Lecture 20: Minimum Spanning Trees

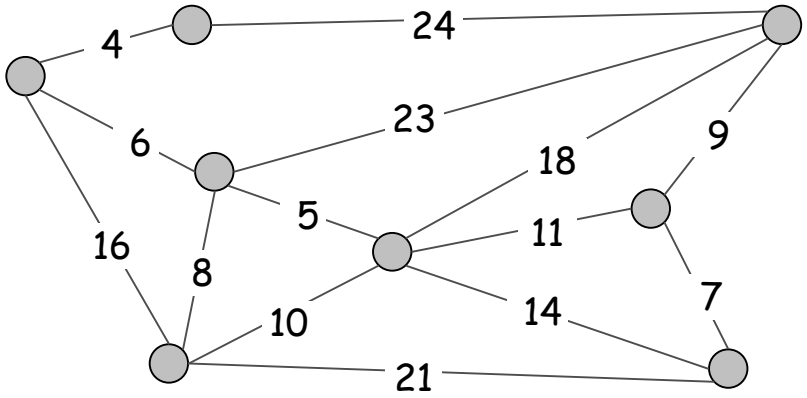
- A (simple) connected undirected graph G(V,E) has between V-1 and V(V-1)/2 edges.
  - $E=O(V^2)$
  - $O(\log E)=O(\log V^2)=O(2\log V)=O(\log V)$
- An undirected graph G(V,E) with fewer than V-1 edges is not connected.
- A connected undirected graph G(V,E) with V-1 edges, is a tree.
- An undirected graph G(V,E) with more than V-1 edges contains at least one cycle.

Minimum Spanning Trees

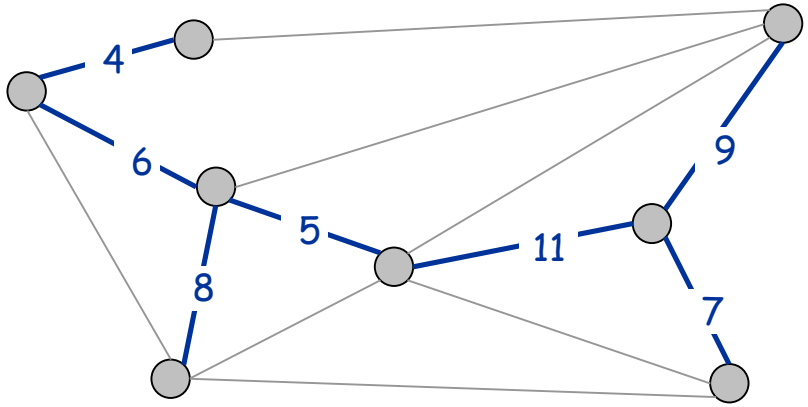
- Definition
- Prim's Algorithm
- The Cut Lemma
  - Correctness of Prim's Algorithm
  - Uniqueness of MSTs (under distinct weight assumption)
- Kruskal's Algorithm
  - Basic Idea
  - Union-Find Data Structure
  - Kruskal's algorithm
  - Correctness of Kruskal's Algorithm
- Removing distinct weight assumption

Minimum Spanning Tree Example

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.



$G = (V, E)$



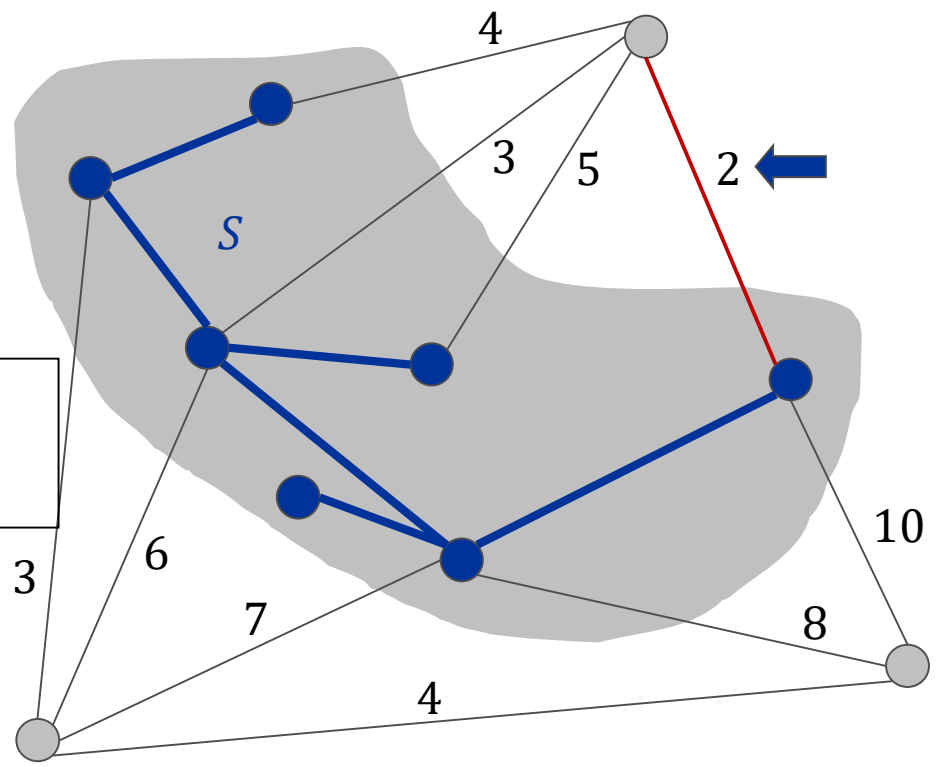
$T, \Sigma_{e \in T}\, w(e) = 50$

Applications: telephone networks, electrical and hydraulic systems , TV cables, computer networks, road systems

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 $S = V$



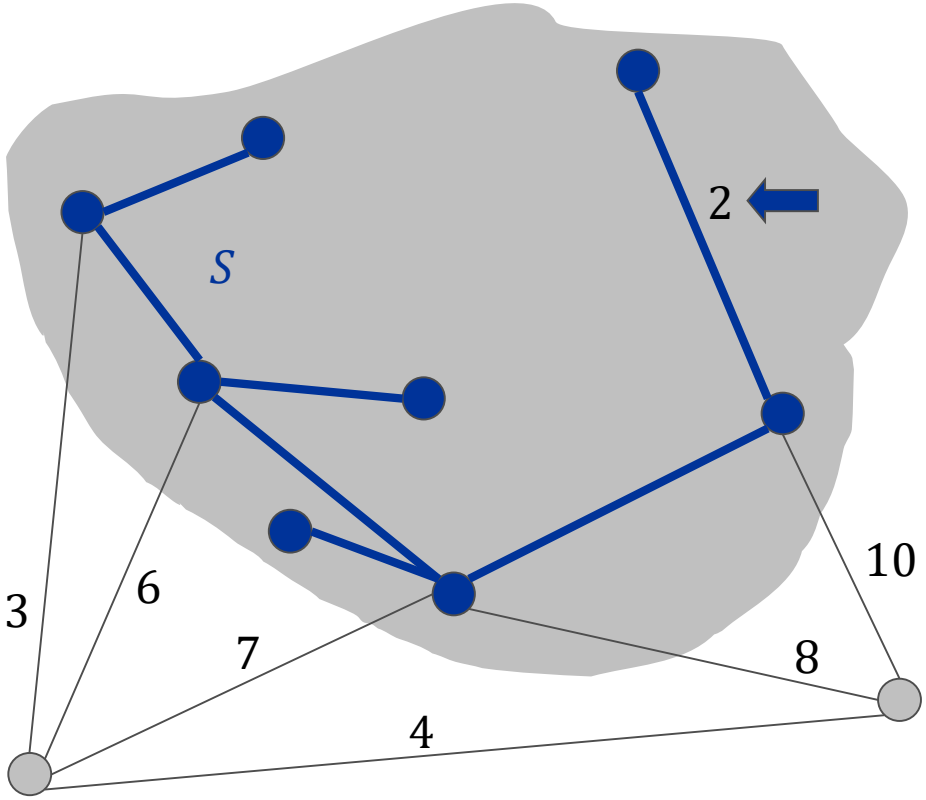
S = blue vertices

e = red (lightest) edge

Prim's Algorithm: Idea (cont)

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 $S = V$

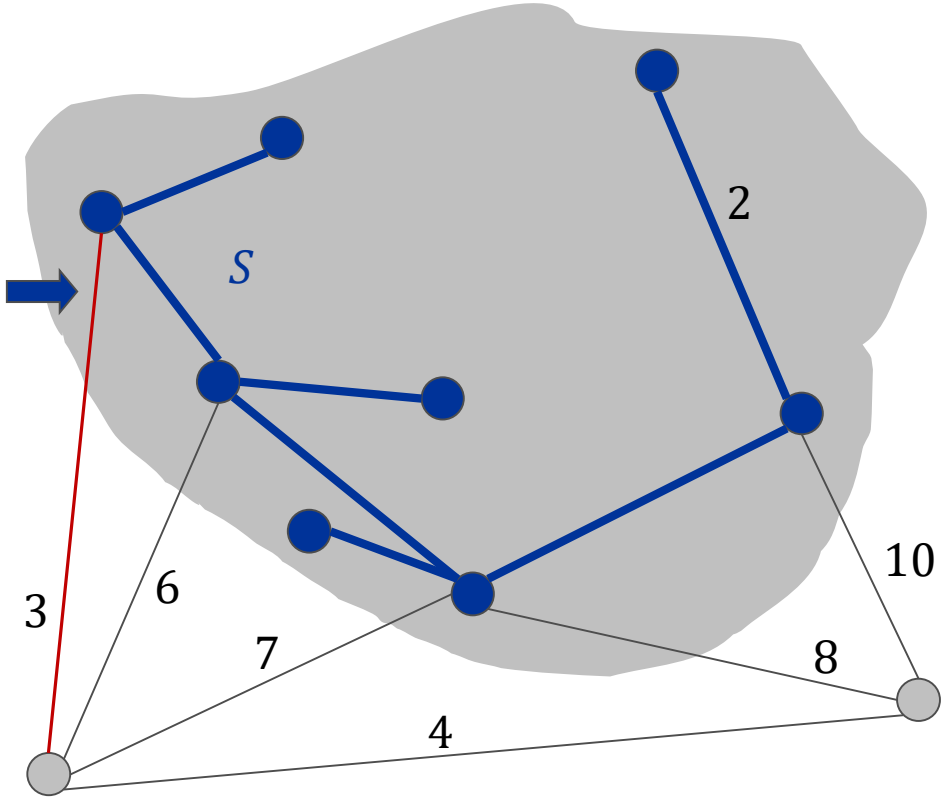


Prim's Algorithm: Idea (cont)

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 $S = V$

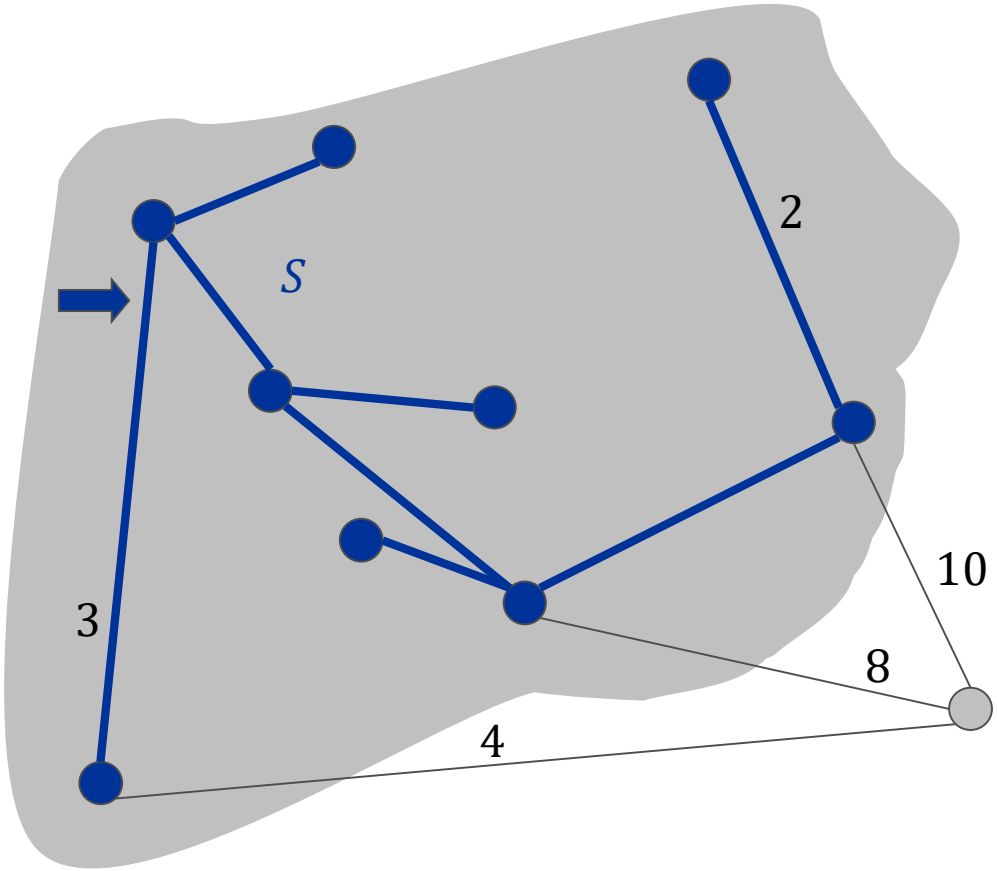
S = blue vertices
e = red edge



Prim's Algorithm: Idea (cont)

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 $S = V$



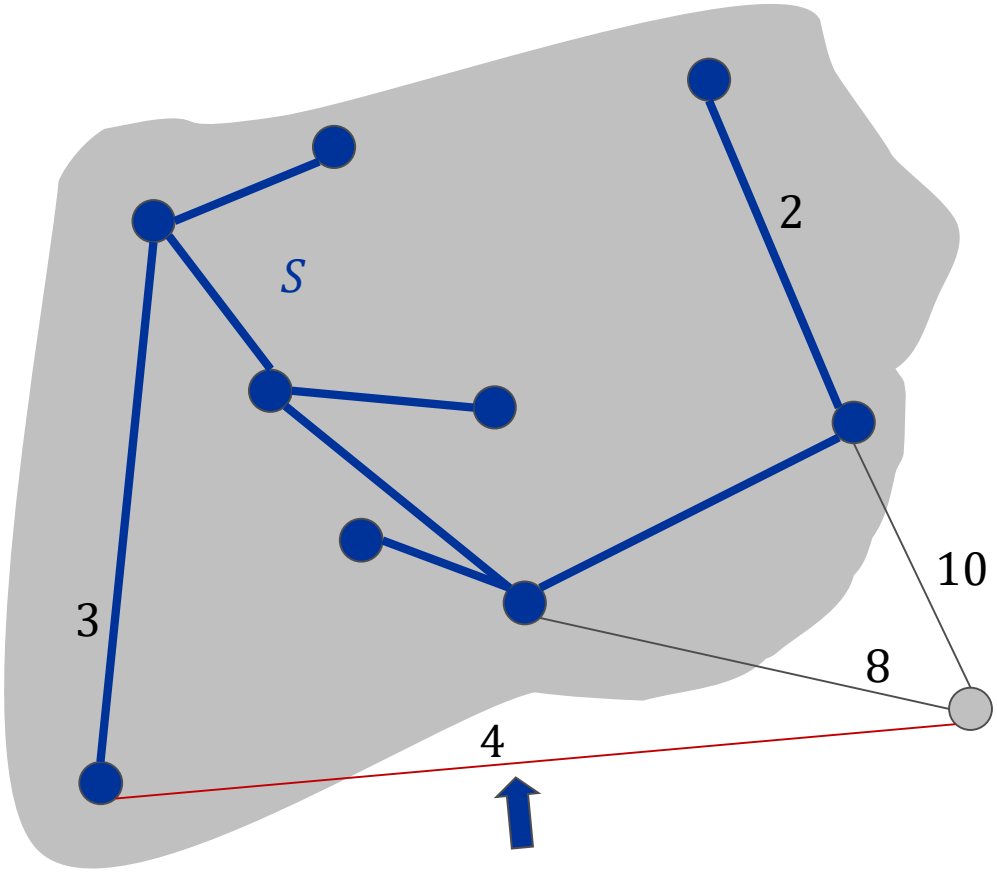
Prim's Algorithm: Idea (cont)

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 $S = V$

S = blue vertices

e = red edge

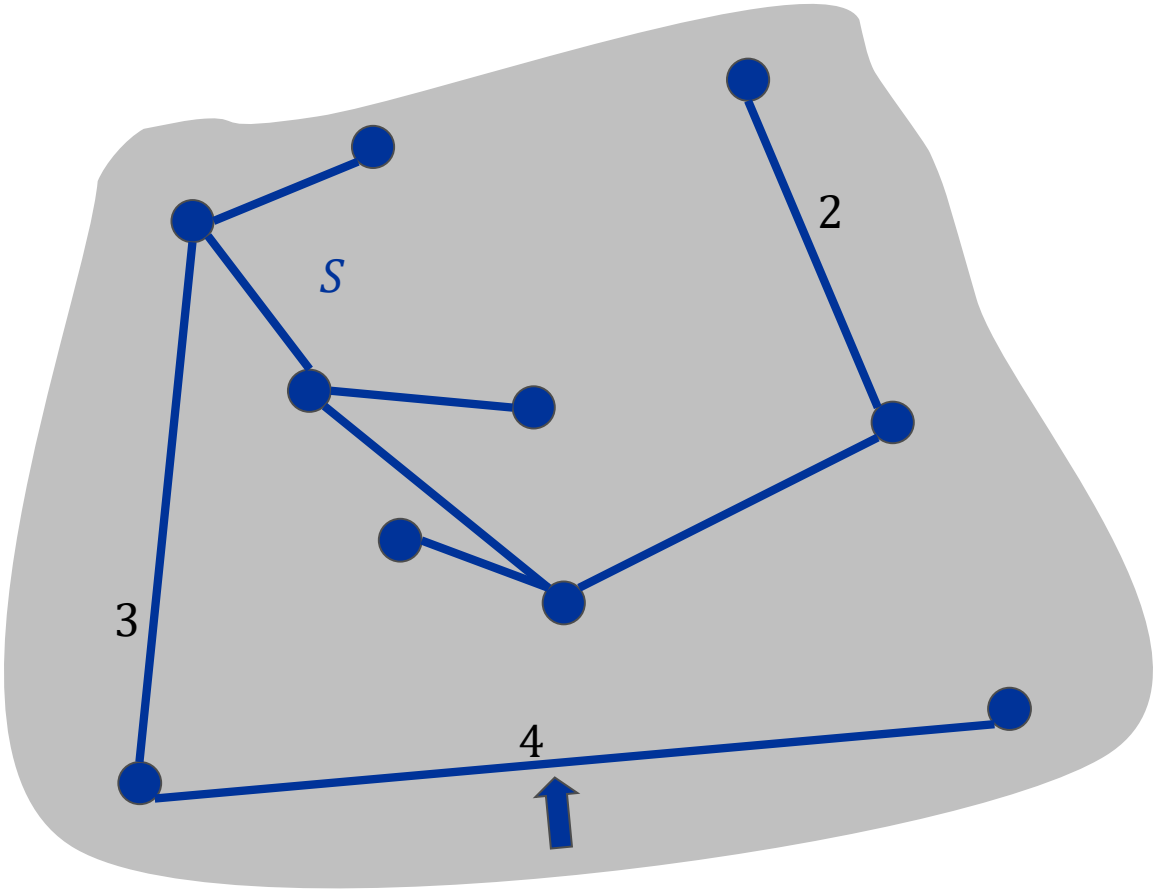


Prim's Algorithm: Idea (cont)

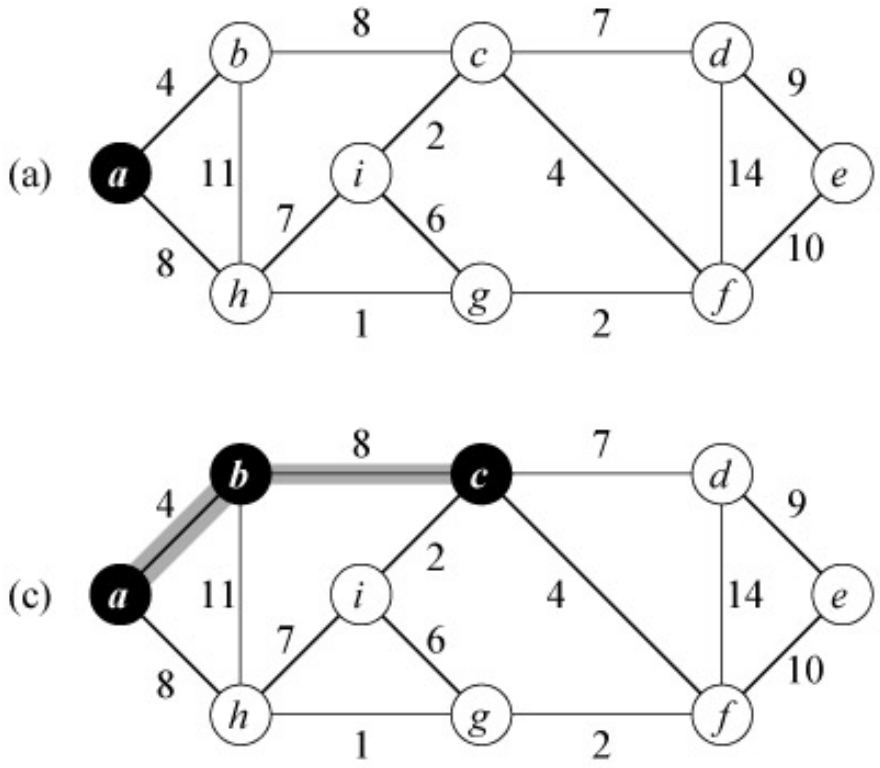
Prim's algorithm

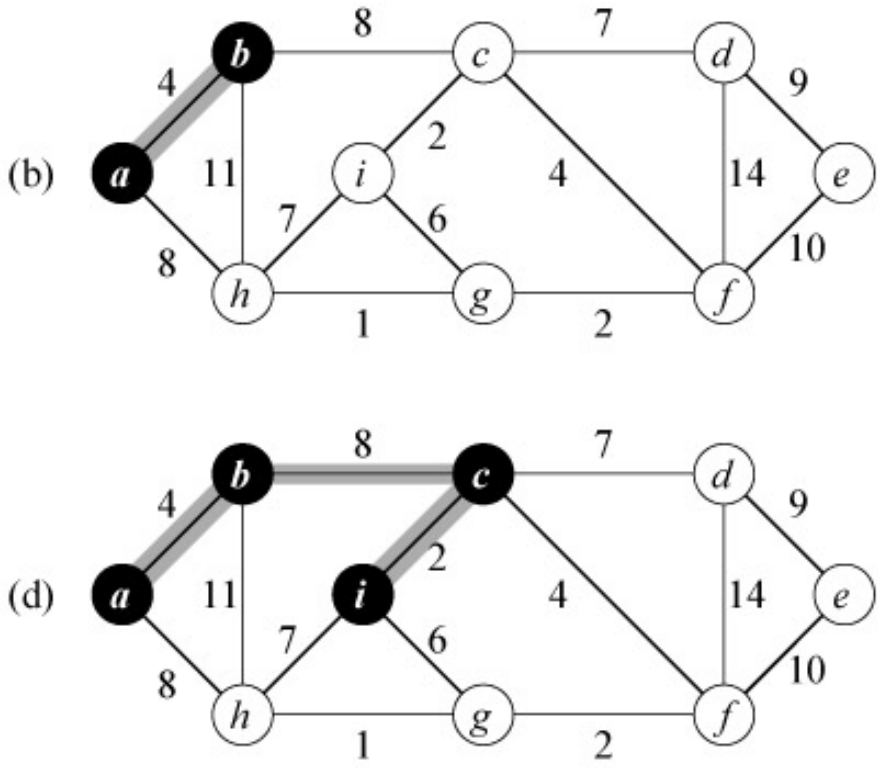
- Initialize S = {any one node}.
- Add min cost edge $e = (u, v)$ with $u \in S$ and $v \in V - S$ to T.
- Add v to S.
- Repeat until $S = V$

S = blue vertices
$= V$
form a tree



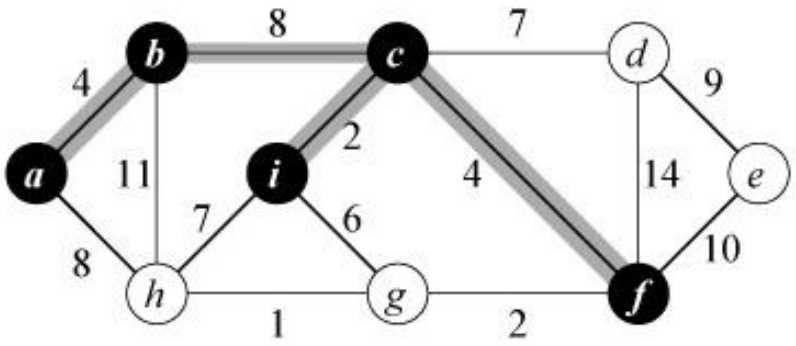
Prim's Algorithm: Example



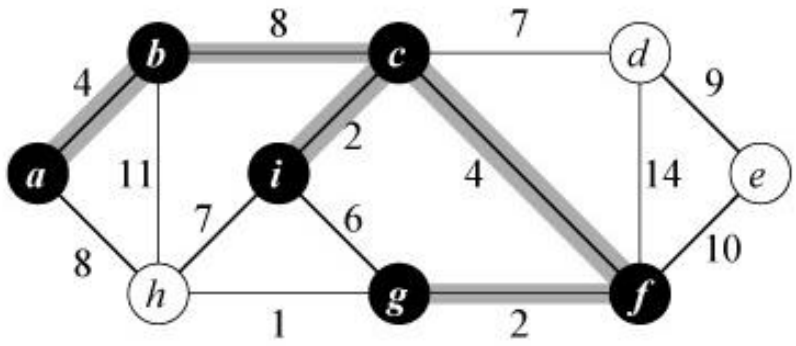


Prim's Algorithm: Example (continued)

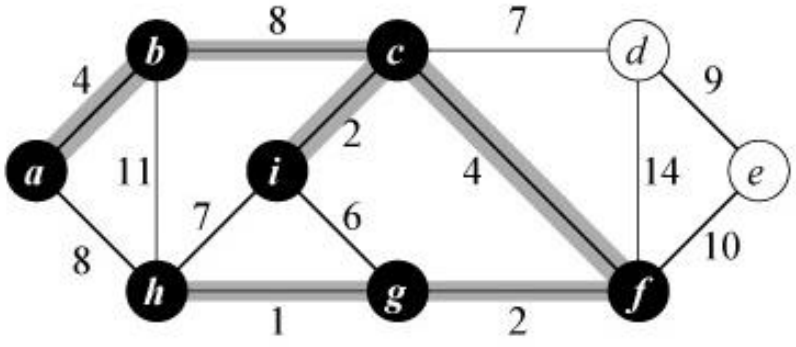
(e)



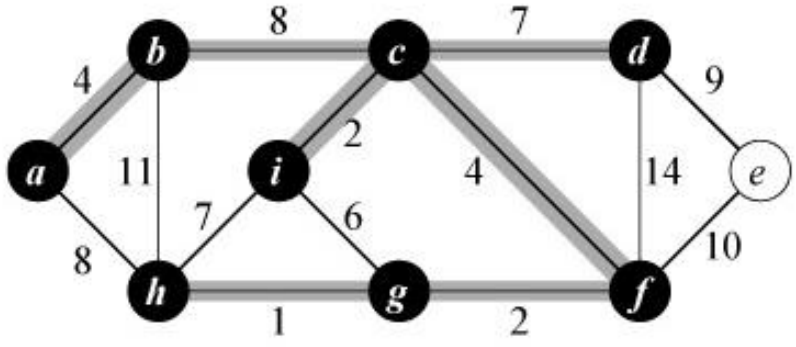
(f)



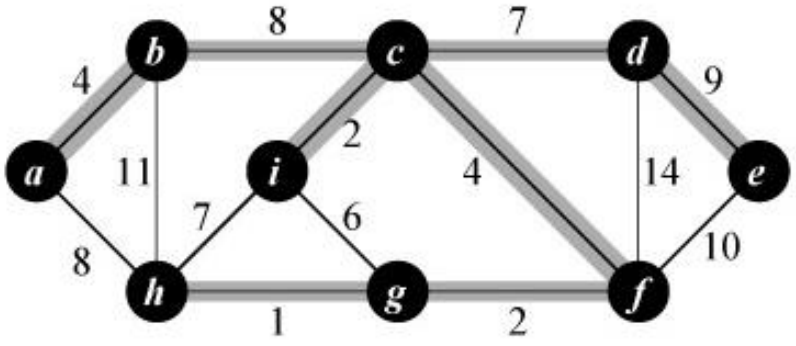
(g)



(h)



(i)



Prim's Algorithm: Implementation

Implementation.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain ***cheapest*** edge from v to 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
insert all keys in 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?

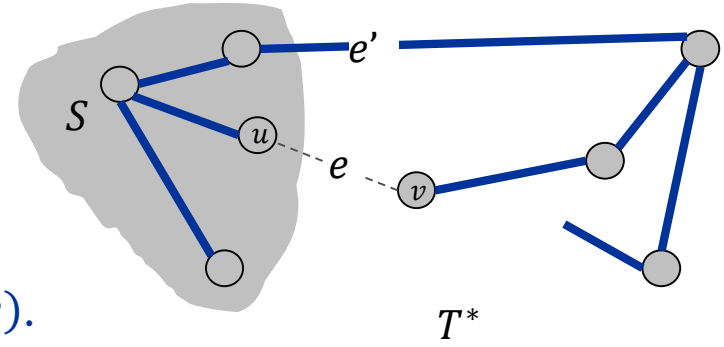
Cut Lemma

Simplifying assumption. All edge weights are distinct.

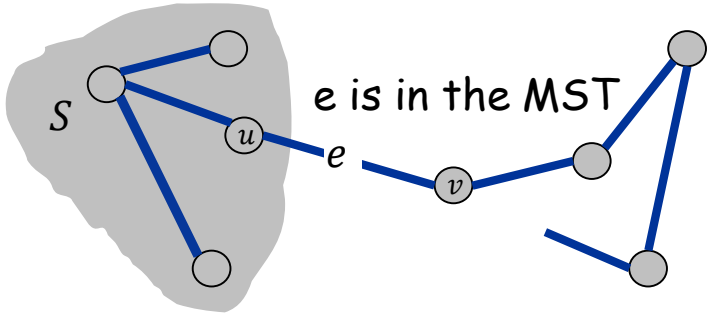
Cut lemma. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then every MST must contain e.

Pf of Cut Lemma. (exchange argument)

- Let T^* be any MST.
- Let $e = (u, v)$ and suppose $e \notin T^*$.
- There is a path in T^* that connects u to v, which must cross cut separating S from $V - S$ using some other edge $e' \in T^*$ with $w(e') > w(e)$.



- If we replace e' in T^* with e, then T^* is still a spanning tree, but the total cost will be lowered, **contradicting fact that T^* is an MST.**

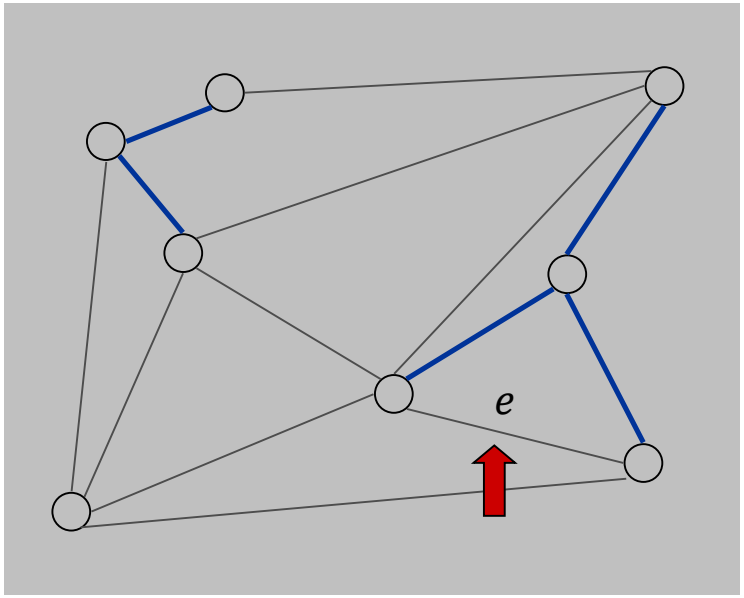


=> e is in every MST!

Kruskal's Algorithm: Idea

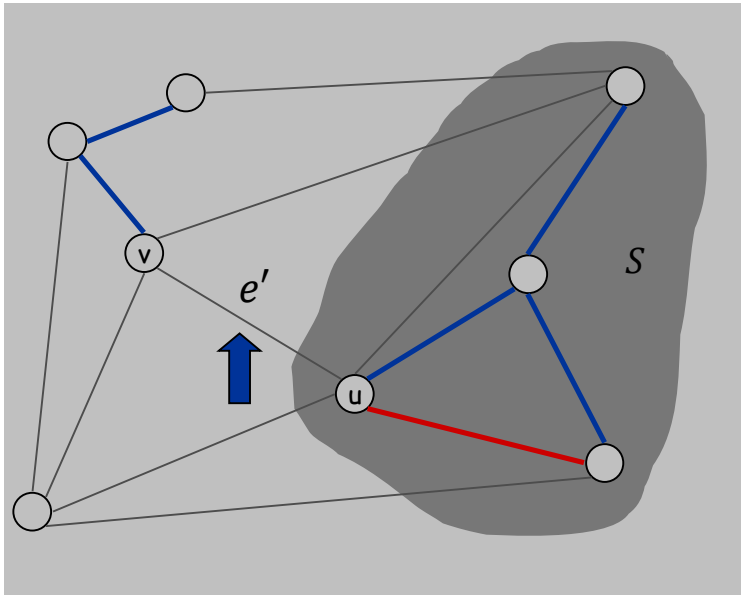
Kruskal's algorithm.

- Starts with an empty tree T
- Consider edges in increasing 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

e creates cycle.
Don't add e to T

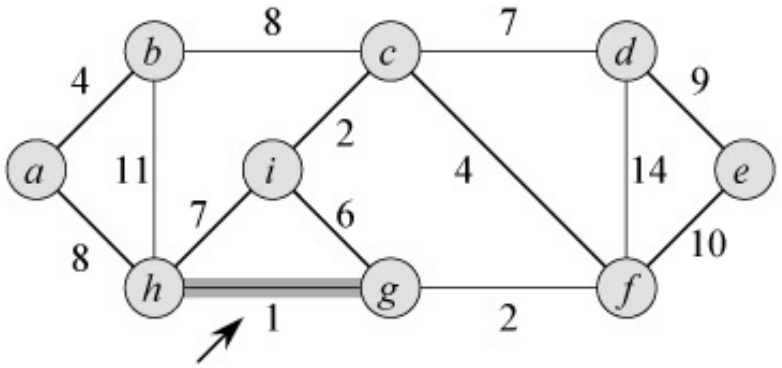


Case 2

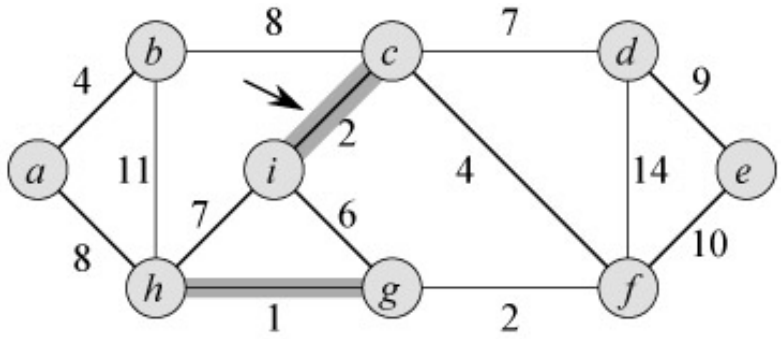
e' doesn't create cycle.
Add e' to T

Kruskal's Algorithm: Example

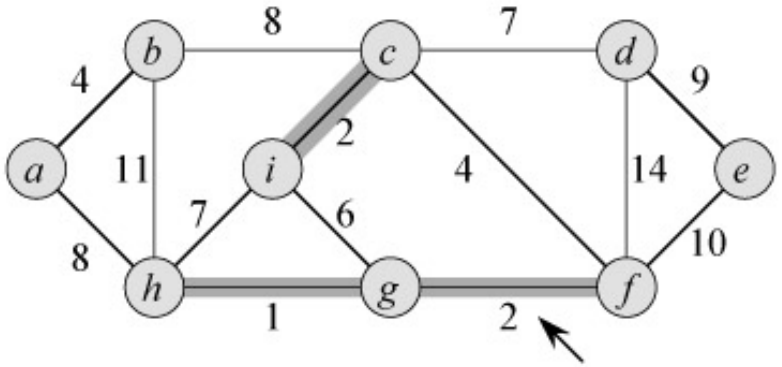
(a)



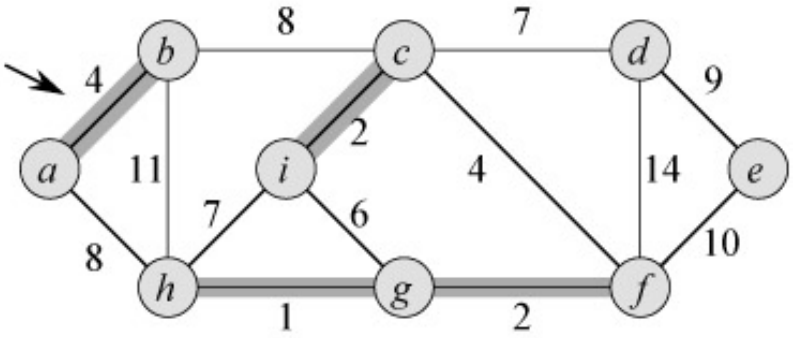
(b)



(c)

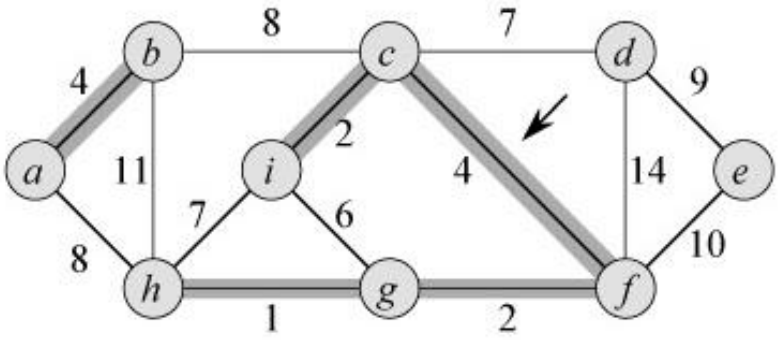


(d)

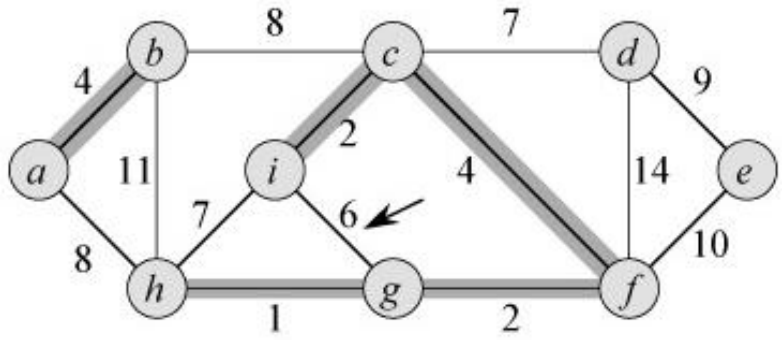


Kruskal's Algorithm: Example (continued)

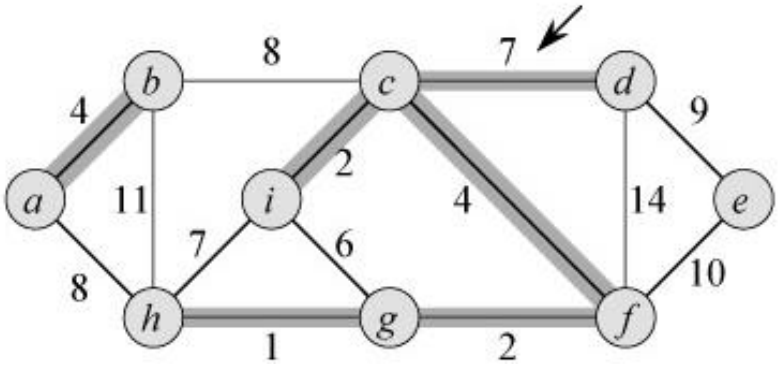
(e)



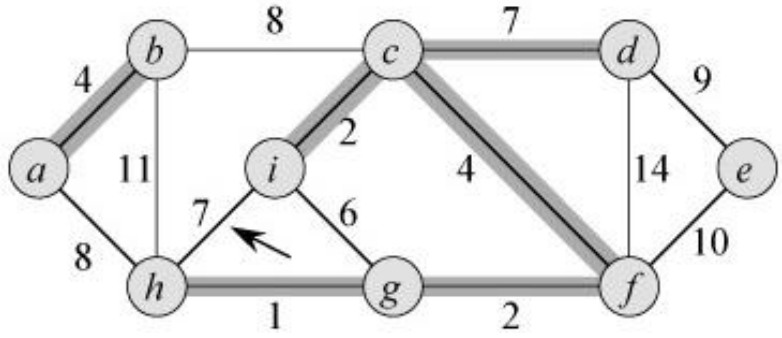
(f)



(g)

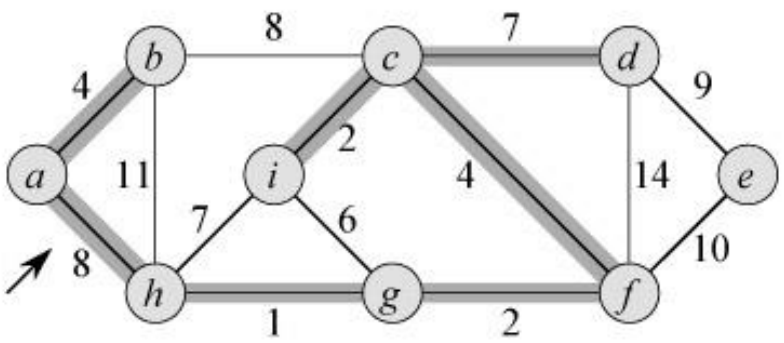


(h)

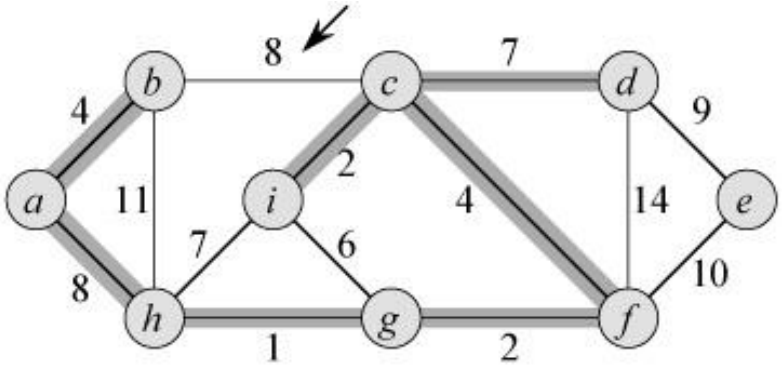


Kruskal's Algorithm: Example (continued)

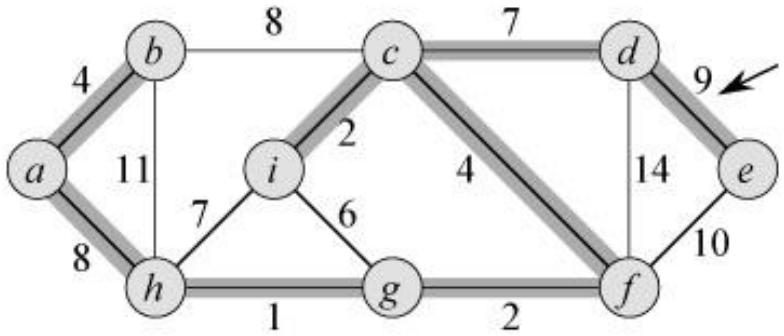
(i)



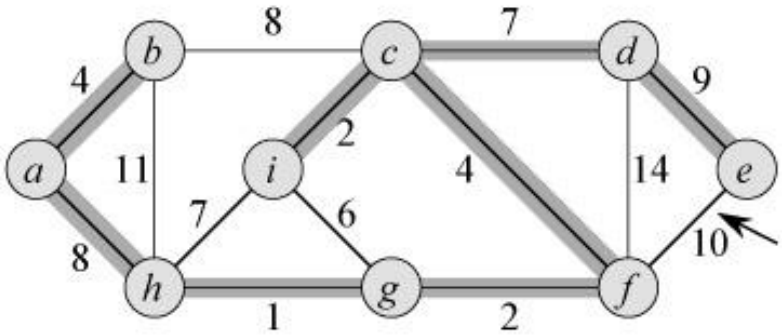
(j)



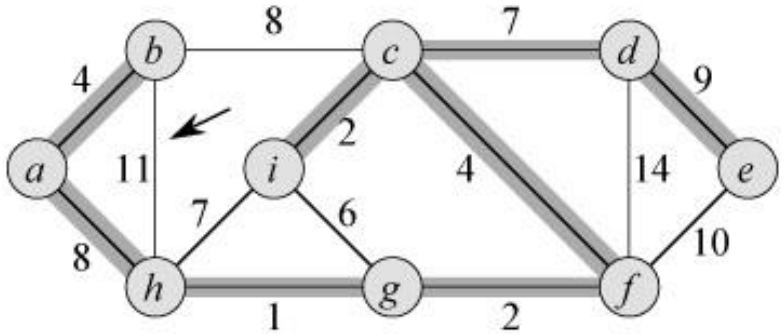
(k)



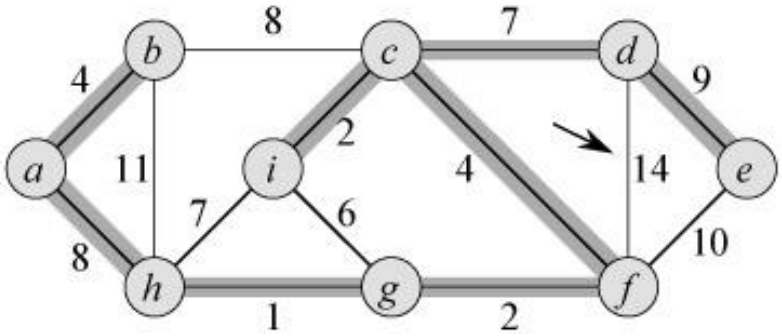
(l)



(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?

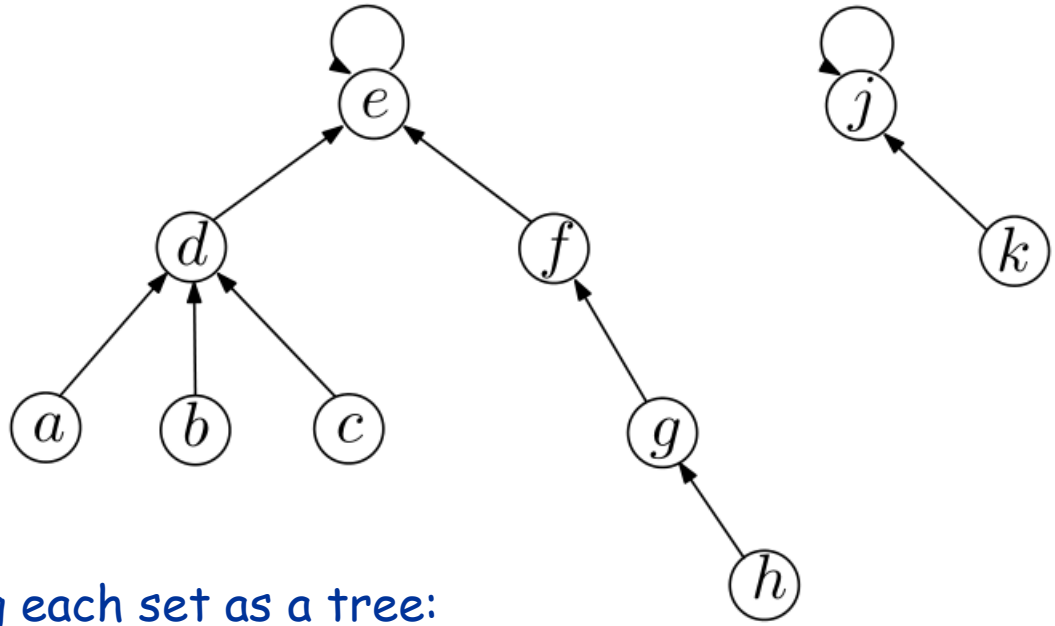
Observations:

- 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 trees in the union-find data structure are NOT the same as the partial MST trees!
- 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, 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
```

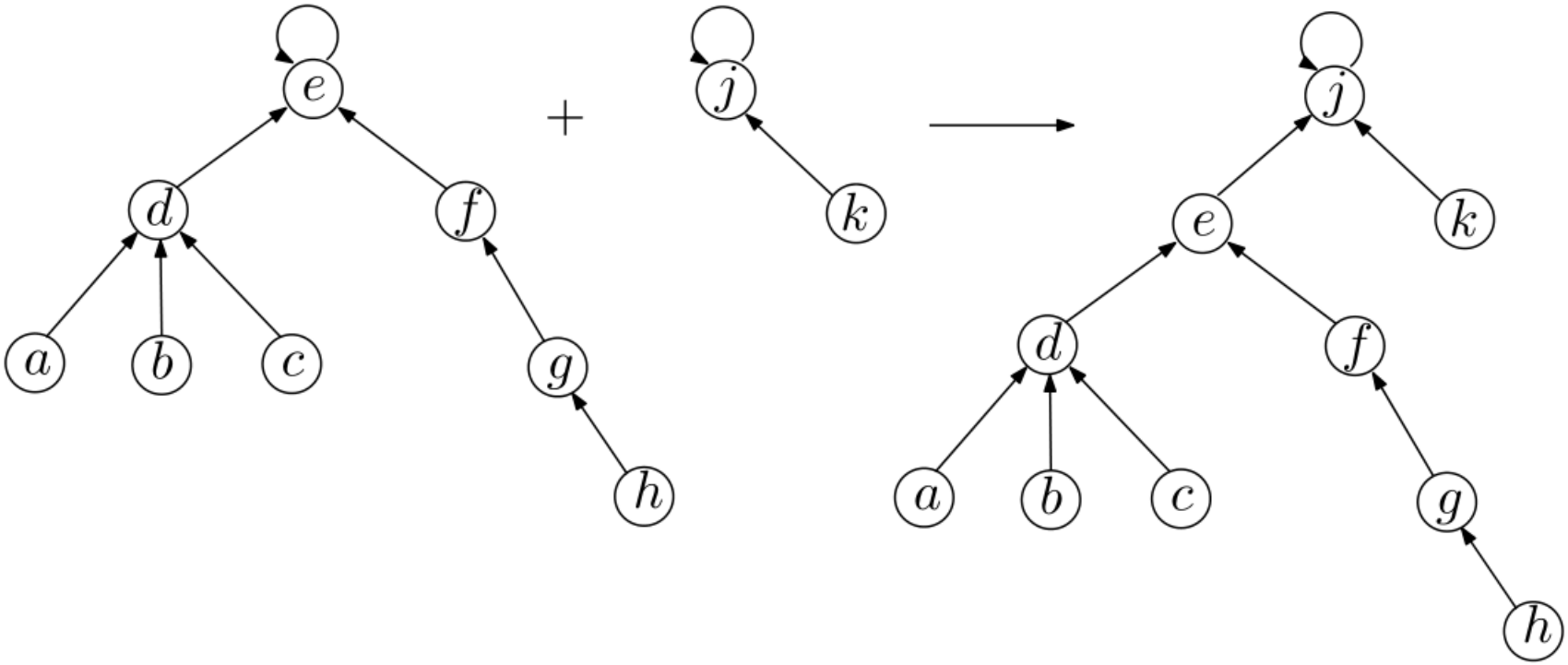
Running time proportional to the height of the tree.

Union(x, y)

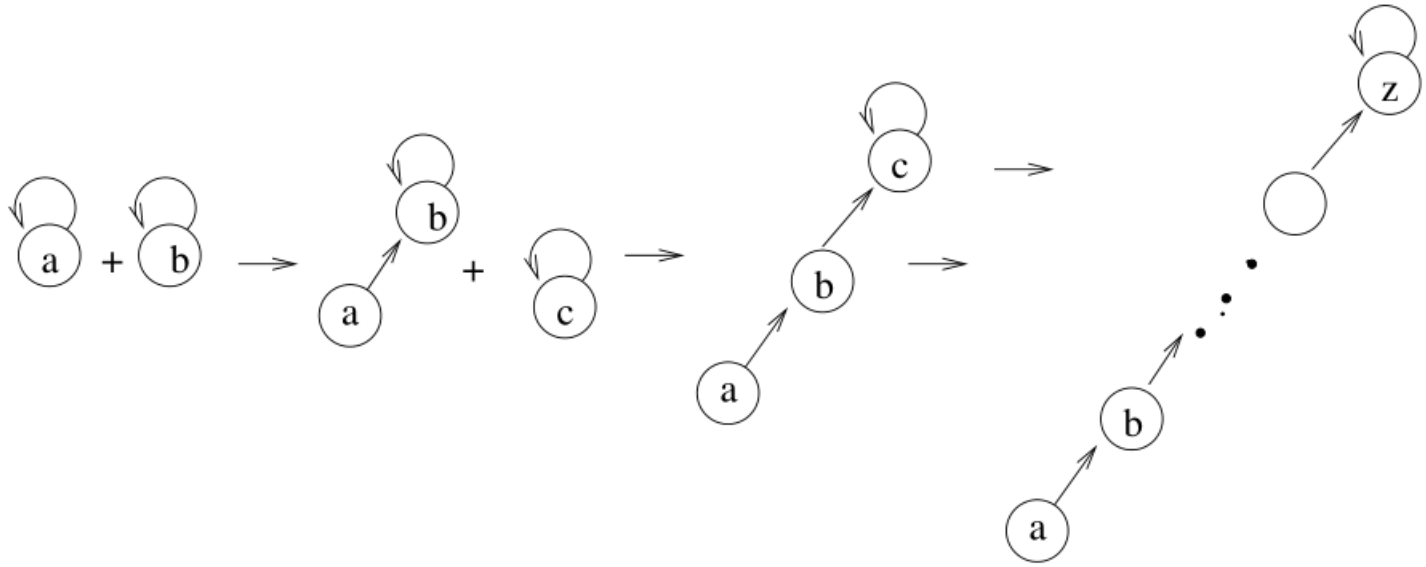
Assumption: x and y are the roots of their trees.

- 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)**:**

$a \leftarrow$ **Find-Set**(x)

$b \leftarrow$ **Find-Set**(y)

if $a.height \leq b.height$ **then**

  if $a.height = b.height$ **then**

    $b.height \leftarrow b.height + 1$

  $a.parent \leftarrow b$

else

  $b.parent \leftarrow a$

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 (i.e., it contains at least 2^h nodes)

- 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 $h(x')$.

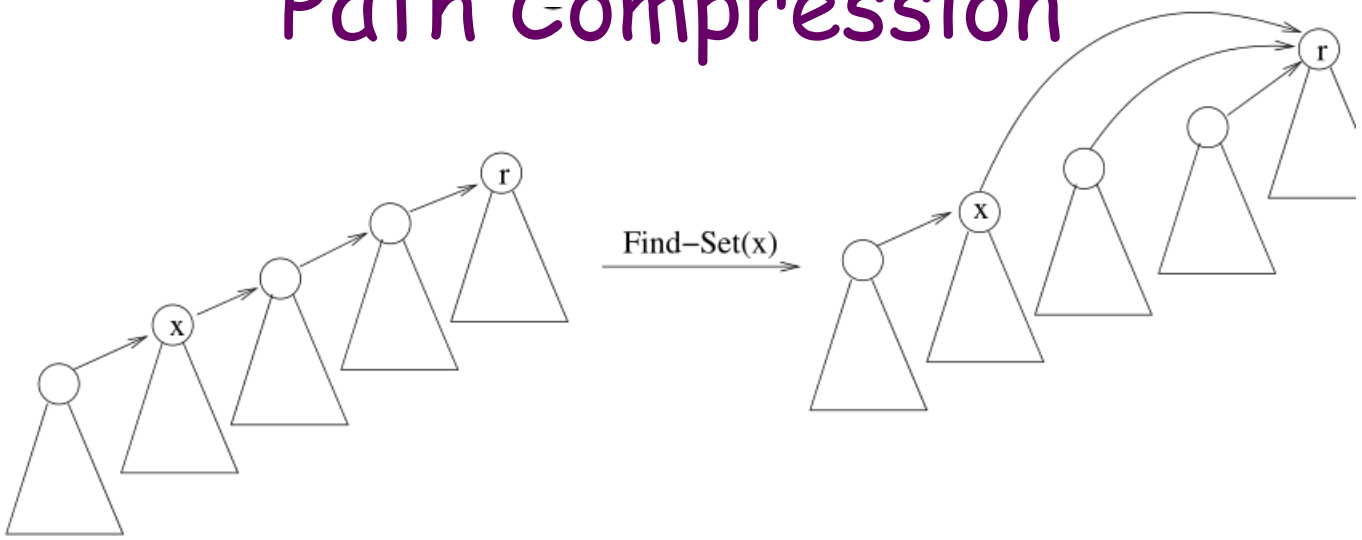
  - Case 1: $h(x) < h(y)$, we have
    $size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}$.

  - 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.

- Consider any tree during the running of the algorithm. It must have $\leq n$ elements. Since a tree with height $\log n$ has at least n elements the tree has height $\leq \log n$. Thus all operations take $O(\log n)$ time.

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 $O(E)$.

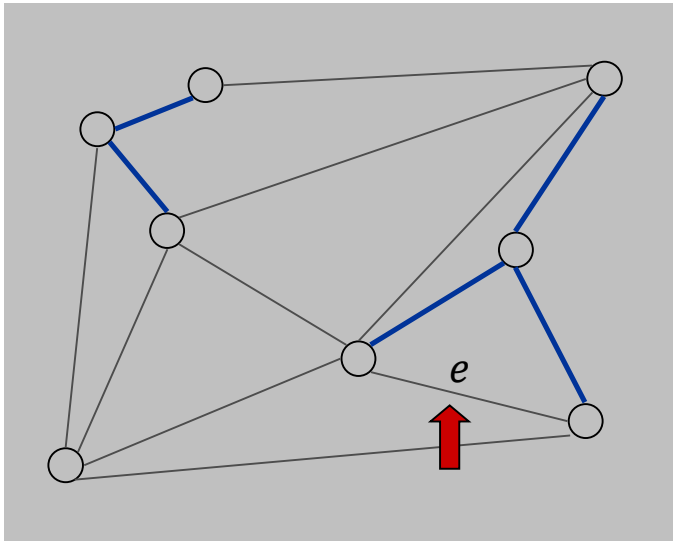
Current best MST algorithm:

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

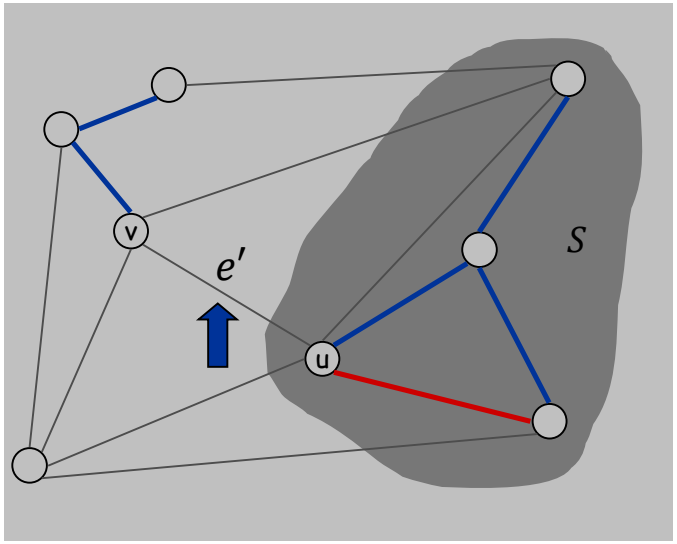
Correctness of Kruskal's Algorithm

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 e creates cycle. Don't add e to T



Case 2 e' doesn't create cycle. Add e' to T

Simplifying assumption. All edge weights are distinct.

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

Observations on Prim's and Kruskal's algorithms

Both algorithms are Greedy since they make the choice that looks best at the moment

- Prim adds to MST lightest edge from S
- Kruskal adds to MST lightest edge that does not create a cycle.

For both Prim's and Kruskal's algorithm, we assumed that all the edges have different weights.

If we remove this assumption (and allow some or even all edges to have the same weight) the algorithms still work. The only thing that needs to be changed is that, instead of choosing **the** smallest cost edge, we choose **a** smallest cost edge (breaking ties arbitrarily).

Questions on Minimum Spanning Trees

Q: Suppose that (u, v) is an edge of a graph G and it has the least cost among all edges that are incident to vertex v. Does every minimum spanning tree contain (u, v) assuming that all edges have distinct costs?

Yes. Suppose that T is a minimum spanning tree of G and T does not contain (u, v). If (u, v) is added to T, then a cycle is formed. Assume that (w, v) is the other edge in the cycle that is incident to v. Replacing (w, v) with (u, v) in T will induce a spanning tree whose cost is smaller than T, a contradiction.

Q: Let G be an undirected connected graph with distinct edge weights. Let e_{max} be the edge with maximum weight and e_{min} the edge with minimum weight. Which of the following statements is false?

1. Every minimum spanning tree of G must contain e_{min}.
2. No minimum spanning tree contains e_{max}
3. If e_{max} is in a minimum spanning tree, then its removal must disconnect G
4. G has a unique minimum spanning tree

1,3,4 are true

Exercise on Maximum Spanning Trees

Design an algorithm for finding the maximum spanning tree; i.e., the spanning tree that maximizes the sum of edge weights.

Solution. I sort the edges in decreasing order of weights and I keep adding edges, provided that I do not create cycles.

Exercise on Bottleneck Weight

Let $G = (V,E)$ be a connected undirected graph with weights on the edges. The *bottleneck weight* of any spanning tree T of G is the maximum weight of an edge in T. Prove that the minimum spanning tree (MST) minimizes the bottleneck weight over all spanning trees.

Solution. We prove by contradiction.

Let T be the MST, whose heaviest edge is (a, b).

Removing (a, b) from T divides T into two connected components A (containing a) and B (containing b).

Assume another spanning tree T' whose bottleneck weight is smaller than that of T.

In T', there must be a unique path from a to b; therefore there must be an edge (a', b') such that $a' \in A$ and $b' \in B$. According to the assumption, the heaviest edge in T is heavier than any edge in T', so (a', b') is lighter than (a, b).

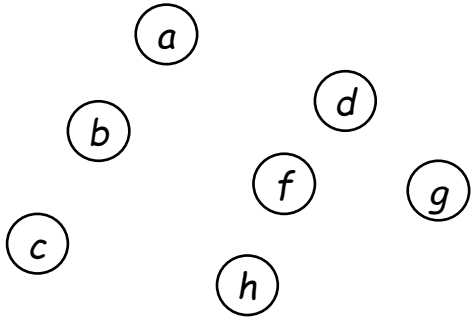
Adding (a', b') will connect A and B to form a new spanning tree whose total weight is smaller than T's, which contradicts the assumption that T is MST.

Exercise on Euclidean Traveling Salesman (Optional)

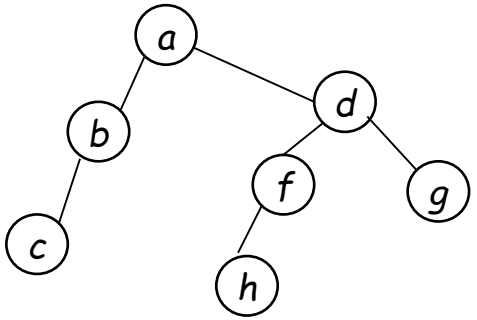
- Input: Undirected Graph $G = (V,E)$ and a cost $C(e)$ for each edge e.
- Output: A cycle that visits each vertex exactly once and has minimum total cost
- Euclidean Traveling Salesman – Vertices are points on the plane and the cost is the Euclidian distance between them – Implies triangle inequality:
- $C(u,v) <= C(u,x) + C(x,v)$
- No polynomial time algorithms known for optimal solution
- 2-Approximate Solution Using MST

Euclidean Traveling Salesman: Steps

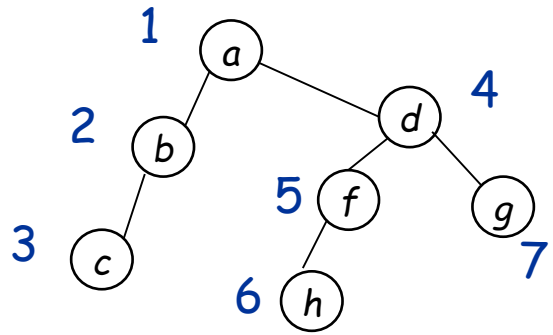
Nodes: Can be thought of as fully connected graph. Edge weight is the Euclidean distance



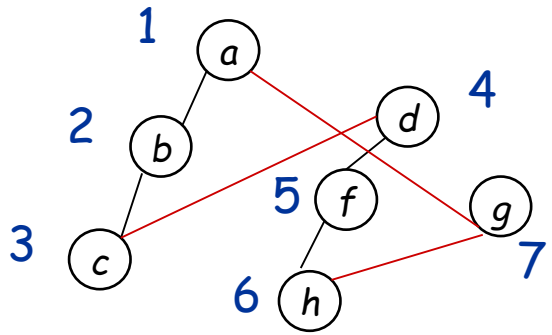
MST: T



DFS on T starting from a



Start from a
Visit nodes in DFS order

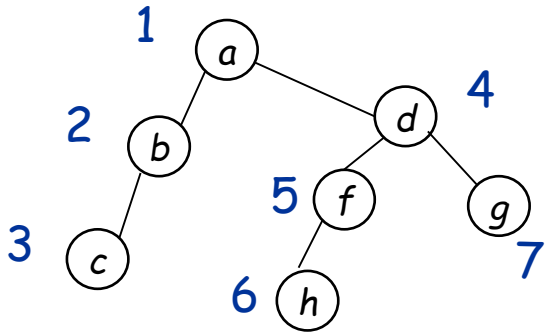


Euclidean Traveling Salesman: Proof of 2-approximation

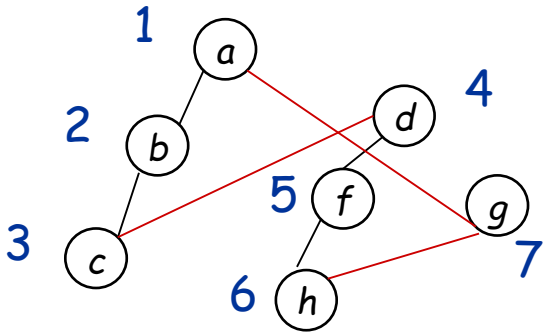
T: MST
W: DFS Walk (visit nodes using MST edges)
H: Computed Tour
H*: Optimal Tour

$C(W)=2C(T)$
$C(H) \le C(W) = 2C(T)$ (triangular inequality)
$C(T) \le C(H^*)$
$C(H) \le 2C(H^*)$

W: DFS Walk



H: Computed Tour



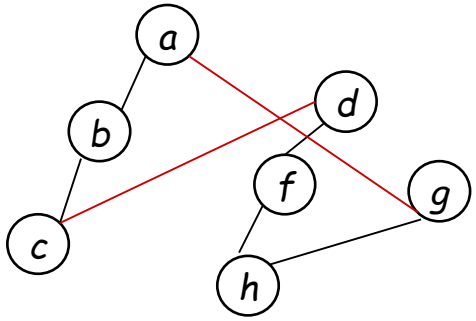
Euclidean Traveling Salesman:
Improving Solution with Local Search

Find an initial tour H

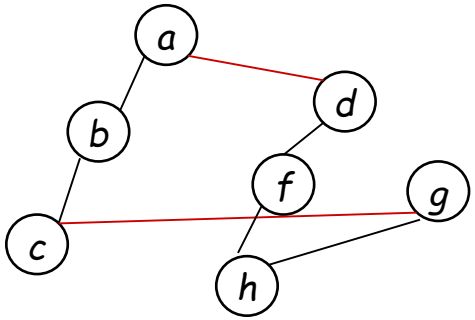
1. For every pair of edges (x,y), (u,v) in H
   if C(x,u) + C(y,v) < C(x,y) + C(u,v) then
   H := H – {(x,y),(u,v)}union {(x,u),(y,v)}
   exit for loop and go to 1

Return H

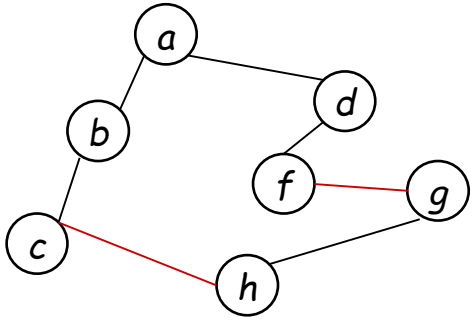
Initial H



After 1 swap



After 2 swaps



After 3 swaps