Lecture 9 – Graph Algorithms

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Minimum Spanning Tree
  - Kruskal Algorithm
  - Disjoint Set
  - Prime Algorithm
- Topological Sort
- Graph Search

Minimum Spanning Tree

Minimum Spanning Trees

Spanning tree: Given a connected undirected graph, a **spanning tree** is a subset of the edges with that connects all the vertices together without any cycles

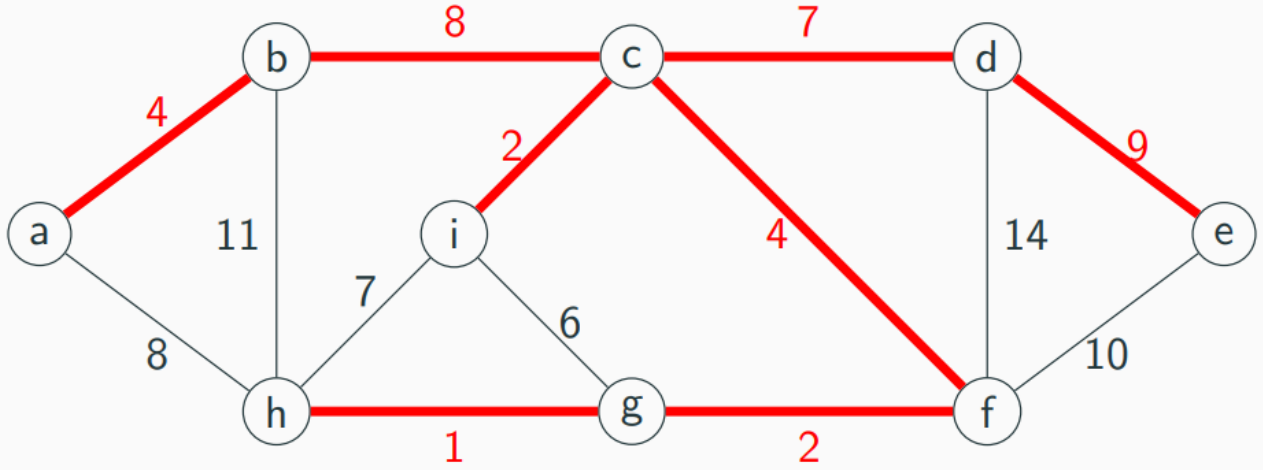
Minimum spanning tree (**MST**): a spanning tree with the minimum possible total edge weight

These edges always form a tree

- Start from any node, regard each node it links to as the child, an so on

Famous algorithms:

- Kruskal's algorithm
- Prim's algorithm



Minimum Spanning Trees

(Properties) For a connected graph with n vertices:

- Any spanning tree has exactly $n - 1$ edges
  - Needs at least $n - 1$ edges to connect n nodes
  - If there are n or more edges, there forms a cycle
- Any $n - 1$ edges without forming a cycle form a spanning tree
  - These $n - 1$ edges must connect all the n nodes because otherwise there is a cycle

An idea to solve the problem: select $n - 1$ edges without forming a cycle

Example Problem

Consider a country with n cities, and the government plans to build roads to connect them. Given a number of roads you can choose to connect the cities, each has a cost associated with it, representing the expense of construction. The goal is to connect all cities with the minimum total cost.

City 1	City 2	Cost
A	B	4
A	C	2
B	C	5
B	D	10
C	D	3

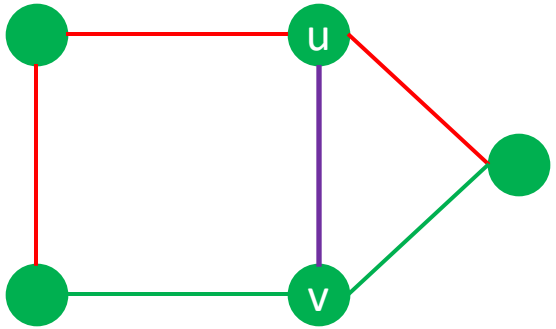
Optimal Substructure

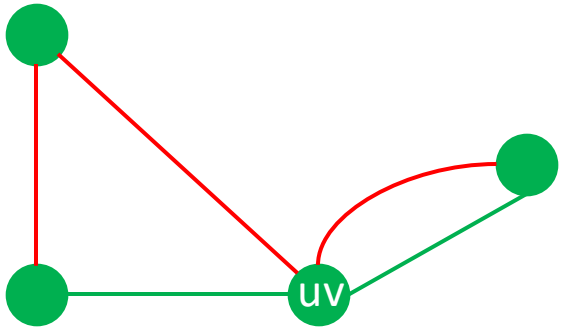
Given an MST $C =< c_1, c_2, \ldots, c_{n-1} >$ for graph $G =< V, E >$ with n nodes. For any edge $e_{c_i} =< u, v >$ contained in it, $C - \{c_i\}$ is an MST for $G' = g(G, < u, v >) =< V', E' >$, where

$$V' = V - \{u, v\} + uv,$$

$$E' = \{< uv \; if \; u' = u \; else \; u', \; uv \; if \; v' = v \; else \; v' > \mid < u', v' > \in E\}$$

where uv is a new node by merging u and v.





Recurrence for Optimized Value

Given an MST $C =< c_1, c_2, \dots, c_{n-1} >$ for graph $G =< V, E >$ with n nodes. For any edge $e_{c_i} =< u, v >$ contained in it, $C - \{c_i\}$ is an MST for $G' = g(G, < u, v >) =< V', E' >$, where

$$V' = V - \{u, v\} + uv,$$

$$E' = \{< uv \; if \; u' = u \; else \; u', \; uv \; if \; v' = v \; else \; v' > \mid < u', v' > \in E\}$$

where uv is a new node by merging u and v.

- Recurrence for optimized value: $f(G) = \min_{e \in E}(f(g(G, < e.u, e.v >)) + e.w)$
- Large time complexity!
- Intuition: Greedily select the edge that has the smallest weight
- Is it optimal (Greedy choice property)?

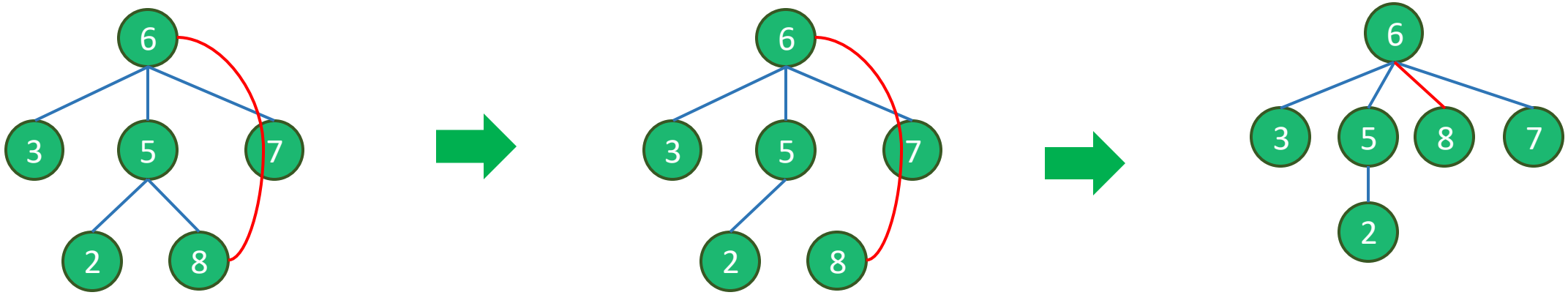
—— prove the edge with smallest weight is contained in MST

Greedy Choice Property

There is an MST(G) that contains the edge with lowest weight e_x

Proof: Given any MST C that does not contain x, we can construct an MST C'' containing x

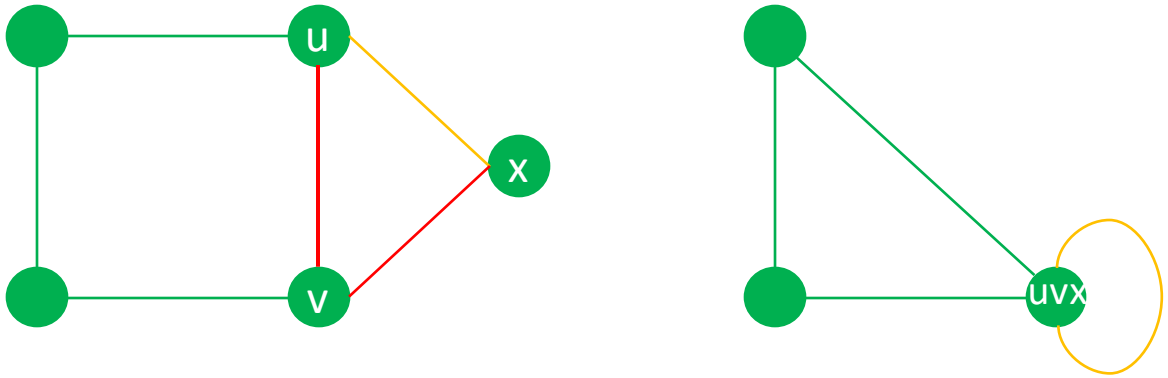
- Add x and build $C' =< c_1, c_2, \dots c_{n-1}, x >$, C' has 1 and only 1 cycle $C_l =< c_{k_1}, c_{k_2}, \dots c_{k_l} >$
- Delete $c_u = \arg \max_{u^* \in \{k_1, \dots, k_l\}} e_{c_{u^*}}.w$, we have $C'' = C - \{c_u\} + \{x\}$ is a spanning tree.
- Since $e_x.w$ is the smallest, $e_x.w \leq c_u.w$. Thus $|C''| \leq |C|$, with C being MST, C'' is also MST



Kruskal's Algorithm

Substitution on merging the nodes: Greedily select the edge that has the smallest weight and does not form a cycle

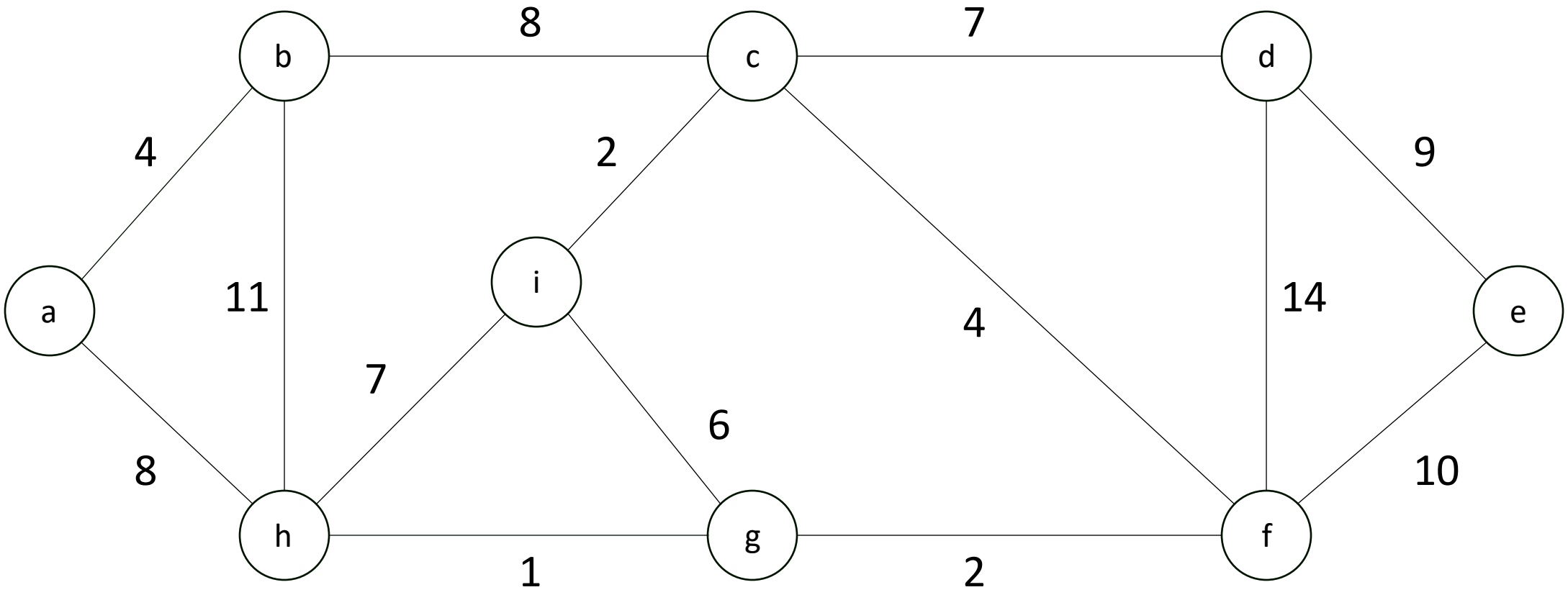
- Edges that can form a cycle can be regarded as self-loop in the emerged graph, which are eliminated after merging



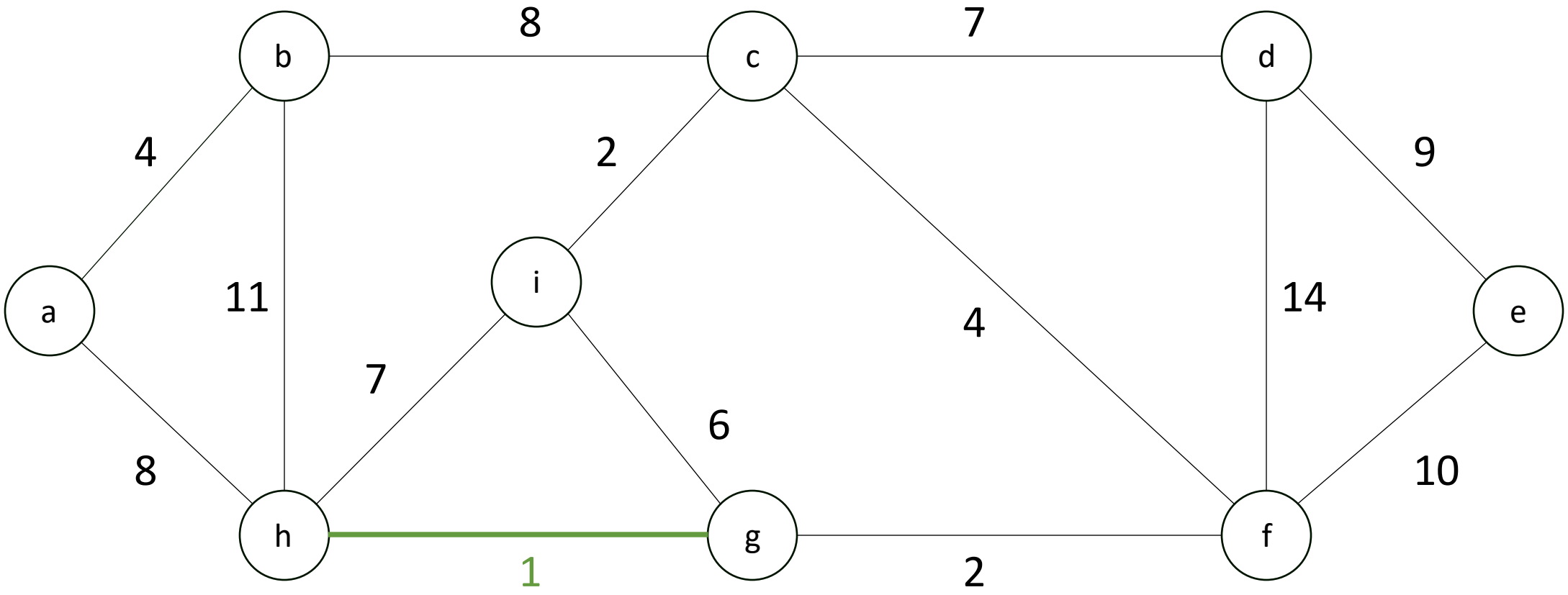
Kruskal's Algorithm Example

try

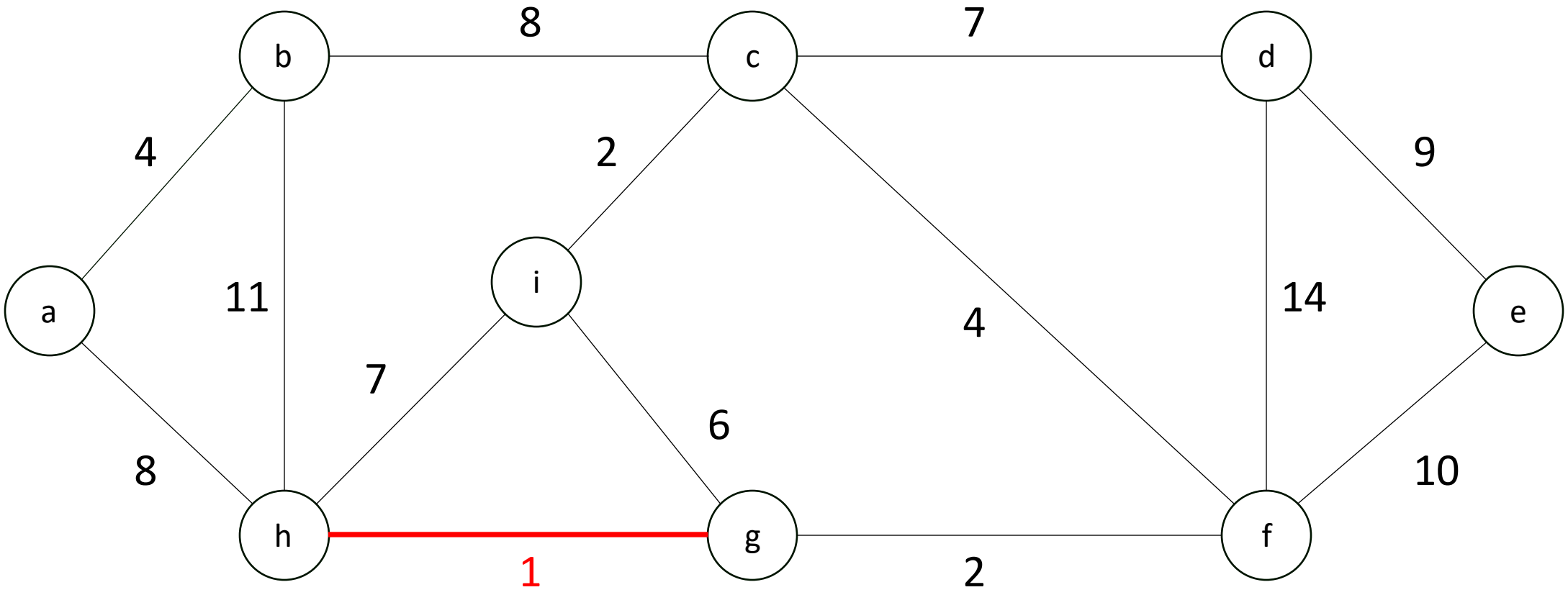
verify



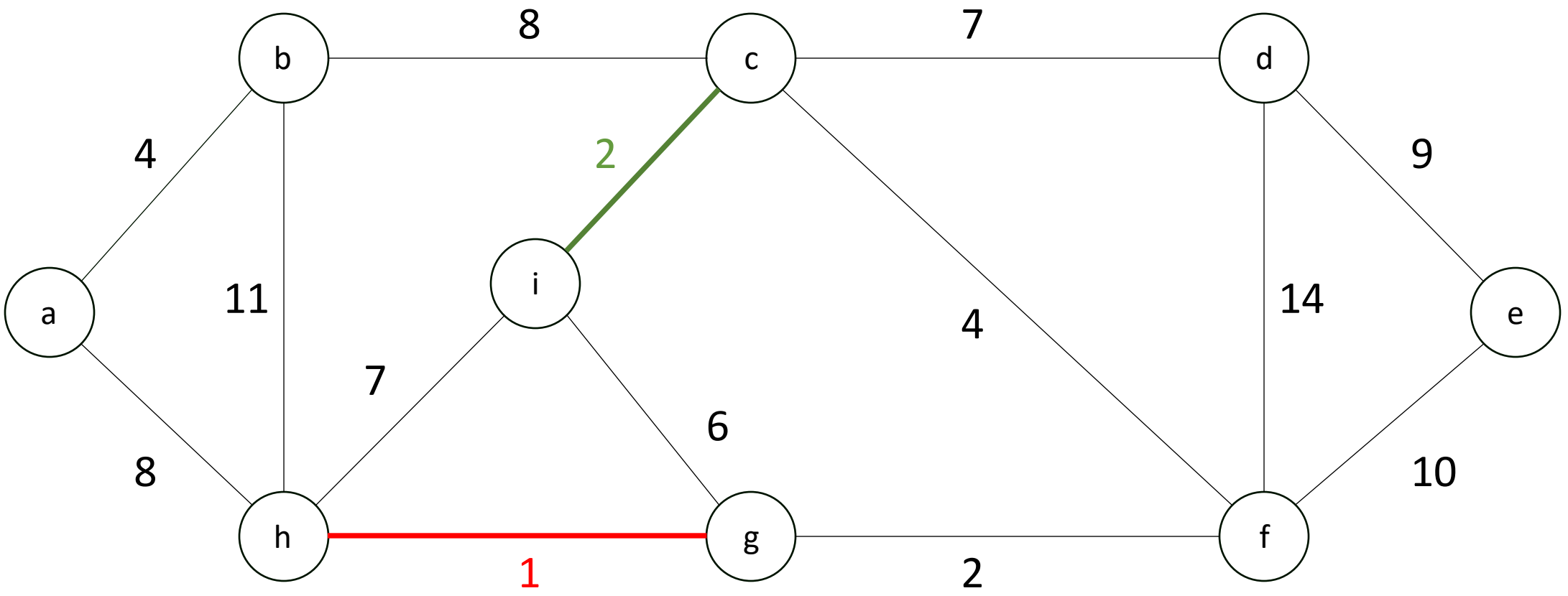
Kruskal's Algorithm Example



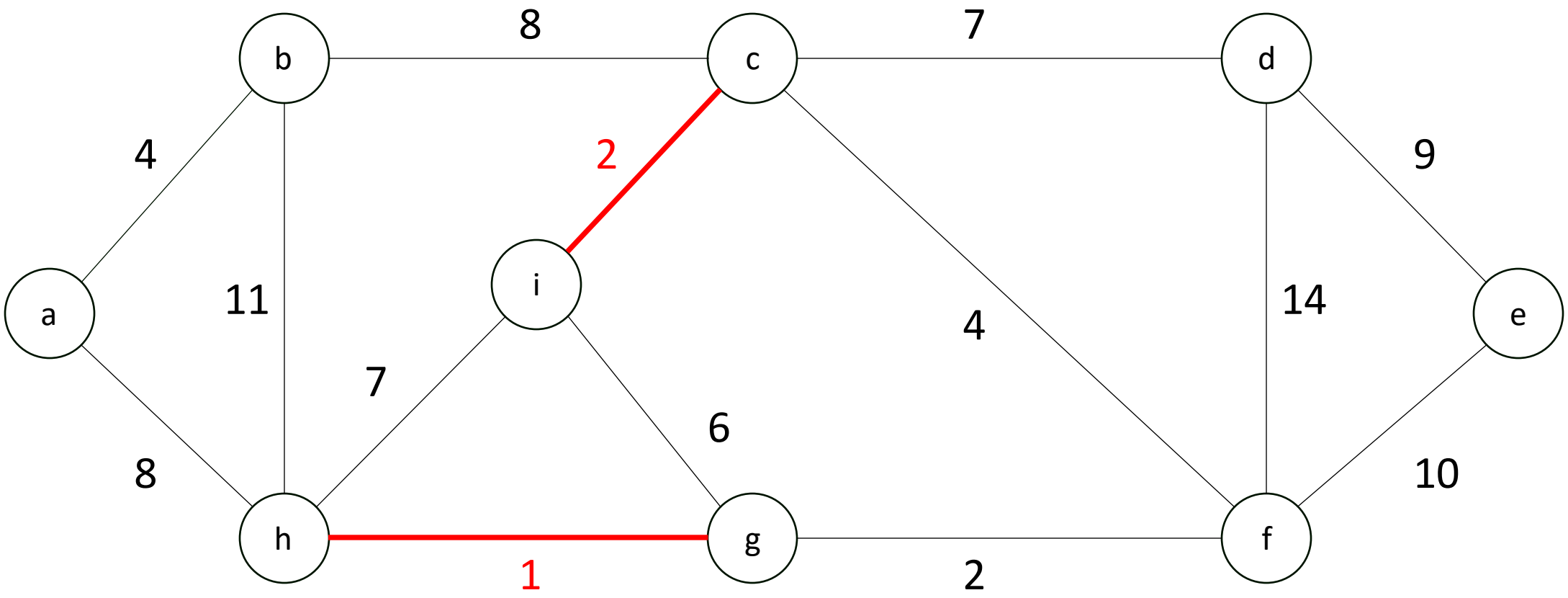
Kruskal's Algorithm Example



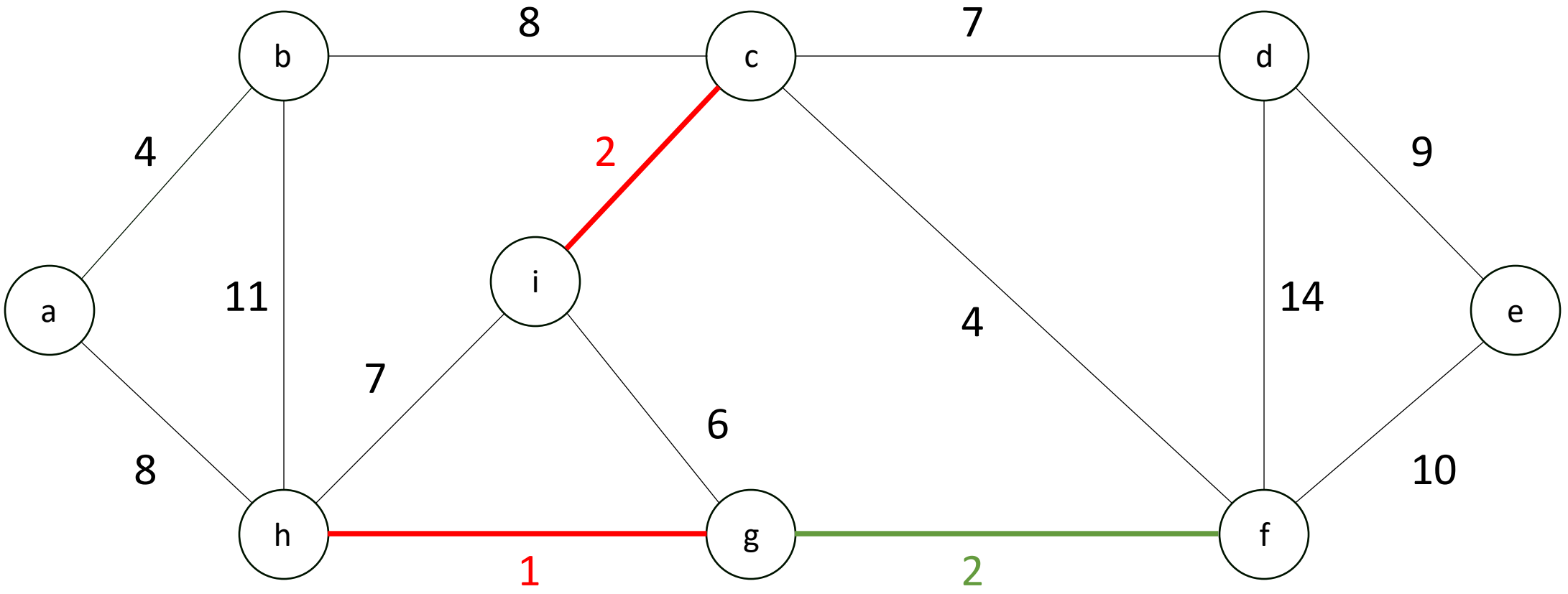
Kruskal's Algorithm Example



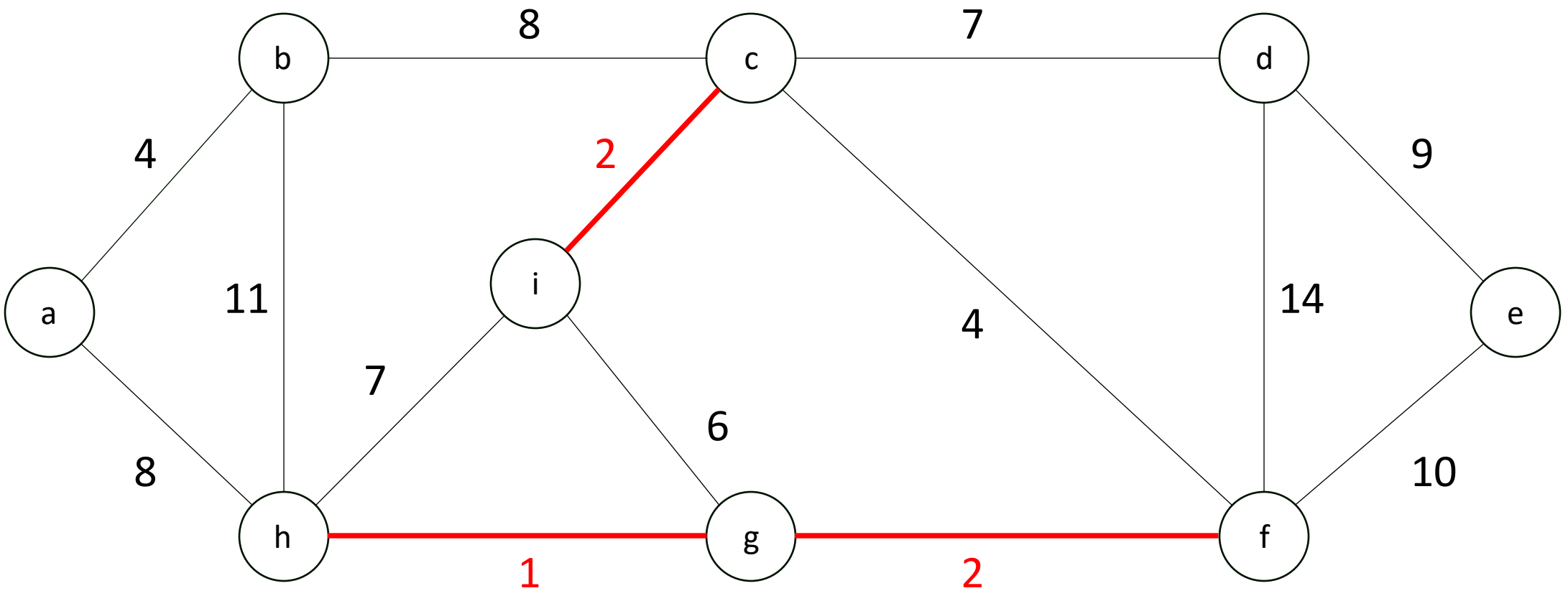
Kruskal's Algorithm Example



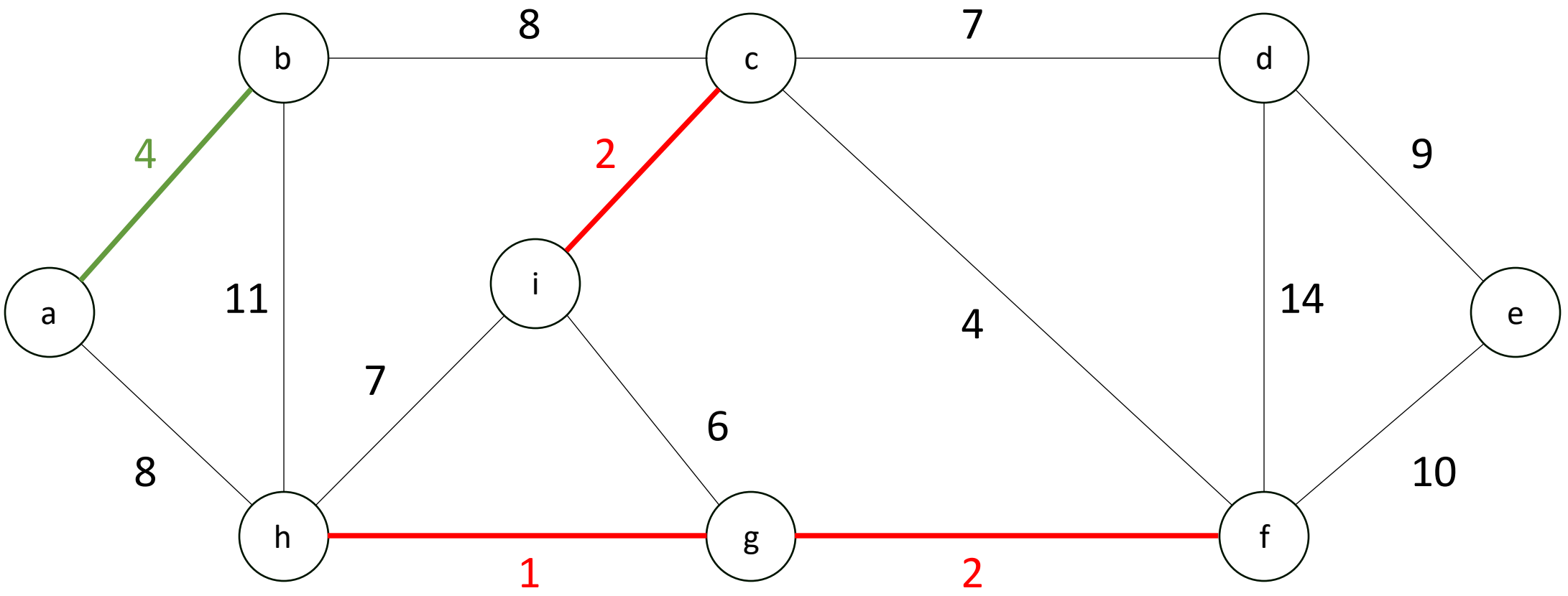
Kruskal's Algorithm Example



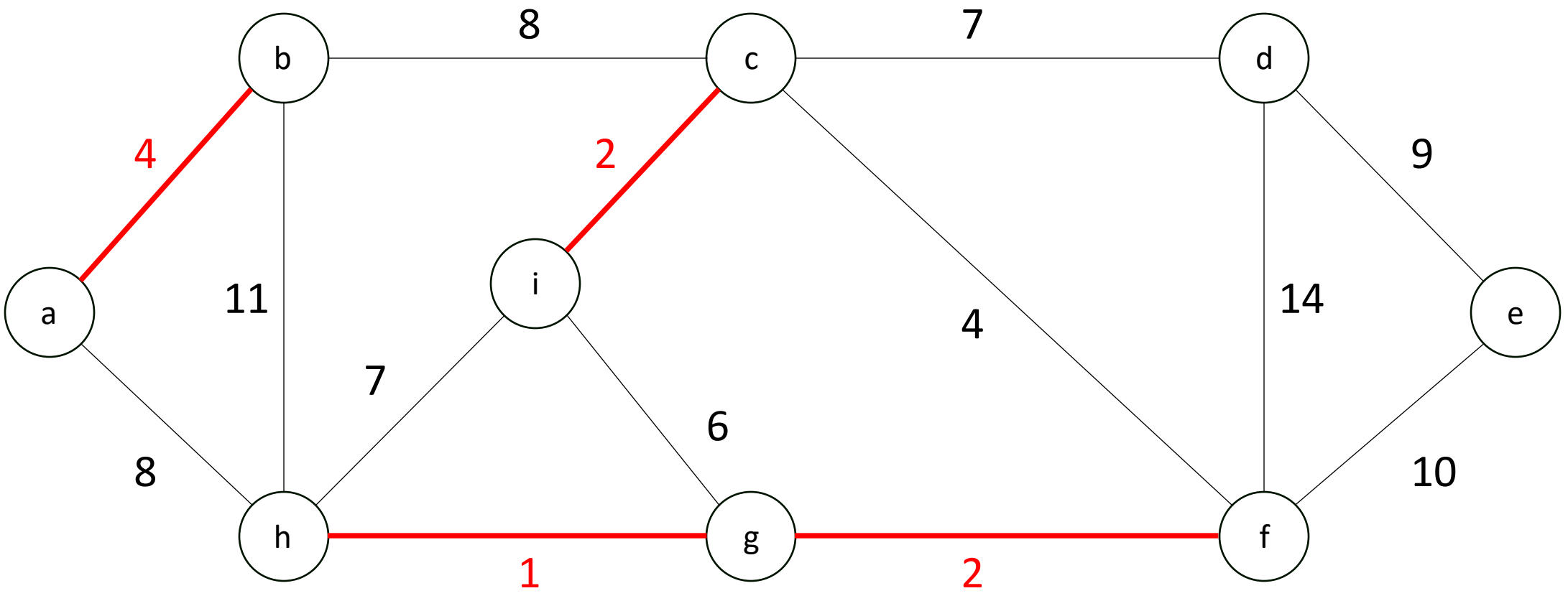
Kruskal's Algorithm Example



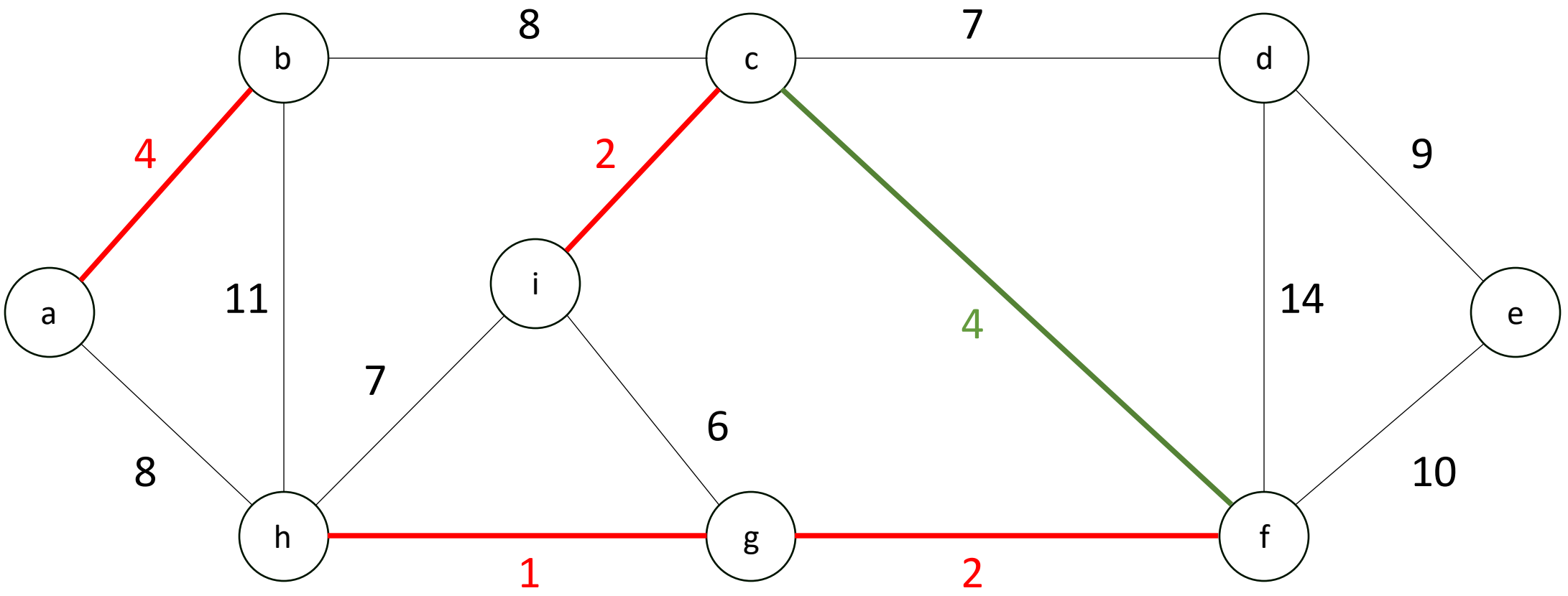
Kruskal's Algorithm Example



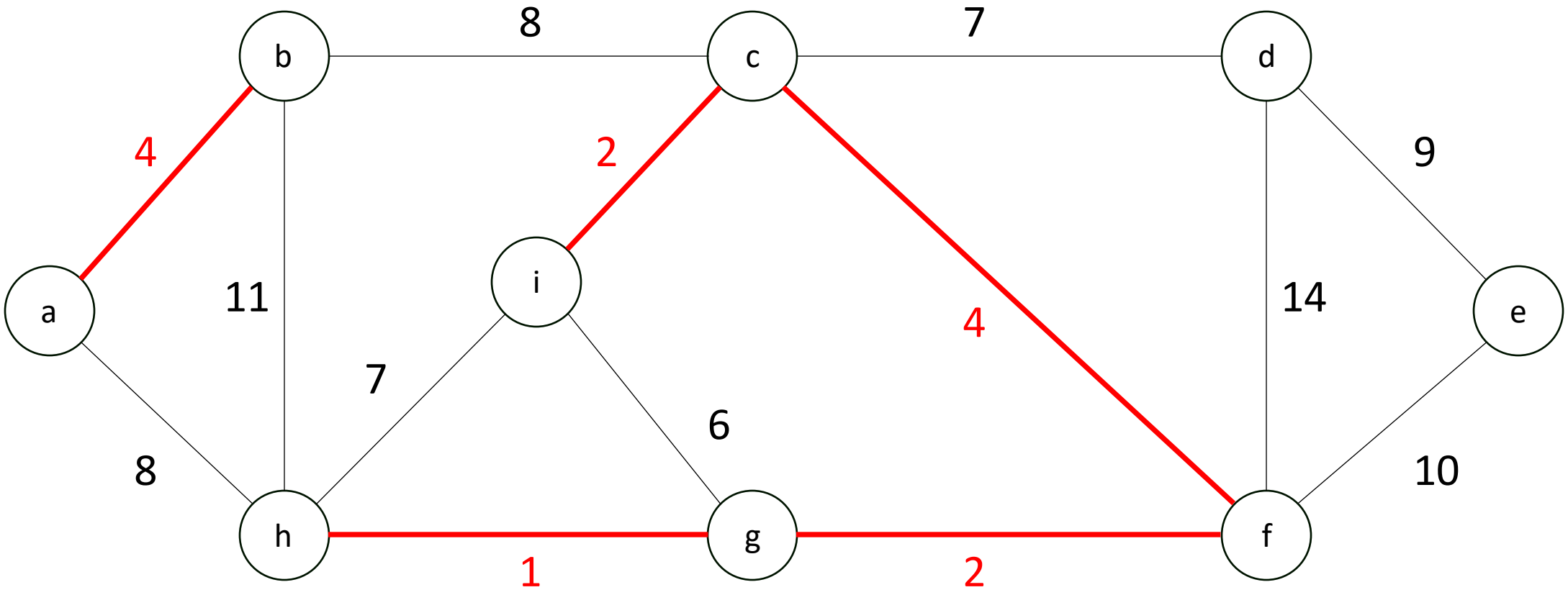
Kruskal's Algorithm Example



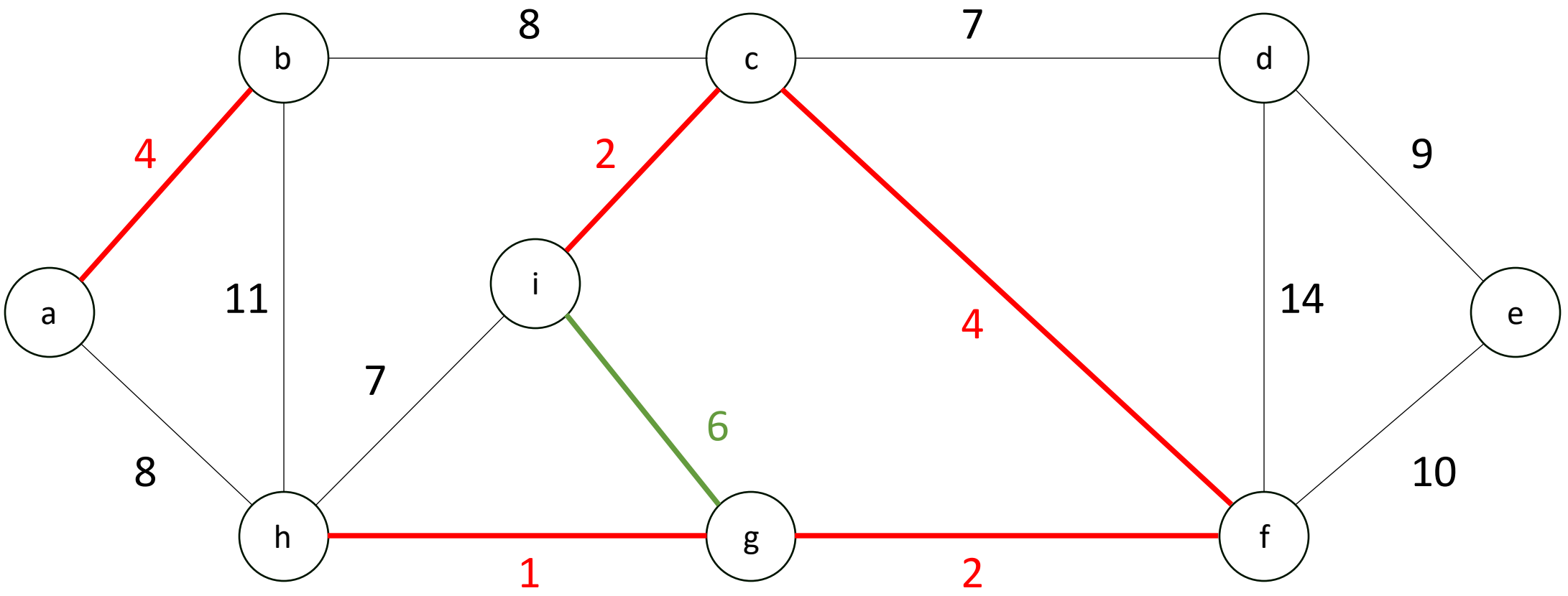
Kruskal's Algorithm Example



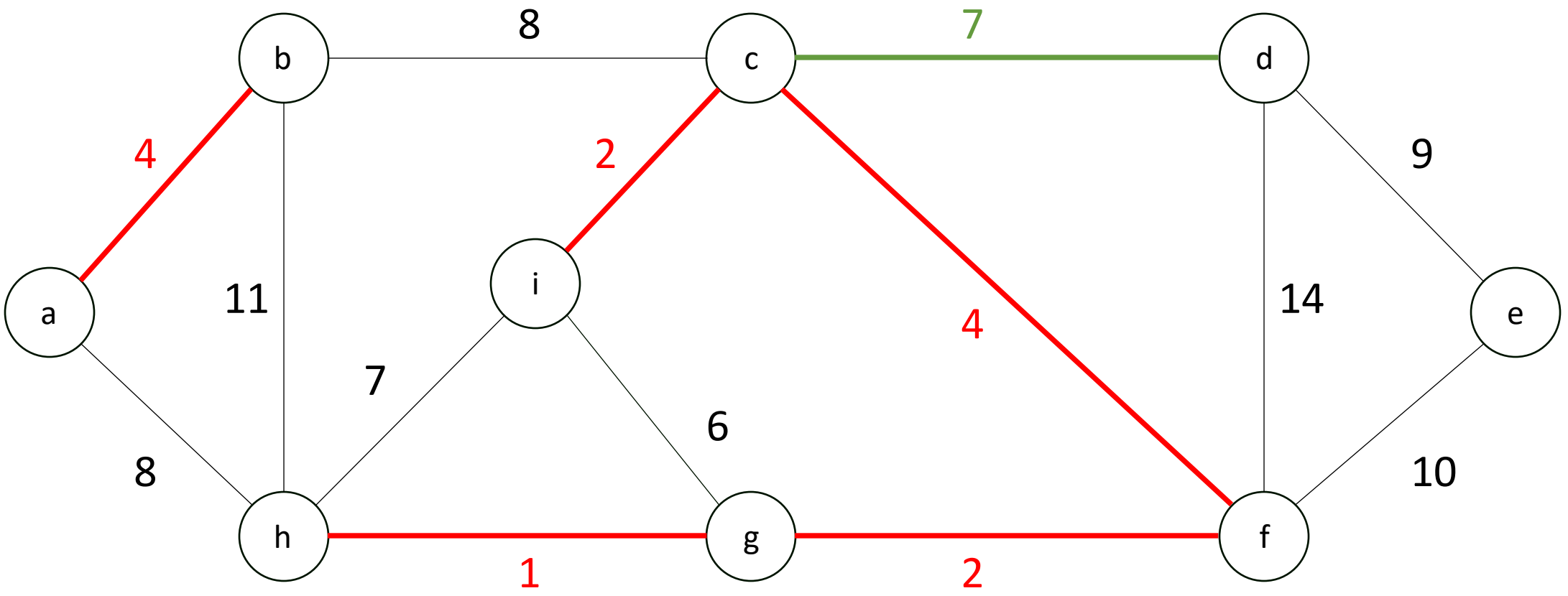
Kruskal's Algorithm Example



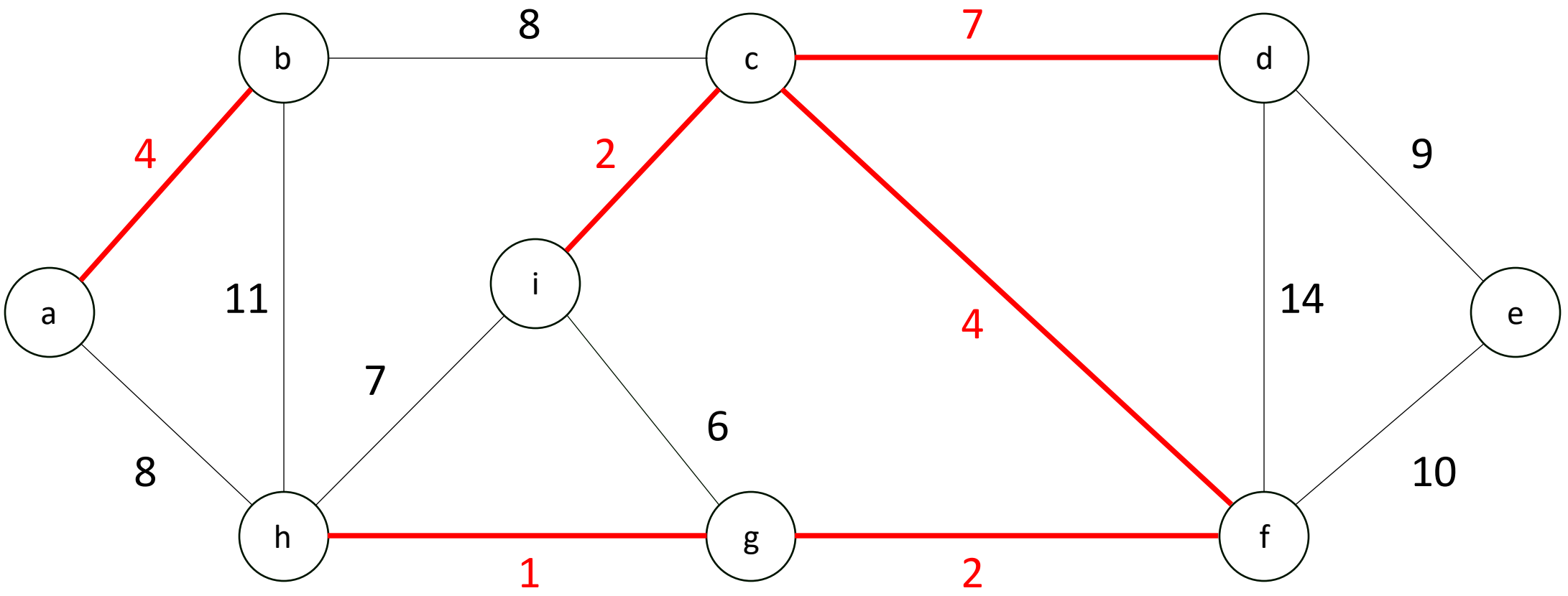
Kruskal's Algorithm Example



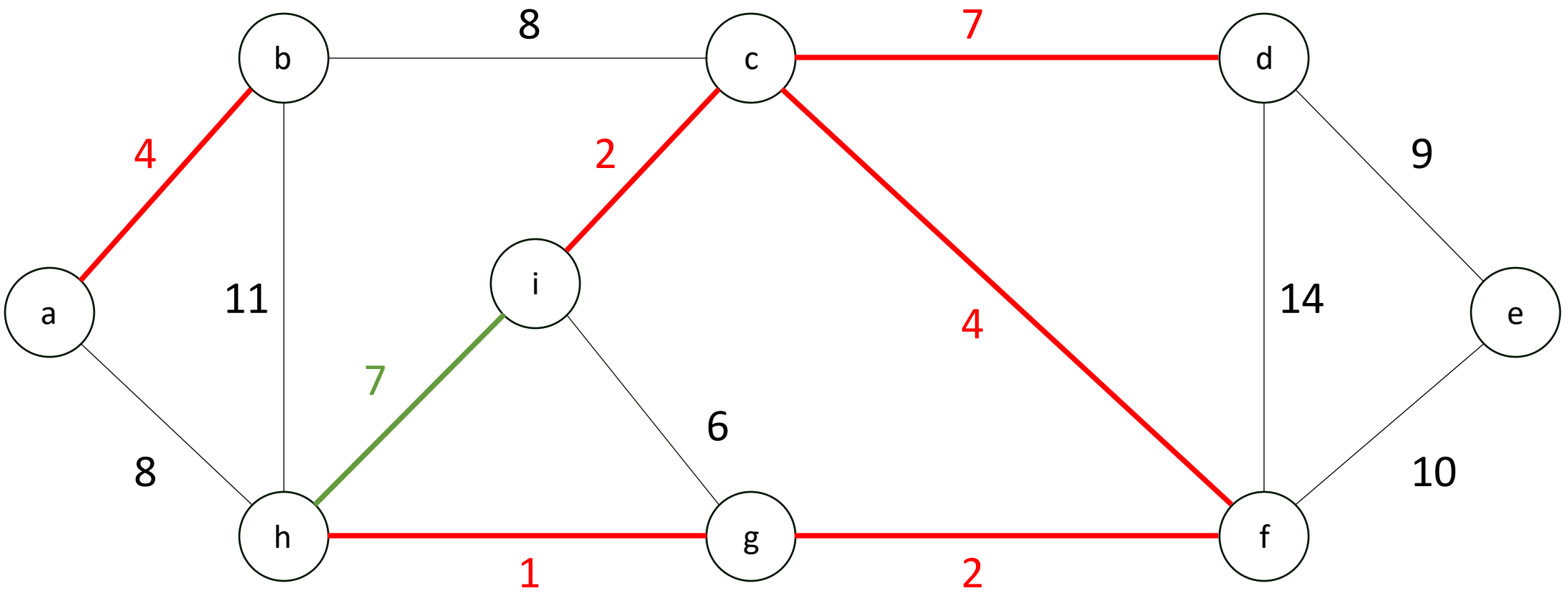
Kruskal's Algorithm Example



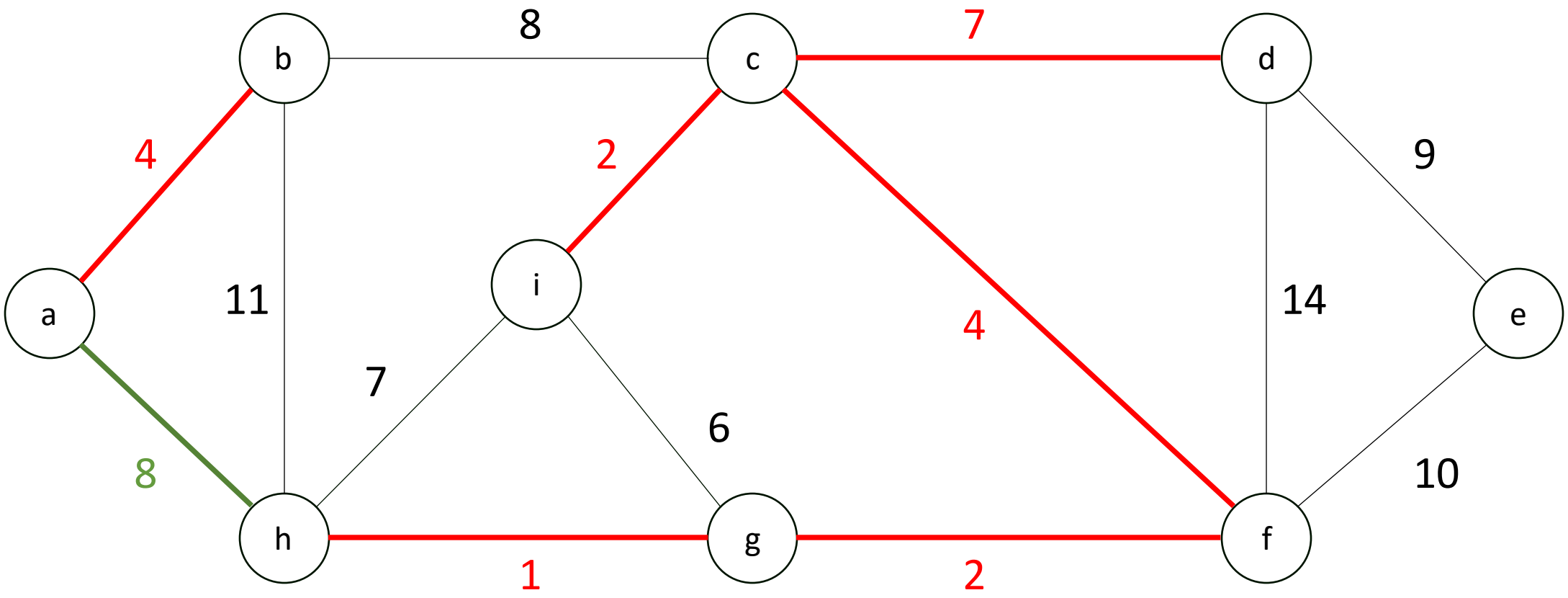
Kruskal's Algorithm Example



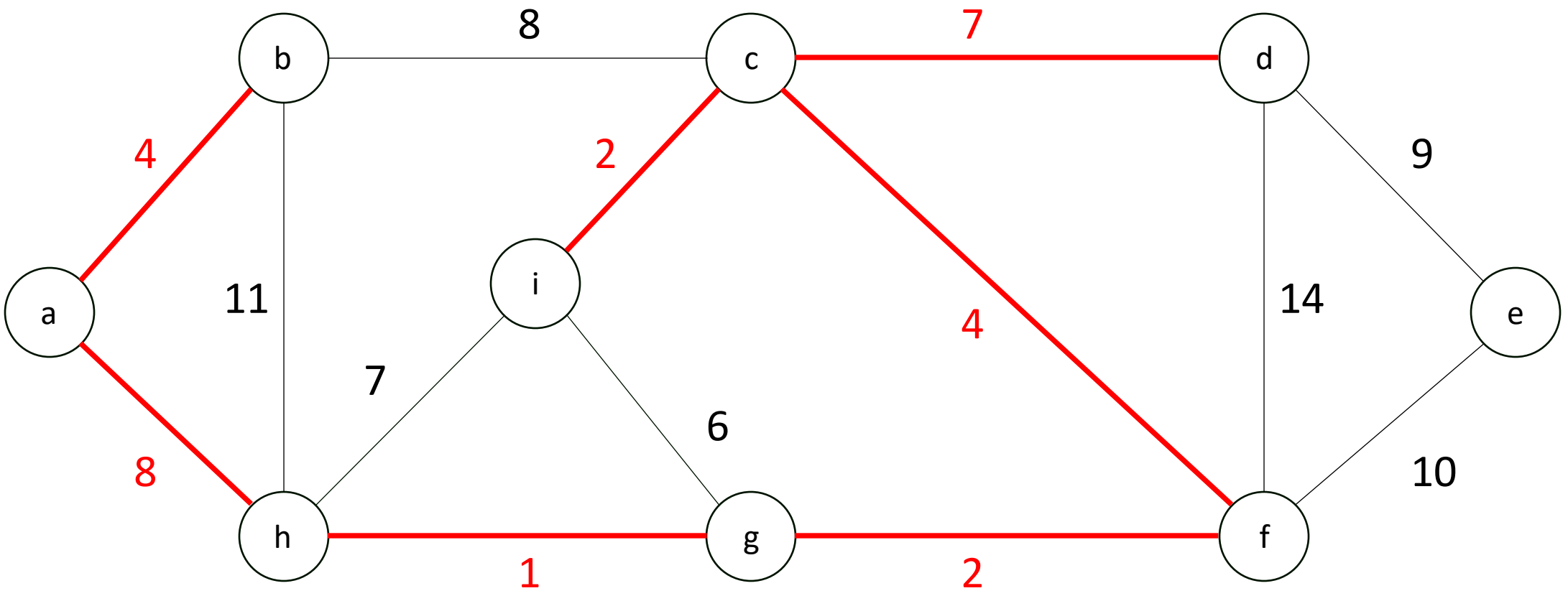
Kruskal's Algorithm Example



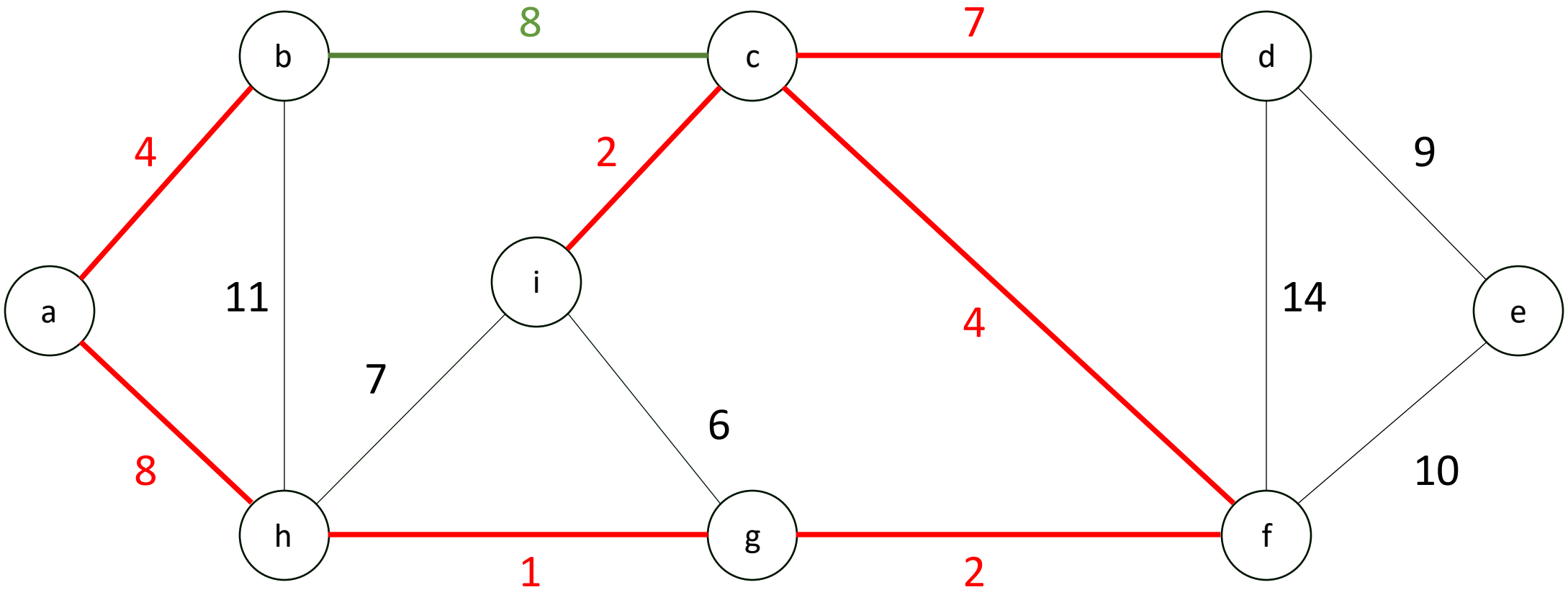
Kruskal's Algorithm Example



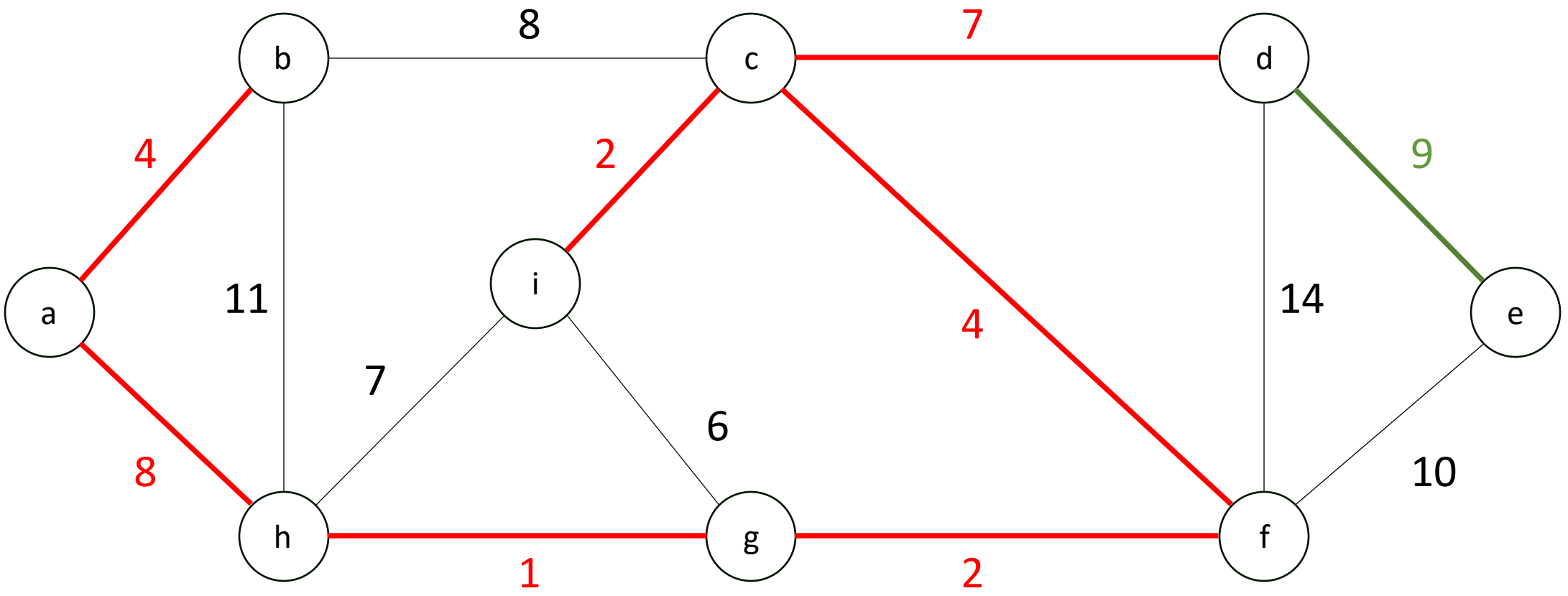
Kruskal's Algorithm Example



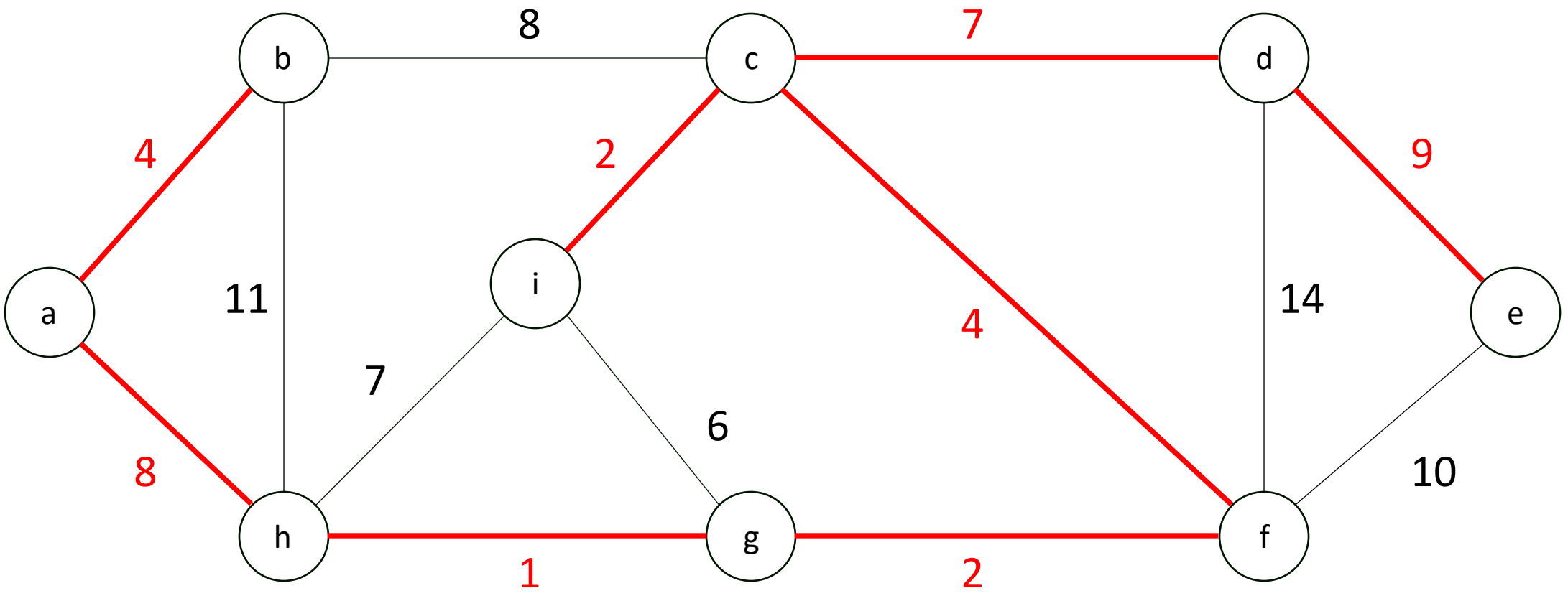
Kruskal's Algorithm Example



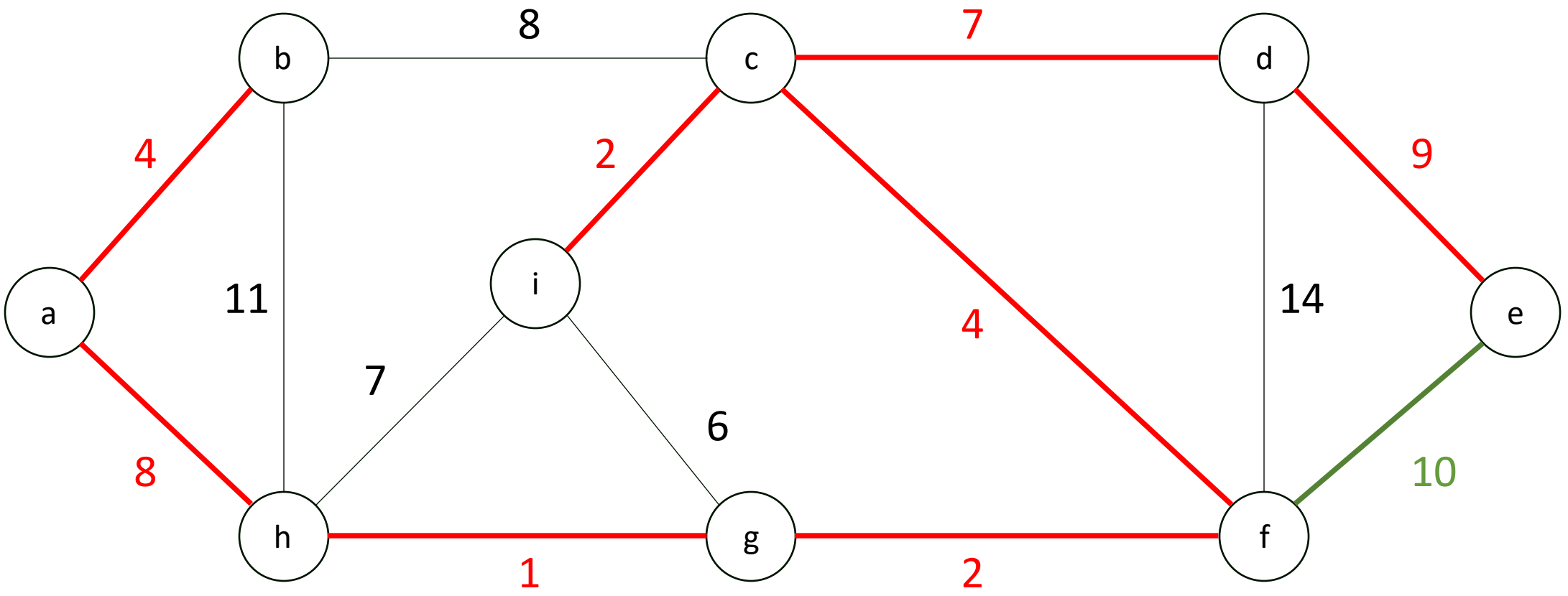
Kruskal's Algorithm Example



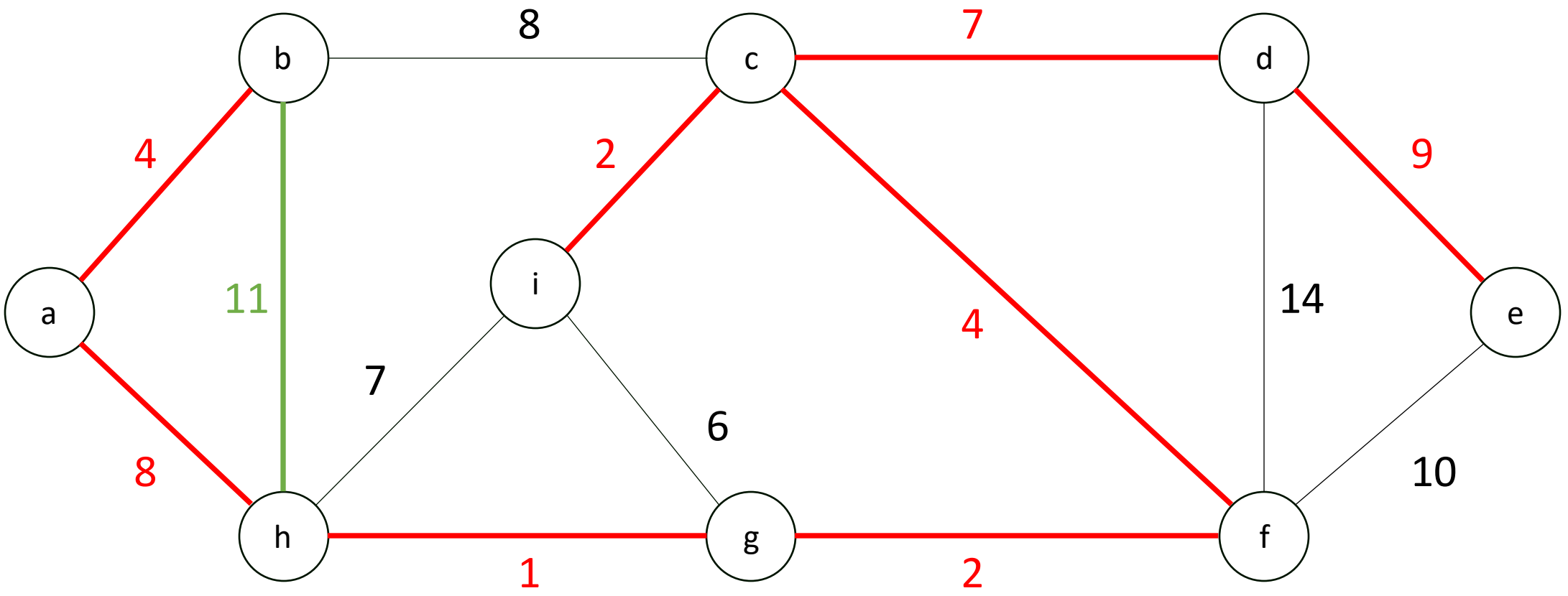
Kruskal's Algorithm Example



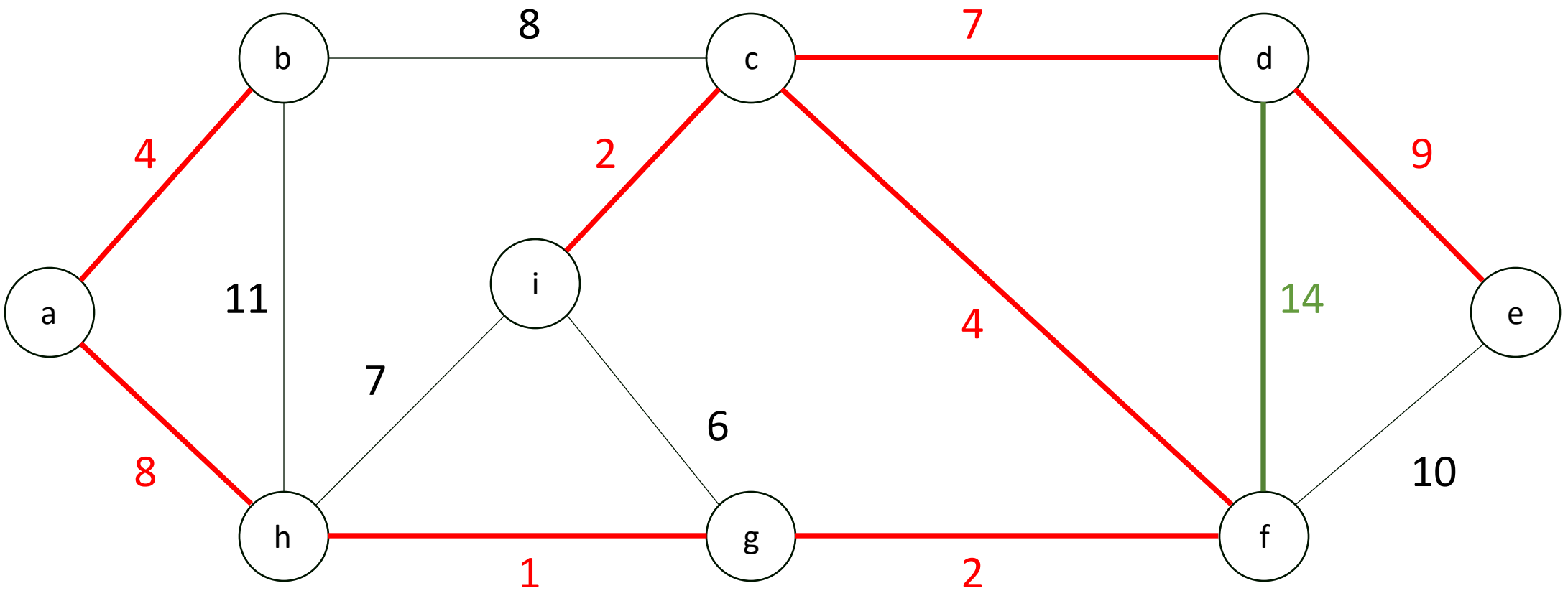
Kruskal's Algorithm Example



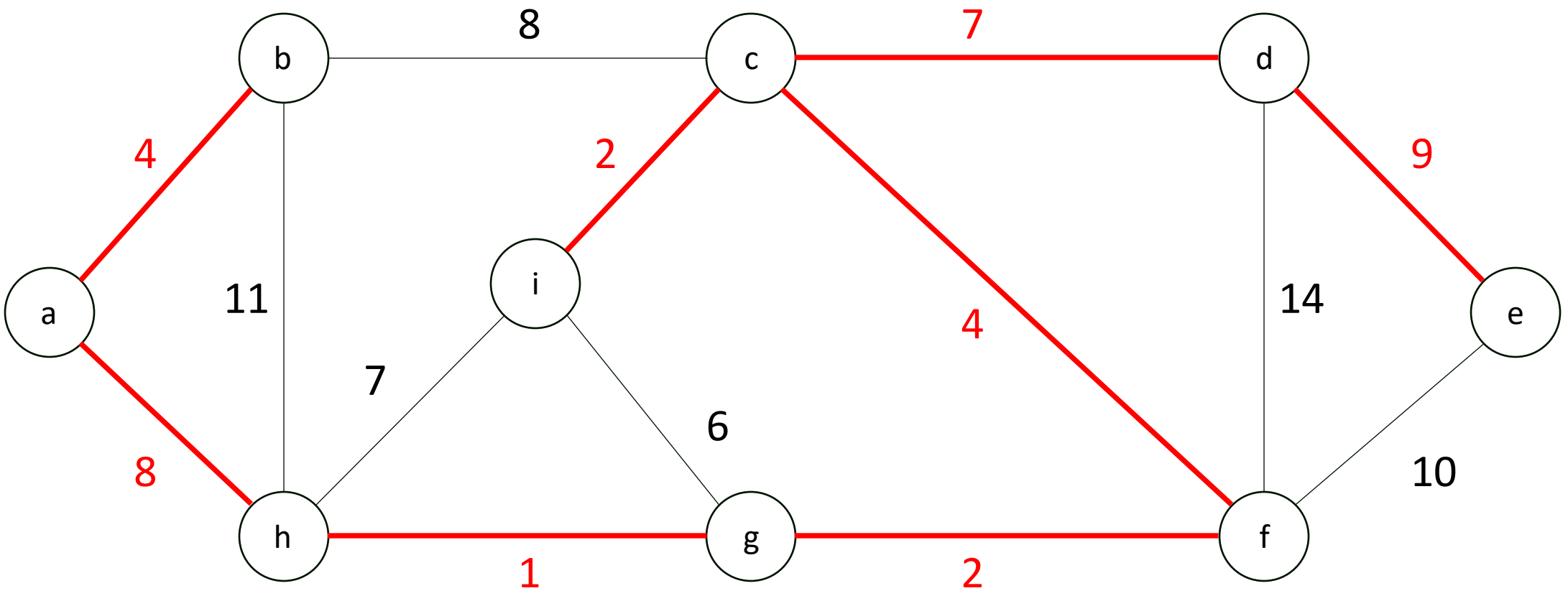
Kruskal's Algorithm Example



Kruskal's Algorithm Example



Kruskal's Algorithm Example



Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1. A = ∅
2. sort G.E into weight-increasing order
3. for each edge (u, v) ∈ G.E
4.     if A ∪ {(u, v)} does not have a cycle:
5.         A = A ∪ {(u, v)}
6. return A
```

- Sort: $O(Elog\ E)$
- Check whether to form a cycle?

Disjoint set

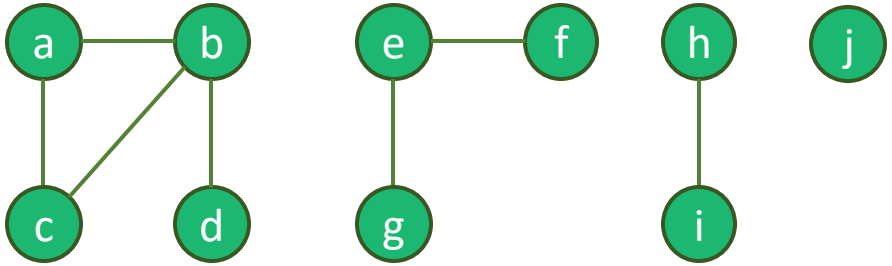
Requirement

Consider a situation: given some persons, how to quickly support the following operations?

- Find whether individual x and y are linked through direct/indirect friend relations
- Add a new friendship relation between x and y

Naïve solution: record the links, then search for all the reachable nodes from x (graph searching)

- High time and space complexity



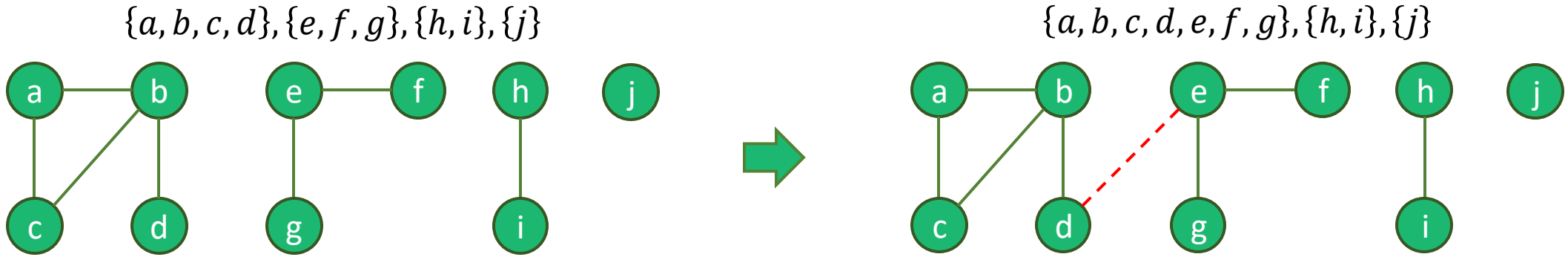
Requirement

Consider a situation: given some persons, how to quickly support the following operations?

- Find whether individual x and y are linked through direct/indirect friend relations
- Add a new friendship relation between x and y

A set-based view

- Nodes in each linked components can reach each other
- Only care about the linked components – ignore links and record with set
- Add a link is to union two sets



Disjoint Set

Store a collection of non-overlapping sets, also called union–find data structure or merge–find set, support operations:

- Distinguish: whether two elements are in the same set

$$< \{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}, < a, c >> \Rightarrow same$$

- Union: given two elements, merge their set

$$\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\} \Rightarrow \{a, b, c, d\}, \{e, f, g, h, i\}, \{j\}$$

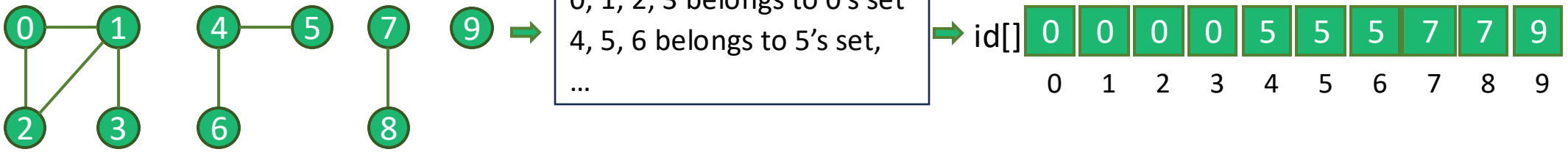
How to realize a disjoint set?

Straightforward solution: create separate collection for each set

- Distinguish: traverse all the collections and elements
- Union: find the corresponding collection, copy elements in one to the other
- **Problem**: large space and time complexity for massive number of sets and operations

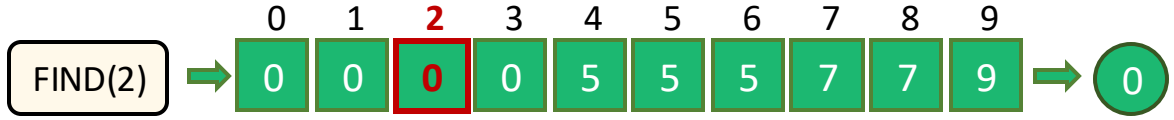
Another Naïve Solution

Core idea: represent each set with a member (representative), record each element's set's representative with an array



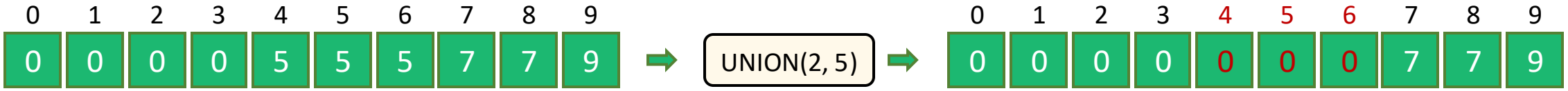
Distinguish: whether two elements are in the same set

- Find and compare the representative: $\Theta(1)$



Union: given two elements, merge their set

- Traverse the array and modify the corresponding elements: $\Theta(n)$



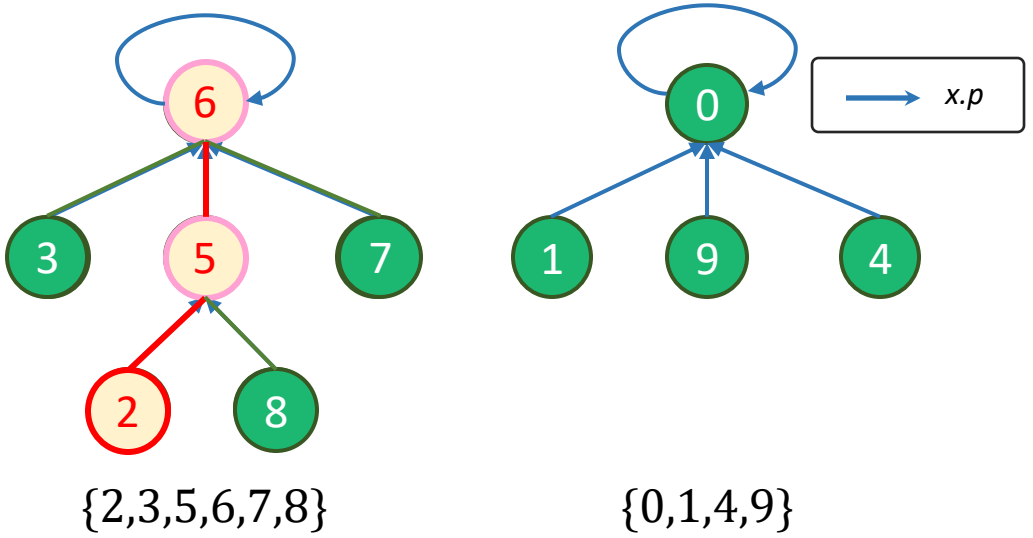
Disjoint Set with Tree

Represent each set with a tree, where each element is a node

- Use root node as the representative

Distinguish whether two elements are in the same set

- Find the root through parent pointer (set the parent of the root to itself to ease coding)



$\{2,3,5,6,7,8\}$ $\{0,1,4,9\}$

```
FIND(x)
1. if x.p != x:
2.     return FIND(x.p)
3. else:
4.     return x
```

Time complexity: $O(h)$

Disjoint Set with Tree

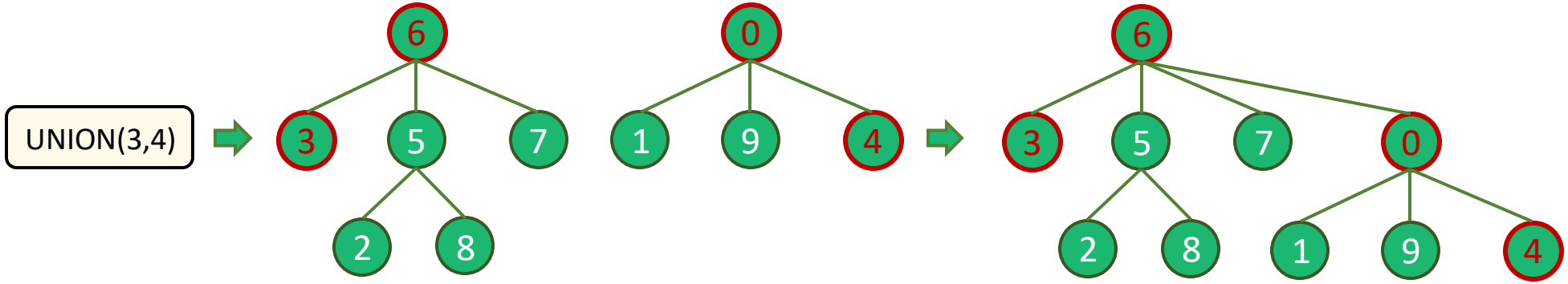
Represent each set with a tree, where each element is a node

- Use root node as the representative

Union: given two elements (x and y), merge their set

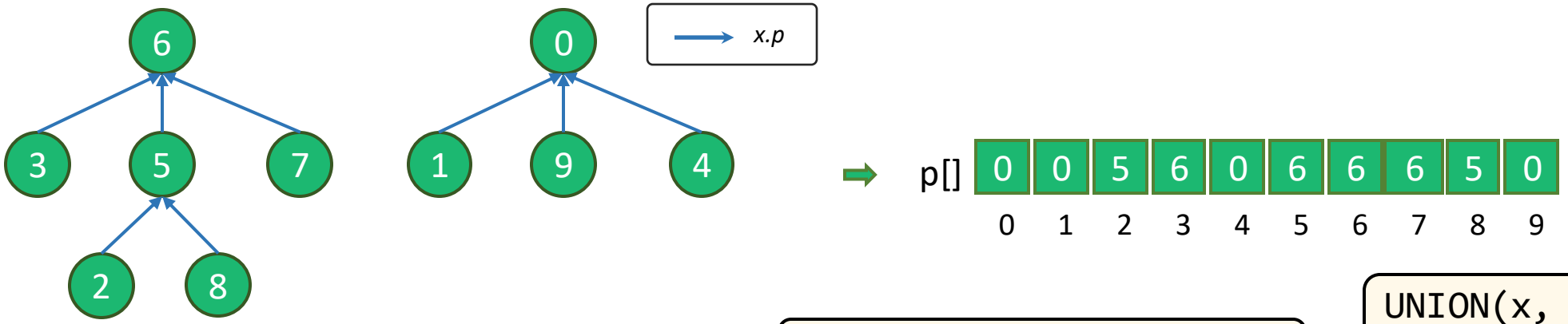
- Find the roots of x and y's corresponding tree $O(h)$
- Make x's roots as the parent of y's root $O(1)$

```
UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     root_y.p = root_x
```



Disjoint Set with Tree

Store tree structure in an array: since only parent information is needed



{2,3,5,6,7,8}

{0,1,4,9}

```
FIND(x)
1. if p[x] != x:
2.     return FIND(p[x])
3. else:
4.     return x
```

```
UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     p[root_y] = root_x
```

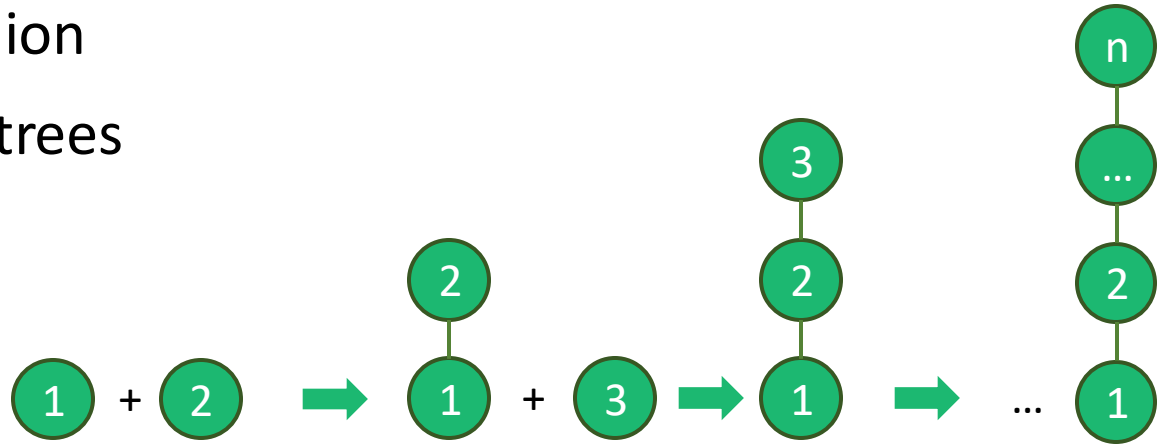
Disjoint Set with Tree

Any potential deficiency?

- Link $< 1, 2 >, < 2, 3 >, < 3, 4 >, < 4, 5 >, < 5, 6 >, \ldots \ < n-1, n >$
- $\mathrm{T}(n) \ = \Theta(n)$ for find and union

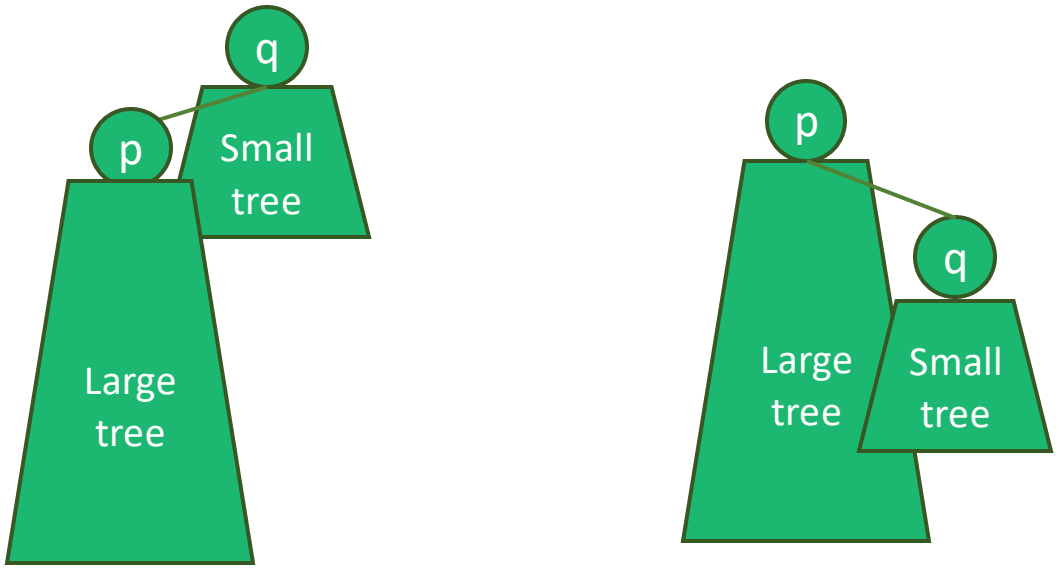
Improve complexity: avoid tall trees

- Union-by-Rank
- Path Compression



Union by Rank

Keep track each tree's size (number of nodes), link roots of smaller tree to root of larger tree.



Intuitively reduce height

```
Weight_UNION(x, y)
1. root_x = FIND(x)
2. root_y = FIND(y)
3. if root_x == root_y:
4.     return
5. else:
6.     if root_x.size < root_y.size:
7.         root_x.p = root_y
8.         root_y.size += root_x.size
9.     else:
10.        root_y.p = root_x
11.        root_x.size += root_y.size
```

Union by Rank

Proposition: With union by rank, depth of any node x is at most $\log n$.

Proof:

- Notation: $d(x)$ - depth of x, $|T|$ - size of a tree T
- $d(x)$ possibly increase when the tree (T) containing x is merged with another tree T':
  - $d(x)$ increases by 1 only when $|T'| \geq |T|$
  - The new tree's size $|T''| = |T| + |T'| \geq 2|T|$, at least doubles.
  - If $d(x) = h$, then $2^h \leq |T|$. Since $|T| \leq n$, $h \leq \log n$

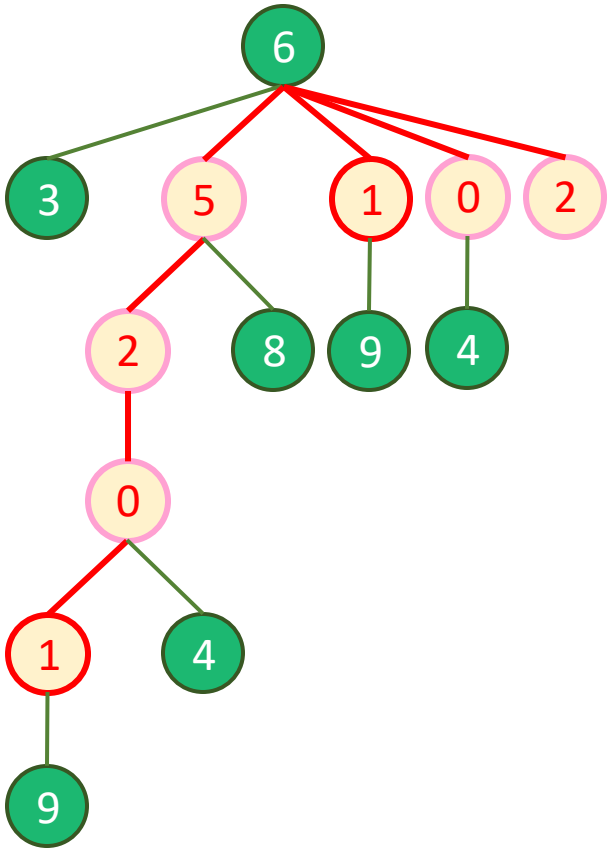
Conclusion: with union by rank, time complexity for find and union operation are $O(\log n)$

Path Compression

After finding the root for each node, move the node to be its child

- Find: O(1) after many find operations
- Union: with m arbitrary linking operations, time complexity is roughly $O((m+n)\log n)$[1]

Path compression + union by rank: nearly O(1) on average




```
FIND_PC(x)
1. if x.p != x:
2.     x.p = FIND_PC(x.p)
3. return x.p
```

[1] R. Seidel and M. Sharir. Top-Down Analysis of Path Compression. Siam J. Computing, 2005, Vol. 34, No. 3, pp. 515-525.

Kruskal's Algorithm

- Not to form a cycle: merge the two nodes connected and ignore the self loops with both ends in the merged group

```
MST-KRUSKAL(G, w)
1. A = ∅
2. for each vertex v ∈ G.V
3.     MAKE-SET(v)
4. sort G.E into weight-increasing order
5. for each edge (u, v) ∈ G.E
6.     if FIND-SET(u) ≠ FIND-SET(v)
7.         A = A ∪ {(u, v)}
8.         UNION(u, v)
9. return A
```

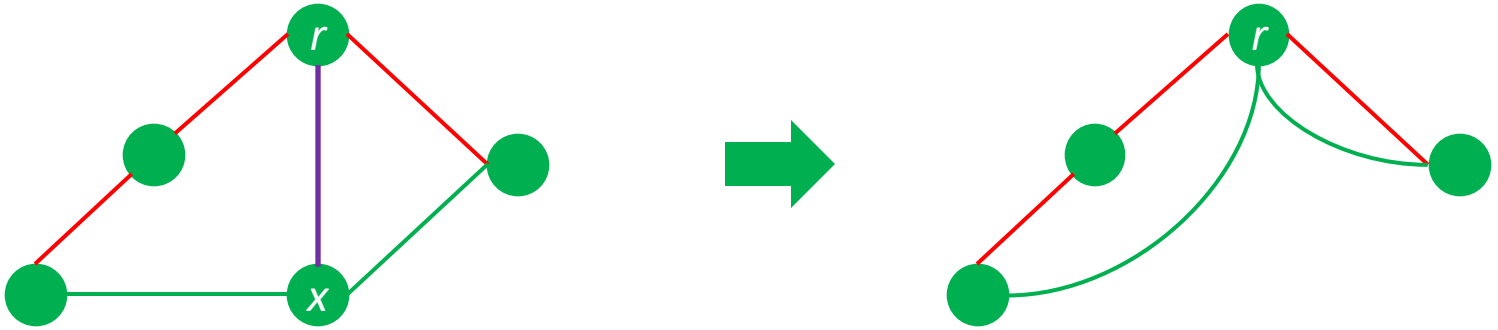
- Sort: $O(Elog\ E)$
- Check whether to form a cycle: $O(E)$ or $O((V+E)\log V)$

Another Optimal Substructure

Given an MST $C =< c_1, c_2, \ldots, c_{n-1} >$ for graph $G =< V, E >$ with n nodes. For any edge $e_{c_r} =< root, x > or < x, root >$ contained in C that links root with another node, $C - \{c_r\}$ is an MST for $G' = g(G, x) =< V', E' >$, where

$$V' = V - \{x\},$$

$$E' = \{< root\ if\ u' = x\ else\ u',\ root\ if\ v' = x\ else\ v' > \mid < u', v' > \in E\}$$



Recurrence for Optimized Value

Given an MST $C =< c_1, c_2, \dots, c_{n-1} >$ for graph $G =< V, E >$ with n nodes. For any edge $e_{c_r} =< root, x > or < x, root >$ contained in C that links root with another node, $C - \{c_r\}$ is an MST for $G' = g(G, x) =< V', E' >$, where

$$V' = V - \{x\},$$

$$E' = \{< root\ if\ u' = x\ else\ u',\ root\ if\ v' = x\ else\ v' > | < u', v' > \in E\}$$

Recurrence for optimized value: $f(G) = \min_{e=<x,root>\in E}(f(g(G, < x, root >)) + e.w)$

Complexity: exponential (2^{n-1}) subproblems (subgraphs)

Intuition: greedily select the edge linking with root with least weight $\min_{e=<x,root>\in E} e.w$

Is it optimal?

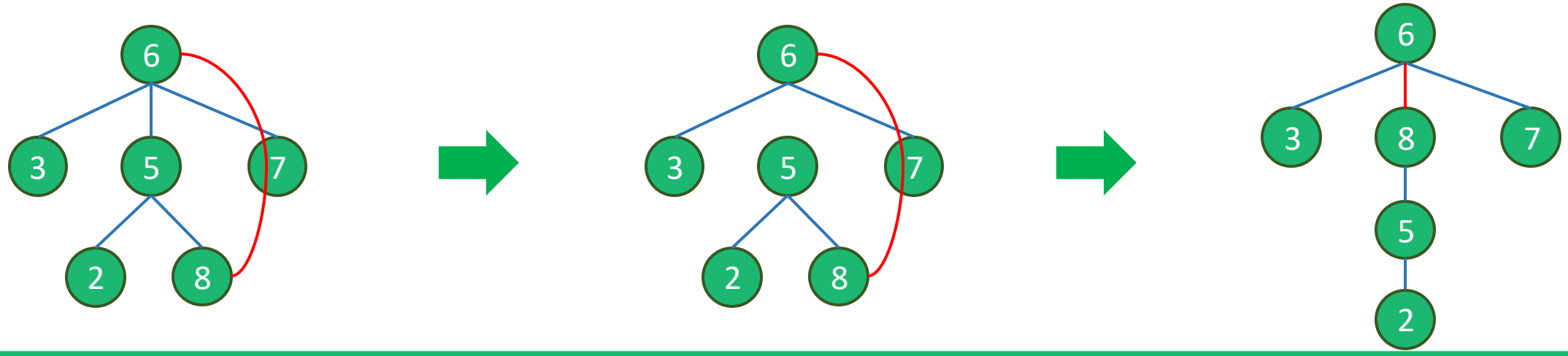
—— prove the edge with least distance connecting root is contained in an MST

Greedy Choice Property

There exists an MST that contains the edge $e^* = \min_{e=<x,root>\in E} e.w$

Proof: Given any MST T not containing e^*, we can construct an MST T' containing e^*

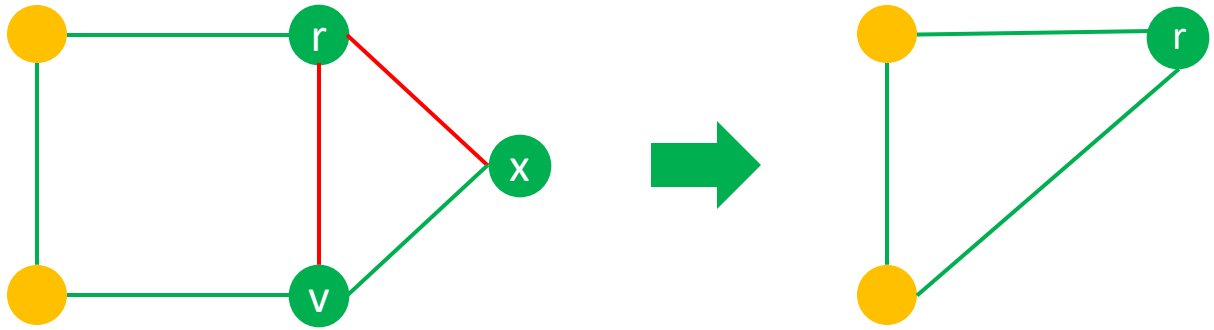
- Add $e^* =< r, x >$ to T, there is a cycle containing r and x and a child y of r
- Delete $e' =< y, r >$ to obtain the new tree T'. Since $e^* = \min_{e=<x!=root,root>\in E} e.w$, $e^*.w \leq e'.w$. $T'.w - T.w = e^*.w - e'.w \leq 0$, since T is MST, T' is MST that contains e^*



Prim's Algorithm

Substitution on merging the nodes:

- Since all the selected nodes are merged into root
- Greedily select the edge that has the smallest weight linking an unselected node to the current selected nodes

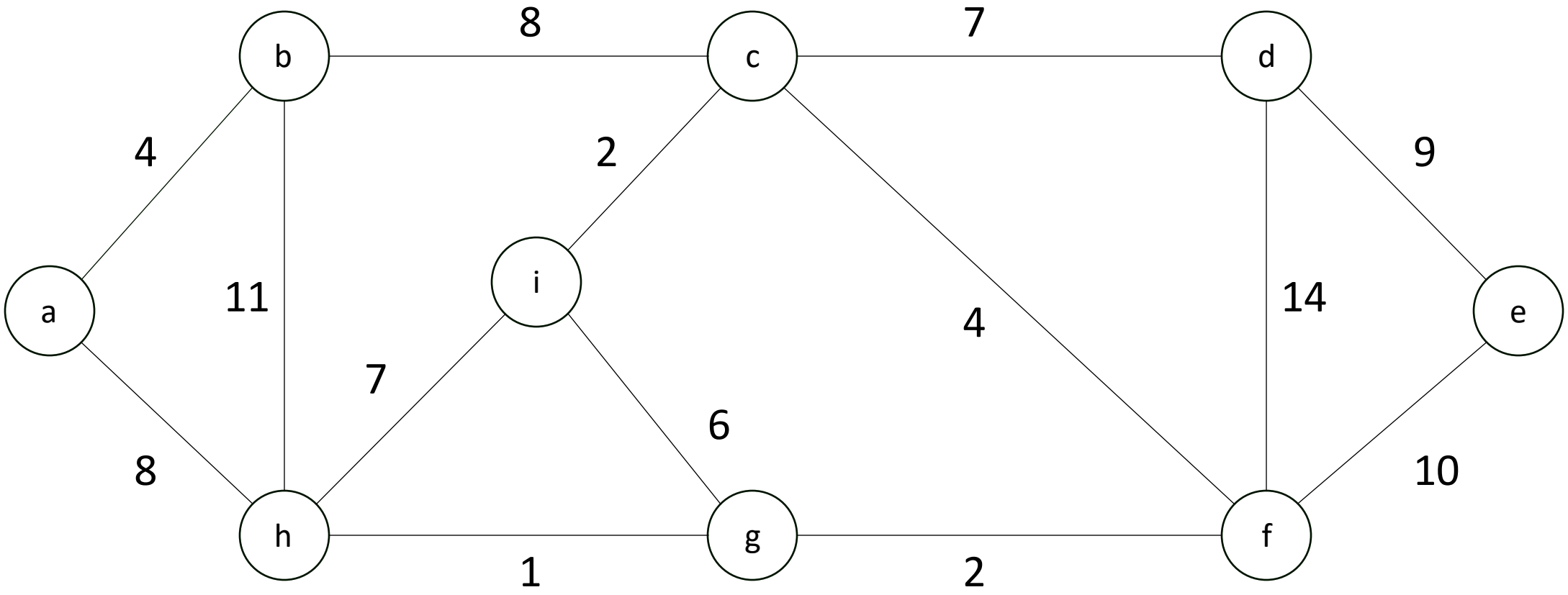


Prim's Algorithm

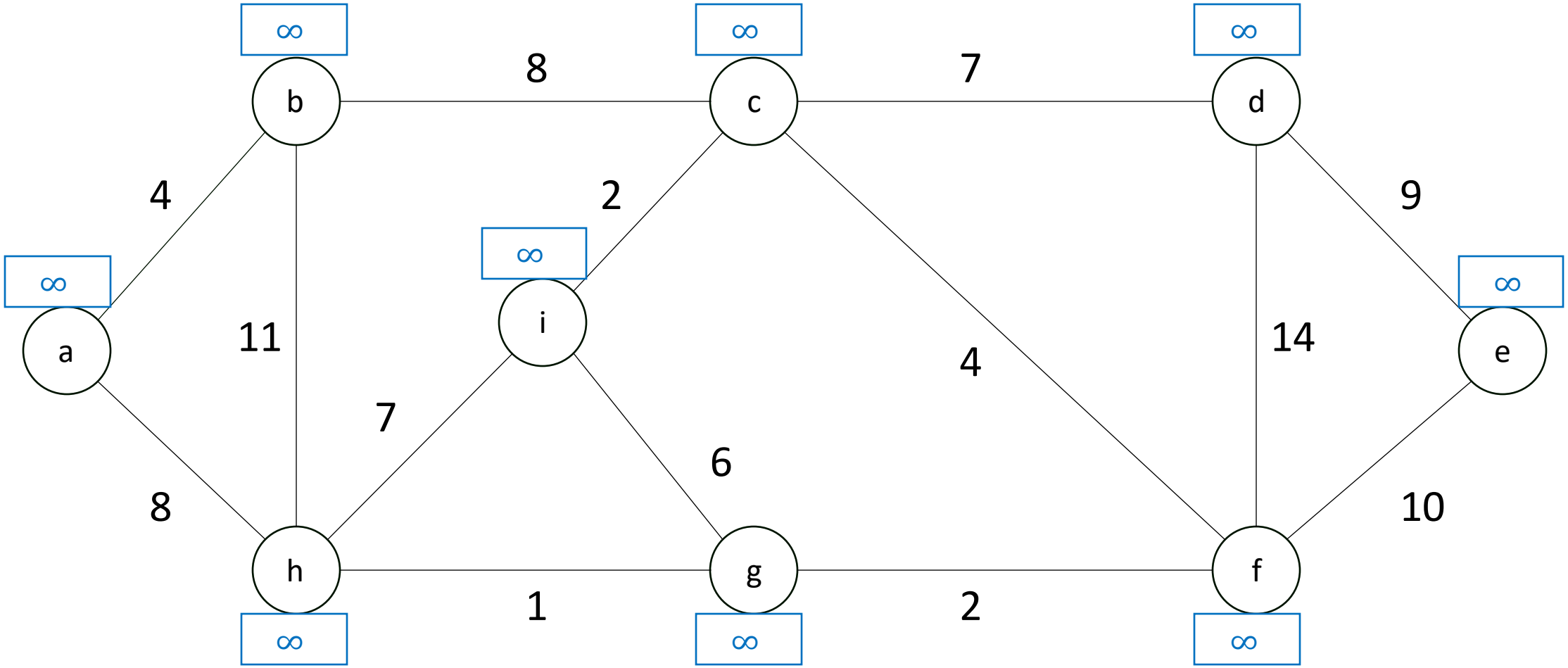
Another algorithm for minimum spanning tree that operates the nodes instead of edges (add nodes to the MST)

1. Randomly pick a node as the root
2. Calculate the minimum distance from each nodes outside of the tree to a node in the tree
3. Pick the node with smallest distance, add to a tree
4. Update the distances
5. Repeat 3-4 until all the nodes are in the tree

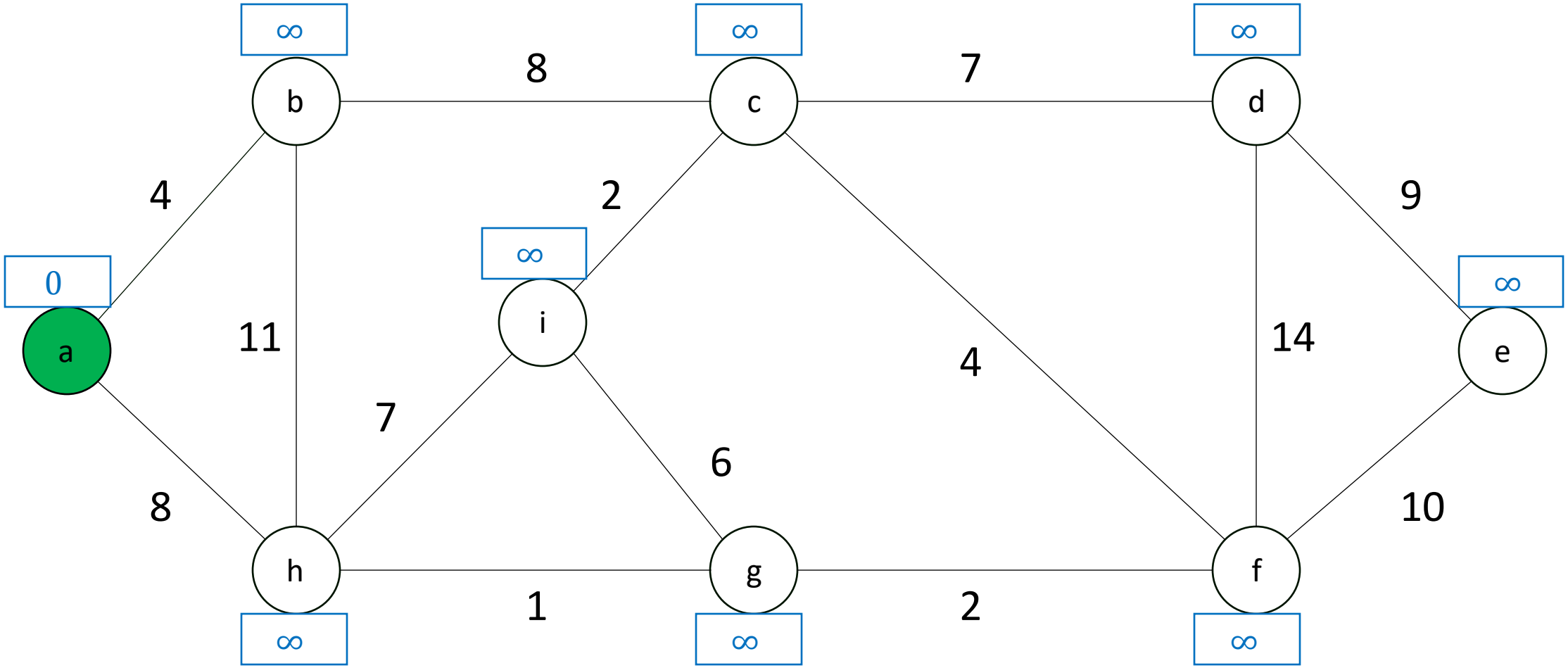
Prim's Algorithm Example



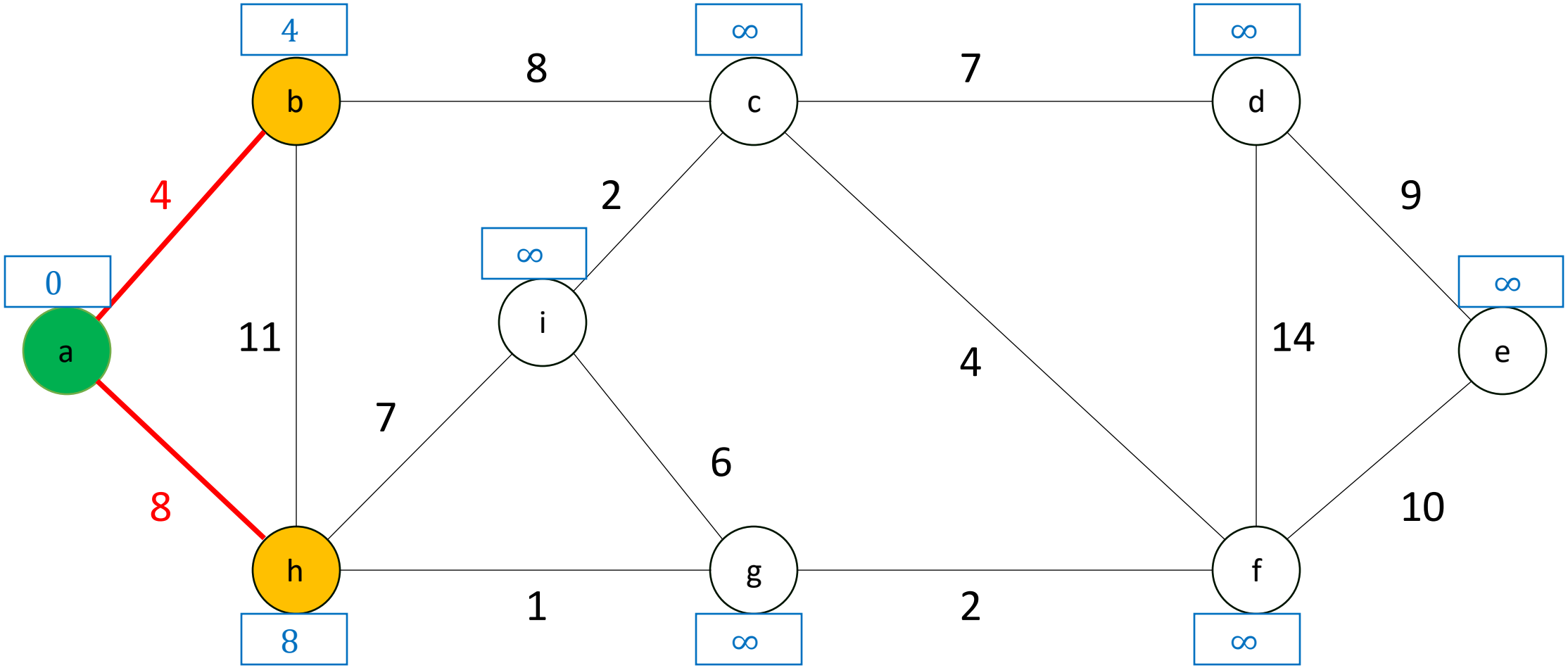
Prim's Algorithm Example



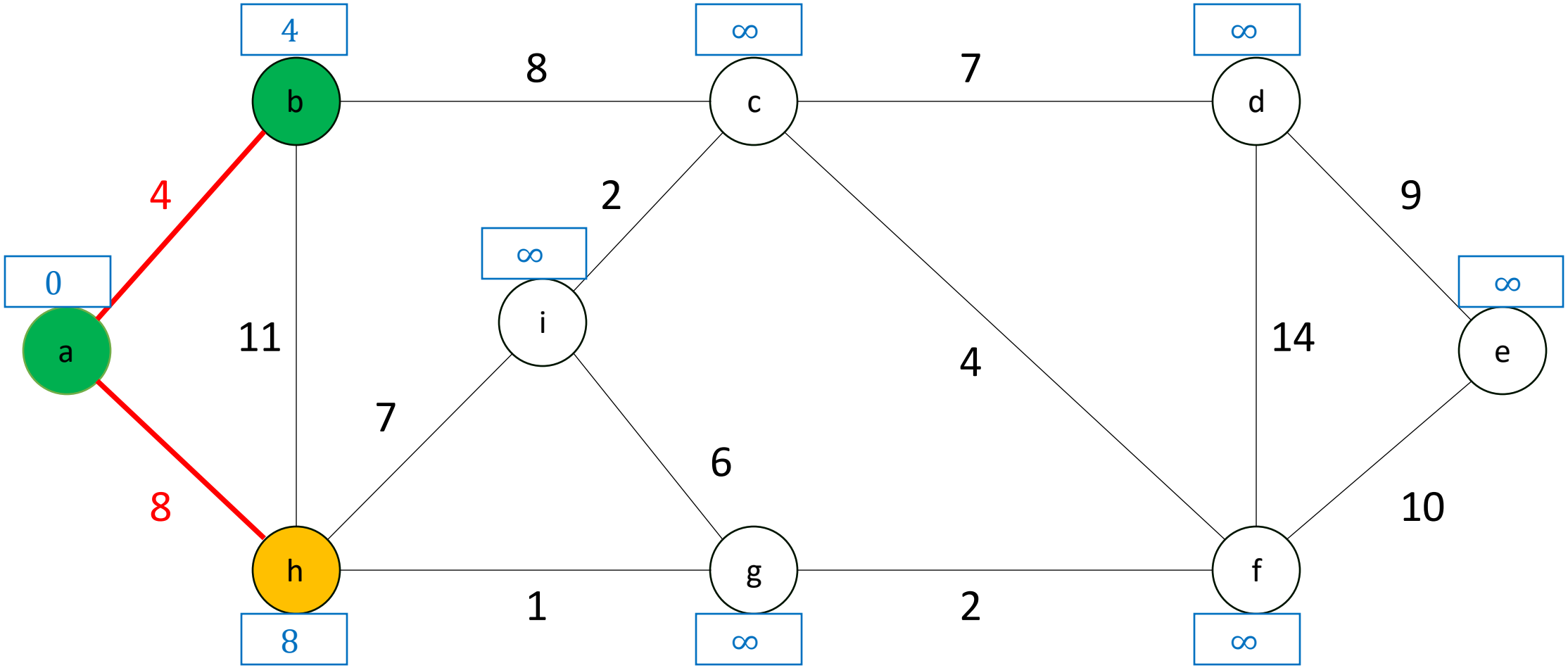
Prim's Algorithm Example



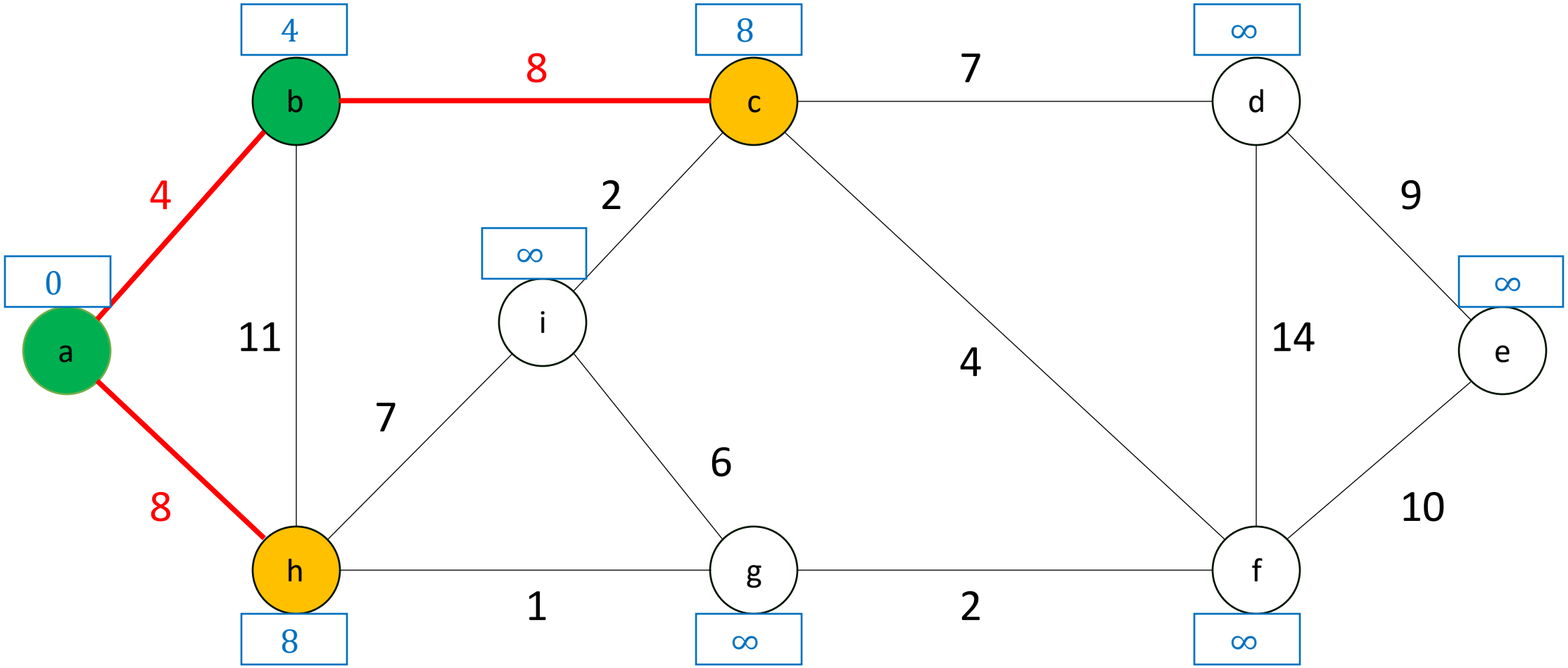
Prim's Algorithm Example



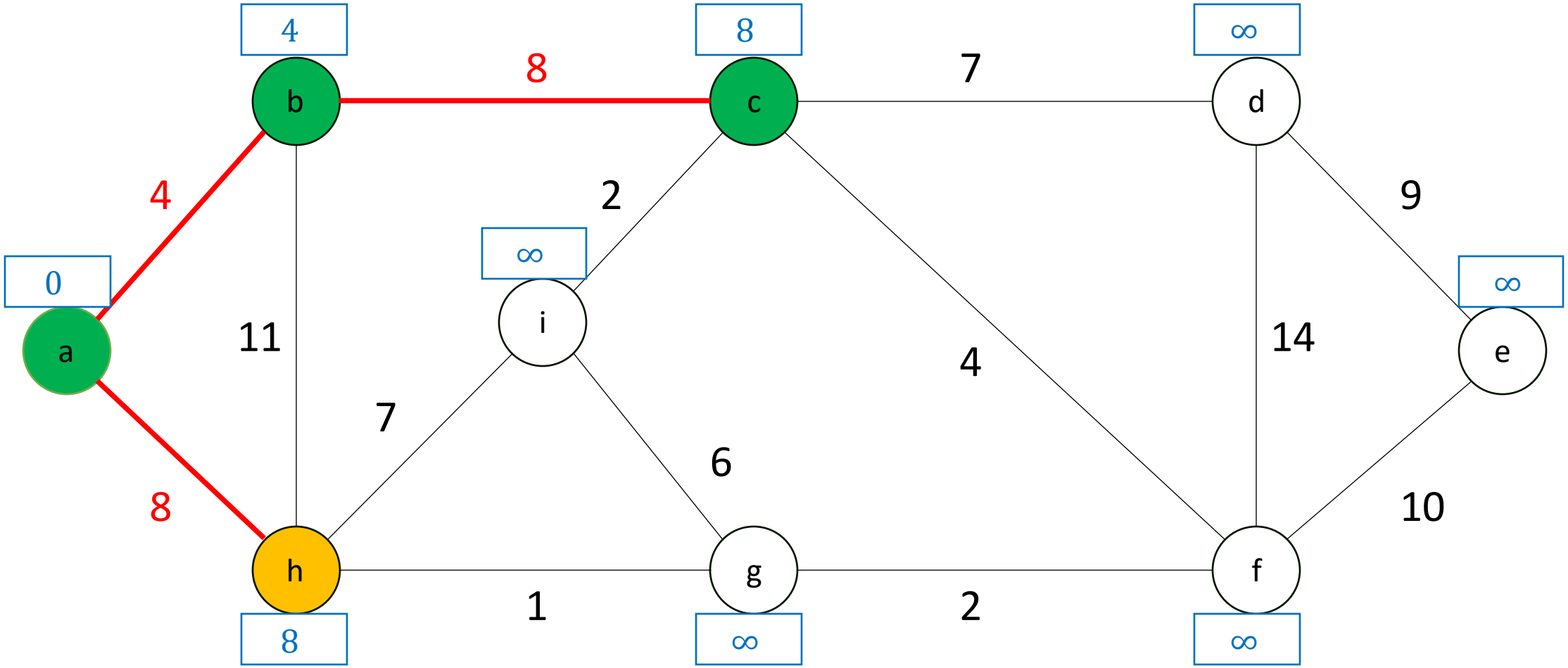
Prim's Algorithm Example



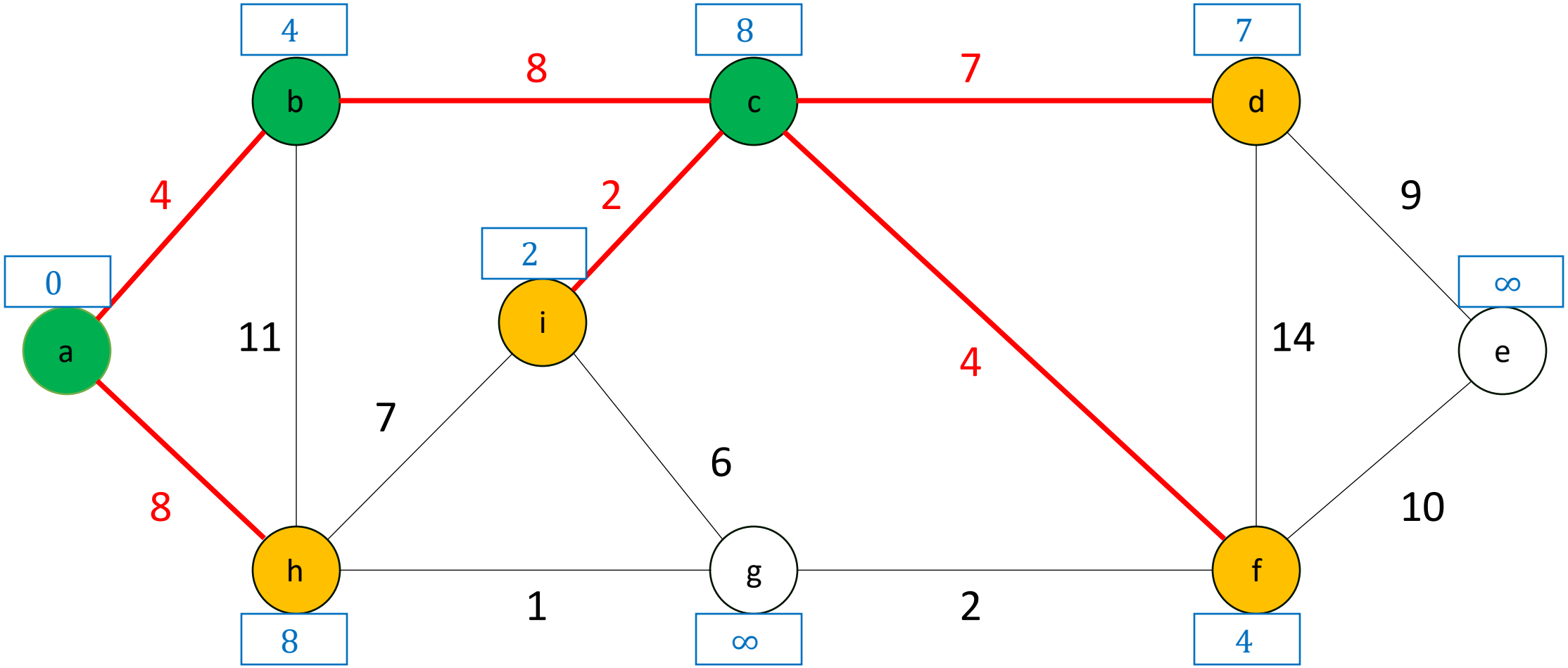
Prim's Algorithm Example



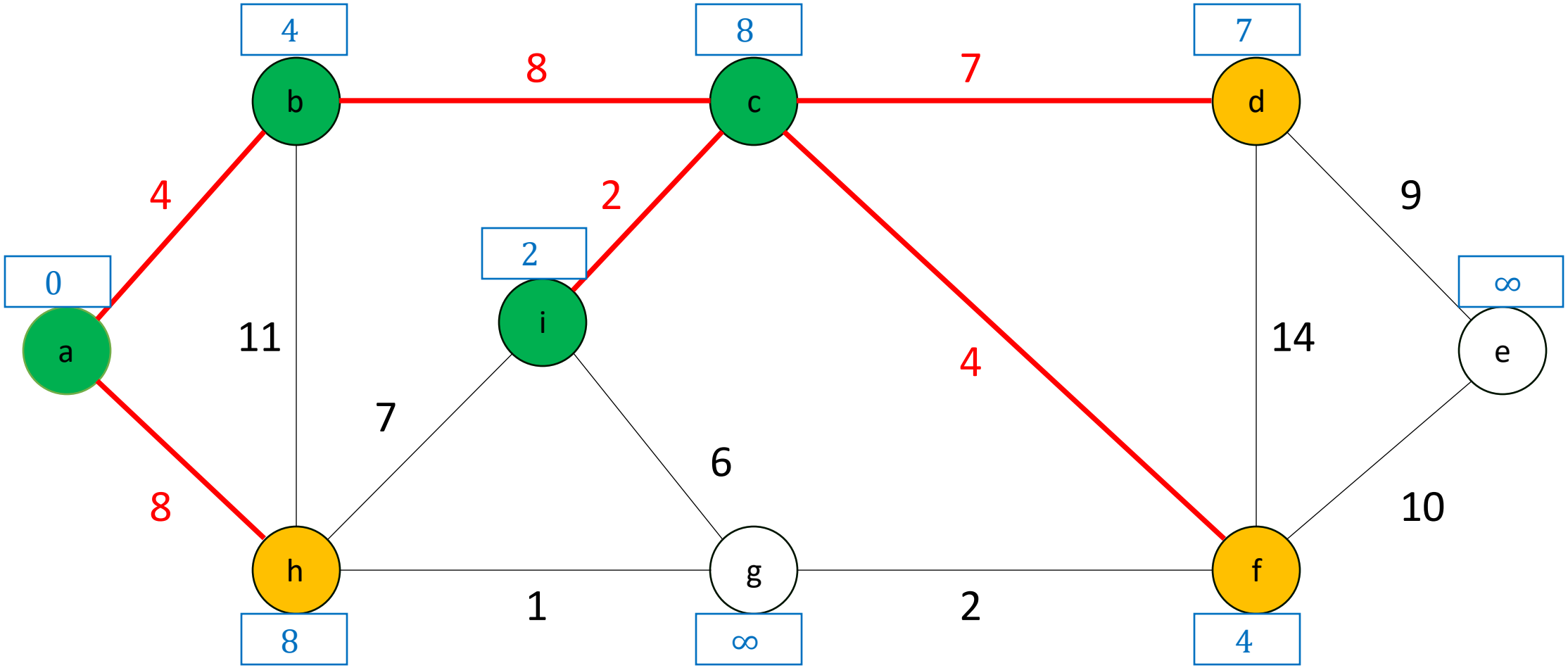
Prim's Algorithm Example



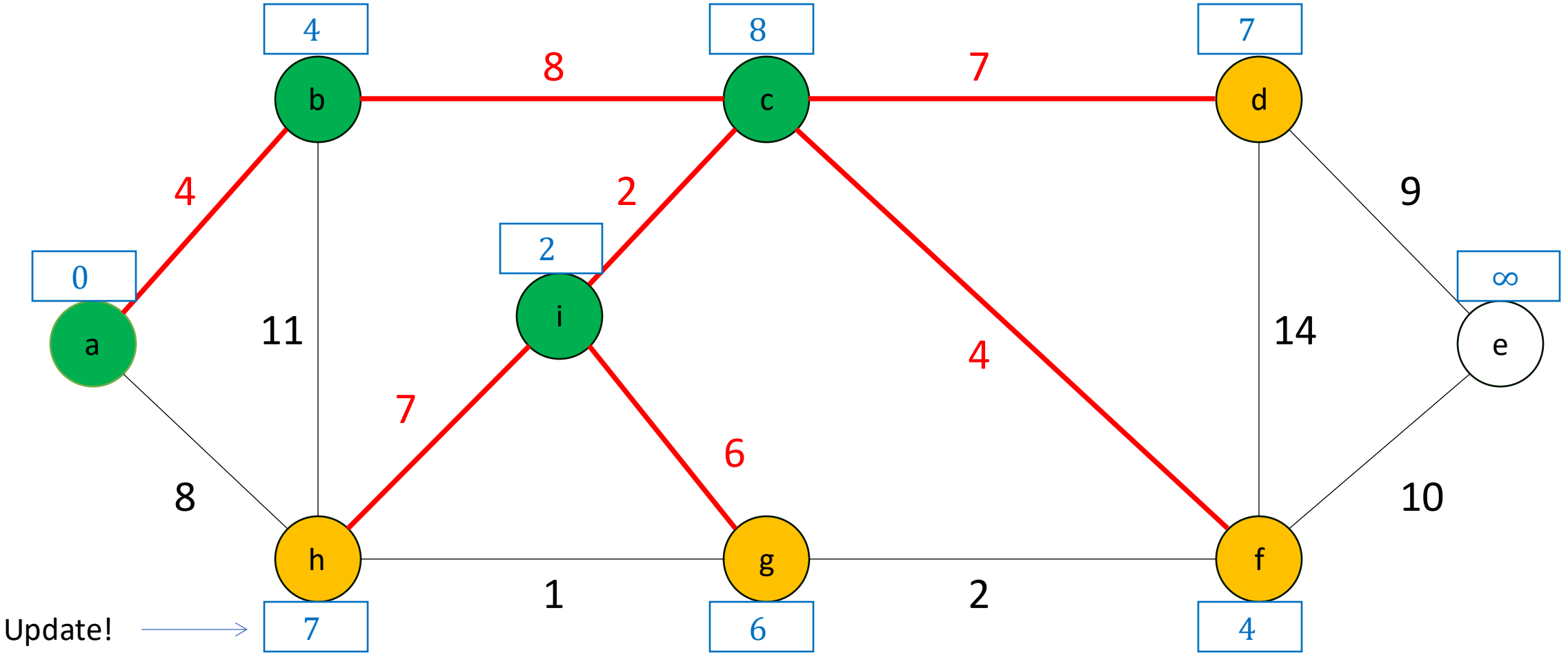
Prim's Algorithm Example



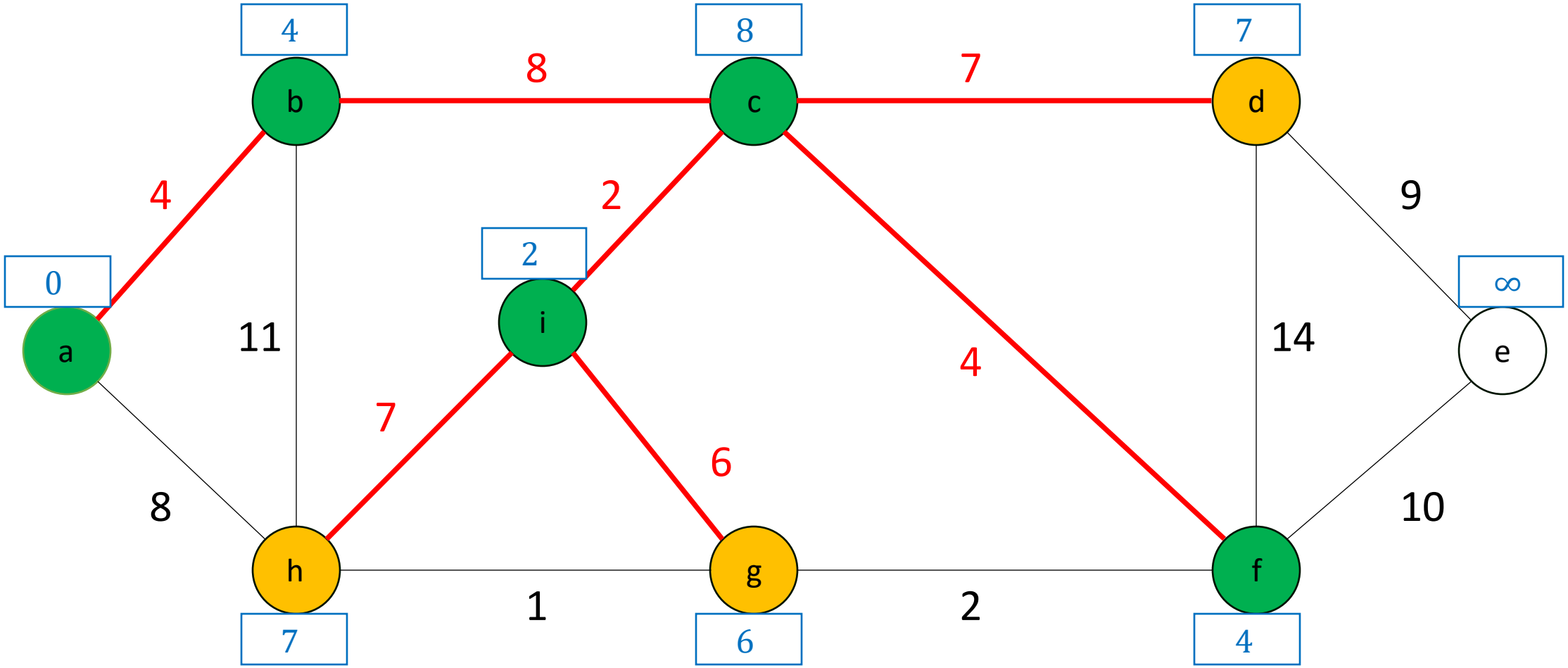
Prim's Algorithm Example



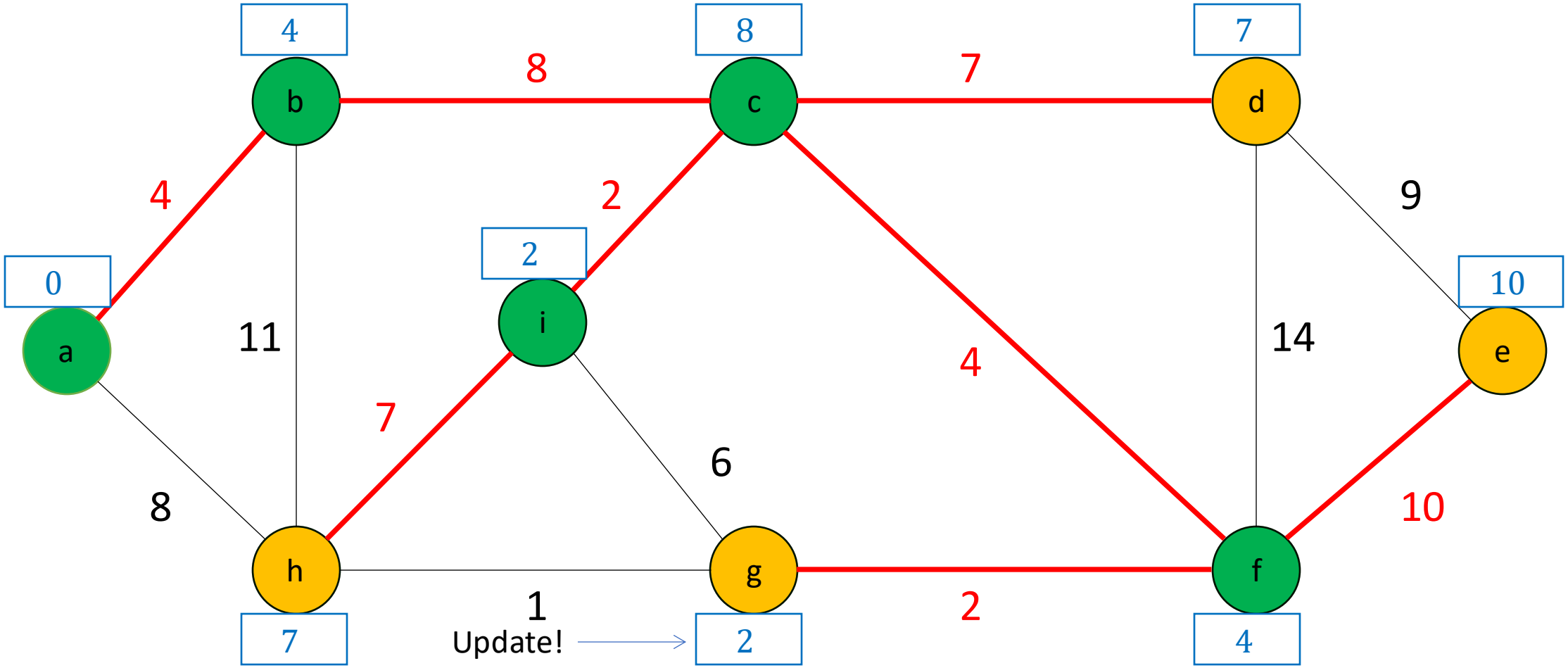
Prim's Algorithm Example



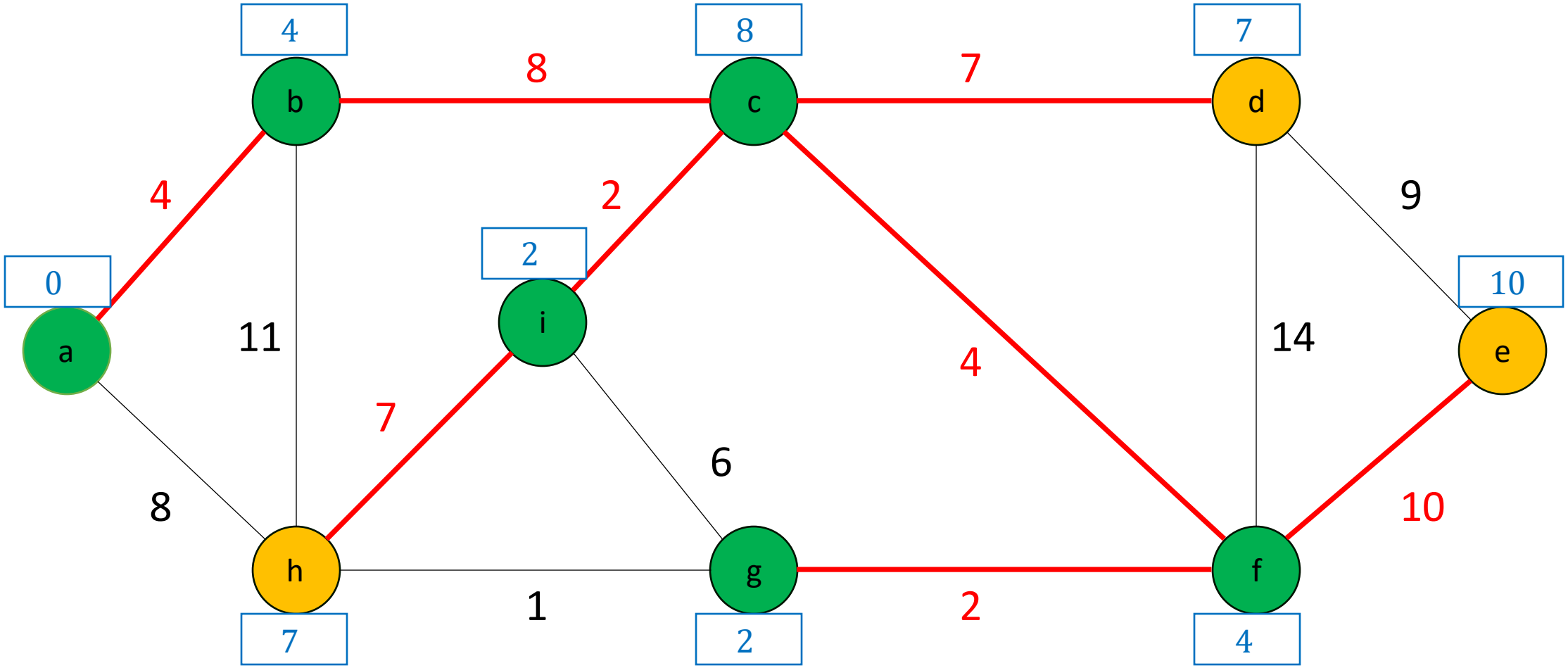
Prim's Algorithm Example



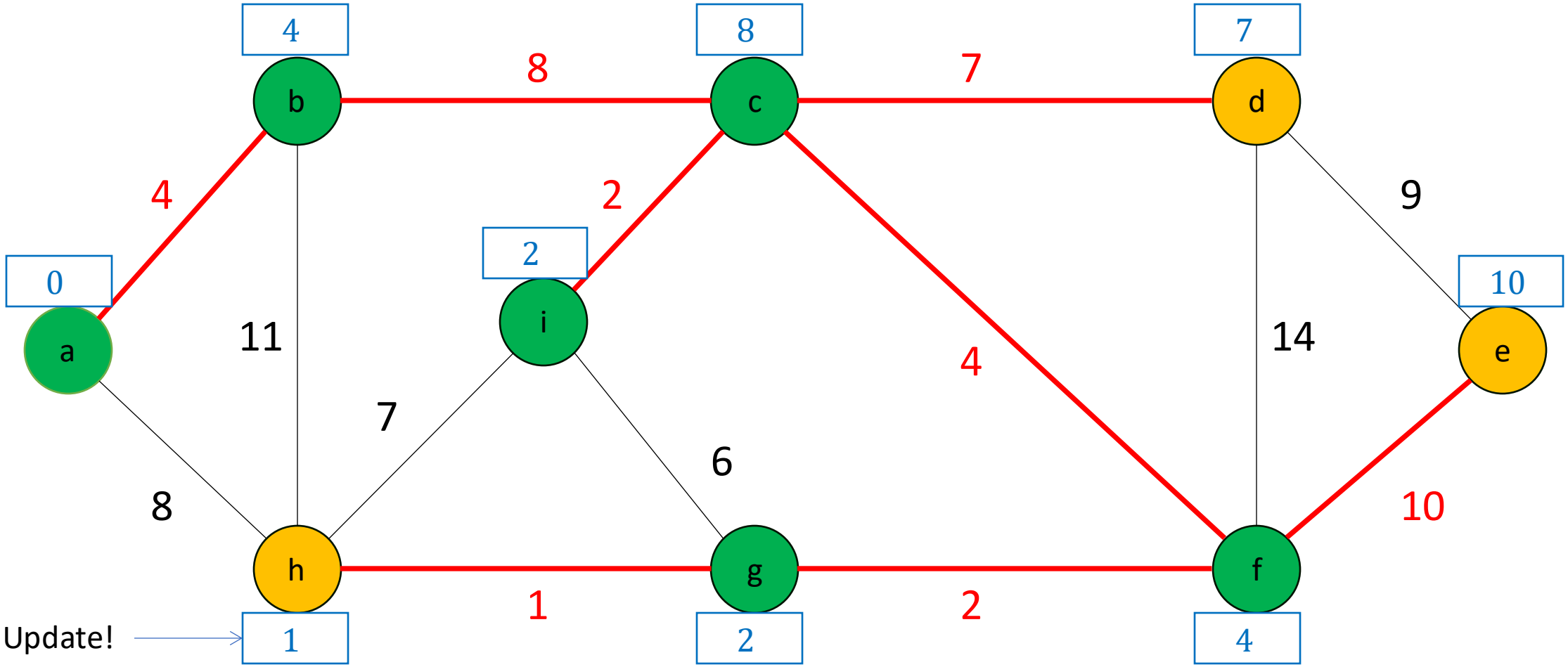
Prim's Algorithm Example



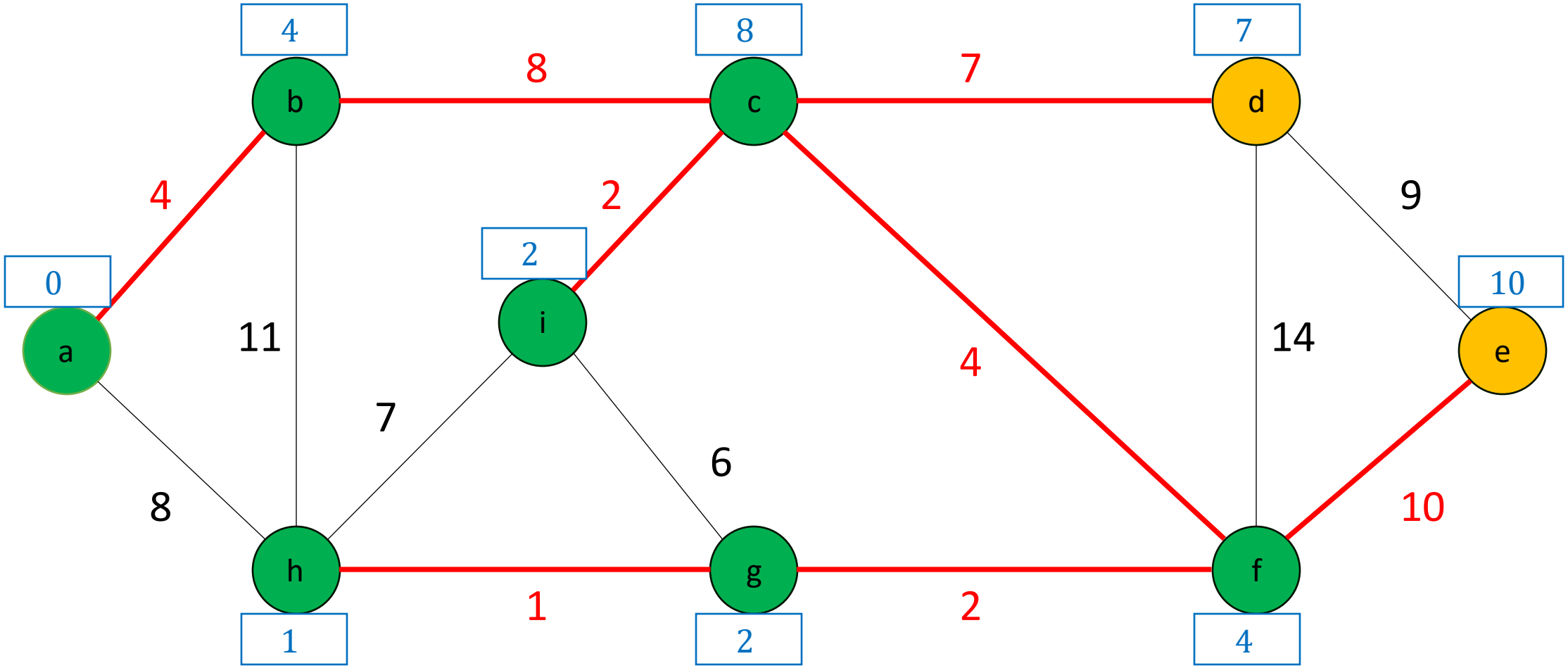
Prim's Algorithm Example



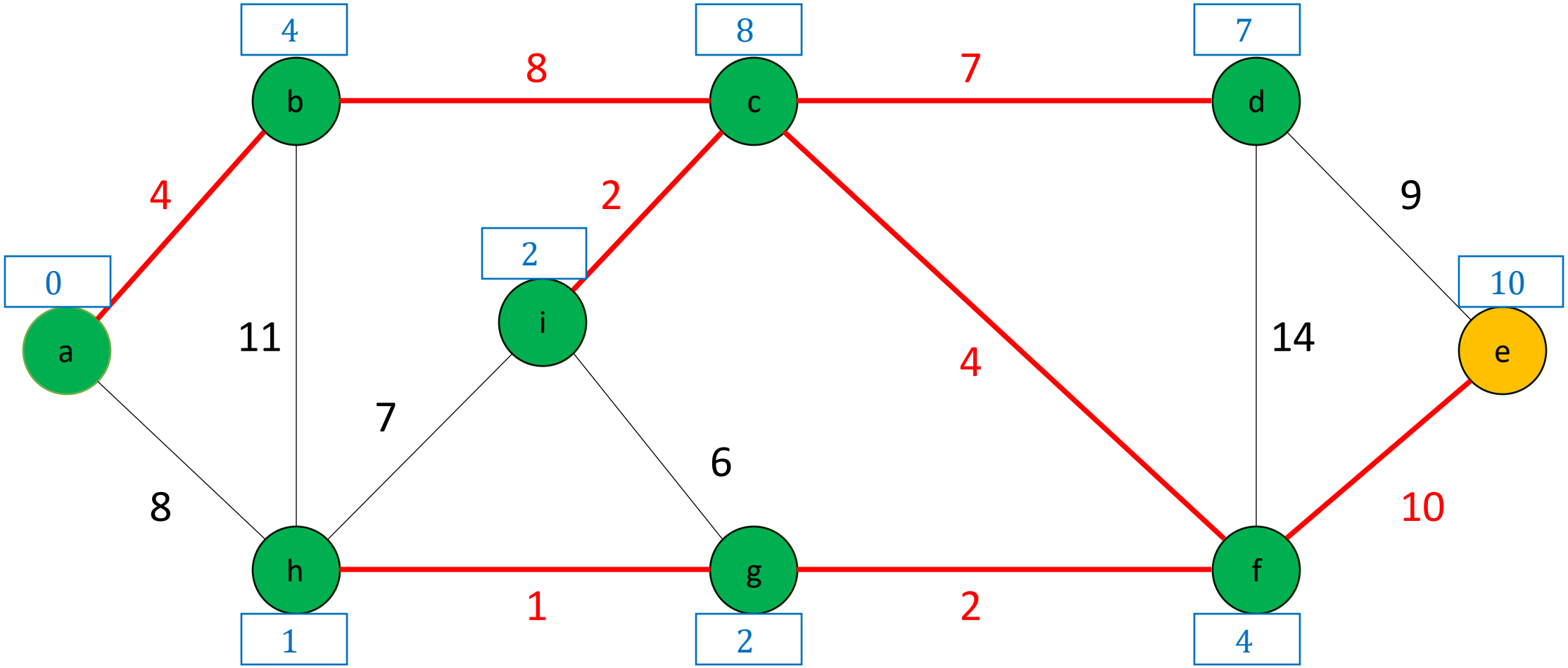
Prim's Algorithm Example



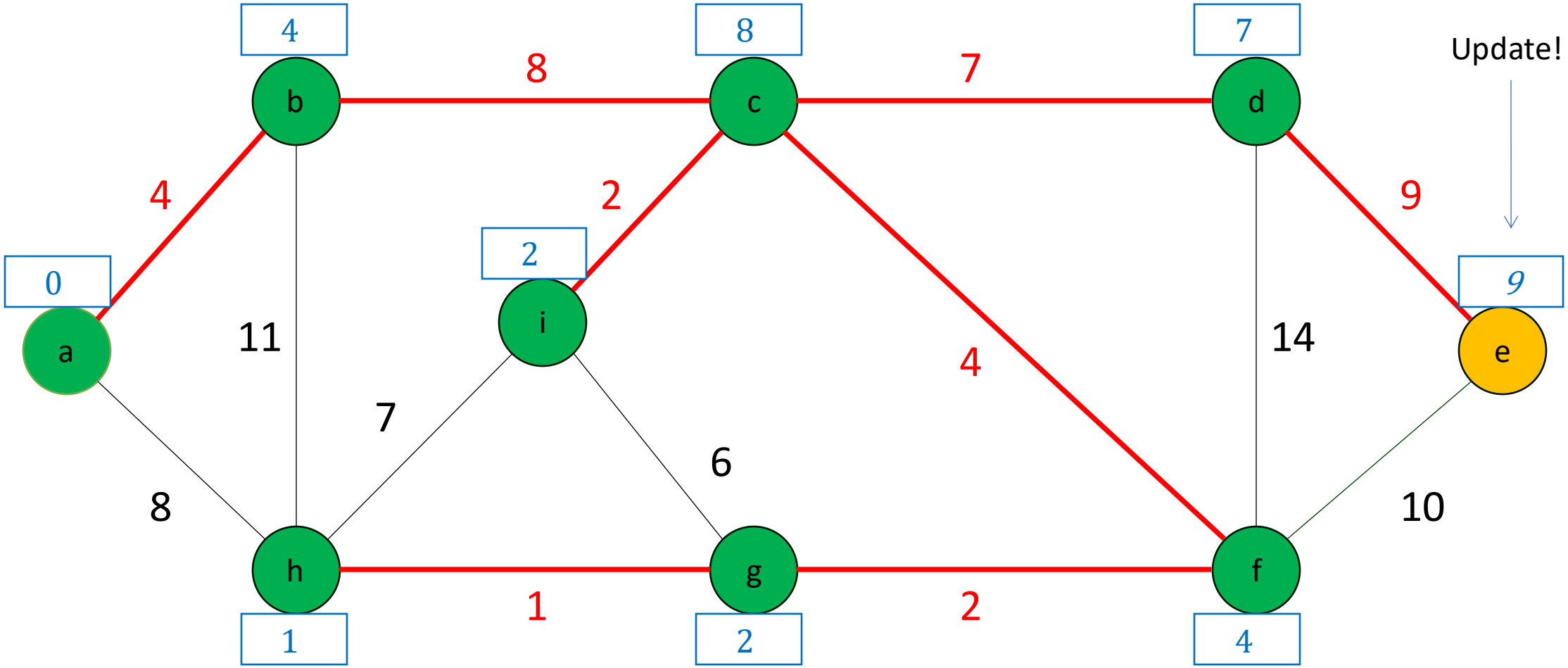
Prim's Algorithm Example



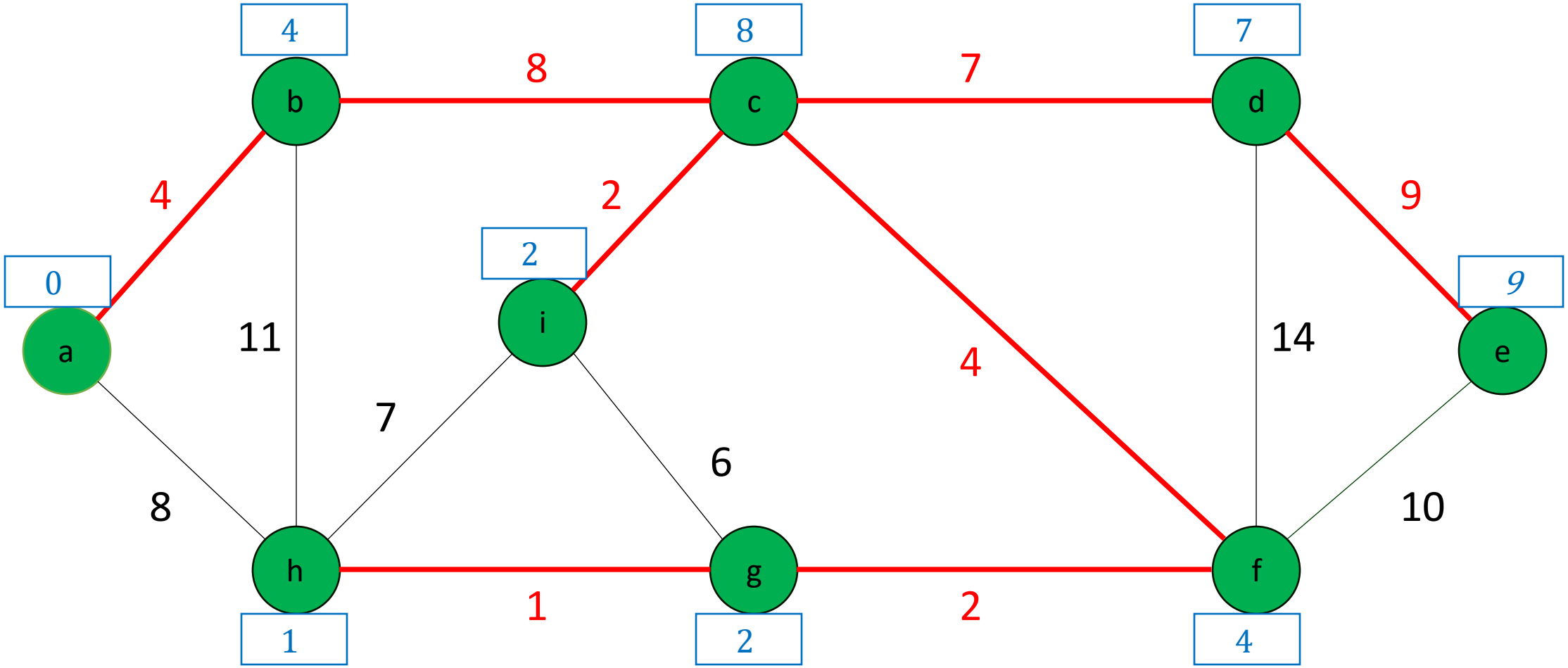
Prim's Algorithm Example



Prim's Algorithm Example



Prim's Algorithm Example



Pseudo Code

```
MST-PRIM(G, w, r)
1. for each vertex u ∈ G.V                    O(V)
2.     u.key = ∞
3.     u.π = NIL
4.  r.key = 0
5.  Q = ∅
6. for each vertex u ∈ G.V                    O(VlogV)
7.     INSERT(Q, u)
8. while Q ≠ ∅                                V iterations
9.     u = EXTRACT − MIN(Q)                   O(logV)
10.    for each vertex v ∈ G.Adj[u]           |u.edges| iterations
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))  O(logV)
```

- The total time: $O(V \log V + \sum_{u \in V} |u.edges| \log V) = O((E + V) \log V)$

Source: Introduction to algorithms, edition four

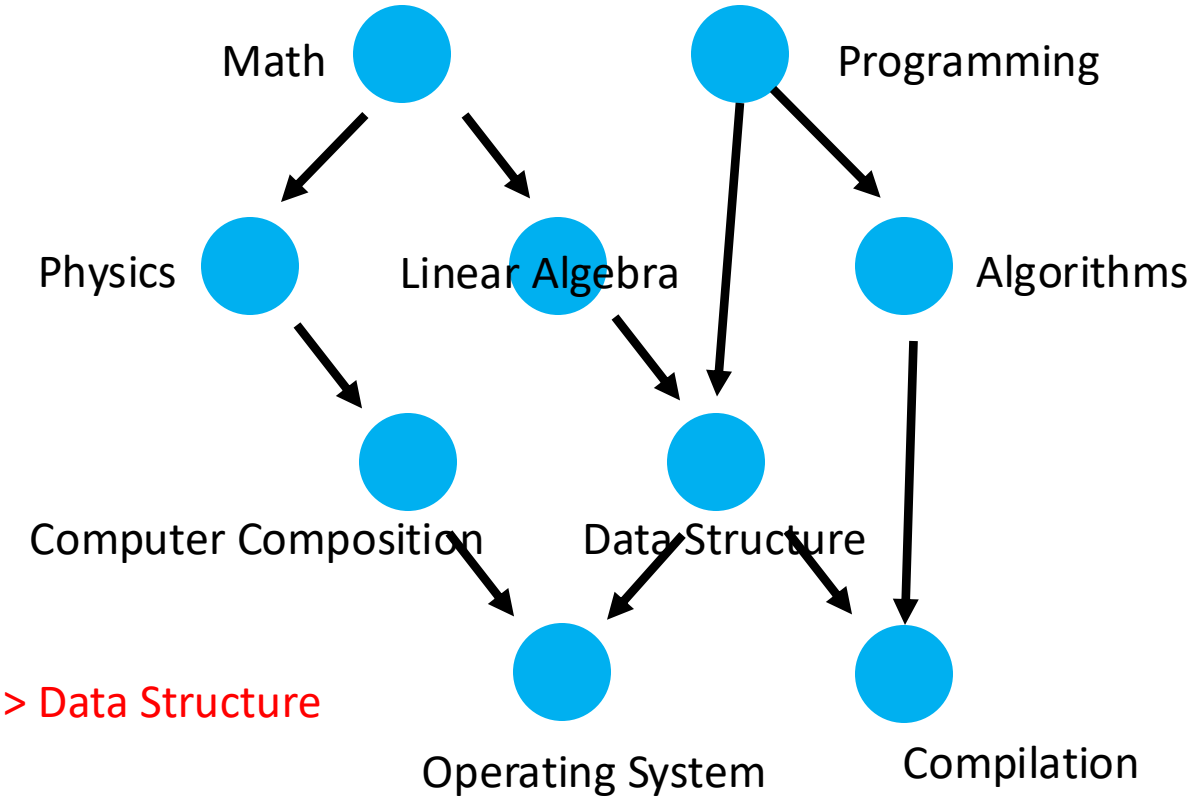
Topological Sort

Topological Sort

Given a directed acyclic graph $G =< V, E >$. Find an order $O =< o_1, o_2, \ldots, o_V >$ satisfying there does not exist pair $< i, j >$ that $i < j$ and $< o_j, o_i > \in E$

Example:

- Course Selection
- Manufacturing Workflows



Intuitive understanding

- Find an order of doing things that respect the given pair-wise dependencies

Math > Physics > Programming > Linear Algebra > Algorithms > Data Structure > Computer Composition > Operating System > Compilation

Optimal Substructure

Given a topological sort $O =< o_1, o_2, \ldots, o_v >$ for graph $G =< V, E >$, deleting o_1 generates a topological sort of a graph $G' =< V', E' >$, where

$$V' = V - \{o_1\},$$

$$E' = \{< u', v' > | < u', v' > \in E, u' ! = o_1, v' ! = o_1\}$$

Proof: obvious, for the original order, there is no $< i, j >$ that $i < j$ and $< o_j, o_i > \in E$, deleting a node does not change of pair-wise relationship of the rest of the nodes

Recurrence for Optimized Value

Given a topological sort $O =< o_1, o_2, \dots, o_v >$ for graph $G =< V, E >$, deleting o_1 generates a topological sort of a graph $G' =< V', E' >$, where

$$V' = V - \{o_1\},$$

$$E' = \{< u', v' > | < u', v' > \in E, u'! = o_1, v'! = o_1\}$$

Recurrence: $f(G)$ - whether you can find a topological sort for G

$$f(G) = \max_{u \in V} f\big(g(G,u)\big)\mathbb{I}\{|\{< v,u > | v \in V - \{u\}, < v,u > \in E\}| = 0\}$$

Complexity: exponential subproblems (subgraphs)

Greedy Intuition: choose a node that does not have precedents

Is it optimal?

—— For any node without precedents, there exists a topological order starting with it

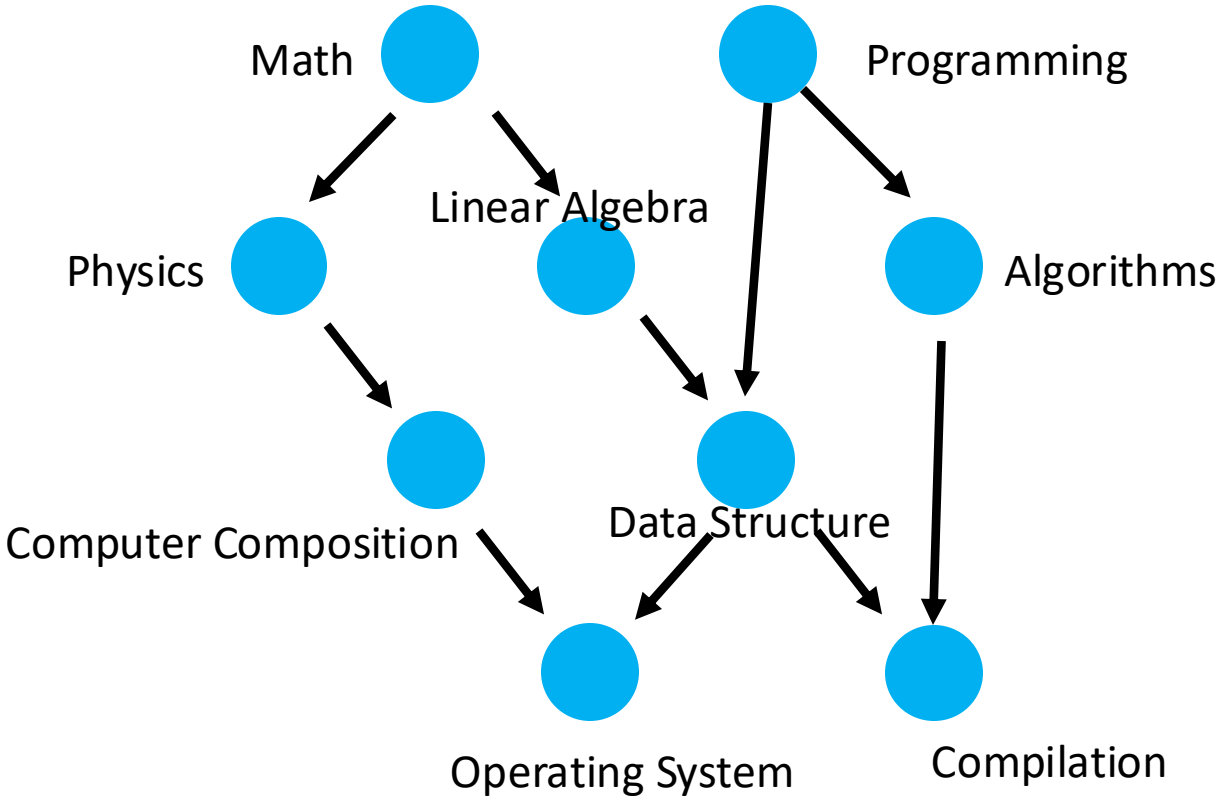
Greedy Choice Property

For any node (u) without precedents, there exists a topological order starting with it

Proof:

- Given any topological sort $O =< o_1, \ldots o_k, u, o_{k+2}, \ldots o_n >$ where u does not appear as the first, $O' =< u, o_1, \ldots o_k, o_{k+2}, \ldots o_n >$ is a topological sort

Kahn Algorithm



Math Physics Programming Linear Algebra Algorithms Data Structure Computer Composition Operating System Compilation

Kahn Algorithm

Greedily delete a node that has no predecessor (in-degree is 0)

1. Record the in-degree of each node
2. Pick the node whose in-degree is 0 (delete and put to the list)
3. Update the in-degree of the successors
4. Repeat 2-3 until all the nodes have been picked

```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
ordering := {}
while Q is not empty:
    u = Q.pop()
    ordering.append(v)
    for v ∈ {v'| < u, v' >∈  E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
```

Complexity

- Each node will be inserted to the queue once and deleted once – $O(V)$
- Edges from a node are traversed when it is deleted: $O(E)$ in total
- Total time complexity: $O(V + E)$
- Auxiliary Space complexity: $O(V)$

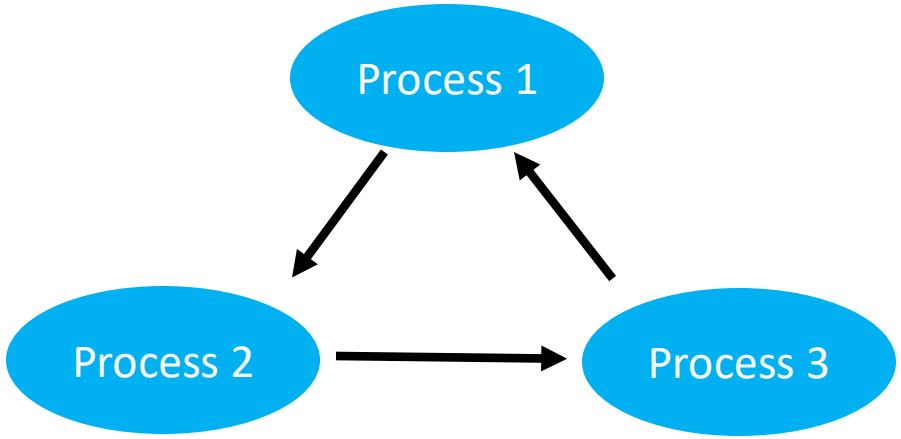
```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
ordering := {}
while Q is not empty:
    u = Q.pop()
    ordering.append(v)
    for v ∈ {v'| < u, v' >∈ E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
```

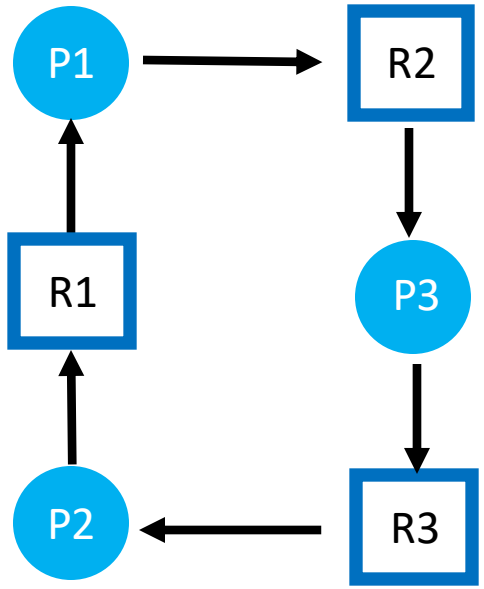
Extension: Cycle Detection

Given a directed graph $G =< V, E >$, detect whether there is cycle in G

- Example applications: detecting deadlocks in Operating Systems
- Solution: a graph with cycle is non-empty when the topological sort algorithm stops

```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
while Q is not empty:
    u = Q.pop()
    for v ∈ {v'| < u, v' >∈ E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
for v in V:
    if deg[v] != 0: return "has cycle!"
return "no cycle."
```





Resource Allocation Graph

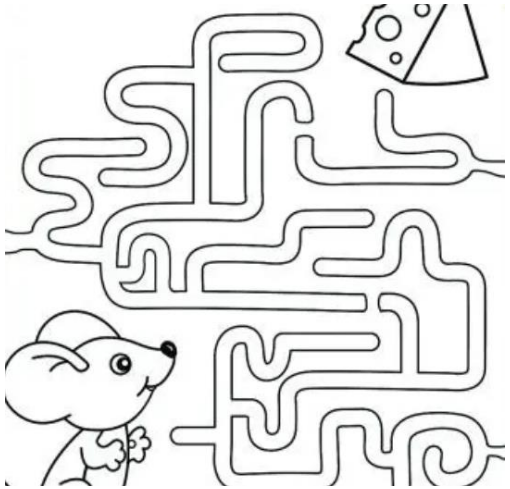
Basic Graph Search Algorithms

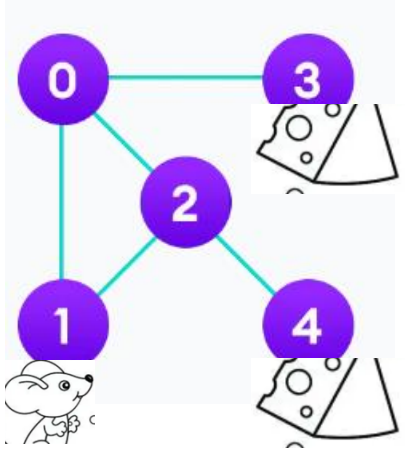
Graph Search

Problem: Given a maze as a graph $G = < V, E >$, where node $v \in V$ means rooms and edges $e \in E$ means connection between rooms, some room contain some cheese $c[v]$. Given a starting point s of the mouse , what is the maximum number of cheese the mouse can get?

Graph search/traversal algorithms

- Visit edges and nodes for general discovery or explicit search
- Commonly used in graph algorithms



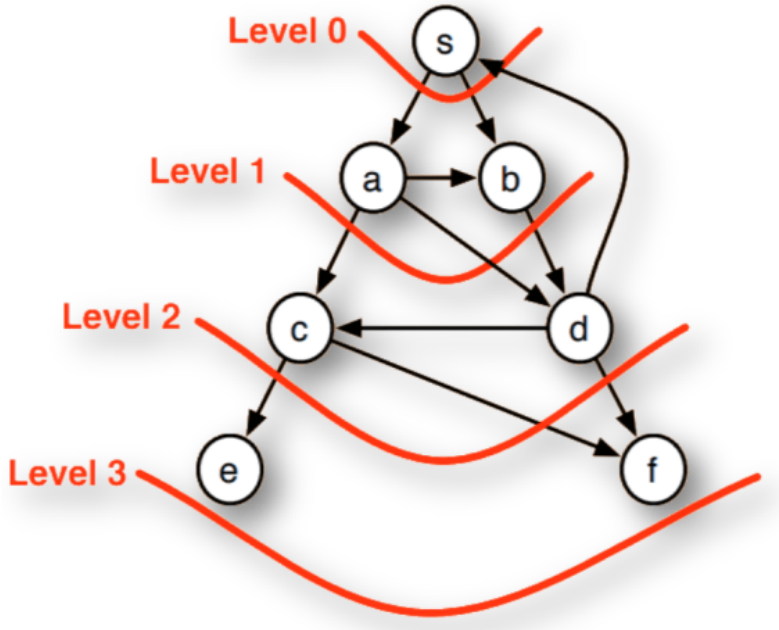


Breadth-First Search

Search a graph data structure for a node

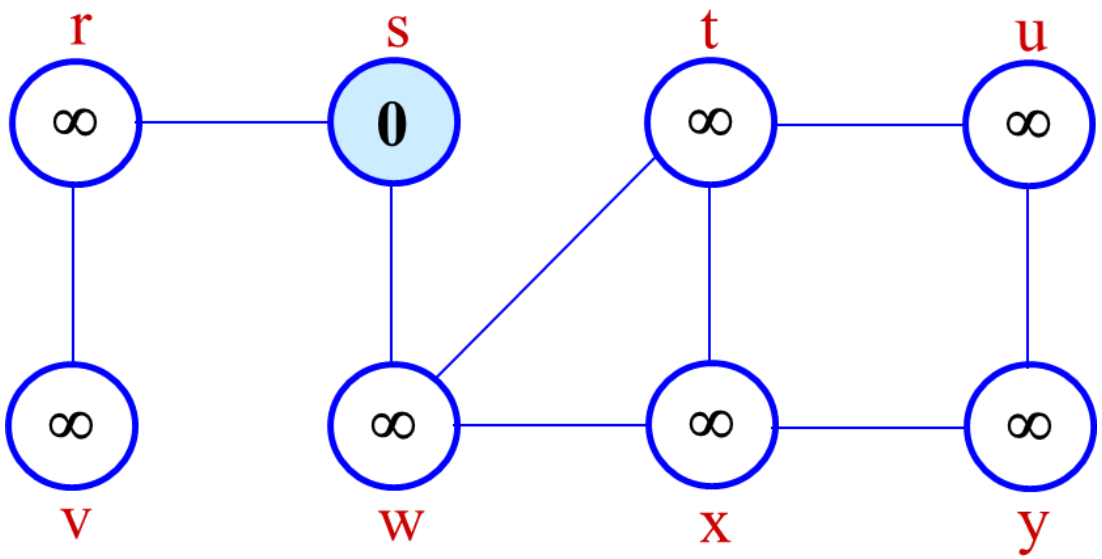
- Starts at the root of the graph
- Breadth-first: finish visiting all nodes you can reach now before moving to the next level

Example: grab the general ideas of various knowledge before going deep into any of them



Breadth-First Search

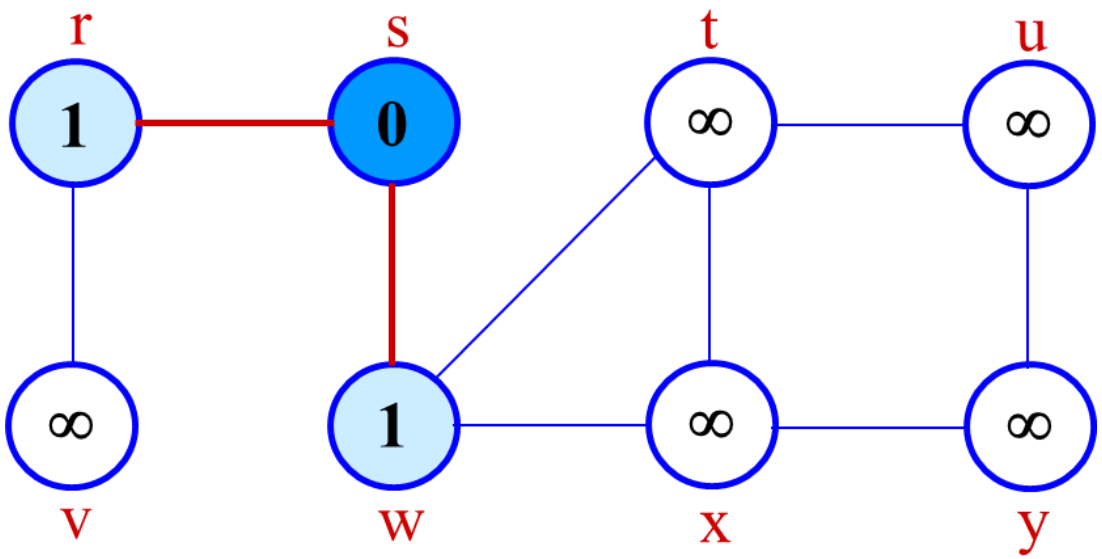
- Track of the nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q: s
0

Breadth-First Search

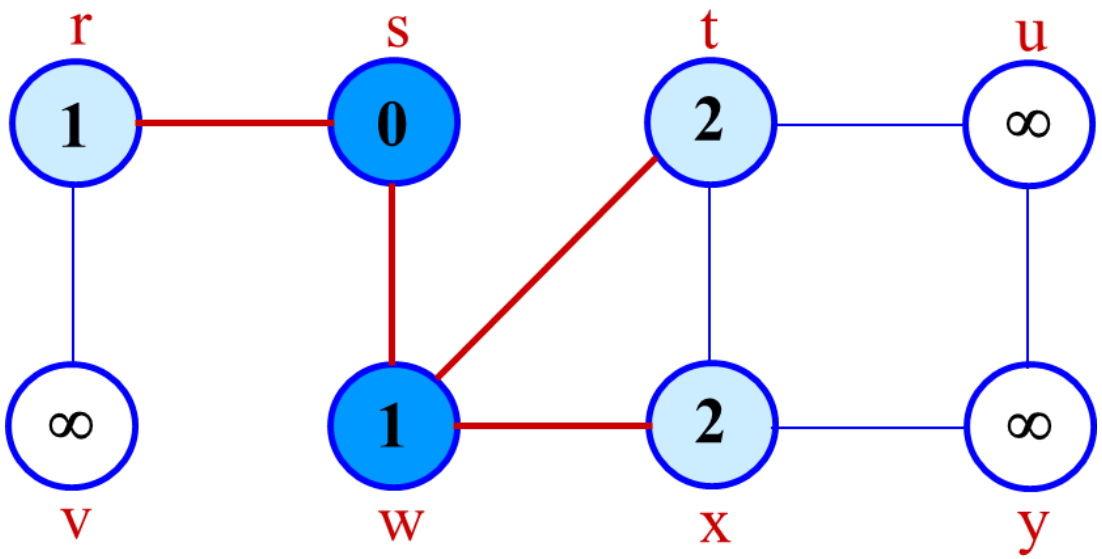
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	w	r
	1	1

Breadth-First Search

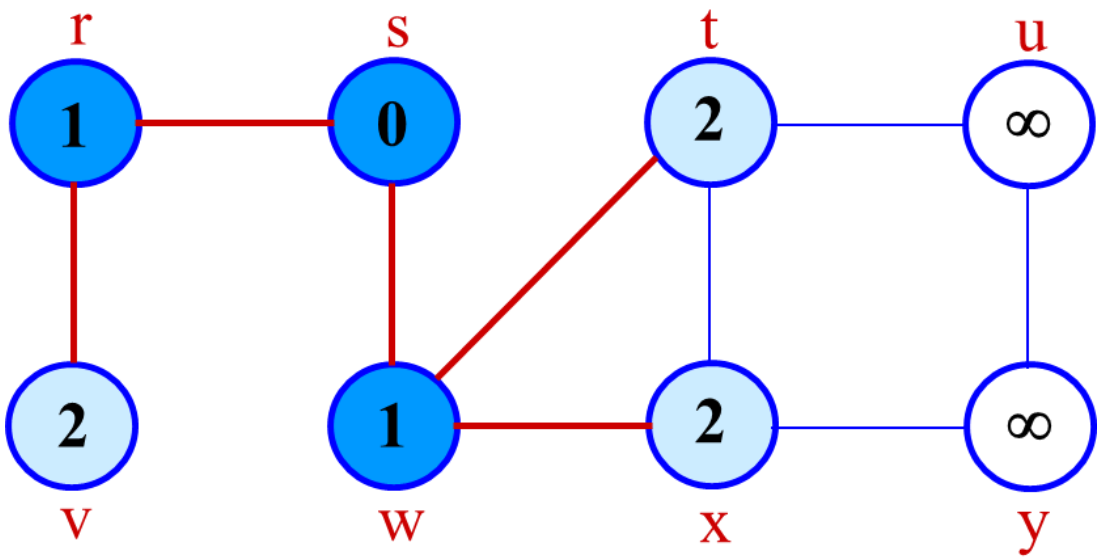
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	r	t	x
	1	2	2

Breadth-First Search

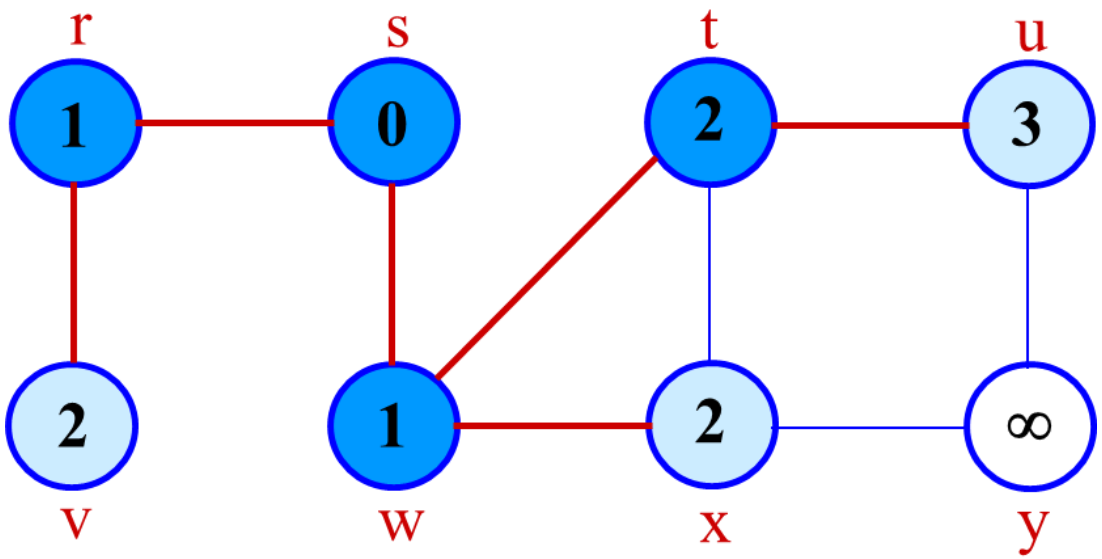
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	t	x	v
	2	2	2

Breadth-First Search

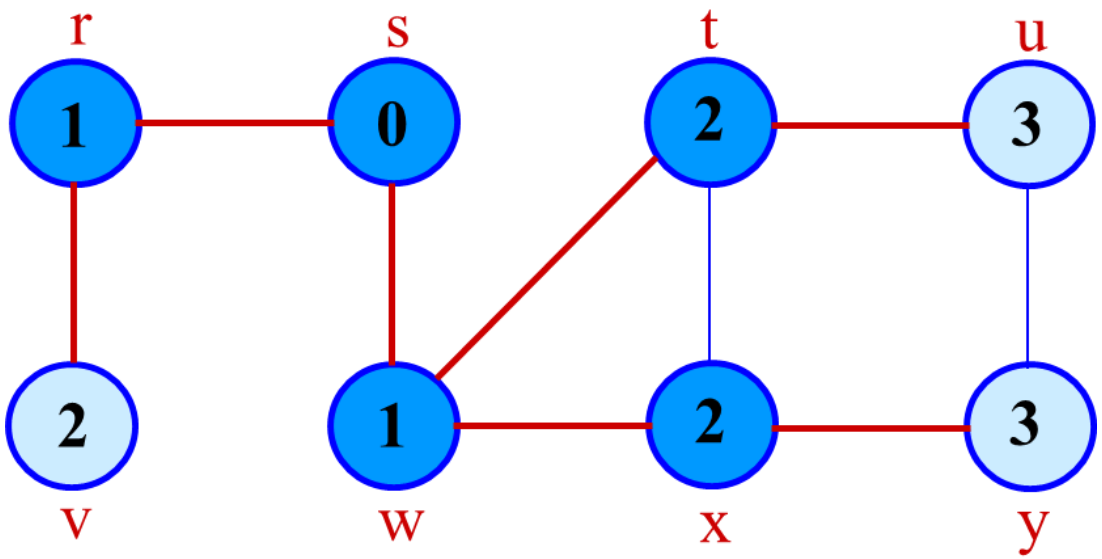
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	x	v	u
	2	2	3

Breadth-First Search

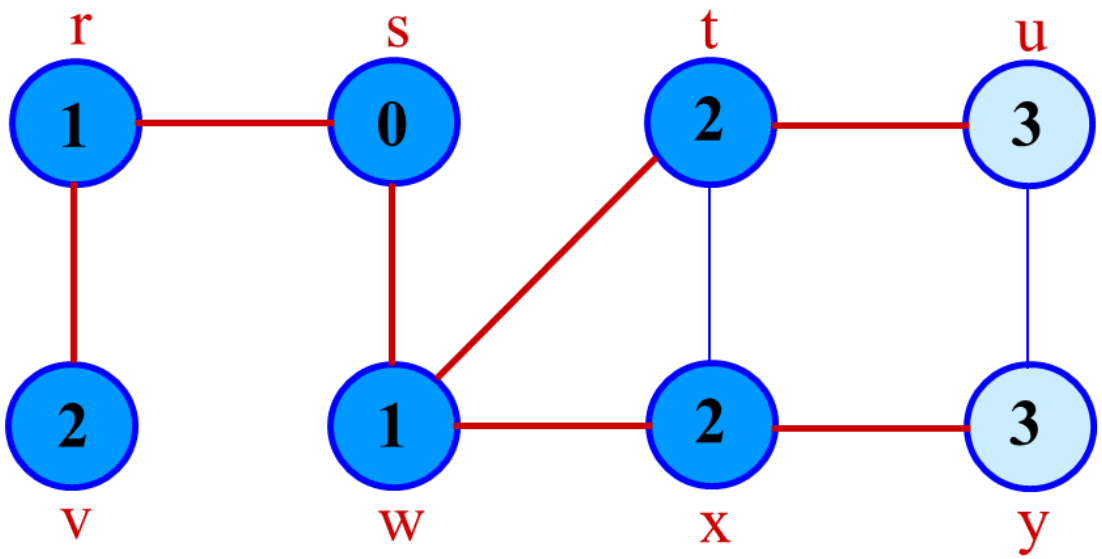
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	v	u	y
	2	3	3

Breadth-First Search

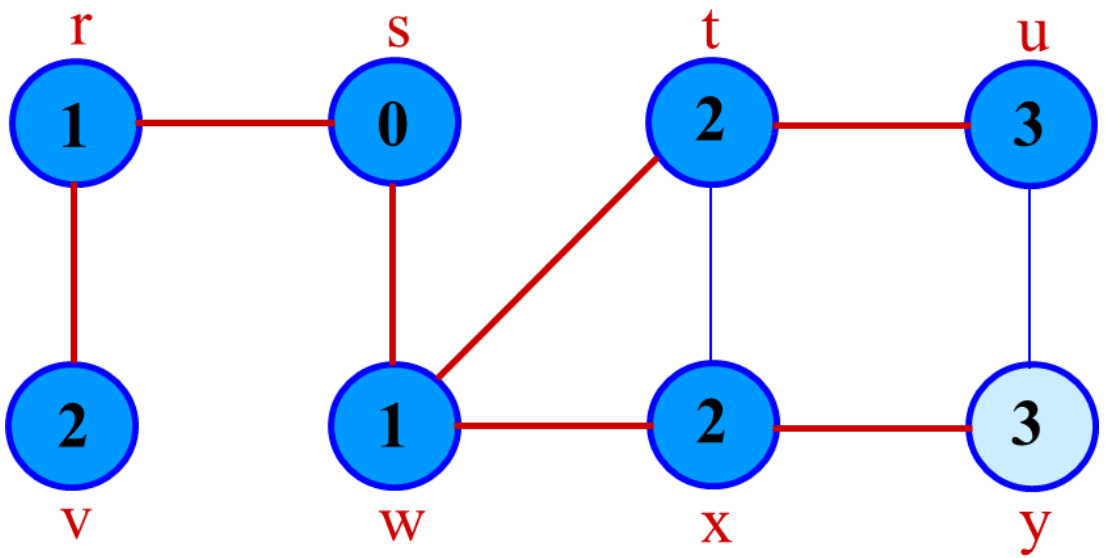
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	u	y
	3	3

Breadth-First Search

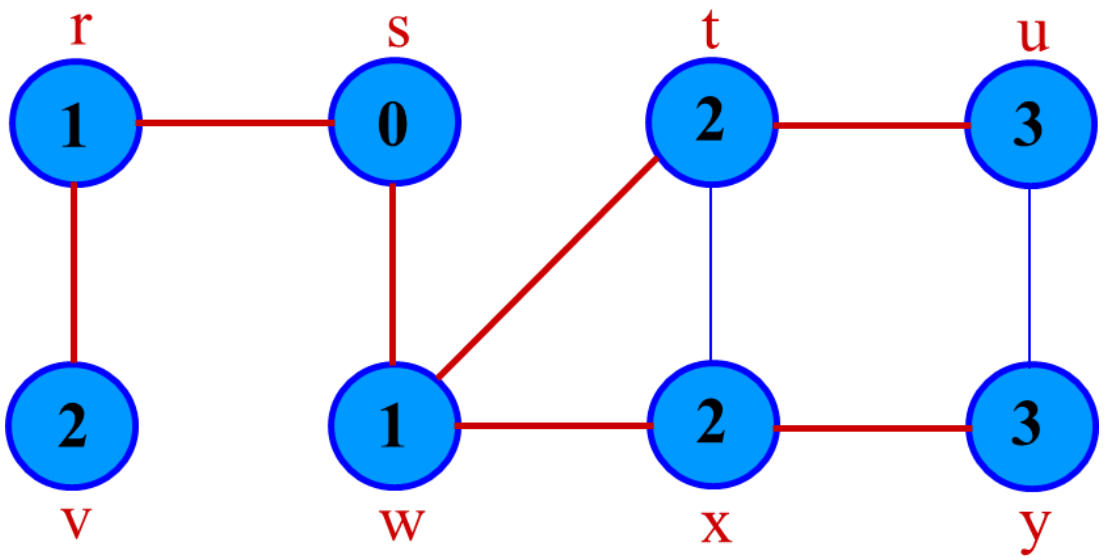
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	y
	3

Breadth-First Search

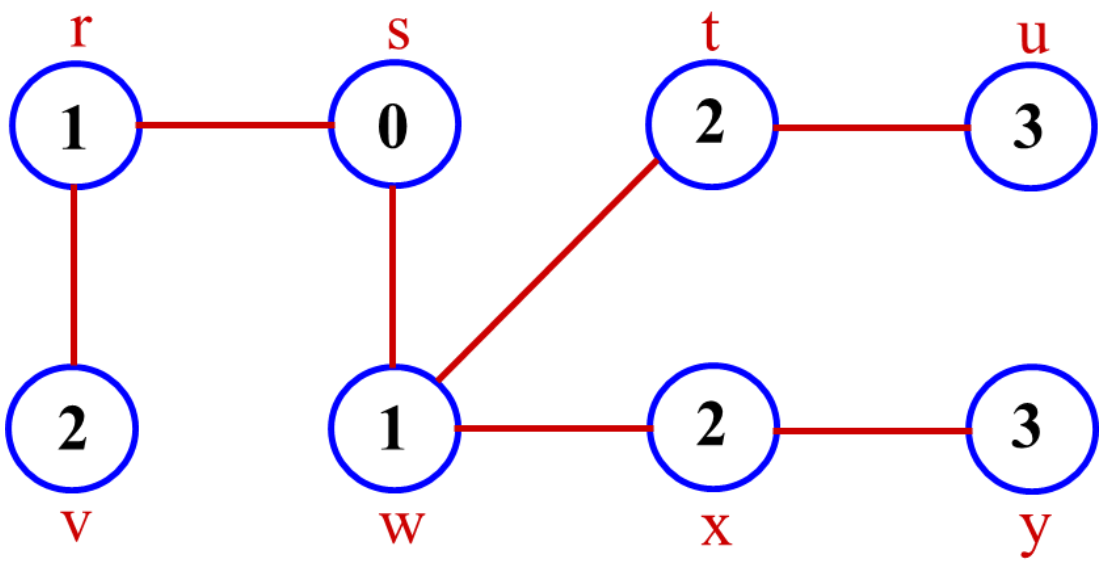
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped

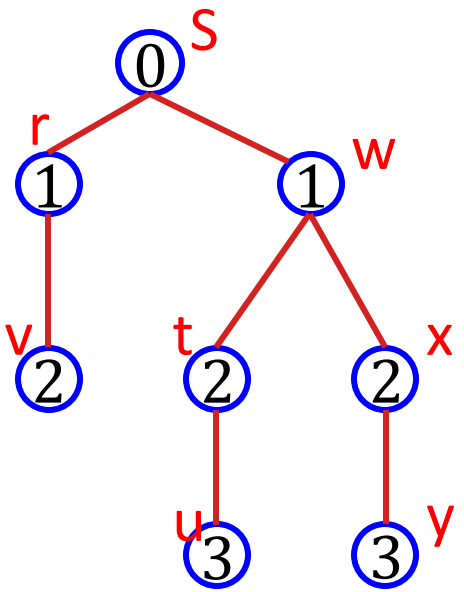


Q: ∅

Breadth-First Search

Produces a “breadth-first tree” with root s that contains all reachable vertices





BF Tree

Breadth-First Search

Notations

- $v.d$: the depth of v in the bfs tree
- $v.\pi$: the parent node of v in the bfs tree

Time Complexity

- **Initialization:** $O(V)$
- **Traversal Loop**
  - Vertex: enqueued and dequeued at most once, $O(V)$
  - Edge: Edges from each node traversed once, $O(E)$
- **Total:** $\boldsymbol{O(V+E)}$

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Breadth-First Search

Theorem (Correctness of breadth-first search)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Notations

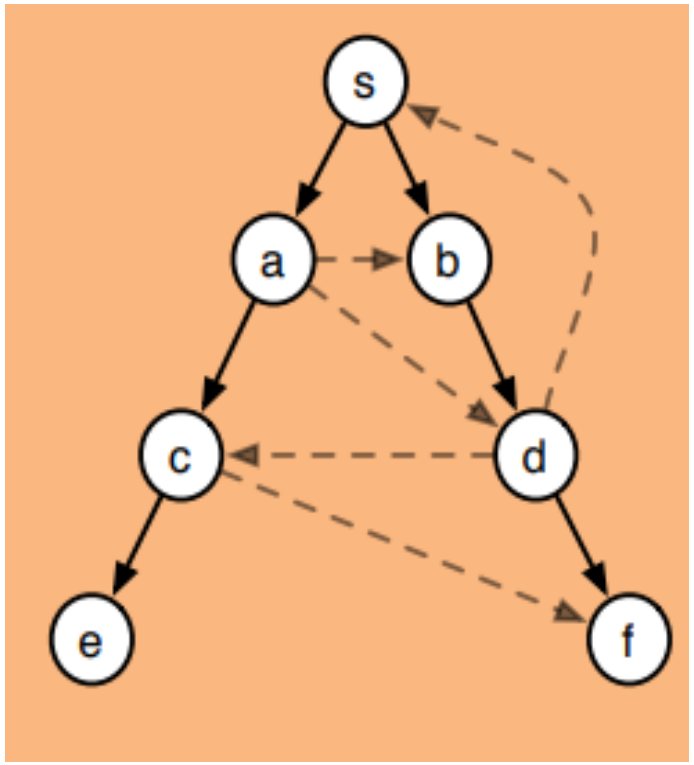
- $\delta(s, v)$: the shortest path distance from s to v (minimum number of edges)
  - $\delta(s, v) = \infty$ if no path exists

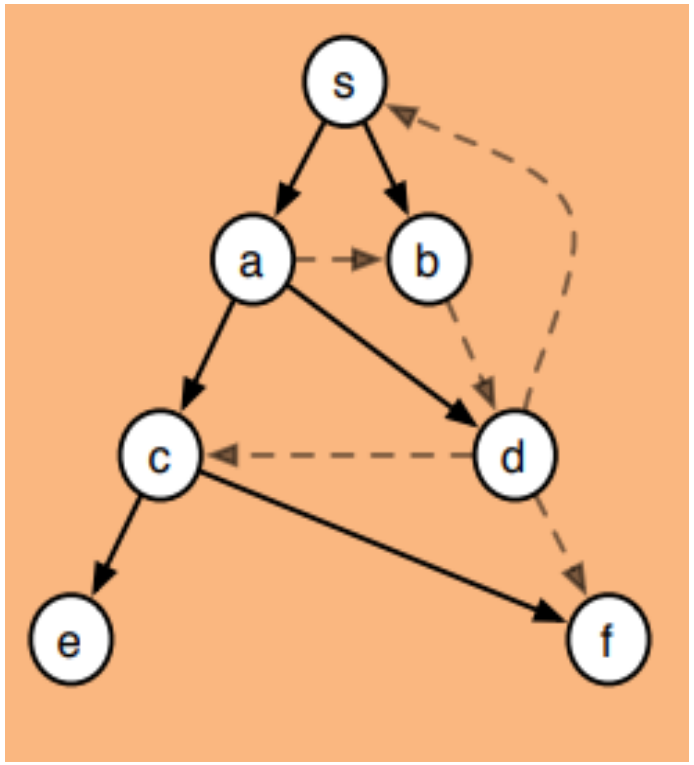
Proof: mathematical induction, $\delta(s, v) = 1$ then $v.d = 1$, $\delta(s, v) = 2$ then $v.d = 2$, …

Breadth-First Search

Each path from source to a vertex is the shortest path between them on the graph

Example: an undirected graph and two possible BFS trees with distances from s





Thank you!

AIAA 5037 Advanced Algorithms and Data Structures