Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
computing a subgraph with minimum cost

Input: $G = (V, E)$ is a connected graph, with a cost $c_e > 0$ for every edge $e \in E$.

Output: $T \subseteq E$ so the graph $(V, T)$ is connected and $
\sum_{e \in T} c_e$ is minimal over all subsets $S$ of $E$
for which $(V, S)$ is connected.
any subgraph with minimum cost is a tree

Proposition

Consider a connected graph $G = (V, E)$ with cost $c_e > 0$ for all $e \in E$. Let $T \subseteq E$ such that $\sum_{e \in T} c_e$ is minimal over all $S \subseteq E$ for which $(V, S)$ is connected, then $(V, T)$ is a tree.

Proof. We show that $(V, T)$ has no cycles, by contradiction. Assume $T$ contains the cycle $C$. Take an edge $f \in C$. Removing $f$ from $C$ leaves the vertices in $C$ connected. So $T \setminus \{f\}$ is also connected, but $c_f > 0$ and

$$\left(\sum_{e \in T} c_e\right) - c_f = \sum_{e \in T \setminus \{f\}} c_e < \sum_{e \in T} c_e$$

is a contradiction. The assumption that $T$ contains a cycle is false. A connected graph with no cycles is a tree. Q.E.D.
spanning trees

Definition

For a connected graph \( G = (V, E) \) and \( T \subseteq E \) such that \((V, T)\) is connected, \((V, T)\) is a spanning tree for \( G \).

Dijkstra’s algorithm produces a tree of shortest paths from a given source vertex. A tree of shortest paths is a spanning tree.

Definition

For a connected graph \( G = (V, E) \) with \( c_e > 0 \) for all \( e \in E \) and \( T \subseteq E \) such that \((V, T)\) is connected and \( \sum_{e \in T} c_e \) is minimal over all \( S \subseteq E \) for which \((V, S)\) is connected, \((V, T)\) is a minimum spanning tree for \( G \).

Observe the \( a \), there may be more than one minimum spanning tree.

Is a tree of shortest paths from a given source vertex always a minimum spanning tree?
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
Kruskal’s algorithm

Input: \( G = (V, E) \) is a connected graph, with a cost \( c_e > 0 \) for every edge \( e \in E \).

Output: \( T \subseteq E \), a minimum spanning tree.

sort edges: \( e_i < e_j \) for all \( e_i, e_j \in E: c_{e_i} < c_{e_j} \)

\( T := \emptyset \)
for all \( e_i \in E \) do
  if \( e_i \) does not introduce a cycle in \( (V, T) \) then
    \( T := T \cup \{ e_i \} \)

Kruskal’s algorithm inserts edges in order of increasing cost.
Prim’s algorithm

Input: $G = (V, E)$ is a connected graph,
   with a cost $c_e > 0$ for every edge $e \in E$.
Output: $T \subseteq E$, a minimum spanning tree.

choose any vertex $s \in V$

$S := \{ s \}$

while $S \neq V$ do

   choose $v \in V \setminus S$, such that $c(u,v) = \min_{u \in S \atop (u,w) \in E} c(u,w)$

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

Prim’s algorithm adds the vertex with the smallest attachment cost.
the reverse-delete algorithm

Input: $G = (V, E)$ is a connected graph, with a cost $c_e > 0$ for every edge $e \in E$.
Output: $T \subseteq E$, a minimum spanning tree.

sort edges: $e_i < e_j$ for all $e_i, e_j \in E$: $c_{e_i} > c_{e_j}$

$T := E$

for all $e_i \in E$ do
  if $(V, T \setminus \{ e_i \})$ is connected then
    $T := T \setminus \{ e_i \}$

This algorithm deletes edges in order of decreasing cost.
Exercise 1:
Find an example of a connected graph so that the minimum spanning trees computed by Kruskal’s, Prim’s, and the reverse-delete algorithm are all different from each other.
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
Proposition (the cut property)

Consider a connected graph \( G = (V, E) \) and the following:

- All edge costs are distinct: for all \( e, e' \in E: e \neq e' \) implies \( c_e \neq c'_{e} \).
- Let \( S \) be the set of all vertices with minimum attachment cost, \( S \subset V, S \neq \emptyset \) and \( S \neq V \),
- Let \( e = (u, v) \in E: u \in S \) and \( v \in V \setminus S \) and \( e \) is the minimum-cost edge between \( S \) and \( V \setminus S \).

Every minimum spanning tree contains \( e \).

Proof. We prove by contradiction, using an exchange argument. Assume \( T \) collects the edges of a minimum spanning tree and \( e \not\in T \).

We will find an edge \( e' \in T \) with higher cost than \( e \). Replacing \( e' \) with \( e \), we obtain then a spanning tree with smaller total cost than the tree \((V, T)\). This swap contradicts the assumption that \( T \) collects the edges of a minimum spanning tree.
If $e \not\in T$, then $e' = (u', v') \in T$ as $e'$ is on the path from $u$ to $v$, with $u' \in S$ and $v' \in V \setminus S$.

Swapping $e'$ for $e$ leads to $T' = T \setminus \{e'\} \cup \{e\}$. $(V, T')$ is connected because any path that contained $e'$ can be rerouted using $e$. 
proof continued

Note in $T \cup \{ e \} = T' \cup \{ e' \}$ the only cycle is the cycle which contains the path from $u$ to $v$ in $T$ and $e$. Swapping $e'$ on the path from $u$ to $v$ by $e$ removes the cycle and therefore $(V, T')$ is a tree.

Note that $e' = (u', v')$, with $u' \in S$ and $v' \in V \setminus S$. But $e = (u, v)$ is the minimum cost edge between a vertex in $S$ and a vertex outside $S$, thus $c_e < c_{e'}$. The inequality is strict as all edges have a different cost.

Swapping $e'$ by $e$ leads to a spanning tree $(V, T')$ with a reduction in cost by $c_{e'} - c_e > 0$ compared to the total cost of $(V, T)$. $(V, T)$ is not a minimum spanning tree, a contradiction.

Thus every minimum spanning tree must contain the minimum-cost edge between a vertex in $S$ and a vertex in $V \setminus S$. Q.E.D.
Theorem

*Prim’s algorithm produces a minimum spanning tree.*

**Proof.** Prim’s algorithm stops after $n - 1$ steps, $n = \#V$, when $S = V$. The output $T$ of Prim’s algorithm defines a spanning tree $(V, T)$.

The $S$ in Prim’s algorithm is the $S$ of the cut property.

Every step of Prim’s algorithm chooses $v \in V \setminus S$, such that

$$c(u,v) = \min_{u \in S} \{ c(u,w) \}.$$ 

By the cut property, every edge added by Prim’s algorithm to $T$ is in every minimum spanning tree.

So, $(V, T)$ is a minimum spanning tree. Q.E.D.
Theorem

Kruskal’s algorithm produces a minimum spanning tree.

Proof. Consider any edge $e = (u, v)$ added by Kruskal’s algorithm. Let $S$ be the set of all connected vertices before the addition of $e$:

- $u \in S$, but $v \in V \setminus S$, because adding $e$ does not make a cycle.
- Kruskal’s algorithm adds edges in order of increasing cost, $e$ is the cheapest edge between a vertex in $S$ and a vertex in $V \setminus S$.

Thus the cut property applies and $e$ belongs to every minimum spanning tree. For the output $T$ of Kruskal’s algorithm,

- $(V, T)$ contains no cycles, so $(V, T)$ is a tree.
- $(V, T)$ is a spanning tree because $G$ is connected and as long as $S \neq V$, there is an edge from a vertex in $S$ to a vertex in $V \setminus S$.

Q.E.D.
Proposition (the cycle property)

Consider a connected graph \( G = (V, E) \) and the following:

- All edge costs are distinct: for all \( e, e' \in E: e \neq e' \) implies \( c_e \neq c_{e'} \).
- Let \( C \) be a cycle in \( G \).
- Let \( e = (u, v) \) be the edge in \( C \) with the highest cost.

Then \( e \) does not belong to any minimum spanning tree of \( G \).

Proof. We proceed by contradiction, using an exchange argument. Assume \( T \) collects the edges of a minimum spanning tree and \( e \in T \).

We will find an edge \( e' \not\in T \) with lower cost than \( e \).
Replacing \( e \) with \( e' \), we obtain then a spanning tree with smaller total cost than the tree \( (V, T) \). This swap contradicts the assumption that \( T \) collects the edges of a minimum spanning tree.
proof continued

Because $T$ is a spanning tree, deleting $e = (u, v)$ from $T$ partitions the set of vertices in $S$ and $V \setminus S$, with $u \in S$ and $v \in V \setminus S$.

Let $e'$ be the edge which has the minimum attachment cost to connect $S$ with $V \setminus S$. The edges $e$ and $e'$ belong to a cycle and $e$ is the edge on the cycle which has the highest cost.
Replacing $e$ by $e'$ leads to $T' = T \setminus \{ e \} \cup \{ e' \}$.

The replacement of $e$ removes the only cycle, so $(V, T')$ is a tree which connects all vertices in $V$. Thus, $(V, T')$ is a spanning tree.

Because $e$ is the edge with the highest cost on a cycle, we have that $c_e > c_{e'}$. Swapping $e$ by $e'$ reduces the cost by $c_e - c_{e'} > 0$ and $(V, T')$ is a spanning tree with lower total cost than $(V, T)$. $(V, T)$ is not a minimum spanning tree, a contradiction.

Thus every minimum spanning tree cannot contain the edge with the highest cost on a cycle.

Q.E.D.
Theorem

The reverse-delete algorithm produces a minimum spanning tree.

Proof. The reverse-delete algorithm deletes edges in order of decreasing cost. Every time an edge is removed the graph stays connected so it must be the most expensive edge on a cycle. Thus the cycle property applies and the removed edges do not belong to any minimum spanning tree. The reverse-delete algorithm removes all cycles by deleting the most expensive edges. All edges that do not belong to a minimum spanning tree are removed.

We still have to show that we obtain a spanning tree. The reverse-delete algorithm does not remove edges that causes $V$ to become disconnected. The graph produced by the algorithm has no cycles and is connected. The output is a spanning tree and thus a minimum spanning tree. Q.E.D.
on the assumption that all edge costs are distinct

For graphs with edges $e$ that have the same cost, we add a random infinitesimally small cost:

$$c_e := c_e + \epsilon_e, \quad \epsilon_e > 0, \epsilon_e \ll 1.$$  

Each $\epsilon_e$ is a different number, so all costs are different.

The $\epsilon_e$'s serve as tie breakers in the algorithms. Because the costs are all different, there is a unique selection of edges in the minimum spanning tree for each algorithm.

The $\epsilon_e$'s are too small to make a difference in an edge swap. So the spanning tree without $\epsilon_e$'s is the same as the spanning tree with the $\epsilon_e$'s.
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
Kruskal’s algorithm adds an edge \((u, v)\) in each step. For an efficient implementation of the addition \((u, v)\), we need
- to find the components that contain \(u\) and \(v\); and
- to merge two components in case \(u\) and \(v\) belong to two disjoint components of the graph.

Pseudo code to add the edge \((u, v)\):

\[
A := \text{Find}(u) \\
B := \text{Find}(v) \\
\text{if } A = B \text{ then } \\
\quad \text{insert the edge } (u, v) \text{ into } A \\
\text{else } \\
\quad \text{Union}(A, B)
\]
operations on a Union-Find data structure

We need three operations:

- **Make**($S$), for a set $S$ of vertices, returns a Union-Find data structure with $\#S$ components.

- **Find**($u$), for a vertex $u$, returns the name of the component which contains $u$.

- **Union**($A$, $B$), for two disjoint components $A$ and $B$, changes the Union-Find data structure, merging $A$ and $B$. 
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
an array for a Union-Find

For a set $S$ of $n$ vertices, $S = \{1, 2, \ldots, n\}$, use an array $\text{Component}$ of size $n$, where the names of components are labels in $\{1, 2, \ldots, n\}$.

- $\text{Make}(S)$ does $\text{Component}[s] := s$ for all $s \in S$, runs in $O(n)$ time.
- $\text{Find}(u)$ takes $O(1)$ time.
- $\text{Union}(A, B)$ updates $\text{Component}[s] := A$ for all $s \in B$.

Consider $\text{Union}(\text{Component}[k], \text{Component}[k+1])$, $k = 4, 3, 2, 1$:

```
  1 | 1   1 | 1   1 | 1   1 | 1
  2 | 2   2 | 2   2 | 2   2 | 2
  3 | 3   3 | 3   3 | 3   2 | 3
  4 | 4   4 | 4   3 | 4   2 | 4
  5 | 5   4 | 5   3 | 5   2 | 5
```
storing the size of each component

That the name of $\text{Union}(A, B)$ is $A$ is problematic if $B$ is large, because then the cost for $\text{Union}$ is $O(n)$.

If the array $\text{size}$ stores the size of $A$ in $\text{size}[A]$, then $\text{Union}(A, B)$ is implemented as follows:

if $\text{size}[A] \geq \text{size}[B]$ then
  for all $b \in B$: $\text{Component}[b] := A$
  $\text{size}[A] := \text{size}[A] + \text{size}[B]$
else
  for all $a \in A$: $\text{Component}[a] := B$
  $\text{size}[B] := \text{size}[A] + \text{size}[B]$

The name of the component of $\text{Union}(A, B)$ is the name of the largest set.
bounding the cost of \texttt{Union}

\section*{Theorem}
Consider the array implementation for the Union-Find data structure for a set $S$ of size $n$, where unions take the name of the largest set. Any sequence of $k$ \texttt{Union} operations takes at most $O(k \log(k))$ time.

\textbf{Proof.} After $k$ \texttt{Union} operations, at most $2k$ vertices are touched. For example, take in each \texttt{Union} a different pair of vertices.

Consider a $v \in S$, because \texttt{Union} takes the name of the largest set, the size of the set containing $v$ at least doubles. The largest set after $k$ \texttt{Union} operations has at most $2k$ vertices. The last two statements imply that \texttt{Component}[v] has been updated at most $\log_2(2k)$ times.

At most $2k$ elements are involved in the sequence of \texttt{Union} operations, at most $\log_2(2k)$ times. This gives the bound $O(k \log(k))$. Q.E.D.
Exercise 2:
Consider the following problem:

Input: \( G = (V, E) \), a graph with \( n = \#V \) and \( m = \#E \).
Output: a set of Union-Find data structures, for every connected component of \( G \) there is exactly one Union-Find data structure.

1. Define an algorithm to solve the above problem. Write the elementary steps in the algorithm with the operations on a Union-Find data structure.
2. Prove the correctness of your algorithm.
3. Express the running time of your algorithm in \( n \) and \( m \). Justify the formula for the running time.
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
implementing Kruskal’s algorithm

Theorem

On $G = (V, E)$, a connected graph with $n = \#V$ and $m = \#E$, Kruskal’s algorithm runs in time $O(m \log(n))$.

Proof. The edges are sorted by increasing cost, which takes $O(m \log(m))$ time. As $m \leq n^2$, $O(m \log(m))$ is $O(m \log(n))$.

On each edge $(u, v)$, $\text{Find}(u)$ and $\text{Find}(v)$ is applied and if their components are different, $\text{Union}(\text{Find}(u), \text{Find}(v))$ is executed.

As $m = \#E$, at most $2m$ $\text{Find}$ operations are done and each $\text{Find}$ takes $O(1)$ time, so the total cost of all $\text{Find}$ executions is $O(m)$.

As $n = \#V$, $n - 1$ $\text{Union}$ operations are done. By the previous theorem, a sequence of $n - 1$ $\text{Union}$ operations is $O(n \log(n))$.

As $G$ is connected, $m \geq n - 1$, $O(n \log(n))$ is $O(m \log(n))$ and $O(m) + O(m \log(n))$ is $O(m \log(n))$. Q.E.D.
Spanning Trees

1 Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2 The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3 Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
the Heap or Priority Queue

A *complete* binary tree is a *heap* if

1. the root is the largest element; and
2. the subtrees are also heaps.

If the root is largest, we have a *max* heap. If the root is smallest, we have a *min* heap.

The root is called the *top* of the heap.

The *bottom* of the heap is the rightmost element at the deepest level of the tree.
storing a heap with an array

For node at $p$: left child is at $2p + 1$, right child is at $2p + 2$. Parent of node at $p$ is at $(p - 1)/2$. 
pushing to a heap and popping from a heap

The algorithm to push an item to a heap:

place the item at the new bottom
while the item is larger than the parent do
    swap the item with the parent.

The algorithm to pop an item from a heap:

remove the item, replace it with the bottom B
while B has larger children do
    swap B with its larger child.

Theorem

The cost of the push or pop heap algorithms is linear in the depth of the tree, or equivalently, logarithmic in the number of items stored.

Application: $n$ numbers can be sorted in $O(n \log(n))$. 
Spanning Trees

1. Spanning Trees
   - the minimum spanning tree problem
   - three greedy algorithms
   - analysis of the algorithms

2. The Union-Find Data Structure
   - managing an evolving set of connected components
   - implementing a Union-Find data structure
   - implementing Kruskal’s algorithm

3. Priority Queues
   - managing priorities
   - implementing Prim’s algorithm
implementing Prim’s algorithm

Prim’s algorithm adds the vertex with the smallest attachment cost. As $S$ grows, the attachment cost for vertices in $V \setminus S$ changes. All $v \in V \setminus S$ are stored in a heap with keys $a(v) = \min_{(u, v) \in E} c(u, v)$. We need two operations on a heap $H$ of size $n$:

- **ExtractMin**($H$) pops the element with the smallest value from $H$ in $O(\log(n))$ time.
- **ChangeKey**($H$, $v$, $\alpha$) changes the value of the key $v$ to $\alpha$ in $O(\log(n))$ time.
The cost of Prim’s algorithm

Theorem

On \( G = (V, E) \), a connected graph with \( n = \#V \) and \( m = \#E \), Prim’s algorithm runs in time \( O(m \log(n)) \).

Proof. A priority queue stores attachment costs of vertices as keys.

As \( n = \#V \), the algorithm takes \( n - 1 \) steps and in each step does a \text{ExtractMin} in \( O(\log(n)) \) time. So we have \( O(n \log(n)) \).

As \( m = \#E \), the \text{ChangeKey} happens at most once per edge, and costs \( O(\log(n)) \), which leads to \( O(m \log(n)) \).

As \( G \) is connected, \( m \geq n - 1 \), \( O(n \log(n)) \) is \( O(m \log(n)) \).

Thus the total cost is \( O(m \log(n)) \). Q.E.D.