



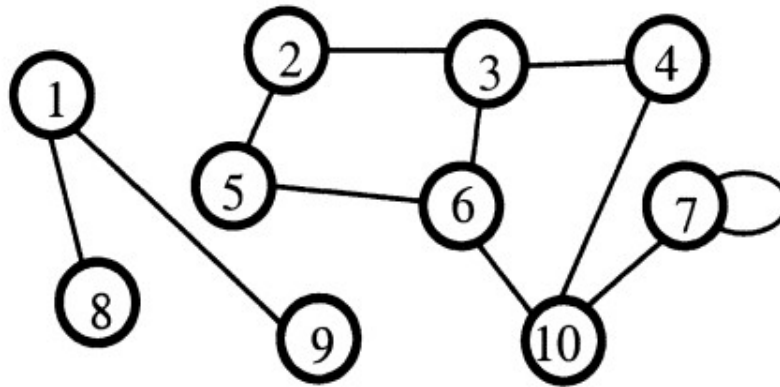
Graphs

Graphs

- A graph is a pair (V, E) , where
 - V is a set of nodes, called vertices
 - E is a collection of pairs of vertices, called edges
 - Vertices and edges are positions and store elements

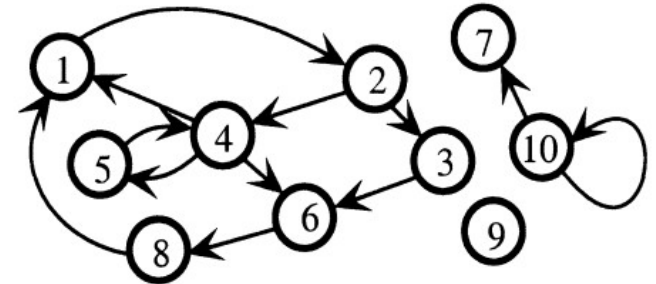
$V = \{1, 2, \dots, 10\}$

$E = \{(1, 8), (1, 9), (2, 3), (2, 5), (3, 4), (3, 6), (4, 10), (5, 6), (6, 10), (7, 7), (7, 10)\}$



Edge Types

- ❑ Directed edge
 - ordered pair of vertices (u, v)
 - first vertex u is the origin
 - second vertex v is the destination
 - e.g., a flight
- ❑ Undirected edge
 - unordered pair of vertices (u, v)
 - e.g., a flight route
- ❑ Directed graph
 - all the edges are directed
 - e.g., route network
- ❑ Undirected graph
 - all the edges are undirected
 - e.g., flight network

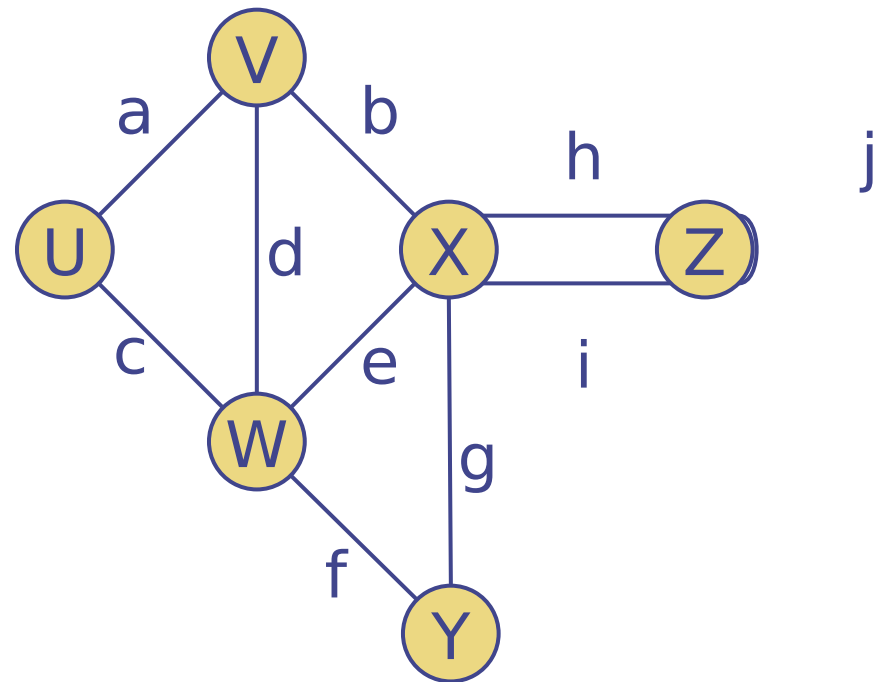


Applications

- Shortest path finding (navigation systems),
- Minimum spanning tree (communication networks),
- Scheduling and resource allocation
- Social networks
- Flight/Road networks
- Image processing
- Machine Learning
- Game Development and many many more...

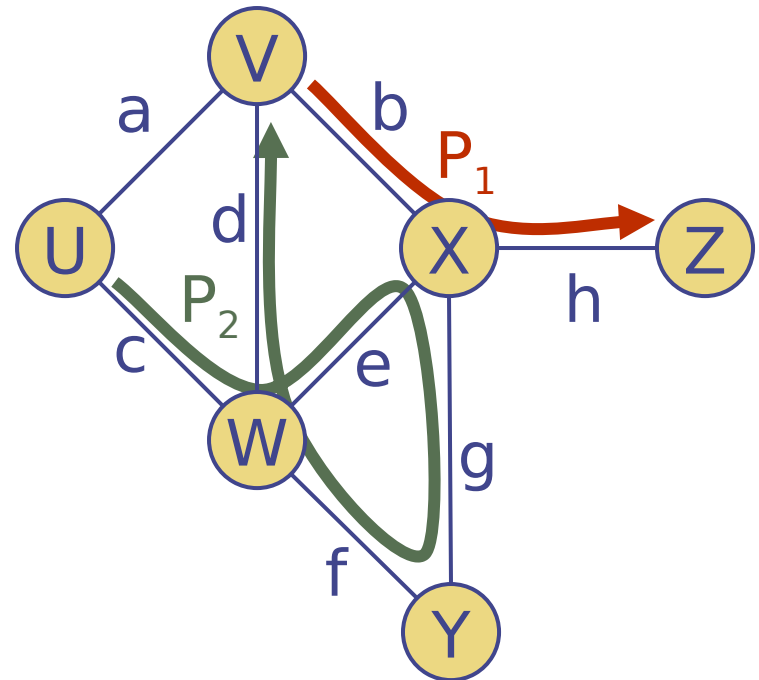
Terminology

- End vertices (or endpoints) of an edge
 - U and V are the endpoints of a
- Edges incident on a vertex
 - a, d, and b are incident on V
- Adjacent vertices
 - U and V are adjacent
- Degree of a vertex
 - X has degree 5
- Parallel edges
 - h and i are parallel edges
- Self-loop
 - j is a self-loop



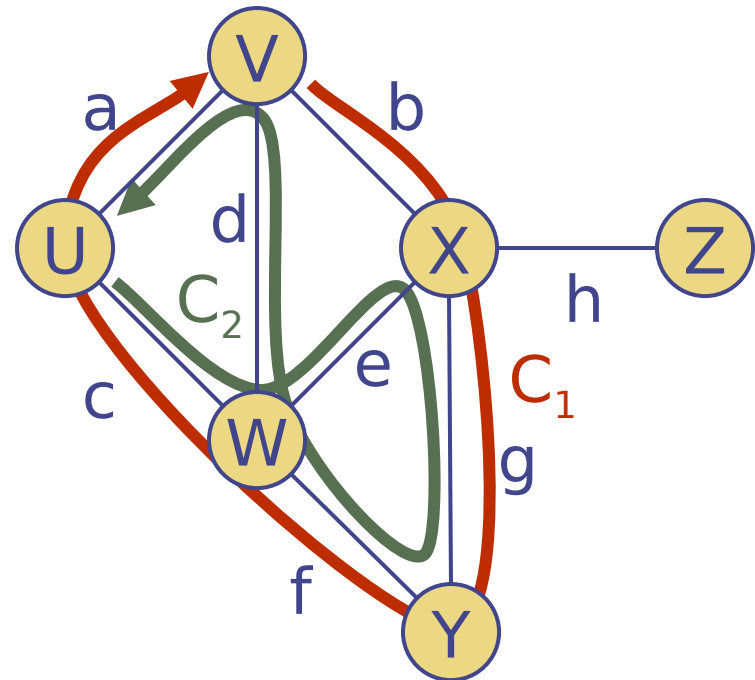
Terminology (cont.)

- Path
 - sequence of alternating vertices and edges
 - begins with a vertex
 - ends with a vertex
 - each edge is preceded and followed by its endpoints
- Simple path
 - path such that all its vertices and edges are distinct
- Examples
 - $P_1 = (V, b, X, h, Z)$ is a simple path
 - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple



Terminology (cont.)

- Cycle
 - circular sequence of alternating vertices and edges
 - each edge is preceded and followed by its endpoints
- Simple cycle
 - cycle such that all its vertices and edges are distinct
- Examples
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is a cycle that is not simple



Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

Notation

n number of vertices

m number of edges

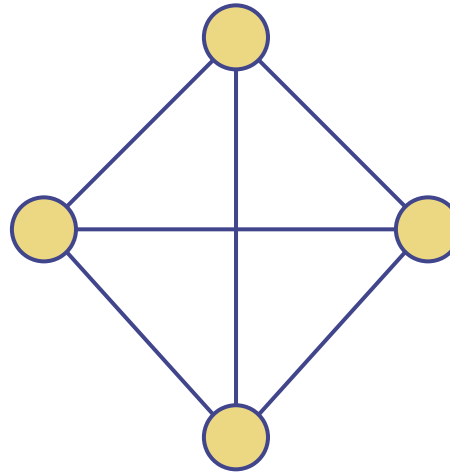
$\deg(v)$ degree of vertex v

Property 2

In an undirected graph with no self-loops and no multiple edges

$$m \leq n(n-1)/2$$

Proof: each vertex has degree at most $(n-1)$



Example

■ $n = 4$

■ $m = 6$

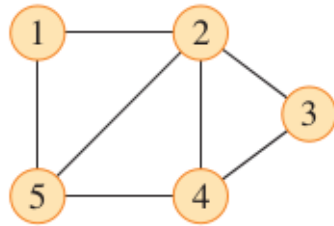
■ $\deg(v) = 3$

What is the bound for a directed graph?

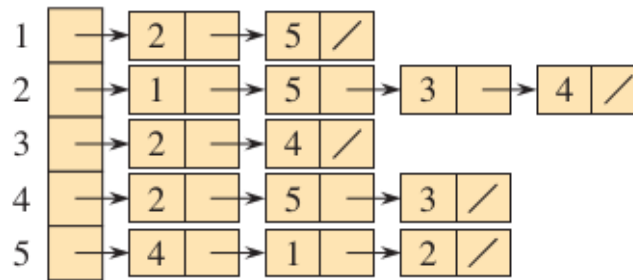
Graph ADT

- `vertex_count()`: Return the number of vertices of the graph.
- `vertices()`: Return an iteration of all the vertices of the graph.
- `edge_count()`: Return the number of edges of the graph.
- `edges()`: Return an iteration of all the edges of the graph.
- `get_edge(u,v)`: Return the edge from vertex u to vertex v , if one exists; otherwise return `None`. For an undirected graph, there is no difference between `get_edge(u,v)` and `get_edge(v,u)`.
- `degree(v, out=True)`: For an undirected graph, return the number of edges incident to vertex v . For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v , as designated by the optional parameter.
- `incident_edges(v, out=True)`: Return an iteration of all edges incident to vertex v . In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to `False`.
- `insert_vertex(x=None)`: Create and return a new Vertex storing element x .
- `insert_edge(u, v, x=None)`: Create and return a new Edge from vertex u to vertex v , storing element x (`None` by default).
- `remove_vertex(v)`: Remove vertex v and all its incident edges from the graph.
- `remove_edge(e)`: Remove edge e from the graph.

Graph Representations



(a)



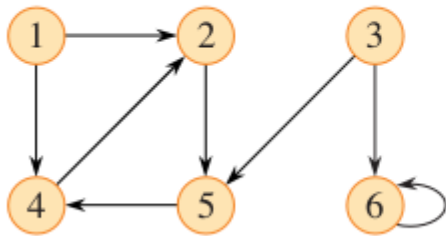
(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

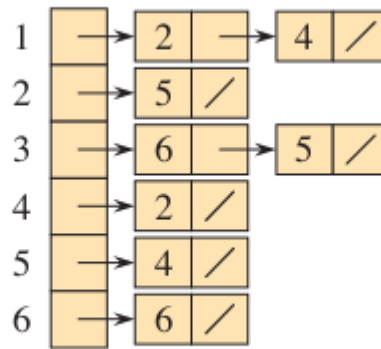
(c)

Two representations of an undirected graph. (a) An undirected graph G with 5 vertices and 7 edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G

Graph Representations



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Two representations of a directed graph. (a) A directed graph G with 6 vertices and 8 edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

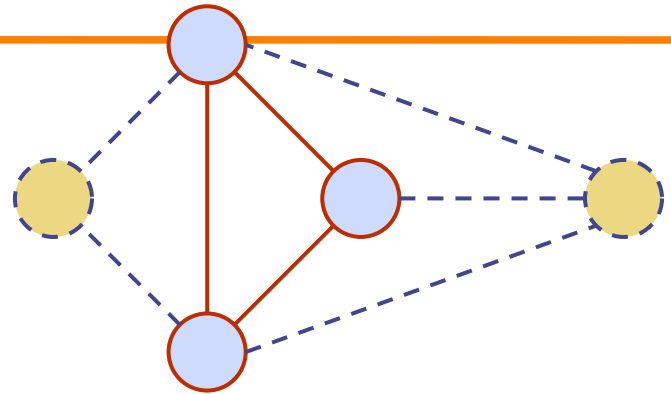
Performance

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

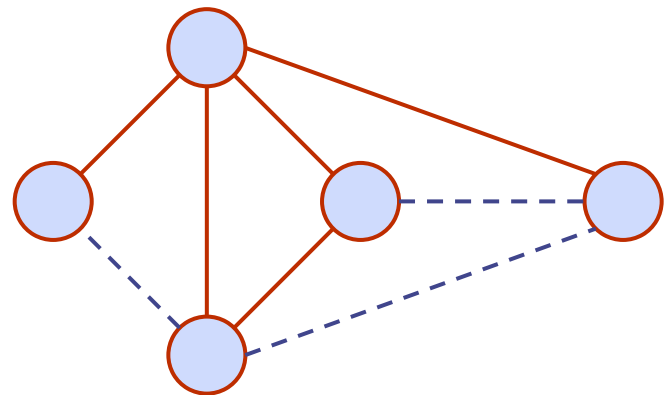
* disallows parallel edges

Subgraphs

- A subgraph S of a graph G is a graph such that
 - The vertices of S are a subset of the vertices of G
 - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



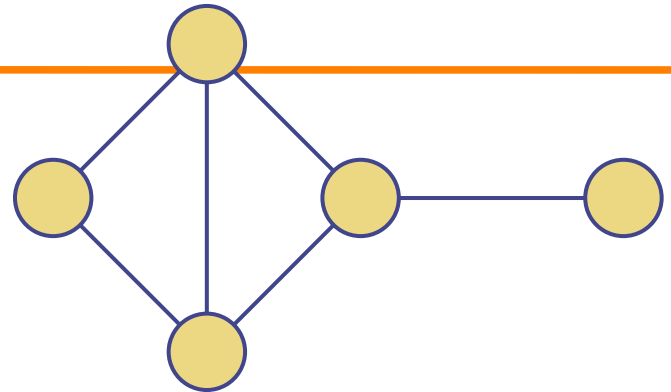
Subgraph



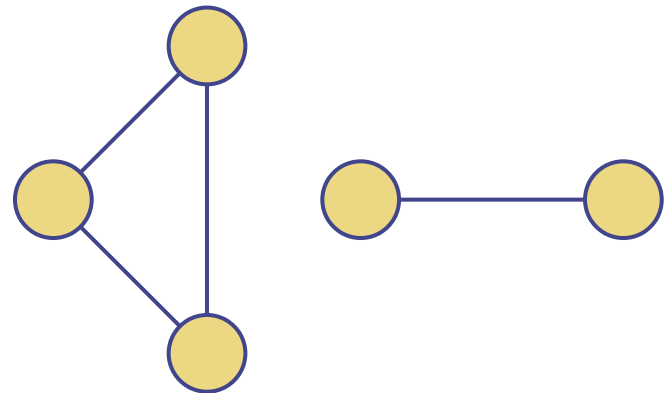
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- If G is not connected, a connected component of a graph G is a maximal connected subgraph of G



Connected graph



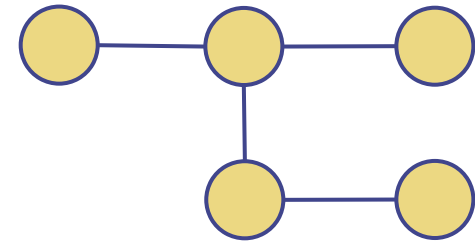
Non connected graph with two connected components

Trees and Forests

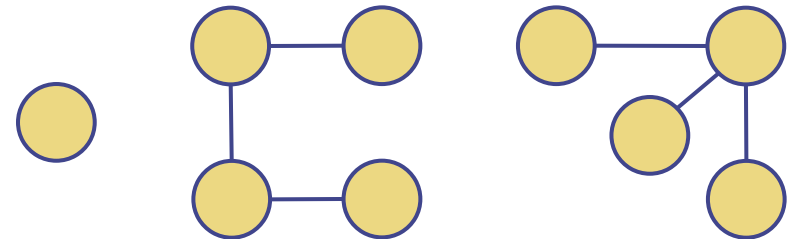
- A tree is an undirected graph T such that
 - T is connected
 - T has no cycles

This definition of tree is different from the one of a rooted tree

- A forest is a collection of disjoint trees.
- The connected components of a forest are trees



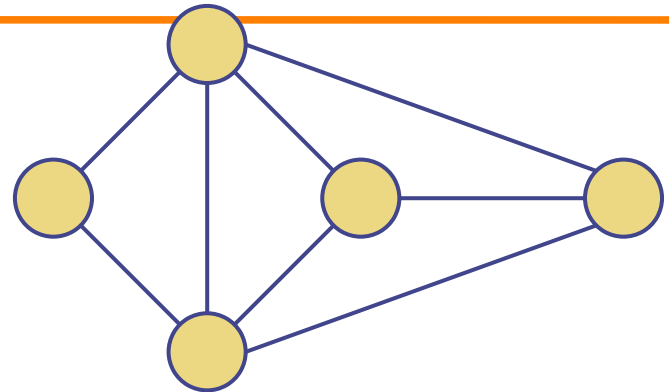
Tree



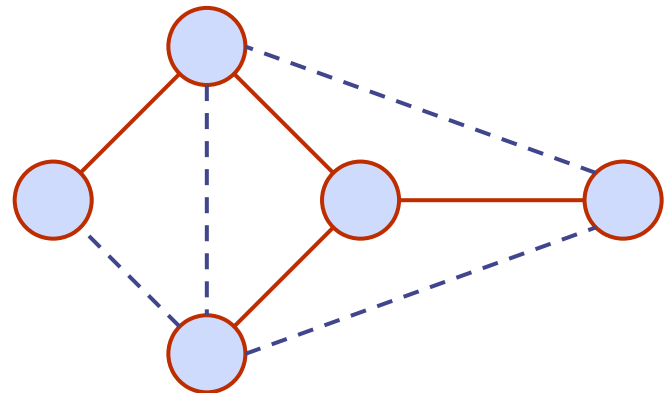
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest is a collection of trees that covers all the vertices (nodes) in a graph but avoids cycles.

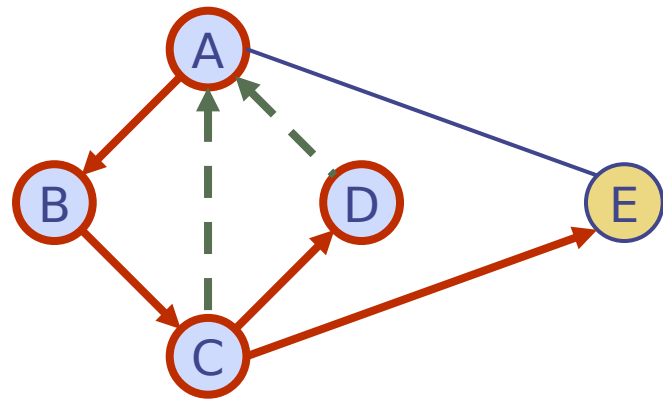


Graph



Spanning tree

Depth-First Search



Depth-First Search

Idea: Similar to pre-order traversal of a tree, visit all vertices reachable from an adjacent vertex before visiting another adjacent vertex.






Algorithm:

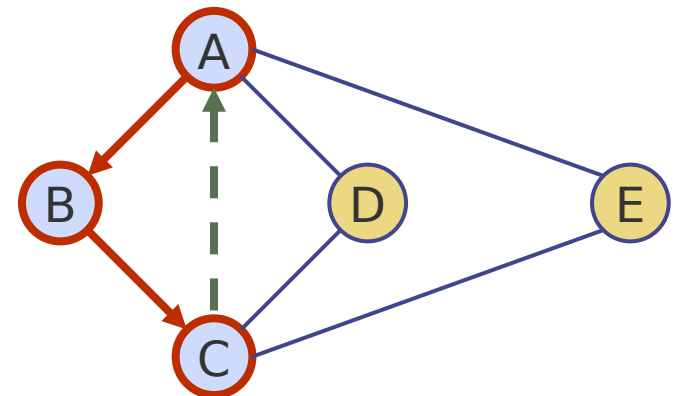
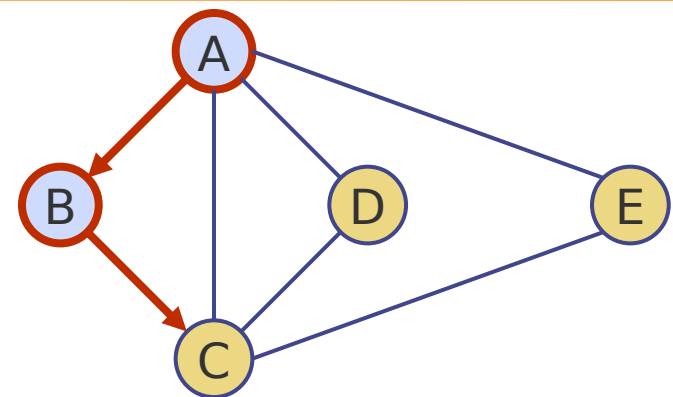
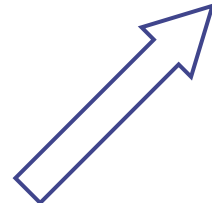
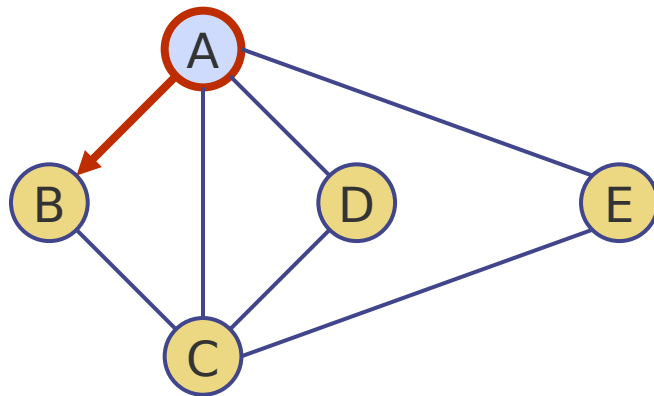
```
procedure DFS( $v, G$ ):  
    if  $v$  is not marked then begin  
        { visit  $v$  }  
        Mark  $v$ .  
        for each vertex  $w$  adjacent to  $v$  do DFS( $w, G$ )  
    end  
end
```

Initialize each vertex in the graph G to be "unmarked".

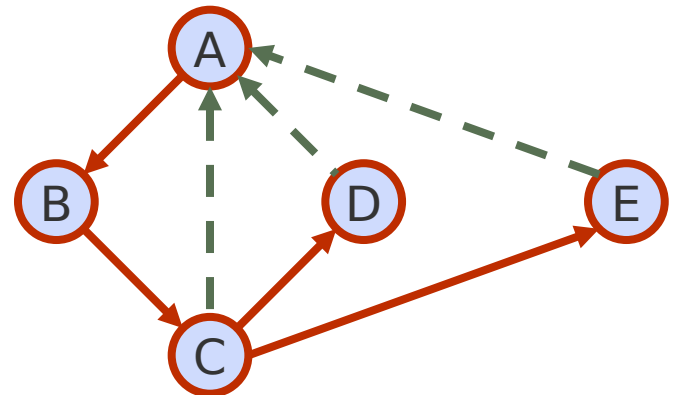
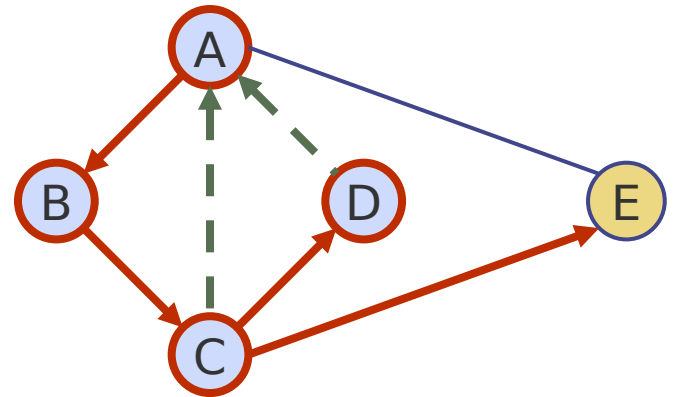
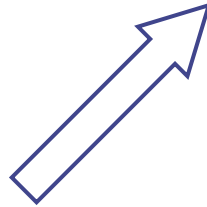
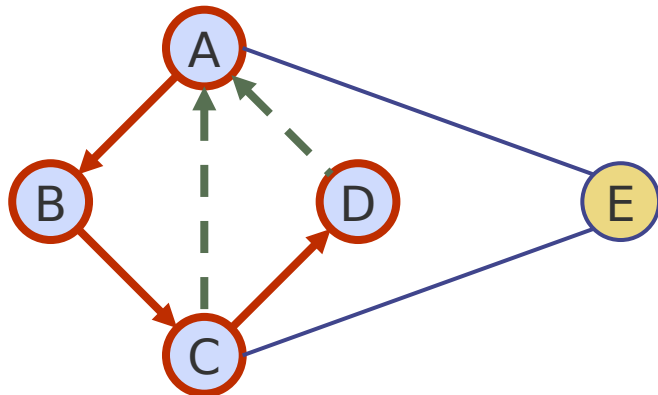
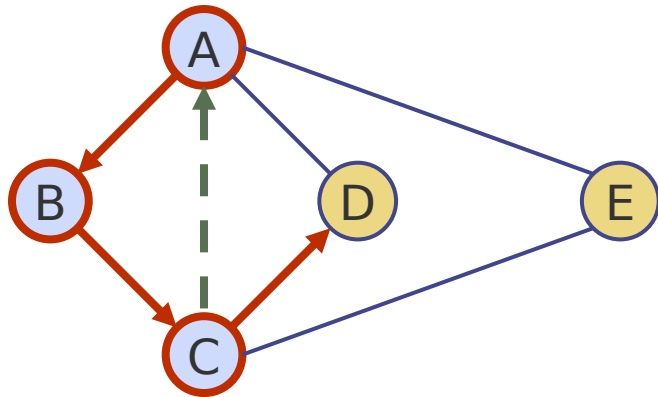
```
for each vertex  $v$  do DFS( $v, G$ )
```

Example

-  unexplored vertex
-  visited vertex
-  unexplored edge
-  discovery edge
-  back edge



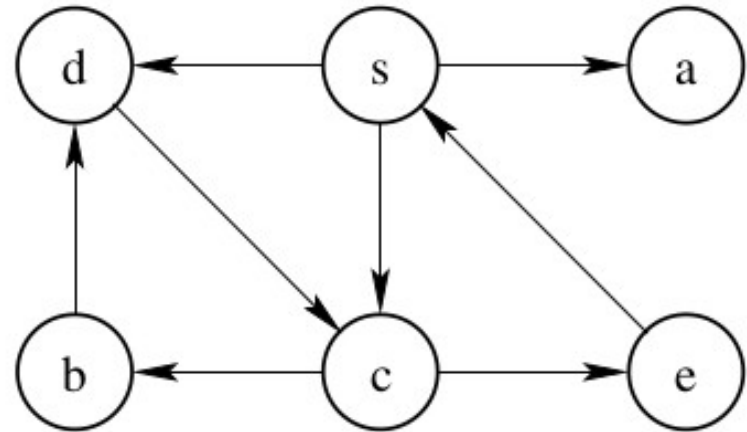
Example (cont.)



Exercise

Find the order in which nodes are visited starting from node s having adjacency list given below:

$adj(s) = [a, c, d]$,
 $adj(a) = []$,
 $adj(c) = [e, b]$,
 $adj(b) = [d]$,
 $adj(d) = [c]$,
 $adj(e) = [s]$.



DFS to classify edges in Graphs

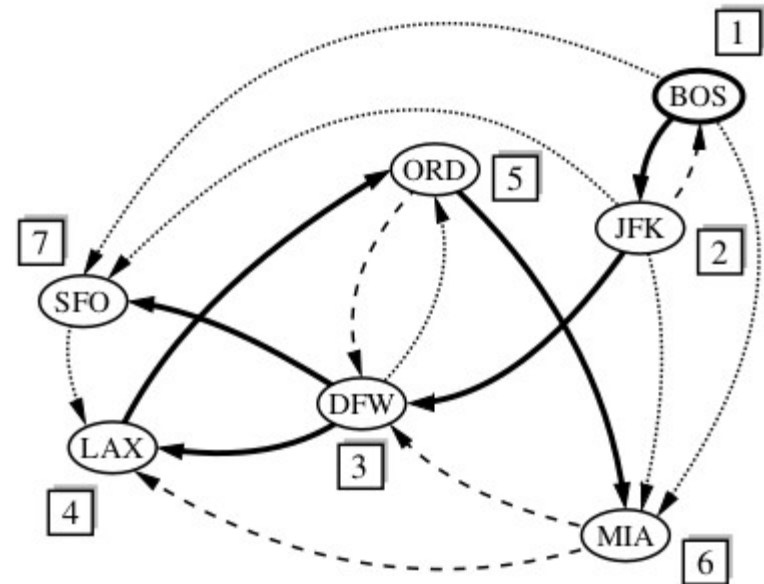
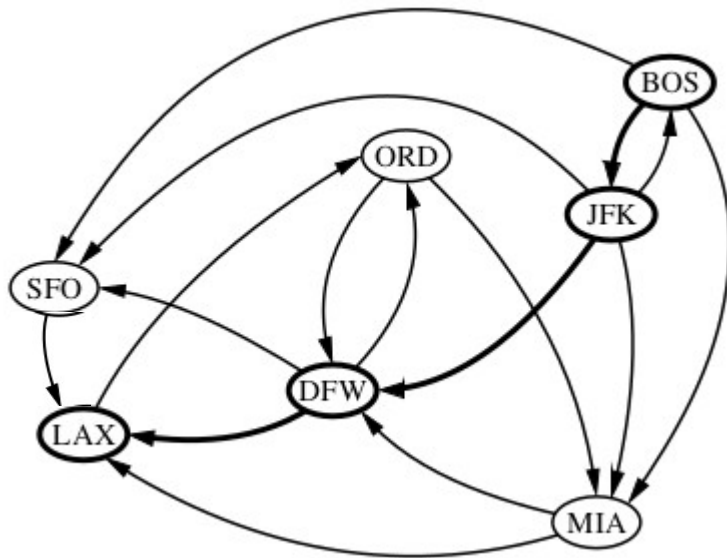
There can be 2 categories of edges: I) discovery edge or tree edge II) non-tree edges

In an undirected graph there is only one type of non-tree edge: **back edge**

In directed graph, there are 3 possible kinds of non-tree edges:

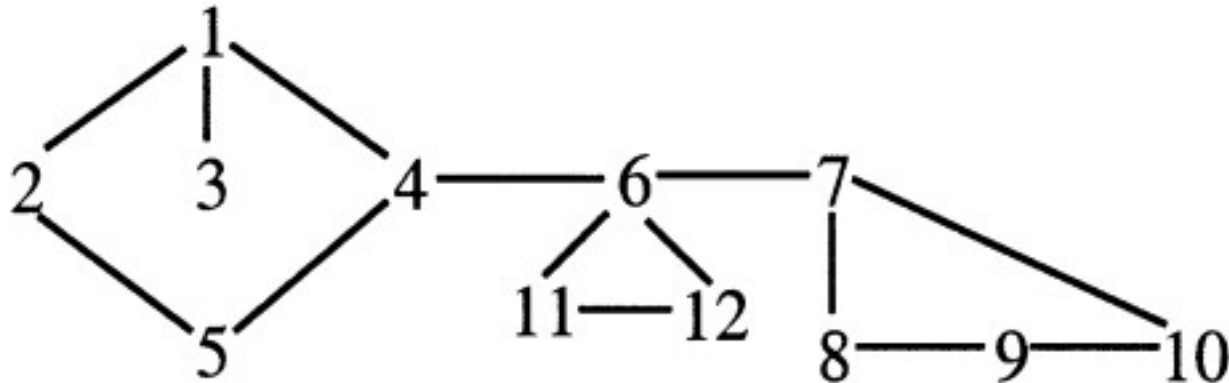
- a) **back edges**, which connect a vertex to an ancestor in the DFS tree
- b) **forward edges**, which connect a vertex to a descendant in the DFS tree
- c) **cross edges**, which connect a vertex to a vertex that is neither its ancestor nor its descendant

Find the various types of edges



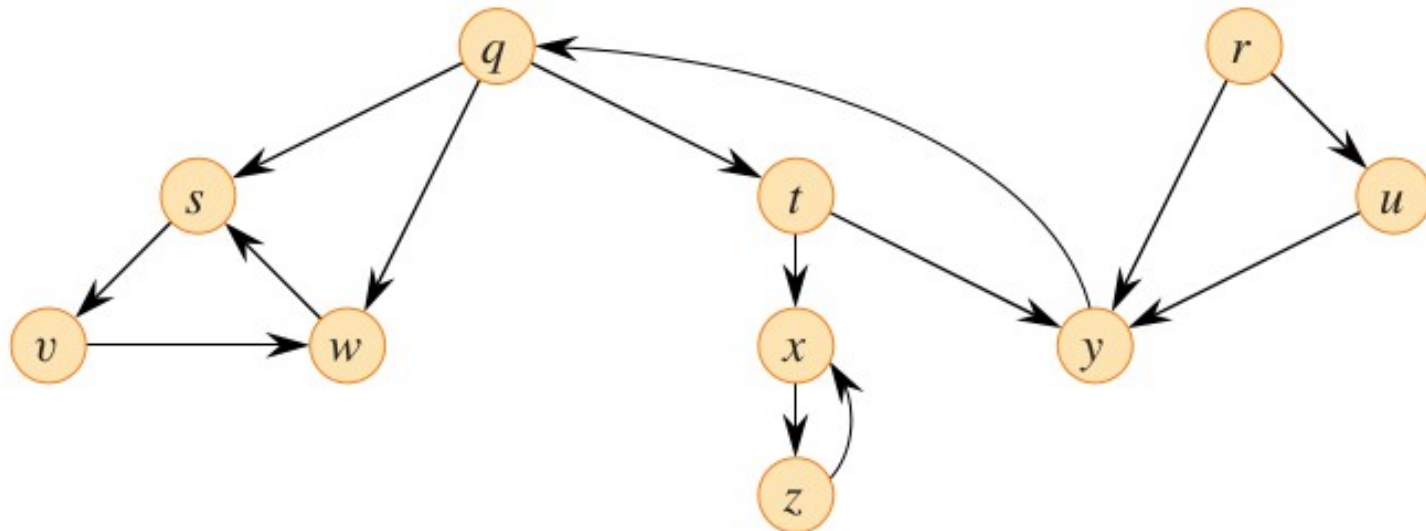
Exercise

In the graph below, build the adjacency list assuming it is ordered so that lower numbered vertices come before higher numbered ones. Give the order of nodes visited.



Exercise

In the graph below, build the adjacency list assuming it is ordered alphabetically. Give the order of nodes visited and show the classification of each edge.



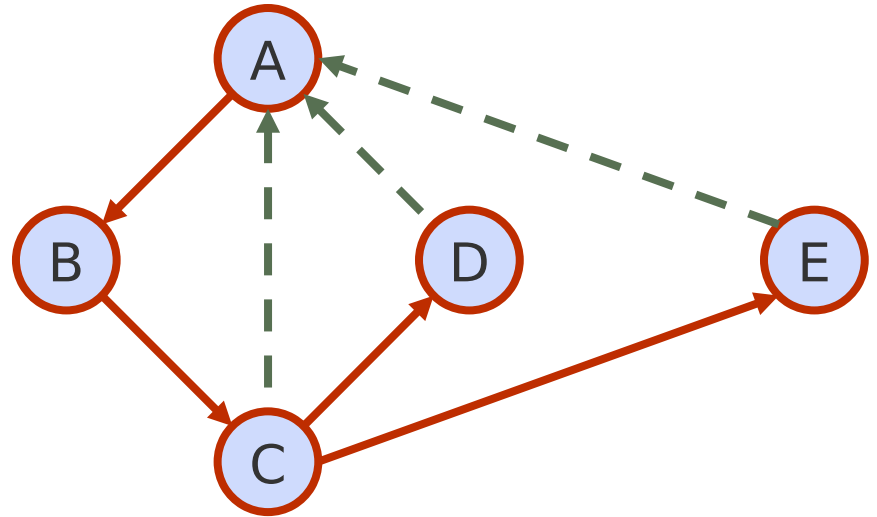
Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v



Analysis of DFS

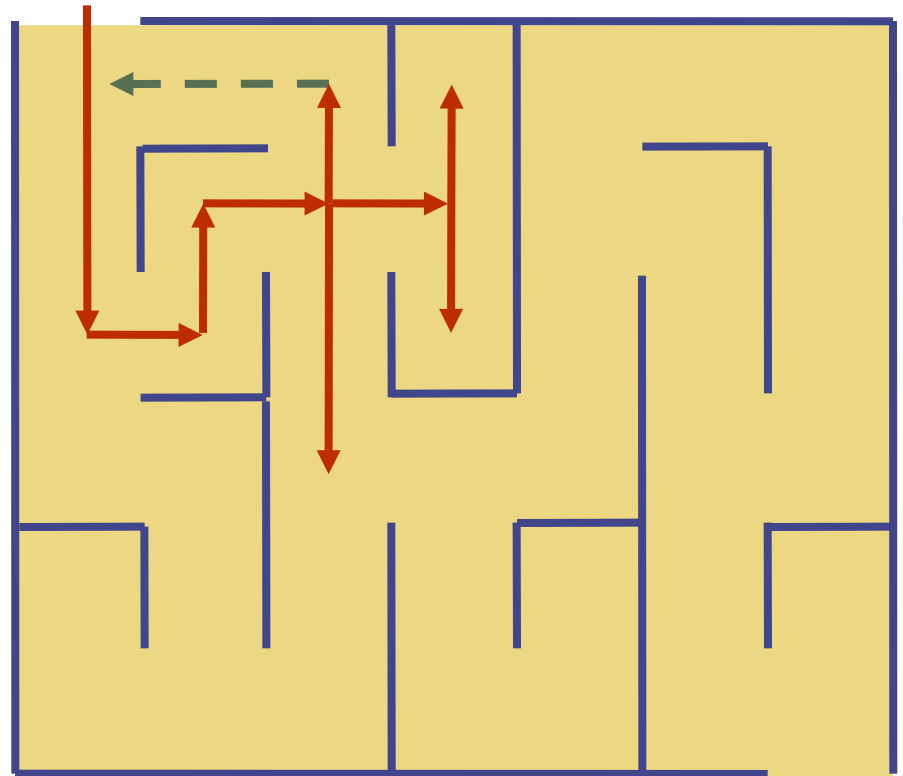
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as UNEXPLORED
 - once as VISITED
- Each edge is labeled twice
 - once as UNEXPLORED
 - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$

Depth-First Search - Applications

- Finding connected components of G
- Finding whether G is connected
- Topological Sorting: This sorting is useful in various scenarios, such as scheduling tasks with dependencies or resolving symbol dependencies in compilers.
- Finding paths between nodes:
- Detecting cycles in undirected graphs
- Other areas: Maze exploration, Network analysis, File system exploration: used to traverse directory structures in a file system

DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
 - We mark each intersection, corner and dead end (vertex) visited
 - We mark each passage (edge) traversed
 - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)





Breadth-First Search

Breadth-First Search

Idea: Similar to level-order traversal of a tree, visit all adjacent vertices before going deeper:

procedure BFS(v, G)

 Initialize a queue to contain v .

 Mark v .

while queue is not empty **do begin**

 DEQUEUE a vertex v

 {visit v }

for each vertex w adjacent to v **do begin**

if (w is unmarked) **then begin**

 Mark w

 ENQUEUE(w)

end

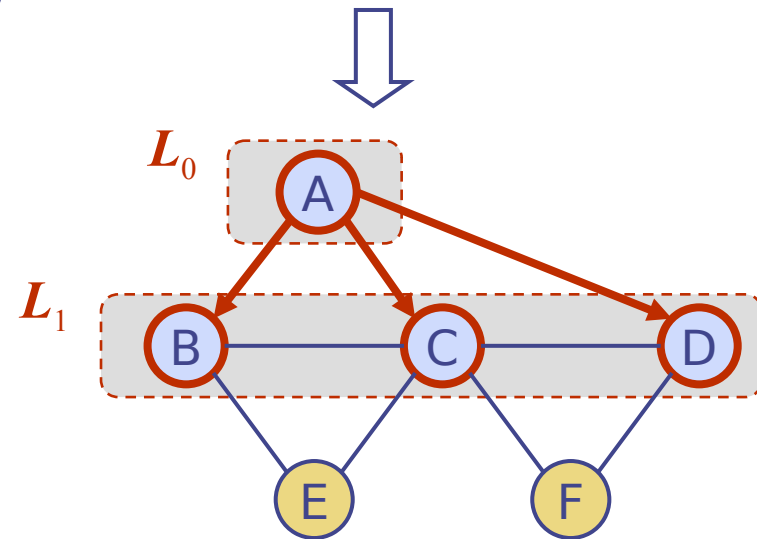
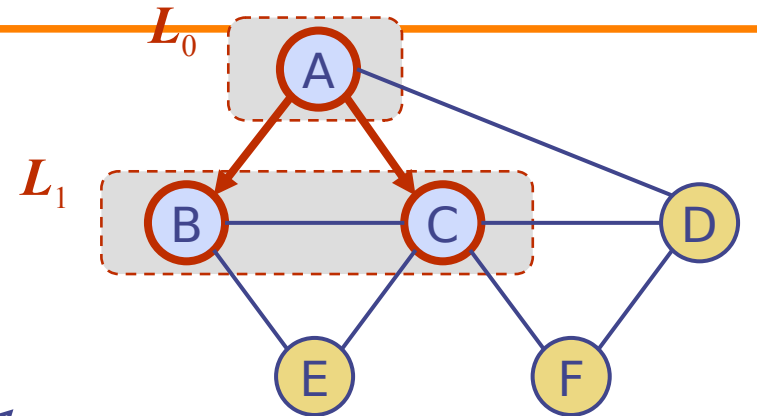
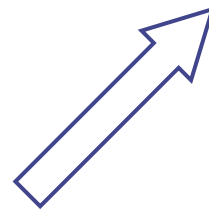
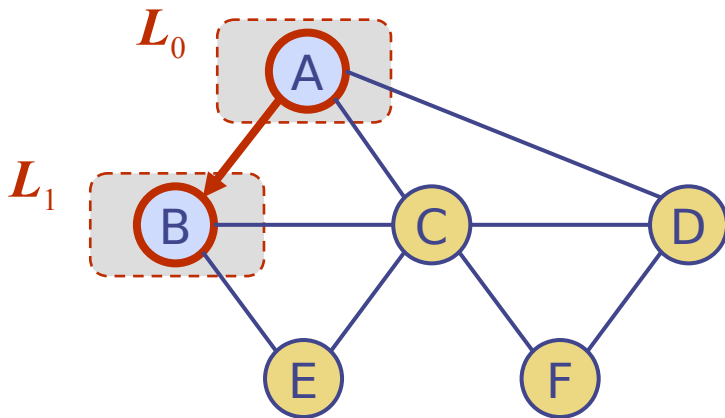
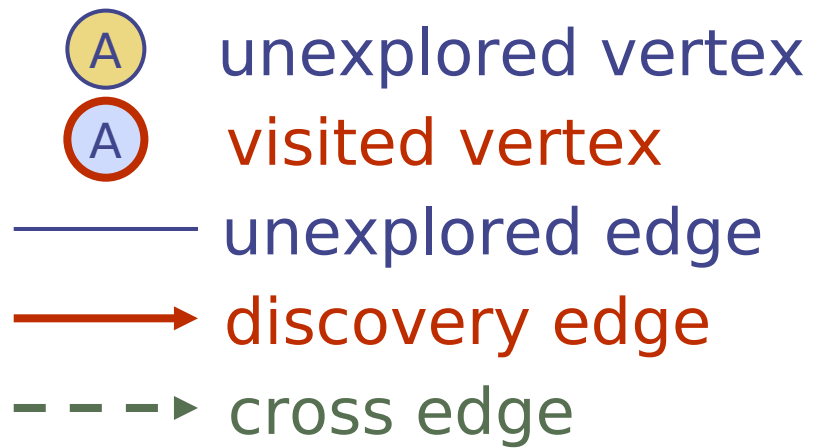
end

end

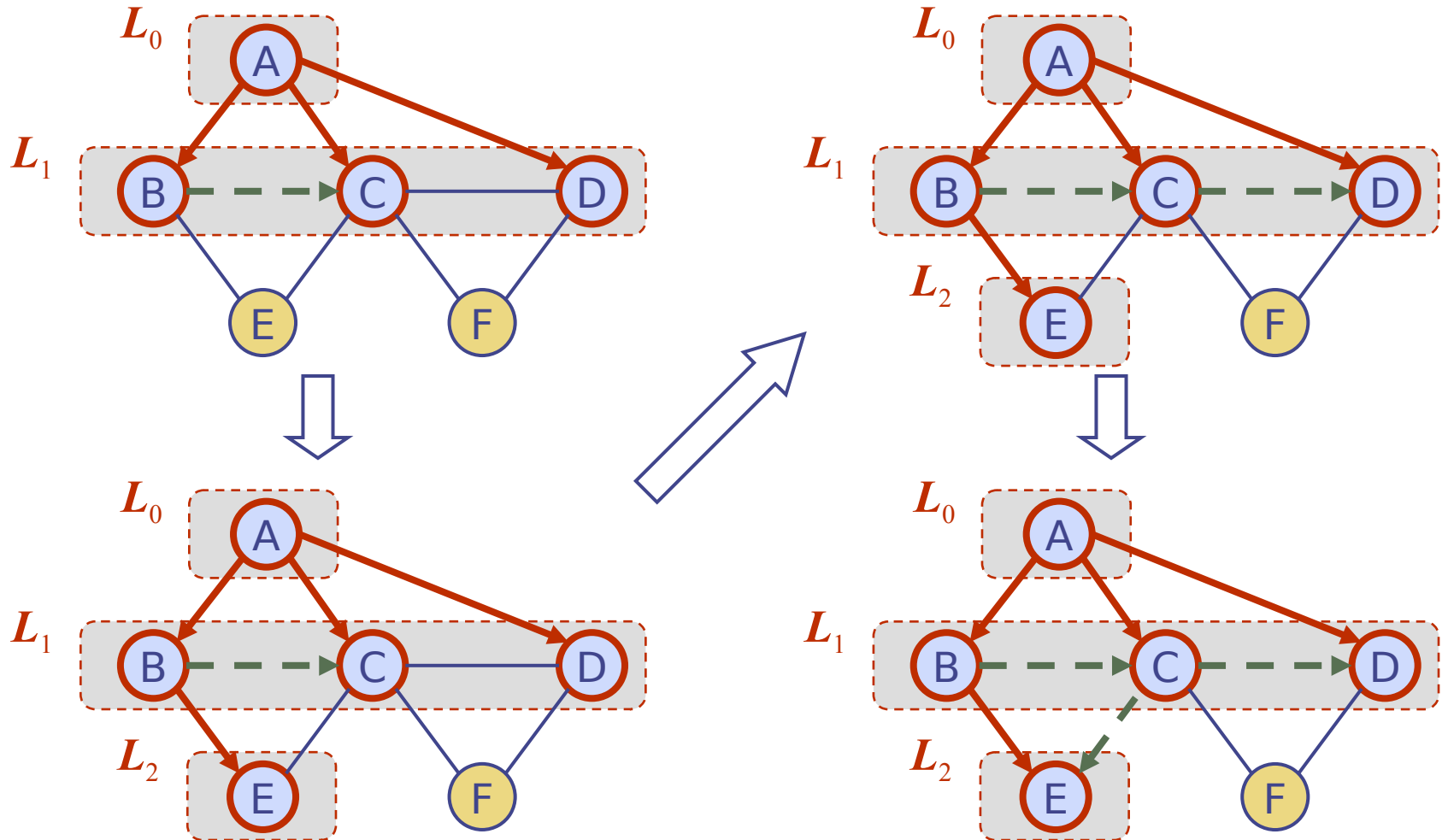
Initialize each vertex in the graph G to be "unmarked".

BFS(v, G) // Call the function

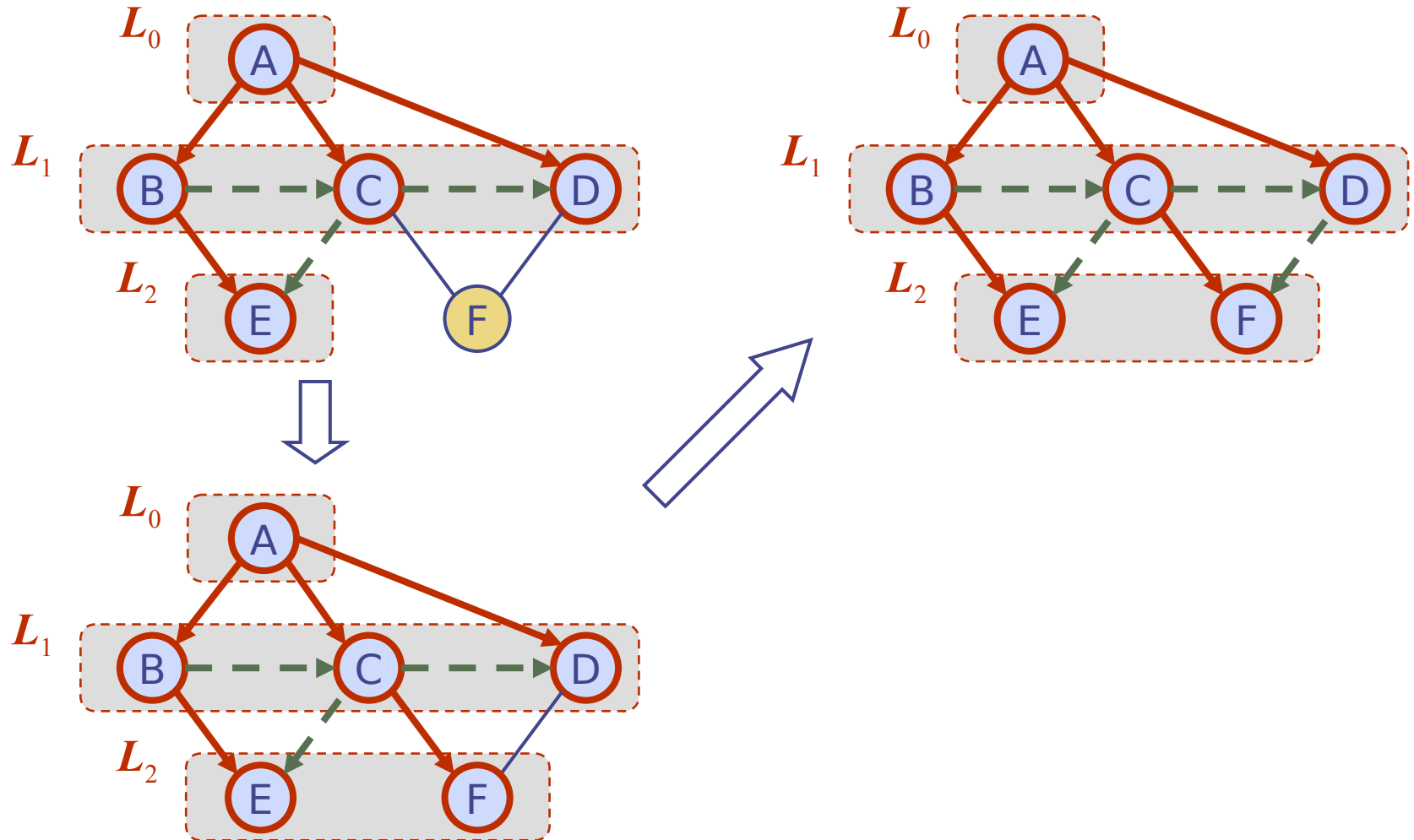
Example



Example (cont.)



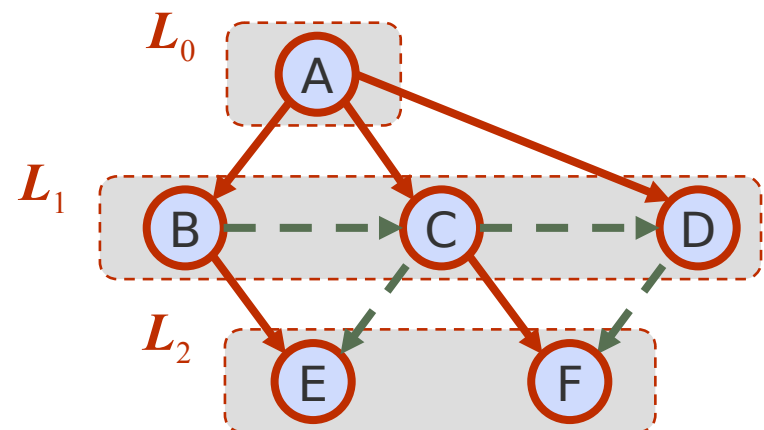
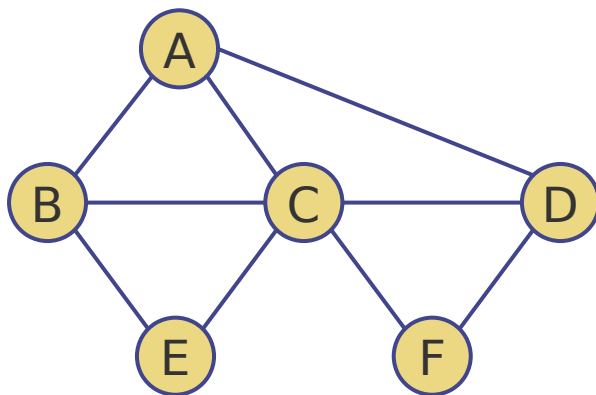
Example (cont.)



Properties

Proposition 14.16: *Let G be an undirected or directed graph on which a BFS traversal starting at vertex s has been performed. Then*

- *The traversal visits all vertices of G that are reachable from s .*
- *For each vertex v at level i , the path of the BFS tree T between s and v has i edges, and any other path of G from s to v has at least i edges.*
- *If (u, v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the level number of u .*



Analysis

Setting/getting a vertex/edge label takes $O(1)$ time

Each vertex is labeled twice

- once as UNEXPLORED
- once as VISITED

Each edge is labeled twice

- once as UNEXPLORED
- once as DISCOVERY or CROSS

Each vertex is inserted once into a sequence L_i

Method incidentEdges is called once for each vertex

BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

- Recall that $\sum_v \deg(v) = 2m$

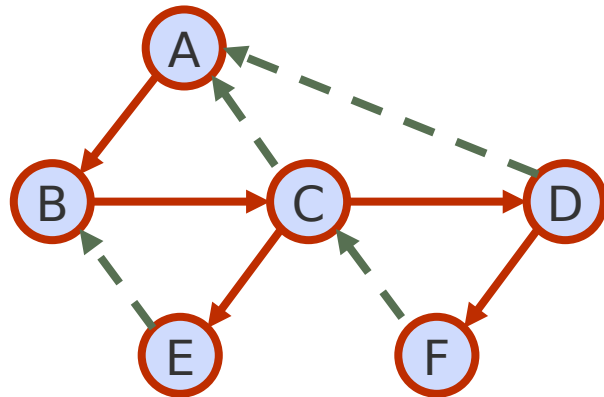
Applications

Using the **template method pattern**, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time

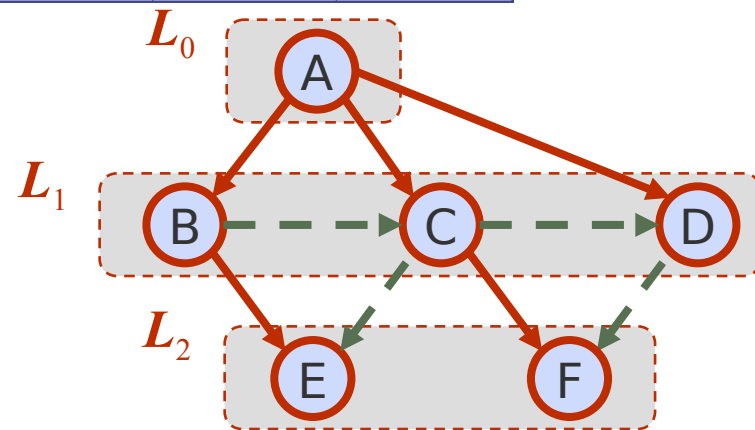
- Compute the connected components of G
- Compute a spanning forest of G
- Find a simple cycle in G , or report that G is a forest
- Given two vertices of G , find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



DFS

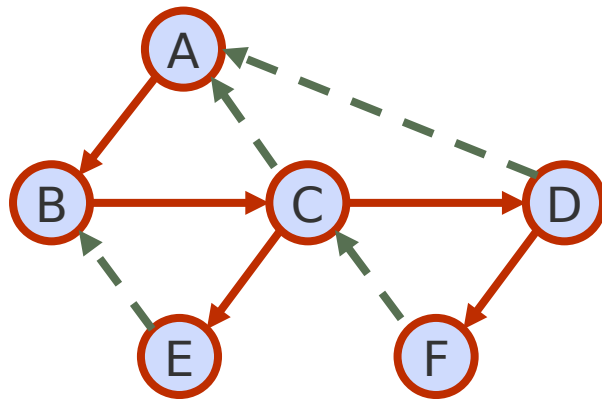


BFS

DFS vs. BFS (cont.)

Back edge (v, w)

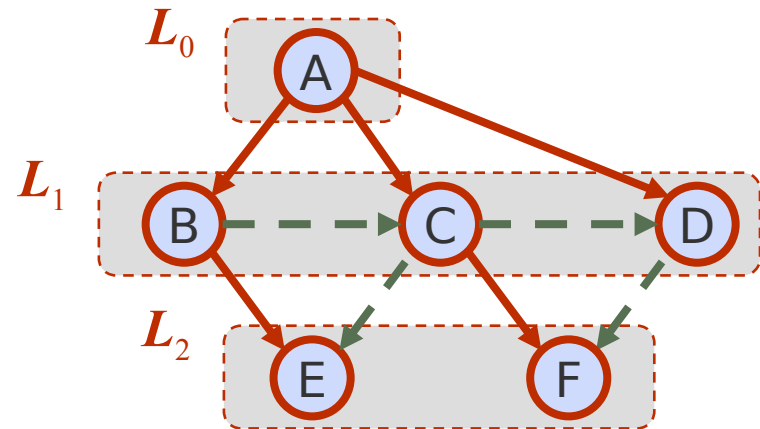
- w is an ancestor of v in the tree of discovery edges



DFS

Cross edge (v, w)

- w is in the same level as v or in the next level



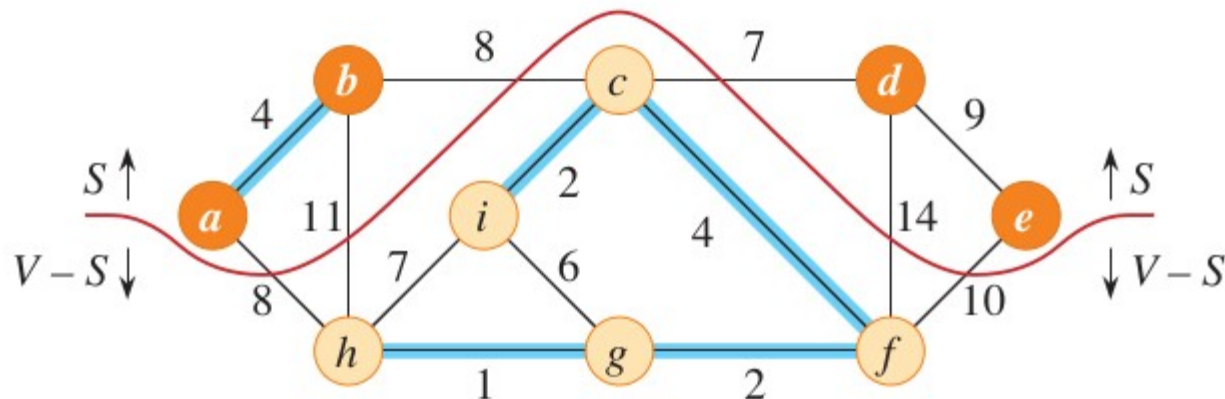
BFS

Minimum spanning trees

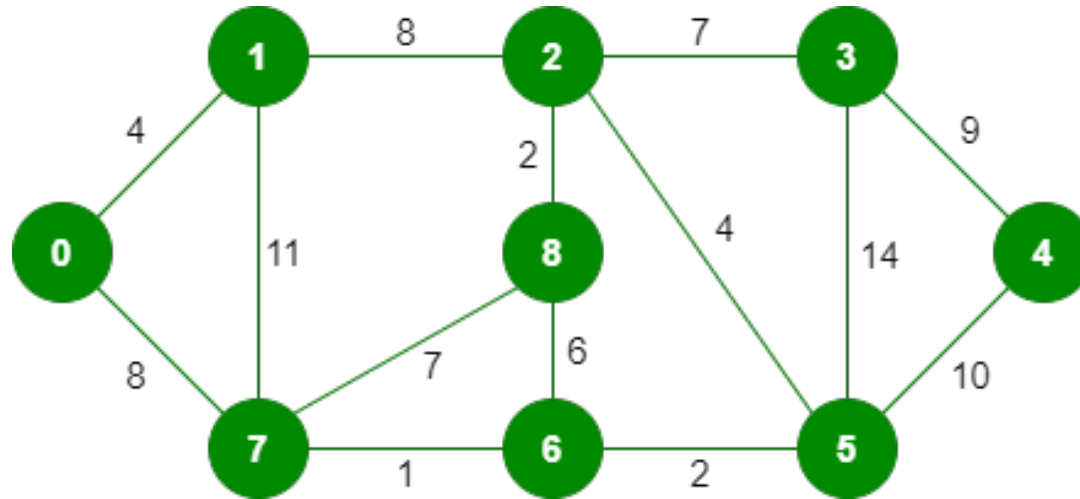
Problem Definition

Given an undirected, weighted graph G , we are interested in finding a tree T that contains all the vertices in G and minimizes the sum

$$w(T) = \sum_{(u,v) \text{ in } T} w(u,v).$$



Minimum spanning trees



Kruskal's algorithm

MST-KRUSKAL(G, w)

```
1   $A = \emptyset$ 
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  create a single list of the edges in  $G.E$ 
5  sort the list of edges into monotonically increasing order by weight  $w$ 
6  for each edge  $(u, v)$  taken from the sorted list in order
7      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
8           $A = A \cup \{(u, v)\}$ 
9          UNION( $u, v$ )
10 return  $A$ 
```

Prim's algorithm

MST-PRIM(G, w, r)

```
1  for each vertex  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = \emptyset$ 
6  for each vertex  $u \in G.V$ 
7      INSERT( $Q, u$ )
8  while  $Q \neq \emptyset$ 
9       $u = \text{EXTRACT-MIN}(Q)$            // add  $u$  to the tree
10     for each vertex  $v$  in  $G.Adj[u]$  // update keys of  $u$ 's non-tree neighbors
11         if  $v \in Q$  and  $w(u, v) < v.key$ 
12              $v.\pi = u$ 
13              $v.key = w(u, v)$ 
14             DECREASE-KEY( $Q, v, w(u, v)$ )
```

Shortest path algorithms

Given a weighted, digraph $G = (V, E)$, with a weight function $w : E \rightarrow \mathbb{R}$ mapping edges to real-valued weights. The weight $w(p)$ of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

We define the *shortest-path weight* $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

A *shortest path* from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

Types of Shortest Path Algorithms:

1. Single-Source shortest paths
2. Single-destination shortest-paths
3. Single-pair shortest path
4. All-pairs shortest paths

Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$, but it requires nonnegative weights on all edges: $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm as generalizing breadth first search to weighted graphs

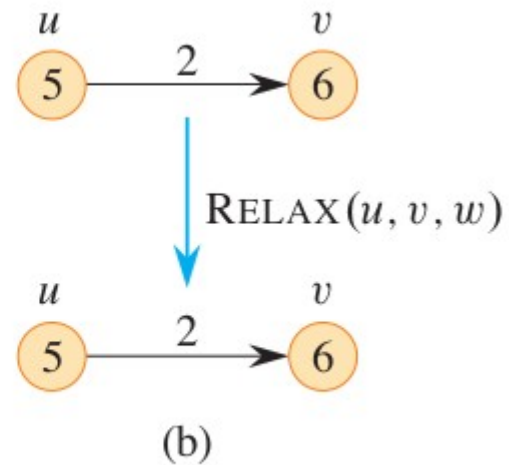
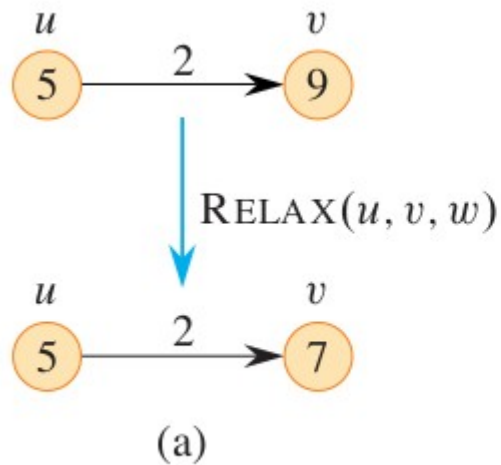
INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex  $v \in G.V$ 
2       $v.d = \infty$ 
3       $v.\pi = \text{NIL}$ 
4   $s.d = 0$ 
```

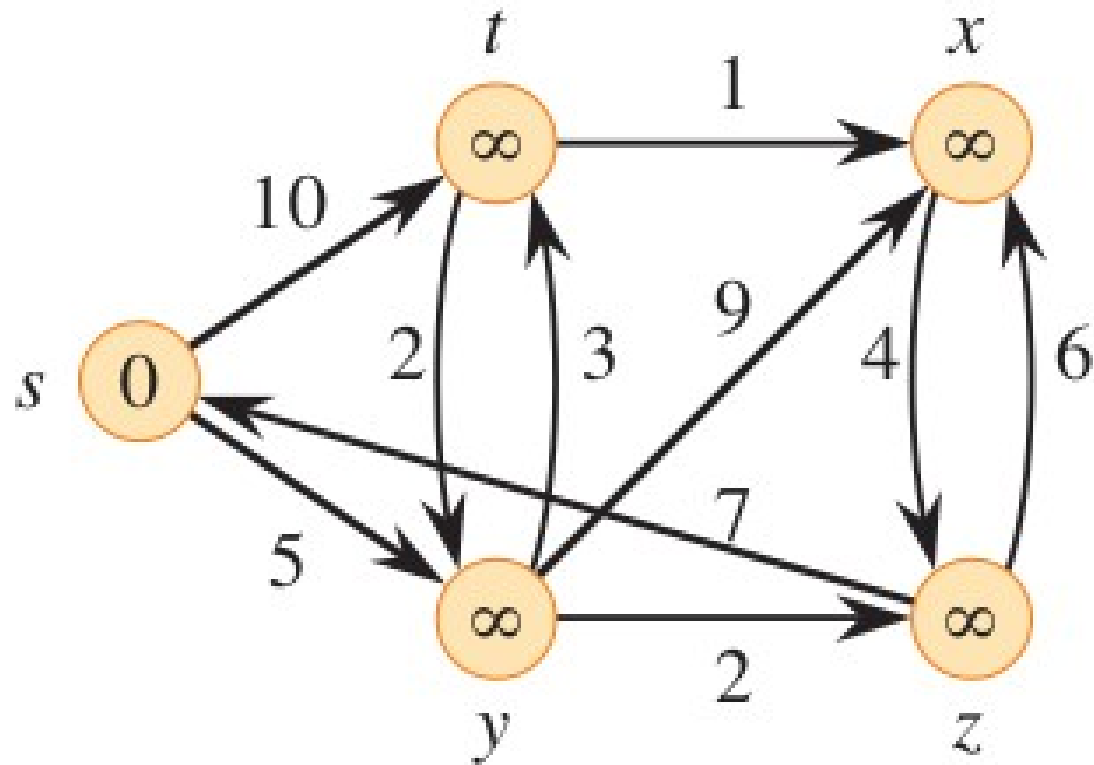
Relax an edge

RELAX(u, v, w)

- 1 **if** $v.d > u.d + w(u, v)$
- 2 $v.d = u.d + w(u, v)$
- 3 $v.\pi = u$



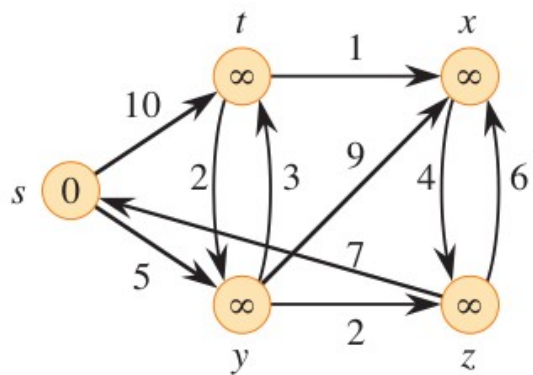
Example



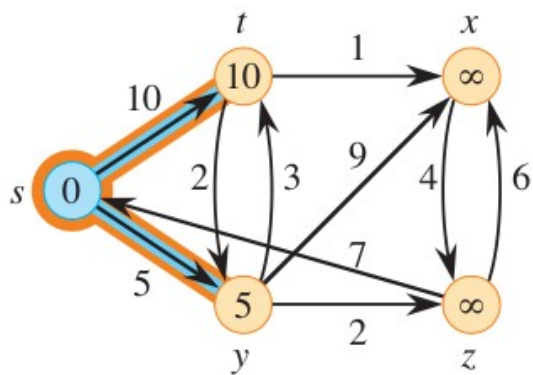
Dijkstra's algorithm

DIJKSTRA(G, w, s)

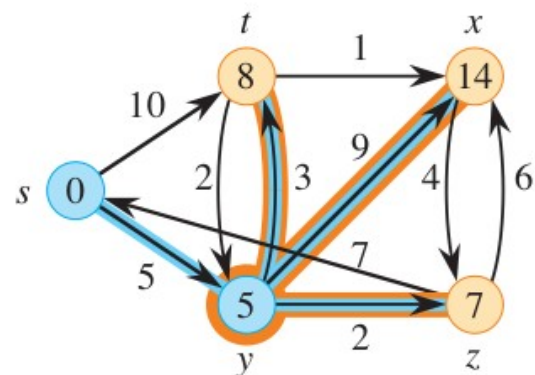
```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = \emptyset$ 
4  for each vertex  $u \in G.V$ 
5      INSERT( $Q, u$ )
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8       $S = S \cup \{u\}$ 
9      for each vertex  $v$  in  $G.Adj[u]$ 
10         RELAX( $u, v, w$ )
11         if the call of RELAX decreased  $v.d$ 
12             DECREASE-KEY( $Q, v, v.d$ )
```



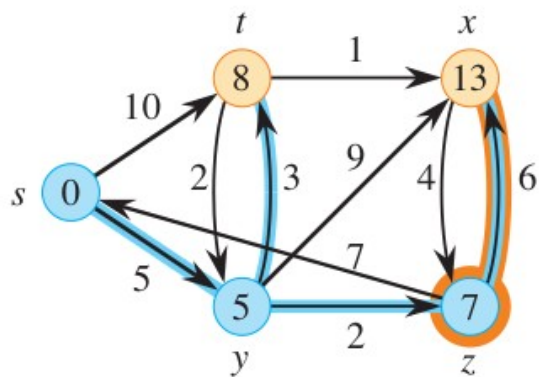
(a)



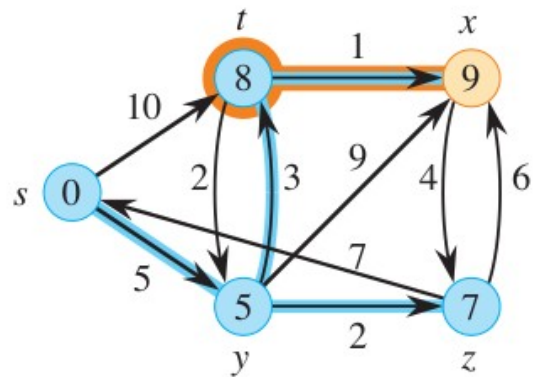
(b)



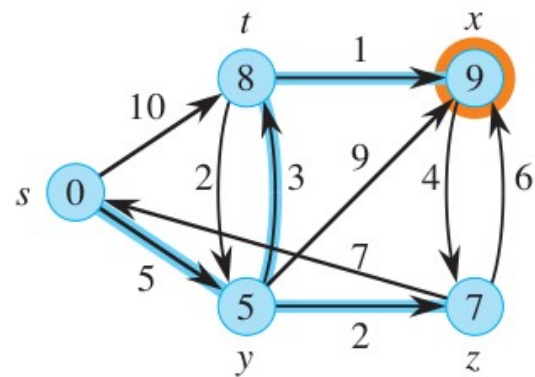
(c)



(d)



(e)



(f)

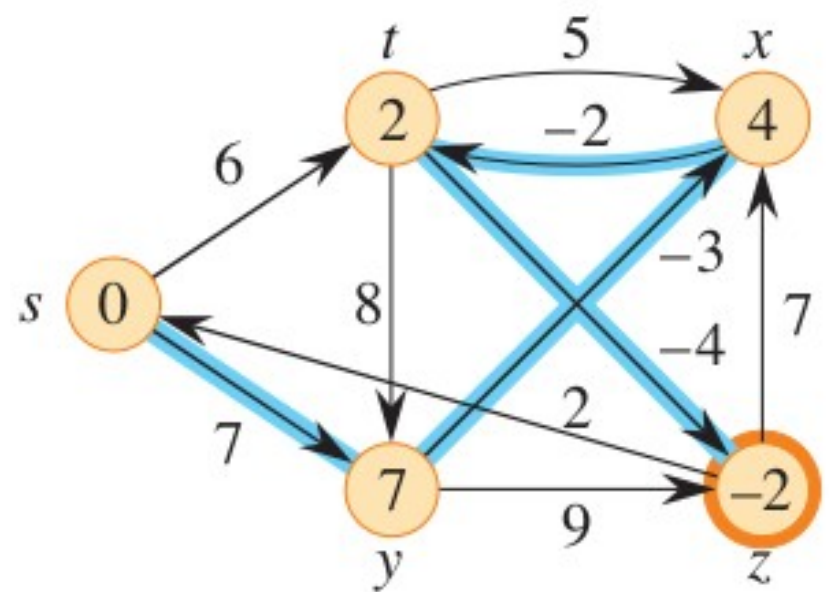
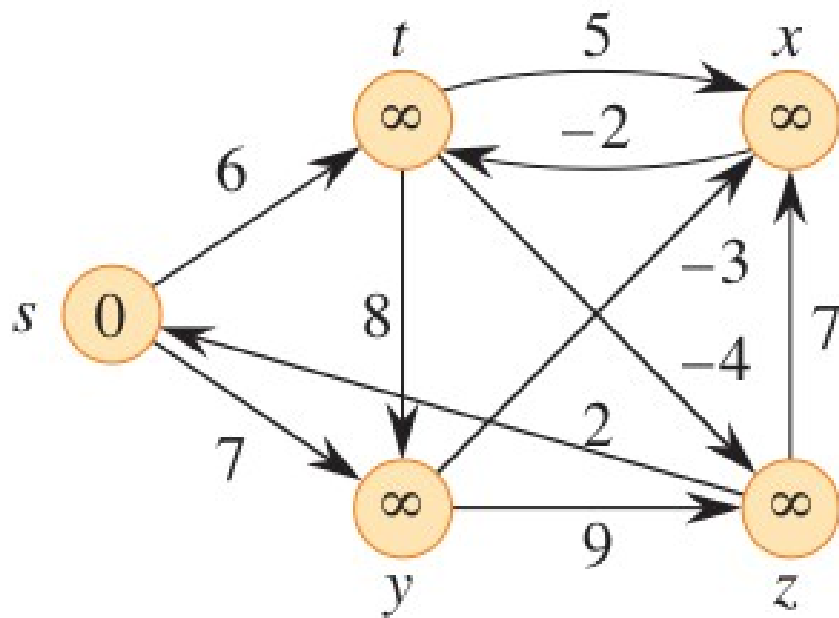
Bellman-Ford algorithm

This solves the single-source shortest-paths problem in the general case in which edge weights may be negative.

It can also detect the presence of negative cycles in the graph

```
BELLMAN-FORD( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3      for each edge  $(u, v) \in G.E$ 
4          RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6      if  $v.d > u.d + w(u, v)$ 
7          return FALSE
8  return TRUE
```

Example



References

Data Structures and Algorithms in Python

Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms

Leiserson, Stein, Rivest, Cormen

Algorithms, 4th Edition

Robert Sedgewick and Kevin Wayne

Few Images from the internet