Algorithms and Data Structures

12th Lecture: Graph Algorithms II

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

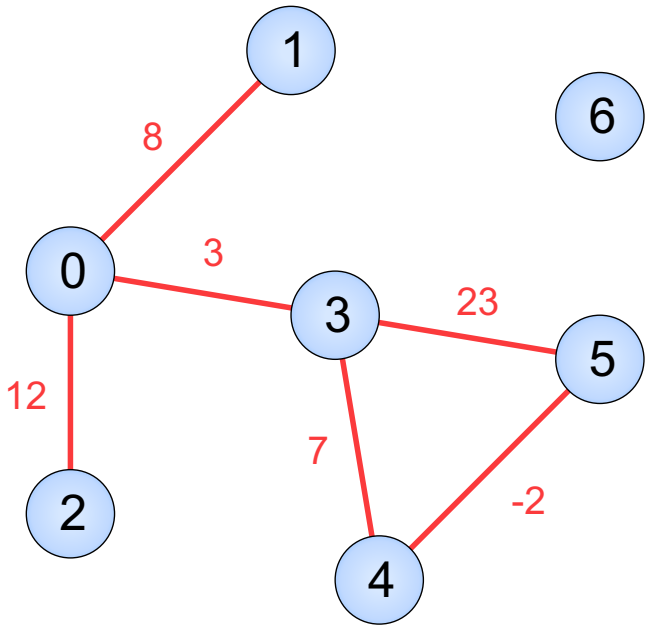
Last Updated: 2025/01/26

Outline

- Weighted Graphs
- Minimum Spanning Tree
- Single-Source Shortest Paths

Adjacency Matrix Representation: Undirected Weighted Graph

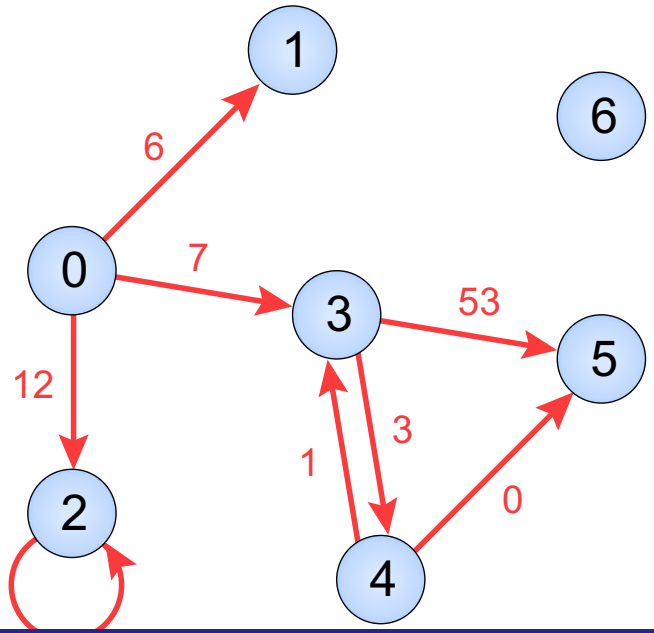
- For a given connected, undirected graph $G = (V, E)$ where for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v.

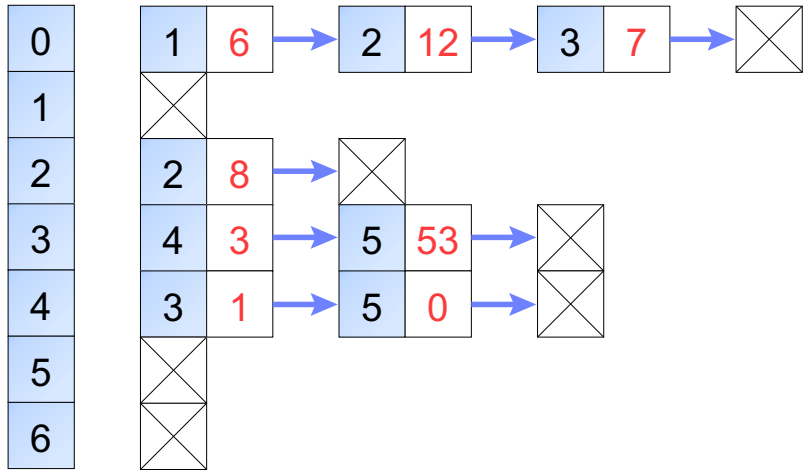


i \ j	0	1	2	3	4	5	6
0	∞	8	12	3	∞	∞	∞
1	8	∞	∞	∞	∞	∞	∞
2	12	∞	∞	∞	∞	∞	∞
3	3	∞	∞	∞	7	23	∞
4	∞	∞	∞	7	∞	-2	∞
5	∞	∞	∞	23	-2	∞	∞
6	∞	∞	∞	∞	∞	∞	∞

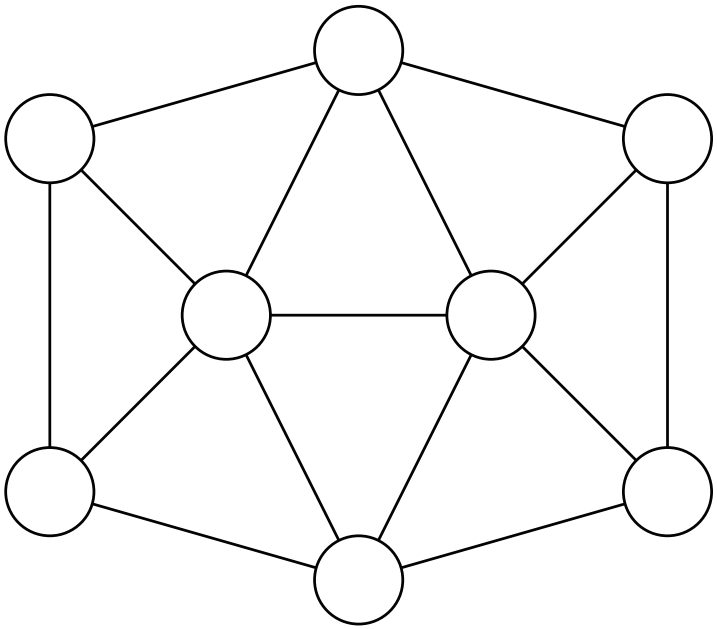
Adjacency List Representation: Directed Weighted Graph

- For a given connected, directed graph $G = (V, E)$ where for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v.

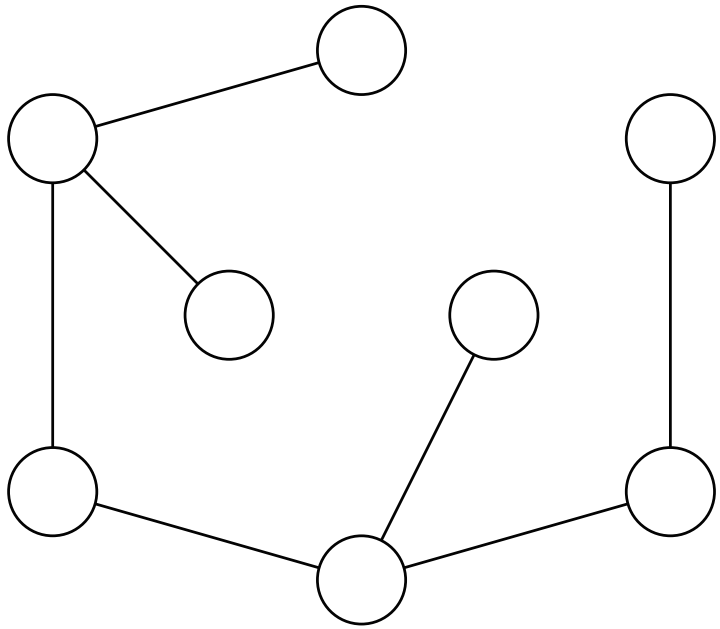




Spanning Tree

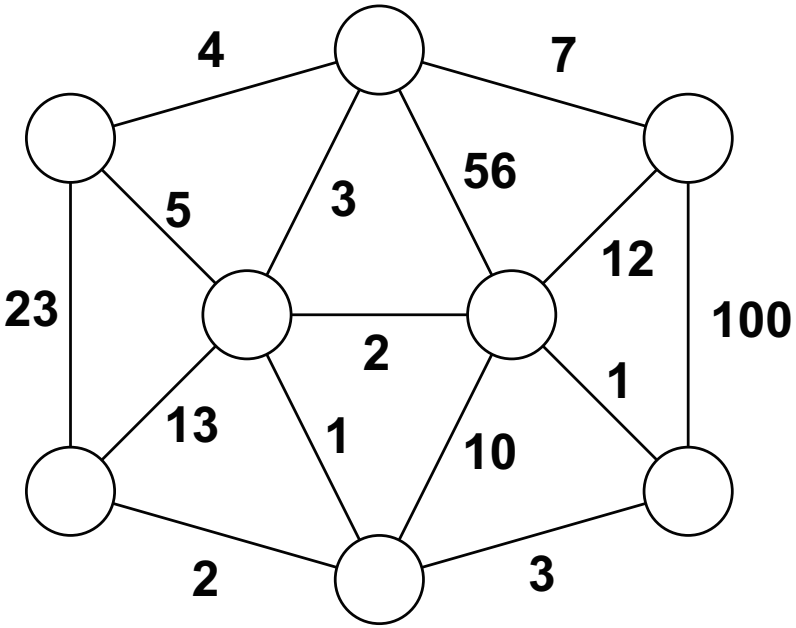


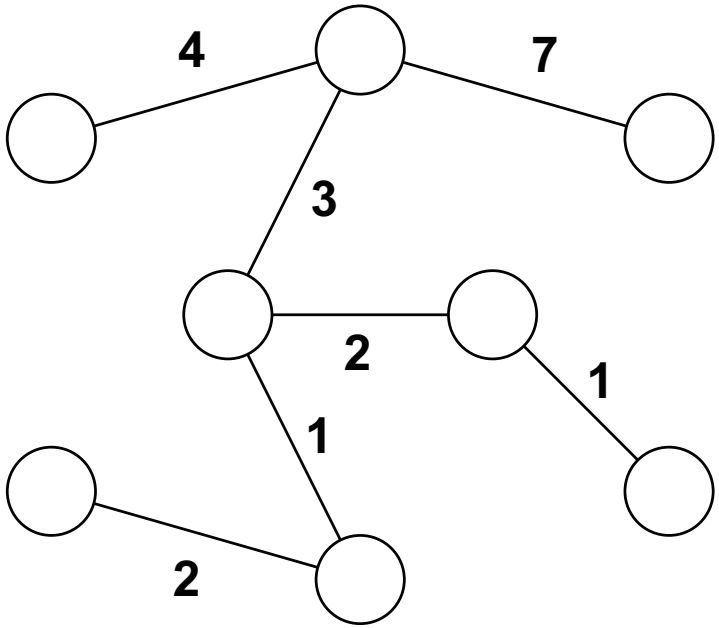
(a)



(b)

Minimum Spanning Tree (1)





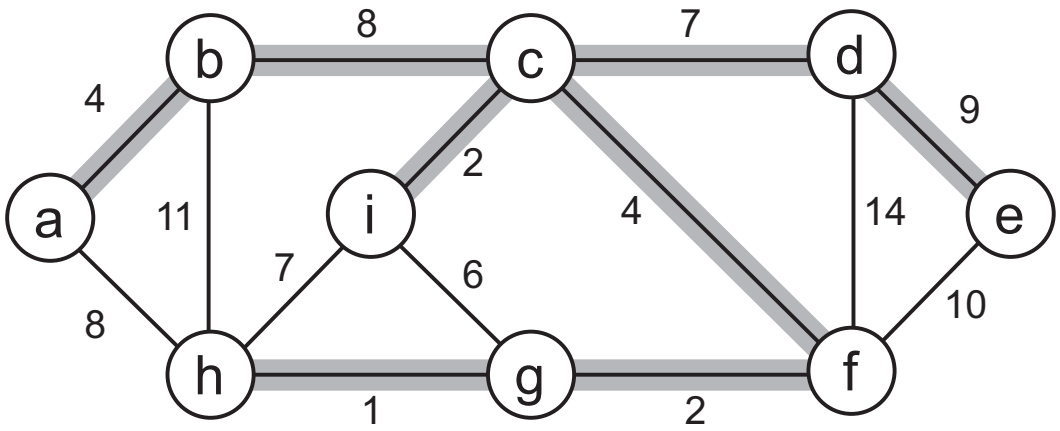
Minimum Spanning Tree (2)

- We wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

  is minimized.
- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree.
- We call the problem of determining the tree T the minimum spanning-tree (MST) problem.

Minimum Spanning Tree (3)



- A minimum spanning tree for a connected graph.
- The weights on edges are shown, and the edges in a minimum spanning tree are shaded.
- The total weight of the tree is 37.

Growing a Minimum Spanning Tree (1)

- Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow R$, and we wish to find a MST for G.
- We use a greedy approach. It is captured by the "generic" algorithm, which grows the minimum spanning tree one edge at a time.
- The algorithm manages a set of edges T, maintaining the following loop invariant:
  - Prior to each iteration, T is a subset of some minimum spanning tree.

Growing a Minimum Spanning Tree (2)

- At each step, we determine an edge (u, v) that can be added to T without violating this invariant, in the sense that $T \cup \{(u, v)\}$ is also a subset of a MST.
- We call such an edge a safe edge for T, since it can be safely added to T while maintaining the invariant.

Generic Algorithm for MST

```
genericMST(G, w)
    T = empty
    while T does not form a spanning tree
        find an edge (u, v) that is safe for T
        T = T ∪ {(u, v)}
    return T
```

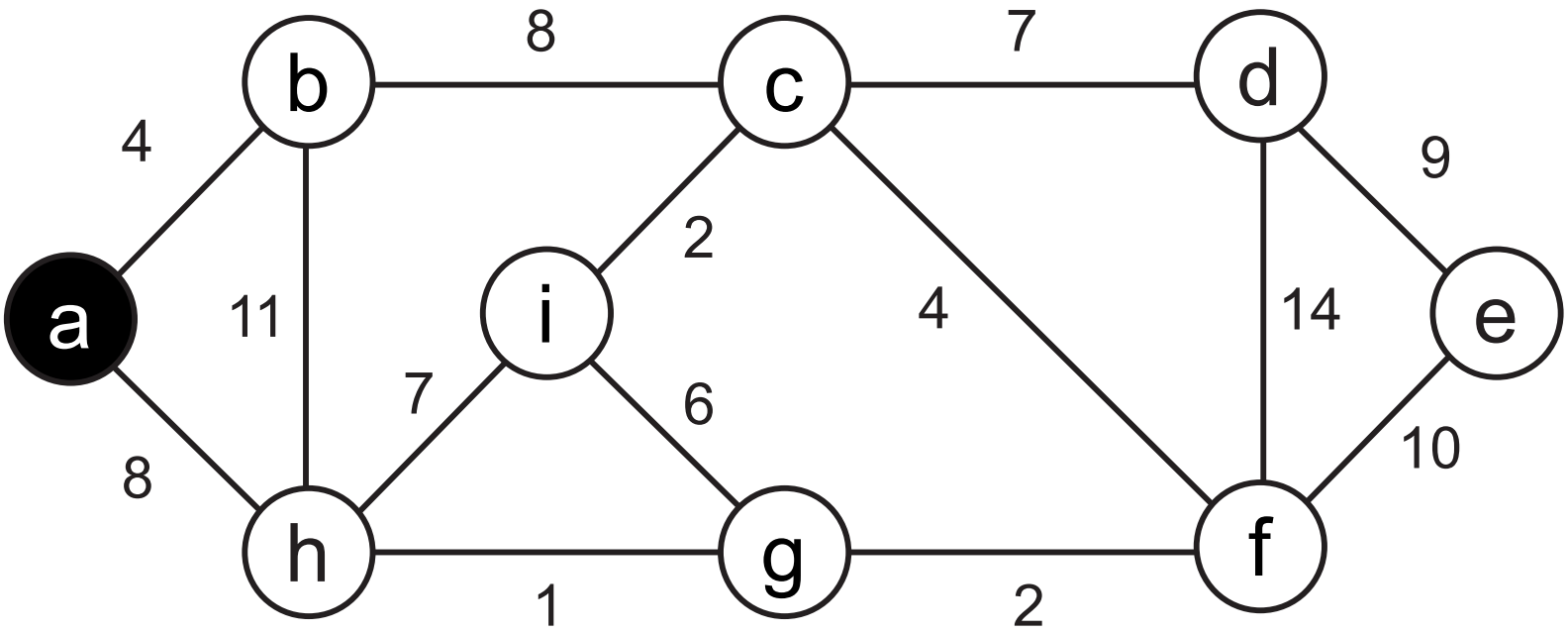
This is a "generic" algorithm which can be a basis to implement different algorithms.

- Prim's Algorithm
- Kruskal's Algorithm

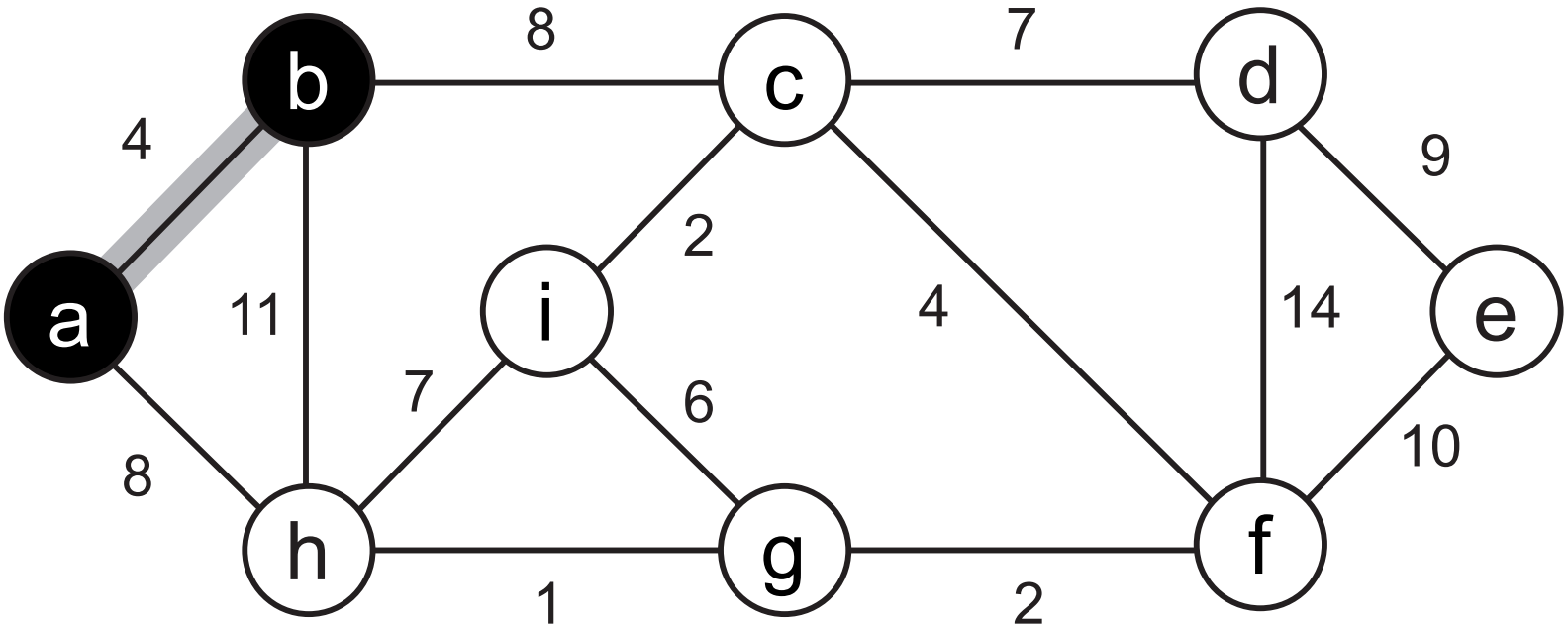
Prim's Algorithm for MST

- Prim's algorithm is a special case of the generic MST algorithms.
- Prim's algorithm has the property that the edges in the set T always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.
- At each step, a light edge is added to the tree T that connects T to an isolated vertex of $G_T = (V, T)$. This rule adds only edges that are safe for T.
- This strategy is greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

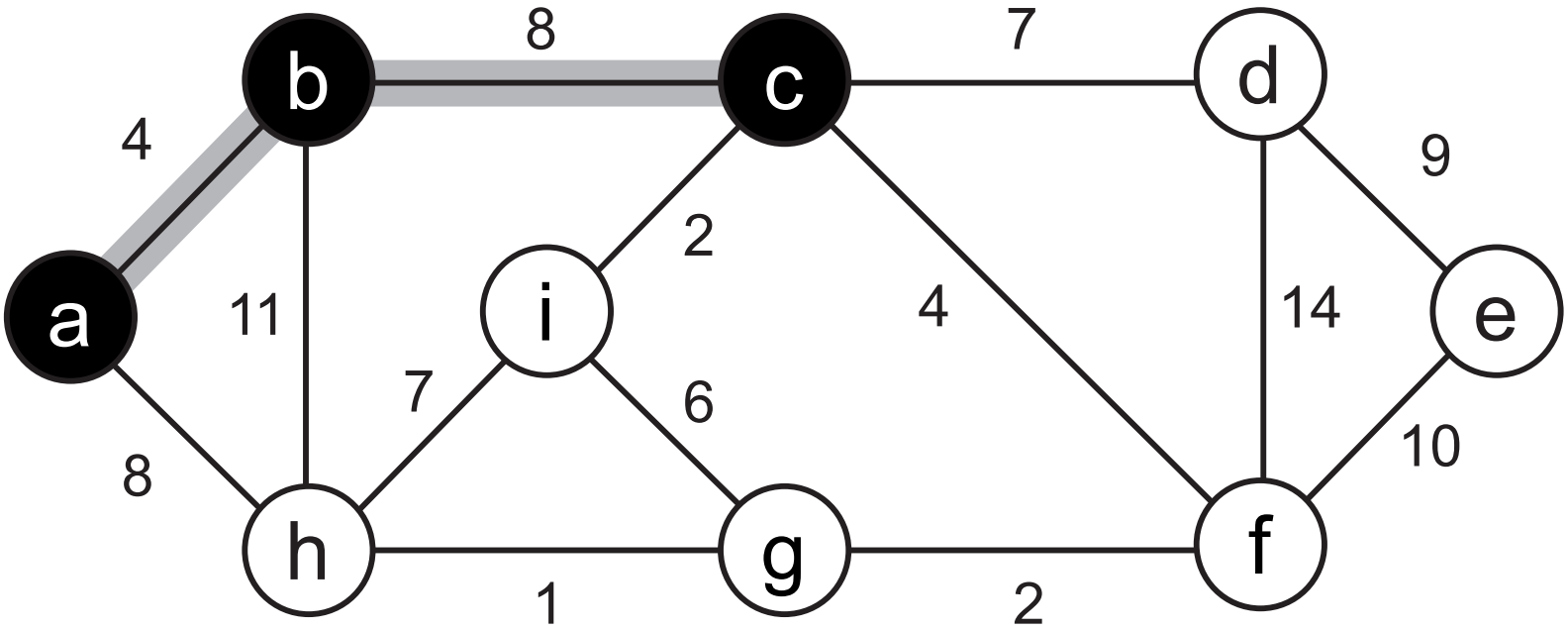
Prim's Algorithm based on the Generic Algorithm (1)



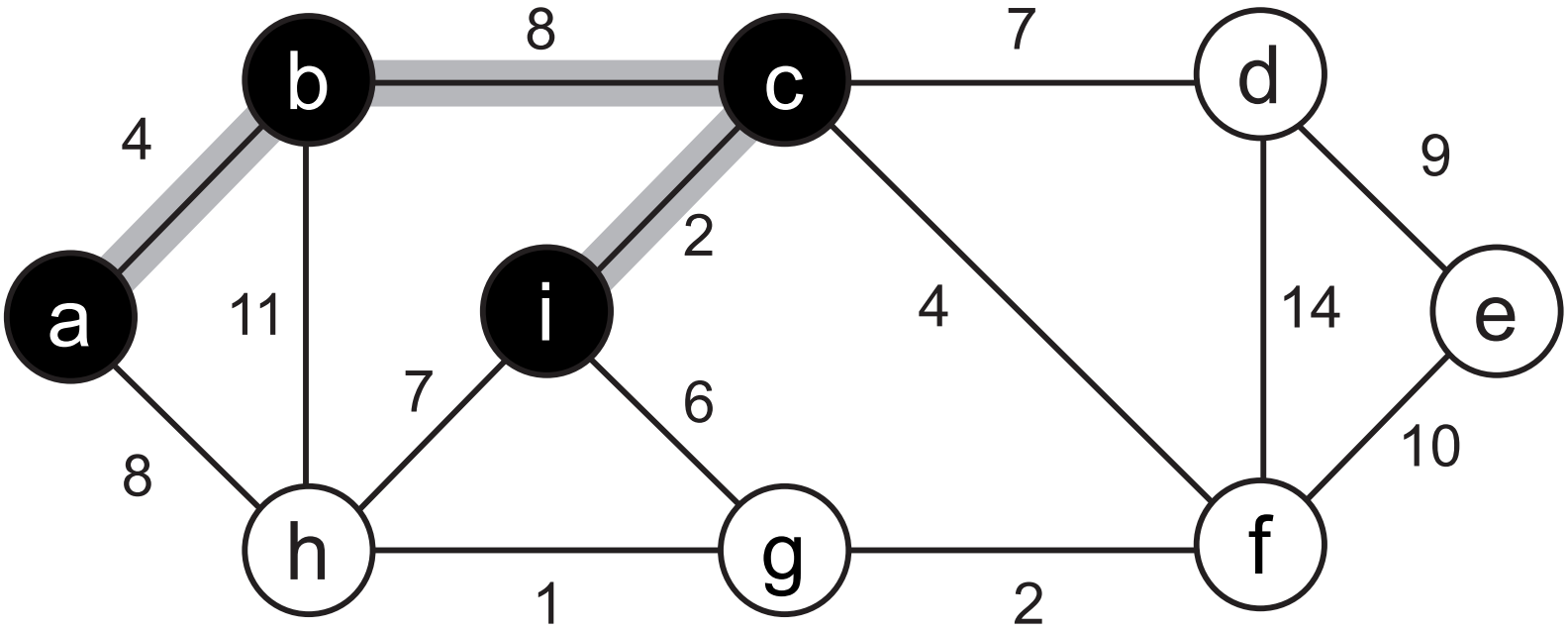
Prim's Algorithm based on the Generic Algorithm (2)



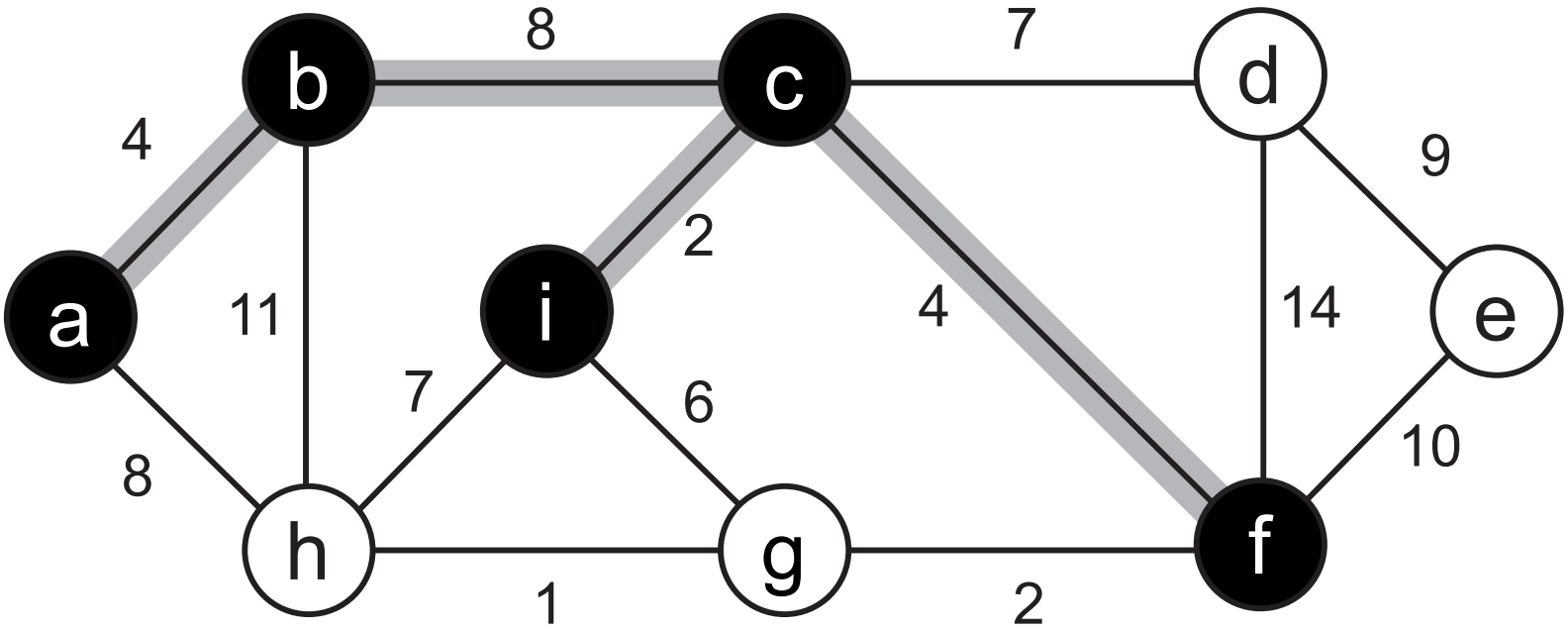
Prim's Algorithm based on the Generic Algorithm (3)



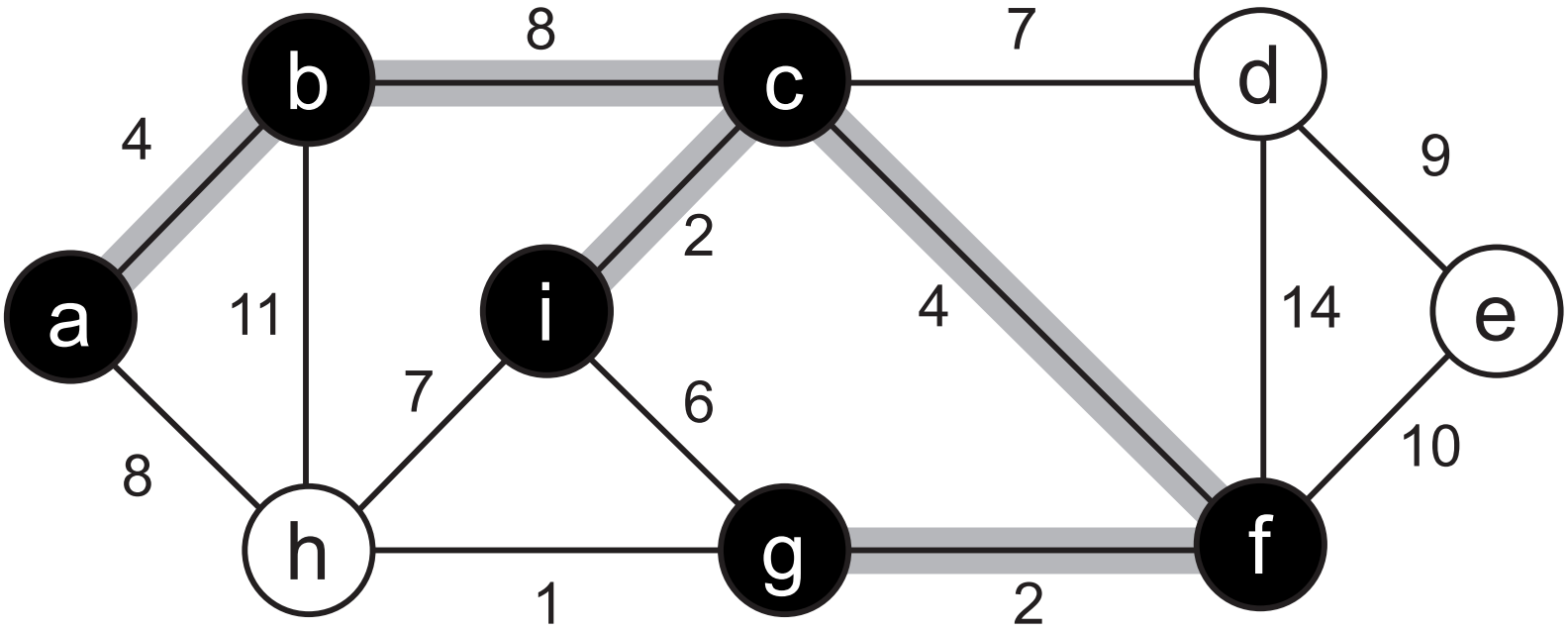
Prim's Algorithm based on the Generic Algorithm (4)



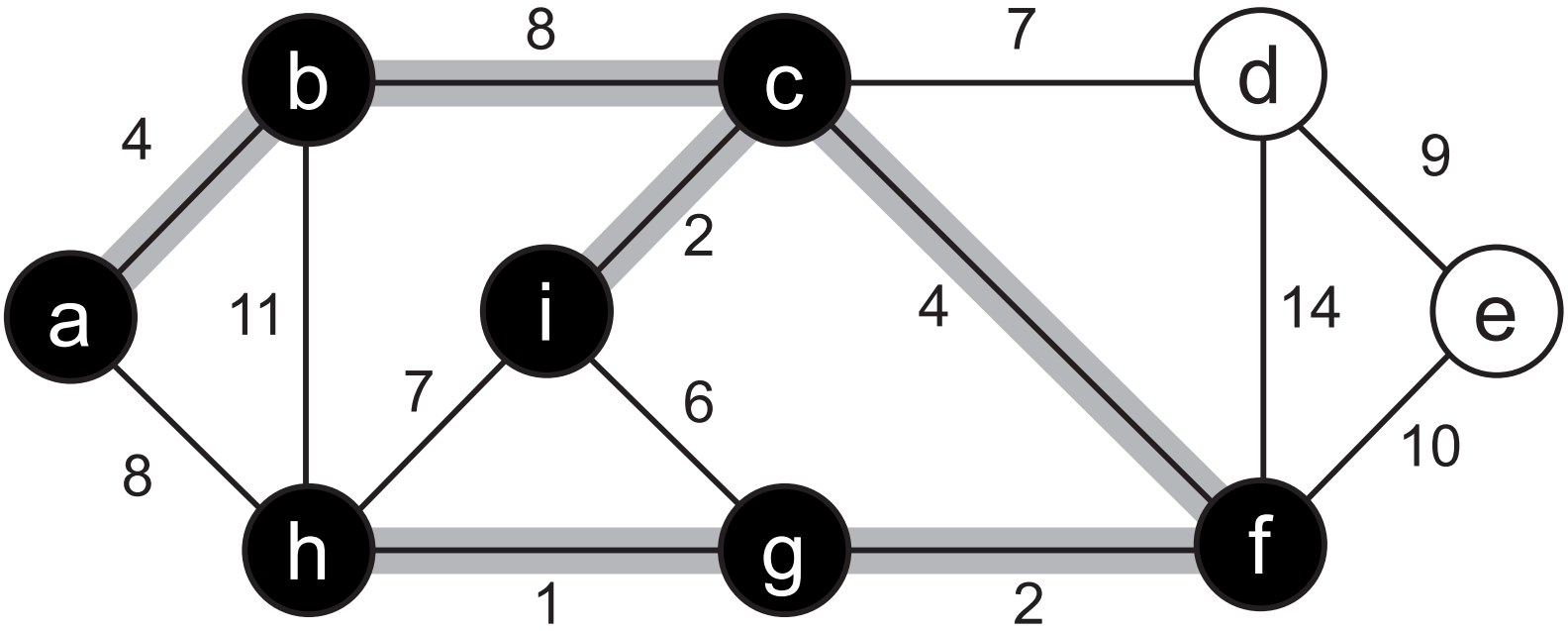
Prim's Algorithm based on the Generic Algorithm (5)



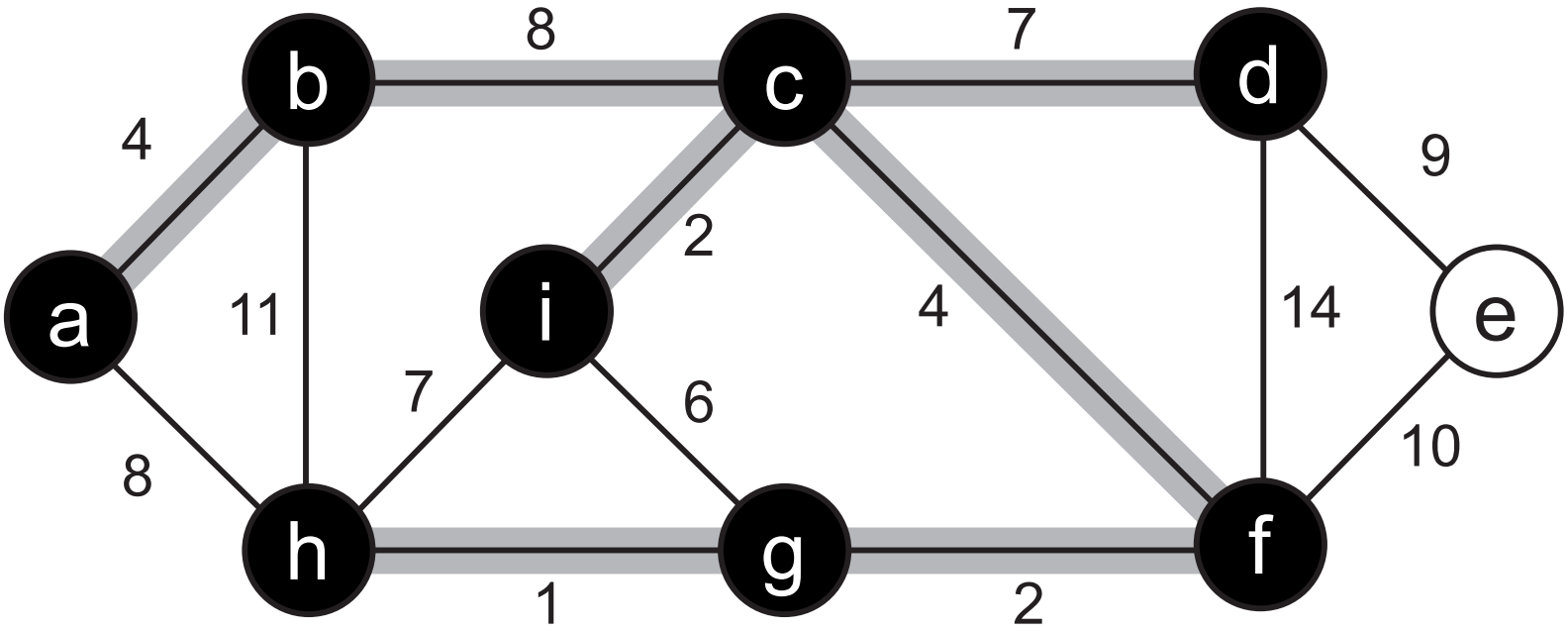
Prim's Algorithm based on the Generic Algorithm (6)



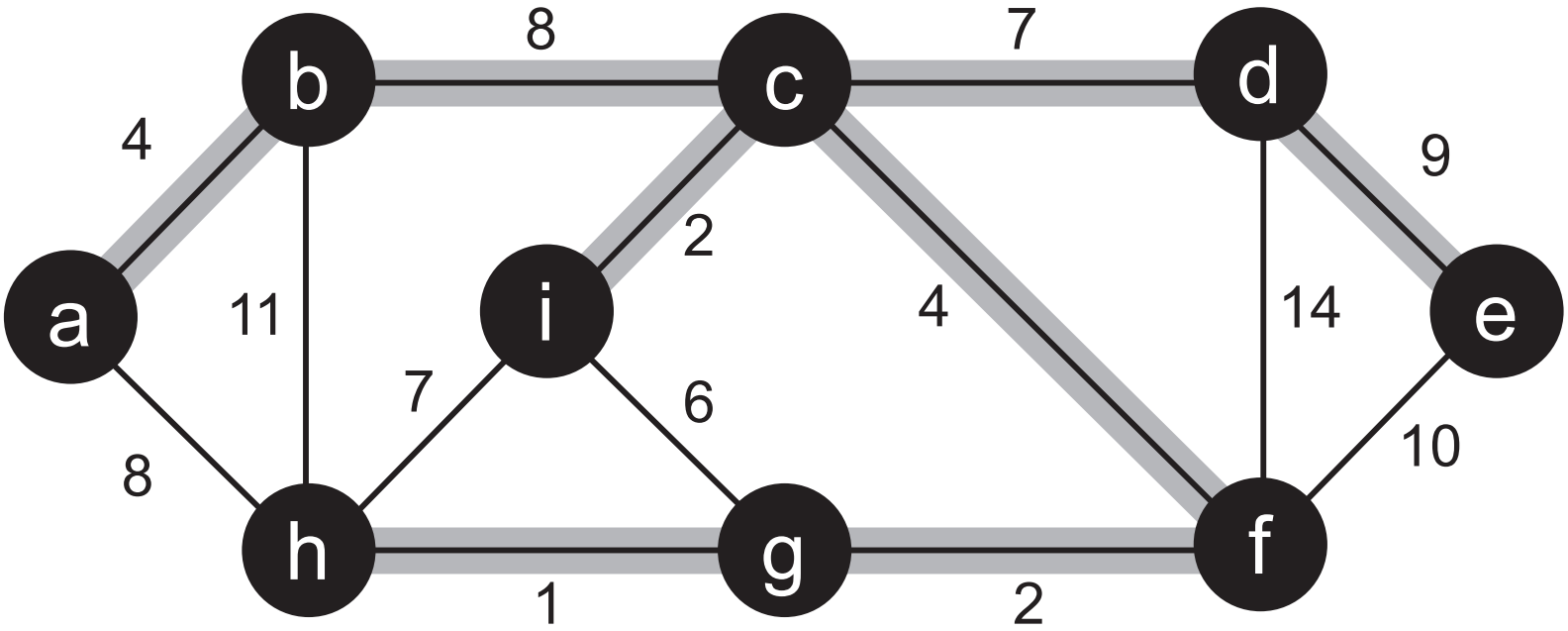
Prim's Algorithm based on the Generic Algorithm (7)



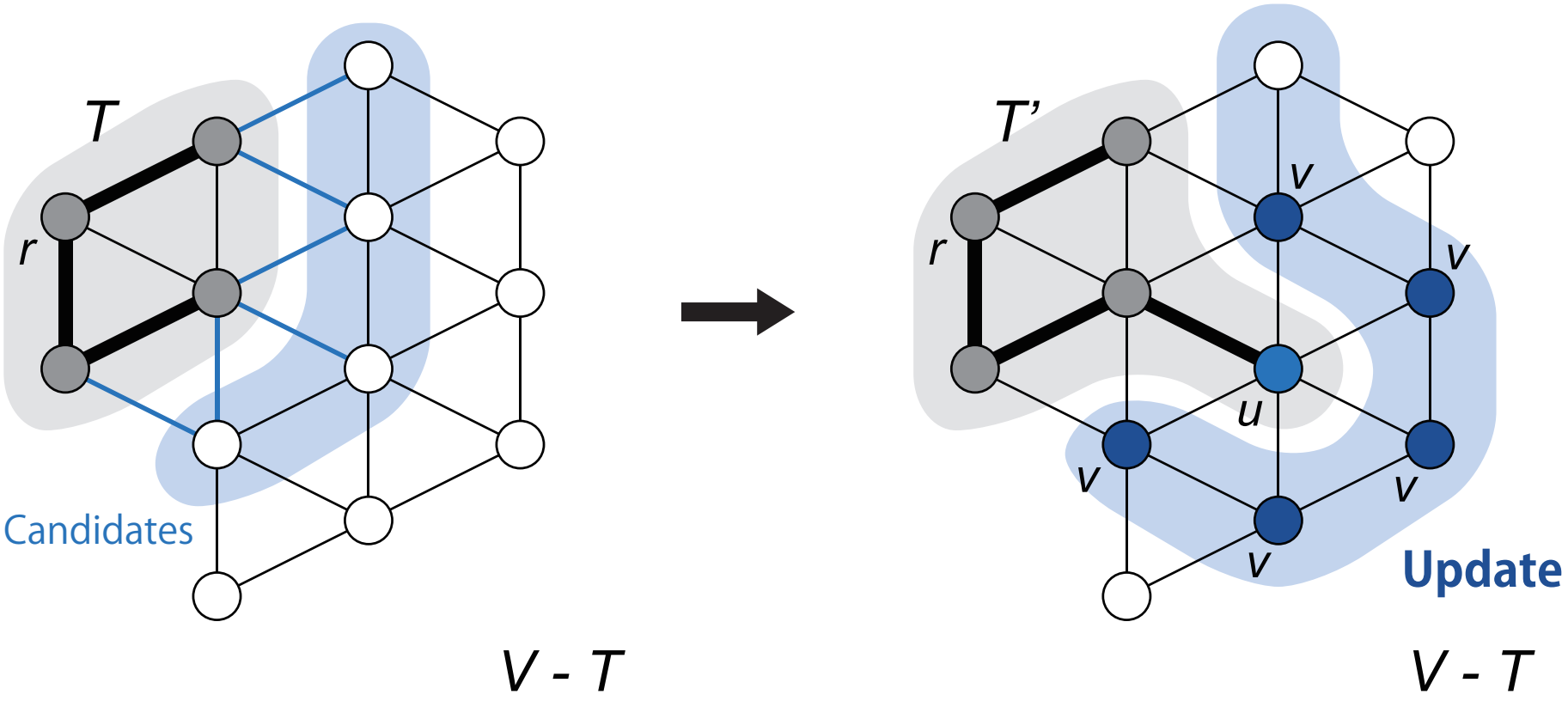
Prim's Algorithm based on the Generic Algorithm (8)



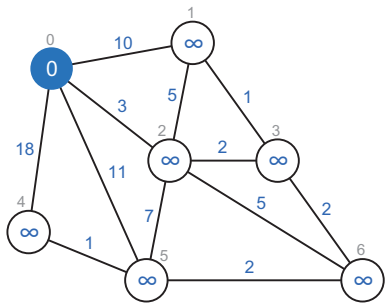
Prim's Algorithm based on the Generic Algorithm (9)

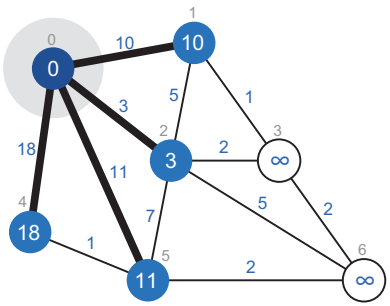


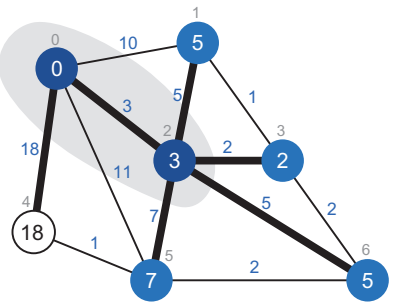
Prim's Algorithm: Implementation

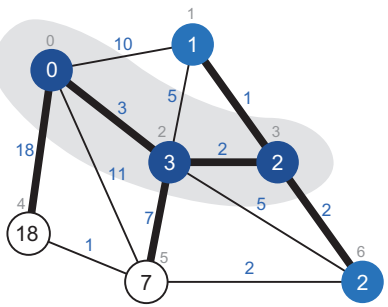


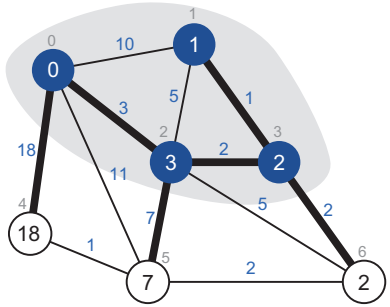
Prim's Algorithm

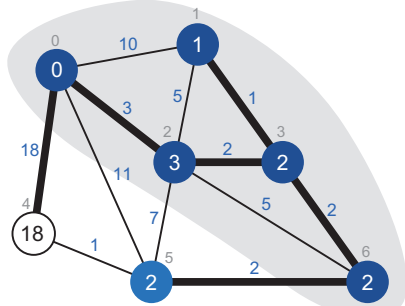
1


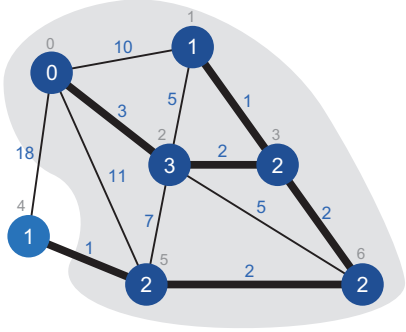
2


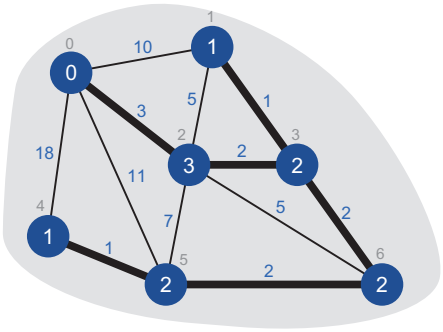
3


4


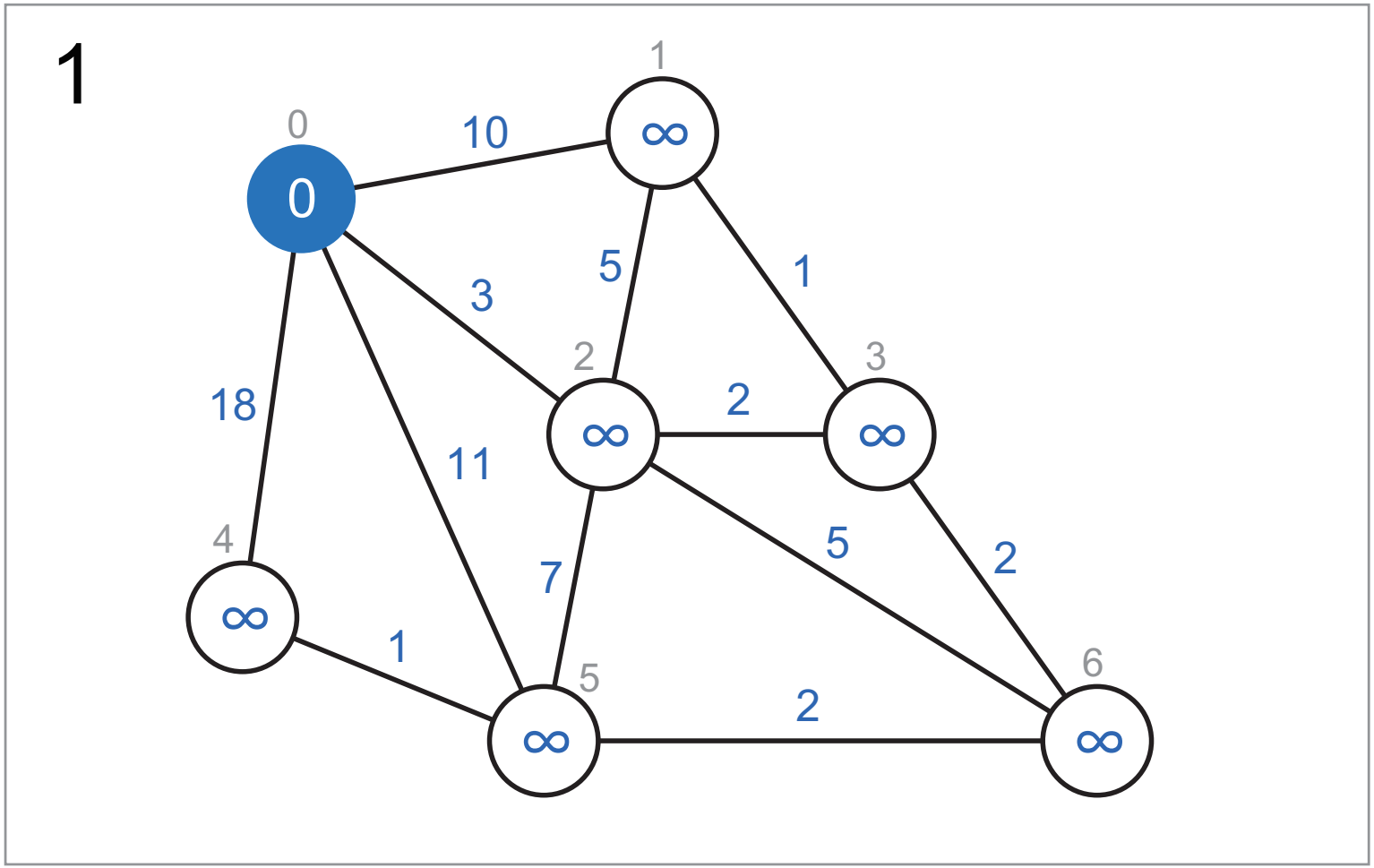
5


6


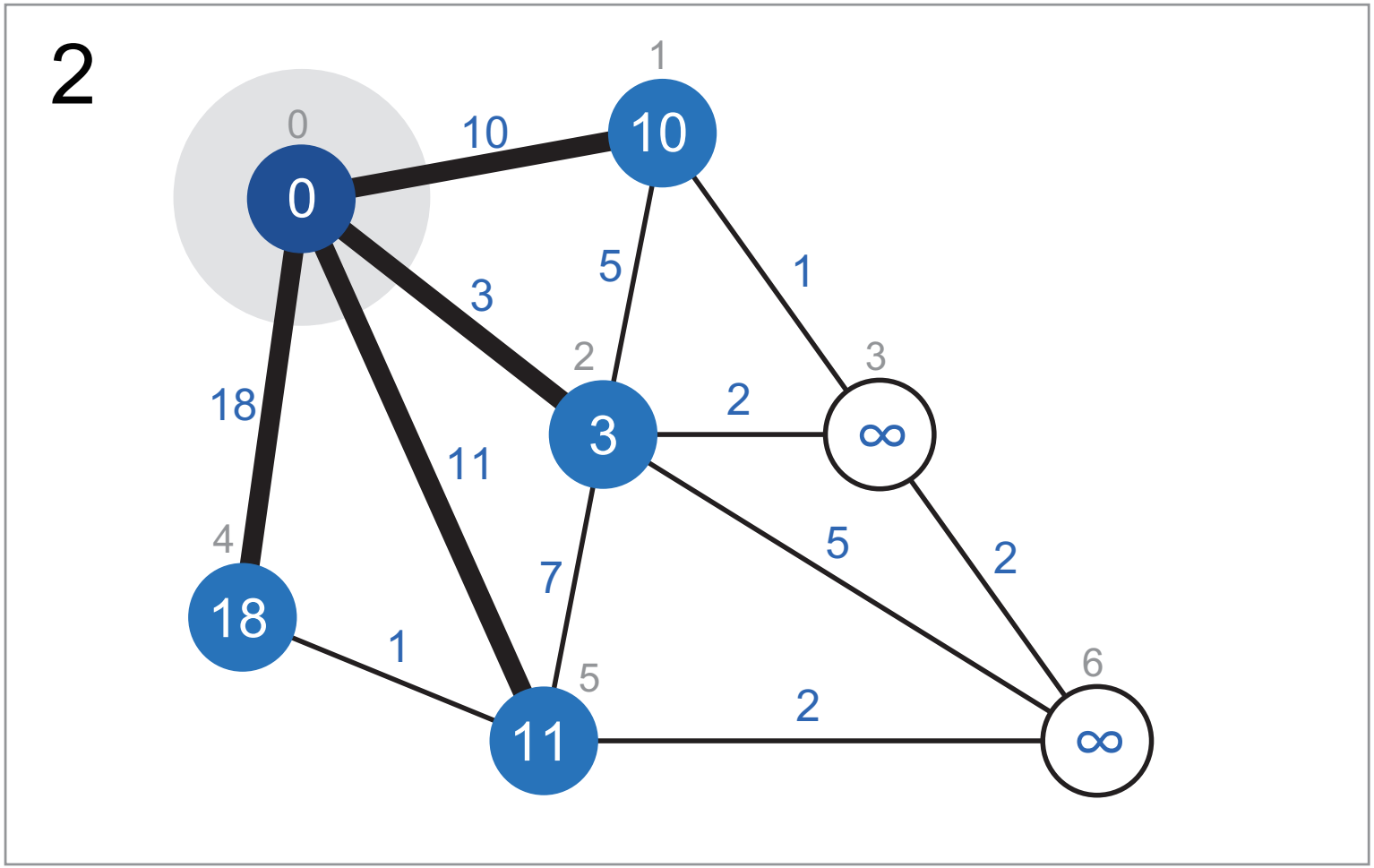
7


8


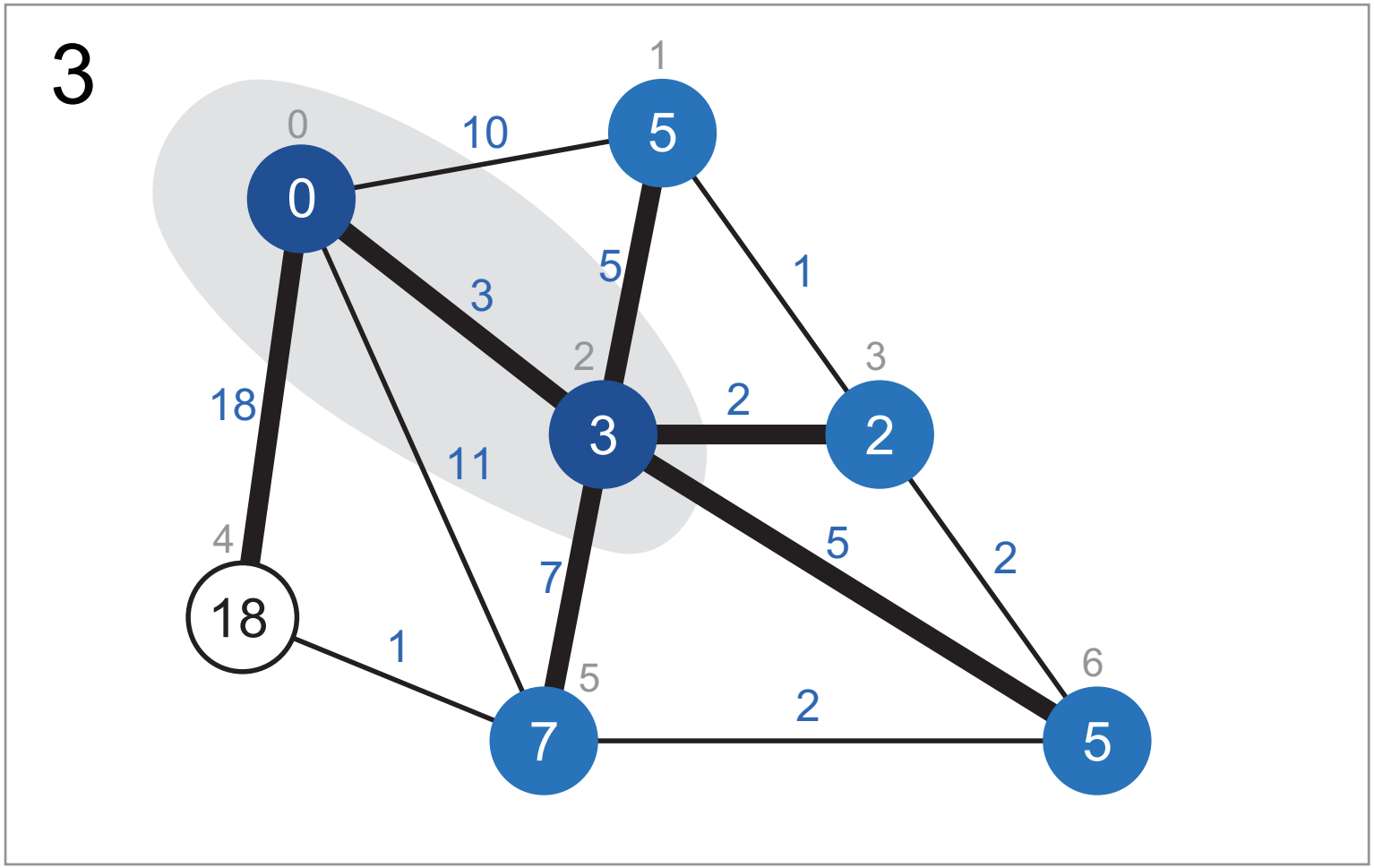
Prim's Algorithm (1)



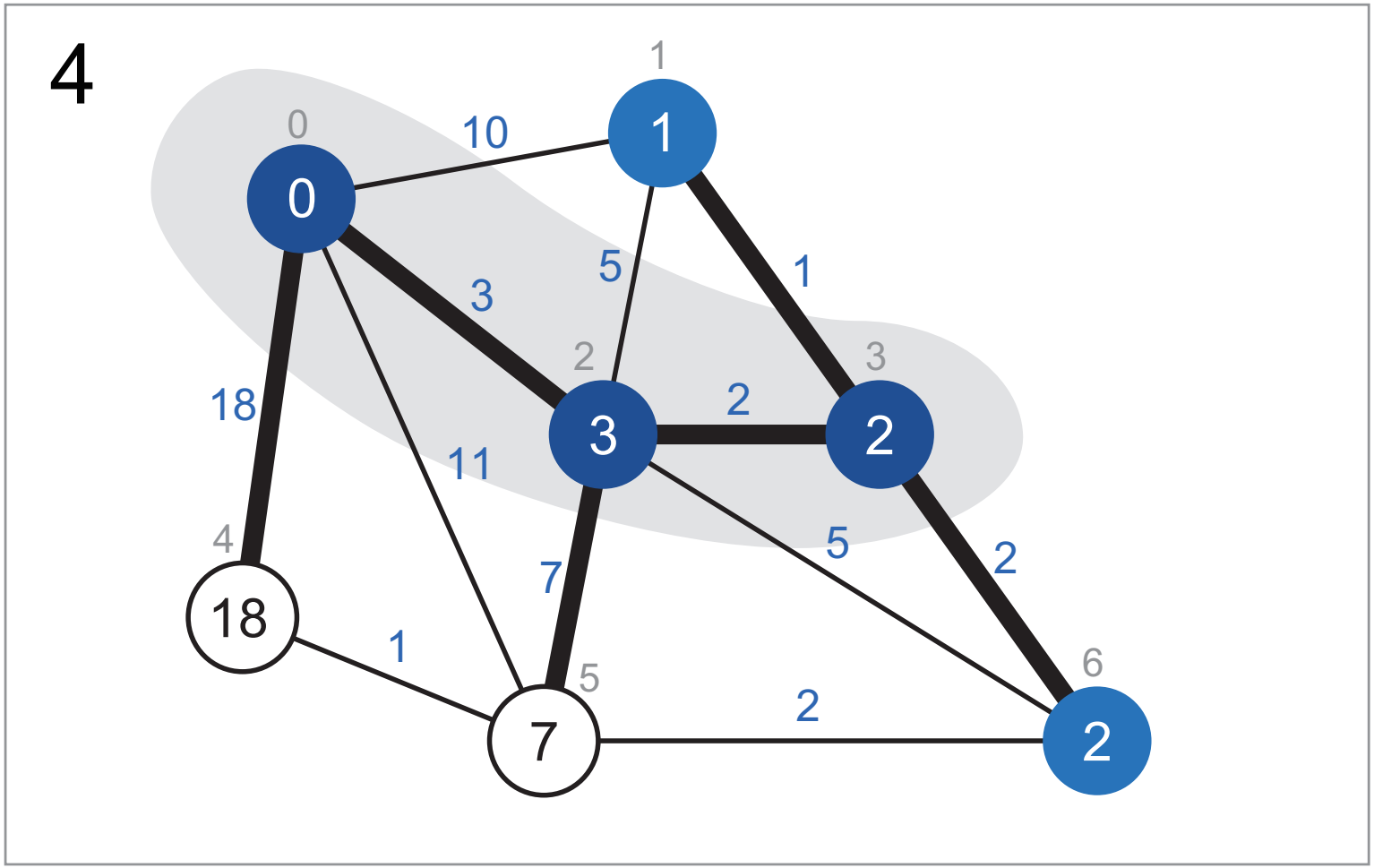
Prim's Algorithm (2)



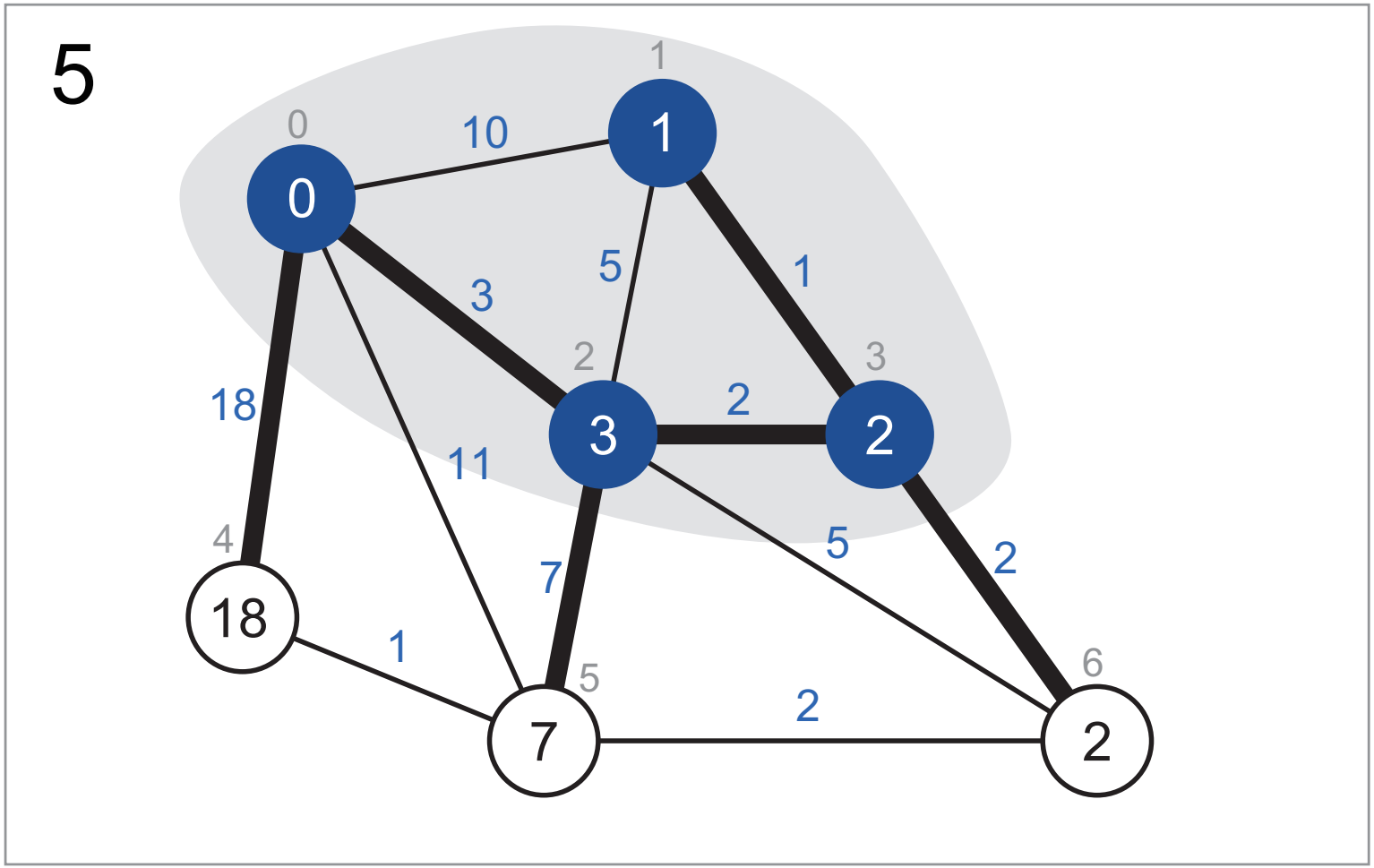
Prim's Algorithm (3)



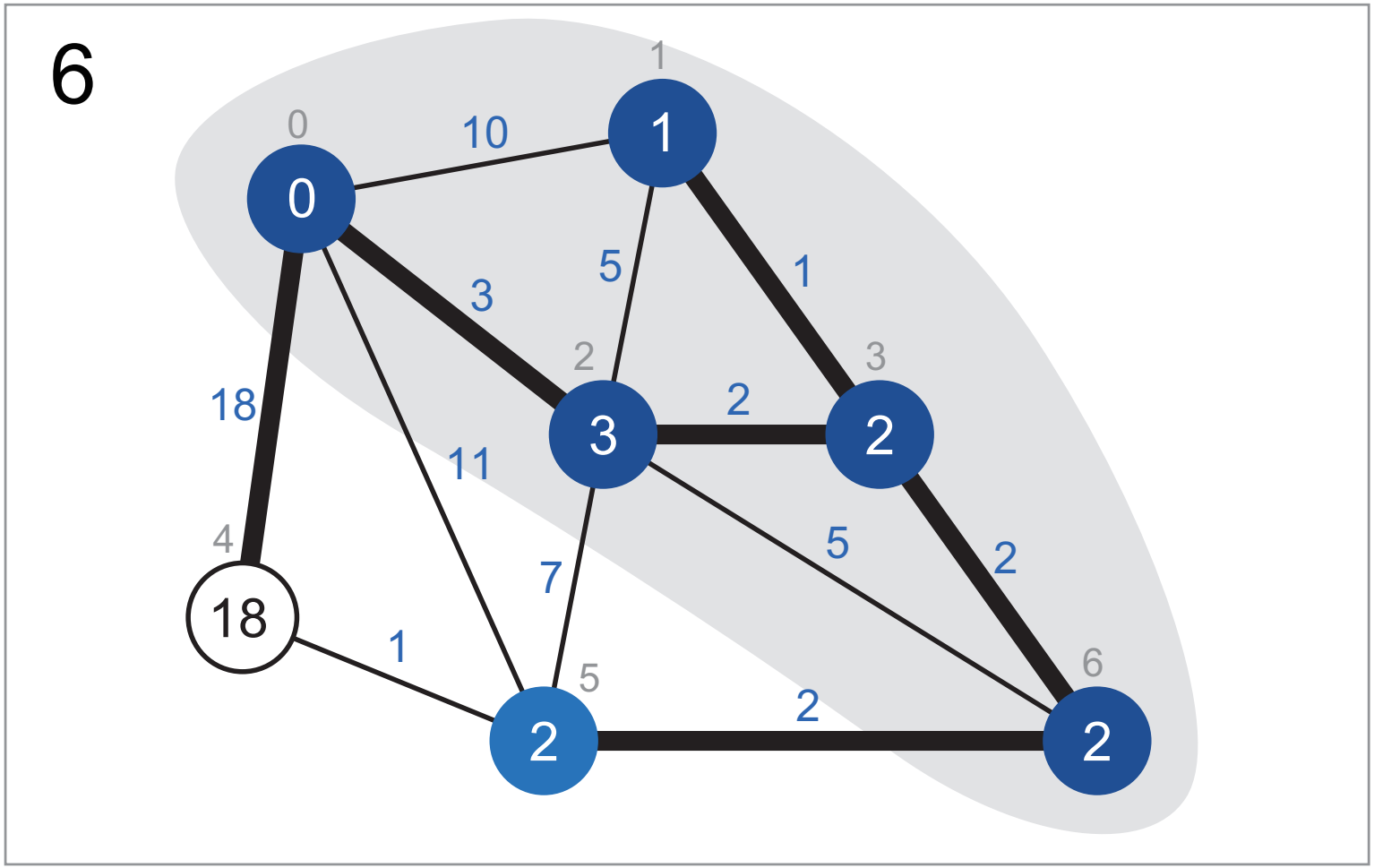
Prim's Algorithm (4)



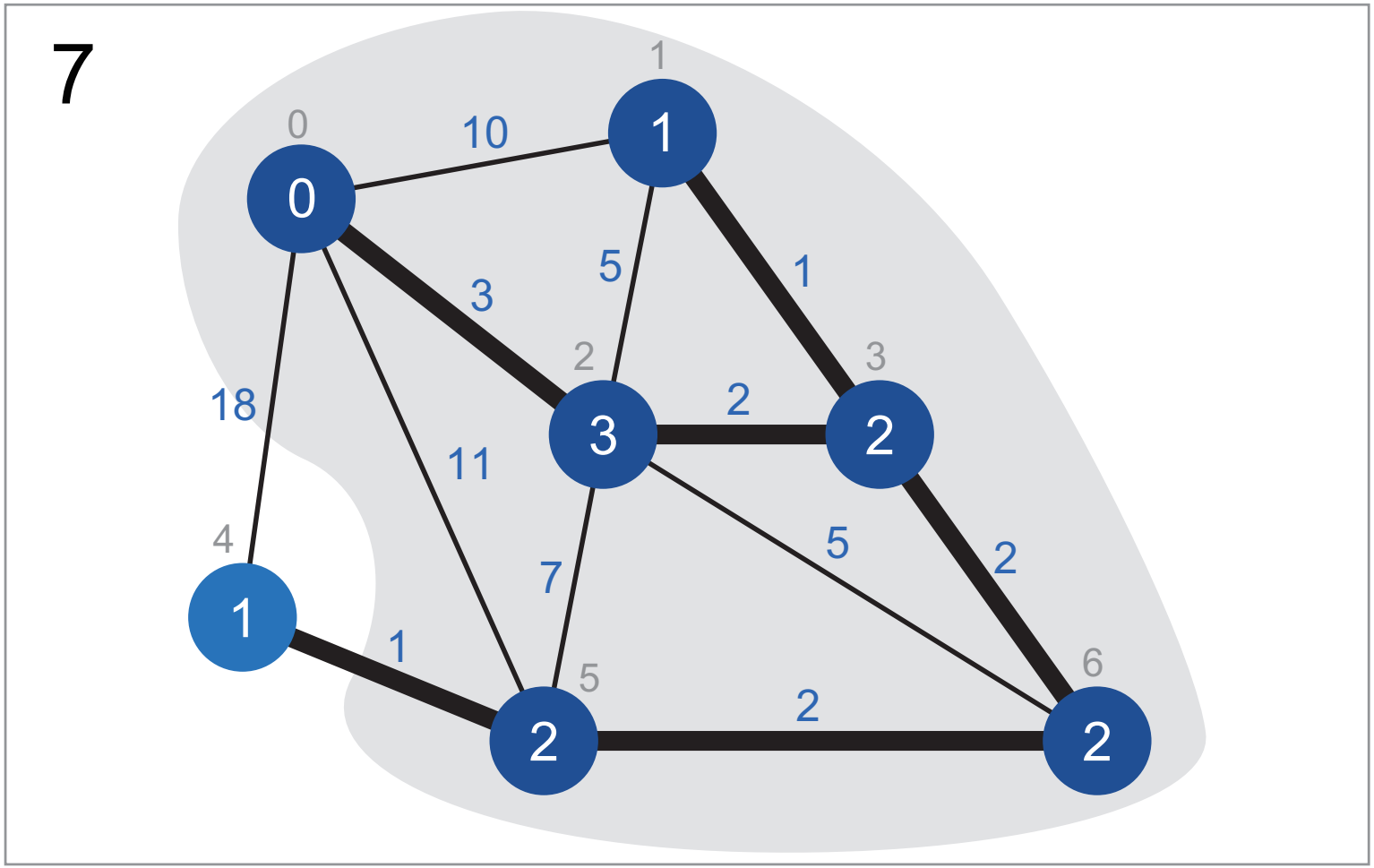
Prim's Algorithm (5)



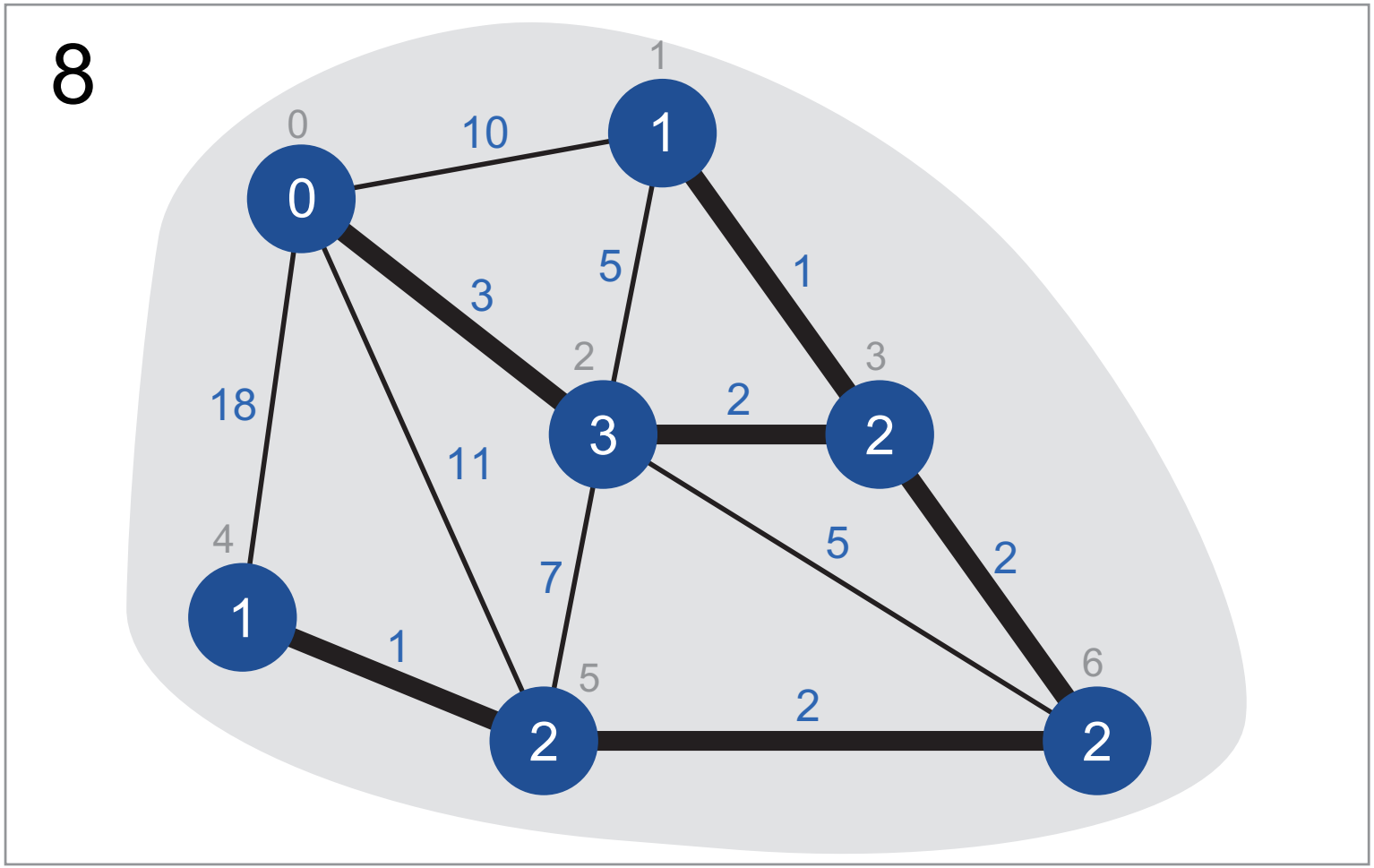
Prim's Algorithm (6)



Prim's Algorithm (7)



Prim's Algorithm (8)



Prim's Algorithm: $O(|V|^2)$ Implementation (1)

```
prim(G, w, r)
    for each u in G
        d[u] = INF
        pi[u] = NIL
        color[u] = WHITE

    d[r] = 0
```

Prim's Algorithm: $O(|V|^2)$ Implementation (2)

```
while true
    mincost = INF
    for each i in G
        if color[i] != BLACK and d[i] < mincost
            mincost = d[i]
            u = i

    if mincost == INF
        break

    color[u] = BLACK

    for each v in Adj[u]
        if color[v] != BLACK and w(u, v) < d[v]
            pi[v] = u
            d[v] = w(u, v)
```

Prim's algorithm: $((|V| + |E|) \log |V|)$ Implementation (1)

```
prim(G, w, r)
    for each u in G
        d[u] = INF
        pi[u] = NIL
```

Prim's algorithm: $((|V| + |E|) \log |V|)$ Implementation (2)

- For an efficient implementation, we use a min-heap H of vertices, keyed by their d values.

```
d[r] = 0
H = buildMinHeap(V)
while H is not empty
    u = H.extractMin()
    color[u] = BLACK
    for each v in Adj[u]
        if color[v] != BLACK
            if w(u, v) < d[v]
                pi[v] = u
                d[v] = w(u, v)
                heapDecreaseKey(H, v, w(u, v))
```

Single-Source Shortest Paths (1)

- We are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$ mapping edges to real-valued weights.
- The weight of path $p = \{v_0, v_1, ..., v_k\}$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Single-Source shortest Paths (2)

- We define the shortest-path weight from u to v by

$$\delta(u,v) = \begin{cases} min\{w(p) : u \rightarrow v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$.
- Given a graph $G = (V,E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

Initialize

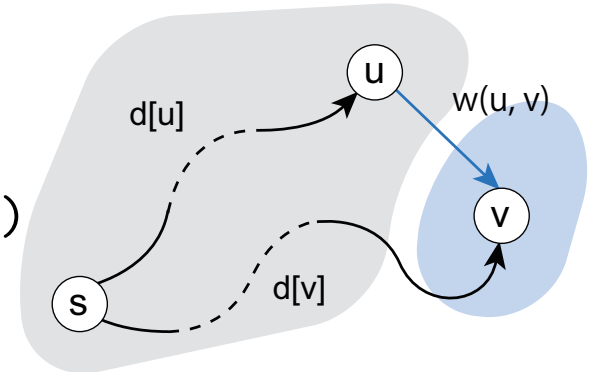
- For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v.
- We call $d[v]$ a shortest-path estimate.
- We initialize the shortest-path estimates and predecessors by the following procedure.

```
initializeSingleSource(G, s)
    for each v in G
        d[v] = INF
        pi[v] = NIL
    d[s] = 0
```

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $pi[v]$.
- A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v's predecessor field $pi[v]$.

```
relax(u, v, w)
    if d[v] > d[u] + w(u, v)
        d[v] = d[u] + w(u, v)
        pi[v] = u
```

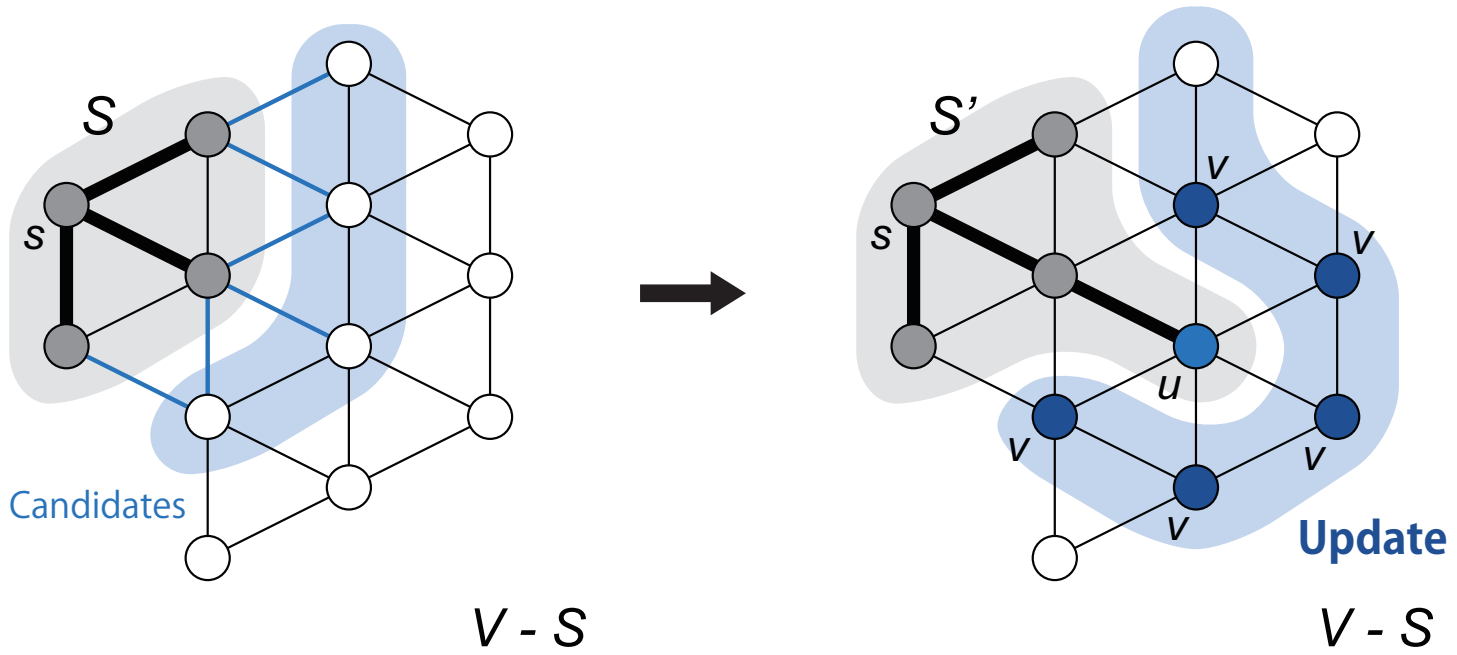


Dijkstra's Algorithm for SSSP (1)

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined.

Dijkstra's Algorithm for SSSP (2)

- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.



Dijkstra's Algorithm: $O(|V|^2)$ Implementation

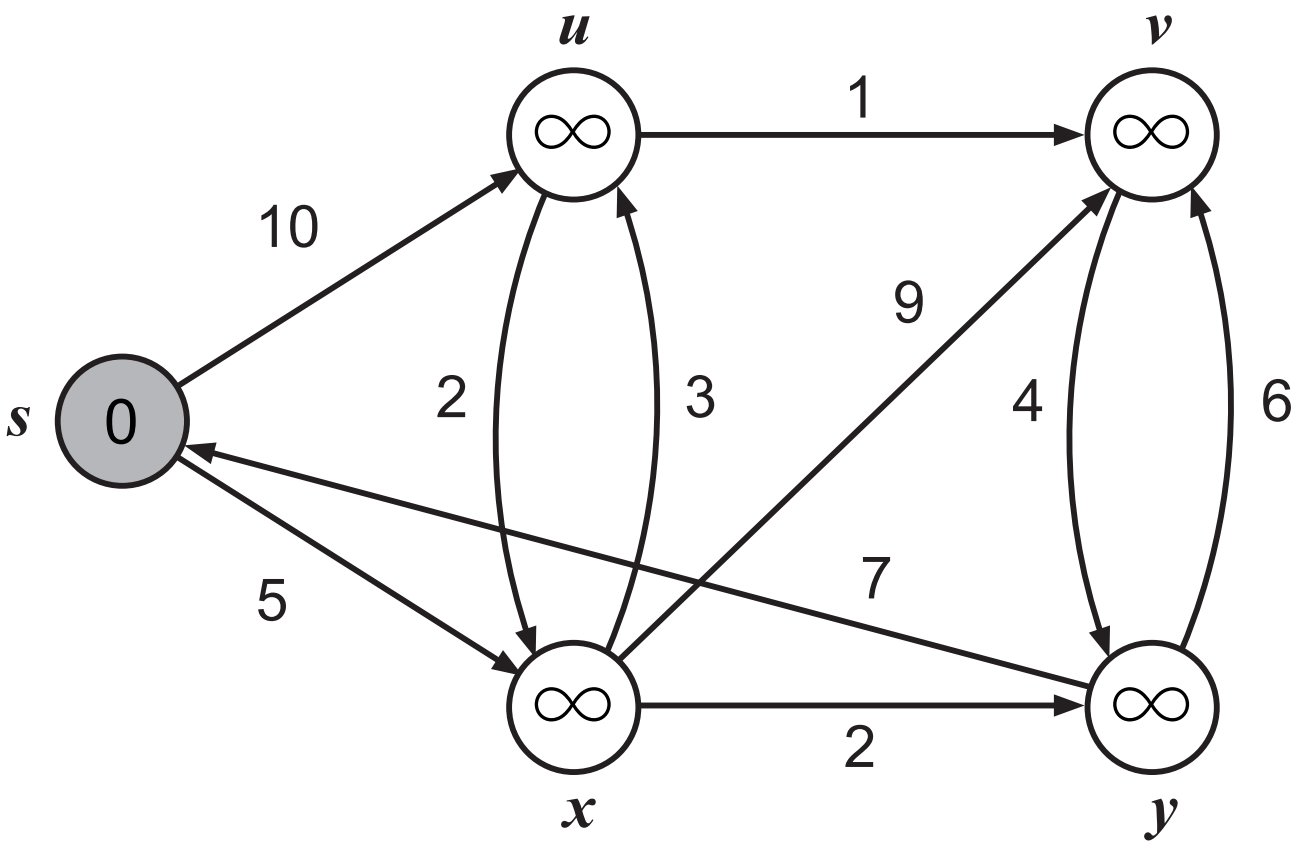
```
dijkstra(G, w, s)
    initializeSingleSource(G, s)
    while true
        mincost = INF
        for each i in G
            if color[i] != BLACK and d[i] < mincost
                mincost = d[i]
                u = i
        if mincost == INF
            break
        color[u] = BLACK
        for each v in Adj[u]
            if color[v] != BLACK and d[u] + w(u, v) < d[v]
                pi[v] = u
                d[v] = d[u] + w(u, v)
```

Dijkstra's Algorithm: $O((|V| + |E|) \log |V|)$ Implementation

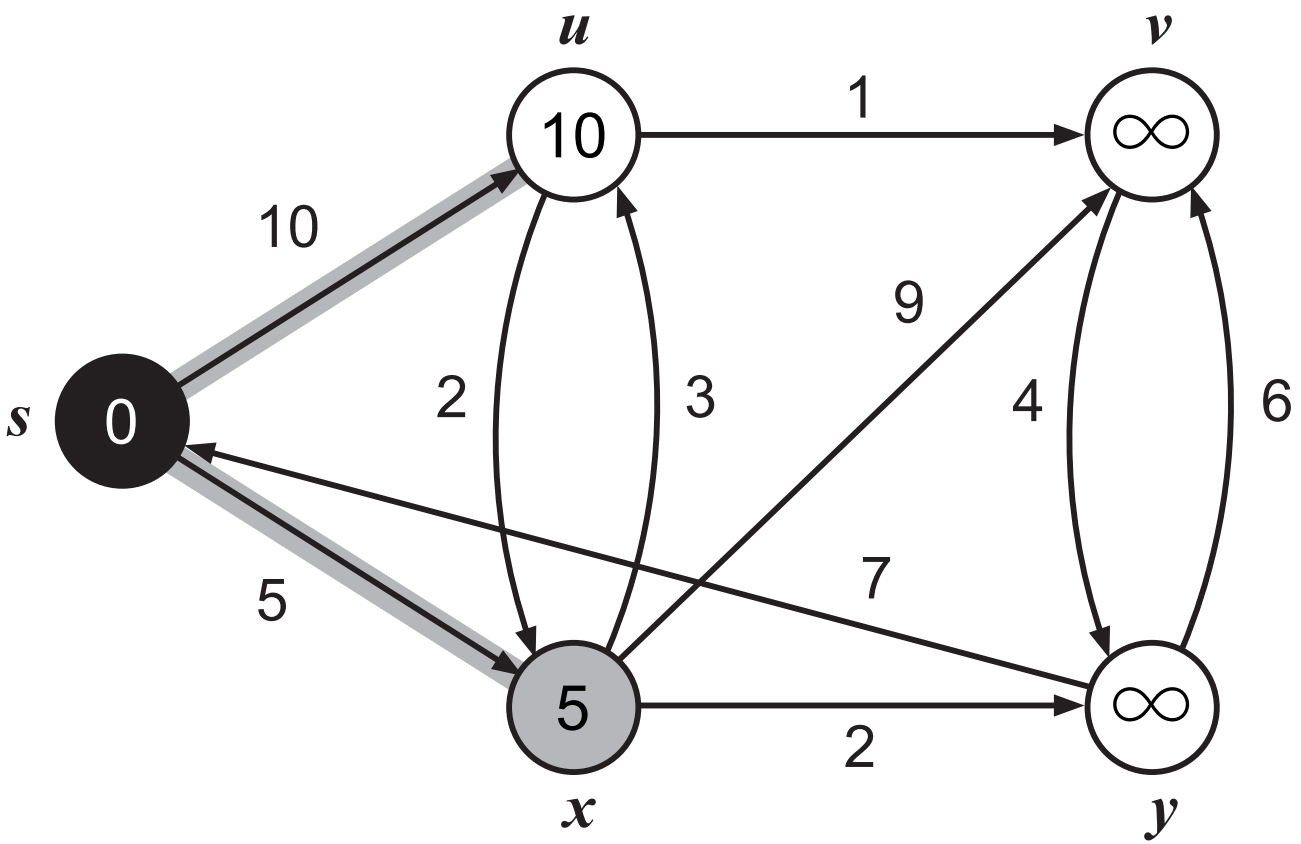
- For an efficient implementation, we use a min-heap H of vertices, keyed by their d values.

```
dijkstra(G, w, s)
    initializeSingleSource(G, s)
    H = buildMinHeap(V)
    while H is not empty
        u = H.extractMin()
        color[u] = BLACK
        for each v in Adj[u]
            if color[v] != BLACK
                if d[u] + w(u, v) < d[v]
                    d[v] = d[u] + w(u, v)
                    color[v] = GRAY
                    hepaDecreaseKey(H, v, d[u] + w(u, v))
```

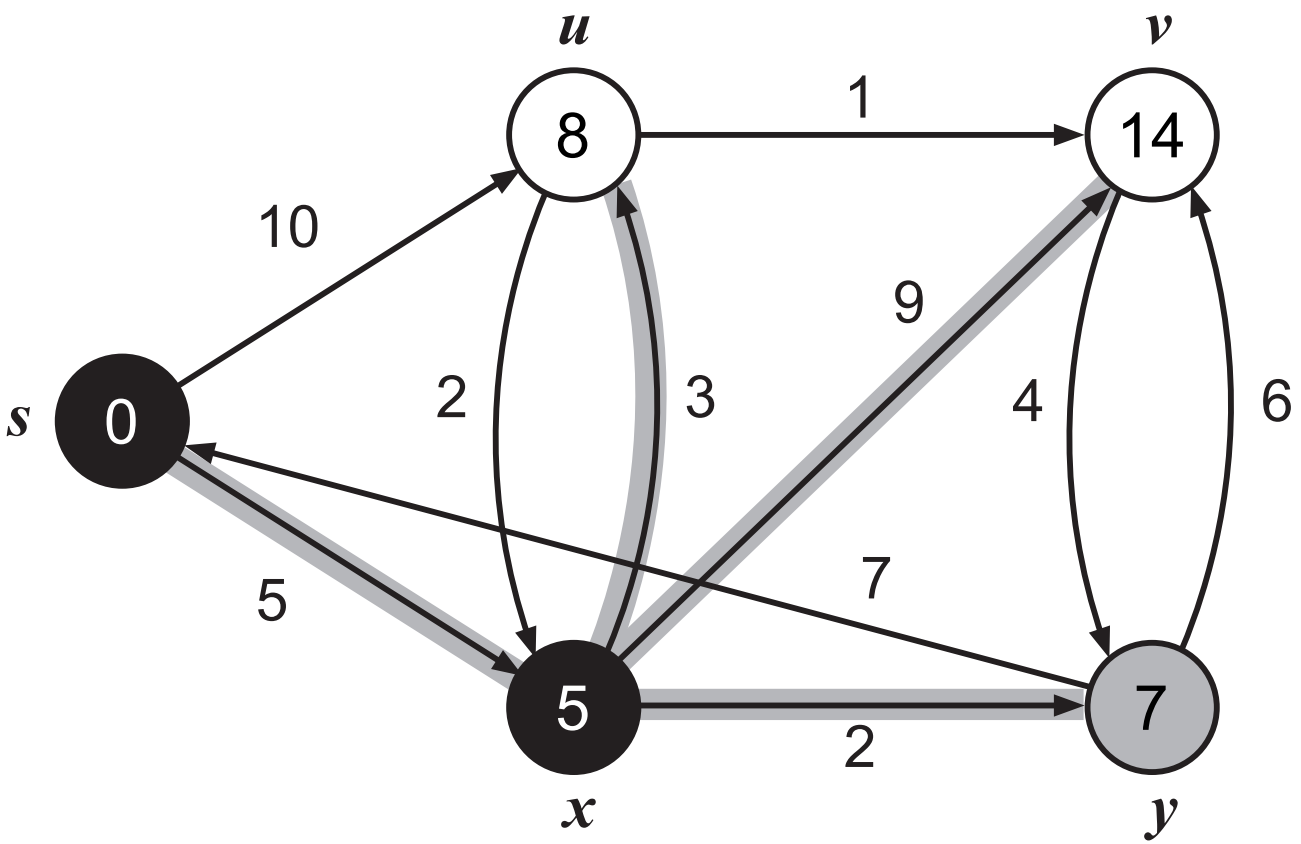
Dijkstra's Algorithm (1)



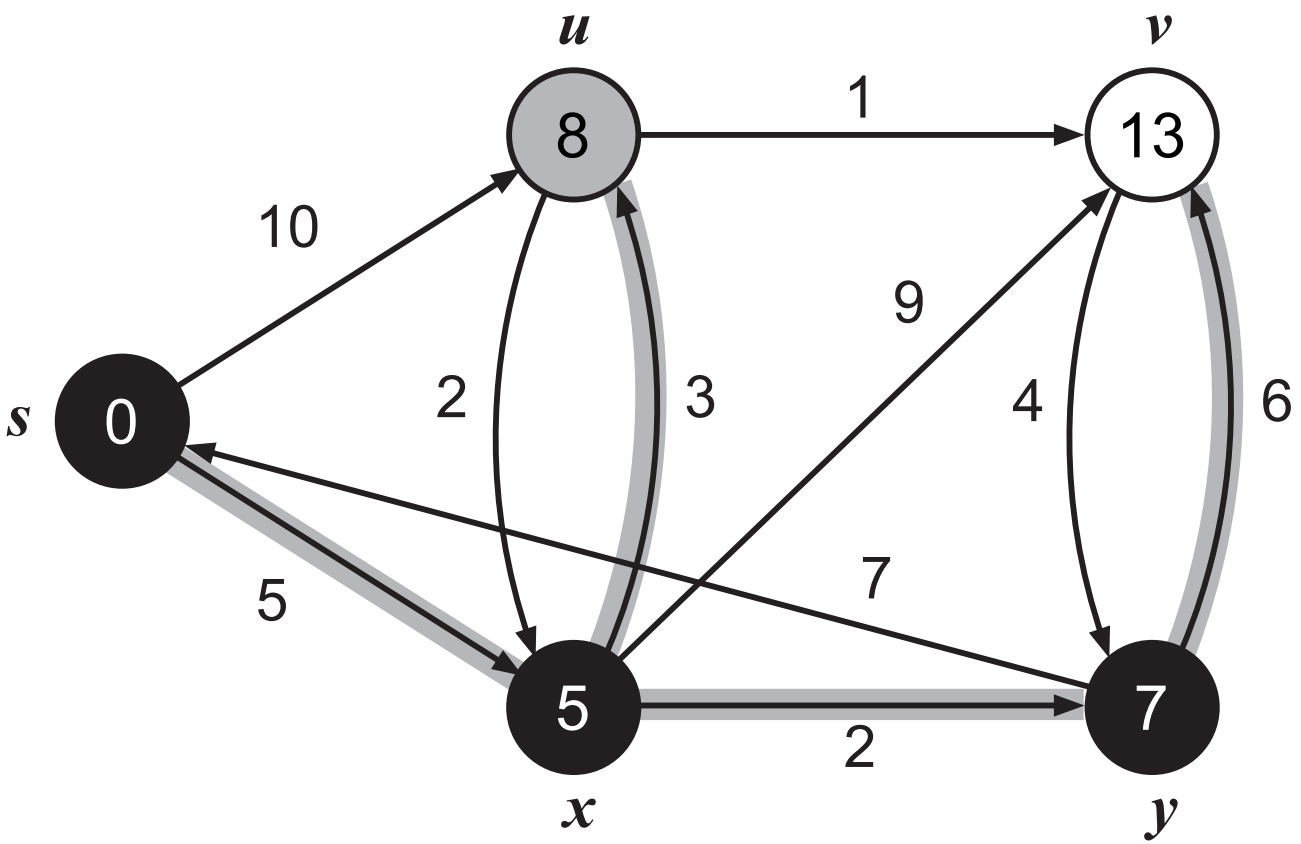
Dijkstra's Algorithm (2)



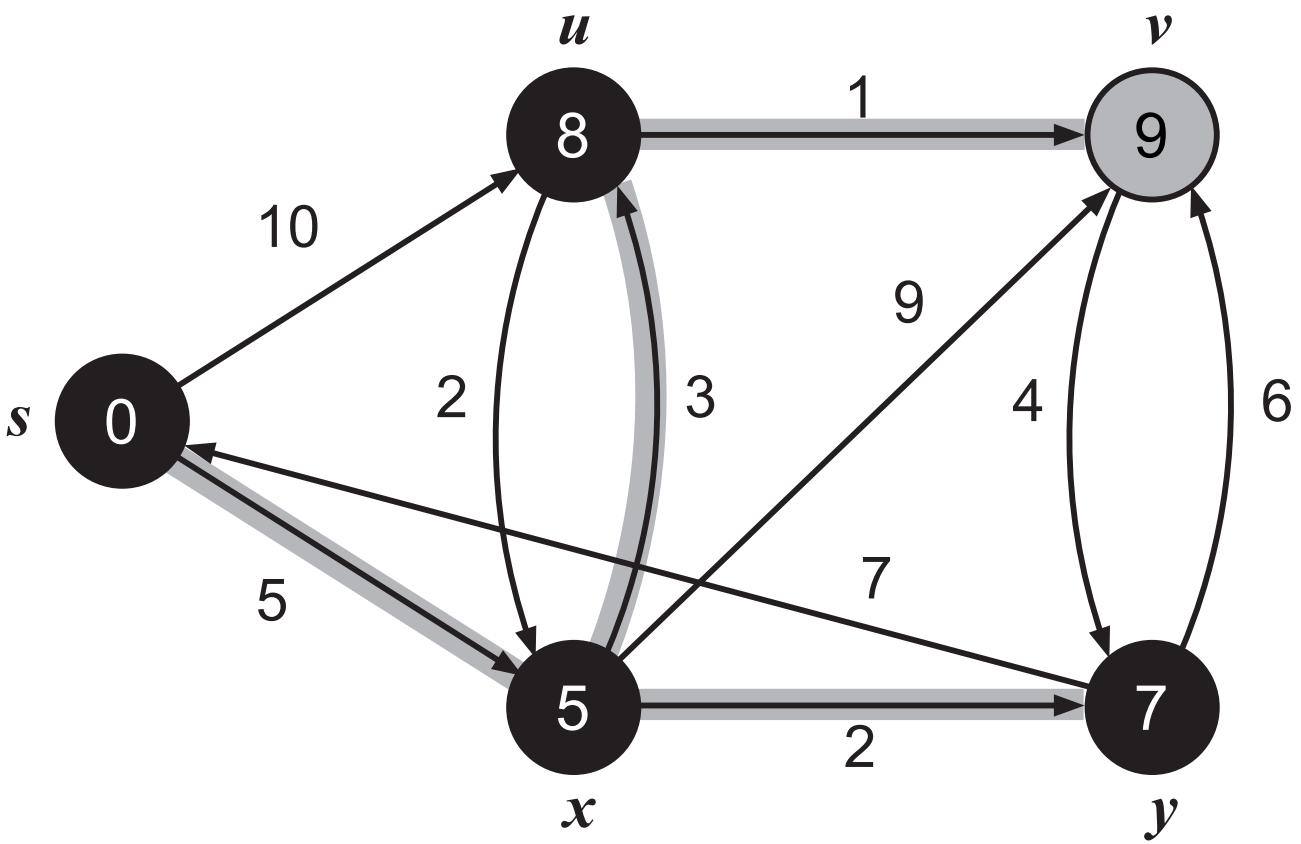
Dijkstra's Algorithm (3)



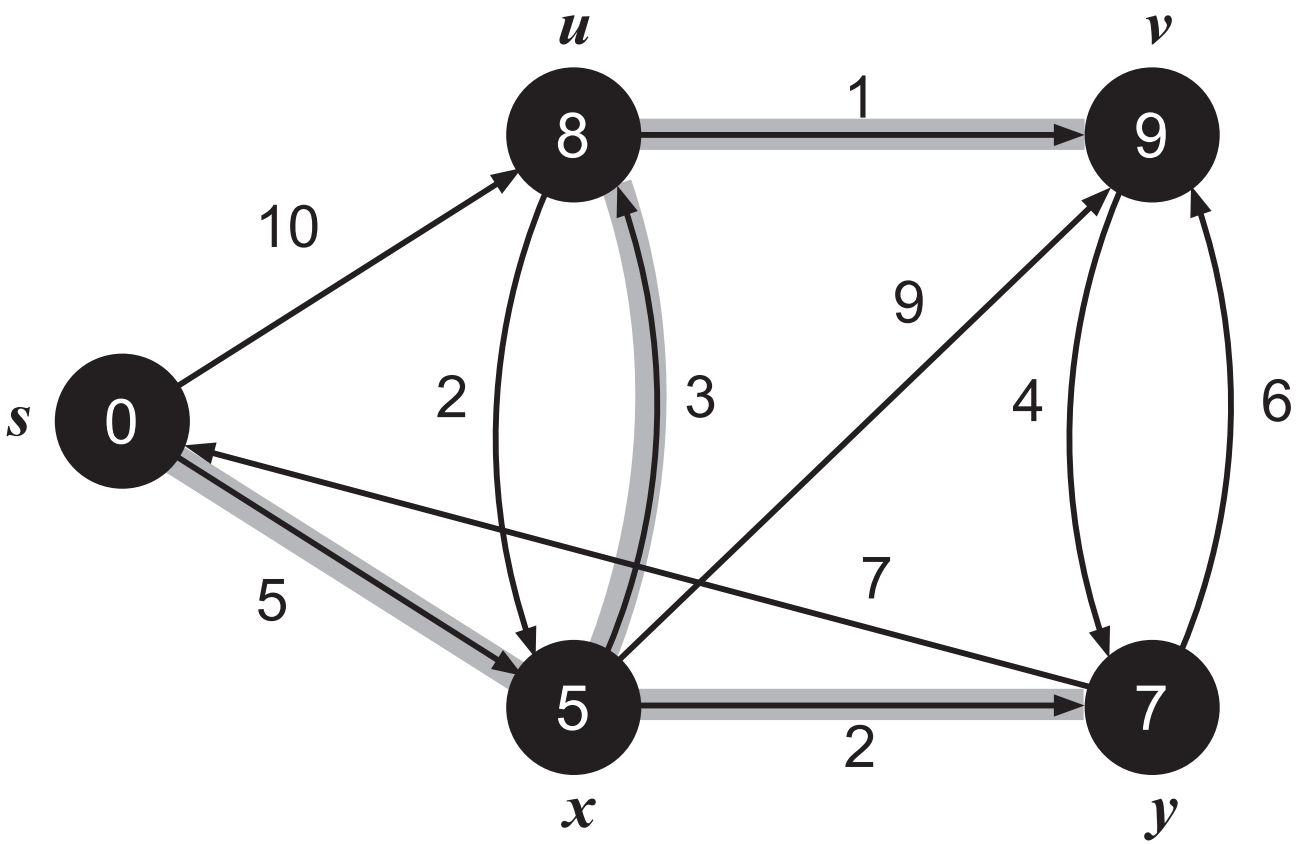
Dijkstra's Algorithm (4)



Dijkstra's Algorithm (5)

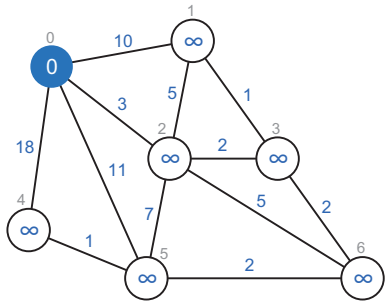


Dijkstra's Algorithm (6)

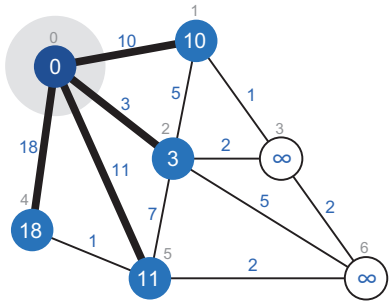


Dijkstra's Algorithm: Another Example

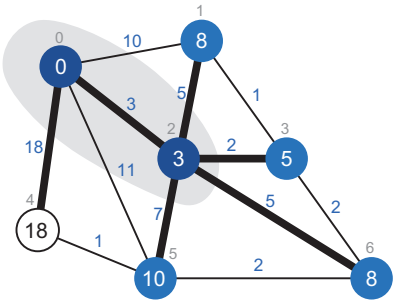
1



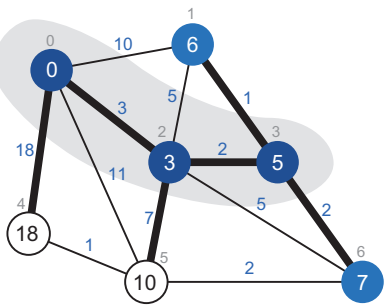
2



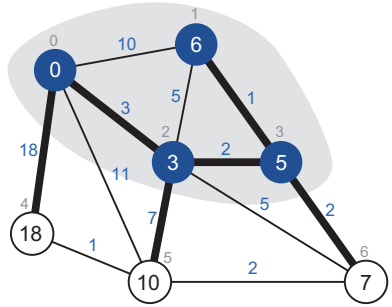
3



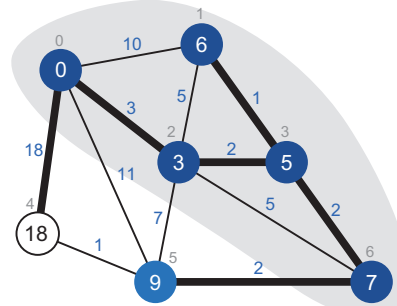
4



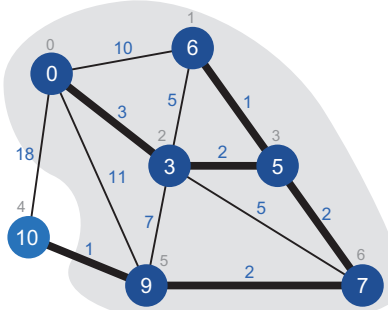
5



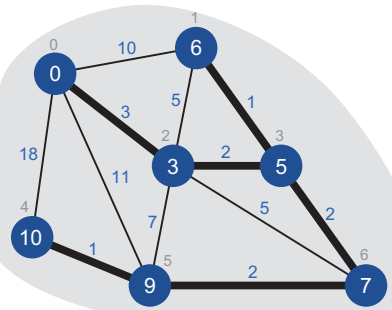
6



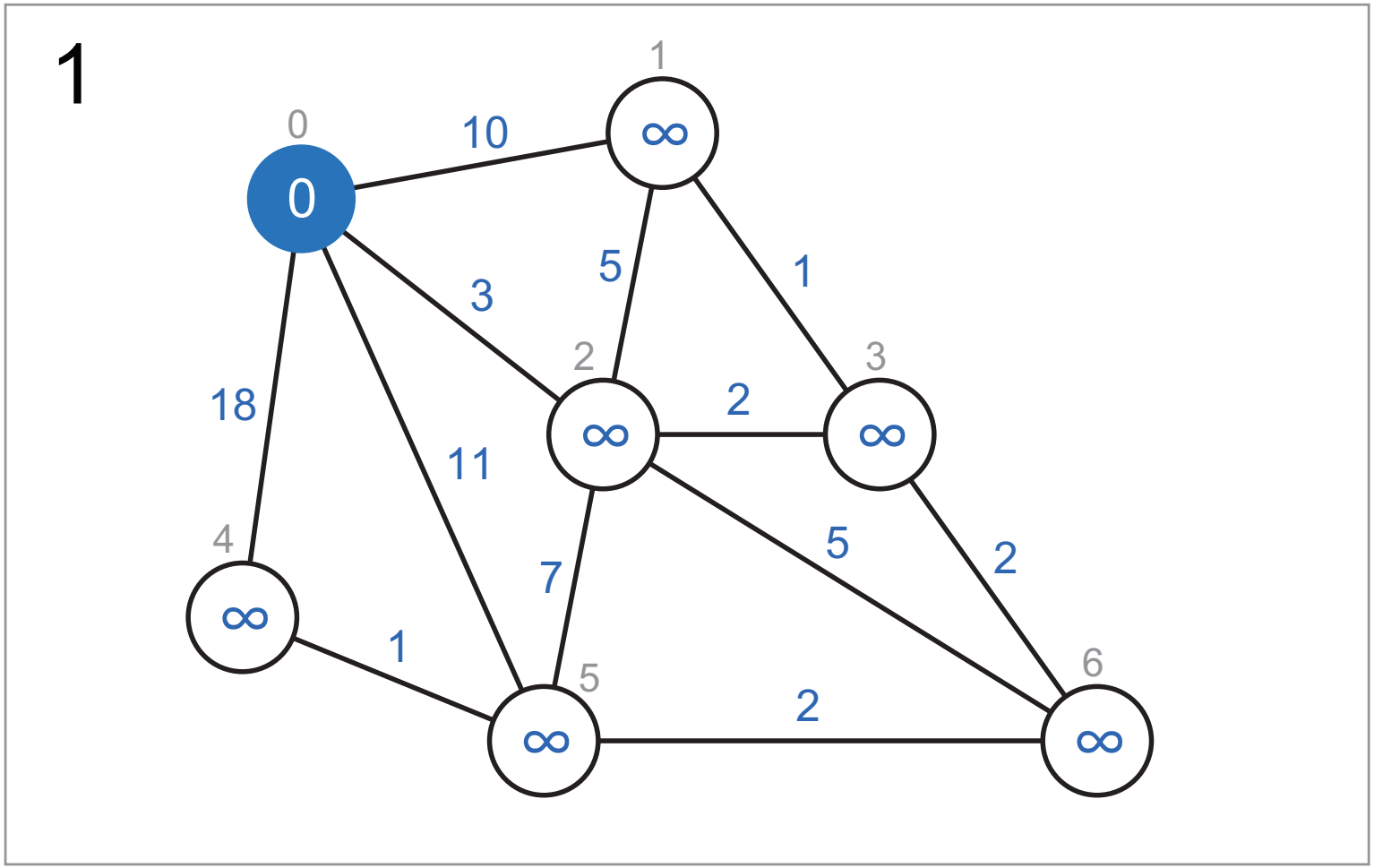
7



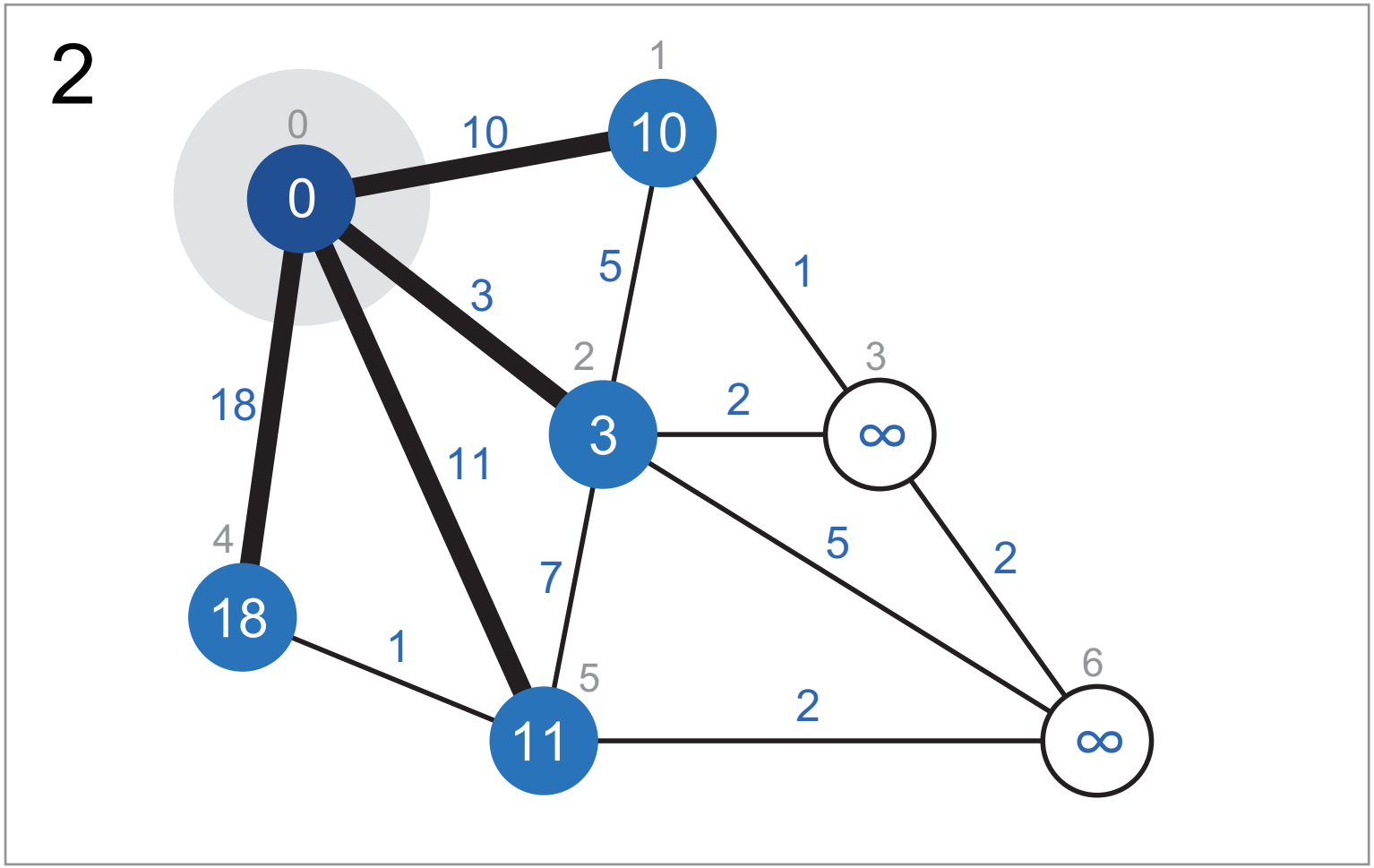
8



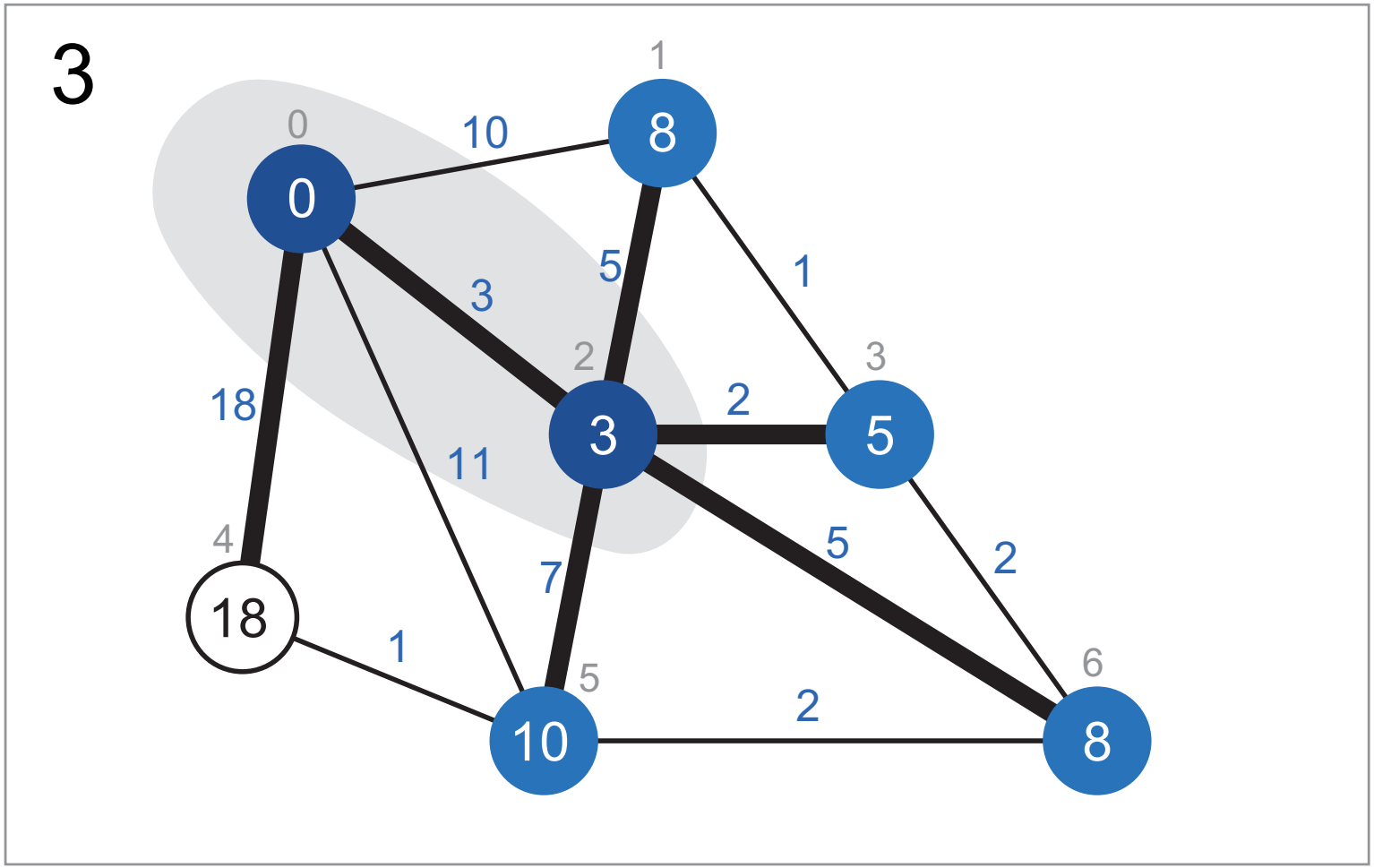
Dijkstra's Algorithm: Another Example (1)



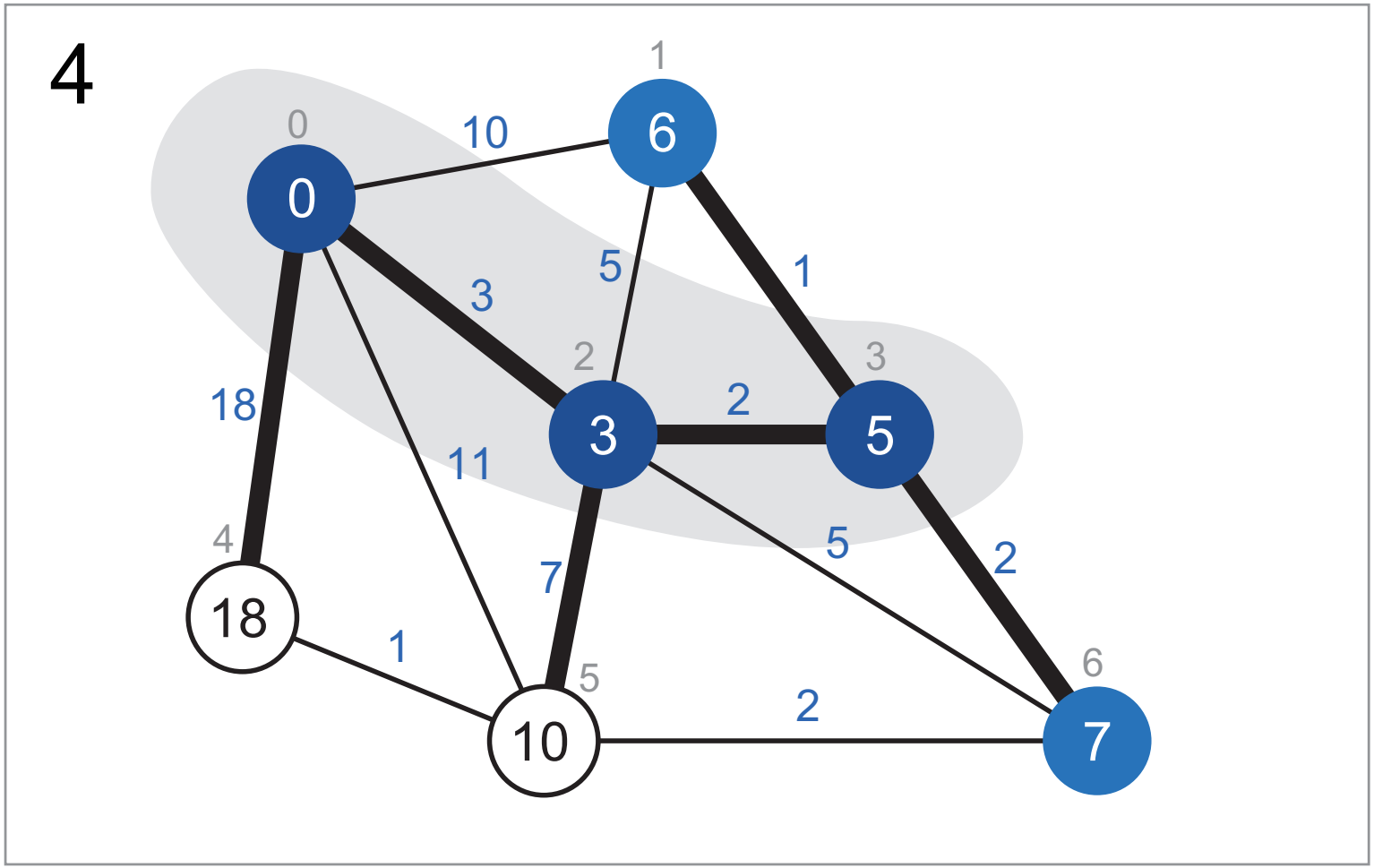
Dijkstra's Algorithm: Another Example (2)



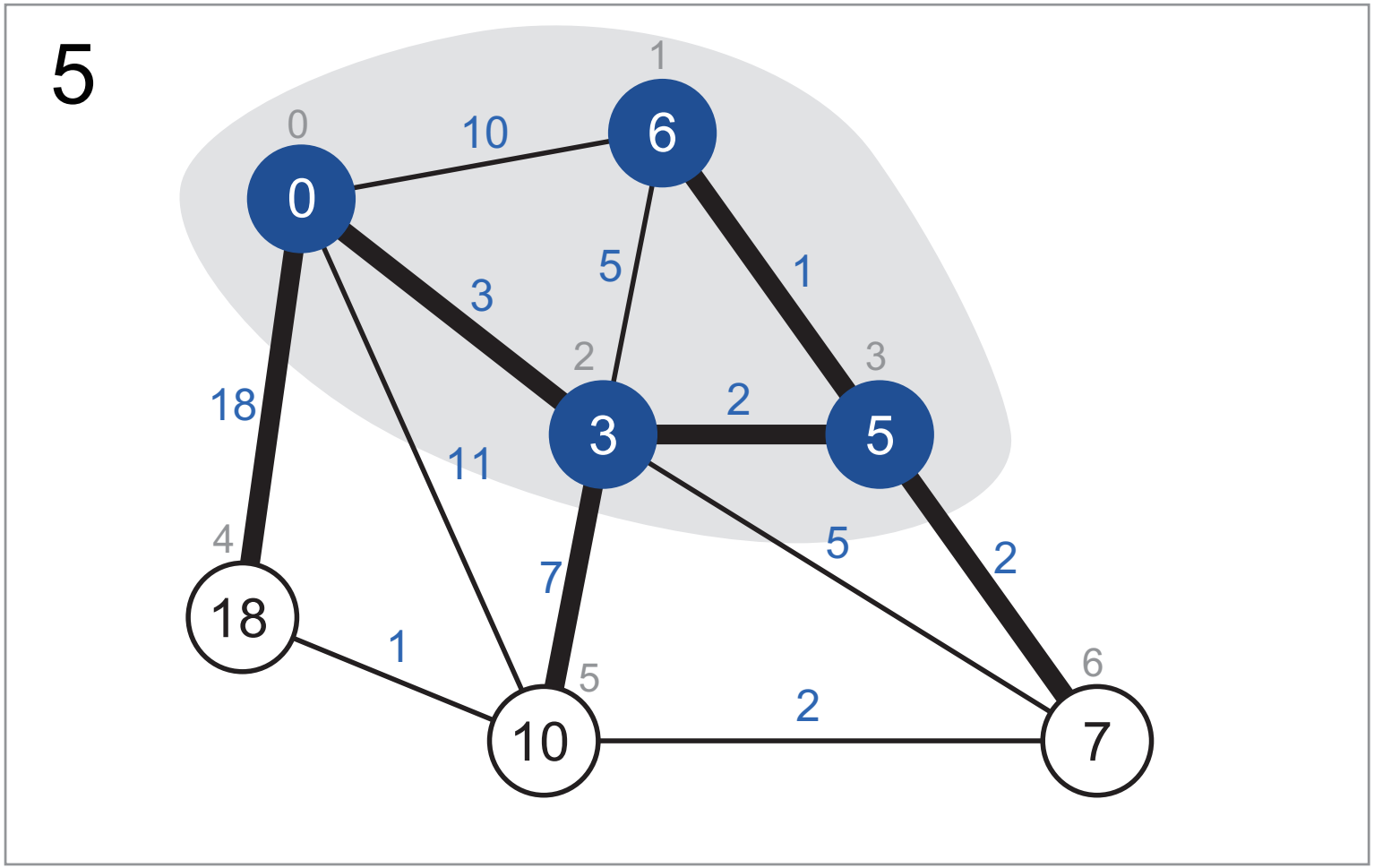
Dijkstra's Algorithm: Another Example (3)



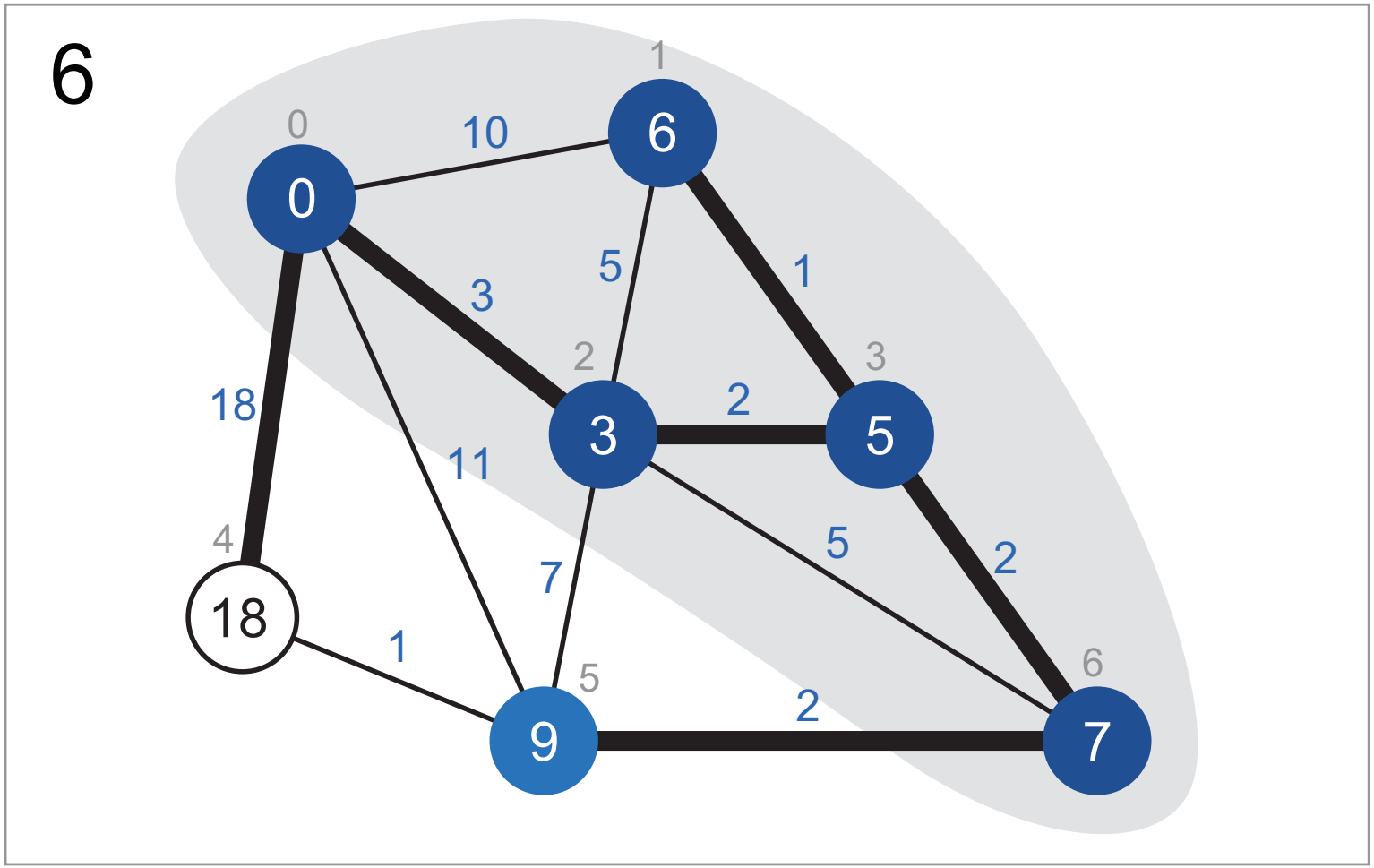
Dijkstra's Algorithm: Another Example (4)



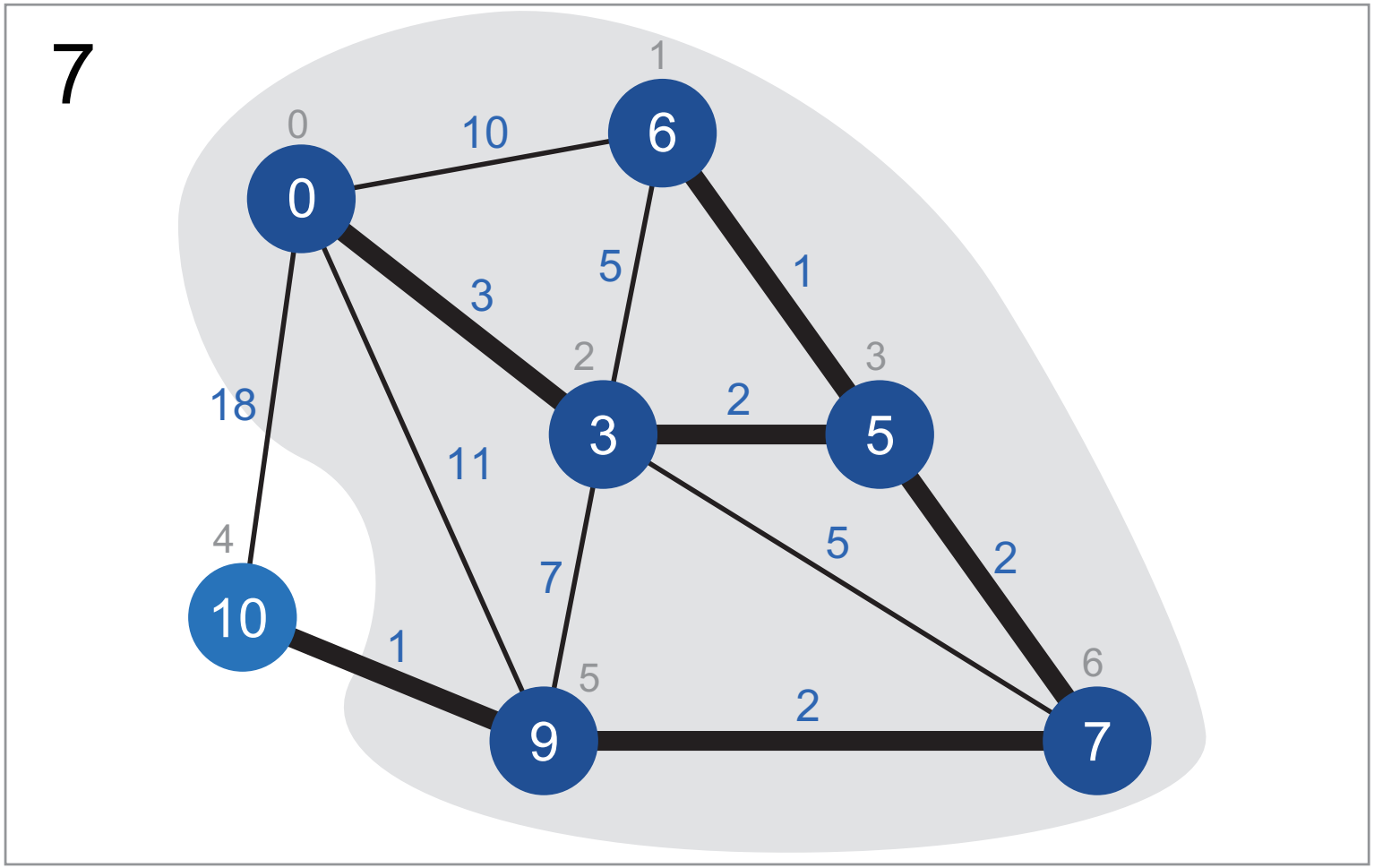
Dijkstra's Algorithm: Another Example (5)



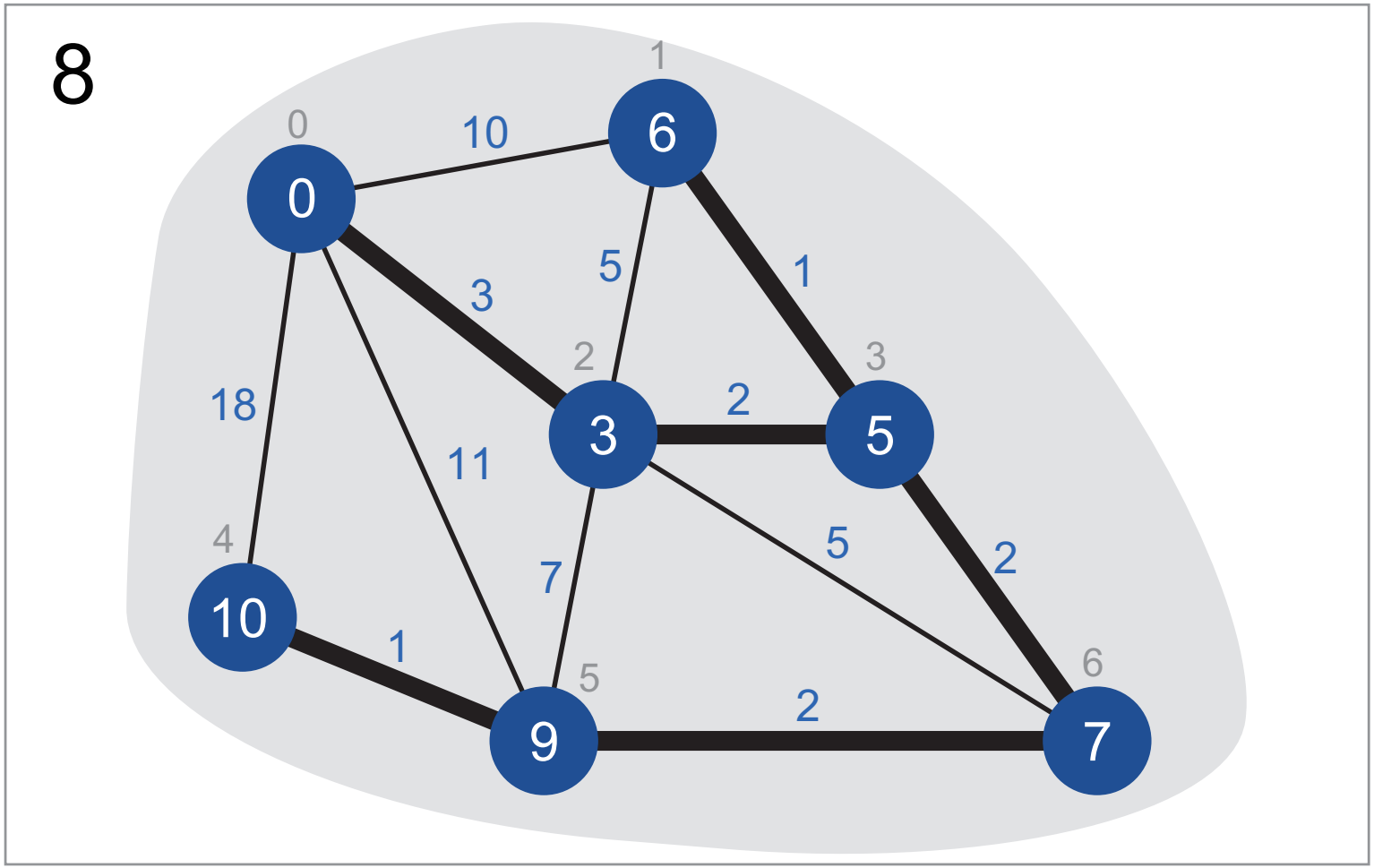
Dijkstra's Algorithm: Another Example (6)



Dijkstra's Algorithm: Another Example (7)



Dijkstra's Algorithm: Another Example (8)



Reference

1 Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.