otherwise we would be forming a cycle.) A greedy algorithm might try to keep the cost down by choosing the lowest cost edge that connects previously unconnected vertices. (Perhaps we should call it a "stingy" algorithm!)

But is this greedy strategy "safe"? How do we know that after adding this edge we still have a subset of an MST?

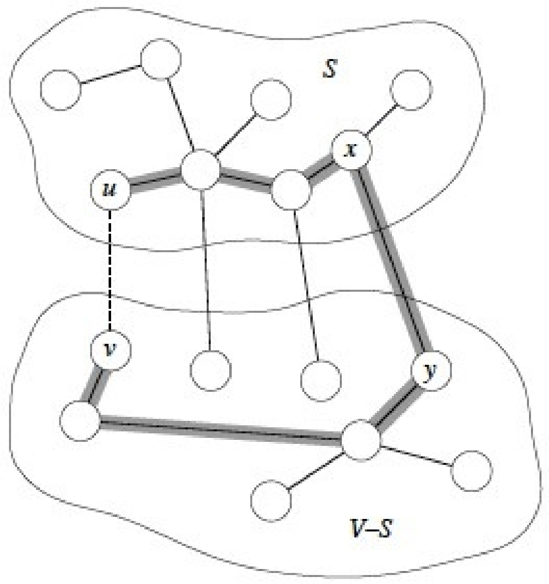
First some definitions:

- A **cut** $(S, V - S)$ is a partition of vertices into disjoint sets S and $V - S$.
- Edge $(u,v) \in E$ **crosses** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut **respects** A iff no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut iff its weight is minimum over all edges crossing the cut. (There may be more than one light edge for a given cut.)

The following figure illustrates a cut and will be used in the proof below. There are two sets of vertices S and $V - S$. Four edges cross the cut $(S, V - S)$. Whether or not this respects A depends on what is in A.

Suppose $A = \{(u,v)$ such that $u \in S$ and $v \in S\}$. Does the cut $(S, V - S)$ depicted in the figure respect A?

Now suppose A is the shaded edges in the figure. Does the cut $(S, V - S)$ respect A?



Safe Edge Theorem

Let $G = (V, E)$ be a graph, A be a subset of some MST for G, $(S, V - S)$ be a cut that respects A, and (u,v) be a light edge crossing $(S, V - S)$. Then (u,v) is safe for A.

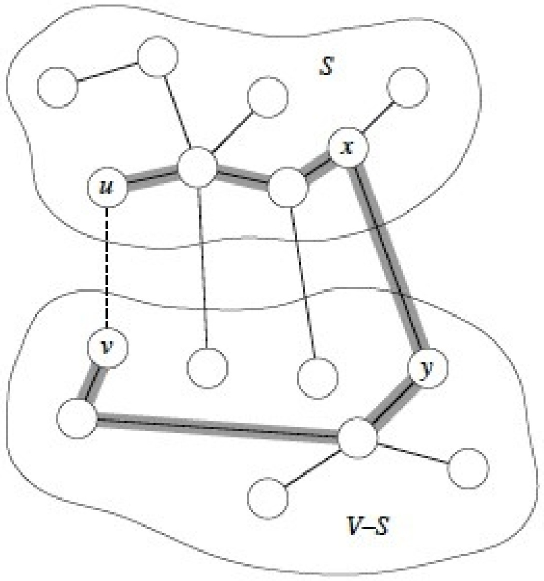
(*Paraphrased: A light edge that crosses a cut that respects A is safe for A.*)

Proof: Let T be a MST that includes A. Consider two cases:

Case 1: T contains (u,v). Then the theorem is proven, since $A \cup \{(u,v)\} \subseteq T$ is a subset of some MST for G.

Case 2: T does not contain (u,v). We will show that we can construct a tree T' that is a MST for G and that contains $A \cup \{(u,v)\}$.

Since T is a tree it contains a unique path p between u and v. Path p must cross the cut $(S, V - S)$ at least once (otherwise T would be disconnected). Let (x,y) be an edge of p that crosses the cut.



(Except for the dashed edge (u,v), all the edges shown in the figure are in T. A is not shown in the figure, but it cannot contain any edges that cross the cut, since the cut respects A. Shaded edges are the path p.)

Since the cut respects A, edge (x,y) is not in A.

To form T' from T: Remove (x,y). This breaks T into two components. Add (u,v). This reconnects the tree. So $T' = T - \{(x,y)\} \cup (u,v)$ is a spanning tree.

To show that T' is a minimal spanning tree: $w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$ since (u,v) is light.

We still need to show that (u,v) is safe for A. Since $A \subseteq T$ and $(x,y) \notin A$ then $A \subseteq T'$. Therefore $A \cup \{(u,v)\} \subseteq T'$, a MST. ♦

Further Observations

A is a forest containing connected components. Initially each component is a single vertex. Any safe edge merges two of these components into one. Each component so constructed is a tree. Since an MST has exactly $|V| - 1$ edges, the loop iterates $|V| - 1$ times before we are down to one component.

Corollary

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u,v) is a light edge connecting C to some other component C' in G_A -- that is, (u,v) is a light edge crossing the cut $(V_C, V - V_C)$ -- then (u,v) is safe for A.

Proof: Set $S = V_C$ in the theorem. ♦

This idea (of thinking in terms of components rather than vertices) leads to Kruskal's algorithm ...

Kruskal's Algorithm

Kruskal's algorithm starts with each vertex being its own component, and repeatedly merges two components into one by choosing the light edge that connects them. It does this greedily (or stingily?) by scanning the edges in increasing order by weight. A disjoint-set data structure is used to determine whether an edge connects vertices in two different components.

This algorithm has similarities with the connected components algorithm we previously saw in Topic 16:

```
CONNECTED-COMPONENTS(G)
1  for each vertex $v \in G.V$
2      MAKE-SET($v$)
3  for each edge $(u, v) \in G.E$
4      if FIND-SET($u$) $\neq$ FIND-SET($v$)
5          UNION($u, v$)
```

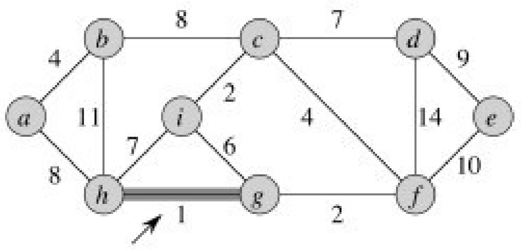
Here is Kruskal's version:

```
MST-KRUSKAL(G, w)
1  $A = \emptyset$
2  for each vertex $v \in G.V$
3      MAKE-SET($v$)
4  sort the edges of $G.E$ into nondecreasing order by weight $w$
5  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6      if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          $A = A \cup \{(u, v)\}$
8          UNION($u, v$)
9  return $A$
```

Example

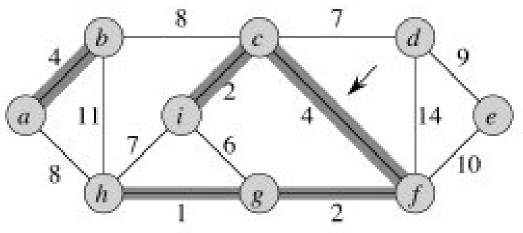
Let's start with this example. The first edge has been chosen.

(a)



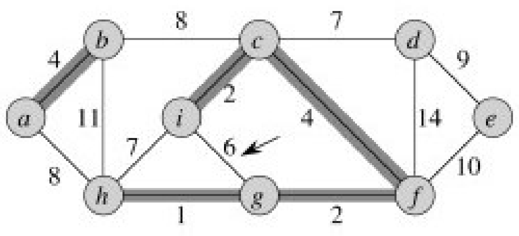
Add 4 more edges (notice we could add edges of weight 2 in either order, and similarly for 4) ...

(e)



The next edge considered is not added because it would connect already connected vertices:

(f)



Keep going until the MST is constructed, and click to see the final tree.

Analysis

```
MST-KRUSKAL($G, w$)
$A = \emptyset$
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($u$) $\neq$ FIND-SET($v$)
        $A = A \cup \{(u, v)\}$
        UNION($u, v$)
return $A$
```

The costs are:

- Initialize A: O(1)
- First `for` loop: $|V|$ `MAKE-SET` operations
- Sort E: O(E lg E)
- Second `for` loop: O(E) `FIND-SET`s and `UNION`s

If we use the tree implementation of the disjoint-set data structure with union by rank and path compression (Topic 16), the amortized cost per `MAKE-SET`, `UNION` and `FIND-SET` operation (across $|E|$ operations) is O($\alpha(V)$), where α is a *very* slowly growing function, the inverse of Ackermann's function. (Lemma 21.11 states that MAKE-SET in isolation is O(1), but here we must treat it as O($\alpha(V)$) since we are making a statement about the amortized cost per operation in a sequence of m operations: see section 24.1. Also, using O($\alpha(V)$) simplifies the expression below.)

Droping the lower order O(1) and substituting $\alpha(V)$ for the disjoint-set operations, the above list of costs sums to O(($V + E$)·$\alpha(V)$)+ O(E lg E).

Since G is connected, $|E| \geq |V| - 1$, so we can replace V with E to simplify the first term for the disjoint-set operations, O(($V + E$)·$\alpha(V)$), to O(($E + E$)·$\alpha(V)$) or O(E·$\alpha(V)$).

Furthermore, $\alpha(V)$ = O(lg V) = O(lg E), so O(E·$\alpha(V)$) is O(E lg E), and hence the entire expression we started with, O(($V + E$)·$\alpha(V)$)+ O(E lg E), simplifies to O(E lg E).

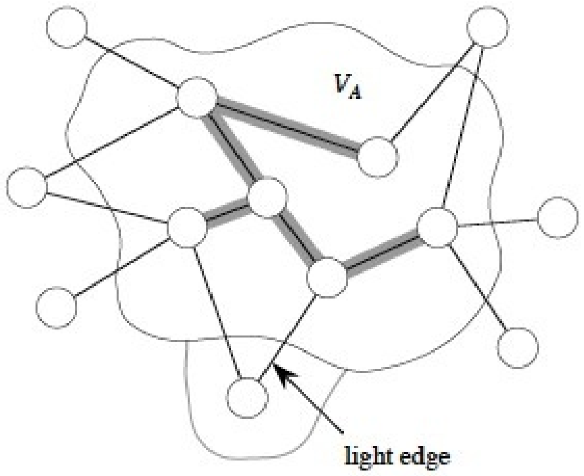
Finally, since $|E| \leq |V|^2$, lg $|E|$ = O(2 lg V) = O(lg V), so we can write the result as **O(E lg V)** to obtain the growth rate in terms of both $|E|$ and $|V|$.

(It is usually a good idea to include both V and E when giving growth rates for graph algorithms, unless one of them can be strictly limited to the other. Shortly we will see that O(E lg V) enables comparison to Prim's algorithm.)

Prim's Algorithm

This algorithm is also a greedy (stingy) algorithm, but it builds one tree, choosing the lightest edge incident on the growing tree, so the set A is always a tree. The tree is initialized to be a single vertex, designated r for root.

At each step it finds a light edge crossing the cut (V_A, V - V_A), where V_A= vertices that are incident on A, and adds this edge to A. (Initially, A = {} and V_A = {r}.)



(Edges of A are shaded in the illustration.)

General Idea

To find the light edge quickly we use a priority queue Q (Topic 09):

- Each queued object is a vertex in $V - V_A$ (the vertices that have not yet been connected to A).
- The key is the minimum weight of any edge (u,v) where $u \in V_A$. (We update this weight for v whenever a new edge is found that reaches v at a lower cost than before.)
- Thus the vertex returned by `EXTRACT-MIN` is for v such that $\exists\, u \in V_A$ and (u,v) is a light edge crossing $(V_A, V - V_A)$.
- If v is not adjacent to any vertices in V_A, the key of v is ∞

The edges of A will form a rooted tree with root r, given as input (r can be any vertex).

- Each vertex keeps track of its parent by the attribute $v.\pi$ = parent of v, or NIL if $v = r$ or has no parent yet.
- As the algorithm progresses, $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- At termination, Q is empty, so A is a MST.

Pseudocode

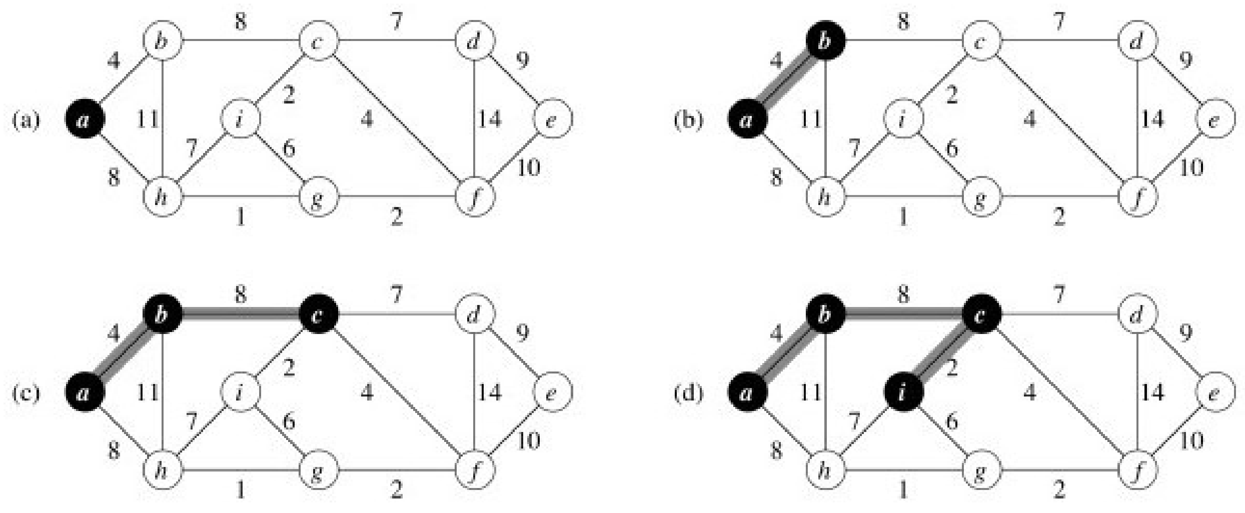
This code differs from the book's version in having explicit calls to the heap methods:

```
PRIM($G, w, r$)
$Q = \emptyset$
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
    INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)        // $r.key = 0$
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            DECREASE-KEY($Q, v, w(u, v)$)
```

Notice that it is possible for the last `if` to execute multiple times for a given *v*. In other words, we may find an edge reaching vertex *v*, but before we choose to use it (because other edges have lower key values), we find another edge reaching *v* for lower cost (key value). *Watch for this situation in the example below.*

Example

Let's try it with this graph. The first three steps are shown. Every time a vertex is dequed, it is colored black and the cost of all adjacent vertices are updated as needed. For example, when **a** is dequeued, the cost of **b** is updated from infinite to 4, and the cost of **h** is updated from infinite to 8. Then when **b** is dequeued, its neighbors are updated and so on.



Did you see where a vertex's key was lowered from one non-infinite value to another? Which one?

Now finish it and click on the image to see final solution.

Analysis

Performance depends on the priority queue implementation. With a binary heap implementation (Topic 09), the costs are:

- Initialize *Q* and iterate over |*V*| vertices in first `for` loop to insert in queue, each insert being O(lg *V*): O(*V* lg *V*) total.
- Decrease key of r: O(lg *V*)

- The `while` loop has |*V*| `EXTRACT-MIN` calls → O(*V* lg *V*)
- By aggregate analysis, the inner `for each` loop processes Θ(|*E*|) edges, O(*E*) of which result in O(lg *V*) `DECREASE-KEY` calls → O(*E* lg *V*)

The sum of the dominating terms is O(*V* lg *V*) + O(*E* lg *V*).

If G is connected, |*E*| ≥ |*V*| − 1, so we can replace O(*V* lg *V*) with O(*E* lg *V*), and the total is **O(*E* lg *V*)**.

This is asympotitically the same as Kruskal's algorithm. A faster implementation of O(*E* + *V* lg *V*) is possible with Fibonacci Heaps, as explained in the text.

```
PRIM($G, w, r$)
$Q = \emptyset$
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = $ NIL
    INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)        // $r.key = 0$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            DECREASE-KEY($Q, v, w(u, v)$)
```

If your graph is not connected, which algorithm would correctly produce a forest of minimum spanning trees, one for each connected component? How would you modify the other one to do this?

Dan Suthers
Last modified: Sun Apr 5 03:56:02 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.