CS 341: ALGORITHMS
Lecture 13: graph algorithms IV – minimum spanning trees
Readings: see website
Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

MINIMUM SPANNING TREE (MST)
A tree (connected acyclic graph) that includes every node, and minimizes the total sum of edge weights

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 [PAPER]
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
Input to image recognition alg.

Just for fun, don’t need to know this
USEFUL TREE FACTS

• 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.

THE CUT OF A GRAPH

Definition: \( a \) cut in a graph \( G = (V, E) \) is a partition of \( V \) into two non-empty subsets \( S \) and \( V \setminus S \)

A CUT OF A GRAPH

In every MST, the minimum weight edge in the cutset is in every MST for \( G \)

THE CUT PROPERTY

Theorem: for any cut \( (S, V \setminus S) \) of a graph \( G \), the minimum weight edge in the cutset is in every MST for \( G \)

PROOF OF THE CUT PROPERTY

Let \( e = (u, v) \) be the lightest edge crossing the cut \((u \in S, v \in V \setminus S)\)

PROOF OF THE CUT PROPERTY

We construct spanning \( T \) s.t. \( w(T) < w(T) \) for contra.

T is spanning, so exists path \( u \rightarrow v \)

Path starts in \( S \) and ends in \( V \setminus S \)

so contains an edge \( e' = (u', v') \) with \( u' \in S, v' \in V \setminus S \)

Let \( T' = T - (e') + (e) \)
**PROOF OF THE CUT PROPERTY**

Let $T' = T - (e') + (e)$

**RECAP: THE CUT PROPERTY**

Theorem: for any cut $(S, V \setminus S)$ of a graph $G$, the minimum weight (lightest) edge in the cutset (crossing the cut) is in every MST for $G$.

**EXAMPLE EXECUTION**

How can we test for cycles as we go?

- Kruskal's algorithm [introduced in this 3-page paper from 1955]
- Greedy
  - Sort edges from lightest to heaviest
  - For each edge $e$ in this order
    - Add $e$ to $T$ if it does not create a cycle

Increasing edge weights: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

- 6 would create a cycle a, b, c, d
- 11 would create a cycle a, b, c, d
- 14 would create a cycle a, b, c, d
- 15 would create a cycle a, b, c, d
- 16 would create a cycle a, b, c, d
- 17 would create a cycle a, b, c, d
- 18 would create a cycle a, b, c, d
- 19 would create a cycle a, b, c, d
- 20 would create a cycle a, b, c, d

Done!
PROOF

- Let $T$ be partial spanning tree just before adding $e = (u, v)$, the lightest edge that does not create a cycle.
- Let $S$ be the connected component of $T$ that contains $u$.

\[
\text{PROOF}
\]

- Note $e = (u, v)$ crosses the cut $(S, V \setminus S)$ or it would create a cycle.
- Out of all edges crossing the cut, $e$ is considered first, so it is the lightest of these edges.

IMPLEMENTING KRUSKAL’S

- Sort edges from lightest to heaviest.
- For each edge $e$ in this order:
  - Add $e$ to $T$ if it does not create a cycle.

How can we determine whether adding $e$ would create a cycle?

UNION FIND

- Represents a partition of $S = \{e_1, \ldots, e_n\}$ into disjoint subsets.
- Initially $n$ disjoint subsets $S_i = \{e_i\}$.
- Operations:
  - $\text{Unmerge}(S_i, S_j)$ replaces $S_i$ and $S_j$ by their union $S_i \cup S_j$.
  - $\text{Find}(e_i)$ returns the label of the set containing $e_i$.

To avoid strange/long names, keep one of the original set names.

KRUSKAL’S USING UNION-FIND

- Each graph node is initially in its own subset.
- Add an edge $\rightarrow$ union two subsets.
- An edge creates a cycle IFF its endpoints are in the same subset.

PSEUDOCODE FOR KRUSKAL’S USING UNION-FIND

```
// Kruskal(V[1..m], E[1..m])
1. sort E[1..m] in increasing order by weight
2. set uf = new UnionFind data structure
3. set S = new list
4. for i = 1..m
5.   set a = uf.find(E[i].source)
6.   set b = uf.find(E[i].target)
7.   if Set a = set b
8.     uf.add(E[i])
9. else
10.    S.add(uf.merge(a, b))
11. return S
```
TIME COMPLEXITY?

- Kruskal(W,U,E)
- sort E in increasing order by weight
- uf = new UnionFind data structure
- mat = new List
- for j = 1..E
  - set_a = uf.find(E[j].source)
  - set_b = uf.find(E[j].target)
  - if set_a != set_b:
    - mat.add(E[j])
  - uf.union(set_a, set_b)
- return mat

Need to know runtime for union-find.

For an efficient union-find algorithm (with union-by-rank and path compression), we get a total running time for Kruskal's algorithm of $O(\alpha(m) + n \log n)$, where $\alpha(x)$ is the inverse Ackermann function.

For all practical $x$, $\alpha(x) \leq 5$, so this is pseudo-linear.

A simpler implementation with union-by-rank only yields $O(m \log n)$.

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 lightest edge that crosses cut $(T, V \setminus T)$
  - Visualization: https://www.cs.usfca.edu/~galles/visualization/Prim.html
- 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

A FUN APPLICATION: MAZE BUILDING

- Create grid graph with edges up/down/left/right
- Randomize edge weights
- then run Kruskal’s

VISUALIZING KRUSKAL’S (WITHOUT PATH COMPRESSION)

- https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

PROOF VIA EXCHANGE

- Supose $K$ is not an MST for contradiction. Let $O$ be an (optimal) MST. Note $O \subseteq K$.

  - Let $f_j$ be first edge not in $O$
  - Let $e'$ be smallest edge in $C \setminus K$
  - Add $e'$ to $O$ and delete edge $f_j$

  - Now $w(f_j) < w(e') < \cdots < w(f_{j-1})$
  - If $f_j$ at same as $f_j$ but with $e'$ and $f_j$ swapped

BONUS SLIDES

- Kruskal’s proof via exchange argument instead
- Implementing union-find efficiently
UNION FIND IMPLEMENTATION
- Suppose we are partitioning set \( \{1, \ldots, n\} \) into subsets \( S_1, \ldots, S_n \).
- Represent the partition as a forest of trees
- Initially one single-node tree per subset
- Each node has a parent pointer
- \( \text{Find}(i) \) returns the root of the tree containing element \( i \)
- \( \text{Union}(i, j) \) makes one root the parent of the other

\[ \begin{align*}
\text{Union-find forest (physical):} \\
\text{parent} & \quad 1 \quad 2 \quad 3 \quad 4 \\
\text{Union-find forest (logical):} \\
& \quad 1 \quad 2 \quad 3 \quad 4 \\
& \quad 1 \quad 2 \quad 3 \quad 1
\end{align*} \]

PROBLEM: SLOW FIND()
- Long paths \( \rightarrow \) slow find()
- Find runtime could be \( O(\text{number of unions performed}) \)

UNION-FIND WITH UNION BY RANK
- Keep track of heights of trees
- Make root with greater height be the parent
- Union of two trees with height \( h \) has height \( h + 1 \)
- Union of tree with height \( h \) and tree with height \( < h \) has height \( h \)
- Runtime with union by rank?

RUNTIME OF UNION BY RANK
- Can prove the following lemma by induction:
  - Each tree of height \( h \) contains at least \( 2^h \) nodes
  - There are only \( n \) nodes in the graph
    - So height \( h \) is at most \( \log n \)
    - \( \text{(Lemma: a tree of height } \log n \text{ contains at least } 2^\log n \text{ nodes) and } 2^\log n = n \)
- So the longest path in the union-find forest is \( \log n \)
- So all union-find operations run in \( \Theta(\log n) \) time!
TIME COMPLEXITY USING UNION BY RANK

- Sort edges in increasing order by weight
- \( \mathcal{O}(n \log n) \)
- Initialize UF data structure
- \( \mathcal{O}(n) \)
- For each edge \( e \) in sorted order:
  - Union by rank:
    - \( \mathcal{O}(1) \)
    - \( \mathcal{O}(1) \)
    - \( \mathcal{O}(1) \)
  - Find:
    - \( \mathcal{O}(\log n) \)
  - \( \mathcal{O}(m \log n) + \mathcal{O}(m \log m) \)

Trick:
\[
\log m \leq \log n^2 = 2 \log n \leq \mathcal{O}(\log n)
\]
So runtime is in \( \mathcal{O}(\log n) \) \( \times \)

MAKING THIS EVEN FASTER

- 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 runtime of \( \mathcal{O}(m \alpha(m) + n \log^* n) \), where \( \alpha(n) \) is the inverse Ackermann function.

For all practical \( n \), \( \alpha(n) \leq 5 \), so this is pseudo-linear.

EFFICIENT UNION-FIND

- \( \mathcal{O}(1) \) initialization
- \( \mathcal{O}(1) \) path compression
- \( \mathcal{O}(1) \) union by rank
- \( \mathcal{O}(1) \) free memory at end