Lecture 11
Shortest Paths

Department of Computer Science
Hofstra University

Lecture Goals

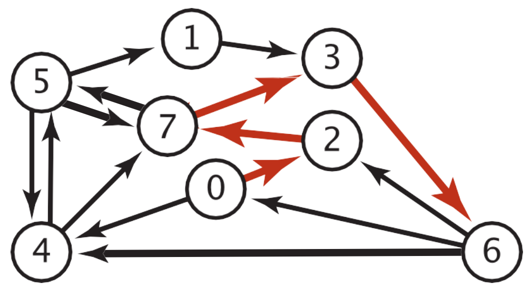
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- We introduce and analyze Dijkstra's algorithm for shortest-paths problems with nonnegative weights.
- Next, we consider Topological Sort for Edge-weighted DAG, which works even if the weights are negative.
- Next, we consider the Bellman–Ford algorithm for edge-weighted digraphs with no negative cycles.
- We conclude with the Floyd Warshall Algorithm for all-pairs shortest path

Shortest Paths in an Edge-weighted Digraph

Given an edge-weighted digraph, find the shortest path from s to t.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



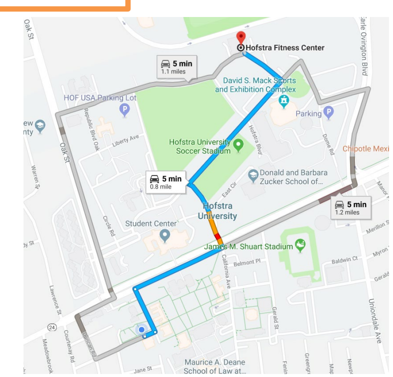
shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Can we use BFS?

Variants

- **Which vertices?**
  - Single source: from one vertex s to every other vertex.
  - Source-sink: from one vertex s to another t.
  - All pairs: between all pairs of vertices.
- **Nonnegative weights?**
- **Cycles?**
  - Negative cycles.



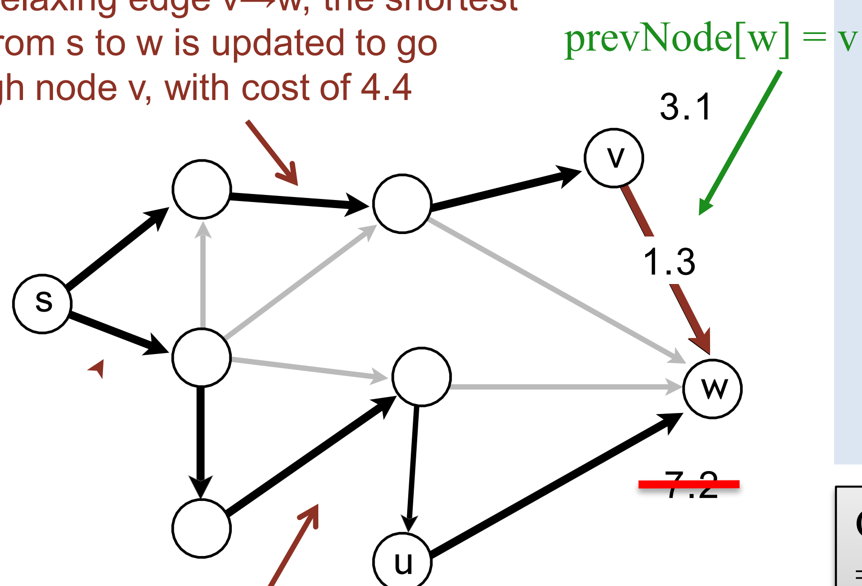
Simplifying assumption: Each vertex is reachable from s.

Edge Relaxation

Relax edge e = v→w. (basic of building SPT)

- distTo[v] is length of shortest known path from s to v.
- distTo[w] is length of shortest known path from s to w.
- prevNode[w] is the previous node on shortest known path from s to w.
- If e = v→w gives shorter path to w through v, update distTo[w] and prevNode[w].
  - distTo[w] = min(distTo[w], distTo[v] + e.weight()); prevNode[w]=v

After relaxing edge v→w, the shortest path from s to w is updated to go through node v, with cost of 4.4



Previous shortest path from s to w goes through node u, with cost of 7.2

```
private void relax(DirectedEdge e)
{
    int  v  = e.from(), w  = e.to();
    if  (distTo[w] >  distTo[v] + e.weight())
    {
        distTo[w] =  distTo[v] +
        e.weight();
        prevNode[w] = v;
    }
}
```

OLD distTo[w] = 7.2 > distTo[v] + e.weight() = 3.1+1.3 = 4.4

NEW distTo[w] ← distTo[v] + e.weight() = 4.4, prevNode[w] = v

Generic Shortest-paths Algorithm

Generic algorithm (to compute SPT from s)
For each vertex v: distTo[v] = ∞.
For each vertex v: prevNode[v] = null.
distTo[s] = 0.
Repeat until done:
- Relax any edge.

Proposition. Generic algorithm computes SPT (if it exists) from s.

Pf.

- Throughout algorithm, distTo[v] is the length of a simple path from s to v (and prevNode[v] is its previous node on the path).
- Each successful relaxation decreases distTo[v] for some v.
- The entry distTo[v] can decrease at most a finite number of times.

Efficient implementations. How to choose which edge to relax?

- Ex 1. Dijkstra's algorithm. (no negative weights).
- Ex 2. Bellman–Ford algorithm. (negative weights, no negative cycles).

Dijkstra's Algorithm

- Initialization:
  - Set the distance to the source node as 0 and to all other nodes as infinity.
  - Mark all nodes as unvisited and store them in a priority queue.
- Main Loop:
  - Visit the unvisited node v with the shortest known distance from the queue.
  - For each unvisited neighbor node w of node v, calculate its tentative distance through the current node. If this distance is smaller than the previously recorded distance, update it with edge relaxation for edge v-w.
  - Mark the current node as visited once all its neighbors are processed.
- Termination:
  - The algorithm continues until all reachable nodes are visited, or until the shortest path to a specific destination is found.
- (Note: Dijkstra's algorithm works for both undirected and directed graphs. The only difference is the function for getting the neighbors of node v, which follows the edge arrow direction for directed graphs.)

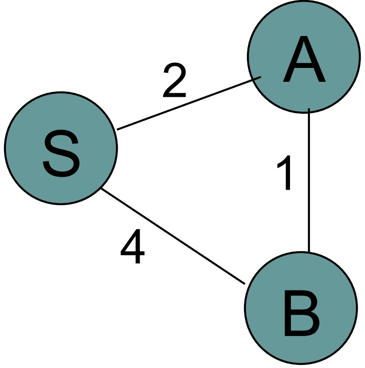
Dijkstra's Algorithm: Correctness Proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge e = v→w is relaxed exactly once (when v is relaxed),
  - leaving distTo[w] ≤ distTo[v] + e.weight().
- Inequality holds until algorithm terminates because:
  - distTo[w] cannot increase ← distTo[ ] values are monotone decreasing
  - distTo[v] will not change ← we choose lowest distTo[ ] value at each step (and edge weights are nonnegative)
- Thus, upon termination, shortest-paths optimality conditions hold.

Toy Example: Run Dijkstra's algorithm starting from source node S



N1	SD	PN
S	0	
A	∞	
B	∞	

Visit S

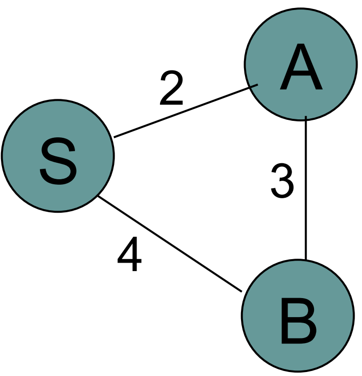
N1	SD	PN
S	0	
A	2	S
B	4	S

Visit A

N1	SD	PN
S	0	
A	2	S
B	3	A

Visit B

N1	SD	PN
S	0	
A	2	S
B	3	A



N1	SD	PN
S	0	
A	∞	
B	∞	

Visit S

N1	SD	PN
S	0	
A	2	S
B	4	S

Visit A

N1	SD	PN
S	0	
A	2	S
B	4	S

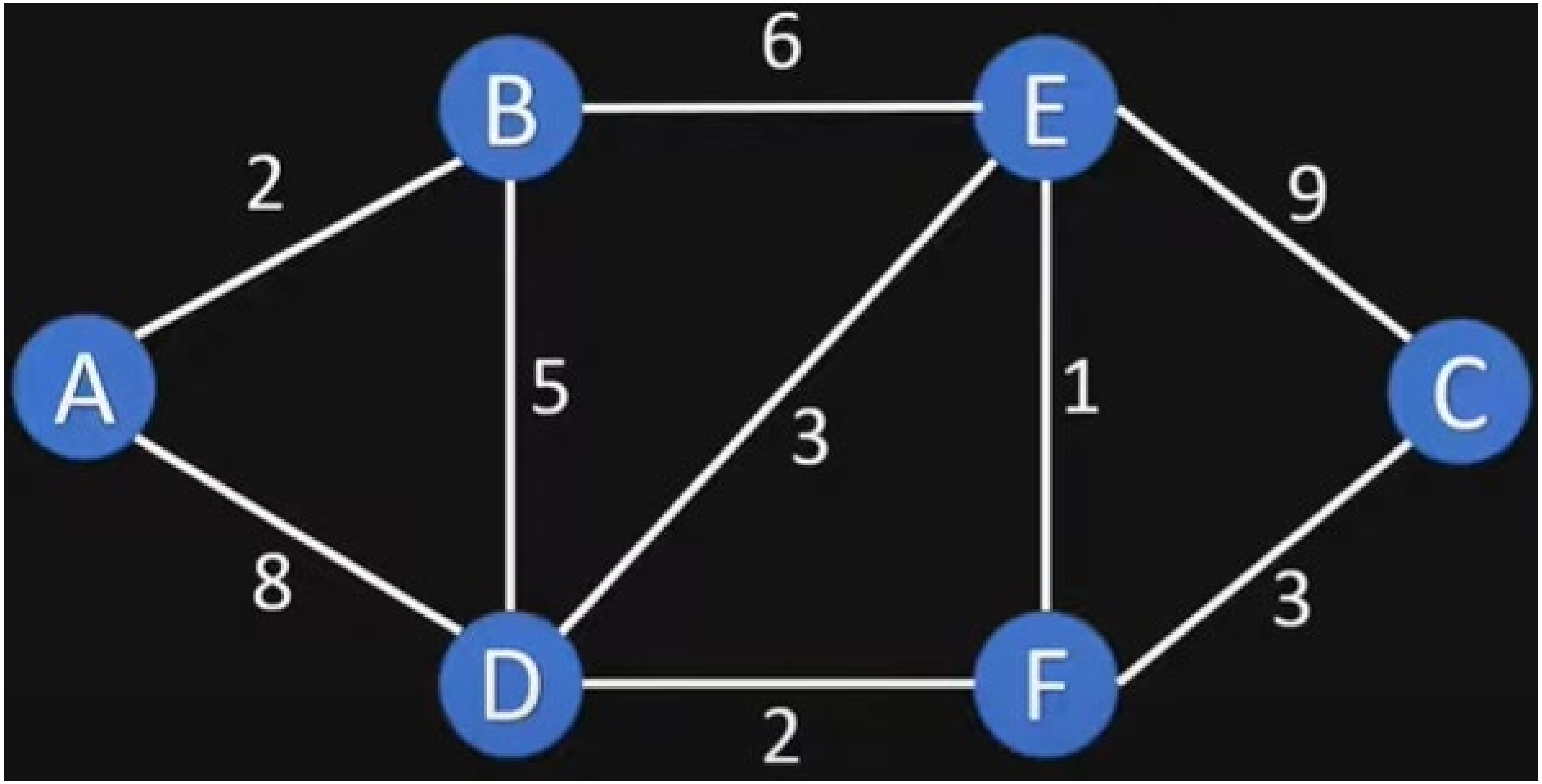
Visit B

N1	SD	PN
S	0	
A	2	S
B	4	S

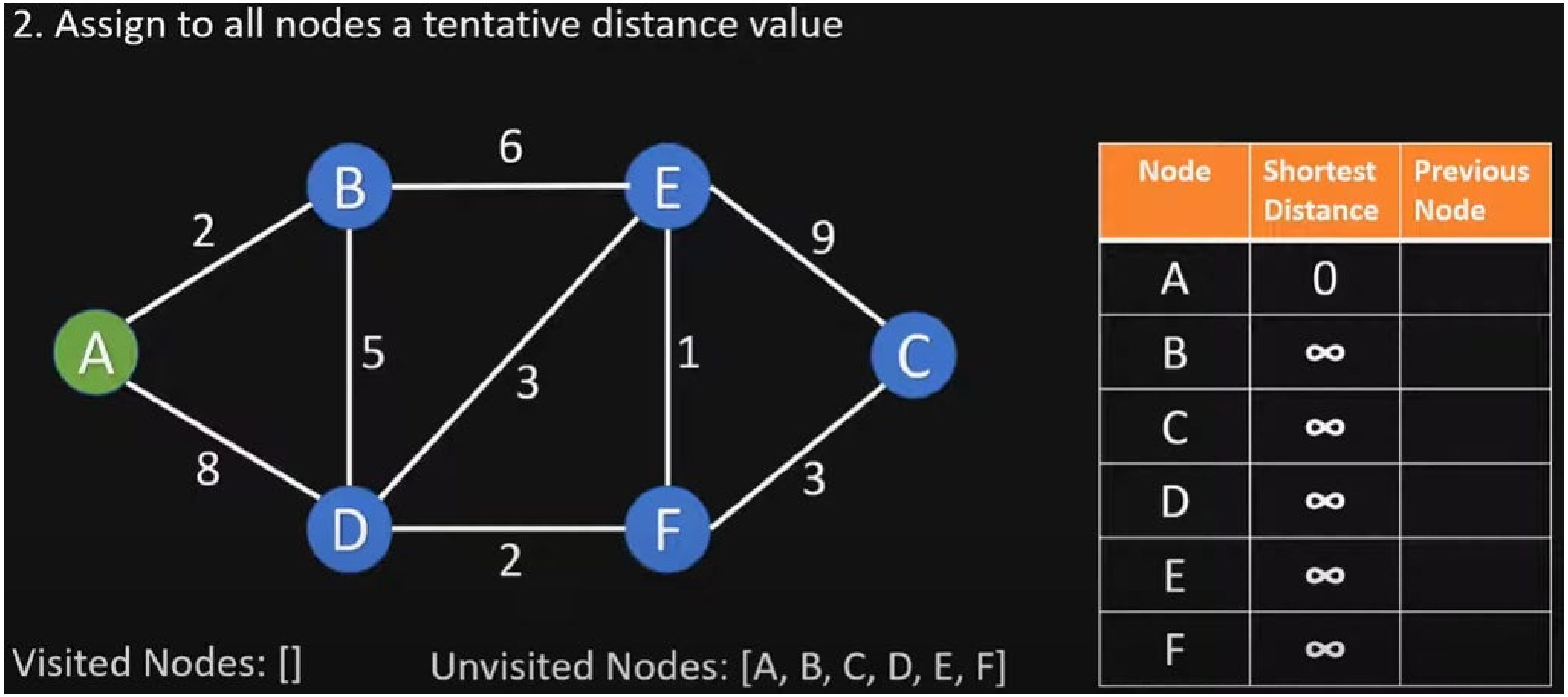
Video Tutorials

- Dijkstras Shortest Path Algorithm Explained | With Example | Graph Theory
  - https://www.youtube.com/watch?v=bZkzH5x0SKU
  - The following lecture slides are based on this video
- Dijkstra's algorithm in 3 minutes
  - https://www.youtube.com/watch?v=_lHSawdgXpI
- Bellman-Ford in 4 minutes — Theory
  - https://www.youtube.com/watch?v=9PHkk0UavIM
- Bellman-Ford in 5 minutes — Step by step example
  - https://www.youtube.com/watch?v=obWXjtg0L64

Example Graph

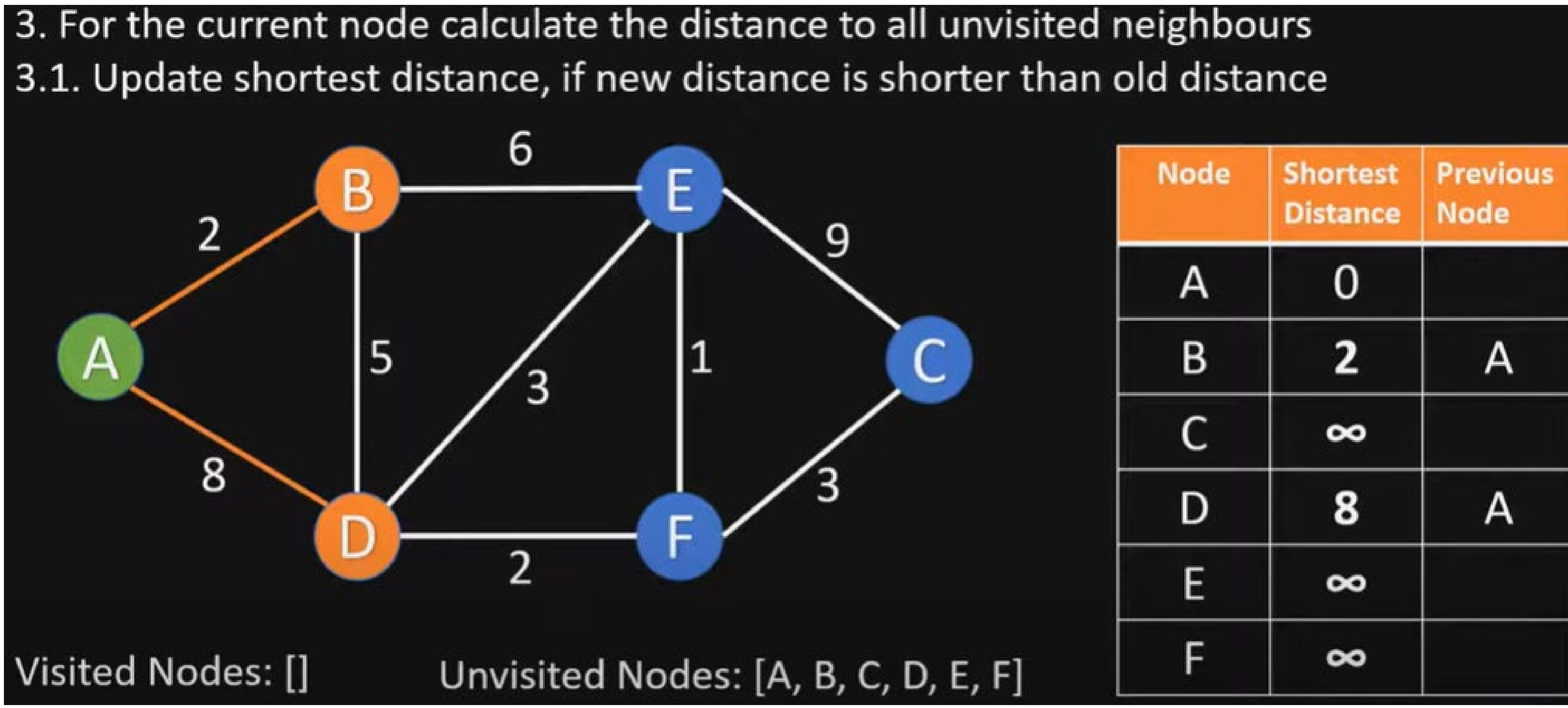


Initialize



Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Visit Node A

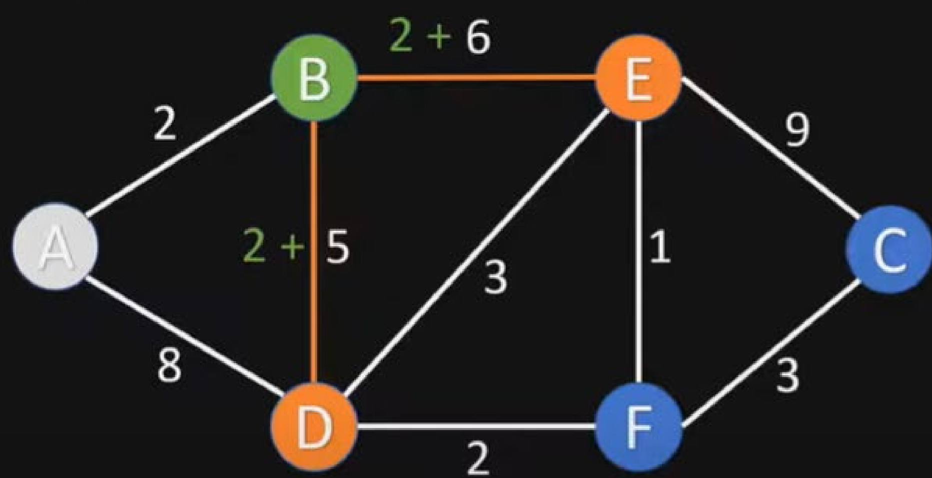


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

OLD distTo[B] = ∞ > distTo[A] + e[A][B].weight() = 0+2 = 2
NEW distTo[B] $\leftarrow$ distTo[A] + e[A][B].weight() = 2, prevNode[B] = A
OLD distTo[D] = ∞ > distTo[A] + e[A][D].weight() = 0+8 = 8
NEW distTo[D] $\leftarrow$ distTo[A] + e[A][D].weight() = 8, prevNode[D] = A

Visit Node B

3. For the current node calculate the distance to all unvisited neighbours
3.1. Update shortest distance, if new distance is shorter than old distance



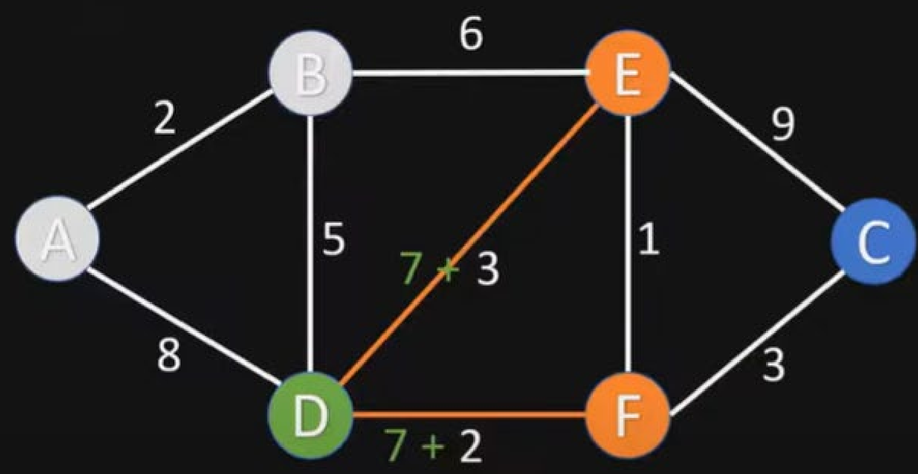
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

Visited Nodes: [A]   Unvisited Nodes: [B, C, D, E, F]

OLD distTo[D] = 8 > distTo[B] + e[B][D].weight() = 2+5 = 7
NEW distTo[D] ← distTo[B] + e[B][D].weight() = 7, prevNode[D] = B
OLD distTo[E] = ∞ > distTo[B] + e[B][E].weight() = 2+6 = 8
NEW distTo[E] ← distTo[B] + e[B][E].weight() = 8, prevNode[E] = B

Visit Node D

3. For the current node calculate the distance to all unvisited neighbours
3.1. Update shortest distance, if new distance is shorter than old distance



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	**9**	D

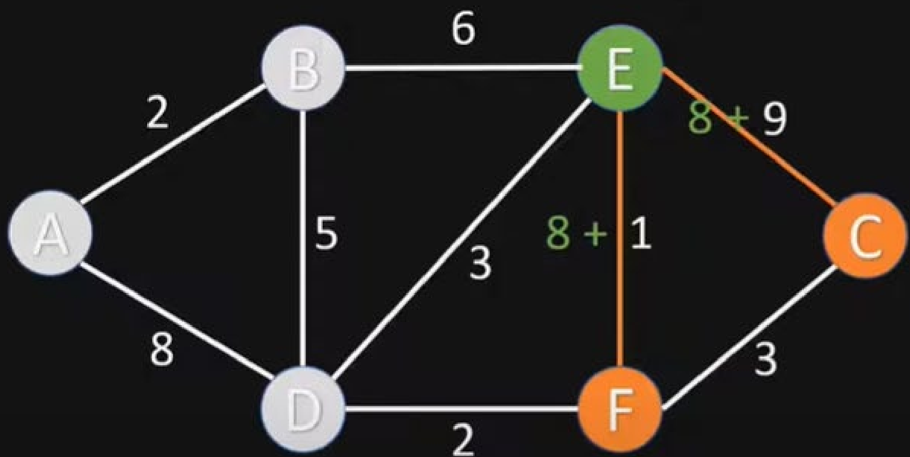
Visited Nodes: [A, B]   Unvisited Nodes: [C, D, E, F]

OLD distTo[E] = 8 < distTo[D] + e[D][E].weight() = 7+3 = 10
No update, distTo[E] stays 8, prevNode[E] stays B
OLD distTo[F] = ∞ > distTo[D] + e[D][F].weight() = 7+2 = 9
NEW distTo[F] ← distTo[D] + e[D][E].weight() = 9, prevNode[F] = D

Visit Node E

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance



Visited Nodes: [A, B, D] Unvisited Nodes: [C, E, F]

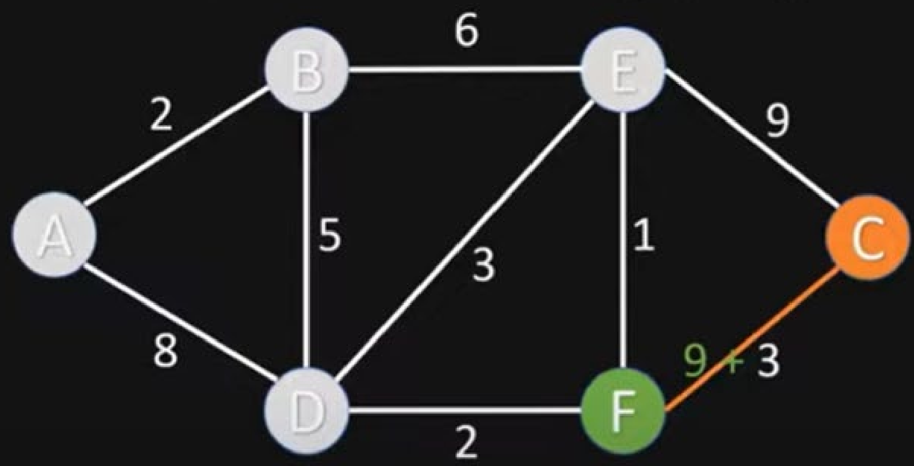
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

OLD distTo[C] = ∞ > distTo[E] + e[E][C].weight() = 8+9 = 17
NEW distTo[C] ← distTo[E] + e[E][C].weight() = 17, prevNode[C] = E
OLD distTo[F] = 9 = distTo[E] + e[E][F].weight() = 8+1 = 9
No update, distTo[F] stays 9, prevNode[F] = D (You can also update prevNode[F] = E.)

Visit Node F

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance



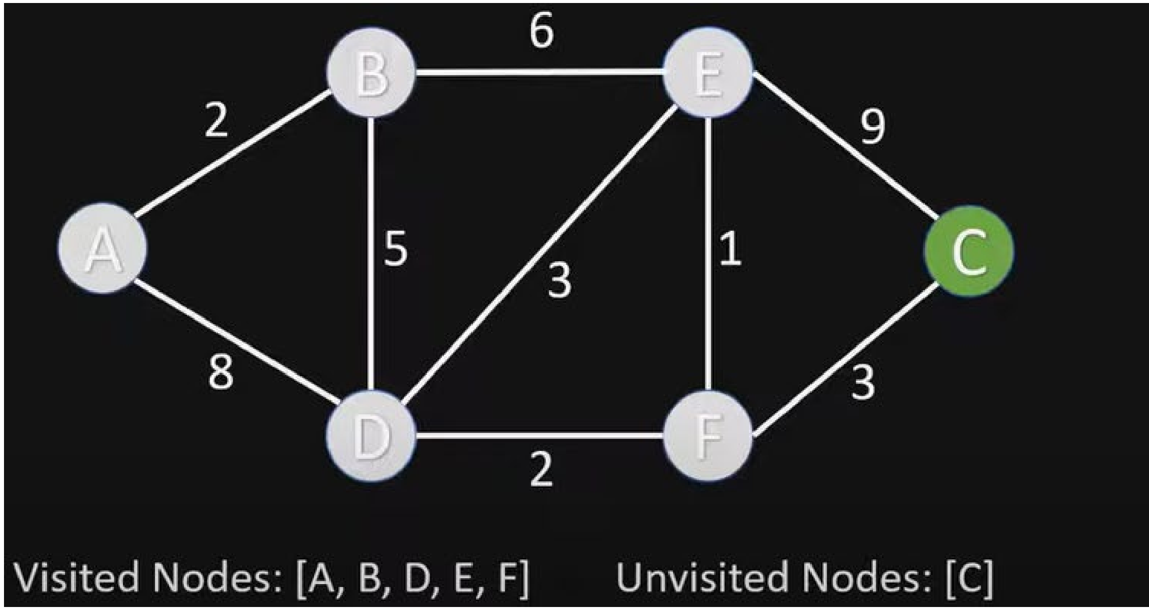
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D, E]   Unvisited Nodes: [C, F]

OLD distTo[C] = 17 > distTo[F] + e[F][C].weight() = 9+3 = 12

NEW distTo[C] ← distTo[F] + e[F][C].weight() = 12, prevNode[C] = F

Visit Node C

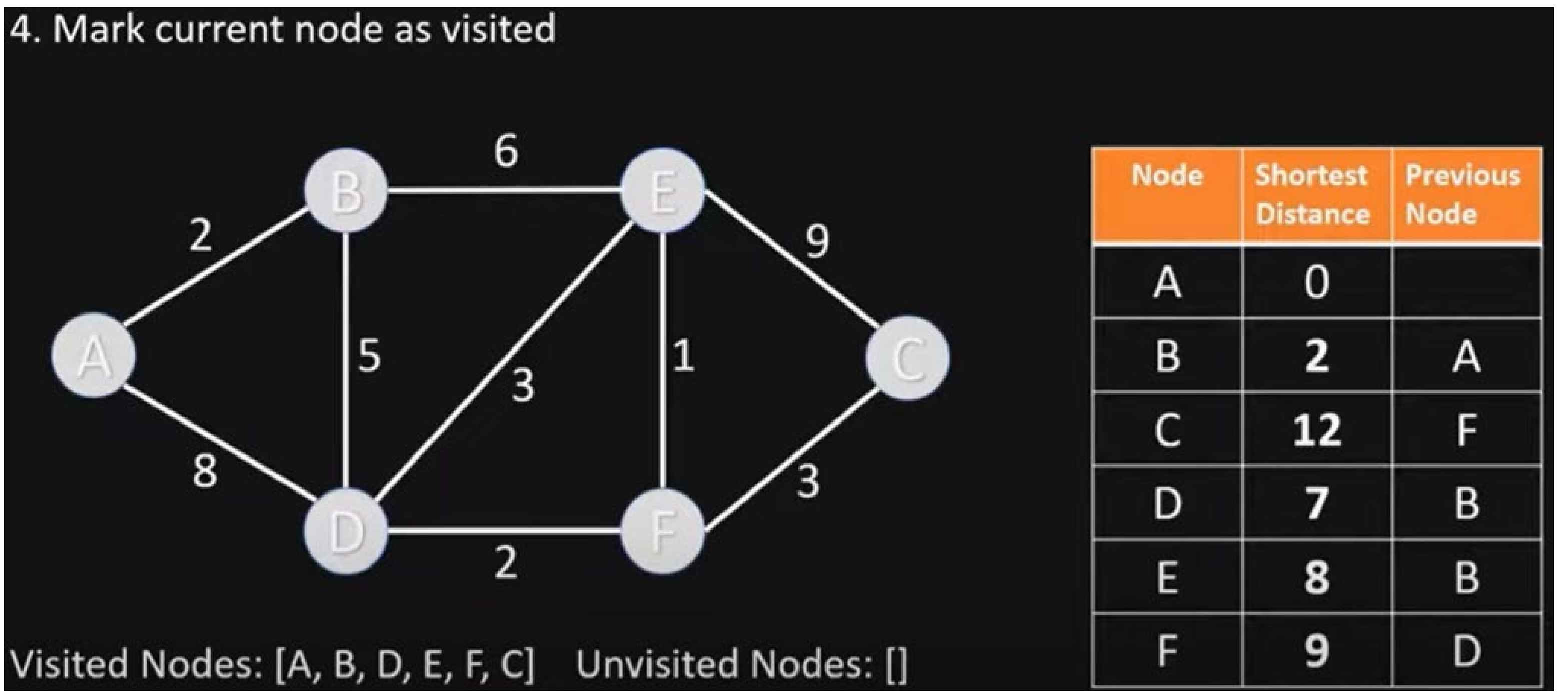


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Nothing changes, since C has no unvisited neighbor nodes

End of Algorithm

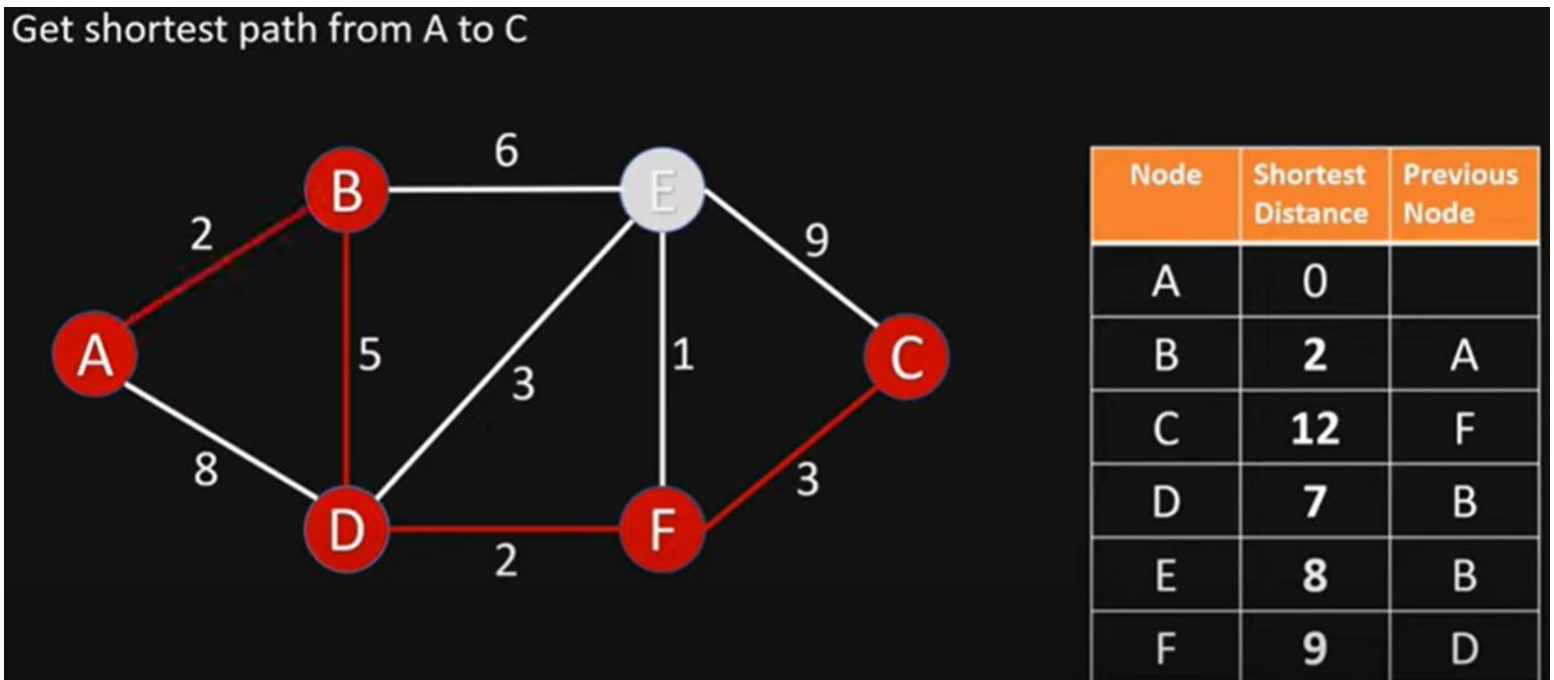
- Table contains the shortest distance to each node N from the source node A, and its previous node in the shortest path



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

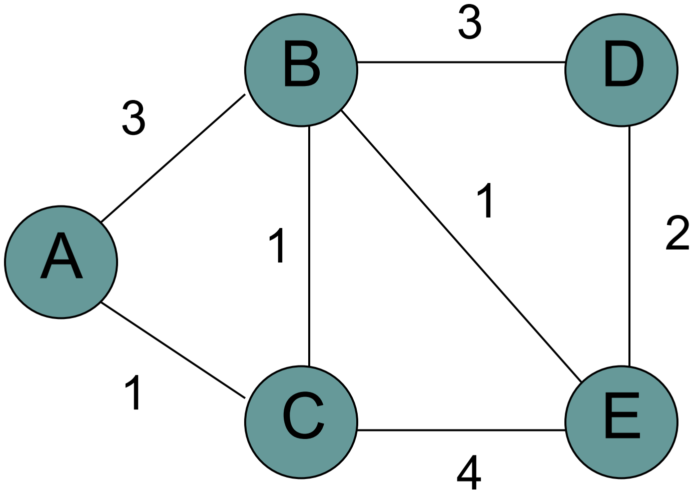
Getting the Shortest Path from A to C

- C's previous node is F; F's previous node is D; D's previous node is B; B's previous node is A
- Shortest Path from A to C is ABDFC

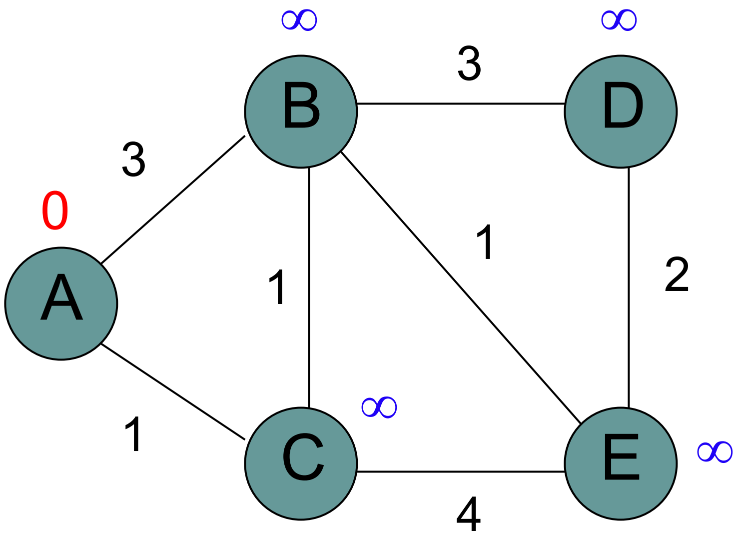


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Dijkstra’s Algorithm Example 2

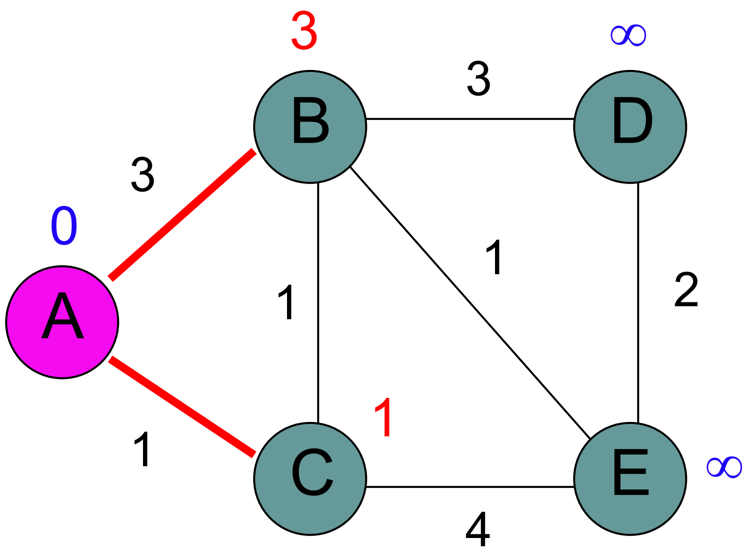


Initialize



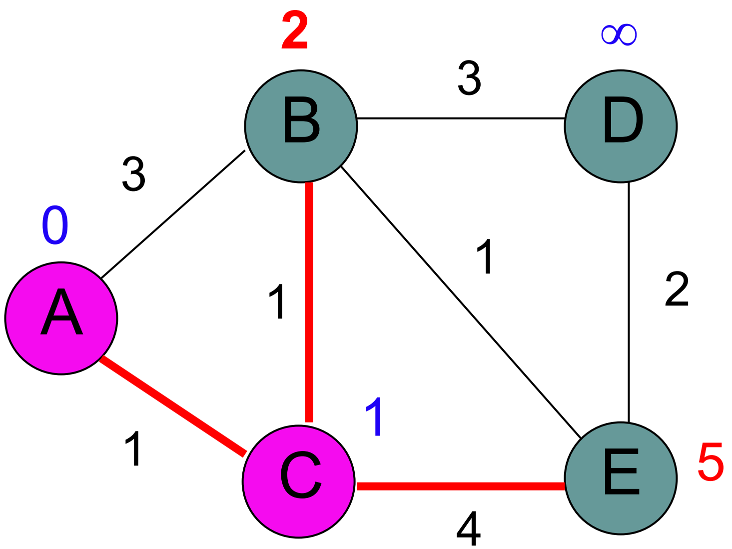
N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visit Node A



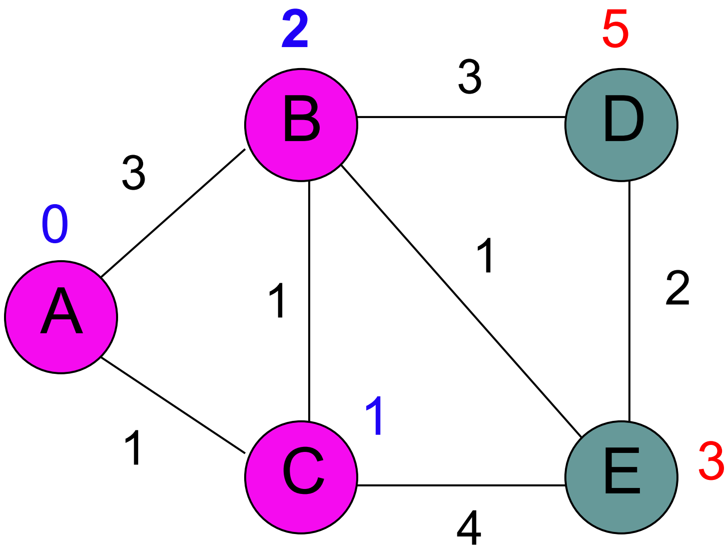
N	SD	PN
A	0	
B	3	A
C	1	A
D	∞	
E	∞	

Visit Node C



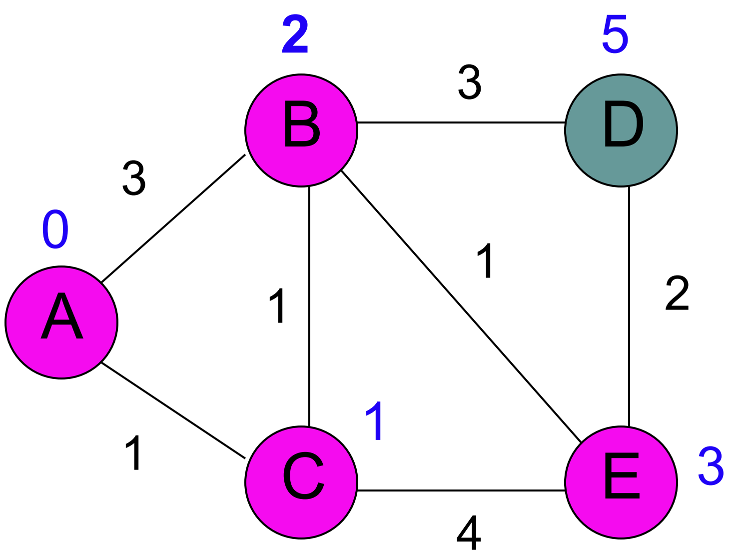
N	SD	PN
A	0	
B	2	C
C	1	A
D	∞	
E	5	C

Visit Node B



N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

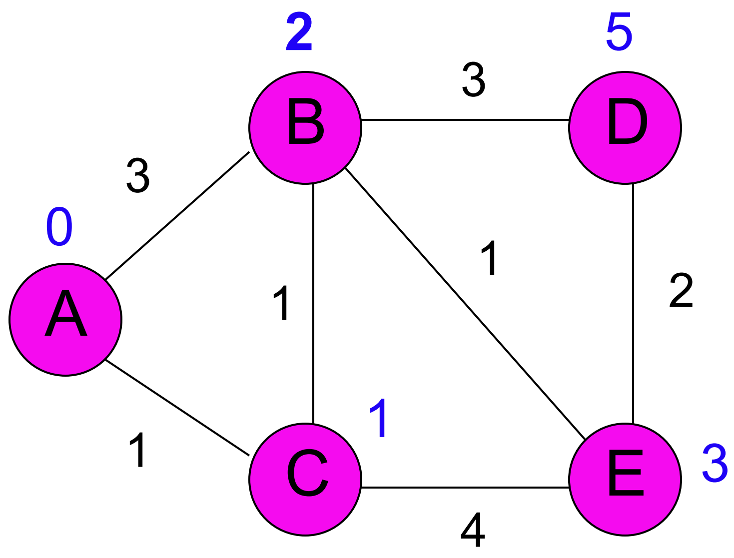
Visit Node E



N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

Nothing changes

Visit Node D



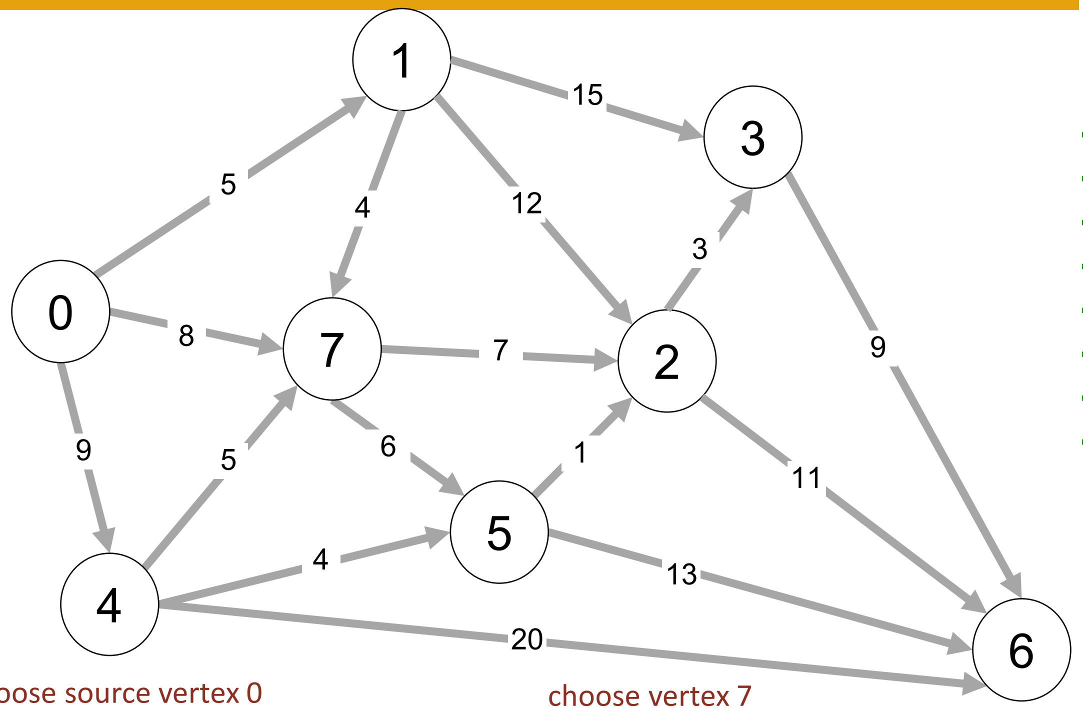
N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

Nothing changes

Dijkstra's Algorithm Example 3

choose vertex 5
relax all edges adjacent from 5
choose vertex 2
relax all edges adjacent from 2
choose vertex 3
relax all edges adjacent from 3
choose vertex 6
relax all edges adjacent from 6

- Consider vertices in increasing order of distance from s
  - (non-tree vertex with the lowest distTo[ ] value).
- Add vertex to tree and relax all edges pointing from that vertex.



choose source vertex 0
relax all edges adjacent from 0
choose vertex 1
relax all edges adjacent from 1

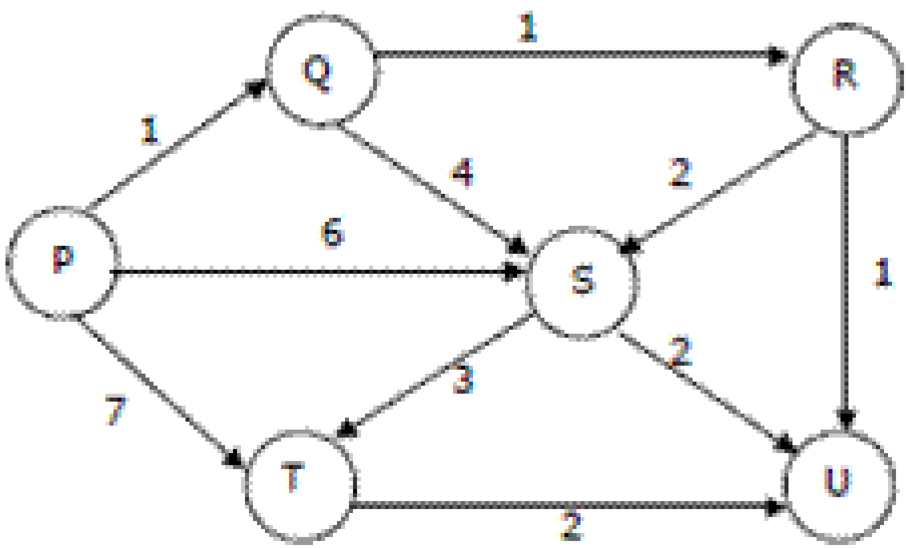
choose vertex 7
relax all edges adjacent from 7
choose vertex 4
relax all edges adjacent from 4

v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	~~15~~	14
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	~~14~~	13	
6	~~∞~~	~~29~~	~~26~~	25
7	~~∞~~	8		

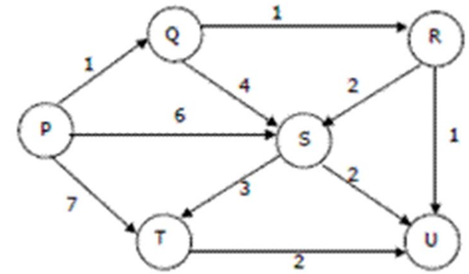
v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	~~7~~	4	
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

Quiz: Dijstra's Algorithm

- Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?
- ANS: P, Q, R, U, S, T



SD: Shortest Distance
PN: Previous Node



N	SD	PN
P	0	
Q	∞	
R	∞	
S	∞	
T	∞	
U	∞	

Visit P

N	SD	PN
P	0	
Q	1	P
R	∞	
S	6	P
T	7	P
U	∞	

Visit Q

N		
P	0	
Q	1	P
R	2	Q
S	5	Q
T	7	P
U	∞	

Visit R

P	0	
Q	1	P
R	2	Q
S	4	Q
T	7	P
U	3	R

Visit U (nothing changes)

N	SD	PN
P	0	
Q	1	P
R	2	Q
S	4	Q
T	7	P
U	3	R

Visit S (nothing changes)

N	SD	PN
P	0	
Q	1	P
R	2	Q
S	4	Q
T	7	P
U	3	R

Visit T (nothing changes)

N	SD	PN
P	0	
Q	1	P
R	2	Q
S	4	Q
T	7	P
U	3	R

Finished

N	SD	PN
P	0	
Q	1	P
R	2	Q
S	4	Q
T	7	P
U	3	R

Bellman-Ford Algorithm

- Initialize distance array distTo[] for each vertex **v** as **distTo[v] = ∞**, and **distTo[s] = 0** to source vertex s.
- Relax all **edges |V|-1** times.

```
private void relax(DirectedEdge e)
{
    int  v  = e.from(), w  = e.to();
    if  (distTo[w] >  distTo[v] + e.weight())
    {
        distTo[w] =  distTo[v] +
        e.weight();
        edgeTo[w] = e;
    }
}
```

Recall:

Generic algorithm (to compute SPT from s)

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat until done:

- Relax any edge.

Bellman–Ford algorithm

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat |V | -1 times:

- Relax each edge.

Bellman-Ford Algorithm Proof of Correctness

- Relaxing edges |V|-1 times in the Bellman-Ford Algorithm guarantees that the algorithm has explored all possible paths of length up to |V|-1, which is the maximum possible length of a shortest path in a graph with |V| vertices. This allows the algorithm to correctly calculate the shortest paths from the source vertex to all other vertices, given that there are no negative-weight cycles.

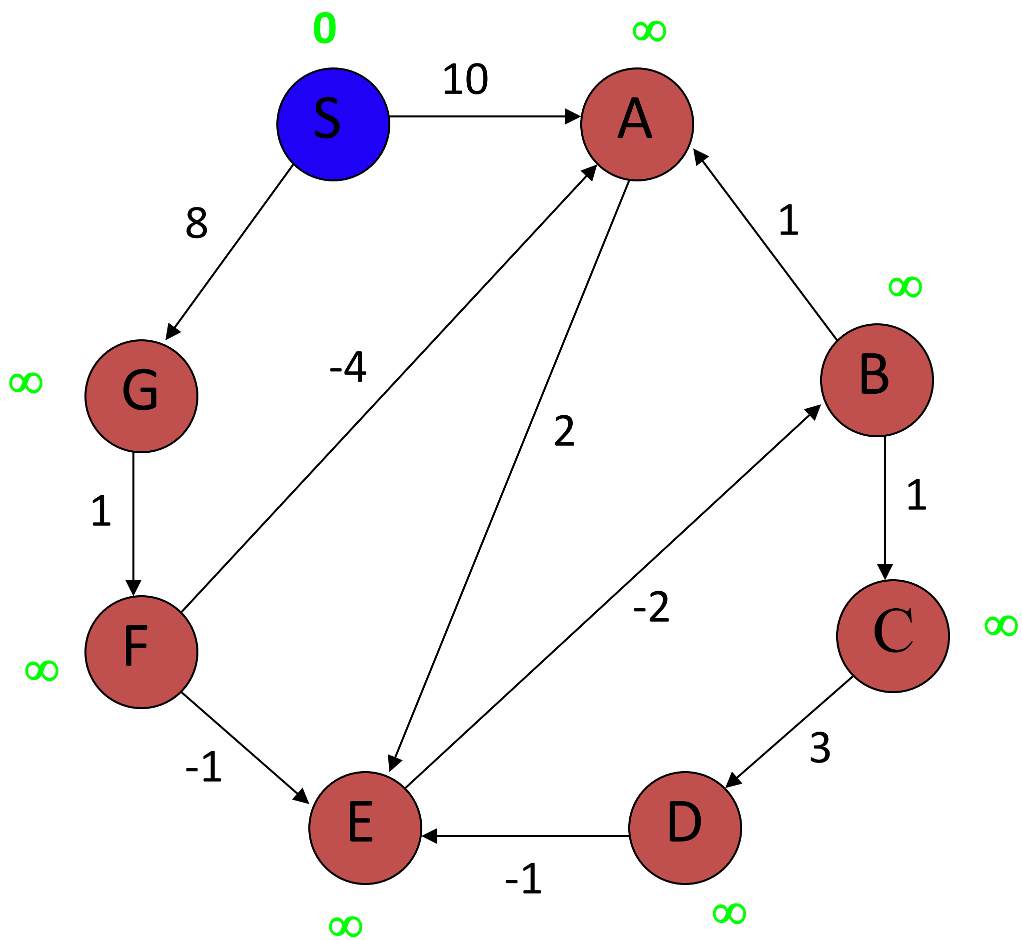
Bellman-Ford Algorithm with Negative Cycle Detection

- Initialize distance array distTo[] for each vertex **v** as **distTo[v] = ∞**, and **distTo[s] = 0** to source vertex s.
- Relax all **edges |V|-1** times.
- Relax all the edges one more time i.e. the **N-th** time:
  - Case 1 (Negative cycle exists): if any edge can be further relaxed, i.e., for any **edge e,** if **distTo[w] > distTo[v]** + **e.weight())**
  - Case 2 (No Negative cycle) : case 1 fails for all the edges.

Time Complexity of Bellman-Ford Algorithm

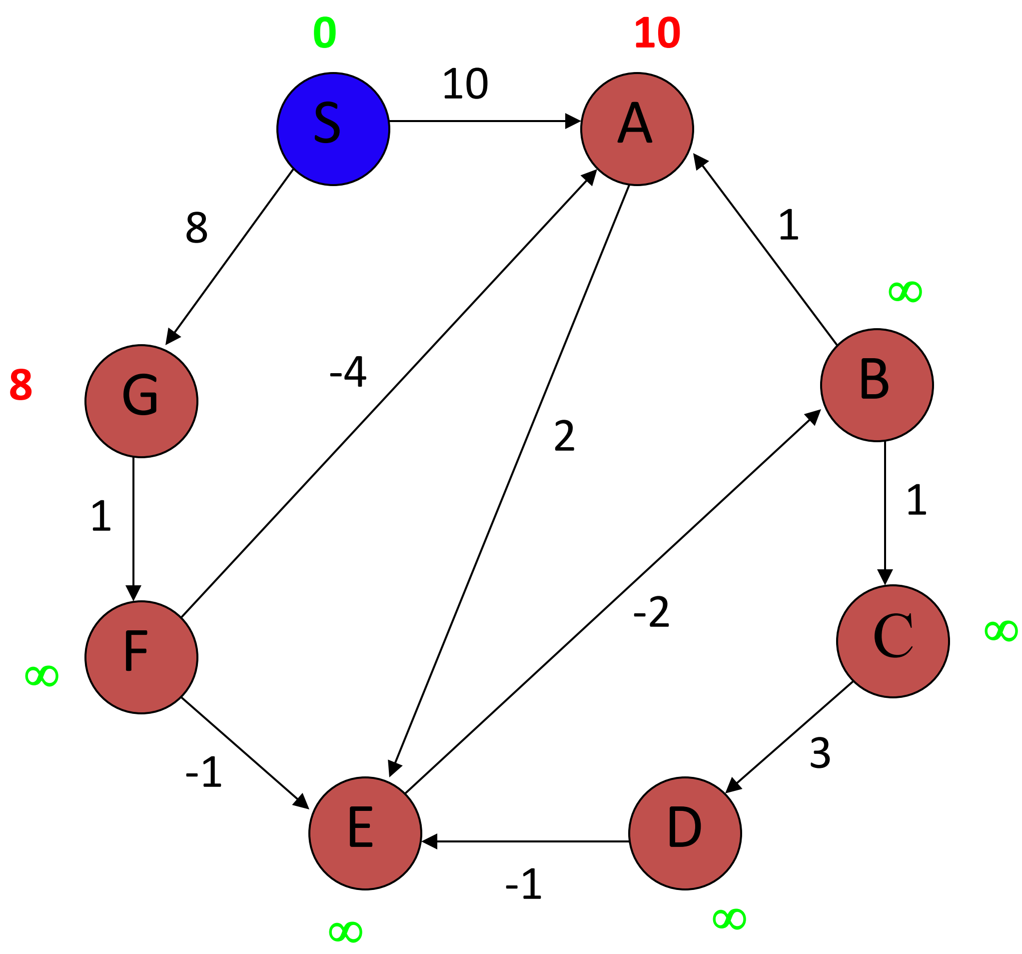
- Time complexity for connected graph:
- Best Case: O(|E|), when distance array after 1st and 2nd relaxation are same , we can simply stop further processing after one iteration
- Average Case: O(|V|*|E|)
- Worst Case: O(|V|*|E|)
  - If the graph is complete, the value of E becomes $O(|V|^2)$. So overall time complexity becomes $O(|V|^3)$

Bellman-Ford Algorithm Example 1



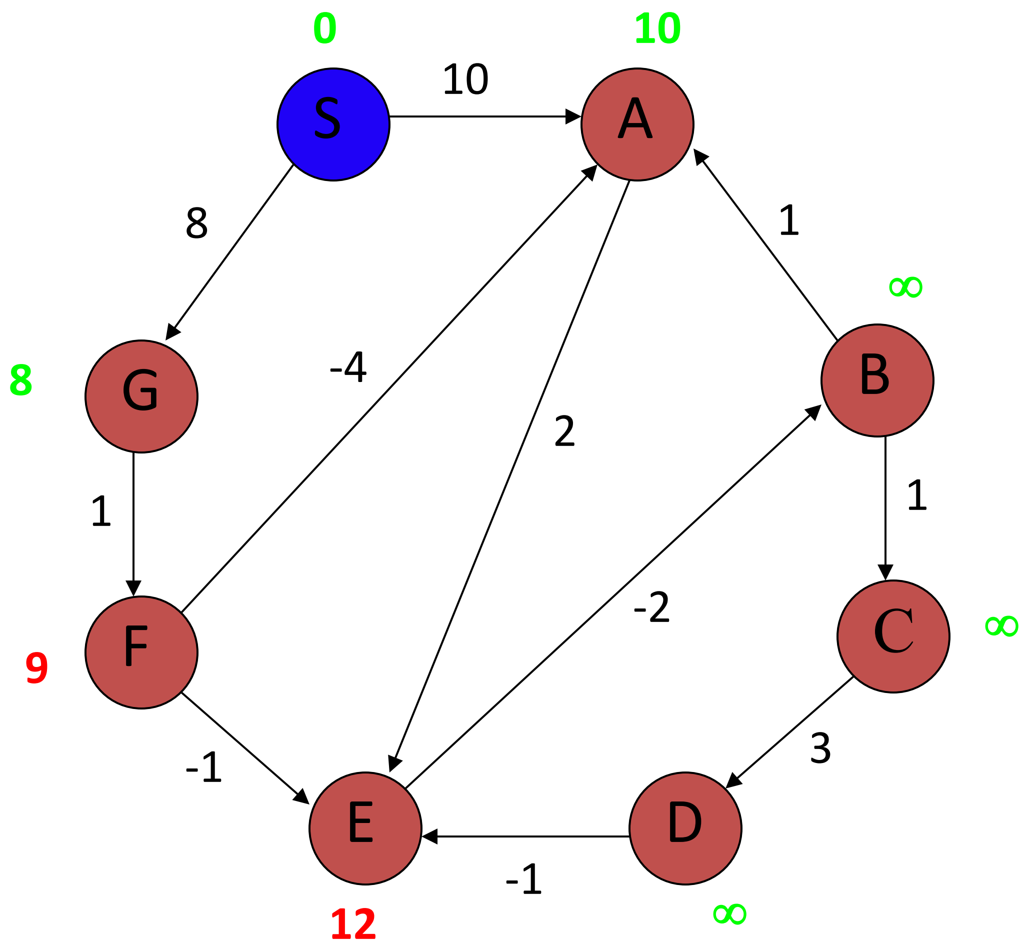
Iteration: 0

Bellman-Ford Algorithm Example 1



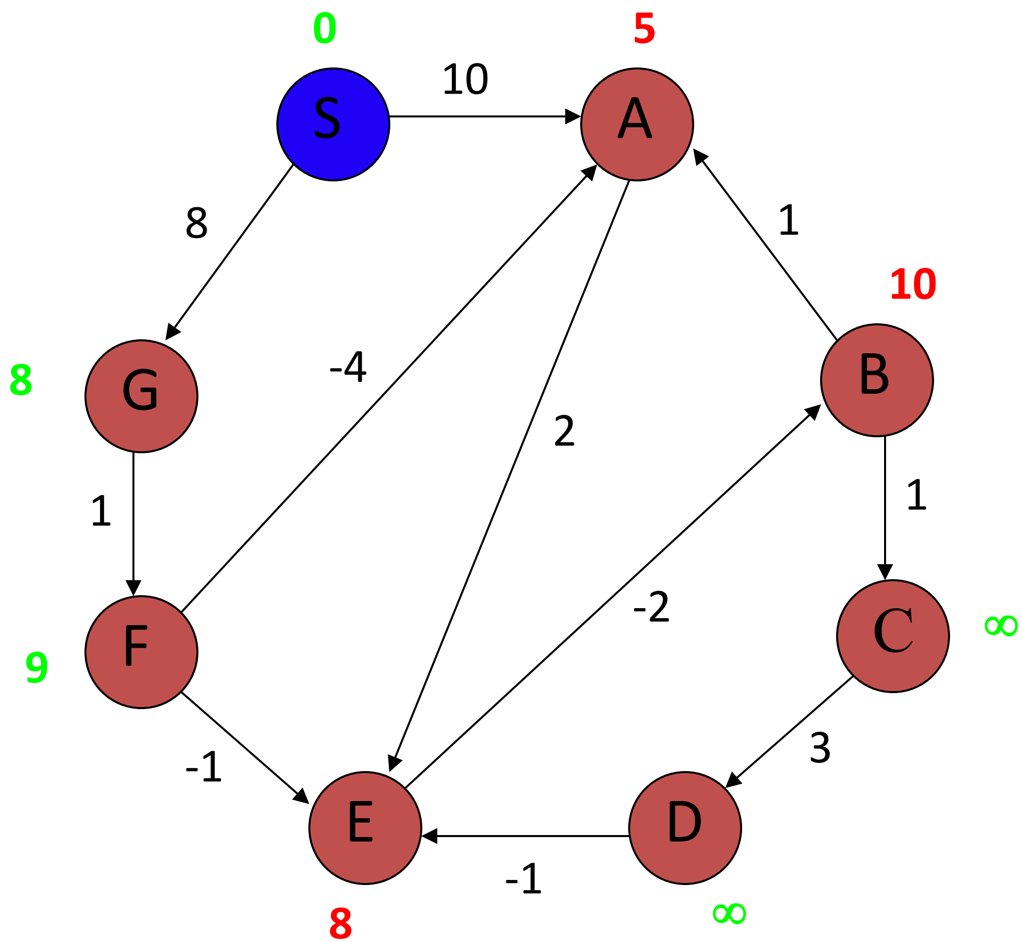
Iteration: 1

Bellman-Ford Algorithm Example 1



Iteration: 2

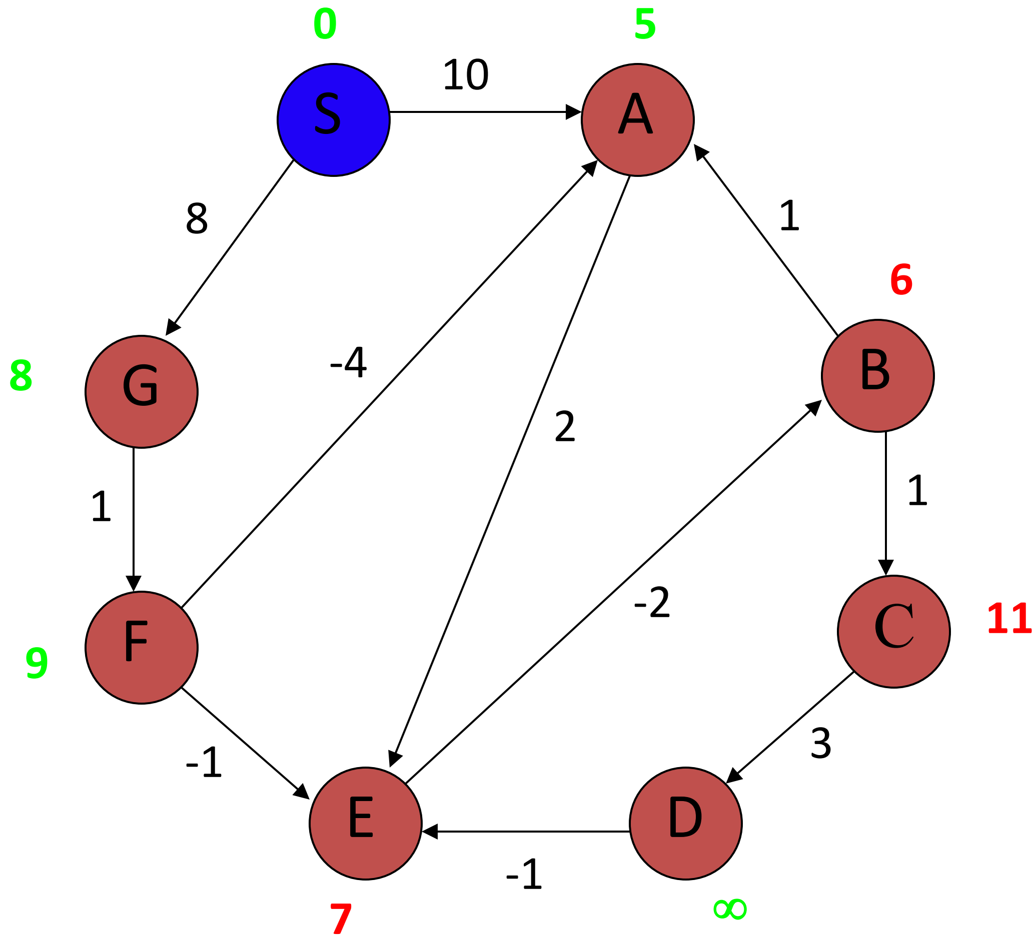
Bellman-Ford Algorithm Example 1



Iteration: 3

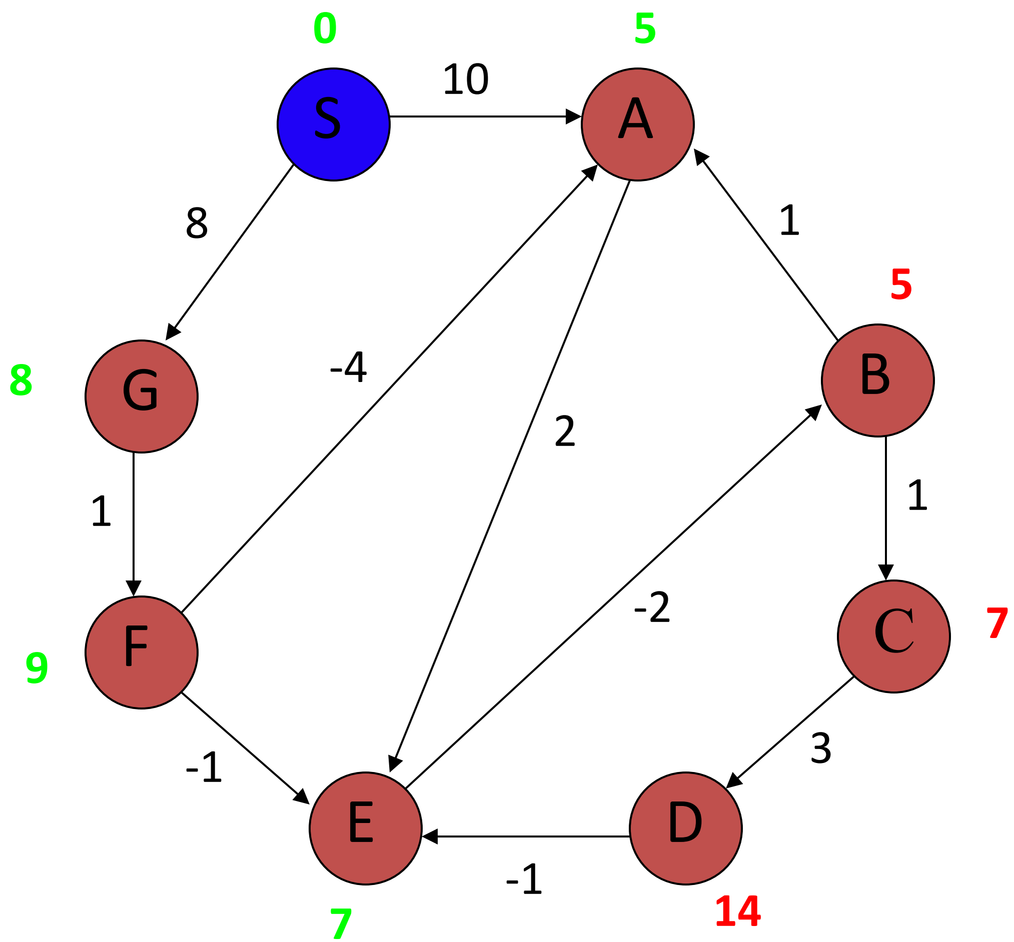
A has the correct distance and path

Bellman-Ford Algorithm Example 1



Iteration: 4

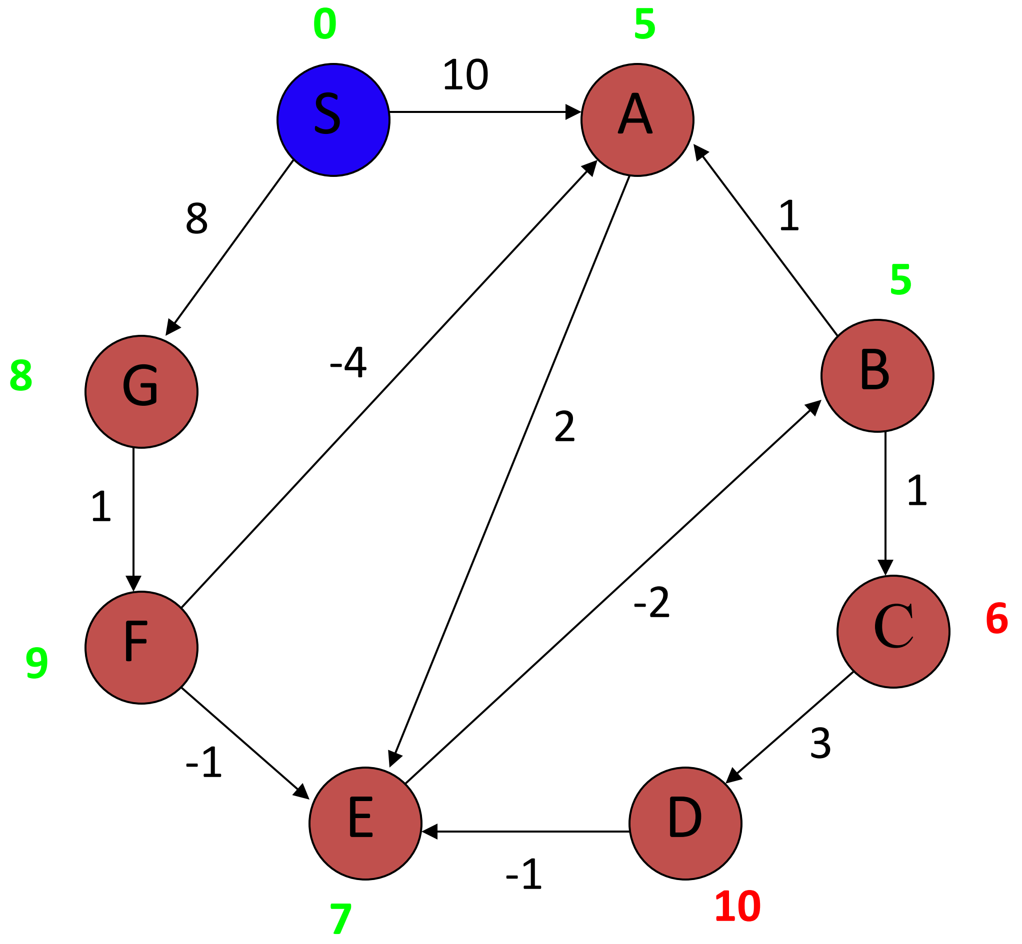
Bellman-Ford Algorithm Example 1



Iteration: 5

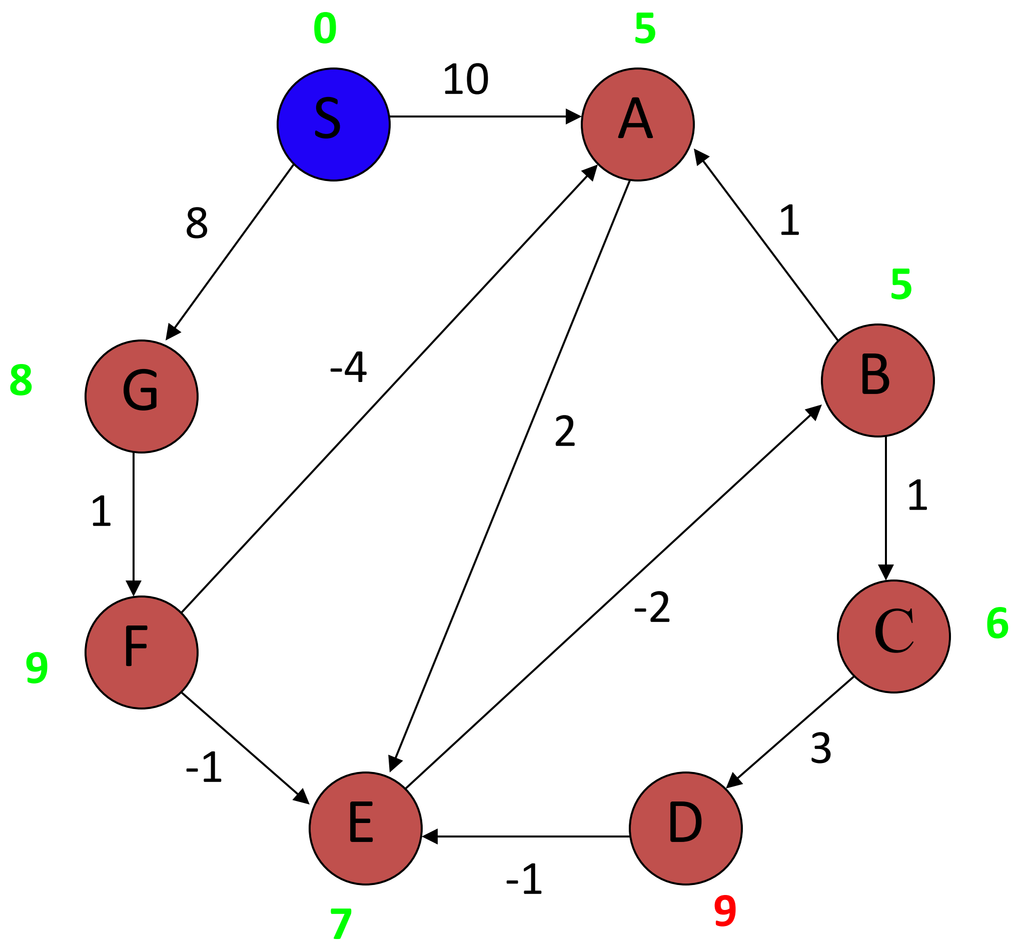
B has the correct distance and path

Bellman-Ford Algorithm Example 1



Iteration: 6

Bellman-Ford Algorithm Example 1

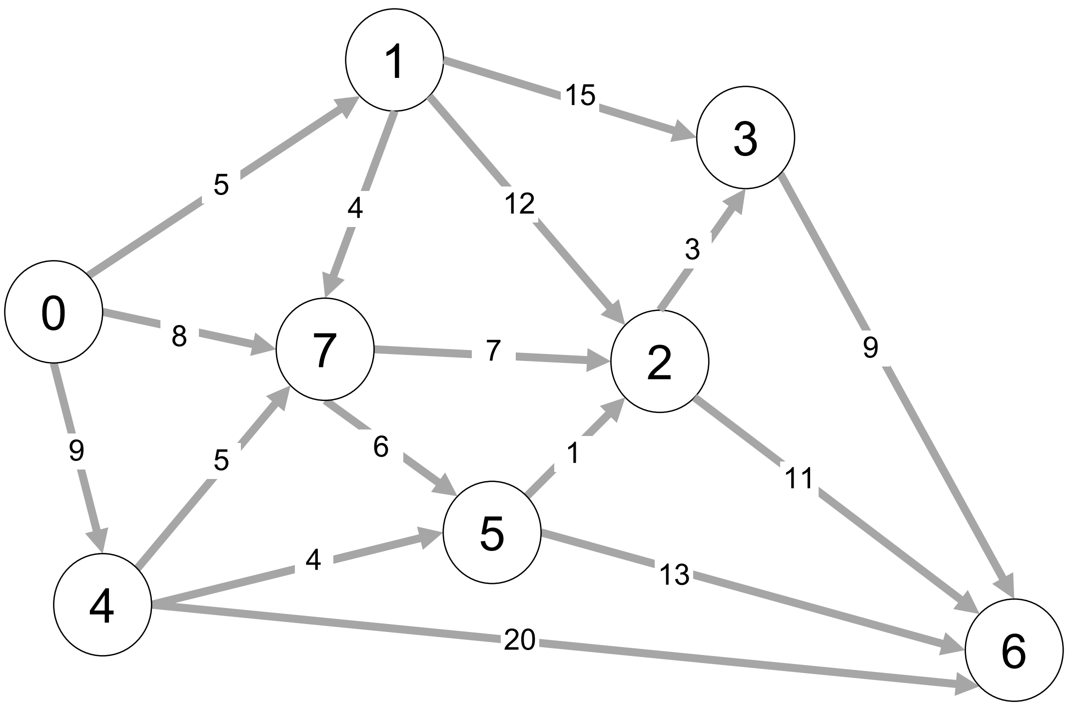


Iteration: 7

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

Bellman-Ford Algorithm Example 2

Repeat V − 1 times: relax all E edges.



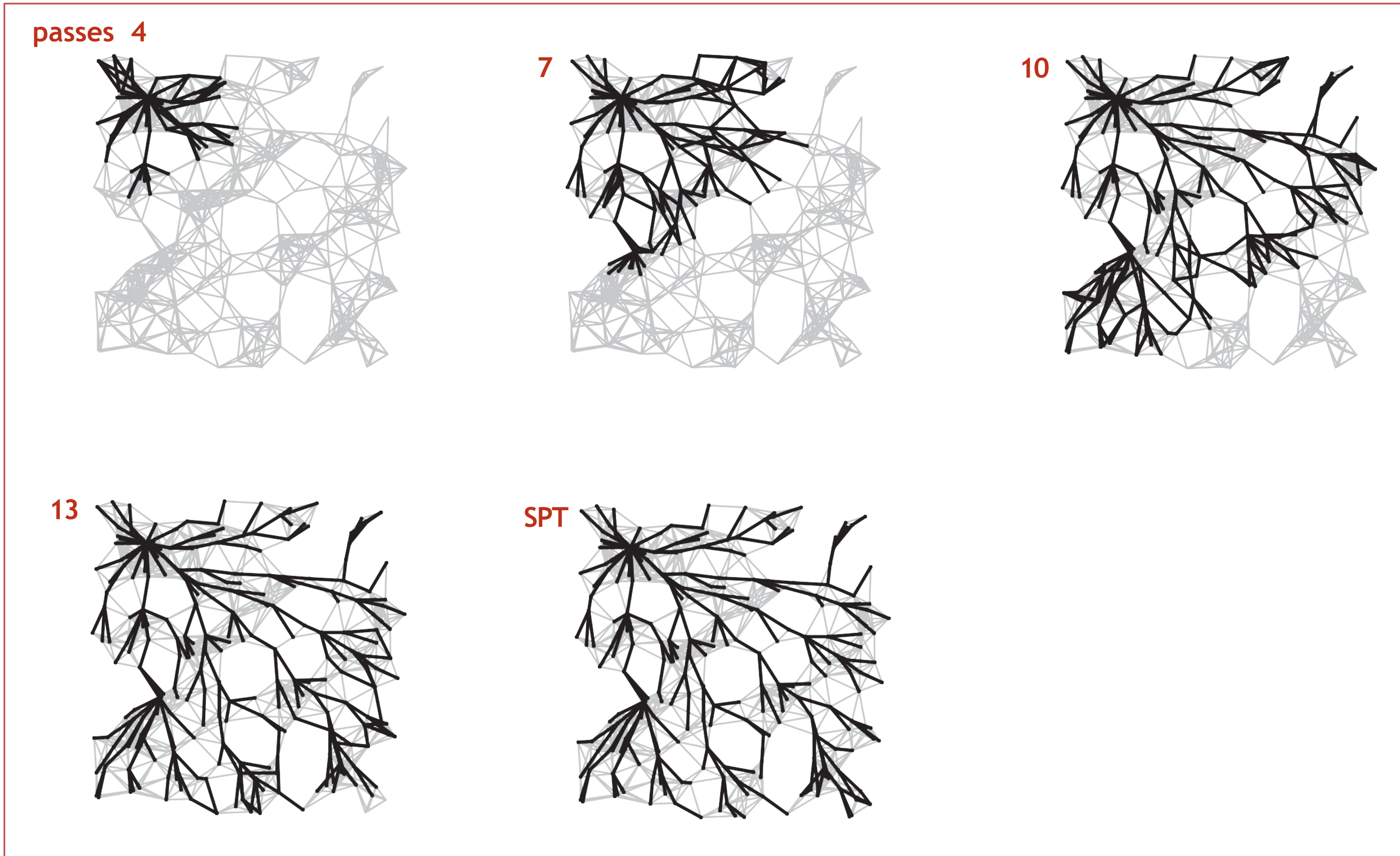
v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	14	
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~28~~	~~26~~	25
7	~~∞~~	8		

v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	5	
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	4		
6	~~-~~	~~2~~	~~5~~	2
7	~~-~~	0		

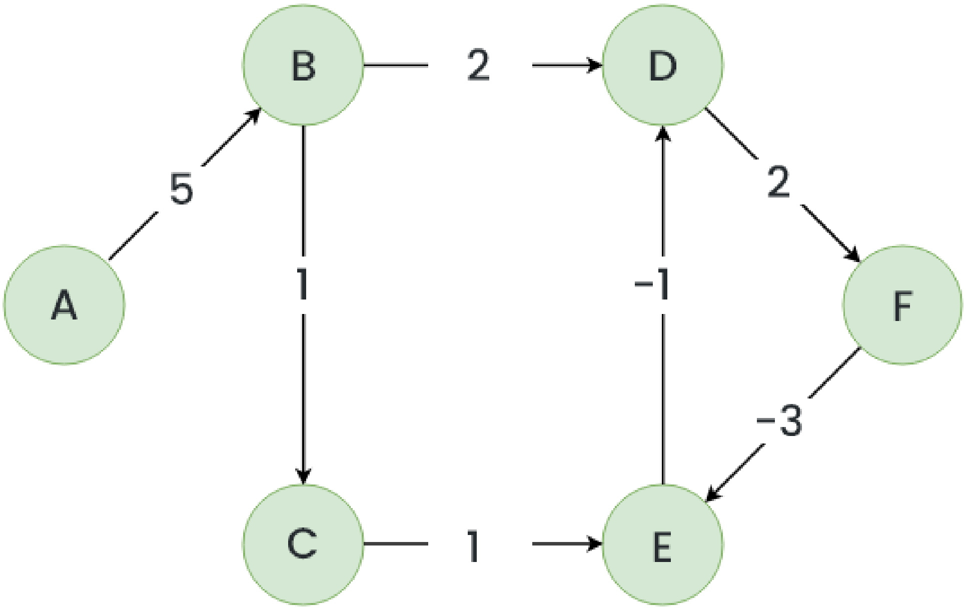
pass 1 pass 2 pass 3 (no further changes) pass 4-7 (no further changes)

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→2 7→5

Bellman-Ford Algorithm Visualization

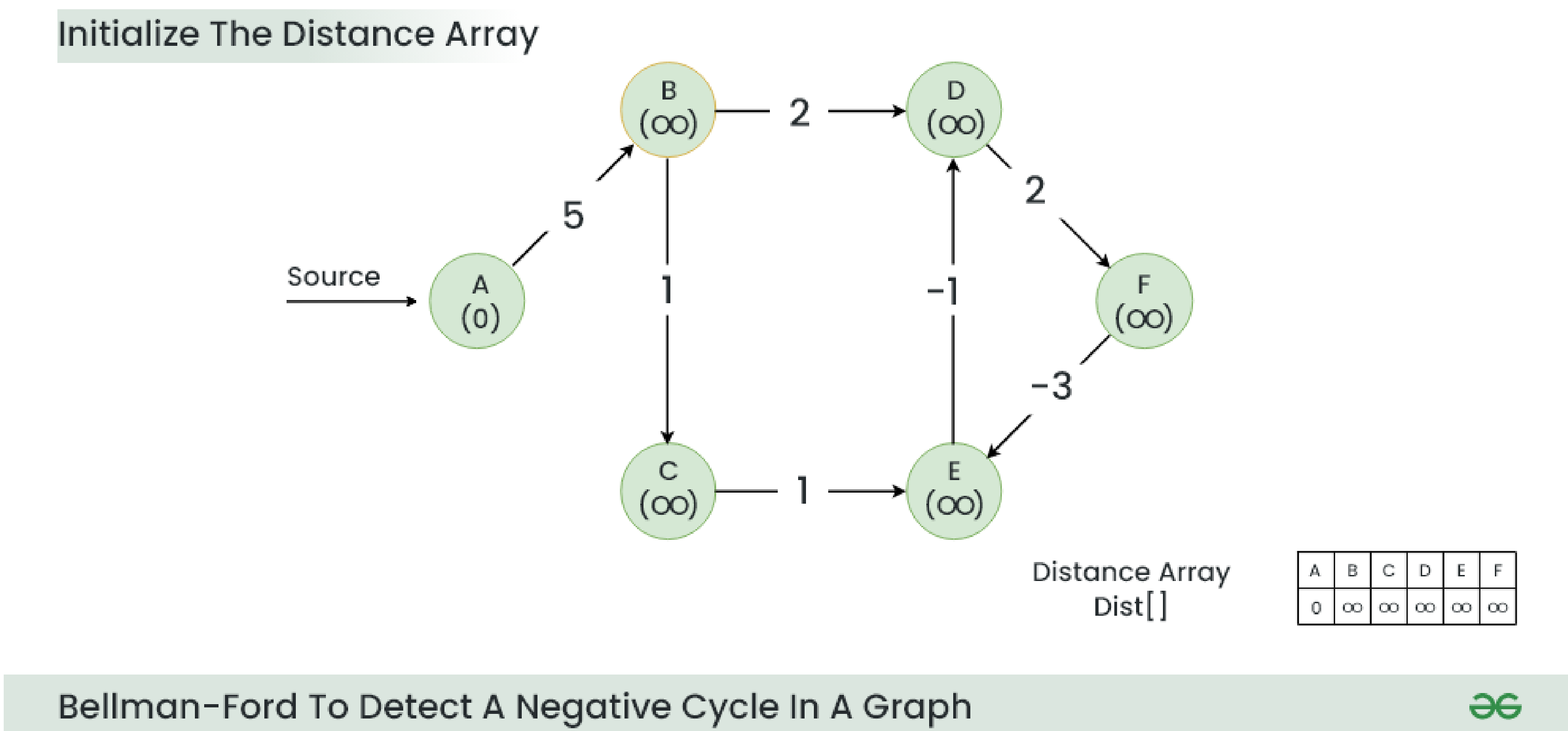


Bellman-Ford Algorithm Example 3 w. Negative Cycle

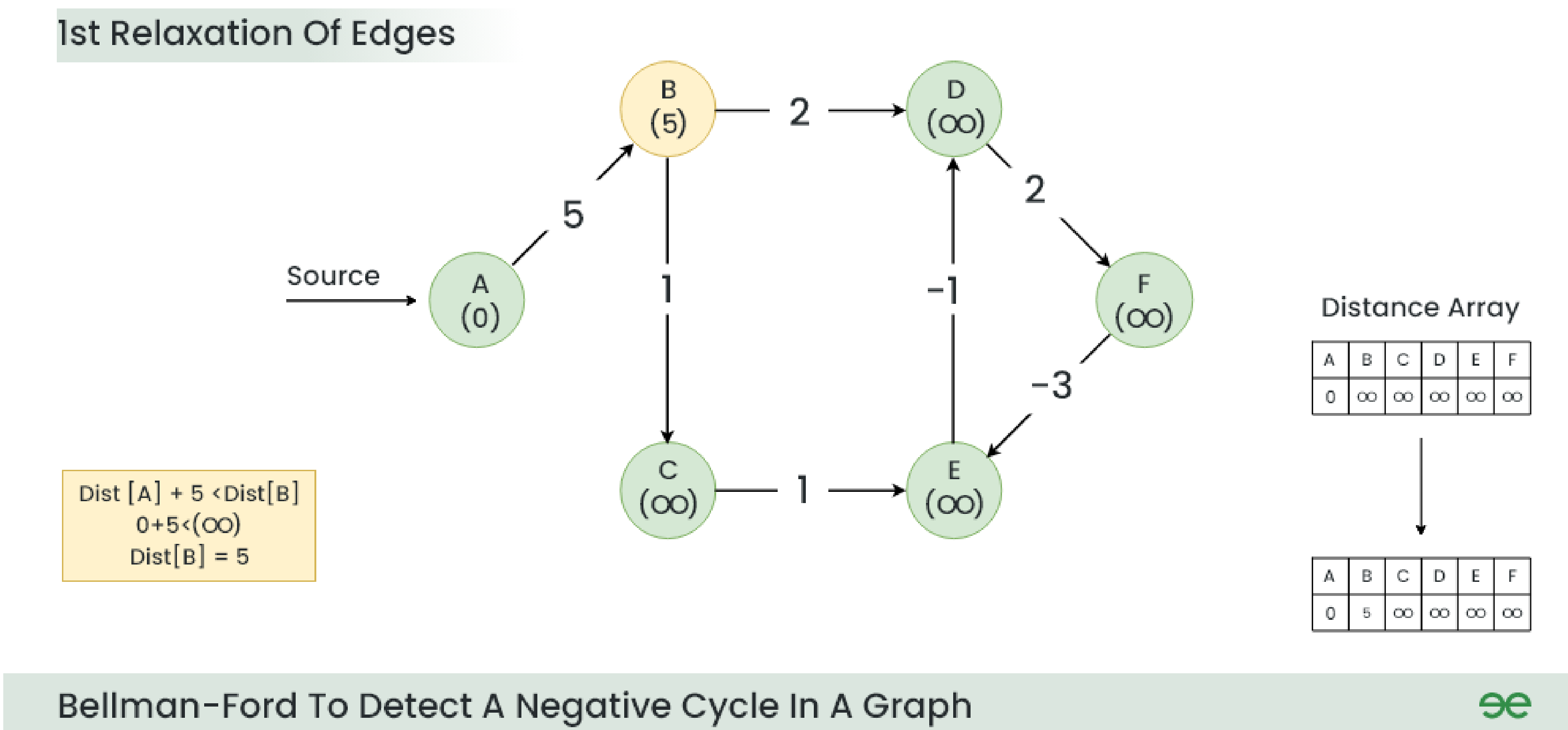


Bellman-Ford To Detect A Negative Cycle In A Graph

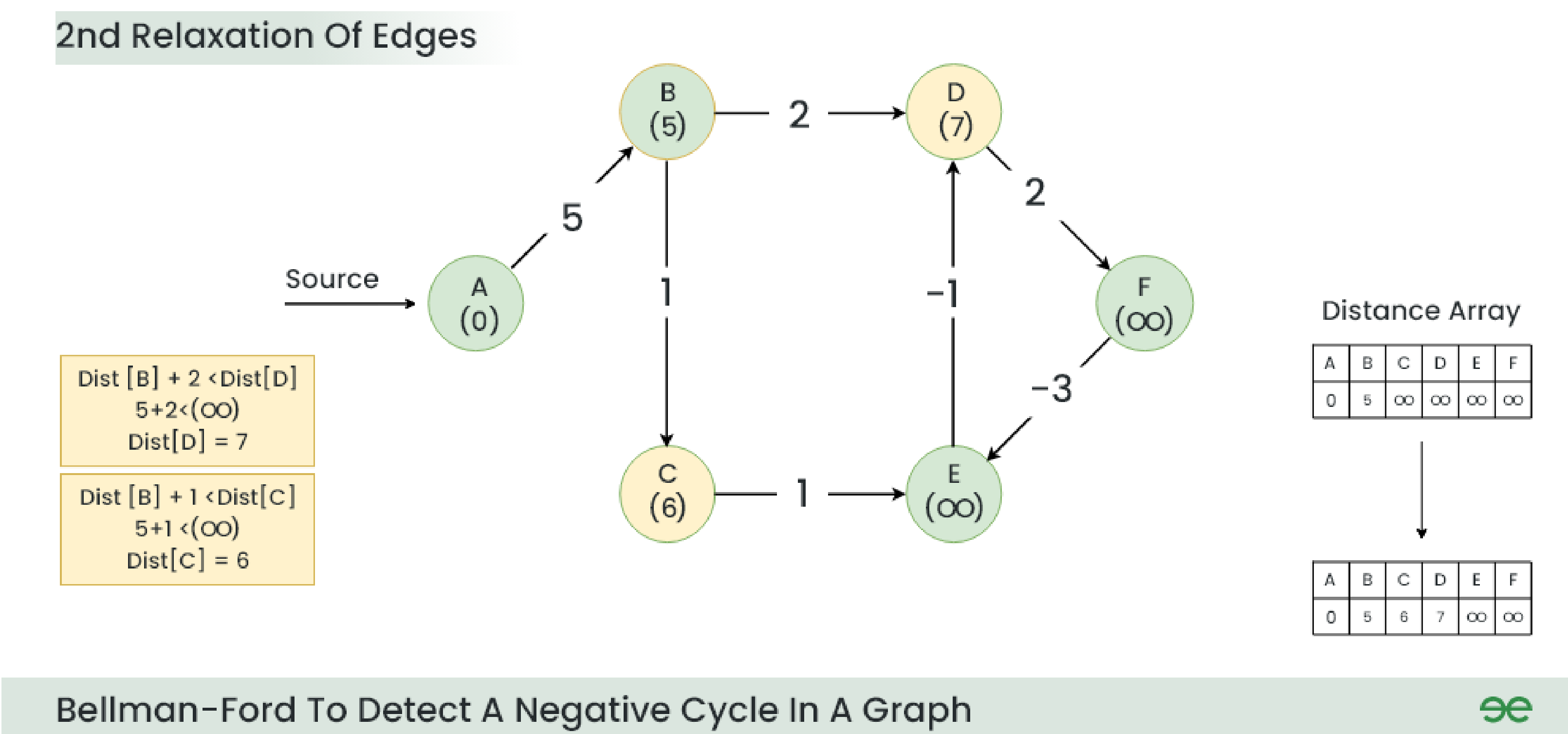
- Step 1: Initialize a distance array Dist[] to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY. distTo[N] = ∞



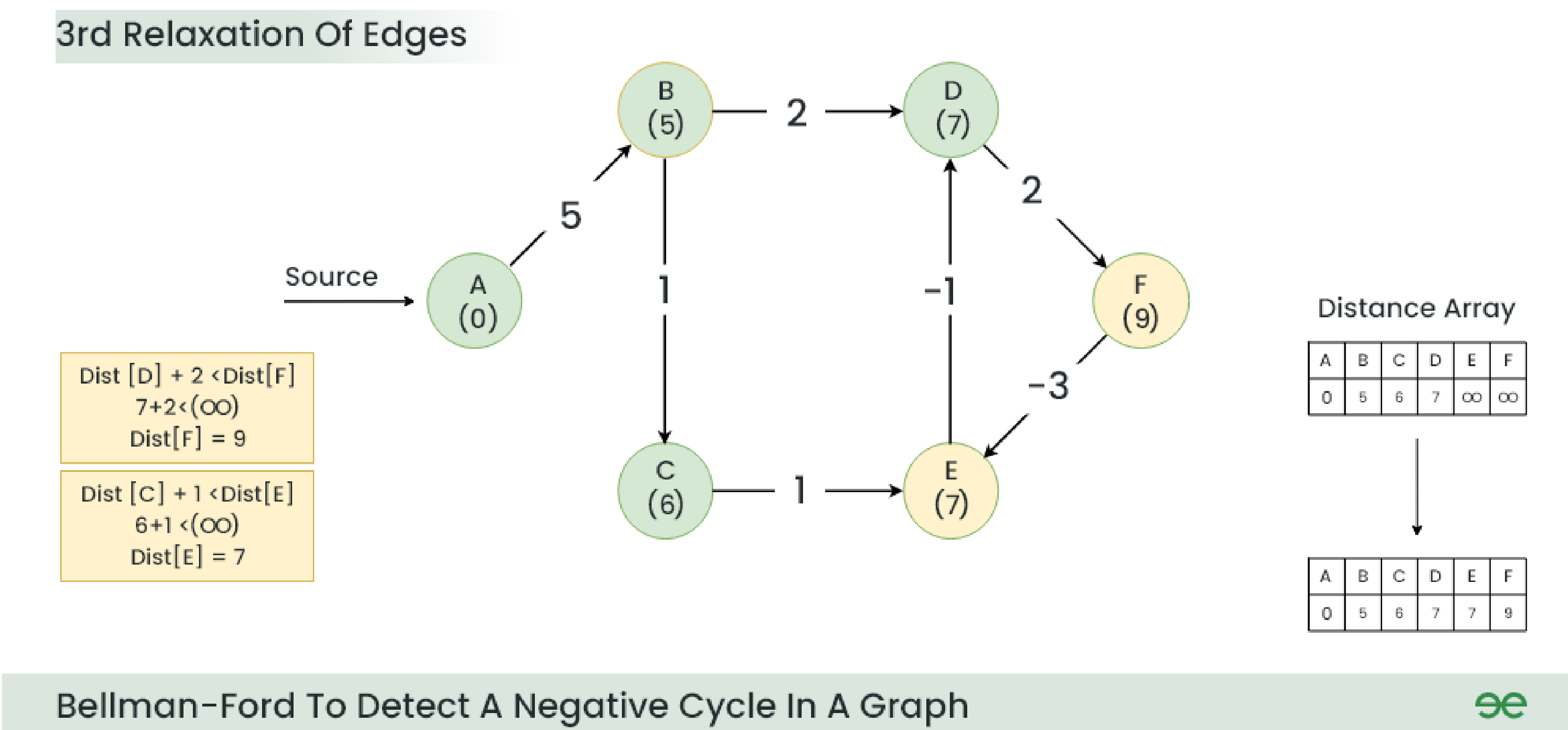
- Step 2: Start relaxing the edges, during 1st relaxation:
- OLD distTo[B] = ∞ > distTo[A] + e[A][B].weight() = 0+5 = 5
- NEW distTo[B] = distTo[A] + e[A][B].weight() = 5



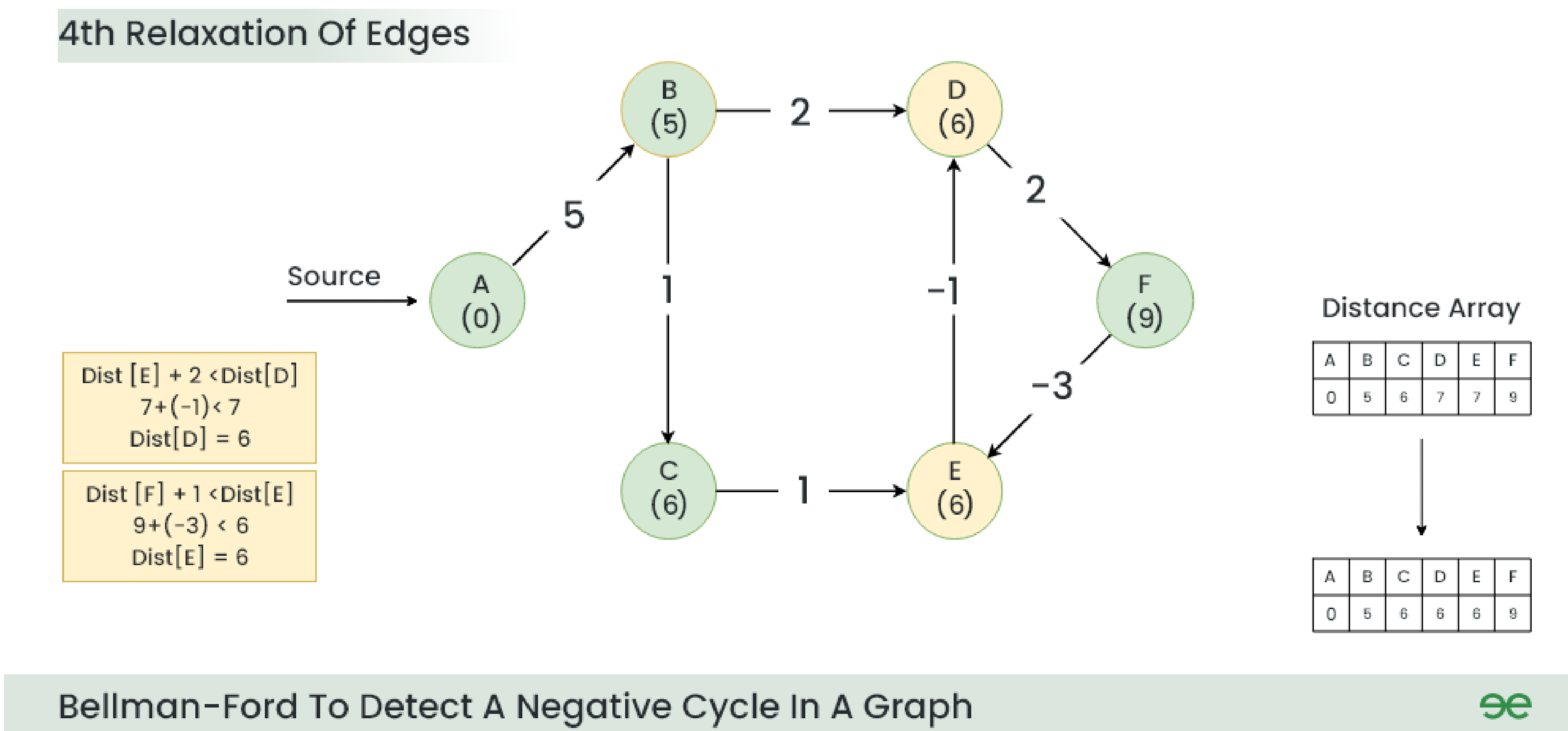
- Step 3: During 2nd relaxation:
- OLD distTo[D] = ∞ > distTo[B] + e[B][D].weight() = 5+2 = 7
- NEW distTo[D] ← distTo[B] + e[B][D].weight() = 7
- OLD distTo[C] = ∞ > distTo[B] + e[B][C].weight() = 5+1 = 6
- NEW distTo[C] ← distTo[B] + e[B][C].weight() = 6



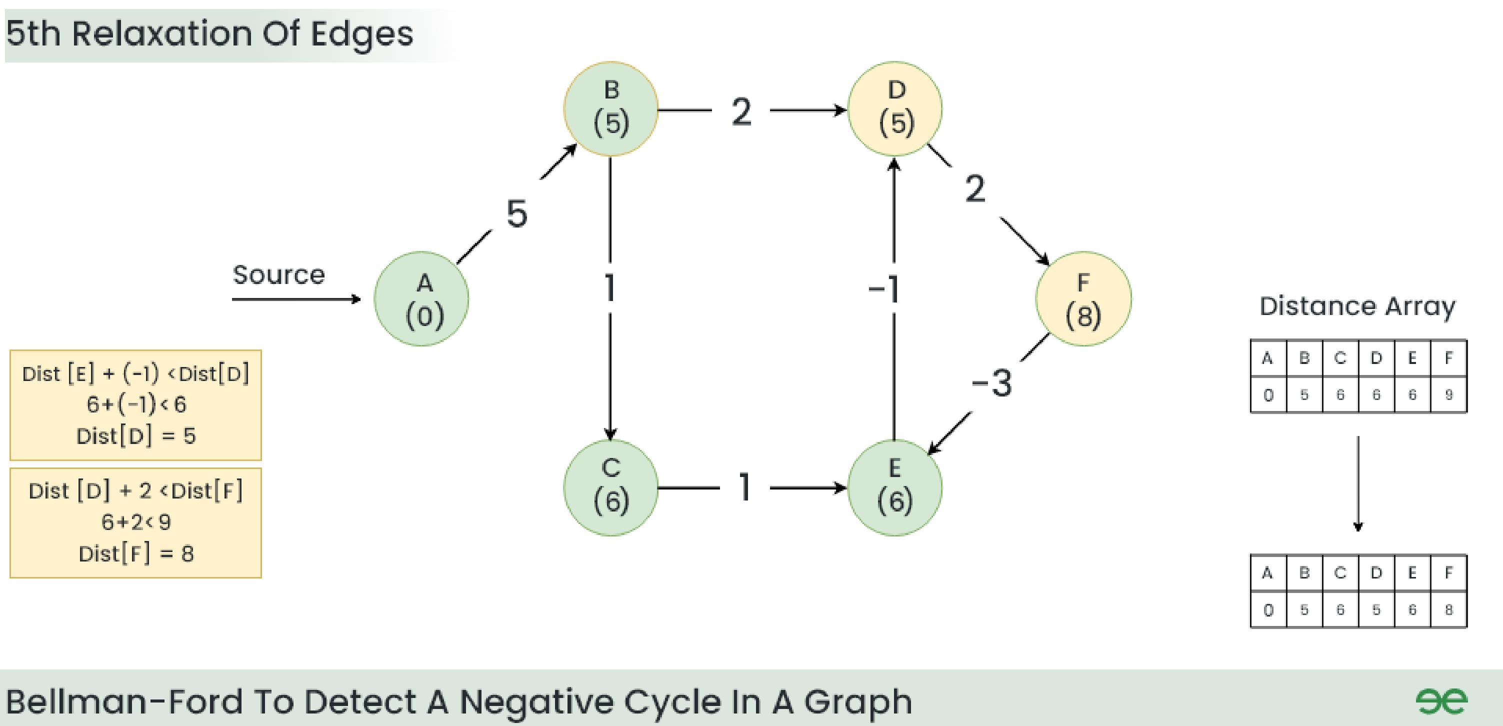
- Step 4: During 3rd relaxation:
- OLD distTo[F] = ∞ > distTo[D] + e[D][F].weight() = 7+2 = 9
- NEW distTo[F] ← distTo[D] + e[D][F].weight() = 9
- OLD distTo[E] = ∞ > distTo[C] + e[C][E].weight() = 6+1 = 7
- NEW distTo[E] ← distTo[C] + e[C][E].weight() = 7



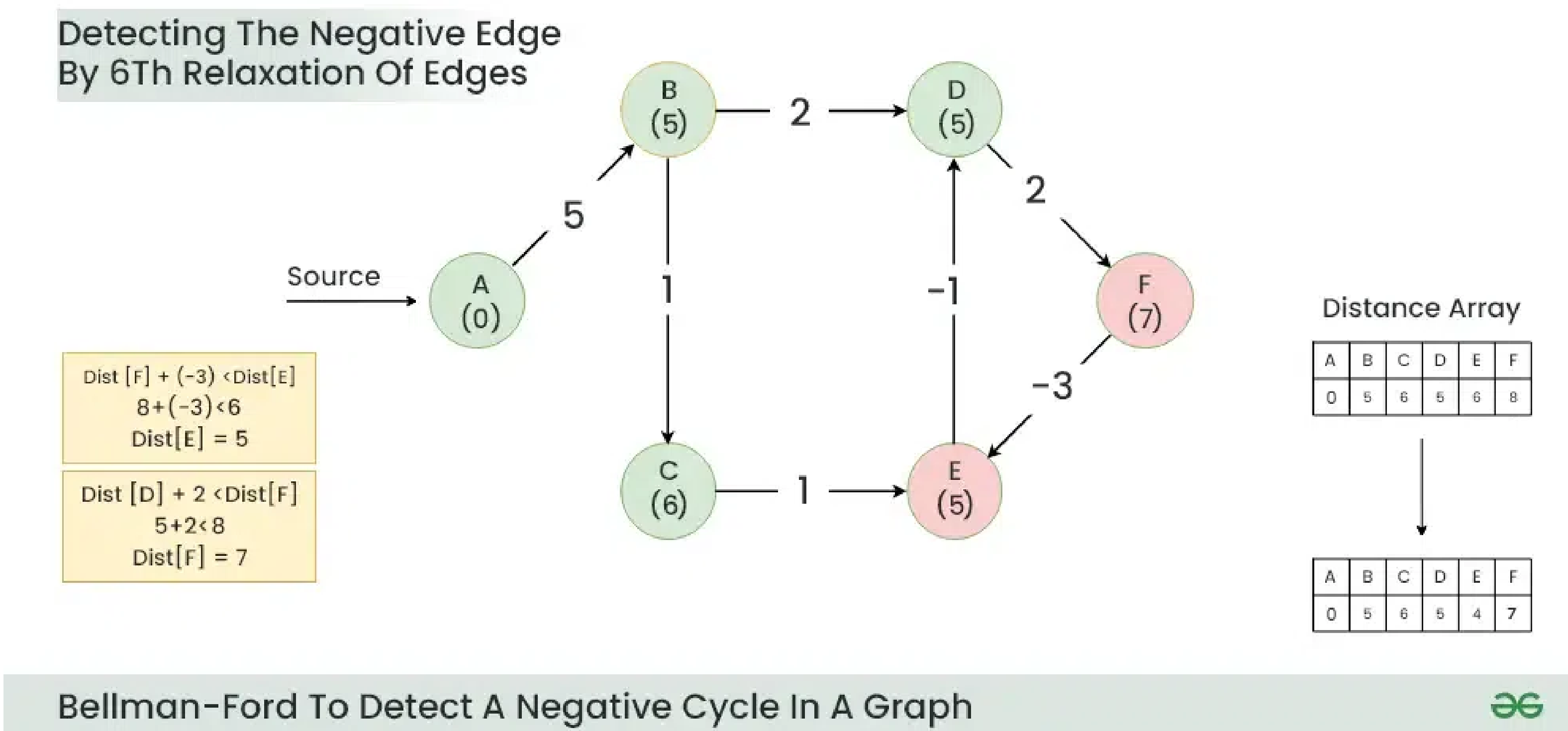
- Step 5: During 4th relaxation:
- OLD distTo[D] = 7 > distTo[E] + e[E][D].weight() = 7-1 = 6
- NEW distTo[D] ← distTo[E] + e[E][D].weight() = 6
- OLD distTo[E] = 7 > distTo[F] + e[F][E].weight() = 9-3 = 6
- NEW distTo[E] ← distTo[F] + e[F][E].weight() = 6



- Step 6: During 5th relaxation:
- OLD distTo[F] = 9 > distTo[D] + e[D][F].weight() = 6+2 = 8
- NEW distTo[D] ← distTo[D] + e[D][F].weight() = 8
- OLD distTo[D] = 6 > distTo[E] + e[E][D].weight() = 6-1 = 5
- NEW distTo[E] ← distTo[E] + e[E][D].weight() = 5
- Since the graph h 6 vertices, So during the 5th relaxation the shortest distance for all the vertices should have been calculated.



- Step 7: Now the final relaxation i.e. the 6th relaxation should indicate the presence of negative cycle if there is any changes in the distance array of 5th relaxation.
- During the 6th relaxation, following changes can be seen:
- OLD distTo[E] = 6 > distTo[F] + e[F][E].weight() = 8-3 = 5
- NEW distTo[D] ← distTo[F] + e[F][E].weight() = 5
- OLD distTo[F] = 8 > distTo[D] + e[D][F].weight() = 5+2 = 7
- NEW distTo[E] ← distTo[D] + e[D][F].weight() = 7
- Since, we observer changes in the Distance array. Hence ,we can conclude the presence of a negative cycle in the graph (D->F->E).



Dijkstra's Algorithm vs. Bellman-Ford Algorithm

- Dijkstra's Algorithm:
  - Uses a priority queue to select the next vertex to process.
  - Greedily selects the vertex with the smallest tentative distance to source node.
  - Works only on graphs with non-negative edge weights.
  - Time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph.
- Bellman-Ford Algorithm:
  - Iteratively relaxes all edges V-1 times, where V is the number of vertices.
  - Does not use a priority queue.
  - Can handle graphs with negative edge weights, and can detect negative cycles.
  - Time complexity of O(VE), where V is the number of vertices and E is the number of edges in the graph.
- Dijkstra's algorithm is faster and more efficient for graphs with non-negative weights, the Bellman-Ford Algorithm Example 1 is more versatile as it can handle negative weights and detect negative cycles, albeit at the cost of lower efficiency.

Dijkstra's Algorithm does not work for Negative Edge Weights

- Dijkstra's Algorithm is greedy and optimal: any node that has been visited should have its shortest distance to the source. After visiting A, C, D, we have got D's shortest distance to A is 2, but after visiting D, D's distance to A is updated to -10, which violates the greedy optimal assumption of Dijkstra's Algorithm.

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	

Visit A

N	SD	PN
A	0	
B	10	A
C	1	A
D	∞	

Visit C

N	SD	PN
A	0	
B	10	A
C	1	A
D	2	C

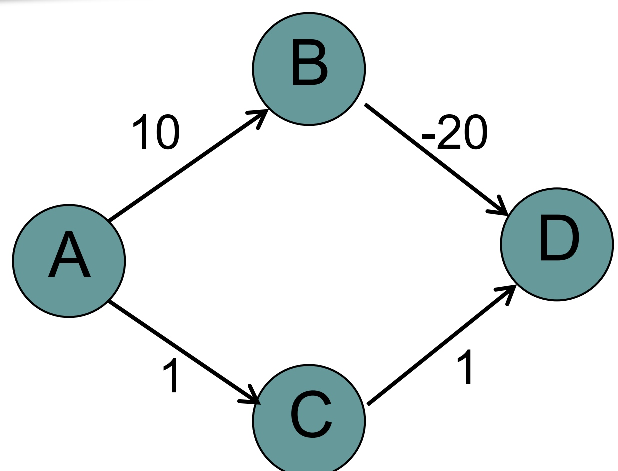
Visit D

N	SD	PN
A	0	
B	10	A
C	1	A
D	2	C

Visit B

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B

Incorrect



Bellman Ford Algorithm works for Negative Edge Weights

- We run for V-1=3 iterations, then run one more iteration with no change. Hence we conclude that The Bellman-Ford algorithm successfully calculated the shortest paths from node A to all other nodes. The shortest path from node A to node D goes through node B with a total cost of -10. There are no negative weight cycles.

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	

Iter 1

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B

Iter 2

No change

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B

Iter 3

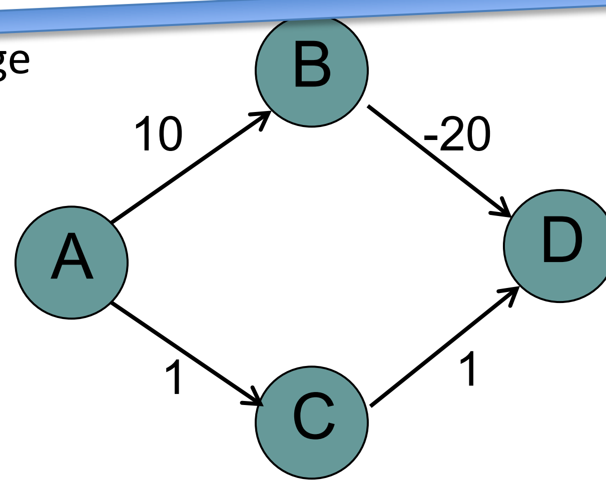
No change

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B

Iter 4

No change

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B



Quiz

- Which of the following algorithm can be used to efficiently calculate single source shortest paths in a Directed Acyclic Graph?
  - Dijkstra
  - Bellman-Ford
  - Topological Sort
  - Strongly Connected Component
- ANS: Topological Sort
- Using Topological Sort, we can find single source shortest paths in O(V+E) time which is the most efficient algorithm

Quiz

- Given a graph where all edges have positive weights, the shortest paths produced by Dijsktra and Bellman Ford algorithm may be different but path weight would always be same.
- ANS: True
- Dijkstra and Bellman-Ford both work fine for a graph with all positive weights, but they are different algorithms and may pick different edges for shortest paths.

Quiz

- Match the following
  - Group A
  - a) Dijkstra's single shortest path algo
  - b) Bellmen Ford's single shortest path algo
  - c) Floyd Warshall's all pair shortest path algo
  - Group B
  - p) Dynamic Programming
  - q) Backtracking
  - r) Greedy Algorithm
- Dijkstra is a greedy algorithm where we pick the minimum distant vertex from not yet finalized vertices. Bellman Ford and Floyd Warshall both are Dynamic Programming algorithms where we build the shortest paths in bottom up manner.

Quiz

- Let G be a directed graph whose vertex set is the set of numbers from 1 to 100. There is an edge from a vertex i to a vertex j if either $j = i + 1$ or $j = 3i$. The minimum number of edges in a path in G from vertex 1 to vertex 100 is
- A. 4 B. 7 C. 23 D. 99
- ANS: 7
- The task is to find minimum number of edges in a path in G from vertex 1 to vertex 100 such that we can move to either i+1 or 3i from a vertex i.
- Since the task is to minimize number of edges, we would prefer to follow 3*i. Let us follow multiple of 3. 1 => 3 => 9 => 27 => 81, now we can't follow multiple of 3 anymore. So we will have to follow i+1. This solution gives a long path.
- What if we begin from end, and we reduce by 1 if the value is not multiple of 3, else we divide by 3. 100 => 99 => 33 => 11 => 10 => 9 => 3 => 1
- So we need total 7 edges.