Lecture 5.0
Shortest Paths

Department of Computer Science
Hofstra University

IMPORTANT

Lecture Goals

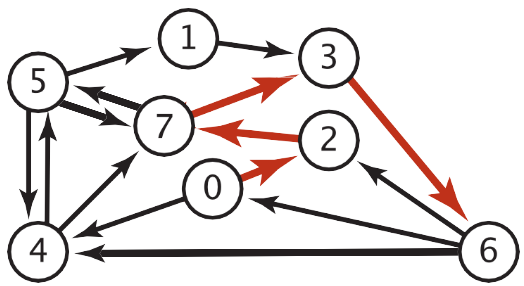
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- We introduce and analyze Dijkstra's algorithm for shortest-paths problems with nonnegative weights.
- We conclude with the Bellman–Ford algorithm for edge-weighted digraphs with no negative cycles.

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



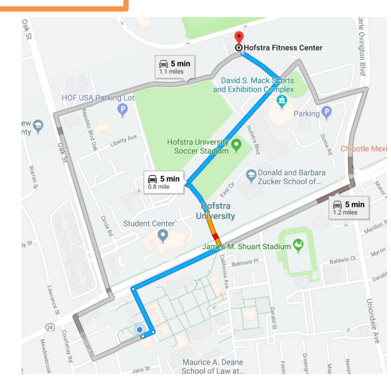
shortest path from 0 to 6

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.

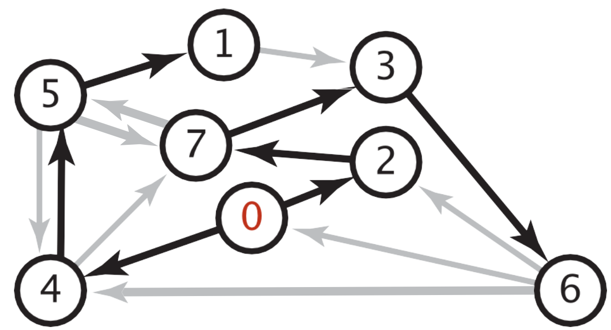
Data Structures for Single-source Shortest Paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists.

Consequence. Can represent the SPT with two vertex-indexed arrays:

- distTo[v] is length of shortest path from s to v.
- edgeTo[v] is last edge on shortest path from s to v.



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

```
public double distTo(int v)
{ return distTo[v]; }
```

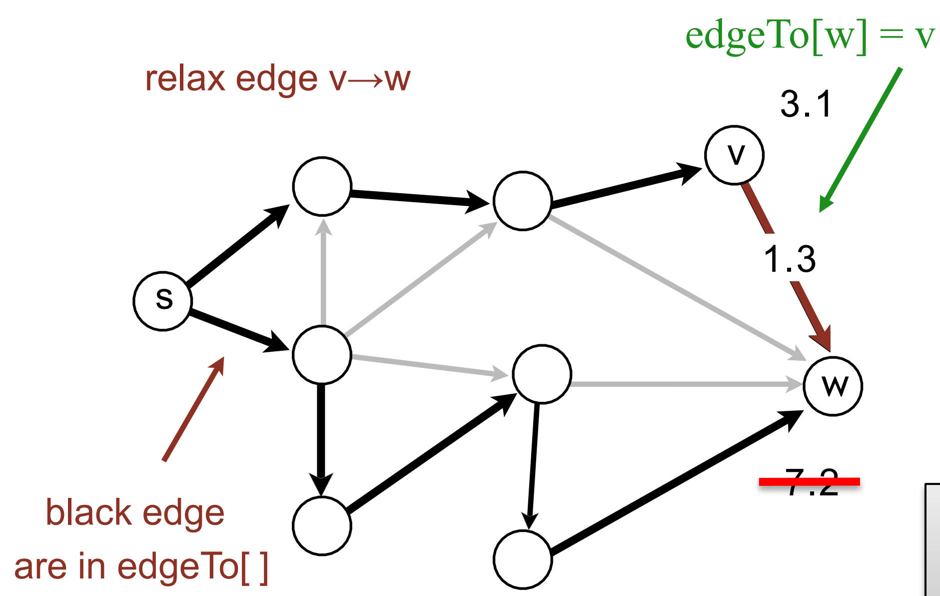
```
public Iterable<DirectedEdge> pathTo(int v)
{
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for  (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

IMPORTANT

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




```
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;
    }
}
```

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

IMPORTANT

Generic Shortest-paths Algorithm

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.

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 edgeTo[v] is last edge on 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. (nonnegative weights, directed cycles).
- Ex 2. Bellman–Ford algorithm. (no negative cycles).

IMPORTANT

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:
  - Extract the unvisited node with the smallest known distance from the queue.
  - For each neighboring node, calculate its tentative distance through the current node. If this distance is smaller than the previously recorded distance, update it with relaxation.
  - 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.

IMPORTANT

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.

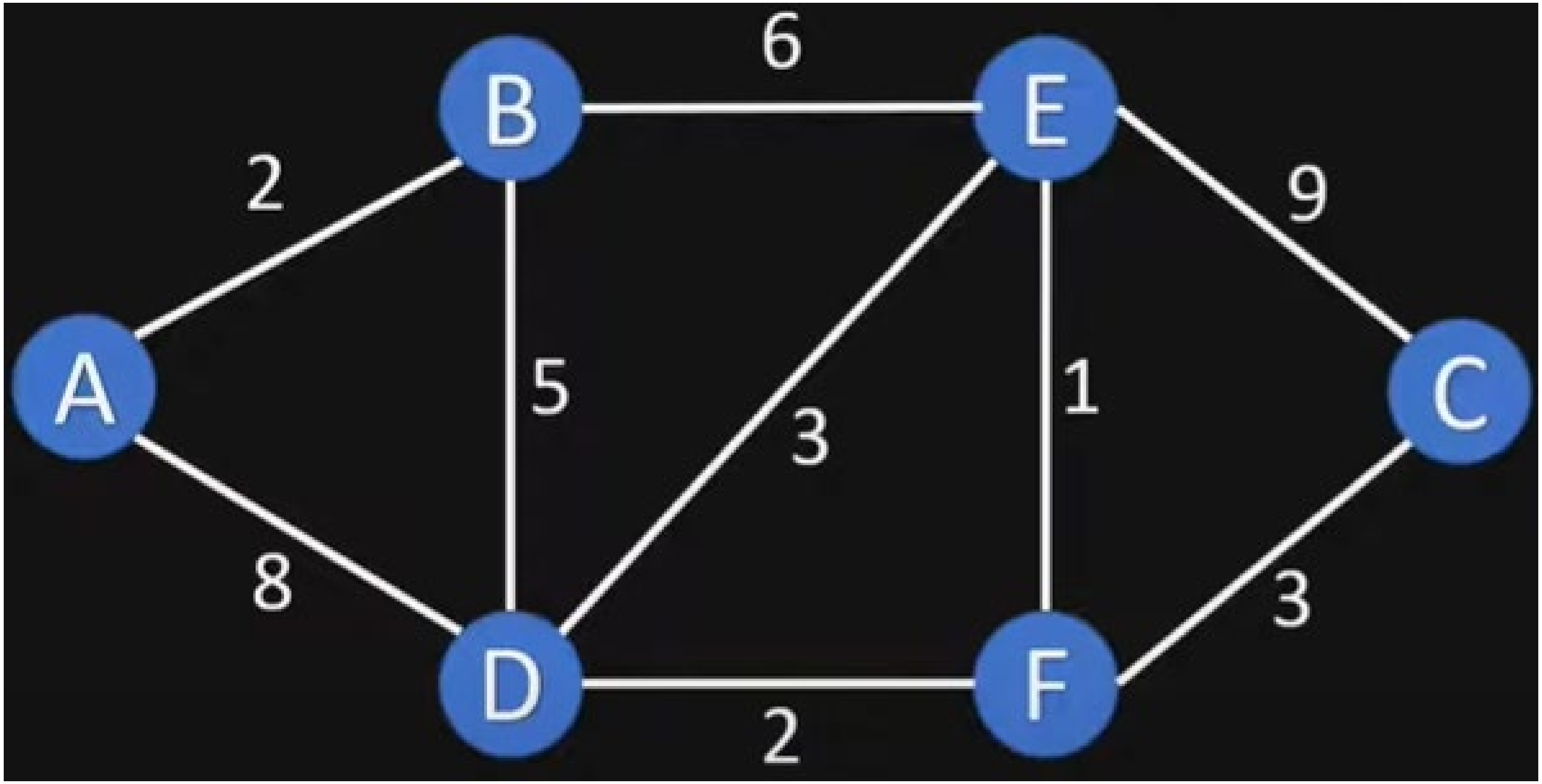
IMPORTANT

Video Tutorial

- The following lecture slides are based on this video:
- Dijkstras Shortest Path Algorithm Explained | With Example | Graph Theory
  - https://www.youtube.com/watch?v=bZkzH5x0SKU

IMPORTANT

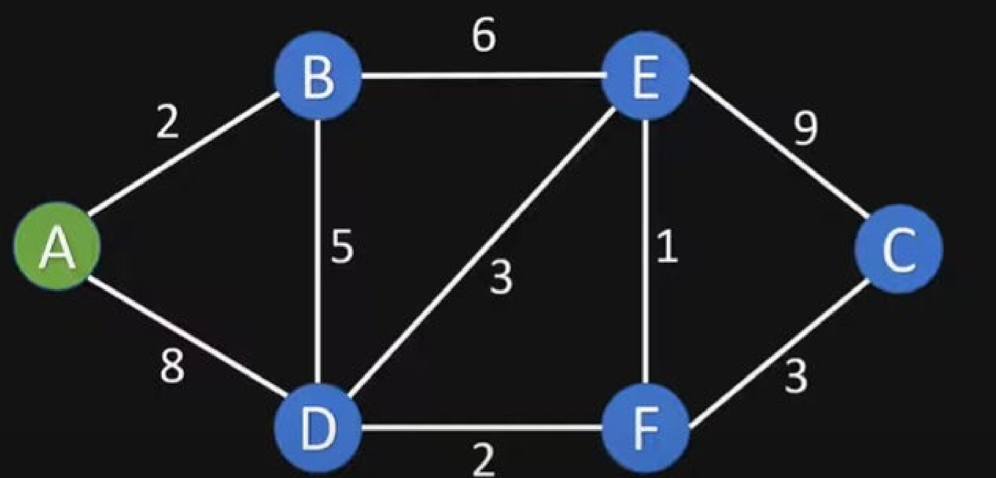
Example Graph



IMPORTANT

Initialize

2. Assign to all nodes a tentative distance value



Visited Nodes: []

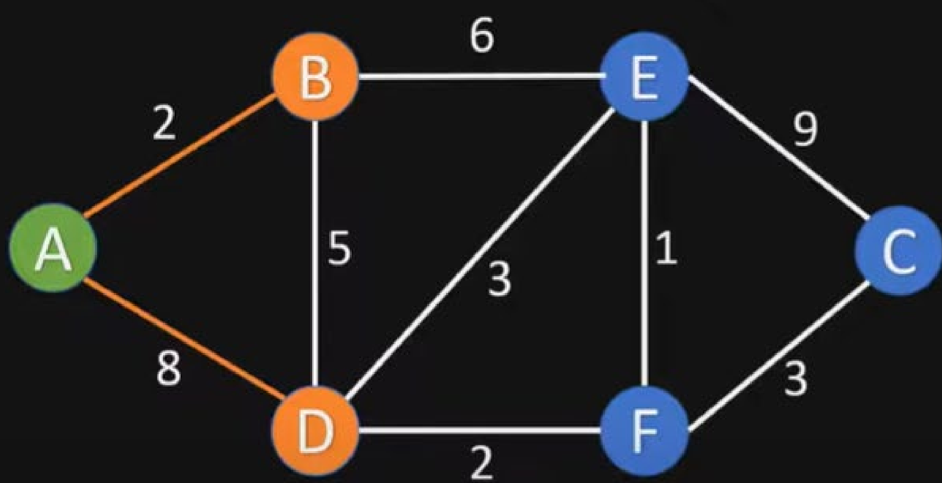
Unvisited Nodes: [A, B, C, D, E, F]

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

IMPORTANT

Visit Node A

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	**8**	A
E	∞	
F	∞	

Visited Nodes: []

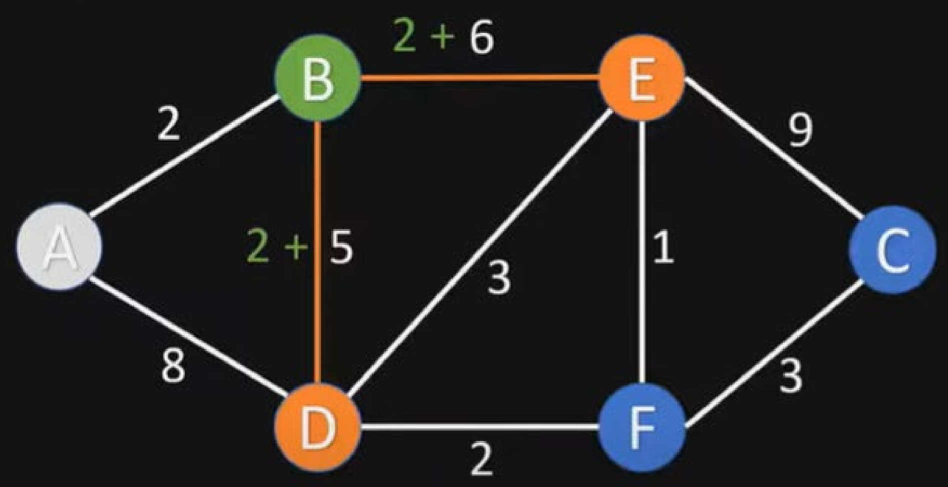
Unvisited Nodes: [A, B, C, D, E, F]

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

IMPORTANT

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	∞	

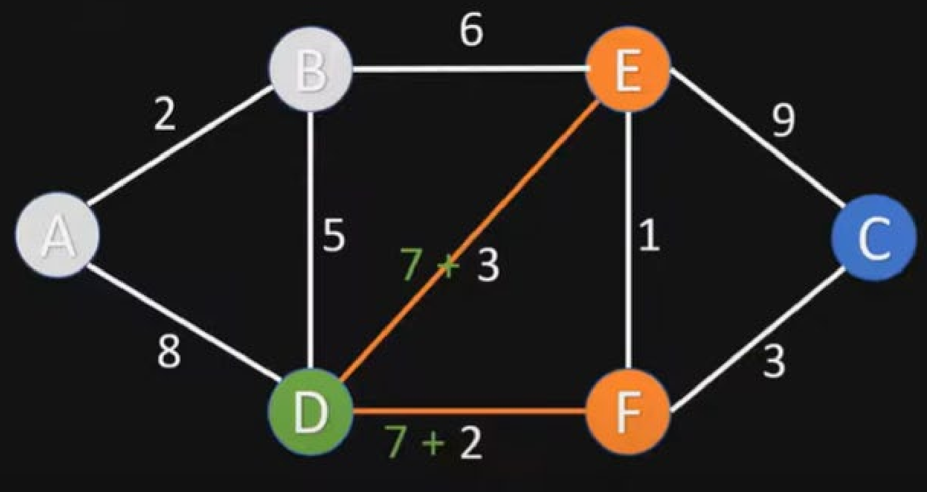
Visited Nodes: [A]   Unvisited Nodes: [B, C, D, E, F]

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

IMPORTANT

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

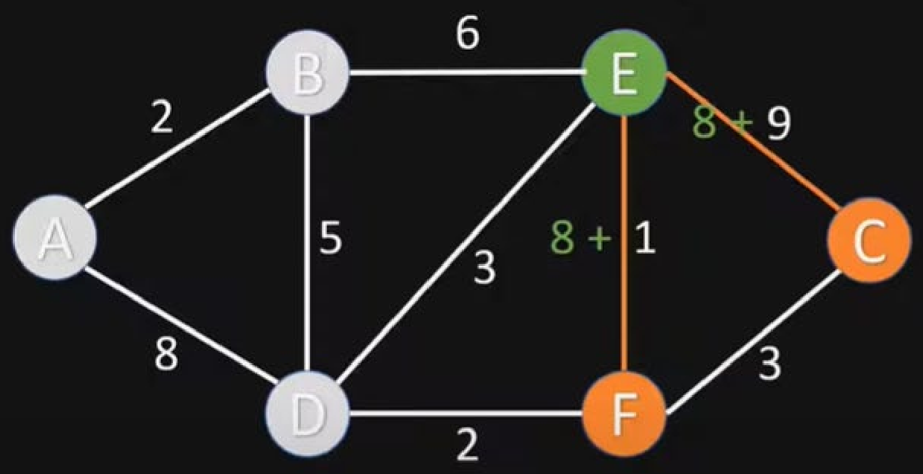
Visited Nodes: [A, B]   Unvisited Nodes: [C, D, E, F]

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

IMPORTANT

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



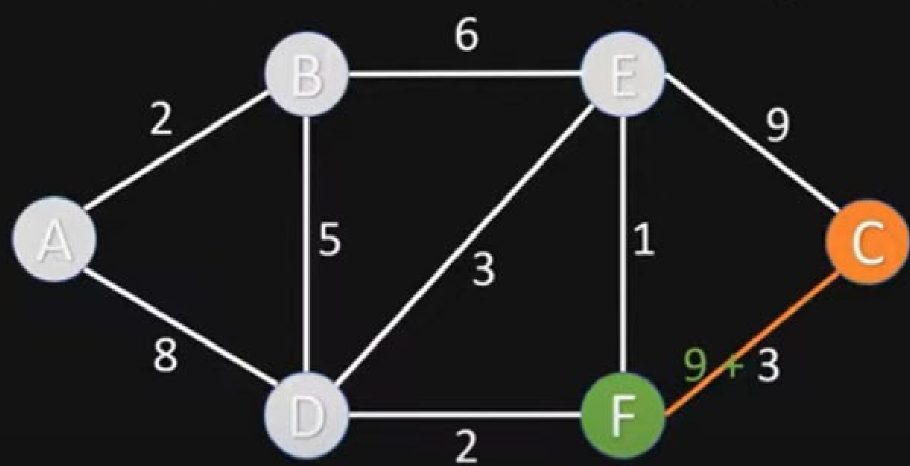
Visited Nodes: [A, B, D] Unvisited Nodes: [C, E, F]

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
OLD distTo[F] = 9 = distTo[E] + e[E][F].weight() = 8+1 = 9
No update, distTo[F] stays 9

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]

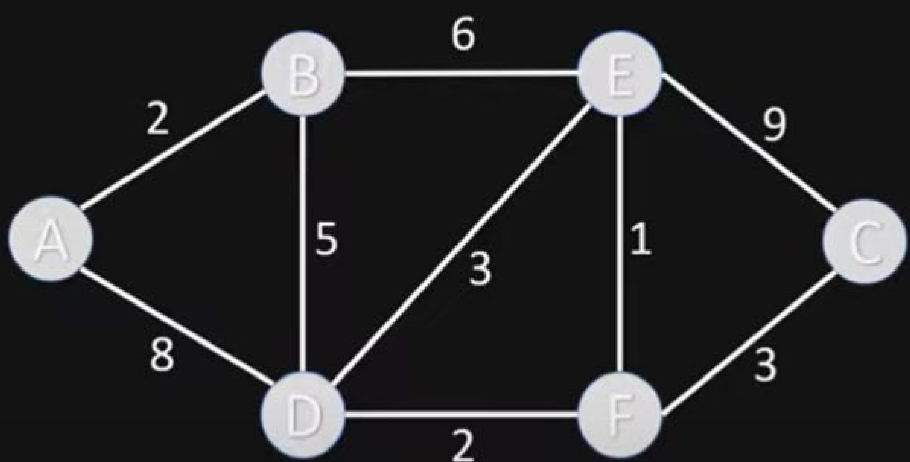
OLD distTo[C] = 17 > distTo[F] + e[F][C].weight() = 9+3 = 12
NEW distTo[C] ← distTo[F] + e[F][C].weight() = 12

IMPORTANT

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

4. Mark current node as visited



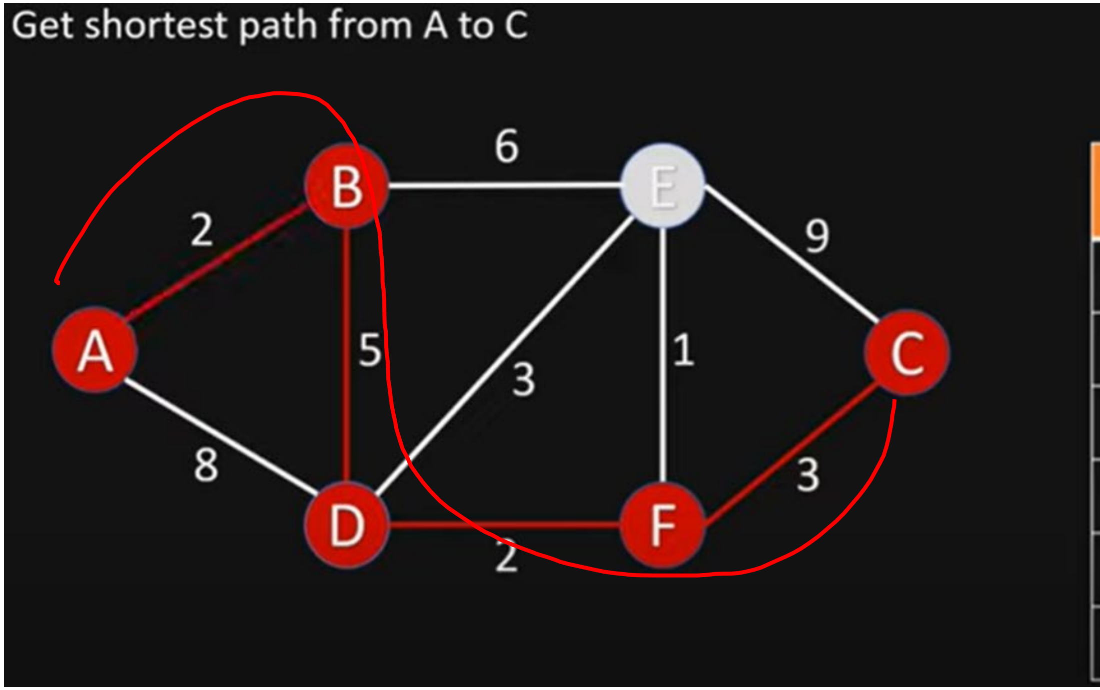
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

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

IMPORTANT

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

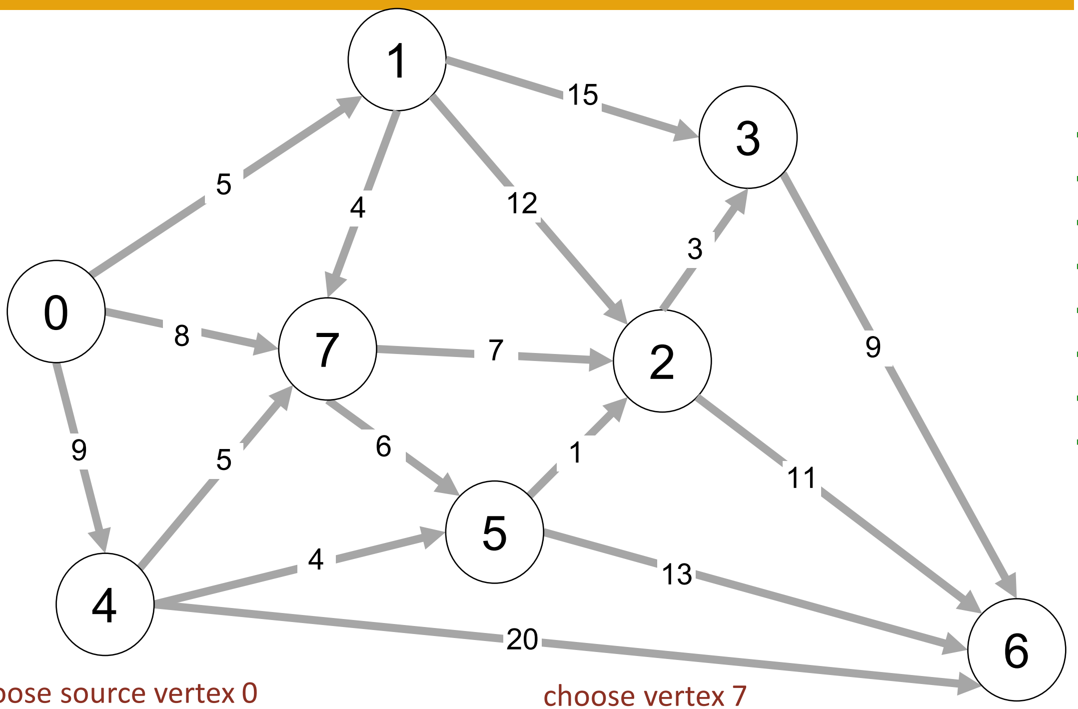


Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

IMPORTANT

Dijkstra's Algorithm Example 2

- 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

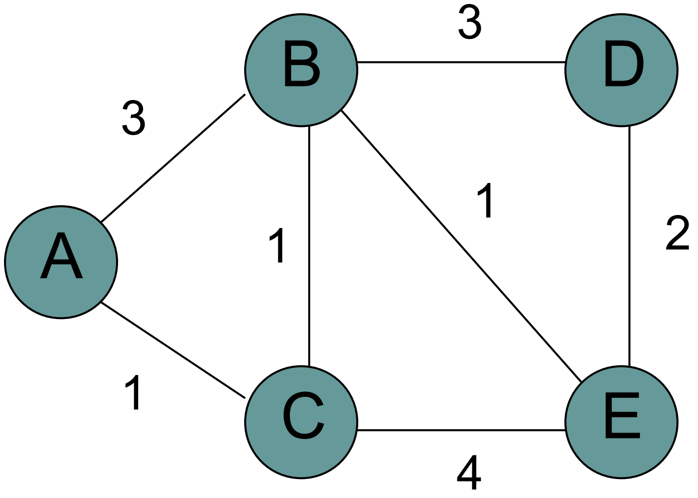
choose vertex 5
relax all edges adjacent from 5
choose vertex 2
relax all edges adjacent from 2
choose vertex 3
relax all edges adjacent from 3
choose vertex 6
relax all edges adjacent from 6

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		

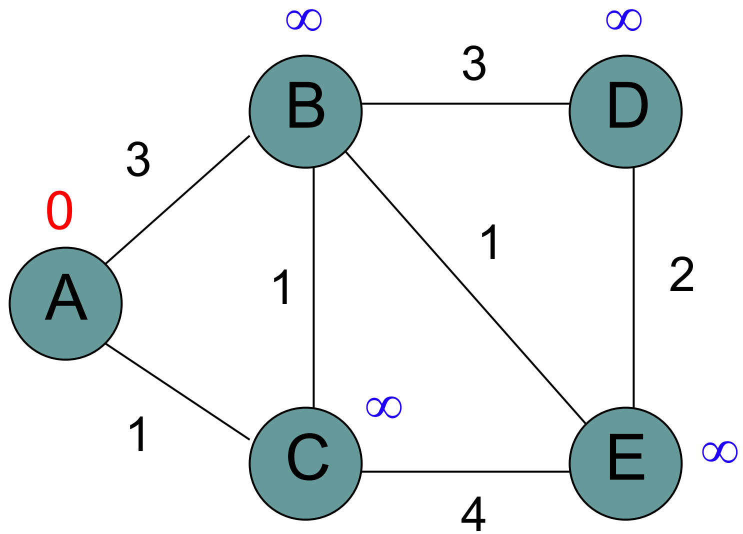
IMPORTANT

Dijkstra's Algorithm Example 3



IMPORTANT

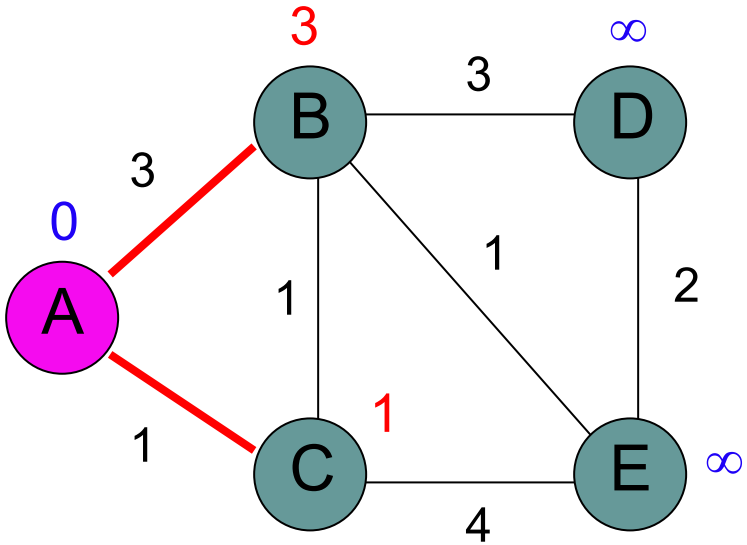
Initialize



N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

IMPORTANT

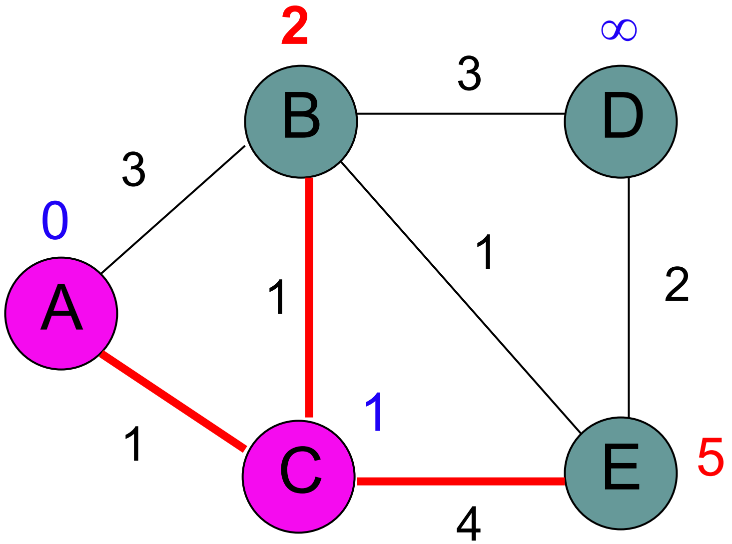
Visit Node A



N	SD	PN
A	0	
B	3	A
C	1	A
D	∞	
E	∞	

IMPORTANT

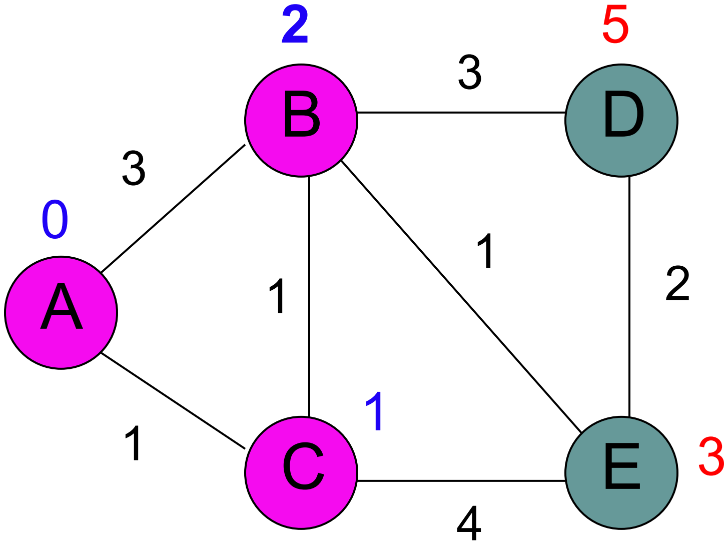
Visit Node C



N	SD	PN
A	0	
B	2	C
C	1	A
D	∞	
E	5	C

IMPORTANT

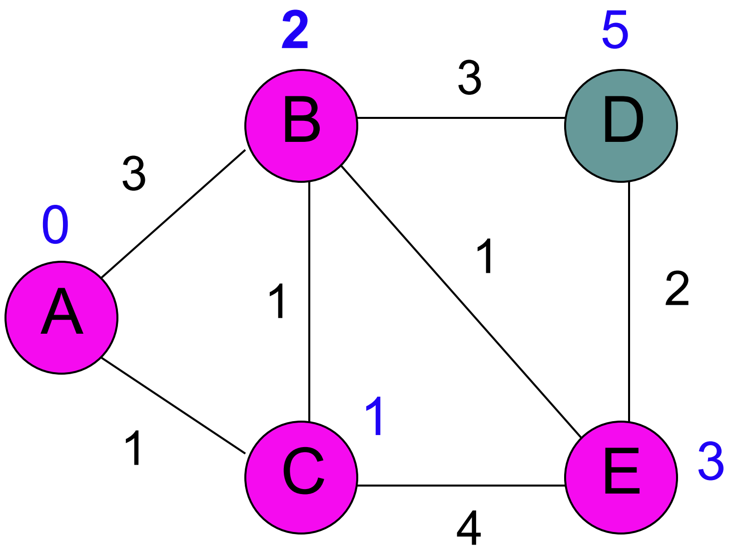
Visit Node B



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

IMPORTANT

Visit Node E

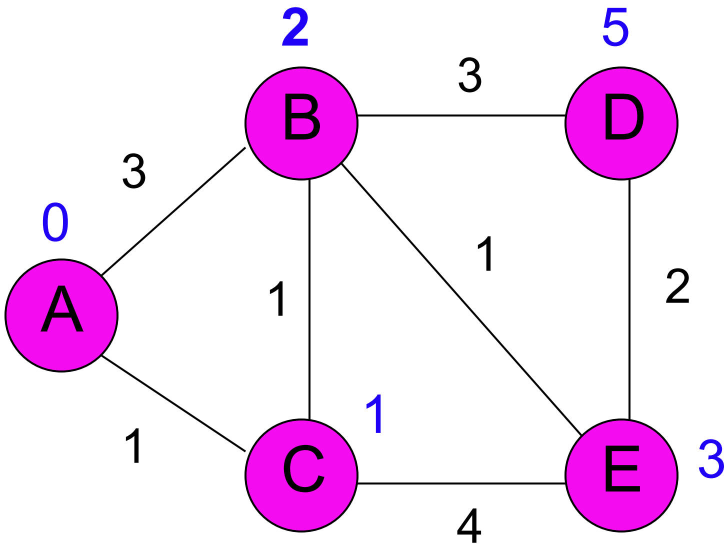


N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

Nothing changes

IMPORTANT

Visit Node D



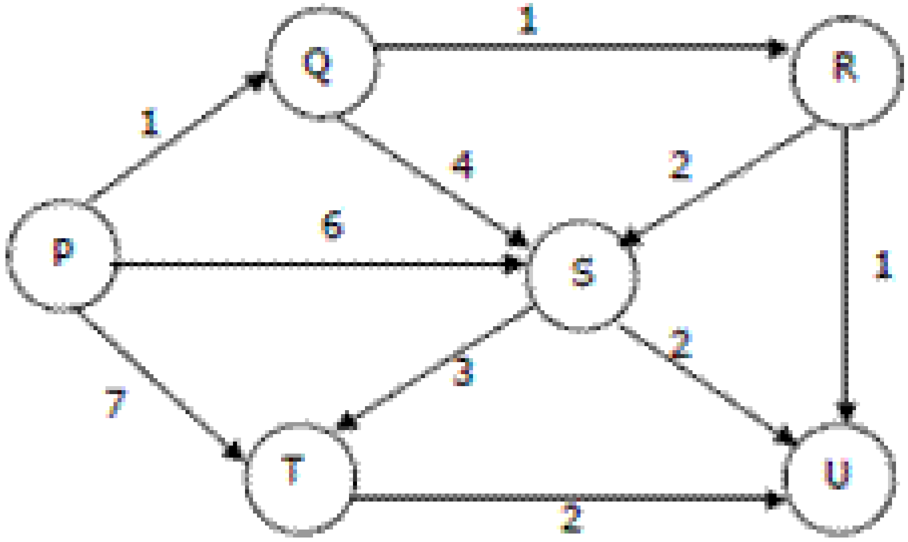
N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

Nothing changes

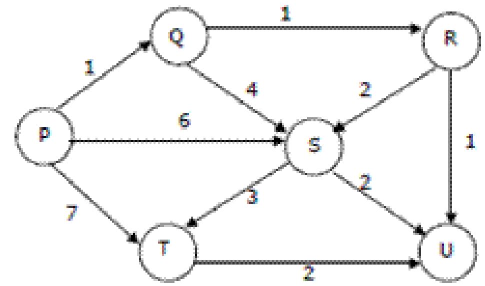
IMPORTANT

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

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

IMPORTANT

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.

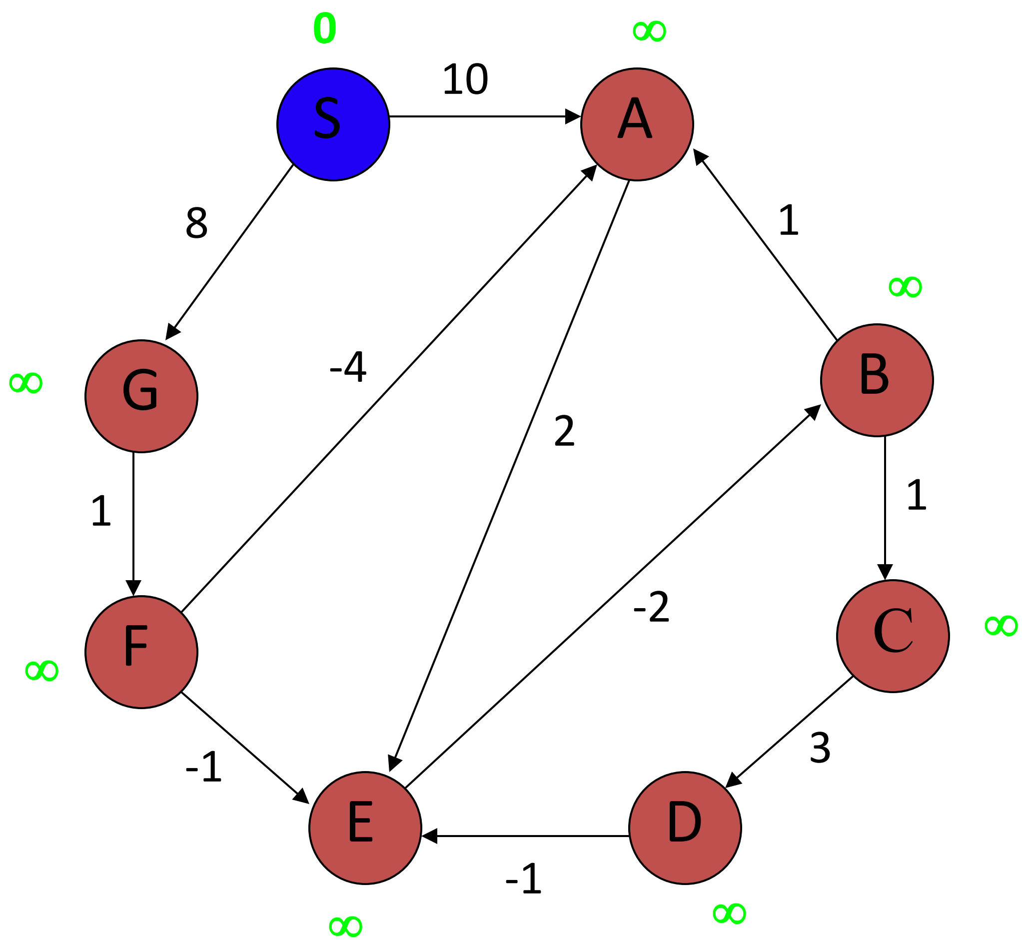
Bellman-Ford Algorithm with Negative Cycle Detection

- 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

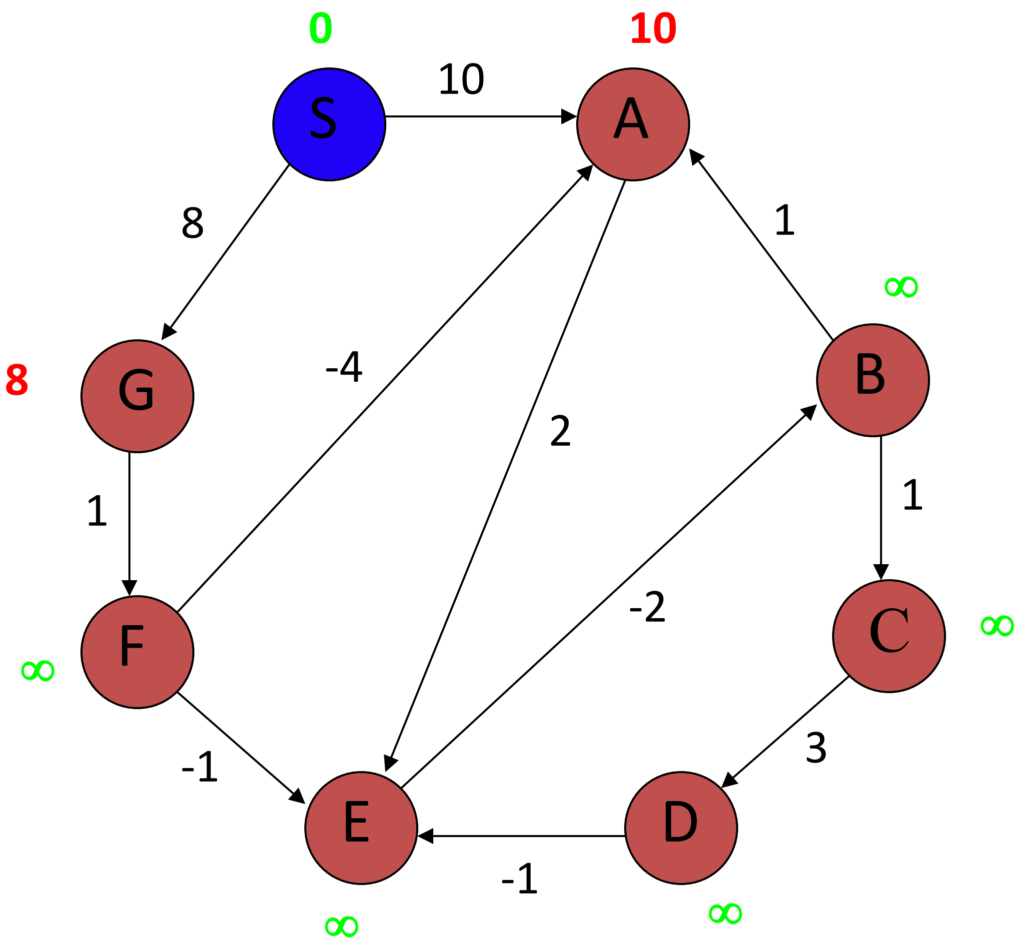
- Time complexity for connected graph:
- Best Case: O(|E|), when distance array after 1st and 2nd relaxation are same , we can simply stop further processing after one iteration
- Average Case: O(|V|*|E|)
- Worst Case: O(|V|*|E|)
  - If the graph is complete, the value of E becomes $O(|V|^2)$. So overall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

Bellman-Ford Algorithm Example 1



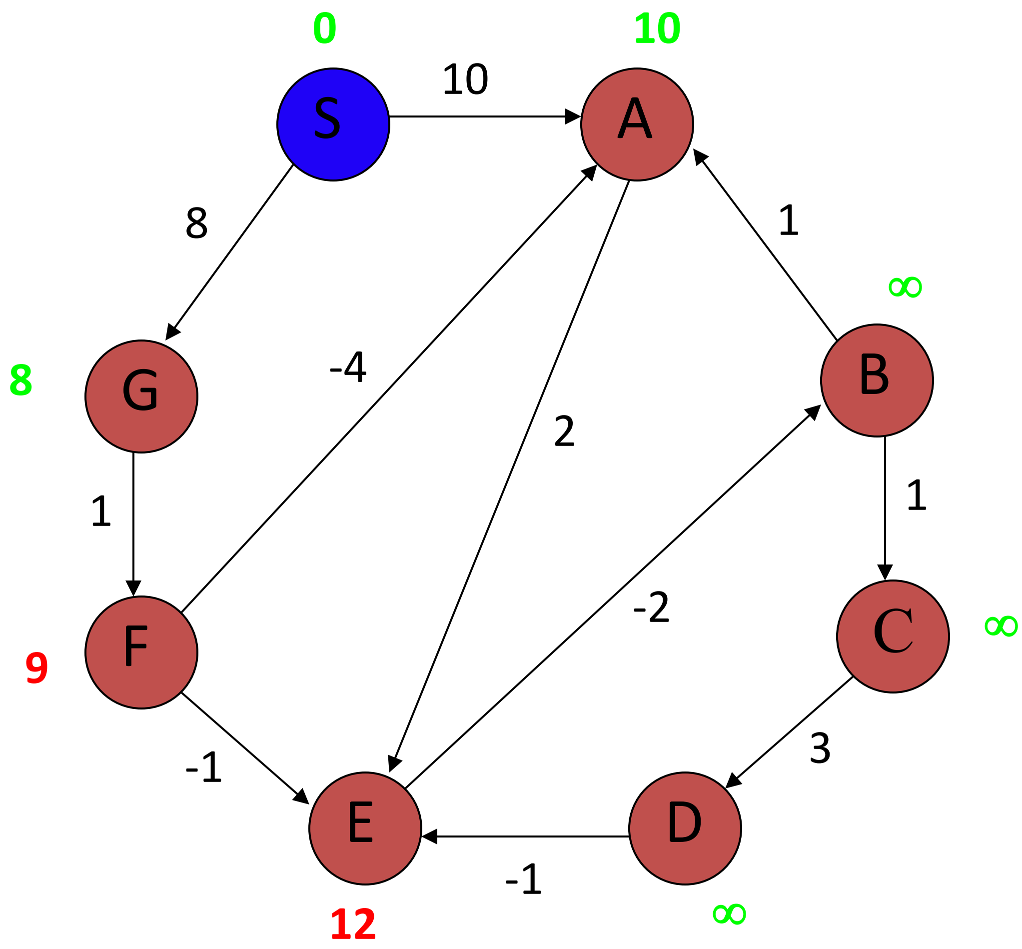
Iteration: 0

Bellman-Ford Algorithm Example 1



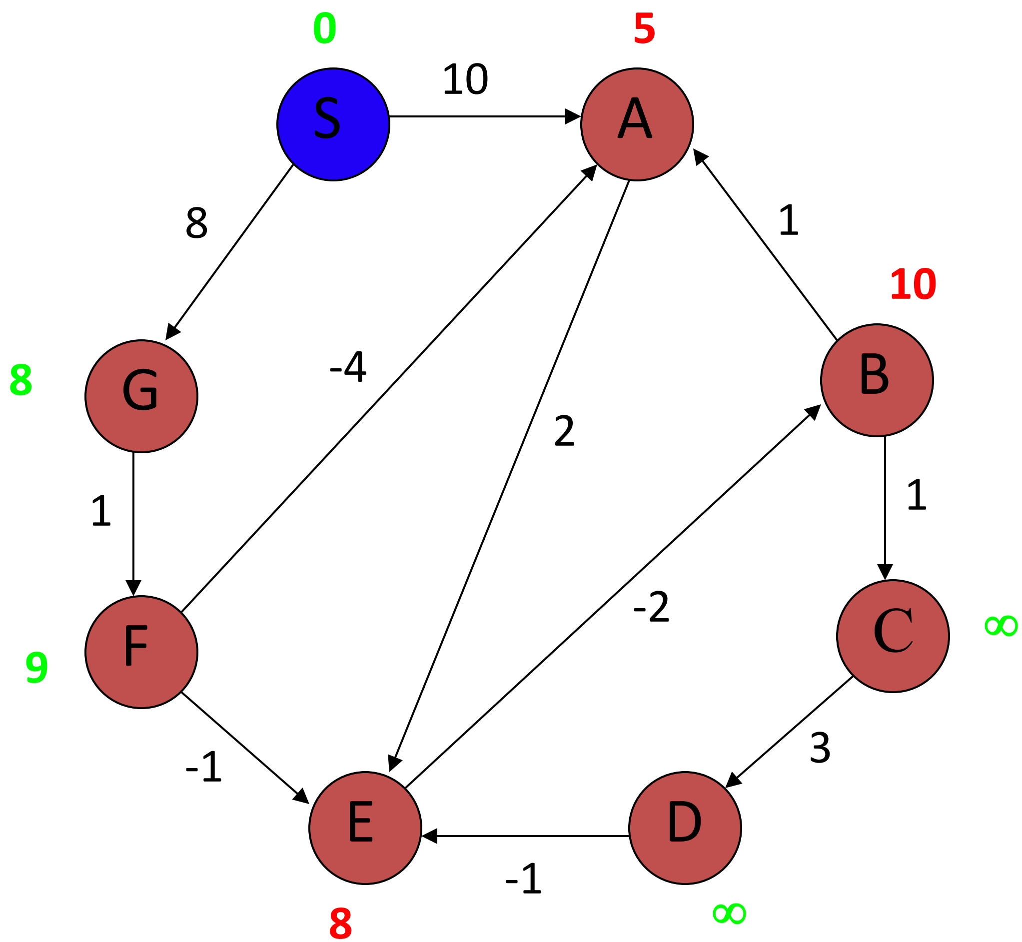
Iteration: 1

Bellman-Ford Algorithm Example 1



Iteration: 2

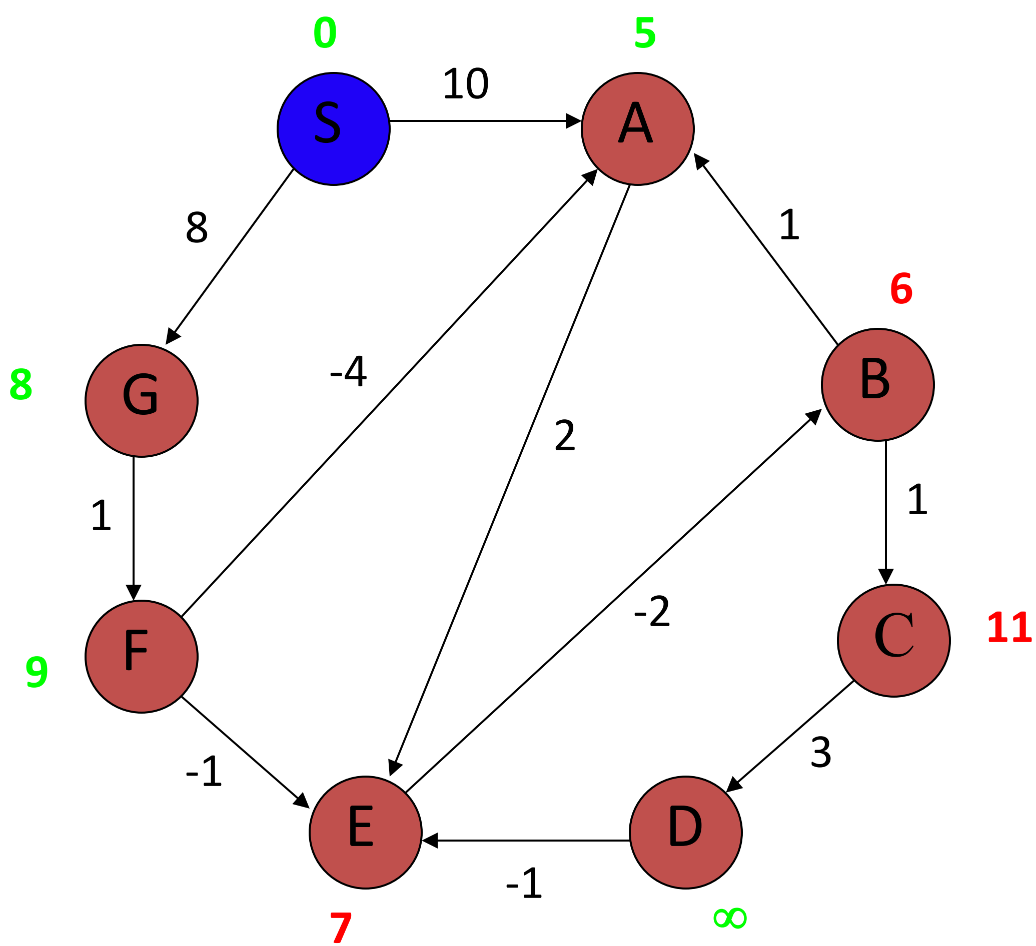
Bellman-Ford Algorithm Example 1



Iteration: 3

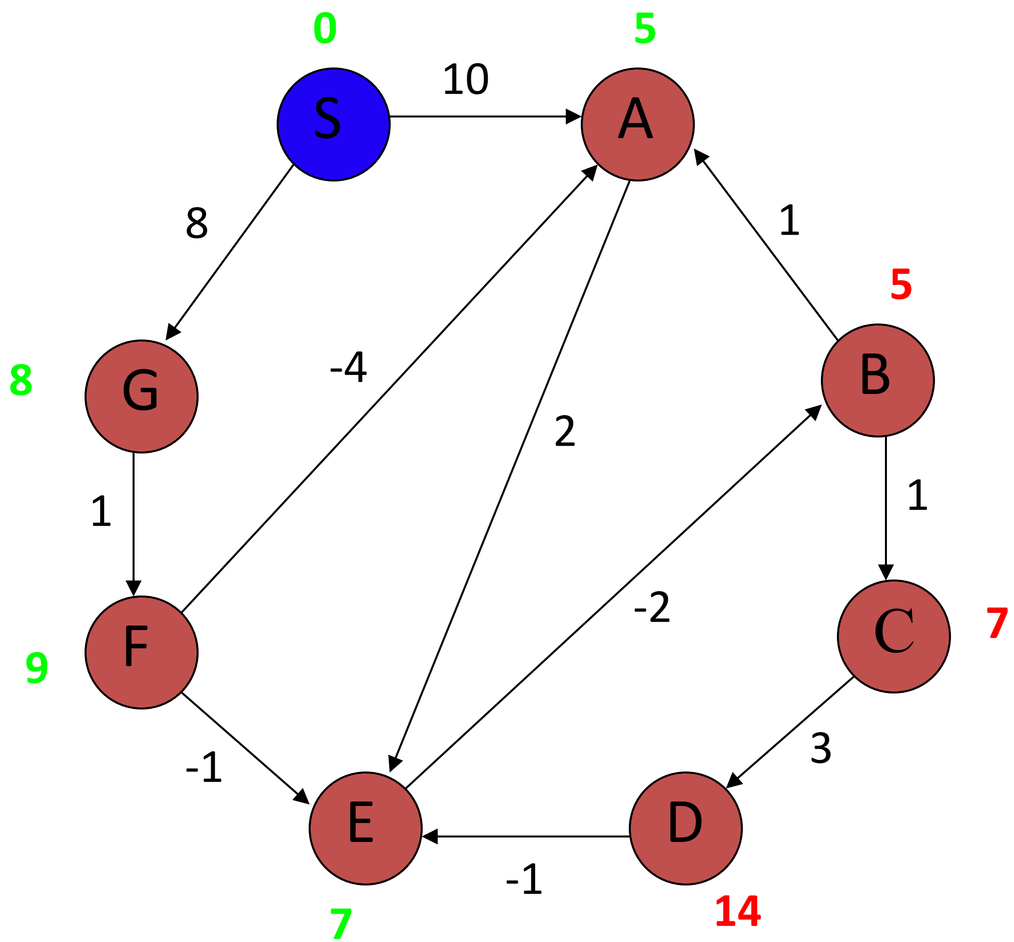
A has the correct distance and path

Bellman-Ford Algorithm Example 1



Iteration: 4

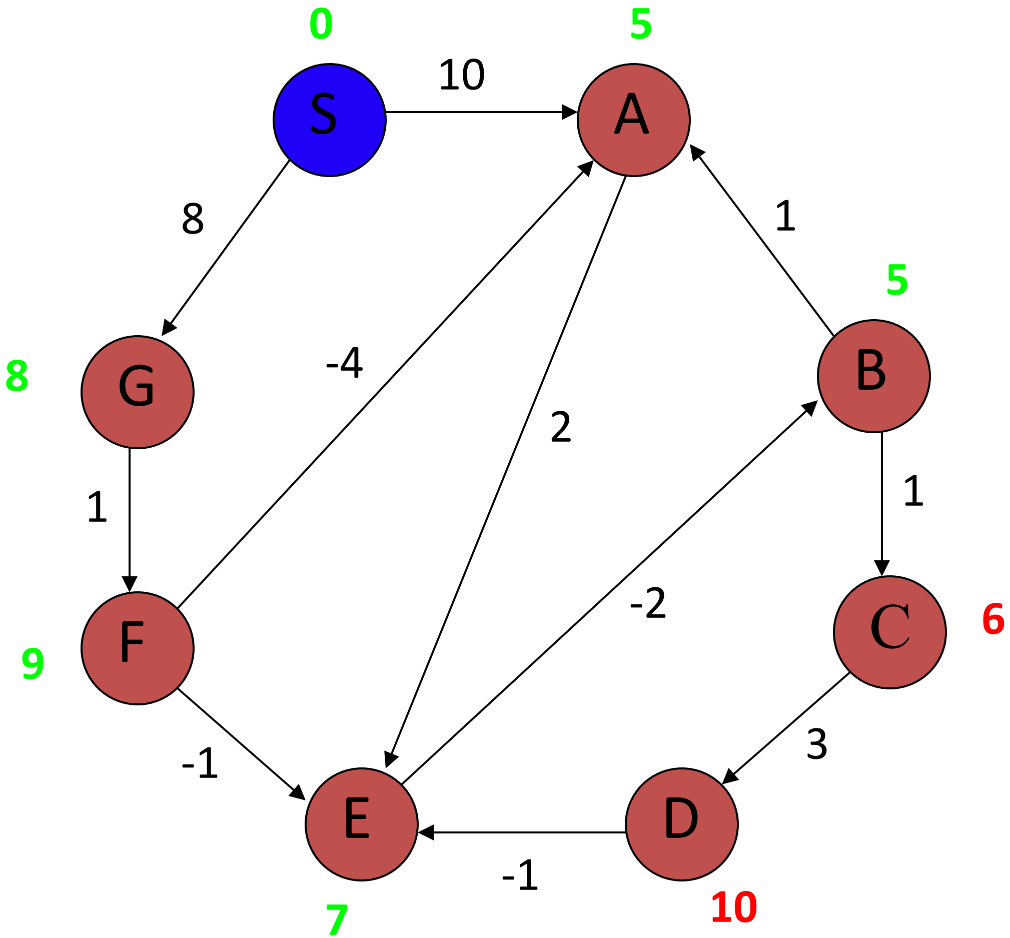
Bellman-Ford Algorithm Example 1



Iteration: 5

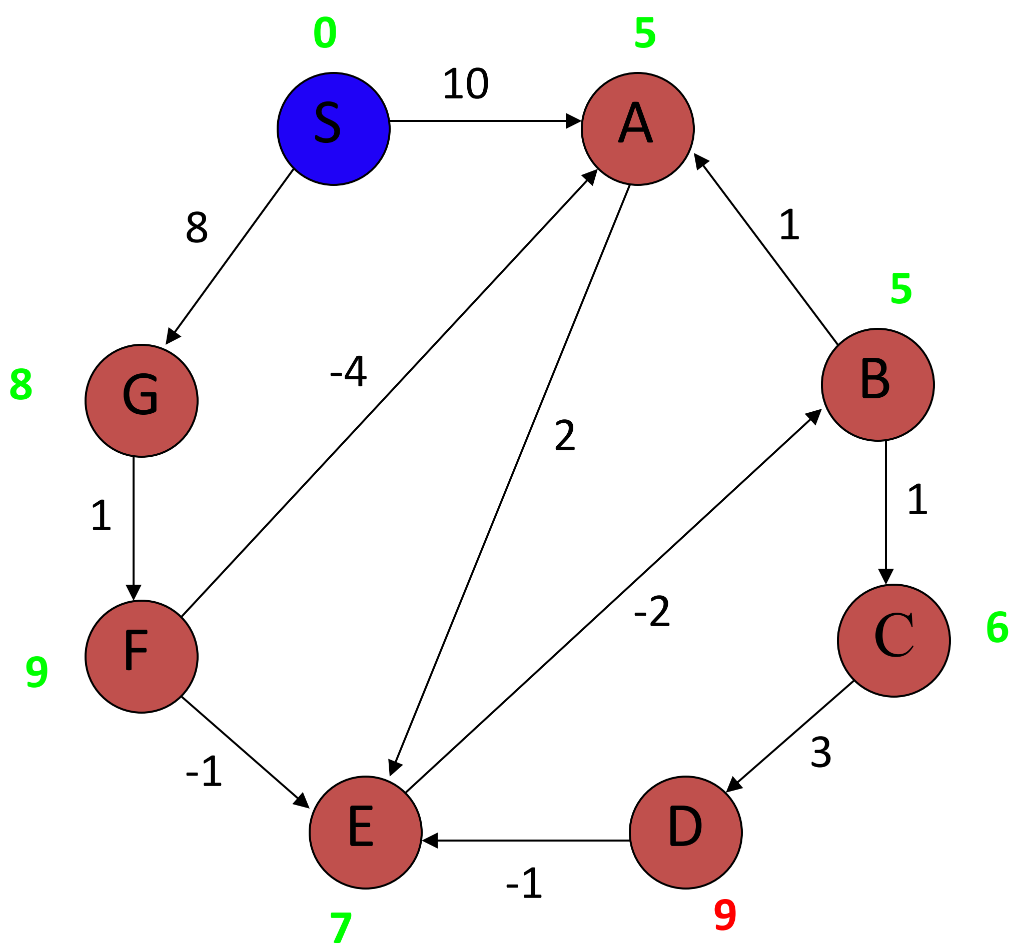
B has the correct distance and path

Bellman-Ford Algorithm Example 1



Iteration: 6

Bellman-Ford Algorithm Example 1

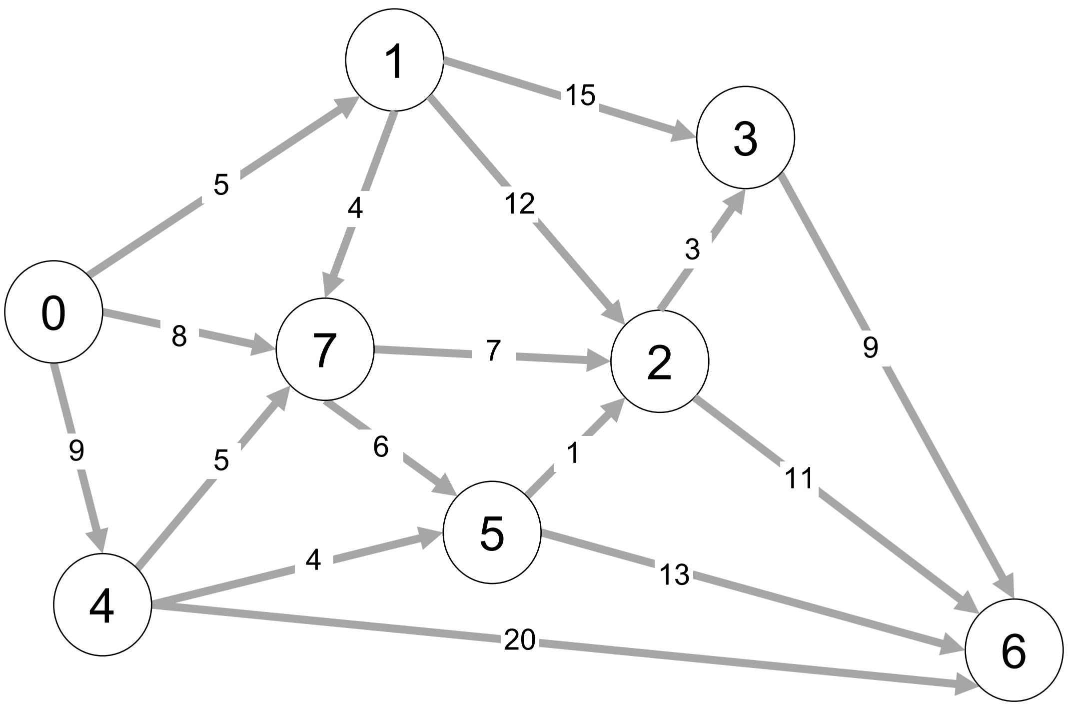


Iteration: 7

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

Bellman-Ford Algorithm Example 2

Repeat V − 1 times: relax all E edges.



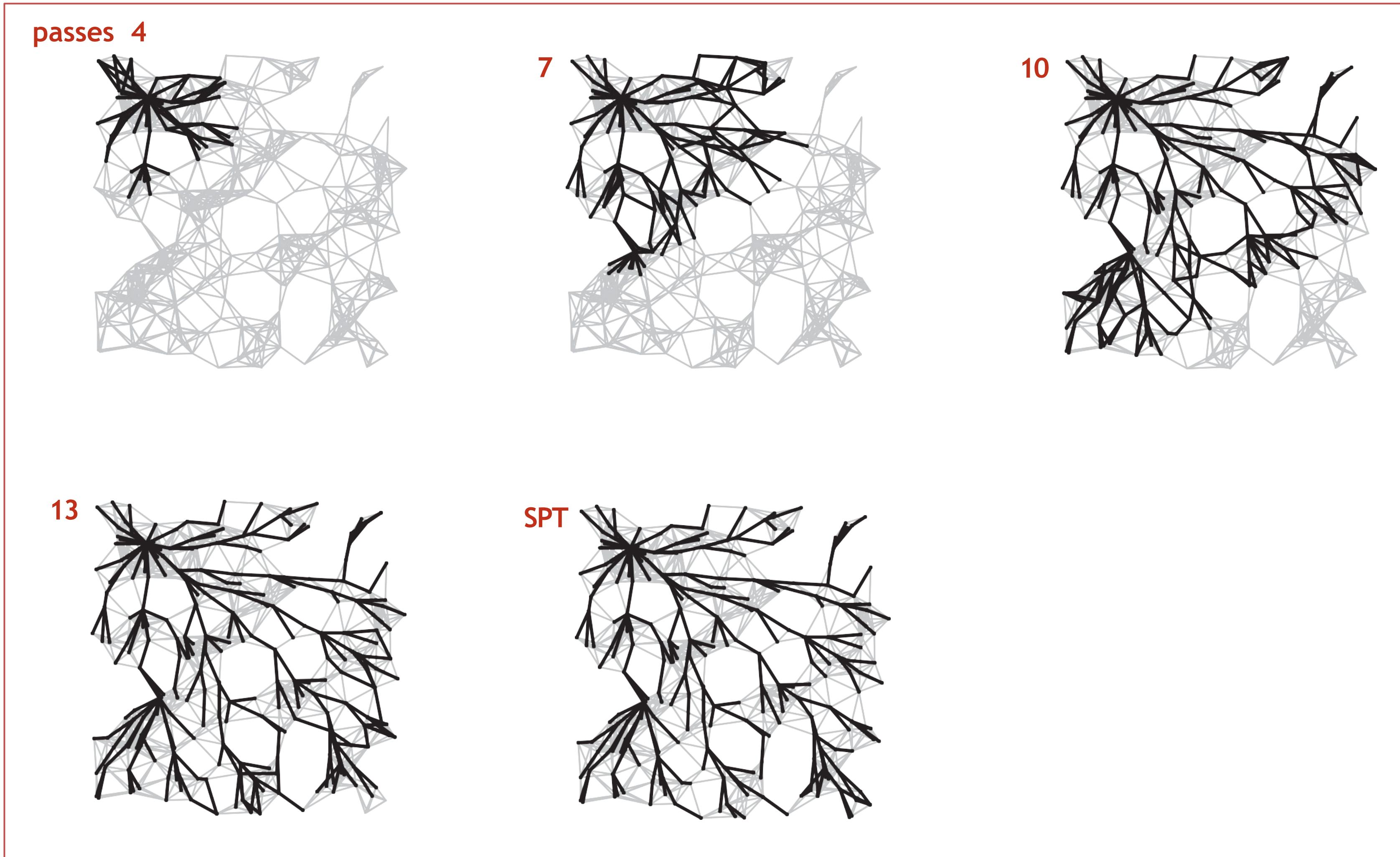
v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	14	
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~28~~	~~26~~	25
7	~~∞~~	8		

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



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

- 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.