Shortest Path in Graph

Dijkstra algorithm &

Bellman Ford Algorithm

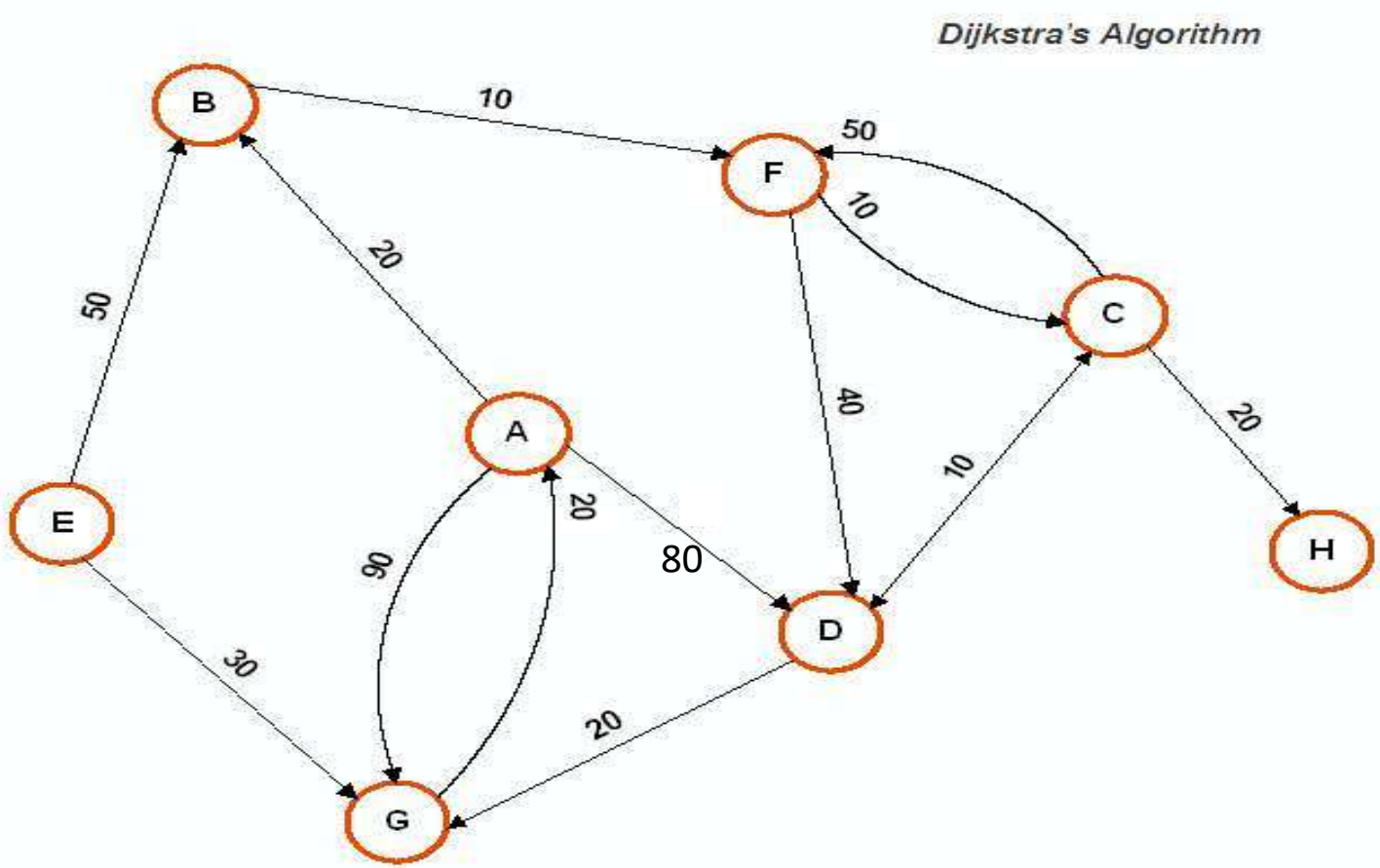
Dijkstra’s Algorithm Intro

- **Dijkstra's Algorithm** derived by a Dutch computer scientist ‘Edsger Dijkstra’ in 1956 and published in 1959.
- It’s a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree.
- This algorithm is often used in routing and as a subroutine in other graph algorithms.

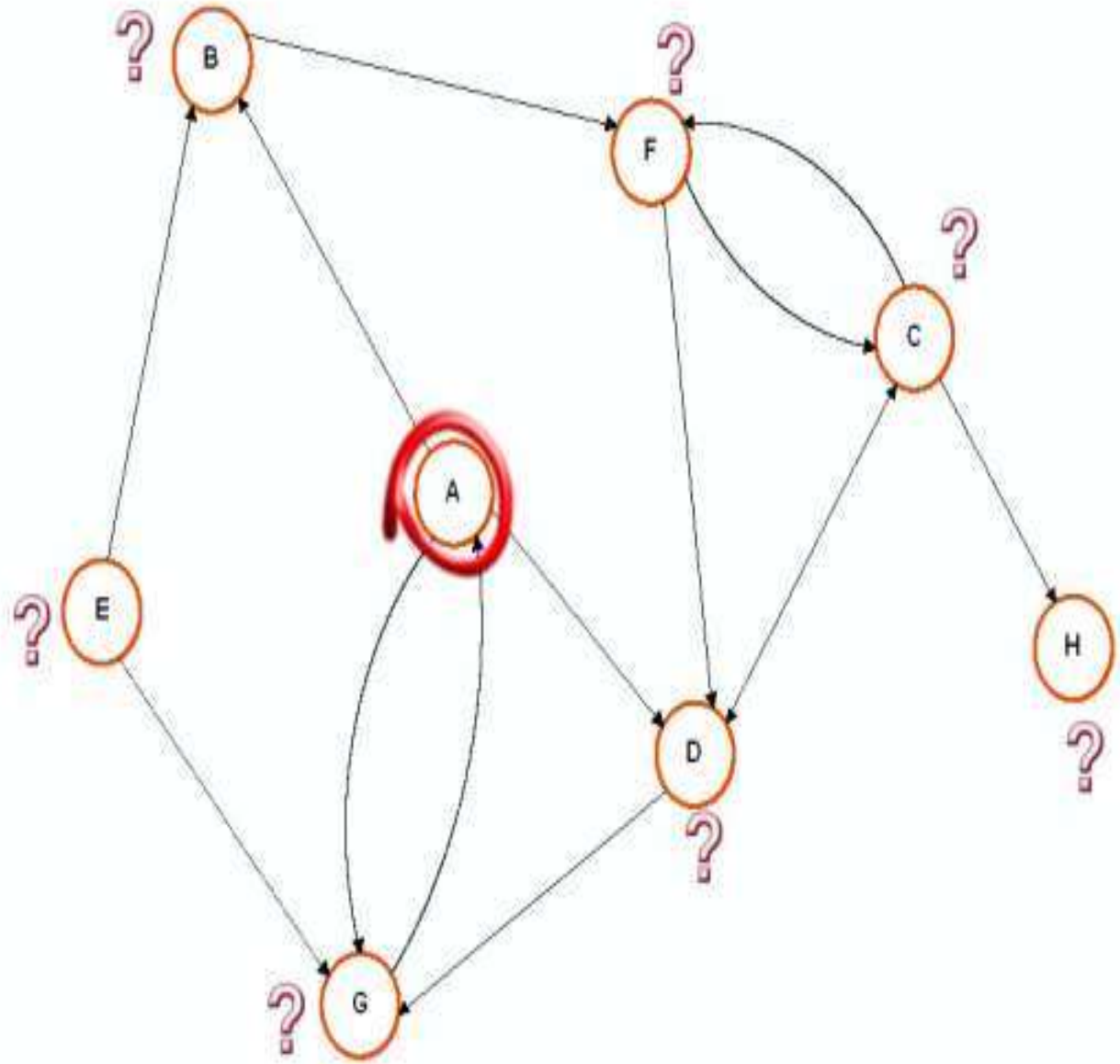
How to Solve

- This algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.
- According to this algorithm, to solve a given problem, we need to solve different parts of problems.
- Use Greedy Approach.

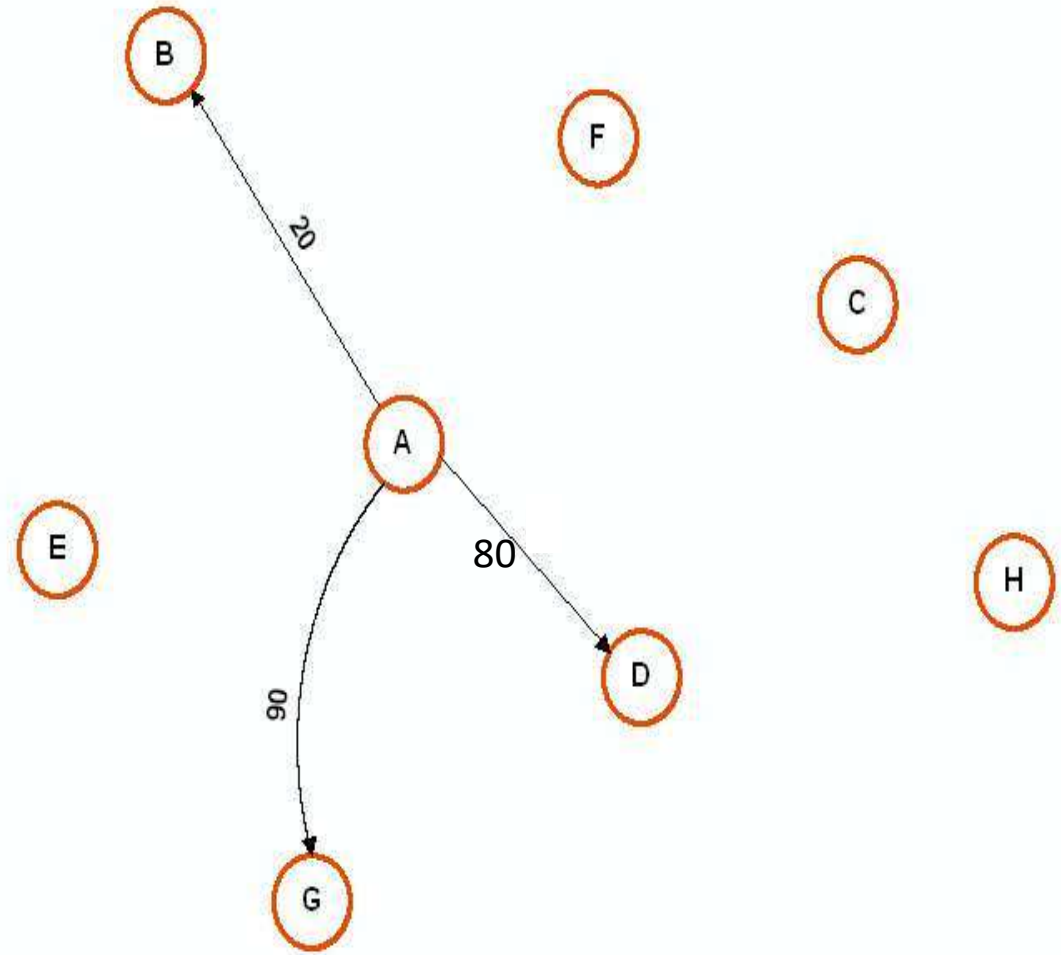
Example


Dijkstra's Algorithm
B
F
C
A
E
D
G
H
10
50
10
20
50
40
10
20
20
90
80
30
20

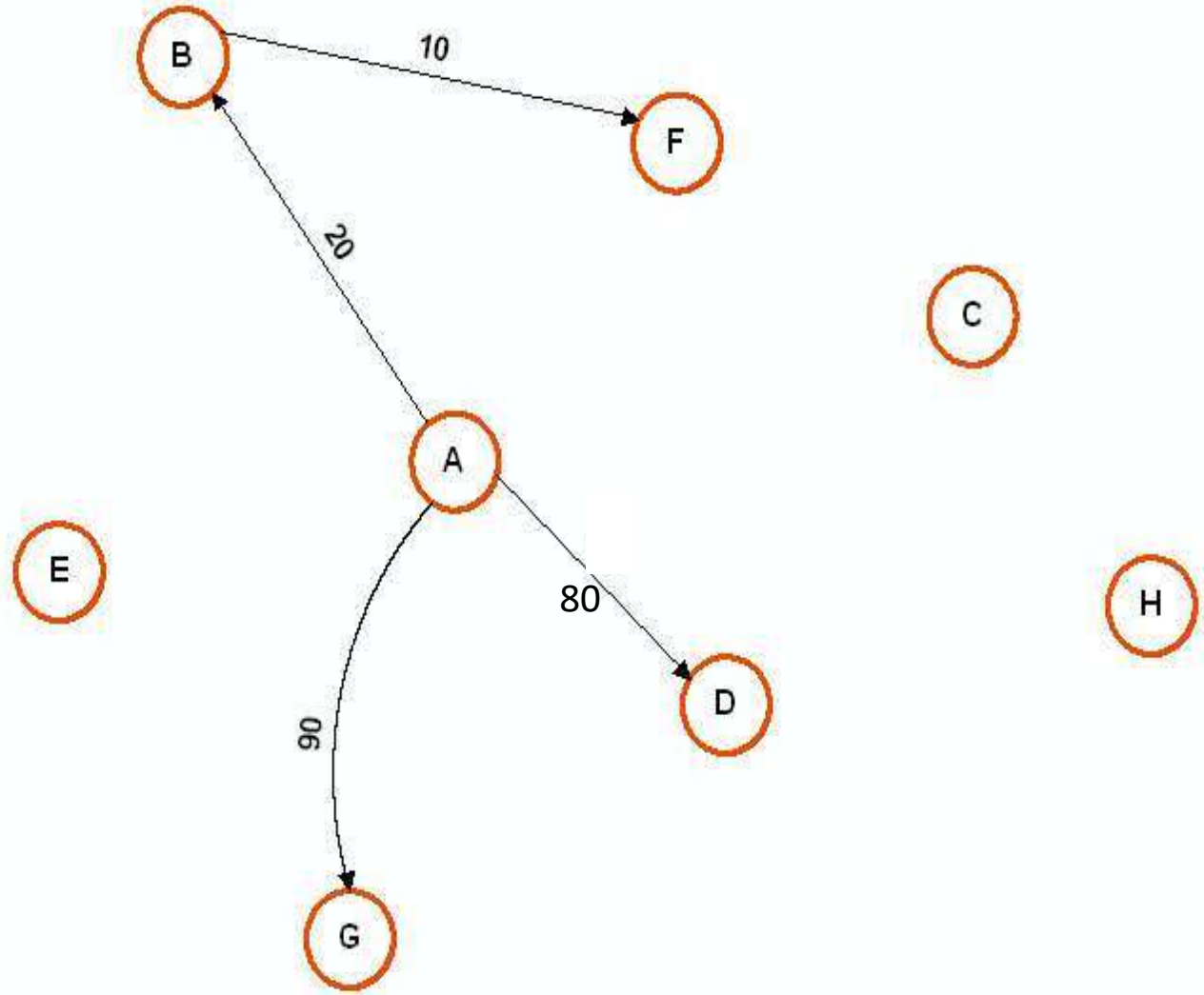
From	A → B	C	D	E	F	G	H
1							
2							
3							
4							
5							
6							
7							
8							



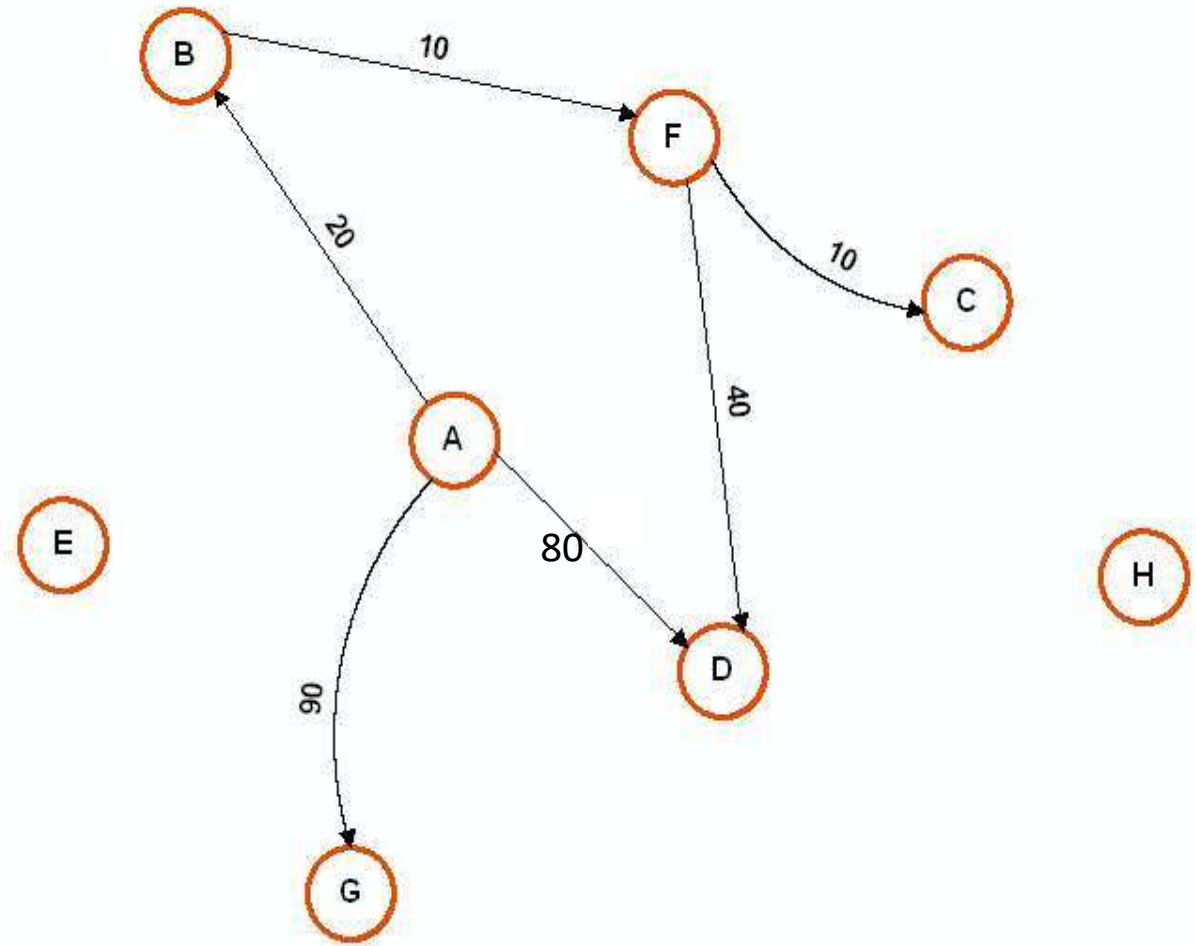
	From A → B	C	D	E	F	G	H
1	20		80			90	
	A	∞	A	∞	∞	A	∞
2							
3							
4							
5							
6							
7							
8							



	From A → B	C	D	E	F	G	H
1	20		80			90	
	A	∞	A	∞	∞	A	∞
2	20		80		30	90	
	B	∞	A	∞	B	A	∞
3							
	F						
4							
5							
6							
7							
8							

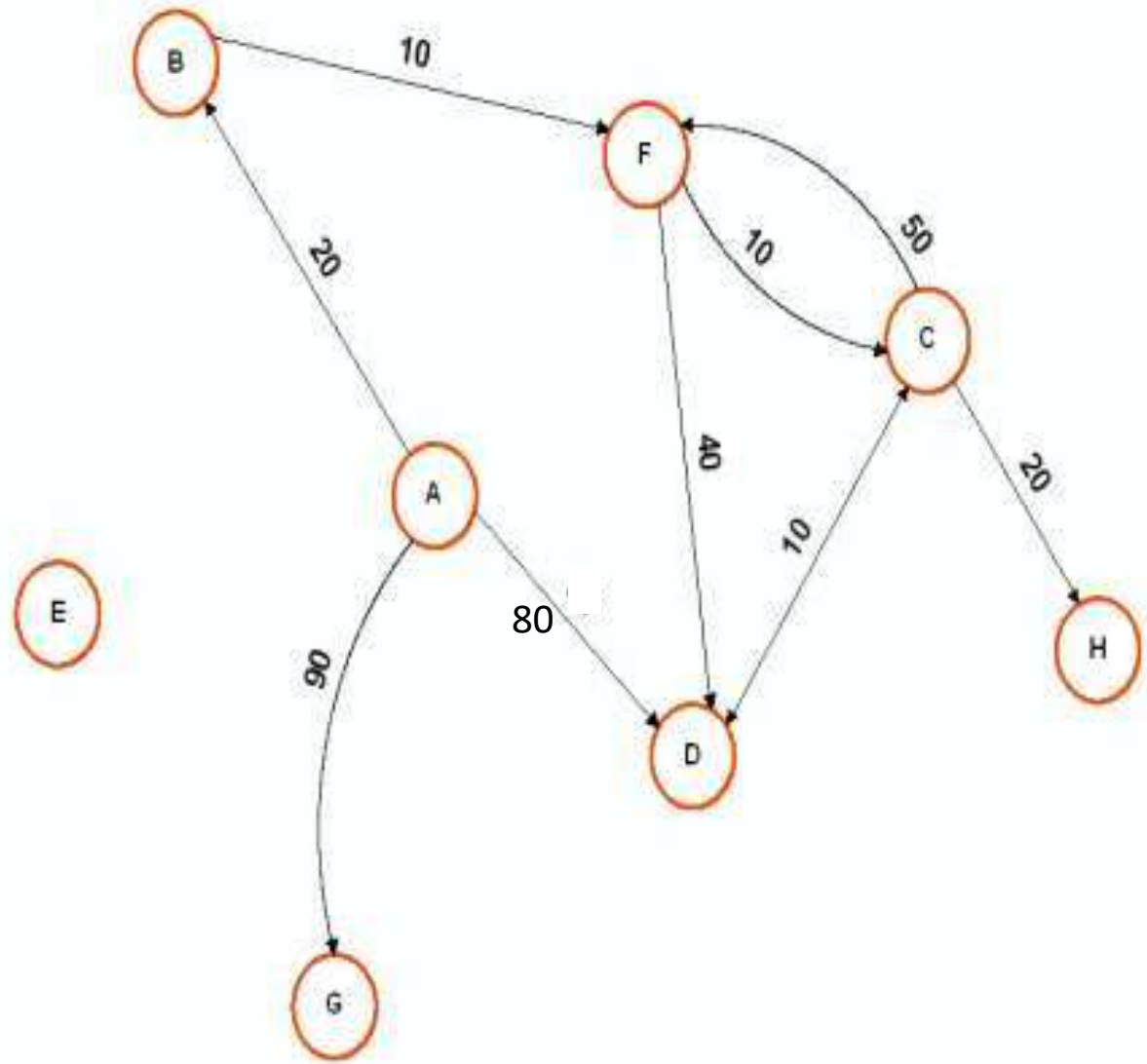


	From A → B	C	D	E	F	G	H
1	20		80			90	
	A	∞	A	∞	∞	A	∞
2	20		80		30	90	
	B	A	∞	A	∞	B	A	∞
3	20	40	70		30	90	
	F	A	F	F	∞	B	A	∞
4							
	C						
5							
6							
7							
8							



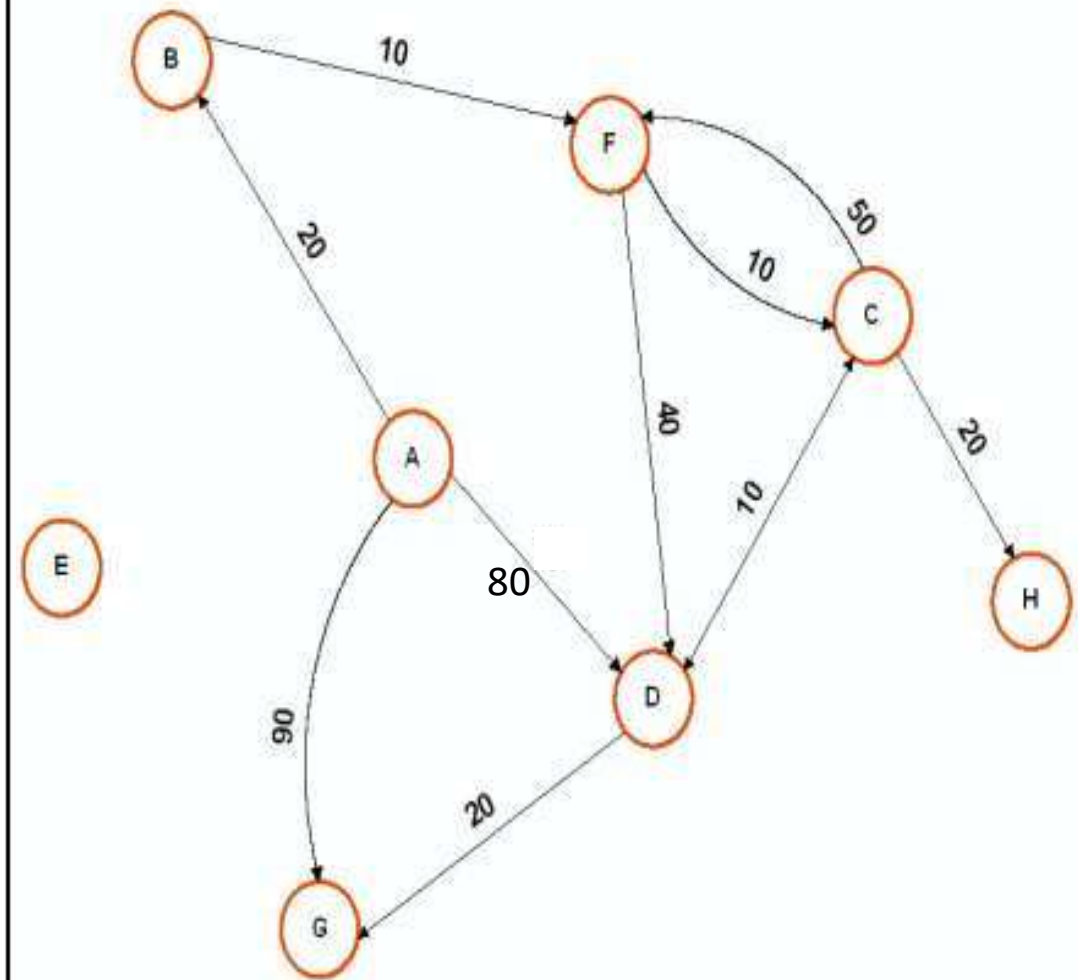
From

	A →	B	C	D	E	F	G	H
1		20		80			90	
	A	A	∞	A	∞	∞	A	∞
2		20		80		30	90	
	B	A	∞	A	∞	B	A	∞
3		20	40	70		30	90	
	F	A	F	F	∞	B	A	∞
4		20	40	50		30	90	60
	C	A	F	C	∞	B	A	C
5								
	D							
6								
7								
8								



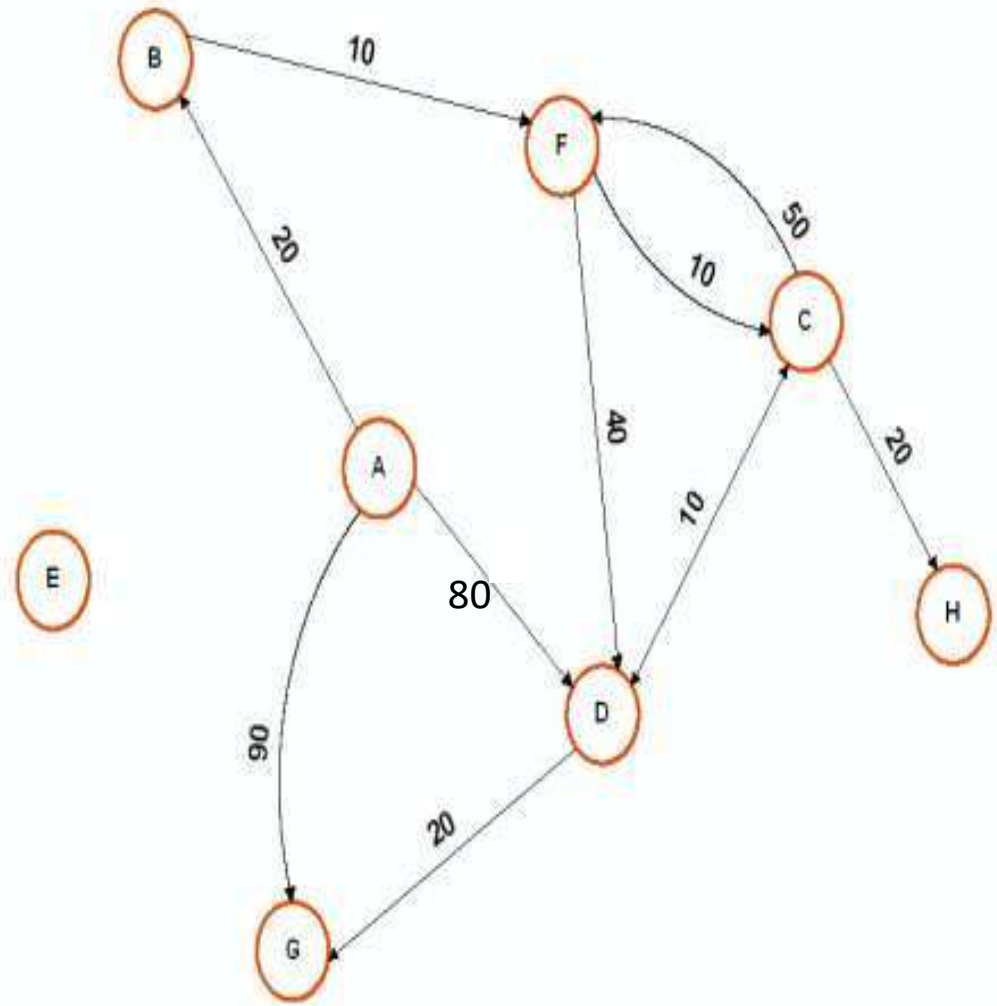
From

	A →	B	C	D	E	F	G	H
1		20		80			90	
	A	A	∞	A	∞	∞	A	∞
2		20		80		30	90	
	B	A	∞	A	∞	B	A	∞
3		20	40	70		30	90	
	F	A	F	F	∞	B	A	∞
4		20	40	50		30	90	60
	C	A	F	C	∞	B	A	C
5		20	40	50		30	70	60
	D	A	F	C	∞	B	D	C
6								
	H							
7								
8								

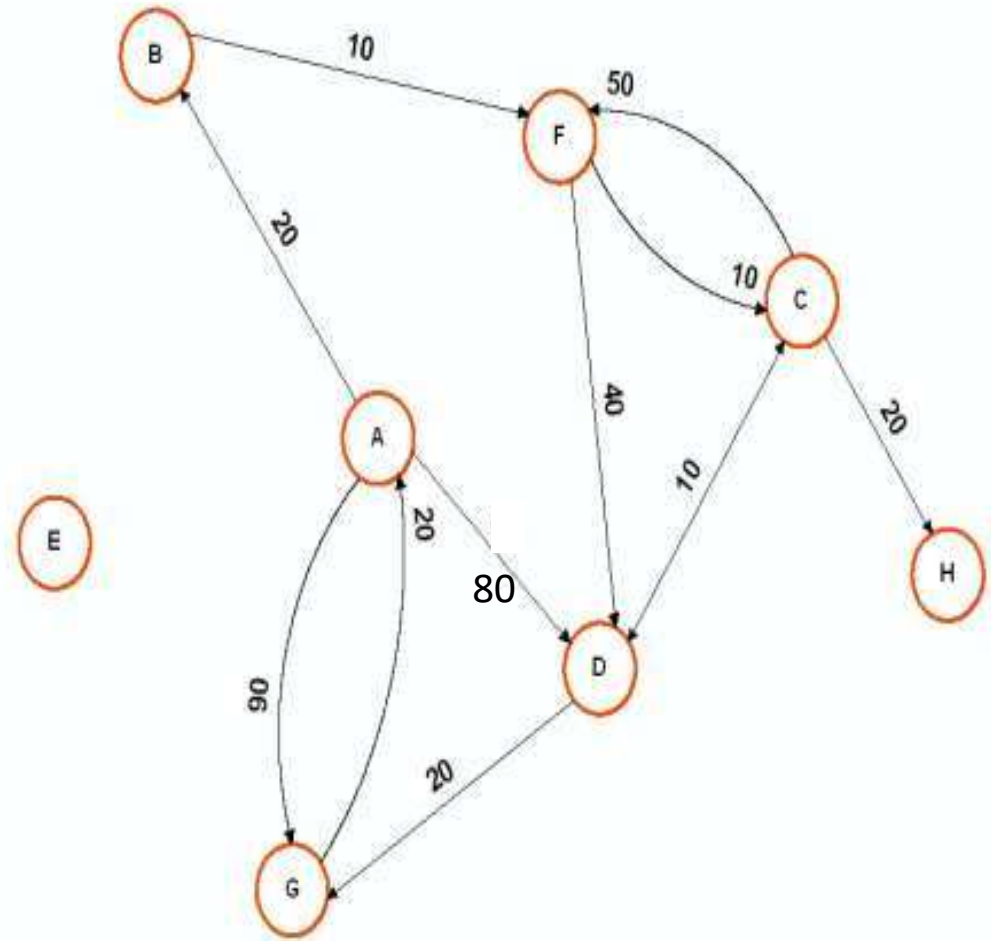


From

	A →	B	C	D	E	F	G	H
1		20		80			90	
	A	A	∞	A	∞	∞	A	∞
2		20		80		30	90	
	B	A	∞	A	∞	B	A	∞
3		20	40	70		30	90	
	F	A	F	F	∞	B	A	∞
4		20	40	50		30	90	60
	C	A	F	C	∞	B	A	C
5		20	40	50		30	70	60
	D	A	F	C	∞	B	D	C
6		20	40	50		30	70	60
	H	A	F	C	∞	B	D	C
7								
	G							
8								



	From A → B	C	D	E	F	G	H
1	20		80			90	
	A	∞	A	∞	∞	A	∞
2	20		80		30	90	
	B	A	∞	A	∞	B	A	∞
3	20	40	70		30	90	
	F	A	F	F	∞	B	A	∞
4	20	40	50		30	90	60
	C	A	F	C	∞	B	A	C
5	20	40	50		30	70	60
	D	A	F	C	∞	B	D	C
6	20	40	50		30	70	60
	H	A	F	C	∞	B	D	C
7	20	40	50		30	70	60
	G	A	F	C	∞	B	D	C
8							



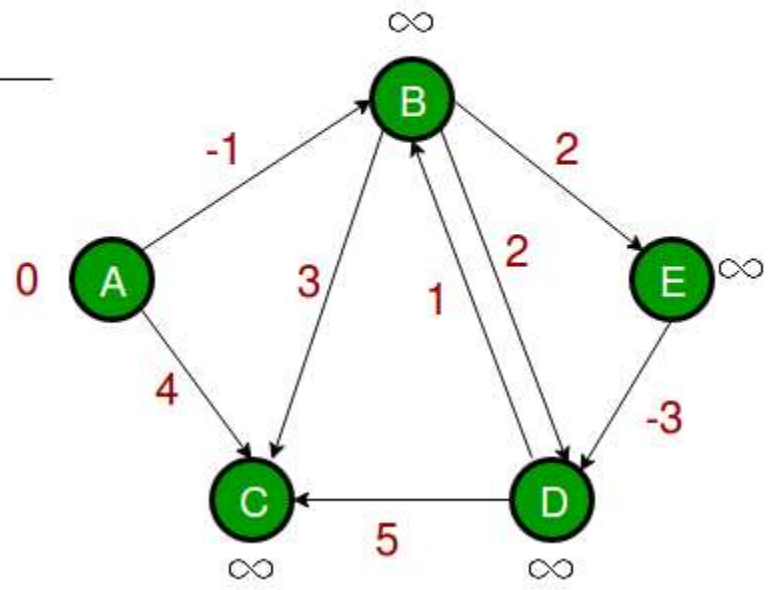
- So with this ‘Graph Algorithm’ we found our best lowest cost route in this interconnected Vertex.
- And the best lowest cost path is given below:
  A→B → F→C → D→(H)→G
- So total cost from ‘A’ to ‘G’ vertex is ‘70’ which is lowest cost from other Vertex.

Dijkstra Algorithm

```
1:function Dijkstra(Graph, source):
2:       for each vertex v in Graph
3:                dist[v] := infinity
4:                previous[v] := undefined
5:       dist[source] := 0       // Distance from source to source
6:       Q := the set of all nodes in Graph
7:       while Q is not empty:
8:                u := node in Q with smallest dist[ ]
9:                remove u from Q
10:               for each neighbor v of u:
11:                       alt := dist[u] + dist_between(u, v)
12:                       if alt < dist[v]          // Relax (u,v)
13:                       dist[v] := alt
14:                       previous[v] := u
15:      return previous[ ]
```

Bellman Ford Algorithm

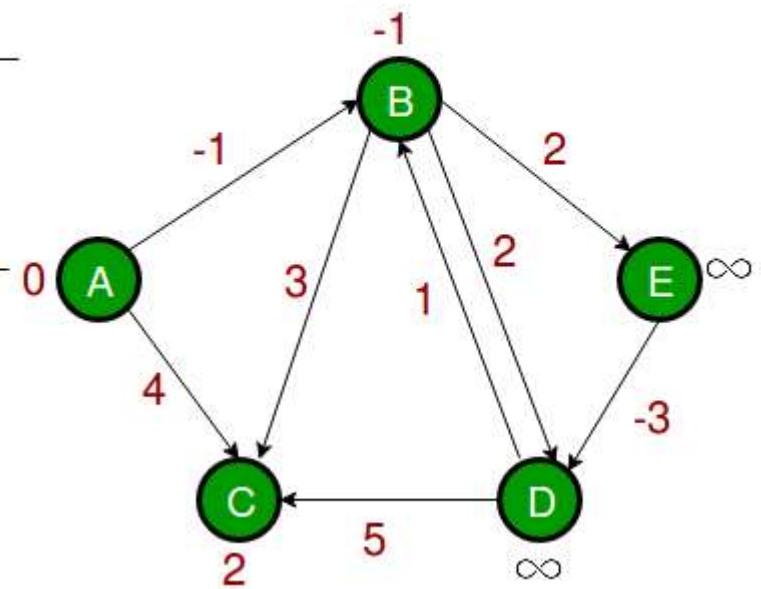
A	B	C	D	E
0	∞	∞	∞	∞



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

Bellman Ford Algorithm

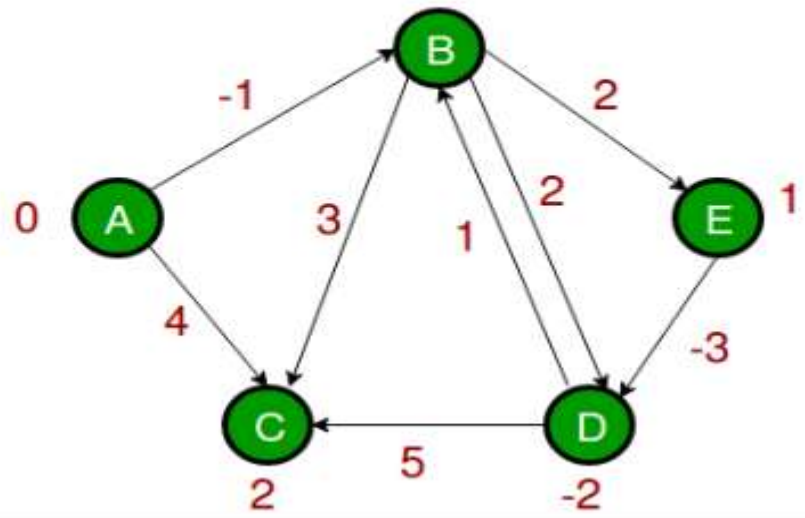
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

Bellman Ford Algorithm

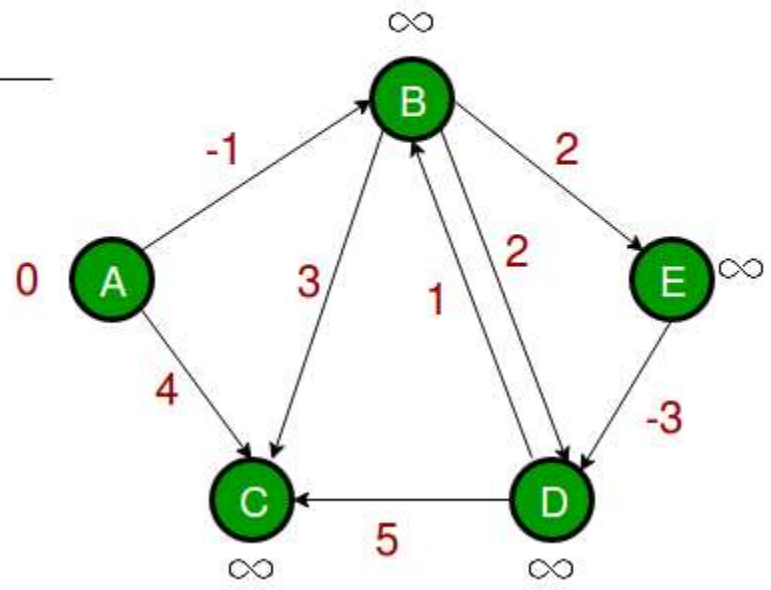
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

Bellman Ford Algorithm

A	B	C	D	E
0	∞	∞	∞	∞



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

Bellman Ford Algorithm

- Benefit:
  - It manages negative weights as well.
- Drawback:
  - It does not manage loops.

Algorithm 1: Bellman Ford Algorithm

$\forall v \in V, d[v] \leftarrow \infty$ // set initial distance estimates
//optional: set $\pi(v) \leftarrow$nil for all v, $\pi(v)$ represents the predecessor of v
$d[s] \leftarrow 0$ // set distance to start node trivially as 0
for i *from* $1 \rightarrow n-1$ **do**
    for $(u, v) \in E$ **do**
        $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$ // update estimate of v
        // optional - if $d[v]$ changes, then $\pi(v) \leftarrow u$
// Negative Cycle Step
for $(u, v) \in E$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
        return "Negative Cycle"; // negative cycle detected
return $d[v] \; \forall \; v \in V$