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
• Consider an undirected graph in which each edge has a weight (or cost)
A tree (connected acyclic graph) that includes every node, and **minimizes** the total sum of edge **weights**

Problem can also be defined for minimum spanning **forest**. Algorithm taught here works.
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.

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

Want a machine to recognize this object

Edge detection algorithm

MST

“Hair” removal

Input to image recognition alg.

Final result

Just for fun, don’t need to know this

Paper
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 $e \not\in T$ 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.
• Definition: a cut in a graph $G = (V,E)$ is a partition of $V$ into two non-empty subsets $S$ and $V \setminus S$
THE CUTSET 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 \(V \setminus S\). Edges in the cutset are also said to "cross the cut".
Theorem: for any cut $(S, V \setminus S)$ of a graph $G$, the minimum weight edge in the cutset is in every MST for $G$. The minimum weight edge is also called the "lightest edge". This can also be referred to as the lightest edge crossing the cut.
PROOF OF THE CUT PROPERTY

• Let $e = (u, v)$ be the lightest edge crossing the cut ($u$ in $S$, $v$ in $V \setminus S$)
• Let $T$ be an MST and suppose $e \notin T$ for contradiction
We construct spanning $T'$ s.t. $w(T') < w(T)$ for contra.

- $T$ is spanning, so exists path $u \leadsto v$
- 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**

This edge crosses the cut

Exchanging edges that cross the cut
Let $T' = T - \{e'\} + \{e\}$

PROOF OF THE CUT PROPERTY

$T = \text{white edges}$
PROOF OF THE CUT PROPERTY

Let $T' = T - \{e'\} + \{e\}$

Adding $e$ would create a cycle with $e$'

But removing $e'$ breaks that cycle and results in a tree

And a tree contains all-to-all paths
PROOF OF THE CUT PROPERTY

\[ T' = T - \{ e' \} + \{ e \} \]

So \( T' \) is still a spanning tree.

And \( w(T') = w(T) - w(e') + w(e) \)

But \( e \) and \( e' \) both cross the cut, and \( e \) is the lightest edge crossing the cut!

So \( w(e) < w(e') \), which means \( w(T') < w(T) \)

So \( T \) cannot be an MST if it doesn't contain \( e \)
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 \(G\).
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
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!
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 $u$. 
• 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

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?
UNION FIND

• Represents a **partition** of set $S = \{e_1, ..., e_n\}$ into **disjoint subsets**
  • Initially $n$ disjoint subsets $S_i = \{e_i\}$

• Operations
  • $\text{Union}(S_i, S_j)$ 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

Graph:

Union-find:

Both endpoints already in same set! Do not add.
PSEUDOCODE FOR KRUSKAL'S USING UNION-FIND

1. Kruskal(V[1..n], E[1..m])
2.     sort E[1..m] in increasing order by weight
3.     uf = new UnionFind data structure
4.     mst = new List
5.     for j = 1..m
6.         set_a = uf.find(E[j].source)
7.         set_b = uf.find(E[j].target)
8.         if set_a != set_b
9.             mst.add(E[j])
10.            uf.merge(set_a, set_b)
11. return mst
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.

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

```java
Kruskal(V[1..n], E[1..m])
    sort E[1..m] in increasing order by weight
    uf = new UnionFind data structure
    mst = new List
    for j = 1..m
        set_a = uf.find(E[j].source)
        set_b = uf.find(E[j].target)
        if set_a != set_b
            mst.add(E[j])
            uf.merge(set_a, set_b)
    return mst
```
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](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

Use priority queue to store outgoing edges from $T$ (and repeatedly extract the minimum weight one)

There is also a fast parallel hybrid of Prim and Borůvka
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
BONUS SLIDES

- Kruskal’s proof via exchange argument instead
- Implementing union-find efficiently
PROOF VIA EXCHANGE

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

Let $O'$ be same as $O$ but with $e'$ and $f_j$ swapped.

Note $w(O') = w(O) + w(f_j) - w(e')$

$w(O') \geq w(O)$ since $O$ is optimal

So $w(f_j) - w(e') \geq 0$, so $w(f_j) > w(e')$

Kruskal considers $e'$ before $f_j$, and rejects $e'$ despite taking $f_1, ..., f_{j-1}$

So, $f_1, ..., f_{j-1}, e'$ contains a cycle $C'$

But $f_1, ..., f_{j-1}, e' \in O$. Contradiction!

Label edges so $w(f_1) < w(f_2) < \cdots < w(f_{n-1})$.

We prove this for distinct weights.

Let $f_j = \text{first edge not in } O$.

Adding $f_j$ to $O$ would create cycle $C$.

Let $e' = \text{smallest edge in } C \setminus K$.

(Exists since no cycles in $K$.)
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

• \texttt{Find}(i) returns the root of the tree containing element \(i\)

• \texttt{Union}(i, j) makes one root the parent of the other

Let's union the sets containing elements 1 and 2
\[
\begin{align*}
\text{find}(1) & \to 1, \quad \text{find}(2) \to 2 \\
\text{Union}(1,2): \quad \text{parent}[1] &= 2
\end{align*}
\]

How about elements 4 and 1?
\[
\begin{align*}
\text{find}(4) & \to 4, \quad \text{find}(1) \to 2 \\
\text{Union}(4,2): \quad \text{parent}[2] &= 4
\end{align*}
\]

How about elements 3 and 1?
\[
\begin{align*}
\text{find}(3) & \to 3, \quad \text{find}(1) \to 4 \\
\text{Union}(3,4): \quad \text{parent}[3] &= 4
\end{align*}
\]
PROBLEM: SLOW FIND()

Long paths $\rightarrow$ slow find()

Find runtime could be $O$(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?

**Union-find forest:**

Let's union the sets containing **elements** 1 and 2

$\text{find}(1) \rightarrow 1, \text{ find}(2) \rightarrow 2$

Union(1,2): **same height** $\rightarrow$ parent[1] = 2

How about elements 4 and 1?

$\text{find}(4) \rightarrow 4, \text{ find}(1) \rightarrow 2$

Union(4,2): **2's height is greater** $\rightarrow$ parent[4] = 2
RUNTIME OF UNION BY RANK

• Can prove the following lemma by induction:
  • Each tree of height \( h \) contains at least \( 2^h \) nodes

Case 1: trees of different height

By I.H.,
left tree already has \( \geq 2^h \) nodes.
So result has height \( h \) and \( \geq 2^h \) nodes
RUNTIME OF UNION BY RANK

• Can prove the following lemma by induction:
  • Each tree of height \( h \) contains at least \( 2^h \) nodes

Case 2: trees of same height

By I.H., each tree has \( \geq 2^h \) nodes.
Result has height \( h + 1 \) and \( \geq 2^h + 2^h \) nodes

And \( 2^h + 2^h = 2^{h+1} \). QED
RUNTIME OF UNION BY RANK

• How does the lemma help?
  • 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

Kruskal(V[1..n], E[1..m])
   sort E[1..m] in increasing order by weight
   uf = new UnionFind data structure
   mst = new List
   for j = 1..m
      set_a = uf.find(E[j].source)
      set_b = uf.find(E[j].target)
      if set_a != set_b
         mst.add(E[j])
         uf.merge(set_a, set_b)
   return mst

Total $O(m \log n + m \log m)$

Trick: $\log m \leq \log n^2 = 2 \log n \in O(\log n)$

So runtime is in $O(m \log n)$
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 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**.
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]++;
    }
};