Unit 11: Greedy Algorithms

Agenda:

- Greedy Algorithms
- The Minimum Spanning Tree (MST) problem
- Prim's Algorithm
- Kruskal's Algorithm
  - Union-Find data structure

Reading:

- CLRS Ch. 16: 414-428
- CLRS Ch. 23: 624-642

The Greedy Paradigm

- Iterative algorithms where:
  - At each iteration we take a simple greedy choice
    — an easy / most-rewarding / least-costly choice
  - We commit to this choice and *never* revoke it
- While the algorithm is iterative (loop-based)...
- .. the mind-set when designing the algorithm is recursive.
  (The iterative version is mostly there for the purpose of a faster runtime.)
- The basic recursive approach. Given the input I:
  1. Make the greedy/easy choice from I
  2. Commit to this choice — build I' the result of removing the greedy choice from I as well as anything in conflict with it. Recurse on I'.
- For the greedy approach to work we therefore need two properties:
  - Recursion is a valid approach to solve the problem
    Subproblem Optimality: After committing to a choice, we get a subproblem that is like the original problem (same input, same notion of optimal solution).
  - It is possible to take a simple/greedy choice and remain optimal:
    Substitution Property: There always exists an optimal solution containing our "easy" choice.
    If we picked e, then there's an optimal solution S such that $e \in S$.
    "Substitution": We assume $e \notin S$, so we substitute some $e' \in S$ with w, creating a different optimal solution $S' = S \setminus \{e\} \cup \{e\}$.

Simple Example: Largest Subset of Size k

- Input: U: a set of n numbers $v_1, ..., v_n$. An integer k.
  Output: A subset $A \subset U$ of size k with largest total value $(v(A) = \sum_{x \in A} x)$
- The greedy approach

```
procedure LargestSubset(U, k)
if (k > 0) then
    x ←FindMax(U)
    return( {x} ∪ LargestSubset(U \ {x}, k − 1) )
```

- In this simple example, committing to taking x results in a very simple instance (all elements but x).
- Instead of repeatedly finding the max-element (in time $O(n)$) in each level of the recursion, we use an iterative algorithm.

```
procedure LargestSubset(U, k)
Build Priority-Queue Q over U by item value
A ← ∅
for (i from 1 upto k) do
    x ←ExtractMax(Q)
    A ← A ∪ {x}
return A
```

- Runtime $O(n + k\log(n))$.

Simple Example: Largest Subset of Size k

- procedure LargestSubset(U, k)
  Build Priority-Queue Q on U by item values
  $A \leftarrow \emptyset$
  for (i from 1 upto k) do
      $x \leftarrow$ExtractMax(Q)
      $A \leftarrow A \cup \{x\}$
  return A
- To prove correctness, use the loop invariant:
  "At every iteration, there exists an optimal set S such that $A \subset S$."
  - Initially: $A = \emptyset$. LI holds for any optimal S.
  - Maintenance: Assume LI holds at the beginning of some iteration, we show it also holds at the beginning of the following iteration.
  - We start the iteration with A and add x. LI: exists optimal S s.t. $A \subset S$.
  - If $x \in S$ we are done: $A \cup \{x\} \subset S$.
  - So assume $x \notin S$. Take any $y \in S \setminus A$. Since $y \notin A$ then $y \in Q$. Since x is a largest element in Q we have $v(x) \geq v(y)$.
  - So take $S' = S \setminus \{y\} \cup \{x\}$. This is the substitution.
  - S' is a different set with k elements and $v(S') = v(S) - v(y) + v(x) \geq v(S)$ so S' is also optimal set. Now $A \cup \{x\} \subset S'$.
  - Termination: for-loop clearly terminates (i only increases).
  - Conclusion (termination #2): when done, $A \subset S$ where S is an optimal solution and $|A| = |S| = k$, so $A = S$ and we return an optimal set. ■

Minimum Spanning Tree (MST) problem:

- Input: simple, undirected connected graph G with weights on the edges: $w : E(G) \to \mathbb{R}_{\geq 0}$.
- Notions:
  - subgraph, forest = acyclic graph, tree
    (subgraph $G' = (V, E')$, where $E' \subset E$)
  - spanning subgraph: subgraph including all the vertices
  - spanning tree: spanning subgraph which is a tree — acyclic, connected, has exactly $n - 1$ edges
    e.g., BFS/DFS-tree is a spanning tree of the graph
  - minimum spanning tree: minimum weight on tree edges
- The MST Problem: Find a minimum spanning tree for the input graph.
  - Find **a** MST, not the MST — there could be more than one.
  - Example: all weights are the same, both BFS/DFS produce a MST...
- Important for:
  - Min-cost set of edges that we need so that all vertices can reach one another
  - Simple reachability: on a tree always exists a *unique* $u \to v$ path.
  - Learning value: a canonical example for greedy algorithms.
  - Useful info derived from the MST algorithms...
- The Minimum Spanning Forest problem:
  If the given graph is not necessarily connected: find MST for each CC.

Greedy algorithms and MST problem:

- Greedy algorithms:
  - greedy — each step makes the best choice (locally minimum)
  - and don't look back
  - Optimal substructure: an optimal solution to the original problem contains within it optimal solutions to subproblems:
    T is MST for $G = (V, E) \Rightarrow$ for any $U \subset V$ where $T[U]$ is connected, $T[U]$ is a min-spanning tree.
- The general MST algorithm outline:
  1. A is a set of "safe" edges: they are contained in some MST T
  2. $A = \emptyset$ initially
  3. `while` $(|A| < n - 1)$ `do`: find a safe edge $e = (u, v)$ and set $A \leftarrow A \cup \{e\}$.

  - What happens when we halt?
  - A is a MST since $A \subset T$ and both has size $n - 1$.
- Two greedy solutions
  - Prim's Algorithm (Actually: Prim + Dijkstra + Boruvka)
    Grow T vertex-wise: A is always a MST on some $S \subset V$
  - Kruskal's (Actually: Kruskal + Boruvka)
    Grow T edge-wise: A is always a minimal acyclic set of edges (forest)

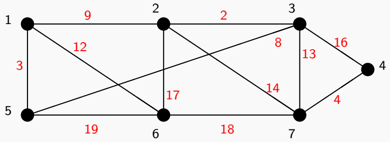
Prim's algorithm for the MST problem:

- Input: an edge-weighted (simple, undirected, connected) graph (positive weights)
- Output: a MST
- Idea:
  - Suppose we have already an MST A spanning subset S of vertices (Initially: S =a single vertex, $A = \emptyset$)
  - Grow A to span one more vertex $v \in \bar{S} = V \setminus S$ by adding a single edge (u, v) for some $u \in S$ and $v \notin S$.
  - Which edge to pick?
  - Greedy! min-weight edge from all possible edges crossing the $(S, \bar{S})$ cut.
- First sketch:

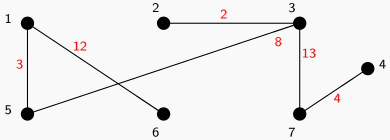
```
procedure PrimMST(G, w)            ** G = (V, E)
S ← {s}                            ** for an arbitrary start vertex s
A ← ∅
while (|S| < |V|) do
   find a minimum weight edge e = (u, v):   u ∈ S and v ∉ S
   S ← S ∪ {v}
   A ← A ∪ {e}
return A
```

Prim's algorithm for the MST problem — an example:

- Input graph G:



- $\texttt{primMST}(G, w, 1)$ returns:



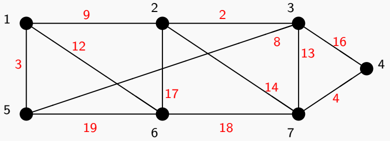
- First we prove correctness of Prim's algorithm
- Then we improve the naïve algorithm to reduce runtime
  Not surprisingly, finding the min-edge quickly is going to be useful (heaps!)

Prim's algorithm for the MST problem — correctness:

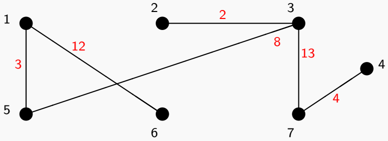
- In the proof we basically show the substitution property.
- LI: There exists some MST T such that $A \subset T$.
- At any iteration, we begin with $A \subset T$ for some MST T. We show that there exists a MST T' that contains $A \cup \{e\}$.
- If $e \in T$, we are done.
- O/w, T — a spanning tree — connect u with v. Since $u \in S$ whereas $v \notin S$, so on the unique $u \to v$ path there has to be some edge $e' = (u', v')$ that crosses the $(S, \bar{S})$-cut (with $u' \in S$ and $v' \in \bar{S}$).
  Clearly, $w(e) \leq w(e')$.
  We argue $T' = T \setminus \{e'\} \cup \{e\}$ is spanning V; and as it has $n-1$ edge it has to a spanning tree, with cost $\leq w(T)$.
- In fact, it is enough to argue u' and v' remain connected:
  Any $x \to y$ path on T either avoids the edge e' or goes through e'. In the former case, the $x \to y$ path remains in the T'; in the latter case — use the new $u' \to v'$ path to connect x with y.
- So why do u' and v' remain connected?
- Well, in T there's a $u \to v$ path that e' was a part of. So u is connected to u' and v' is connected to v. So, $u' \to \underbrace{u, v}_{e} \to v'$ is a path connecting u' and v' in T'. ■

Prim's algorithm for the MST problem — faster implementation:

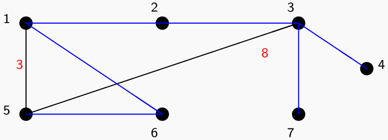
- Example: input graph G:

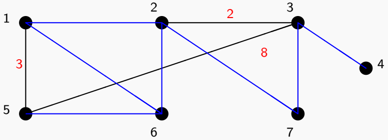


- $\texttt{primMST}(G, w, 1)$ returns:



- $\texttt{primMST}(G, w, 1)$: an intermediate tree
  Already picked black edges spanning 1,5,3; what are the candidate edges?





Prim's algorithm for the MST problem — faster implementation:

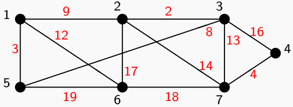
- Idea:
  For each node $\notin S$ — keep track of the min edge that connects it to S.
  Update the information only for the neighbors of the node that is currently being added to S.
  Uses a priority queue Q on $\bar{S}$ (so S is implicit — all the nodes **not** in Q)
- Pseudocode:

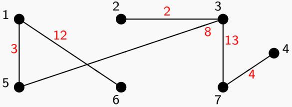
```
procedure primMST(G)                    **G = (V, E)
for each v ∈ V(G) do
    v.key ← ∞
    v.predec ← NIL
s.key ← 0                               **for some arbitrary start vertex s
Initialize a min-Priority Queue Q on V using key
while (Q ≠ ∅) do
    u ← ExtractMin(Q)                         **s extracted first
    foreach v neighbor of u do
        if (v ∈ Q and w(u, v) < v.key) then
            v.predec ← u
            decrease-key(Q, v, w(u, v))   **v.key is now w(u, v)
```

Prim's algorithm for the MST problem — faster implementation:

- Analysis of the improved algorithm:
  - correctness: almost done — need to prove that ExtractMin(Q) does extract a node v such that the edge $(v, v.predec)$ is a minimum weight edge that crosses the $(Q, \bar{Q})$-cut. (Simple proof by contradiction)
  - running time: $\Theta\Big(n + \sum_{u \in V} \Big(\log(n) + t_{\texttt{FindNeighbors}}(u) + \deg(u) \cdot \log(n)\Big)\Big)$
    –so: $\Theta((n+m)\log n) = \Theta(m \log(n))$ — adjacency list graph representation for a connected graph $m \geq n-1$.
    –or $\Theta(n^2 + m \log(n))$ in the adjacency matrix representation.
- Using a more sophisticated data-structure (Fibonacci heap), runtime of Prim is reduced to $O(n \log(n) + m)$.

What does Prim Teach Us?

- Theorem: Let T be a spanning tree of G.
  Then T is a MST iff each edge is the min-edge crossing the cut it induces.
- I.e., take T, and some $e \in T$. Removing e from T disconnects the tree and creates two components $C_1, C_2 = V \setminus C_1$.
  Then the claim is that e is a minimal edge out of all the edges the cross the (C_1, C_2)-cut.
  Moreover, this is true for all edges $e \in T$.
- E.g.: 

  MST: 

  min-weight of edge crossing $(\{1,5,6\},\{2,3,4,7\})$-cut $= 8$
  min-weight of edge crossing $(\{1,2,3,5,6\},\{4,7\})$-cut $= 13$
- Proof. $\Rightarrow$ If T is a MST, pick any e. ASOC e isn't a minimum edge crossing the cut it induces – replace e with an edge of strictly smaller weight. The resulting graph is a spanning tree (has $n-1$ edges and spans V) with cost strictly smaller than T. Contradiction.
- $\Leftarrow$ Given some tree T where all edges in T satisfy this property — how do we prove it is a MST?
- Use Prim!

What does Prim Teach Us?

- Theorem: Let T be a spanning tree of G.
  Then T is a MST iff each edge is the min-edge crossing the cut it induces.
- $\Leftarrow$ If all edges in T satisfy this property — how do we prove it is a MST?
- Use Prim!
  Claim: there's an instantiation of Prim that builds T.
  In other words: we can run Prim and maintain the invariant that $T[S]$ is a single connected component ($T[S]$ is a spanning tree of S).
- Proof by induction on S. Clearly true for $|S| = 1$ and $T[S]$ is empty.
- The induction step:
  - Suppose in the transition from S to $S \cup \{v\}$ Prim picks an edge $e = (u, v)$ of weight w. Let $e_1 = (a_1, b_1), e_2 = (a_2, b_2), ..., e_k = (a_k, b_k)$ be all the edges in T with one vertex (a_i) in S and one vertex (b_i) in $\bar{S}$.
  - Since Prim picks the min-edge connecting S with $\bar{S}$, we have $w(e) \leq w(e_1), w(e) \leq w(e_2), ..., w(e) \leq w(e_k)$.
  - ASOC $w(e) < \min\{w(e_1), w(e_2), ..., w(e_k)\}$ — the weight of the edge chose by Prim is strictly smaller than all of the edges in T that leave S.
  - Look at the path $v \to u$ on T. It starts at $\bar{S}$ and ends at S, so it must use some edge e_j. So the removal of e_j separates u from v.
  - Hence e_j isn't the min-edge that separates the cut (e also crosses the same cut). Contradiction!
  - Thus $w(e) = \min\{w(e_1), ...w(e_k)\} = w(e_j)$.
  - Instantiate the priority-queue of Prim to pick b_j rather than v (both have the same key, so break ties in favor of b_j rather than v).

Kruskal's algorithm for the MST problem:

- Input: an edge-weighted (simple, undirected, connected) graph (positive weights)
- Output: an MST
- Idea:
  - Expand the set A of safe edges with one edge at the time.
  - $A \subset T$, so A is always acyclic / a forest.
  - Which edge to add to A?
  - Greedy! Minimum weight edge e that keeps $A \cup \{e\}$ acyclic.
- (First draft) Psuedocode:

  procedure kruskal(G)

  $A \leftarrow \emptyset$

  sort edges in $E(G)$ in a non-decreasing weight order

  foreach edge $e = (u, v)$ do

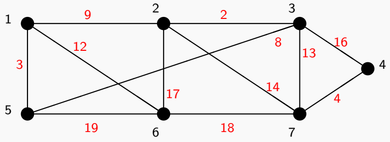
      if (e doesn't close a cycle with A) then

          $A \leftarrow A \cup \{e\}$

  return A
- Like before: first example, then correctness, then runtime.

Kruskal's algorithm for the MST problem — An Example:

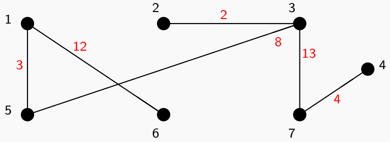
- An example:



- Sorting the edges:

2	3	4	8	9	12	13	14	16	17	18	19
(2,3)	(1,5)	(4,7)	(3,5)	(1,2)	(1,6)	(3,7)	(2,7)	(3,4)	(6,2)	(6,7)	(5,6)

- `kruskalMST`(G) returns:



Kruskal's algorithm for the MST problem — Correctness:

- First note, we only need to consider each edge once:
  If it closes a cycle with A now, later we only append edges to A, so it will still close the same cycle.
- Proof of correctness: Assuming there exists a MST T such that $A \subset T$ we show that there exists a MST T' such that $A \cup \{e\} \subset T'$.
- If $e \in T$, we are done.
- Otherwise, $T \cup \{e\}$ contains n edges, so it closes a cycle.
- Pick some edge on this cycle that doesn't belong to $A \cup \{e\}$. (There must be one, as $A \cup \{e\}$ is acyclic). Call it e'.
- Note: $A \cup \{e'\}$ doesn't have a cycle since only by adding e we closed this cycle. But Kruskal picked e over e', so $w(e') \geq w(e)$.
- Finally, as this is a cycle, all nodes on it remain connected once e' is removed.
- Therefore $T' = T \setminus \{e'\} \cup \{e\}$ is a tree: has $n-1$ edges and it connects all vertices.
  All vertices in V are connected to some vertex on this cycle, all vertices in this cycle remain connected by replacing e' with e, so T still spans V. $\blacksquare$

- What about runtime analysis?

Kruskal's algorithm for the MST problem:

- Idea for runtime improvement
  - Avoid the need to search for a cycle, it's enough to know a cycle exists.
  - When does $e = (u, v)$ closes a cycle with A?
  - When u and v are already connected by A.
  - I.e. when u and v are in the same connected component.
  - We need each vertex to quickly point us to its CC label
  - ...and we need a way to quickly update CC labels:
    When we put the edge (u, v) then all vertices in $CC(u)$ and $CC(v)$ should have the same label from now on.
- procedure kruskal (G)

```
A ← ∅
foreach v ∈ V(G) do
   set-singleton-cluster CC(v) ← {v}
sort edges in E in a non-decreasing weight order
foreach edge e = (u,v) do
   if (CC(u) ≠ CC(v)) then
      A ← A ∪ {e}
      union clusters CC(u) and CC(v)
return A
```

- How to implement this?

Union-Find, a data-structure for Kruskal:

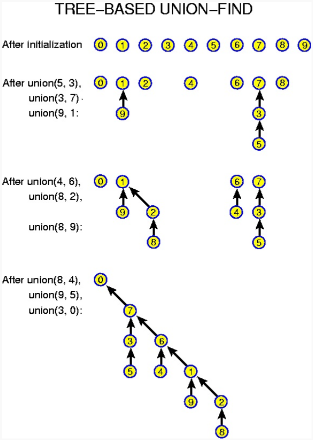
- The idea: maintain a set of rooted trees, one per CC. Each node points to a predecessor, and the root's predecessor is itself.
- Needs to support: `createSingleton`(v); `findCC`(v); `union`(u, v)

  - `procedure createSingleton`(v)
    $v.predec \leftarrow v$

    $O(1)$-time

  - `procedure findCC`(v)
    `if` $(v.predec = v)$ `then`
    `return` v
    `else`
    `return findCC`$(v.predec)$

  - `procedure union`(u, v)
    $x \leftarrow$ `findCC`(u)
    $y \leftarrow$ `findCC`(v)
    $x.predec \leftarrow y$



- What's the worst-case runtime of `findCC()`?
- The longest leaf $\rightarrow$ root path in this rooted forest.

Union-Find, a data-structure for Kruskal:

- The idea: maintain a set of rooted tree, one per CC. Each node points to a predecessor, and the root's predecessor is itself.
- We want to make the longest leaf$\rightarrow$root path short as possible.
- The path only increases with each `union()` call.
- Each root maintains $rank \stackrel{\text{def}}{=}$ longest path from any of its leafs.
- Set predecessors based on rank.
  - <u>`procedure createSingleton`(v)</u>
    $v.predec \leftarrow v$
    $v.rank \leftarrow 0$
  - <u>`procedure union`(u, v)</u>
    $x \leftarrow$ `findCC(`u`)`
    $y \leftarrow$ `findCC(`v`)`
    `if` $(x.rank > y.rank)$ `then`
        $y.predec \leftarrow x$ **$x.rank$ unchanged:
        **longest leaf$\rightarrow x$ path — same as before `union()`
    `else if` $(x.rank < y.rank)$ `then`
        $x.predec \leftarrow y$
    `else` **$x.rank = y.rank$
        $y.predec \leftarrow x$
        $x.rank \leftarrow x.rank + 1$

Union-Find, a data-structure for Kruskal:

- Each root maintains $rank \stackrel{\text{def}}{=}$ longest path from any of its leafs. Set predecessors based on rank.
- Denote $RANK = \max rank$ of all roots.
- Claim: After n calls to `union()`, $RANK \leq \log(n+1)$.
- Proof: Denote $R(k)$ — minimal number of `union()` calls needed to get $RANK = k$. Then $R(0) = 0$, and $R(k) = 2R(k-1) + 1$ because we need to construct *two* $rank = k - 1$ trees on *two* disjoint set of vertices in order to be able to build one tree of $rank = k$.
- This recursion solves to: $R(k) = 2^k - 1$.
- So with $n - 1$ `union()` calls we can only make $RANK$ be $\log(n)$.

- Conclusion: Runtime of Kruskal's algorithm, including sorting m edges and $O(m)$ calls to `findCC()`

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

because $n - 1 \leq m \leq \binom{n}{2}$

What Does Kruskal Teach Us?

- <u>Thm:</u> Let T be a spanning tree of G.
  Then T is MST iff each non-tree edge is the heaviest on the cycle it closes with T.
- T spans all nodes. For any u, v there's a path from u to v in T.
- Take any non-tree edge $e = (u, v)$. We take the path from u to v in T, then the non-tree edge e and get back to u. Hence, we closed a cycle.
- Our claim: for any $e \notin T$, $w(e) \geq w(e')$ for any e' on the path in T that connects the endpoint of e.
  - Proof: First the easy direction.
  - Assume T is a MST. ASOC that there exists some $e = (u, v) \notin T$ for which $w(e) < w(e')$ for some edge $e' \in T$ on the $u \to v$ path in T.
  - Construct the tree $T' = T \setminus \{e'\} \cup \{e\}$. (Take e' out of T, add e to T.)
  - It is a spanning tree: take any two nodes x and y. If the path in T didn't use e' it is still there. If the path $x \to y$ uses $e' = (u', v')$ we just walk from $x \to u'$, from $u' \to u$ (that path still lies in T), from u to v using e, and from $v \to v' \to y$ using only edges that remain in T.
  - T' is thus a spanning tree, whose cost is strictly smaller than the cost of T. Contradiction.

  - How will we prove the opposite direction?
  - Suppose T has this property, that for any non-edge $e \notin T$ we have $w(e) \geq w(e')$ for each e' on the cycle created by $T \cup \{e\}$.
    How do we show T is a MST?

What Does Kruskal Teach Us?

- Thm: Let T be a spanning tree of G.
  Then T is MST iff each non-tree edge is the heaviest on the cycle it closes with T.
  - Suppose T has this property, that for any non-edge $e \notin T$ we have $w(e) \geq w(e')$ for each e' on the cycle created by $T \cup \{e\}$. How to show T is a MST?
  - We argue that we can sort the edges in a way such that Kruskal returns T.
  - Sort the edges:
    - Clearly, for any $x < y$, all edges that have weight x must appear before all the edges with weight y.
    - The point is this: how to arrange all the edges that have the same weight x?
    - Answer: put first all edges in T that have the weight x, and only after those put the edges $\notin T$ of weight x.
  - We argue that according to this ordering, Kruskal picks exactly the edges that belong to T and no edge that doesn't belong to T.
  - The formal loop invariant: at the beginning of each iteration, out of the edges traversed up to now, Kruskal picked *only* those that belong to T.
    - Initially: no edges have been considered yet, holds vacuously.
    - Maintenance: consider the edge in this iteration, e.
      If $e \in T$: it closes no cycle with all edges in T, let alone the subset of edges picked thus far by Kruskal. Thus Kruskal takes e.
      If $e \notin T$: we sorted the edges so that all edges that belong to T and have weight $\leq w(e)$ appear before the non-tree edge e. Moreover, as $w(e)$ is the largest on the cycle it closes with T, thus all other edges in this cycle appear before e. By the LI we have that e closes a cycle with the set of edges that Kruskal already picked and so Kruskal doesn't pick e.
    - Termination: the loop takes a finite amount of time, and so eventually ends. This means Kruskal's tree is composed of all edge in T, I.e., T. $\square$

Some Conclusions from Kruskal:

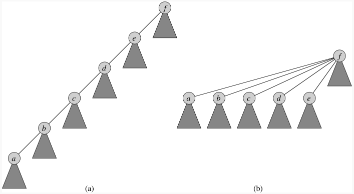
1. Claim: If all the weights in the graph are different, then the MST is unique.
    - Suppose not. Take two MSTs: T created by Kruskal, and T' some other MST. Take an edge $e \in T'$ but $e \notin T$.
    - Kruskal-type idea: $T \cup \{e\}$ closes a cycle, but for any other edge e' on this cycle, $T \cup \{e\} \setminus \{e'\}$ is a spanning tree.
    - Kruskal didn't pick the edge e. This means that the path connecting e's endpoints was in A there when Kruskal considered e, making e close a cycle.
    - Hence e is the heaviest edge on the cycle it creates in $T \cup \{e\}$ (all weights are unique so $w(e)$ is strictly greater than any weight of any other edge in this cycle).
    - All other edges on this cycle belong to A so they belong to T. Some edge $\tilde{e}$ on this cycle must not belong to the acyclic T'. Thus the spanning tree $T' \cup \{\tilde{e}\} \setminus \{e\}$ has strictly smaller weight then the MST T'. Contradiction.

2. Kruskal can be thought of as an *hierarchical clustering* algorithm, called *Singe-Linkage*.
   By looking at the connected components formed by the Kruskal forest.
    - Start with all singleton-clusters, merge two clusters with minimum edge between them.
    - Continue until a single cluster is formed.
    - Or if you want k clusters: continue until k clusters are formed (remove the $k-1$ heaviest edges in the MST).

OPTIONAL I: Better Union-Find:

- Turns out we can do much better in terms of Union-Find, through revising `findCC()`
- It is called path-compression

```
procedure findCC(v)
if (v.predec = v) then
    return v
else
    x ← findCC(v.predec)
    v.predec ← x
    return x
```



- Combining path-compression with $rank$ tricks leads to a really low amortized-cost runtime: at most $\alpha(n)$ (the inverse Ackerman function)
  E.g. $\alpha(2^{2048}) \leq 4$
  In particular $\alpha(n) \in o(\log^*(n))$
- This doesn't change Kruskal's asymptotic runtime. Kruskal requires we sort all edges so it is still $O(m\log(m))$.

OPTIONAL II: Greedy Algorithms and Matroids

- Greedy algorithms work when the possible sets one can pick are a *Matroid*
- Definition: Given a ground-set/universe U of elements, a set $\mathcal{M}$ of subsets of U is called a matroid if it satisfies the following properties:
  1. (Hereditary:) For any $A \in \mathcal{M}$ and any $B \subset A$ we have that $B \in \mathcal{M}$ (So definitely $\emptyset \in \mathcal{M}$)
  2. (Exchange property:) For any $A, B \in \mathcal{M}$ such that $|B| < |A|$ there exists some $x \in A$ which isn't an elements of B such that $B \cup \{x\} \in \mathcal{M}$.
- One can prove that when the problem's underlying structure is a matroid, then the greedy approach finds and optimal solution.
- You can check and see that (i) the collection of sets of lin. ind. vectors, (ii) the collection of sets of edges forming acyclic graphs are both matroids.
- Read more about matroids in CLRS Ch.16.4
- Also recommended: Huffman codes (Ch.16.3)

OPTIONAL III: Casting the MST Problem as a (Weighted) Linearly-Independent-Set Problem

- It is possible to cast down the MST problem as the problem of finding a set of $n-1$ linearly independent vectors.
- Given a graph on n nodes, first number the nodes: $v_1, ..., v_n$.
- For any edge $e = (v_i, v_j)$ where $i < j$ create a n-dimensional vector $\vec{u}_e$: put 1 in the i-th coordinate, -1 in the jth coordinate, 0 on the other $n-2$ coordinates.
- One can prove the a set of edges A is acyclic iff the set of corresponding vectors are linearly independent:
  - If there's a cycle in A, $(v_{i_0}, v_{i_1}, ..., v_{i_{t-1}}, v_{i_0})$ then for every edge $e_i = (v_{i_k}, v_{i_{k+1}})$ on this cycle set its λ_i coefficient to be 1 if $i_k < i_{k+1}$ or -1 if $i_k > i_{k+1}$. You can check and see that $\sum_i \lambda_i \vec{u}_{e_i} = \vec{0}$, so we have a non lin. ind. set.
  - If there isn't a cycle in A, we prove the corresponding set of vectors is lin. ind. by induction. Suppose that the vectors have some non-trivial linear combination that sums to $\vec{0}$. Pick a leaf in the forest created by A — the edge connecting the leaf has a 1 coordinate on some vertex where all other edges have a 0 coordinate (no other edge touches this leaf node). Thus, this vector corresponding to the leaf-edge cannot have a non-zero coefficient in this sum. So the sum is a combination of $|A|-1$ vectors, which by induction are linearly independent.
- So assign each vector $\vec{u}_e$ the weight of the edge e, and by finding a min-weight set of $n-1$ linearly independent vectors we find a min-weight MST.