Lecture 11
Shortest Paths

Department of Computer Science
Hofstra University

Lecture Goals

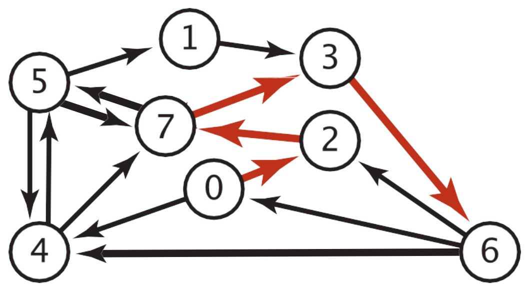
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- We introduce and analyze Dijkstra's algorithm for shortest-paths problems with nonnegative weights.
- We conclude with the Bellman–Ford algorithm 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



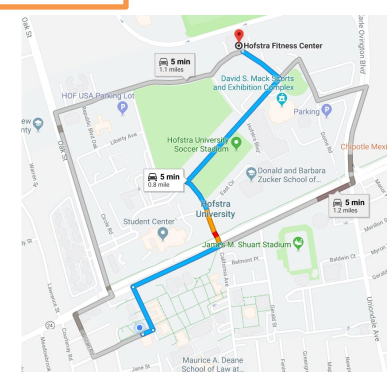
shortest path from 0 to 6

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.

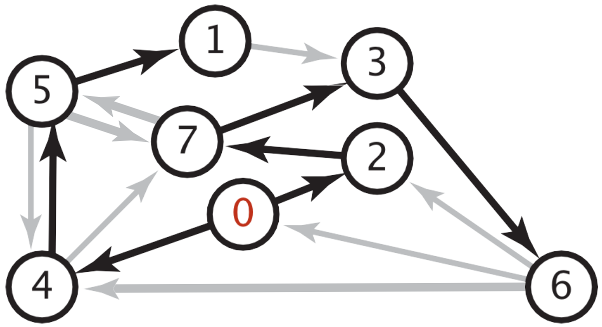


Simplifying assumption: Each vertex is reachable from s.

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.



shortest-paths tree from 0

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.

	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

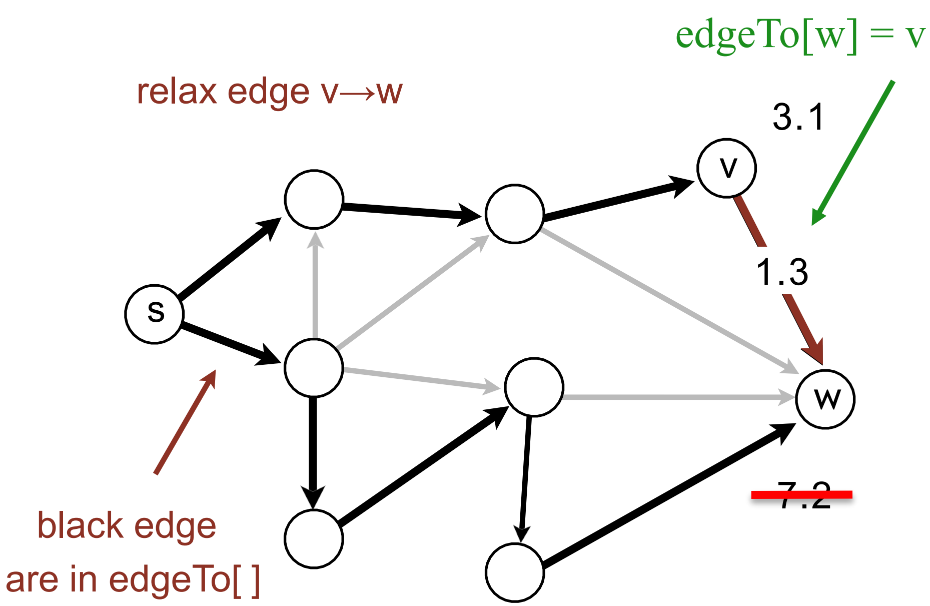
```
public double distTo(int v)
{ return distTo[v]; }
```

```
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;
}
```

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].




```
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
= 3.1+1.3 = 4.4

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.

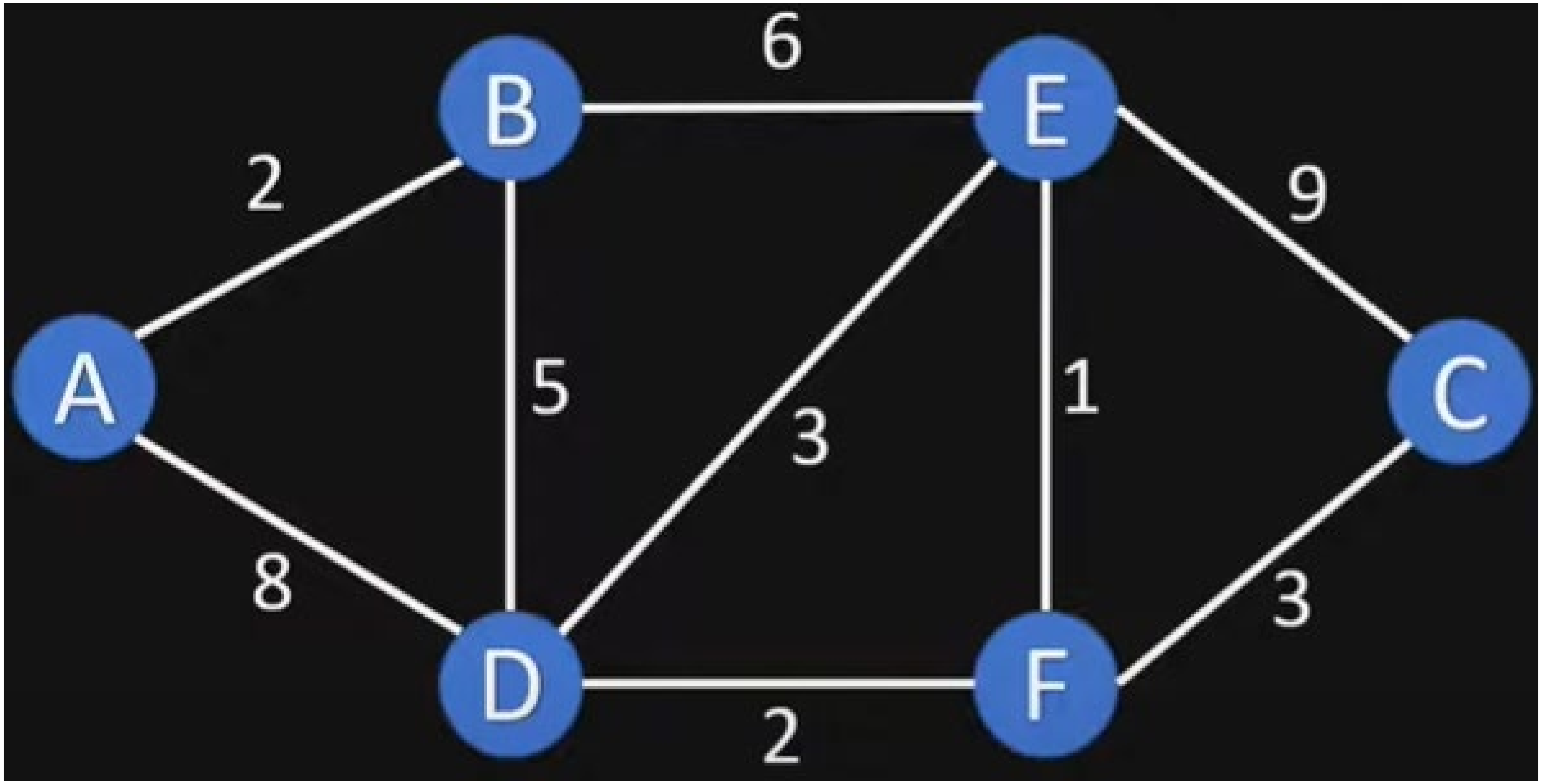
Efficient implementations. How to choose which edge to relax?

- Ex 1. Dijkstra's algorithm. (nonnegative weights, directed cycles).
- Ex 2. Topological sort algorithm. (no directed cycles).
- Ex 3. Bellman–Ford algorithm. (no negative cycles).

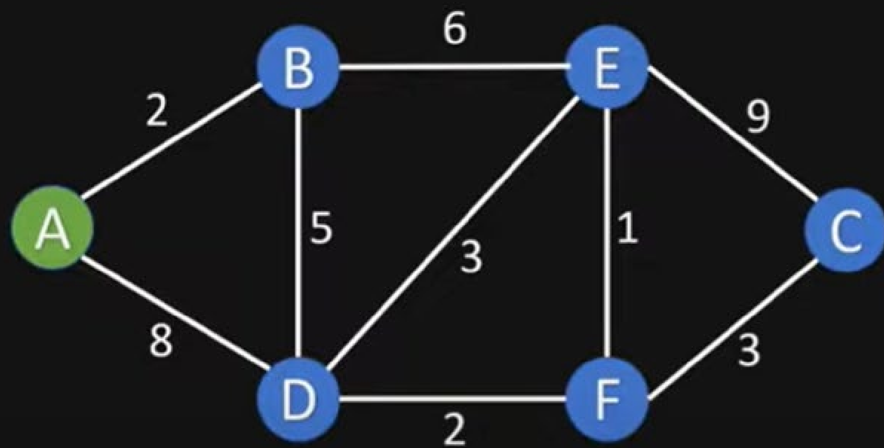
Dijkstra's Algorithm

- Initialization:
  - Set the distance to the source node as 0 and to all other nodes as infinity.
  - Mark all nodes as unvisited and store them in a priority queue.
- Main Loop:
  - 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.
  - 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.

Example Graph

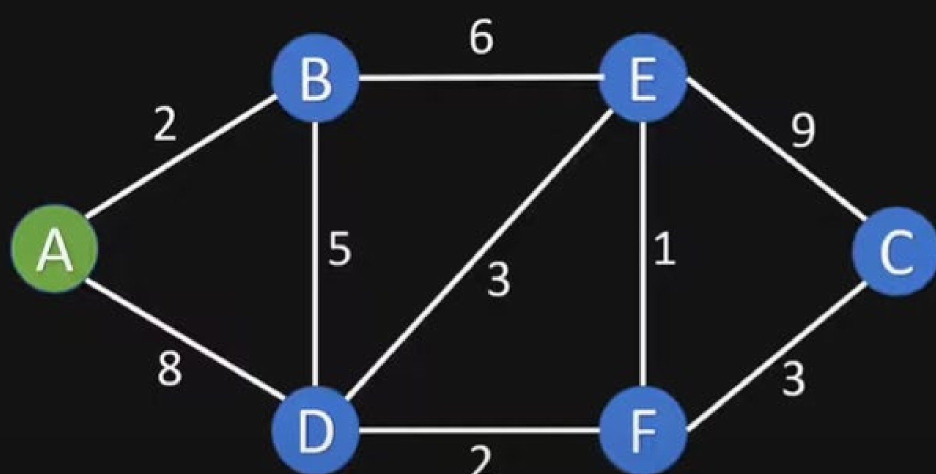


1. Mark all nodes as unvisited



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

2. Assign to all nodes a tentative distance value

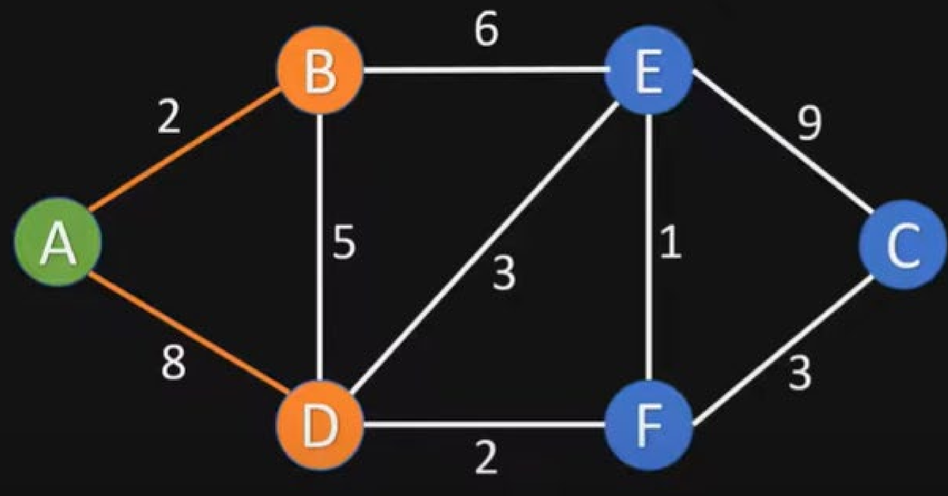


Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

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

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

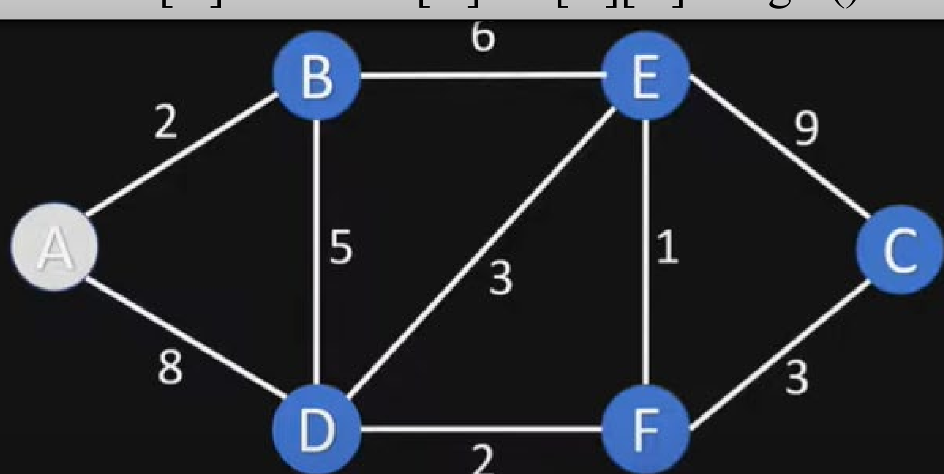


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

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

OLD distTo[D] = ∞ > distTo[A] + e[A][D].weight() = 0+8 = 8

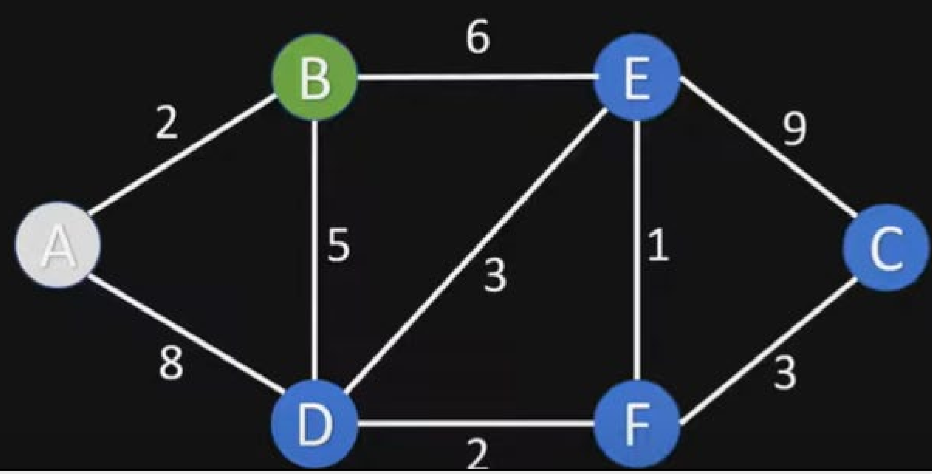
NEW distTo[D] ← distTo[A] + e[A][D].weight() = 8



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

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

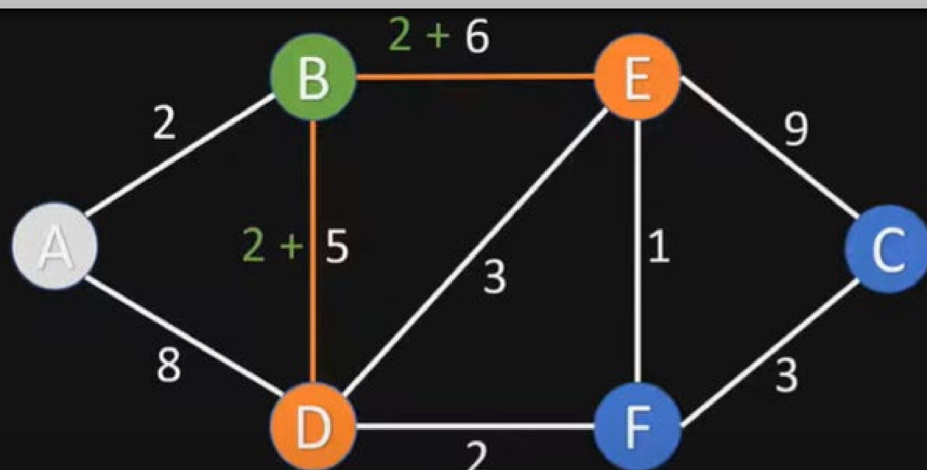
5. Choose new current node from unvisited nodes with minimal distance



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

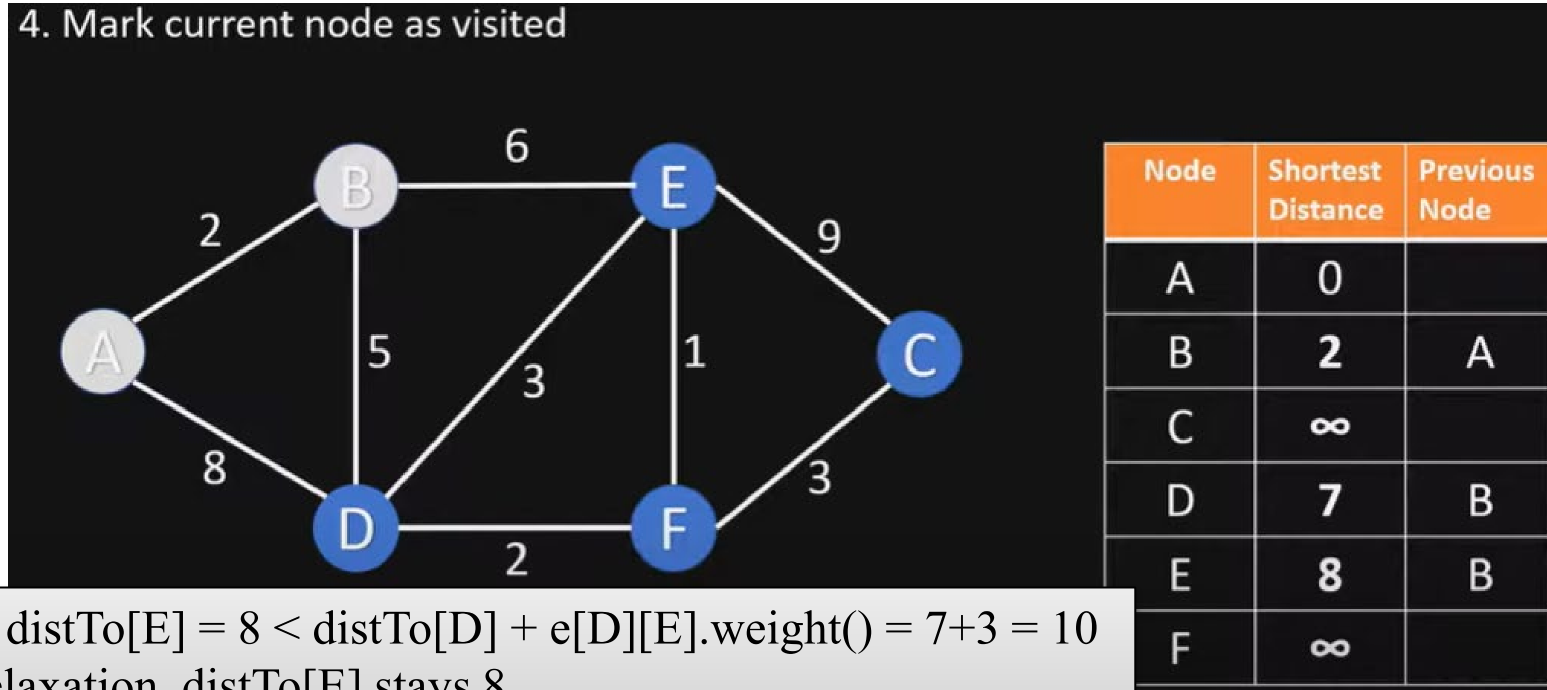
OLD distTo[D] = 8 > distTo[B] + e[B][D].weight() = 2+5 = 7
NEW distTo[D] ← distTo[B] + e[B][D].weight() = 7
OLD distTo[E] = ∞ > distTo[B] + e[B][E].weight() = 2+6 = 8
NEW distTo[E] ← distTo[B] + e[B][E].weight() = 8

ghbours
distance

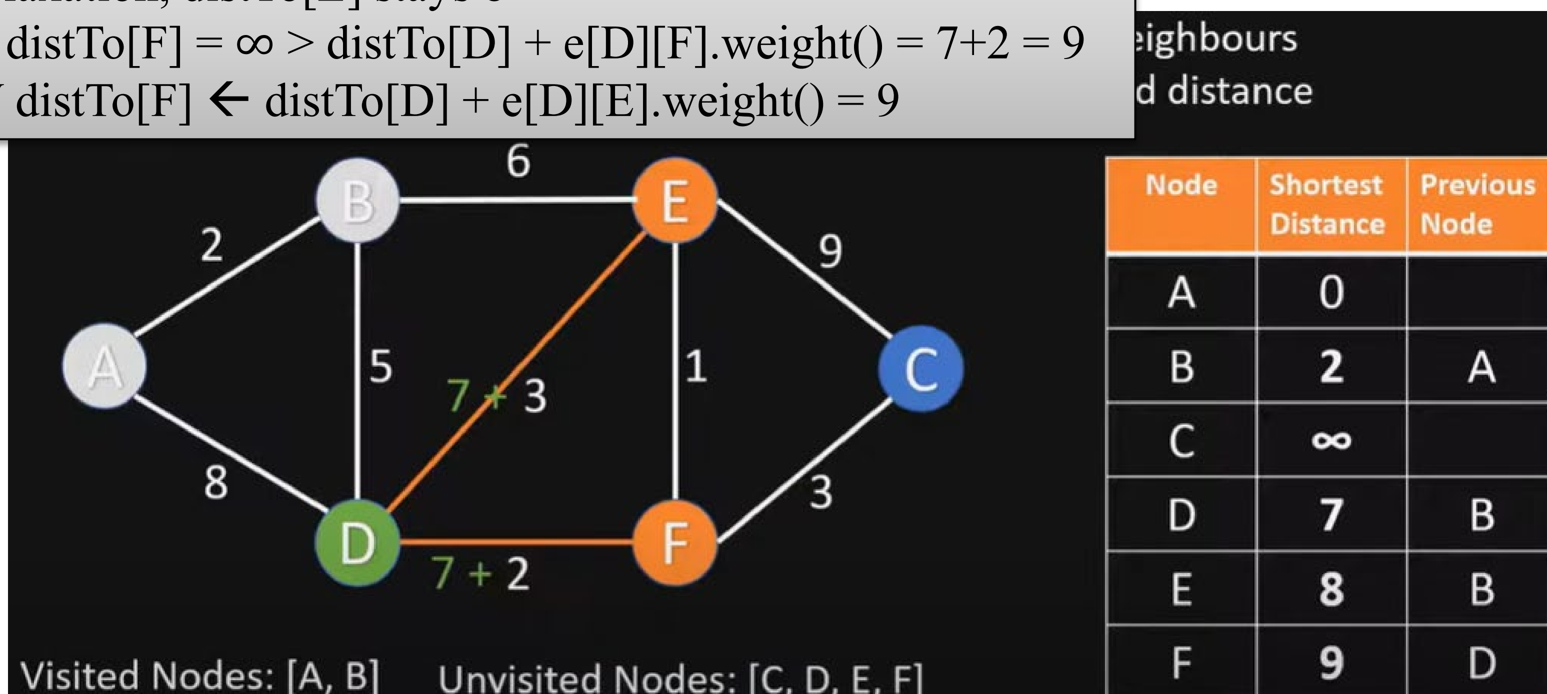


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

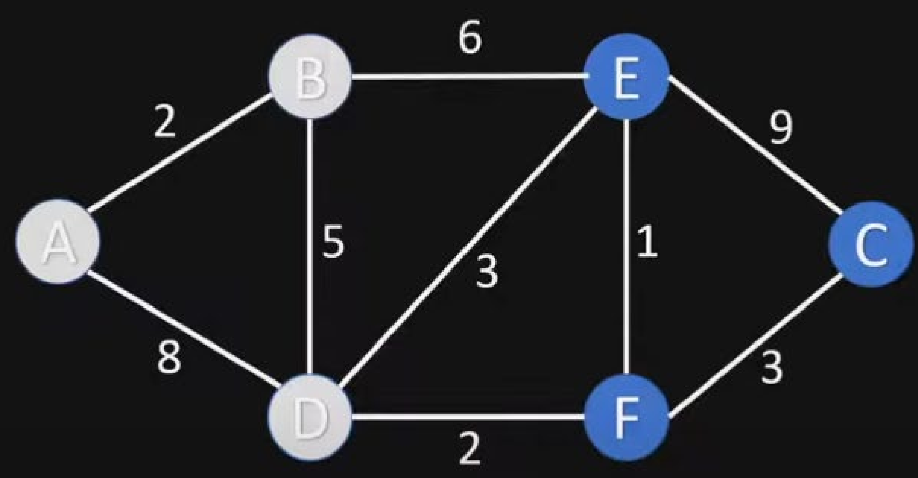
Visited Nodes: [A]   Unvisited Nodes: [B, C, D, E, F]



OLD distTo[E] = 8 < distTo[D] + e[D][E].weight() = 7+3 = 10
No relaxation, 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



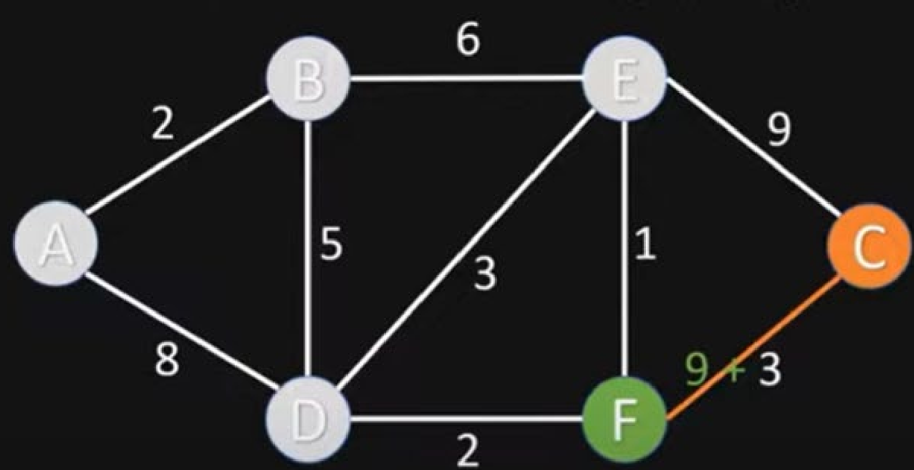
4. Mark current node as visited



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

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

neighbours
old distance

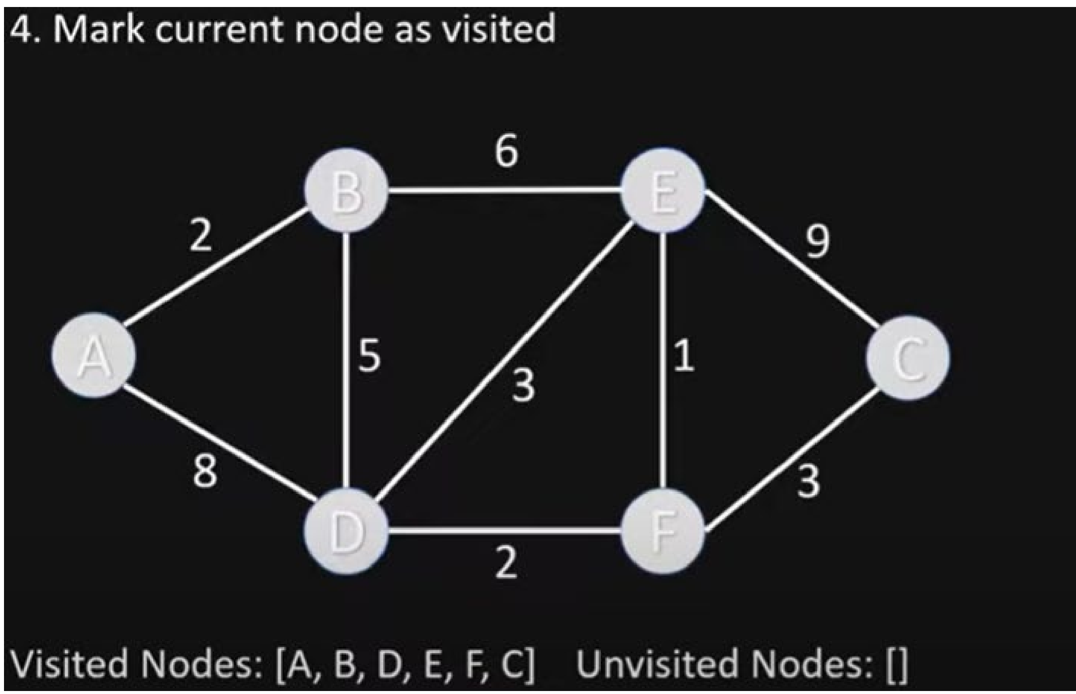


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

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

End of Algorithm

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

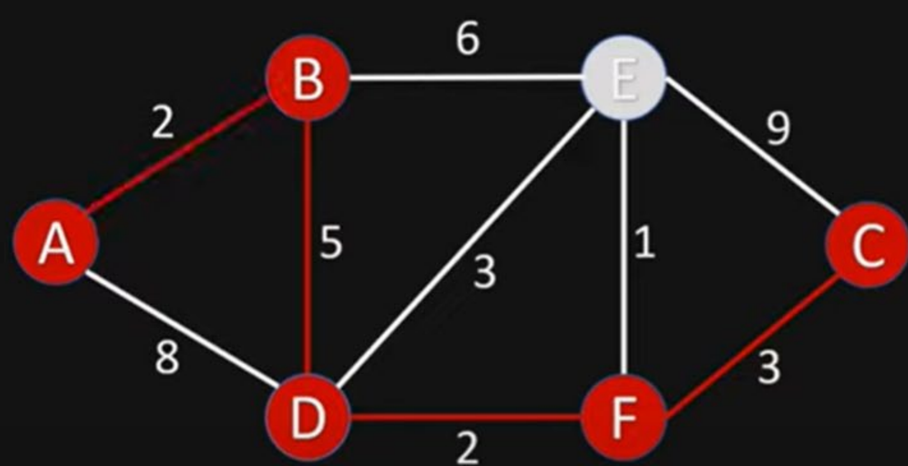


Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

Getting the Shortest Path from A to C

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

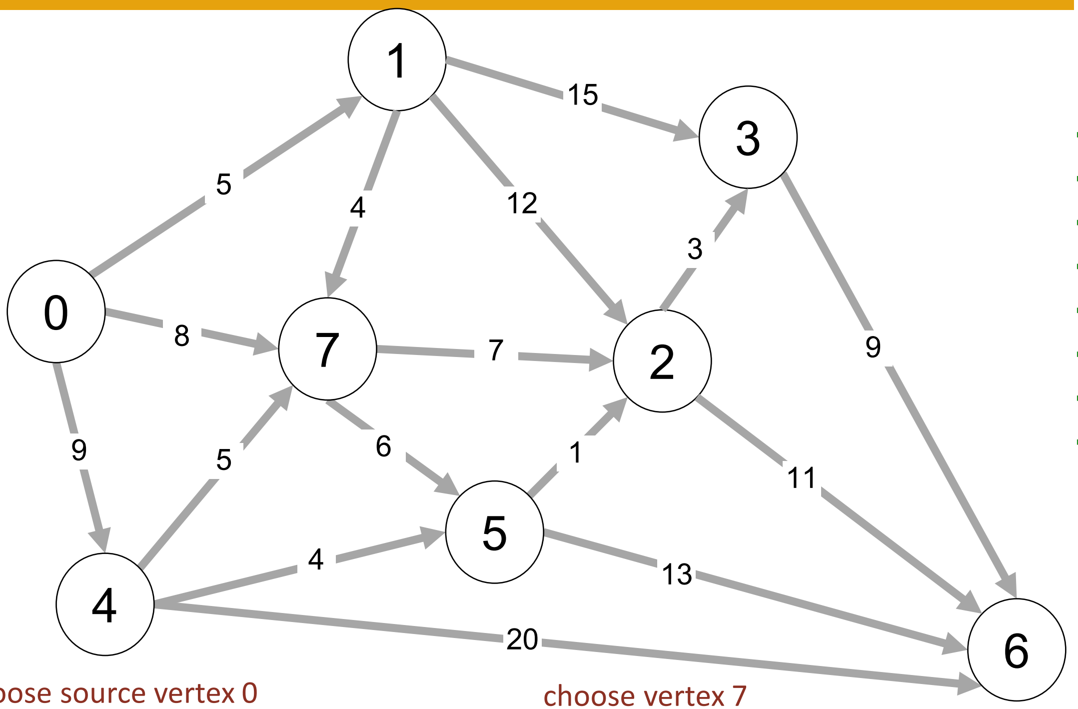
Get shortest path from A to C



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 2nd Example

- 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

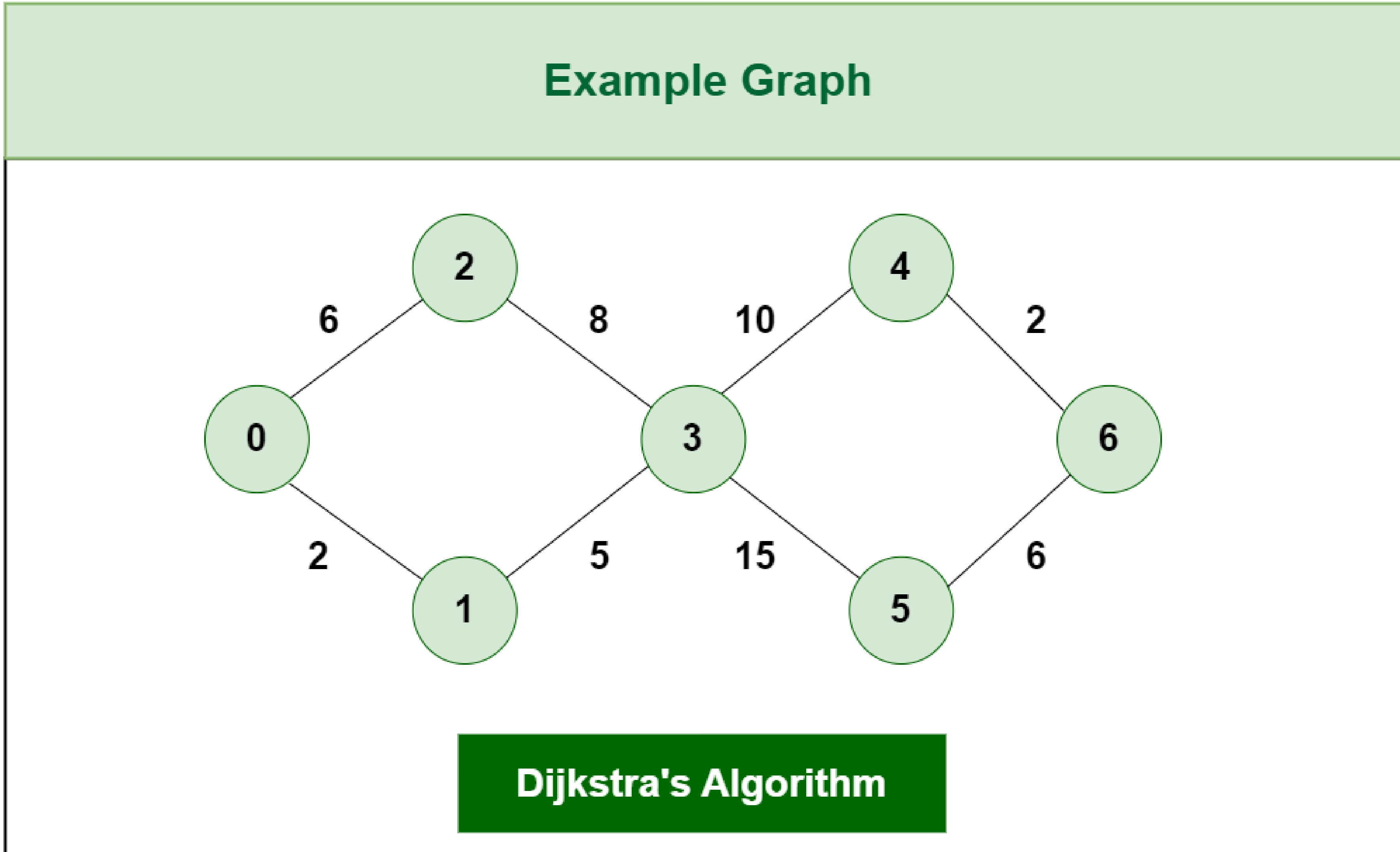
choose vertex 7
relax all edges adjacent from 7
choose vertex 4
relax all edges adjacent from 4

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

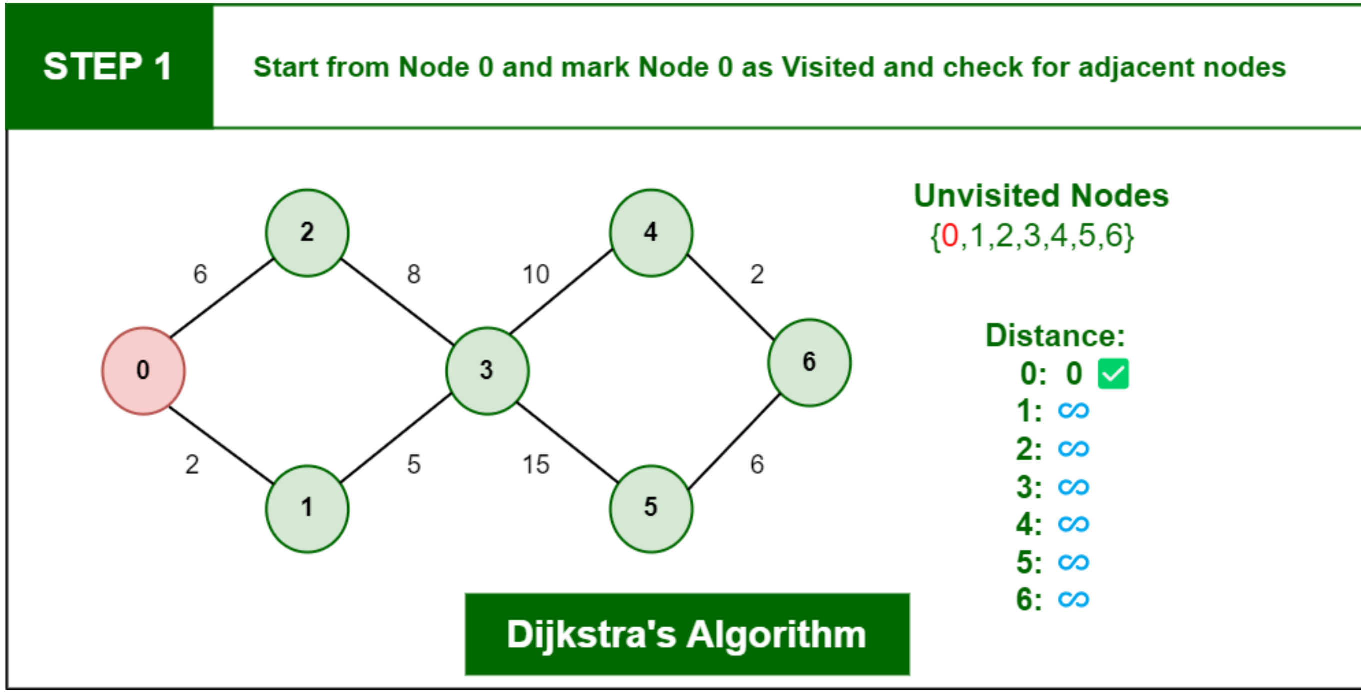
v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	~~15~~	14
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	~~14~~	13	
6	~~∞~~	~~29~~	~~26~~	25
7	~~∞~~	8		

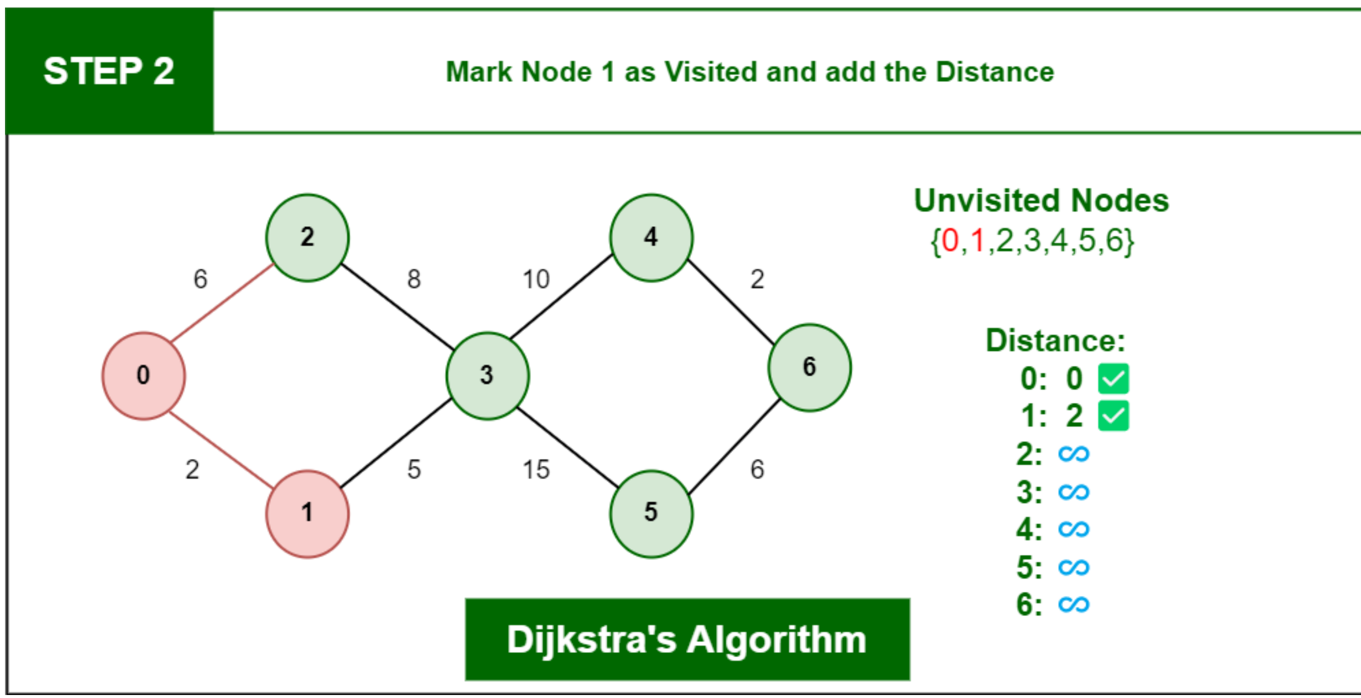
v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	~~7~~	4	
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

Dijkstra's Algorithm 3rd Example

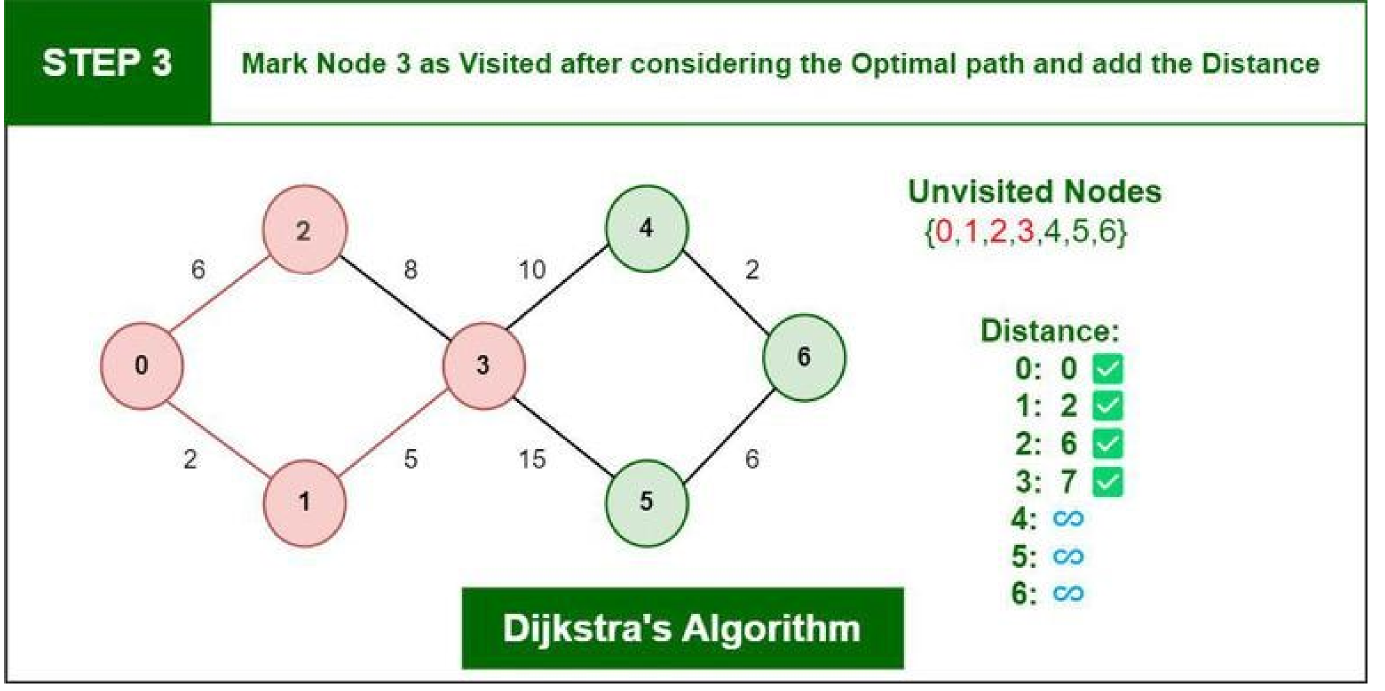


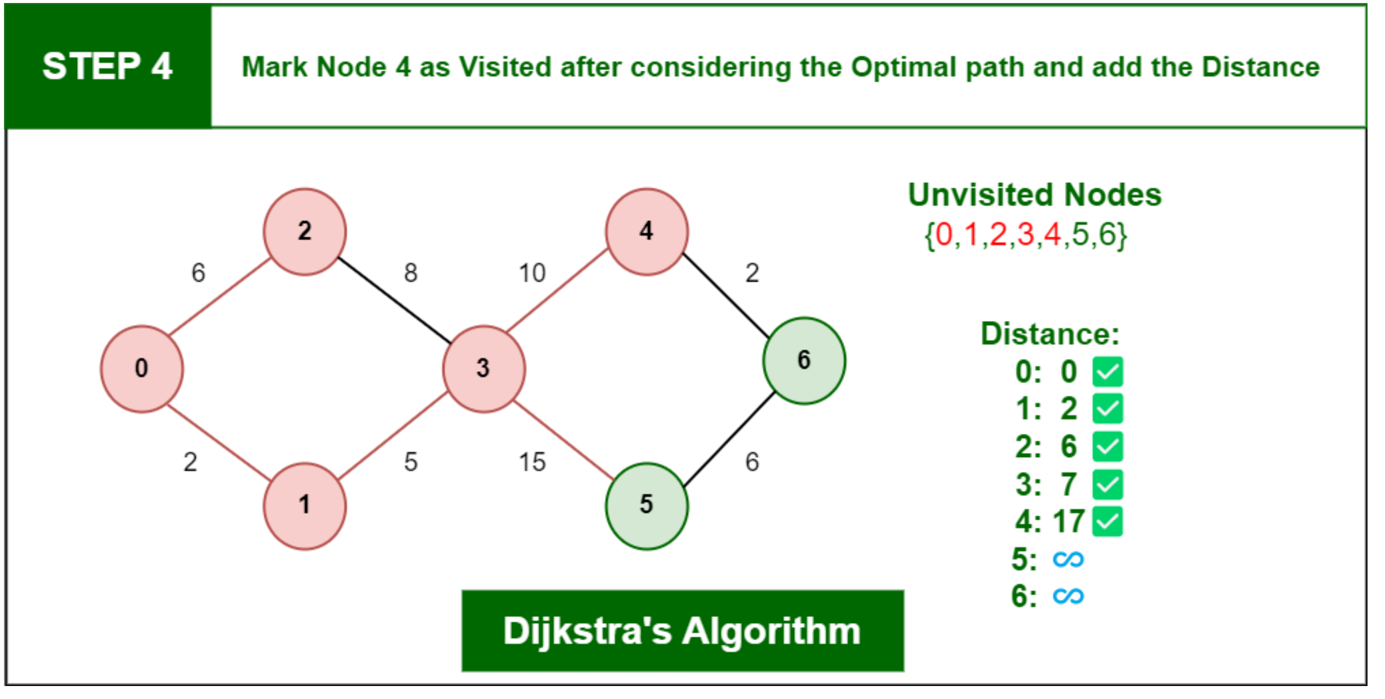
- ***Step 1:*** *Start from Node 0 and mark it as visited*
- ***Step 2:*** *Check for adjacent Nodes, Now we have 2 choices (Either Node1 with distance 2 or Node 2 with distance 6) and choose Node with minimum distance. In this step* ***Node 1*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*
- ***Distance: Node 0 -> Node 1 = 2***



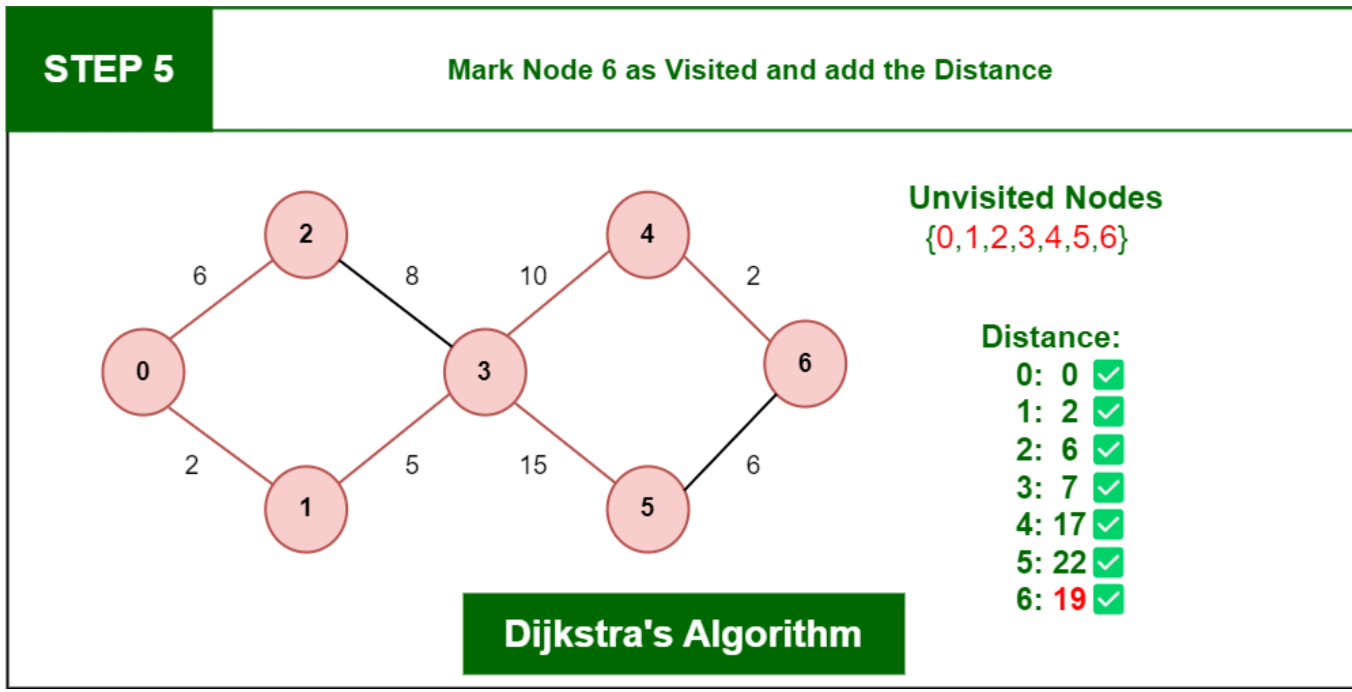


- ***Step 3:*** *Check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:*
- ***Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7***
- *Step 4: We have two choices for adjacent Nodes (Either Node 4 with distance 10 or Node 5 with distance 15) so choose Node 4, so marked it as visited and add up the distance.*
- ***Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17***





- ***Step 5:*** *Check for adjacent Node which is **Node 6**, so marked it as visited and add up the distance, Now the distance will be:*
- ***Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19***



Dijkstra's Algorithm: Correctness Proof

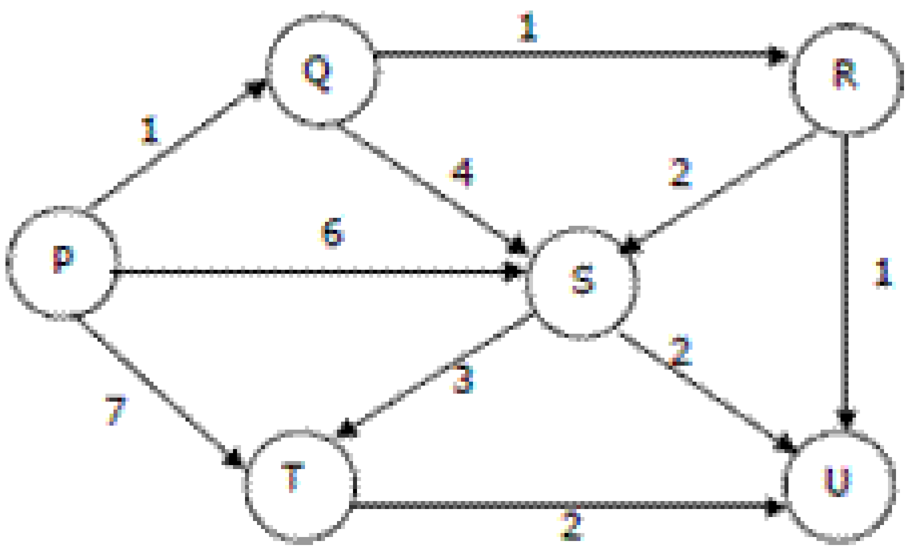
Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

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.

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



Bellman-Ford Algorithm

- Initialize distance array dist[] for each vertex '**v**' as **dist[v] = INFINITY**.
- Assume any vertex (let's say '0') as source and assign **dist = 0**.
- Relax all the **edges(u,v,weight) N-1** times as per the below condition:
  - **dist[v] = minimum(dist[v], distance[u] + weight)**

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.

```
for (int i = 1; i < G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

pass i (relax each edge)

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

Average Case: O(V*E)

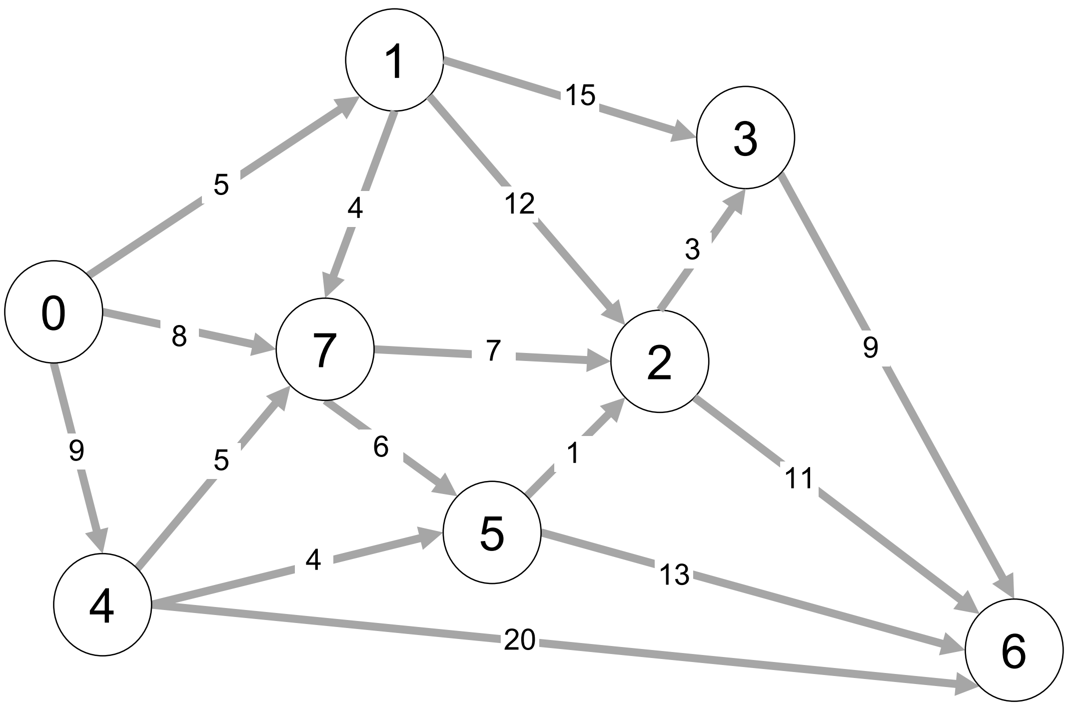
Worst Case: O(V*E)

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 Example

Repeat V − 1 times: relax all E edges.



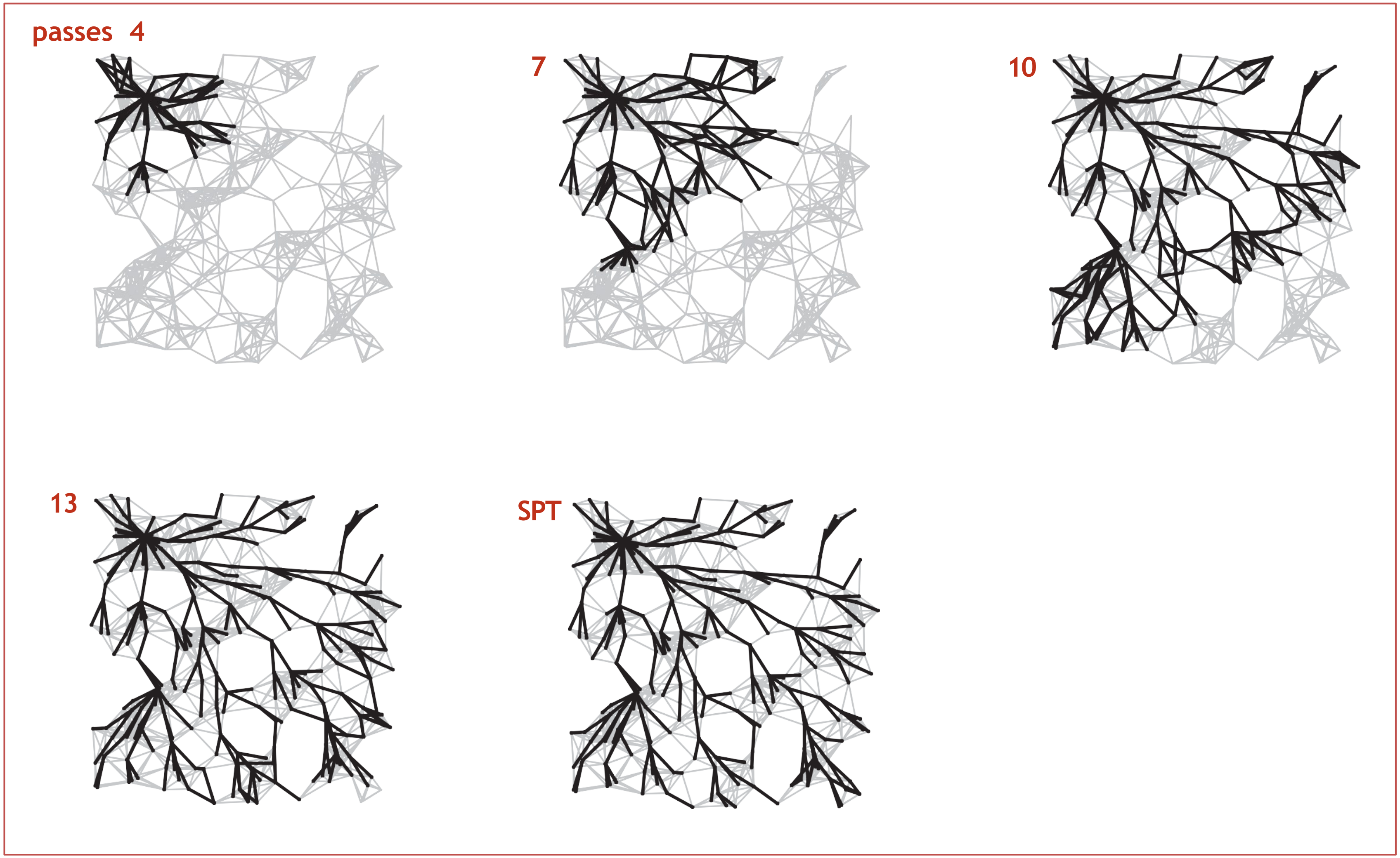
v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	14	
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~28~~	~~26~~	25
7	~~∞~~	8		

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



Dijkstra's Algorithm vs. Bellman-Ford Algorithm

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

Time complexity of Bellman-Ford

- What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of n vertices?
- ANS: O(V3)
- Time complexity of Bellman-Ford algorithm is O(VE) where V is number of vertices and E is number edges . If the graph is complete, the value of E becomes O(V2). So overall time complexity becomes O(V3)

Quiz

- Given a directed graph where weight of every edge is same, we can most efficiently find shortest path from a given source to destination using?
  - A. Breadth First Traversal
  - B. Dijkstra\'s Shortest Path Algorithm
  - C. Neither Breadth First Traversal nor Dijkstra\'s algorithm can be used
  - D. Depth First Search
- ANS: A
- With BFS, we first explore vertices at one edge distance, then all vertices at 2 edge distance, and so on.

Quiz

- Let G = (V, E) be an undirected graph with a subgraph G1 = (V1, El). Weights are assigned to edges of G as follows :

$$w(e) = \begin{cases} 0 \text{ if } e \in E_1 \\ 1 \text{ otherwise} \end{cases}$$

- A single-source shortest path algorithm is executed on the weighted graph (V, E, w) with an arbitrary vertex v1 of V1 as the source. Which of the following can always be inferred from the path costs computed?
- A. The number of edges in the shortest paths from v1 to all vertices of G
- B. G1 is connected or not
- C. V1 forms a clique in G
- D. G1 is a tree
- ANS: B
- When shortest path shortest path from v1 (one of the vertices in V1) is computed. G1 is connected if the distance from v1 to any other vertex in V1 is greater than 0, otherwise G1 is disconnected.

Quiz

- Let G = (V, E) be a simple undirected graph, and s be a particular vertex in it called the source. For $x \in V$, let d(x) denote the shortest distance in G from s to x. A breadth first search (BFS) is performed starting at s. Let T be the resultant BFS tree. If (u, v) is an edge of G that is not in T, then which one of the following CANNOT be the value of d(u) – d(v)?
- A. -1 B. 0 C. 1 D. 2
- ANS: D
- Note that the given graph is undirected, so an edge (u, v) also means (v, u) is also an edge. Since a shorter path can always be obtained by using edge (u, v) or (v, u), the difference between d(u) and d(v) can not be more than 1.

Video Tutorials

- Dijkstras Shortest Path Algorithm Explained | With Example | Graph Theory
  - https://www.youtube.com/watch?v=bZkzH5x0SKU
- 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

Quiz

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

Quiz

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

Quiz

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

Quiz

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