Minimum Spanning Tree Problem: Given a graph $G = (V, E)$ with edge weights $w: E \rightarrow \mathbb{R}_{\geq 0}$, find a minimum weight subset of the edges that connects all the vertices.

The weight of a set $F \subseteq E$ is $\sum\{w(e): e \in F\}$.

Examples

Assuming the graph is connected, the edge subset will be a tree, called the minimum spanning tree.

How many edges in a spanning tree? $n-1$

If the edge weights are all 1, how can we do this?

BFS or DFS will find a spanning tree if $G$ is connected. $O(n+m)$

First check $m \geq n-1$ otherwise no spanning tree $O(1)$
Minimum spanning trees can be found using greedy algorithms (you have seen some of this in Math 239)

There are several possible greedy approaches, with different implementation challenges:
- add minimum weight edge first, never build a cycle. Kruskal’s algorithm.
- grow a connected graph from one vertex. Prim’s algorithm.
- throw away heavy edges, never disconnect.

### Kruskal’s algorithm

order edges by weight, $e_1, e_2, \ldots, e_m$ with $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$

$T := \emptyset$

for $i = 1 \ldots m$

if $e_i$ does not make a cycle with $T$ then

$T := T \cup \{ e_i \}$

For example:

![Graph example](image)

![Graph example](image)
Kruskal's algorithm

order edges by weight, $e_1, e_2, \ldots, e_m$ with $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$

$T := \emptyset$

for $i = 1 \ldots m$

if $e_i$ does not make a cycle with $T$ then

$T := T \cup \{e_i\}$

General situation

edge $e$ makes a cycle with $T$ iff $e$ joins vertices in the same connected component

Correctness — an exchange proof

Prove by induction on $i$ that there is a MST matching $T$ on the first $i$ edges.

- Basis: $i = 0$
- General case: assume by induction that there is an MST matching $T$ on first $i-1$ edges
**Correctness** — an exchange proof

Prove by induction on $i$ that there is a MST matching $T$ on the first $i$ edges.

**general case:** assume by induction that there is an MST matching $T$ on first $i-1$ edges

**algorithm**

$$T = t_1, t_2, \ldots, t_{i-1}, t_i, \ldots, t_{n-1}$$

$$M = t_1, t_2, \ldots, t_{i-1}, t_i, \ldots, t_{n-1}$$

edges ordered by weight

Let $t_i = e = (a,b)$. Let $C$ be the connected component containing $a$ when $t_i$ is added.

If we add $e$ to $M$, it makes a cycle which contains an edge $e'$ from $C$ to $V-C$.

Note $w(e') \geq w(e)$ by ⊗ and $e'$ comes after $e$ in edge order.

**Exchange:**

$$M' = M \cup \{e\} - \{e'\}$$

Claim $M'$ is a MST.

Then we're done since $M'$ matches $T$ on first $i$ edges.
Claim. $M' = M - \{e'\} \cup \{e\}$ is a MST.

Proof.

1. $M'$ is a spanning tree because it connects all vertices and $|M'| = |M|$

2. $w(M') \leq w(M)$
   
   $w(M') = w(M) - w(e') + w(e)$
   
   and $w(e') \geq w(e)$
   
   So $w(M') \leq w(M)$
Implementing and analyzing Kruskal’s algorithm  \(|V| = n, |E| = m\)

Kruskal’s algorithm

order edges by weight, \(e_1, e_2, \ldots, e_m\) with \(w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)\)

\(T := \emptyset\)

for \(i = 1 \ldots m\)

if \(e_i\) does not make a cycle with \(T\) then

\(T := T \cup \{e_i\}\)

\(O(m \log m)\) to sort edges by weight = \(O(m \log n)\)

Why is \(O(m \log m) = O(m \log n)\)?

\[
\leq m \in O(n^2) \quad \alpha(m \log m) \leq O(m \log (n^2)) = O(m \log n)
\]

\[
\geq \text{by assumption} \quad m \geq n-1
\]

Next: the algorithm must maintain connected components as we add edges. And we must be able to test if an edge \((a,b)\) has \(a,b\) in the same component or different components

Union-Find Problem

Maintain a collection of disjoint sets

Operations:

- Find(x) — which set contains element \(x\)
- Union — unite two sets

In our case the **elements** are vertices, and the **sets** are connected components of the forest \(T\) so far
Implementing and analyzing Kruskal’s algorithm \( |V| = n, |E| = m \)

Kruskal’s algorithm **using Union-Find**
- order edges by weight, \( e_1, e_2, \ldots, e_m \) with \( w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m) \)
- \( T := \emptyset \)
- initialze one component per vertex
- for \( i = 1 \ldots m \)
  - suppose \( e_i = (a,b) \)
  - if \( \text{Find}(a) \neq \text{Find}(b) \) then
    - \( T := T \cup \{ e_i \} \)
    - \( \text{Union}(\text{Find}(a), \text{Find}(b)) \)

Note that the names of the connected components do not matter. We only want to know is \( \text{Find}(a) = \text{Find}(b) \).
Union-Find Problem

Maintain a collection of disjoint sets
Operations:
- Find(x) — which set contains element x
- Union — unite two sets

We will see a simple implementation that gives $O(m \log n)$ time for Kruskal.
So the $O(m \log n)$ for sorting dominates the runtime.

There is a fancier union-find implementation that is faster:
- the algorithm is simple,
- the analysis is hard,
- the runtime involves inverse Ackerman’s function (it’s almost linear time).
But this does not speed up Kruskal’s algorithm.

Next: simple implementation of Union Find
Simple implementation of Union Find

Suppose elements are 1, 2, . . . , n  (as in our application for MST)
Keep an array S[1..n] where S[i] = the set containing element i, and keep a linked list of the elements in each set.

**Example**  
\( n=7 \)

- A = 1, 3, 5, 6  
- B = 2, 4  
- C = 7

**S:**

```
1  2  3  4  5  6  7  
A  B  A  B  A  A  C  
```

Union (A,B) — we can name the new set anything we like

- A = 1, 3, 5, 6, 2, 4  
- C = 7

**S:**

```
1  2  3  4  5  6  7  
```

Find(x) takes O(1) — just take S(x)

Union — must join two linked lists O(1), and update S entries for one of the sets so O(n) worst case

BUT it is better to update S for the smaller of the two sets!

If an element’s set number changes then the new set (more than) doubles. This can happen at most log n times. So total work for all Unions is O(n log n).

Total runtime for Union Find:  \( O(#\text{Finds}) + O(n \log n) \)
returning to Kruskal:

Kruskal’s algorithm using **Union-Find**

order edges by weight, \( e_1, e_2, \ldots, e_m \) with \( w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m) \)

\[
T := \emptyset
\]

initialize one component per vertex

for \( i = 1 \ldots m \)

suppose \( e_i = (a, b) \)

if \( \text{Find}(a) \neq \text{Find}(b) \) then

\[
T := T \cup \{ e_i \}
\]

\[
\text{Union} (\text{Find}(a), \text{Find}(b))
\]

Using the above simple Union Find, the runtime of Kruskal is:

\[
\mathcal{O}(m \log n) + \mathcal{O}(m) + \mathcal{O}(n \log n)
\]

Sort: \( \mathcal{O}(m \log n) \)

FindS: \( \mathcal{O}(m) \)

Unions: \( \mathcal{O}(n \log n) \)

Total: \( \mathcal{O}(m \log n) \)

since \( m \leq n-1 \)
Union-Find Problem

Maintain a collection of disjoint sets

Operations:
- Find(x) — which set contains element x
- Union — unite two sets

Fancier Union-Find

Store each set as a tree of parent pointers, with the set name at the root.

Union — make the “smaller” tree point to the larger one
Find — follow parent pointers from element to root to learn the set name, adding parent shortcuts as you go
Fancier Union-Find

Store each set as a tree of parent pointers, with the set name at the root.

Union — make the “smaller” tree point to the larger one
Find — follow parent pointers from element to root to learn the set name, adding parent shortcuts as you go

Runtime: n elements, k = #Find + #Union

simple approach was $O(k + n \log n)$ — this is best if $k$ is large, but when $k$ is small, the simple approach spends too much time on Unions

new approach, manageable analysis: $O(k \log^* n)$
new approach, harder analysis: $\Theta(k \text{ Inverse-Ackerman}(k,n))$

$log n = \# \text{ times you divide by 2 to get } \leq 1$
$log^* n = \# \text{ times you take log to get } \leq 1$

Example

$2^{16} = 65,536$

$log^*(2^{16}) = 4, \quad log^*(2^{2^{16}}) = 5$

$log^*$ is VERY slow growing.
Inverse-Ackerman is EVEN MORE slow growing. It is $\leq 5$ for all practical purposes.
Generalizing Kruskal’s algorithm

<table>
<thead>
<tr>
<th>Kruskal’s algorithm variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>order edges by weight, $e_1$, $e_2$, ... , $e_m$ with $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$</td>
</tr>
<tr>
<td>$T := \emptyset$</td>
</tr>
<tr>
<td>for $i = 1 \ldots m$</td>
</tr>
<tr>
<td>if $e_i$ does not make a cycle with $T$ then</td>
</tr>
<tr>
<td>$T := T \cup {e_i}$</td>
</tr>
</tbody>
</table>

Note: This variant algorithm finds a maximum weight tree/forest.

What properties make this greedy algorithm work?

1. If $F \subseteq E$ has no cycles (“is independent”) and $G \subseteq F$, then $G$ is independent.
2. If $F, G \subseteq E$ are both independent and $|G| \leq |F|$ then there is some $f \in F$ such that $G \cup \{f\}$ is independent.

These properties define a **matroid**. Kruskal’s greedy algorithm finds a maximum weight independent set for any matroid.

Another example of a matroid: elements are vectors, independence = linear independence. Kruskal’s greedy algorithm find a maximum weight set of linearly independent vectors.
Summary of Lecture 15

- Minimum Spanning Tree

- Kruskal’s greedy algorithm and implementation using Union Find

What you should know from Lecture 15:

- what is an MST

- what is Kruskal’s algorithm

- how to implement it and what is the runtime

- what is Union Find and the simple solution

- appreciate that Union Find has a fancier implementation, and that Kruskal’s algorithm generalizes to matroids

Next:

- algorithms for shortest paths in graphs