CS 341: ALGORITHMS

Lecture 15: 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)

Problem can also be defined for directed graphs...
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**
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

Input to image recognition alg.

“Hair” removal

**MST**

**Final result**

[Paper]

Just for fun, don’t need to know this
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.
USING THESE FACTS TO **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:** \( \text{Kruskal}(G, w) \)

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

Suppose we have an oracle wouldCreateCycle(\( e_j \))
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, 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...

How can we test for cycles as we go?

Done!
UNION FIND

- Represents a **partition** of set $S = \{e_1, \ldots, e_n\}$ 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:**

- Node **a**
- Node **b**
- Node **c**
- Node **d**
- Edge **a** to **b** with weight 2
- Edge **b** to **d** with weight 3
- Edge **a** to **d** with weight 7
- Edge **c** to **d** with weight 8

**Union-find:**

- Set $\{a, b\}$
- Set $\{a, b, d\}$
- Set $\{a, b, c, d\}$
- Set $\{a\}$
- Set $\{b\}$
- Set $\{c\}$
- Set $\{d\}$

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

```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
```
Correspondence

\( f_n - 1 \neq f_1 \neq f_3 \)

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

Let \( f_j = \text{first edge in } K \setminus O \) (exists since \( K \neq O \)).

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

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

Need to know runtime for union find...
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.
- Find\(i\) returns the root of the tree containing element \(i\).
- Union\(i, j\) makes one root the parent of the other.

Let's union the sets containing elements 1 and 2.
\[
\text{find}(1) \rightarrow 1, \quad \text{find}(2) \rightarrow 2 \\
\text{Union}(1,2): \ parent[1] = 2
\]

How about elements 4 and 1?

\[
\text{find}(4) \rightarrow 4, \quad \text{find}(1) \rightarrow 2 \\
\text{Union}(4,2): \ parent[2] = 4
\]

How about elements 3 and 1?

\[
\text{find}(3) \rightarrow 3, \quad \text{find}(1) \rightarrow 4 \\
\text{Union}(3,4): \ parent[3] = 4
\]
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

- $\text{find}(1) \rightarrow 1$, $\text{find}(2) \rightarrow 2$
- $\text{Union}(1,2)$: **same height** $\rightarrow$ parent[1] = 2

How about elements 4 and 1?

- $\text{find}(4) \rightarrow 4$, $\text{find}(1) \rightarrow 2$
- $\text{Union}(4,2)$: 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])
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

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 $O(m \log n)$
In addition to union by rank, union-find can be implemented with **path compression** making this even faster.

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

This variant is introduced in this paper.
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]++;
    }
};
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 $T$
  - 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
  - There is also a fast parallel hybrid of Prim and Borůvka

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