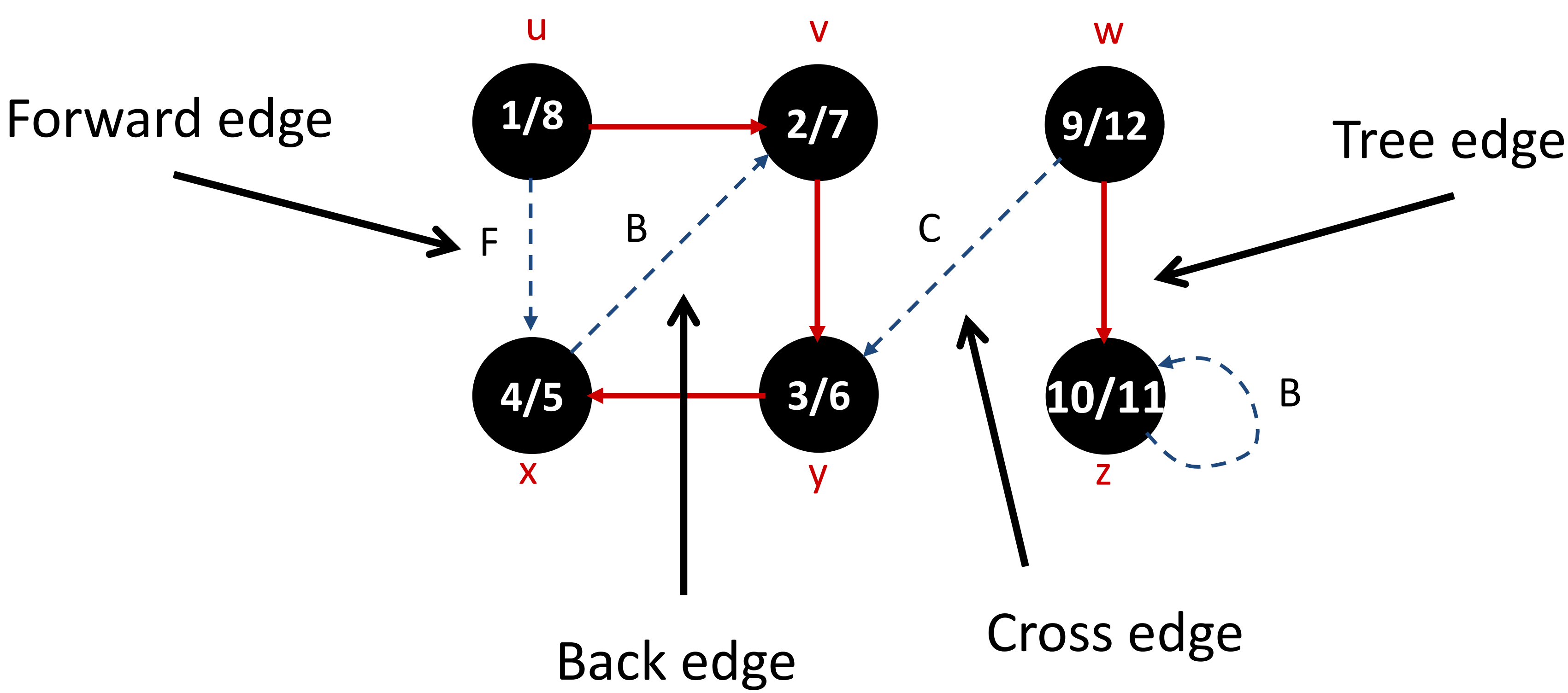
COMP251: Minimum Spanning Trees

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Cormen *et al.*, 2002)

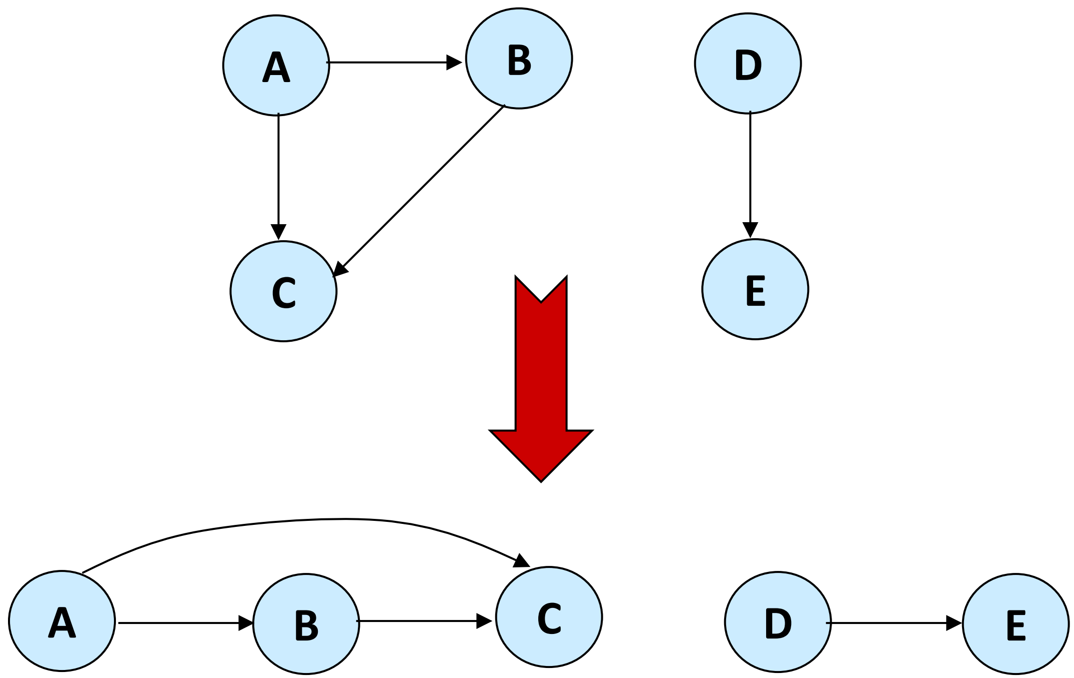
Based on slides from D. Plaisted (UNC)

Recap: Edge Classification



Recap: Topological Sort

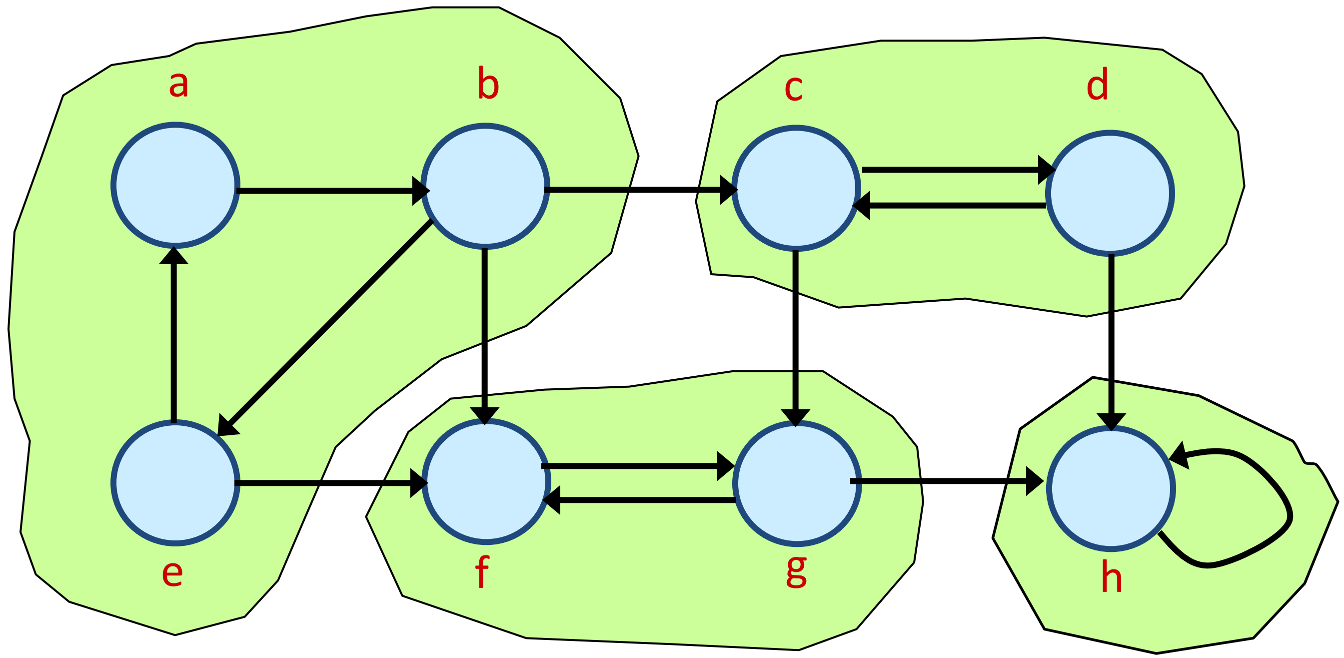
Want to “sort” a directed acyclic graph (DAG).



Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

Recap: Strongly Connected Components



Recap: G^{SCC} is a DAG

Lemma 2
Let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof:

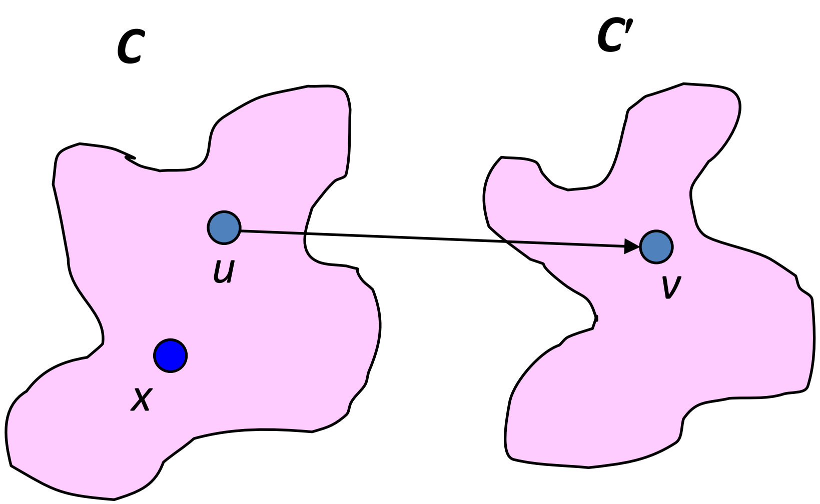
- Suppose there is a path $v' \rightsquigarrow v$ in G.
- Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G.
- Therefore, u and v' are reachable from each other, so they are not in separate SCC's.

Recap: SCCs and DFS finishing times

Lemma 3
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 1: $d(C) < d(C')$
  - Let x be the first vertex discovered in C.
  - At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.
  - By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.
  - By the parenthesis theorem, $f[x] = f(C) > f(C')$.

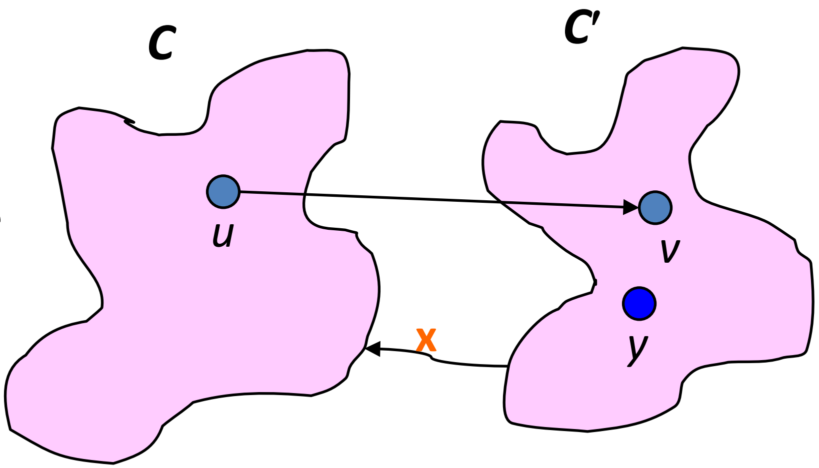


Recap: SCCs and DFS finishing times

Lemma 3
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 2: $d(C) > d(C')$
  - Let y be the first vertex discovered in C'.
  - At $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in C' $\Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.
  - At $d[y]$, all vertices in C are also white.
  - By **lemma 2**, since there is an edge (u, v), we cannot have a path from C' to C.
  - So no vertex in C is reachable from y.
  - Therefore, at time $f[y]$, all vertices in C are still white.
  - Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.



Recap: SCCs and DFS finishing times

Corollary 1
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof:

- $(u, v) \in E^T \Rightarrow (v, u) \in E$.
- SCC's of G and G^T are the same $\Rightarrow$ $f(C') > f(C)$, by Lemma 2.

Recap: Correctness of SCC

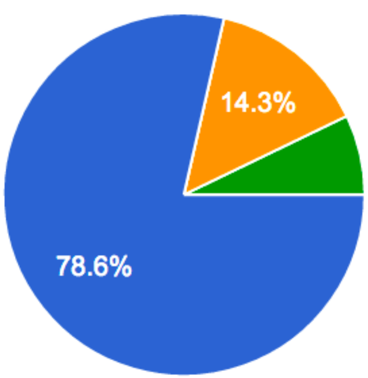
- When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum.
  - The second DFS starts from some $x \in C$, and it visits all vertices in C.
  - Corollary 1 says that since $f(C) > f(C')$ for all $C \neq C'$, there are no edges from C to C' in G^T.
  - Therefore, DFS will visit *only* vertices in C.
  - Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

Recap: Correctness of SCC

- The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C.
  - DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited*.
  - Therefore, the only tree edges will be to vertices in C'.
- We can continue the process.
- Each time we choose a root for the second DFS, it can reach only
  - vertices in its SCC—get tree edges to these,
  - vertices in SCC' s *already visited* in second DFS—get *no* tree edges to these.

Let G be a directed graph. After DFS, we found that G has a back edge.

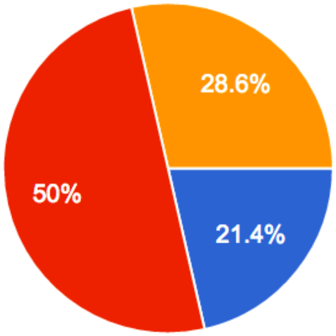
- G has one cycle ✓
- G is a tree ✗
- G is a direct acyclic graph (DAG) ✗
- G is connected ✗



G has one cycle	11	78.6%
G is a tree	0	0%
G is a direct acyclic graph (DAG)	2	14.3%
G is connected	1	7.1%

Let G be a DAG. Let u and v be two vertices of G, such that there is a path from u to v in G. During the execution of topological sort algorithm, we discover u before v.

- v appears before u in the total order. ✗
- v appears after u in the total order. ✓
- we cannot say anything about the order of u and v. ✗



v appears before u in the total order.	3	21.4%
v appears after u in the total order.	7	50%
we cannot say anything about the order of u and v.	4	28.6%

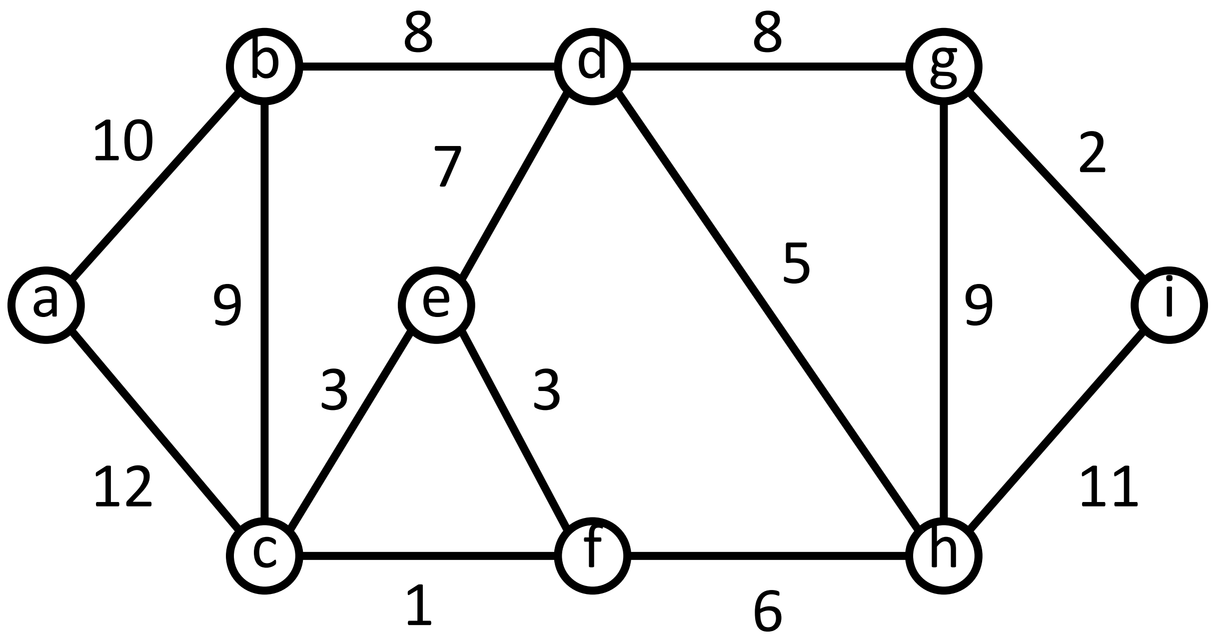
Minimum Spanning Tree (Example)

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses *u* and v has a repair cost w(*u*, v).

Goal: Repair enough (and no more) roads such that:

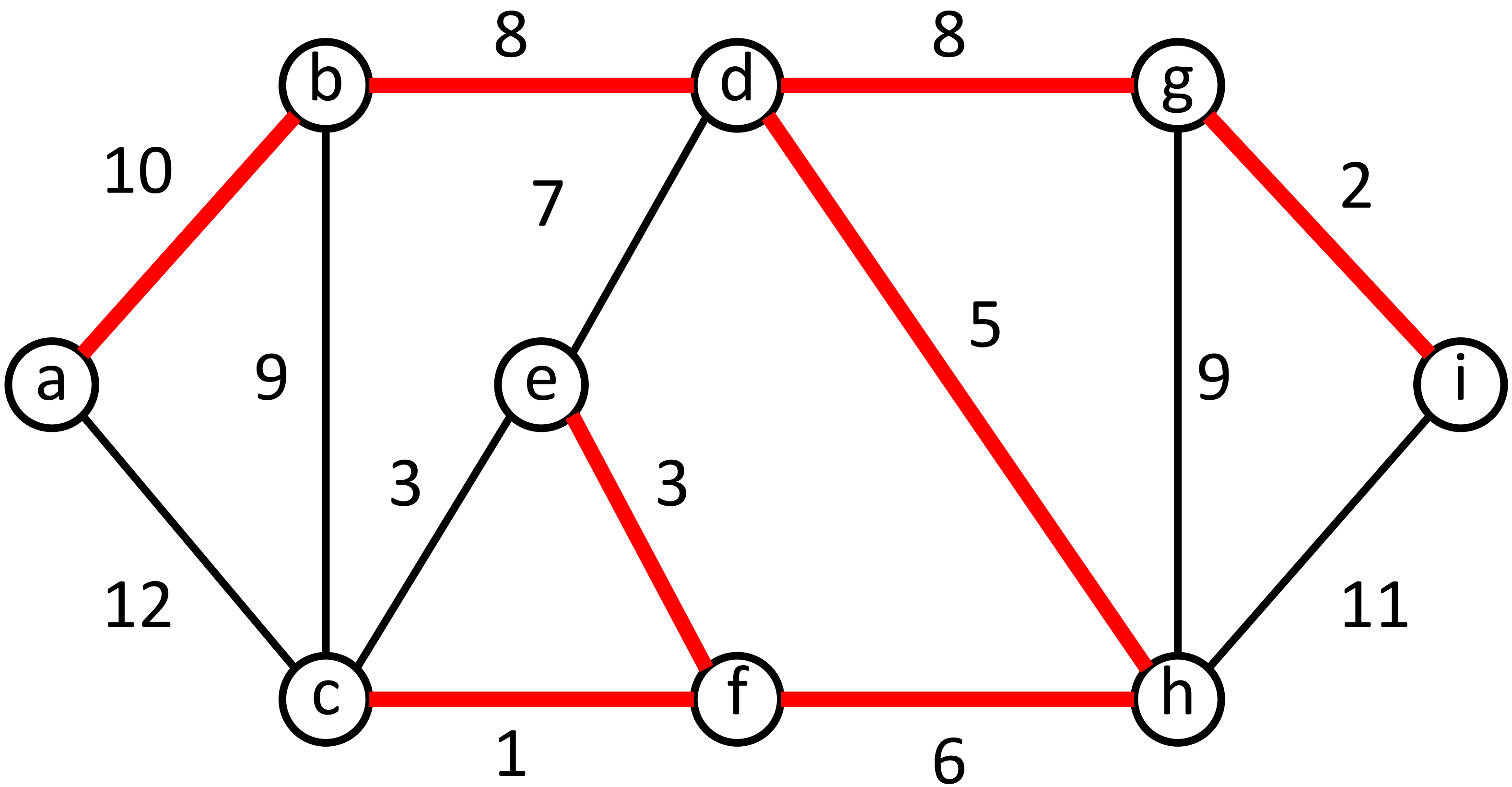
1. everyone stays connected: can reach every house from all other houses, and
2. total repair cost is minimum.

Model as graph



- Undirected graph $G = (V, E)$.
- ***Weight*** w(u, v) on each edge (u, v) $\in$ E.
- Find $T \subseteq E$ such that:
  1. T connects all vertices (T is a ***spanning tree***),
  2. $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.

Minimum Spanning Tree (MST)



- It has $|V| - 1$ edges.
- It has no cycles.
- It might not be unique.

Generic Algorithm

- Initially, A has no edges.
- Add edges to A and maintain the **loop invariant**:
  "A is a subset of some MST".

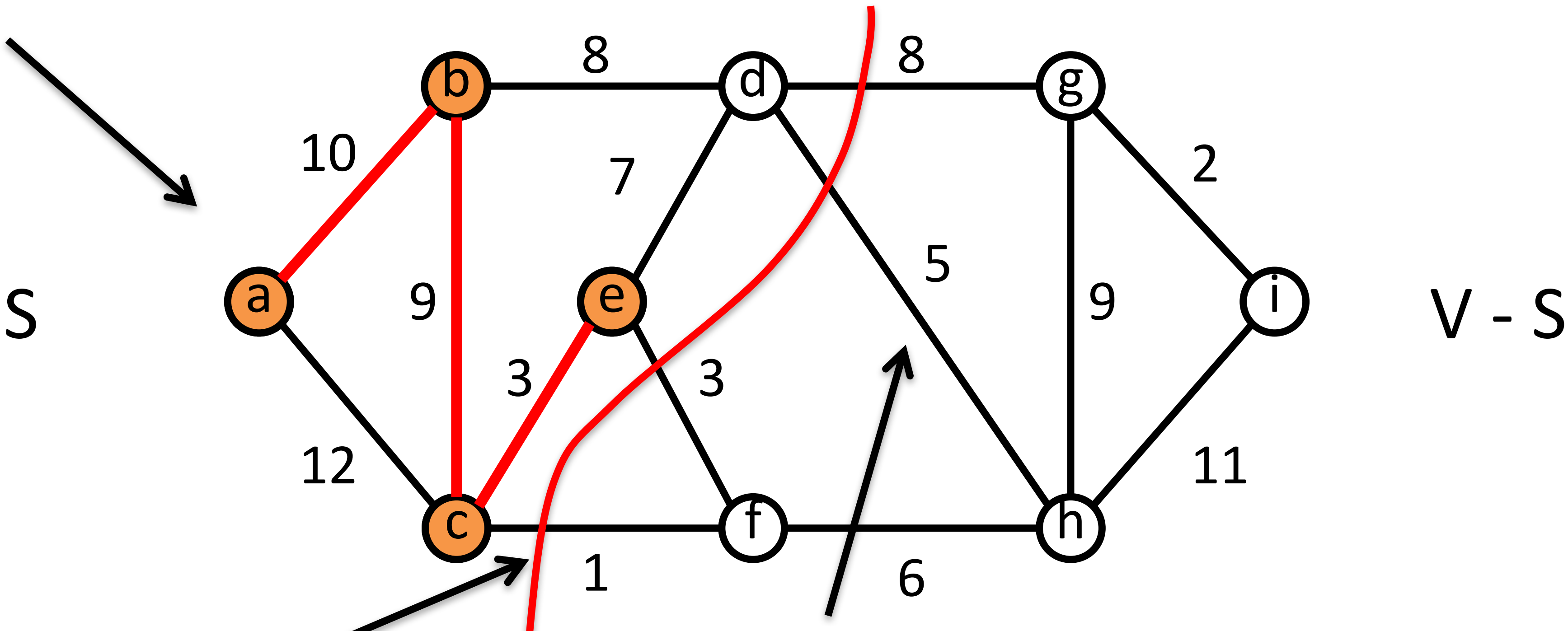
```
A ← ∅;
while A is not a spanning tree do
    find a edge (u, v) that is safe for A;
    A ← A ∪ {(u, v)}
return A
```

- **Initialization:** The empty set trivially satisfies the loop invariant.
- **Maintenance:** We add only safe edges, *A* remains a subset of some MST.
- **Termination:** All edges added to *A* are in an MST, so when we stop, *A* is a spanning tree that is also an MST.

Definitions

A cut **respects** *A* if and only if no edge in *A* crosses the cut.

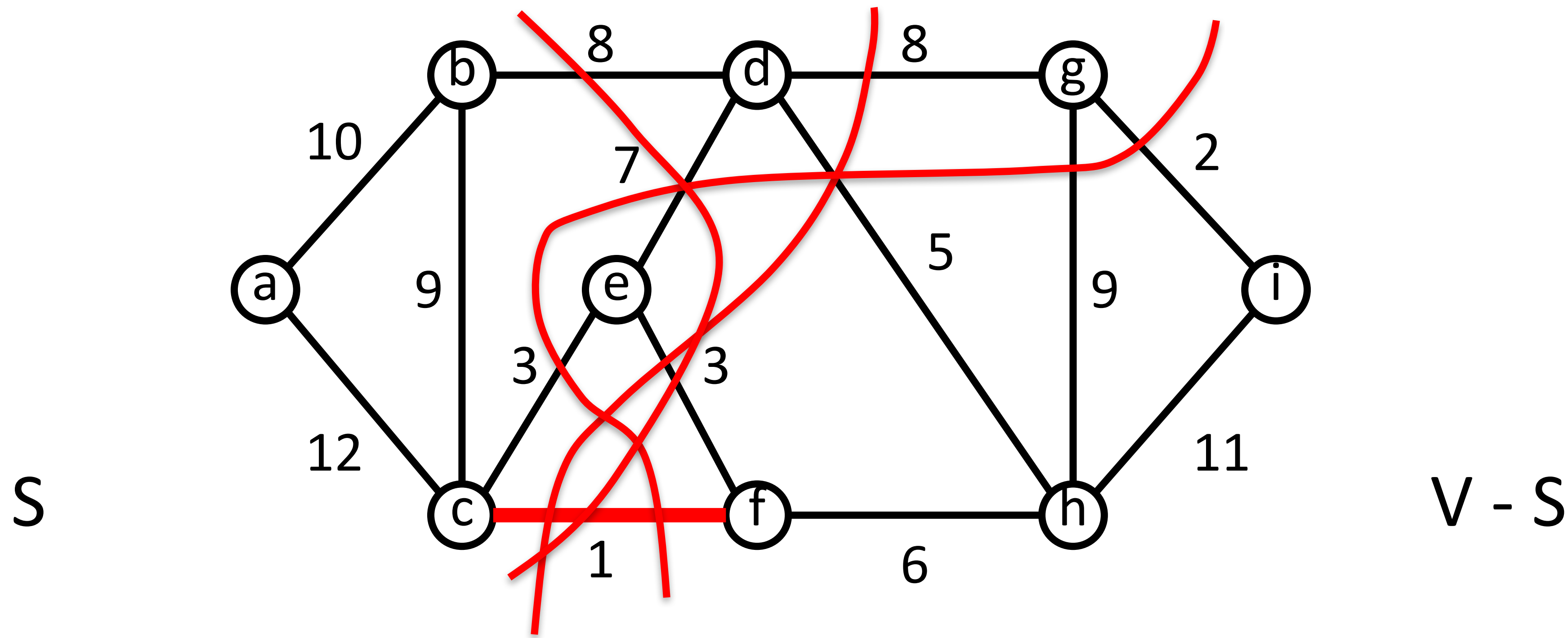
cut partitions vertices into disjoint sets, *S* and *V* – *S*.



A **light** edge crossing cut (may not be unique)

This edge **crosses** the cut. (one endpoint is in *S* and the other is in *V* – *S*.)

What is a safe edge?



Intuitively: Is (c,f) safe when A=∅?

- Let S be any set of vertices including c but not f.
- There has to be one edge (at least) that connects *S* with *V* – *S*.
- Why not choosing the one with the minimum weight?

Safe edge

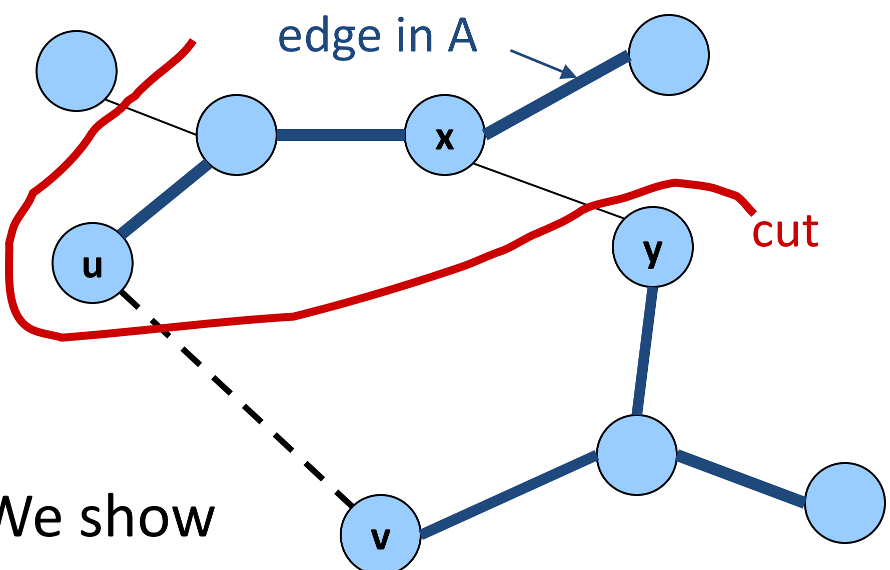
Theorem 1: Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing (S, V-S). Then, (u, v) is safe for A.

Proof:

Let T be a MST that includes A.

Case 1: (u, v) in T. We're done.

Case 2: (u, v) not in T. We have the following:



We show edges in T

(x, y) crosses cut.

Let $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Because (u, v) is light for cut, $w(u, v) \leq w(x, y)$. Thus, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.

Hence, T′ is also a MST.

So, **(u, v) is safe for A**.

Corollary

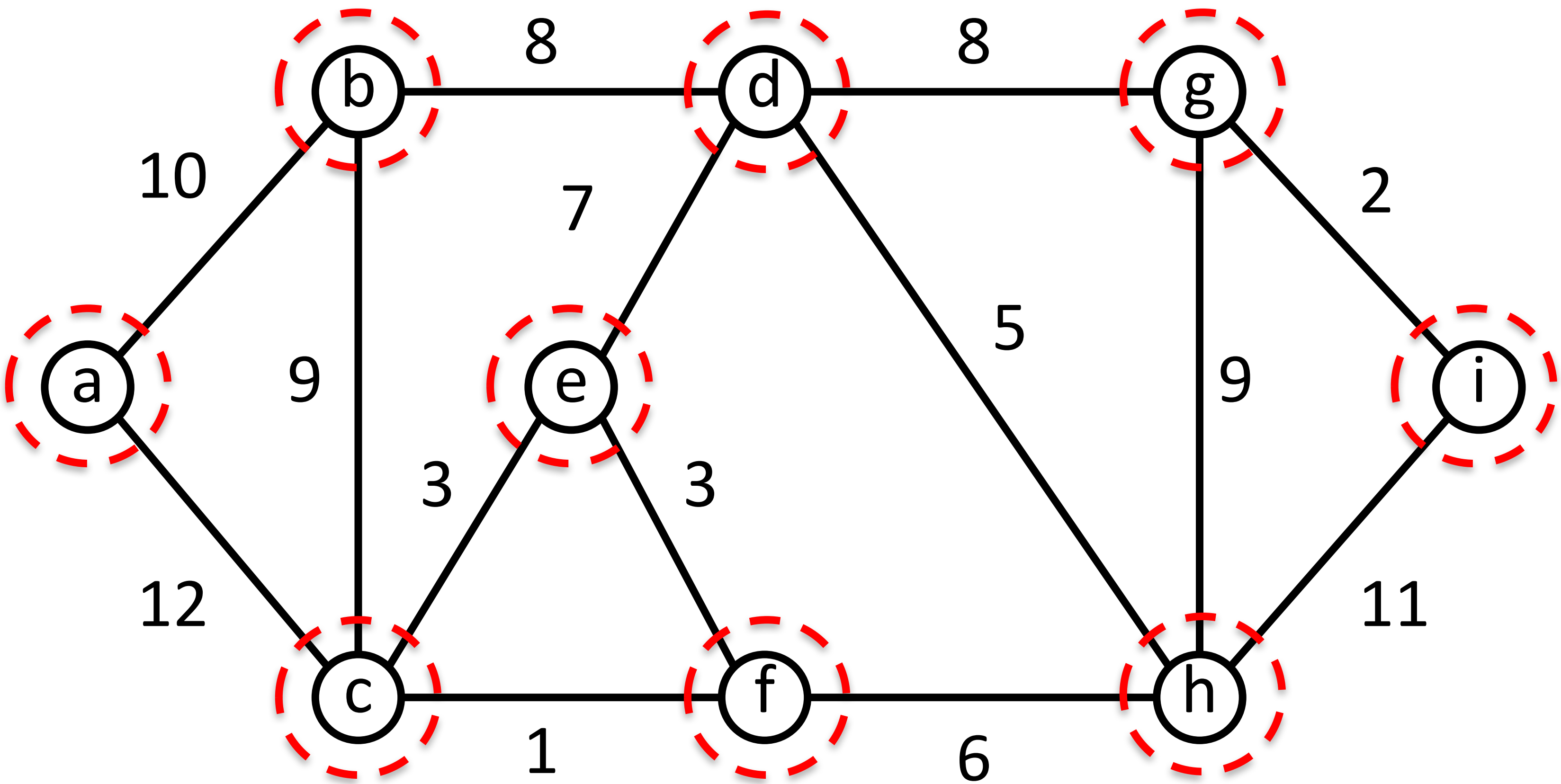
In general, A will consist of several connected components.

Corollary: If (u, v) is a light edge connecting one CC in (V, A) to another CC in (V, A), then (u, v) is safe for A.

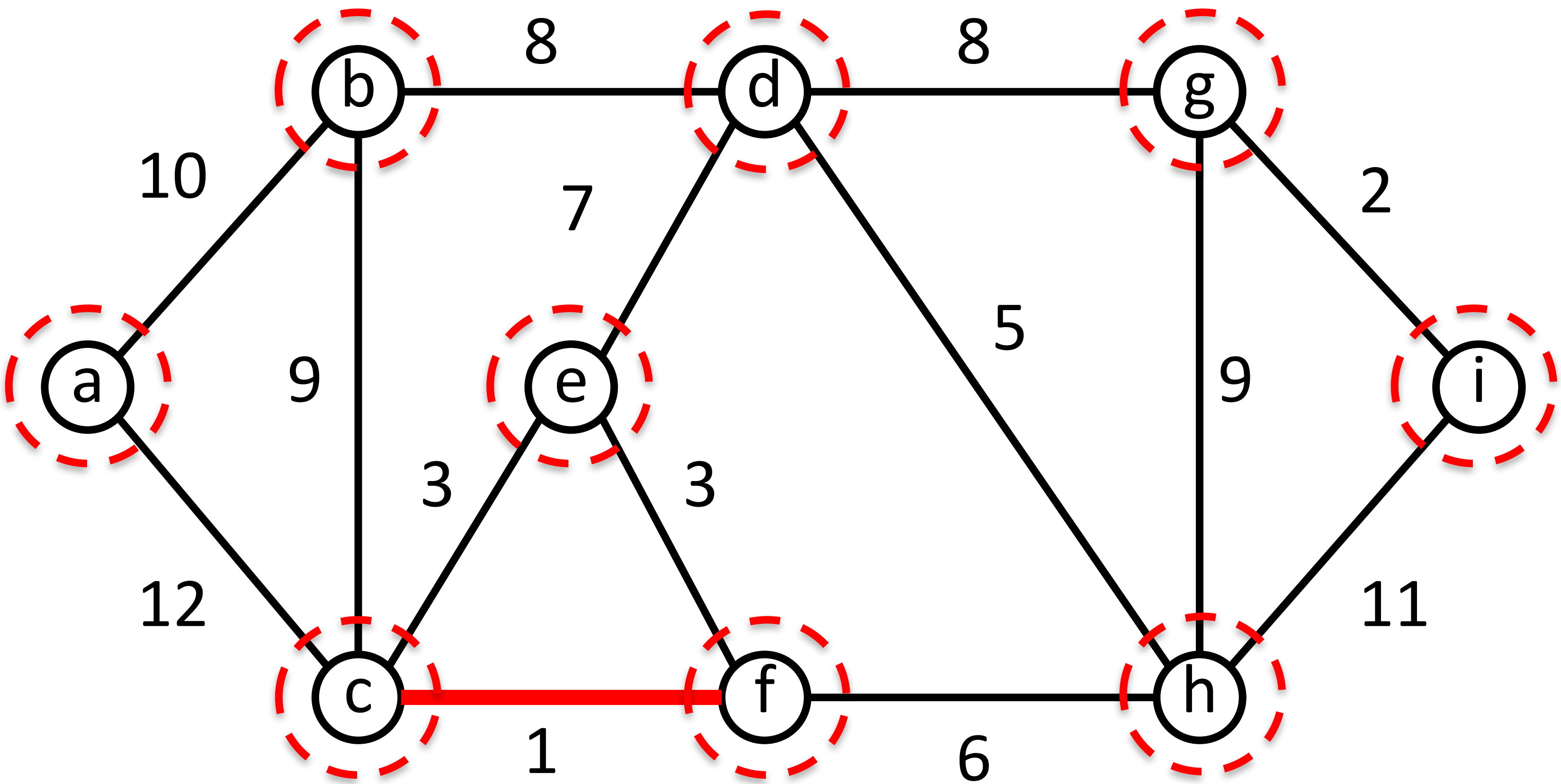
Kruskal's Algorithm

1. Starts with each vertex in its own component.
2. Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
3. Scans the set of edges in monotonically increasing order by weight.
4. Uses a **disjoint-set data structure** to determine whether an edge connects vertices in different components.

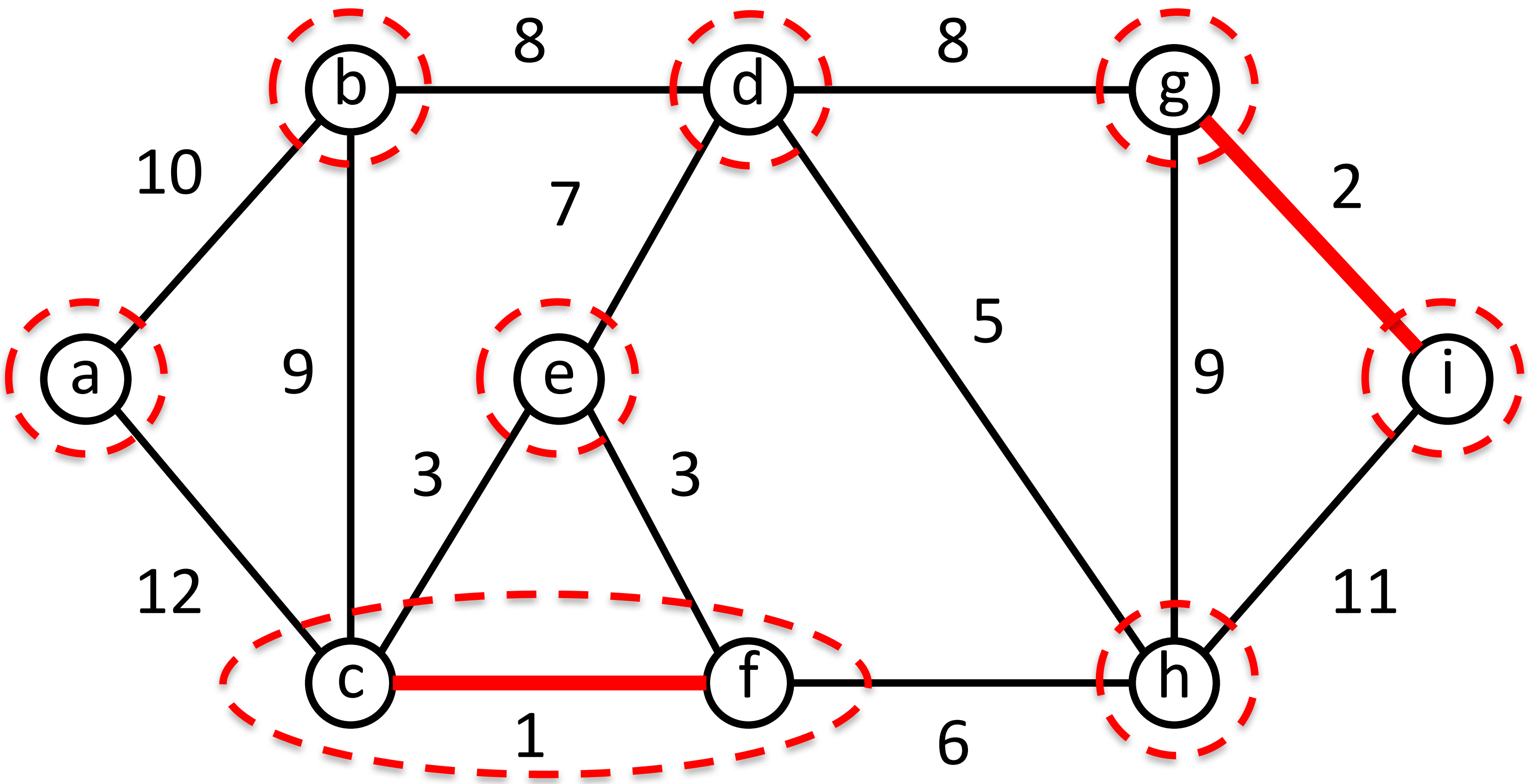
Example



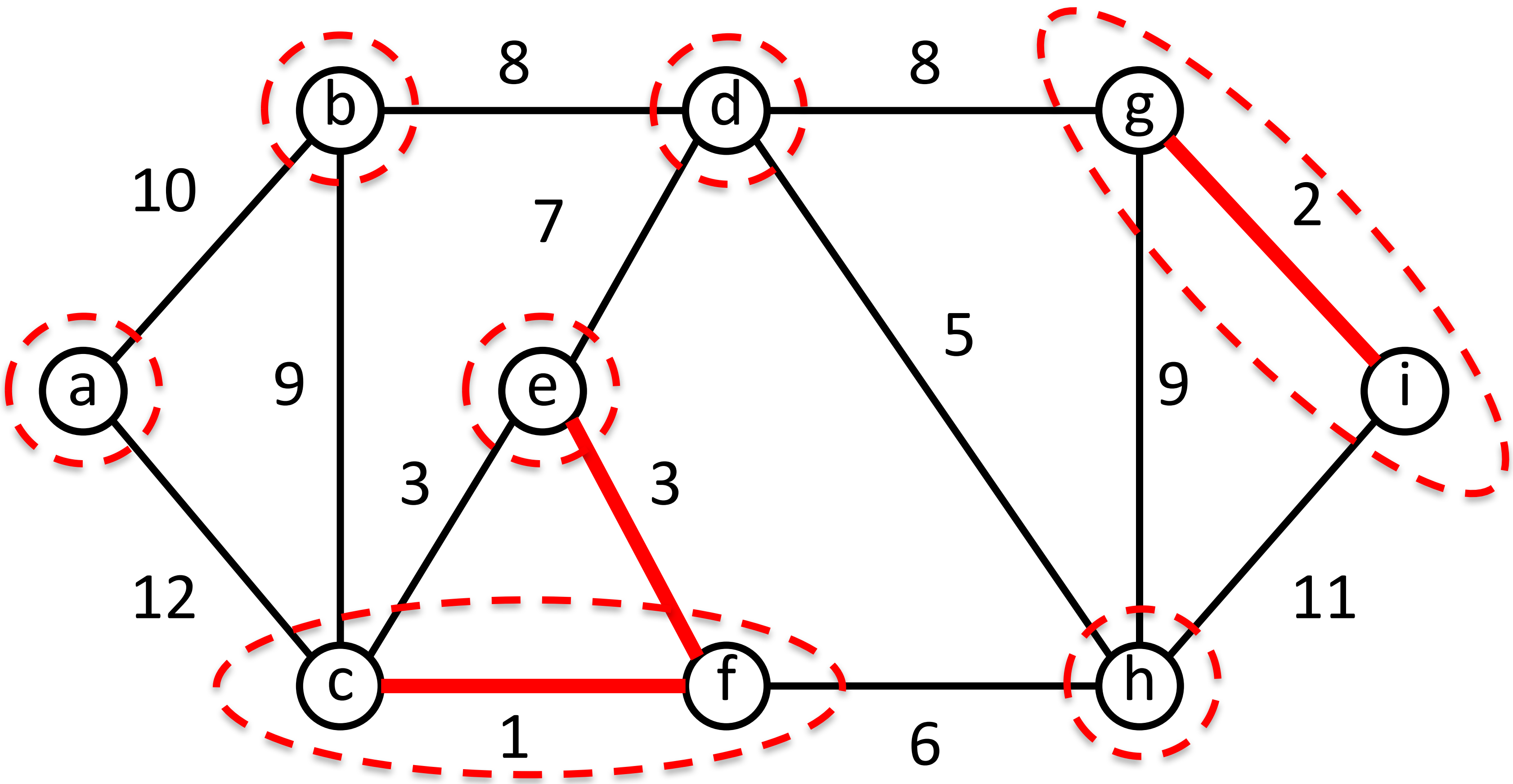
Example



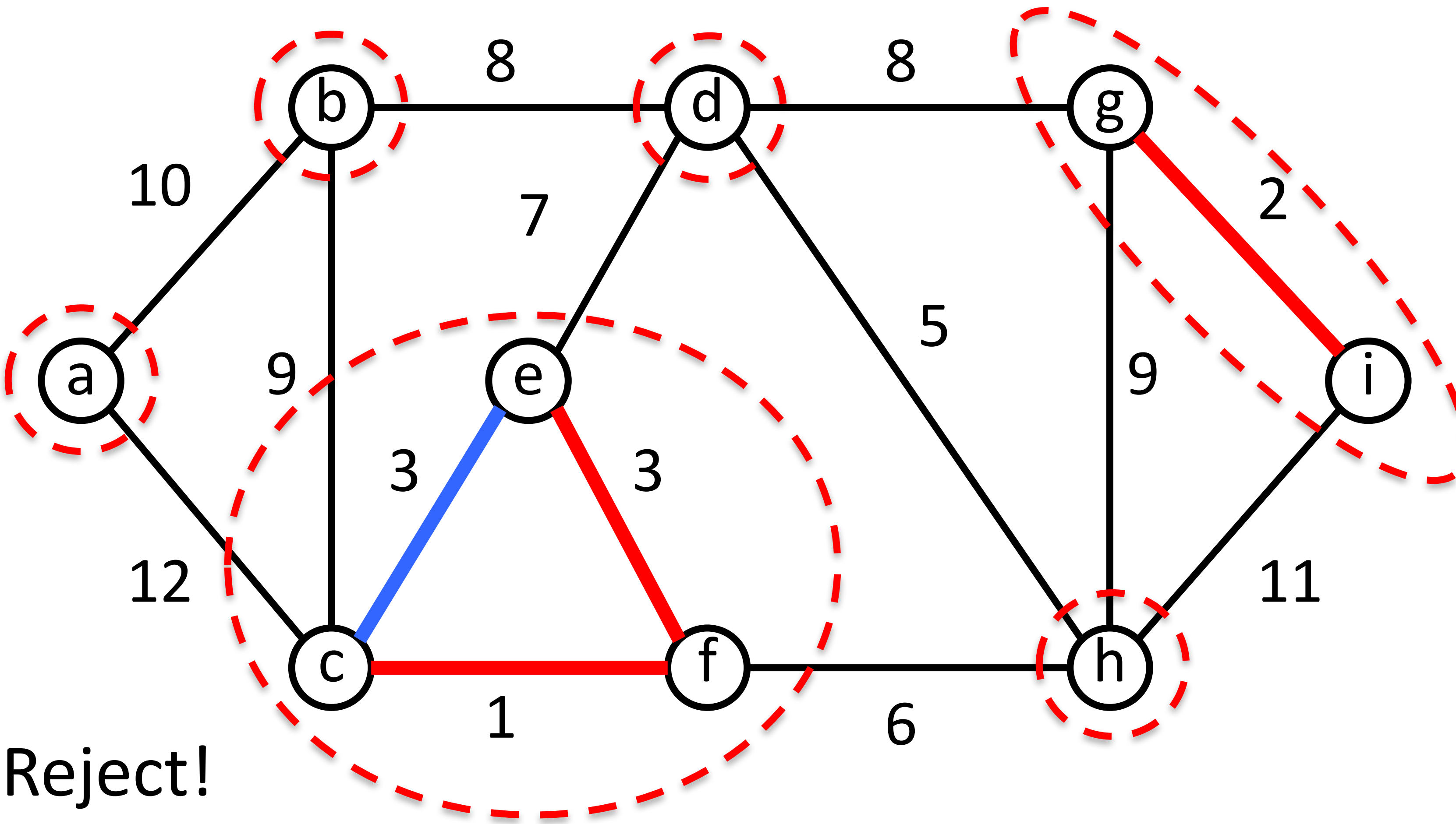
Example



Example

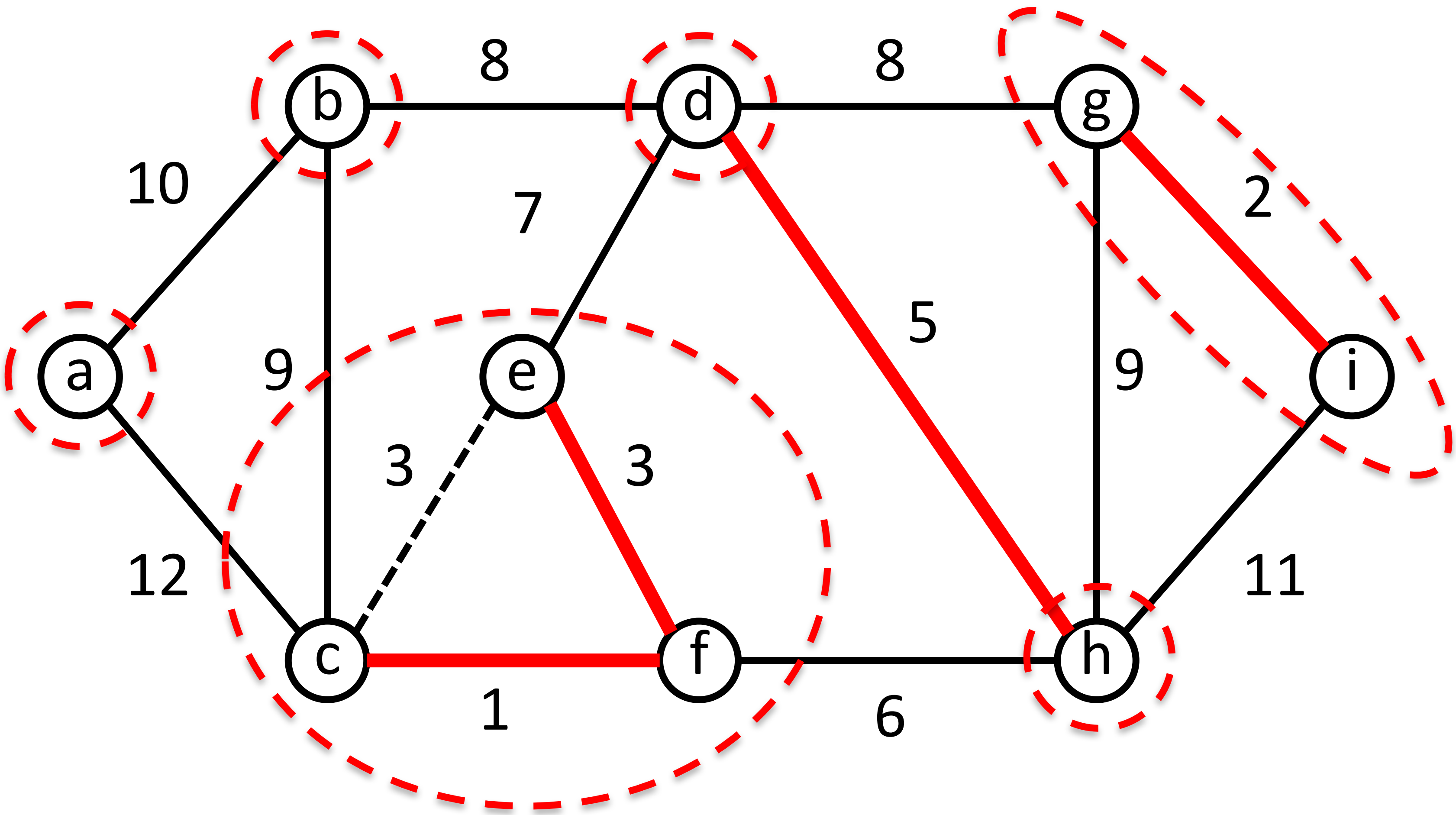


Example

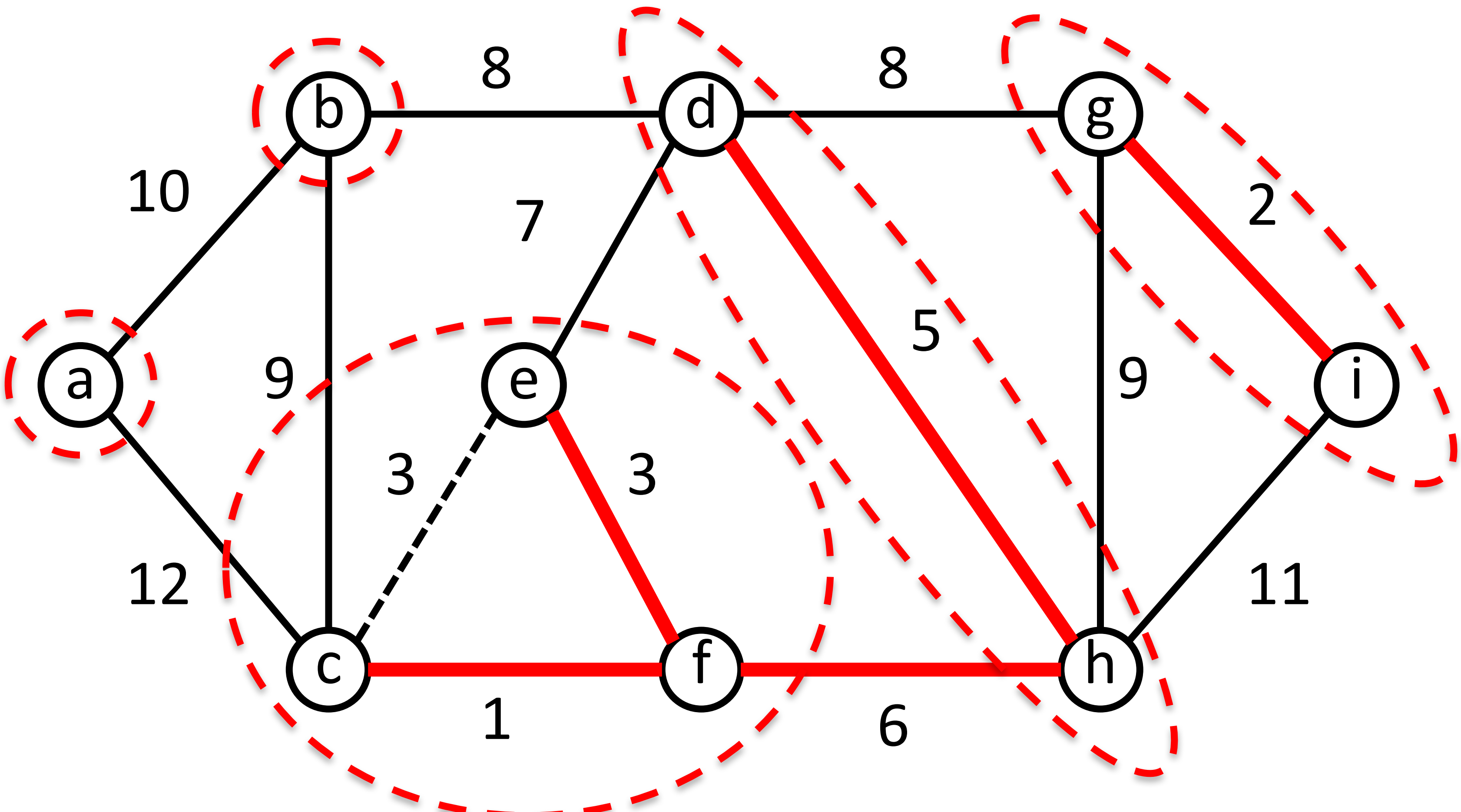


Reject!

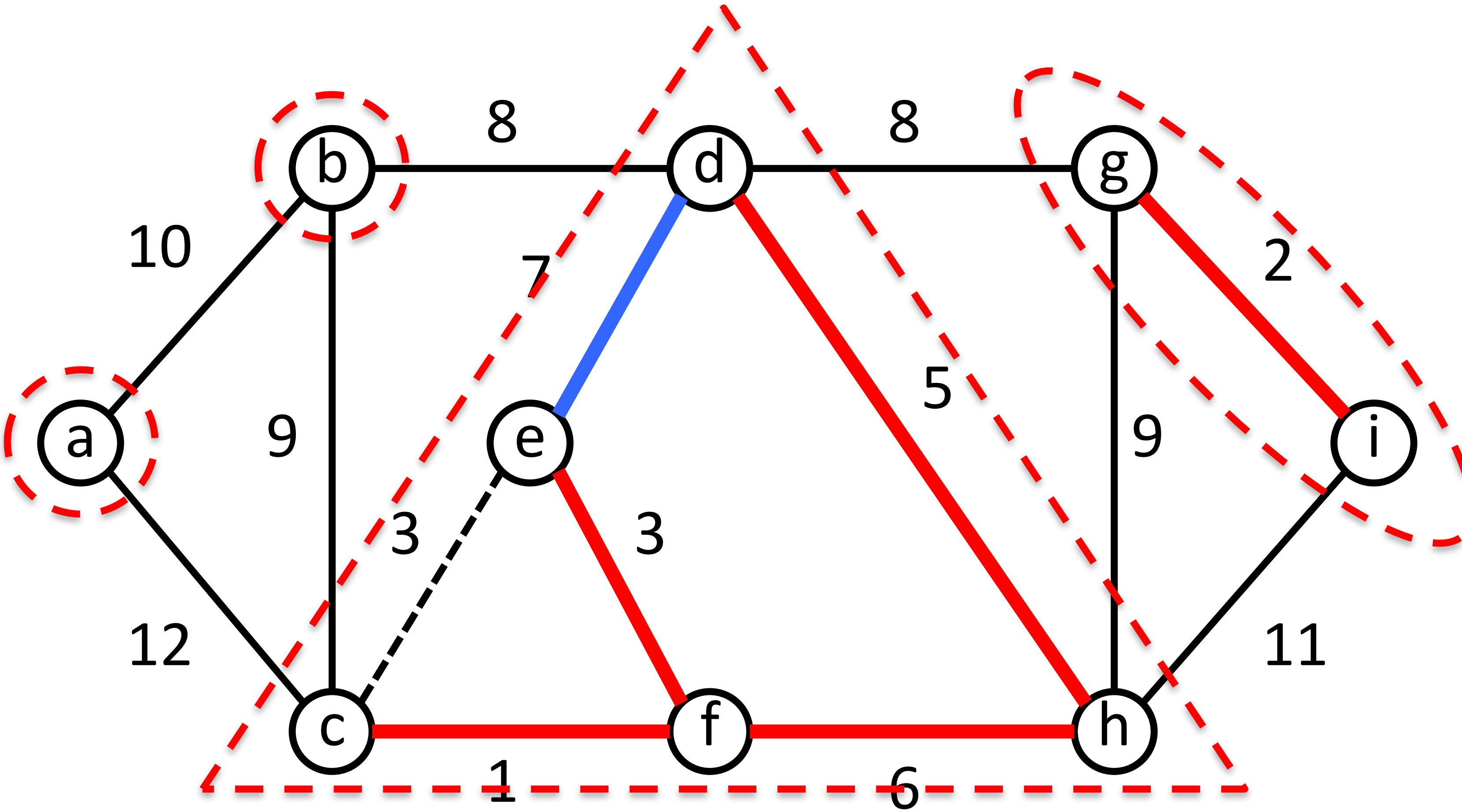
Example



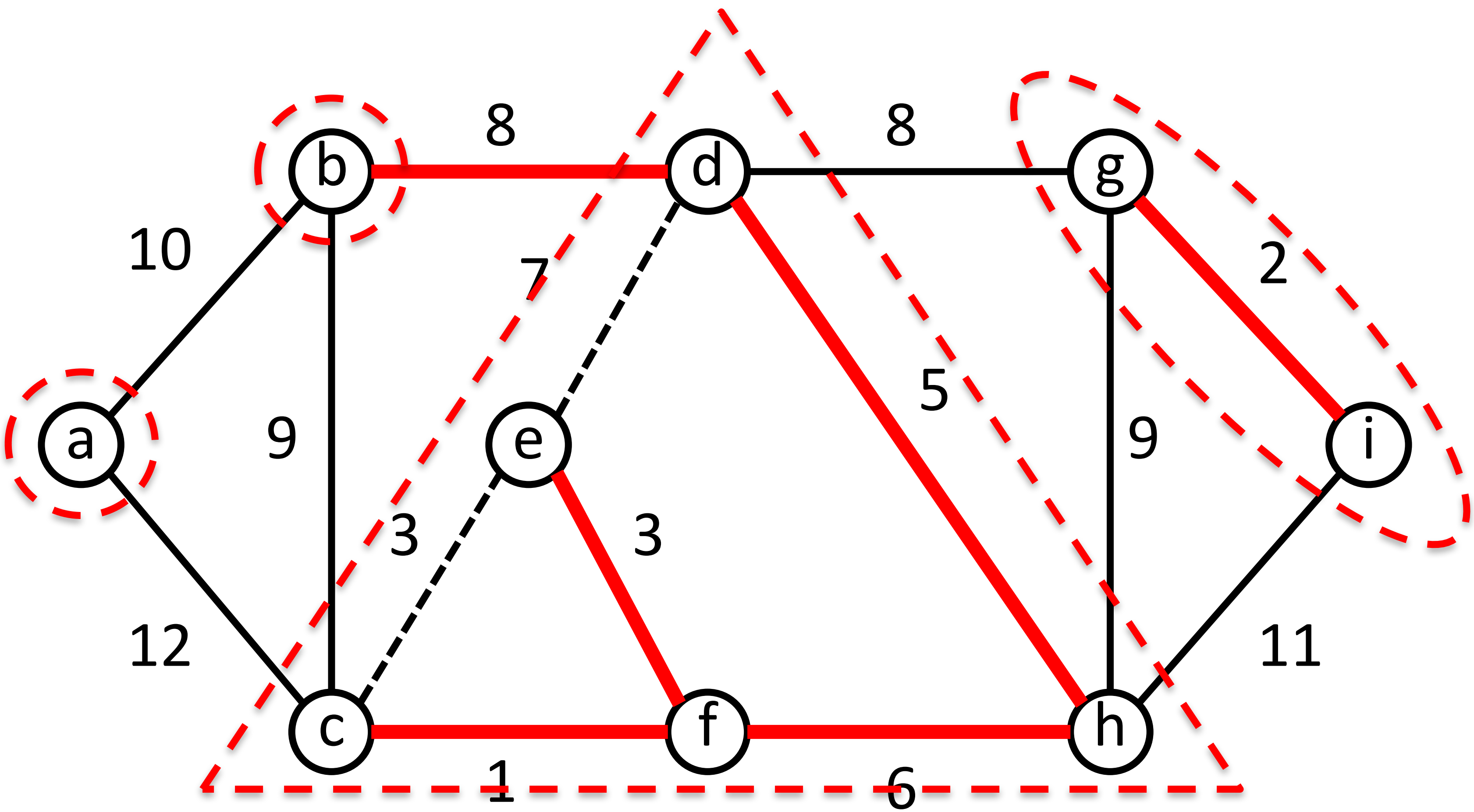
Example



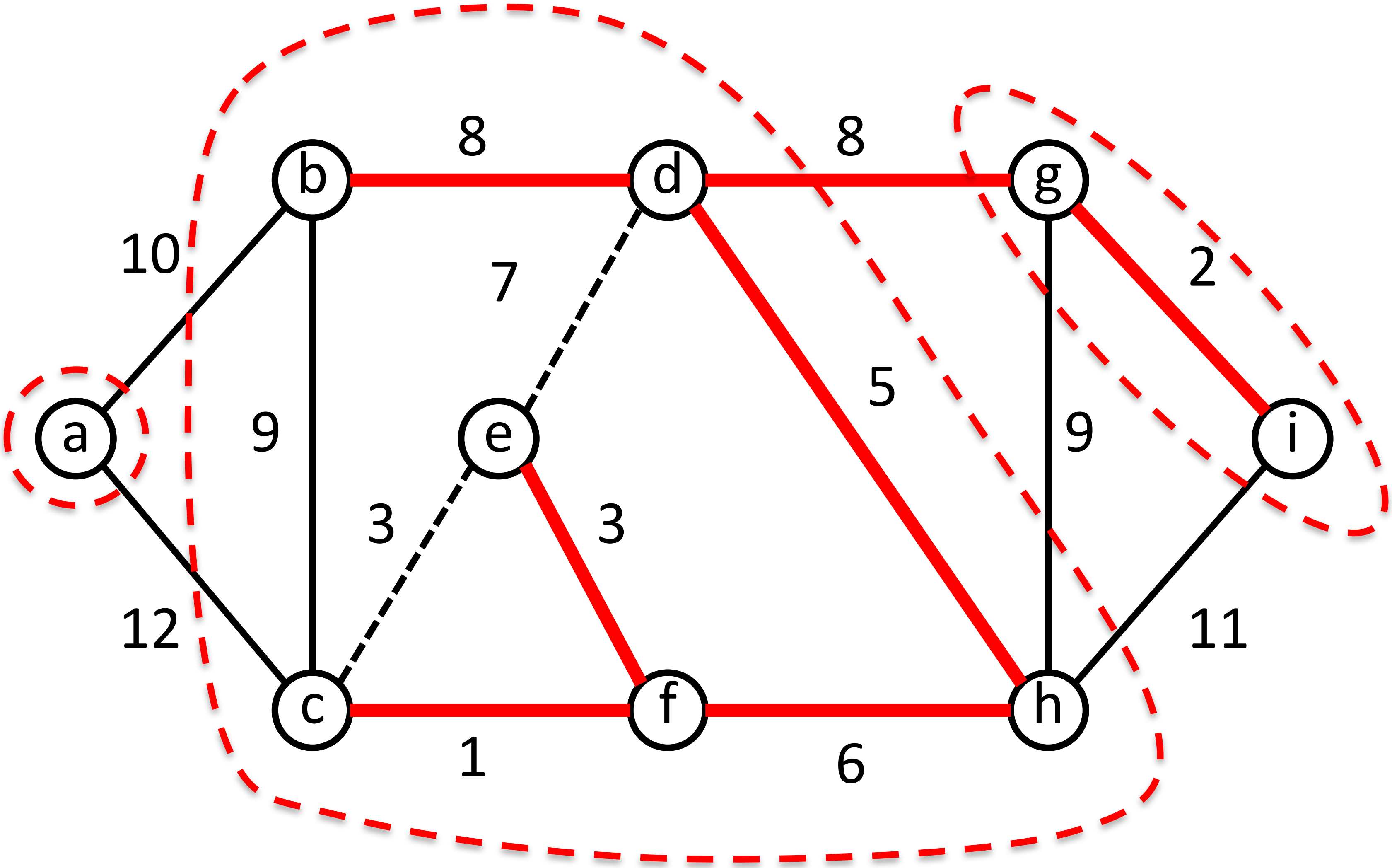
Example



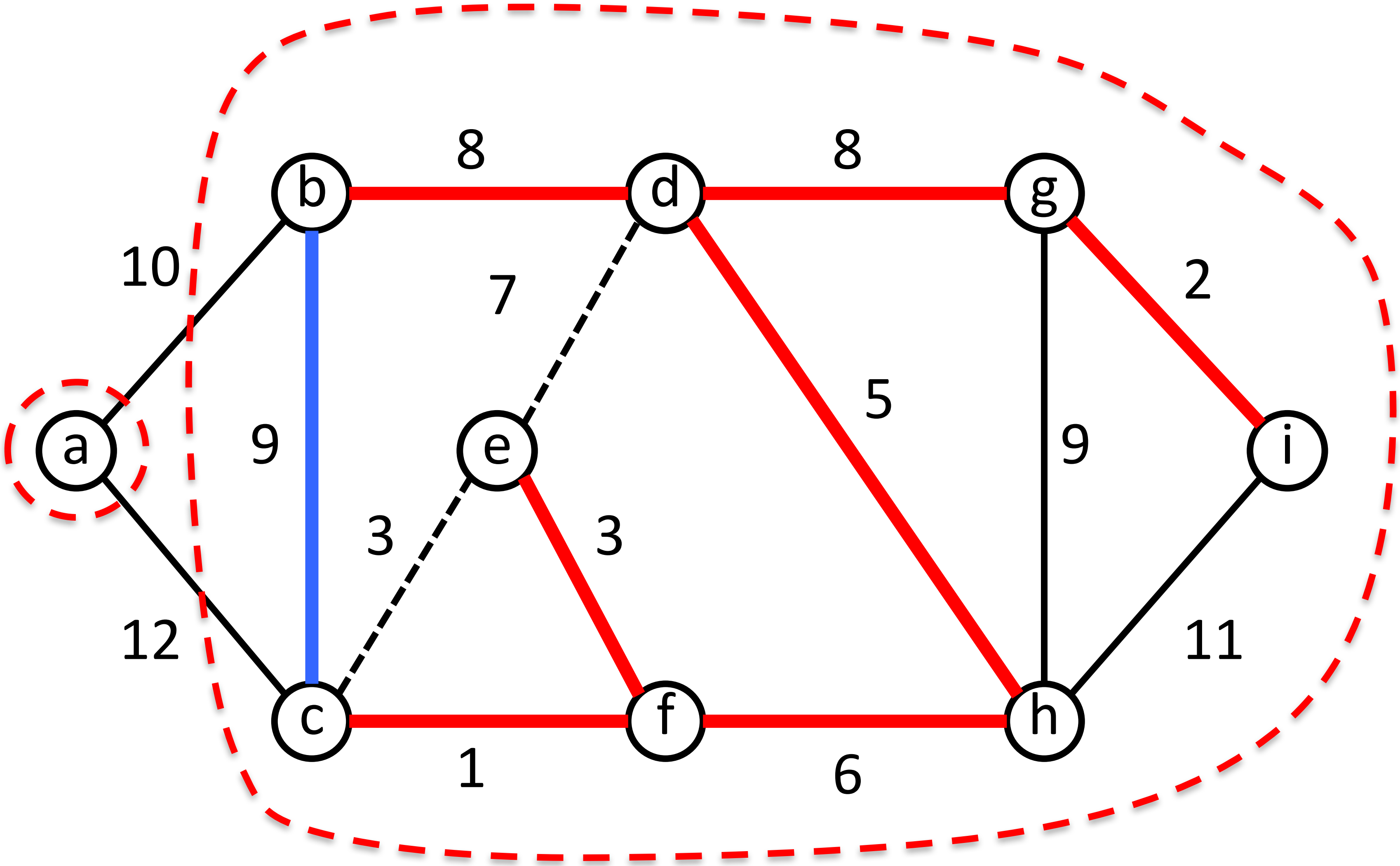
Example



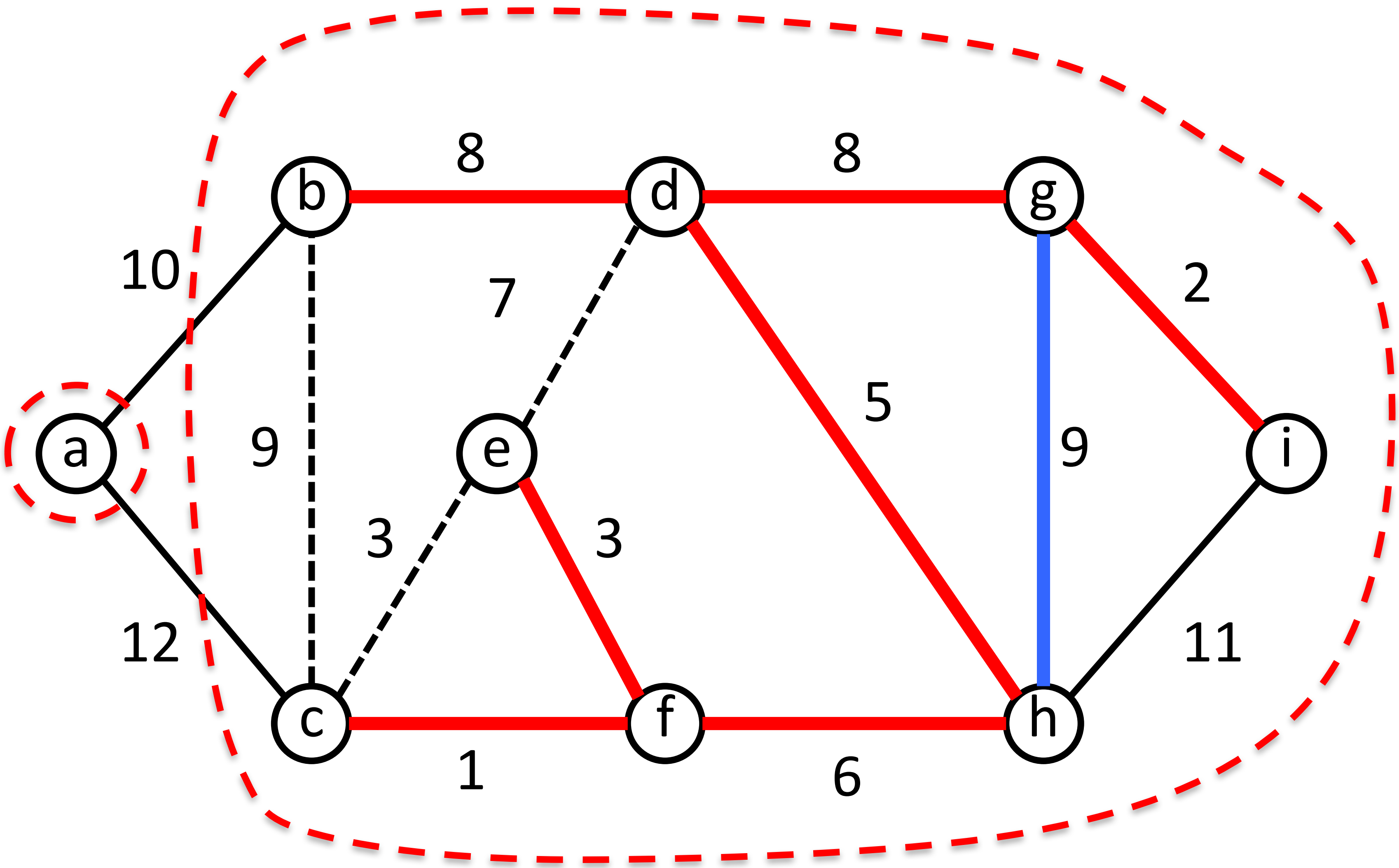
Example



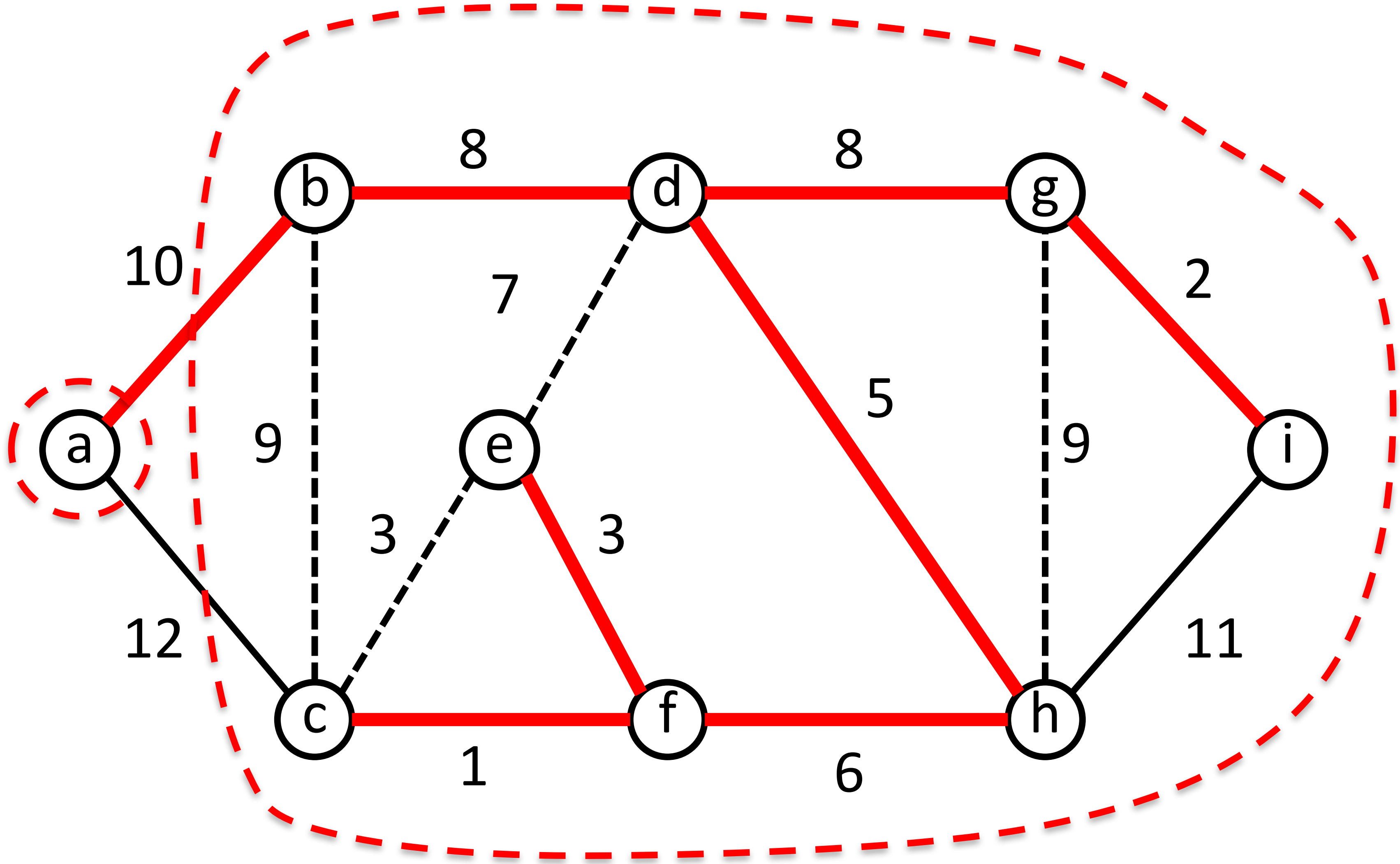
Example



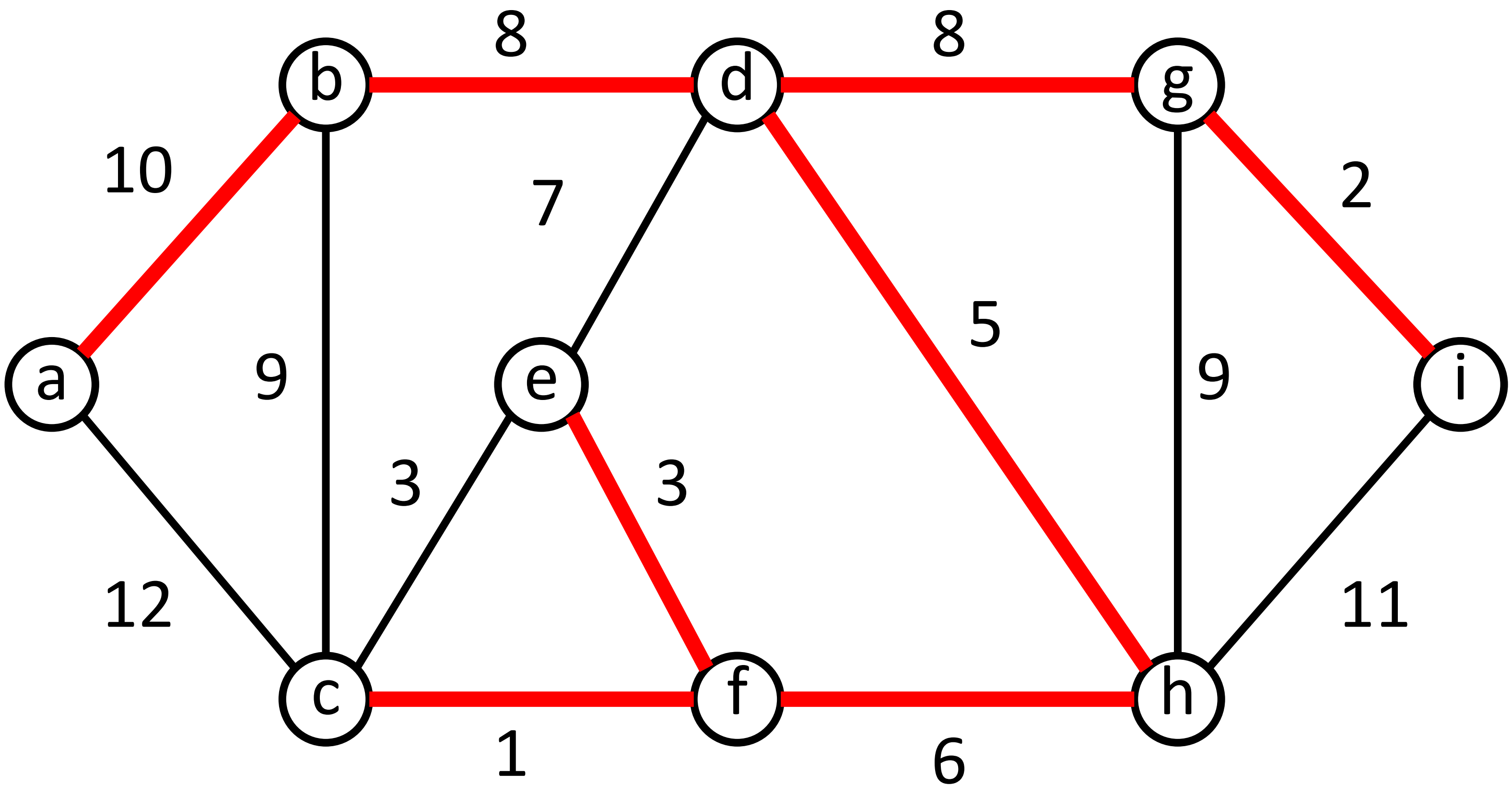
Example



Example



Example



Kruskal's complexity

- Initialize A: $O(1)$
- First **for** loop: $|V|$ MAKE-SETs
- Sort E: $O(E \lg E)$
- Second **for** loop: $O(E)$ FIND-SETs and UNIONs

Assuming union by rank and path compression:
$O((V+E)\alpha(V))+O(E \lg E)$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|)= O(\lg V)= O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg|E| = O(2\lg V) = O(\lg V)$.

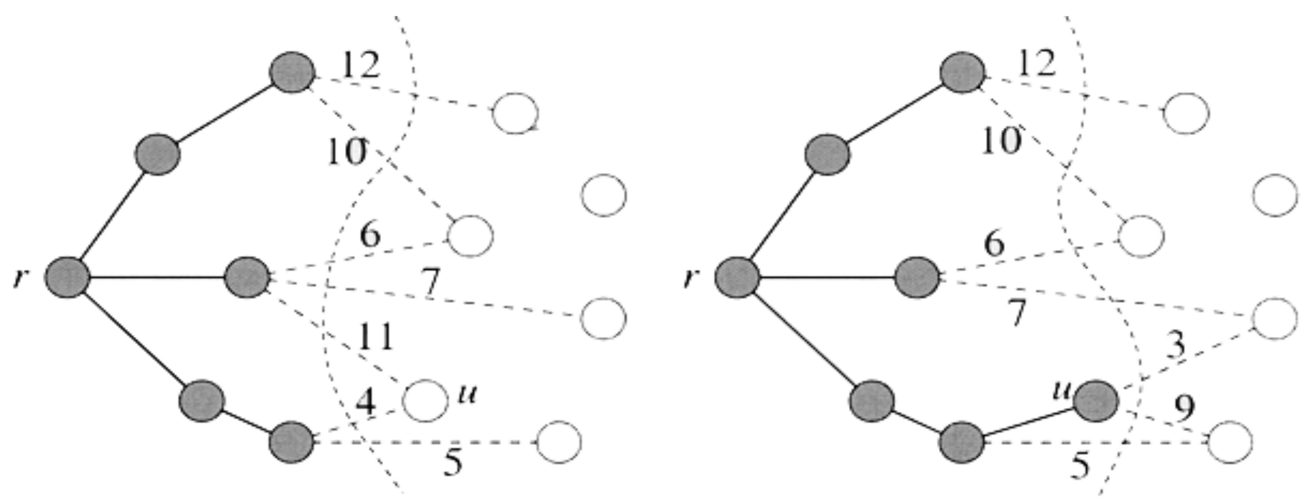
$\Rightarrow O(E \lg V)$ time

Prim's Algorithm

1. Builds **one tree**, so A is always a tree.
2. Starts from an arbitrary "root" r .
3. At each step, **adds a light edge** crossing cut $(V_A, V - V_A)$ to A.
   - Where $V_A =$ vertices that A is incident on.

Intuition behind Prim's Algorithm

- Consider the set of vertices S currently part of the tree, and its complement (V-S). We have a cut of the graph and the current set of tree edges A is respected by this cut.
- Which edge should we add next? *Light edge!*



Finding a light edge

1. Uses a **priority queue *Q*** to find a light edge quickly.
2. Each object in *Q* is a vertex in $V - V_A$.
3. Key of *v* has minimum weight of any edge (u, v), where $u \in V_A$.
4. Then the vertex returned by Extract-Min is *v* such that there exists $u \in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
5. Key of *v* is ∞ if *v* is not adjacent to any vertex in V_A.

Basics of Prim 's Algorithm

- It works by adding leaves on at a time to the current tree.
  - Start with the root vertex r (it can be any vertex). At any time, the subset of edges A forms a single tree. S = vertices of A.
  - At each step, a light edge connecting a vertex in S to a vertex in V- S is added to the tree.
  - The tree grows until it spans all the vertices in V.
- Implementation Issues:
  - How to update the cut efficiently?
  - How to determine the light edge quickly?

Implementation: Priority Queue

- Priority queue implemented using heap can support the following operations in $O(lg\ n)$ time:
  - Insert (Q, u, key): Insert u with the key value key in Q
  - u = Extract_Min(Q): Extract the item with minimum key value in Q
  - Decrease_Key(Q, u, new_key): Decrease the value of u's key value to new_key
- All the vertices that are *not* in the S (the vertices of the edges in A) reside in a priority queue Q based on a key field. When the algorithm terminates, Q is empty. $A = \{(v, \pi[v]): v \in V - \{r\}\}$

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
    key[u] := ∞
    π[u] := Nil;
    Insert(Q,u)
Decrease-Key(Q,r,0);
while Q ≠ ∅ do
    u := Extract-Min(Q);
    for each v ∈ Adj[u] do
        if v ∈ Q ∧ w(u, v) < key[v] :
            π[v] := u;
            Decrease-Key(Q,v,w(u,v));
```

Complexity:

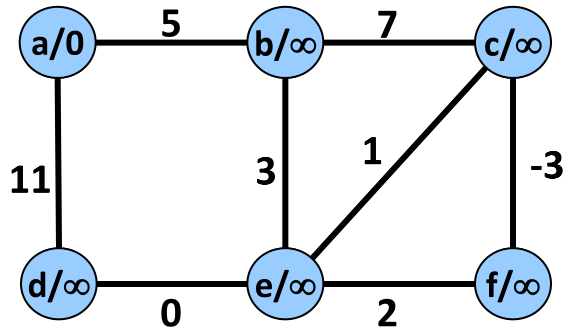
Using binary heaps: $O(E \lg V)$.

- Initialization: $O(V)$.
- Building initial queue: $O(V)$.
- V Extract-Min: $O(V \lg V)$.
- E Decrease-Key: $O(E \lg V)$.

Using Fibonacci heaps: $O(E + V \lg V)$.

Notes: (i) $A = \{(v, \pi[v]) : v \in v - \{r\} - Q\}$. (ii) r is the root.

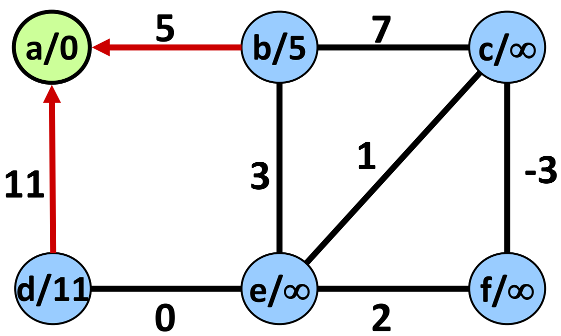
Example of Prim's Algorithm



Not in tree

Q = a b c d e f
0 ∞ ∞ ∞ ∞ ∞

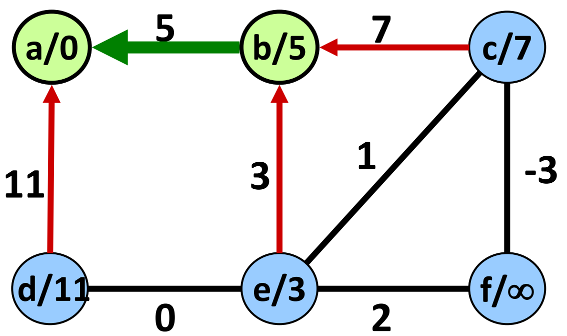
Example of Prim’s Algorithm



Q = b d c e f

5 11 ∞ ∞ ∞

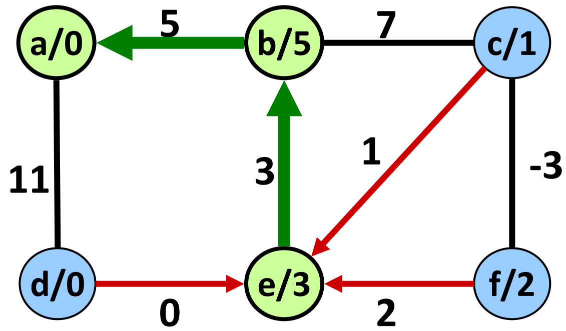
Example of Prim's Algorithm



Q = e c d f

3 7 11 ∞

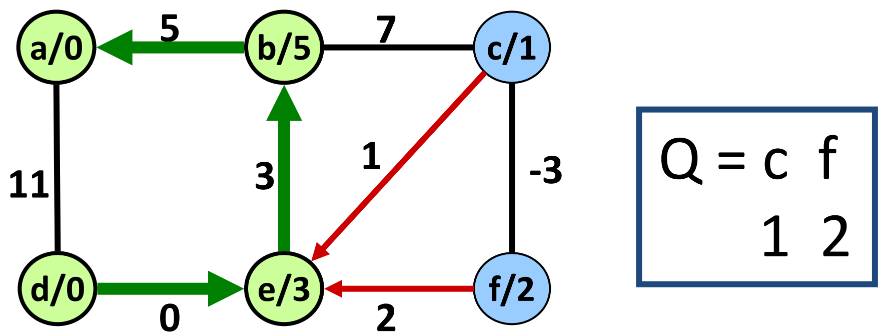
Example of Prim's Algorithm



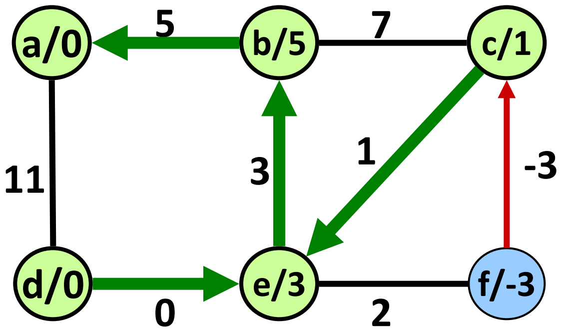
Q = d c f

0 1 2

Example of Prim’s Algorithm



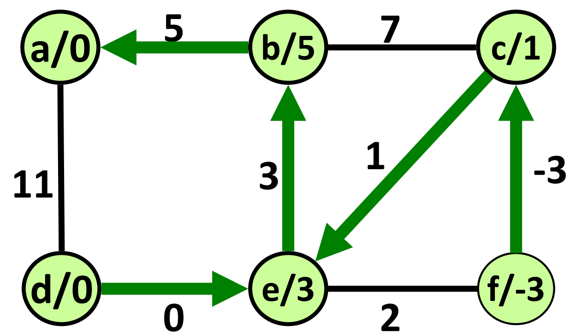
Example of Prim's Algorithm



Q = f

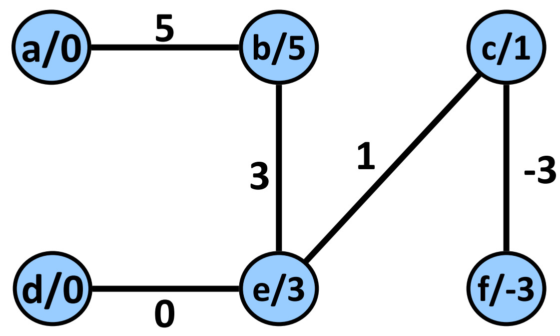
-3

Example of Prim's Algorithm



$Q = \varnothing$

Example of Prim's Algorithm



Correctness of Prim

- Again, show that every edge added is a safe edge for A
- Assume (u, v) is next edge to be added to A.
- Consider the cut $(A, V\text{-}A)$.
  - This cut respects A
  - and (u, v) is the light edge across the cut
- Thus, by the Theorem 1, (u,v) is safe.