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

WEIGHTED UNDIRECTED GRAPH
• Consider an undirected graph in which each edge has a weight (or cost)

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
• Fun 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

APPLICATION: CURVILINEAR FEATURE EXTRACTION
[Paper]
• Want one line to recognize the object
  • Edge detection algorithm
  • Break into segments
  • MST
  • Result
  • Just for fun, don’t need to know this
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.

**The Cut of A Cut**

- Definition: given a cut $(S, V \setminus S)$, the cutset is the set of edges with one endpoint in $S$ and the other in $V \setminus S$.

**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**

- We construct spanning $T$ s.t. $w(T') < w(T)$ for contradiction.
- $T$ is spanning, so exists path $u \to v$.
- Path starts in $S$ and ends in $V \setminus S$.
- So contains an edge $e' = (u', v')$ with $u' \in S$ and $v' \in V \setminus S$.
- Let $T' = T - (e') + (e)$.

**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 $T \cup \{e\} \setminus C$ is a tree.
**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**

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

---

**BUILDING AN MST**

- **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
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 $u$

So the cut property implies $e$ is in every MST of the graph

So every edge chosen by Kruskal's is in every MST

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?

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

Both endpoints already in same set! Do not add.

PSEUDOCODE FOR KRUSKAL'S USING UNION-FIND

```python
Kruskal(G, w):
    sort E in increasing order by weight
    uf = new union-find data structure
    mat = new list
    for (u, v, w) in E:
        set_a = uf.find(u, source)
        set_b = uf.find(v, target)
        if set_a == set_b:
            uf.union(set_a, set_b)
    return mat
```
**TIME COMPLEXITY?**

For an efficient union-find algorithm with union by rank and path compression, we get a total running time for Kruskal’s algorithm of \( O(\alpha(m) + n m + n) \), where \( \alpha(x) \) is the inverse Ackermann function. For all practical \( x \), \( \alpha(x) \leq 5 \), so this is pseudo-linear.

**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 \( (T, \overline{T}) \)
  - Visualization: [https://www.cs.usfca.edu/~galles/visualization/Prim.html](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**

Suppose \( K \) is not an MST for contradiction. Let \( O \) be an (optimal) MST. Note \( O = K \).

Label edges so \( w(f_1) \leq w(f_2) \leq \cdots \leq w(f_n) \) where \( f_1 \) is the first edge not in \( O \).

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

**VISUALIZING KRUSKAL’S**

- [With Path Compression](https://www.cs.usfca.edu/~galles/visualization/Kruskal.html)

**BONUS SLIDES**

- Kruskal’s proof via exchange argument instead
- Implementing union-find efficiency
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.

Union-find forest (physical):

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

Union-find forest (logical):

```
1 2 3 4
2 2 2 2
4 3 4 2
4 3 4 2
```

PROBLEM: SLOW FIND()

- Long paths \( \Rightarrow \) slow \( \text{Find}() \).
- 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 is at most \( \log n \).
  - (Lemma: a tree of height \( \log n \) contains at least \( 2^{\log n} \) 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

Using Union by Rank, the time complexity can be calculated as follows:

- Total time: \( O(m \log n) + m \log m \)
- \( \log m \leq \log n \)
- \( 2 = 2 \log n \in O(\log n) \)

So runtime is in \( O(m \log n) \).

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

Using both union by rank and path compression, we get a total running time for Kruskal's algorithm of \( O(\alpha(m) + n + m \log n) \), where \( \alpha(x) \) is the inverse Ackermann function. For all practical cases, \( \alpha(x) \leq 5 \), so this is pseudo-linear.

EFFICIENT UNION-FIND

Path compression
- Free memory at end
- Union by rank
- Initialization

This variant is introduced in this paper.