IE170: Algorithms in Systems Engineering: Lecture 19

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

March 16, 2007

Taking Stock

Last Time

- Minimum Spanning Trees

This Time

- More Spanning Trees
- Strongly Connected Components

Spanning Tree

- We model the problem as a graph problem.
- $G = (V, E)$ is an undirected graph
- Weights $w : E \to \mathbb{R}^{|E|}$
  - $w_{uv}\ \forall (u, v) \in E$
- Find $T \subset E$ such that
  1. T connects all vertices
  2. The weight
  $$w(T) \stackrel{\text{def}}{=} \sum_{(u,v)\in T} w_{uv}$$
  is minimized

Kruskal's Algorithm

1. Start with each vertex being its own component
2. Merge two components into one by choosing the light edge that connects them
3. Scans the set of edges in increasing order of weight
4. It uses an abstract "disjoint sets" data structure to determine if an edge connects different vertices in different sets.
5. We used Java Collections Classes

Kruskal's Algorithm

```
KRUSKAL(V, E, w)
A ← ∅
for each v in V
do MAKE-SET(v)
SORT(E, w)
for each (u, v) in (sorted) E
do if FIND-SET(u) ≠ FIND-SET(v)
      then A ← A ∪ {(u, v)}
           UNION(u, v)return A
```


Analysis

- Let $\mathcal{T}(\mathcal{X})$ be the running time of the method $\mathcal{X}$

Task	**Running Time**
Initialize A	$O(1)$
First for loop	$\lvert V \rvert \mathcal{T}(\text{MAKE-SET})$
Sort E	$O(E \lg E)$
Second for loop	$O(E)(\mathcal{T}(\text{FIND-SET} + \text{UNION}))$

We Skipped That Chapter!

- If we use a clever data structure for FIND-SET and UNION, the running time can go to $\alpha(m, n)$, where m is the total number of operations, and n is the number of unions.
- $\alpha(m, n)$ is the inverse of the Ackerman function, which is a slowly growing function.
- $\alpha(m, n) \leq 4$ for all practical purposes
- In this case, we have that the operations take $\alpha(|E|, |V|)$

Kruskal Analysis

- Also, you should know that $\alpha(|E|, |V|) = O(\lg |V|)$
- Finally, not that $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg |V|) = O(\lg |V|)$
- Therefore the running time for Kruskal's Algorithm is $O(|E| \lg |V|)$.
- If the edges are already sorted, it runs in $O(|E|\alpha(|E|, |V|))$, which is essentially linear

Prim's Algorithm

- Builds one tree, so A is always a tree
- Let V_A be the set of vertices on which A is incident
- Start from an arbitrary root r
- At each step find a light edge crossing the cut $(V_A, V \setminus V_A)$

Main Question for Prim..

How do we find a light edge crossing the cut quickly?

Prim's Algorithm

The Answer!

- Use a priority queue!
- We built a priority queue in Lab 4. heaps are priority queues

- Each object in the queue is a vertex in $V \setminus V_A$ (A vertex that might be linked to our MST)
- The key of v is the minimum weight of any edge (u, v) such that $u \in V_A$.
- The key of v is ∞ if v is not adjacent to any vertices in V_A

Prim's Algorithm

- Prim's Algorithm starts from an (arbitrary) vertex (the root r)
- It keeps track of the parent $\pi[v]$ of every vertex v. ($\pi[r] =$ NIL).
- As the algorithm processes $A = \{(v, \pi[v]) \mid v \in V \setminus \{r\} \setminus Q\}$
- At termination, $V_A = V \Rightarrow Q = \emptyset$, so MST is

$$A = \{(v, \pi[v]) \mid v \in V \setminus \{r\}\}.$$

Pseudocode for Prim

```
PRIM(V, E, w, r)
  Q ← ∅
  for each u ∈ V
  do key[u] ← ∞
     π[u] ← NILINSERT(Q, u)
     key[r] = 0
  while Q ≠ ∅
  do u ← EXTRACT-MIN(Q)
     for each v ∈ Adj[u]
     do if v ∈ Q and w_uv < key[v]
           then π[v] ← u
                key[v] = w_uv
```

Demo Time!

- Udom and I wrote some code so that we can display our graphs.

Strongly Connected Components

- Given a directed graph $G = (V, E)$, a strongly connected component of G is a maximal set of vertices $C \subseteq V$ such that $\forall u, v, \in C$ there exists a directed path both from u to b and from v to u
- The algorithm uses the transpose of a directed graph $G = (V, E)$, where the orientations are flipped:

$$G^T = (V, E^T), \text{ where } E^T = \{(v, u) \mid (u, v) \in E\}$$

- What is running time to create G^T?
- Note: G and G^T have the same Strongly Connected Components
  - u and v are both reachable from each other in G if and only if they are both reachable with the orientations flipped (G^T).

Finding Strongly Connected Components

1. Call $\text{DFS}(G)$ to topologically sort G
2. Compute G^T
3. Call $\text{DFS}(G^T)$ but consider vertices in topologically sorted order (from G)
4. Vertices in each tree of depth-first forest for SCC

Component Graph

- $G^{\text{SCC}}(G) = (V^{\text{SCC}}, E^{\text{SCC}})$
- V^{SCC} has one vertex for each strongly connected component of G
- $e \in E^{\text{SCC}}$ is there is an edge between corresponding SCC's in G

Lemma

G^{SCC} is a DAG

- For $C \subseteq V$, $f(C) \stackrel{\text{def}}{=} \max_{v \in C}\{f[v]\}$

More Lemma

Lemma

Let C and C' be distinct SCC in G,

- if $(u,v) \in E$ and $u \in C, v \in C'$, then $f(C) > f(C')$
- if $(u,v) \in E^T$ and $u \in C, v \in C'$, then $f(C) < f(C')$
- If $f(C) > f(C')$, there is no edge from C to C' in G^T

Why SCC Works. (Intuition)

- DFS on G^T starts with SCC C such that $f(C)$ is maximum. Since $f(C) > f(C')$, there are no edges from C to C' in G^T
- This means that the DFS will visit only vertices in C
- The next root has the largest $f(C')$ for all $C' \neq C$. DFS visits all vertices in C', and any other edges must go to C, which we have already visited..

Next Time

- Shortest Paths