Data Structure and Algorithm
CSE-225

Graph

- Terminology
- Sequential Representation of Graphs: Adjacency Matrix, Path Matrix
- Warshall's Algorithm (Path Matrix)
- Shortest Path
- Graph Traversal
  - Breath First Search (BFS)
  - Depth First Search (DFS)
- Spanning Tree
  - Minimum Spanning Tree
  - Kruskal Algorithm
  - Prime's Algorithm

Graph

- A graph is a collection of nodes (or vertices) and edges (or arcs)
  - Each node contains an element
  - Each edge connects two nodes together (or possibly the same node to itself) and may contain an edge attribute

- A directed graph is one in which the edges have a direction
- An undirected graph is one in which the
  - edges do not have a direction

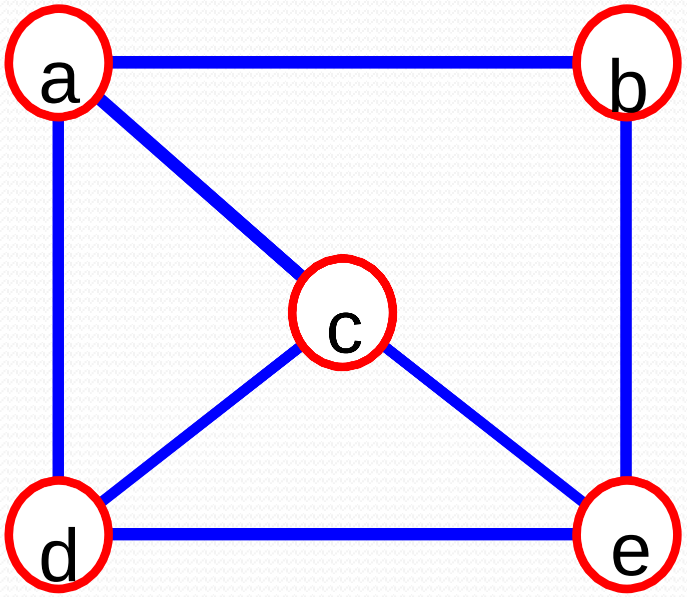
What is a Graph?

A graph G = (V,E) is composed of:

V: set of vertices

E: set of edges connecting the vertices in V

- An edge e = (u,v) is a pair of vertices
- Example:



V= {a,b,c,d,e}

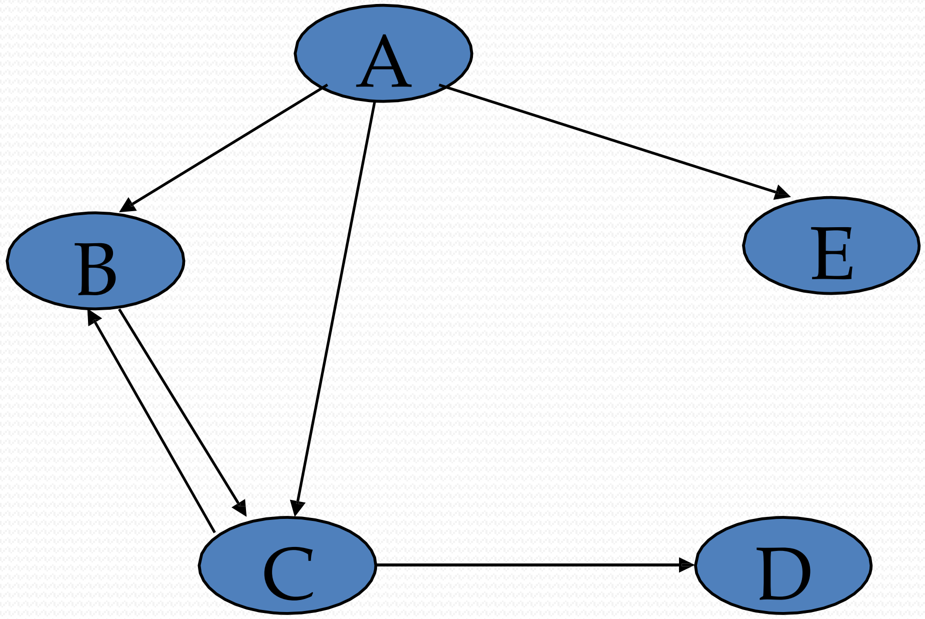
E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e), (d,e)}

Some problems that can be represented by a graph

- computer networks
- airline flights
- road map
- course prerequisite structure
- tasks for completing a job
- flow of control through a program

  and many more

A Directed Graph or digraph



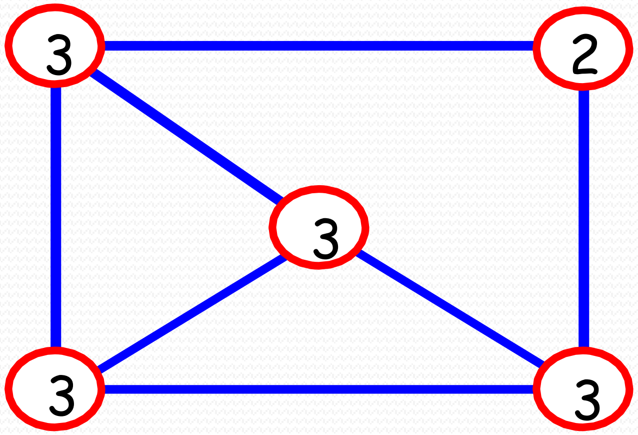
V = [A, B, C, D, E]
E = [<A,B>, <B,C>, <C,B>, <A,C>, <A,E>, <C,D>]

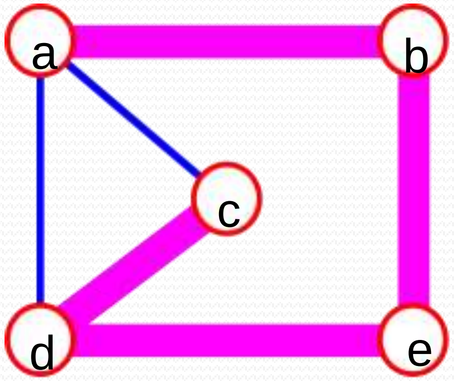
Graph terminology

- The size of a graph is the number of nodes in it
- The empty graph has size zero (no nodes)
- If two nodes are connected by an edge, they are neighbors (and the nodes are adjacent to each other)
- The degree of a node is the number of edges it has
- For directed graphs,
  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D
    - S is a predecessor of D, and D is a successor of S
  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

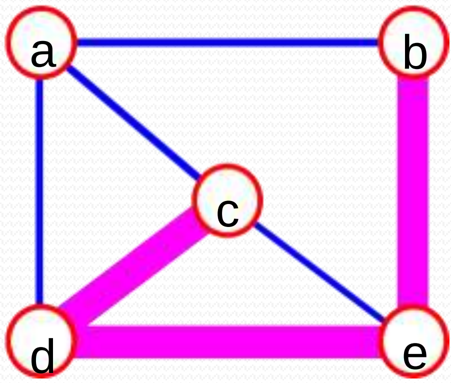
Terminology:

- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.





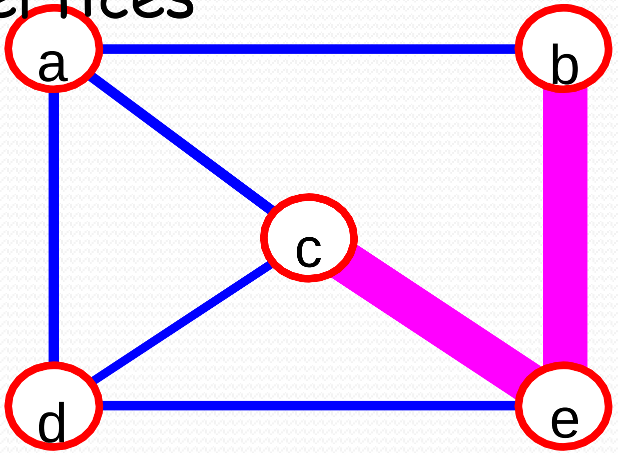
a b e d c



b e d c

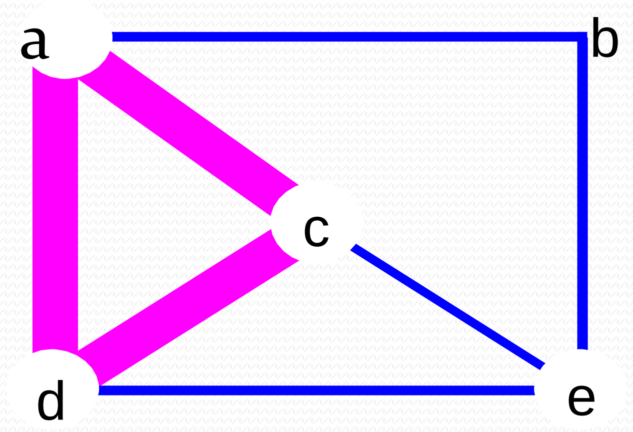
More Terminology

- simple path: no repeated vertices

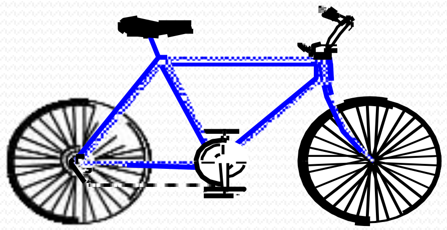


b e c

- cycle: simple path, except that the last vertex is the same as the first vertex

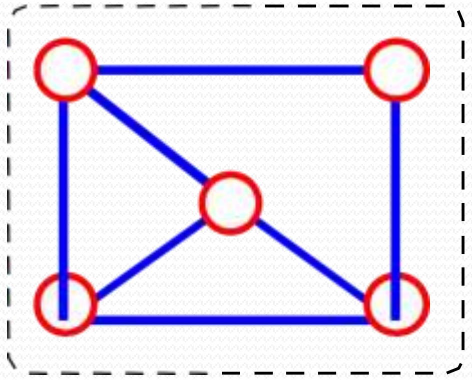


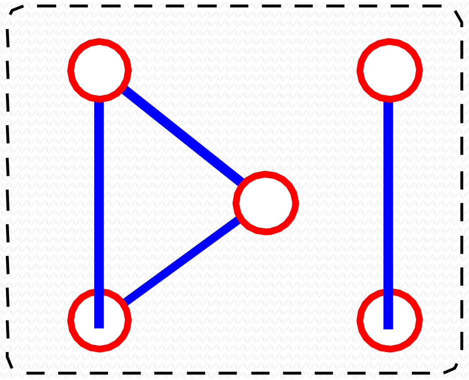
a c d a



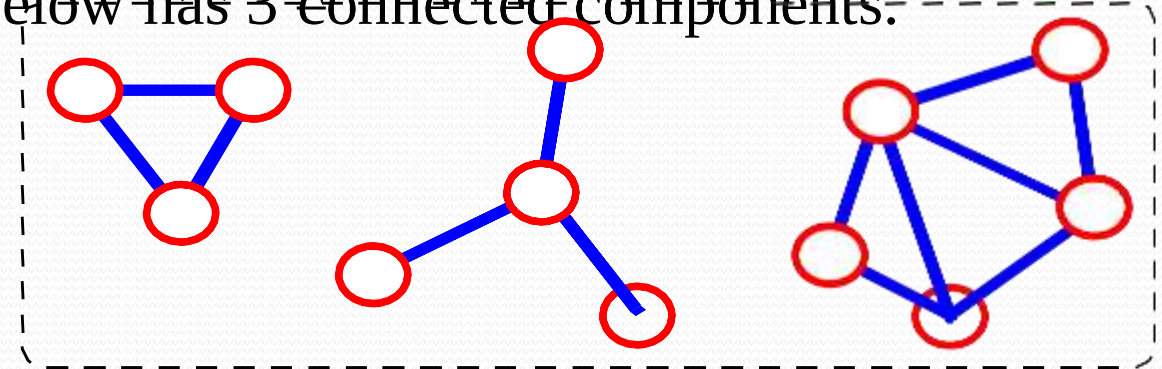
Even More Terminology

- connected graph: any two vertices are connected by some path

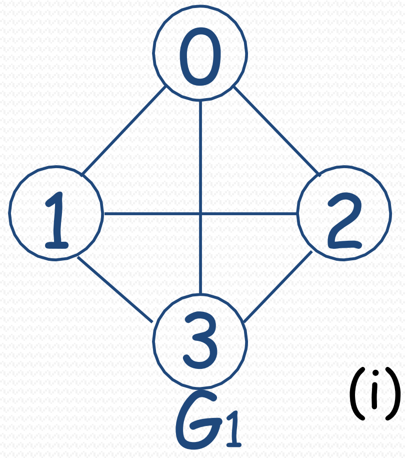

connected

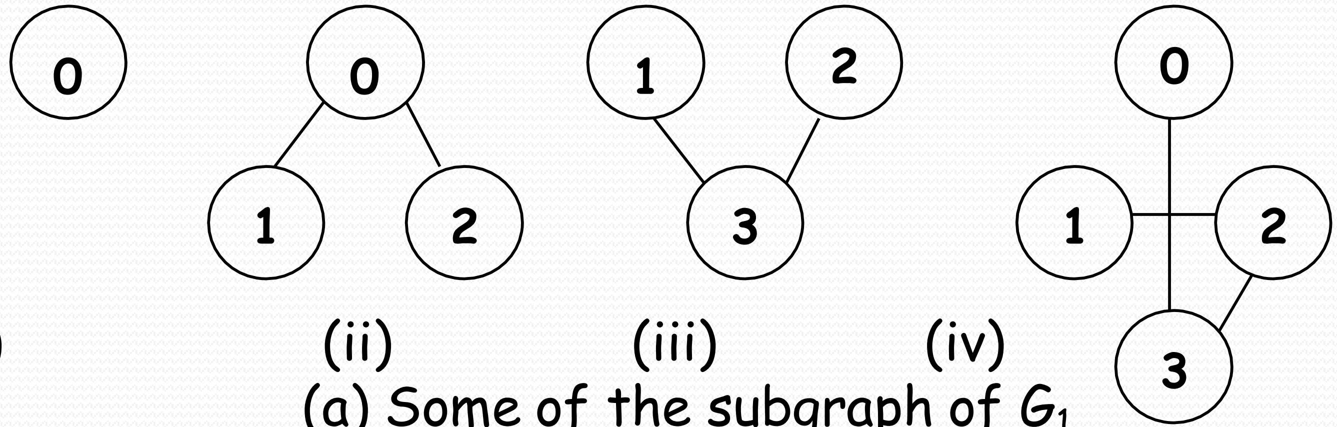

not connected

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.

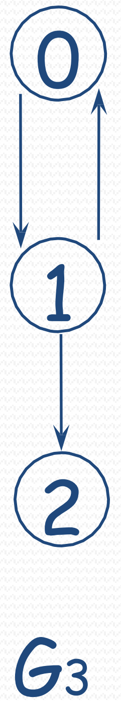


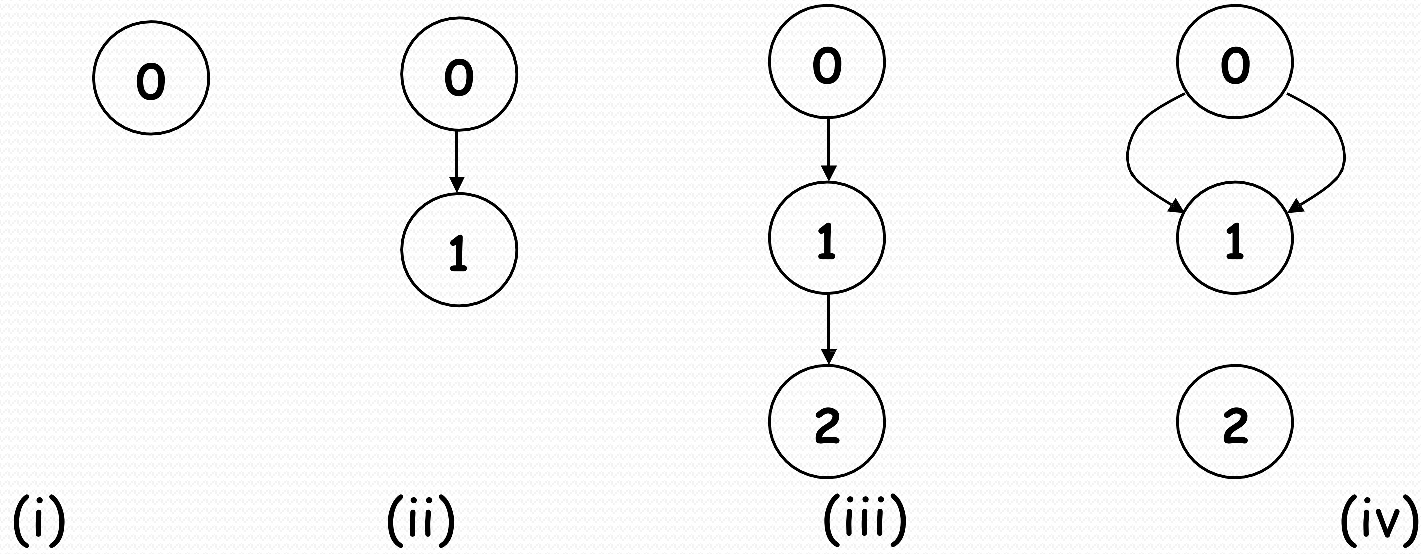
Subgraphs Examples





(a) Some of the subgraph of G_1

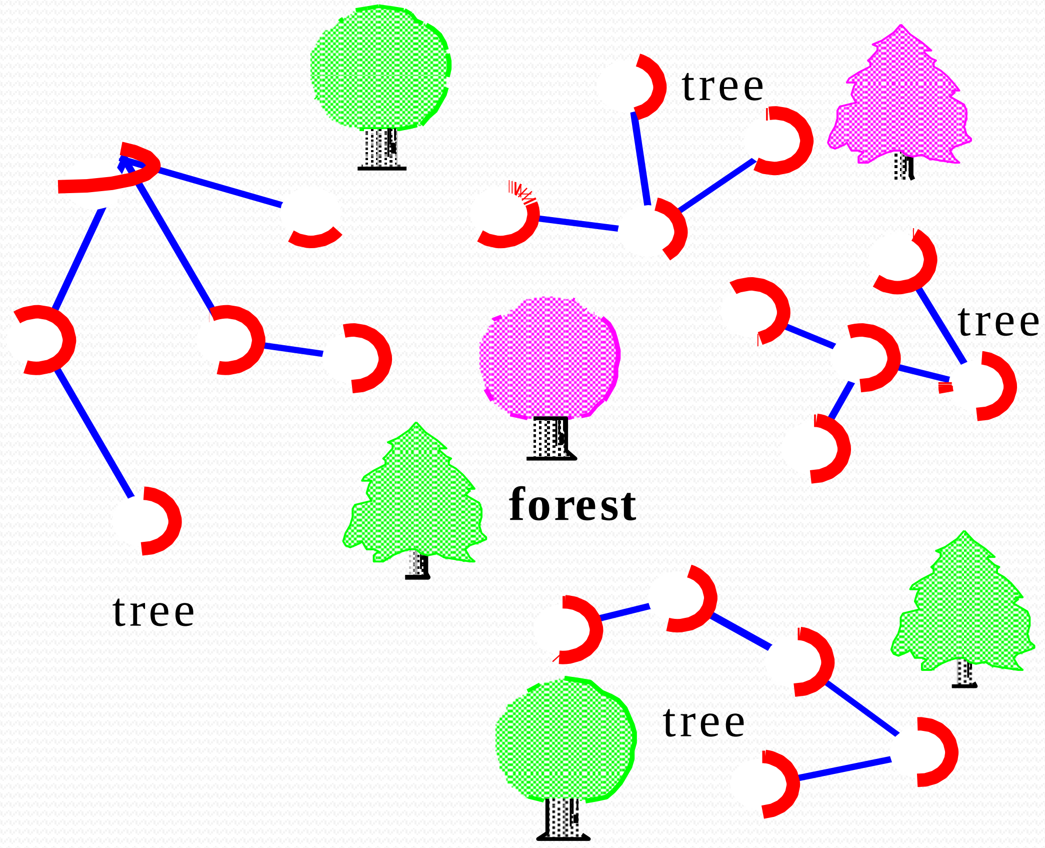




(b) Some of the subgraph of G_3

More…

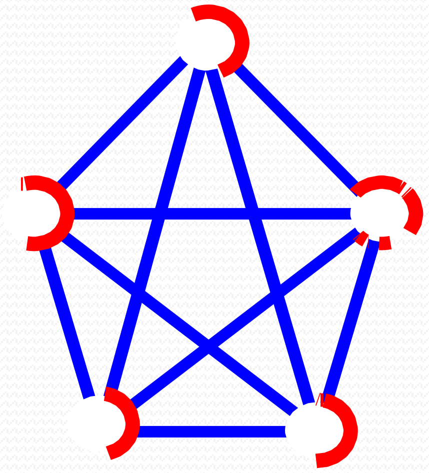
- tree - connected graph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- How many total edges in a complete graph?
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice!

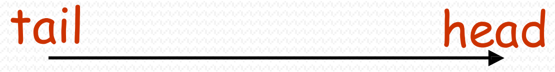
  Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



n = 5
m = (5 * 4)/2 = 10

Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered,
  $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices,
  $<v_0, v_1> \; != \; <v_1, v_0>$



Graph terminology

- An undirected graph is connected if there is a path from every node to every other node
- A directed graph is strongly connected if there is a path from every node to every other node
- Node X is reachable from node Y if there is a path from Y to X

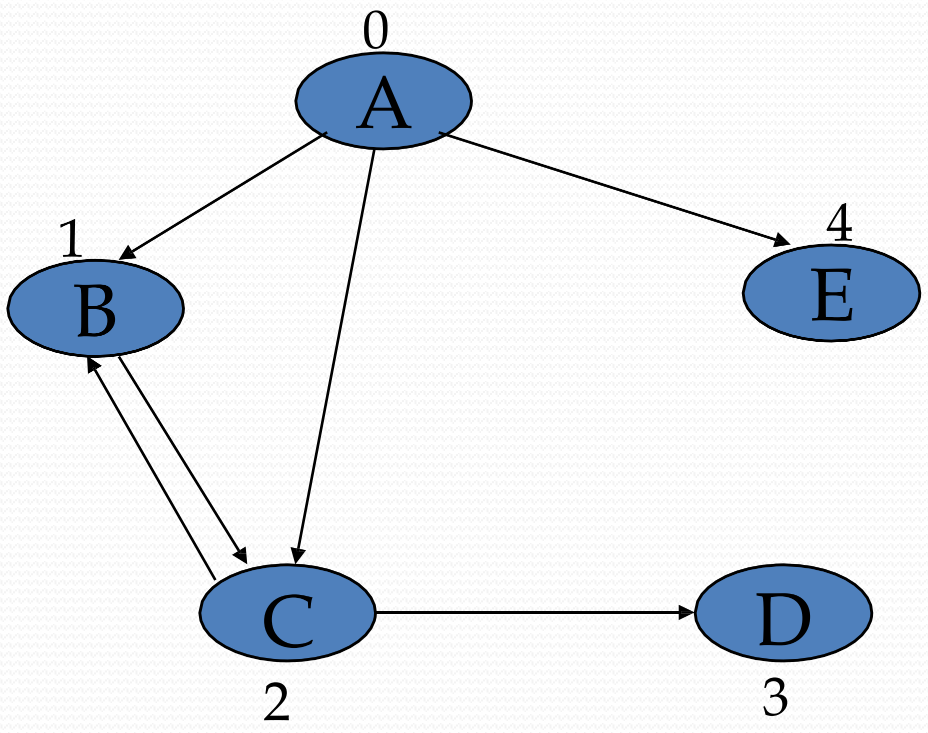
Graph data structures

- storing the vertices
  - each vertex has a unique identifier and, maybe, other information
  - for efficiency, associate each vertex with a number that can be used as an index
- storing the edges
  - adjacency matrix – represent all possible edges
  - adjacency lists – represent only the existing edges

storing the vertices

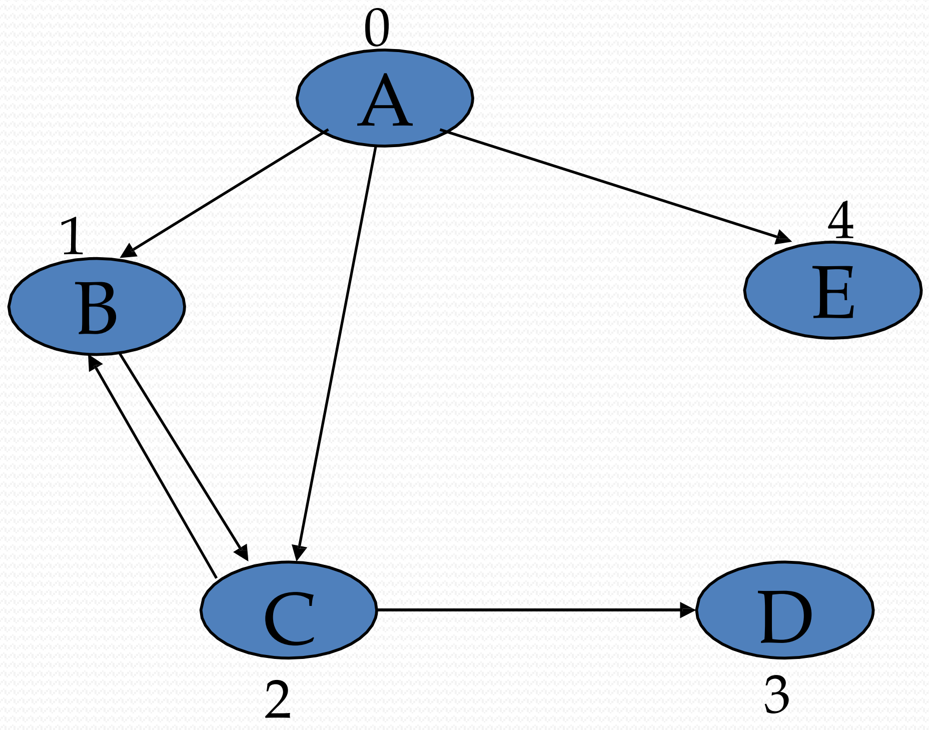
- when a vertex is added to the graph, assign it a number
  - vertices are numbered between 0 and n-1
- graph operations start by looking up the number associated with a vertex
- many data structures to use
  - any of the associative data structures
  - for small graphs a vector can be used
    - search will be O(n)

the vertex vector



0	A
1	B
2	C
3	D
4	E

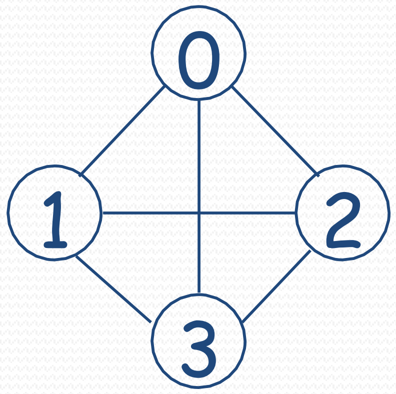
adjacency matrix



0	A
1	B
2	C
3	D
4	E

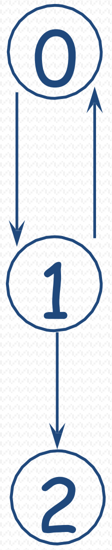
$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Examples for Adjacency Matrix



G_1

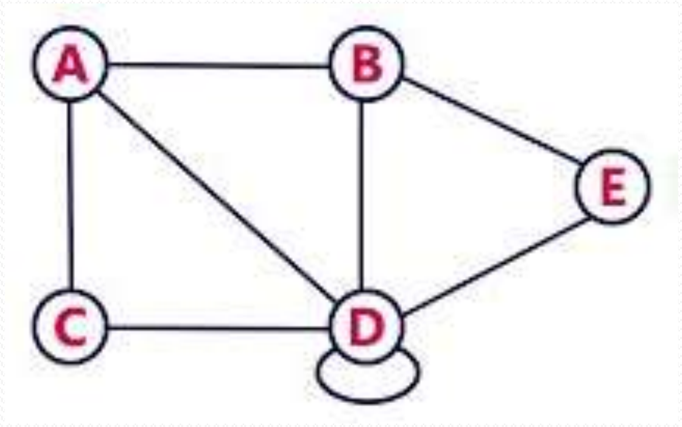
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$



G_2

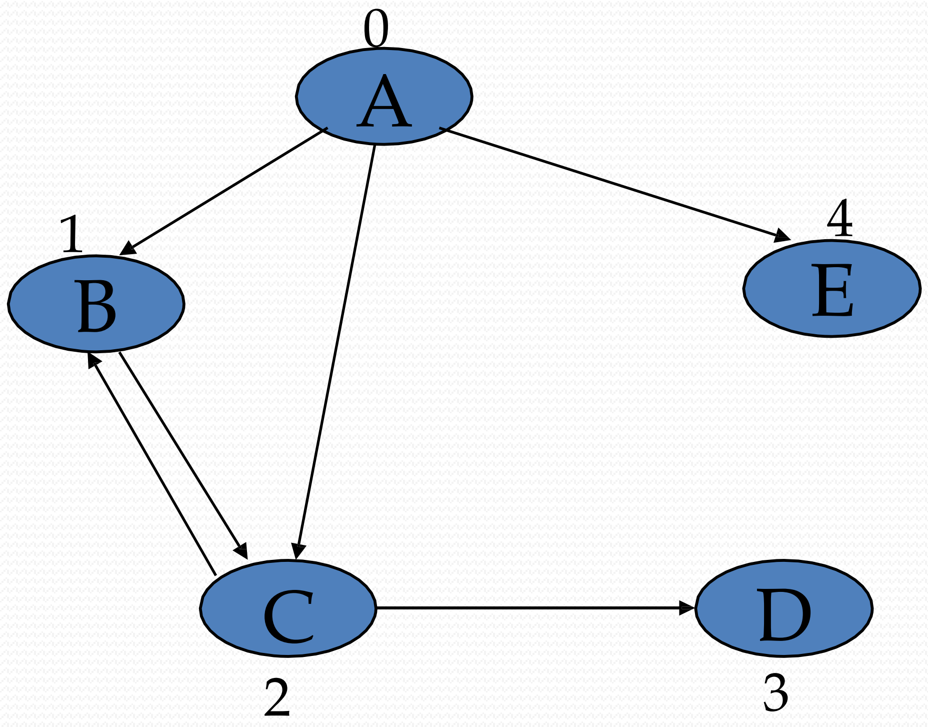
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

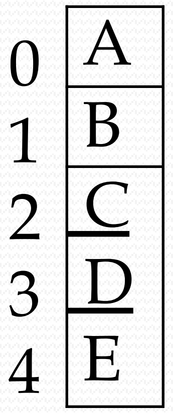
Examples for Adjacency Matrix

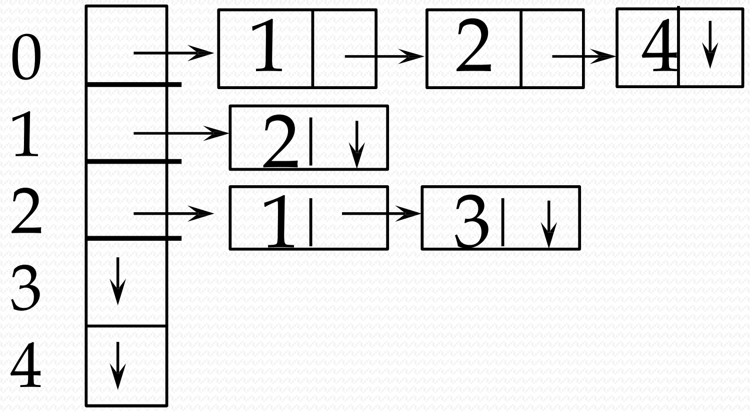


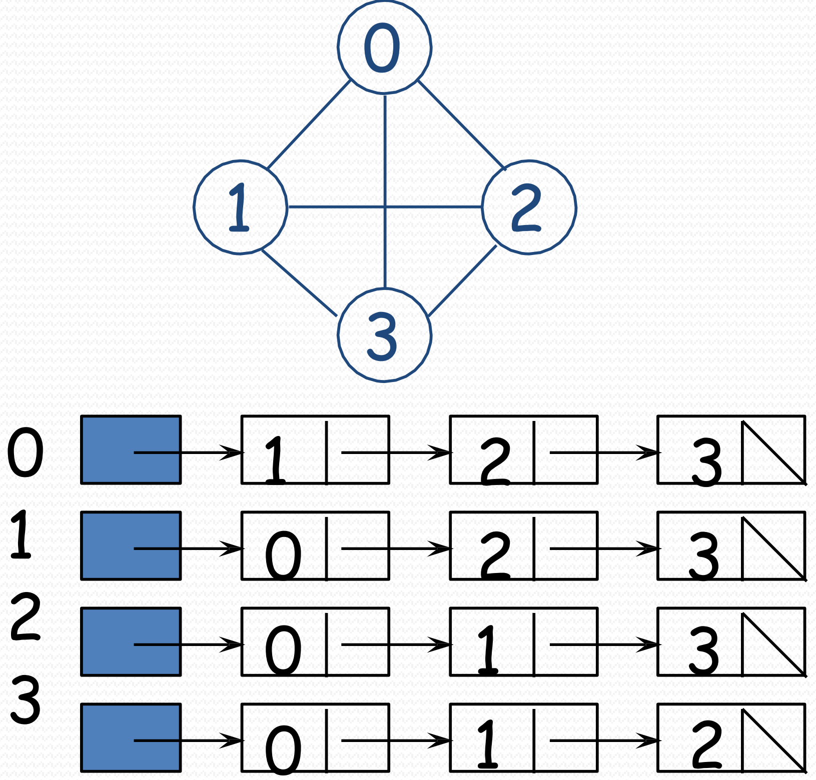
	A	B	C	D	E
A					
B					
C					
D					
E					

adjacency lists

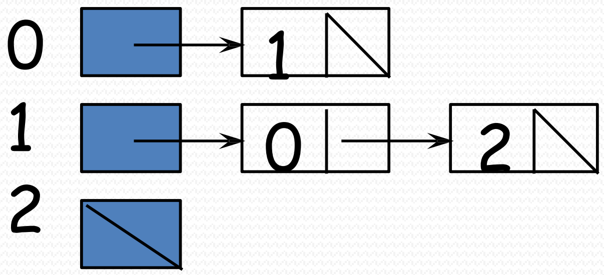


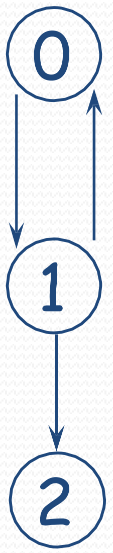






G_1

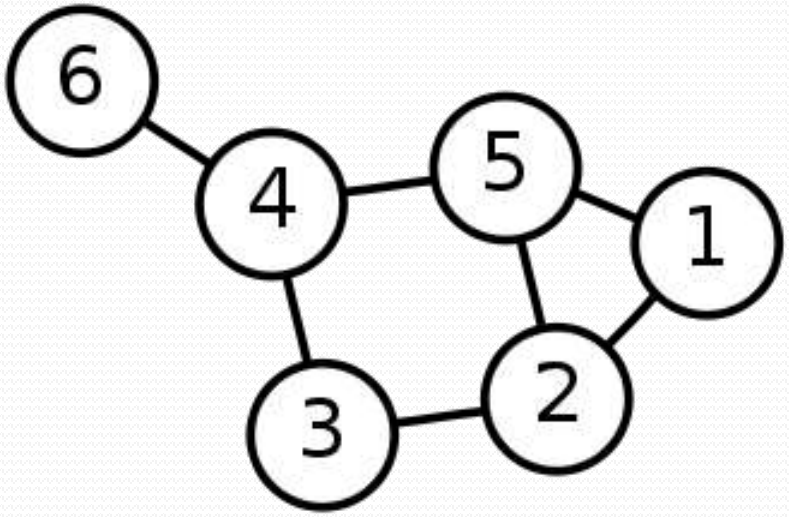




G_3

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex *v*∈V, such that all edge weights are nonnegative

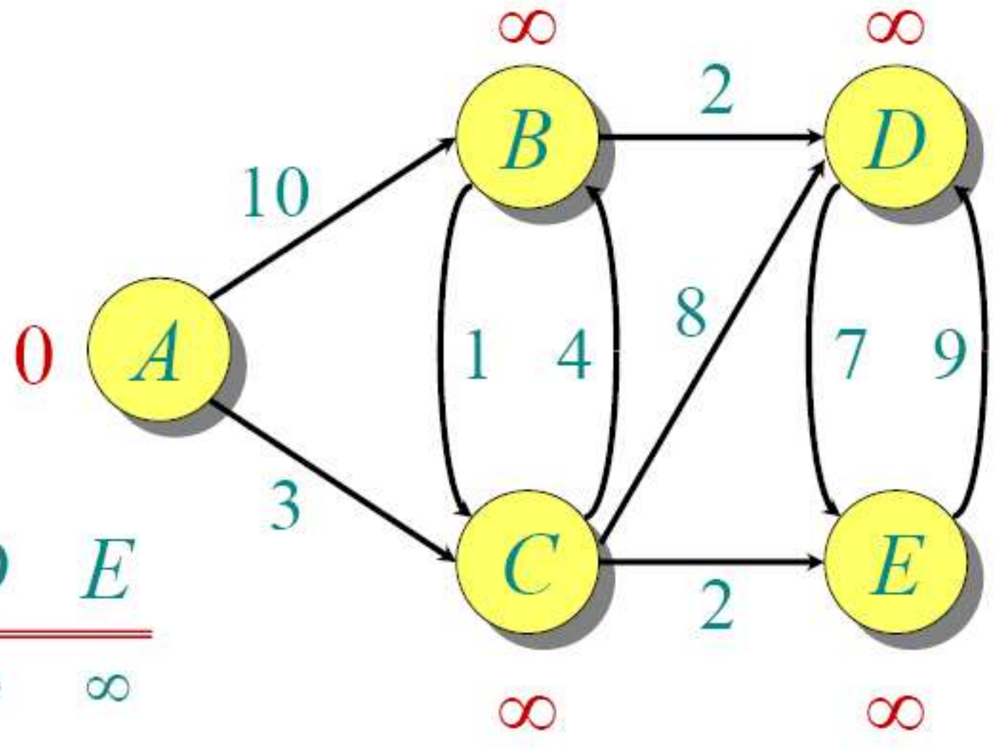
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex *v*∈V to all other vertices

Dijkstra's algorithm - Pseudocode

```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                      (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)    (select the element of Q with the min. distance)
      S←S∪{u}                               (add u to list of visited vertices)
       for all v ∈ neighbors[u]
             do  if   dist[v] > dist[u] + w(u, v)              (if new shortest path found)
                      then      d[v] ←d[u] + w(u, v)      (set new value of shortest path)
                            (if desired, add traceback code)
return dist
```

Dijkstra Animated Example

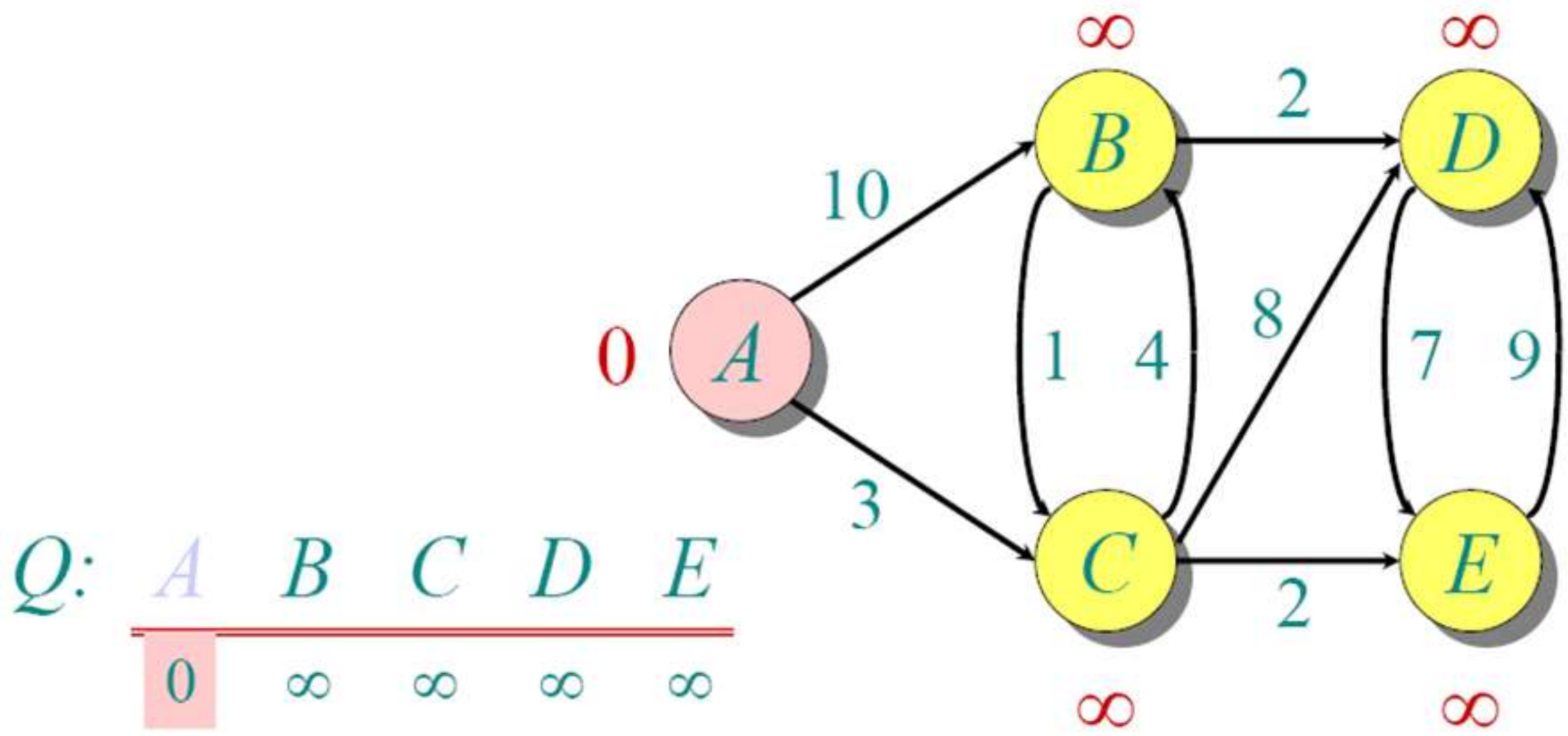
Initialize:



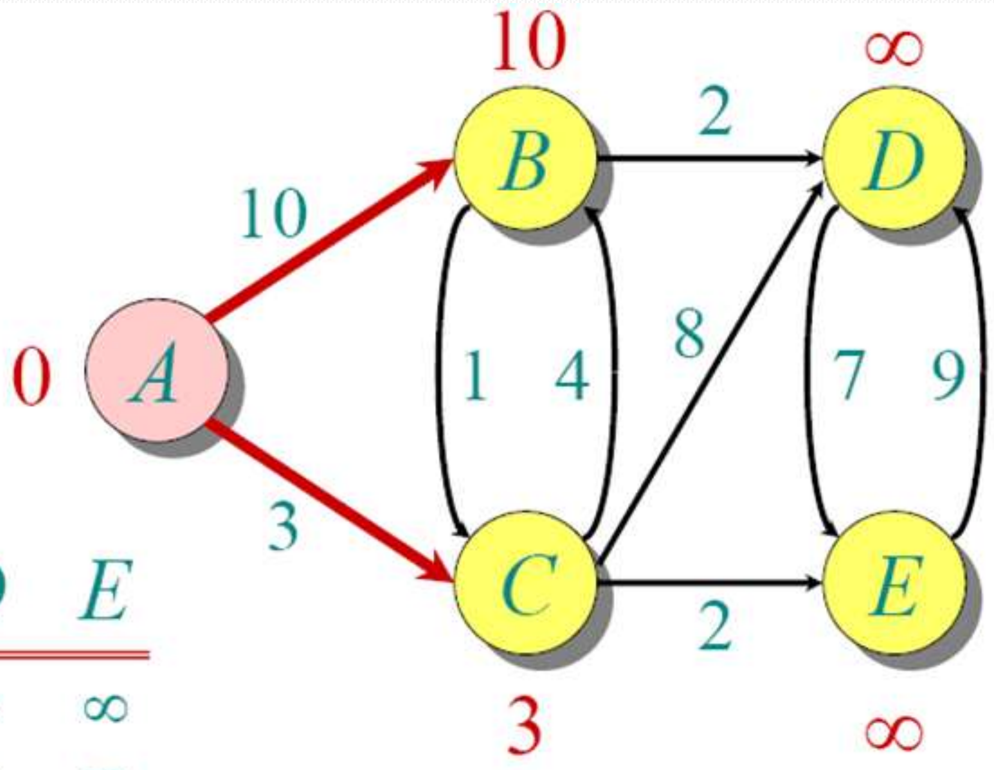
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Dijkstra Animated Example



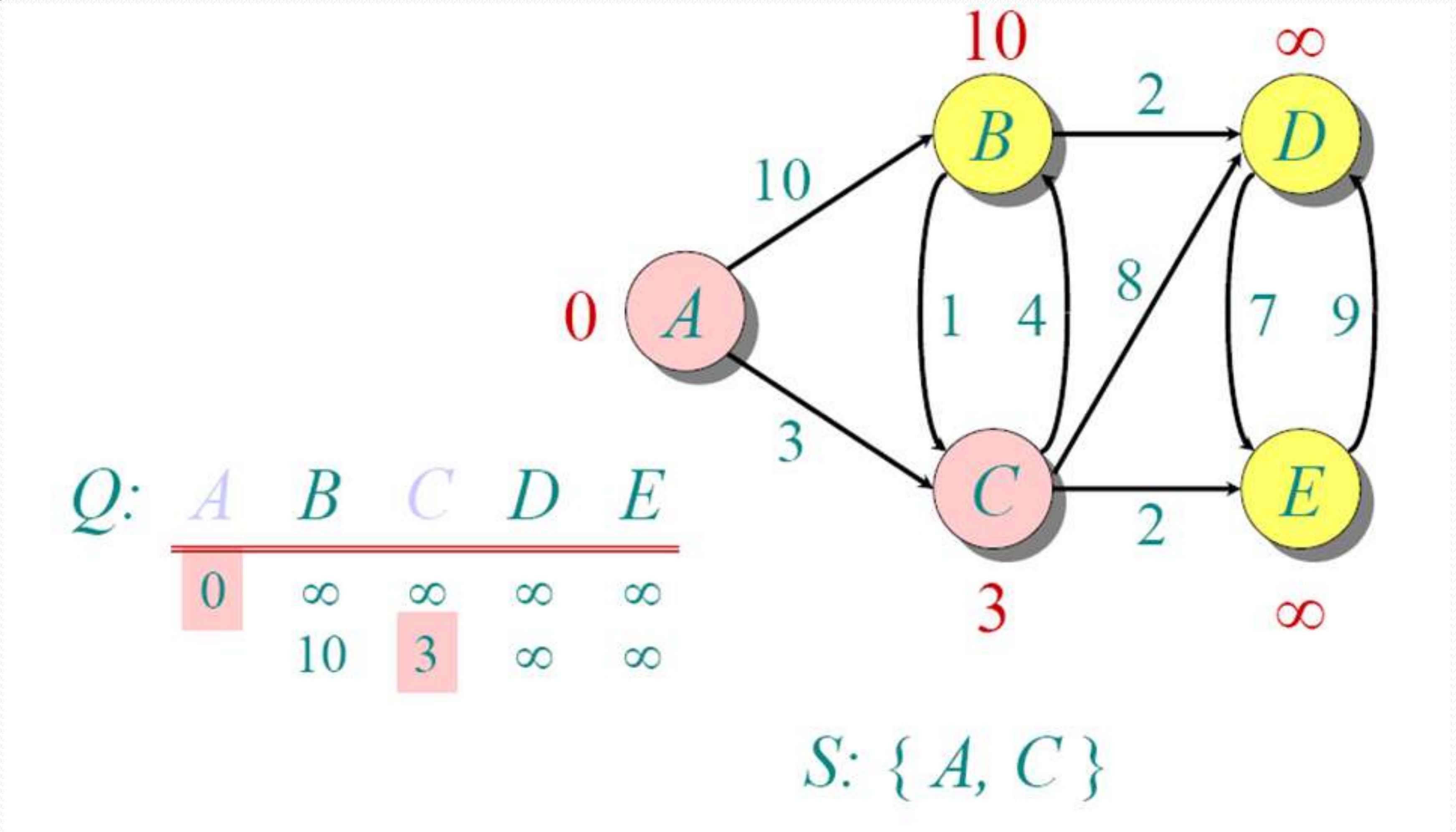
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

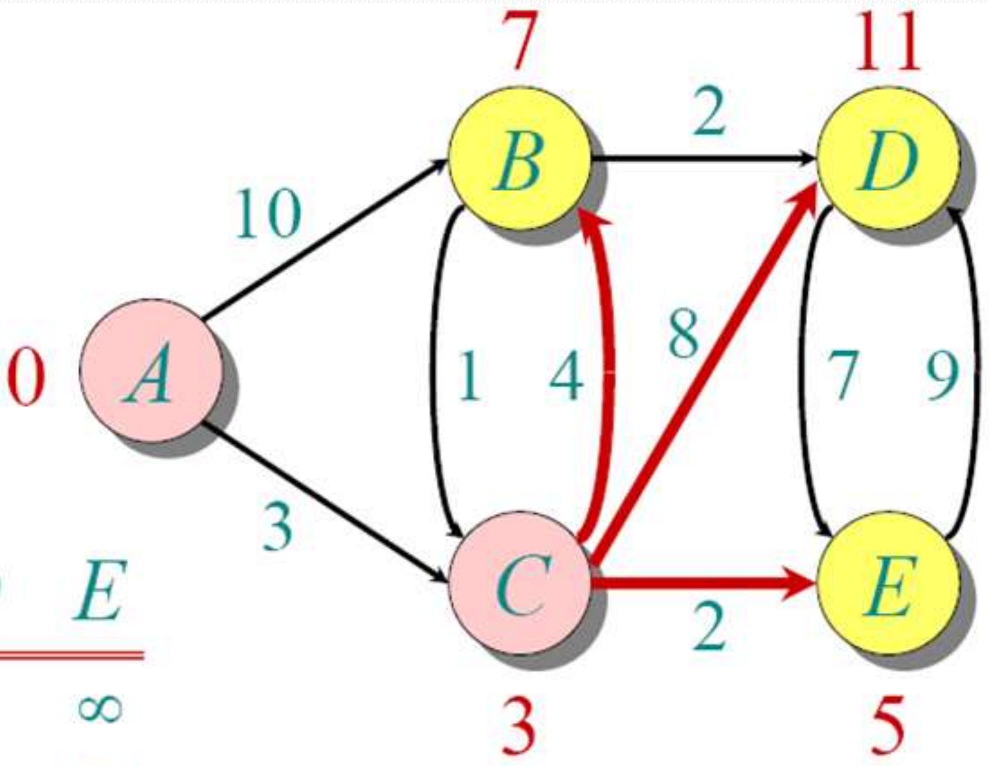
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

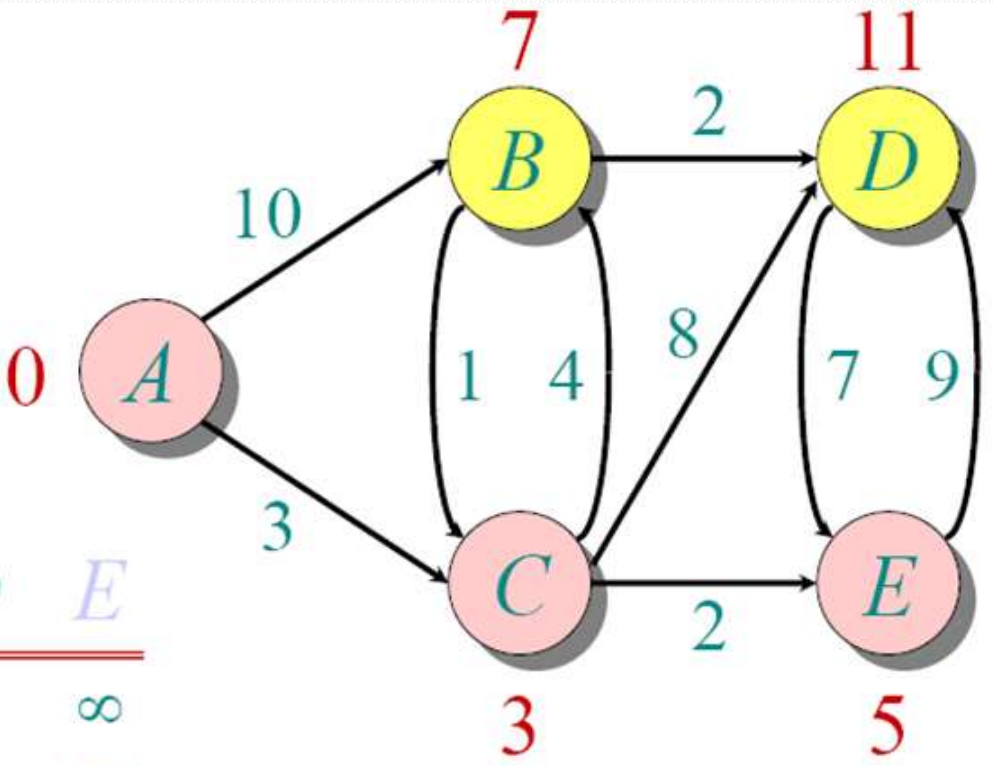
Dijkstra Animated Example



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

S: { A, C }

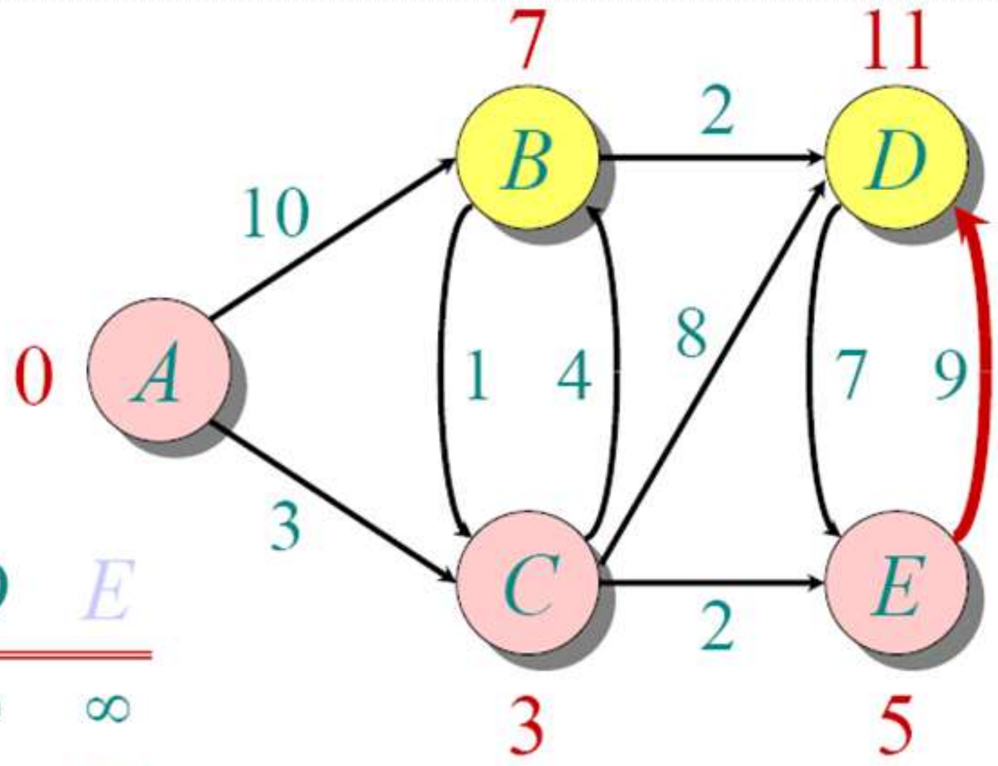
Dijkstra Animated Example



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

S: { A, C, E }

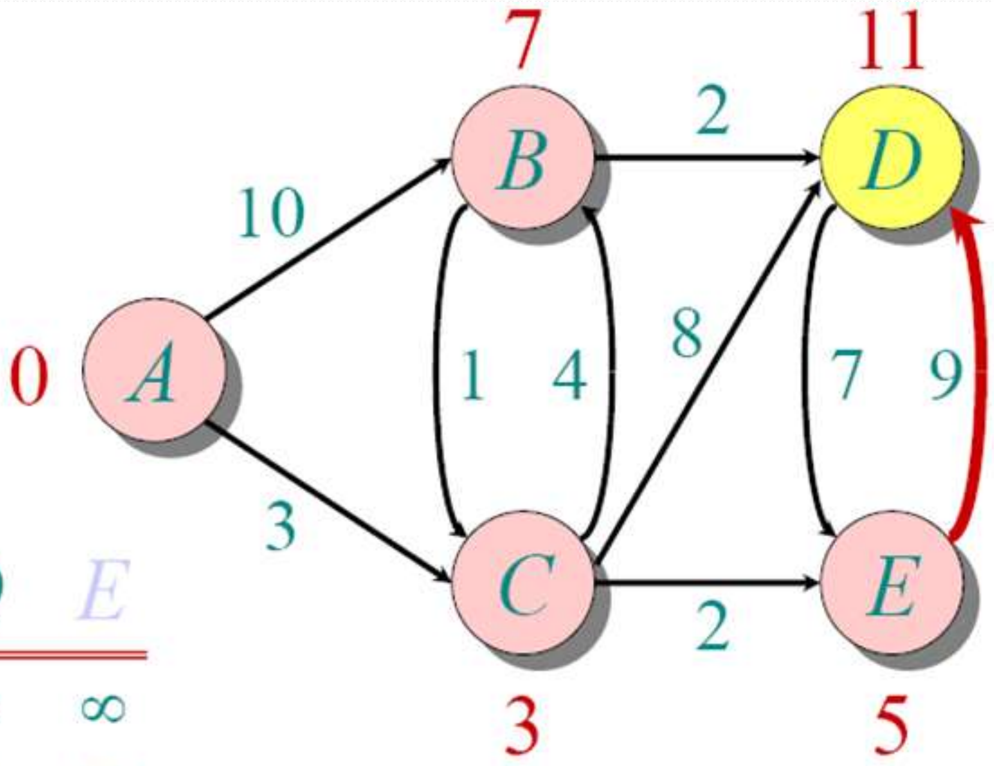
Dijkstra Animated Example



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

S: { A, C, E }

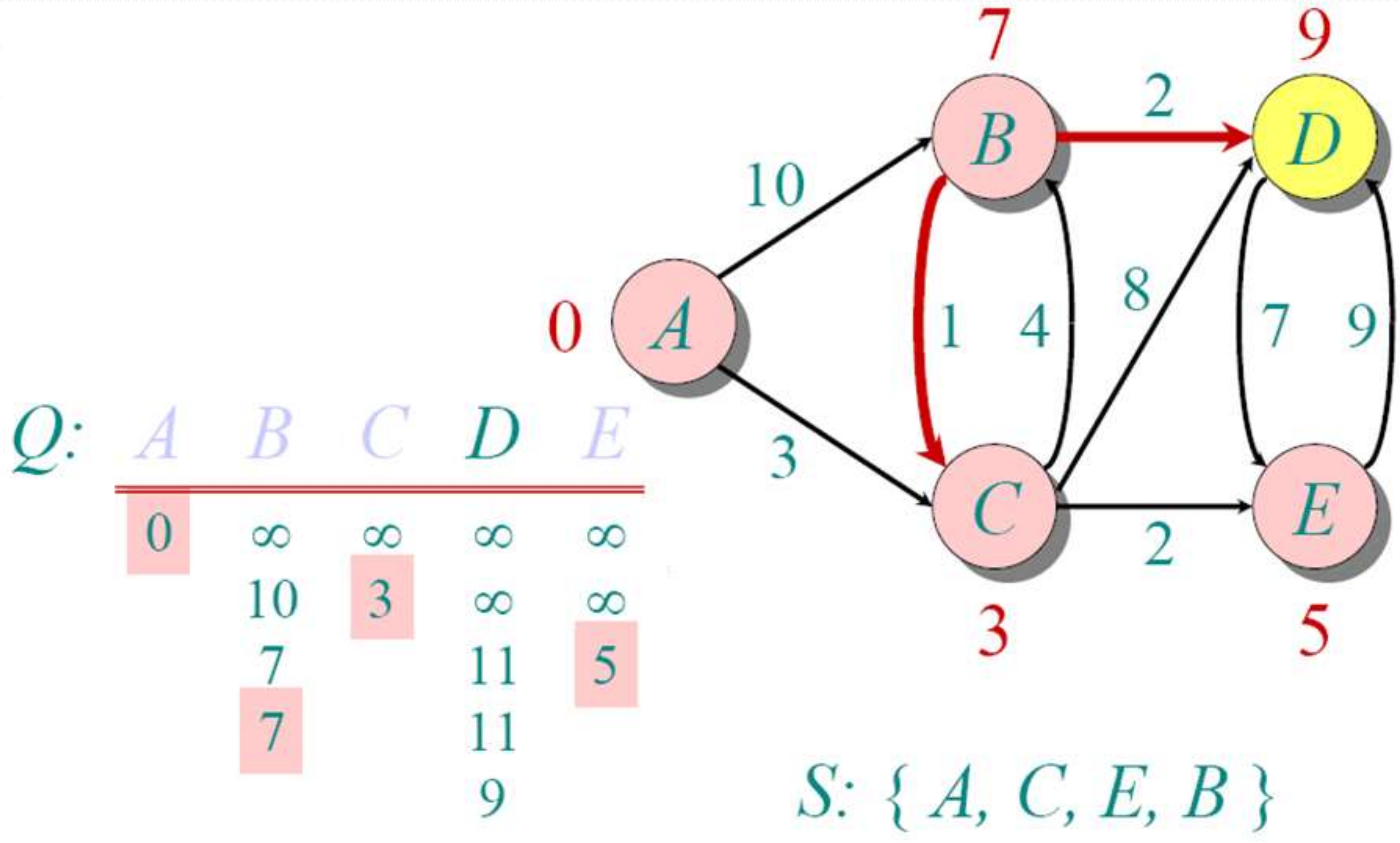
Dijkstra Animated Example



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

S: { A, C, E, B }

Dijkstra Animated Example

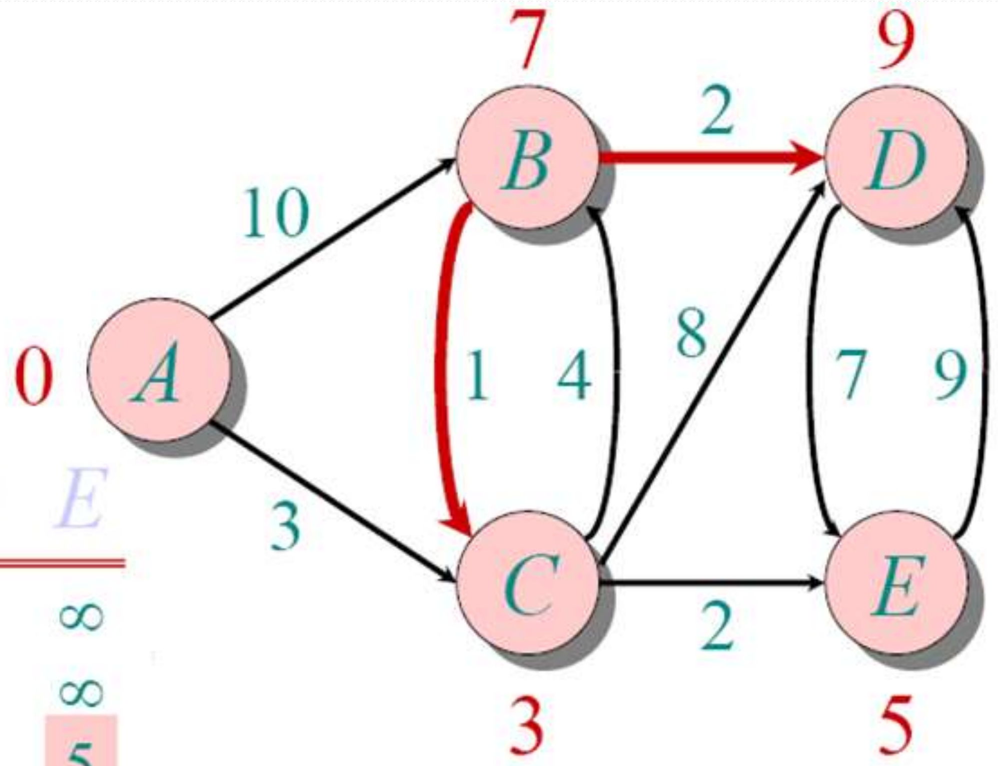


Q:

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

S: { A, C, E, B }

Dijkstra Animated Example



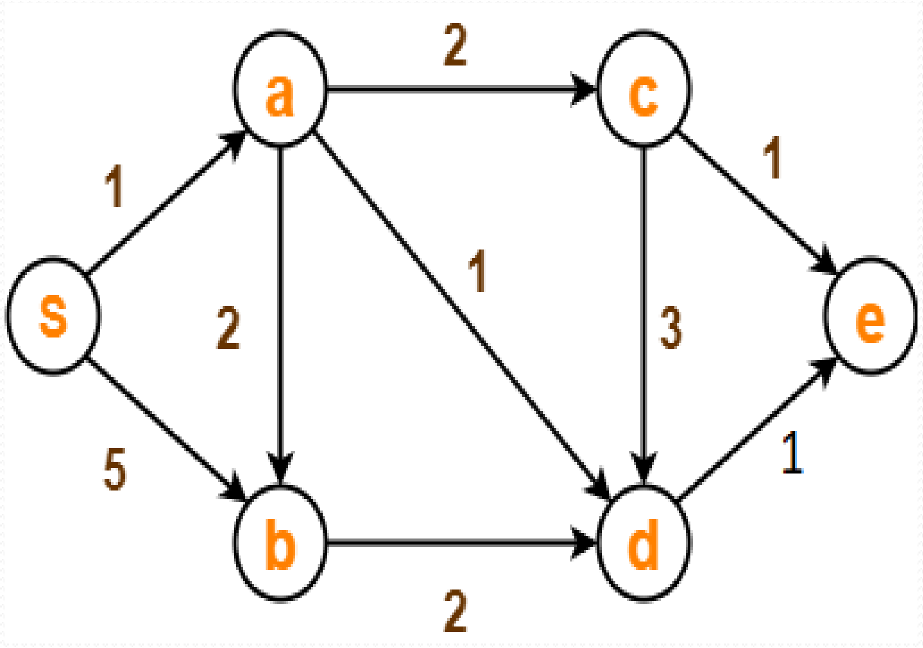
Q:

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

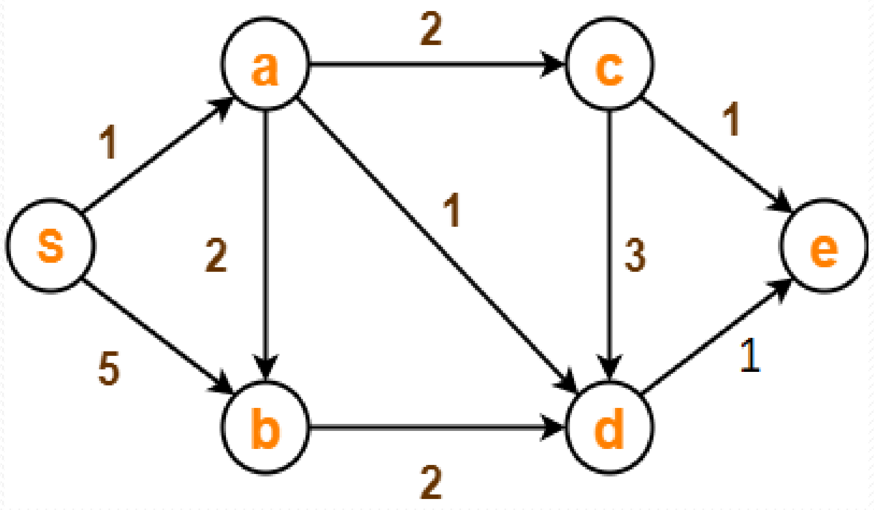
S: { *A*, *C*, *E*, *B*, *D* }

Dijkstra Example

- Step through Dijkstra's Algorithm to calculate the single source shortest path from **s** to every other vertex. You only need to show your final table, but you should show your steps in the table below for partial credit. Show your steps by crossing through values that are replaced by a new value
-



Dijkstra Example

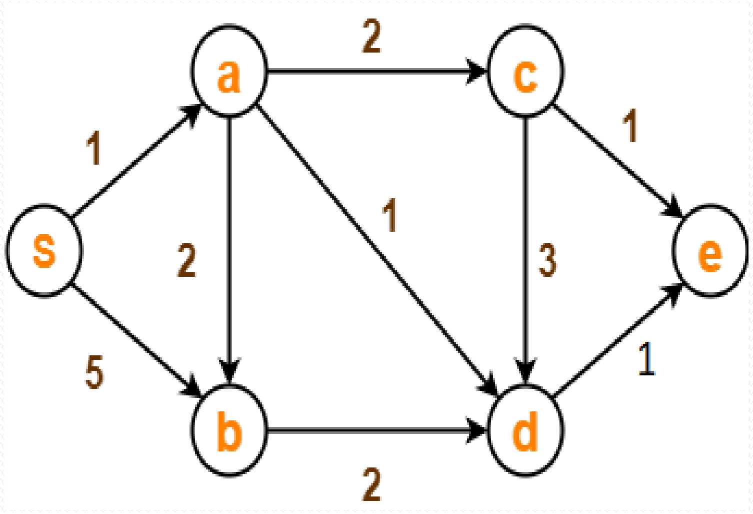


S={a}

Q	s	a	b	c	d	e

Dijkstra Example

- Step through Dijkstra's Algorithm to calculate the single source shortest path from **s** to every other vertex. You only need to show your final table, but you should show your steps in the table below for partial credit. Show your steps by crossing through values that are replaced by a new value
- S={s,a,d,b,c,e}



s	a	b	c	d	e
0	∞	∞	∞	∞	∞
	1 (s,a)	5	∞	∞	∞
		3	3	2 (s,a,d)	∞
		3 (s,a,b)			3
			3 (s,a,c)		
					3 (s,a,d,e)

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

$O(|V|^2 + |E|)$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can’t sleep unless you see a proof, see the second reference or ask us where you can find it.

Dijkstra’s Algorithm

- Graph *G*, weight function *w*, root *s*

```
DIJKSTRA($G, w, s$)
 1  for each $v \in V$
 2          do $d[v] \leftarrow \infty$
 3  $d[s] \leftarrow 0$
 4  $S \leftarrow \emptyset$    $\triangleright$ Set of discovered nodes
 5  $Q \leftarrow V$
 6  while $Q \neq \emptyset$
 7          do $u \leftarrow$ EXTRACT-MIN($Q$)
 8             $S \leftarrow S \cup \{u\}$
 9             for each $v \in Adj[u]$
10                 do if $d[v] > d[u] + w(u, v)$
11                       then $d[v] \leftarrow d[u] + w(u, v)$
```

relaxing edges