Prim's Minimum Spanning Tree Algorithm

(eager version)

William Fiset

Previous Video: Lazy Prim’s MST

<insert video clip>

Link to lazy Prim’s in the description

Eager Prim's

The lazy implementation of Prim's inserts up to E edges into the PQ. Therefore, each poll operation on the PQ is **O(log(E))**.

Instead of blindly inserting edges into a PQ which could later become stale, the eager version of Prim's tracks **(node, edge) key-value pairs** that can easily be **updated** and **polled** to determine the next best edge to add to the MST.

Eager Prim's

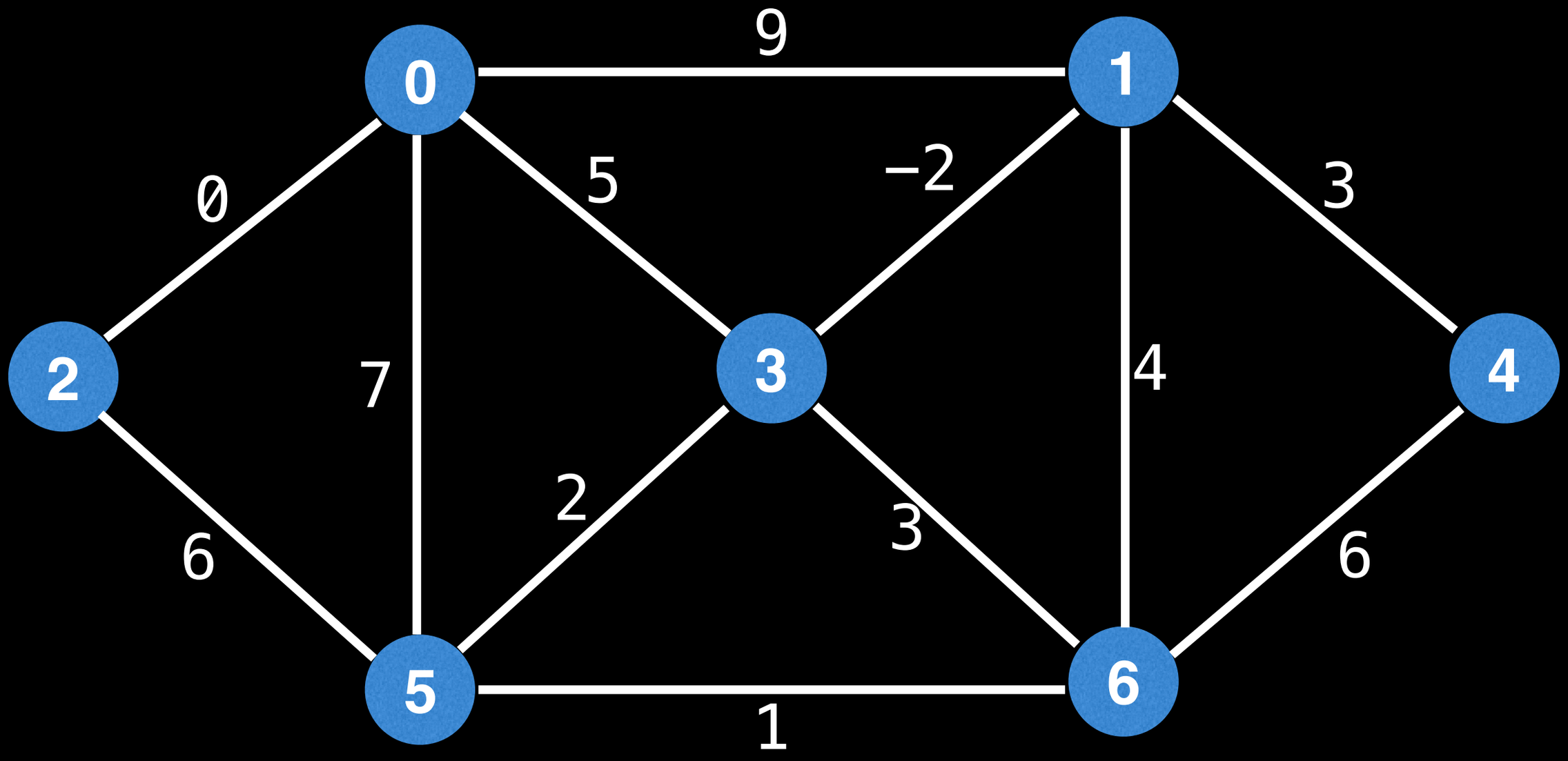
Key realization: for any MST with directed edges, each node is paired with **exactly one** of its incoming edges (except for the start node).

This can easily be seen on a directed MST where you can have multiple edges leaving a node, but at most one edge entering a node.

Let's take a closer look…

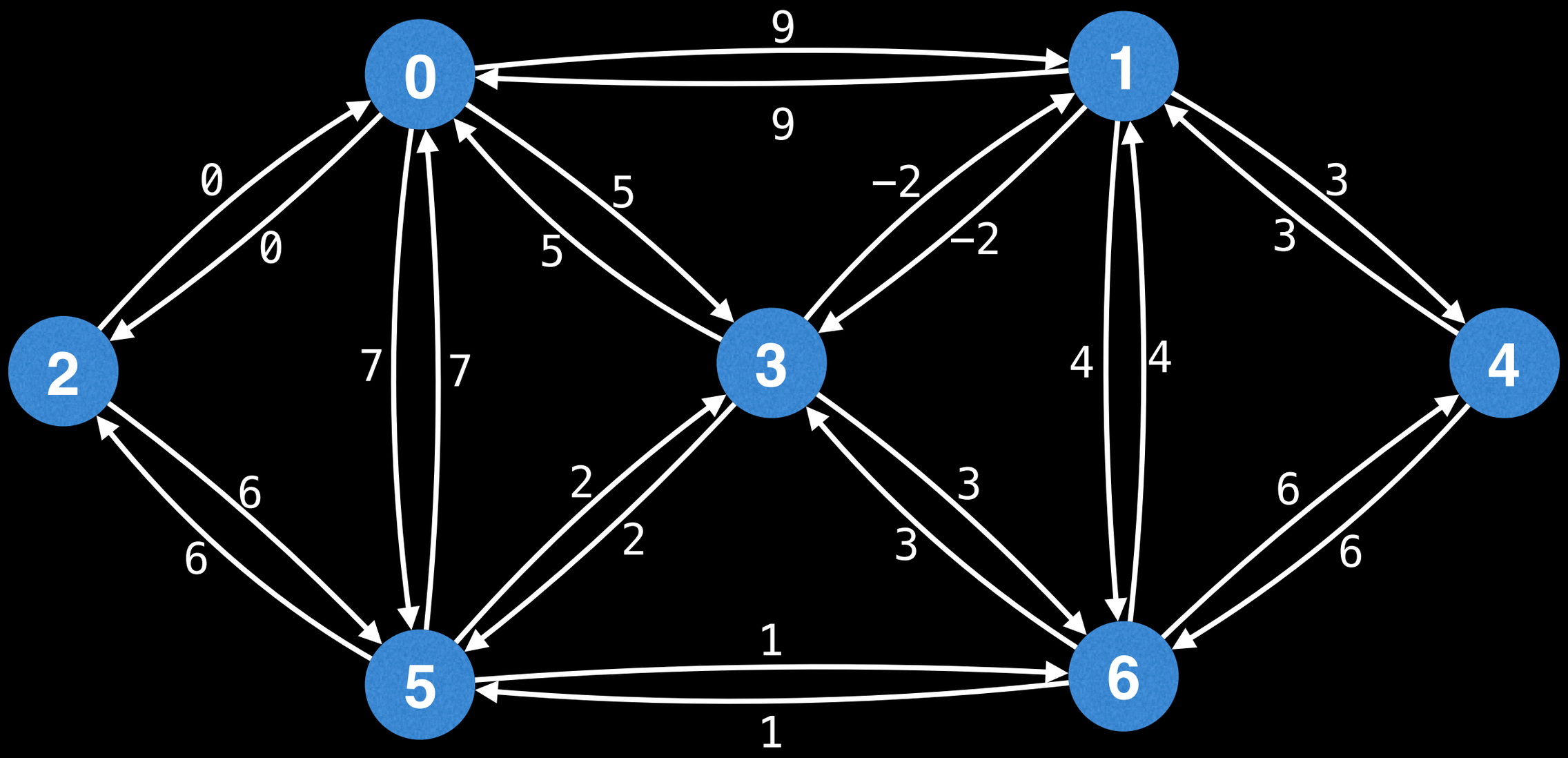
Eager Prim's

Original undirected graph.



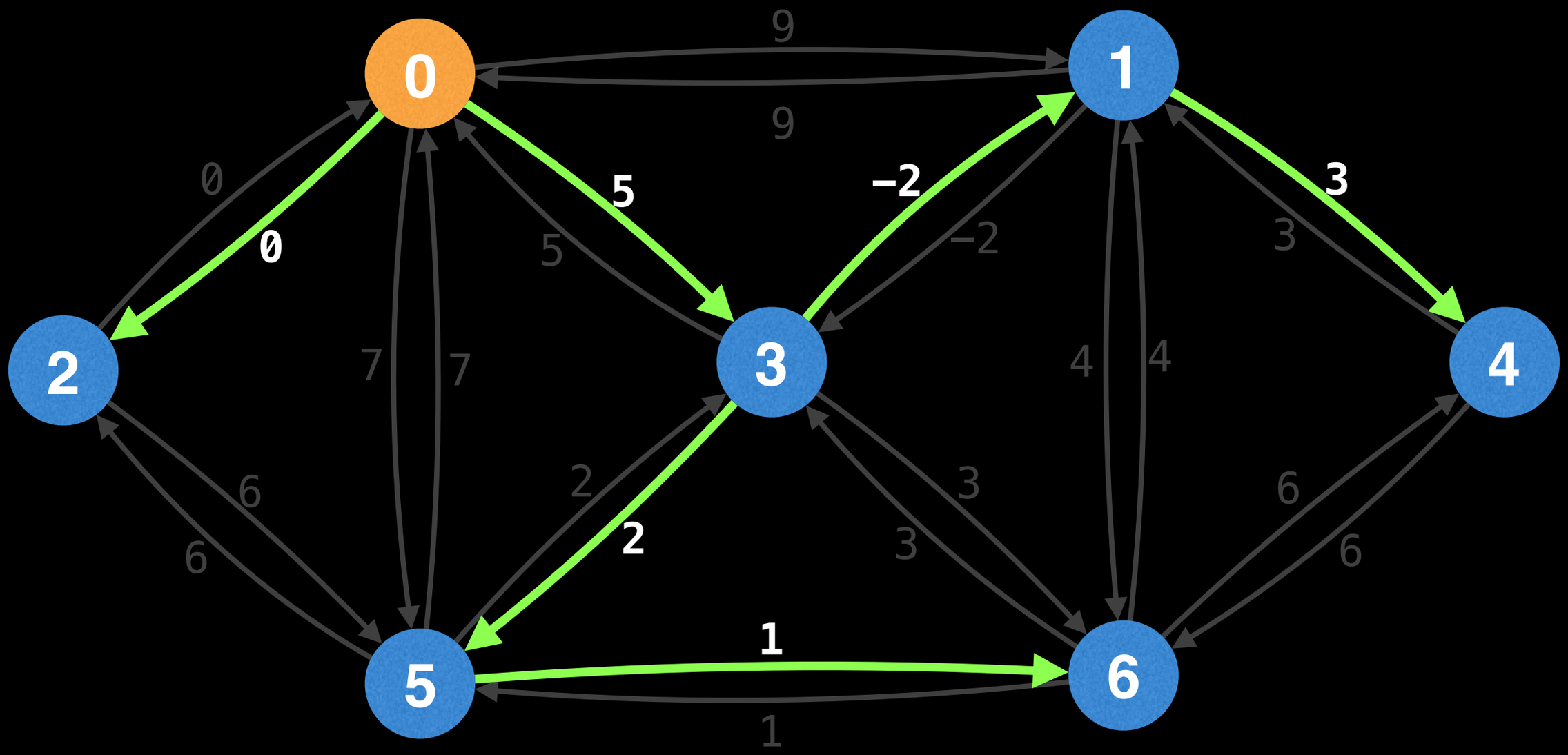
Eager Prim's

Equivalent directed version.



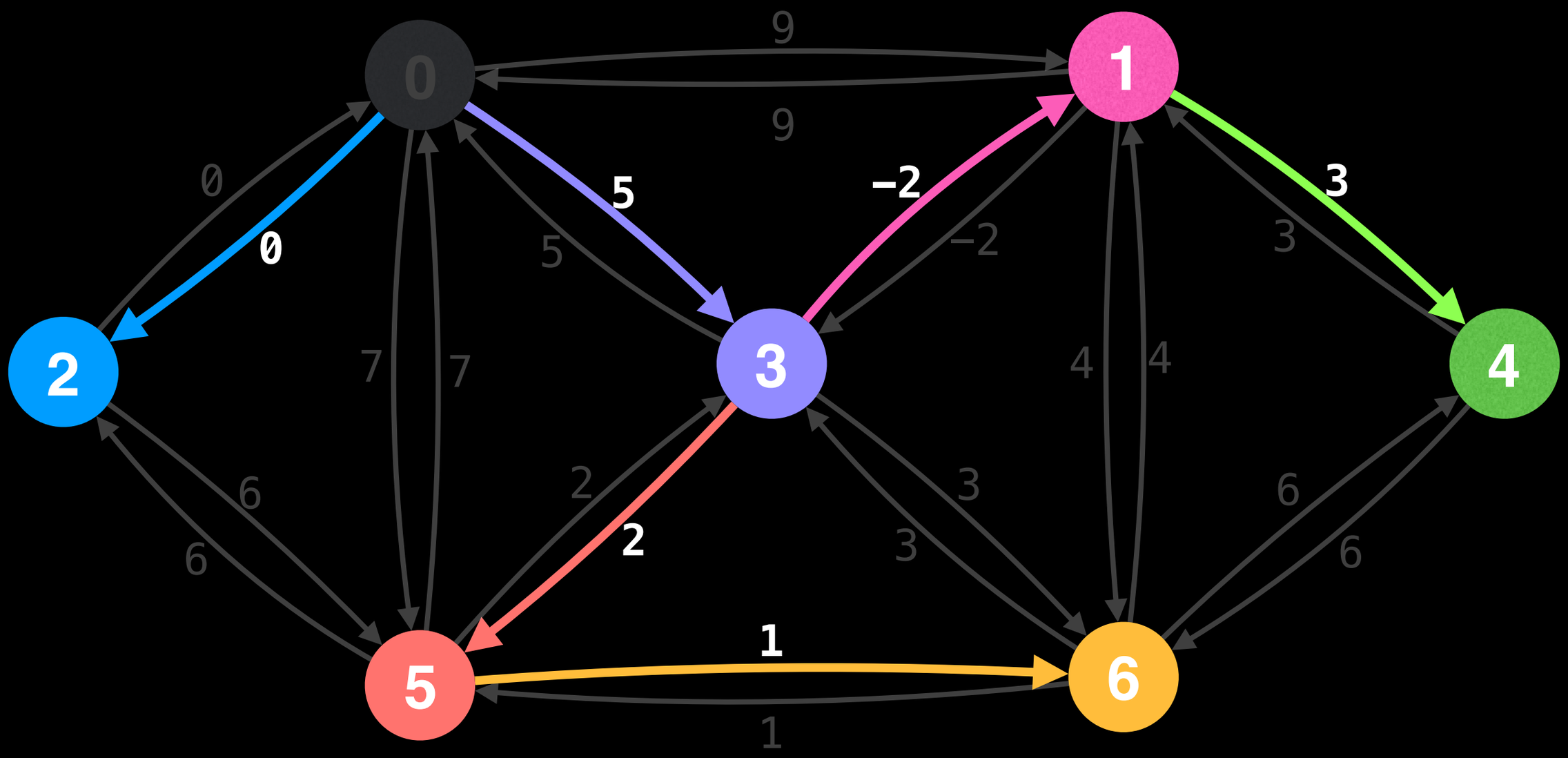
Eager Prim's

MST starting from node 0.



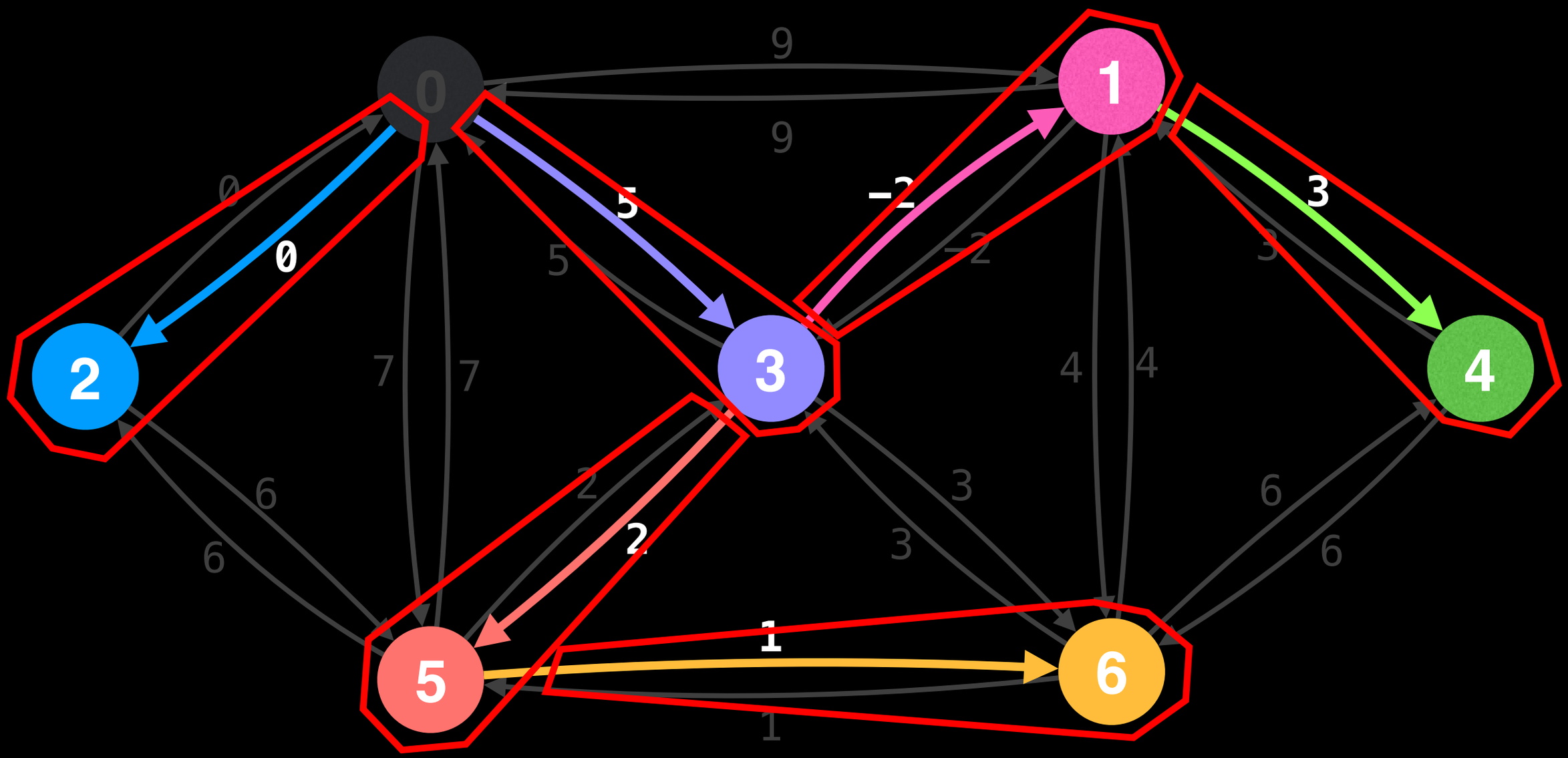
Eager Prim's

Looking at the directed MST, you can see that each node is paired with an incoming edge except for the starting node.



Eager Prim's

Looking at the directed MST, you can see that each node is paired with an incoming edge except for the starting node.



Eager Prim's

In the eager version of Prim's we are trying to determine which of a node's incoming edges we should select to include in the MST.

A slight difference from the lazy version is that instead of adding edges to the PQ as we iterate over the edges of node, we're going to **relax** (update) the destination node's most promising incoming edge.

Eager Prim's

A natural question to ask at this point is how are we going to efficiently update and retrieve these (node, edge) pairs?

One possible solution is to use an **Indexed Priority Queue (IPQ)** which can efficiently update and poll key-value pairs. This reduces the overall time complexity from **O(E*logE)** to **O(E*logV)** since there can only be V (node, edge) pairs in the IPQ, making the update and poll operations **O(logV)**.

Indexed Priority Queue DS Video

<insert video clip>

Link to IPQ video the description

Eager Prim's Algorithm

Maintain a **min Indexed Priority Queue (IPQ)** of size V that sorts vertex-edge pairs (v, e) based on the min edge cost of e. By default, all vertices v have a best value of (v, ∅) in the IPQ.

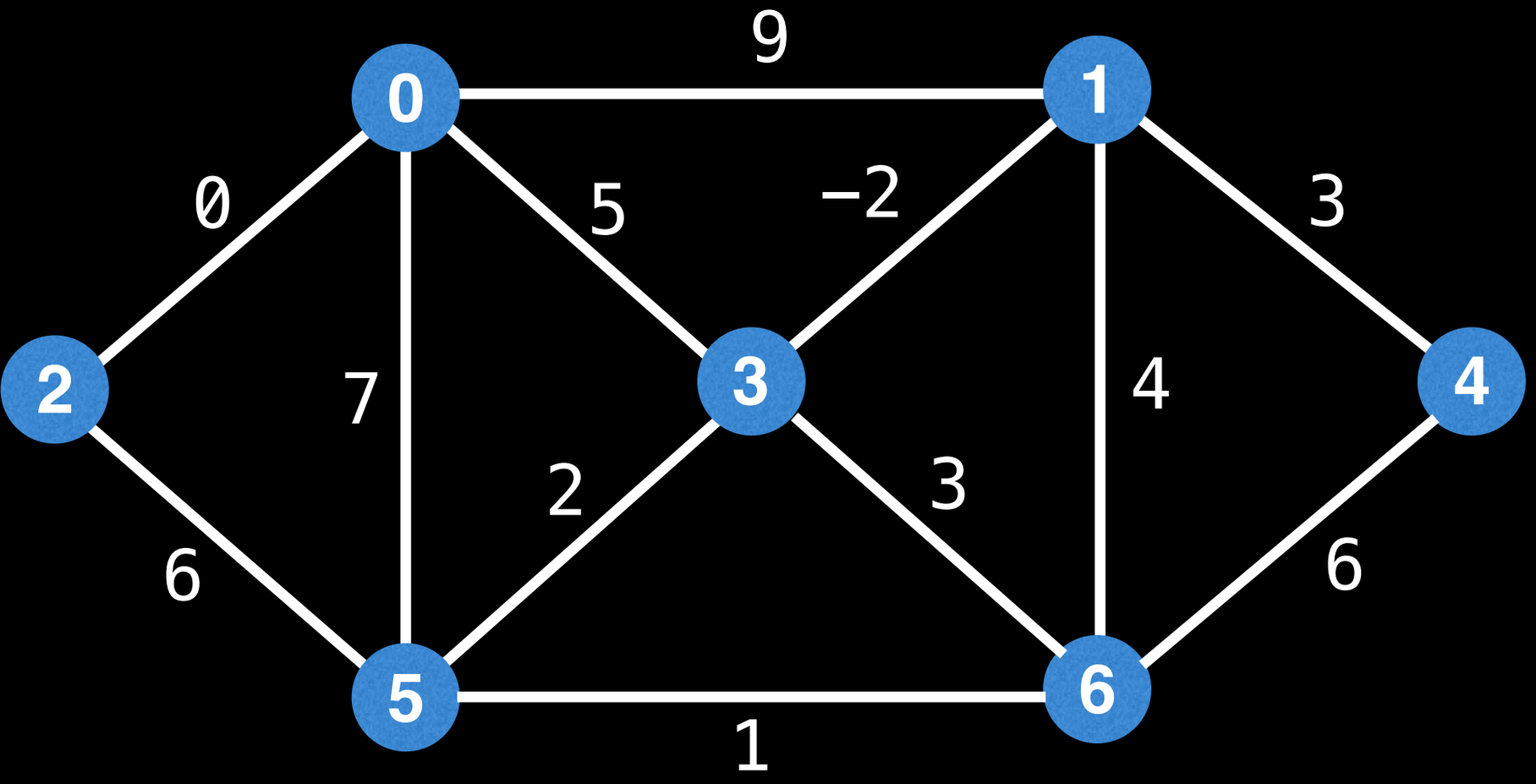
Start the algorithm on any node 's'. Mark **s** as visited and **relax** all edges of **s**.

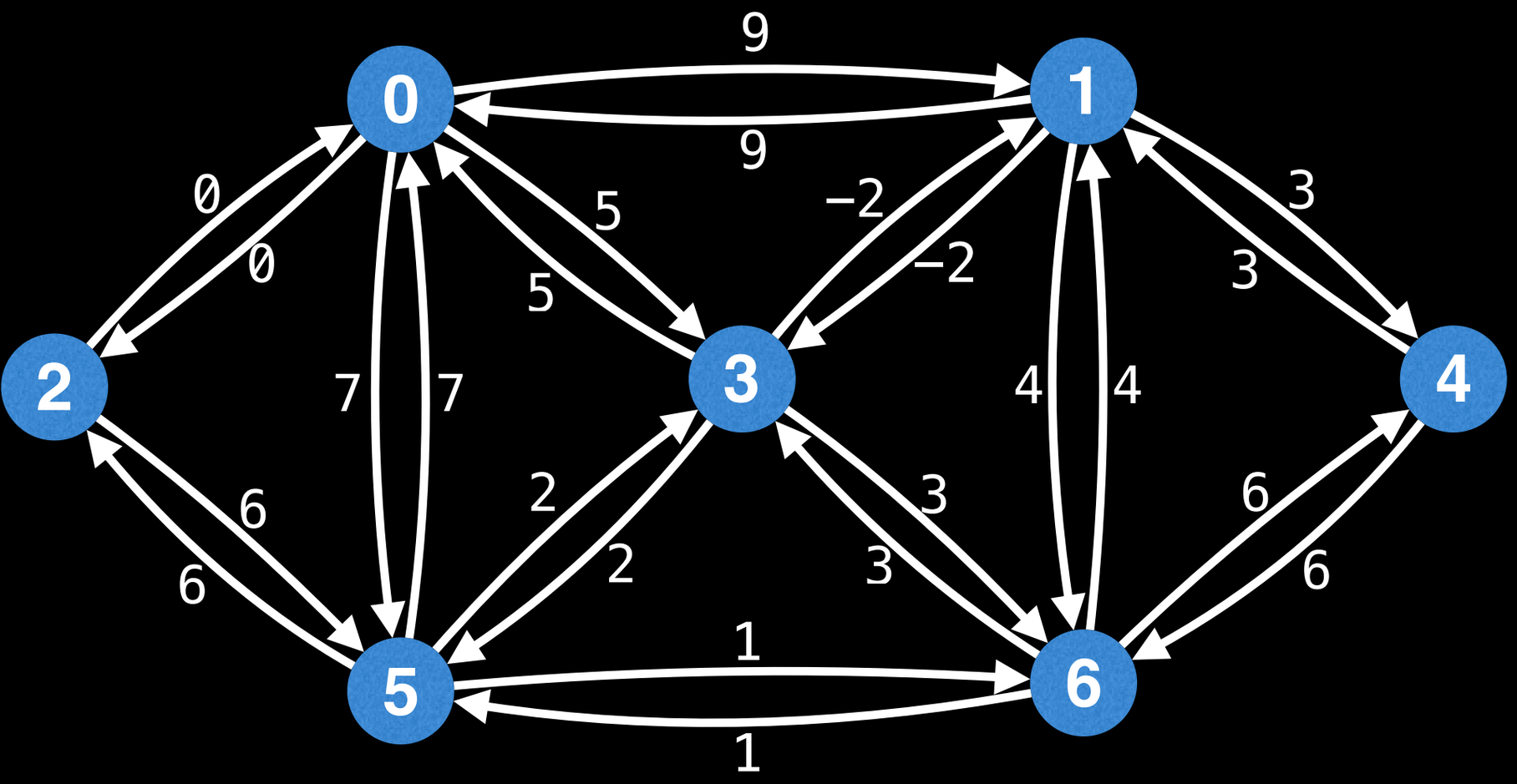
While the IPQ is not empty and a MST has not been formed, dequeue the next best (v, e) pair from the IPQ. Mark node v as visited and add edge e to the MST.

Next, relax all edges of v while making sure not to relax any edge pointing to a node which has already been visited.

---

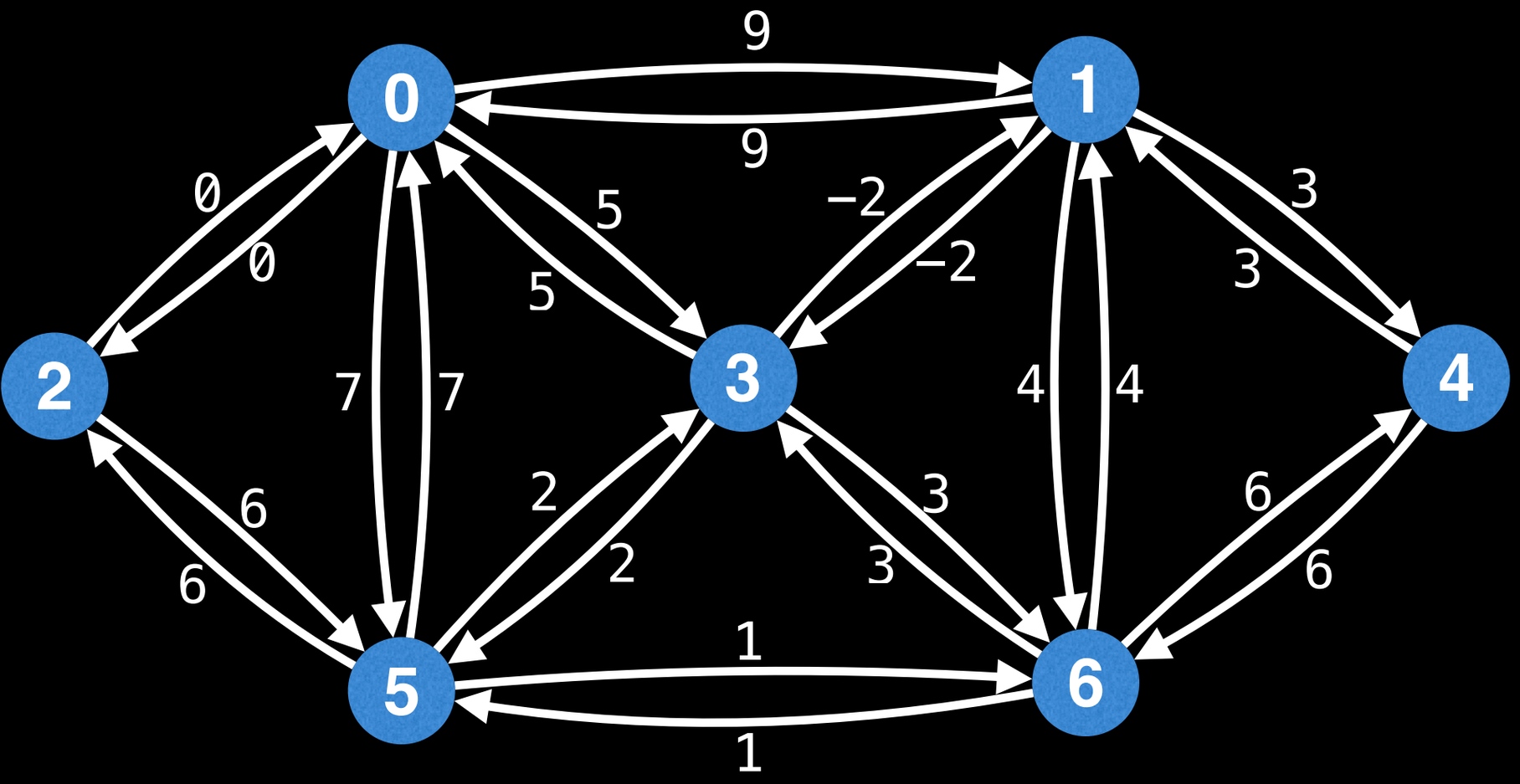
relaxing in this context refers to updating the entry for node v in the IPQ from (v, e_{old}) to (v, e_{new}) if the new edge e_{new} from u -> v has a lower cost than e_{old}.


9
0
1
0
5
−2
3
2
7
3
4
4
2
3
6
6
5
1
6

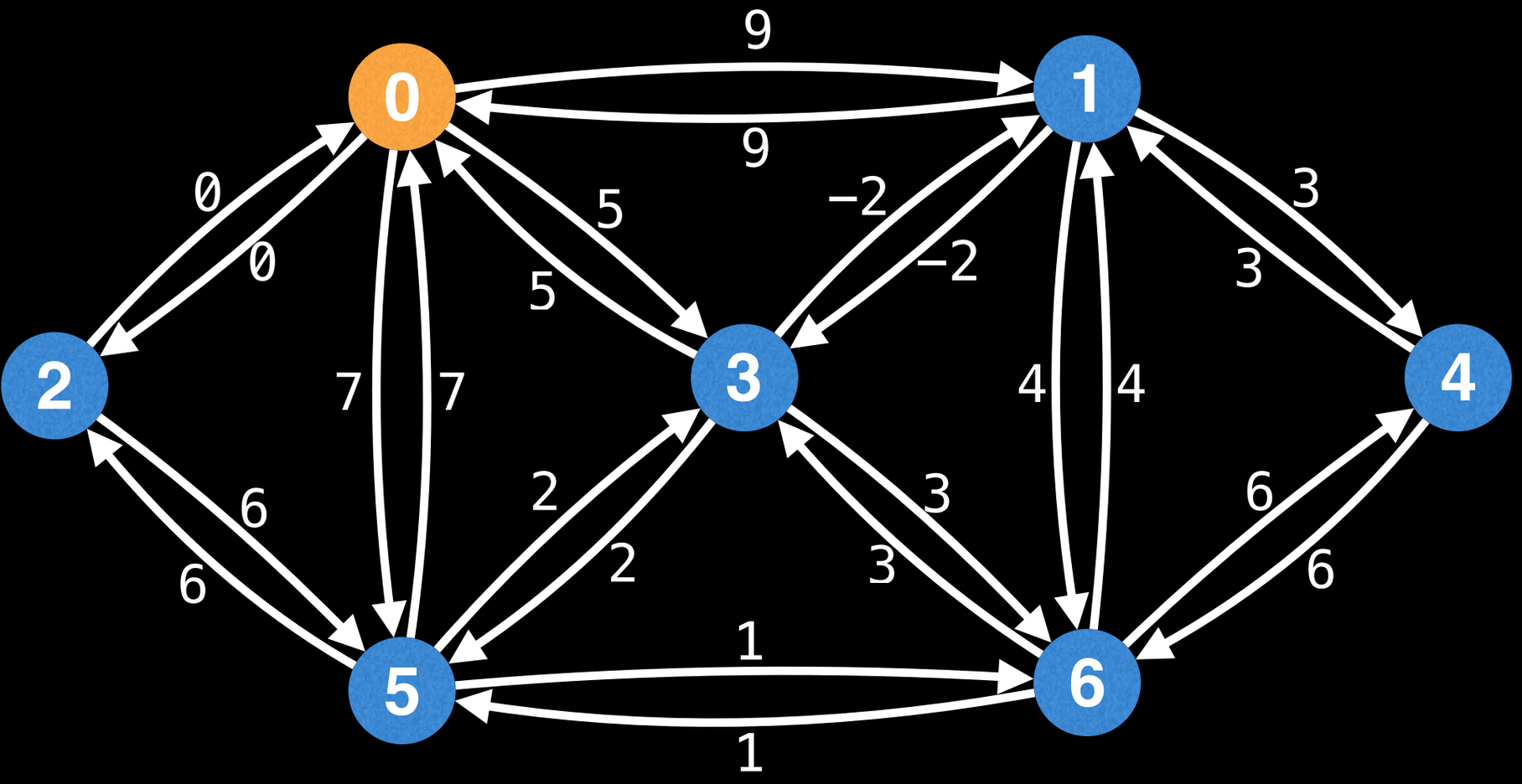


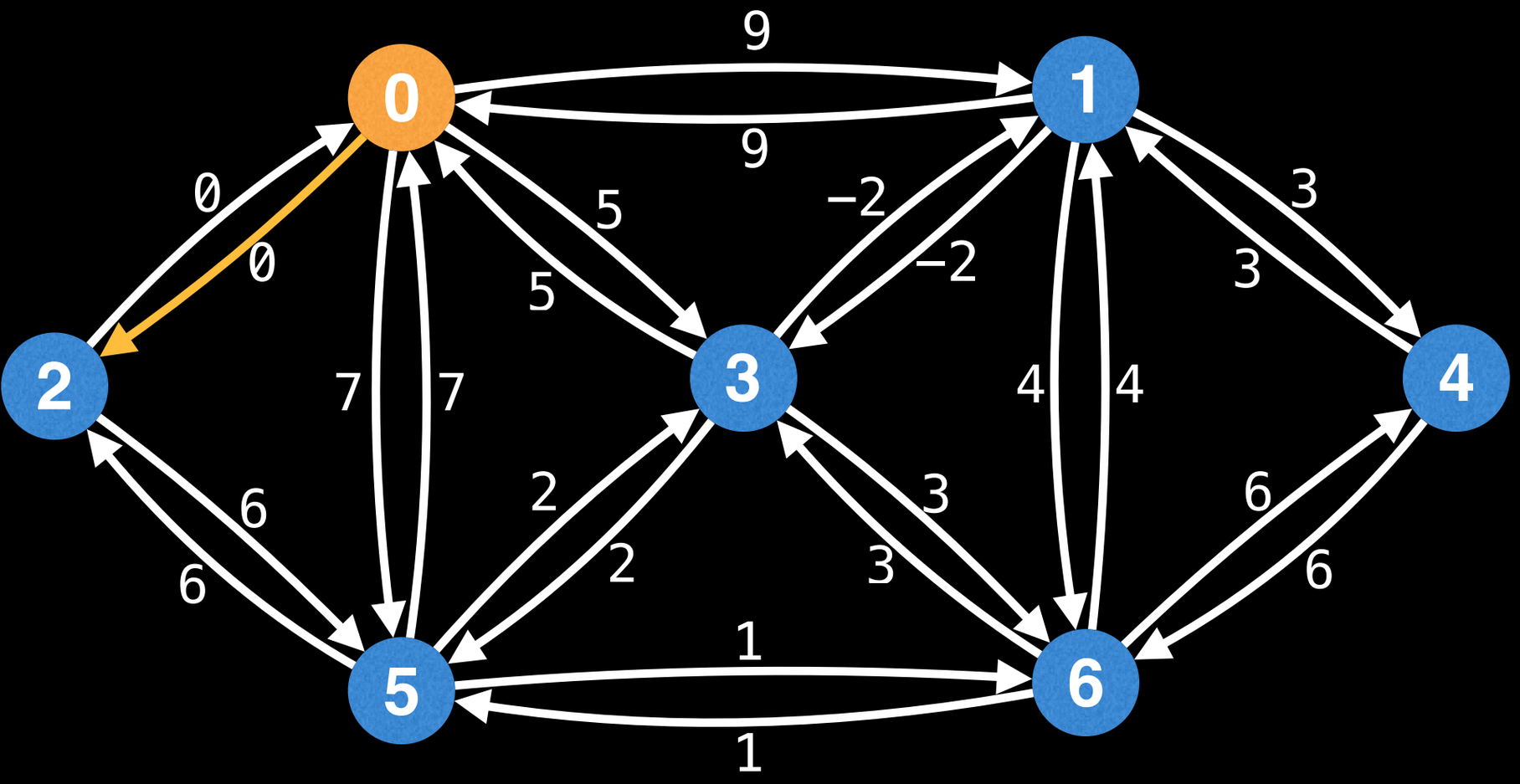
One thing to bear in mind is that while the graph above represents an **undirected graph**, the internal adjacency list representation typically has each undirected edge stored as **two directed edges**.

(node, edge) key-value pairs in IPQ



(node, edge) key-value pairs in IPQ





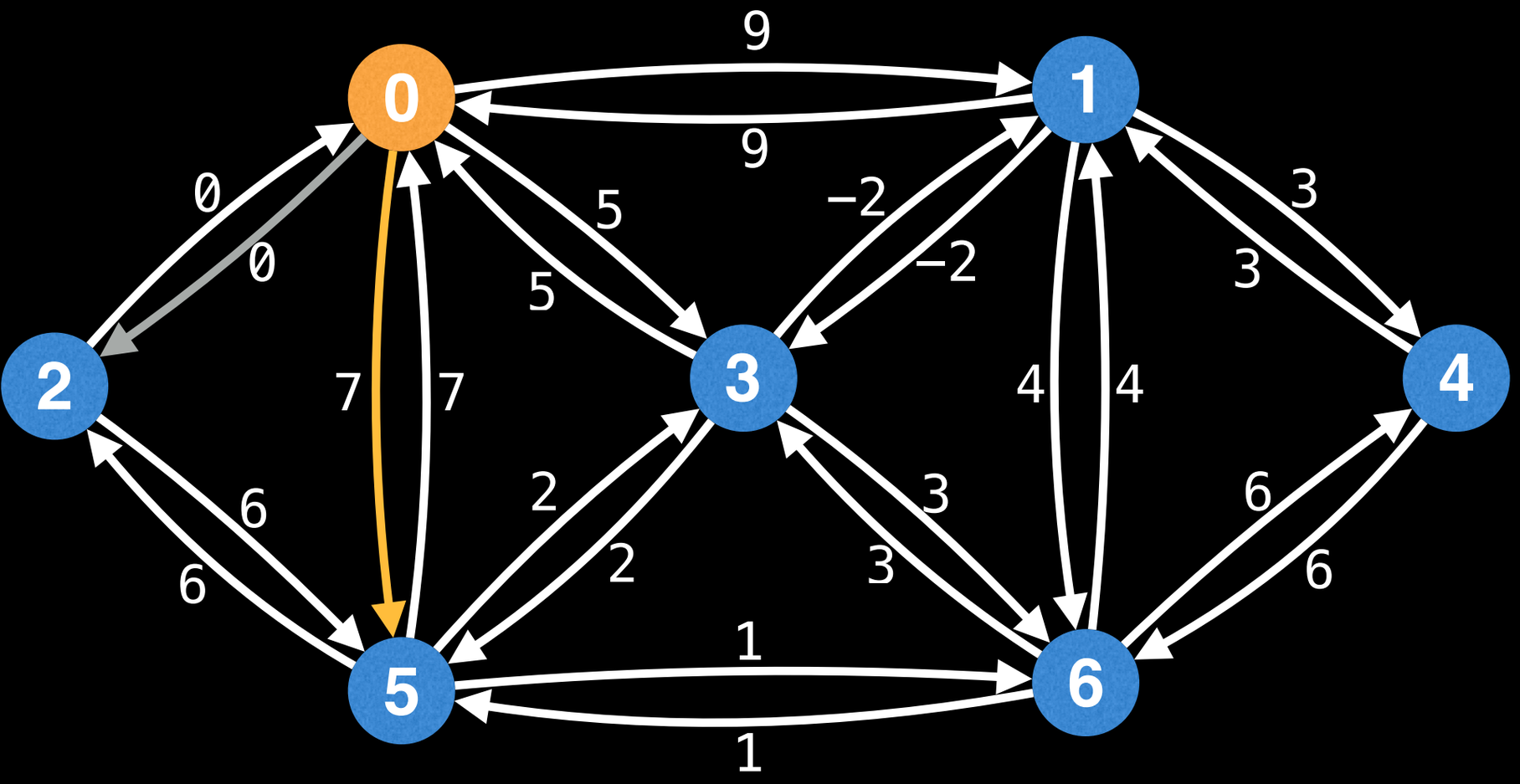
(node, edge) key-value pairs in IPQ

2 -> (0, 2, 0)

node index -> (start node, end node, edge cost)

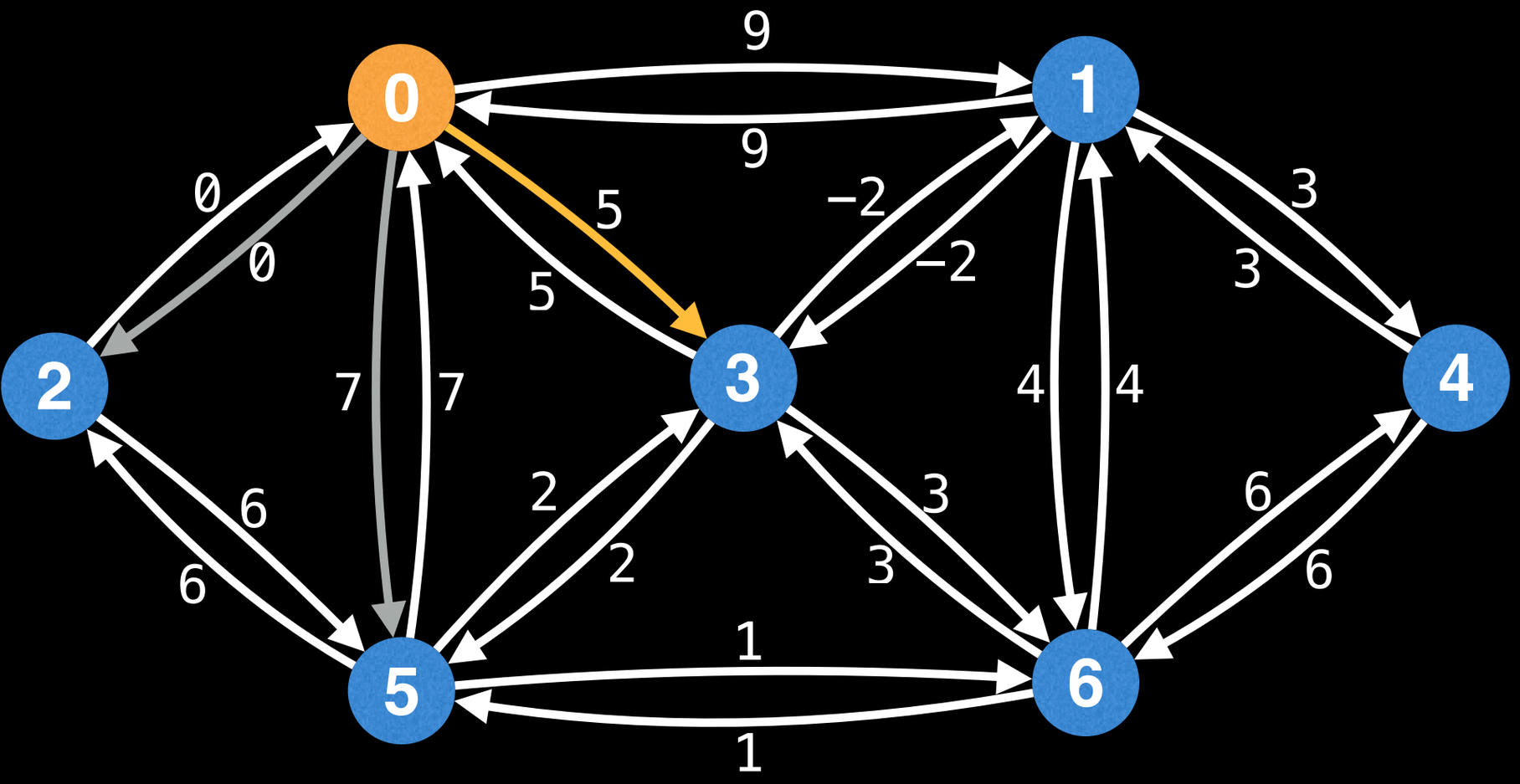
(node, edge) key-value pairs in IPQ

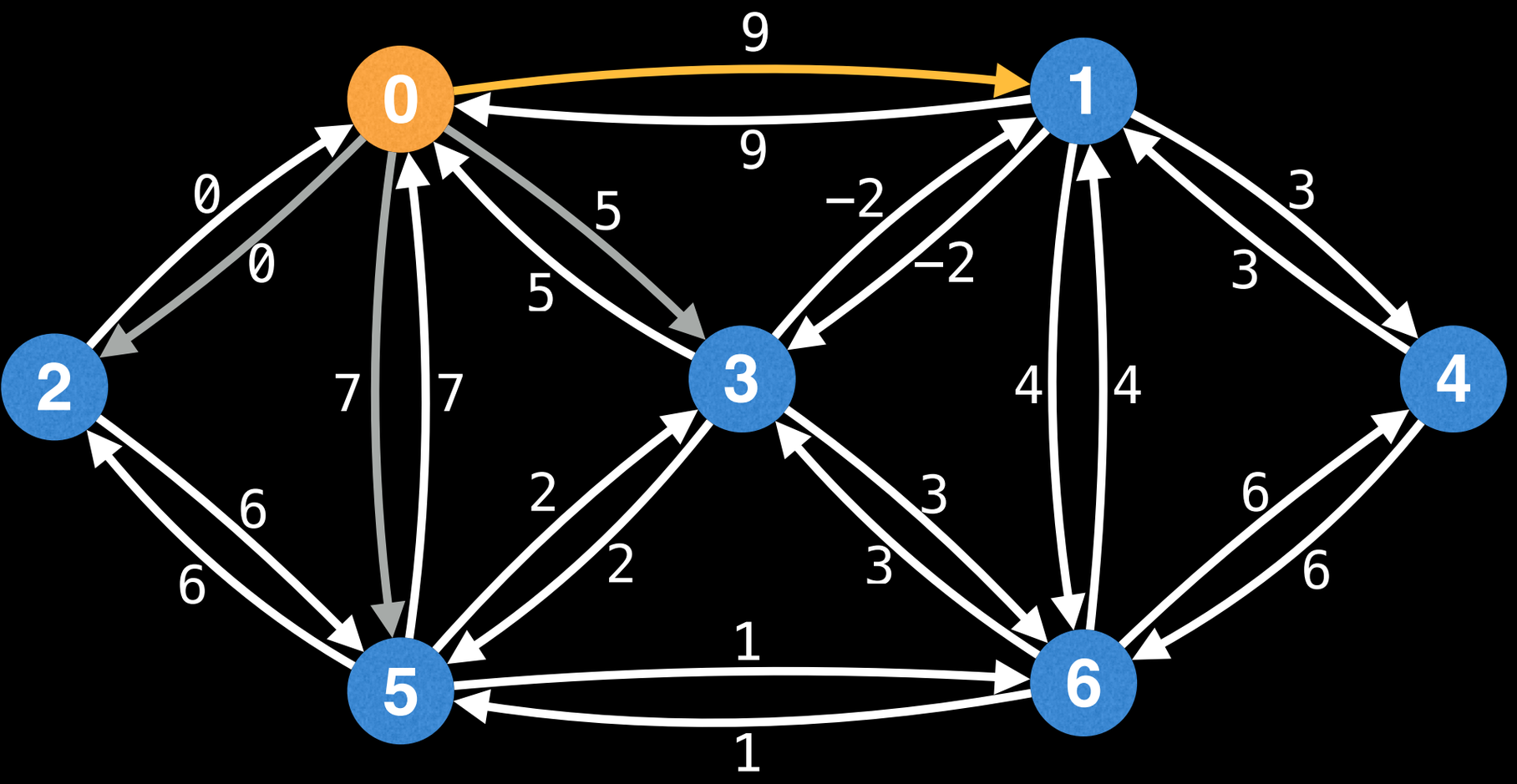
2 -> (0, 2, 0)
5 -> (0, 5, 7)



(node, edge) key-value pairs in IPQ

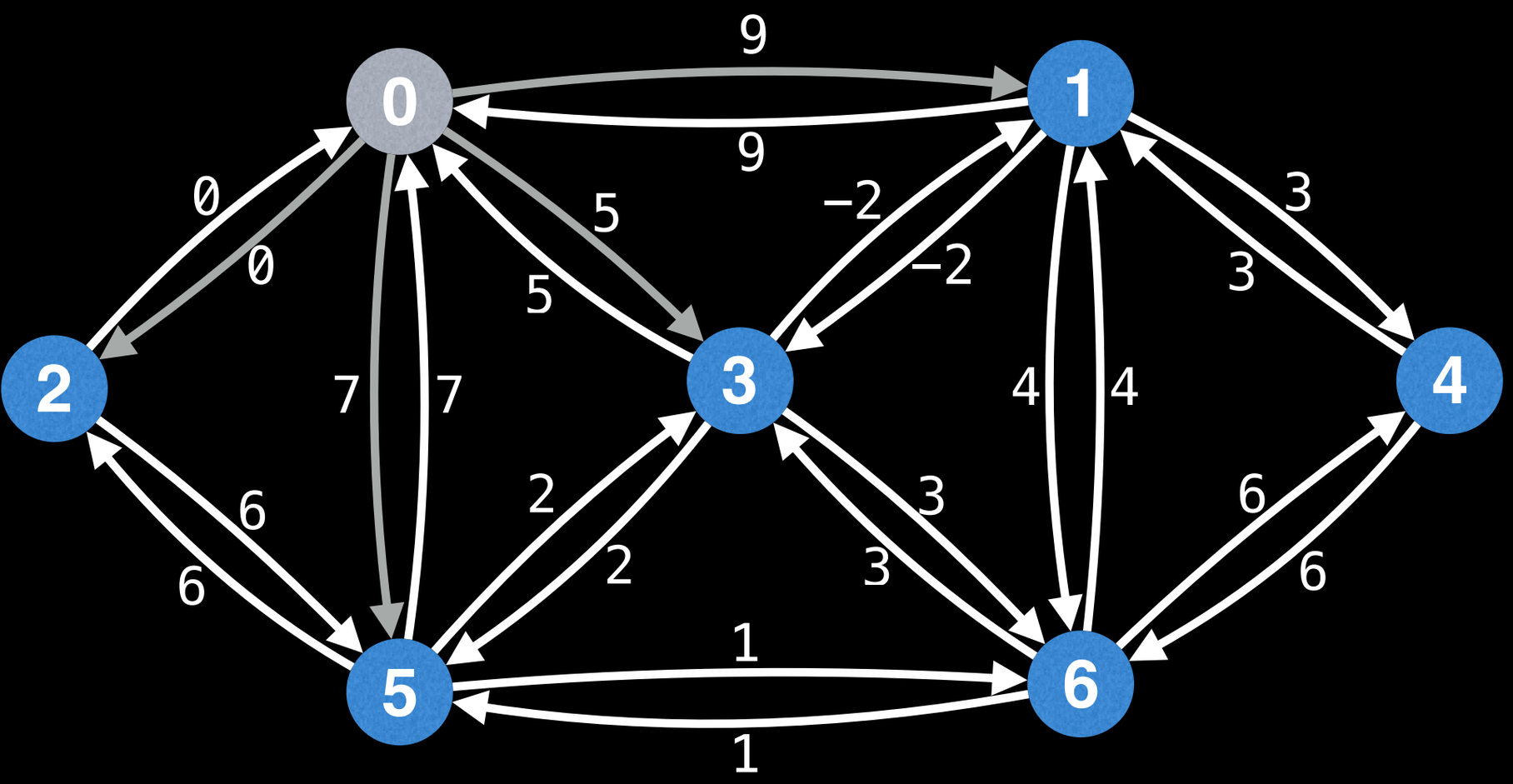
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)





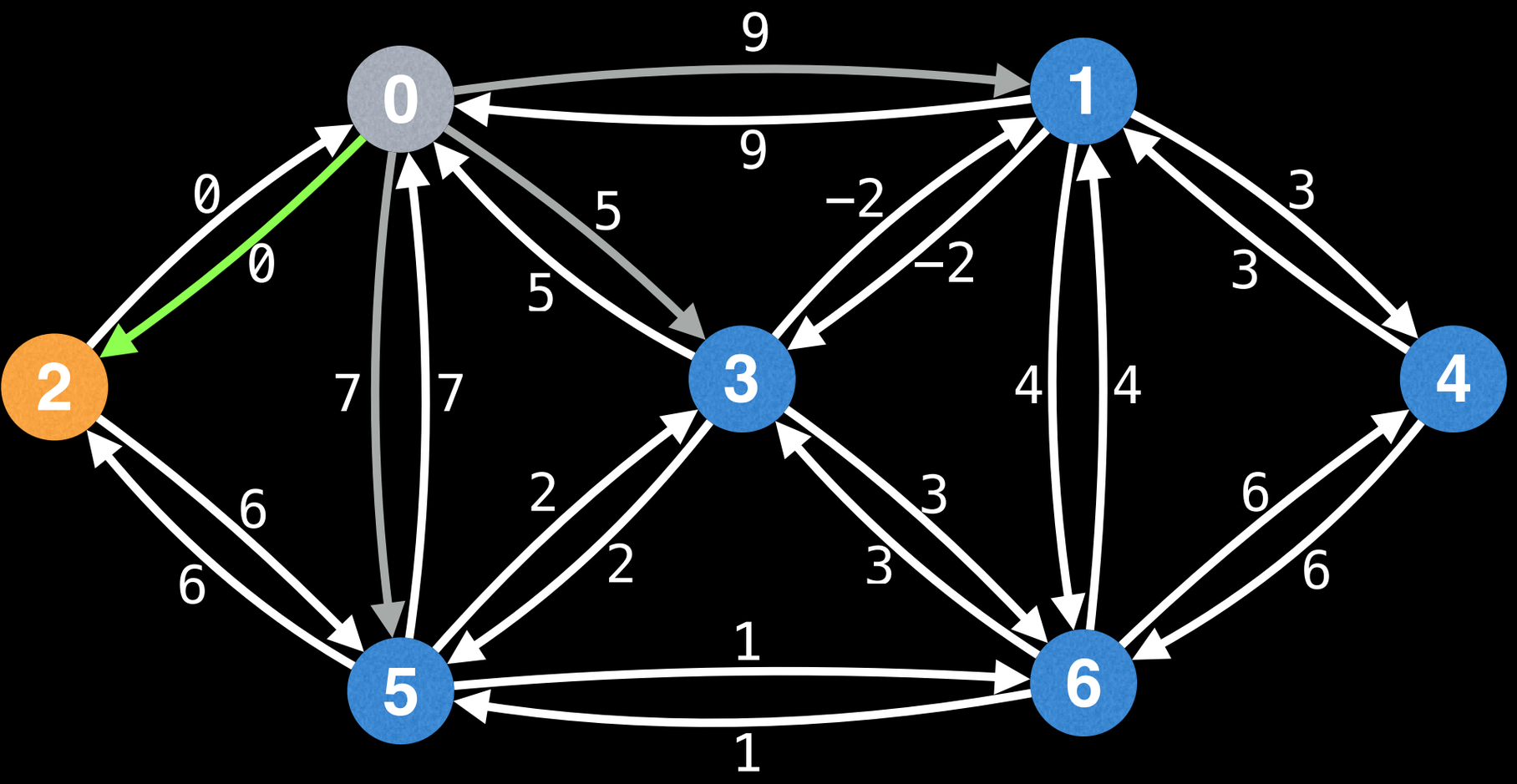
(node, edge) key-value pairs in IPQ

```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)
```



(node, edge) key-value pairs in IPQ

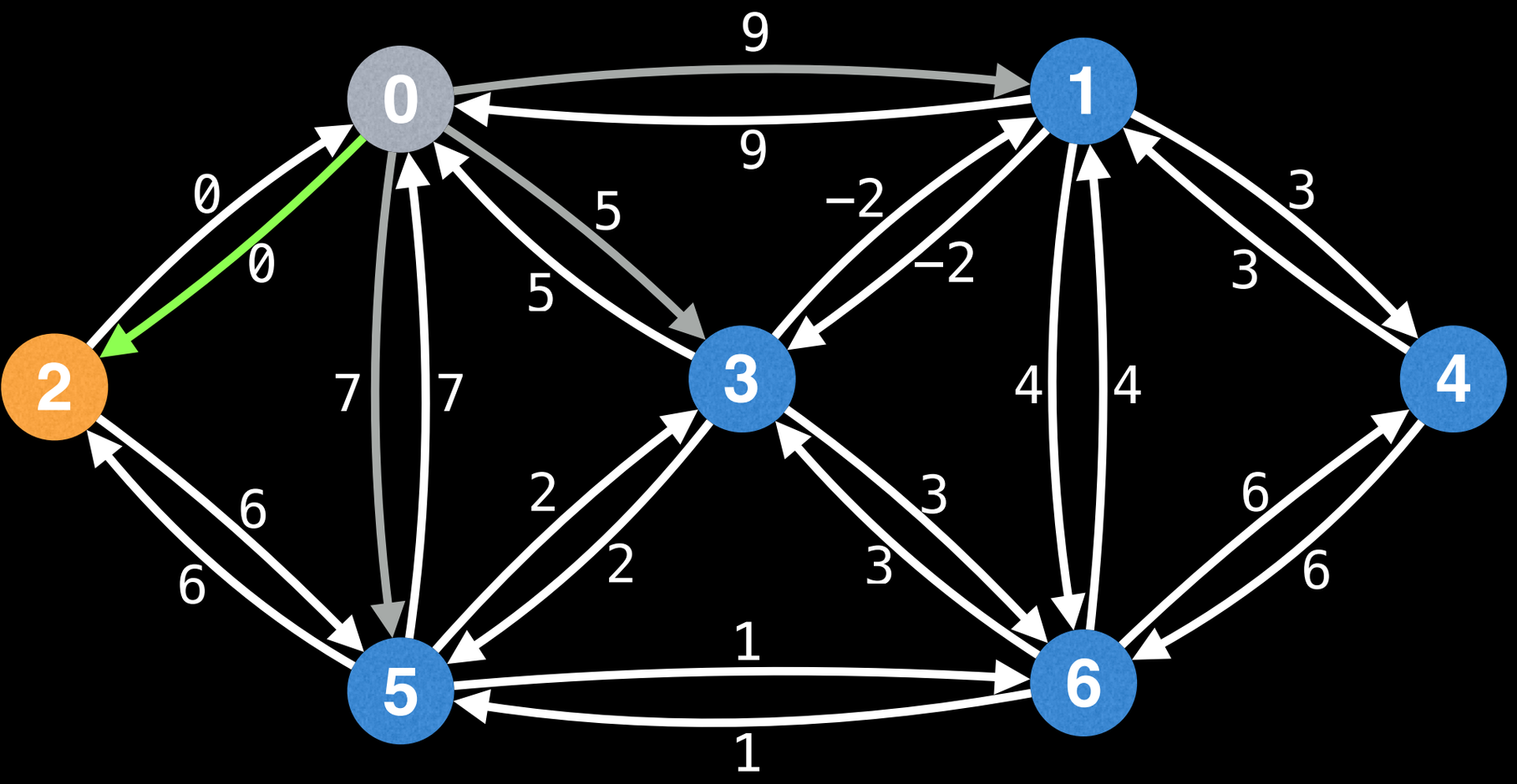
```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)
```



(node, edge) key-value pairs in IPQ

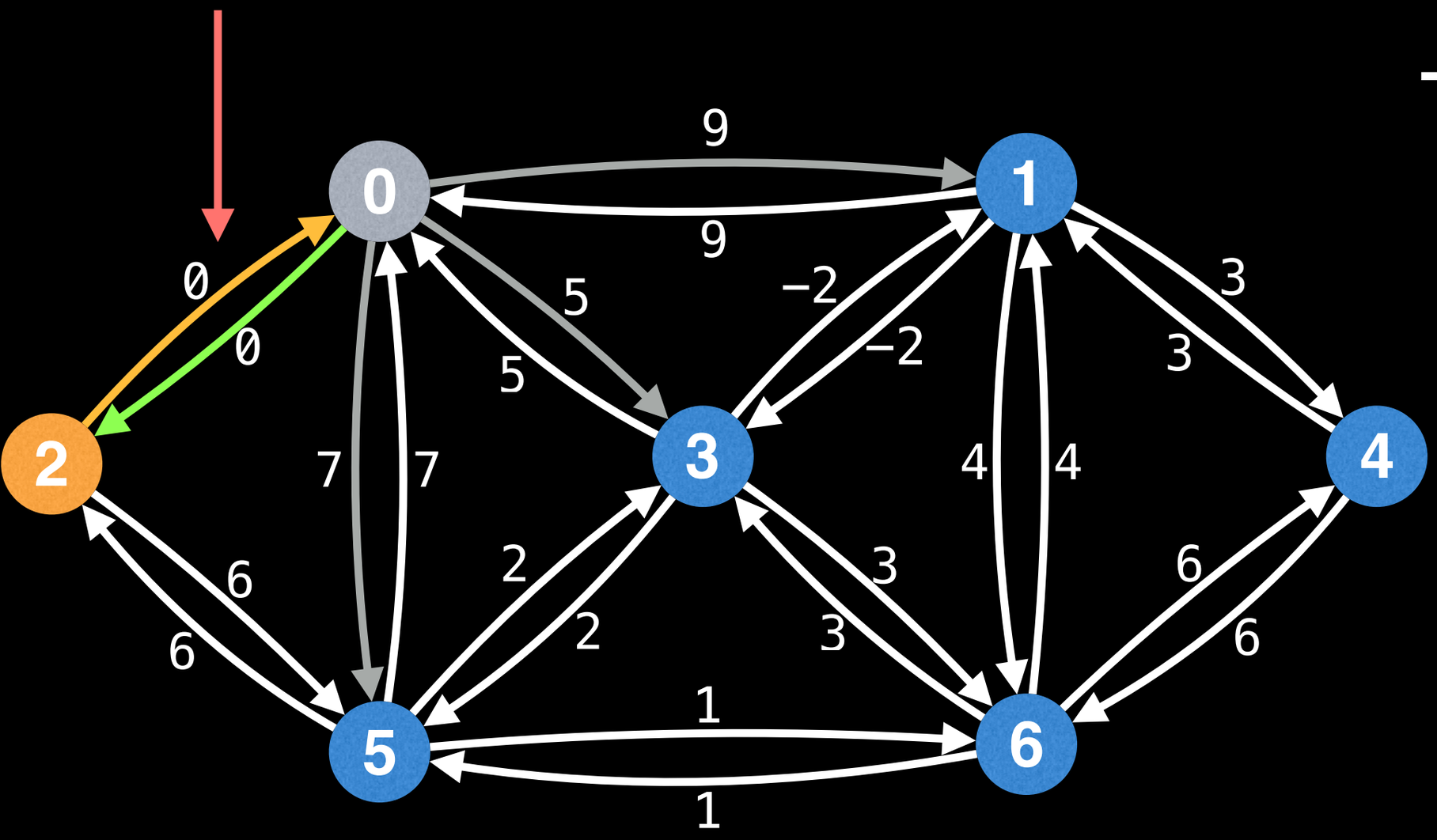
```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)
```

The next cheapest (node, edge) pair is 2 -> (0, 2, 0).



(node, edge) key-value pairs in IPQ

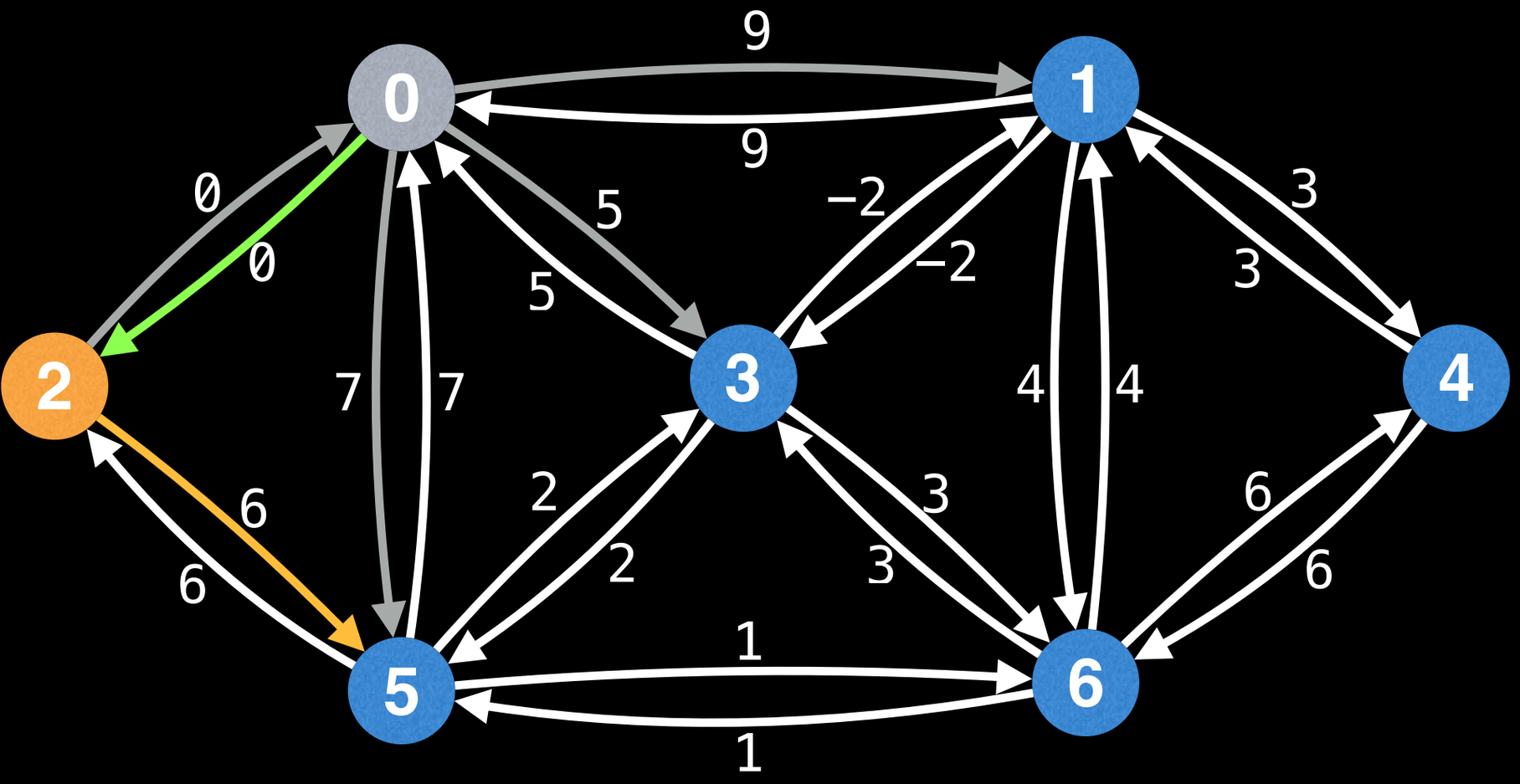
~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)



(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

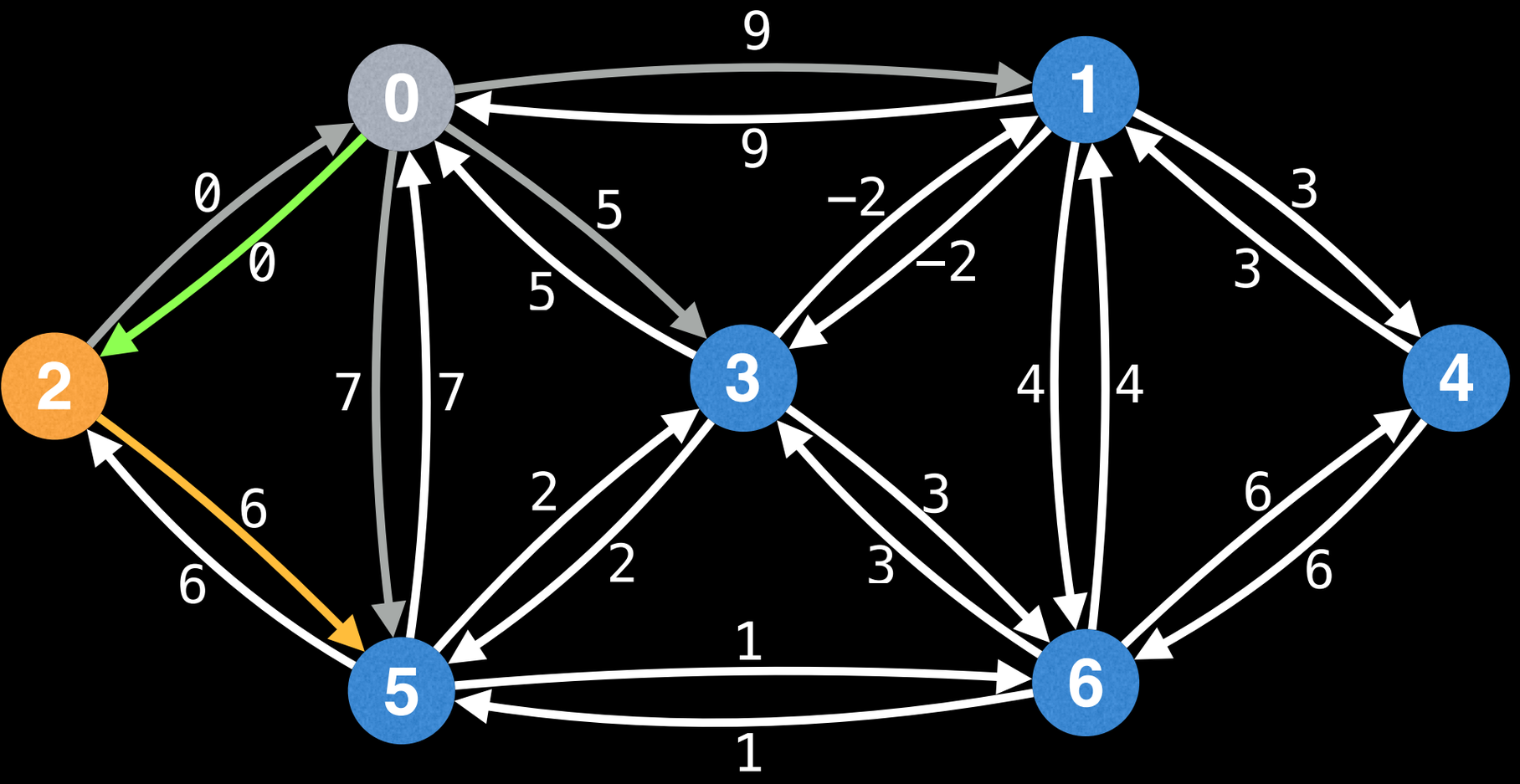
Ignore edges pointing to already-visited nodes.



(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

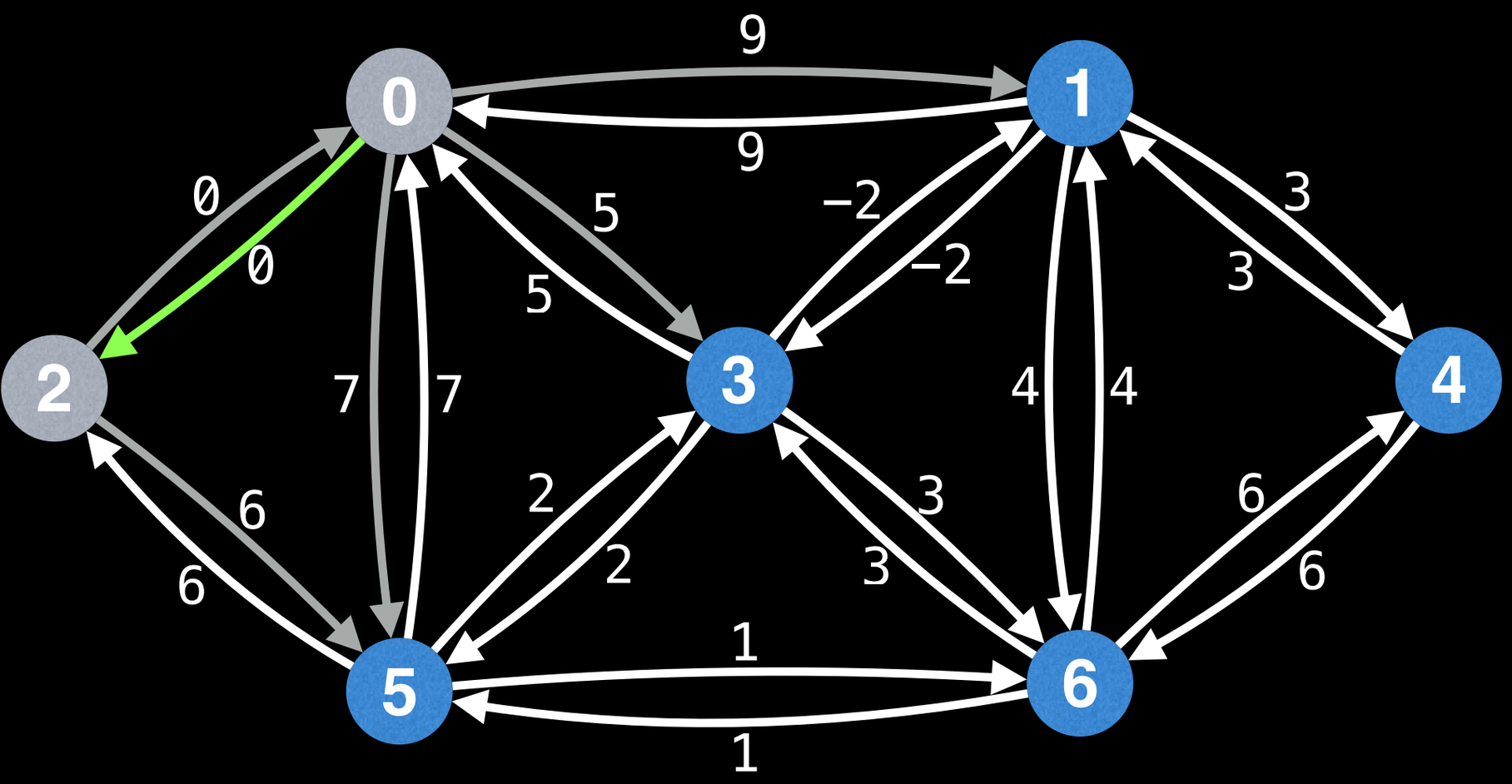
Edge (2, 5, 6) has a lower cost going to node 5 so update the IPQ with the new edge.



(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

Edge (2, 5, 6) has a lower cost going to node 5 so update the IPQ with the new edge.

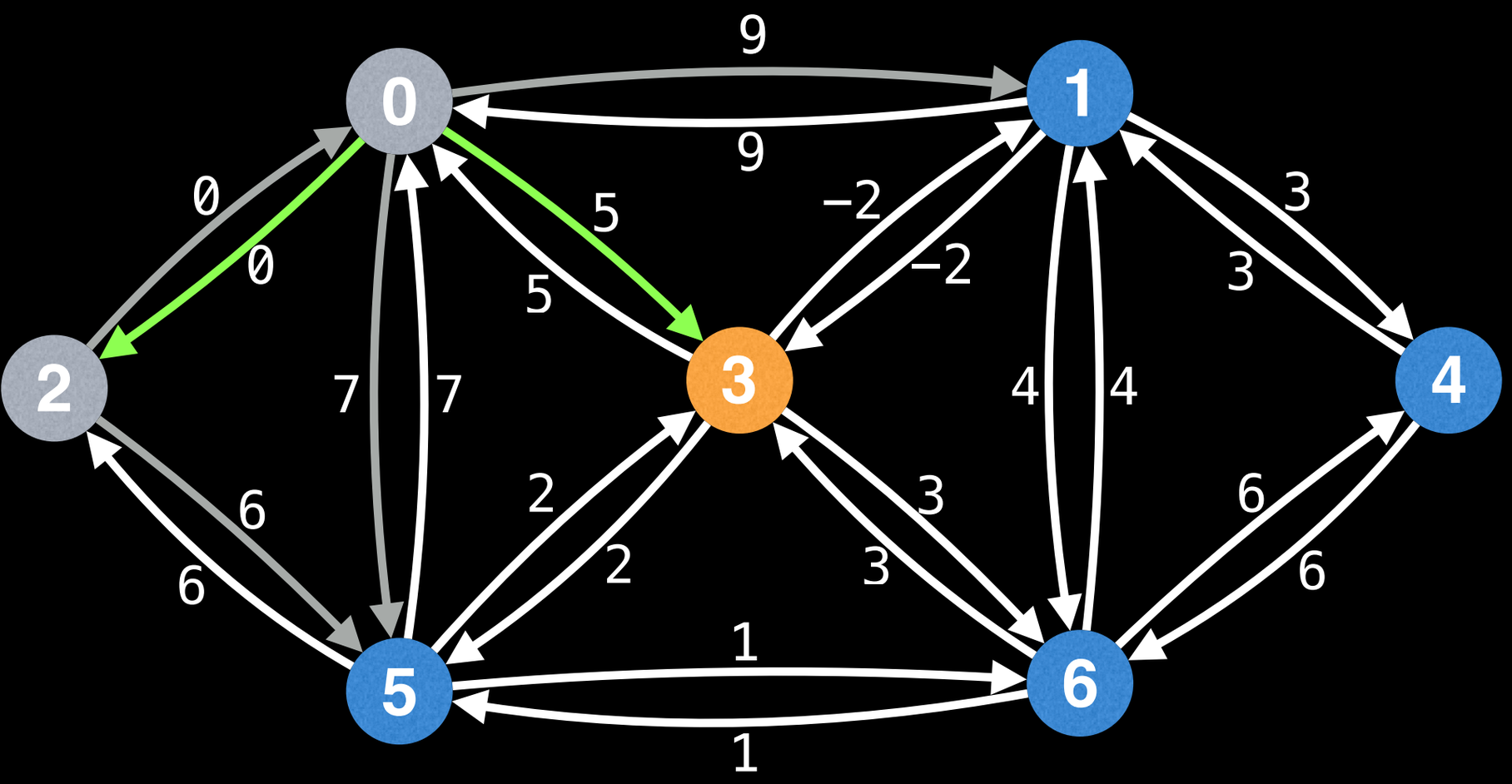


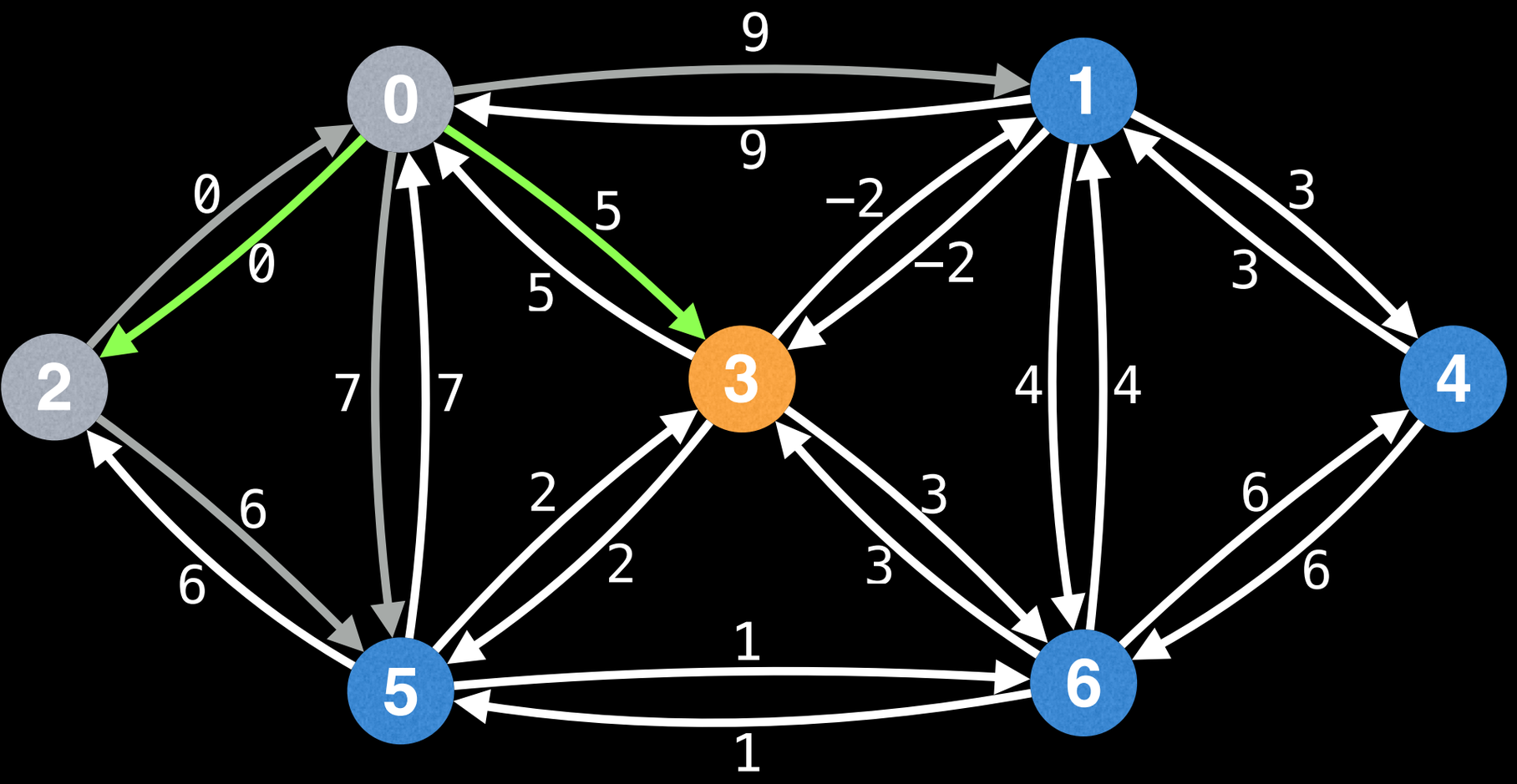
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)



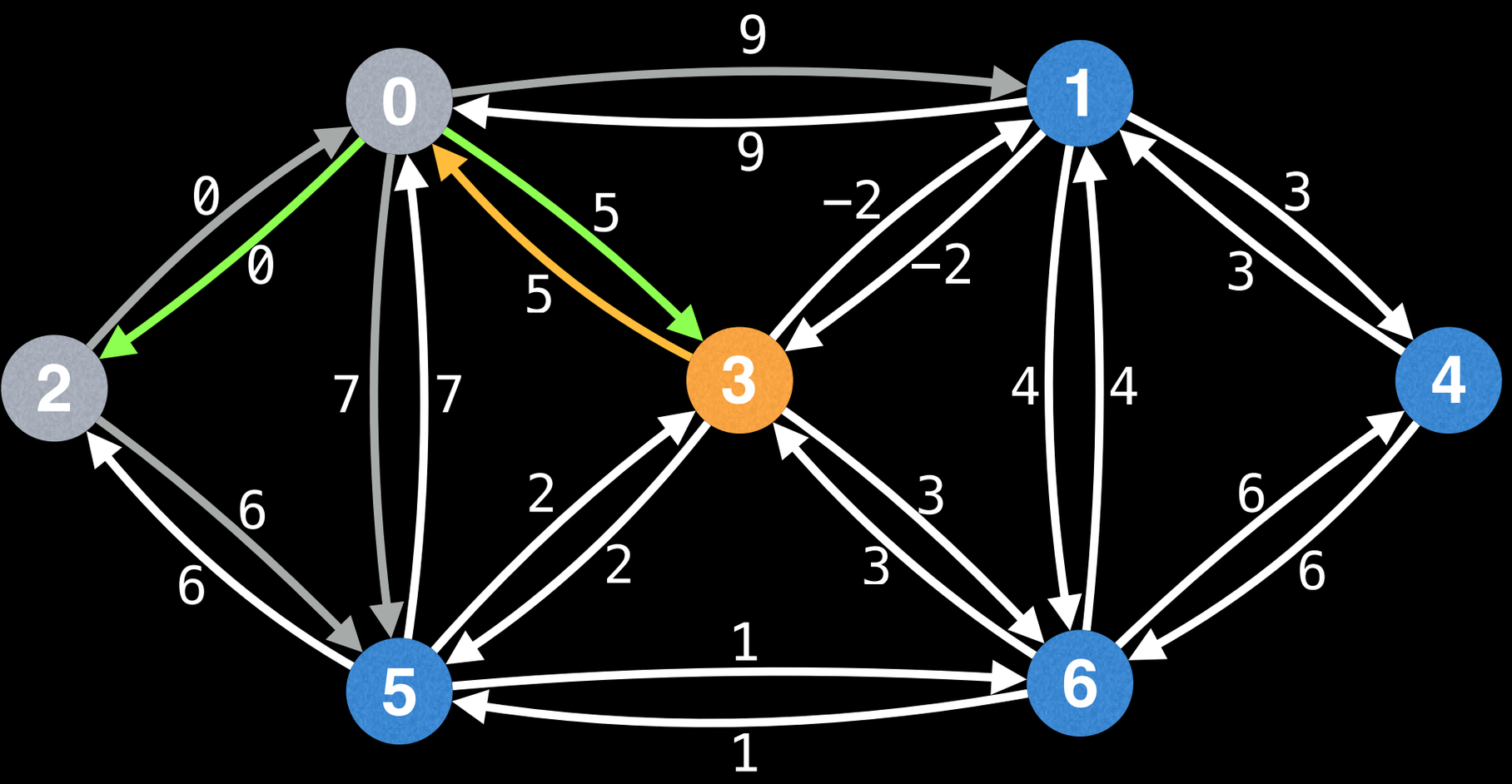


(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (0, 1, 9)

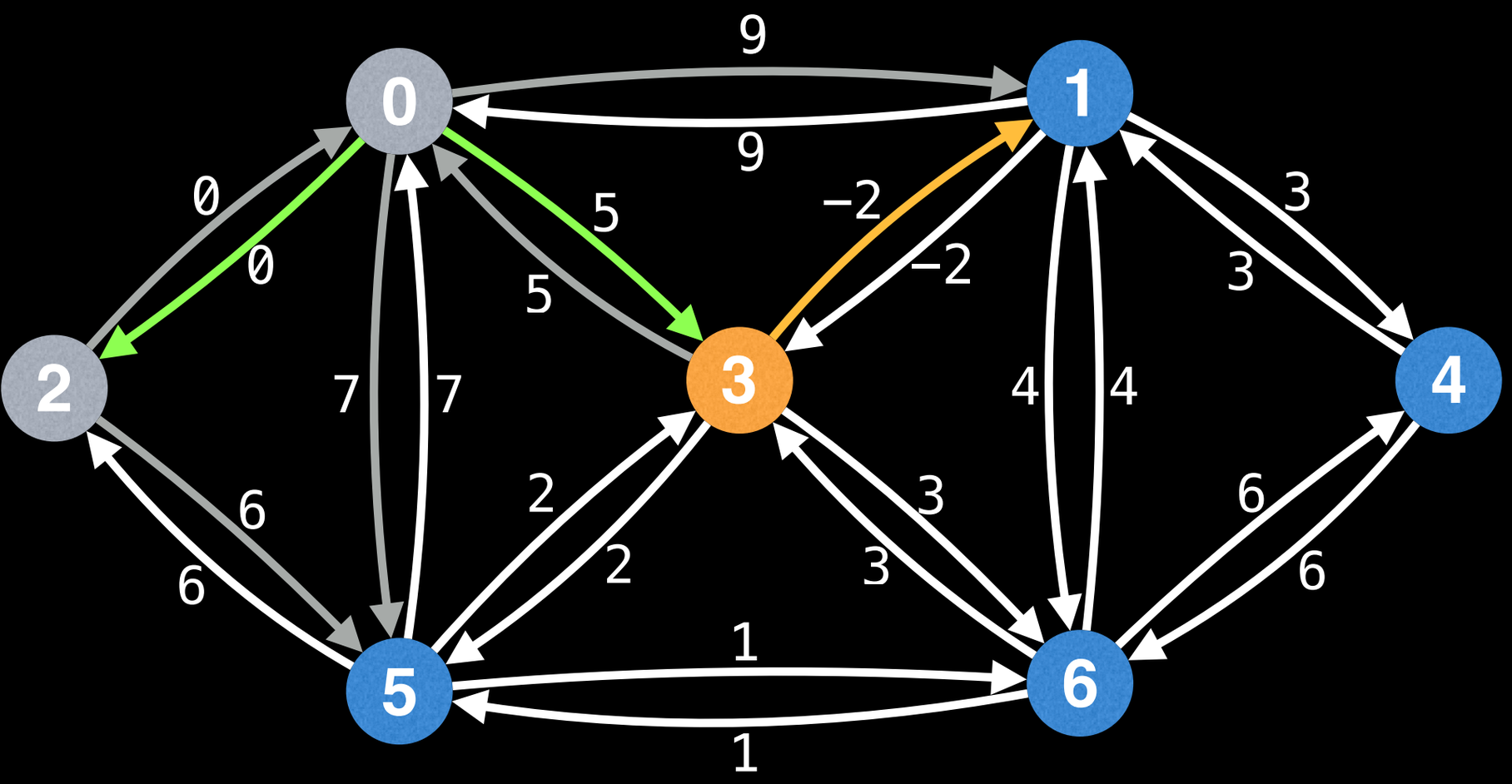
(node, edge) key-value pairs in IPQ

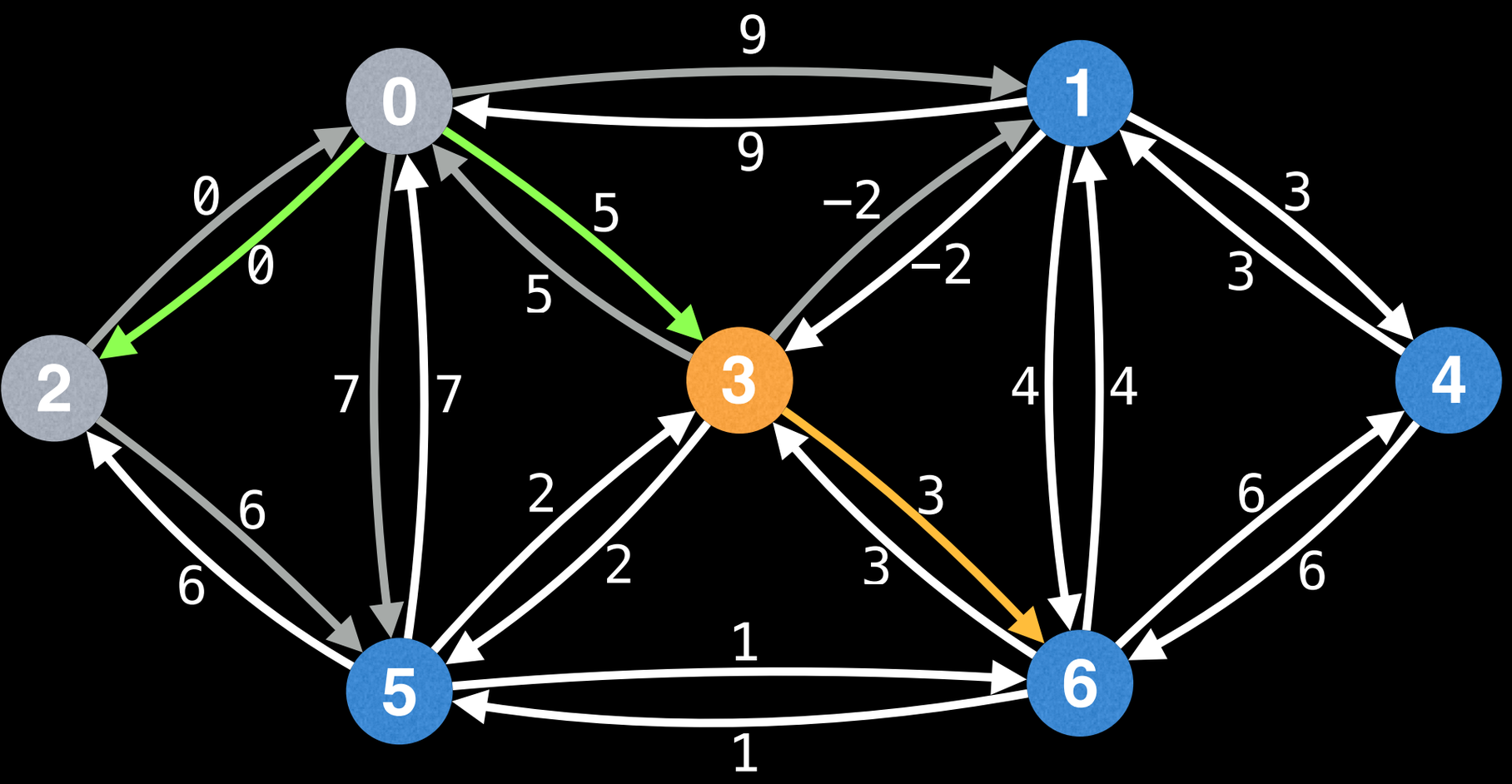
~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (0, 1, 9)



(node, edge) key-value pairs in IPQ

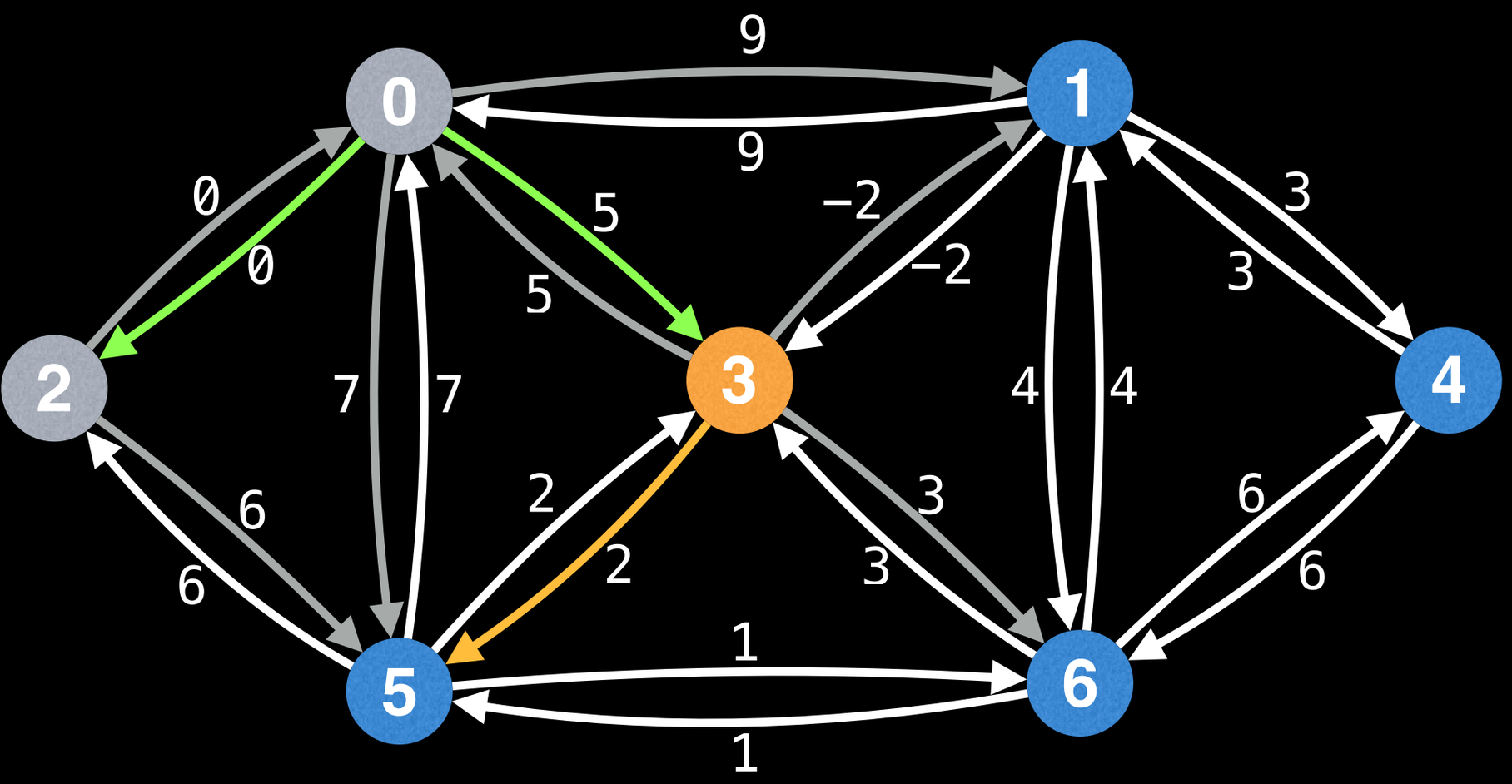
~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)





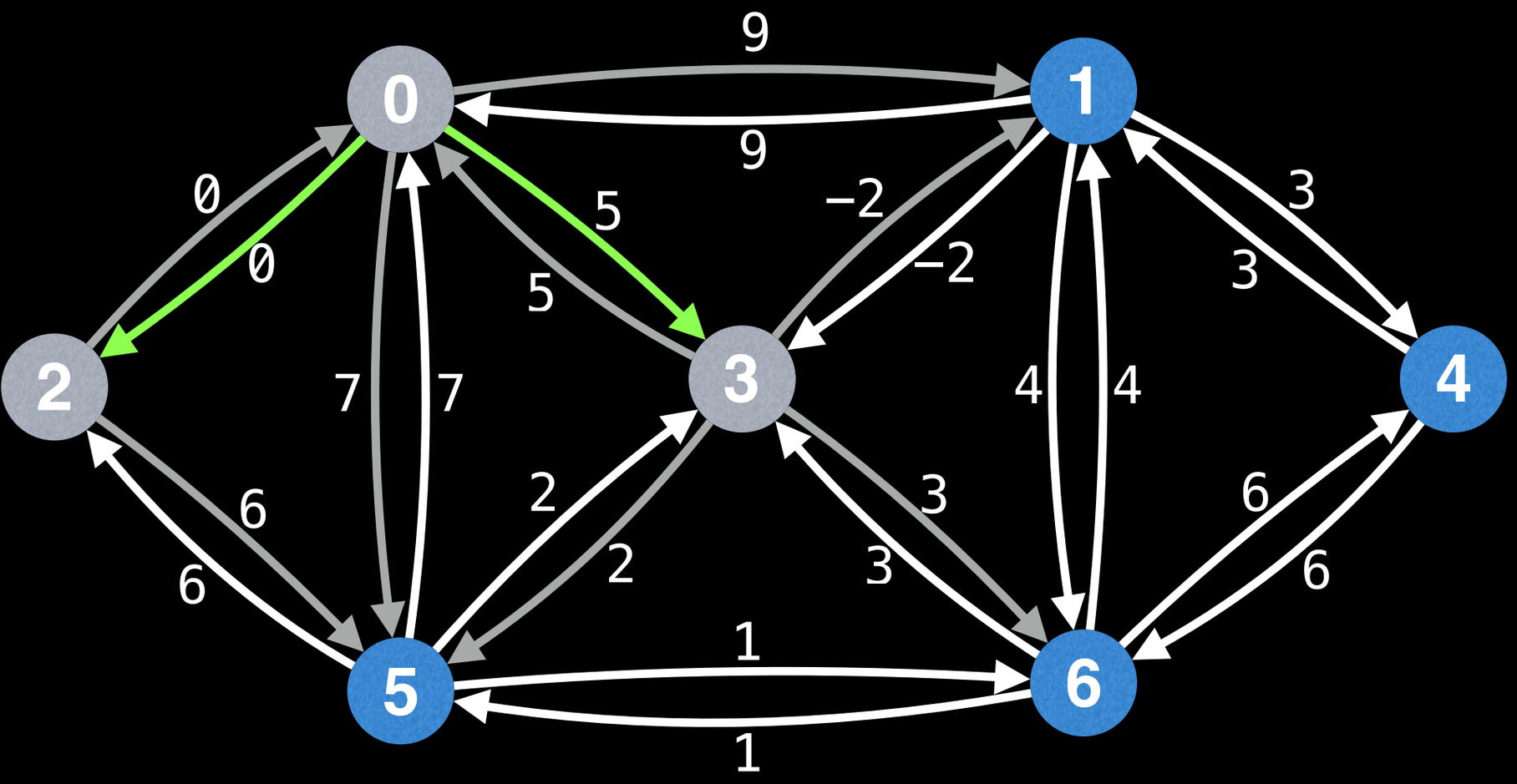
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)



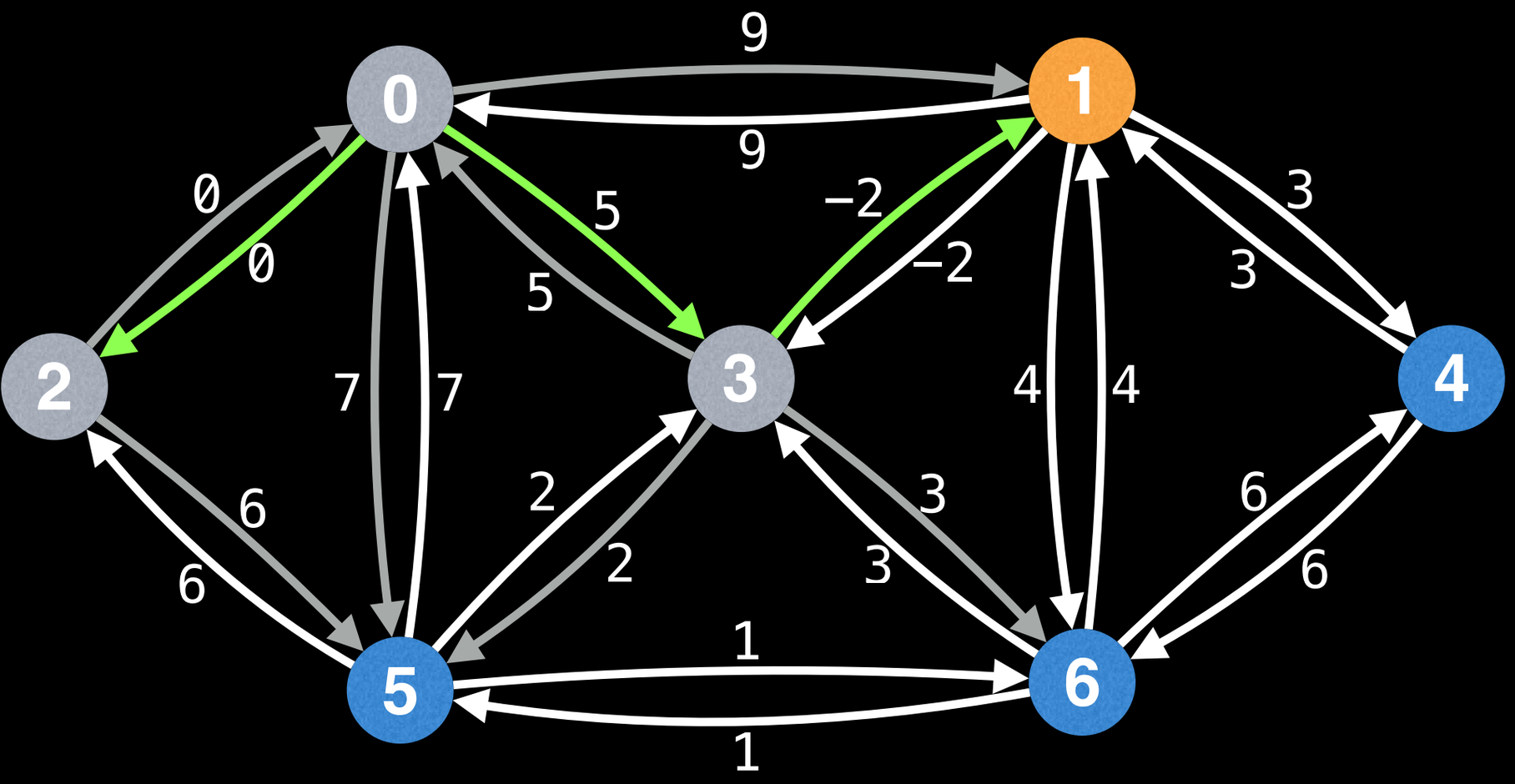
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)



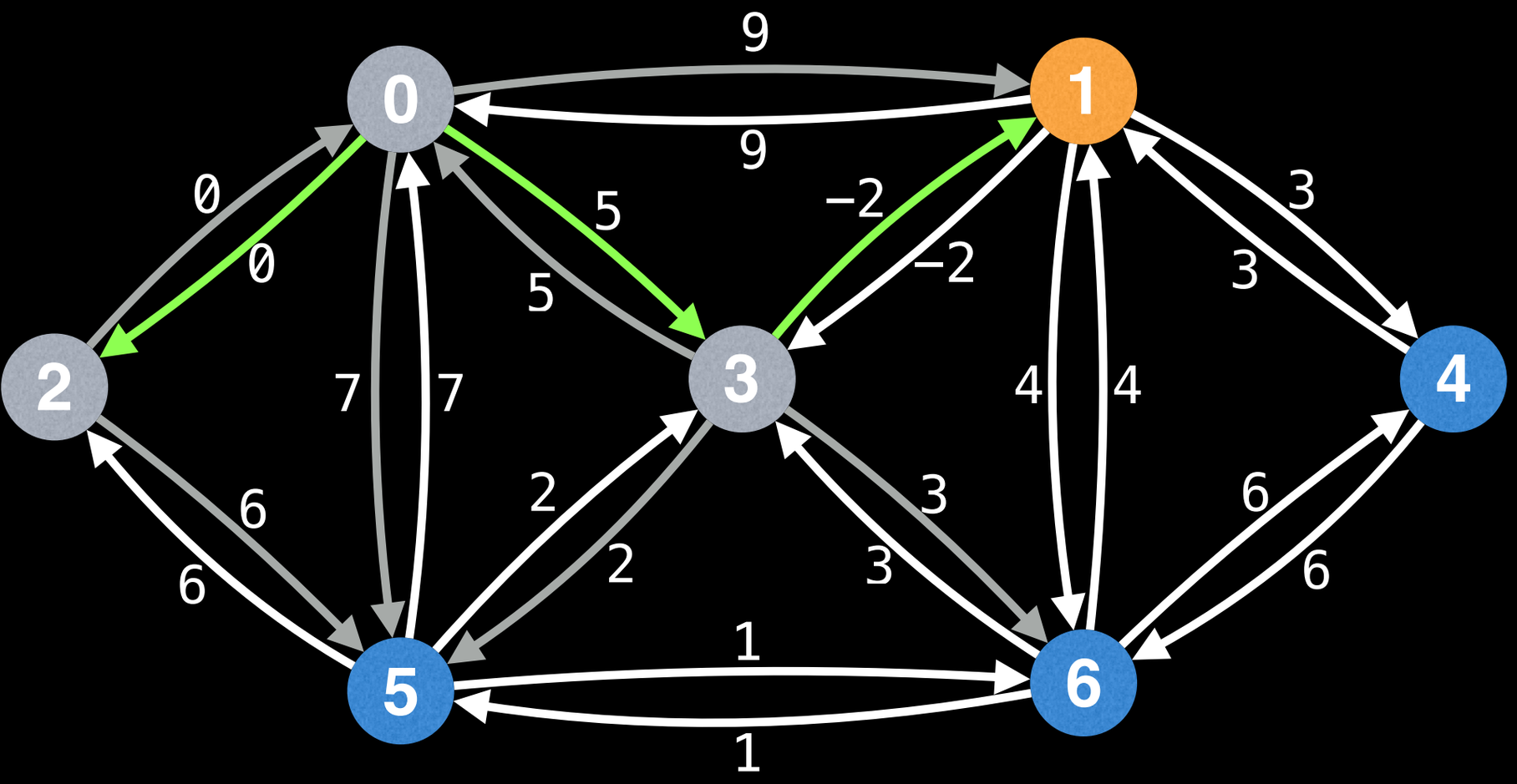
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)



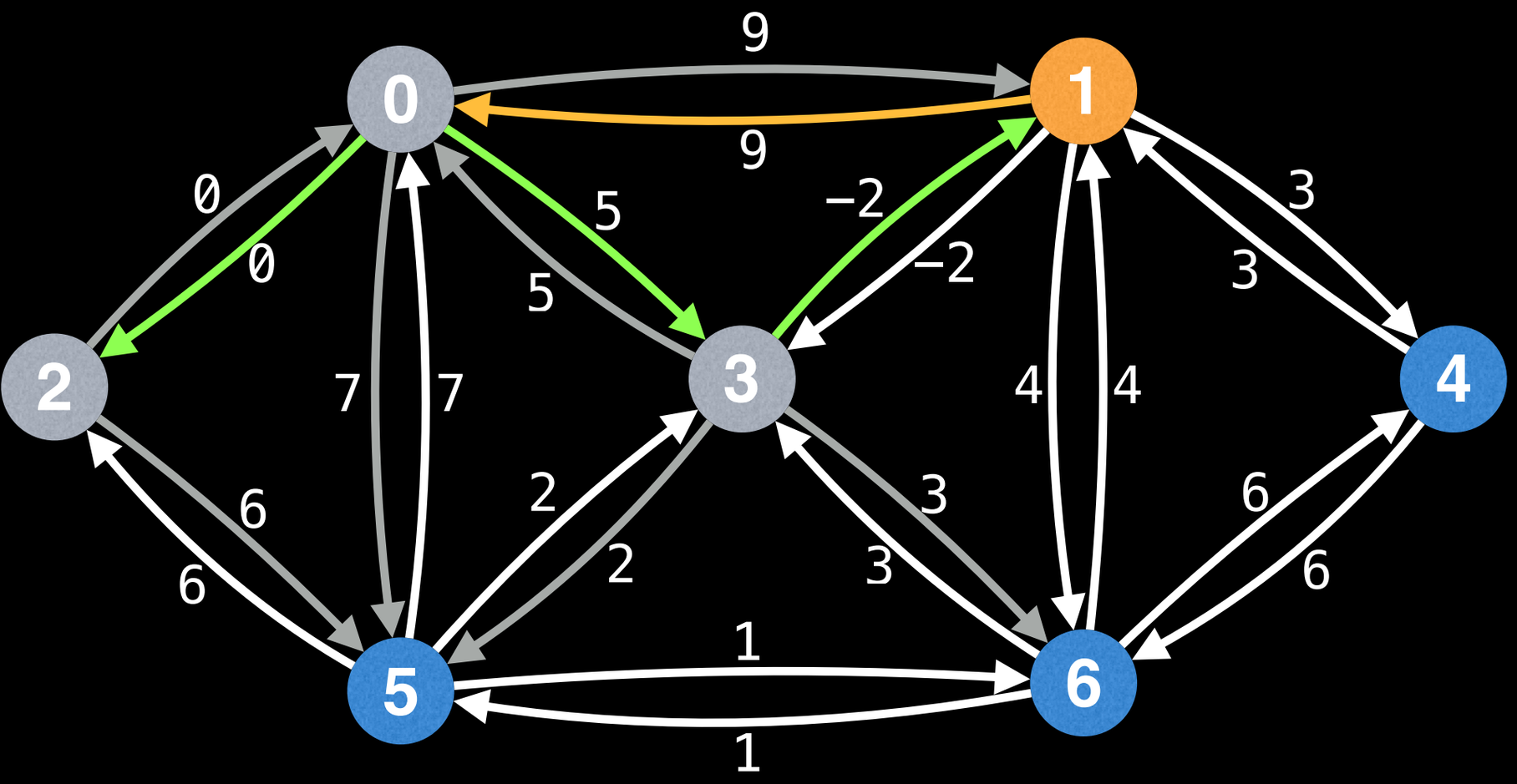
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)



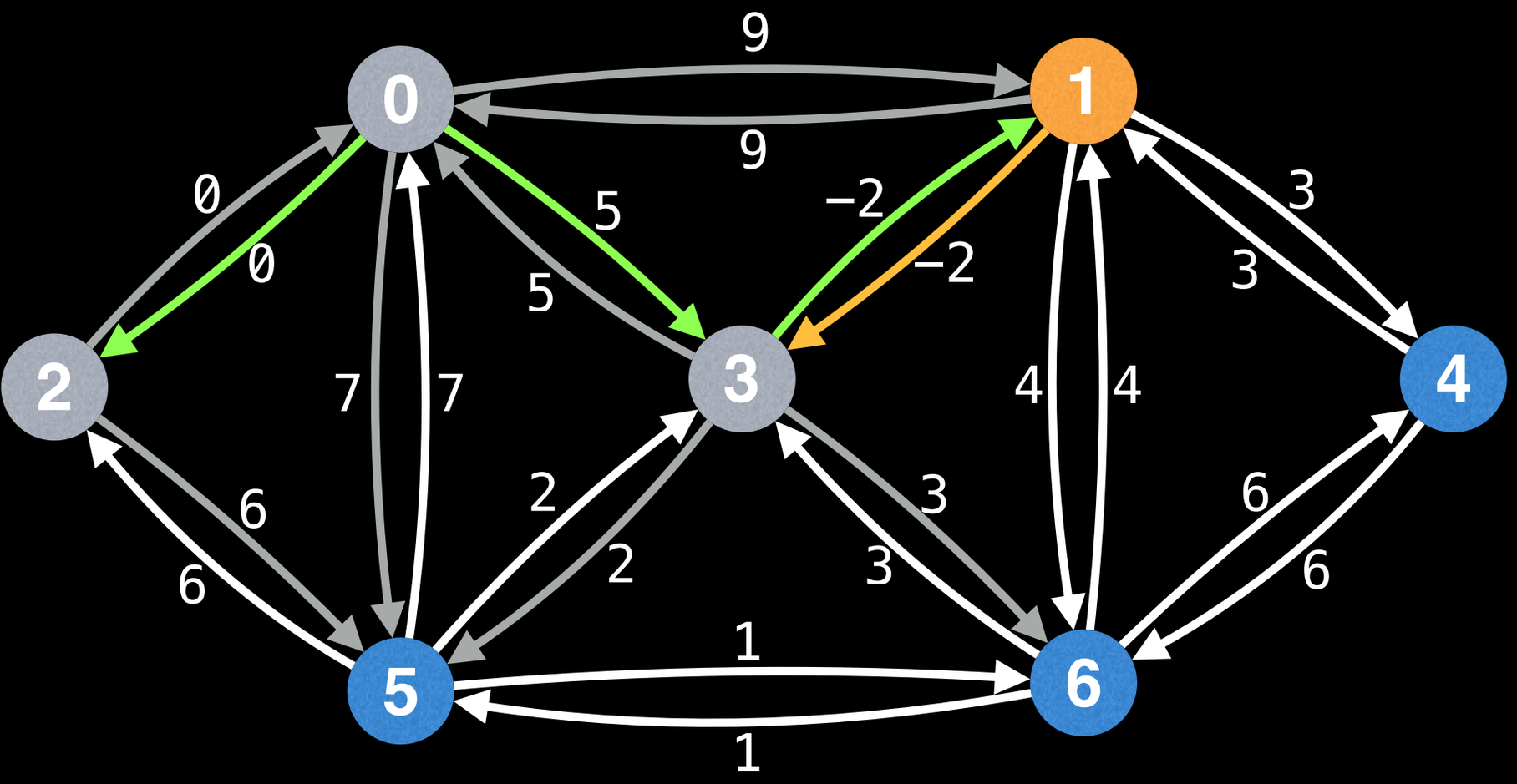
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)



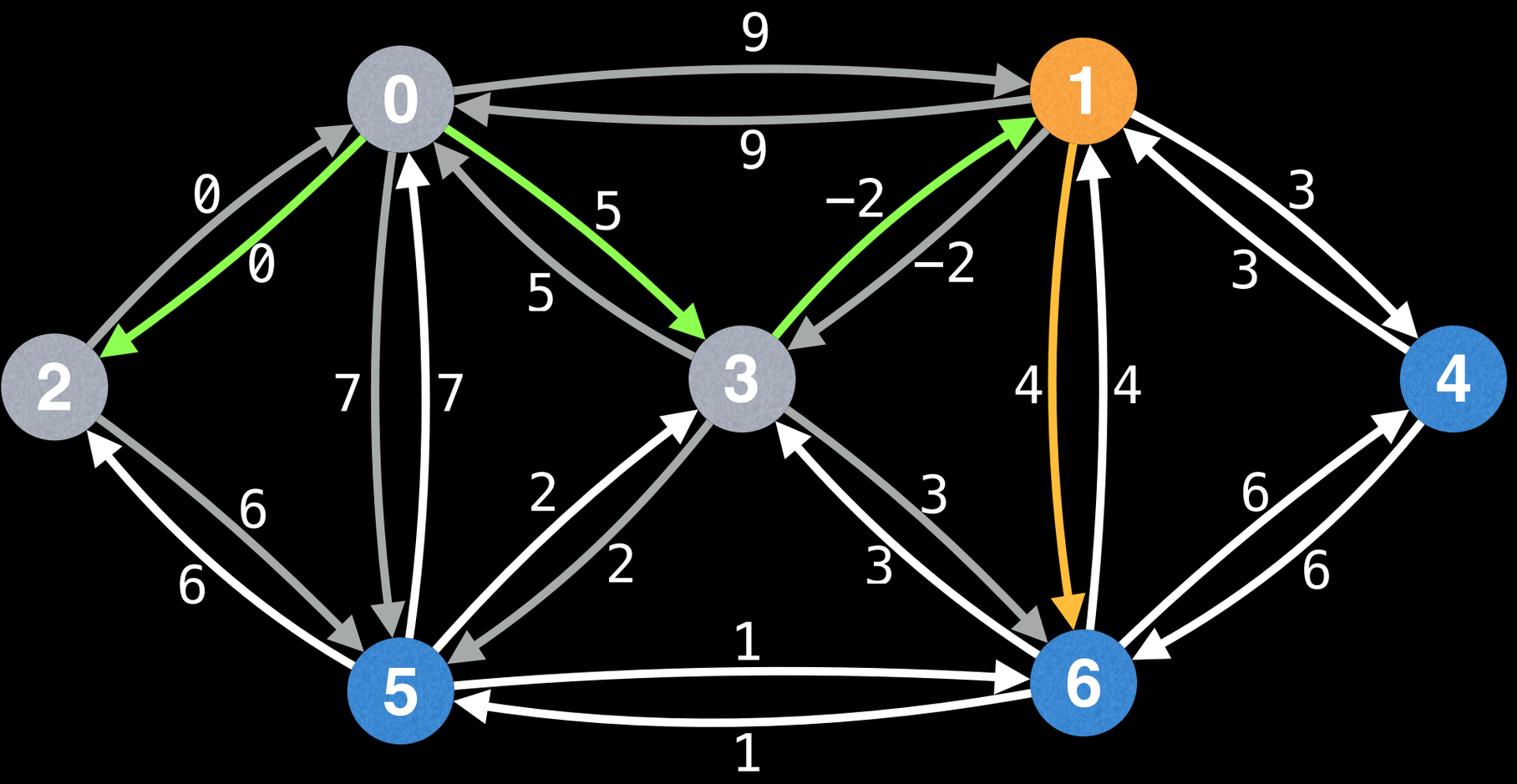
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)



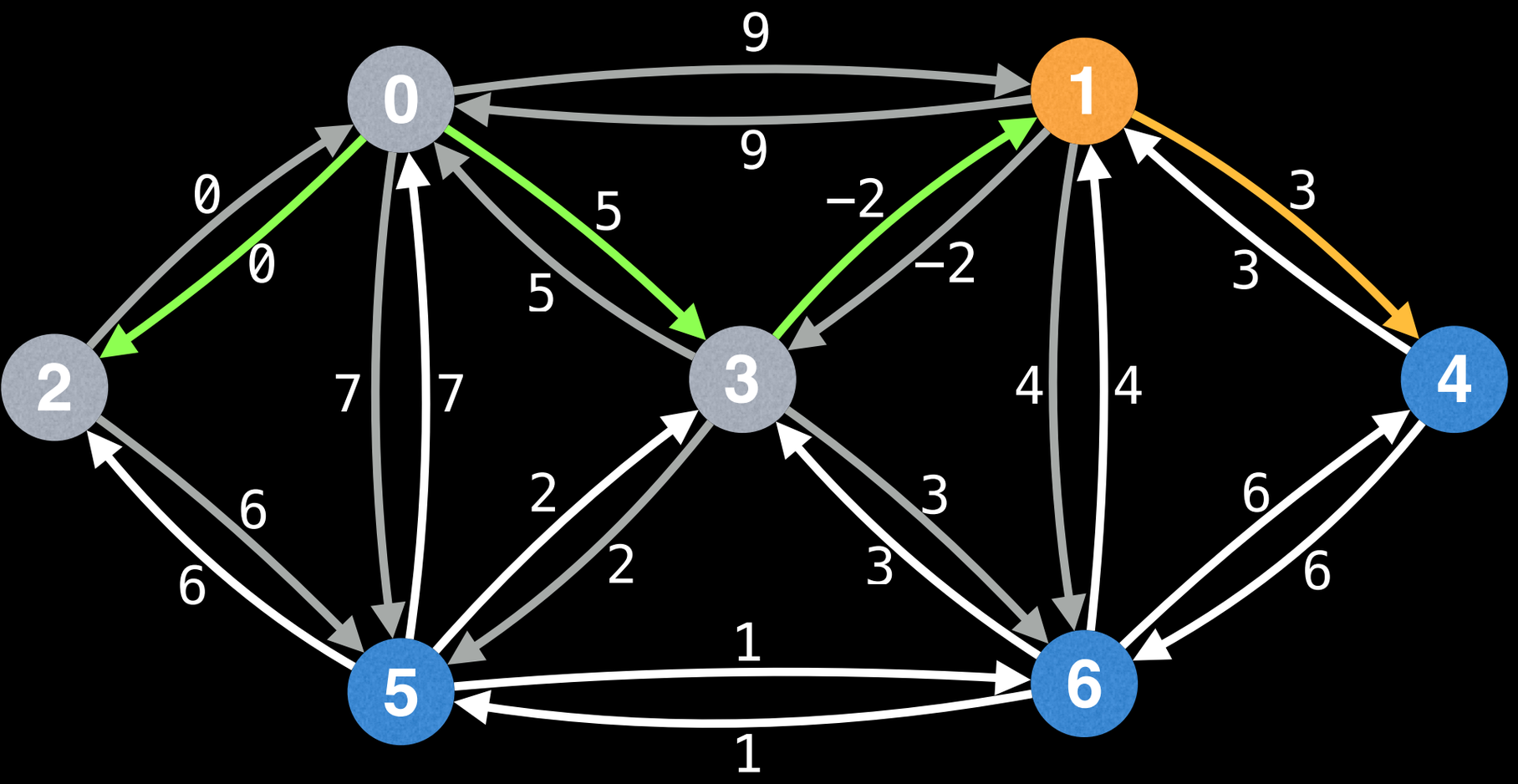
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)



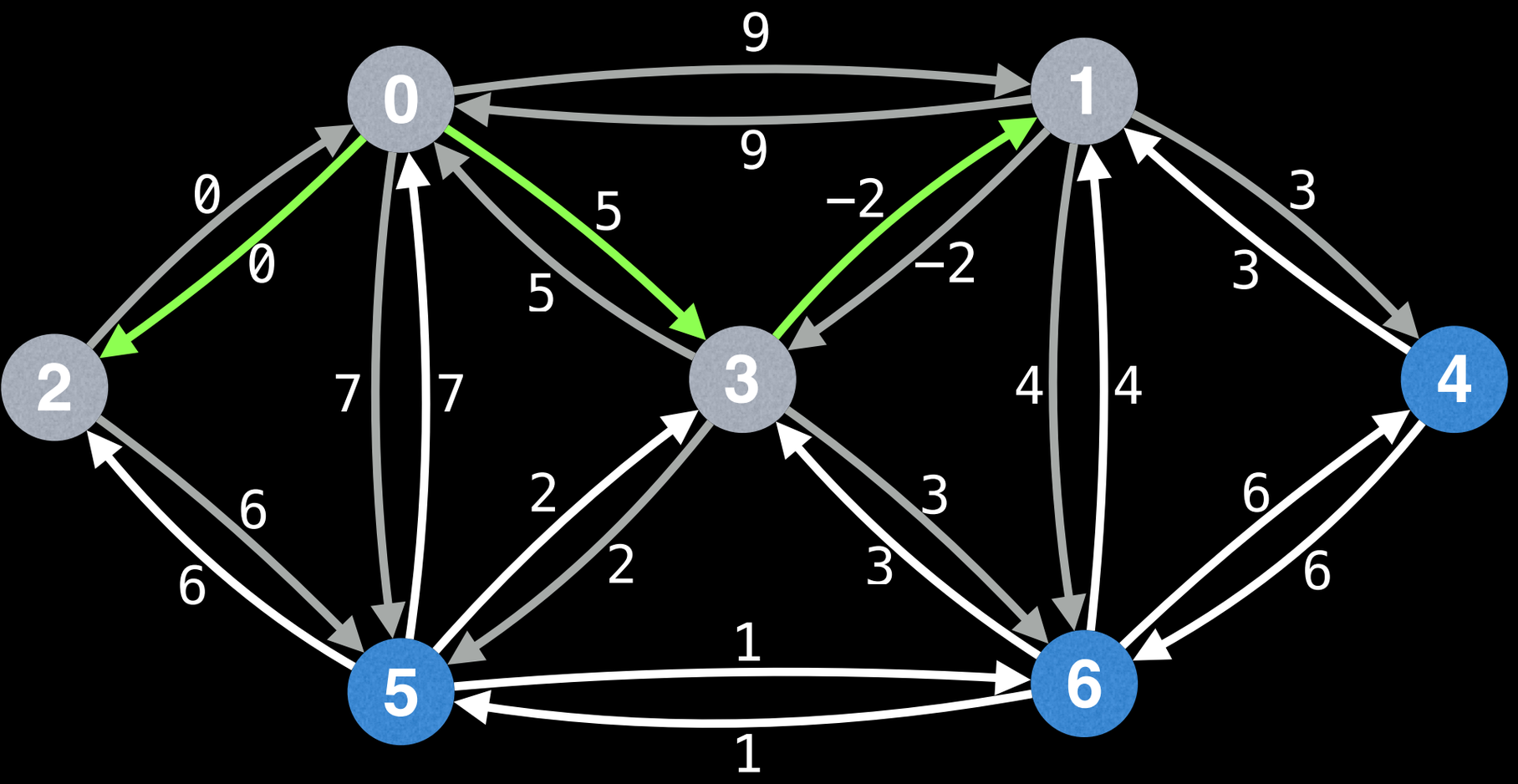
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)



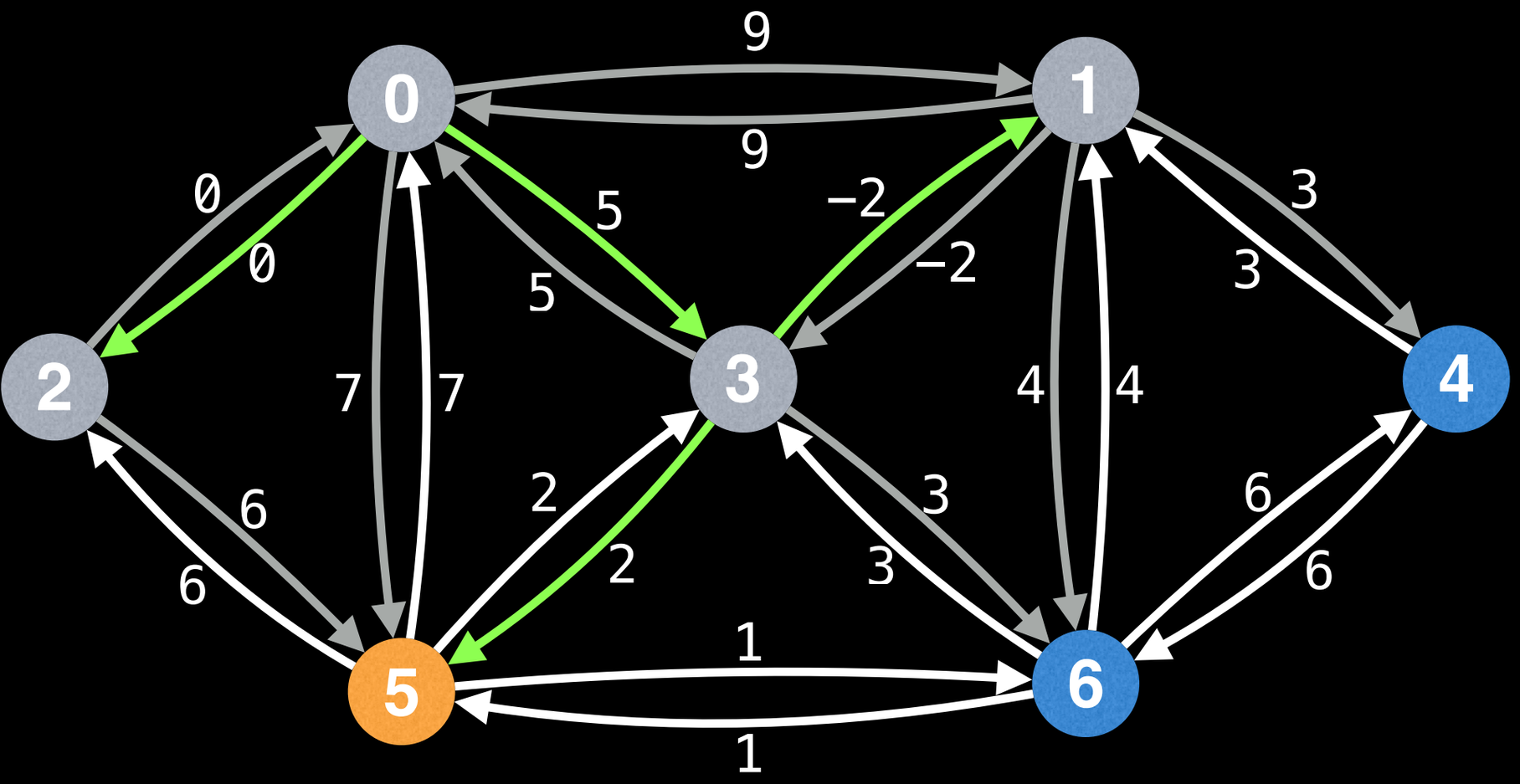
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



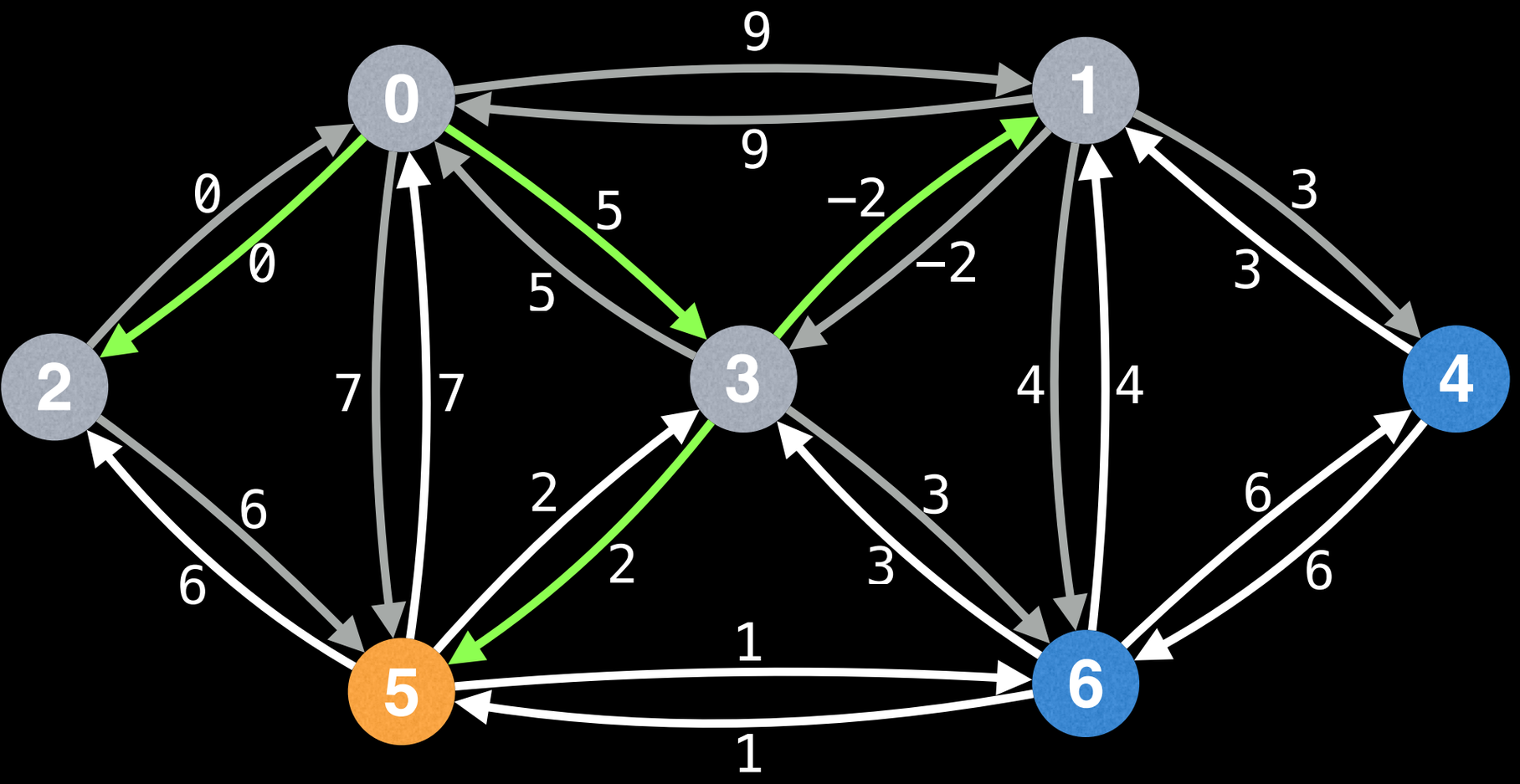
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



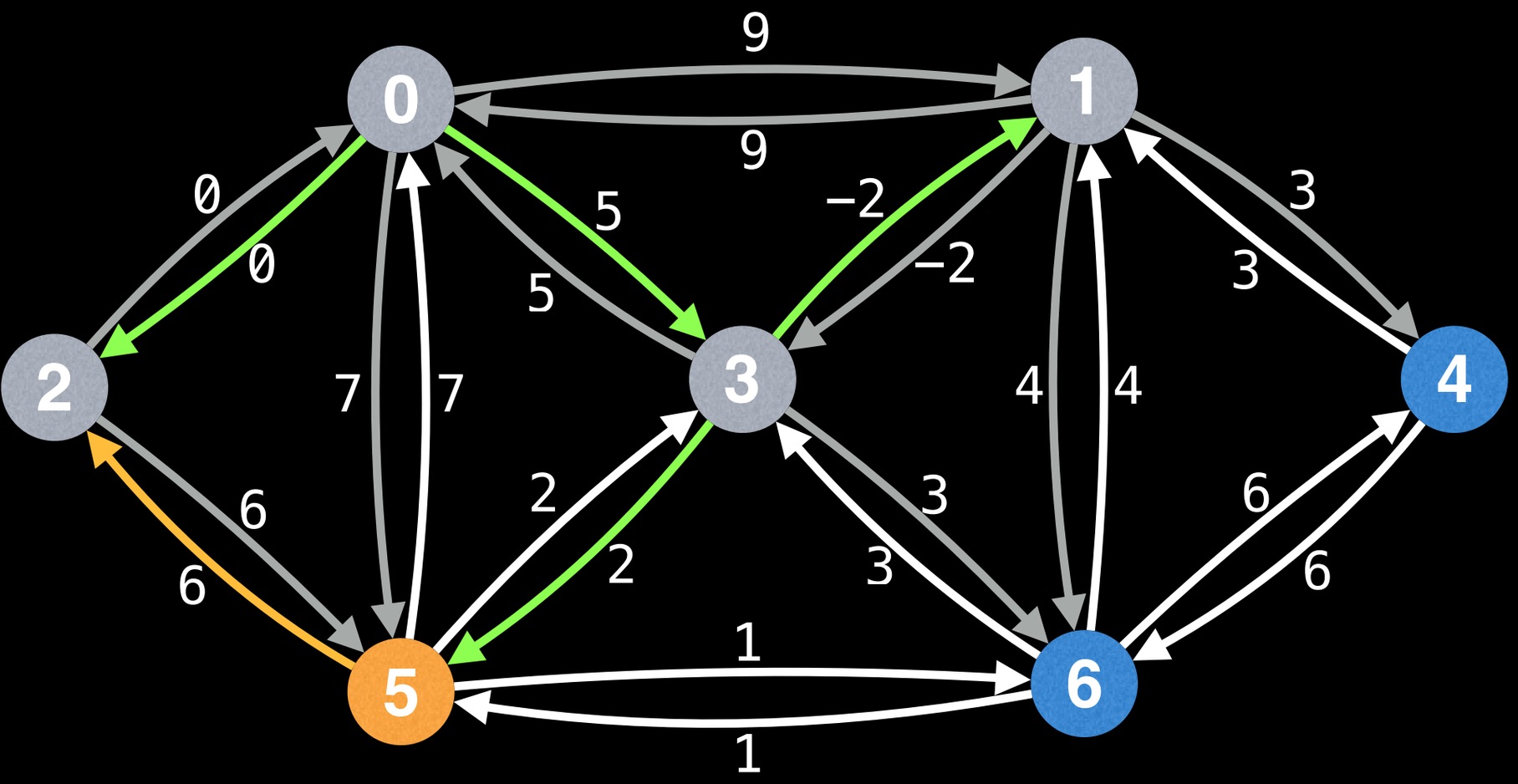
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



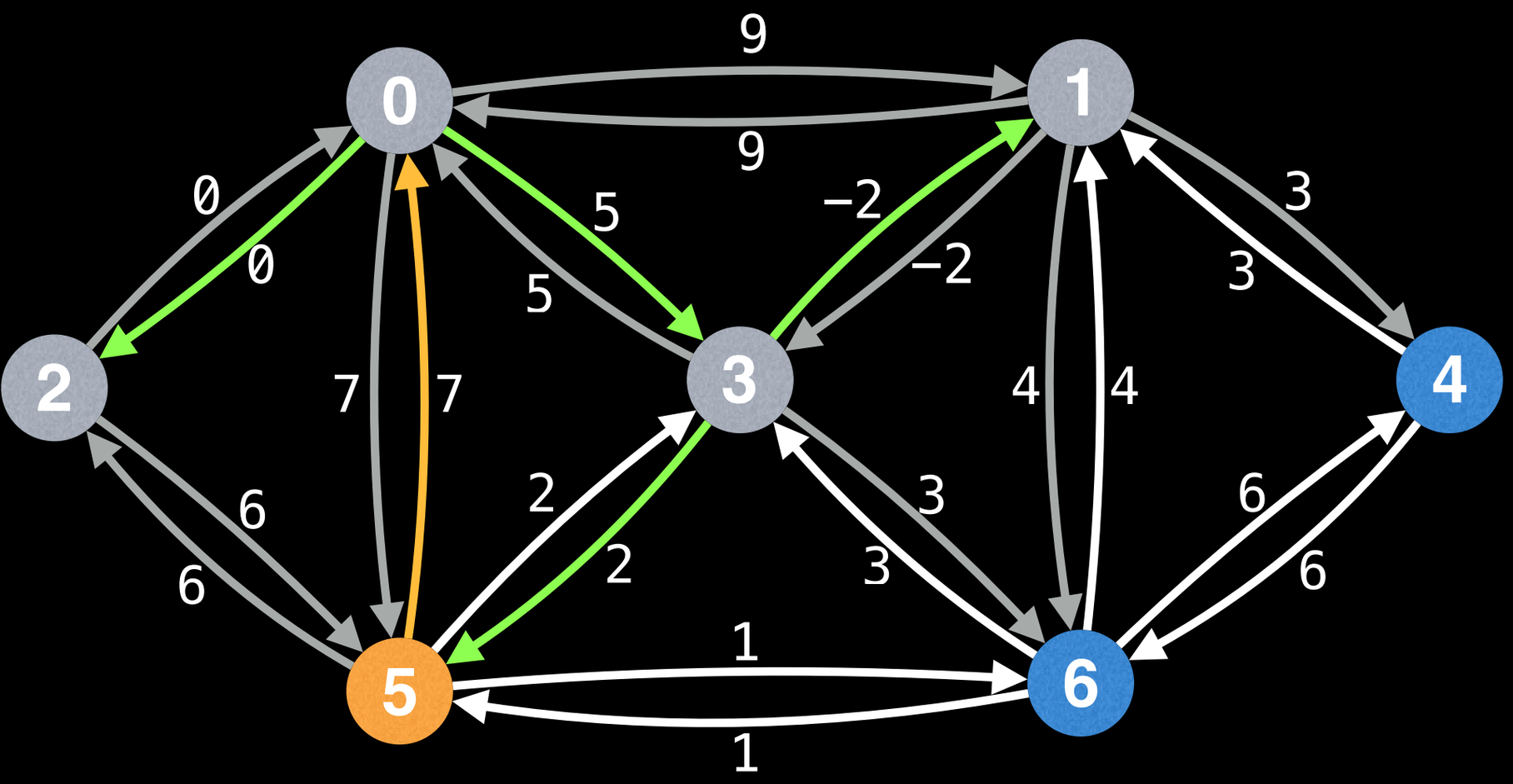
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



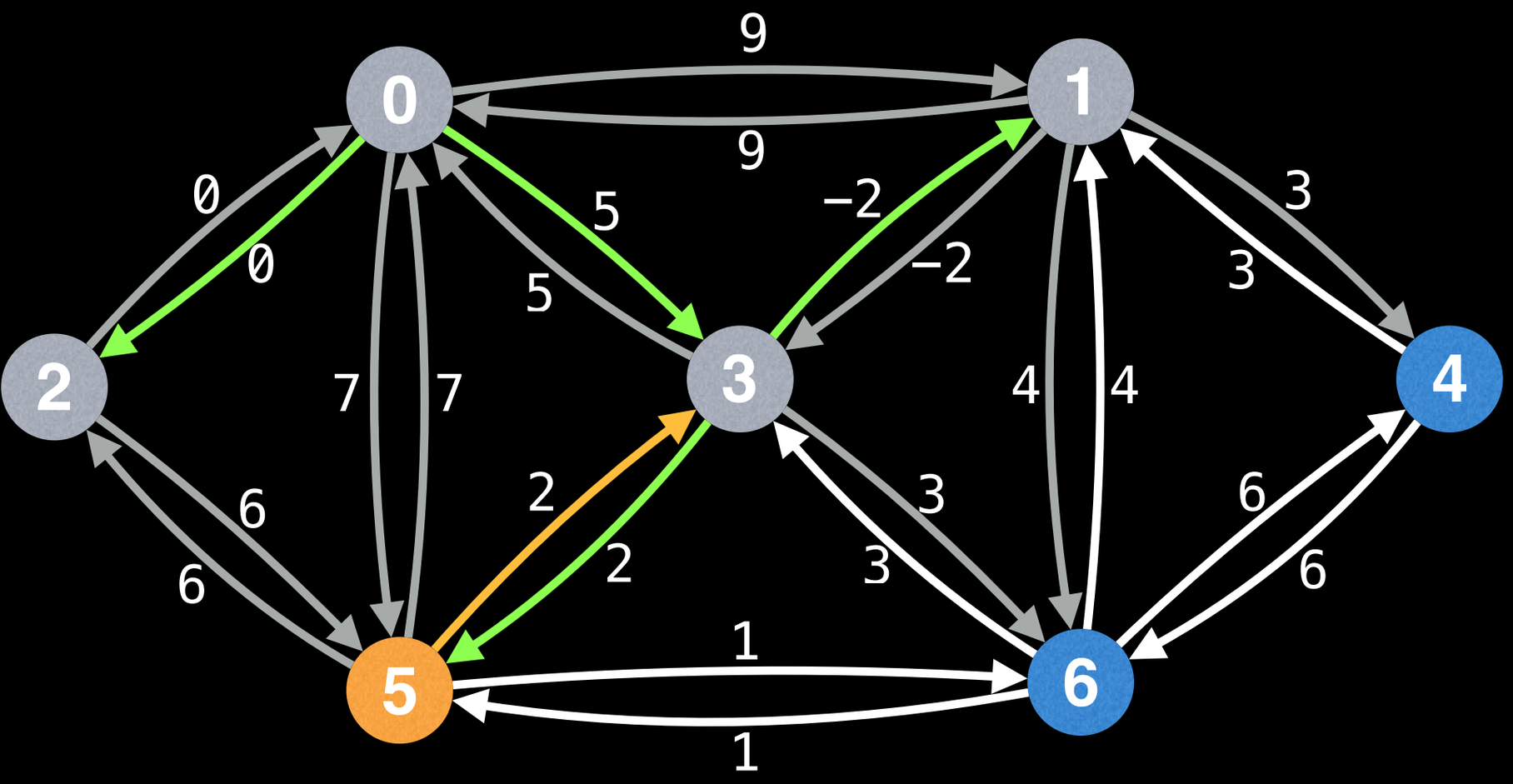
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



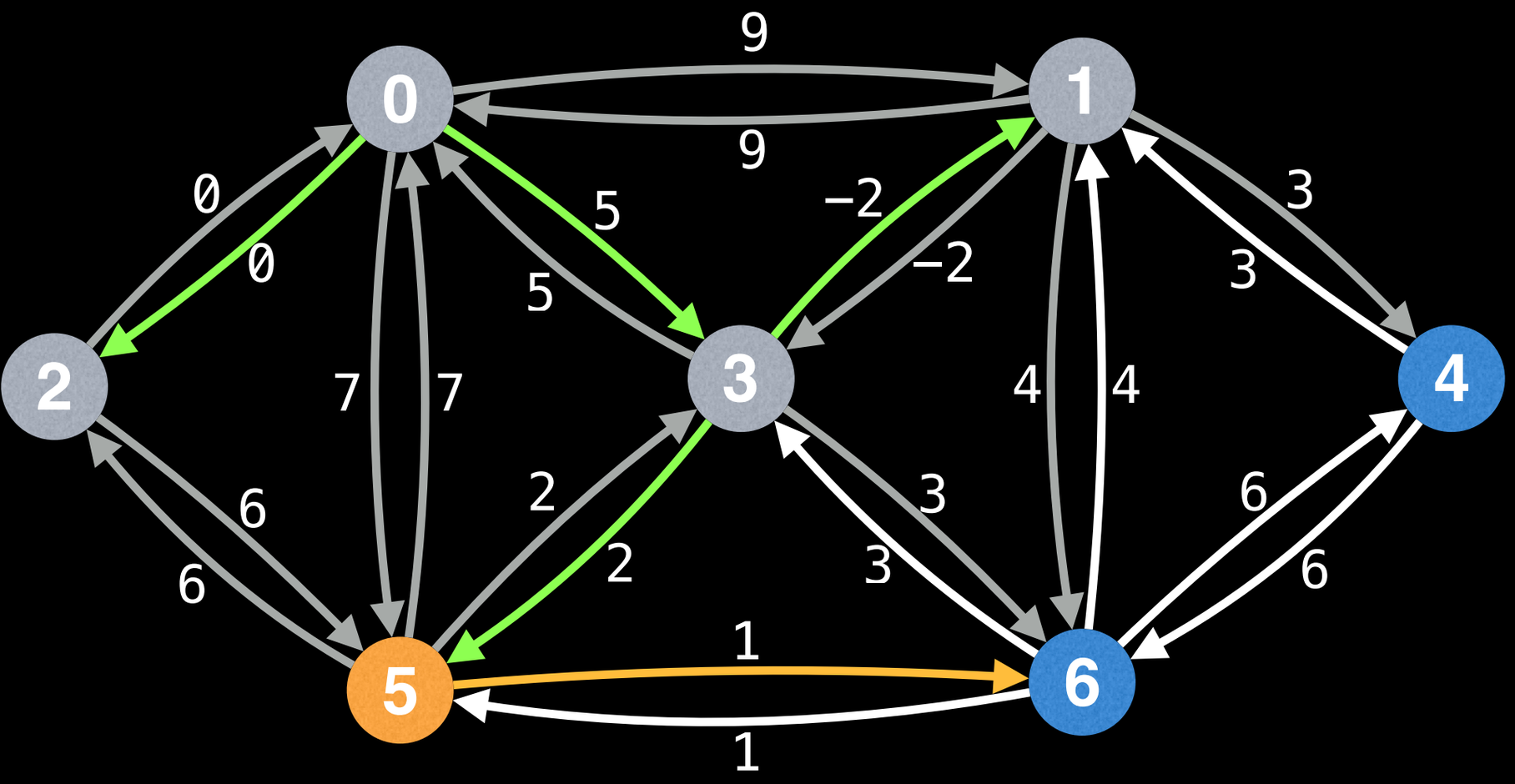
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



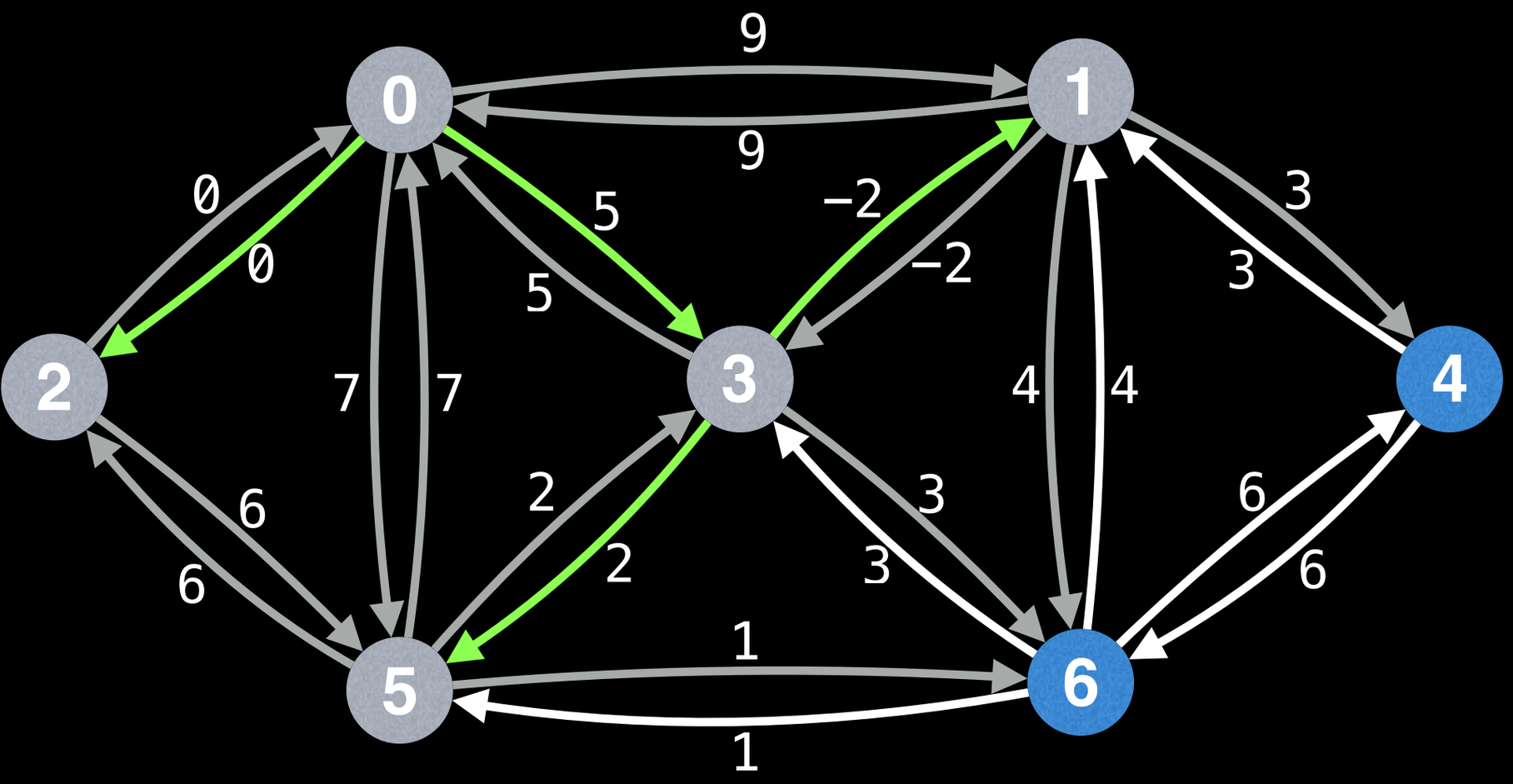
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)



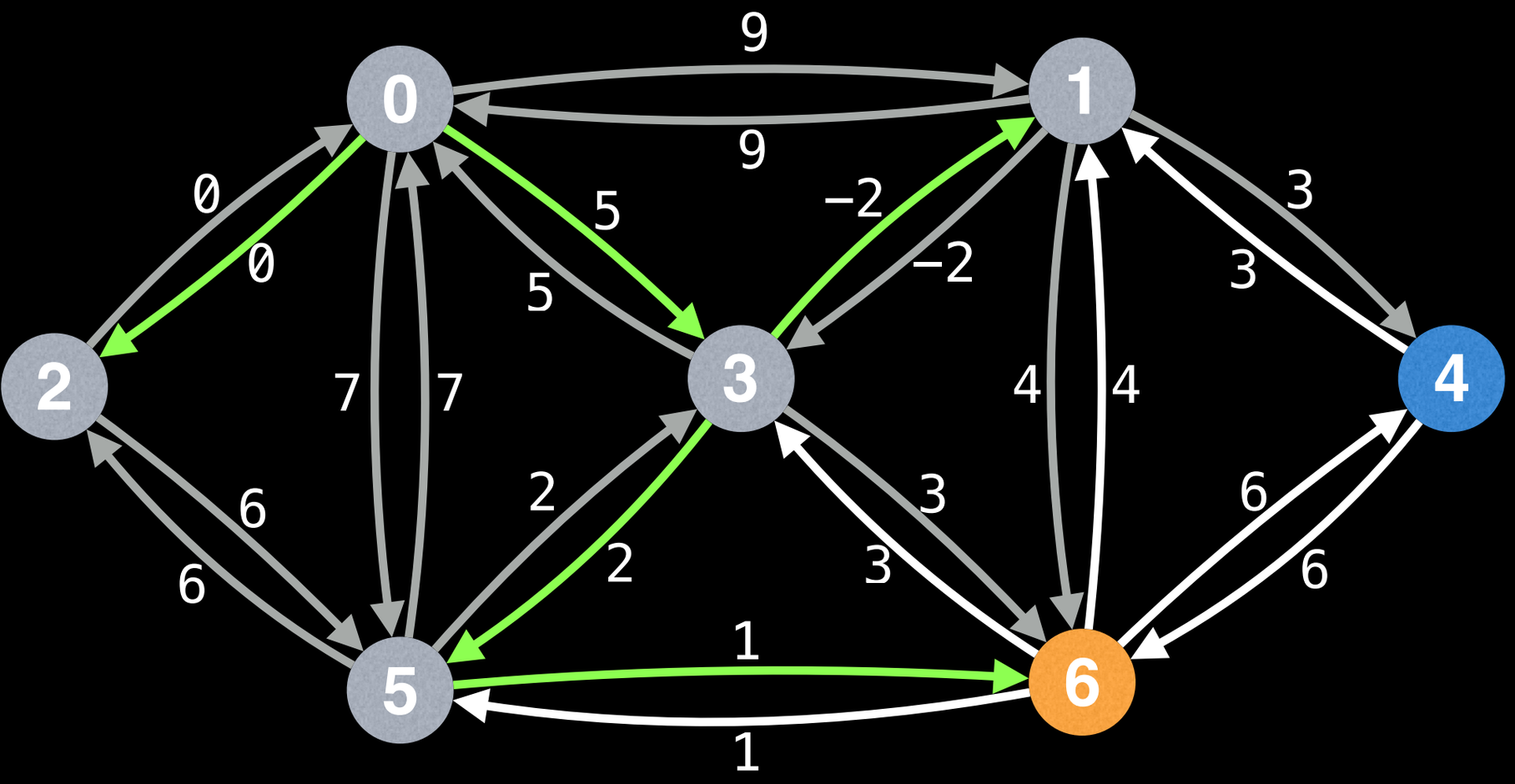
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)



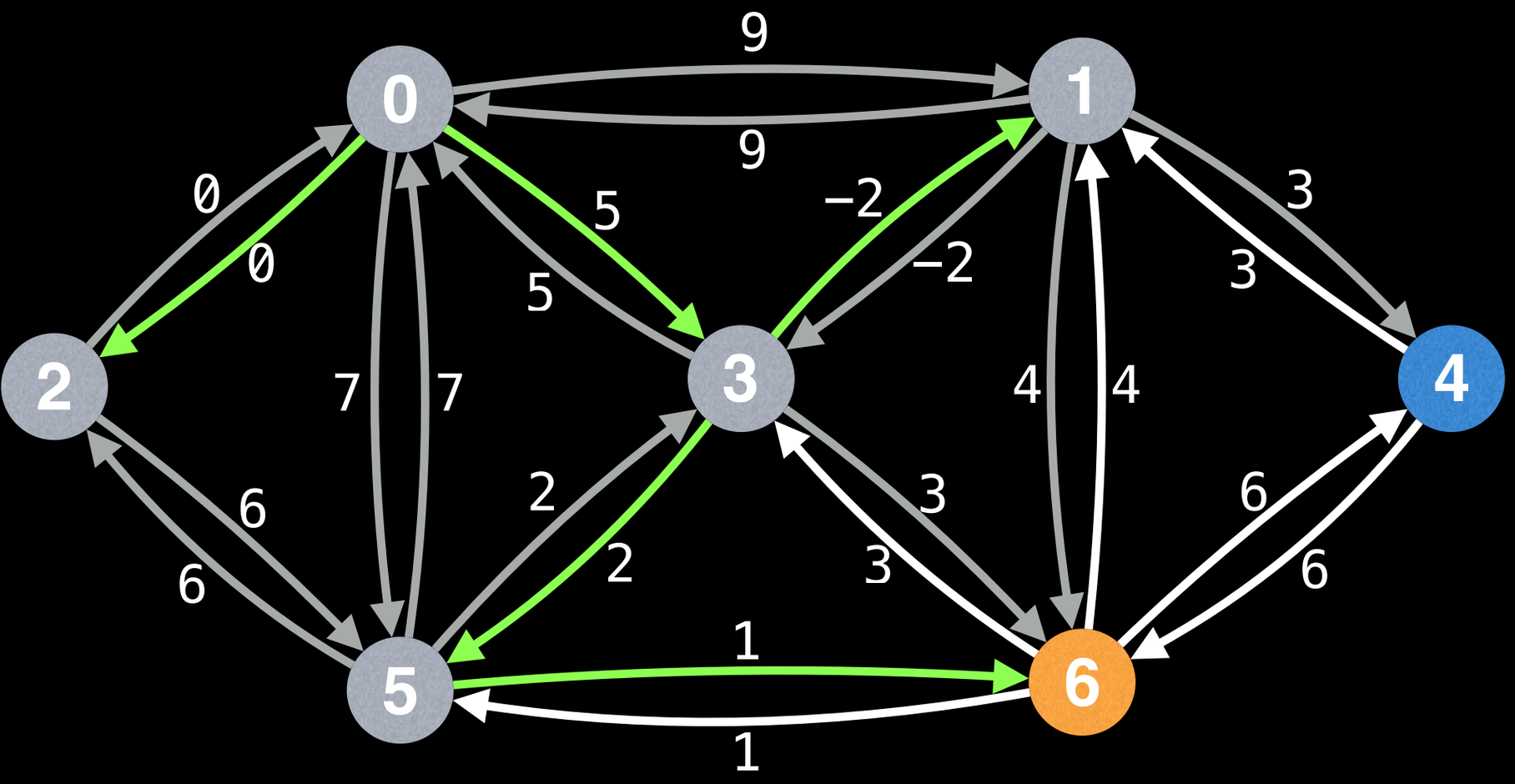
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)



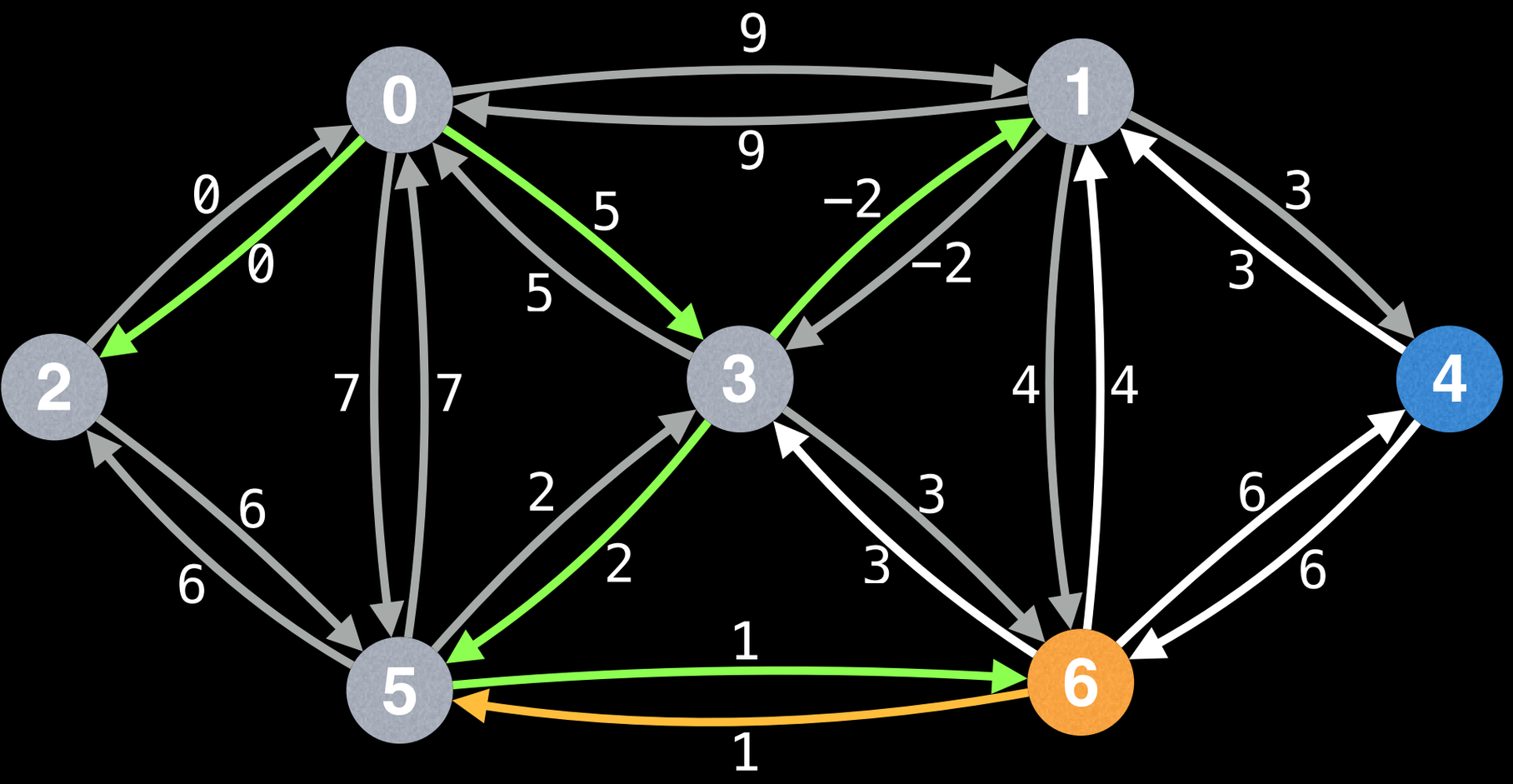
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)



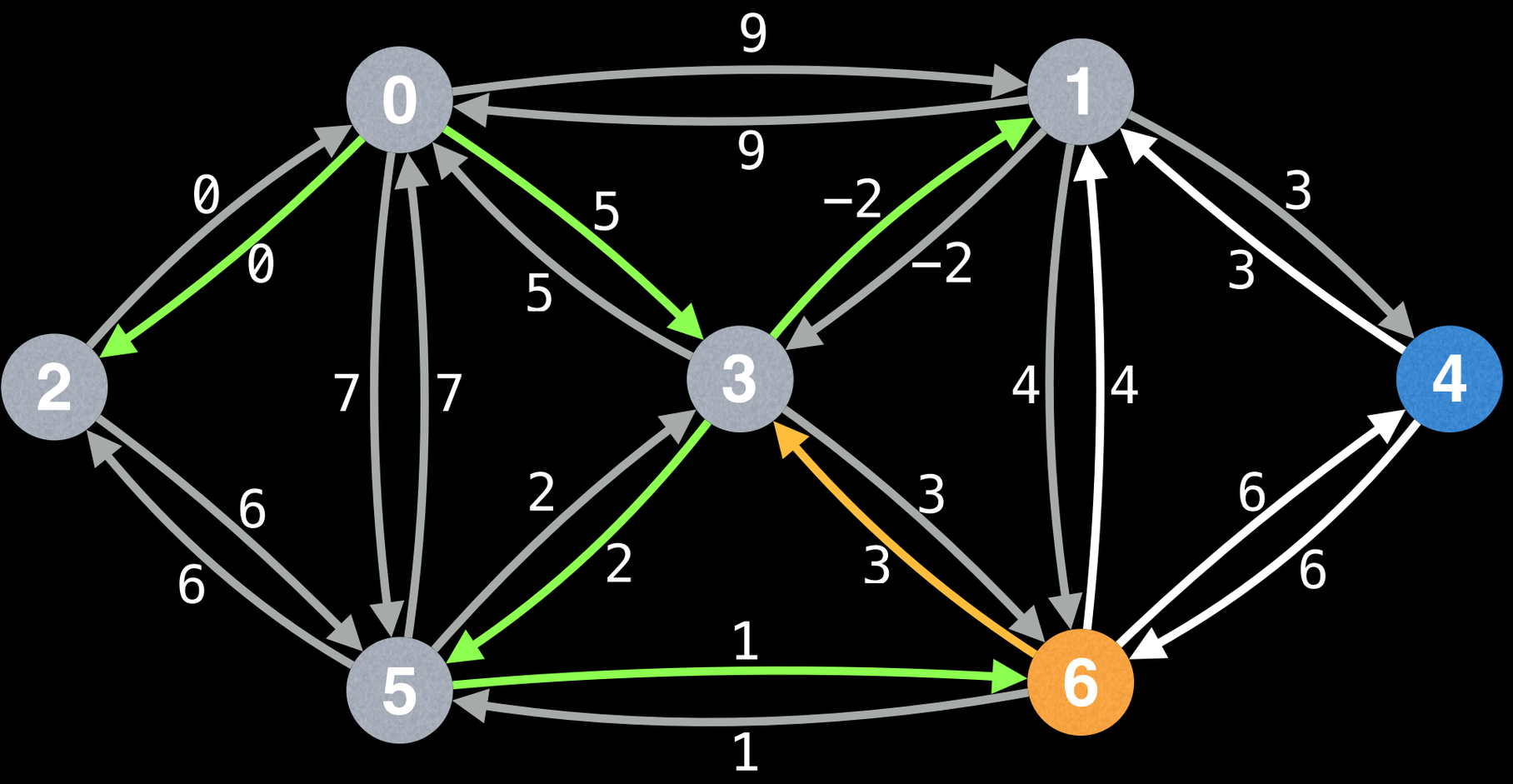
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



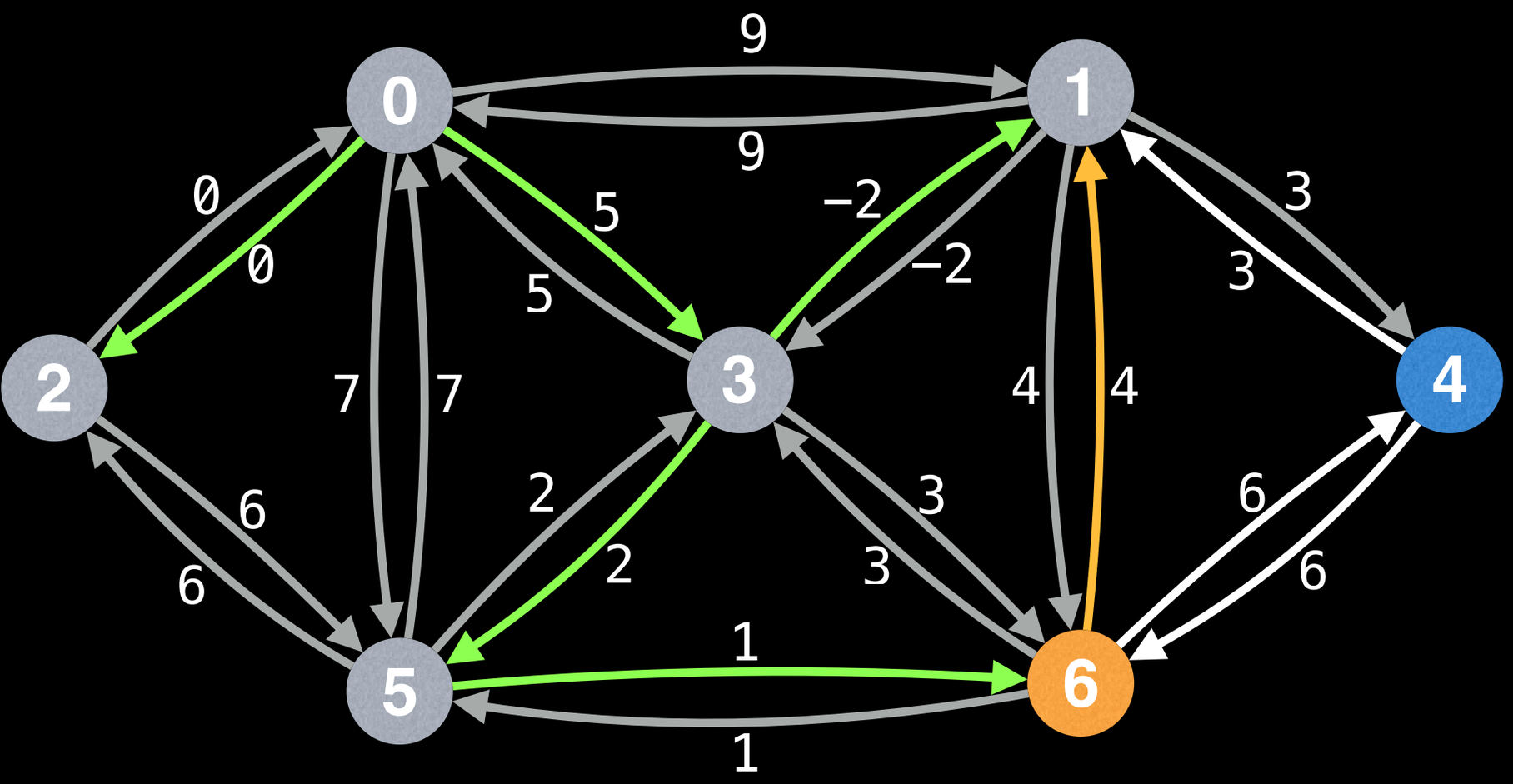
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



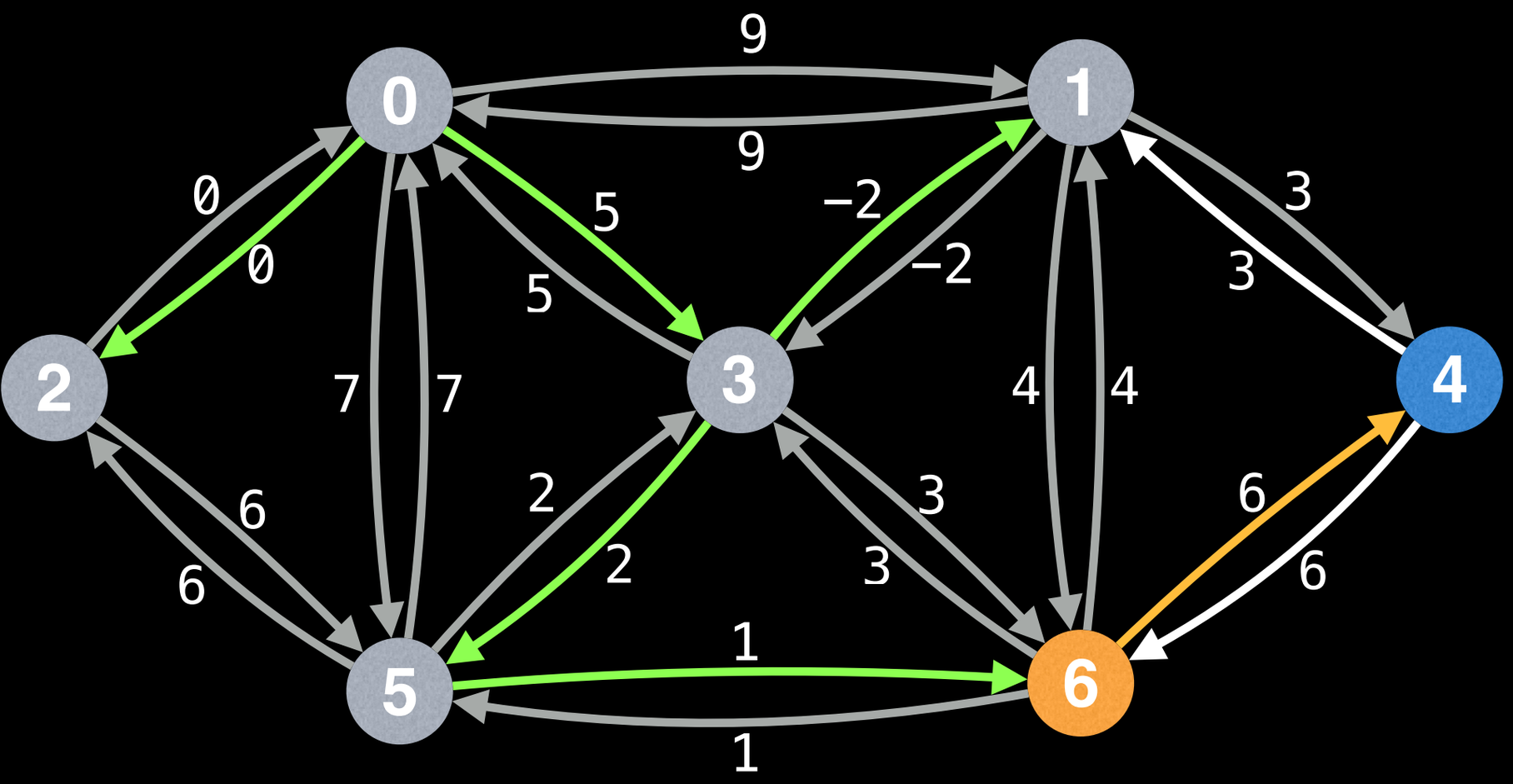
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



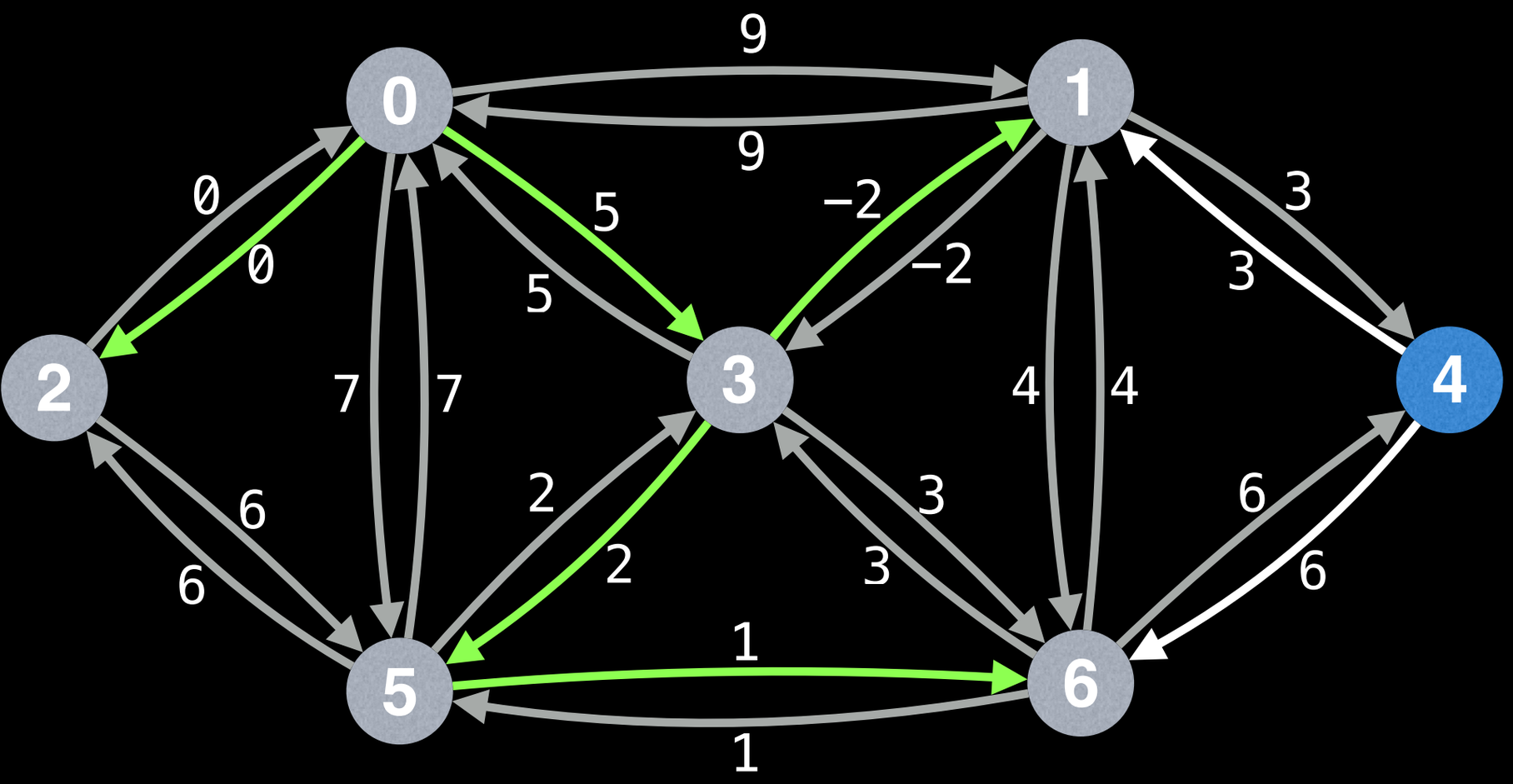
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



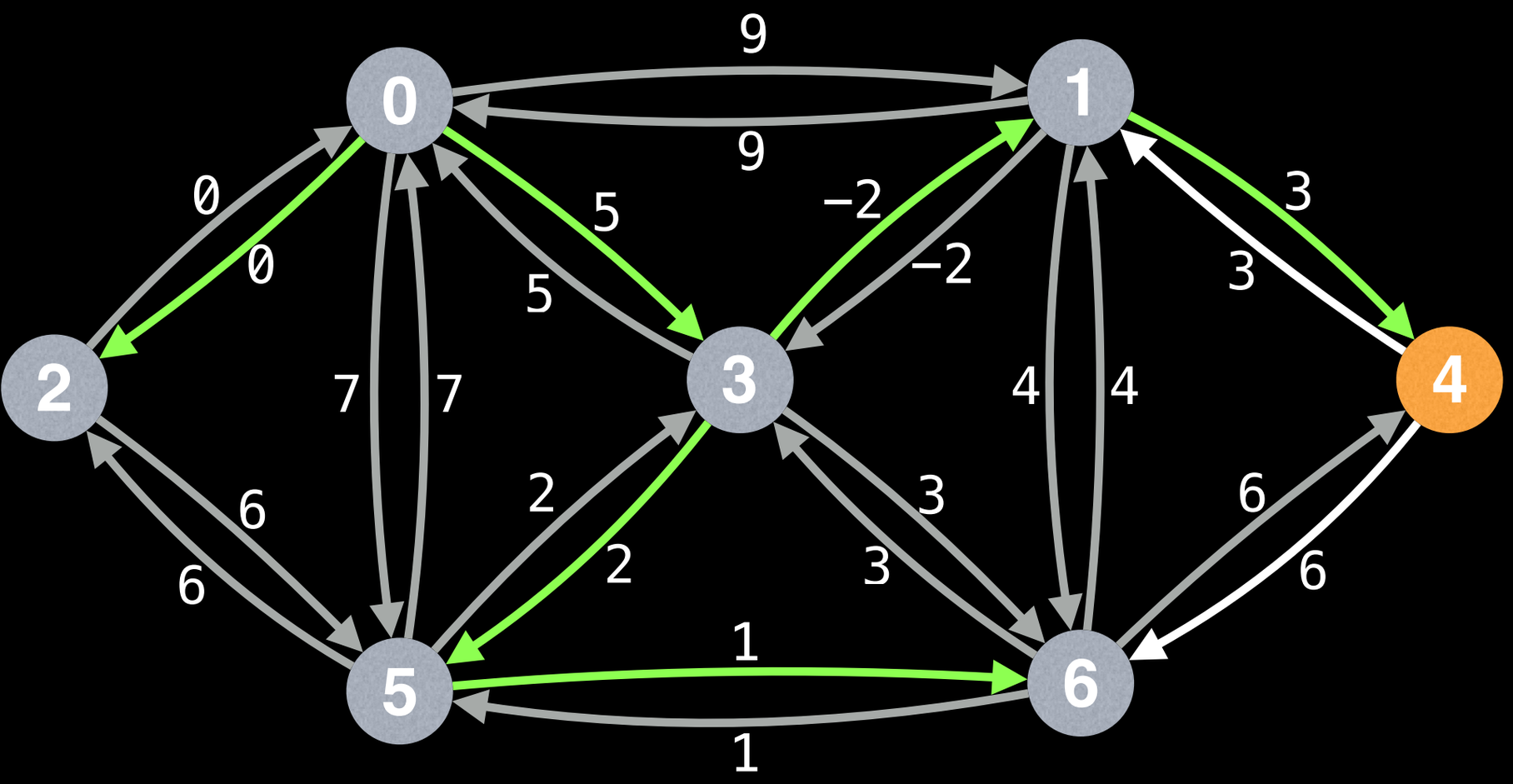
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



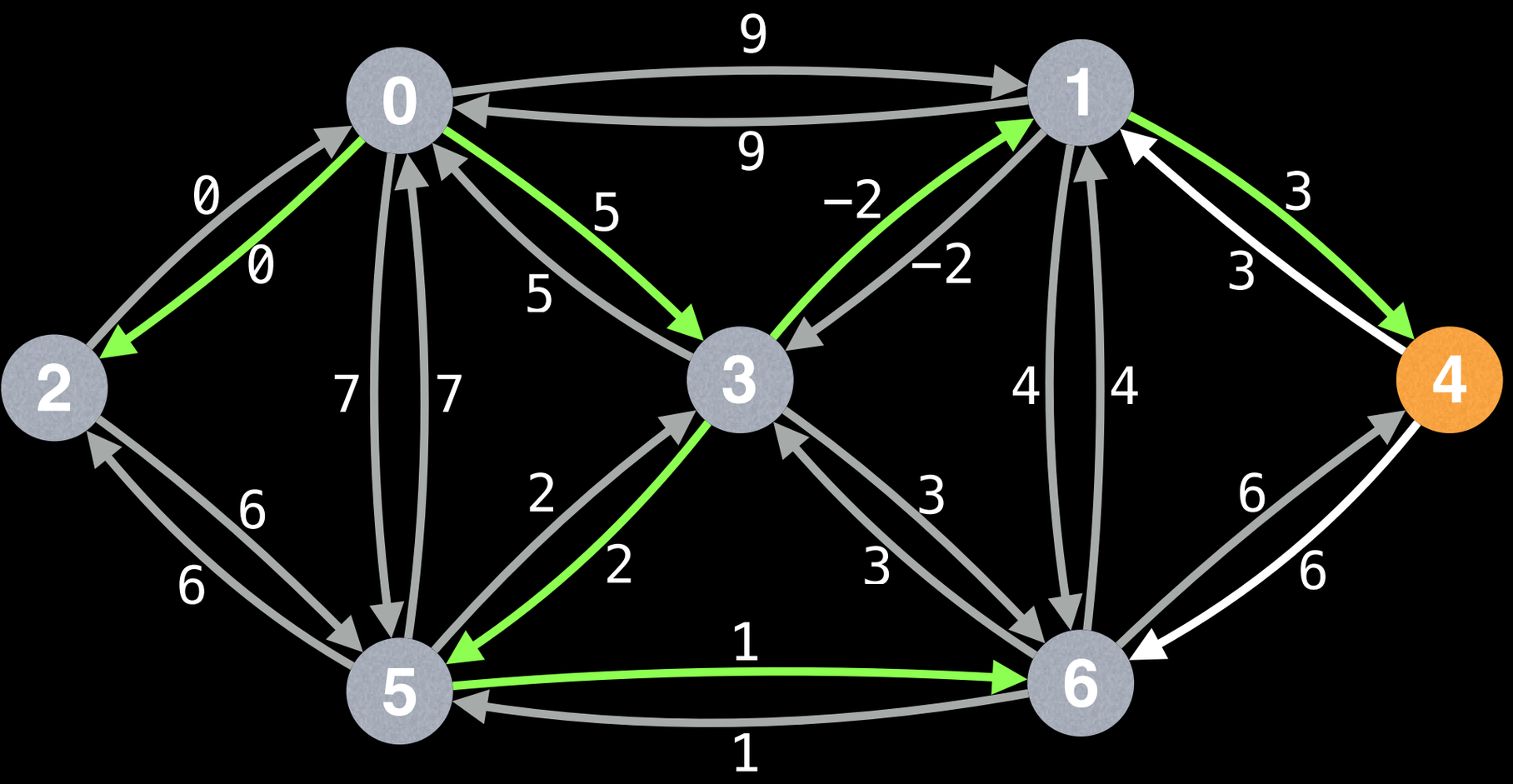
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



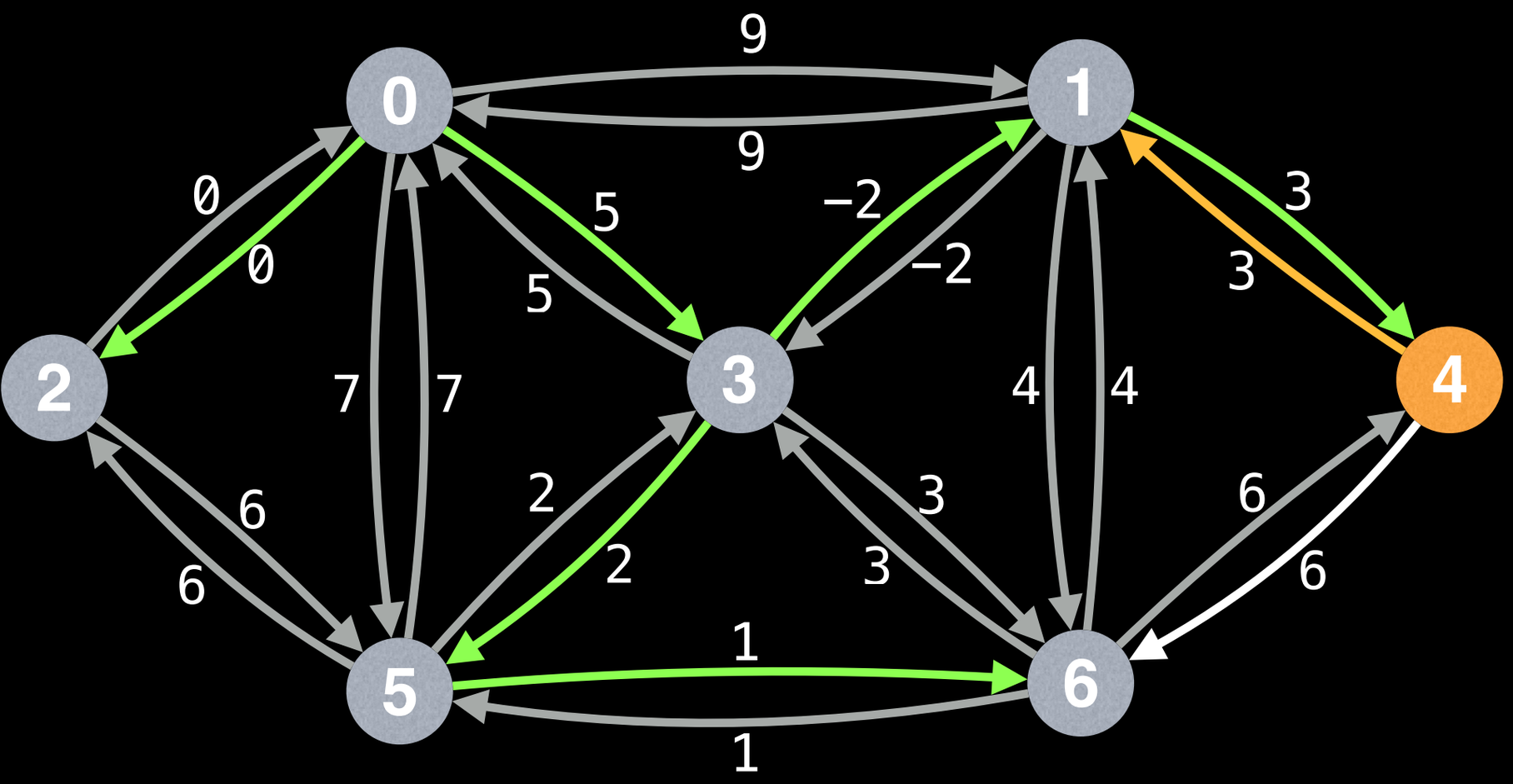
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)



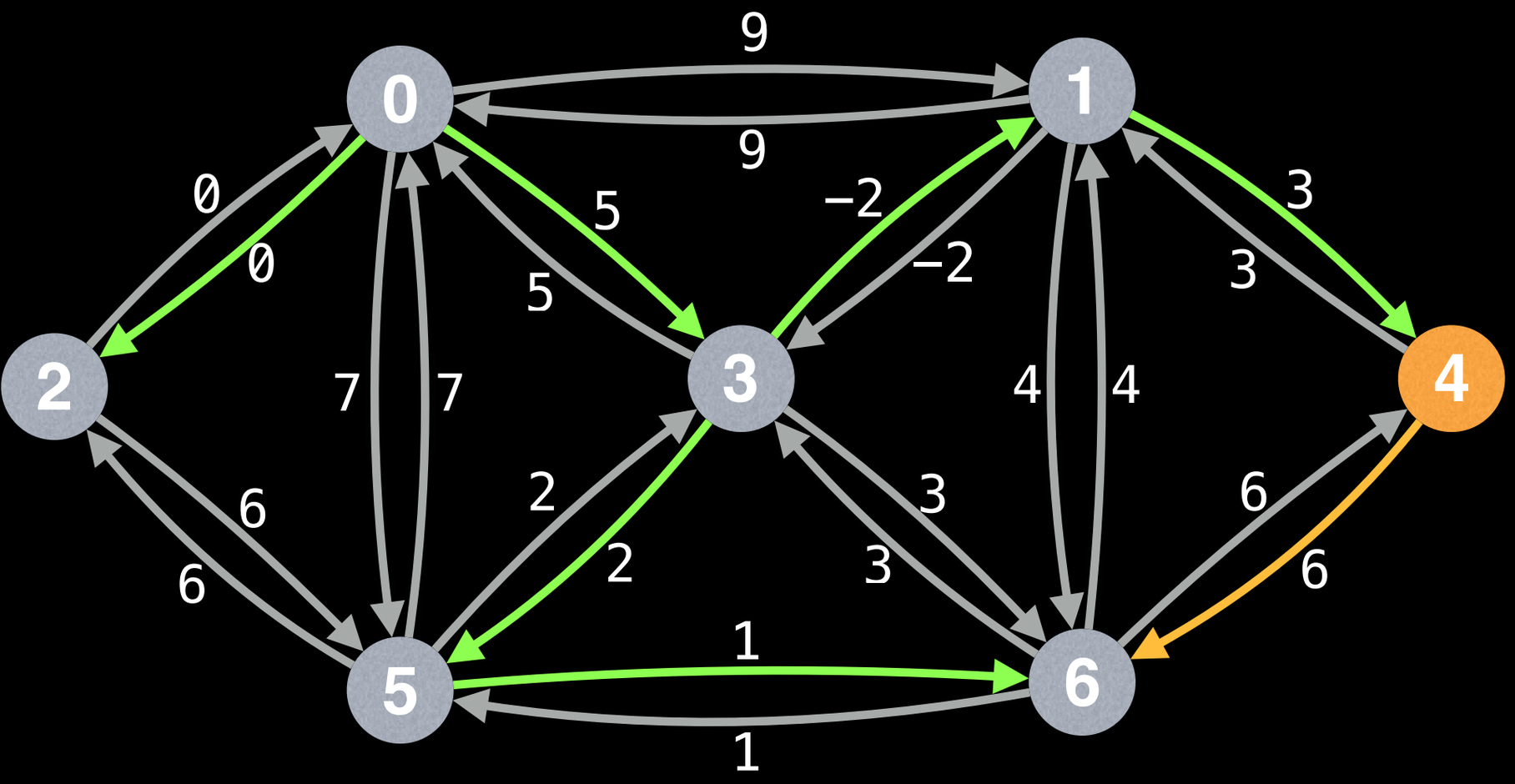
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~



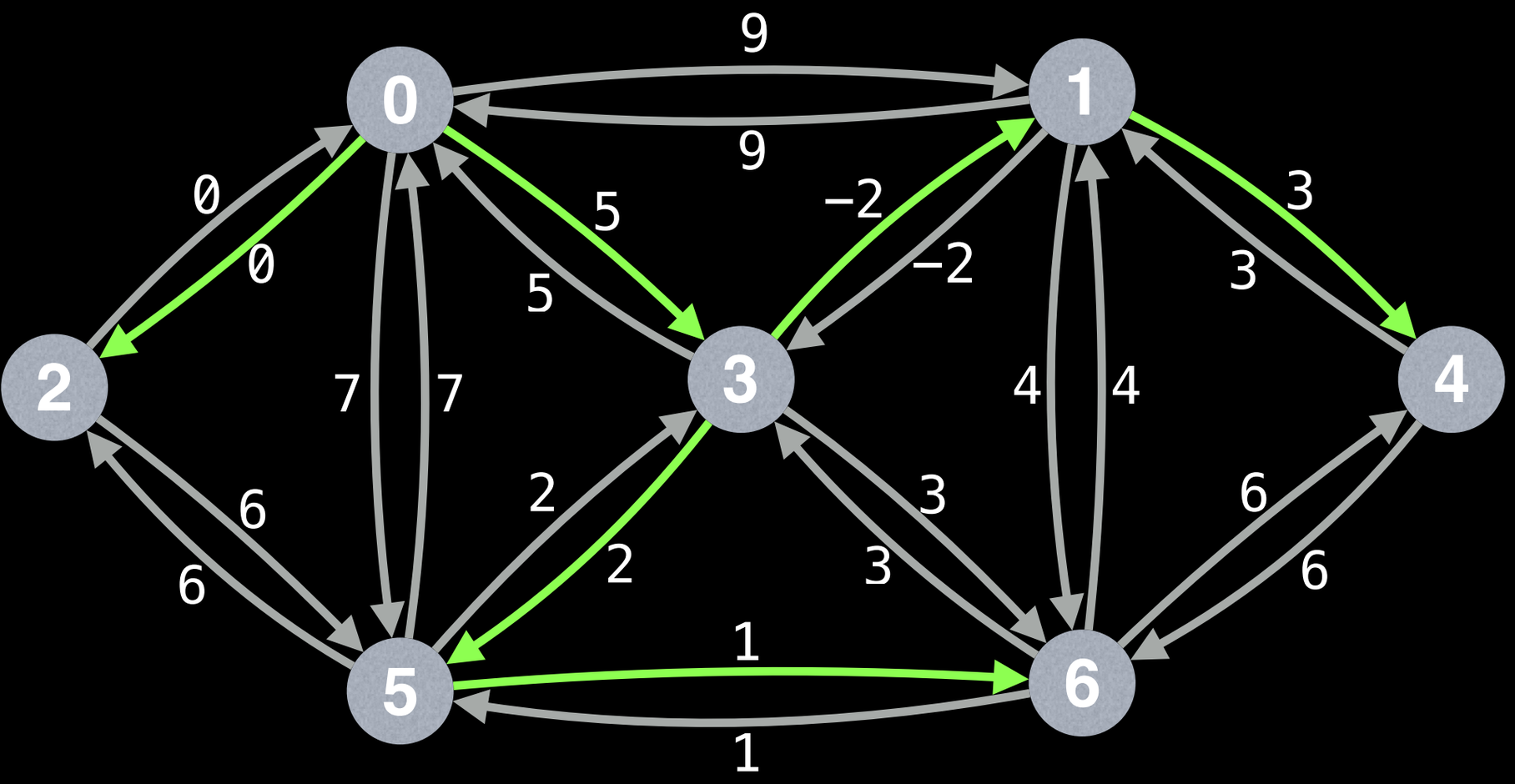
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~



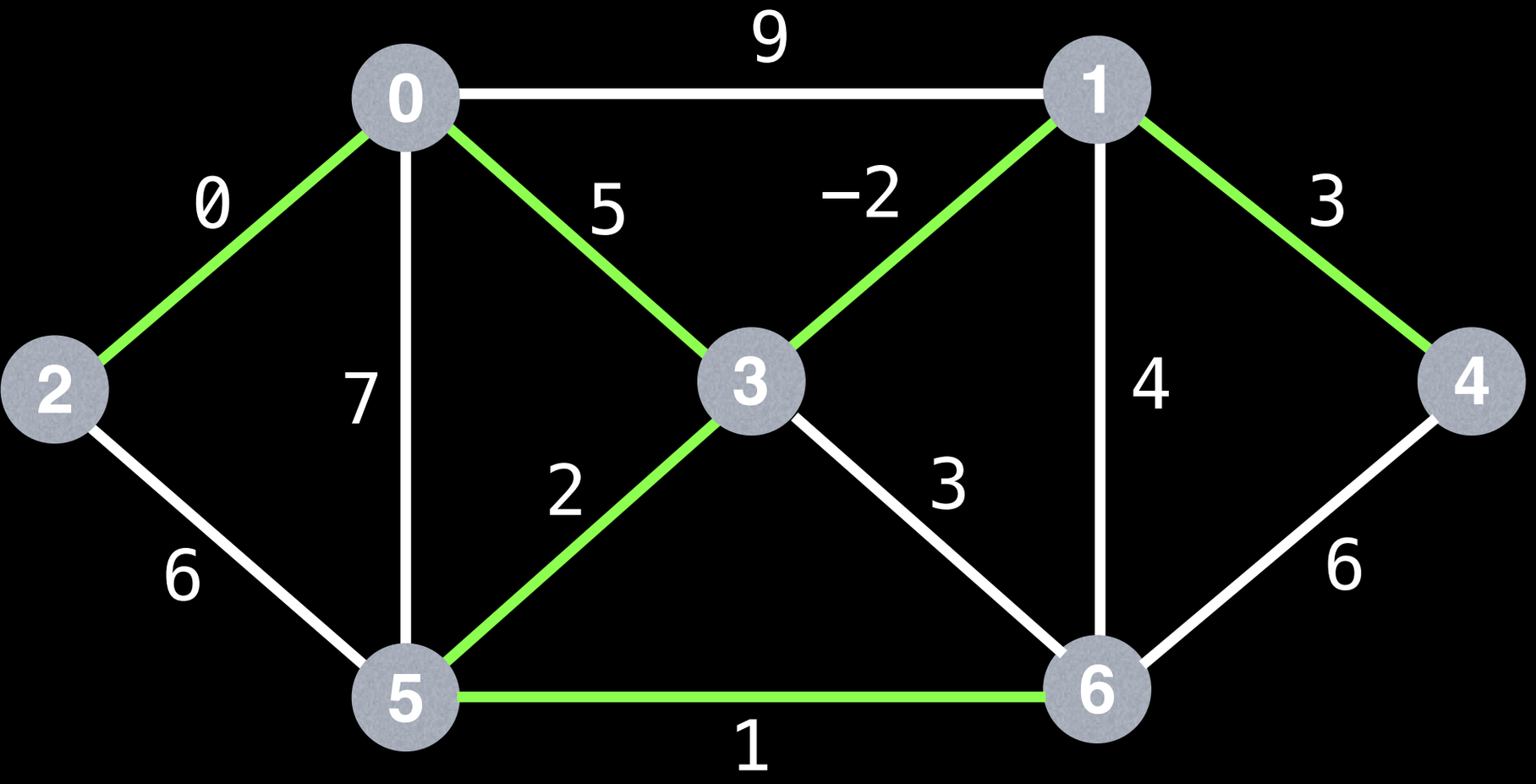
(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~

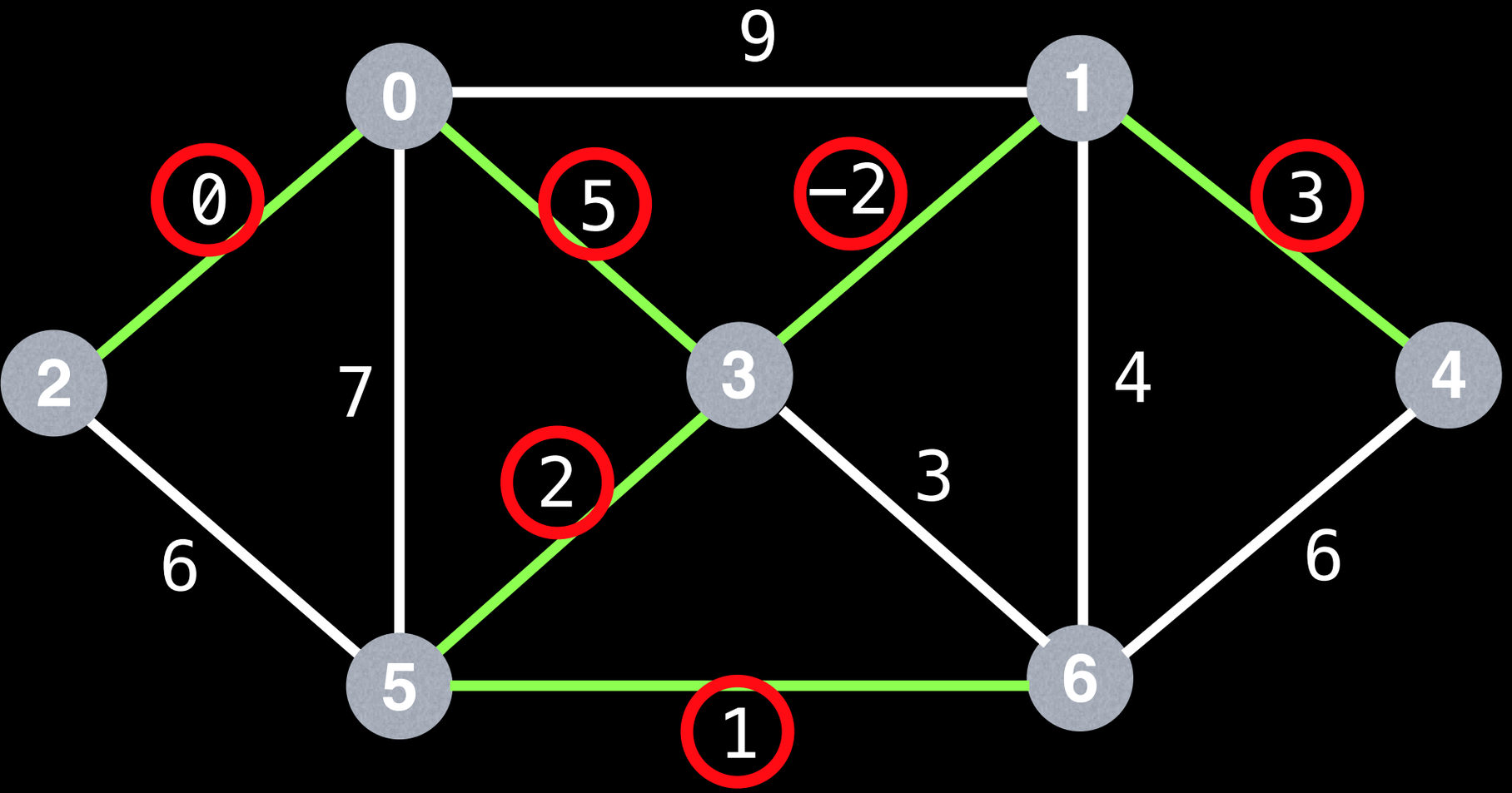


(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~



If we collapse the graph back into the undirected edge view it becomes clear which edges are included in the MST.



The MST cost is:

0 + 5 + 2 + 1 + −2 + 3 = 9

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.
```

```
g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

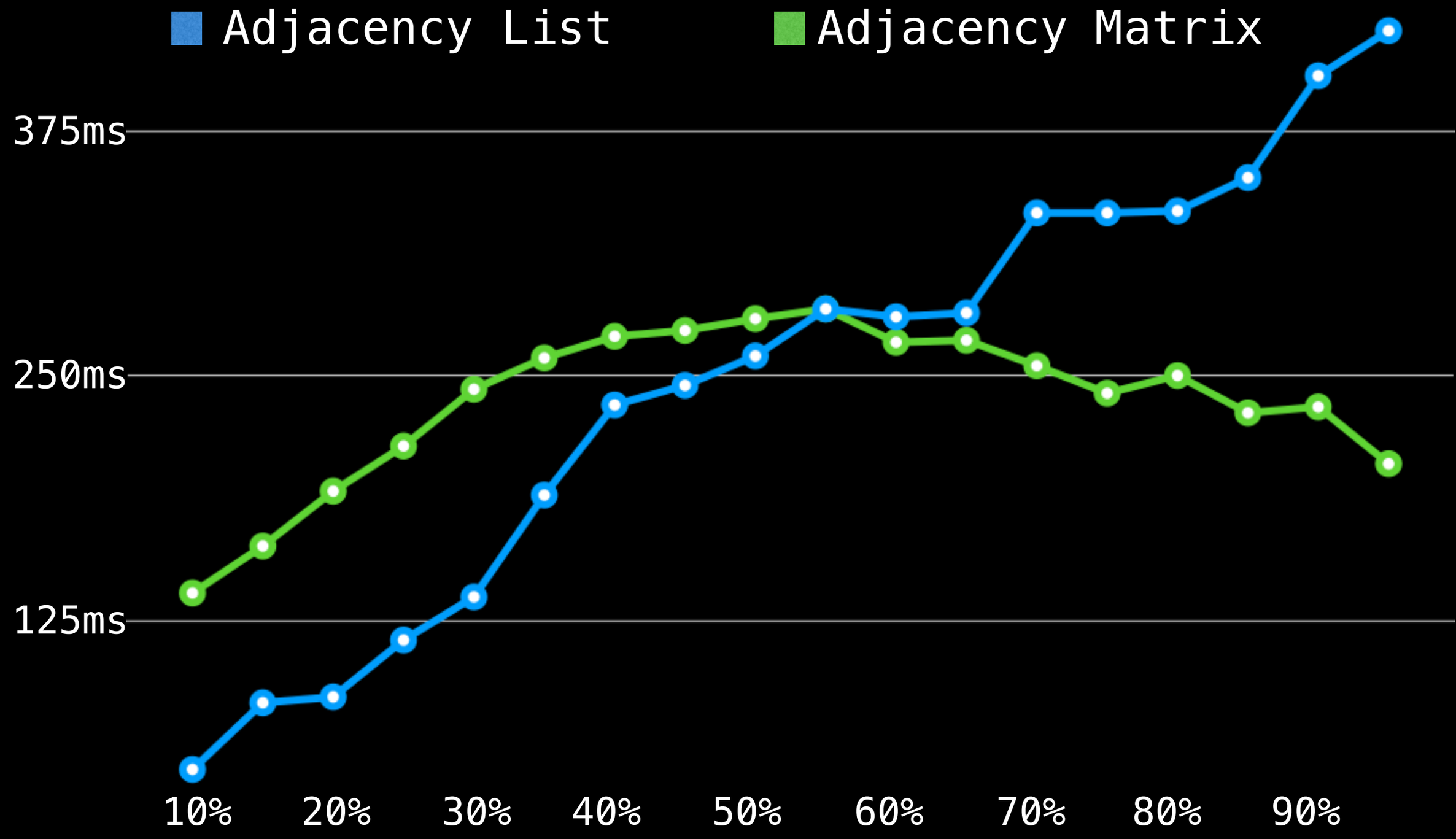
```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Graph edge density analysis



X-axis: Edge density percentage on graph with 5000 nodes.

Y-axis: time required to find MST in milliseconds.

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.
```

```
g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.
```

```
visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

if edgeCount != m:
  return (null, null) # No MST exists!

return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

if edgeCount != m:
  return (null, null) # No MST exists!

return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

if edgeCount != m:
  return (null, null) # No MST exists!

return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```