COMP251: Minimum Spanning Trees

Giulia Alberini & Jérôme Waldispühl
School of Computer Science
McGill University

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

Based on slides from D. Plaisted (UNC)

Announcements

- A2 has been posted.
  - Due on Nov. 10

- Midterm:
  - Where: Crowdmark (online)
  - When: Nov. 1st, 1h20min long starting at 10:05am (written to be completed in 1h10)
  - Format: 50% multiple choice/short answers + 50% long answer (with 1 small proof)

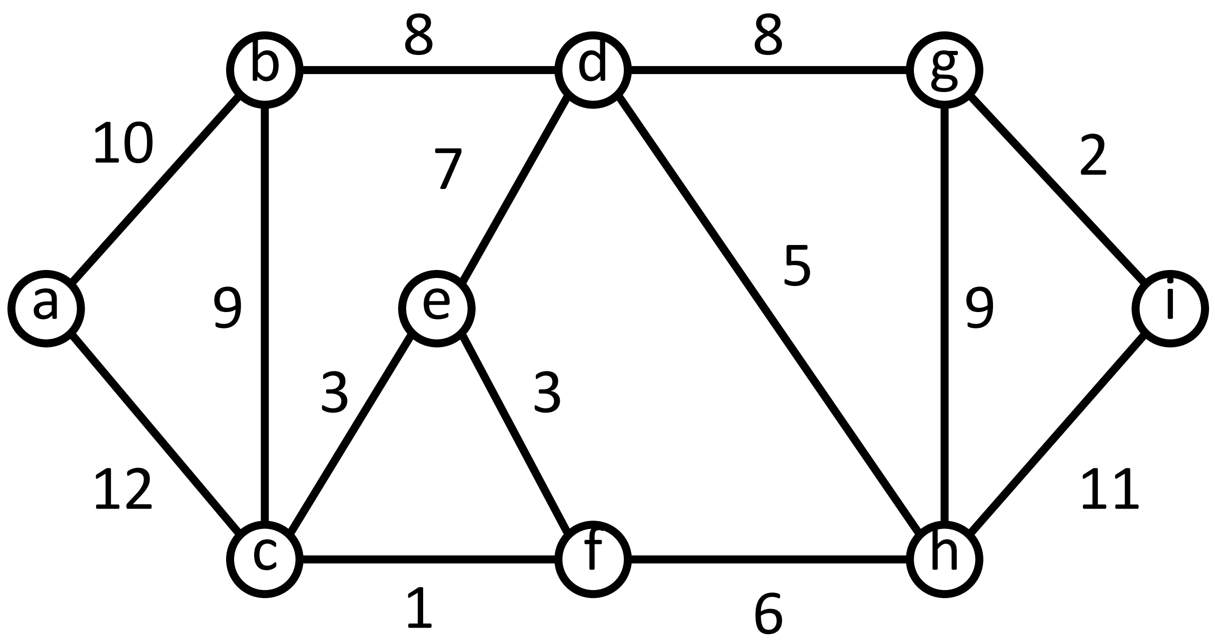
Minimum Spanning Tree (Example)

- 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:

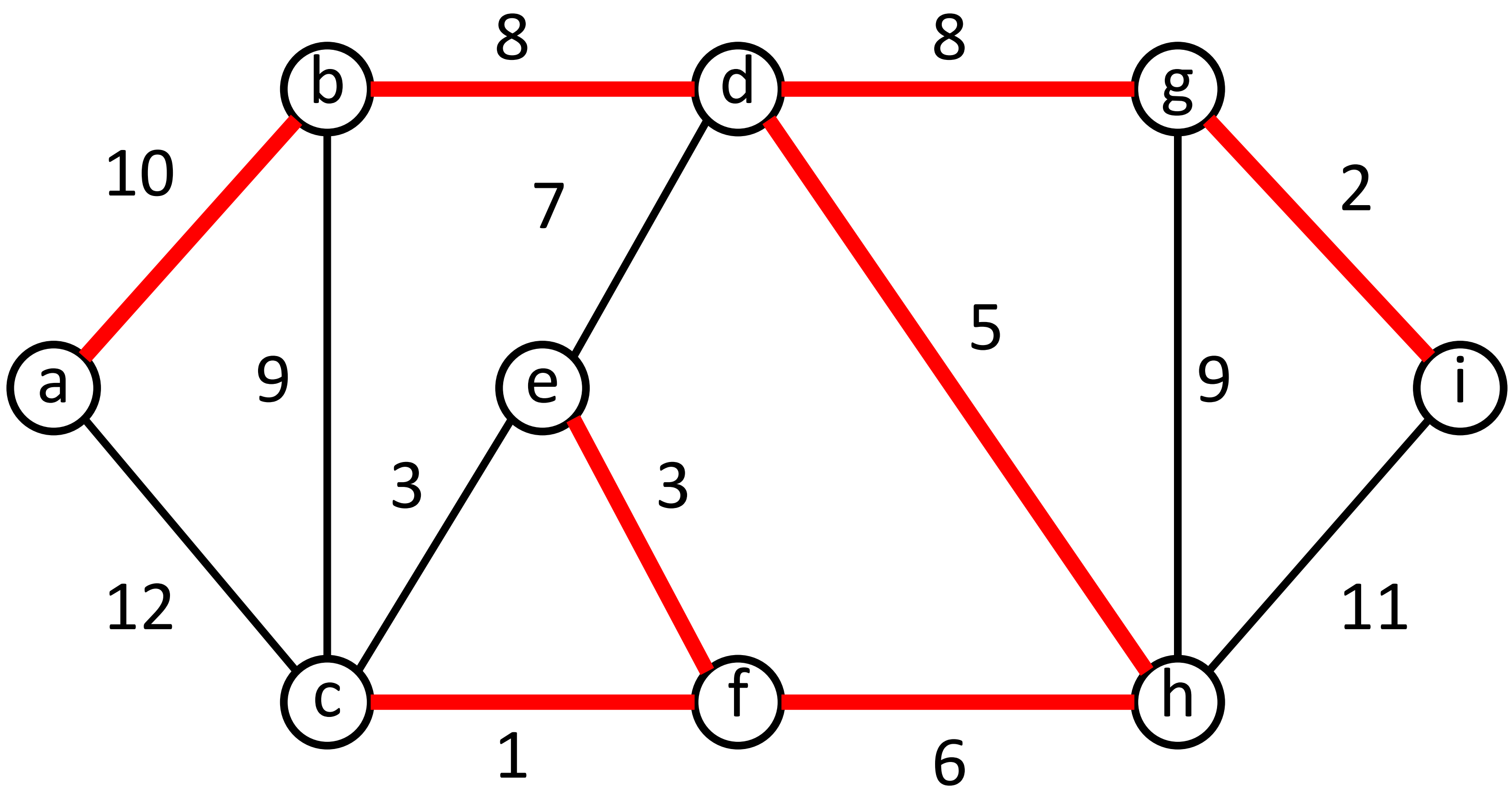
1. everyone stays connected: can reach every house from all other houses, and
2. total repair cost is minimum.

Model as graph



- Undirected graph *G* = (*V*, *E*).
- ***Weight*** w(*u*, v) on each edge (*u*, v) ∈ *E*.
- Find *T* ⊆ *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
```

Safe: can be added without breaking the property

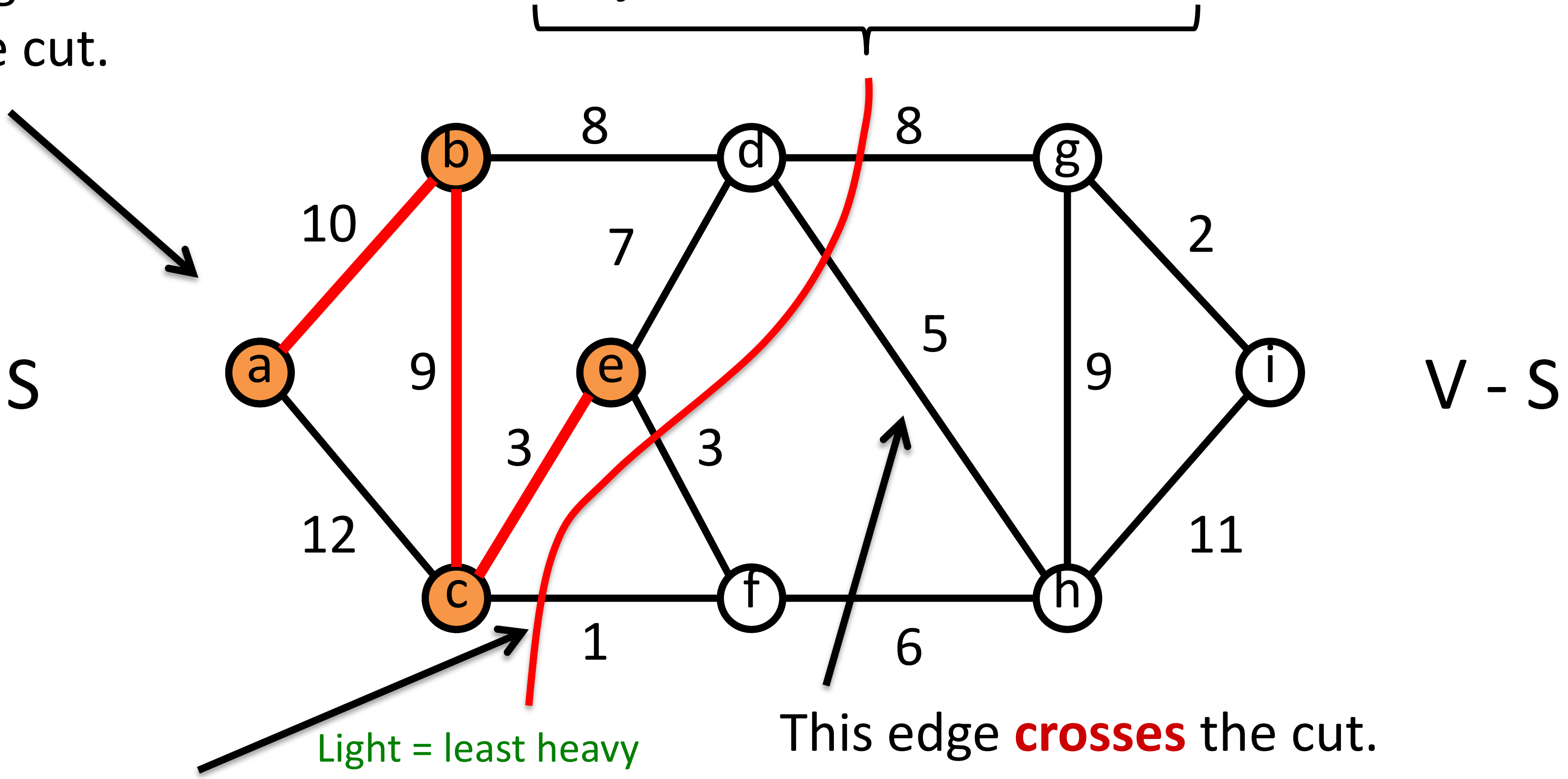
- **Initialization:** The empty set trivially satisfies the loop invariant.
- **Maintenance:** We add only safe edges, *A* remains a subset of some MST.
- **Termination:** All edges added to *A* are in an MST, so when we stop, *A* is a spanning tree that is also an MST.

Definitions

A is a set of edges

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*.

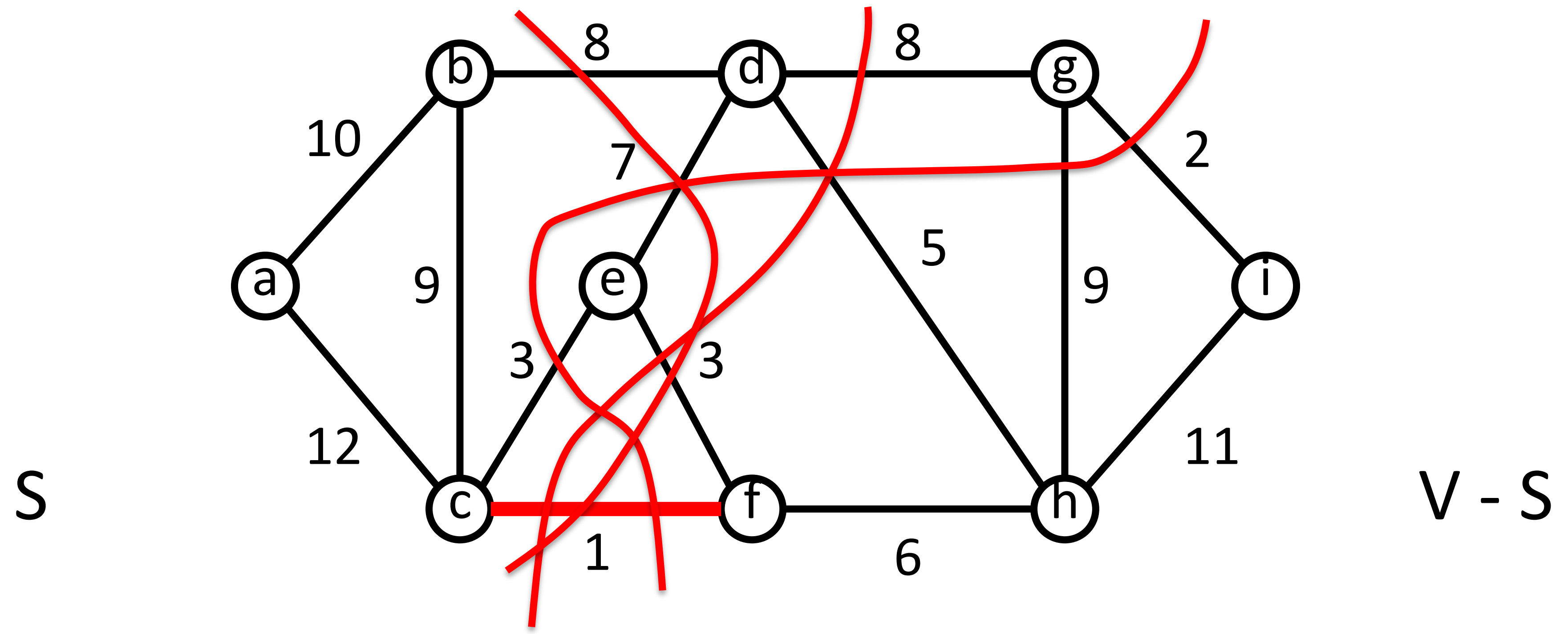


Light = least heavy

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?



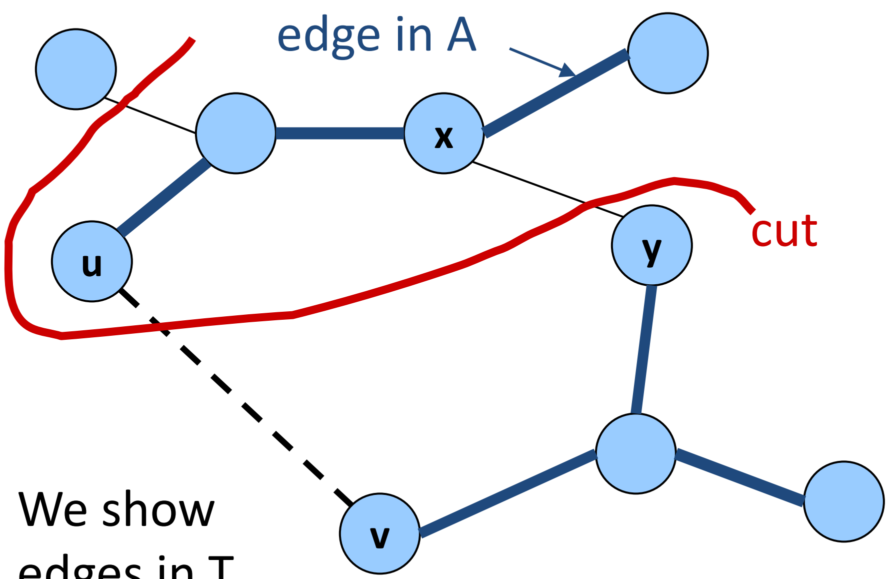
Intuitively: Is (c, f) safe when $A = \emptyset$?

- 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. Let's assume (x, y) is the crossing edge in T.



We show edges in T

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

Because (u, v) is light for the cut, $w(u, v) \le w(x, y)$. Thus,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) \end{aligned}$$

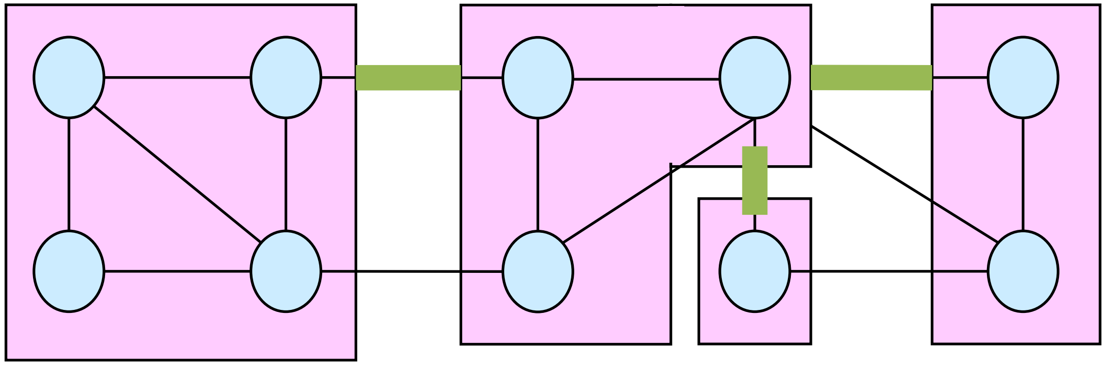
Hence, T' is also a MST. So, $(\boldsymbol{u}, \boldsymbol{v})$ **is safe for** $\boldsymbol{A}$.

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.

Intuitively: if you are connecting two disconnected parts of a MST through a light edge, you are safe

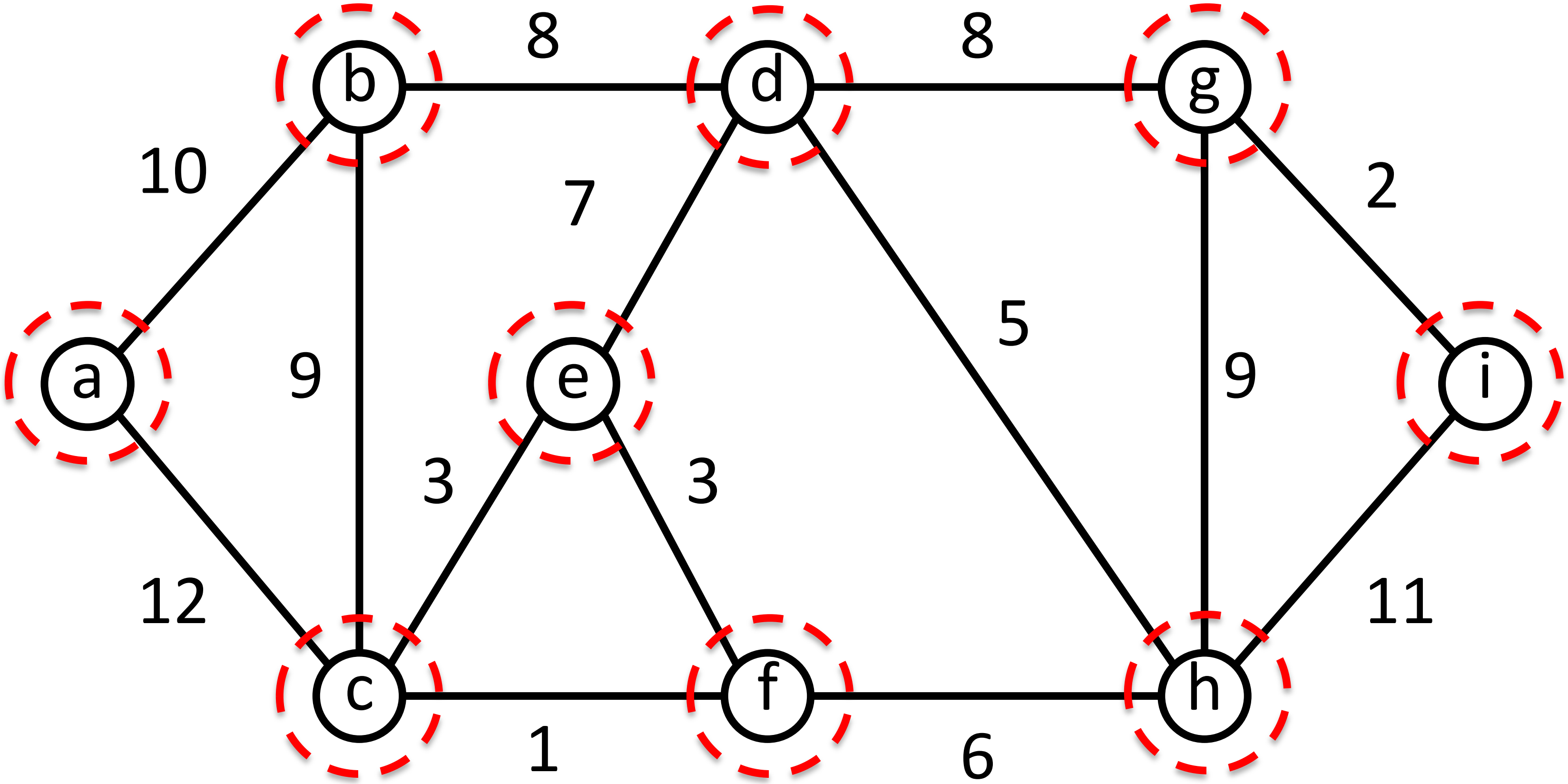


Kruskal's Algorithm

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. Remember: the lowest weight edge of the graph must be safe
4. Uses a **disjoint-set data structure** to determine whether an edge connects vertices in different components.

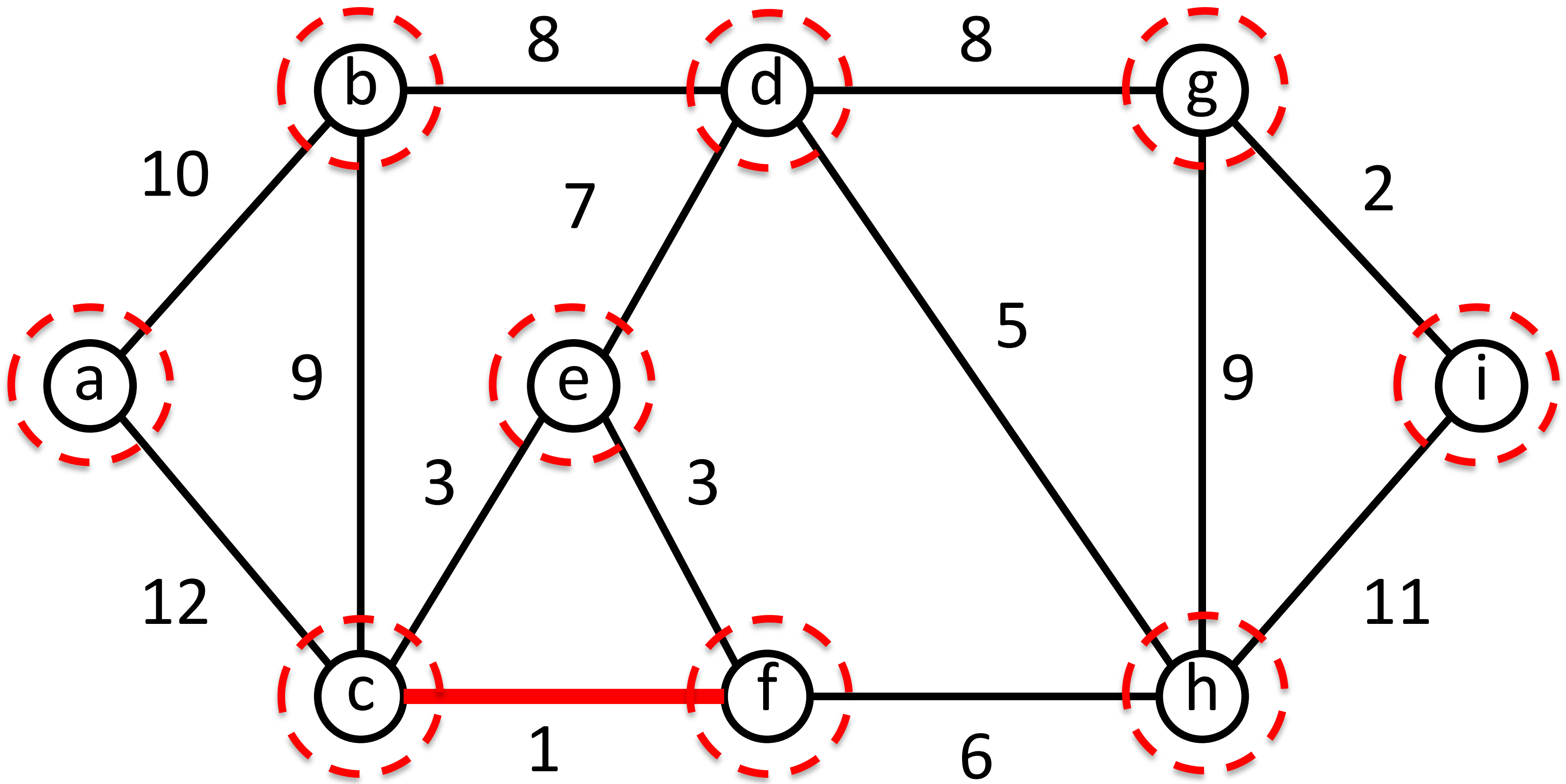
A MST cannot contain cycles!

Example



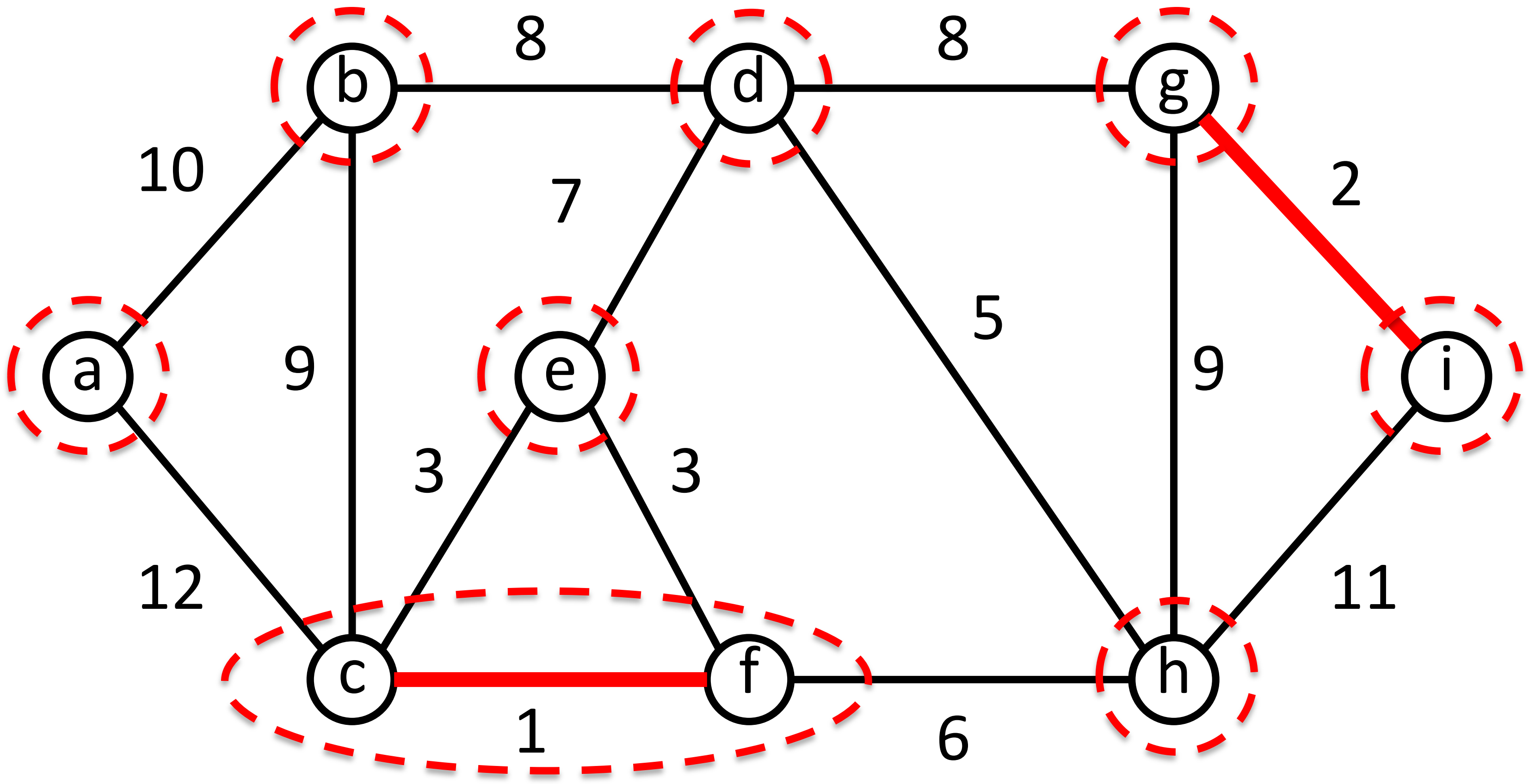
We start with each vertex being a component. We add edges start from the lowest weight

Example



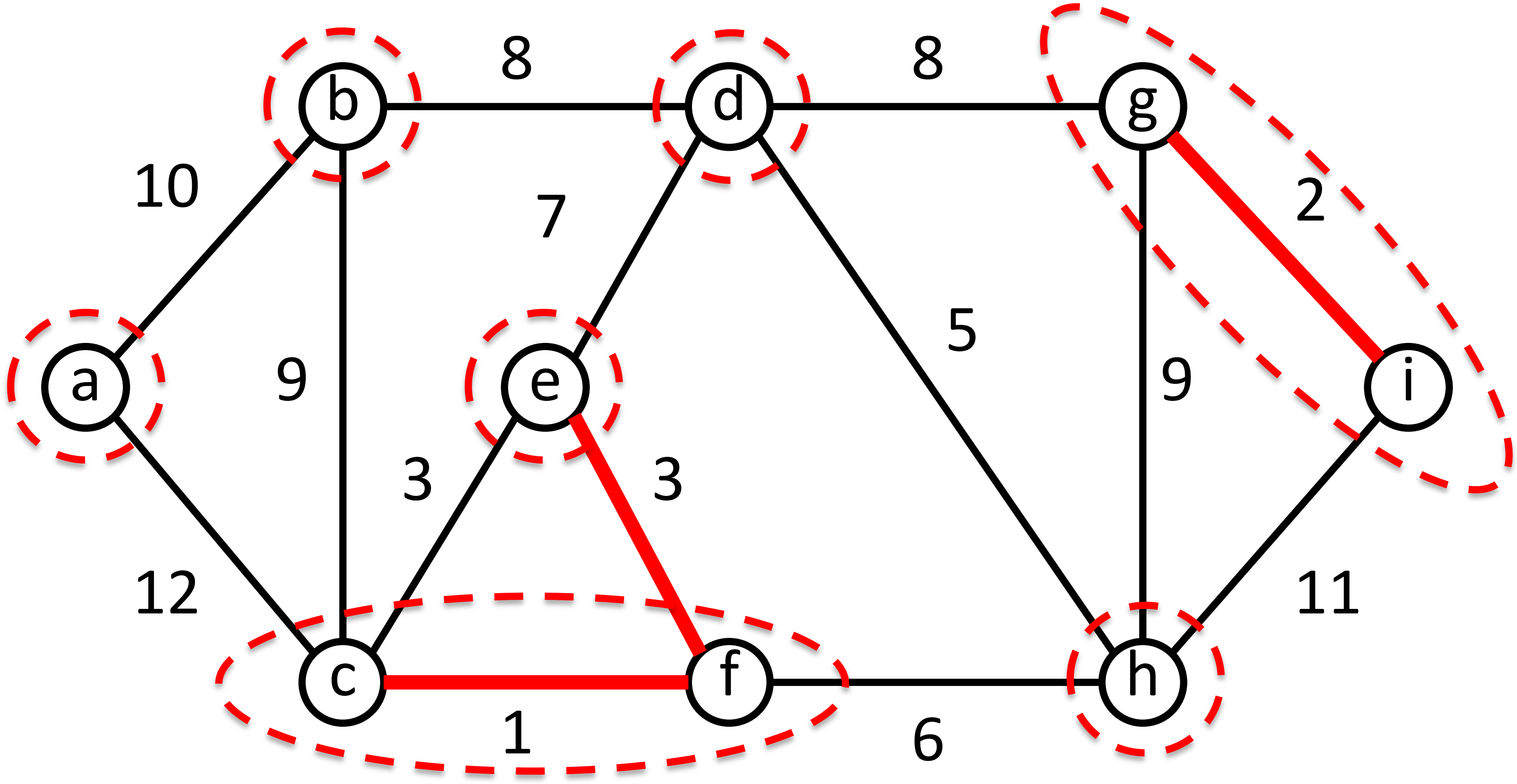
C and f are now merged into one component. From here on, we want to make sure at each step that the edge being added connects disconnected components.
Next edge inspected: g-i

Example

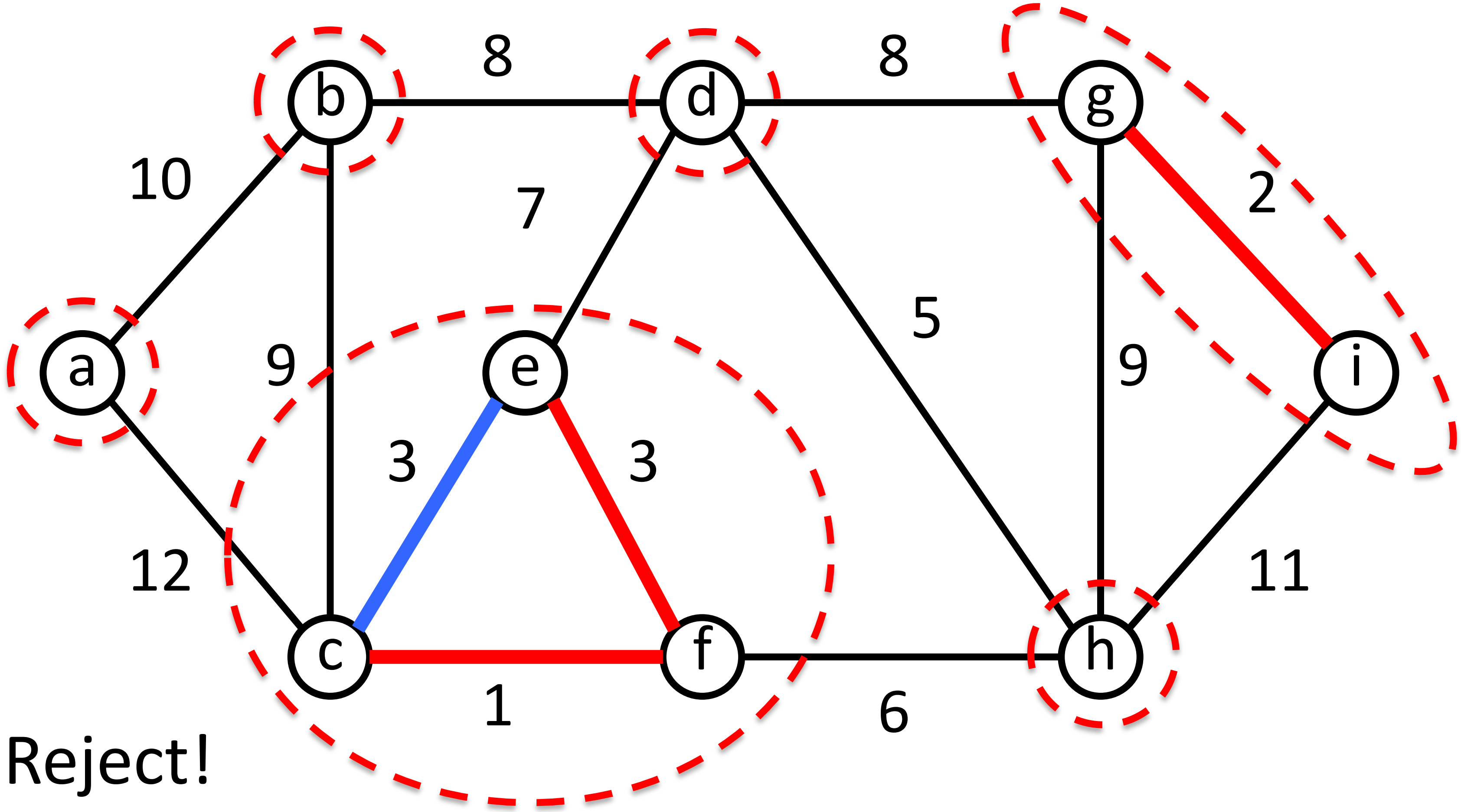


We continue by inspecting the lowest weight edges one by one and making sure we're safe

Example

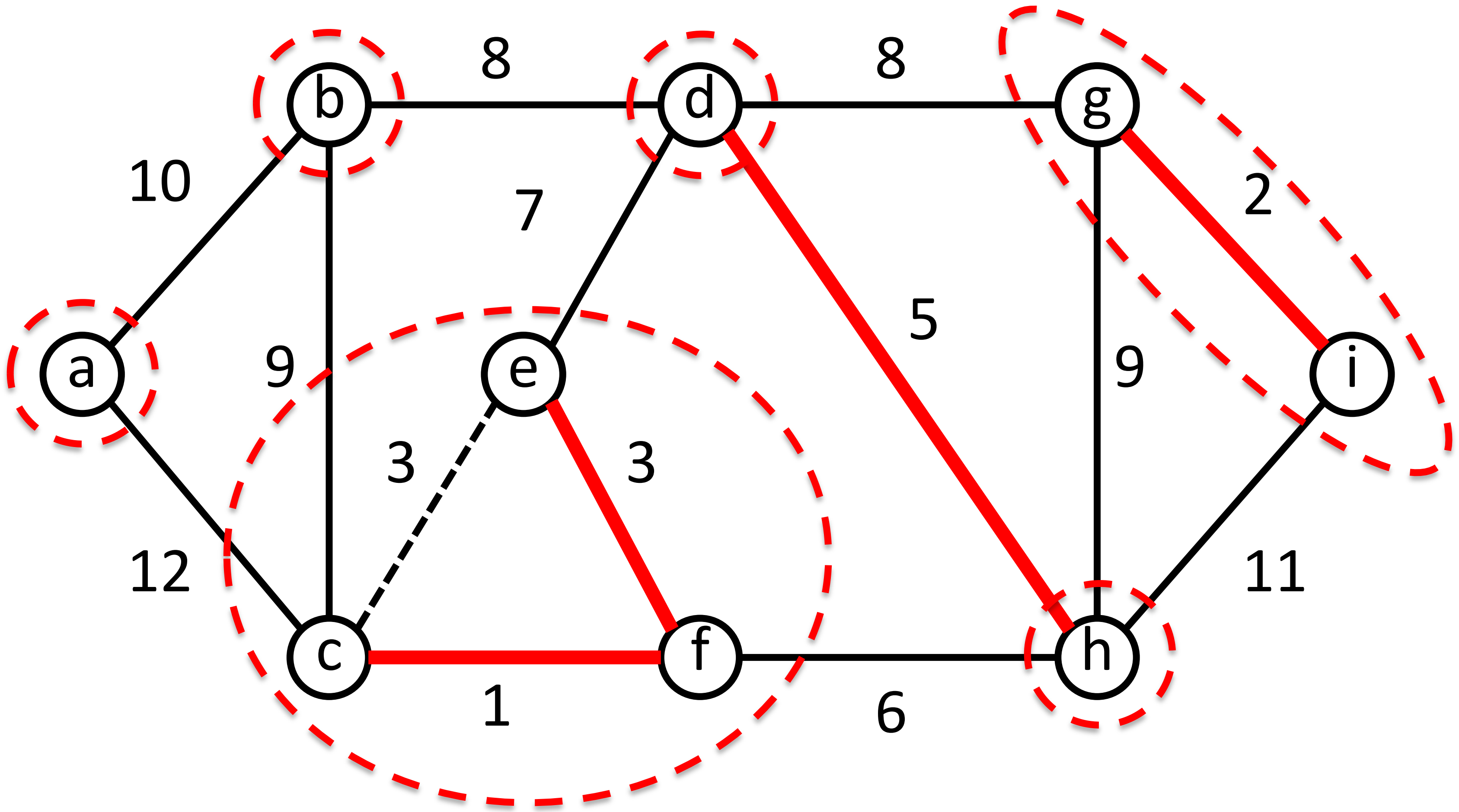


Example

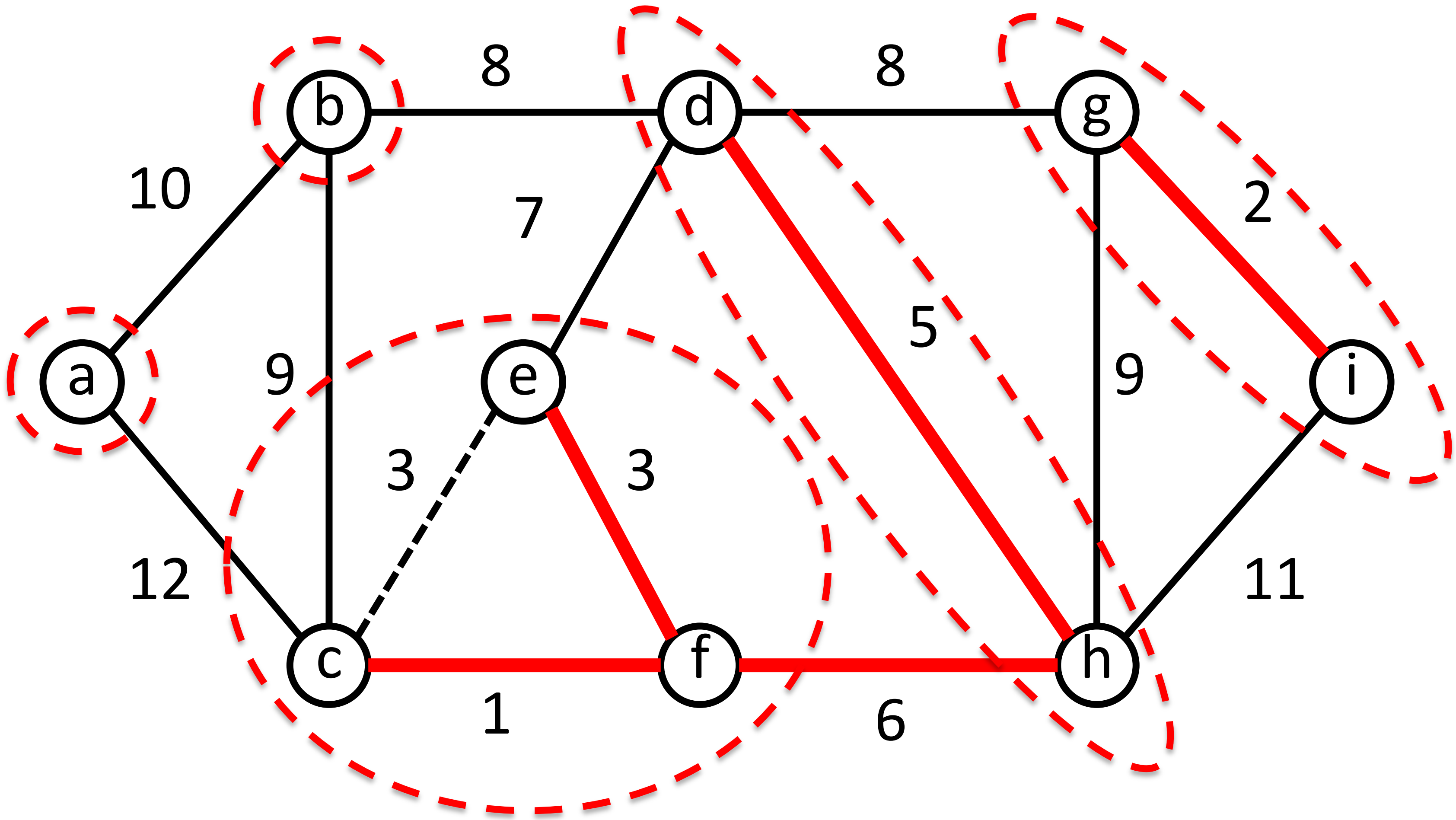


Reject!

Example

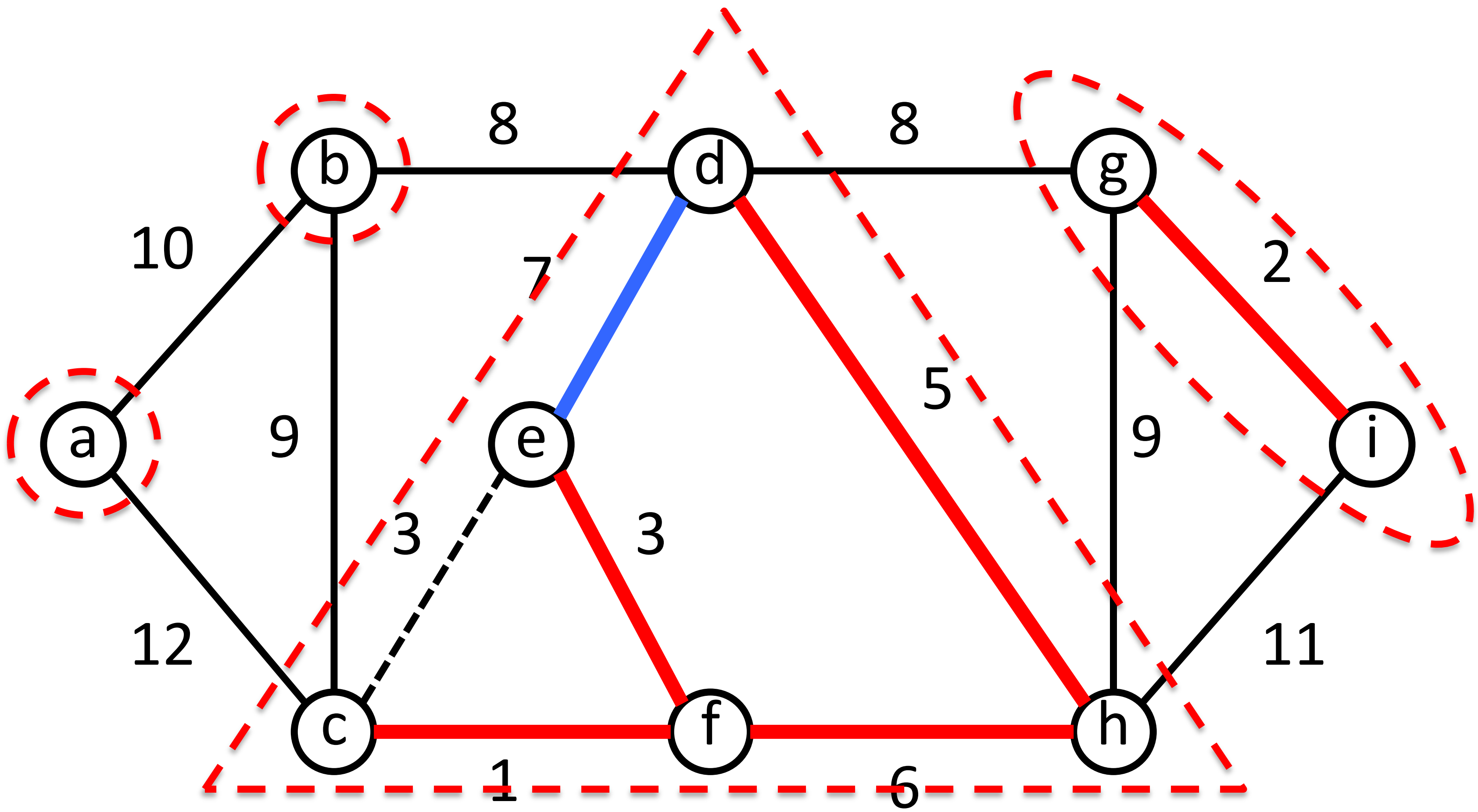


Example

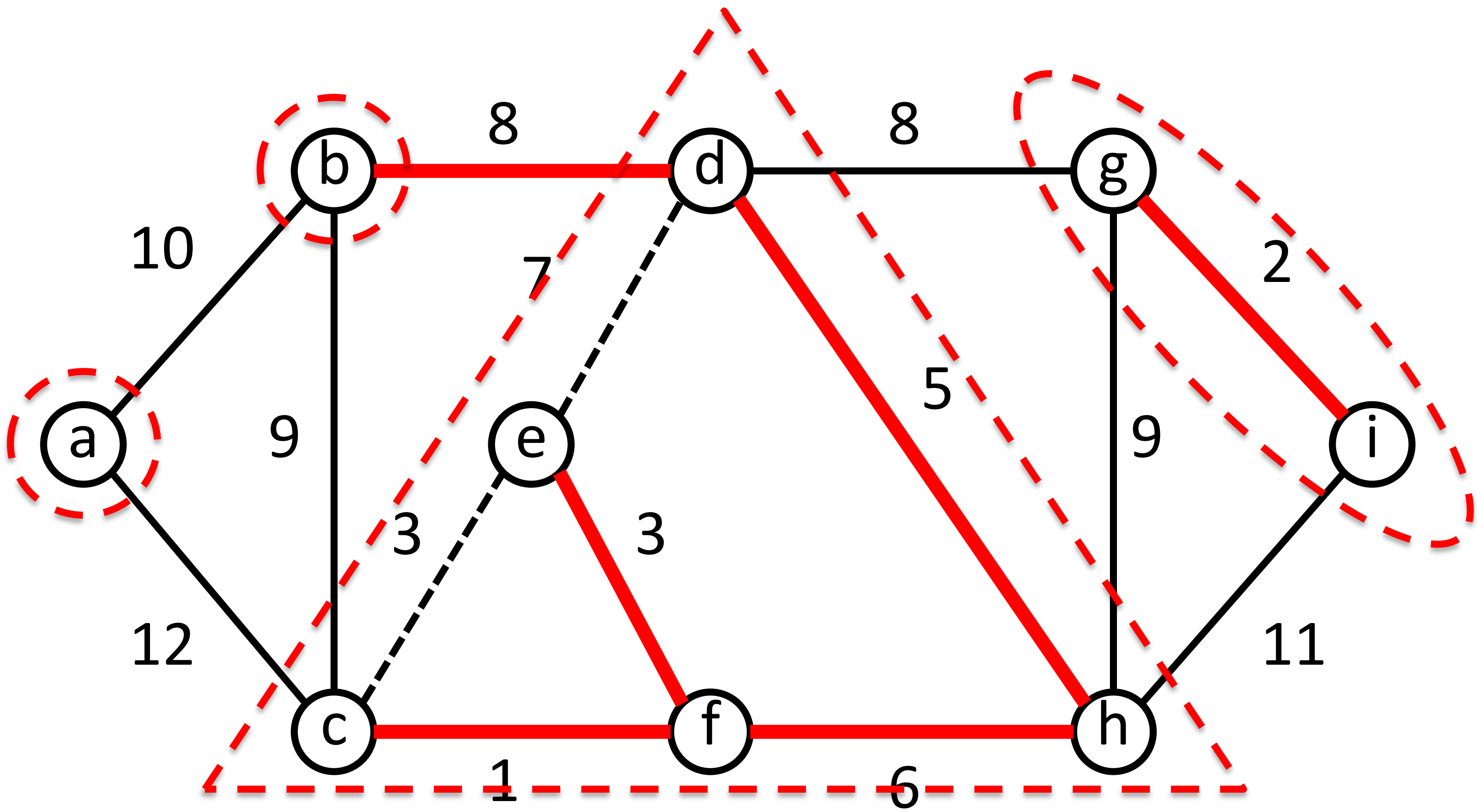


We keep merging components through light edges between them

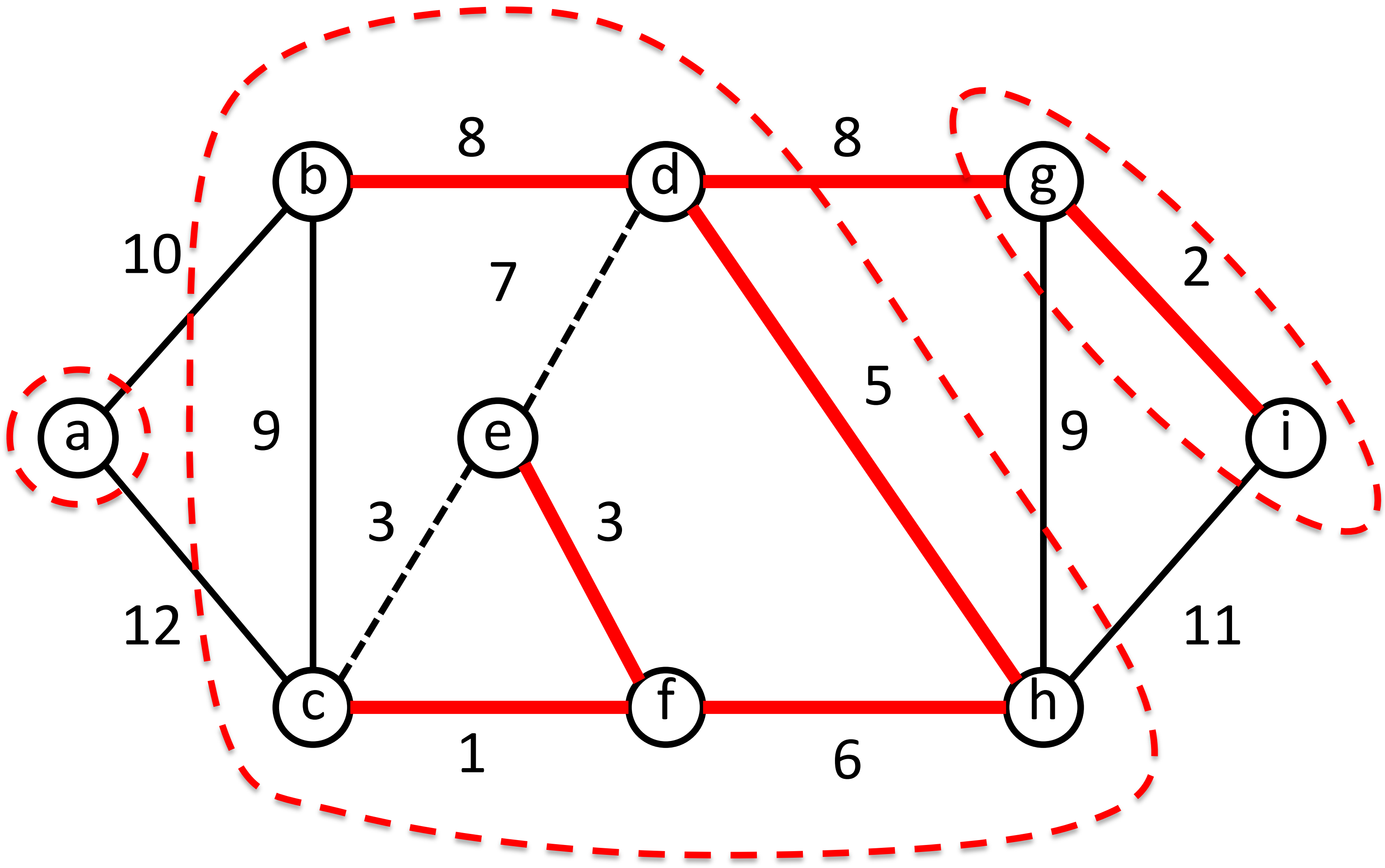
Example



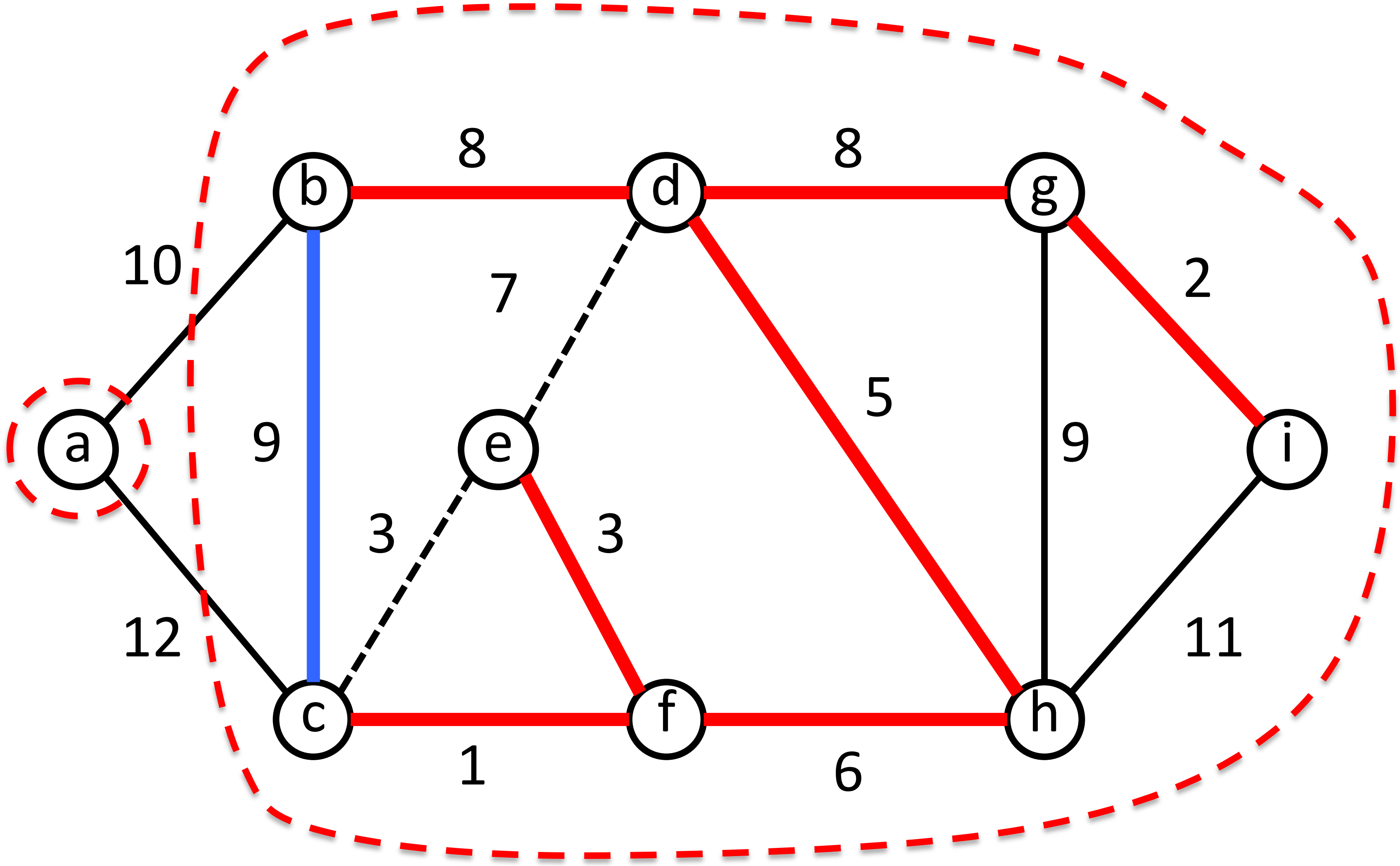
Example



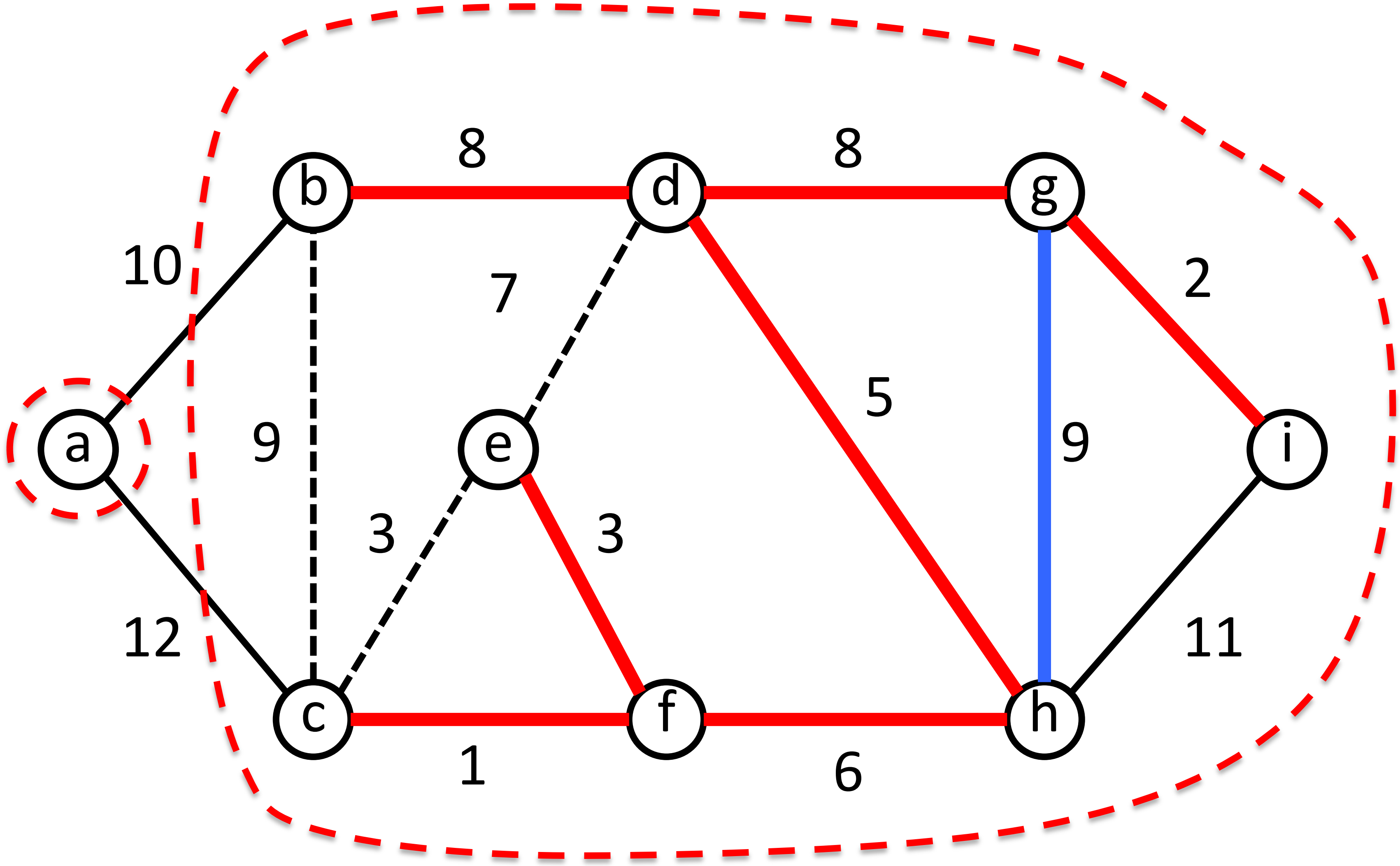
Example



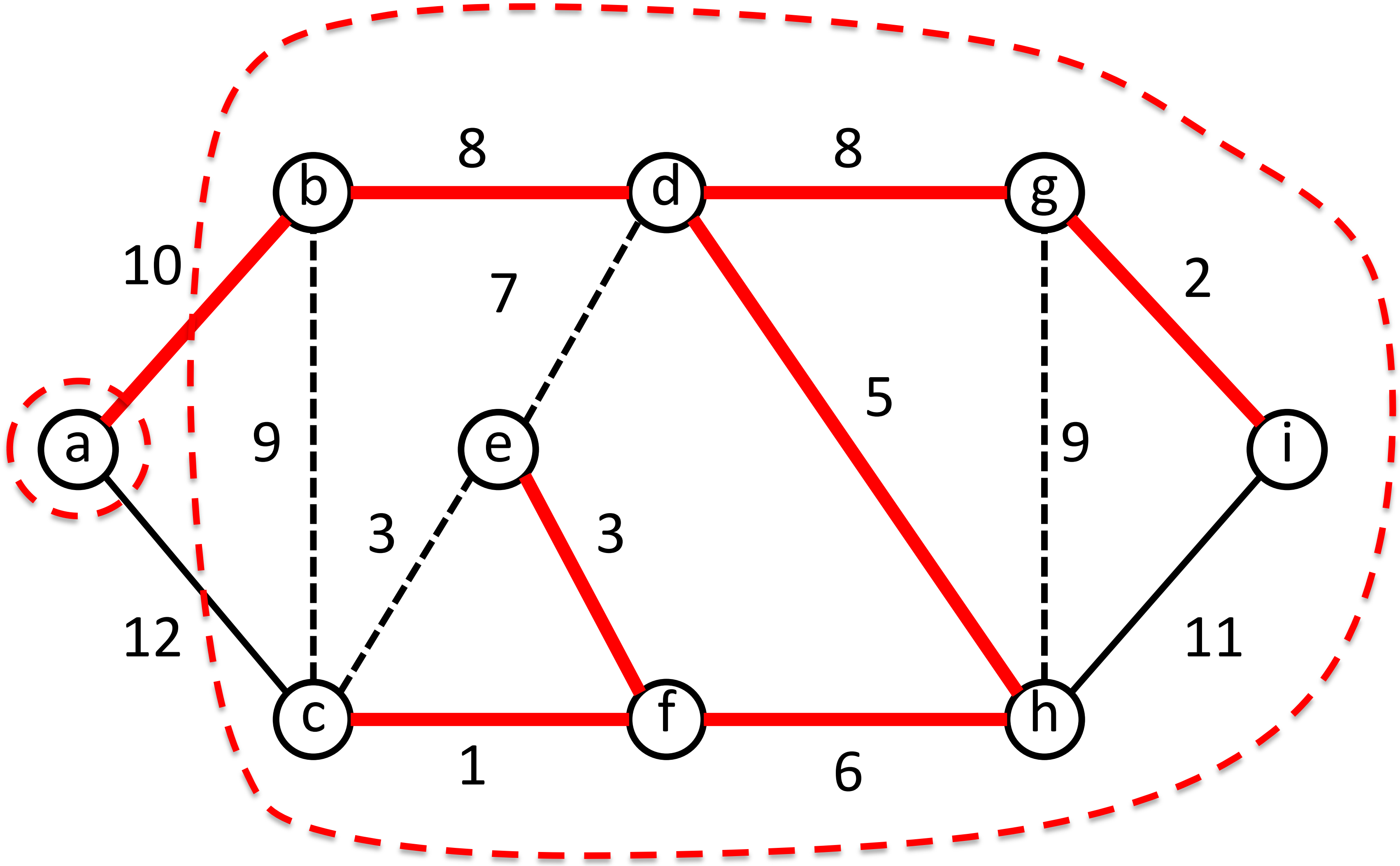
Example



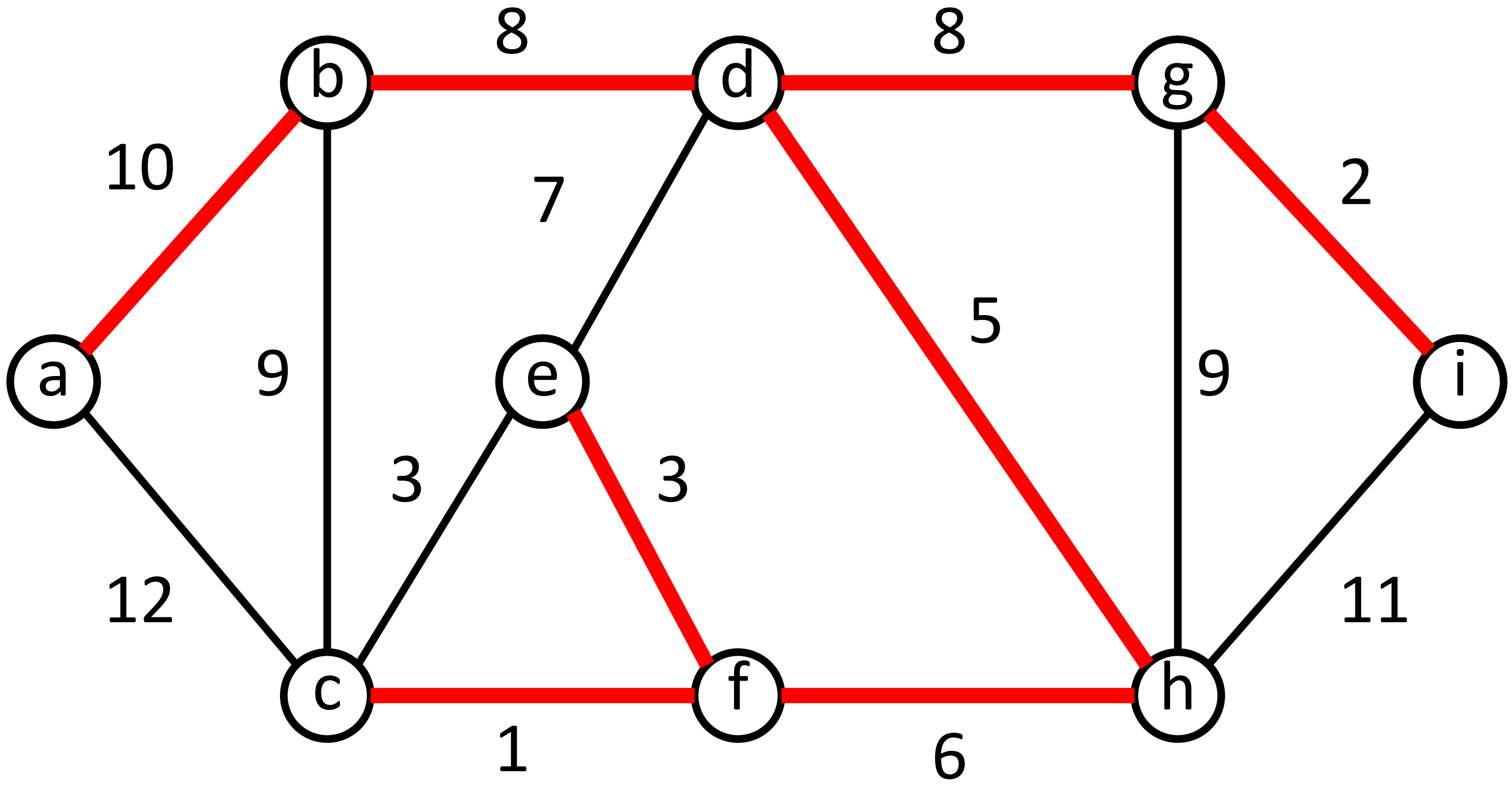
Example



Example



Example



Kruskal’s complexity

- Initialize A: $O(1)$
- First **for** loop: $|V|$ MAKE-SETs Define all vertices as disjoint sets
- Sort E: $O(E \lg (E))$ Sort all edges to determine order of visits
- 2nd **for** loop: $O(E)$ FIND-SETs and UNIONs ~E find and union operations

Assuming union by size and path compression, $\boldsymbol{m}$ find/union operations on a set with n objects is $O(m \cdot \alpha(n))$: See disjoint sets lecture!

$\Longrightarrow O(E \cdot \alpha(V)) + O(E \cdot \log(E))$

Moreover, $\alpha(V)$ *is* $O(\log V)$ *is* $O(\log E)$; $(|E| \geq |V| - 1)$

$\Longrightarrow O(E \cdot \log(E)) + O(E \cdot \log(E))$ *is* $O(E \cdot \log(E))$

Since, $|E| \leq |V|^2 \Longrightarrow \log|E|$ *is* $O(2 \log V)$ *is* $O(\log V)$

$\Longrightarrow \boldsymbol{O(E \cdot log(E))}$ *is* $\boldsymbol{O(E \cdot \log(V))}$

Both expressions are correct!

Prim's Algorithm

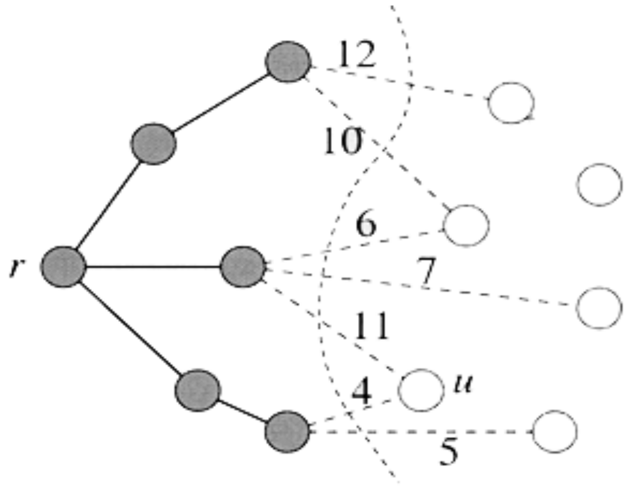
1. Builds **one tree**, so A is always a tree.
2. Starts from an arbitrary "root" r .
3. At each step, **adds a light edge** crossing cut (V_A, V - V_A) to A.
   - Where V_A = vertices that A is incident on.

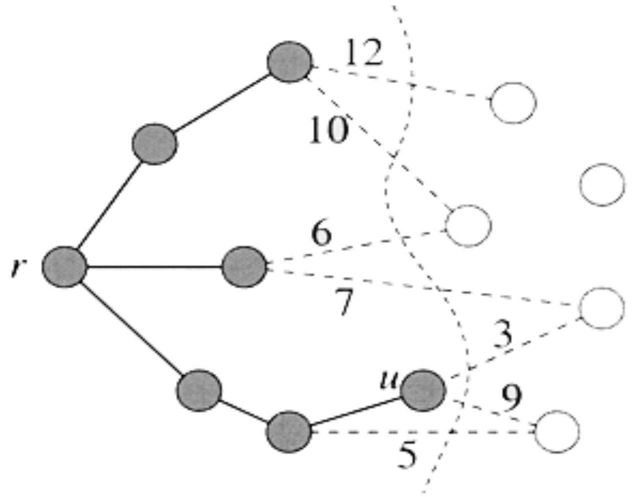
Kruskal was building the MST by assembling an acyclic set of edges

Prim's is actually directly building a tree

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!*





Basics of Prim 's Algorithm

- It works by adding leaves one 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?**

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 $\mathrm{V} - V_A$. The non-MST part
3. Key of v has minimum weight of any edge (u, v), where $u \in V_A$. Key of v is ∞ if v is not adjacent to any vertex 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 a light edge crossing $(V_A, V - V_A)$.

Intuition: we store all the vertices that have not been added to the MST in the queue. At each step, we use the priority queue to extract the node that has the light edge for the cut between the "resolved" region and the "unresolved" region, we add it to the tree via that edge, then we continue until the "resolved" region covers the whole graph.

Implementation: Priority Queue

- Priority queue implemented using heap can support the following operations in $O(lg\ n)$ time:
  - Insert (Q, v, key): Insert v with the key value key in Q
  - v = Extract_Min(Q): Extract the item with minimum key value in Q
  - Decrease_Key(Q, v, new_key): Decrease the value of v's key value to new_key

The key is the weight of v's edge crossing the cut

Need to update the keys when we update the cut

- All the vertices that are *not* in 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.

$$A = \{(v, \pi[v]):\ v \in V - \{r\}\}$$

Pi represents the parent of v. So this means that A contains all the edges between nodes and their parent, except r which does not have a parent.

Prim's Algorithm

Each vertex is extracted once by extract-min
For DecreaseKey we will have to update the value of every edge.

```
Q := V[G];
for each u ∈ Q do
    key[u] := $\infty$        Initialize tree by selecting random u
    $\pi$[u] := Nil;          Put every vertex in prioQ
    Insert(Q,u)               Run min-heapify V times
Decrease-Key(Q,r,0);          Run min-heapify ~E times
while Q $\neq \varnothing$ do
    u := Extract-Min(Q);
    for each v ∈ Adj[u] do
        if v ∈ Q $\wedge$ w(u, v) < key[v] :
            $\pi$[v] := u;
            Decrease-Key(Q,v,w(u,v));
```

Complexity:

Using binary heaps: O(E lg V).

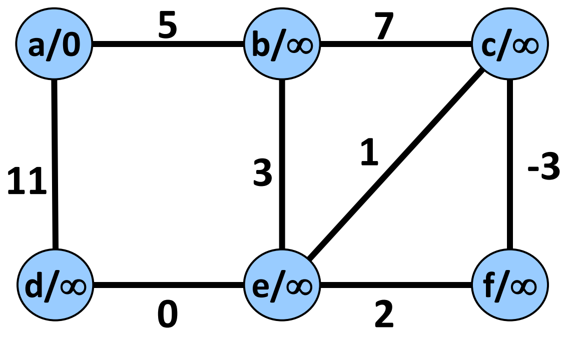
- Initialization: O(V).
- Building initial queue: O(V).
- V Extract-Min: O(V lg V).
- E Decrease-Key: O(E lg V).

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

This is a more advanced method

Notes: (i) A = {(v, π[v]) : v ∈ v - {r} - Q}. (ii) r is the root.

Example of Prim's Algorithm

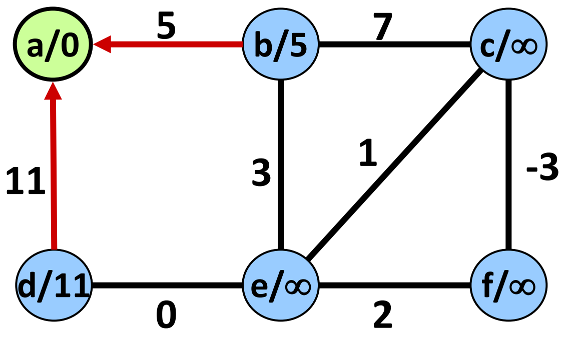


Not in tree

$$\overbrace{\boxed{\begin{array}{lcccccc} Q = & a & b & c & d & e & f \\ & 0 & \infty & \infty & \infty & \infty & \infty \end{array}}}$$

At the start, we determine the root is **a.** Until we add a to the tree, **a** is adjacent to r with w=0, but no other vertex is adjacent to r, so their w= inf

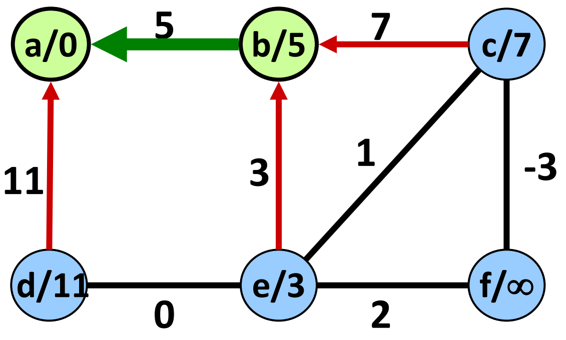
Example of Prim’s Algorithm



Q = b d c e f
5 11 ∞ ∞ ∞

We then add a to the tree, and decrease the key of nodes adjacent to a.

Example of Prim's Algorithm

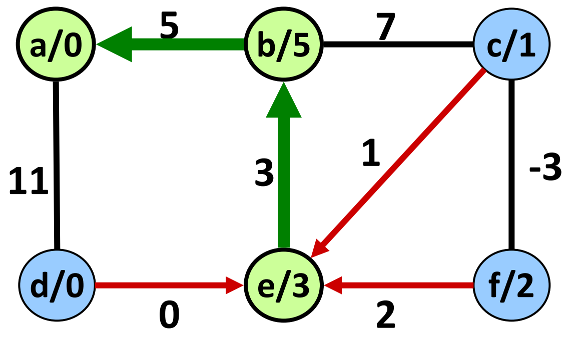


$$\boxed{\begin{array}{lllll} Q = & e & c & d & f \\ & 3 & 7 & 11 & \infty \end{array}}$$

We choose the lightest edge available, so extract-min returns b, then b is added to the tree and we reduce the key of the neighbors of b.

We use a dark green arrow to show parentality in the the minimum spanning tree.

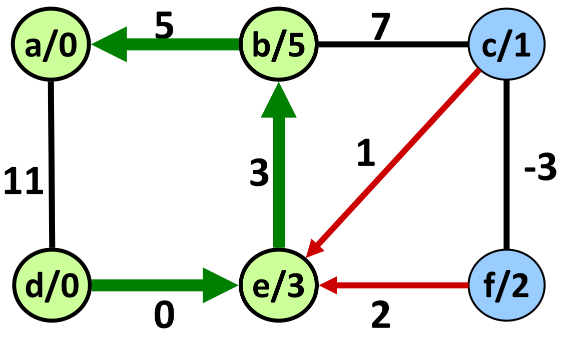
Example of Prim's Algorithm



Q = d c f
      0 1 2

We choose the lightest edge available, so extract-min returns e, then e is added to the tree and we reduce the key of the neighbours of e

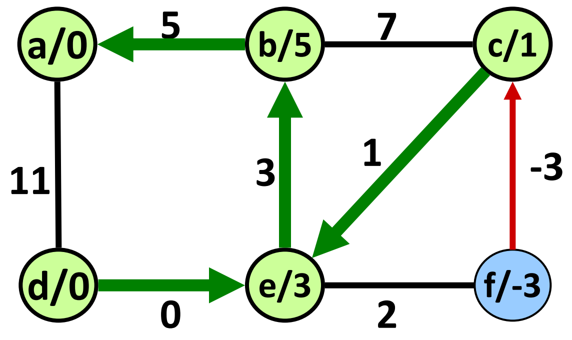
Example of Prim's Algorithm

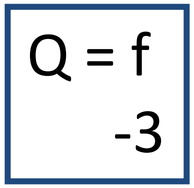


$$\boxed{\begin{array}{lll} Q = & c & f \\ & 1 & 2 \end{array}}$$

We choose the lightest edge available, so extract-min returns d, then d is added to the tree and we reduce the key of the neighbours of d, but all of d's neighbours have already been added to the tree. The priority queue only includes nodes that are not in the tree yet.

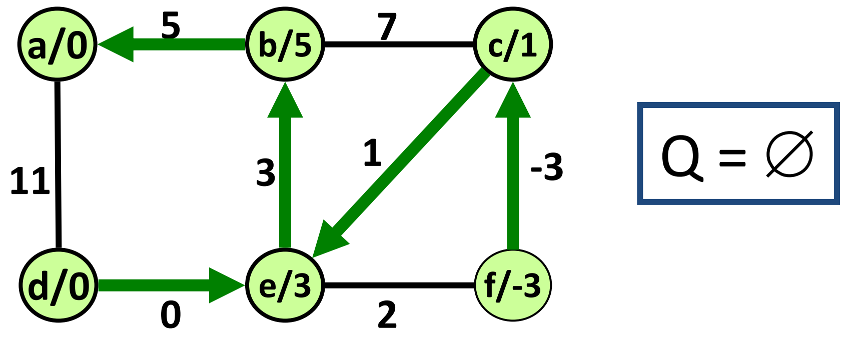
Example of Prim's Algorithm





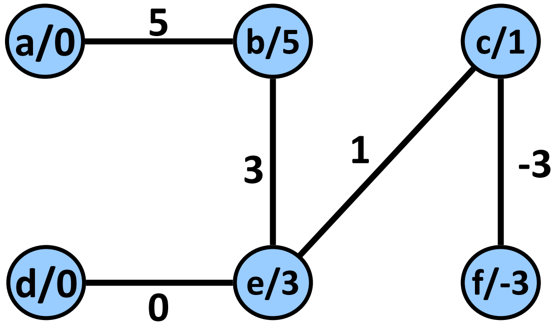
We choose the lightest edge available, so extract-min returns c, then c is added to the tree and we reduce the key of the neighbors of e based on this new information. In this case, this has the result of updating the weight of f in the queue, since c-f has lower weight than e-f.

Example of Prim's Algorithm



f is added to the tree, the queue is now empty, thus the algorithm terminates. The tree spans the whole graph; it is a minimum spanning tree.

Example of Prim's Algorithm



Correctness of Prim’s

- 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.

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.

We proved this earlier!

Time Complexity of Prim's

- Initialization: $O(V+E)$
- Extract the light edge from the queue: $\mathrm{O}(\mathrm{V}\log V)$ See earlier slides
- Relax the neighbour edges: $\mathrm{O}(E\log V)$

$\Longrightarrow O(V\log V+E\log V)=O(E\log V)$ // same as Kruskal

Intuition: there are typically more edges than vertices

Note: Using Fibonacci heaps, we can obtain $O(E+\log V)$.

We do not cover Fibonacci heaps in this class, but basically you should know that while the "traditional" method we show for Kruskal and Prim's is "decently fast", there exist faster methods, namely with Fibonacci heaps.