Shortest Path Algorithms

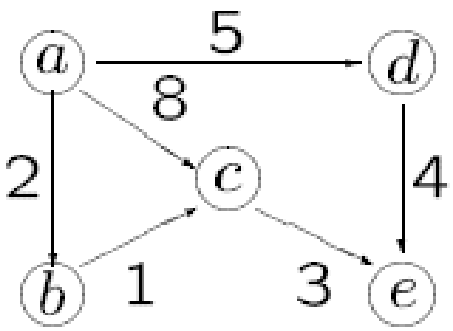
Shortest Path for Non-Weighted Graph

Let $G = (V, E)$ be a (di)graph.

- The shortest path between two vertices is a path with the shortest **length** (least number of edges). Call this the **link-distance**.
- Breadth-first-search is an algorithm for finding shortest (link-distance) paths from a single source vertex to all other vertices.
- BFS processes vertices in increasing order of their distance from the root vertex.
- BFS has running time $O(|V| + |E|)$.

Shortest Path for Weighted Graph

Let $G = (V, E)$ be a weighted digraph, with weight function $w : E \mapsto \mathbf{R}$ mapping edges to real-valued weights. If $e = (u, v)$, we write $w(u, v)$ for $w(e)$.



- The **length** of a path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the weights of its constituent edges:

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

$\text{length}(\langle a, b, c, e \rangle) = 6$
distance from a to e is 6

- The **distance** from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v; and is ∞ otherwise.

Single-Source Shortest-Paths Problem

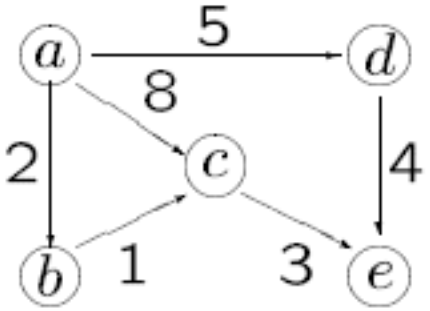
The Problem: Given a digraph with non-negative edge weights $G = (V, E)$
and a distinguished *source vertex*, $s \in V$,
determine the distance and a shortest path from the source vertex to every vertex in the digraph.

Question: How do you design an efficient algorithm for this problem?

Single-Source Shortest-Paths Problem

Important Observation: Any subpath of a shortest path must also be a shortest path. Why?

Example: In the following digraph, $\langle a, b, c, e \rangle$ is a shortest path. The subpath $\langle a, b, c \rangle$ is also a shortest path.



length($\langle a, b, c, e \rangle$) = 6
distance from a to e is 6

The Rough Idea of Dijkstra's Algorithm

- Maintain an *estimate* $d[v]$ of the length $\delta(s, v)$ of the shortest path for each vertex v.

- Always $d[v] \geq \delta(s, v)$ and $d[v]$ equals the length of a known path
  ($d[v] = \infty$ if we have no paths so far).

- Initially $d[s] = 0$ and all the other $d[v]$ values are set to ∞. The algorithm will then process the vertices one by one in some order.
  The processed vertex's estimate will be validated as being real shortest distance, i.e. $d[v] = \delta(s, v)$.

Here "processing a vertex u" means finding new paths and updating $d[v]$ for all $v \in Adj[u]$ if necessary. The process by which an estimate is updated is called **relaxation**.

When all vertices have been processed, $d[v] = \delta(s, v)$ for all v.

Question 1: How does the algorithm find new paths and do the relaxation?

Question 2: In which order does the algorithm process the vertices one by one?

Answer to Question 1

- Finding new paths. When processing a vertex u, the algorithm will examine all vertices $v \in Adj[u]$. For each vertex $v \in Adj[u]$, a new path from s to v is found (path from s to u + new edge).

- Relaxation. If the length of the new path from s to v is shorter than $d[v]$, then update $d[v]$ to the length of this new path.

Remark: Whenever we set $d[v]$ to a finite value, there exists a path of that length. Therefore $d[v] \geq \delta(s, v)$.

(Note: If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.)

Implementing the Idea of Relaxation

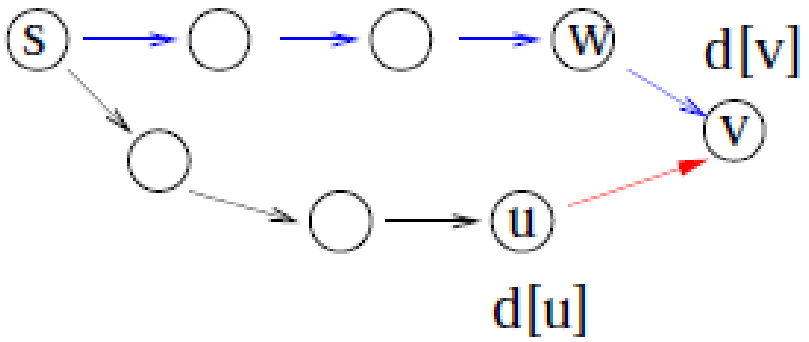
Consider an edge from a vertex u to v whose weight is $w(u, v)$. Suppose that we have already processed u so that we know $d[u] = \delta(s, u)$ and also computed a current estimate for $d[v]$. Then

- There is a (shortest) path from s to u with length $d[u]$.
- There is a path from s to v with length $d[v]$.

Combining this path from s to u with the edge (u, v), we obtain another path from s to v with length $d[u] + w(u, v)$.

If $d[u]+w(u,v) < d[v]$, then we replace the old path $\langle s, \ldots, w, v \rangle$ with the new shorter path $\langle s, \ldots, u, v \rangle$. Hence we update

- $d[v] = d[u] + w(u,v)$
- $pred[v] = u$ (originally, $pred[v] == w$).



The Algorithm for Relaxing an Edge

Relax(u,v)

{

if ($d[u] + w(u, v) < d[v]$)

{ $d[v] = d[u] + w(u, v)$;

$pred[v] = u$;

}

}

Remark: The predecessor pointer $pred[\,]$ is for determining the shortest paths.

Idea of Dijkstra's Algorithm: Repeated Relaxation

- Dijkstra's algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we know the true distance, that is $d[v] = \delta(s, v)$.

- Initially $S = \emptyset$, the empty set, and we set $d[s] = 0$ and $d[v] = \infty$ for all others vertices v. One by one we select vertices from $V \setminus S$ to add to S.

- The set S can be implemented using an array of vertex colors. Initially all vertices are white, and we set $color[v] =$ black to indicate that $v \in S$.

The Selection in Dijkstra's Algorithm

Recall Question 2: What is the best order in which to process vertices, so that the estimates are guaranteed to converge to the true distances.
That is, how does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm. For each vertex in $u \in V \setminus S$, we have computed a distance estimate $d[u]$. The next vertex processed is always a vertex $u \in V \setminus S$ for which $d[u]$ is minimum, that is, we take the unprocessed vertex that is closest (by our estimate) to s.

The Selection in Dijkstra's Algorithm

Question: How do we perform this selection efficiently?

Answer: We store the vertices of $V \setminus S$ in a *priority queue*, where the key value of each vertex v is $d[v]$.

[Note: if we implement the priority queue using a heap, we can perform the operations **Insert(), Extract_Min()**, and **Decrease_Key()**, each in $O(\log n)$ time.]

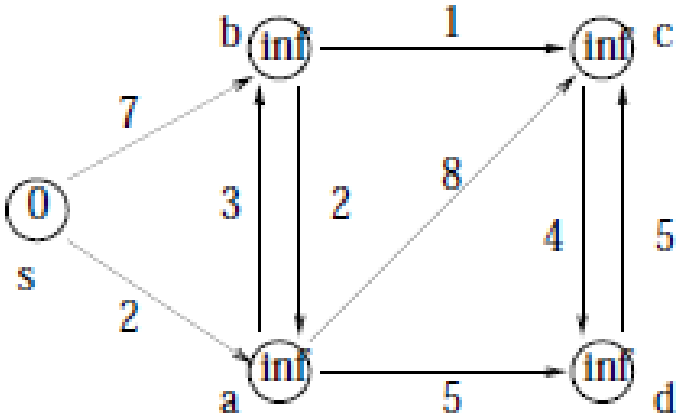
Description of Dijkstra's Algorithm

```
Dijkstra(G,w,s)
{                                                  % Initialize
    for (each u ∈ V)
    {
        d[u] = ∞;
        color[u] =white;
    }
    d[s] = 0;
    pred[s] = NIL;
    Q = (queue with all vertices);

    while (Non-Empty(Q))                           % Process all vertices
    {
        u = Extract-Min(Q);                        % Find new vertex
        for (each v ∈ Adj[u])
            if (d[u] + w(u, v) < d[v])             % If estimate improves
            {
                d[v] = d[u] + w(u, v);                 relax
                Decrease-Key(Q, v, d[v]);
                pred[v] = u;
            }
        color[u] =black;
    }
}
```

Dijkstra's Algorithm

Example:

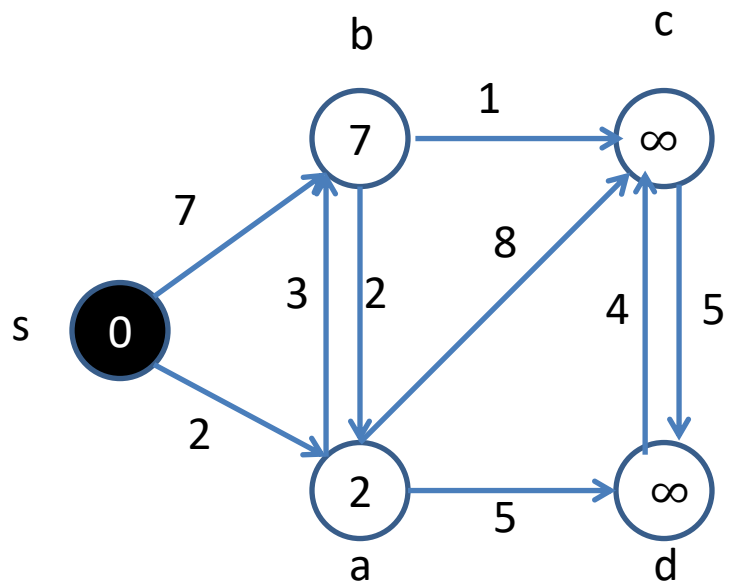


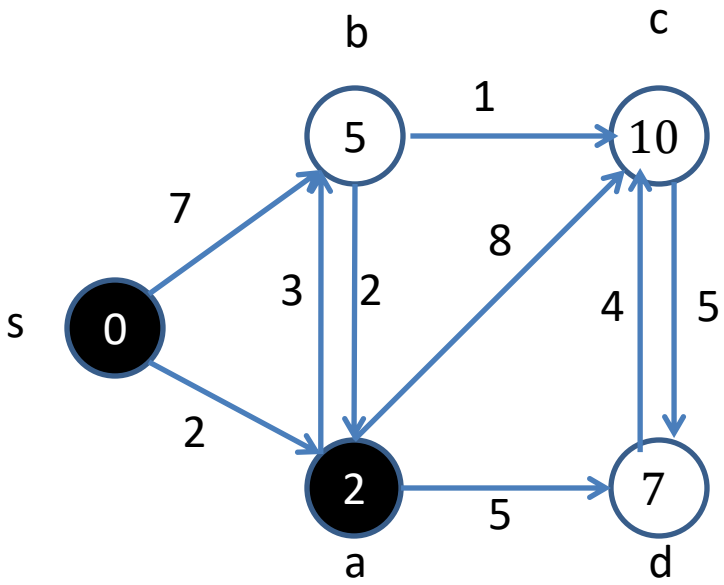
Step 0: Initialization.

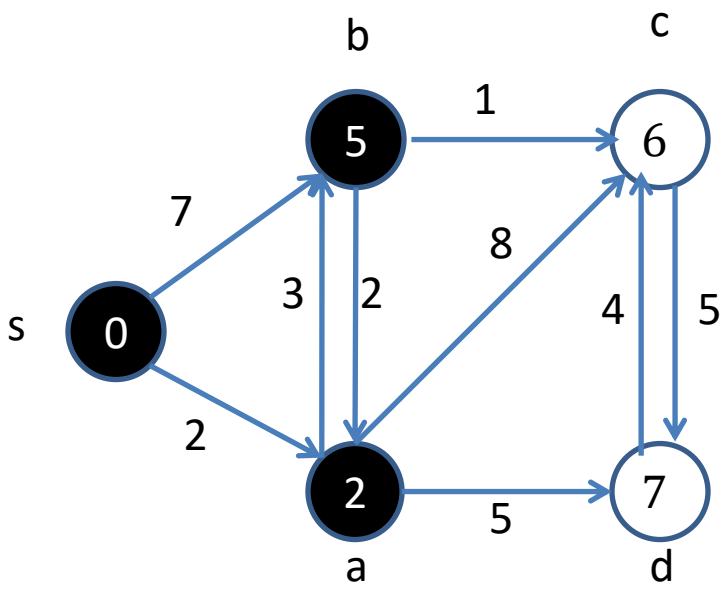
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

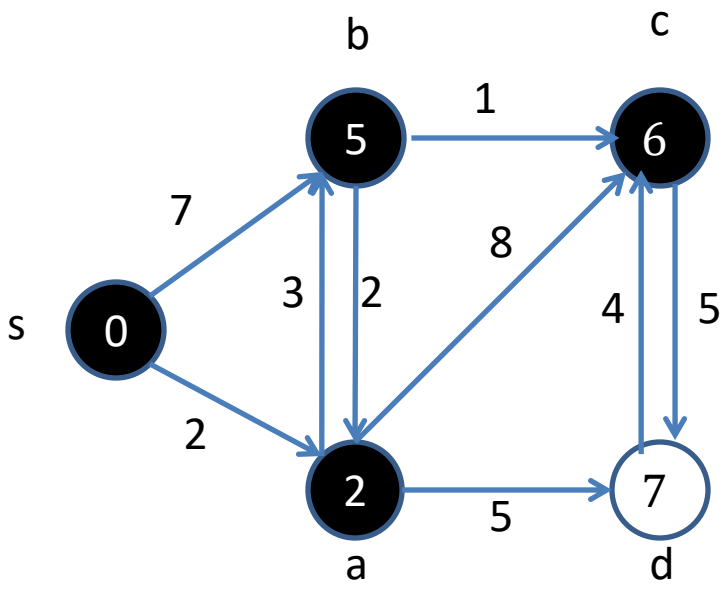
Priority Queue:

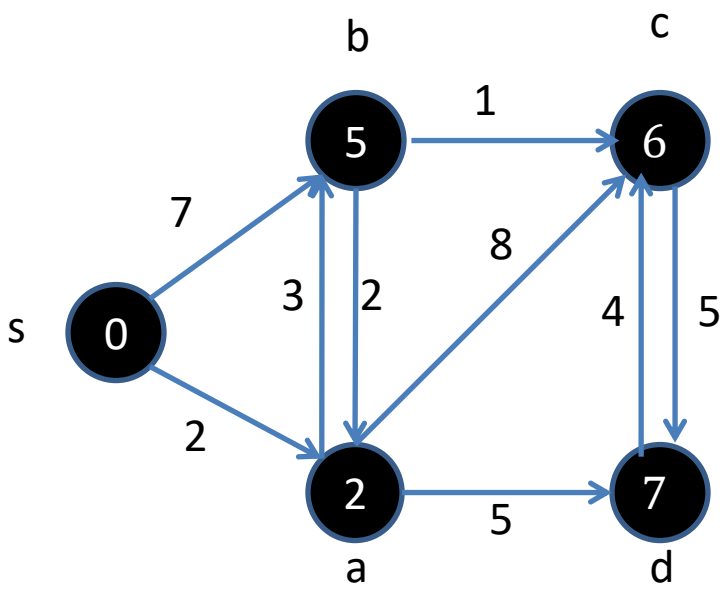
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞


b
c
7
1
∞
7
8
3
2
4
5
s
0
2
2
5
∞
a
d


b
c
5
1
10
7
8
3
2
4
5
s
0
2
2
5
7
a
d


b
c
1
5
6
7
8
3
2
4
5
s
0
2
2
7
5
a
d


b
c
1
5
6
7
8
3
2
4
5
s
0
2
2
7
5
a
d

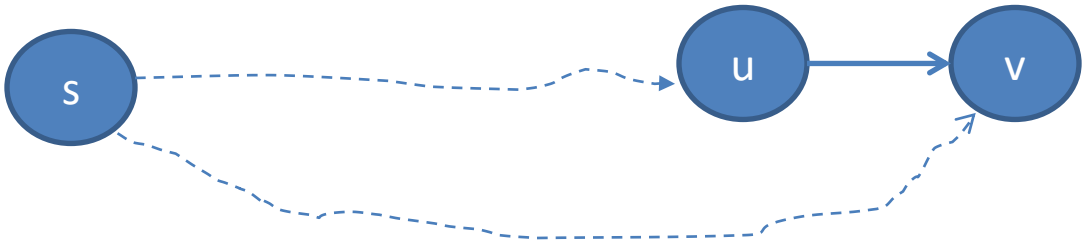

b
c
1
5
6
7
8
3
2
4
5
s
0
2
2
7
5
a
d

Graph Property

- Triangular Inequality:

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and source vertex s. Then, for all edges $(u, v) \in E$, we have

$$\delta(s, v) \leq \delta(s, u) + w(u, v) .$$



Upper Bound Property

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Let $s \in V$ be the source vertex, and let the graph be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Then, $v.d \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once $v.d$ achieves its lower bound $\delta(s, v)$, it never changes.

For the basis, $v.d \geq \delta(s, v)$ is certainly true after initialization, since $v.d = \infty$ implies $v.d \geq \delta(s, v)$ for all $v \in V - \{s\}$, and since $s.d = 0 \geq \delta(s, s)$

For the inductive step, consider the relaxation of an edge (u, v). By the inductive hypothesis, $x.d \geq \delta(s, x)$ for all $x \in V$ prior to the relaxation. The only d value that may change is $v.d$. If it changes, we have

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(by the inductive hypothesis)} \\ &\geq \delta(s, v) && \text{(by the triangle inequality)} , \end{aligned}$$

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $(u, v) \in E$. Then, immediately after relaxing edge (u, v) by executing RELAX(u, v, w), we have $v.d \leq u.d + w(u, v)$.

Proof If, just prior to relaxing edge (u, v), we have $v.d > u.d + w(u, v)$, then $v.d = u.d + w(u, v)$ afterward. If, instead, $v.d \leq u.d + w(u, v)$ just before the relaxation, then neither $u.d$ nor $v.d$ changes, and so $v.d \leq u.d + w(u, v)$ afterward. ■

Convergence Property

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, let $s \in V$ be a source vertex, and let $s \rightsquigarrow u \to v$ be a shortest path in G for some vertices $u, v \in V$. Suppose that G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps that includes the call RELAX(u, v, w) is executed on the edges of G. If $u.d = \delta(s, u)$ at any time prior to the call, then $v.d = \delta(s, v)$ at all times after the call.

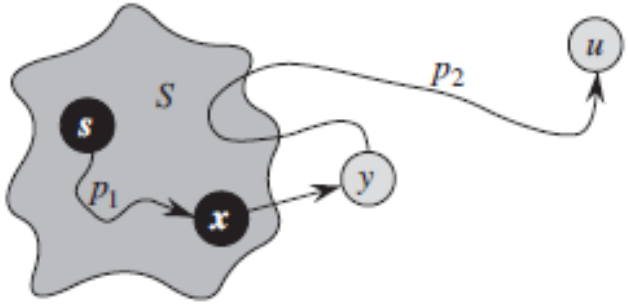
Proof By the upper-bound property, if $u.d = \delta(s, u)$ at some point prior to relaxing edge (u, v), then this equality holds thereafter. In particular, after relaxing edge (u, v), we have

$$\begin{aligned} v.d &\leq u.d + w(u, v) \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \end{aligned}$$

By the upper-bound property, $v.d \geq \delta(s, v)$, from which we conclude that $v.d = \delta(s, v)$, and this equality is maintained thereafter. ■

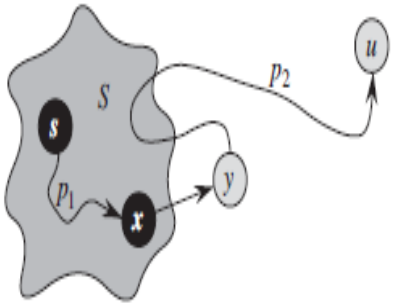
Correctness of Dijkstra's Algorithm

Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.



At the start of each iteration of the **while** loop , v.*d* = δ(s,v) for each vertex $v \in S$

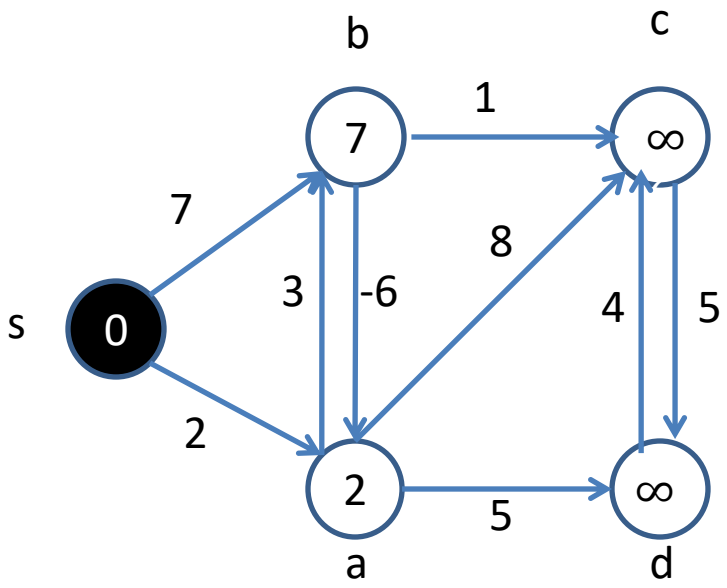
- At the beginning when only s ∈ S this is true as d.s =0 (as per initialization) which is the shortest path distance of δ(s,s).
- Assume that u is the first node for which it is violating.
- So, when u is added S is non empty as s is already included
- It can be also argued that there is a shortest path between s and u otherwise u.d and δ(s,u) both would be equal to $\propto$.
- Let us consider that iteration in which node u is added to set S
- Assume that in that path x is the last node that belongs to S and y is the first node that belongs to the V-S



- We can decompose the path like $s \overset{p1}{\rightsquigarrow} x \rightarrow y \overset{p2}{\rightsquigarrow} u.$ where there might not be any edge in either p1 or p2 or in both p1 and p2.
- For x, x.d=δ(s,x) as we have assumed that u is the first violating node
- From the convergence property y.d=δ(s,y)
- As y appears before u then δ(s,y)<δ(s,u)

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \end{aligned}$$

Problem with negative edge weight



Path Relaxation Property

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $s \in V$ be a source vertex. Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to v_k. If G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of the edges of p.

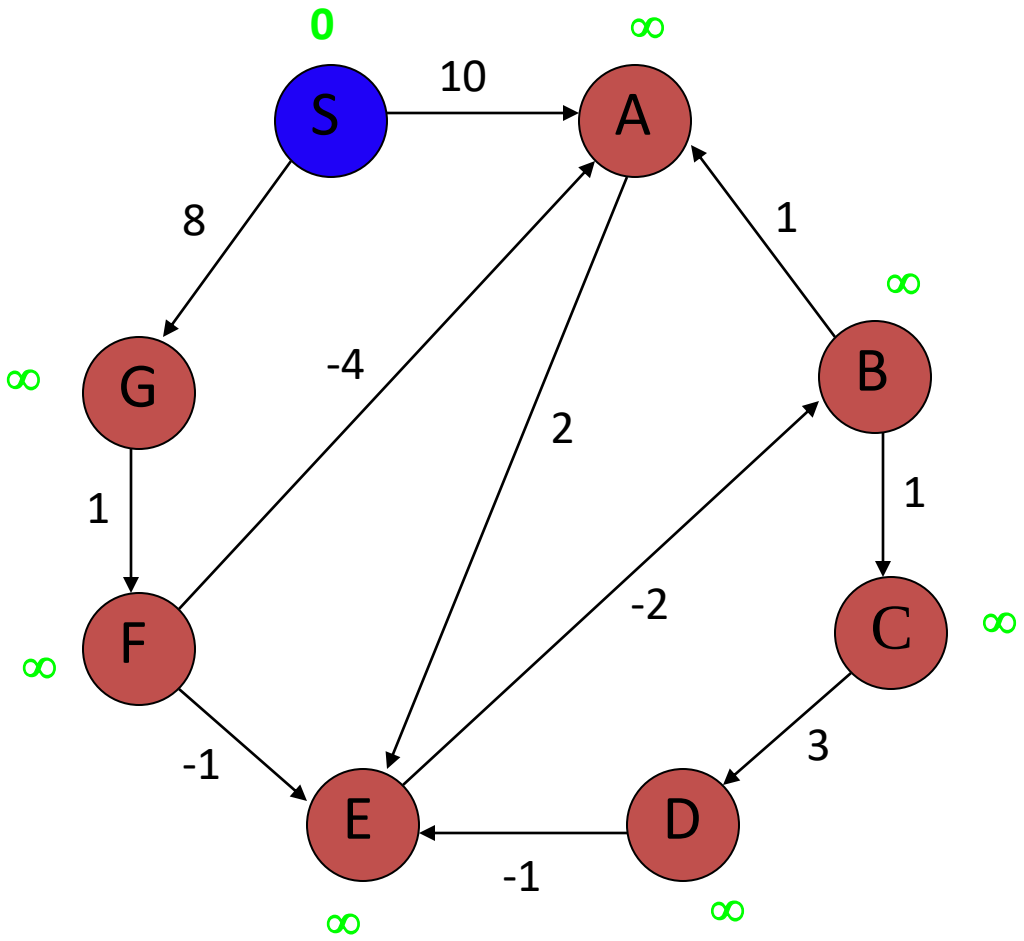
Proof We show by induction that after the ith edge of path p is relaxed, we have $v_i.d = \delta(s, v_i)$. For the basis, $i = 0$, and before any edges of p have been relaxed, we have from the initialization that $v_0.d = s.d = 0 = \delta(s, s)$. By the upper-bound property, the value of $s.d$ never changes after initialization.

For the inductive step, we assume that $v_{i-1}.d = \delta(s, v_{i-1})$, and we examine what happens when we relax edge (v_{i-1}, v_i). By the convergence property, after relaxing this edge, we have $v_i.d = \delta(s, v_i)$, and this equality is maintained at all times thereafter. ■

Bellman Ford Algorithm

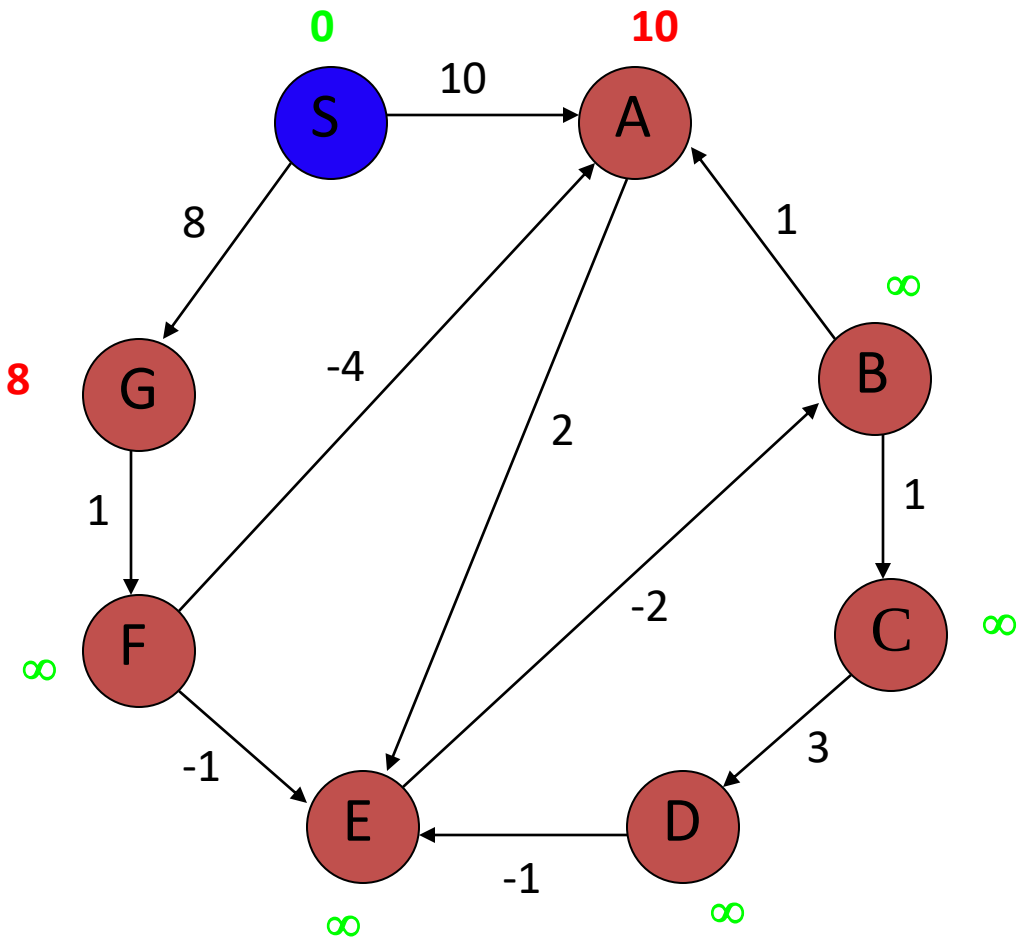
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Bellman-Ford algorithm



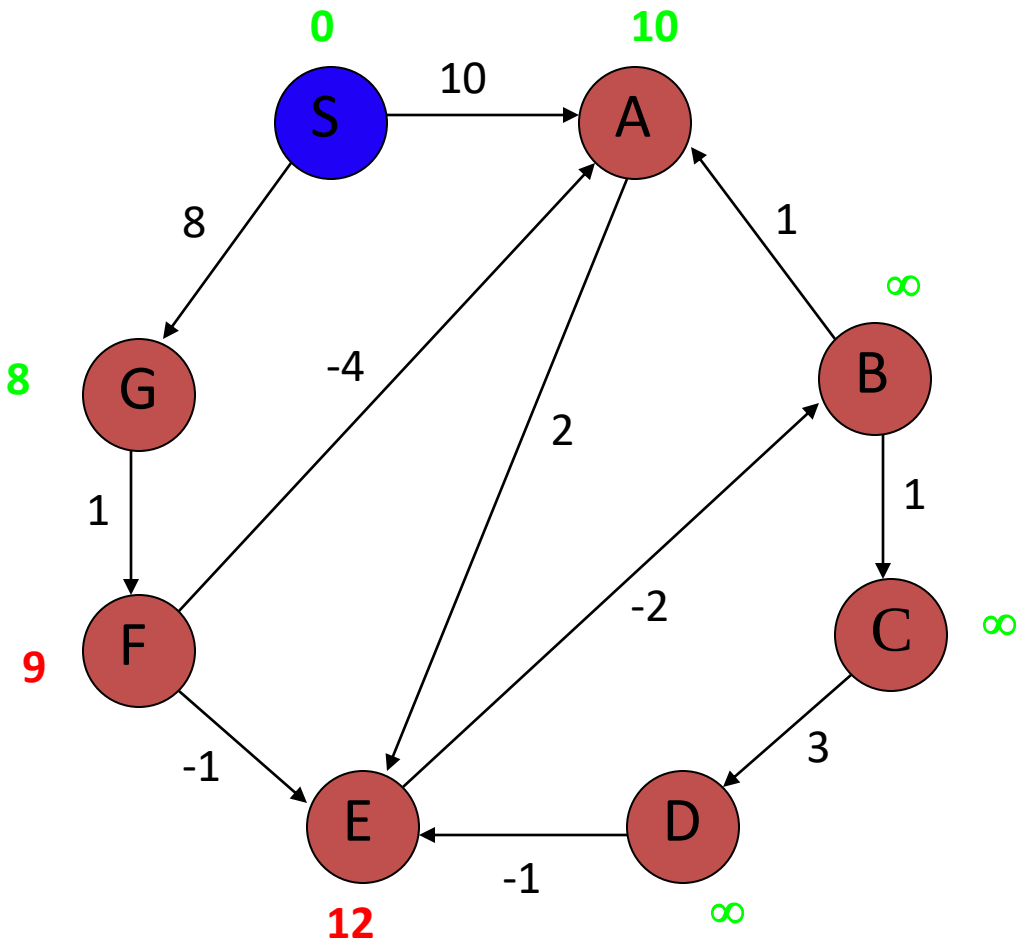
Iteration: 0

Bellman-Ford algorithm



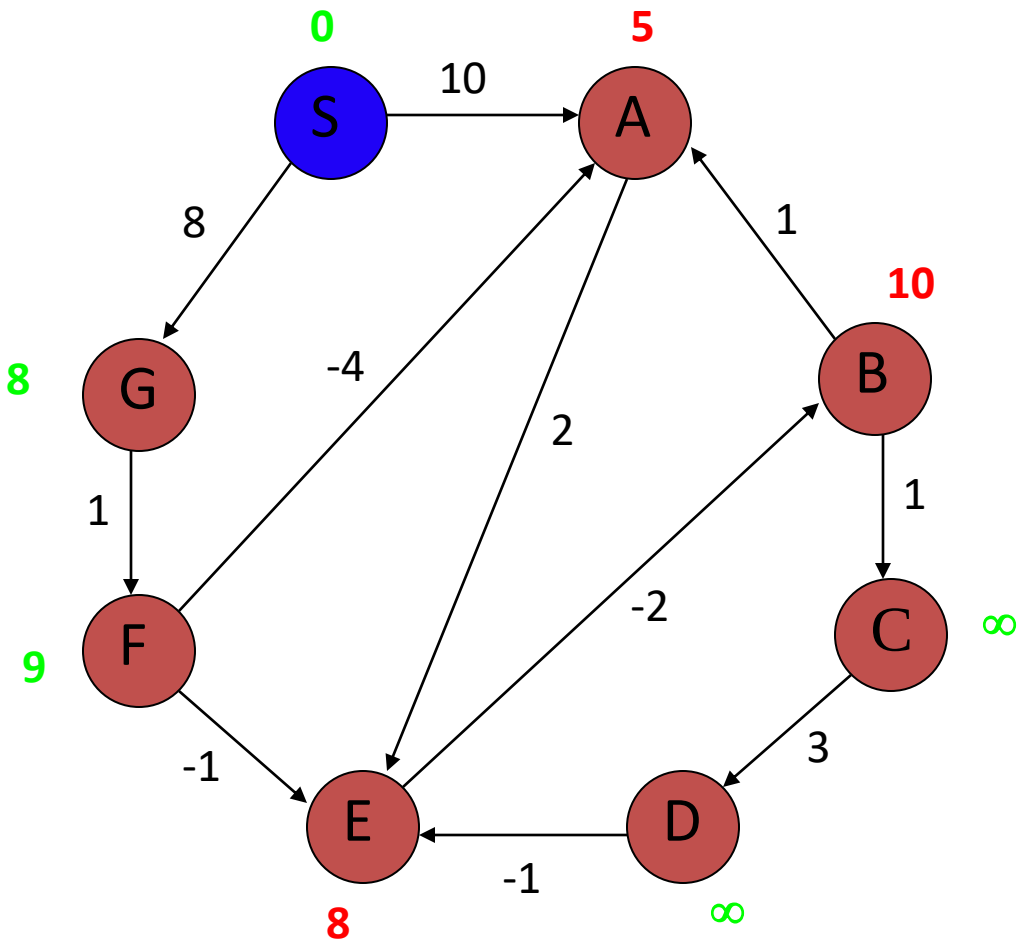
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

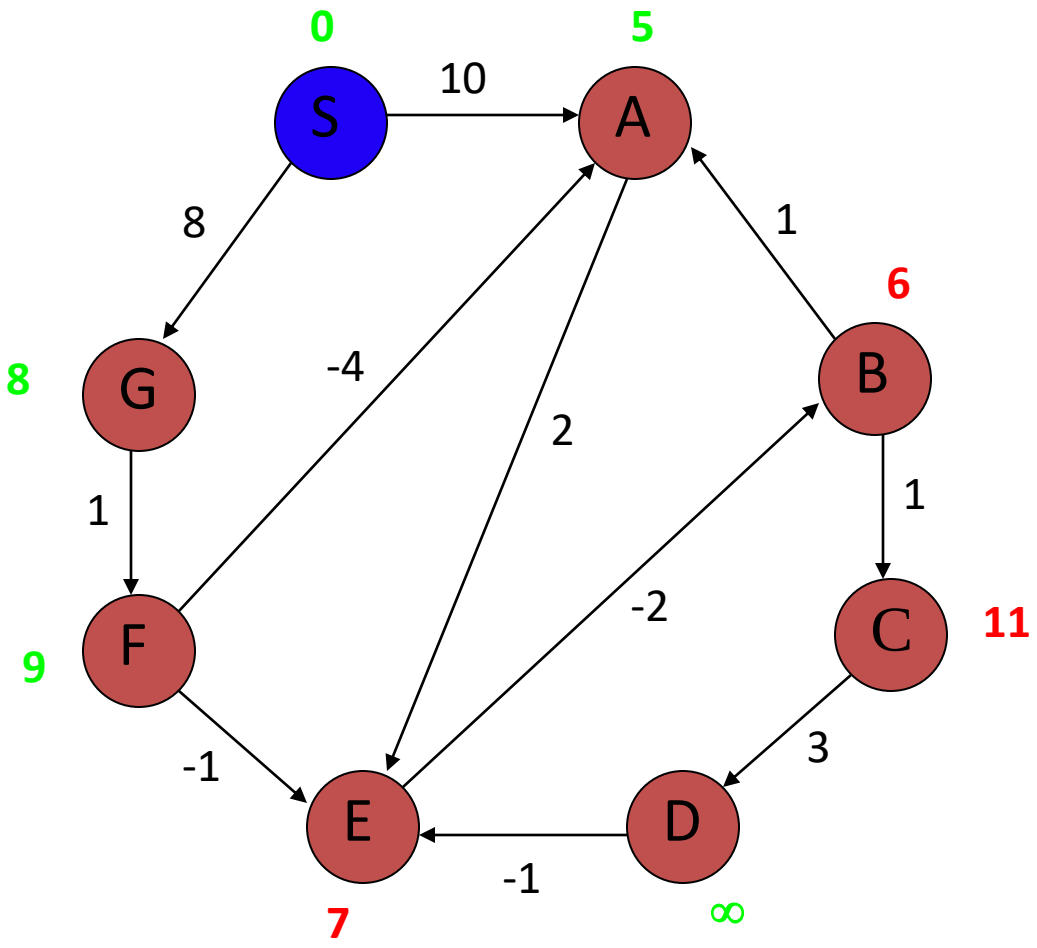
Bellman-Ford algorithm



Iteration: 3

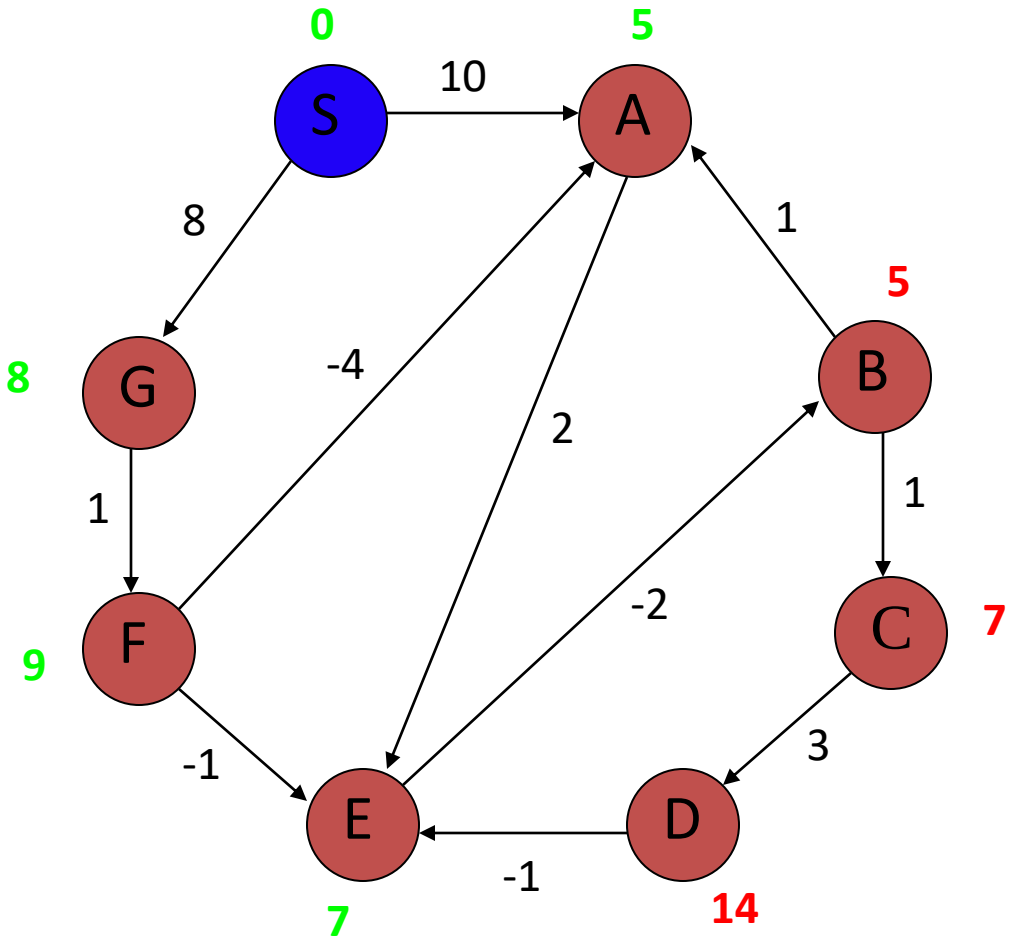
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

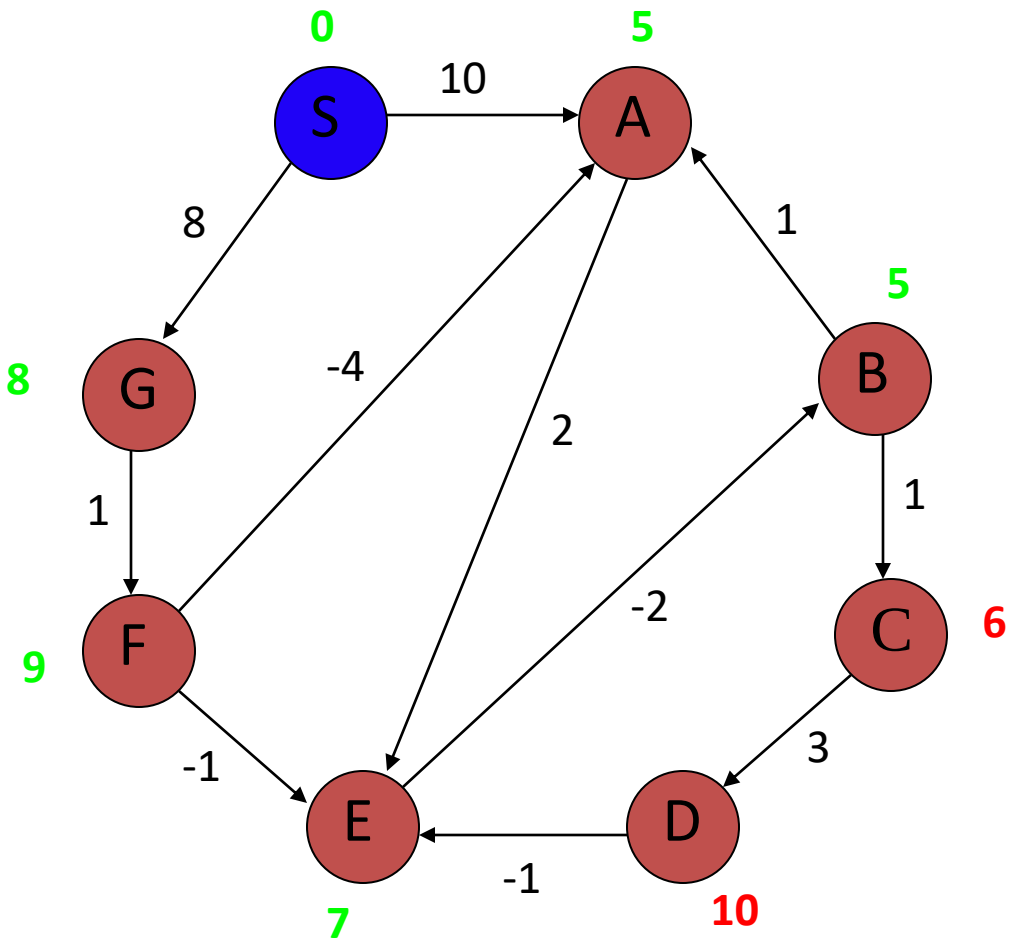
Bellman-Ford algorithm



Iteration: 5

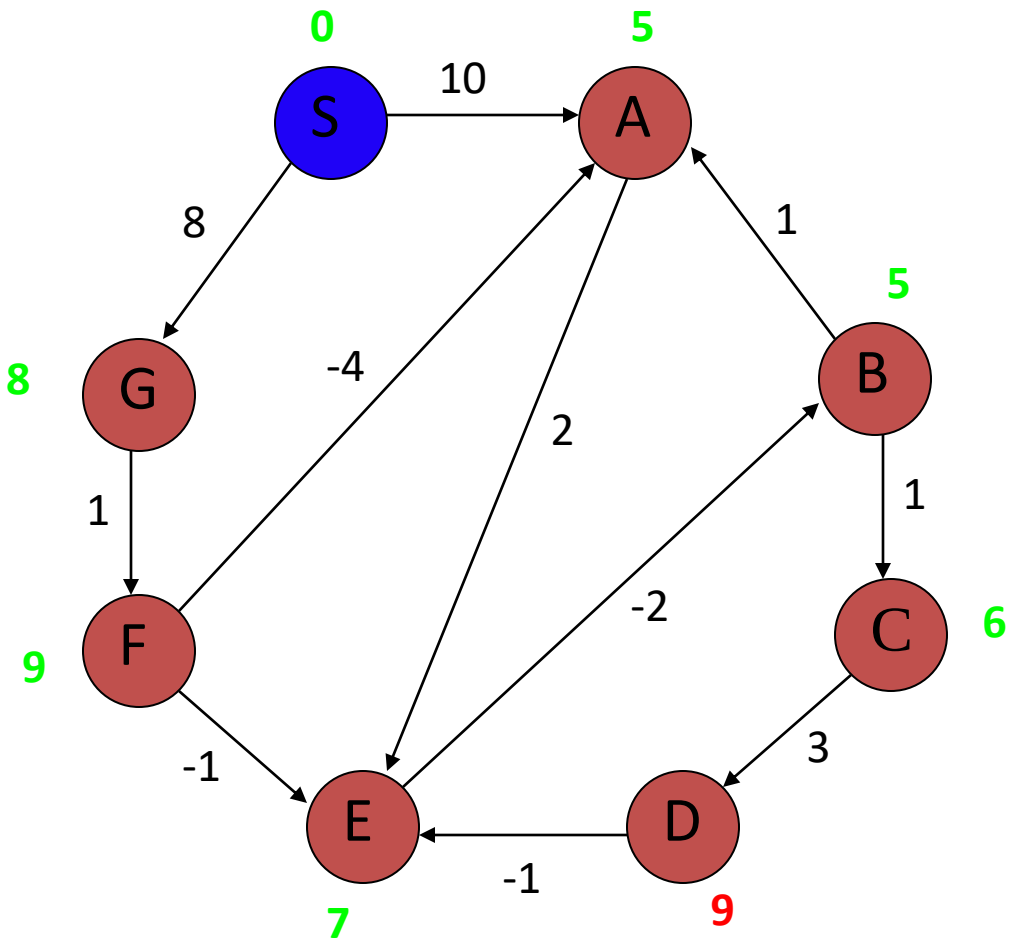
B has the correct distance and path

Bellman-Ford algorithm



Iteration: 6

Bellman-Ford algorithm



Iteration: 7

D (and all other nodes) have the correct distance and path

Correctness of Bellman-Ford

- Loop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances

All-Pairs Shortest Path Problem

Suppose we are given a directed graph G=(V,E) and a weight function w: E->R.

We assume that G does not contain cycles of weight 0 or less.

The All-Pairs Shortest Path Problem asks to find the length of the shortest path between any pair of vertices in G.

Quick Solutions

If the weight function is nonnegative for all edges, then we can use Dijkstra's single source shortest path algorithm for all vertices to solve the problem.

This yields an O(EVlogV) algorithm on graphs with |V| vertices (on dense graphs and with a simple implementation).

Quick Solution

For arbitrary weight functions, we can use the Bellman-Ford algorithm applied to all vertices. This yields an $O(V^4)$ algorithm for graphs with $|V|$ vertices.

Floyd-Warshall

We will now investigate a dynamic programming solution that solved the problem in $O(n^3)$ time for a graph with n vertices.

This algorithm is known as the Floyd-Warshall algorithm.

Representation of the Input

We assume that the input is represented by a weight matrix $W= (w_{ij})_{i,j \text{ in } E}$ that is defined by

$w_{ij}= 0$ if $i=j$

$w_{ij}= w(i,j)$ if $i \neq j$ and (i,j) in E

$w_{ij}= \infty$ if $i \neq j$ and (i,j) not in E

Format of the Output

If the graph has n vertices, we return a distance matrix (d_{ij}), where d_{ij} the length of the path from i to j.

Intermediate Vertices

Without loss of generality, we will assume that V={1,2,…,n}, i.e., that the vertices of the graph are numbered from 1 to n.

Given a path $p=(v_1, v_2,\ldots, v_m)$ in the graph, we will call the vertices v_k with index k in {2,…,m-1} the intermediate vertices of p.

Remark 2

Consider a shortest path p from i to j such that the intermediate vertices are from the set {1,…,k}.

- If the vertex k is not an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

  If the vertex k is an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Interestingly, in either case, the subpaths contain merely nodes from {1,…,k-1}.

Remark 2

Therefore, we can conclude that

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Recursive Formulation

If we do not use intermediate nodes, i.e., when k=0, then

$$d_{ij}^{(0)} = w_{ij}$$

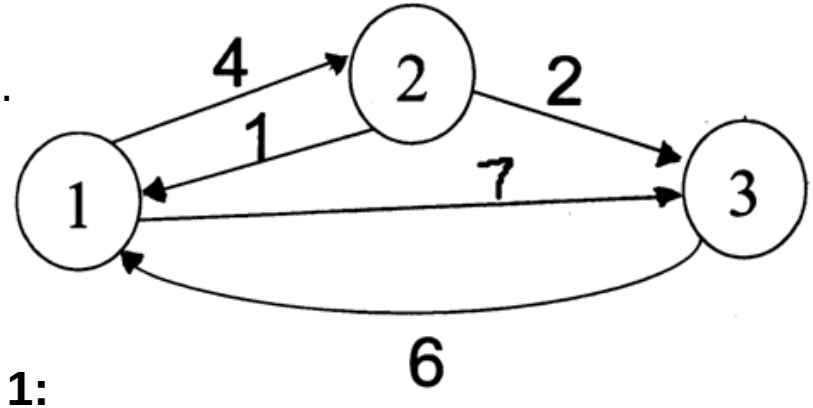
If k>0, then

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Floyd Warshall Algorithm - Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes.

The Floyd-Warshall Algorithm

Floyd-Warshall(W)
n = # of rows of W;
$D^{(0)} = W$;
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$;
        od;
    od;
od;
return $D^{(n)}$;

Floyd Warshall – Path Reconstruction

- The path matrix will store the last vertex visited on the path from i to j.
  - So path[i][j] = k means that in the shortest path from vertex i to vertex j, the LAST vertex on that path before you get to vertex j is k.

- Based on this definition, we must initialize the path matrix as follows:
  - path[i][j] = i if i!=j and there exists an edge from i to j
  - = NIL otherwise

- The reasoning is as follows:
  - If you want to reconstruct the path at this point of the algorithm when you aren't allowed to visit intermediate vertices, the previous vertex visited MUST be the source vertex i.
  - NIL is used to indicate the absence of a path.

Floyd Warshall – Path Reconstruction

- Before you run Floyd Warshall, you initialize your distance matrix ***D*** and path matrix ***P*** to indicate the use of no intermediate vertices.
  - (Thus, you are only allowed to traverse direct paths between vertices.)

- Then, at each step of Floyd Warshall, you essentially find out whether or not using vertex k as intermediate vertex will *improve* an estimate between the distances between vertex i and vertex j.

Floyd Warshall – Path Reconstruction

- If it ***does improve*** the estimate here's what you need to record:
  1) record the new shortest path weight between i and j
  2) record the fact that the shortest path between i and j goes through k
     - **We want to store the last vertex from the shortest path from vertex k to vertex j. *This will NOT necessarily be k, but rather, it will be path[k][j].***

This gives us the following update to our algorithm:

```
if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as intermediate vertex
    D[i][j] = D[i][k]+D[k][j];
    path[i][j] = path[k][j];
}
```

Path Reconstruction

- Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path.
  - Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

NIL	**3**	**4**	**5**	**1**
4	**NIL**	**4**	**2**	**1**
4	**3**	**NIL**	**2**	**1**
4	**3**	**4**	**NIL**	**1**
4	**3**	**4**	**5**	**NIL**

- Reconstruct the path from vertex1 to vertex 2:
  - First look at path[1][2] = 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2.
    - Thus, the path is now, 1…3->2.
  - Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4.
  - So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5,
  - thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1,
  - which means our path really is 1->5->4->3->2.

Johnson's algorithm for sparse graphs

- If all edge weights w in a graph G = (V,E) are nonnegative then we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex.
- If all edges are not non-negative, then we have to do some ***reweighting*** so that all edge weights become non-negative

  w→ $\hat{w}$

1. For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\hat{w}$.
2. For all edges (u, v), the new weight $\hat{w}(u, v)$ is nonnegative.

Preserving shortest paths by reweighting

- ***Reweighting does not change shortest paths***

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) .$$

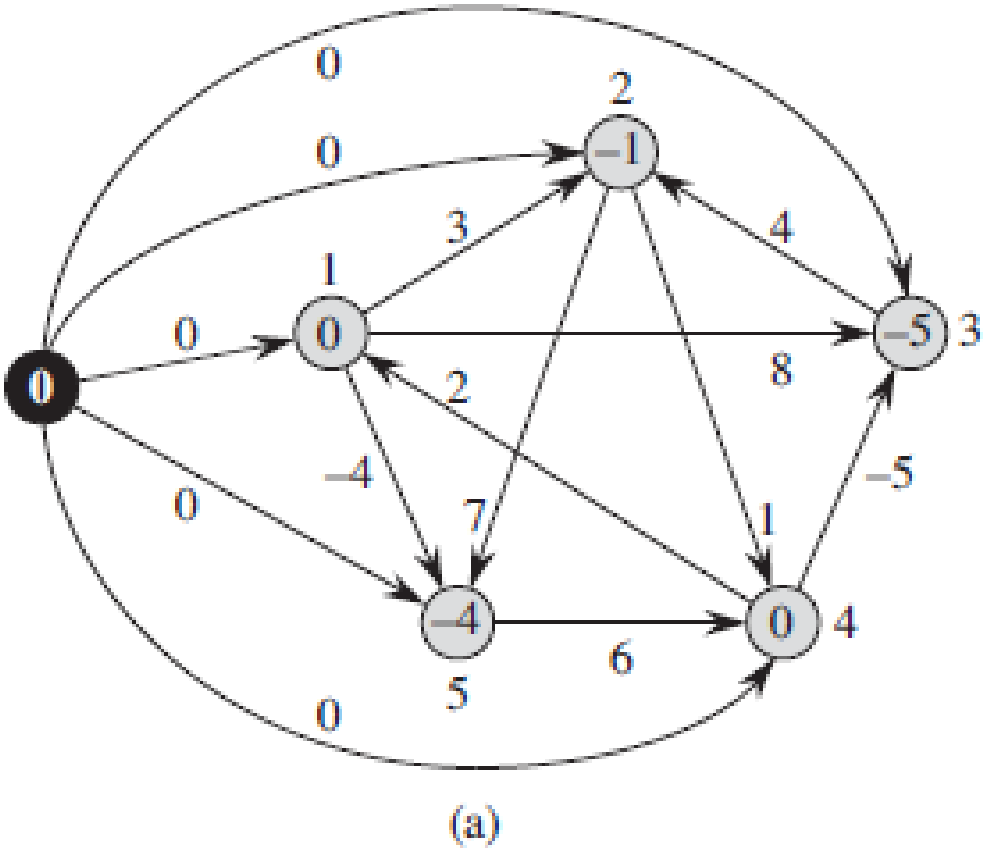
Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex v_0 to vertex v_k. Then p is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function $\hat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\hat{w}$.

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) .$$

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\hat{w}$. Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$.

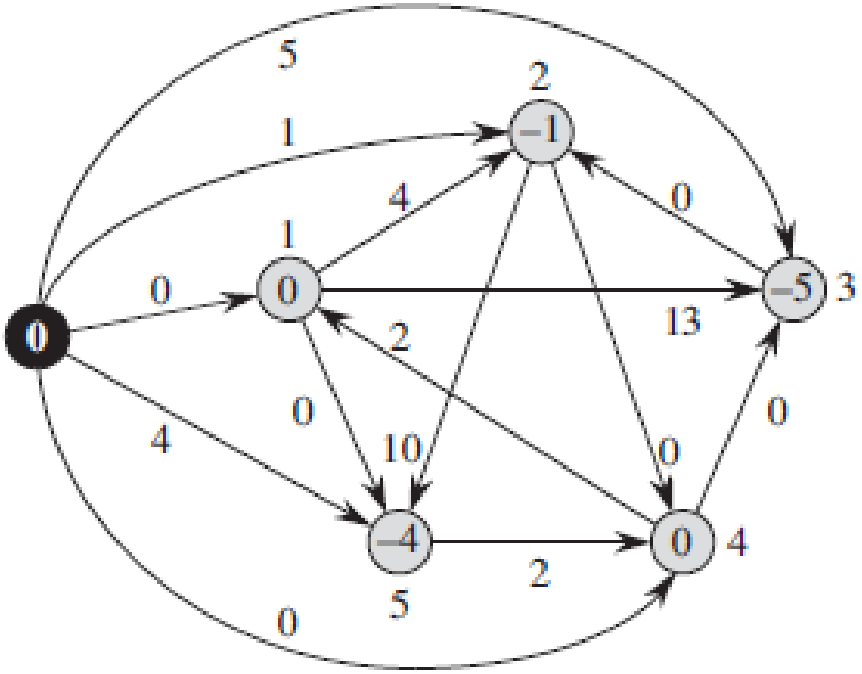
$$\begin{aligned} \hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) , \end{aligned}$$

Producing nonnegative weights by reweighting

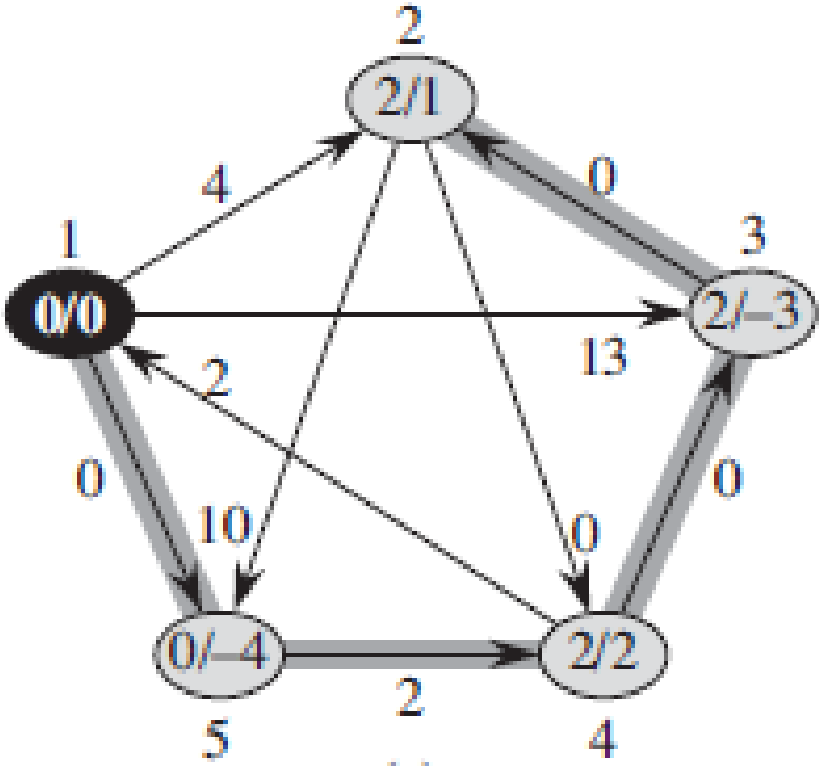


(a)

A new vertex is inserted and all existing vertices are connected from new vertex using edge with weight 0 and then bellman ford is executed using new node as source



$$\hat{w}(u, v) = w(u, v) + h(u) - h(v).$$


2
2/1
4
0
1
3
0/0
2/–3
13
2
0
10
0
0
0/–4
2/2
2
5
4

```
JOHNSON(G, w)
 1  compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
 2  if BELLMAN-FORD(G', w, s) == FALSE
 3      print "the input graph contains a negative-weight cycle"
 4  else for each vertex v ∈ G'.V
 5          set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
 6      for each edge (u, v) ∈ G'.E
 7          ŵ(u, v) = w(u, v) + h(u) − h(v)
 8      let D = (d_uv) be a new n × n matrix
 9      for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11          for each vertex v ∈ G.V
12              d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```