Lecture 13
Shortest Paths

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

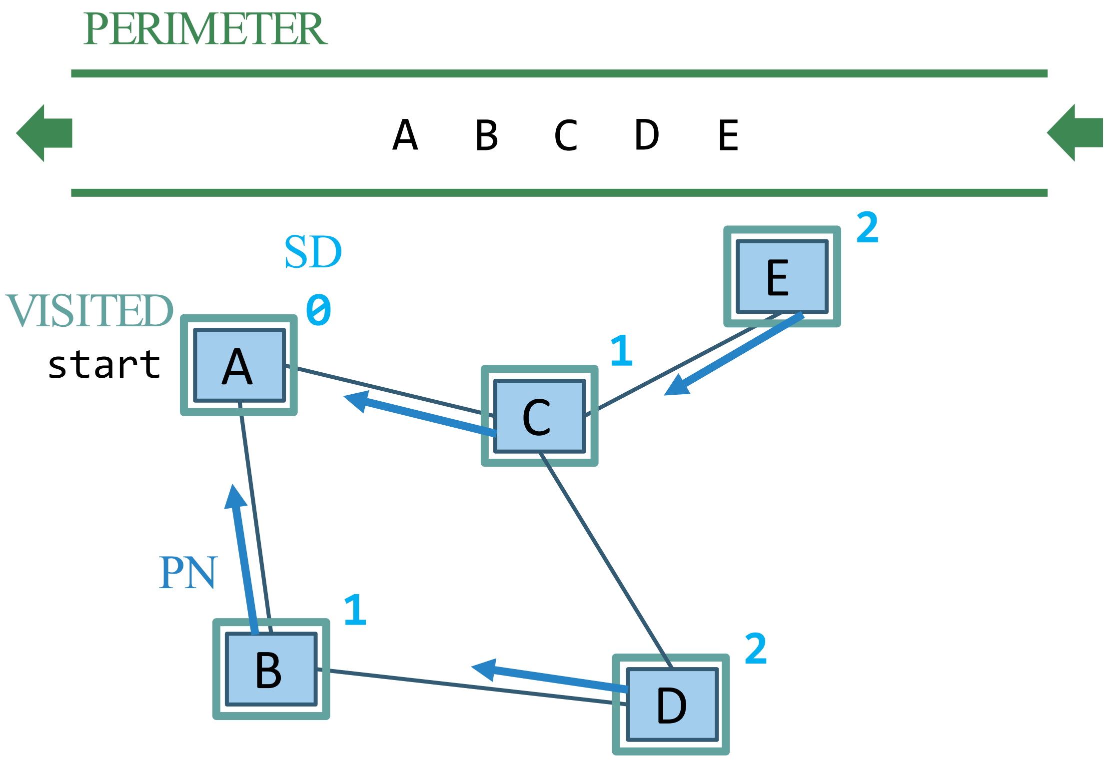
BFS for (Unweighted) Shortest Path Problem

- BFS (Breadth-First Search) can find shortest paths in an unweighted graph
  - BFS visits nodes in order of their distance from the source node, ensuring the first path found to any node is the shortest possible path in terms of the number of edges
  - Time complexity: O(V+E)
- Advantages:
  - Optimal for unweighted graphs
  - Simple implementation
- Limitations:
  - Only works for unweighted graphs

(Unweighted) Shortest Path Problem
Given source node s (start) and a target node t, how long is the shortest path from s to t? What edges makeup that path?

BFS for Shortest Paths in an Unweighted Graph

Keep track of how far each node is from the start with two maps
SD: Shortest Distance from source node
PN: Previous Node stores backpointers: each node remembers what node was used to arrive at it

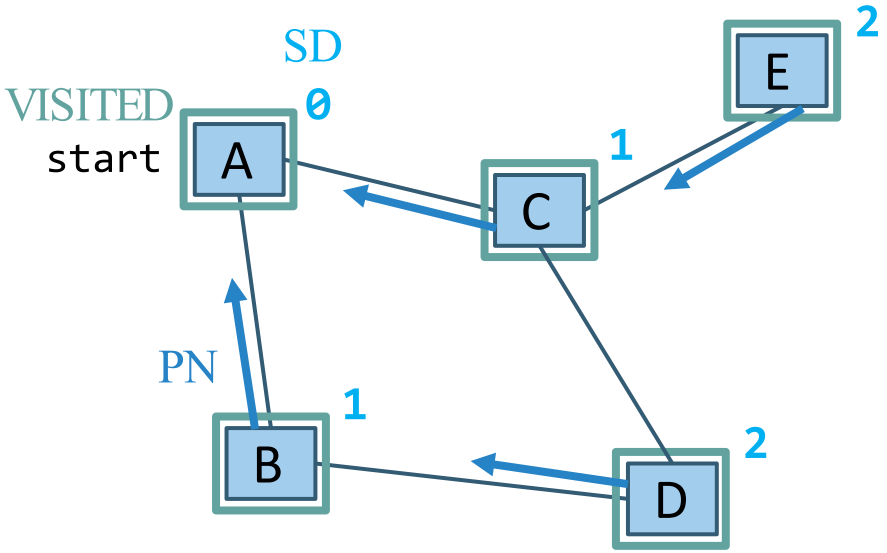



```
  ...
  Map<Node, Edge> PN = ...
  Map<Node, Double> SD = ...

  PN.put(start, null);
  SD.put(start, 0.0);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Node to = edge.to();
      if (!visited.contains(to)) {
        PN.put(to, edge);
        SD.put(to, SD.get(from) + 1);
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
  return PN;
}
```

Shortest Path Tree



Node	SD	PN
A	0	/
B	1	A
C	1	A
D	2	B or C
E	2	C

The table of SD/PN contains the Shortest Path Tree, which encodes the shortest path and distance from start to every other node

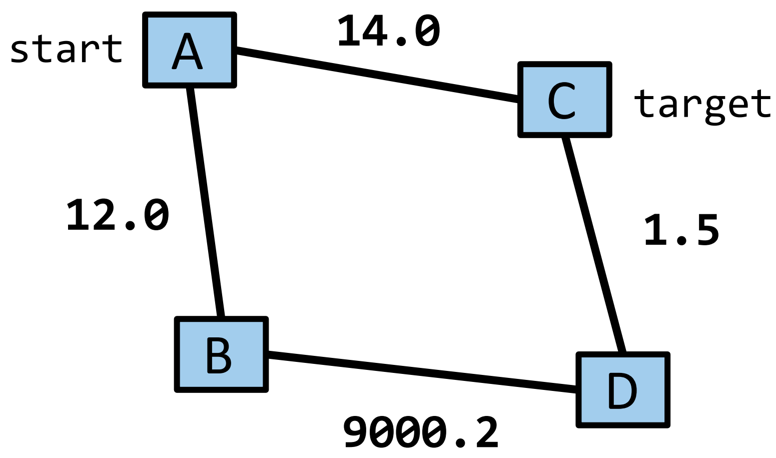
Shortest path to any node can be obtained from SPT

- Length of shortest path from A to D?
  - Lookup in SD map: 2
- What's the shortest path from A to D?
  - Build the path backwards from PN: start at D, follow backpointer to B, follow backpointer to A – the shortest path is ABD

Depending on the order of visiting A's successors with BFS: either B before C, or C before B, D's PN may be either B or C

Dijkstra's Algorithm

- Named after its inventor, Edsger W. Dijkstra (1930-2002)
  - 1972 Turing Award
- Solves the Shortest Path Problem on a weighted graph



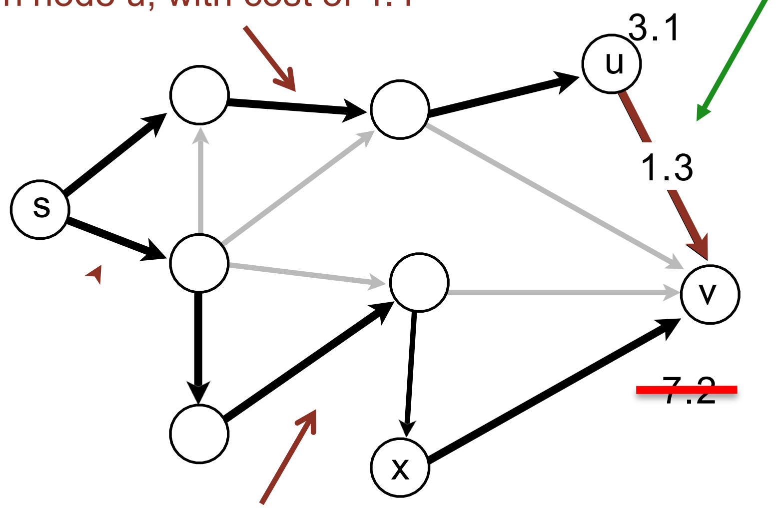
Edge Relaxation

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

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

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

PN[v]=u



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

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

OLD PN(v)=x, SD[v] = 7.2 > SD[u] + w(u,v) = 3.1+1.3 = 4.4
NEW SD[v] ← SD[u] + w(u,v) = 4.4, PN[v] = u

Generic Shortest-paths Algorithm

Generic algorithm (to compute SPT from s)

For each node v: SD[v] = ∞.

For each node v: PN[v] = null.

SD[s] = 0.

Repeat until done:

- Relax any edge.

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

Pf.

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

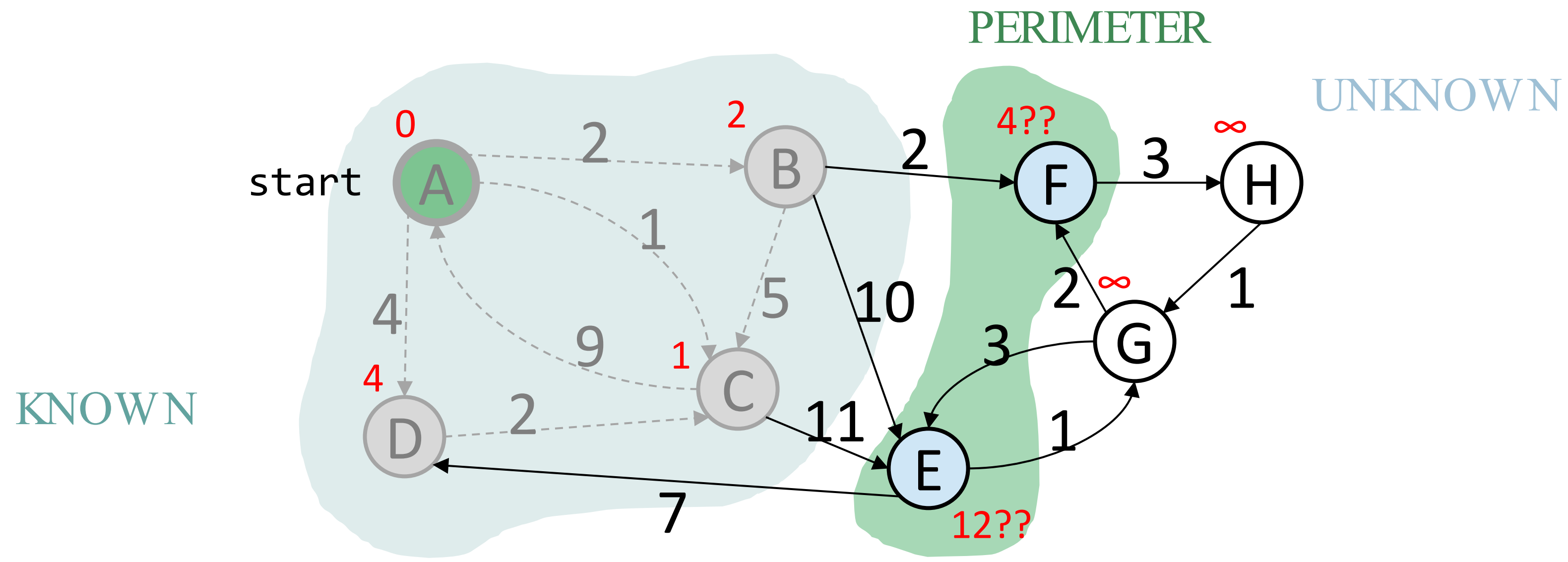
Efficient implementations. How to choose which edge to relax?

- 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

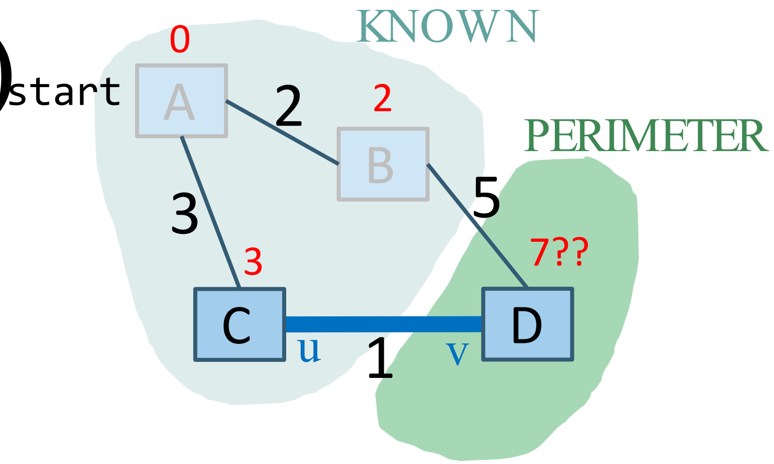
- Initialization:
  - Set the distance to the source node as 0 and to all other nodes as infinity.
  - Mark all nodes as unvisited and store them in a priority queue.
- Main Loop:
  - Visit the unvisited node u with the shortest known distance from the queue.
  - For each unvisited neighbor node v of node u, calculate its tentative distance through the current node. If this distance is smaller than the previously recorded distance, update it with edge relaxation for edge uv.
  - Mark the current node as visited once all its neighbors are processed.
- Termination:
  - The algorithm continues until all reachable nodes are visited.
- Notes:
  - Greedy and optimal algorithm: any node that has been visited should have its shortest distance to the source.
  - It works for both undirected and directed graphs. The only difference is how to get neighbors of node v, as each undirected edge is treated as two directed edges in both directions.

Dijkstra's Algorithm: Idea



- Initialization:
  - Start node has distance 0; all other nodes have distance ∞
- At each step:
  - Pick closest unknown node v
  - Add it to the "cloud" of known nodes (set of nodes whose shortest distance has been computed)
  - Update "best-so-far" distances for nodes with edges from v

Dijkstra's Pseudocode (High-Level)



- Suppose we already visited B, SD[D] = 7
- Now considering edge (C, D):
  - `oldDist = 7`
  - `newDist = 3 + 1`
  - That's better! Update SD[D], PN[D]

Similar to "visited" in BFS, "known" is set of nodes that have been visited and we know shortest paths to them

Init all paths to infinite.

Greedy algo: visit closest node first!

Consider all nodes reachable from the newly-added node u: would getting there through u be a shorter path than their current path length?

```
dijkstraShortestPath(G graph, V start)
  Set known; Map PN, SD;
  initialize SD with all nodes mapped to ∞, except start to 0

  while (there are unknown nodes):
    let u be the closest unknown node
    known.add(u);
    for each edge (u,v) from u with weight w:
      oldDist = SD.get(v)      // previous best path to v
      newDist = SD.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        SD.put(v, newDist)
        PN.put(v, u)
```

Dijkstra's Algorithm: Key Properties

Once a node is marked known, its shortest path is known. Can reconstruct path by following back-pointers (in PN map)

While a node is not yet known, another shorter path might be found. We call this update relaxing the distance because it only ever shortens the current best path

Being greedy and visiting the closest node first, means that no shorter path will be found later once a node is marked known

```
dijkstraShortestPath(G graph, V start)
  Set known; Map PN, SD;
  initialize SD with all nodes mapped to ∞, except start to 0

  while (there are unknown nodes):
    let u be the closest unknown node
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = SD.get(v)      // previous best path to v
      newDist = SD.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        SD.put(v, newDist)
        PN.put(v, u)
```

Dijkstra's Algorithm: Runtime

O(V)

O(V)

O(log V) using binary min-heap implementation of a priority queue

O(E)

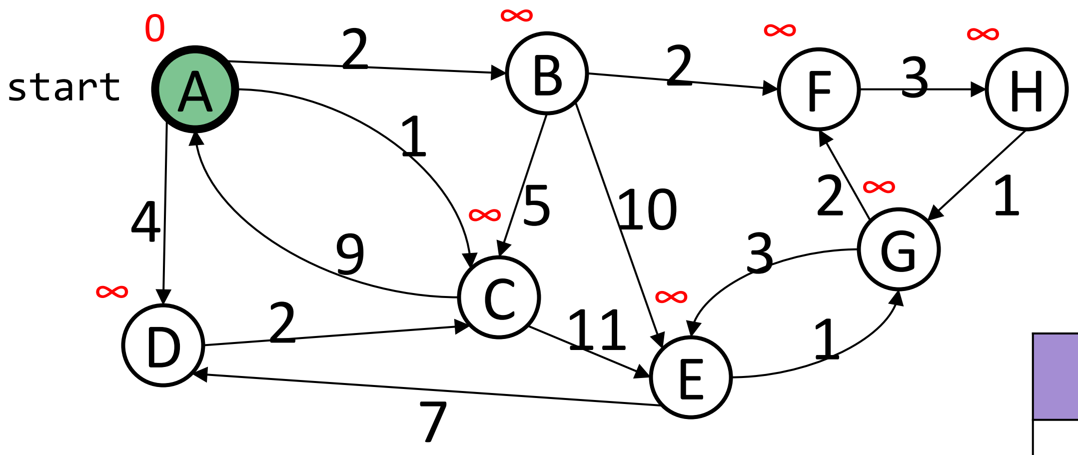
O(log V)

```
dijkstraShortestPath(G graph, V start)
  Set known; Map PN, SD;
  initialize SD with all nodes mapped to ∞, except start to 0

  while (there are unknown nodes):
    let u be the closest unknown node
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = SD.get(v)      // previous best path to v
      newDist = SD.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        SD.put(v, newDist)
        PN.put(v, u)
        update distance in list of unknown nodes
```

Initialization: O(V)
Extracting nodes: O(V log V)
Edge relaxations: O(E log V)
Total runtime: O((V+E) log V)

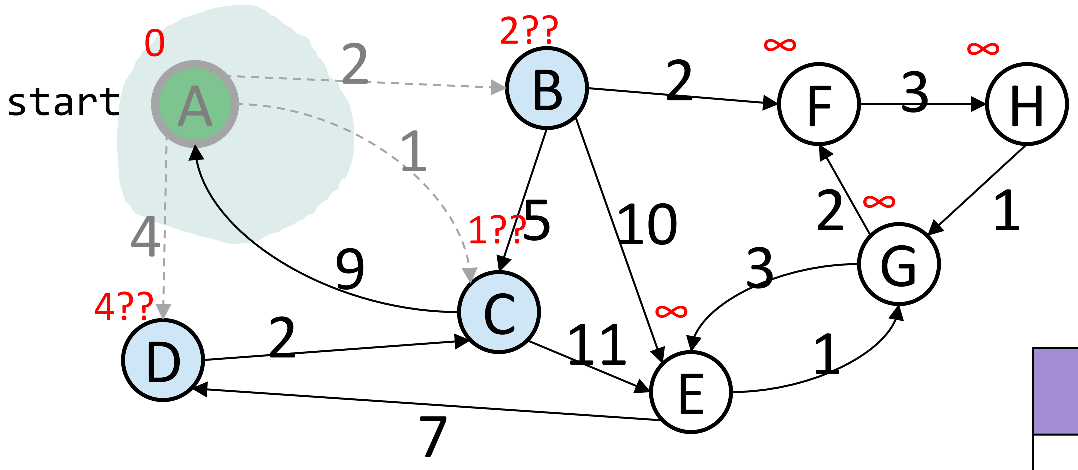
Example I



Visit Order

Node	SD	PN
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Example I

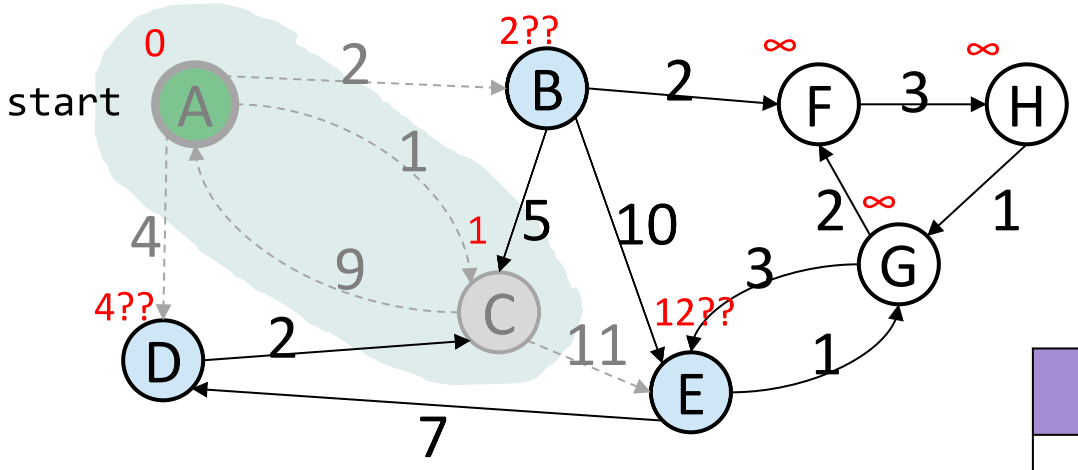


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	∞	
F	∞	
G	∞	
H	∞	

<u>Visit Order</u>

A

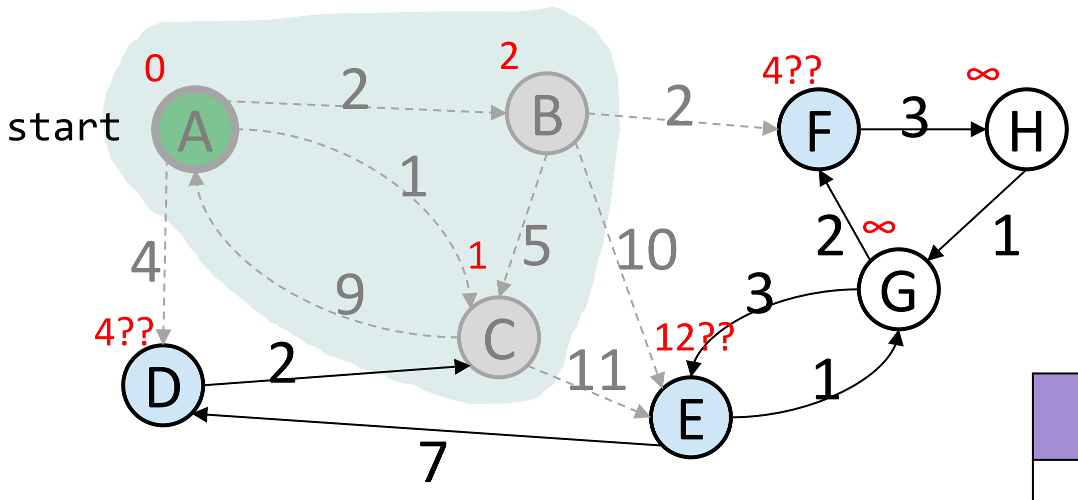
Example I



Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	∞	
G	∞	
H	∞	

<u>Visit Order</u>
A, C

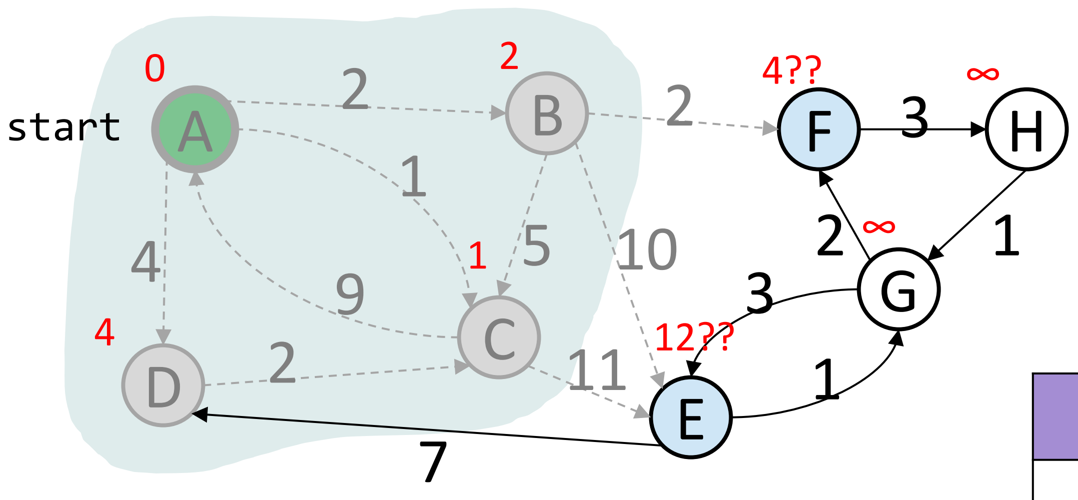
Example I



Visit Order
A, C, B

Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	∞	
H	∞	

Example I

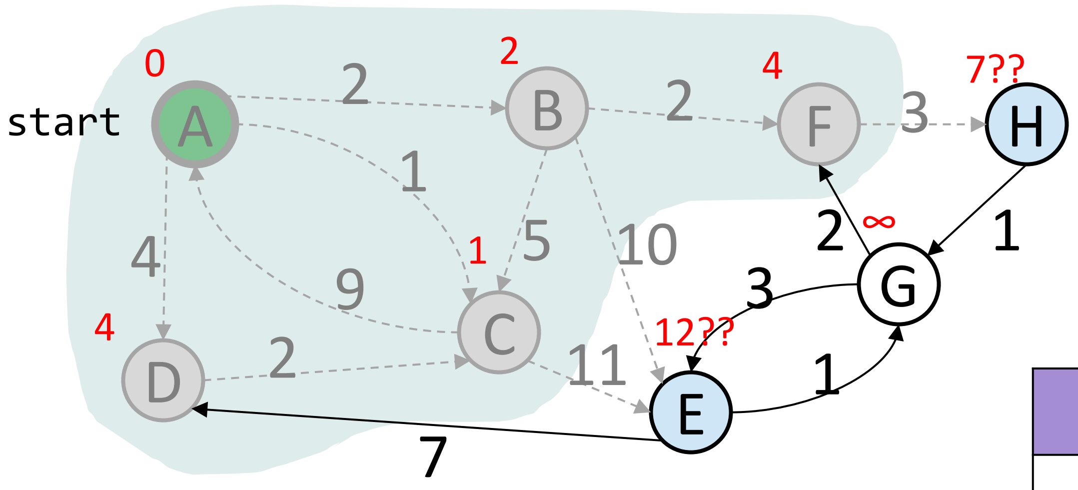


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	∞	
H	∞	

<u>Visit Order</u>

A, C, B, D

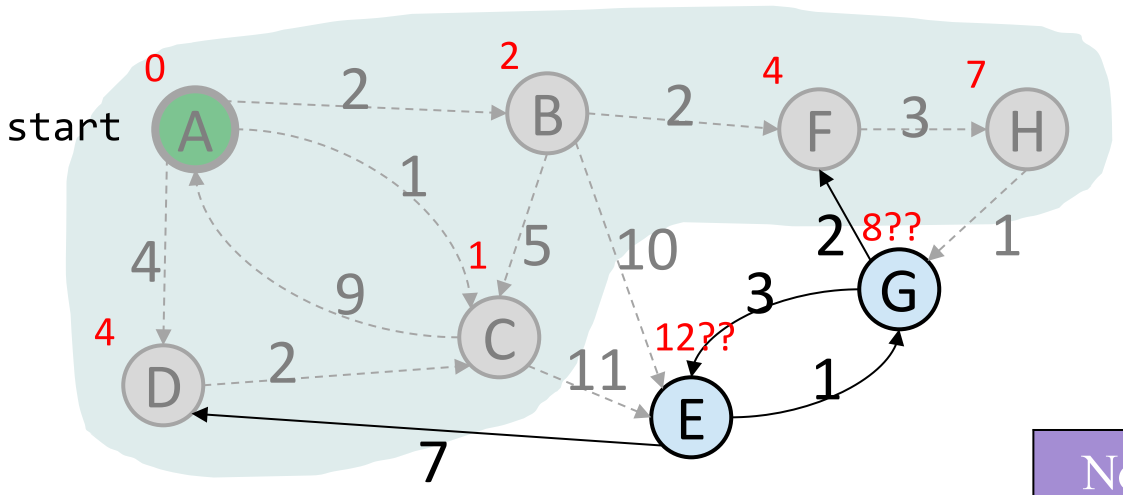
Example I



Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	∞	
H	7	F

<u>Visit Order</u>
A, C, B, D, F

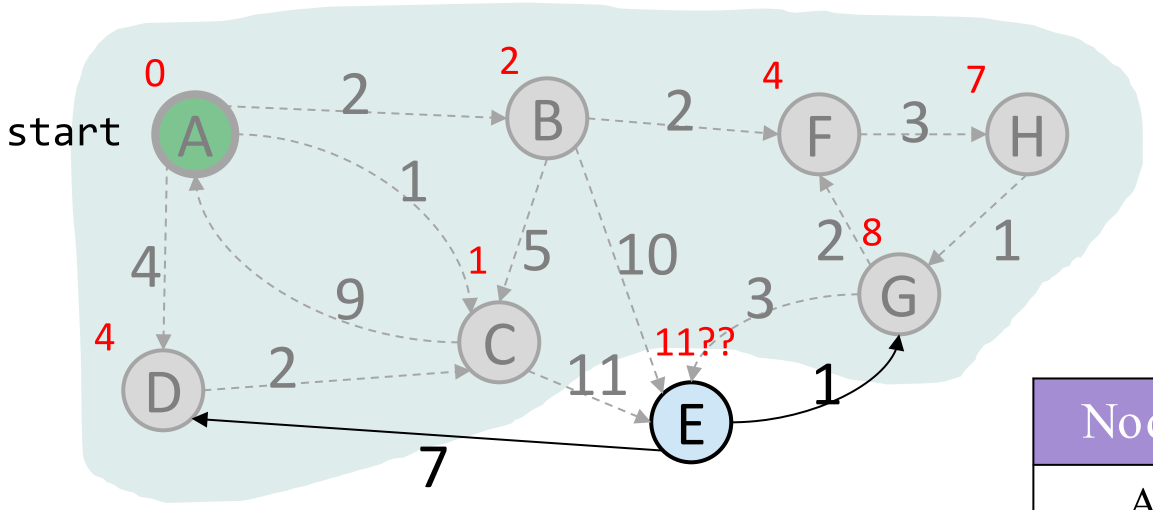
Example I



Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	8	H
H	7	F

Visit Order
A, C, B, D, F, H

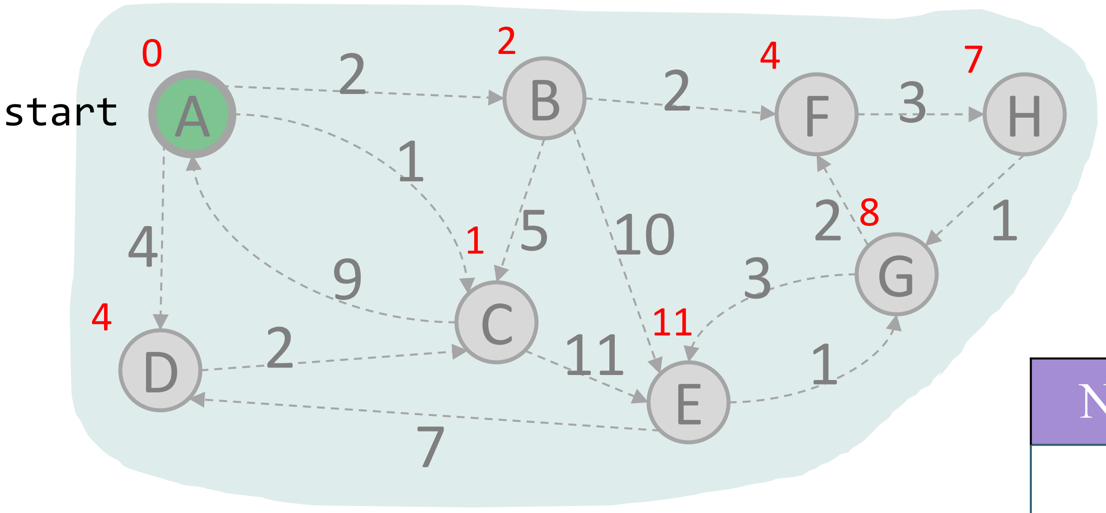
Example I



Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	11	G
F	4	B
G	8	H
H	7	F

Visit Order
A, C, B, D, F, H, G

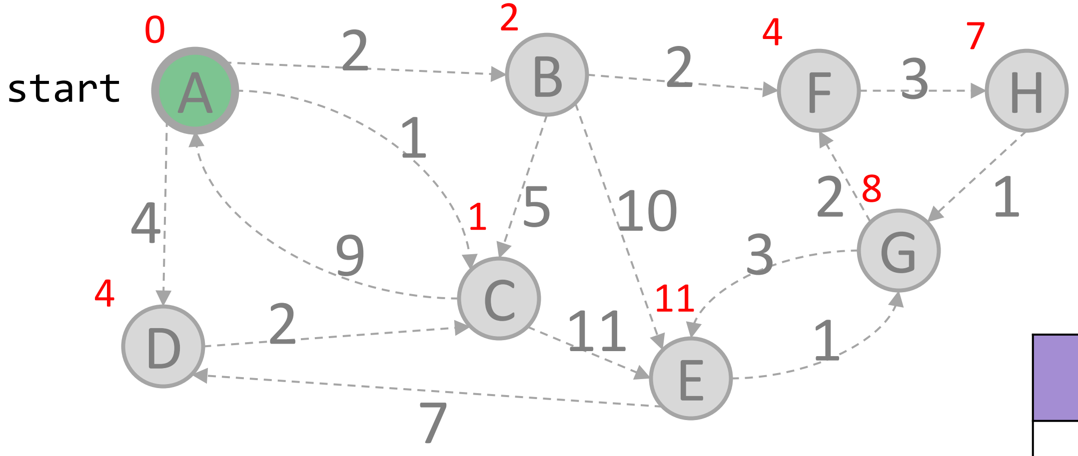
Example I



Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	11	G
F	4	B
G	8	H
H	7	F

Visit Order
A, C, B, D, F, H, G, E

Example I: Interpreting the Results



Now that we're done, how do we get the path from A to E?

- Follow PN backpointers!
- SD and PN make up the shortest path tree

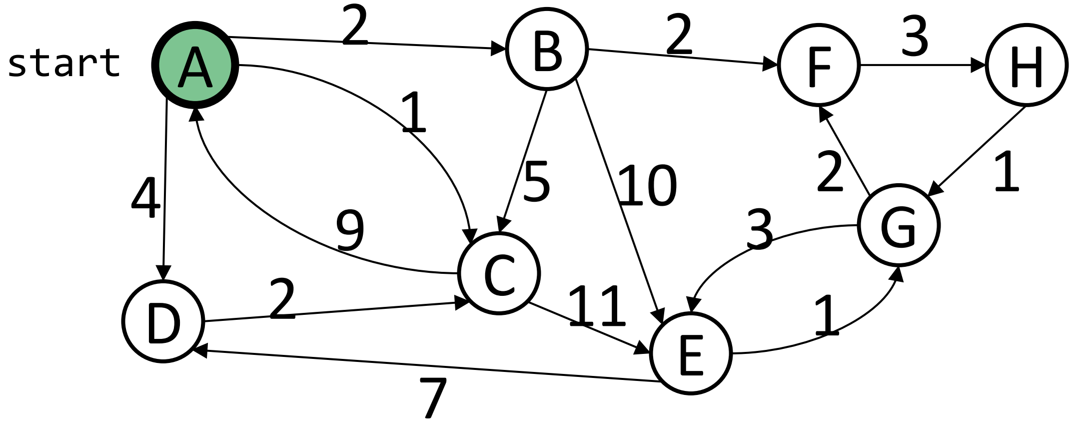
Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	11	G
F	4	B
G	8	H
H	7	F

<u>Visit Order</u>

A, C, B, D, F, H, G, E

Example I: Final Answer (for Exams)

Exam question: Given this directed graph, run Dijkstra's Algo to find shortest paths starting from source node A. Give the node visit order, and fill in this table of SN (Shortest Distance) and PN (Previous Node), crossing out old items as you find a shortcut path



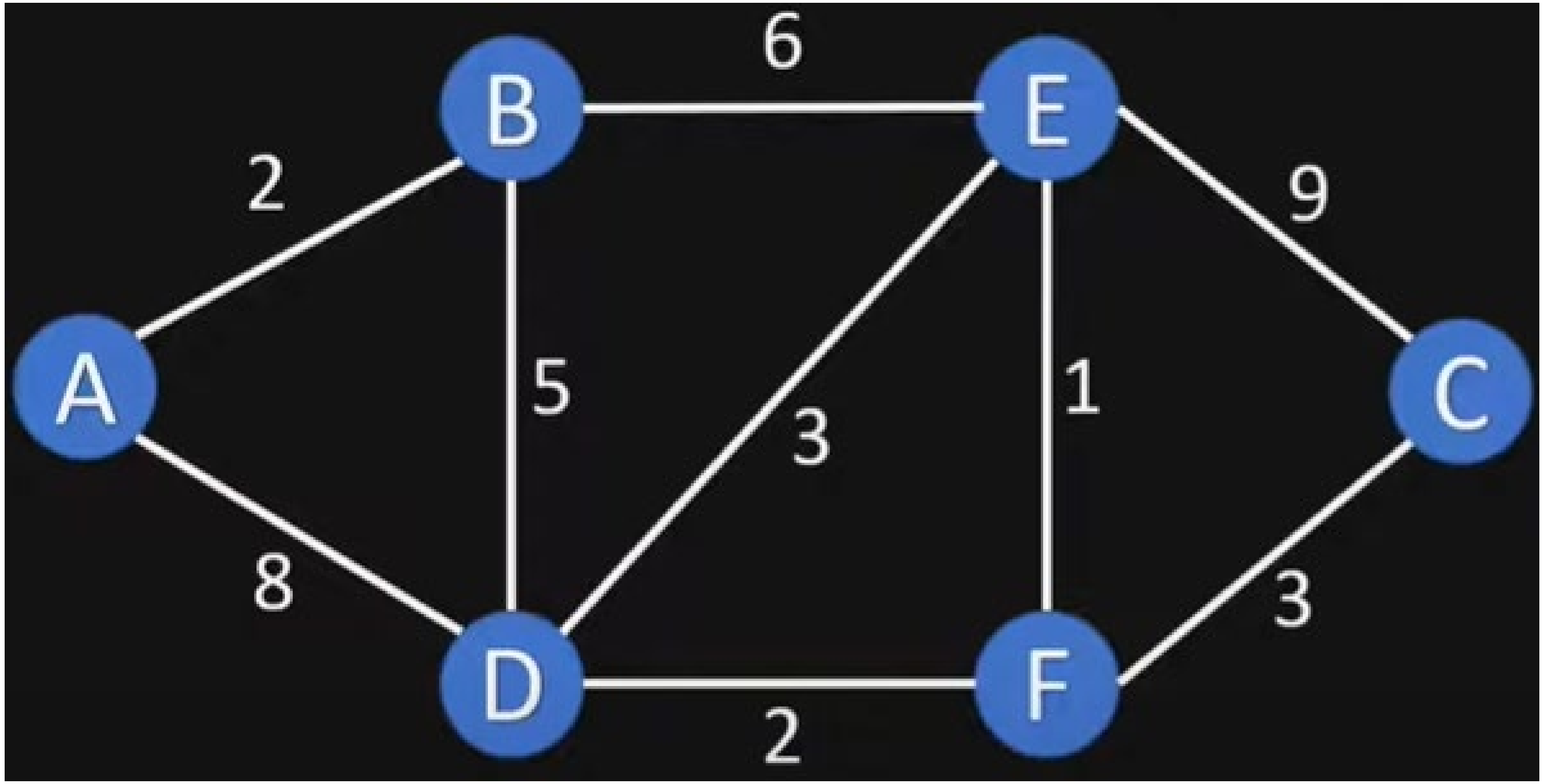
Visit Order

A, C, B, D, F, H, G, E

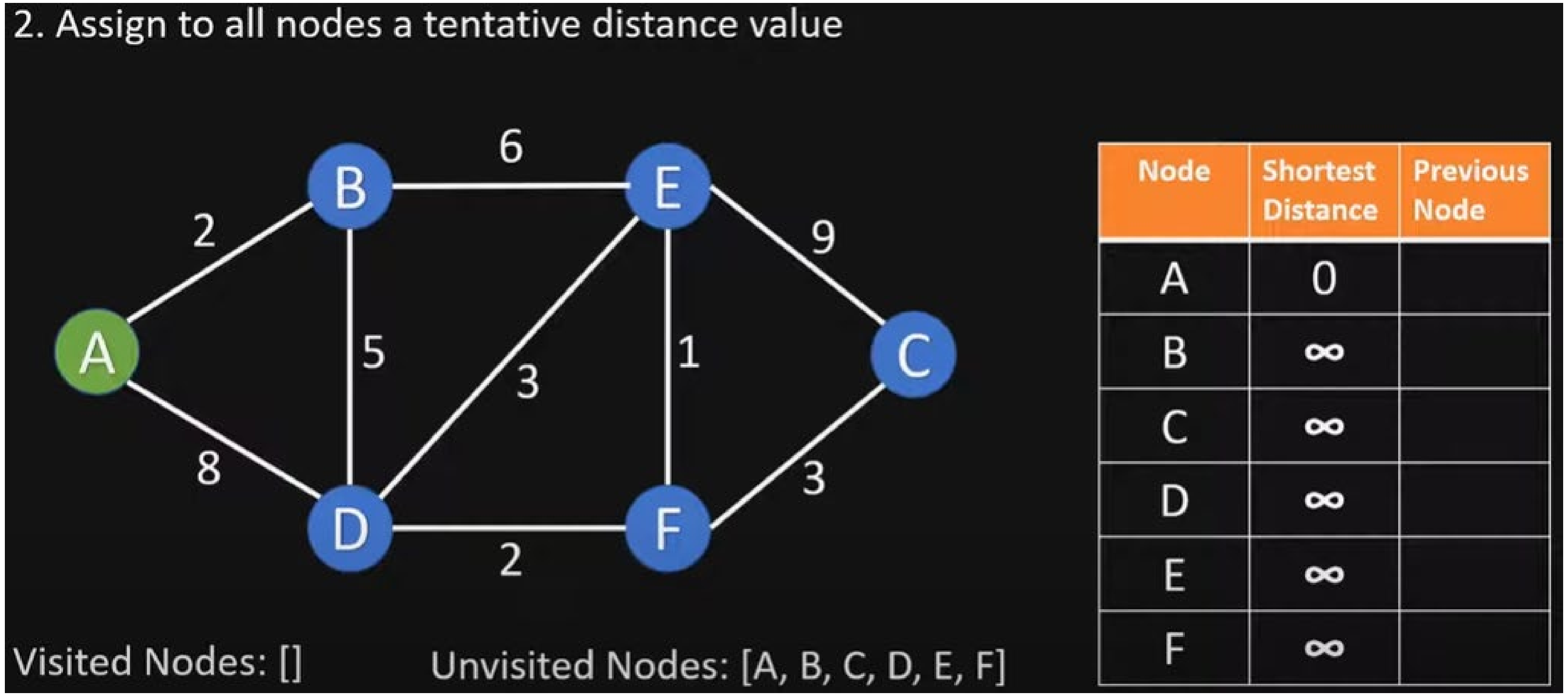
Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	~~12~~ 11	~~C~~ G
F	4	B
G	8	H
H	7	F

Example II

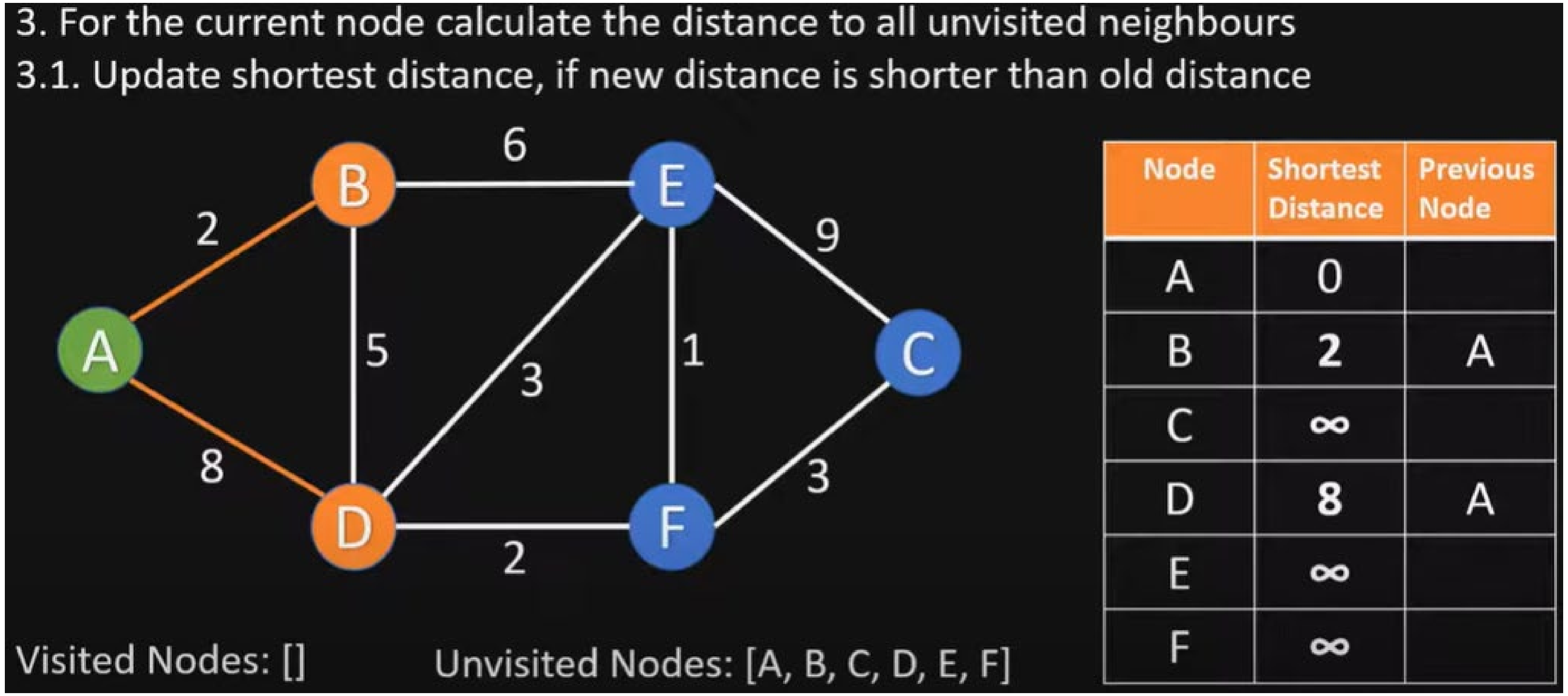
- Dijkstras Shortest Path Algorithm Explained | With Example | Graph Theory
  - https://www.youtube.com/watch?v=bZkzH5x0SKU



Initialize

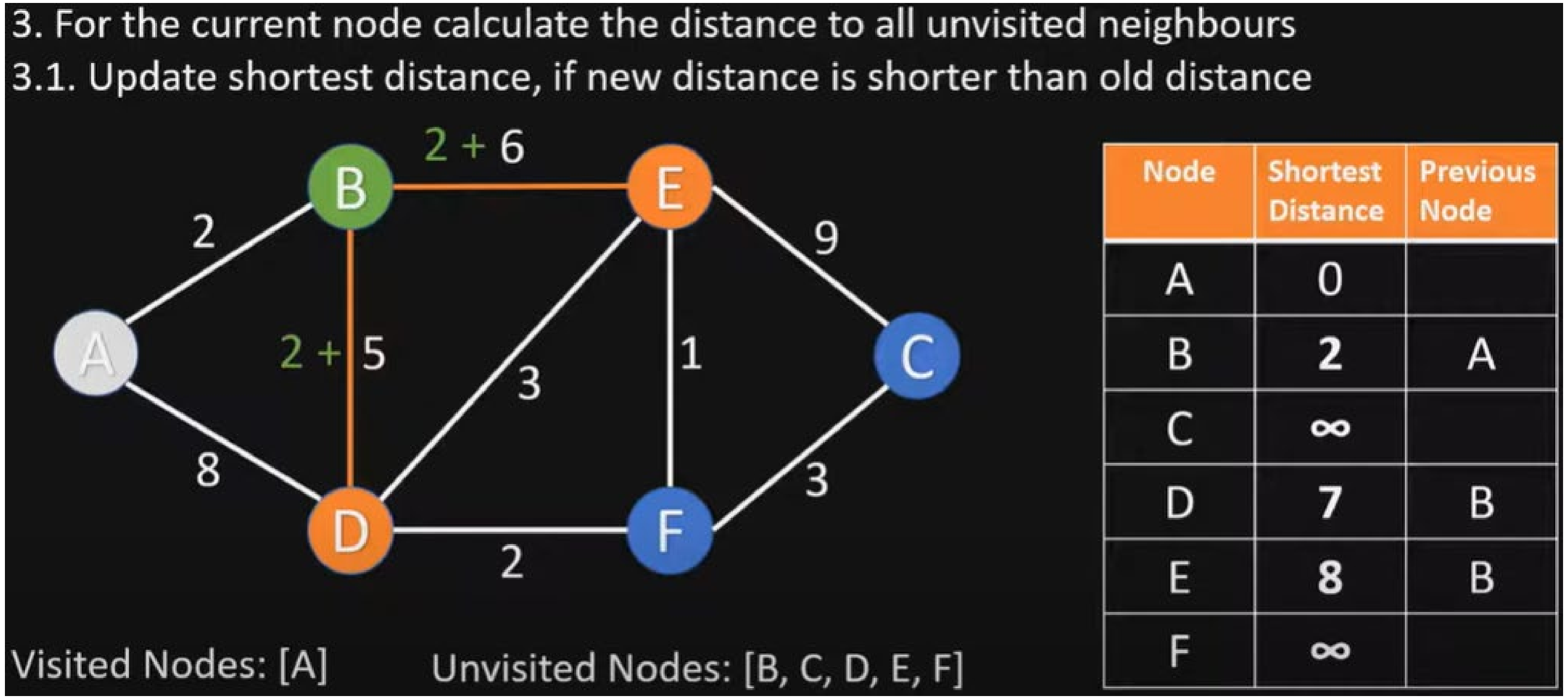

2. Assign to all nodes a tentative distance value
6
2
9
5
3
1
8
3
2
A
B
C
D
E
F
Node
Shortest Distance
Previous Node
A 0
B ∞
C ∞
D ∞
E ∞
F ∞
Visited Nodes: []
Unvisited Nodes: [A, B, C, D, E, F]

Visit node A



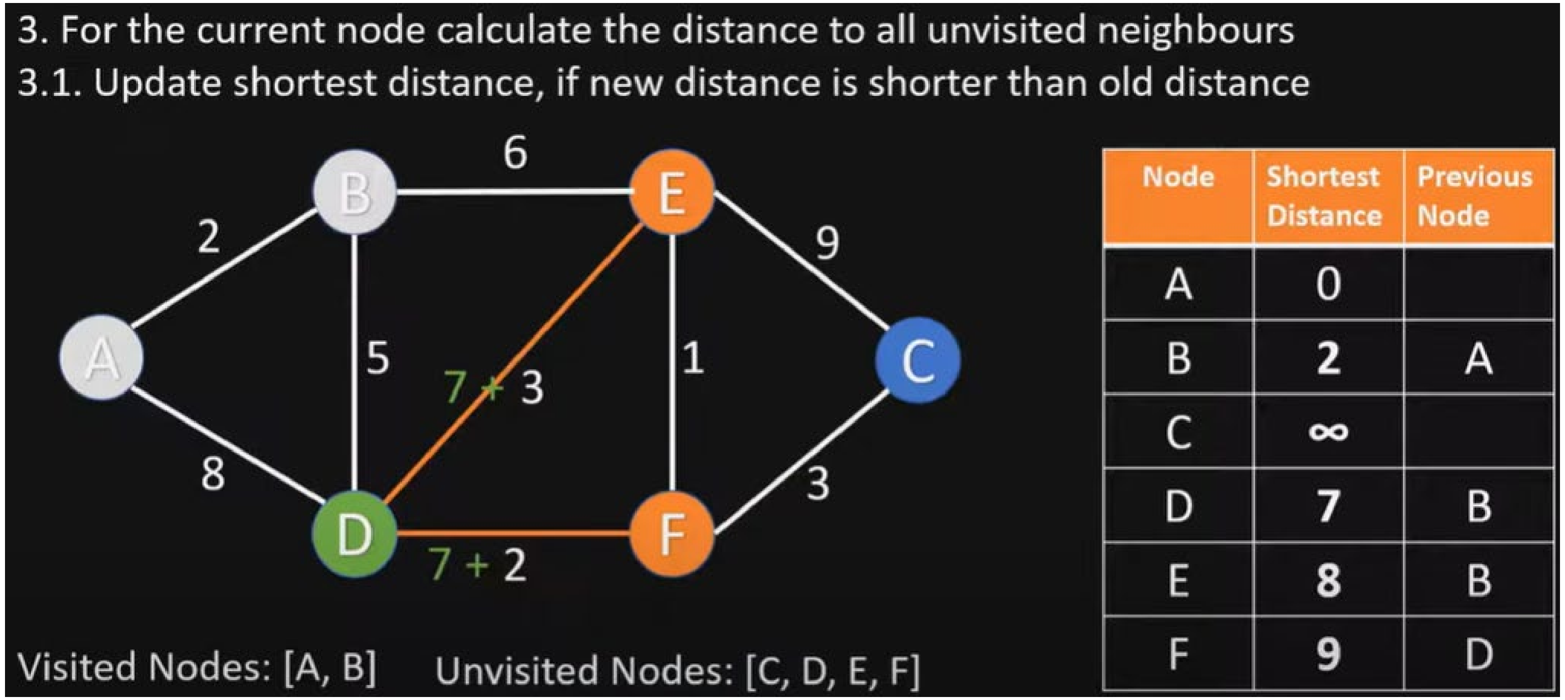
OLD SD[B] = ∞ > SD[A] + w(A,B) = 0+2 = 2
NEW SD[B] ← SD[A] + w(A,B) = 2, PN[B] = A
OLD SD[D] = ∞ > SD[A] + w(A,D) = 0+8 = 8
NEW SD[D] ← SD[A] + w(A,D) = 8, PN[D] = A

Visit node B



OLD $SD[D] = 8 > SD[B] + w(B,D) = 2+5 = 7$
NEW $SD[D] \leftarrow SD[B] + w(B,D) = 7$, $PN[D] = B$
OLD $SD[E] = \infty > SD[B] + w(B,E) = 2+6 = 8$
NEW $SD[E] \leftarrow SD[B] + w(B,E) = 8$, $PN[E] = B$

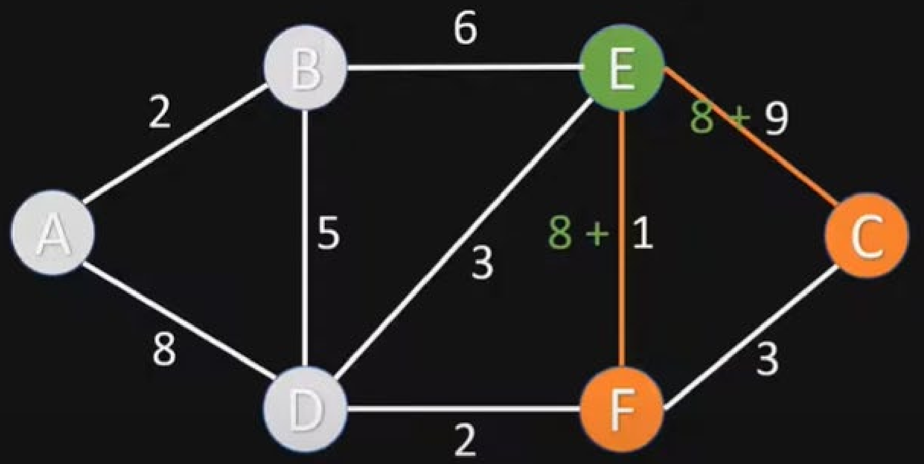
Visit node D



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

Visit node E

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

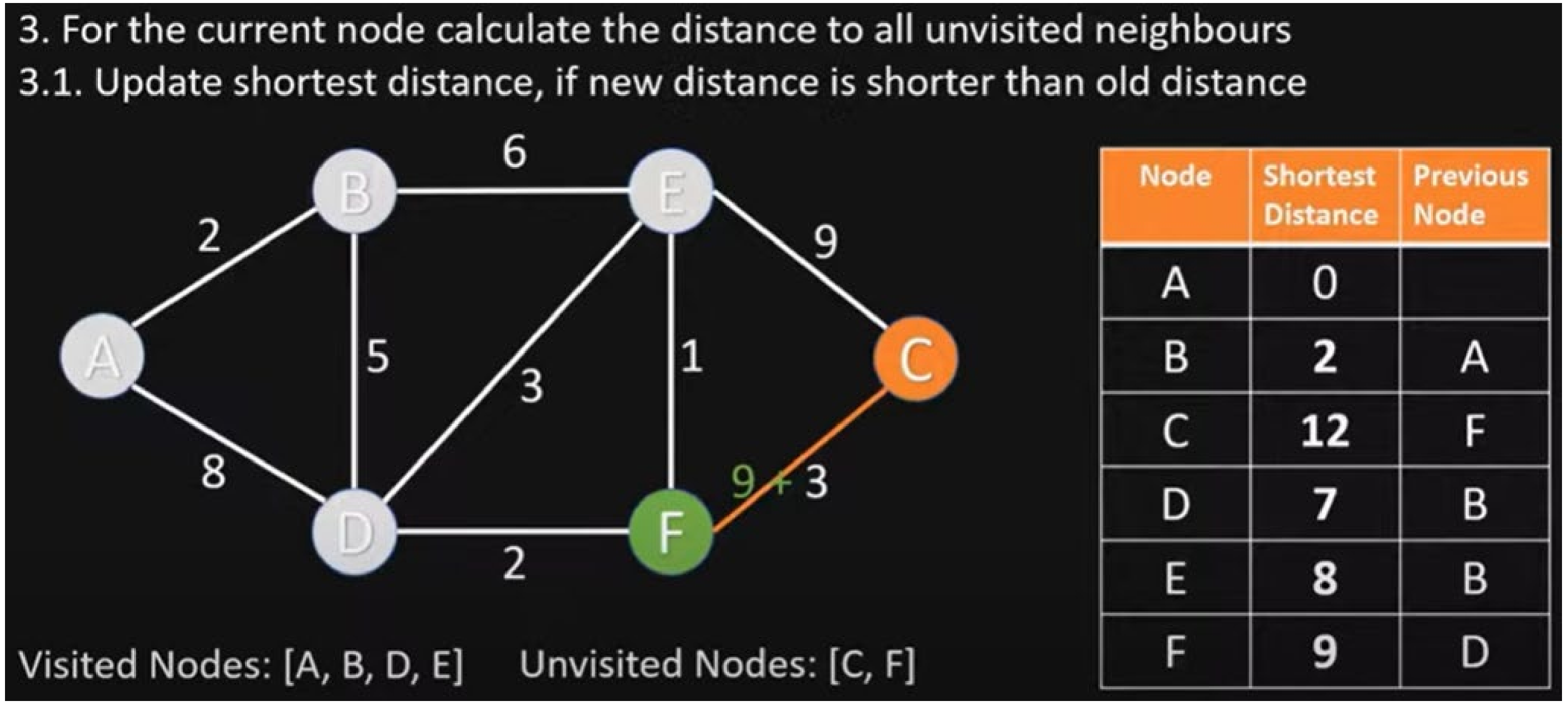


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

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

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

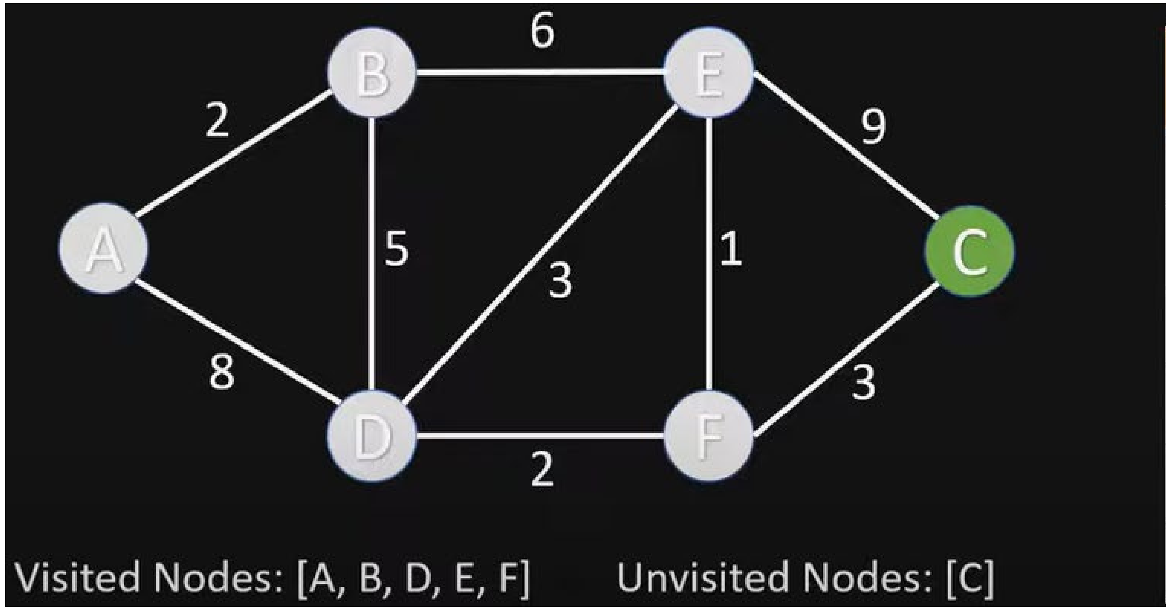
Visit node F



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

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

Visit node C



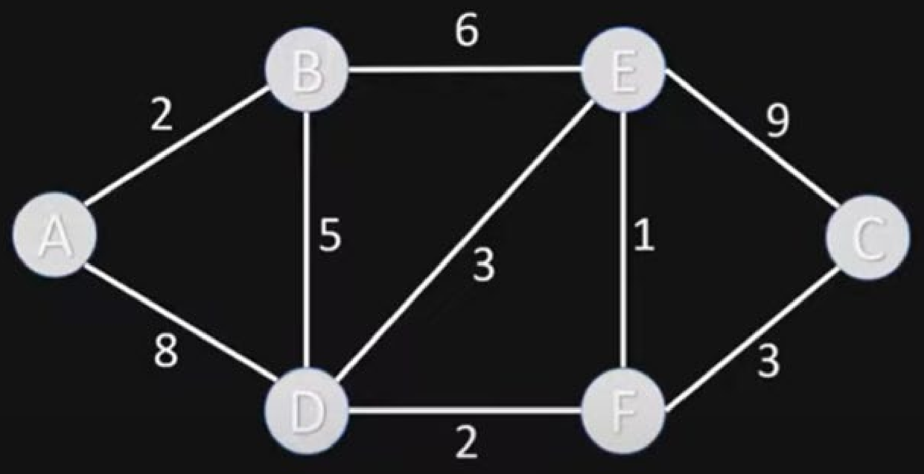
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Nothing changes, since C has no unvisited neighbor nodes

End of Algorithm

- Table now 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

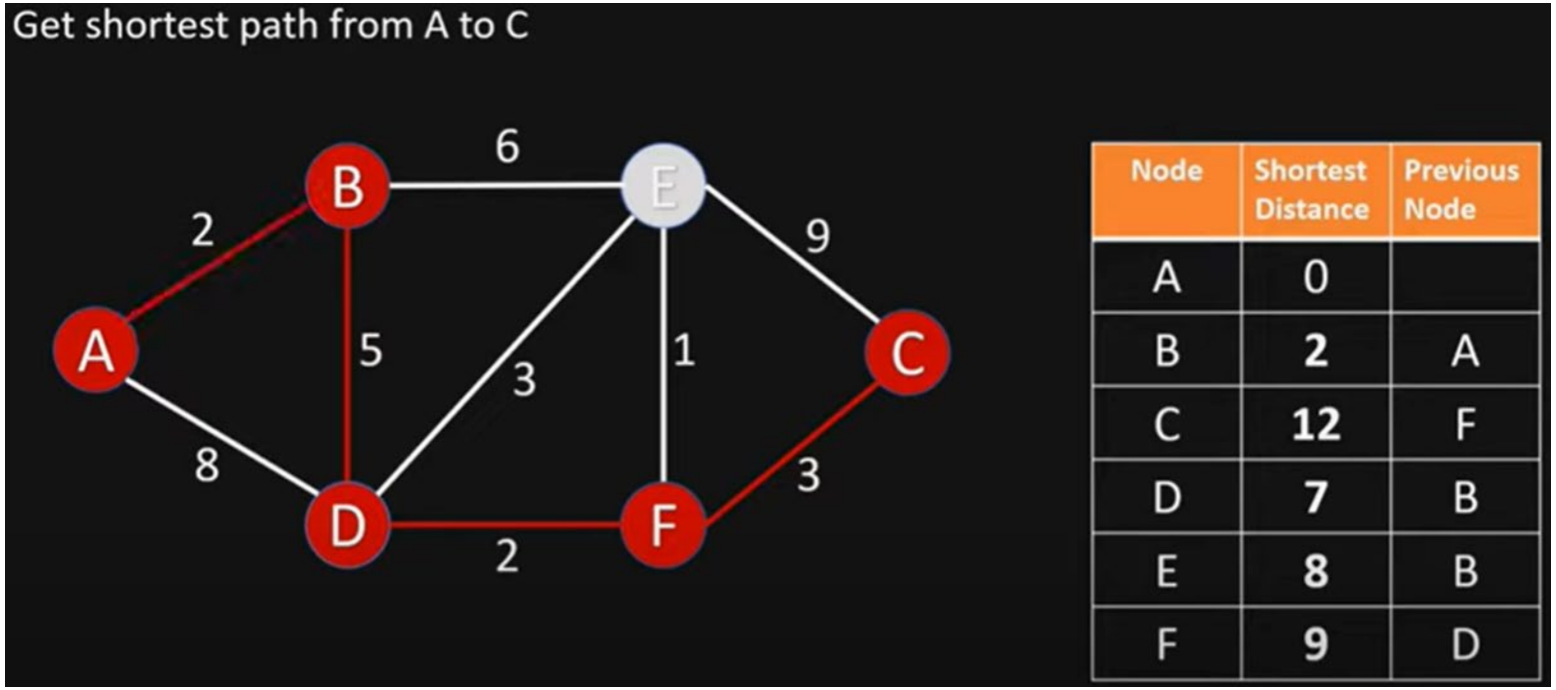


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

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

Getting the Shortest Path from A to C

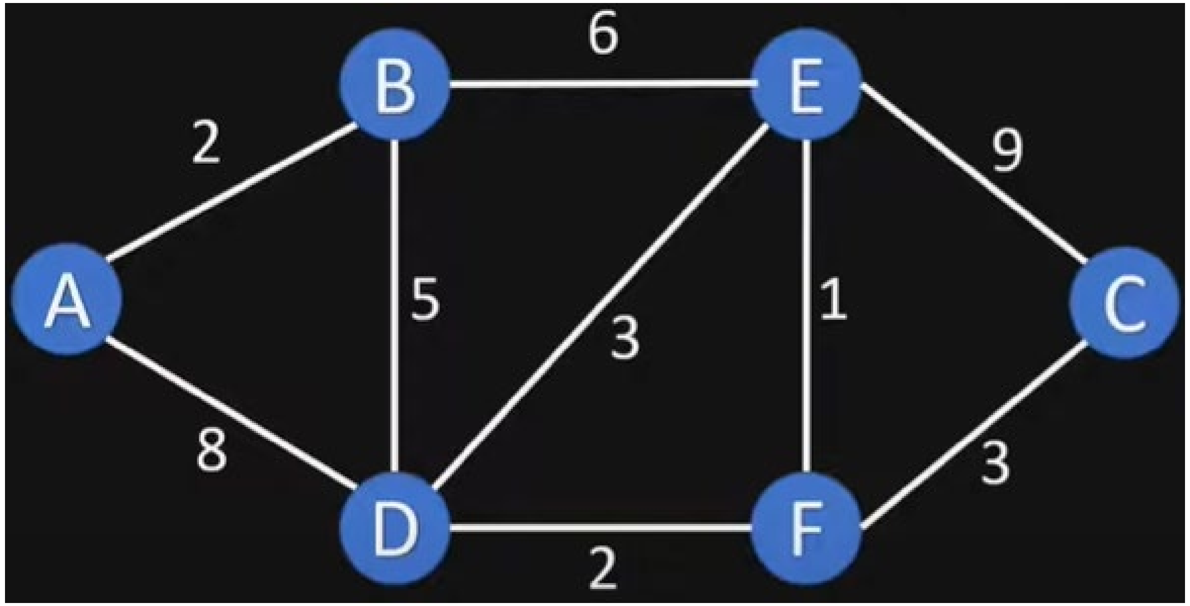
- C’s previous node is F; F’s previous node is D; D’s previous node is B; B’s previous node is A
- Shortest Path from A to C is ABDFC



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

Example II: Final Answer (for Exams)

Exam question: Given this directed graph, run Dijkstra's Algo to find shortest paths starting from source node A. Give the node visit order, and fill in this table of SN (Shortest Distance) and PN (Previous Node), crossing out old items as you find a shortcut path



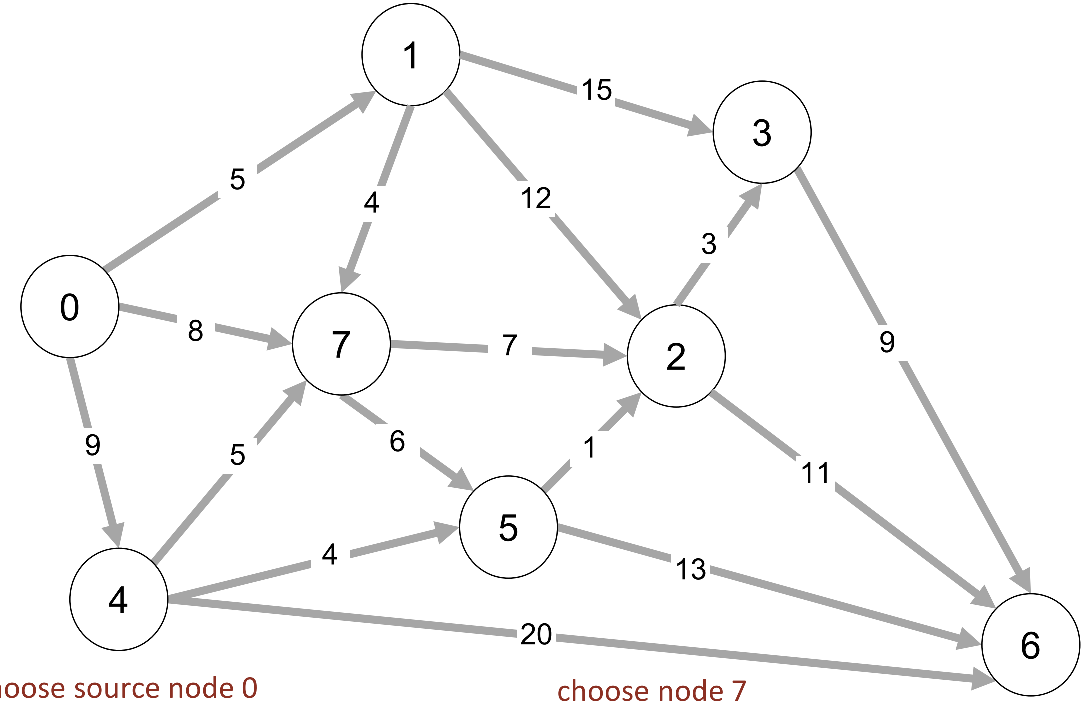
Visit Order

A, B, D, E, F, C

Node	SD	PN
A	0	/
B	2	A
C	~~17~~ 12	~~E~~ F
D	~~8~~ 7	~~A~~ B
E	8	B
F	9	D

Example III

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



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

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

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

v	PN			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	~~7~~	4	
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

Visit Order
0 , 1, 7, 4, 5, 2, 3, 6

Node	SD	PN
0	0	/
1	5	0
2	~~17~~ ~~15~~ 14	~~1~~ ~~7~~ 5
3	~~20~~ 17	~~1~~ 2
4	9	0
5	~~14~~ 13	~~7~~ 4
6	~~29~~ ~~26~~ 25	4 ~~5~~ 2
7	8	0

Topological Sort for Shortest Paths in Edge-weighted DAG

- Suppose that a graph is a Directed Acyclic Graph (DAG), i.e., it has no directed cycles. It is easier and faster to find shortest paths than in a general digraph.
- Idea: Consider nodes in topological order. Relax all outgoing edges from that node

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

  For every adjacent node v of u

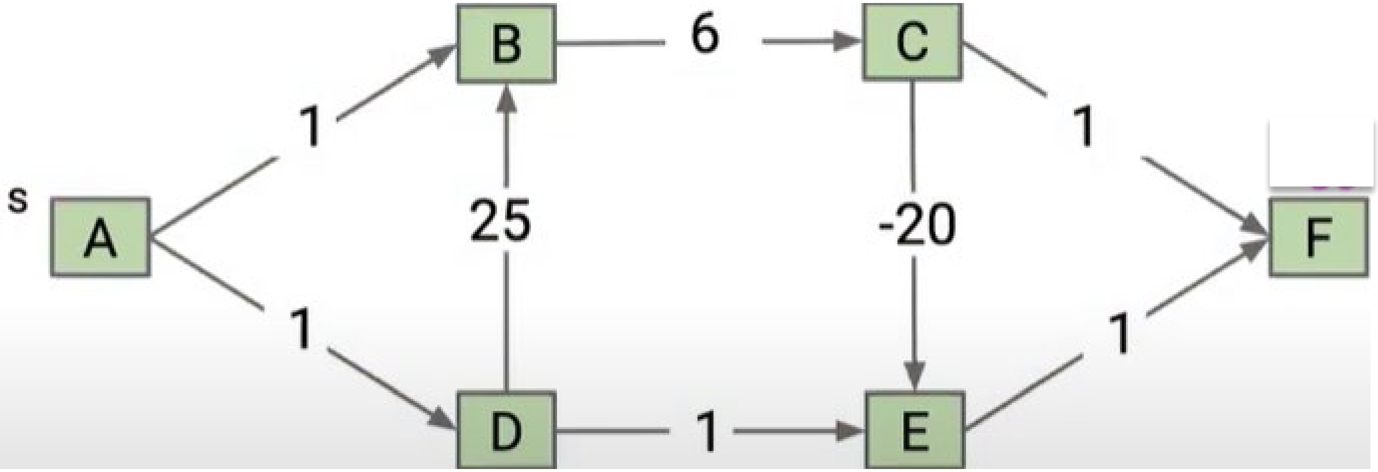
  if (dist[v] > dist[u] + weight(u, v)) //relax edge uv

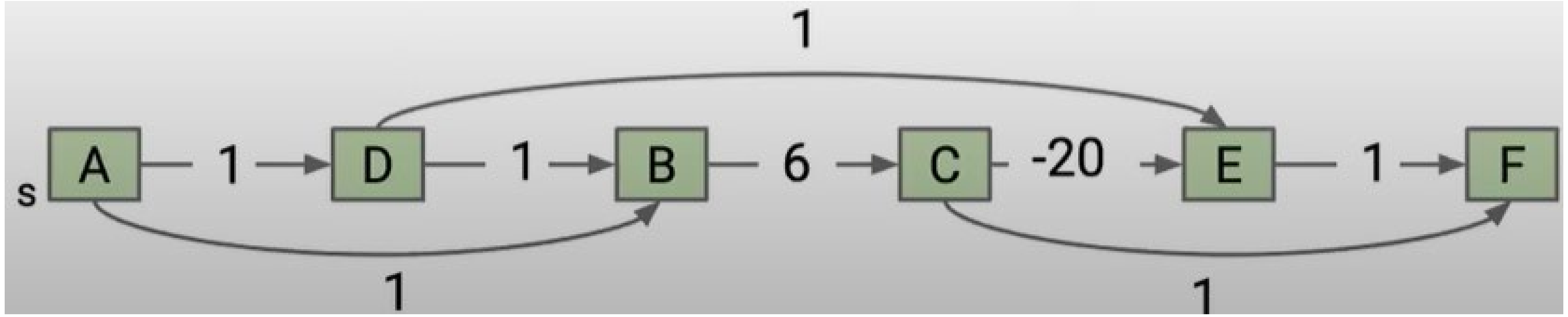
  dist[v] = dist[u] + weight(u, v)

- Time Complexity: Time complexity of topological sort is O(V+E). After finding topological order, the algorithm process all nodes and for every node, it runs a loop for all adjacent nodes. Total adjacent nodes 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 I

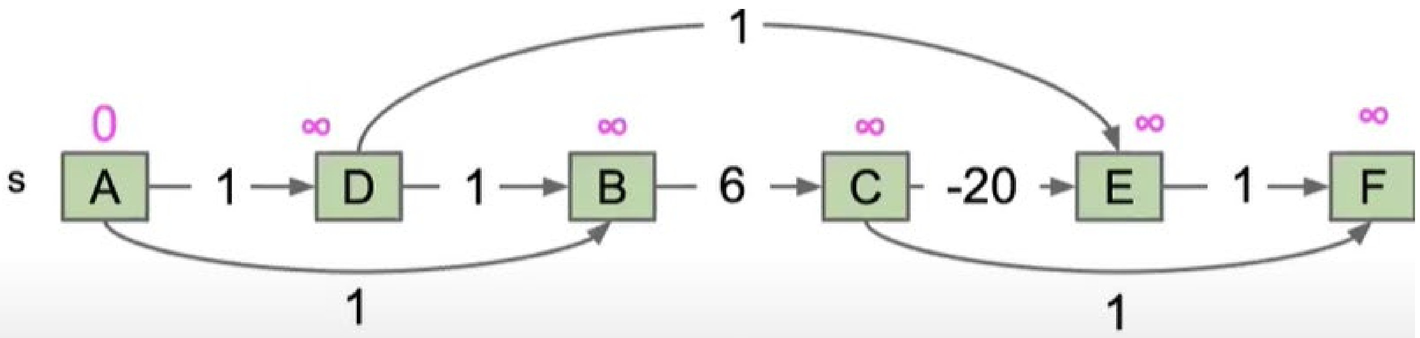
- Consider this DAG and a topological order ADBCEF





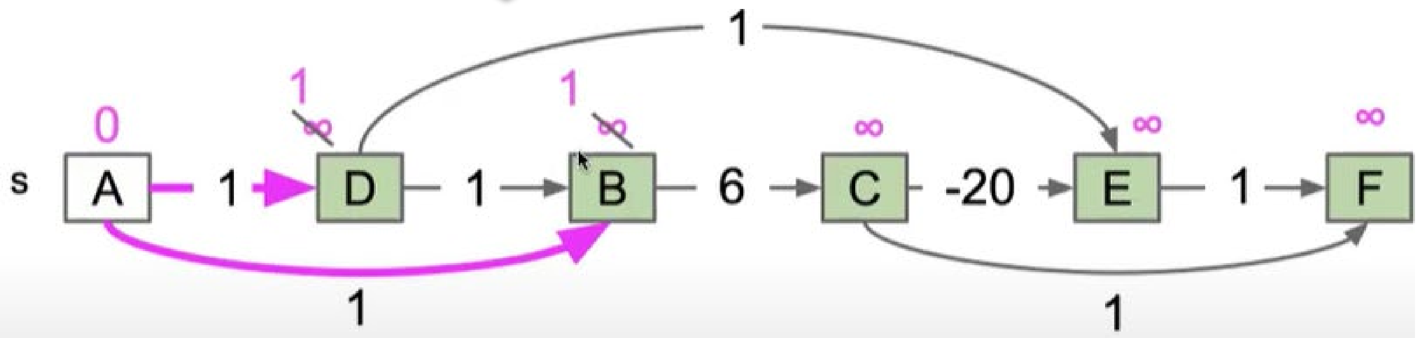
Initialize

	distTo	edgeTo
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-



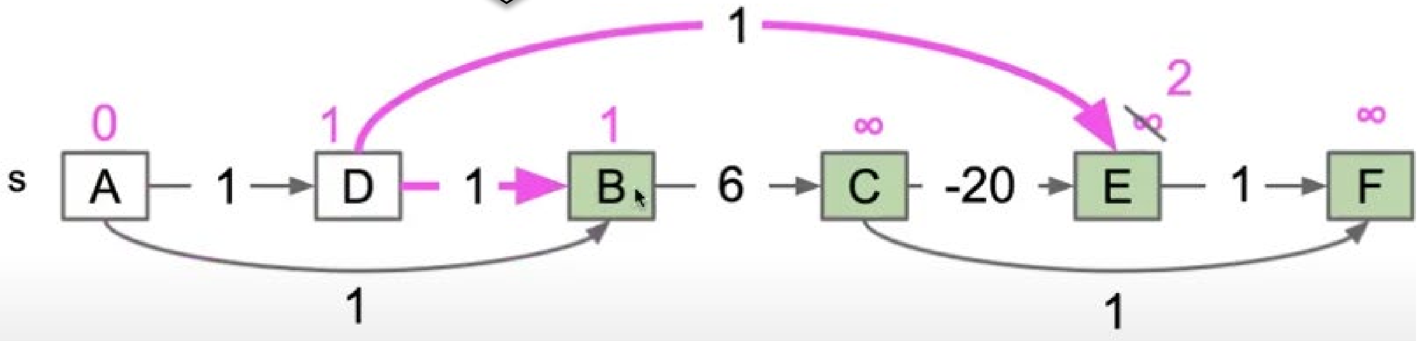
Visit A

	distTo	edgeTo
A	0	-
B	1	A
C	∞	-
D	1	A
E	∞	-
F	∞	-



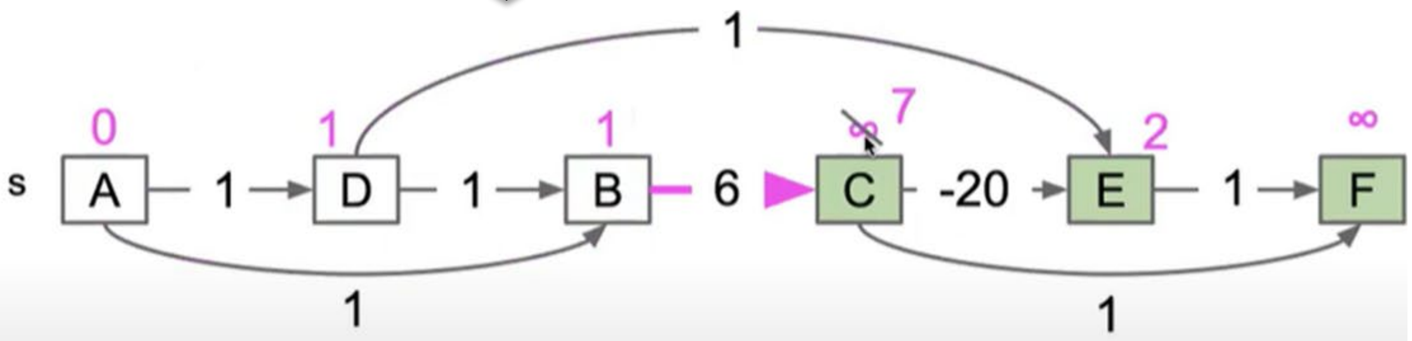
Visit D

	distTo	edgeTo
A	0	-
B	1	A
C	∞	-
D	1	A
E	2	D
F	∞	-



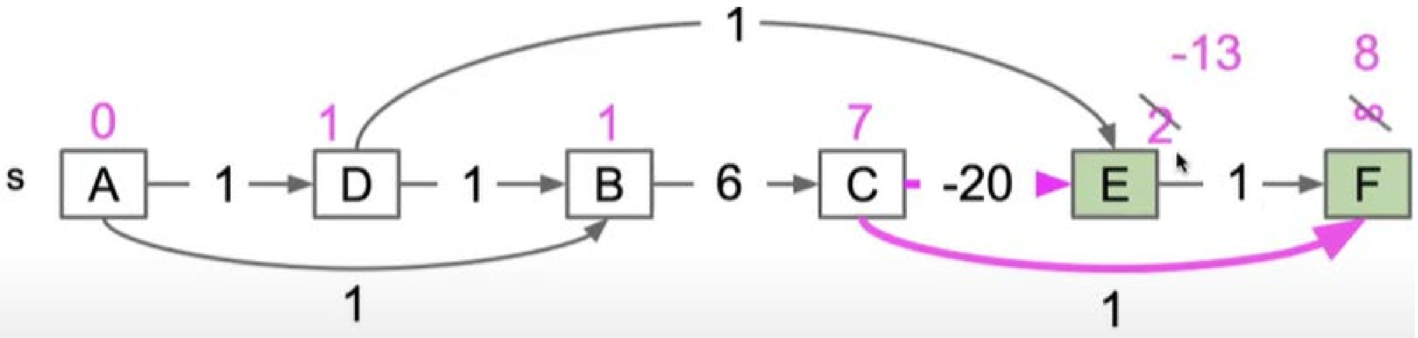
Visit B

	distTo	edgeTo
A	0	-
B	1	A
C	7	B
D	1	A
E	2	D
F	∞	-



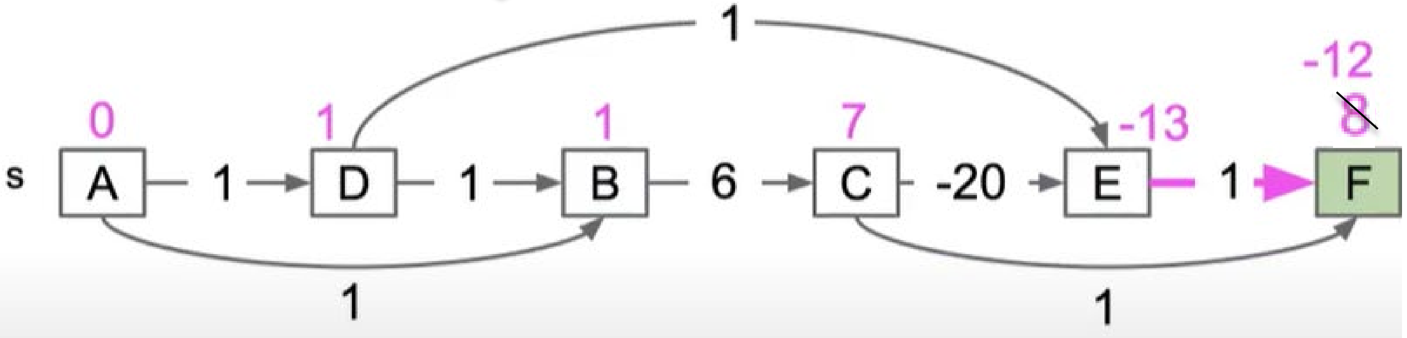
Visit C

	distTo	edgeTo
A	0	-
B	1	A
C	7	B
D	1	A
E	-13	C
F	8	C

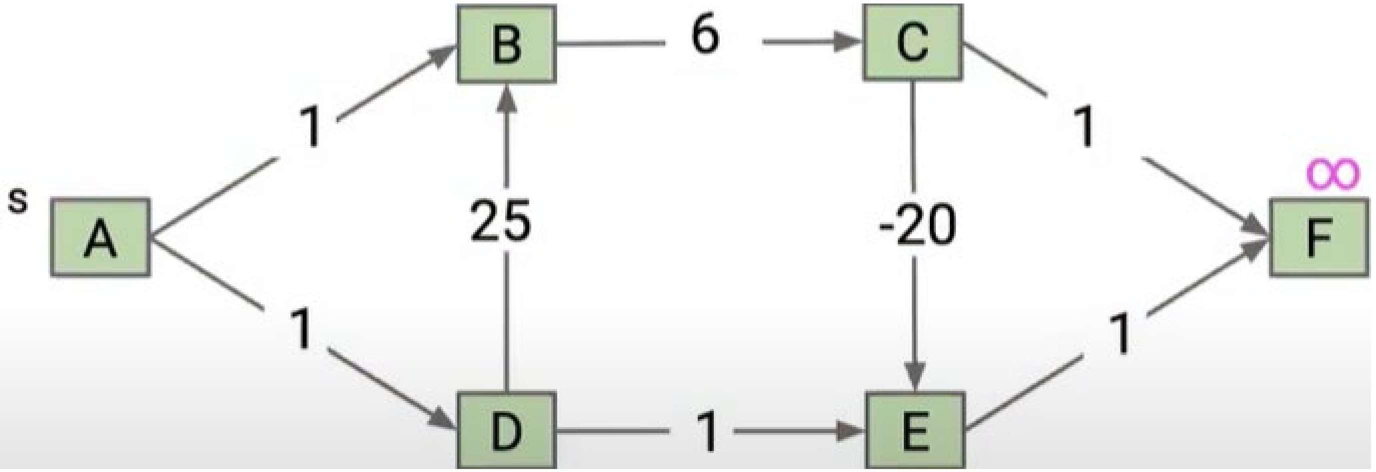


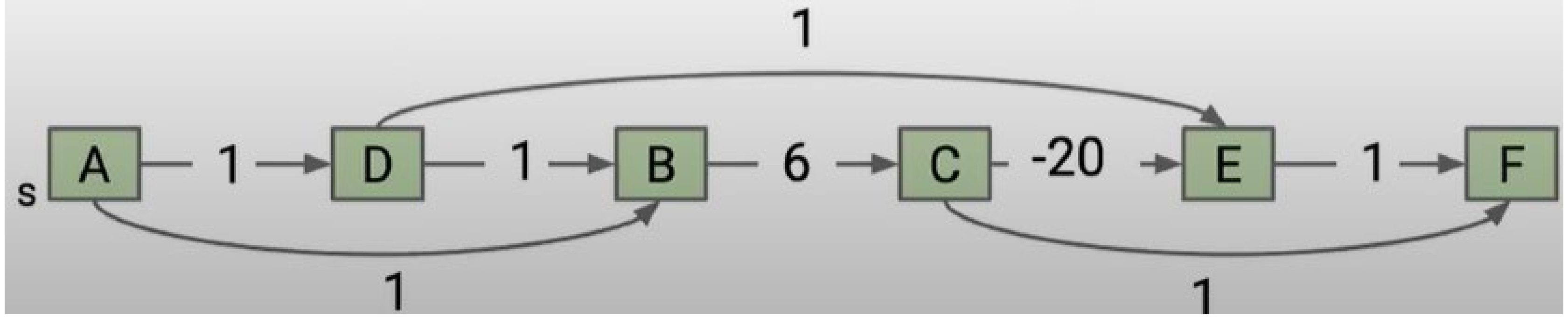
Visit E

	distTo	edgeTo
A	0	-
B	1	A
C	7	B
D	1	A
E	-13	C
F	-12	E



Topological Sort Example I: Final Answer (for Exams





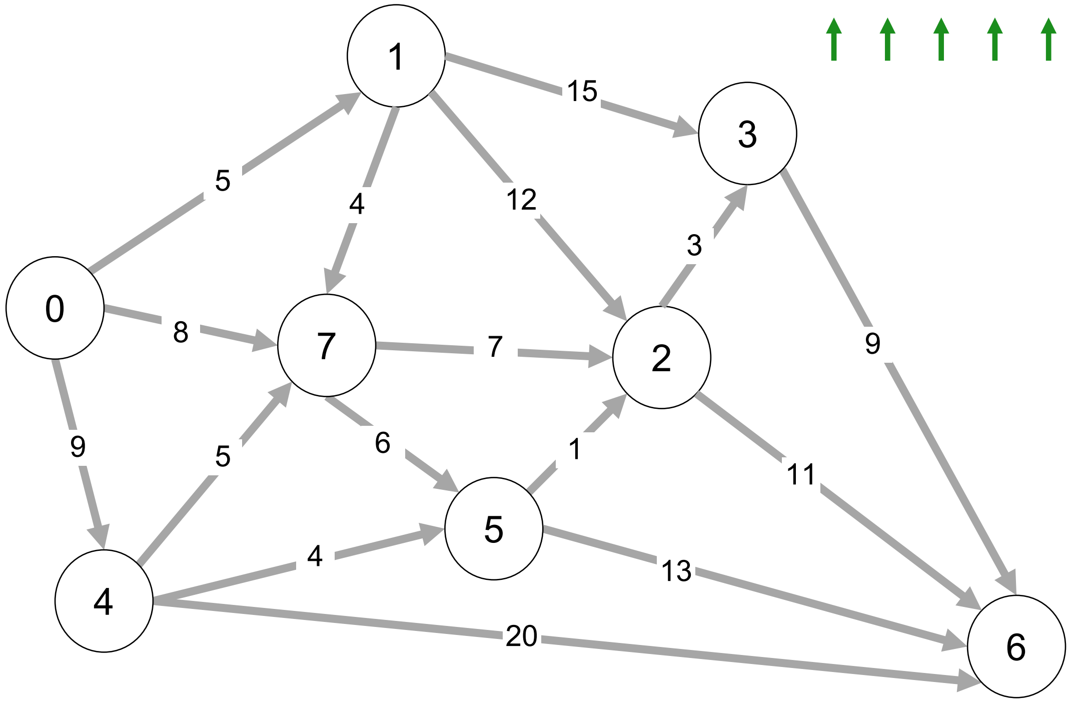
Visit Order
A, D, B, C, E, F

Node	SD	PN
A	0	/
B	1	A
C	7	B
D	1	A
E	~~2~~ - 13	~~D~~ C
F	~~8~~ - 12	~~C~~ E

Topological Sort Example II

- Consider this DAG and a topological order 0 1475236

0 1 4 7 5 2 3 6



V	SD			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	~~15~~	14
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~29~~	~~26~~	25
7	~~∞~~	8		

v	PN			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	4		
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

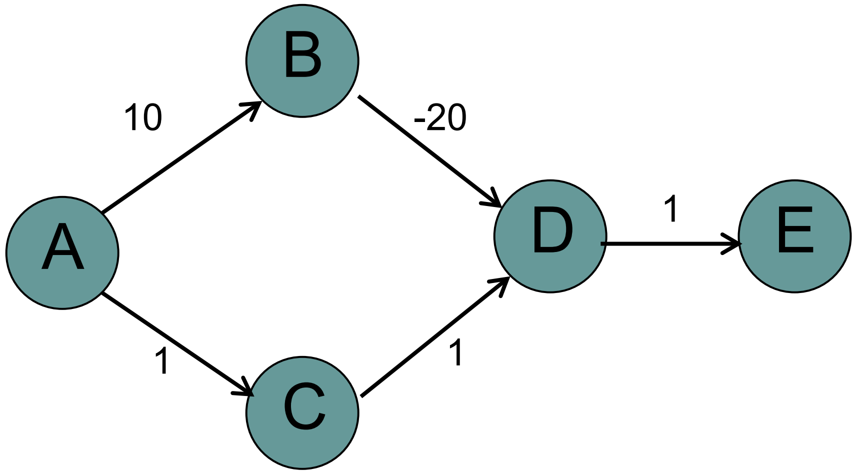
Visit Order

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

Node	SD	PN
0	0	/
1	5	0
2	~~17~~ ~~15~~ 14	~~1~~ ~~7~~ 5
3	~~20~~ 17	~~1~~ 2
4	9	0
5	13	4
6	~~29~~ ~~26~~ 25	~~4~~ ~~5~~ 2
7	8	0

Topological Sort Example III

- Consider this DAG and a topological order ABCDE



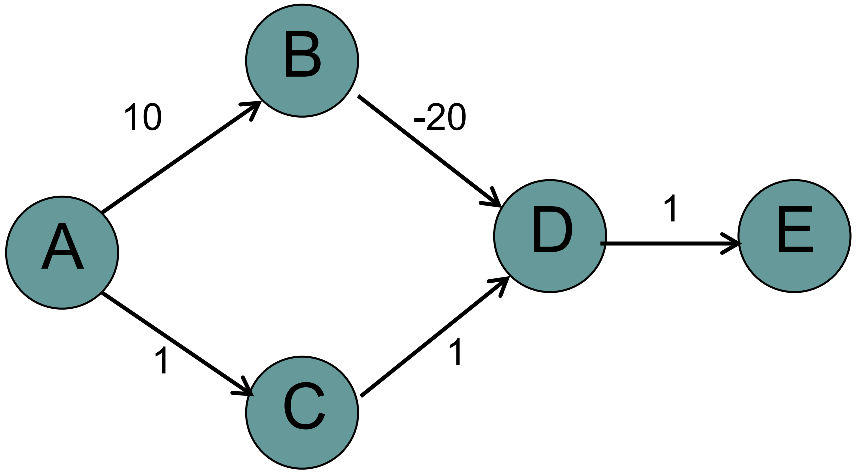
Visit Order

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

Node	SD	PN
A	0	
B	10	A
C	1	A
D	- 10	B
E	- 9	D

Topological Sort Example III

- Consider this DAG and a topological order ACBDE
  - Different visit order/topological order results in different table of SD/PN



<u>Visit Order</u>
0, 1, 4, 7, 5, 2, 3, 6

Node	SD	PN
A	0	
B	10	A
C	1	A
D	~~2~~ - 10	~~C~~ B
E	- 9	D

Single Source Shortest-paths Algorithms Summary

Algorithm	Restriction	Worst-Case Complexity
Dijkstra (Fibonacci heap)	Undirected or directed graph; no negative weights/cycles	O(V log V + E)
Topological Sort	Directed Acyclic Graph (DAG) (directed graph, no cycles)	O(E+V)

Review: Key Features

- Once a node is marked known, its shortest path is known
  - Can reconstruct path by following backpointers
- While a node is not known, another shorter path might be found!
- The order in which nodes are added to the known set is unimportant
- If we only need path to a specific node, can stop early once that node is known
  - Because its shortest path cannot change!
  - Return a partial shortest path tree

Greedy Algorithms

- At each step, do what seems best at that step
  - “instant gratification”
  - “make the locally optimal choice at each stage”
- Dijkstra’s is “greedy” because once a node is marked as “processed” we never revisit
  - This is why Dijkstra’s does not work with negative edge weights

Other examples of greedy algorithms are:

- Kruskal and Prim’s minimum spanning tree algorithms (next week)
- Huffman compression

References

- 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
- Breadth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=N6wicLpEmHY&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=5
- Depth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=5GcSvYDgiSo&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=6
- Topological Sort Visualized and Explained
  - https://www.youtube.com/watch?v=7J3GadLzydI&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=7