CSE 421

Greedy Alg: Union Find/Dijkstra's Alg

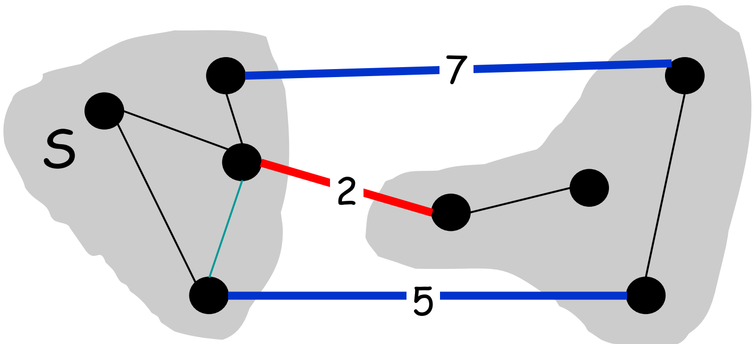
Shayan Oveis Gharan

Properties of the OPT

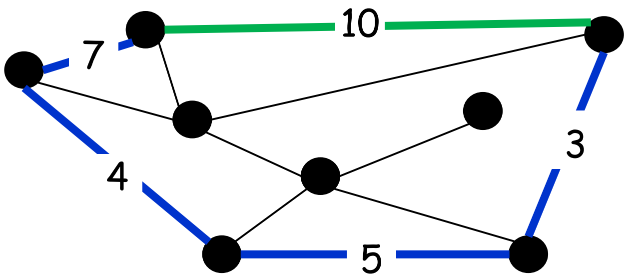
Simplifying assumption: All edge costs c_e are distinct.

Cut property: Let S be any subset of nodes (called a cut), and let e be the min cost edge with exactly one endpoint in S. Then every MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then no MST contains f.



red edge is in the MST



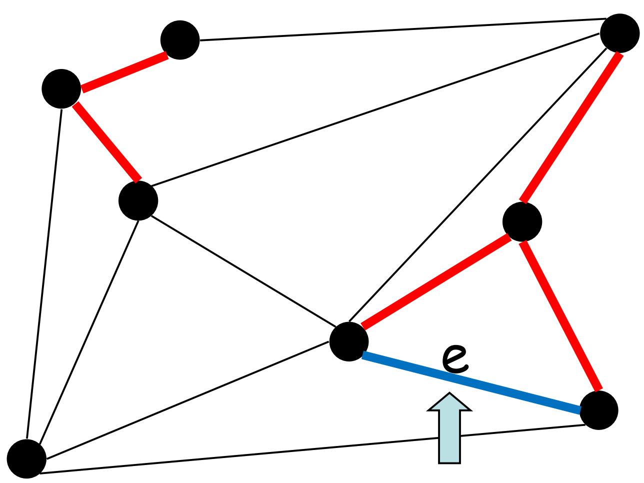
Green edge is not in the MST

Kruskal's Algorithm: Pf of Correctness

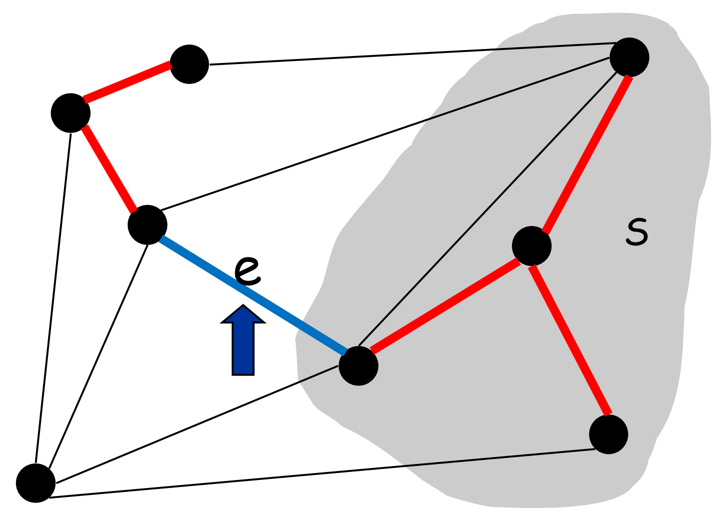
Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

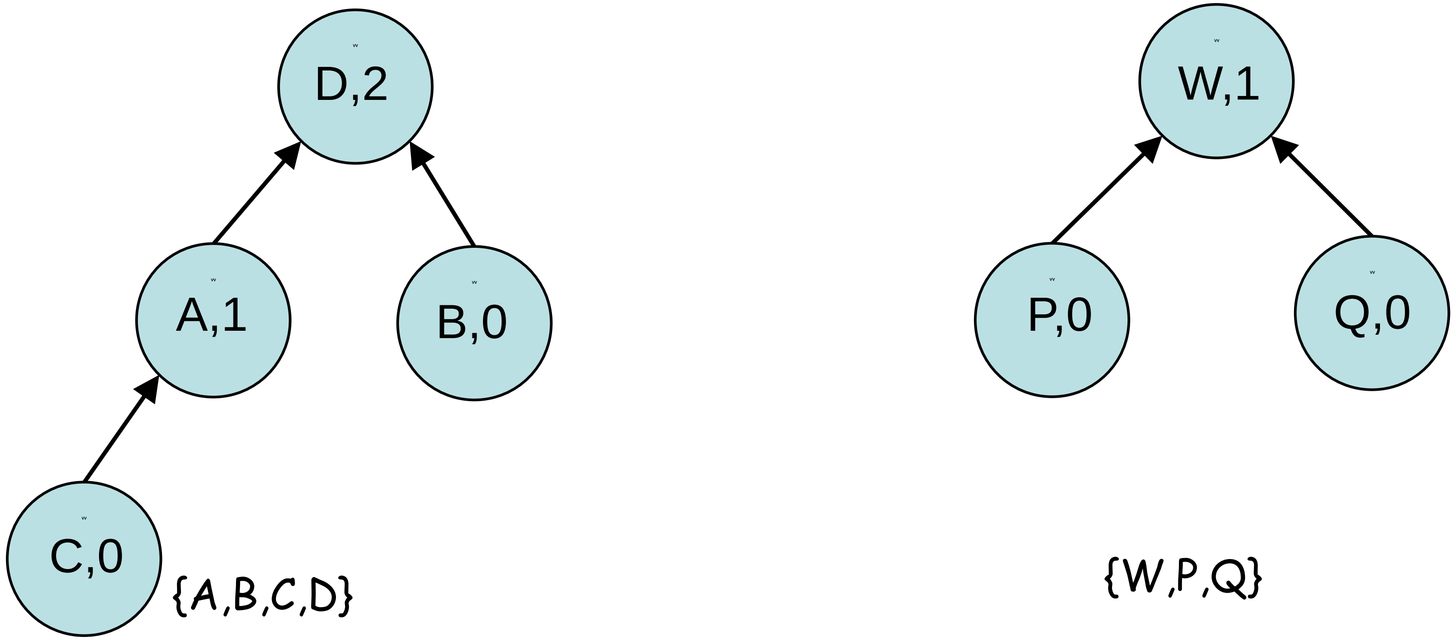
- Build set T of edges in the MST.
- Maintain a set for each connected component.
- O(m log n) for sorting and O(m log n) for union-find

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← ∅

   foreach (u ∈ V) make a set containing singleton {u}

   for i = 1 to m
      Let (u,v) = eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }
   return T
}
```

Union Find Data Structure

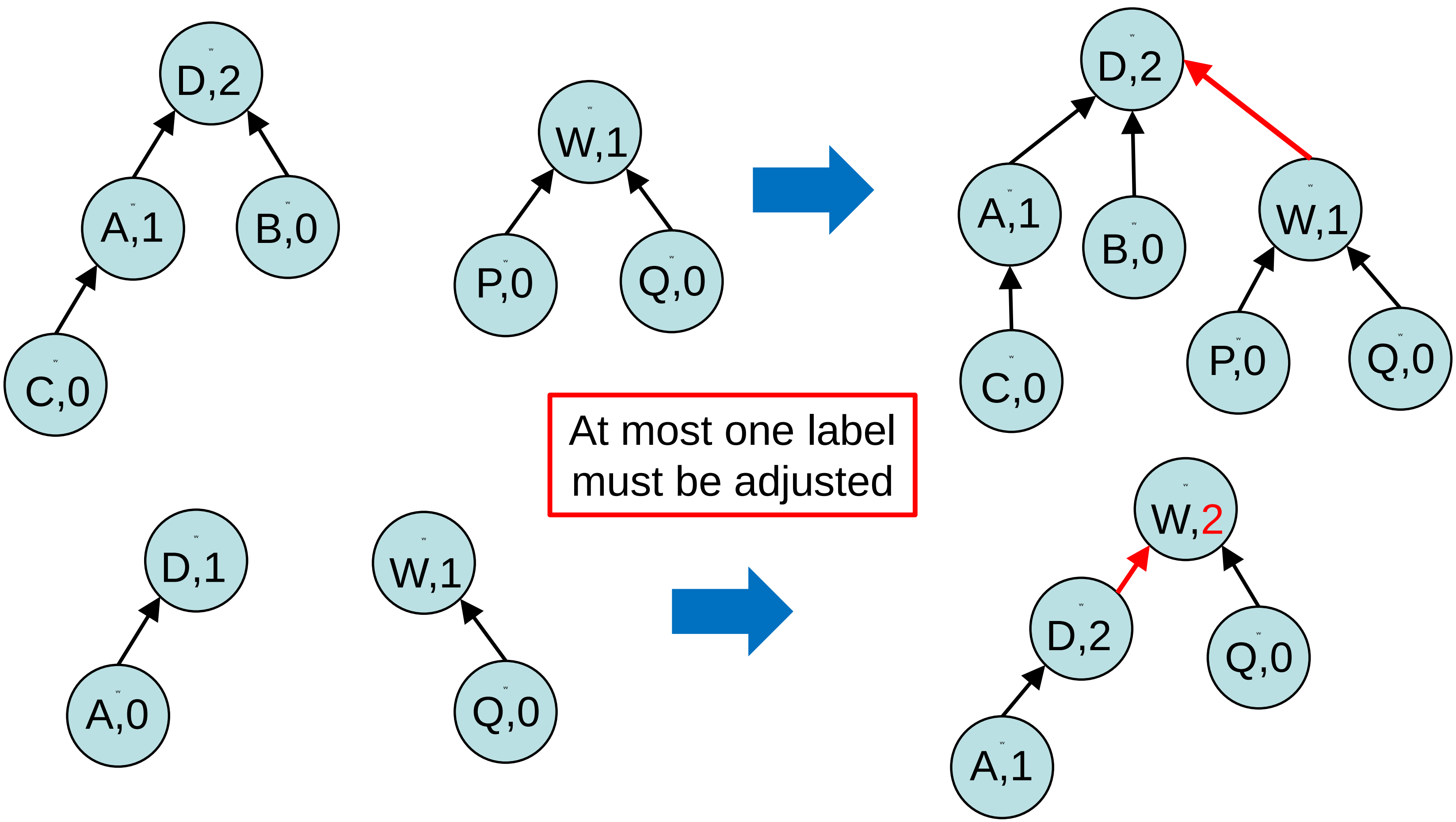
Each set is represented as a tree of pointers, where every vertex is labeled with longest path ending at the vertex

To check whether A,Q are in same connected component, follow pointers and check if root is the same.



Union Find Data Structure

Merge: To merge two connected components, make the root with the smaller label point to the root with the bigger label (adjusting labels if necessary). Runs in O(1) time

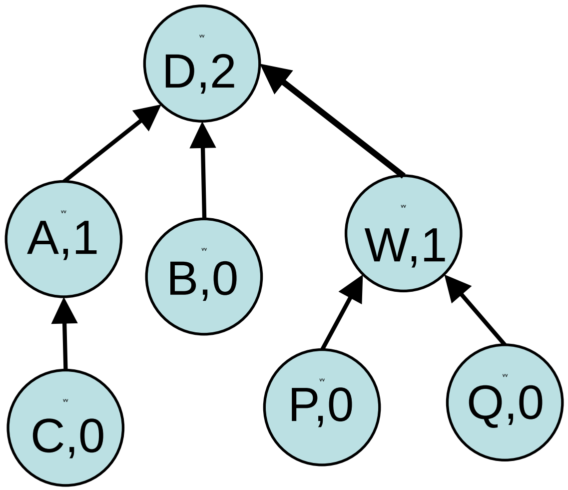


Depth vs Size

Claim: If the label of a root is k, there are at least 2^k elements in the set.

Therefore the depth of any tree in algorithm is at most log n

So, we can check if u, v are in the same component in time $O(\log n)$



Depth vs Size: Correctness

Claim: If the label of a root is k, there are at least 2^k elements in the set.

Pf: By induction on k.

Base Case (k = 0): this is true. The set has size 1.

IH: Suppose the claim is true until some time t

IS: If we merge roots with labels $k_1 > k_2$, the number of vertices only increases while the label stays the same.

If $k_1 = k_2$, the merged tree has label $k_1 + 1$,

and by induction, it has at least

$$2^{k_1} + 2^{k_2} = 2^{k_1+1}$$

elements.

Kruskal's Algorithm with Union Find

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain a set for each connected component.
- O(m log n) for sorting and O(m log n) for union-find

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← ∅

   foreach (u ∈ V) make a set containing singleton {u}

   for i = 1 to m
      Let (u,v) = eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }
   return T
}
```

Find roots and compare

Merge at the roots

Removing weight Distinction Assumption

Suppose edge weights are not distinct, and Kruskal's algorithm sorts edges so

$$c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_m}$$

Suppose Kruskal finds tree T of weight $c(T)$, but the optimal solution T^* has cost $c(T^*) < c(T)$.

Perturb each of the weights by a very small amount so that

$$c'_{e_1} < c'_{e_2} < \cdots \leq c'_{e_m}$$

If the perturbation is small enough, $c'(T^*) < c(T)$.

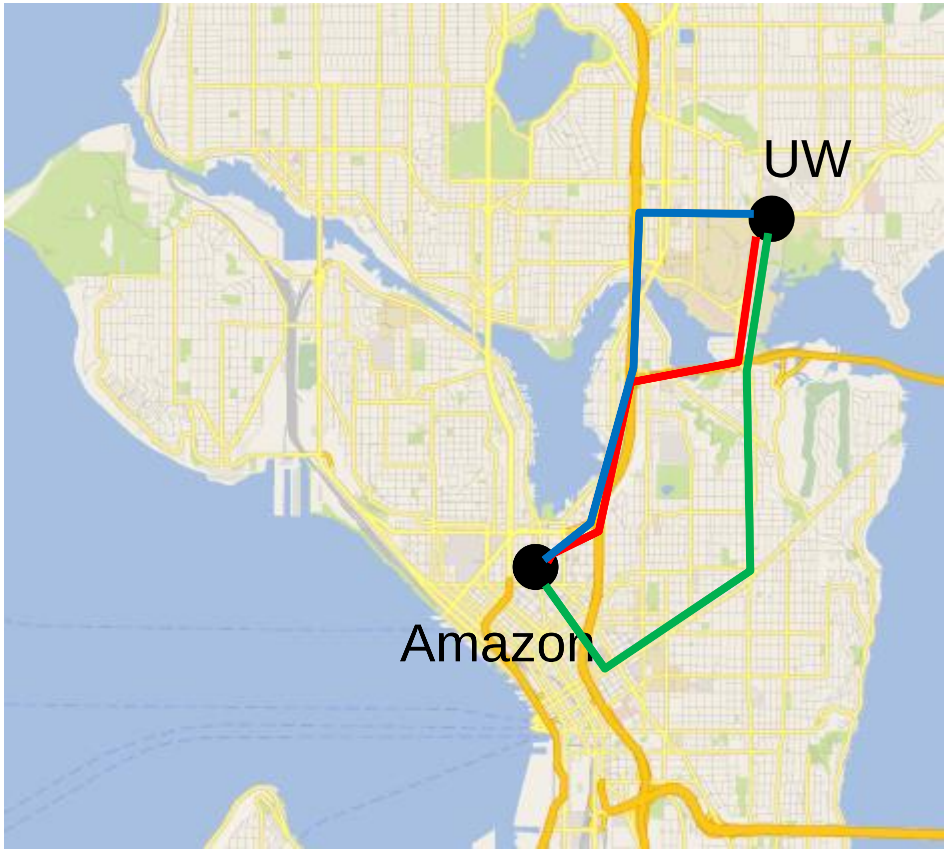
However, this contradicts the correctness of Kruskal's algorithm, since the algorithm will still find T, and Kruskal's algorithm is correct if all weiths are distinct.

■

Single Source Shortest Path

Given an (un)directed graph G=(V,E) with non-negative edge weights $c_e \geq 0$ and a start vertex s

Find length of shortest paths from s to each vertex in G



Dijkstra's Algorithm

Maintain a set S of vertices whose shortest paths are known

- initially **S={s}**

Maintaining current best lengths of paths that only go through **S** to each of the vertices in G

- Path-lengths to elements of **S** will be right, to **V-S** they might not be right

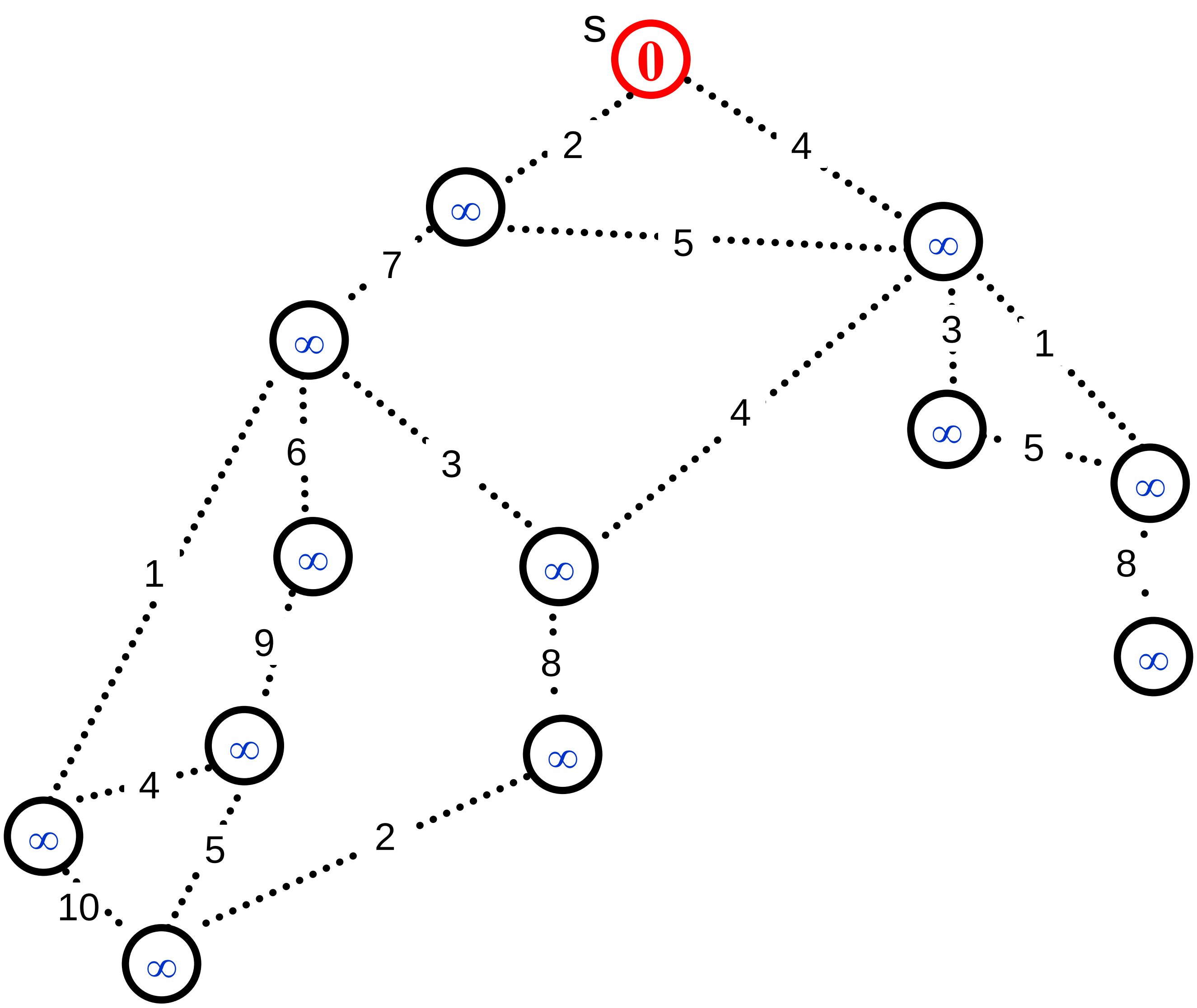
Repeatedly add vertex **v** to **S** that has the shortest path-length of any vertex in **V-S**

- Update path lengths based on new paths through v

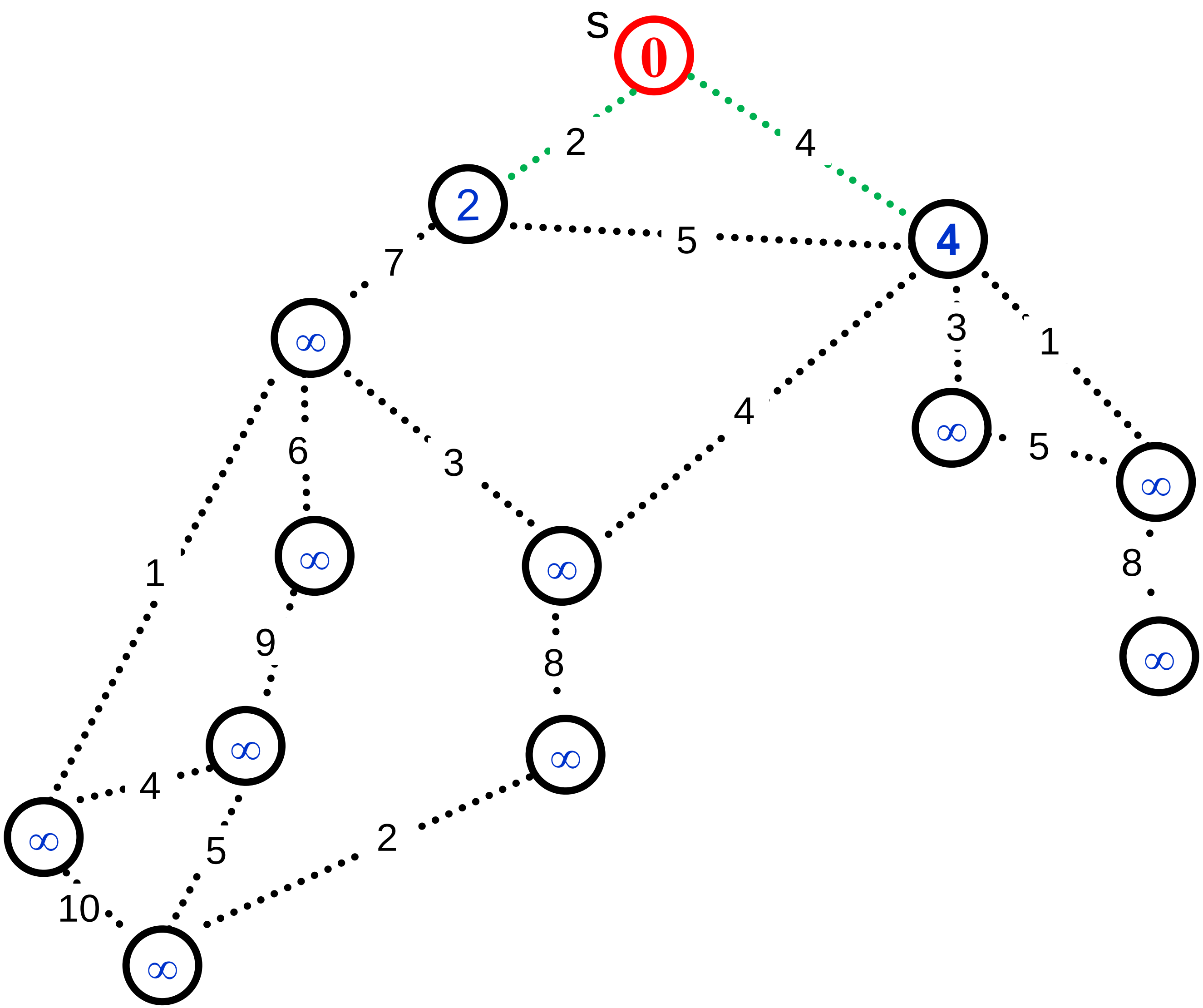
Dijkstra's Algorithm

```
Dijkstra(G, c, s) {
   d[s] ← 0
   foreach (v ∈ V) d[v] ← ∞ //This is the key of node v
   foreach (v ∈ V) insert v onto a priority queue Q
   Initialize set of explored nodes S ← {s}

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (d[u]+c_e < d[v]))
             d[v] ← d[u] + c_e
             Decrease key of v to d[v].
             Parent(v) ← u
}
```

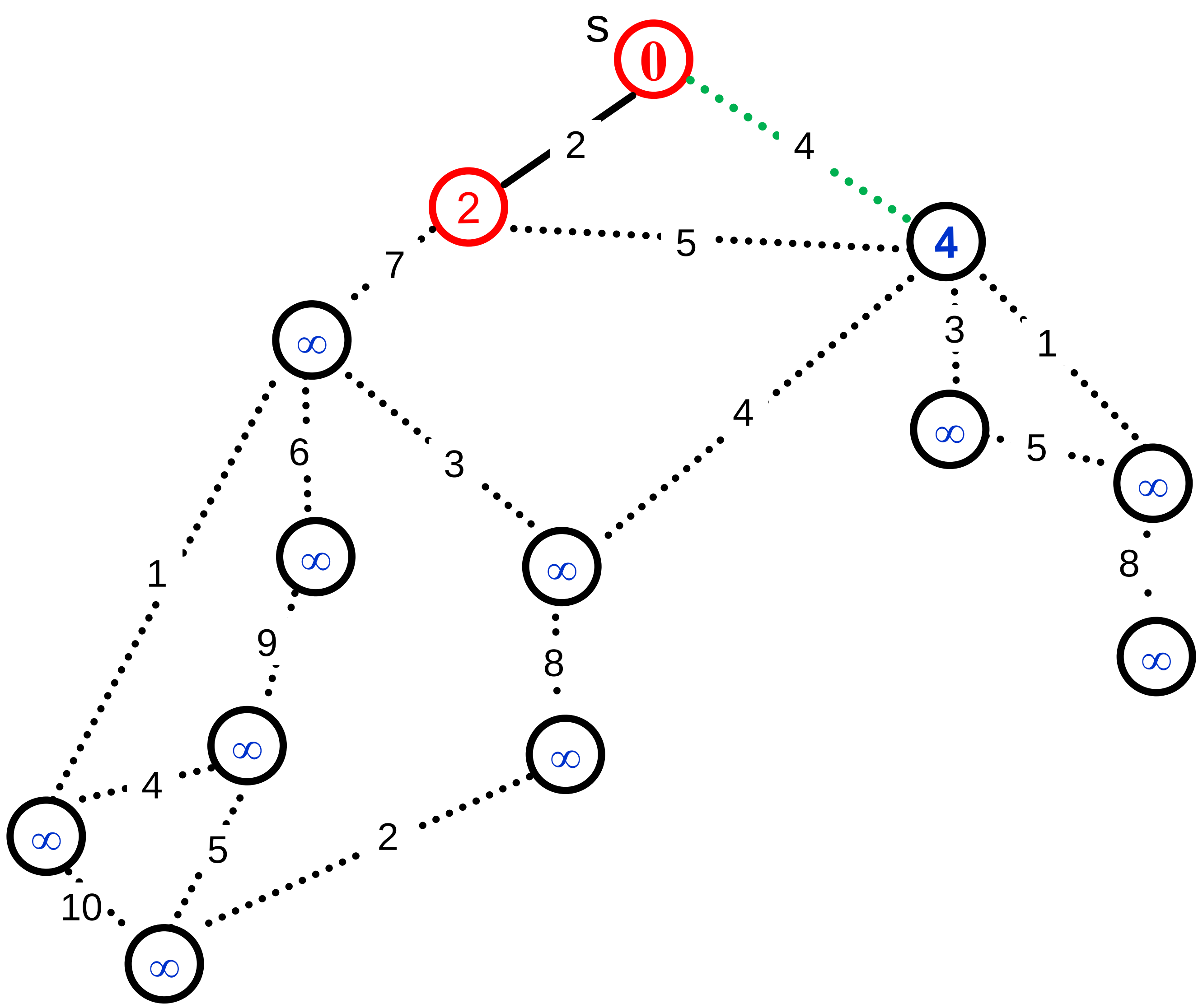
Dijkstra’s Algorithm: Example



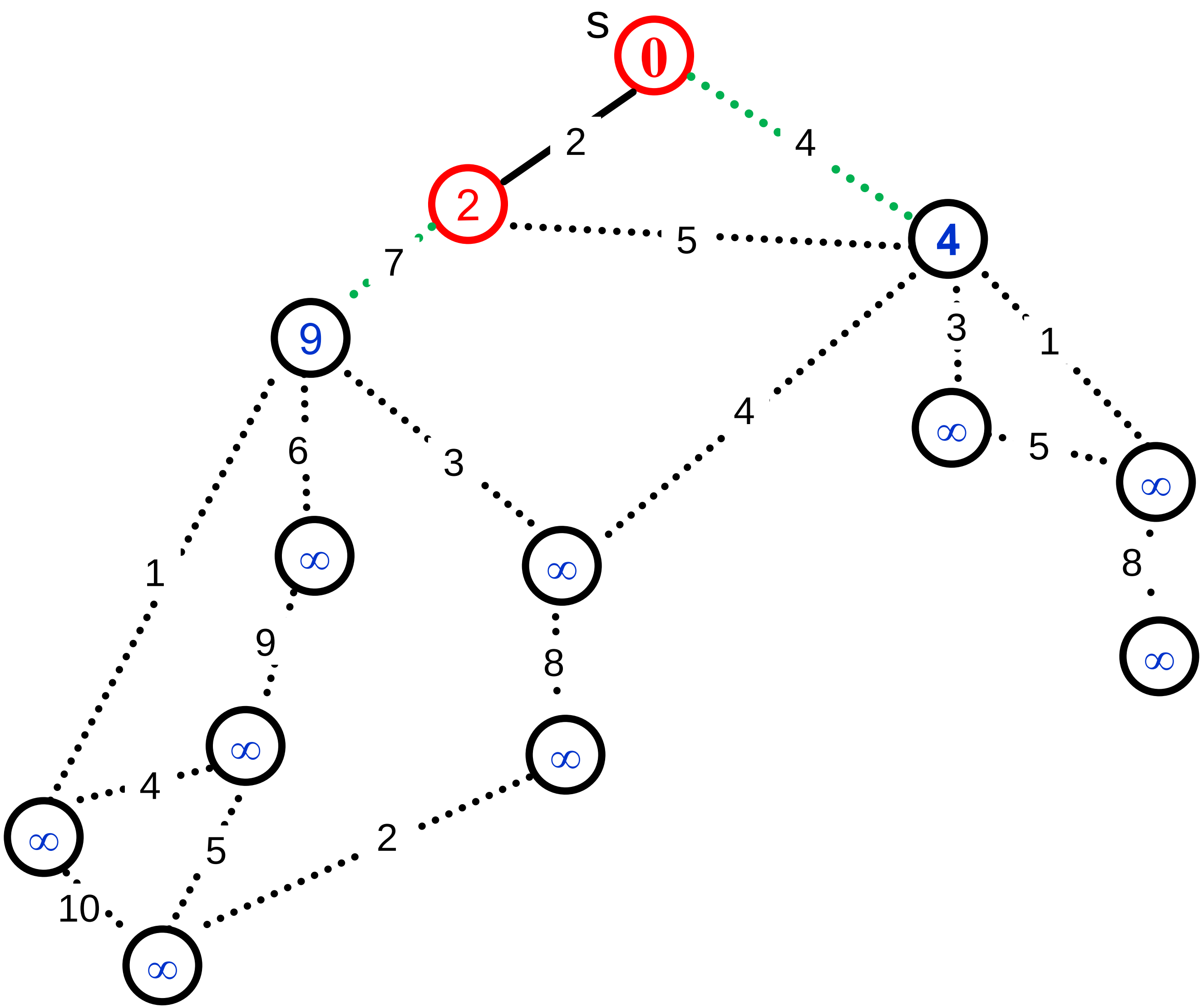
Dijkstra’s Algorithm: Example



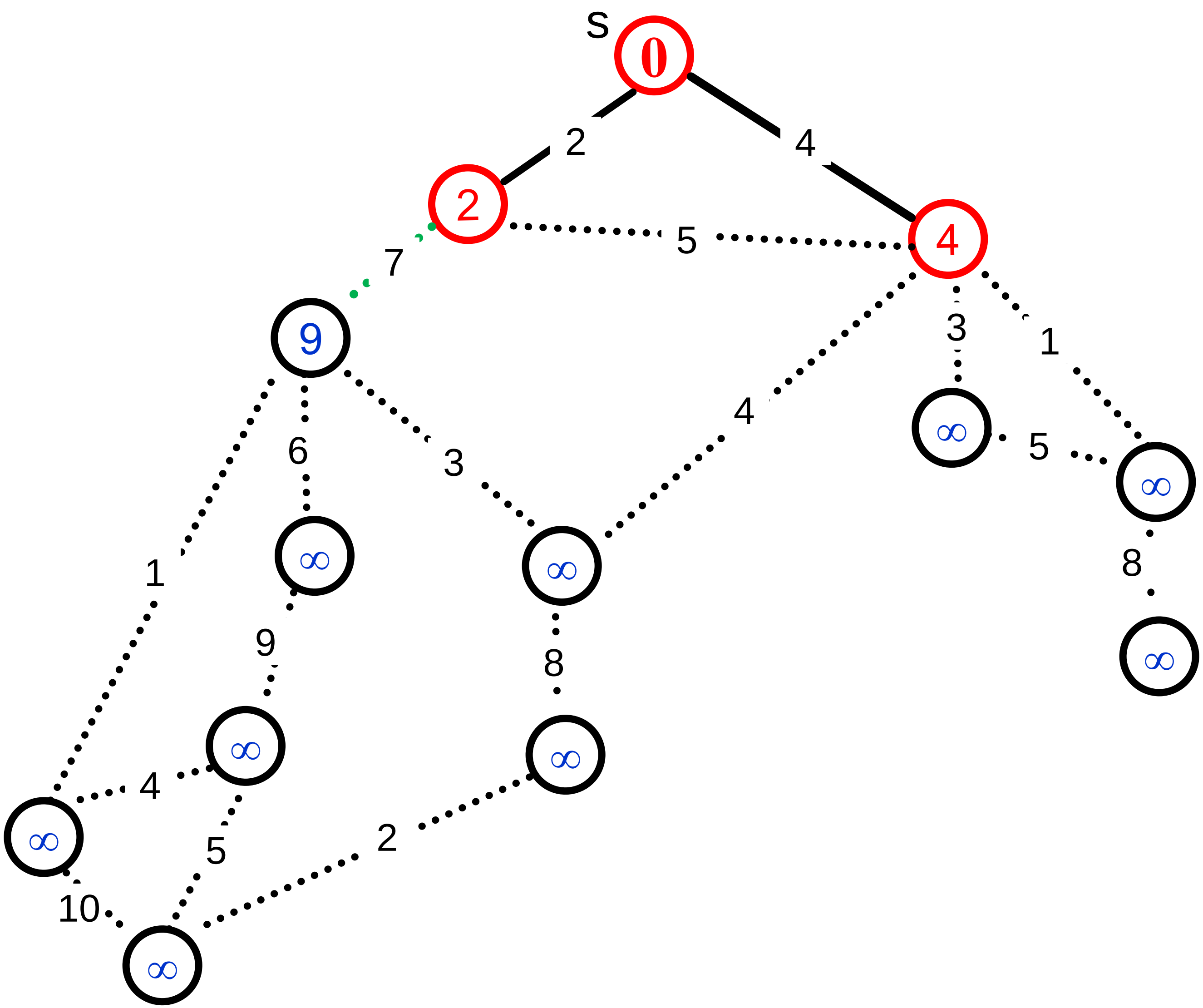
Dijkstra's Algorithm: Example



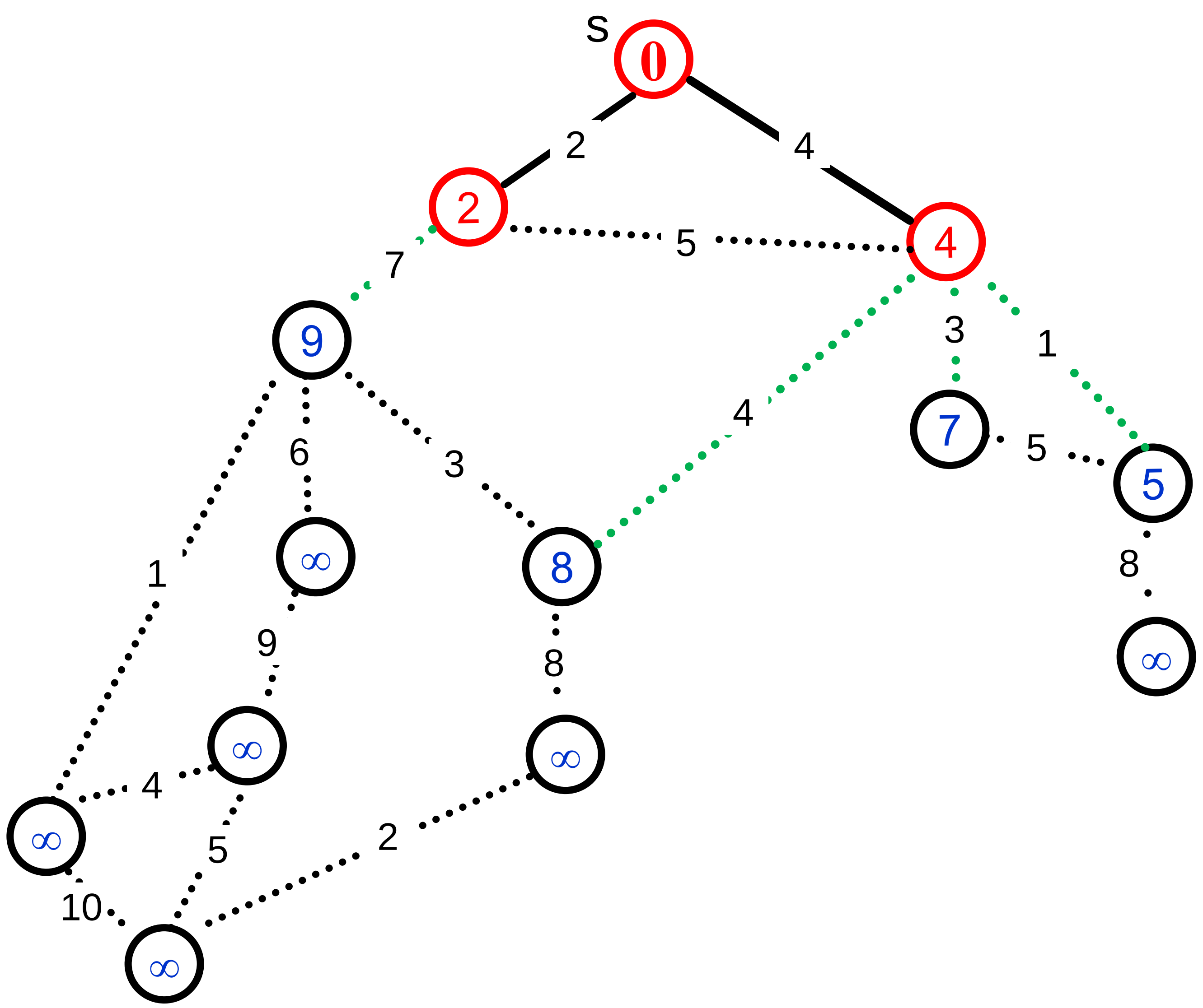
Dijkstra's Algorithm: Example



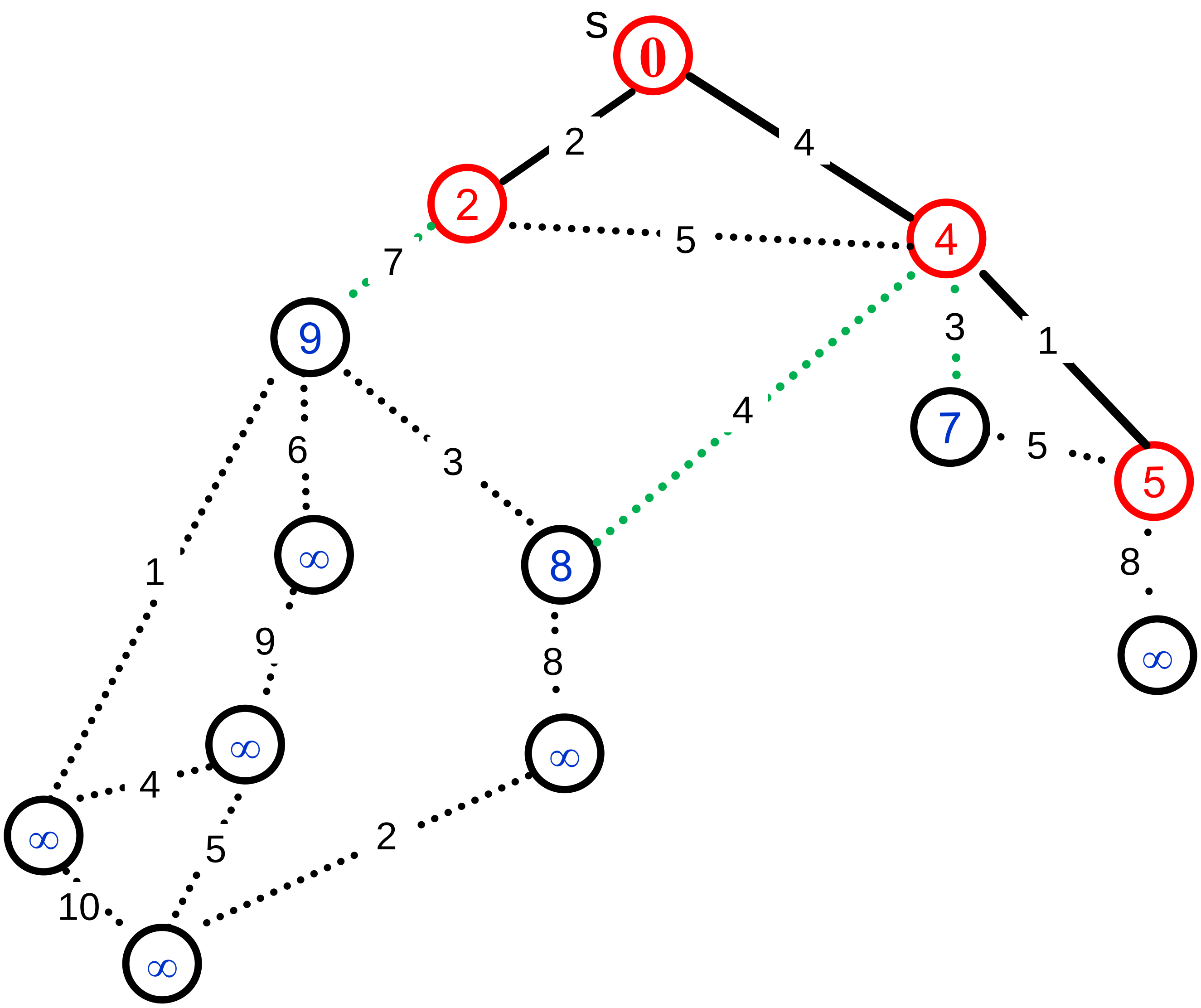
Dijkstra’s Algorithm: Example



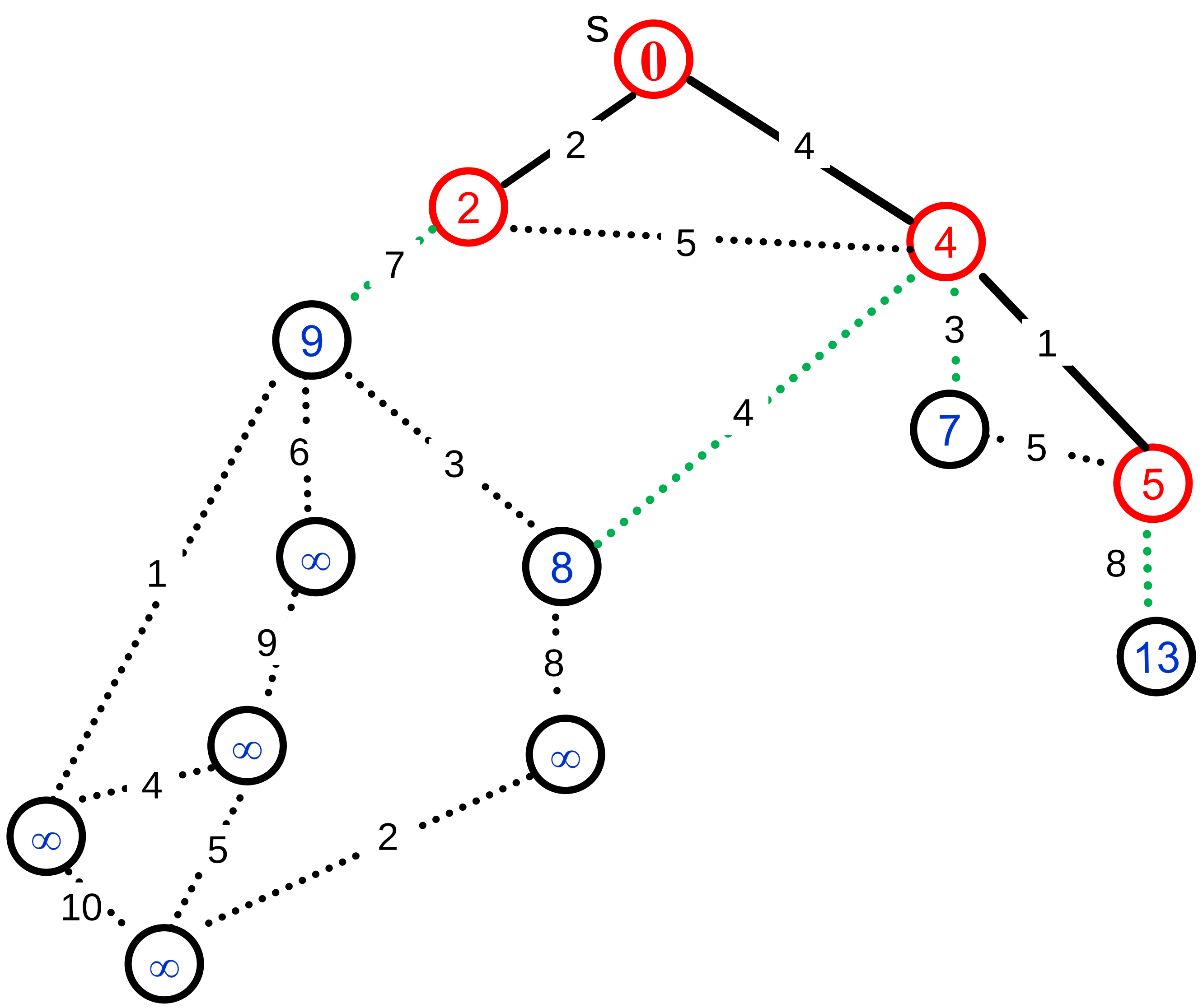
Dijkstra's Algorithm: Example



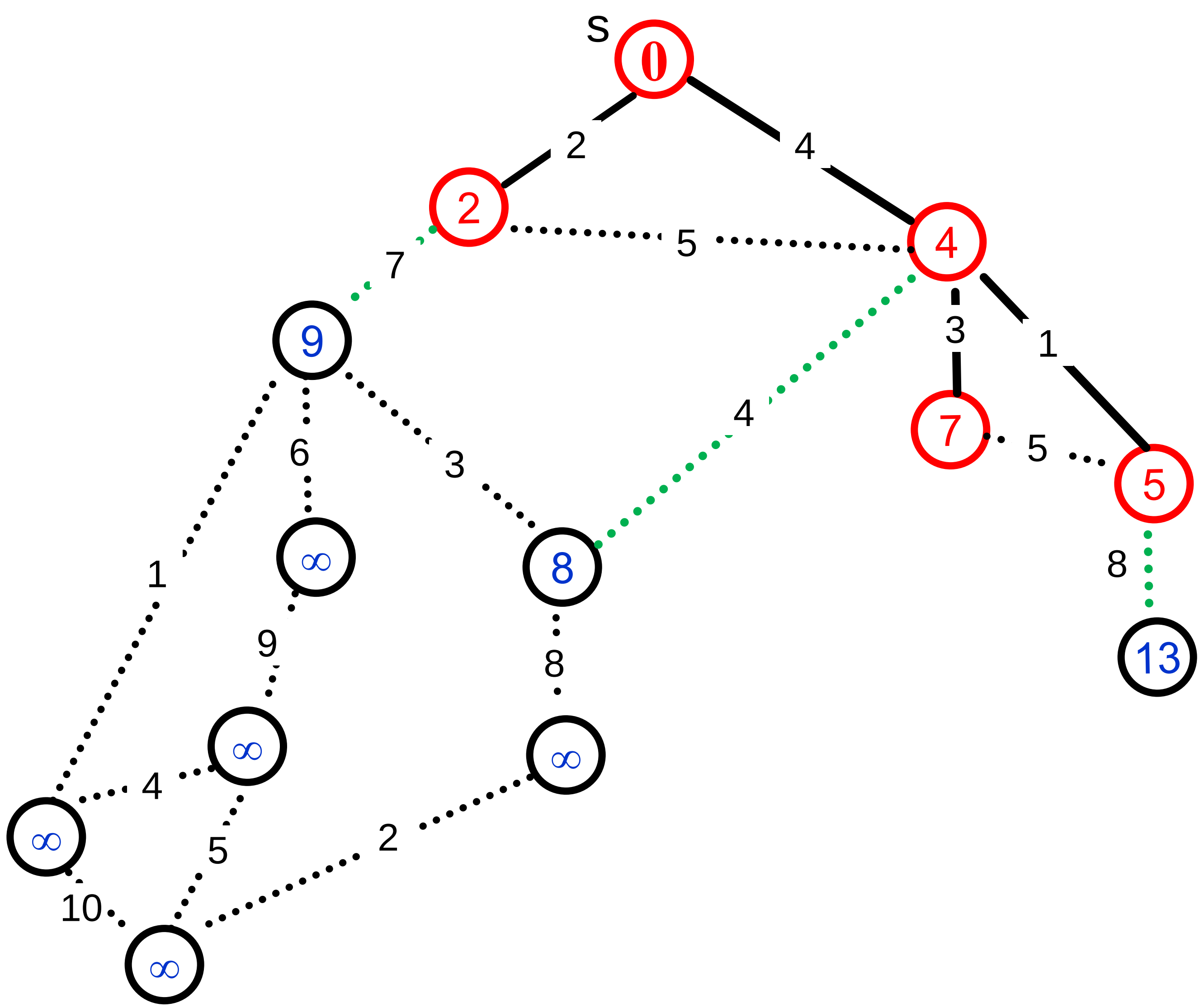
Dijkstra's Algorithm: Example



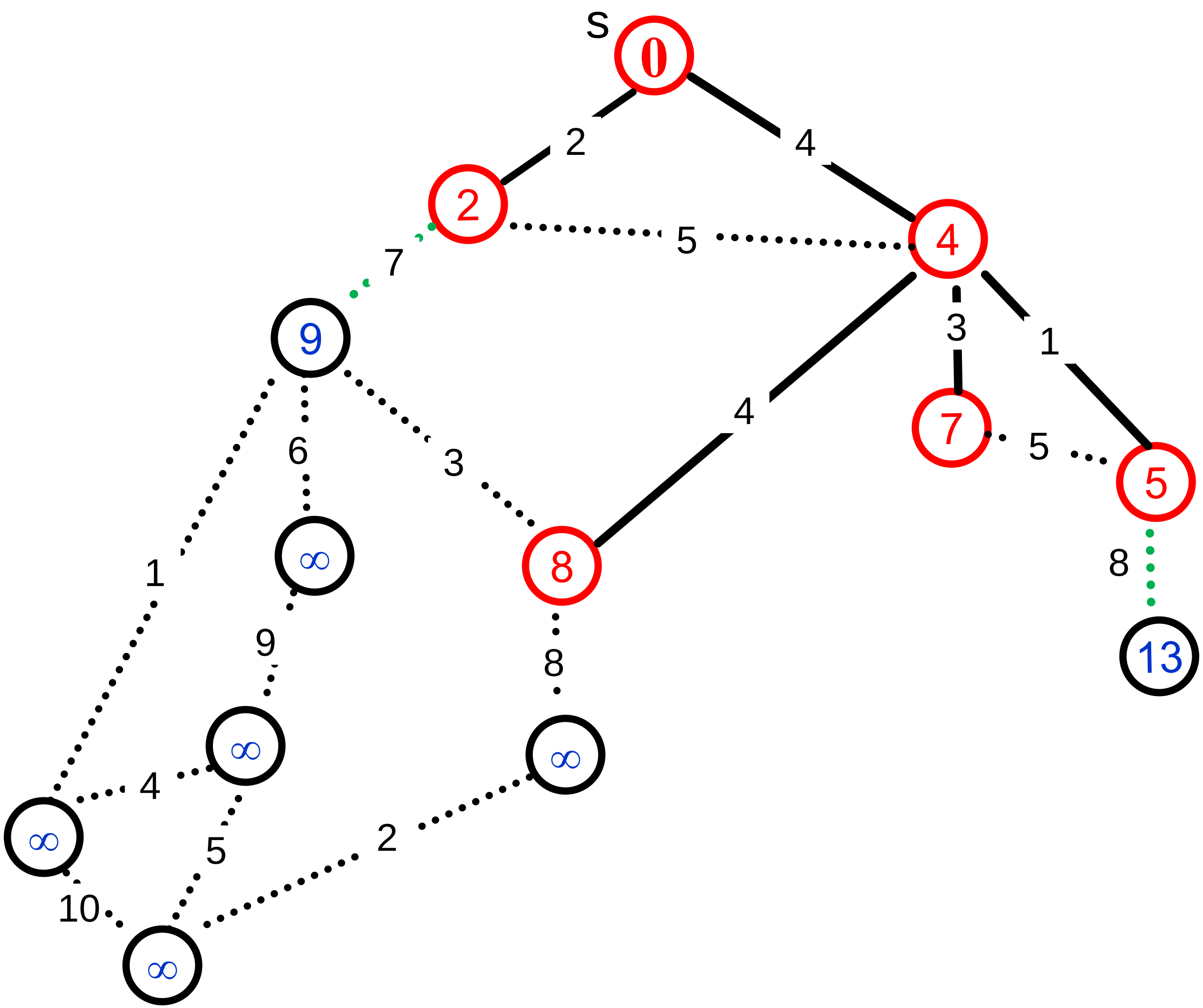
Dijkstra's Algorithm: Example



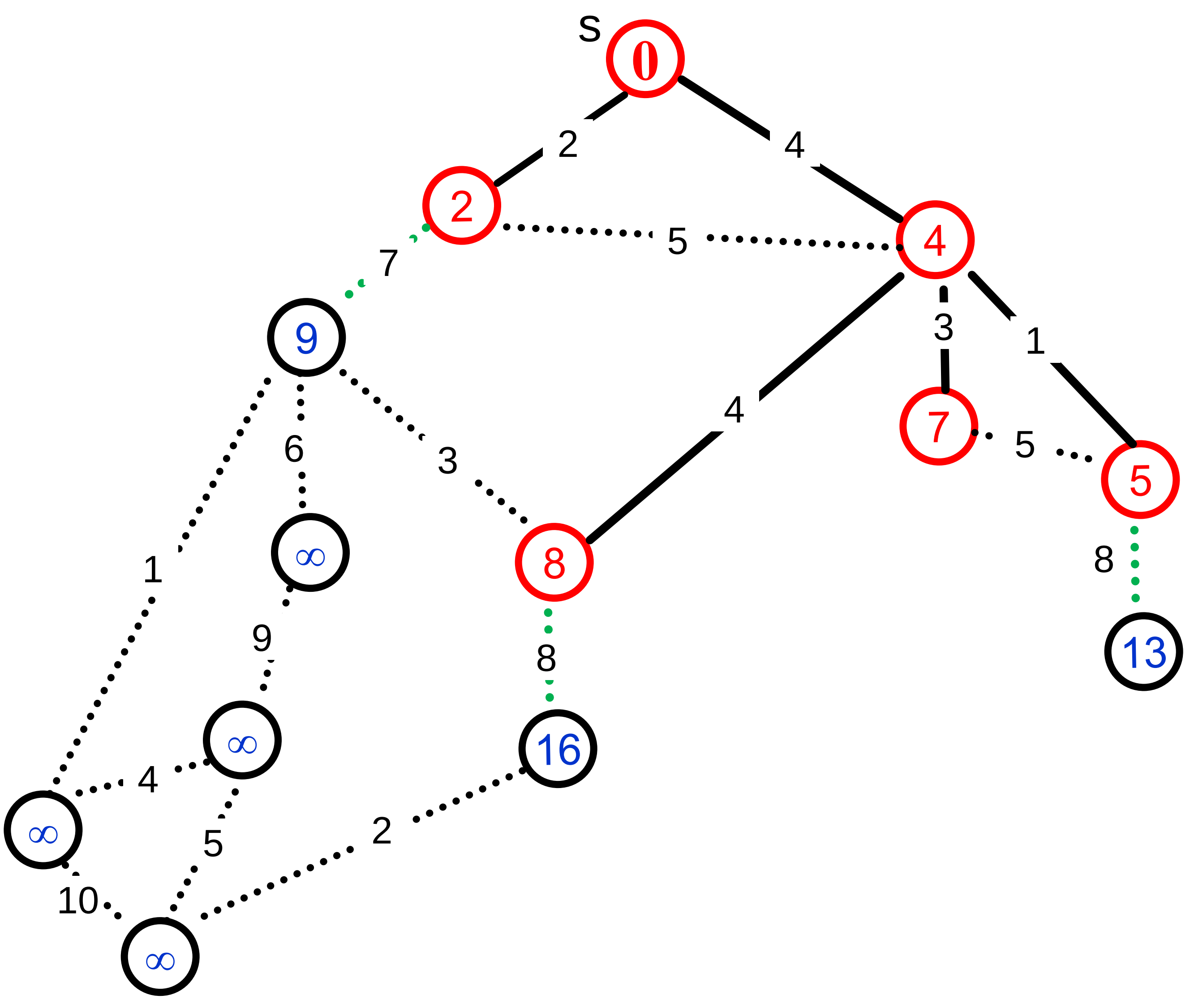
Dijkstra’s Algorithm: Example



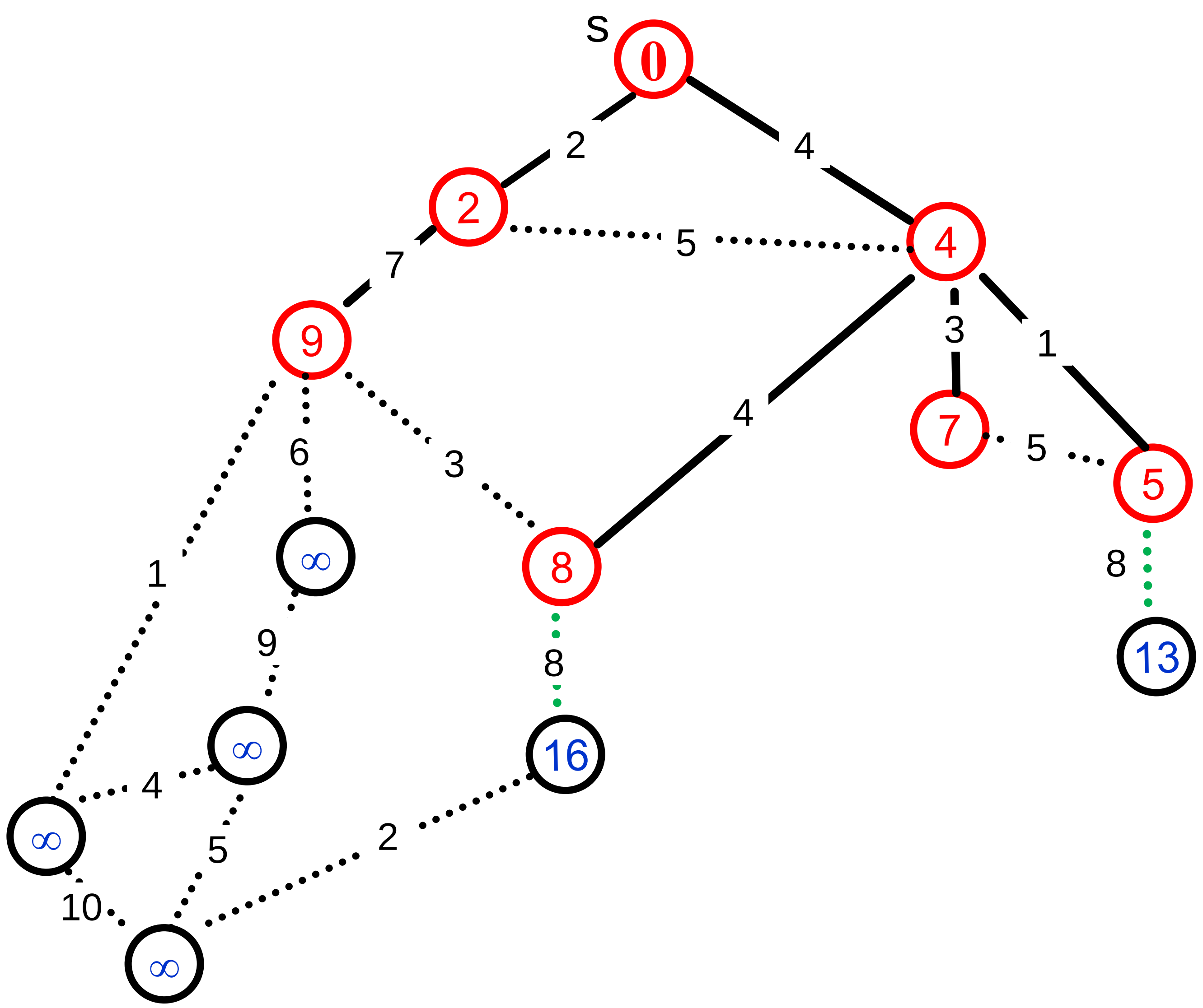
Dijkstra's Algorithm: Example



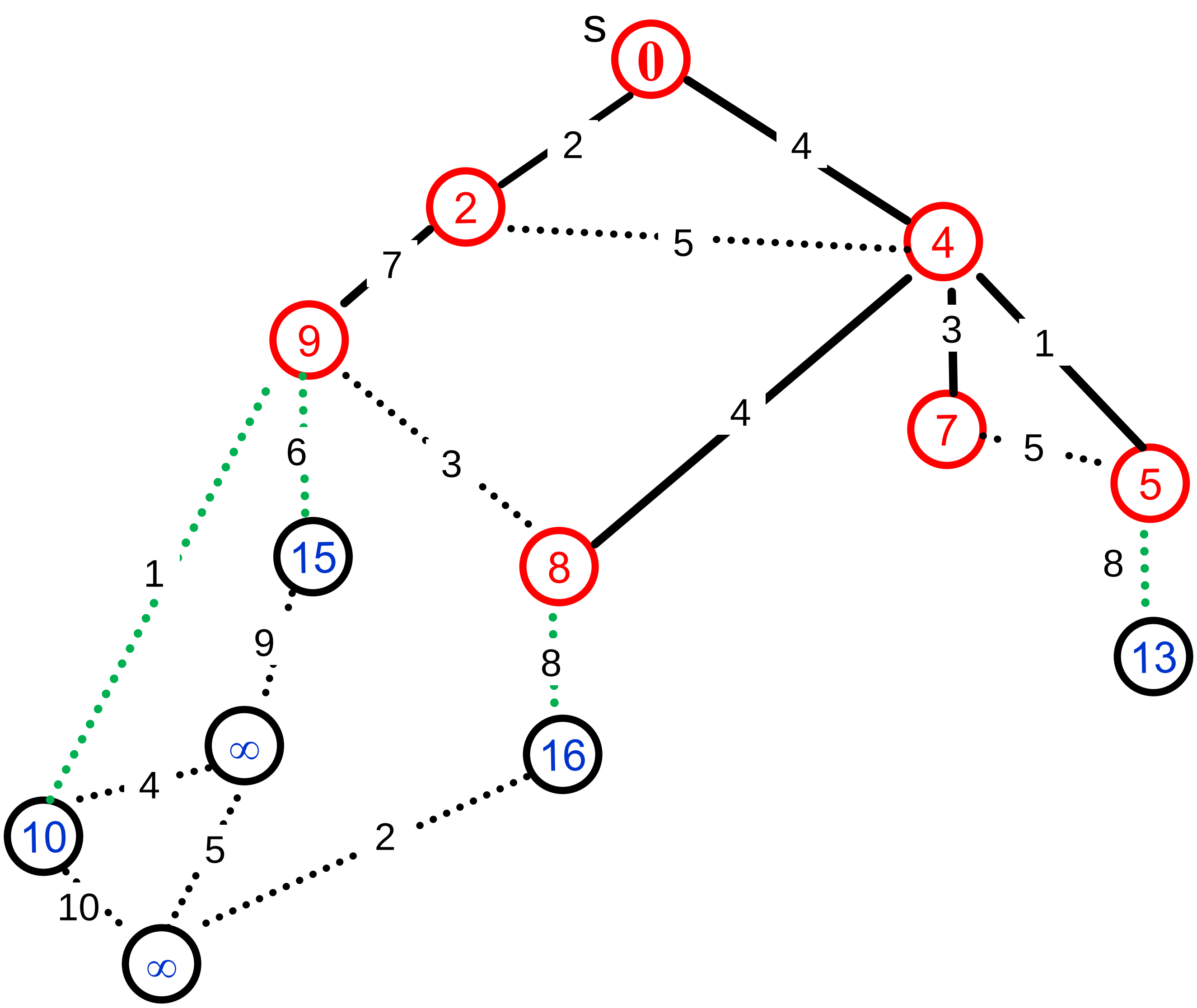
Dijkstra's Algorithm: Example



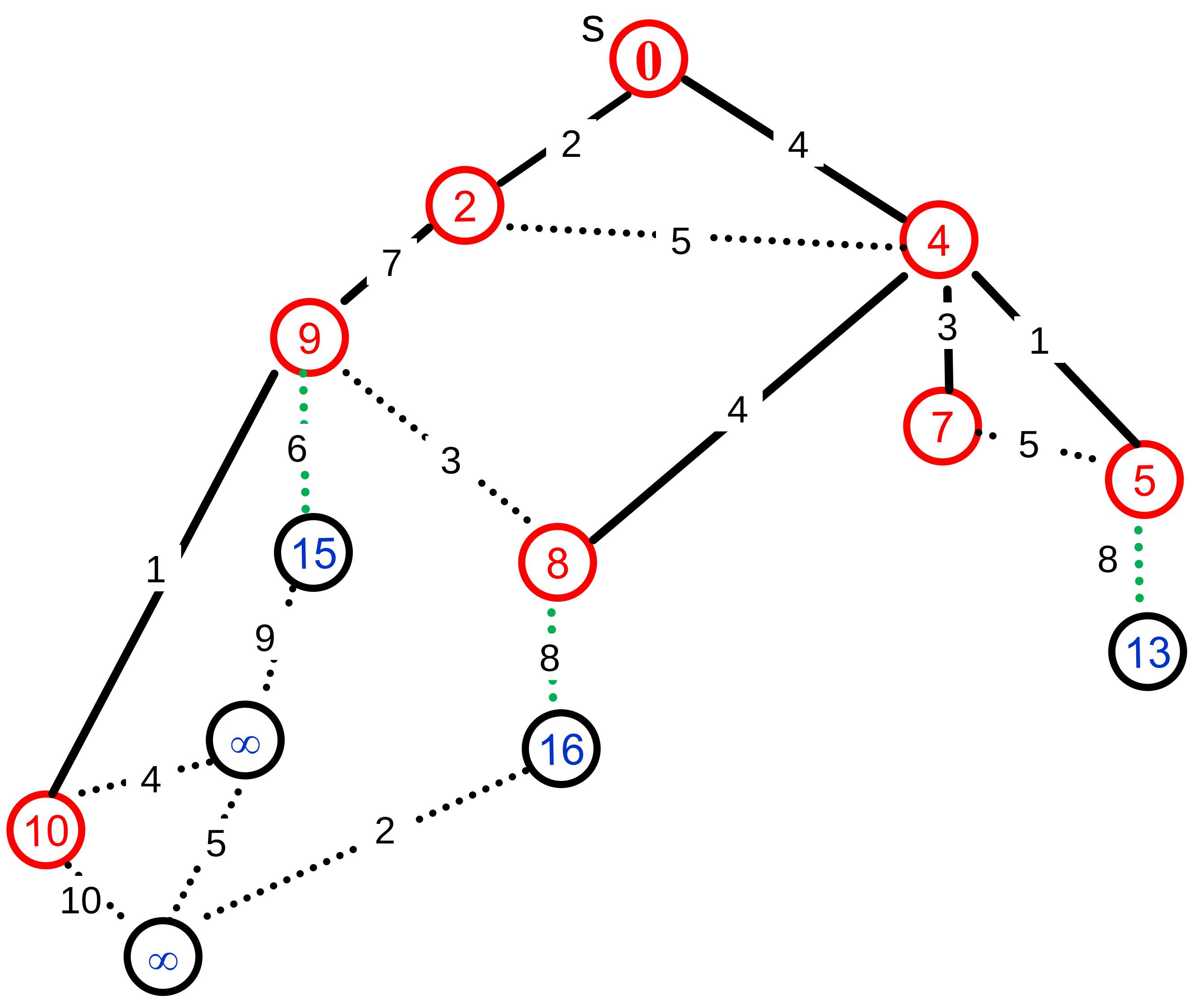
Dijkstra’s Algorithm: Example



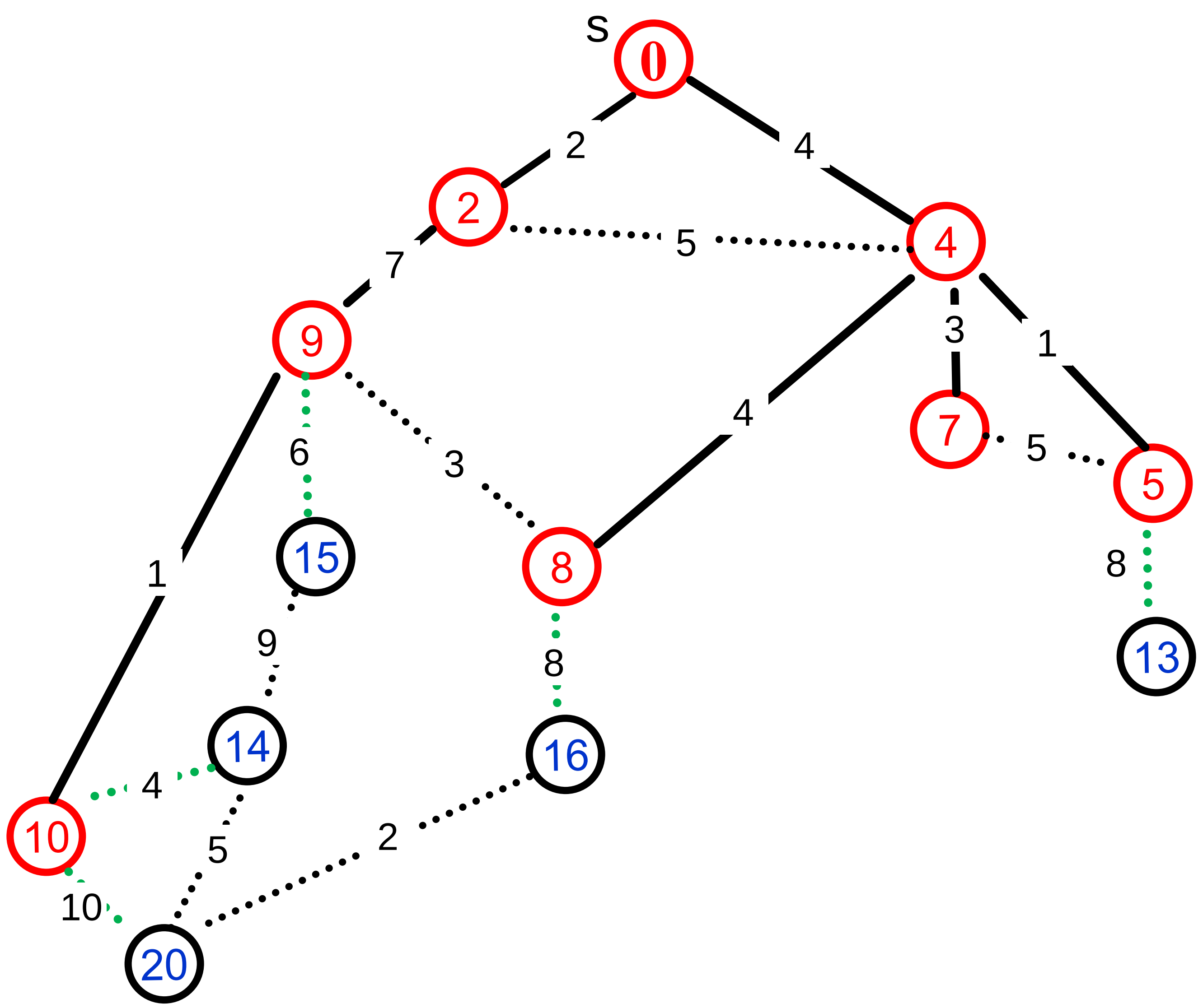
Dijkstra's Algorithm: Example



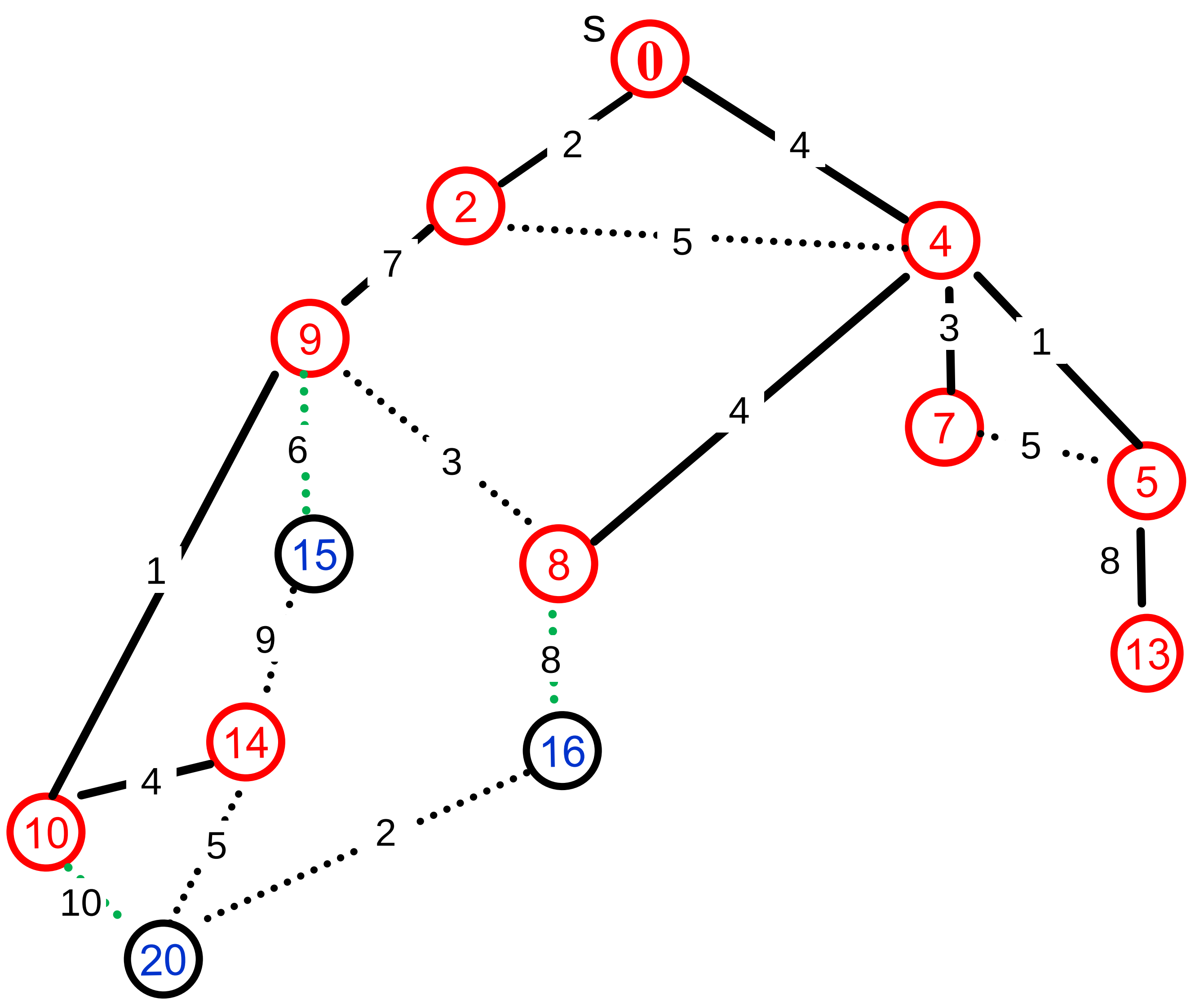
Dijkstra's Algorithm: Example



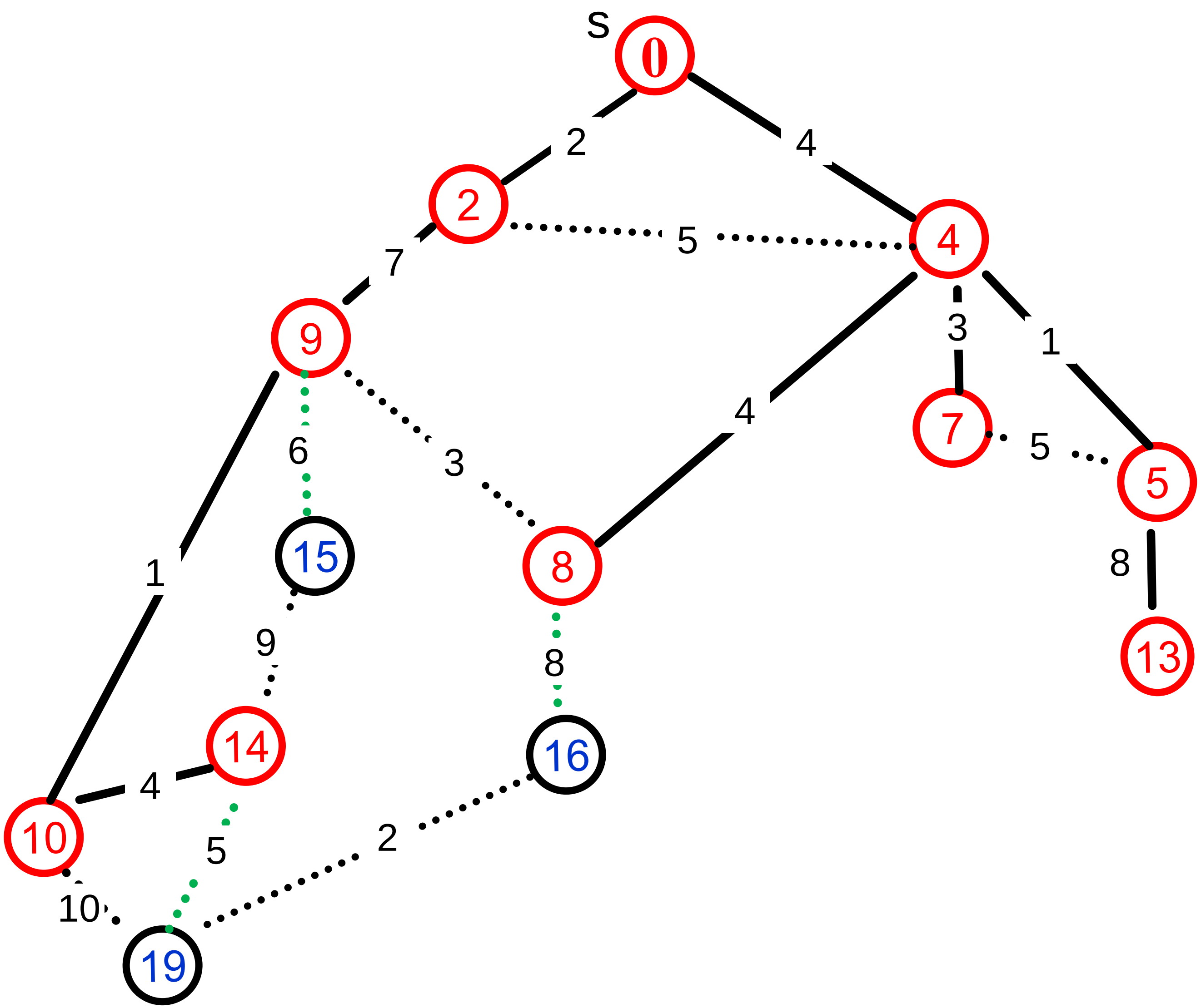
Dijkstra’s Algorithm: Example



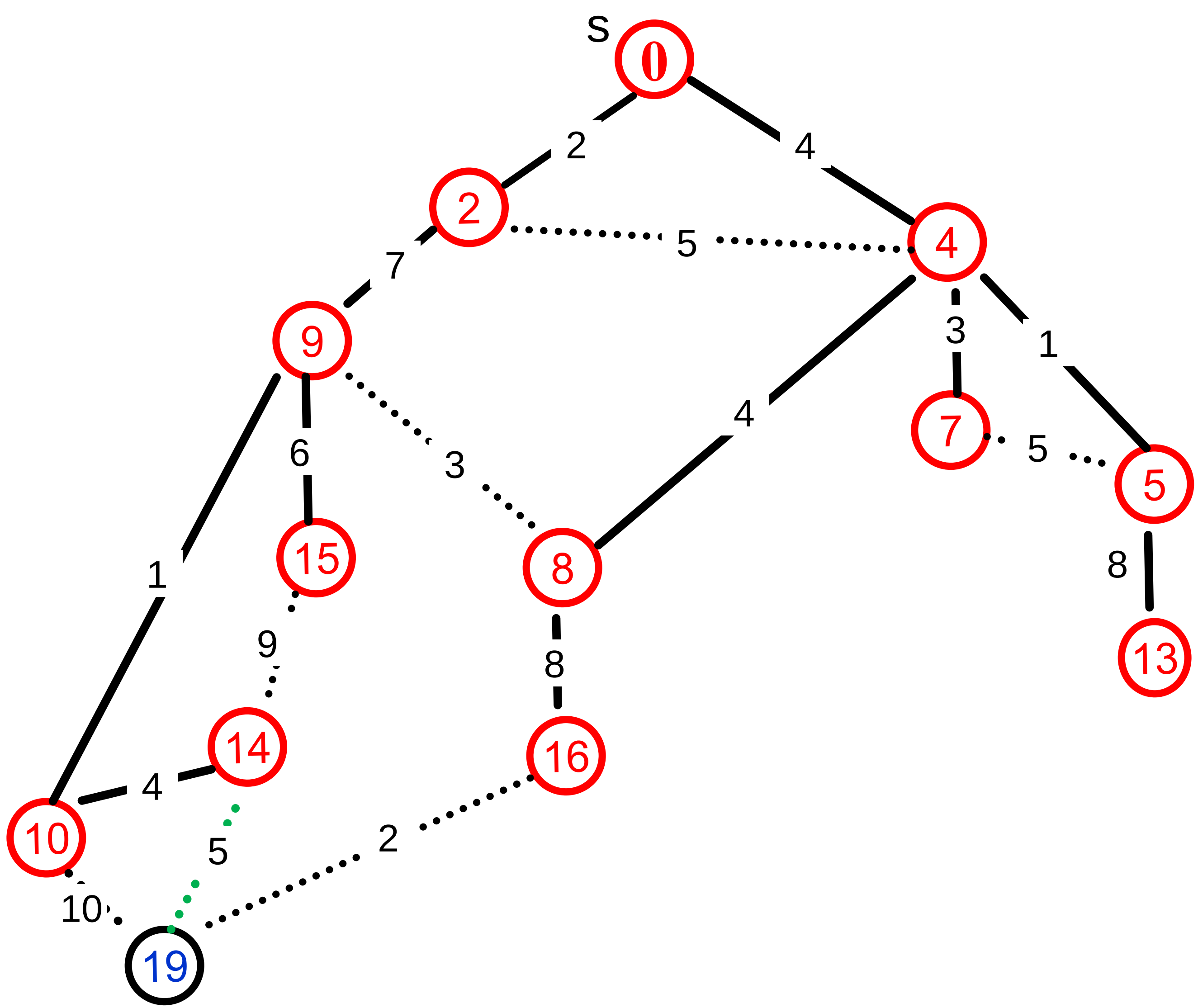
Dijkstra's Algorithm: Example



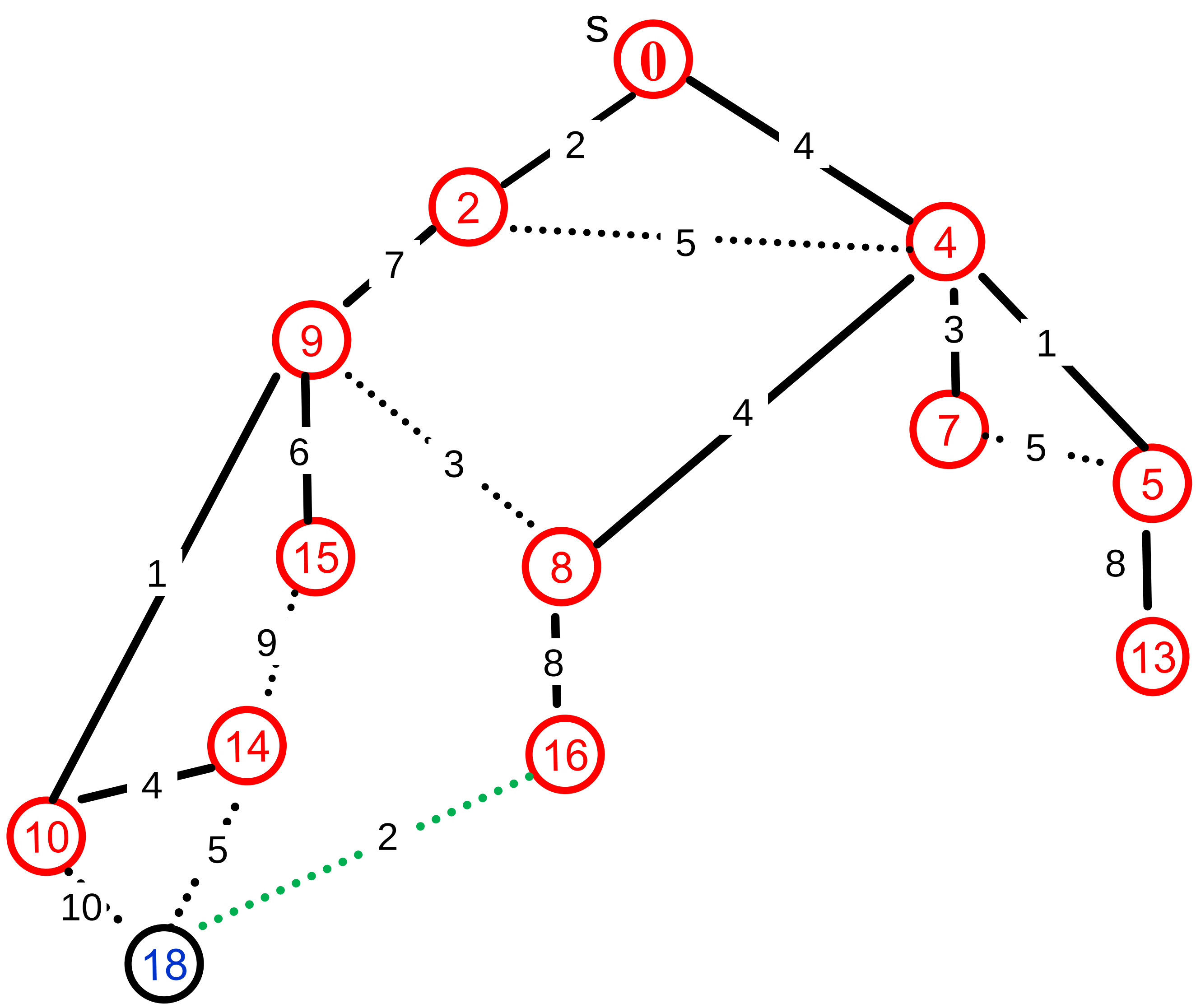
Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Example



Dijkstra’s Algorithm: Example