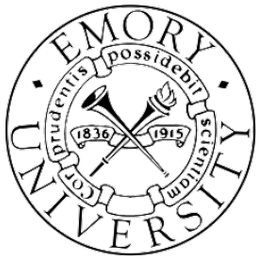
Minimum Spanning Tree

Data Structures and Algorithms

Emory University

Jinho D. Choi

EMORY
UNIVERSITY



Types of Graphs

Types of Graphs

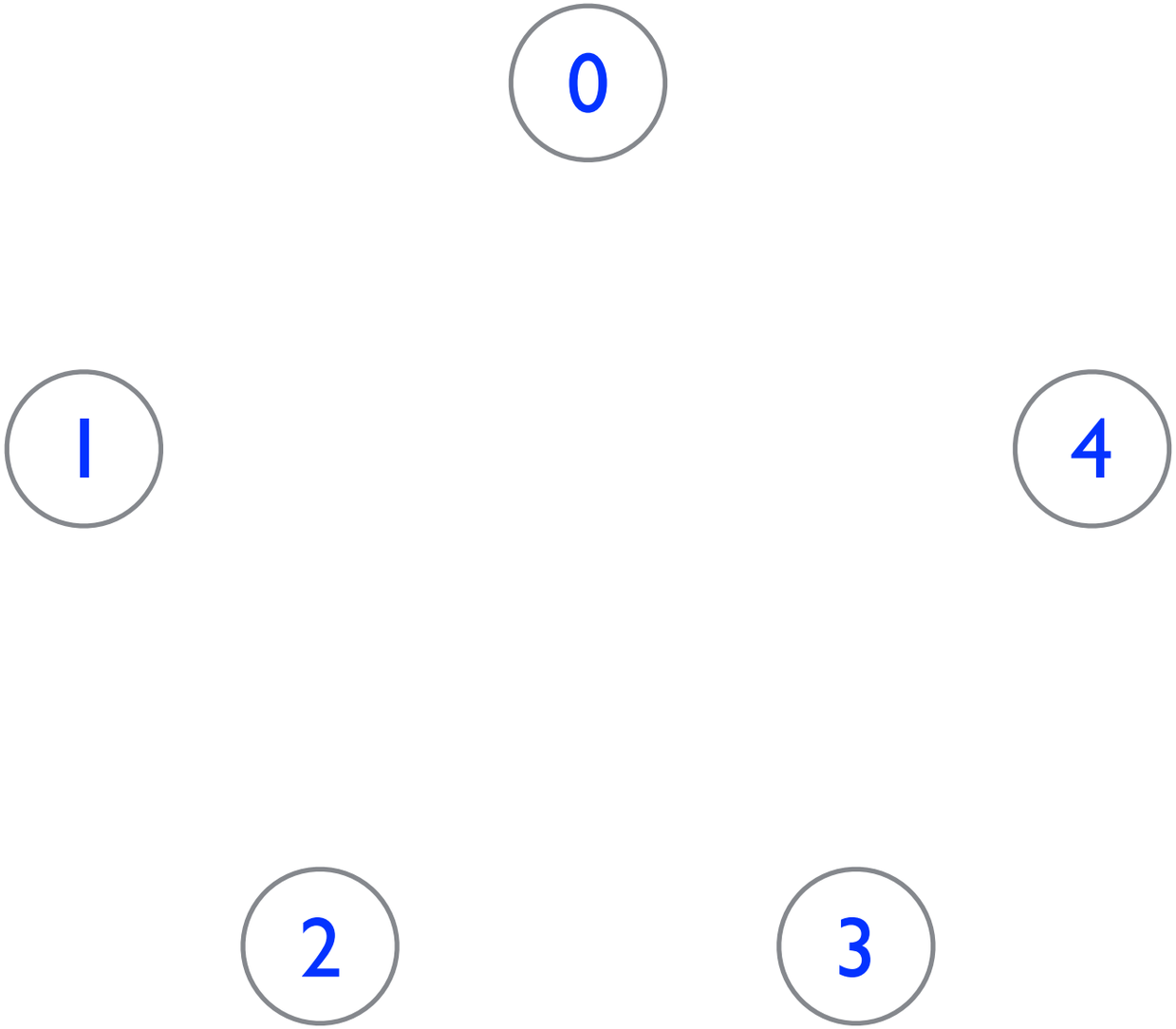
G = (,)

Types of Graphs

$G = (V, \quad)$

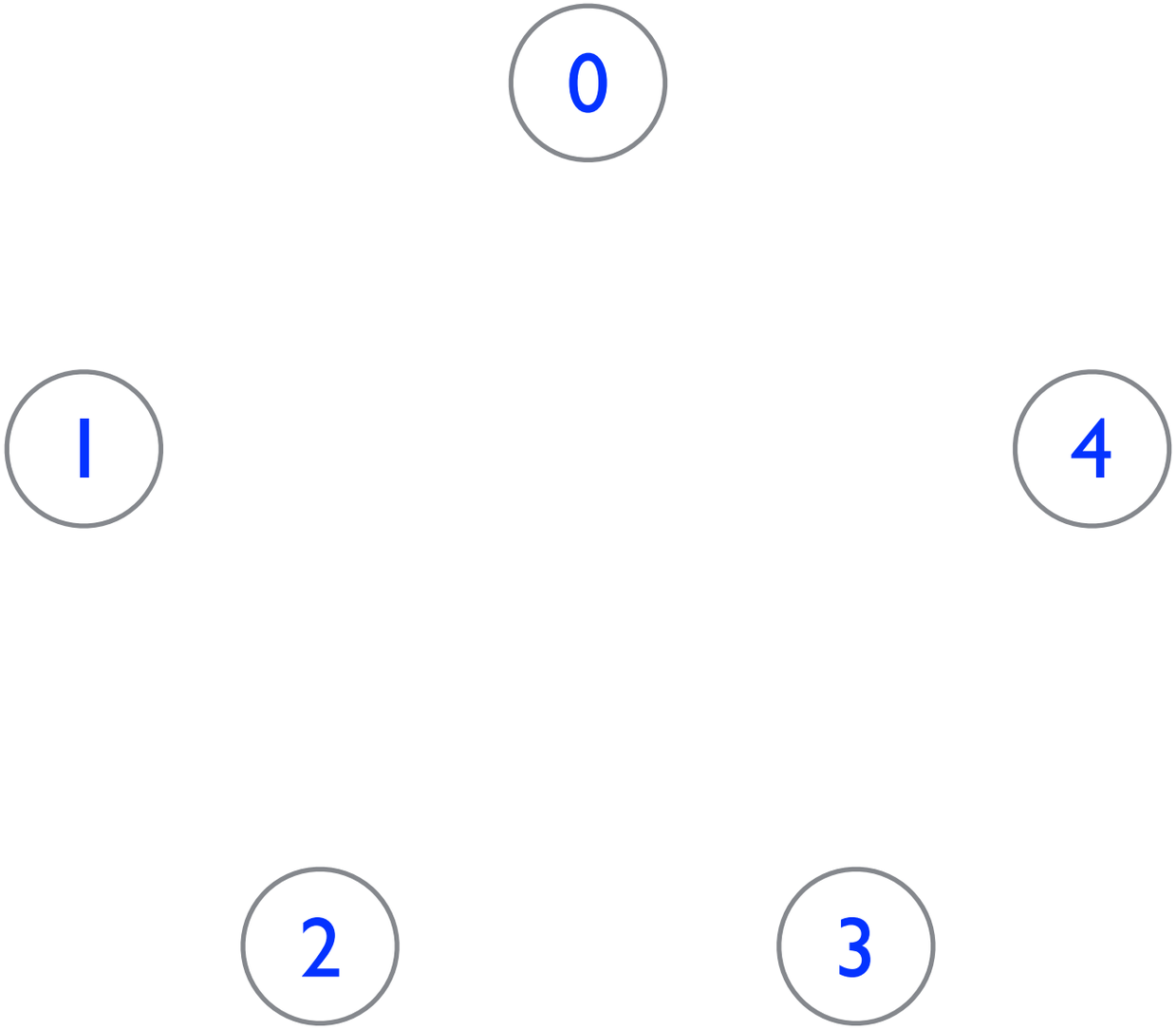
Types of Graphs

G = (V , E)



Types of Graphs

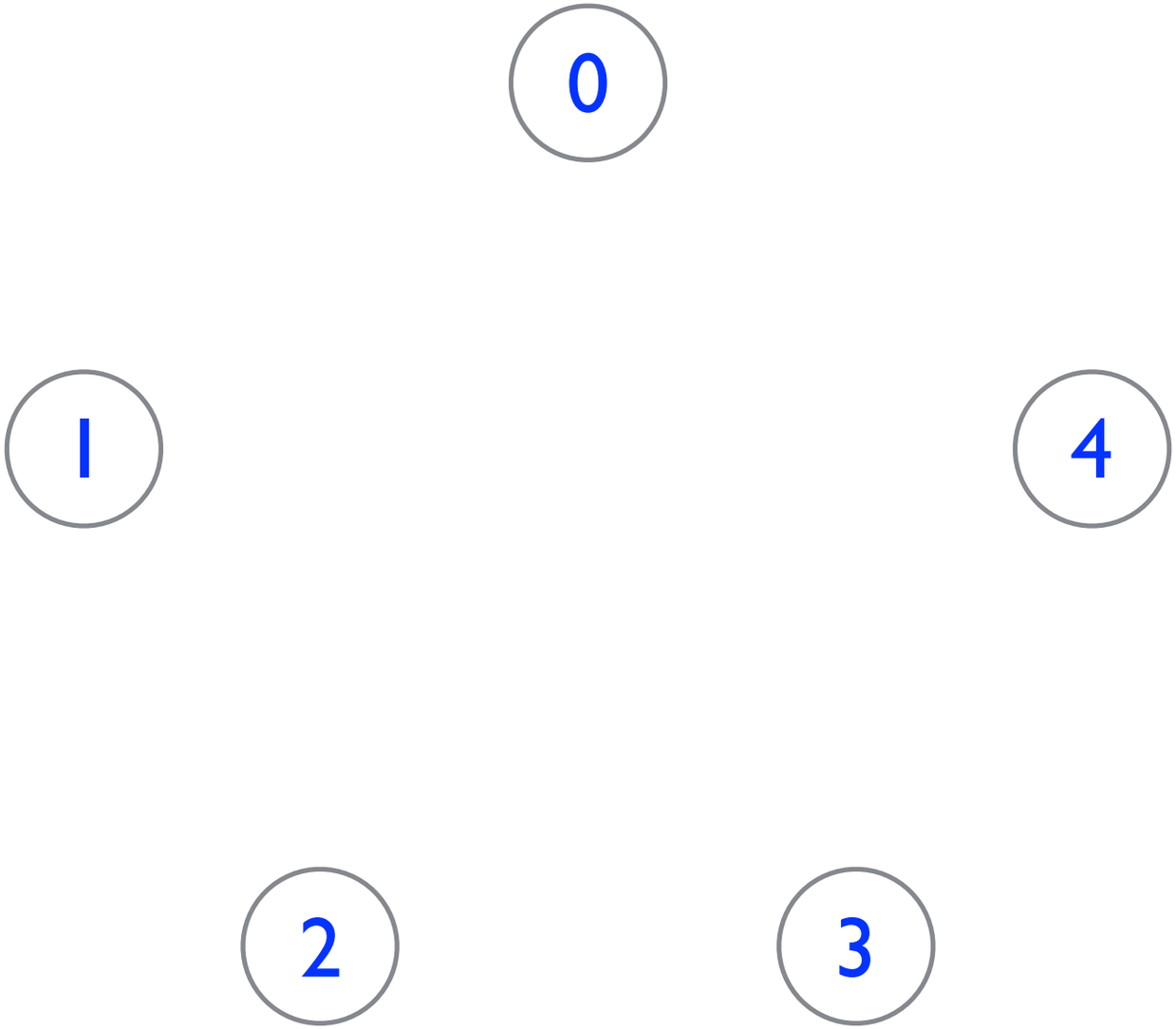
$G = (V, E)$



Types of Graphs

G = (V , E)

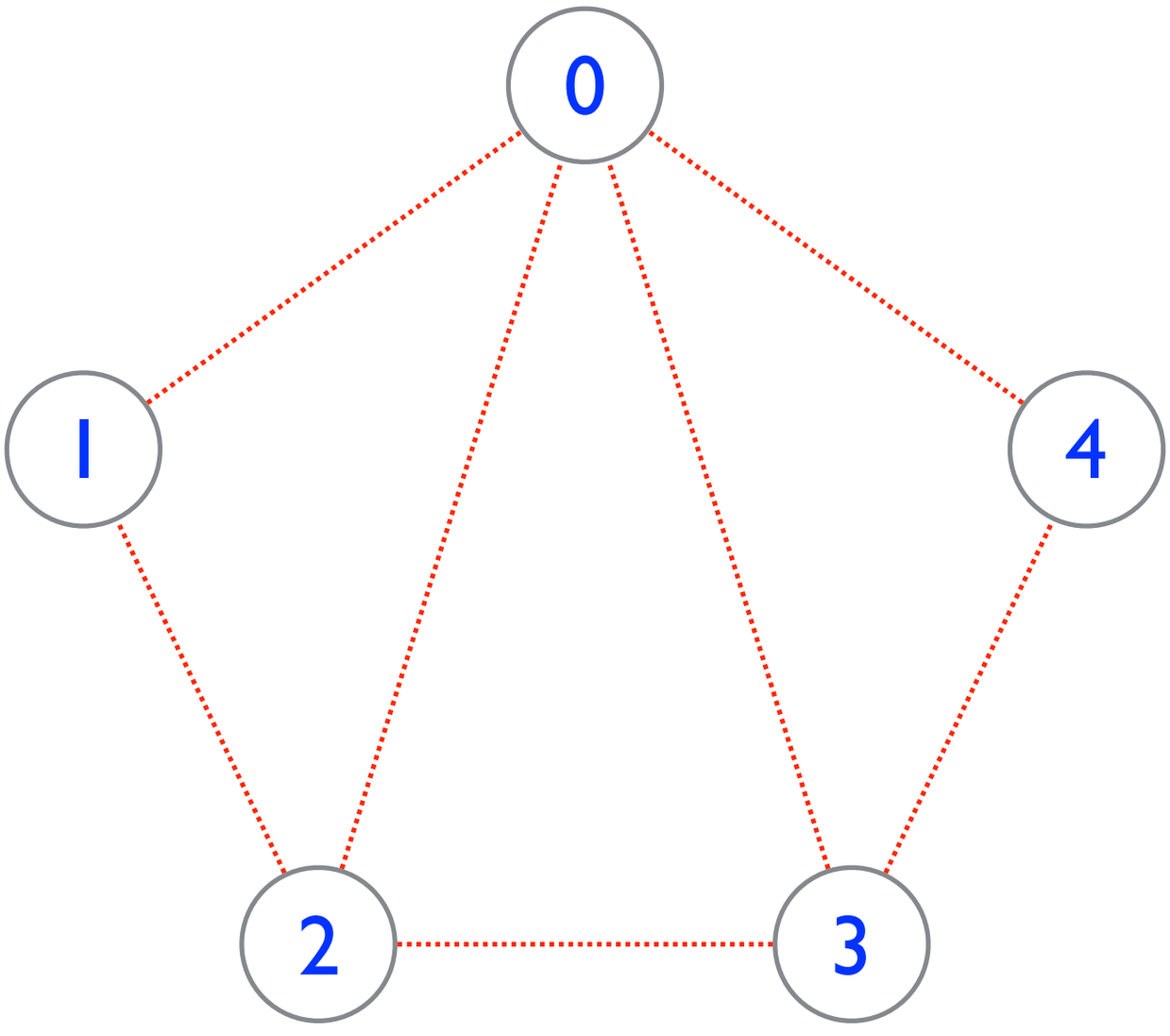
Undirected



Types of Graphs

G = (V , E)

Undirected

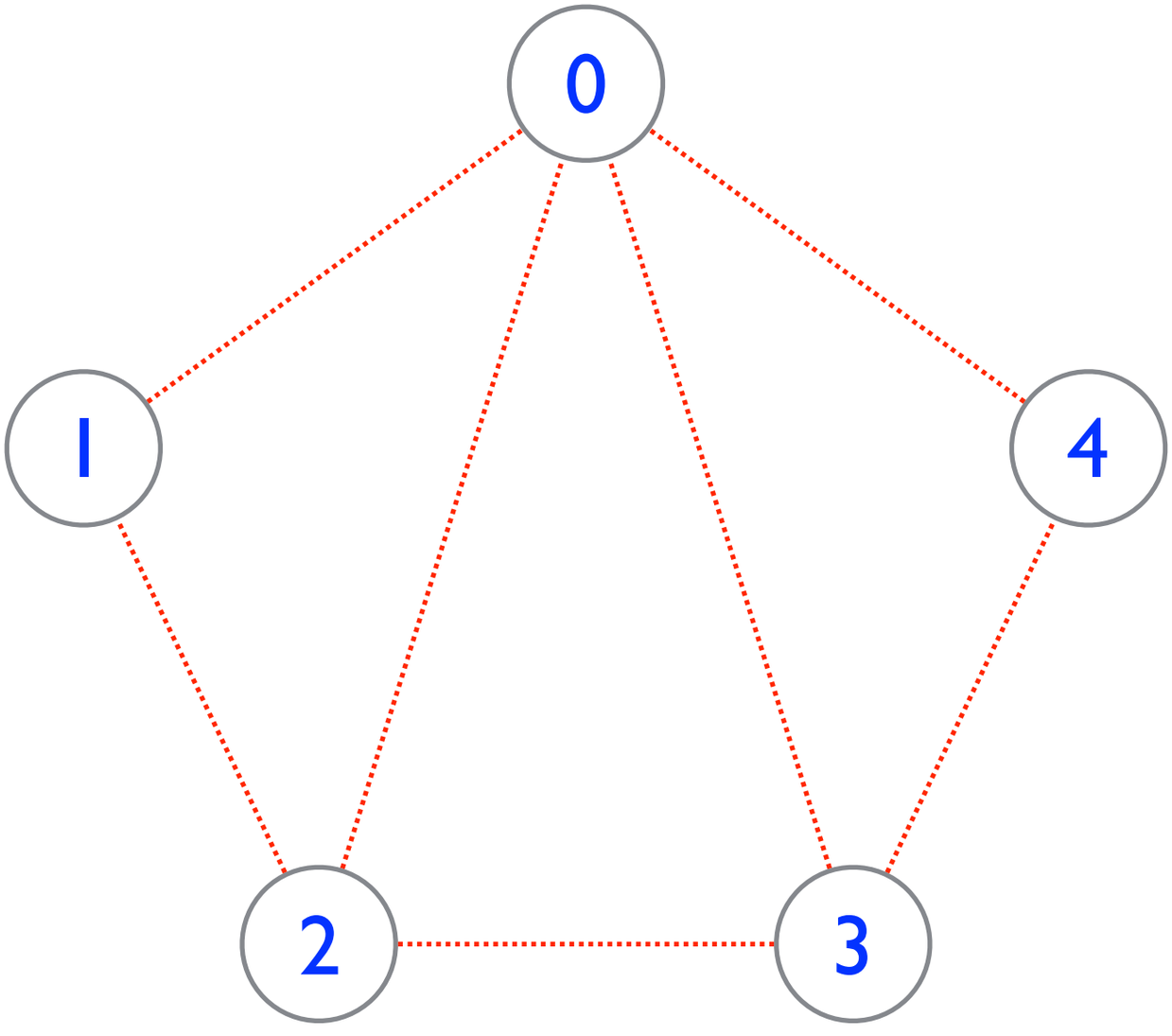


Types of Graphs

G = (V , E)

Undirected

Weighted



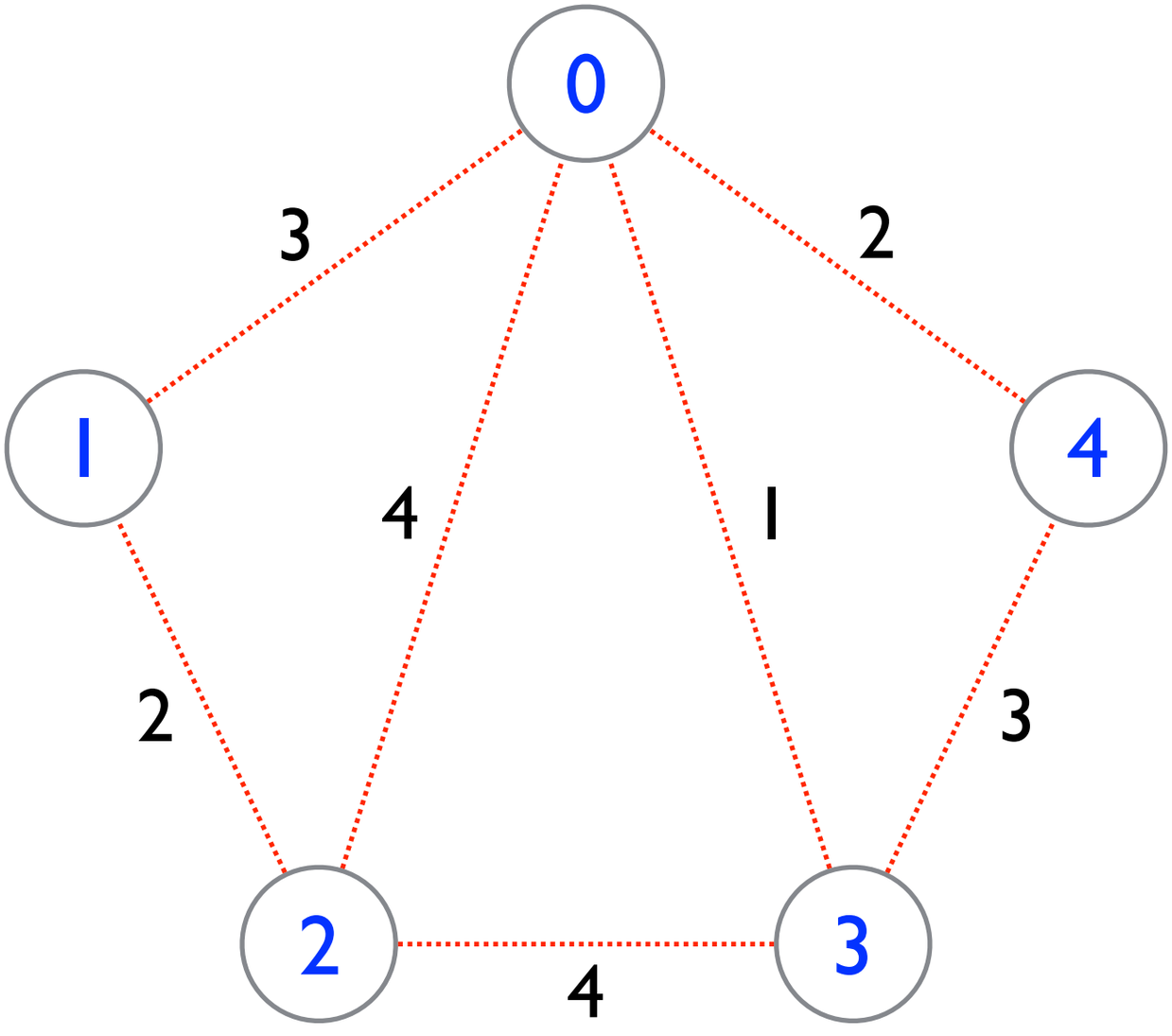
EMORY UNIVERSITY

Types of Graphs

G = (V , E)

Undirected

Weighted



EMORY UNIVERSITY

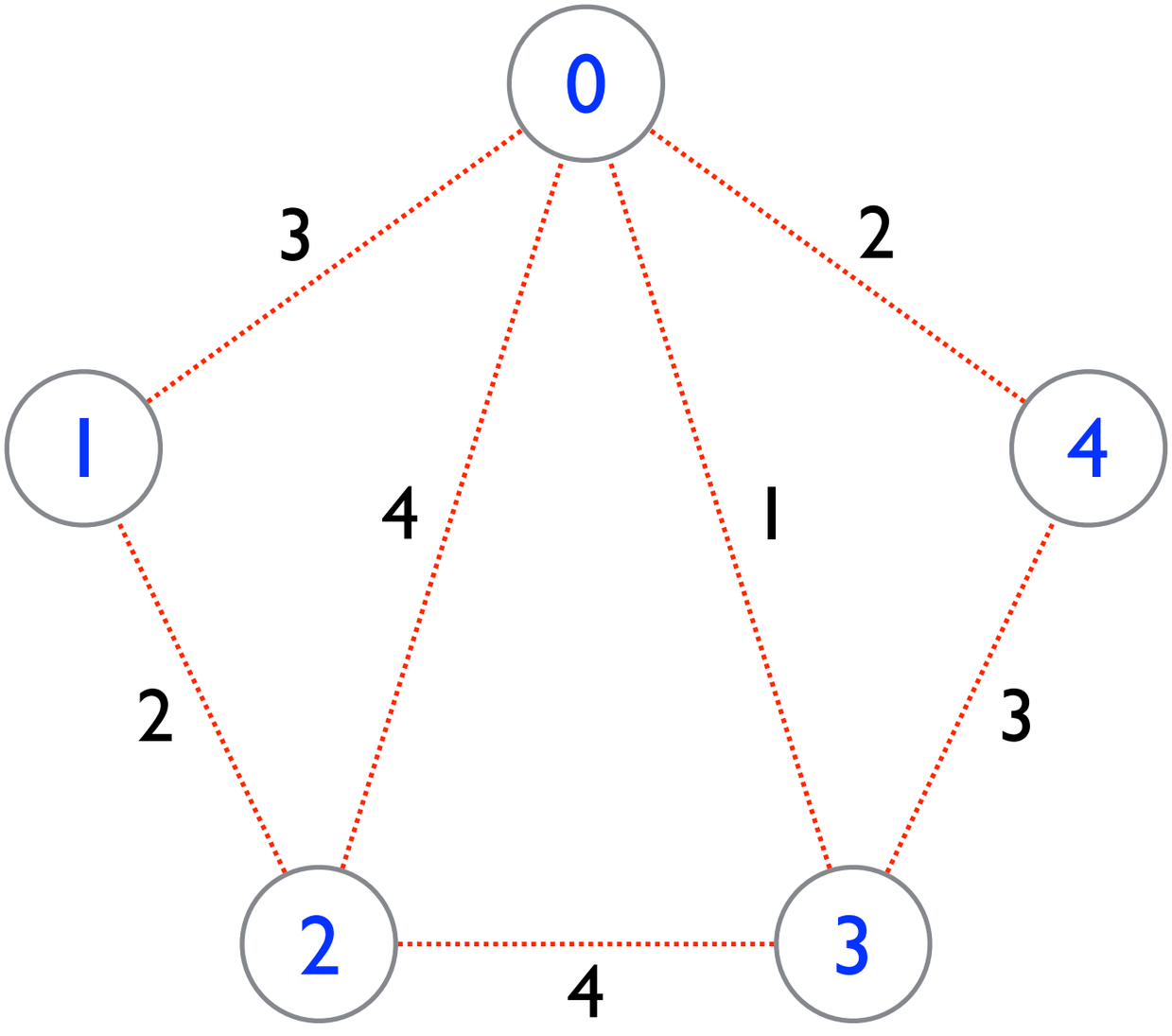
Types of Graphs

G = (V , E)

Undirected

Weighted

Directed



EMORY UNIVERSITY

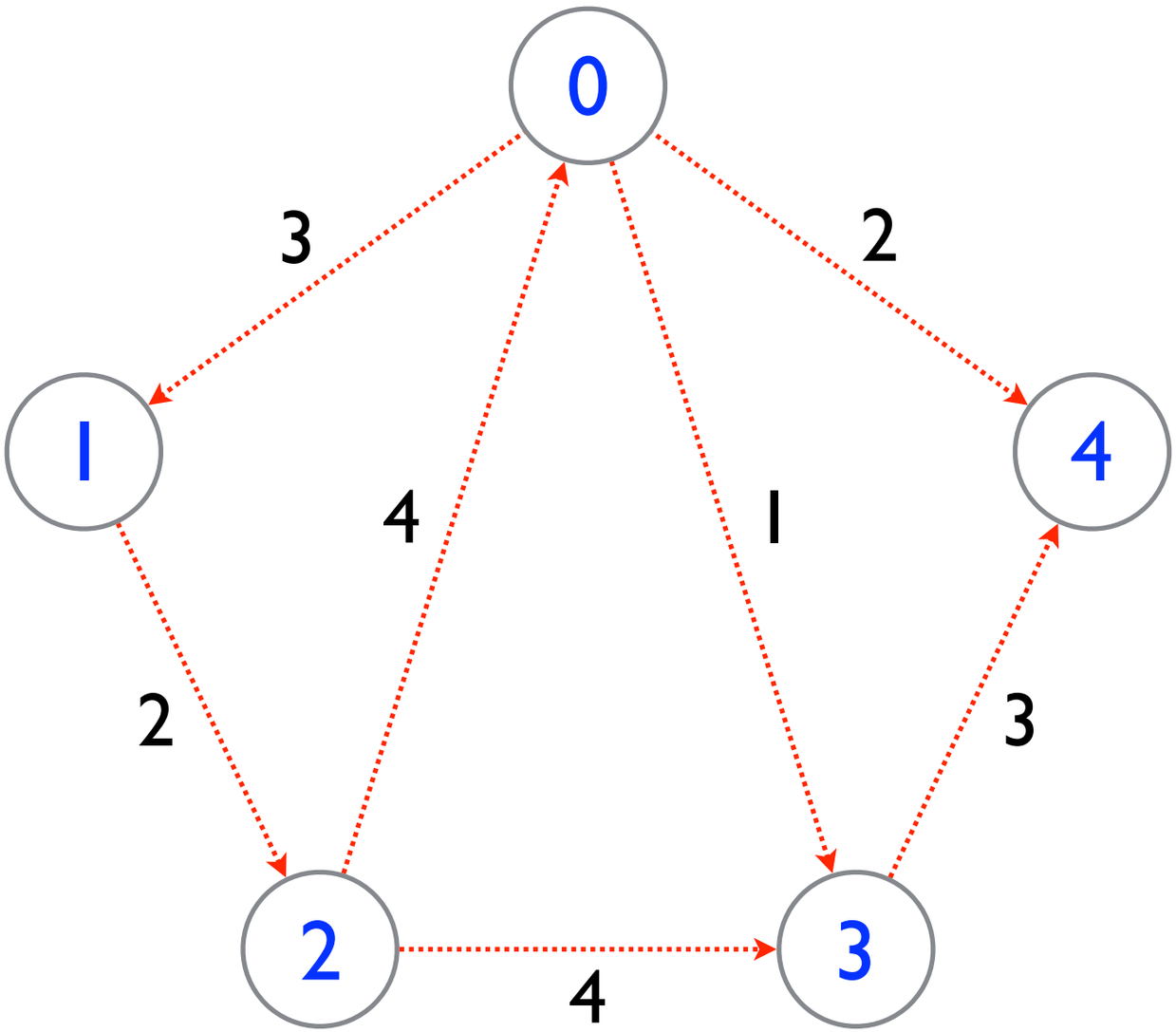
Types of Graphs

G = (V , E)

Undirected

Weighted

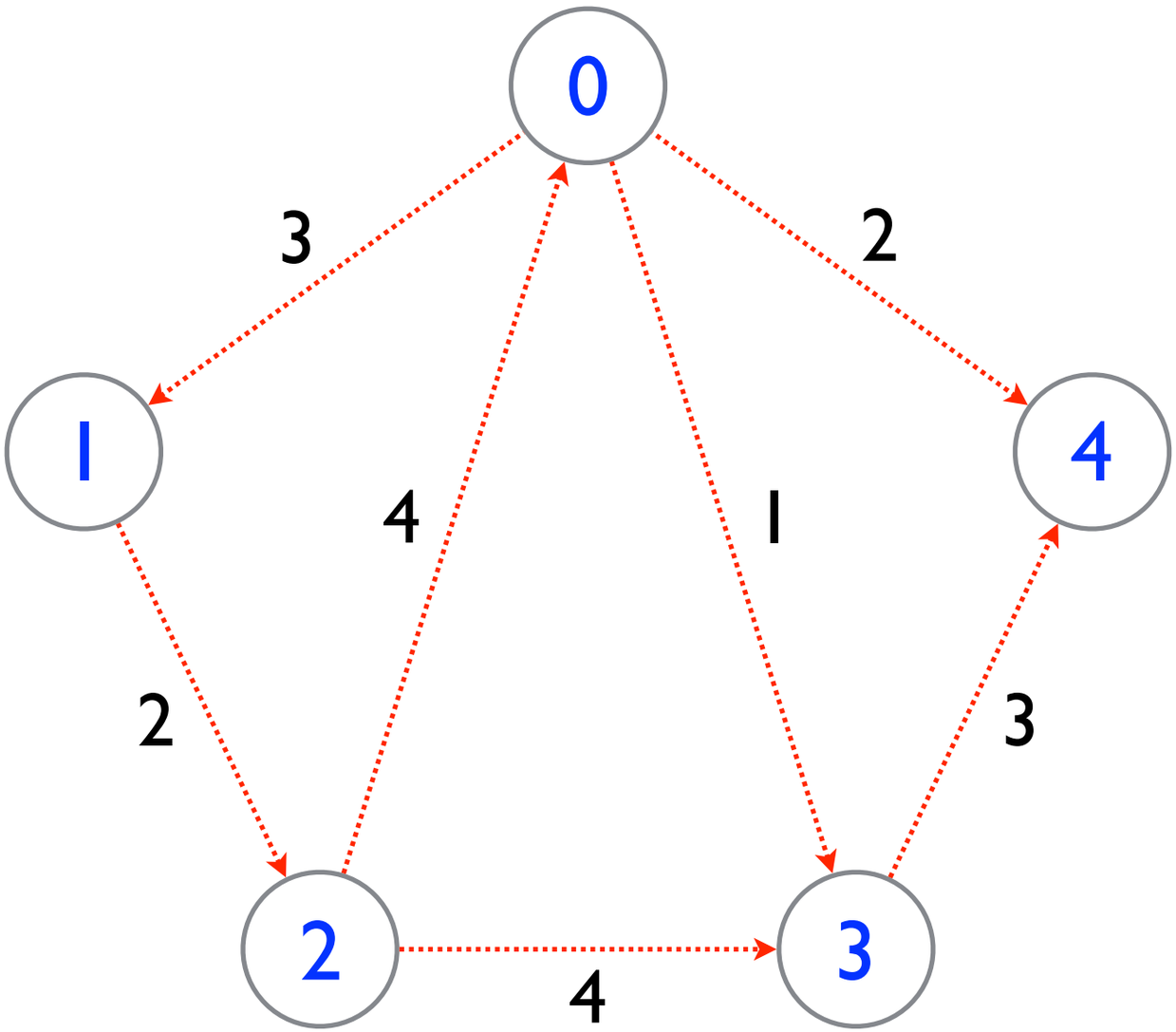
Directed



EMORY UNIVERSITY

Types of Graphs

G = (V , E)



Undirected

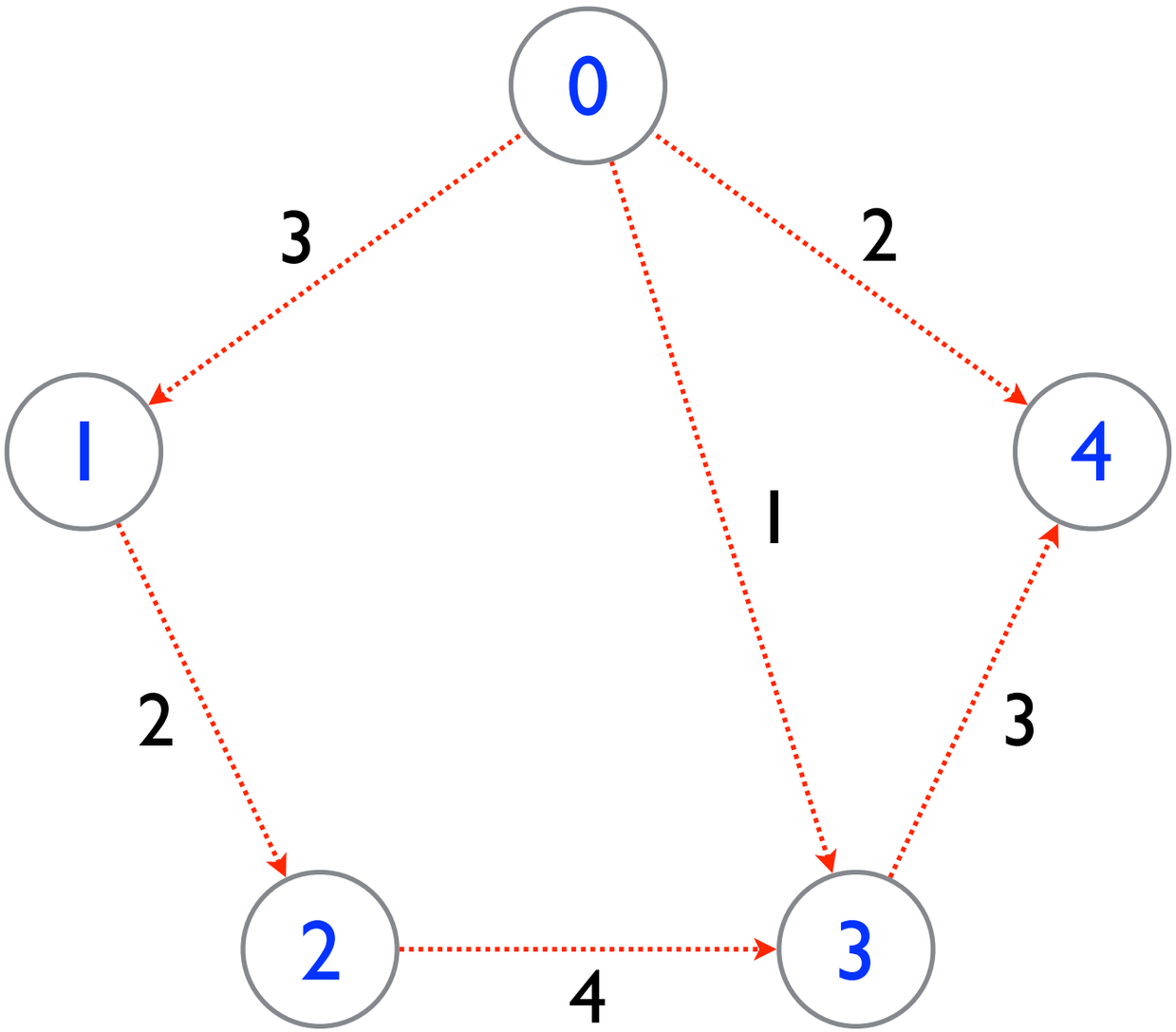
Weighted

Directed

Acyclic

Types of Graphs

G = (V , E)



Undirected

Weighted

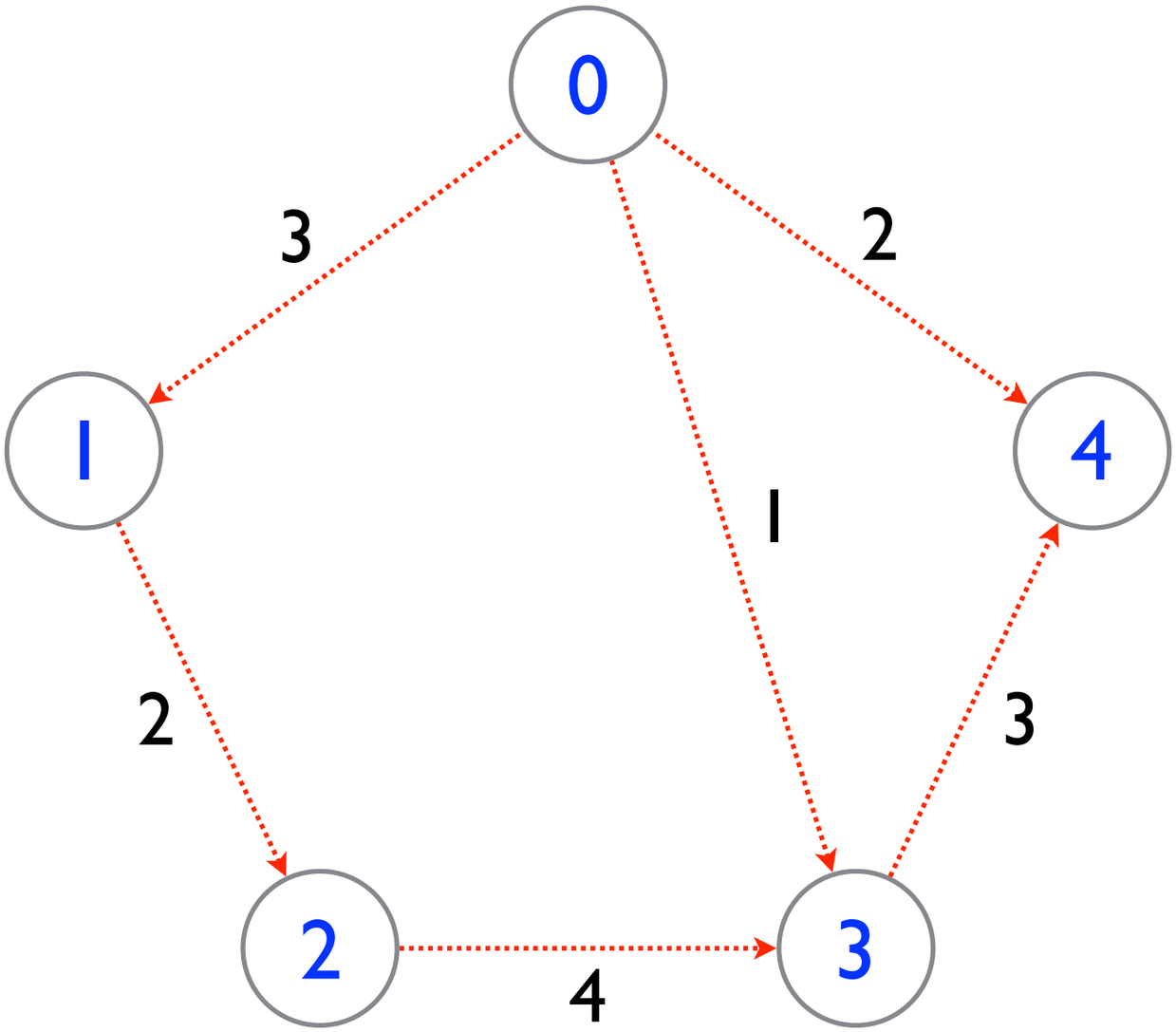
Directed

Acyclic

EMORY UNIVERSITY

Types of Graphs

G = (V , E)



Undirected

Weighted

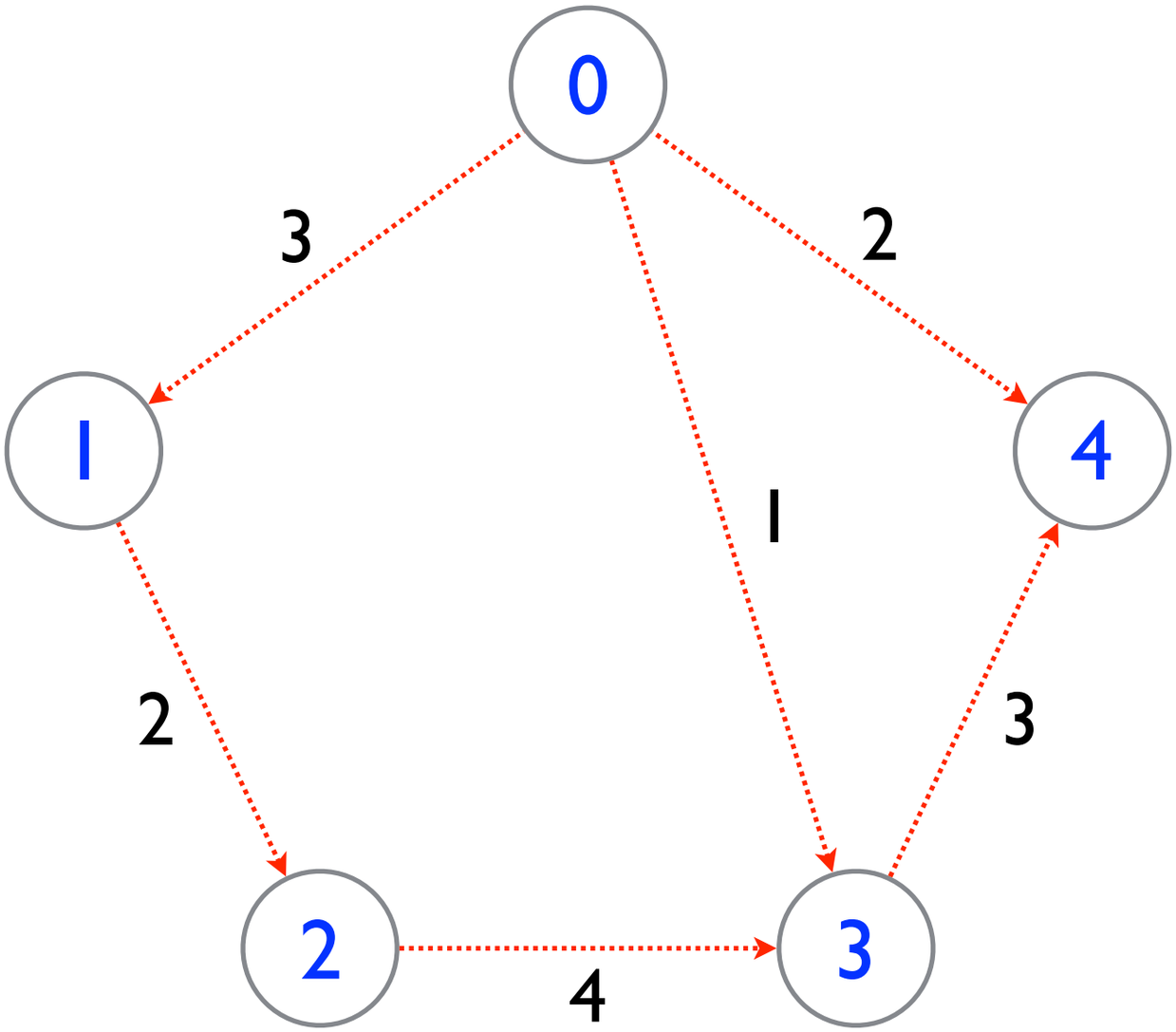
Directed

Acyclic

Tree

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

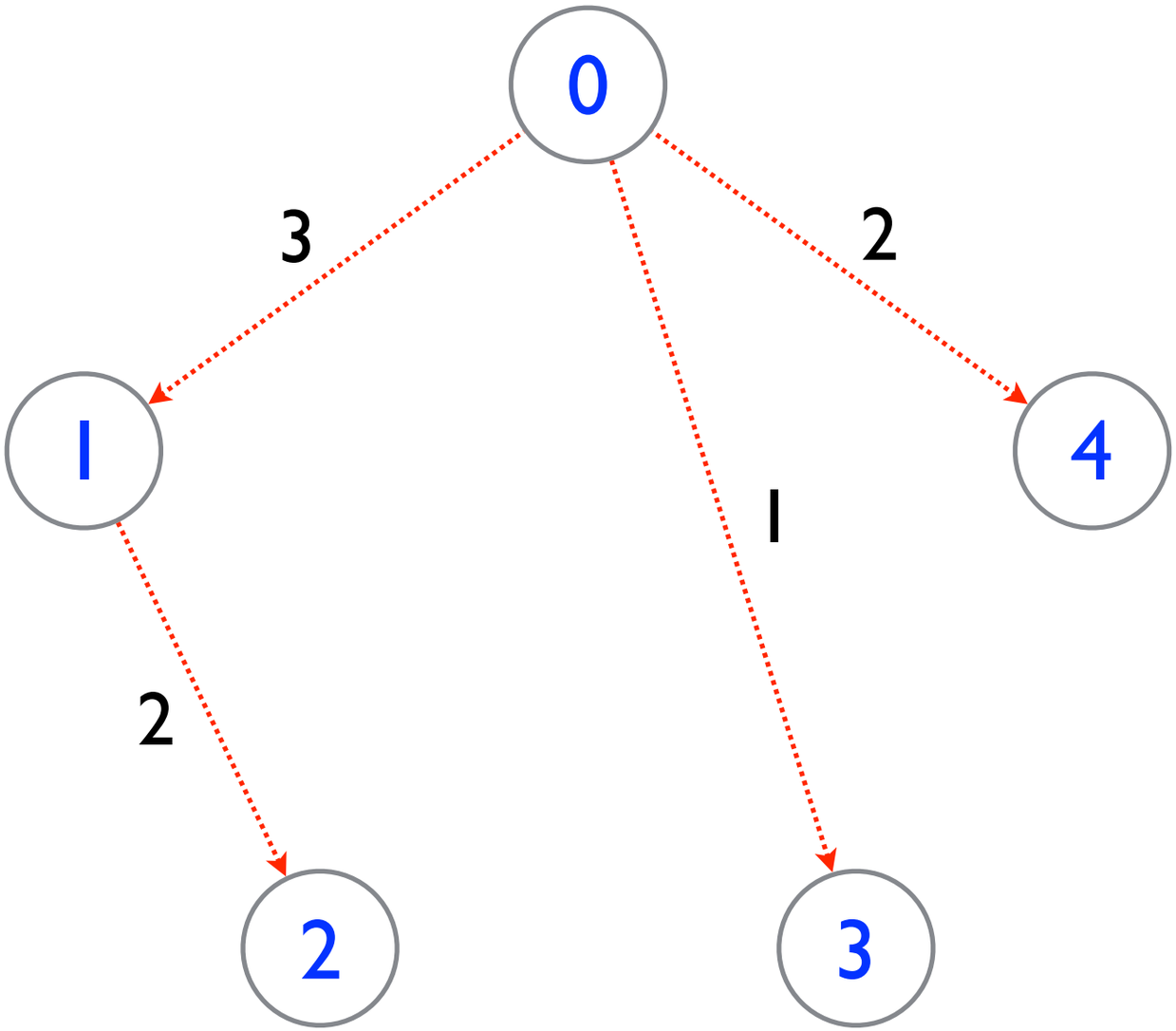
Acyclic

Tree

Every node except for the root must have exactly one incoming edge.

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

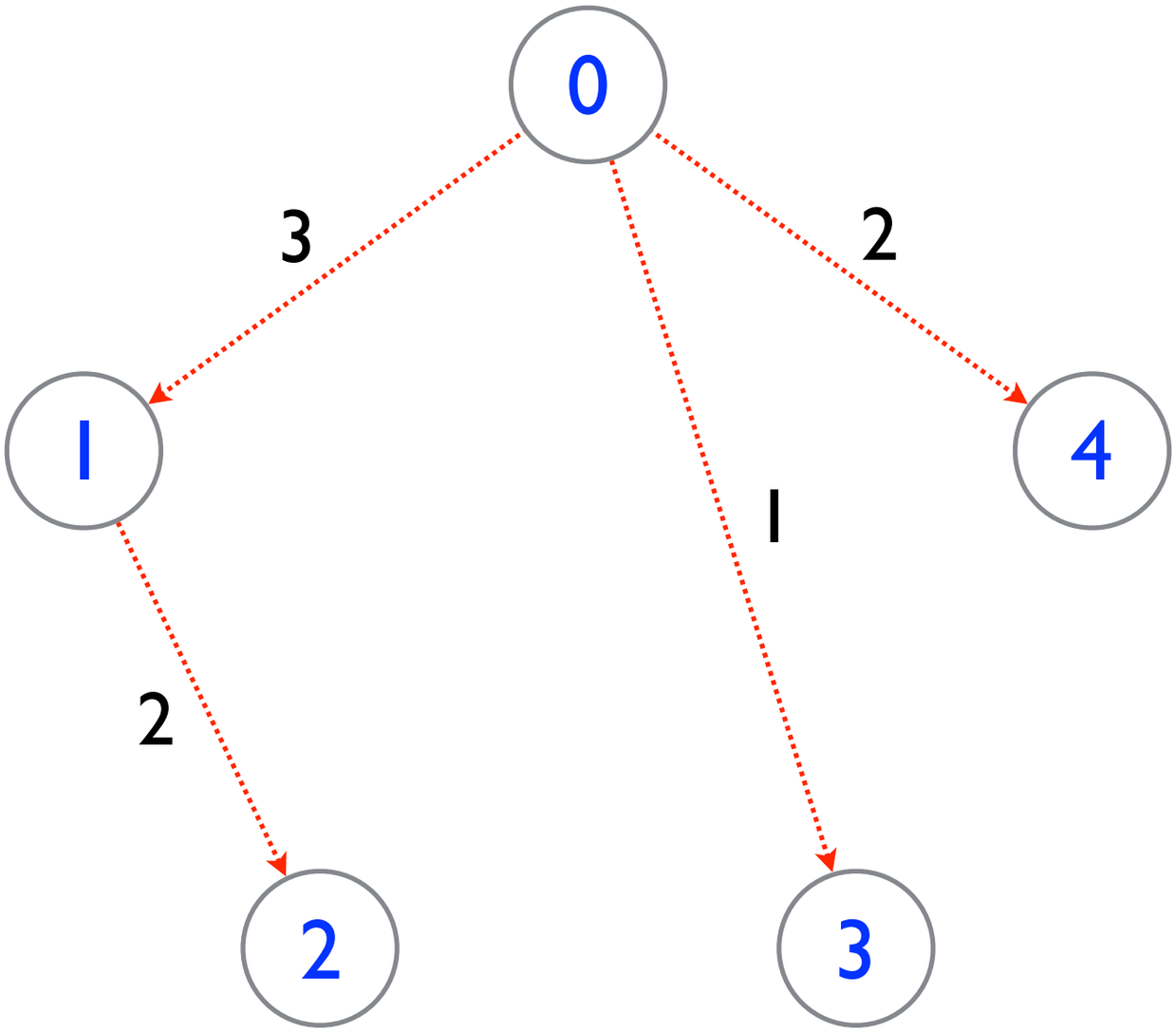
Acyclic

Tree

Every node except for the root must have exactly one incoming edge.

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

Acyclic

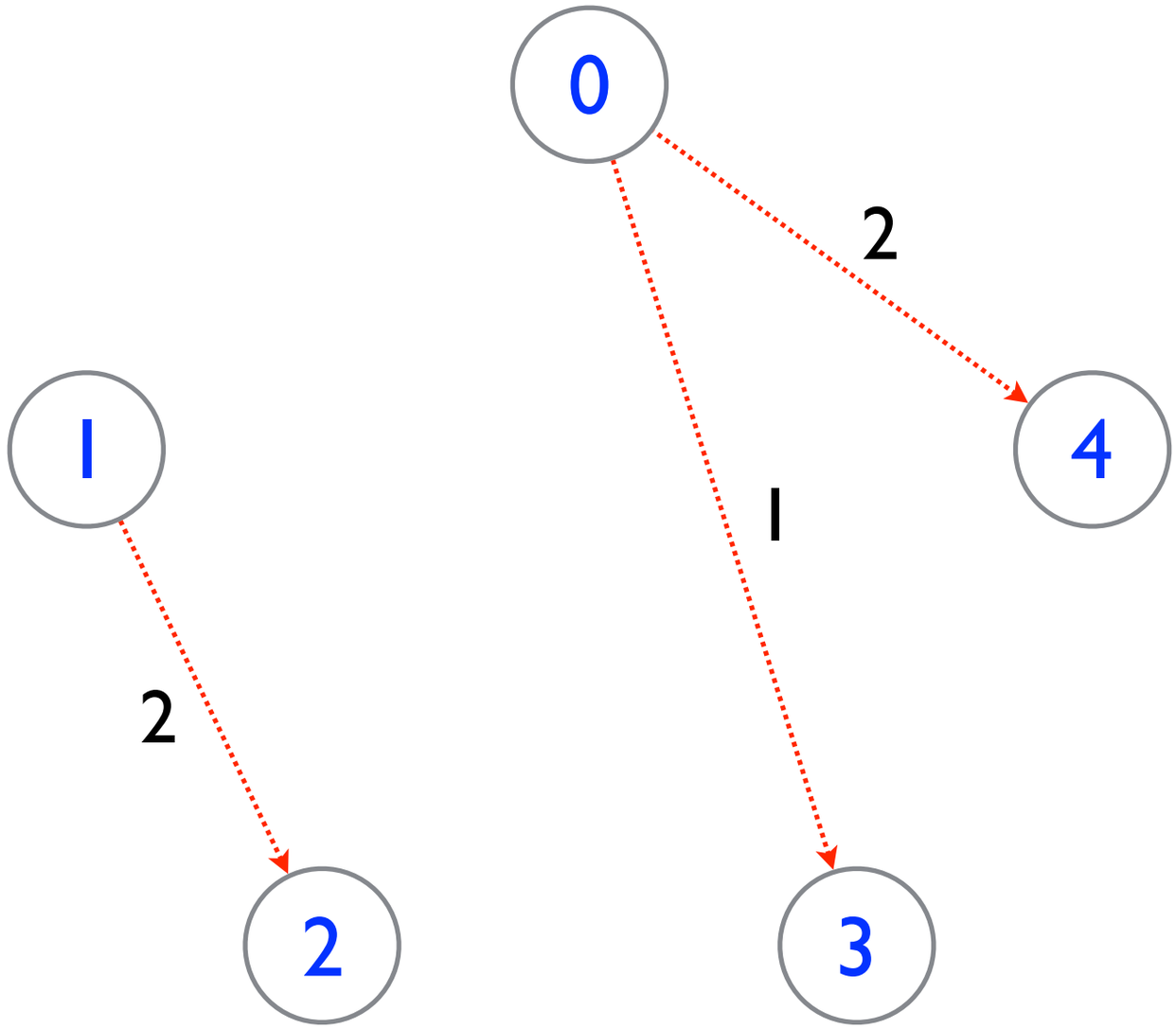
Tree

Every node except for the root must have exactly one incoming edge.

Every node must be reachable from the root.

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

Acyclic

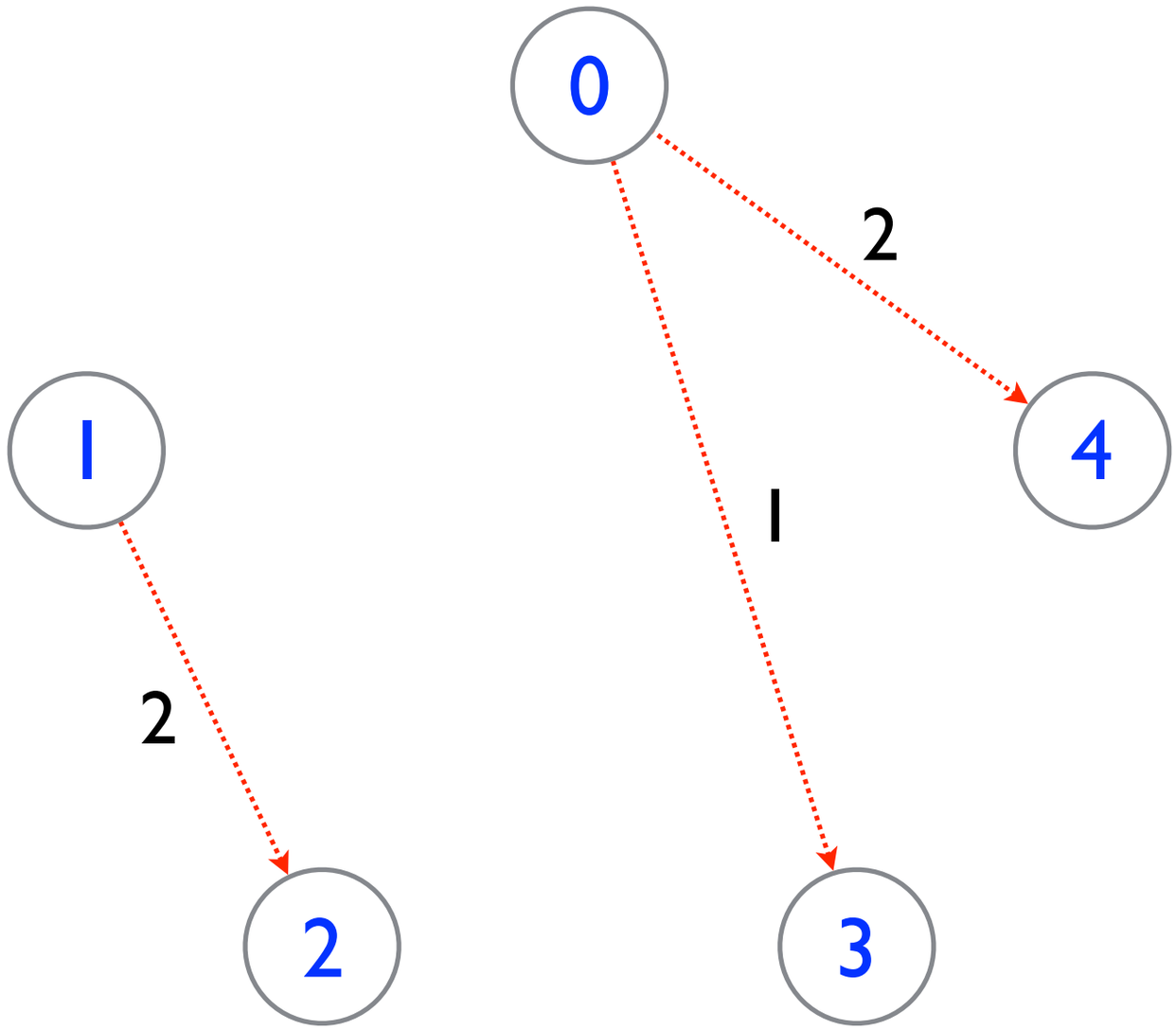
Tree

Every node except for the root must have exactly one incoming edge.

Every node must be reachable from the root.

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

Acyclic

Tree

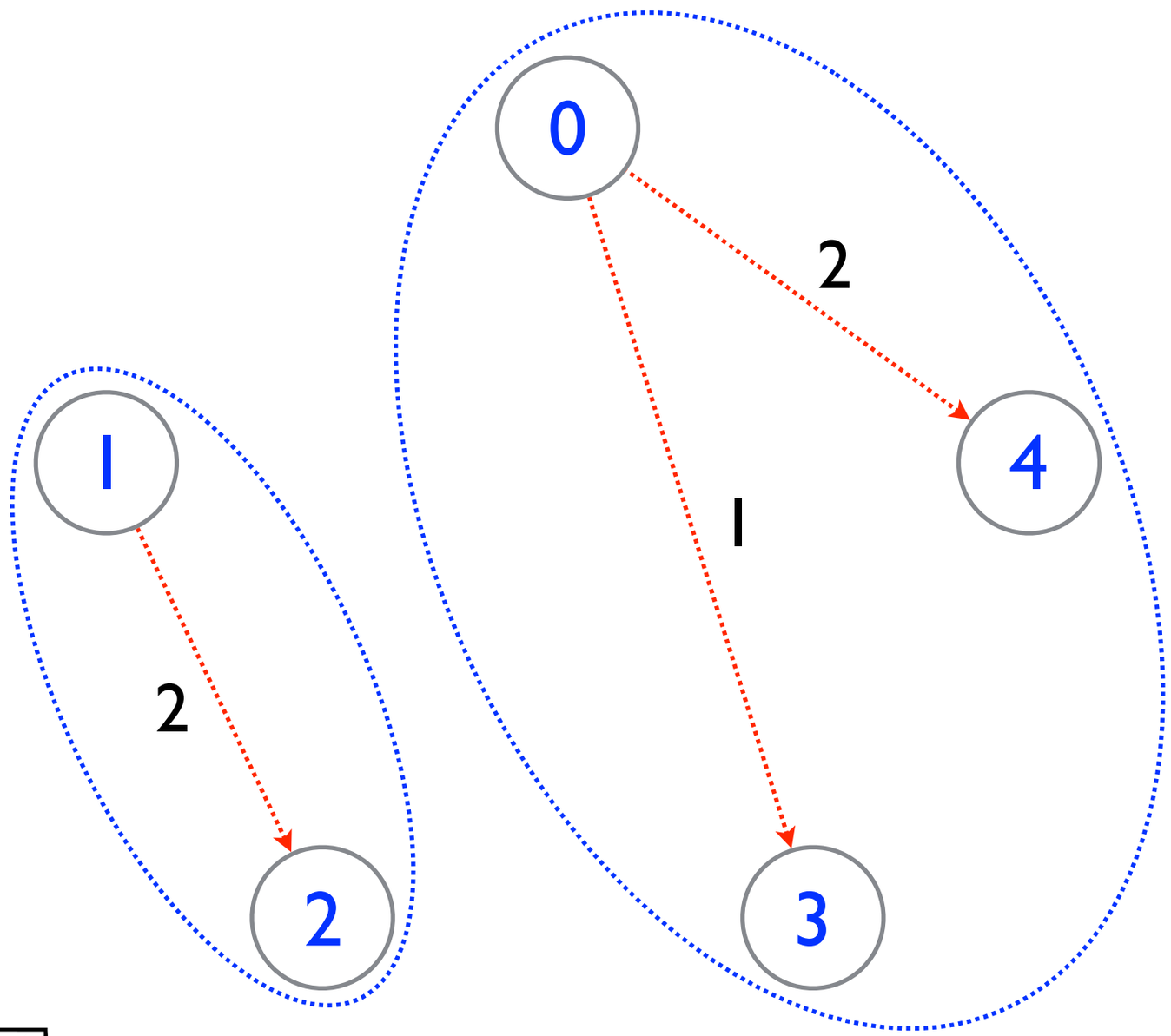
Forest

Every node except for the root must have exactly one incoming edge.

Every node must be reachable from the root.

Types of Graphs

G = (V , E)



Undirected

Weighted

Directed

Acyclic

Tree

Forest

Every node except for the root must have exactly one incoming edge.

Every node must be reachable from the root.

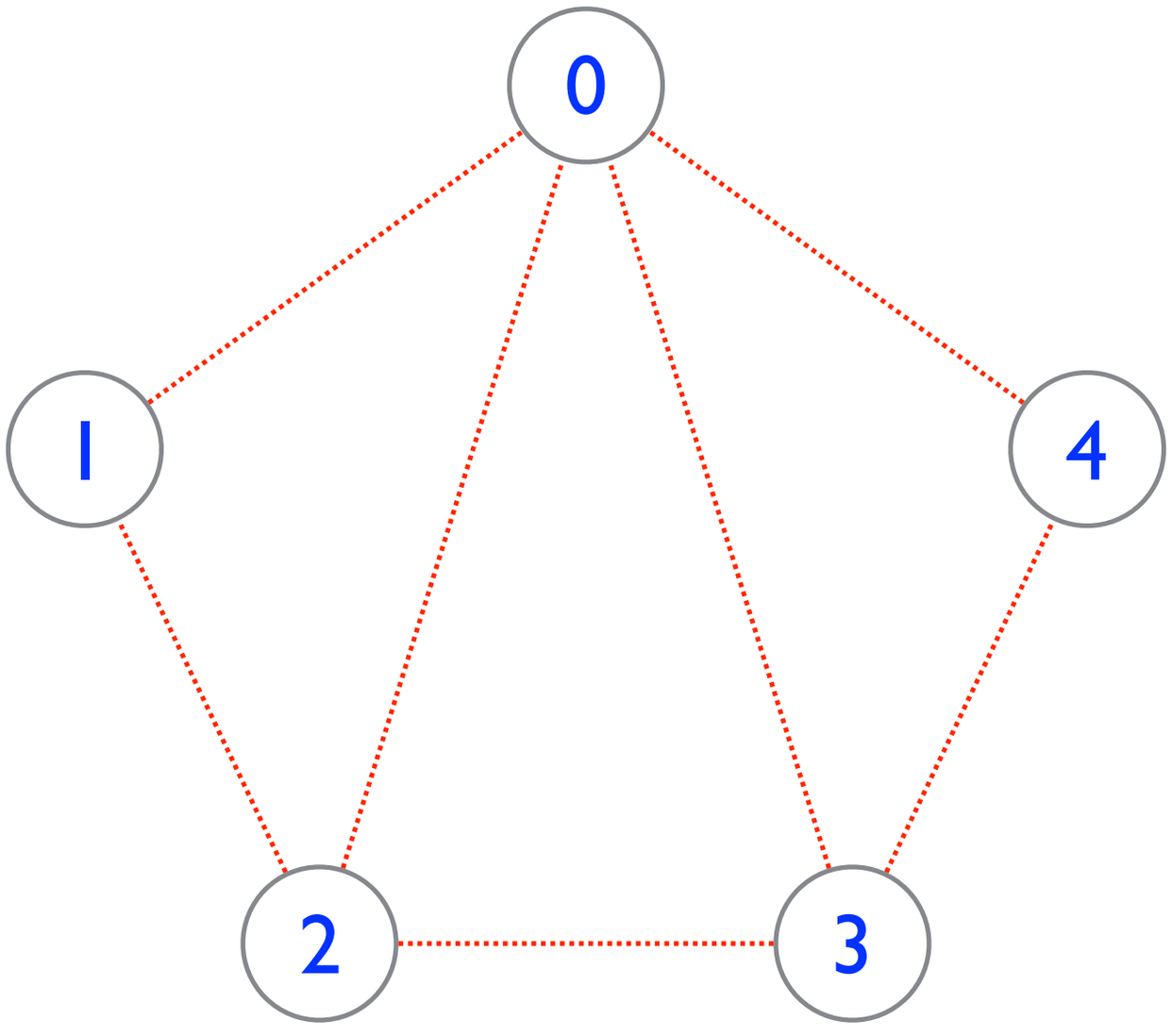
Undirected vs. Directed Graph

Undirected vs. Directed Graph

Can we represent undirected graphs
using directed graphs?

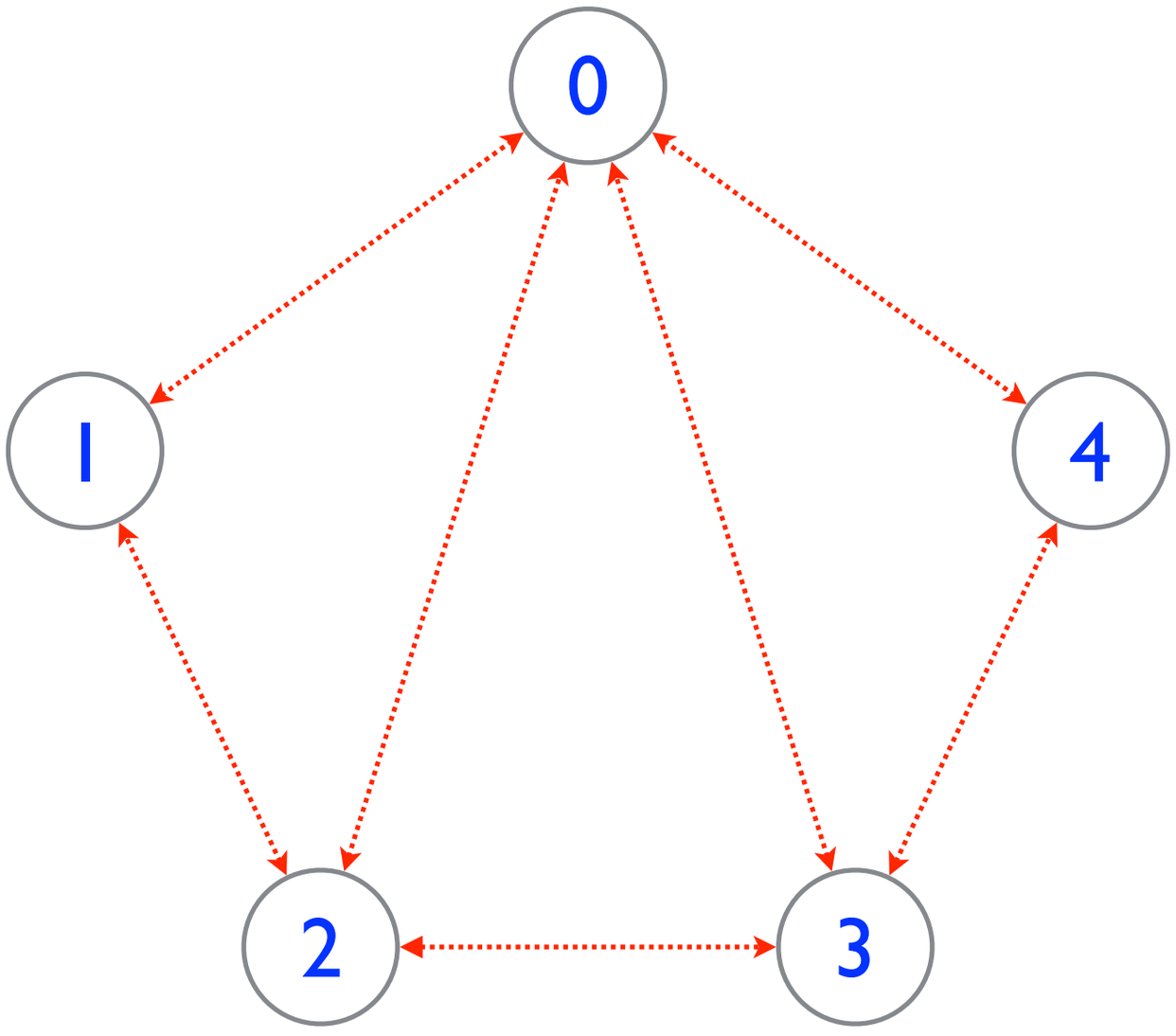
Undirected vs. Directed Graph

Can we represent undirected graphs
using directed graphs?



Undirected vs. Directed Graph

Can we represent undirected graphs
using directed graphs?



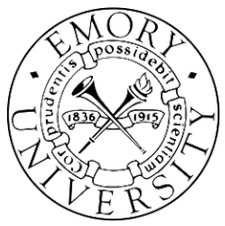
Edge

Edge

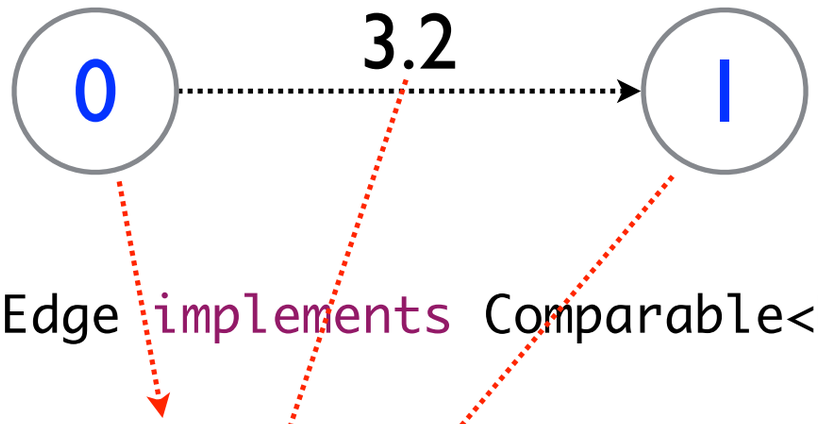
Edge


```
public class Edge implements Comparable<Edge>
{
  private int    source;
  private int    target;
  private double weight;

  public Edge(int source, int target, double weight)
  {
    setSource(source);
    setTarget(target);
    setWeight(weight);
  }
}
```

EMORY
UNIVERSITY

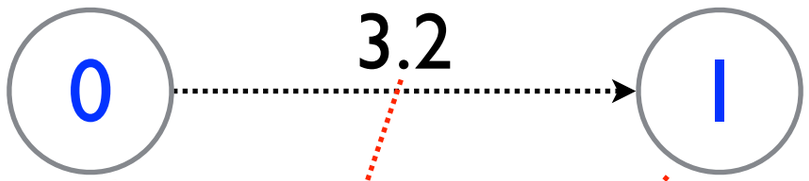


Edge




```
public class Edge implements Comparable<Edge>
{
  private int    source;
  private int    target;
  private double weight;

  public Edge(int source, int target, double weight)
  {
    setSource(source);
    setTarget(target);
    setWeight(weight);
  }
}
```

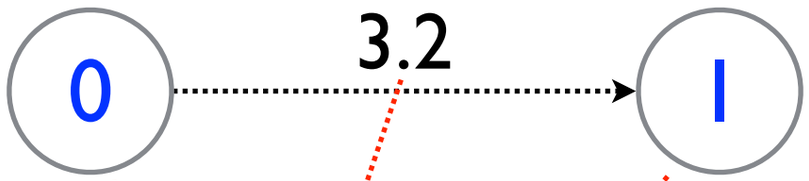
Edge




```
public class Edge implements Comparable<Edge>
{
  private int    source;
  private int    target;
  private double weight;

  public Edge(int source, int target, double weight)
  {
    setSource(source);
    setTarget(target);
    setWeight(weight);
  }
}
```

by weight

Edge



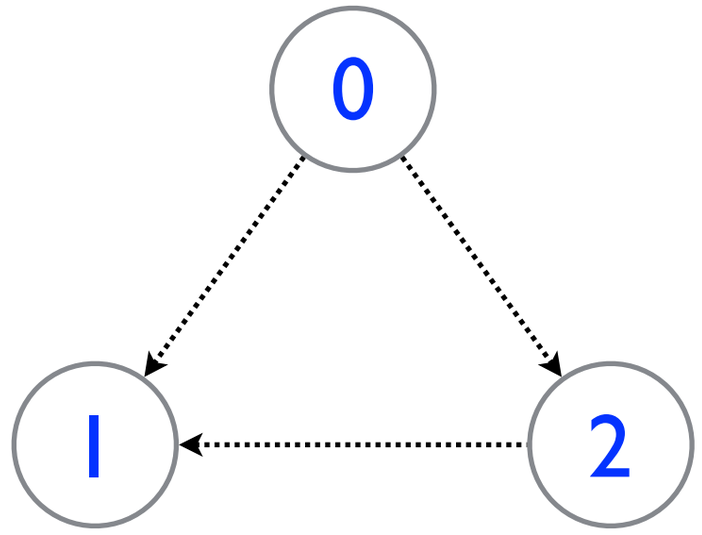

```
public class Edge implements Comparable<Edge>
{
  private int    source;
  private int    target;
  private double weight;

  public Edge(int source, int target, double weight)
  {
    setSource(source);
    setTarget(target);
    setWeight(weight);
  }
}
```

by weight

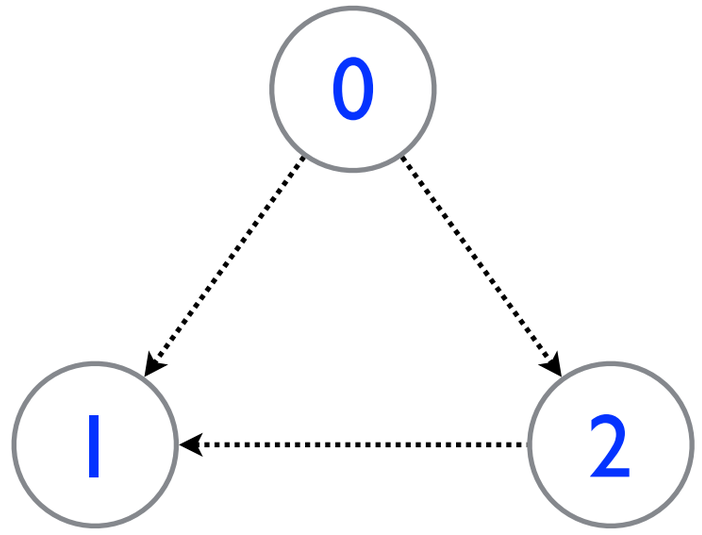
```
public int compareTo(Edge edge)
{
  double diff = weight - edge.weight;
  if      (diff > 0)  return  1;
  else if (diff < 0)  return -1;
  else                return  0;
}
```

Graph

Graph

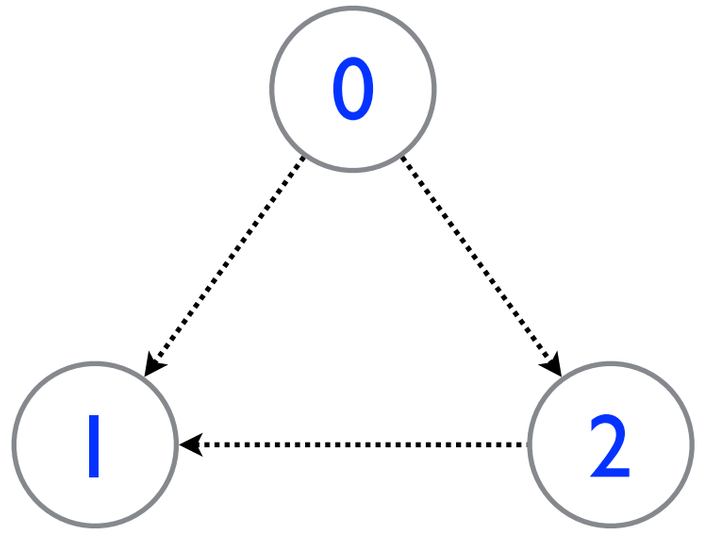


Graph




```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                     .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

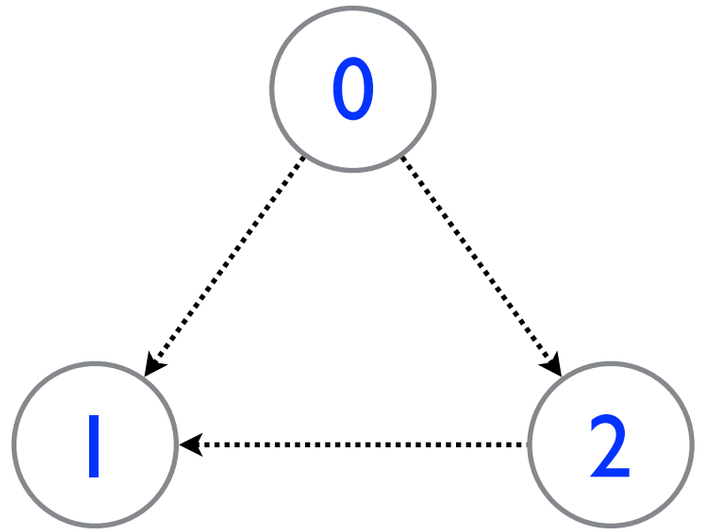
Graph




```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                     .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

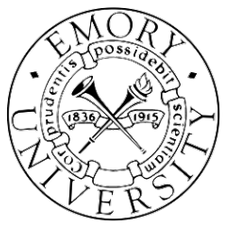
Graph

```
incoming_edges[0] = {};
incoming_edges[1] = {1 <- 0, 1 <- 2};
incoming_edges[2] = {2 <- 0};
```



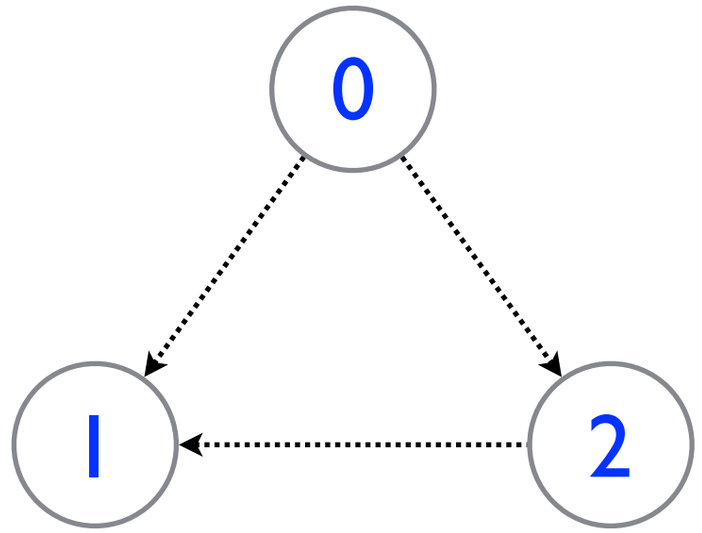

```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                       .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

EMORY
UNIVERSITY



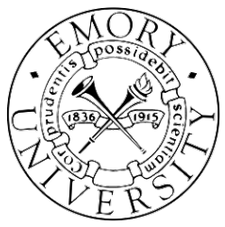
Graph

```
incoming_edges[0] = {};
incoming_edges[1] = {1 <- 0, 1 <- 2};
incoming_edges[2] = {2 <- 0};
```



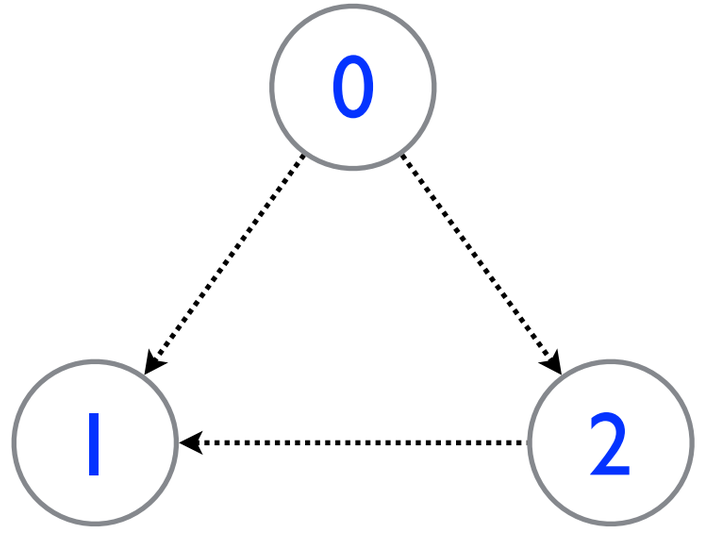

```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                     .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

EMORY
UNIVERSITY



Graph

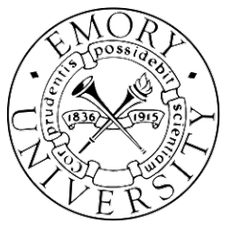
```
incoming_edges[0] = {};
incoming_edges[1] = {1 <- 0, 1 <- 2};
incoming_edges[2] = {2 <- 0};
```




```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                     .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

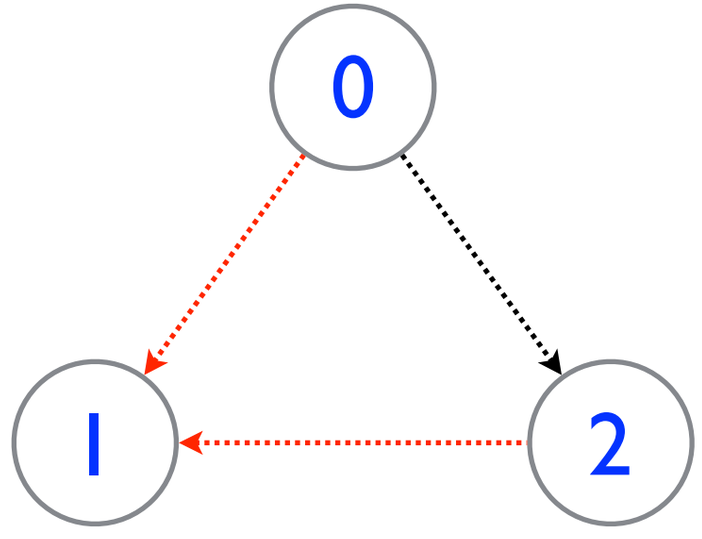
```
getIncomingEdges(1);
```

EMORY
UNIVERSITY



Graph

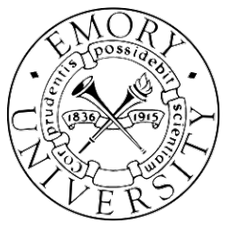
```
incoming_edges[0] = {};
incoming_edges[1] = {1 <- 0, 1 <- 2};
incoming_edges[2] = {2 <- 0};
```




```
public class Graph {
  private List<Edge>[] incoming_edges;

  public Graph(int size) {
    incoming_edges = (List<Edge>[])Stream.generate(ArrayList<Edge>::new)
                     .limit(size).toArray(List<?>[]::new);
  }

  public List<Edge> getIncomingEdges(int target) {
    return incoming_edges[target];
  }
}
```

```
getIncomingEdges(1);
```

EMORY
UNIVERSITY



Graph

Graph

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
    List<Edge> edges = getIncomingEdges(target);
    edges.add(new Edge(source, target, weight));
}
```

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}
```

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}
```

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}
```

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}
```

Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}

public void setUndirectedEdge(int source, int target, double weight)
{
  setDirectedEdge(source, target, weight);
  setDirectedEdge(target, source, weight);
}
```

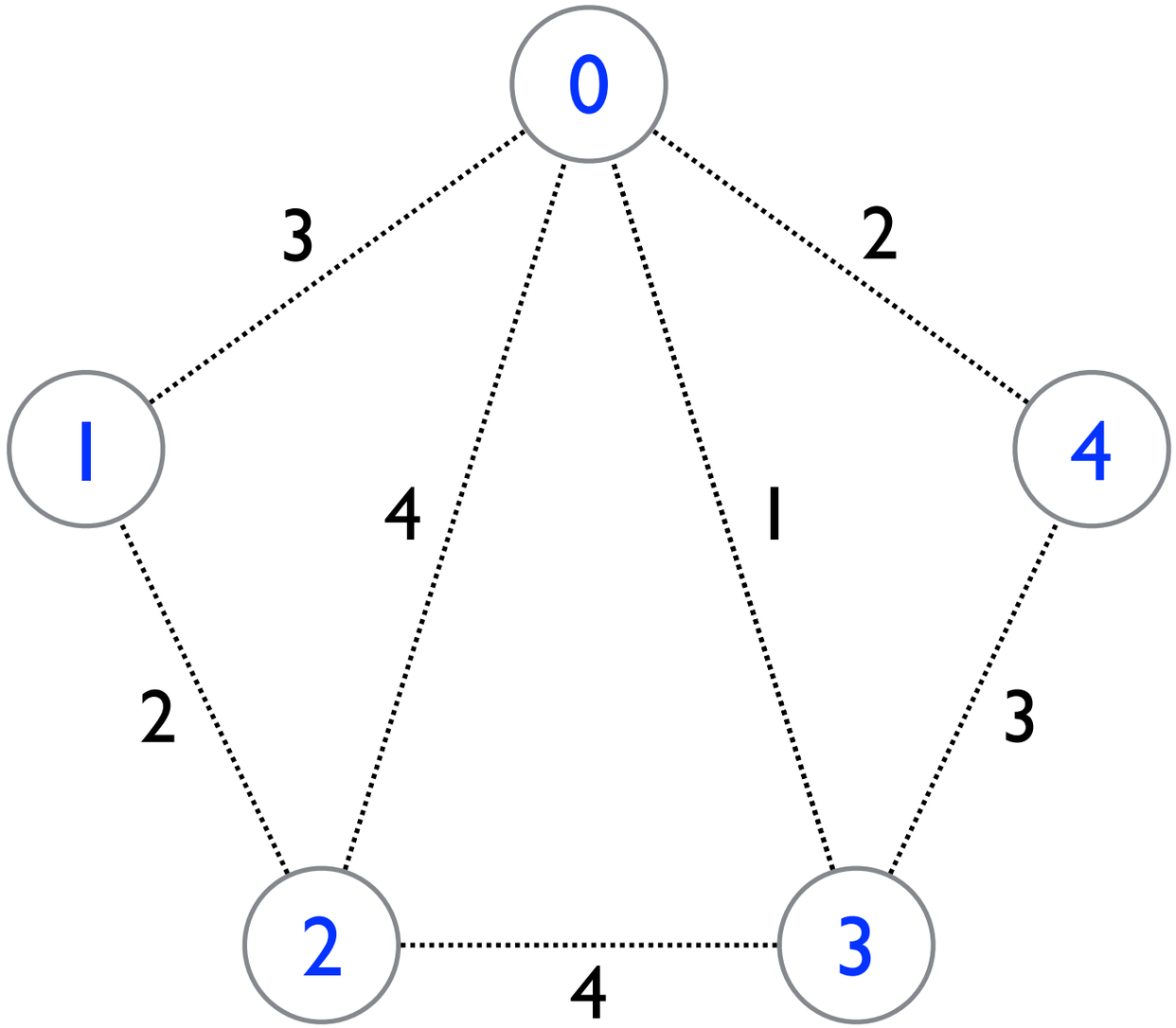
Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}

public void setUndirectedEdge(int source, int target, double weight)
{
  setDirectedEdge(source, target, weight);
  setDirectedEdge(target, source, weight);
}
```

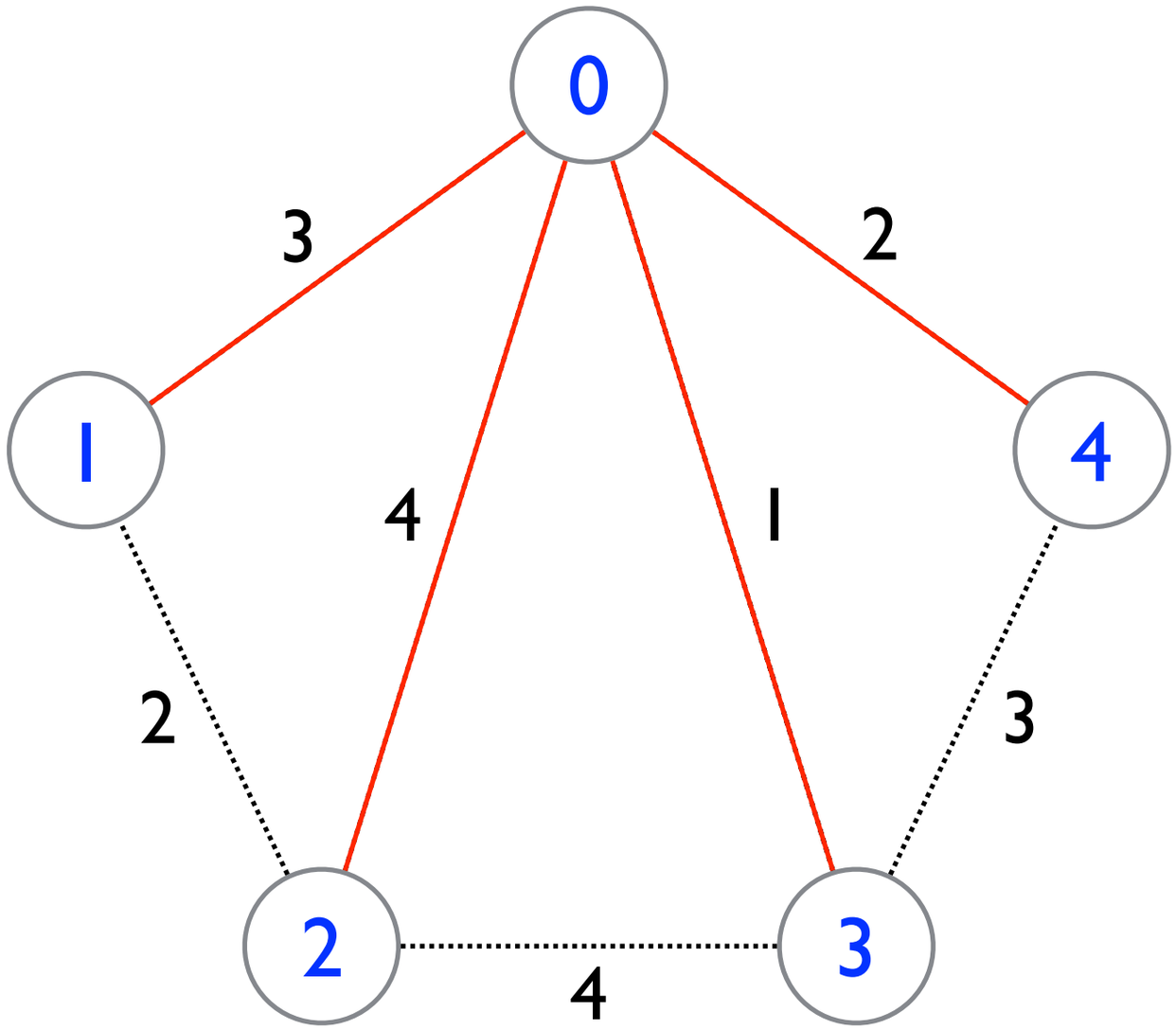
Graph


```
public void setDirectedEdge(int source, int target, double weight)
{
  List<Edge> edges = getIncomingEdges(target);
  edges.add(new Edge(source, target, weight));
}

public void setUndirectedEdge(int source, int target, double weight)
{
  setDirectedEdge(source, target, weight);
  setDirectedEdge(target, source, weight);
}
```

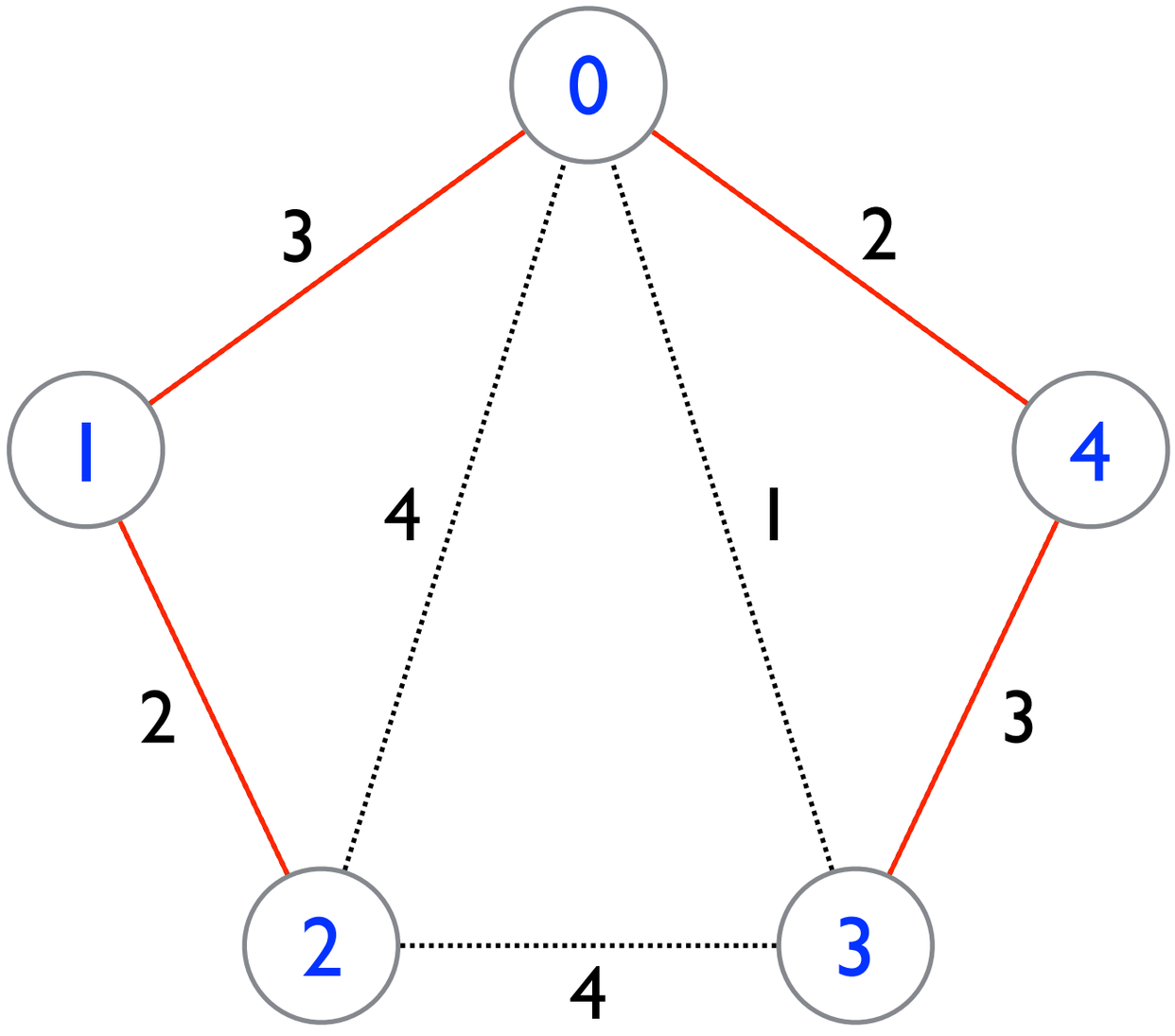
Minimum Spanning Tree



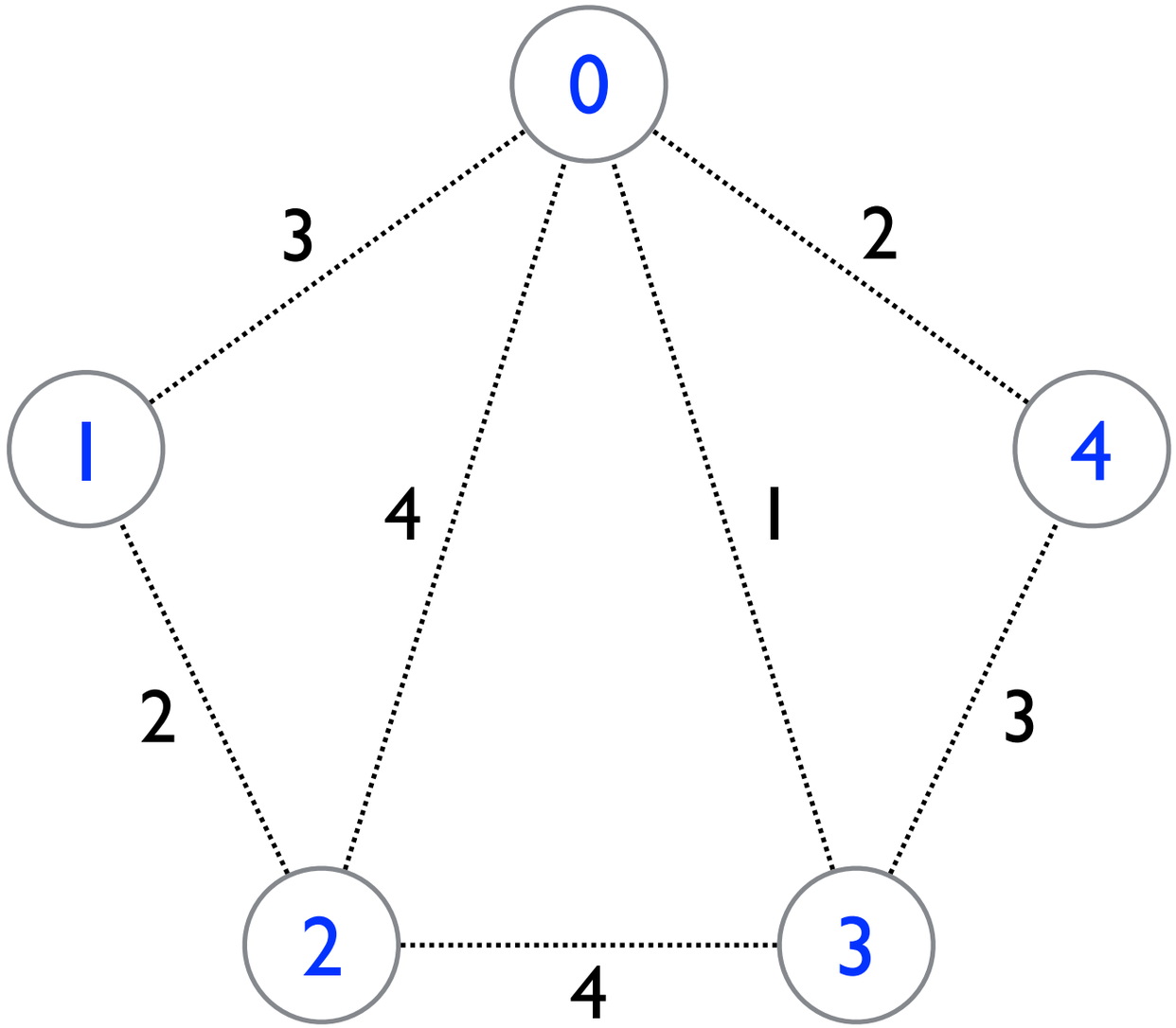
Minimum Spanning Tree



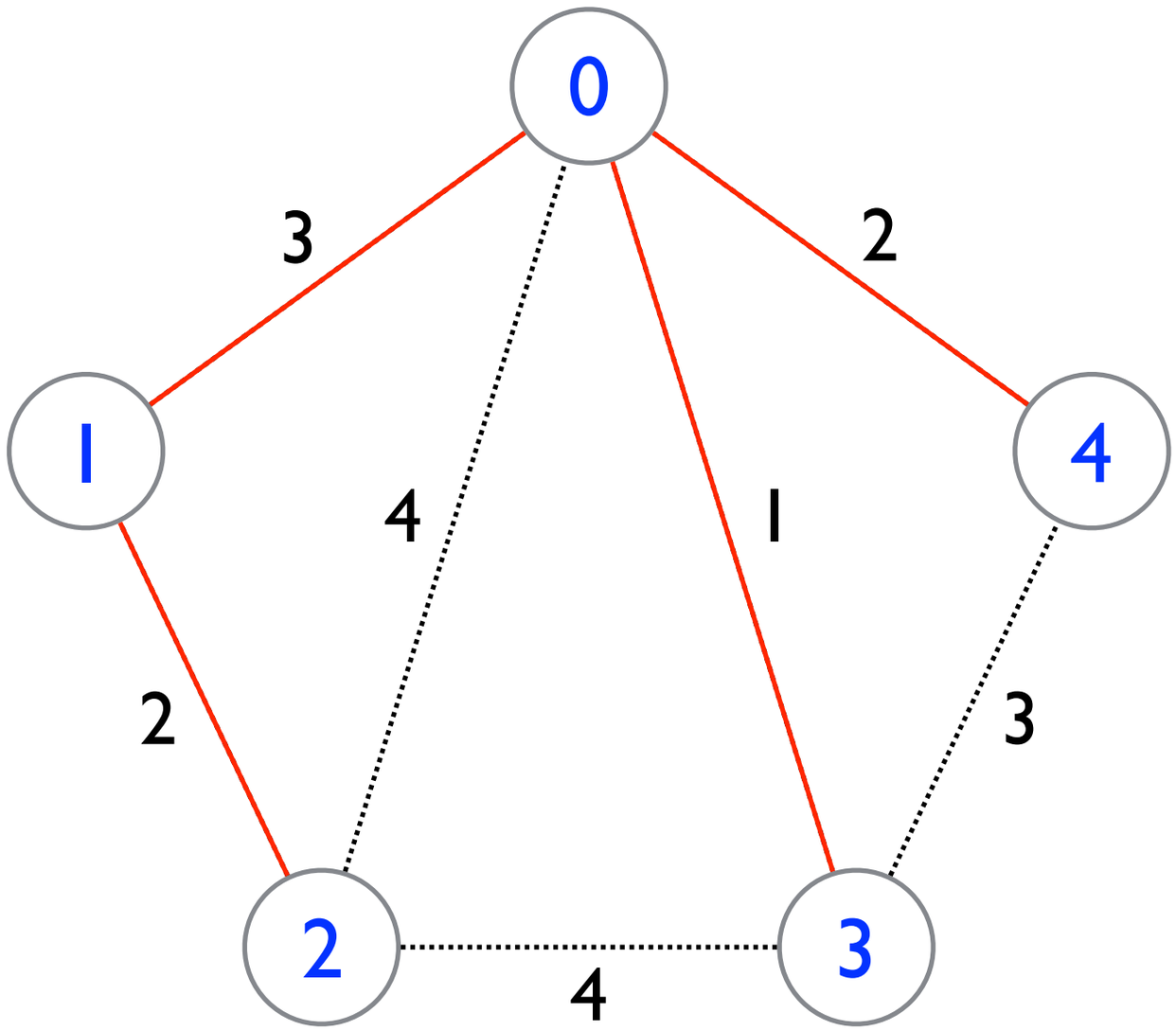
Minimum Spanning Tree



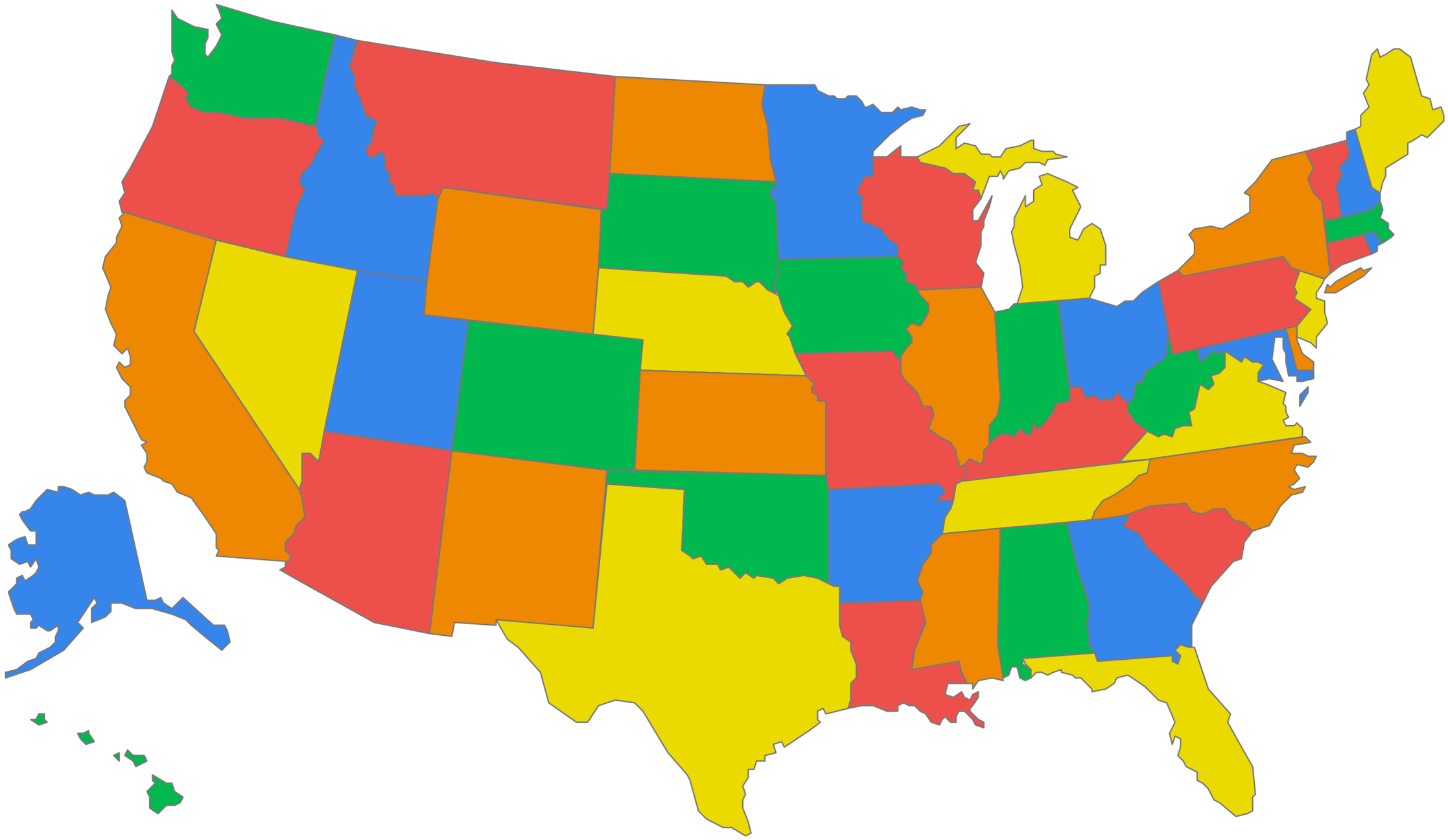
Minimum Spanning Tree



Minimum Spanning Tree

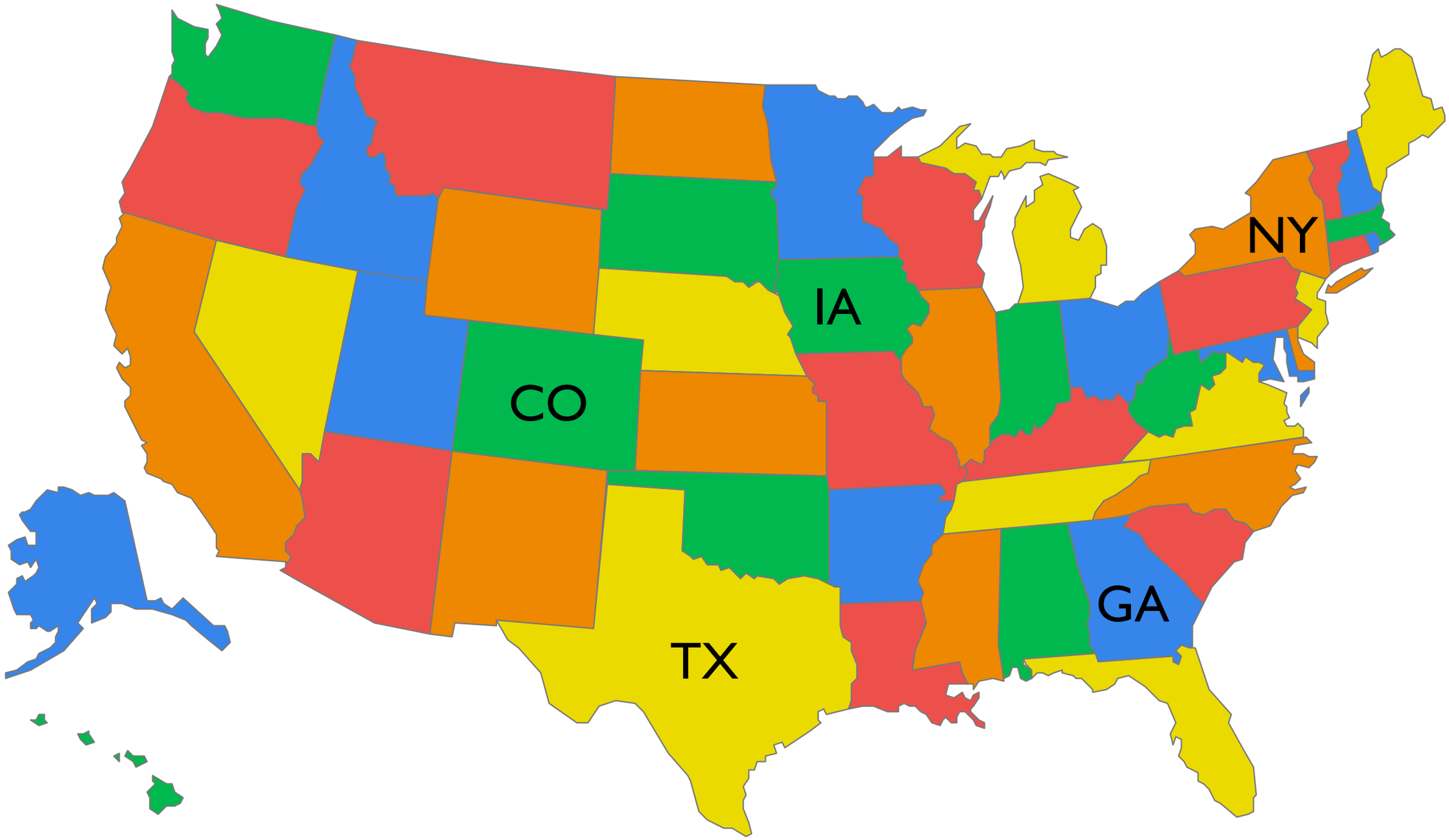


Minimum Spanning Tree

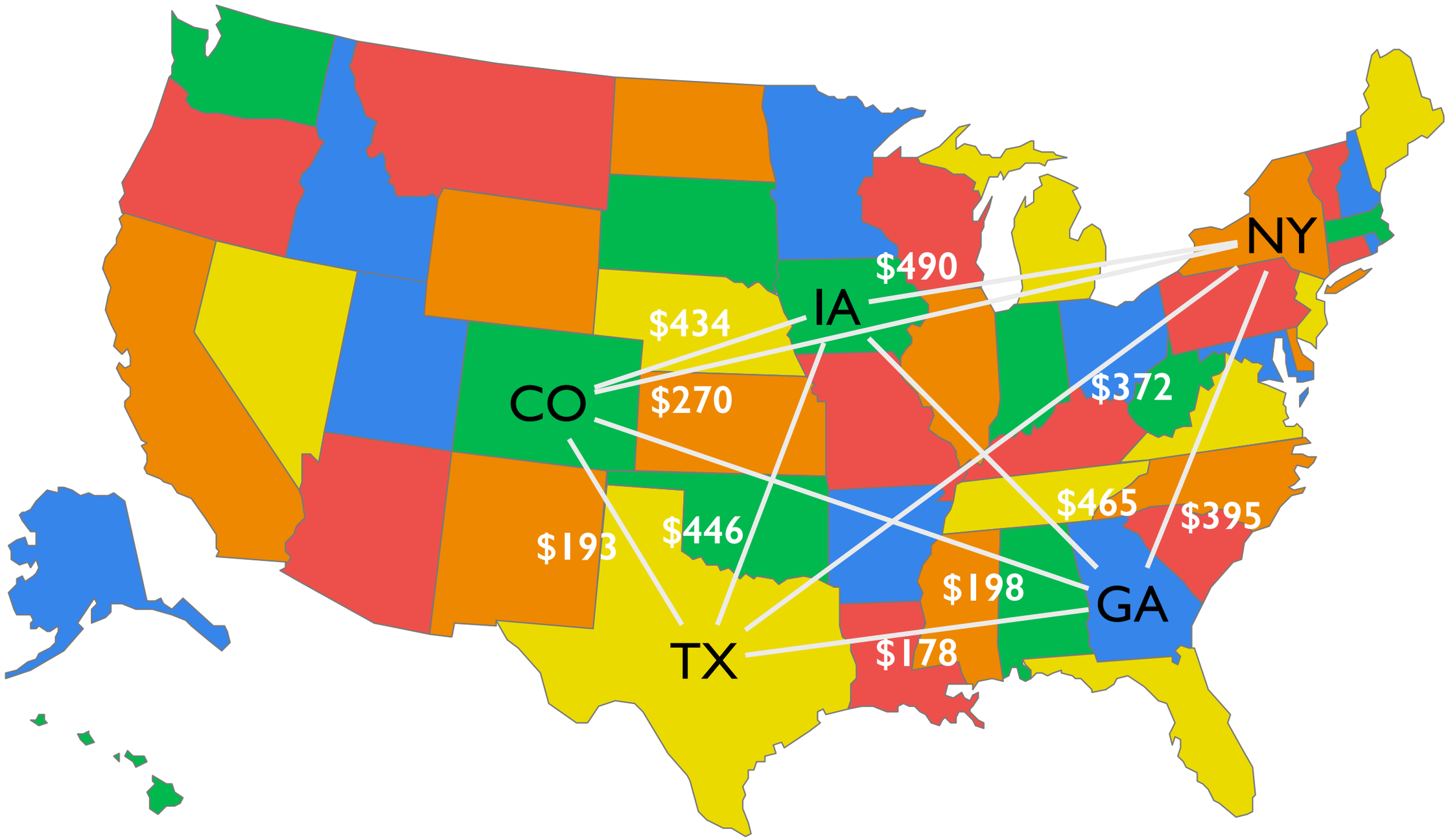


EMORY UNIVERSITY

Minimum Spanning Tree



Minimum Spanning Tree



Spanning Tree

```
public class SpanningTree implements Comparable<SpanningTree>
{
  private List<Edge> edges;
  private double total_weight;

  public SpanningTree()
  {
    edges = new ArrayList<>();
  }

  public void addEdge(Edge edge)
  {
    edges.add(edge);
    total_weight += edge.getWeight();
  }
}
```

```
public int size()
{
  return edges.size();
}

public double getTotalWeight()
{
  return total_weight;
}

public int compareTo(SpanningTree tree)
{
  double diff = total_weight
              - tree.total_weight;
  if      (diff > 0) return 1;
  else if (diff < 0) return -1;
  else               return 0;
}
```

Spanning Tree

```
public class SpanningTree implements Comparable<SpanningTree>
{
  private List<Edge> edges;
  private double total_weight;

  public SpanningTree()
  {
    edges = new ArrayList<>();
  }

  public void addEdge(Edge edge)
  {
    edges.add(edge);
    total_weight += edge.getWeight();
  }
}
```

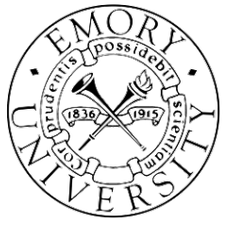
```
  public int size()
  {
    return edges.size();
  }

  public double getTotalWeight()
  {
    return total_weight;
  }

  public int compareTo(SpanningTree tree)
  {
    double diff = total_weight
                - tree.total_weight;
    if      (diff > 0) return 1;
    else if (diff < 0) return -1;
    else               return 0;
  }
```

Spanning Tree

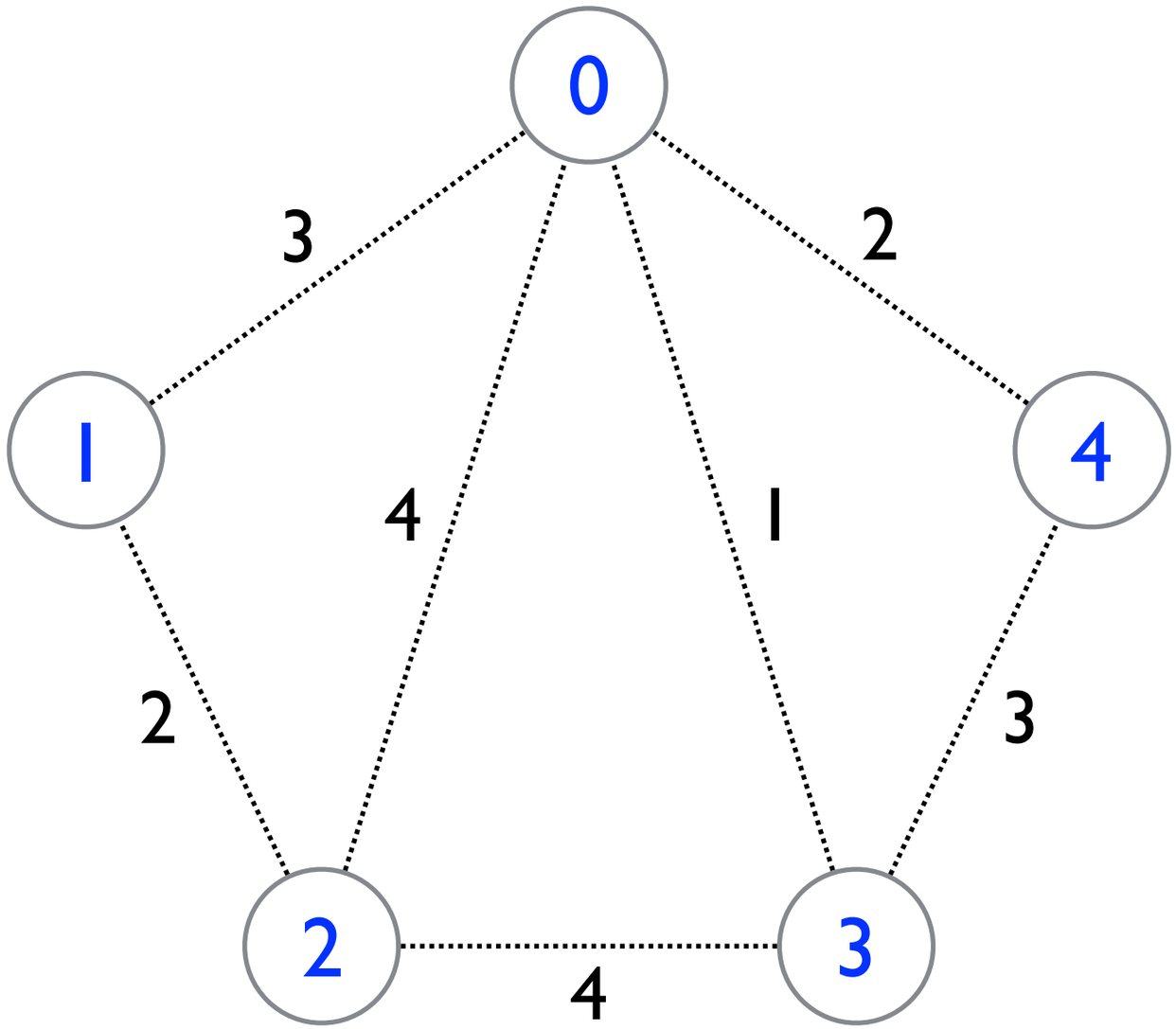
```
public class SpanningTree implements Comparable<SpanningTree>
{
  private List<Edge> edges;
  private double total_weight;

  public SpanningTree()
  {
    edges = new ArrayList<>();
  }

  public void addEdge(Edge edge)
  {
    edges.add(edge);
    total_weight += edge.getWeight();
  }
}
```

```
public int size()
{
  return edges.size();
}

public double getTotalWeight()
{
  return total_weight;
}

public int compareTo(SpanningTree tree)
{
  double diff = total_weight
              - tree.total_weight;
  if      (diff > 0) return 1;
  else if (diff < 0) return -1;
  else               return 0;
}
```

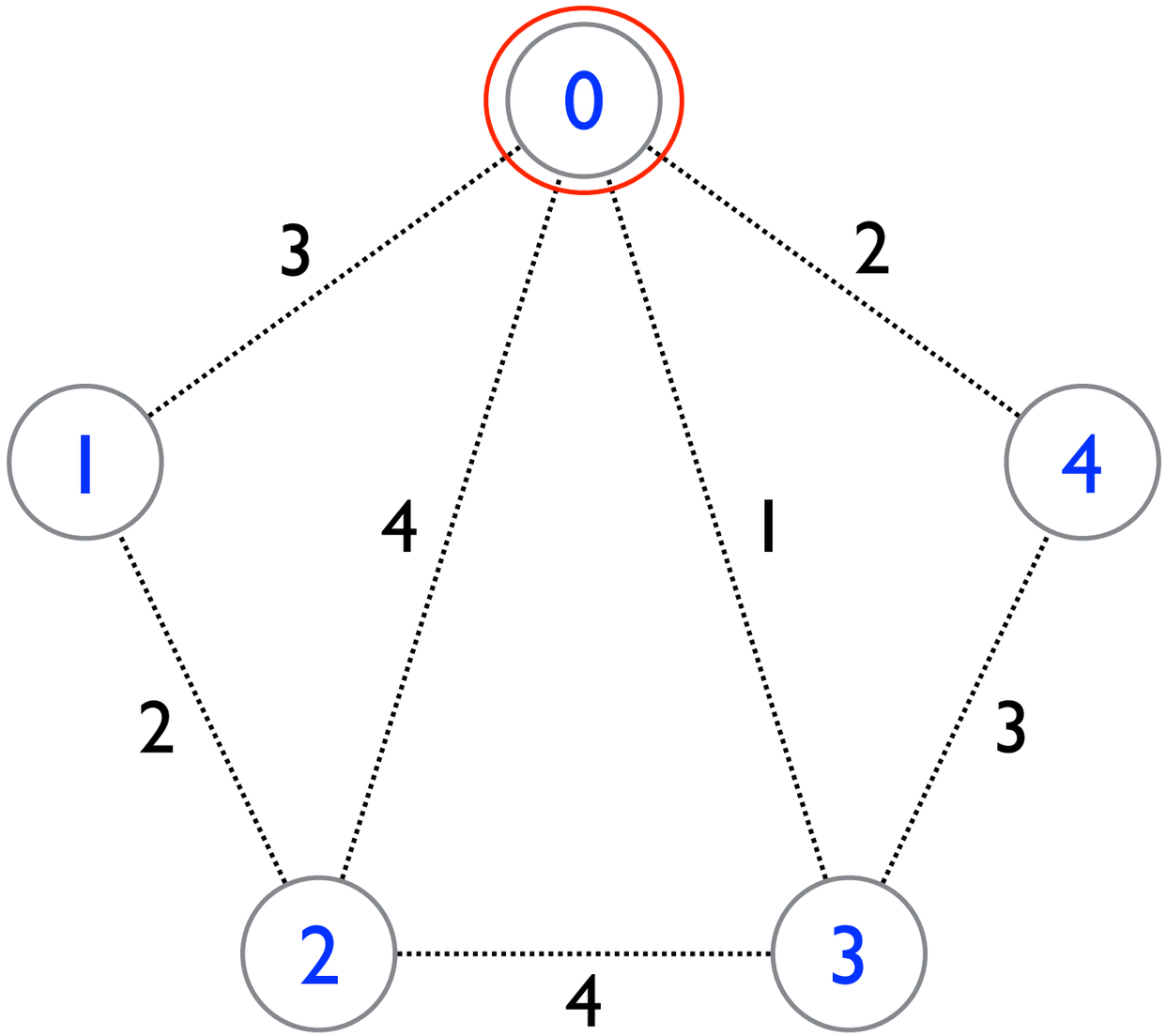
EMORY
UNIVERSITY



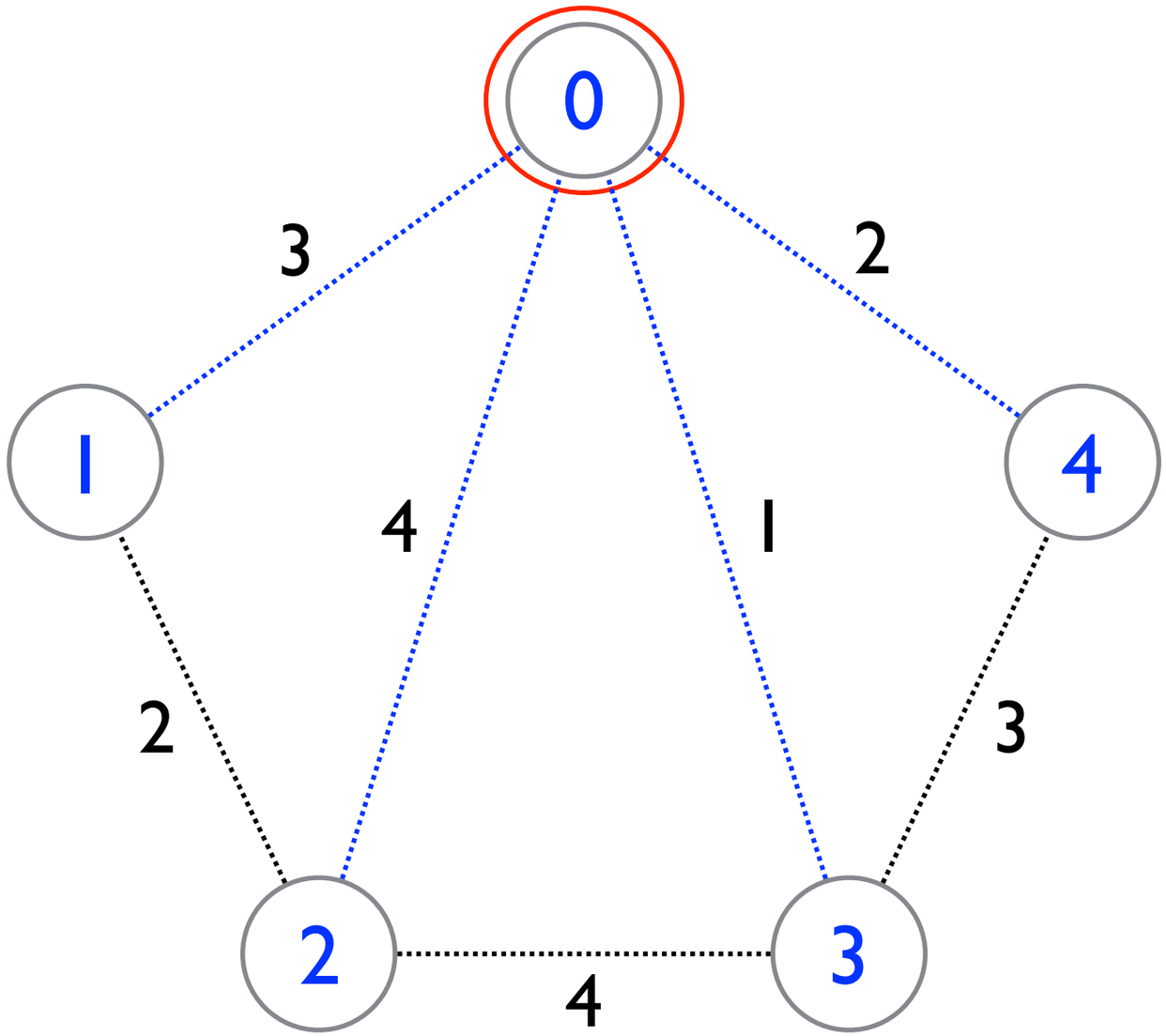
Prim's Algorithm



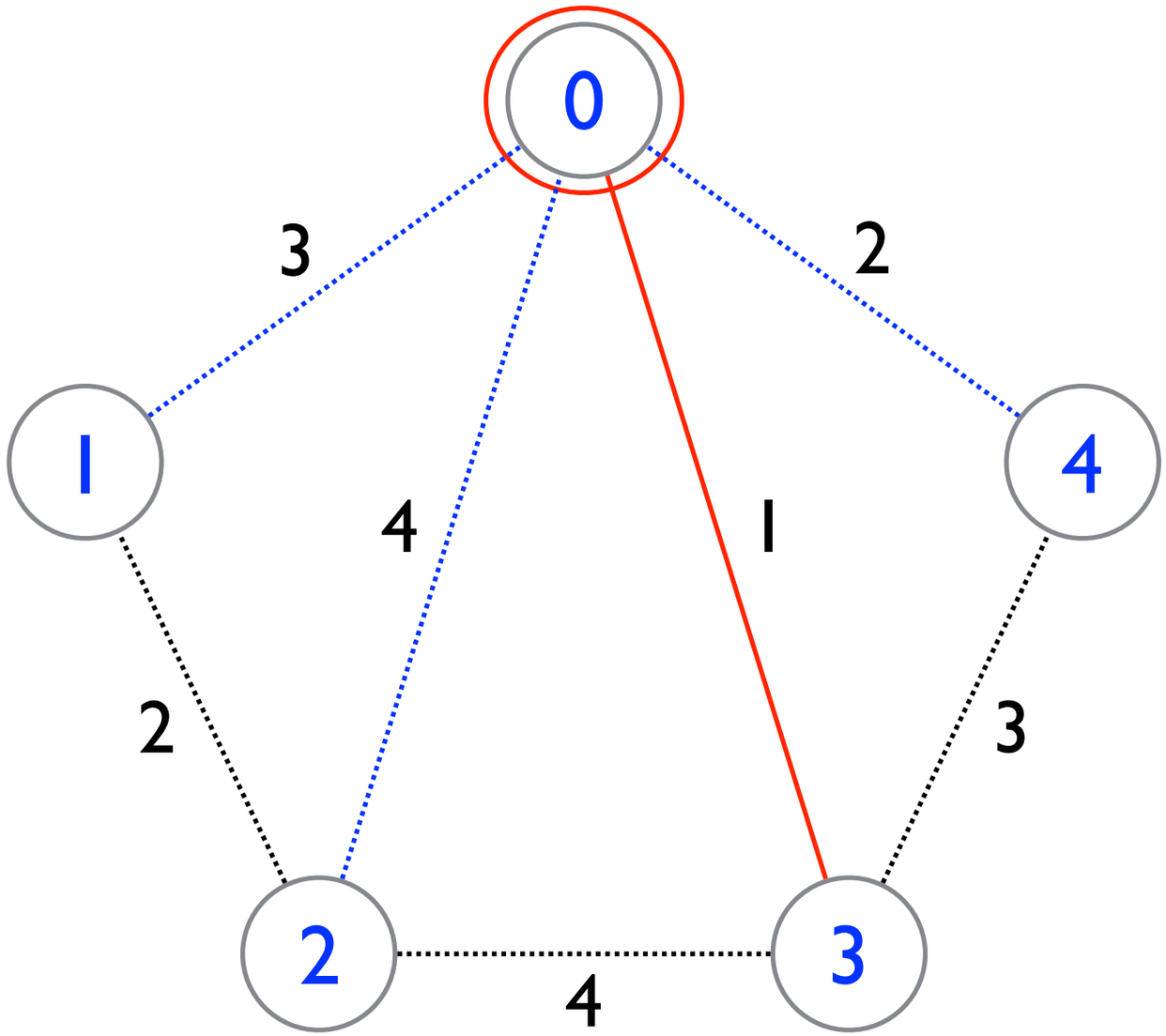
Prim's Algorithm



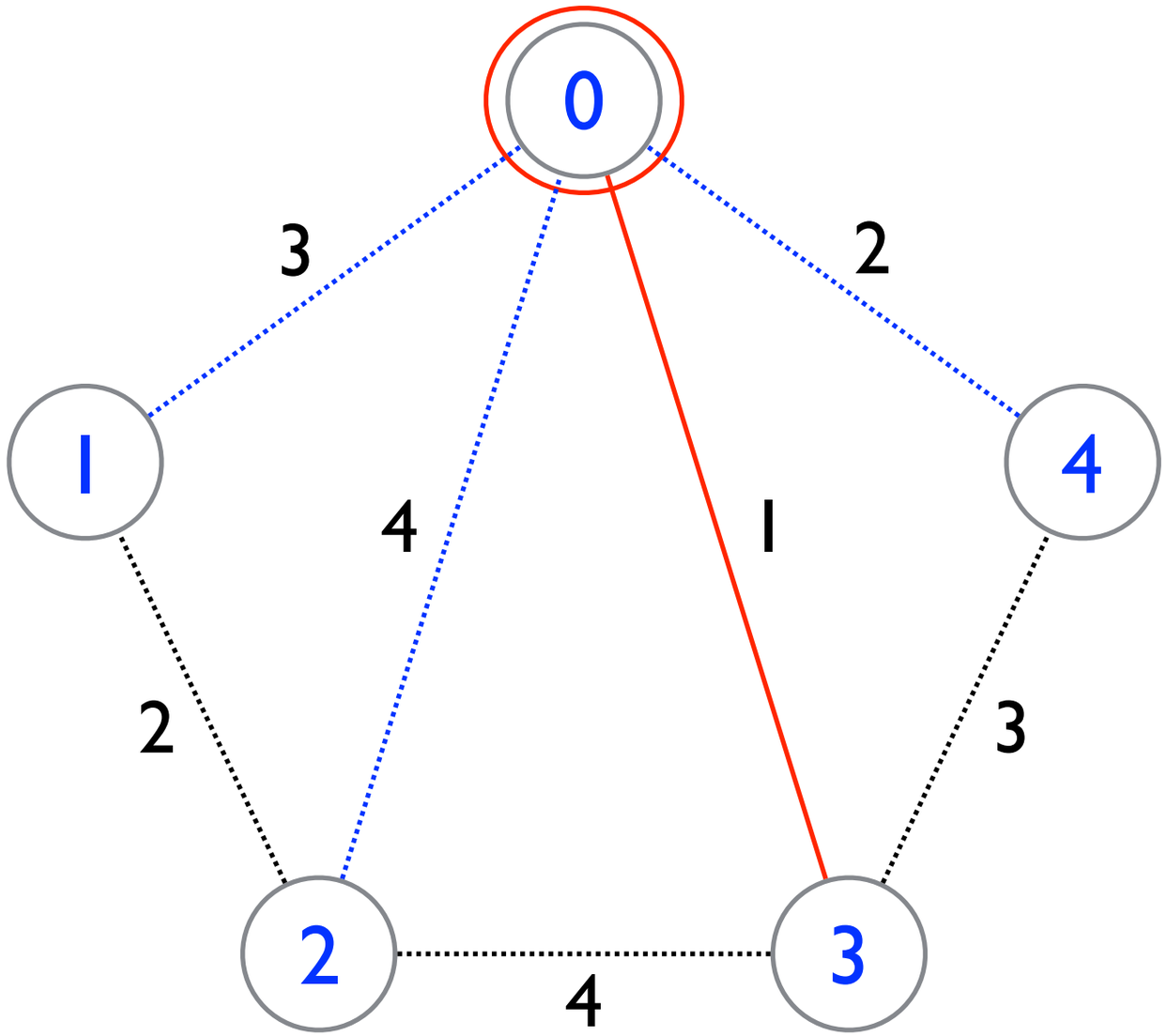
Prim's Algorithm



Prim's Algorithm

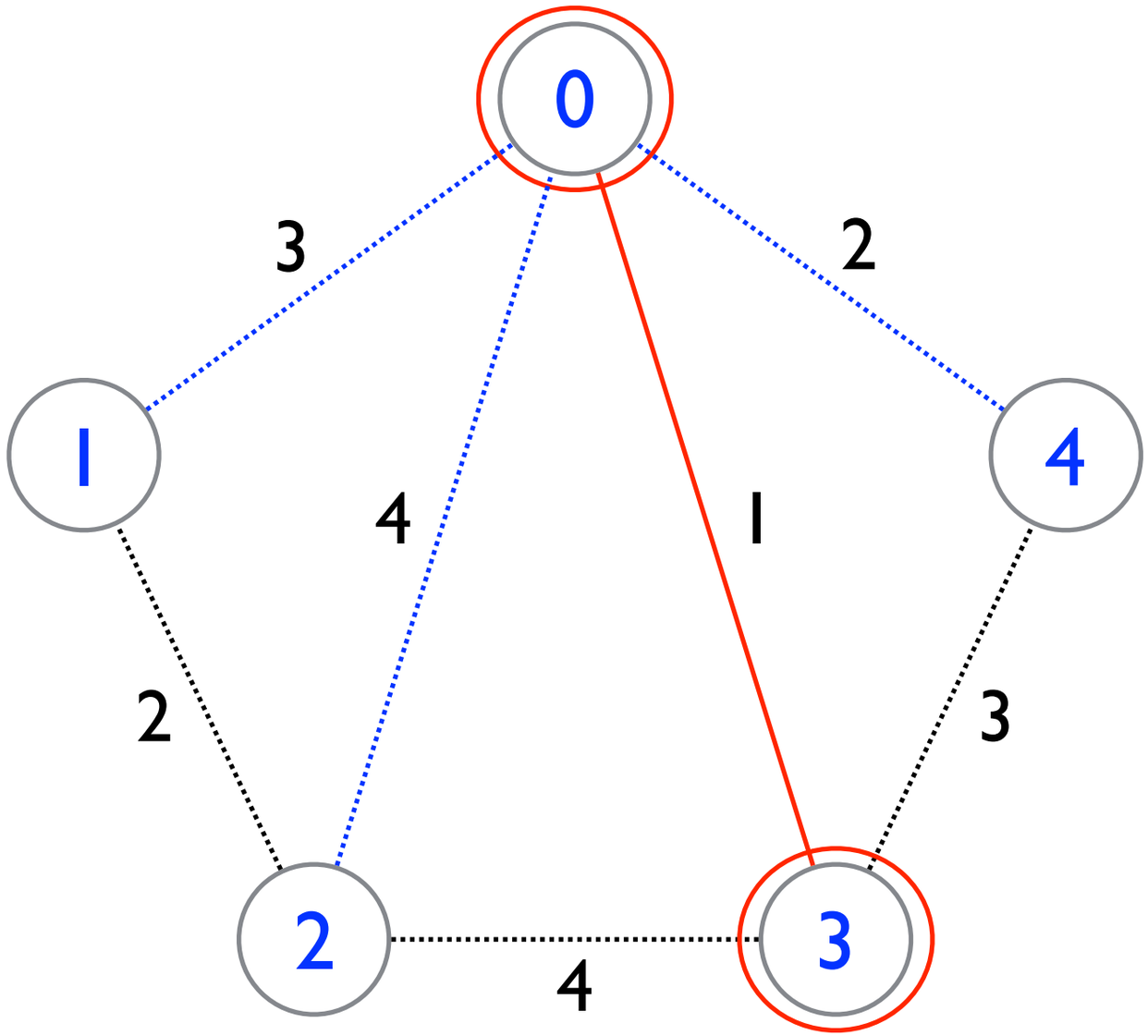


Prim's Algorithm



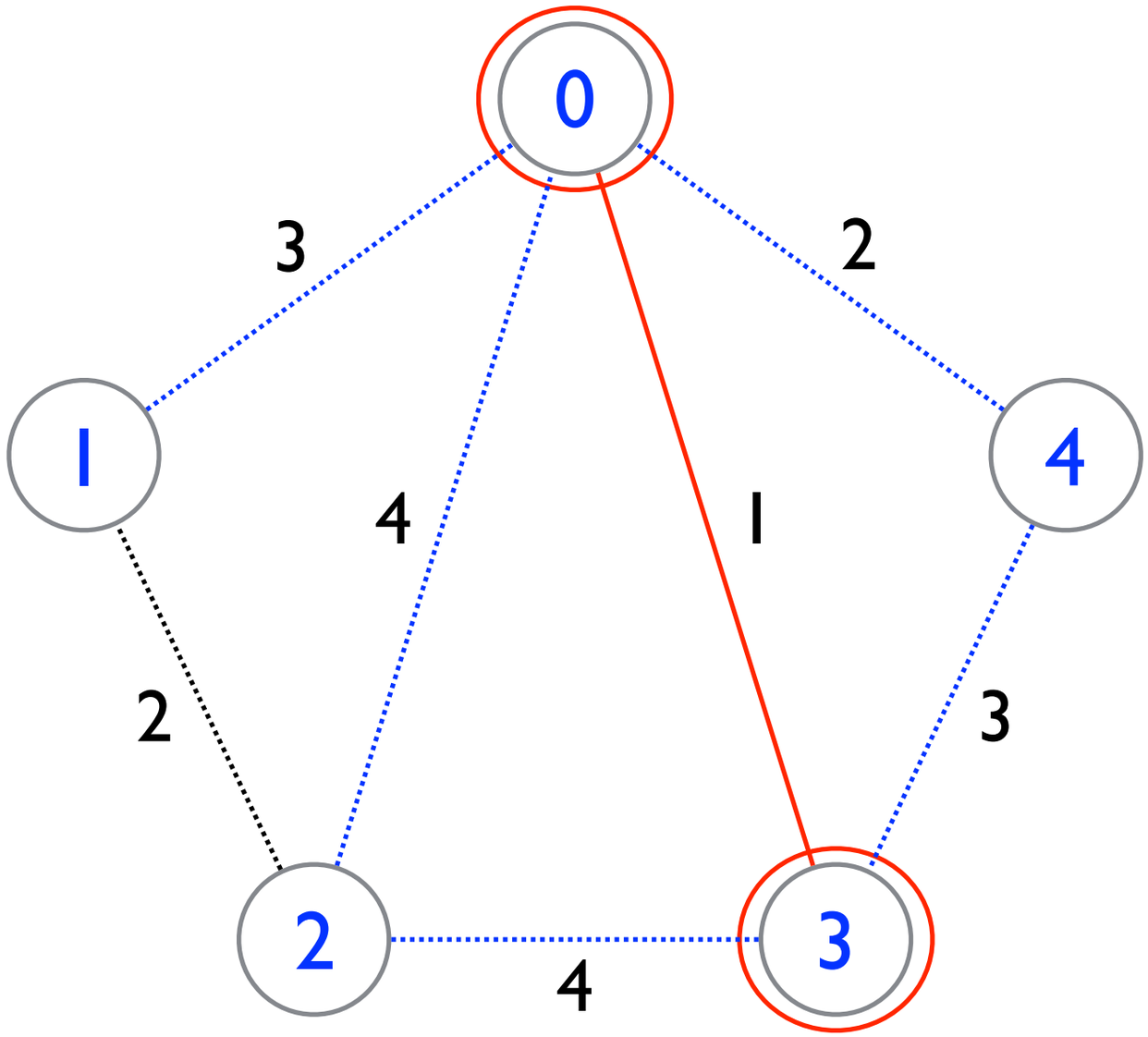

```
0 <- 3
```

Prim's Algorithm



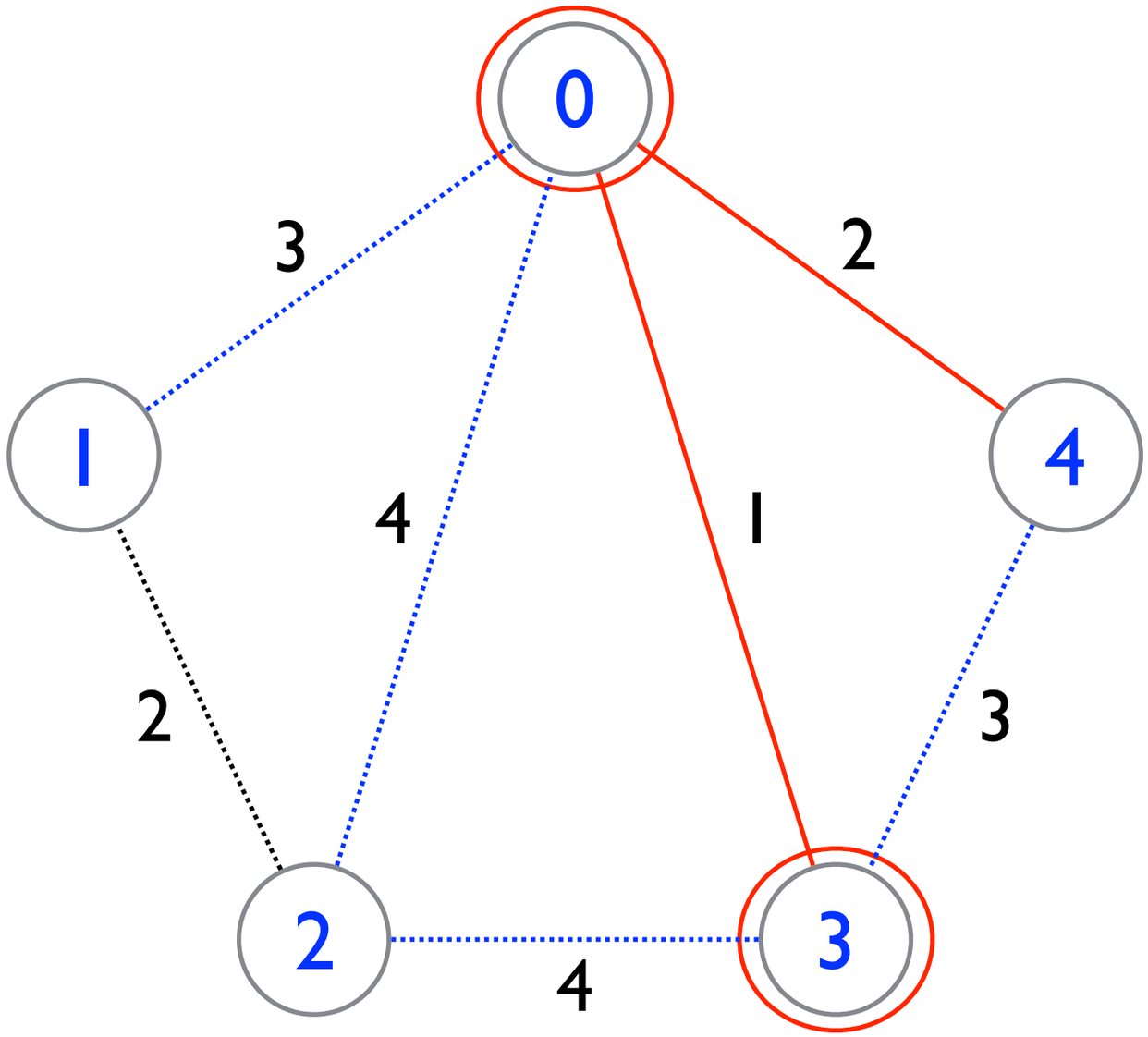
0 <- 3

Prim's Algorithm



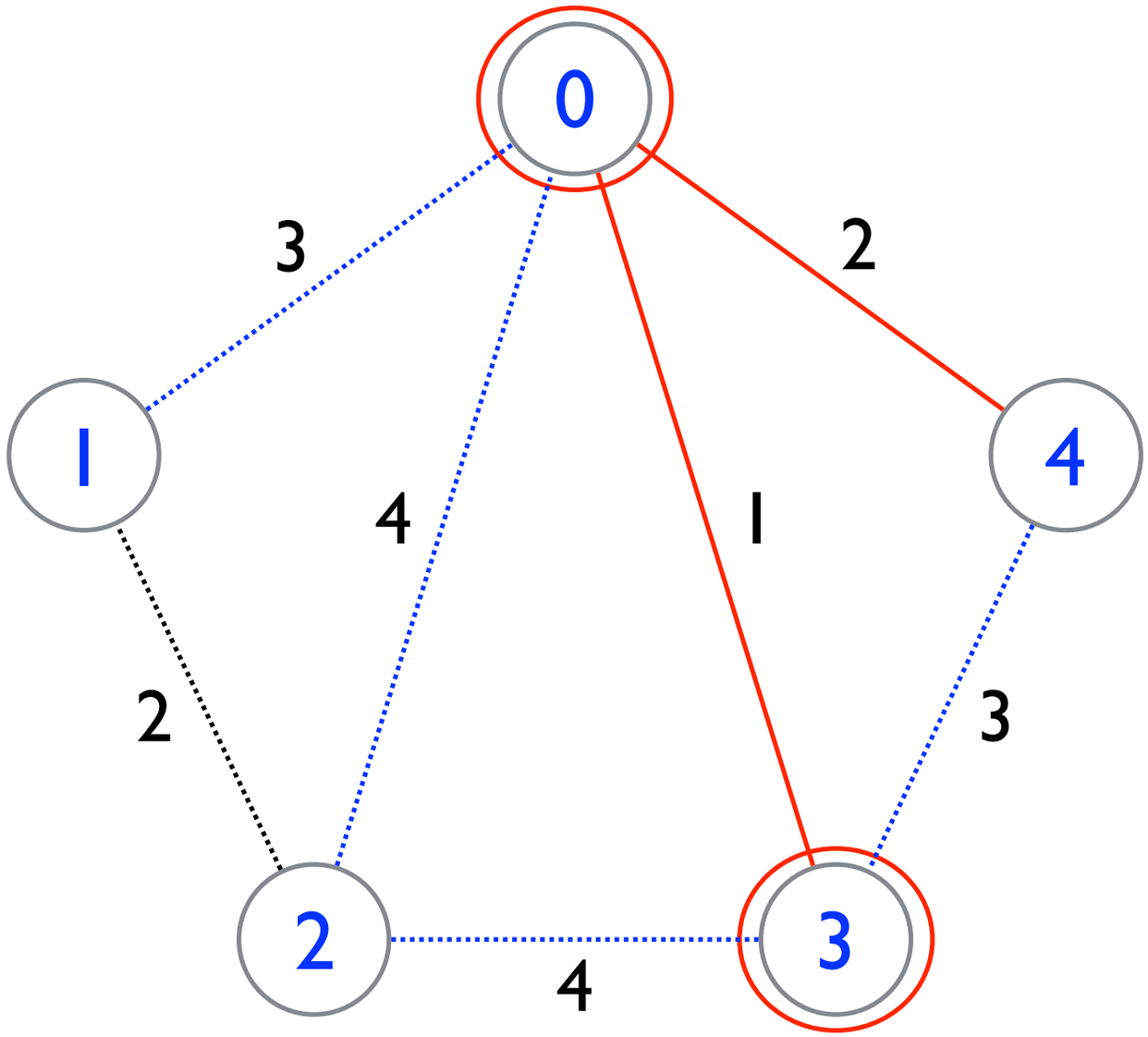
0 <- 3

Prim's Algorithm



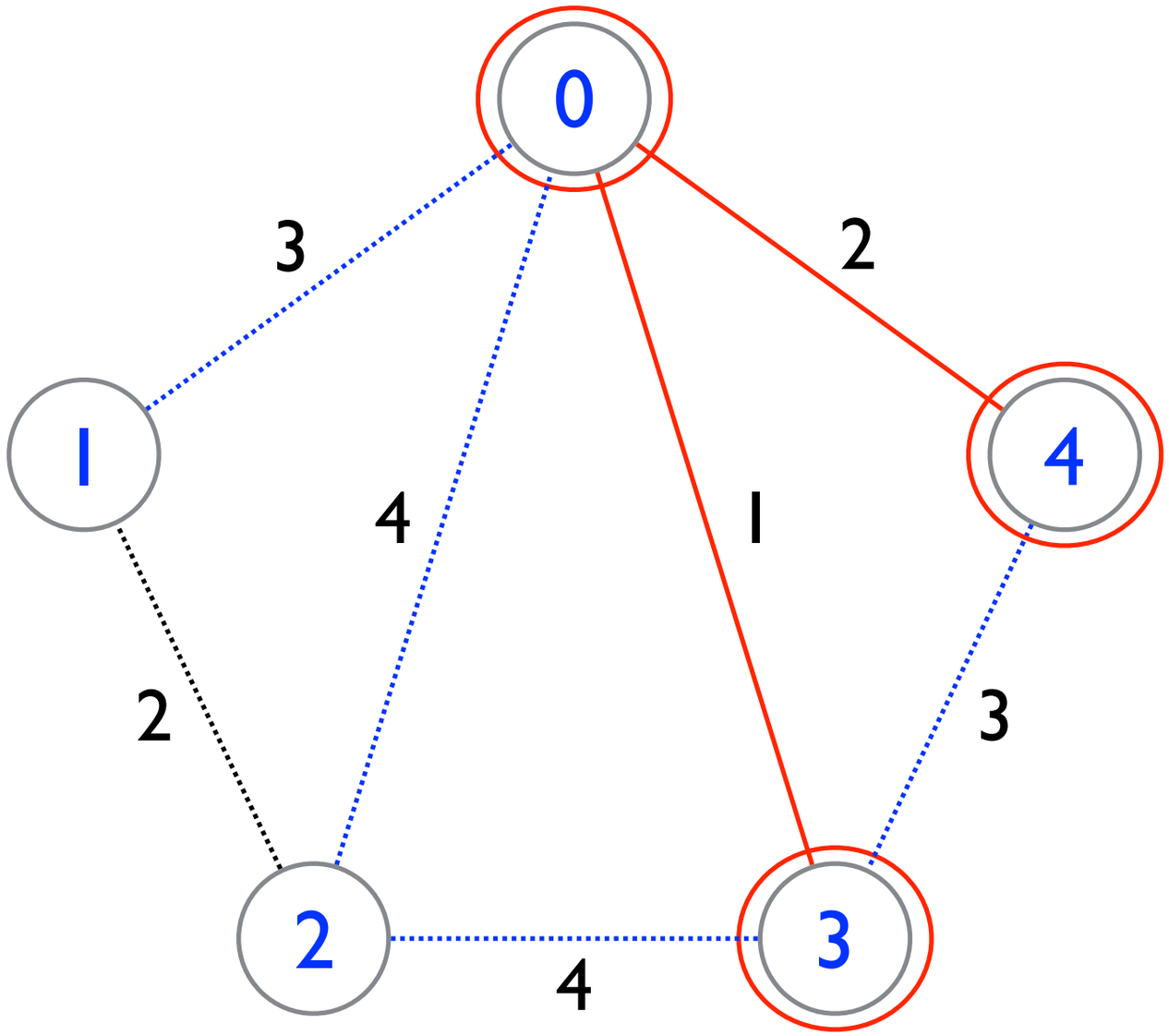
0 <- 3

Prim's Algorithm



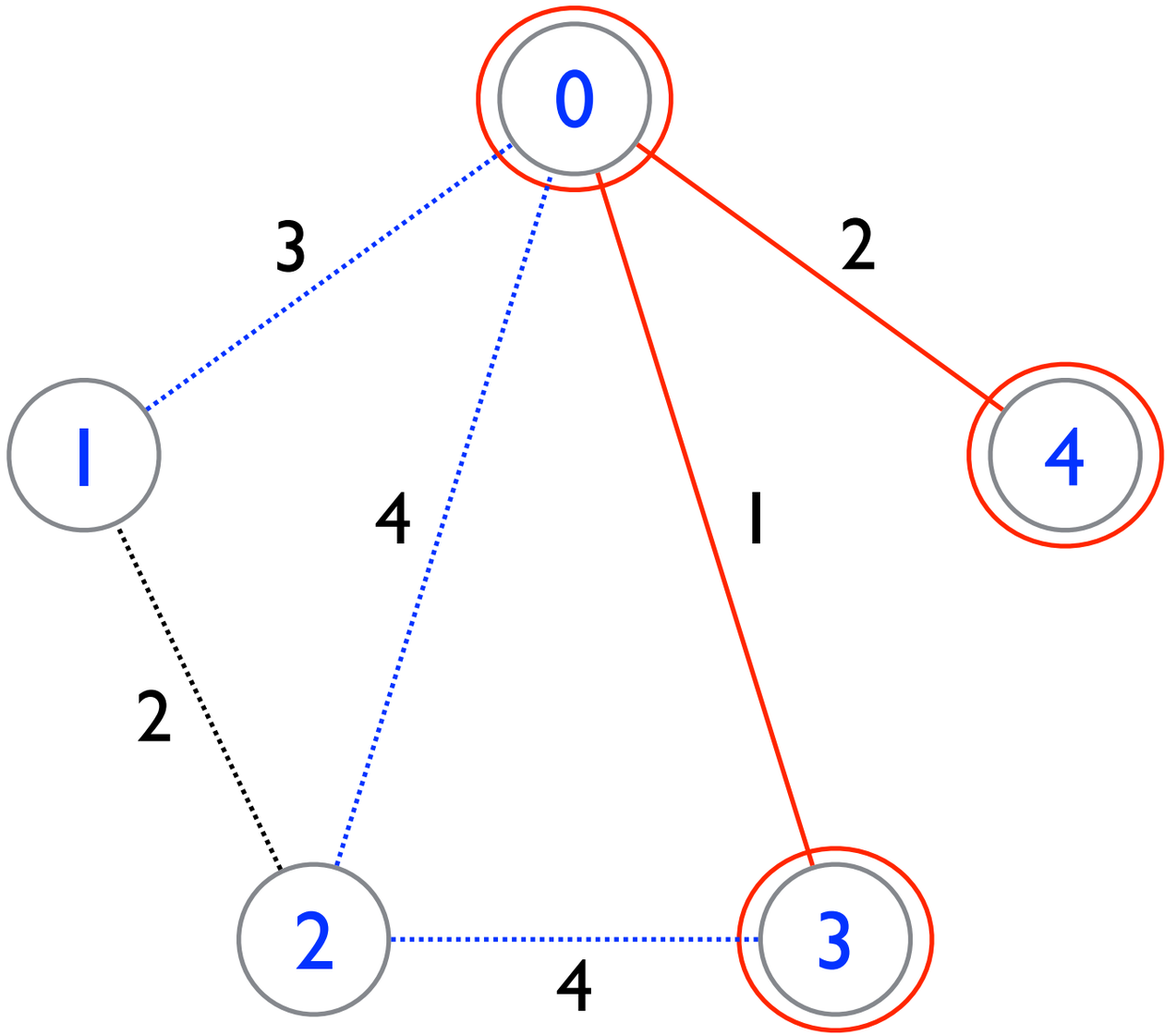

```
0 <- 3
0 <- 4
```

Prim's Algorithm



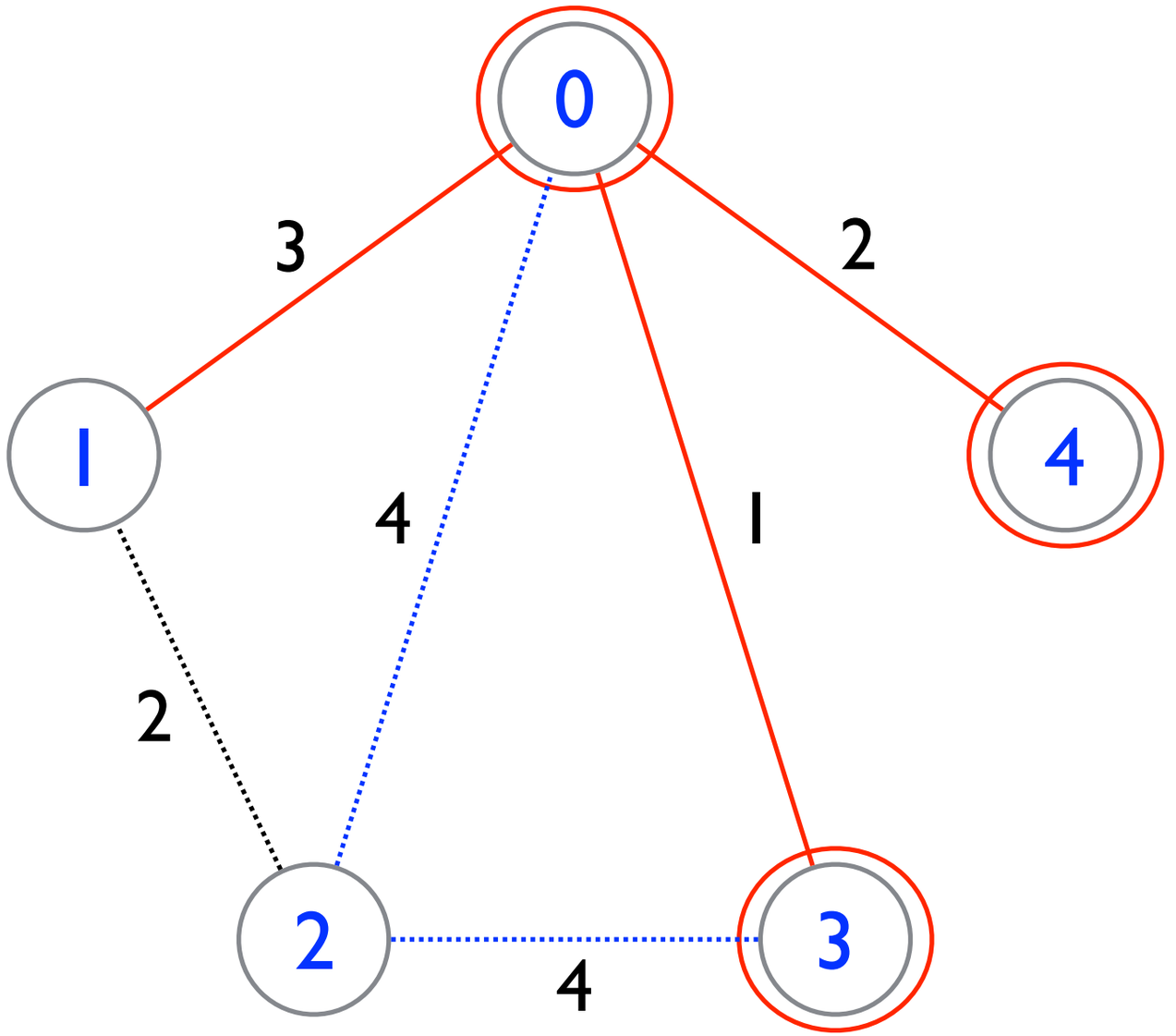

```
0 <- 3
0 <- 4
```

Prim's Algorithm




```
0 <- 3
0 <- 4
```

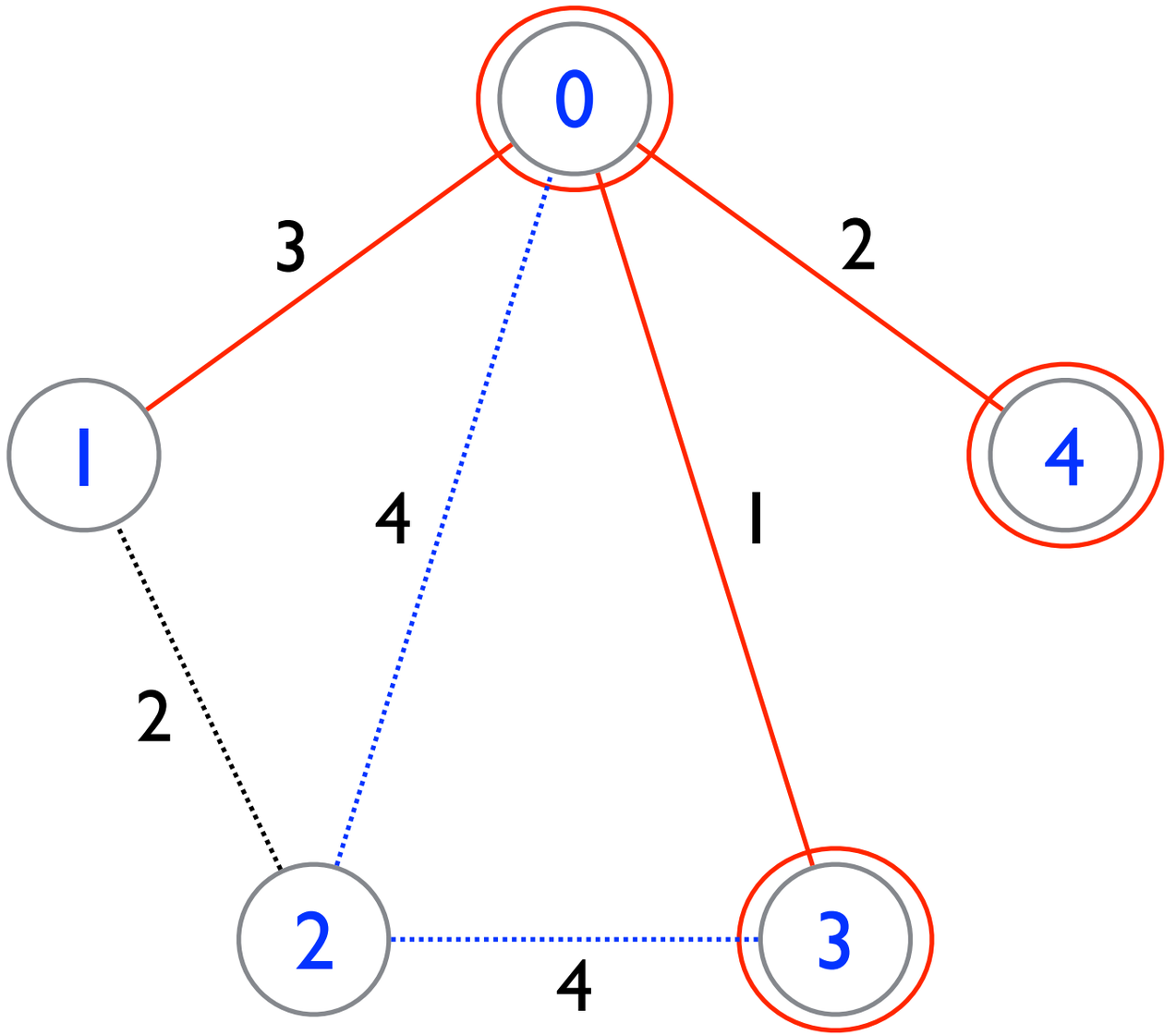
Prim's Algorithm



0 <- 3

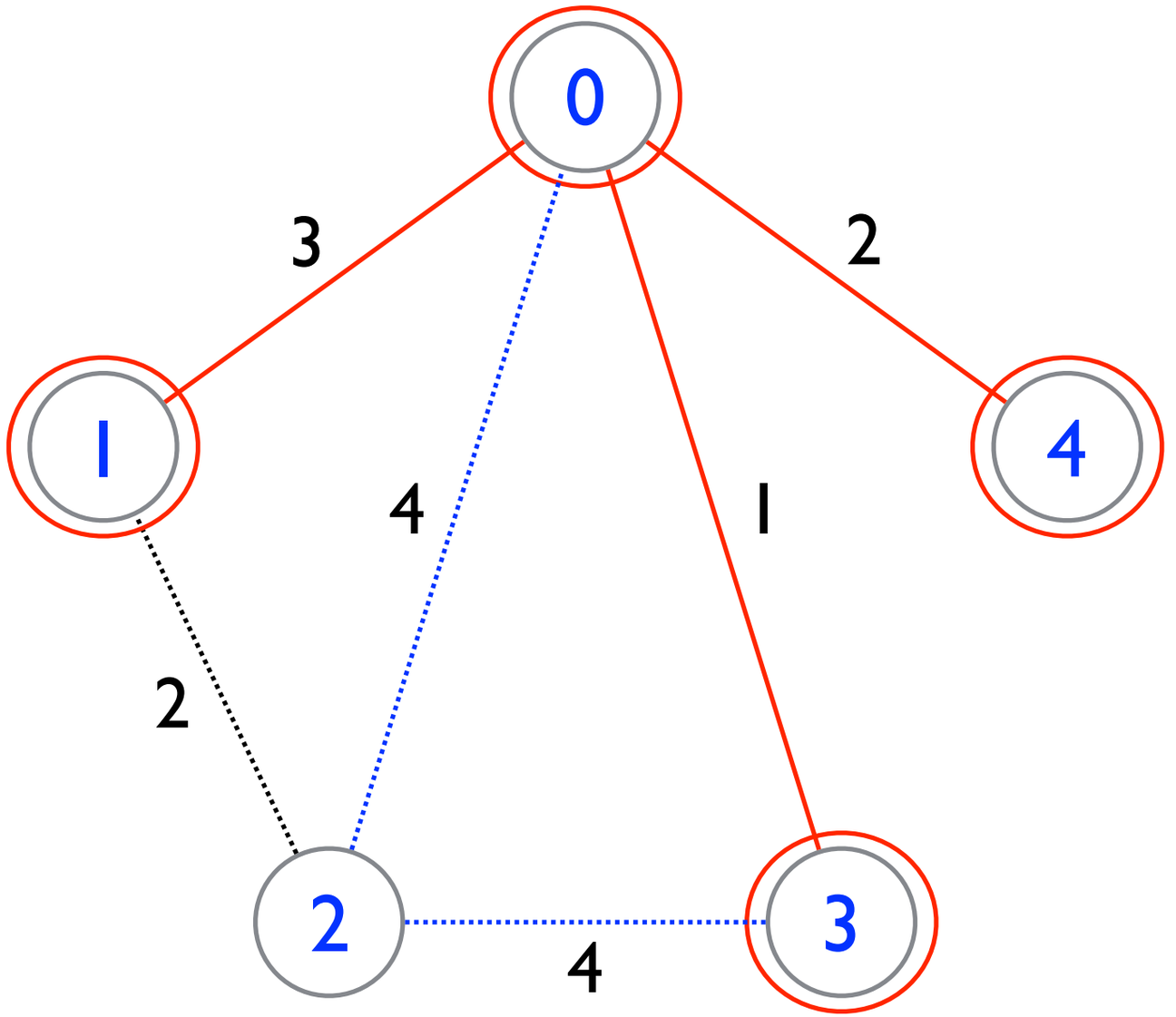
0 <- 4

Prim's Algorithm



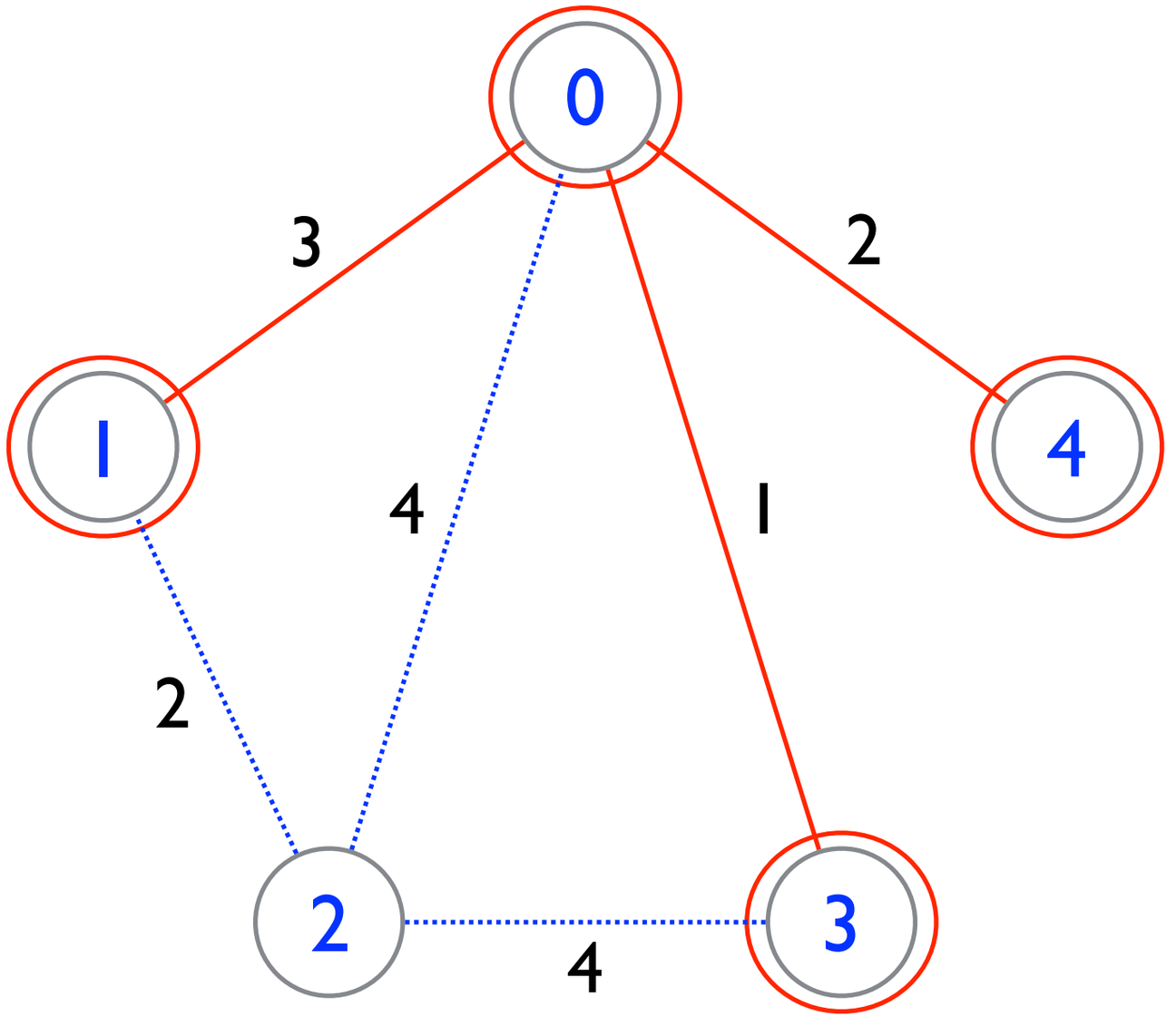

```
0 <- 3
0 <- 4
0 <- 1
```

Prim's Algorithm



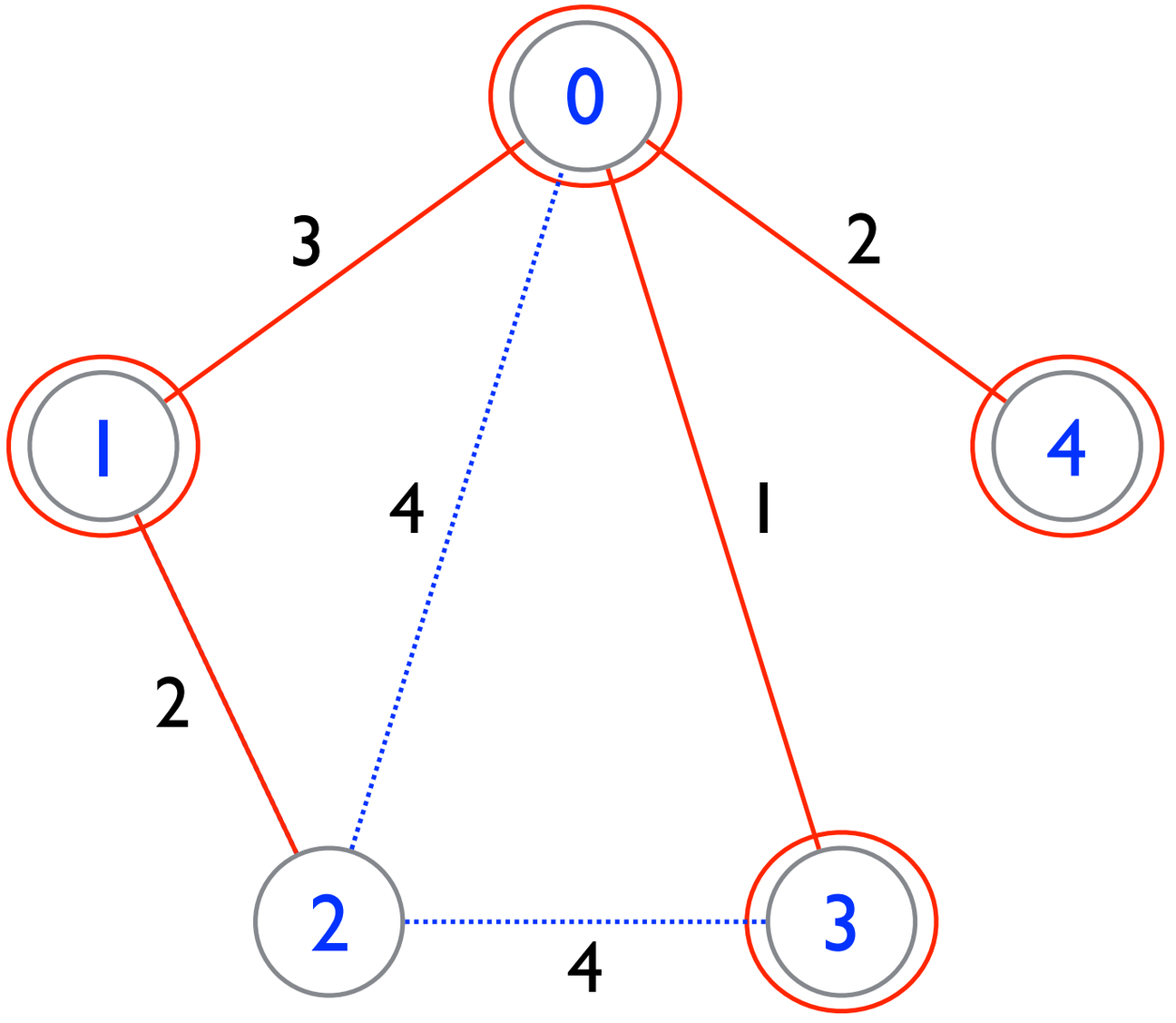

```
0 <- 3
0 <- 4
0 <- 1
```

Prim's Algorithm



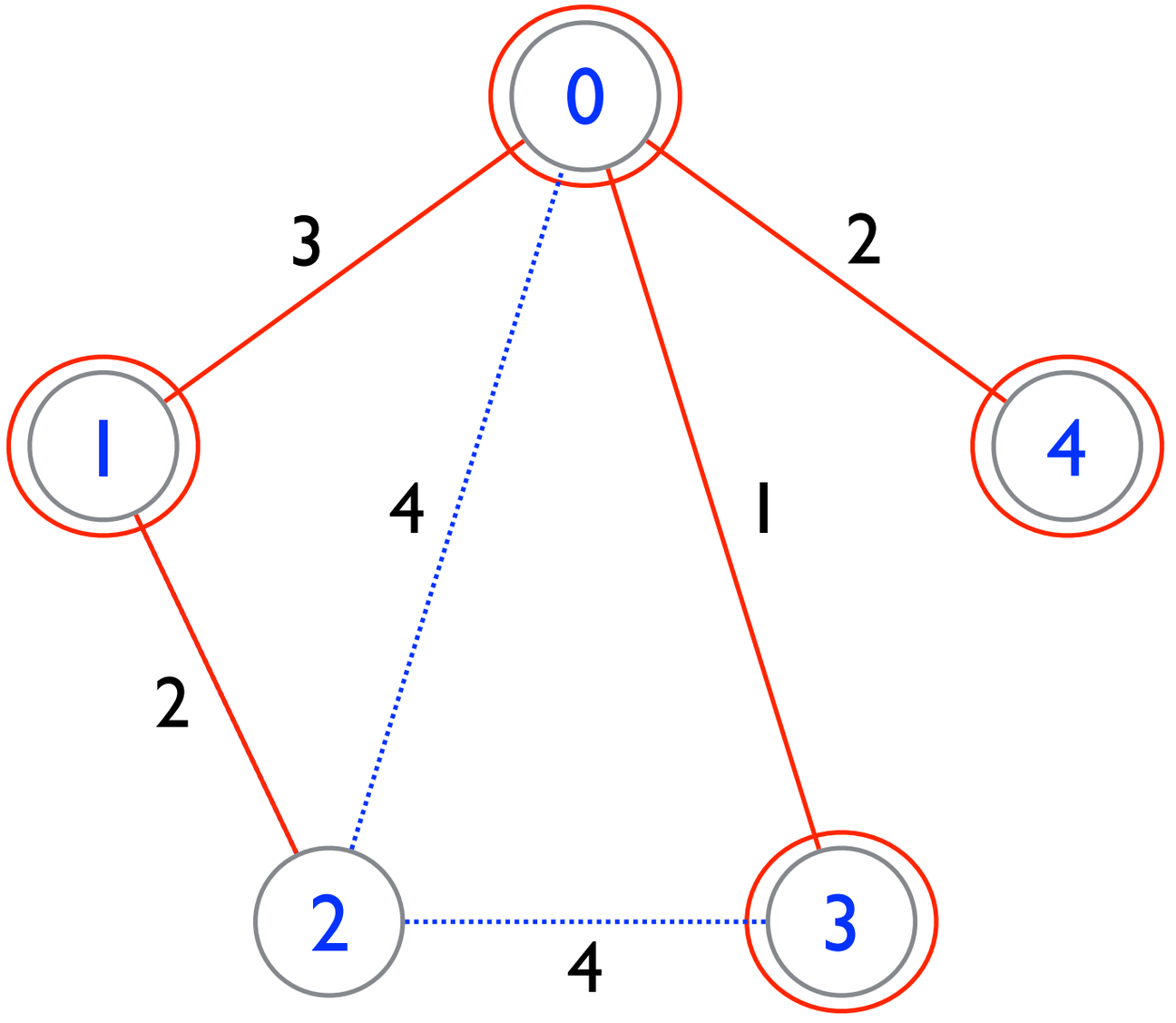

```
0 <- 3
0 <- 4
0 <- 1
```

Prim's Algorithm



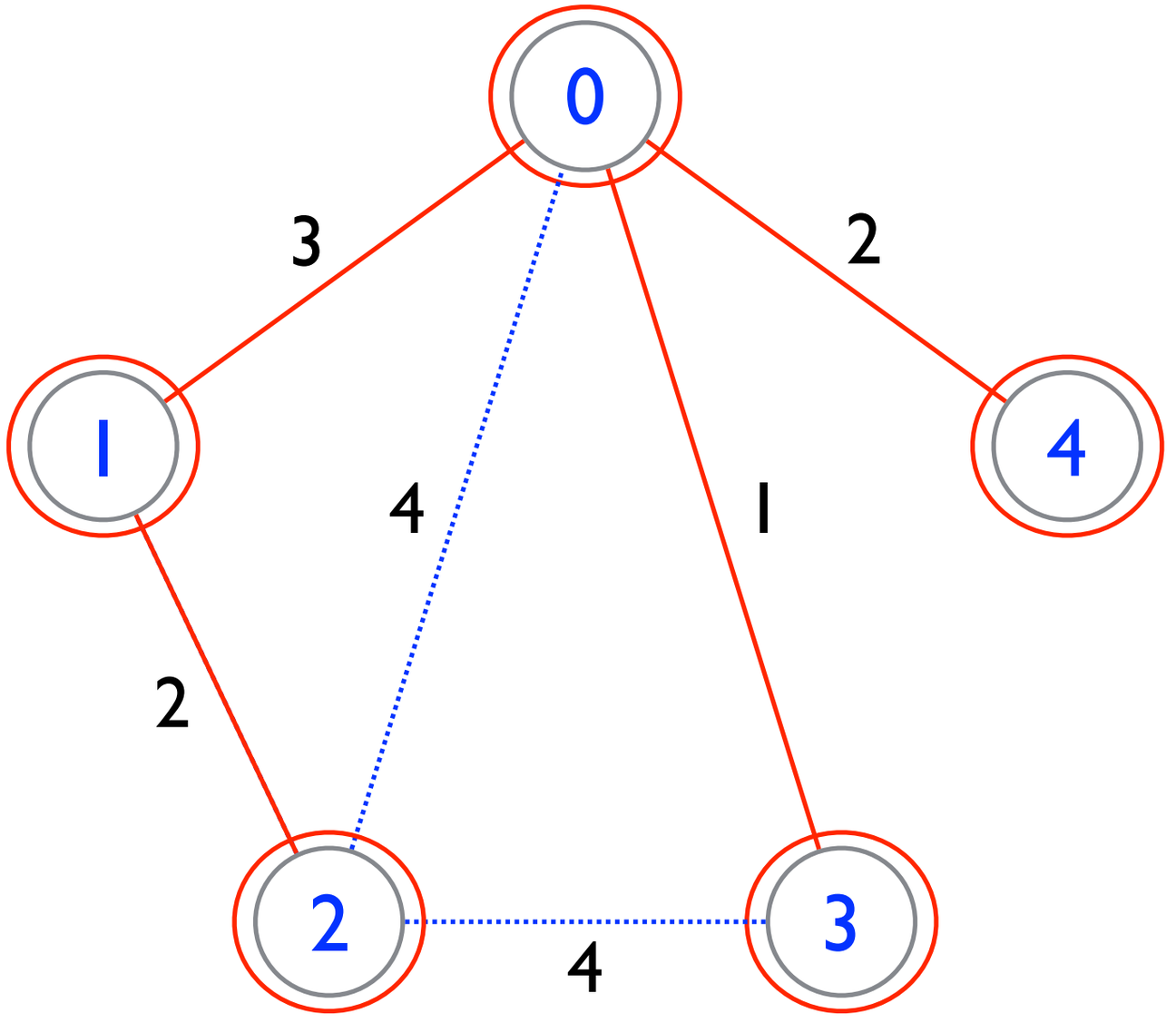

```
0 <- 3
0 <- 4
0 <- 1
```

Prim's Algorithm



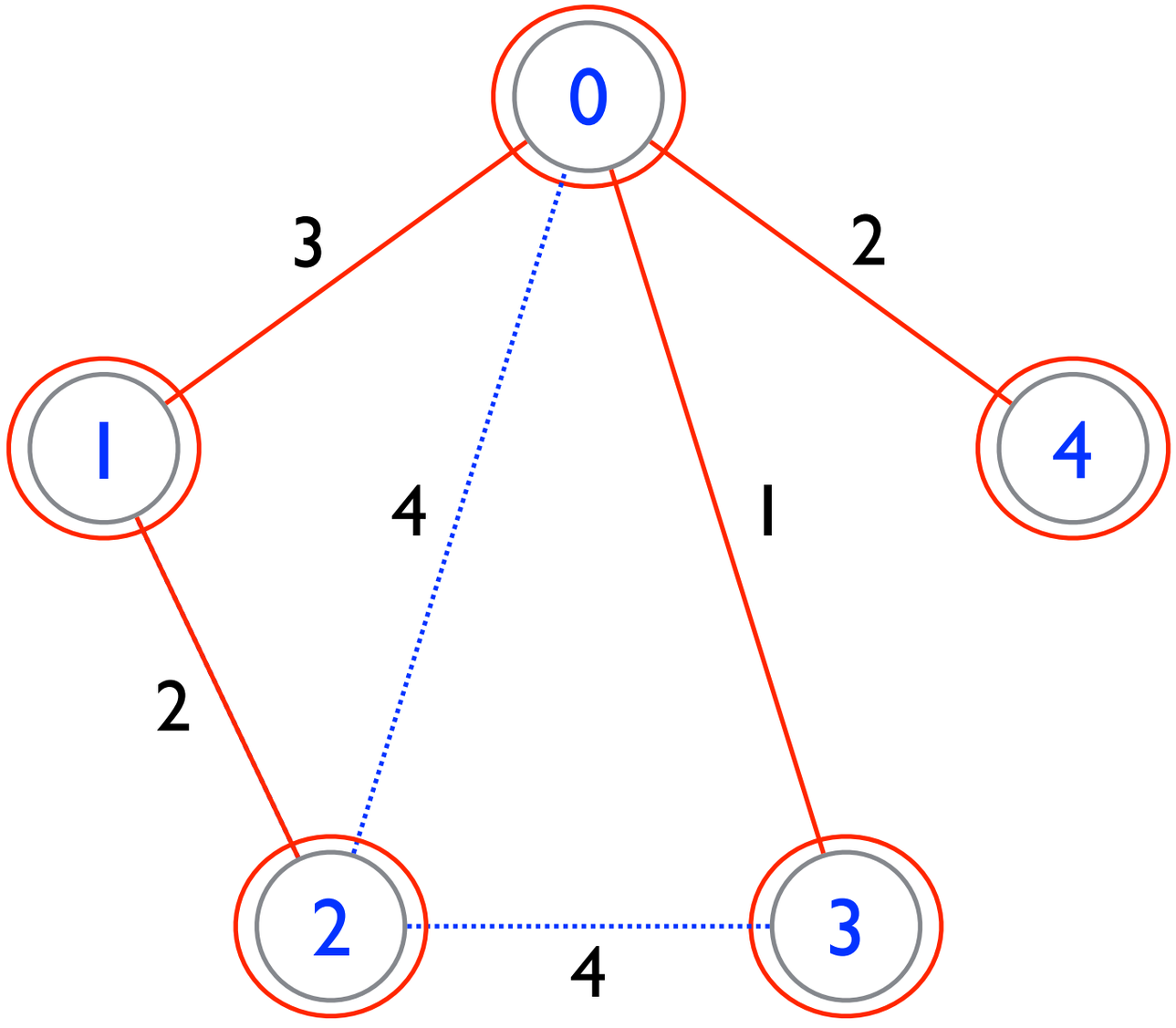

```
0 <- 3
0 <- 4
0 <- 1
1 <- 2
```

Prim's Algorithm




```
0 <- 3
0 <- 4
0 <- 1
1 <- 2
```

Prim's Algorithm




```
0 <- 3
0 <- 4
0 <- 1
1 <- 2
```

A spanning tree is found!

Prim's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph)
{
  PriorityQueue<Edge> queue = new PriorityQueue<>();
  SpanningTree tree = new SpanningTree();
  Set<Integer> set = new HashSet<>();
  Edge edge;

  add(queue, set, graph, 0);

  while (!queue.isEmpty())
  {
    edge = queue.poll();

    if (!set.contains(edge.getSource()))
    {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      add(queue, set, graph, edge.getSource());
    }
  }

  return tree;
}
```

Prim's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph)
{
  PriorityQueue<Edge> queue = new PriorityQueue<>();
  SpanningTree tree = new SpanningTree();
  Set<Integer> set = new HashSet<>();
  Edge edge;

  add(queue, set, graph, 0);

  while (!queue.isEmpty())
  {
    edge = queue.poll();

    if (!set.contains(edge.getSource()))
    {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      add(queue, set, graph, edge.getSource());
    }
  }

  return tree;
}
```


EMORY UNIVERSITY

Prim's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph)
{
  PriorityQueue<Edge> queue = new PriorityQueue<>();
  SpanningTree tree = new SpanningTree();
  Set<Integer> set = new HashSet<>();
  Edge edge;

  add(queue, set, graph, 0);

  while (!queue.isEmpty())
  {
    edge = queue.poll();

    if (!set.contains(edge.getSource()))
    {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      add(queue, set, graph, edge.getSource());
    }
  }

  return tree;
}
```


EMORY UNIVERSITY

Prim's Algorithm

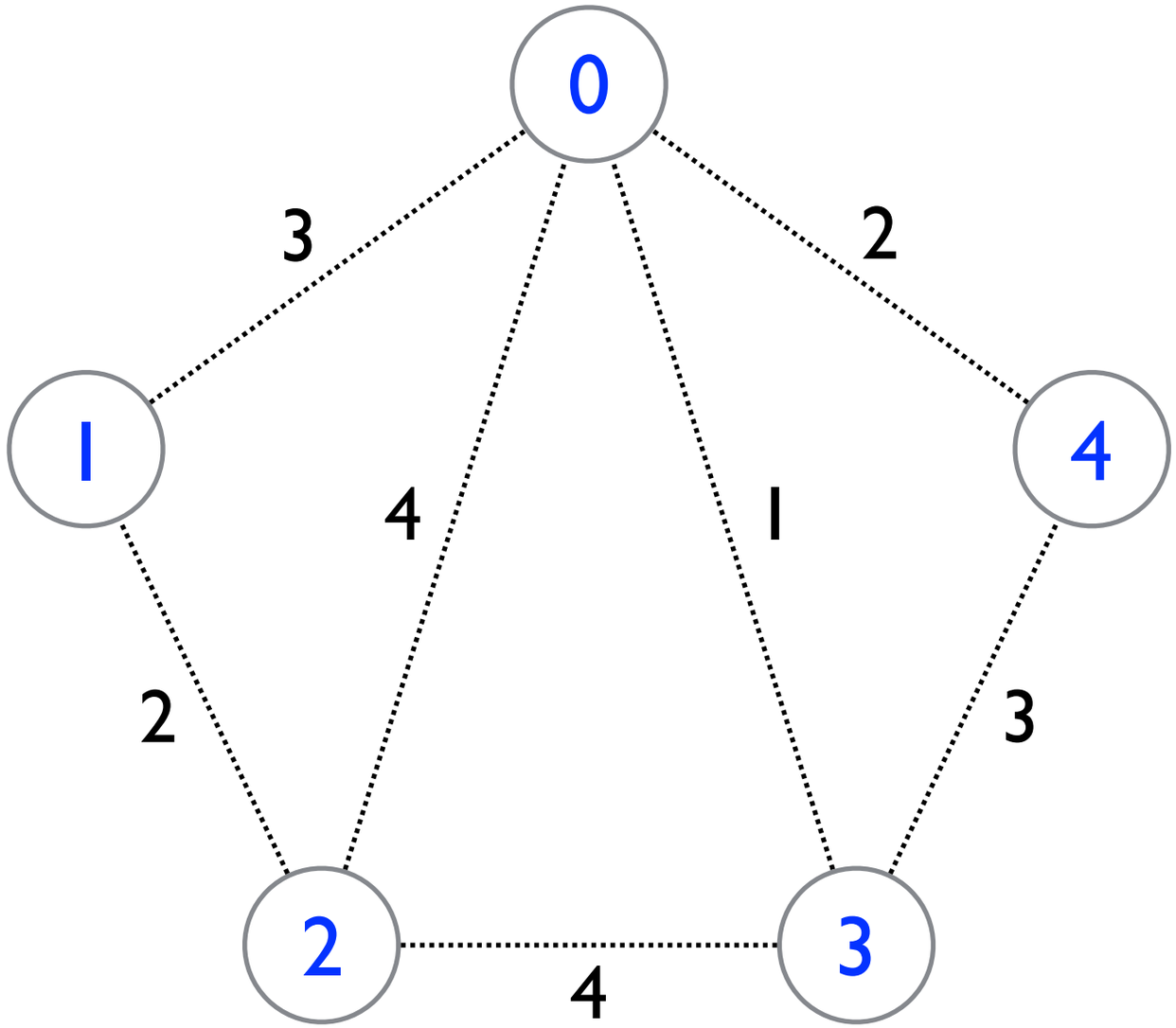
```
public SpanningTree getMinimumSpanningTree(Graph graph)
{
  PriorityQueue<Edge> queue = new PriorityQueue<>();
  SpanningTree tree = new SpanningTree();
  Set<Integer> set = new HashSet<>();
  Edge edge;

  add(queue, set, graph, 0);

  while (!queue.isEmpty())
  {
    edge = queue.poll();

    if (!set.contains(edge.getSource()))
    {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      add(queue, set, graph, edge.getSource());
    }
  }

  return tree;
}
```

Prim's Algorithm

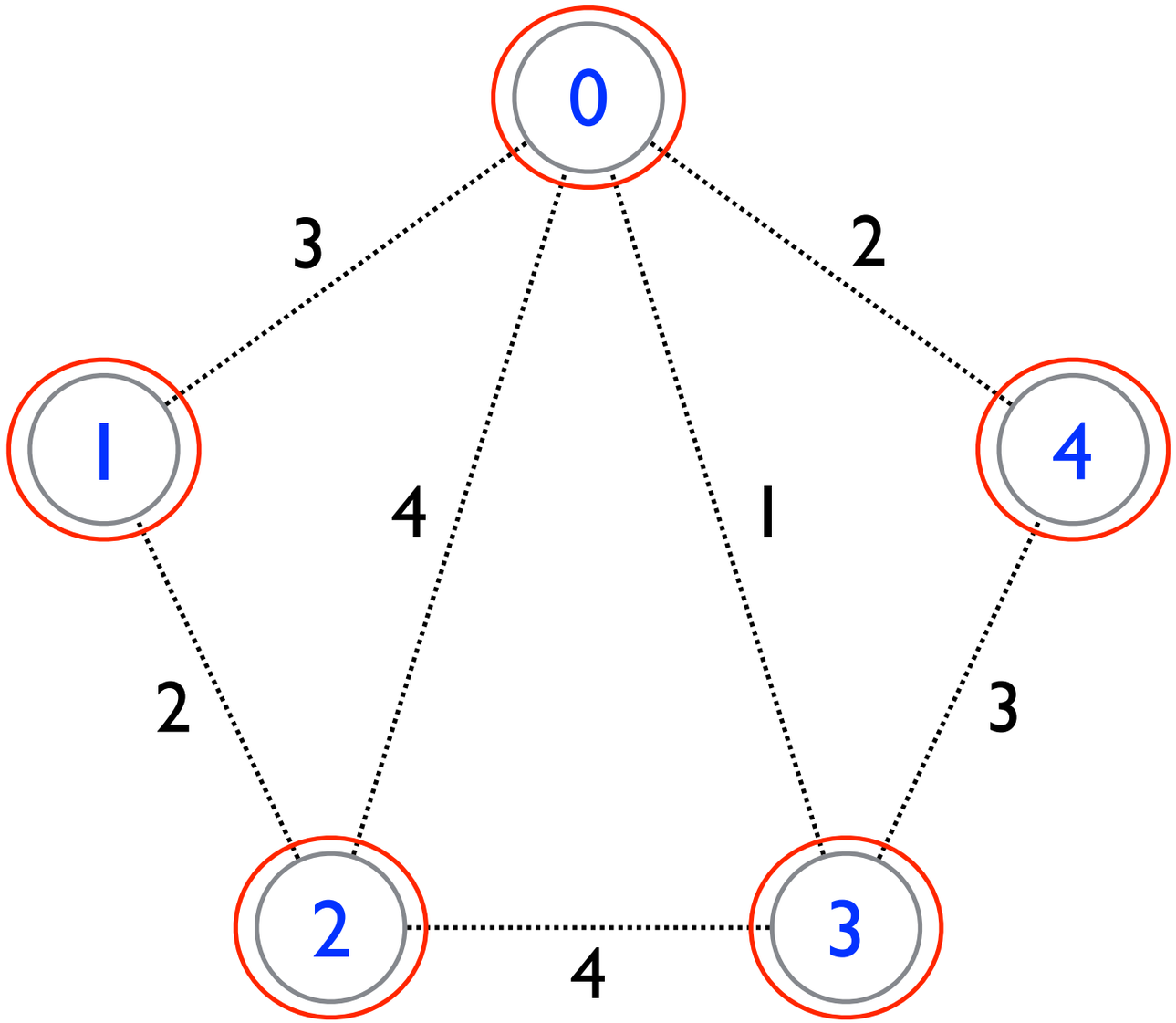
```
public SpanningTree getMinimumSpanningTree(Graph graph)
{
  PriorityQueue<Edge> queue = new PriorityQueue<>();
  SpanningTree tree = new SpanningTree();
  Set<Integer> set = new HashSet<>();
  Edge edge;

  add(queue, set, graph, 0);

  while (!queue.isEmpty())
  {
    edge = queue.poll();

    if (!set.contains(edge.getSource()))
    {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      add(queue, set, graph, edge.getSource());
    }
  }

  return tree;
}
```

Complexity?

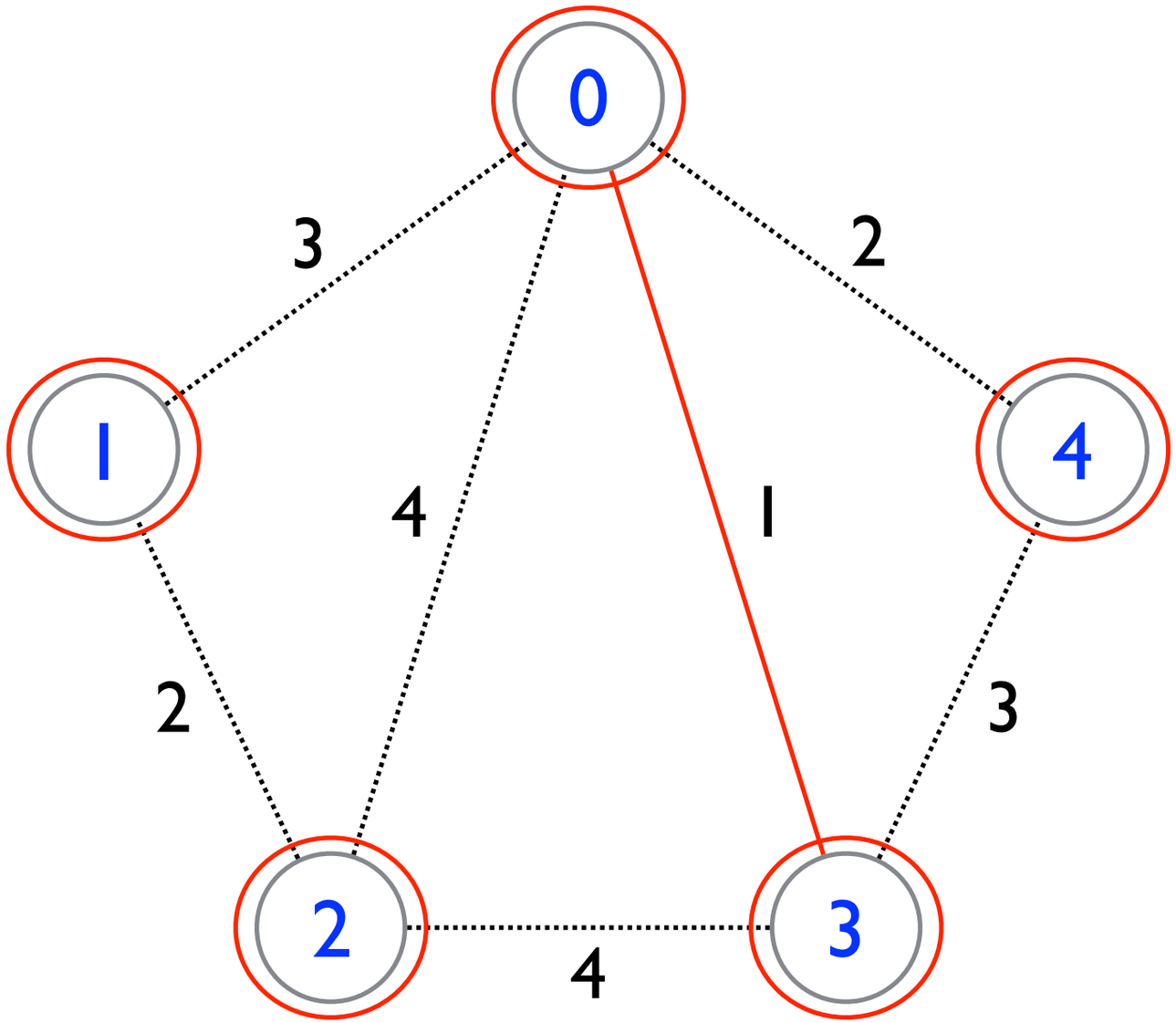
Kruskal's Algorithm



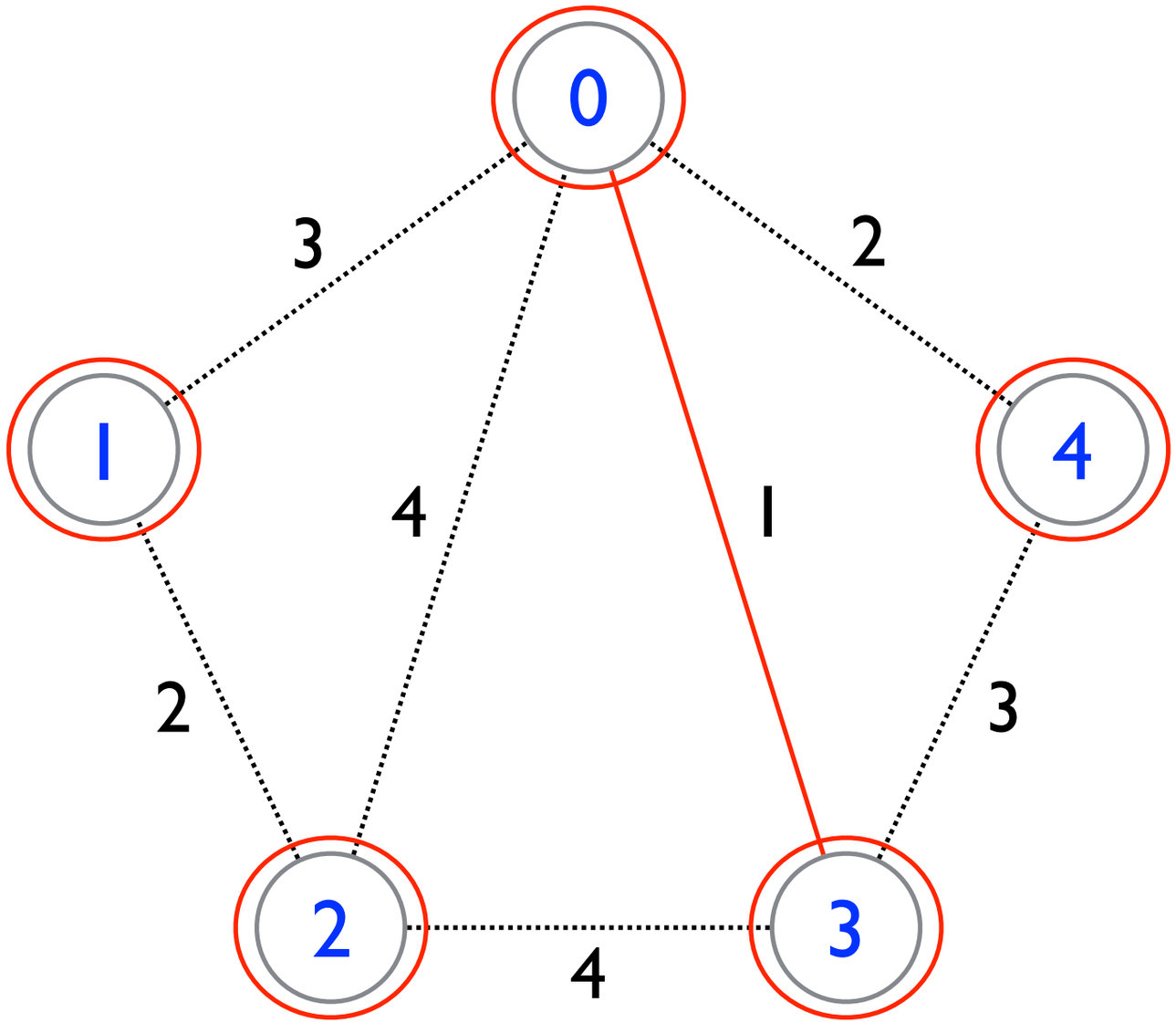
Kruskal's Algorithm



Kruskal's Algorithm

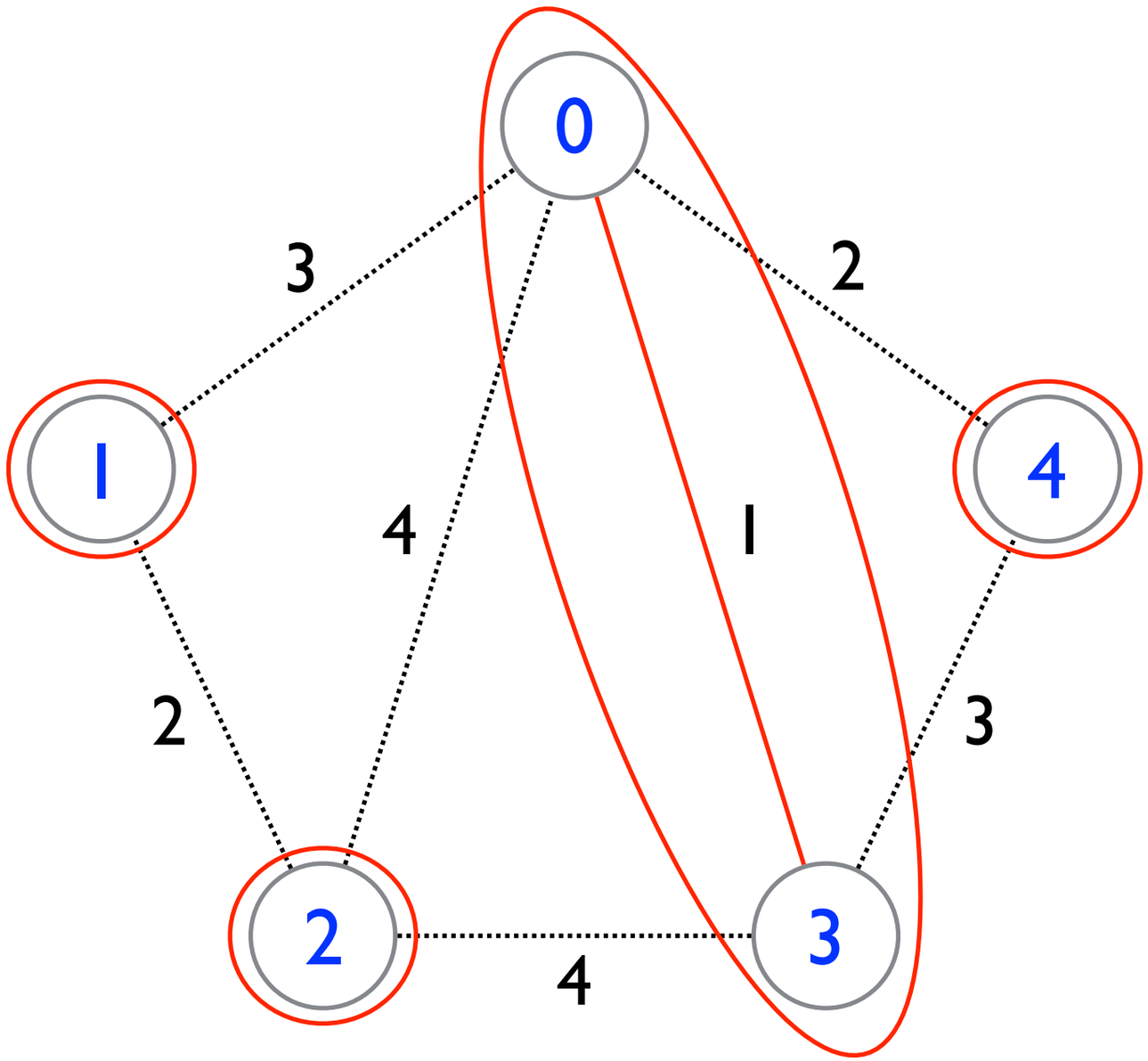


Kruskal's Algorithm



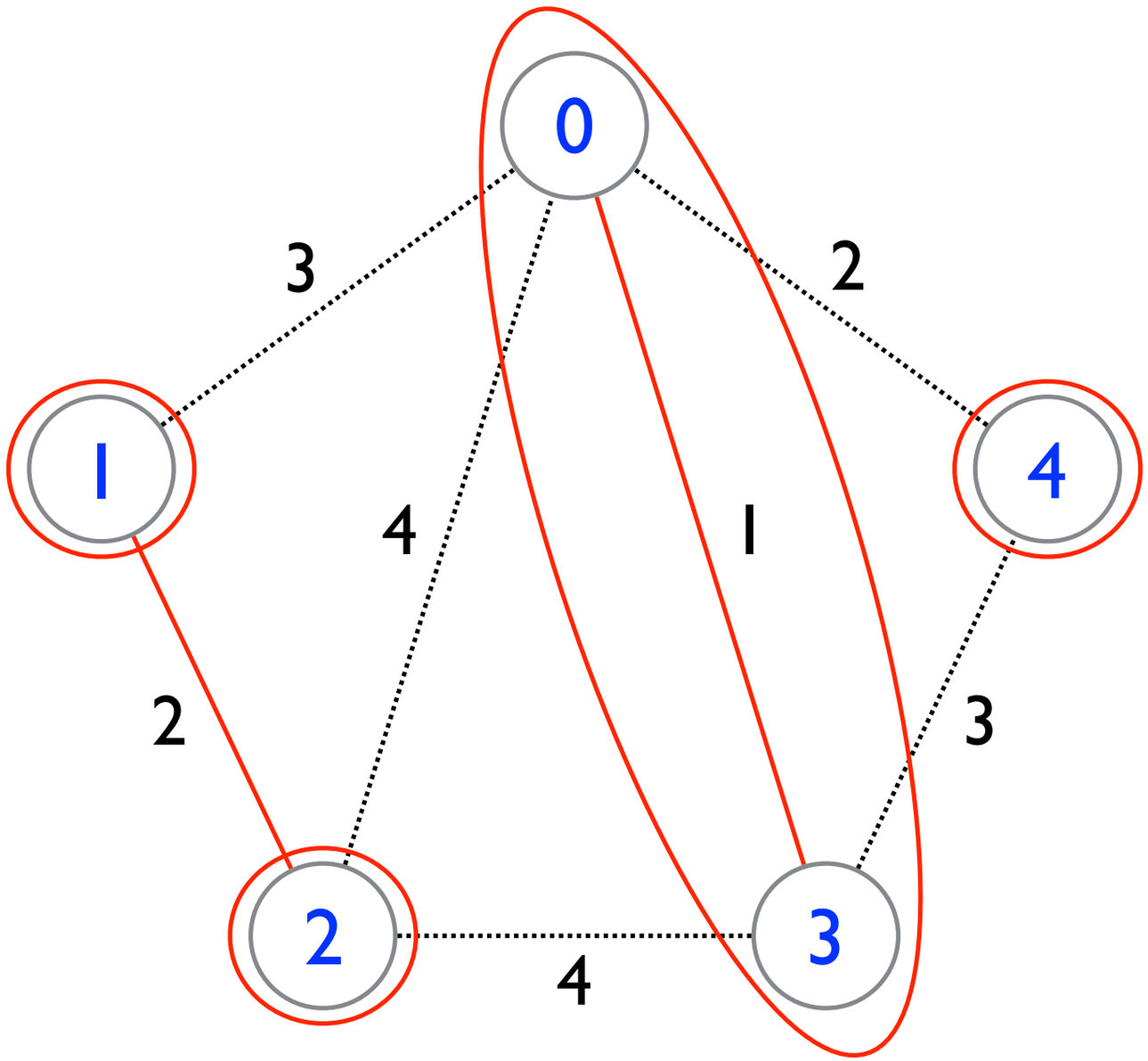

```
0 <- 3
```

Kruskal's Algorithm



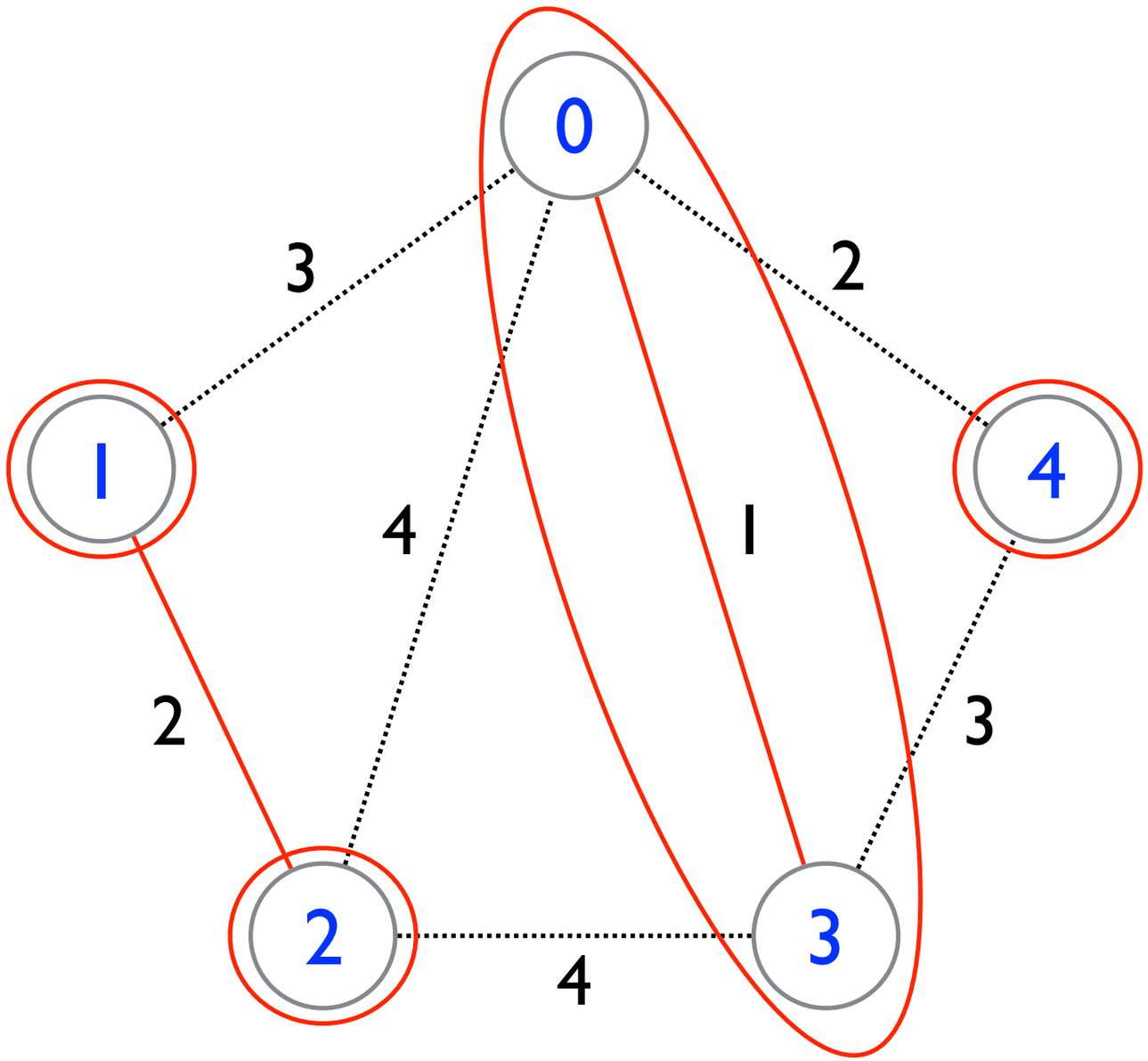

```
0 <- 3
```

Kruskal's Algorithm



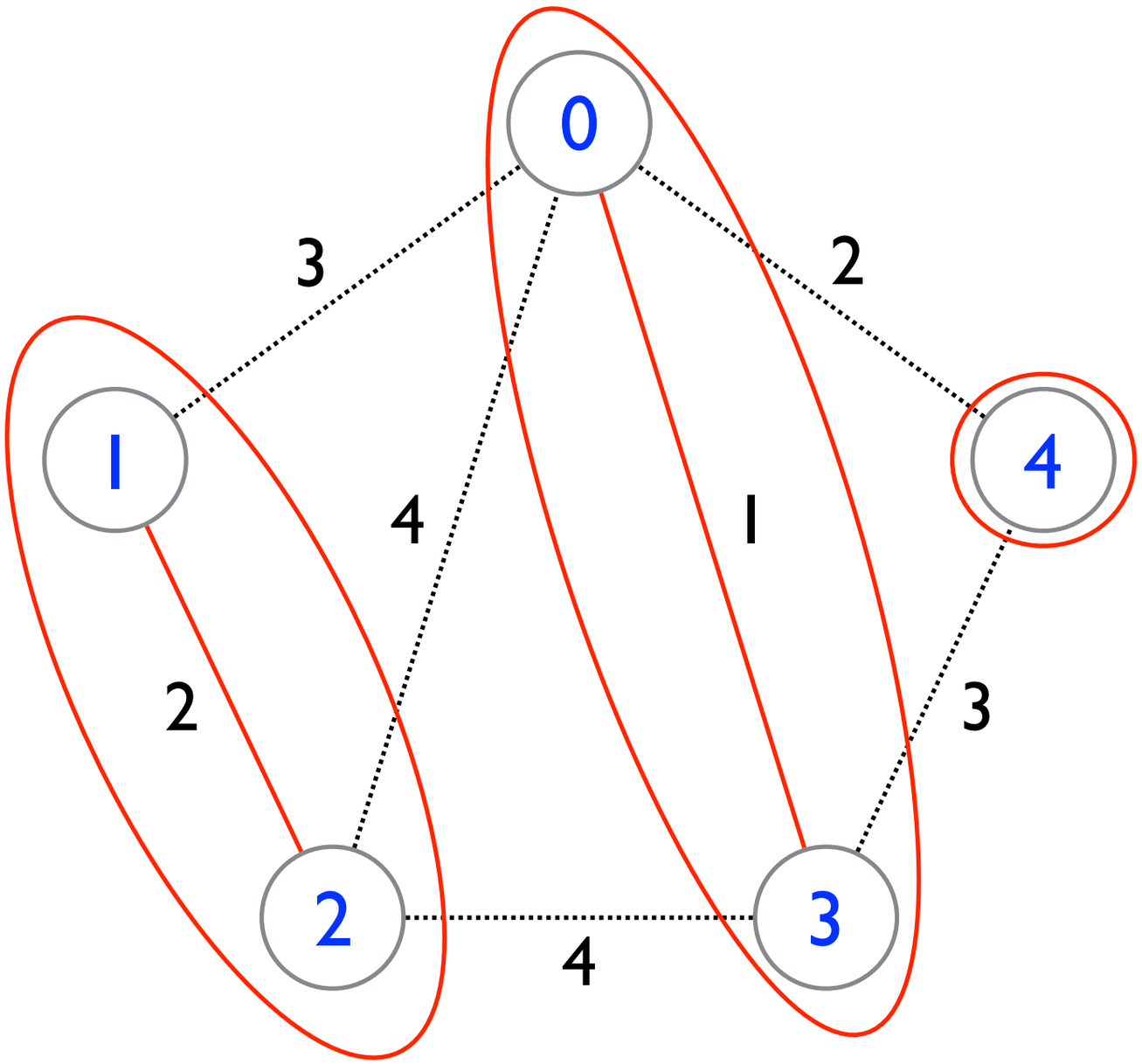

```
0 <- 3
```

Kruskal's Algorithm



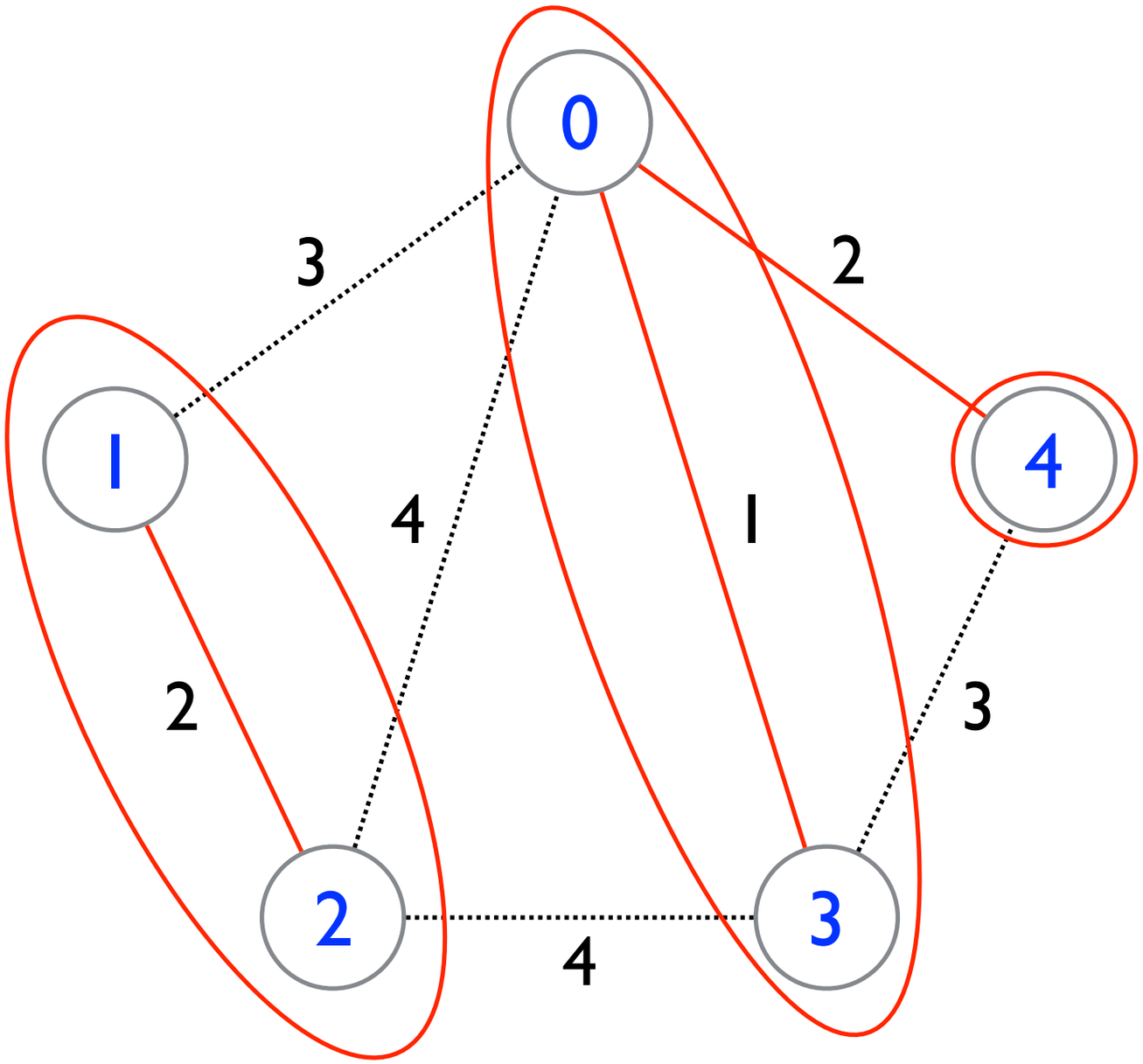

```
0 <- 3
1 <- 2
```

Kruskal's Algorithm



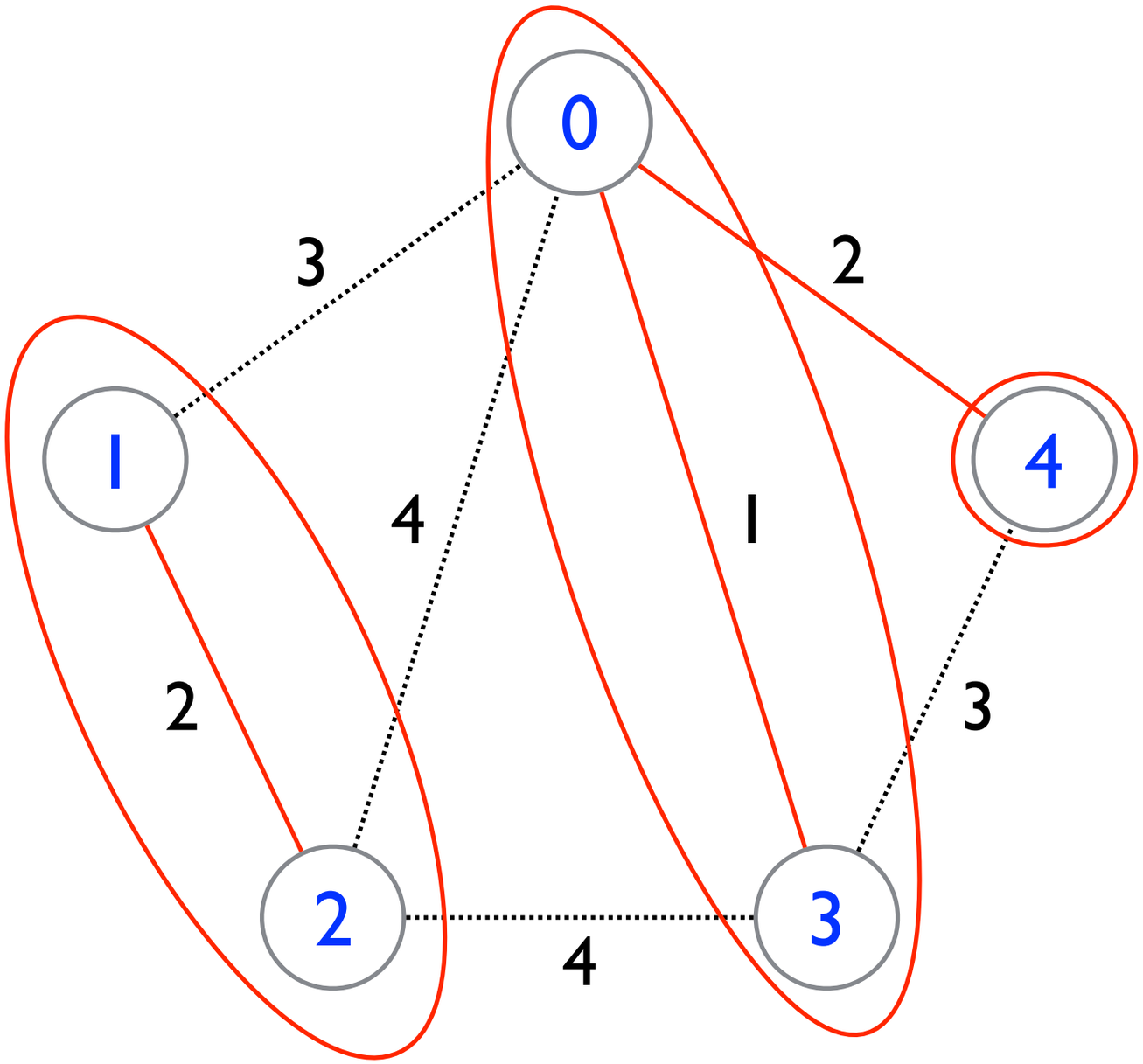

```
0 <- 3
1 <- 2
```

Kruskal's Algorithm



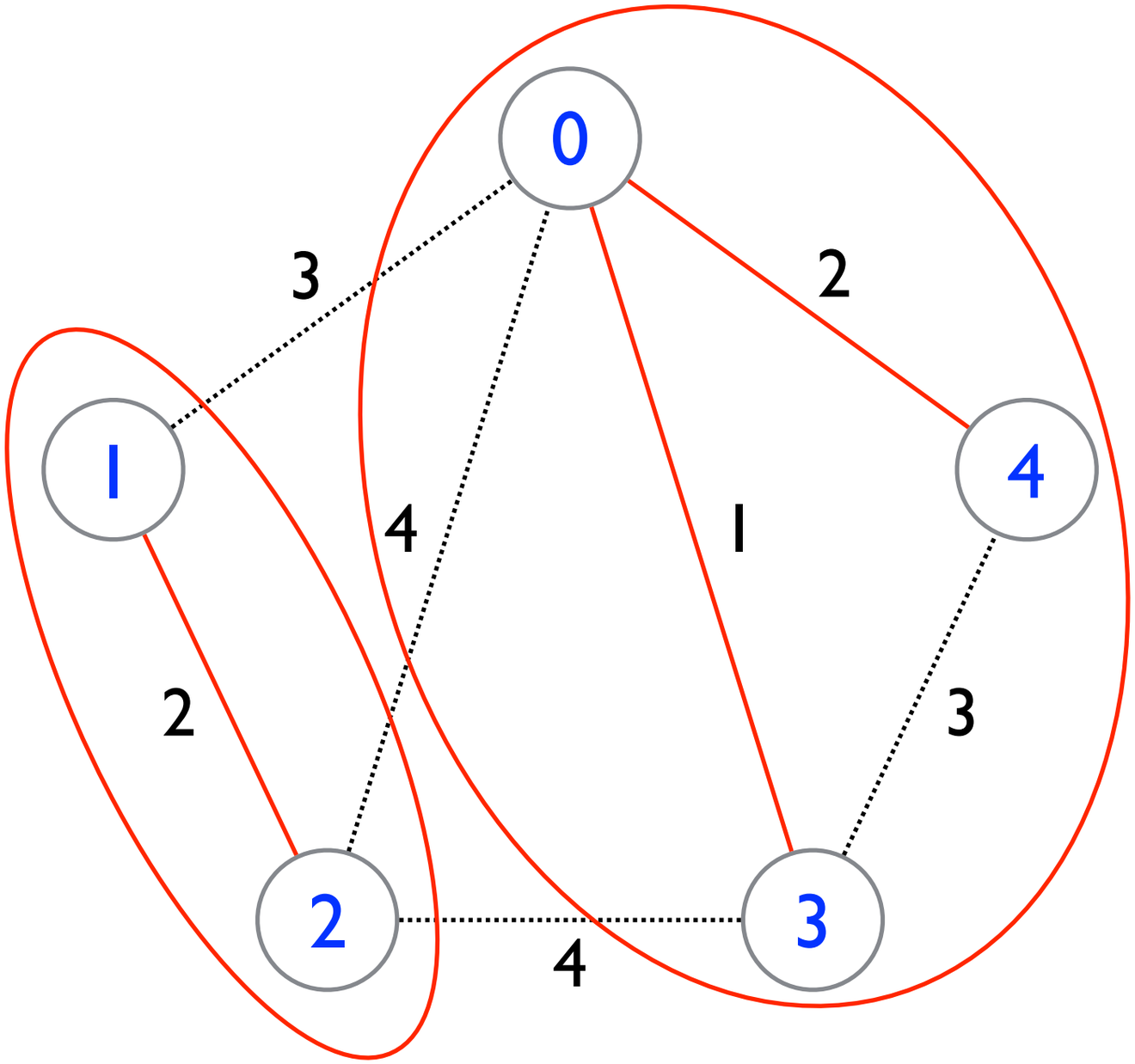

```
0 <- 3
1 <- 2
```

Kruskal's Algorithm



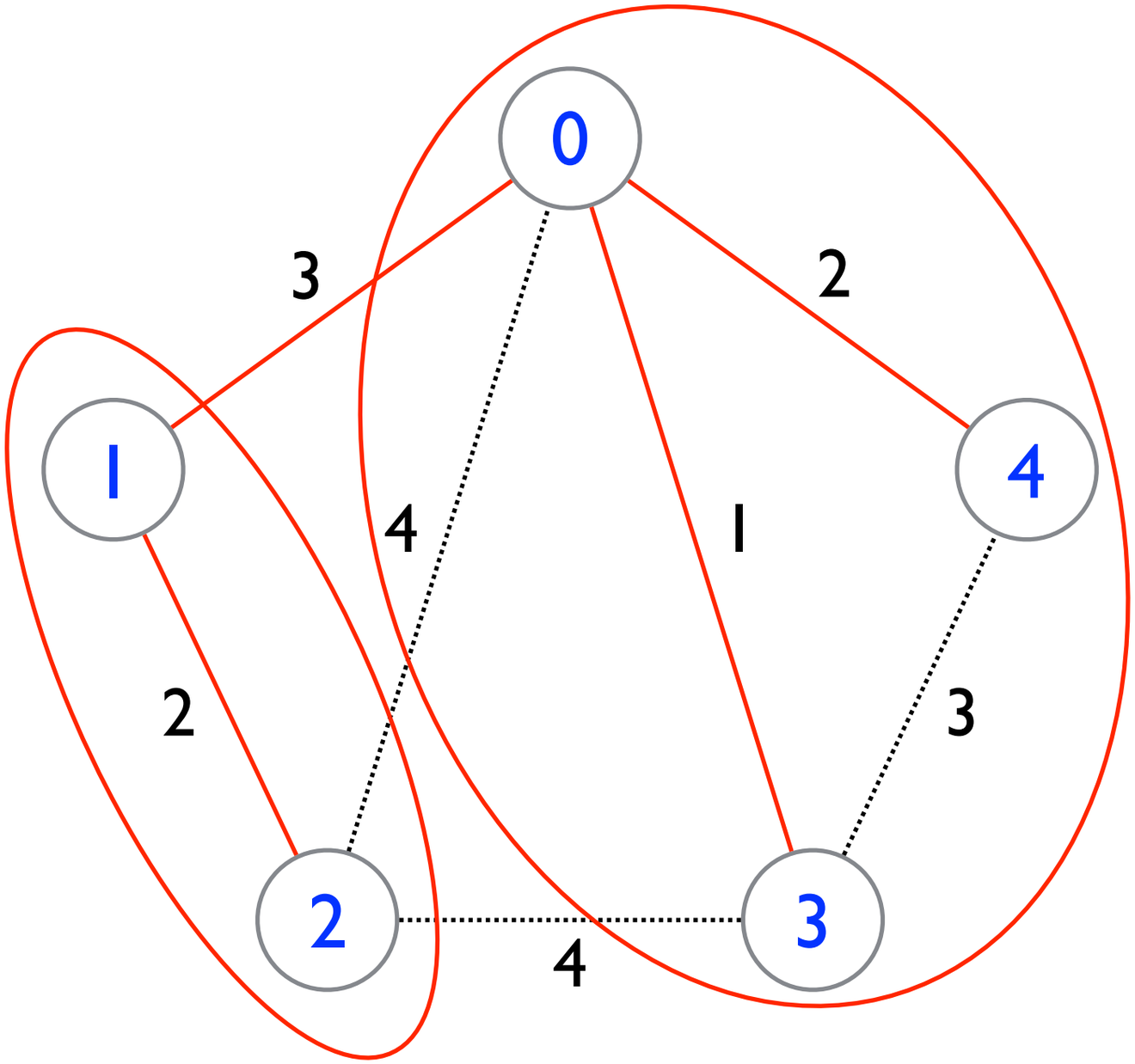

```
0 <- 3
1 <- 2
0 <- 4
```

Kruskal's Algorithm



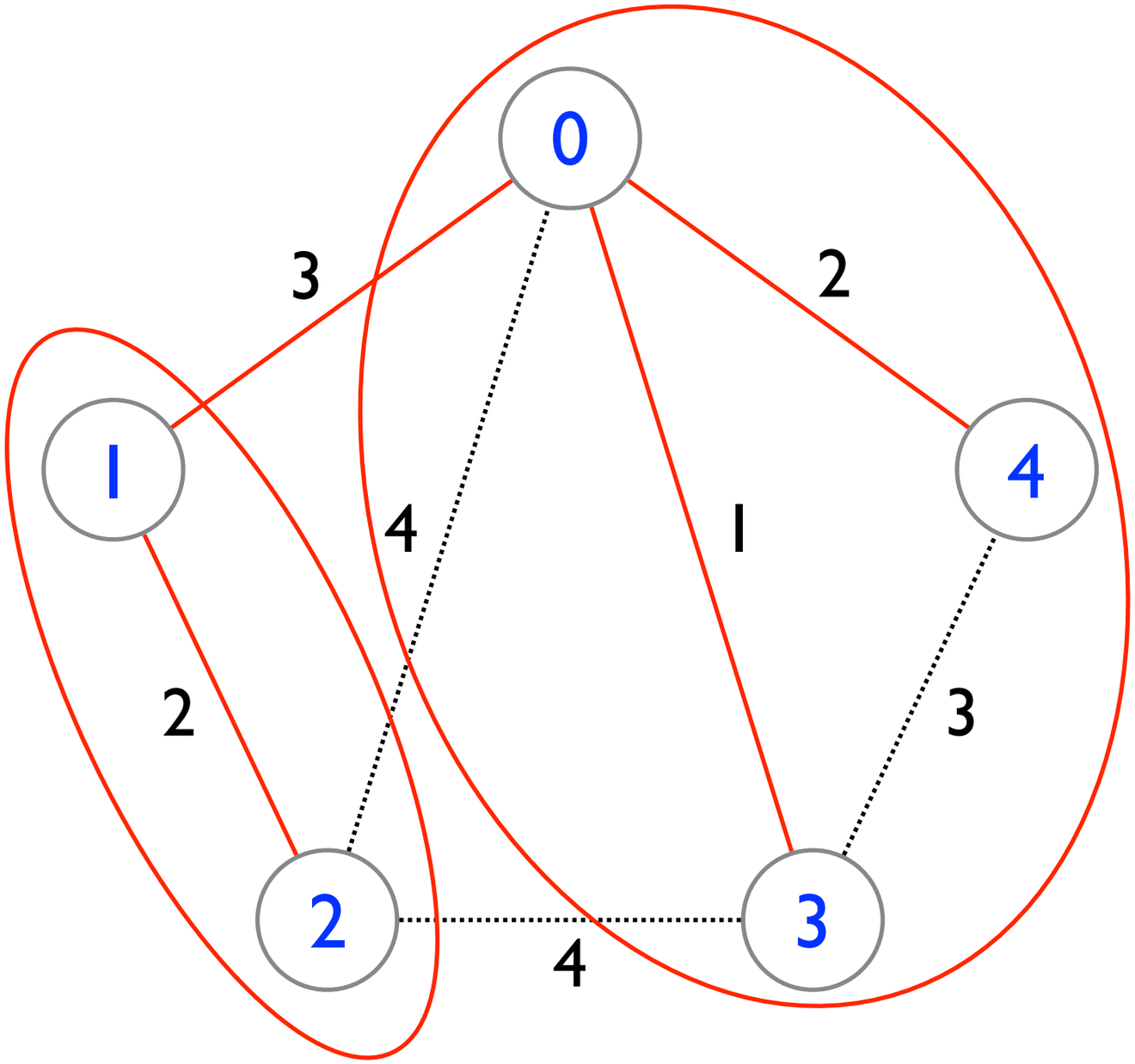

```
0 <- 3
1 <- 2
0 <- 4
```

Kruskal's Algorithm



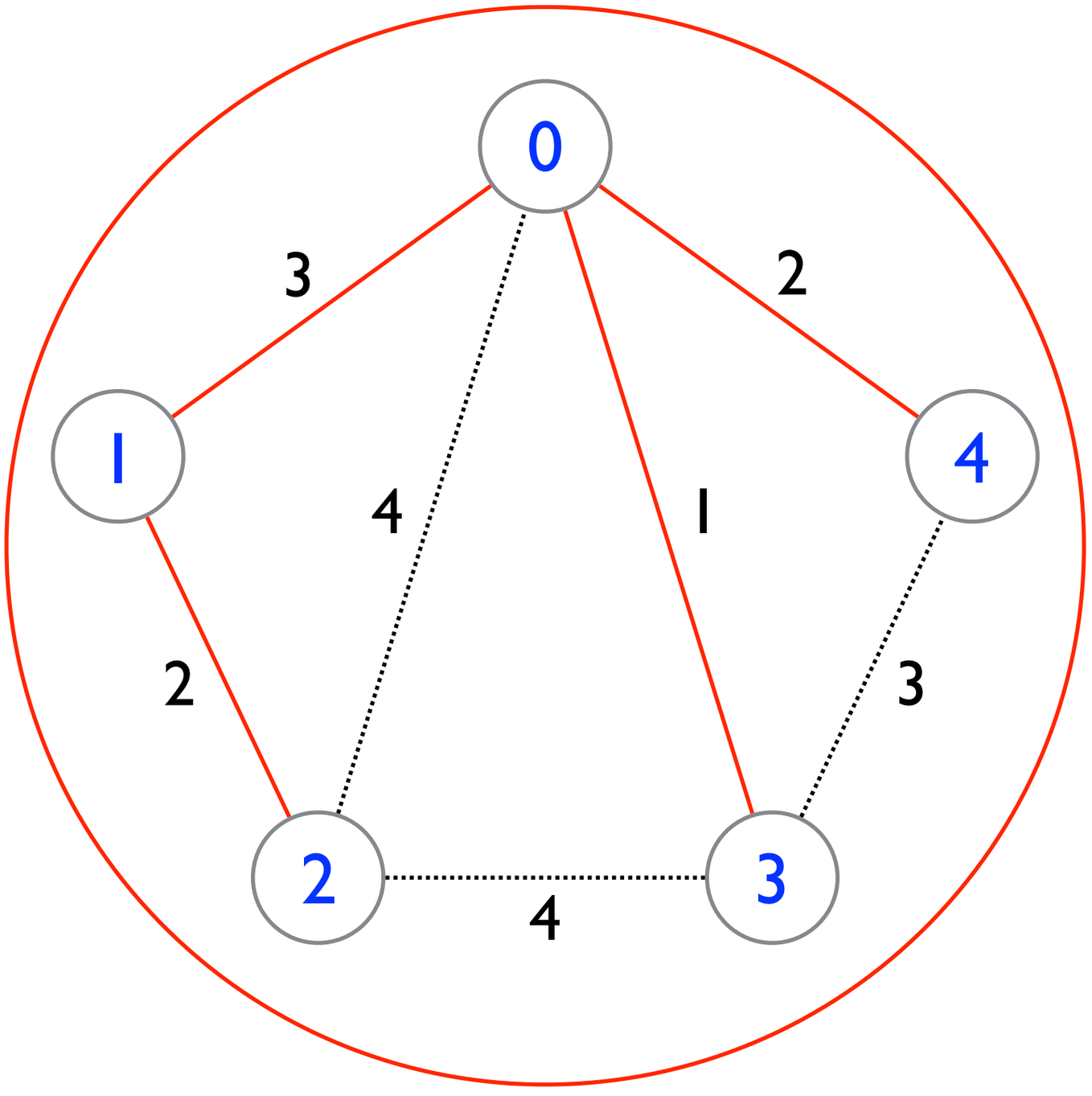

```
0 <- 3
1 <- 2
0 <- 4
```

Kruskal's Algorithm




```
0 <- 3
1 <- 2
0 <- 4
0 <- 1
```

Kruskal's Algorithm

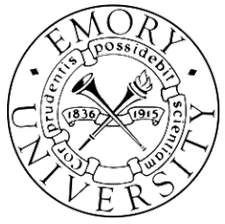



```
0 <- 3
1 <- 2
0 <- 4
0 <- 1
```

Kruskal's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph) {
  PriorityQueue<Edge> queue = new PriorityQueue<>(graph.getAllEdges());
  DisjointSet forest = new DisjointSet(graph.size());
  SpanningTree tree = new SpanningTree();
  Edge edge;

  while (!queue.isEmpty()) {
    edge = queue.poll();
    if (!forest.inSameSet(edge.getTarget(), edge.getSource())) {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      forest.union(edge.getTarget(), edge.getSource());
    }
  }

  return tree;
}
```

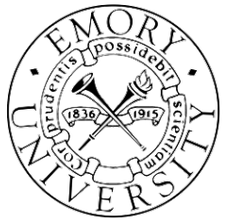
EMORY
UNIVERSITY



Kruskal's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph) {
  PriorityQueue<Edge> queue = new PriorityQueue<>(graph.getAllEdges());
  DisjointSet forest = new DisjointSet(graph.size());
  SpanningTree tree = new SpanningTree();
  Edge edge;

  while (!queue.isEmpty()) {
    edge = queue.poll();
    if (!forest.inSameSet(edge.getTarget(), edge.getSource())) {
      tree.addEdge(edge);
      if (tree.size()+1 == graph.size()) break;
      forest.union(edge.getTarget(), edge.getSource());
    }
  }

  return tree;
}
```

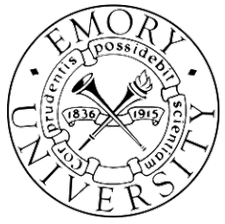
EMORY
UNIVERSITY



Kruskal's Algorithm

```
public SpanningTree getMinimumSpanningTree(Graph graph) {
  PriorityQueue<Edge> queue = new PriorityQueue<>(graph.getAllEdges());
  DisjointSet forest = new DisjointSet(graph.size());
  SpanningTree tree = new SpanningTree();
  Edge edge;

  while (!queue.isEmpty()) {
     edge = queue.poll();
     if (!forest.inSameSet(edge.getTarget(), edge.getSource())) {
       tree.addEdge(edge);
       if (tree.size()+1 == graph.size()) break;
       forest.union(edge.getTarget(), edge.getSource());
     }
   }

   return tree;
}
```

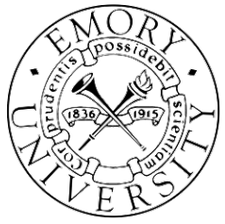
EMORY
UNIVERSITY



Kruskal's Algorithm

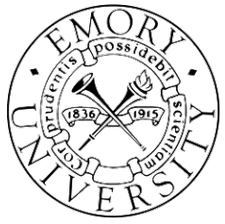
```
public SpanningTree getMinimumSpanningTree(Graph graph) {
  PriorityQueue<Edge> queue = new PriorityQueue<>(graph.getAllEdges());
  DisjointSet forest = new DisjointSet(graph.size());
  SpanningTree tree = new SpanningTree();
  Edge edge;

  while (!queue.isEmpty()) {
     edge = queue.poll();
     if (!forest.inSameSet(edge.getTarget(), edge.getSource())) {
       tree.addEdge(edge);
       if (tree.size()+1 == graph.size()) break;
       forest.union(edge.getTarget(), edge.getSource());
     }
  }

  return tree;
}
```

EMORY
UNIVERSITY



Chu–Liu-Edmonds' Algorithm

EMORY
UNIVERSITY



Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm

1. Initially, every vertex is considered a tree.

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm
  1. Initially, every vertex is considered a tree.
  2. For each tree, keep 1 incoming edge with the minimum weight.

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm
  1. Initially, every vertex is considered a tree.
  2. For each tree, keep 1 incoming edge with the minimum weight.
  3. If there is no cycle, go to #5.

Chu–Liu-Edmonds' Algorithm

- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm

1. Initially, every vertex is considered a tree.
2. For each tree, keep 1 incoming edge with the minimum weight.
3. If there is no cycle, go to #5.
4. If there is a cycle, merge trees with the cycle into one and update scores for all incoming edges to this tree, and goto #2.

Chu–Liu-Edmonds' Algorithm

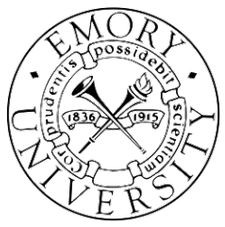
- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm
  1. Initially, every vertex is considered a tree.
  2. For each tree, keep 1 incoming edge with the minimum weight.
  3. If there is no cycle, go to #5.
  4. If there is a cycle, merge trees with the cycle into one and update scores for all incoming edges to this tree, and goto #2.
     - For each vertex in the tree, add the weight of its outgoing edge chain to its incoming edges not in the tree.

Chu–Liu-Edmonds' Algorithm

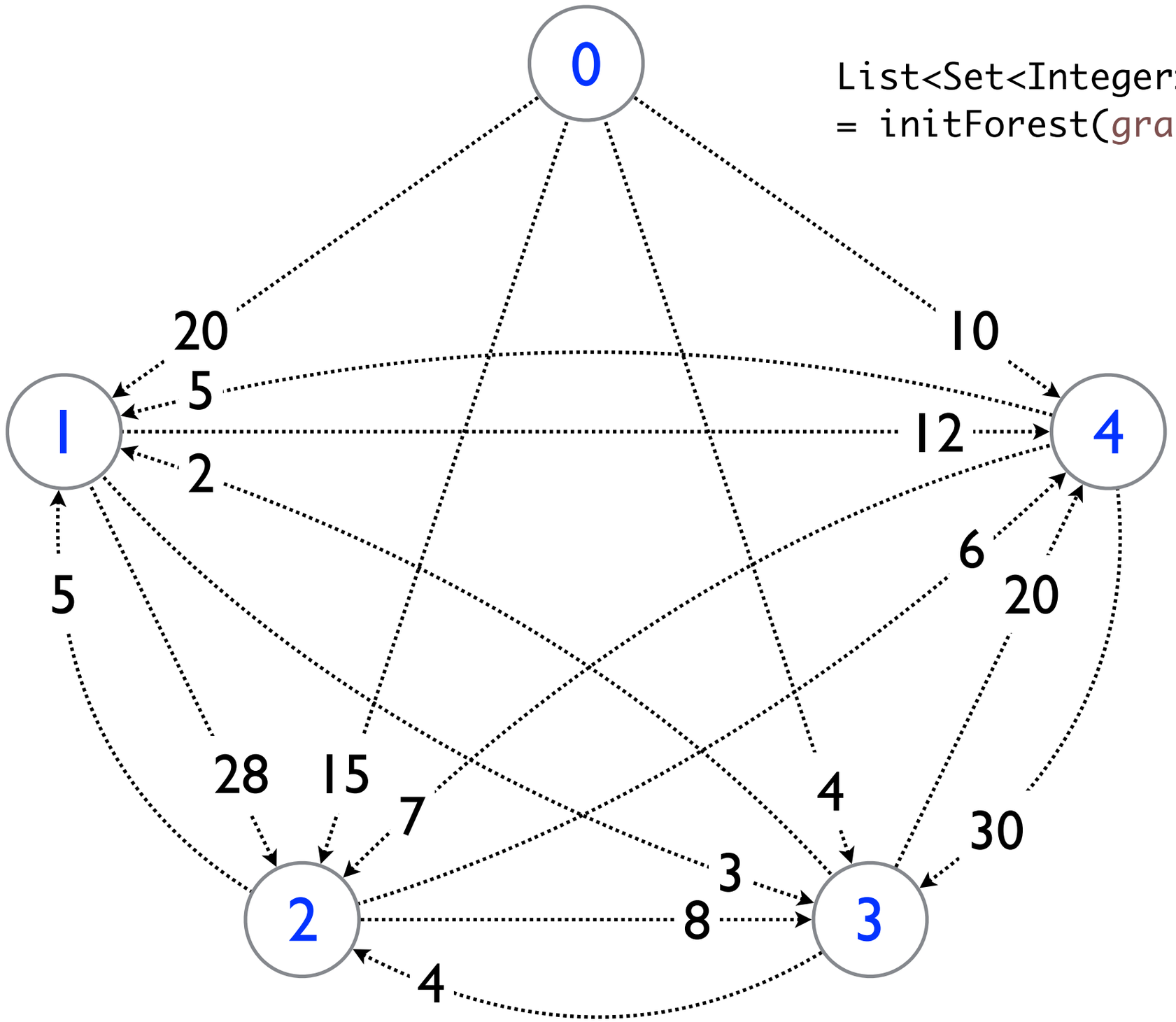
- Chu–Liu-Edmonds' algorithm
  - Finding minimum spanning trees in directed graphs.
- Algorithm

1. Initially, every vertex is considered a tree.
2. For each tree, keep 1 incoming edge with the minimum weight.
3. If there is no cycle, go to #5.
4. If there is a cycle, merge trees with the cycle into one and update scores for all incoming edges to this tree, and goto #2.
   - For each vertex in the tree, add the weight of its outgoing edge chain to its incoming edges not in the tree.
5. Break all cycles by removing edges that cause multiple parents.

EMORY
UNIVERSITY



Chu–Liu-Edmonds' Algorithm

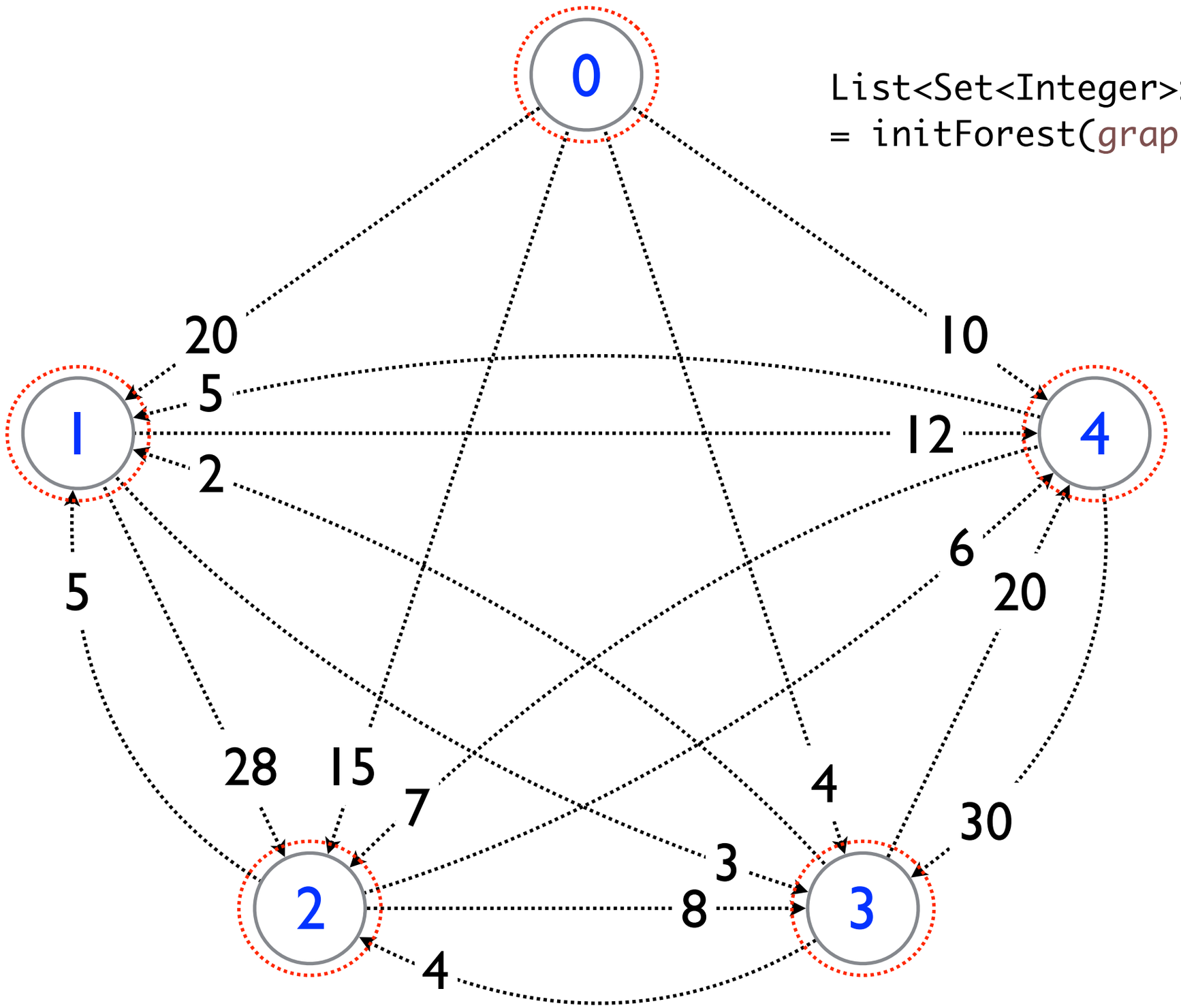



```
List<Set<Integer>> forest
= initForest(graph.size());
```

15

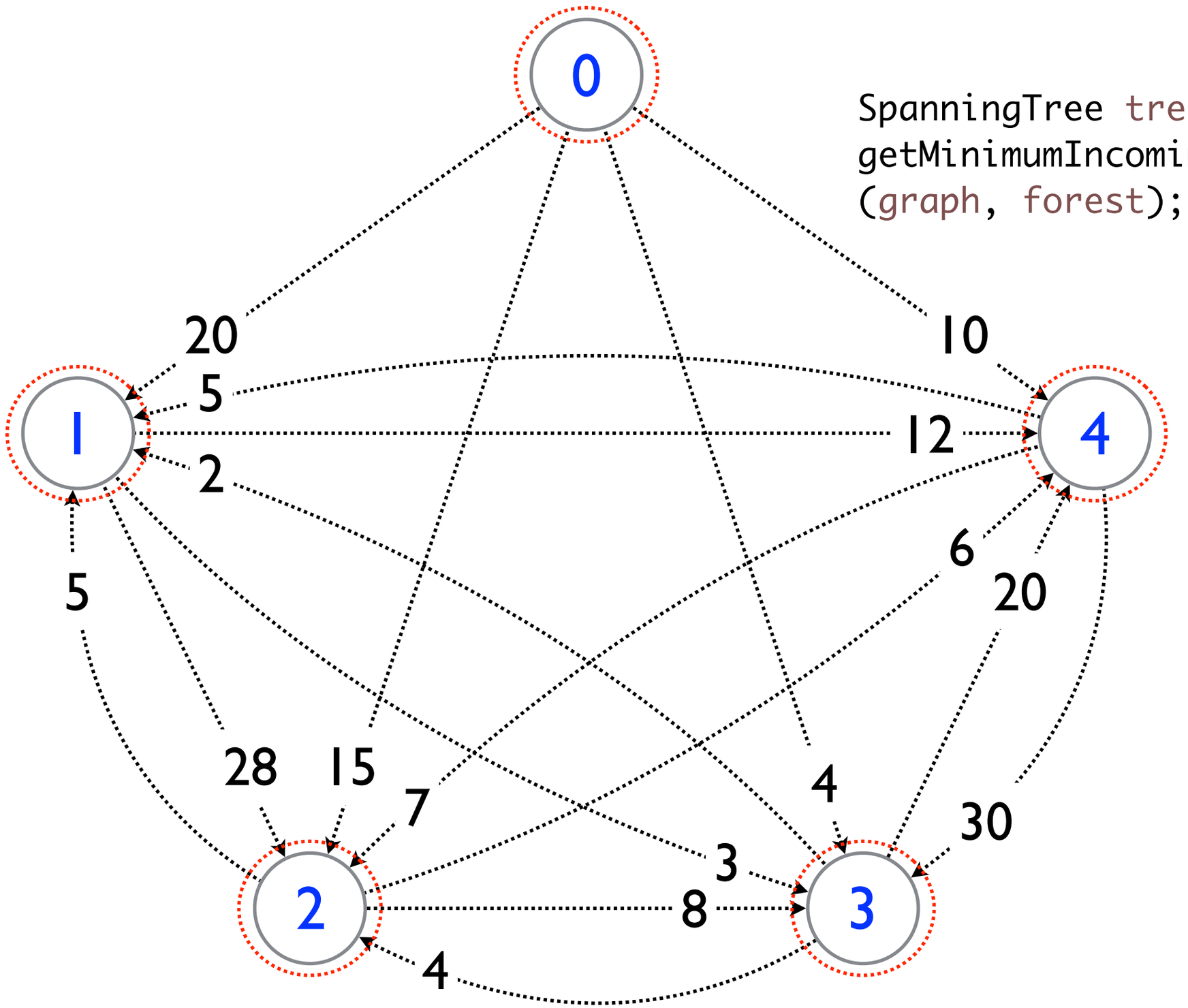
EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm




```
List<Set<Integer>> forest
= initForest(graph.size());
```

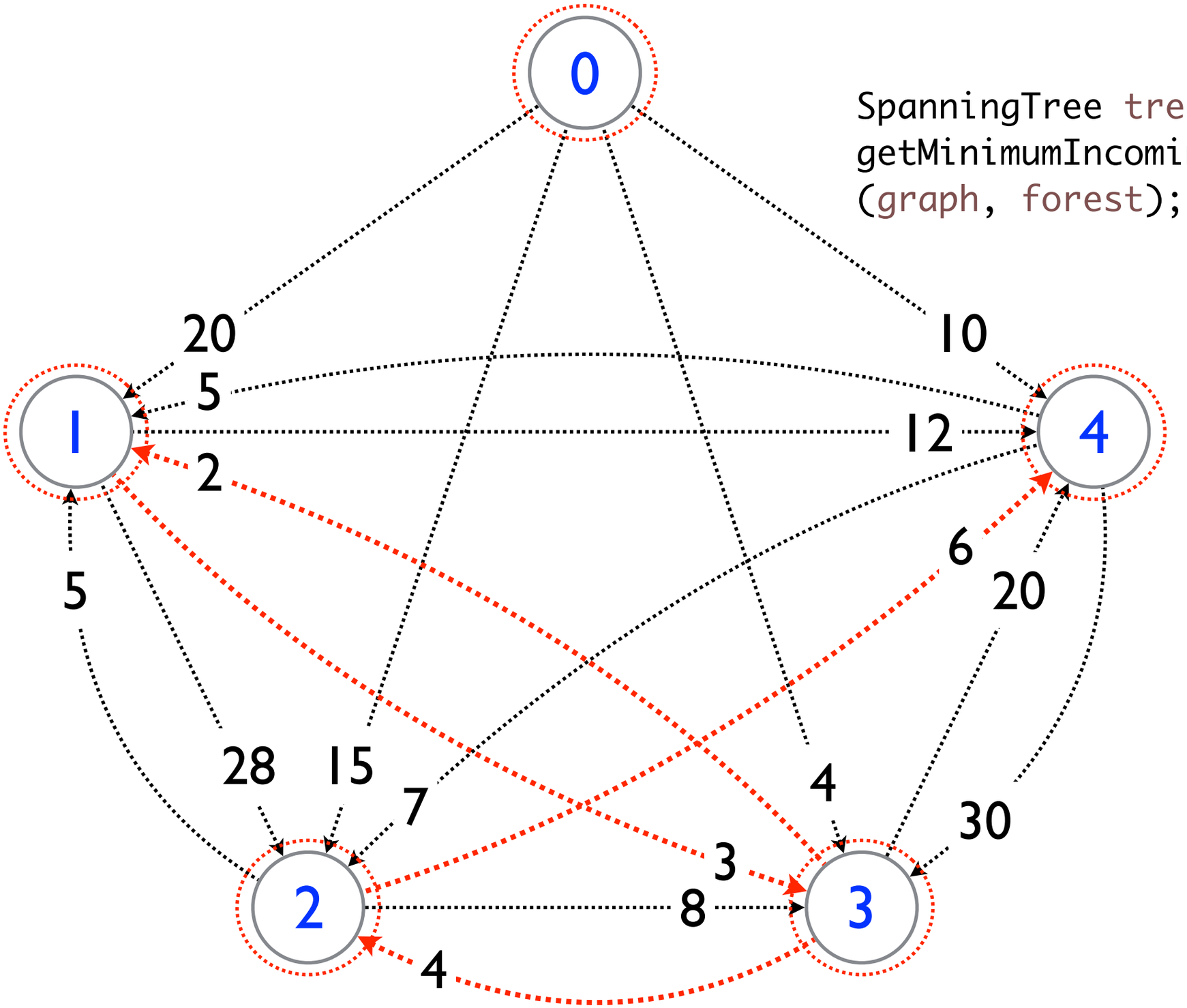
Chu–Liu-Edmonds' Algorithm




```
SpanningTree tree =
getMinimumIncomingEdges
(graph, forest);
```


EMORY UNIVERSITY

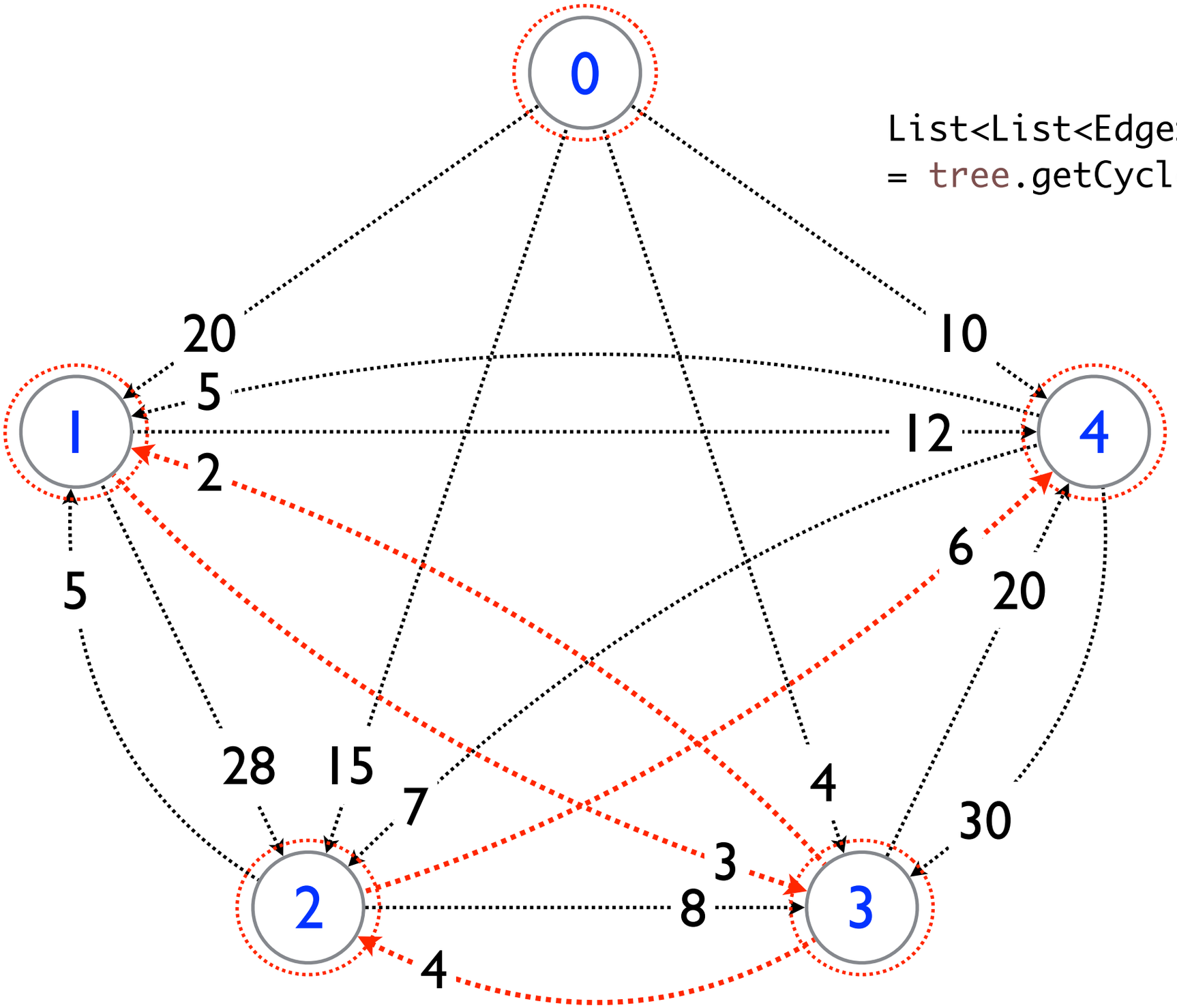
Chu–Liu-Edmonds' Algorithm




```
SpanningTree tree =
getMinimumIncomingEdges
(graph, forest);
```


EMORY UNIVERSITY

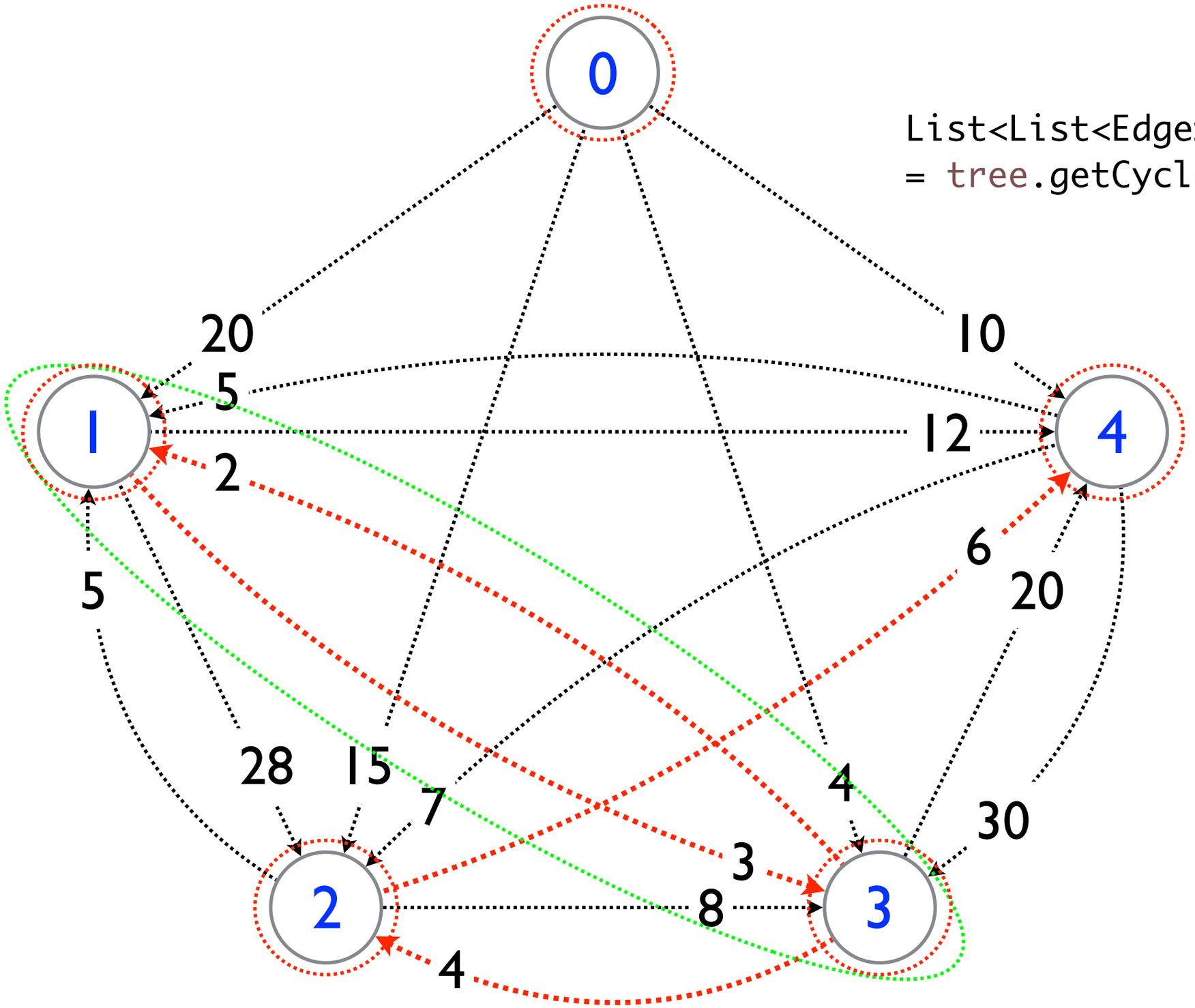
Chu–Liu-Edmonds' Algorithm




```
List<List<Edge>> cycles
= tree.getCycles();
```


EMORY UNIVERSITY

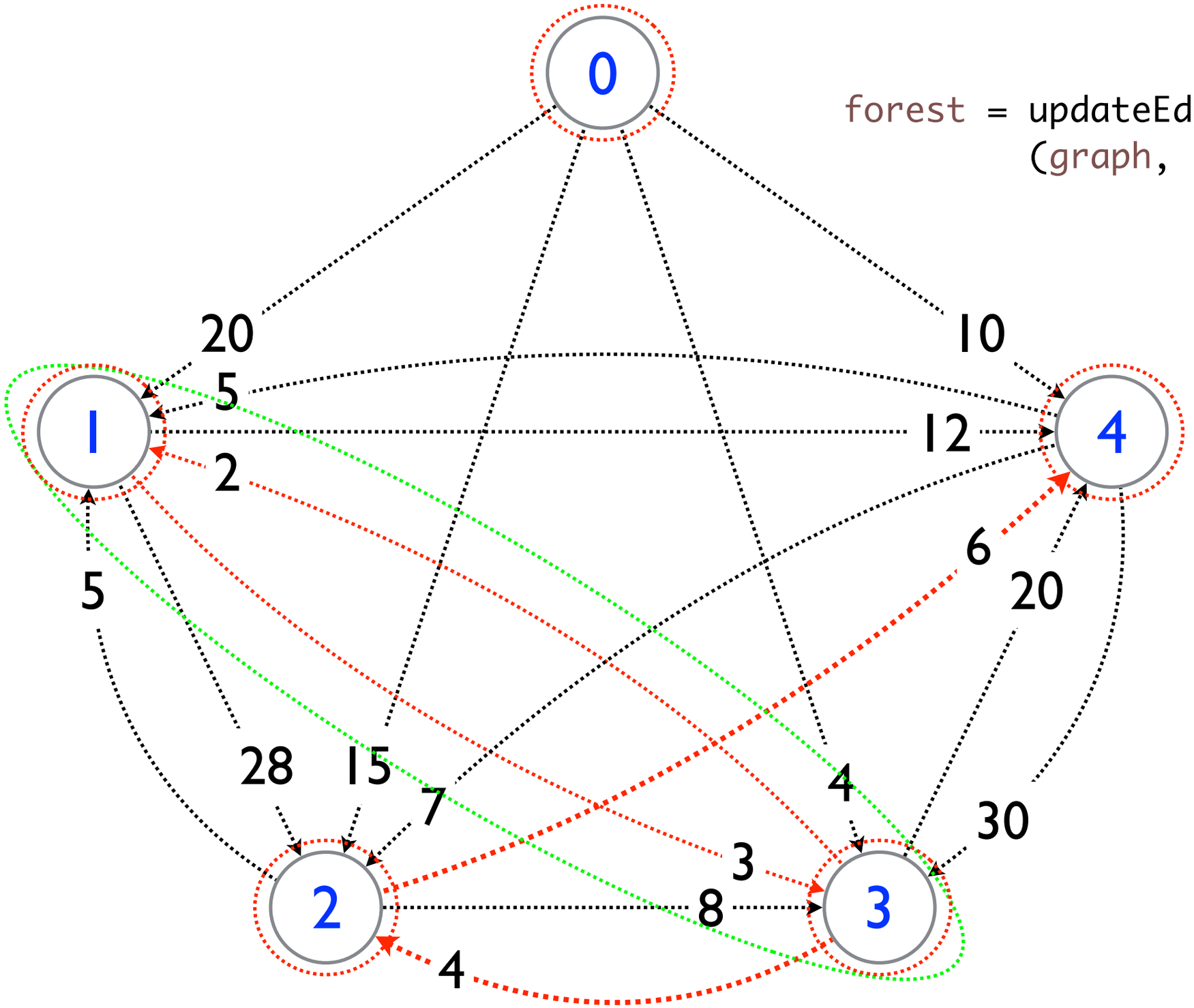
Chu–Liu-Edmonds' Algorithm




```
List<List<Edge>> cycles
= tree.getCycles();
```


EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm

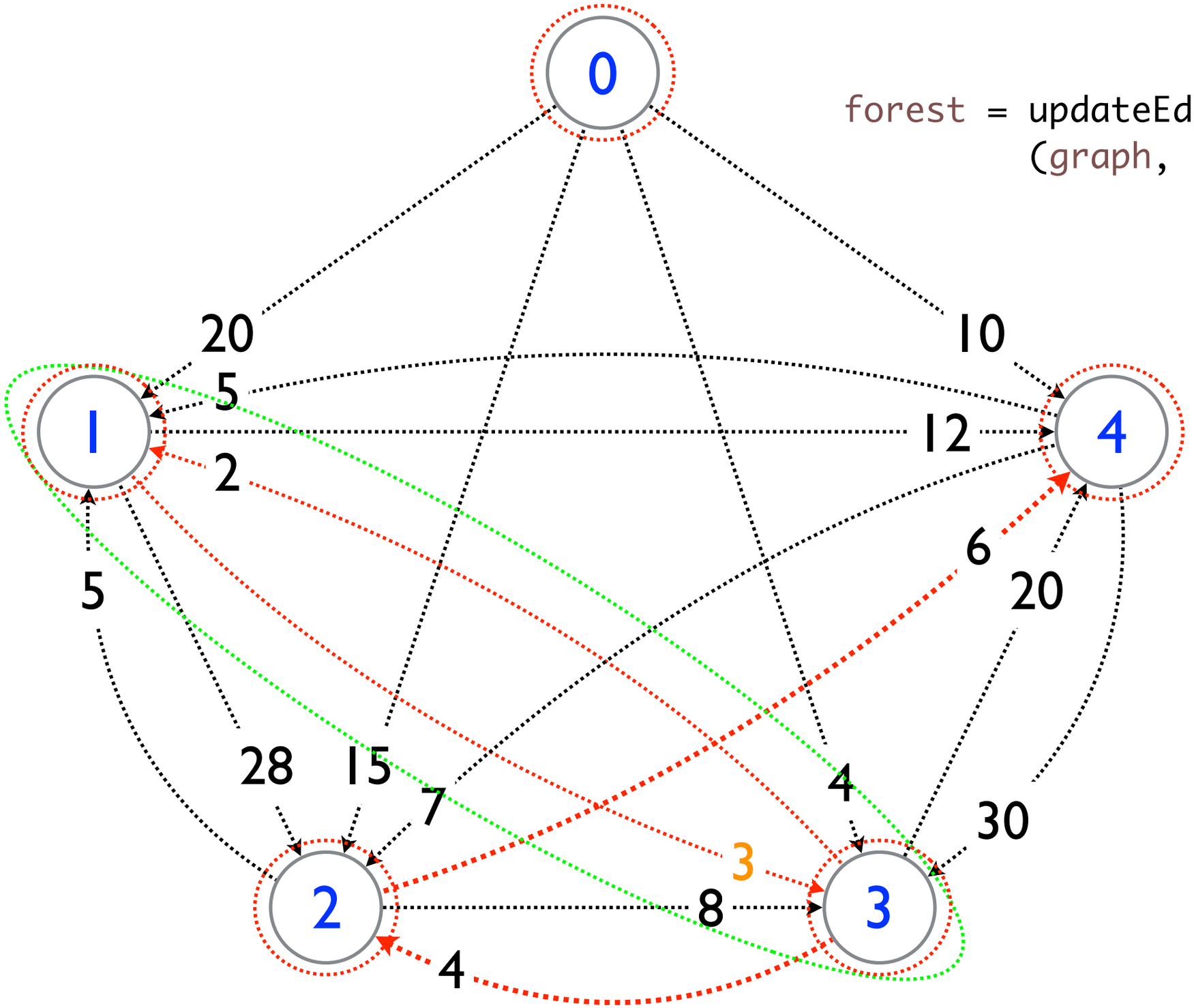



```
forest = updateEdgeWeights
         (graph, cycles);
```

EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm

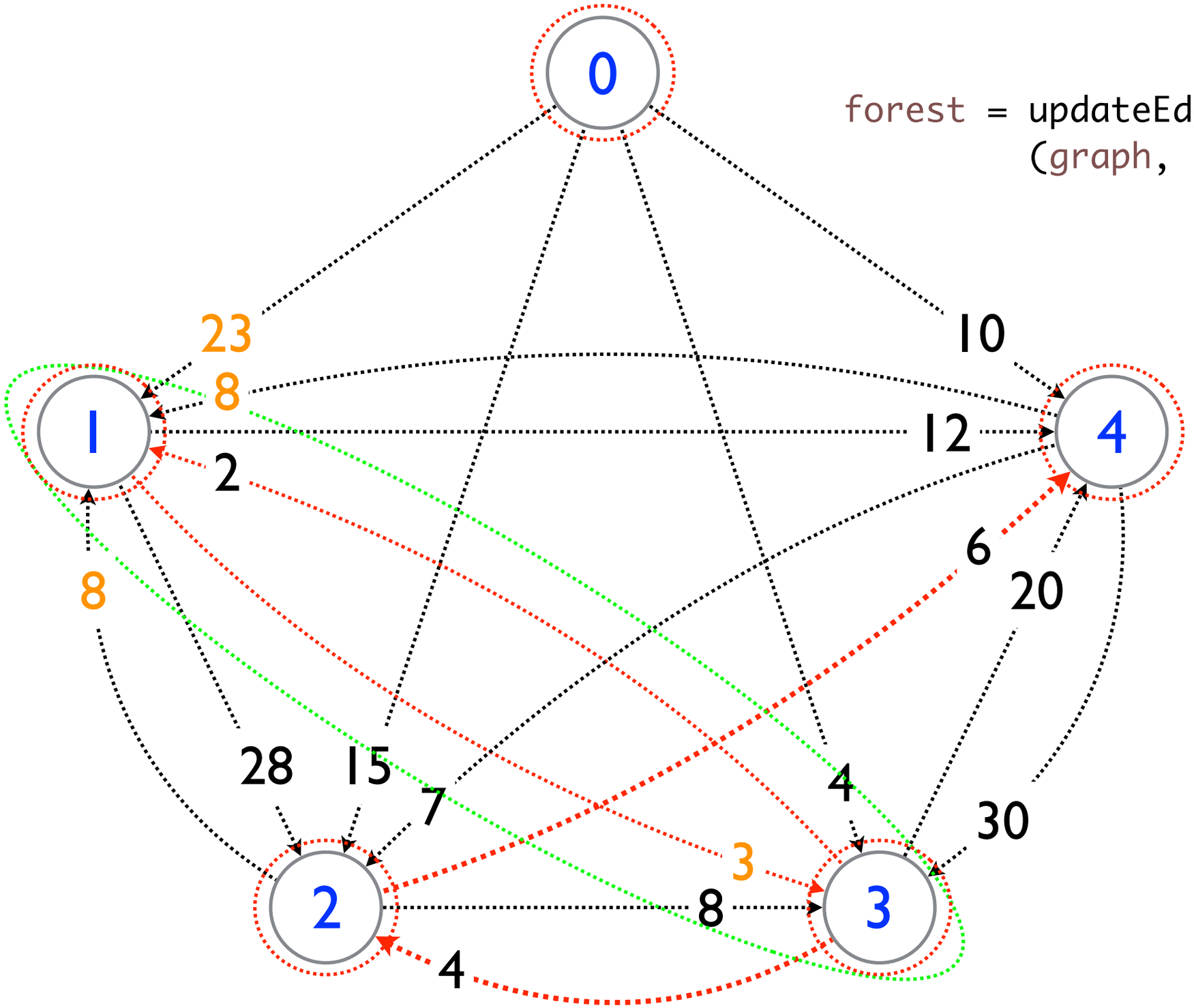
```
forest = updateEdgeWeights
         (graph, cycles);
```



EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm

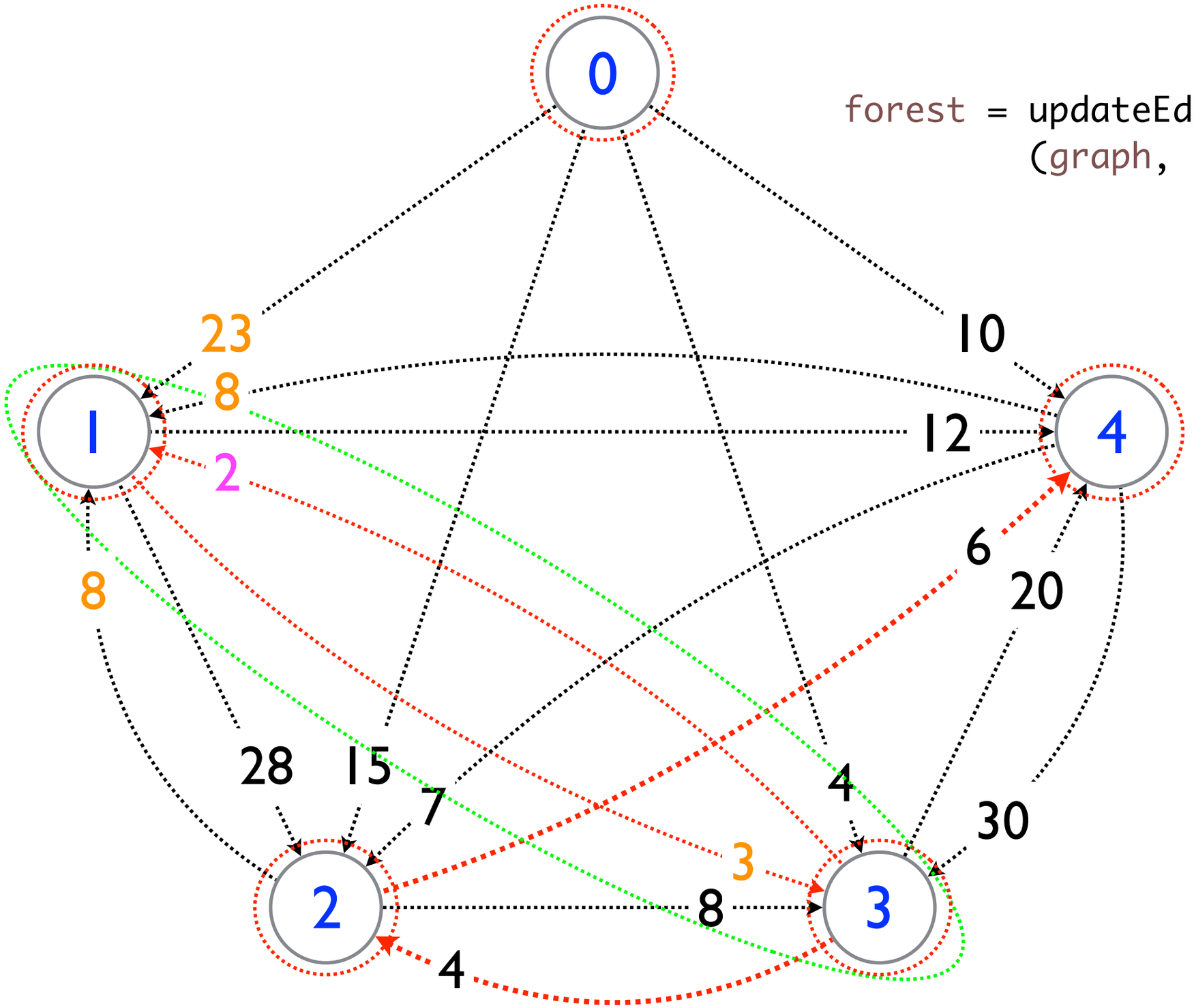
```
forest = updateEdgeWeights
         (graph, cycles);
```



EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm

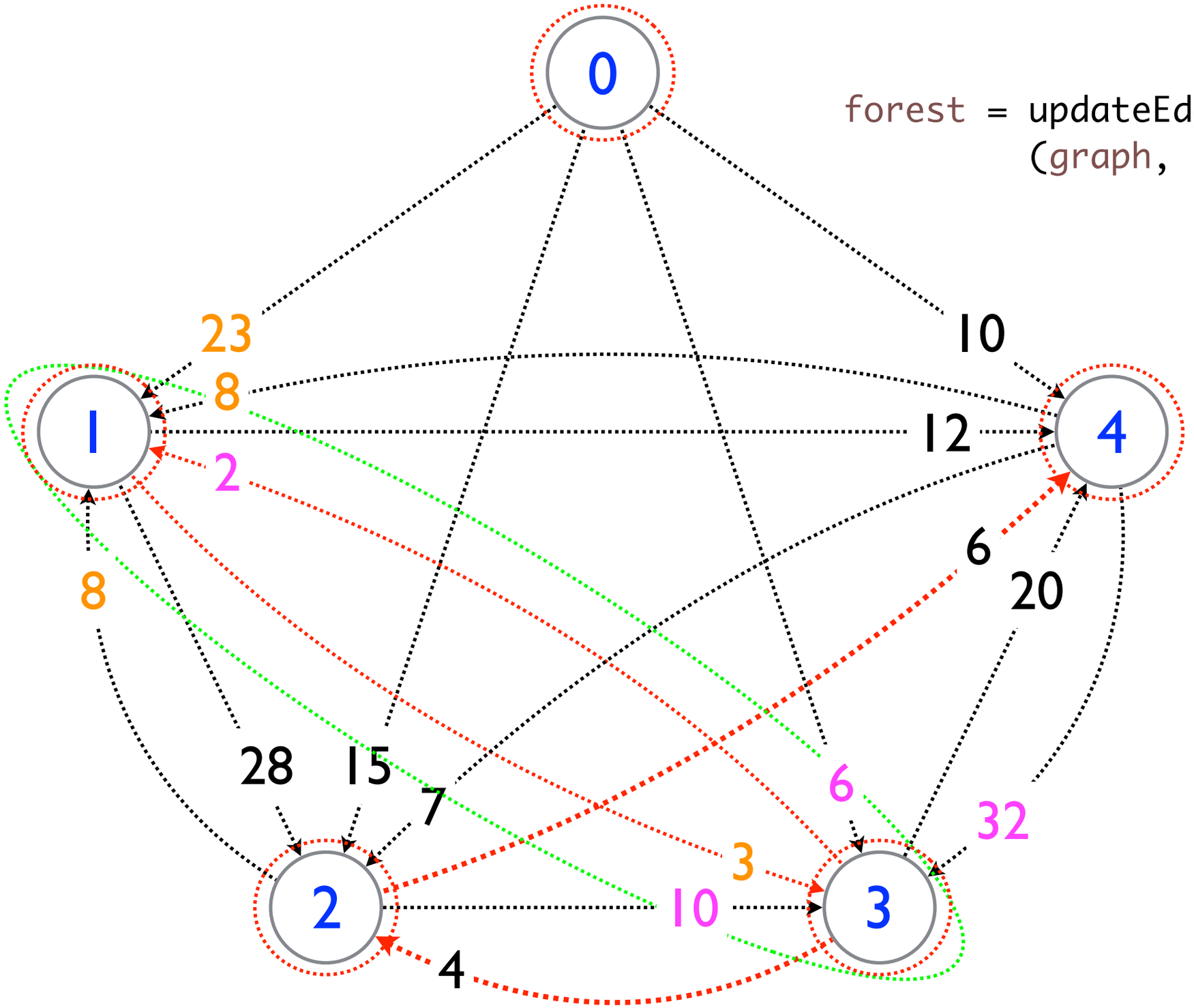
```
forest = updateEdgeWeights
         (graph, cycles);
```



EMORY UNIVERSITY

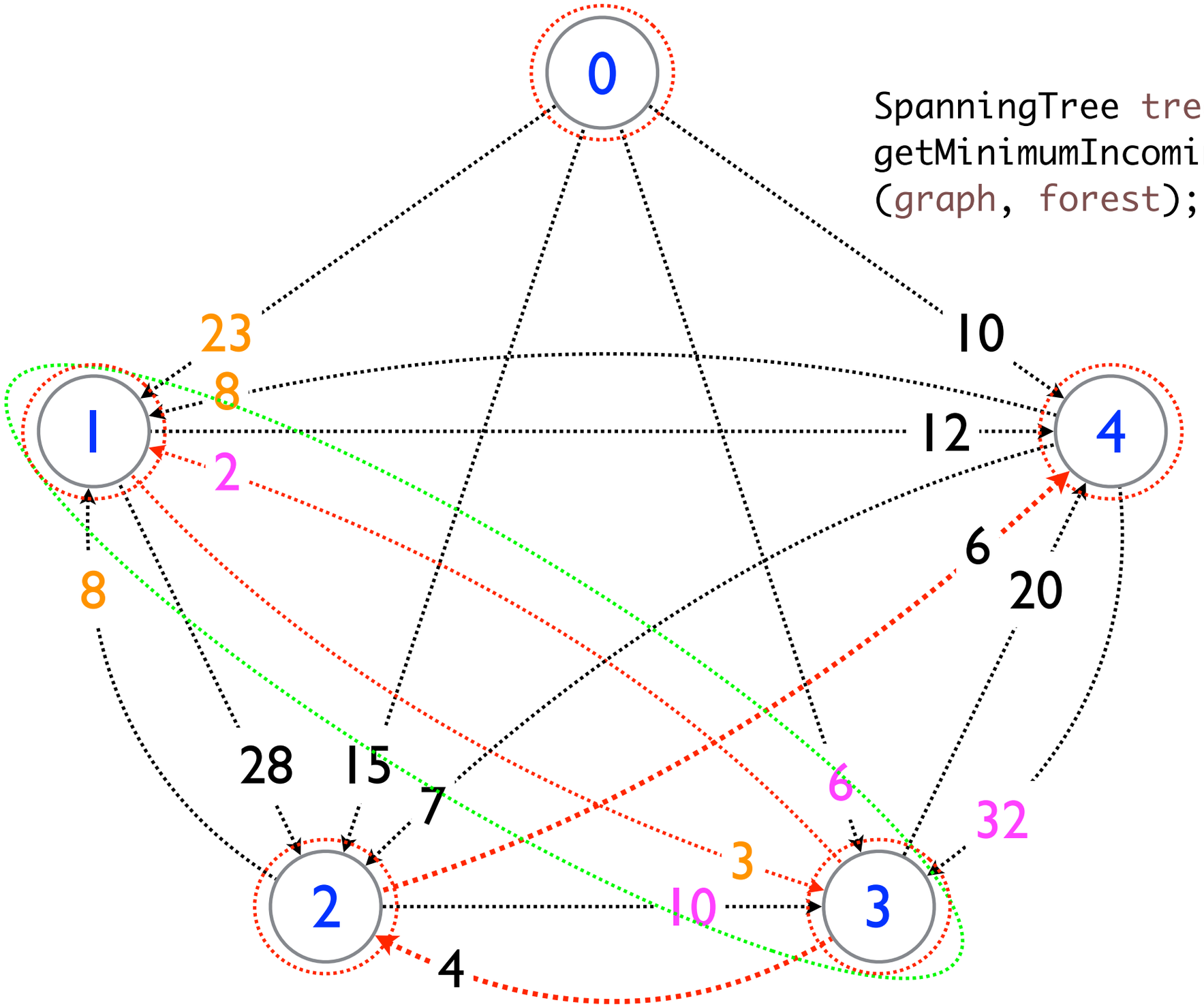
Chu–Liu-Edmonds' Algorithm

```
forest = updateEdgeWeights
         (graph, cycles);
```



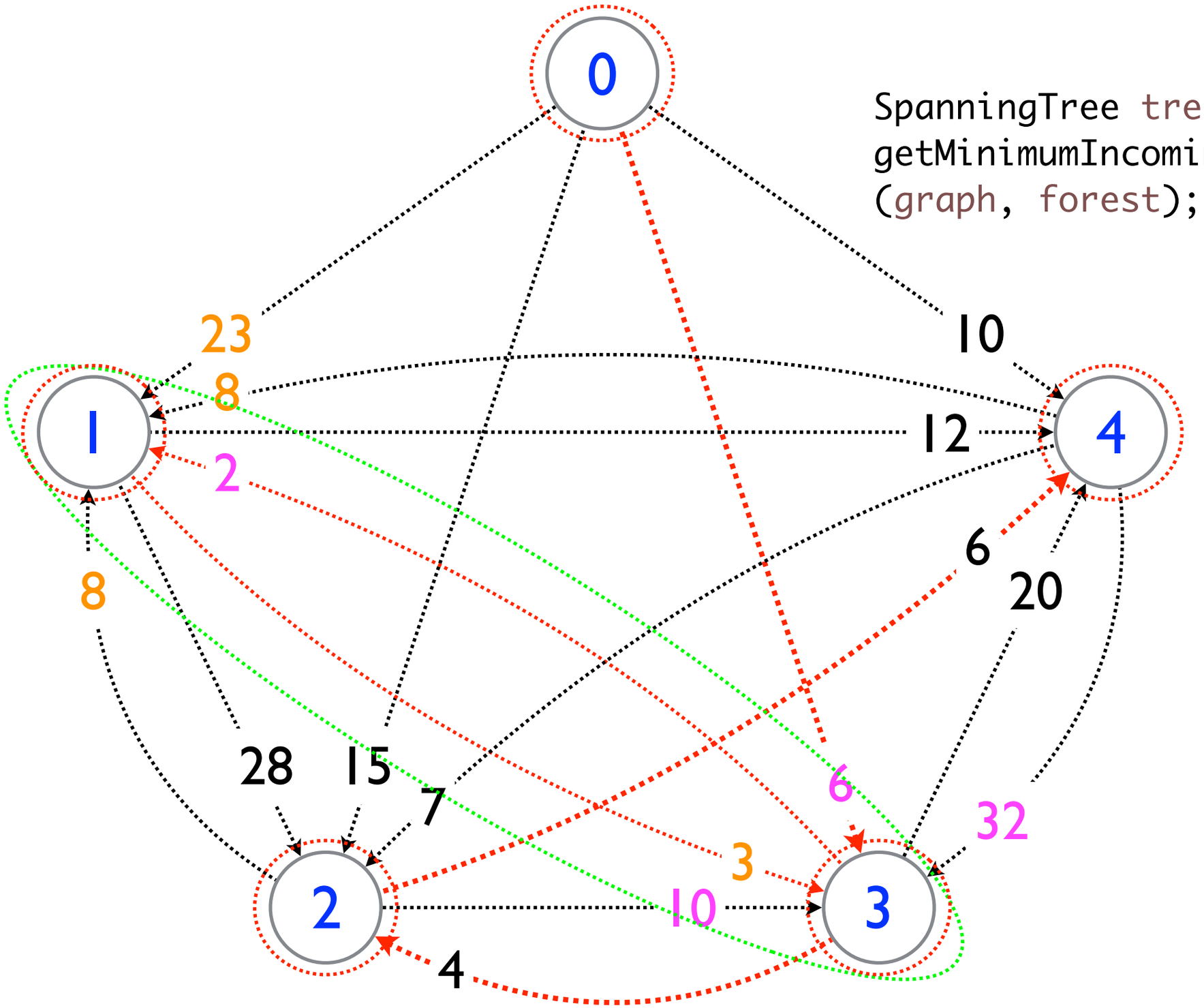
EMORY UNIVERSITY

Chu–Liu-Edmonds' Algorithm

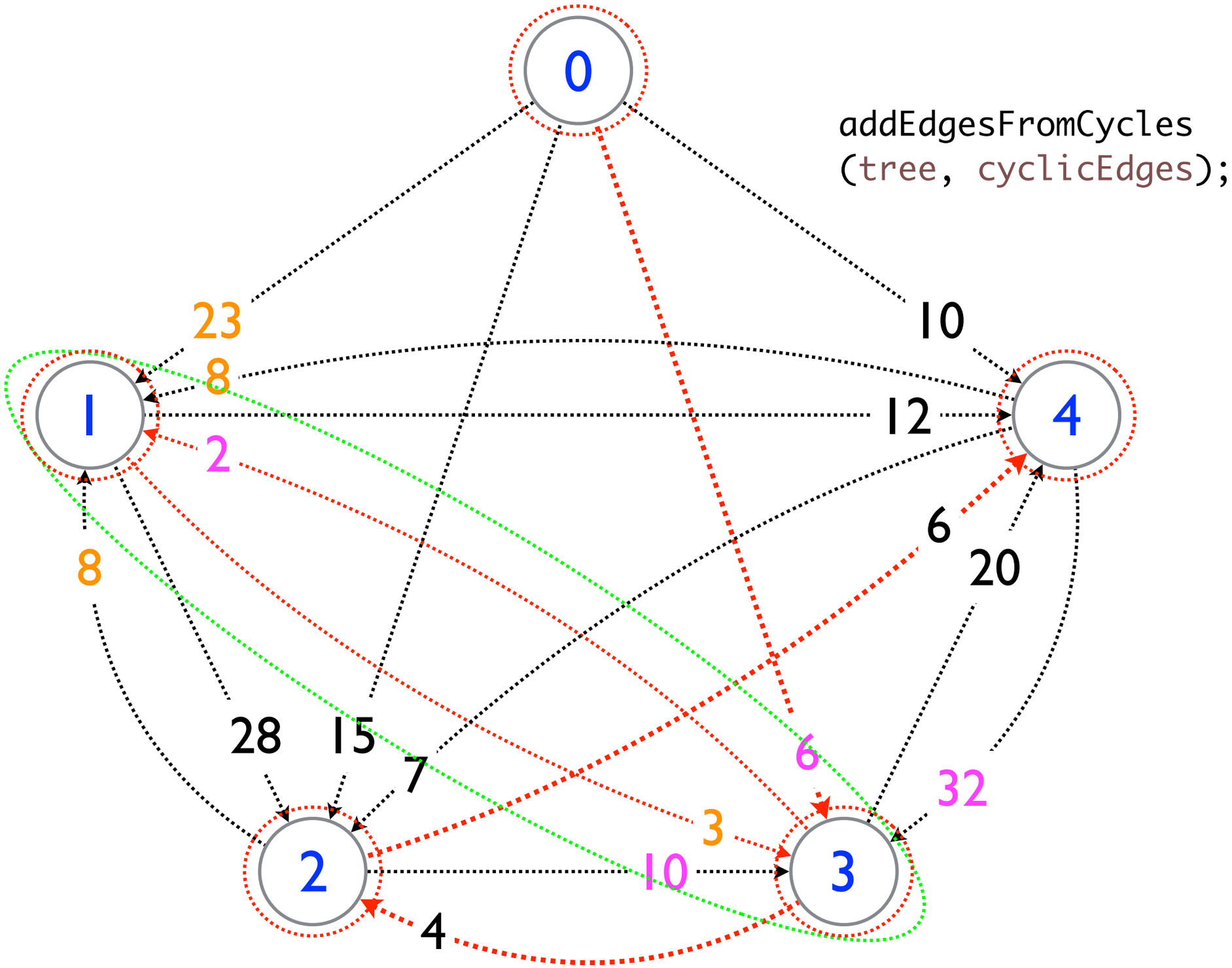



```
SpanningTree tree =
getMinimumIncomingEdges
(graph, forest);
```

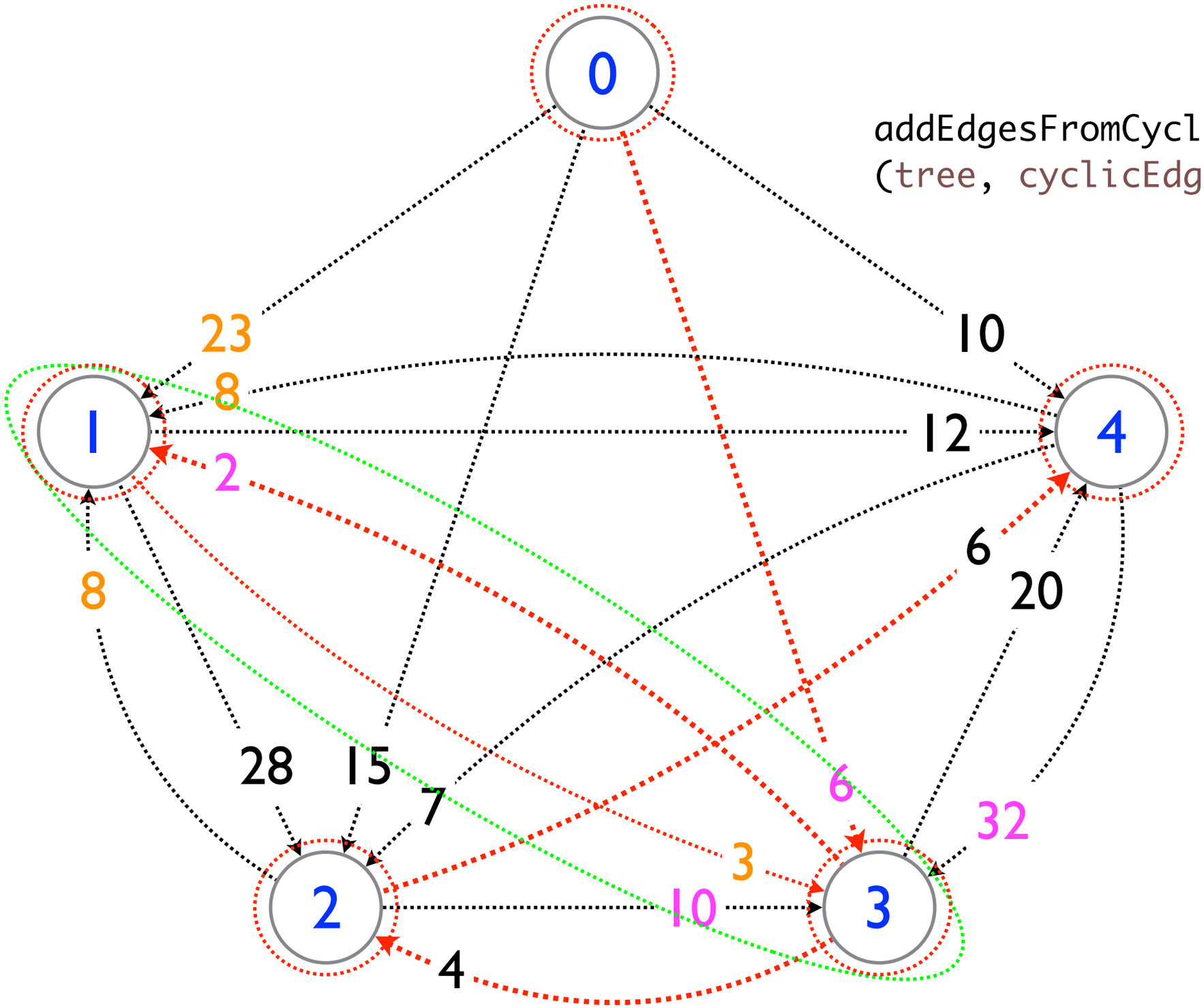
Chu–Liu-Edmonds' Algorithm



Chu–Liu-Edmonds' Algorithm

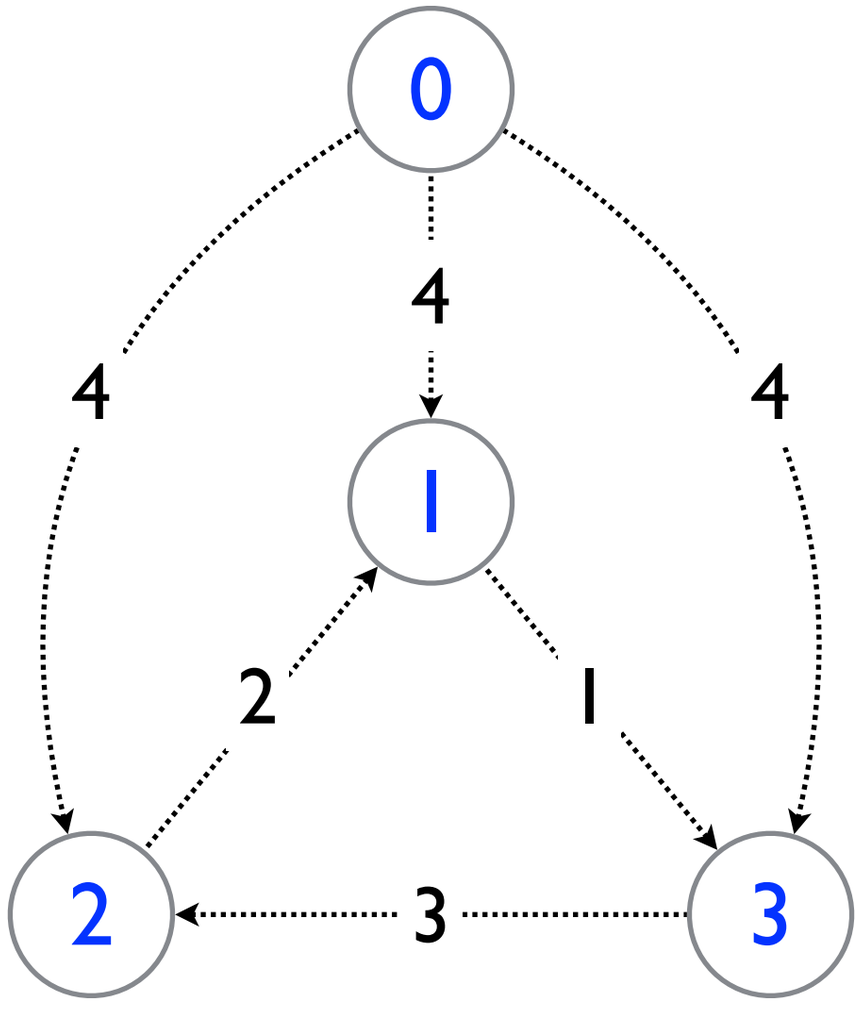


Chu–Liu-Edmonds' Algorithm

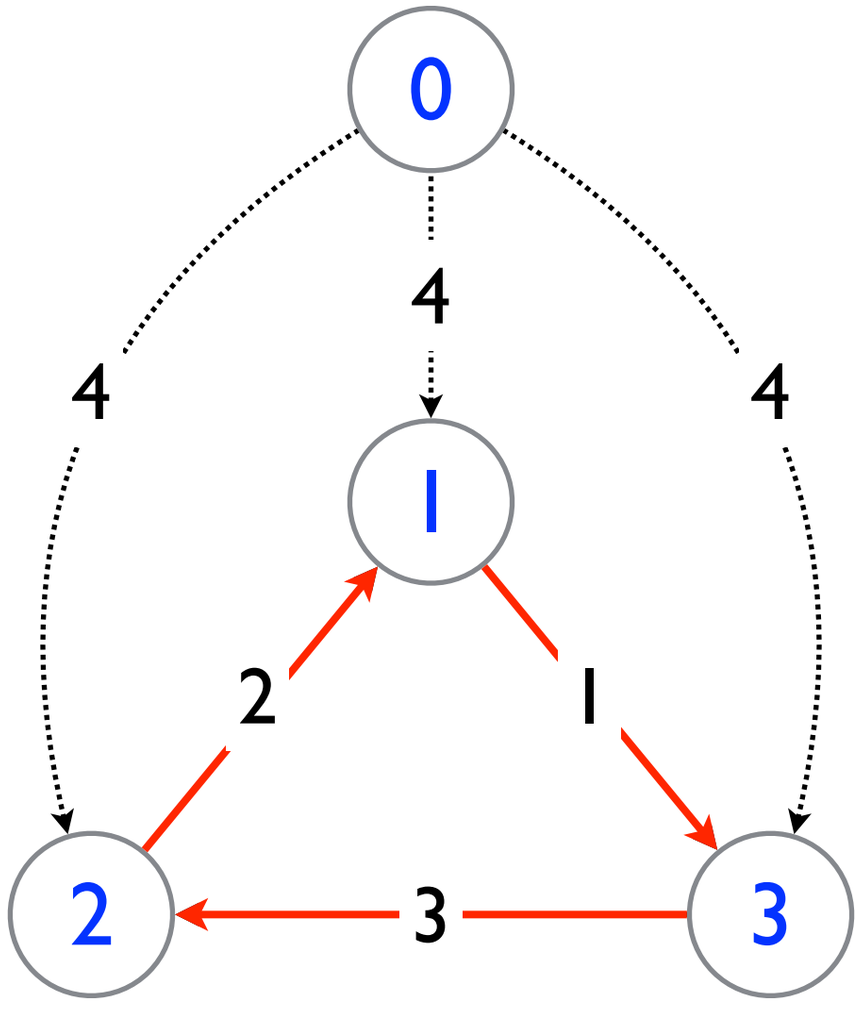



```
addEdgesFromCycles
(tree, cyclicEdges);
```

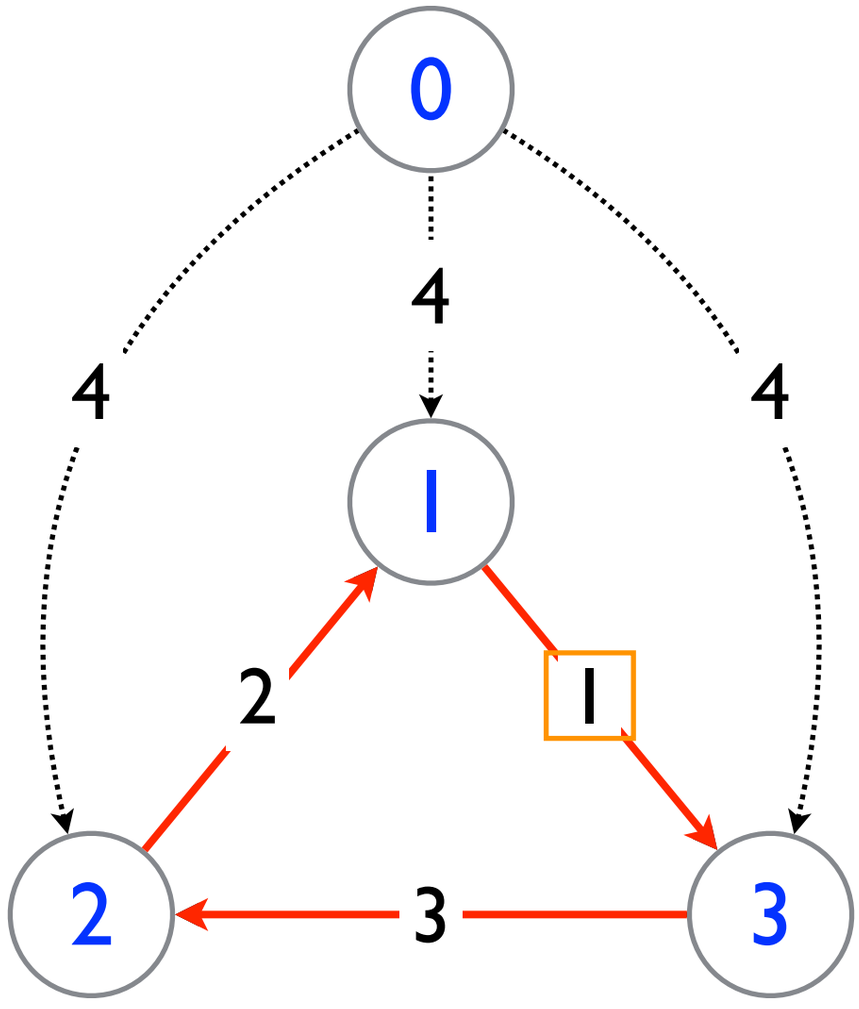
Chu–Liu-Edmonds' Algorithm



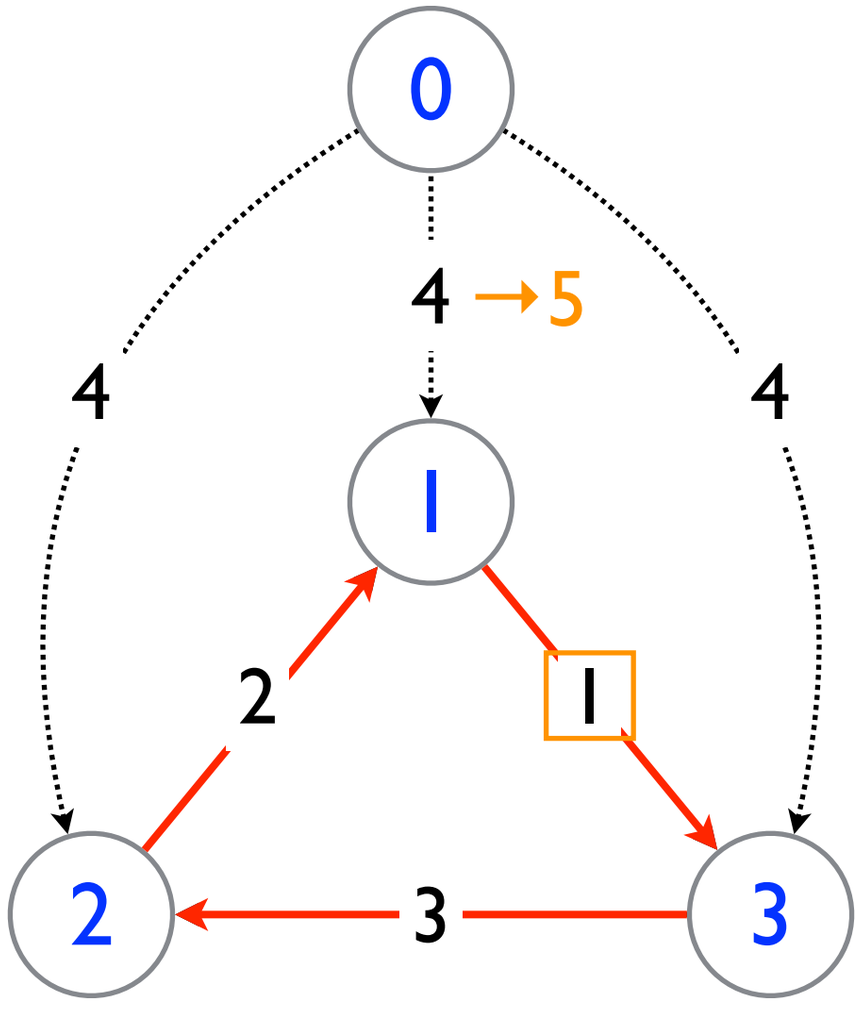
Chu–Liu-Edmonds' Algorithm



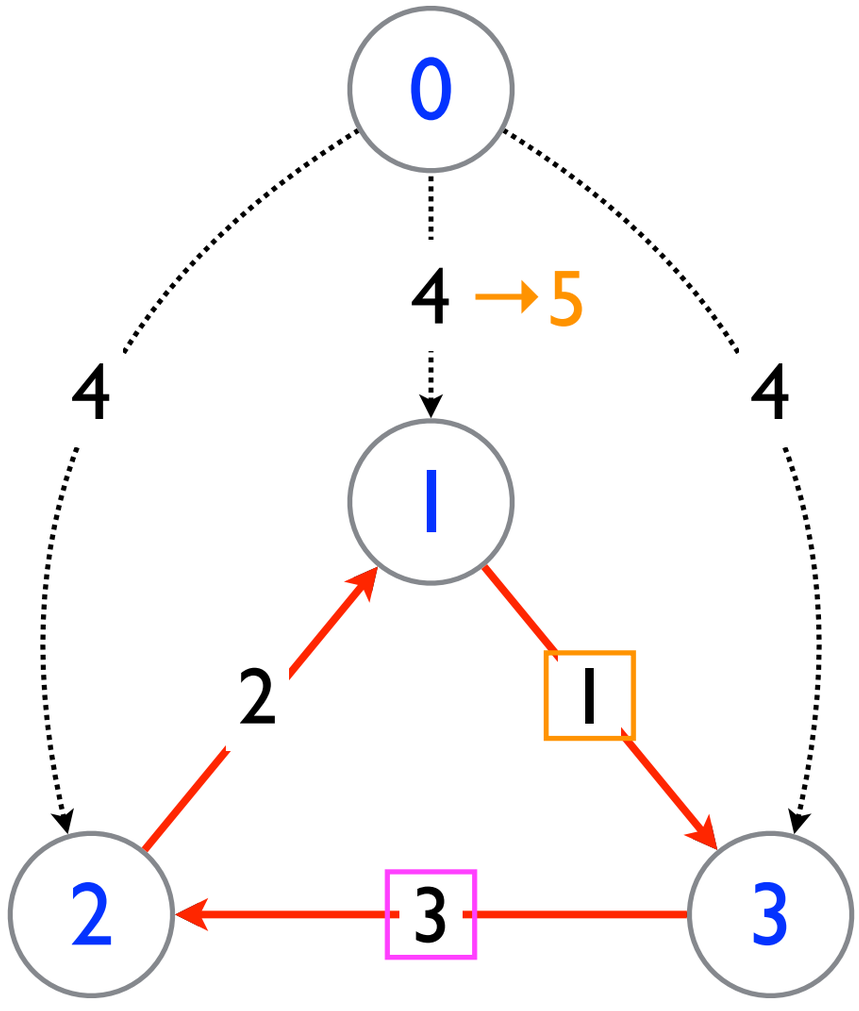
Chu–Liu-Edmonds' Algorithm



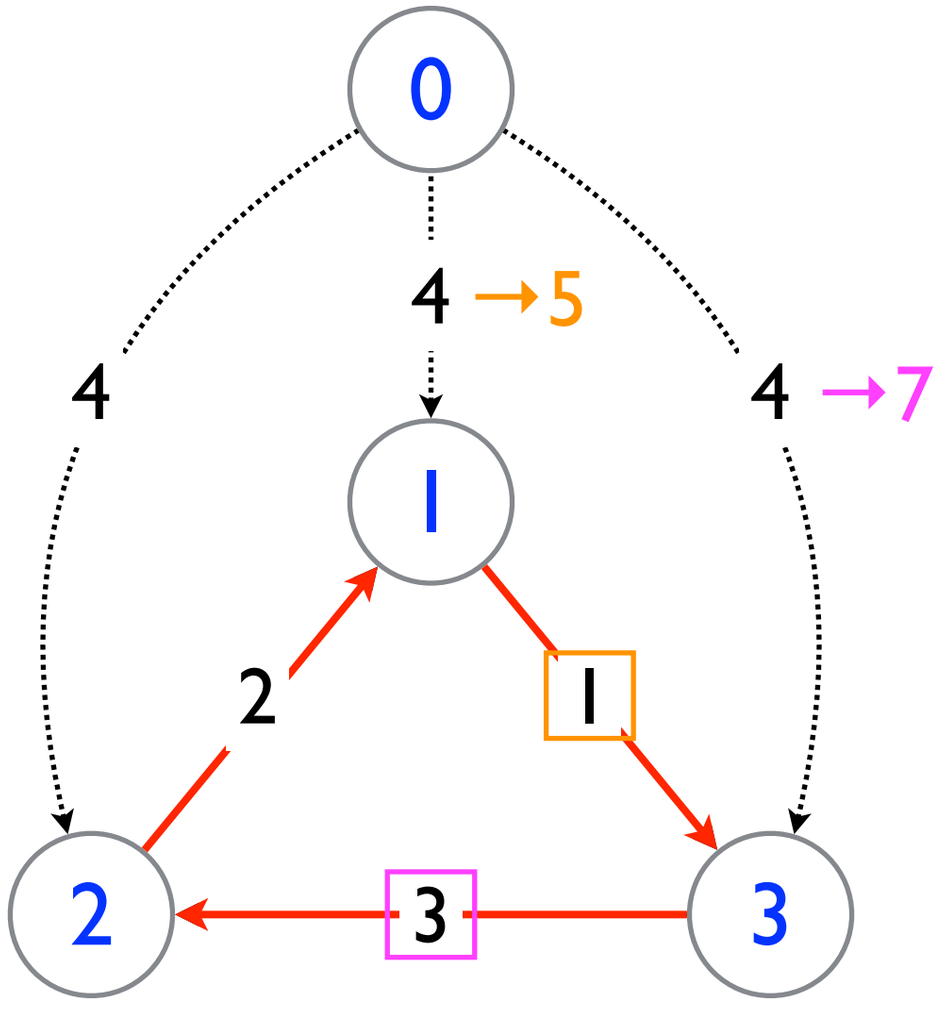
Chu–Liu-Edmonds' Algorithm



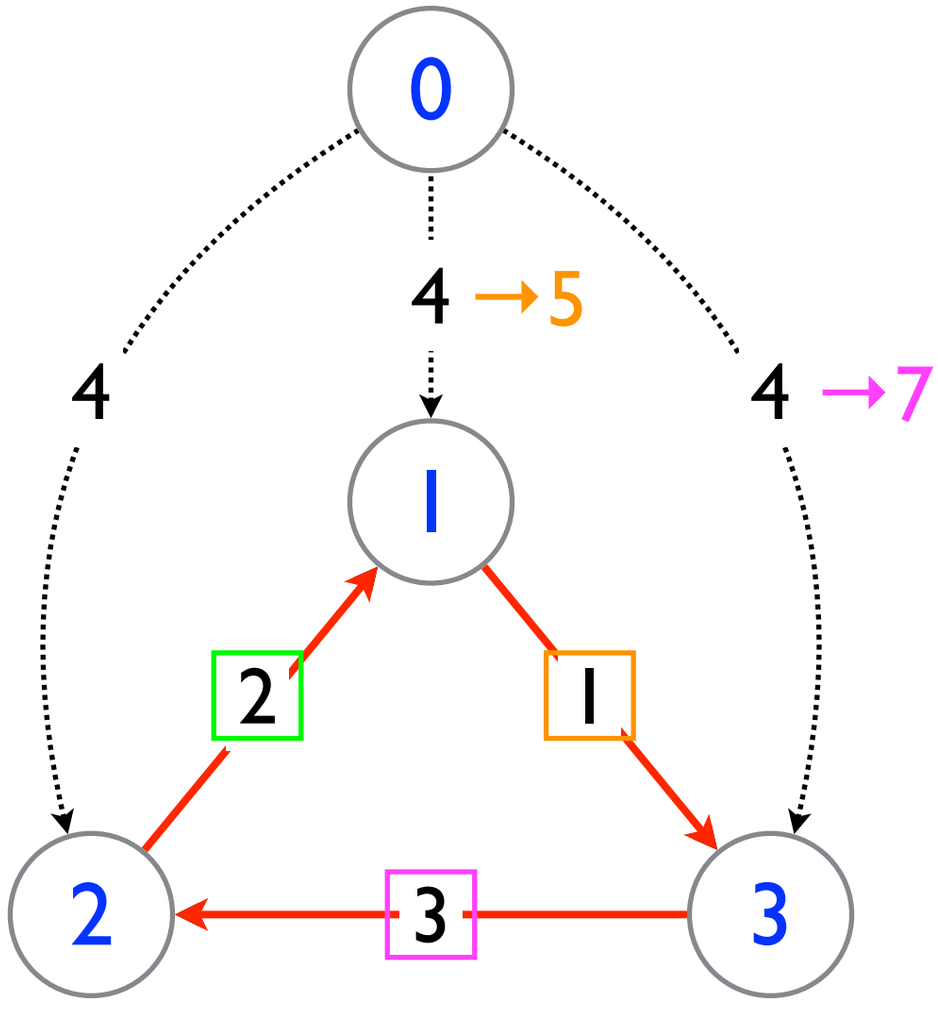
Chu–Liu-Edmonds' Algorithm



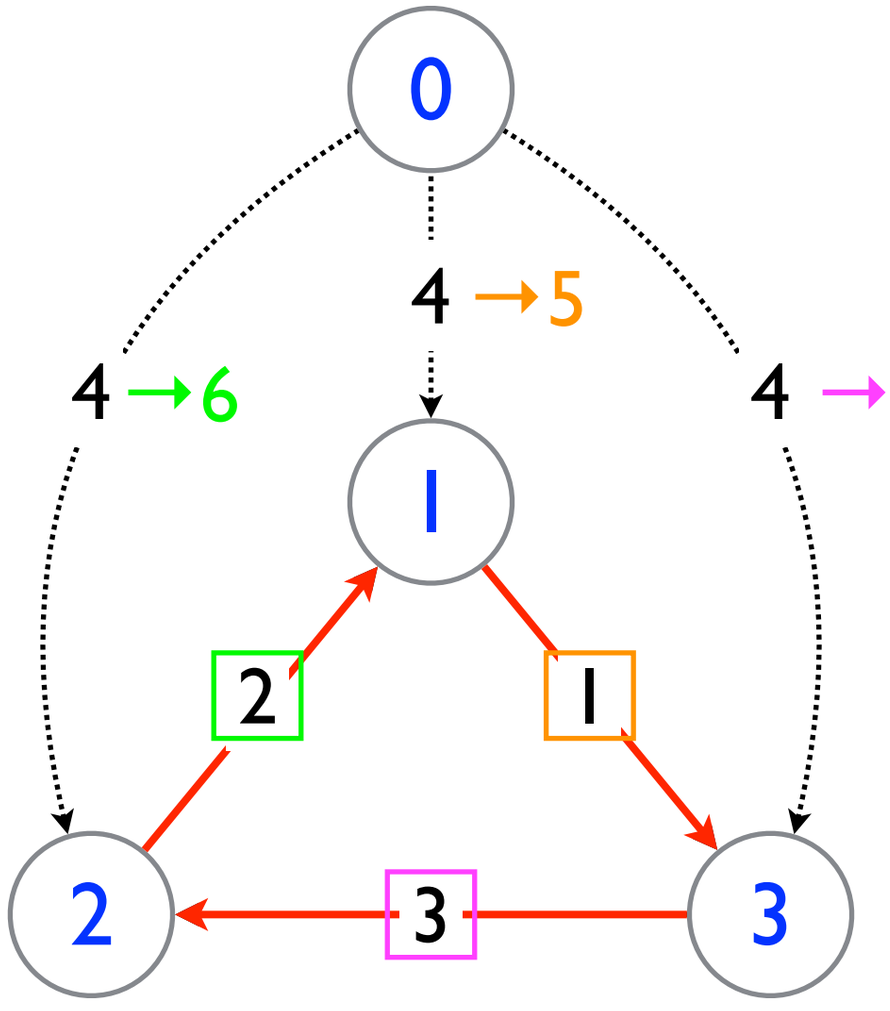
Chu–Liu-Edmonds' Algorithm



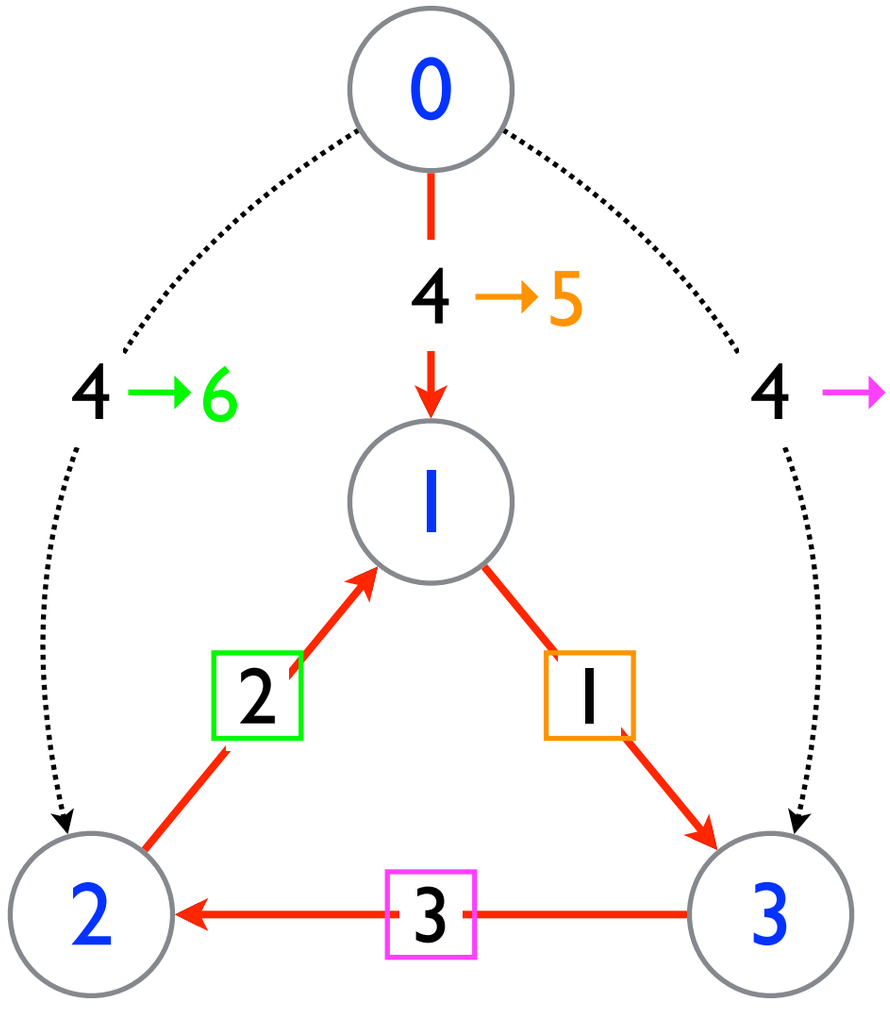
Chu–Liu-Edmonds' Algorithm



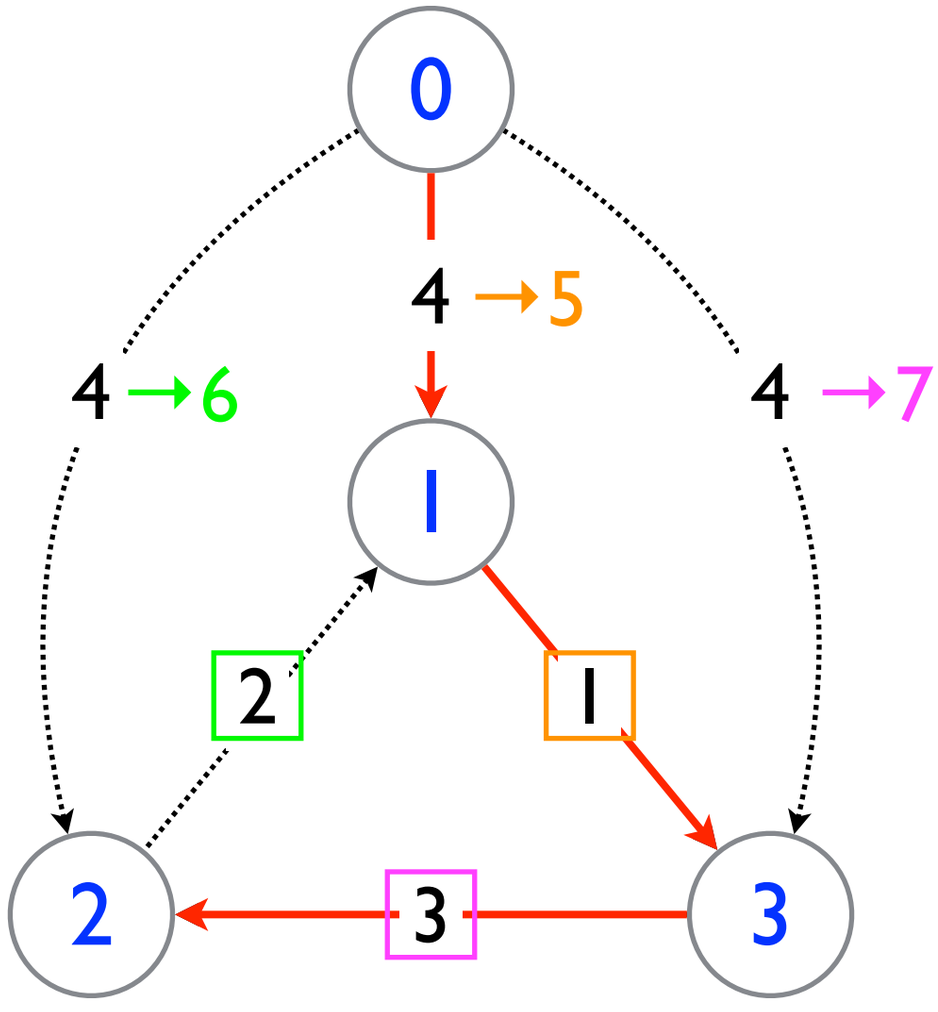
Chu–Liu-Edmonds' Algorithm



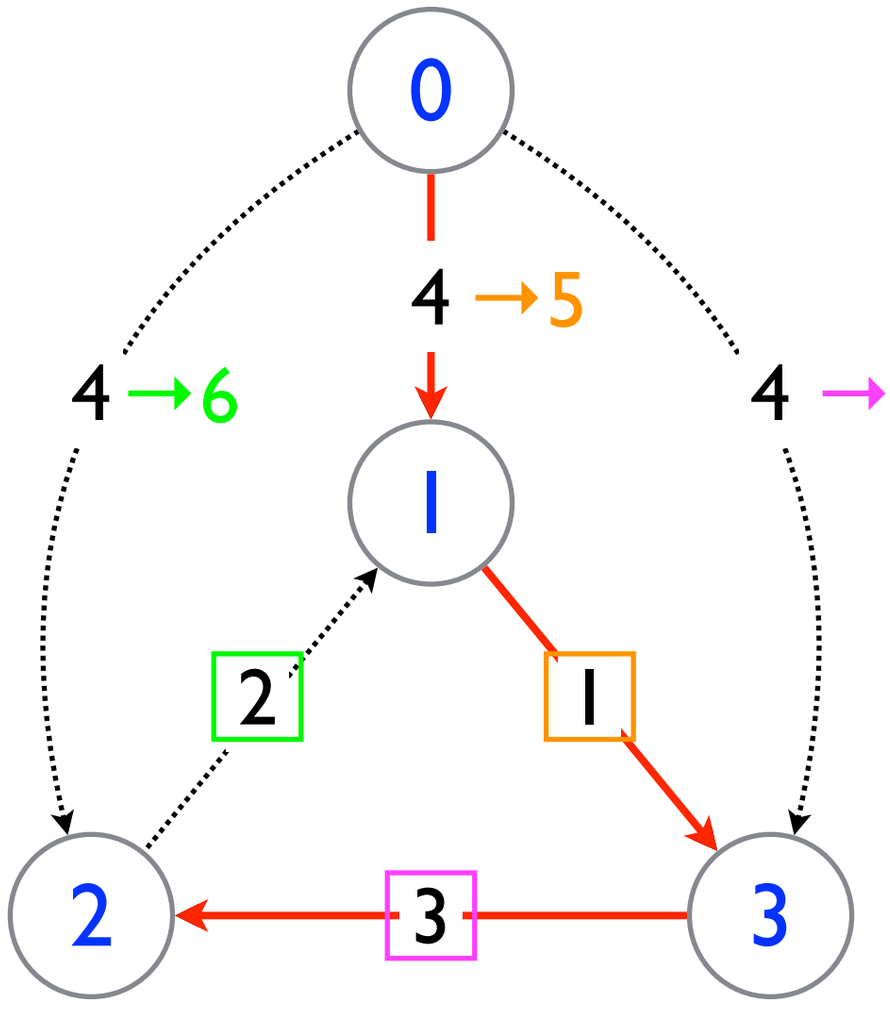
Chu–Liu-Edmonds' Algorithm



Chu–Liu-Edmonds' Algorithm

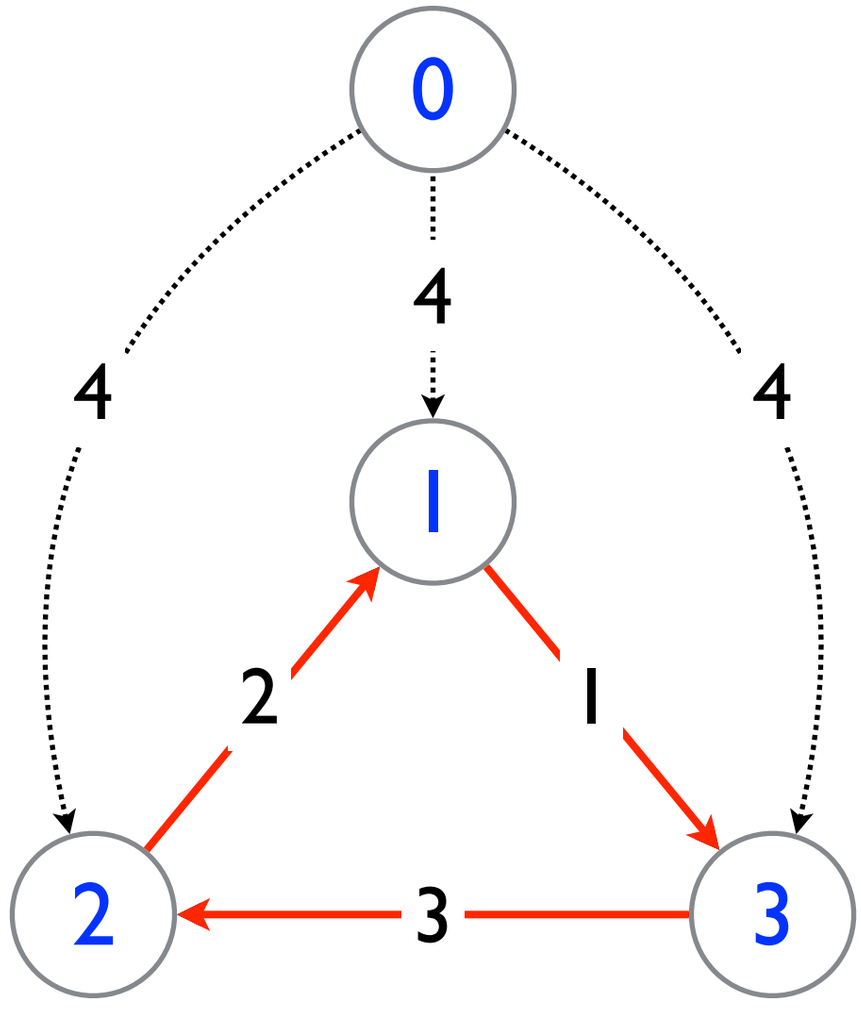


Chu–Liu-Edmonds' Algorithm

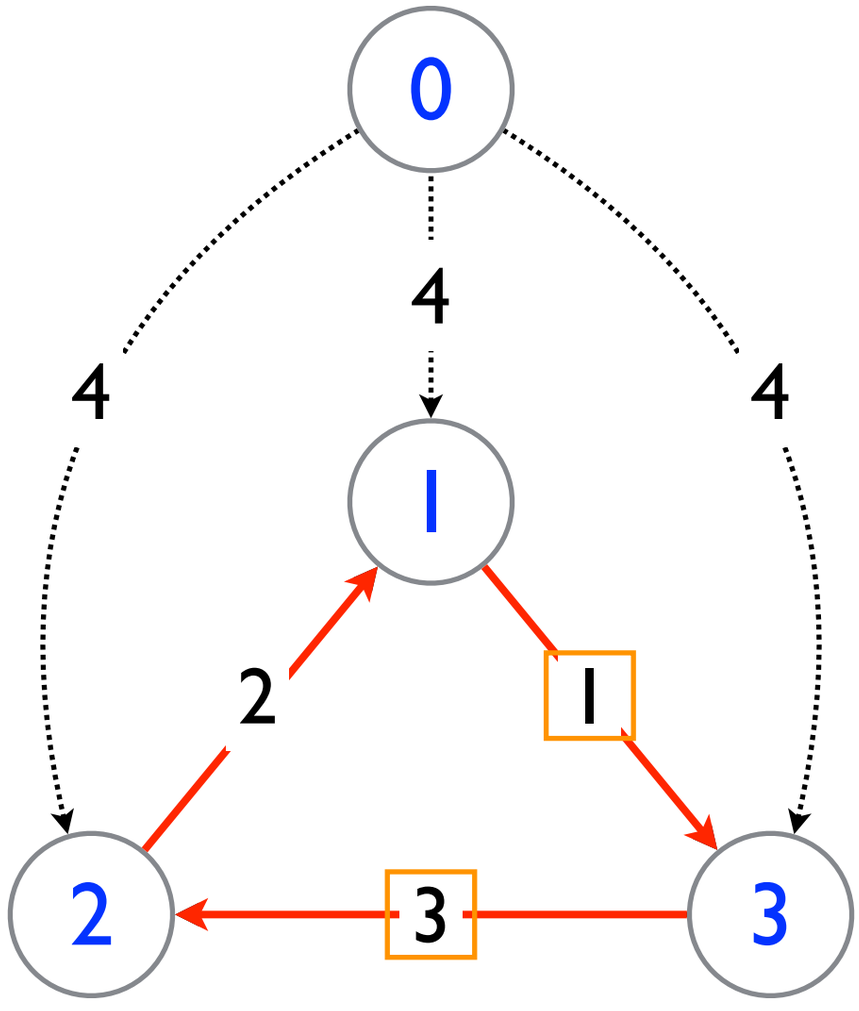


Minimum Spanning Tree?

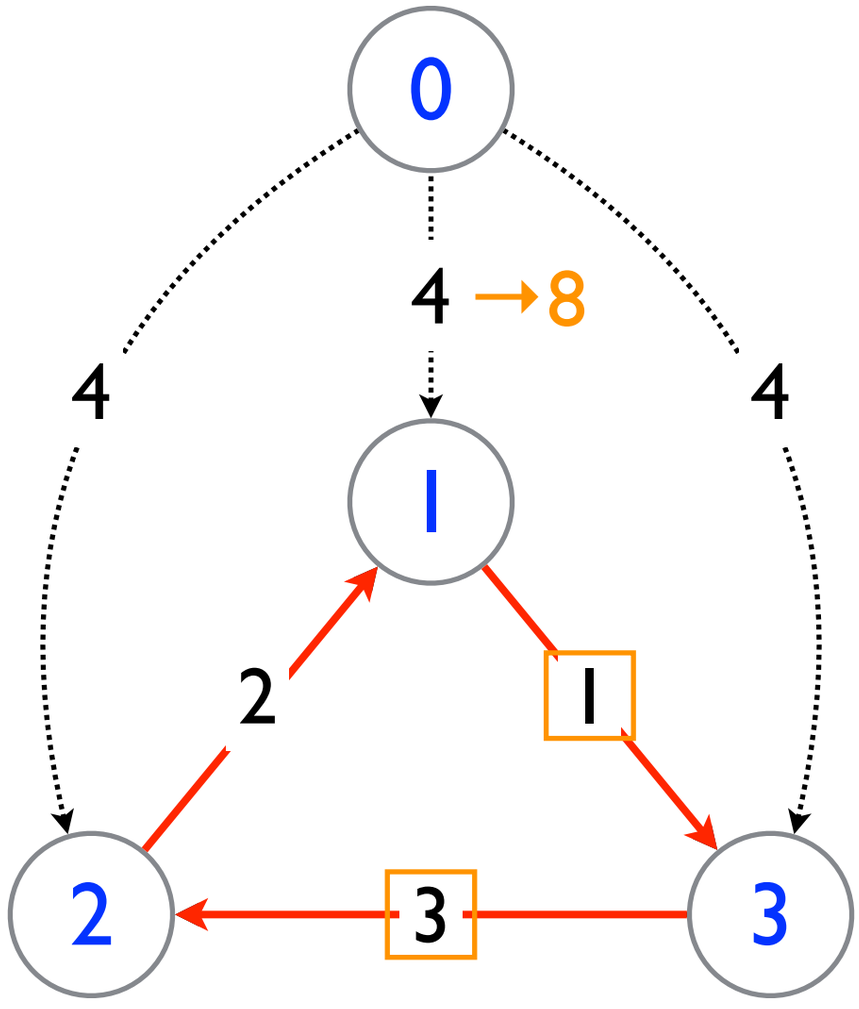
Chu–Liu-Edmonds' Algorithm



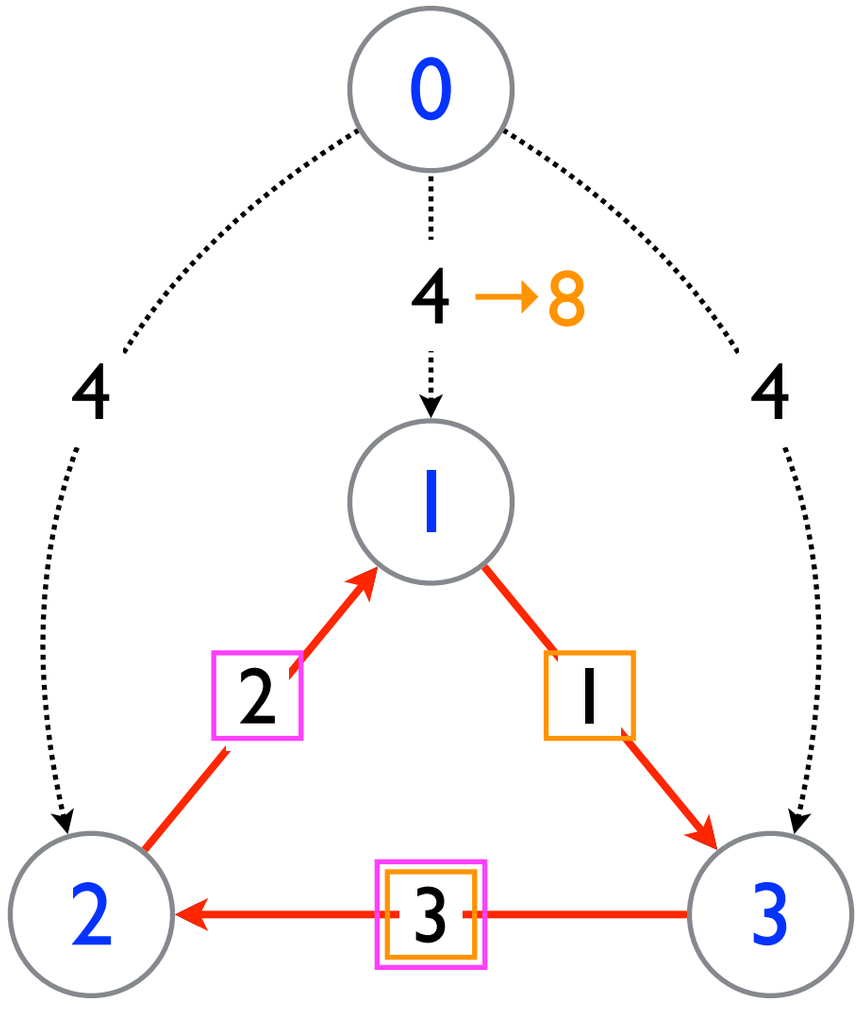
Chu–Liu-Edmonds' Algorithm



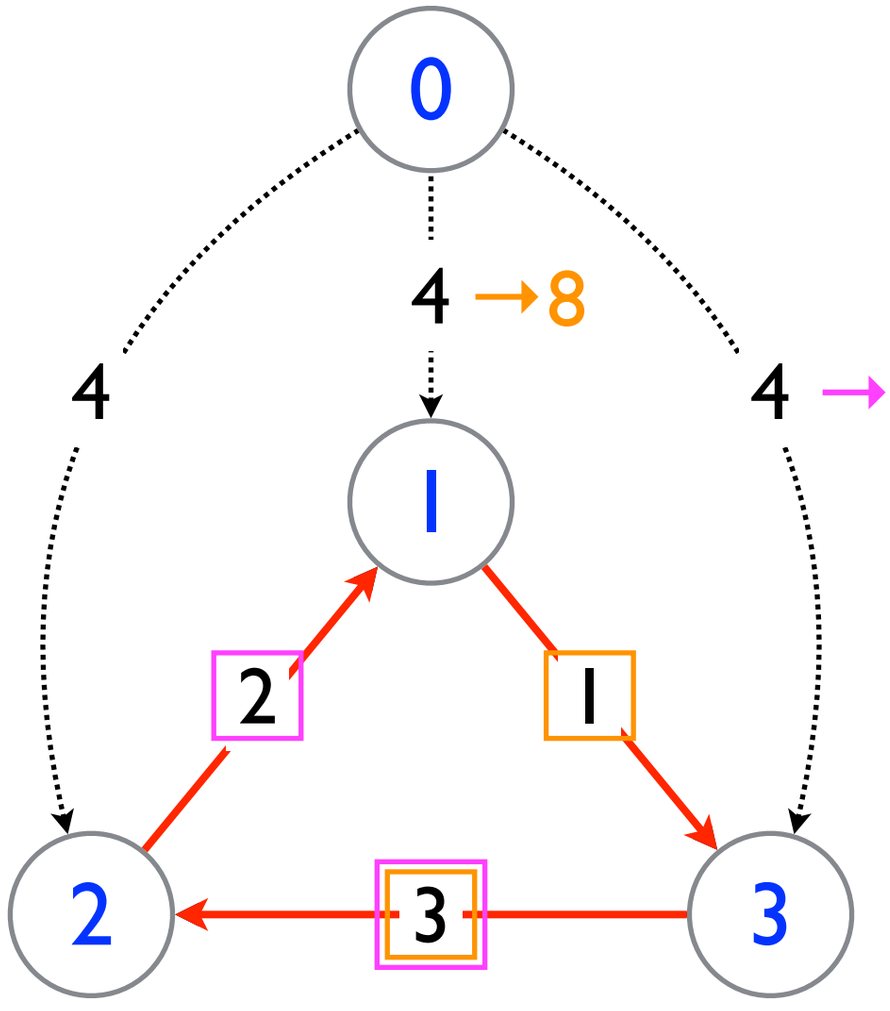
Chu–Liu-Edmonds' Algorithm



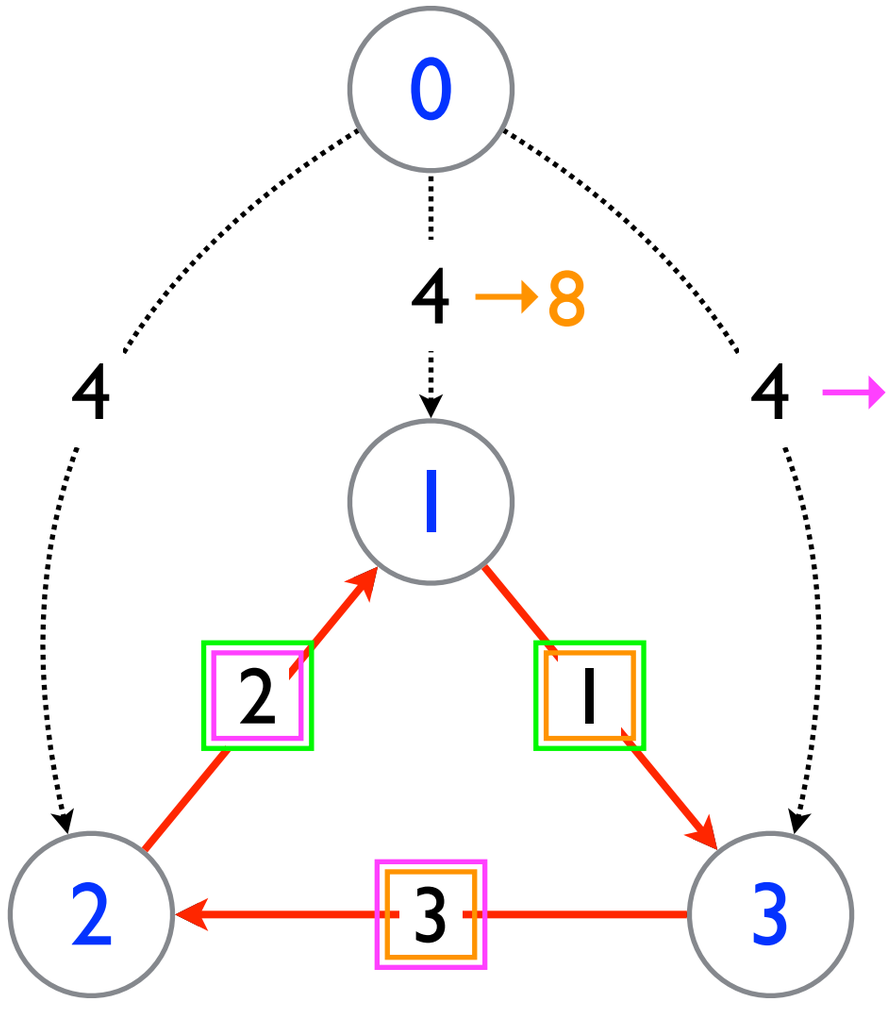
Chu–Liu-Edmonds' Algorithm



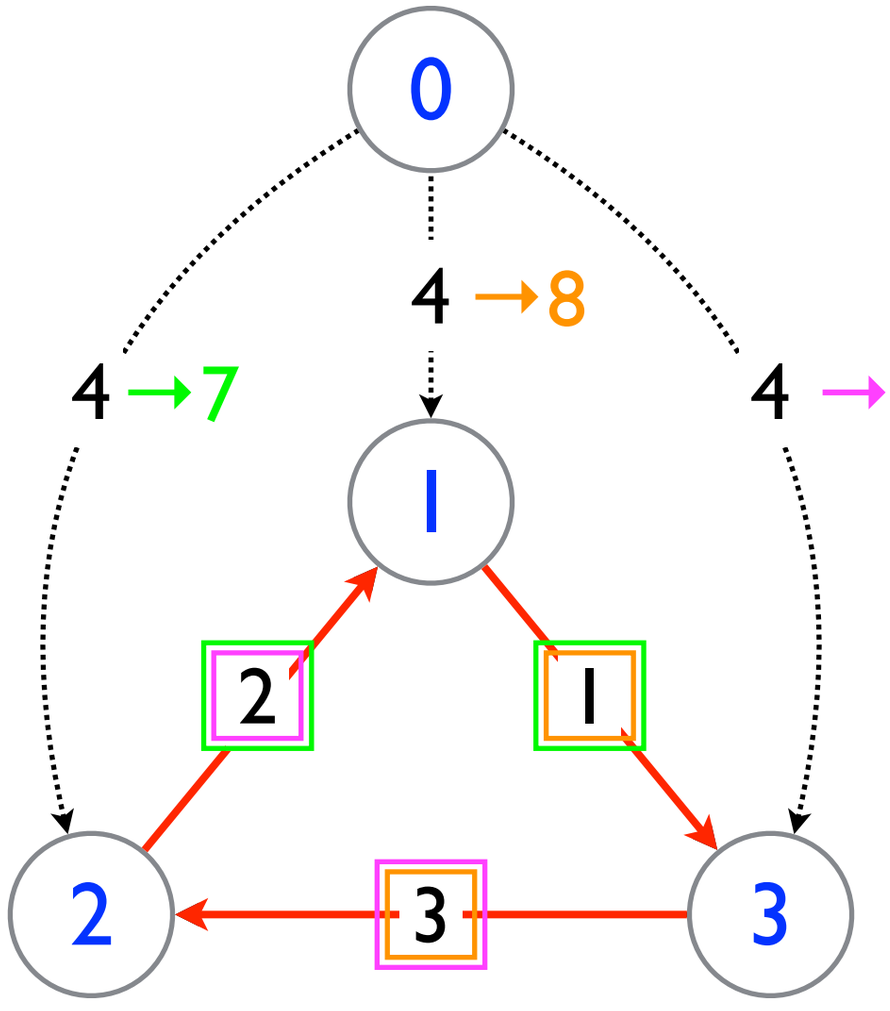
Chu–Liu-Edmonds' Algorithm



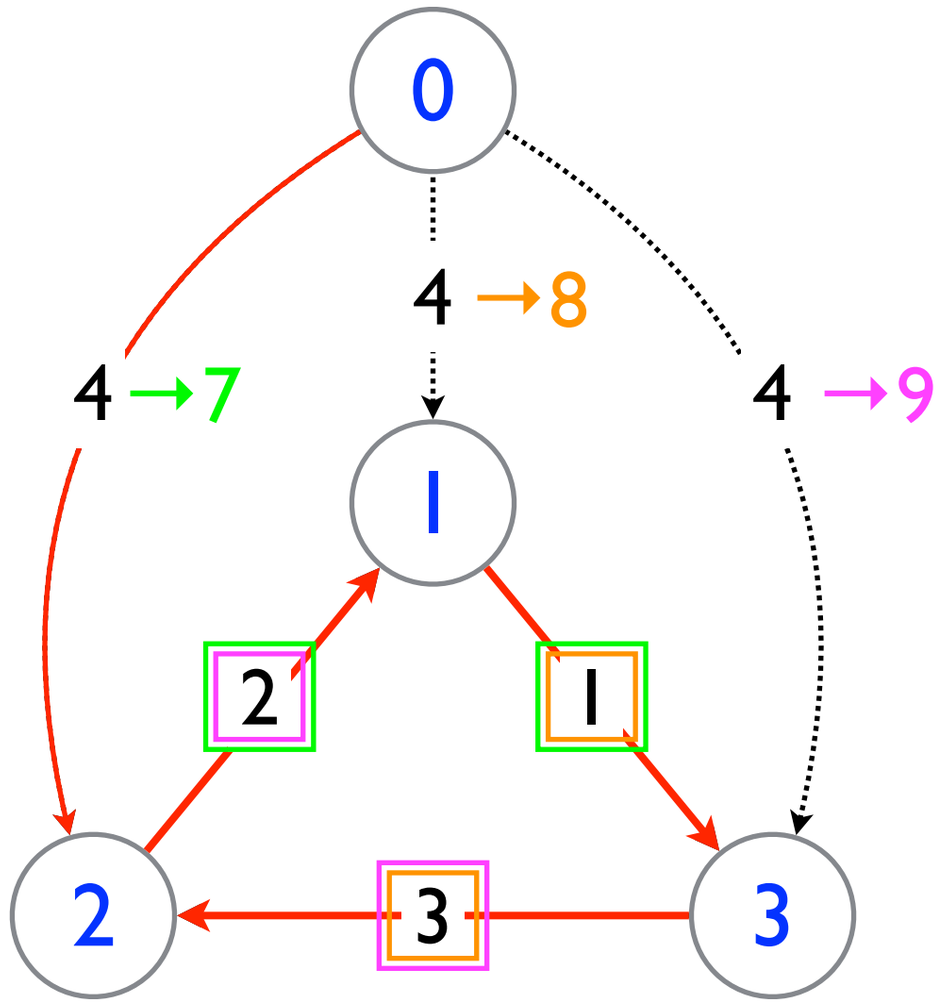
Chu–Liu-Edmonds' Algorithm



Chu–Liu-Edmonds' Algorithm



Chu–Liu-Edmonds' Algorithm



Chu–Liu-Edmonds' Algorithm