# Lecture 11
# Shortest Paths

Department of Computer Science
Hofstra University

# Lecture Goals

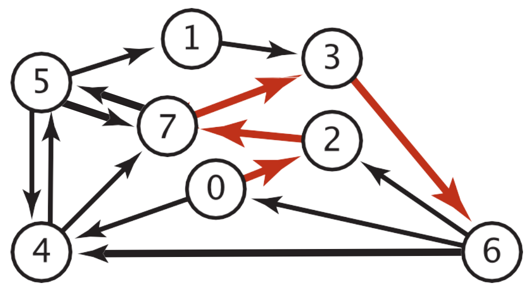
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- For single-source shortest path, we consider:
  - Dijkstra's algorithm for shortest-paths problems with nonnegative weights.
  - Topological Sort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

# 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 |



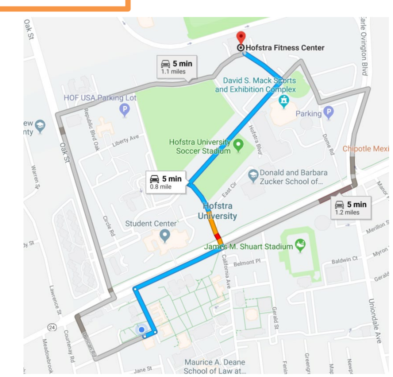

shortest path from 0 to 6

| | |
|---|---|
| 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.

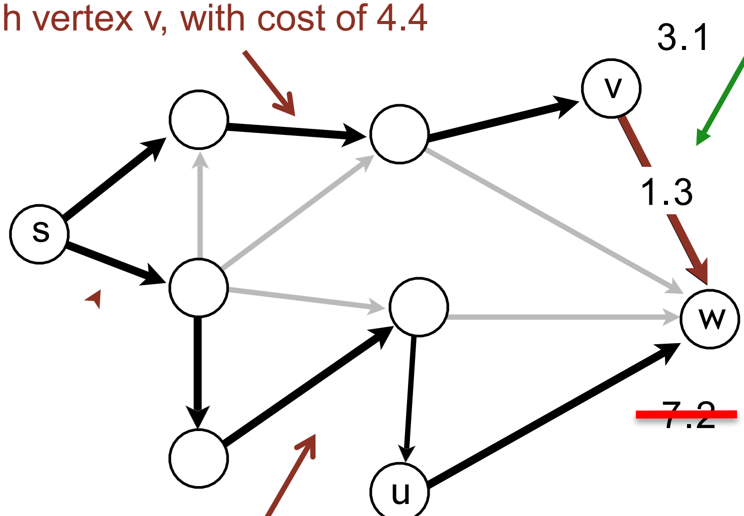
# Edge Relaxation

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

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

prevvertex[w] = v




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

```
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();
        prevvertex[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, prevvertex[w] = v

# Generic Shortest-paths Algorithm

---

Generic algorithm (to compute SPT from s)

---

For each vertex v: distTo[v] = ∞.

For each vertex v: prevvertex[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 prevvertex[v] is its previous vertex 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 vertex as 0 and to all other vertices as infinity.
  - Mark all vertices as unvisited and store them in a priority queue.
- Main Loop:
  - Visit the unvisited vertex v with the shortest known distance from the queue.
  - For each unvisited neighbor vertex w of vertex v, calculate its tentative distance through the current vertex. If this distance is smaller than the previously recorded distance, update it with edge relaxation for edge v-w.
  - Mark the current vertex as visited once all its neighbors are processed.
- Termination:
  - The algorithm continues until all reachable vertices are visited, or until the shortest path to a specific destination is found.
- Time complexity: O(E log V) for Binary Heap implementation
- Notes
  - It works for both undirected and directed graphs. The only difference is the function for getting the neighbors of vertex v, as each undirected edge is treated as two directed edges in opposite directions.)

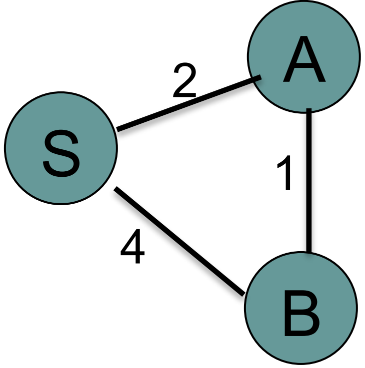
# Dijkstra's Algorithm: Correctness Proof

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: find shortest path starting from source vertex S for undirected graph




| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | ∞ | |
| B | ∞ | |

Visit S

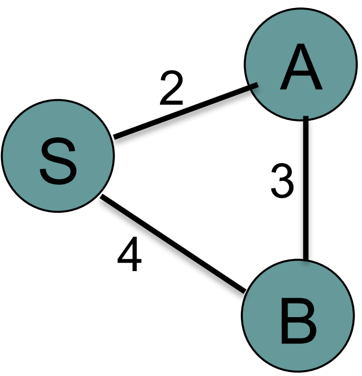
| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | 2 | S |
| B | 4 | S |

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

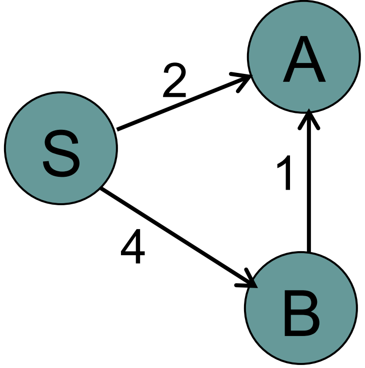
| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | 2 | S |
| B | 4 | S |

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 |




# Toy Example: find shortest path starting from source vertex S for directed graph

| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | ∞ | |
| B | ∞ | |

Visit S

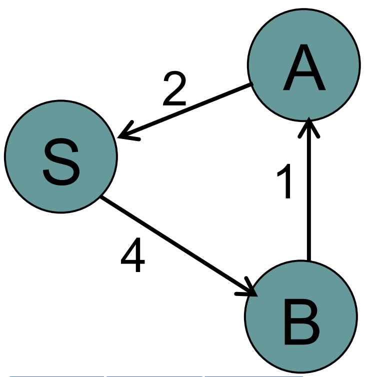
| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |

Visit

| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |

Visit B

| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |




| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | ∞ | |
| B | ∞ | |

Visit S

| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |

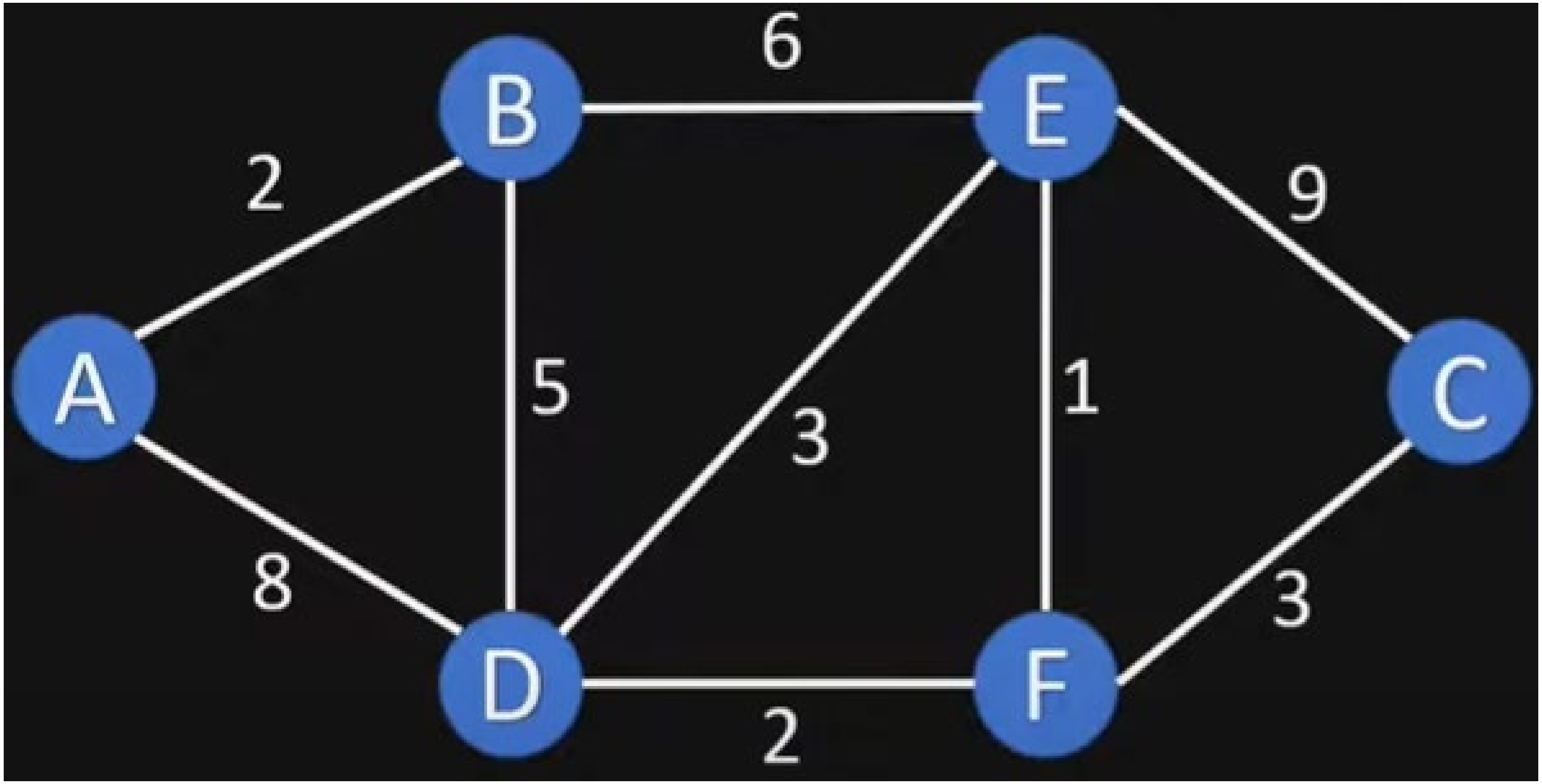
Visit A

| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |

Visit B

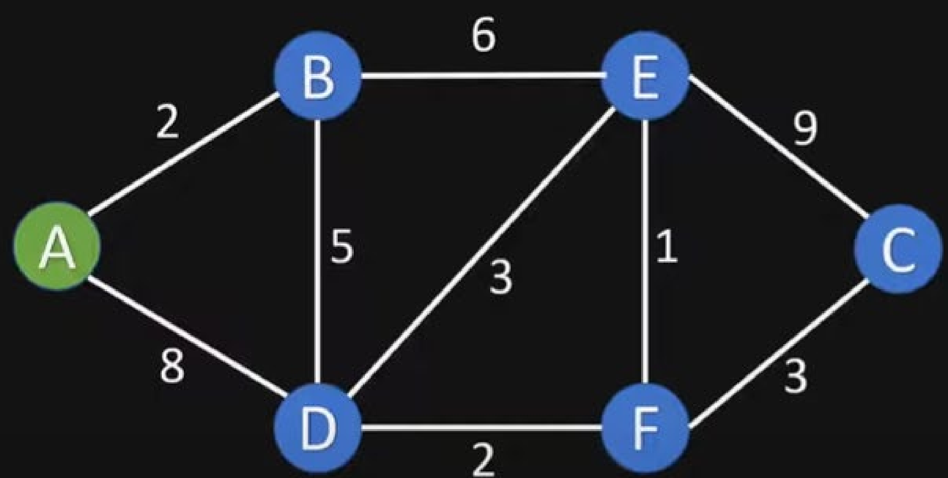
| N1 | SD | PN |
|---|---|---|
| S | 0 | |
| A | | |
| B | | |

# Example Graph



# Initialize

2. Assign to all nodes a tentative distance value

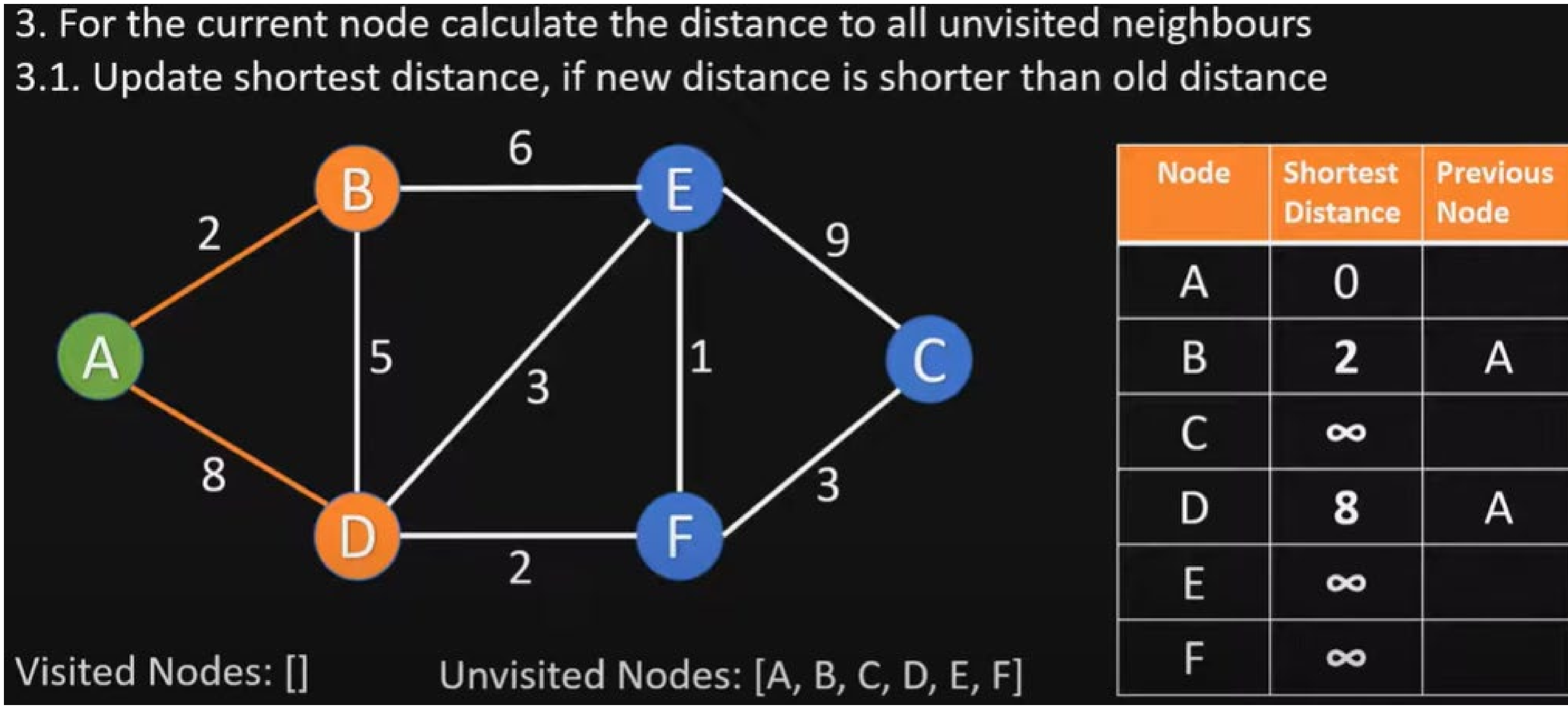



| Node | Shortest Distance | Previous Node |
|---|---|---|
| A | 0 | |
| B | ∞ | |
| C | ∞ | |
| D | ∞ | |
| E | ∞ | |
| F | ∞ | |

Visited Nodes: []

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

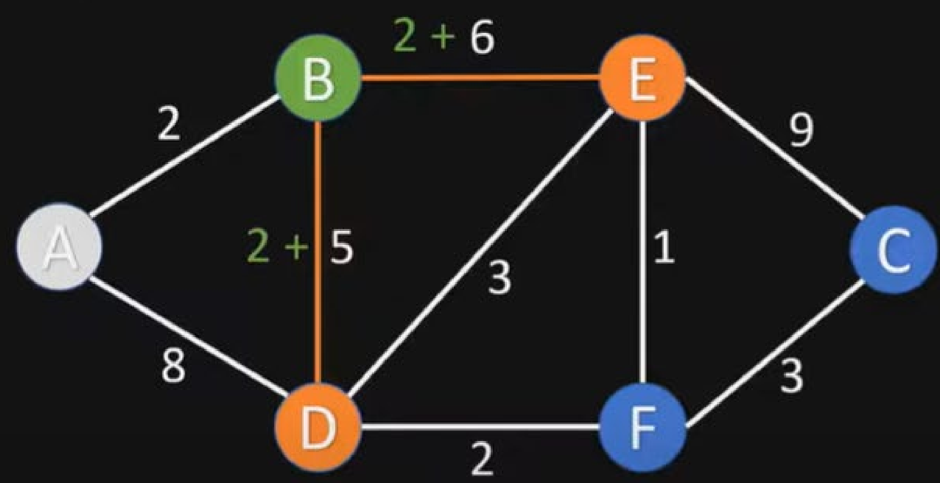
# Visit vertex A




OLD distTo[B] = ∞ > distTo[A] + e[A][B].weight() = 0+2 = 2
NEW distTo[B] ← distTo[A] + e[A][B].weight() = 2, prevvertex[B] = A
OLD distTo[D] = ∞ > distTo[A] + e[A][D].weight() = 0+8 = 8
NEW distTo[D] ← distTo[A] + e[A][D].weight() = 8, prevvertex[D] =

# Visit vertex 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



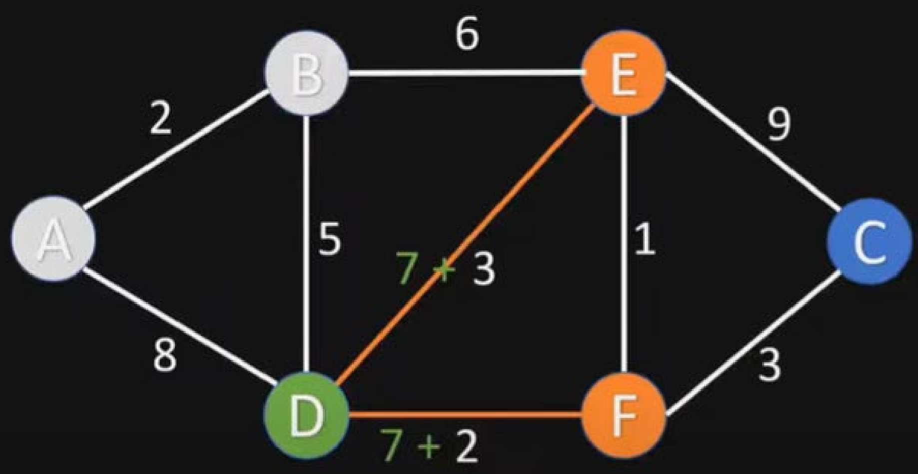

| Node | Shortest Distance | Previous Node |
|---|---|---|
| A | 0 | |
| B | **2** | A |
| C | ∞ | |
| D | **7** | B |
| E | **8** | B |
| F | ∞ | |

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

# Visit vertex 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

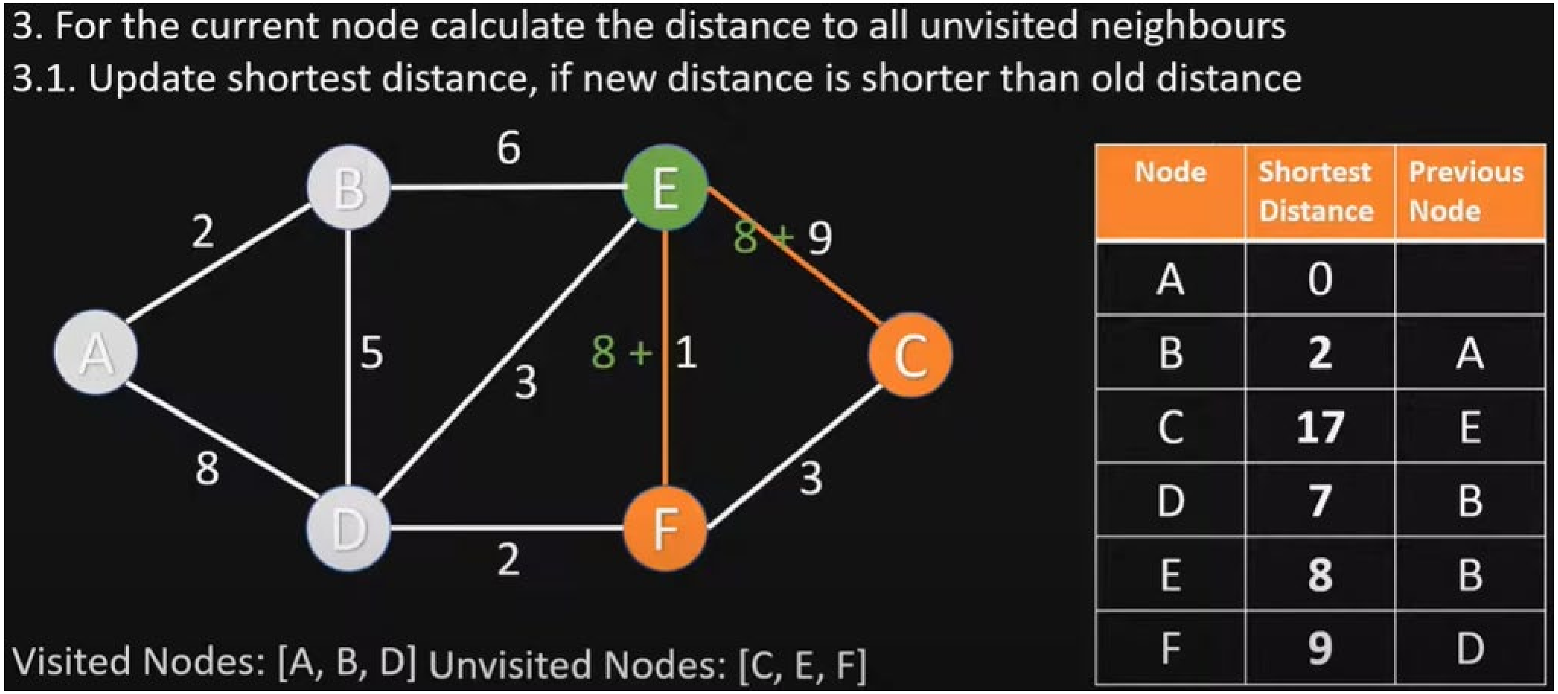



| Node | Shortest Distance | Previous Node |
|---|---|---|
| A | 0 | |
| B | **2** | A |
| C | ∞ | |
| D | **7** | B |
| E | **8** | B |
| F | **9** | D |

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

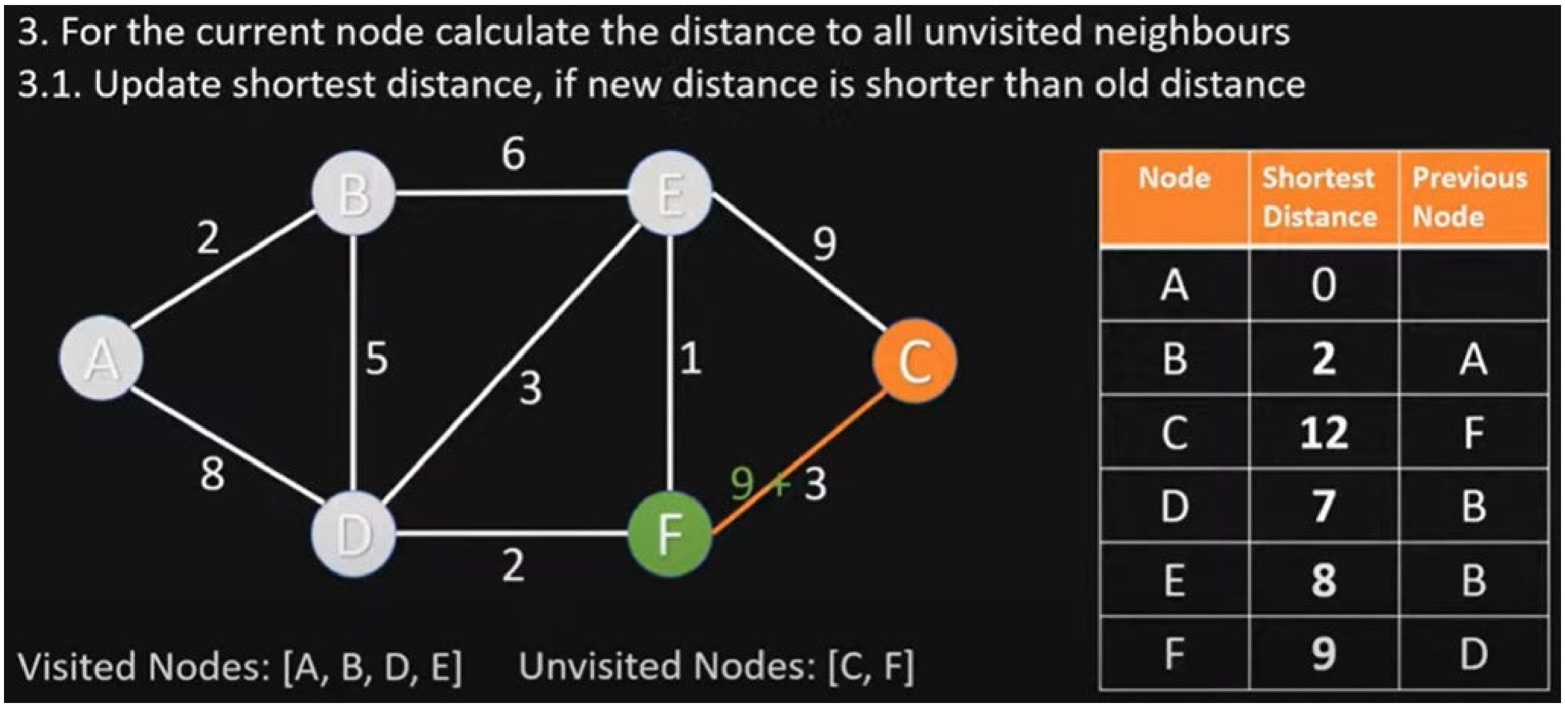
OLD distTo[E] = 8 < distTo[D] + e[D][E].weight() = 7+3 = 10
No update, distTo[E] stays 8, prevvertex[E] stays B
OLD distTo[F] = ∞ > distTo[D] + e[D][F].weight() = 7+2 = 9
NEW distTo[F] ← distTo[D] + e[D][E].weight() = 9, prevvertex[F] = D

# Visit vertex E



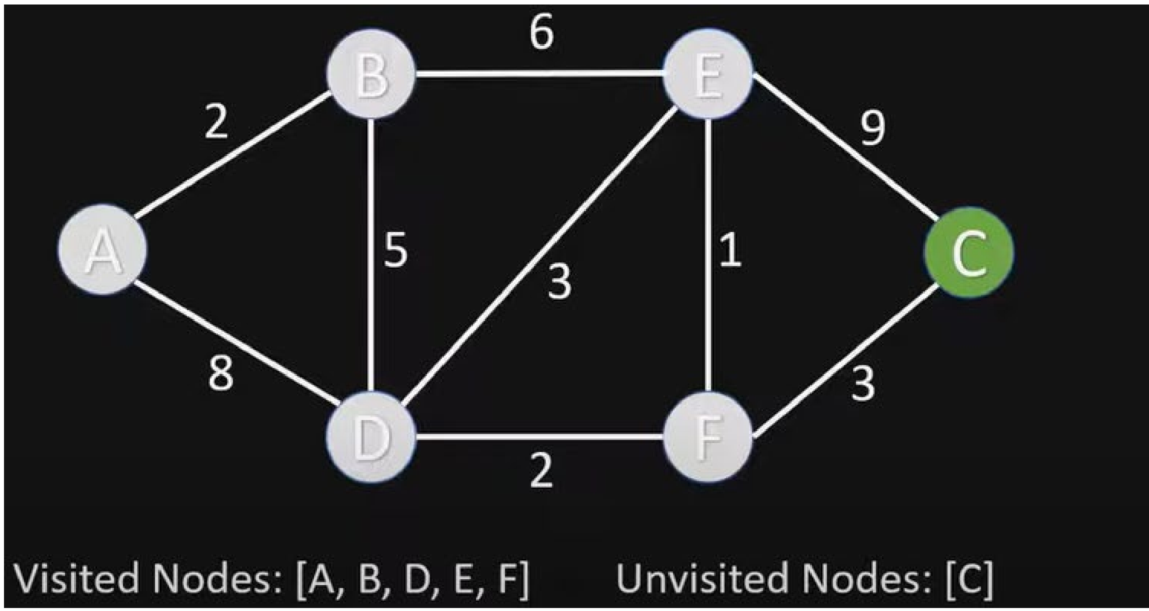

OLD distTo[C] = ∞ > distTo[E] + e[E][C].weight() = 8+9 = 17
NEW distTo[C] ← distTo[E] + e[E][C].weight() = 17, prevvertex[C] = E
OLD distTo[F] = 9 = distTo[E] + e[E][F].weight() = 8+1 = 9
No update, distTo[F] stays 9, prevvertex[F] = D (You can also update prevvertex[F] = E.)

# Visit vertex F




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

# Visit vertex C

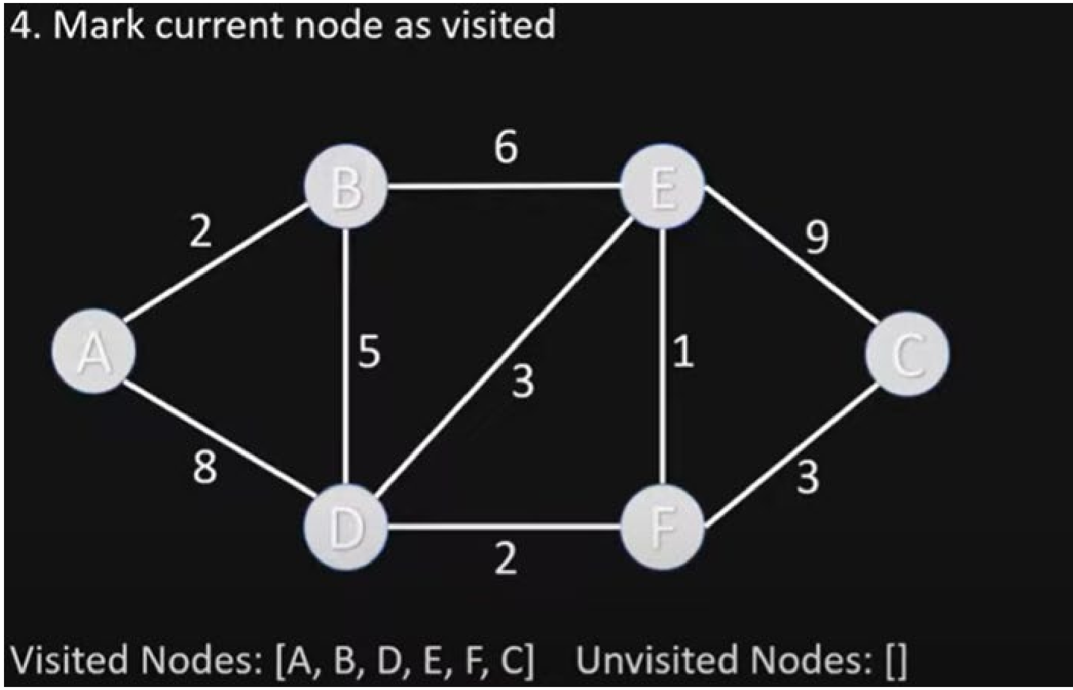



| Node | Shortest Distance | Previous Node |
|---|---|---|
| A | 0 | |
| B | **2** | A |
| C | **12** | F |
| D | **7** | B |
| E | **8** | B |
| F | **9** | D |

Nothing changes, since C has no unvisited neighbor vertices

# End of Algorithm

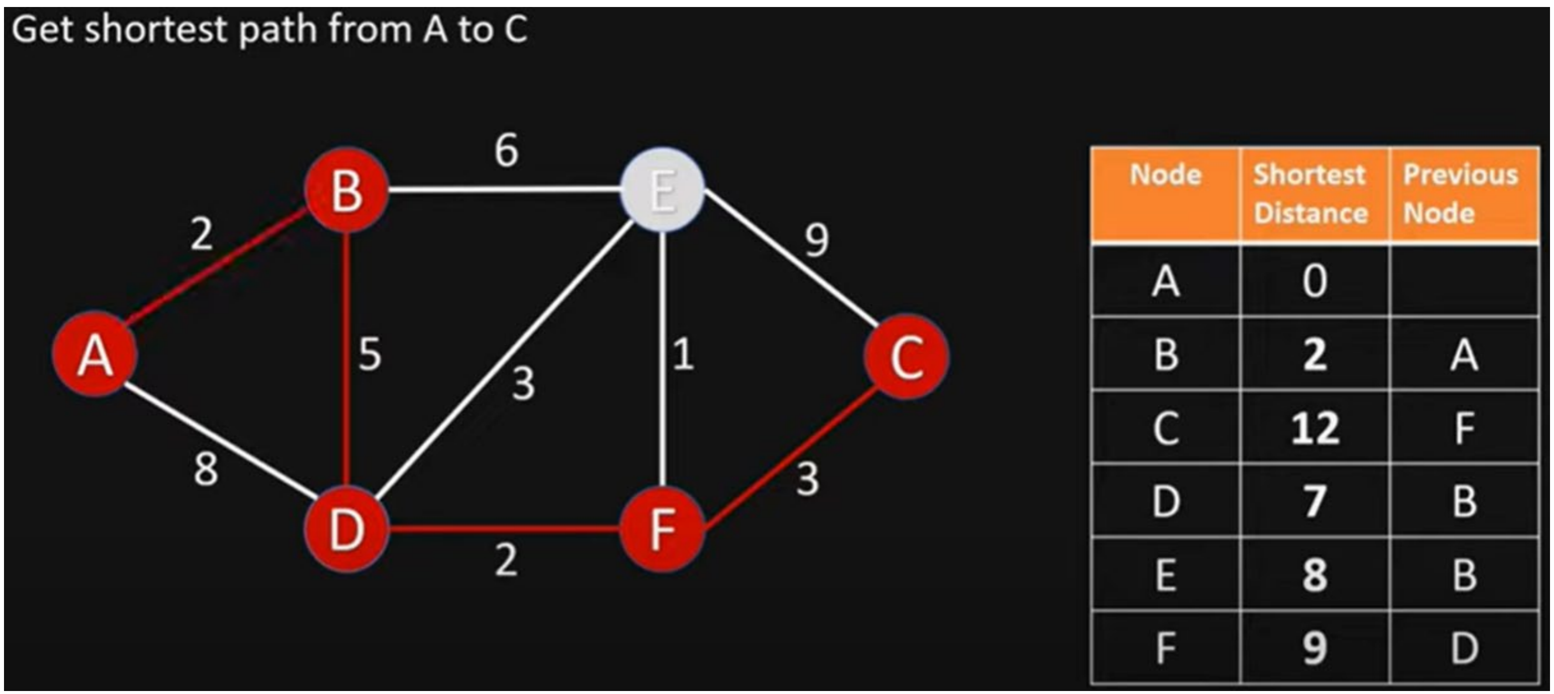
- Table contains the shortest distance to each vertex N from the source vertex A, and its previous vertex in the shortest path




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

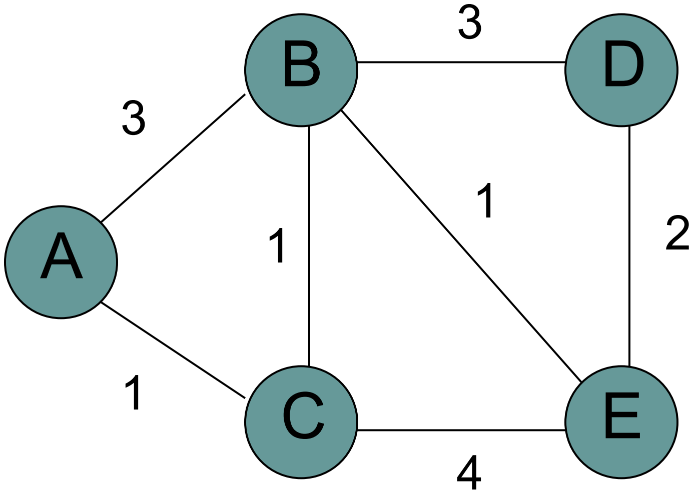
# Getting the Shortest Path from A to C

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

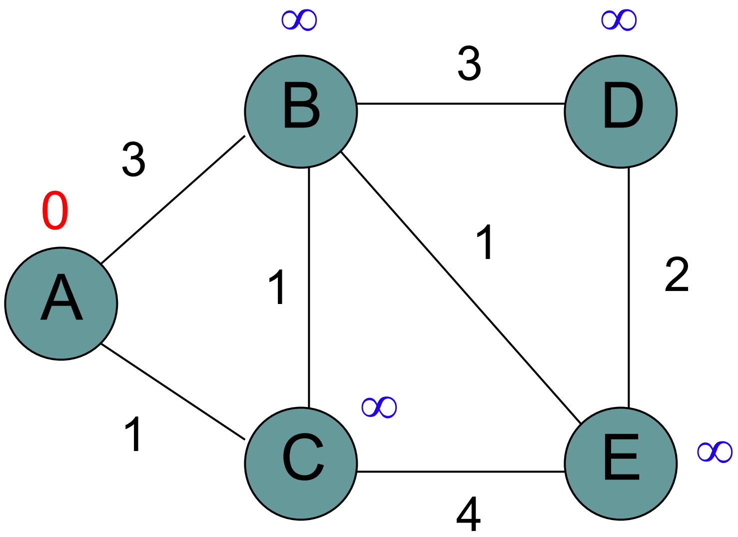



| Node | Shortest Distance | Previous Node |
|---|---|---|
| A | 0 | |
| B | 2 | A |
| C | 12 | F |
| D | 7 | B |
| E | 8 | B |
| F | 9 | D |

# Dijkstra’s Algorithm Example 2

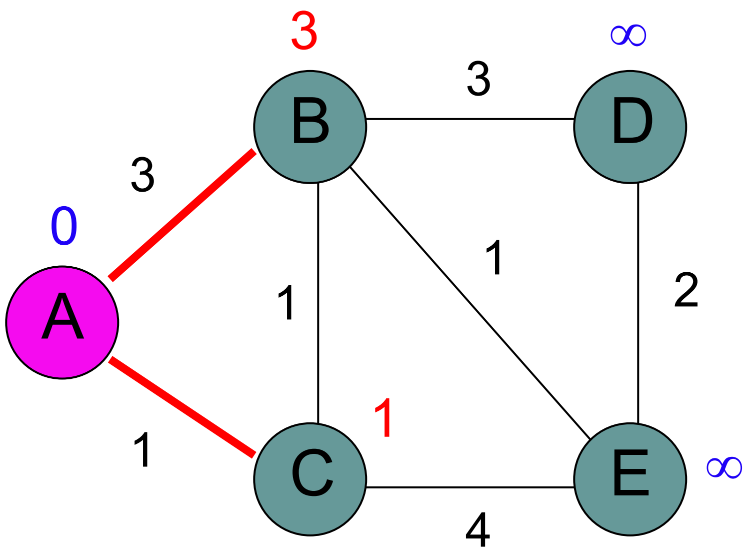


# Initialize



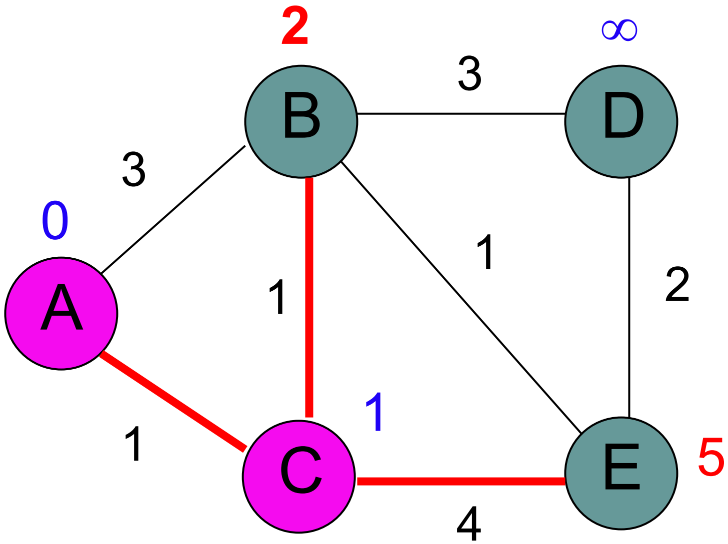

| N | SD | PN |
|---|---|---|
| A | 0 | |
| B | ∞ | |
| C | ∞ | |
| D | ∞ | |
| E | ∞ | |

# Visit vertex A



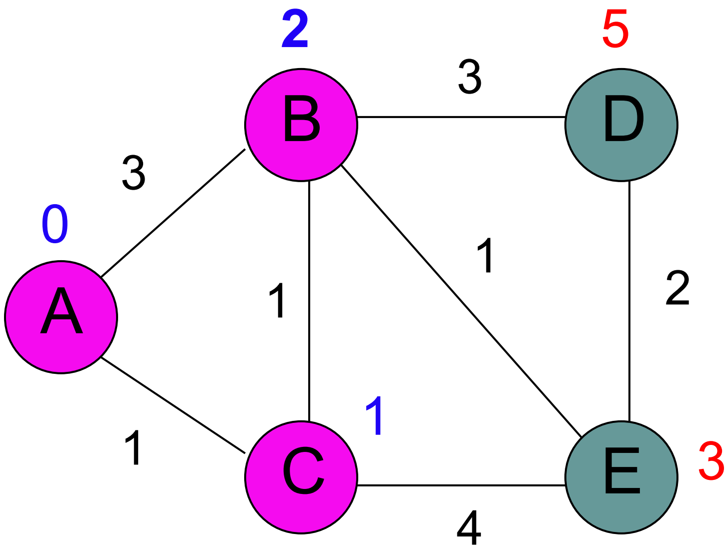

| N | SD | PN |
|---|---|---|
| A | 0 | |
| B | 3 | A |
| C | 1 | A |
| D | ∞ | |
| E | ∞ | |

# Visit vertex C



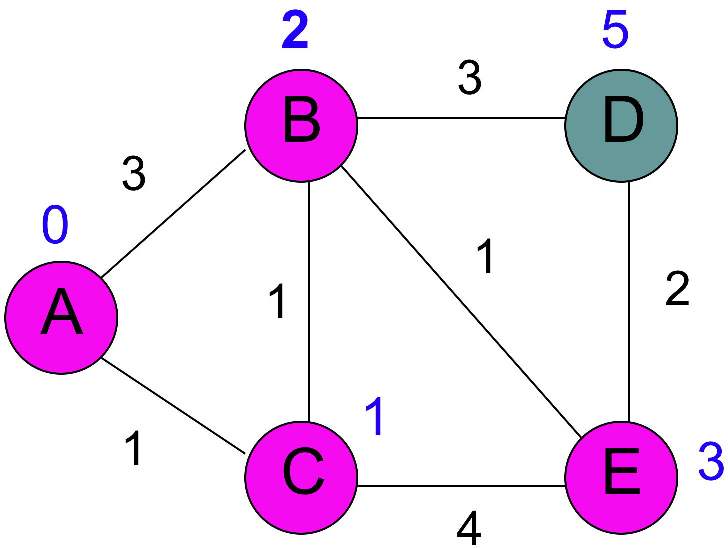

| N | SD | PN |
|---|---|---|
| A | 0 | |
| B | 2 | C |
| C | 1 | A |
| D | ∞ | |
| E | 5 | C |

# Visit vertex B




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

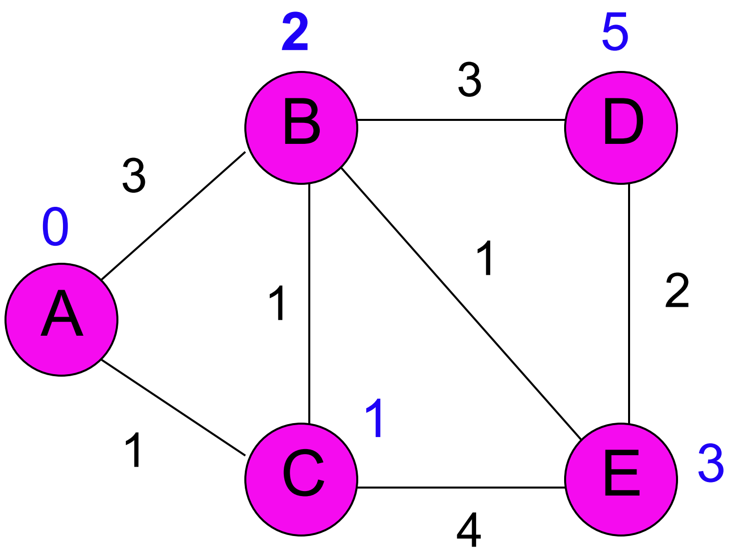
# Visit vertex E




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

Nothing changes

# Visit vertex D



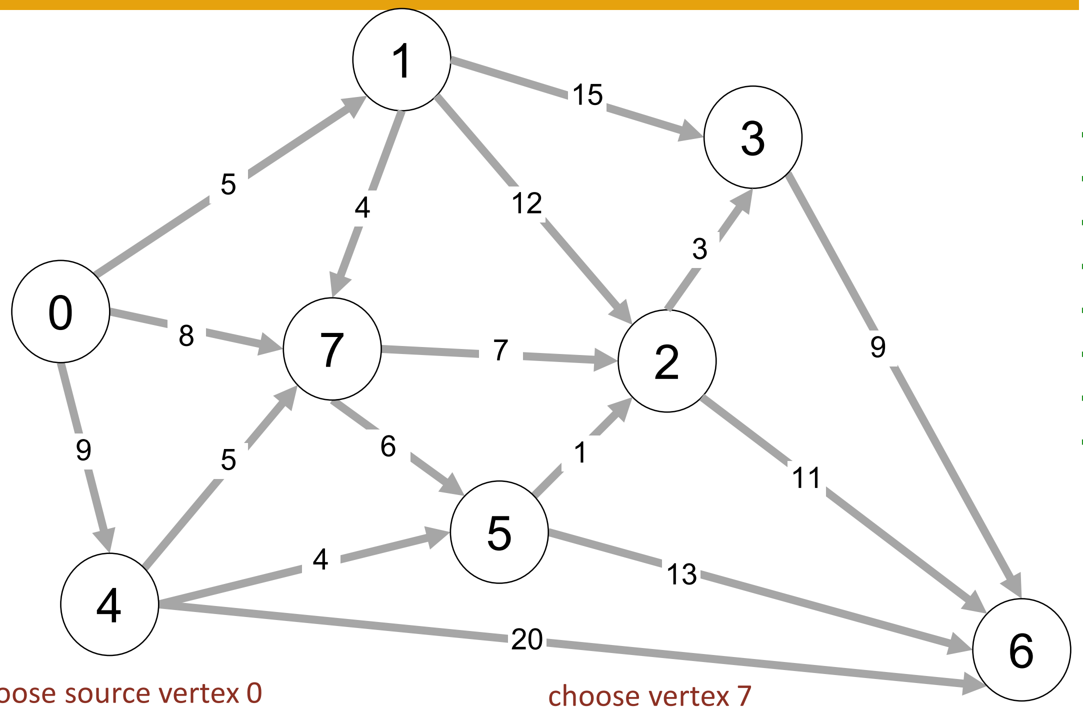

| N | SD | PN |
|---|---|---|
| A | 0 | |
| B | 2 | C |
| C | 1 | A |
| D | 5 | B |
| E | 3 | B |

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

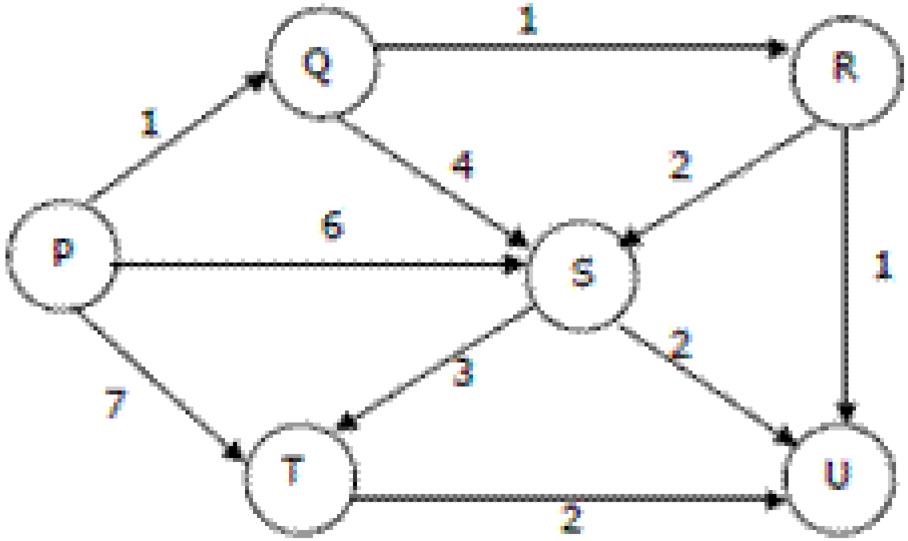
choose vertex 7
relax all edges adjacent from 7
choose vertex 4
relax all edges adjacent from 4

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

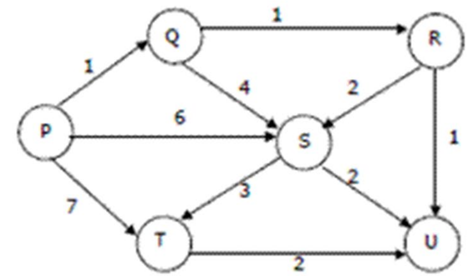
| v | edgeTo[] | | | |
|---|---|---|---|---|
| 0 | - | | | |
| 1 | ~~-~~ | 0 | | |
| 2 | ~~-~~ | ~~1~~ | ~~7~~ | 5 |
| 3 | ~~-~~ | ~~1~~ | 2 | |
| 4 | ~~-~~ | 0 | | |
| 5 | ~~-~~ | ~~7~~ | 4 | |
| 6 | ~~-~~ | ~~4~~ | ~~5~~ | 2 |
| 7 | ~~-~~ | 0 | | |

# Dijkstra’s Algorithm Example 4

- 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 vertices 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 vertex




| 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.

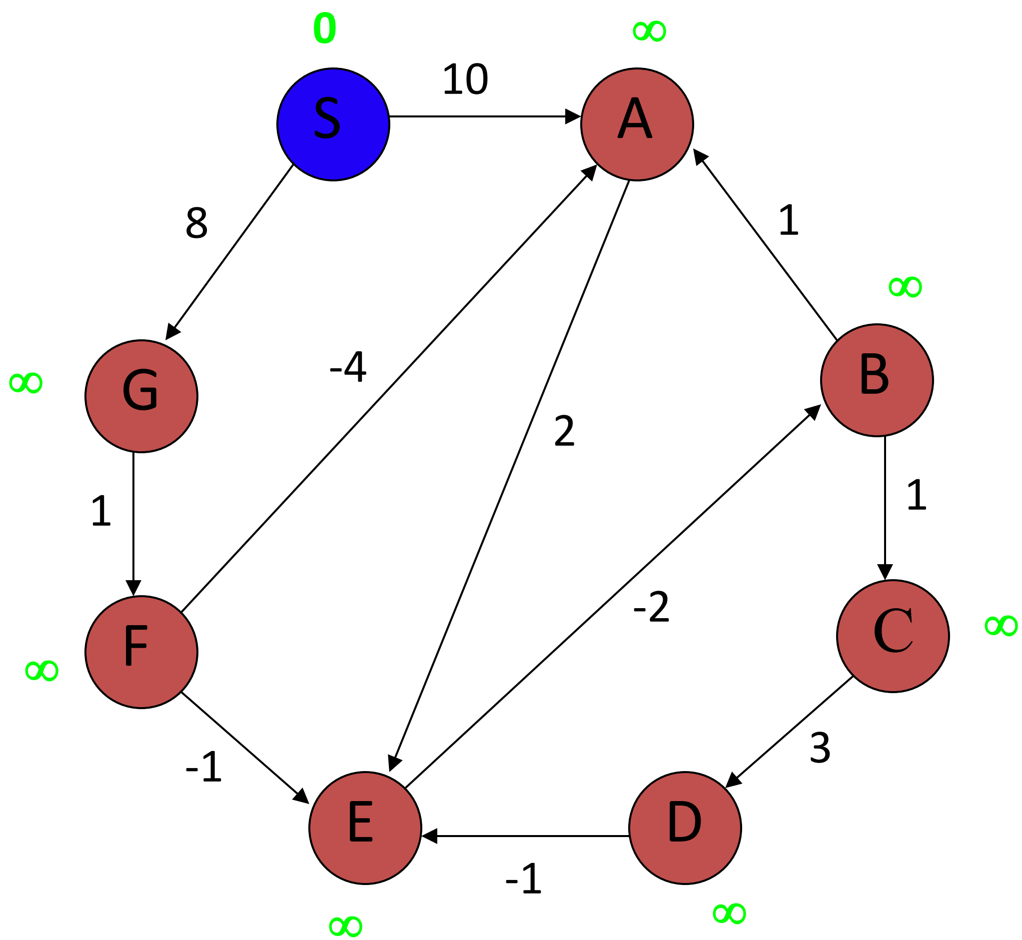
# Bellman-Ford Algorithm with Negative Cycle Detection

- 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.
- Notes:
  - It can find any negative cycle that is reachable from source vertex s (but not negative cycles that are unreachable from s).
  - If there is a negative cycle that is reachable from source vertex s, then any paths that go through the cycle has distance $-\infty$, since the cost can be reduced by traversing the cycle infinite number of times.

# Time Complexity of Bellman-Ford Algorithm

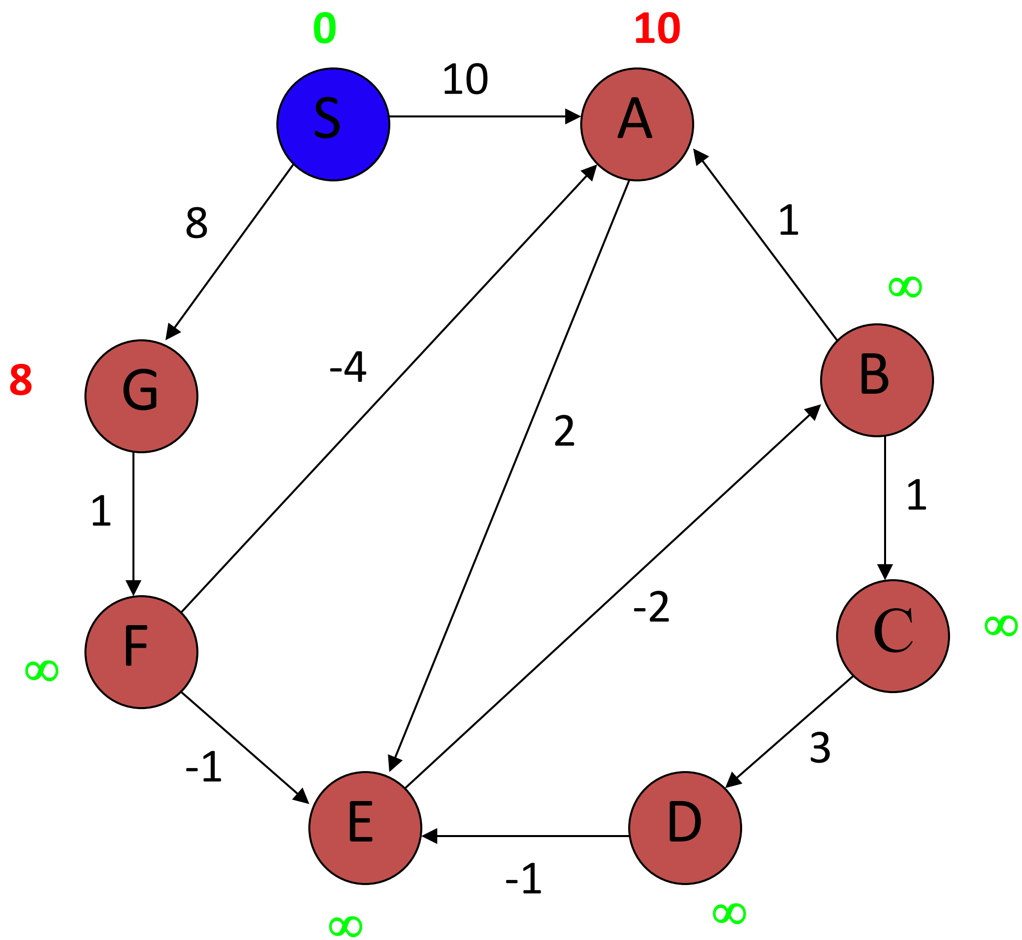
- Time complexity for connected graph:
- Average Case: O(VE)
- Worst Case: O(VE)
  - If the graph is dense or complete, the value of E becomes $O(V^2)$. So overall time complexity becomes $O(V^3)$

# Bellman-Ford Algorithm Example 1



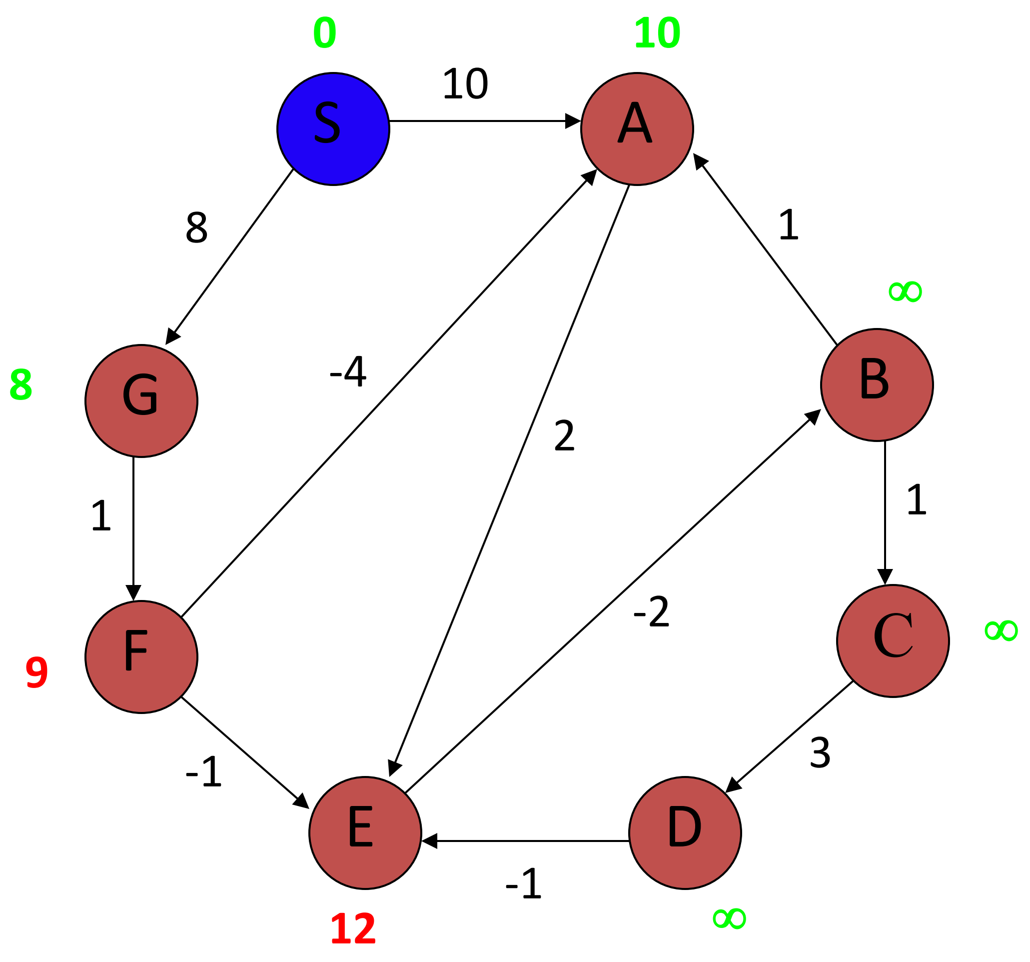

Iteration: 0

# Bellman-Ford Algorithm Example 1



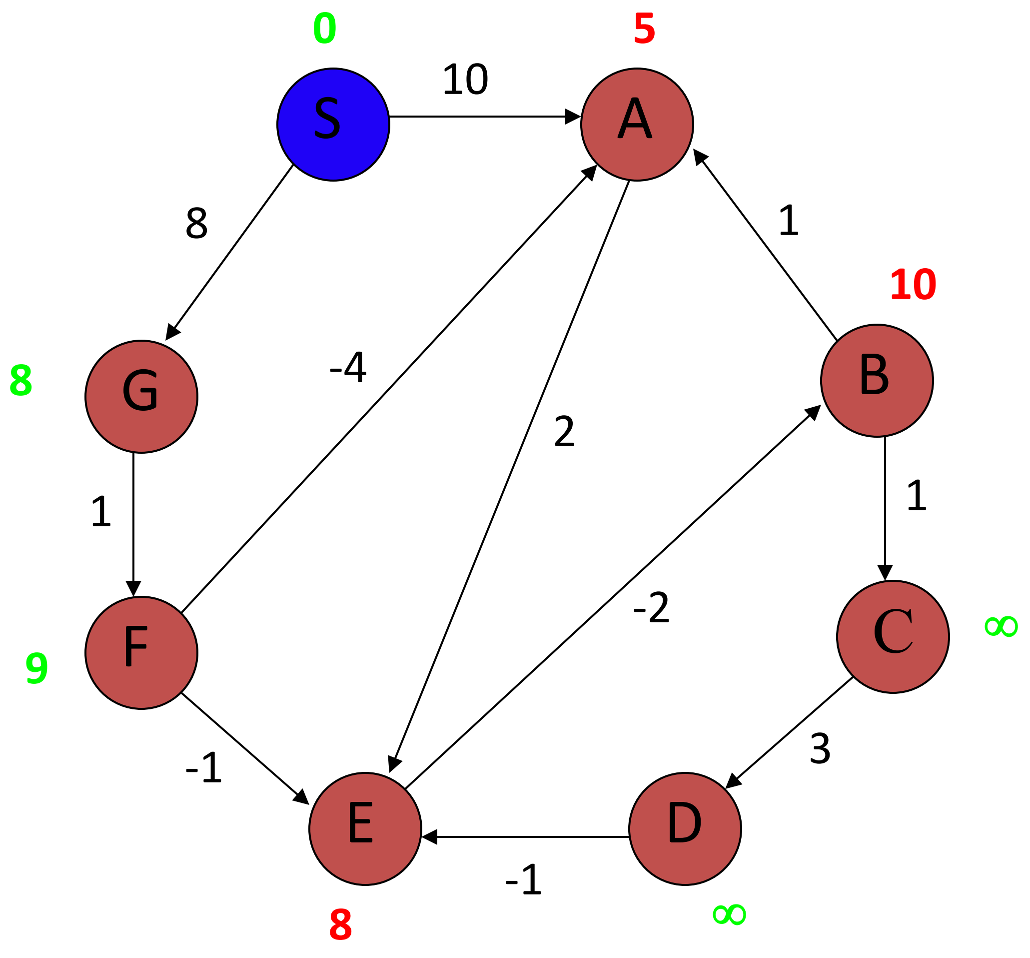

Iteration: 1

# Bellman-Ford Algorithm Example 1




Iteration: 2

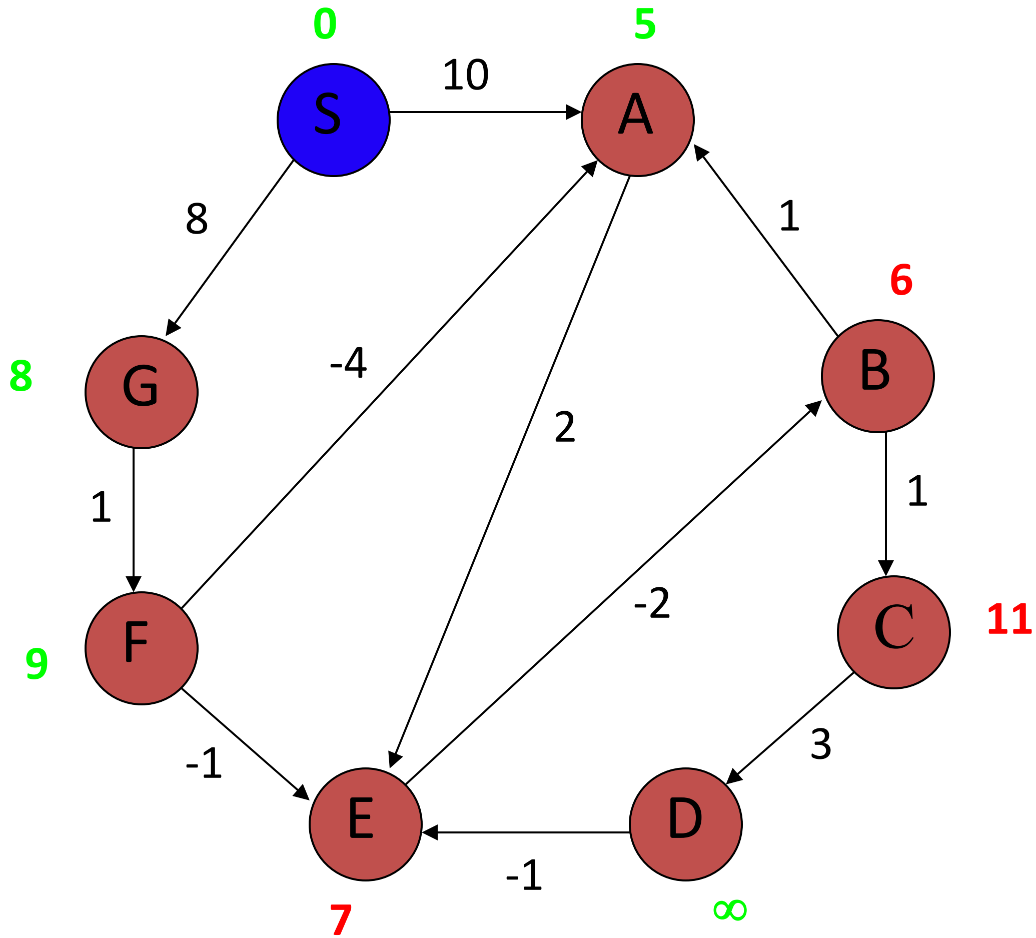
# Bellman-Ford Algorithm Example 1




Iteration: 3

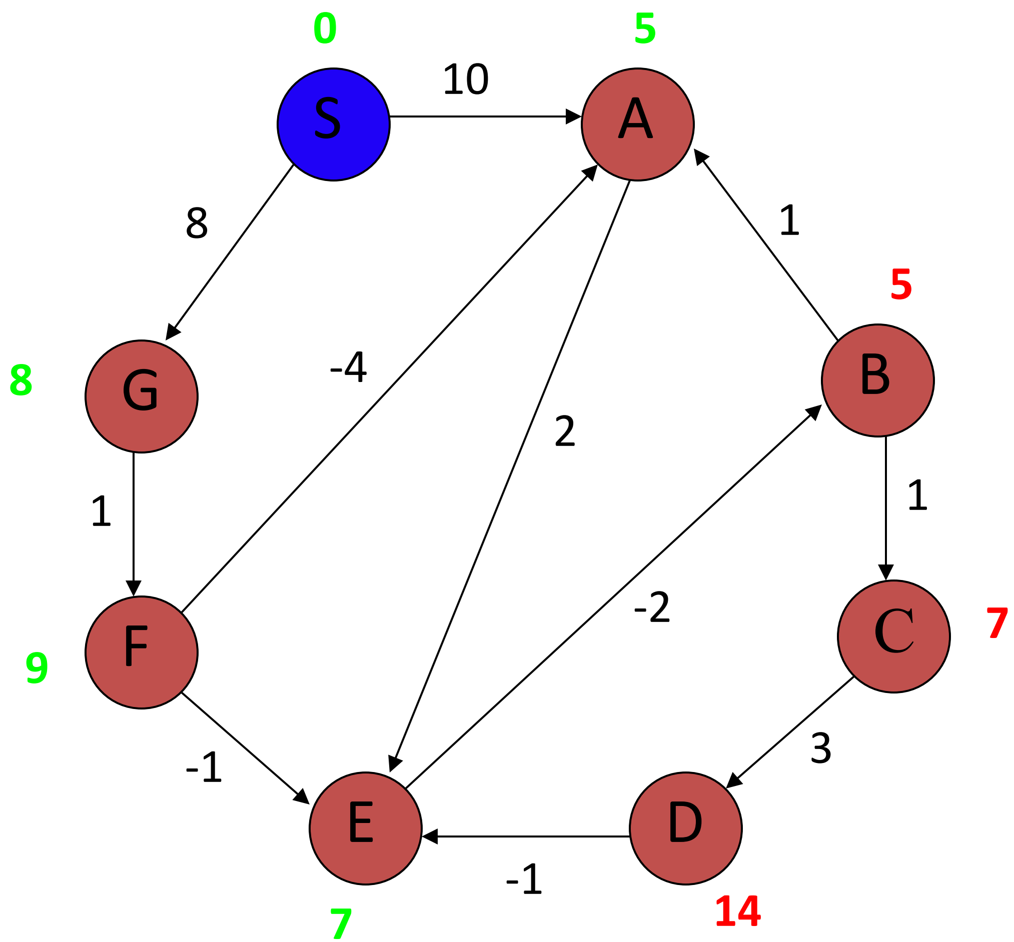
A has the correct distance and path

# Bellman-Ford Algorithm Example 1




Iteration: 4

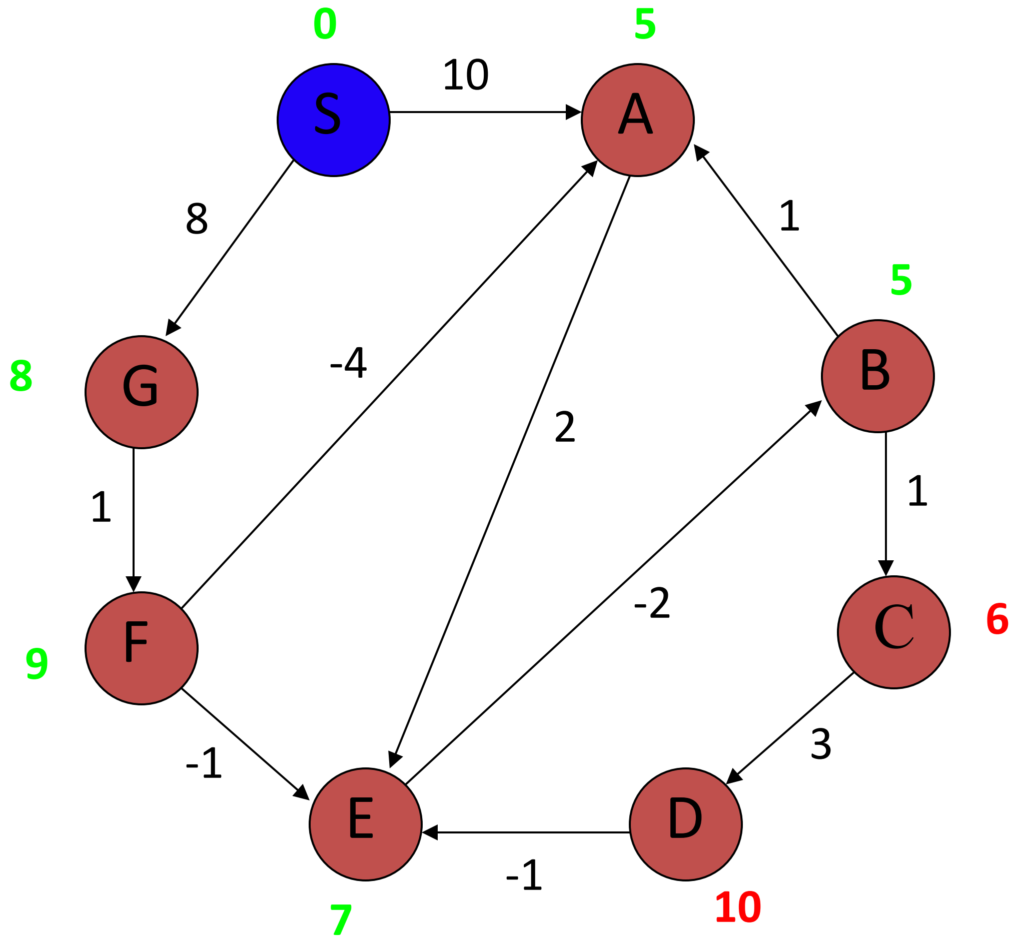
# Bellman-Ford Algorithm Example 1




Iteration: 5

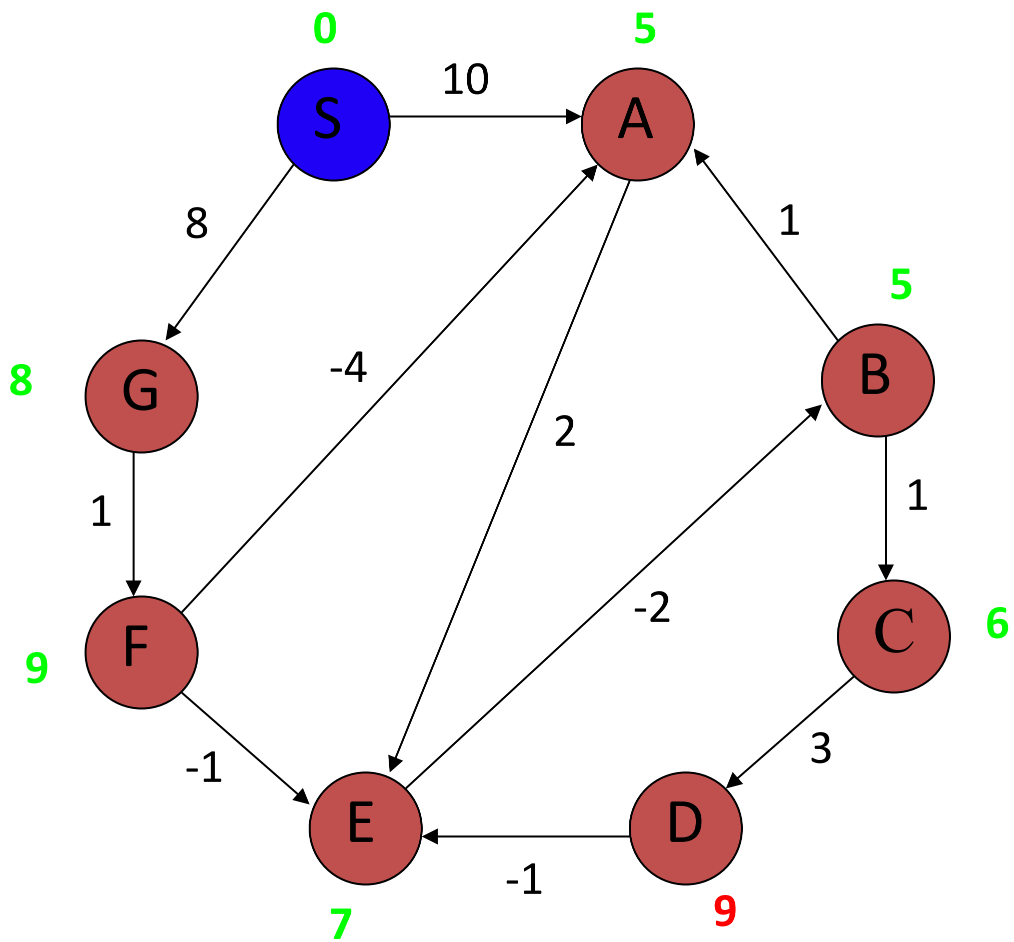
B has the correct distance and path

# Bellman-Ford Algorithm Example 1




Iteration: 6

# Bellman-Ford Algorithm Example 1

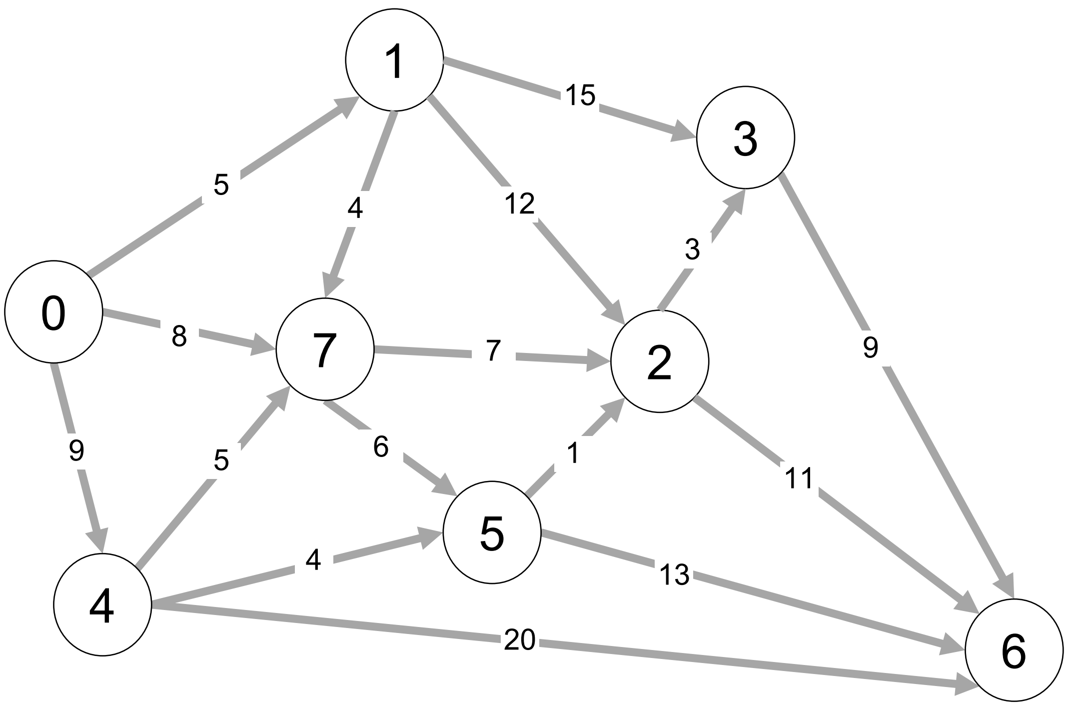



Iteration: 7

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

# Bellman-Ford Algorithm Example 2

Repeat V − 1 times: relax all E edges.



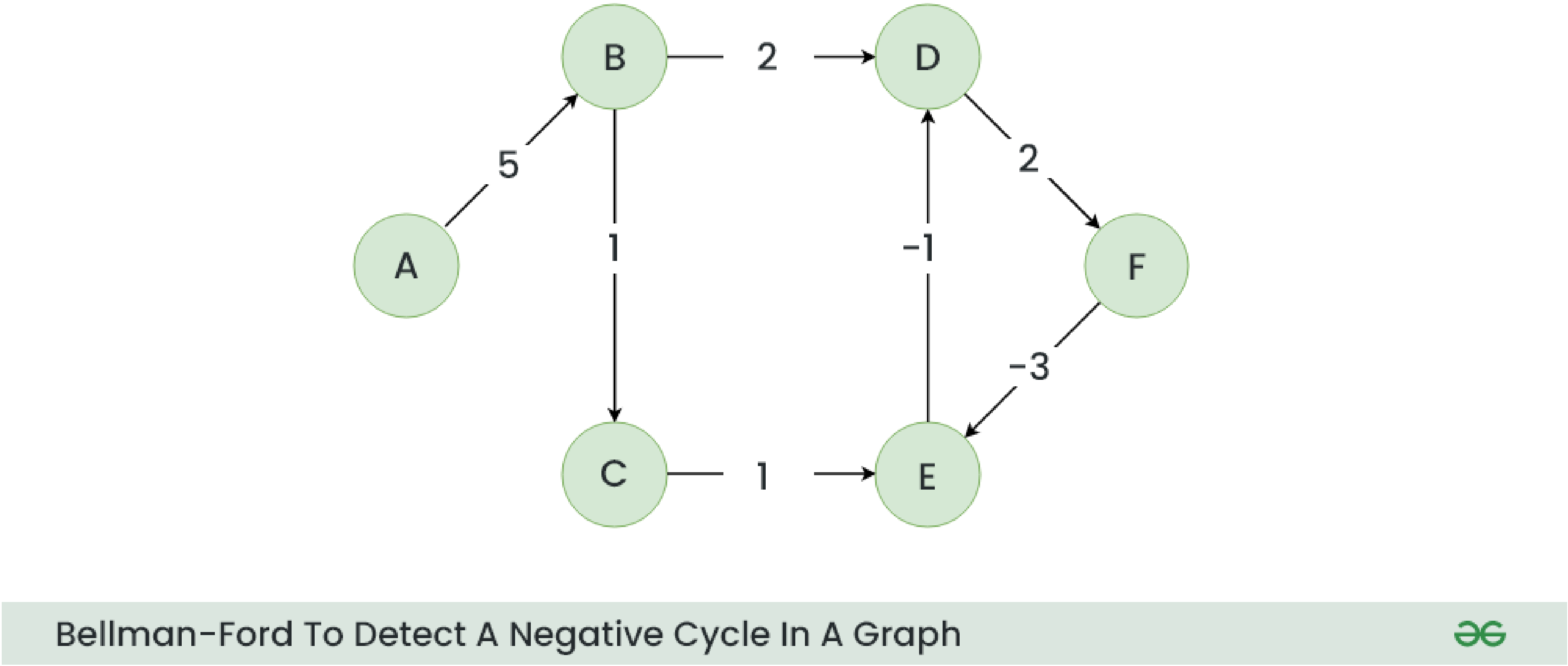

| v | distTo[] | | | |
|---|---|---|---|---|
| 0 | ~~∞~~ | 0 | | |
| 1 | ~~∞~~ | 5 | | |
| 2 | ~~∞~~ | ~~17~~ | 14 | |
| 3 | ~~∞~~ | ~~20~~ | 17 | |
| 4 | ~~∞~~ | 9 | | |
| 5 | ~~∞~~ | 13 | | |
| 6 | ~~∞~~ | ~~28~~ | ~~26~~ | 25 |
| 7 | ~~∞~~ | 8 | | |

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

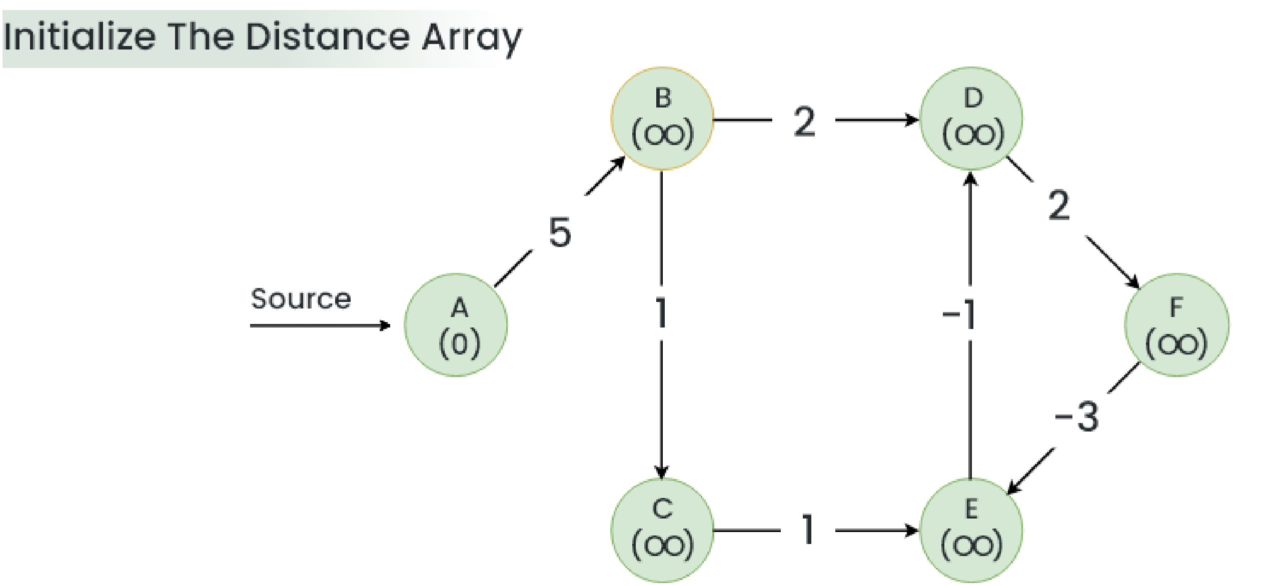
pass 1 pass 2 pass 3 (no further changes) pass 4-7 (no further changes)

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 Example 3 w. Negative Cycle



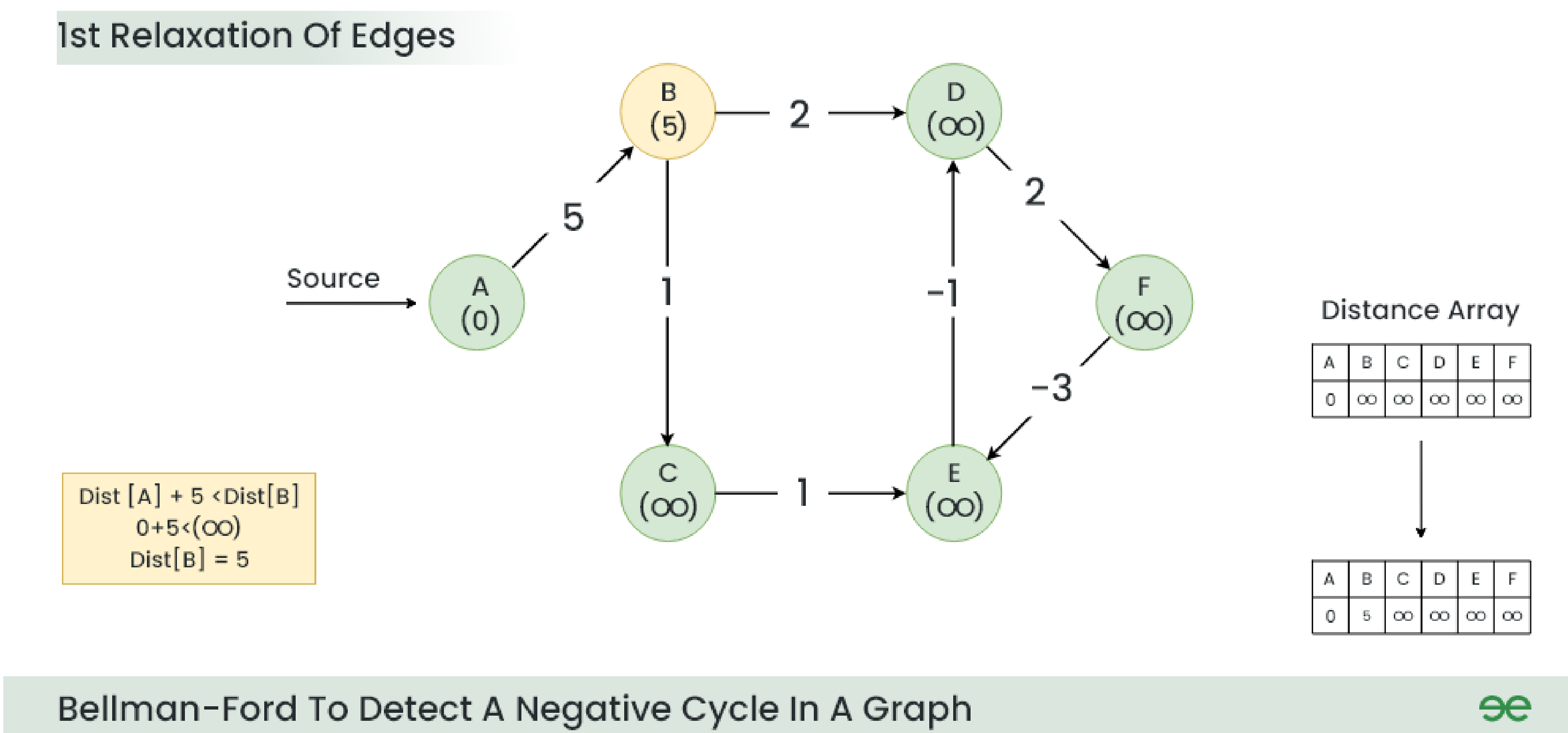
- Step 1: Initialize a distance array Dist[] to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY. distTo[N] = ∞



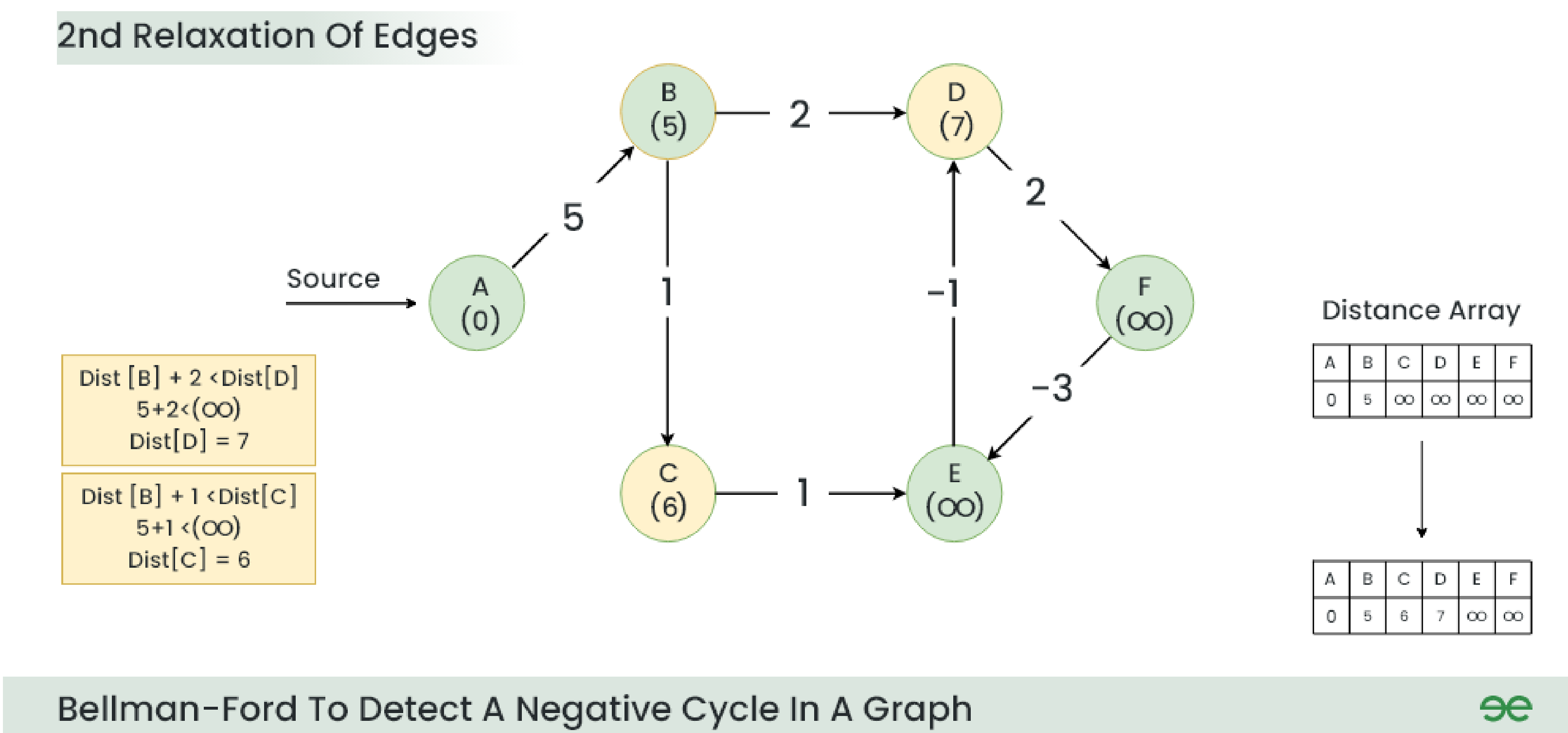

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ |

Bellman-Ford To Detect A Negative Cycle In A Graph

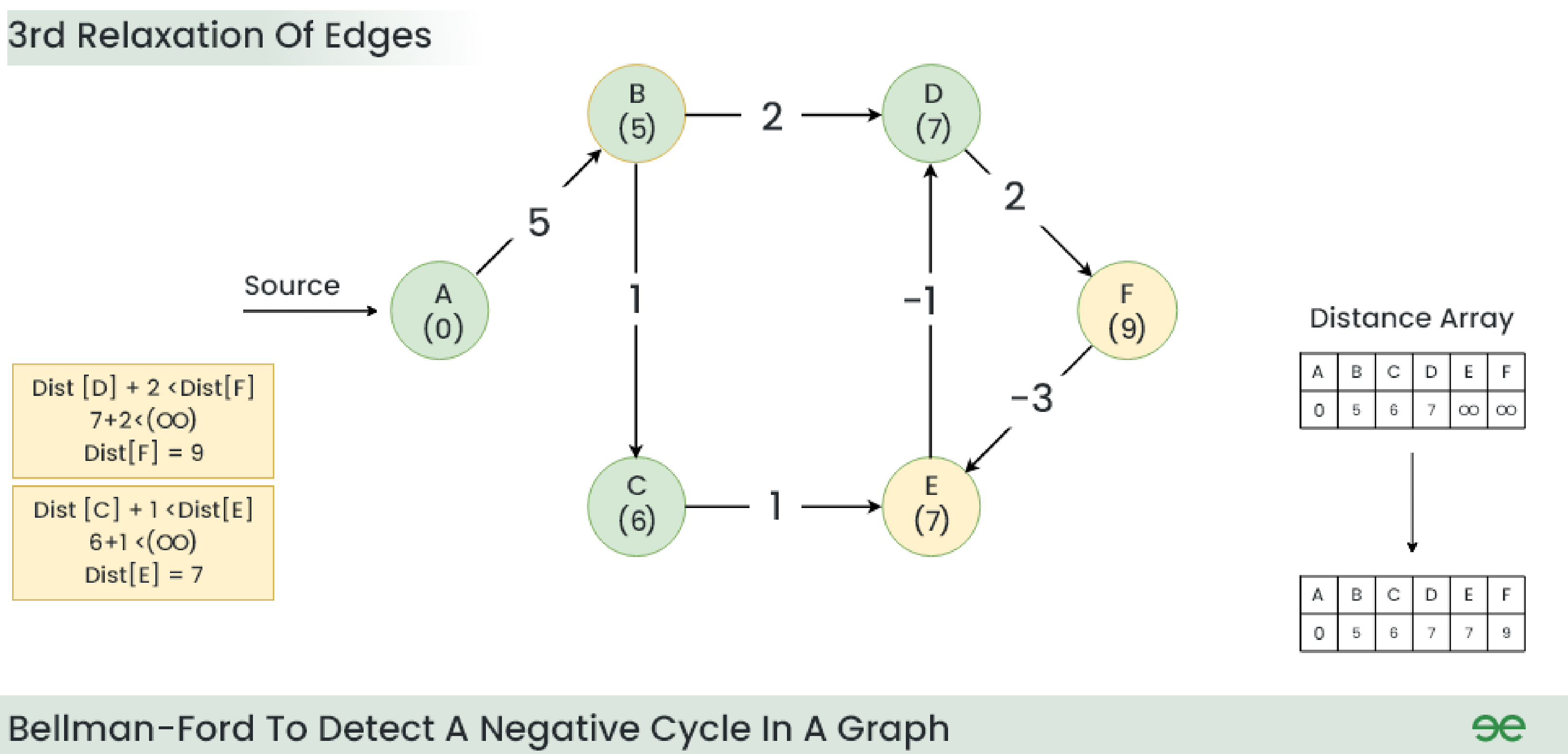
- Step 2: Start relaxing the edges, during 1st relaxation:
- OLD distTo[B] = $\infty$ > distTo[A] + e[A][B].weight() = 0+5 = 5
- NEW distTo[B] = distTo[A] + e[A][B].weight() = 5



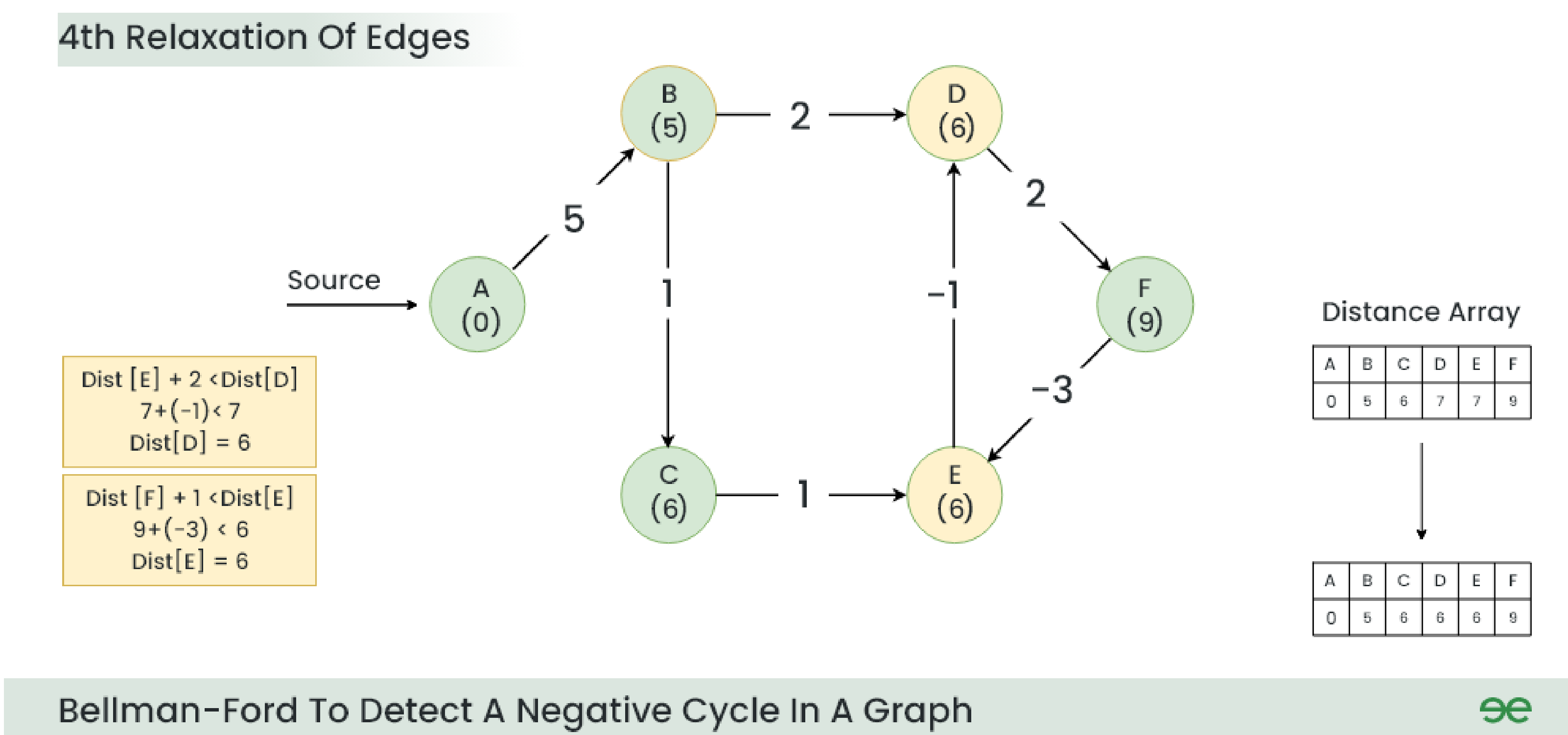
- Step 3: During 2nd relaxation:
- OLD distTo[D] = ∞ > distTo[B] + e[B][D].weight() = 5+2 = 7
- NEW distTo[D] ← distTo[B] + e[B][D].weight() = 7
- OLD distTo[C] = ∞ > distTo[B] + e[B][C].weight() = 5+1 = 6
- NEW distTo[C] ← distTo[B] + e[B][C].weight() = 6



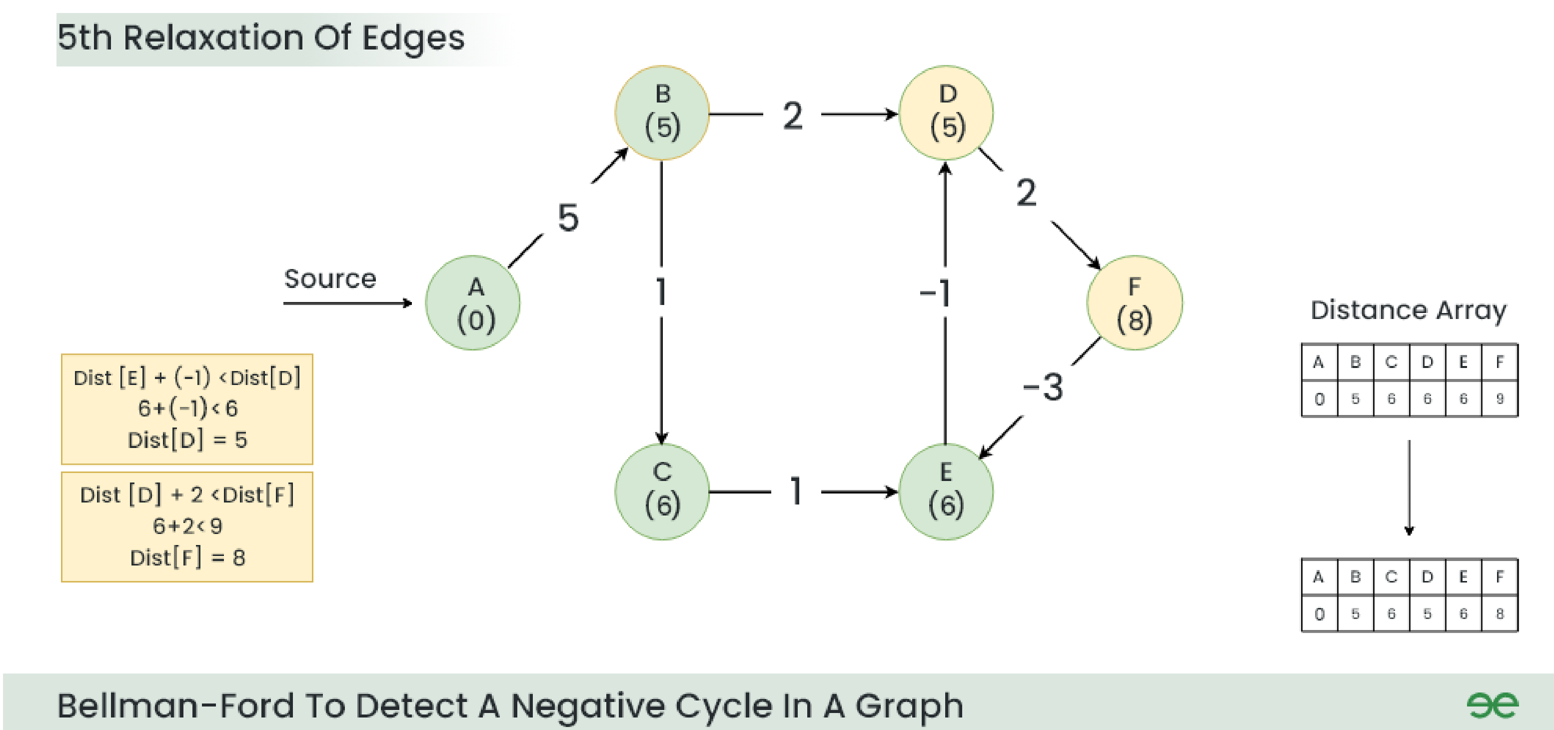
- Step 4: During 3rd relaxation:
- OLD distTo[F] = ∞ > distTo[D] + e[D][F].weight() = 7+2 = 9
- NEW distTo[F] ← distTo[D] + e[D][F].weight() = 9
- OLD distTo[E] = ∞ > distTo[C] + e[C][E].weight() = 6+1 = 7
- NEW distTo[E] ← distTo[C] + e[C][E].weight() = 7



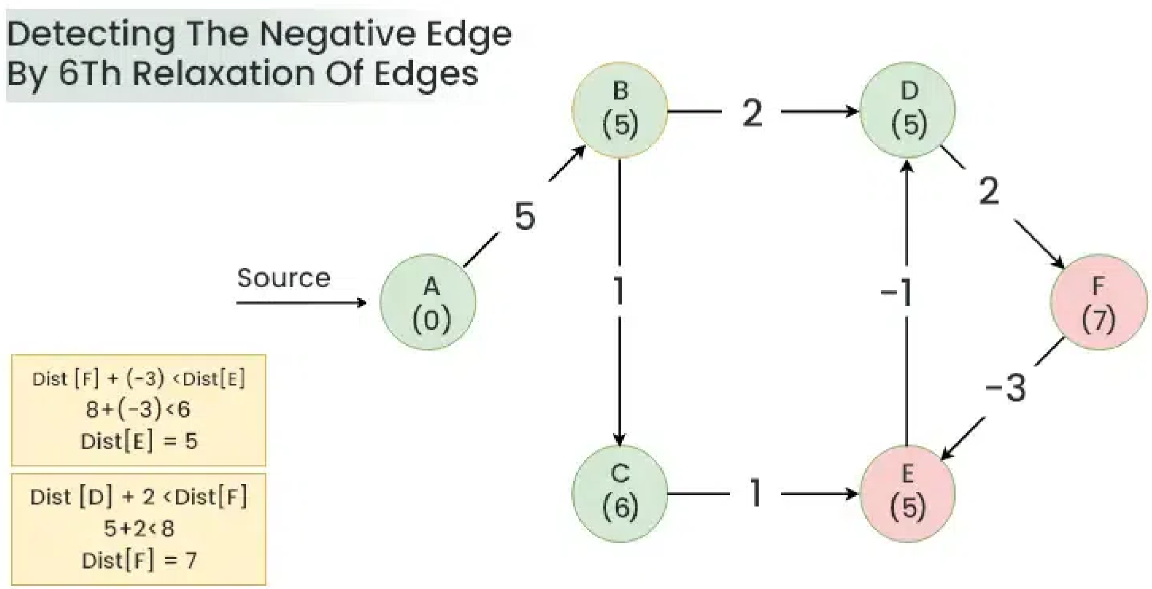
- Step 5: During 4$^{th}$ relaxation:
- OLD distTo[D] = 7 > distTo[E] + e[E][D].weight() = 7-1 = 6
- NEW distTo[D] ← distTo[E] + e[E][D].weight() = 6
- OLD distTo[E] = 7 > distTo[F] + e[F][E].weight() = 9-3 = 6
- NEW distTo[E] ← distTo[F] + e[F][E].weight() = 6



- Step 6: During $5^{th}$ 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).




| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | 5 | 6 | 5 | 6 | 8 |

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | 5 | 6 | 5 | 4 | 7 |

Bellman-Ford To Detect A Negative Cycle In A Graph

# 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 vertex.
  - Works only on graphs with non-negative edge weights.
- Bellman-Ford Algorithm:
  - Iteratively relaxes all edges V-1 times.
  - Does not use a priority queue.
  - Can handle graphs with negative edge weights, and can detect negative cycles.
- Dijkstra's algorithm is faster and more efficient for graphs with non-negative weights; Bellman-Ford Algorithm 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 vertex 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 |

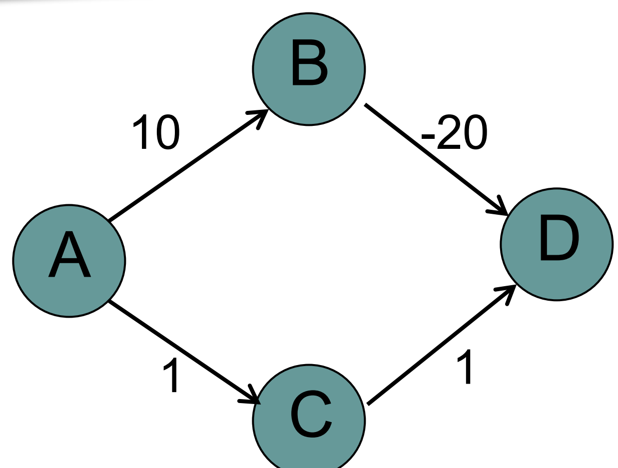
Visit D

| 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 vertex A to all other vertices. The shortest path from vertex A to vertex D goes through vertex 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

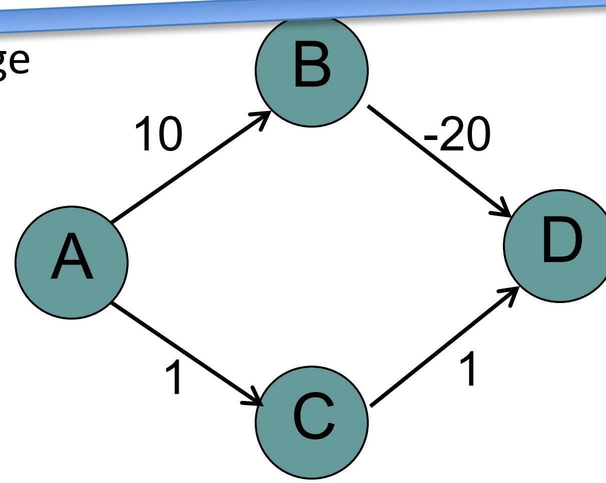
No change

| 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 |



# Topological Sort for Shortest Paths in Edge-weighted DAG

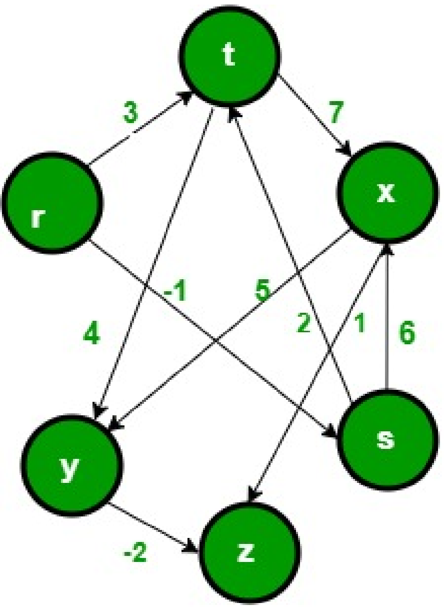
- Suppose that an edge-weighted digraph has no directed cycles. Is it easier to find shortest paths than in a general digraph?
- Idea: Consider vertices in topological order. Relax all edges pointing from that vertex

- Initialize dist[] = {∞,∞, ….} and dist[s] = 0 where s is the source vertex.
- Create a topological order of all vertices.
- For every vertex u in topological order

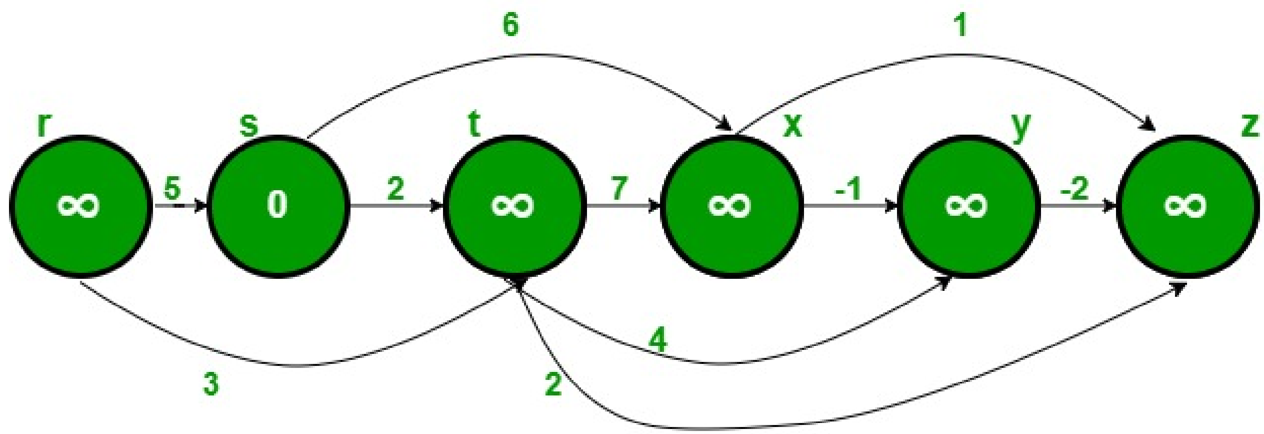
```
        For every adjacent vertex v of u
            if (dist[v] > dist[u] + weight(u, v)) //relax edge uv
                dist[v] = dist[u] + weight(u, v)
```

- Time Complexity: Time complexity of topological sort is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E), so the double for loop has complexity O(V+E). Therefore, overall time complexity is O(V+E).

# Topological Sort Example 1



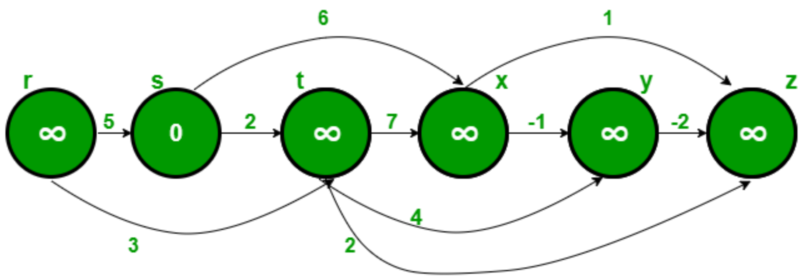

(a)



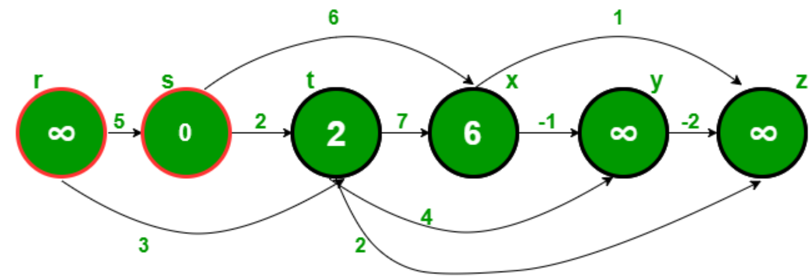

(b)

Initialization

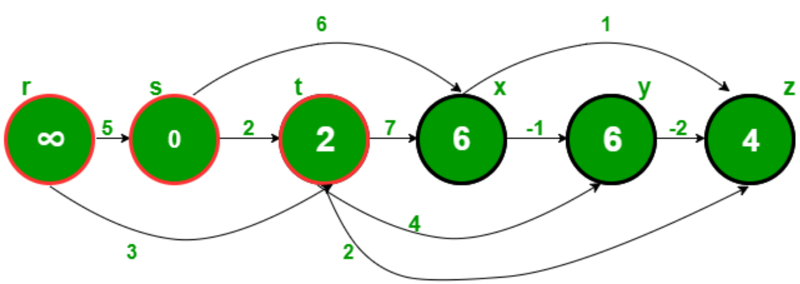
# Topological Sort Example 1



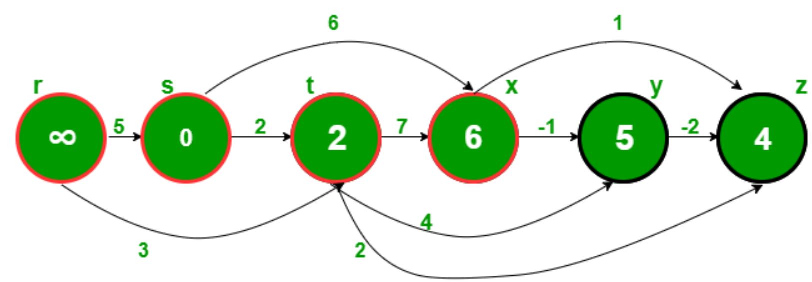
(c)



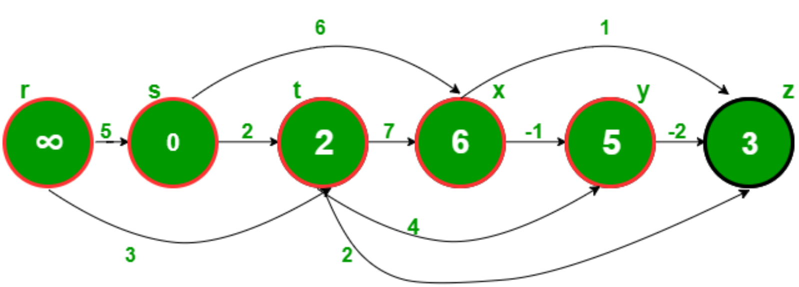
(d)



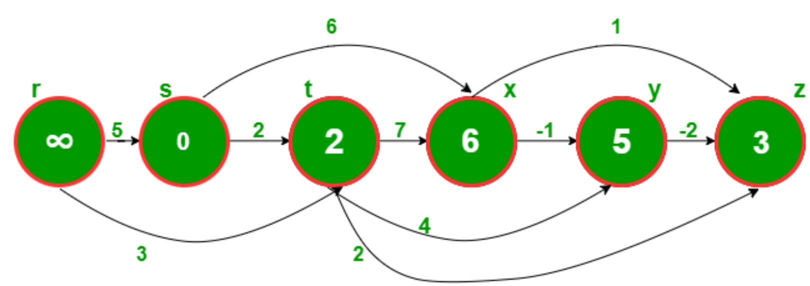
(e)



(f)



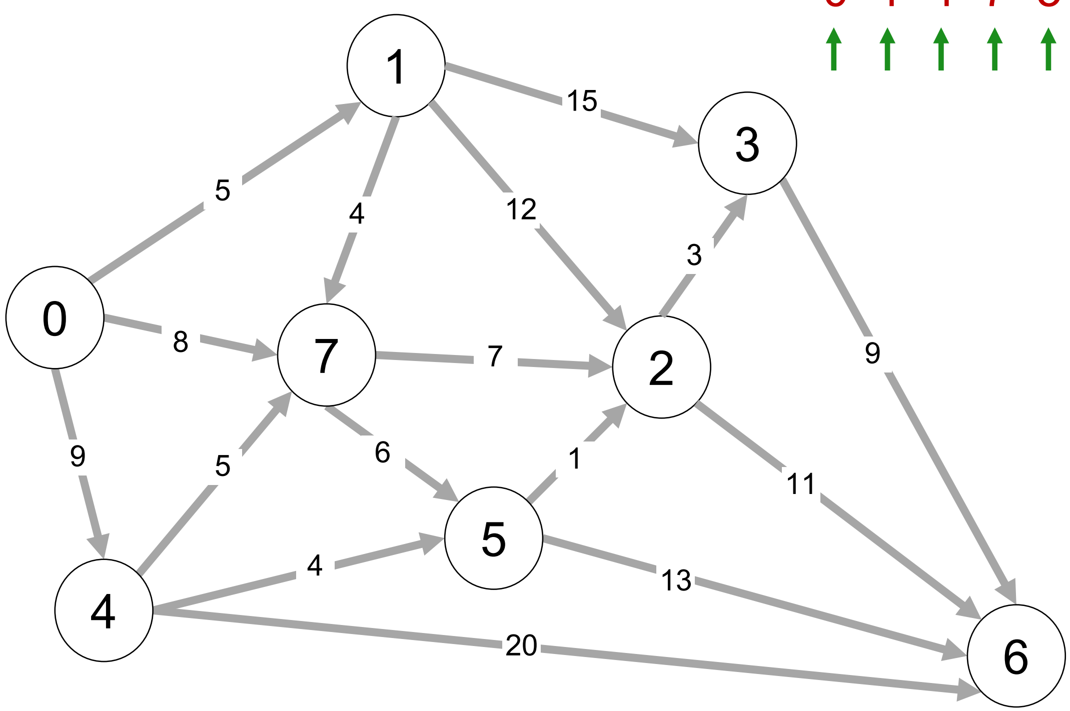
(g)



(h)

# Topological Sort Example 2

Yes!




0 1 4 7 5 2 3 6

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

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

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

# Shortest Paths in Edge-weighted DAG: Correctness Proof

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to E + V.

edge weights
can be negative!

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[v] will not change
- Thus, upon termination, shortest-paths optimality conditions hold.

distTo[ ] values are monotone decreasing

because of topological order, no edge
pointing to v will be relaxed after v is relaxed

# Single Source Shortest-paths Algorithms Summary

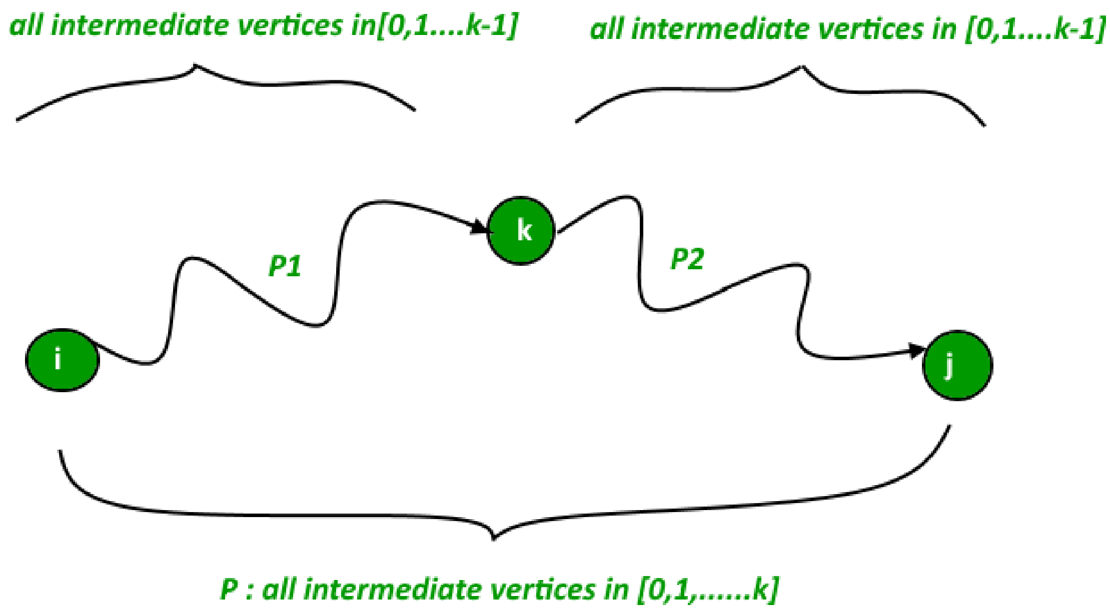
| Algorithm | Restriction | Worst-Case Complexity |
|---|---|---|
| Dijkstra (binary heap) | Undirected or directed graph; no negative weights/cycles | O(E log V) |
| Bellman-Ford | Directed graph with negative weights; undirected graph with no negative weights (since a negative weight edge forms a negative cycle by itself) | O(EV) |
| Topological Sort | Directed Acyclic Graph (DAG) (no cycles) | O(E+V) |

# Floyd Warshall Algorithm for all-pairs shortest paths

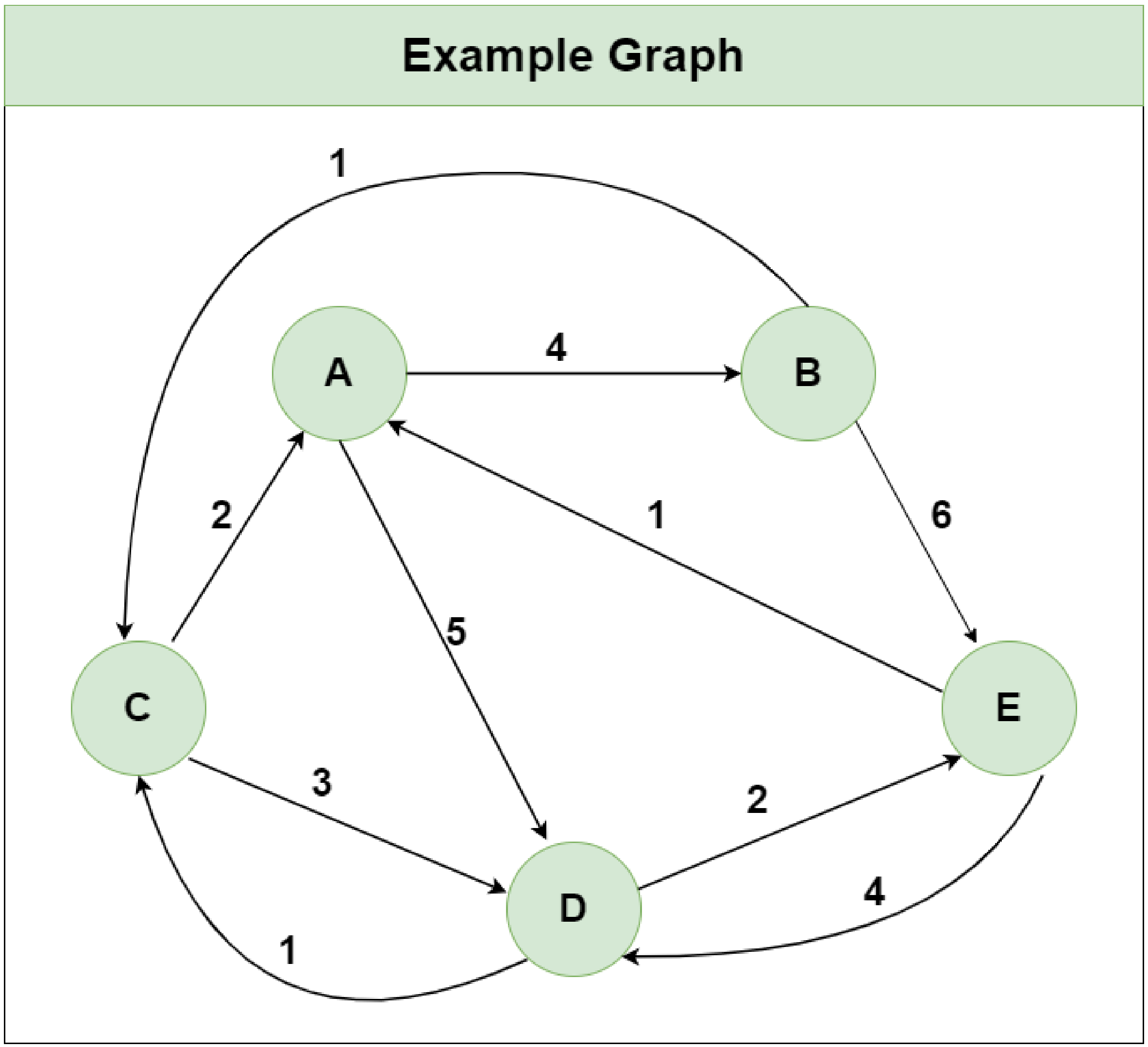
- The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms.
- It works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles
- It follows Dynamic Programming approach to check every possible path going via every possible vertex in order to calculate shortest distance between every pair of vertices.
- *For k = 0 to n – 1*
  *For i = 0 to n – 1*
  *For j = 0 to n – 1*
  *Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])*
- *where i = source vertex, j = Destination vertex, k = Intermediate vertex*
- Time Complexity: $O(V^3)$, where V is the number of vertices in the graph and we run three nested loops each of size V

# Floyd Warshall Algorithm is Dynamic Programming

- Floyd Warshall Algorithm is a Dynamic Programming based algorithm. It finds all pairs shortest paths using following recursive nature of problem. For every pair (i, j) of source and destination vertices respectively, there are two possible cases. 1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is. 2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j]. The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.
- Since there are overlapping subproblems in recursion, it uses dynamic programming



# Floyd Warshall Algorithm Example

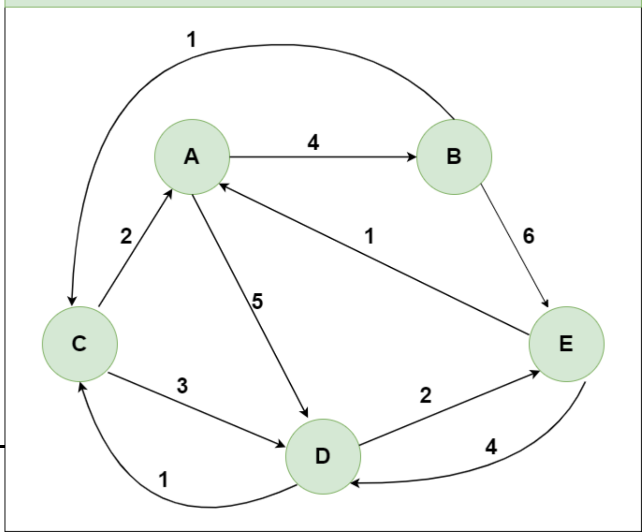


- ***<u>Step 1</u>:*** *Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from **i** to **j**, also Distance[i][j] = Infinity if there is no edge from **i** to **j**.*

**Step1: Initializing Distance[ ][ ] using the Input Graph**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | 0 | 4 | ∞ | 5 | ∞ |
| **B** | ∞ | 0 | 1 | ∞ | 6 |
| **C** | 2 | ∞ | 0 | 3 | ∞ |
| **D** | ∞ | ∞ | 1 | 0 | 2 |
| **E** | 1 | ∞ | ∞ | 4 | 0 |

**Example Graph**



- *__Step 2__: Treat vertex A as an intermediate vertex and calculate the Distance[][] for every {i,j} vertex pair using the formula:*
- *= Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])*

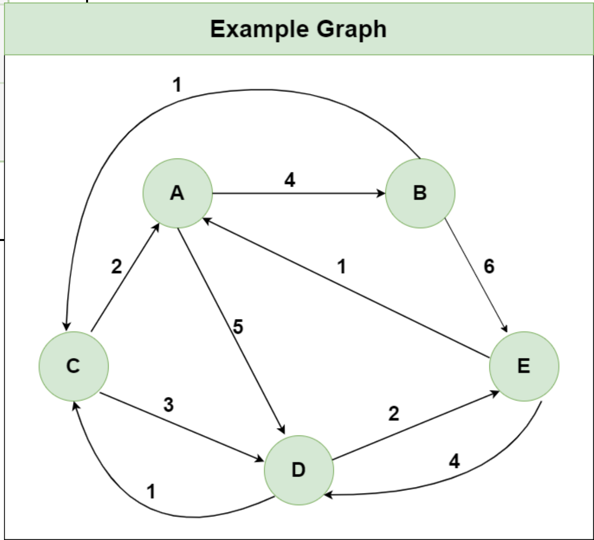
**Step 2: Using Node A as the Intermediate node**

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 4 | ∞ | 5 | ∞ |
| B | ∞ | ? | ? | ? | ? |
| C | 2 | ? | ? | ? | ? |
| D | ∞ | ? | ? | ? | ? |
| E | 1 | ? | ? | ? | ? |

→

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 4 | ∞ | 5 | ∞ |
| B | ∞ | 0 | 1 | ∞ | 6 |
| C | 2 | 6 | 0 | 3 | 12 |
| D | ∞ | ∞ | 1 | 0 | 2 |
| E | 1 | 5 | ∞ | 4 | 0 |



- ***Step 3***: *Treat vertex **B** as an intermediate vertex and calculate the Distance[][] for every {i,j} vertex pair using the formula:*
  *= Distance[i][j] = minimum (Distance[i][j], Distance[i][**B**] + Distance[**B**][j])*

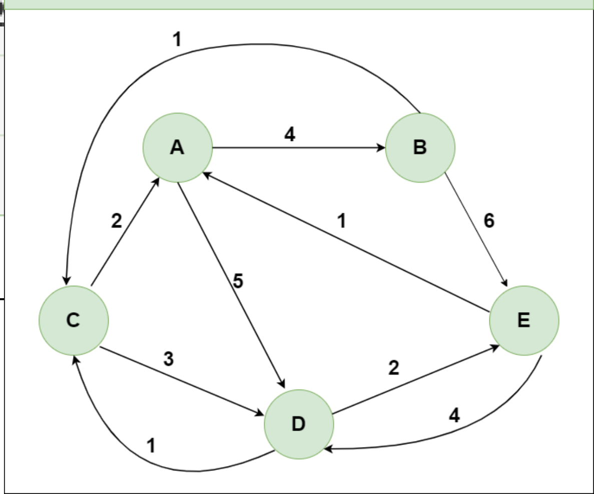
**Step 3: Using Node B as the Intermediate node**

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | ? | 4 | ? | ? | ? |
| **B** | ∞ | 0 | 1 | ∞ | 6 |
| **C** | ? | 6 | ? | ? | ? |
| **D** | ? | ∞ | ? | ? | ? |
| **E** | ? | 5 | ? | ? | ? |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | 0 | 4 | 5 | 5 | 10 |
| **B** | ∞ | 0 | 1 | ∞ | 6 |
| **C** | 2 | 6 | 0 | 3 | 12 |
| **D** | ∞ | ∞ | 1 | 0 | 2 |
| **E** | 1 | 5 | 6 | 4 | 0 |

**Example Graph**



- ***Step 4***: *Treat vertex **C** as an intermediate vertex and calculate the Distance[][] for every {i,j} vertex pair using the formula:*
  *= Distance[i][j] = minimum (Distance[i][j], Distance[i][**C**] + Distance[**C**][j])*

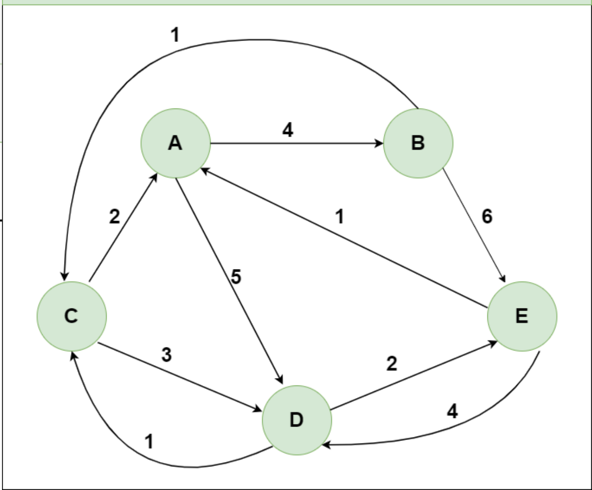
**Step 4: Using Node C as the Intermediate node**

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | ? | ? | 5 | ? | ? |
| **B** | ? | ? | 1 | ? | ? |
| **C** | 2 | 6 | 0 | 3 | 12 |
| **D** | ? | ? | 1 | ? | ? |
| **E** | ? | ? | 6 | ? | ? |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | 0 | 4 | 5 | 5 | 10 |
| **B** | 3 | 0 | 1 | 4 | 6 |
| **C** | 2 | 6 | 0 | 3 | 12 |
| **D** | 3 | 7 | 1 | 0 | 2 |
| **E** | 1 | 5 | 6 | 4 | 0 |

**Example Graph**



- ***Step 5***: *Treat vertex **D** as an intermediate vertex and calculate the Distance[][] for every {i,j} vertex pair using the formula:*
  *= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])*

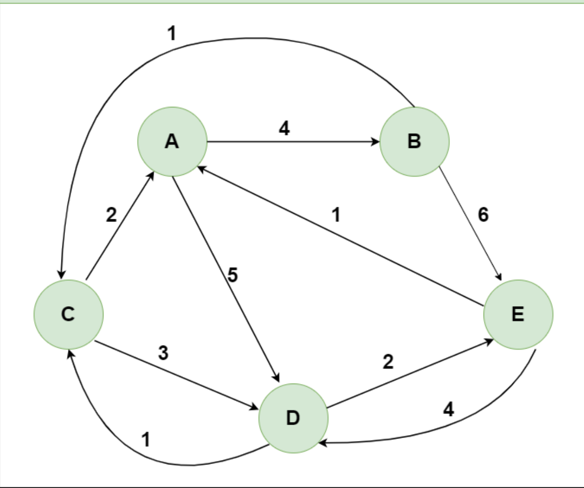
**Step 5: Using Node D as the Intermediate node**

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | ? | ? | ? | 5 | ? |
| B | ? | ? | ? | 4 | ? |
| C | ? | ? | ? | 3 | ? |
| D | 3 | 7 | 1 | 0 | 2 |
| E | ? | ? | ? | 4 | ? |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 4 | 5 | 5 | 7 |
| B | 3 | 0 | 1 | 4 | 6 |
| C | 2 | 6 | 0 | 3 | 5 |
| D | 3 | 7 | 1 | 0 | 2 |
| E | 1 | 5 | 5 | 4 | 0 |

**Example Graph**



- ***Step 6***: *Treat vertex **E** as an intermediate vertex and calculate the Distance[][] for every {i,j} vertex pair using the formula:*
  *= Distance[i][j] = minimum (Distance[i][j], Distance[i][**E**] + Distance[**E**][j])*

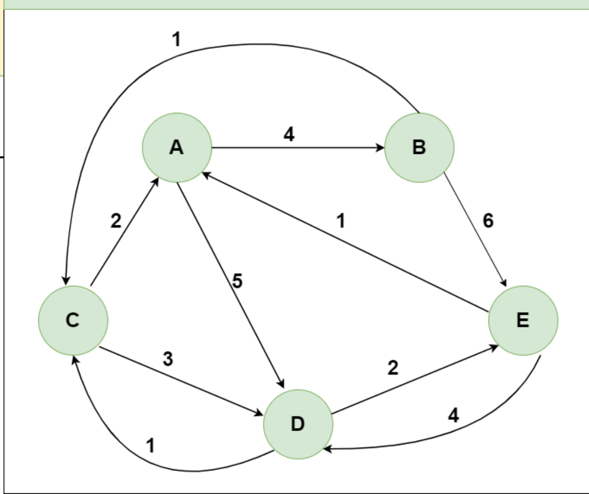
**Step 6: Using Node E as the Intermediate node**

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | ? | ? | ? | ? | 7 |
| B | ? | ? | ? | ? | 6 |
| C | ? | ? | ? | ? | 5 |
| D | ? | ? | ? | ? | 2 |
| E | 1 | 5 | 5 | 4 | 0 |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 4 | 5 | 5 | 7 |
| B | 3 | 0 | 1 | 4 | 6 |
| C | 2 | 6 | 0 | 3 | 5 |
| D | 3 | 7 | 1 | 0 | 2 |
| E | 1 | 5 | 5 | 4 | 0 |

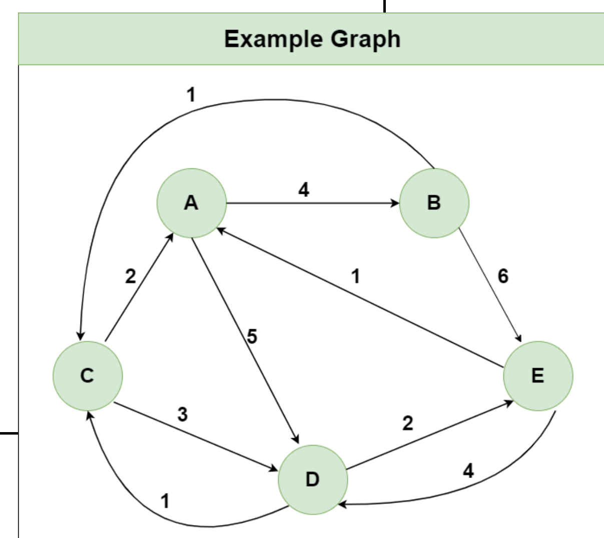
**Example Graph**



- ***Step 7***: *Since all the vertices have been treated as an intermediate vertex, we can now return the updated Distance[][] matrix as our answer matrix.*

**Step 7: Return Distance[ ][ ] matrix as the result**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | 0 | 4 | 5 | 5 | 7 |
| **B** | 3 | 0 | 1 | 4 | 6 |
| **C** | 2 | 6 | 0 | 3 | 5 |
| **D** | 3 | 7 | 1 | 0 | 2 |
| **E** | 1 | 5 | 5 | 4 | 0 |



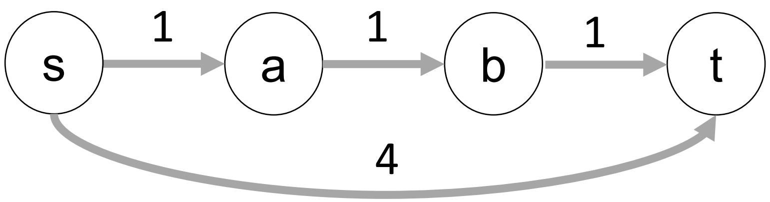
# Johnson’s Algorithm for all-pairs shortest paths

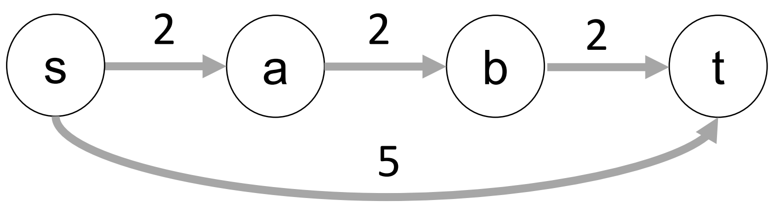
- Johnson’s algorithm uses both Dijkstra and Bellman-Ford as subroutines. If we apply Dijkstra’s Single Source shortest path algorithm for every vertex, considering every vertex as the source, we can find all pair shortest paths in O(V*VLogV) time.
- Dijkstra’s algorithm doesn’t work for negative weight edge. The idea of Johnson’s algorithm is to re-weight all edges and make them all positive, then apply Dijkstra’s algorithm for every vertex.
- How to transform a given graph into a graph with all non-negative weight edges?

https://www.geeksforgeeks.org/johnsons-algorithm/

# Increase weight of every edge by a constant?

- True or False: In a weighted graph, assume that the shortest path from source s to destination t is correctly calculated using a shortest path algorithm. If we increase weight of every edge by a constant, the shortest path always remains same.
- False. See the following counterexample. There are 4 edges s-a, a-b, b-t and s-t of wights 1, 1, 1 and 4 respectively. The shortest path from s to t is s-a, a-b, b-t=3. If we increase weight of every edge by 1, the shortest path changes to s-t = 5.



# Double the original weights?

- True or False: Is the following statement valid about shortest paths? Given a graph, suppose we have calculated shortest path from a source to all other vertices. If we modify the graph such that weights of all edges is becomes double of the original weight, then the shortest path remains same only the total weight of path changes.
- True. The shortest path remains same. It is like if we change unit of distance from meter to kilo meter, the shortest paths don\'t change. But this does not make weights positive.

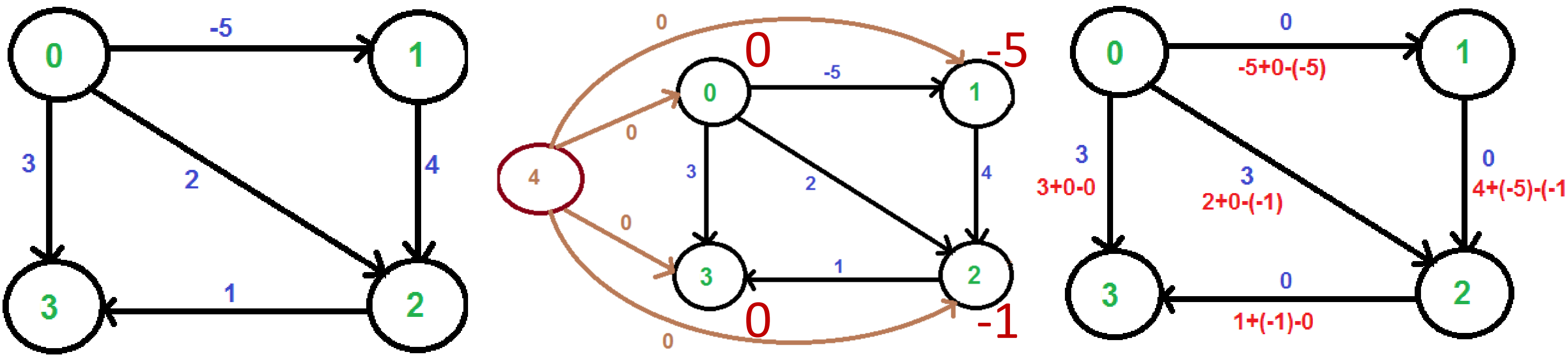
# Johnson's algorithm for All-pairs shortest paths

1. Let the given graph be G. Add a new vertex s to the graph, add edges from the new vertex to all vertices of G. Let the modified graph be G'.
2. Run the Bellman-Ford algorithm on G' with s as the source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return.
3. Reweight the edges of the original graph. For each edge (u, v), assign the new weight as "original weight + h[u] – h[v]".
4. Remove the added vertex s and run Dijkstra's algorithm for every vertex.

Time complexity: The main steps in the algorithm are Bellman-Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(V Log V). So overall time complexity is $O(V^2 \log V + VE)$.

# Johnson's Algorithm Example

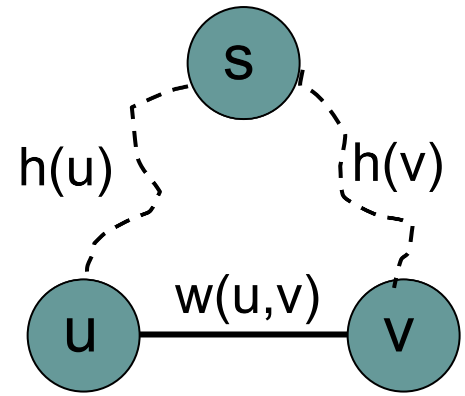
- We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.
- We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e., h[] = {0, -5, -1, 0}. Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula. w(u, v) = w(u, v) + h[u] – h[v].
- Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as the source.




Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectiviely.

# Johnson’s Algorithm: Proof

- The following property is always true about h[] values as they are the shortest distances.
- h[v] <= h[u] + w(u, v)
- The property simply means that the shortest distance from s to v must be smaller than or equal to the shortest distance from s to u plus the weight of the edge (u, v). The new weights are w(u, v) + h[u] – h[v]. The value of the new weights must be greater than or equal to zero because of the inequality “h[v] <= h[u] + w(u, v)”.
- After reweighting, all set of paths between any two vertices is increased by the same amount and all negative weights become non-negative. Consider any path between two vertices s and t, the weight of every path is increased by h[s] – h[t], and all h[] values of vertices on the path from s to t cancel each other.



# 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
- Shortest Path Algorithms Explained (Dijkstra's & Bellman-Ford https://www.youtube.com/watch?v=AE5I0xACpZs
- Floyd–Warshall algorithm in 4 minutes
  - https://www.youtube.com/watch?v=4OQeCuLYj-4

# Tutorials from Geeksforgeeks

- https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/
- https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
- https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/
- https://www.geeksforgeeks.org/johnsons-algorithm/

# 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
- ANS: Topological Sort
- Topological Sort has complexity O(V+E), which is the most efficient algorithm among the three

# 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.