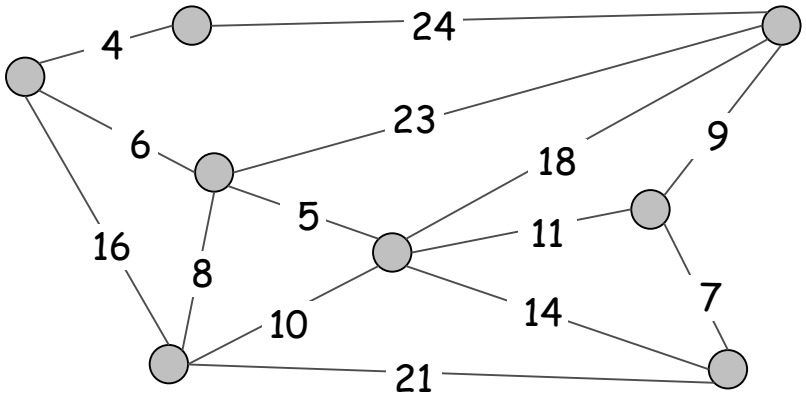
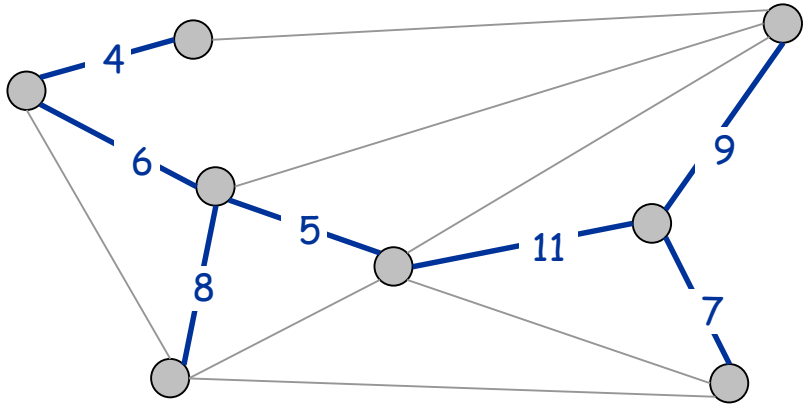
Lecture 16: Minimum Spanning Trees

Minimum Spanning Tree

Minimum spanning tree. Given a connected undirected graph $G = (V, E)$ with real-valued edge weights $w(e)$, an MST is a subset of the edges $T \subseteq E$ such that T is a tree that connects all nodes whose sum of edge weights is minimized.



$G = (V, E)$



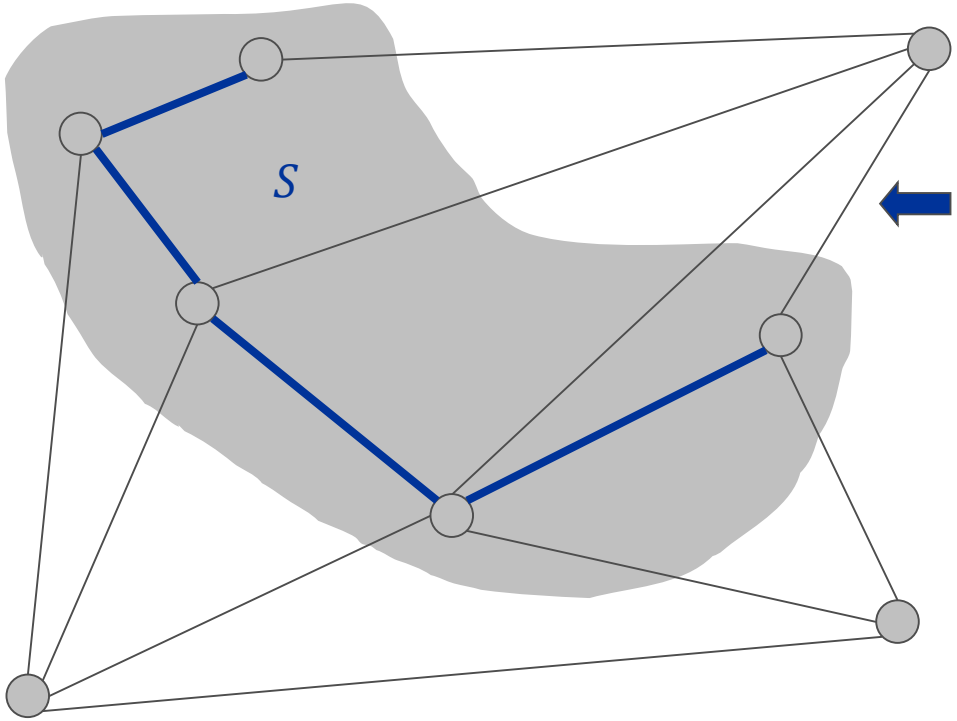
$T, \Sigma_{e \in T}\, w(e) = 50$

Applications: telephone, electrical, hydraulic, TV cable, computer, road

Prim's Algorithm: Idea

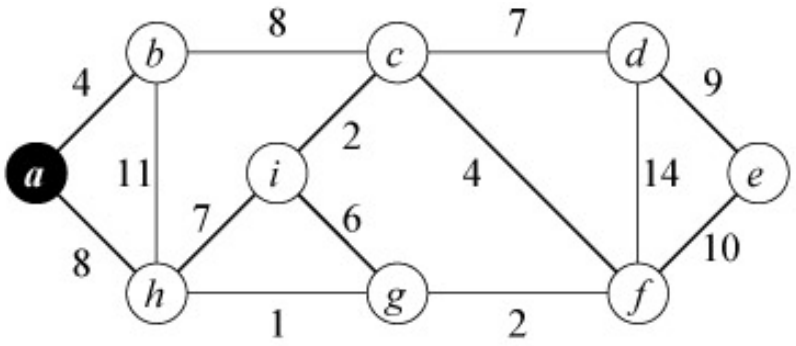
Prim's algorithm

- Initialize S = {any one node}.
- Add min cost edge $e = (u, v)$ with $u \in S$ and $v \in V - S$ to T.
- Add v to S.
- Repeat until $S = V$

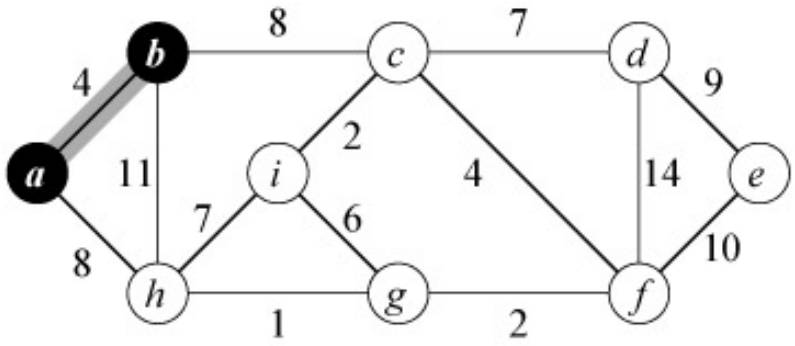


Prim's Algorithm: Example

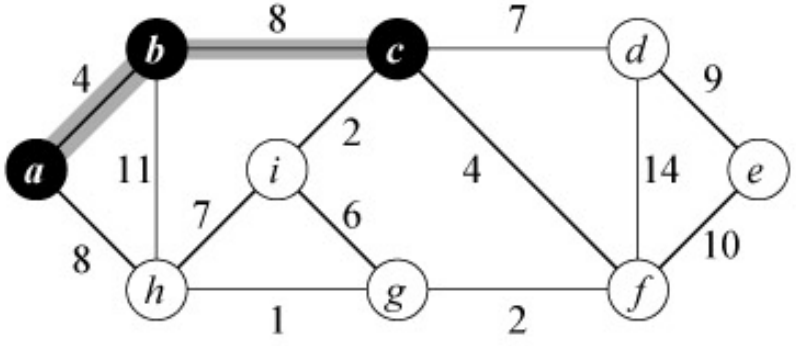
(a)



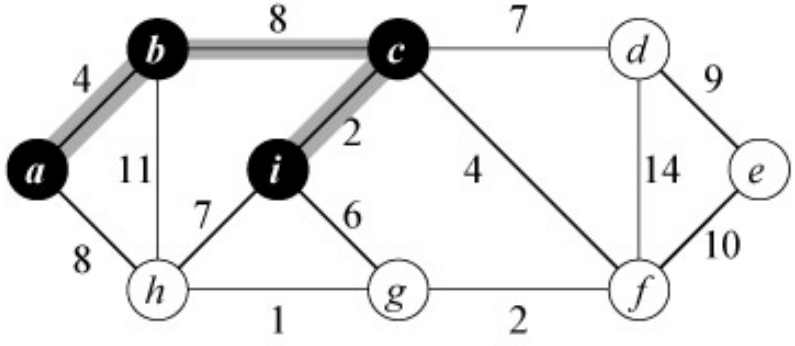
(b)



(c)

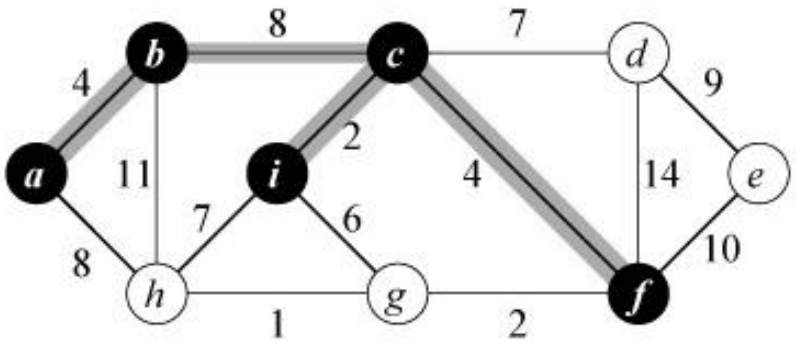


(d)

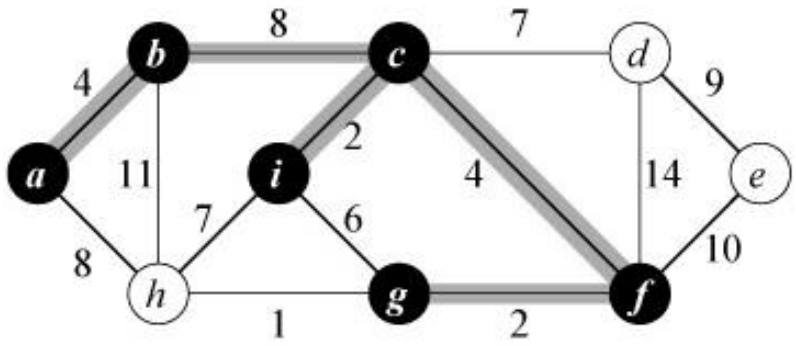


Prim's Algorithm: Example (continued)

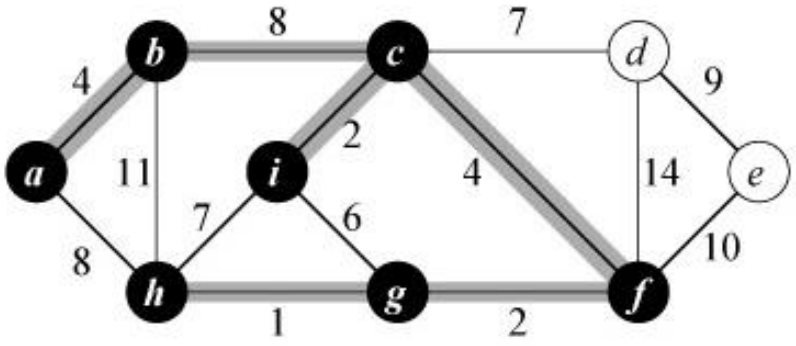
(e)



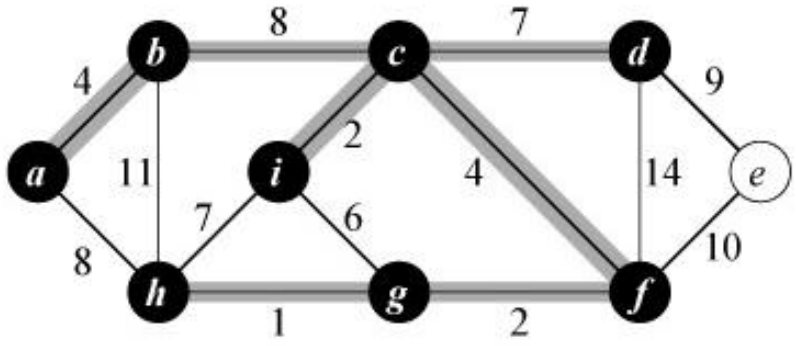
(f)



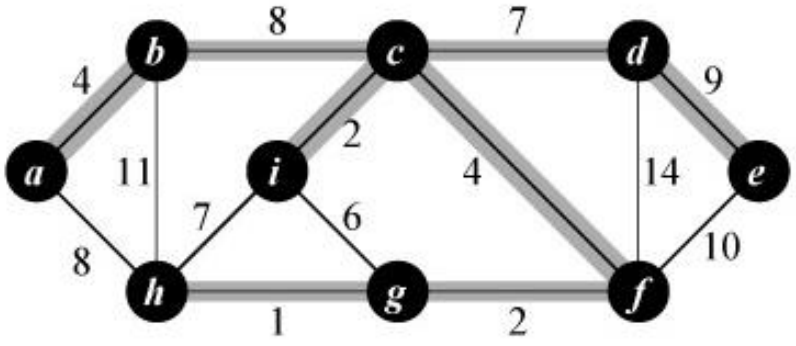
(g)



(h)



(i)



Prim's Algorithm: Implementation

Implementation.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain the cheapest edge from v to a node in S.
- Maintain all nodes in a priority queue with this cheapest edge as key

```
Prim(G, r):
for each v ∈ V do
    v.key ← ∞, v.p ← nil, v.color ← white
r.key ← 0
create a min priority queue Q on V
while Q ≠ ∅
    u ← Extract-Min(Q)
    u.color ← black
    for each v ∈ Adj[u] do
        if v.color = white and w(u,v) < v.key then
            v.p ← u
            v.key ← w(u,v)
            Decrease-Key(Q, v, w(u,v))
```

Note: In the end, the parent pointers form the MST.

Running time: $O(E \log V)$

Q: Decrease-key needs the location of the key in the heap. How to get that?

Cut Lemma

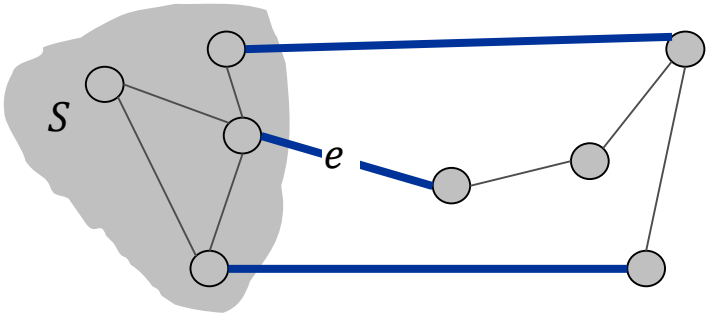
Simplifying assumption. All edge weights are distinct.

Cut lemma. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then any MST must contains e.

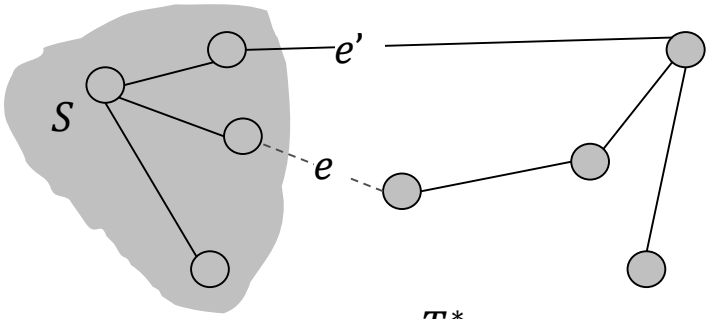
Correctness of Prim's Algorithm: Apply the lemma between the black and white vertices.

Pf. (exchange argument)

- Let T^* be any MST.
- Let $e = (u, v)$ and suppose e does not belong to T^*.
- There is a path in T^* that goes from u to v, which must cross the cut using some other edge e' with $w(e') > w(e)$.
- If we replace e' with e in T^*, then T^* is still a spanning tree, but the total cost will be lower, which contradicts with the fact that T^* is an MST.



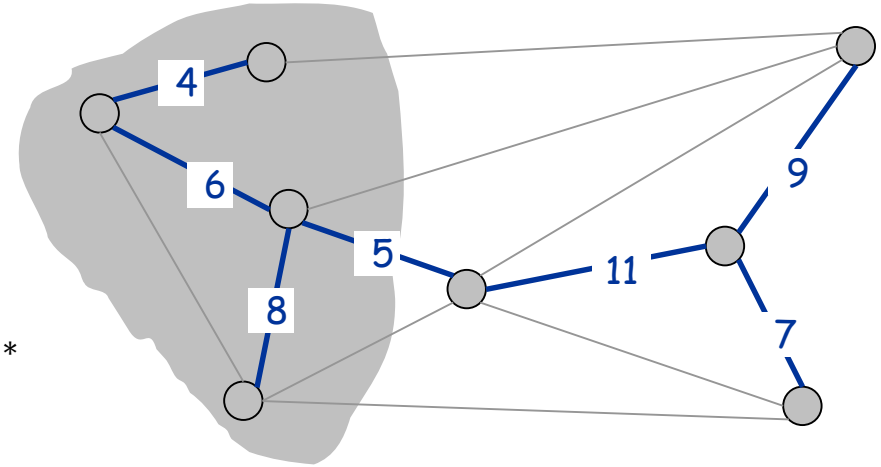
e is in the MST



T^*

Uniqueness of MST

Theorem: The MST is unique.



Pf:

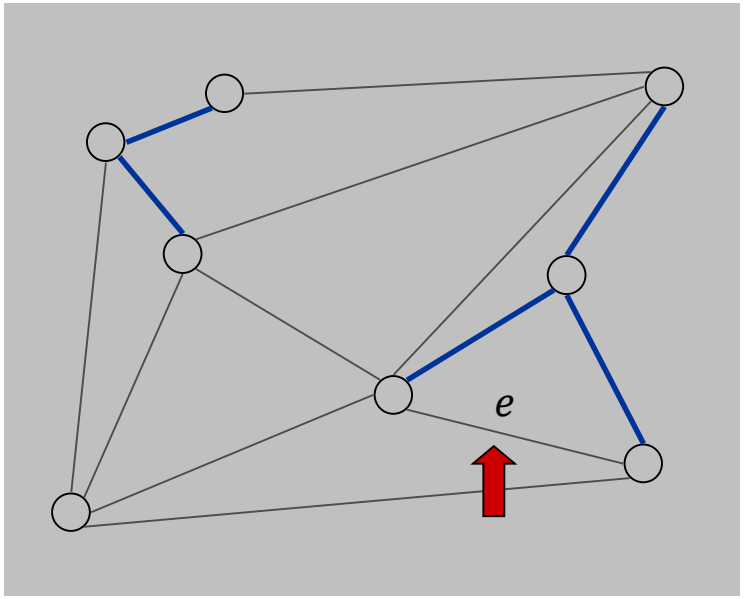
- Let T^* be an MST.
- Consider any edge $e \in T^*$
- Removing e from T^* breaks T^* into two parts S and $V - S$
- e must be the min cost edge crossing the cut $(S, V - S)$. (If not, we can replace e with the min cost edge and improve the MST.)
- Applying the cut lemma on S, we know that any MST must contain e.
- Applying the above argument to every edge of T^*, we have
  - There are $V - 1$ edges in the graph such that any MST must contain all of them.
  - Any spanning tree must have exactly $V - 1$ edges.
  - So, any MST must have those $V - 1$ edges, i.e., the same as T^*.

Note: If there are edges with equal weights, then the MST may not be unique.

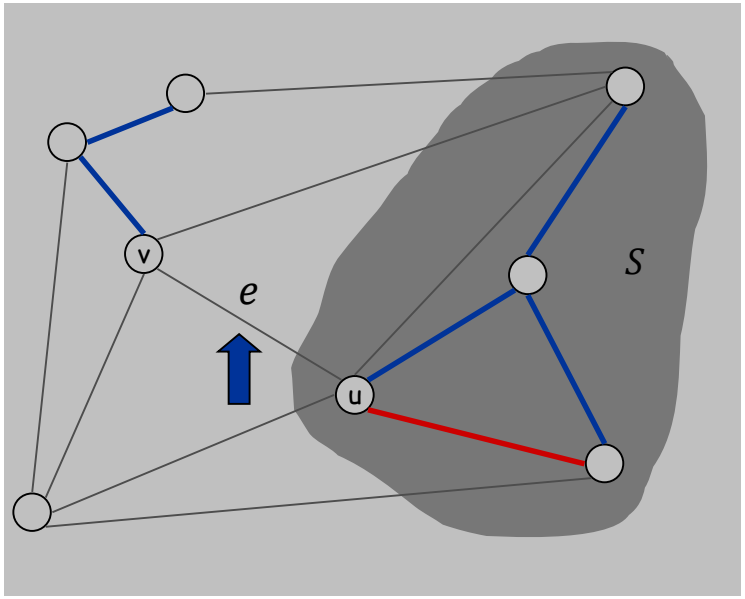
Kruskal's Algorithm: Idea

Kruskal's algorithm.

- Starts with an empty tree T
- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e.
- Case 2: Otherwise, insert $e = (u, v)$ into T according to cut lemma



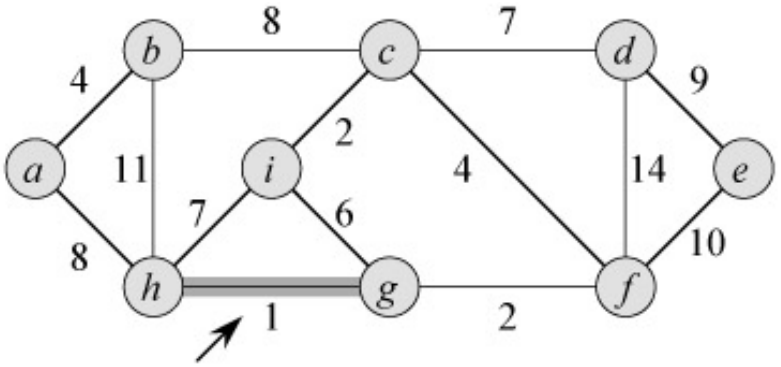
Case 1



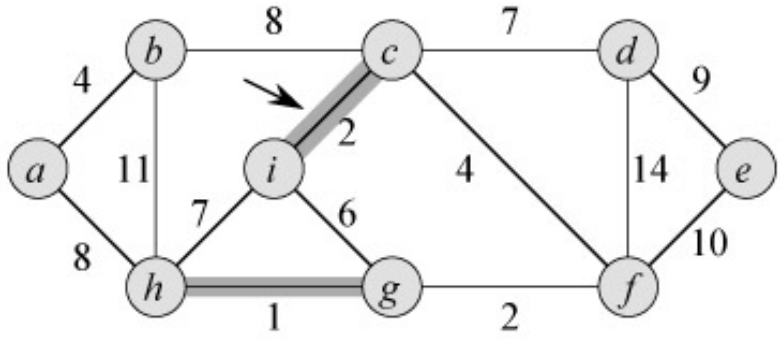
Case 2

Kruskal's Algorithm: Example

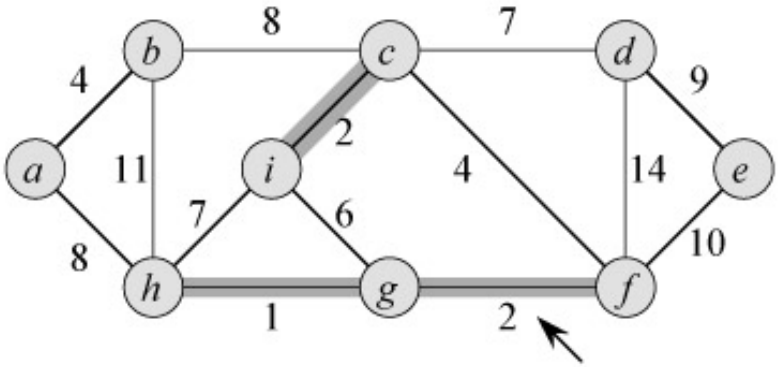
(a)



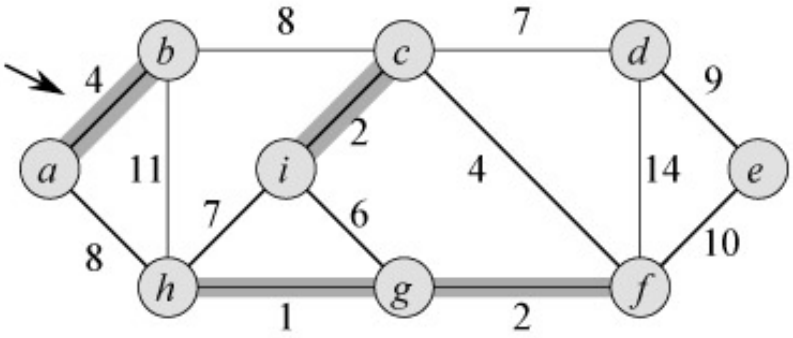
(b)



(c)

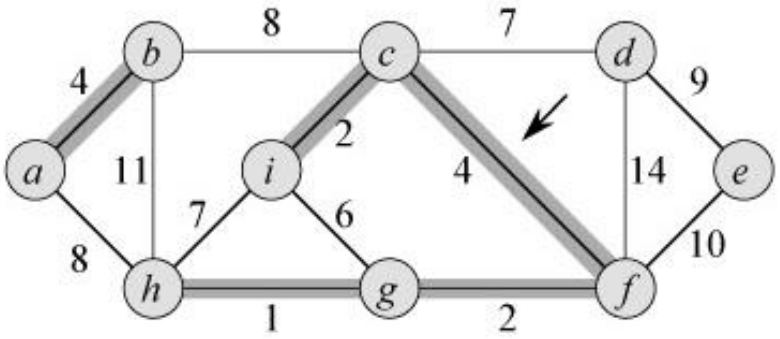


(d)

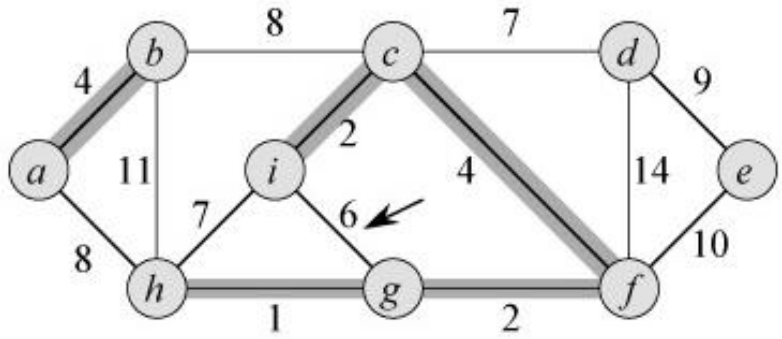


Kruskal's Algorithm: Example (continued)

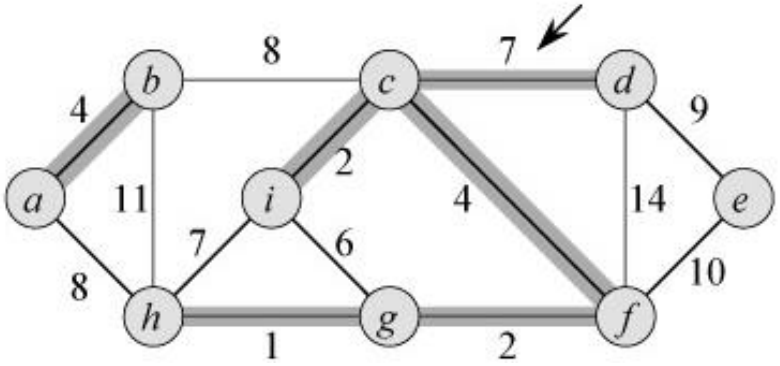
(e)



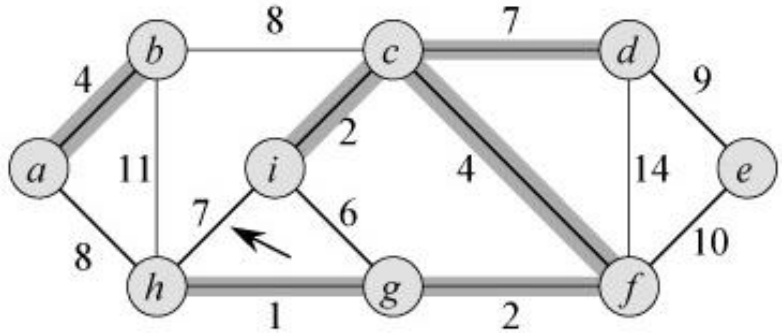
(f)



(g)

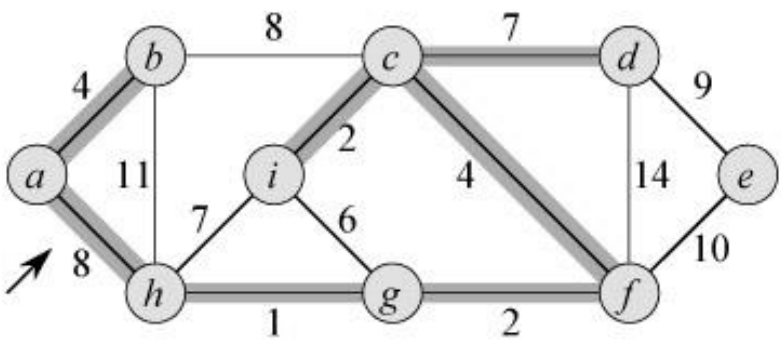


(h)

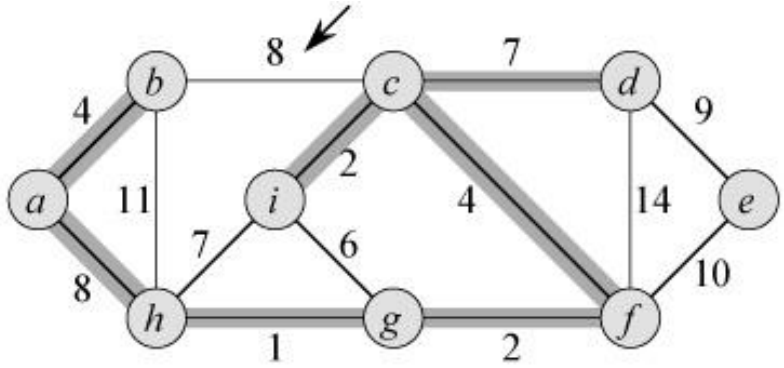


Kruskal's Algorithm: Example (continued)

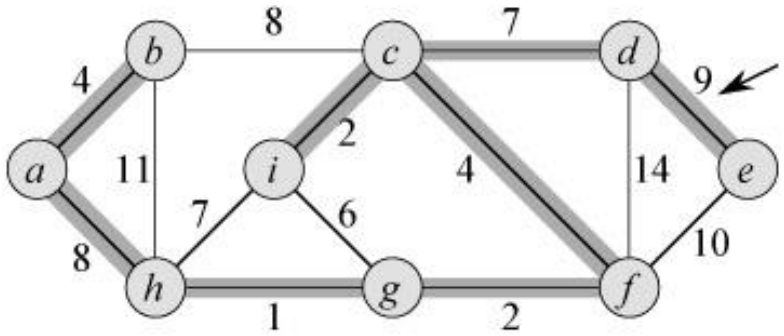
(i)



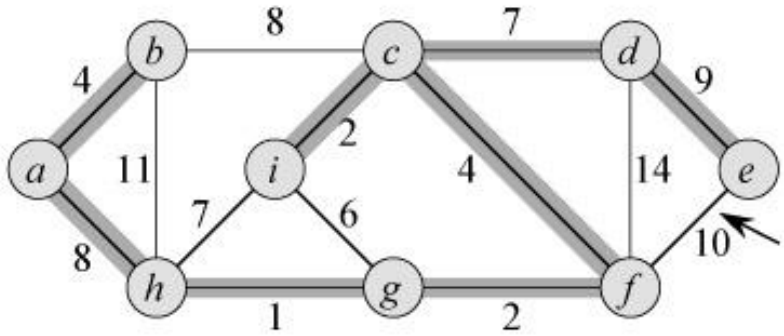
(j)



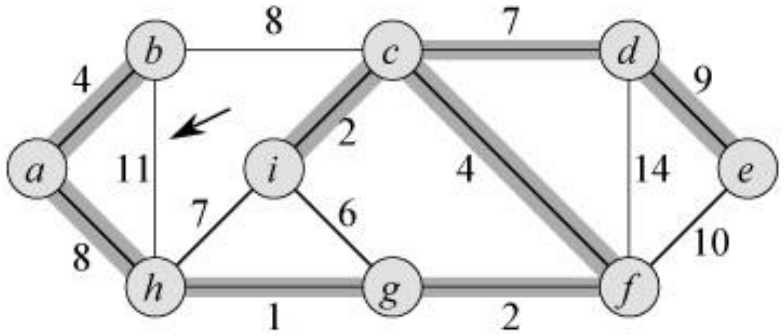
(k)



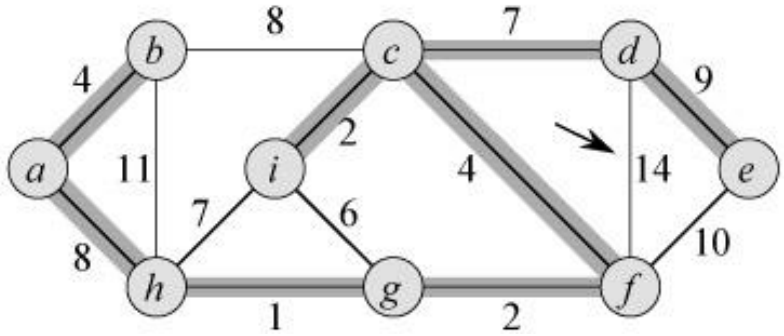
(l)



(m)



(n)



Kruskal's Algorithm: Implementation

Key question: How to check whether adding e to T will create a cycle?

- Use DFS?
  - Would result in $O(E \cdot V)$ total time.
- Can we do the checking in $O(\log V)$ time?

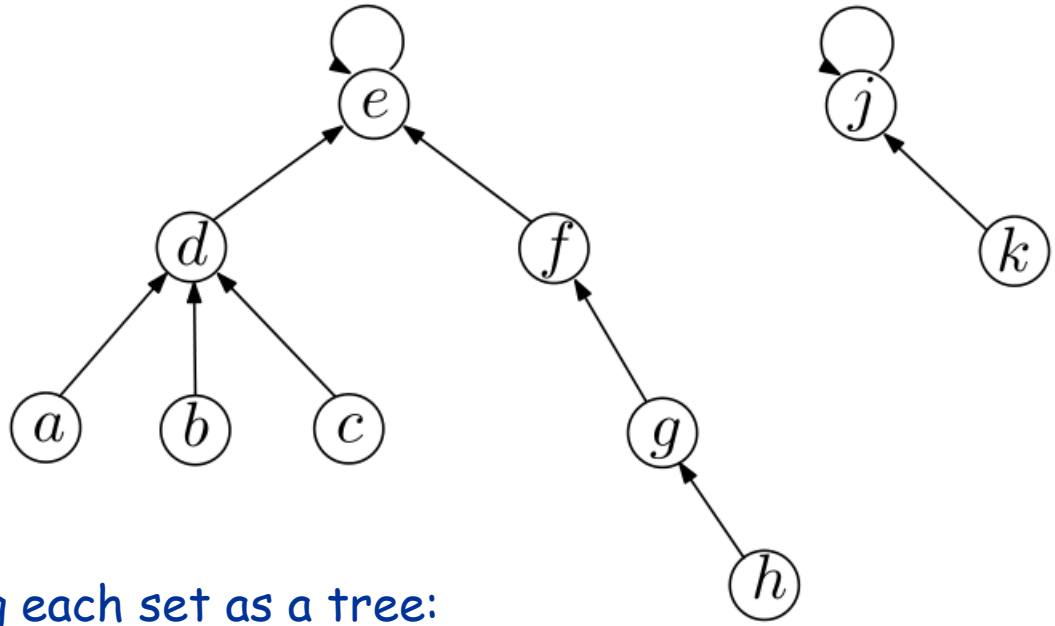
Observations:

- The actual structure of each component of T does not matter.
  - Each component can be considered as a set of nodes.
- After an edge is added, two sets "union" together.

Need such a "union-find" data structure:

- Maintain a collection of sets to support the following two operations:
- **`Find-Set(`**u**`)`**: For a given node u, find which set this node belongs to.
- **`Union`**(u, v): For two given nodes u and v, merge the two sets containing u and v together.

The union-find data structure



Representing each set as a tree:

- The tree in the union-find data structure may not be the same as that in the partial MST!
- The root of the tree is the representative node of all nodes in that tree (i.e., use the root's ID as the unique ID of the set).
- Every node (except the root), has a pointer pointing to its parent.
  - The root has a parent pointer to itself.
  - No child pointers (unlike BST), so a node can have many children.

Make-Set(x) and Find-Set(x)

Create-Set(x):

```
Make-Set(x):
x.parent ← x
x.height ← 0
```

Find-Set(x):

```
Find-Set(x):
while x != x.parent do
      x ← x.parent
return x
```

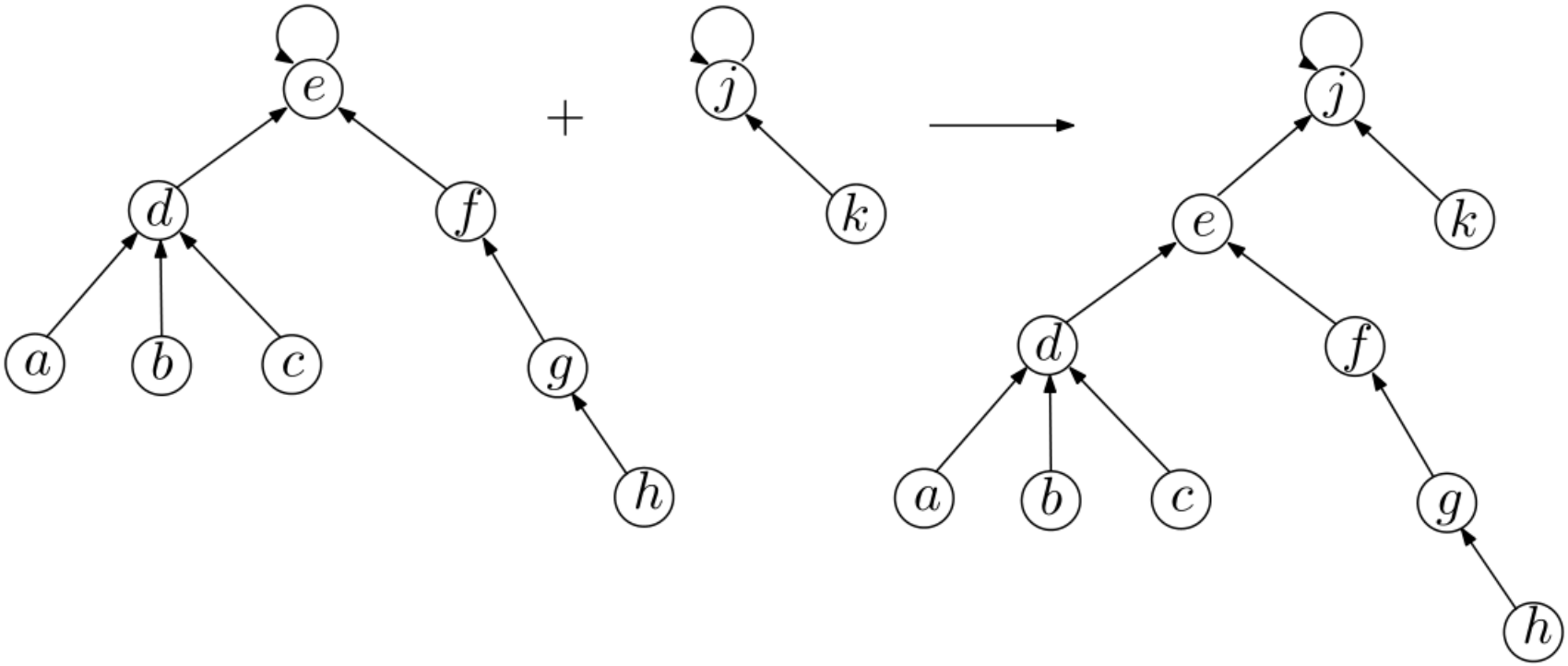
Running time proportional to the height of the tree.

Union(x, y)

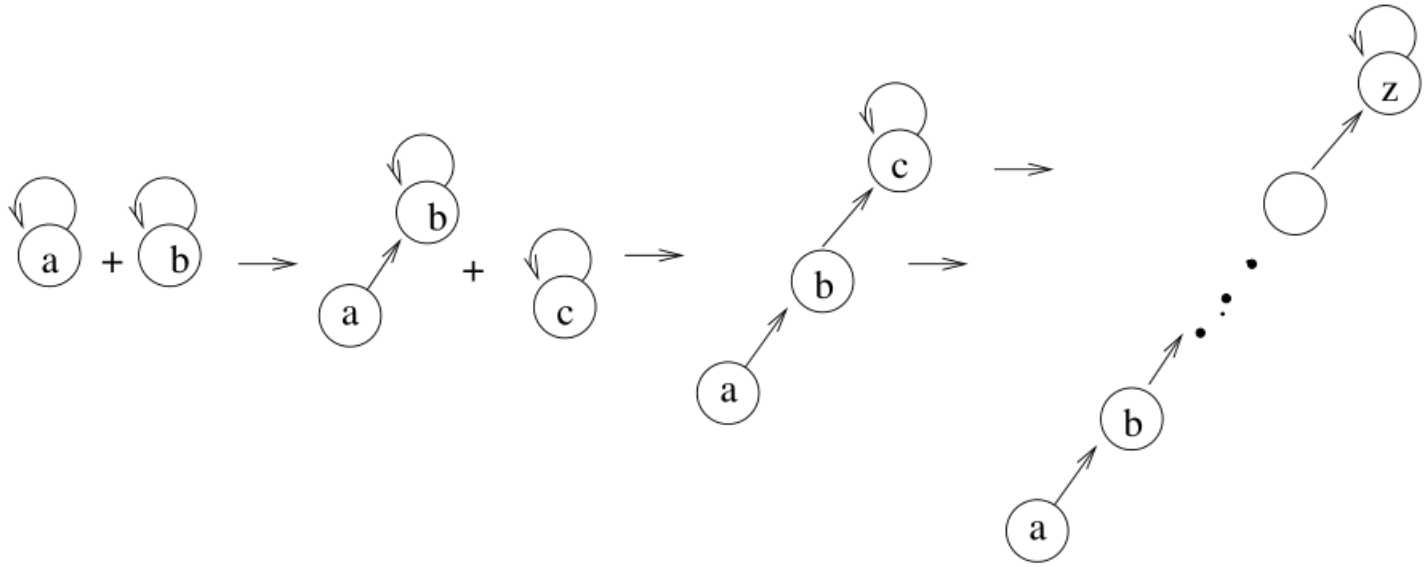
Assumption: x and y are the roots of the their trees.

- If not, do Find-Set first

Idea: Set $x.parent \leftarrow y$



But, what if…



Solution (union by height):

- When we union two trees together, we always make the root of the taller tree the parent of shorter tree.
- Need to maintain the height of each tree

```
Union(x, y):
a ← Find-Set(x)
b ← Find-Set(y)
if a.height ≤ b.height then
      if a.height = b.height then
            b.height ← b.height + 1
      a.parent ← b
else
      b.parent ← a
```

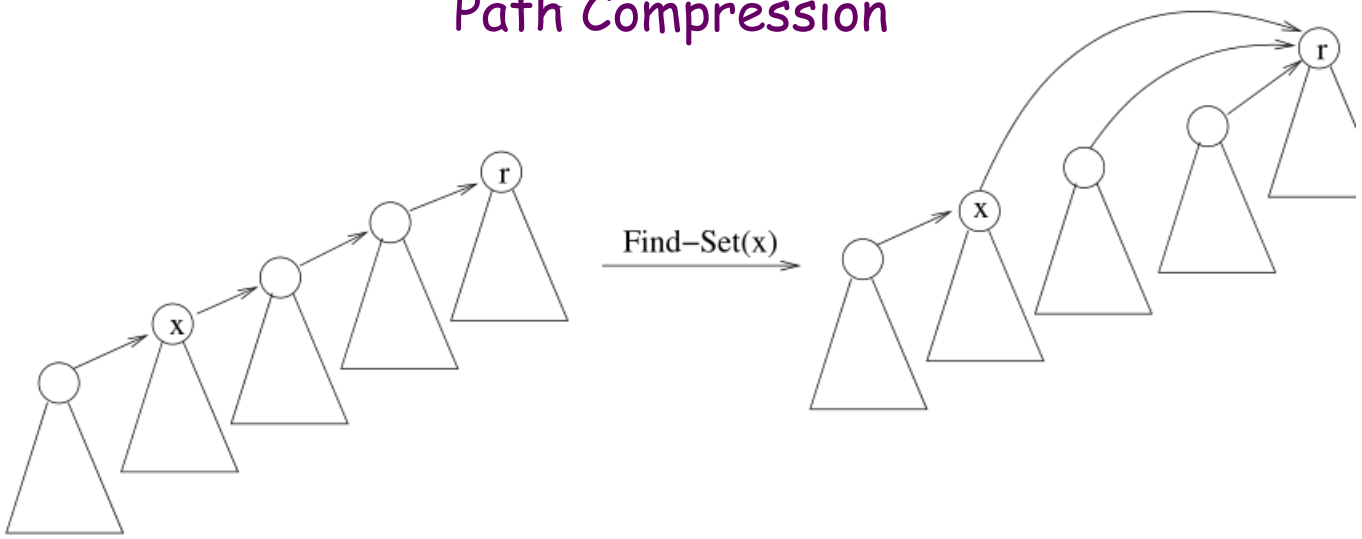
The union-find data structure: Analysis

Theorem: The running time of Find-Set and Union is $O(\log n)$

Pf: We will show (by induction) that for any tree with height h, its size is at least 2^h.

- At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0$.
- Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
  - Case 1: $h(x) < h(y)$, we have
    $size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}$.
  - Case 2: $h(x) = h(y)$, we have
    $size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}$.
  - Case 3: $h(x) > h(y)$, similar to case 1.

Path Compression



Idea:

- We have visited a number of nodes after Find-Set(x), and have reached the root r.
- We already know that these nodes belong to the set represented by r.
- Why not just set the parent pointers of these nodes to r directly?
  - Future operations will be faster!

Analysis:

- This results in a running time that is practically a constant (but theoretically not).
- See textbook for details (not required).

Kruskal's Algorithm

```
MST-Kruskal(G):
for each vertex v ∈ V
    Make-Set(v)
sort the edges of G into increasing order by weight
for each edge (u,v) ∈ E taken in the above order
    if Find-Set(u) ≠ Find-Set(v) then
        output edge (u,v)
        Union(u,v)
```

Running time:

- $O(E \log E + E \log V) = O(E \log V)$

Note: If edges are already sorted and we use path compression, then the running time is close to $O(E)$.

Current best MST algorithm:

- An algorithm by Seth and Ramachandran (2002) has been shown to be optimal, but its running time is still unknown…

Removing the distinct weight assumption

Idea: Use a tie-breaker to make equal weights look different

```
boolean less(i,j)
    if w(e_i) < w(e_j) then return true
    else if w(e_i) > w(e_j) then return false
    else if i < j then return true
```

Why does this work?

- Imagine that the weight of e_i is $w(e_i) + i \cdot \delta$, where δ is a sufficiently small number
- Running the algorithm with the above tie-breaker is the same as running the original algorithm on the modified weights
- The MST on the modified weights must also be an MST on the original weights, for δ small enough

Note: In fact, even if we don't use a tie breaker, both Prim's and Kruskal's algorithm are still correct. But the proof of correctness is more complicated (see textbook for details).