CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

- Proving the main SCC lemma from last time
- Minimum (cost) spanning tree (MST)
  - Kruskal’s algorithm
- Shortest (weighted) path
  - Disjkstra’s algorithm
THE MAIN SCC LEMMA

We skimmed over it last time, and shouldn’t have
**PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH**

**Definition:** For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$

**Lemma:** if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

**Proof.** Case 1 ($d[C_i] < d[C_j]$):

- Let $u$ be the earliest discovered node in $C_i$
- All nodes in $C_i \cup C_j$ are white-reachable from $u$, so they are descendants in the DFS forest and finish before $u$
- So $f[C_i] > f[C_j]$
PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH

• **Definition:** For a strongly connected component \( C \), let \( d[C] = \min\{d[v] : v \in C\} \) and \( f[C] = \max\{f[v] : v \in C\} \)

• **Lemma:** if \( C_i, C_j \) are SCCs and there is an edge \( C_i \rightarrow C_j \), then \( f[C_i] > f[C_j] \)

• **Proof.** Case 2 (\( d[C_i] > d[C_j] \)):
  • Since component graph is a DAG, there is no edge \( C_j \rightarrow C_i \)
  • Thus, no nodes in \( C_i \) are reachable from \( C_j \)
  • So we discover \( C_j \) and finish \( C_j \) without discovering \( C_i \)
  • Therefore \( d[C_j] < f[C_j] < d[C_i] < f[C_i] \). QED
CONSEQUENCE OF THE LEMA

• So, if we perform $DFSVisit(u)$ on nodes from largest to smallest finishing time, any other SCCs reachable from the current SCC must already be finished/black

• So each $DFSVisit(u)$ call will explore precisely one SCC
MINIMUM SPANNING TREE
WEIGHTED UNDIRECTED GRAPH

- Consider an undirected graph in which each edge has a weight (or cost)
MINIMUM SPANNING TREE (MST)

- A tree (connected DAG) that includes every node, and **minimizes** the total sum of edge **weights**
Some facts about trees:

- A tree on $n$ vertices has $n - 1$ edges.
- There is a unique path between any two vertices in a tree.
- If $T$ is a tree and an edge $e \notin T$ is added to $T$, then the resulting graph contains a unique cycle $C$.
- If $e' \in C$ then $T \cup \{e\} \setminus \{e'\}$ is a tree.

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 \textit{minimize total cost}
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.
APPLICATION: CURVILINEAR FEATURE EXTRACTION

Want a machine to **recognize** this object

Edge detection algorithm

MST

Final result

"Hair" removal

Input to image recognition alg.
APPLICATION: K-CLUSTERING ANALYSIS

Want to cluster data into three groups? Build MST then delete 3 most costly edges.
APPLICATION: APPROXIMATION ALGORITHMS

Problem: plan traveling salesman routes (same start & end, visit each stop once, hopefully minimum total distance)

NP hard! Exponential time.
But can get route that is within 50% of optimal distance fast w/MST
[link to the approximation algorithm]
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:** \( \text{Kruskal}(G, w) \)

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

return \( (A) \)
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 \( v \to L[v] \to L[L[v]] \cdots \) 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

\[
\begin{align*}
    u' &\leftarrow \text{find}(u) \\
v' &\leftarrow \text{find}(v) \\
    L[u'] &\leftarrow v'.
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \text{find}(a) \rightarrow a )</th>
<th>( \text{find}(d) \rightarrow d )</th>
<th>( \text{find}(c) \rightarrow c )</th>
<th>( \text{find}(c) \rightarrow b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L[a] = b )</td>
<td>( L[d] = b )</td>
<td>( L[c] = b )</td>
<td>( \text{no merge!} )</td>
</tr>
</tbody>
</table>
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: $O(m \log m) = O(m \log n)$. Why? Some intuition: $m \log m \in O(m \log n^2) = O(m \cdot 2 \log n) = O(m \log n)$. This is called union by rank
Making this faster

- 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.
Radix sort

Union-Find with path compression and union by rank

Is this hard to implement? No!
# Just How Simple is Efficient Union-Find?

```c++
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**: The variable `parent` and `rank` are initialized.
- **Path compression**: The `find` function reduces the depth of the tree by setting the parent of the searched node to the root.
- **Union by rank**: The `merge` function uses the rank to determine which root becomes the new root. If the ranks are equal, the rank of the root is incremented.
Proof of Correctness

Let’s assume that all edge weights are distinct. Let $A$ be the spanning tree constructed by Kruskal’s algorithm and let $A'$ be an arbitrary MST.

Suppose the edges in $A$ are named $f_1, f_2, \ldots, f_{n-1}$, where $w(f_1) < w(f_2) \cdots < w(f_{n-1})$. Suppose $A \neq A'$ and let $f_j$ be the first edge in $A \setminus A'$.

$A' \cup \{f_j\}$ contains a unique cycle, say $C$. Let $e'$ be the first (i.e., lowest weight) edge of $C$ that is not in $A$ (such an edge exists because $C \not\subseteq A$). Define $A'' = A' \cup \{f_j\} \setminus \{e'\}$. Then $w(A'') = w(A') + w(f_j) - w(e')$.

Since $A'$ is an MST, we must have $w(A'') \geq w(A')$. Therefore, $w(f_j) \geq w(e')$. The edge weights are all distinct, so $w(f_j) > w(e')$.

What happened when Kruskal’s algorithm considered the edge $e'$? This occurred before it considered the edge $f_j$, because $w(f_j) > w(e')$. Since Kruskal’s algorithm rejected the edge $e'$, the edges $f_1, \ldots, f_{j-1}, e'$ must contain a cycle. However, $A'$ contains all these edges and $A'$ is a tree, so we have a contradiction. Therefore $A = A'$ and $A$ is an MST.
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 $v \neq s$, output a path $s \rightarrow v$ with the **smallest total weight** (among all paths $s \rightarrow v$)
- I.e., each path $P$ should minimize $w(P) = \sum_{e \in P} w(e)$

```
Suppose this is $s$
Shortest path to $i$
Shortest path to $d$
Shortest path to $c$
```

And so on... one path for each node.
DIJKSTRA’S ALGORITHM

• Idea:
  • Maintain a set $S$ of nodes for which shortest paths are known
  • Also maintain a distance $D[v]$ to node $v$ and predecessor $\pi[v]$
    • For $v \in S$, $D[v]$ should be the weight of the shortest path to $v$ that uses only nodes in $S$
    • For $v \not\in S$, $D[v]$ should be weight of the shortest path to $v$ that uses only nodes in $S \cup \{v\}$
  • Until $S = V$: choose $v \not\in S$ with the smallest $D[v]$, add $v$ to $S$, and update $D$ and $\pi$ appropriately

$u \in S$, so $D[u]$ considers only nodes in $S$

$D[s] = 0$

$u' \not\in S$, so $D[u']$ considers nodes in $S \cup \{u'\}$

Why can we add $v$ to $S$? (Why is the shortest path $s \rightarrow v$ known?)

How to update $D$ and $\pi$?

What’s special about the smallest $D[v]$ value?
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 \( s \rightarrow v \), which we denote \( P \).

Proof.

Suppose there is a path \( s \rightarrow v \) \( P' \) 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 \( s \rightarrow v' \) \( P_1 \) and a path \( v' \rightarrow v \) \( P_2 \). We have

\[
\begin{align*}
    w(P') &= w(P_1) + w(P_2) \\
    &\geq D[v'] + w(P_2) \\
    &\geq D[v] \quad (w(P_2) \geq 0 \text{ because } w \text{ is non-negative})
\end{align*}
\]

This is a contradiction because we assumed \( w(P') < w(P) \). Hence, we know the weight of the shortest path \( s \rightarrow v \).

So, we can add \( v \) to \( S \).
Considering a new candidate path to $v'$, and possibly updating $D[v']$ and $\pi[v']$, is known as **relaxing** $v'$.
WE STOPPED HERE
We know the shortest path $s \rightarrow s$

Relax the neighbours of $s$

Repeatedly find the node with the minimum $D$-value and relax its neighbours
DIJKSTRA’S ALGORITHM

- Initially $S = \{s\}$ and $D(v) = \begin{cases} 0, & v = s \\ \infty, & v \neq s \end{cases}$

- Until $S = V$: choose $v \notin S$ with the smallest $D[v]$, then add $v$ to $S$, and relax its neighbours.

Consider Dijkstra’s from here

Showing $D$-values

$a \notin S$ has smallest $D$-value

relax...

Done!

$D$ holds distances to each node. Reverse edges in $\pi[v], \pi[\pi[v]], ...$ to get shortest path to $v$!
Algorithm: $\text{FindPath}(s, \pi, v)$

\begin{align*}
\text{path} & \leftarrow v \\
u & \leftarrow v \\
\text{while } u \neq s \quad \text{do} \\
& \quad u \leftarrow \pi[u] \\
& \quad \text{path} \leftarrow u \parallel \text{path} \\
\text{return } (\text{path})
\end{align*}
Algorithm: \( \text{Dijkstra}(G, w, s) \)

\[
\begin{align*}
S & \leftarrow \{s\} \\
D[s] & \leftarrow 0 \\
\text{for all } v \in V \setminus \{s\} & \text{ do} \\
& \begin{cases}
D[v] & \leftarrow w(s, v) \\
\pi[v] & \leftarrow s
\end{cases} \\
\text{while } |S| < n & \text{ do} \\
& \begin{cases}
\text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \\
S & \leftarrow S \cup \{v\}
\end{cases} \\
& \text{for all } v' \in V \setminus S \text{ do} \\
& \begin{cases}
\text{if } D[v] + w(v, v') < D[v'] \\
& \begin{cases}
D[v'] & \leftarrow D[v] + w(v, v') \\
\pi[v'] & \leftarrow v
\end{cases}
\end{cases}
\text{end}
\end{align*}
\]

return \((D, \pi)\)

Build heap with all distances set to \(\infty\) except for \(s\)

After \text{relaxing} \(v\), we call \(Q.\text{updatePriority}(v, D[v])\)

How can we more efficiently implement this step?

Keep nodes in a priority queue \(Q\), sorted by \(D\) values!

\(v := Q.\text{removeMin}()\)

Can use array of bool to maintain \(S\)

Adjacency lists & priority queue: \(O((n + m) \log n)\)

[Simple implementation]

Adjacency matrix & linear search: \(O(n^2)\)

After \text{relaxing} \(v'\), we call \(Q.\text{updatePriority}(v', D[v'])\)
LIMITATIONS OF DIJKSTRA’S

• No negative edge weights
• Not efficient if you want all-pairs shortest paths

• Algorithms that address these downsides next time!