CS 341: ALGORITHMS

Lecture 17: graphs – minimum spanning trees, single-source shortest paths

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca

(http://tbrown.pro)
MINIMUM SPANNING TREE
Problem can also be defined for directed graphs…
Problem can also be defined for minimum spanning forest. Algorithm taught here works.
If you add an edge $e$ to a tree and this creates a cycle $C$, then removing any other edge $e' \in C$ will break the cycle and produce a tree.

Why would we compute an MST? Let’s see some applications…
APPLICATION: INTERNET BACKBONE PLANNING

- Want to connect n cities with internet backbone links
  - Direct links possible between each pair of cities
  - Each link has a certain dollar cost (excavation, materials, distance & time, legal costs...)
- Want to **minimize total cost**
APPLICATION: IMAGE SEGMENTATION

break image into regions by colour similarity via other techniques

turn regions into nodes, and add edges between them with weights = “dissimilarity,” then build MST

break MST into large, highly similar segments, and assign the dominant colour to each segment

Segments are easier for a machine learning algorithm to understand.

Just for fun, don’t need to know this
APPLICATION: CURVILINEAR FEATURE EXTRACTION

Want a machine to **recognize** this object

Edge detection algorithm

MST

Input to image recognition alg.

Final result

Just for fun, don’t need to know this

"Hair" removal

**Paper**
HOW TO ACTUALLY BUILD AN MST?

• Kruskal’s algorithm [introduced in this 3-page paper from 1955]

Assume that \( w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m) \), where \( m = |E| \).

Algorithm: \( Kruskal(G, w) \)

\[
A \leftarrow \emptyset \\
\text{for } j \leftarrow 1 \text{ to } m \\
\quad \text{do } \begin{cases} 
\quad \text{if } A \cup \{e_j\} \text{ does not contain a cycle} \\
\quad \text{then } A \leftarrow A \cup \{e_j\}
\end{cases}
\]

return \( (A) \)

Suppose we have an oracle \( \text{wouldCreateCycle}(e_j) \)
Increasing edge weights: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

8 would create a cycle: a, c, b, d, a
11 would create a cycle: d, e, b, d
14 would create a cycle: c, f, e, b, d, a, c
15 would create a cycle: g, f, h, j, l, g
16 would create a cycle...
17 would create a cycle...
18 would create a cycle...
19 would create a cycle...
20 would create a cycle...

Done!
Implementation Details for Kruskal’s Algorithm

We use a **union-find** data structure to determine if an edge $uv$ has vertices in two different trees.

Every tree $T$ will contain a **leader vertex**.

To find the leader vertex from a vertex $v$, we use an auxiliary array $L$.

From $v$, follow a directed path $v \rightarrow L[v] \rightarrow L[L[v]] \cdots$ until we reach a vertex $w$ with $L[w] = w$; then $w = \text{find}(v)$ is the leader vertex for the tree containing $v$.

Two vertices $u$ and $v$ are in the same tree if and only if $\text{find}(u) = \text{find}(v)$.

Initially, there are $n$ one-vertex trees and $L[v] = v$ for all $v$.

When we use an edge $uv$ to merge two trees, we perform the following **union** operation:

- $u' \leftarrow \text{find}(u)$
- $v' \leftarrow \text{find}(v)$
- $L[u'] \leftarrow v'$.
Suppose we also keep track of the depth of each tree. In step 3, we always take $u'$ to be the leader of the tree having smaller depth.

If we merge two trees of depth $d$, we get a tree of depth $d + 1$. If we merge a tree of depth $d$ and one of depth $< d$, we have a tree of depth $d$.

Then union and find each run in $O(\log n)$ time (this is because a tree of depth $d$ has at least $2^d$ vertices, a fact that can be proven by induction on $d$).

This leads to an algorithm for MST having complexity $O(m \log n)$ (the pre-sort has complexity $O(m \log m)$, and the iterative part of the algorithm has complexity $O(m \log n)$).

Note: $O(m \log m) = 0(m \log n)$.
Why? $m \log m \in O(m \log^2 n) = O(m \cdot 2\log n) = O(m \log n)$. This is called union by rank
We can sort in $O(m)$ time using a non-comparison sort such as 
radix sort or counting sort.

In addition to union by rank, union-find can be 
implemented with path compression.

Using both union by rank and path compression, we get a total 
running time for Kruskal’s algorithm of $O(\alpha(m + n)(m + n))$, 
where $\alpha(x)$ is the inverse Ackermann function. 
For all practical $x$, $\alpha(x) \leq 5$, so this is pseudo-linear.
Kruskal(V[1..n], E[1..m])
1. sort E[1..m] in increasing order by weight
2. uf = new UnionFind data structure
3. mst = new List
4. for j = 1..m
5.     set_a = uf.find(E[j].source)
6.     set_b = uf.find(E[j].target)
7.     if set_a != set_b
8.         mst.add(E[j])
9.         uf.merge(set_a, set_b)
10. return mst

Radix sort
Union-Find with path compression and union by rank
Is this hard to implement? No!
```cpp
class UnionFind {
    int * parent;
    int * rank;

    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i=0; i<n; i++) {
            rank[i] = 0;
            parent[i] = i;
        }
    }

    ~UnionFind() {
        delete[] parent;
        delete[] rank;
    }

    int find(int u) {
        if (u != parent[u]) parent[u] = find(parent[u]);
        return parent[u];
    }

    void merge(int x, int y) {
        x = find(x), y = find(y);
        if (rank[x] > rank[y]) parent[y] = x;
        else parent[x] = y;
        if (rank[x] == rank[y]) rank[y]++;
    }
};
```

**Initialization**

**Free memory at end**

**Path compression**

**Union by rank**
Suppose \( K \) is not an MST, for contradiction. Let \( O \) be an (optimal) MST. Note \( O \neq K \).

Label edges so \( w(f_1) < w(f_2) < \cdots < w(f_{n-1}) \).
(we prove this for distinct weights)

Adding \( f_j \) to \( O \) would create cycle \( C \)

Let \( e' = \text{smallest edge in } C \setminus K \)
(exists since no cycles in \( K \))

Kruskal considers \( e' \) before \( f_j \), and rejects \( e' \) despite taking \( f_1, \ldots, f_{j-1} \)

But \( f_1, \ldots, f_{j-1}, e' \in O \). Contradiction!
OTHER NOTABLE MST ALGORITHMS

• Prim’s algorithm
  • Incrementally extend a tree $T$ into an MST, by:
  • Initializing $T$ to contain any arbitrary node in $G$
  • Repeatedly selecting the smallest weight edge from any node in $T$ to any node outside of $T$

• Borůvka’s algorithm
  • Like Kruskal (merging components), but with phases
  • In each phase, select an outgoing edge for every component, and add all edges found in the phase

There is also a fast parallel hybrid of Prim and Borůvka
DIJKSTRA’S ALGORITHM

Single-source shortest path in a graph with non-negative edge weights
PROBLEM: SINGLE SOURCE SHORTEST PATHS (SSSP)

- Input: graph $G = (V, E)$ and a non-negative weight function $w(e)$ defined for every edge $e$
- Problem: for every node $v \neq s_0$, output a path $s_0 \rightarrow v$ with the smallest total weight (among all paths $s_0 \rightarrow v$)
- I.e., each path $P$ should minimize $w(P) = \sum_{e \in P} w(e)$

Let's study directed $G$.
Can also be defined for undirected $G$...

"Shortest" means minimum weight

And so on... one path for each node.
Dijkstra’s Algorithm

- Iteratively construct a set $S$ of nodes for which we know the shortest path from $s_0$ (initially $S = \{s_0\}$).
- Maintain a distance $D[v]$ for each node $v$:
  - If $v \in S$ then $D[v] = \text{weight of the shortest path } s_0 \rightarrow v$
  - Otherwise, $D[v] = \text{weight of the shortest path } s_0 \rightarrow v$ such that all interior nodes on the path are in $S$
  - If there is no such path then $D[v] = \infty$
- Grow $S$ by adding the $v \notin S$ that has the smallest $D[v]$ value.

Why can we add this $v$ to $S$? Do we know the shortest path $s_0 \rightarrow v$?

Suppose $D[v]$ is correct for all $v$ before adding $v$ to $S$. We show this holds after...
Lemma 6.12

Suppose \( v \) has the smallest \( D \)-value of any vertex not in \( S \). Then \( D[v] \) equals the weight of the shortest path \( s_0 \rightarrow v \), which we denote \( P \).

Proof.

Suppose there is a path \( s_0 \rightarrow v' \) with weight less than \( D[v] \). Let \( v' \) be the first vertex of \( P' \) not in \( S \). Observe that \( v' \neq v \). Decompose \( P' \) into two paths: a path \( s_0 \rightarrow v' \) \( P_1 \) and a path \( v' \rightarrow v \) \( P_2 \). We have

\[
\begin{align*}
    w(P') &= w(P_1) + w(P_2) \\
    &\geq D[v'] + w(P_2) \\
    &\geq D[v]
\end{align*}
\]

Because \( D[v'] \geq D[v] \) (by assumption) and \( w(P_2) \geq 0 \) (non-negative weights).

This is a contradiction because we assumed \( w(P') < w(P) \).

Hence, we know the weight of the shortest path \( s_0 \rightarrow v \).

Suppose \( v' = v \) for contra. Then \( D[v'] = D[v] \), so \( w(P') \geq D[v] \).

Contradicts \( w(P') < D[v] \).
Considering a new candidate path to $v'$, and possibly updating $D[v']$, is known as **relaxing** $v'$

To reconstruct an actual shortest path from distances, we also maintain a **predecessor** $\pi[v]$ for each node ($\pi[v] =$ previous node on a path $s_0 \leadsto v$ with weight $D[v]$)