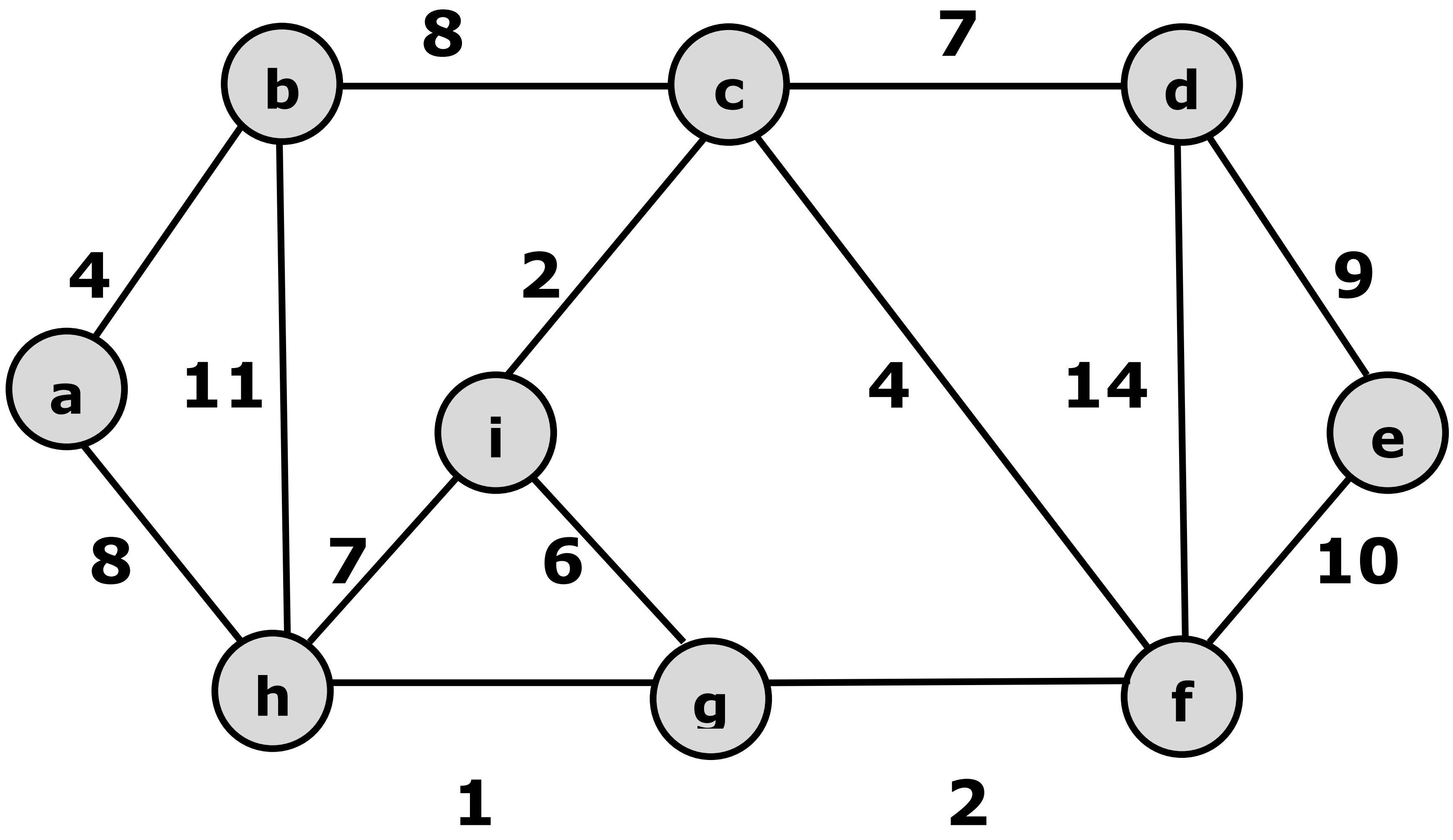
Kruskal algorithm

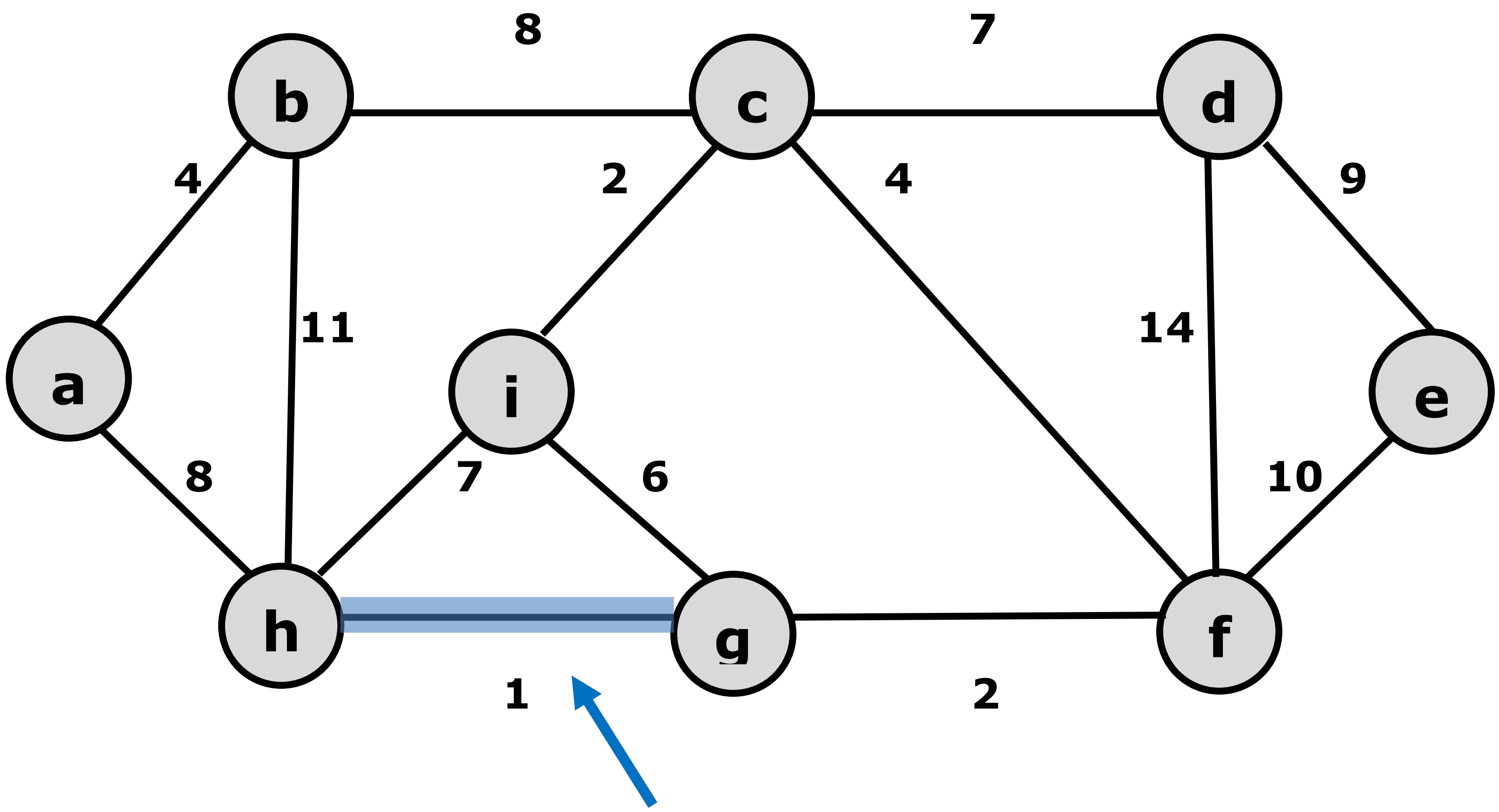
Graph G = (V,E) w : E →R

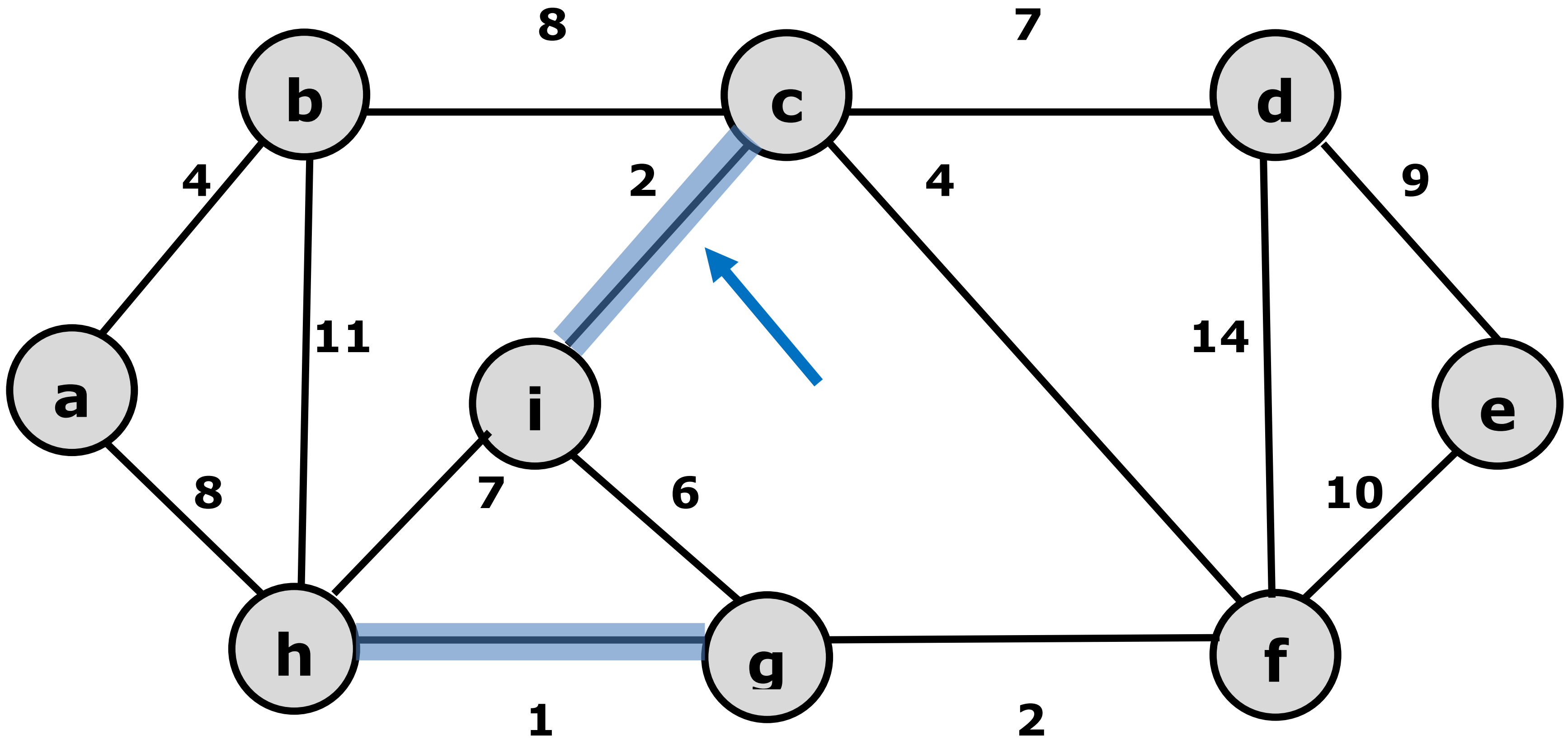


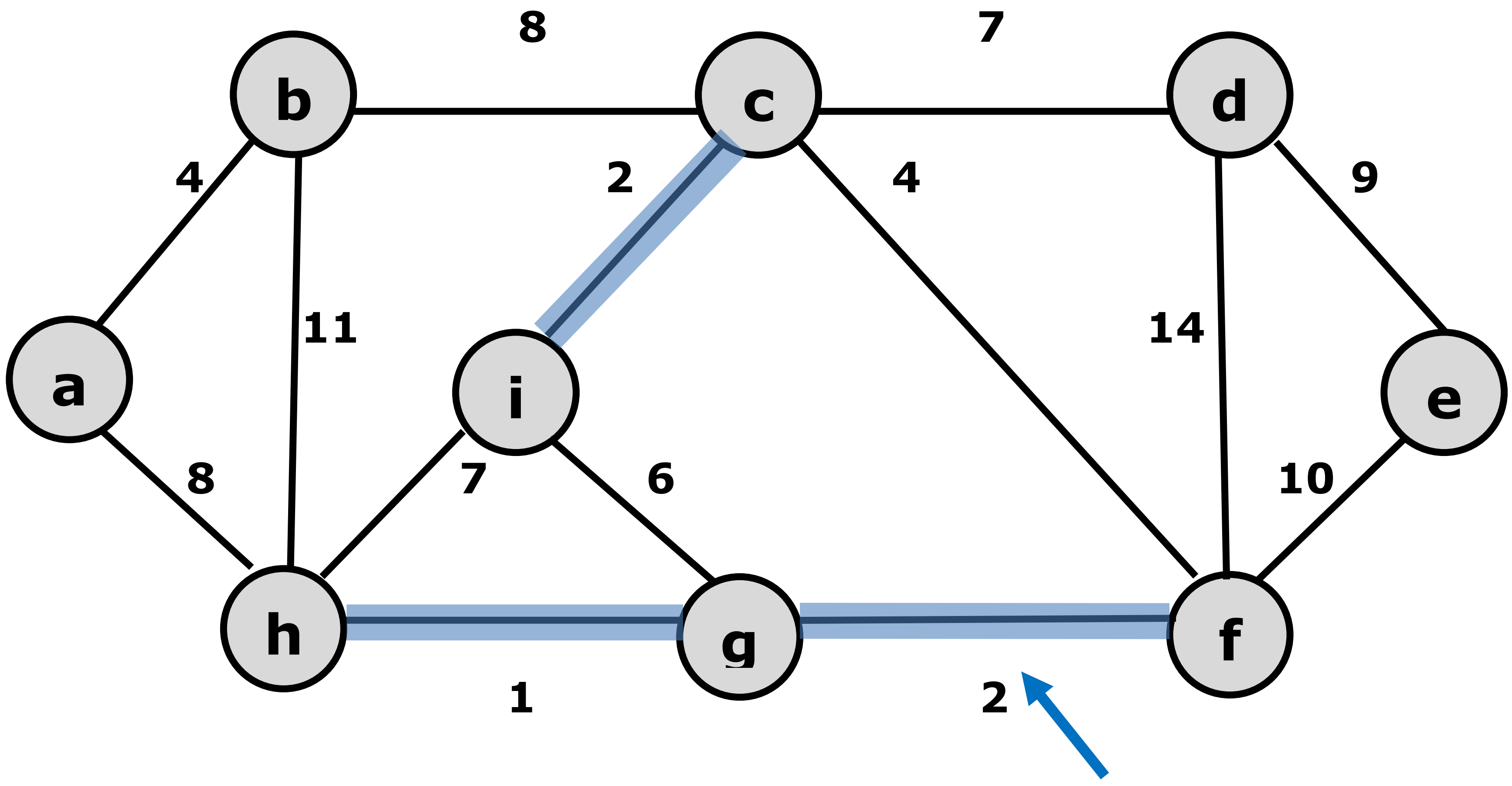
Build MSP (minimum spanning tree)

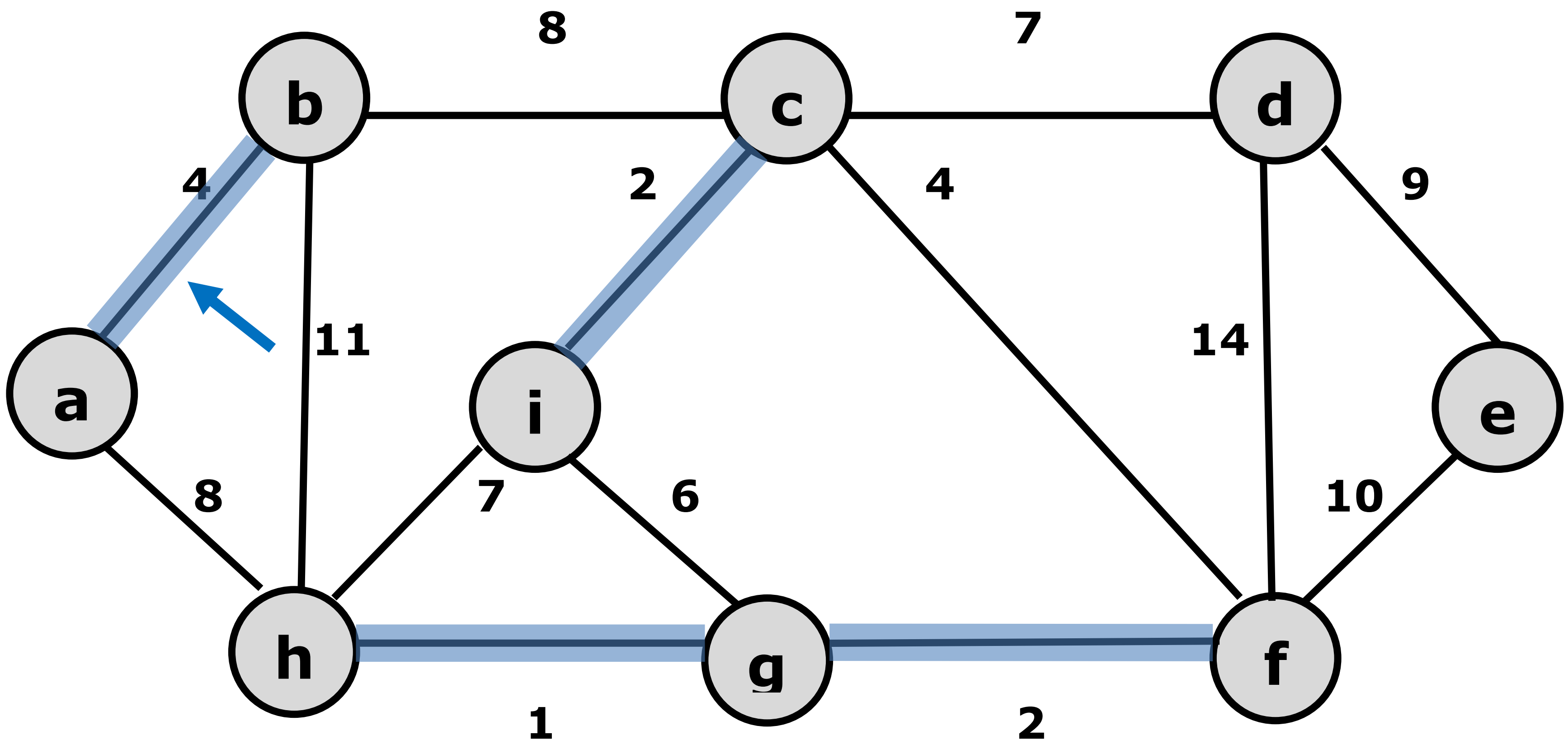
Sort edges for weight

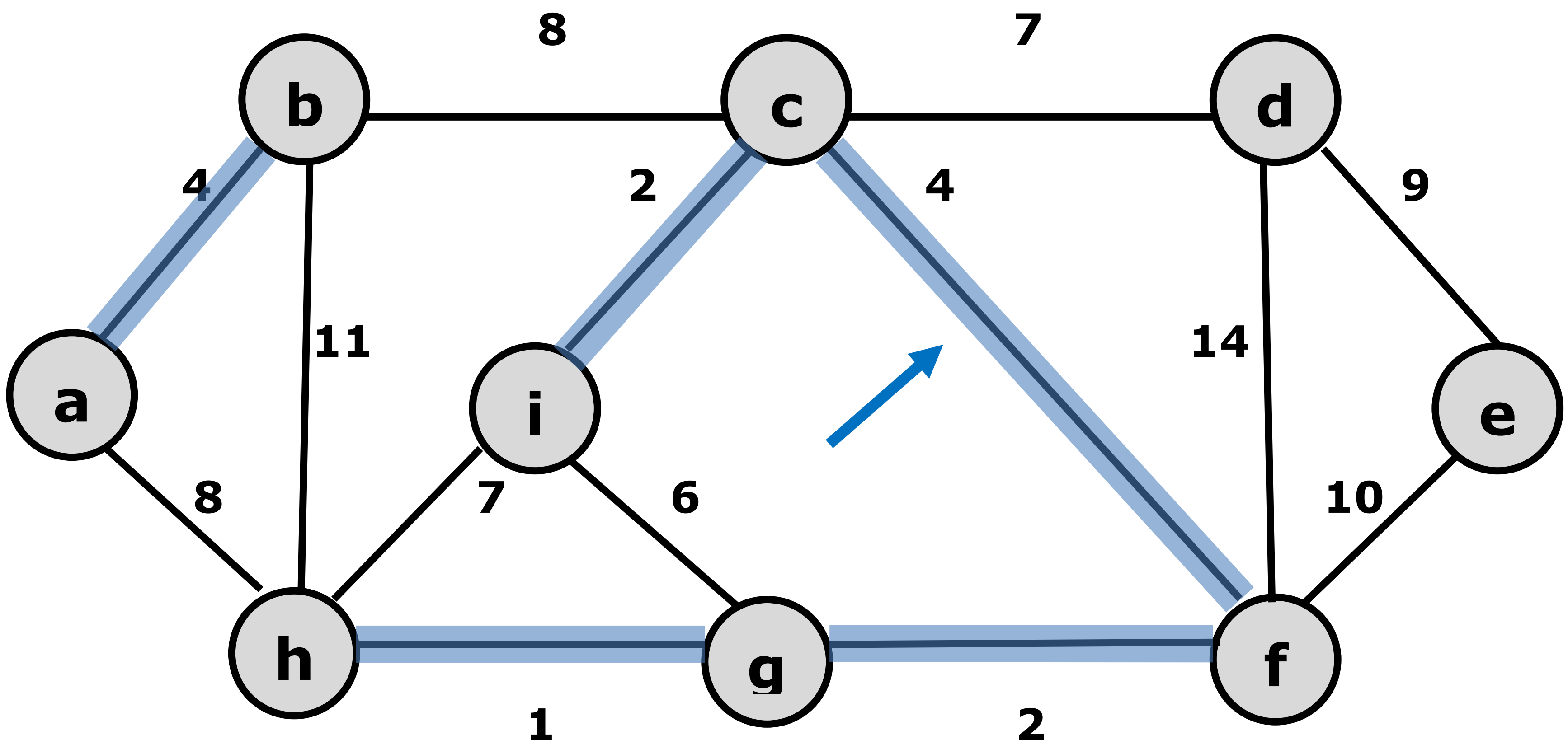
##	edge	weight
1	h-g	1
2	g-f	2
3	i-c	2
4	a-b	4
5	c-f	4
6	i-g	6
7	c-d	7
8	h-i	7
9	b-c	8
10	a-h	8
11	d-e	9
12	e-f	10
13	b-h	11
14	d-f	14


8
7
b
c
d
4
2
4
9
11
14
a
i
e
8
7
6
10
h
g
f
1
2

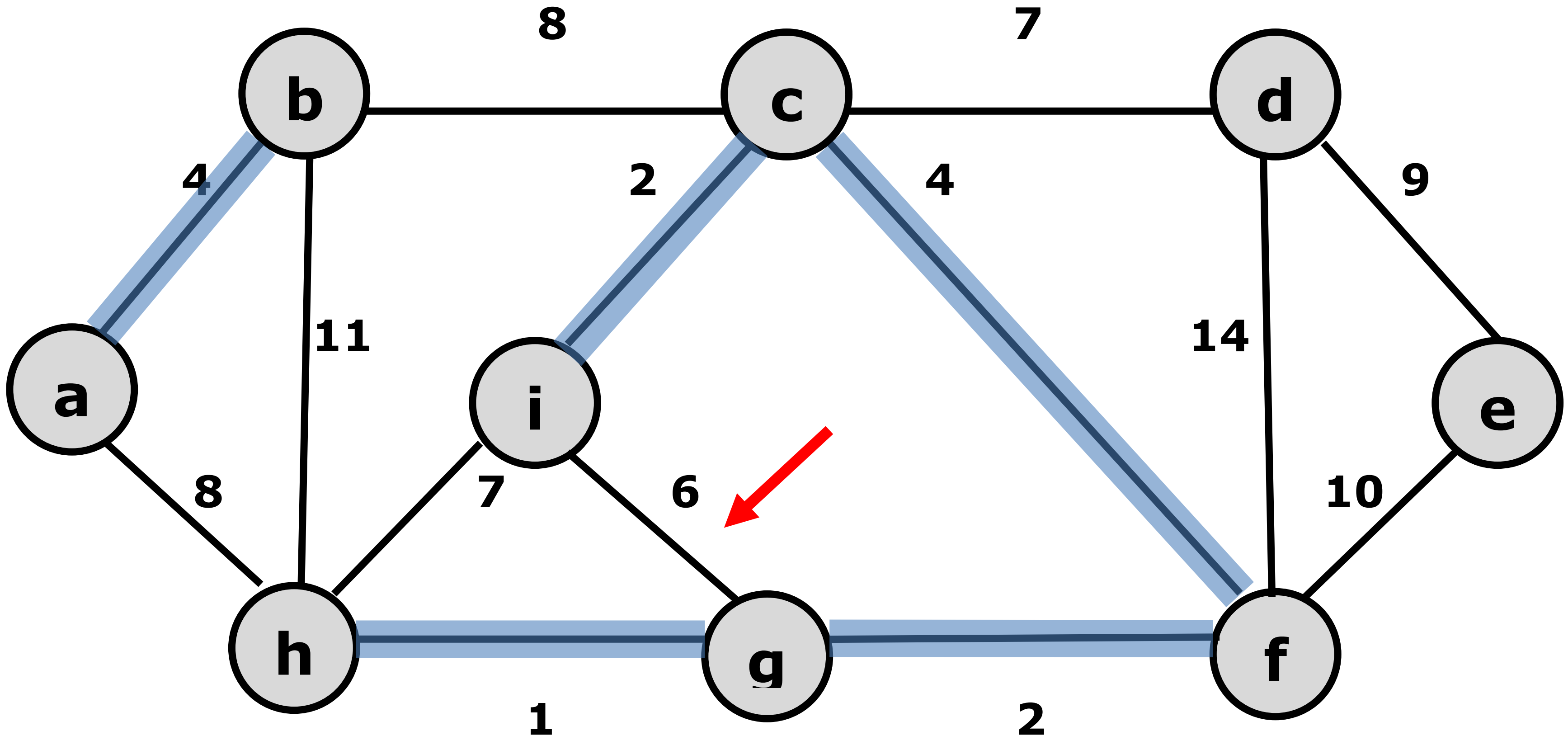

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
9
14
10
7
6
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

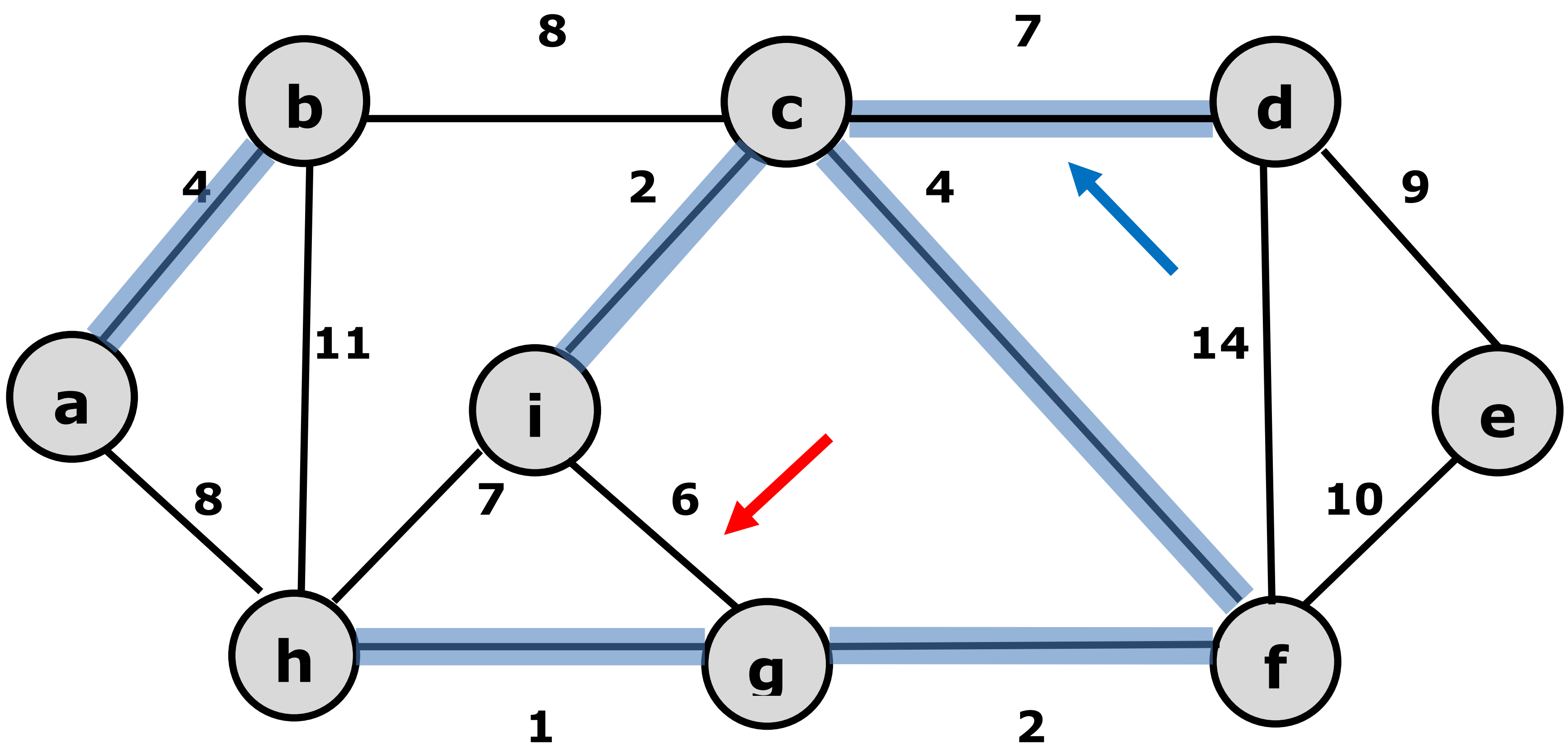

a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
8
7
6
10
1
2

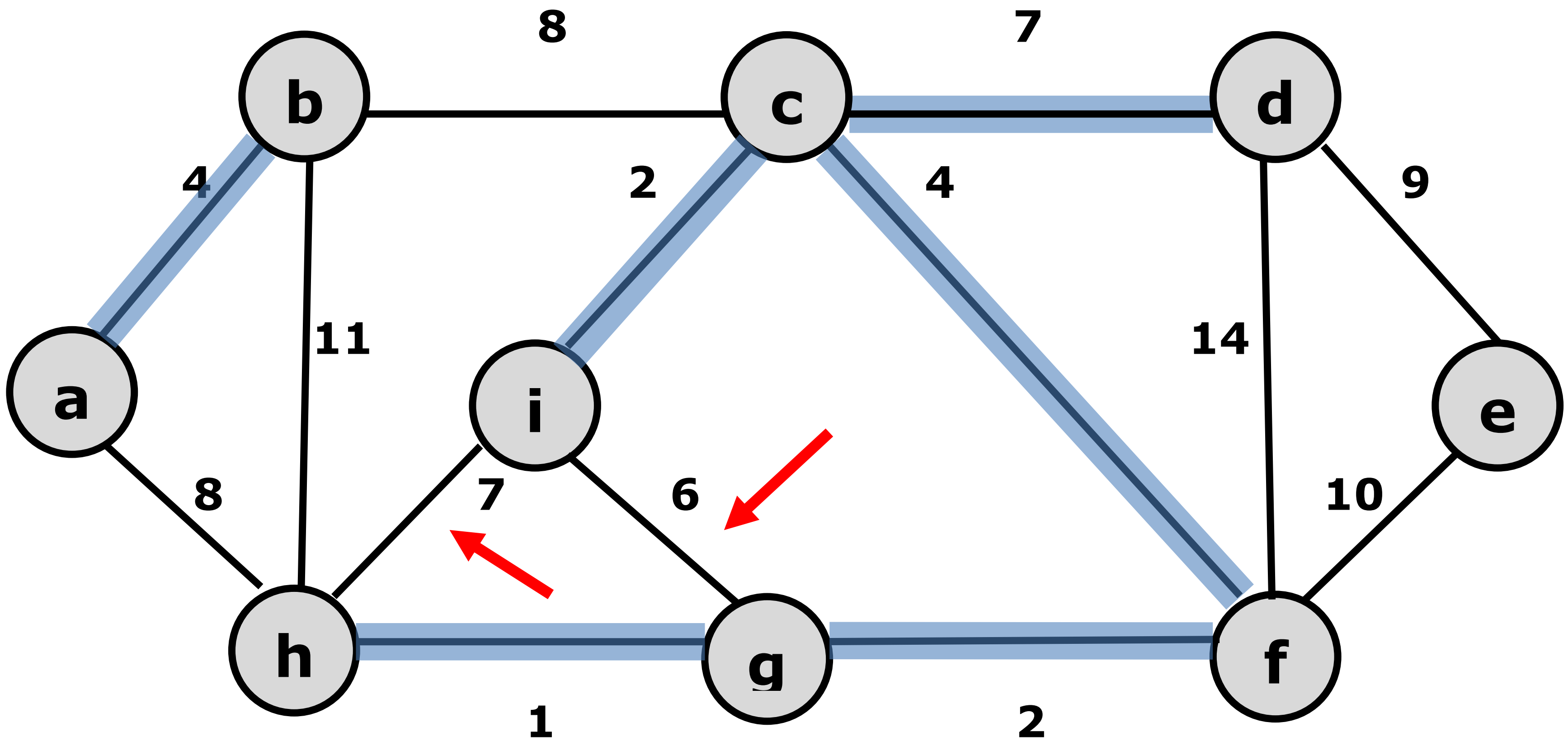


8
7
b
c
d
4
2
4
9
11
14
a
i
e
8
7
6
10
h
g
f
1
2

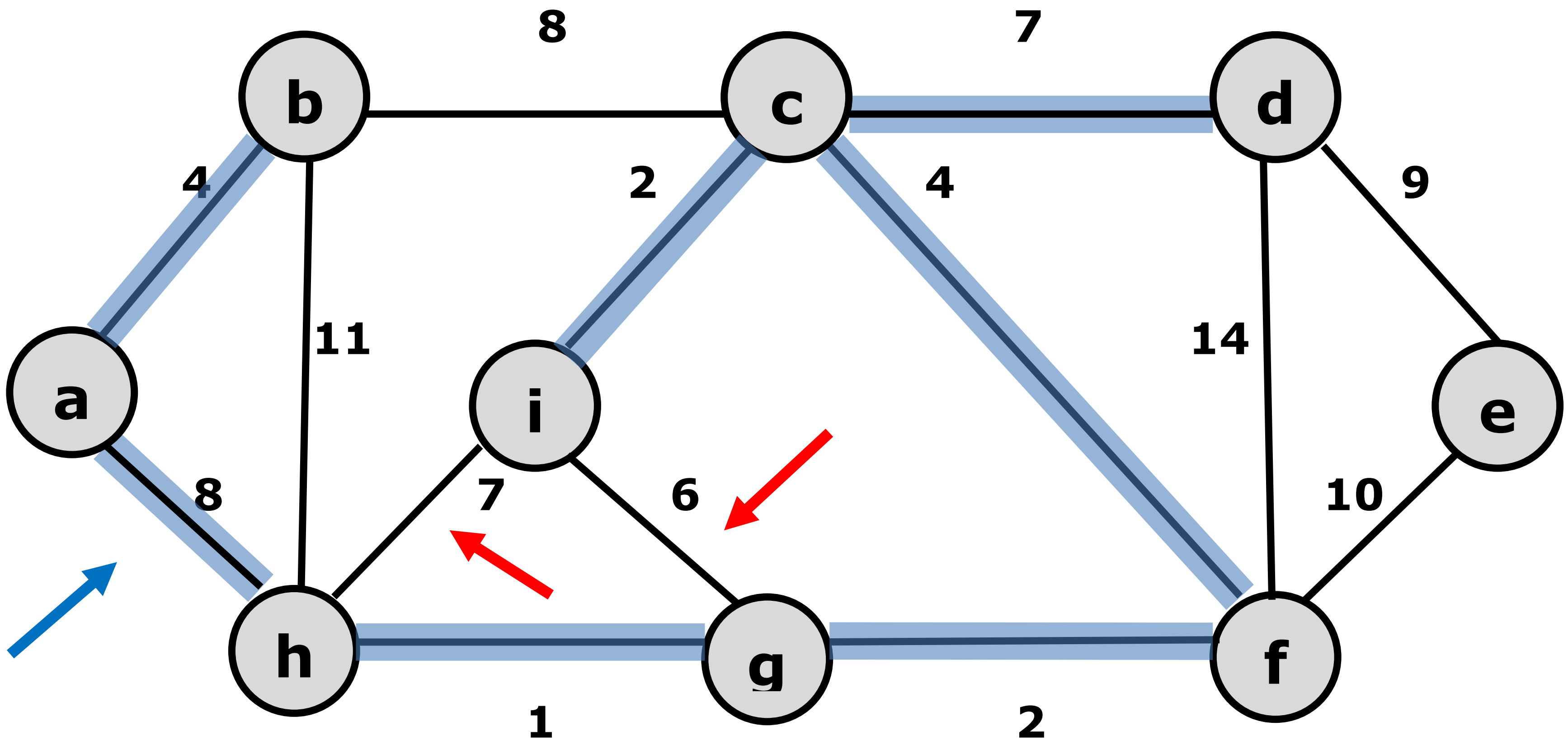


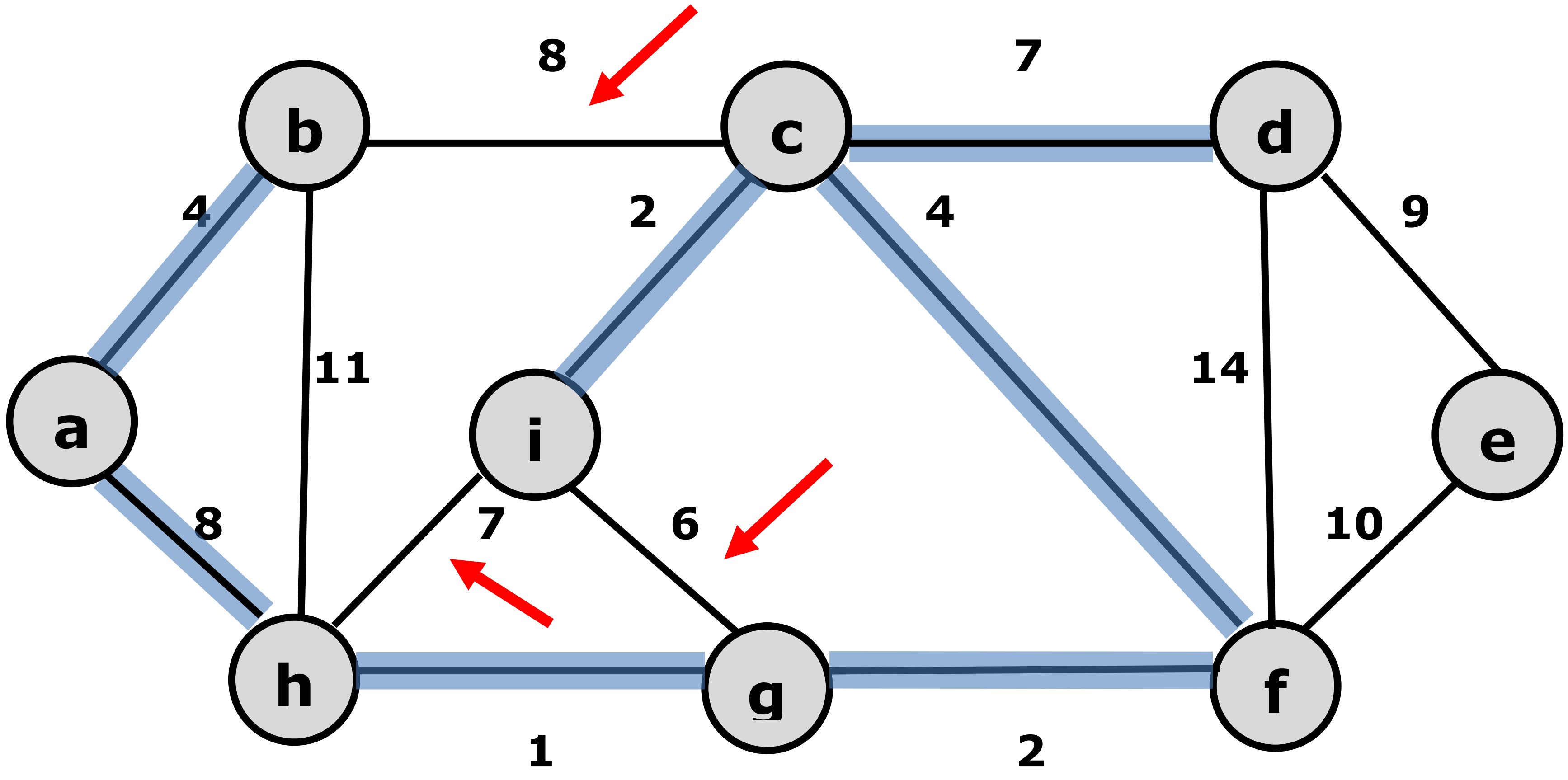
Loop : c-f-g-i-c


a
b
c
d
e
f
g
h
i
8
7
4
2
4
9
11
14
8
7
6
10
1
2

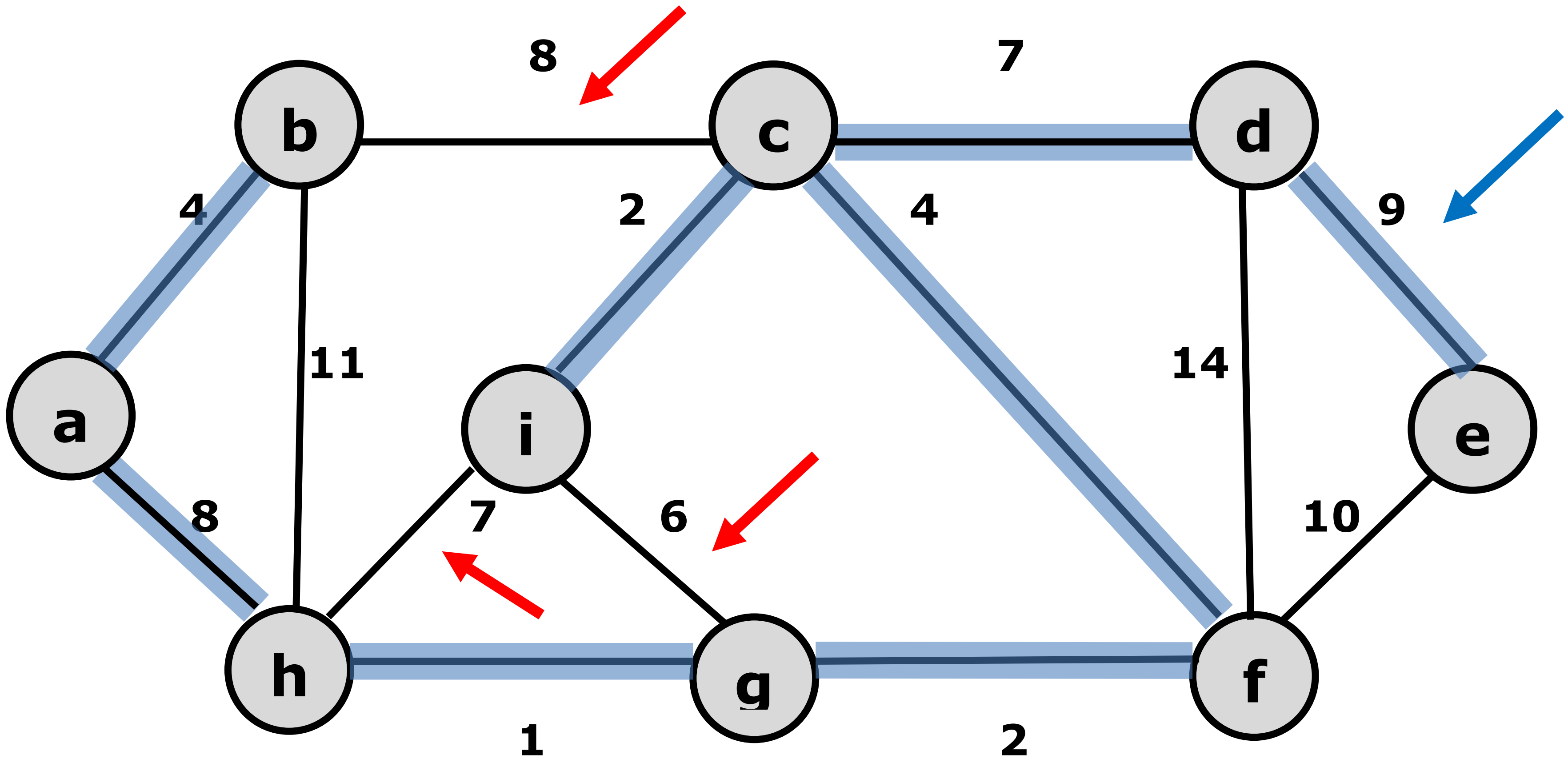


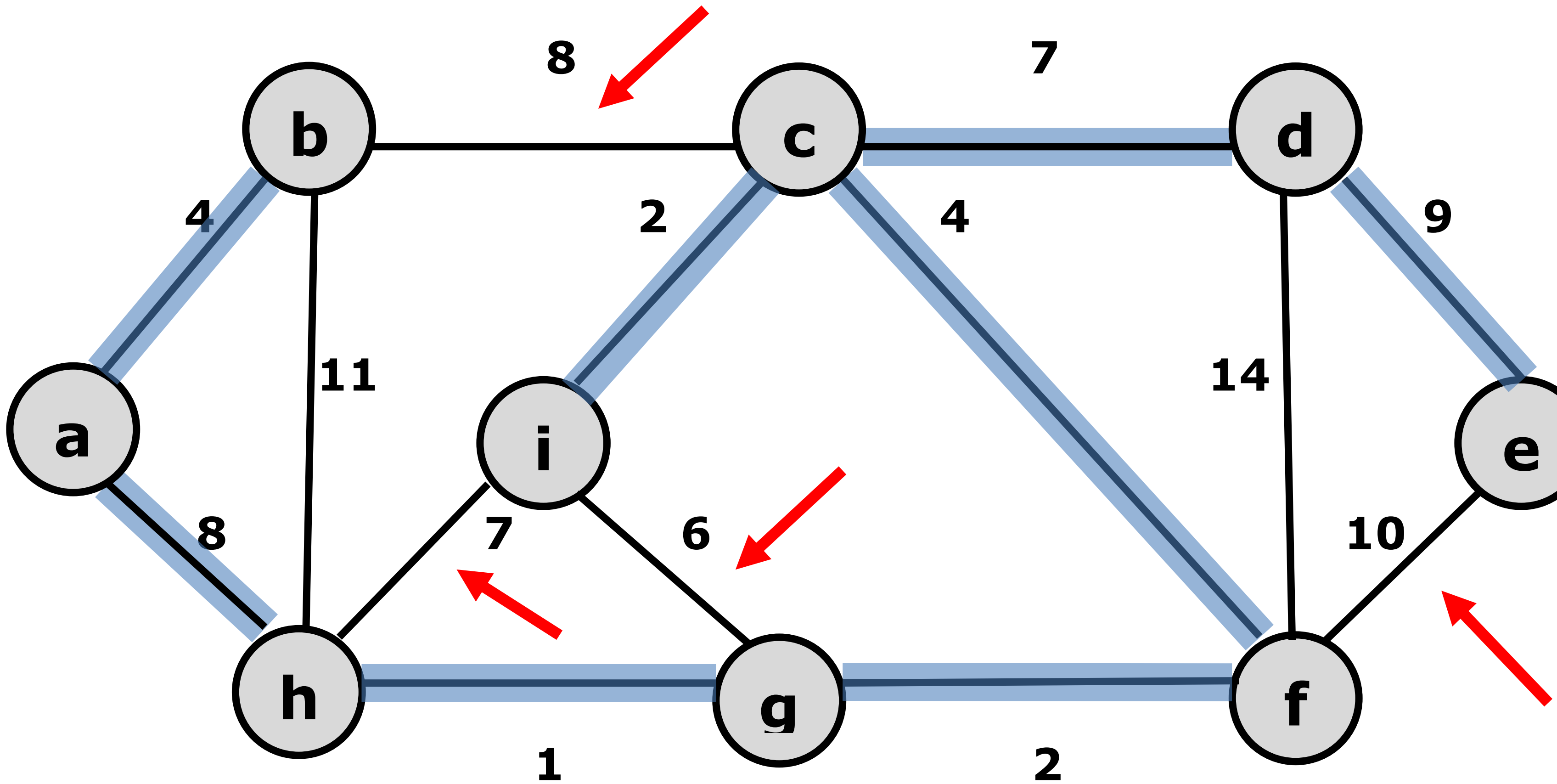
Loop : i-g-f-c-i


a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
8
7
6
10
1
2

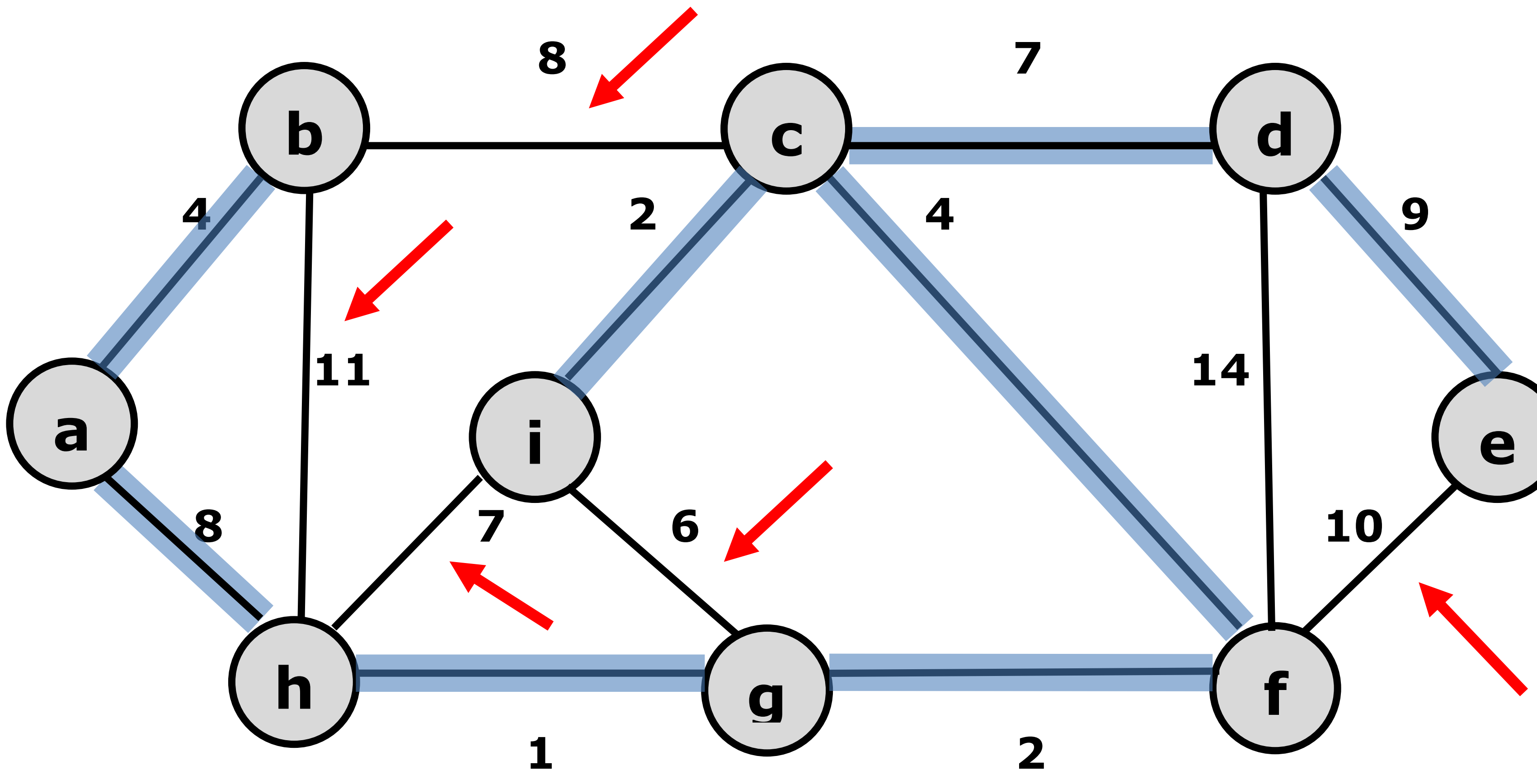


Loop : b-c-f-g-h-a-b

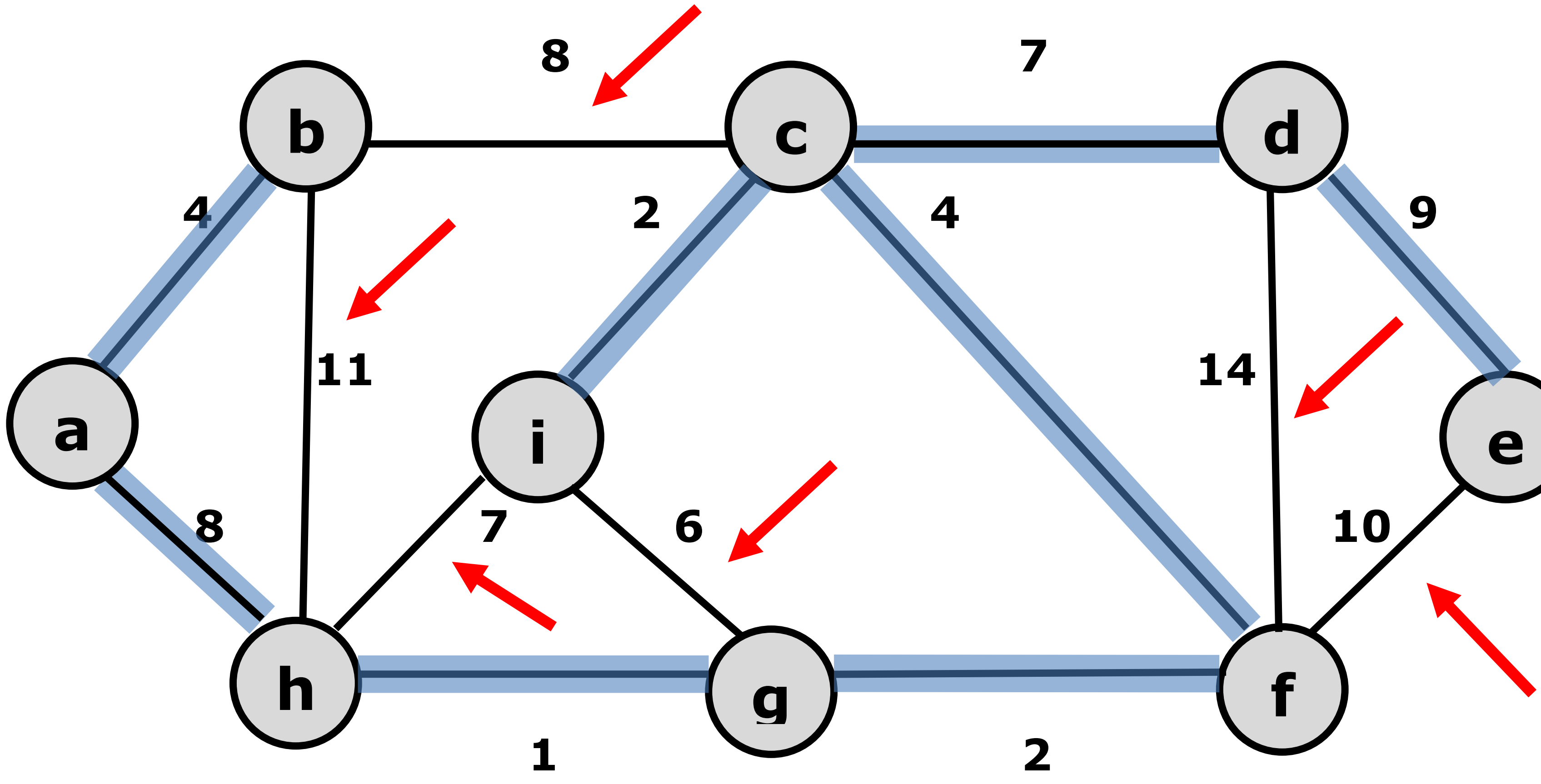

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
9
14
10
7
6
1
2



Loop : c-d-e-f-c

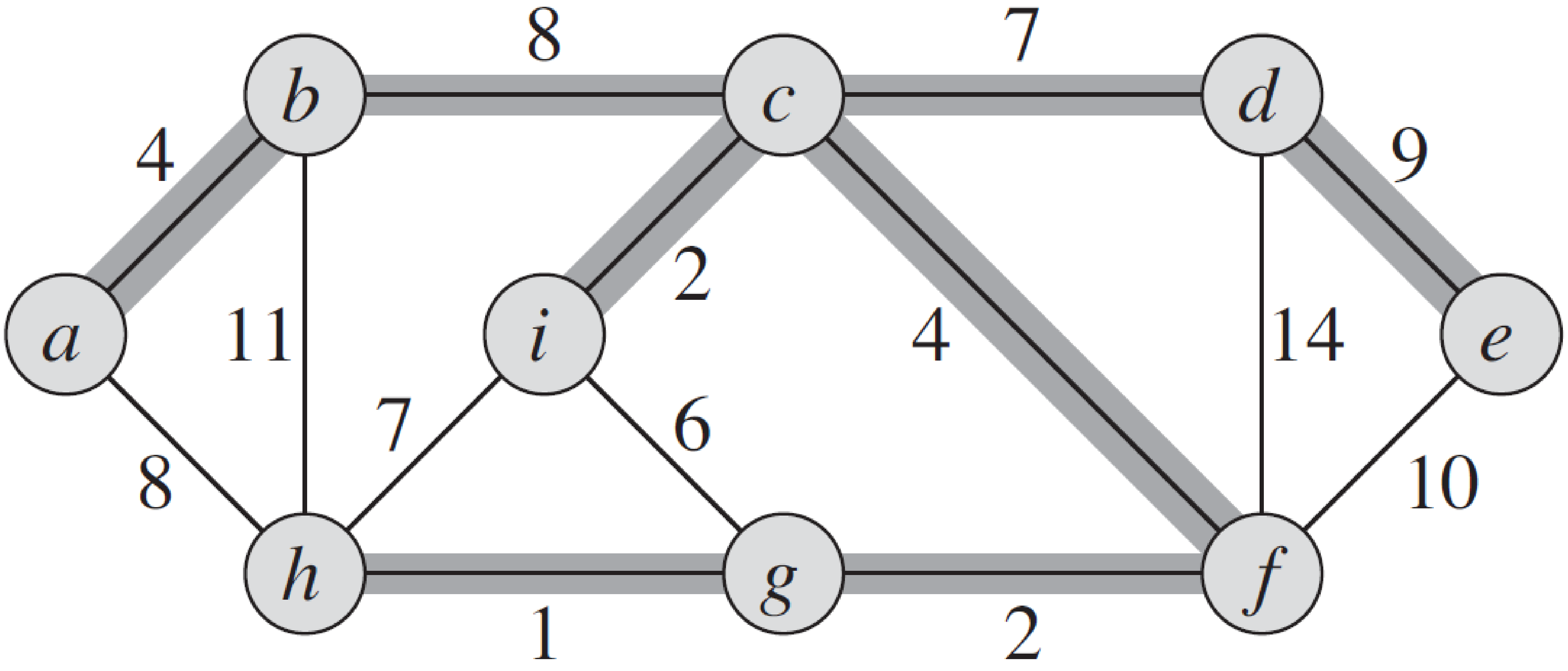


Loop : a-b-h-a



Loop : c-d-f-c

Growing a minimum spanning tree



class Node

```
package kruskal;

public class Node{


    int node;
    int weight;


    public Node(int node, int weight){
        this.node = node;
        this.weight = weight;
    }


    public String toString(){
        return "node: " + node + ", weight: " + weight;
    }
    public static void main(String[] args) {

        Node node1 = new Node(1,14);
        Node node2 = new Node(2,8);
        Node node3 = new Node(3,11);

        System.out.println("Node1 : " + node1.toString());
        System.out.println("Node2 : " + node2.toString());
        System.out.println("Node3 : " + node3.toString());
    }

}
```

Result :

```
Node1 : node: 1, weight: 14

Node2 : node: 2, weight: 8

Node3 : node: 3, weight: 11
```

class Edge

```
package kruskal;

/**
   The Edge class represents an edge: (A,B) with the weight
*/
public class Edge implements Comparable<Edge>{

    private int vertexA, vertexB;
    private int weight;


    // Constructors
    public Edge(int vertexA, int vertexB, int weight){
        this.vertexA = vertexA;
        this.vertexB = vertexB;
        this.weight = weight;
    }
    //get vertex A
    public int getVertexA(){
        return vertexA;
    }

    //get vertex B
    public int getVertexB(){
        return vertexB;
    }

    //get weight
    public int getWeight(){
        return weight;
    }

    public String toString(){
        return "(" + vertexA + "," + vertexB + ",w:" + weight+")";
    }


//***************************************************************
```

```
// implements compareTo()function of Comparable Interface
// weight ONLY

public int compareTo(Edge edge){
    int ans = 0;
    if (this.weight < edge.weight)
        ans = -1;
    else
        if (this.weight > edge.weight)
            ans = 1;
        else
            ans = 0;
    return ans;
}


//***************************************************************
```

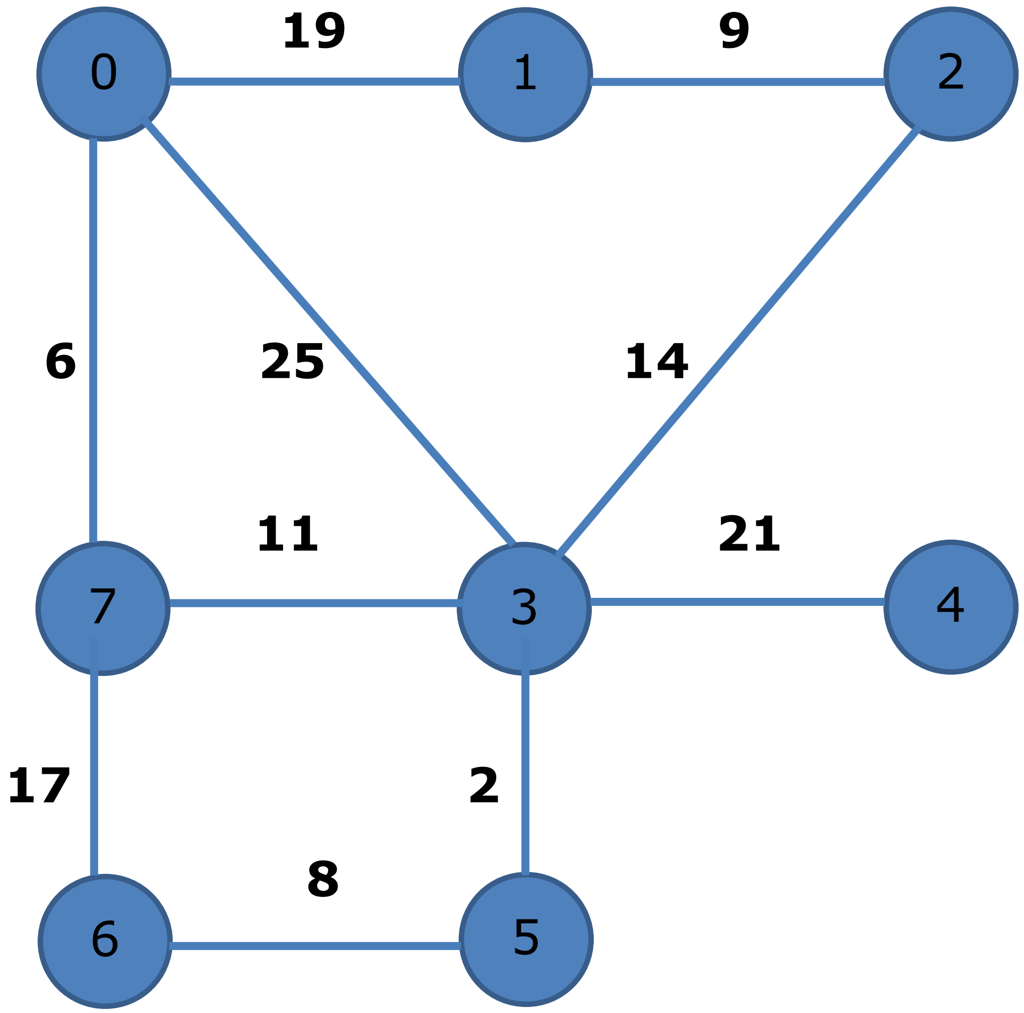
```java
// vertexA and vertexB ONLY
public boolean equals(Edge edge){
    boolean ans = (this.vertexA==edge.vertexA &&
                   this.vertexB==edge.vertexB) ||
                  (this.vertexA==edge.vertexB &&
                   this.vertexB==edge.vertexA);
    return ans;
}

public static void main(String[] args) {
    Edge edge1 = new Edge(1,2,10);
    Edge edge2 = new Edge(2,3,16);
    Edge edge3 = new Edge(3,4,9);
    Edge edge4 = new Edge(3,4,15);
    Edge edge5 = new Edge(4,5,10);

    System.out.println("Edge1 : " + edge1.toString());
    System.out.println("Edge2 : " + edge2.toString());
    System.out.println("Edge3 : " + edge3.toString());
    System.out.println("Edge4 : " + edge4.toString());
    System.out.println("Edge5 : " + edge5.toString());

    int ed1_5 = edge1.compareTo(edge5);
```

```
        int ed1_3 = edge1.compareTo(edge3);
        System.out.println("ed1_5 : " + ed1_5);
        System.out.println("ed1_3 : " + ed1_3);

        boolean eq1_5 = edge1.equals(edge5);
        boolean eq1_3 = edge3.equals(edge4);
        System.out.println("eq1_5 : " + eq1_5);
        System.out.println("eq1_3 : " + eq1_3);
    }

}
```

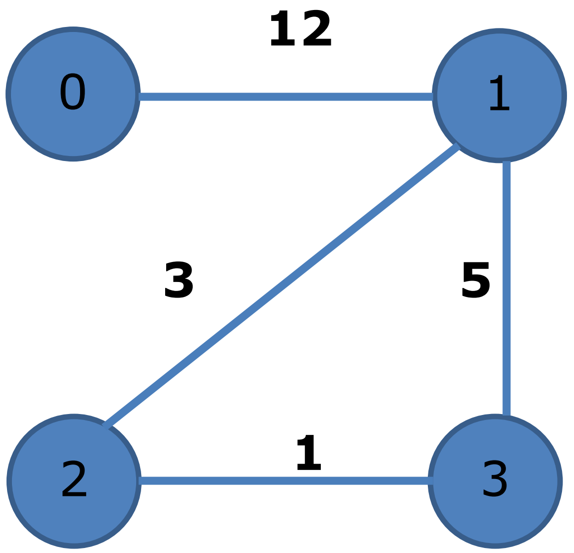
Result :

```
Edge1 : (1,2,w:10)
Edge2 : (2,3,w:16)
Edge3 : (3,4,w:9)
Edge4 : (3,4,w:15)
Edge5 : (4,5,w:10)
ed1_5 : 0
ed1_3 : 1
eq1_5 : false
eq1_3 : true
```

Example 1



Example 2



class InitGraphs

```
package kruskal;

import java.util.ArrayList;
import java.util.Arrays;

public class InitGraphs {

    public static ArrayList<Node>[] init1(){
        int n = 8;
        ArrayList<Node>[] graph = new ArrayList[n];
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<Node>();
        }
        graph[0].add(new Node(1, 19));
         graph[0].add(new Node(7,6));
         graph[0].add(new Node(3,25));

         graph[1].add(new Node(0,19));
         graph[1].add(new Node(2,9));
         graph[2].add(new Node(1,9));
         graph[2].add(new Node(3,14));

         graph[3].add(new Node(2,14));
         graph[3].add(new Node(4,21));
         graph[3].add(new Node(0,25));
         graph[3].add(new Node(5,2));
         graph[3].add(new Node(7,11));

         graph[4].add(new Node(3,21));

         graph[5].add(new Node(3,2));
         graph[5].add(new Node(6,8));

         graph[6].add(new Node(5,8));
         graph[6].add(new Node(7,17));

         graph[7].add(new Node(0,6));
         graph[7].add(new Node(6,17));
         graph[7].add(new Node(3,11));

         return graph;
    }
```

```
public static ArrayList<Node>[] init2(){
    int n = 4;
    ArrayList<Node>[] graph = new ArrayList[n];
    for (int i = 0; i < graph.length; i++) {
        graph[i] = new ArrayList<Node>();
    }
    graph[0].add(new Node(1,12));

     graph[1].add(new Node(0,12));
     graph[1].add(new Node(2,3));
     graph[1].add(new Node(3,5));

     graph[2].add(new Node(1,3));
     graph[2].add(new Node(3,1));

     graph[3].add(new Node(1,5));
     graph[3].add(new Node(2,1));
    return graph;
}
```

```
public static Edge[] getEdges(ArrayList<Node>[] graph){
    int numOfEdges = 0, n = graph.length;
    for (int i = 0; i < graph.length; i++) {
        numOfEdges = numOfEdges + graph[i].size();
    }
    numOfEdges = numOfEdges/2;
    Edge[] edges  = new Edge[numOfEdges];
    for (int i=0, k=0; k<numOfEdges && i<n; i++){
        for (int j = 0; j < graph[i].size(); j++) {
            Node nn = graph[i].get(j);
            Edge e = new Edge(i, nn.node, nn.weight);
            if (!contains(edges, e, k)) edges[k++] = e;
        }
    }
    return edges;
}
```

```
    private static boolean contains(Edge[] edges, Edge e, int k){
        boolean ans = false;
        for (int i = 0; !ans && i < k; i++) {
            if (edges[i].equals(e)) ans = true;
        }
        return ans;
    }

    public static void main(String[] args) {
        Edge edges[] = getEdges(init1());
        System.out.println(Arrays.toString(edges));
        edges = getEdges(init2());
        System.out.println(Arrays.toString(edges));
    }

}
```

Result :

```
[(0,1,w:19), (0,7,w:6), (0,3,w:25), (1,2,w:9), (2,3,w:14),
 (3,4,w:21), (3,5,w:2), (3,7,w:11), (5,6,w:8), (6,7,w:17)]

[(0,1,w:12), (1,2,w:3), (1,3,w:5), (2,3,w:1)]
```

class **DisjointSets**

Disjoint-set operations

A ***disjoint-set data structure*** maintains a collection S = {S_1, S_2, ... ,S_k} of disjoint dynamic sets. We identify each set by a ***representative***, which is **some member** of the set. In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered).
As in the other dynamic-set implementations we have studied, we represent each element of a set by an object. Letting x denote an object, we wish to support the following operations:

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of Sx [Sy, although many implementations of UNION specifically choose the
representative of either Sx or Sy as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets Sx and Sy,
removing them from the collection S. In practice, we often absorb the elements
of one of the sets into the other set.
FIND-SET.x/ returns a pointer to the representative of the (unique) set containing
x.

```
public class DisjointSets {

    private int[] parent;
    private int[] rank;                //rank[k] >= height of tree number k
```

Pseudocode

MAKE-SET(x)

```
public void makeSet(int k)
            k  →  parent[k]
            0  →  rank[k]
```

FIND-SET(x)

```
public int find(int v)

        parent[v] → int p
        if (v == p)
            return v
        return parent[v] = find(parent[p])
```

UNION(x, y)

```
public boolean union(int u, int v)

            boolean ans = false
            find(u) →   int rootU
            find(v) →   int rootV
            if (rootV == rootU)
                ans = false
            else
                ans = true
                if (rank[rootU] > rank[rootV])
                    rootU →          parent[rootV]
                else
                    if (rank[rootV] > rank[rootU])
                        rootV →          parent[rootU]
                    else {
                        rootU →          parent[rootV]
                        rank[rootU]++
            return ans
```

```
public String toString() {
    return "<DisjointSets\np " + Arrays.toString(parent) +
            "\nr " + Arrays.toString(rank) + "\n>";
}
```

```
public static void main(String[] args) {
        int numSets = 5;
        DisjointSets ds = new DisjointSets(numSets);
        for (int i = 0; i < numSets; i++) {
            ds.makeSet(i);
        }
        System.out.println(ds);

        ds.union(1,2);
        System.out.println("union 1 2");
                System.out.println(ds);

        ds.union(3,4);
        System.out.println("union 3 4");
        System.out.println(ds);

        ds.union(1,3);
        System.out.println("union 1 3");
        System.out.println(ds);

        ds.union(1,4);
        System.out.println("union 1 4");
        System.out.println(ds);
```

```
    }
}
```

Example : **{0} {1} {2} {3} {4}**

```
makeSet method begin
parent :     [0, 0, 0, 0, 0]
rank   :     [0, 0, 0, 0, 0]
makeSet method end


makeSet method begin
parent :     [0, 1, 0, 0, 0]
rank   :     [0, 0, 0, 0, 0]
makeSet method end


makeSet method begin
parent :     [0, 1, 2, 0, 0]
rank   :     [0, 0, 0, 0, 0]
makeSet method end
makeSet method begin
parent :      [0, 1, 2, 3, 0]
rank   :      [0, 0, 0, 0, 0]
makeSet method end


makeSet method begin
parent :      [0, 1, 2, 3, 4]
rank   :      [0, 0, 0, 0, 0]
makeSet method end


<<<           DisjointSets
                                   p    [0, 1, 2, 3, 4]
                                   r    [0, 0, 0, 0, 0]
>>>

------------------------------------------------------
```

```
union 1 2

union   method begin

        find    method begin
        v = 1
        p = 1
        v == p :    p = 1
        find    method end

        find    method begin
        v = 2
        p = 2
        v == p :    p = 2
        find    method end

        rootU = 1     rootV = 2

        rank[rootU] == rank[rootV] ::  rank[rootU] = 1   rank[rootV] =0
        rank[rootU] = 1

union   method end
```

{0} {1 , 1} {3} {4}

```
<<<DisjointSets
                    p   [0, 1, 1, 3, 4]
                    r   [0, 1, 0, 0, 0]
>>>
```

```
union 3 4

union    method begin

        find    method begin
        v = 3
        p = 3
        v == p :    p = 3
        find    method end

        find    method begin
        v = 4
        p = 4
        v == p :    p = 4
        find    method end

rootU = 3     rootV = 4
rank[rootU] == rank[rootV] ::  rank[rootU] = 1   rank[rootV] =0
rank[rootU] = 1

union    method end
```

{0}   {1 , 1}   {3 , 3}

```
<<<          DisjointSets
                          p   [0, 1, 1, 3, 3]
                          r   [0, 1, 0, 1, 0]
>>>
```

union 1 3

union   method begin

```
        find    method begin
        v = 1
        p = 1
        v == p : 1
        find    method end

        find    method begin
        v = 3
        p = 3
        v == p : 3
        find    method end
```

rootU = 1   rootV = 3
rank[rootU] == rank[rootV] :: rank[rootU] = 2   rank[rootV] =1
rank[rootU] = 2
union   method end

{0} {1 , 1 , 1} {3}

```
<<<DisjointSets
                p   [0, 1, 1, 1, 3]
                r   [0, 2, 0, 1, 0]
>>>
```

union 1 4

```
union   method begin

        find    method begin
        v = 1
        p = 1
        v == p : 1
        find    method end

        find    method begin
        v = 4
        p = 3
        find recurs   method end

        find    method begin
        v = 1
        p = 1
        v == p : 1
        find    method end
rootU = 1     rootV = 1
rootV == rootU :   ans = false
union   method end
```

{0}   {1 , 1 , 1 , 1}

```
<<<DisjointSets
                    p   [0, 1, 1, 1, 1]
                    r   [0, 2, 0, 1, 0]
>>>
```

Kruskal algorithm

```
public class Kruskal {

    Edge[]          graph;
    Edge[]          tree;
    DisjointSets    vertexGroup;
    int             numOfEdges,
                    numOfVertices,
                    numEdgesInMST;
```

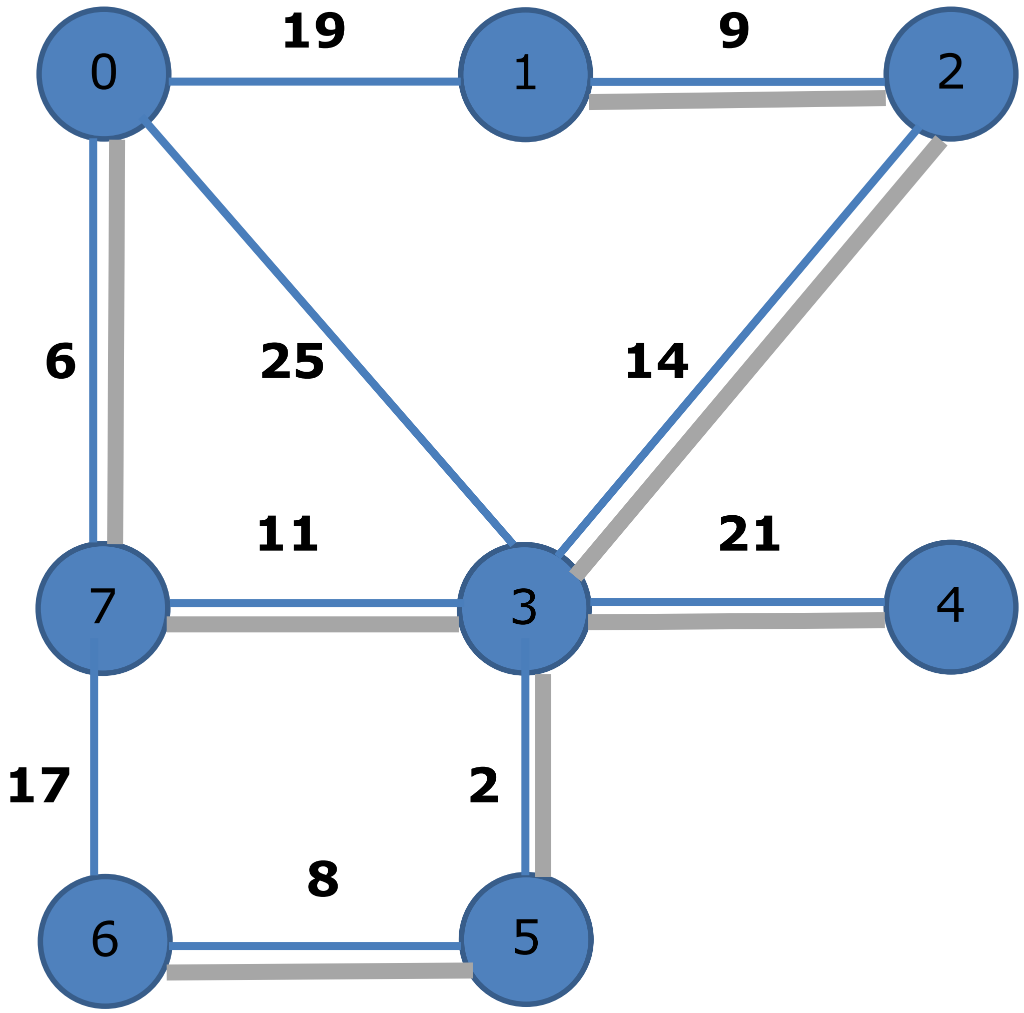
```
public Kruskal(Edge[] graph, int n)

    graph.length    → numOfEdges
    0               → numEdgesInMST
    graph           → this.graph
    Arrays.sort(this.graph)
    n               → numOfVertices
    tree = new Edge [numOfVertices]
    vertexGroup = new DisjointSets(numOfVertices)

    loop (i  from  0   to numOfVertices -1   step 1)
        vertexGroup.makeSet(i)
```

```
public void CreateMSP()

    int  i=0;
    loop while (    numEdgesInMST < numOfVertices &&
                    i < numOfEdges)
                 if(vertexGroup.union(graph[i].getVertexA(),
                                      graph[i].getVertexB()))
                      tree[numEdgesInMST++] = graph[i];
                 i++
public double calcSummWieight()

    double w = 0
    loop  from i=0  to  numEdgesInMST - 1      step 1
        w = w + tree[i].getWeight()

    return w
```

```
public void printTree(){
    for (int i=0; i<numEdgesInMST; i++){
        System.out.println(tree[i].toString());
    }
 }
```

```
public static void main(String[] args) {
    Kruskal kruskal =
      new  Kruskal(InitGraphsDS.getEdges(InitGraphsDS.init1()), 10);
    kruskal.CreateMSP();
    kruskal.printTree();
    System.out.println("sum weight =    "
                            + kruskal.calcSummWieight());

    kruskal =
      new Kruskal(InitGraphsDS.getEdges(InitGraphsDS.init2()), 4);
    kruskal.CreateMSP();
    kruskal.printTree();
    System.out.println("sum weight = "
                            + kruskal.calcSummWieight());
   }

}
```

Example 1



Example 2