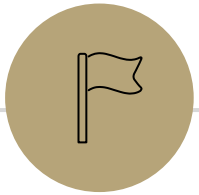
Lecture 13
Shortest Paths

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures



BFS

Dijkstra’s Algorithm

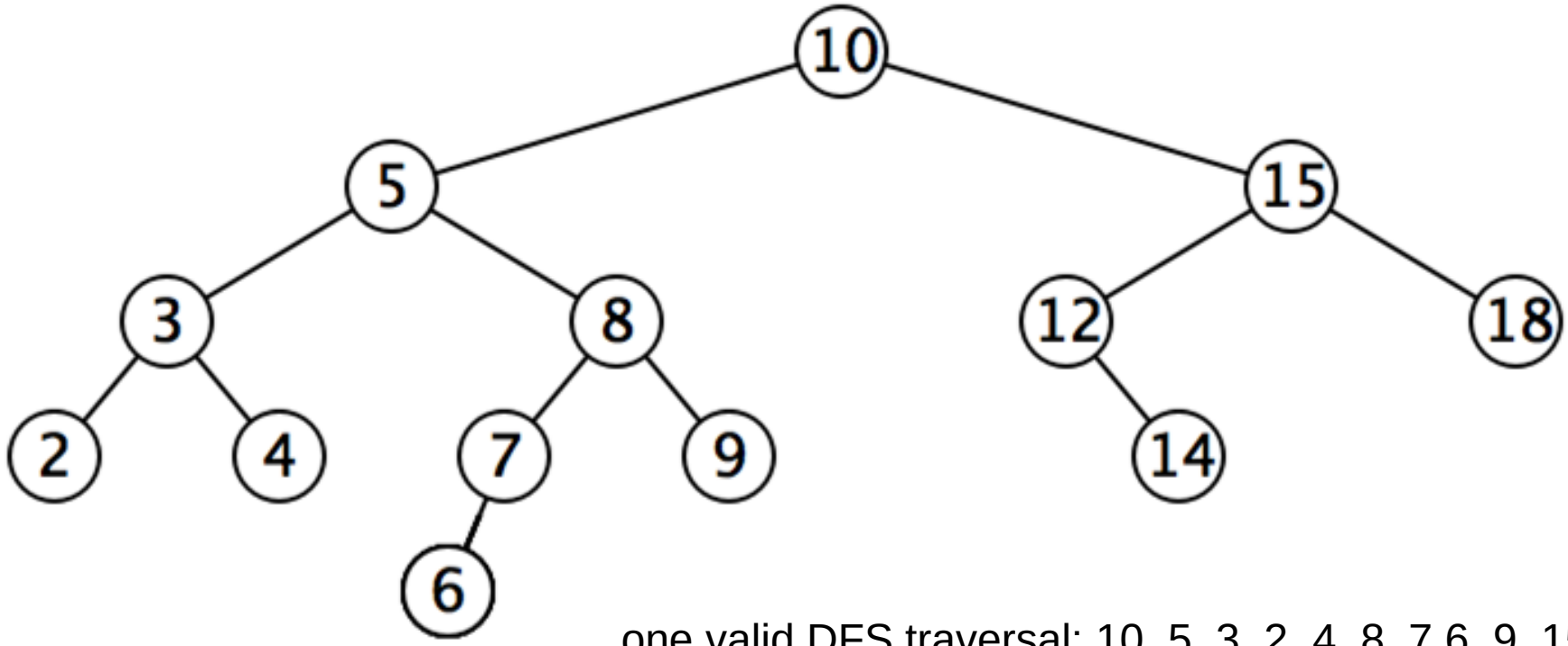
Topological Sort

Bellman-Ford Algorithm

Johnson’s Algorithm

Review: Graph traversal w/ DFS

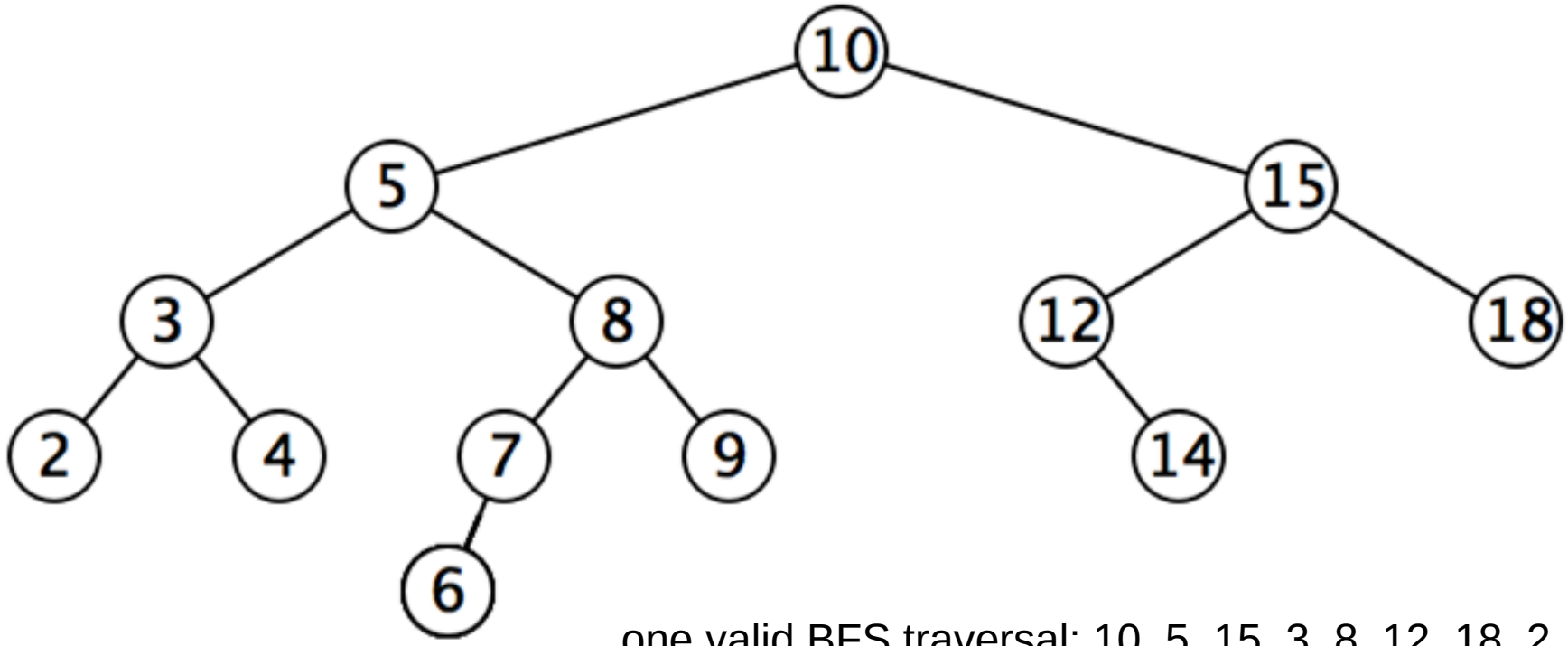
- **Depth** First Search: go as far as you can down one path till you hit a dead end (no neighbors, or no unvisited neighbors). Once you hit a dead end, backtrack and try other edges that you have not tried yet
- Analogy of wandering a maze – if you get stuck at a dead end, trace your steps backwards to the previous fork in the road, and try a different path



one valid DFS traversal: 10, 5, 3, 2, 4, 8, 7,6, 9, 15, 12, 14, 18

Review: Graph traversal w/ BFS

- **Breadth** First Search - traverse level by level, and visit 1-hop neighbors before 2-hop neighbors before 3-hop neighbors…
- Analogy: sound wave spreading from a starting point, going outwards in all directions; mold on a piece of food spreading outwards so that it eventually covers the whole surface



one valid BFS traversal: 10, 5, 15, 3, 8, 12, 18, 2, 4, 7, 9, 14, 6

BFS for (Unweighted) Shortest Path Problem

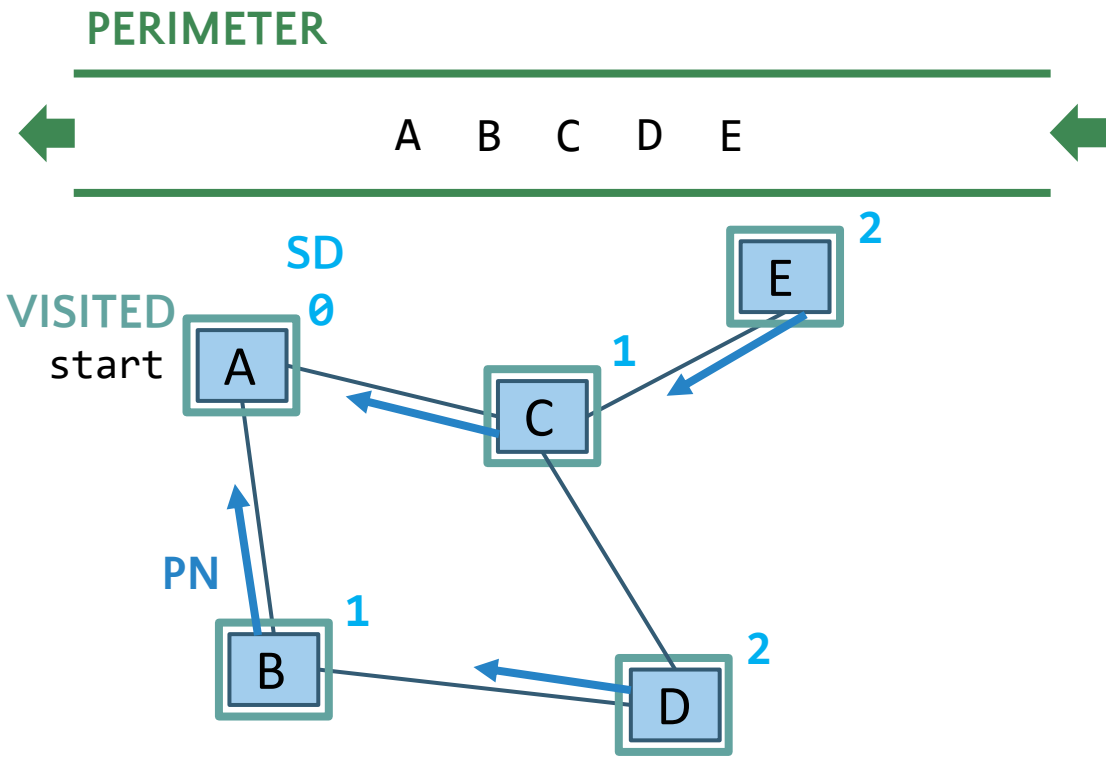
- BFS can find shortest paths in an unweighted graph
  - BFS visits nodes in order of their distance from the source node, ensuring the first path found to any node is the shortest possible path in terms of the number of edges
  - Time complexity: O(V+E)
- Advantages:
  - Optimal for unweighted graphs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

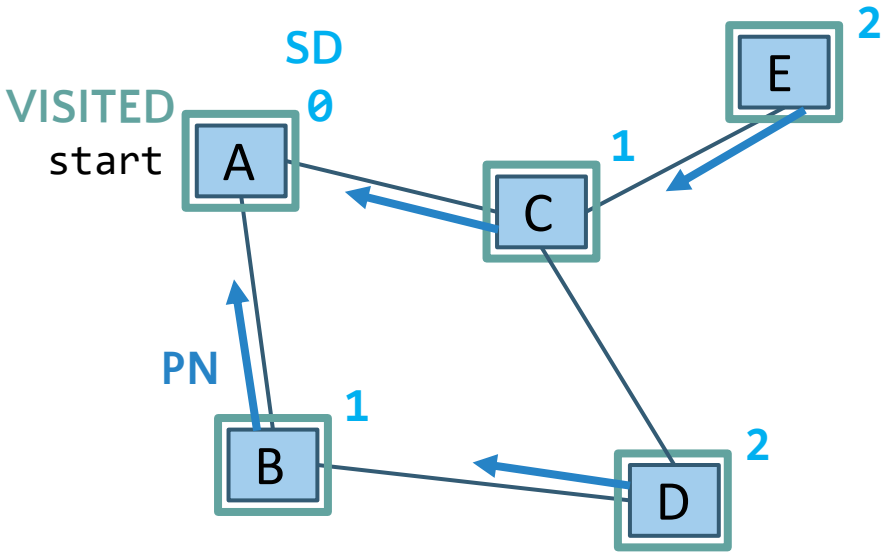



```
...
Map<Node, Edge> PN = ...
Map<Node, Double> SD = ...

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 encodes the Shortest Path Tree (SPT), which encodes the shortest path and distance from the start node to *every other node*

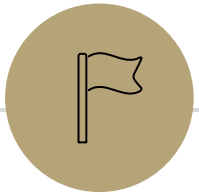
Shortest path to any node can be obtained from SPT

- **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

BFS Time Complexity

- Using Adjacency List: O(V + E)
  - Each node is processed exactly once: O(V)
  - Each edge is examined exactly once: O(E)
  - Total complexity: O(V + E)
  - Efficient for sparse graphs (where E is much less than V^2)
- Using Adjacency Matrix: $O(V^2)$
  - For each node, we must check all possible edges to other vertices
  - This results in $O(V^2)$ operations regardless of the actual number of edges



BFS

Dijkstra’s Algorithm

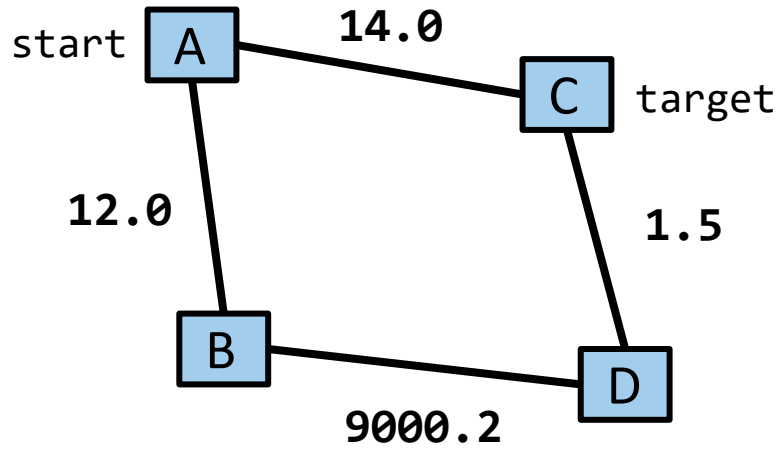
Topological Sort

Bellman-Ford Algorithm

Johnson’s Algorithm

Dijkstra's Algorithm

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

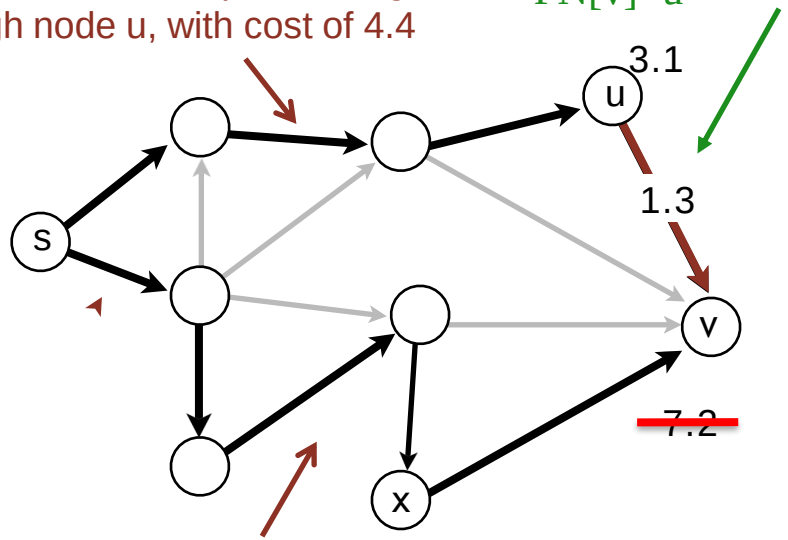


Edge Relaxation

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



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.

Proof.

- 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.

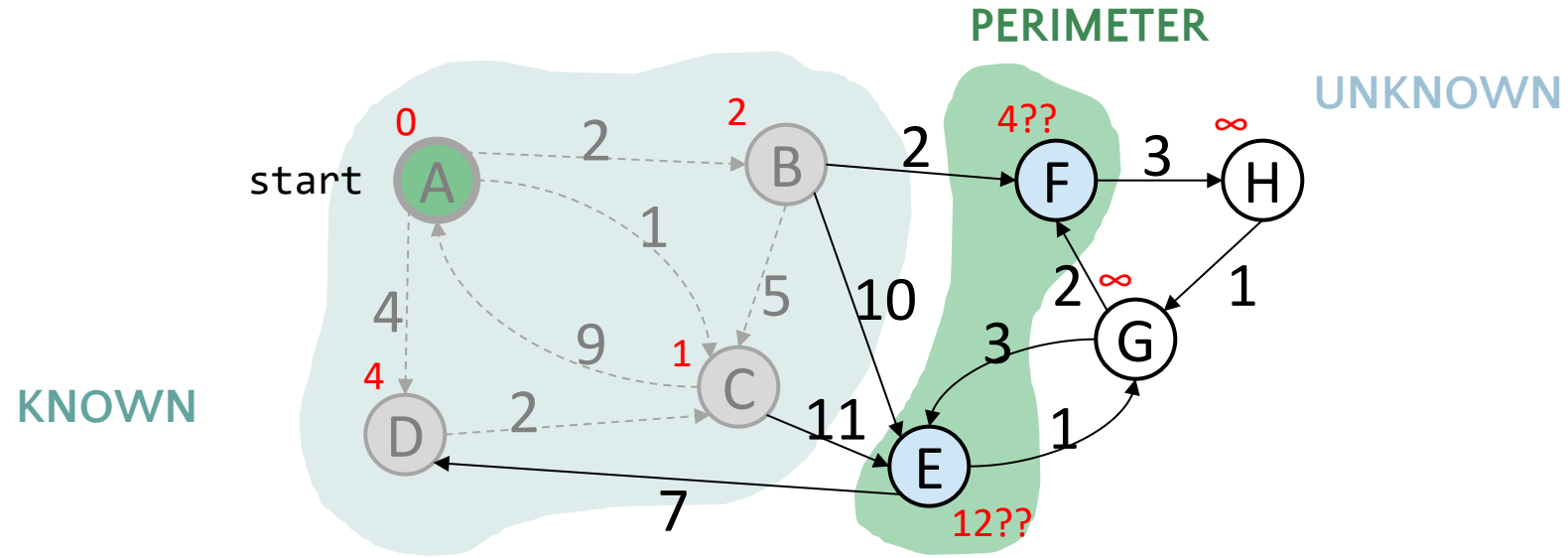
Efficient implementations. How to choose which edge to relax?

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

Dijkstra's Algorithm

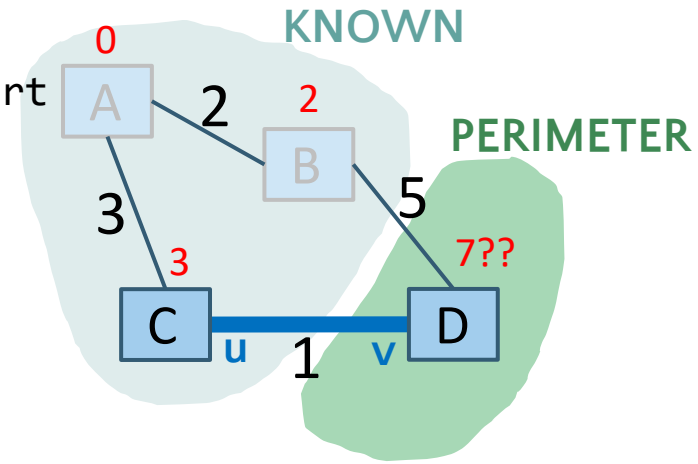
- Initialization:
  - Set the distance to the source node as 0 and to all other nodes as infinity.
  - Mark all nodes as unvisited and store them in a priority queue.
- Main Loop:
  - Visit the unvisited node u with the shortest known distance (minimum SD) from the queue.
  - For each unvisited neighbor 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 the closest unknown node v (with smallest SD)
  - 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`
  - Relaxation updates 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 visited (marked known), its shortest path is known. Can reconstruct path by following back-pointers (in PN map)

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

If we only need path to a specific node, can stop early once that node is visited, and return a partial shortest path tree

```
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

```
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
```

O(V)

O(V)

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

O(E)

O(log V)

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

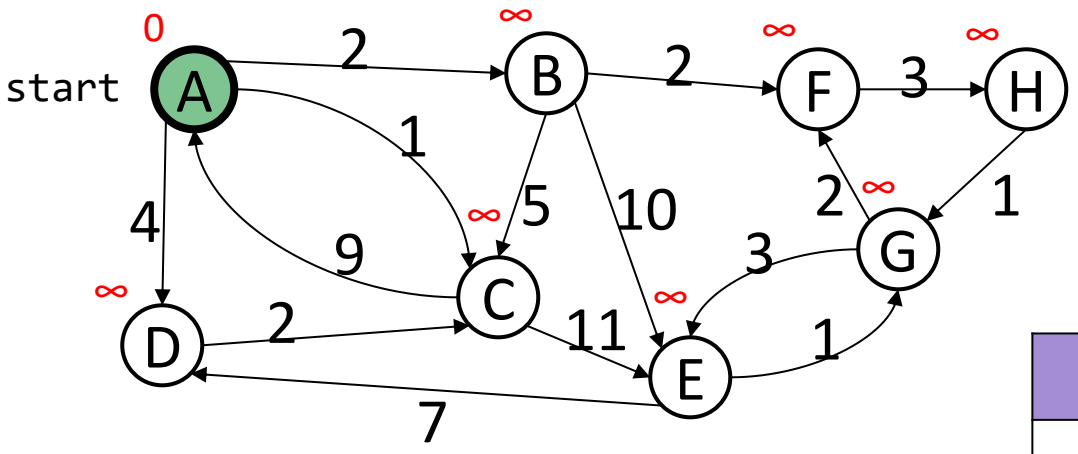
Greedy Algorithms

- A greedy algorithm makes the locally optimal choice at each step
- Dijkstra's is "greedy" because once a node is marked as visited, it is never revisited
  - This is why Dijkstra's does not work with negative edge weights
- In the lecture and exams, when there are multiple possible orders of visiting the next node (with equal SD value), select the next node in alphabetical or numerical order
  - The intermediate steps will depend on the order, but final result will be the same
- Other examples of greedy algorithms are:
  - Kruskal and Prim's minimum spanning tree algorithms

Resolving Ambiguities

- As There are typically multiple possible orders of the same graph. In the lecture and exams, we often use the following rule to resolve any ambiguities:
- “When there are multiple possible orders of visiting the next node, select the next node in alphabetical or numerical order.”

Example I

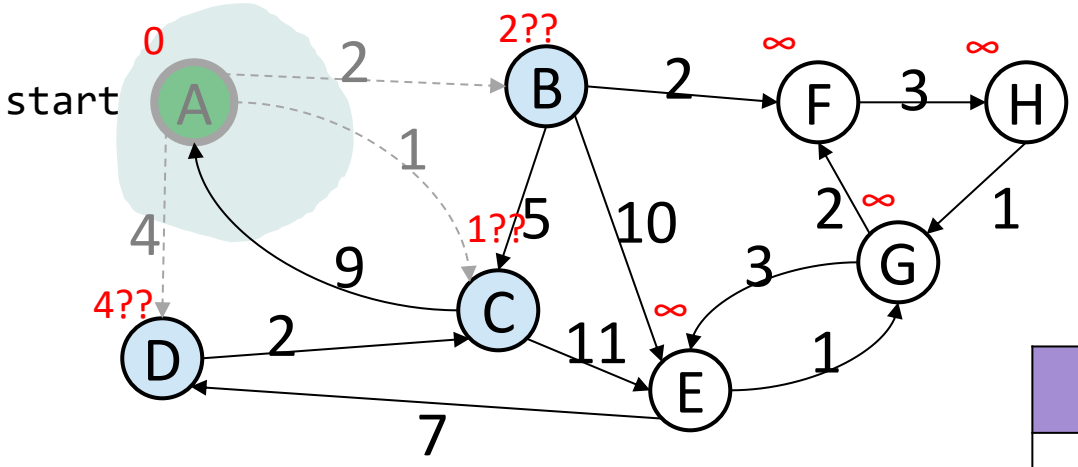


Node	SD	PN
A	0	/
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Visit Order

Start from the source node A

Example I



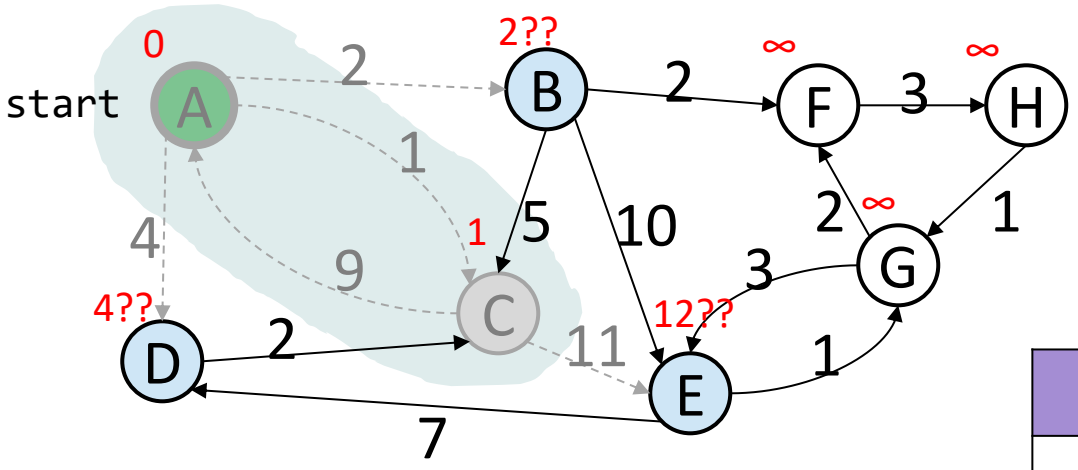
?? Means that SDs have not yet been finalized, as a shortcut may be found in the future.

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

Visit Order
A

Visit C next, since C has the smallest SD of 1 among all unknown (unvisited) nodes

Example I

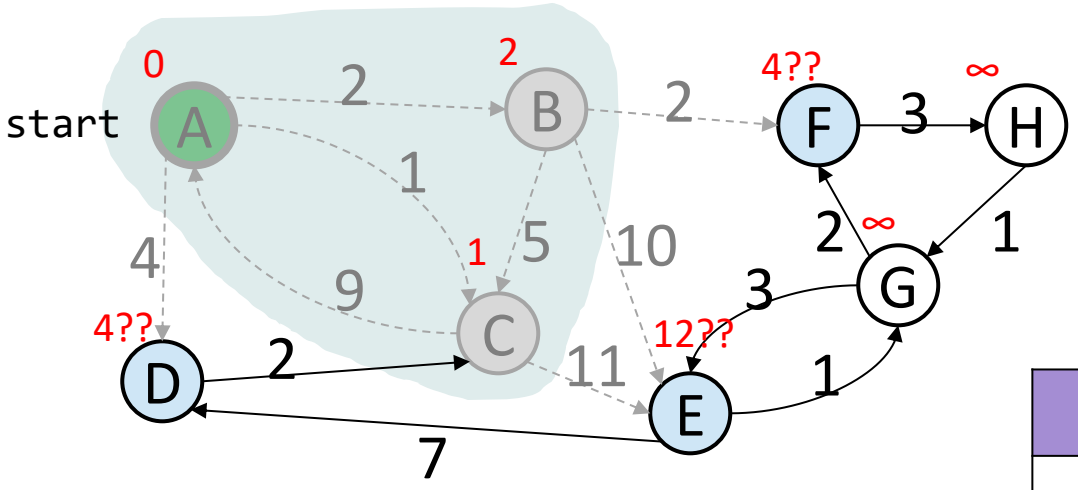


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	∞	
G	∞	
H	∞	

Visit Order
A, C

Visit B next, since B has the smallest SD of 2 among all unvisited nodes

Example I

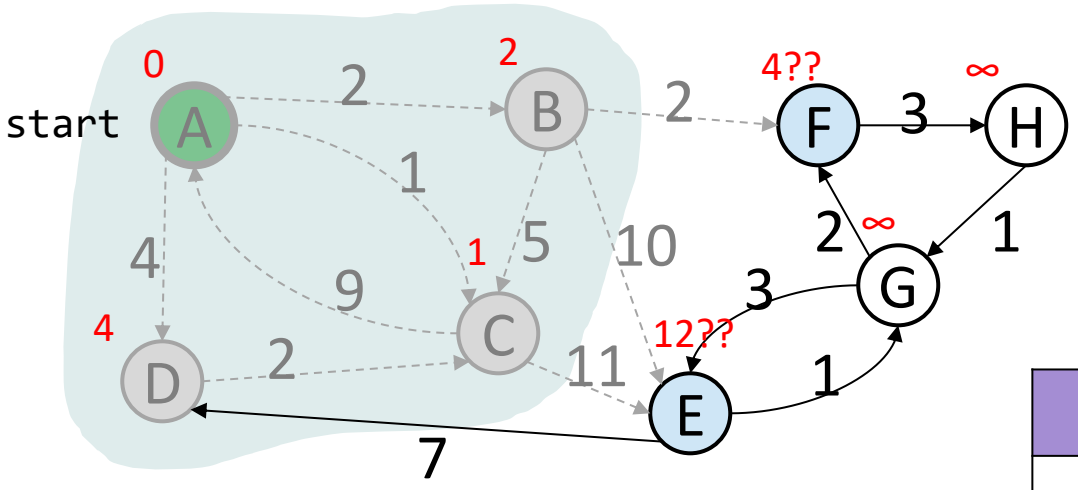


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	**4**	**B**
G	∞	
H	∞	

Visit Order
A, C, B

We can choose to visit either D or F next, since they have equal smallest SD of 4 among all unvisited nodes. Let's visit D in alphabetical order

Example I

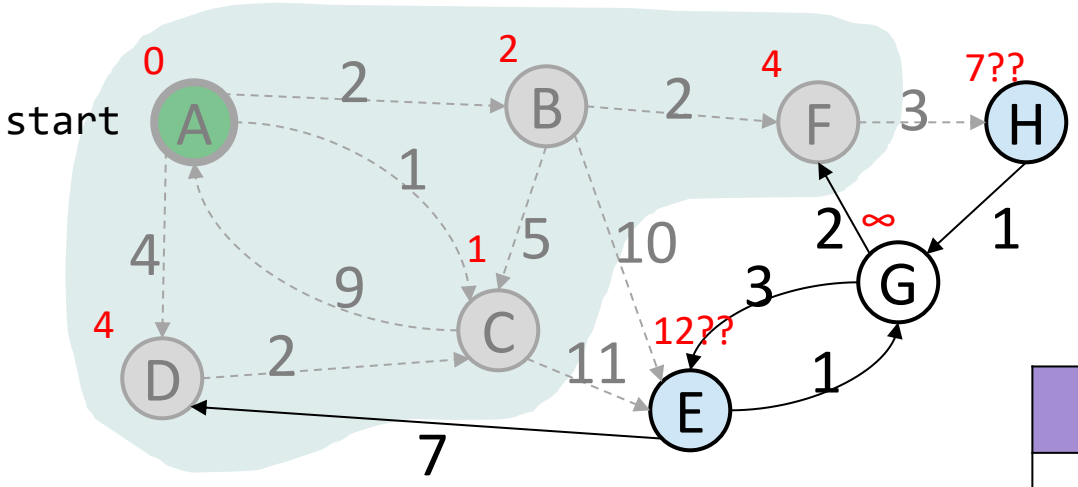


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	∞	
H	∞	

Visit Order
A, C, B, D

Visit F next, since F has the smallest SD of 4 among all unvisited nodes

Example I

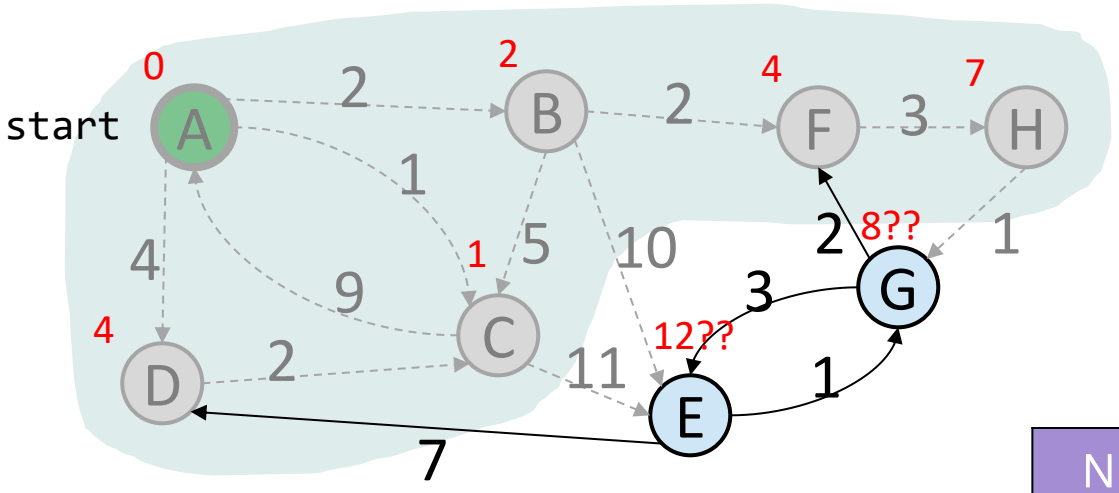


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	∞	
H	**7**	**F**

Visit Order
A, C, B, D, F

Visit H next, since H has the smallest SD of 7 among all unvisited nodes

Example I

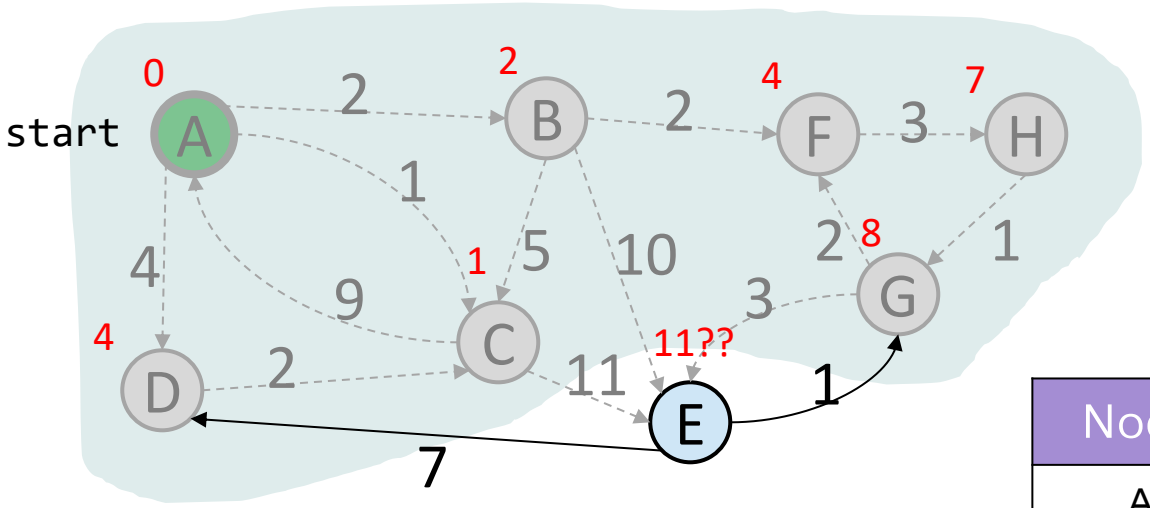


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	12	C
F	4	B
G	**8**	**H**
H	7	F

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

Visit G next, since G has the smallest SD of 8 among all unvisited nodes

Example I

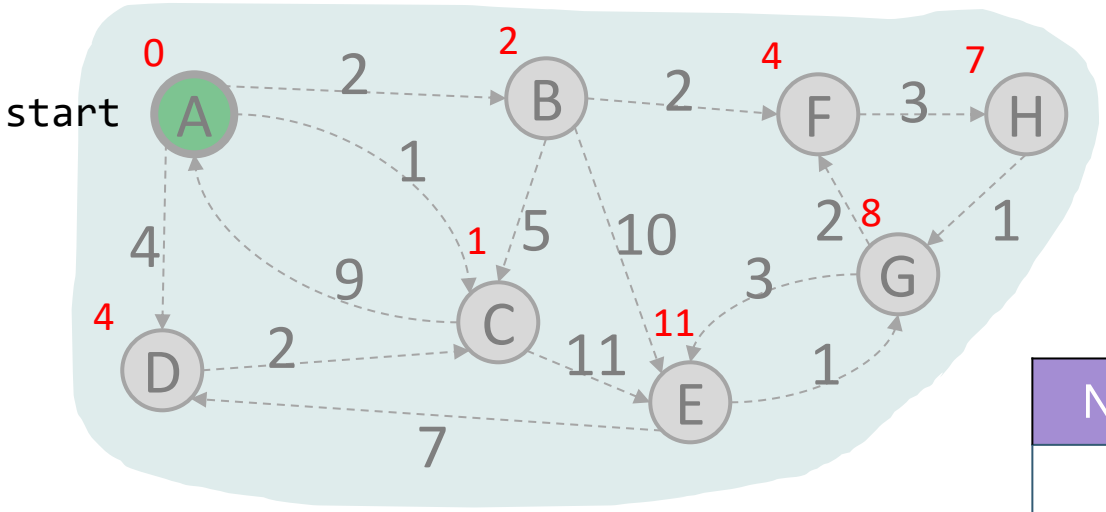


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	~~12~~ 11	~~C~~ G
F	4	B
G	8	H
H	7	F

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

We found a shortcut to E going through G, so we update SD and PN for E. Visit E next, since it is the last unvisited node

Example I Final

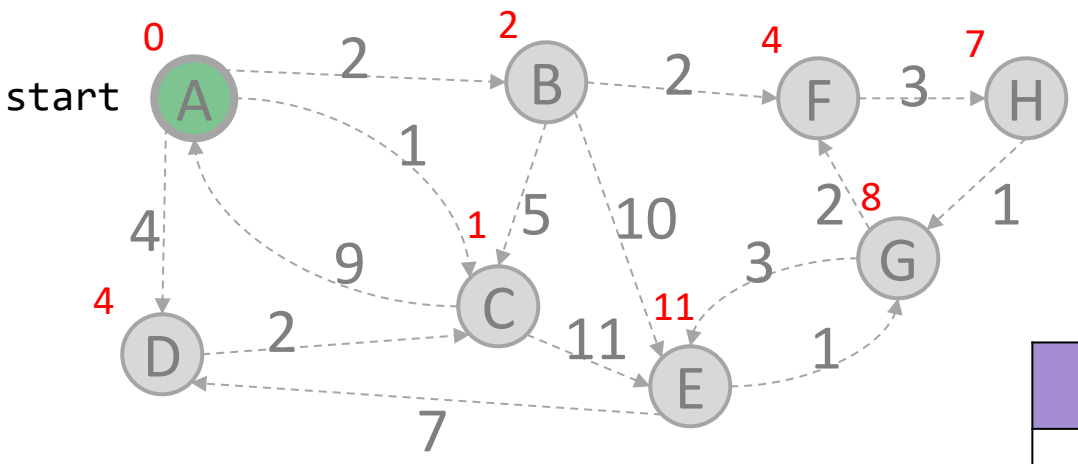


Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	~~12~~ 11	~~C~~ G
F	4	B
G	~~7~~ 8	~~8~~ H
H	7	F

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

All nodes have now been visited and are known

Example I: Interpreting the Results



How to get the shortest path from A to E?

- Follow PN backpointers to get path ABFHGE

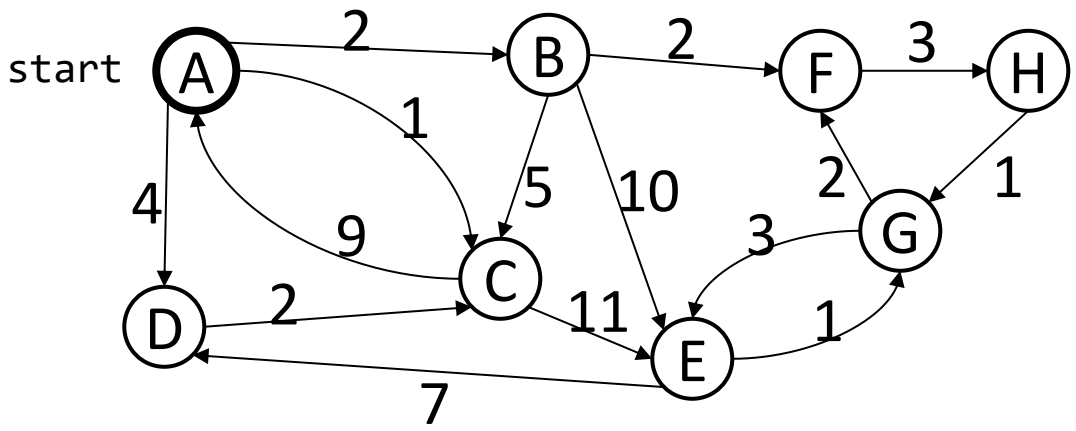
Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	11	G
F	4	B
G	8	H
H	7	F

Visit Order

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

Example I Exam Question and Answer

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 SD and PN as you find a shortcut path with smaller SD



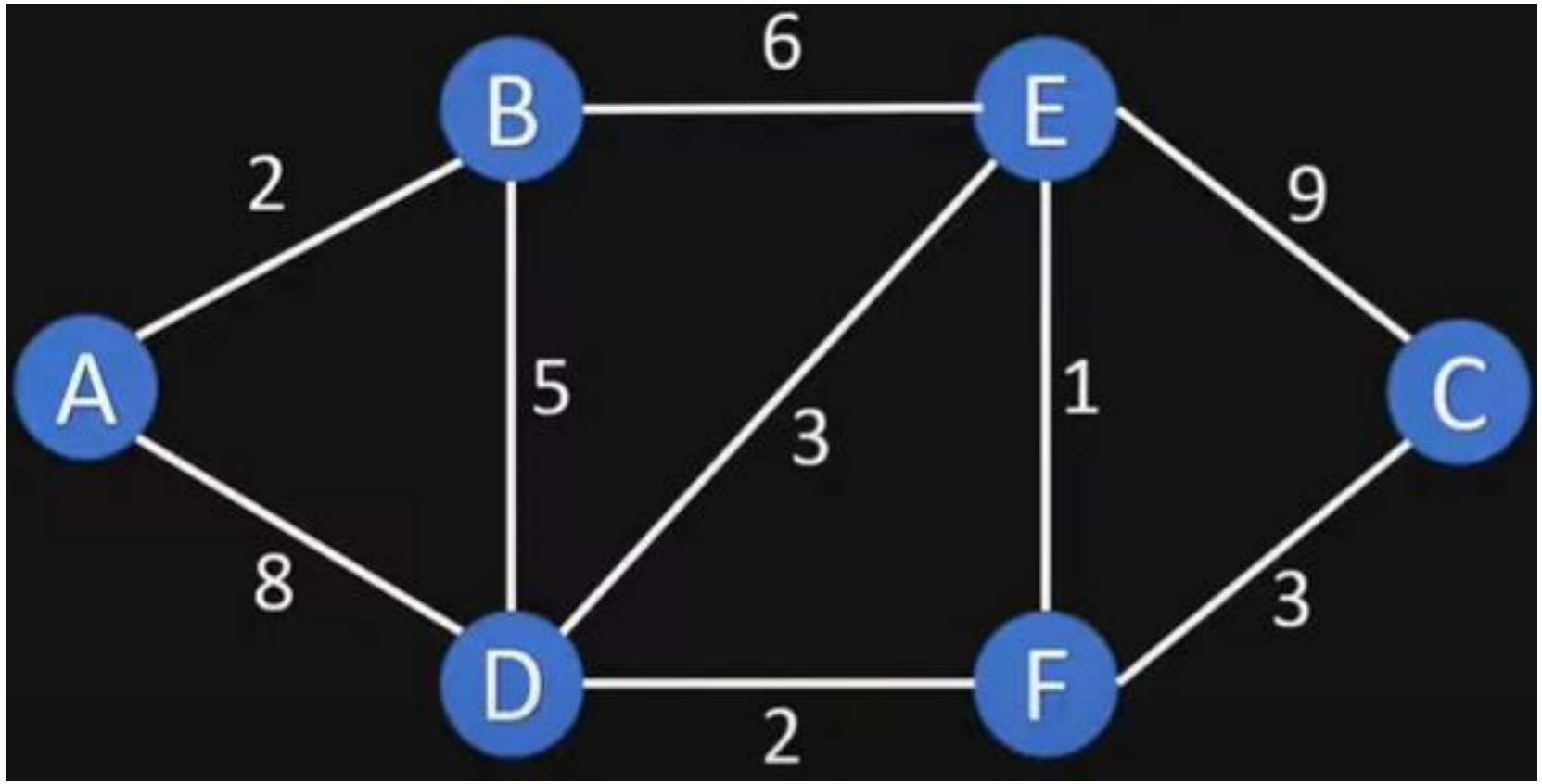
Visit Order

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

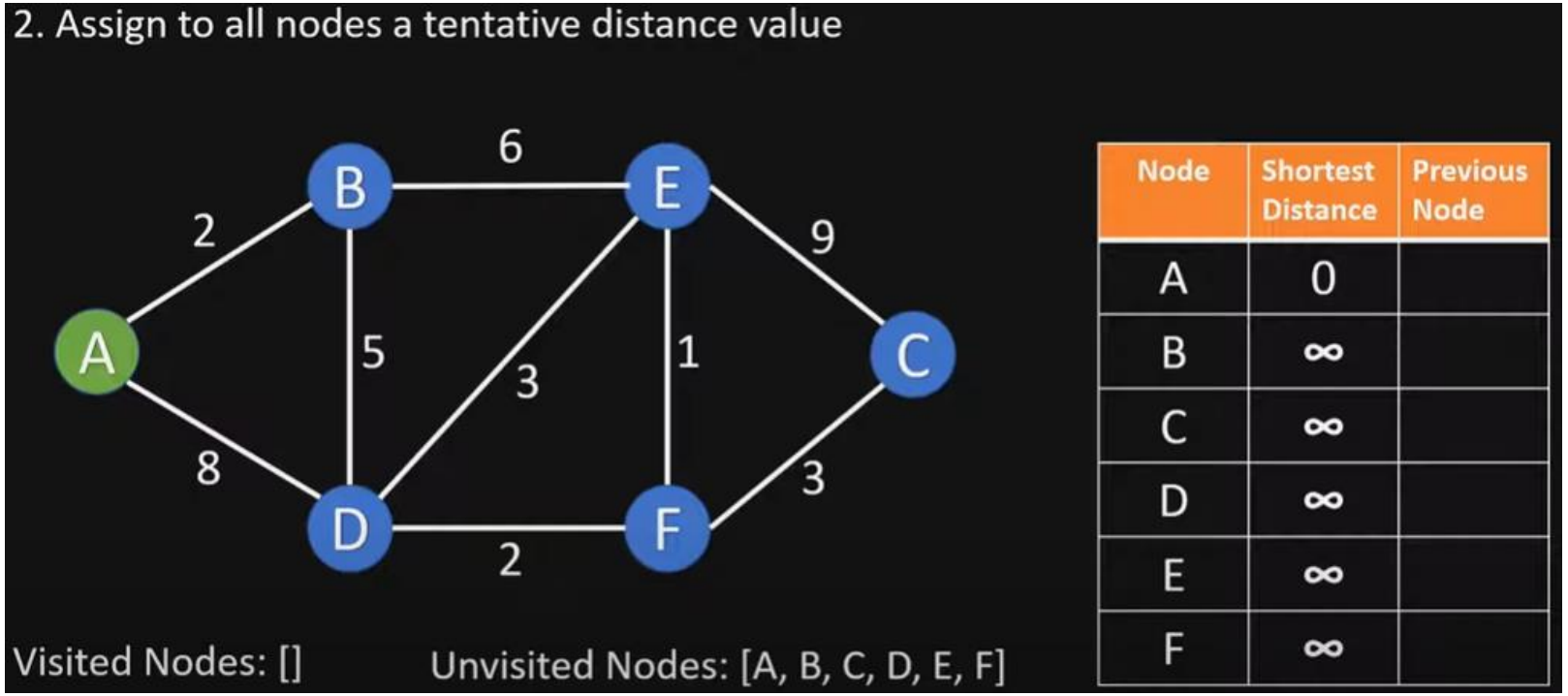
Node	SD	PN
A	0	/
B	2	A
C	1	A
D	4	A
E	~~12~~ 11	~~C~~ G
F	4	B
G	~~7~~ 8	~~8~~ H
H	7	F

Example II

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



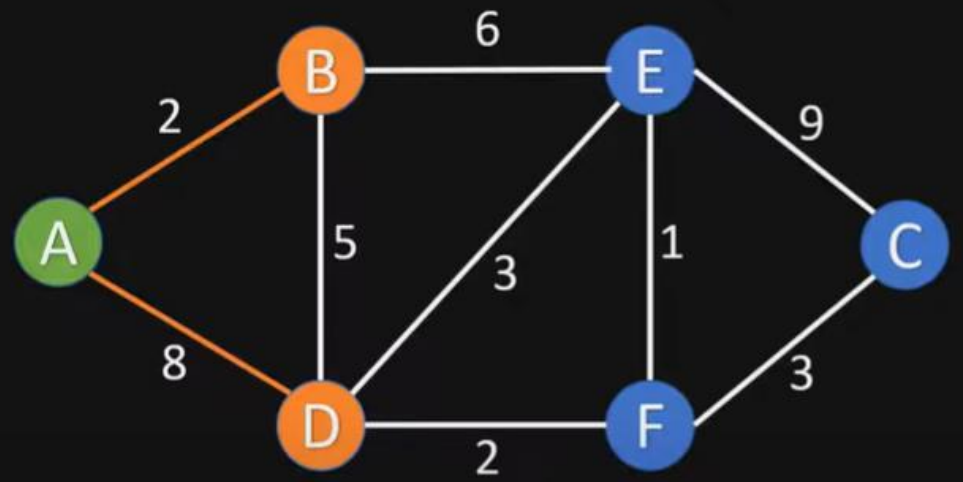
Initialize



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

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



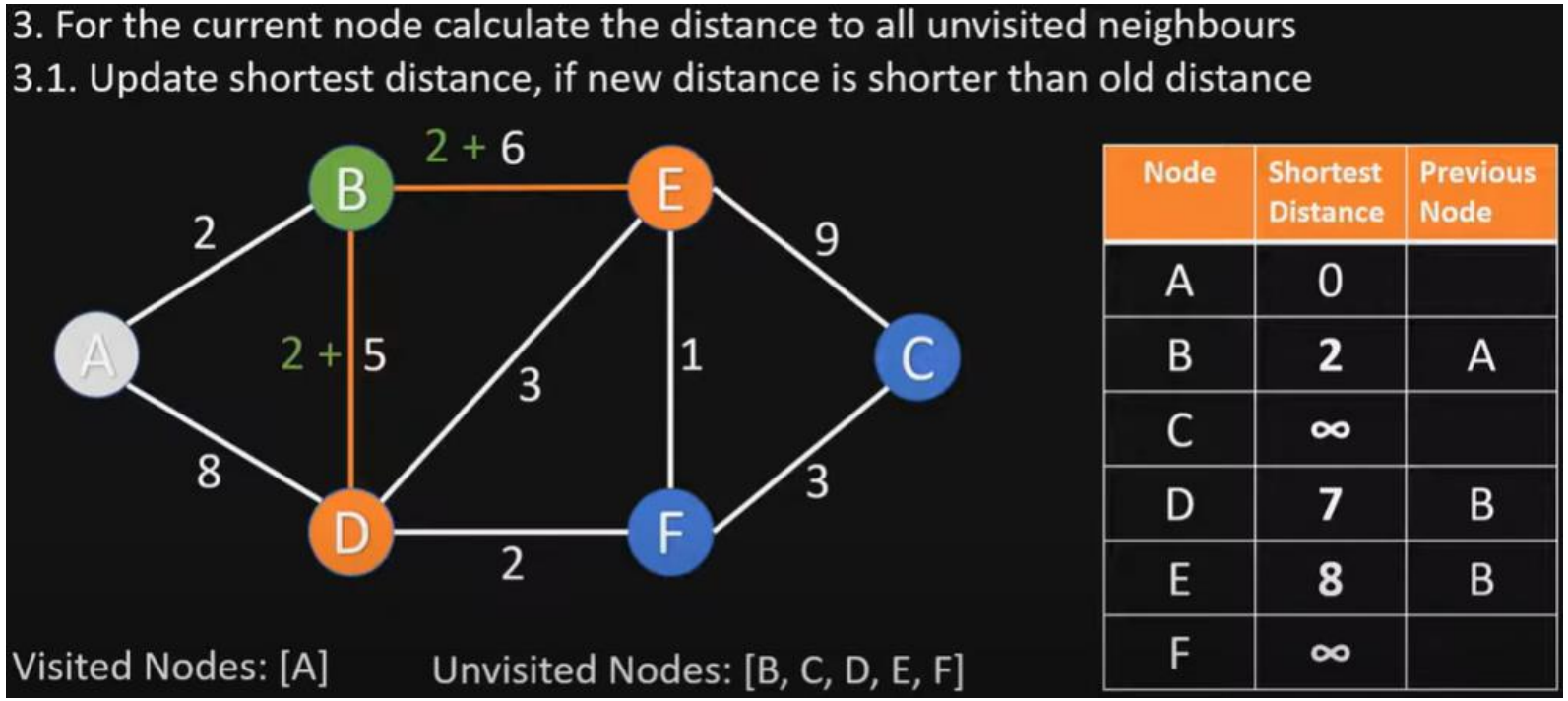
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: []

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

OLD 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

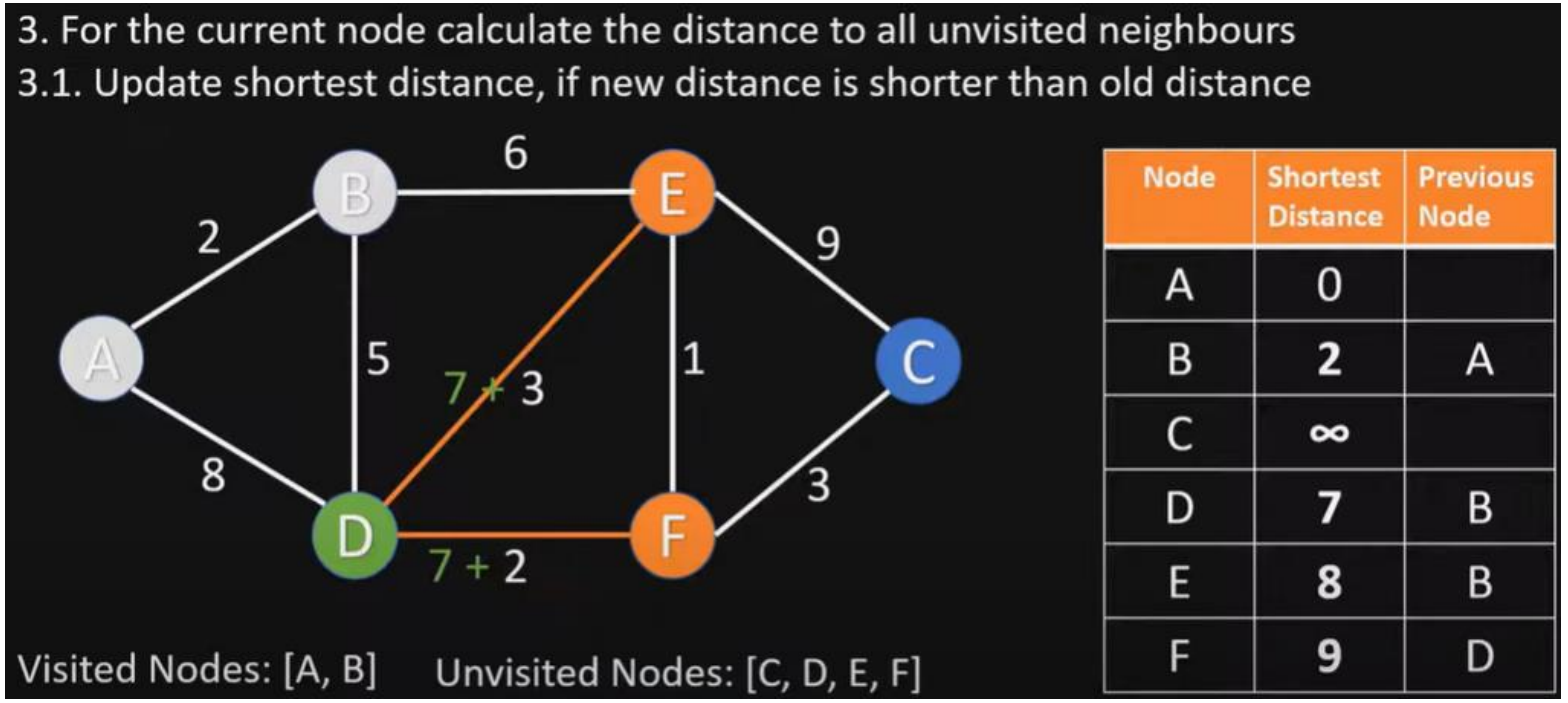
Visit node B



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

OLD SD[D] = 8 > SD[B] + w(B,D) = 2+5 = 7
NEW SD[D] ← SD[B] + w(B,D) = 7, PN[D] = B
OLD SD[E] = ∞ > SD[B] + w(B,E) = 2+6 = 8
NEW SD[E] ← SD[B] + w(B,E) = 8, PN[E] = B

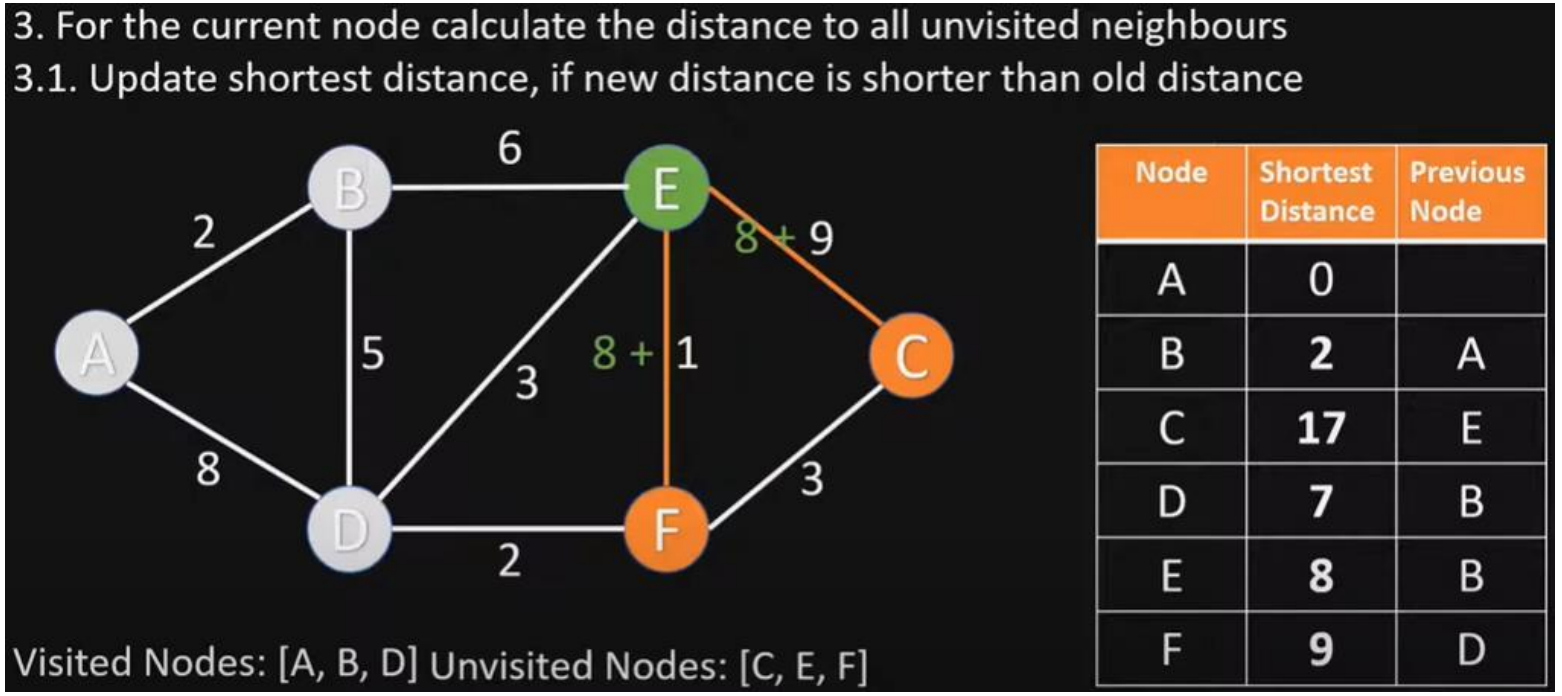
Visit node D



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

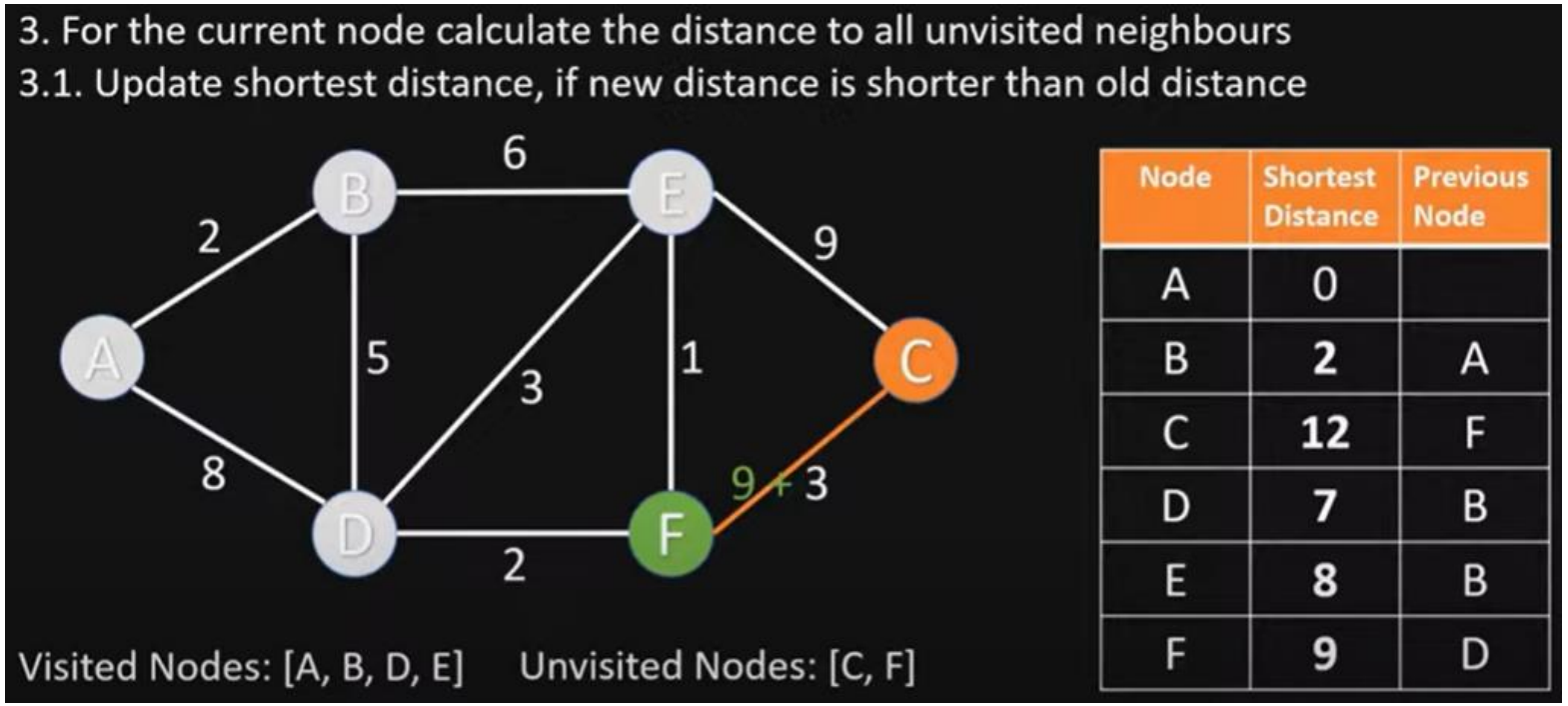
Visit node E



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

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.)

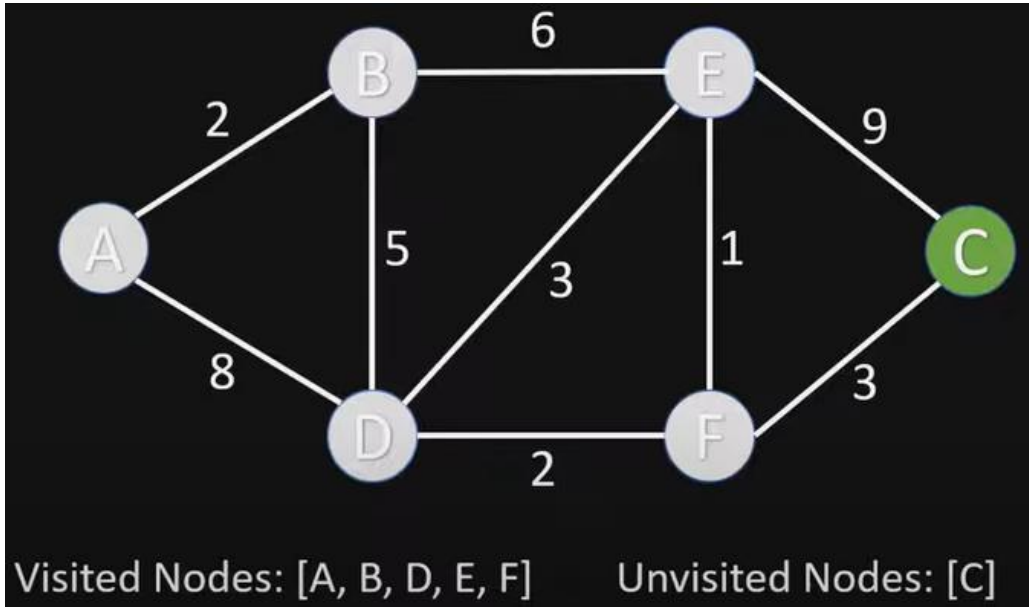
Visit node F



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

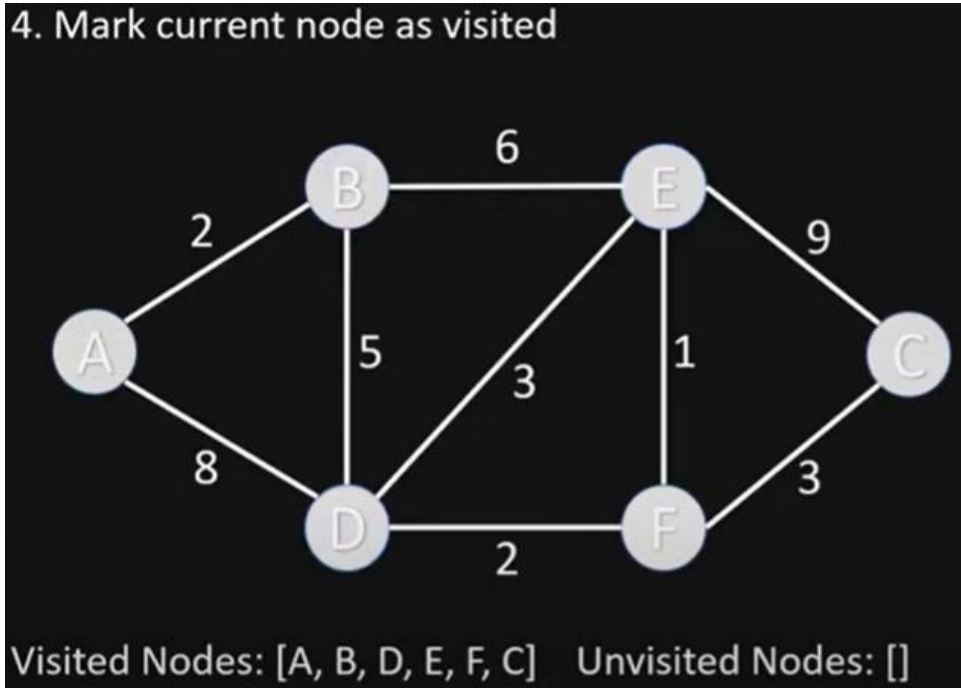


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Nothing changes, since C has no unvisited neighbor nodes

End of Algorithm

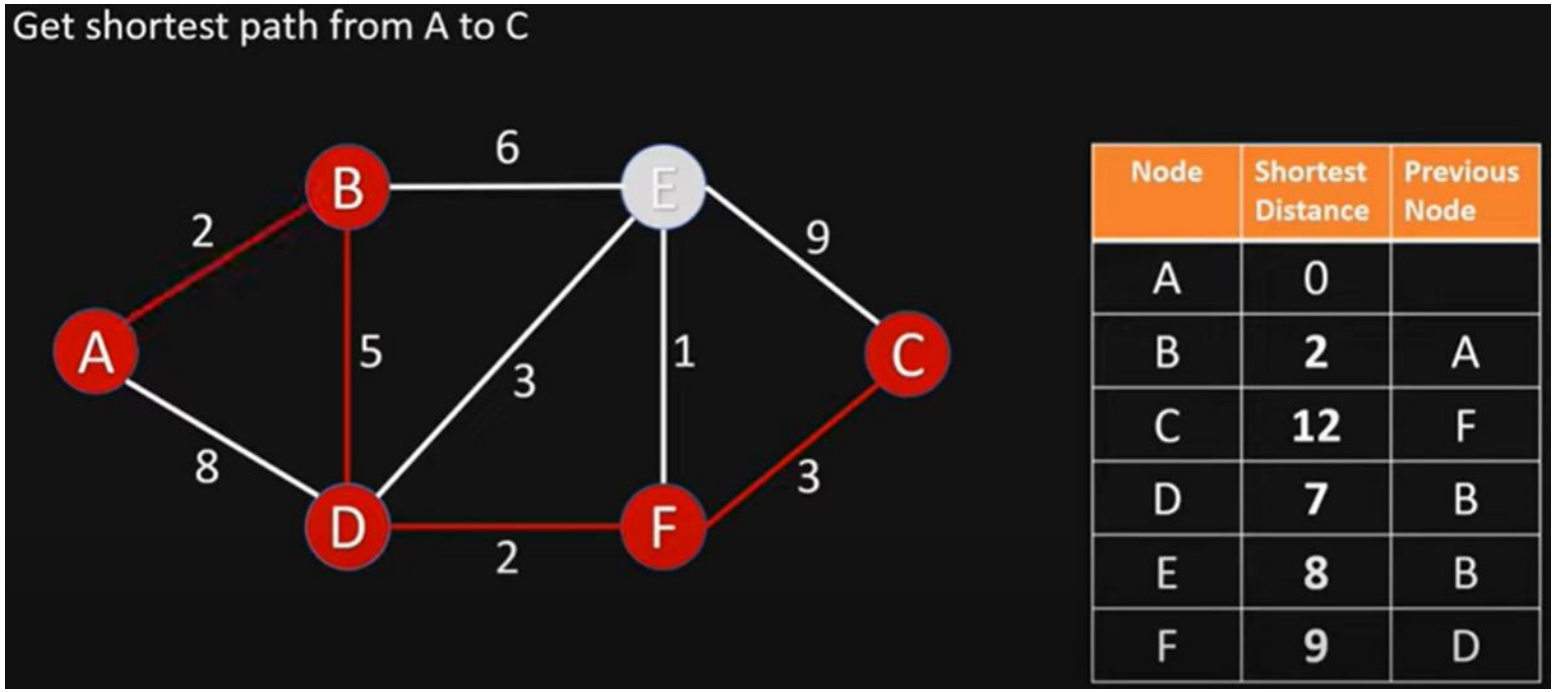
- The table now contains the SD (shortest distance) to each node N from the source node A, and its PN (previous node) in the shortest path



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

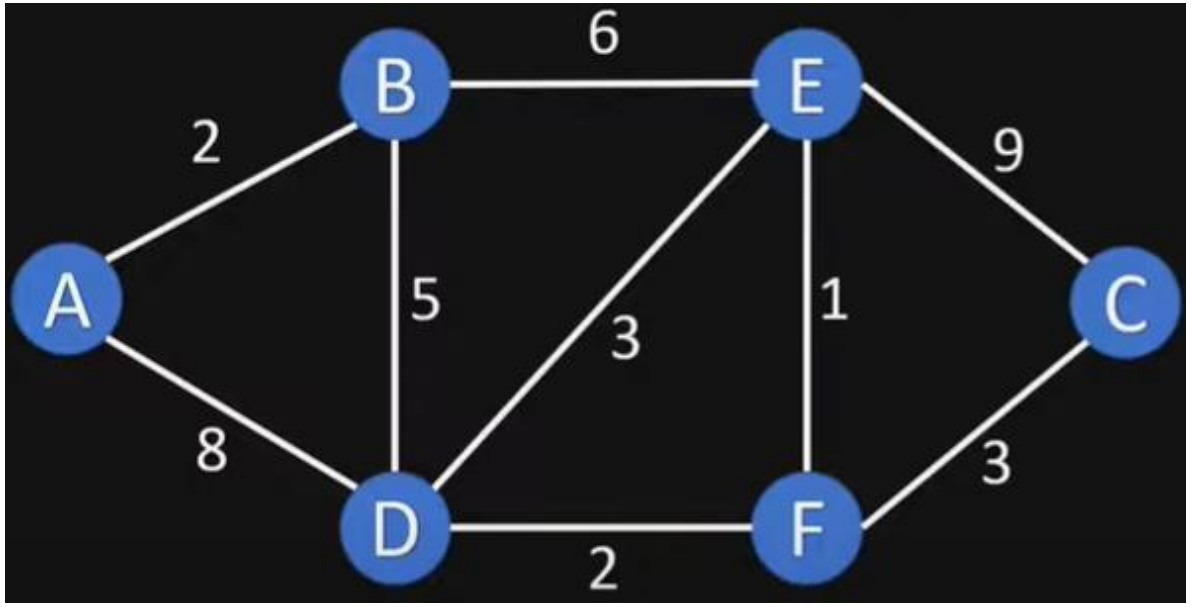
- C’s previous node is F; F’s previous node is D; D’s previous node is B; B’s previous node is A
- Shortest Path from A to C is ABDFC



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

Example II Exam Question and Answer

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 SD and PN as you find a shortcut path with smaller SD

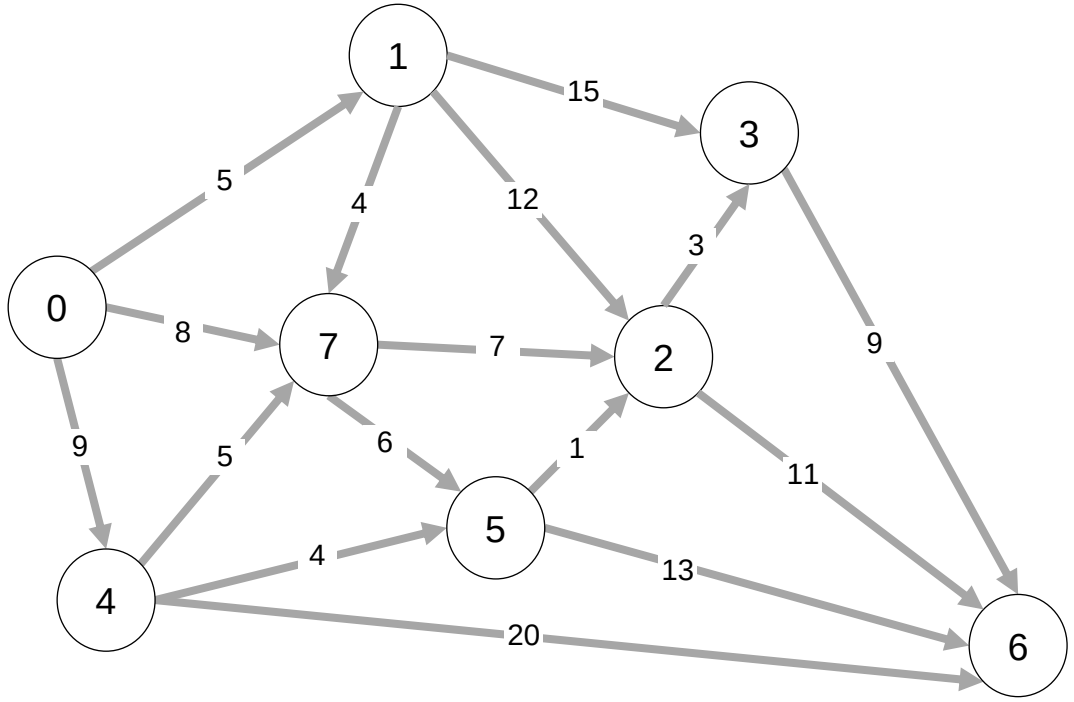


Visit Order

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 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

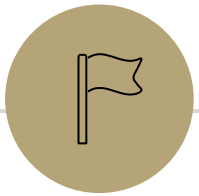
choose node 5
relax all edges adjacent from 5
choose node 2
relax all edges adjacent from 2
choose node 3
relax all edges adjacent from 3
choose node 6
relax all edges adjacent from 6

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



BFS

Dijkstra’s Algorithm

Topological Sort

Bellman-Ford Algorithm

Johnson’s Algorithm

Topological Sort for Shortest Paths in 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
- Time Complexity: 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-based SPT for a DAG

For each node v: SD [v] = ∞.

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

SD [s] = 0.

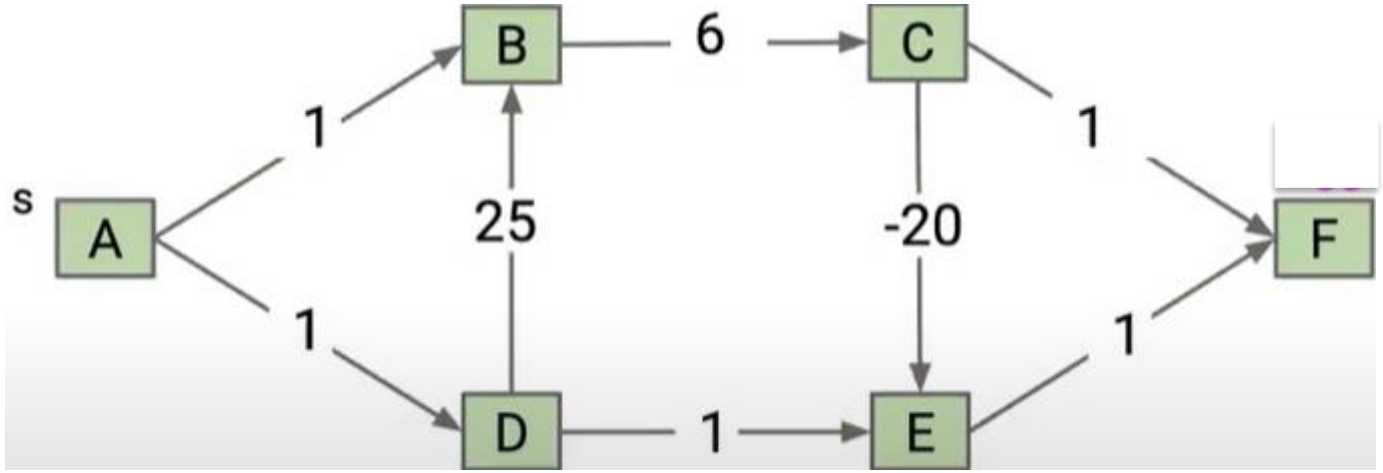
Create a topological order of the DAG

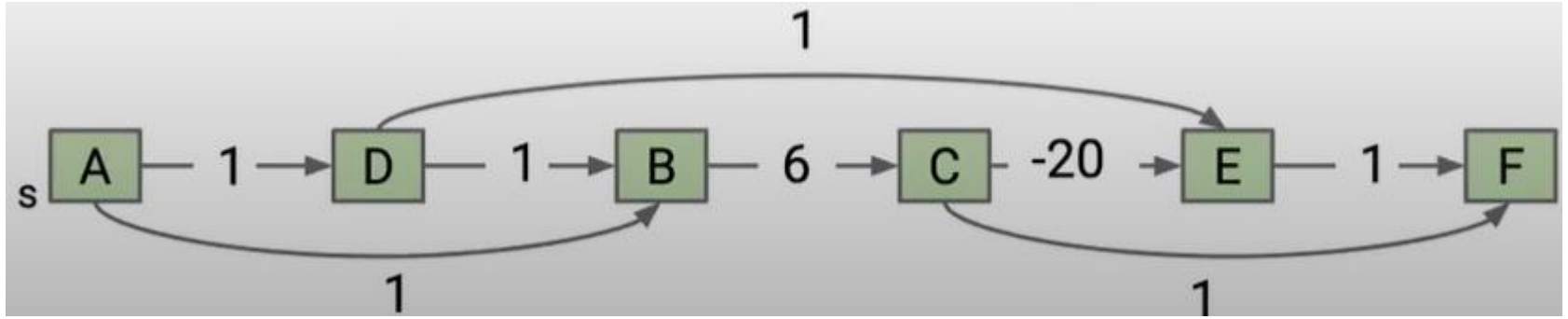
For every node u in topological order:

- Relax all its outgoing edges

Topological Sort Example I

- Consider this DAG, use Topological Sort to find Shortest Paths in DAG, considering the topological order ADBCEF (Different topological orders will result in different algorithm process, but final results will be the same)

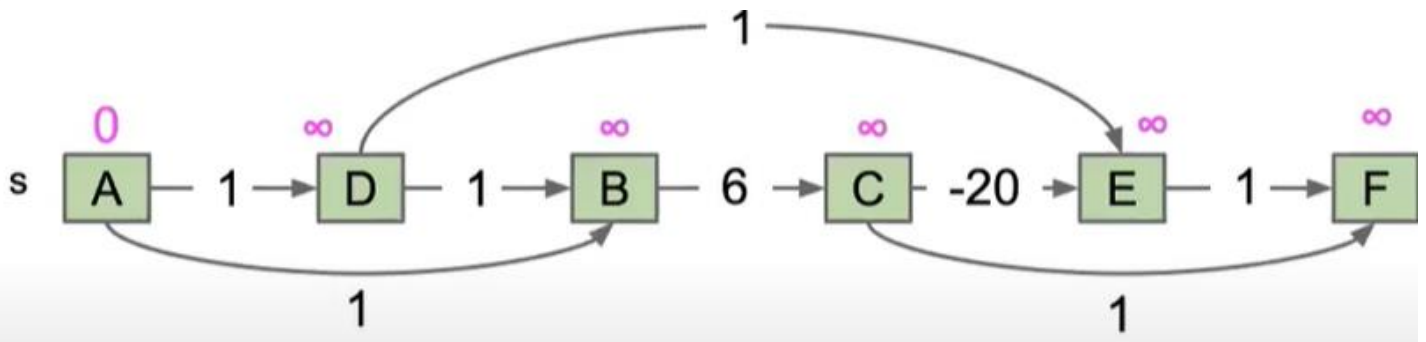




Here we use distTo to denote SD, edgeTo to denote PN

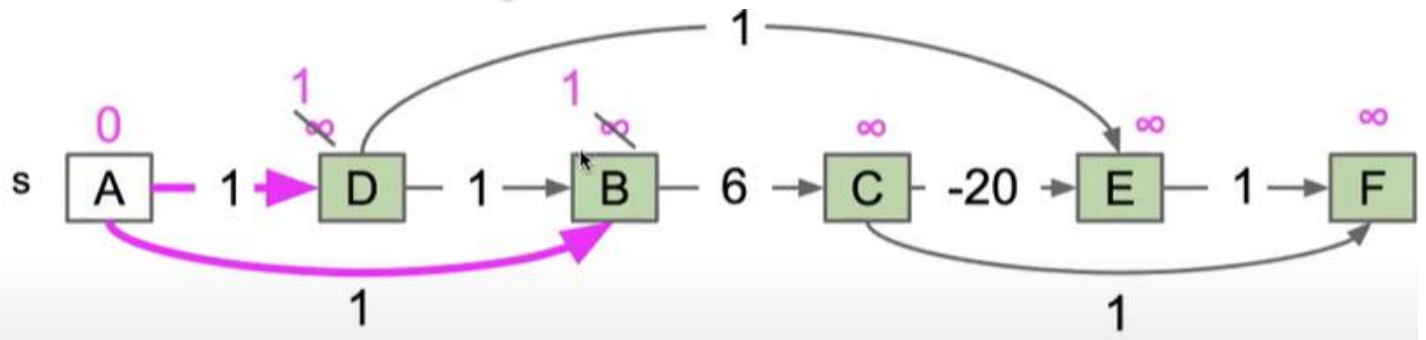
Initialize

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



Visit A

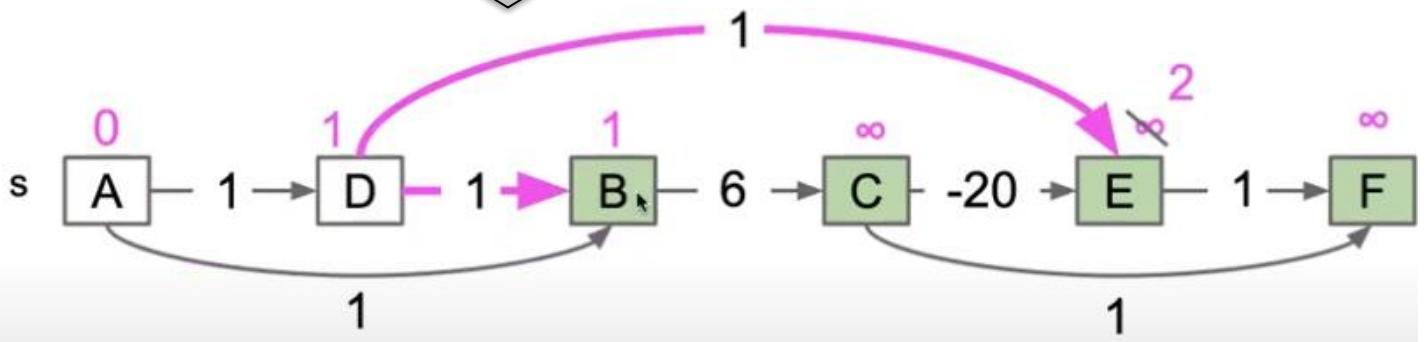
	distTo	edgeTo
A	0	-
B	1	A
C	∞	-
D	1	A
E	∞	-
F	∞	-



Here nodes B and D both have distTo equal to 1, so we may visit either one next. Suppose we choose to visit D next.

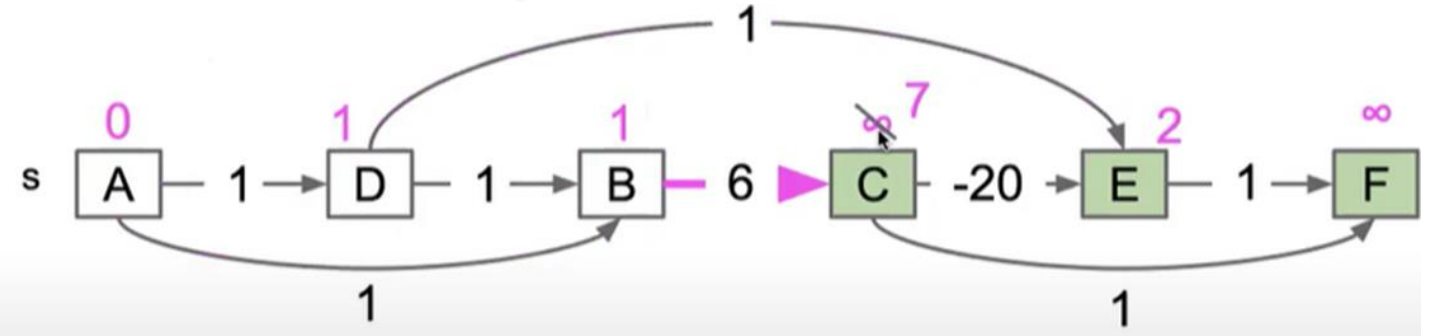
Visit D

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



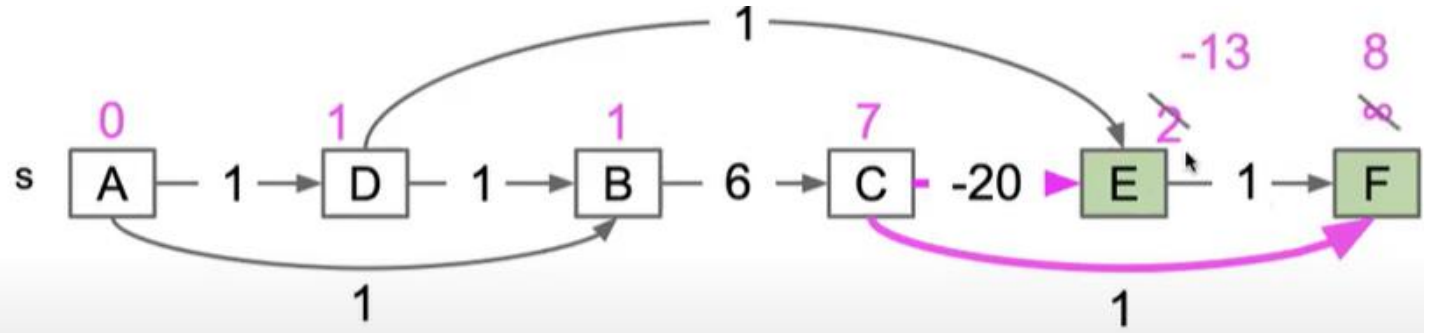
Visit B

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



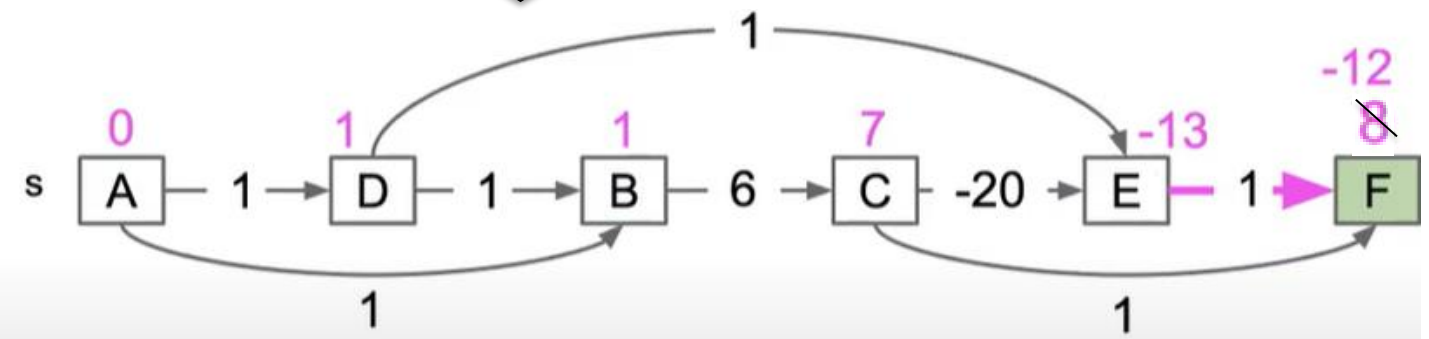
Visit C

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

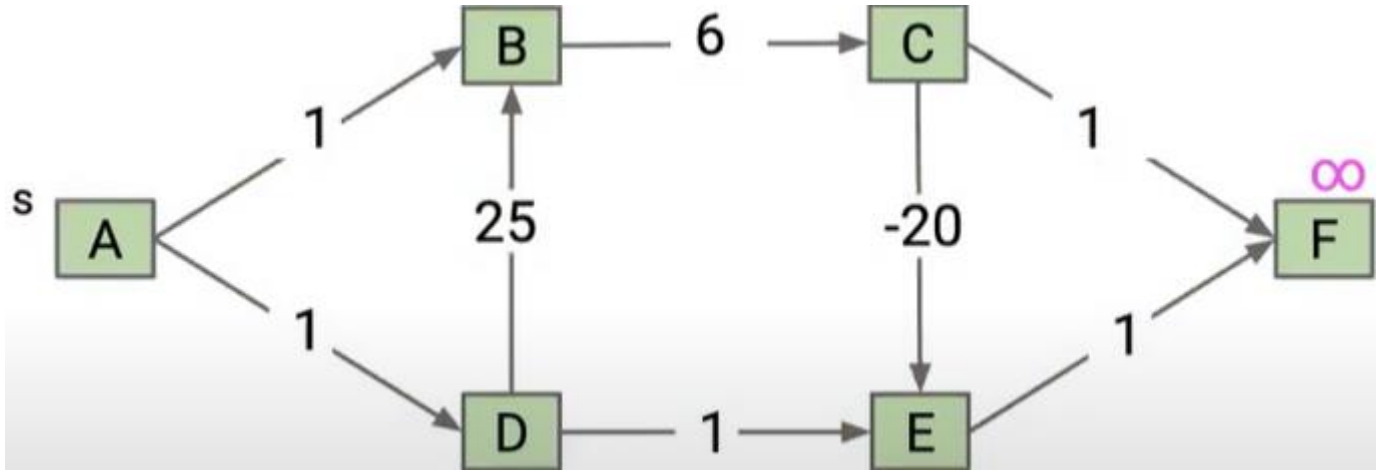


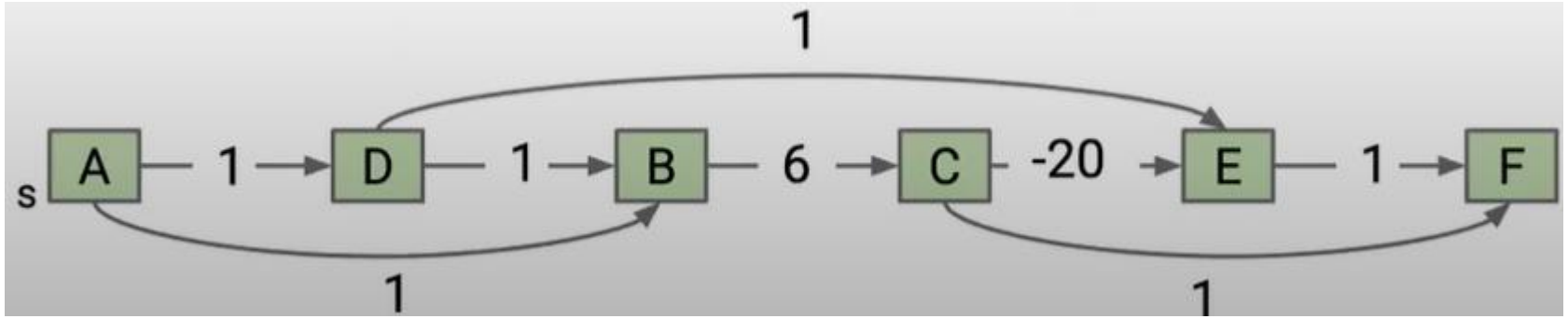
Visit E

	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



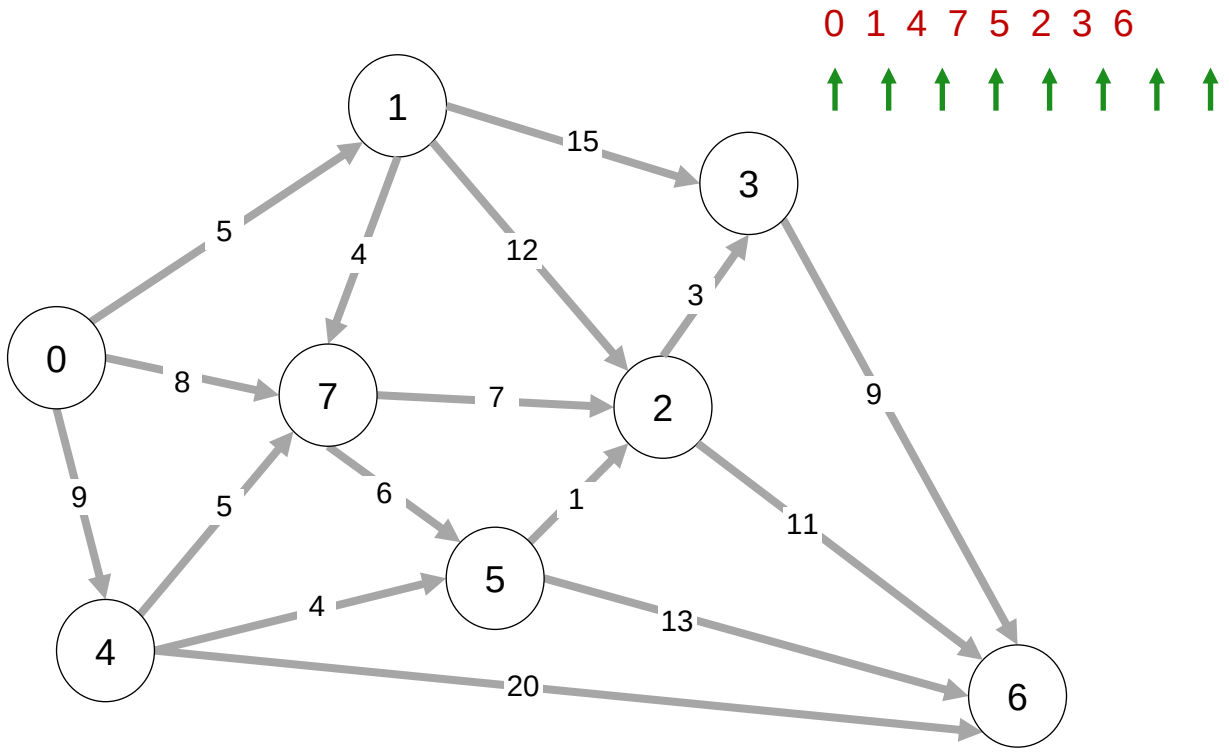


Visit Order
A, D, B, C, E, F

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 01475236



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		

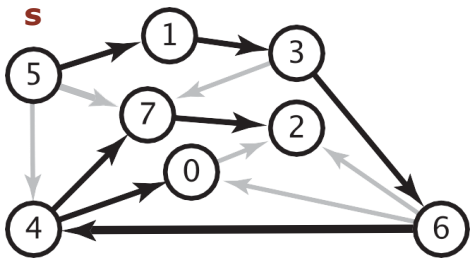
Visit Order
0, 1, 4, 7, 5, 2, 3, 6

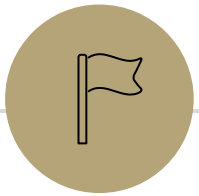
Node	SD	PN
0	0	/
1	5	0
2	~~17~~ ~~15~~ 14	~~1~~ ~~7~~ 5
3	~~20~~ 17	~~1~~ 2
4	9	0
5	13	4
6	~~29~~ ~~26~~ 25	~~4~~ ~~5~~ 2
7	8	0

Longest Paths in a DAG

- Formulate as a shortest paths problem in edge-weighted DAGs.
  - Negate all weights
  - Find shortest paths
  - Negate all the weights and path lengths
- For general graphs, the longest paths problem is an unsolved problem (exponential time at best)

longest paths input		shortest paths input	
5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93





BFS
Dijkstra’s Algorithm
Topological Sort
Bellman-Ford Algorithm
Johnson’s Algorithm

Bellman-Ford Algorithm

- A shortest path algorithm that works with negative edge weights
- There can be at most V - 1 edges in our shortest path
  - If there are V or more edges in a path that means there's a cycle/repeated node
- Run V - 1 iterations of shortest path analysis through the graph
  - Repeatedly revisit and update SD and PN
- Look at each node's outgoing edges in each iteration
- It is slower than Dijkstra's because it will revisit previously assessed nodes
- Can terminate early when all SD values have converged
- The order of edge relaxations affects algorithm efficiency but not correctness
- Time complexity:
  - Worst Case: O(VE)
  - Average 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, by Michael Sambol
  - https://www.youtube.com/watch?v=9PHkk0UavIM
  - https://www.youtube.com/watch?v=obWXjtg0L64

Bellman–Ford algorithm

For each node v: SD [v] = ∞.

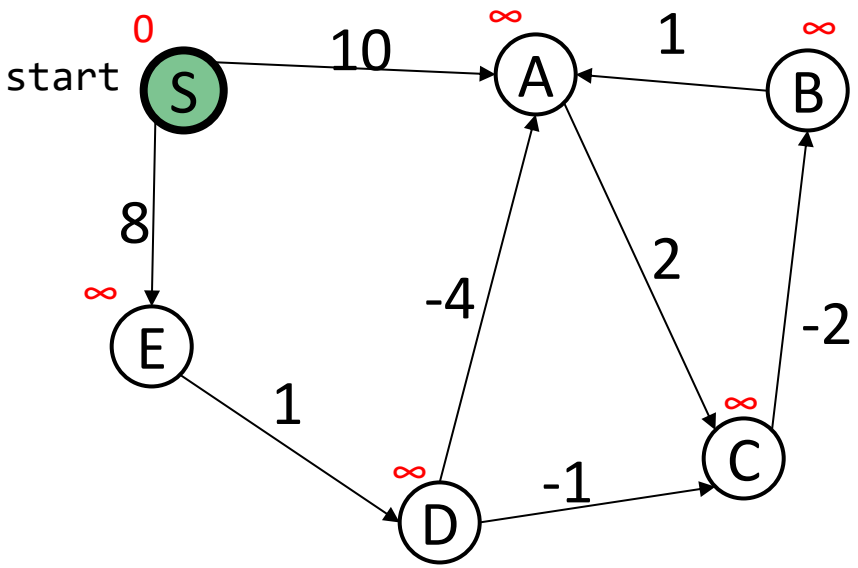
For each node v: PN[v] = null.

SD [s] = 0.

Repeat V-1 times:

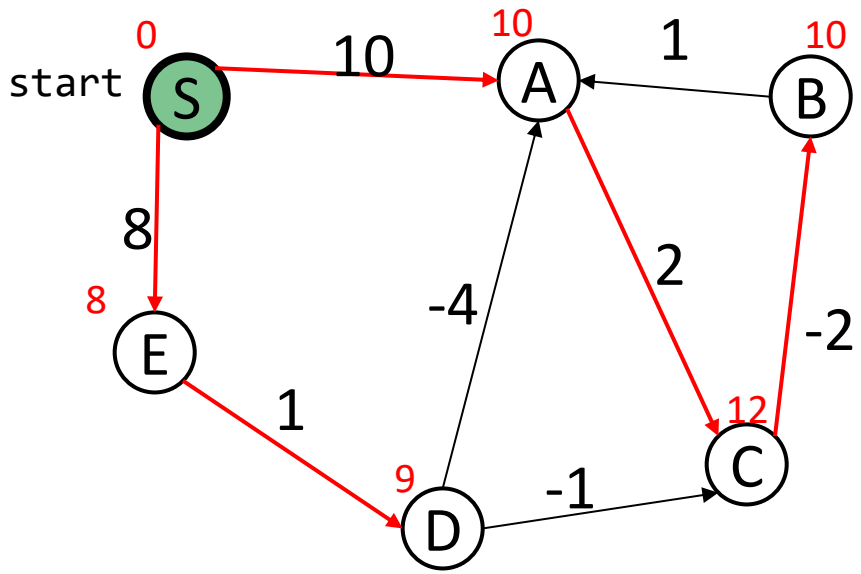
- Relax each edge.

Bellman-Ford Example



node	SD	PN
S	0	
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	

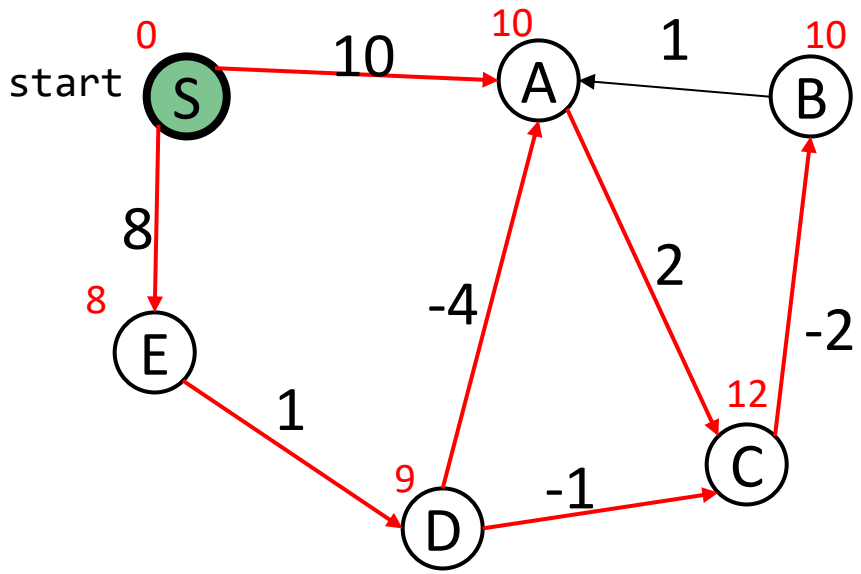
Bellman-Ford Example



Iteration 1 - for each node's outgoing edge, does that give us a shorter way to get to a new node?

node	SD	PN
S	0	-
A	10	S
B	10	C
C	12	A
D	9	E
E	8	A

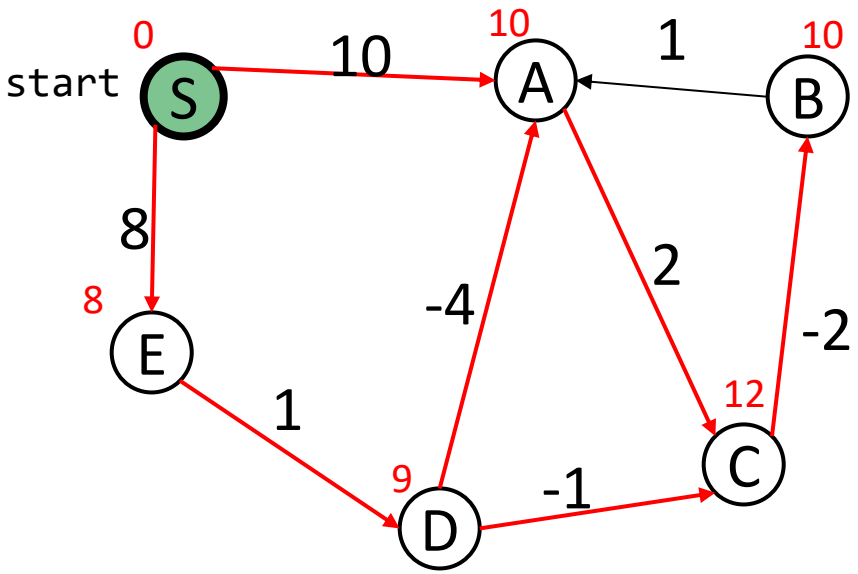
Bellman-Ford Example



Iteration 2 - re-examining outgoing edges, can we improve the distance to any given node?

node	SD	PN
S	0	-
A	~~10~~ **5**	~~S~~ **D**
B	10	C
C	~~12~~ **8**	~~A~~ **D**
D	9	E
E	8	A

Bellman-Ford Example



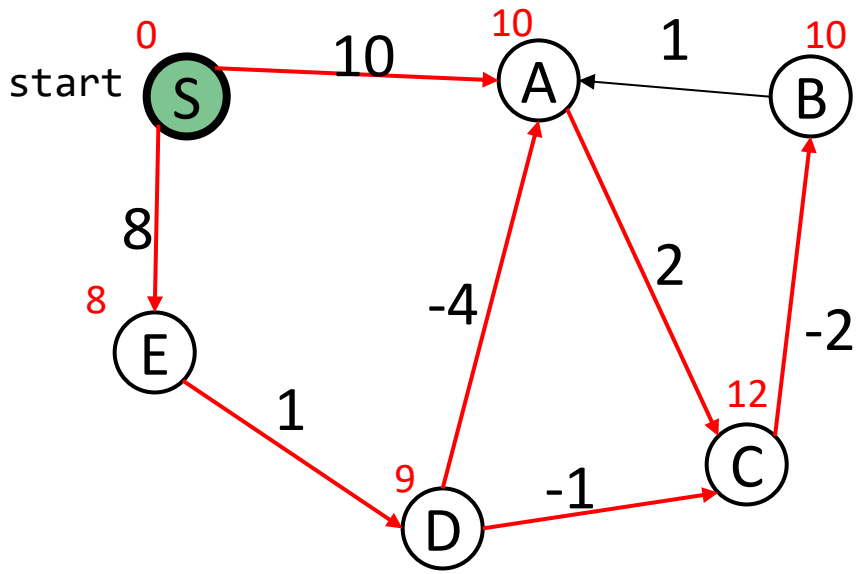
Iteration 3 - repeat!

node	SD	PN
S	0	-
A	5	D
B	~~10~~ **5**	C
C	~~8~~ **7**	A
D	9	E
E	8	A

* With a shortened distance to A from iteration 2 we can improve the distance to C

* With a shortened distance to C in this iteration we can improve distance to B

Bellman-Ford Example



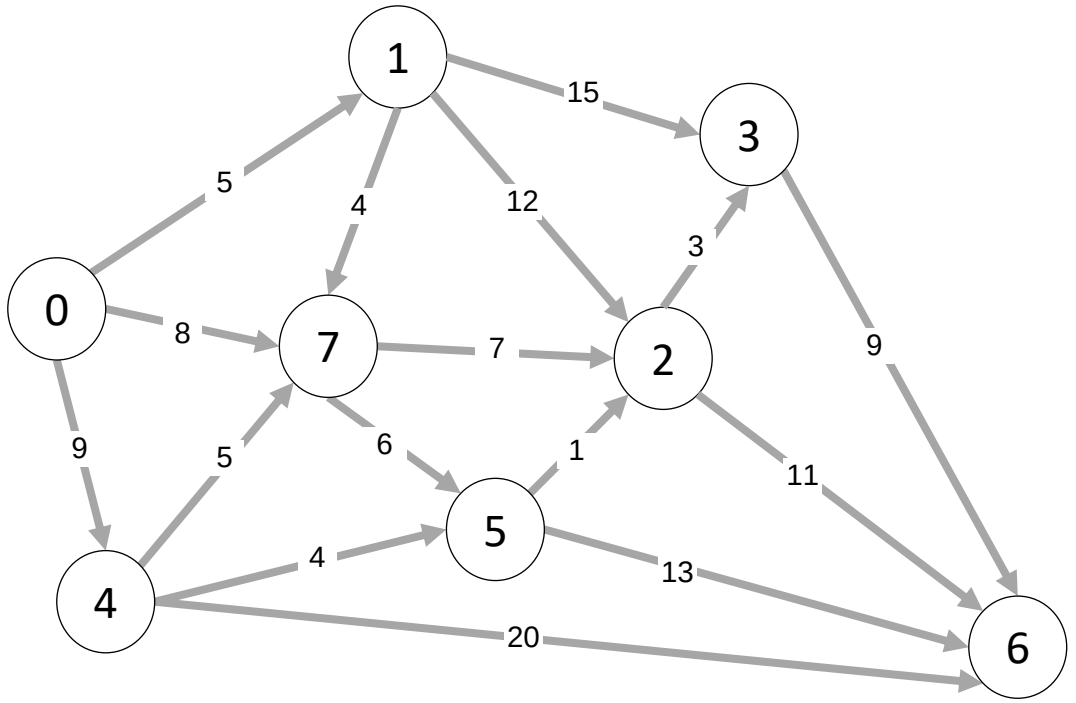
Iteration 4 - repeat!

node	SD	PN
S	0	-
A	5	D
B	5	C
C	7	A
D	9	E
E	8	A

No changes!
this means we can stop early

I will not ask you to run Bellman-Ford algo in the final exam

Bellman-Ford Example II



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

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

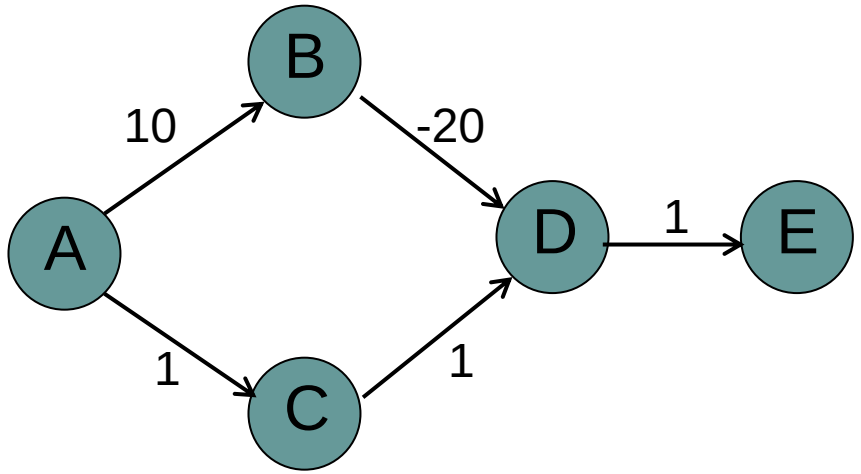
Reverse order of edge relaxations will result in slower convergence

Iter 1 Iter 2 Iter 3 (converged, no further changes, so stop here)

I will not ask you to run Bellman-Ford algo in the final exam

A Toy Example with Negative Edge Weights

- Let's run Dijkstra's algorithm, Topological Sort, and Bellman Ford Algorithm on this DAG with a negative edge weight



Dijkstra's algorithm does not work for a graph w/ negative edge weights

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visit A

N	SD	PN
A	0	
B	10	A
C	1	A
D	∞	
E	∞	

Visit C

N	SD	PN
A	0	
B	10	A
C	1	A
D	2	C
E	∞	

Visit D

N	SD	PN
A	0	
B	10	A
C	1	A
D	2	C
E	3	D

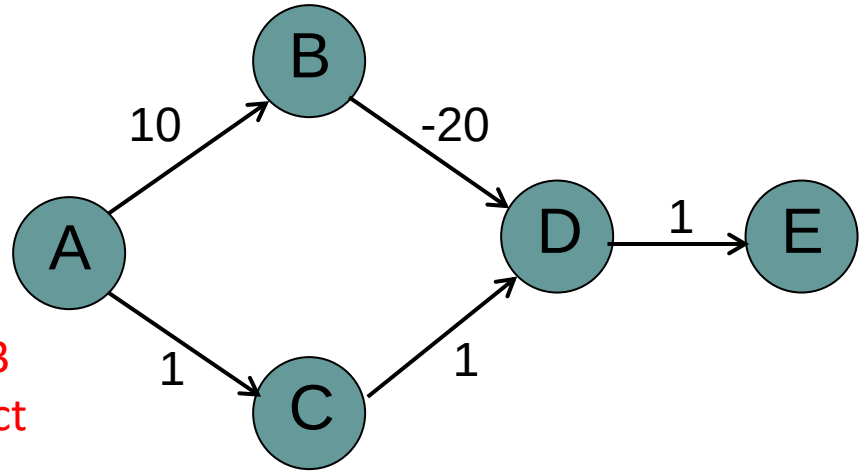
Visit B

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	3	D

Visit E

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	3	D

E's SD of 3 is Incorrect



Dijkstra's algorithm does not work for a graph w/ negative edge weights

- Dijkstra's Algorithm is greedy: each node is visited once, and any node that has been visited should have its shortest distance from the source.
- After visiting A, C, D, we have computed D's SD is 2, but after visiting B, D's SD is updated to -10, which violates the greedy optimal assumption.
- Even if we update D's SD to be -10, its downstream node E's SD will not be updated.
- We cannot visit B before D, since we must visit the closest unknown node (with smallest SD), which is D. (Unlike topological sort, which visits B before D and gets the correct result,)

Topological sort works for a DAG w/ negative edge weights

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visit A

N	SD	PN
A	0	
B	10	A
C	1	A
D	∞	
E	∞	

Visit B

N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	∞	

Visit C

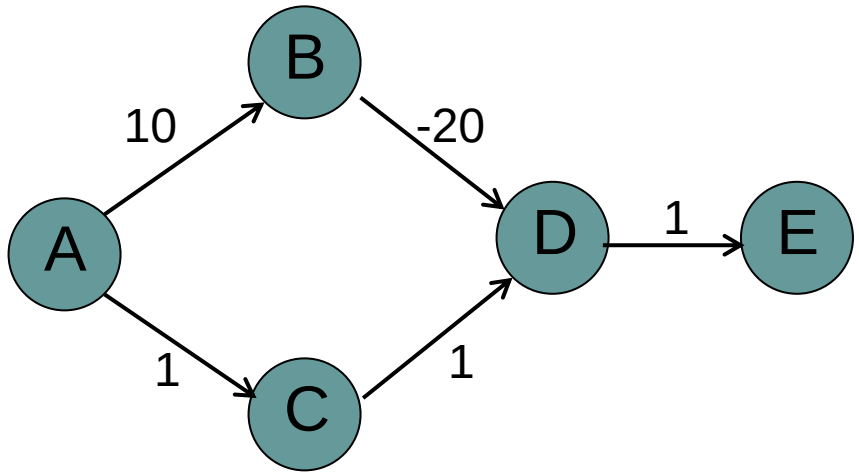
N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	∞	

Visit D

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

Visit E

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



Consider topological order ABCDE

Topological sort works for a DAG w/ negative edge weights

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visit A

N	SD	PN
A	0	
B	10	A
C	1	A
D	∞	
E	∞	

Visit C

N	SD	PN
A	0	
B	10	A
C	1	A
D	2	C
E	∞	

Visit B

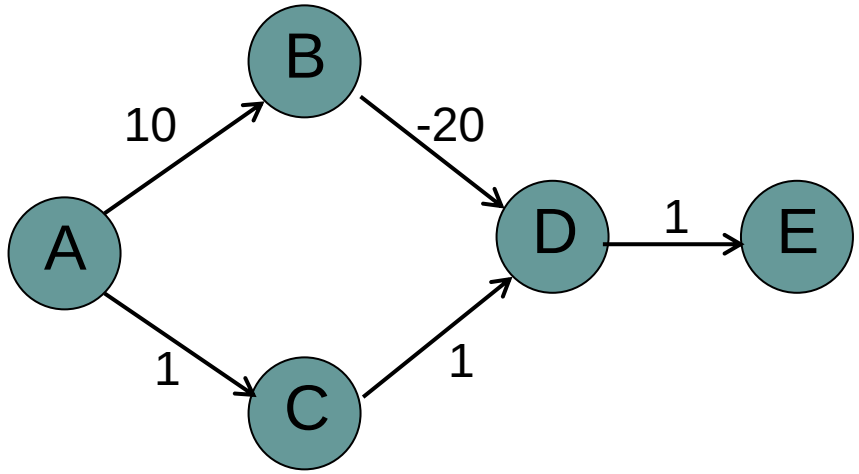
N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	∞	

Visit D

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

Visit E

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



Consider topological order ACBDE

Bellman Ford works for a graph with negative edge weights

N	SD	PN
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Iter 1

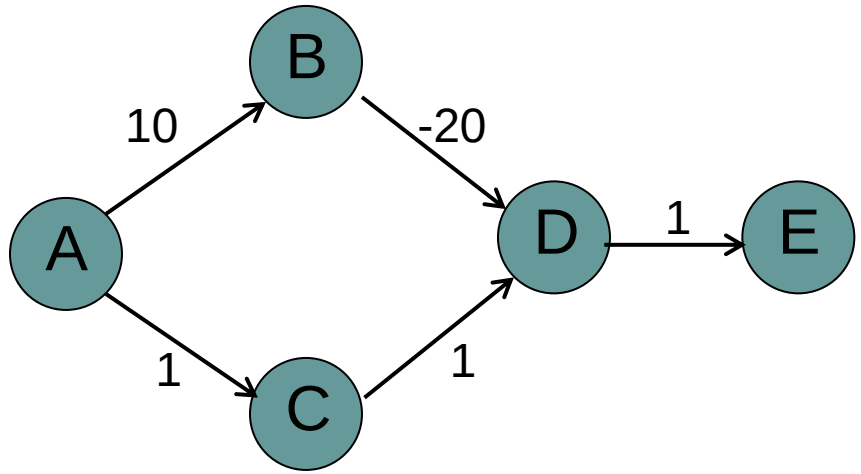
N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	-9	D

Iter 2

No change converged

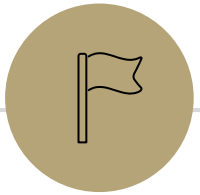
N	SD	PN
A	0	
B	10	A
C	1	A
D	-10	B
E	-9	D

Suppose we visit nodes in this order at each iteration: A, B, C, D, E. We run for 2 iterations (less than V-1=3 iterations), and converge to the correct SPT



Dijkstra's Algorithm vs. Bellman-Ford Algorithm

- Dijkstra's Algorithm:
  - Uses a priority queue to select the next node to process.
  - Greedily selects the node with the smallest tentative distance to source node.
  - Works only on graphs with non-negative edge weights.
- Bellman-Ford Algorithm:
  - Iteratively relaxes all edges V-1 times.
  - Can handle graphs with negative edge weights, and can detect negative cycles.
    - Relax all the edges one more time, i.e. the V-th time. Negative cycle exists if any edge can be further relaxed
    - It can find any negative cycle that is reachable from source node s (but not negative cycles that are unreachable from s).
    - If there is a negative cycle that is reachable from source node 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.
- 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



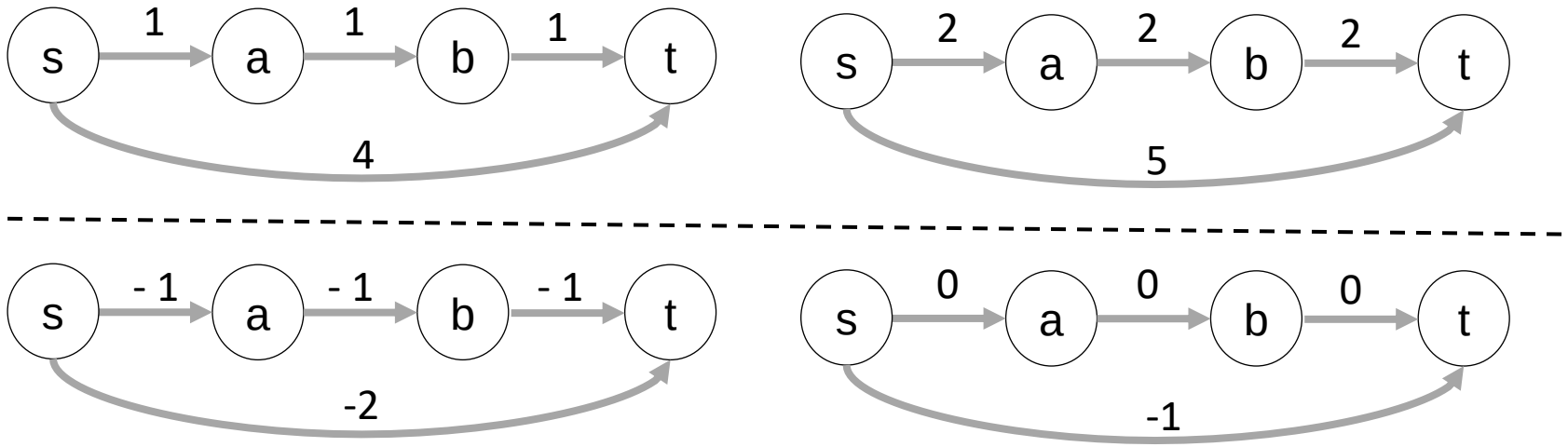
BFS
Dijkstra’s Algorithm
Topological Sort
Bellman-Ford Algorithm
Johnson’s Algorithm

Johnson’s Algorithm for all-pairs shortest paths

- Idea: run Dijkstra’s Single Source shortest path algorithm with every node as the source.
- Dijkstra’s algorithm doesn’t work for negative weight edge. The idea of Johnson’s algorithm is to reweight all edges and make them all positive, then run Dijkstra’s algorithm with every node as the source.
  - We can run Bellman-Ford algorithm with every node as the source without reweighting, since Bellman-Ford algorithm can handle negative edge weights, but the time complexity is much higher than running Dijkstra’s algorithm.
- How to transform a given graph into a graph with non-negative weight edges without changing the shortest paths?

Example 1: 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 sa, ab, bt and st with weights 1, 1, 1 and 4 respectively. The shortest path from s to t is sabt with cost 3. If we increase weight of every edge by 1, the shortest path changes to st with cost 5.
- Similarly for negative weight edges. There are 4 edges sa, ab, bt and st with weights -1, -1, -1 and -2 respectively. The shortest path from s to t is sabt with cost -3. If we increase weight of every edge by 1, the shortest path changes to st with cost -1.



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 becomes double of the original weight, then the shortest path remains same, and 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 do not change. But this does not make weights positive.

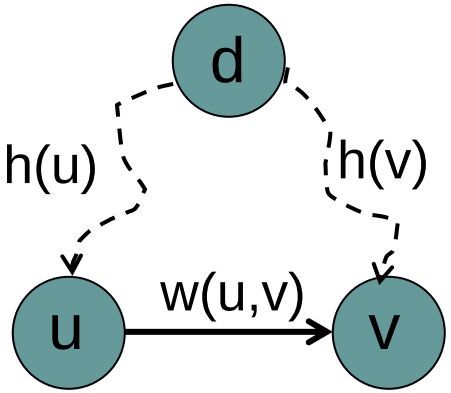
Johnson's algorithm for All-pairs Shortest Paths

1. Let the given graph be G. Add a dummy source node d, and add edges with weight 0 from d to all vertices of G. Let the modified graph be G'.
2. Run Bellman-Ford on G' with d as the source. Let the shortest distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. (We run Bellman-Ford algorithm since it can handle negative edge weights.)
3. Reweight the edges of the original graph. For each edge (u, v), assign the new weight as w'(u, v) = w(u, v) + h[u] – h[v], which is greater than or equal to 0.
4. Remove the added dummy node d, and run Dijkstra's algorithm with every node as the source to obtain all-pairs shortest paths. Subtract h[s] – h[t] from length of each shortest path from s to t to obtain the lengths of shortest paths in the original graph.

Time complexity: Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines. 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 + E) log V). So overall time complexity is O((V^2 + VE) log V).

Johnson's Algorithm: Proof

- The following property is always true since h[] values are the shortest distances from the dummy source code d:
  - h[v] <= h[u] + w(u, v)
- The property states that the shortest distance from u to v must be smaller than or equal to the shortest distance from s to u plus edge weight w(u, v).
  - If h[v] > h[u] + w(u, v), then Bellman-Ford with starting node d will set h[v] = h[u] + w(u, v), after visiting u and performing edge relaxation
- Because of this inequality, the new weights w'(u, v) = w(u, v) + h[u] – h[v] must be greater than or equal to 0.
- After reweighting, all weights become non-negative, and lengths of all paths between any two vertices s and t is increased by the same amount, hence the shortest paths remain the same as the original graph before reweighting.
  - Consider any path between two vertices s and t, the weight of every path is increased by h[s] – h[t], since the added h[] values for all intermediate vertices on the path from s to t cancel each other out
  - Consider two path between s and t: s->x->y->t, and s->x'->y'->t. Distance of the 1st path s->x->y->t is increased by h(s)-h(x)+h(x)-h(y)+h(y)-h(t)=h(s)-h(t), and distance of the 2nd path s->x'->y'->t is increased by h(s)-h(x')+h(x')-h(y')+h(y')-h(t)=h(s)-h(t)

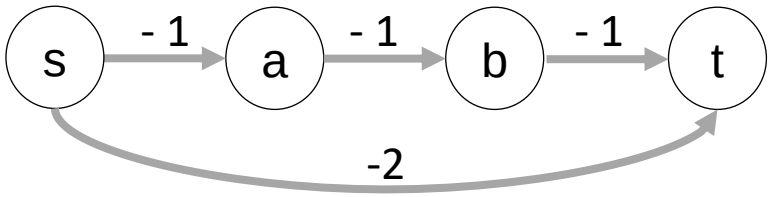


Johnson's Algorithm Example I

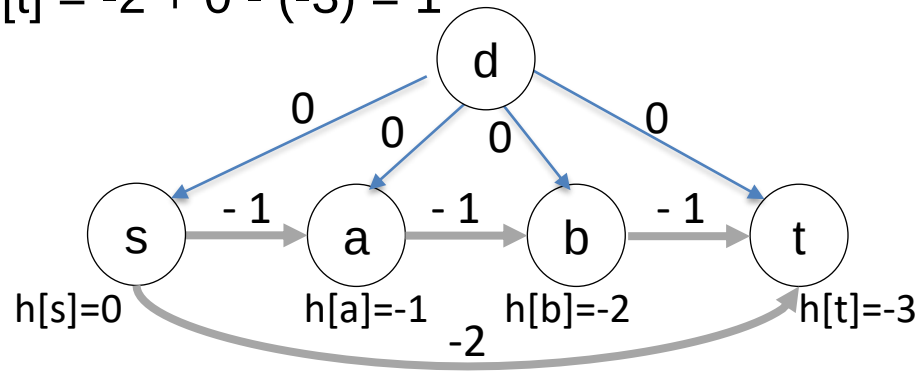
- Add a dummy source node d and add edges with weight 0 from s to all vertices of the original graph.
- Run Bellman-Ford algorithm to calculate the shortest distances from d to all other vertices. The shortest distances from d to s, a, b and t are
  - h[s]=0 (path d→0), h[a]=-1 (path d→s→a), h[b]=-2 (path d→s→a→b), h[t]=-3 (path d→s→a→b→t)
- Remove node d and reweight each edge uv as: w'(u, v) = w(u, v) + h[u] – h[v].
  - w'(s, a) = w(s, a) + h[s] – h[a] = -1 + 0 - (-1) = 0
  - w'(a, b) = w(a, b) + h[a] – h[b] = -1 + -1 - (-2) = 0
  - w'(b, t) = w(b, t) + h[b] – h[t] = -1 + -2 - (-3) = 0
  - w'(s, t) = w(s, t) + h[s] – h[t] = -2 + 0 - (-3) = 1

N	h(N)
s	0
a	-1
b	-2
t	-3

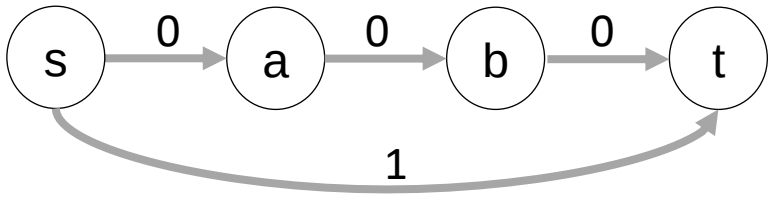
Shortest paths starting from dummy node



Original graph



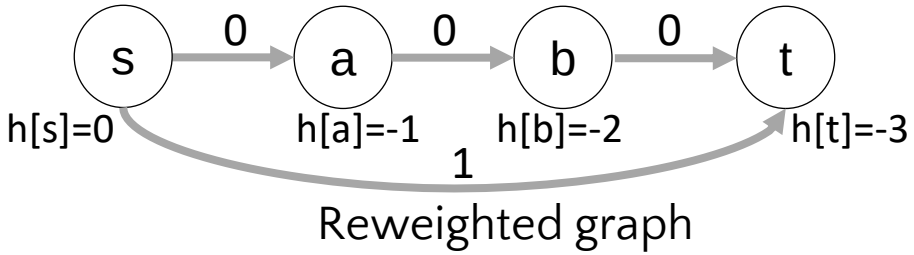
Original graph w/ dummy node



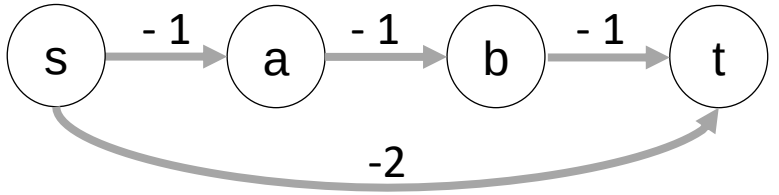
Reweighted graph

Johnson's Algorithm Example I Con't

- Since all weights are now greater than or equal to 0, run Dijkstra's shortest path algorithm on the reweighted graph with every node as the source. Let's start from source node s for example, and obtain the shortest paths table for the reweighted graph
- We then subtract h[s] – h[t] from length of each shortest path from s to t to obtain the shortest paths table for the original graph (PN stays the same)
  - SD(a) = SD'(a) - (h[s] - h[a]) = 0 - (0 - (-1)) = -1
  - SD(b) = SD'(b) - (h[s] - h[b]) = 0 - (0 - (-2)) = -2
  - SD(t) = SD'(t) - (h[s] - h[t]) = 0 - (0 - (-3)) = -3



Reweighted graph



Original graph

N	h(N)
s	0
a	-1
b	-2
t	-3

Shortest paths starting from dummy node

N	SD'	PN
s	0	/
a	0	s
b	0	a
t	0	b

Shortest paths from s in reweighted graph

N	SD	PN
s	0	/
a	-1	s
b	-2	a
t	-3	b

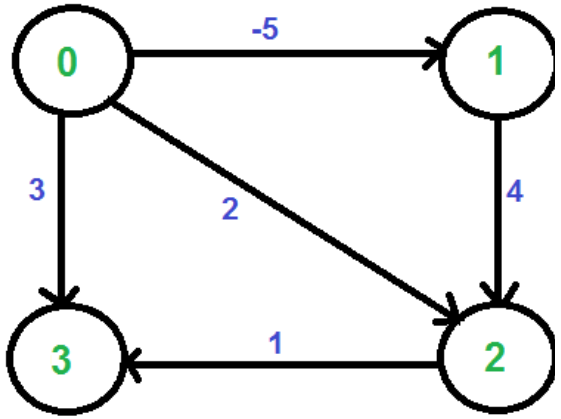
Shortest paths from s in original graph

Johnson's Algorithm Example II

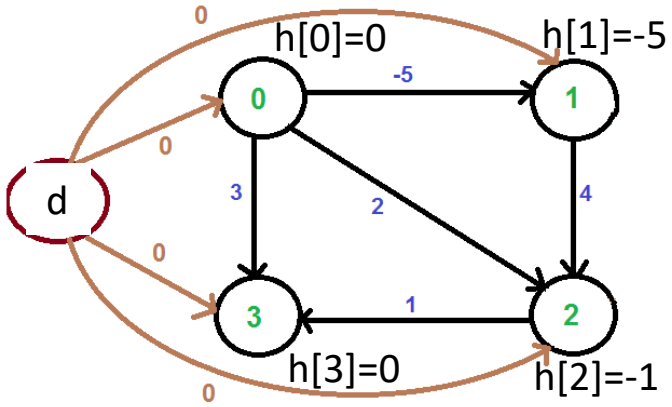
- Add a dummy source node d and add edges with weight 0 from s to all vertices of the original graph.
- Run Bellman-Ford algorithm to calculate the shortest distances from d to all other vertices. The shortest distances from d to 0, 1, 2 and 3 are
  - h[0]=0 (path d→0), h[1]=-5 (path d→0→1), h[2]=-1 (path d→0→1→2), h[3]=0 (path d→3)
- Remove node d and reweight each edge uv as: w'(u, v) = w(u, v) + h[u] – h[v].
  - w'(0,1)=-5+0-(-5)=0, w'(1,2)=4+(-5)-(-1)=0, w'(2,3)=1+(-1)-0=0, w'(0,3)=3+0-0=3, w'(0,2)=2+0-(-1)=3
- Since all weights are now greater than or equal to 0, run Dijkstra's algorithm on the reweighted graph with every node as the source

N	h(N)
0	0
1	-5
2	-1
3	0

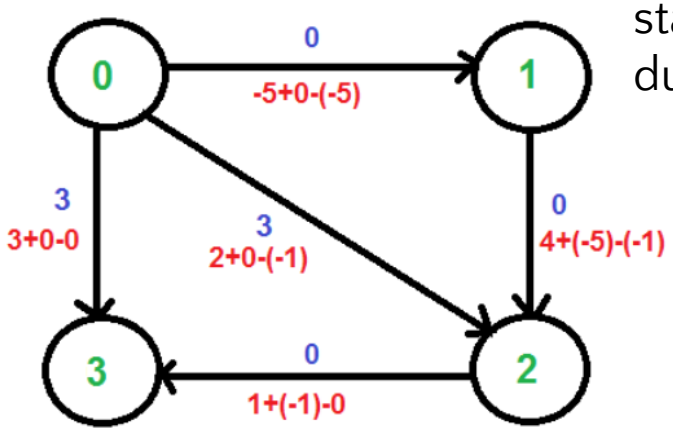
Shortest paths starting from dummy node



Original graph



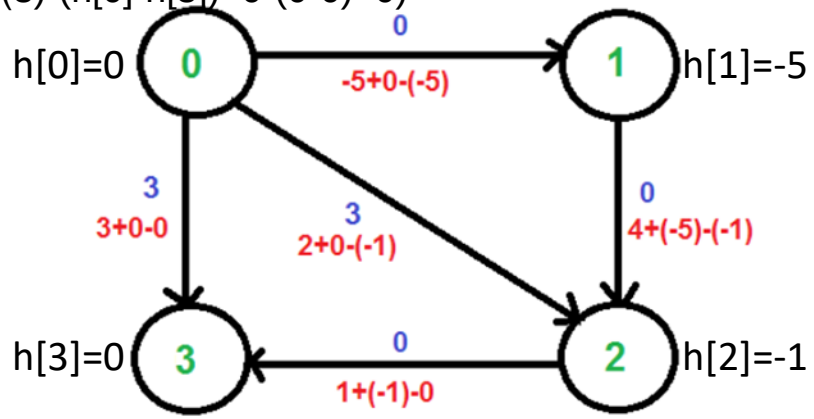
Original graph w/ dummy node



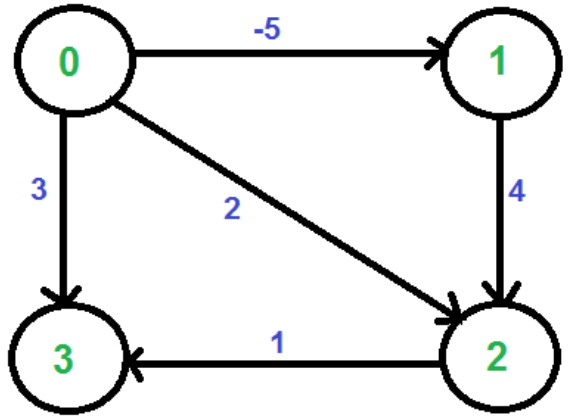
Reweighted graph

Johnson's Algorithm Example II Con't

- Let's run Dijkstra's algorithm starting from source node 0, and obtain the shortest paths table for the reweighted graph
- We then subtract h[s] – h[t] from length of each shortest path from s to t to obtain the shortest paths table for the original graph (PN stays the same)
  - SD(1)=SD'(1)-(h[0]-h[1])=0-(0-(-5))=-5
  - SD(2)=SD'(2)-(h[0]-h[2])=0-(0-(-1))=-1
  - SD(3)=SD'(3)-(h[0]-h[3])=0-(0-0)=0)



Reweighted graph



Original graph

N	h(N)
0	0
1	-5
2	-1
3	0

Shortest paths starting from dummy node

N	SD'	PN
0	0	/
1	0	0
2	0	1
3	0	2

Shortest paths from 0 in reweighted graph

N	SD	PN
0	0	/
1	-5	0
2	-1	1
3	0	2

Shortest paths from 0 in original graph

Single Source Shortest-paths Algorithms Summary

Algorithm	Applicability	Worst-Case Complexity
Breadth First Search (BFS)	Unweighted, undirected or directed graph	Adjacency List: $O(V + E)$ Adjacency Matrix: $O(V^2)$
Dijkstra's algorithm	Undirected or directed graph; no negative weights/cycles	$O((V+E) \log V)$ (binary min-heap)
Topological Sort	Directed Acyclic Graph (DAG) (no cycles, negative weights OK)	$O(V+E)$
Bellman-Ford algorithm	Directed graph with negative weights; undirected graph with no negative weights (since a negative weight edge forms a negative cycle by itself)	$O(VE)$
Johnson's algorithm	Same as Bellman-Ford	$O((V^2 + VE) \log V)$

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
- Topological sort
  - https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/
- Bellman-Ford, Michael Sambol
  - https://www.youtube.com/watch?v=9PHkk0UavIM
  - https://www.youtube.com/watch?v=obWXjtg0L64
- Bellman Ford Shortest Path Algorithm, ByteQuest
  - https://www.youtube.com/watch?v=B5PmlJACZ9Y
- Shortest Path Algorithms Explained (Dijkstra's & Bellman-Ford), b001
  - https://www.youtube.com/watch?v=TtQi1LVVOUI
- Johnson’s algorithm for All-pairs shortest paths
  - https://www.geeksforgeeks.org/johnsons-algorithm/
- Johnson's Algorithm Explained, Basics Strong
  - https://www.youtube.com/watch?v=MV7EAD9zL64
  - @14:36 The 0->4 edge weight was incorrectly changed from 1 to 0

References

- [CSE 373 WI24] Lecture 15: Shortest Path
  - https://www.youtube.com/watch?v=L8nhMwhUn4U&list=PLEcoVsAaONjd5n69K84sSmAuvTrTQT_Nl&index=14