COMP 550
Algorithm and Analysis

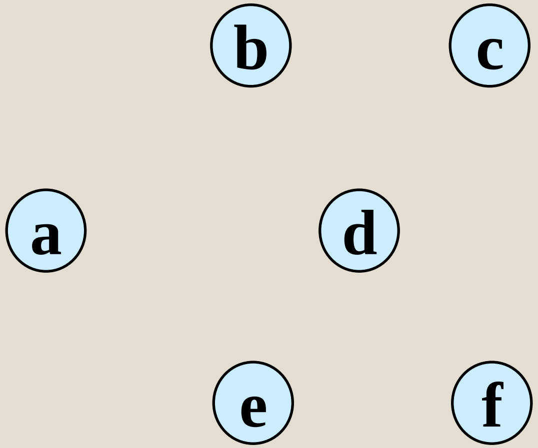
Minimum Spanning Tree

Based on CLRS Sec. 21

Some slides are adapted from ones by Prof. Jim Anderson

Minimum-Cost Communication Network

- We want to set up a communication network on n locations so that each pair of locations are connected
- The cost of a direct link is proportional to their distance
- Construct the network as cheaply as possible



Minimum-Cost Communication Network

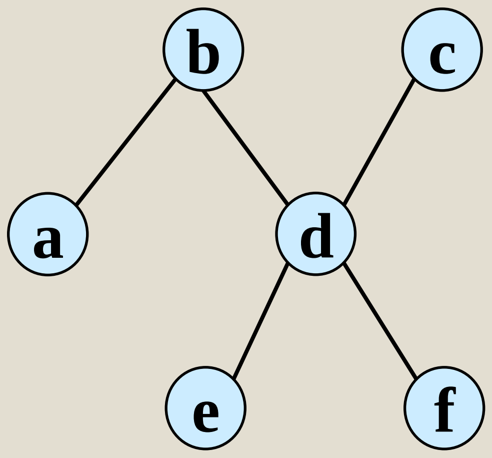
- The number of direct link should be as small as possible
- <u>Property 1</u>: *Acyclic*. (The cheapest network is a tree)

A tree has no cycle (unlike this)

Source: https://www.dailymail.co.uk/news/article-2255706/Amazing-images-bikes-left-long-trees-them.html

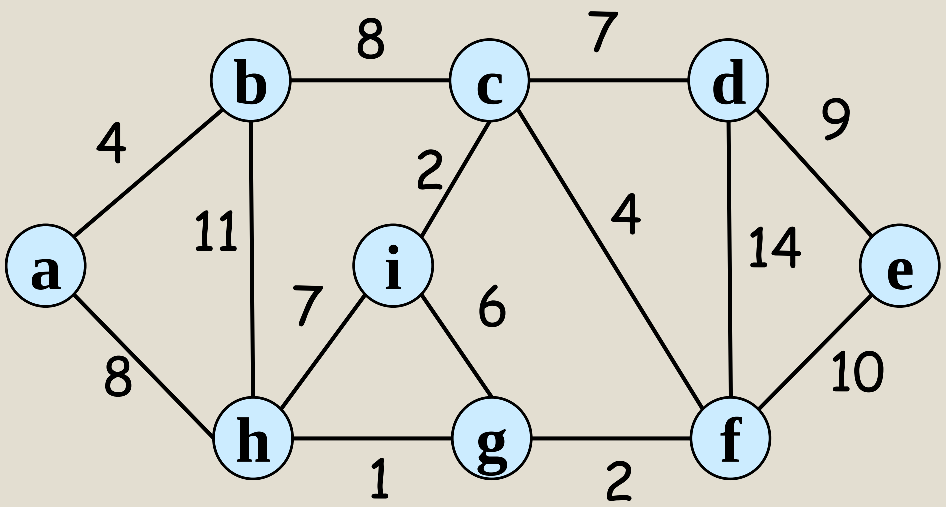
Minimum-Cost Communication Network

- The number of direct link should be as small as possible
- Property 2: Barely connected (formally, *minimally connected*). Removal of any link disconnects some location from the network
- Property 2: *Maximally acyclic*. Addition of any link creates a cycle

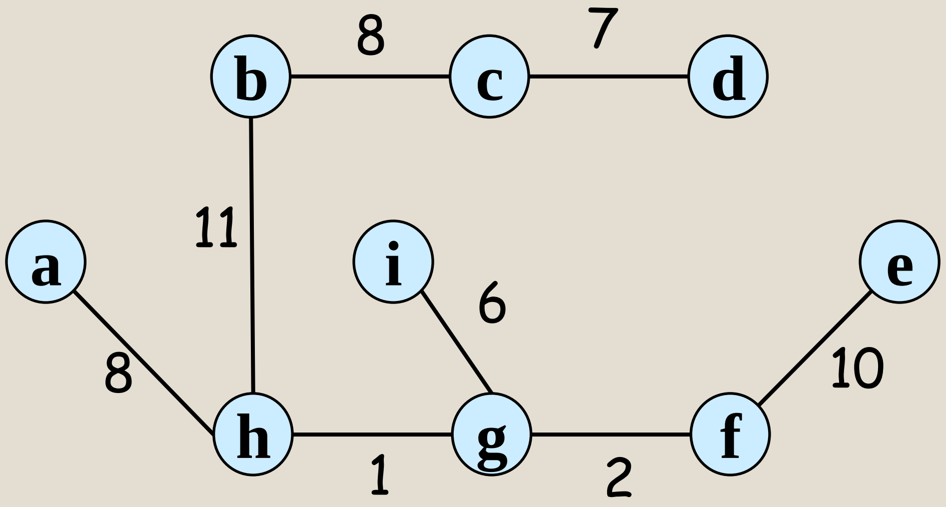


Spanning Tree

- $T = (V_T, E_T)$ is a spanning tree of a connected graph $G = (V, E)$ if T connects ("spans") all vertices in G
  - G is an undirected graph



A graph G

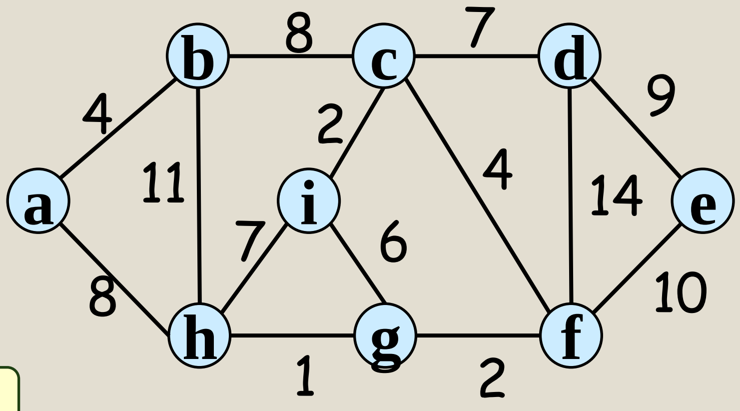


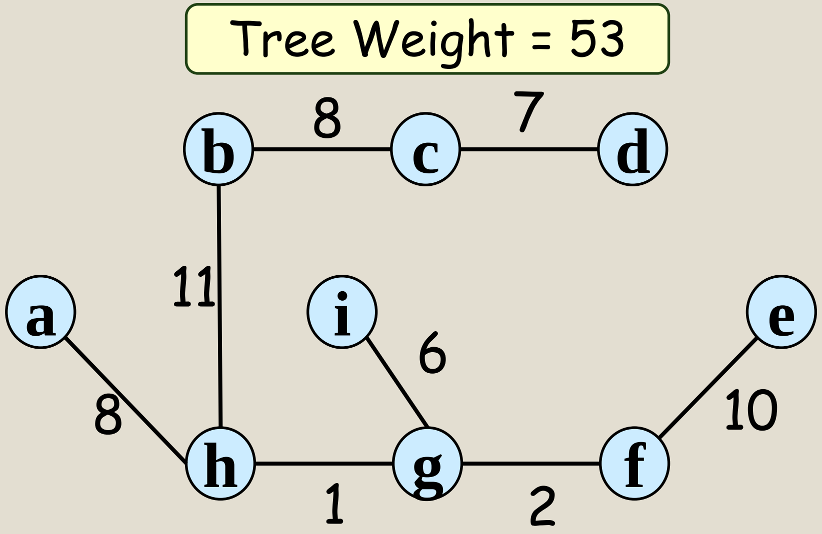
A spanning tree of G

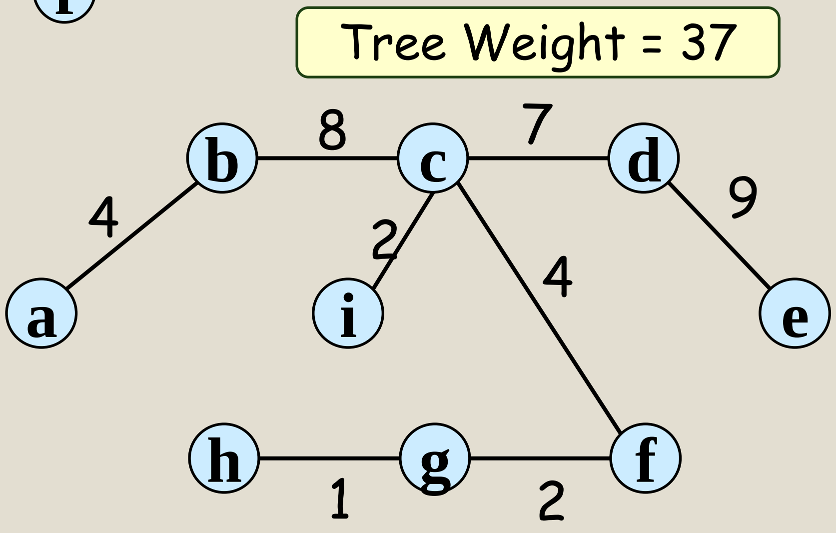
Weight/Cost of a spanning tree $= \sum$ weight of each tree edge

Minimum Spanning Tree (MST)

- A graph G can have many spanning trees
- *Minimum Spanning Tree (MST)*: spanning tree with minimum weight/cost







The MST Problem

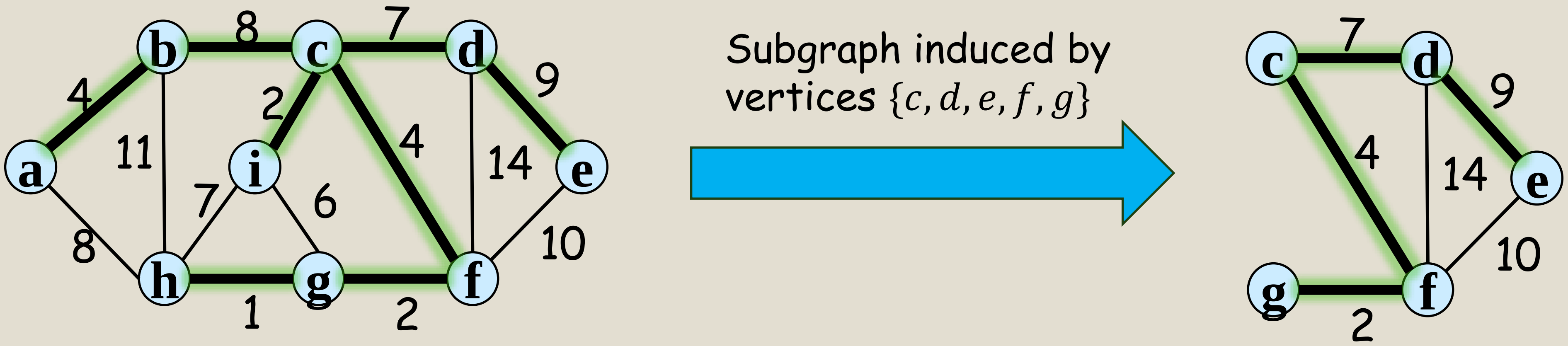
- Input: An undirected weighted graph $G = (V, E)$ with weight function $w: E \to \mathbf{R}$
- Output: An MST of G

- Question: What about MST of an undirected unweighted graph?

Optimal Substructure

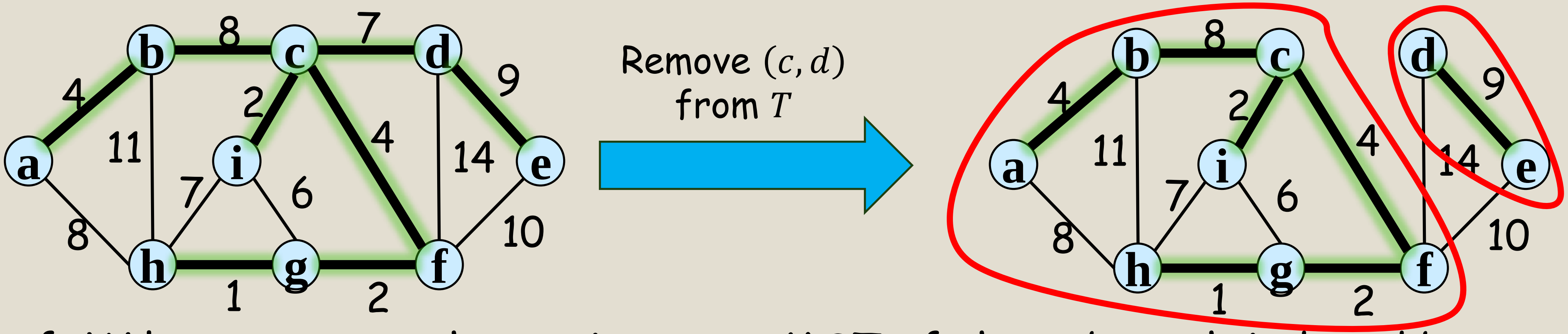
- Does MST have optimal substructure property?
  - Does MST of a graph G contain MST of G's subgraph?
- Yes!

Lemma. Let T_1 and T_2 be two trees after removing an edge (u, v) from an MST T of graph G. Then, T_1 and T_2 are MSTs of the *subgraphs induced* by nodes of T_1 and T_2, respectively.



Optimal Substructure

Lemma. Let T_1 and T_2 be two trees after removing an edge (u, v) from an MST T of graph G. Then, T_1 and T_2 are MSTs of the *subgraphs induced* by nodes of T_1 and T_2, respectively.

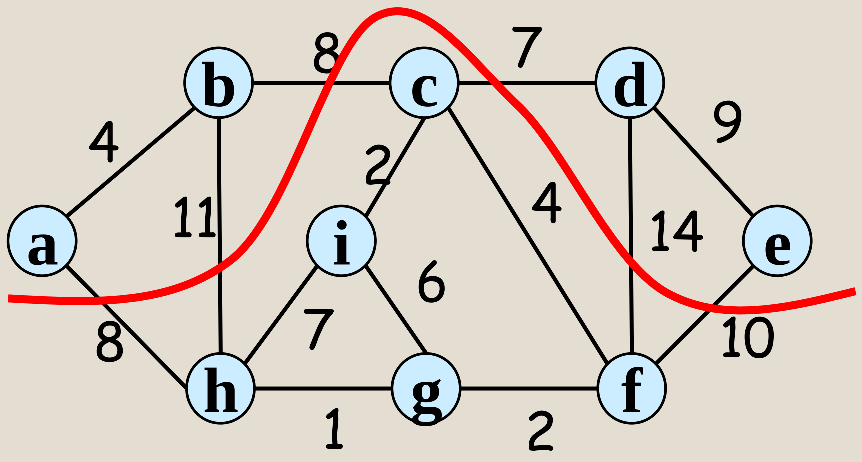


Proof: W.l.o.g., assume that T_1 is not an MST of the subgraph induced by T_1.

Assume T_1' is an MST of that subgraph.

Then, we can create an MST T' of G by connecting T_1' and T_2 by (u, v). T' has smaller weight than T, contradiction.

Cut



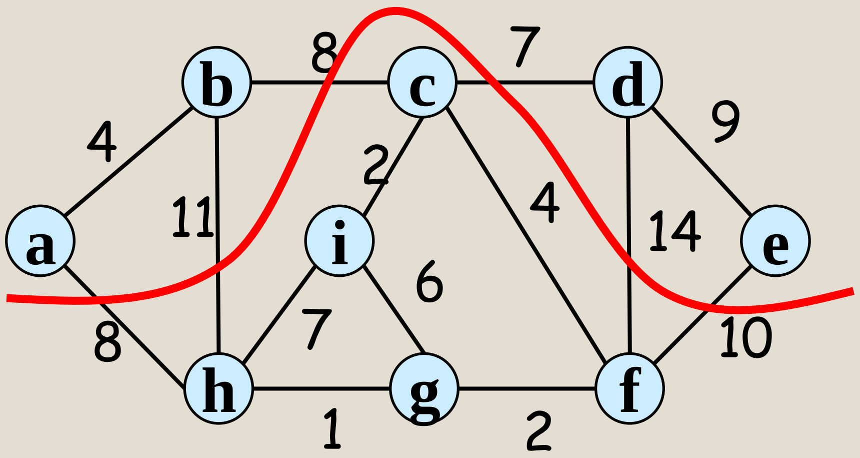
- A cut $(S, V - S)$ of an undirected graph is a partition of V
- An edge *crosses* cut $(S, V - S)$ if its one endpoint is in S and the other in $V - S$
- An edge is a *light edge* crossing a cut is its weight is the minimum crossing that cut

<u>Example</u>: $S = \{a, b, d, e\}$ and $V - S = \{c, f, g, h, i\}$

- Edges $(a, h), (b, h), (b, c), (c, d), (d, f)$ *cross* the cut.
- (c, d) is the *light edge* for this cut

Cut Property

Lemma. Let $(S, V - S)$ be a cut of a graph G and (u, v) be a light edge crossing the cut. Then, there is an MST that has (u, v) as one of its edge.



There is an MST that has the edge (c, d) as one of its edge

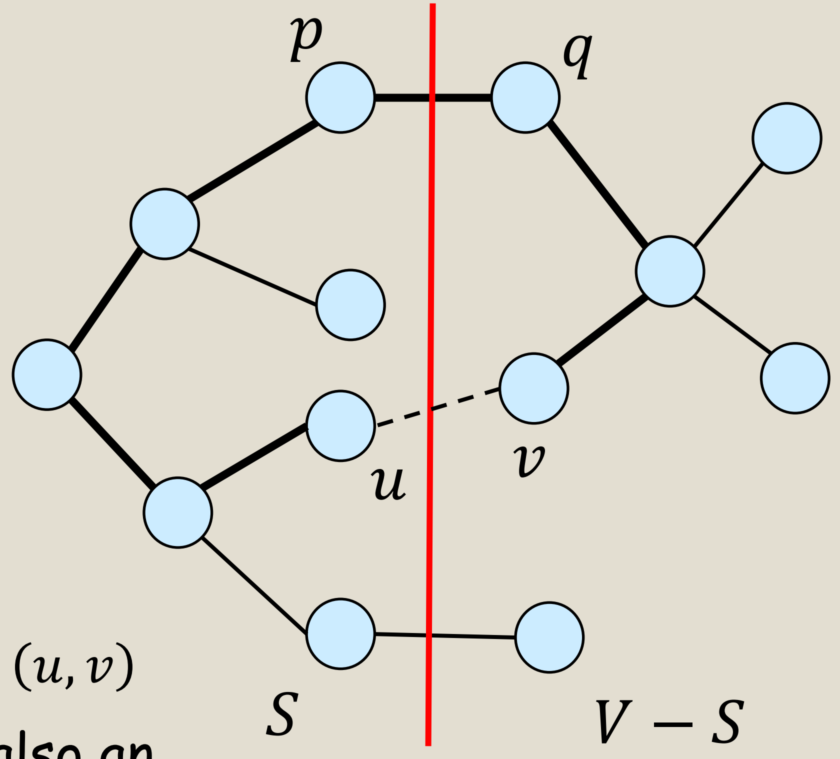
This property is basically a greedy-choice property

Cut Property

Lemma. Let $(S, V-S)$ be a cut of a graph G and (u, v) be a light edge crossing the cut. Then, there is an MST that has (u, v) as one of its edge.

Proof: Let T' be an MST the does not have (u, v) as one of its edge.

- W.l.o.g., assume that $u \in S$ and $v \in V-S$
- There is a path P from u to v in T'
- Let q be the first node in $V-S$ in the P
- Let p be the node just before q in P (so, $p \in S$)
- Thus, edge (p, q) crosses the cut $(S, V-S)$
- Since (u, v) is a light edge of this cut, $w(p, q) \geq w(u, v)$
- We can create a spanning tree by exchanging (p, q) with (u, v)
- The new tree's cost is no more than T'. The new tree is also an MST



MST Algorithms

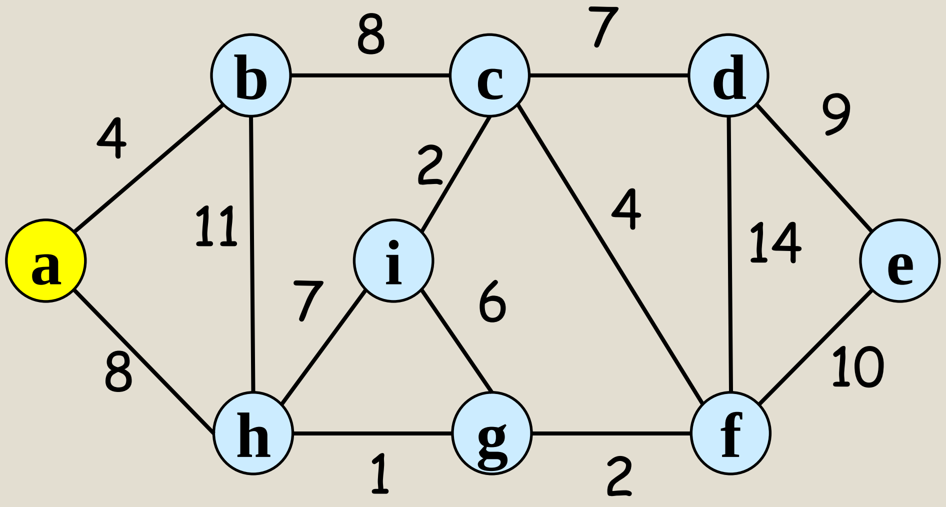
We will see two algorithms based on the cut property

1. Prim's algorithm
2. Kruskal's algorithm

Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding a node by the *cheapest possible edge*

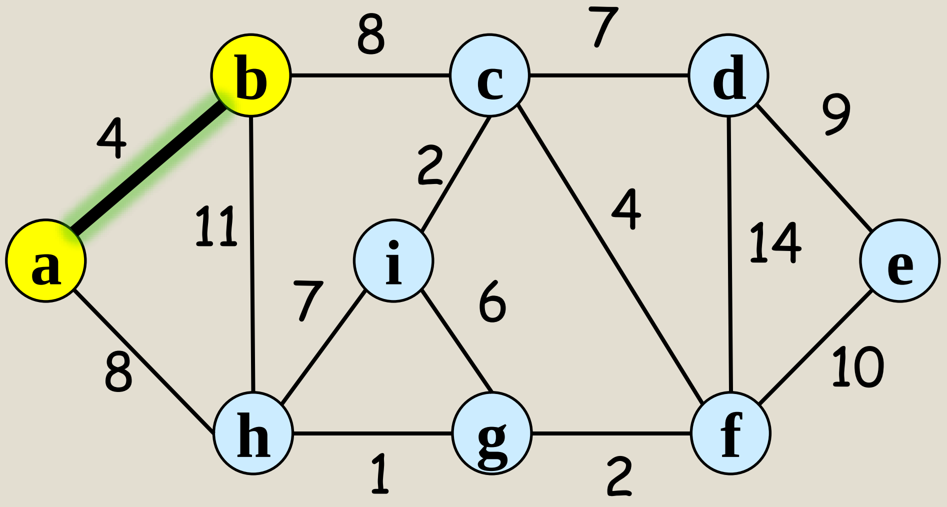
Example:



Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding the *cheapest possible edge* that does not form a cycle

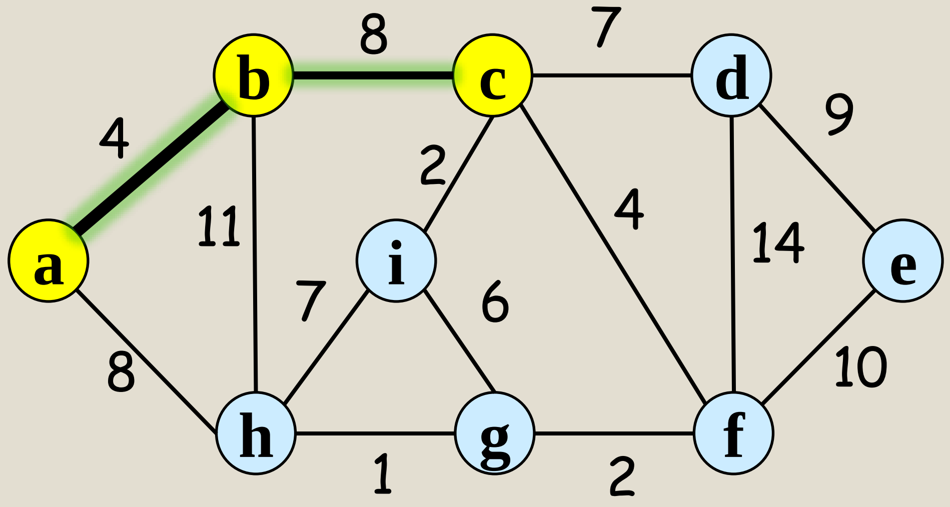
Example:



Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding the *cheapest possible edge* that does not form a cycle

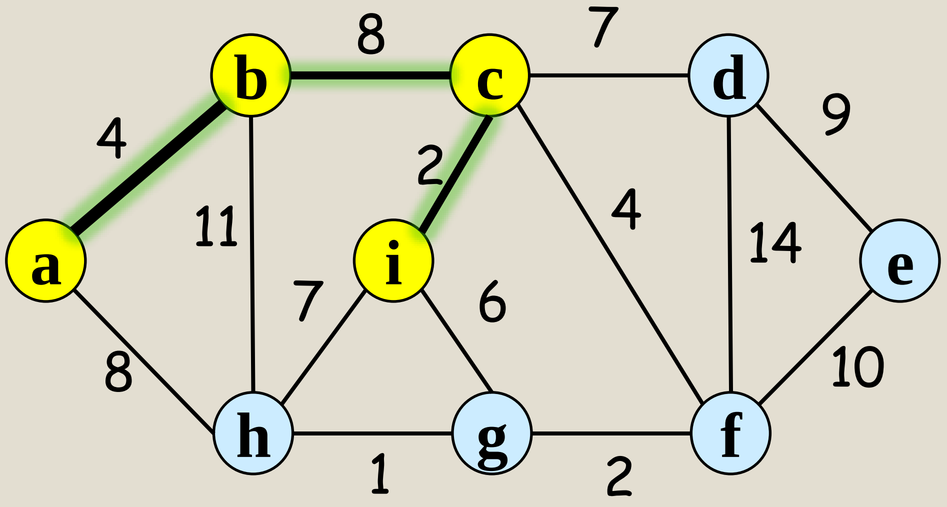
Example:



Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding the *cheapest possible edge* that does not form a cycle

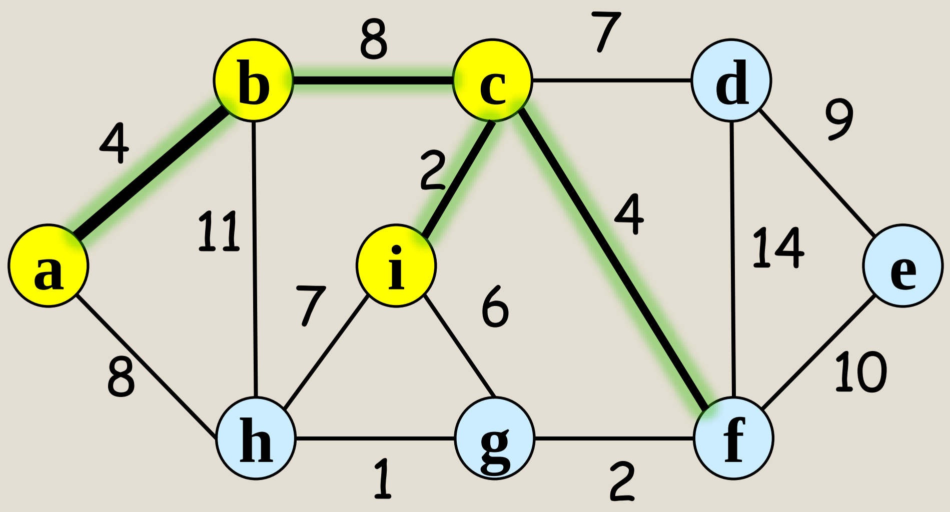
Example:



Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding the *cheapest possible edge* that does not form a cycle

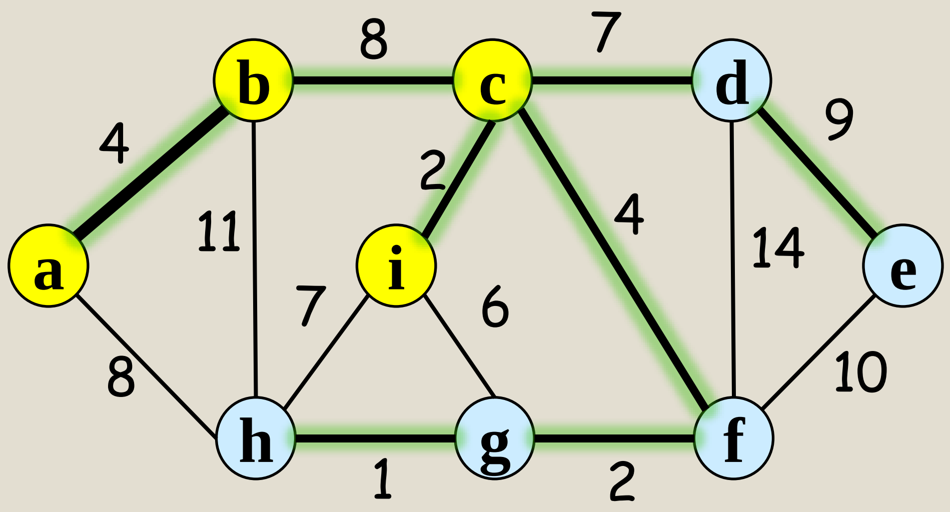
Example:



Prim's Algorithm

- Start with a node s and greedily grow a tree outward
  - Always maintain a partially-constructed tree
- At each step, extend the partial tree by adding the *cheapest possible edge* that does not form a cycle

Example:



Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = Ø
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ Ø
    u = EXTRACT-MIN(Q)    // add u to the tree
    for each vertex v in G.Adj[u]    // update keys of u's non-tre neighbors
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            DECREASE-KEY(Q, v, w(u, v))
```

Q: Min Priority Queue, orders elements based on $v.key$

Greedily take a node that can be added using cheapest-possible edge from the partial tree

Update newly discovered cheaper edge to reach v from the partial tree

- Similar to Dijkstra's shortest-path algorithm

- Correctness is due to the cut property

Time Complexity

```
MST-PRIM(G, w, r)
1  for each vertex u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = Ø
6  for each vertex u ∈ G.V
7      INSERT(Q, u)
8  while Q ≠ Ø
9      u = EXTRACT-MIN(Q)   // add u to the tree
10     for each vertex v in G.Adj[u]   // update keys of u's non-tre neighbors
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```

Running time: $O\big((V + E)\lg V\big)$ using Binary Heap

This is $O(E \lg V)$ if $|E| = \Omega(V)$

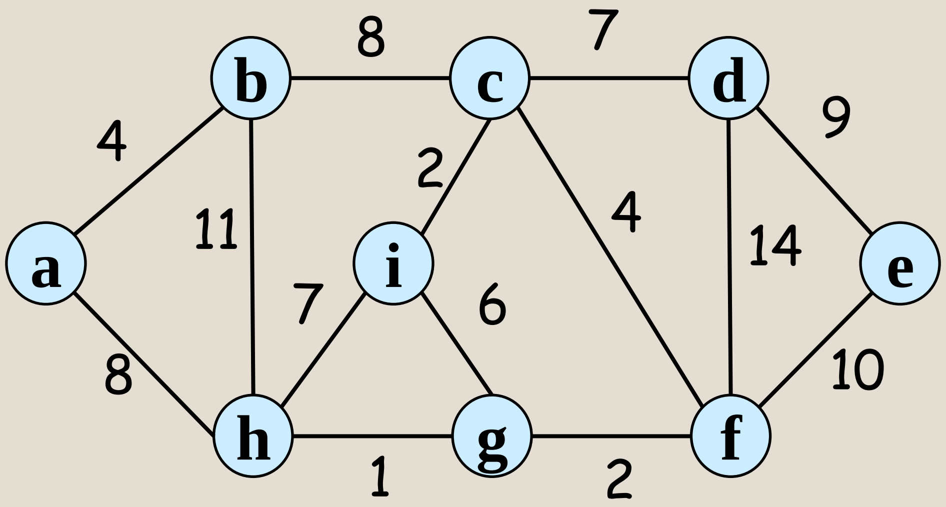
- Like Dijkstra's algorithm, running time depends on priority queue implementation
  - Assume that we use Binary Heap
- Line 6-7 can be done in $O(V)$ time by `Build-Min-Heap` procedure
  - Actually, `Build-Min-Heap()` isn't needed. Why?
- Line 9 takes $O(V \lg V)$ time
  - Total V calls to `Extract-Min`
  - `Extract-Min` takes $O(\lg V)$ time per call
- Line 14 takes $O(E \lg V)$ time
  - The inner loop executes $O(E)$ times in total (exactly $2|E|$ times)
  - Each call of `Decrease-Key` take $O(\lg V)$ time

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

All blue colored nodes are single-node trees

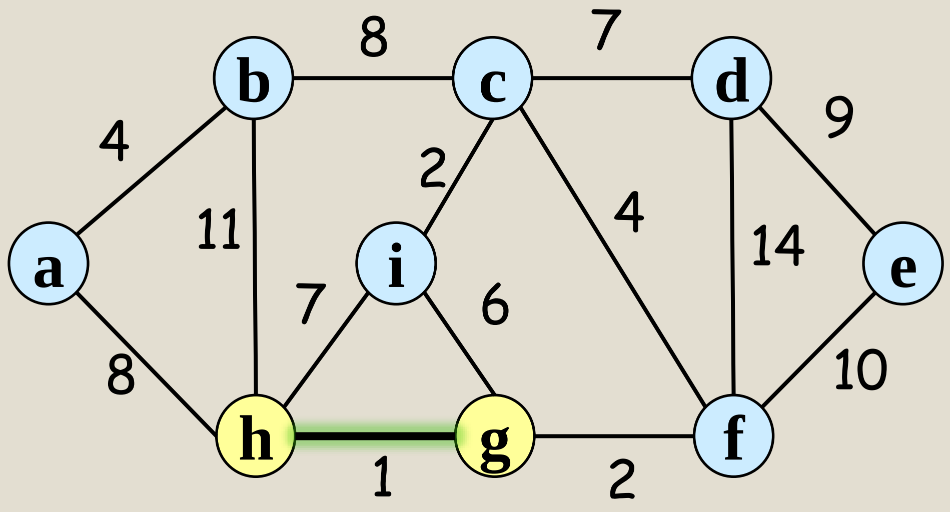
Example:



Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



All blue colored nodes are single-node trees

(h,g) forms a yellow-colored tree

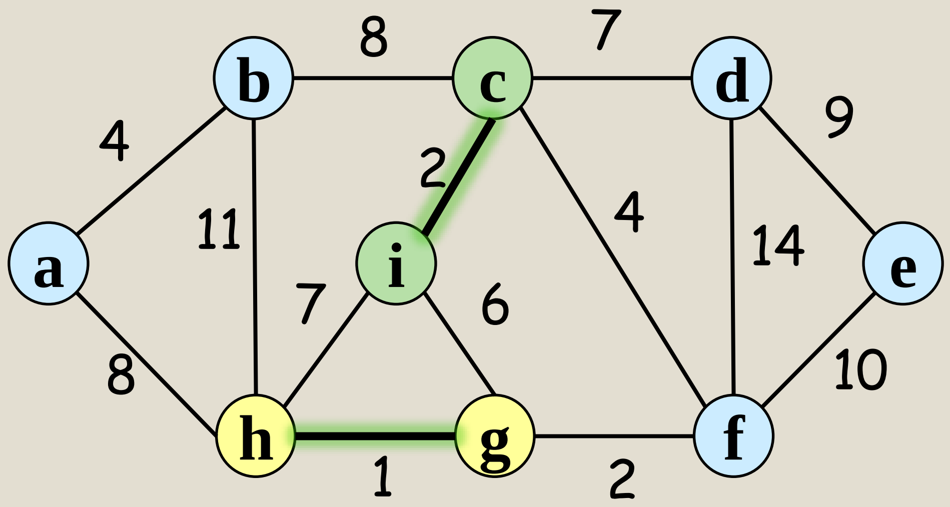
Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

All blue colored nodes are single-node trees

(c,i) forms a green-colored tree

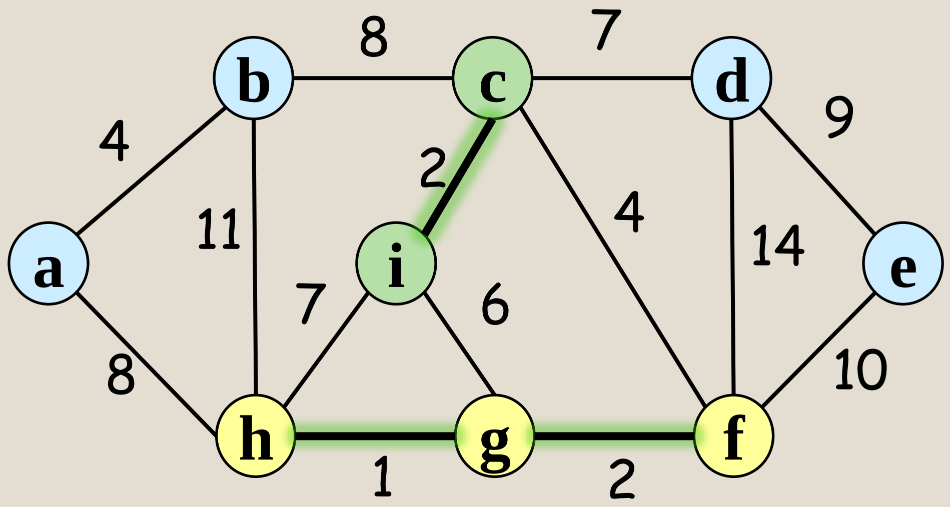
Example:



Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



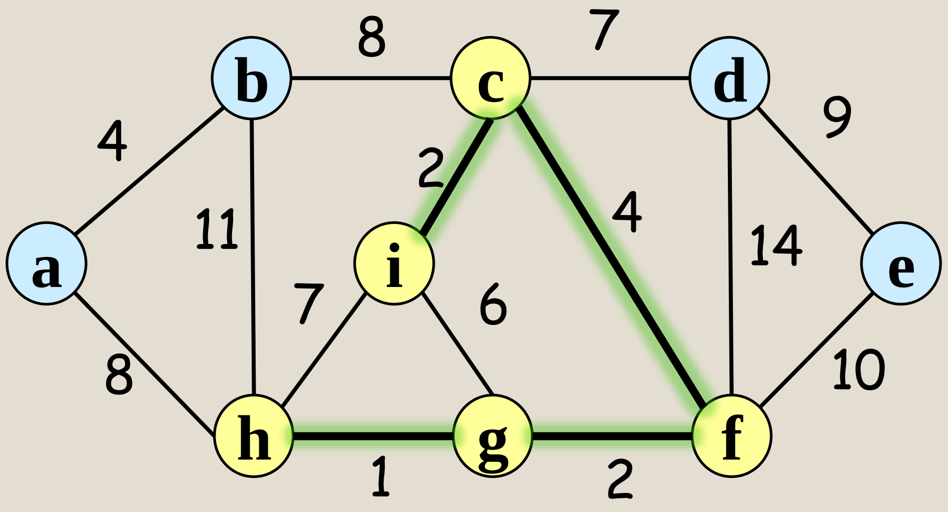
All blue colored nodes are single-node trees

(g,f) extends the yellow tree

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



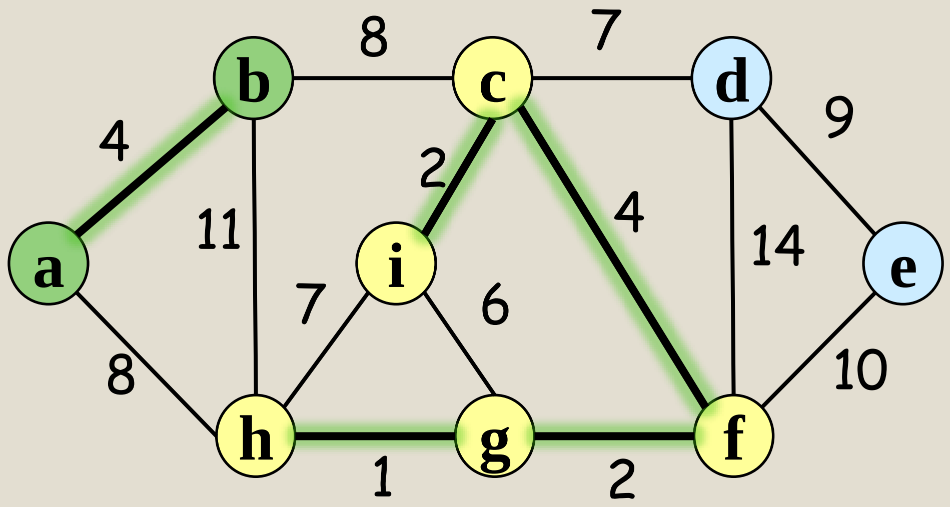
All blue colored nodes are single-node trees

Edge (c,f) combines yellow and green trees into a single tree (colored yellow, assume that majority color is retained)

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



All blue colored nodes are single-node trees

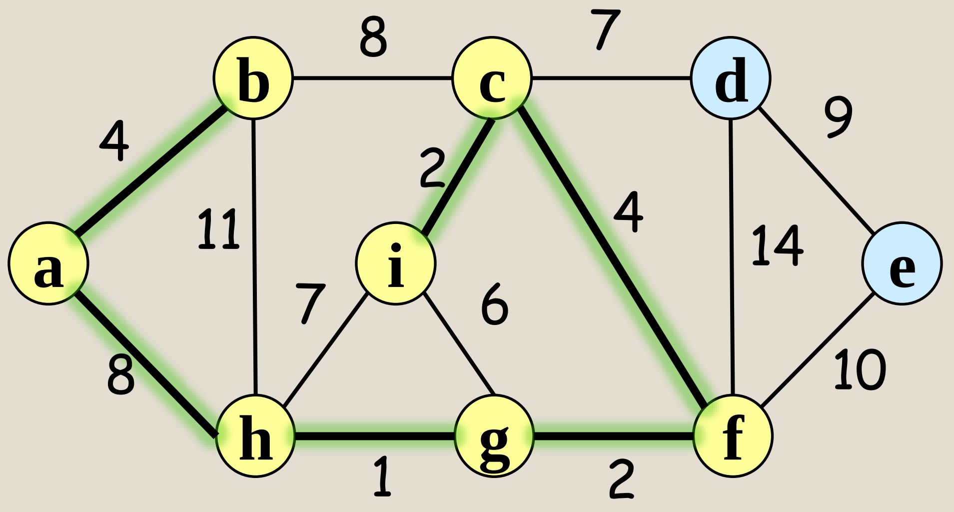
(a,b) forms a green-colored tree

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

All blue colored nodes are single-node trees

Example:



Assuming (a,h) wins tie-breaker, (a,h) combines yellow and green trees into a single yellow tree

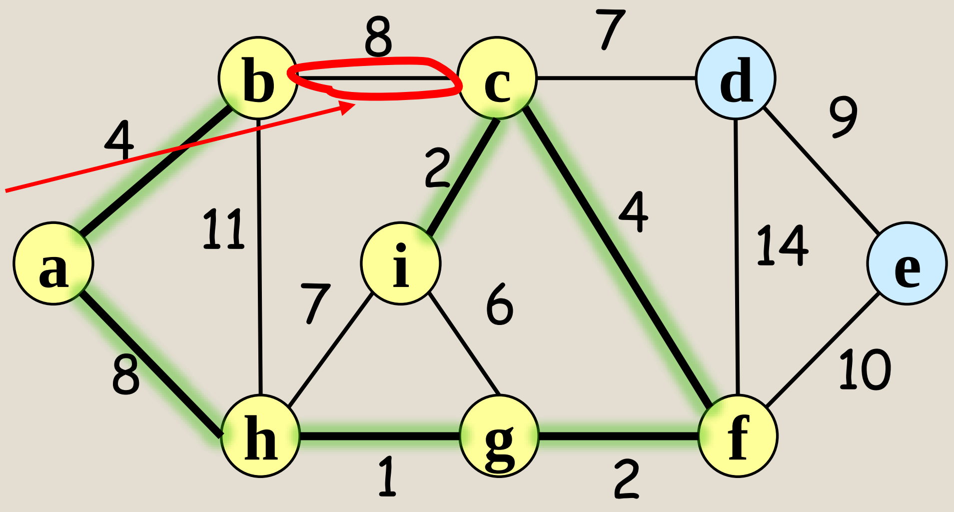
Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

All blue colored nodes are single-node trees

<u>Example</u>:

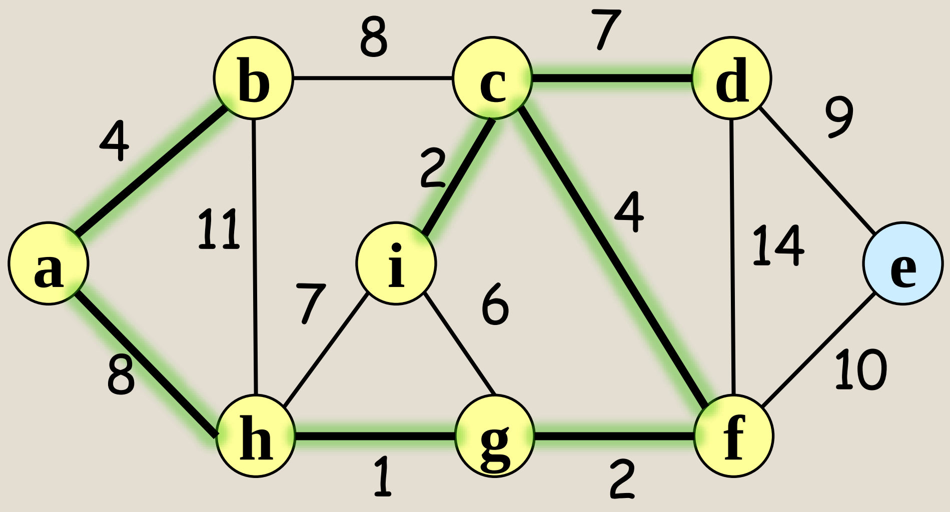
Cannot add this edge



Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



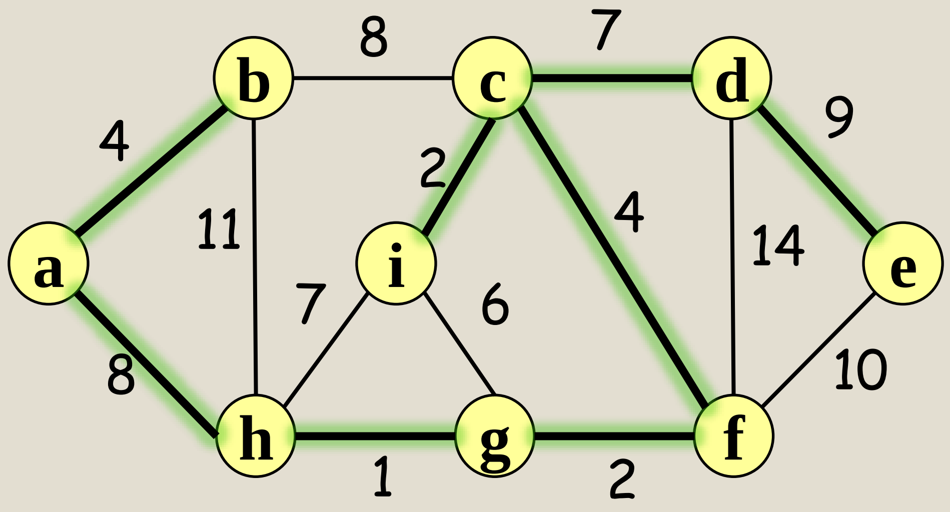
All blue colored nodes are single-node trees

(c,d) extends the yellow tree

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



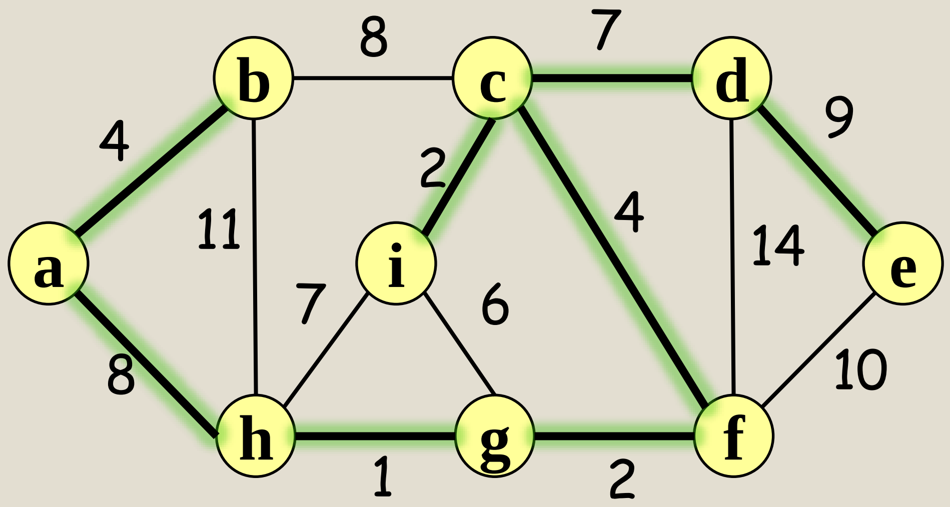
All blue colored nodes are single-node trees

(d,e) extends the yellow tree

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:



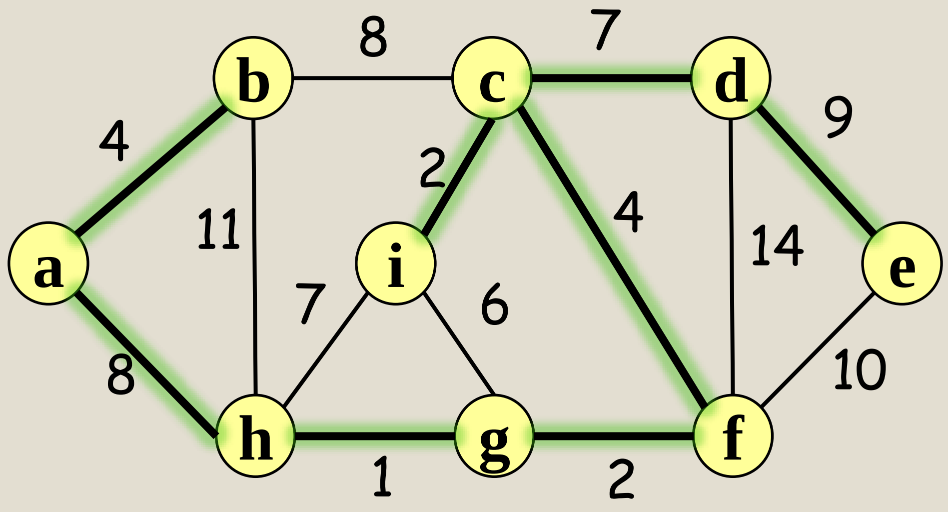
All blue colored nodes are single-node trees

(d,e) extends the yellow tree

Kruskal's Algorithm

- Instead of maintaining a partial tree, maintain a forest that eventually becomes MST
- Start with a forest of all $|V|$ nodes and no edges
- Consider edges in increasing order of weights and add an edge if it does not create a cycle

Example:

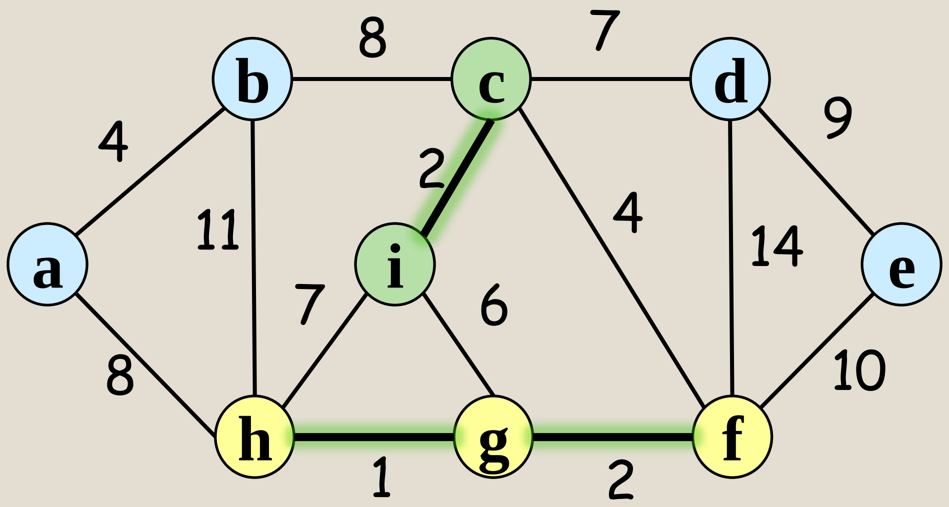


All blue colored nodes are single-node trees

(d,e) extends the yellow tree

Kruskal's Algorithm Implementation

- Need to maintain a set of trees
  - Example: $\{a\},\{b\},\{c,i\},\{f,g,h\},\{d\},\{e\}$ in the state of Kruskal's execution in the right



- After picking an edge, need to determine which set the endpoints belong to
  - Example: (c,f) is picked by the Kruskal's algorithm. We need to determine that they belong to sets $\{c,f\}$ and $\{f,g,h\}$, respectively
- Need to union to sets into one set
  - Example: After adding edge (c,f), the new combined set is $\{c,f,g,h,i\}$

- Three operations:
  - `Make-Set(v)`: make a set of node v
  - `Find-Set(v)`: find the set of node v
  - `Union(u,v)`: Combine the sets of u and v

Kruskal's Algorithm Implementation

"A" contains edges of the MST

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
create a single list of the edges in G.E
sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Does not create cycle if u and v are in different set.

Time Complexity

- Running time depends on running time of `Make-Set(v)`, `Find-Set(v)`, `Union(u,v)`, and Sorting of edges
- Sorting takes $O(E \lg E) = O(E \lg V)$ time (since $|E| = O(|V|^2)$)

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
create a single list of the edges in G.E
sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

	Make-Set(v)			Find-Set(v)			Union(u,v)			Sorting
	# calls	Running time per call	Total running time	# calls	Running time per call	Total running time	# calls	Running time per call	Total running time	
Trivial (linked-list)	$O(V)$	$O(1)$	$O(V)$	$O(E)$	$O(V)$	$O(VE)$	$O(E)$	$O(V)$	$O(VE)$	$O(E \lg V)$

Total running time with trivial implementation = $O(VE)$

Time Complexity

- Running time can be improved if we use Disjoint-Set data structures (CLRS Sec. 19)
- With Disjoint-Set, a sequence of k Make-Set, Union, and Find-Set takes $O\big(k \cdot \alpha(n)\big)$, here $n =$ the number of items
  - Here, $\alpha(n)$ is a very slow growing function
  - For simplicity, we use $O(k \lg n)$ instead

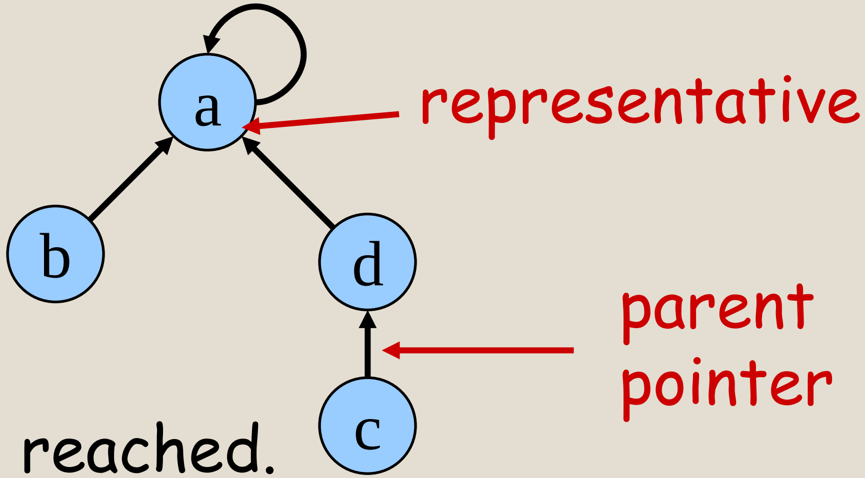
```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
create a single list of the edges in G.E
sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

	Make-Set(v) + Union(v) + Find-Set(V)		Sorting
	#calls	Total running time	
Disjoint-Set	$O(V+E)$	$O\big((V+E)\lg V\big)$ (which is $O(E\lg V)$ if $E = \Omega(V)$)	$O(E\lg V)$

Total running time with Disjoint-Set = $O(E\lg V)$

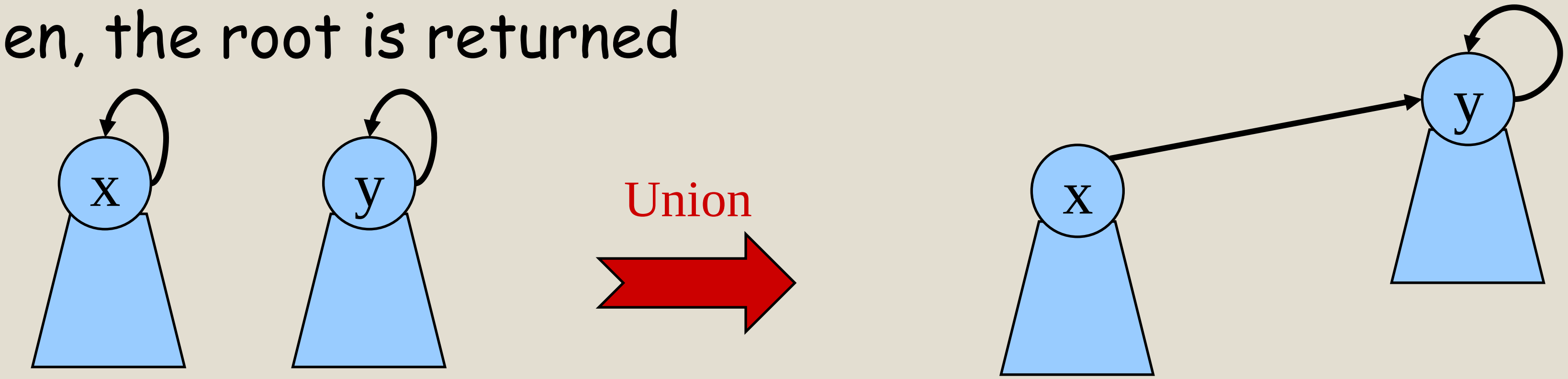
Disjoint-Set

- Each set is represented as a rooted tree
  - The root node is the representative of the set
  - `Find-Set()` returns the representative



- `Make-Set()`: Create a single-node tree
- `Find-Set()`: Traverse by parent pointer until root is reached.

Then, the root is returned



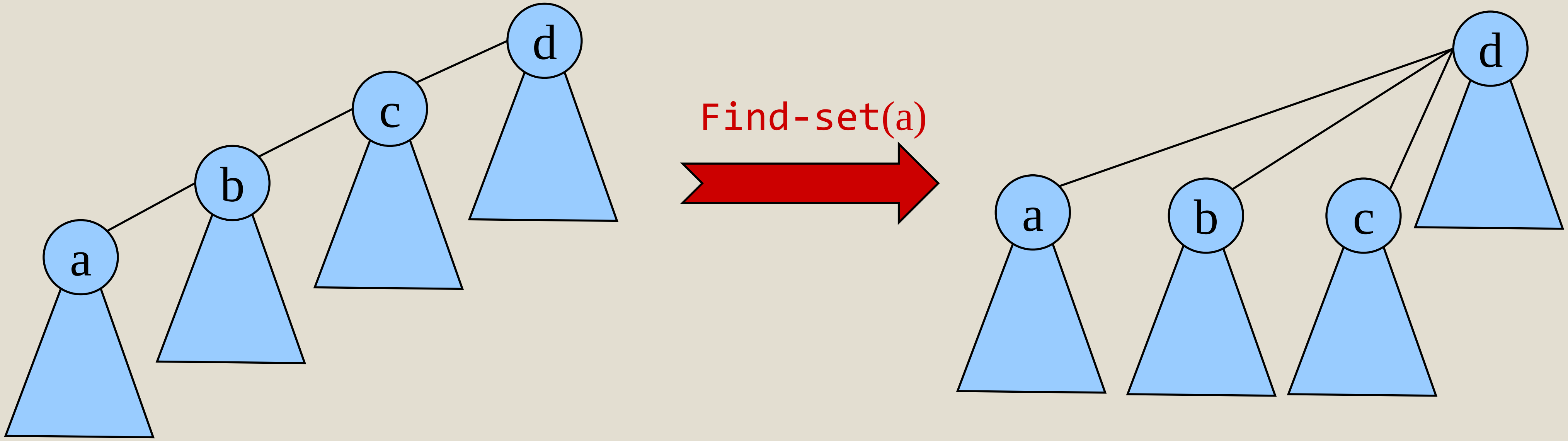
Can speed up sequence of the operations by means of two heuristics

Slide Courtesy: Prof. Anderson

Disjoint-Set

1) Union by Rank
   - Store rank of tree in rep.
     - Rank ≈ tree size.
   - Make root with smaller rank point to root with larger rank.

2) Path Compression
   - During `Find-Set`, "flatten" tree.



Conclusion

- MST can be computed in $O(E \lg V)$ time by Prim's and Kruskal's algorithm
  - Both are greedy algorithms
- Karger, Klein, and Tarjan gave an algorithm that runs in $O(E \lg V)$ at worst case, but $O(V + E)$ in average case
  - https://cs.brown.edu/research/pubs/pdfs/1995/Karger-1995-RLT.pdf
- Bernard Chazelle gave an $O(E\ \alpha(|E|, |V|)$-time algorithm (known best so far!)
  - $\alpha(\cdot,\cdot)$ is the slow growing function mentioned in disjoint set data structure
- It is still open whether MST can be solved in $O(V + E)$ worst-case running time.

Thank You!