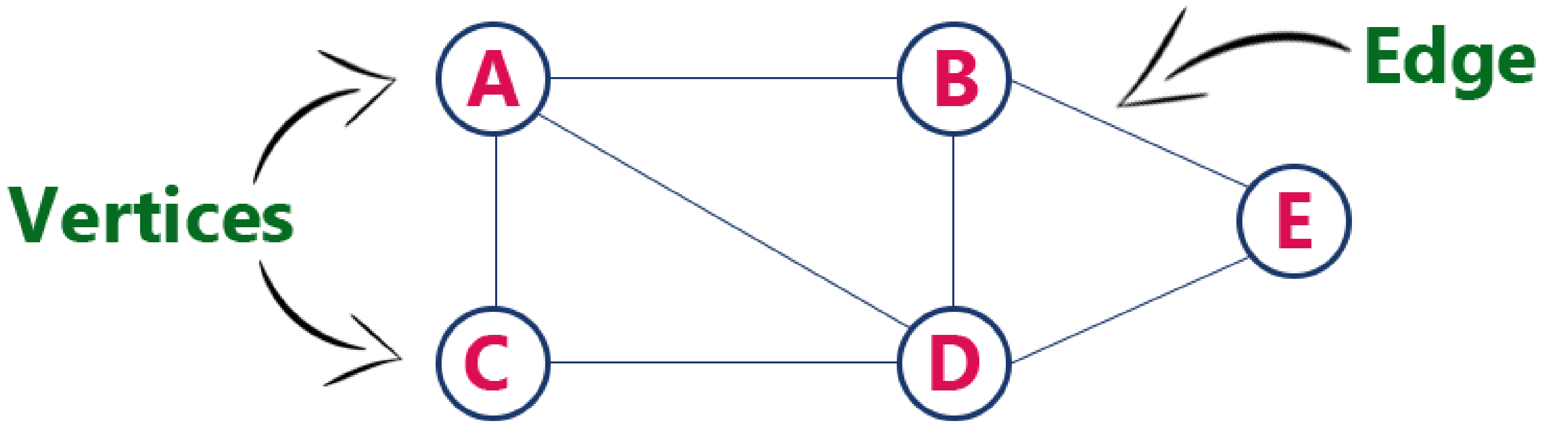
CS141: Intermediate Data Structures and Algorithms

Graphs

Amr Magdy

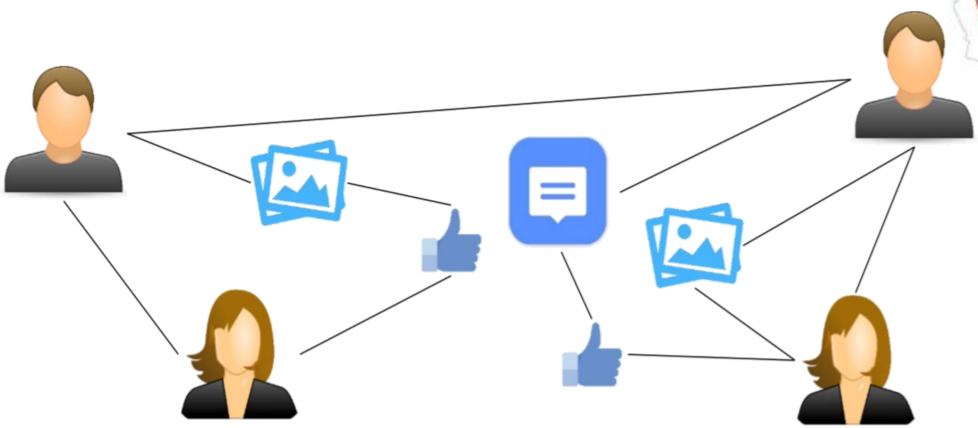
Graph Data Structure

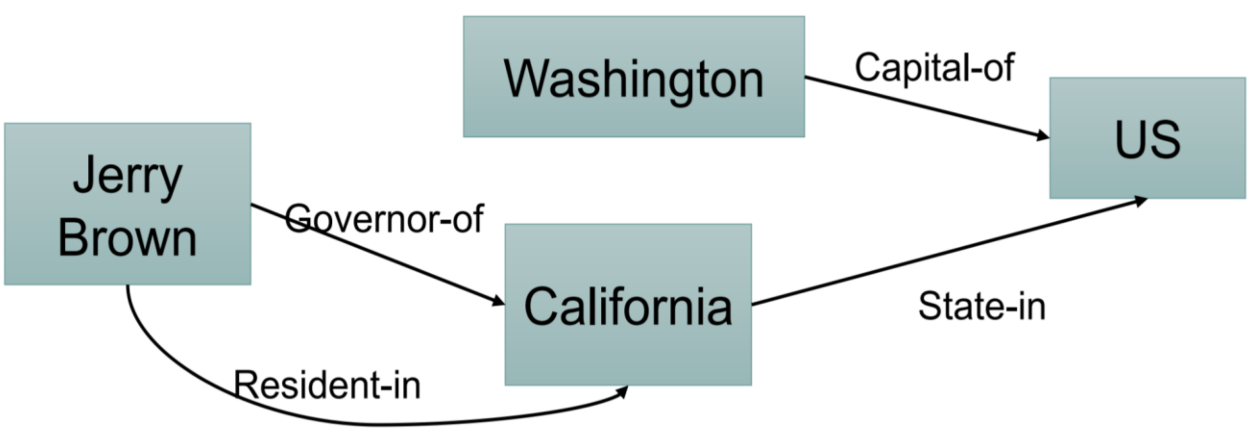
- A set of nodes (vertices) and edges connecting them



Graph Applications

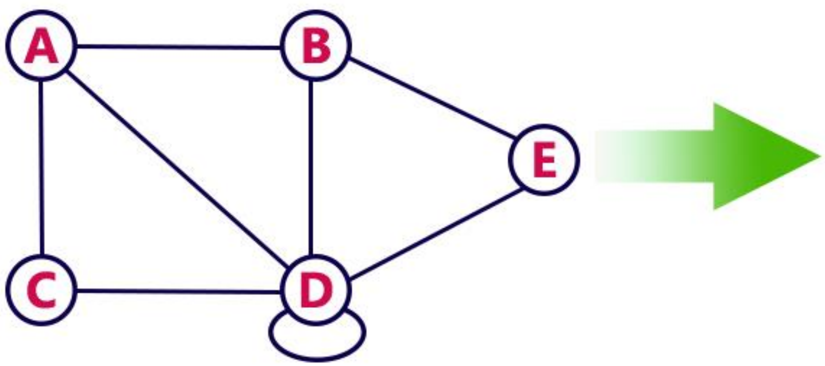
- Road network
- Social media networks
- Knowledge bases





Graph Representations

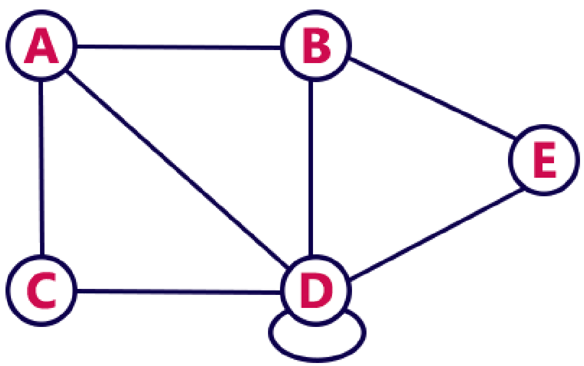
- Adjacency matrix
  - Storage and access efficient when many edges exist



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Graph Representations

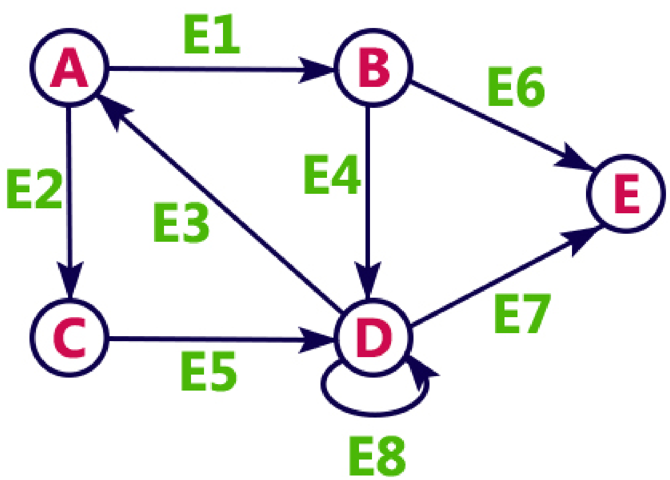
- Adjacency matrix
  - Storage and access efficient when many edges exist



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Graph Representations

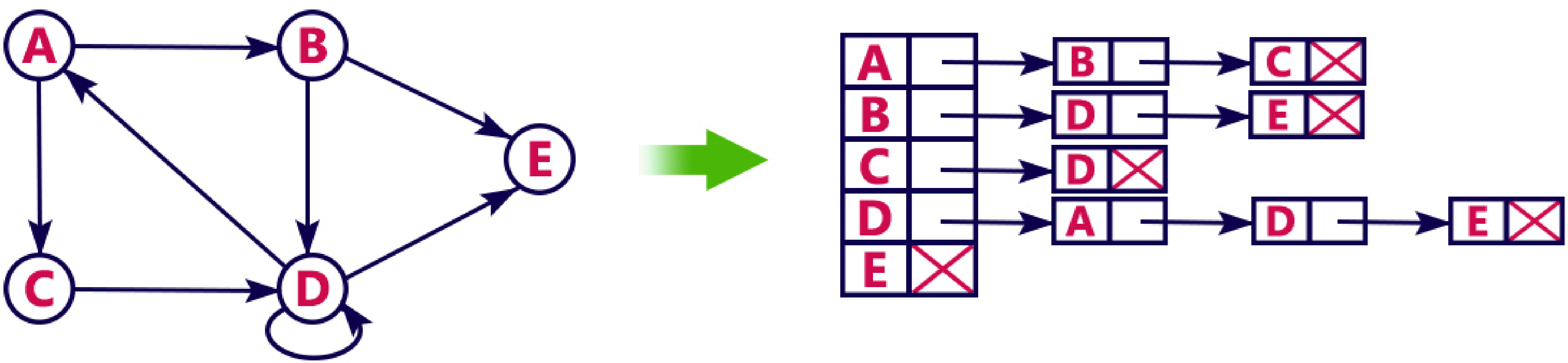
- Incidence Matrix
  - Expensive storage, not popular



	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

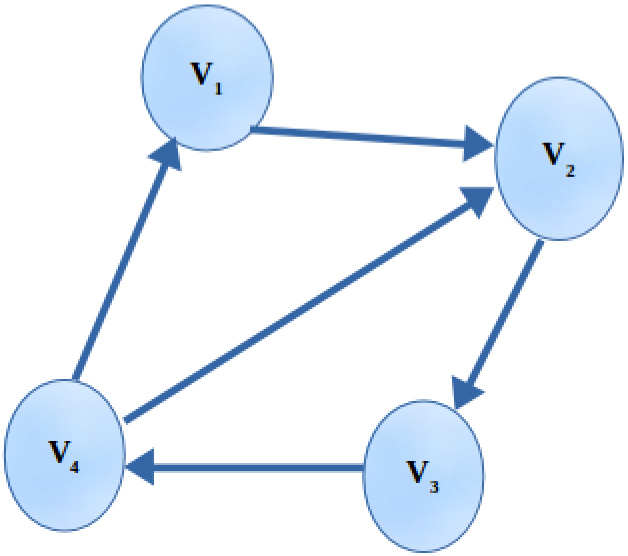
Graph Representations

- Adjacency list
  - Storage efficient when few edges exit (sparse graphs)
  - Sequential access to edges (vs random access in matrix)

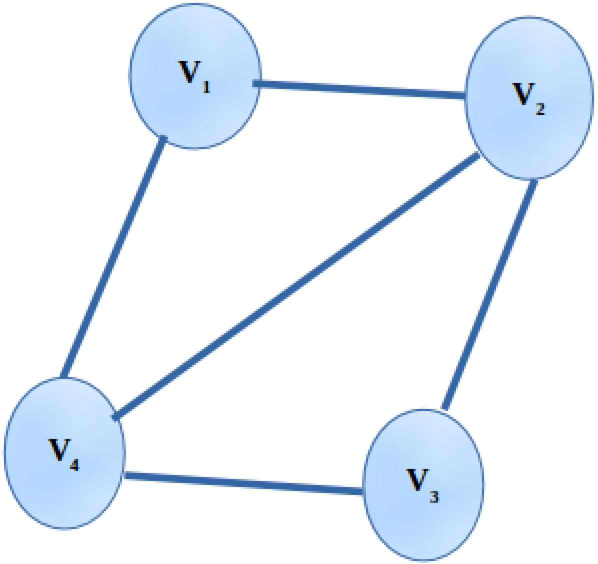


Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest



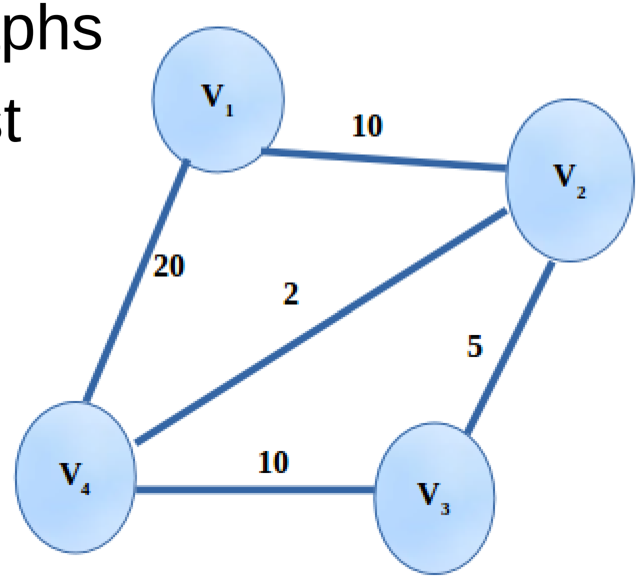
Directed Graph



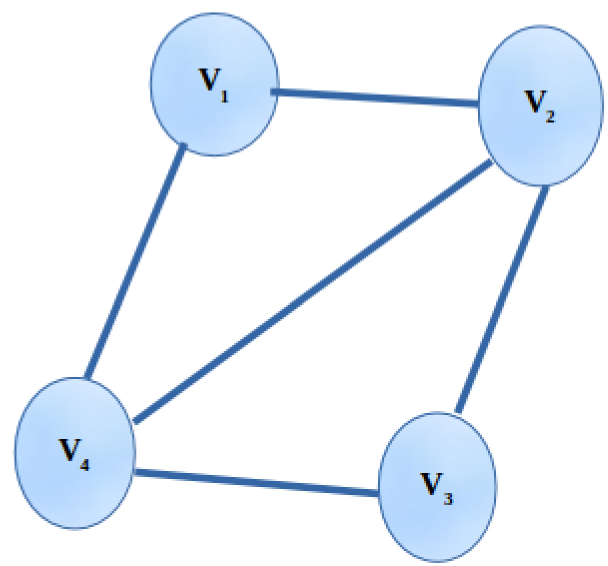
Undirected Graph

Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest



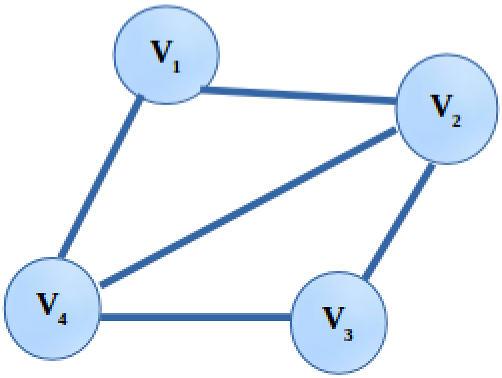
Weighted Graph



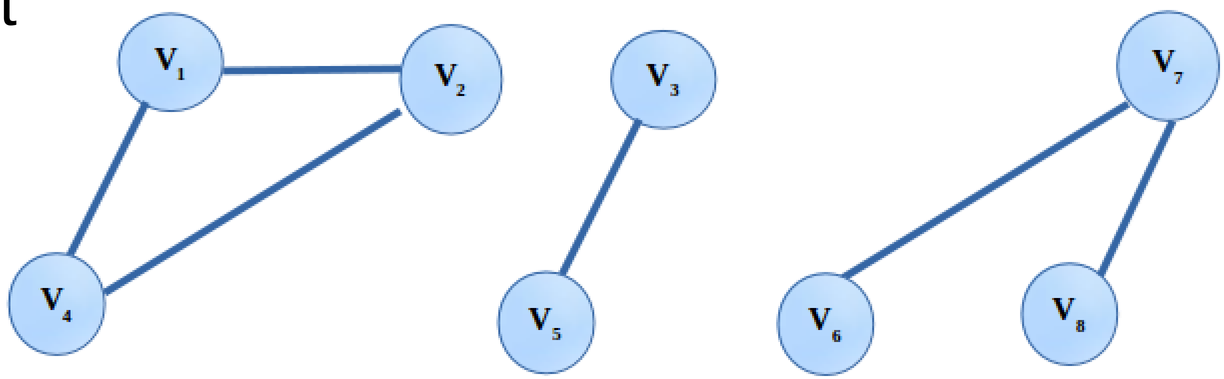
Unweighted Graph

Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest



Fig(i): Connected Graph

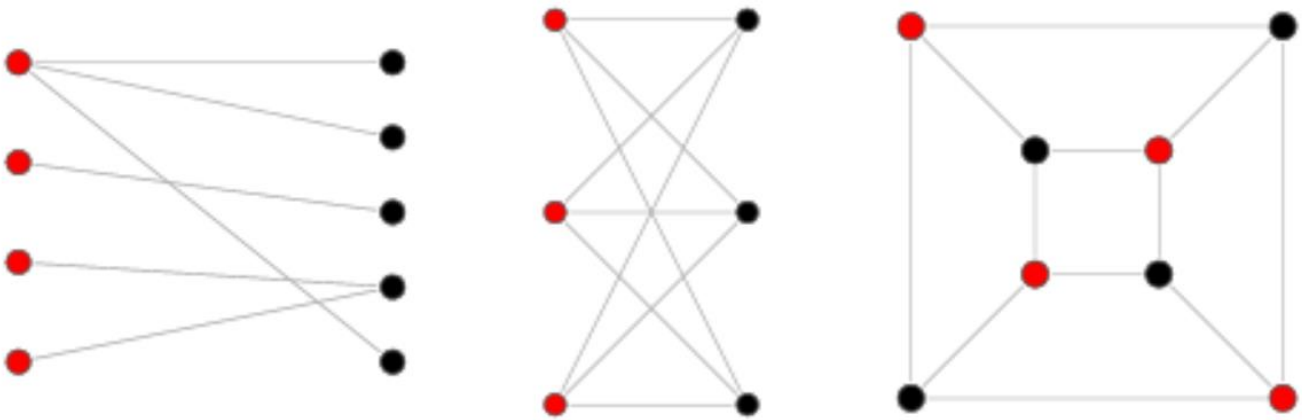


Fig(ii): Unconnected Graph

There are three component of above unconnected graph

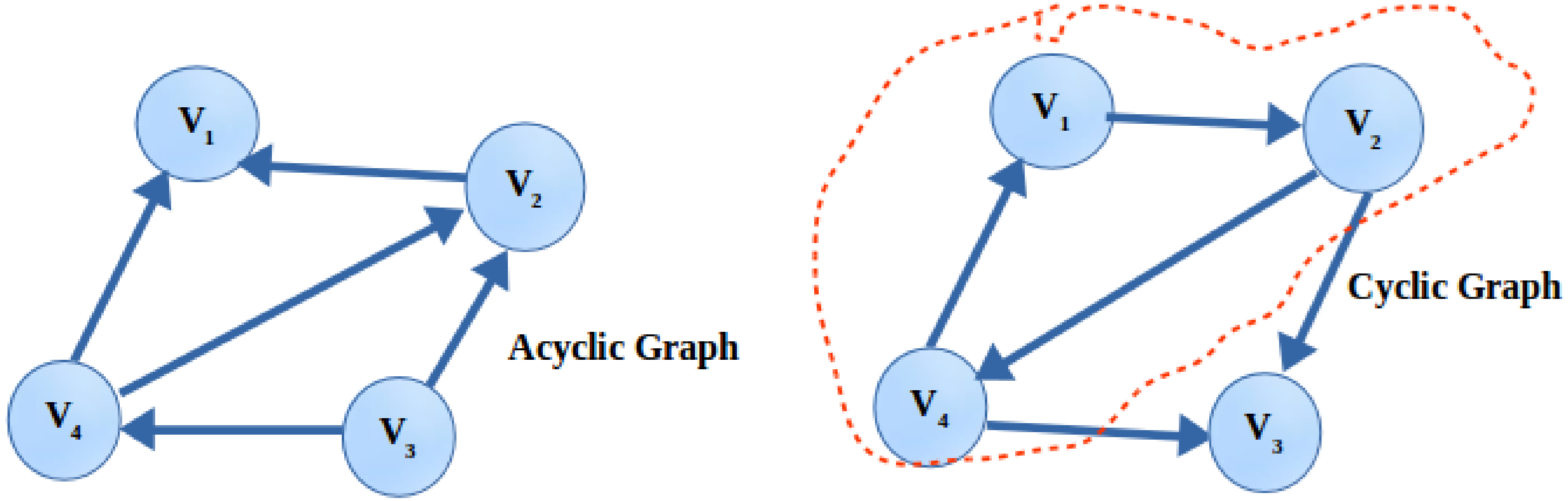
Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest



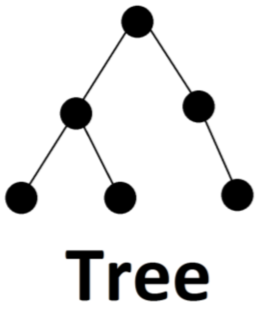
Types of Graphs

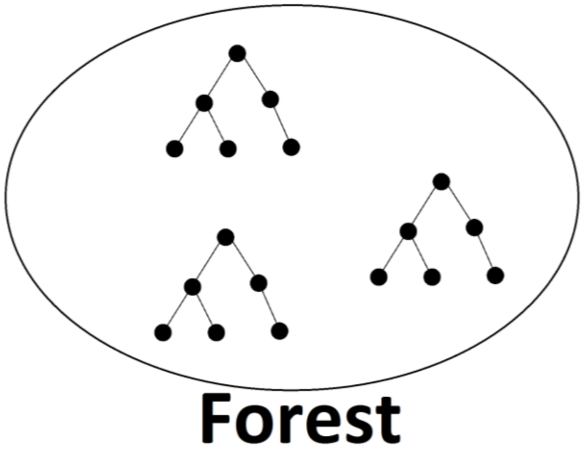
- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest



Types of Graphs

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs
- Tree/Forest
  - Tree: directed acyclic graph with max of one path between any two nodes
  - Forest: set of disjoint trees



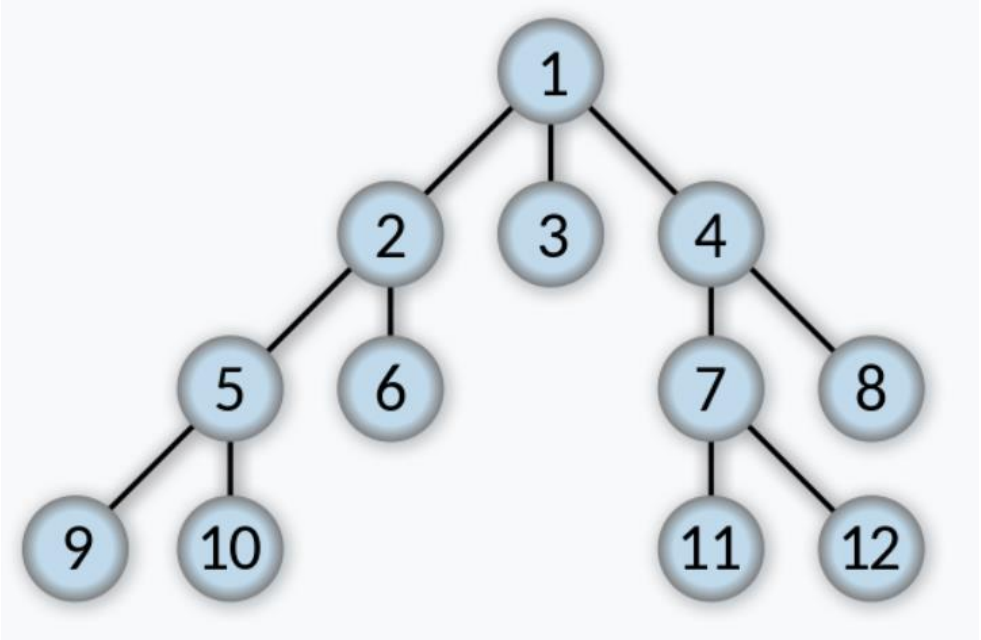


Basic Graph Algorithms

- Graph traversal algorithms
  - Bread-first Search (BFS)
  - Depth-first Search (DFS)
- Topological Sort
- Graph Connectivity
- Cycle Detection

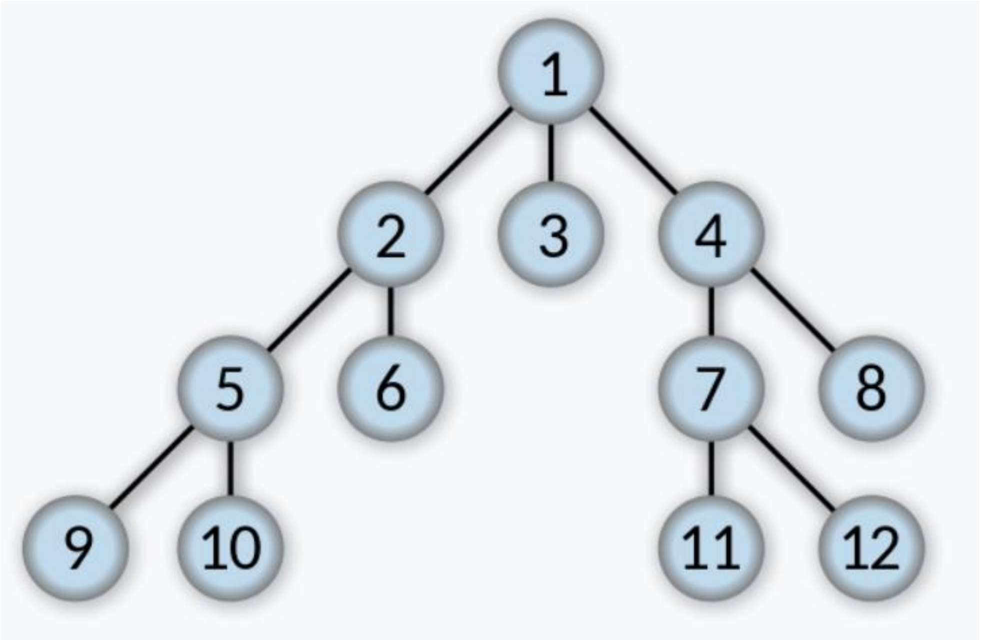
Breadth-first Search (BFS)

- How to traverse?



Breadth-first Search (BFS)

- How to traverse?
- Use a queue



Breadth-first Search (BFS)

- How to traverse?
- Use a queue
- Start at a vertex s

  Mark s as visited

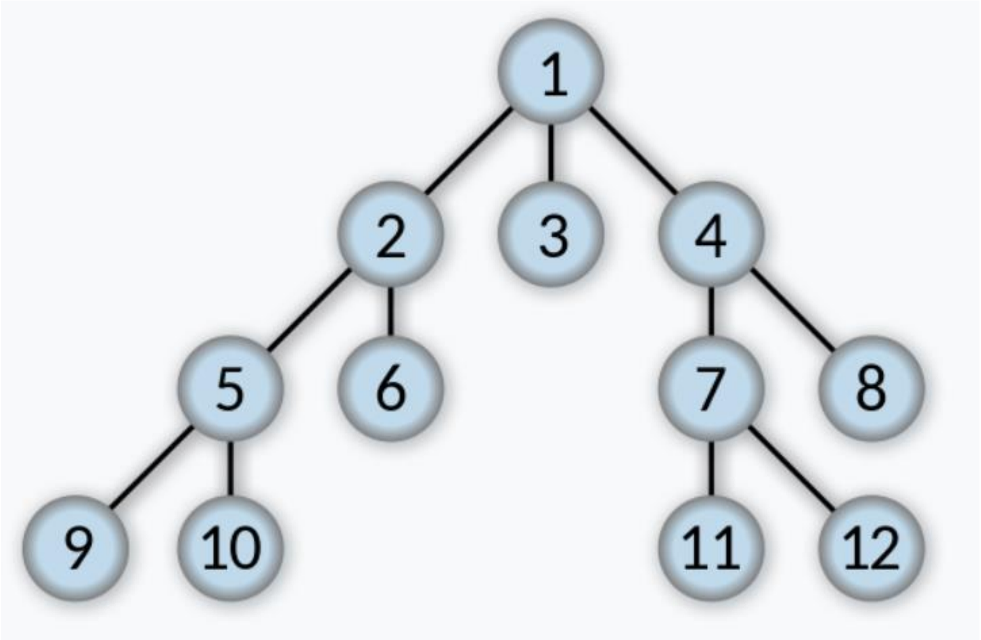
  Enqueue neighbors of s

  while Q not empty

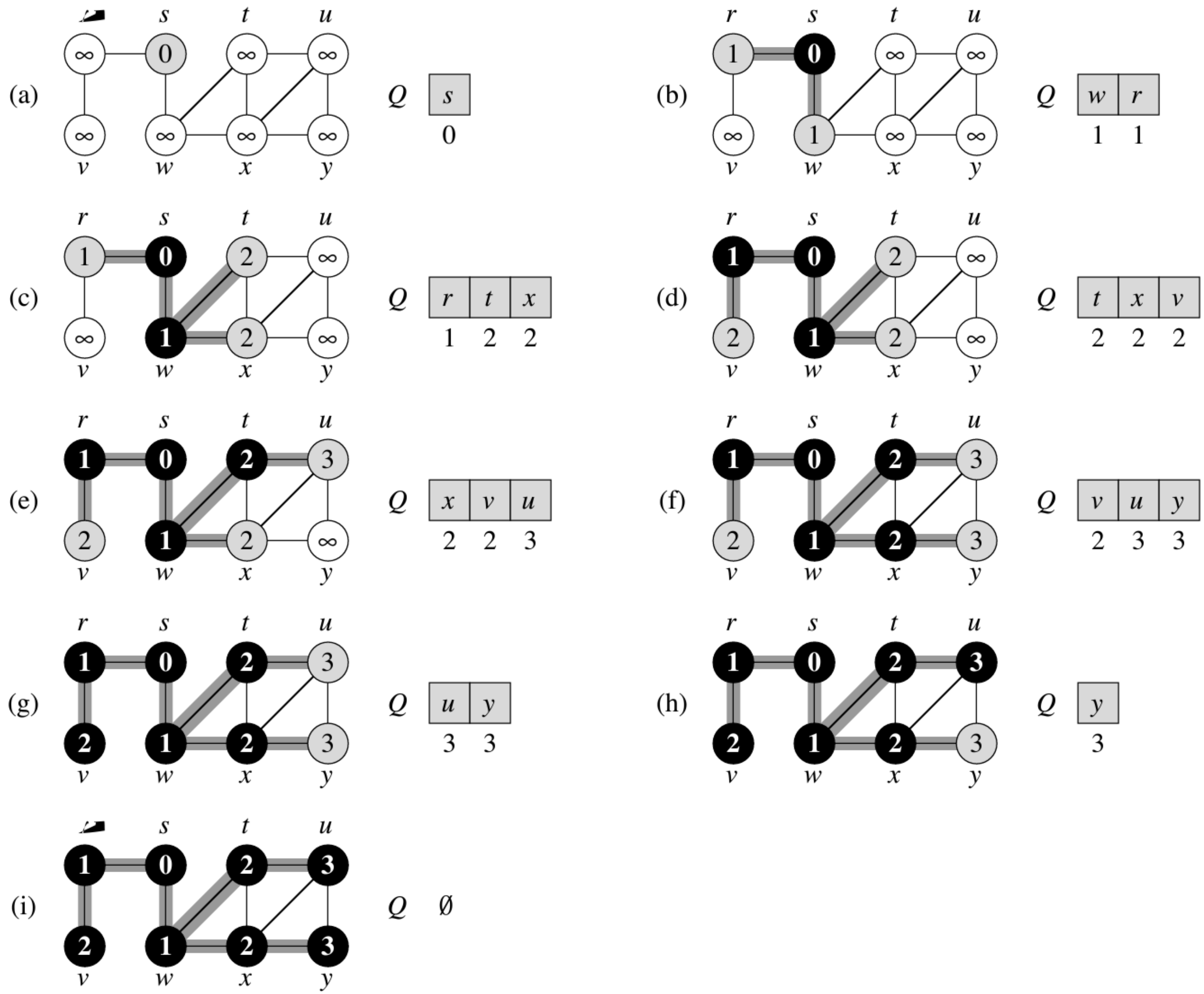
    Dequeue vertex u

    Mark u as visited

    Enqueue unvisited neighbors of u

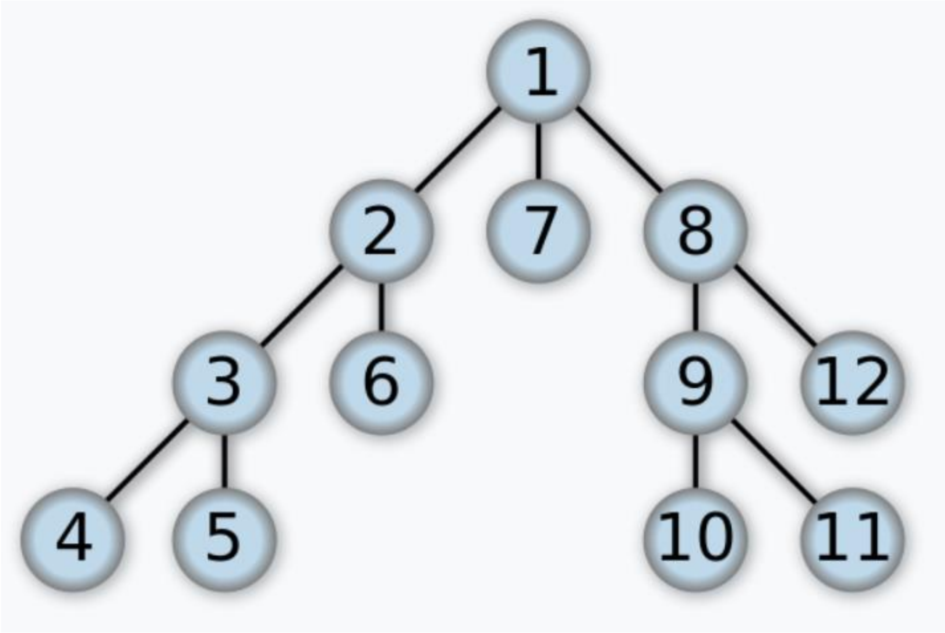


Breadth-first Search (BFS)



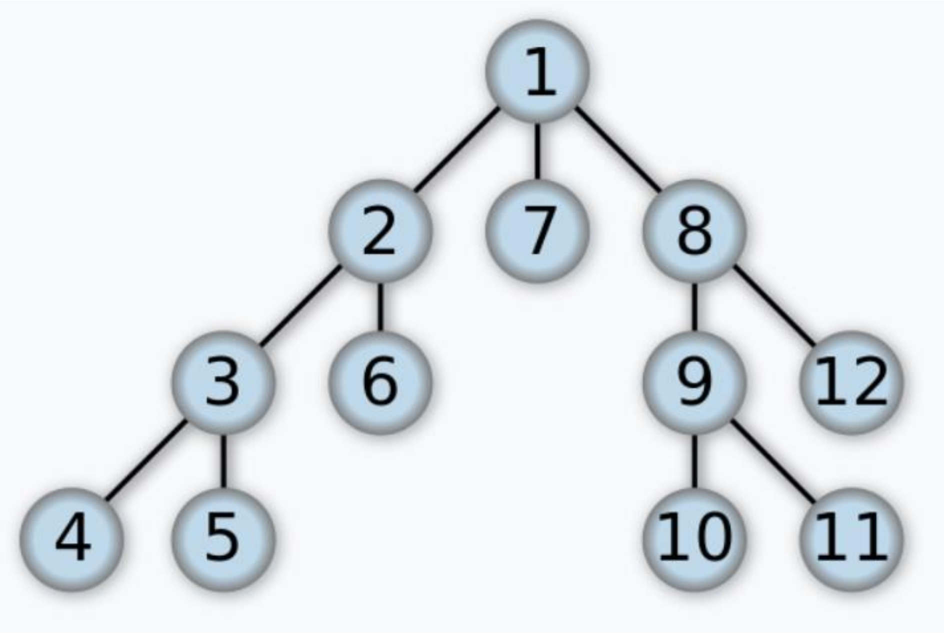
Depth-first Search (DFS)

- How to traverse?



Depth-first Search (DFS)

- How to traverse?
- Use a stack



Depth-first Search (DFS)

- How to traverse?
- Use a stack
- Start at a vertex s

  Mark s as visited

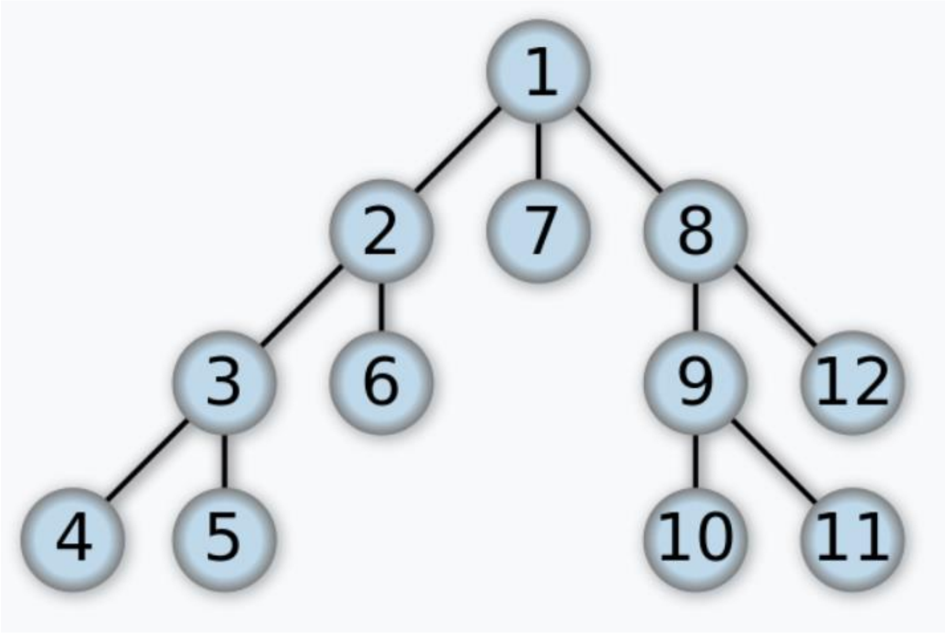
  Push neighbors of s

  while Stack not empty

      Pop vertex u

      Mark u as visited

      Push unvisited neighbors of u



Complexity of Graph Traversal

- For G = (V,E), V set of vertices, E set of edges
- BFS
  - Time: O(|V|+|E|)
  - Space: O(|V|) (plus graph representation)
- DFS
  - O(|V|+|E|)
  - Space: O(|V|) (plus graph representation)

Graph Connectivity

- Checking if graph is connected:

Graph Connectivity

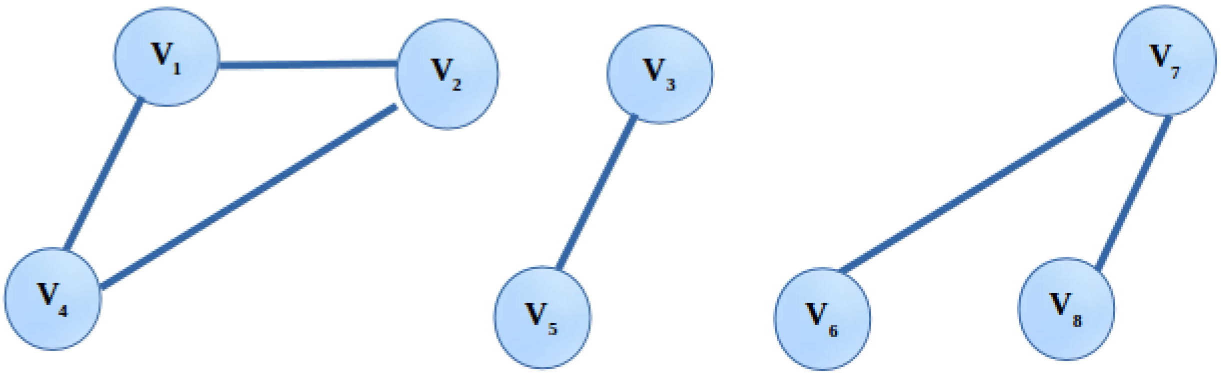
- Checking if graph is connected:

```
IsConnected(G)
{
        DFS(G)
        if any vertex not visited
                return false
        else
                return true
}
```

Time Complexity: O(|V|+|E|)

Graph Connected Components

- Getting the graph connected components



Fig(ii):
Unconnected Graph

There are three component of above unconnected graph

Graph Connected Components

- Getting the graph connected components
- Mark all nodes as unvisited

visitCycle = 1

while(there exists unvisited node n)

{

- Start DFS(G) at n, mark visited node with *visitCycle*

- Output all nodes with current visitCycle as one connected component

- *visitCycle* = *visitCycle*+1

}

Time Complexity: O(|V|+|E|)

Cycle Detection

- Does a connected graph G contain a cycle? (non-trivial cycle)

Cycle Detection

- Does a connected graph G contain a cycle? (non-trivial cycle)
- General idea: if DFS procedure tries to revisit a visited node, then there is a cycle

Cycle Detection

- Does a graph G contain a cycle? (non-trivial cycle)

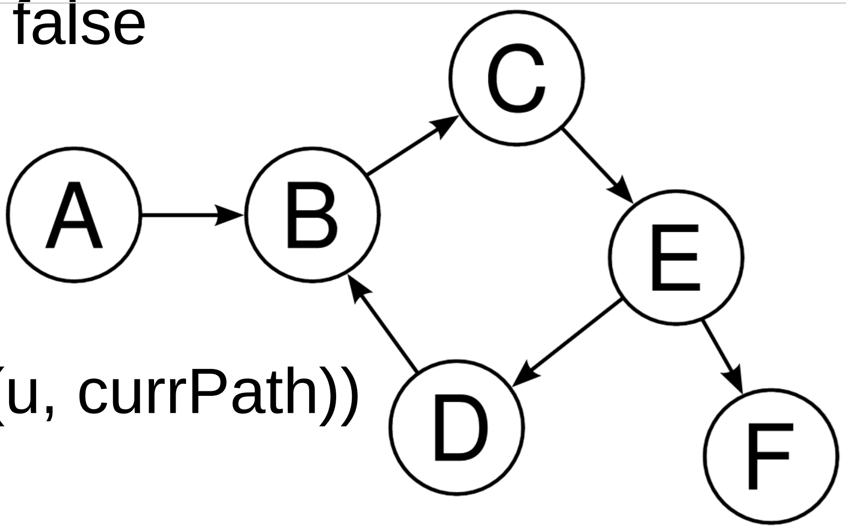
```
IsAcyclic(G) {
        Start at unvisited vertex s
        Mark “s” as visited
        Push neighbors u of s in stack <node:u, parent:s>
        while stack not empty
                Pop vertex u
                Mark u as visited
                if u has a visited neighbor v
                        & v is non-parent for u
                        return true
                Push unvisited neighbors v of u <node:v, parent:u>
        return false
```

Cycle Detection

- Does a connected graph G contain a cycle? (non-trivial cycle)
- General idea: if DFS procedure tries to revisit a visited node, then there is a cycle
- Why checking if v non-parent for u?
  - To eliminate trivial cycles, a cycle that involve only two nodes

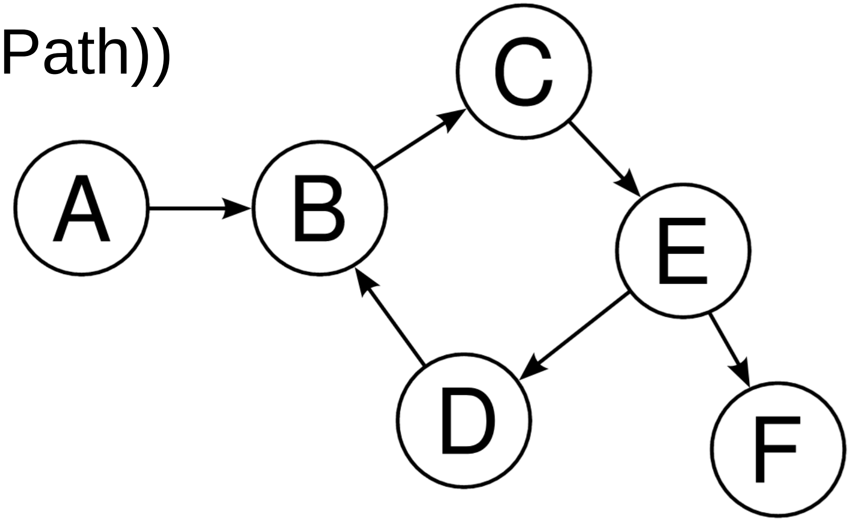
Cycle Detection in Directed Graphs

```
IsAcyclicDirected(node s, currPath) {
        if s in currPath          return true
        if s is visited           return false
        Mark s as visited
        Add s to currPath
        for each neighbor u of s
                if(IsAcyclicDirected(u, currPath))
                        return true
        remove s from currPath
        return false
}
```



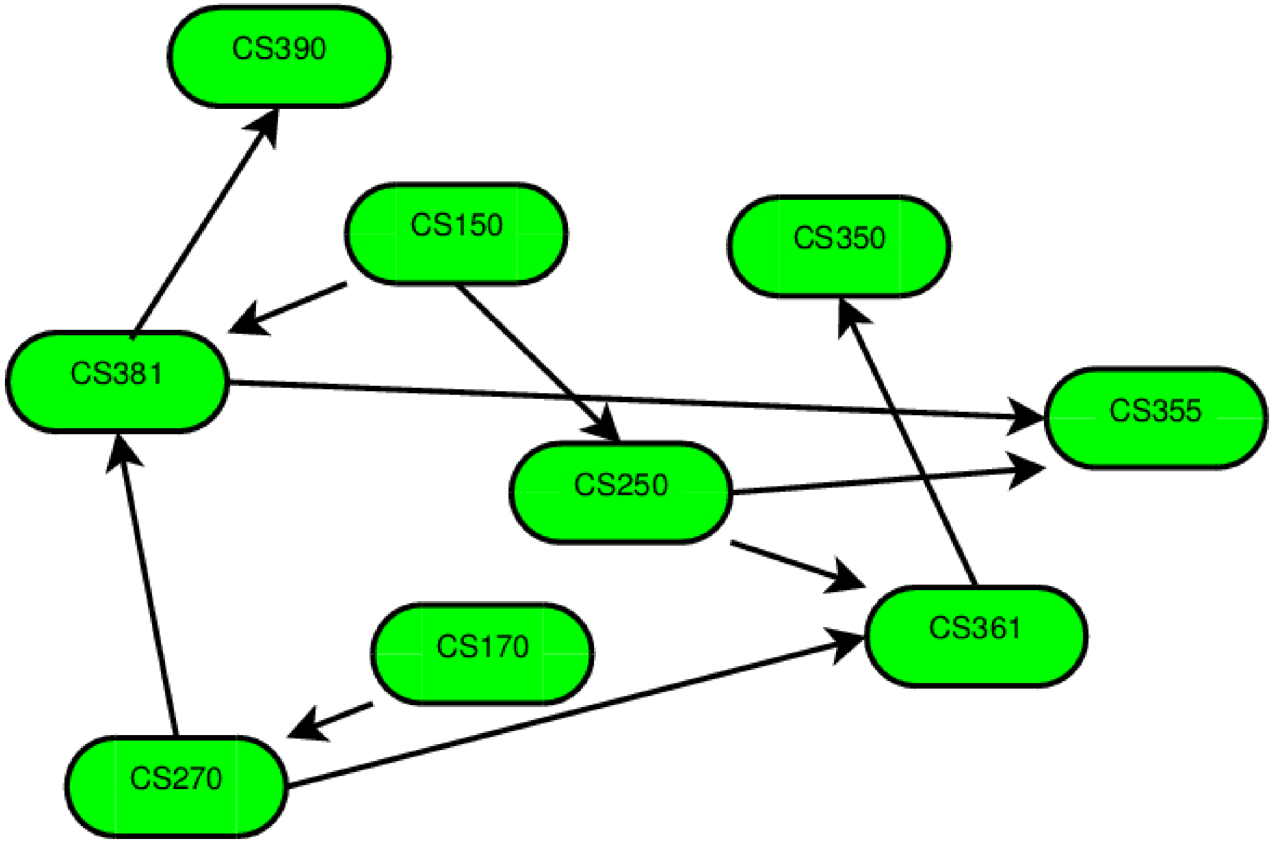
Cycle Detection in Directed Graphs

```
while(there is unvisited node s)
{
        currPath = {}
        if(IsAcyclicDirected(s, currPath))
                return true
}
return false
```



Topological Sort

- Determine a linear order for vertices of a directed acyclic graph (DAG)
  - Mostly dependency/precedence graphs
  - If edge (u,v) exists, then u appears before v in the order



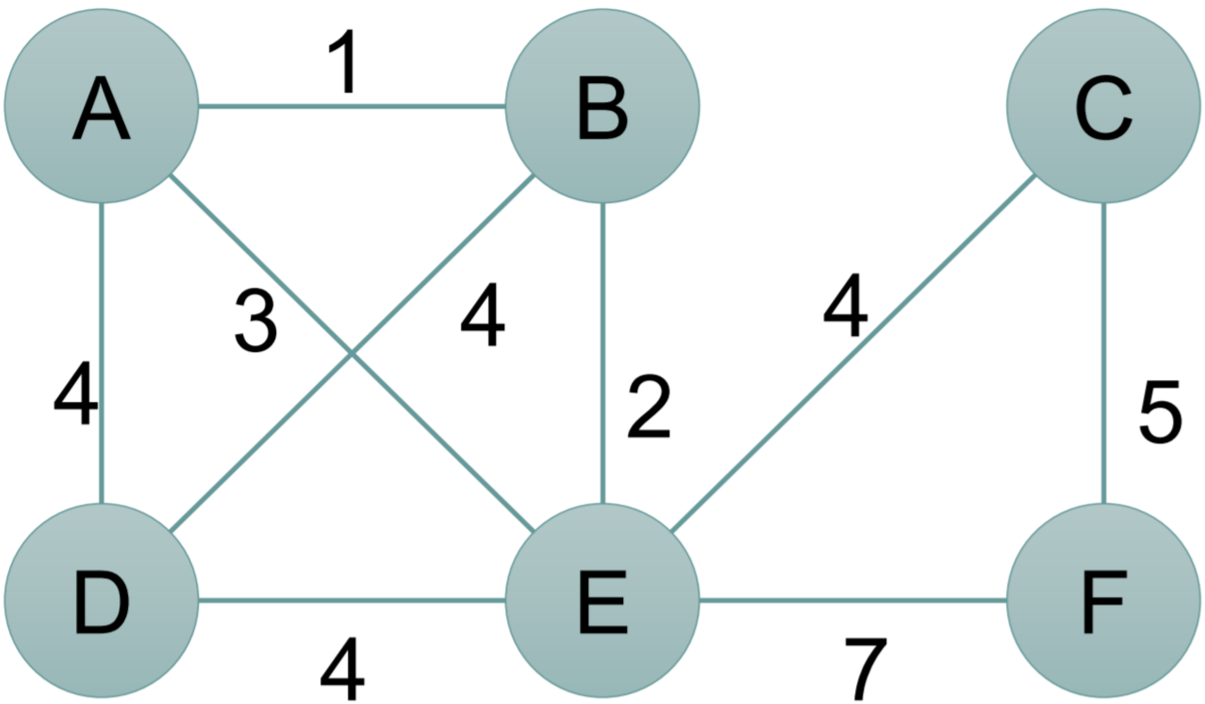
Topological Sort

```
L ← Empty list
S ← Set of all nodes with no incoming edge
while S is non-empty do
    remove a node n from S
    add n to end of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
return L (a topologically sorted order)
```

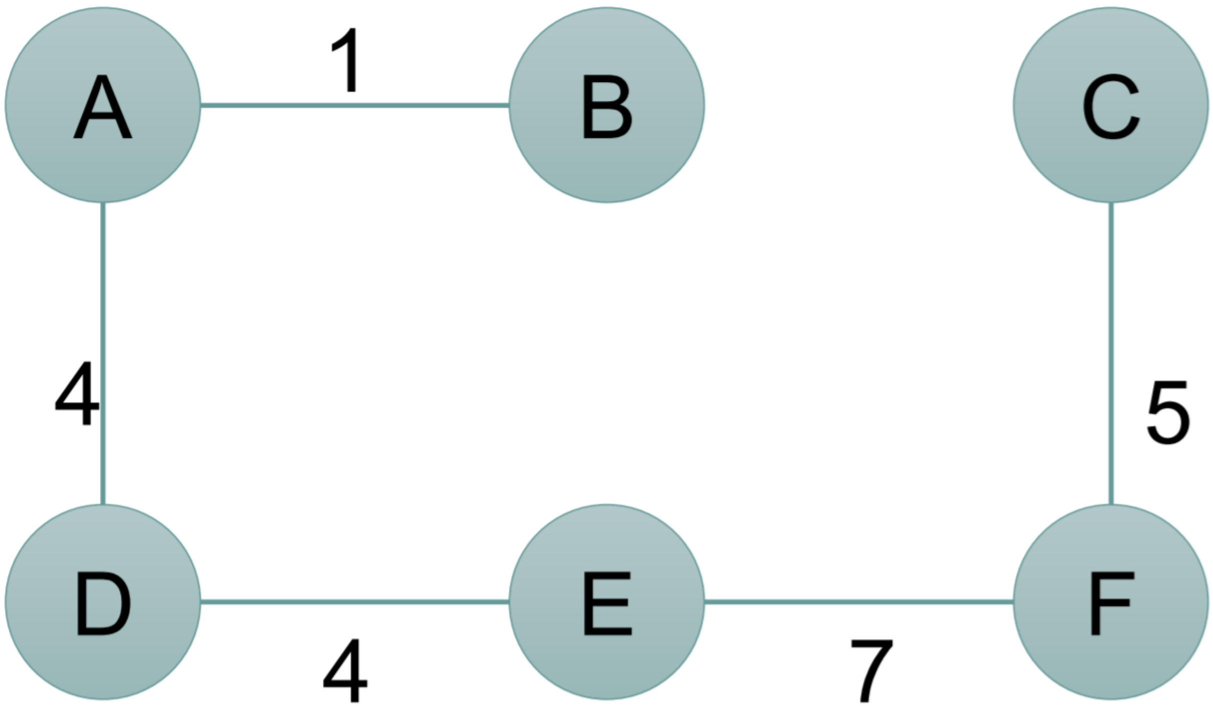
Spanning Tree

- Given a connected graph G=(V,E), a spanning tree T ⊆ E is a set of edges that “spans” (i.e., connects) all vertices in V.
- A **Minimum Spanning Tree (MST)**: a spanning tree with minimum total weight on edges of T
- Application:
  - The wiring problem in hardware circuit design

Spanning Tree: Example

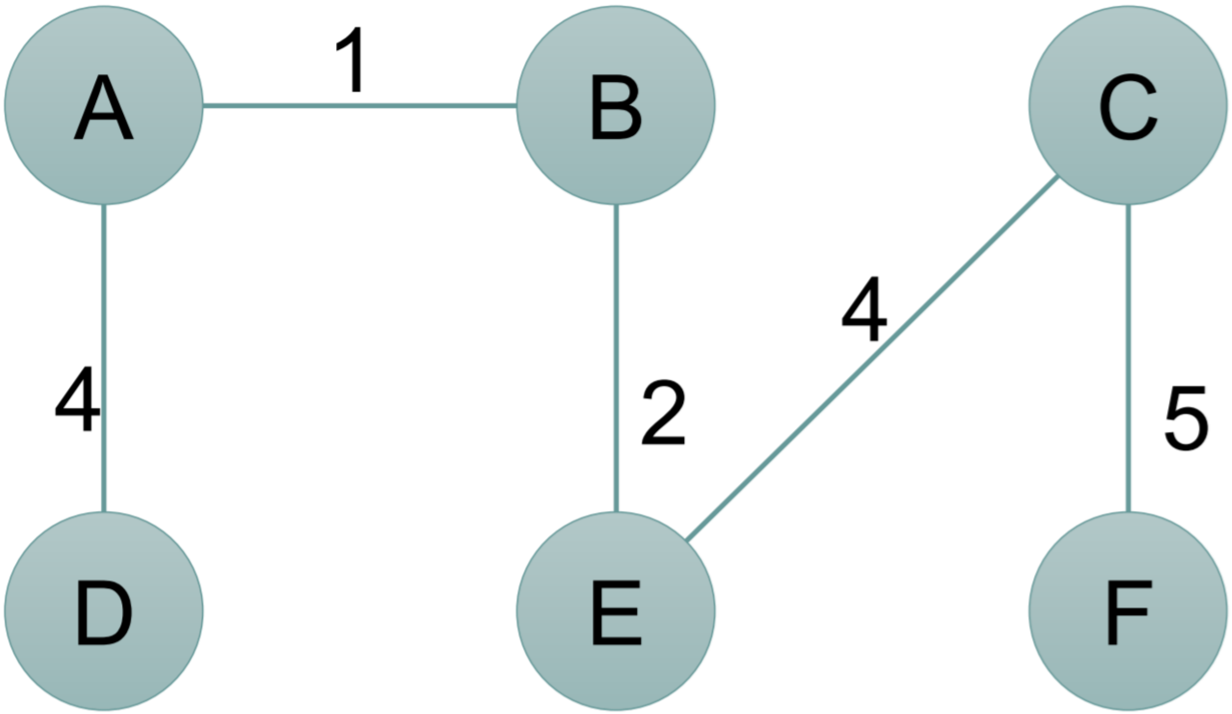


Spanning Tree: Not MST



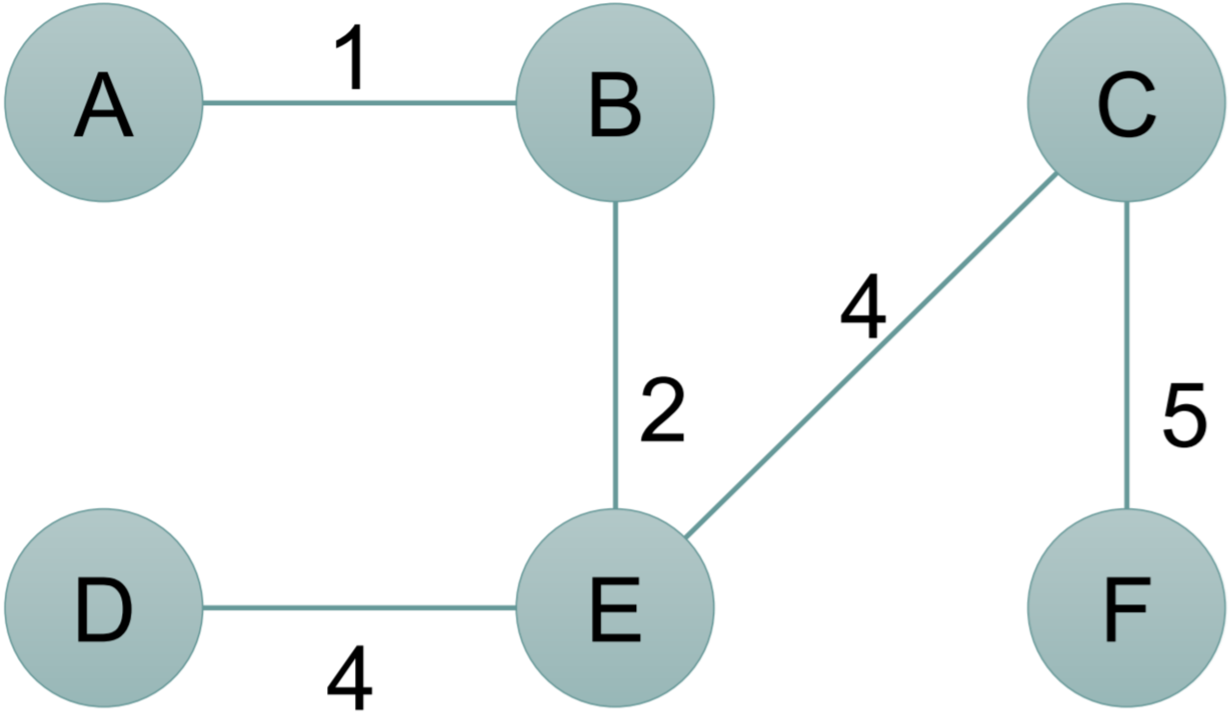
Total weight = 21

Spanning Tree: MST



Total weight = 16

Spanning Tree: Another MST

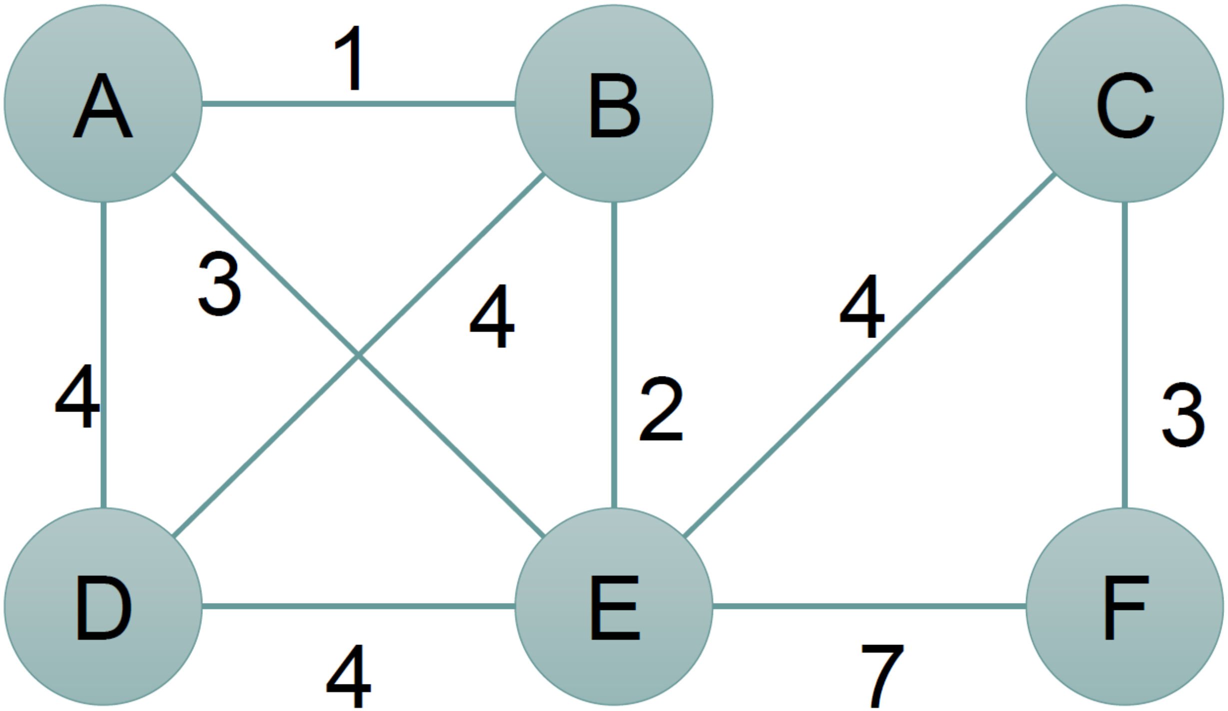


Total weight = 16

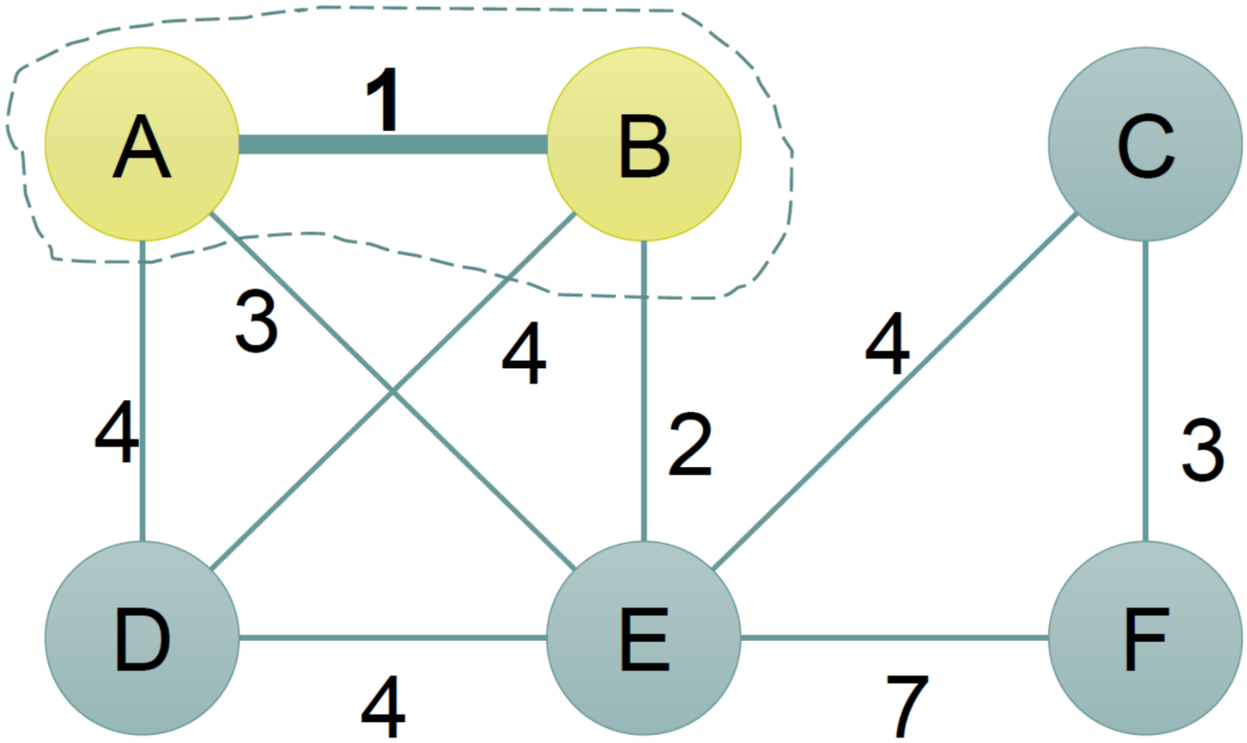
Finding MST: Kruskal's algorithm

- Sort all the edges by weight
- Scan the edges by weight from lowest to highest
- If an edge introduces a cycle, drop it
- If an edge does not introduce a cycle, pick it
- Terminate when n-1 edges are picked
  (n: number of vertices)

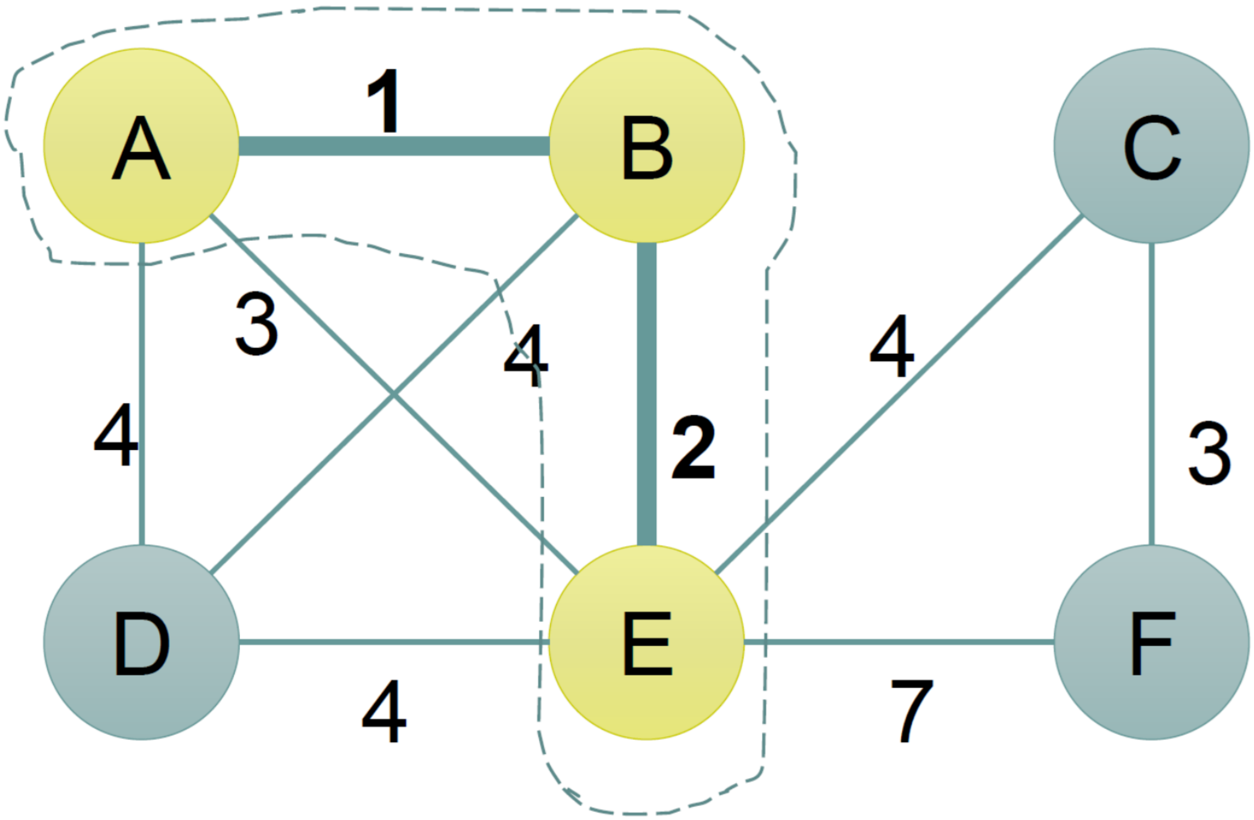
Finding MST: Kruskal's algorithm



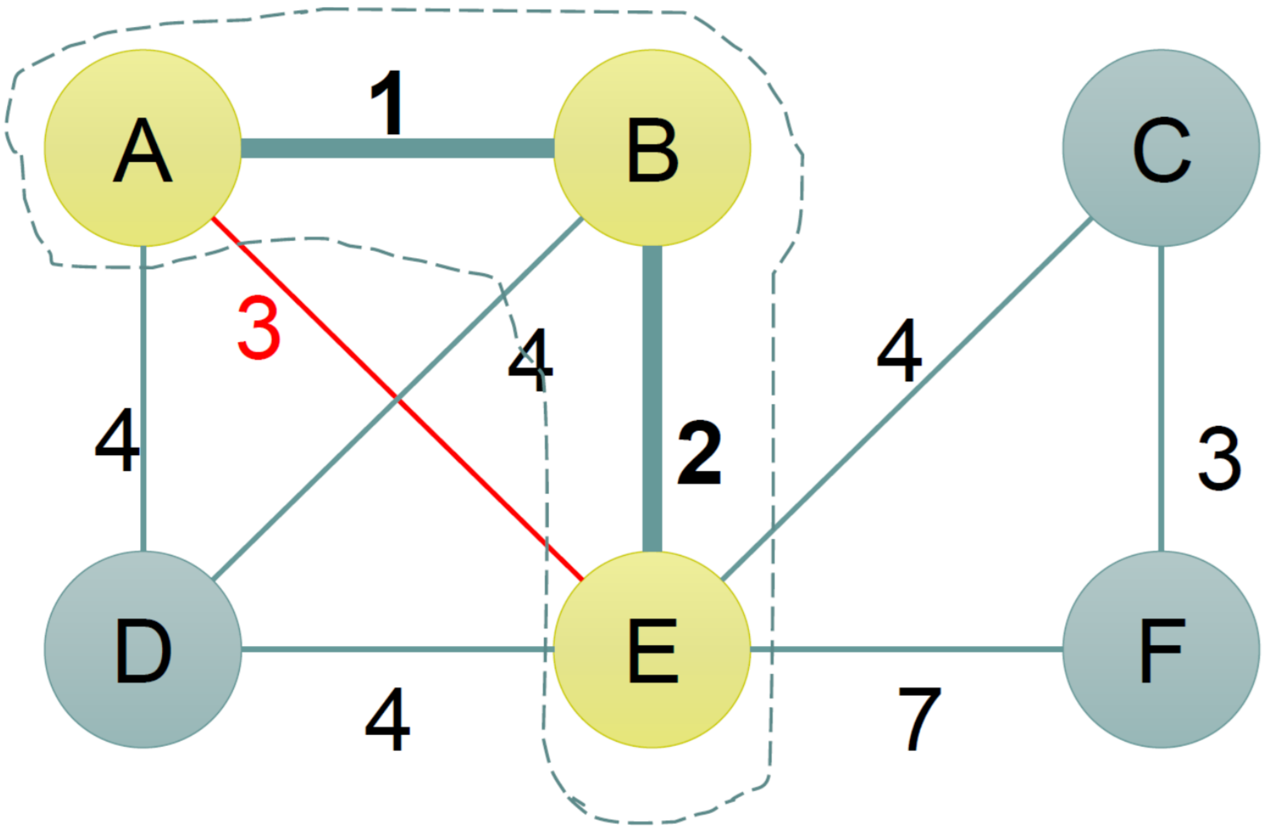
Finding MST: Kruskal's algorithm



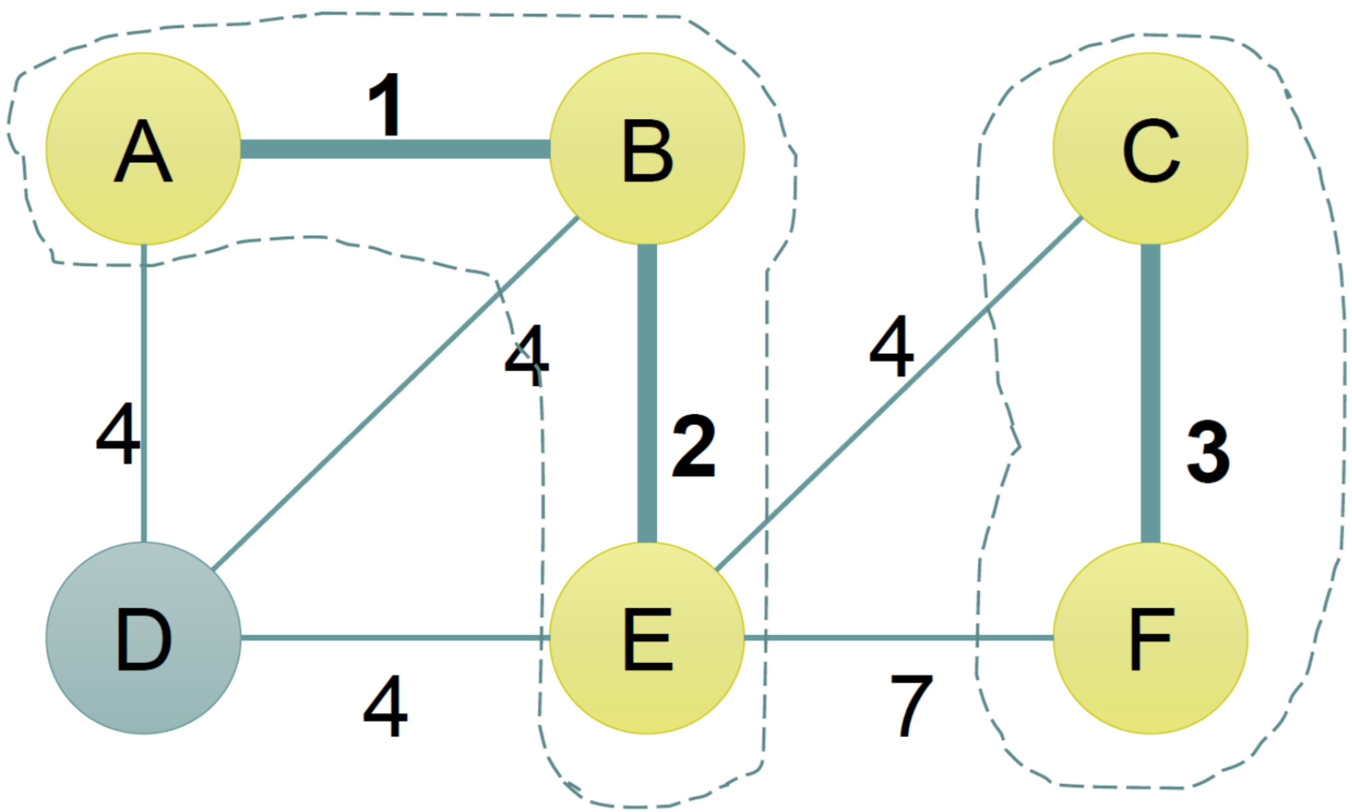
Finding MST: Kruskal's algorithm



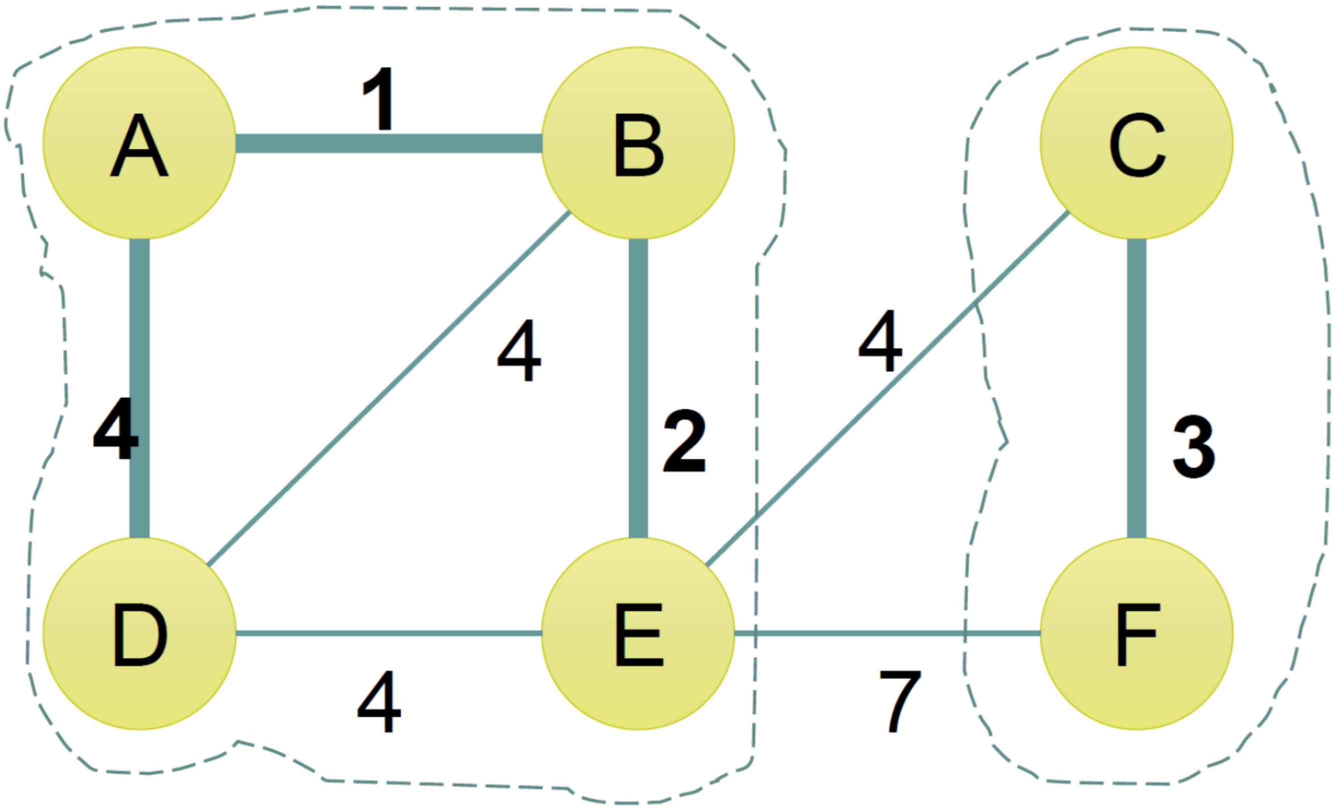
Finding MST: Kruskal's algorithm



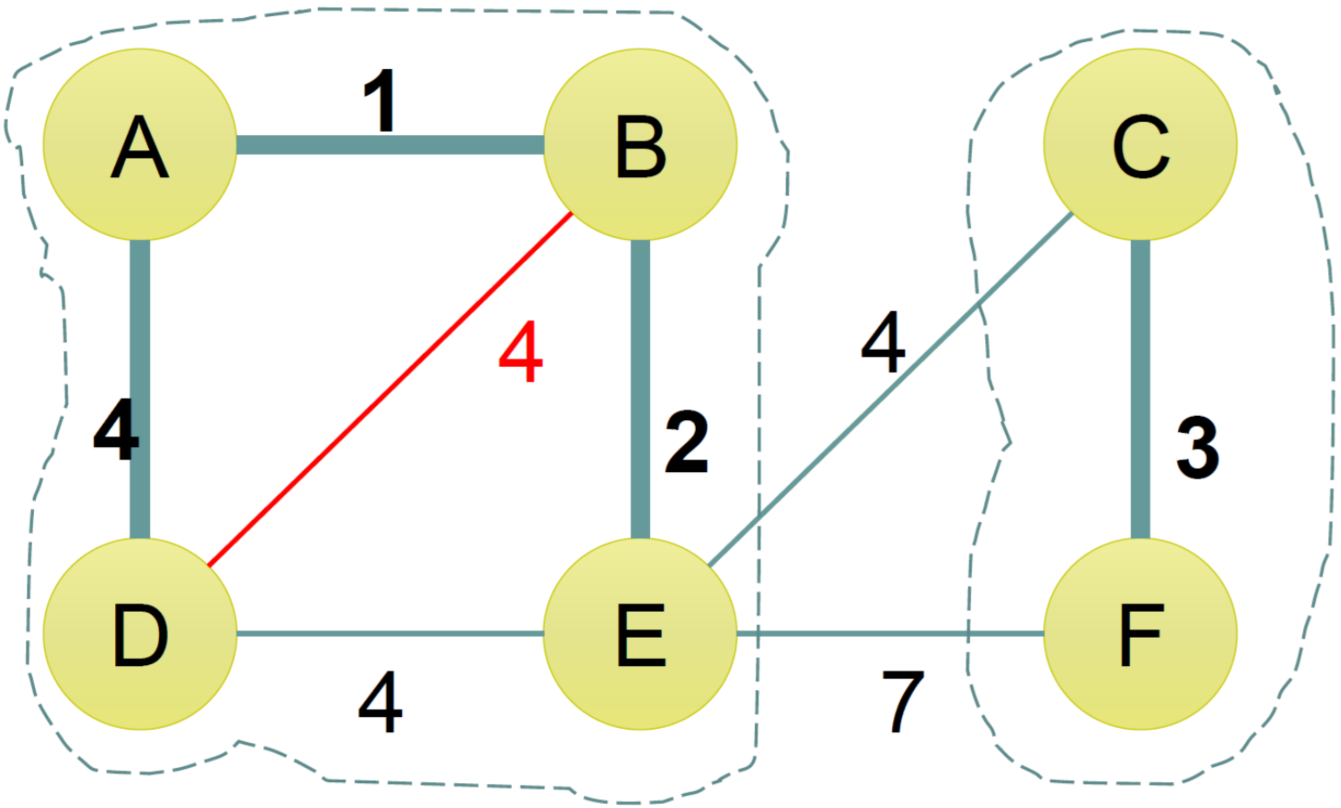
Finding MST: Kruskal's algorithm



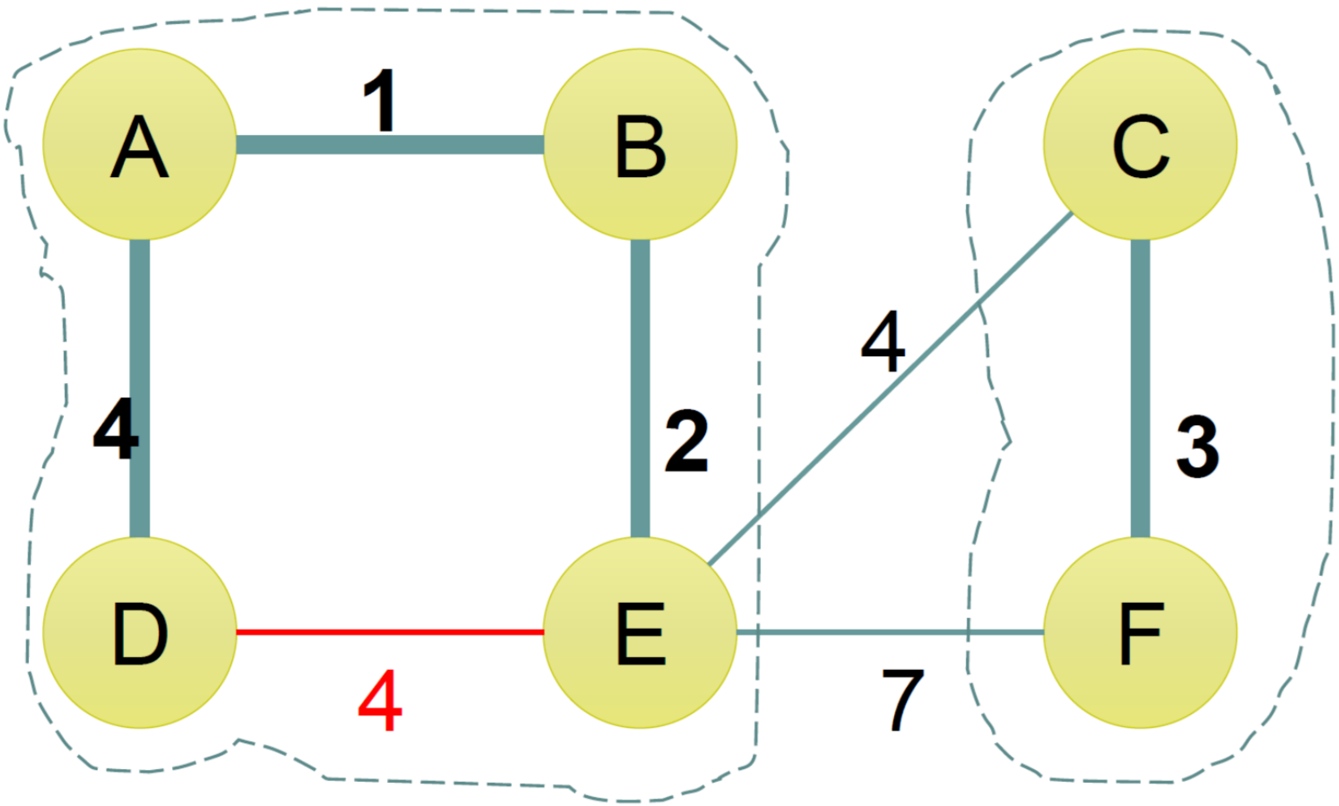
Finding MST: Kruskal's algorithm



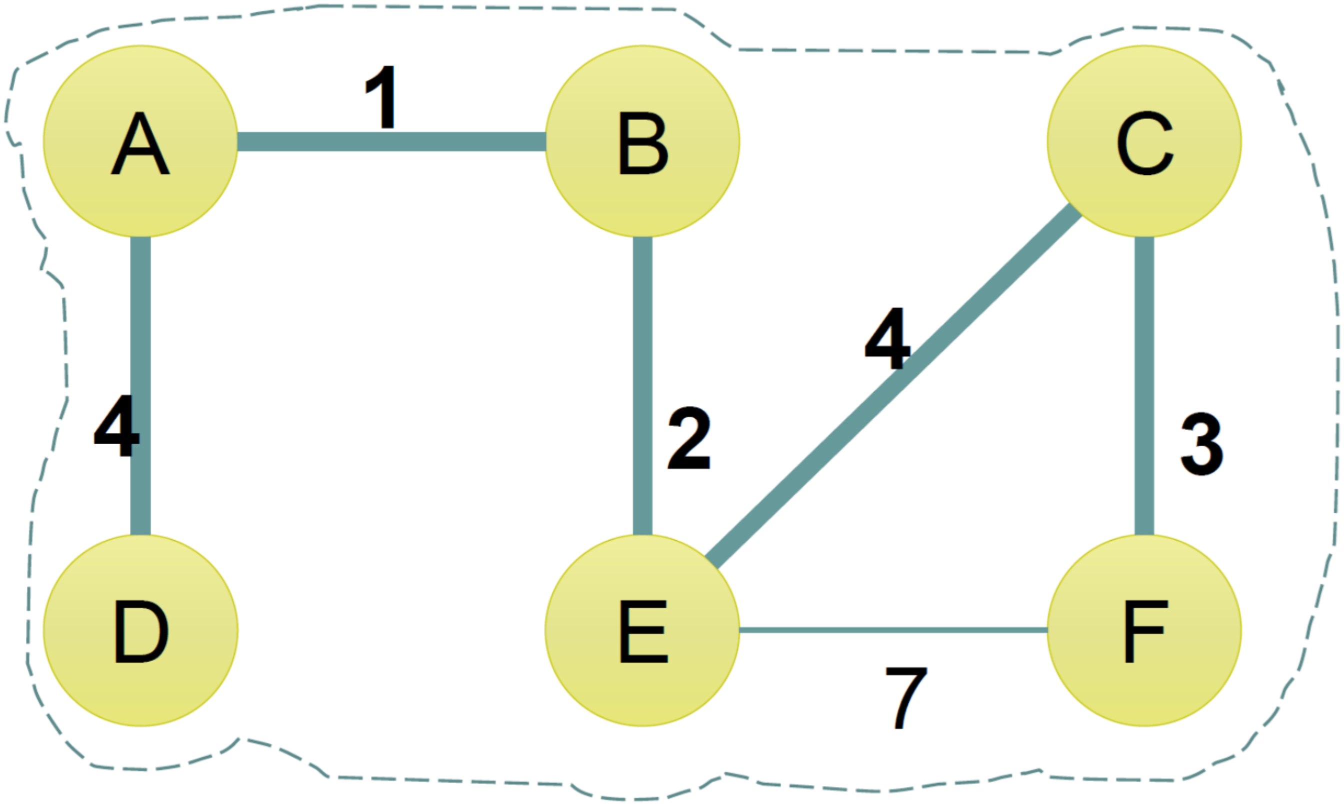
Finding MST: Kruskal's algorithm



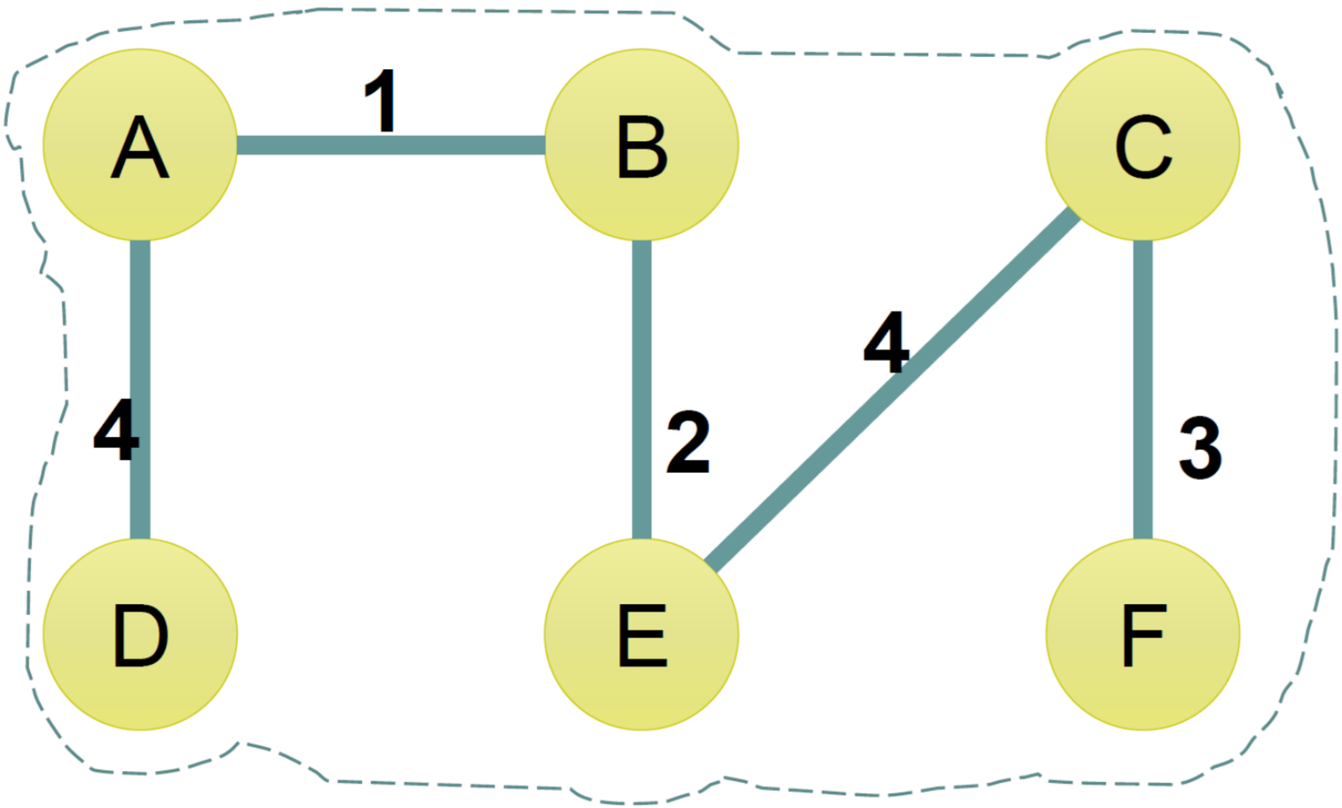
Finding MST: Kruskal's algorithm



Finding MST: Kruskal's algorithm



Finding MST: Kruskal's algorithm



Finding MST

- Kruskal's algorithm: greedy
  - Greedy choice: least weighted edge first
  - Complexity: O(E log E) – sorting edges by weight
  - Edge-cycle detection: O(1) using hashing of O(V) space
- Prim's algorithm: greedy
  - Complexity: O(E+ V log V) – using Fibonacci heap data structure

Shortest Paths in Graphs

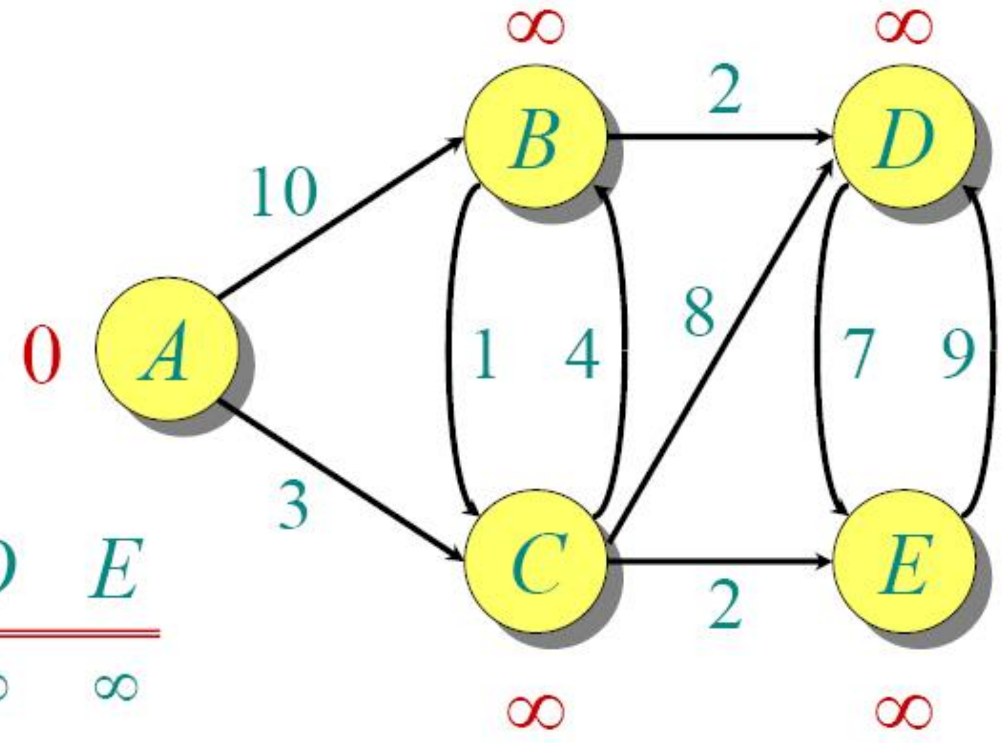
- Given graph G=(V,E), find shortest paths from a given node *source* to all nodes in V. (Single-source All Destinations)

Shortest Paths in Graphs

- Given graph G=(V,E), find shortest paths from a given node *source* to all nodes in V. (**Single-source All Destinations**)
- If negative weight cycle exist from s→t, shortest is undefined
  - Can always reduce the cost by navigating the negative cycle
- If graph with all +ve weights → Dijkstra's algorithm
- If graph with some -ve weights → Bellman-Ford's algorithm

Dijkstra's Algorithm

Initialize:

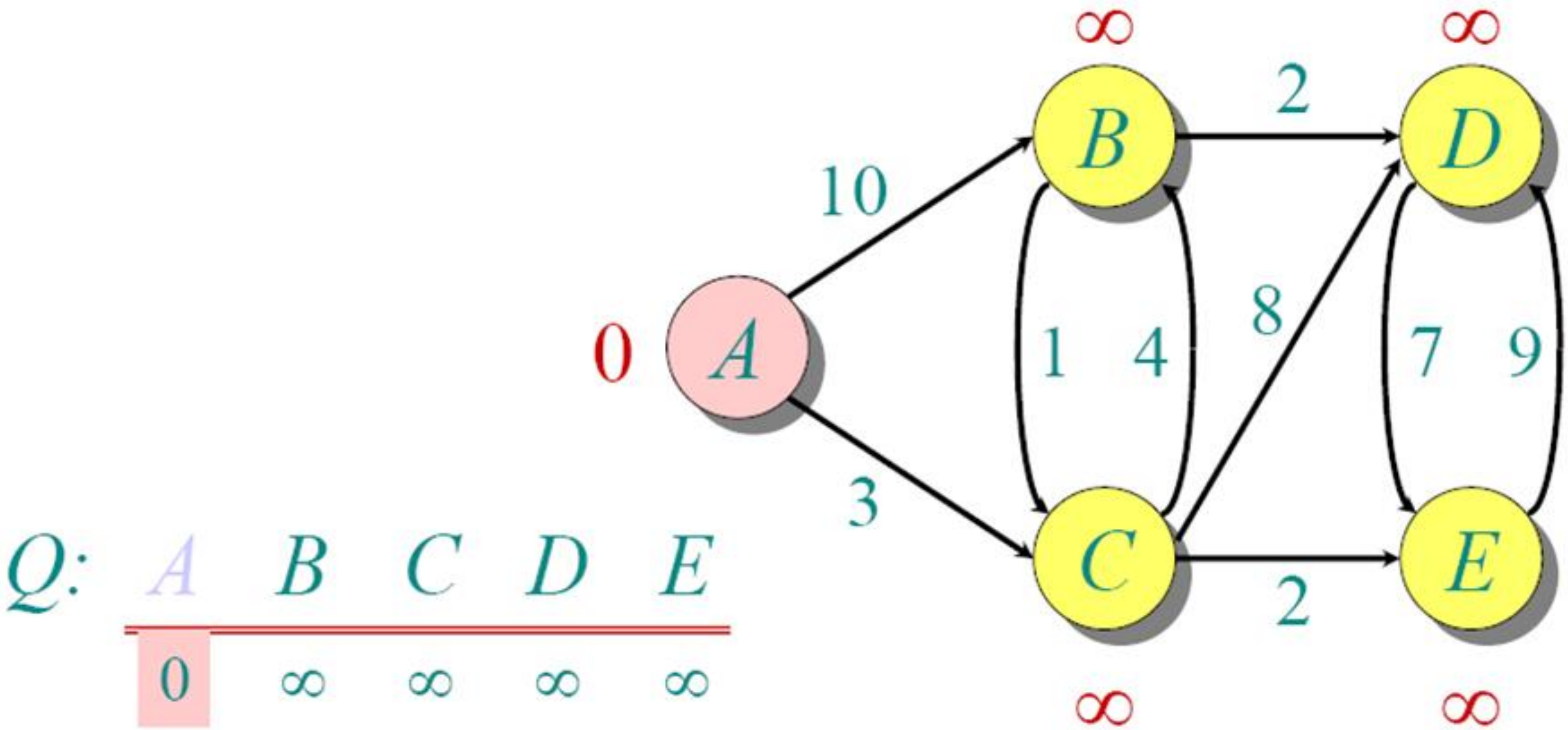


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

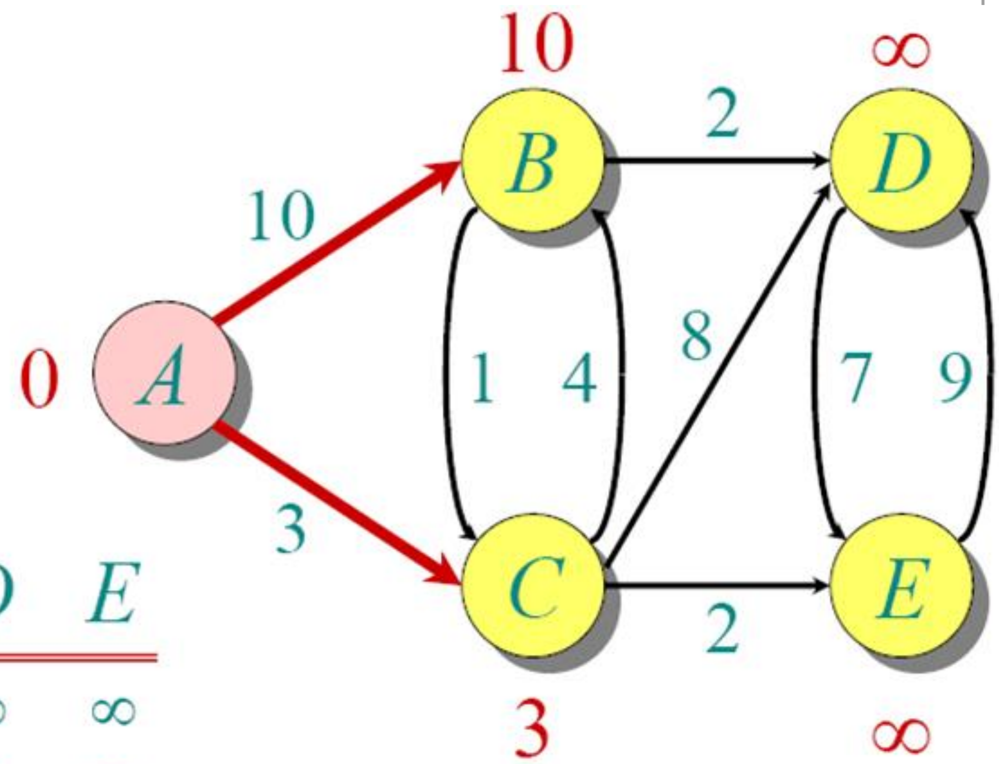
S: {}

Prev: {A,U,U,U,U}

Dijkstra’s Algorithm



Dijkstra's Algorithm

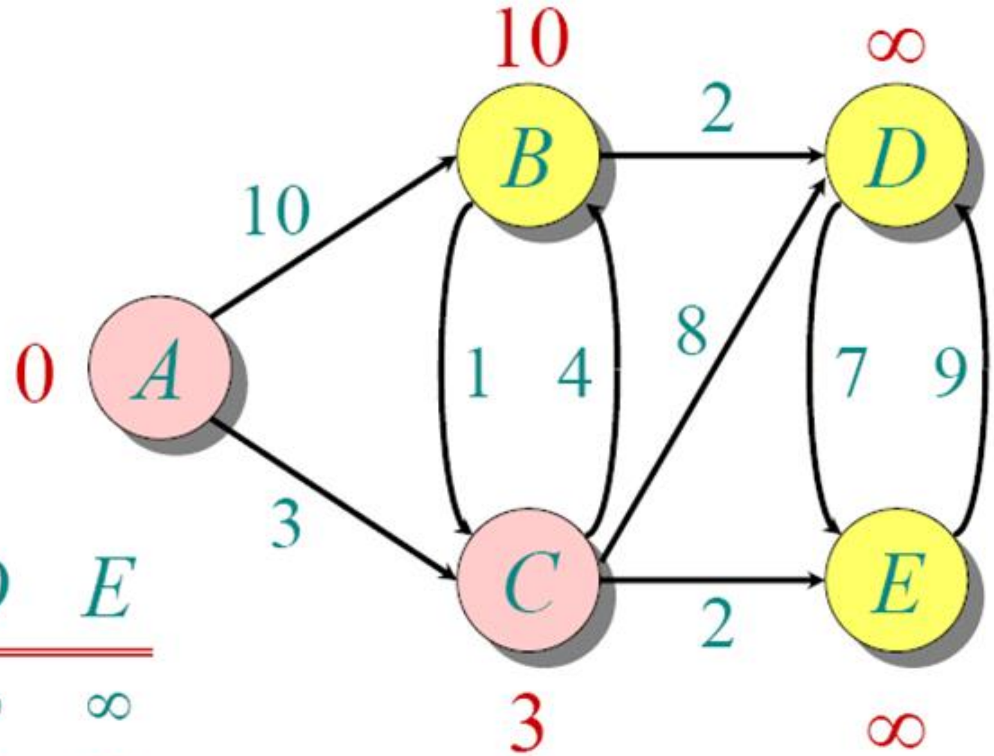


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }

Prev: {A,A,A,U,U}

Dijkstra's Algorithm

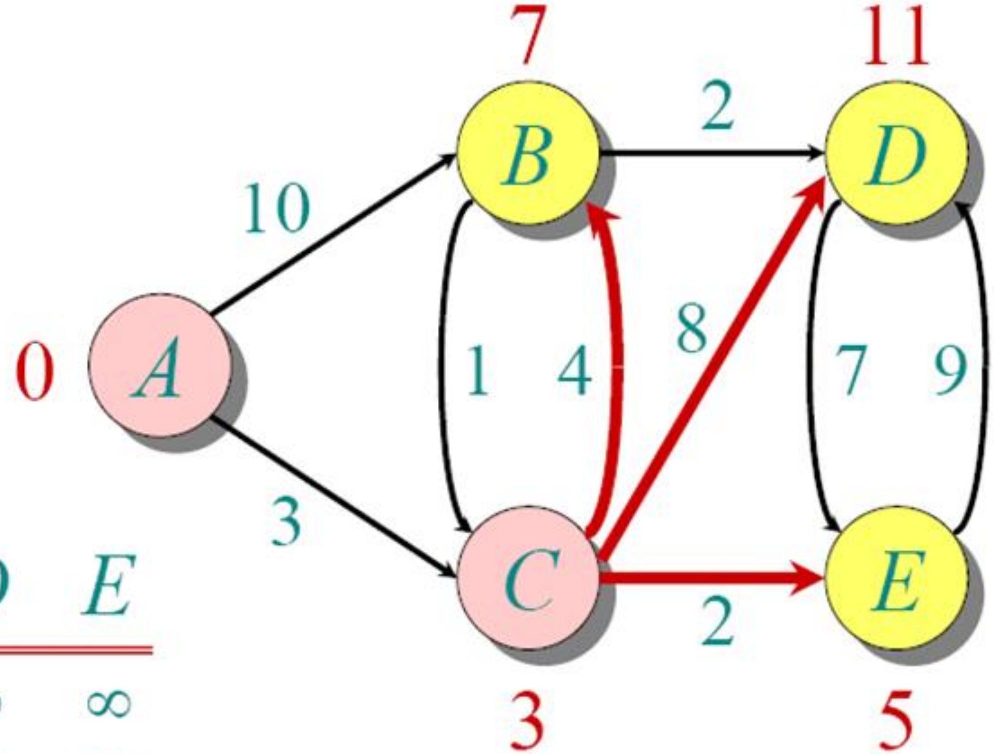


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

Prev: {A,A,A,U,U}

Dijkstra's Algorithm



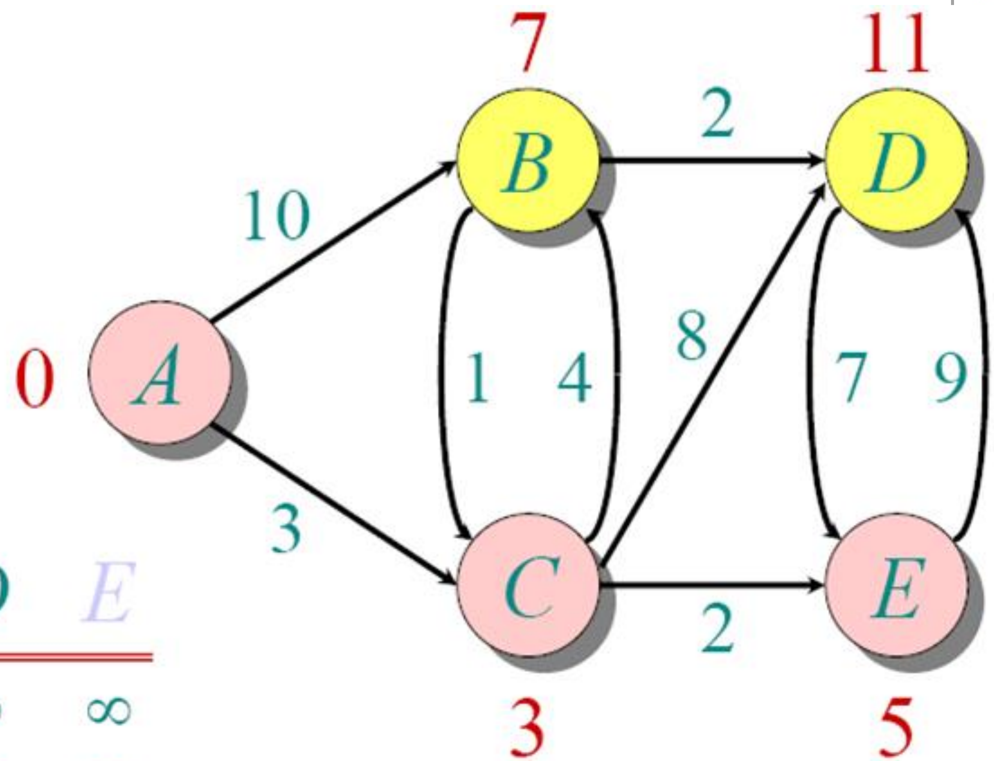
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Prev: {A,C,A,C,C}

Dijkstra's Algorithm

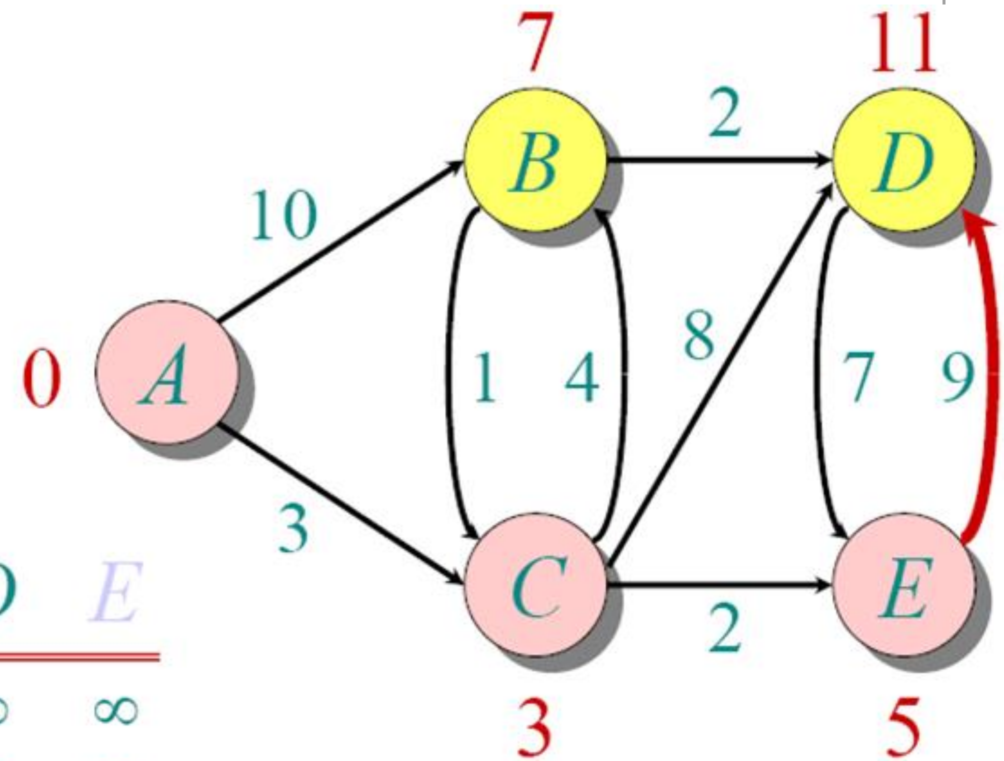
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5



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

Prev: {A,C,A,C,C}

Dijkstra's Algorithm

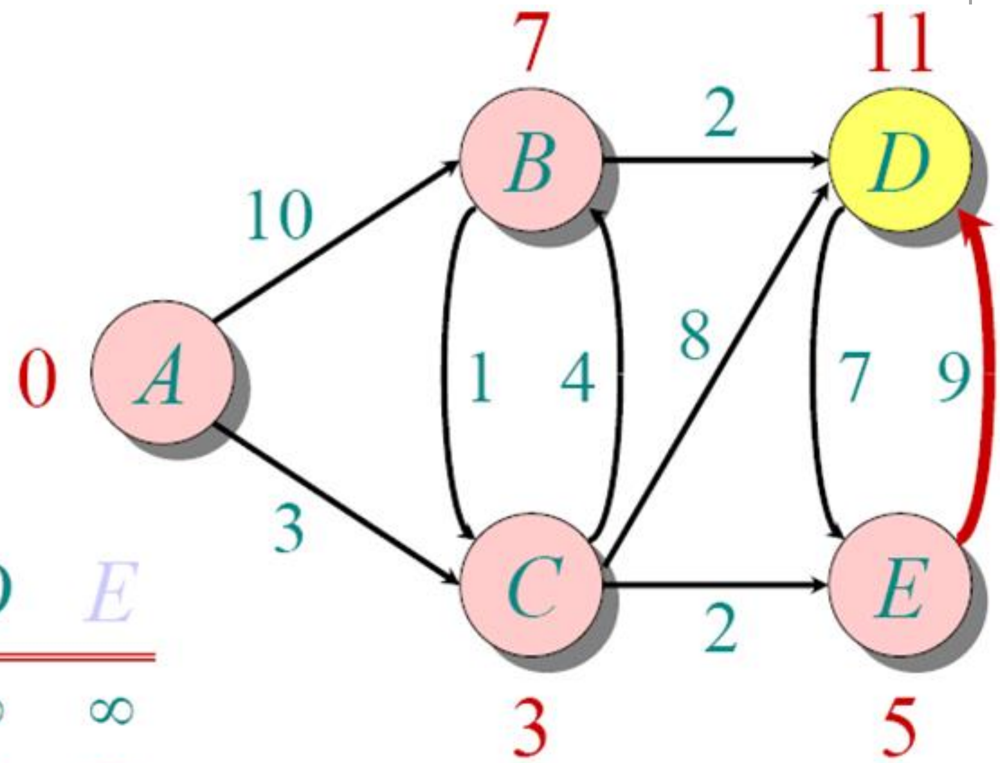


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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

Prev: {A,C,A,C,C}

Dijkstra's Algorithm

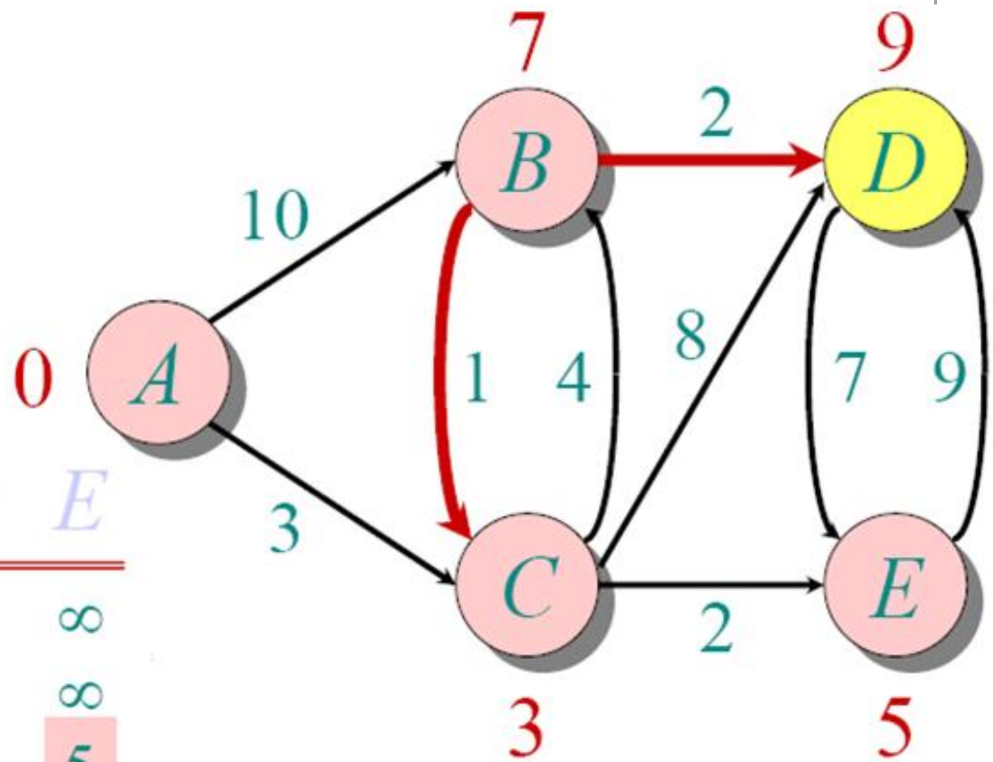


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Prev: {A,C,A,C,C}

Dijkstra's Algorithm

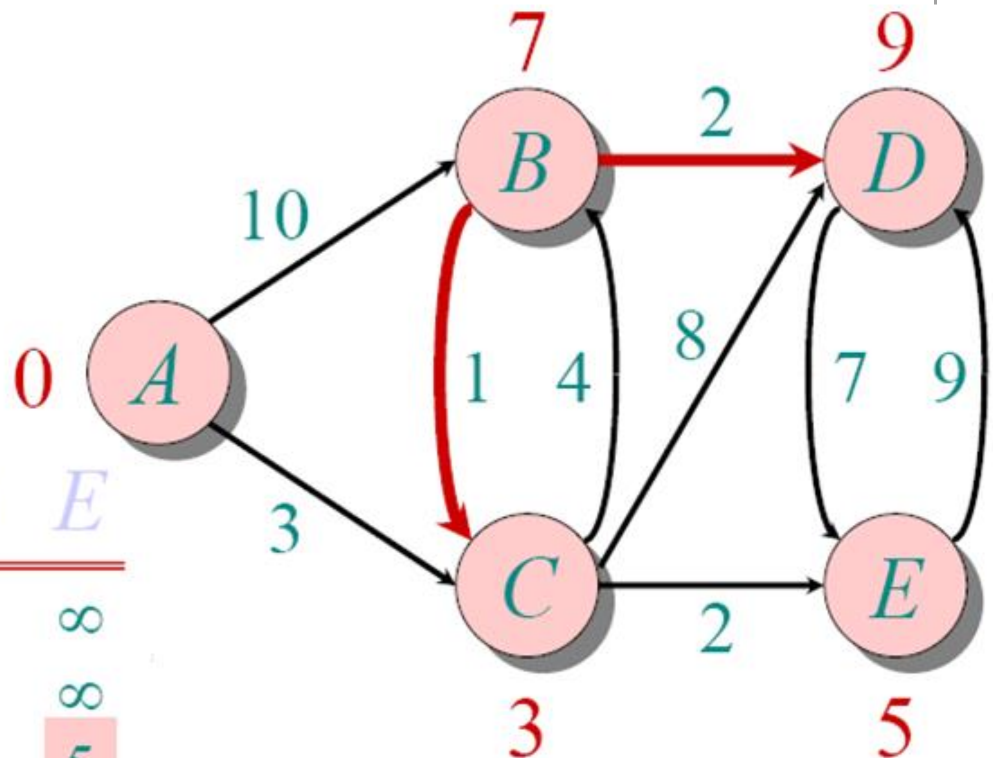


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Prev: {A,C,A,B,C}

Dijkstra's Algorithm

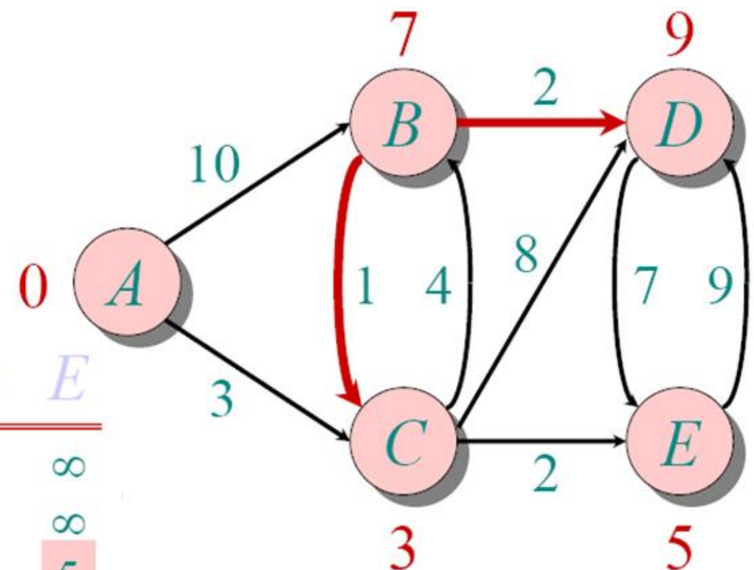


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

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

Prev: {A,C,A,B,C}

Dijkstra's Algorithm



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

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

Prev: {A,C,A,B,C}

A: A → A

B: A → C → B

C: A → C

D: A → C → B → D

E: A → C → E

Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
      create vertex set Q
      for each vertex v in Graph:    //Initialization
          Dist[v] ← INFINITY          //Unknown distance from source to v
          Prev[v] ← UNDEFINED        //Previous node in path from source to v
          add v to Q                 //All nodes initially unvisited (in Q)
      Dist[source] ← 0               // Distance from source to source = 0
      Prev[source] ← source
      while Q is not empty:
          u ← vertex in Q with min Dist[u] //Node with the least distance
                                           // will be selected first
          remove u from Q
          for each neighbor v of u in Q:        //v is still in Q.
              tmp ← Dist[u]+edge_length(u, v) //trying u as "source->u->v"
              if tmp < Dist[v]:       //A shorter path to v has been found
                  Dist[v] ← tmp
                  Prev[v] ← u
      return Dist[], S[]
```

Book Readings & Credits

- Book Readings:
  - Ch. 22, 23.2, 24.3
- Credits:
  - Figures:
    - Wikipedia
    - btechsmartclass.com
    - https://www.codingeek.com/data-structure/graph-introductions-explanations-and-applications/
  - Prof. Ahmed Eldawy notes
  - Laksman Veeravagu and Luis Barrera