CS 1332

DATA STRUCTURES AND ALGORITHMS

Programming in Java

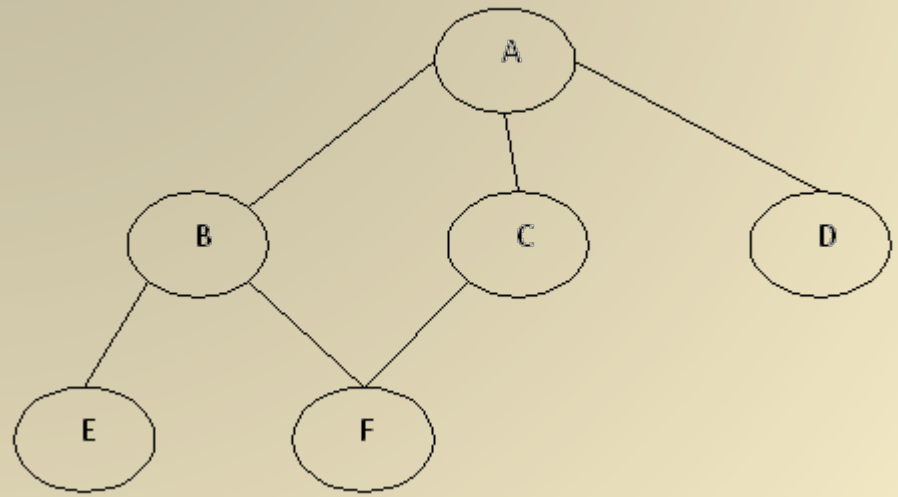
Dr. Mary Hudachek-Buswell

GRAPHS

Chapter 14

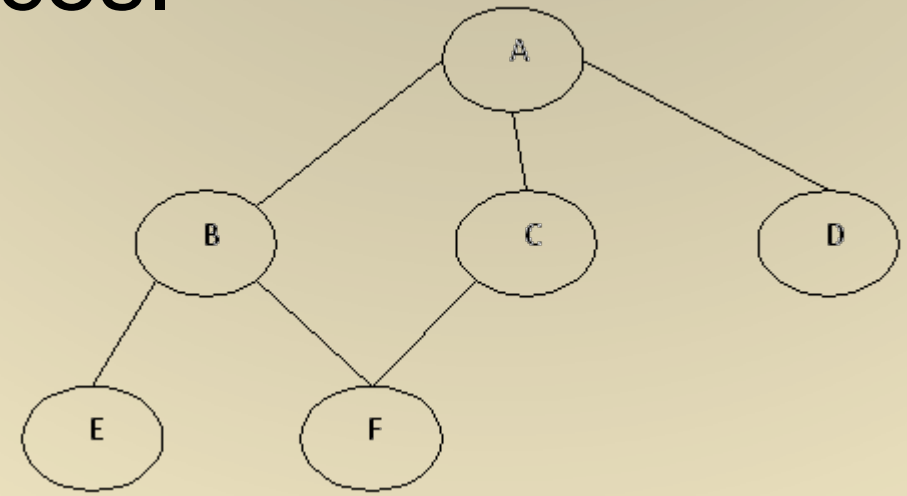
Graphs

- Graph Terminology
- Graph Modeling
- Searching
  - Breadth First Search
  - Depth First Search



Graphs

- A ***graph*** is a set of ***nodes or vertices***, *V*, and a collection of ***edges or arcs***, *E*, that each connect a pair of vertices.



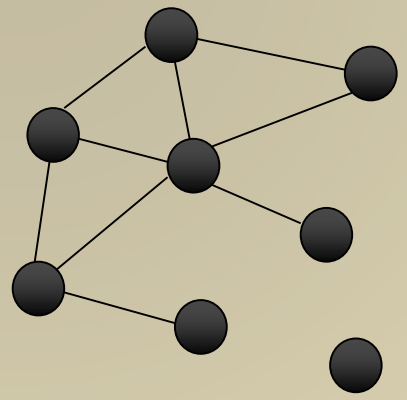
Graphs

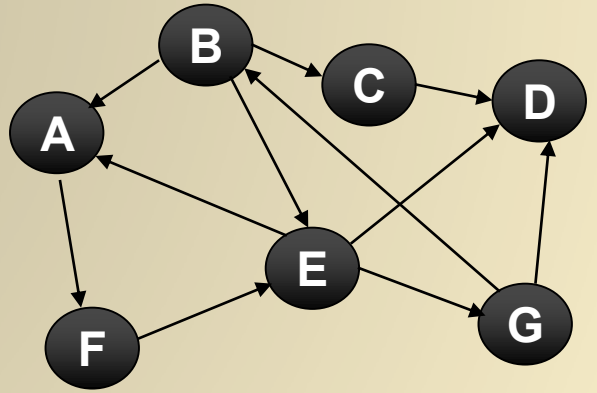
- The *order* of a graph is the number of vertices and the *size* is the edge count. The *degree* of a vertex is the number of edges incident to the vertex. (In-degree + out-degree = degree of a digraph vertex.) If all degrees are the same, then the graph is said to have that degree.

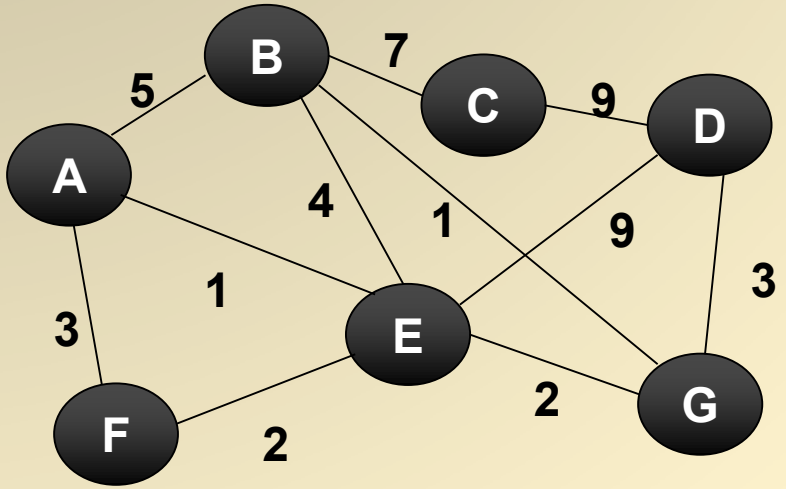
- A *path* is a set of edges connecting two nodes.

Graphs

- Graphs can have vertices not connected to the graph.
- Graphs can be *directed*, where their edges flow in one direction
- Graphs can be *undirected*, where their edges flow in both directions
- Graphs can be *weighted,* where there are weights/cost associated with each edge.

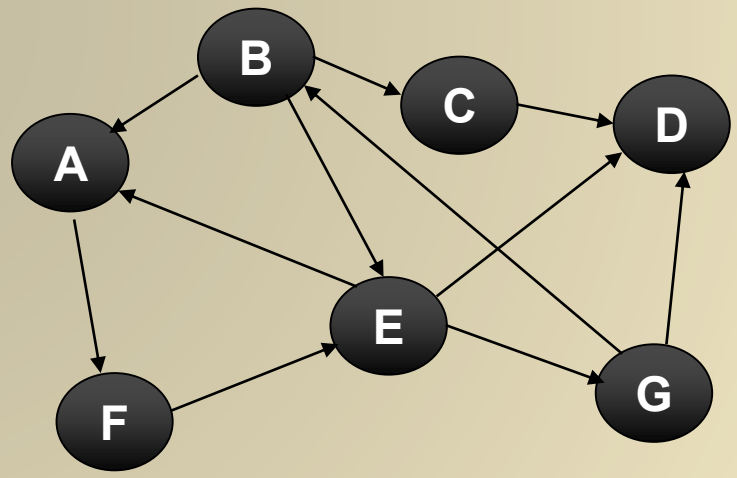


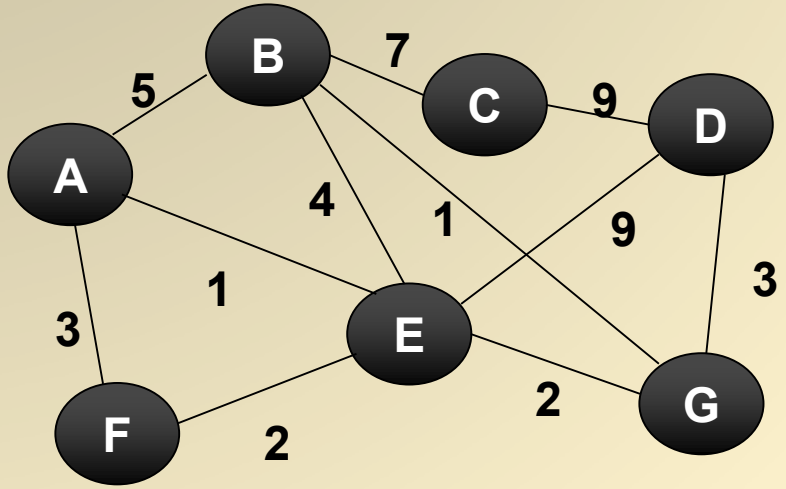




Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
- Undirected edge
  - unordered pair of vertices (u,v)
- Directed graph
  - all the edges are directed
- Undirected graph
  - all the edges are undirected





Graph Terms

- A *self-loop* is an edge that connects a vertex to itself.
- Two edges are *parallel* if they connect the same pair of vertices.
- When an edge connects two vertices, we say that the vertices are *adjacent to* one another and that the edge is *incident on* both vertices.
- The *degree* of a vertex is the number of edges incident on it.
- A *path* in a graph is a sequence of vertices connected by edges. A *simple path* is one with no repeated vertices.
- A *cycle* is a path (with at least one edge) whose first and last vertices are the same. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).
- The *length* of a path or a cycle is its number of edges.

Graph ADT

Graph Information Procedures:

- vertices(), returns iteration of all vertices
- edges(), returns iteration of all edges
- numVertices(), returns count of all vertices
- numEdges(), returns count of all edges

Vertex Information Procedures:

- endVertices(e), returns the endpoint vertices of an edge
- getEdge(u, v), returns an edge from vertex u to vertex v, if one exists; else, returns null
- numEdges(v), returns iteration of all outgoing edges to v
- outDegree(v), returns number of outgoing edges to v
- opposite(v, e), returns the other vertex u that is incident to v for edge e

Graph Access Procedures:

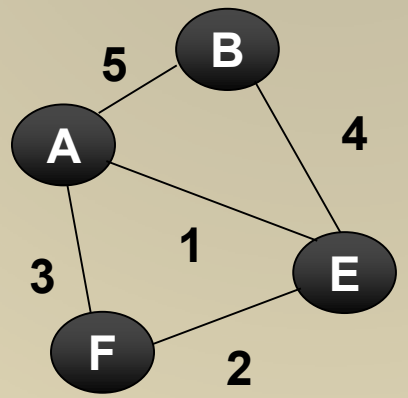
- insertVertex(d), creates and returns a new Vertex storing d
- insertEdge(u, v, e), creates and returns a new Edge e from vertex u to v
- removeVertex(v), removes vertex v and all incident edges for v from the graph
- removeEdge(e), removes edge e from the graph

Graph ADT Structures

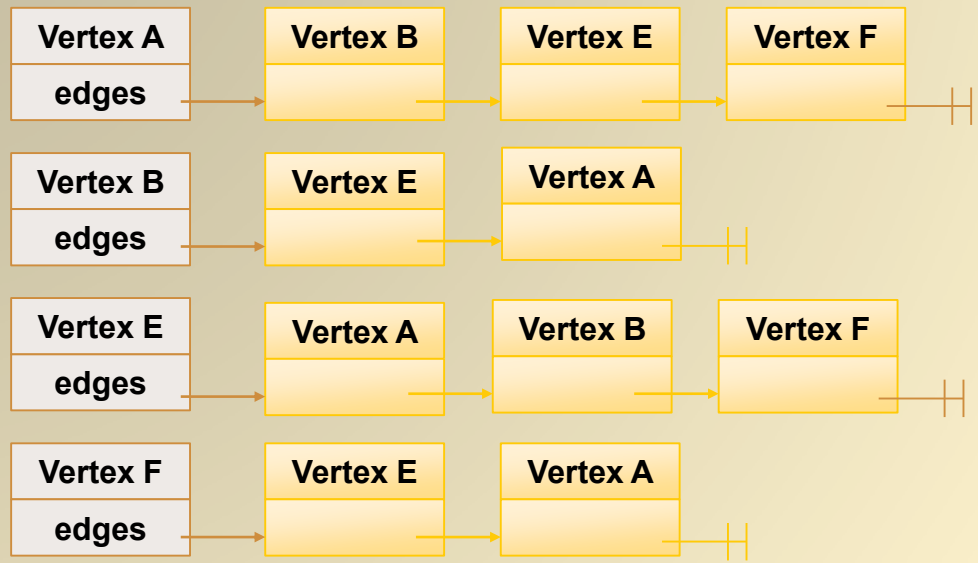
The two most common implementations of a graph are the *adjacency matrix* and the *adjacency list*. The former is a static implementation and the latter is a dynamic implementation. We also have *edge list* representation of a graph.

Adjacency Matrix

	A	B	E	F
A	-	5	1	3
B	5	-	4	
E	1	4	-	2
F	3		2	-



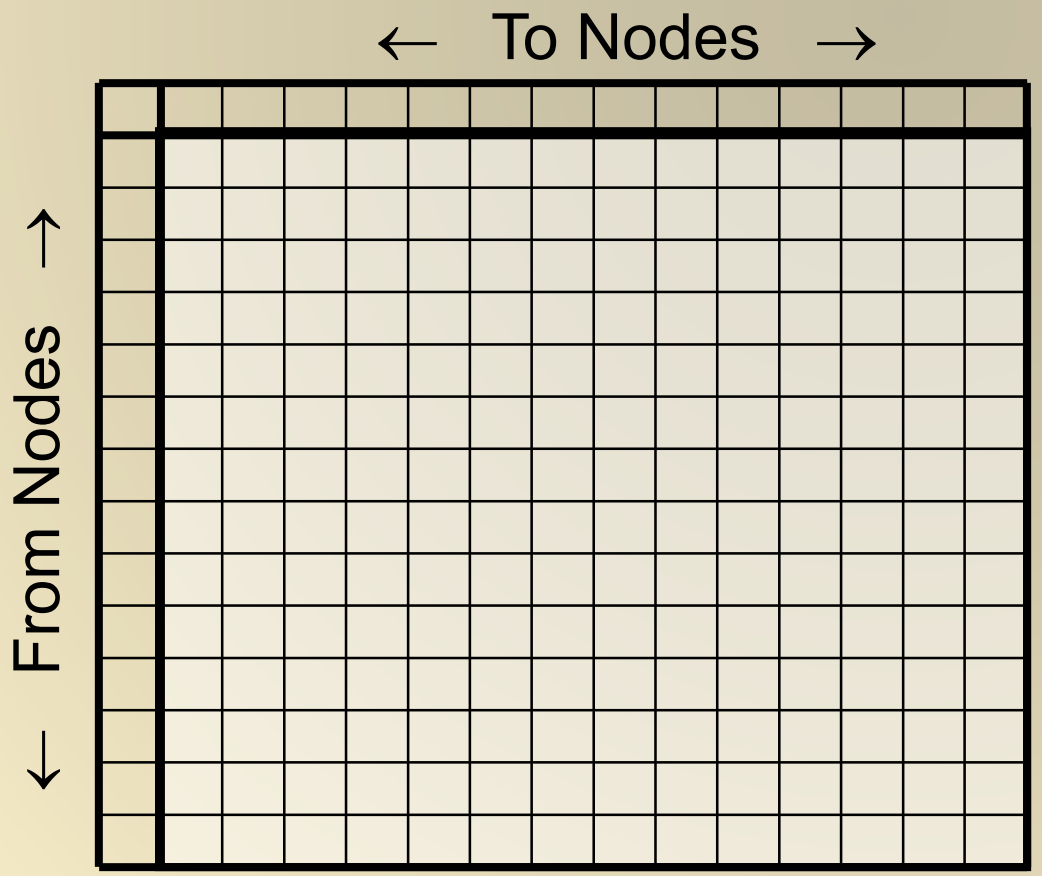
Adjacency List



The matrix is symmetrical, that is matrix[i][j] = matrix[j][i]. This is not necessarily the case for a directed graph. Waste of memory, inflexible for adding or removing vertices

The dynamic structure has definite advantages for finding adjacent nodes and allows one to show if a node is disconnected in the graph. More complex to find an edge between two vertices

Adjacency Matrix



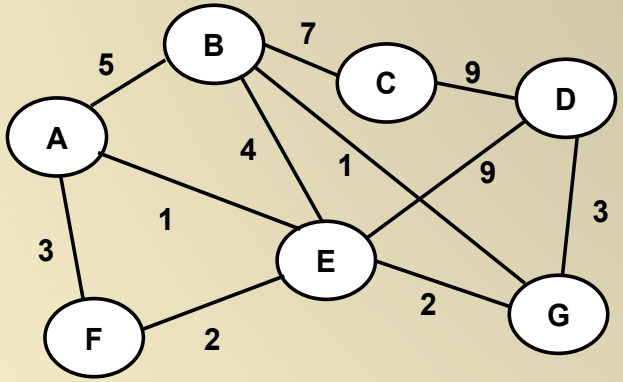
- **Initially empty**
- **Each edge adds an entry**
- **Undirected graph can**
  - **Put in 2 entries per edge**
  - **Use just upper or lower diagonal**
- **Directed graph uses entire matrix**
- **Unweighted graph inserts ‘1’**
- **Weighted graph inserts the weight**

- **Size is $O(N^2)$**
- **Memory is usually sparsely utilized**

Adjacency Matrix

Undirected Graph

Weighted Edges



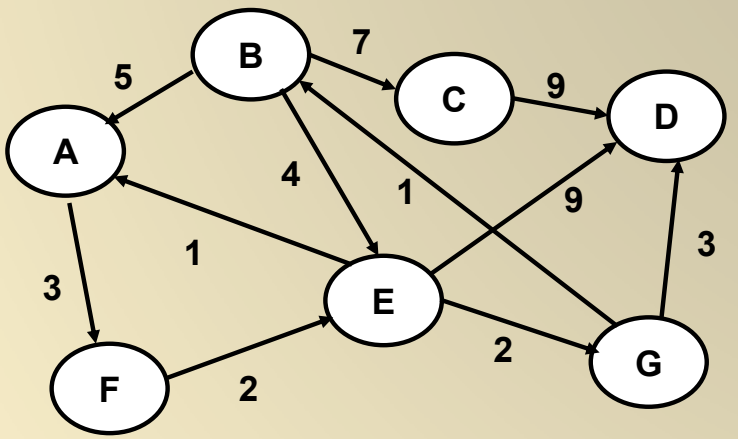
Edge weights represent cost.

	A	B	C	D	E	F	G
A	-	5	.	.	1	3	.
B	5	-	7	.	4	.	1
C	.	7	-	9	.	.	.
D	.	.	9	-	9	.	3
E	1	4	.	9	-	2	2
F	3	.	.	.	2	-	.
G	.	1	.	3	2	.	-

	A	B	C	D	E	F	G
A	-	5	.	.	1	3	.
B		-	7	.	4	.	1
C			-	9	.	.	.
D				-	9	.	3
E					-	2	2
F						-	.
G							-

Adjacency Matrix

Weighted Directed Graphs



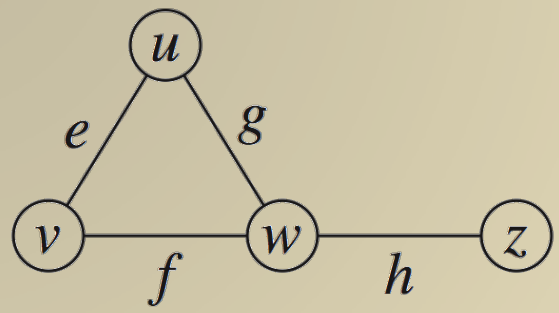
Directed edges only allow movement in one direction.

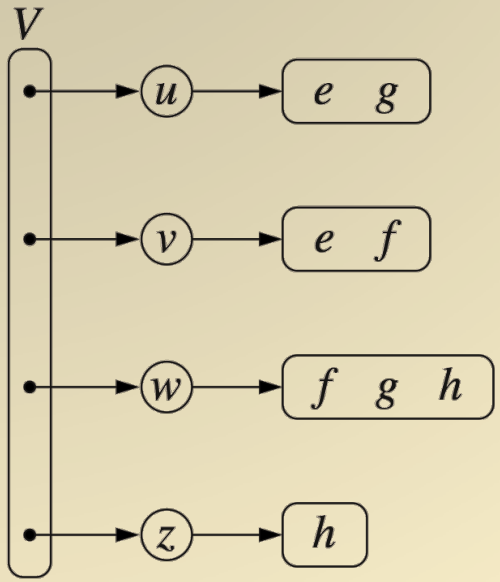
Directed Graph

	A	B	C	D	E	F	G
A	-	.	.	.	.	3	.
B	5	-	7	.	4	.	.
C	.	.	-	9	.	.	.
D	.	.	.	-	.	.	.
E	1	.	.	9	-	.	2
F	3	.	.	.	2	-	.
G	.	1	.	3	.	.	-

Adjacency List Structure

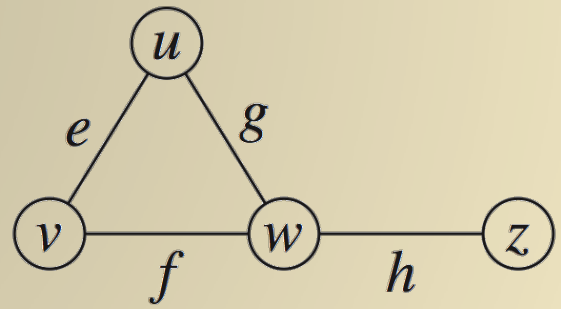
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

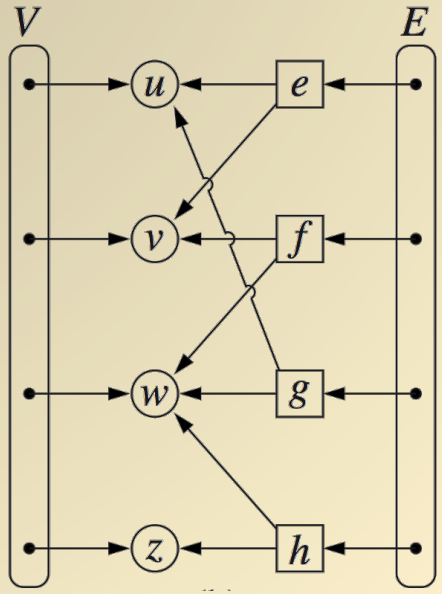




Edge List Structure (3rd)

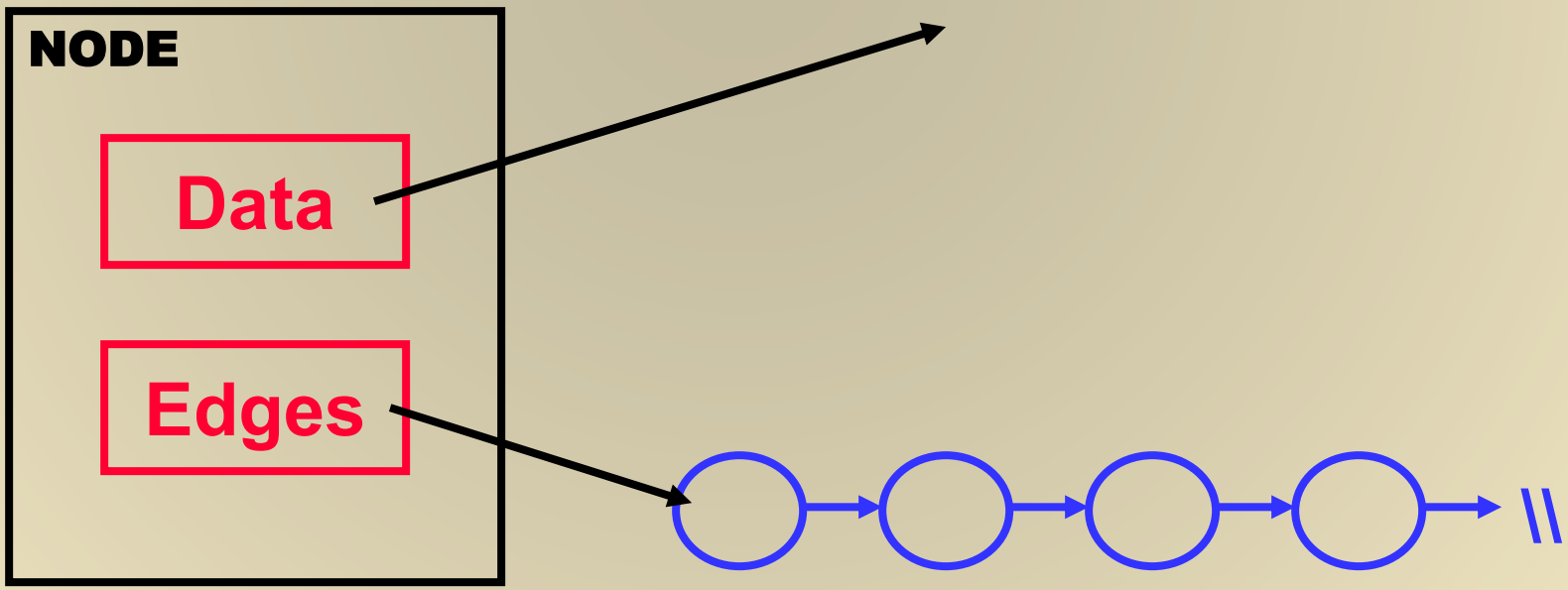
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects





Implementation with Linked Lists

[ArrayList might be more appropriate]



Edges

- **With references to other nodes**
- **Possibly with weights**

Structure Performance

v vertices, e edges	Adjacency List	Adjacency Matrix	Edge List
Space	$v + e$	v^2	$v + e$
incidentEdges(u)	$\deg(u)$	v	e
areAdjacent (u, w)	$\min(\deg(u),\deg(w))$	1	e
insertVertex($data$)	1	v^2	1
insertEdge(u, w, f)	1	1	1
removeVertex(u)	$\deg(v)$	v^2	e
removeEdge(f)	1	1	1

Search Algorithms

- Depth-First Search, DFS -> uses Stack
- Breadth-First Search, BFS -> uses Queue

Shortest Path Algorithm

Dijkstra's Algorithm - Shortest path in a graph

Given two vertices, p and q determine the path with lowest length

Depth-First Search DFS

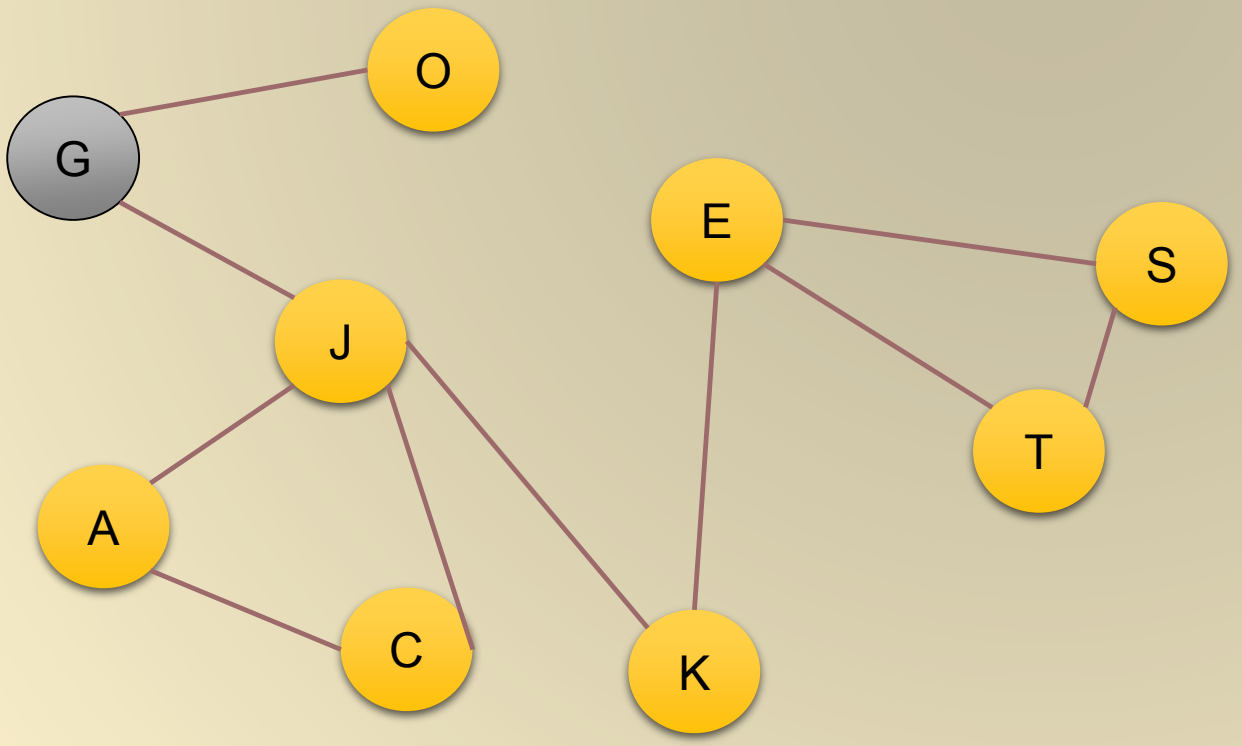
Problem Find a natural way to systematically visit every vertex and every edge of a graph :

- Start from one vertex
- Move forward all along one path (do not pass through a vertex already visited)
- When stuck, turn back until you can step forward to an unvisited vertex
- DFS finds some path from source vertex v to target vertex u.

DFS Pseudo Code

```
dfs(v)
  visit(v)
  for each neighbor w of v
    if w is unvisited
      dfs(w)
      add edge vw to tree T
    end
  end
end
```

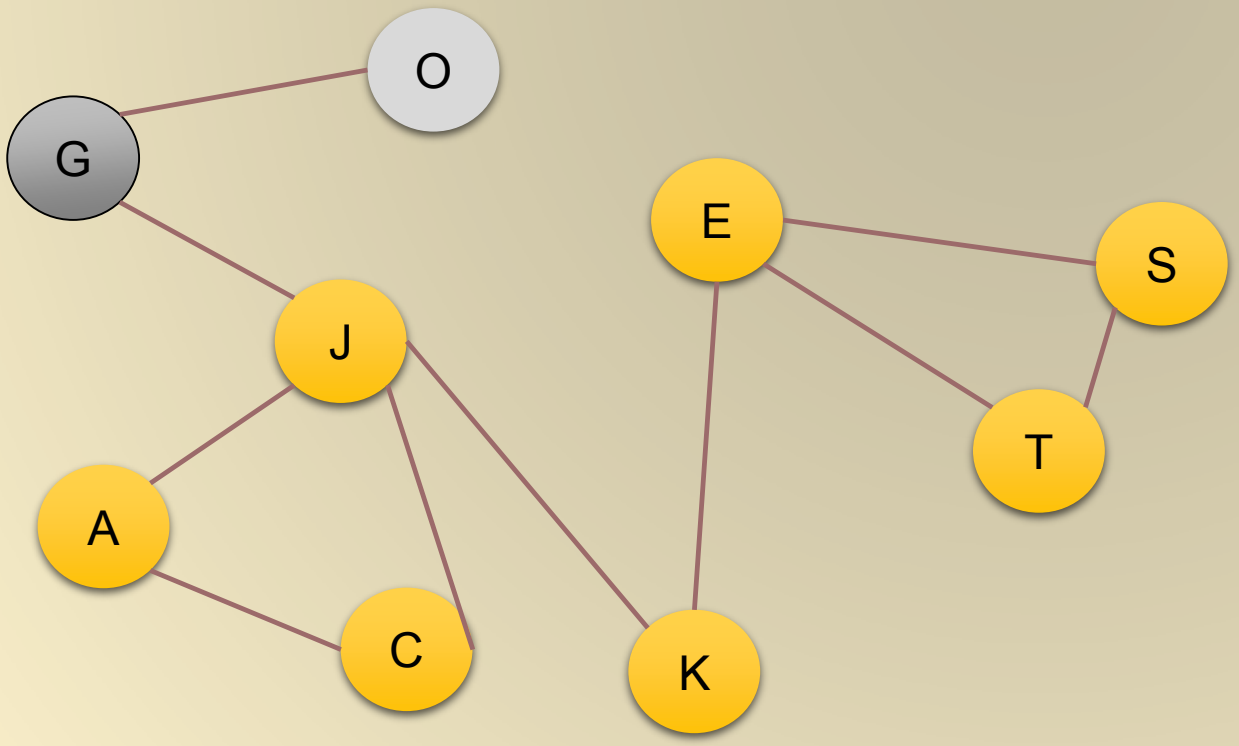
DFS Example
Traverse Graph G



stack
G

Output: G

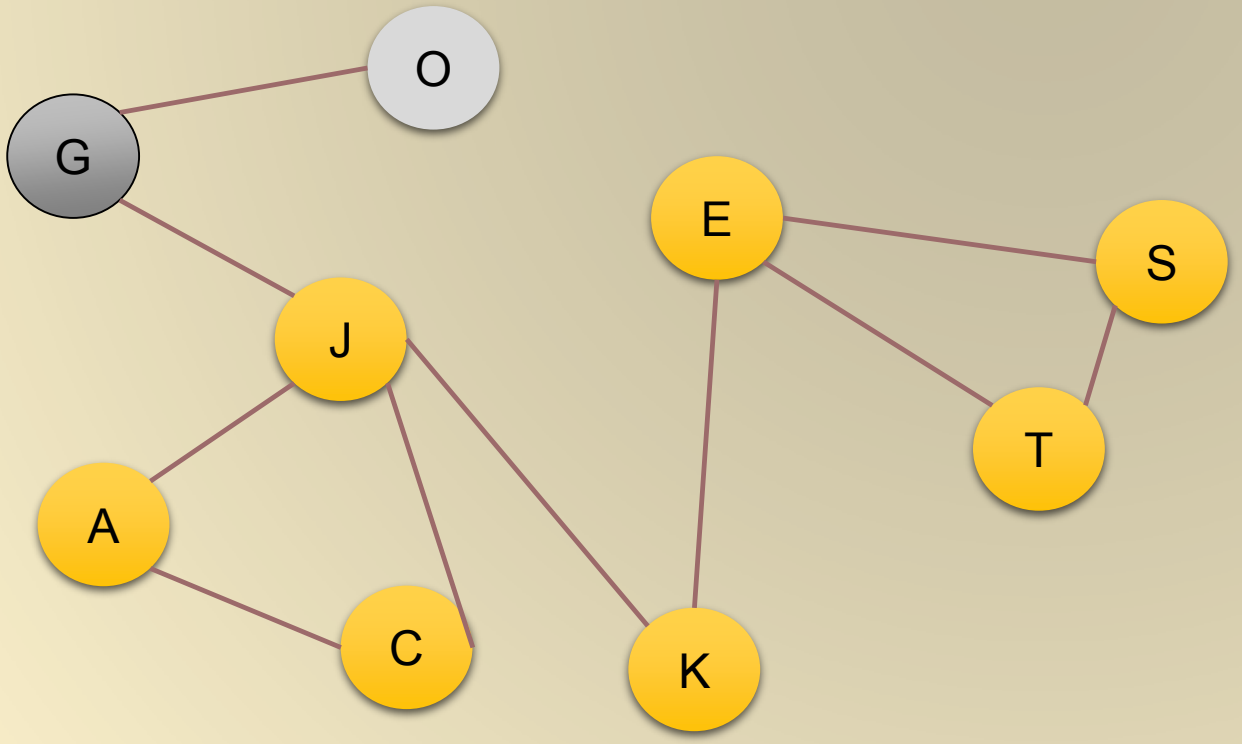
DFS Example
Traverse Graph G



stack
O
G

Output: G O

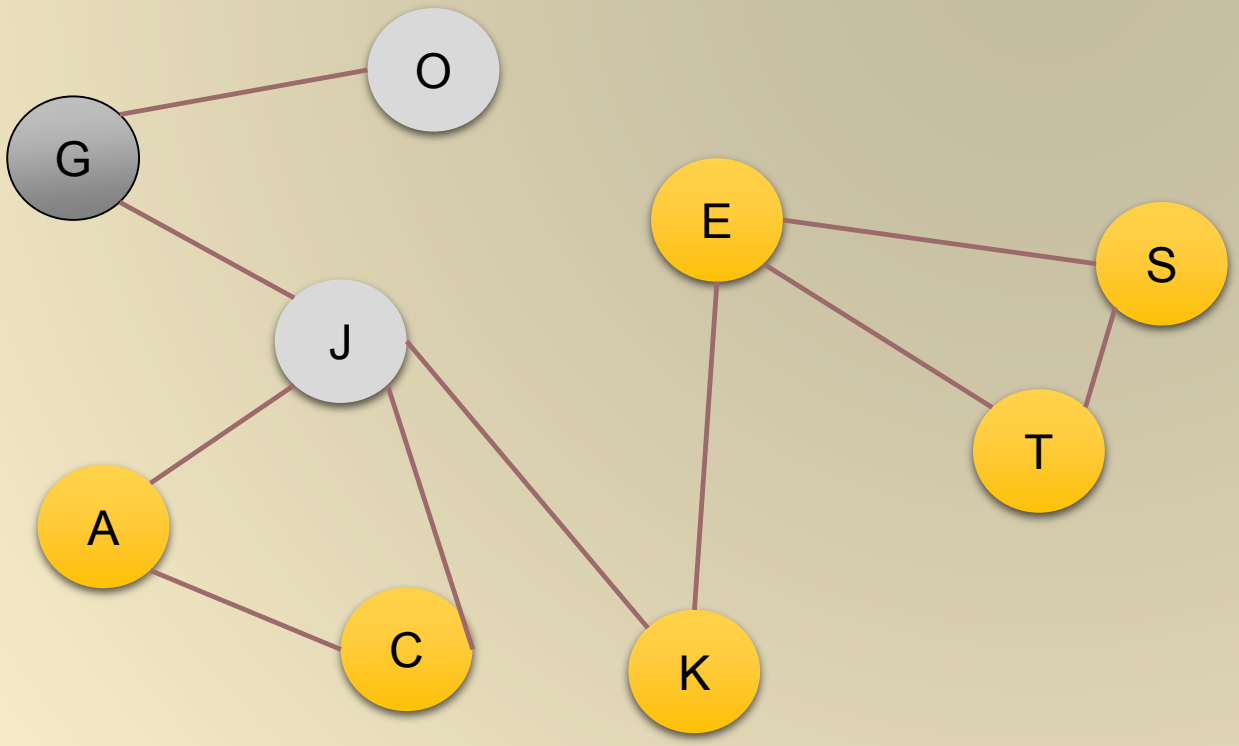
DFS Example
Traverse Graph G



stack
G

Output: G O

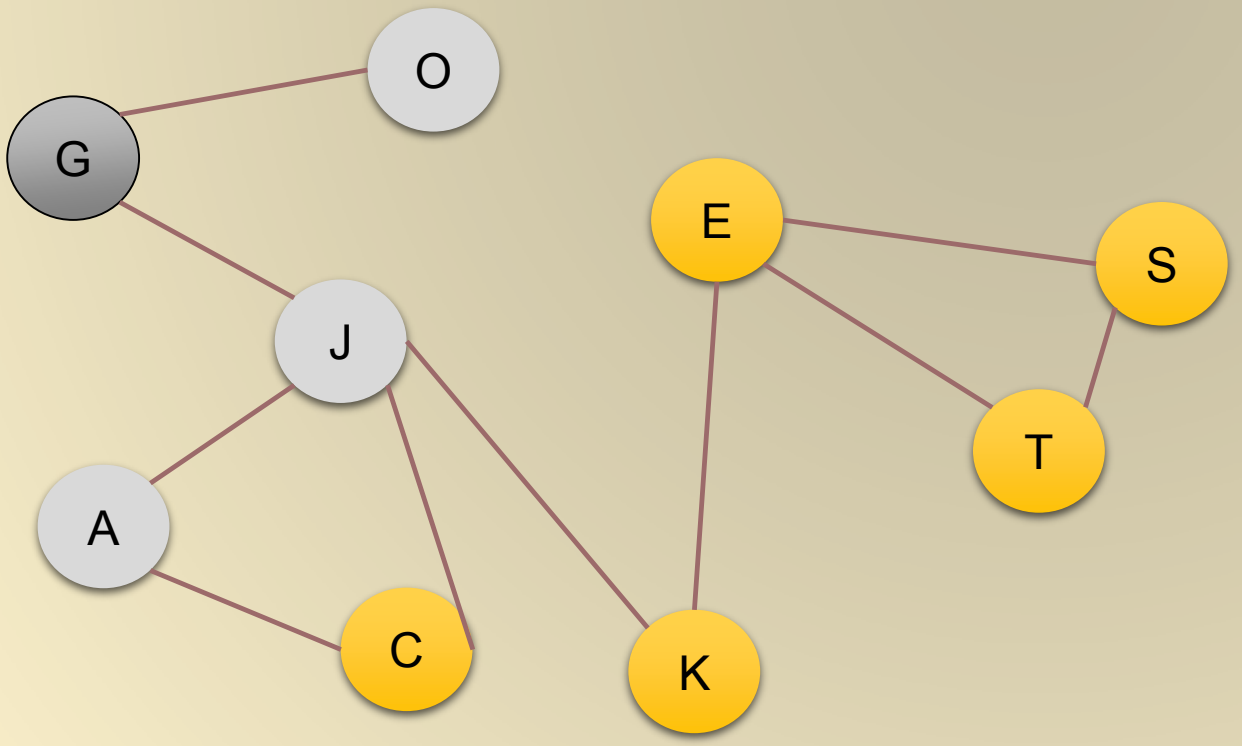
DFS Example
Traverse Graph G



stack
J
G

Output: G O J

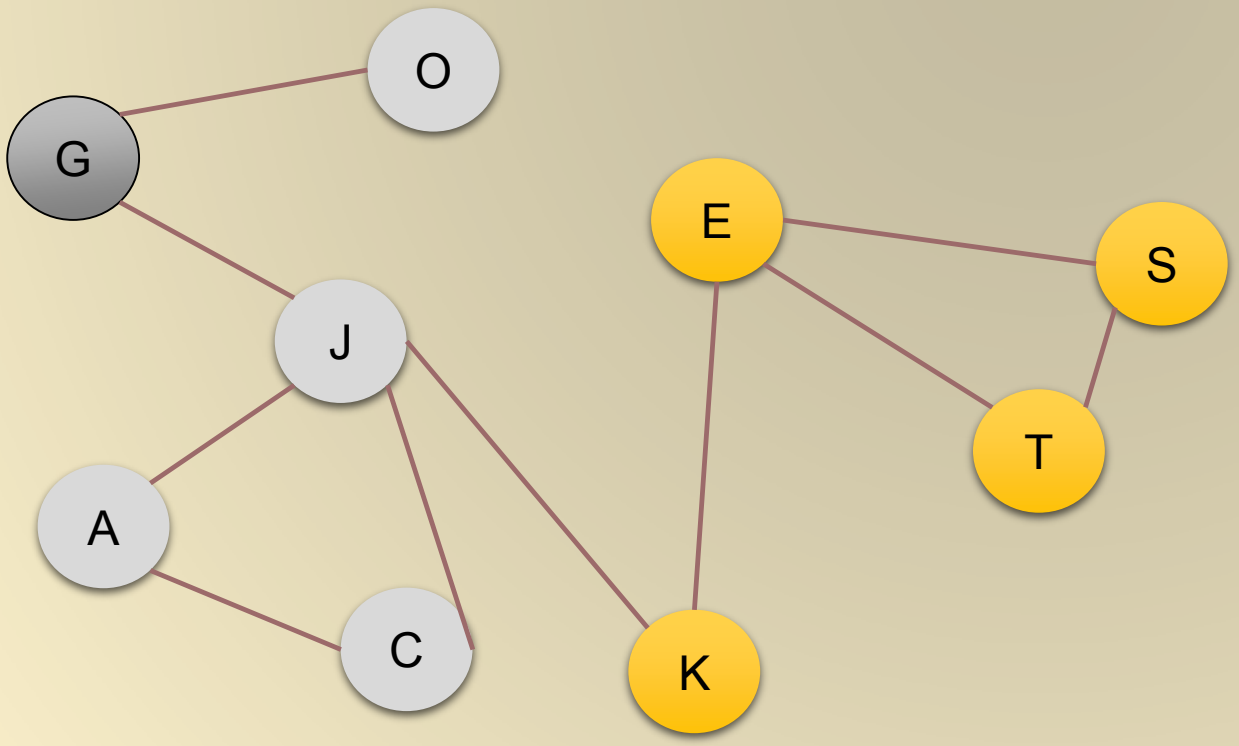
DFS Example
Traverse Graph G



stack
A
J
G

Output: G O J A

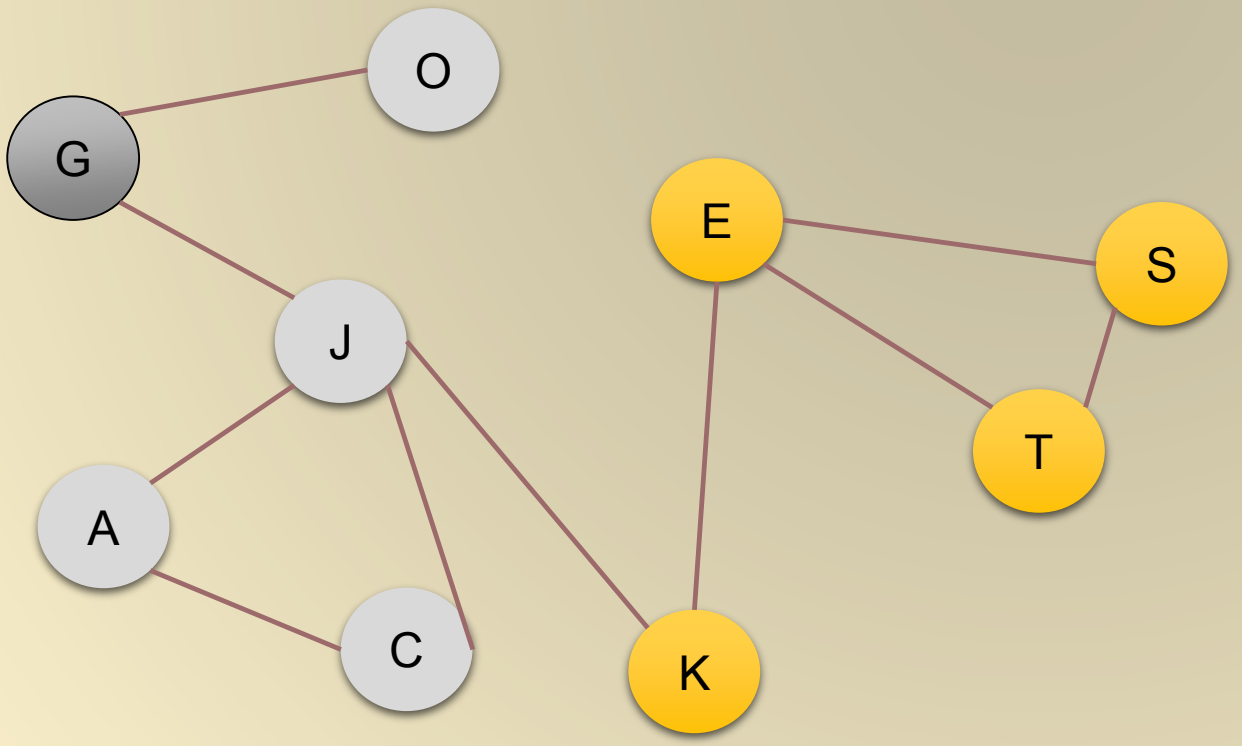
DFS Example
Traverse Graph G



stack
C
A
J
G

Output: G O J A C

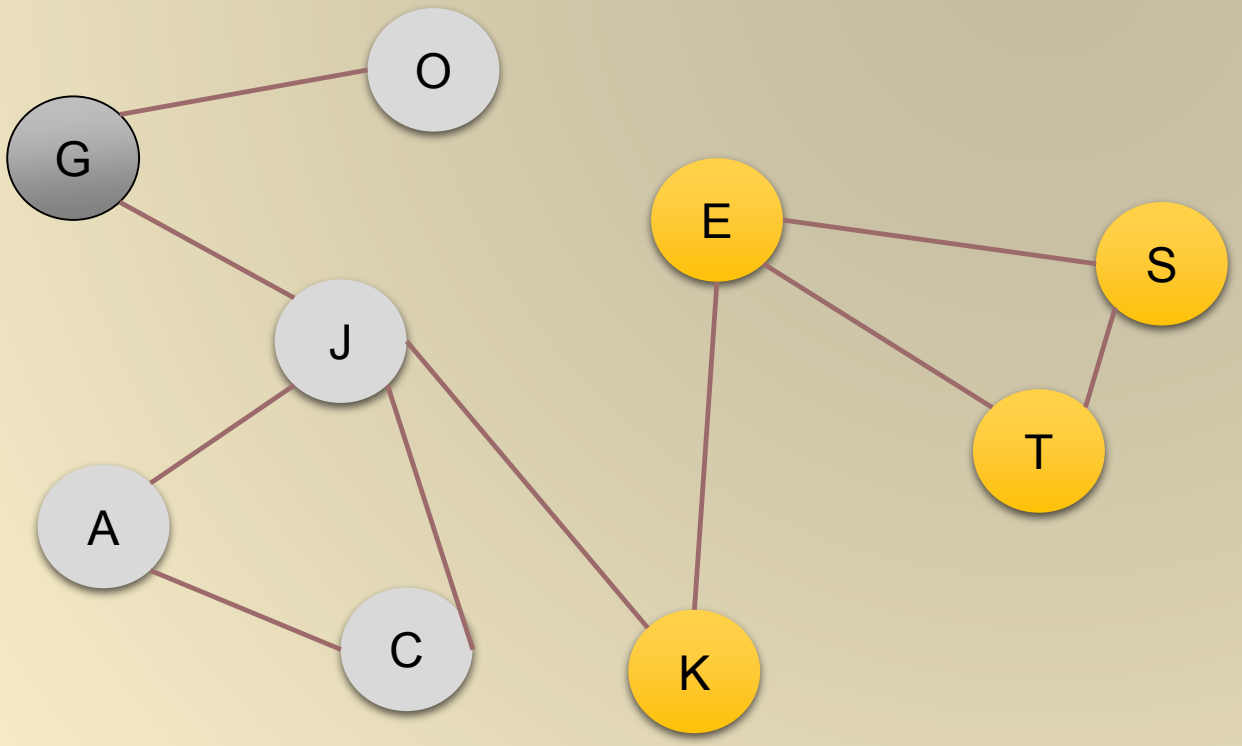
DFS Example
Traverse Graph G



stack
A
J
G

Output: G O J A C

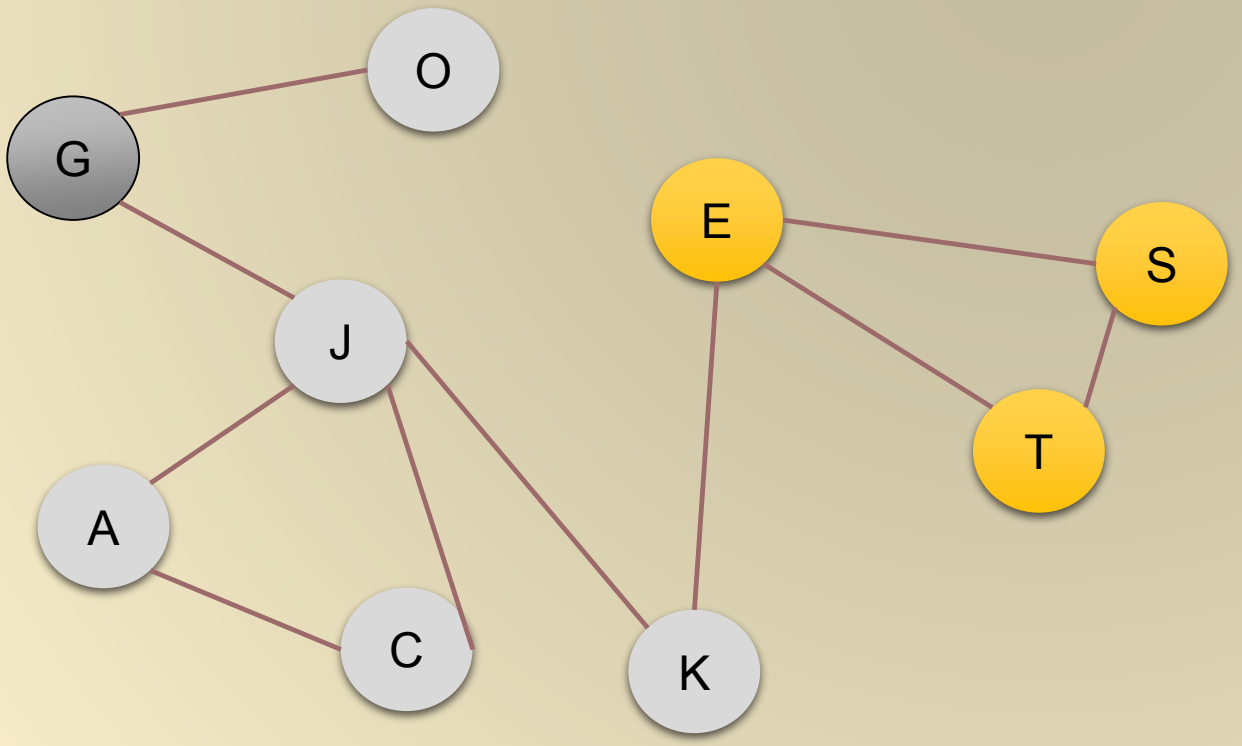
DFS Example
Traverse Graph G



stack
J
G

Output: G O J A C

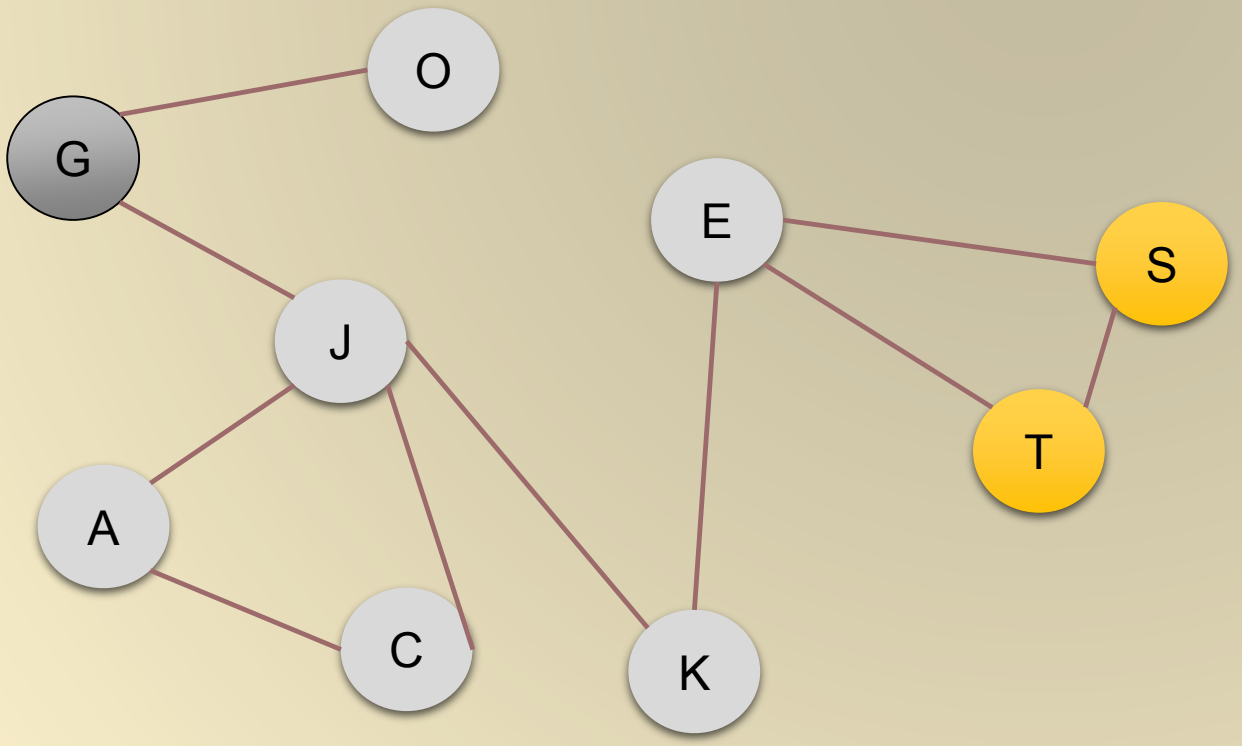
DFS Example
Traverse Graph G



stack
K
J
G

Output: G O J A C K

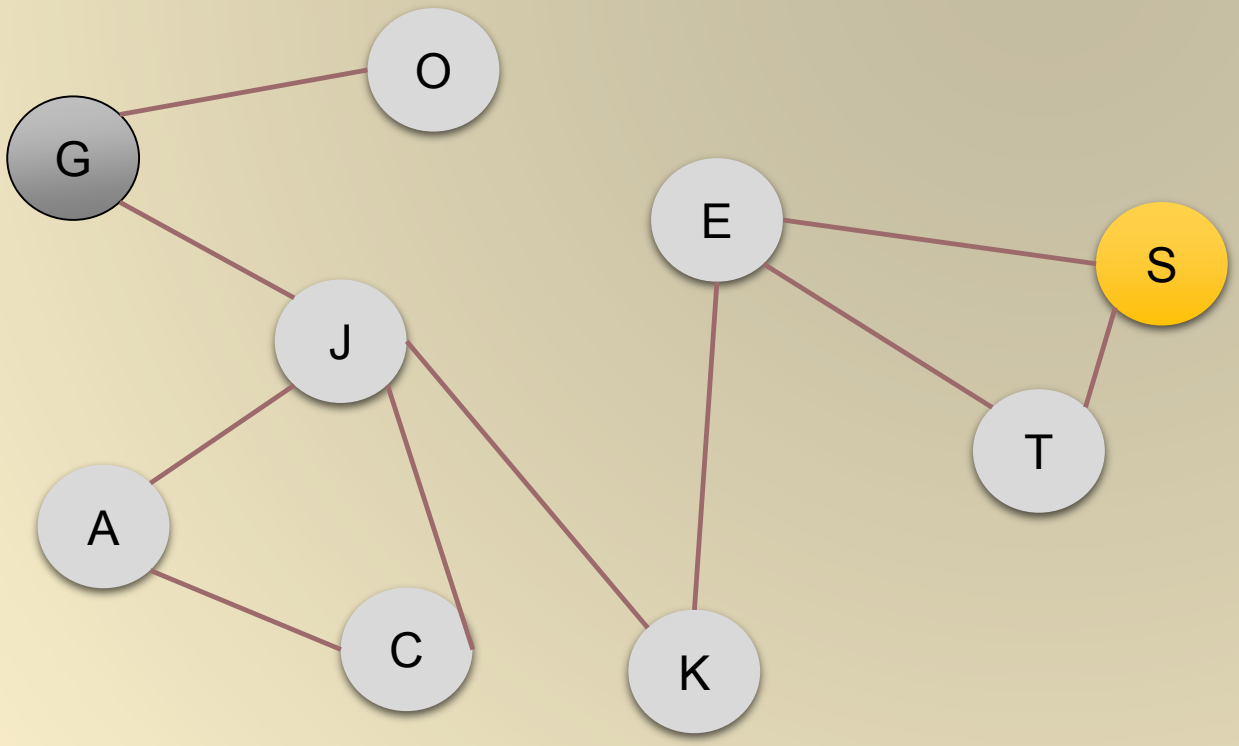
DFS Example
Traverse Graph G



stack
E
K
J
G

Output: G O J A C K E

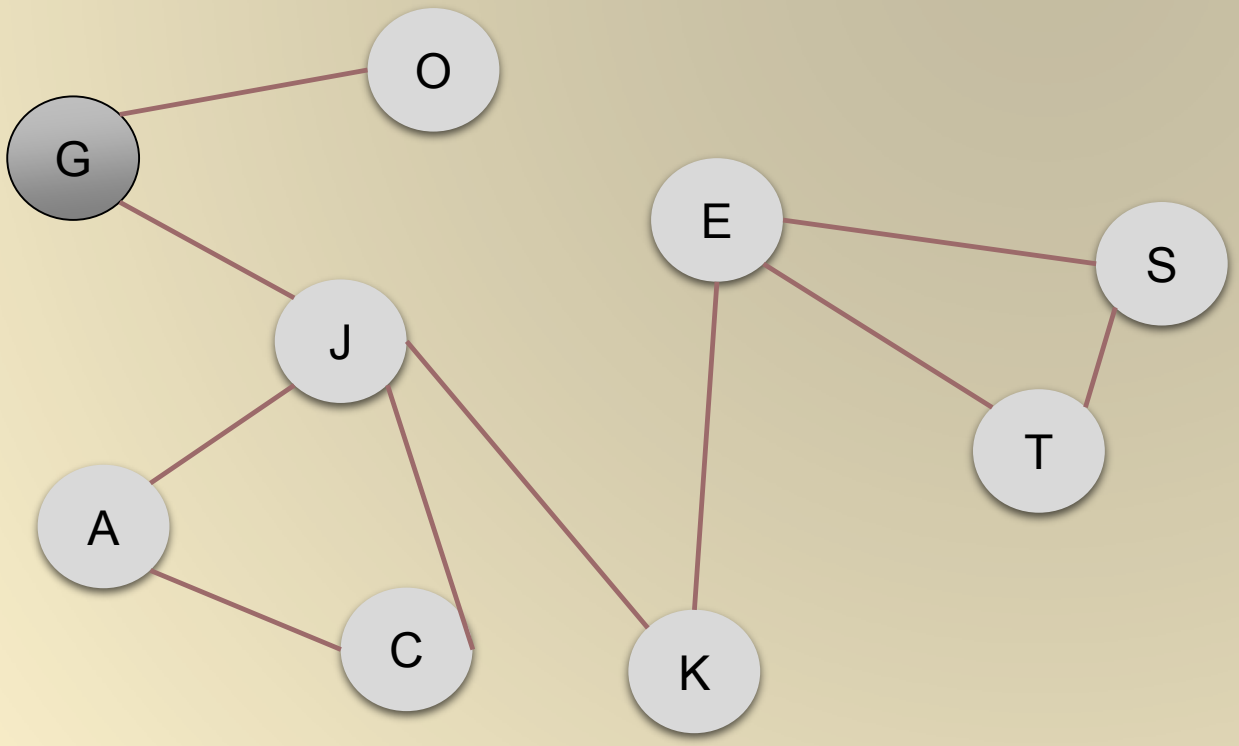
DFS Example
Traverse Graph G



stack
T
E
K
J
G

Output: G O J A C K E T

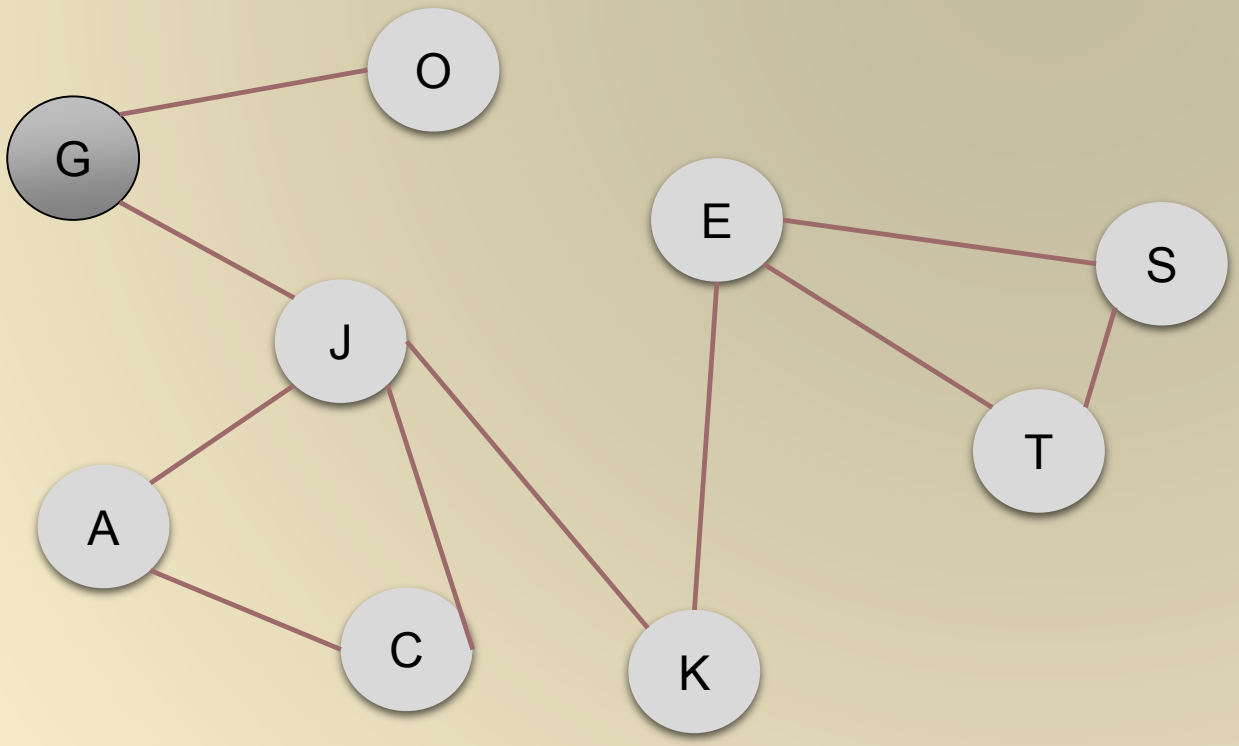
DFS Example
Traverse Graph G



stack
S
T
E
K
J
G

Output: G O J A C K E T S

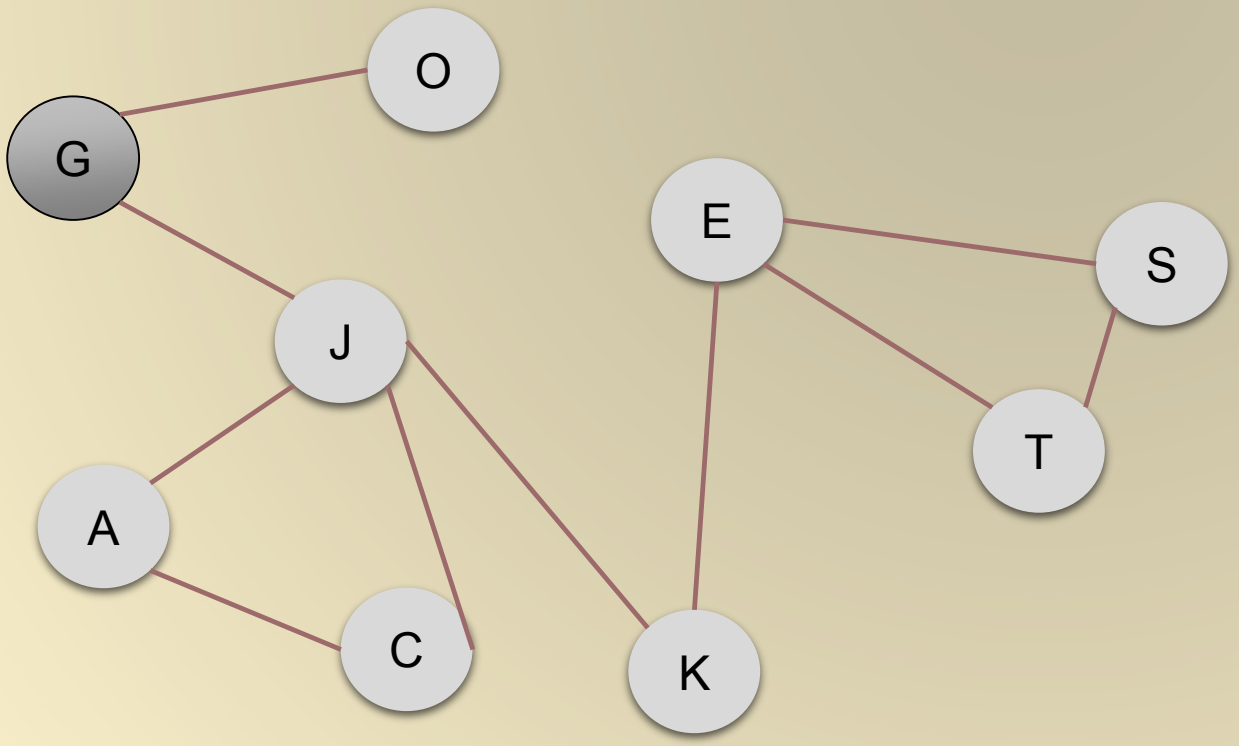
DFS Example
Traverse Graph G



stack
T
E
K
J
G

Output: G O J A C K E T S

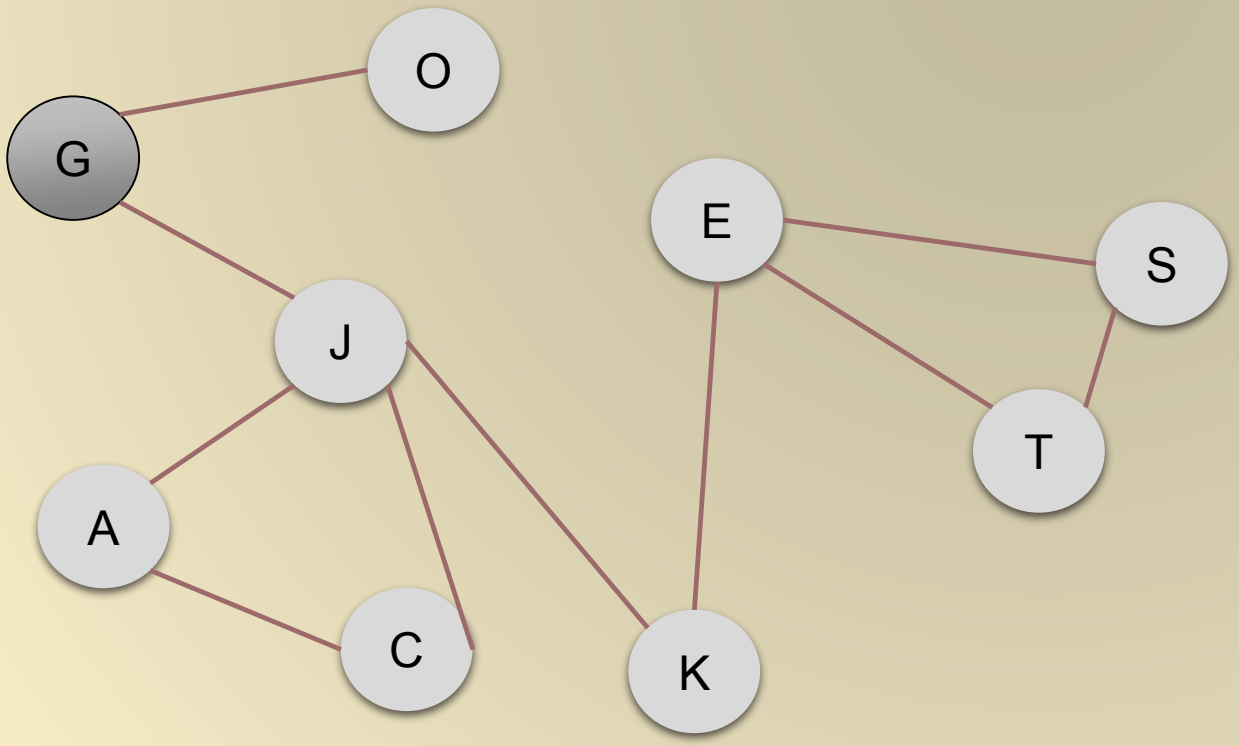
DFS Example
Traverse Graph G



stack
E
K
J
G

Output: G O J A C K E T S

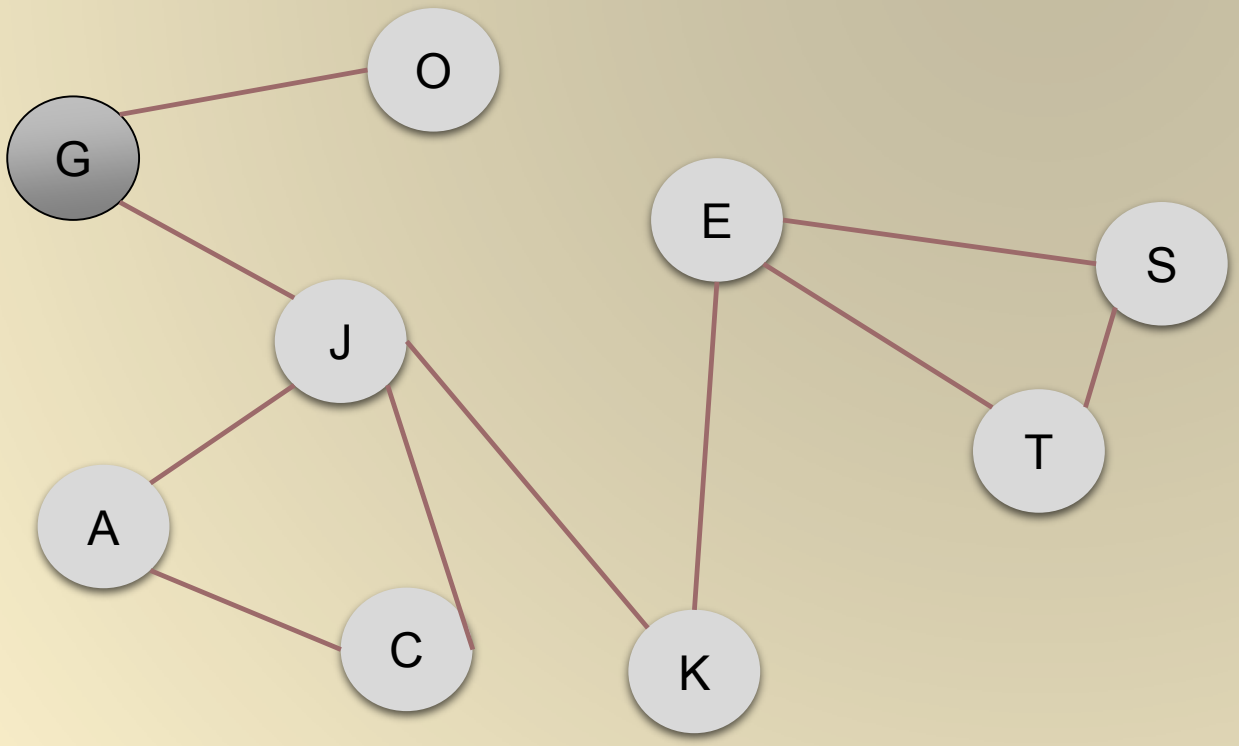
DFS Example
Traverse Graph G



stack
K
J
G

Output: G O J A C K E T S

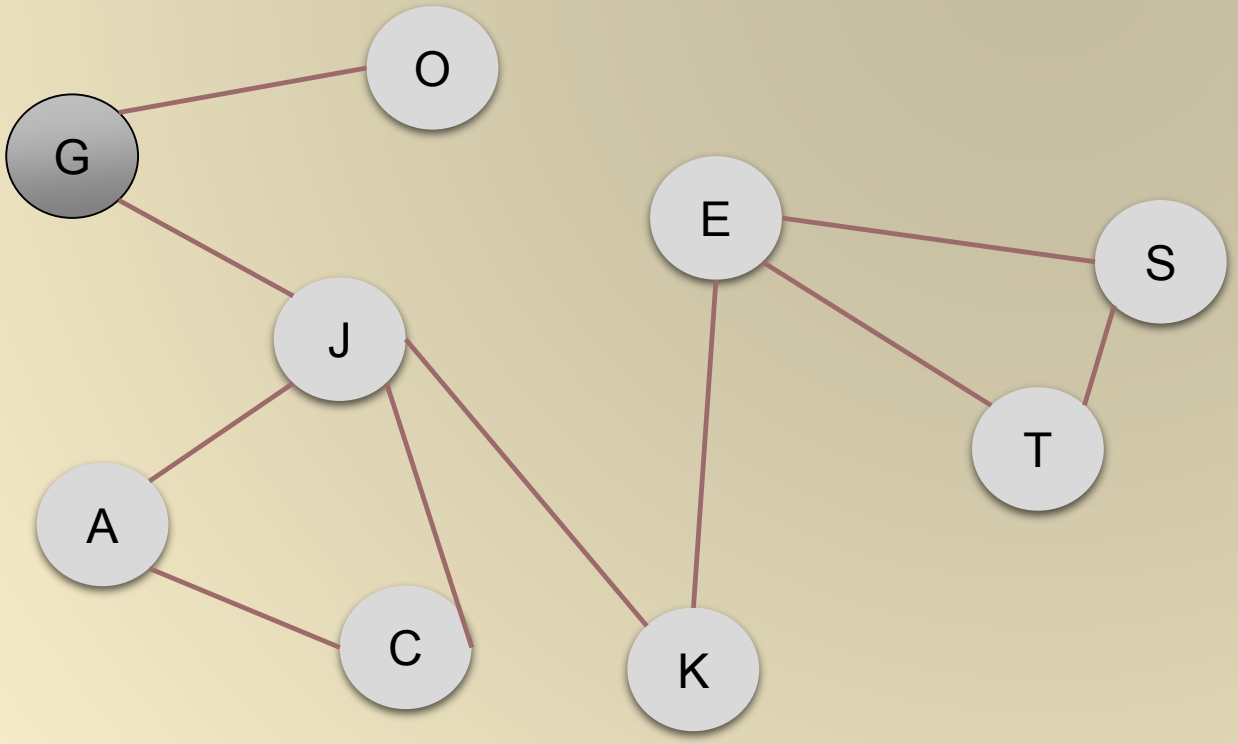
DFS Example
Traverse Graph G



stack
J
G

Output: G O J A C K E T S

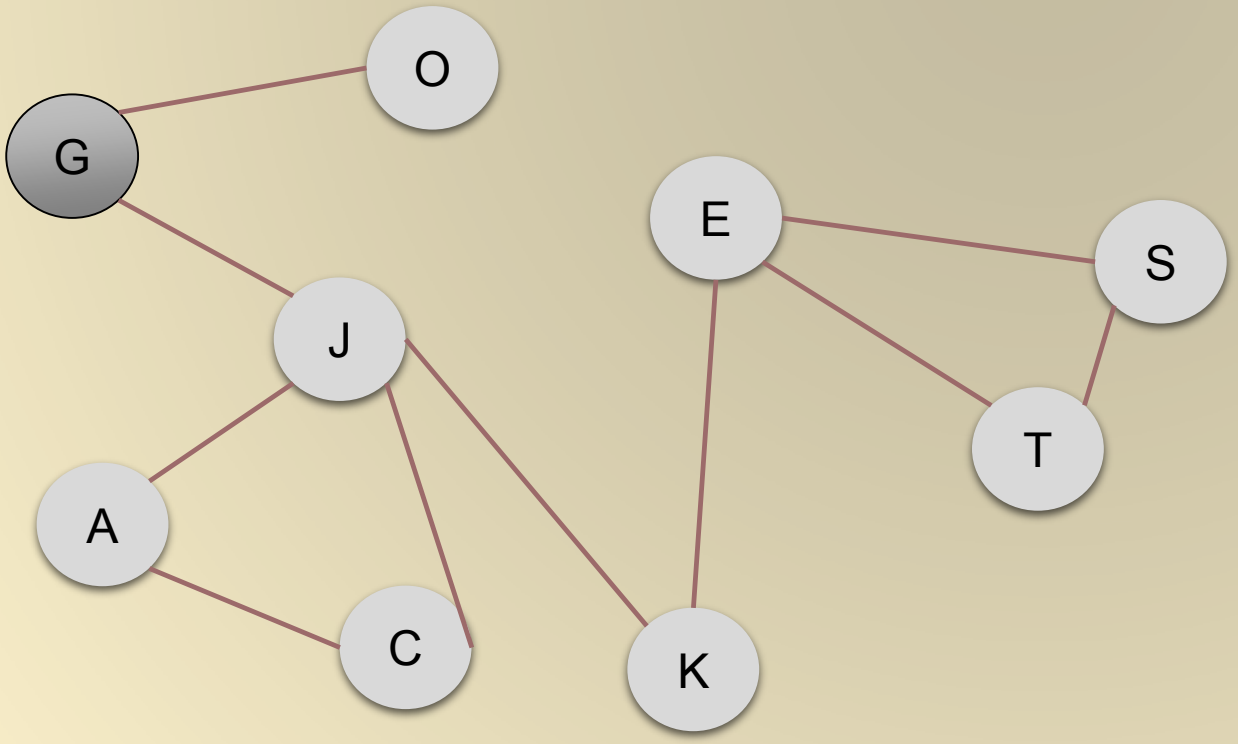
DFS Example
Traverse Graph G



stack
G

Output: G O J A C K E T S

DFS Example
Traverse Graph G



stack

Output: G O J A C K E T S

Breadth-First Search BFS

Problem: It is another systematic way of visiting the vertices of a graph G. Start from a vertex, step forward all vertices adjacent to it, then step forward all vertices adjacent to its sons,...

The Breadth-First Search algorithm is quite the same algorithm as the iterative DFS, you simply replace the stack with a queue

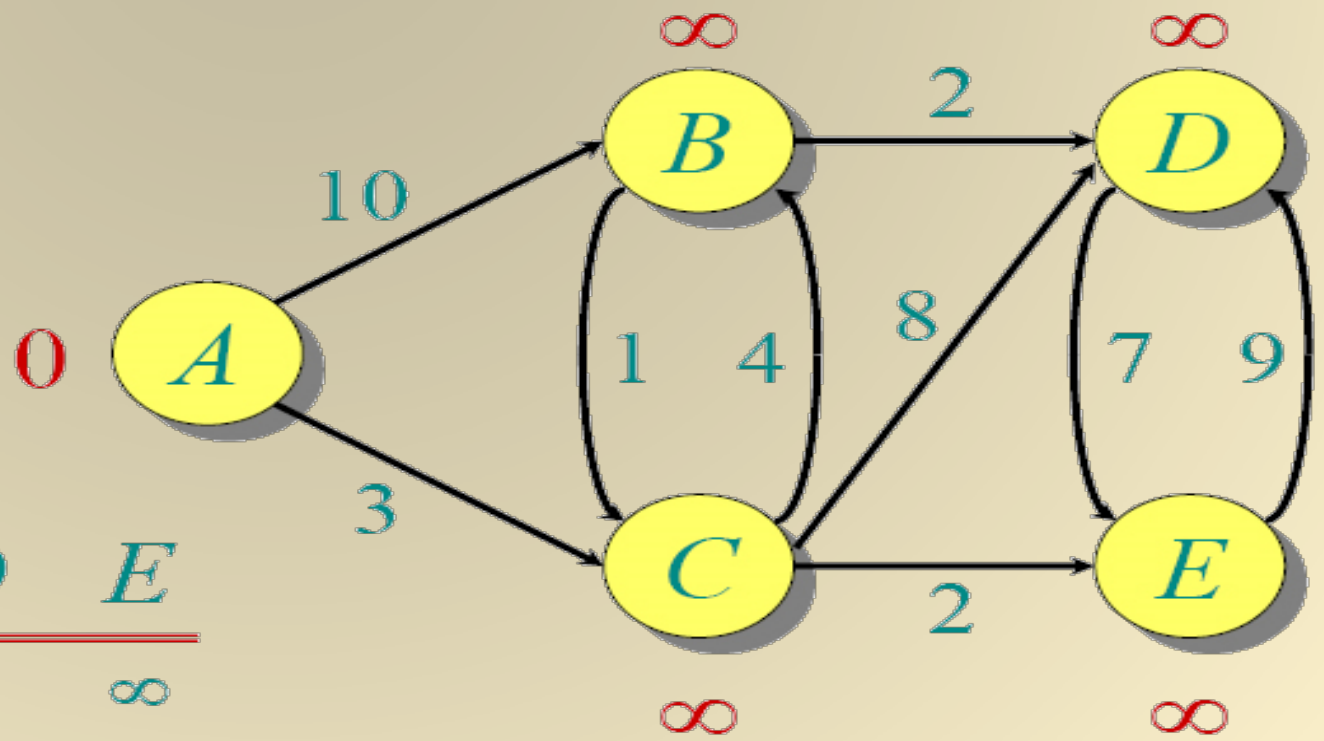
- BFS is classic method to find a path with the fewest nodes from source vertex v to target vertex u.

BFS Pseudo Code

```
bfs(s)
   initialize Q to be a queue with one element s
   while Q not empty
      take a node u from Q
      if explored[u]=false then
         set explored[u]= true
         for each edge (u,v) adjacent to u
            add v to Q
         end
      end
end
```

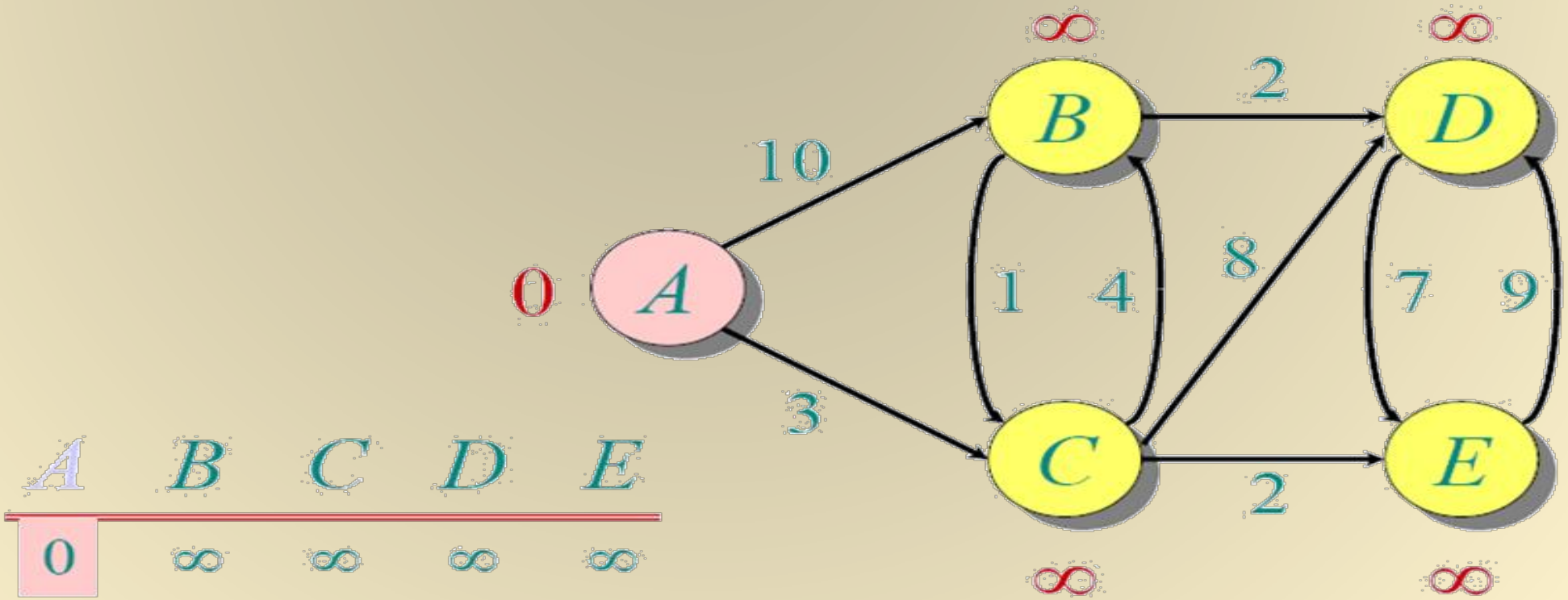
Dijkstra Animated Example

Initialize:

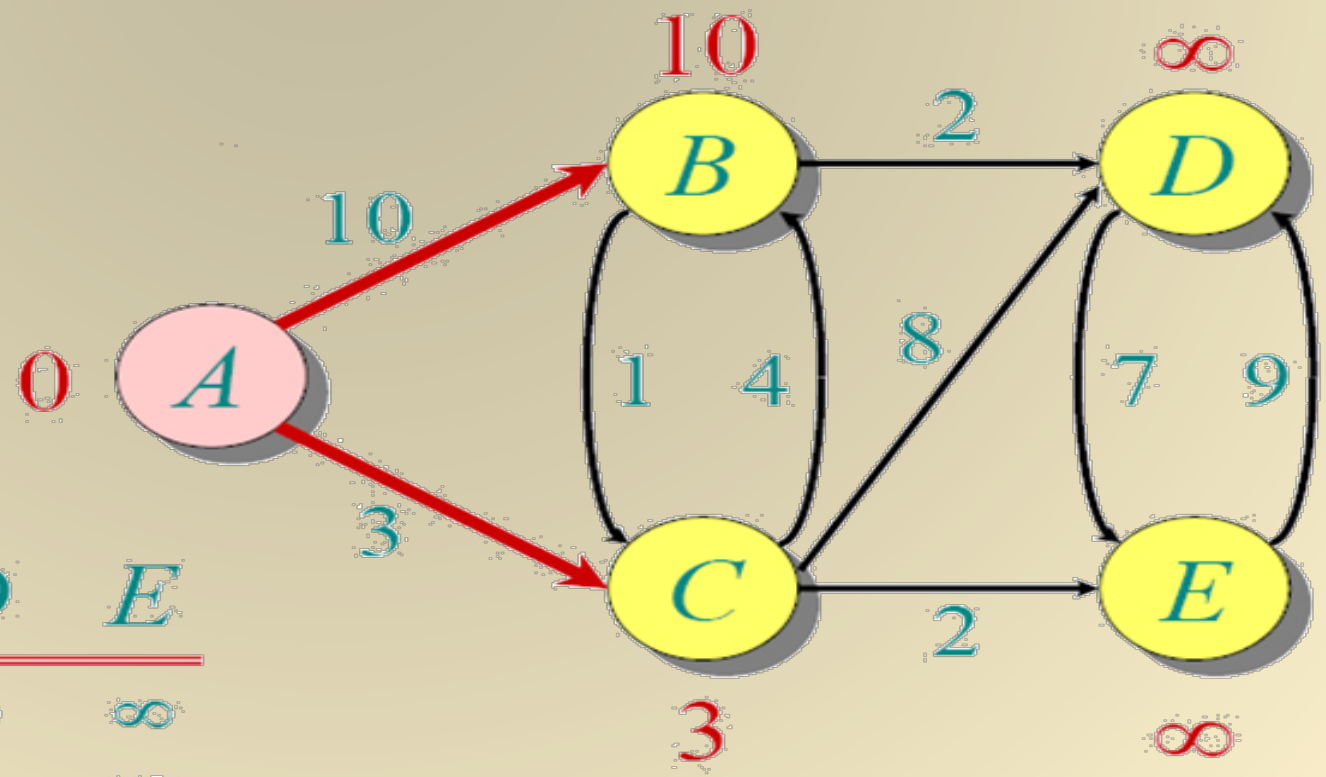


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

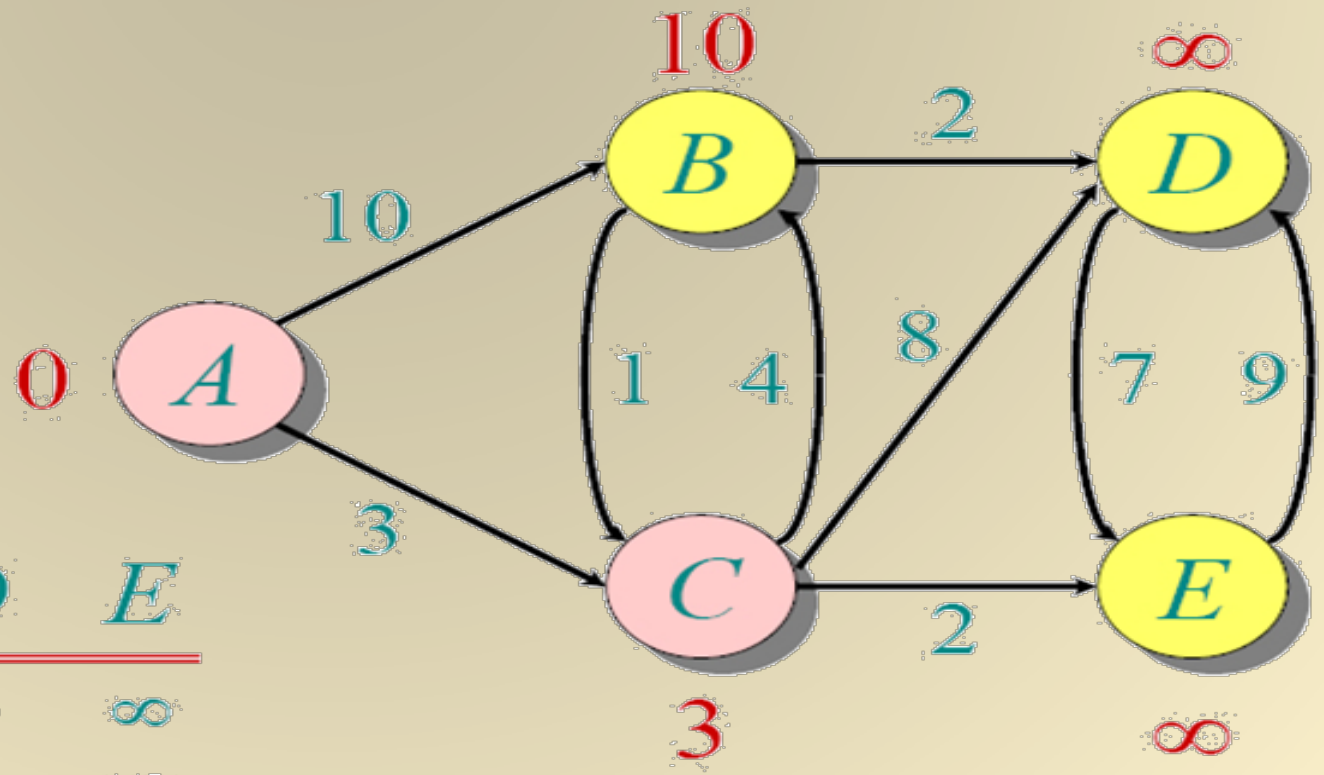


Q:	A	B	C	D	E
	0	∞	∞	∞	∞



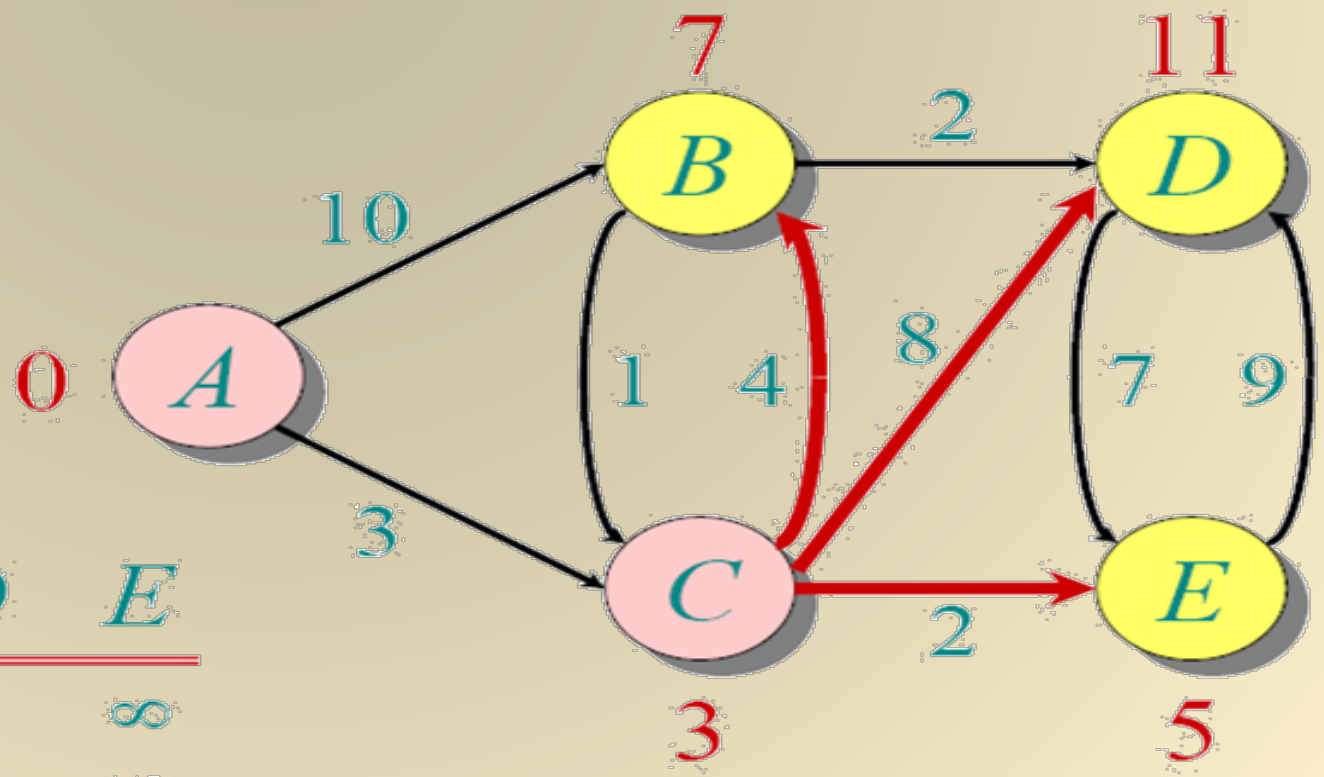
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }



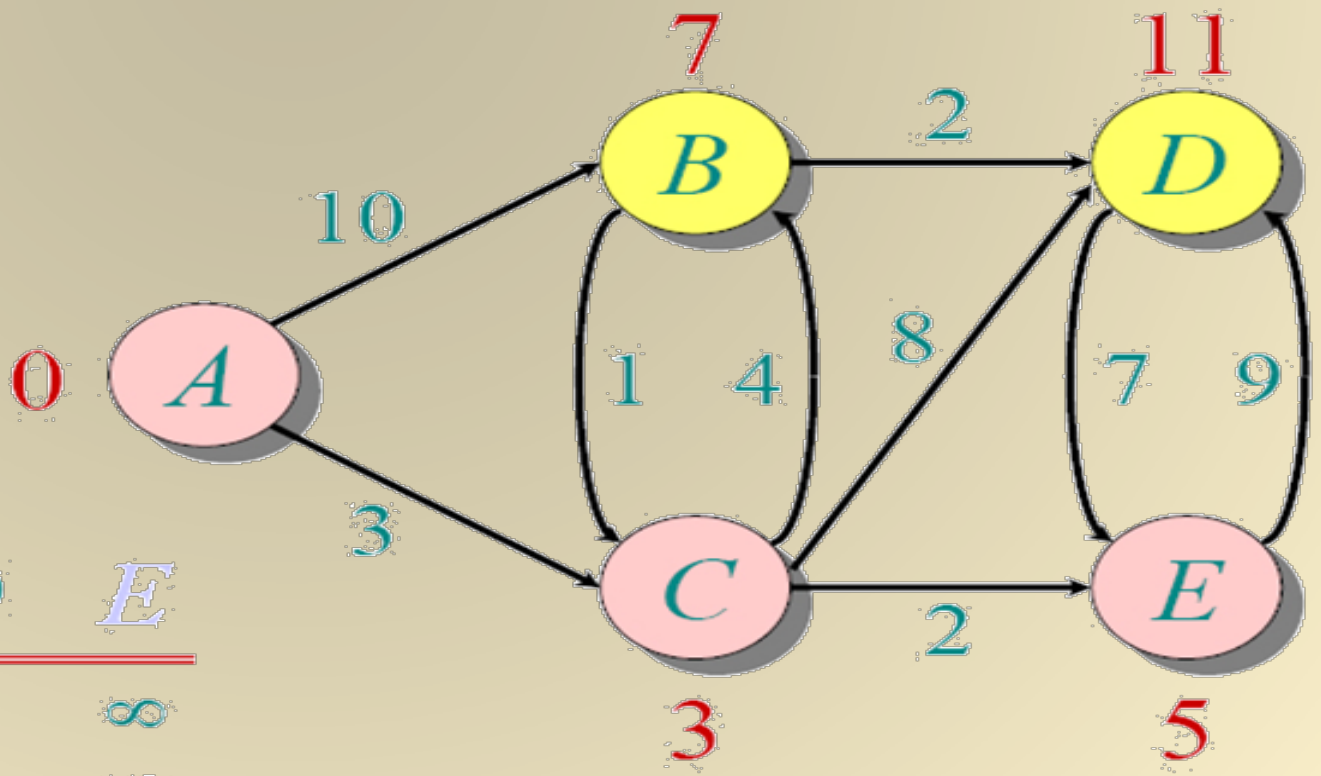
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }



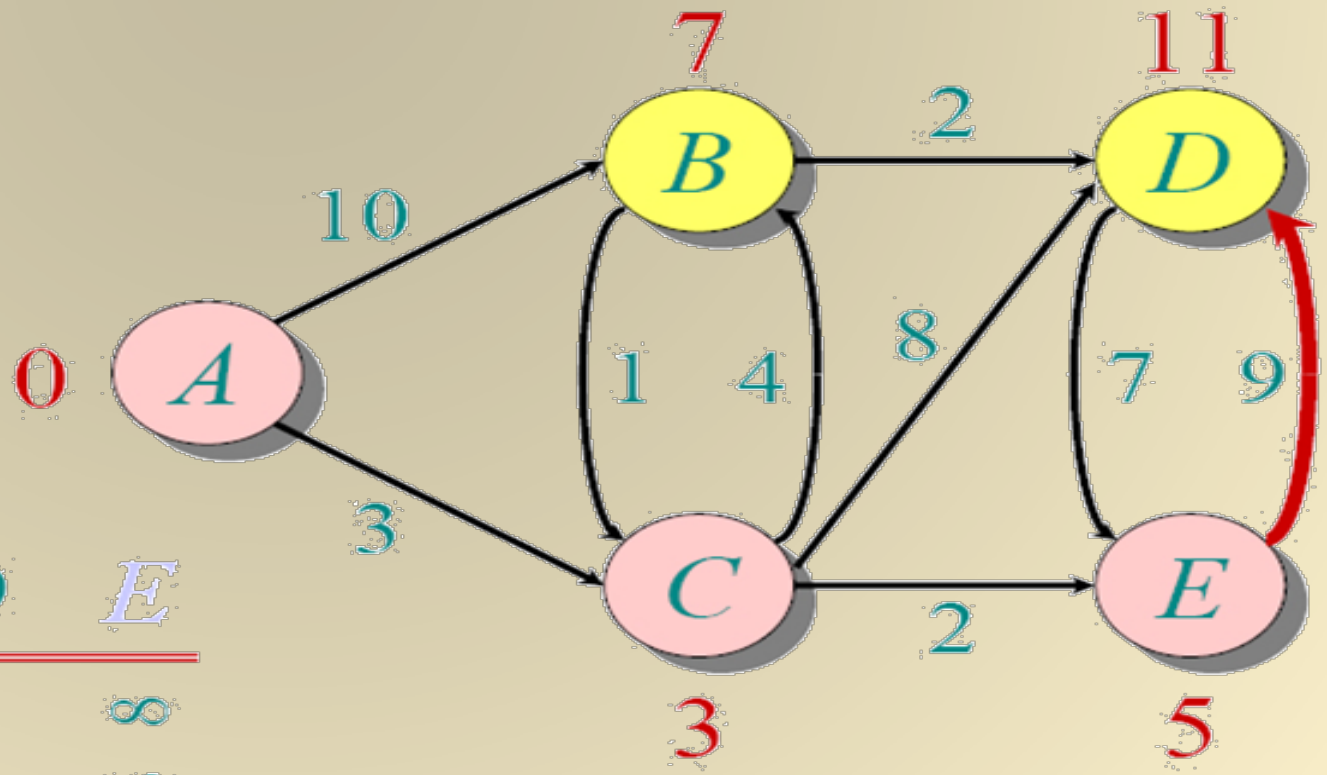
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C \}$



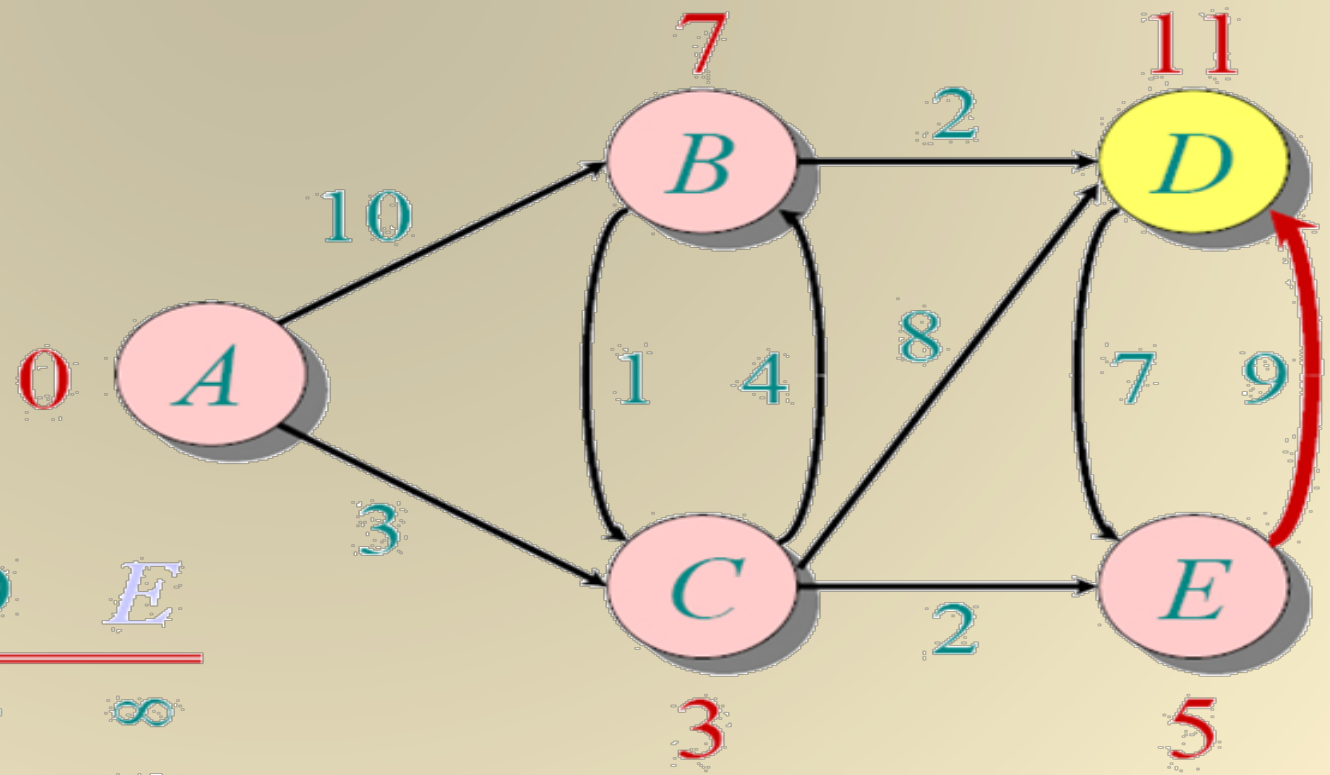
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }



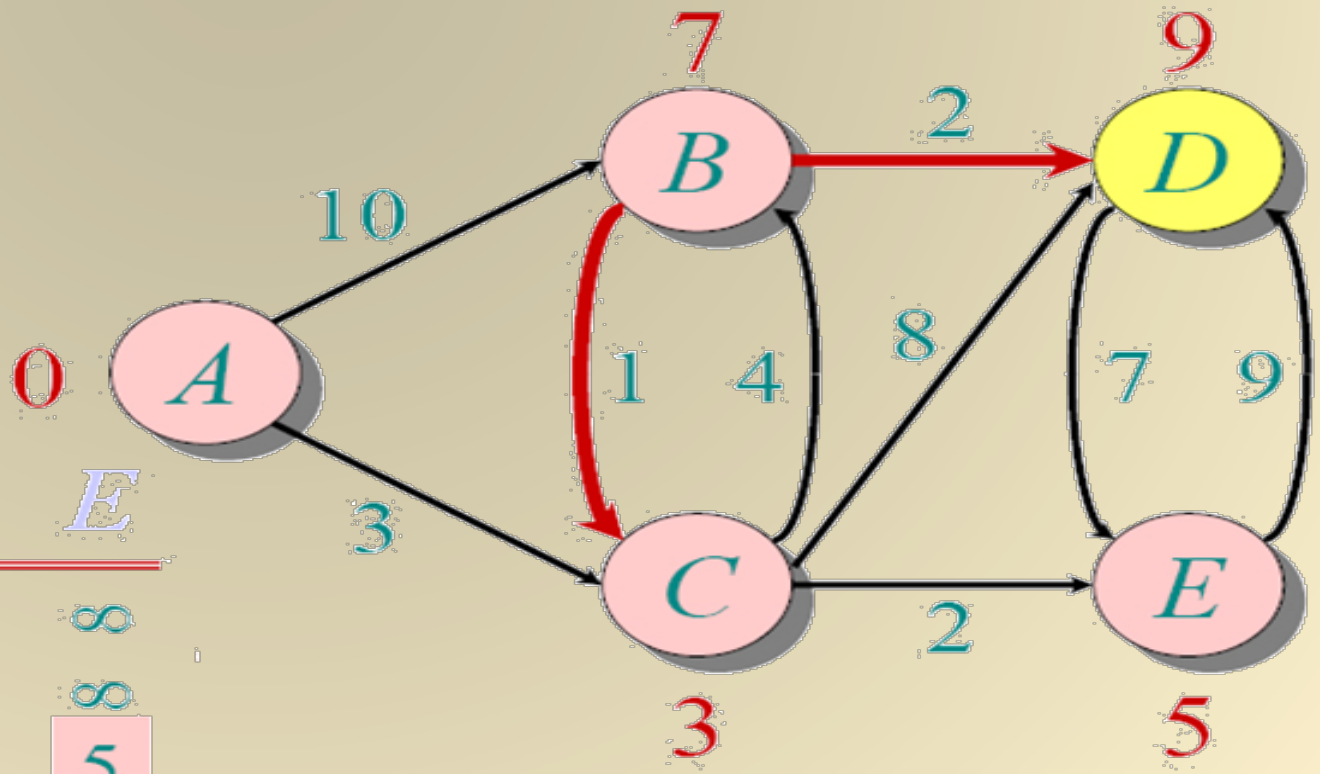
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }



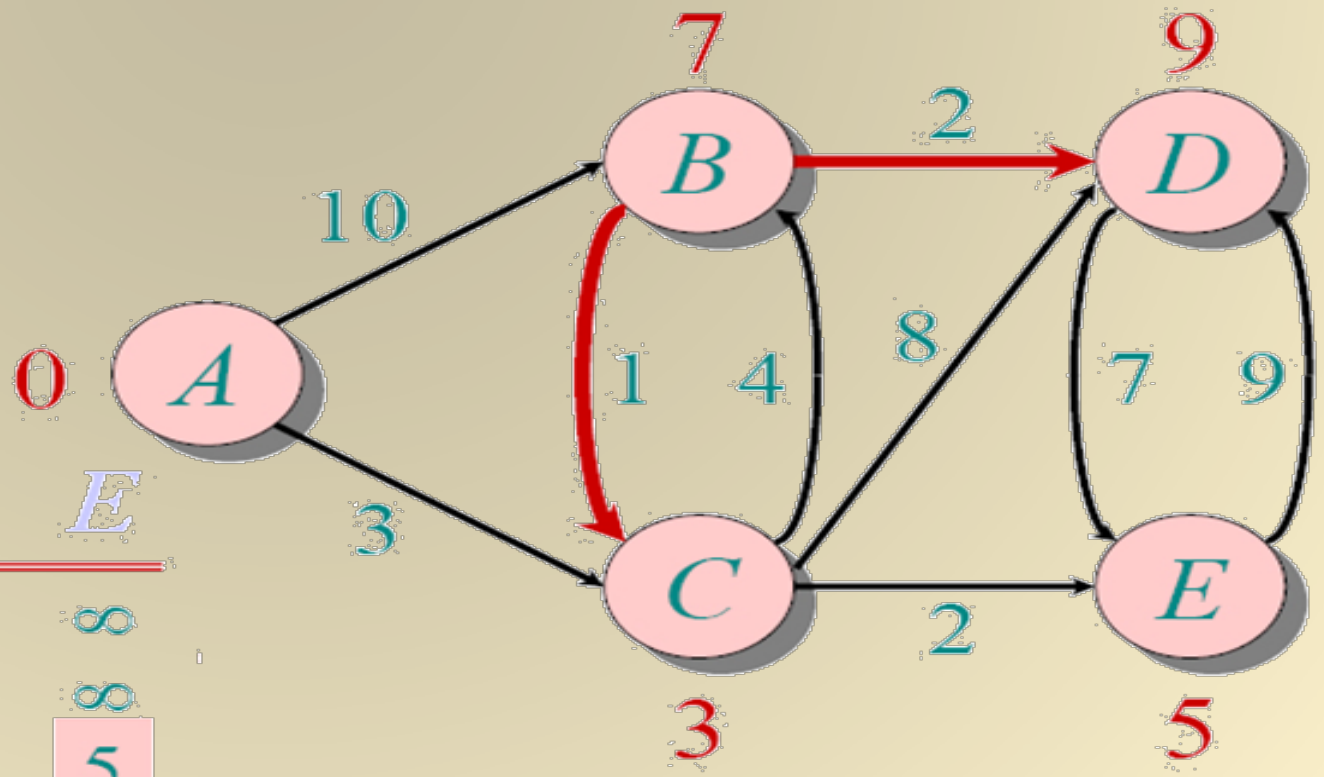
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E, B \}$



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

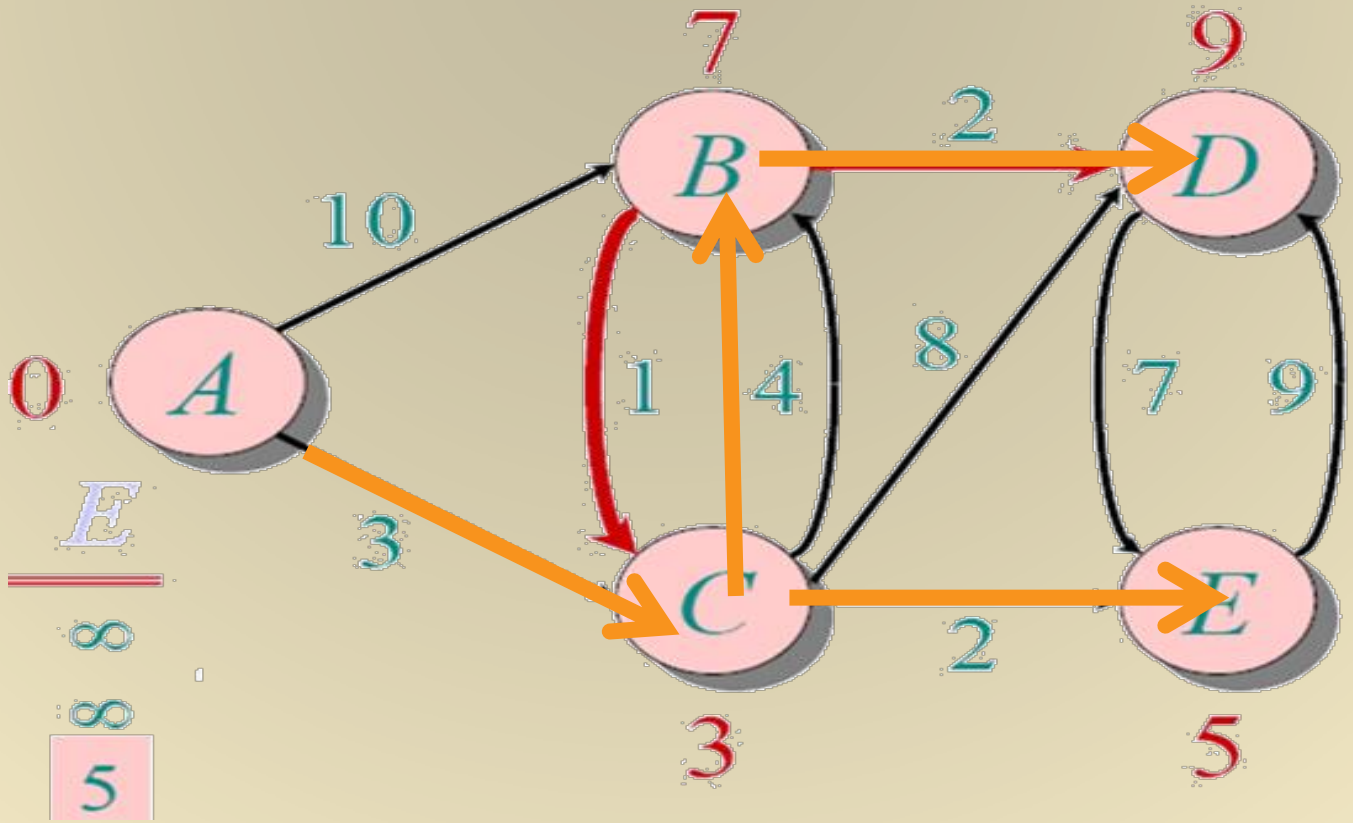
S: { A, C, E, B }



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

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

Shortest Path Tree from A to all other nodes

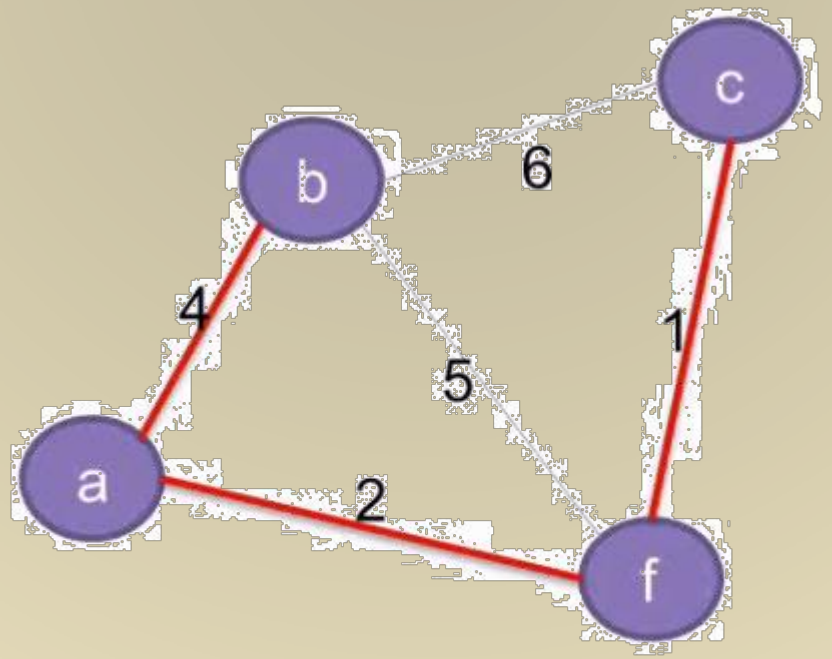


Minimum Spanning Tree MST

A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree that connects all vertices in G and has a minimum total weight

- Concept: Let V be any subset of the vertices of G, and let edge *e* be the *smallest* edge connecting V to G-V. Then *e* is part of the minimum spanning tree.

Minimum Spanning Tree MST



MST = { (a, b), (a, f), (c, f) }

Minimum Spanning Trees

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

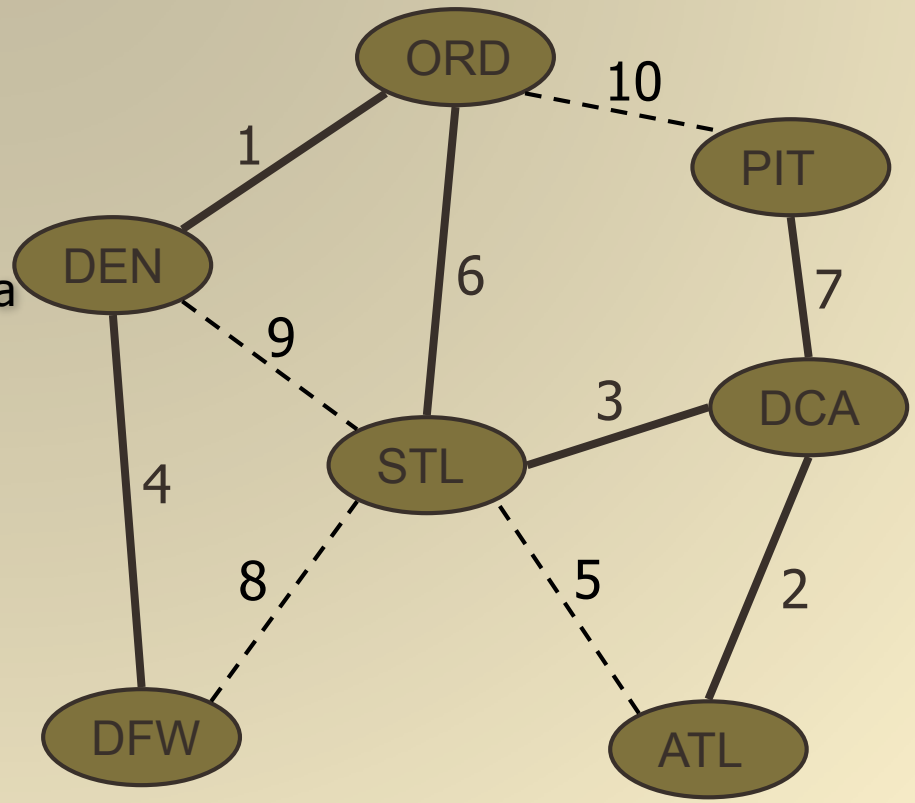
Spanning tree

- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks



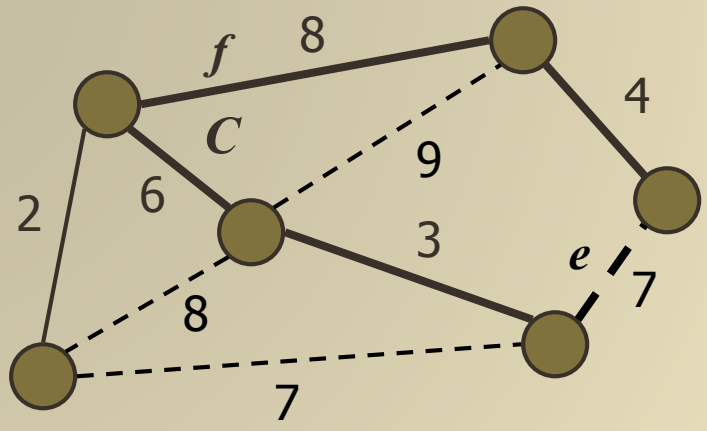
Cycle Property

Cycle Property:

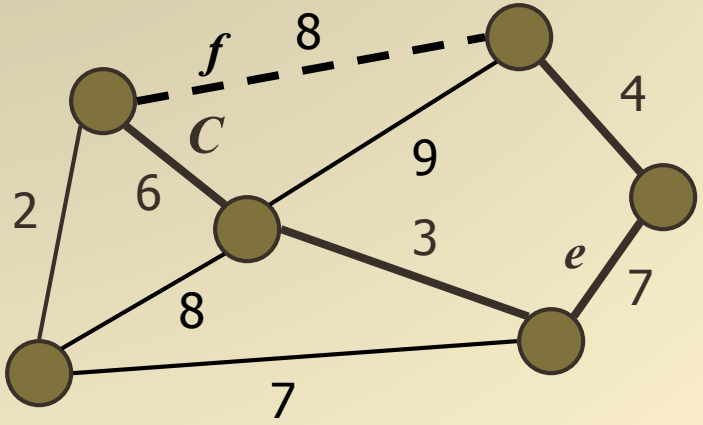
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



Replacing f with e yields a better spanning tree



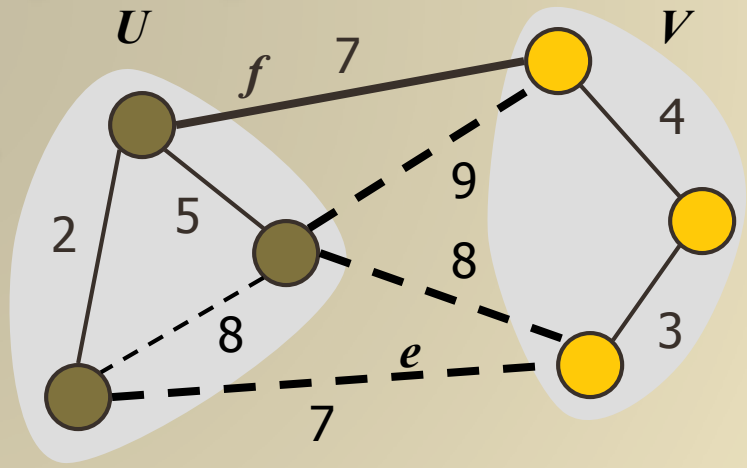
Partition Property

Partition Property:

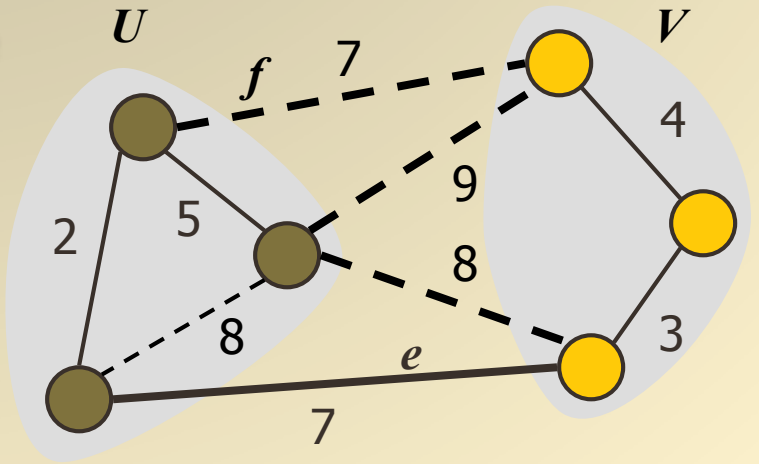
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,
$$weight(f) \leq weight(e)$$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):

Input: An undirected, weighted, connected graph G with n vertices and m edges

Output: A minimum spanning tree T for G

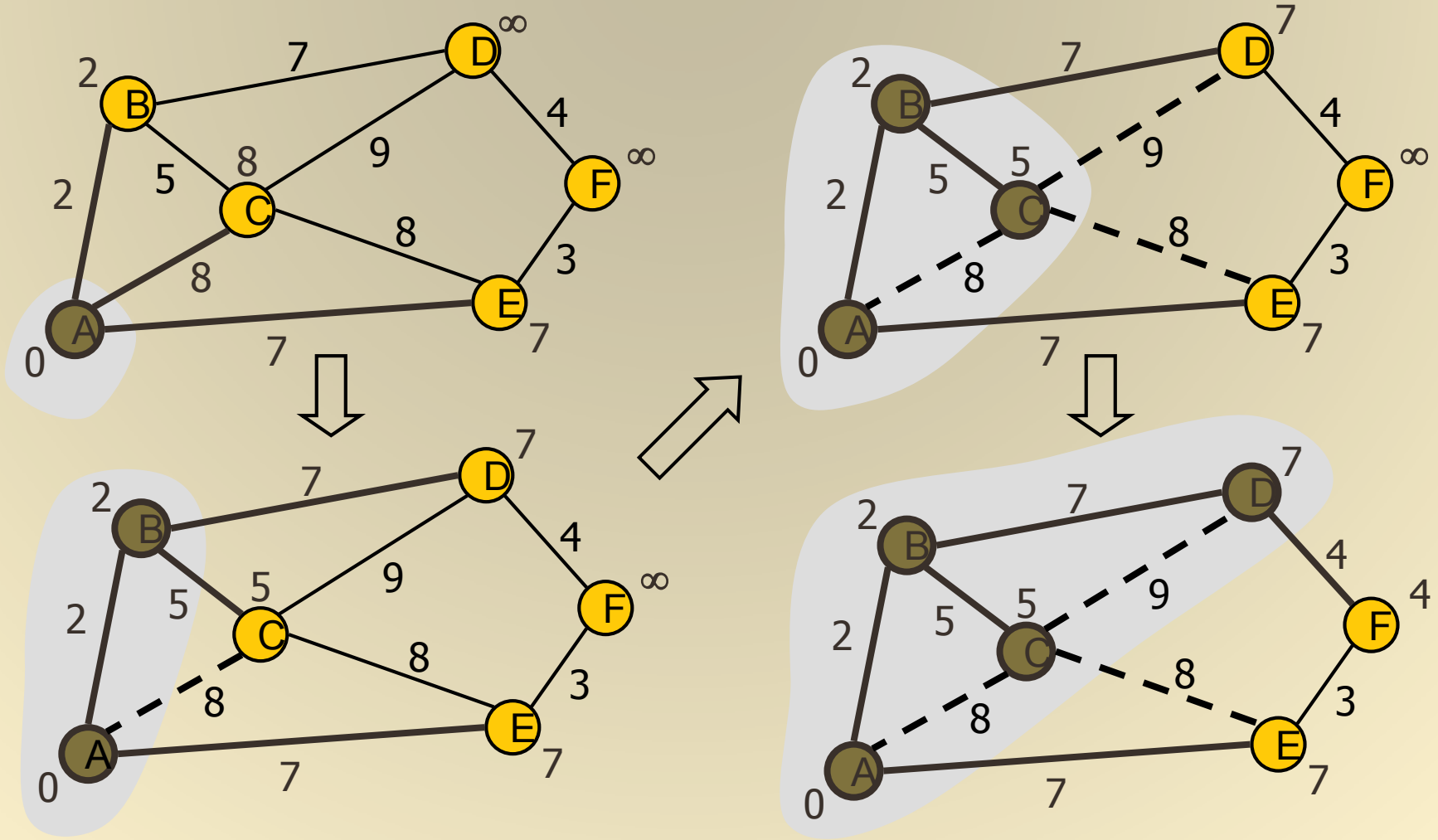
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

MST Pseudo Code

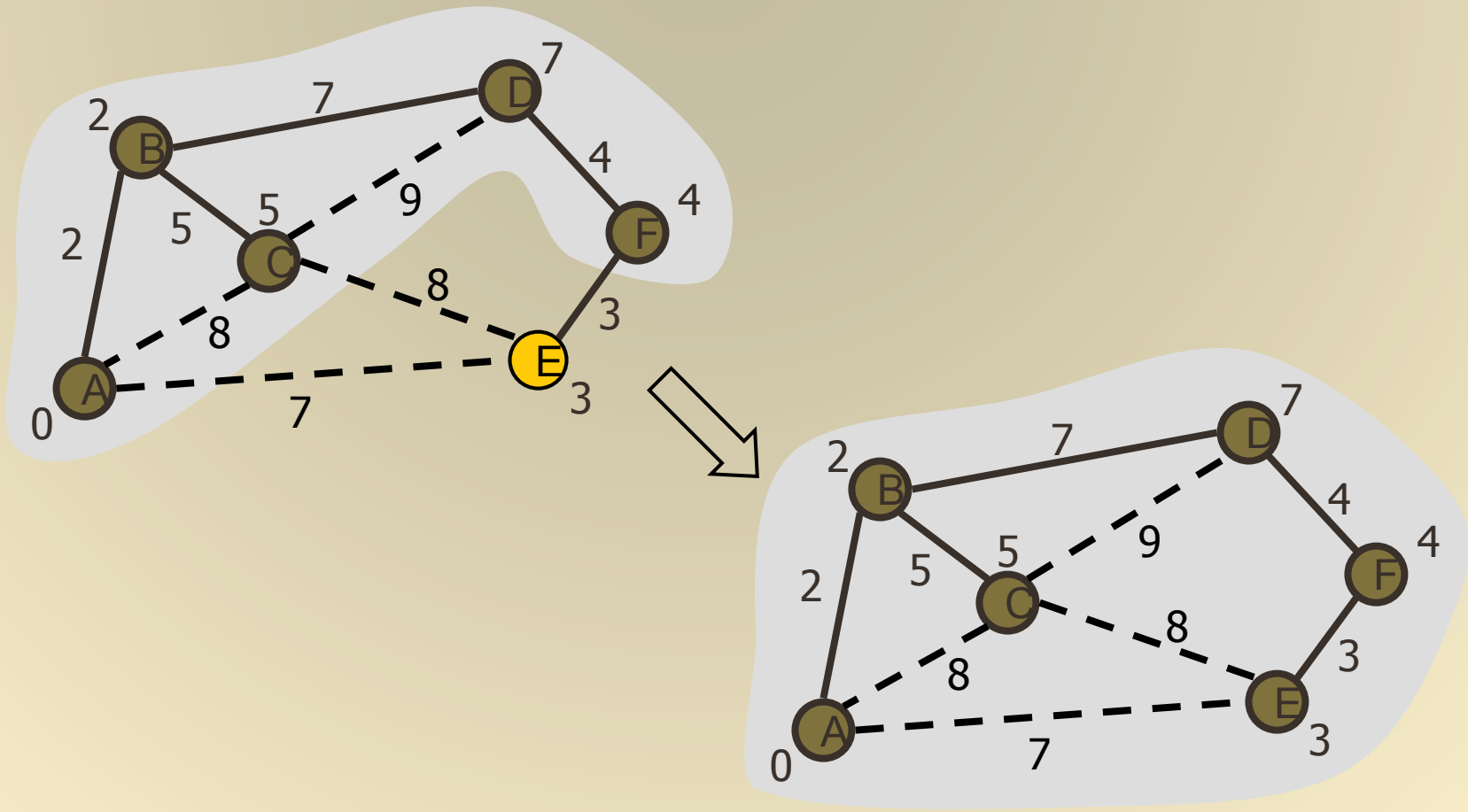
Consider a weighted connected graph G with n vertices. Prim's algorithm finds a minimum spanning tree of G.

```
Prim(G)
  T := a minimum-weight edge
  for i = 1 to n − 2
    e := an edge of minimum weight incident to a vertex in T, and
                 not forming a circuit in T if added to T
    T := T with e added
  end
  return(T)
```

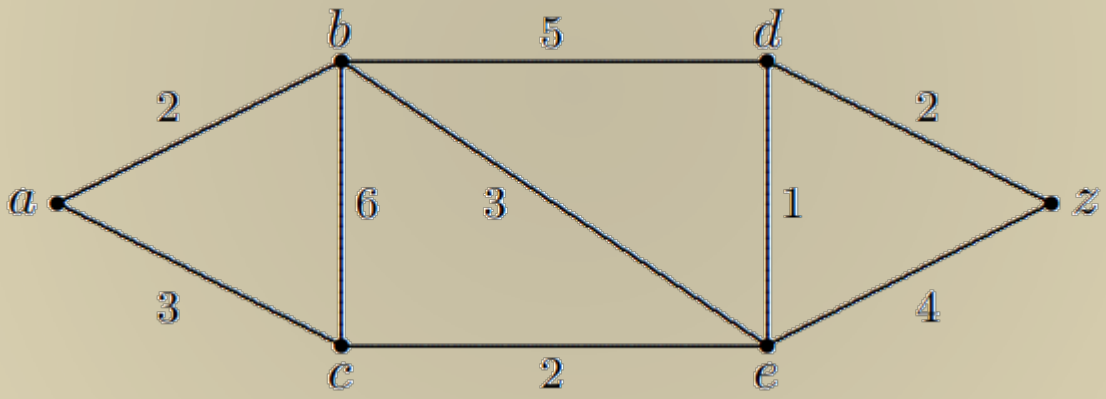
Example



Example (cont.)



MST Example



add edge {d, e}, weight 1

add edge {c, e}, weight 2

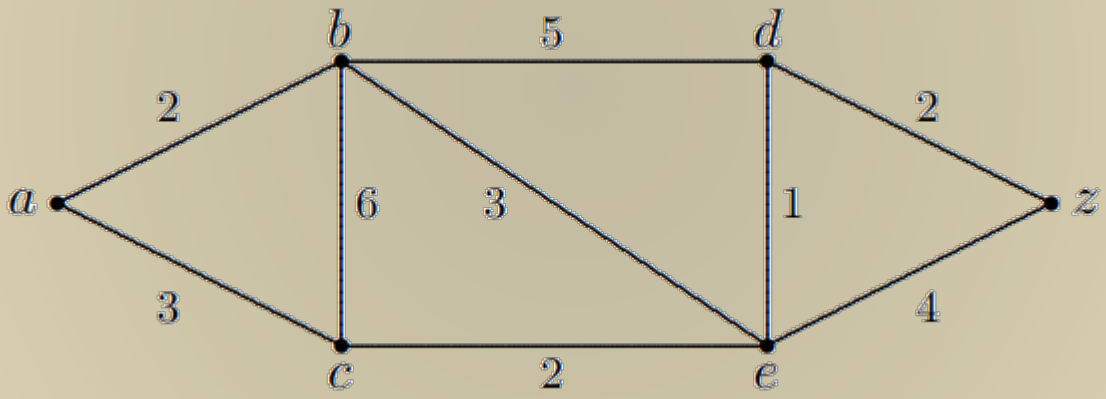
add edge {d, z}, weight 2

add edge {b, e}, weight 3

add edge {a, b}, weight 2

Produces MST weight 10

MST Example



add edge {d, e}, weight 1

add edge {c, e}, weight 2

add edge {d, z}, weight 2

add edge {b, e}, weight 3

add edge {a, b}, weight 2

Produces MST weight 10