COMP S265F Lab 11: Minimum Spanning Tree: Kruskal's Algorithm

Dr. Keith Lee

School of Science and Technology

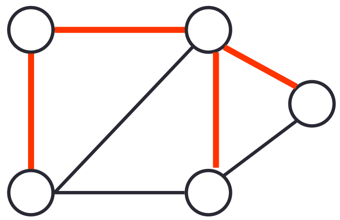
The Open University of Hong Kong

Overview

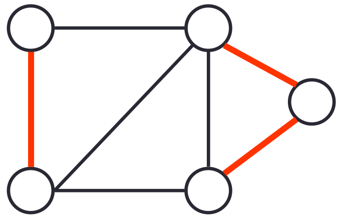
- Spanning tree
  - Some simple facts on a spanning tree
- Minimum (weighted) Spanning Tree
  - Kruskal's algorithm & sample run
  - Proof of correctness: Transformation argument
- Implementing Kruskal's algorithm:
  - FindSet(a vertex), Union(set 1, set 2)
  - Time complexity

Spanning Tree

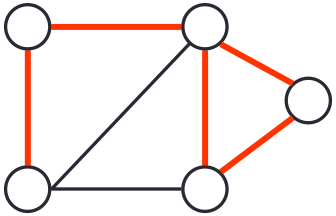
- A *spanning tree* **T** = (V, **E'**) on an ***undirected*** graph **G** = (V, **E**) is a subgraph of G (i.e., **E'** $\subseteq$ **E**) such that
  - for any two vertices **u, v** in V, there is a path in **T** connecting **u, v**; and
  - **T** does not contain any cycle.



A spanning tree



Not connected



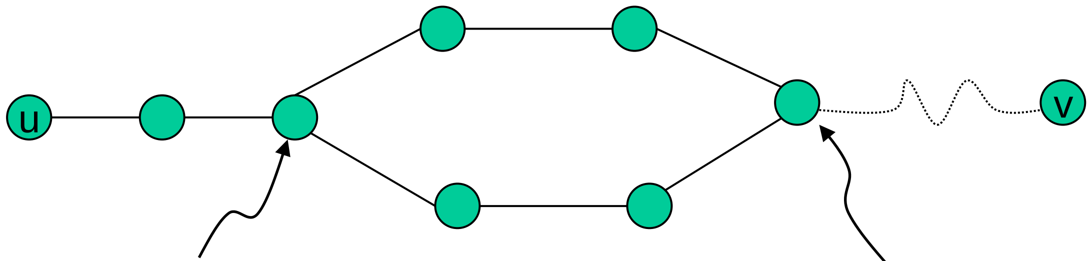
has cycle

Simple facts about a spanning tree T

Fact 1: For any two vertices **u, v** in T, there is a unique path in **T** between **u** and **v**.

Proof:

- By definition, **T** is connected and there is at least one path.
- We can prove that there is only one path by contradiction.
- Suppose there are two paths connecting **u** and **v**.



The first vertex the two paths split. It can be u, but cannot be v (otherwise, the two paths are identical).

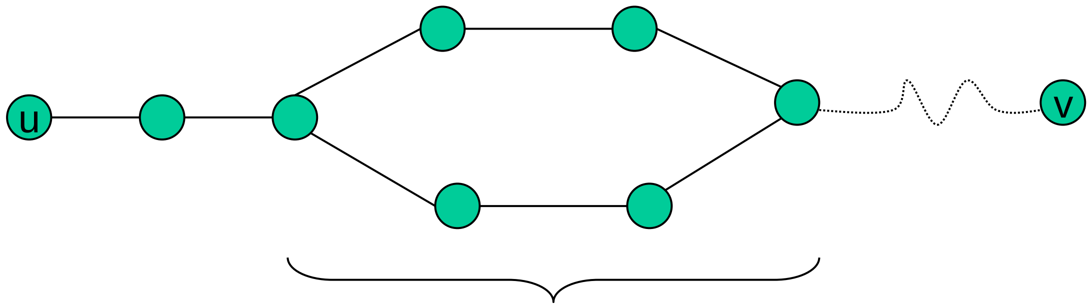
The first vertex the two paths join together. This vertex must exist; the two paths will at least join at v.

Simple facts about a spanning tree T

Fact 1: For any two vertices **u, v** in T, there is a unique path in **T** between **u** and **v**.

Proof:

- By definition, **T** is connected and there is at least one path.
- We can prove that there is only one path by contradiction.
- Suppose there are two paths connecting **u** and **v**.



We find a cycle from these two paths
$\Rightarrow$ A contradiction because **T** does not have cycle.

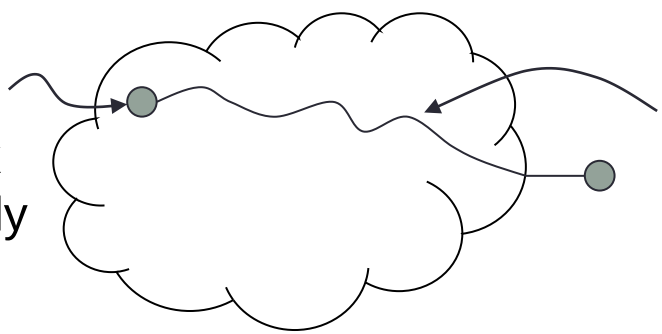
Simple facts about a spanning tree T

Fact 2: Let **n** be the number of vertices, and **m** be the number of edges in **T**. Then, we always have **m** = **n** - 1.

Proof:

- By induction on **n**, the number of vertices.
- Basis step: **n** = 1. A spanning tree with one vertex does not have any edge, i.e., **m** = 0 $\Rightarrow$ **m** = **n** - 1.
- Suppose the fact is true for all trees with *fewer than* **n** vertices.
- Consider a tree **T** with **n** vertices.

Pick
a vertex
arbitrarily



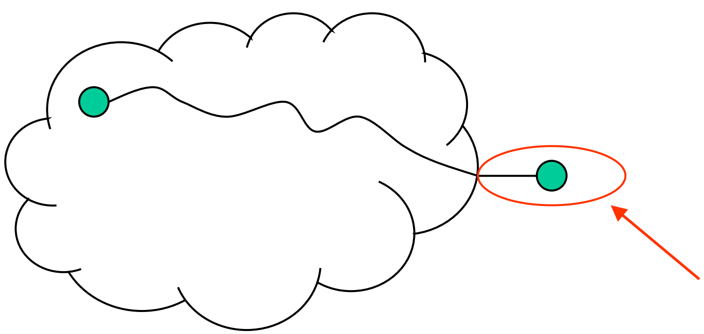
Move forward until hitting some dead end. This must happen eventually because there is no cycle, every step will hit a new vertex, and you can hit at most n-1 new vertices.

Simple facts about a spanning tree T

Fact 2: Let **n** be the number of vertices, and **m** be the number of edges in **T**. Then, we always have **m** = **n** - 1.

Proof:

- By induction on **n**, the number of vertices.
- Basis step: **n** = 1. A spanning tree with one vertex does not have any edge, i.e., **m** = 0 $\Rightarrow$ **m** = **n** - 1.
- Suppose the fact is true for all trees with *fewer than* **n** vertices.
- Consider a tree **T** with **n** vertices.



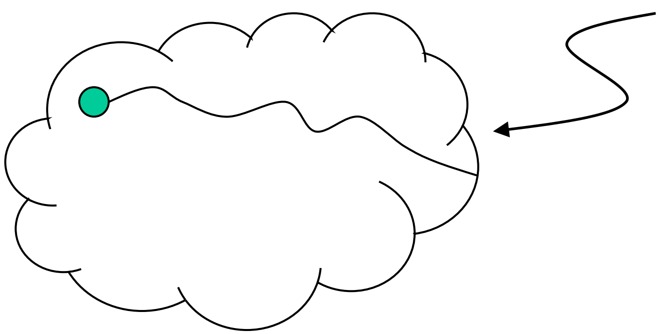
remove this vertex and edge.

Simple facts about a spanning tree T

Fact 2: Let **n** be the number of vertices, and **m** be the number of edges in **T**. Then, we always have **m** = **n** - 1.

Proof:

- By induction on **n**, the number of vertices.
- Basis step: **n** = 1. A spanning tree with one vertex does not have any edge, i.e., **m** = 0 $\Rightarrow$ **m** = **n** - 1.
- Suppose the fact is true for all trees with *fewer than* **n** vertices.
- Consider a tree **T** with **n** vertices.



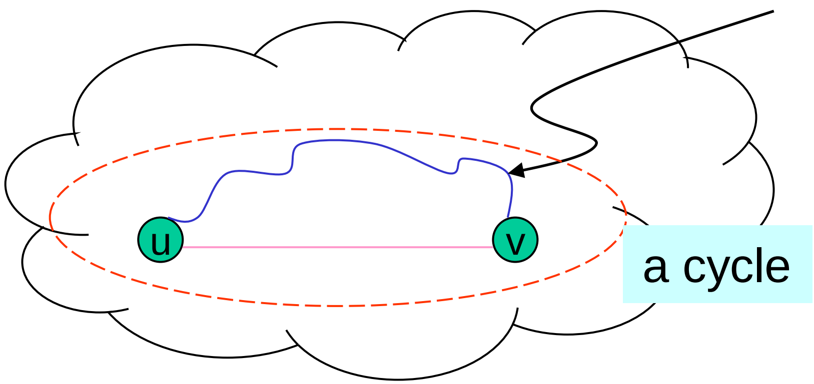
The remain part is
a spanning tree with **n-1** vertices.
By the induction hypothesis,
it has (**n-1**) -1 edges
$\Rightarrow$ The original tree has
(**n-1**) +1 vertices, and
m = (n-1) -1 +1 edges
$\Rightarrow$ i.e., **m** = **n** - 1.

Simple facts about a spanning tree T

Fact 3 (most important): Adding any edge to **T** will create a cycle. And if we remove some edge in this cycle, we will get another tree.

Proof:

- Consider any edge (u,v). Adding (u,v) creates a cycle because



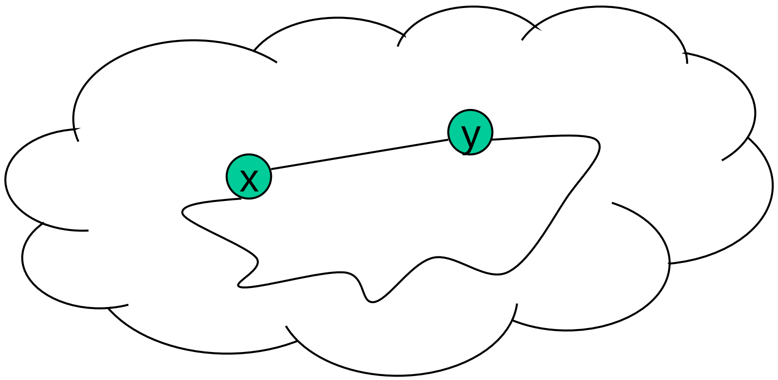
There is a path in **T** connecting u and v.

Simple facts about a spanning tree T

Fact 3 (most important): Adding any edge to **T** will create a cycle. And if we remove some edge in this cycle, we will get another tree.

Proof:

- Deleting any edge (x,y) in the cycle produces another tree.



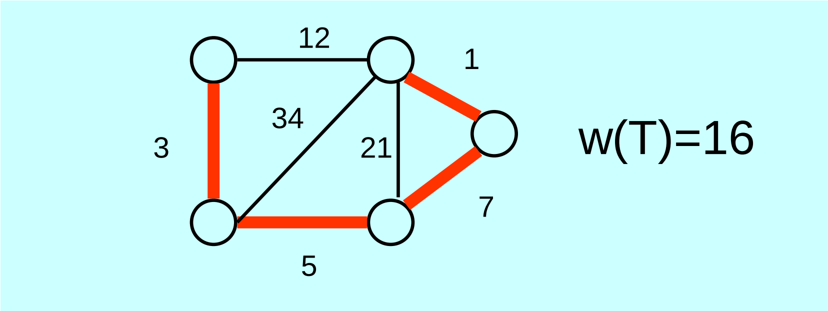
Now, the graph has no cycle, and the graph is still connected $\Rightarrow$ it is a tree.

Minimum (weighted) Spanning Tree

- Let **G** be a general *undirected* graph that is *connected*.
- Suppose that every edge (u, v) of **G** has a weight **w(**u,v**)**.
- Define the weight of a spanning tree **T** of **G**, denoted as **w(T)**, to be the sum of the weight of all edges in **T**, i.e.,

$$w(\boldsymbol{T}) = \sum_{(u,v) \in \boldsymbol{T}} w(u, v)$$

- We say that **T** is a **minimum spanning tree** of **G** if its weight is minimum among all spanning trees of **G**.
- **Example:**



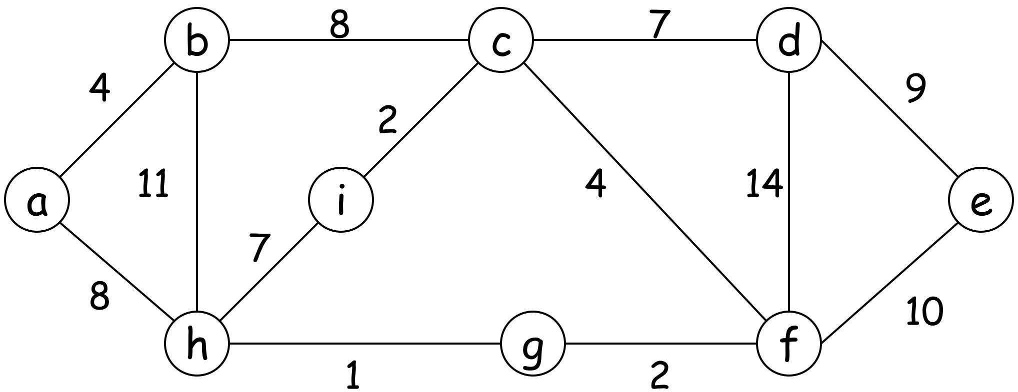
Kruskal's algorithm

- Kruskal has a greedy idea for constructing a spanning tree with minimum weight.

1. Try to construct the tree by adding to the tree one edge at a time, starting from the edge with the smallest weight, and the edge with the next smallest,…, until we finally get the whole spanning tree (i.e., get a tree with **n**-1 edges).
2. When try to add an edge to the solution, we have to make sure that it will not create a cycle.

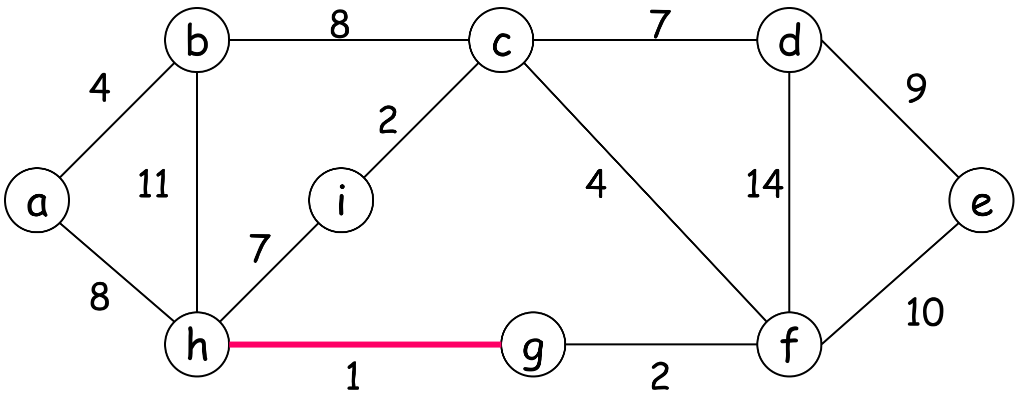
Kruskal's algorithm: Sample run

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



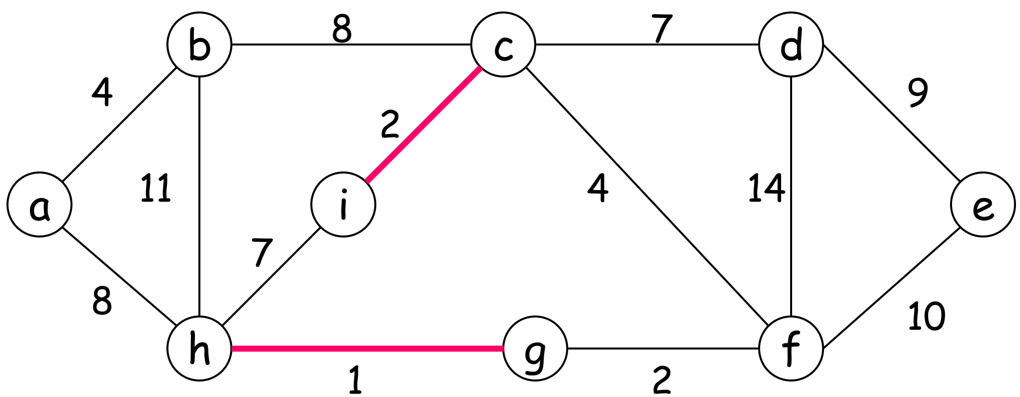
Kruskal’s algorithm: Sample run (Step 1)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



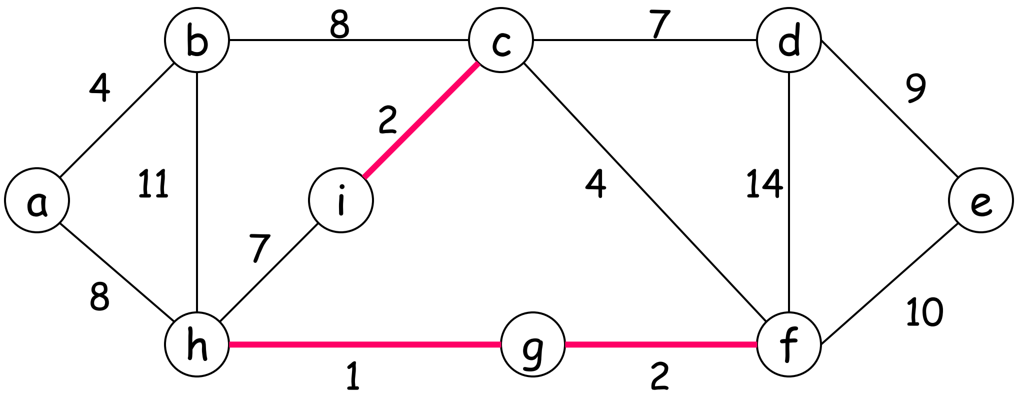
Kruskal's algorithm: Sample run (Step 2)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



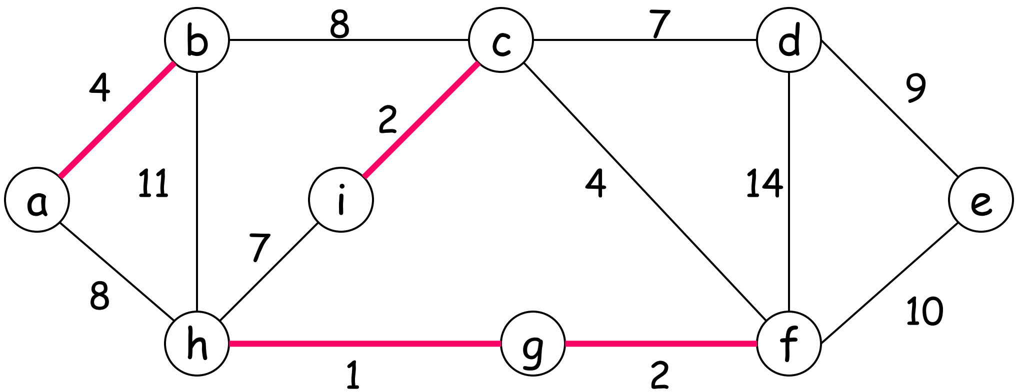
Kruskal’s algorithm: Sample run (Step 3)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



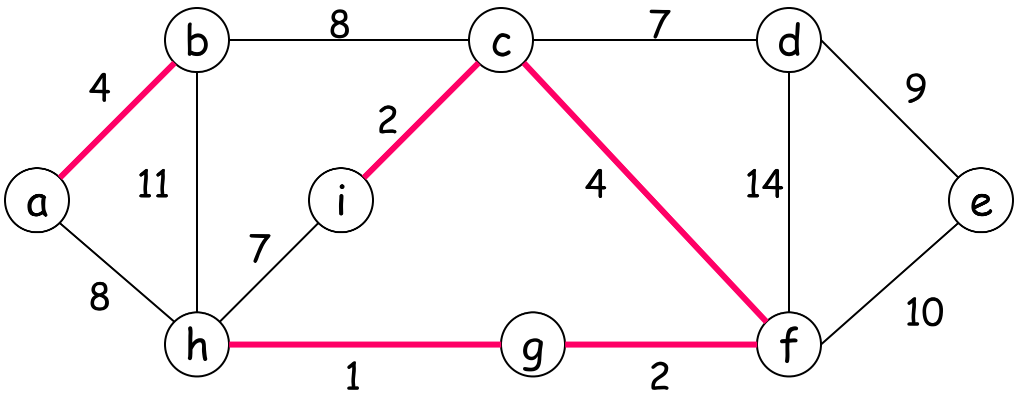
Kruskal's algorithm: Sample run (Step 4)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



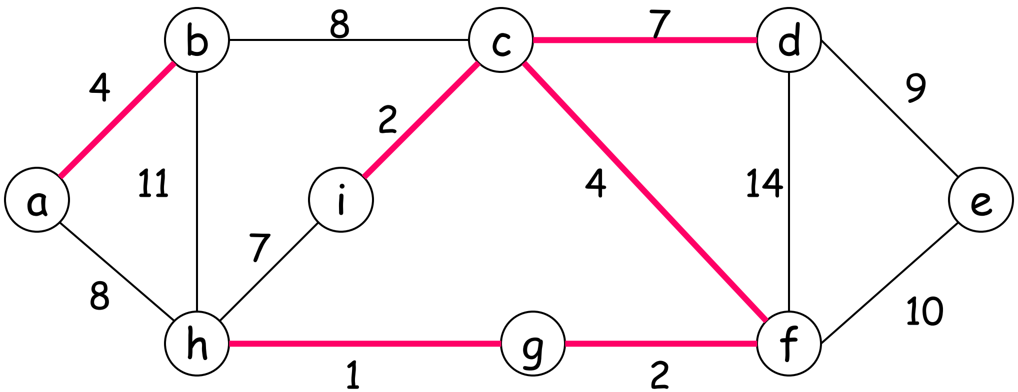
Kruskal's algorithm: Sample run (Step 5)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



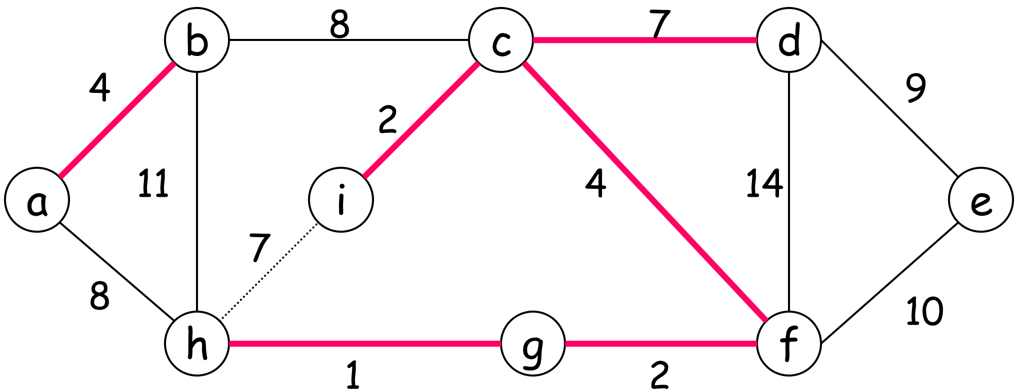
Kruskal's algorithm: Sample run (Step 6)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



Kruskal's algorithm: Sample run (Step 7)

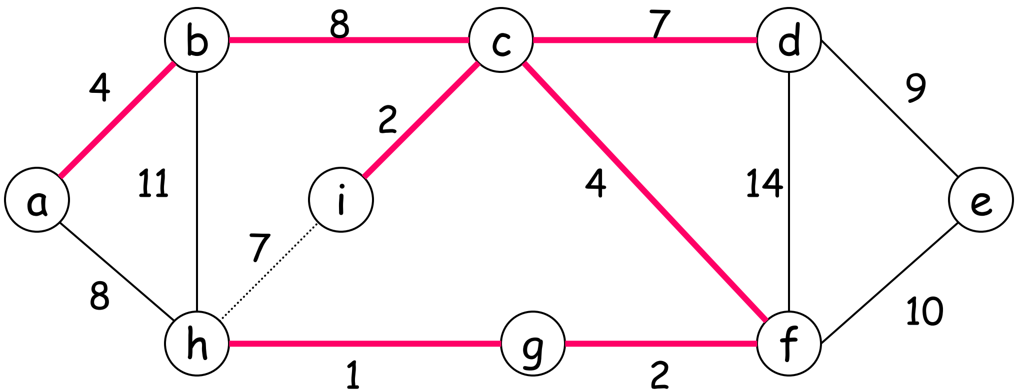
(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



(h,i) cannot be in the solution because it forms a cycle with the previously selected edges.

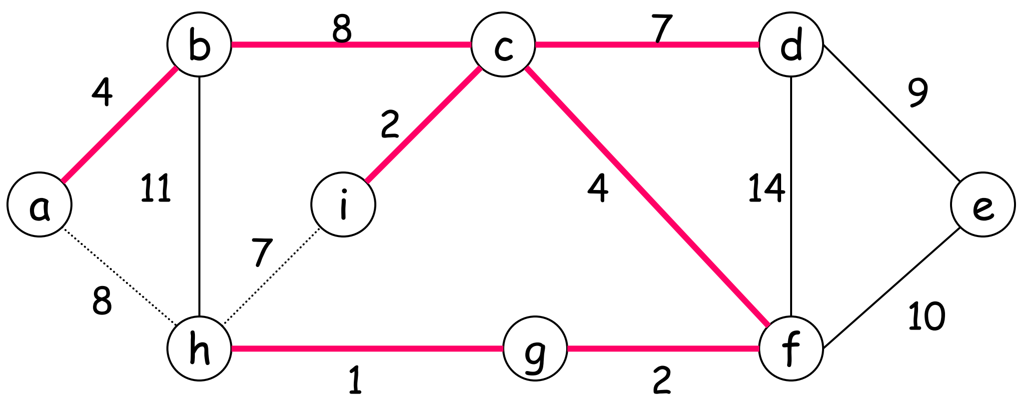
Kruskal’s algorithm: Sample run (Step 8)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



Kruskal's algorithm: Sample run (Step 9)

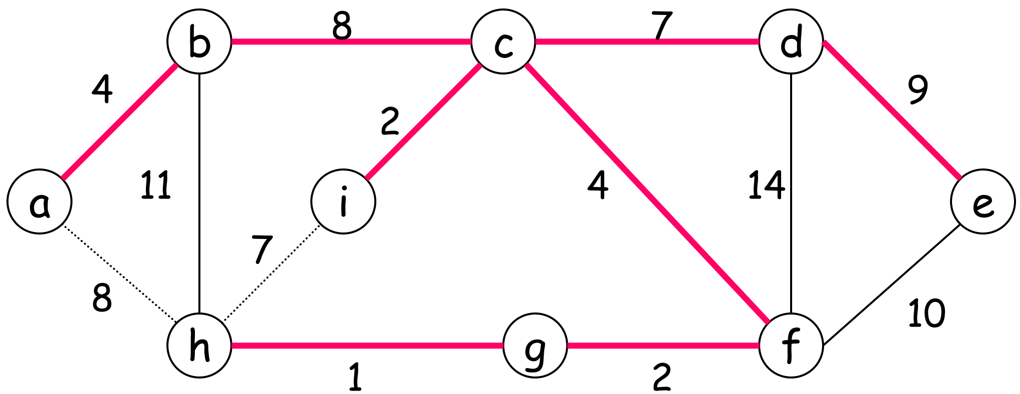
(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



(a,h) cannot be in the solution because it forms a cycle with the previously selected edges.

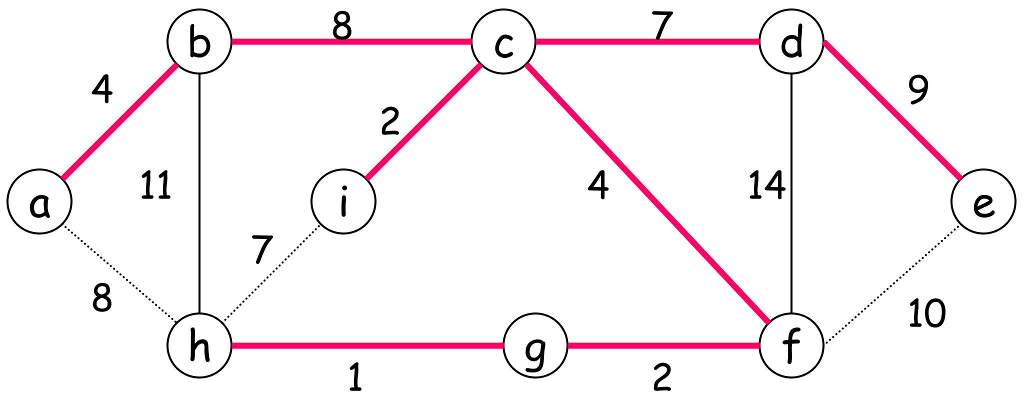
Kruskal's algorithm: Sample run (Step 10)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



Kruskal's algorithm: Sample run (Step 11)

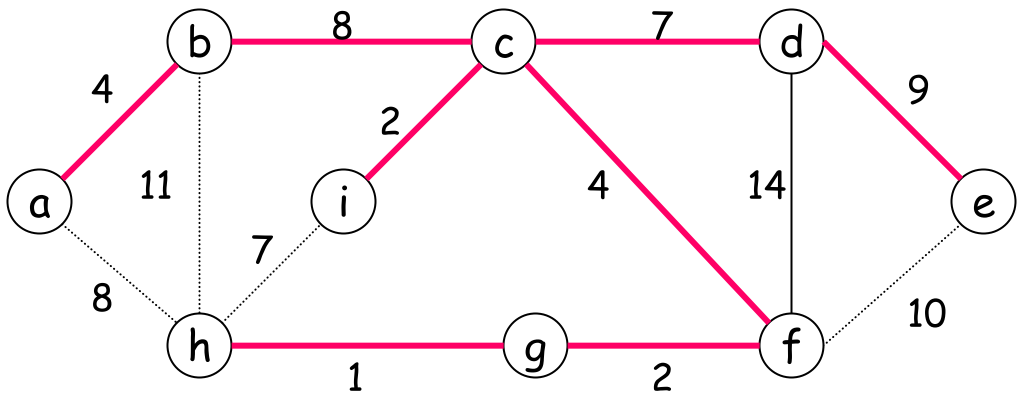
(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
(b,h)	11
(d,f)	14



(f,e) cannot be in the solution because it forms a cycle with the previously selected edges.

Kruskal's algorithm: Sample run (Step 12)

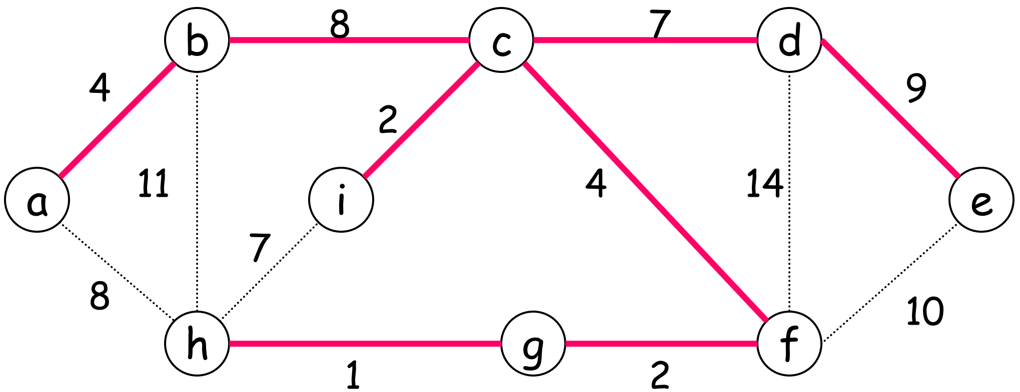
(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
(d,f)	14



(b,h) cannot be in the solution because it forms a cycle with the previously selected edges.

Kruskal's algorithm: Sample run (Step 13)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
~~(d,f)~~	~~14~~



w(T) = 37

Proof of correctness

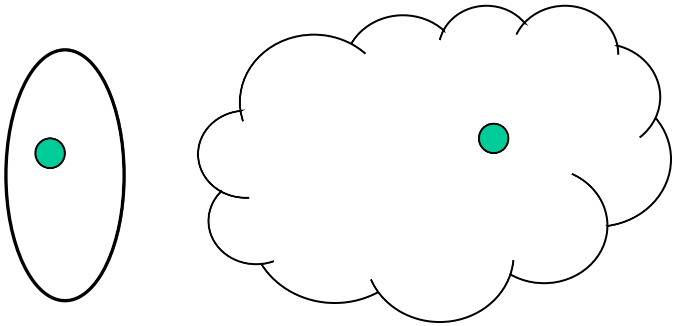
The algorithm always returns a spanning tree because

- the output has no cycle, and
- the output is connected.
  - Why?

Proof of correctness (cont')

The algorithm always returns a spanning tree because

- the output has no cycle, and
- the output is connected.
  - Why? Suppose the output is not connected. That is,

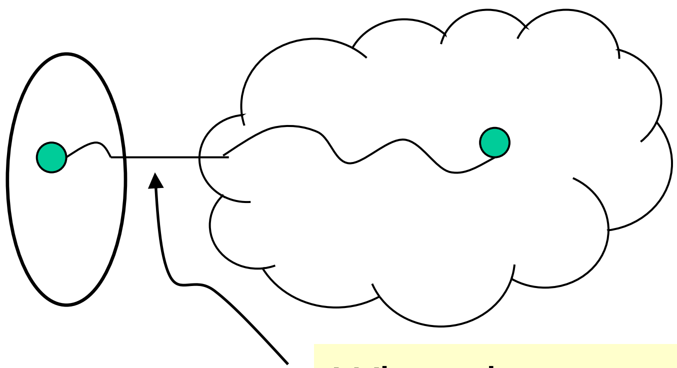


Pick two vertices in two different components

Proof of correctness (cont’)

The algorithm always returns a spanning tree because

- the output has no cycle, and
- the output is connected.
    - Why? Suppose the output is not connected. That is,



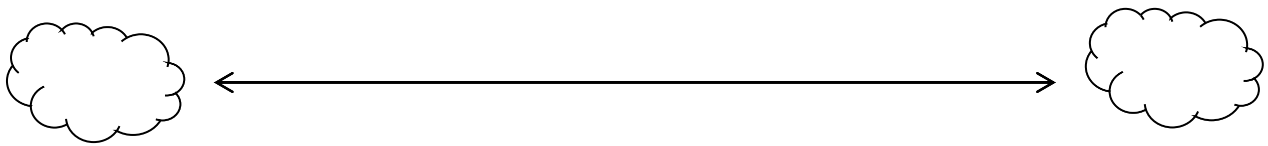
Since the input graph is connected, there must be a path connecting these two vertices.

When the execution checks this edge, it will not delete it as there is no cycle $\Rightarrow$ contradiction

Proof of correctness: Minimum weight

Kruskal’s algorithm always return the minimum spanning tree.

Idea: The framework of the proof is exactly the same as the one we use in proving the optimality of the Huffman code.



$w(T_0) = w(\mathbf{K})?$

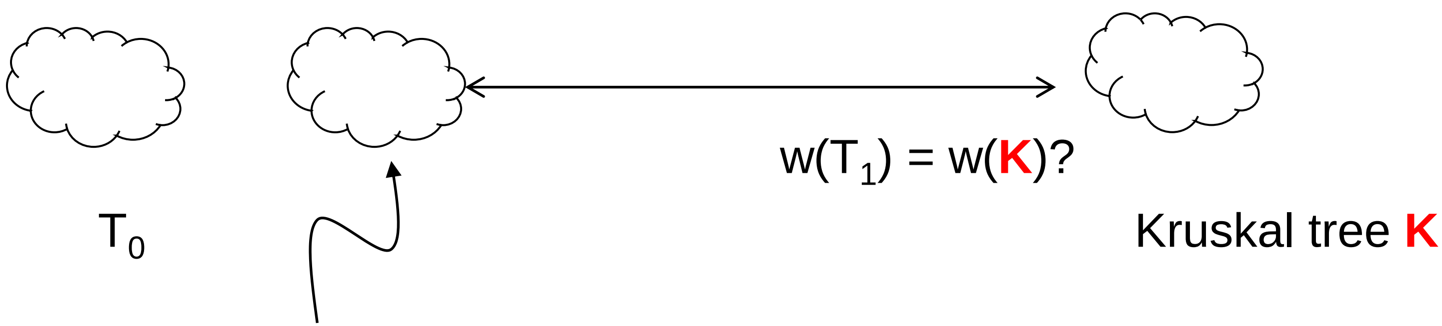
Min.Span.Tree T_0

Kruskal tree **K**

Proof of correctness: Minimum weight

Kruskal's algorithm always return the minimum spanning tree.

Idea: The framework of the proof is exactly the same as the one we use in proving the optimality of the Huffman code.

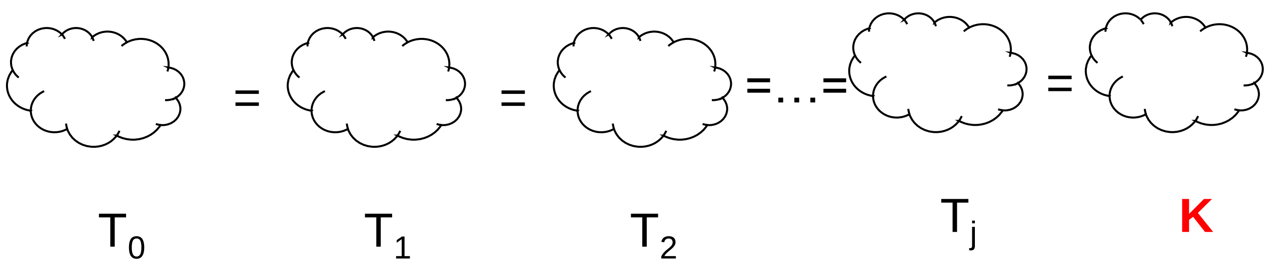


we construct a
T_1 such that
$w(T_0) = w(T_1)$ and
T_1 is "more similar" to **K**

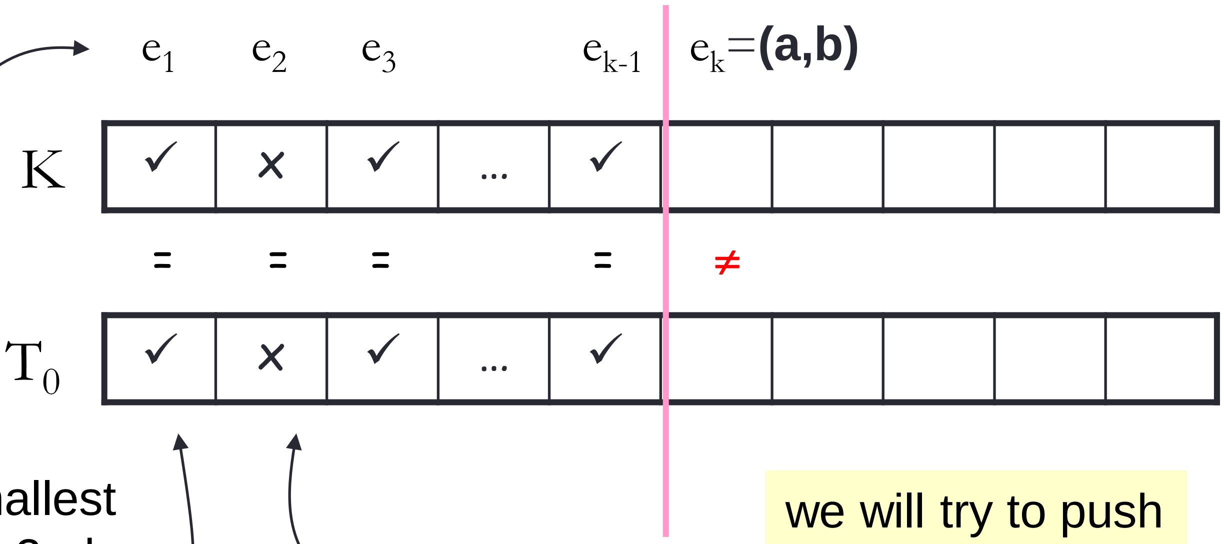
Proof of correctness: Minimum weight

Kruskal's algorithm always return the minimum spanning tree.

Idea: The framework of the proof is exactly the same as the one we use in proving the optimality of the Huffman code.



How to construct T_1?



edges listed in ascending order of weight, i.e., e_1 has the smallest weight, e_2 has the 2nd smallest weight, ..

we will try to push the pink line right.

Both trees contain the edge

Both trees not contain the edge

How to construct T_1? (cont')

						(a,b)				
K	✓	×	✓	…	✓	×				
T_0	✓	×	✓	…	✓	✓				

This case is not possible. Adding **(a,b)** will not form any cycle; otherwise the optimal solution is not a tree.

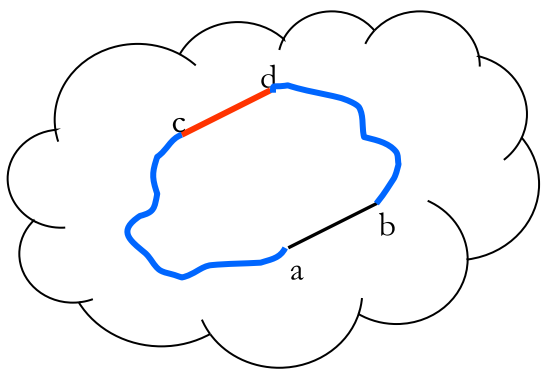
This implies Kruskal will also choose **(a,b)**.

How to construct T_1? (cont’)

						(a,b)				
K	✓	✗	✓	...	✓	✓				
T_0	✓	✗	✓	...	✓	✗				

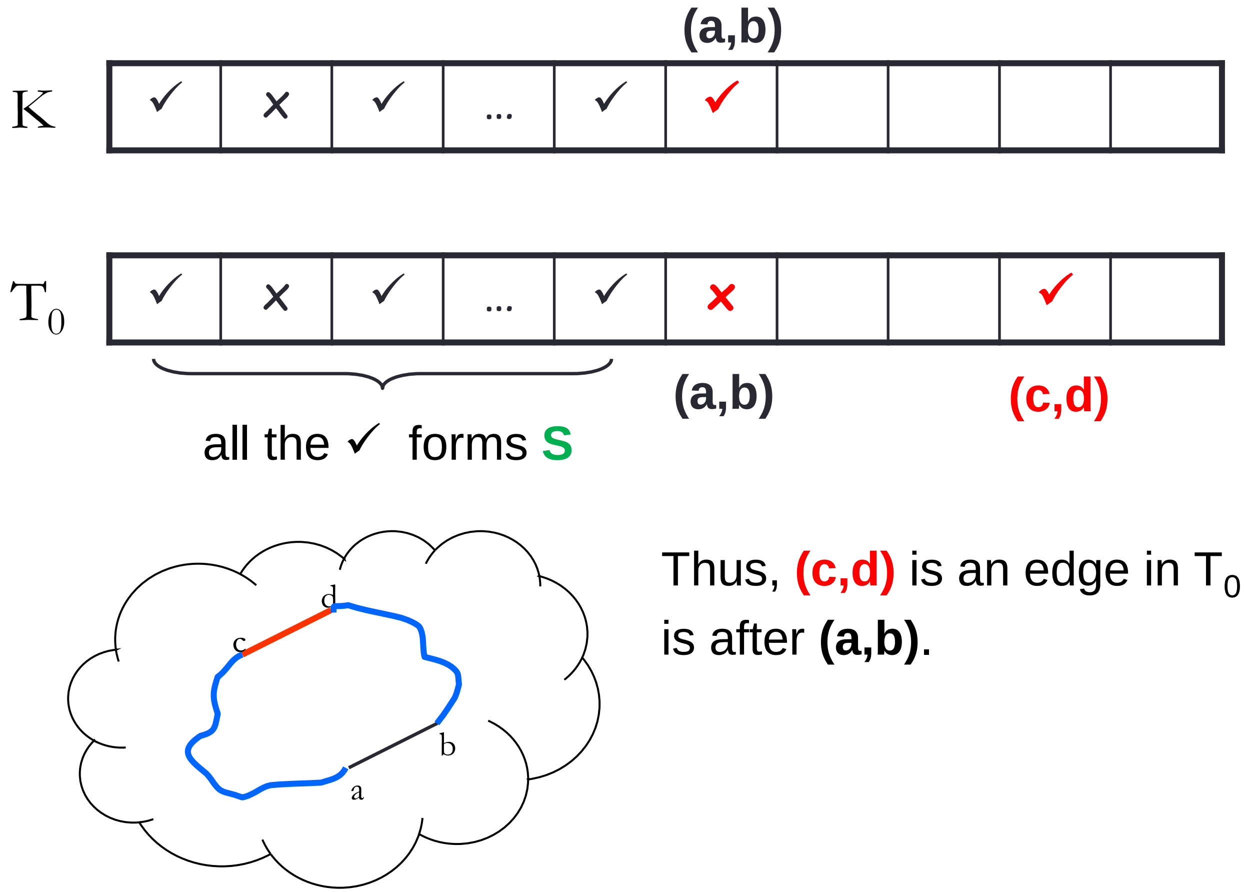
all the ✓ forms **S**

(a,b)



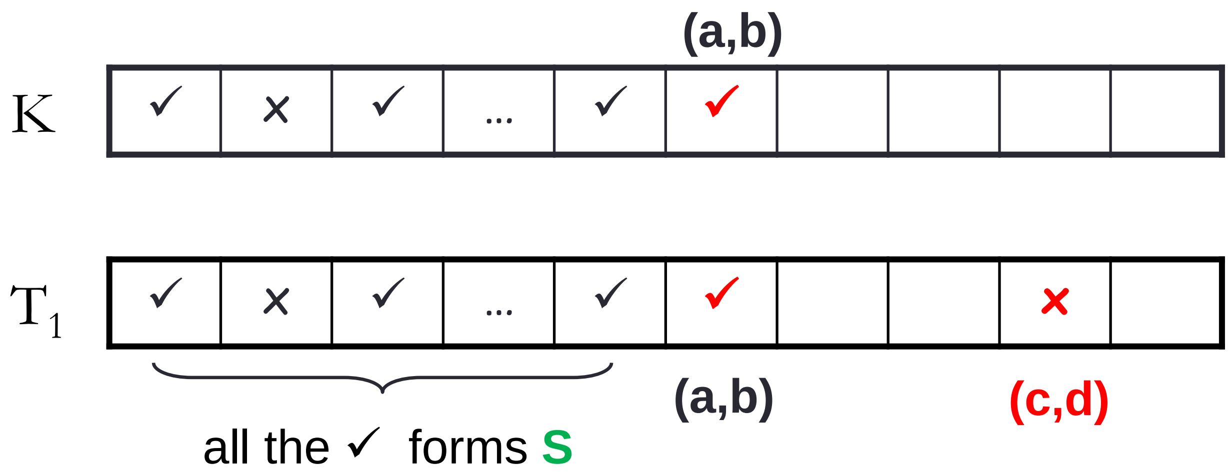
add the edge **(a,b)** to T_o, and we form a cycle, which must contain some edge **(c,d)** not in **S**∪**(a,b)** (otherwise, Kruskal will not choose **(a,b)**).

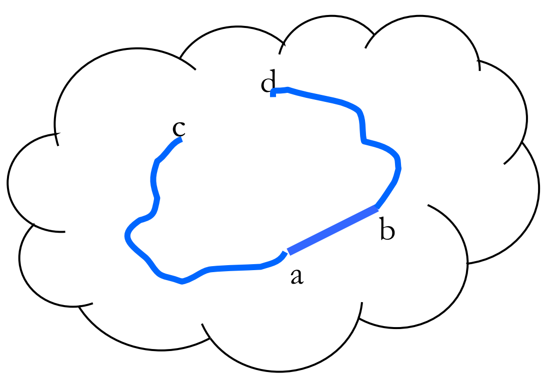
How to construct T_1? (cont')



Thus, **(c,d)** is an edge in $T_{0,}$ and is after **(a,b)**.

How to construct T_1? (cont')



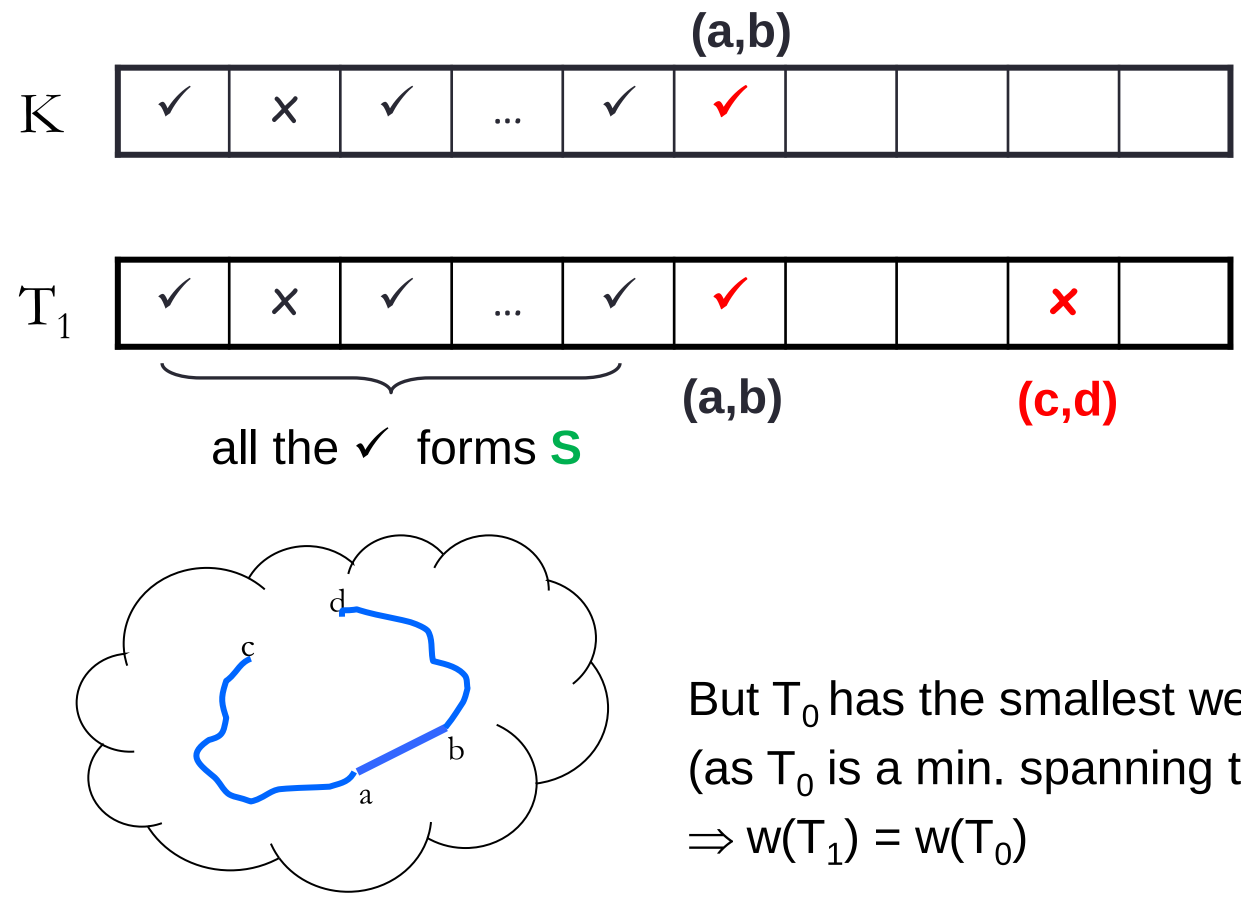


Delete **(c,d)** and add **(a,b)** forms another tree, which is $\mathbf{T_1}$

Since edges are listed in ascending order, w(a,b) ≤ w(c,d)

➔ $w(T_1)$ is no greater than $w(T_0)$

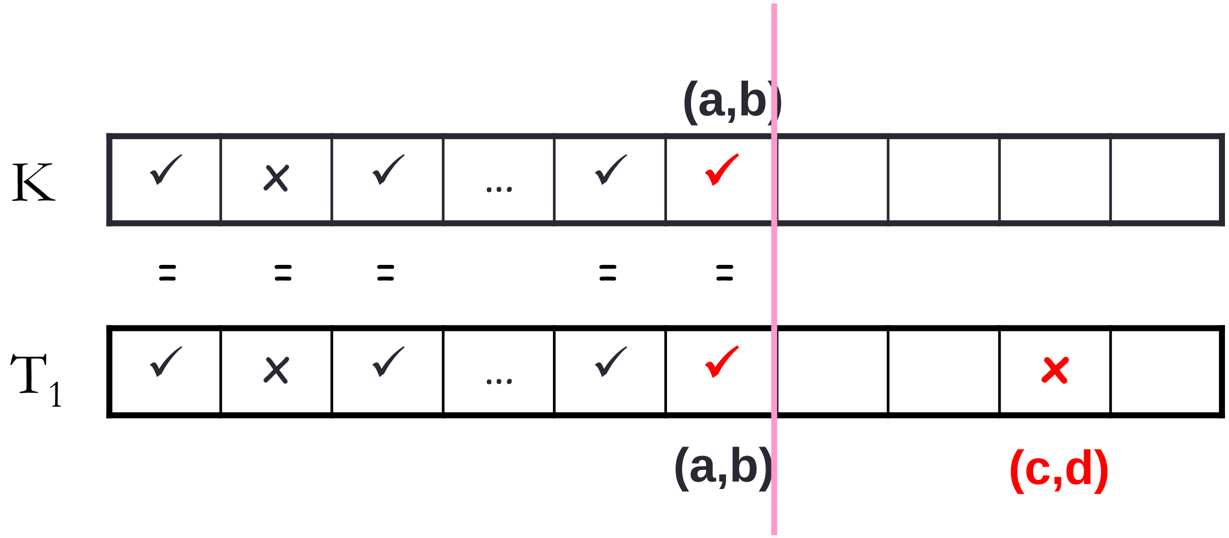
How to construct T_1? (cont')

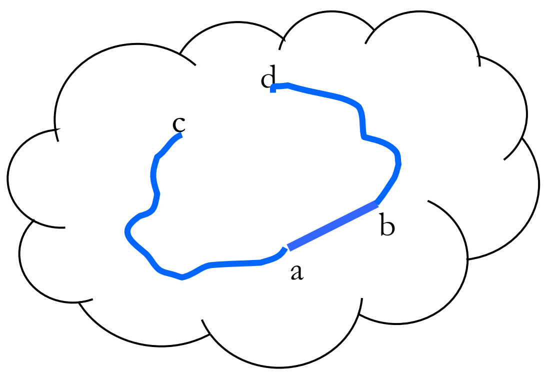


But T_0 has the smallest weight (as T_0 is a min. spanning tree)

$\Rightarrow w(T_1) = w(T_0)$

How to construct T_1? (cont’)

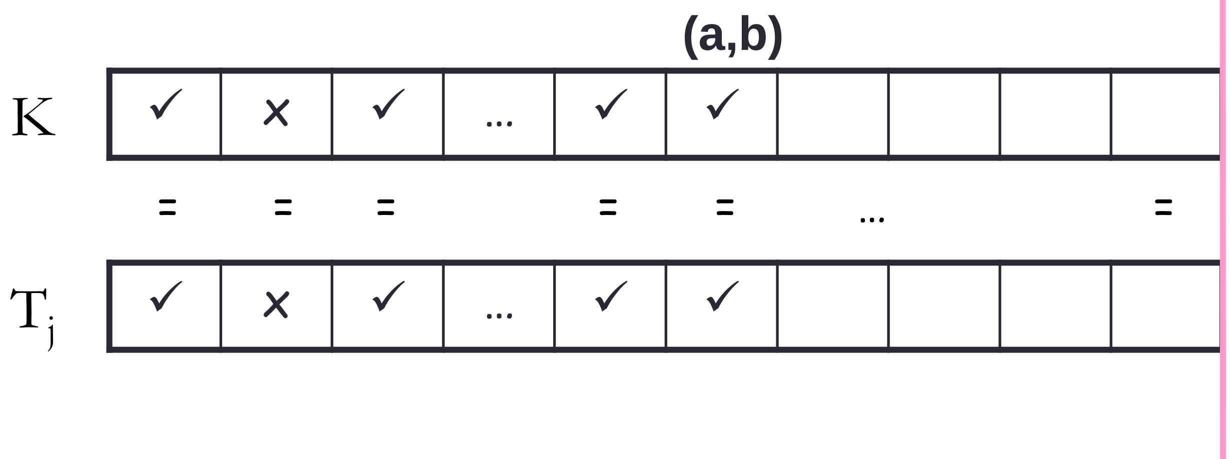




has one more edge
same as K.

Proof of correctness: Minimum weight

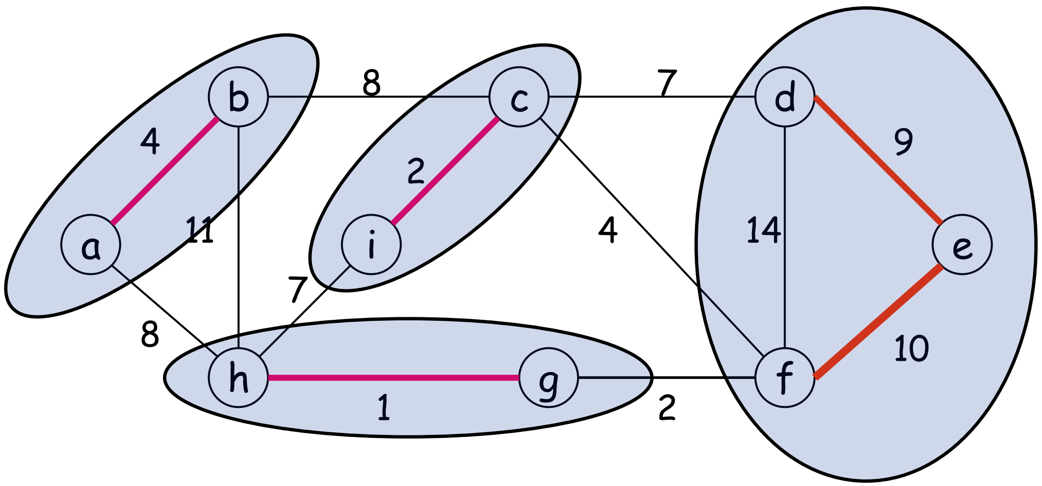
- Repeat the process, we will eventually push the pink line to the rightmost end.



Thus, **K** = T_j,
and we have $w(T_0) = w(T_j) = w($**K**$)$.
This follows that **K** has also the minimum weight; it is a minimum spanning tree (MST).

Implementing Kruskal's algorithm

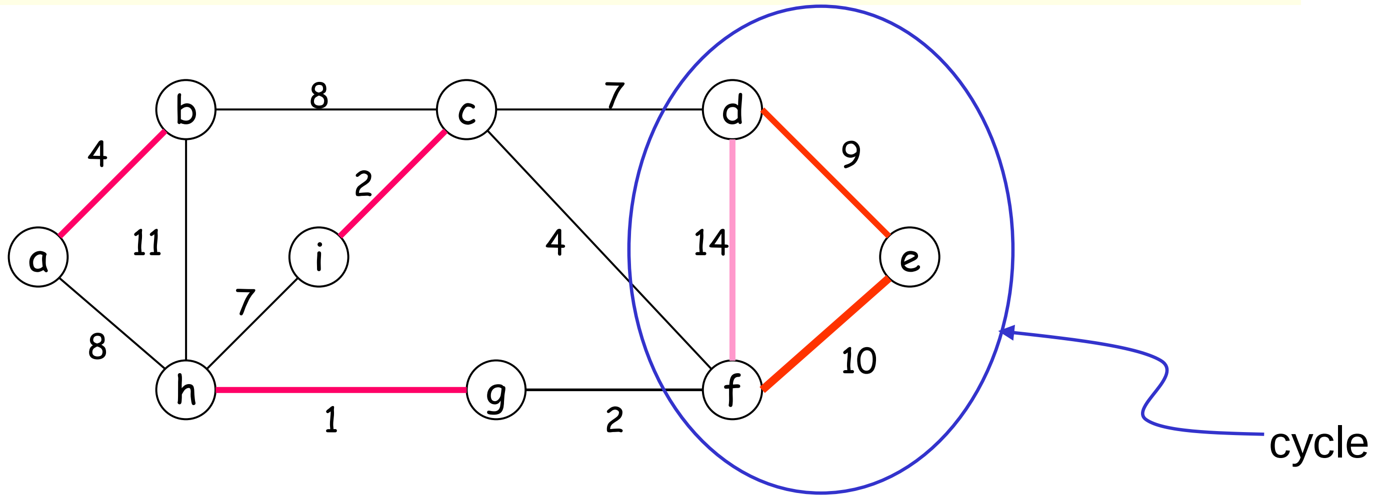
- How to determine whether adding an edge will cause a cycle?
- **Observation:** During the execution of the algorithm, the set of edges in the solution (red edges) forms a set of disjoint trees.



Four "subtrees"

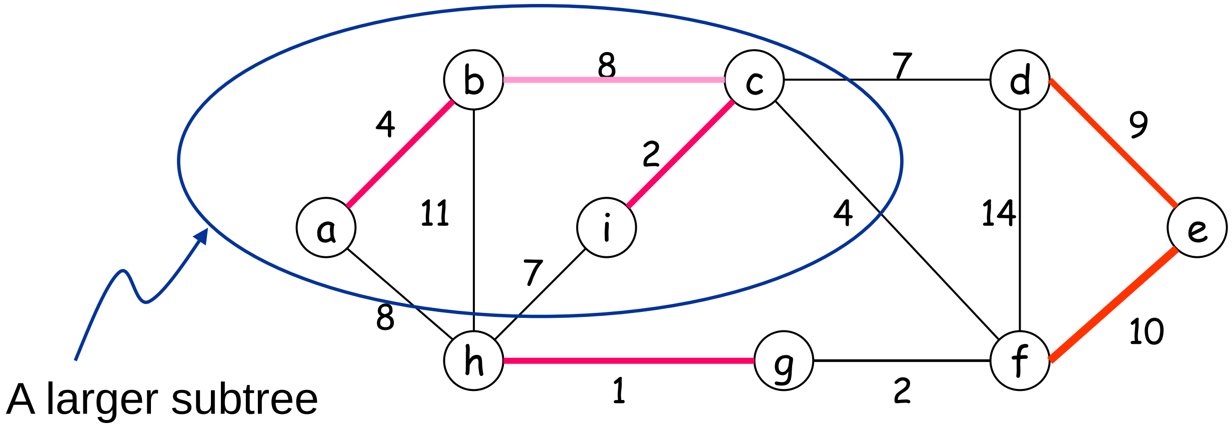
Implementing Kruskal's algorithm (cont')

- **Observation:** During the execution of the algorithm, the set of edges in the solution (red edges) forms a set of disjoint trees.
- **Case 1:** Adding (u,v) where u and v are in the same subtree
  $\Rightarrow$ cycle and thus not add to solution.



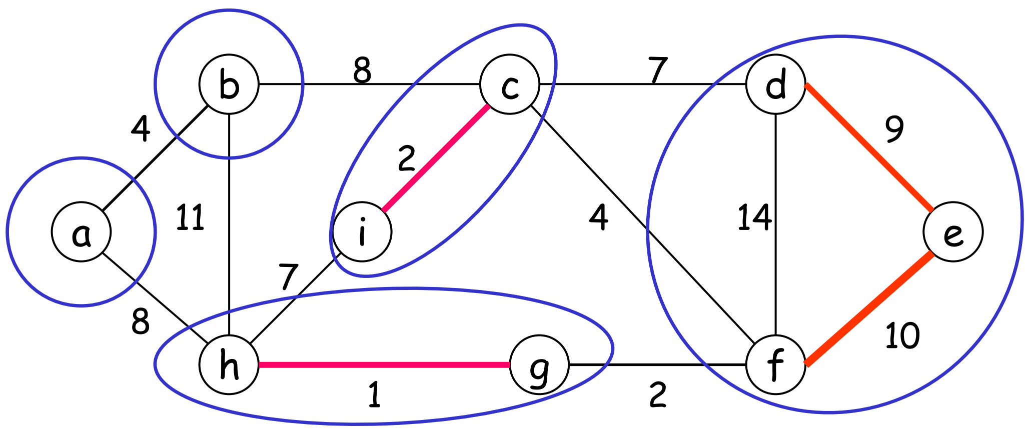
Implementing Kruskal's algorithm (cont')

- **Observation:** During the execution of the algorithm, the set of edges in the solution (red edges) forms a set of disjoint trees.
- **Case 2:** Adding (u,v) where u and v are in the different subtrees
  $\Rightarrow$ add to solution to form a larger subtree



Implementing Kruskal's algorithm (cont')

- **Idea:** Remember the sets of vertices of the subtrees. (We agree that isolated vertex is also a subtree.)



The "forest" (i.e., set of subtrees):

{a}, {b}, {i,c}, {h,g}, {d,e,f}

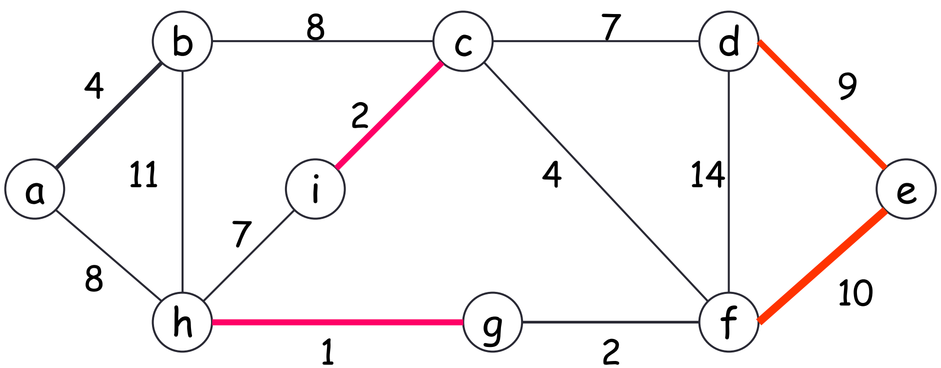
Find-Set(vertex)

- **Case 1: Adding edge (d,f).** Since **Find-Set**(d) **=** **Find-Set**(f),

  ⇒ d, f are in the same set and hence are in the same tree

  ⇒ cycle and don't add (d,f) to the current solution set.

return the set which the argument belongs in the current forest.



The "forest" (i.e., set of subtrees):

{a}, {b}, {i,c}, {h,g}, {d,e,f}

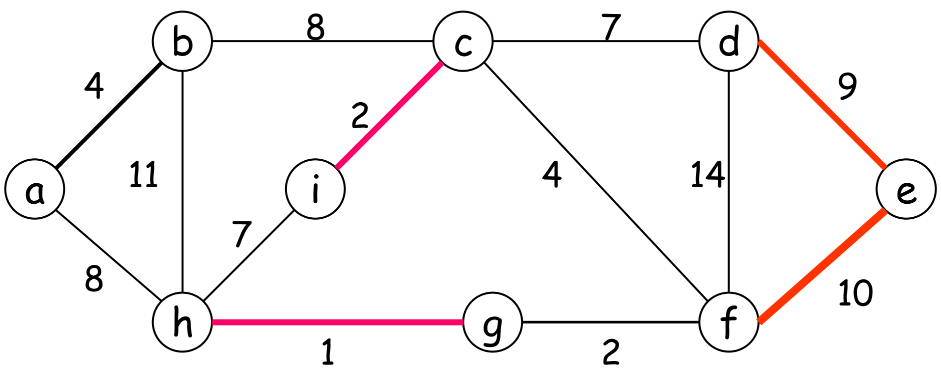
Union(set 1, set 2)

- **Case 2: Adding edge (i,h).** Since **Find-Set**(i) **≠** **Find-Set**(h),

  ⇒ i, h are in different sets, and hence in different trees.

  ⇒ no cycle and add (i,h) to the solution set, and

  ⇒ “merge” the two subtrees by **Union**(Find-Set(i), Find-Set(h)).



The “forest” (i.e., set of subtrees):

{a}, {b}, {i,c}, {h,g}, {d,e,f}

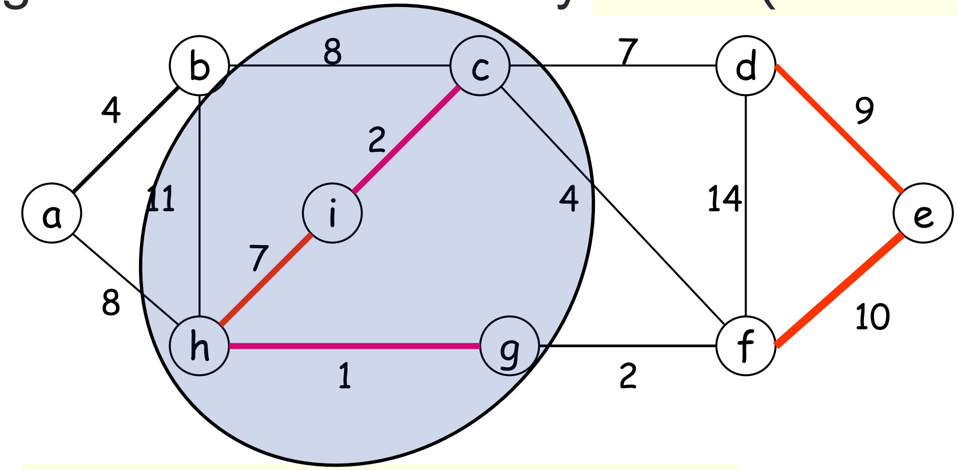
Union(set 1, set 2) (cont')

- **Case 2: Adding edge (i,h).** Since **Find-Set**(i) **≠** **Find-Set**(h),

  ⇒ i, h are in different sets, and hence in different trees.

  ⇒ no cycle and add (i,h) to the solution set, and

  ⇒ "merge" the two subtrees by **Union**(Find-Set(i), Find-Set(h)).



The "forest" (i.e., set of subtrees):

{a}, {b}, {i,c,h,g}, {d,e,f}

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

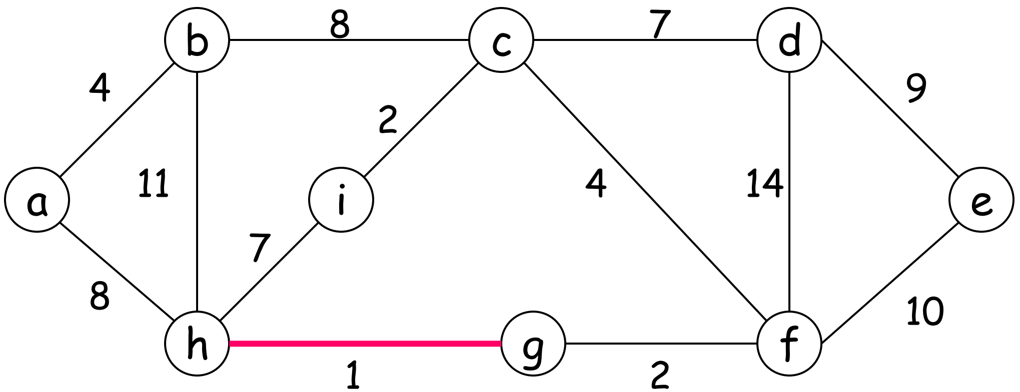
- No. of Make-Set = |V|, No. of Find-Set = 2 |E|, No. of Union = |V| - 1
- *Union-Find Disjoint Sets* take O(V log V) time for all unions.
- Sorting of edges has a time complexity of O(E log E)

$\Rightarrow$ **Total time complexity** = O(E log **E** + 2E + **V** log V)

= O(E log **V²** + **E** log V) [a tree has |V| - 1 edges, so |E| ≥ |V| - 1]

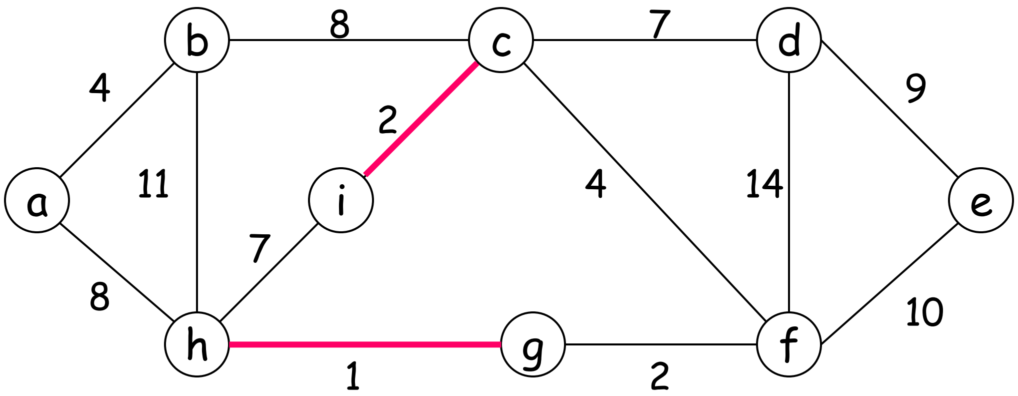
= O(**2**E log **V** + **E** log V) = **O(E log V)**.

Kruskal's algorithm: Sample run (Step 1)



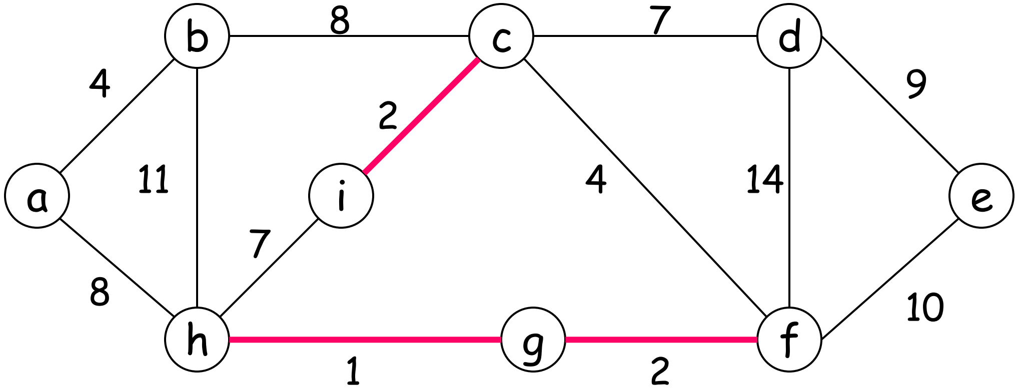
{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

Kruskal's algorithm: Sample run (Step 2)



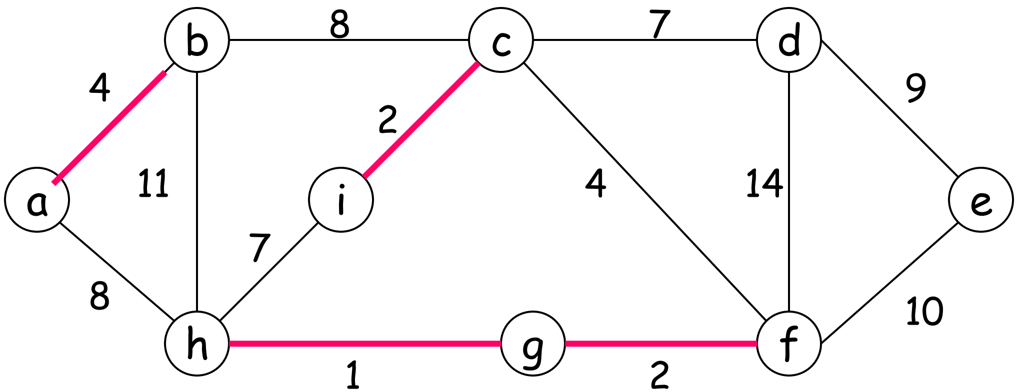
{a}, {b}, {c}, {d}, {e}, {f}, {g,h}, {i}

Kruskal’s algorithm: Sample run (Step 3)



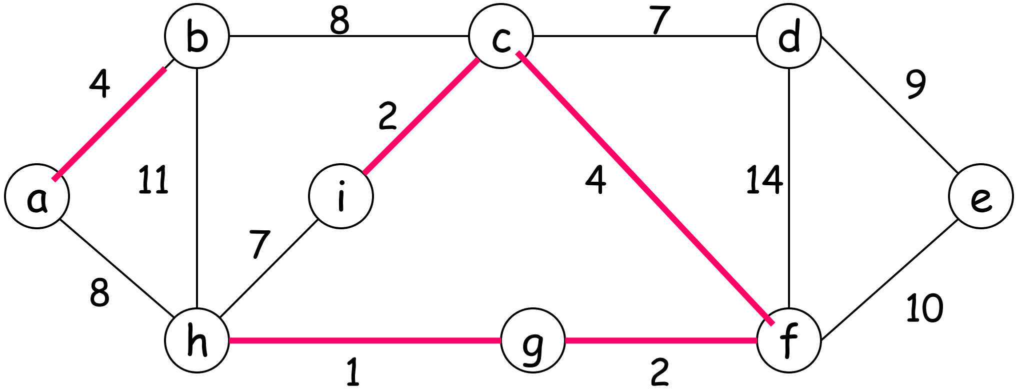
{a}, {b}, {c,i}, {d}, {e}, {f}, {h,g}

Kruskal's algorithm: Sample run (Step 4)



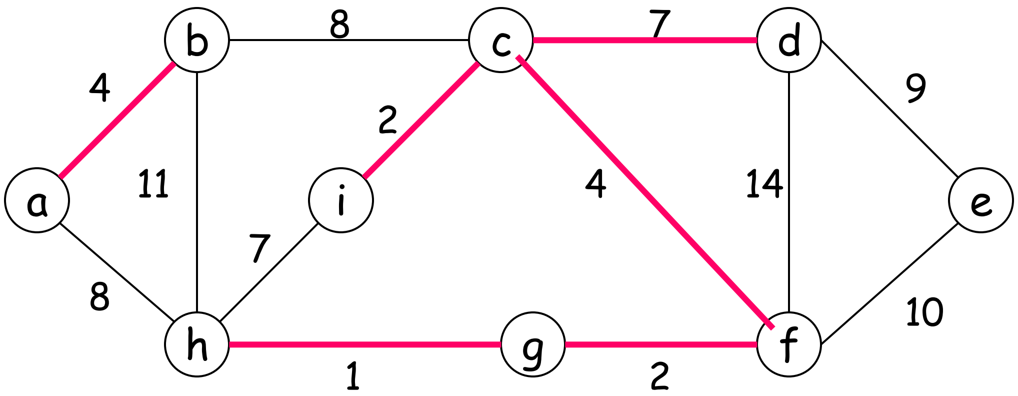
{a}, {b}, {c,i}, {d}, {e}, {f,h,g}

Kruskal’s algorithm: Sample run (Step 5)



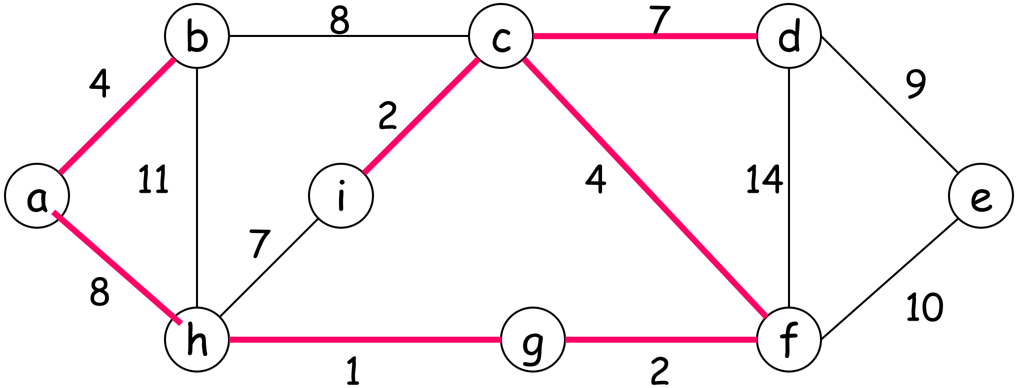
{a,b}, {c,i}, {d}, {e}, {f,h,g}

Kruskal's algorithm: Sample run (Step 6)



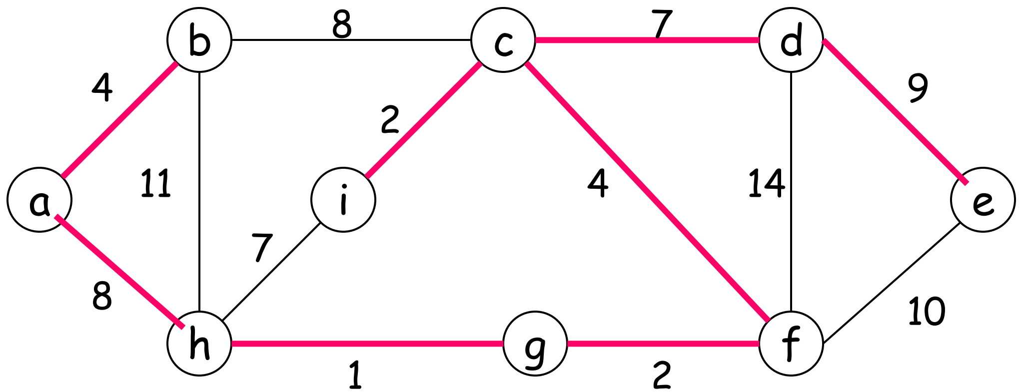
{a,b}, {c,i,f,h,g}, {d}, {e}

Kruskal's algorithm: Sample run (Step 7)



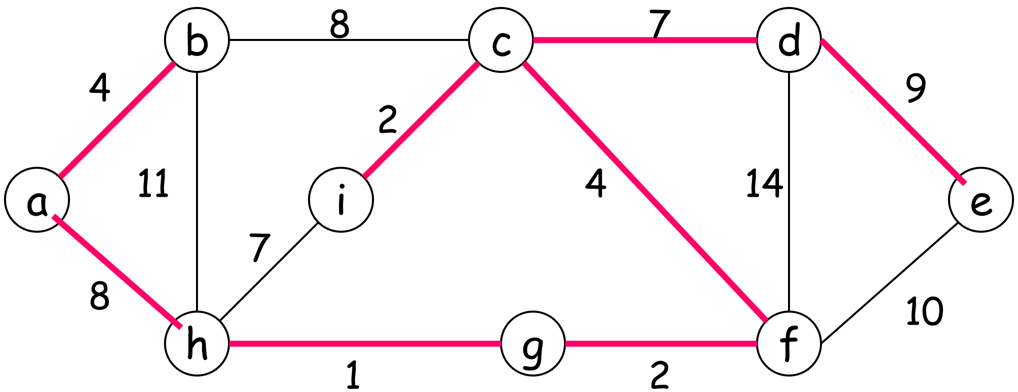
{a,b}, {c,i,f,h,g,d}, {e}

Kruskal’s algorithm: Sample run (Step 8)



{a,b,c,i,f,h,g,d}, {e}

Kruskal's algorithm: Sample run (Step 9)



{a,b,c,i,f,h,g,d,e}