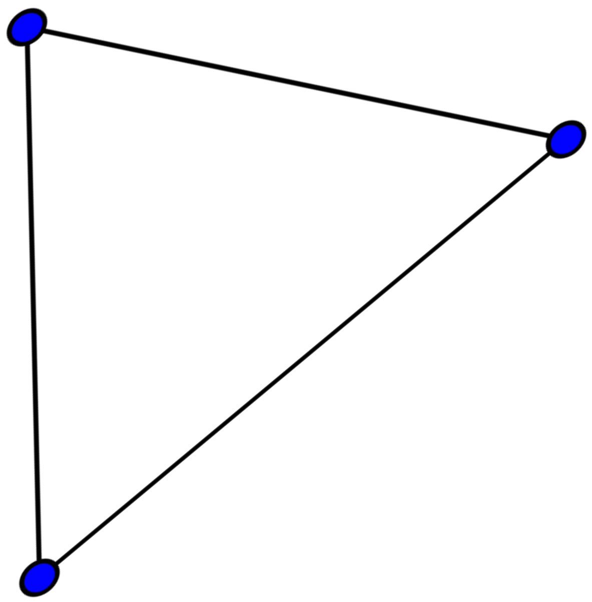
Graphs and Dijkstra’s Shortest Path Algorithm

By

Arun Cyril Jose

Graphs

- Like Trees it is also a non-Linear Data Structure.
- Set of Vertices and Edges.
- Graphs Vs Trees



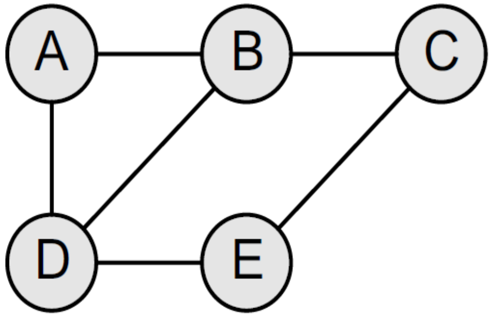
Some Applications of Graphs

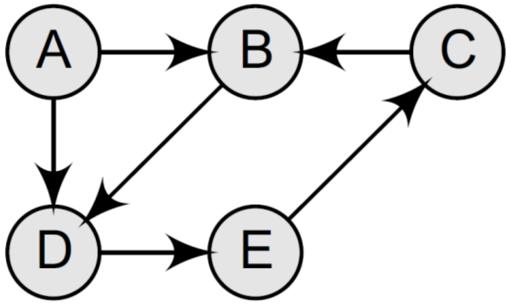
- Airline Scheduling
- Computer Networks and Routing
- Directions in a map
- Plumbing/Hydraulic Systems
- Electrical Circuits.

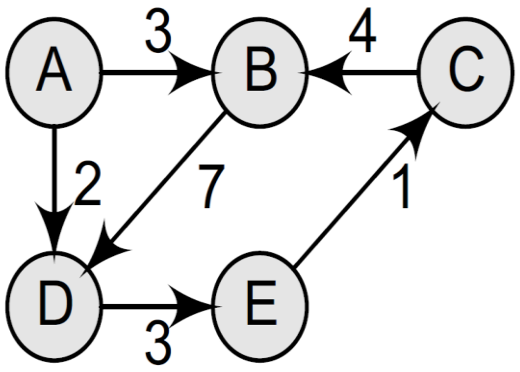
Graphs

- Used for Representing many-to-many relationships

- Two Forms:
  - **Directed** – Finite set of elements called Vertices (nodes) connected by a set of directed edges (arcs).
  - G = {V} {E}
    - V = {V_1, V_2, V_3 …. V_n} E = { (V_{m1}, V_{n1}), (V_{m2}, V_{n2}) …. (V_{mx}, V_{ny}) }
  - **Undirected** – Finite set of vertices connected by a set of undirected (bidirectional) edges.

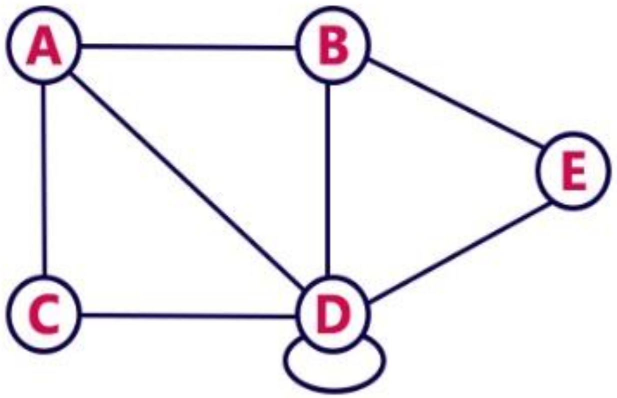






Graph Representations

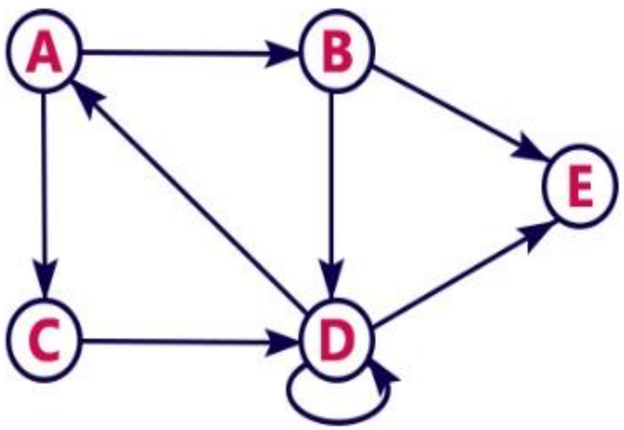
- **Adjacency Matrix**



$$
\begin{array}{c c}
 & \begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

Graph Representations

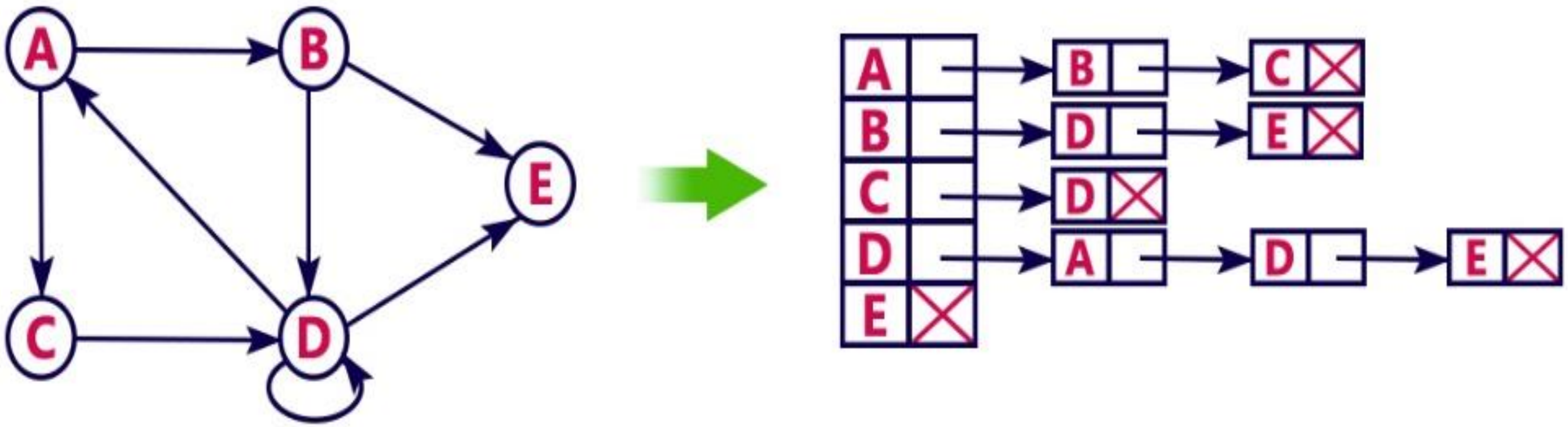
- **Adjacency Matrix**



$$
\begin{array}{c c}
 & \begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

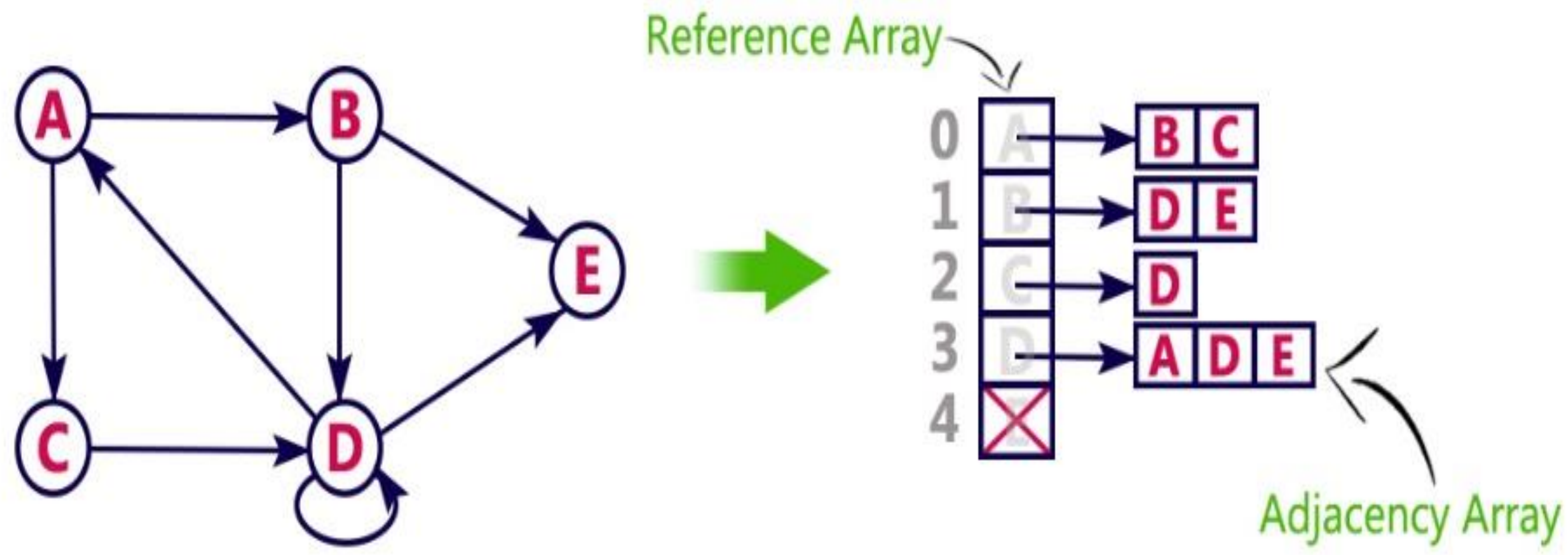
Graph Representations

- **Adjacency List**

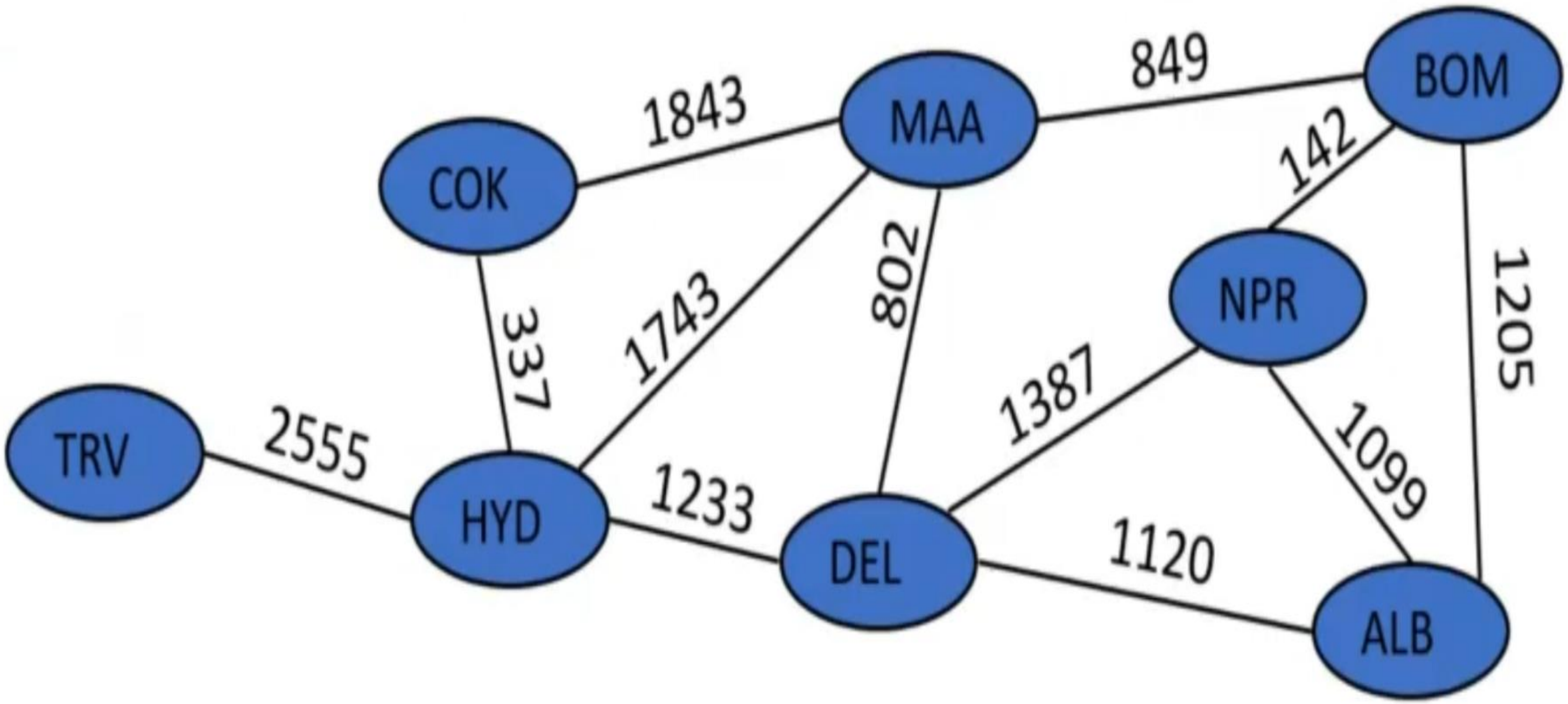


Graph Representations

- **Adjacency List**

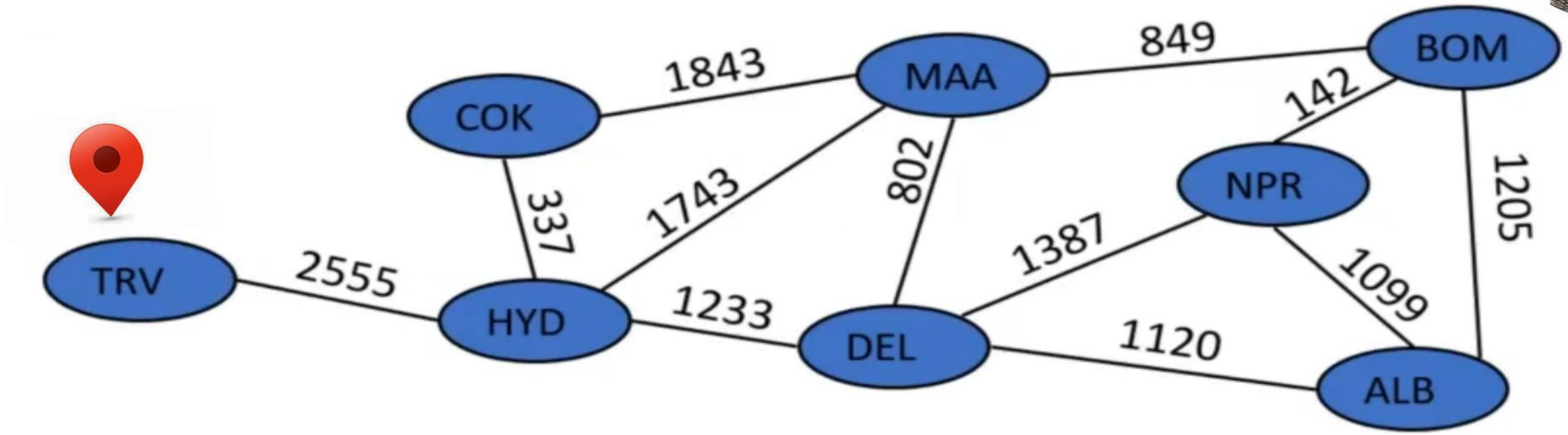


Weighted Graphs



Shortest Path Problem

- In the weighted graph and given two vertices **u** and **v**, we want to find a path of minimum total weight between **u** and **v**.
  - Length of the path is the sum of weights of its edges.
- Applications
  - Internet packet Routing, Flight reservations, Driving directions.

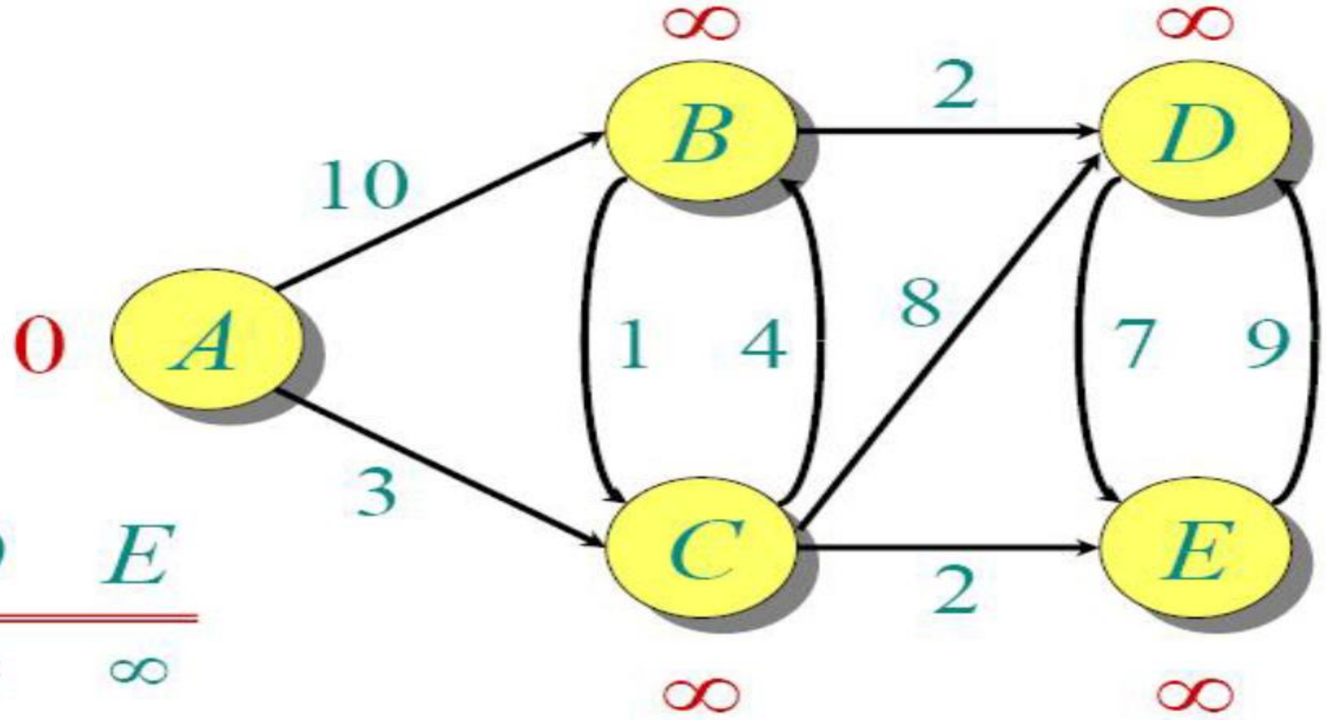


Why Shortest Path algorithms?

- Google Maps.
- Uber Ride Sharing.
- Routing computer Network.
- Almost all Logistical problems

Dijkstra’s Shortest Path Algorithm

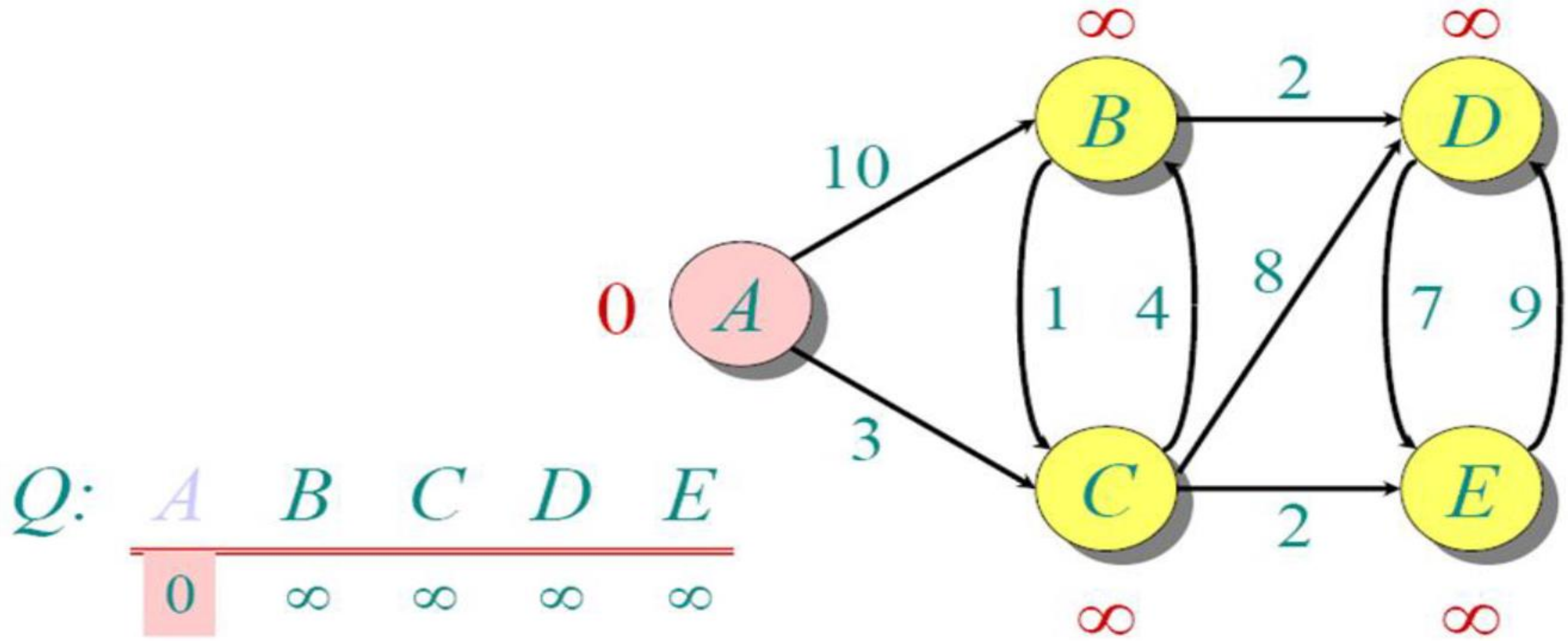
Initialize:



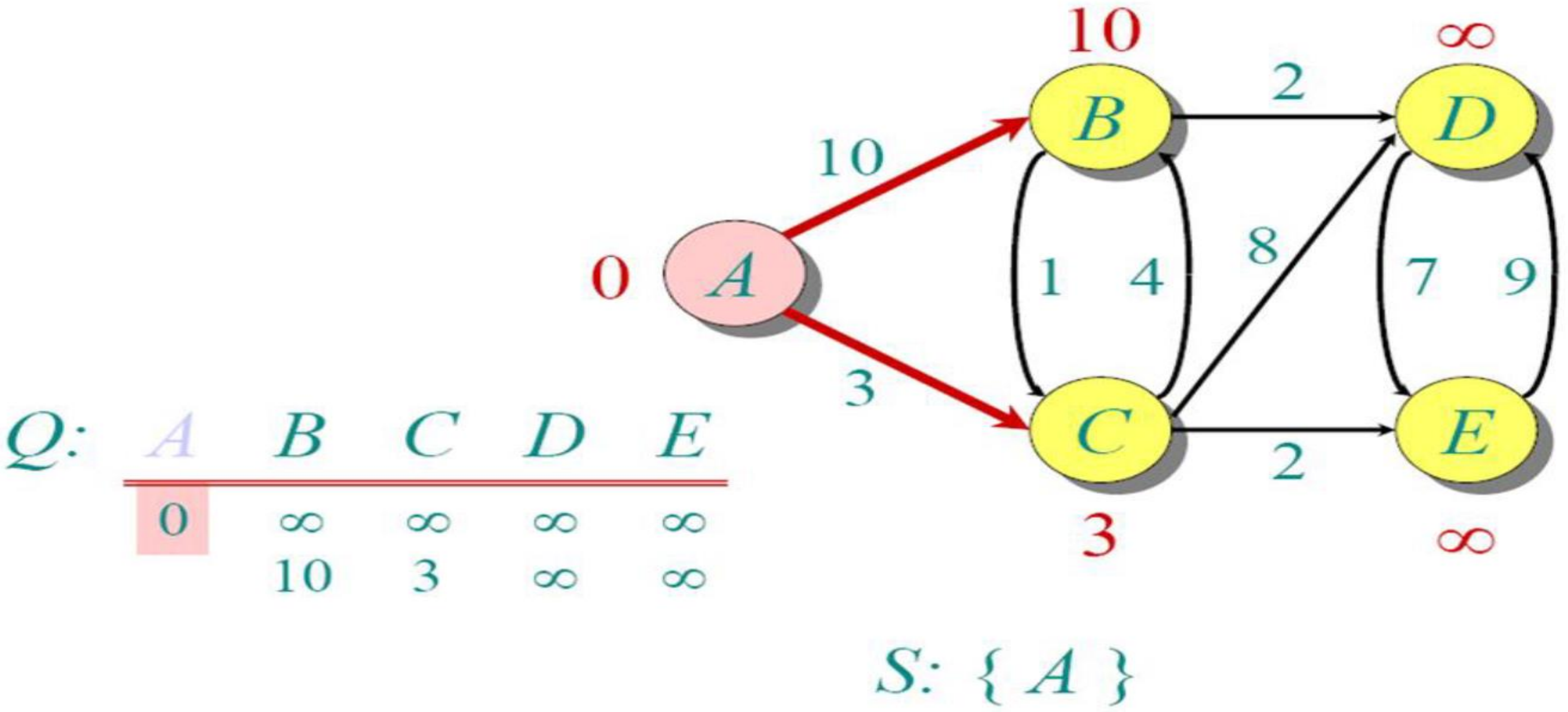
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

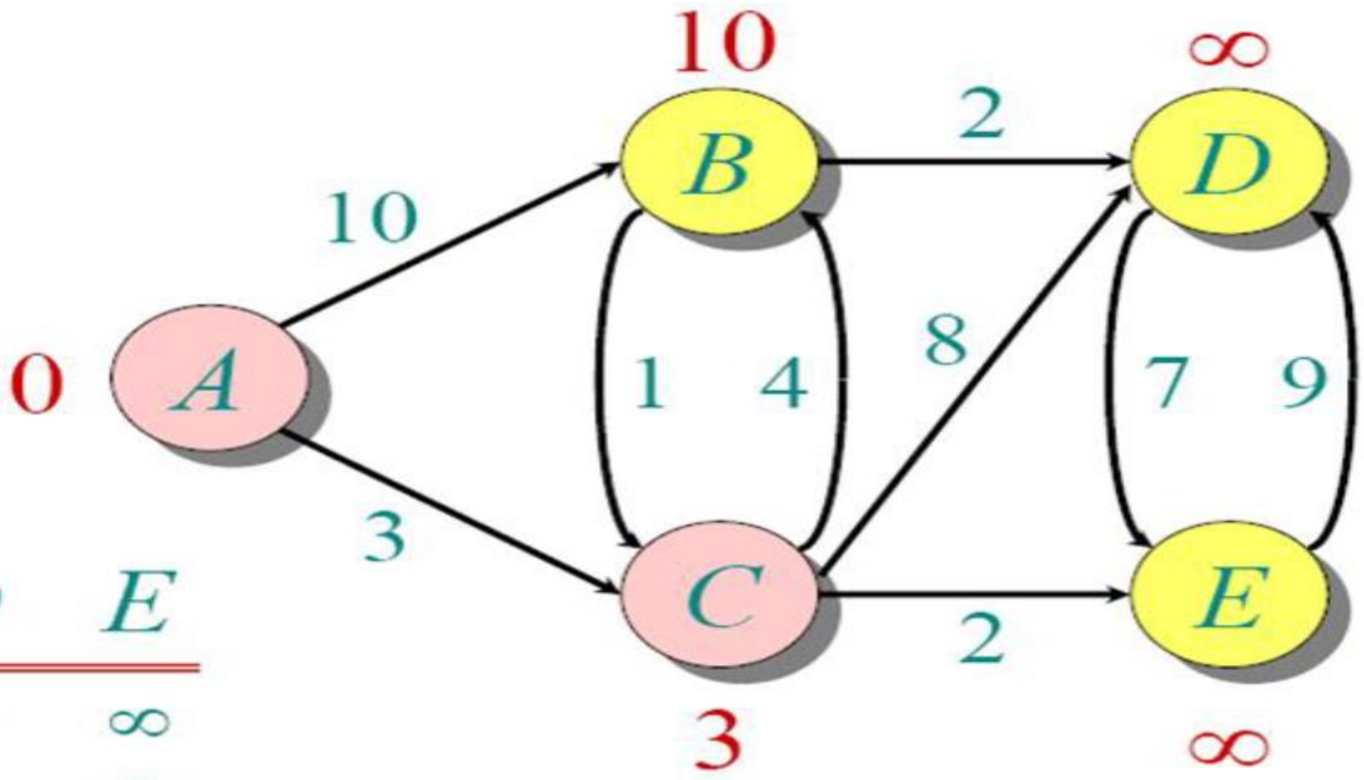
Dijkstra’s Shortest Path Algorithm



Dijkstra’s Shortest Path Algorithm



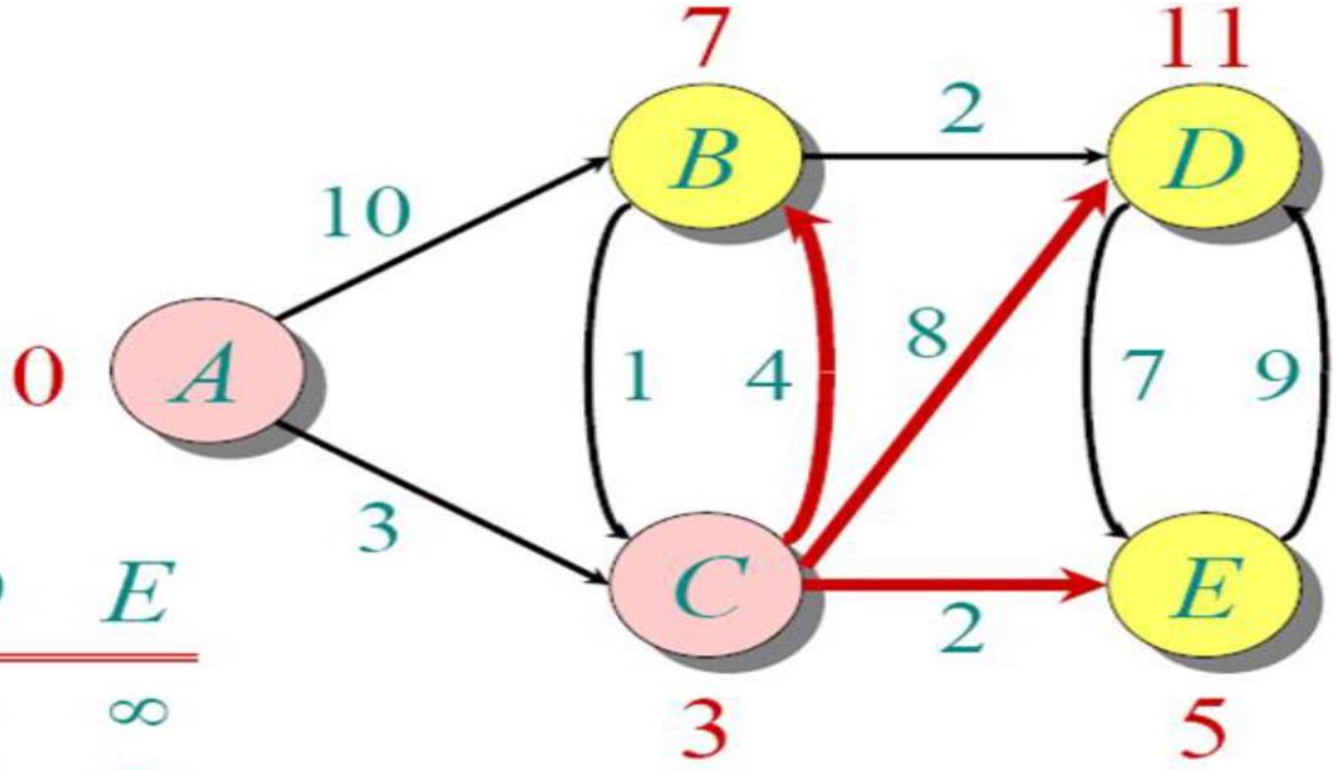
Dijkstra’s Shortest Path Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

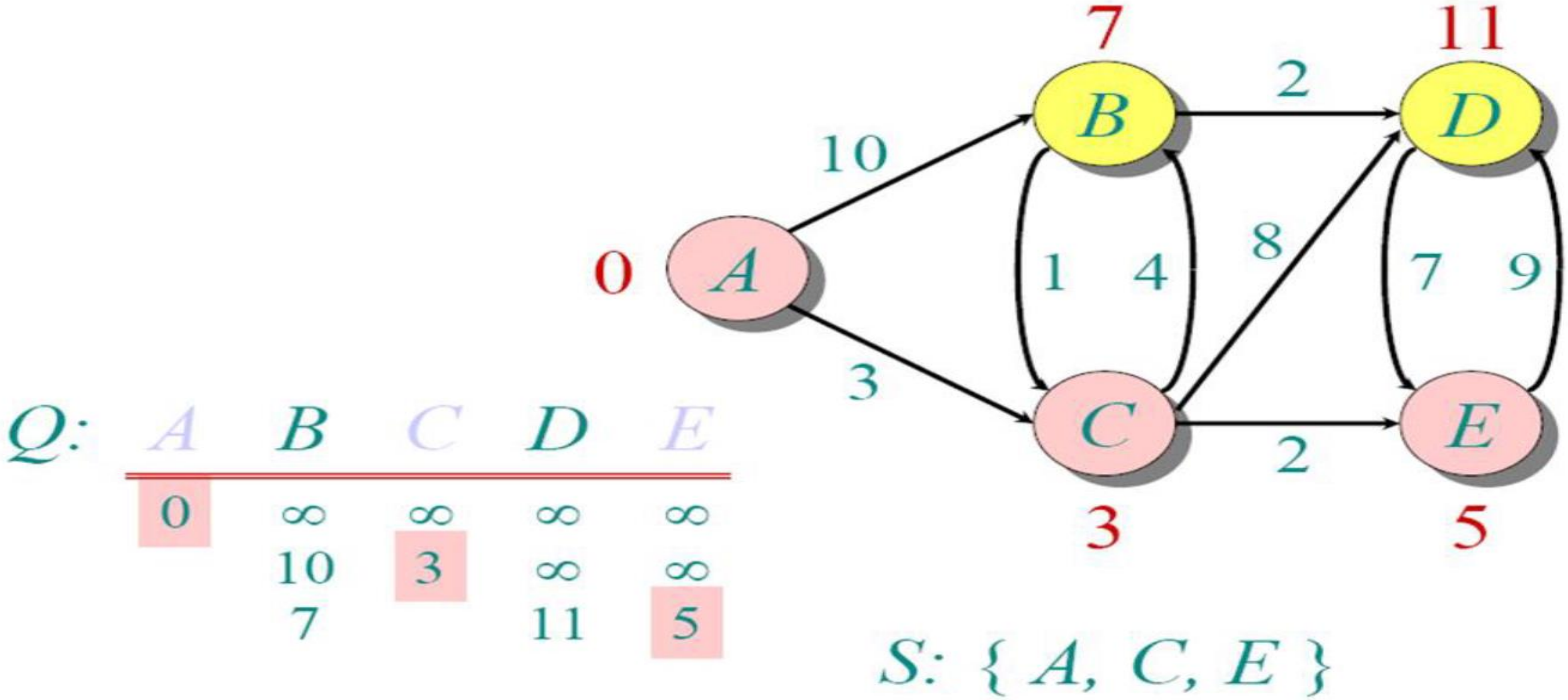
Dijkstra’s Shortest Path Algorithm



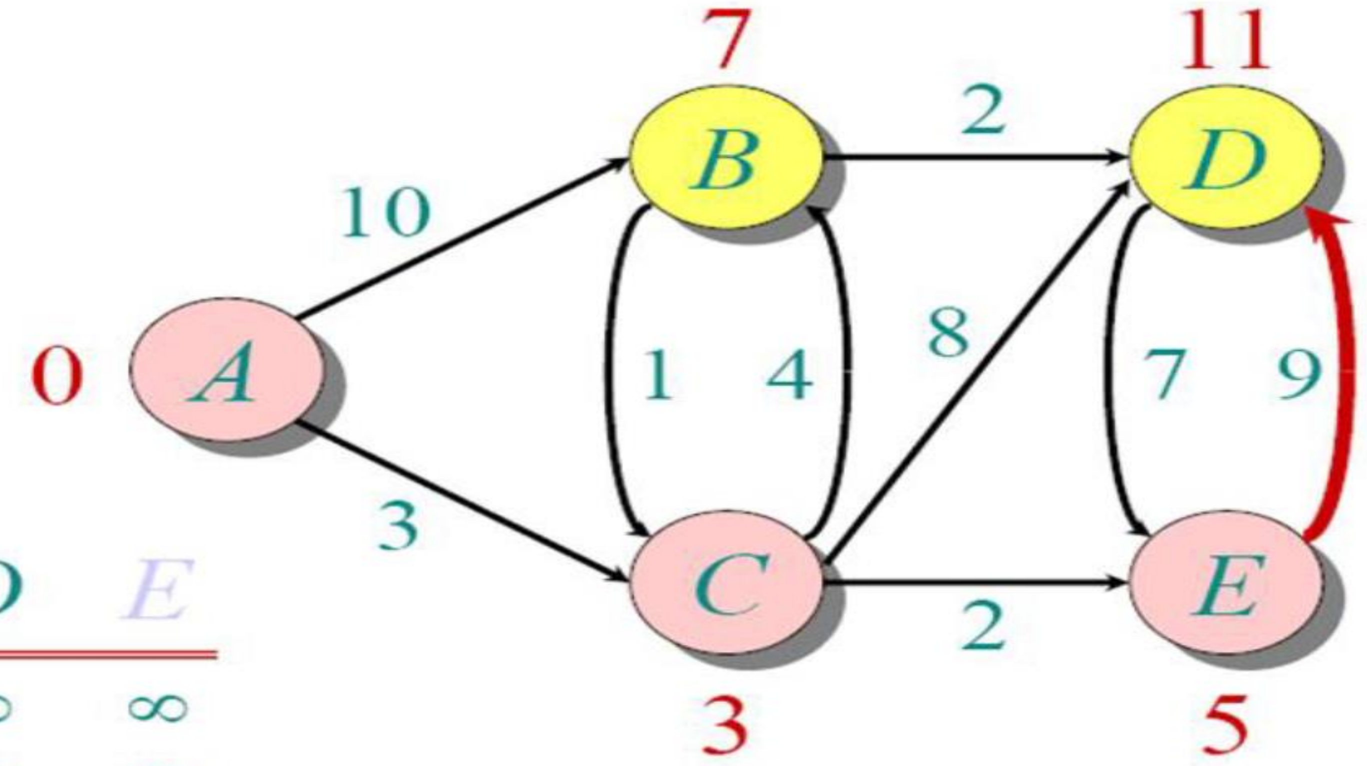
Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	5

S: { A, C }

Dijkstra’s Shortest Path Algorithm



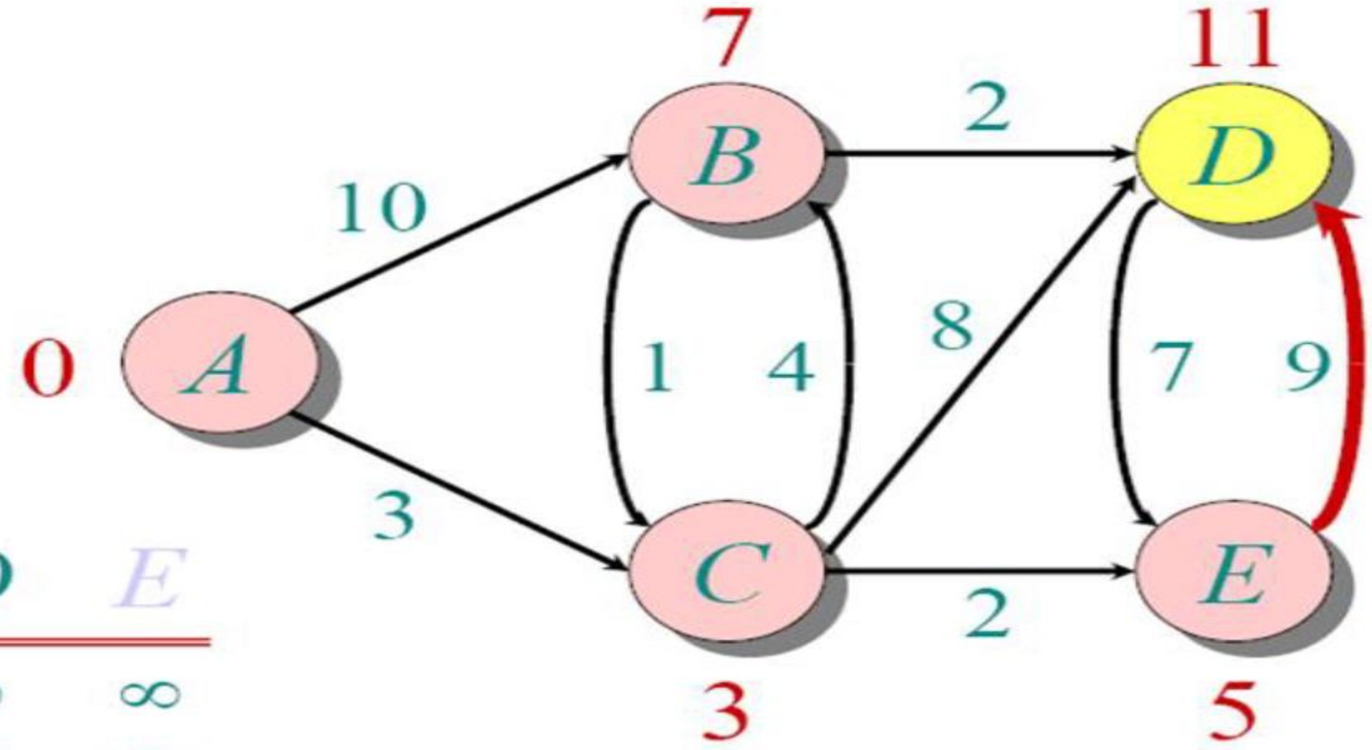
Dijkstra’s Shortest Path Algorithm



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		7		11	

S: { A, C, E }

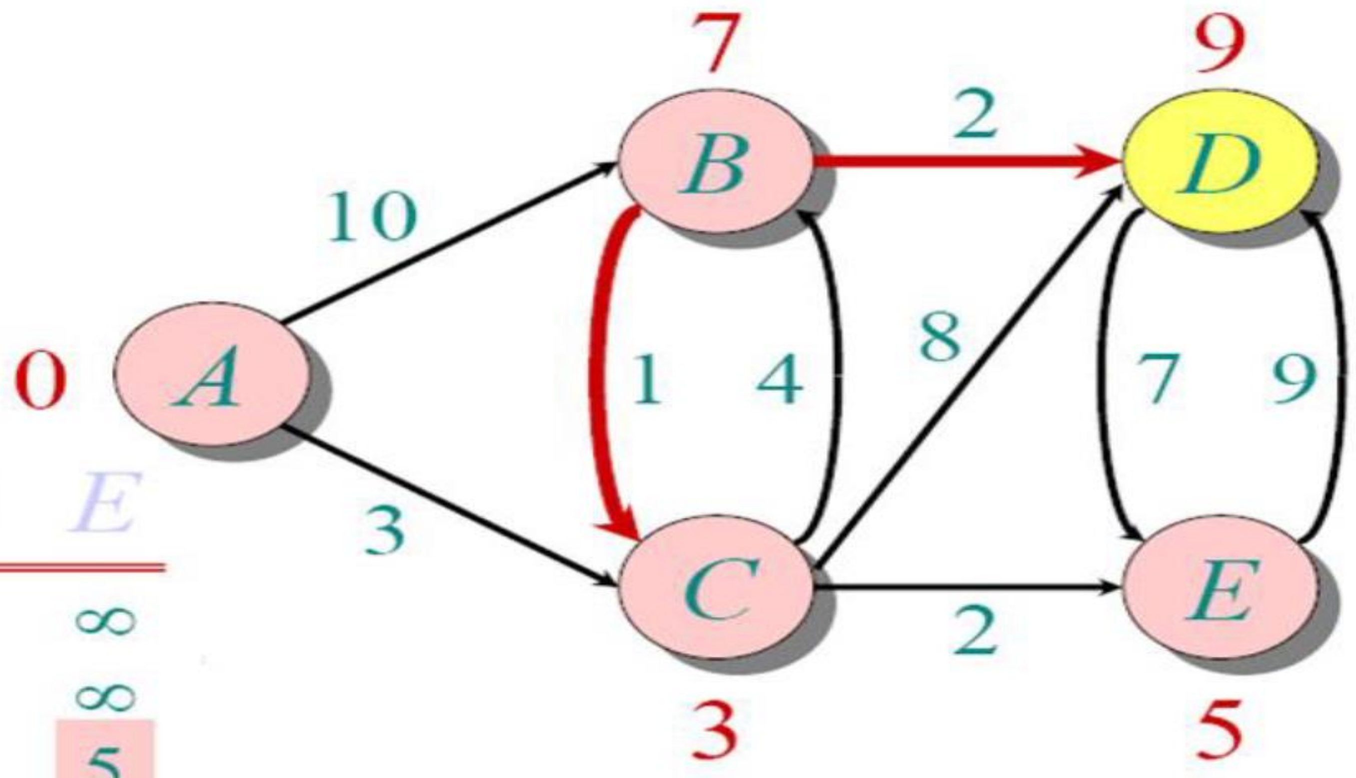
Dijkstra’s Shortest Path Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

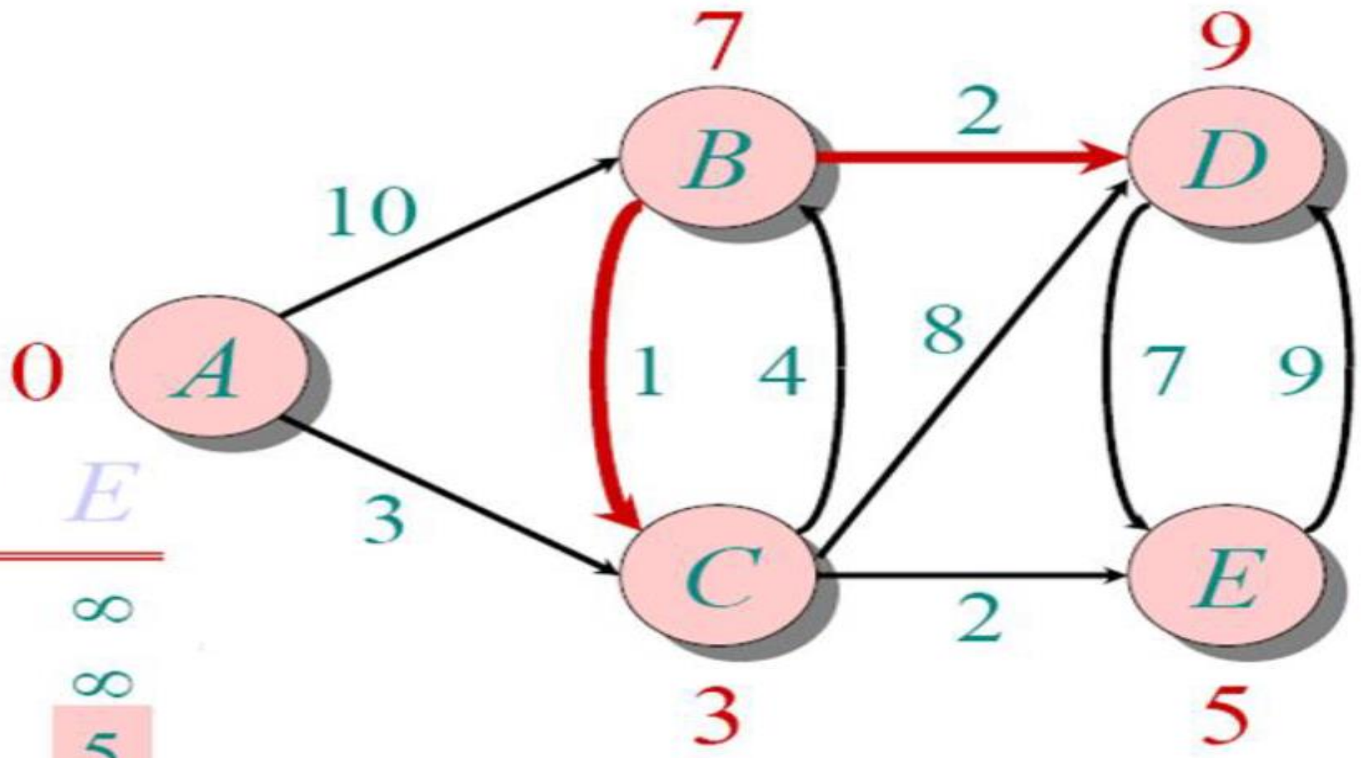
Dijkstra’s Shortest Path Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra’s Shortest Path Algorithm



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S: { A, C, E, B, D }

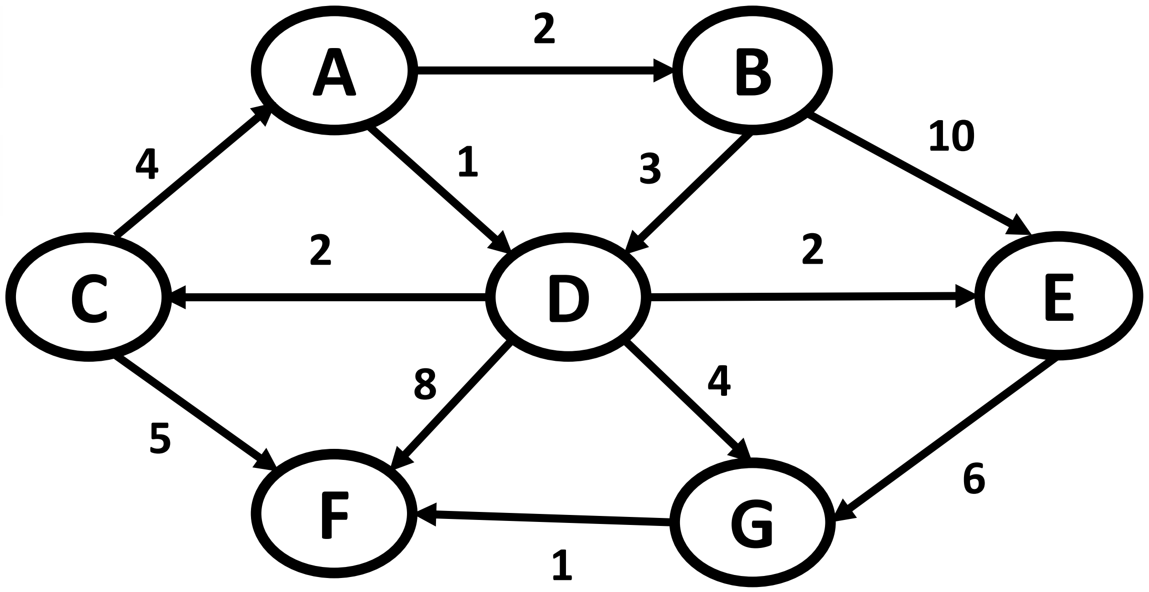
Dijkstra's Shortest Path Algorithm

```
function dijkstra(v1, v2):
   for each vertex v:                     // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost.
      mark v as visited.
      for each unvisited neighbor n of v:
         cost := v's cost + weight of edge (v, n).

         if cost < n's cost:
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1, following previous pointers.
```

Relax Operation

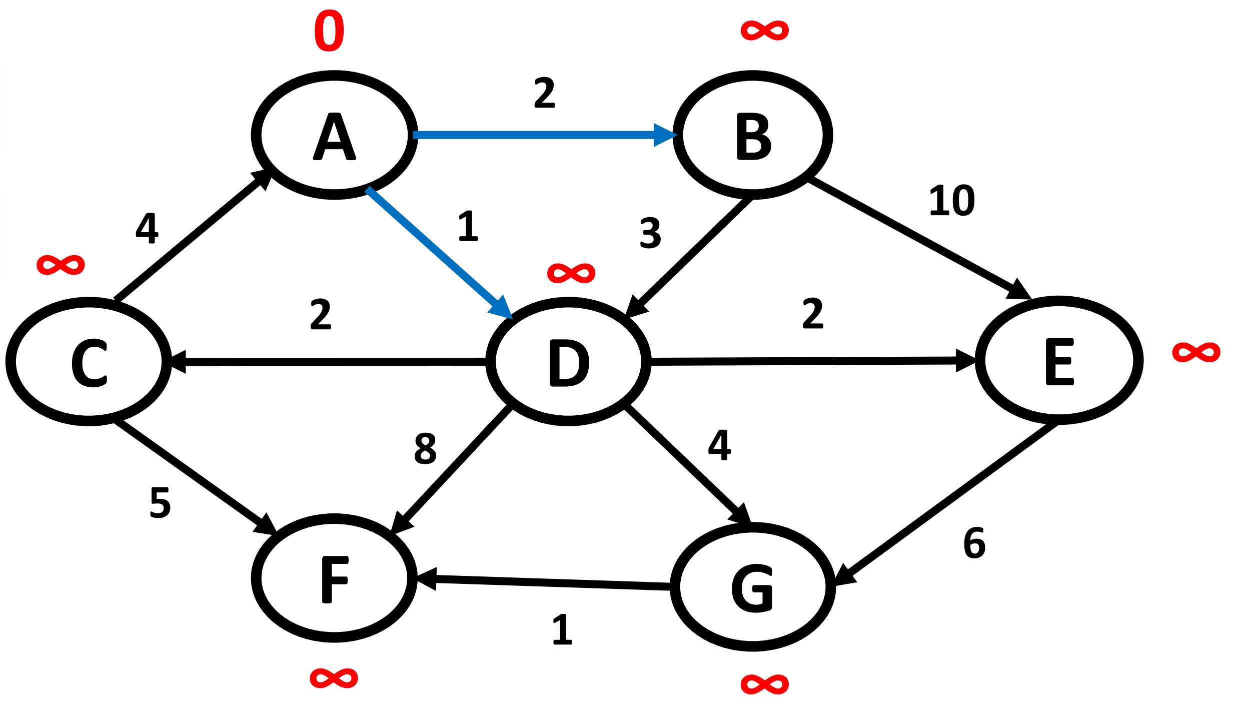
Dijkstra's Shortest Path Algorithm

```
function dijkstra(v1, v2):
   for each vertex v:                    // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost.
      mark v as visited.
      for each unvisited neighbor n of v:
         cost := v's cost + weight of edge (v, n).

         if cost < n's cost:
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1, following previous pointers.
```



Dijkstra's Shortest Path Algorithm

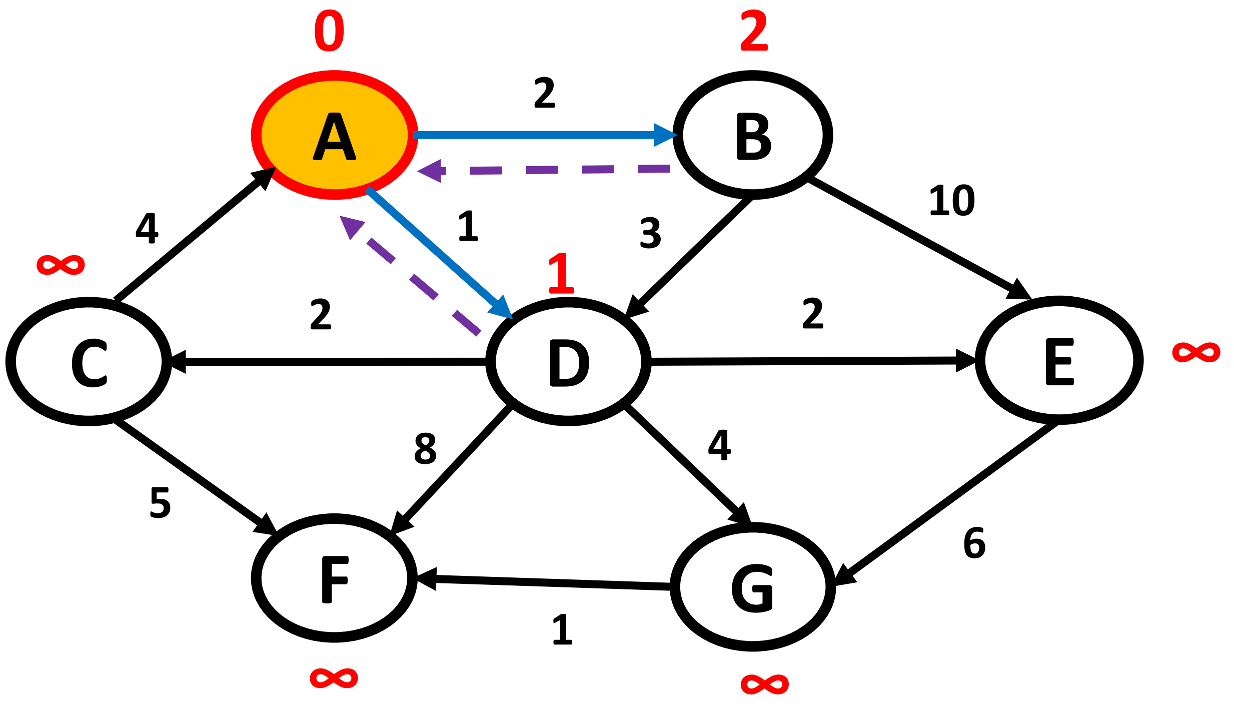
```
function dijkstra(v1, v2):
   for each vertex v:                  // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost. // A
      mark v as visited.
      for each unvisited neighbor n of v:   // B, D
         cost := v's cost + weight of edge (v, n).

         if cost < n's cost:
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞, G: ∞]**

Dijkstra's Shortest Path Algorithm

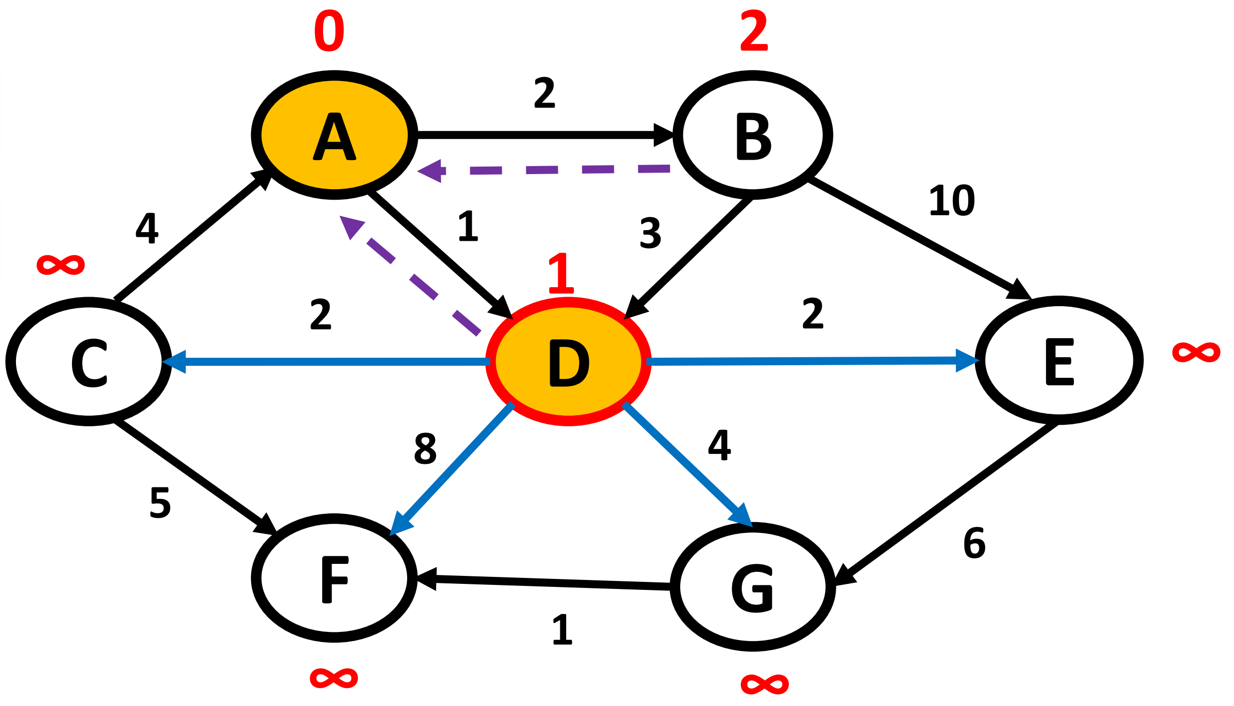
```
function dijkstra(v1, v2):
  for each vertex v:              // Initialize vertex info
     v's cost := infinity.
     v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
     v := remove vertex from pqueue with minimum cost. // A
     mark v as visited.
     for each unvisited neighbor n of v:   // B, D
        cost := v's cost + weight of edge (v, n).

        if cost < n's cost:    // B's cost = 0 + 2
           n's cost := cost.   // D's cost = 0 + 1
           n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[B: 2, C: ∞, D: 1, E: ∞, F: ∞, G: ∞]**

Dijkstra's Shortest Path Algorithm

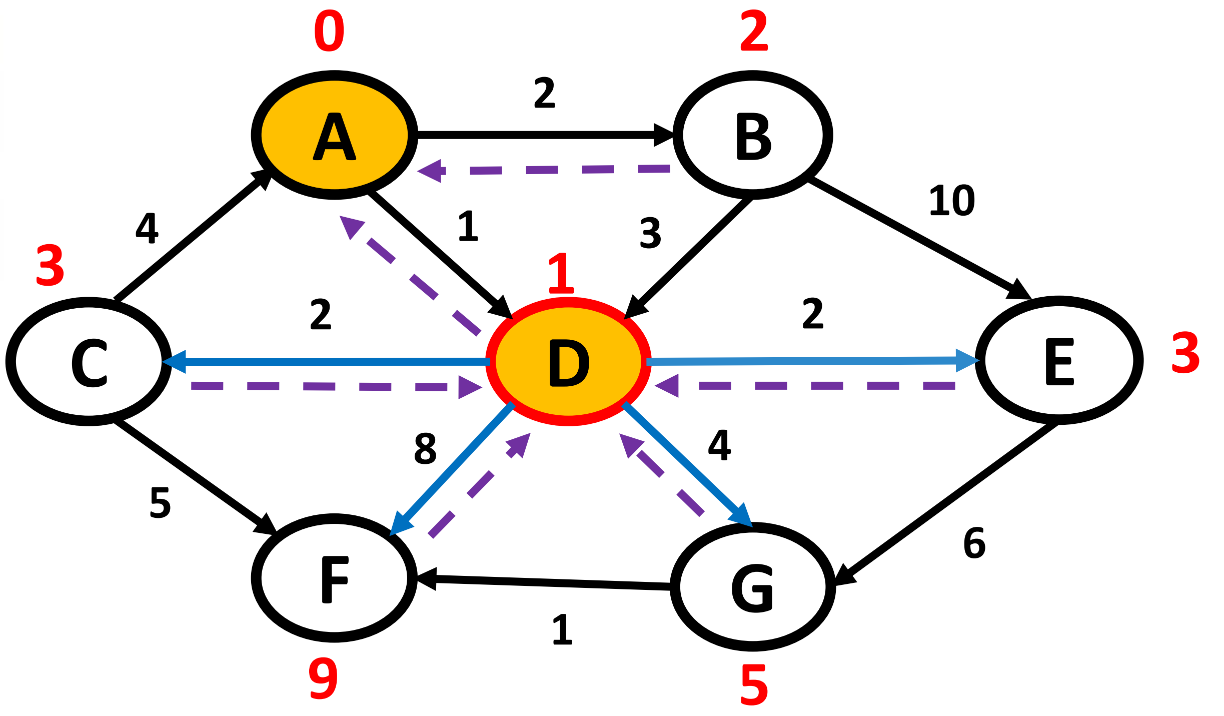
```
function dijkstra(v1, v2):
  for each vertex v:                    // Initialize vertex info
     v's cost := infinity.
     v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
    v := remove vertex from pqueue with minimum cost. // D
    mark v as visited.
    for each unvisited neighbor n of v:   // C, E, F, G
       cost := v's cost + weight of edge (v, n).

       if cost < n's cost:
          n's cost := cost.
          n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[B: 2, C: ∞, E: ∞, F: ∞, G: ∞]**

Dijkstra's Shortest Path Algorithm

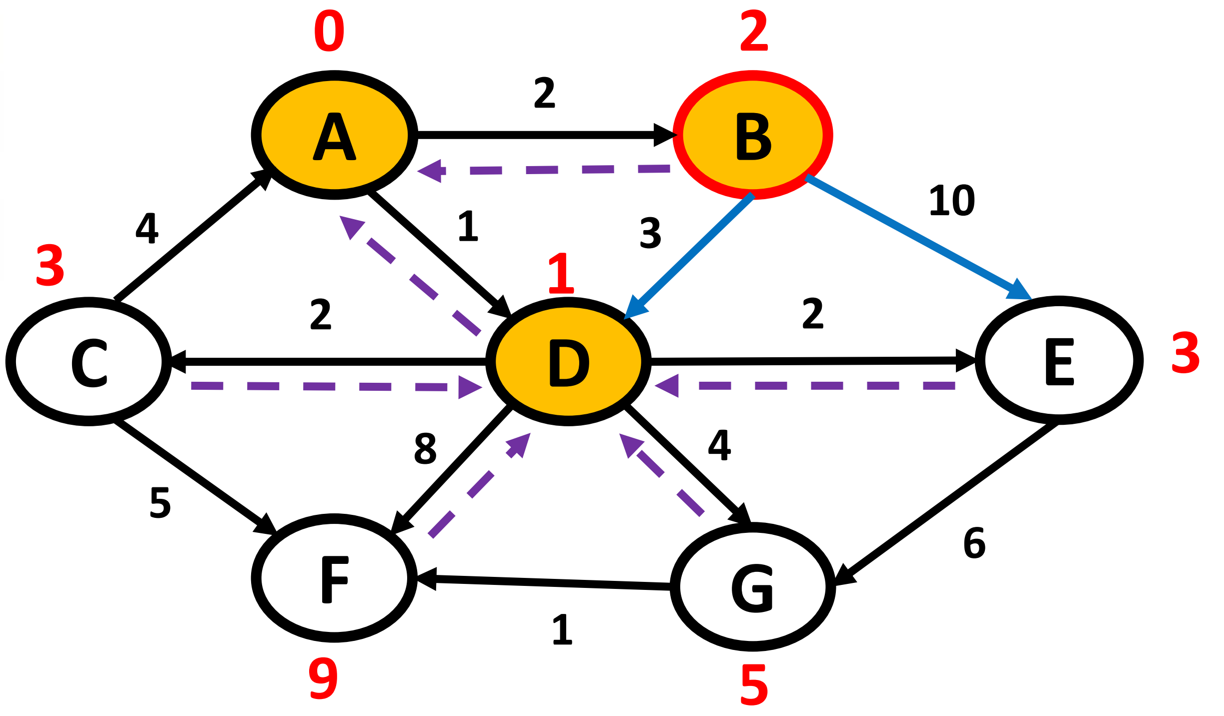
```
function dijkstra(v1, v2):
   for each vertex v:                    // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost. // D
      mark v as visited.
      for each unvisited neighbor n of v:   // C, E, F, G
         cost := v's cost + weight of edge (v, n).
                                    // C's cost = 1 + 2
         if cost < n's cost:        // E's cost = 1 + 2
            n's cost := cost.       // F's cost = 1 + 8
            n's previous := v.      // G's cost = 1 + 4

   reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[B: 2, C: 3, E: 3, F: 9, G: 5]**

Dijkstra's Shortest Path Algorithm

```
function dijkstra(v1, v2):
  for each vertex v:                    // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
     v := remove vertex from pqueue with minimum cost. // B
     mark v as visited.
     for each unvisited neighbor n of v:   // E
        cost := v's cost + weight of edge (v, n).

        if cost < n's cost:
           n's cost := cost.
           n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```

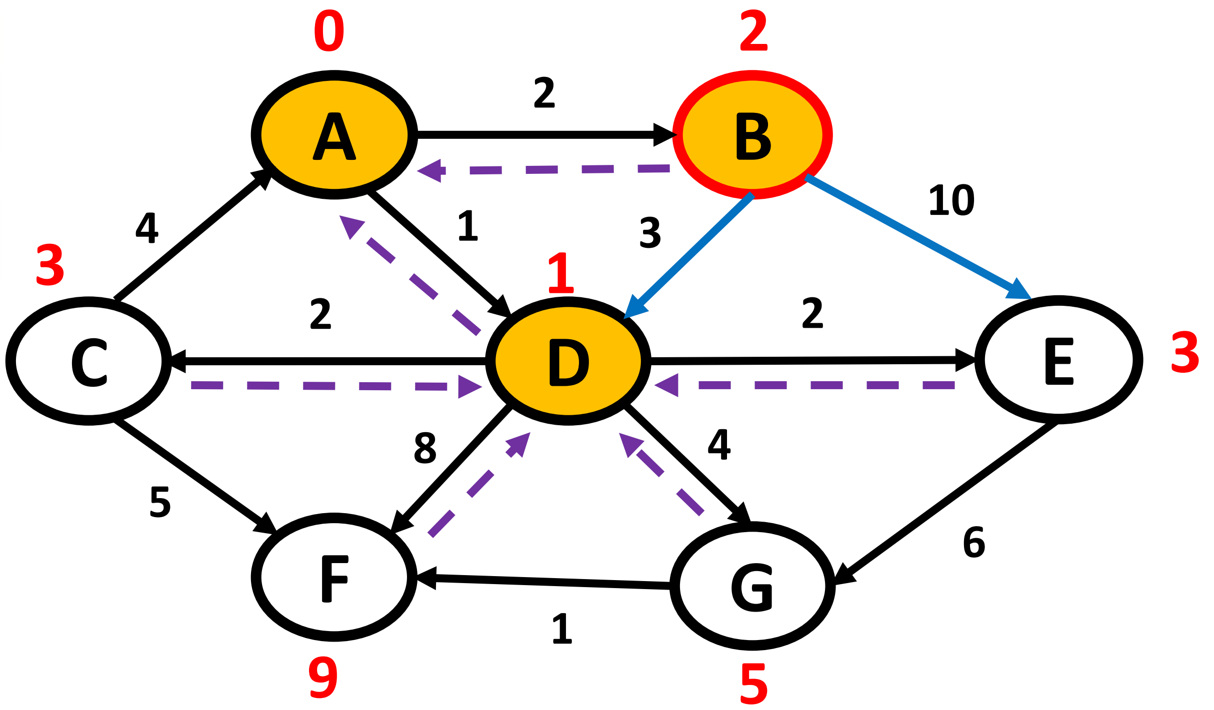


Priority Queue: **[C: 3, E: 3, F: 9, G: 5]**

Indian Institute of Information Technology Kottayam

Dijkstra's Shortest Path Algorithm

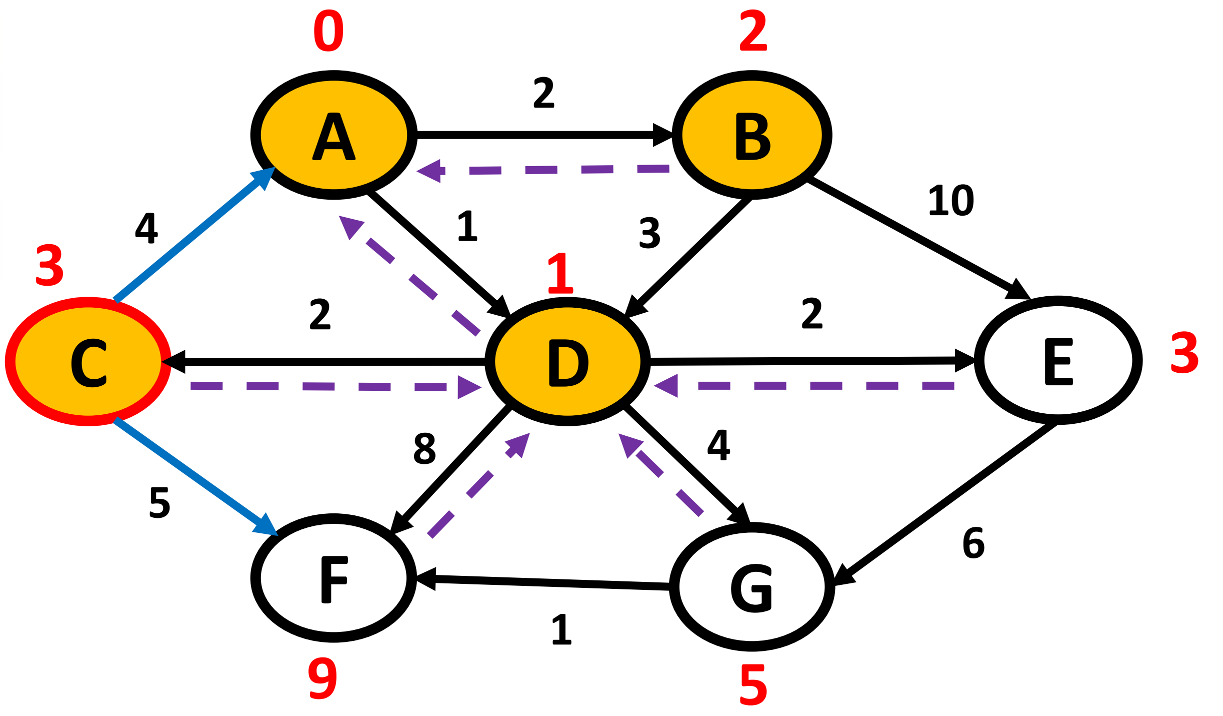
```
function dijkstra(v1, v2):
  for each vertex v:                  // Initialize vertex info
     v's cost := infinity.
     v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
    v := remove vertex from pqueue with minimum cost. // B
    mark v as visited.
    for each unvisited neighbor n of v:   // E
       cost := v's cost + weight of edge (v, n).  // 2 + 10

       if cost < n's cost:  // 12 > 3; False
          n's cost := cost.   // no cost change for E
          n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[C: 3, E: 3, F: 9, G: 5]**

Dijkstra's Shortest Path Algorithm

```
function dijkstra(v1, v2):
   for each vertex v:                // Initialize vertex info
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost. // C
      mark v as visited.
      for each unvisited neighbor n of v:   // F
         cost := v's cost + weight of edge (v, n).

         if cost < n's cost:
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[E: 3, F: 9, G: 5]**

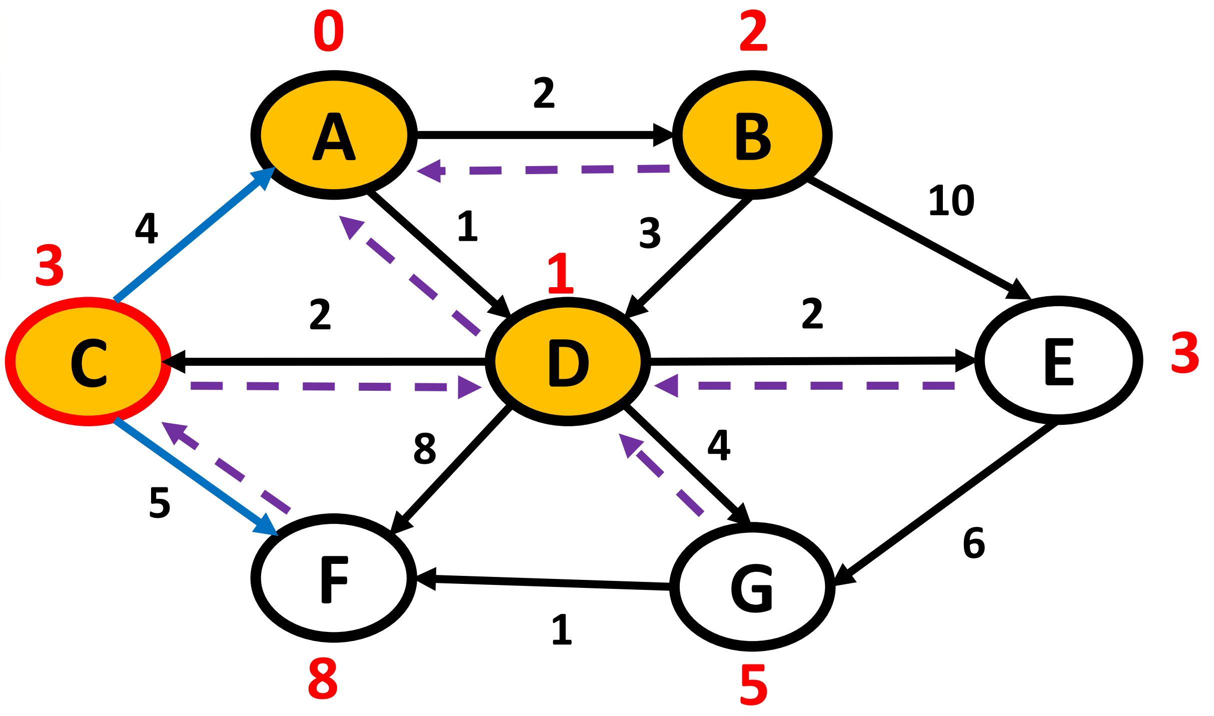
Dijkstra's Shortest Path Algorithm

function **dijkstra**(v_1, v_2):
  for each vertex *v*: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while *pqueue* is not empty:
    v := remove vertex from *pqueue* with minimum cost. **// C**
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*: **// F**
      cost := *v*'s cost + weight of edge (*v*, *n*). **// 3 + 5**

      if *cost* < *n*'s cost: **// 8 < 9; True**
        n's cost := *cost*. **// F's cost is updated to 8**
        n's previous := *v*.

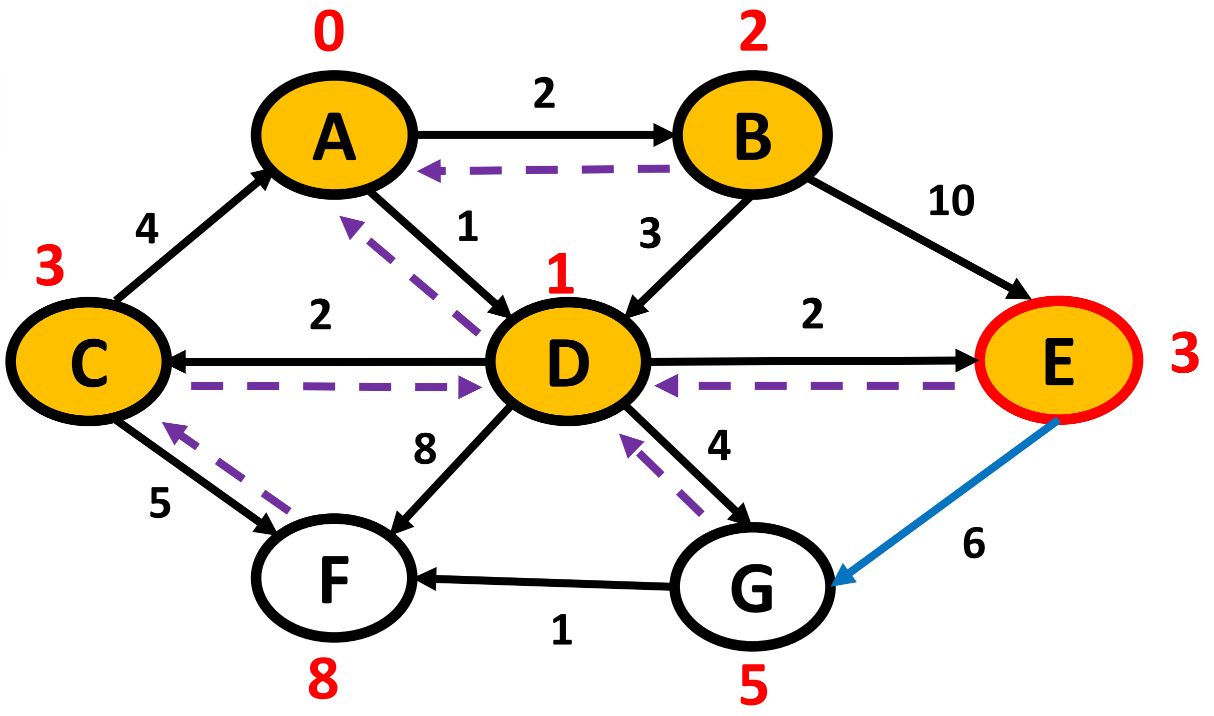
  reconstruct path from v_2 back to v_1, following previous pointers.



Priority Queue: **[E: 3, F: 8, G: 5]**

Dijkstra’s Shortest Path Algorithm

```
function dijkstra(v1, v2):
  for each vertex v:                  // Initialize vertex info
     v's cost := infinity.
     v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
     v := remove vertex from pqueue with minimum cost. // E
     mark v as visited.
     for each unvisited neighbor n of v:   // G
        cost := v's cost + weight of edge (v, n).

        if cost < n's cost:
           n's cost := cost.
           n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[F: 8, G: 5]**

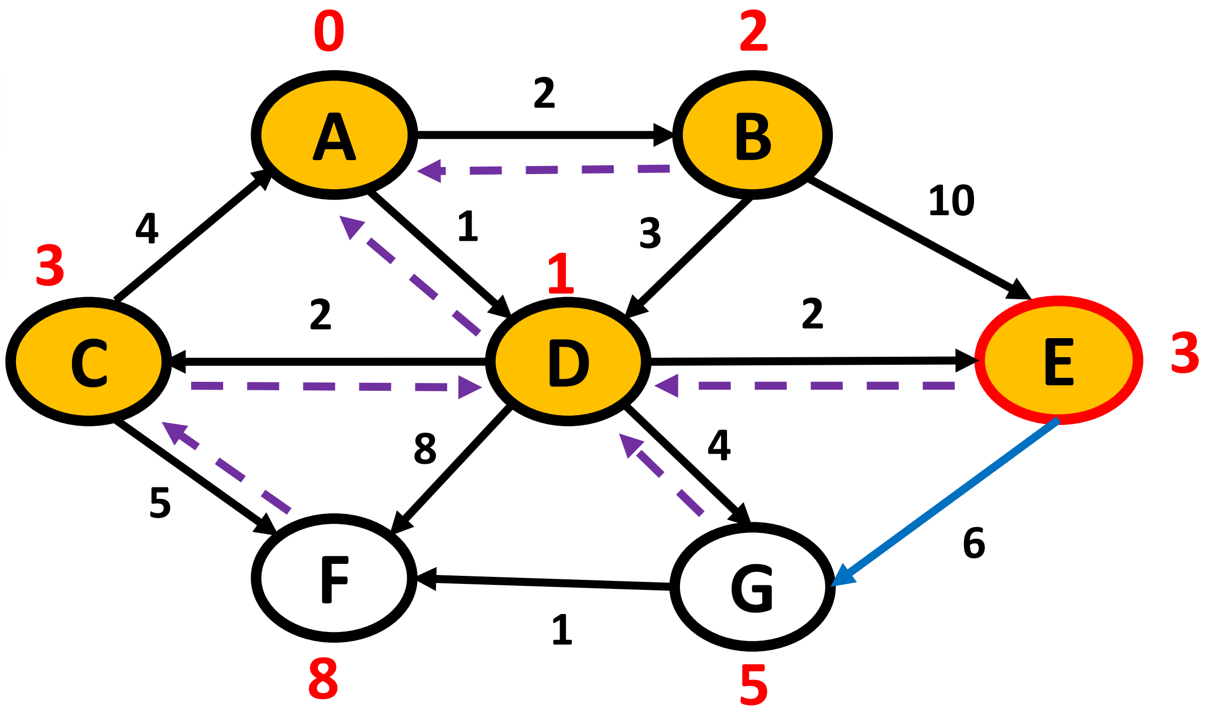
Dijkstra's Shortest Path Algorithm

function **dijkstra**(v_1, v_2):
  for each vertex *v*:   // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while *pqueue* is not empty:
    v := remove vertex from *pqueue* with minimum cost. // E
    mark *v* as visited.
    for each unvisited neighbor *n* of *v*: // G
      cost := *v*'s cost + weight of edge (*v*, *n*). // 3 + 6

      if *cost* < *n*'s cost: // 9 > 5; False
        n's cost := *cost*. // no cost change for G
        n's previous := *v*.

  reconstruct path from v_2 back to v_1, following previous pointers.



Priority Queue: **[F: 8, G: 5]**

Dijkstra's Shortest Path Algorithm

function **dijkstra**(v_1, v_2):

  for each vertex *v*:   // Initialize vertex info

    v's cost := infinity.

    v's previous := none.

  v_1's cost := 0.

  pqueue := {all vertices, ordered by distance}.

  while *pqueue* is not empty:

    v := remove vertex from *pqueue* with minimum cost. // G

    mark *v* as visited.

    for each unvisited neighbor *n* of *v*:   // F

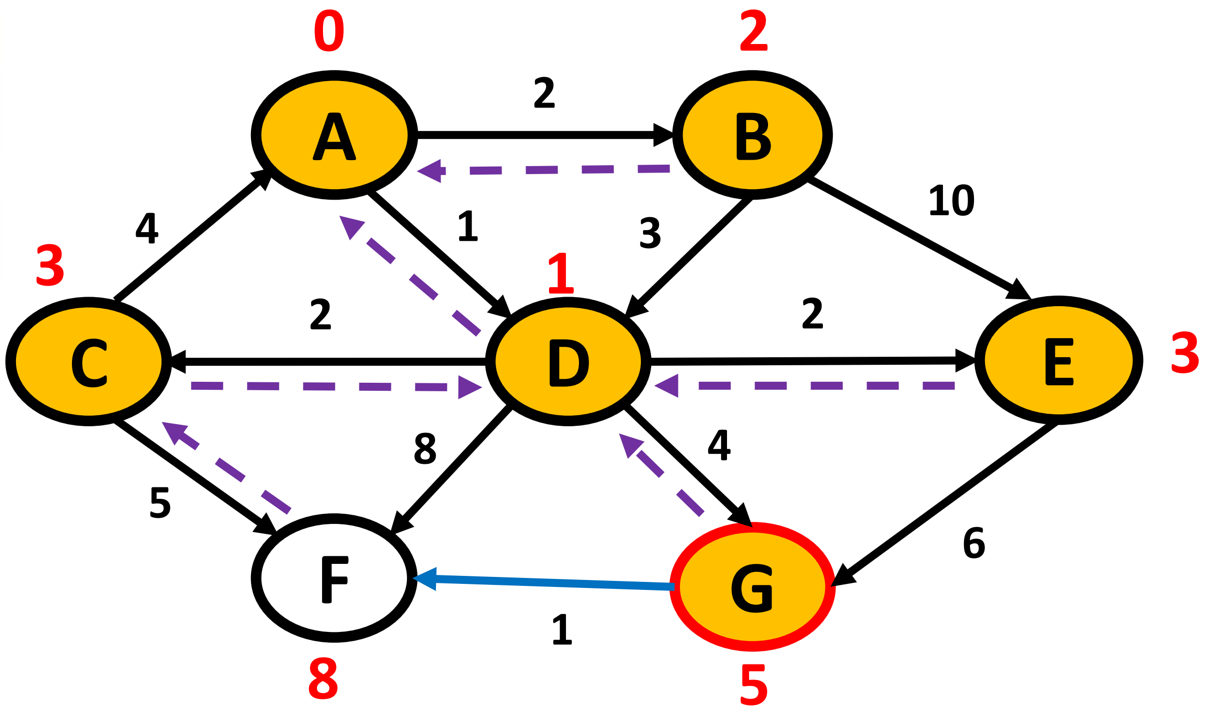
      cost := *v*'s cost + weight of edge (*v*, *n*).

      if *cost* < *n*'s cost:

        n's cost := *cost*.

        n's previous := *v*.

  reconstruct path from v_2 back to v_1, following previous pointers.



Priority Queue: **[F: 8]**

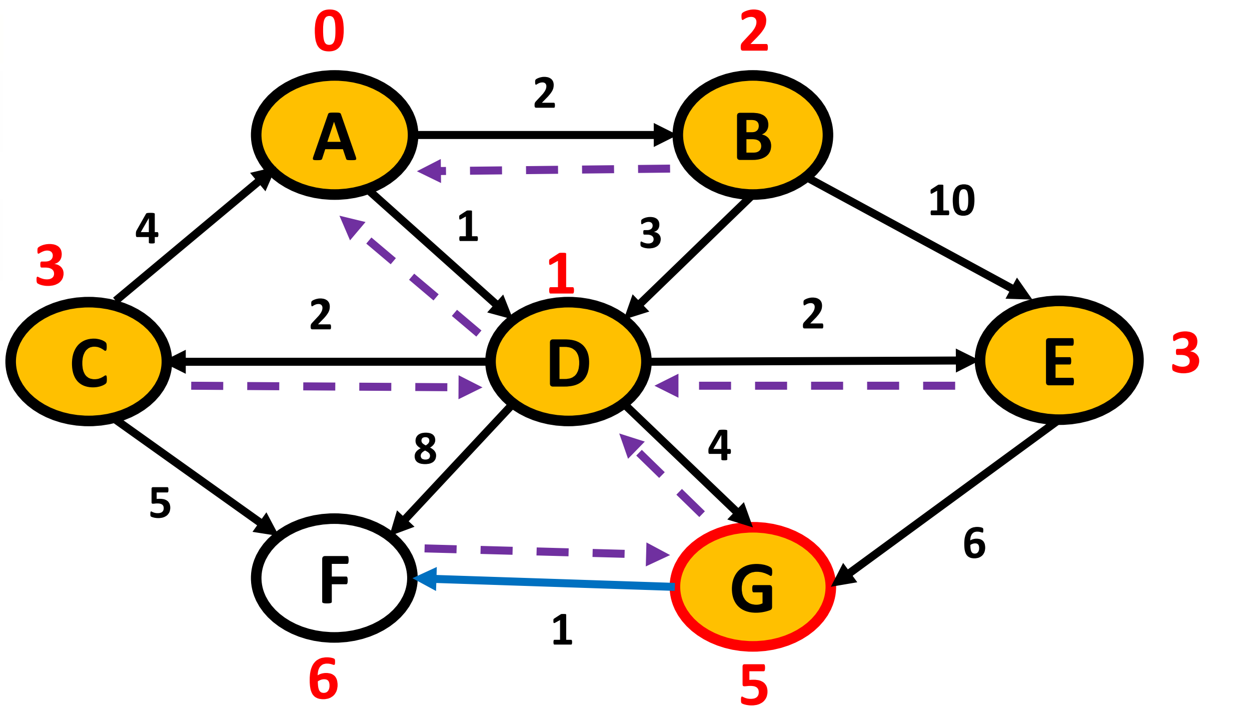
Dijkstra's Shortest Path Algorithm

function **dijkstra**(v_1, v_2):
  for each vertex v: // Initialize vertex info
    v's cost := infinity.
    v's previous := none.
  v_1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while *pqueue* is not empty:
    v := remove vertex from *pqueue* with minimum cost. // G
    mark v as visited.
    for each unvisited neighbor n of v: // F
      cost := v's cost + weight of edge (v, n). // 5 + 1

      if *cost* < n's cost: // 6 < 8; True
        n's cost := *cost*. // F's cost is updated to 6
        n's previous := v.

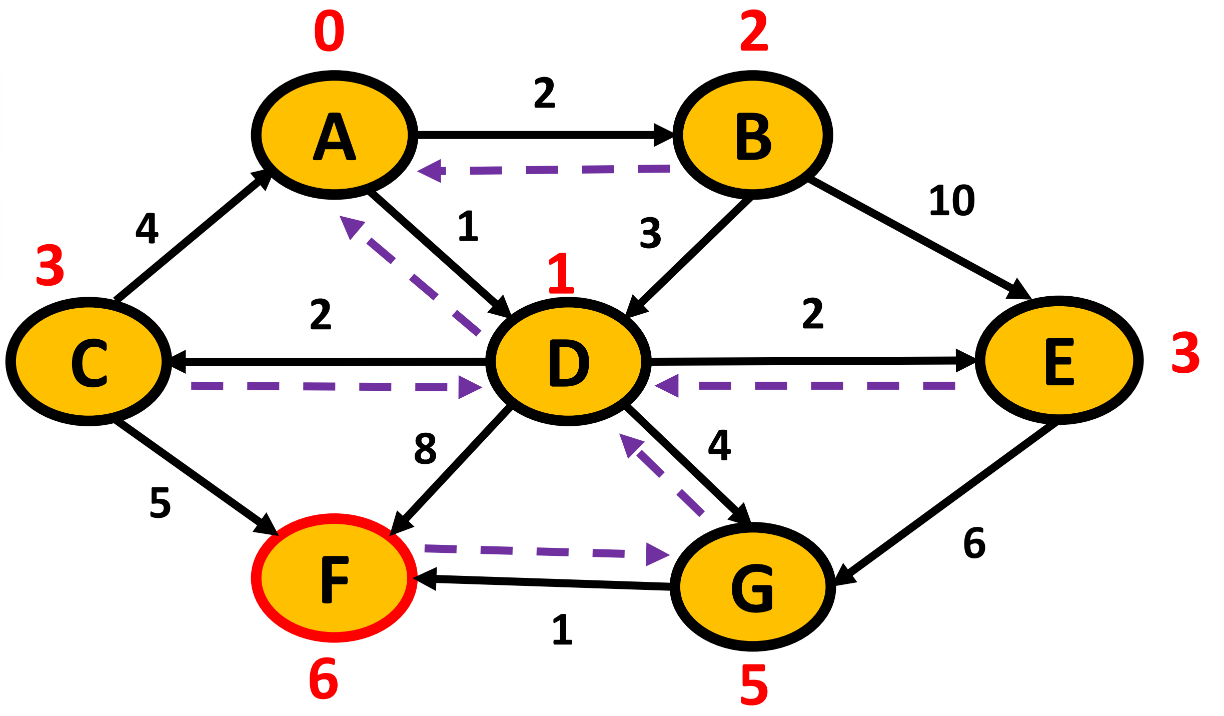
  reconstruct path from v_2 back to v_1, following previous pointers.



Priority Queue: **[F: 6]**

Dijkstra’s Shortest Path Algorithm

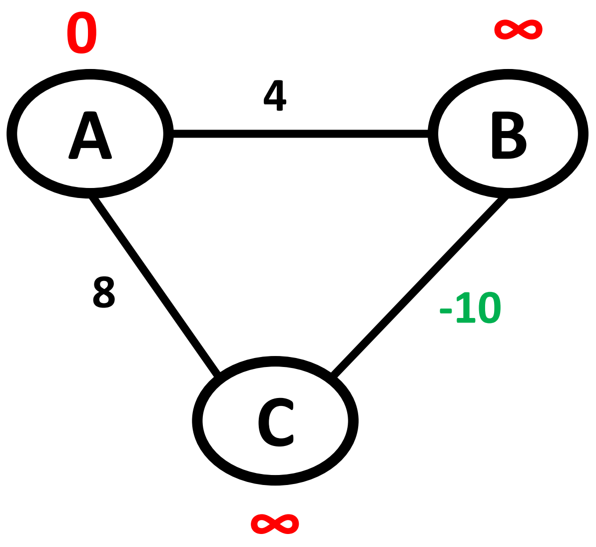
```
function dijkstra(v1, v2):
  for each vertex v:                    // Initialize vertex info
     v's cost := infinity.
     v's previous := none.
  v1's cost := 0.
  pqueue := {all vertices, ordered by distance}.

  while pqueue is not empty:
    v := remove vertex from pqueue with minimum cost. // F
    mark v as visited.
    for each unvisited neighbor n of v:   // none
      cost := v's cost + weight of edge (v, n).

      if cost < n's cost:  // no costs change
        n's cost := cost.
        n's previous := v.

  reconstruct path from v2 back to v1, following previous pointers.
```



Priority Queue: **[ ]**

Dijkstra’s Shortest Path Algorithm

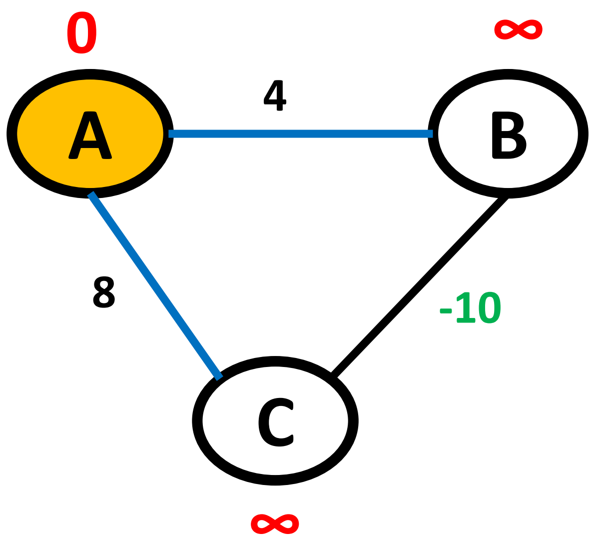
- Does Dijkstra’s Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[A: 0, B: ∞, C: ∞]**

Dijkstra’s Shortest Path Algorithm

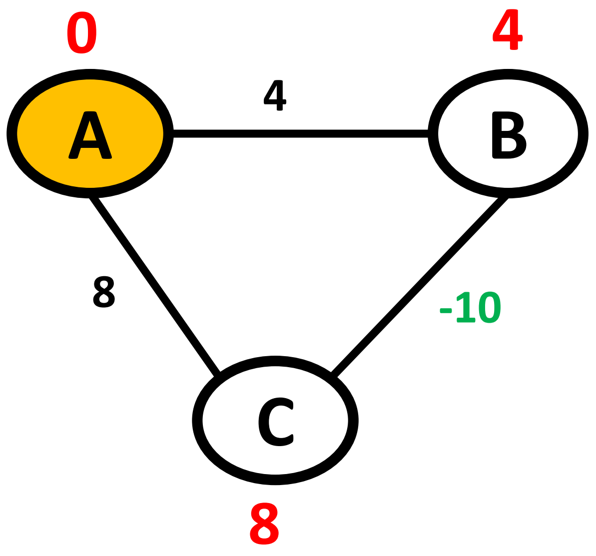
- Does Dijkstra’s Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[B: ∞, C: ∞]**

Dijkstra's Shortest Path Algorithm

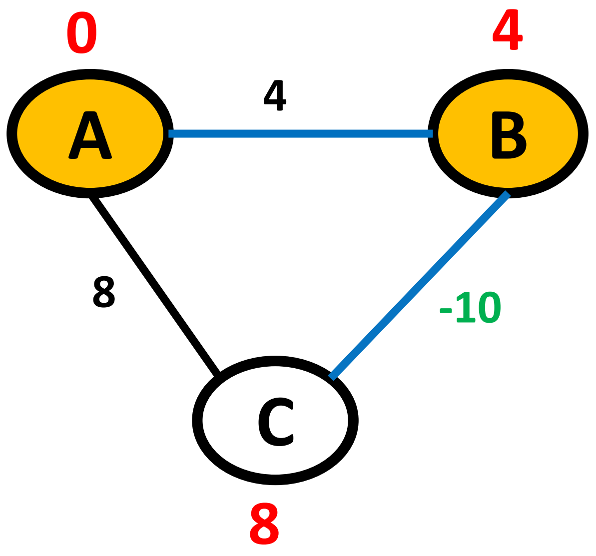
- Does Dijkstra's Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[B: 4, C: 8]**

Dijkstra’s Shortest Path Algorithm

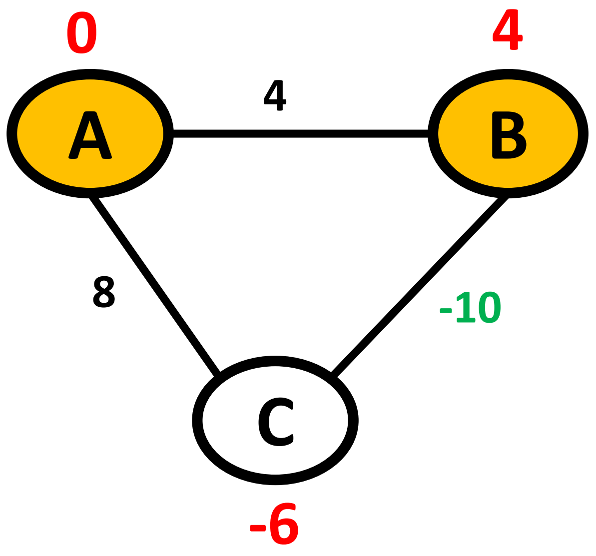
- Does Dijkstra’s Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[C: 8]**

Dijkstra’s Shortest Path Algorithm

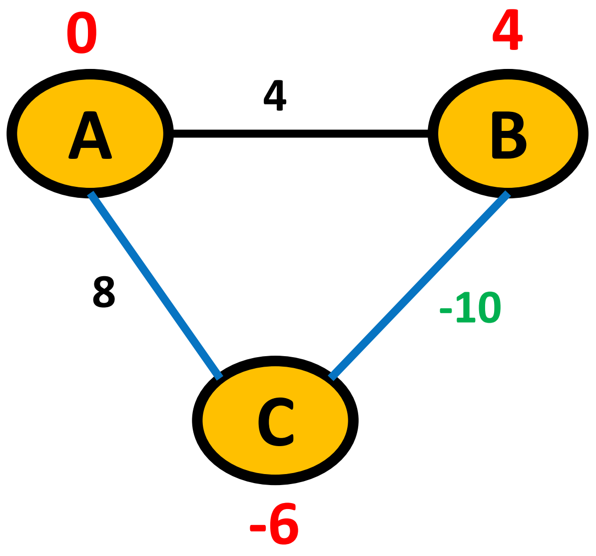
- Does Dijkstra’s Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[C: -6]**

Dijkstra's Shortest Path Algorithm

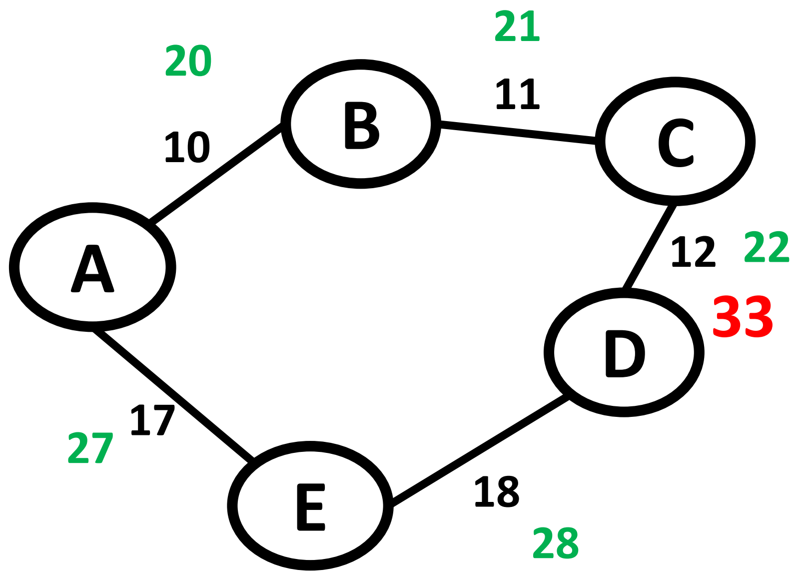
- Does Dijkstra's Shortest Path Algorithm works for **negative edges**?



Priority Queue: **[ ]**

Dijkstra’s Shortest Path Algorithm

- Does the shortest path changes if we **add a weight to all edges** of the original Graph?



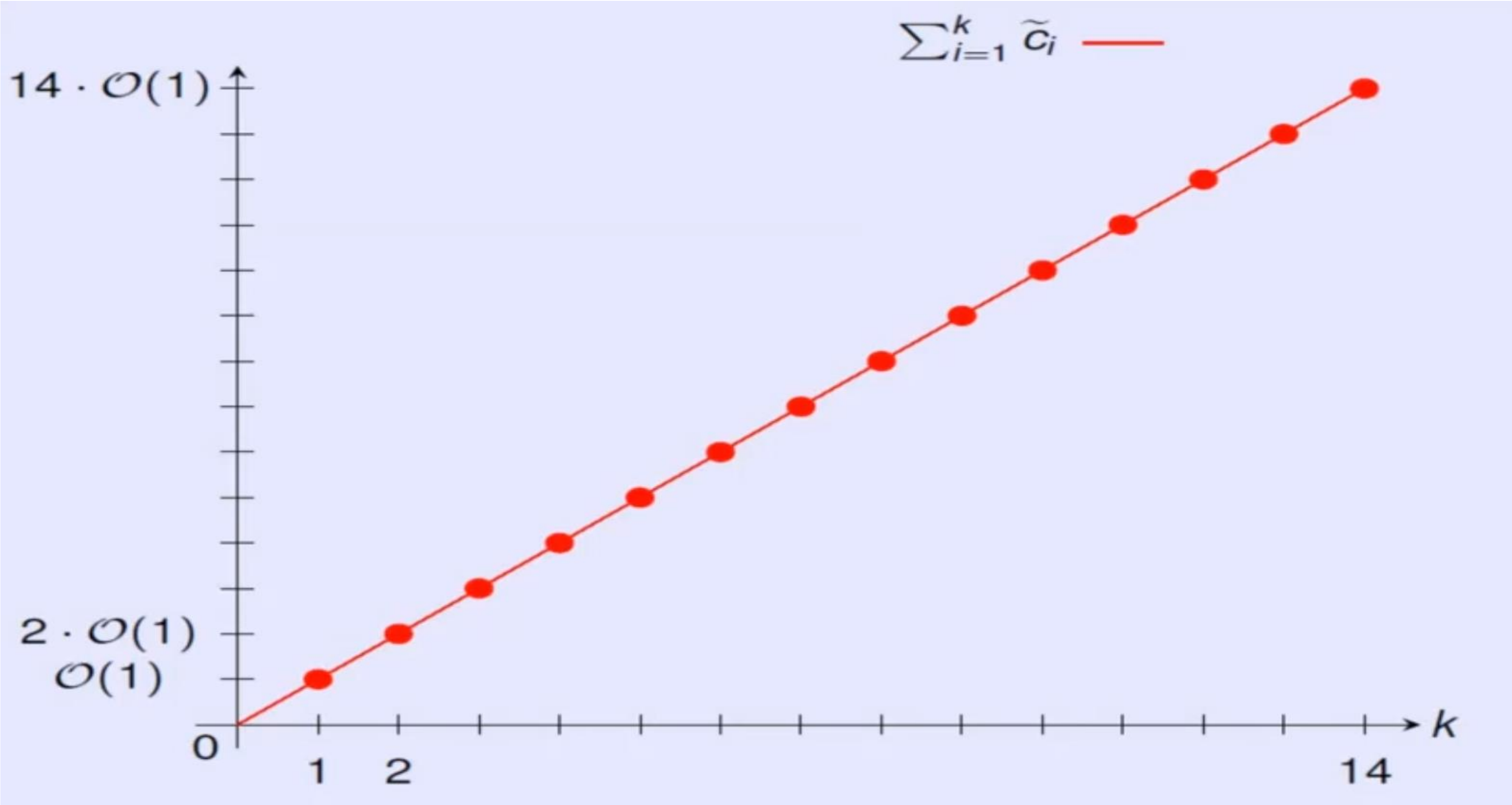
Dijkstra’s Shortest Path Algorithm

Total Complexity = E log V + V log V

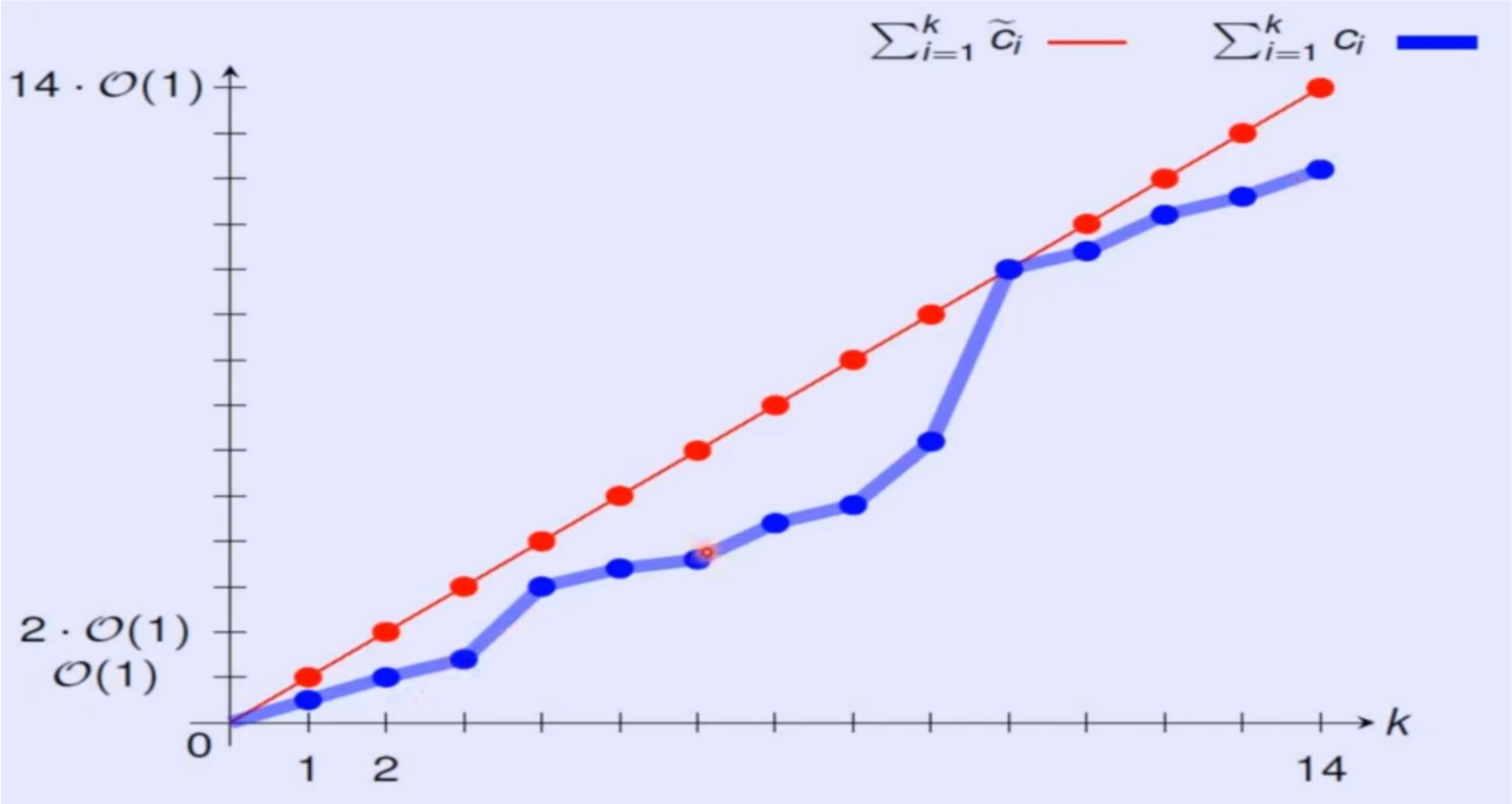
```
function dijkstra(v1, v2):
   for each vertex v:
      v's cost := infinity.
      v's previous := none.
   v1's cost := 0.
   pqueue := {all vertices, ordered by distance}.          } V log V

   while pqueue is not empty:
      v := remove vertex from pqueue with minimum cost.    } O (log V)   } V
      mark v as visited.
      for each unvisited neighbor n of v:
         cost := v's cost + weight of edge (v, n).

         if cost < n's cost:                                  O (log V)     E
            n's cost := cost.
            n's previous := v.

   reconstruct path from v2 back to v1, following previous pointers.
```

Relax Operation

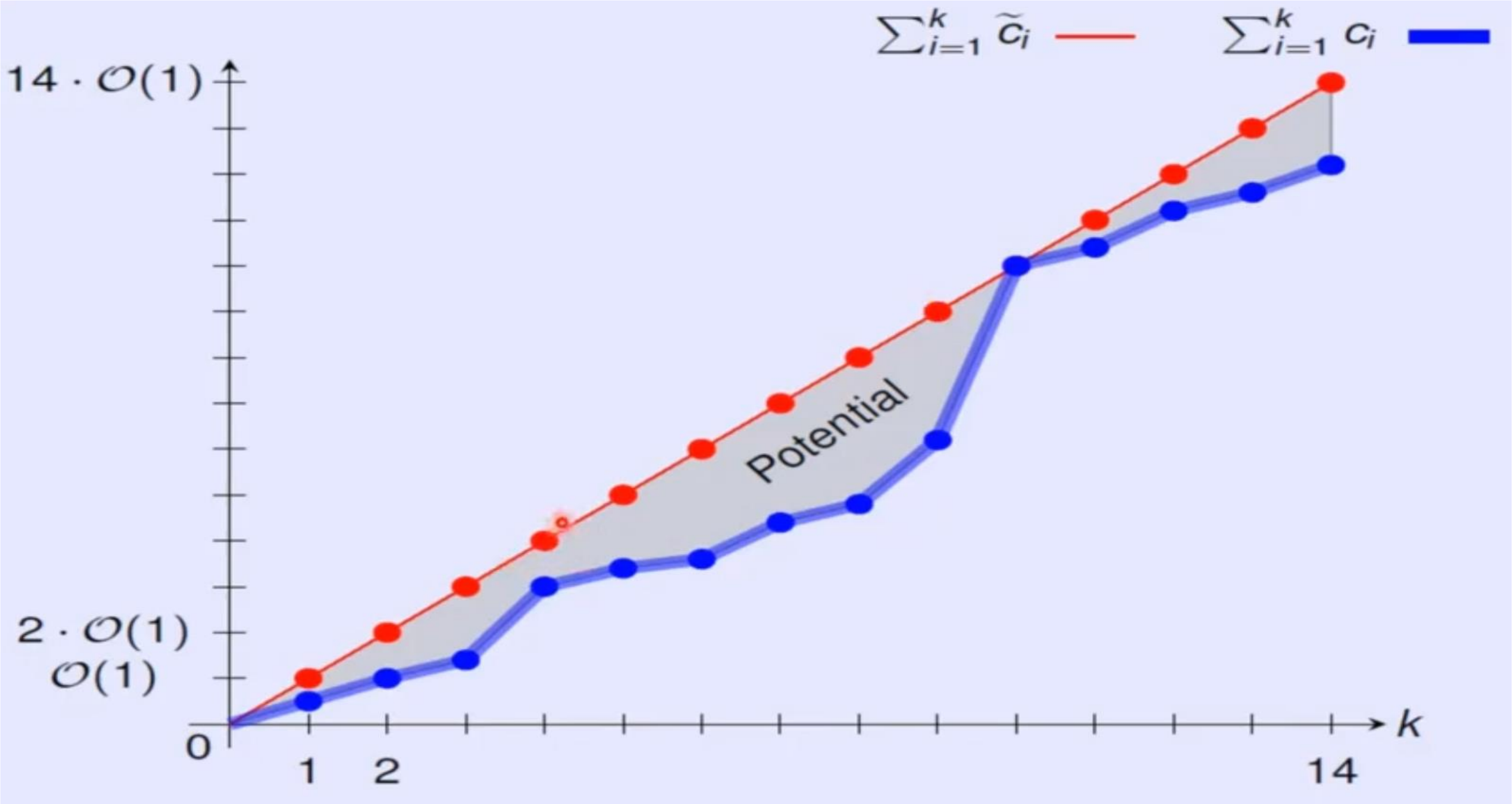
Actual Vs Amortised Cost



Actual Vs Amortised Cost



Actual Vs Amortised Cost



Actual Vs Amortised Cost

PQ implementation	Insert	Extract - Min	Decrease Key	Total
Unordered array	1	V	1	V^2
Binary Heap	log V	log V	log V	E log V
Fibonacci Heap	1*	log V *	1*	E + VlogV