Graph Algorithms
Minimum Spanning Tree

**Input:** a connected, undirected graph \( G = (V, E) \) with weights \( w: E \rightarrow \mathbb{R} \) on the edges

**Output:** a minimum spanning tree \( T \)

- A **spanning tree** of \( G \) is a graph \( (V, T \subseteq E) \) such that \( (V, T) \) is a tree
  - A tree: a connected graph with no cycle
- The weight of a tree:
  \[
  w(T) = \sum_{(u,v) \in T} w(u, v)
  \]
- A **minimum spanning tree**: a tree of minimum weight:
  - subset of edges (of size \( n - 1 \)) that connects all the vertices and has minimum weight
Example of MST
Example of MST

The edges on spanning tree

The weight of the above tree is 6+5+8+3+7+9+15
Minimum Spanning Trees

There are many greedy algorithms for finding MSTs:

- Borůvka's algorithm (1926)
- Kruskal's algorithm (1956)
- Prim's algorithm (1930, rediscovered 1957)

We will explore Kruskal's algorithm and Prim's algorithm in this course.
Minimum Spanning Tree

- Can there be more than one minimum spanning tree (MST) for an undirected graph?
  - Yes

- What happens if the graph is unweighted?
  - All spannings trees are minimum spanning trees
Optimal substructure

MST $T$:

(Other edges of $G$ are not shown.)
Optimal substructure

MST $T$:

(Other edges of $G$ are not shown.)

Remove any edge $(u, v) \in T$. 
Optimal substructure

MST $T$:

(Other edges of $G$ are not shown.)

Remove any edge $(u, v) \in T$. 
Remove any edge \((u, v) \in T\). Then, \(T\) is partitioned into two subtrees \(T_1\) and \(T_2\).
Optimal substructure

MST $T$:

(Other edges of $G$ are not shown.)

Remove any edge $(u, v) \in T$.

Then, $T$ is partitioned into two subtrees $T_1$ and $T_2$.

**Theorem.** The subtree $T_1$ is an MST of $G_1 = (V_1, E_1)$, the subgraph of $G$ induced by the vertices of $T_1$:

- $V_1 =$ vertices of $T_1$,
- $E_1 = \{ (x, y) \in E : x, y \in V_1 \}$.

Similarly for $T_2$. 
Proof of optimal substructure

**Proof.** Cut and paste:

\[ w(T) = w(u, v) + w(T_1) + w(T_2). \]

If \( T_1' \) were a lower-weight spanning tree than \( T_1 \) for \( G_1 \), then \( T' = \{(u, v)\} \cup T_1' \cup T_2 \) would be a lower-weight spanning tree than \( T \) for \( G \).

**Contradiction:** since \( T \) was the minimum spanning tree for \( G \).
Kruskal Algorithm

T = Ø
Repeat
- find the least-weight edge (u,v) so that u and v are not connected in T
- add (u,v) to T
Kruskal Algorithm

T = Ø
Repeat
- find the least-weight edge (u, v) so that u and v are not connected in T
- add (u, v) to T

```
T = Ø
Repeat
  - find the least-weight edge (u, v) so that u and v are not connected in T
  - add (u, v) to T
```
Kruskal Algorithm

T = Ø
Repeat
  • find the least-weight edge (u,v) so that u and v are not connected in T
  • add (u,v) to T
Kruskal Algorithm

\[ T = \emptyset \]

Repeat
- find the least-weight edge \((u,v)\) so that \(u\) and \(v\) are not connected in \(T\)
- add \((u,v)\) to \(T\)
Kruskal Algorithm

\[ T = \emptyset \]

Repeat
  \- find the least-weight edge \((u,v)\) so that \(u\) and \(v\) are not connected in \(T\)
  \- add \((u,v)\) to \(T\)
Kruskal Algorithm

T = Ø
Repeat
  ● find the least-weight edge (u, v) so that u and v are not connected in T
  ● add (u, v) to T
Kruskal Algorithm

T = Ø
Repeat
  ● find the least-weight edge (u,v) so that u and v are not connected in T
  ● add (u,v) to T
Kruskal Algorithm

$T = \emptyset$
Repeat
- find the least-weight edge $(u,v)$ so that $u$ and $v$ are not connected in $T$
- add $(u,v)$ to $T$
Kruskal Algorithm

$T = \emptyset$

Repeat
- find the least-weight edge $(u, v)$ so that $u$ and $v$ are not connected in $T$
- add $(u, v)$ to $T$
Kruskal Algorithm

Another way to look at Kruskal algorithm:

- At each step, the algorithm merges two connected component
Graph Cuts

In a graph $G = (V, E)$

- A **cut** is a partition of the vertices of the graph into two sets $C$, $V - C$
  - We show it by $(C, V - C)$
  - $C \subseteq V$.

- An edge $(u, v)$ crosses the cut $(C, V - C)$ if exactly one of $u$, $v$ is in $C$

- If $G$ is connected, then at least one edge crosses every cut
Tree Facts

- A tree of 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 ∉ T is added to T, then the resulting graph contains a unique cycle C
- If e’ ∈ C then T U {e} \ {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’ ∈ C will break the cycle and produce a tree
  - Proof in the next slide
Theorem. Let $T$ be a tree and $e=(u, v) \notin T$. The graph $T \cup \{e\}$ contains a cycle. For any edge $e'=(x, y)$ on the cycle, the graph $T' = T \cup \{e\} \setminus \{e'\}$ is a tree.

Proof.

- $|T'| = |T| + 1 - 1 = |T| = |V| - 1 \rightarrow$ if $T'$ is connected, then it is a tree. Why?
  - $e \notin T$ and $e' \in T \cup \{e\}$
- Proving $T'$ is connected
  - Consider any $s, t \in V$. Since $T$ is connected, there is some path from $s$ to $t$ in $T$.
    - If that path does not cross $(x, y)$, or if $(x, y) = (u, v)$, then this path is also a path from $s$ to $t$ in $T'$, so $s$ and $t$ are connected in $T'$.
    - If the path from $s$ to $t$ crosses $(x, y)$. Assume WLOG that the path starts at $s$, goes to $x$, crosses $(x, y)$, then goes from $y$ to $t$. Since $(u, v)$ and $(x, y)$ are part of the same cycle, we can modify the original path from $s$ to $t$ so that instead of crossing $(x, y)$, it goes around the cycle from $x$ to $y$. This new path is then a path from $s$ to $t$ in $T'$, so $s$ and $t$ are connected in $T'$. Thus any arbitrary pair of nodes are connected in $T'$, so $T'$ is connected.
Kruskal Algorithm

Proof of correctness (feasibility)

Proof by induction
Kruskal Algorithm: Proof of optimality

- **T**: MST found by Kruskal Algorithm
- **M**: optimal MST

Proof by contradiction. Suppose $T \neq M$.

$T = e_1 e_2 \ldots e_j \ldots e_n$

$M = e_1 e_2 \ldots m_j \ldots m_n$

- $T$ and $M$ are the same up to $j$-th edge. Suppose $e_j = (u, v)$ is in $T$ but not in $M$
- **C**: the connected component containing $u$ when $(u, v)$ was added to $T$
- When $(u, v)$ was added, it was the least-cost edge crossing the cut $(C, V-C)$
  - $(u, v)$ crosses the cut, since $u$ and $v$ were not connected when Kruskal's algorithm selected $(u, v)$
  - Kruskal algorithm select the least-cost edge crossing the cut
- **M** is a MST $\rightarrow$ There must be a path from $u$ to $v$ in $M$. This path begins in $C$ and ends in $V-C$. $\rightarrow$ There must be an edge along that path where $x$ in $C$ and $y$ in $V-C$. Since $(u, v)$ is the least-code edge crossing $(C, V-C)$ $\rightarrow w(u, v) < w(x, y)$
- $M' = M - \{(x, y)\} \cup \{(u, v)\}$. $M'$ is a spanning tree because it connects all vertices. Since $(x, y)$ is on the cycle formed by adding $(u, v)$ $w(M') = w(M) - w(x, y) + w(u, y) < w(M) \rightarrow M'$ is a MST $\rightarrow$ contradiction $M$ was the optimal solution
- We used exchange argument
  - exchanging some part of the optimal solution with some part of the greedy solution improved the optimal solution $\rightarrow$ contradiction
- Note: here we are assuming the edge weights are **unique**, otherwise we do not reach a contradiction
Kruskal Algorithm: Proof of optimality
Kruskal Algorithm: Proof of optimality in general

- **T**: MST found by Kruskal Algorithm
- **M**: optimal MST

**Proof.** We will prove \( w(T) = w(M) \). If \( T = M \), we are done. Otherwise \( T \neq M \), so \( T - M \neq \emptyset \).

- Suppose \( e = (u, v) \) is in \( T \) but not in \( M \)
- **C**: the connected component containing \( u \) when \( (u,v) \) was added to \( T \)
- When \( (u,v) \) was added, it was the least-cost edge crossing the cut \( (C, V-C) \)
  - \((u, v)\) crosses the cut, since \( u \) and \( v \) were not connected when Kruskal's algorithm selected \((u, v)\)
  - Kruskal algorithm selects the least-cost edge crossing the cut
- **M** is a MST → There must be a path from \( u \) to \( v \) in \( M \). This path begins in \( C \) and ends in \( V-C \). → There must be an edge along that path where \( x \in C \) and \( y \in V-C \). Since \( (u,v) \) is the least-cost edge crossing \( (C, V-C) \) → \( w(u,v) \leq w(x,y) \)
- \( M' = M - \{x,y\} \cup \{(u,v)\} \). \( M' \) is a spanning tree because it connects all vertices. Since \( (x,y) \) is on the cycle formed by adding \( (u, v) \)
  - \( w(M') = w(M) - w(x,y) + w(u,y) \rightarrow w(M') \leq w(M) \)
- **M'** is a MST → \( w(M) \leq w(M') \rightarrow w(M') = w(M) \)
- Note that \(|T - M'| = |T - M| - 1\). Therefore, if we repeat this process once for each edge in \( T - M \), we will have converted \( M \) into \( T \) while preserving \( w(M) \). Thus \( w(T) = w(M) \).
- We used exchange argument
  - exchanging one edge of \( M \) with one edge of \( T \) without increasing \( w(M) \)
Kruskal Algorithm: pseudocode

Kruskal(G)
    Sort the edges by non-decreasing weight \( e_1 \ldots e_m \), \( w(e_i) \leq w(e_{i+1}) \)
    \( T = \emptyset \)
    for each edge \( (u, v) \)
        if \( u \) and \( v \) are not connected by \( T \)
            \( T = T \cup \{(u, v)\} \)
    return \( T \)
Kruskal Algorithm: pseudocode

Kruskal(G)

Sort the edges by non-decreasing weight $e_1 \ldots e_m$, $w(e_i) \leq w(e_{i+1})$

$T = \emptyset$

for each edge $(u, v)$

if $u$ and $v$ are not connected by $T$

$T = T \cup \{(u,v)\}$

return $T$

O(E lg E) or O(E lg V)

O(E)

Use DFS $\rightarrow$ the runtime of DFS is O(V+E). here, the runtime is O(V). why?

Runtime: O(VE)

Can we do better?
Kruskal Algorithm: A better implementation

- Union-find data structure:
  - Represents a **partition** of set $S= \{e_1, e_2, \ldots, e_n \}$ into **disjoint subsets**
    - Initially $n$ disjoint subsets $S_i = \{e_i\}$
  - a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$
  - Each element of data belong to exactly one set
  - Each set is identified by a representative (some member of the set)
    - Specifies which set an element belongs to

- Operations of **union-find** data structure
  - **Make-set**(x): Create a set containing one element, $x$
  - **union**(x, y): unites the sets containing $x$ and $y$ into one set
  - **find**(x): returns a pointer to the representative of the set containing $x$
Kruskal Algorithm using union-find data structure

Kruskal(G)
    Sort the edges by non-decreasing weight \( e_1 \ldots e_m \), \( w(e_i) \leq w(e_{i+1}) \)
    \( T = \emptyset \)
    \( S = \text{union-find data structure} \)
    for each \( v \) in \( V \)
        \( S.\text{make-set}(v) \)
    for each edge \((u, v)\)
        if \( S.\text{find}(u) \neq S.\text{find}(v) \)
            \( T = T \cup \{(u, v)\} \)
            \( S.\text{union}(u, v) \)
    return \( T \)
Kruskal Algorithm using union-find data structure

- Each graph node is initially in its own subset
- Add an edge → union two subsets
- An edge creates a cycle iff its endpoints are in the same subset
First 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
- Problem:
  - Long paths → slow find
First implementation

1 2 3 4

find(1) → 1, find(2) → 2

Union(1,2): parent[1] = 2
find(4) → 4, find(1) → 2

2 3 4

find(3) → 3, find(1) → 4

3 4

Union(3,4): parent[3] = 4

1
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 $h$
Union-find with union by rank

find(1) -> 1, find(2) -> 2

Union(1, 2): same height, parent[1] = 2
find(4) -> 4, find(1) -> 2

Union(4, 2): 2's height is greater: parent[4] = 2
Runtime of Union-find with union by rank

- Each tree of height $h$ contains at least $2^h$ nodes
- Proof by induction.
  - Base case: trees with height 0 have $2^0 = 1$ node
  - I.H.: a tree of height $h$ contains at least $2^h$ nodes
  - Induction step: Having I.H, we want to show a tree of height $h+1$ contains at least $2^{h+1}$ nodes.
    - Case 1: Union of trees of height $h$ and height < $h$
      - Left tree has at least $2^h$ nodes
      - Result has height $h$ and $\geq 2^h$ nodes
    - Case 2: Union of trees of same height
      - Each tree has $\geq 2^h$ nodes.
      - Result has height $h+1$ and $\geq 2^h + 2^h$ nodes
        - $2^h + 2^h = 2^{h+1}$
Runtime of Union-find with union by rank

- Each tree of height $h$ contains at least $2^h$ nodes
- There are only $n$ nodes in the graph
- Therefore the height is at most $\log n$
- The longest path in the union-find first is $\log n$
- So all union-find operations run in $\Theta (\log n)$ time
Union-find by rank and **path compression**

- A recursive logic is used to achieve path compression with each call to the find operation.
- The Union operation may increase the height of the trees
- The find operation tries to reduce the height at each call and to achieve flatter trees
- The flatter the trees, lower is the complexity of find and union operations.

```python
def find(x):
    if x != parent[x]:
        parent[x] = find(parent[x])  # path compression during find
    return parent[x]
```
Kruskal Algorithm

\[
\text{Kruskal}(G)
\]

Sort the edges by non-decreasing weight \( e_1 \ldots e_m \), \( w(e_i) \leq w(e_{i+1}) \)

\( T = \emptyset \)

for \( i = 1 \) to \( m \)

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

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

return \( T \)

\[ O(E \lg E) \text{ or } O(E \lg V) \]

Can be done in \( O(\alpha(E+V)) \) using union-find data structure

Runtime: \( O(E \log V) \)
Can we do better?
Acknowledgement

The slides of the following course:

https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/

And the slides of several previous CS 341@waterloo especially Trevor’s Brown slides