Minimum Spanning Tree

Abhiram Ranade

February 3, 2016

Minimum (weight) spanning tree

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

- T is connected.

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

- T is connected.
- T spans G, i.e. each vertex in G is also present in T.

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

- T is connected.
- T spans G, i.e. each vertex in G is also present in T.
- Total weight $w(T) = \sum_{e \in T} w(e)$ of all the edges in T is as small as possible.

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

- T is connected.
- T spans G, i.e. each vertex in G is also present in T.
- Total weight $w(T) = \sum_{e \in T} w(e)$ of all the edges in T is as small as possible.

Application: Edges = potential ways of building roads, communication links. MST = how to build roads/links at minimum cost while ensuring everyone is connected, at least indirectly.

Minimum (weight) spanning tree

Input: Connected Graph G, weight $w(e) \geq 0$ for each edge e.
Output: Subgraph T of G such that

- T is connected.
- T spans G, i.e. each vertex in G is also present in T.
- Total weight $w(T) = \sum_{e \in T} w(e)$ of all the edges in T is as small as possible.

Application: Edges = potential ways of building roads, communication links. MST = how to build roads/links at minimum cost while ensuring everyone is connected, at least indirectly.

Suffices to look for a minimum weight spanning tree..

Connected spanning subgraphs and spanning trees

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

If not, remove edges until we get a tree.

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

If not, remove edges until we get a tree.

Fact: In general there exists a minimum weight spanning subgraph which is also a spanning tree.

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

If not, remove edges until we get a tree.

Fact: In general there exists a minimum weight spanning subgraph which is also a spanning tree.

Fact: Tree in an n vertex graph has $n-1$ edges $\Leftrightarrow$ it is a spanning tree.

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

If not, remove edges until we get a tree.

Fact: In general there exists a minimum weight spanning subgraph which is also a spanning tree.

Fact: Tree in an n vertex graph has $n-1$ edges $\Leftrightarrow$ it is a spanning tree.

Fact: If T is a spanning tree, and e is a non-tree edge. Then $T \cup \{e\}$ contains a cycle C.

Connected spanning subgraphs and spanning trees

Fact: If all weights are positive, then every minimum weight connected spanning subgraph must be a spanning tree.

If not, remove edges until we get a tree.

Fact: In general there exists a minimum weight spanning subgraph which is also a spanning tree.

Fact: Tree in an n vertex graph has $n-1$ edges $\Leftrightarrow$ it is a spanning tree.

Fact: If T is a spanning tree, and e is a non-tree edge. Then $T \cup \{e\}$ contains a cycle C.

Fact: (contd) Let e' be any edge in C. Then $T \cup \{e\} - \{e'\}$ is also a spanning tree.

A greedy strategy?

A greedy strategy?

Attempt: Since we wish to minimize total weight, let us pick the minimum weight edge to construct the MST.

A greedy strategy?

Attempt: Since we wish to minimize total weight, let us pick the minimum weight edge to construct the MST.

Can we prove greedy choice: will there exist an MST which will contains e?

A greedy strategy?

Attempt: Since we wish to minimize total weight, let us pick the minimum weight edge to construct the MST.

Can we prove greedy choice: will there exist an MST which will contains e?

Can we discover the remaining edges by recursing, i.e. is there optimal substructure?

Greedy choice

Greedy choice

Let e be a minimum weight edge in G.

Greedy choice

Let e be a minimum weight edge in G.
Suppose an MST T does not contain e.

Greedy choice

Let e be a minimum weight edge in G.
Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

Greedy choice

Let e be a minimum weight edge in G.
Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

Greedy choice

Let e be a minimum weight edge in G.
Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.
$w(T') = w(T) + w(e) - w(e') \leq w(T)$ $\quad$ e has min weight.

Greedy choice

Let e be a minimum weight edge in G.
Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.
$w(T') = w(T) + w(e) - w(e') \leq w(T)$ $\quad$ e has min weight.

T' is an MST containing e.

Optimal substructure

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

- Strengthen the induction: pose the problem as that of extending the spanning tree.
  "Given a weighted graph and some edges, find a minimum weight spanning tree that is required to contain the given edges."

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

- Strengthen the induction: pose the problem as that of extending the spanning tree.
  "Given a weighted graph and some edges, find a minimum weight spanning tree that is required to contain the given edges."

Most books.

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

- Strengthen the induction: pose the problem as that of extending the spanning tree.
  "Given a weighted graph and some edges, find a minimum weight spanning tree that is required to contain the given edges."

  Most books.
- Construct a new graph G' whose minimum spanning tree will be the same as the rest of the spanning tree of G given the first edge.

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

- Strengthen the induction: pose the problem as that of extending the spanning tree.
  "Given a weighted graph and some edges, find a minimum weight spanning tree that is required to contain the given edges."

  Most books.

- Construct a new graph G' whose minimum spanning tree will be the same as the rest of the spanning tree of G given the first edge.

  Next

Optimal substructure

The problem of extending a spanning tree, after some edges have been discovered, is different from the problem of finding a spanning tree when there are no edges.

Possible Fixes:

- Strengthen the induction: pose the problem as that of extending the spanning tree.
  "Given a weighted graph and some edges, find a minimum weight spanning tree that is required to contain the given edges."

  Most books.

- Construct a new graph G' whose minimum spanning tree will be the same as the rest of the spanning tree of G given the first edge.

  Next

G' : obtained by *contracting* a minimum weight edge.

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

- All edges of E which are not incident on u, v with their weights.

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

- All edges of E which are not incident on u, v with their weights.
- Edges $(u \cdot v, y)$ for every edge $(u, y), y \neq v$ with the same weight as the edge (u, y),

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

- All edges of E which are not incident on u, v with their weights.
- Edges $(u \cdot v, y)$ for every edge $(u, y), y \neq v$ with the same weight as the edge (u, y),
- Edges $(u \cdot v, y)$ for every edge $(v, y), y \neq u$ with the same weight as the edge (v, y)

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

- All edges of E which are not incident on u, v with their weights.
- Edges $(u \cdot v, y)$ for every edge $(u, y), y \neq v$ with the same weight as the edge (u, y),
- Edges $(u \cdot v, y)$ for every edge $(v, y), y \neq u$ with the same weight as the edge (v, y)

The above construction may introduce parallel edges, in which case we retain only the edge having the smaller weight.

Edge contraction

Let $e = (u, v)$ be an edge in $G = (V, E)$. Then G/e is the graph obtained from G by collapsing u, v into a single vertex $u \cdot v$ which inherits all edges of u, v with their weight.

Vertex set of G/e: $V - \{u, v\} \cup \{u \cdot v\}$
Edge set of G/e:

- All edges of E which are not incident on u, v with their weights.
- Edges $(u \cdot v, y)$ for every edge $(u, y), y \neq v$ with the same weight as the edge (u, y),
- Edges $(u \cdot v, y)$ for every edge $(v, y), y \neq u$ with the same weight as the edge (v, y)

The above construction may introduce parallel edges, in which case we retain only the edge having the smaller weight.

If $G' = G/e$, we will use $G' * e$ to mean G.

Edge contraction and spanning trees

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof:

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$:

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$:

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

(2) $T/(u, v)$ is a tree because:

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

(2) $T/(u, v)$ is a tree because:
$T/(u, v)$ spans $G/(u, v)$, and $|T/(u, v)| = |T| - 1 = 1$ less than the number of nodes in $G/(u, v)$.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

(2) $T/(u, v)$ is a tree because:
$T/(u, v)$ spans $G/(u, v)$, and $|T/(u, v)| = |T| - 1 = 1$ less than the number of nodes in $G/(u, v)$.

Lemma: Let T' be a spanning tree for $G' = G/(u, v)$. Then $T' * (u, v)$ is a spanning tree of G.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

(2) $T/(u, v)$ is a tree because:
$T/(u, v)$ spans $G/(u, v)$, and $|T/(u, v)| = |T| - 1 = 1$ less than the number of nodes in $G/(u, v)$.

Lemma: Let T' be a spanning tree for $G' = G/(u, v)$. Then $T' * (u, v)$ is a spanning tree of G. Proof: Exercise.

Edge contraction and spanning trees

Lemma: Let T be a spanning tree of a graph G containing edge (u, v). Then $T/(u, v)$ is a spanning tree of $G/(u, v)$.

Proof: (1) $T/(u, v)$ spans $G/(u, v)$ because:
Let p, q be any vertices in $G/(u, v)$.

- Case $p, q \neq u \cdot v$: T has a $p - q$ path P.
  $(u, v) \notin P$, $\Rightarrow P$ is a $p - q$ path in $T/(u, v)$.
  $(u, v) \in P \Rightarrow P/(u, v)$ will be a $p - q$ path in $T/(u, v)$.
- Case $p = u \cdot v$: T has $u - q$ path.
  $\Rightarrow T/(u, v)$ will have $(u \cdot v) - q$ path.

(2) $T/(u, v)$ is a tree because:
$T/(u, v)$ spans $G/(u, v)$, and $|T/(u, v)| = |T| - 1 = 1$ less than the number of nodes in $G/(u, v)$.

Lemma: Let T' be a spanning tree for $G' = G/(u, v)$. Then $T' * (u, v)$ is a spanning tree of G. Proof: Exercise.

$T' * (u, v) =$ tree induced in G by the edges of T' and (u, v).

Optimal substructure

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof:

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e)$

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'
$w(T) = w(T') + w(e)$

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'
$w(T) = w(T') + w(e) \leq w(T^O)$.

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'
$w(T) = w(T') + w(e) \leq w(T^O)$.
T is a spanning tree for G. Previous Lemma

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'
$w(T) = w(T') + w(e) \leq w(T^O)$.
T is a spanning tree for G. Previous Lemma
Hence T must be an MST.

Optimal substructure

Main claim: Let e be a minimum weight edge in $G = (V, E)$. Let T' be an MST for $G' = G/e$. Then $T = T' * e$ is an MST for G.

Proof: By greedy choice there exists MST T^O for G containing e.
$T'' = T^O/e$ is a spanning tree for G'. Previous Lemma
$w(T'') = w(T^O) - w(e) \geq w(T')$ minimality of T'
$w(T) = w(T') + w(e) \leq w(T^O)$.
T is a spanning tree for G. Previous Lemma
Hence T must be an MST.

Algorithm:
MST(G){
  if G has only 1 vertex return ϕ.
  $e =$ minimum weight edge in G.
  return $e || MST(G/e)$
}

Implementation

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Supervertex: set of vertices of G that have been merged together due to contraction.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Supervertex: set of vertices of G that have been merged together due to contraction.

During execution, the supervertices will form the vertex set of the graph we are dealing with.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Supervertex: set of vertices of G that have been merged together due to contraction.

During execution, the supervertices will form the vertex set of the graph we are dealing with.

When we contract edge (u, v) we must merge the corresponding supervertices into a single vertex.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Supervertex: set of vertices of G that have been merged together due to contraction.

During execution, the supervertices will form the vertex set of the graph we are dealing with.

When we contract edge (u, v) we must merge the corresponding supervertices into a single vertex.

Initially every vertex is to be considered a supervertex.

Implementation

Presorting: We can create a list of edges sorted by weight at the beginning, so that finding the min weight edge is easy.

Implicit contraction: We keep track of which vertices have been merged together without modifying the graph G.

Supervertex: set of vertices of G that have been merged together due to contraction.

During execution, the supervertices will form the vertex set of the graph we are dealing with.

When we contract edge (u, v) we must merge the corresponding supervertices into a single vertex.

Initially every vertex is to be considered a supervertex.

Classic "union-find" data structure.

The (disjoint) union-find data structure

The (disjoint) union-find data structure

Operations allowed:

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"
- Contraction of $(u, v) = Union(Find(u), Find(v))$.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"
- Contraction of $(u, v) = Union(Find(u), Find(v))$.
- We do not need to eliminate "parallel edges" because:

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"
- Contraction of $(u, v) = Union(Find(u), Find(v))$.
- We do not need to eliminate "parallel edges" because:
  - The algorithm will anyway consider the edge with the smaller weight first.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"
- Contraction of $(u, v) = Union(Find(u), Find(v))$.
- We do not need to eliminate "parallel edges" because:
  - The algorithm will anyway consider the edge with the smaller weight first.
- Contracting one out of two parallel edges (u, v) may produce self loops.

The (disjoint) union-find data structure

Operations allowed:

- $Make - Set(v)$: Creates a set consisting of a single element v.
- $Find(v)$: Returns the set containing v.
- $Union(S, S')$: merges S, S' together into a set S''. Calling Find on elements in S, S' would now return S''.

How we use this for MST:

- Initially for all v call $Make - Set(v)$. "Supervertex"
- Contraction of $(u, v) = Union(Find(u), Find(v))$.
- We do not need to eliminate "parallel edges" because:
  - The algorithm will anyway consider the edge with the smaller weight first.
- Contracting one out of two parallel edges (u, v) may produce self loops.
  - Before considering an edge (u, v) for contraction, we check that $Find(u) \neq Find(v)$, i.e. that it is not a self-loop.

Implementing union-find data structure

Implementing union-find data structure

Each set holds its elements in a linked list.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Union(s, s'): (1) Instead of creating a new set s'', we merge the smaller of s, s' into the larger. (2) We update the length of each set. (3) We update $S[]$ to point to the correct set.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Union(s, s'): (1) Instead of creating a new set s'', we merge the smaller of s, s' into the larger. (2) We update the length of each set. (3) We update $S[]$ to point to the correct set.

Time for union operation: Steps (1),(2) take $O(1)$ per operation.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Union(s, s'): (1) Instead of creating a new set s'', we merge the smaller of s, s' into the larger. (2) We update the length of each set. (3) We update $S[]$ to point to the correct set.

Time for union operation: Steps (1),(2) take $O(1)$ per operation.
Step 3: We charge vertex i everytime we change $S[i]$.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Union(s, s'): (1) Instead of creating a new set s'', we merge the smaller of s, s' into the larger. (2) We update the length of each set. (3) We update $S[]$ to point to the correct set.

Time for union operation: Steps (1),(2) take $O(1)$ per operation.
Step 3: We charge vertex i everytime we change $S[i]$.
Every time $S[i]$ changes, the size of $S[i]$ at least doubles.

Implementing union-find data structure

Each set holds its elements in a linked list.
For each set we also maintain its size.
We maintain array $S[1..n]$: $S[i]$ points to set containing vertex i.

Make-set(v): $O(1)$ time

Find(v): $O(1)$ time using S.

Union(s, s'): (1) Instead of creating a new set s'', we merge the smaller of s, s' into the larger. (2) We update the length of each set. (3) We update $S[]$ to point to the correct set.

Time for union operation: Steps (1),(2) take $O(1)$ per operation.
Step 3: We charge vertex i everytime we change $S[i]$.
Every time $S[i]$ changes, the size of $S[i]$ at least doubles.
Thus total time spent on vertex i is $O(\log |S[i]|)$.

Algorthm time analysis

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){

  if G has only 1 vertex return ϕ.

  $e =$ minimum weight edge in G.

  return $e||MST(G/e)$

}

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){

  if G has only 1 vertex return ϕ.

  $e =$ minimum weight edge in G.

  return $e||MST(G/e)$

}

Preprocessing: Construct A = edges sorted by weight.

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){

if G has only 1 vertex return ϕ.

$e =$ minimum weight edge in G.

return $e||MST(G/e)$

}

Preprocessing: Construct A = edges sorted by weight.

Preprocessing Time: $O(m \log m) = O(m \log n)$

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){
  if G has only 1 vertex return ϕ.
  $e =$ minimum weight edge in G.
  return $e||MST(G/e)$
}

Preprocessing: Construct A $=$ edges sorted by weight.
Preprocessing Time: $O(m \log m) = O(m \log n)$
Time for 1 Find: $O(1)$.

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){

if G has only 1 vertex return ϕ.

$e =$ minimum weight edge in G.

return $e||MST(G/e)$

}

Preprocessing: Construct A = edges sorted by weight.

Preprocessing Time: $O(m \log m) = O(m \log n)$

Time for 1 Find: $O(1)$. Total find time: $O(m)$.

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){
  if G has only 1 vertex return ϕ.
  $e =$ minimum weight edge in G.
  return $e||MST(G/e)$
}

Preprocessing: Construct A = edges sorted by weight.
Preprocessing Time: $O(m \log m) = O(m \log n)$
Time for 1 Find: $O(1)$. Total find time: $O(m)$.
Union time: $\sum_i \log |S[i]| = O(n \log n)$

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){

  if G has only 1 vertex return ϕ.

  $e =$ minimum weight edge in G.

  return $e||MST(G/e)$

}

Preprocessing: Construct A = edges sorted by weight.

Preprocessing Time: $O(m \log m) = O(m \log n)$

Time for 1 Find: $O(1)$. Total find time: $O(m)$.

Union time: $\sum_i \log |S[i]| = O(n \log n)$

Total : $O(m \log n)$

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){
  if G has only 1 vertex return ϕ.
  $e =$ minimum weight edge in G.
  return $e||MST(G/e)$
}

Preprocessing: Construct A = edges sorted by weight.
Preprocessing Time: $O(m \log m) = O(m \log n)$
Time for 1 Find: $O(1)$. Total find time: $O(m)$.
Union time: $\sum_i \log |S[i]| = O(n \log n)$

Total : $O(m \log n)$

Improvements in data structure: Possible, using "path compression" ideas. This will make the overall time in union operations essentially $O(m)$.

Algorthm time analysis

Algorithm: (Kruskal)

MST(G){
  if G has only 1 vertex return ϕ.
  $e =$ minimum weight edge in G.
  return $e||MST(G/e)$
}

Preprocessing: Construct A = edges sorted by weight.
Preprocessing Time: $O(m \log m) = O(m \log n)$
Time for 1 Find: $O(1)$. Total find time: $O(m)$.
Union time: $\sum_i \log |S[i]| = O(n \log n)$

Total : $O(m \log n)$

Improvements in data structure: Possible, using "path compression" ideas. This will make the overall time in union operations essentially $O(m)$.

Improved data structure does not help if we need to presort.

Another algorithm for MST

Another algorithm for MST

Let us examine the central greedy choice idea again.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

$w(T') = w(T) + w(e) - w(e') \leq w(T)$ e has min weight.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

$w(T') = w(T) + w(e) - w(e') \leq w(T)$ $\quad$ e has min weight.

T' is an MST containing e.

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

$w(T') = w(T) + w(e) - w(e') \leq w(T)$ e has min weight.

T' is an MST containing e.

For what other choices of e will it work?

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

$w(T') = w(T) + w(e) - w(e') \leq w(T)$ e has min weight.

T' is an MST containing e.

For what other choices of e will it work?

Rule 2: Minimum weight edge leaving any vertex

Another algorithm for MST

Let us examine the central greedy choice idea again.

Greedy choice

Let e be a minimum weight edge in G. "Rule 1"

Suppose an MST T does not contain e.

Let $T' = T \cup \{e\} - \{e'\}$, where $e' \in C, e' \neq e$.

T' is a spanning tree for G.

$w(T') = w(T) + w(e) - w(e') \leq w(T)$ e has min weight.

T' is an MST containing e.

For what other choices of e will it work?

Rule 2: Minimum weight edge leaving any vertex

Rule 3: Minimum weight edge crossing any cut of the graph

Prim's algorithm

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Correctness: The algorithm can be thought of as contracting (u, v) and using rule 2 recursively.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Correctness: The algorithm can be thought of as contracting (u, v) and using rule 2 recursively.

Key question: How to find the min weight edge leaving s.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Correctness: The algorithm can be thought of as contracting (u, v) and using rule 2 recursively.

Key question: How to find the min weight edge leaving s.
Let $d(v) =$ weight of least cost edge from v to s,
$= \infty$ if no such edge.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Correctness: The algorithm can be thought of as contracting (u, v) and using rule 2 recursively.

Key question: How to find the min weight edge leaving s.
Let $d(v) =$ weight of least cost edge from v to s,
$= \infty$ if no such edge.
Maintain a priority queue of vertices with key $= d(v)$.

Prim's algorithm

1. $s =$ any vertex of the graph, $T = \phi$.
2. 2.1 Let e be the least weight edge out of s
   2.2 Add e to T.
   2.3 contract e, name the new vertex s
   2.4 Repeat step if graph has ≥ 1 vetices.
3. Return T.

Correctness: The algorithm can be thought of as contracting (u, v) and using rule 2 recursively.

Key question: How to find the min weight edge leaving s.
Let $d(v) =$ weight of least cost edge from v to s,
$= \infty$ if no such edge.
Maintain a priority queue of vertices with key $= d(v)$.

Representing T: $v =$ root $\Rightarrow T(v) = v$. Else $T(v) =$ parent of v.

Prim's algorithm details

Q = empty min priority queue.
s = any vertex in G. $d(s) = 0$. $T(s) = s$
For all vertices $v \neq s$, $d(v) = \infty$.
Insert each v into Q with key $d(v)$.
while Q is not empty
    u = delete-min from Q.
    for each neighbour v of u:
    if $d(v) > w(u, v)$,
        decrease-key of v to $w(u, v)$ in Q.
        $d(v) = w(u, v)$. $T(v) = u$
    end if
end while

Prim's algorithm details

$Q =$ empty min priority queue.
$s =$ any vertex in G. $d(s) = 0$. $T(s) = s$
For all vertices $v \neq s$, $d(v) = \infty$.
Insert each v into Q with key $d(v)$.
while Q is not empty
    $u =$ delete-min from Q.
    for each neighbour v of u:
    if $d(v) > w(u, v)$,
        decrease-key of v to $w(u, v)$ in Q.
        $d(v) = w(u, v)$. $T(v) = u$
    end if
end while

m decrease key operations, $n - 1$ delete-min. Time $O(m \log n)$.

Remarks

"Fibonacci heap" can be used to reduce the time to $O(m + n \log n)$ instead of the present $O(m \log n)$.

Clustering to maximize spacing

Clustering to maximize spacing

Input: Graph with non-negative edge weights w. Positive integer k.

Clustering to maximize spacing

Input: Graph with non-negative edge weights w. Positive integer k.

Goal: Partition vertices into clusters (non-empty vertex sets) so that "distance between clusters" is maximum.

Clustering to maximize spacing

Input: Graph with non-negative edge weights w. Positive integer k.

Goal: Partition vertices into clusters (non-empty vertex sets) so that "distance between clusters" is maximum.

Notation: Distance between vertex sets V', V'' is defined to be

$$d(V', V'') = \min_{u \in V', v \in V''} w(u, v)$$

Clustering to maximize spacing

Input: Graph with non-negative edge weights w. Positive integer k.

Goal: Partition vertices into clusters (non-empty vertex sets) so that "distance between clusters" is maximum.

Notation: Distance between vertex sets V', V'' is defined to be

$$d(V', V'') = \min_{u \in V', v \in V''} w(u, v)$$

Output: Partition of vertex set into $V_1, \ldots, V_k$ so as to maximize "spacing" defined as:

$$\text{spacing} = \min_{i \neq j} d(V_i, V_j)$$

Greedy algorithm?

Greedy algorithm?

Initially every vertex is a cluster.

Greedy algorithm?

Initially every vertex is a cluster.
Find the minimum inter cluster edge and merge the clusters it connects.

Greedy algorithm?

Initially every vertex is a cluster.
Find the minimum inter cluster edge and merge the clusters it connects.
Repeat until k clusters remain.

Greedy algorithm?

Initially every vertex is a cluster.
Find the minimum inter cluster edge and merge the clusters it connects.
Repeat until k clusters remain.

Sounds familiar?

Greedy algorithm?

Initially every vertex is a cluster.
Find the minimum inter cluster edge and merge the clusters it connects.
Repeat until k clusters remain.

Sounds familiar?

This is Kruskal's algorithm, except that we stop when k supervertices remain.

Greedy algorithm?

Initially every vertex is a cluster.
Find the minimum inter cluster edge and merge the clusters it connects.
Repeat until k clusters remain.

Sounds familiar?

This is Kruskal's algorithm, except that we stop when k supervertices remain.

Alternative: Find MST somehow and delete $k-1$ max weight edges.

Proof of correctness!

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.

The vertices contained in a supervertex constitute a cluster.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster. $\Rightarrow$ Greedy $=$ OPT.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster. $\Rightarrow$ Greedy $=$ OPT.

Case 2: Suppose some greedy cluster V_r contains vertices u, v of distinct clusters C_p, C_q produced by the optimal algorithm.

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster. $\Rightarrow$ Greedy $=$ OPT.

Case 2: Suppose some greedy cluster V_r contains vertices u, v of distinct clusters C_p, C_q produced by the optimal algorithm.
wlog u, v are adjacent in the tree of V_r

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster. $\Rightarrow$ Greedy $=$ OPT.

Case 2: Suppose some greedy cluster V_r contains vertices u, v of distinct clusters C_p, C_q produced by the optimal algorithm.
wlog u, v are adjacent in the tree of V_r

Spacing(OPT) $\leq w(u, v)$

Proof of correctness!

We stop Kruskal's algorithm when k supervertices remain.
The vertices contained in a supervertex constitute a cluster.
The edges contracted to form a supervertex form a tree.

Weight of edges connecting distinct supervertices $\geq$ weight of edges contained in the trees inside each supervertex.
$\Leftarrow$ Kruskal's algorithm contracts edges in non-decreasing weight order.
Spacing(greedy) $=$ weight of some inter supervertex edge
$\geq$ weight of any tree edge.

Case 1: Every greedy cluster contains vertices of a single OPT cluster. $\Rightarrow$ Greedy $=$ OPT.

Case 2: Suppose some greedy cluster V_r contains vertices u, v of distinct clusters C_p, C_q produced by the optimal algorithm.
wlog u, v are adjacent in the tree of V_r

Spacing(OPT) $\leq w(u, v) \leq$ Spacing(Greedy)