Minimum Spanning Trees (14)

1 MINIMUM SPANNING TREES

1.1 Types of Graphs

We've been talking about graphs:

- directed acyclic graph: ordering constraints between a set of tasks
- directed graph: reachability relations between cities
- bipartite graph: compatibility relationships between two set of objects

weighted graph: add quantitative relationships between objects

1.2 Water For Everyone

Here's another problem we can model with graphs:

- We've got a set of nodes representing campus buildings. One of them is a pumping station.
- We've got edges between them representing possible places for building water pipes.
- We need to create a network of pipes that link all the buildings together, so we can supply all of them with water.
- Because of differences in the ground and distances, etc., there are different costs associated with digging pipes between different pairs of buildings: $w(u, v)$ is the cost for constructing a pipe between buildings u and v.

Our task is to dig a set of pipes so that:

- All buildings get water.
- Cost is minimized.

Answer?

1.3 Mathematical Abstraction

A *spanning tree* is a tree (i.e., a set of edges with no undirected cycles). How find one?

Depth first search! Return any edge for which the node on the other side is unmarked.

The *cost* of a spanning tree is the sum of the weights of the edges included in that spanning tree.
A *minimum spanning tree* is a spanning tree with the smallest possible cost.

1.4 Simple Verification

In many of the problems we've looked at, once you find the answer, it is easy to see that it is correct. What are the running times?

- sorting
- building a heap
- building a binary search tree
- finding a stable marriage

Think about these, great exam questions. sorting: $\Theta(n)$ because we just run through the list making sure no adjacent pair is out of order; building a heap: $\Theta(n)$ because we run through the list making sure that no kid is smaller than its parent; building a binary search tree: $\Theta(n)$ because we can do an in-order traversal and check that it is sorted as above; finding a stable marriage: $\Theta(n^2)$ since we have to check all the boy-girl pairs to see whether they'd rather switch to be together.

1.5 Optimization Verification

Optimization problems are trickier. Algorithms lean on powerful theorems to know when they're done.

- girl-optimal stable marriage
- maximum bipartite matching
- minimum spanning tree

Syntactic check, solve independently and compare. That is, checking is no easier than solving from scratch.

2 FINDING MINIMUM SPANNING TREES

2.1 Design Ideas

Rule #1 of Good Algorithm Design: If you already have a good subroutine to solve the problem, use it!
Rule #2 of Good Algorithm Design: Divide-and-Conquer!
Rule #3 of Good Algorithm Design: Keep complex information organized in a good data structure.
Rule #4 of Good Algorithm Design: Be greedy.

2.2 Greedy Approach

Keep adding low-cost edges as long as no cycle is made.
Didn't work for matching. Works here.
GENERIC-MST(G)
1 $T \leftarrow \emptyset$
2 **for** $i \leftarrow 1$ **to** $|G[V]| - 1$
3 **do** Define a cut C s.t. $C \cap T = \emptyset$
4 $e^* \leftarrow \text{argmin}_{e \in C}\, w(e)$
5 $T \leftarrow T \cup \{e^*\}$
6 **return** T

2.3 Cuts

A *cut* is something that separates a graph into two pieces. Two equivalent ways of defining a cut:

- Two sets of nodes that partition V.
- The set of edges that go between two sets of nodes that partition V.

Example...

2.4 Greedy-Grow Lemma

Lemma: If T is a subset of *some* e MST, and C is some cut that doesn't share any edges with T, then there is some MST containing T and the minimum cost edge e^* in C.
Proof: Very standard trick. Let T^* be the MST that contains T. If $e^* \in T^*$, we're done. If not, consider the set of edges $T^* \cup \{e^*\}$. It must have a cycle including e^* and some other edge e (maybe more than one) in C. By definition, $w(e^*) \leq w(e)$. Delete e from T^*.

- Is the resulting set $T^* \cup \{e^*\} - \{e\}$ still a spanning tree?

  Yes. "Electricity" can still flow between all nodes along that cycle.

- What is its cost compared to T^*?

  No larger because we traded a more costly edge for a cheaper one.

2.5 Correctness of GENERIC-MST

The correctness of GENERIC-MST follows from the Greedy-Grow Lemma.

- Proof by induction. The initial set $T = \emptyset$ is a subset of any MST.
- At each step, GENERIC-MST adds an edge that is necessarily part of some MST (by the Greedy-Grow Lemma).
- So, when the set T is a spanning tree, it must be an MST (it is a subset of some MST, itself).

2.6 Efficient Implementation

The GENERIC-MST algorithm and the greedy-grow lemma give some guidance for how to find MSTs simply, but they don't immediately suggest efficient algorithms. How do we quickly find cuts and the smallest edges they have?
There are two classic approaches:

- Prim: Related to shortest path (next lecture).

  Pick a point and grow MST from it. Cut is T vs. $V - E$.

- Kruskal: Related to sorting and "union-find."

  Sort the edges. Continually add the smallest edge that doesn't create a cycle. Cut is defined implicitly by the smallest edge and the disconnected pieces of T.

3 KRUSKAL'S ALGORITHM

3.1 Recap

Start with a weighted graph $G = (V, E, w)$.
Want to select a subset of edges that interconnect all nodes with the smallest possible total weight.
The "greedy-grow lemma" is our main theoretical tool for analyzing algorithms for solving this problem.

3.2 Algorithm

There's an algorithm for doing this.

KRUSKAL-MST(G, w)
1 $T \leftarrow \emptyset$
2 $S \leftarrow$ SORT$(G[E], w)$
3 $Q \leftarrow$ DISCONNECT$(G[V])$
4 **for each** $(u, v) \in S$
5 **do if** NOT(CONNECTED(Q, u, v))
6 **then** $Q \leftarrow$ JOIN(Q, u, v)
7 $T \leftarrow T \cup \{(u, v)\}$
8 **return** T

3.3 Correctness

Theorem: KRUSKAL-MST returns a minimum spanning tree.
Proof: By induction ("for" loop).

- Inductive hypothesis is that T is always a subset of an MST. Since it eventually contains an entire spanning tree, the final result is a minimum spanning tree.
- For the base case, initially, T is empty, and therefore a subset of an MST. Now, consider the processing of an edge (u, v). If u and v are already connected in T, then it can't be part of any MST that contains T, and we toss it out.
- Let's say u and v are not connected. Consider the cut with all the nodes connected to u by T on one side and all other nodes on the other side. This is a cut that doesn't intersect T.
- By the greedy-grow lemma, the smallest edge across the cut can be added to T maintaining the property that T is contained in an MST.
- **Claim:** (u, v) is the minimum weight edge in the cut. Why?

  All smaller edges are either in T (and therefore not in the cut), or would cause a loop with T (and therefore not in the cut since u and v are disconnected).

- Therefore, (u, v) can be added to T en route to an MST.

3.4 Implementation

We need to be able to test u-v connectivity quickly. Three operations unspecified:

- DISCONNECT(V)
- CONNECTED(Q, u, v)

- JOIN(Q, u, v)

How many times are each executed? Total running time?

> Disconnect: once. CONNECTED: $|E|$, since we run through the list of edges and test each one. JOIN: $|V| - 1$, since we call this once on each edge added to the MST. Running time is $O(|E|\log(|E|) + d + |E|c + |V|j)$.

How do we implement these to minimize running time?

3.5 Simple Implementation: DFS

All the work done in CONNECTED. Let JOIN just add the given edge to a set of edges. Now, can use depth-first-search to test connectivity when we need to. Basically, if we start at u and can reach v, they are connected. If not, not.
That gives us: $\Theta(|V| + |E|)|E| = O(|E|^2)$ for all the calls to CONNECTED and JOIN. This dominates that $O(|E|\log|E|)$ for sorting the edges, giving $O(|E|^2)$ running time.
Explanation: for each edge that we consider adding, we do a depth-first search over edges to see if the endpoints are already connected.

3.6 Better Implementation

We can make CONNECTED run fast by precomputing which nodes are connected to which other nodes. Could do this with a $|V| \times |V|$ array, but this is more than is needed. Can do this just an array of size $|V|$. How?

> Give each node a "color" such that two nodes are connected if and only if they have the same color.

Now, all the work will be in JOIN to update who is connected to whom.
So, what ought to happen when join to nodes?

> All the nodes that share their colors must be set to the same color.

We could relabel all the nodes, but this could be expensive. What's the cheap way to make them the same color?

> Color the smaller group with the colors of the larger group.

But, to do this, we'll need to be able to quickly run through all nodes of the same color. How can we do this?

> Keep a linked list running through the like colored nodes. Easy to update (append) when lists are merged.

3.7 Implementation

$C[v]$ is the color of nodes in v's group. $L[c]$ is a linked list of nodes for color c.

```
JOIN(C, L, u, v)
1  if size(L[C[u]]) < size(L[C[v]])
2     then u ↔ v
3  for  each w ∈ L[C[v]]
4       do C[w] ← C[u]
5          append(w, L[C[u]])
6  return (C, L)
```

3.8 Running Time: Bound One

Now, CONNECTED is constant time and we repeat it $|E|$ times.
JOIN is repeated $|V| - 1$ times and can take up to $|V|/2$ time.
Total running time: $O(|E| \log |E| + |V|^2)$.
Too loose! We're not really getting much bang from the fact that we merge smaller sets into larger sets.

3.9 Running Time: Tighter Bound

Informal potential function analysis

- All the work is in relabeling nodes. How many times can a node u be relabeled? In particular, if u is in a set of size k, how many times could it have been relabeled?

  Each time it is relabeled, it joins a set at least twice as big as its old set. Thus, if u is in a set of size k, it can only have been relabeled $\log_2 k$ times!

- The final set size is $|V|$, so how many times can a node be relabeled?

  $\log_2(|V|)$.

- Thus, $\Theta(|V| \log |V|)$ time for all calls to JOIN, since all $|V|$ nodes end up in sets of size $|V|$. This is dominated by the time to sort, so no further improvements to JOIN and CONNECTED will help!

Total: SORT plus JOIN is $O(|E| \log |E|) = O(|E| \log |V|)$.

4 FAST UNION-FIND

4.1 Quick Aside

In the version of Kruskal described above, sorting is the bottleneck.
But, some applications use CONNECTED and JOIN for other things, so it's worth making them fast.

4.2 Clever Data Structure

Can represent the set of groups with a new DAG with one node per vertex from the original weighted graph. Each node points to another vertex in its set, with the "root" being the set's label.
Now, to join u and v, we simply make the root of the smaller set (say v) point to u.
Checking to see if u and v are connected amounts to finding the roots of each DAG to see if they are the same.
As an additional heuristic, when we follow a chain to a root, we do *path compression*: take a second pass through the list and make all the nodes point directly at the root, resulting in a faster test next time. (Remind you of anything?)

> It's a little like the reorganization in a splay tree.

4.3 Culture

I bring this up because it lets me tell you a nice little piece of CS culture.
The running time of a sequence of m JOINs (unions) and CONNECTEDs (finds) implemented this way on n objects is $O((n+m)\alpha(n+m))$.
Here, $\alpha(x)$ is the pseudoinverse of Ackermann's function. It's basically the world's slowest growing function. It's almost impossible to imagine an x large enough so that $\alpha(x) > 4$.
Big improvement over the $O((n+m)\log(n+m))$ data structure I gave earlier! Linear, for all intents and purposes.