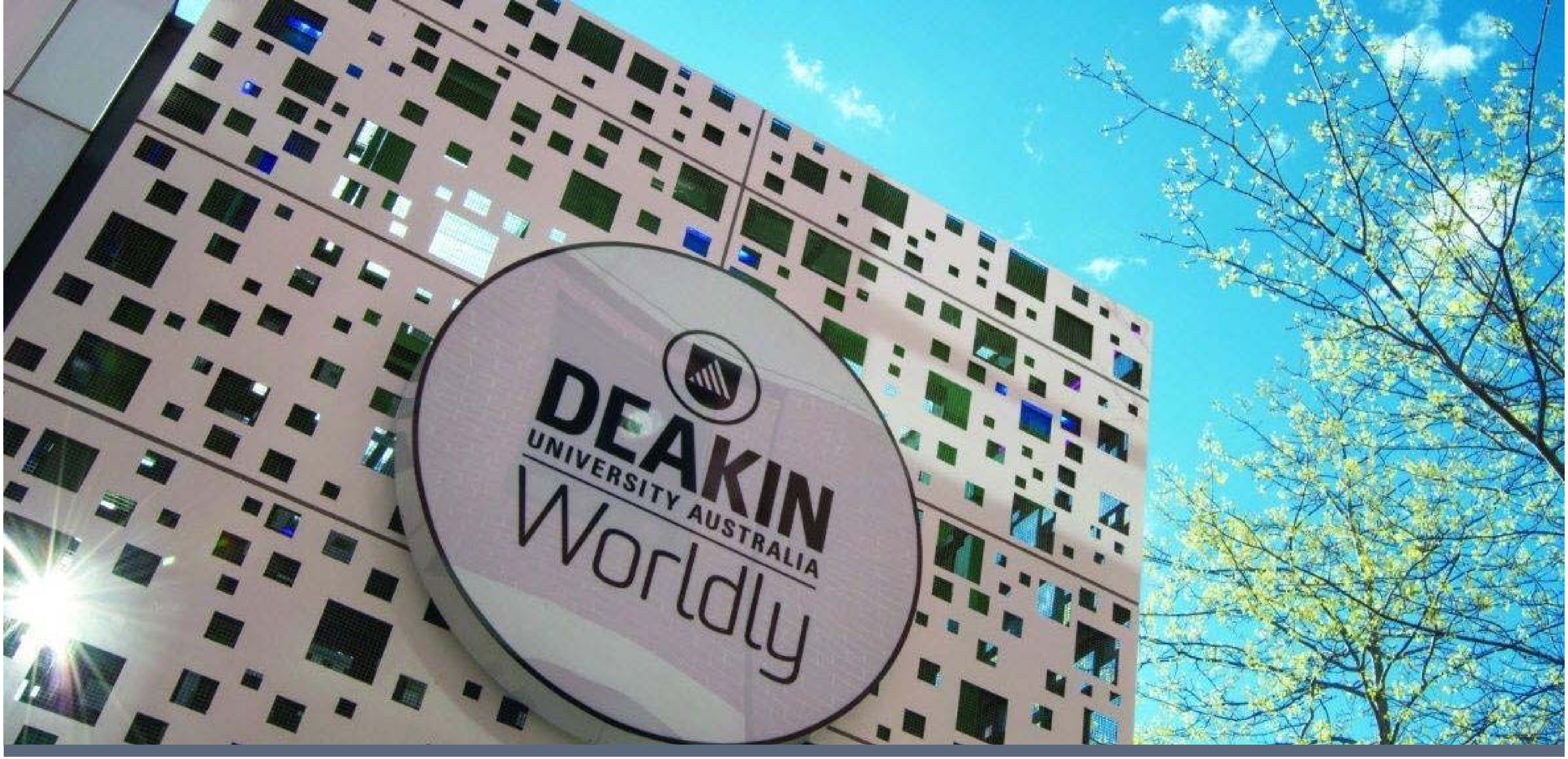


Lecture 9. The Shortest Path Problem

SIT221 Data Structures and Algorithms

Weighted Graph: Notation

- A graph is a pair $G = (V, E)$, where
  - V is a set of nodes, called vertices.
  - E is a collection of pairs of vertices, called edges.
  - We denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges.

- Often there are weights/costs assigned to the edges: $c: E \rightarrow R$
- A graph is weighted if each edge is given a numerical weight.

  Edge weights may represent, distances, costs, etc.

  For example, in a flight route graph, the weight of an edge represents the distance in kilometers between the endpoint airports.

Shortest Path: Problem Formulation

Given a weighted directed graph $G = (V, E)$, and two vertices s and v, we want to find a path of minimum total cost among all possible paths between s and v.

Given a path $p = (e_1, e_2, \dots, e_k)$ consisting of k edges, the cost of the path is the sum of the weights of its edges:

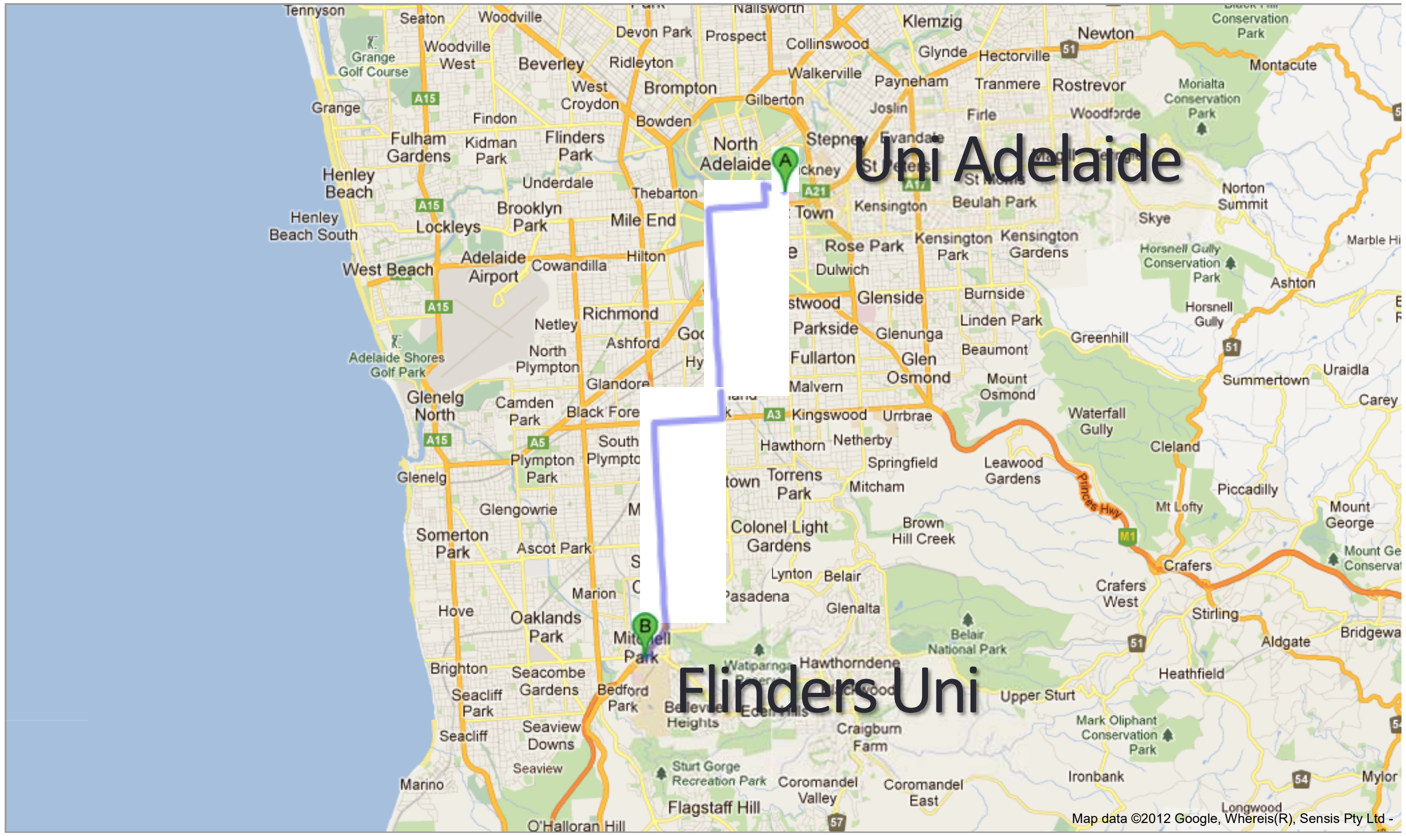
$$c(p) = \sum_{i=1}^{k} c(e_i).$$

Single-Source Shortest Path Problem:

Compute for a given node s of V a shortest path to any other node in V (if it exists).

We assume that edge weights are non-negative.

Shortest Path: Applications



Computation of shortest path is one of the classical problems.

- Route planning / driving directions
- Internet packet routing
- Flight reservations

Shortest Path: Dijkstra's Algorithm

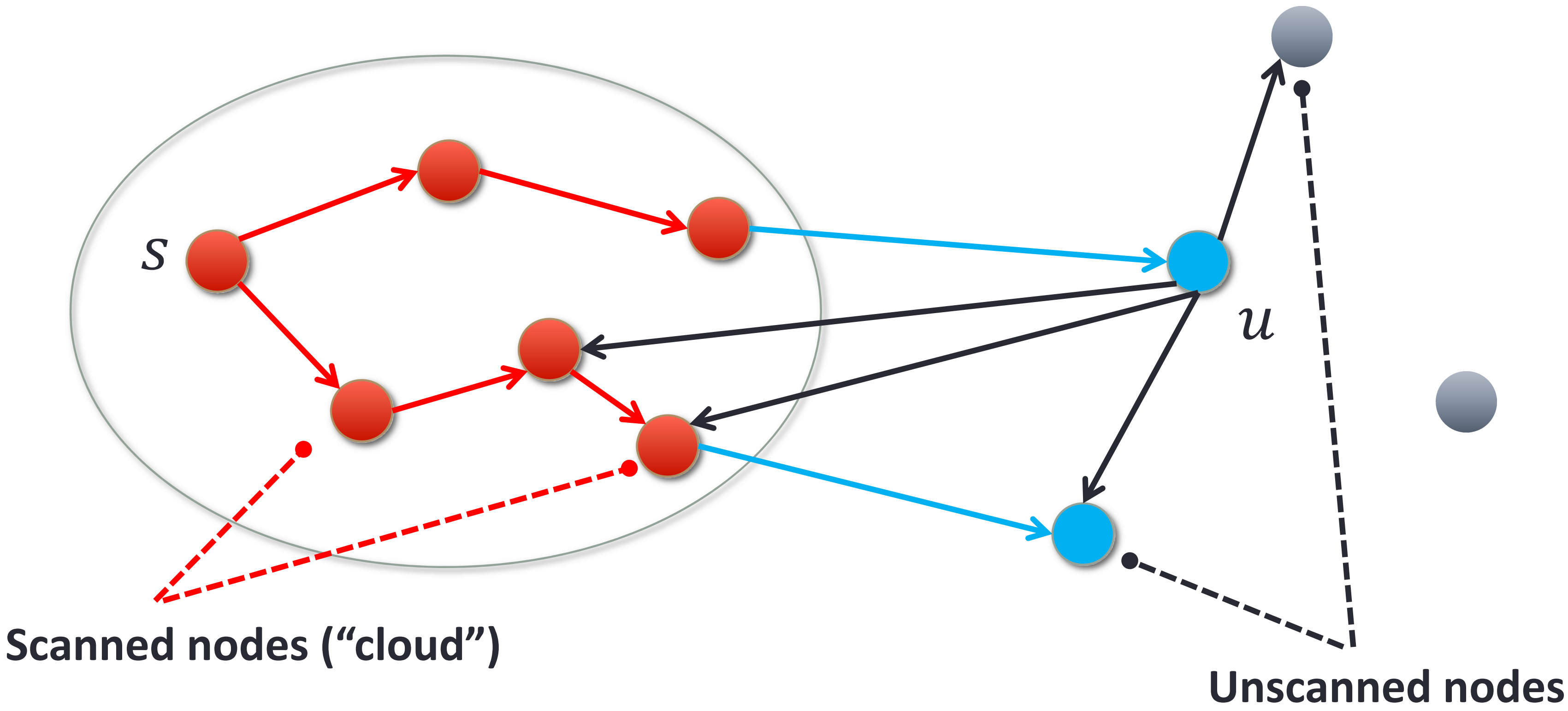
- Remember BFS for computing all shortest paths in an unweighted graph.
- In iteration i, we computed all shortest paths having i edges.
- Dijkstra's algorithm obtains in iteration i a shortest path to the node of the i^{th} smallest distance from s.
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s, assuming that:
  - the graph is connected
  - the edge weights are nonnegative

Dijkstra's Algorithm: Idea

- We grow a "**cloud**" of vertices, beginning with s and eventually covering all the vertices.
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices.
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$.
  - We update the labels of the vertices adjacent to u.

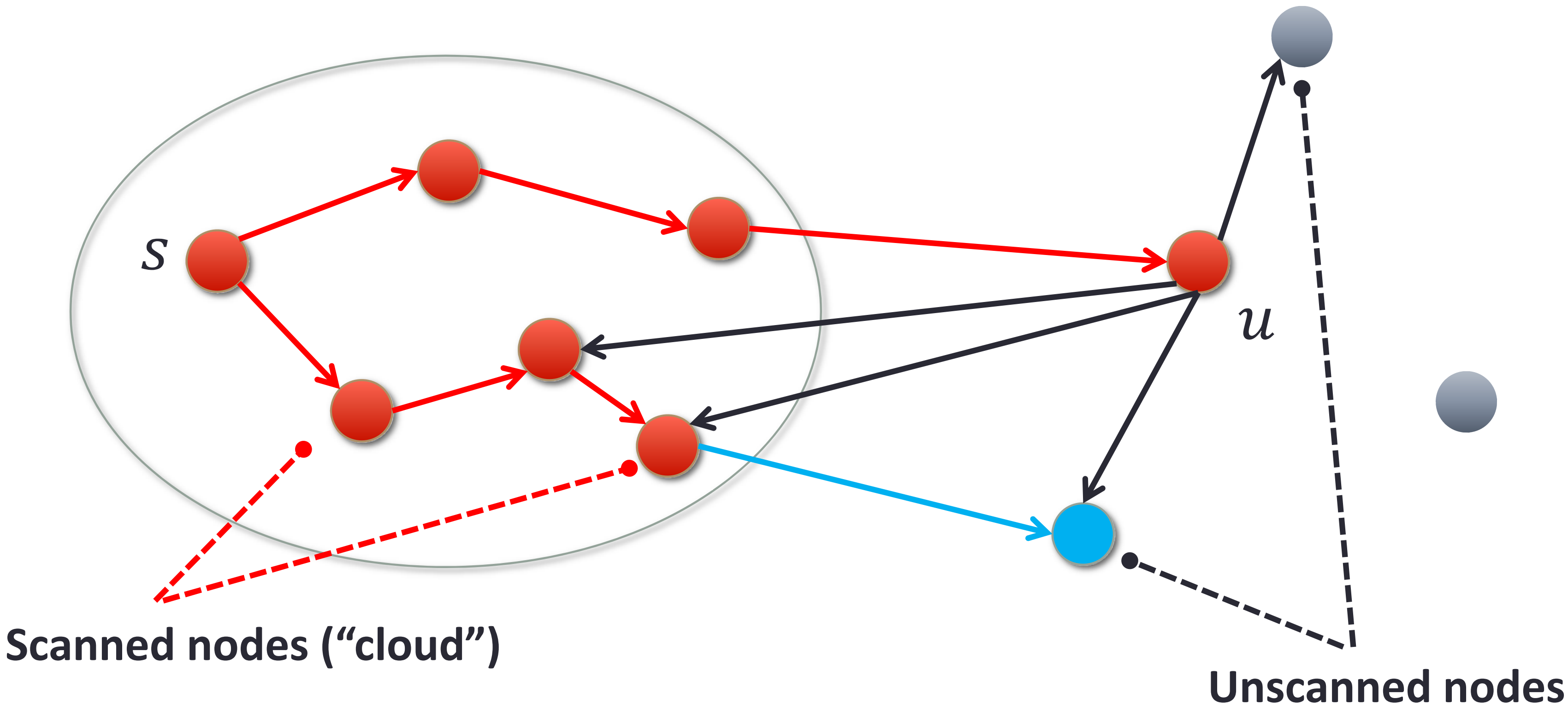
Dijkstra's Algorithm: Idea

We call a node u **unscanned** if no shortest path from s to u has been found so far.



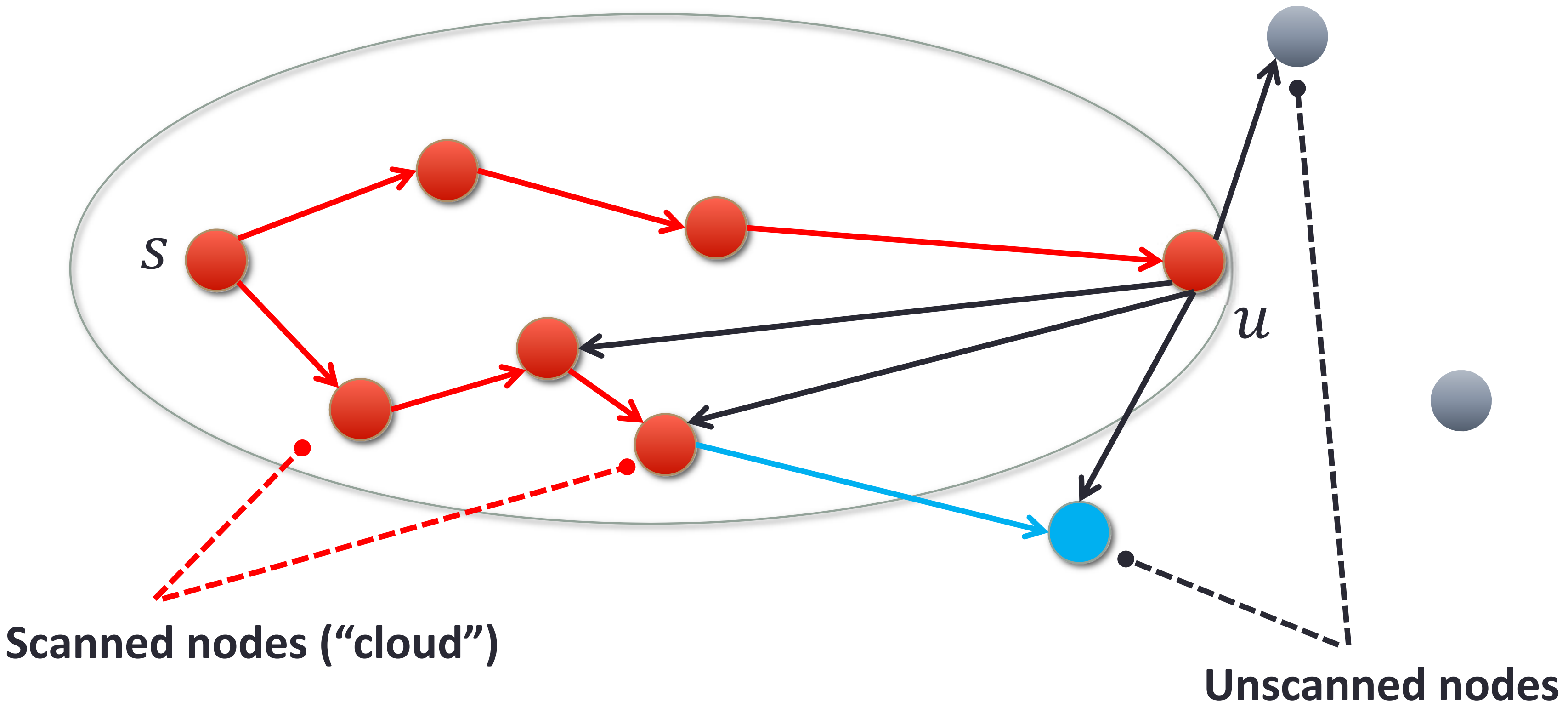
Dijkstra's Algorithm: Idea

Make unscanned node u **scanned** that would get the minimal tentative distance among all unscanned nodes.



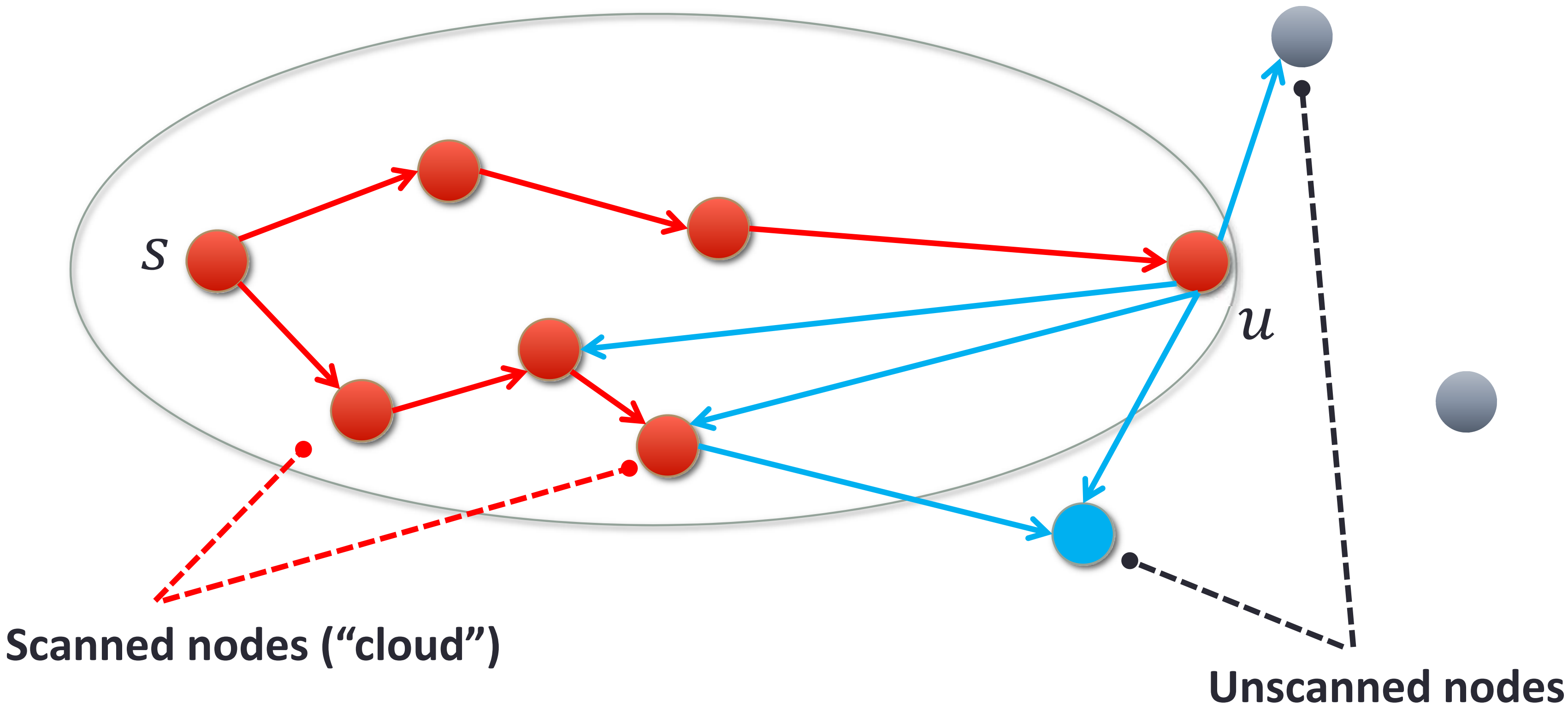
Dijkstra's Algorithm: Idea

Make unscanned node u **scanned** that would get the minimal tentative distance among all unscanned nodes.



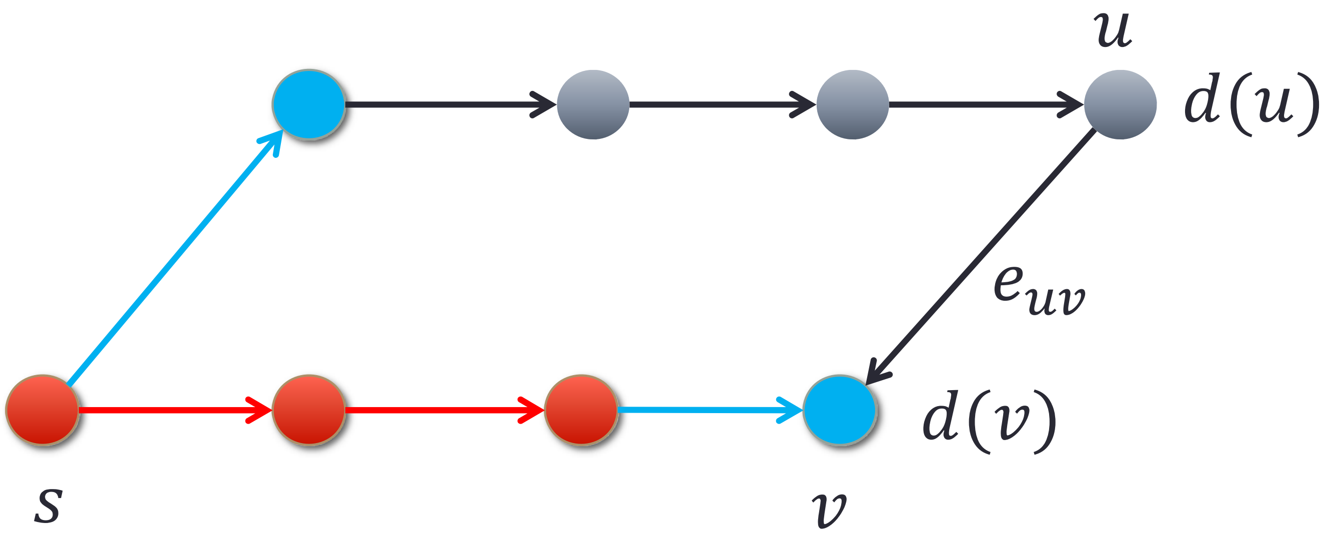
Dijkstra's Algorithm: Idea

Consider all edges leaving u and update distances using the **Relax Routine**.



Dijkstra's Algorithm: Edge Relaxation

We may to update a previous path from s to v if we find a shorter path.

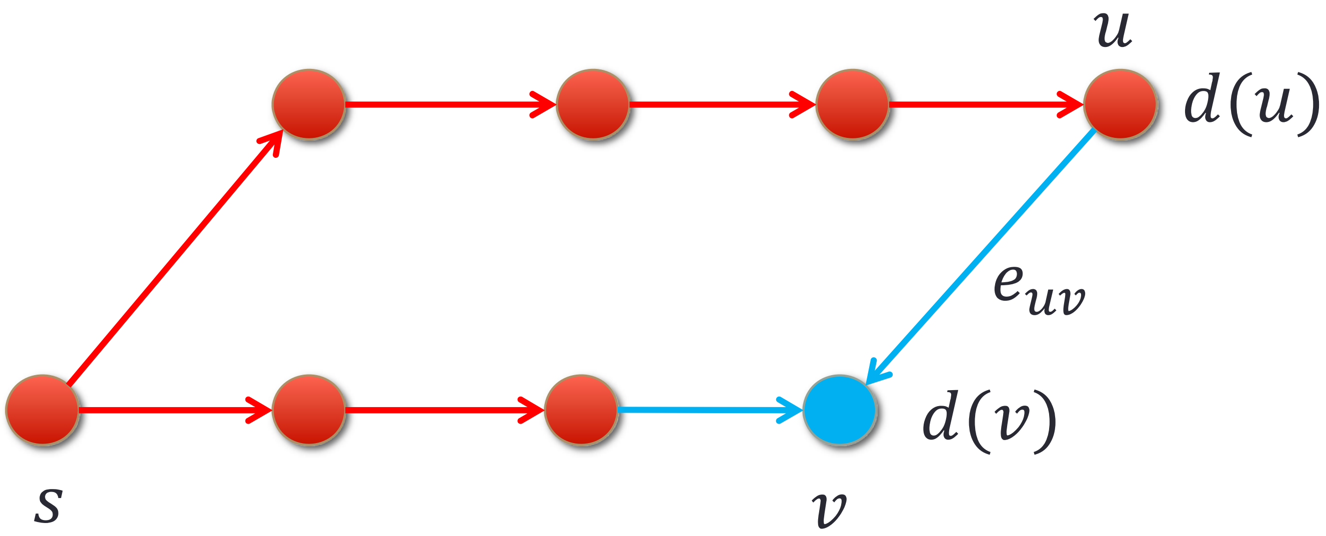


Procedure $relax(e = (u, v) : Edge)$

  if $d[u] + c(e) < d[v]$ **then** $d[v] := d[u] + c(e); \quad parent[v] := u$

Dijkstra's Algorithm: Edge Relaxation

We may to update a previous path from s to v if we find a shorter path.



Procedure $relax(e = (u, v) : Edge)$

  if $d[u] + c(e) < d[v]$ **then** $d[v] := d[u] + c(e); \quad parent[v] := u$

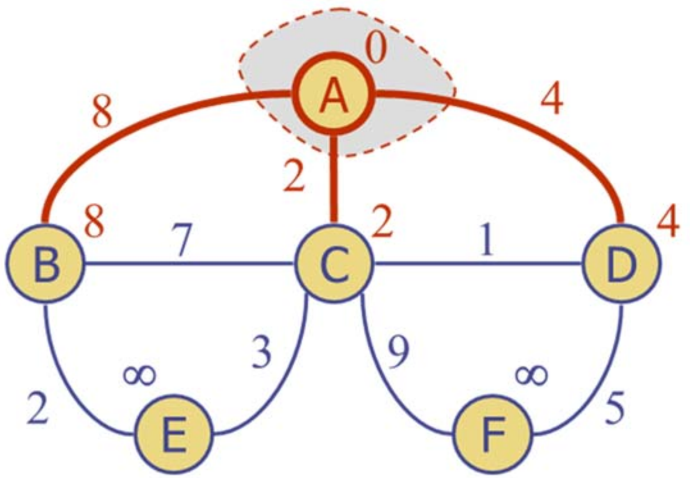
Dijkstra's Algorithm: Sketch

Dijkstra's Algorithm
declare all nodes unscanned
while there is an unscanned node with tentative distance $< +\infty$ **do**

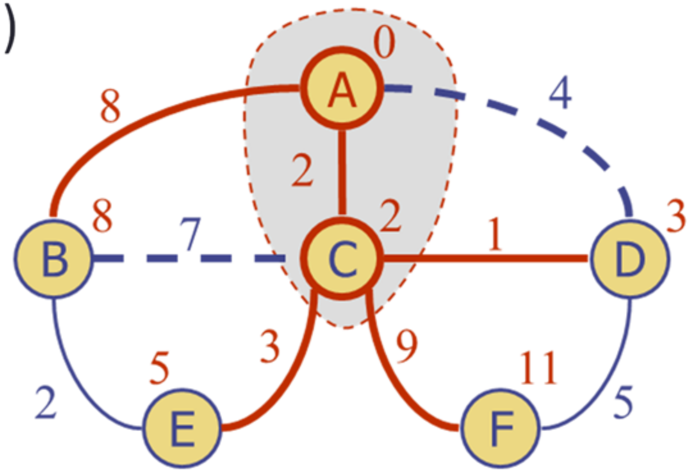
  $u :=$ the unscanned node with minimal tentative distance
  relax all edges (u, v) out of u and declare u scanned

Dijkstra’s Algorithm: Solution Construction

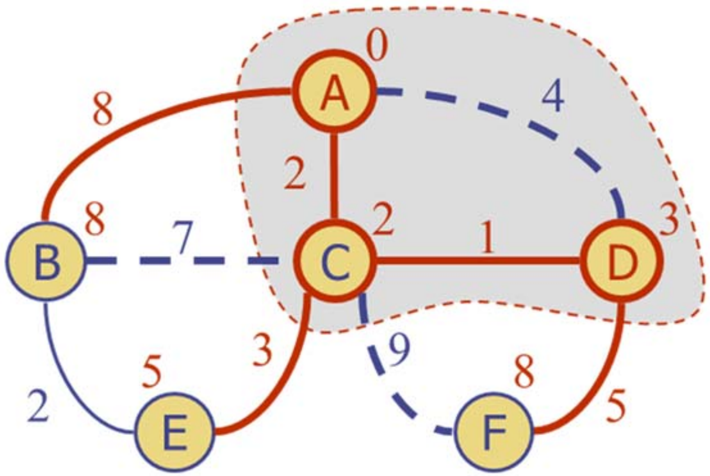
1)



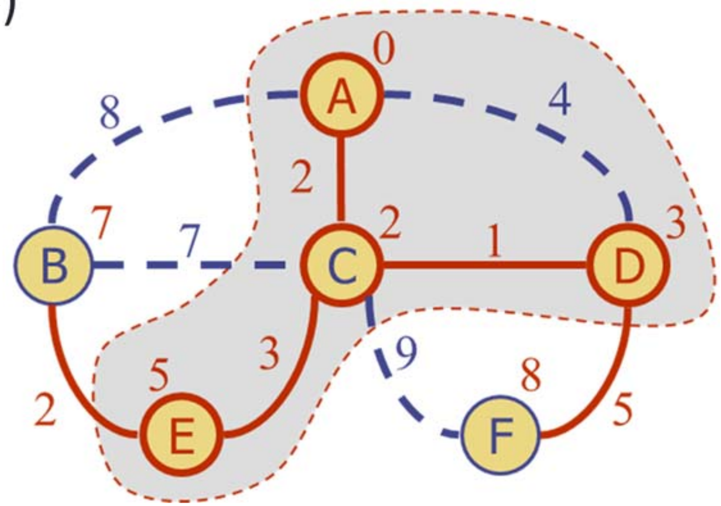
2)



3)

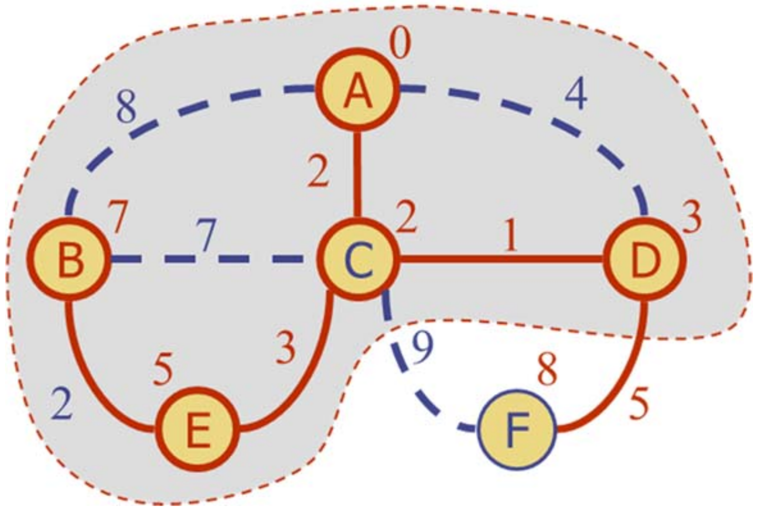


4)

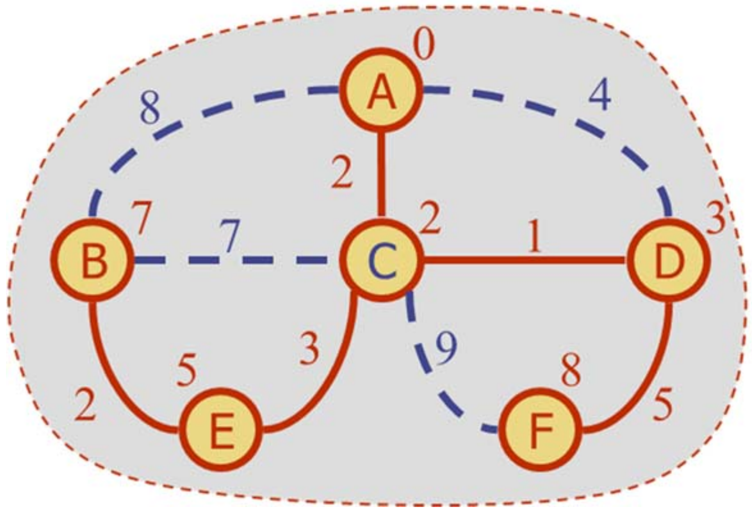


Dijkstra's Algorithm: Solution Construction

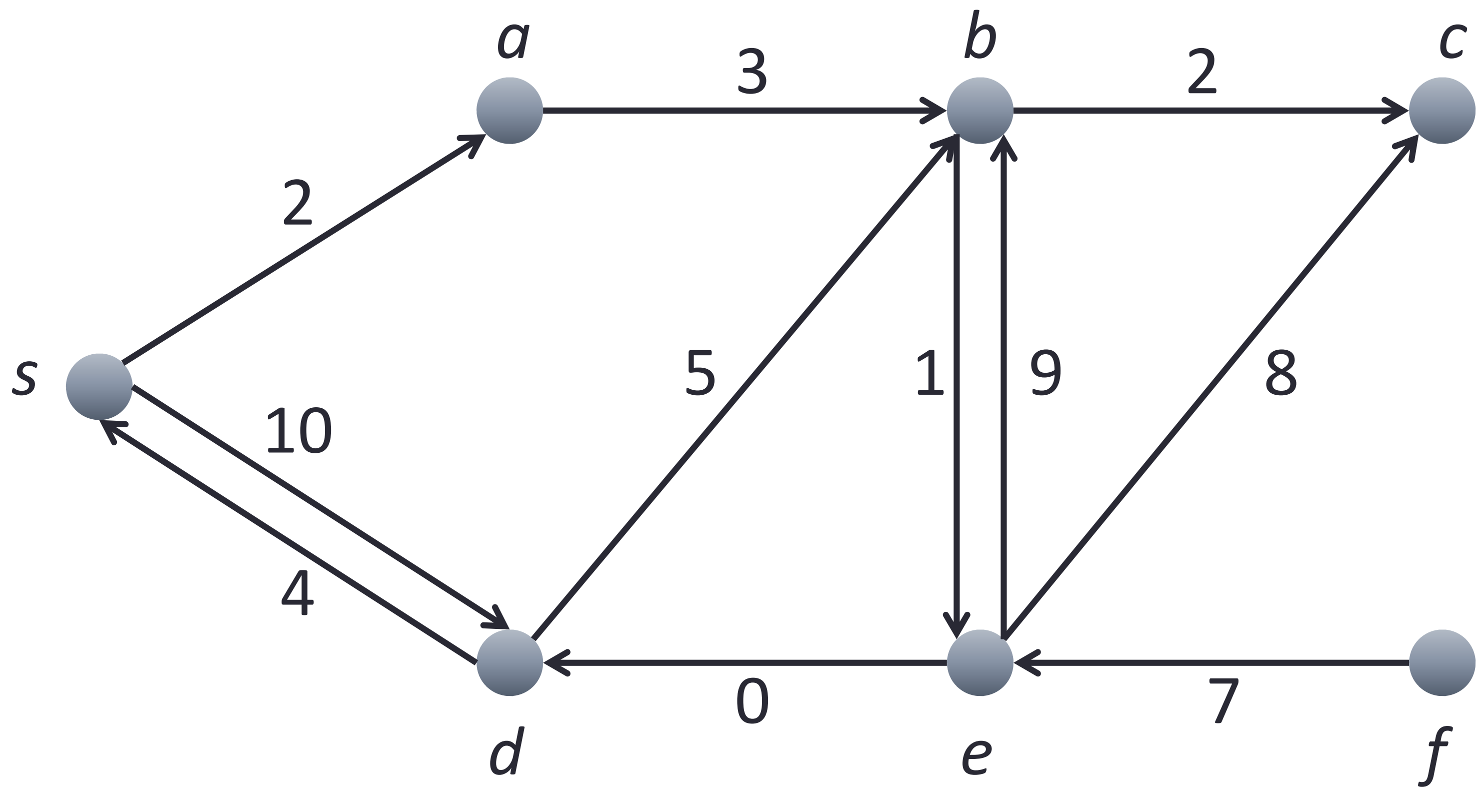
5)



6)



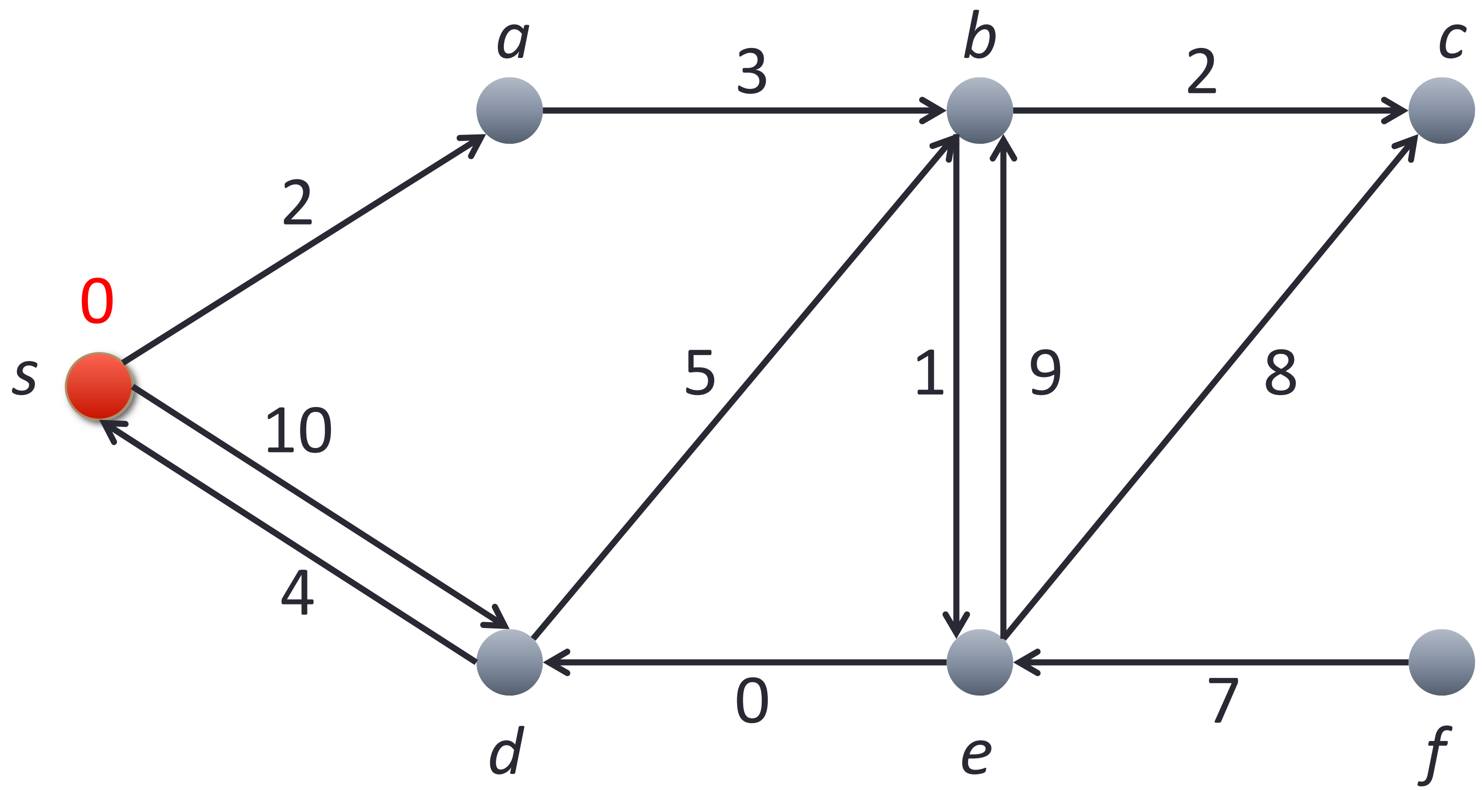
Dijkstra's Algorithm: Example



Nodes for which a shortest path has been computed

Nodes that have already been reached

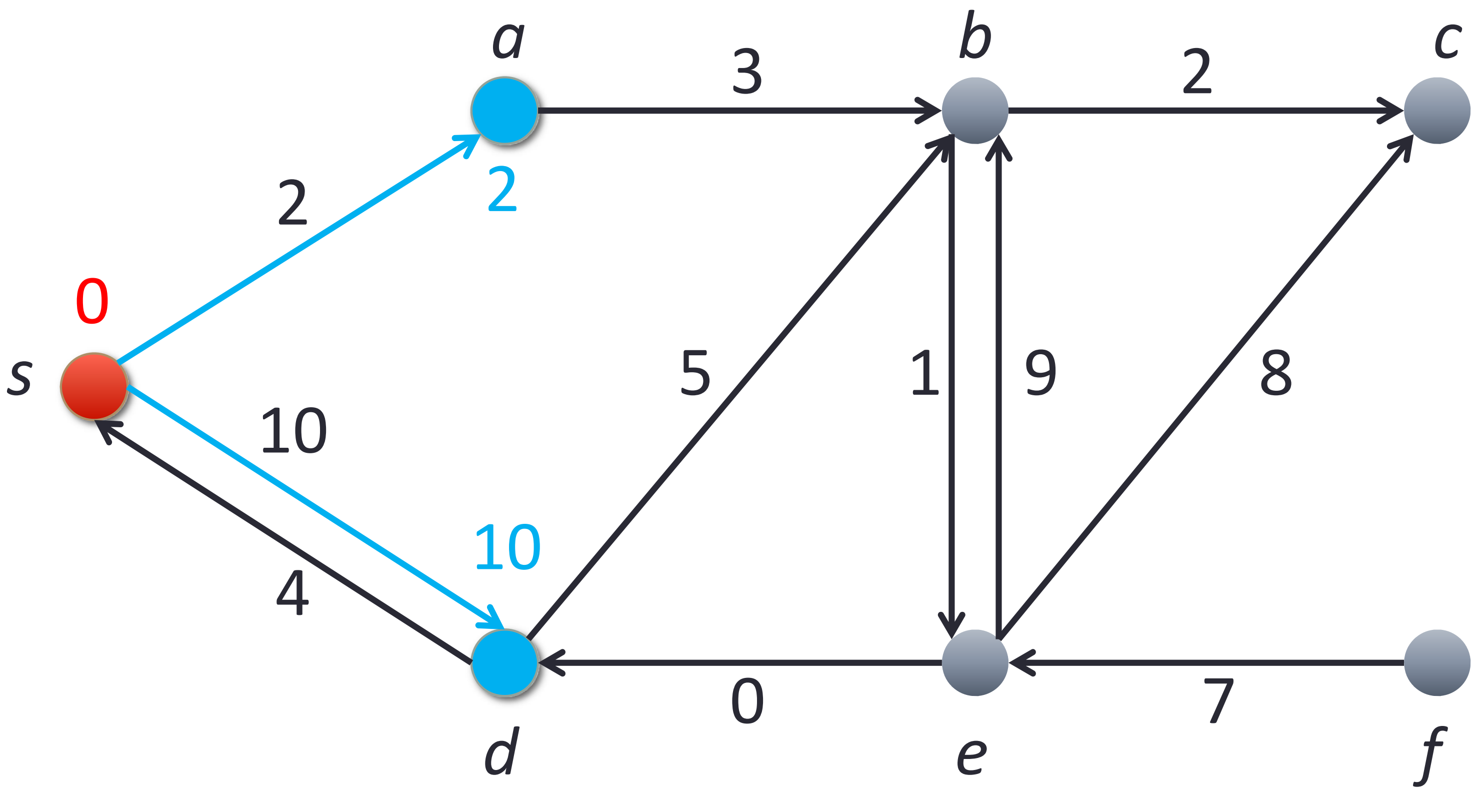
Dijkstra's Algorithm: Example



Nodes for which a shortest path has been computed

Nodes that have already been reached

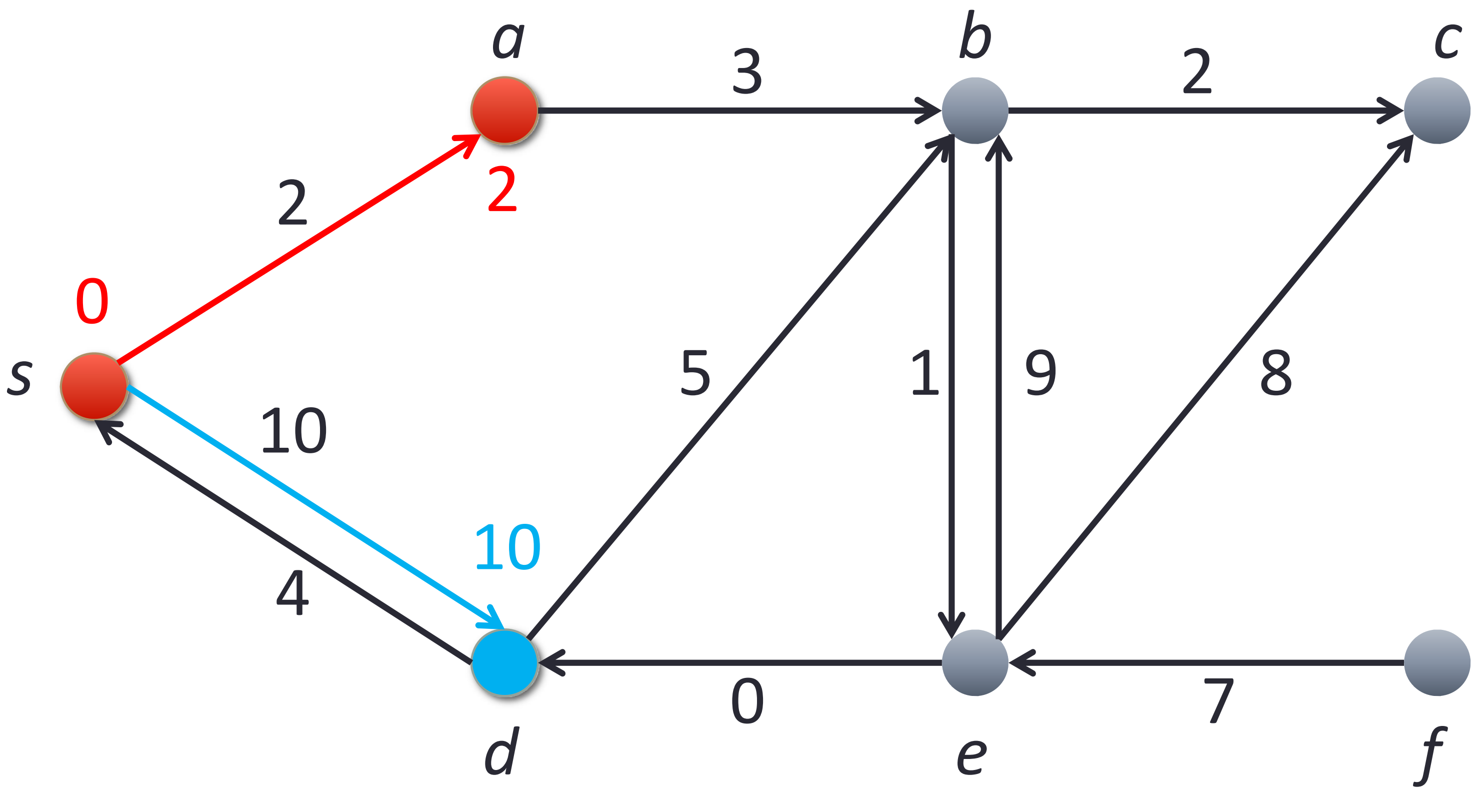
Dijkstra's Algorithm: Example



Nodes for which a shortest path has been computed

Nodes that have already been reached

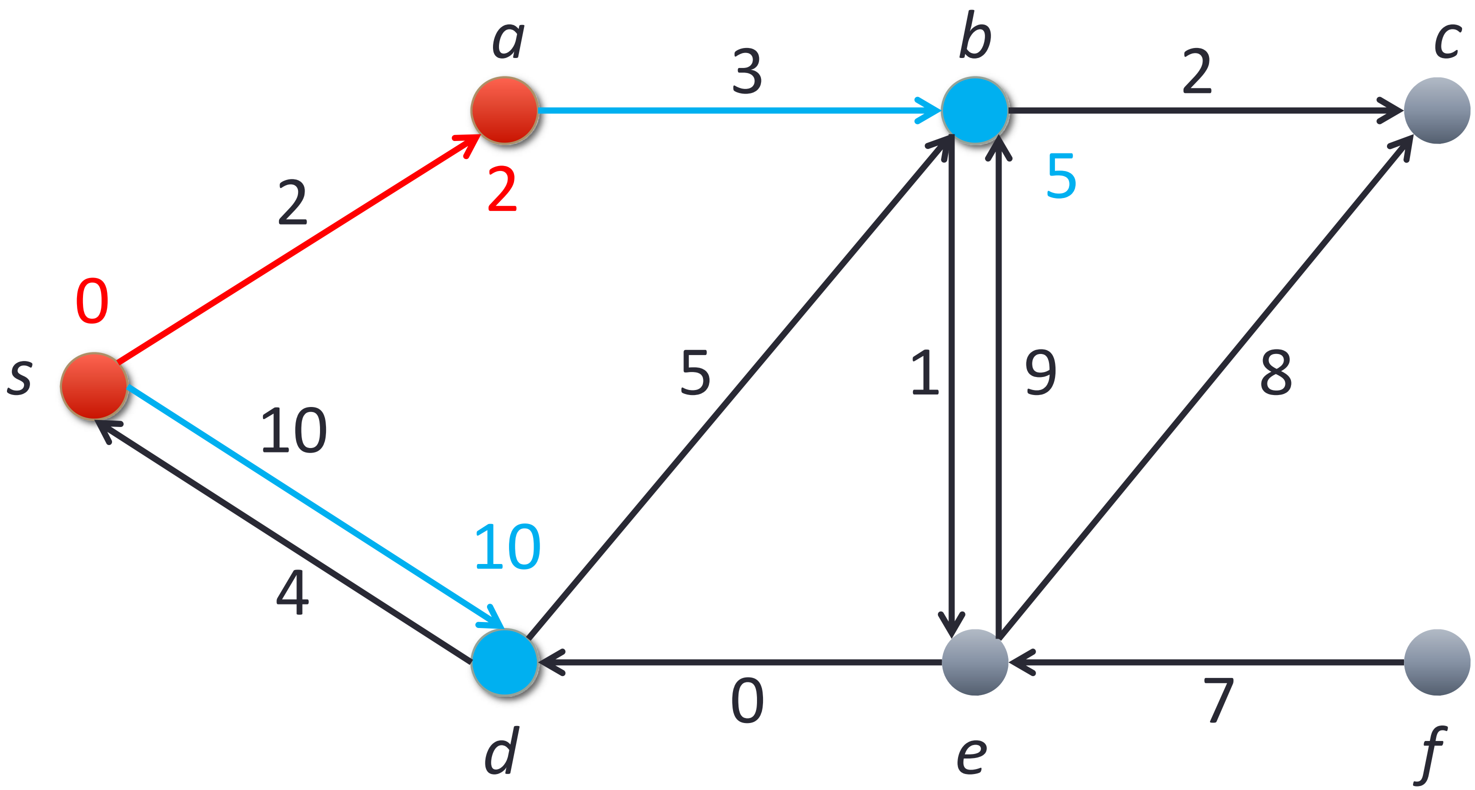
Dijkstra's Algorithm: Example



Nodes for which a shortest path has been computed

Nodes that have already been reached

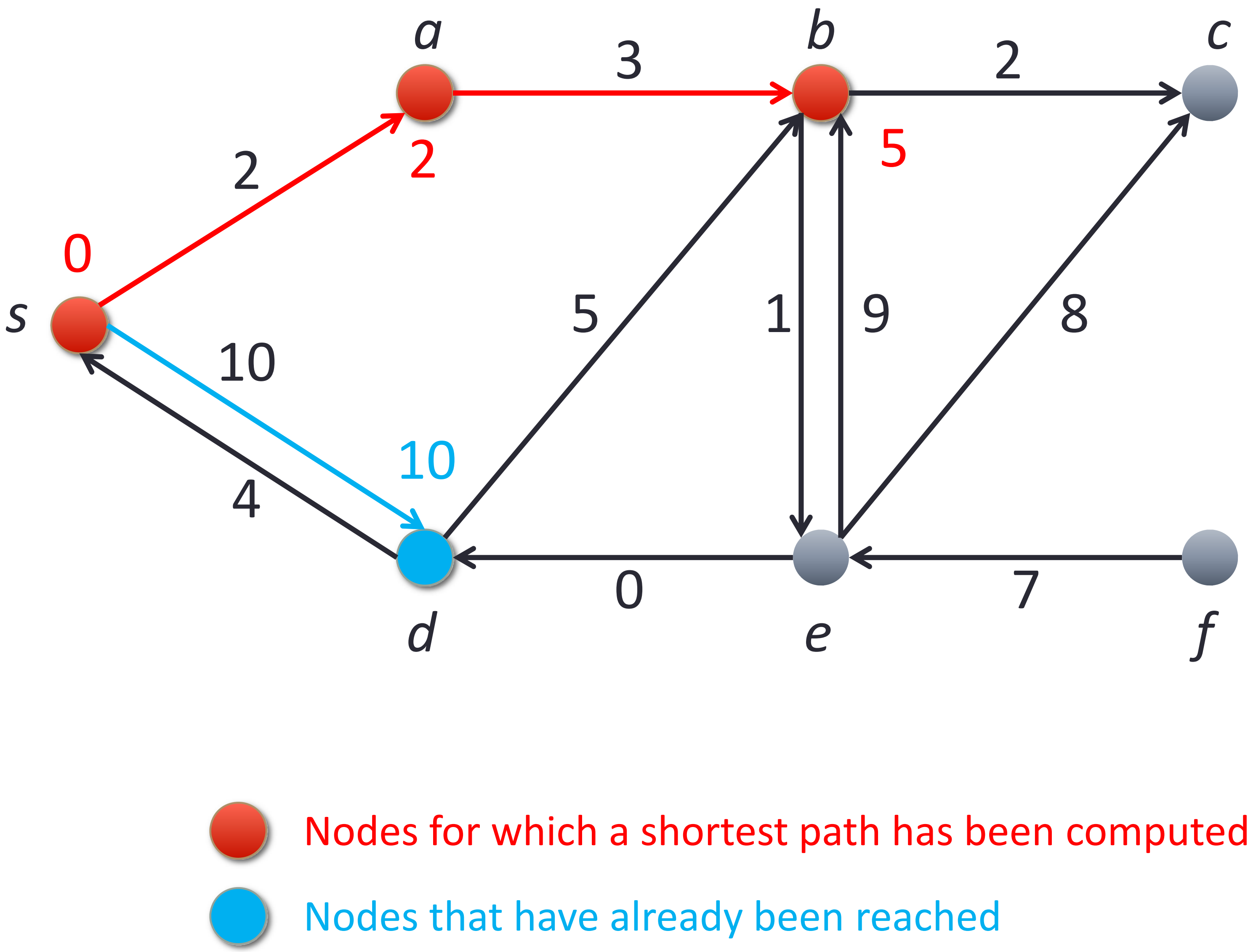
Dijkstra's Algorithm: Example



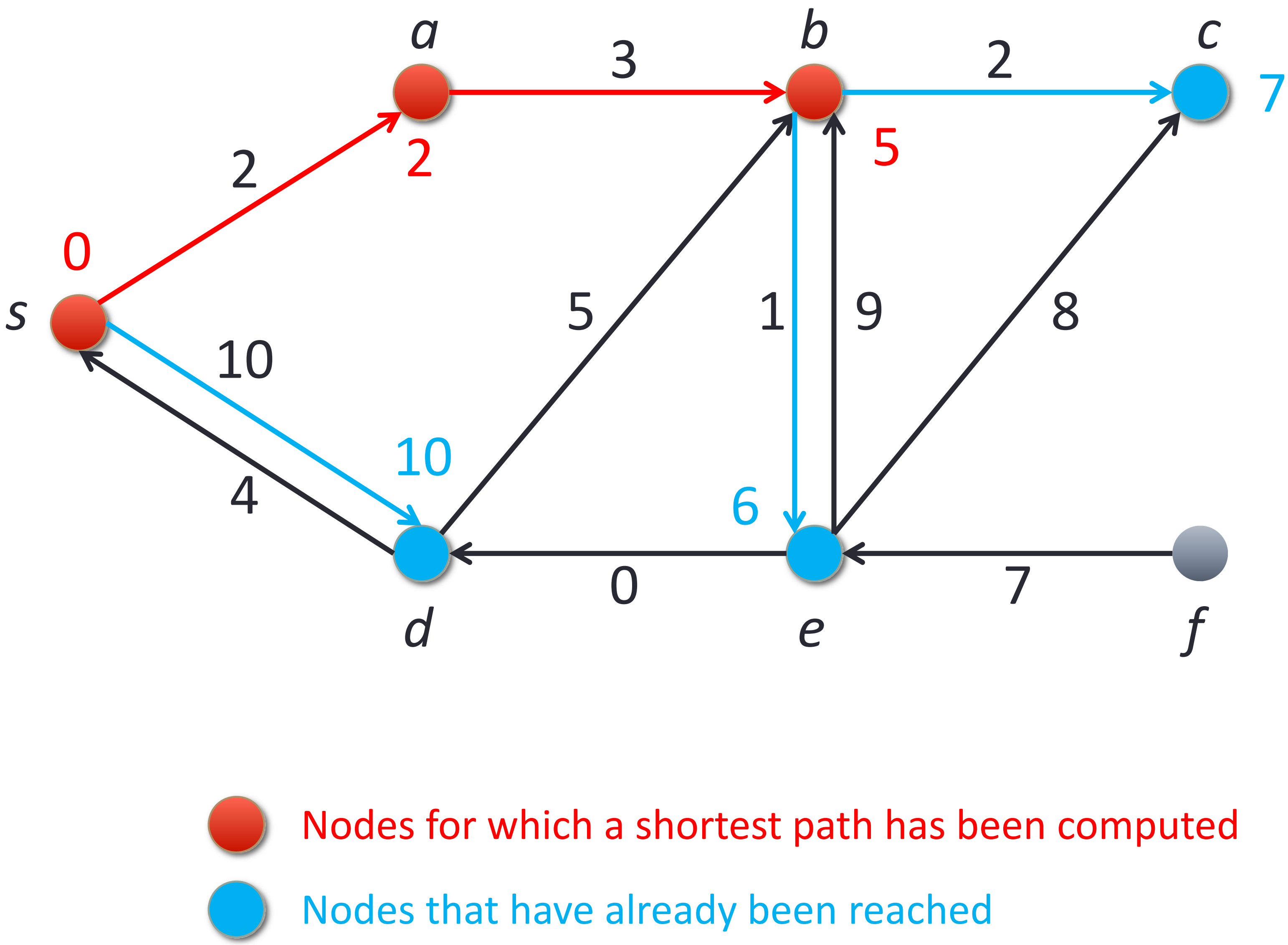
Nodes for which a shortest path has been computed

Nodes that have already been reached

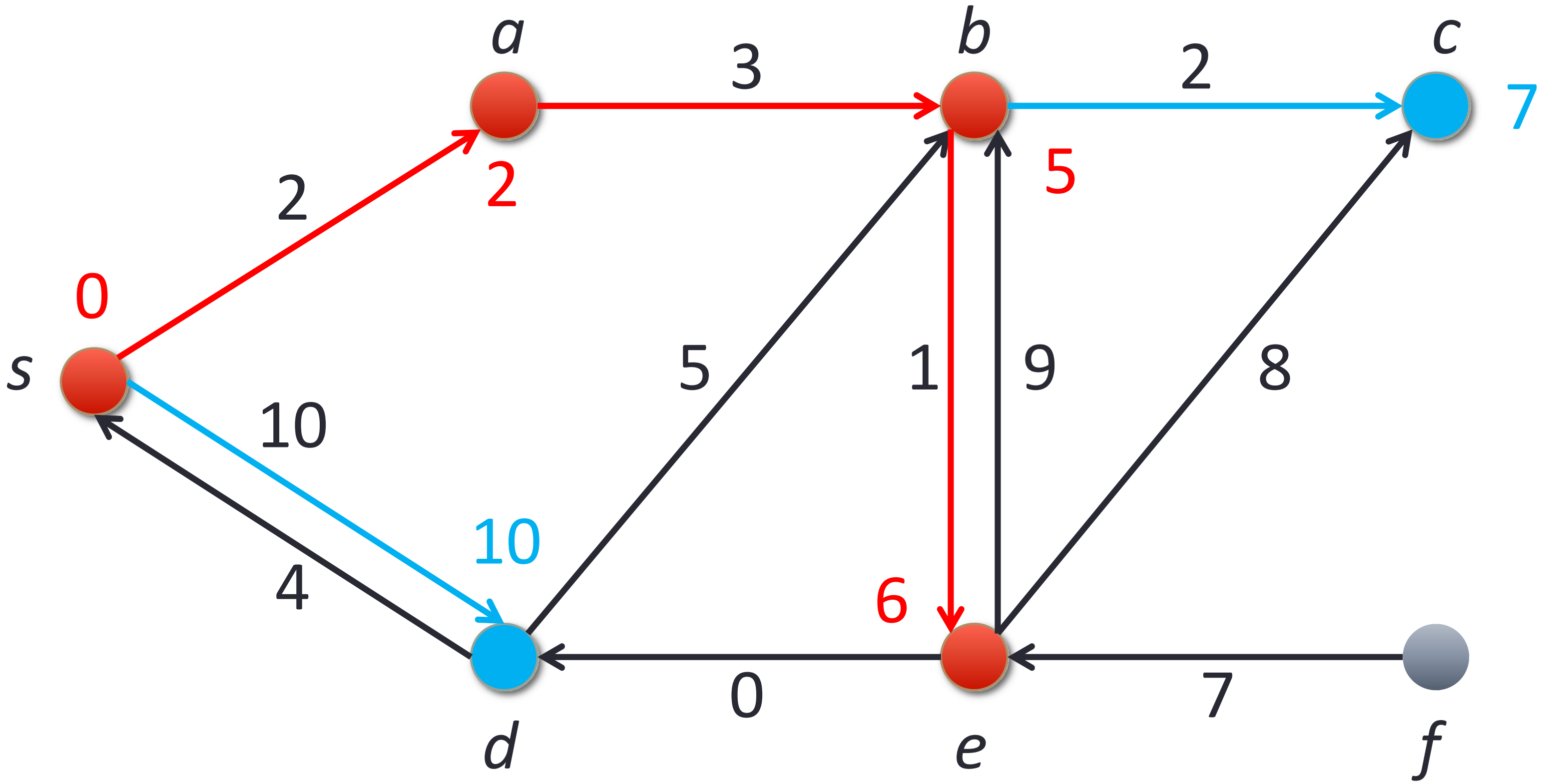
Dijkstra’s Algorithm: Example

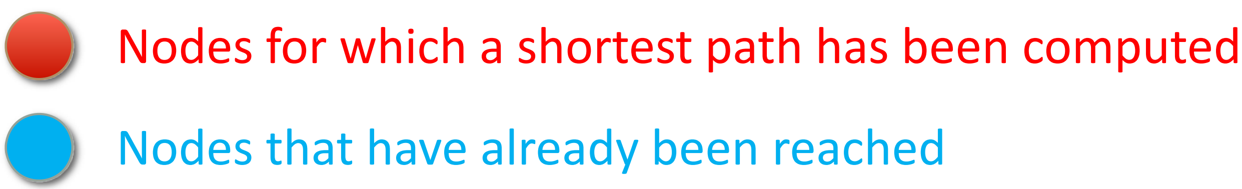


Dijkstra's Algorithm: Example

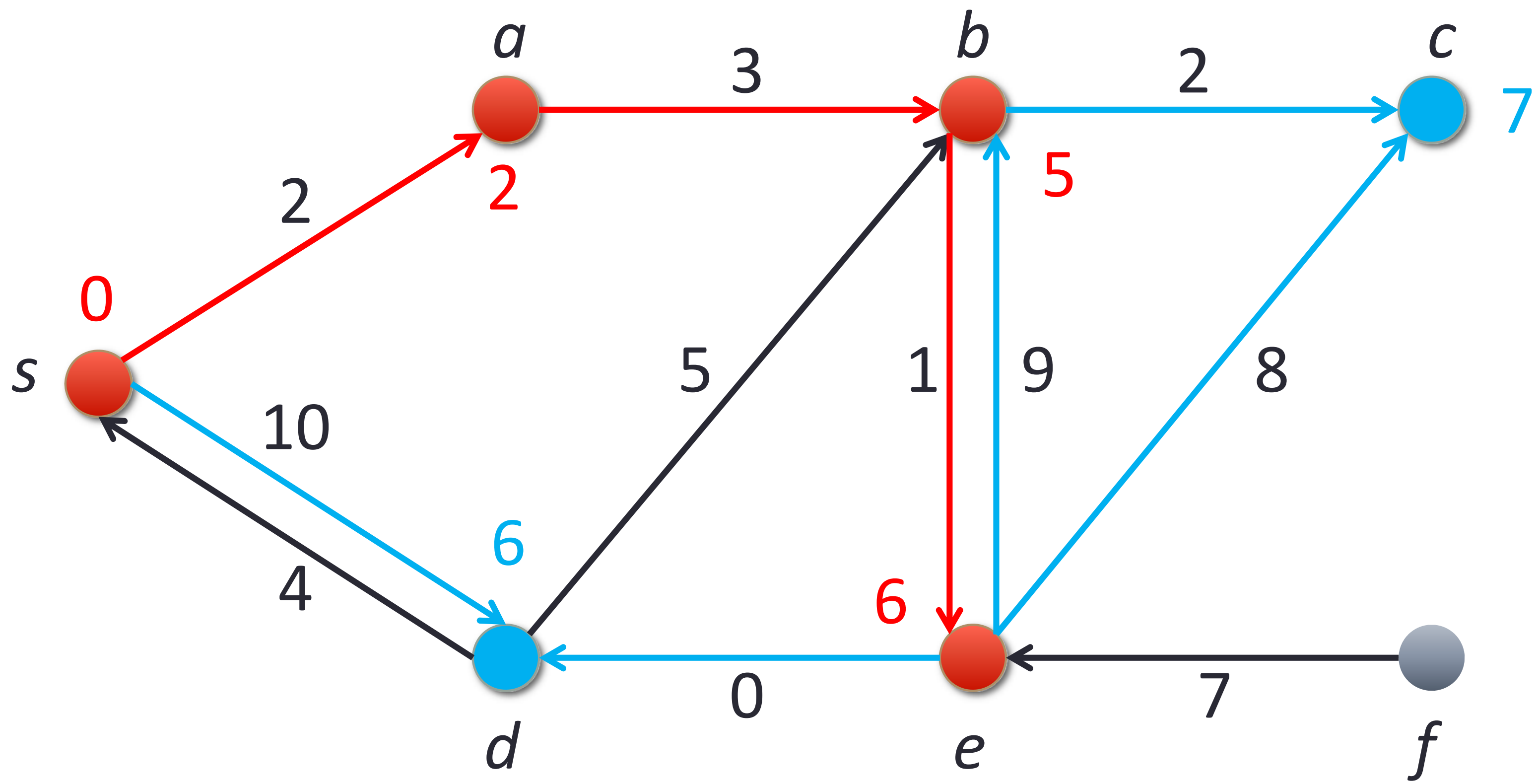


Dijkstra's Algorithm: Example





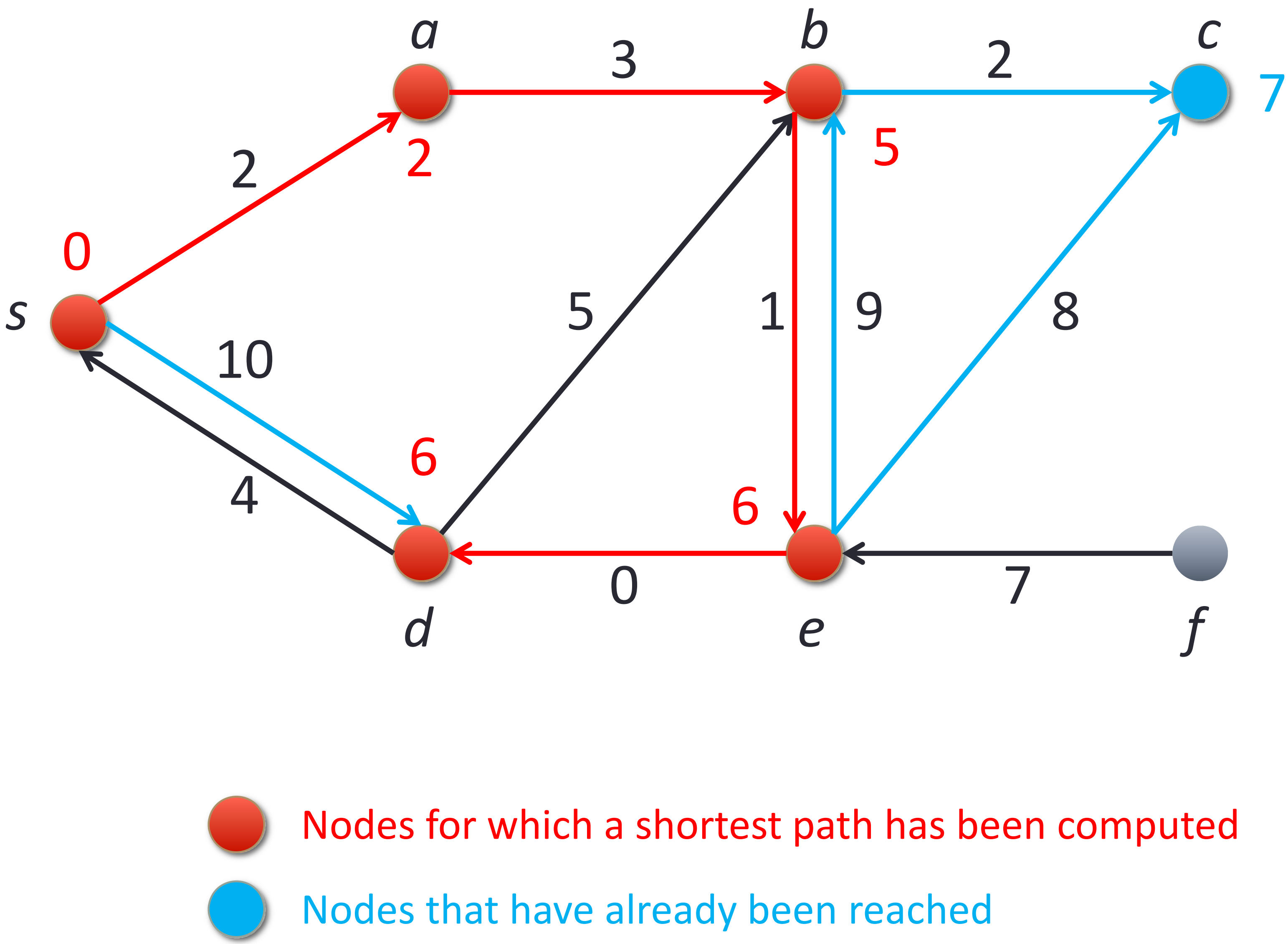
Dijkstra's Algorithm: Example



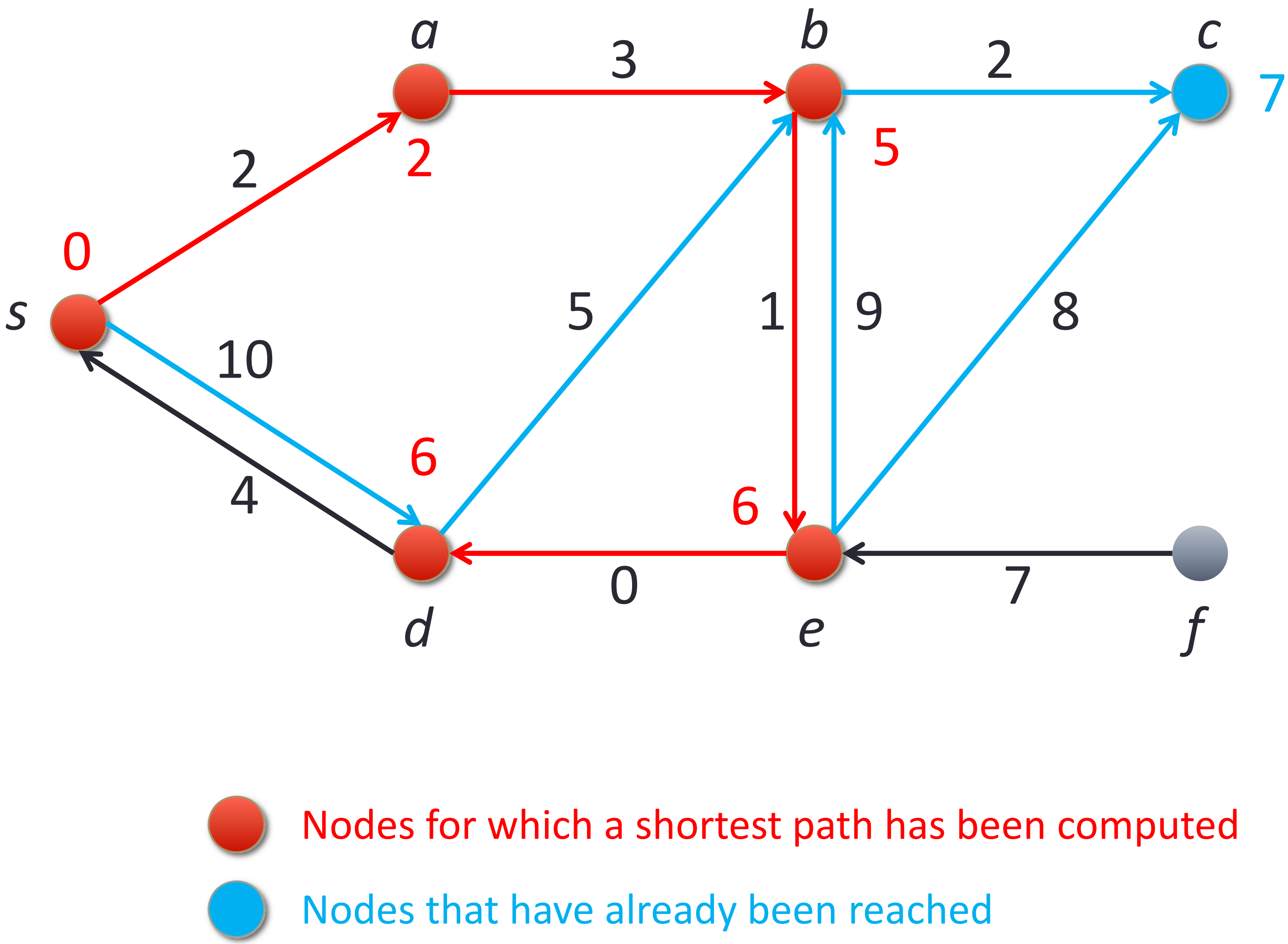
Nodes for which a shortest path has been computed

Nodes that have already been reached

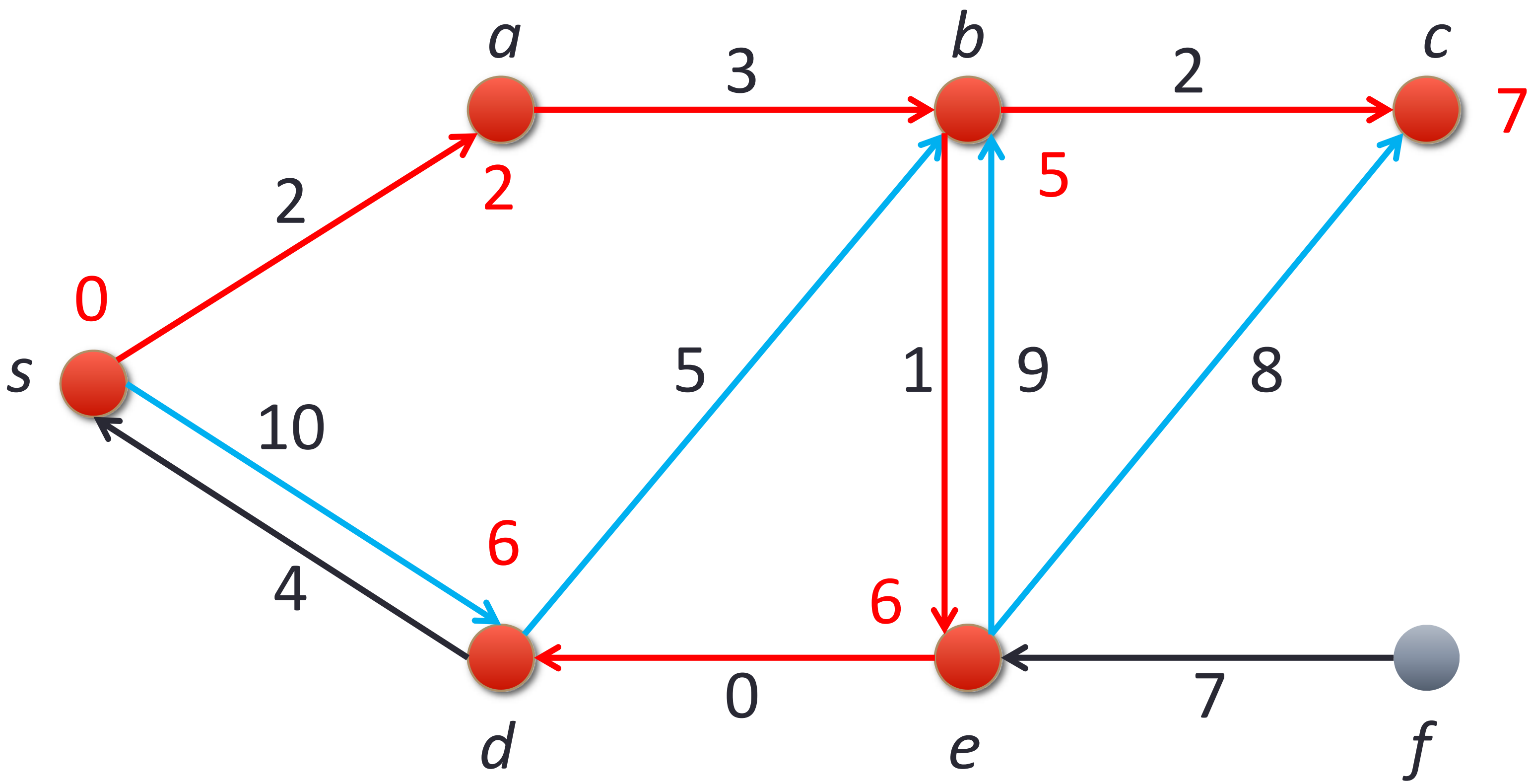
Dijkstra's Algorithm: Example

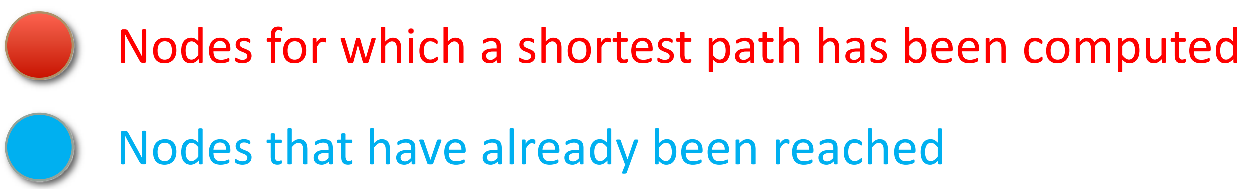


Dijkstra's Algorithm: Example

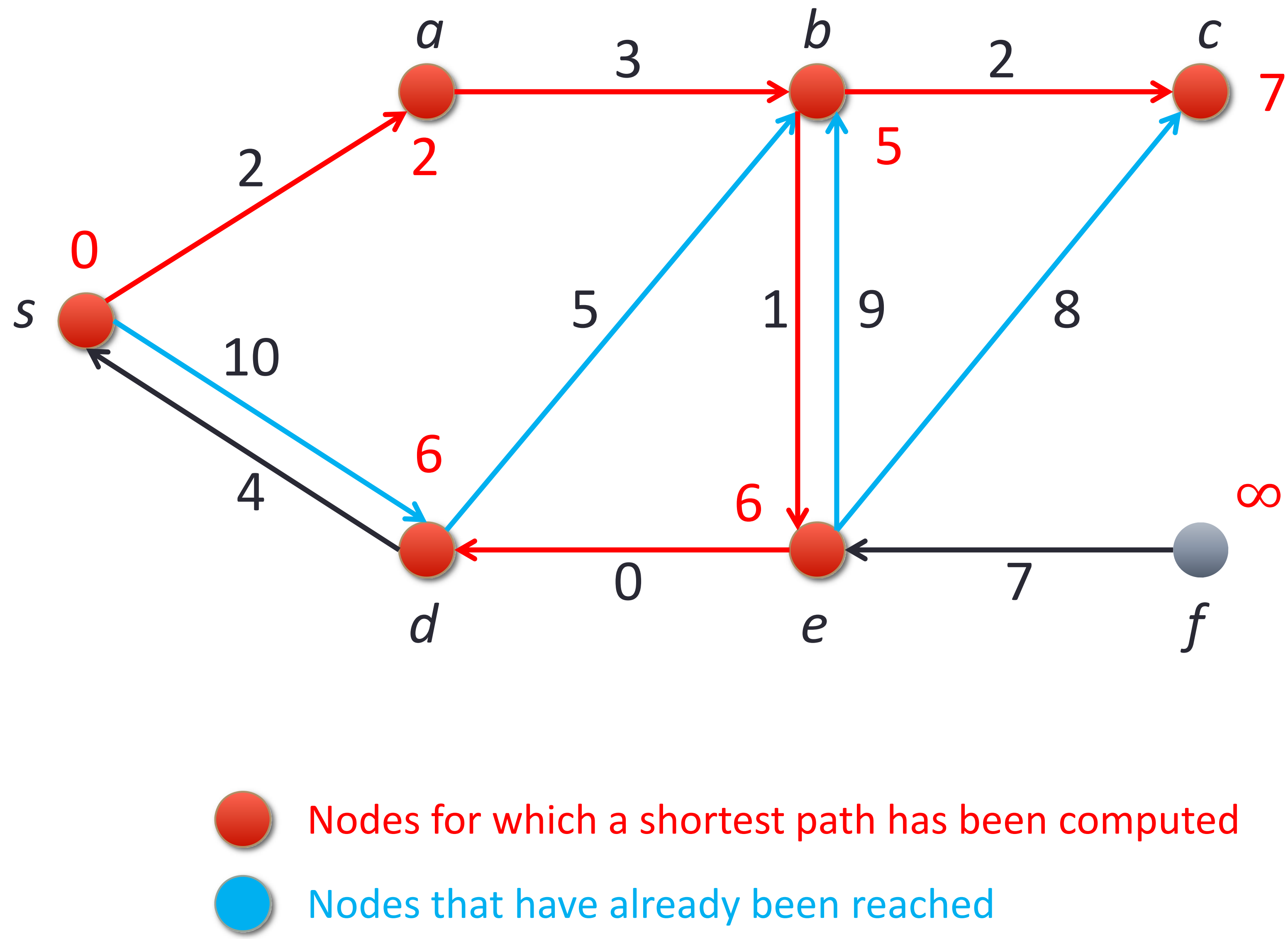


Dijkstra's Algorithm: Example



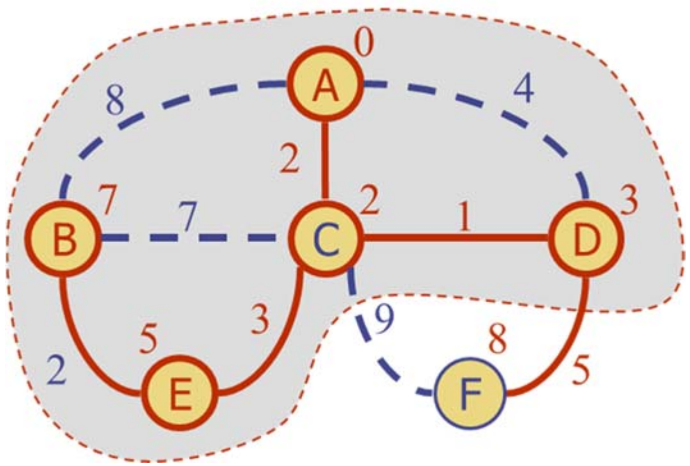


Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Correctness

- Dijkstra's algorithm is based on the greedy method.
  It adds vertices by increasing distance.
- Suppose it did not find all shortest distances.
  Let F be the first wrong vertex the algorithm processed.
- When the previous node, D, on the true shortest path was considered, its distance was correct.
- But the edge (D, F) was relaxed at that time!
- Thus, so long as $\mathrm{d}(F) > d(D)$, F's distance cannot be wrong.
  That is, there is no wrong vertex.



Dijkstra's Algorithm: Correctness

Theorem: Dijkstra's algorithm solves the single-source shortest path problem for graphs with nonnegative edge costs.

Proof:

We show two steps:

- All nodes reachable from s are scanned after termination.
- When a node v becomes scanned then the shortest path from s to v is obtained.

Dijkstra's Algorithm: Correctness

Claim: All nodes reachable from s are scanned after termination.

Proof (by contradiction):

- Assume that there is a node v reachable from s, but never scanned.
- Consider a shortest path $p = (s = v_1, v_2, \dots, v_k = v)$ from s to v.
- Let $i > 1$ be minimal such that v_i is unscanned.
- Implies node v_{i-1} has been scanned.
- When v_{i-1} is scanned $d(v_i)$ is set to $d(v_{i-1}) + c(v_{i-1}, v_i) < \infty$.
- Hence, v_i must be scanned as only nodes u with $d(i) = \infty$ stay unscanned. Contraction to v_i is unscanned.

$\square$

Dijkstra's Algorithm: Correctness

Claim: When a node v becomes scanned then the shortest path from s to v is obtained.

Proof (by contradiction):

- Denote by $\mu(v)$ the length of a shortest path from s to v.
- Consider the first point in time t when v has been scanned and $d(v) > \mu(v)$ holds.
- Consider a shortest path $p = (s = v_1, v_2, \dots, v_k = v)$ from s to v.
- Let $i > 1$ be minimal such that v_i has not been scanned before time t.

$\square$

Dijkstra's Algorithm: Correctness

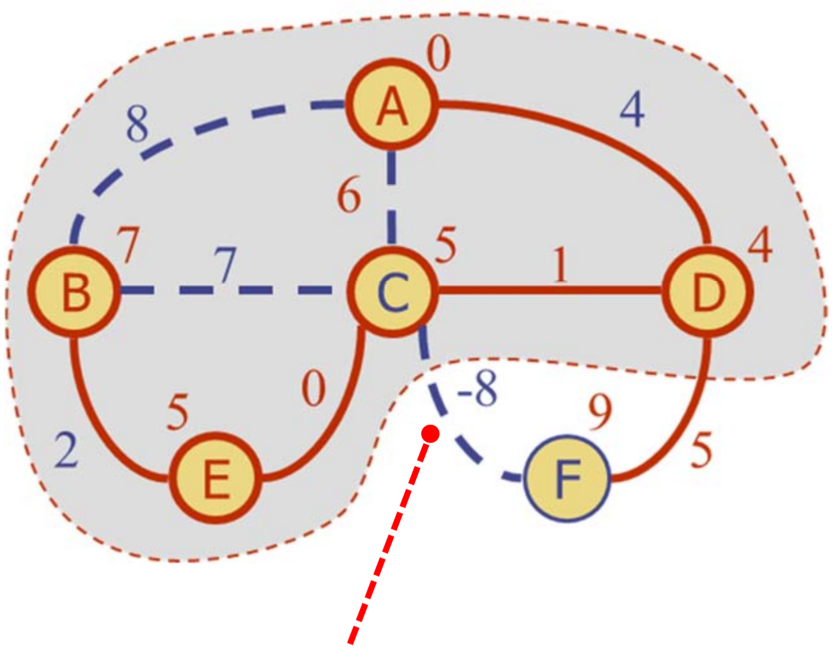
Proof (continued):

- Node v_{i-1} was scanned before time t which implies $\mu(v_{i-1}) = d(v_{i-1})$.
- When v_{i-1} is scanned $d(v_i)$ is set to $d(v_{i-1}) + c(v_{i-1}, v_i) = \mu(v_{i-1}) + c(v_{i-1}, v_i)$.
- We have $d(v_i) = \mu(v_i) \leq \mu(v_k) < d(v_k)$ and hence v_i is scanned instead of v_k, a contradiction.

$\square$

Dijkstra's Algorithm: Negative Weights

- Dijkstra's algorithm is based on the greedy method.
  It adds vertices by increasing distance.
- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with $d(C) = 5$.

Dijkstra's Algorithm: Implementation

Store all unscanned reached nodes in an addressable priority queue Q (using tentative distances as key values)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - Insert(k, e) returns a locator
  - DecreaseKey(l,k) changes the key of an item
- We store two labels with each vertex:
  - Distance ($d(v)$ label)
  - Locator l in priority queue

Dijkstra's Algorithm: Pseudocode

Function $\mathit{Dijkstra}(s : \mathit{NodeId}) : \mathit{NodeArray} \times \mathit{NodeArray}$ // returns (d, parent)

  $d = \langle \infty, \ldots, \infty \rangle \; : \mathit{NodeArray}$ **of** $\mathbb{R} \cup \{\infty\}$ // tentative distance from root

  $\mathit{parent} = \langle \perp, \ldots, \perp \rangle \; : \mathit{NodeArray}$ **of** NodeId

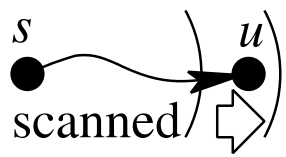
  $\mathit{parent}[s] := s$ // self-loop signals root

  $Q : \mathit{NodePQ}$ // unscanned reached nodes

  $d[s] := 0; \quad Q.\mathit{insert}(s)$

  while $Q \neq \emptyset$ **do**

    $u := Q.\mathit{deleteMin}$ // we have $d[u] = \mu(u)$



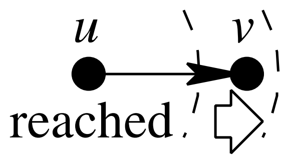
    foreach $\mathit{edge}\; e = (u, v) \in E$ **do**

      if $d[u] + c(e) < d[v]$ **then** // relax

        $d[v] := d[u] + c(e)$

        $\mathit{parent}[v] := u$ // update tree

        if $v \in Q$ **then** $Q.\mathit{decreaseKey}(v)$



        else $Q.\mathit{insert}(v)$

  return (d, parent)

Dijkstra's Algorithm: Runtime Complexity

- Initialization (arrays, priority queue) takes time $O(n)$.
- Every reachable node is inserted and removed once from Q.
- At most n DeleteMin and insert operations.
- Each node is scanned at most once and each edge is relaxed at most once.
- Implies at most m DecreaseKey operations.

Total runtime

$$T_{\text{Dijkstra}} = \mathrm{O}\big(m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n))\big)$$

Dijkstra's Algorithm: Runtime Complexity

Runtime depends on implementation of priority queue.

Original (Dijkstra 1959):

- Maintain the number of reached unscanned nodes.
- An array d storing the distances and an array storing for each node whether it is reached or unscanned.
- Insert and DecreaseKey take time $O(1)$
- DeleteMin takes time $O(n)$
- Total Runtime: $O(m + n^2)$

Improvements:

- Binary Heaps: $O((m + n) \log n)$
- Fibonacci Heaps: $O(m + n \log n)$

Shortest Path: Properties

Property 1: There is a tree of shortest paths from a start vertex to all the other vertices.

Property 2: A subpath of a shortest path is itself a shortest path.

Proof (by contradiction):

Assume that the path p is a shortest path from s to v.

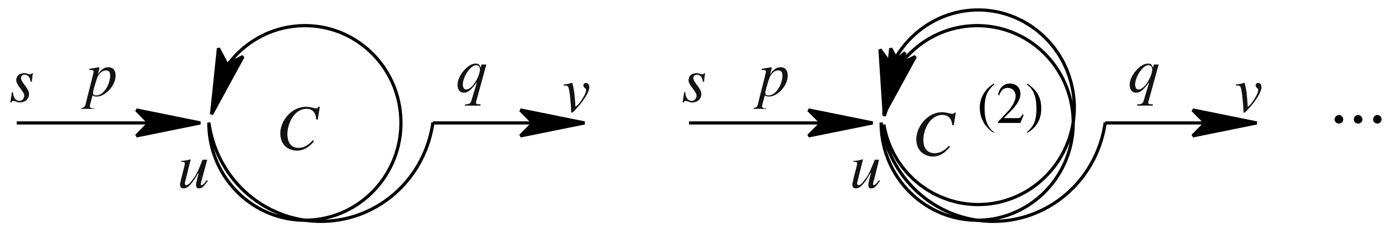
Assume that a subpath from a to b is not a shortest path from a to b

This implies that there is a shorter path from a to b

We can use this path to obtain a shorter path from s to v.

Contradiction to p is shortest path from s to v. $\square$

Shortest Path: Negative Cycles



If a path from s to v contains a negative cycles then a shortest path does not exist (is not defined).

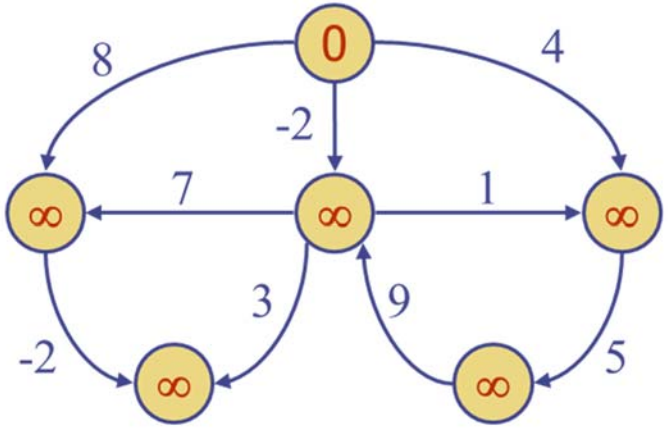
Bellman-Ford Algorithm

- Dijkstra's algorithm works for acyclic graphs and for non-negative edge costs.

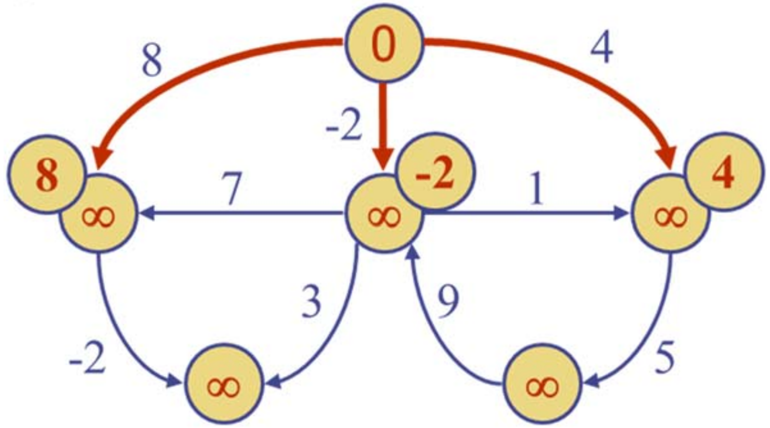
- Bellman-Ford algorithm solves the problem for arbitrary edge costs.
- It uses $n - 1$ rounds and relaxes in each round all edges.
- This works as simple paths have at most $n - 1$ edges.
- After the relaxations are complete, we have all shortest paths to nodes with non-negative cycles.
- We still need to identify the nodes that can be reached by using negative cycles.

Bellman-Ford Algorithm: Solution Construction

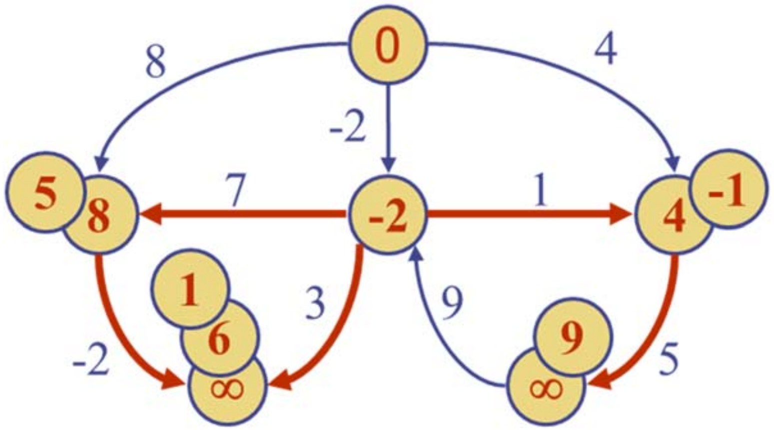
1)



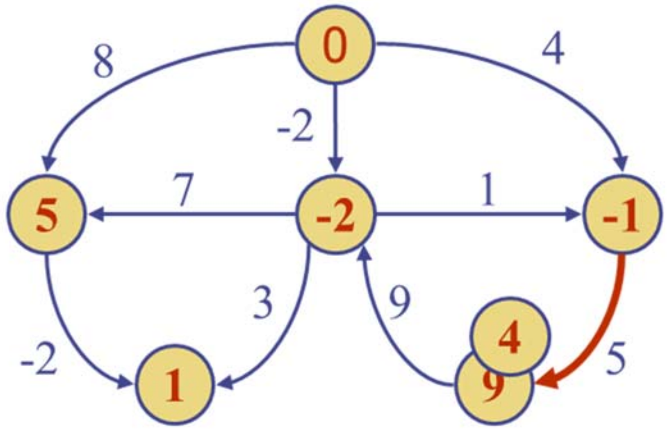
2)



3)



4)



Nodes are labeled with their d(v) values

Bellman-Ford Algorithm: Negative Weights

- Assume that there is an edge $e = (u, v)$ that allows to improve $d(v)$ after the relaxations are complete.
- Then the node v is reachable by using a negative cycle.
- Furthermore, all nodes reachable from v can also be reached by using a negative cycle.
- We set $d(v) = -\infty$ for these nodes v.
- We use postprocessing and the routine **Infect** to find nodes reachable by negative cycles.

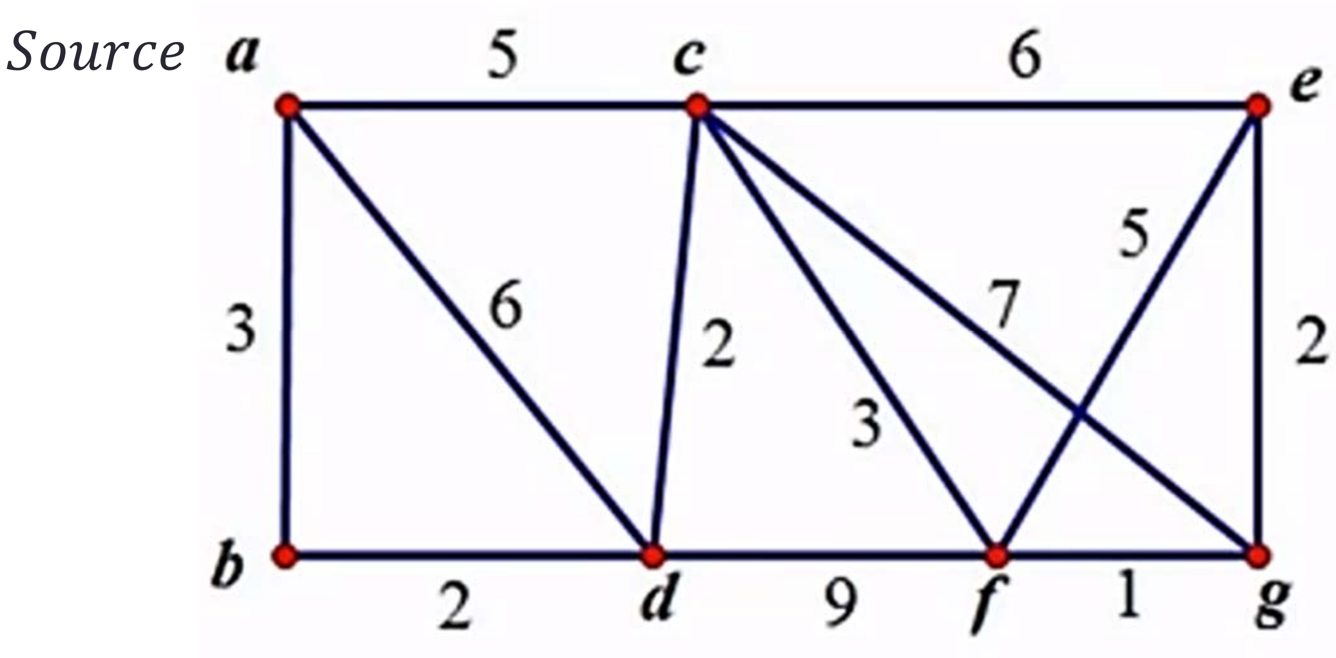
Bellman-Ford Algorithm: Pseudocode

Function $BellmanFord(s : NodeId) : NodeArray \times NodeArray$
    $d = \langle \infty, \ldots, \infty \rangle$: $NodeArray$ **of** $\mathbb{R} \cup \{-\infty, \infty\}$ // distance from root
    $parent = \langle \perp, \ldots, \perp \rangle$: $NodeArray$ **of** $NodeId$
    $d[s] := 0;\quad parent[s] := s$ // self-loop signals root
    for $i := 1$ **to** $n-1$ **do**
        forall $e \in E$ **do** $relax(e)$ // round i
    forall $e = (u, v) \in E$ **do** // postprocessing
        if $d[u] + c(e) < d[v]$ **then** $infect(v)$
    return $(d, parent)$

Procedure $infect(v)$
    if $d[v] > -\infty$ **then**
        $d[v] := -\infty$
        foreach $(v, w) \in E$ **do** $infect(w)$

Running time: $O(nm)$.

Shortest Path: Exercise



Other references and things to do

- Read chapter 14.6 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.