Working with Shortest Path Algorithms

Janani Ravi
CO-FOUNDER, LOONYCORN

www.loonycorn.com

Three Common Graph Problems

Establishing precedence

Getting from point A to point B

Covering all nodes in a graph

Three Common Graph Problems

Establishing precedence

Topological sort

Getting from point A to point B

Shortest path algorithms

Covering all nodes in a graph

Minimum spanning tree algorithms

Three Common Graph Operations

Topological sort

Computation graphs in neural networks

Shortest path

Deliveries from warehouses to customers

Minimum spanning tree

Planning railway lines

Three Common Graph Operations

Topological sort

Computation graphs in neural networks

Shortest path

Deliveries from warehouses to customers

Minimum spanning tree

Planning railway lines

Overview

Shortest path algorithms are widely used in transportation and scheduling

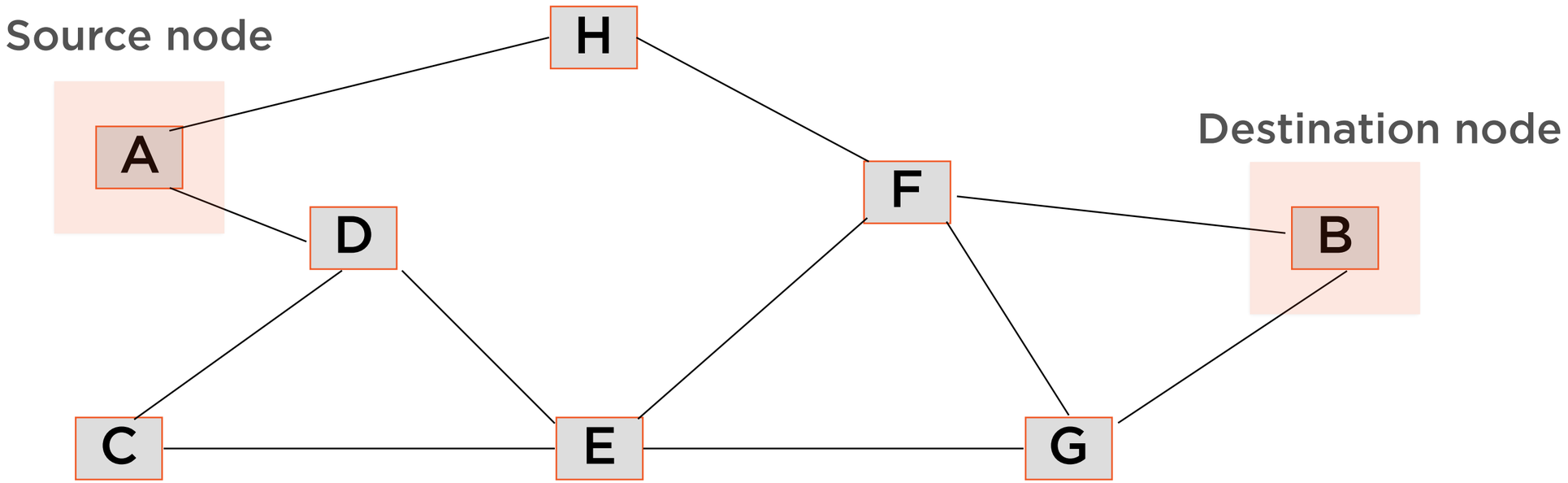
Such algorithms focus on the most efficient route between a pair of nodes

Edge weights determine the cost of a path in such algorithms

If all edge weights are equal, use the unweighted shortest path algorithm

If edge weights are unequal, use Dijkstra's algorithm

Shortest Path Algorithms



Problem: Find the shortest path between a source node and a destination node

Getting from Point A to Point B

Mapping routes

Route through less congested roads

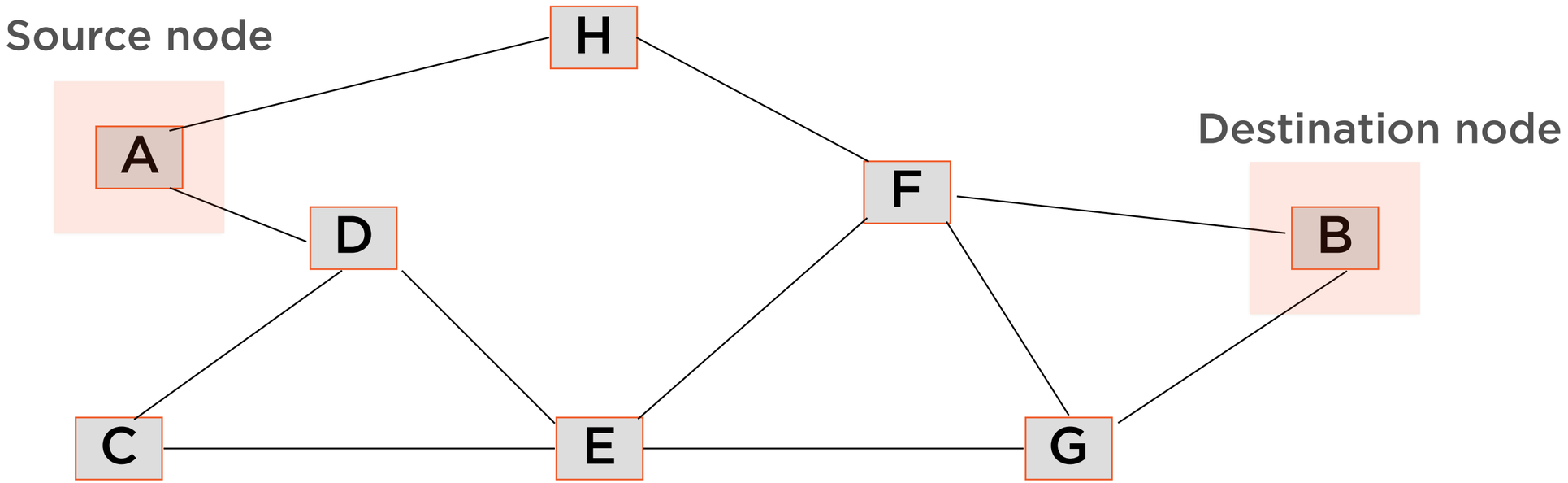
Scheduling deliveries

Multiple deliveries to multiple locations

Building roads

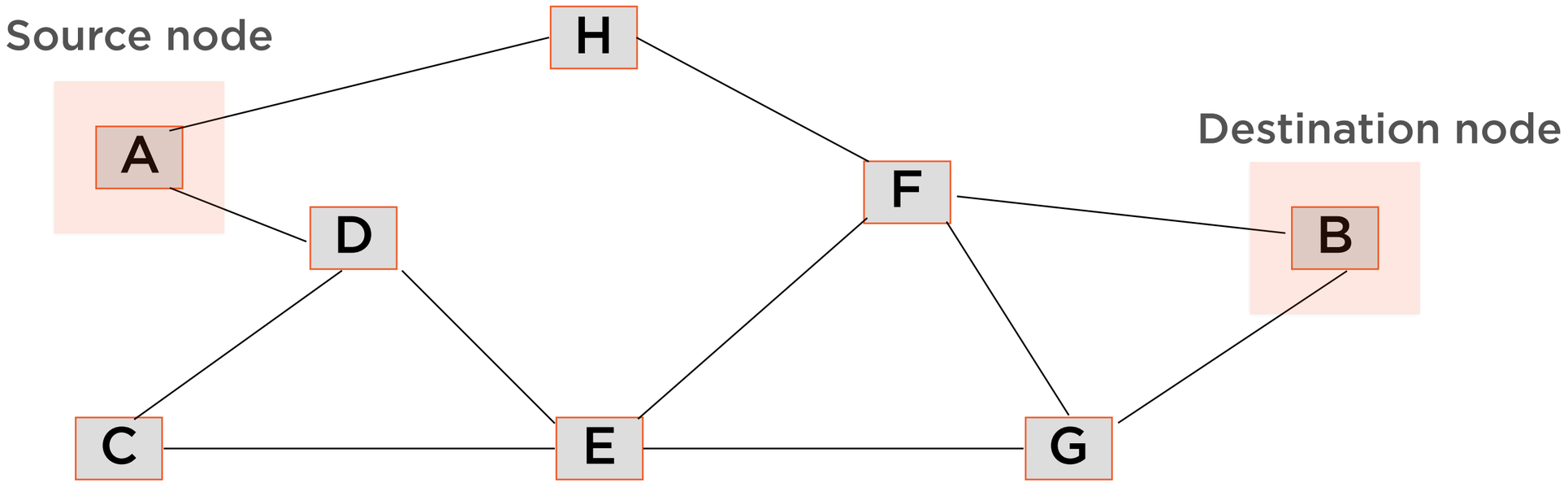
Costly to ford rivers, pass mountains

Shortest Path Algorithms



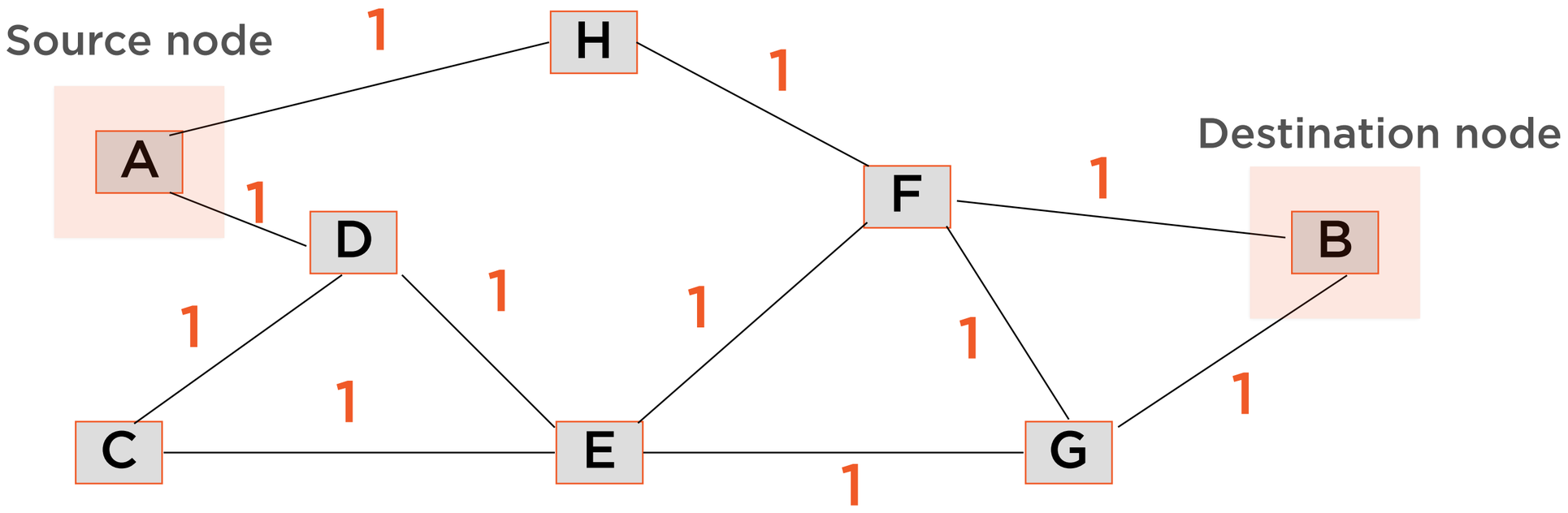
Problem: Find the shortest path between a source node and a destination node

Shortest Path Algorithms



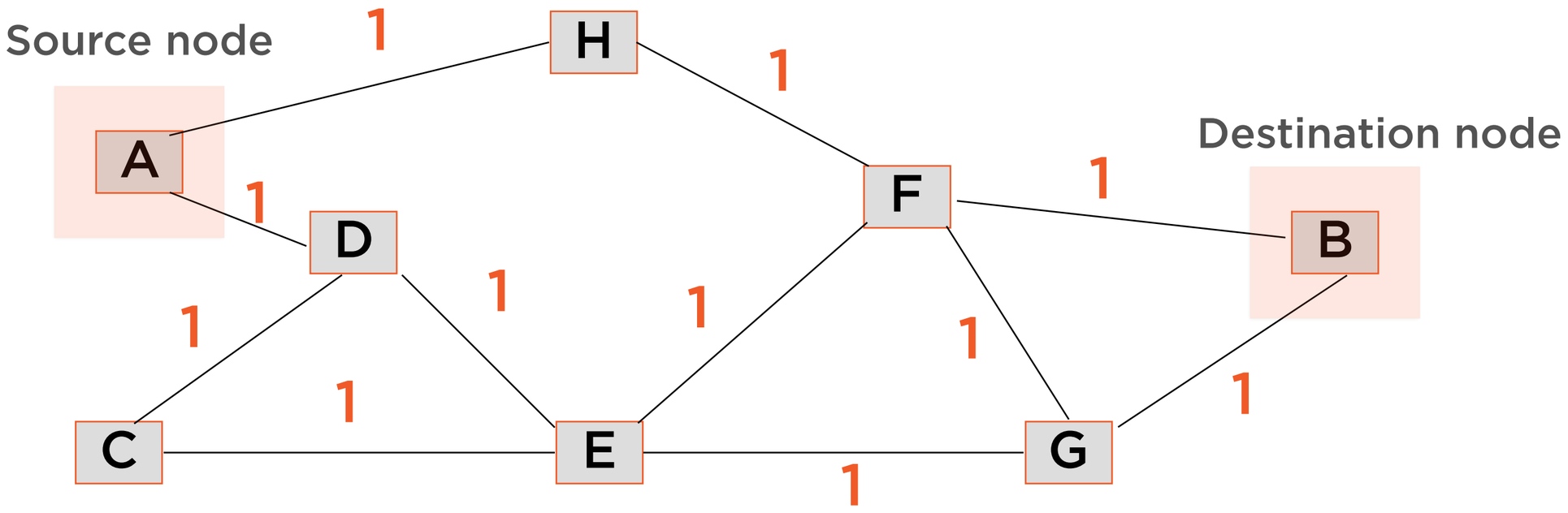
Clearly, the shortest path depends on how we measure the length of an edge

Unweighted Graphs



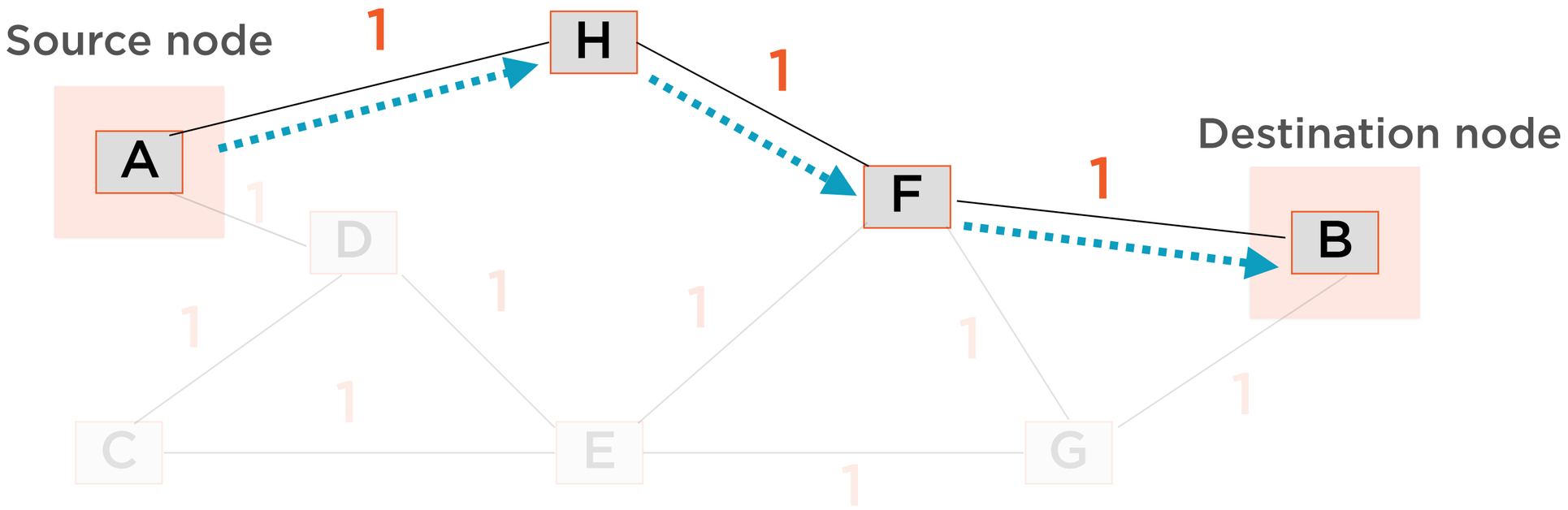
All edges have equal weight (=1)

Unweighted Graphs



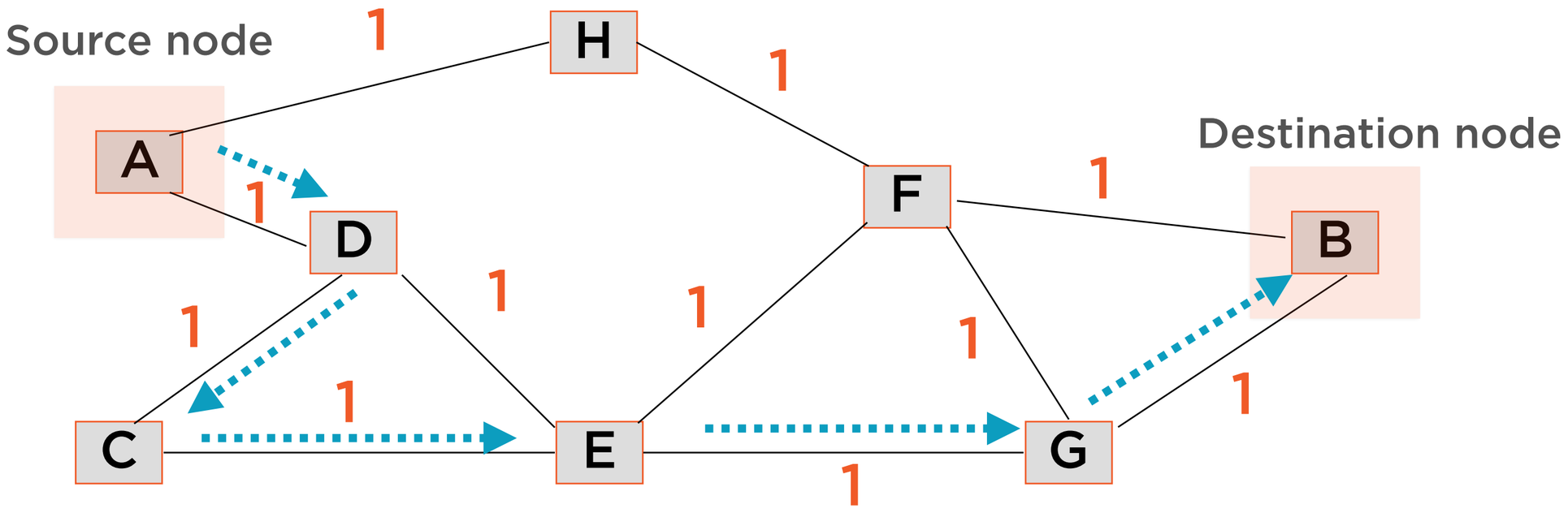
Here the shortest path is the path with the least hops

Unweighted Graphs



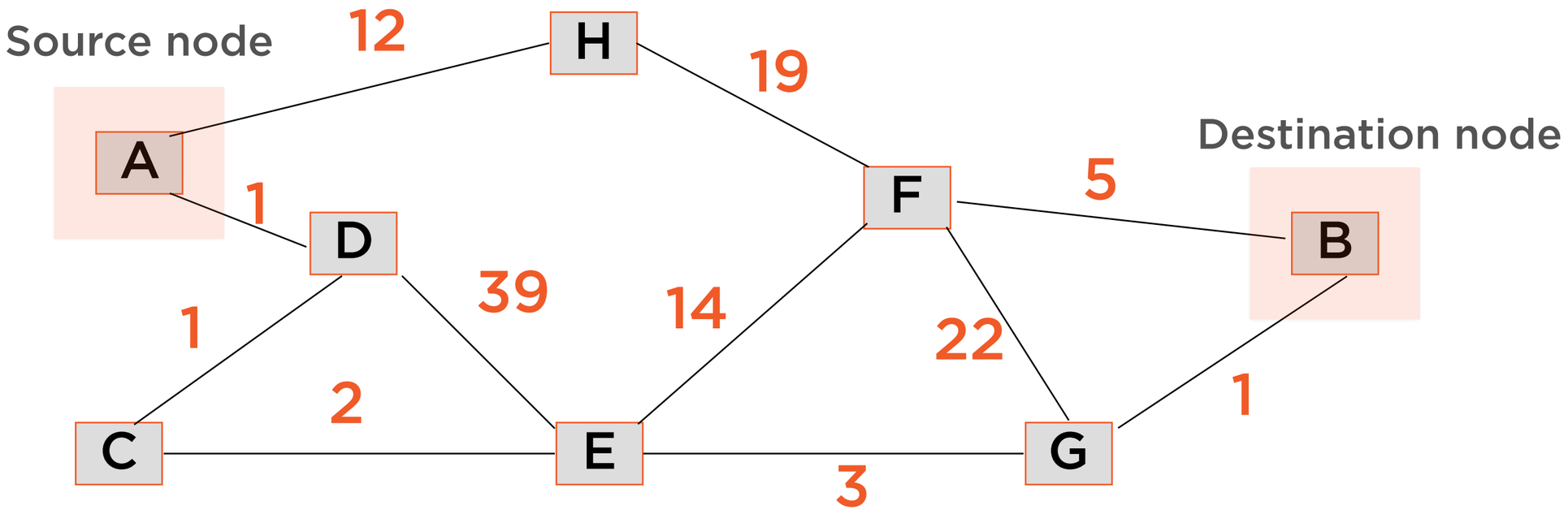
Cost of shortest path = number of hops = 3

Unweighted Graphs



Other longer paths exist, number of hops = 5

Weighted Graphs

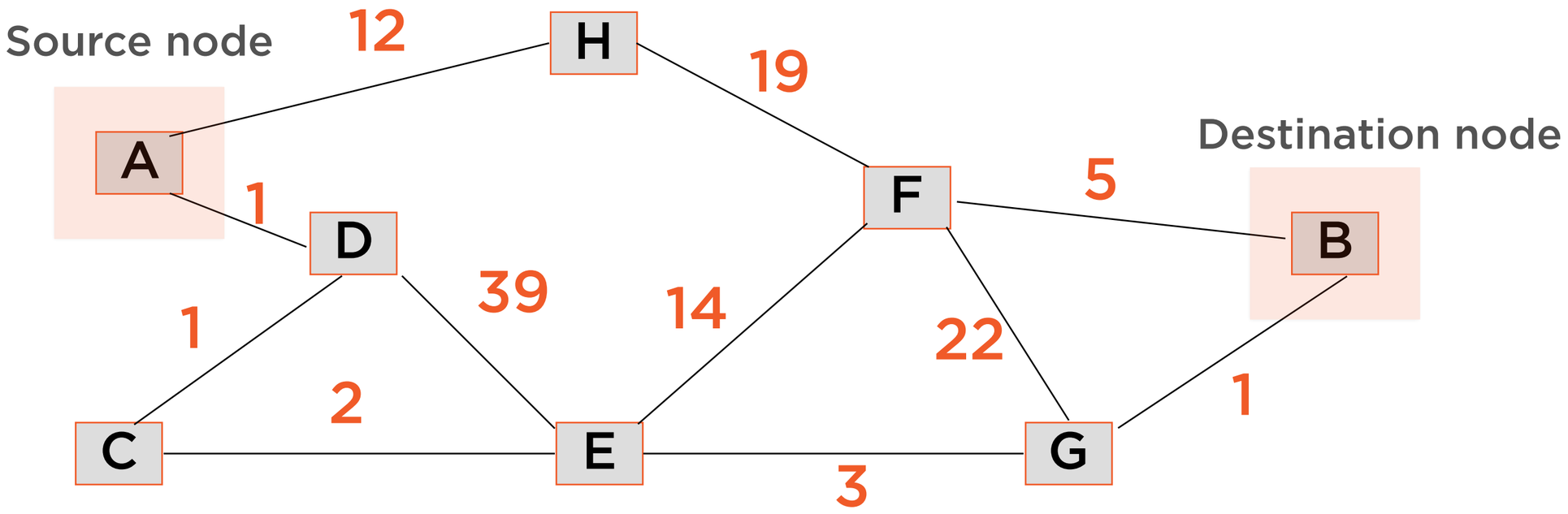


When edges have differing weights, finding shortest path is more complicated

Time taken to drive between two locations

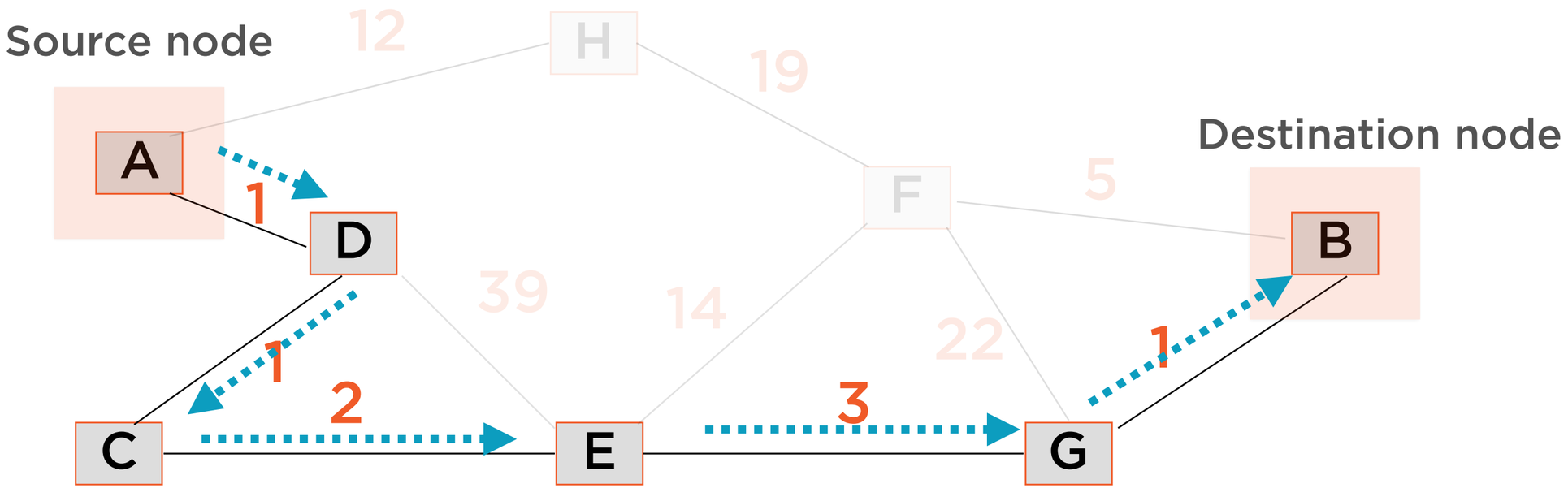
Cost to construct a road between two locations

Weighted Graphs



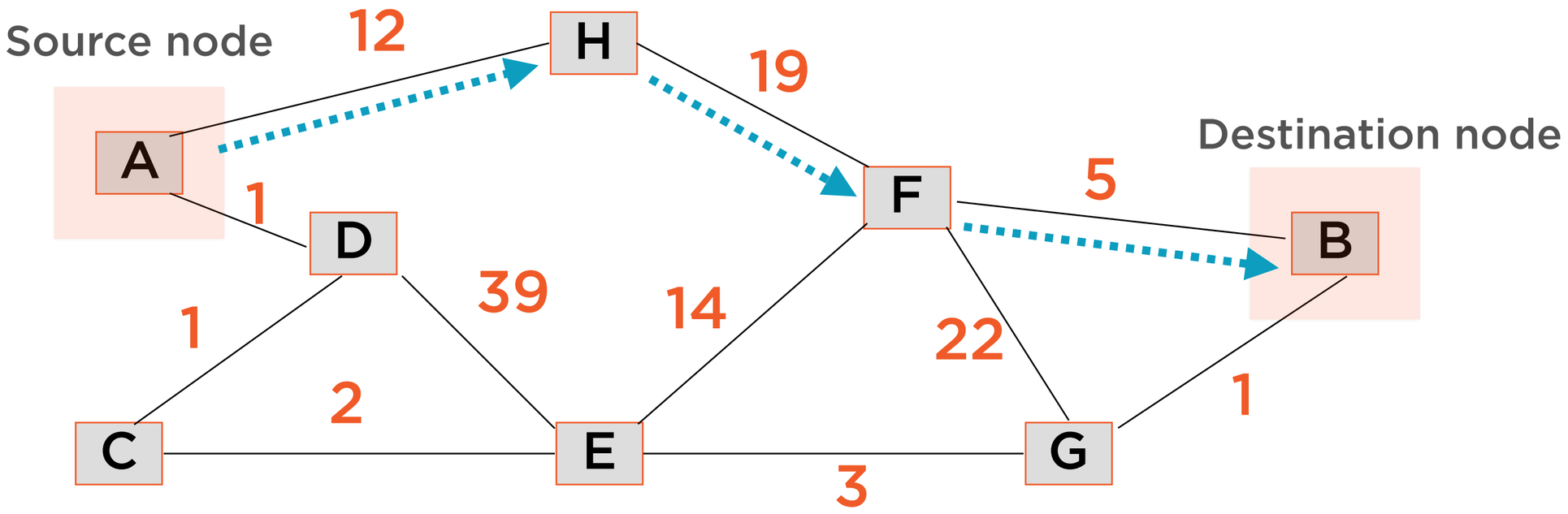
Shortest path minimizes sum of weights of edges

Weighted Graphs



Cost of shortest path = 1 + 1 + 2 + 3 + 1 = 8

Weighted Graphs



Other paths are longer i.e. more expensive

12 + 19 + 5 = 36

In an undirected graph weights represent the cost of traversing the edge in either direction

Shortest Path Algorithms

Unweighted Graphs

All edges have equal weights

Shortest path has smallest number of hops

Unweighted shortest path algorithm

Weighted Graphs

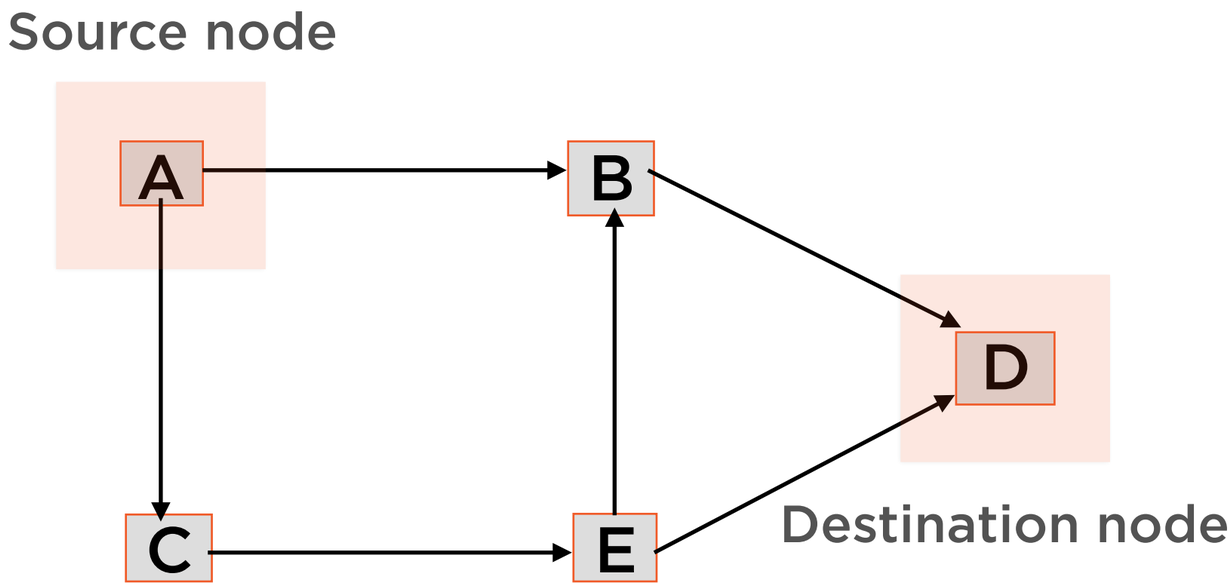
Edges have differing weights

Shortest path has lowest sum of weights along path

Djisktra's algorithm

Unweighted Shortest Path Algorithm

Unweighted Shortest Path Algorithm



Find the shortest path between A and D

All edges have equal weight

Distance Table

Algorithm operates through a data structure called the distance table

Node	Distance	Preceding Node
A		
B		
C		
D		
E		

Distance Table

Contains all nodes in the graph

Node	Distance	Preceding Node
A		
B		
C		
D		
E		

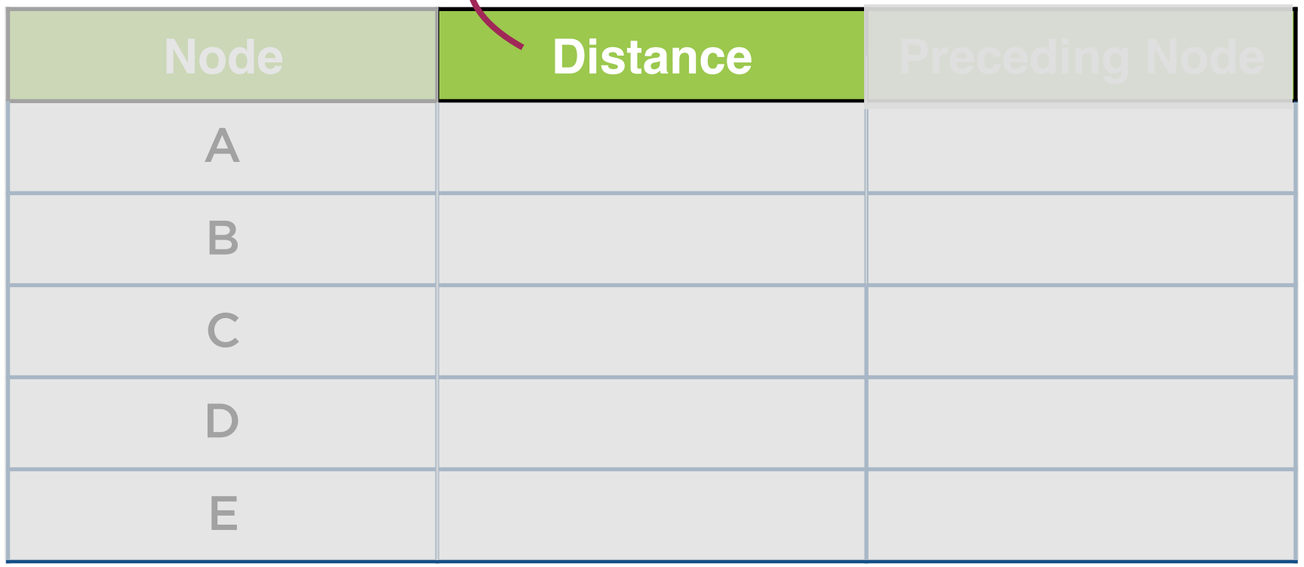
Distance Table

Contains all nodes in the graph

Node	Distance	Preceding Node
A		
B		
C		
D		
E		

Distance Table

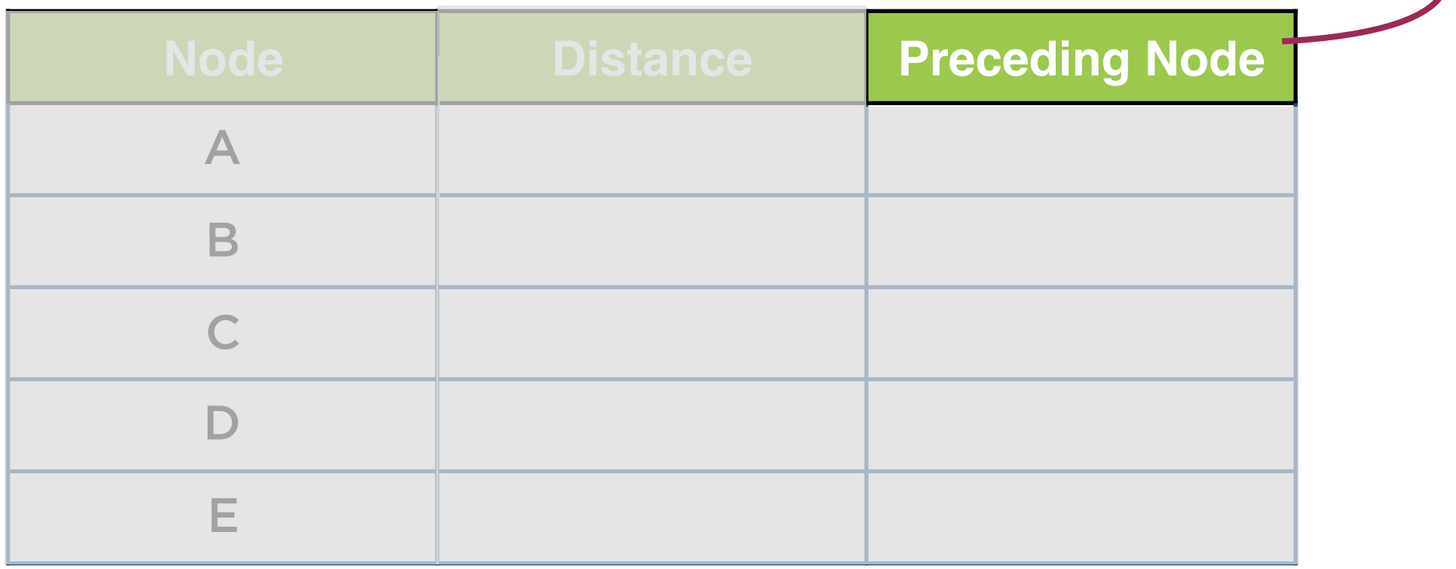
Holds the shortest distance from the source node



Node	Distance	Preceding Node
A		
B		
C		
D		
E		

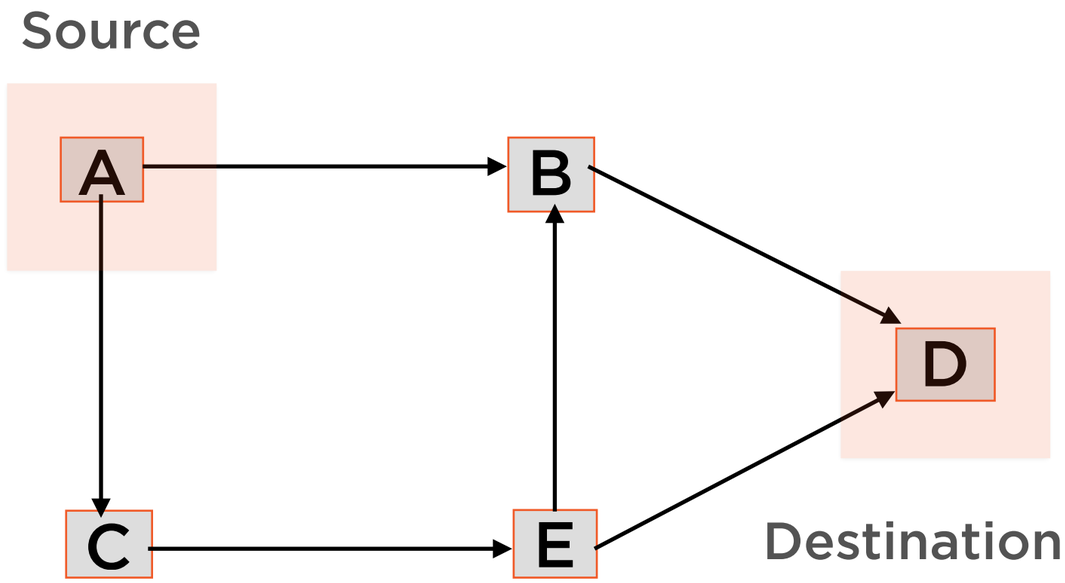
Distance Table

Holds the preceding node in the shortest path from source node to that particular node



Node	Distance	Preceding Node
A		
B		
C		
D		
E		

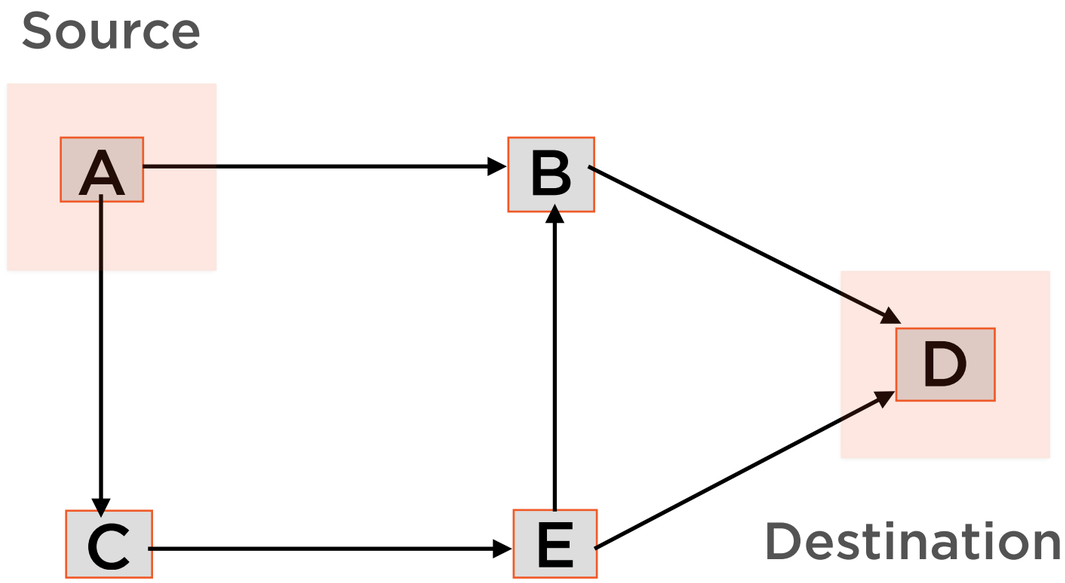
Initial Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	-1	-
C	-1	-
D	-1	-
E	-1	-

At outset, all we know is that source node is at distance 0 from itself

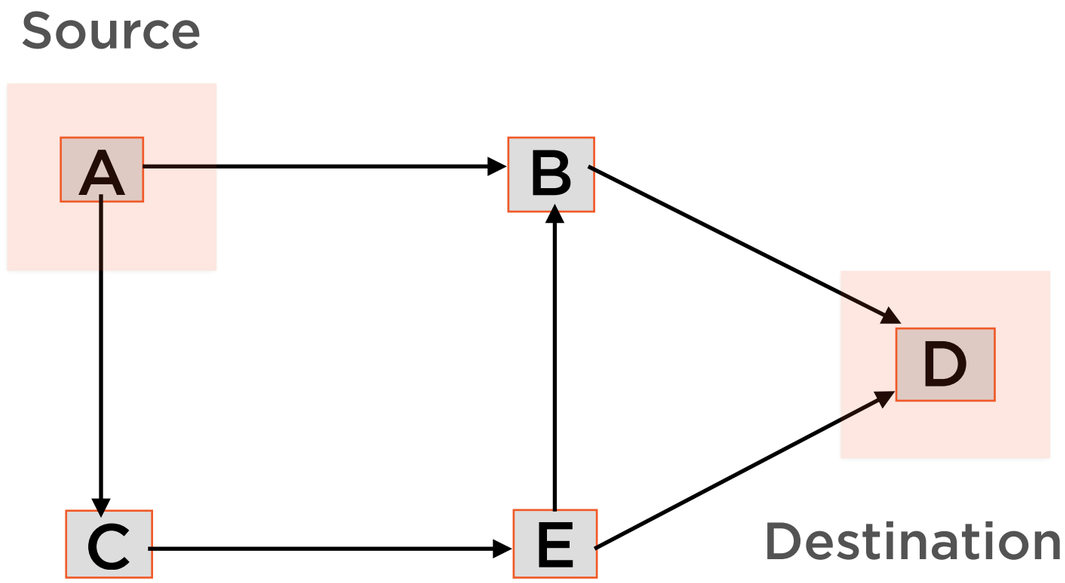
Final Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

By end of procedure, we have a fully populated distance table (more in a bit)

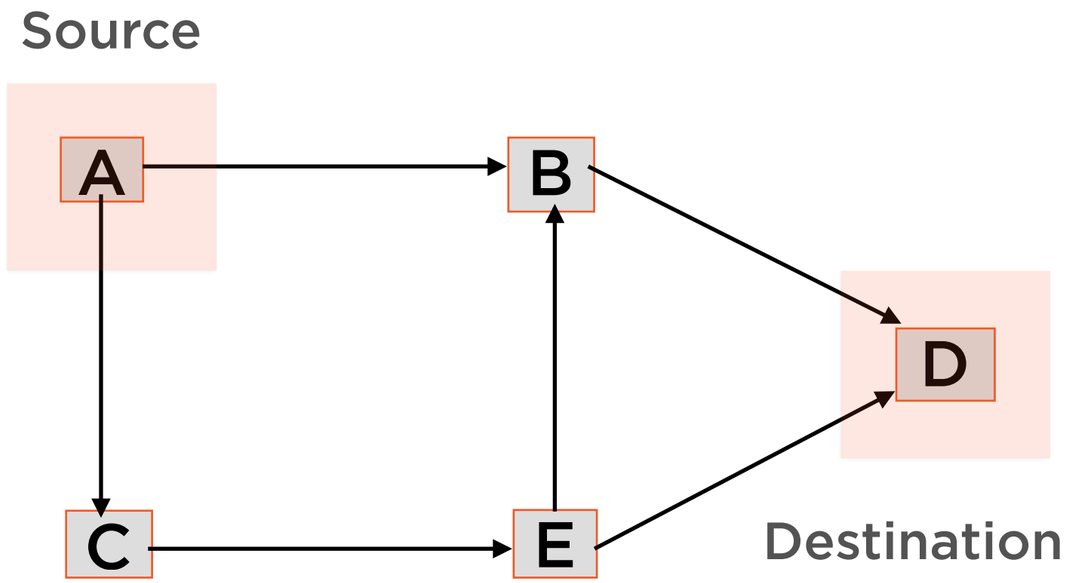
Final Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

By end of procedure, we have a fully populated distance table (more in a bit)

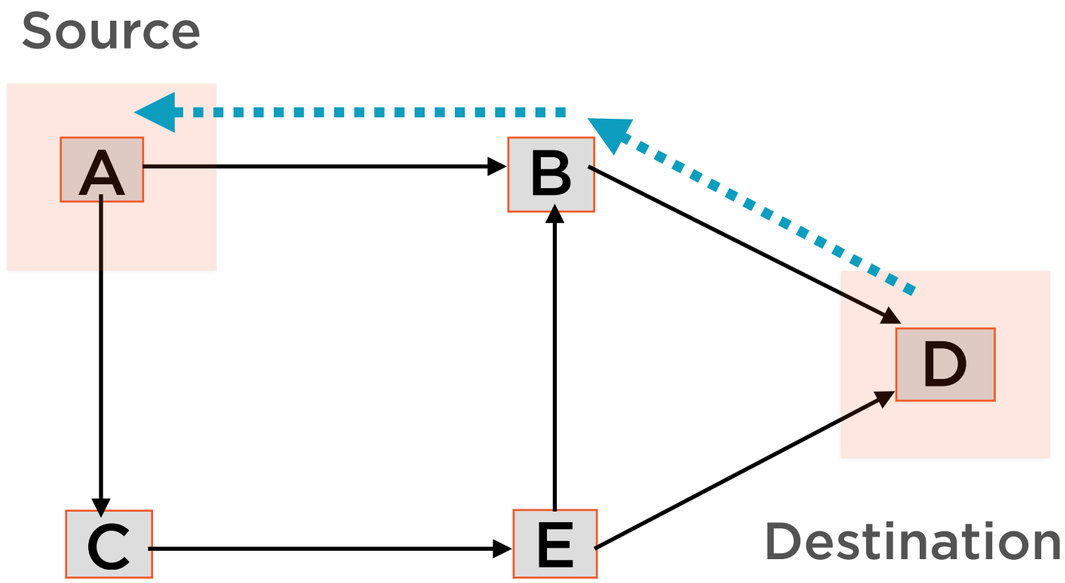
Final Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

By end of procedure, we have a fully populated distance table (more in a bit)

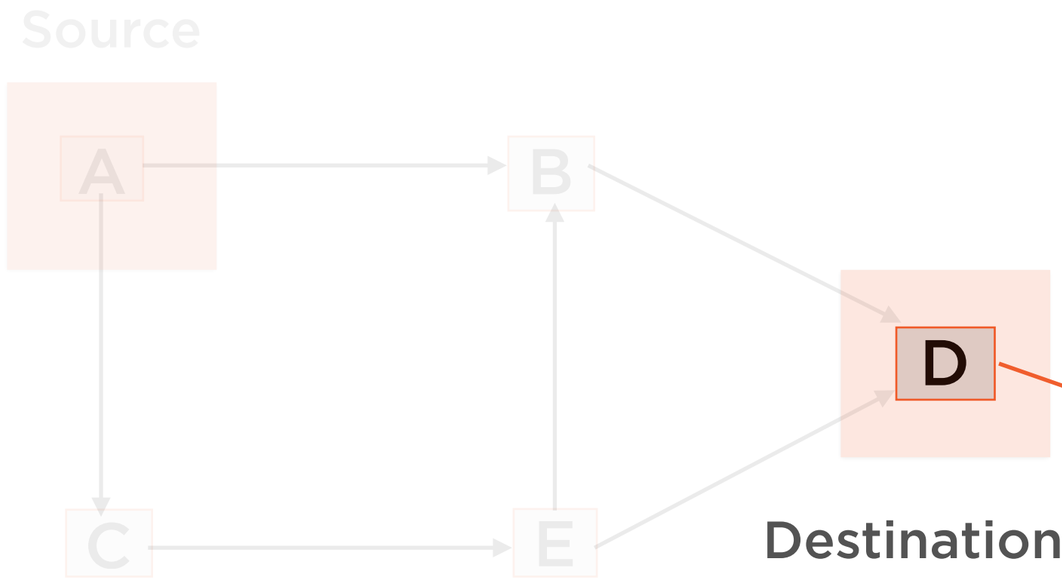
Finding Shortest Path



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

To trace out the shortest path, backtrack from destination D to source A

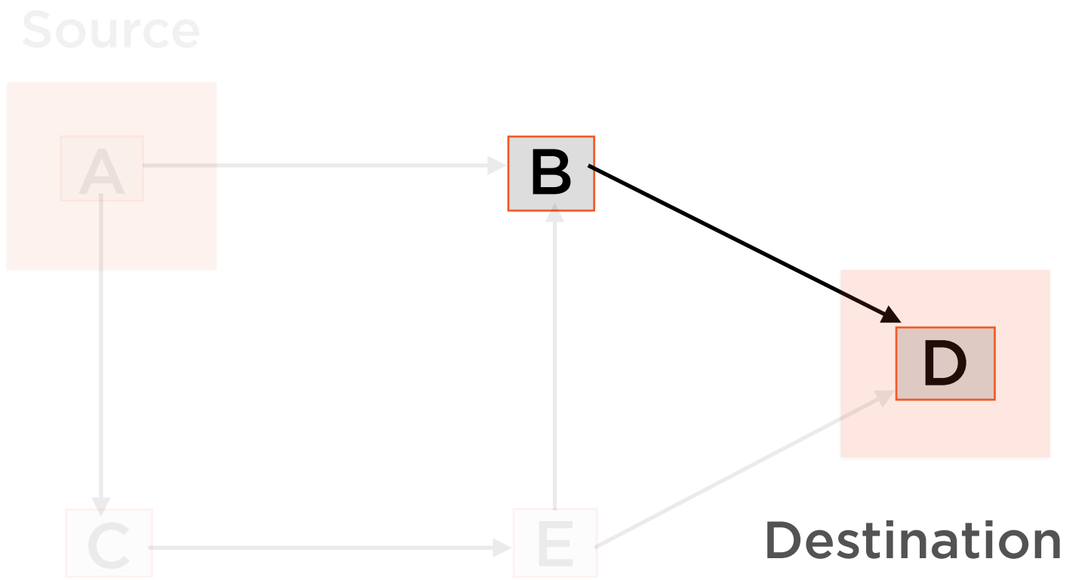
Backtracking



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	**2**	**B**
E	2	C

Shortest Path **D**

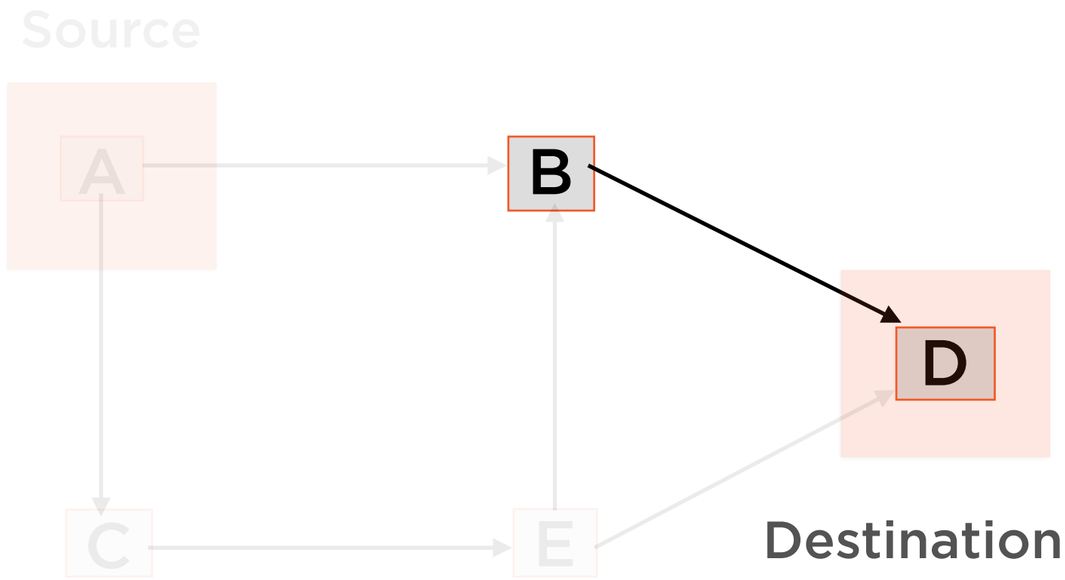
Backtracking



Node	Distance	Preceding Node
A	0	A
B	**1**	**A**
C	1	A
D	2	B
E	2	C

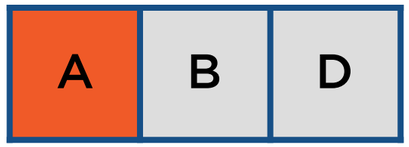
Shortest Path

Backtracking

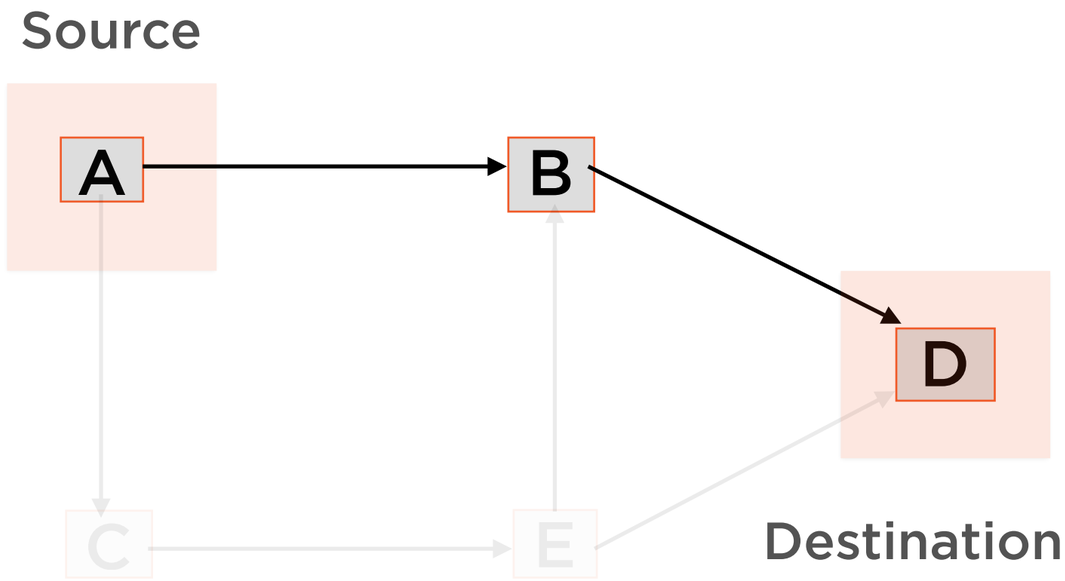


Node	Distance	Preceding Node
A	0	A
B	**1**	**A**
C	1	A
D	2	B
E	2	C

Shortest Path

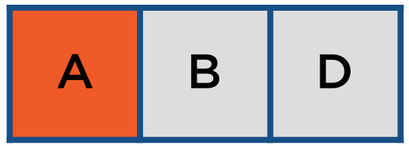


Backtracking

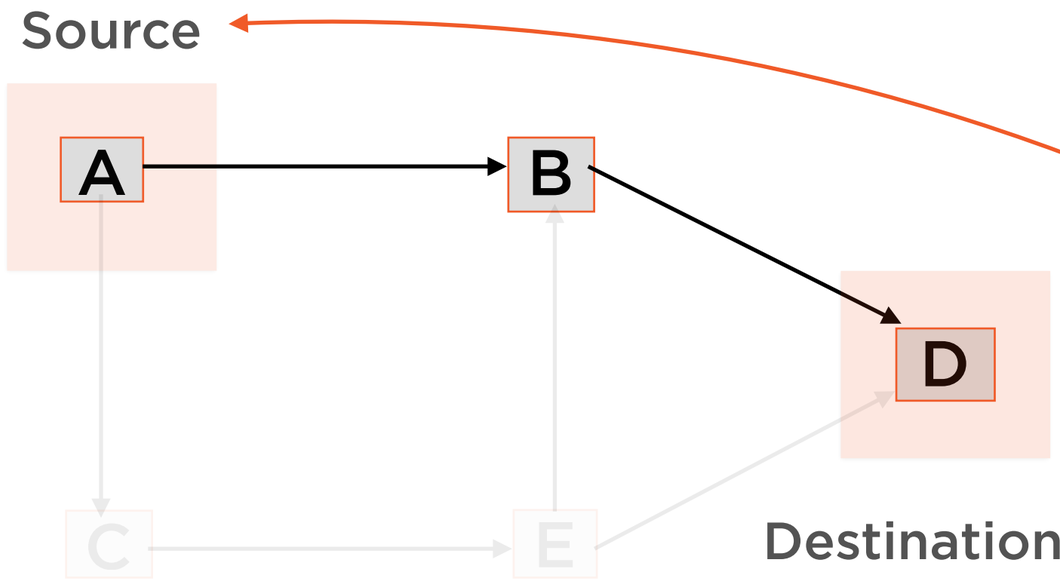


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Shortest Path

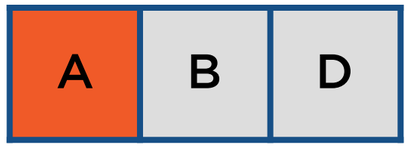


Backtracking

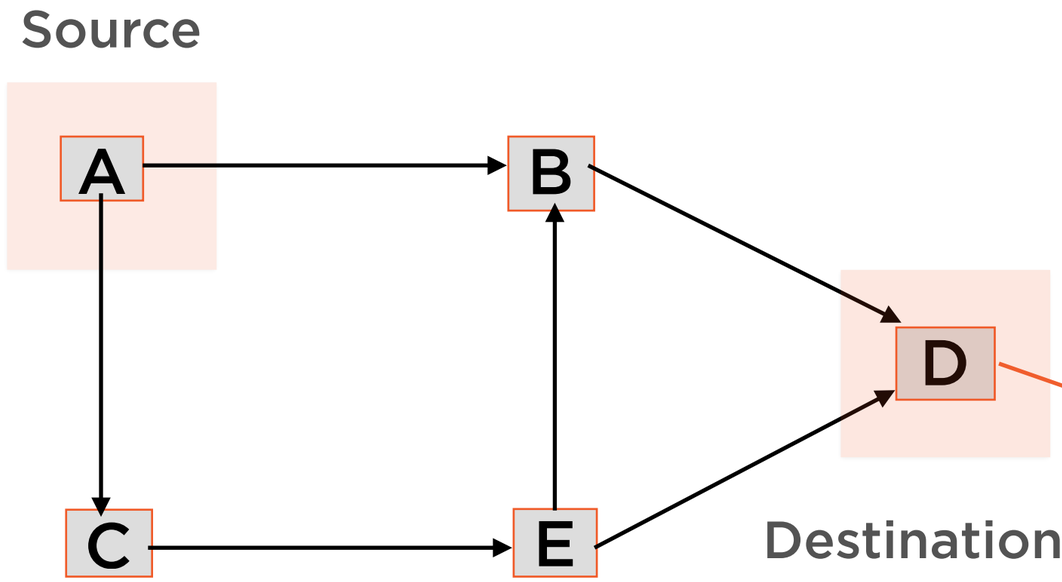


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Shortest Path

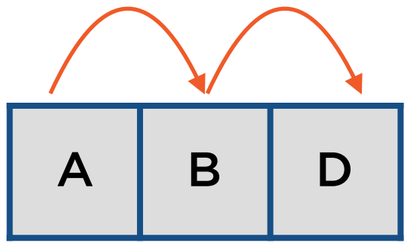


Backtracking



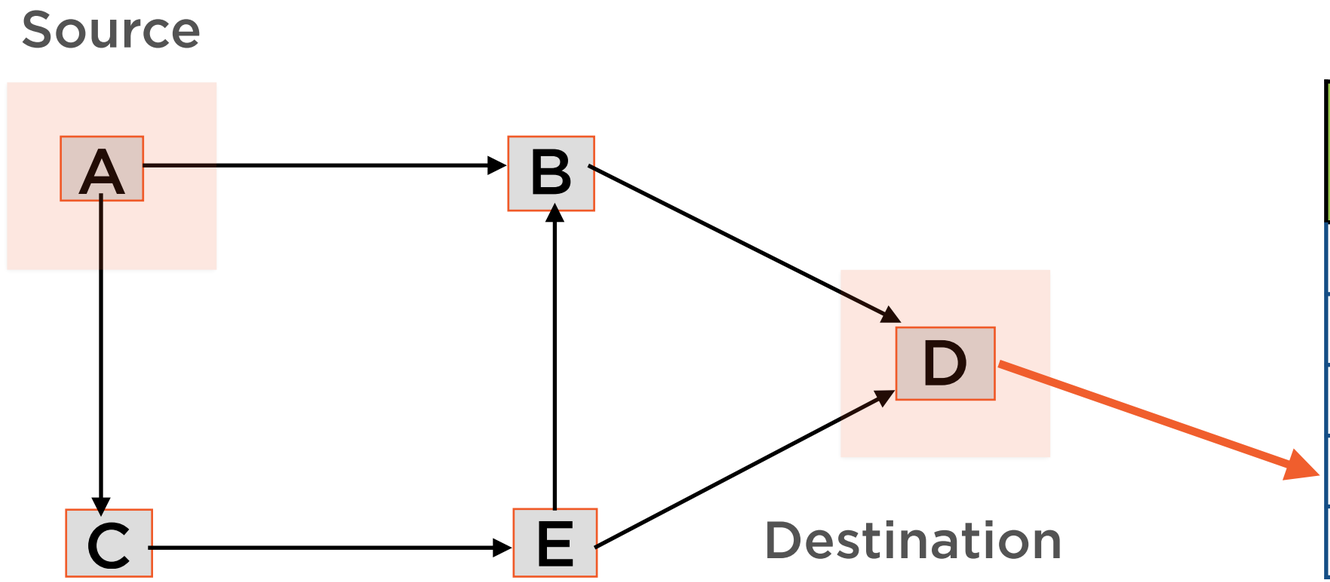
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Shortest Path



Cost of shortest path = number of hops = 2

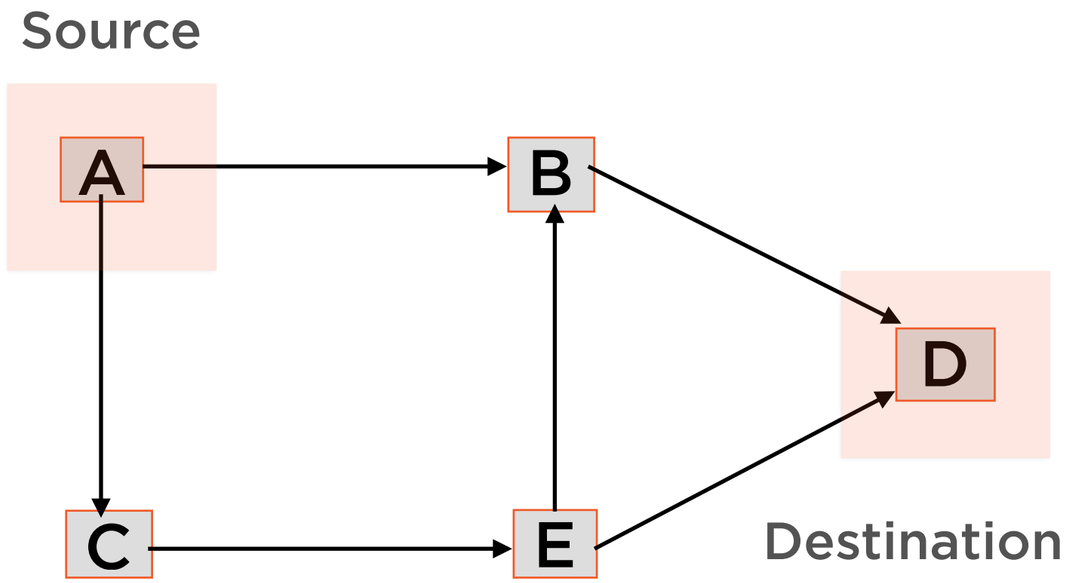
Backtracking



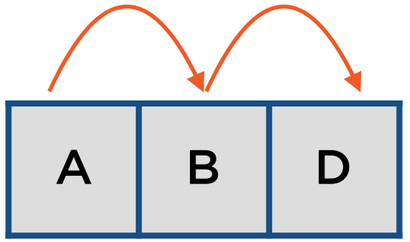
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Notice how we “walk back” the distance table to construct the shortest path

Backtracking

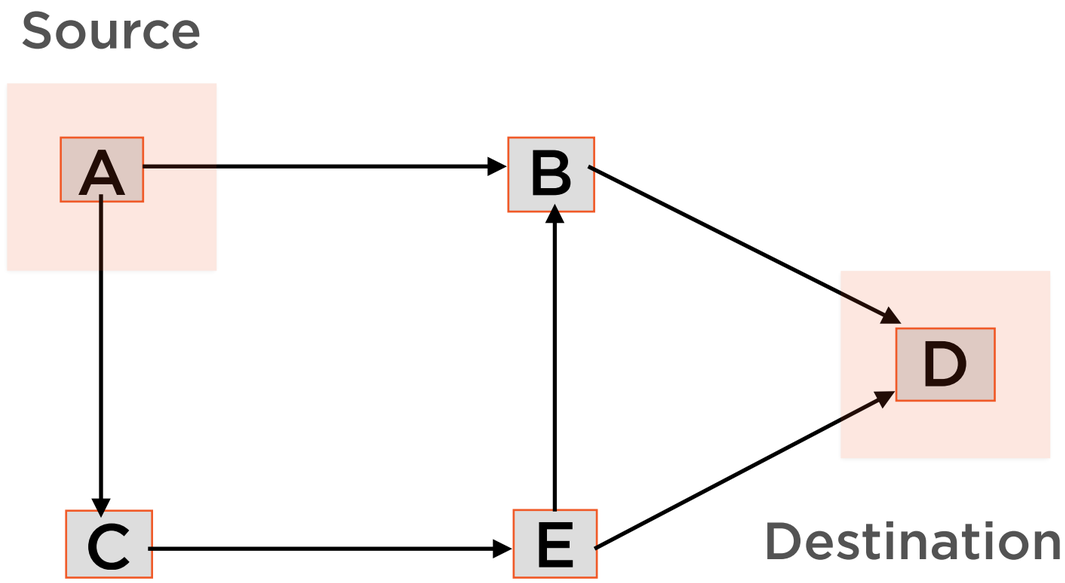


“Last-In-First-Out” => Use a stack



Building the Distance Table

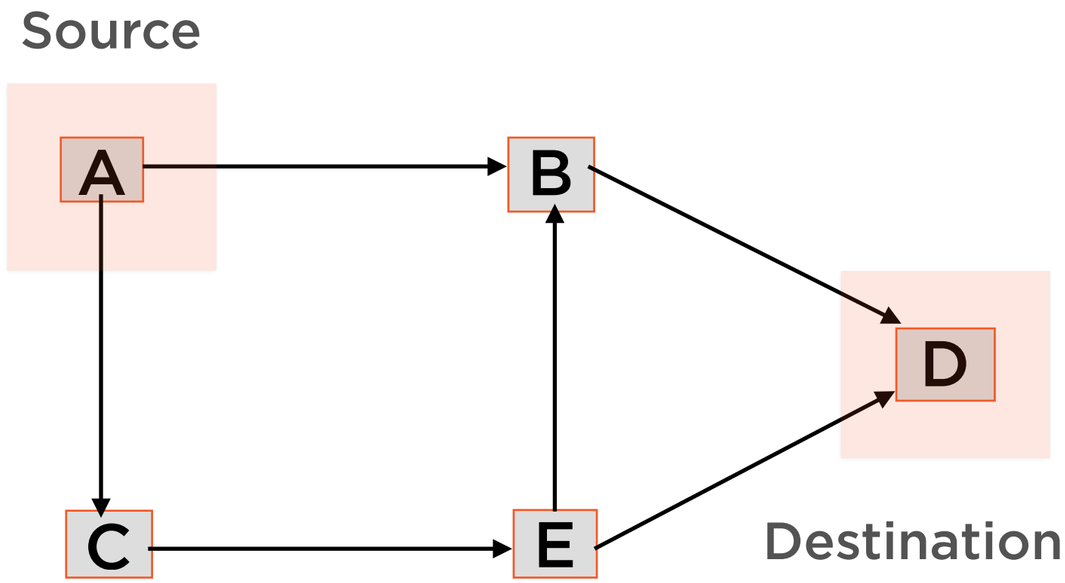
Final Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Where does the fully populated distance table come from?

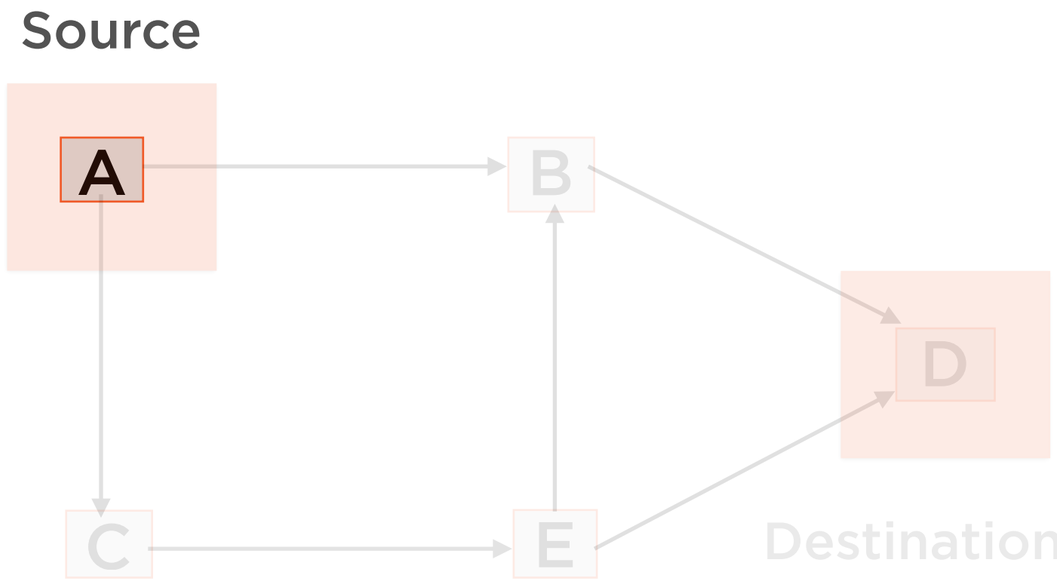
Initial Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	-1	-
C	-1	-
D	-1	-
E	-1	-

At outset, all we know is that source node is at distance 0 from itself

Process Node A

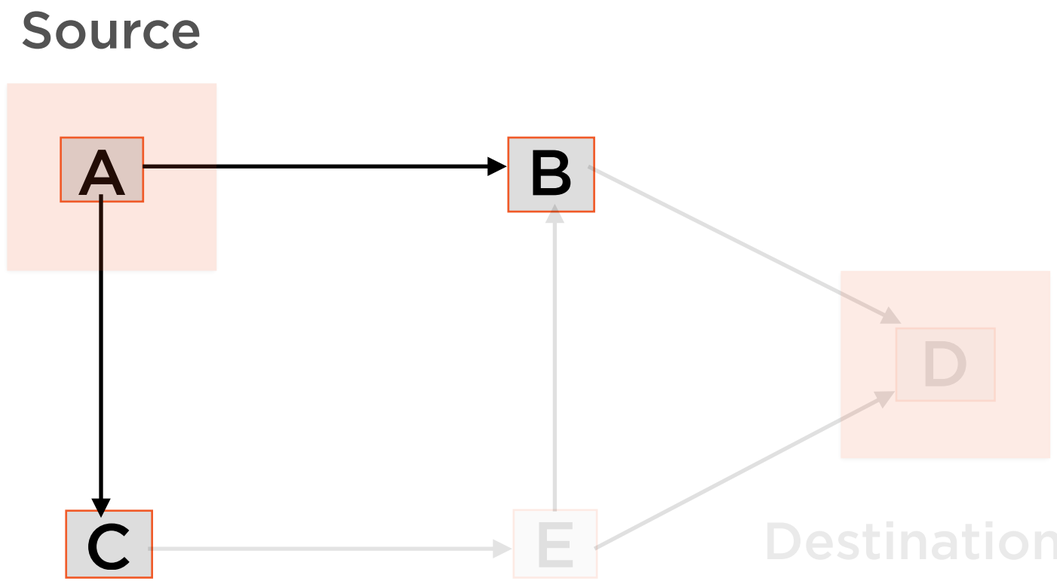


Node	Distance	Preceding Node
A	0	A
B	-1	-
C	-1	-
D	-1	-
E	-1	-

Start at the source, initialize a queue of nodes

Processing Queue

Process Node A

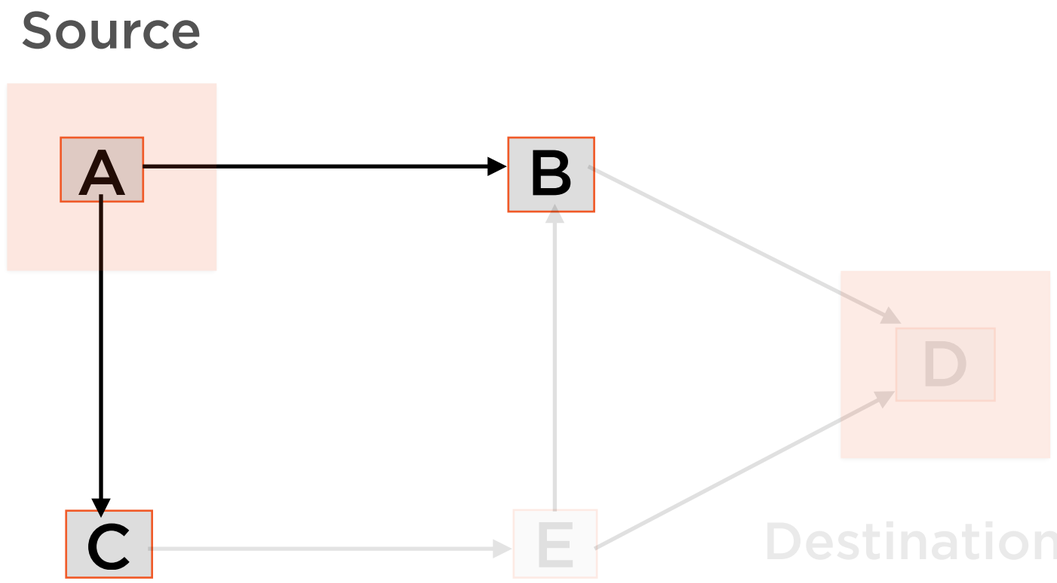


Node	Distance	Preceding Node
A	0	A
B	-1	-
C	-1	-
D	-1	-
E	-1	-

Add immediate neighbors to queue

Processing Queue

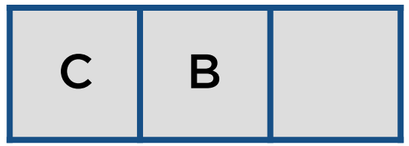
Process Node A



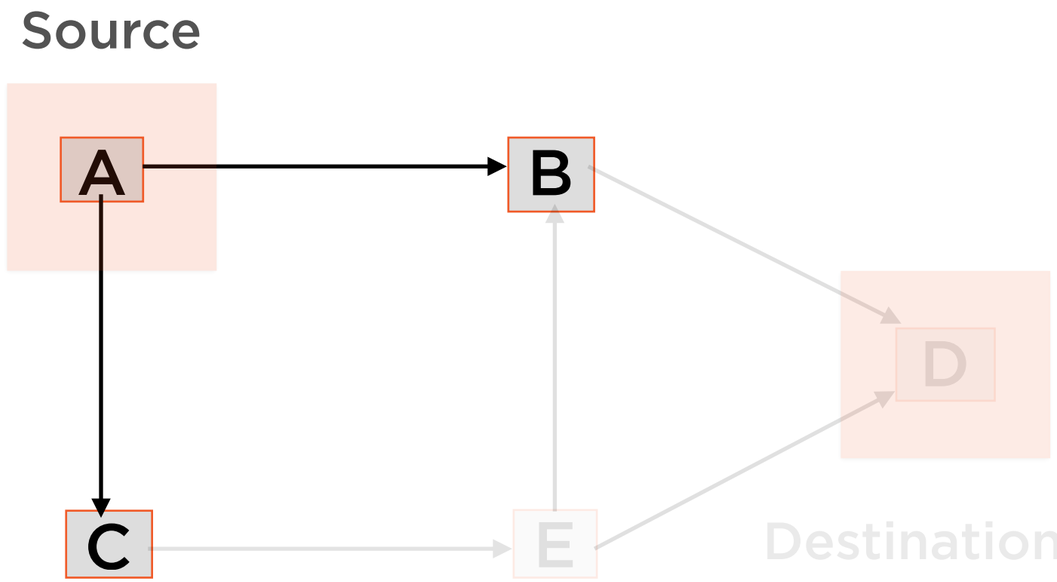
Node	Distance	Preceding Node
A	0	A
B	-1	-
C	-1	-
D	-1	-
E	-1	-

Update distance table for those immediate neighbors

Processing Queue

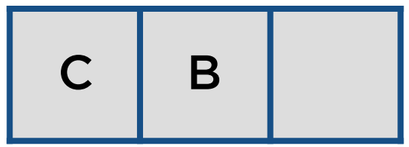


Process Node A

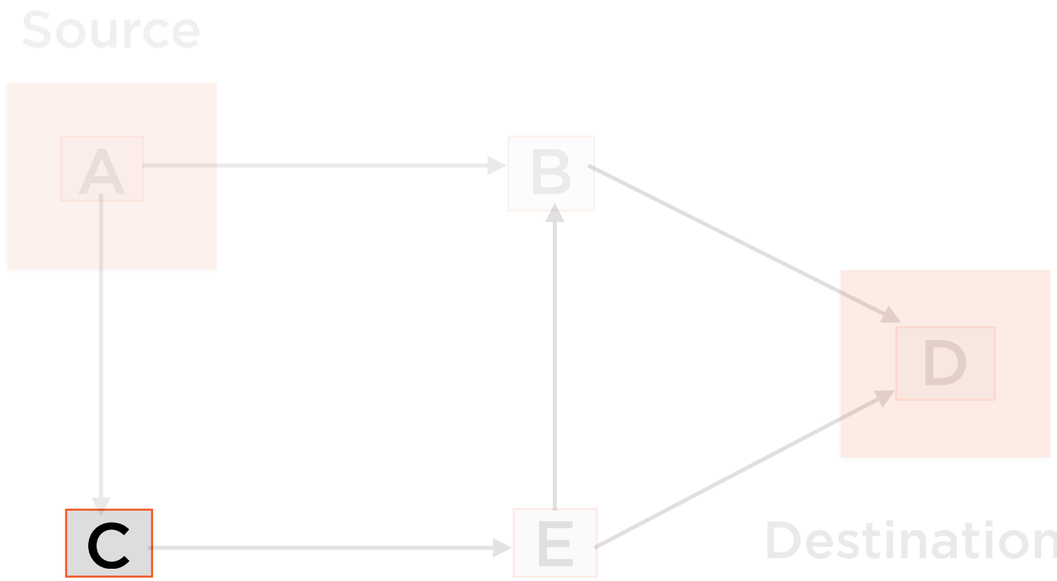


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Processing Queue



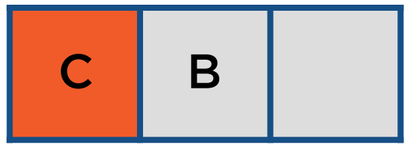
Process Node C



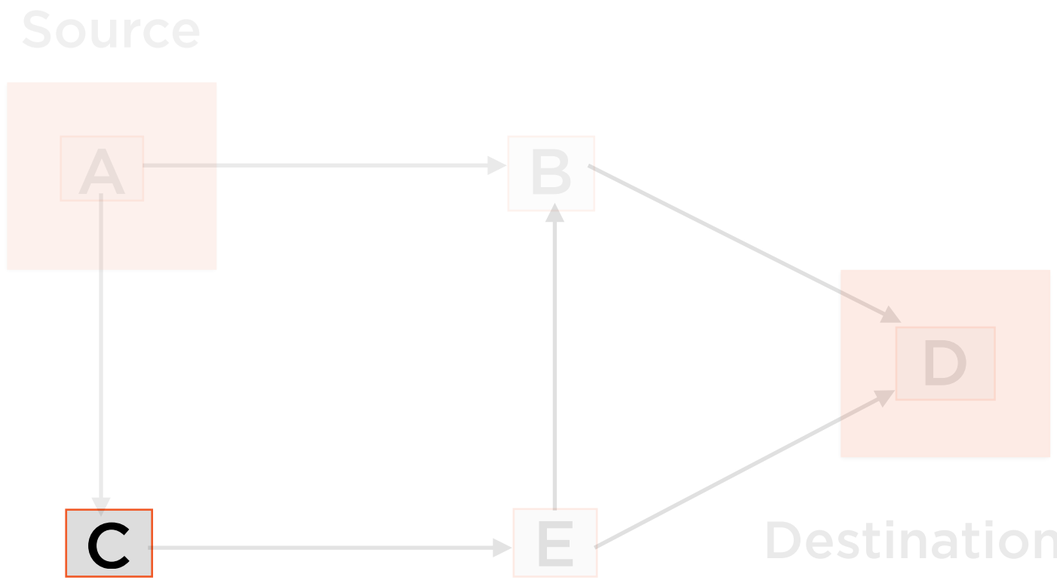
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Dequeue C and process it

Processing Queue



Process Node C

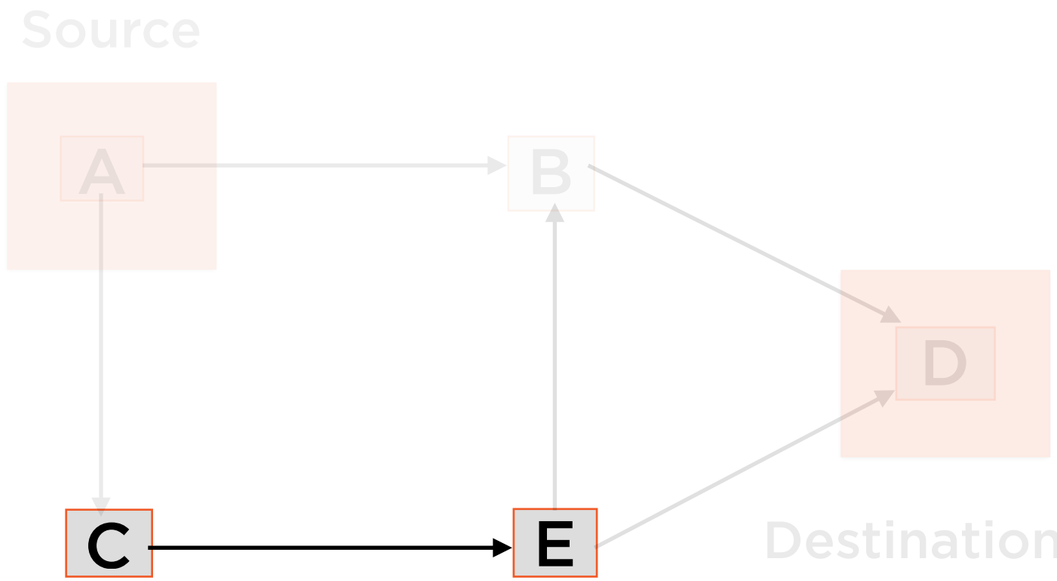


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Dequeue C and process it

Processing Queue

Process Node C

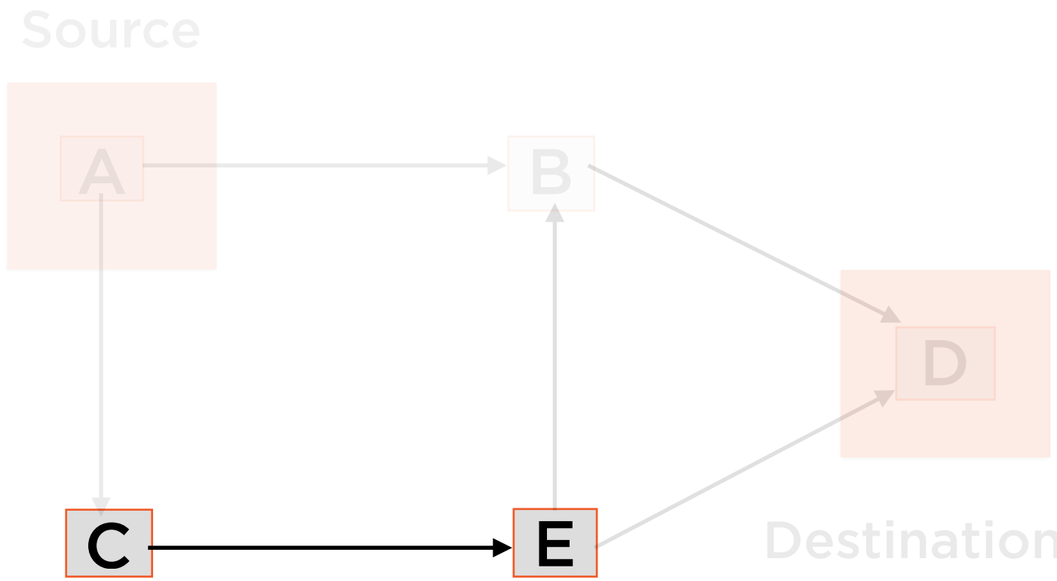


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Add immediate neighbors to queue

Processing Queue

Process Node C

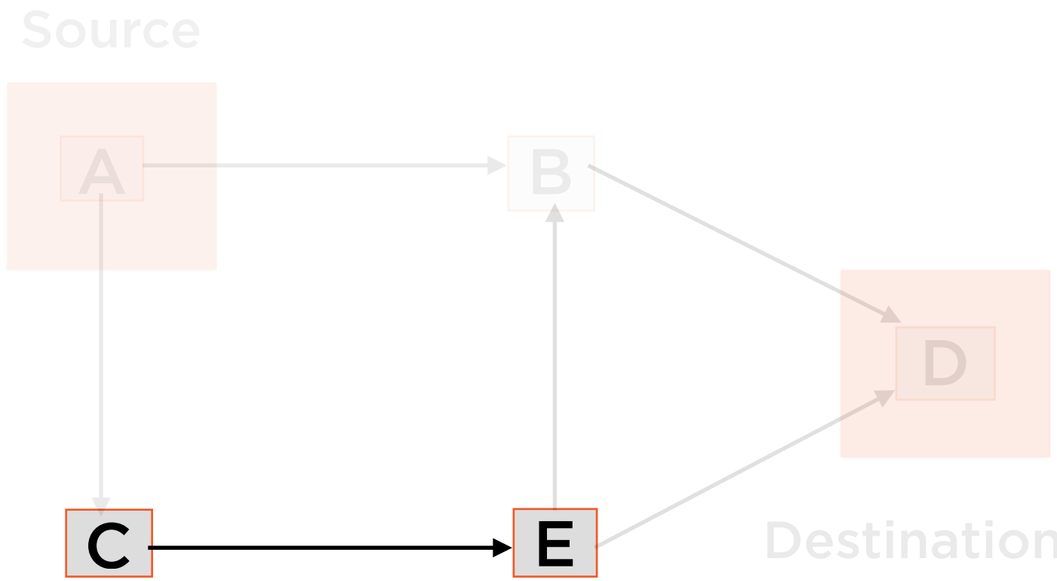


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Update distance table for those immediate neighbors

Processing Queue

Process Node C

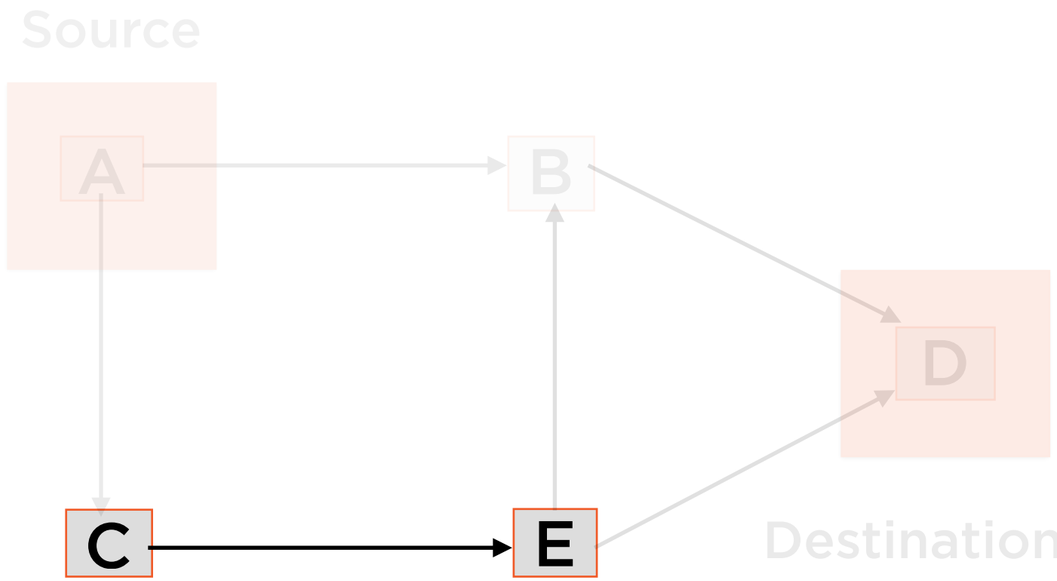


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	-1	-

Now, E is 1 hop from C, and C is 1 hop from A

Processing Queue

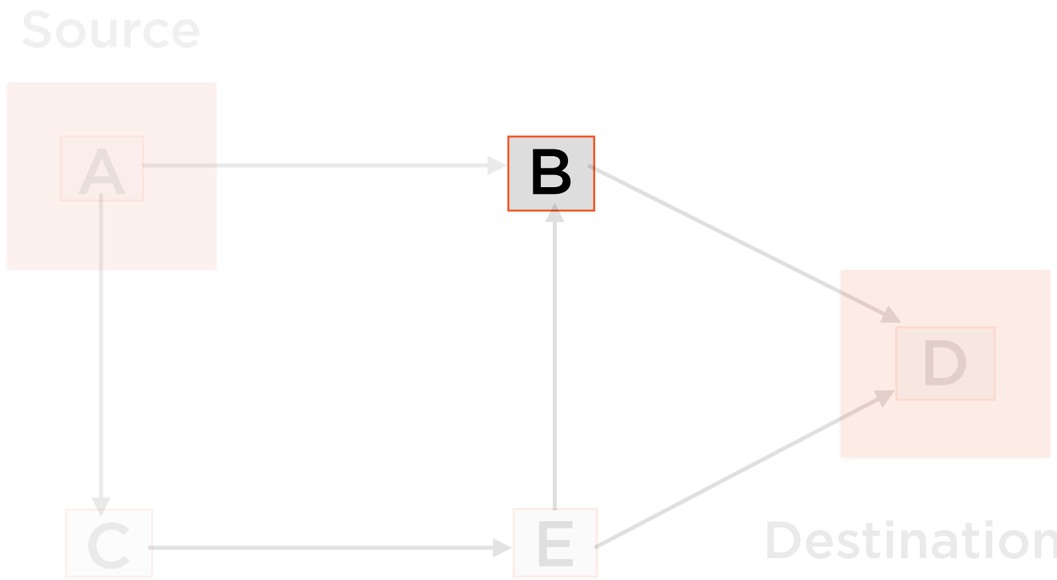
Process Node C



Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Processing Queue

Process Node B

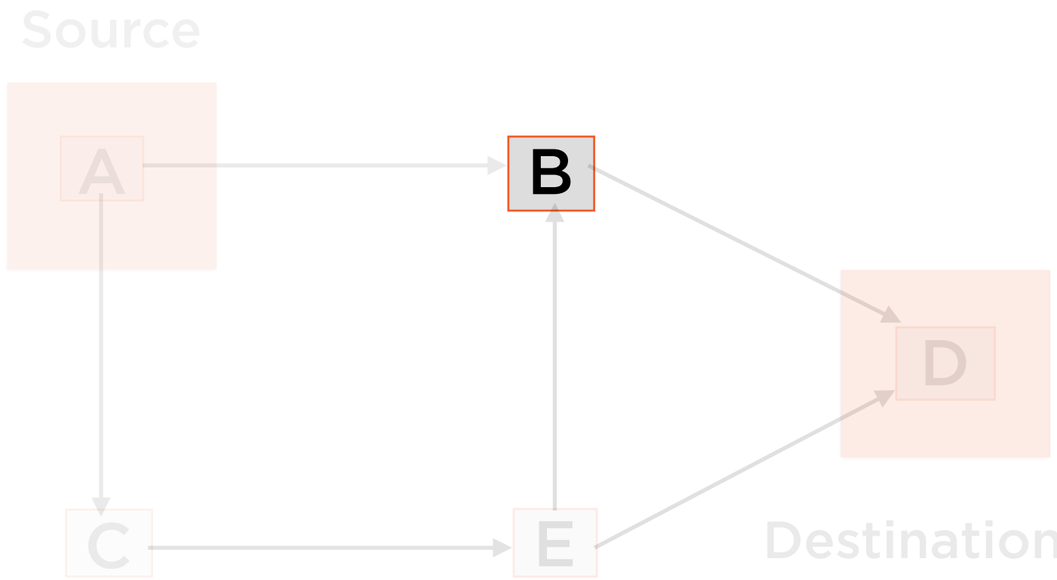


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Dequeue B and process it

Processing Queue

Process Node B

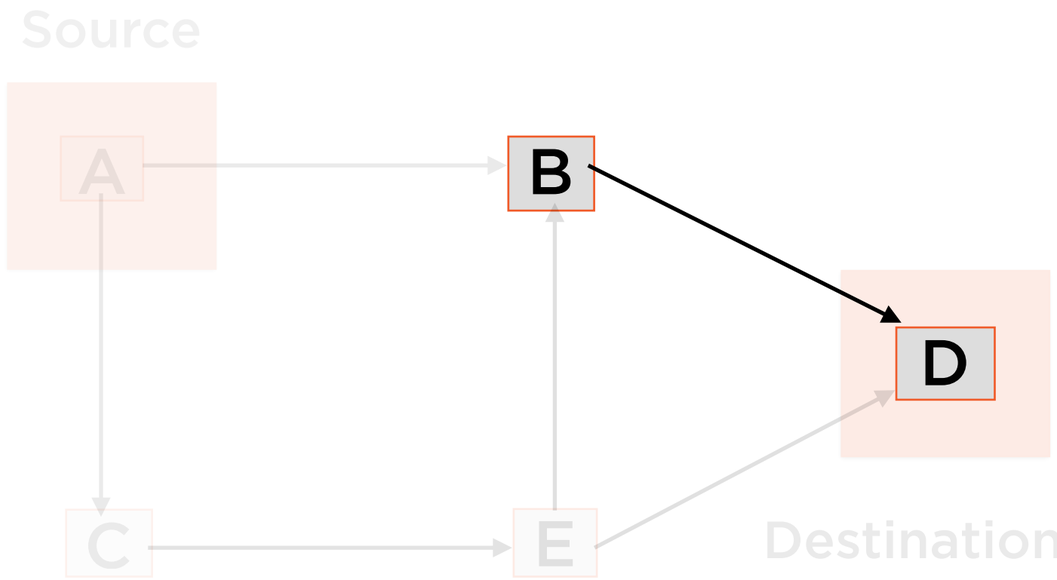


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Dequeue B and process it

Processing Queue

Process Node B

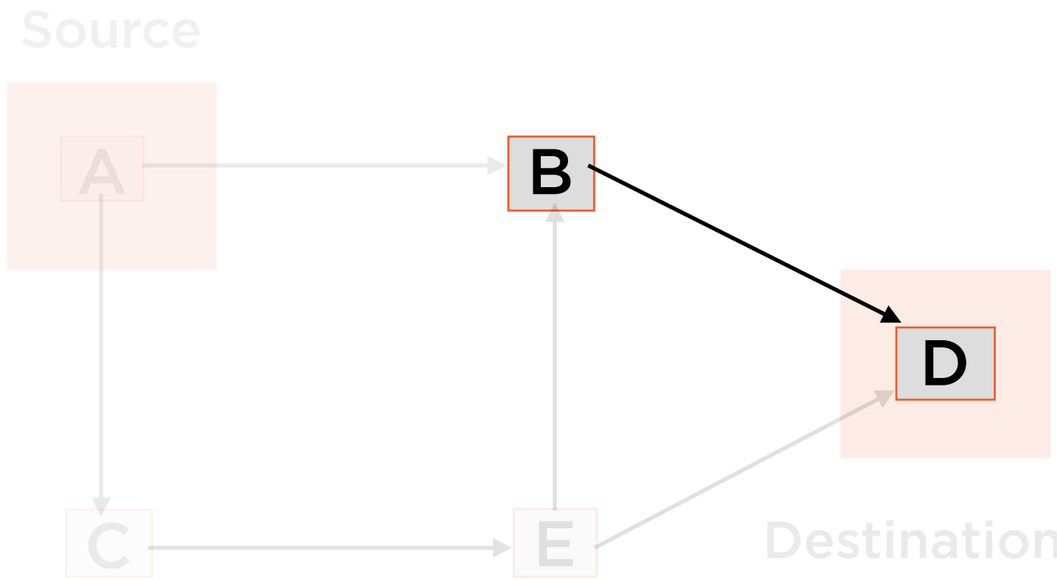


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Add immediate neighbors to queue

Processing Queue

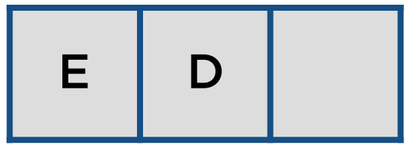
Process Node B



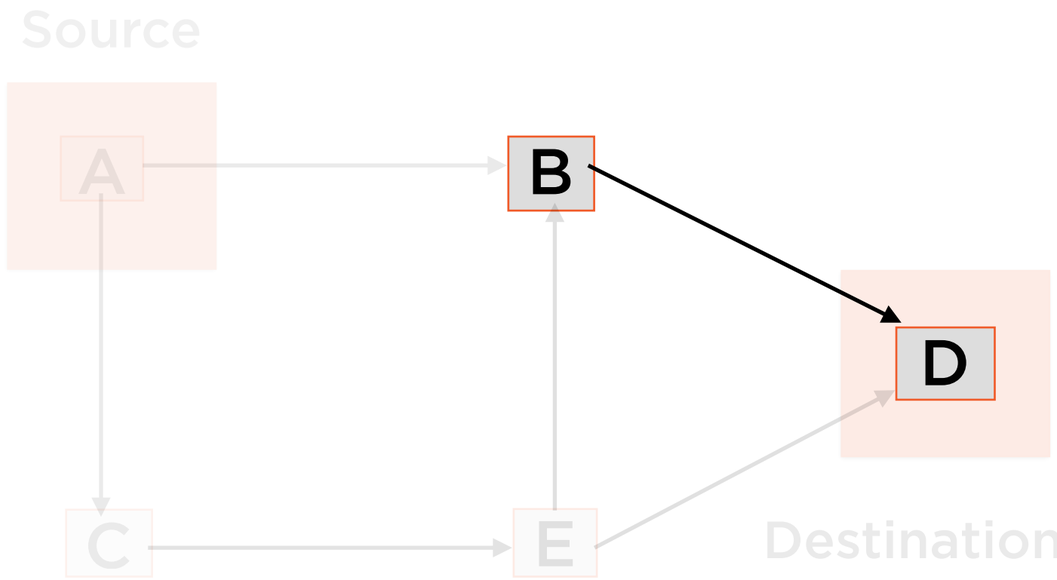
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Update distance table for those immediate neighbors

Processing Queue



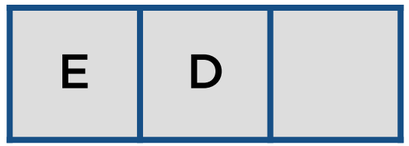
Process Node B



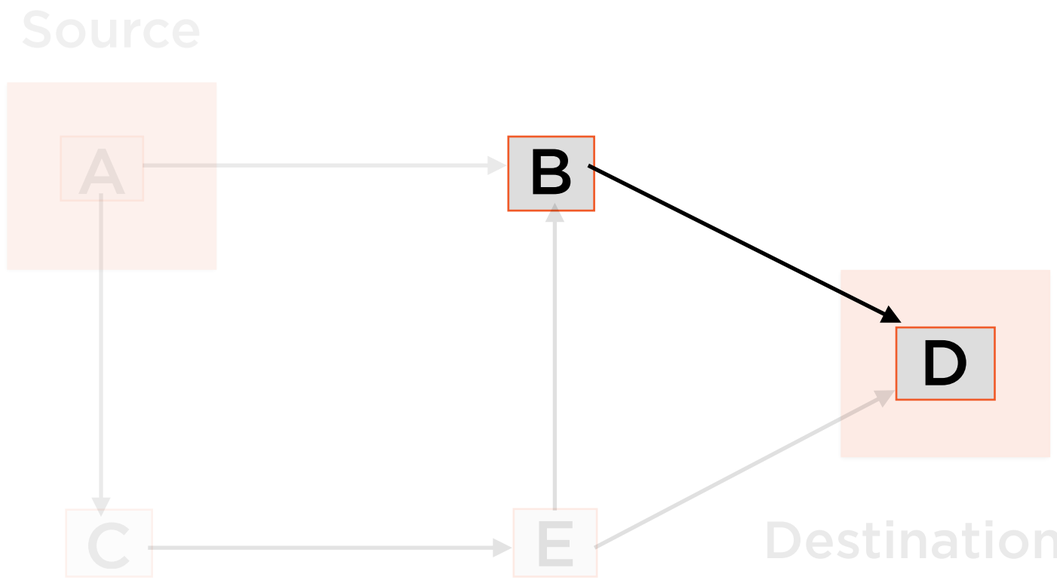
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	-1	-
E	2	C

Now, D is 1 hop from B, and B is 1 hop from A

Processing Queue

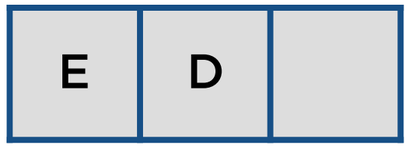


Process Node B

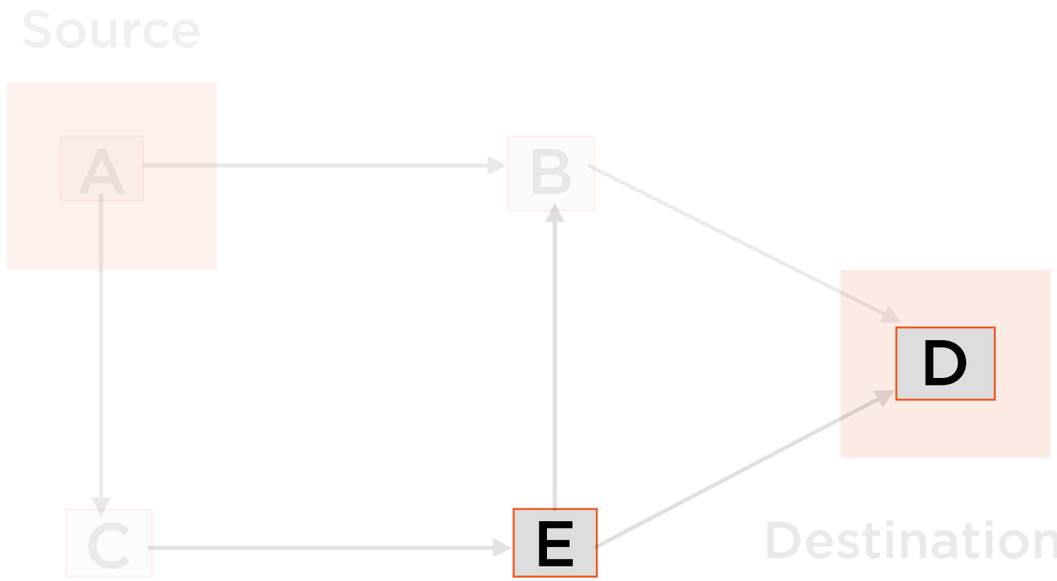


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Processing Queue



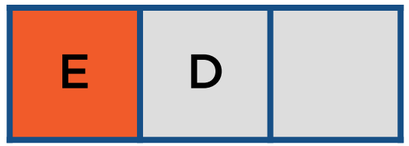
Process Node E



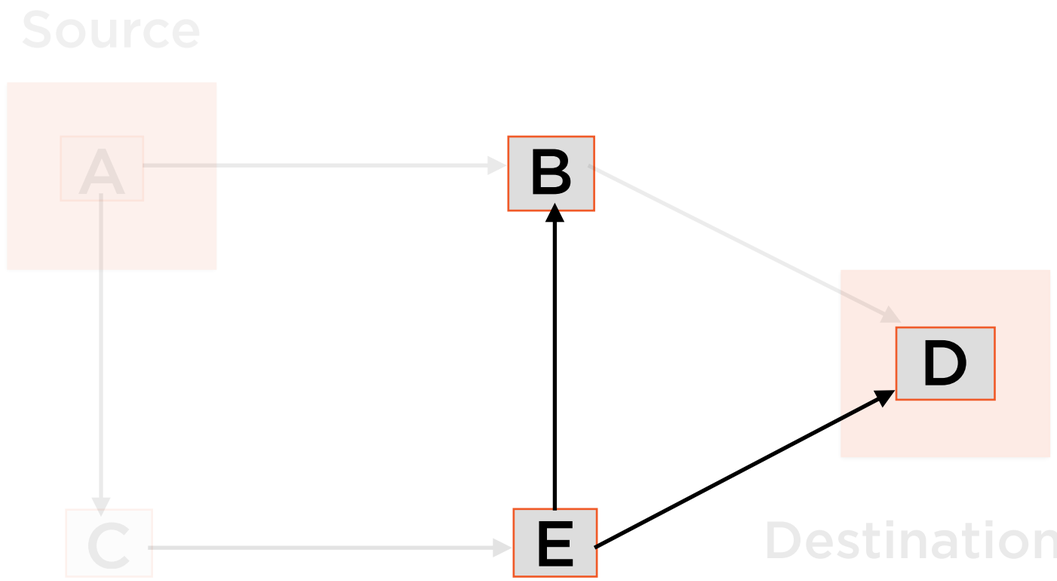
Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Dequeue E and process it

Processing Queue



Process Node E

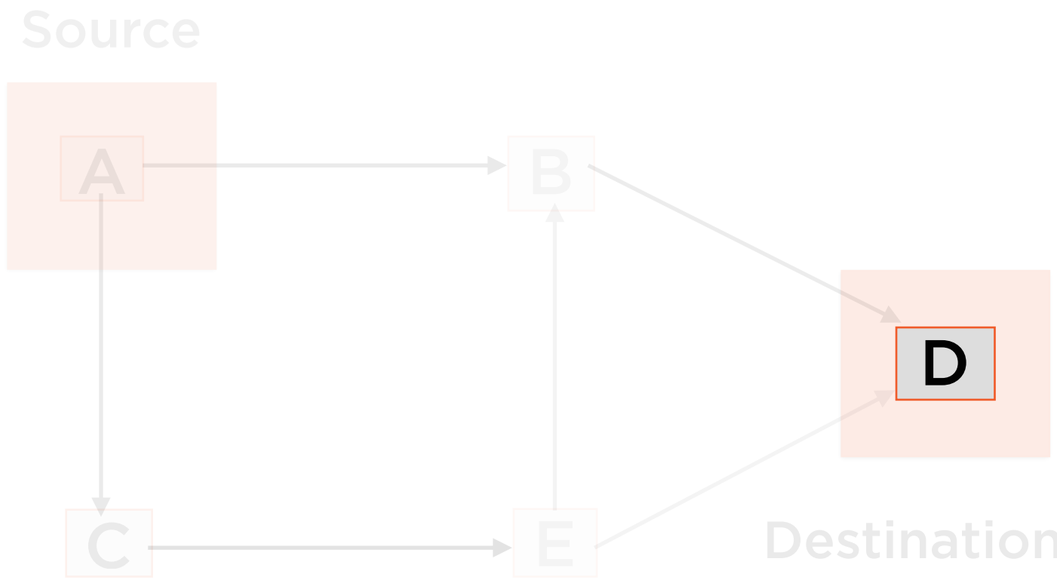


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Immediate neighbors already covered - no processing needed

Processing Queue

Process Node D

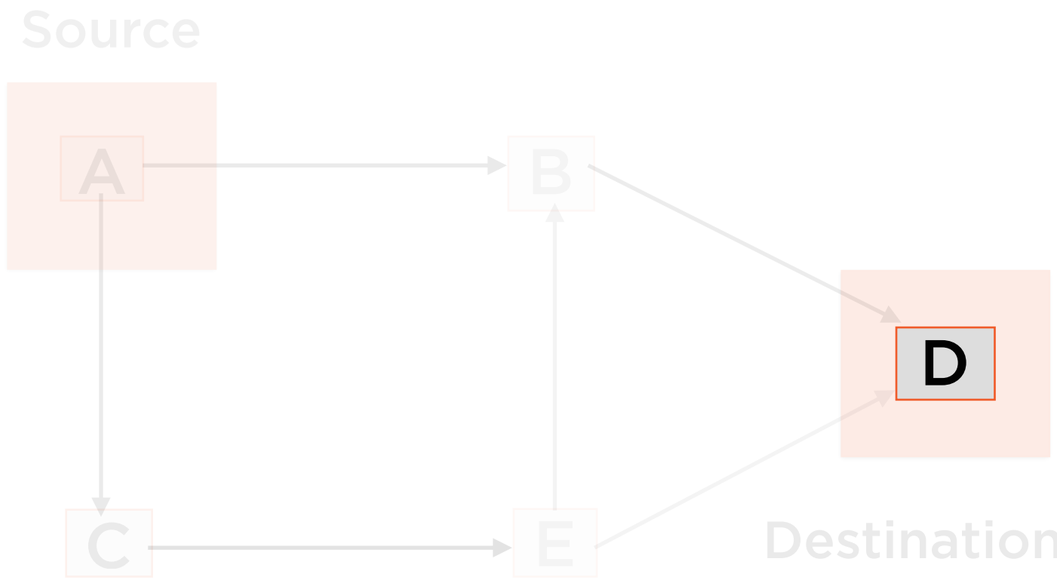


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

Dequeue D and process it

Processing Queue

Process Node D

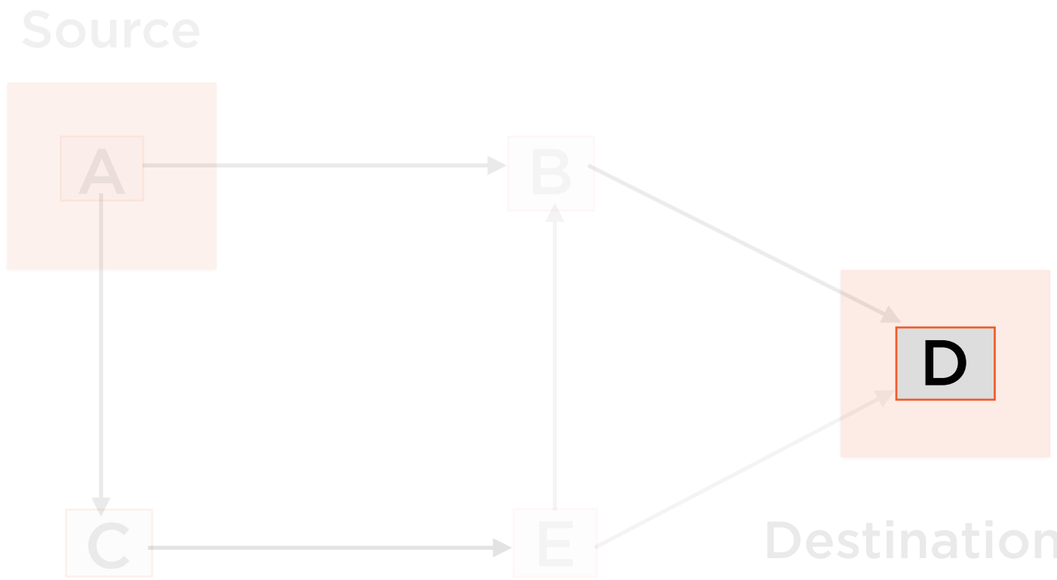


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

No immediate neighbors - no processing needed

Processing Queue

Process Node D

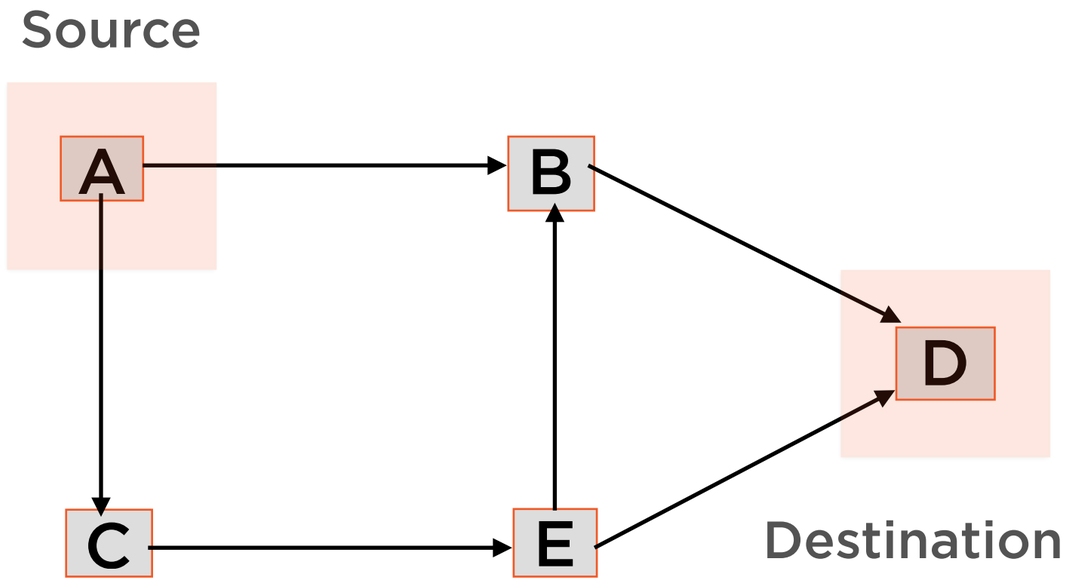


Node	Distance	Preceding Node
A	0	A
B	1	A
C	1	A
D	2	B
E	2	C

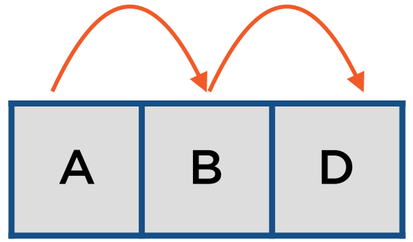
Processing queue empty - algorithm complete

Processing Queue

Backtracking



“Last-In-First-Out” => Use a stack



If stack unwind does not end at source node, no path exists

Unweighted Shortest Path Algorithm

Data	Data Structure
Distance table	3-column array
Backtracking	Stack

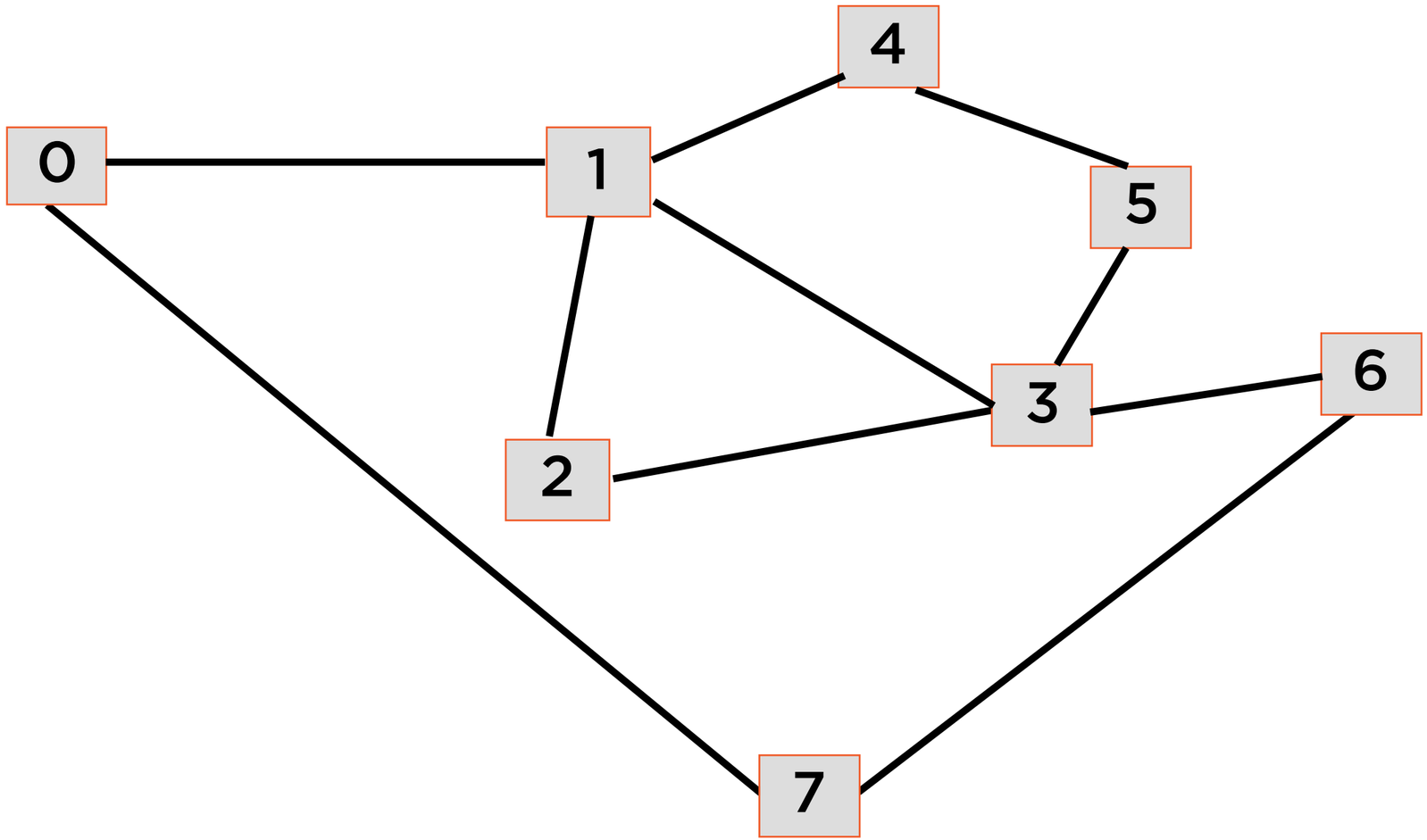
Unweighted Shortest Path Algorithm

Graph Representation	Running Time
Adjacency matrix	$O(V^2)$
Adjacency list or set	$O(V+E)$

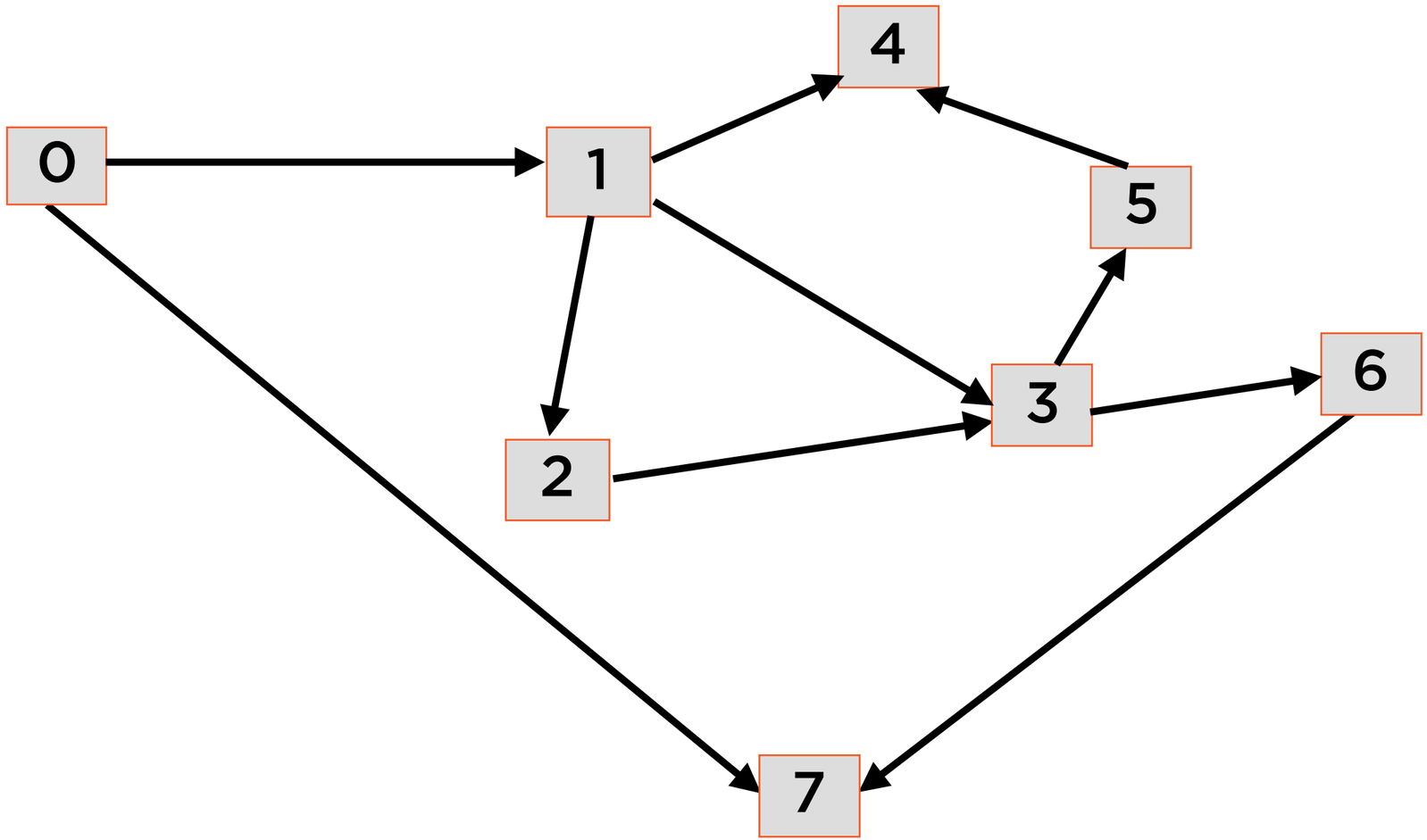
Demo

Calculate the shortest path for unweighted graphs

A Sample Undirected Graph

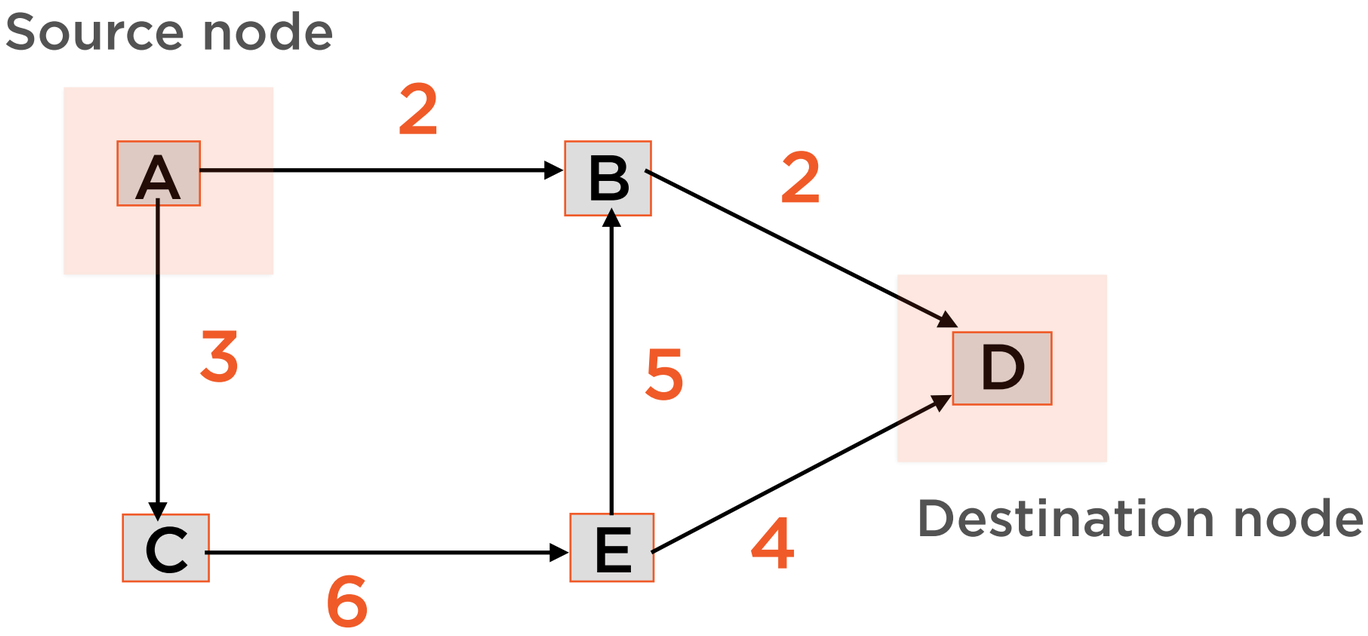


A Sample Directed Graph



Dijsktra's Algorithm

Dijkstra's Algorithm



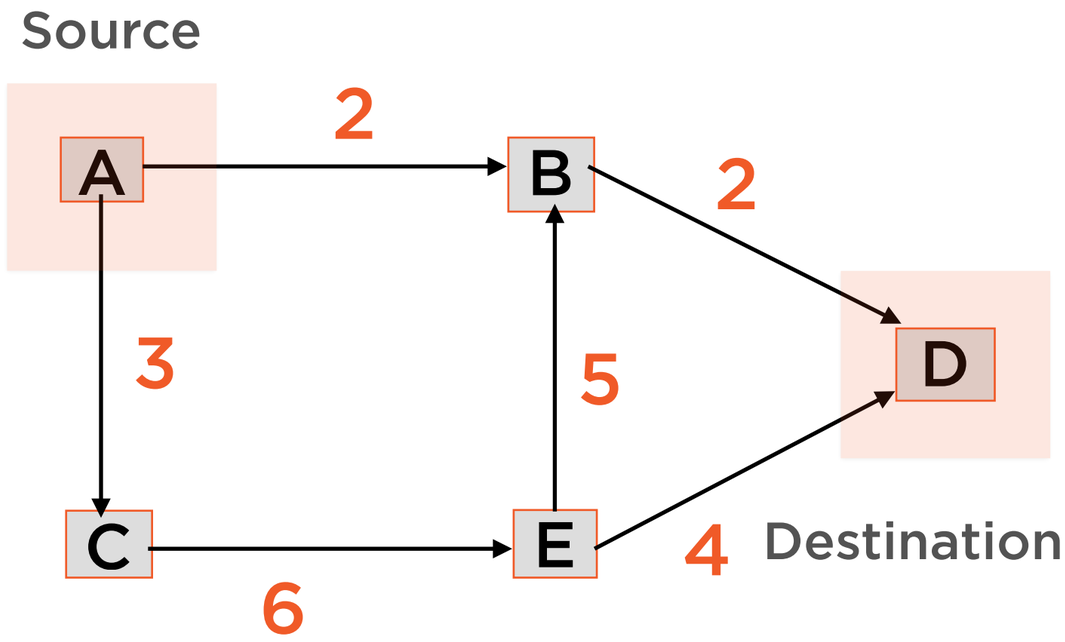
Find the shortest path between A and D

Edges have unequal weights

Shortest Path Algorithms

	Unweighted Shortest Path Algorithm	Djikstra's Algorithm
Enqueuing neighbors	Any order	Decreasing order of weight
Calculating distance	Number of hops	Sum of weights
Visited nodes	Don't update distance to visited nodes	Re-calculate distance to visited nodes, update if needed
Enqueuing visited nodes	Never re-enqueue visited nodes	Re-enqueue if distance was updated

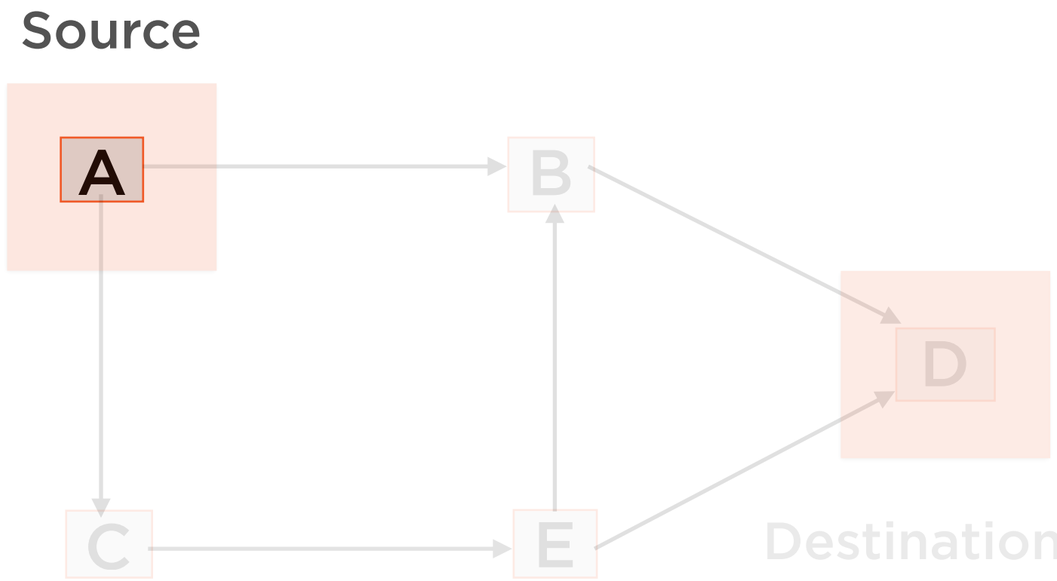
Initial Values in Distance Table



Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

At outset, all we know is that source node is at distance 0 from itself

Process Node A

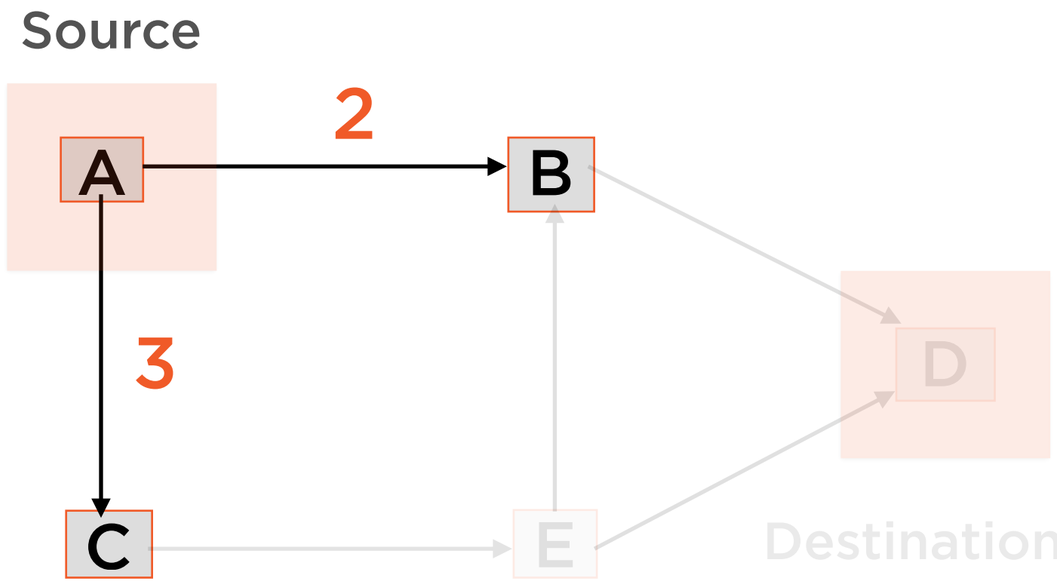


Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

Start at the origin, initialise a queue of nodes to be processed

Processing Queue

Process Node A

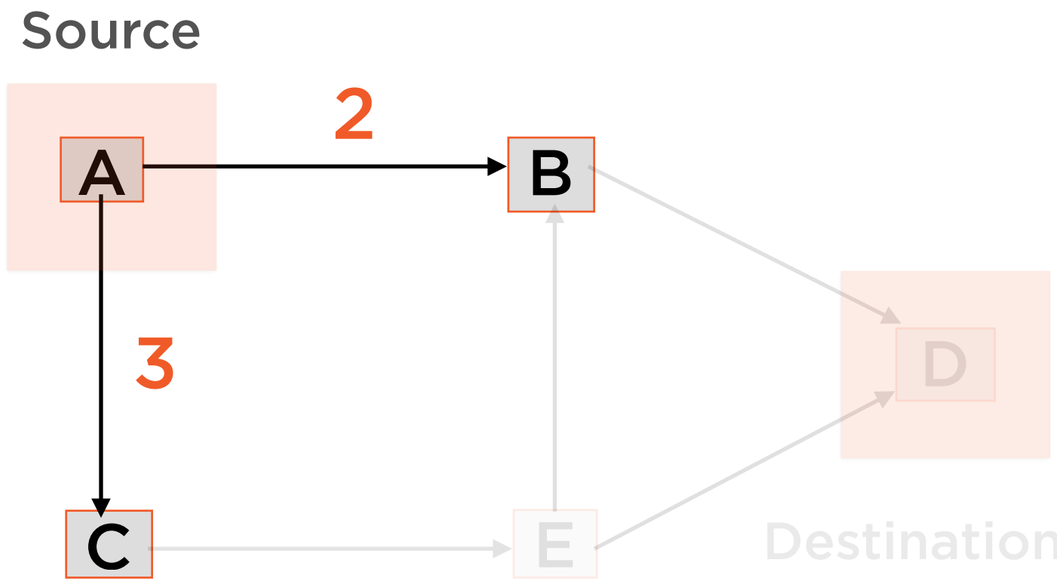


Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

Enqueue immediate neighbors in decreasing order of distance

Processing Queue

Process Node A

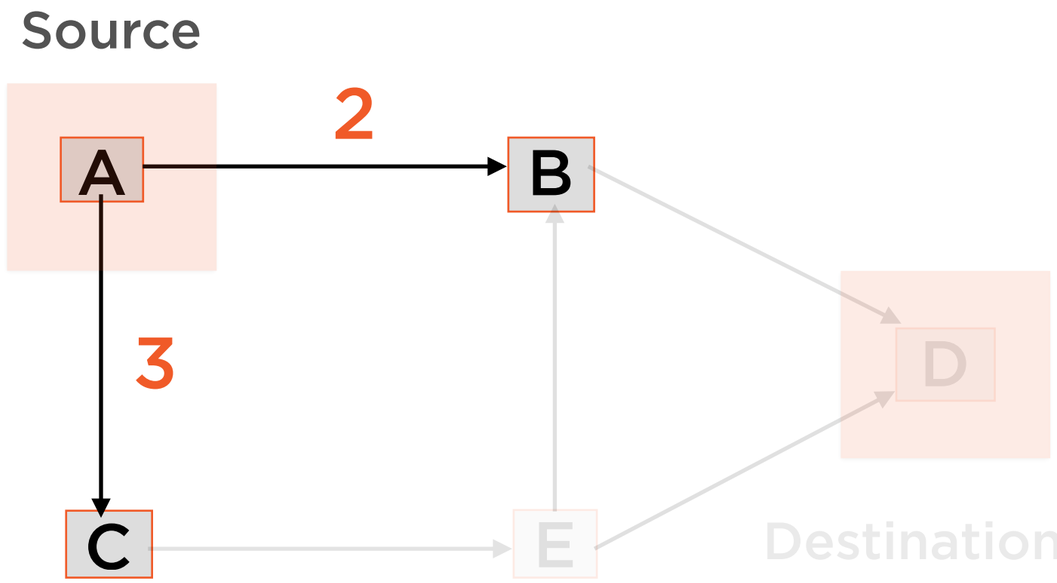


Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

Enqueuing based on distance => use of priority queue

Processing Priority Queue

Process Node A



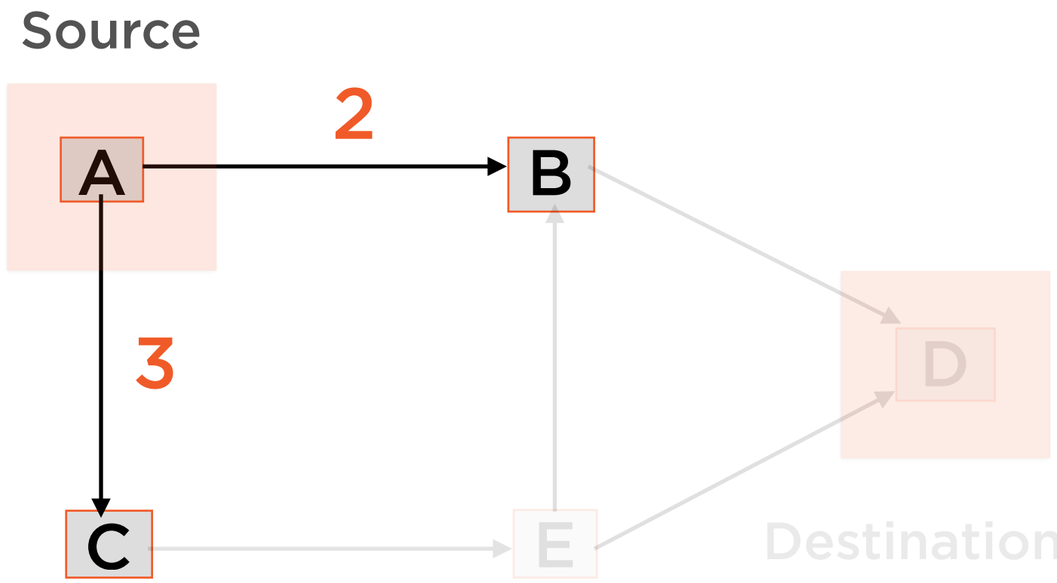
Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

Update distance table for those immediate neighbors

Processing Priority Queue

B,Inf	C,Inf	

Process Node A



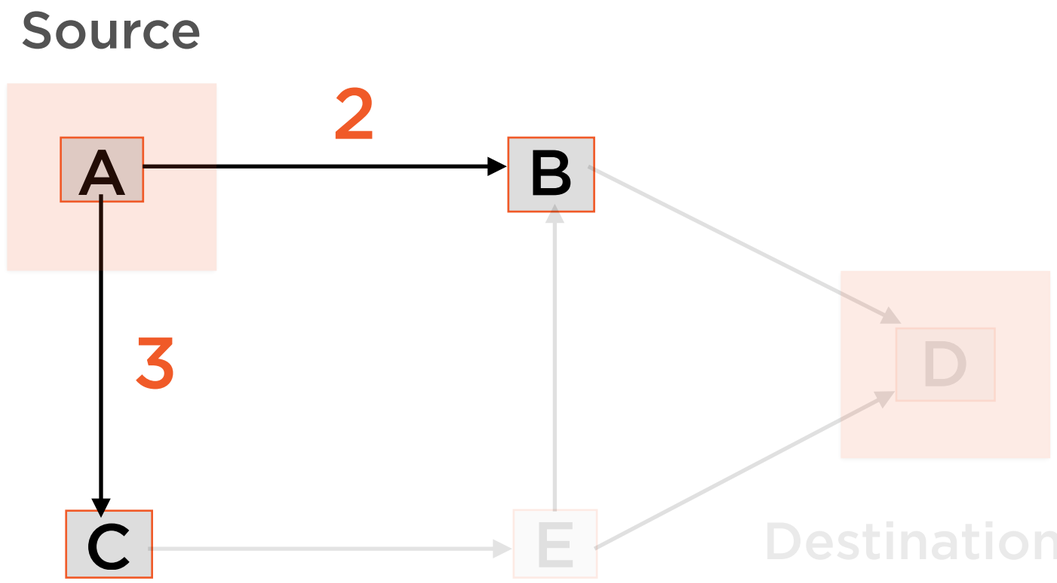
Node	Distance	Preceding Node
A	0	A
B	Inf	-
C	Inf	-
D	Inf	-
E	Inf	-

Distance of B and C from A is given by weights of edges

Processing Priority Queue

B,Inf	C,Inf	

Process Node A

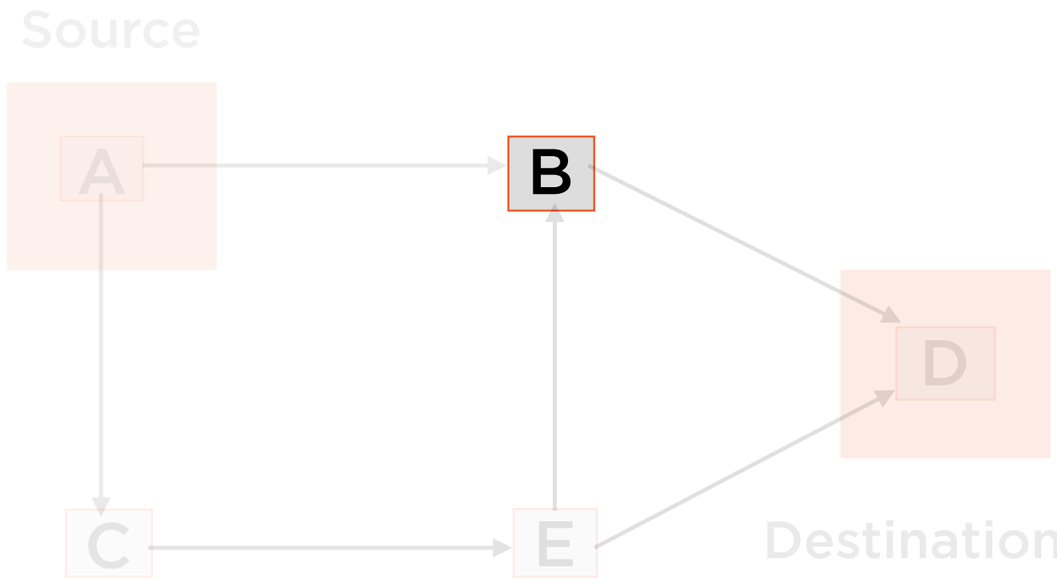


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Updated distance table for B,C - also update the queue

Processing Priority Queue

Process Node B

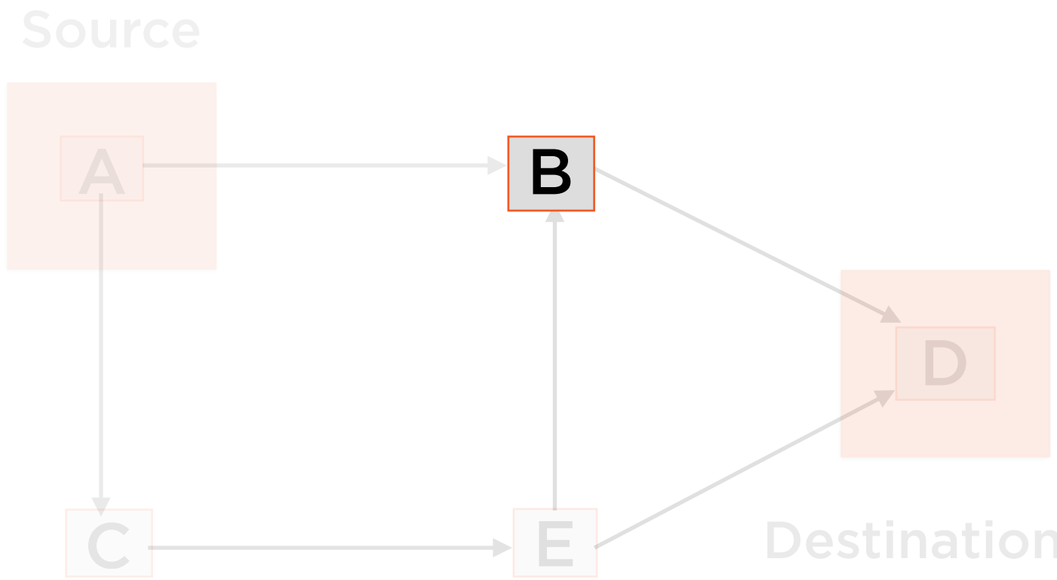


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Dequeue B and process it

Processing Priority Queue

Process Node B

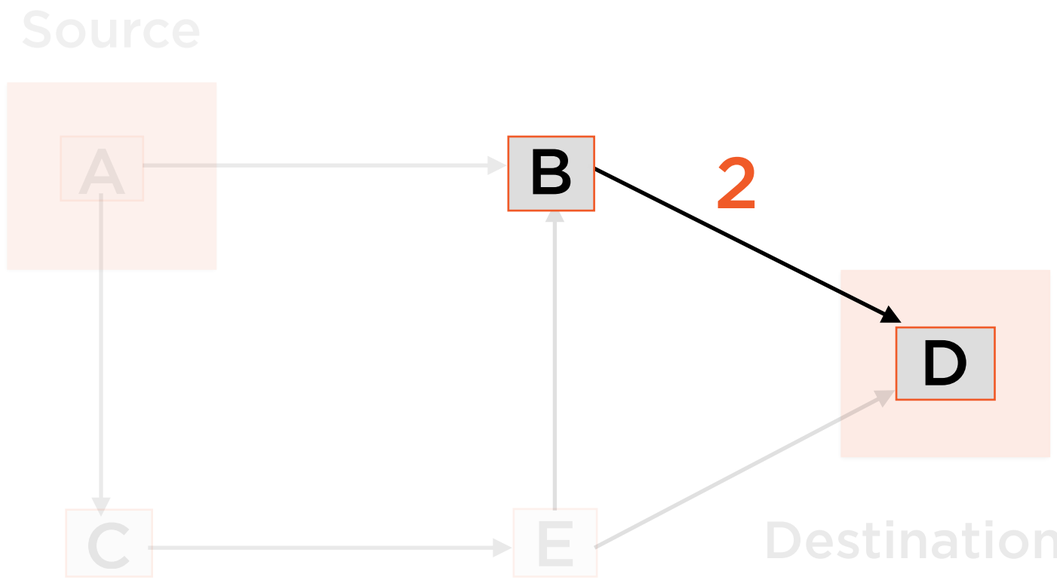


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Dequeue B and process it

Processing Priority Queue

Process Node B

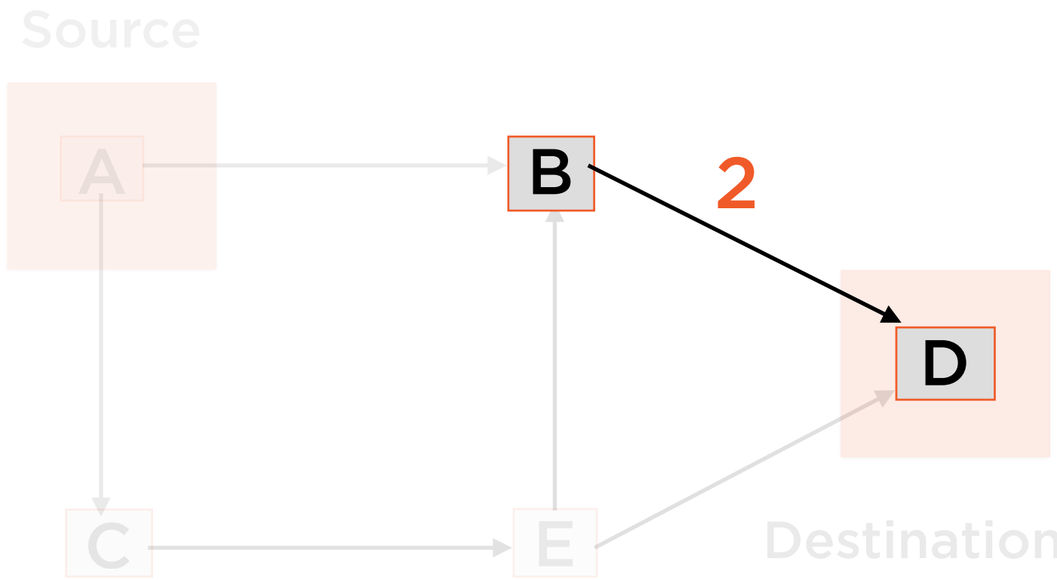


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Add immediate neighbors to queue

Processing Priority Queue

Process Node B



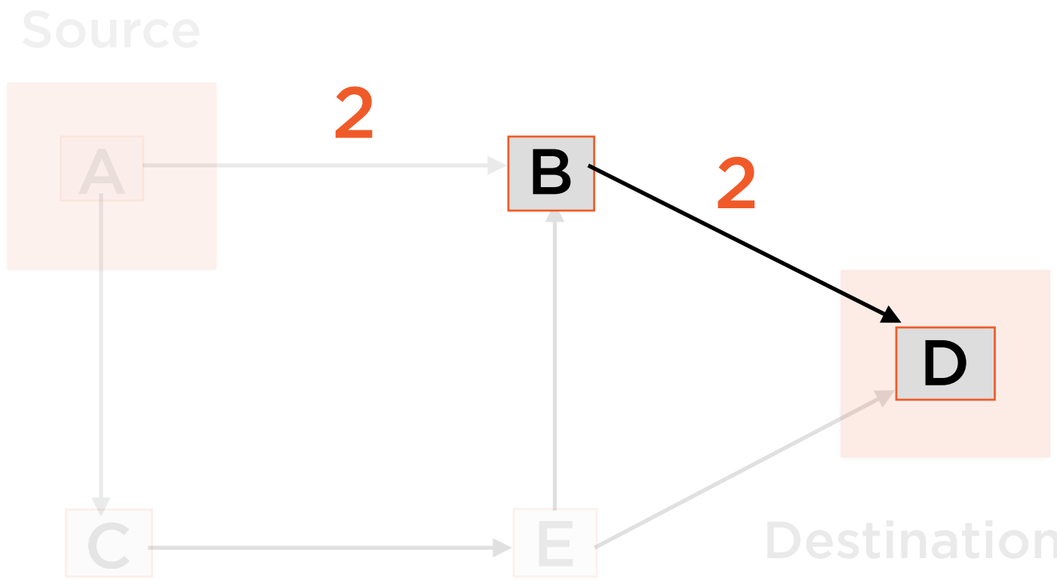
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Update distance table for those immediate neighbors

Processing Priority Queue

C,3	D,Inf	

Process Node B



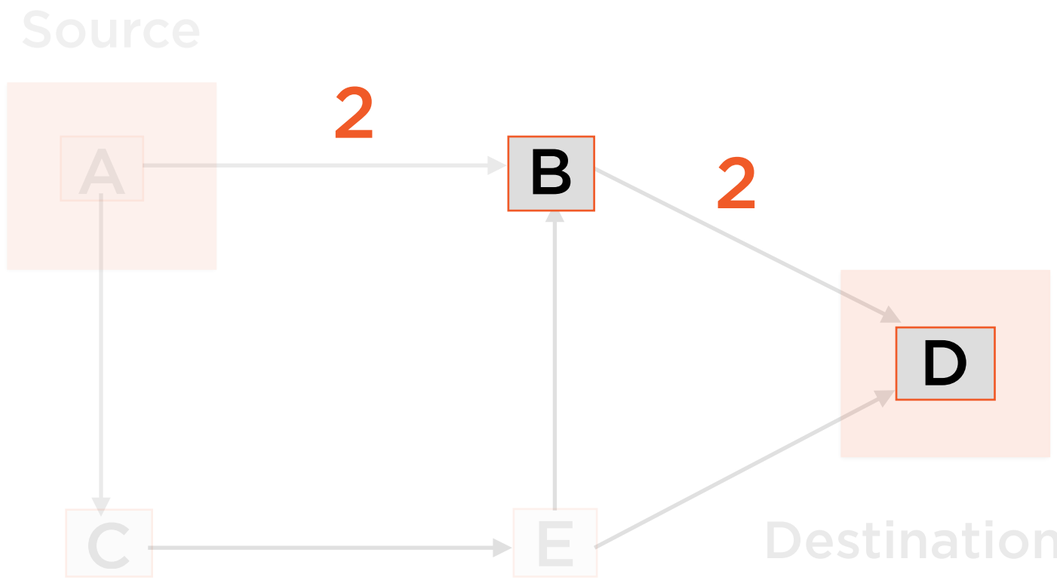
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

Now, D is 2 units from B, and B is 2 units from A

Processing Priority Queue

C,3	D,Inf	

Process Node B



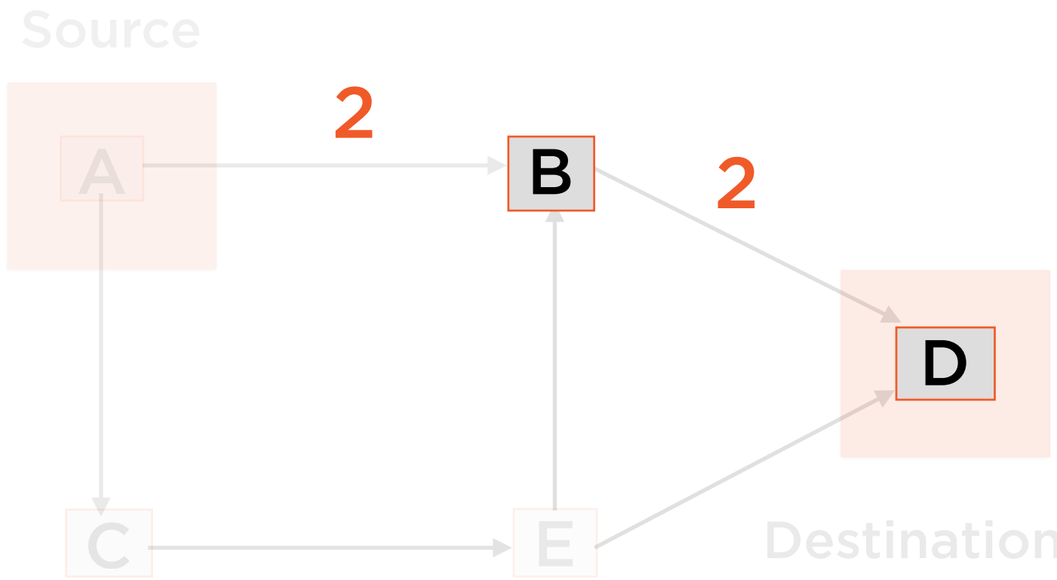
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	Inf	-
E	Inf	-

So, total distance from A to D is 4, and last node before D is B

Processing Priority Queue

C,3	D,Inf	

Process Node B

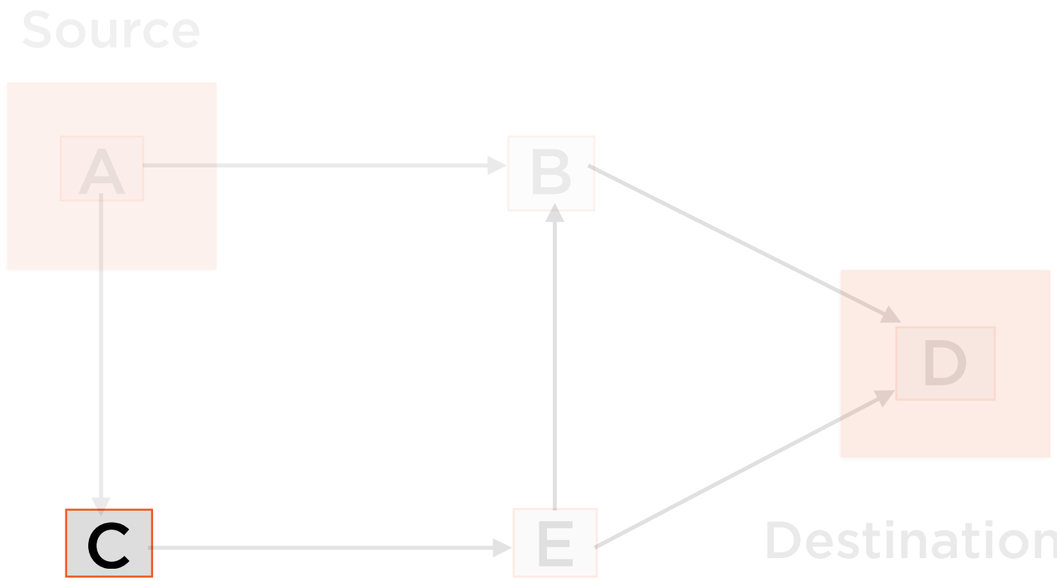


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	inf	-

So, total distance from A to D is 4, and last node before D is B

Processing Priority Queue

Process Node C

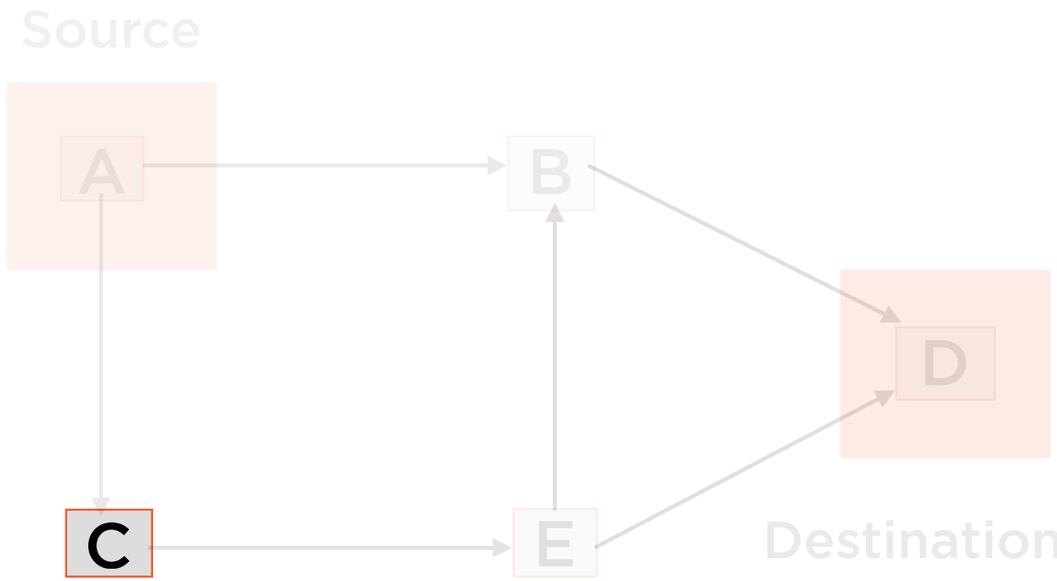


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	Inf	-

Dequeue C and process it

Processing Priority Queue

Process Node C

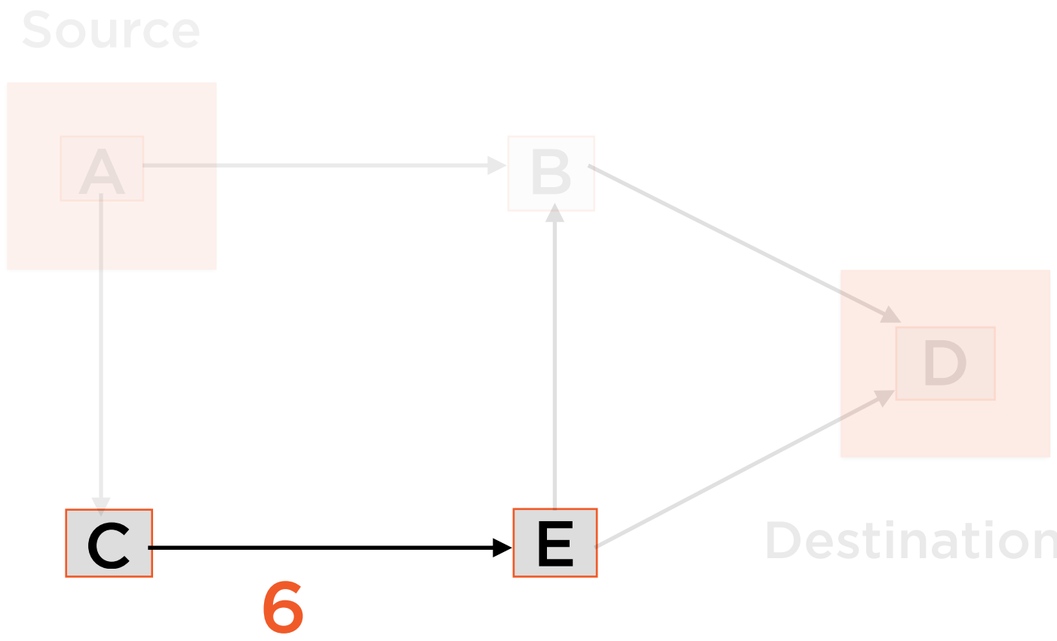


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	Inf	-

Dequeue C and process it

Processing Priority Queue

Process Node C

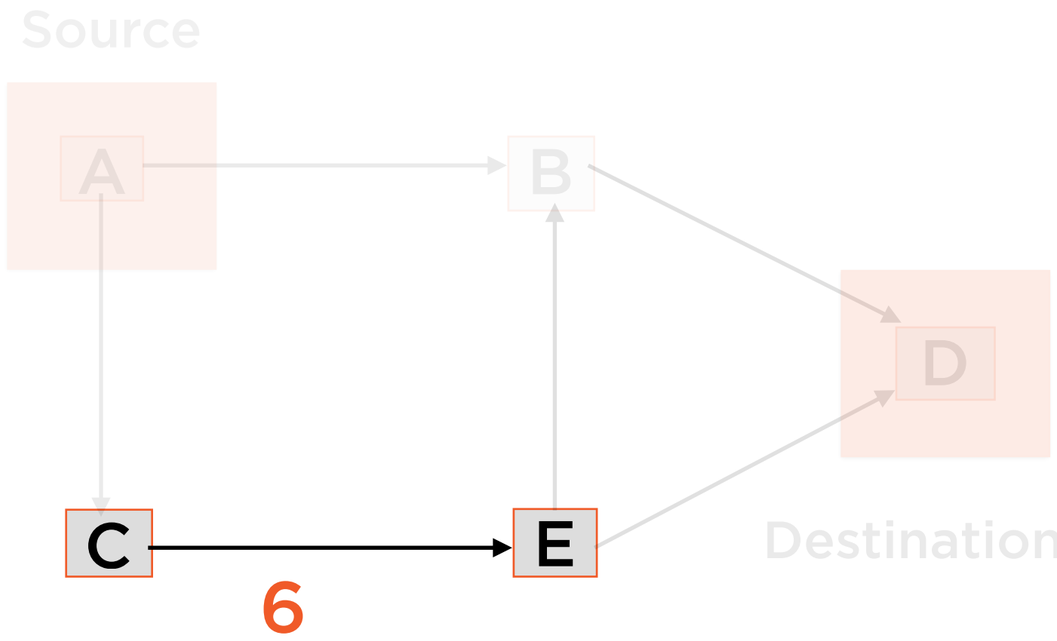


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	Inf	-

Enqueue immediate neighbors (descending order of weight)

Processing Priority Queue

Process Node C



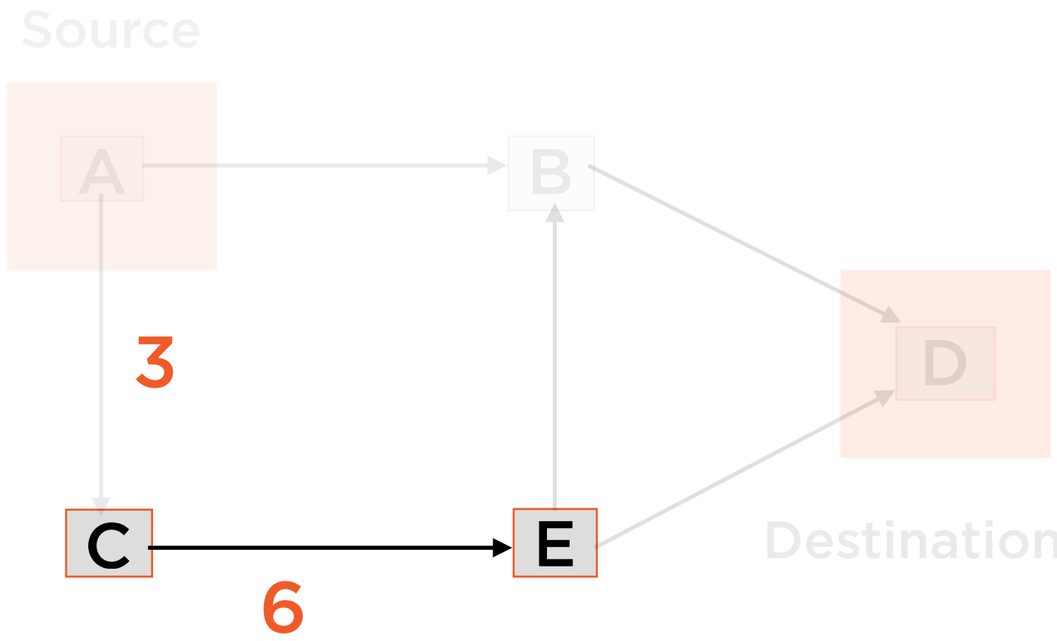
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	Inf	-

Update the distance table for all neighbors

Processing Priority Queue

D,4	E,Inf	

Process Node C



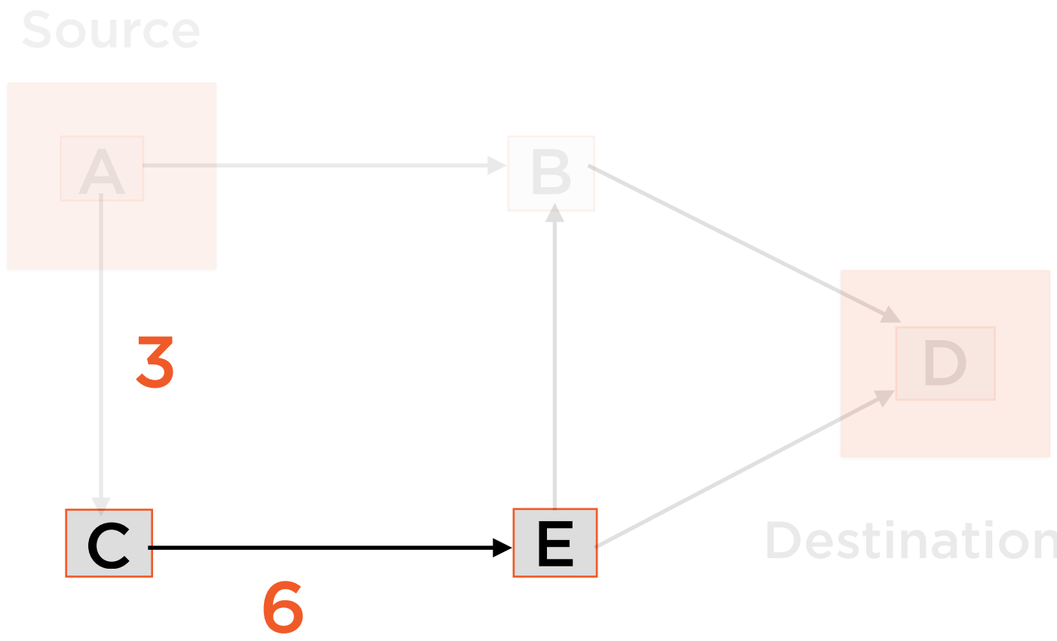
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	Inf	-

Now E is 6 units from C, and C is 3 units from A

Processing Priority Queue

D,4	E,Inf	

Process Node C

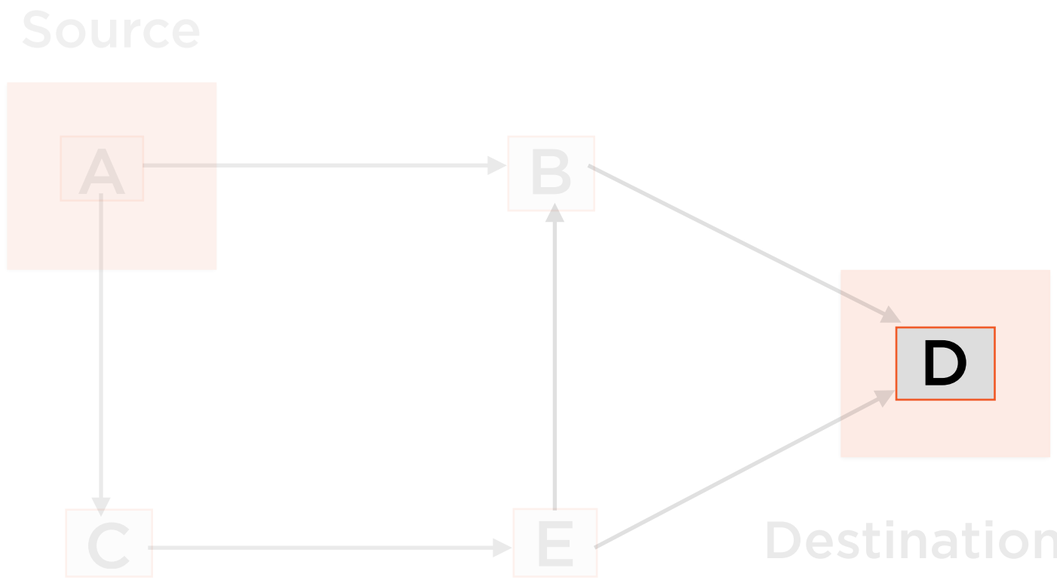


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

So, E is 9 units from A, and last vertex before E is C

Processing Priority Queue

Process Node D

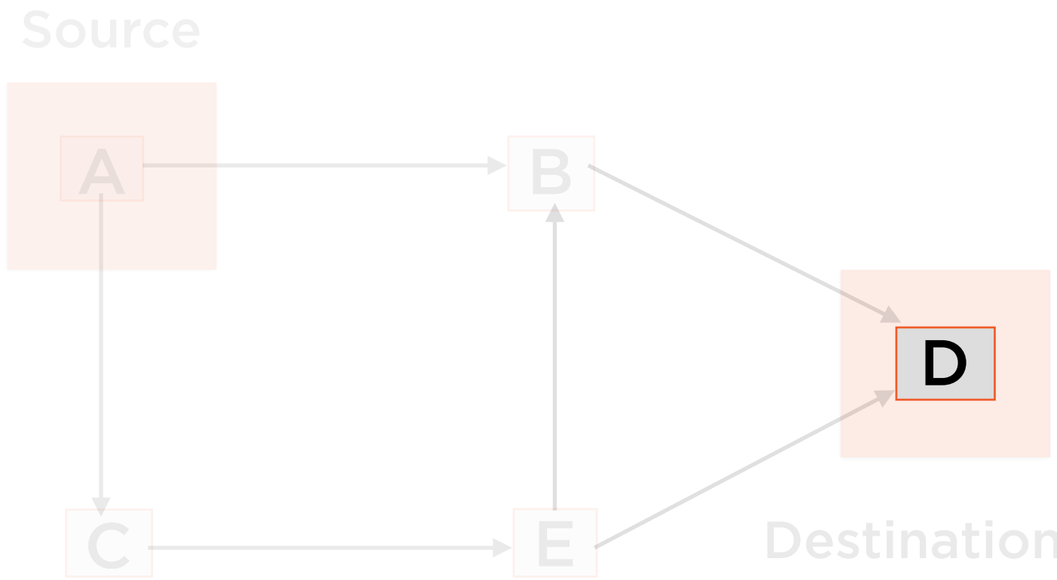


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Now dequeue D and process it

Processing Priority Queue

Process Node D

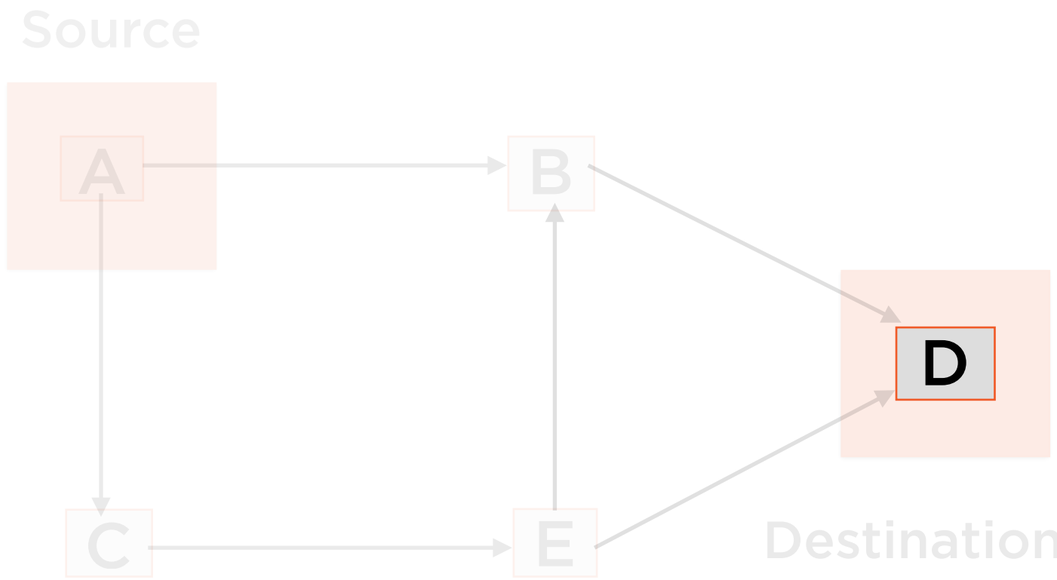


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Now dequeue D and process it

Processing Priority Queue

Process Node D

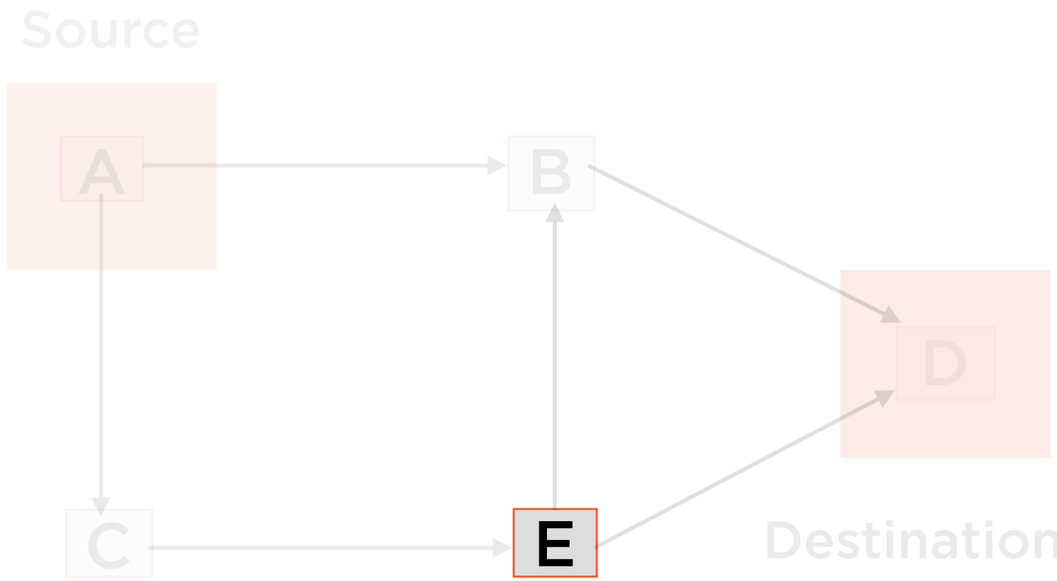


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

No immediate neighbors - move on

Processing Priority Queue

Process Node E

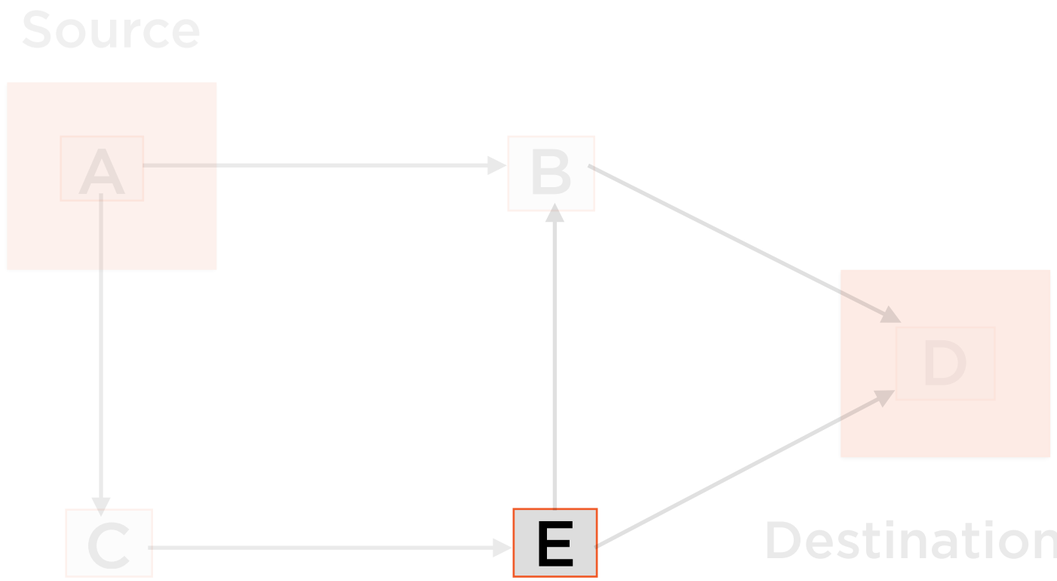


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Dequeue E and process it

Processing Priority Queue

Process Node E

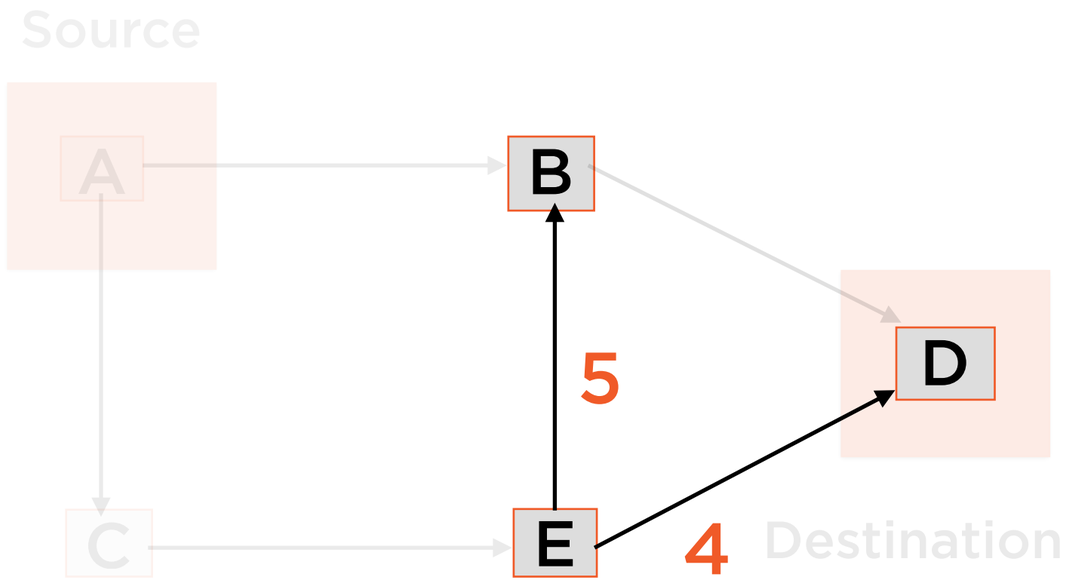


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Dequeue E and process it

Processing
Priority Queue

Process Node E

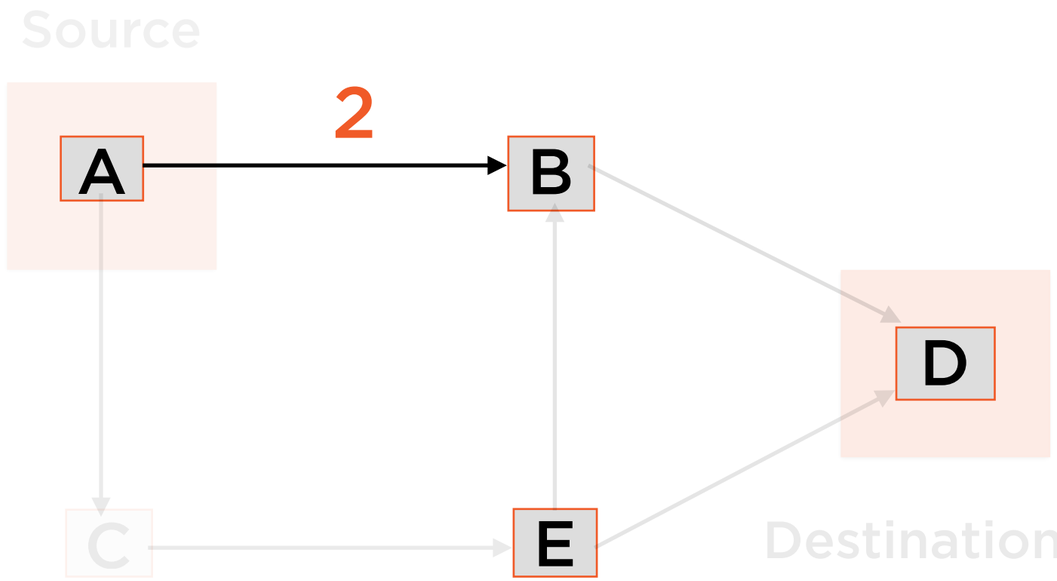


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Immediate neighbors already visited - check if need to re-enqueue

Processing
Priority Queue

Process Node E

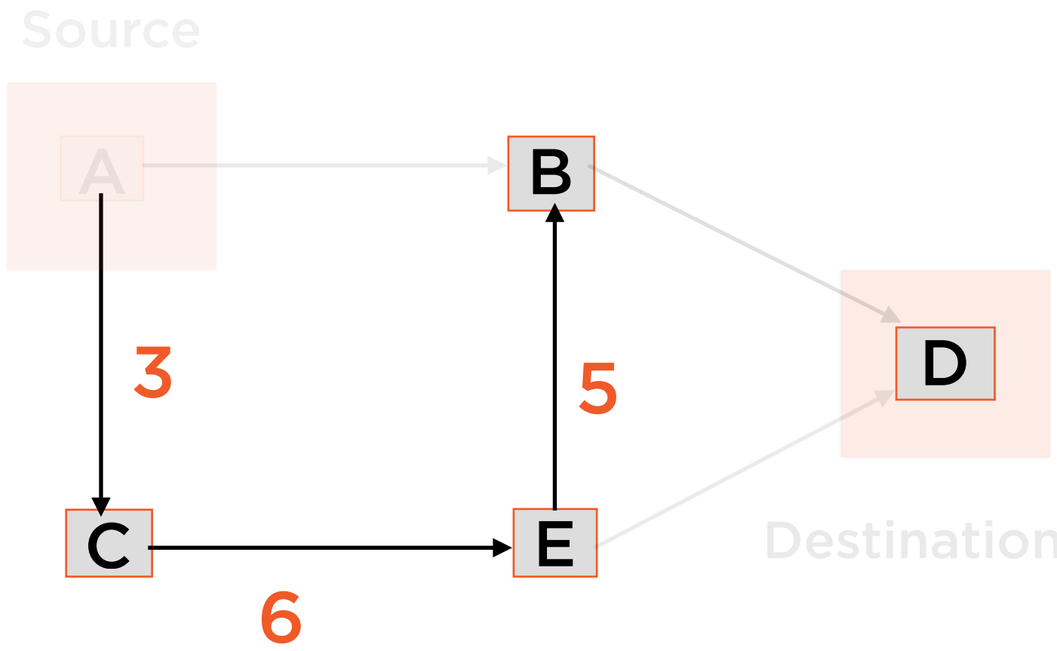


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Current shortest path to B is via A - distance 2

Processing
Priority Queue

Process Node E

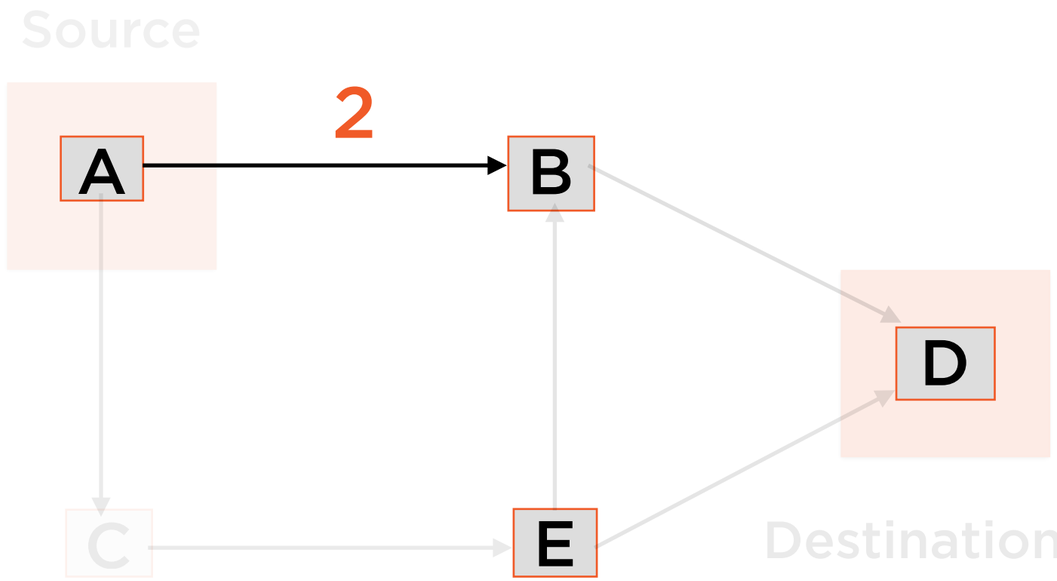


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Path to B via E = 14 units

Processing Priority Queue

Process Node E

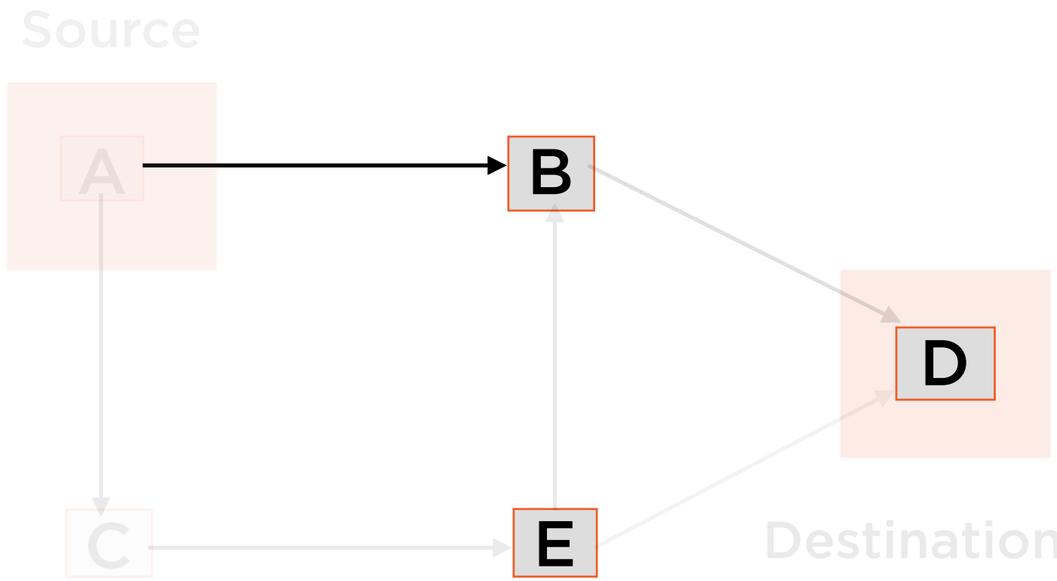


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

No need to update shortest path to B, no need to re-enqueue B

Processing
Priority Queue

Process Node E

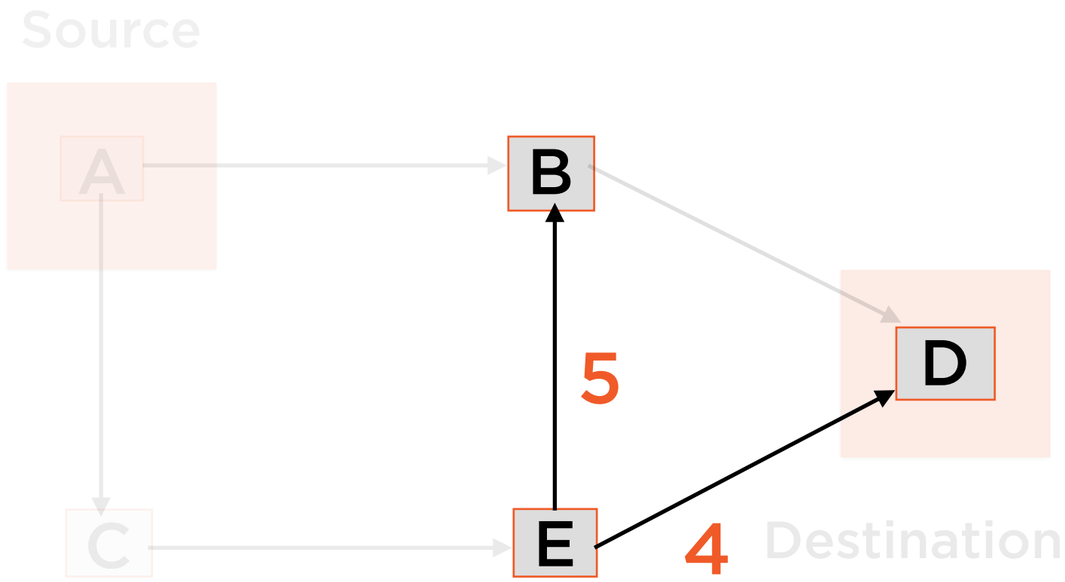


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

No need to update shortest path to B, no need to re-enqueue B

Processing
Priority Queue

Process Node E

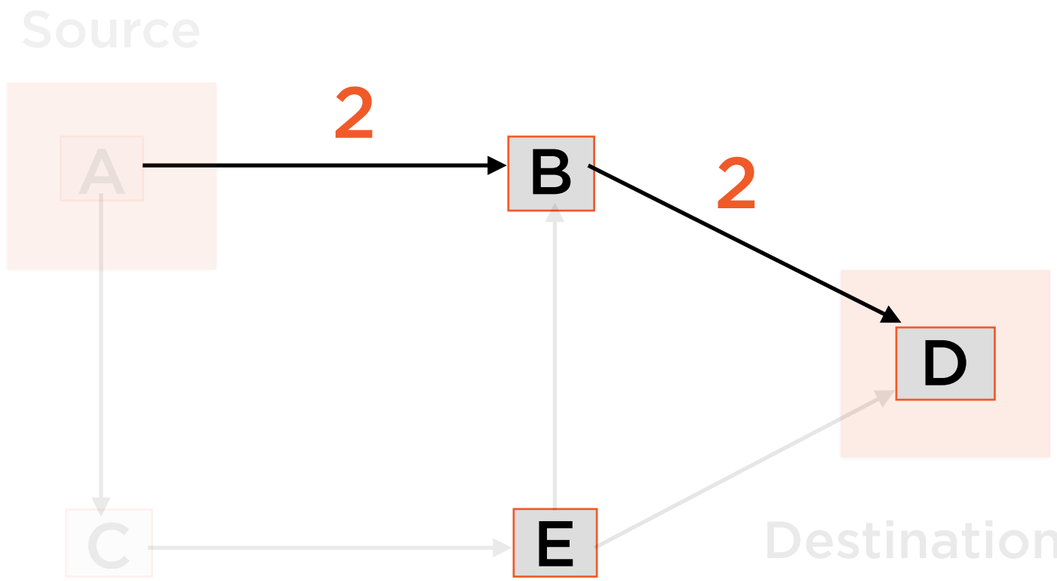


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Checked B, now let's check D

Processing Priority Queue

Process Node E

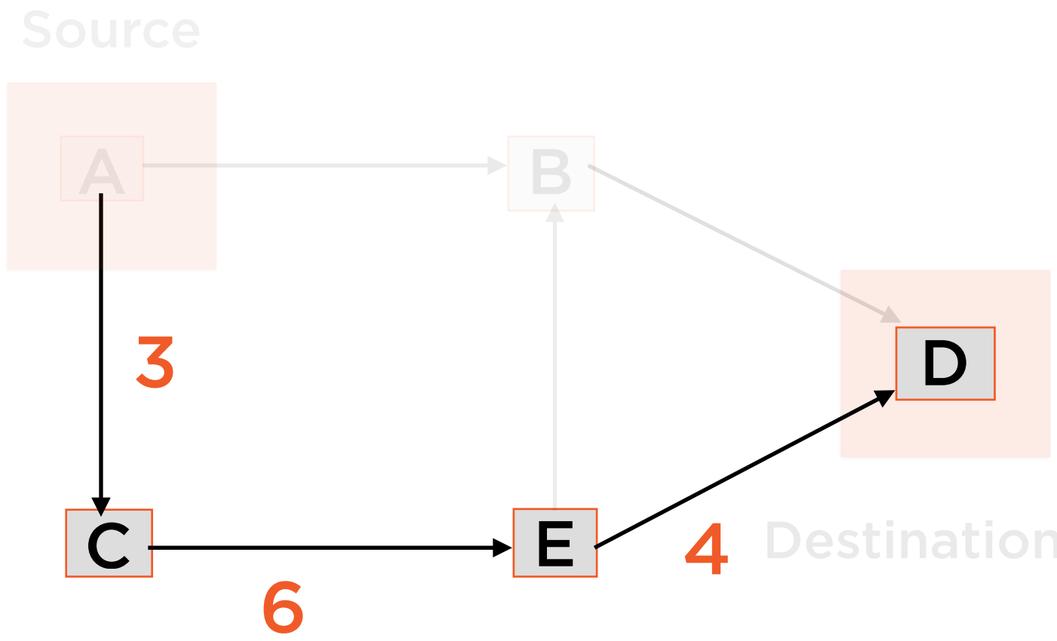


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Current shortest path to D is 4 units, via B

Processing Priority Queue

Process Node E

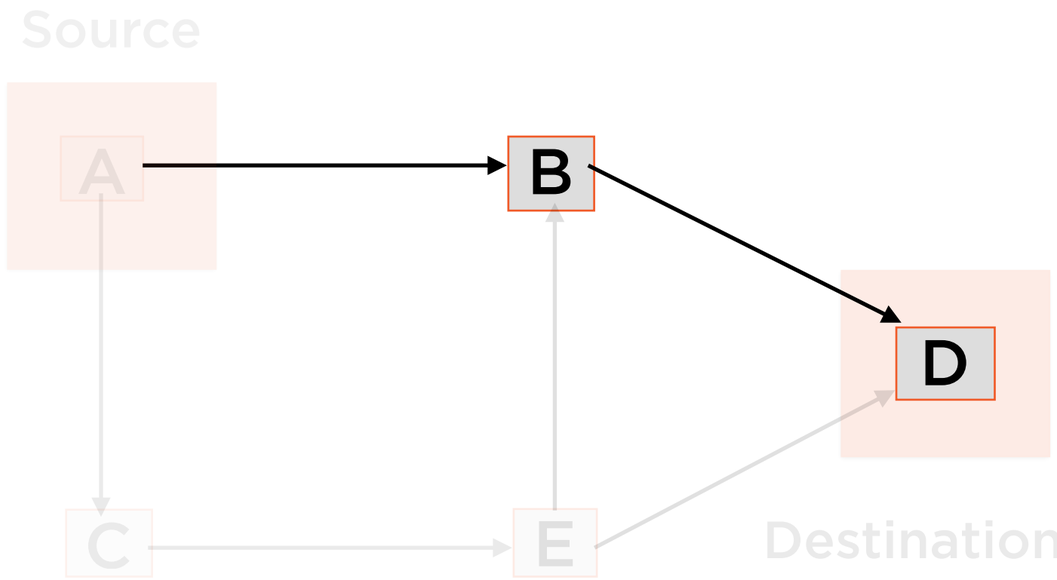


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Path to D via E = 13 units

Processing Priority Queue

Process Node E

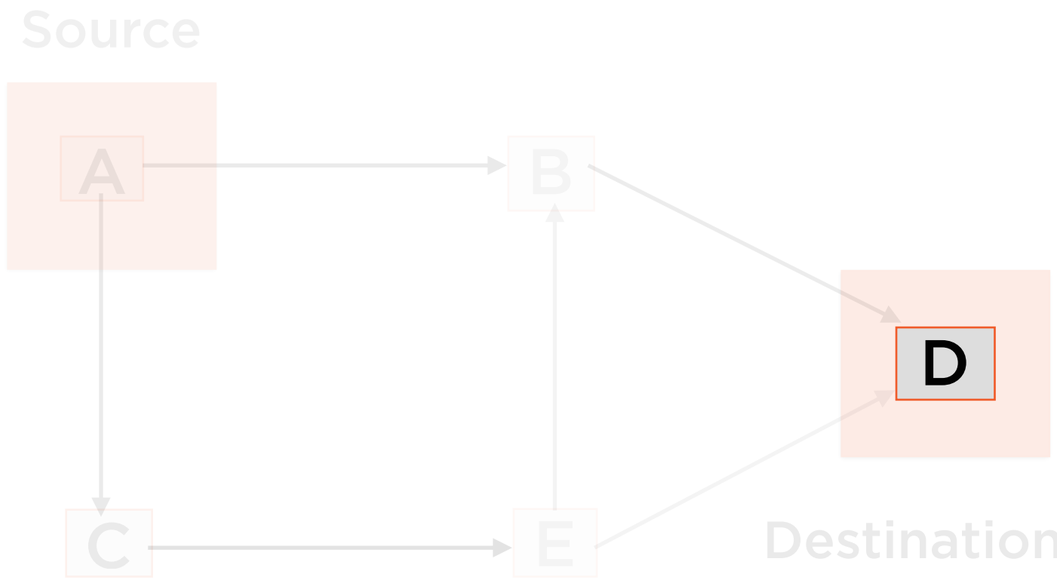


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

No need to update shortest path to D, no need to re-enqueue D

Processing
Priority Queue

Process Node D

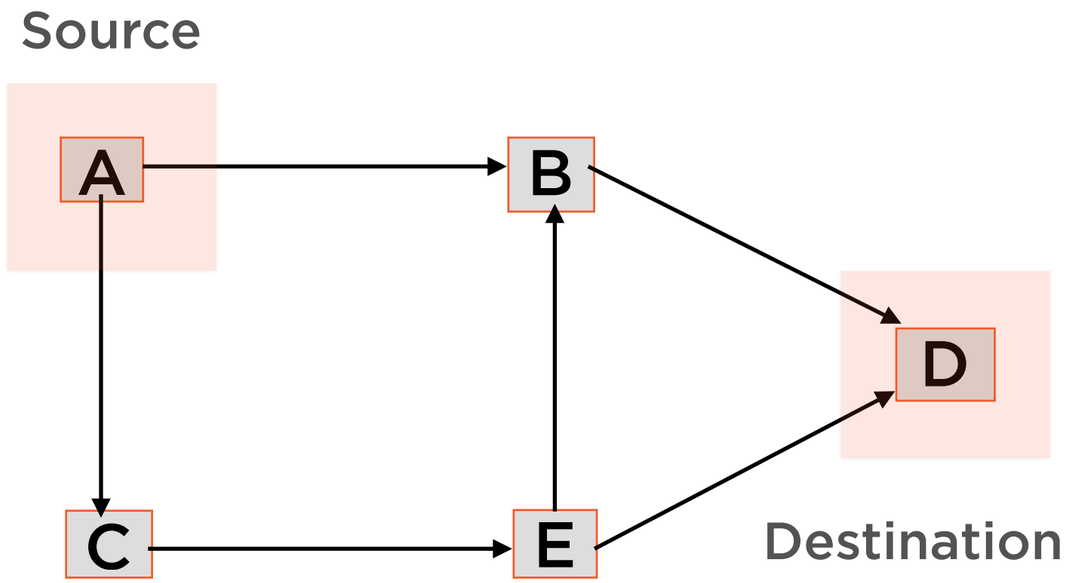


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Processing queue empty - algorithm complete

Processing
Priority Queue

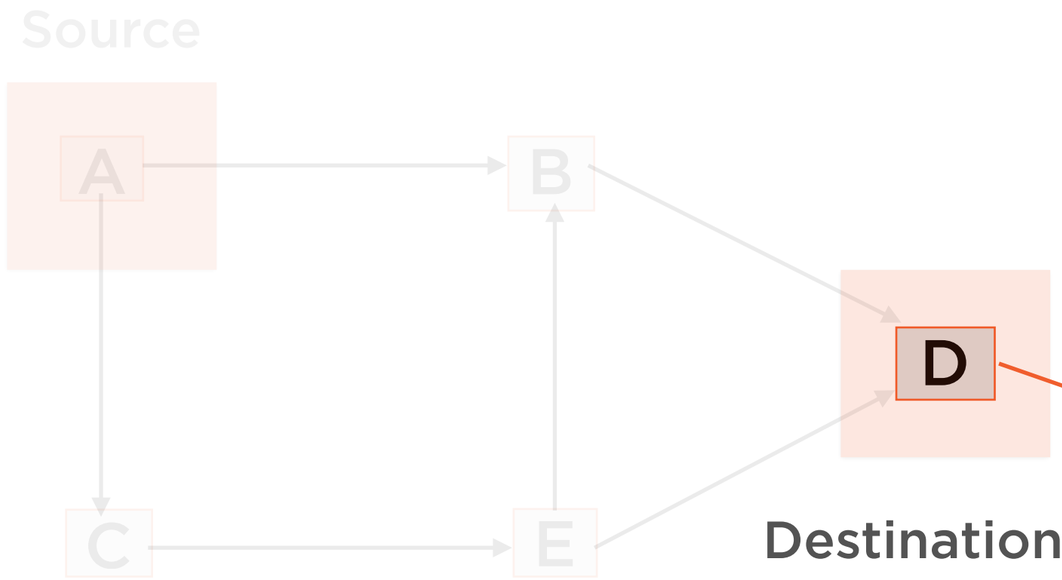
Finding Shortest Path



Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

To trace out the shortest path, backtrack from destination D to source A

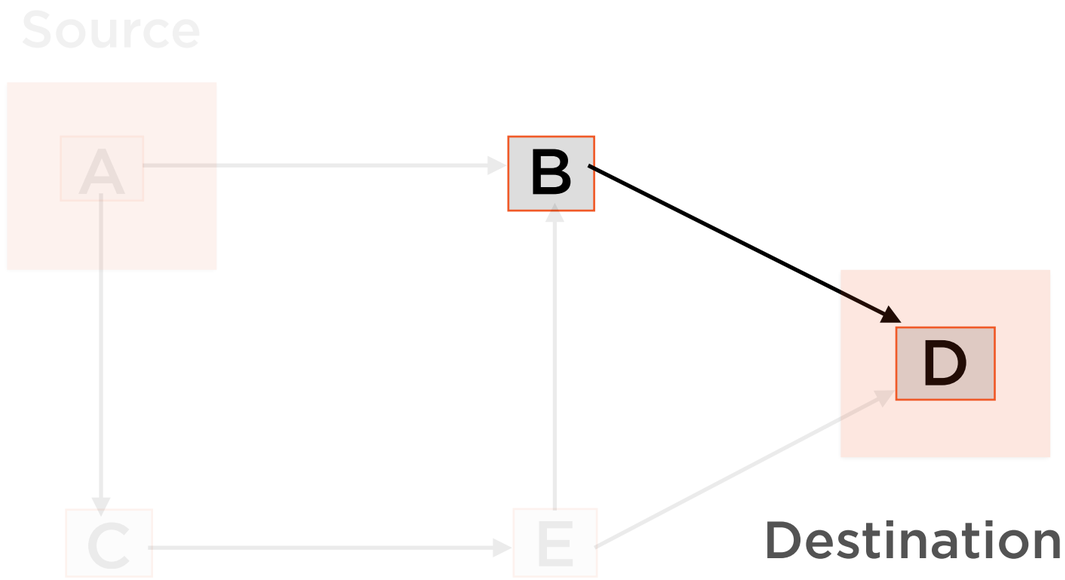
Backtracking



Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Shortest Path D

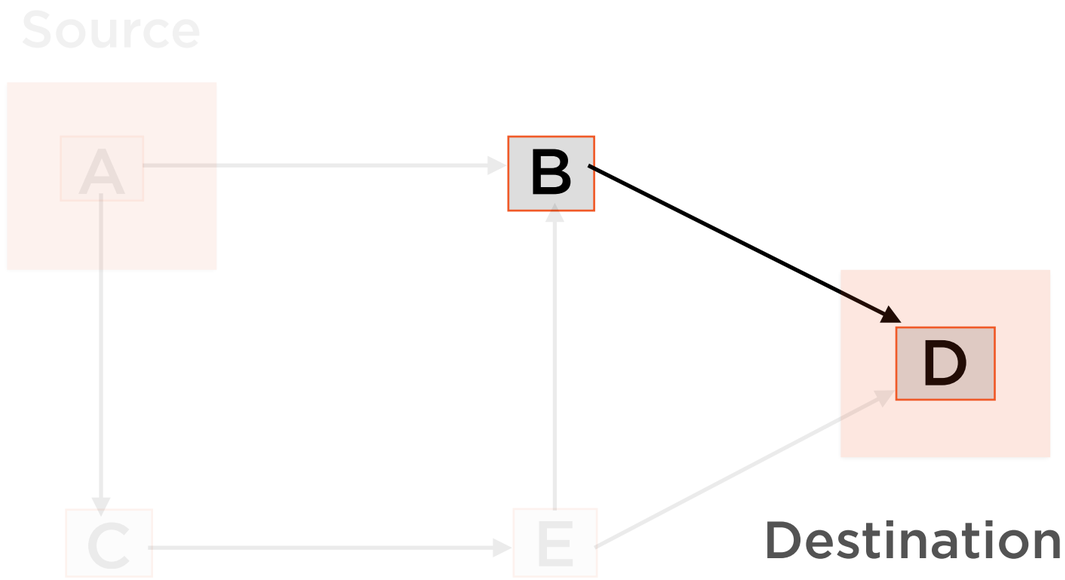
Backtracking



Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

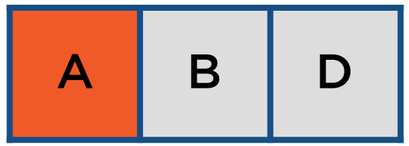
Shortest Path

Backtracking

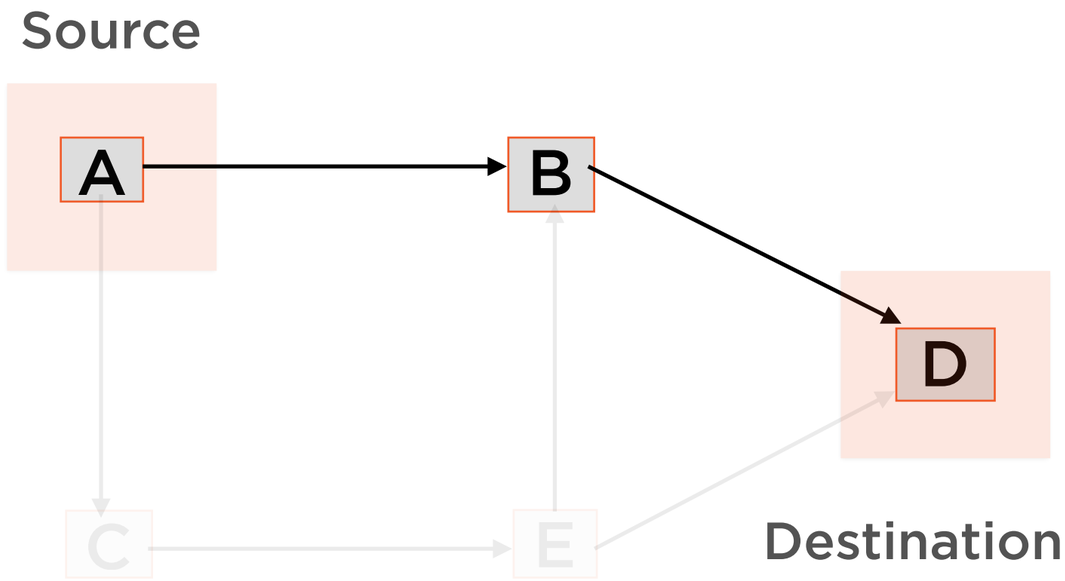


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Shortest Path

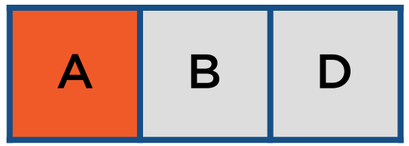


Backtracking

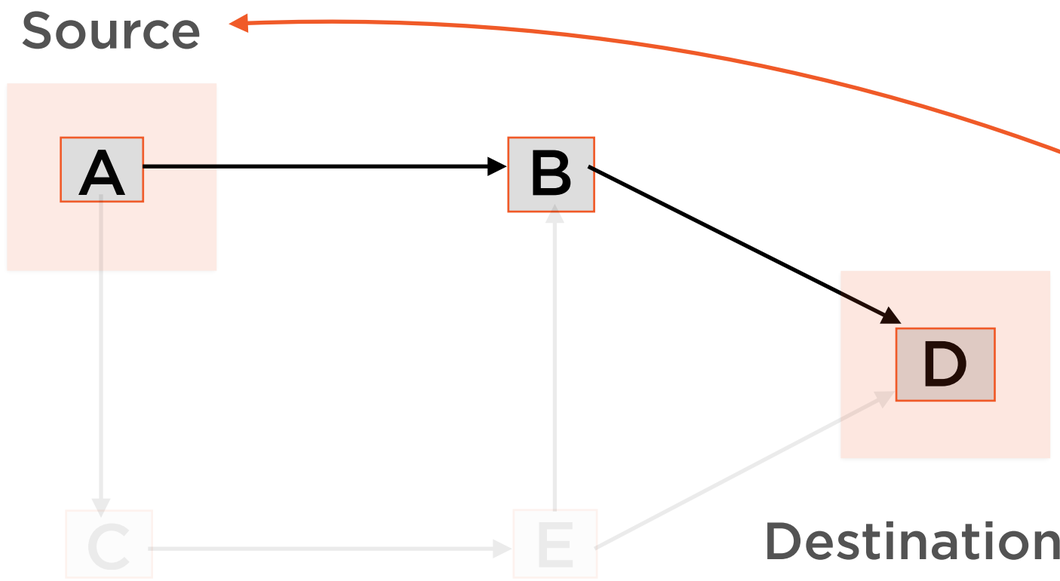


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Shortest Path

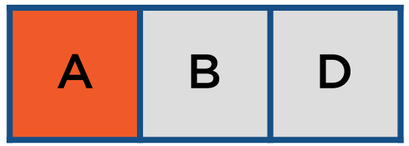


Backtracking

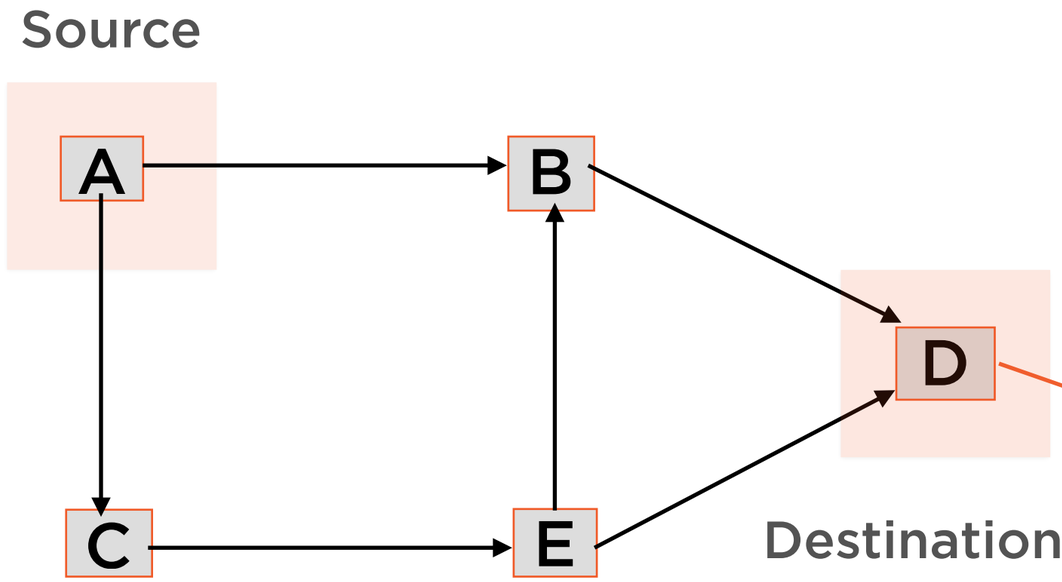


Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Shortest Path

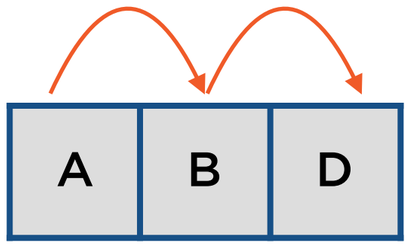


Backtracking



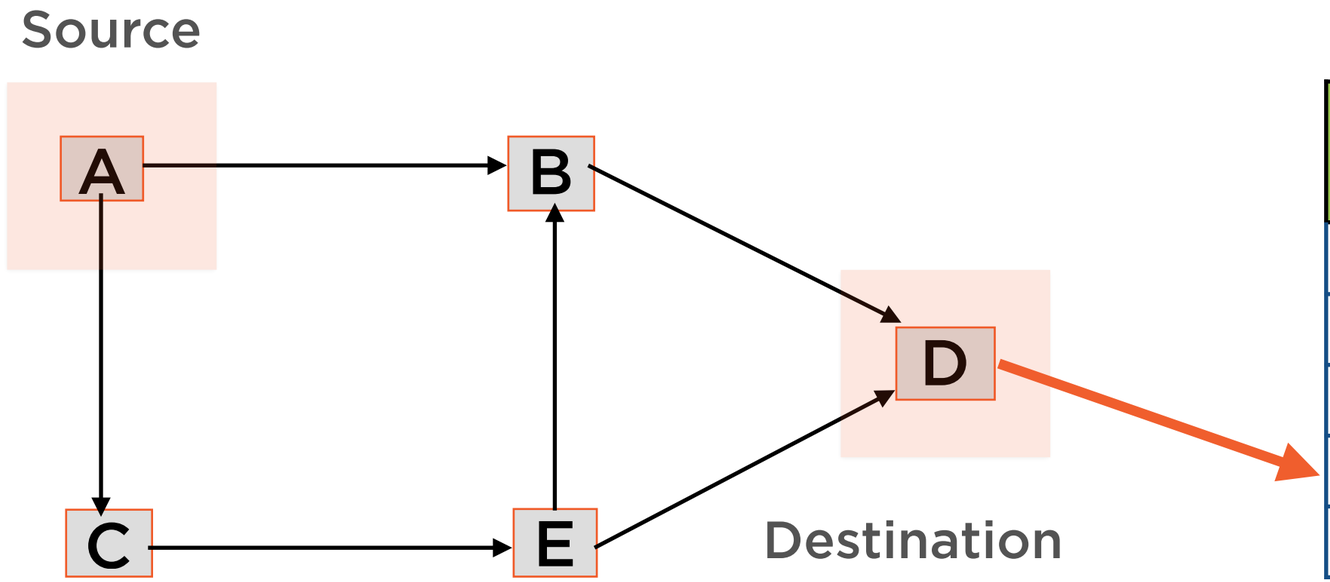
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Shortest Path



Cost of shortest path = sum of weights = 4

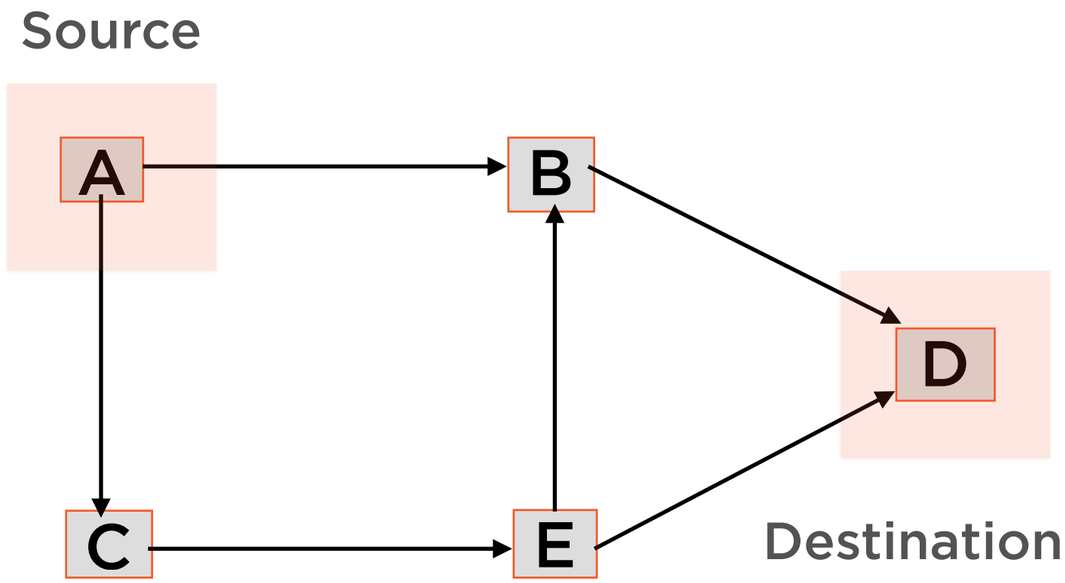
Backtracking



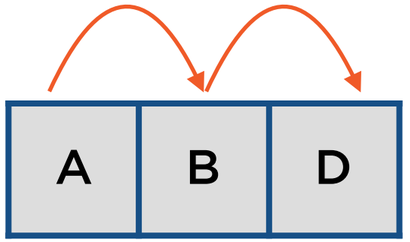
Node	Distance	Preceding Node
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

Notice how we “walk back” the distance table to construct the shortest path

Backtracking



“Last-In-First-Out” => Use a stack



Dijkstra's Algorithm

Data	Data Structure
Distance table	3-column array
Backtracking	Stack
Enqueuing neighbors	Priority queue (binary heap or array)

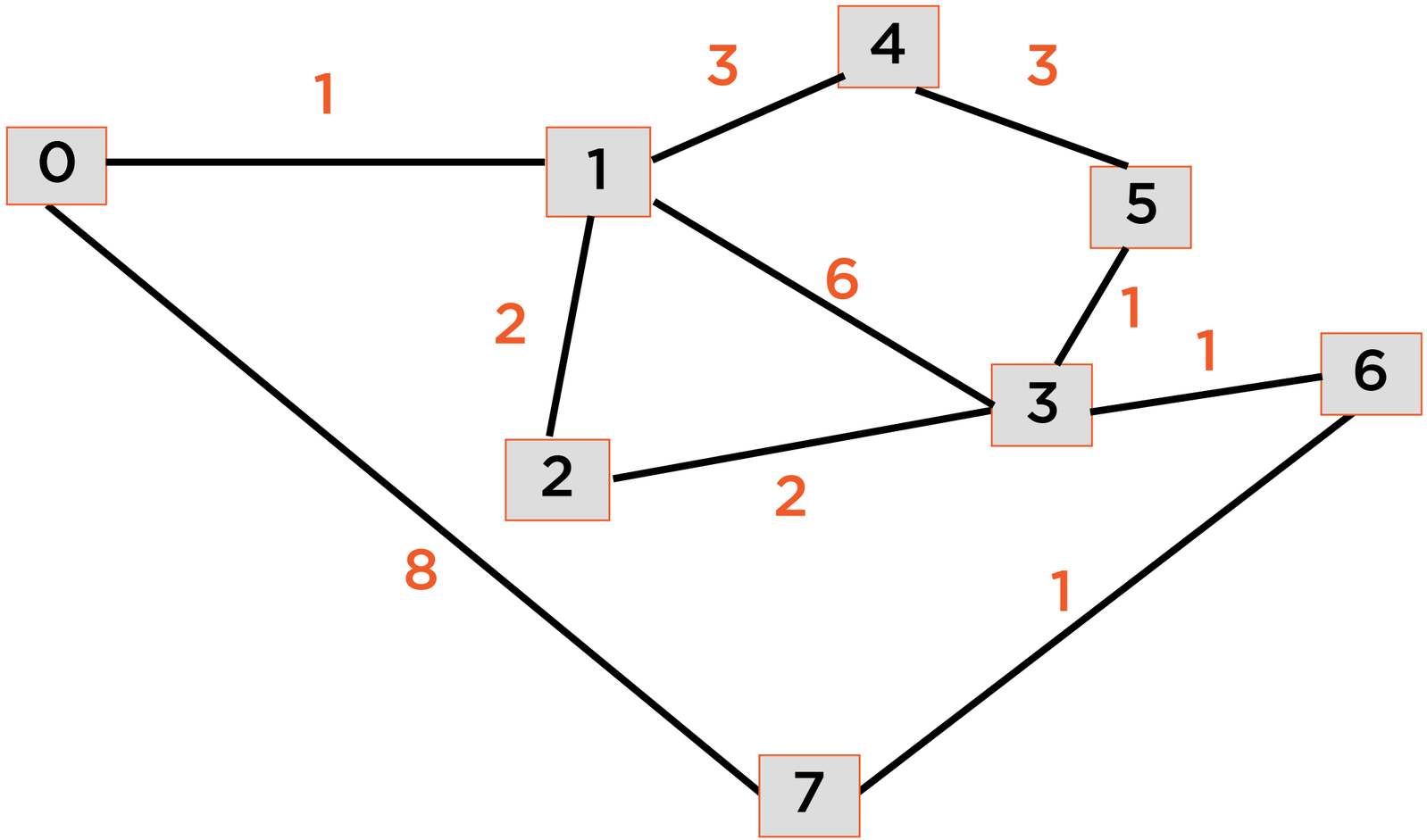
Dijkstra's Algorithm

Queue Data Structure	Running Time
Binary Heap	O(E ln(V))
Array	$O(E + V^2)$

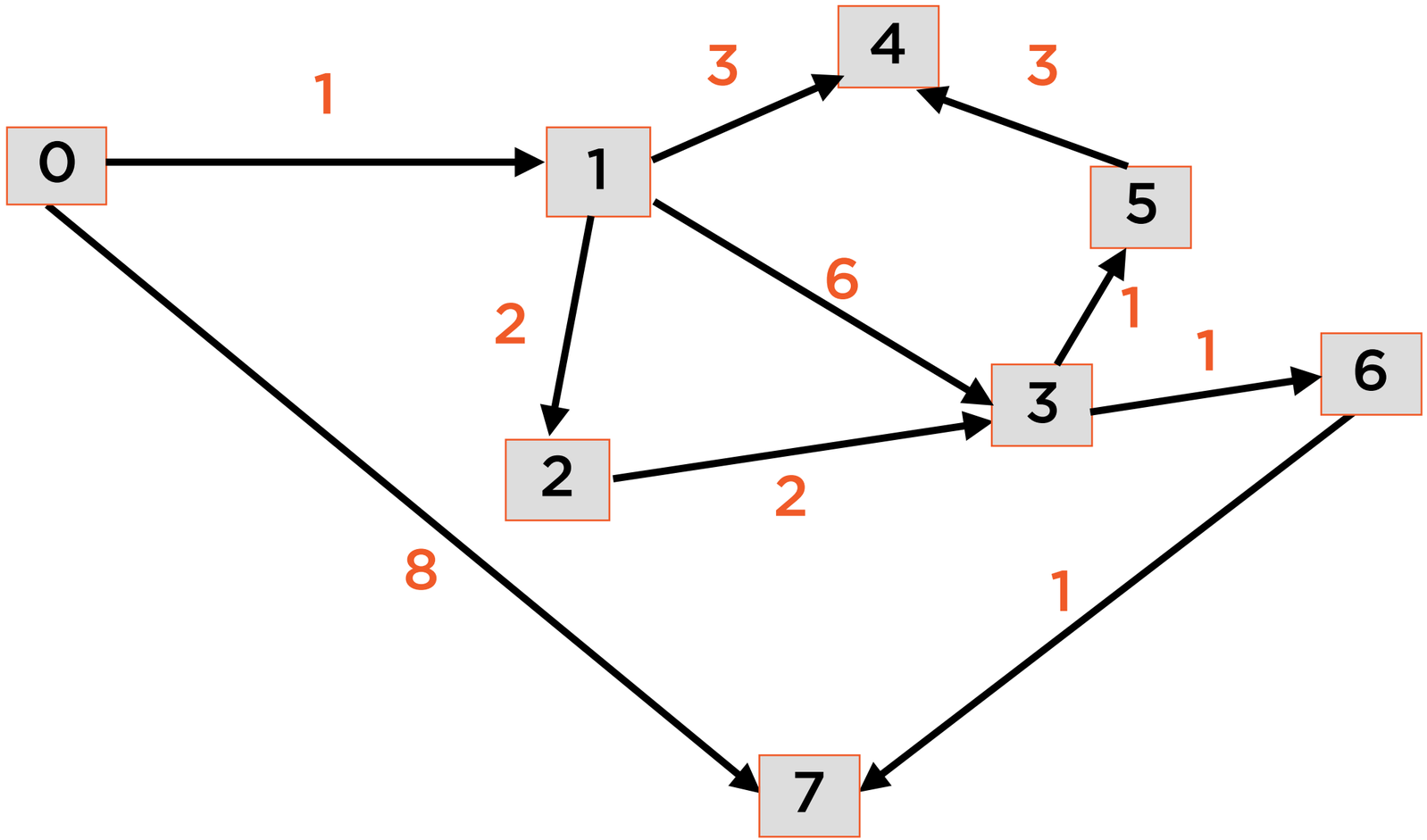
Demo

Calculate the shortest path for weighted graphs using Dijkstra's algorithm

A Sample Undirected Graph



A Sample Directed Graph



Summary

Shortest path algorithms are widely used in transportation and scheduling

Such algorithms focus on the most efficient route between a pair of nodes

Edge weights determine the cost of a path in such algorithms

If all edge weights are equal, use the unweighted shortest path algorithm

If edge weights are unequal, use Dijkstra's algorithm