Algorithms on Graphs: Part II

Minimal Spanning Trees

Spanning Tree. Let $G = \langle V, E, w \rangle$ be a (connected) edge-weighted undirected graph. A *spanning tree* of G is a subset $T \subseteq E$ of edges, such that $G_T = \langle V, T \rangle$ is *connected* and *acyclic*.

The weight of a spanning tree: $w(T) = \sum_{(u,v,)\in T} w(u, v)$.

Example. An example of a spanning tree is shown in Figure 1. The weight of the tree is 17.

Minimal Spanning Tree Problem. Given an undirected edge-weighted graph, find a spanning tree with minimum weight.

Greedy approach. (if we can make it work).

Idea: Manage a set A of edges such that:

> Prior to each iteration A is a *subset of some minimum spanning tree.*

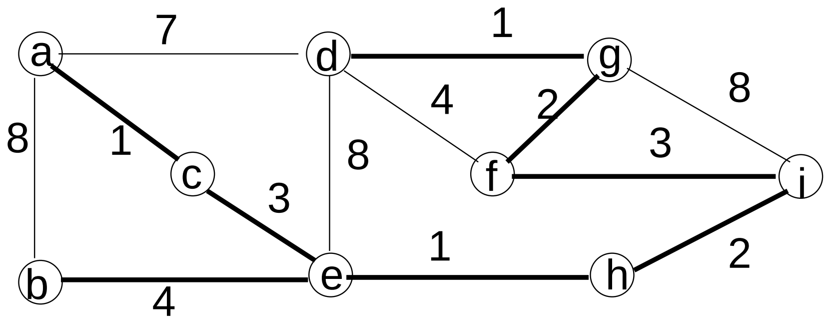

Figure 1: A spanning tree of a graph.

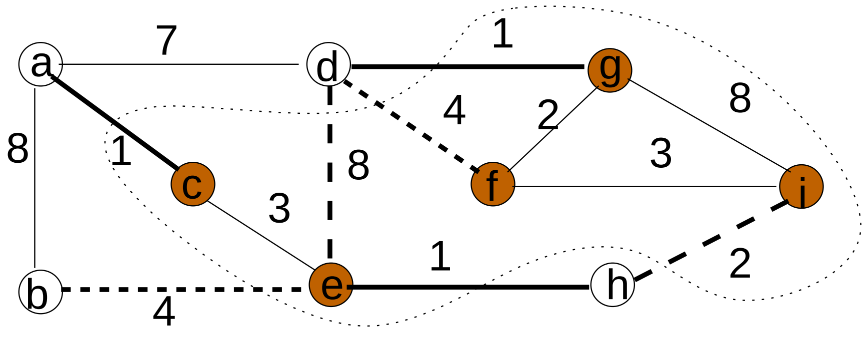

Figure 2: A $(\{c, e, f, g, j\}, \{a, b, d, h\})$-cut in the graph. Crossing edges are in bold. The light edges crossing the cut are in solid bold.

On each step, add to A an edge that is a part of the same minimum spanning tree. (We call such edges **safe**).

A generic greedy algorithm exploiting this idea looks as follows:

ALGORITHM GENERIC_MST(V,Adj,w)
begin
  $A \leftarrow \emptyset$;
  while A is not a spanning tree **do**
    find a safe edge$(u, v) \in E$ to include in A
    $A = A \cup \{(u, v\}$
  endwhile
  return A;
end

How to find a safe edge?

Cuts. A **cut** $(S, V - S)$ in an undirected graph $G = \langle V, E\rangle$ is a partition of V into two sets.

An edge (u, v) **crosses** the cut if $u \in S$ and $v \in V - S$ or vice versa.

A cut **respects** a set A of edges is no edge in A crosses the cut.

An edge (u, v) is a **light edge** crossing the cut is its weight is the minimum among all edges crossing the cut.

Example. Figure 2 shows an example of a cut in the graph from Figure 1. Bold edges cross the cut. Three crossing edges, (e, h), (g, d) and (c, a) have weight of 1. These edges (marked as solid bold lines) are the *light edges crossing the cut*. Other crossing edges are dashed on the figure.

If $A = \{(g, f), (f, j), (c, e)\}$, then the $\{(c, e, f, g, j), (a, b, d, h)\}$-cut depicted in Figure 2 *respects* A (as none of the edges from A cross the cut).

Theorem 3. Let $G = \langle E, V, w\rangle$ be a connected undirected edge-weighted graph. Let $A \subset E$ be in some *minimal spanning tree.* and let $(S, V - S)$ be some cut of G.

If $(S, V - S)$ respects A and (u, v) is a light edge crossing A, then (u, v) is safe for A.

Proof. Illustrated in Figure 3. Consider a minimum spanning tree T

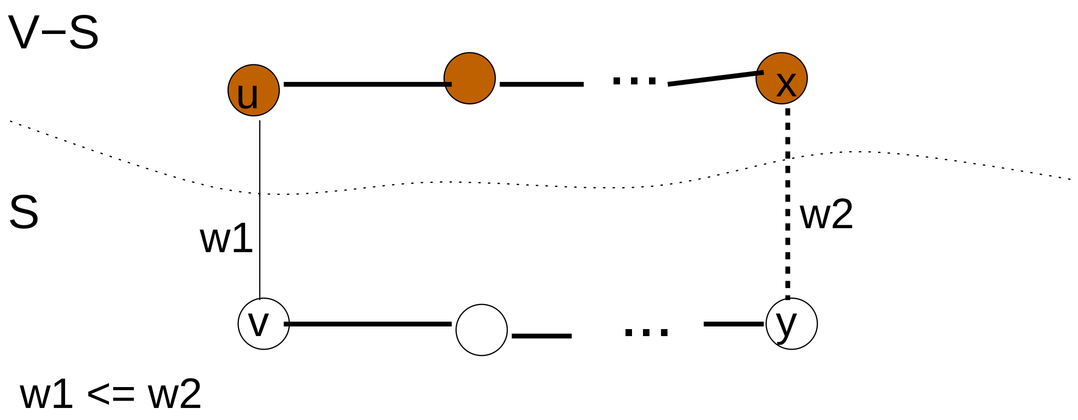

Figure 3: Proving Theorem 3.

that contains A, but does not contain the edge (u, v). If we add (u, v) to T, the resulting graph will have a cycle, as there will now be two distinct paths between u and v (as depicted on Figure 3. Because u and v belong to different partitions, there is at least one edge on the *other* path from u to v in T that crosses the $(S, V - S)$-cut. It is marked on the Figure as the (x, y) edge.

We know that $w(u, v) = w1 \leq w2 = w(x, y)$ (as (x, y) is the light edge crossing the $(S, V - S)$-cut).

Consider now the graph $T' = T - \{(x, y)\} \cup \{(u, v)\}$. We note that:

- T' is a spanning tree. ((x, y) was on the unique path from u to v, and $(S, V - S)$ respects A, so $(x, y) \notin A$.) T' is acyclic.
- $W(T') = W(T) - w(x, y) + w(u, v) \leq W(T)$.

From this, we conclude that T' is the minimum spanning tree.

Kruskal's Algorithm

Greedy Strategy. On each step *pick the edge with the smallest weight*, that keeps the set of selected edges a tree.

Implementation. Our algorithm needs to ensure two things:

- Fast selection of an edge with the smallest weight on each step of the process.
  - This achieved in a *brute-force* way: we **sort** all edges in the input graph by their weight.
- Fast determination, if an edge can be a part of the spanning tree.
  - (u, v) is not a part of the spanning tree if u is already connected to v.
  - Check this using the disjoint set ADT. Algorithm will keep track of all disjoint connected components. On each step, checks if u and v are in the same set. If yes, skip, otherwise, add and edge, and unite the two sets.

The algorithm is formalized below.

```
ALGORITHM MST_KRUSKAL(V,Adj,w)
begin
  A ← ∅;
  foreach v ∈ V do MakeSet(v);
  E ← sort Adj in ascending order by w(e), e ∈ Adj;
  foreach (u, v) ∈ E do     //retrieved in ascending order
    if FindSet(u) ≠FindSet(v) then
      A = A ∪ {(u, v}
      Union(u,v)
    endif
  endfor
  return A;
end
```

Correctness. Follows from **Theorem 3.**

Running Time.

$$T(\mathsf{MST\_KRUSKAL}) = |V|T(\mathsf{MakeSet})+|E|\log_2(|E|)+|E|\,(2T(\mathsf{FindSet}) + T(\mathsf{Union}))+O(1).$$

This, generally speaking depends on the implementation of the disjoint set ADT. Good implementations of this ADT take on the order of $O(\log_2(|E|)) = O(\log_2(|V|)$ running time ($|E| = O(|V|^2)$, hence, $\log_2(E) = O(\log_2(V))$. This, in turn leads to

$$T((\mathsf{MST\_KRUSKAL}) = O(|E|\log_2(|V|)).$$

Note. For dense graphs, this is generally *quadratic* in the number of nodes.

Prim's Algorithm

Kruskal's Algorithm builds a forest of trees, and on each step combines a pair of trees using the *least expensive* edge.

In constrast, Prim's algorithm maintains a single tree and on each step attaches one more node to it.

Greedy Idea. Construct the minimum spanning tree incrementally starting from *some* arbitrarily chosen node in the graph. On each step, nodes in the tree and nodes outside form the $(S, V-S)$ cut. Adding a light edge crossing the cut will connect one more node to the tree.

Implementation. Our algorithm needs to efficiently perform the following operations.

- Given a set S of nodes, determine all edges that cross the $(S, V-S)$ cut.
- Given a set of edges, find the edge with the smallest weight.

Both objectives are achieved using a priority queue structure to store all nodes in V that have not (yet) made it into the tree S. At the beginning all but one node will be in the priority queue. At the end, the queue will become empty.

The **key** on which the nodes are stored in the priority queue is the

> smallest weight of an edge connecting the given node to *any already selected node.*

The minimum spanning tree is kept implicitly in the $\pi[v]$ array, which, for each node v stores the pointer to its *"parent"* node in the spanning tree.

Input. Prim's algorithm takes as input the graph $G = \langle V, E, w\rangle$ and the root node $r \in V$, which will be used to "grow" the MST.

The pseudocode for the algorithm is shown below.

```
ALGORITHM MST_PRIM(V,Adj,w,r)
begin
  foreach v ∈ V do     // initialization step
    key[v] ← ∞;
    π[v] ← NIL;
  endfor
  key[r] ← 0;
  Queue ← V;    // insert all nodes in the priority queue
  while Queue ≠ ∅ do    // main loop:  retrieve nodes from priority queue
    u ← ExtractMin(Queue);
    foreach v ∈ Adj[u] do     // adjust distances
     if  v ∈ Queue and w(u,v) < key[v] then
       key[v] ← w(u,v);
       π[v] ← w(u);
     endif
    endfor
  endwhile
end
```

Correctness. Follows directly from **Theorem 3.**

Computational Complexity.

1. Initialization loop takes $O(|V|)$ time.
2. *Queue* initialization is actually a loop:

   foreach $v \in V$ **do** Insert$(Queue, v)$;

   T(Insert) depends on priority queue implementation. For priority queues implemented as *binary min-heaps*, Insert takes $O(\log_2(|V|)$ time, and hence, the entire initialization will take $(|V|\log_2(|V|)$ time.
3. The body of the **while** loop is executed $|V|$ times. ExtractMin for *binary min-heaps* takes $\log_2(|V|)$ time.
4. The nested **for** loop will execute $O(|E|)$ times over all iterations of the **while** loop. The $v \in Queue$ test can be done in constant time (by using flags for each node v).

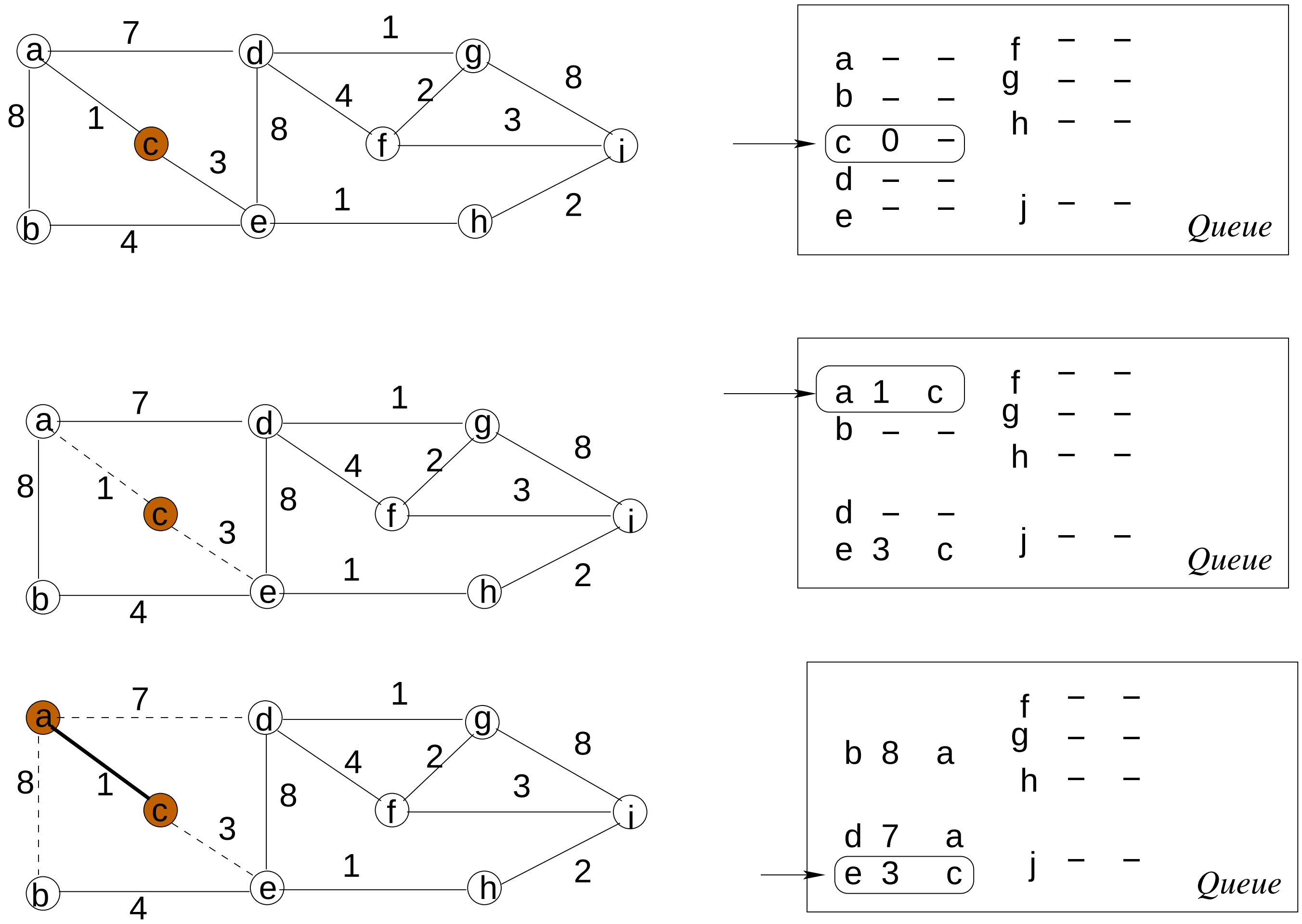

Figure 4: Running Prim's algorithm: Steps 1–3.

The key reassignment, however, triggers the DecreaseKey procedure on the priority queue. For *binary min-heaps* DecreaseKey runs in $O(\log_2(|V|))$.

This means that the **while** loop has the running time

$$O(|V|\log_2(|V|) + |E|\log_2(|V|)).$$

From the observations above:

$$T(\textsf{MST\_PRIM}) = O(|V|) + O(|V|\log_2(|V|) + O(|V|\log_2(|V|) + |E|\log_2(|V|)) =$$

$$O(|E|\log_2(|V|).$$

Example. Figures 4, 5 and 6 show the run of Prim's algorithm on the graph from Figure 1 starting with node c as root. On each step, the graph has the following legend:

- Nodes removed from the queue are shaded.
- Edges "included" (via $\pi[v]$ relationship that will not change) into the minimum spanning tree are in **bold**.

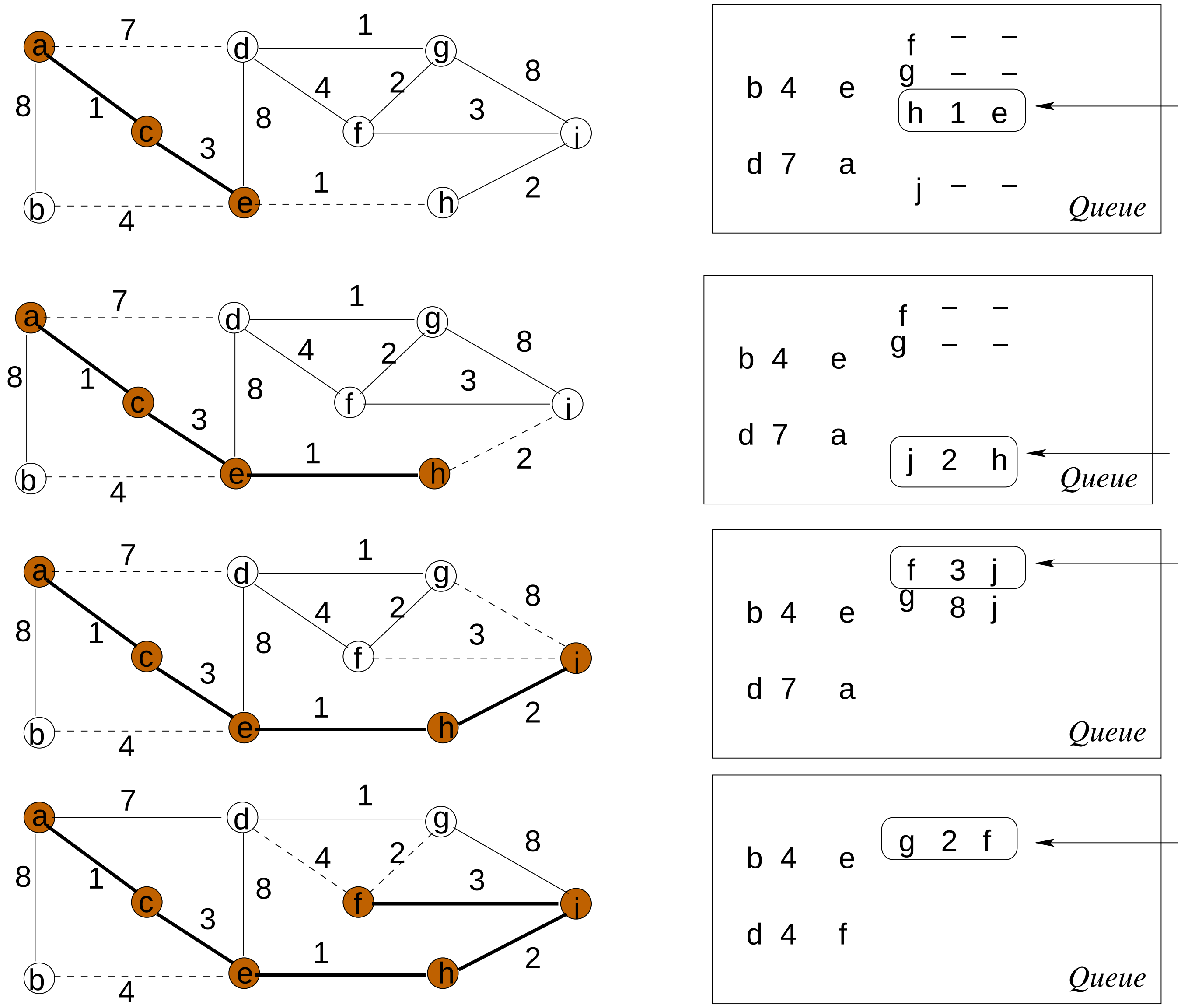

Figure 5: Running Prim's algorithm: Steps 4–7.

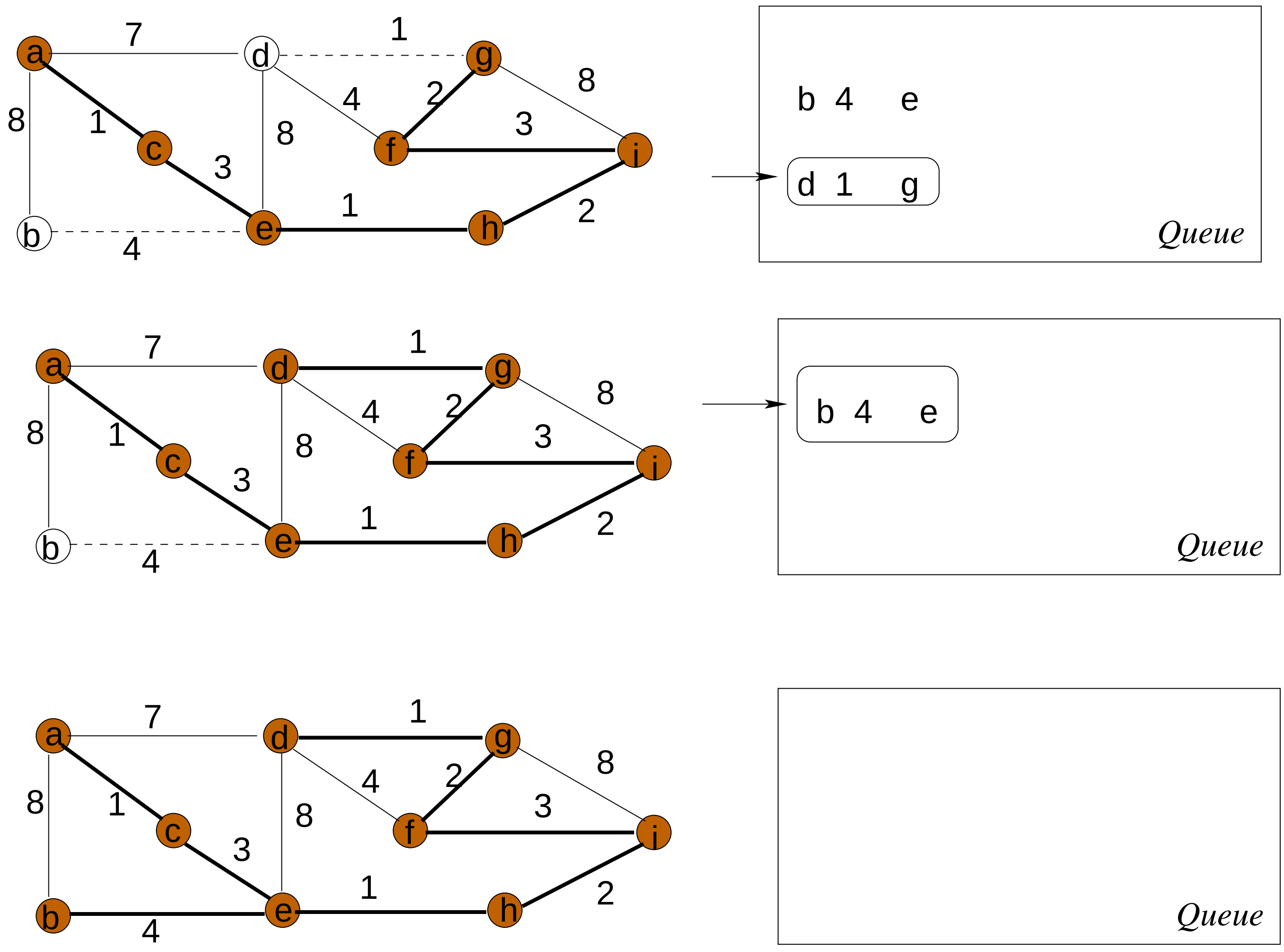

Figure 6: Running Prim's algorithm: Steps 8–10.

- Edges used to establish current distances in the priority queue are *dashed*.

The priority queue structure (rendered as an array) is shown for each step as well. In each row, the name of the node is first, followed by the current smallest edge weight to the spanning tree, followed by the current "parent" of the node. Node with the smallest edge weight (to be removed on the next step) is highlighted.