Computing Minimum Spanning Trees (19)

Michael L. Littman

November 13th, 1997

1 ANNOUNCEMENTS

Exam 2 coming up... November 20th.
Homework deadlines extended.
Tuesday plan: finish today's lecture, review material for exam, snacks.
Action items: What kind of snacks? Bring questions, answers.

2 KRUSKAL'S ALGORITHM

2.1 Recall

Start with a weighted graph $G = (V, E)$.
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.
Today: Kruskal and Prim.
Example.

2.2 Our Algorithm

We devised an algorithm for doing this.

```
Kruskal-MST(G, w)
T ← ∅
S ← Sort(G[E], w)
Q ← Disconnect(G[V])
for each (u, v) ∈ S
    do if not(Connected(Q, u, v))
          then Q ← Join(Q, u, v)
               T ← T ∪ {(u, v)}
return T
```

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

2.4 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?
How do we implement these to minimize running time?

2.5 Simple Implementation

All the work done in CONNECTED. Can use depth-first-search to test connectivity. 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 the inner loop. This dominates that $O(|E| \log |E|)$ for sorting the edges.
Explanation: for each edge that we consider adding, we do a depth-first search over edges to see if the endpoints are already connected.

2.6 Better Implementation

Spend more time when an edge is added (Join) to make it easier to figure out what's connected (Connected).
Idea: Keep a linked list for each connected group of nodes and a pointer from each node to its list. Each time two groups are merged (a new edge is added), merge one list into the other, and change the pointers of the other to point to the one.
$Q[v]$ is a pointer to a linked list of nodes in v's group.
Example.
Which group to relabel to make this fast?

2.7 Implementation

Join(Q, u, v)
if size($Q[v]$) $>$ size($Q[u]$)
  then $u \leftrightarrow v$
$Q[u] \leftarrow Q[u] \cup Q[v]$
for each $w \in Q[v]$
  do $Q[w] \leftarrow Q[u]$
return Q

2.8 Running Time

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.
Too loose!

- Simple amortized analysis: Work is in relabeling nodes. How many times can a node u be relabeled?
- Each time it is relabeled, it joins a set at least twice as big as its old set. If it relabeled x times, what is the smallest set it could possibly be in?
- The final set size is $|V|$, so how many times can a node be relabeled?

Thus, $|V| \log |V|$ for all calls to Join.
Total: Sort plus Join is $O(|E| \log |E|) = O(|E| \log |V|)$.

3 FAST UNION-FIND

3.1 Quick Aside

In the version of Kruskal described above, sorting is the bottleneck.
The calls to Connected and Join use the same amount of time, asymptotically, as the sort, so there is no real push to improve them.

But, some applications use CONNECTED and JOIN for other things, so it's worth making them fast.

3.2 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?)

3.3 Culture

I bring this up because it let's 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.

4 PRIM'S ALGORITHM

4.1 Invent Another Algorithm

Did so well inventing Kruskal's algorithm, let's see if you devise Prim's.
Hint: You must add the next edge spreading out from the source node.
With the right implementation, Prim's algorithm is faster than Kruskal for dense graphs and very sparse graphs.

4.2 Idea

Very simple idea:

- Initially, all vertices are on the far side of the cut.
- Pick one r to be on near side and check the distance of r to all its neighbors.
- From the set of vertices on the far side, pick the one u that is reachable from the near side over the smallest edge.

- Move u to the near side. Update the distances of u to its neighbors on the far side to reflect the edges in the cut.
- Repeat.

Example...

4.3 Algorithm

```
PRIM-MST(G, w)
Q ← ∅
for each u ∈ V[G]
    do INITIALIZE-KEY[u] ← ∞
       Q ← Q ∪ {u}
r ← some vertex of G[V]
edge[r] ← NIL
DECREASE-KEY(r, 0)
T ← ∅
while Q ≠ ∅
    do u ← REMOVE-SMALLEST-KEY(Q)
       T ← T ∪ {edge[u]}
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then DECREASE-KEY(v, w(u, v))
                      edge[u] ← (u, v)
```

4.4 Running-Time Analysis

Three basic operations left to specify:

- INITIALIZE-KEY
- DECREASE-KEY
- REMOVE-SMALLEST-KEY

How often do we do each? What does this remind you of?
Simple data structure: list of nodes Q with their keys. Running time per operation? $O(1)$, $O(1)$, $O(|V|)$.
Total running time: $O(|E| + |V|^2) = O(|V|^2)$

4.5 Better Data Structure

The simple data structure yields a linear-time algorithm for dense graphs, $|E| = \Theta(|V|^2)$.
Can do better for sparse graphs.
Heap: Running time per operation? $O(\log|V|)$, $O(\log|V|)$, $O(\log|V|)$.
Total running time: $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$
Worse for dense graphs, better for sparse graphs.

4.6 Even Better Data Structure

"Fibonacci heaps" have a good amortized running time for DECREASE-KEY: $O(\log|V|)$, $O(1)$, $O(\log|V|)$.
Total running time: $O(|E| + |V|\log|V|)$.
Probably wouldn't really ever want to use this, but it is an asymptotic improvement.

5 HOMEWORK

5.1 Homework 9

Background: Ch. 23.1, 23.4, 23.5, 27.2, 27.3
Due November 14th:

- 9.1: CLR Exercise 23.1-3 (pg. 468).
- 9.2: Extra credit. CLR Exercise 23.1-6 (pg. 468).
- 9.3: CLR Exercise 23.4-1 (pg. 487).
- 9.4: CLR Exercise 23.5-1 (pg. 494).
- 9.5: CLR Exercise 23.5-5 (pg. 494).
- 9.6: Extra credit. In class, we came up with an alternate algorithm for strongly connected components. The algorithm went something like this: Start with a graph. Find a cycle in the graph. If there is no cycle, we're done. If there is, mark all nodes in the cycle as being in the same strongly connected component. Now collapse all those nodes into a single node and repeat. Describe an efficient implementation of this procedure and give its running time.
- 9.7: Give a $O(|V| + |E|)$ (linear time) algorithm for maximal (not maximum) bipartite matching.
- 9.8: Give an example bipartite graph with $2k$ left nodes and $2k$ right nodes with a maximal matching of size k and a maximum matching of size $2k$.

- 9.9: Extra credit. In the previous question, the ratio of the size of the maximum matching to the maximal matching is 2:1. Prove that this is the largest possible ratio. We can conclude from this that a maximal matching is an approximation (to within a factor of 2) of the maximum matching.
- 9.10: Book gives a $O(|V|^3)$ algorithm for network flow. When applied to bipartite matching, is this better or worse than the $O(|V||E|)$ algorithm we discussed?

5.2 Homework 10

Background: Ch. 24.1, 24.2.
Due November 25th:

- 10.1: Given a graph $G = (E, V)$, let e be the second smallest edge in G according to the weight function w. Argue that there is an MST containing e.
- 10.2: Argue that the choice of initial node r in PRIM-MST effects the final tree returned by constructing an example in which two different choices of r lead to two different trees. Argue that, ultimately, the choice is immaterial in that it doesn't effect the cost of the final tree.
- 10.3: Give tight asymptotic ranges of the number of edges for which a Fibonacci-heap-based implementation of Prim's algorithm is asymptotically better than the heap or simple list-based data structure. Your answer should have the form $|E| = \omega(f(|V|))$, $|E| = o(g(|V|))$.
- 10.4: CLR Exercise 24.1-8 (pg. 503). Extra credit.
- 10.5: CLR Exercise 24.2-6 (pg. 510). Hint: You may use the result of the extra credit problem, even if you don't solve it yourself.