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.

Lecture Goals

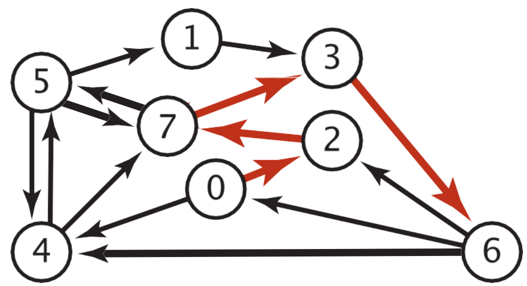
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- For single-source shortest path, we consider:
  - Dijkstra's algorithm
  - Bellman–Ford algorithm
  - Topological Sort for DAG
- For all-pairs shortest 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 source vertex 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

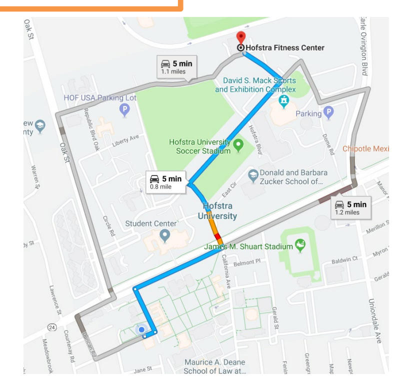
0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Variants

❖ **Which vertices?**
- Single source: from source vertex s to every other vertex.
- Source-sink: from source 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.

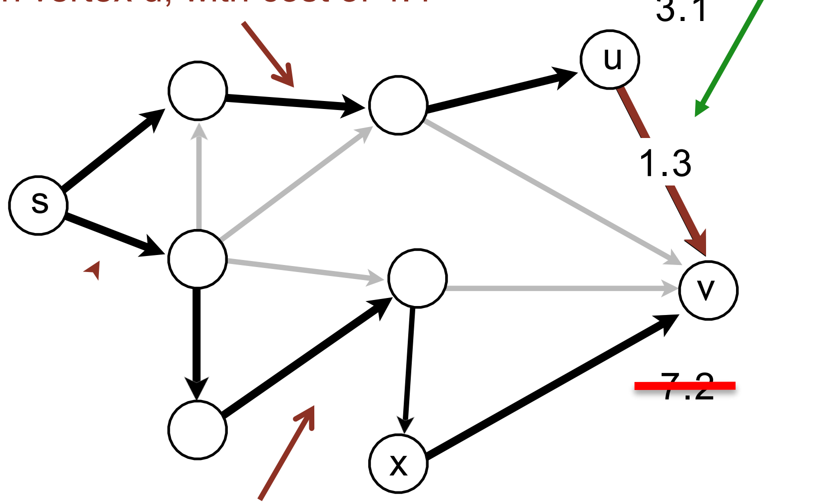
Edge Relaxation

Relax edge e = u→v with weight w(u,v). (We also write uv to denote u→v)

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

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

prevNode[v]=u



Previous shortest path from s to v goes through vertex x, with cost of 7.2

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

OLD distTo[v] = 7.2 > distTo[u] + w(u,v) = 3.1+1.3 = 4.4
NEW distTo[v] ← distTo[u] + w(u,v) = 4.4, prevNode[v] = u

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 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, can detect negative cycles).
- Ex 3. Topological sort. (DAG with no directed 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 u with the shortest known distance from the queue.
  - For each unvisited neighbor vertex v of vertex u, 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 uv.
  - Mark the current vertex as visited once all its neighbors are processed.
- Termination:
  - The algorithm continues until all reachable vertices are visited.
- Time complexity: O(V log V + V) for Binary Heap implementation
- Notes:
  - Dijkstra's Algorithm is greedy and optimal: any vertex that has been visited should have its shortest distance to the source.
  - 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.)

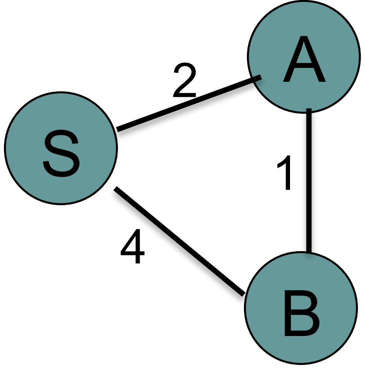
Dijkstra's Algorithm: Correctness Proof

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

Proof.

- Each edge e = u→v is relaxed exactly once (when vertex u is visited), afterwards:
  - distTo[v] ≤ distTo[u] + w(u,v).
- Inequality holds until algorithm terminates because:
  - distTo[v] cannot increase ← distTo[ ] values are monotone decreasing
  - distTo[u] 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
SD: Shortest Distance. PN: Previous Node



N1	SD	PN
S	0	
A	∞	
B	∞	

Visit S

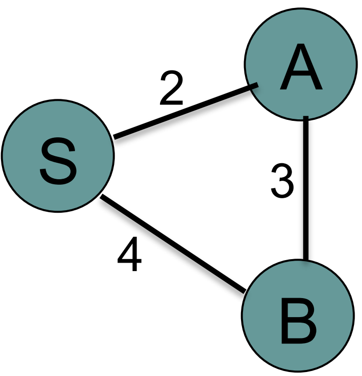
N1	SD	PN
S	0	
A	2	S
B	4	S

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

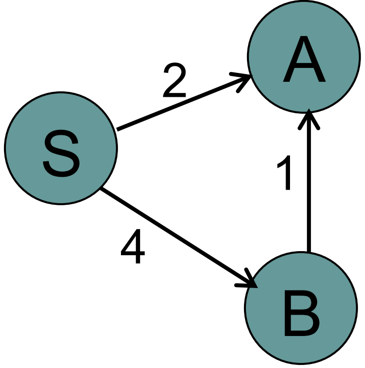
Visit A

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 →

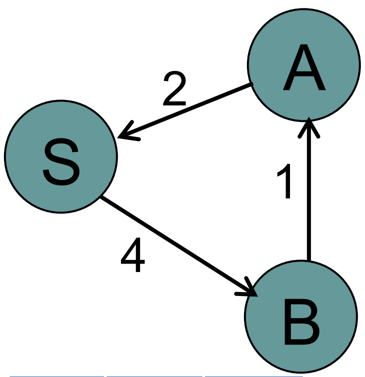
N1	SD	PN
S	0	
A	2	S
B	4	S

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



N1	SD	PN
S	0	
A	∞	
B	∞	

Visit S →

N1	SD	PN
S	0	
A	∞	
B	4	S

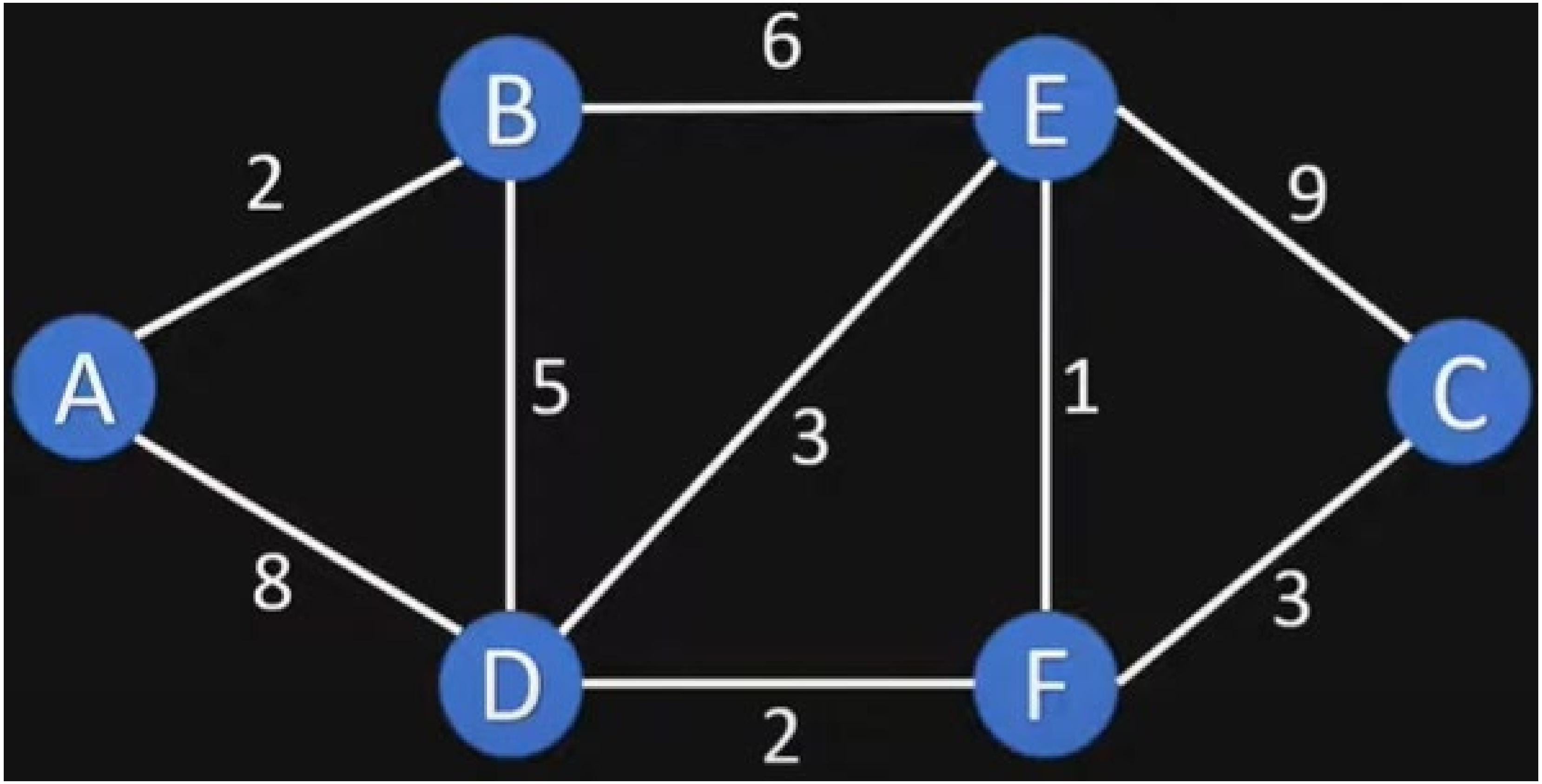
Visit B →

N1	SD	PN
S	0	
A	5	B
B	4	S

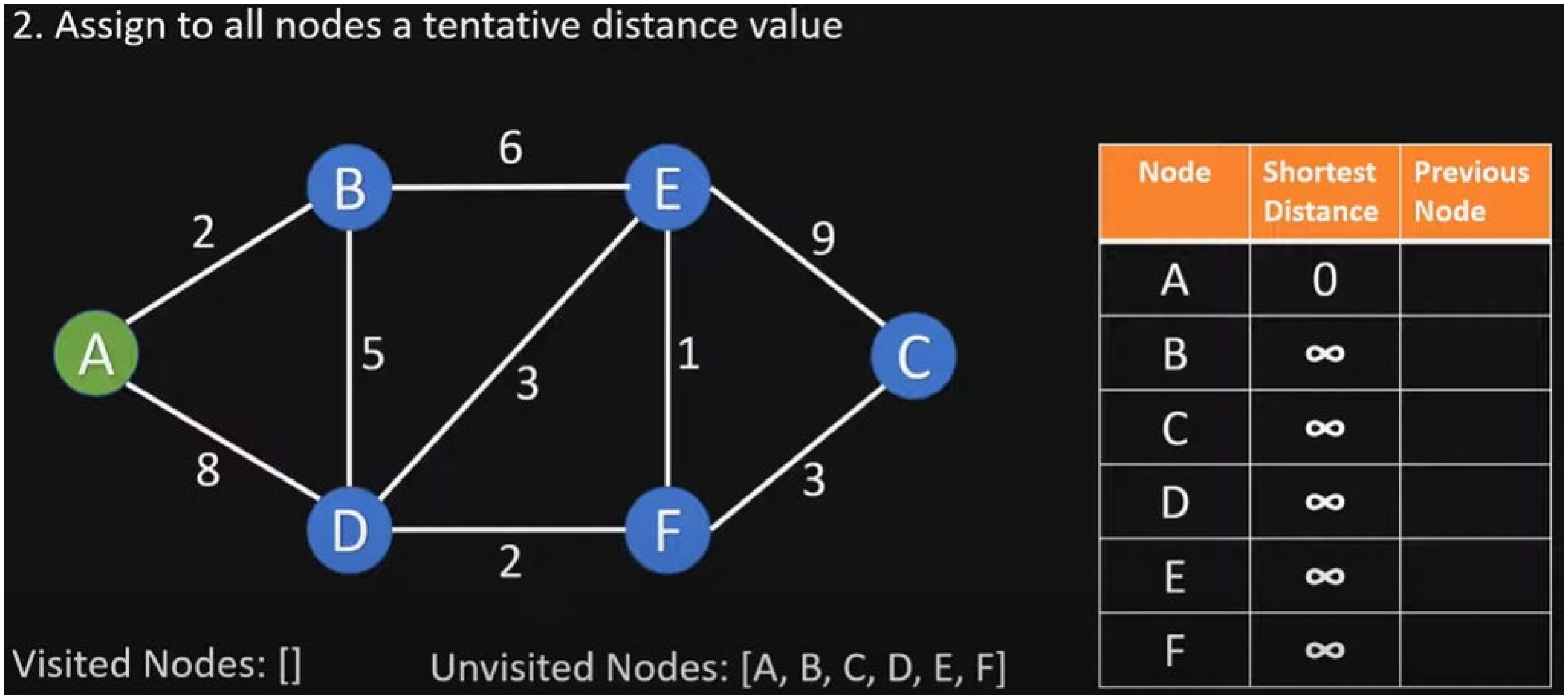
Visit A →

N1	SD	PN
S	0	
A	5	B
B	4	S

Example Graph

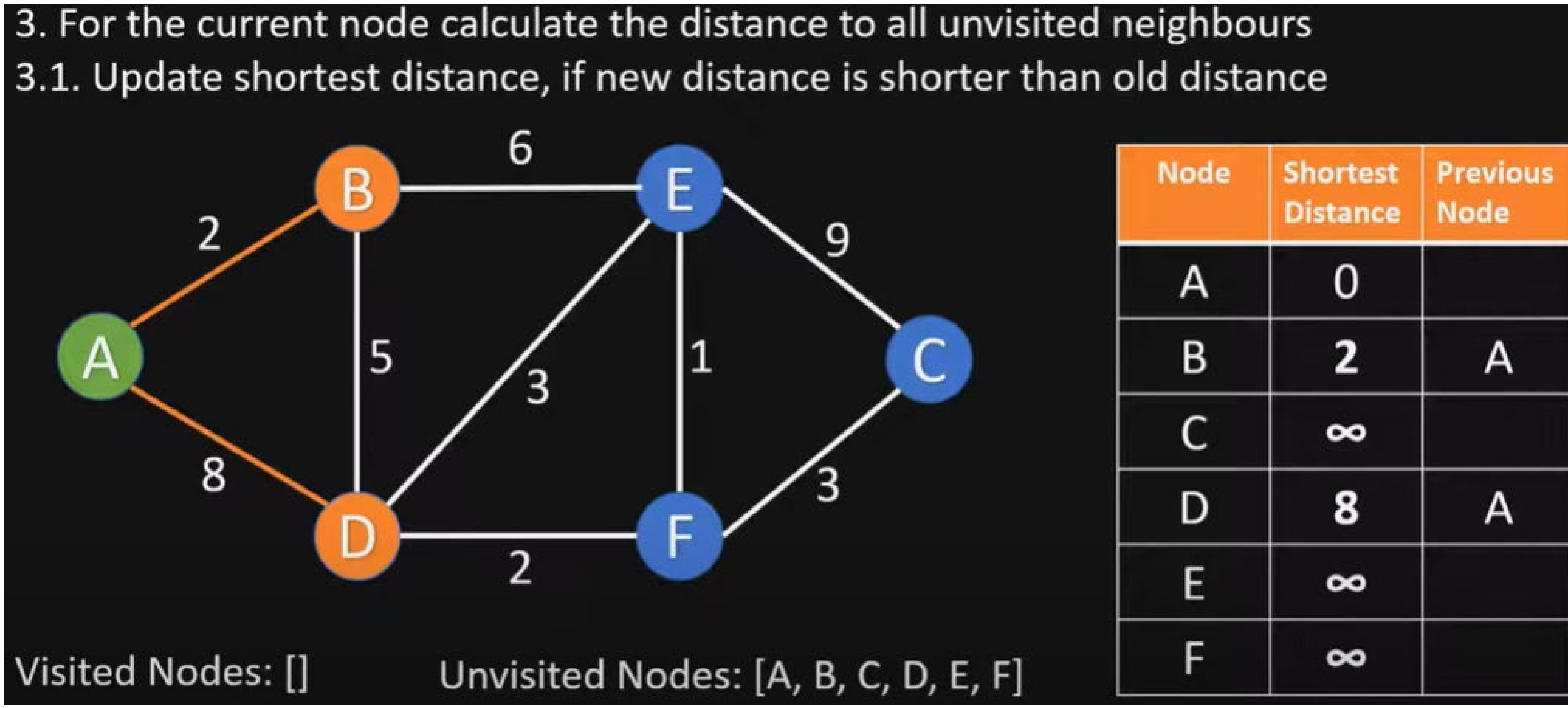


Initialize



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

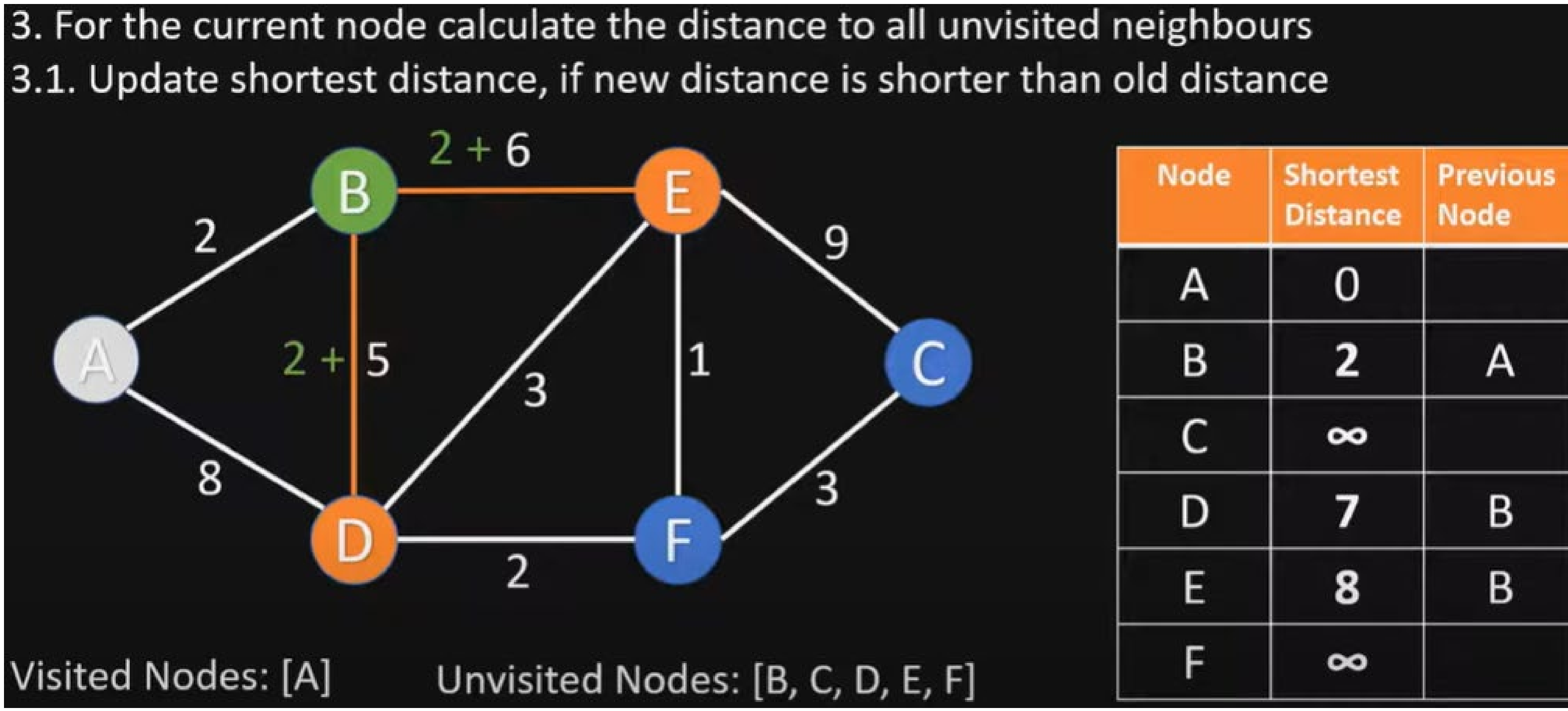
Visit vertex A



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

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

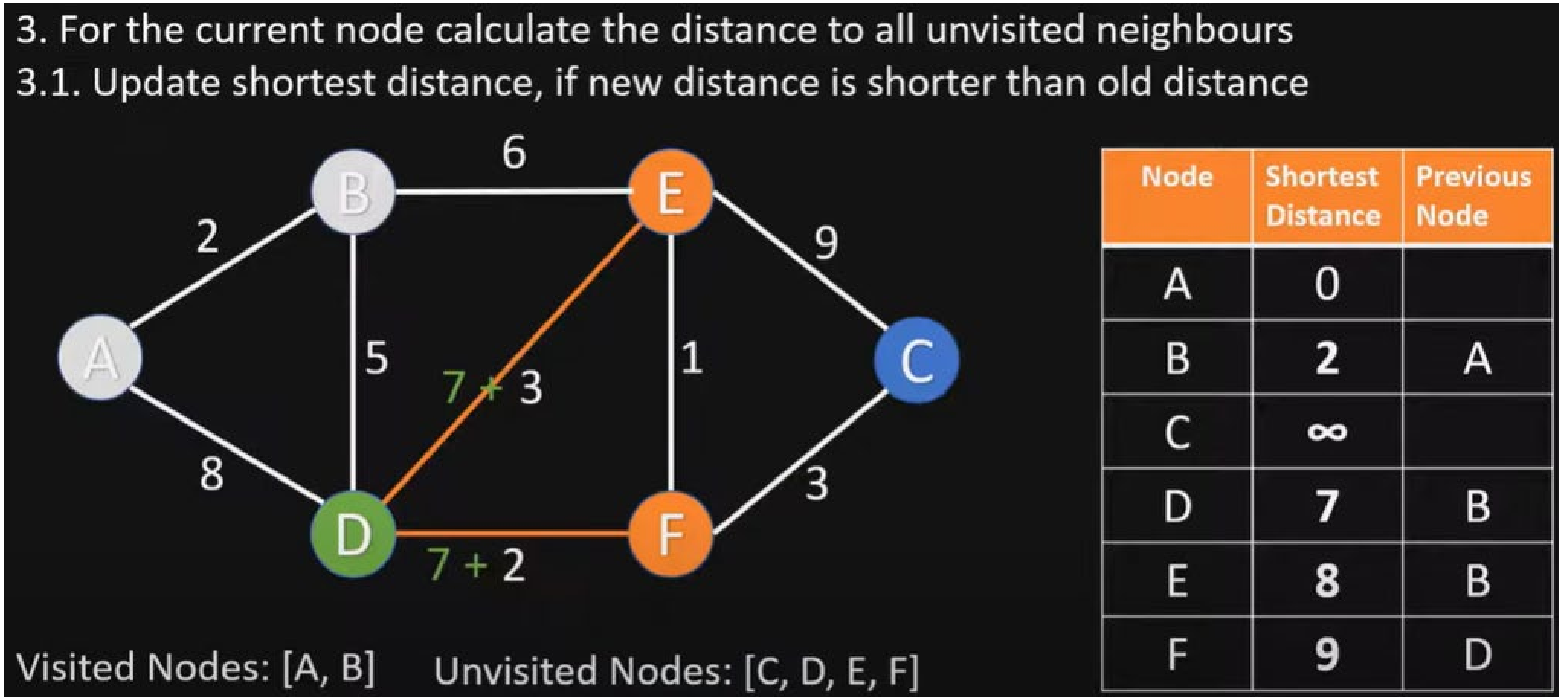
Visit vertex B



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

OLD distTo[D] = 8 > distTo[B] + w(B,D) = 2+5 = 7
NEW distTo[D] $\leftarrow$ distTo[B] + w(B,D) = 7, prevNode[D] = B
OLD distTo[E] = ∞ > distTo[B] + w(B,E) = 2+6 = 8
NEW distTo[E] $\leftarrow$ distTo[B] + w(B,E) = 8, prevNode[E] = B

Visit vertex D

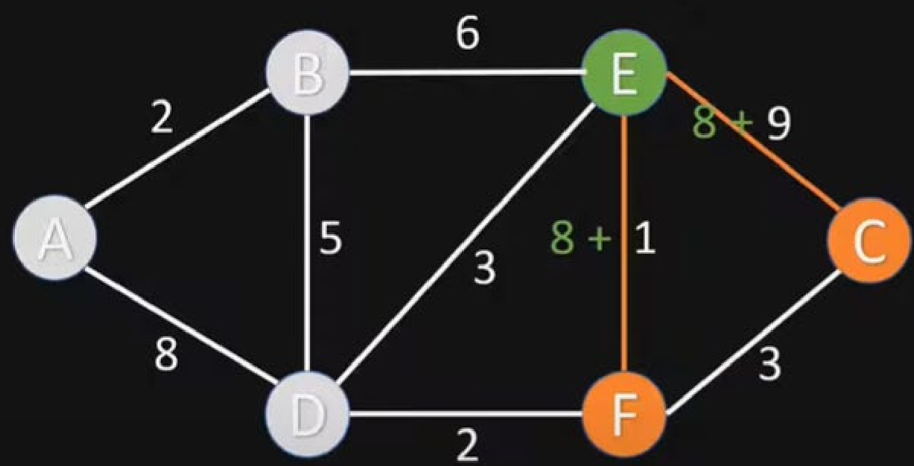


Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	∞	
D	7	B
E	8	B
F	9	D

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

Visit vertex 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

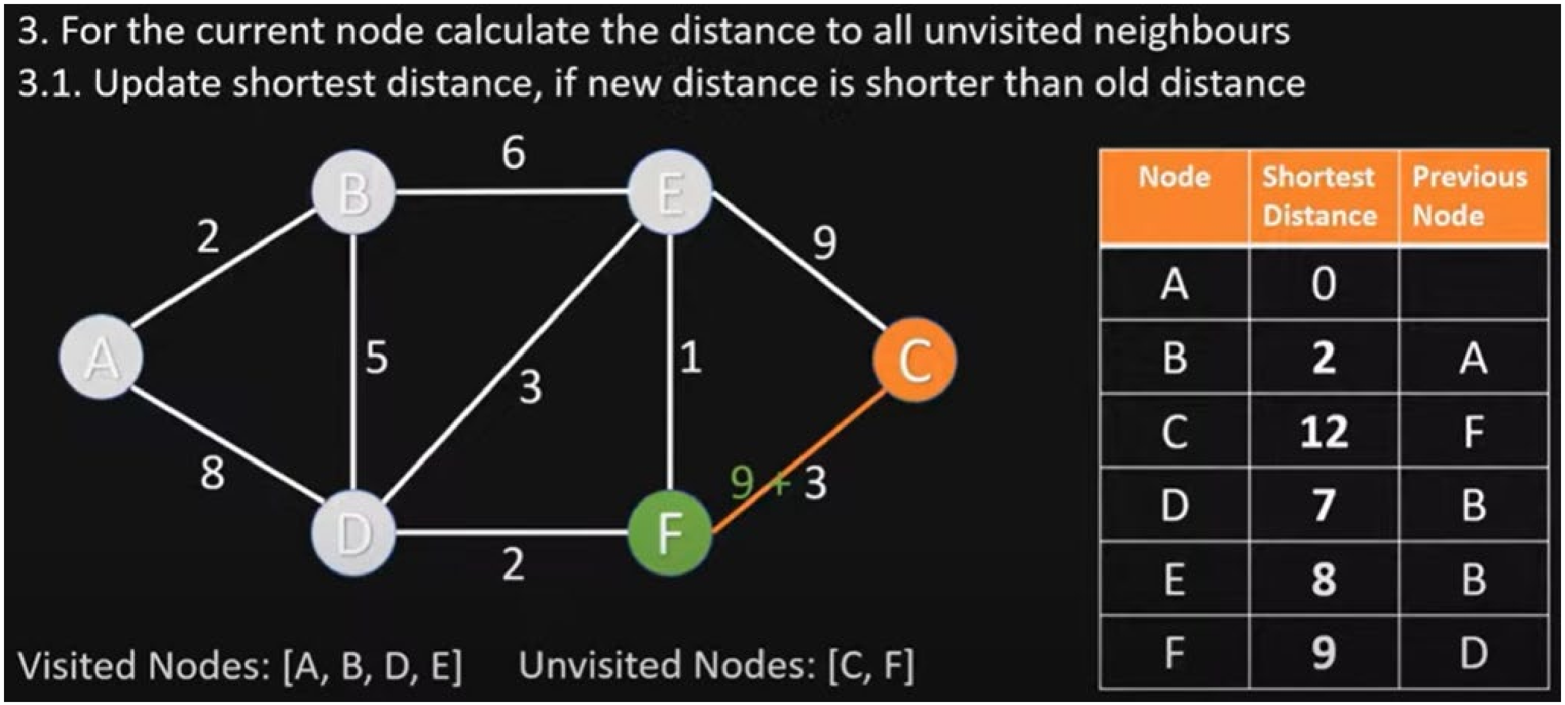


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

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

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

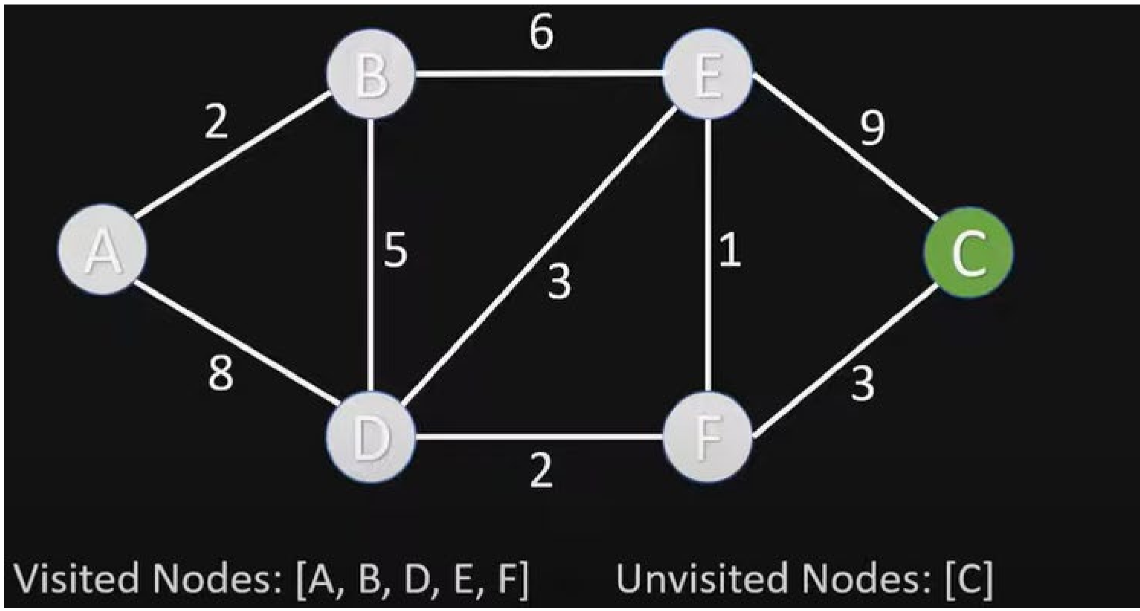
Visit vertex F



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

OLD distTo[C] = 17 > distTo[F] + w(F,C) = 9+3 = 12
NEW distTo[C] ← distTo[F] + w(F,C) = 12, prevNode[C] = F

Visit vertex C

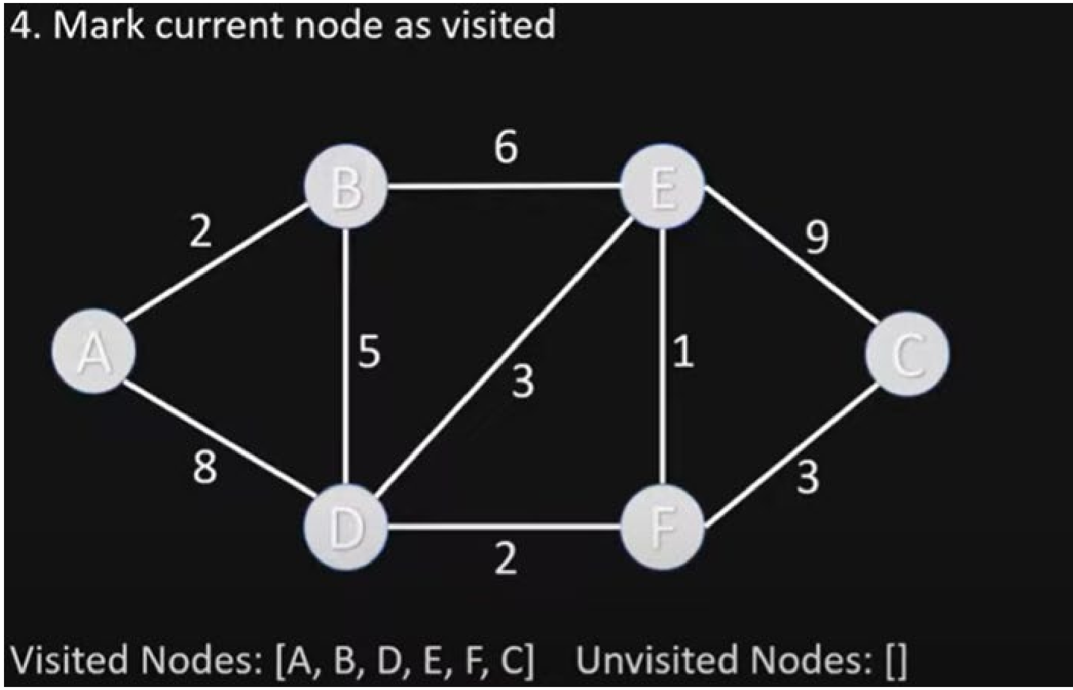


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

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

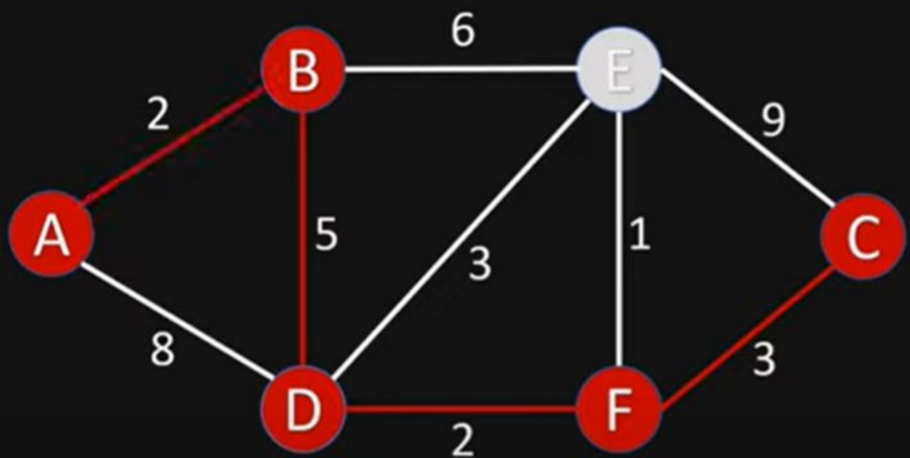


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Getting the Shortest Path from A to C

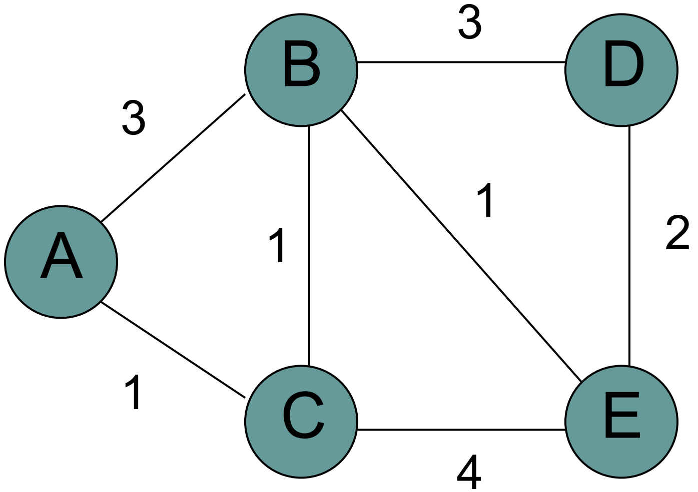
- C's previous vertex is F; F's previous vertex is D; D's previous vertex is B; B's previous vertex is A
- Shortest Path from A to C is ABDFC

Get shortest path from A to C

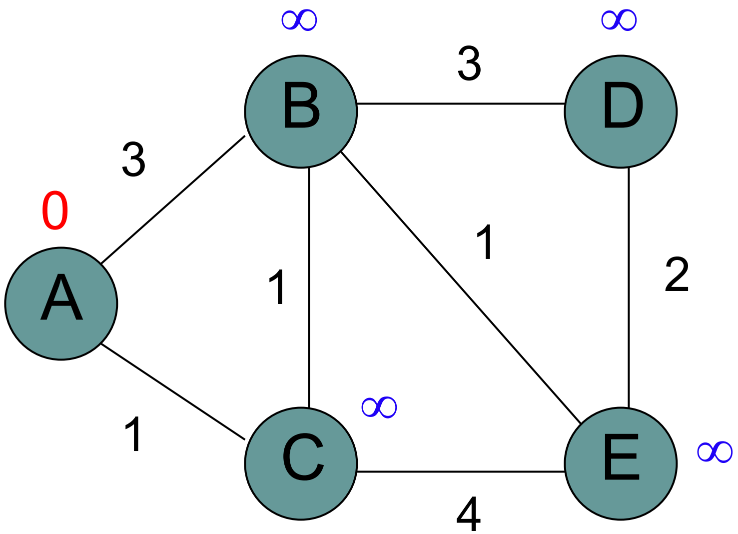


Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

Dijkstra’s Algorithm Example 2

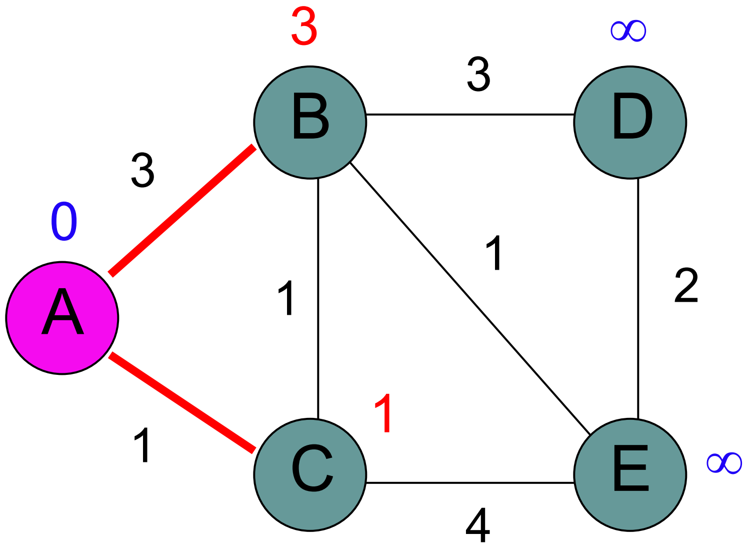


Initialize



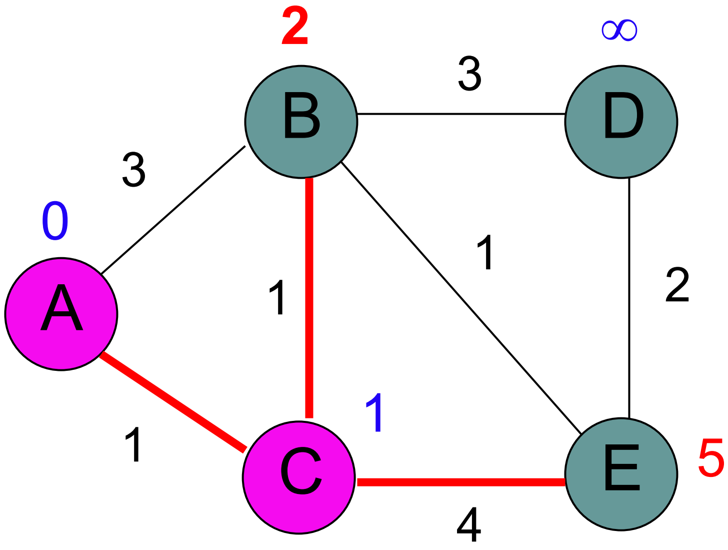
N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visit vertex A



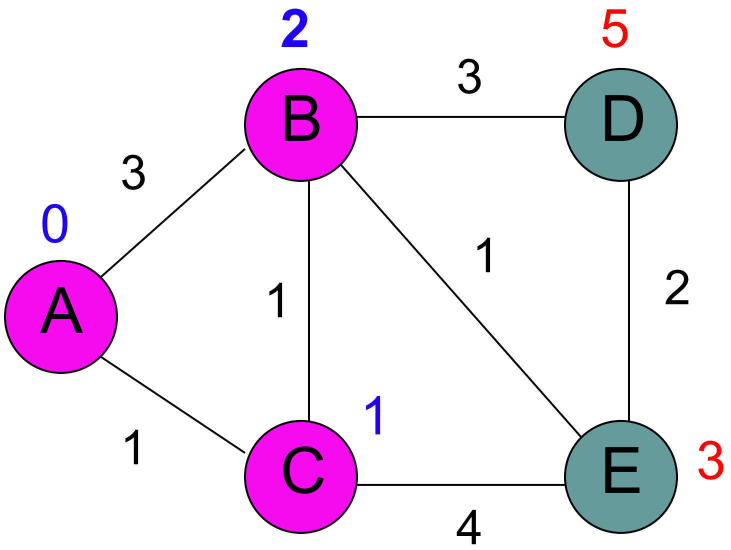
N	SD	PN
A	0	
B	3	A
C	1	A
D	∞	
E	∞	

Visit vertex C



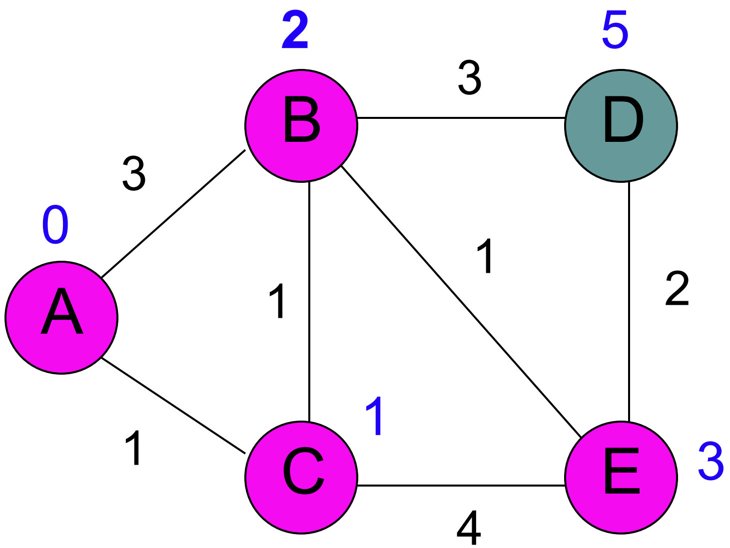
N	SD	PN
A	0	
B	2	C
C	1	A
D	∞	
E	5	C

Visit vertex B



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

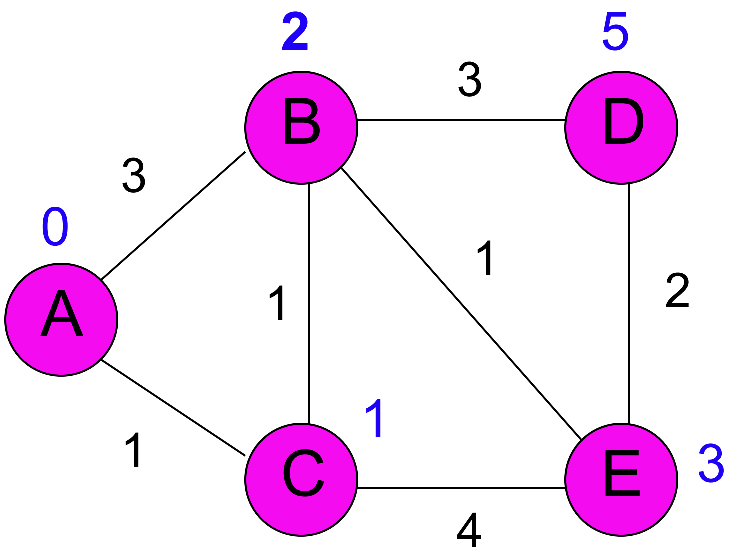
Visit vertex E



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

Nothing changes

Visit vertex D



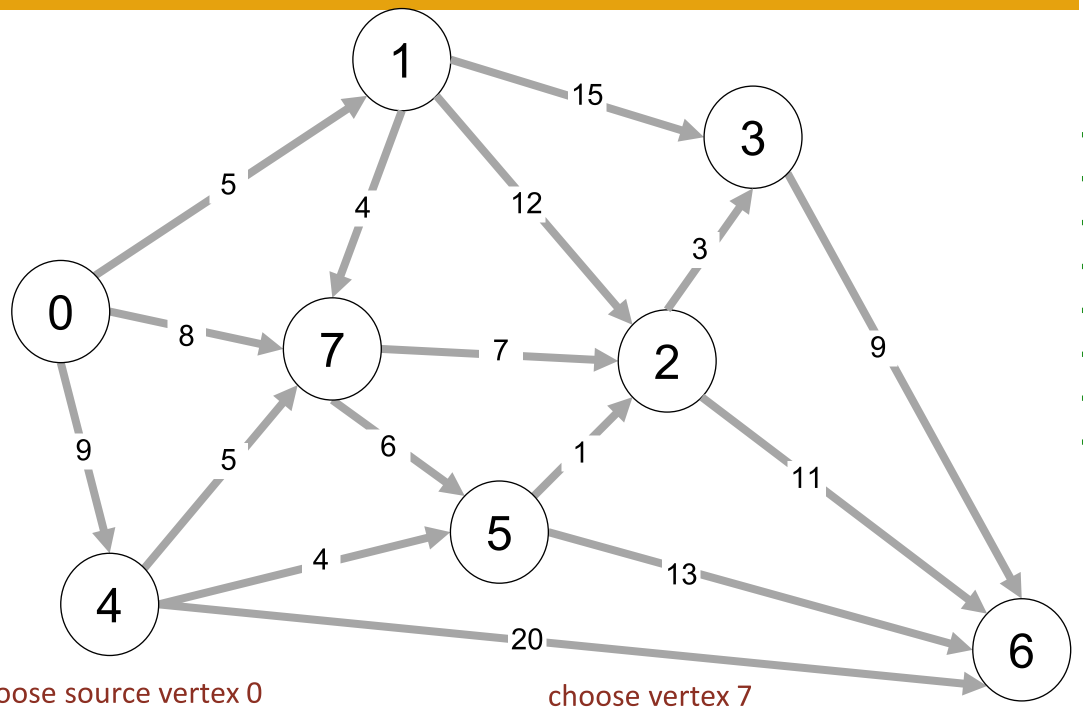
N	SD	PN
A	0	
B	2	C
C	1	A
D	5	B
E	3	B

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

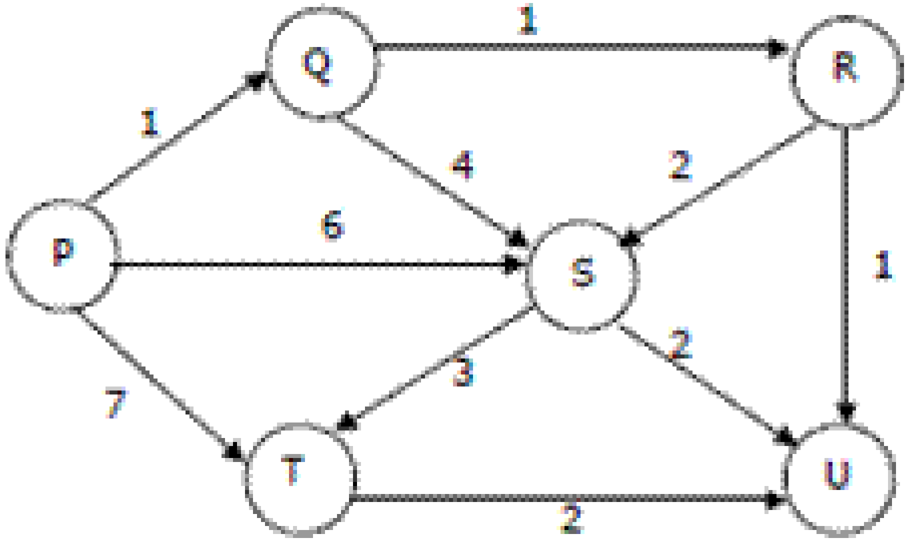
choose vertex 7
relax all edges adjacent from 7
choose vertex 4
relax all edges adjacent from 4

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

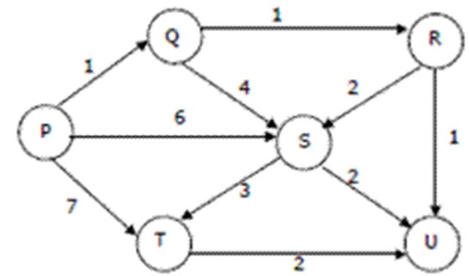
v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	~~7~~	4	
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

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	R
T	7	P
U	3	R

Visit U (nothing changes)

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

Visit S (nothing changes)

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

Visit T (nothing changes)

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

Finished

N	SD	PN
P	0	
Q	1	P
R	2	Q
S	4	R
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.
  - Can terminate early when all distTo[] values have converged
  - The order of edge relaxations affects algorithm efficiency but not correctness.

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.

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

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 with up to **V-1** edges, which is the maximum possible number of edges 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

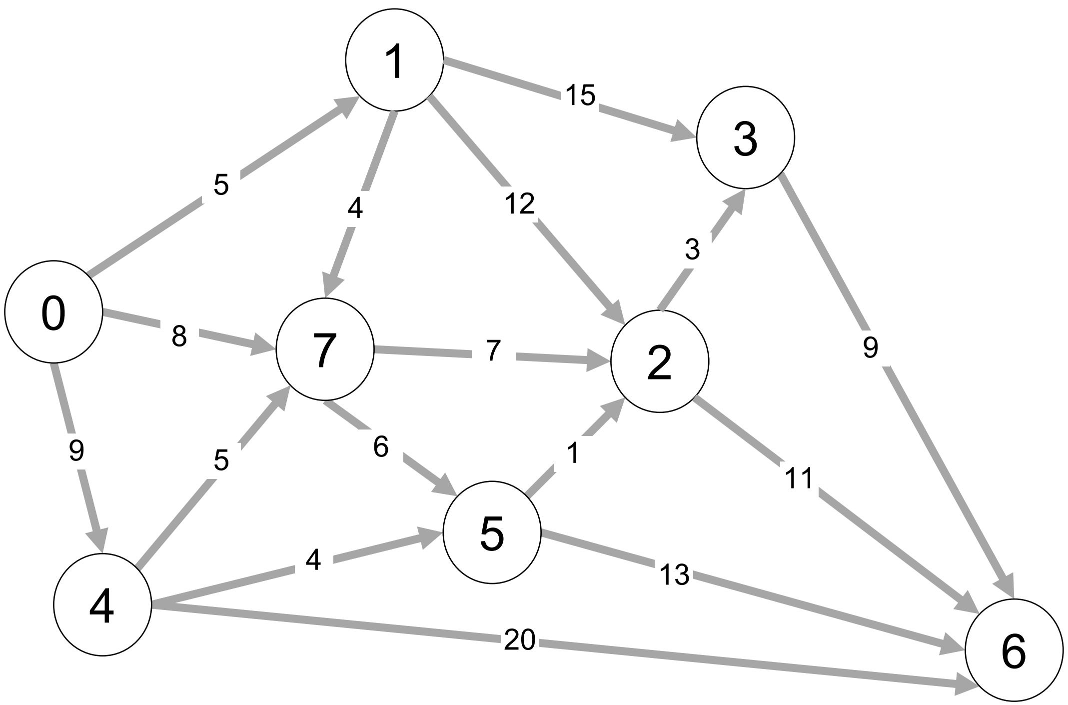
- Initialize distance array distTo[] for each vertex v as distTo[v] = ∞, and distTo[s] = 0 to source vertex s.
- Relax all edges V-1 times.
  - Can terminate early when all distTo[] values have converged
  - The order of edge relaxations affects algorithm efficiency but not correctness. A good heuristic is to follow the Breadth First Search (BFS) order.
- Relax all the edges one more time i.e. the V-th time:
  - Case 1 (Negative cycle exists): if any edge can be further relaxed, i.e., for any edge u→v, if distTo[u] > distTo[u] + w(u,v)
  - 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 −∞, 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

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		

Reverse order of edge relaxations will result in slower convergence

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 (converged, no further changes, so stop here)

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

Order of edge relaxations

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.

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.