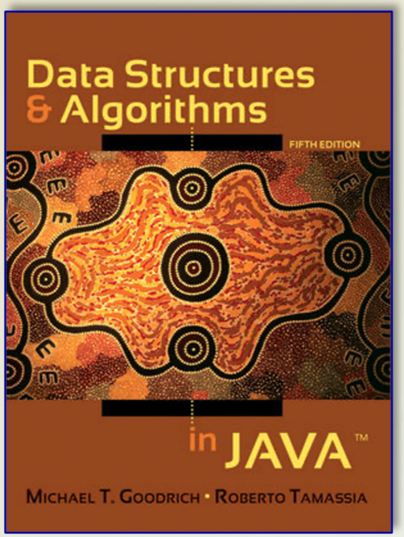
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich
Roberto Tamassia



Chapter 13: Graph Algorithms

CPSC 3200

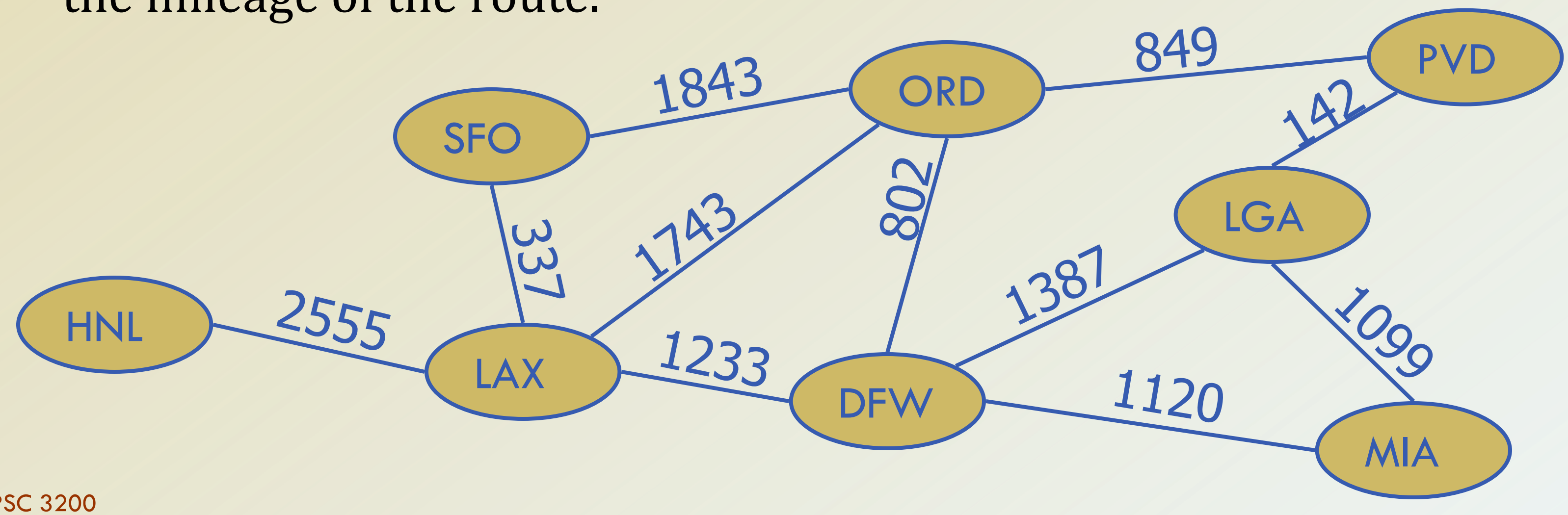
Algorithm Analysis and Advanced Data Structure

Chapter Topics

- Graphs.
- Data Structure for Graphs.
- Graph Traversals.
- Directed Graphs.
- Shortest Paths.

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.
- **Example:**
  - A vertex represents an airport and stores the three-letter airport code.
  - An edge represents a flight route between two airports and stores the mileage of the route.

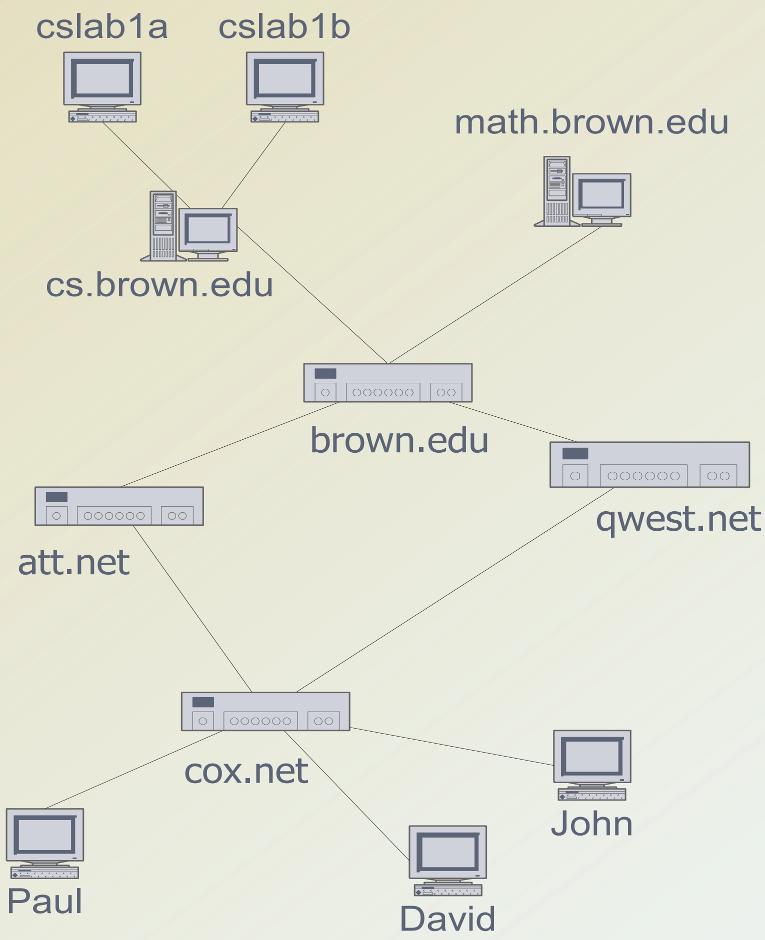


Edge Types

- **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

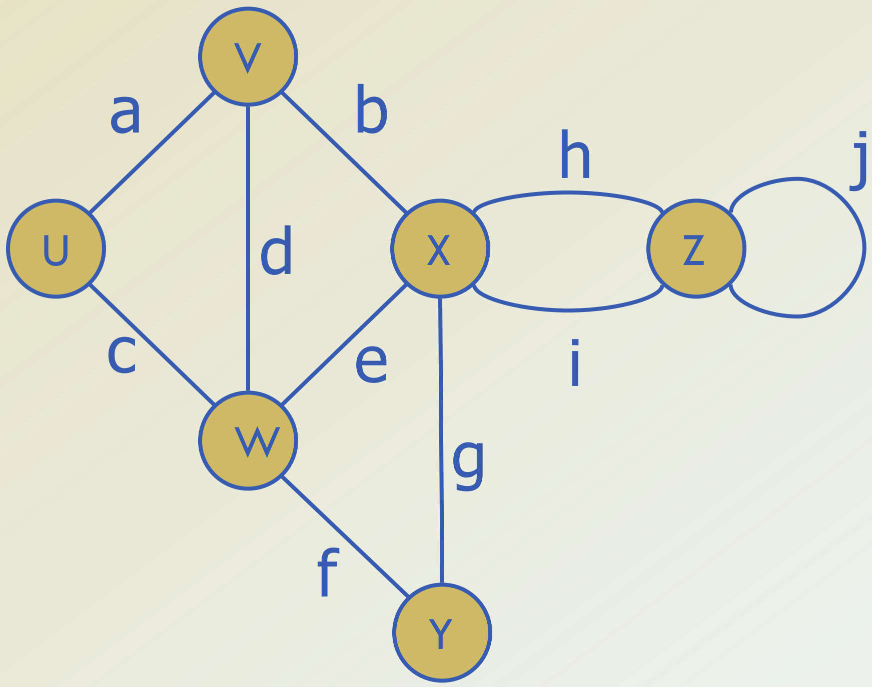
Applications

- **Electronic circuits**
  - Printed circuit board
  - Integrated circuit
- **Transportation networks**
  - Highway network
  - Flight network
- **Computer networks**
  - Local area network
  - Internet
  - Web
- **Databases**
  - Entity-relationship diagram



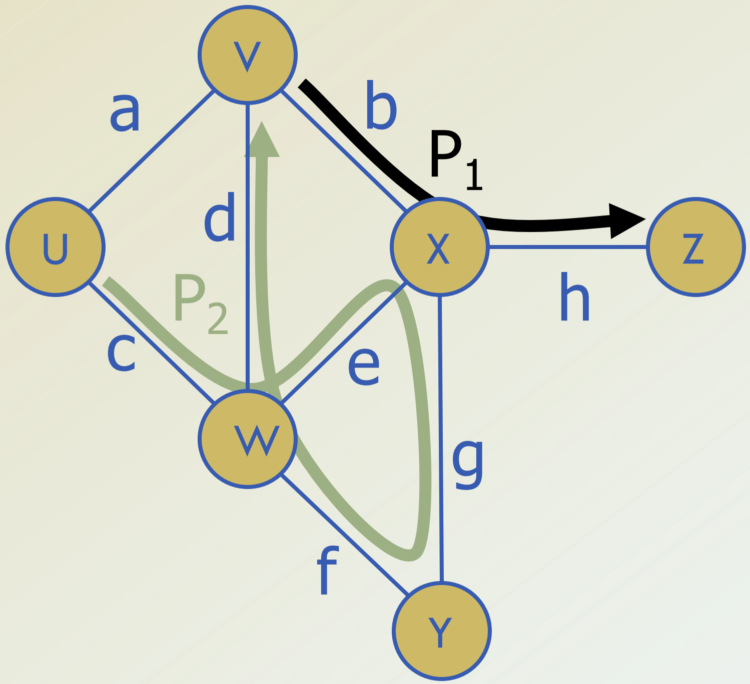
Terminology

- **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



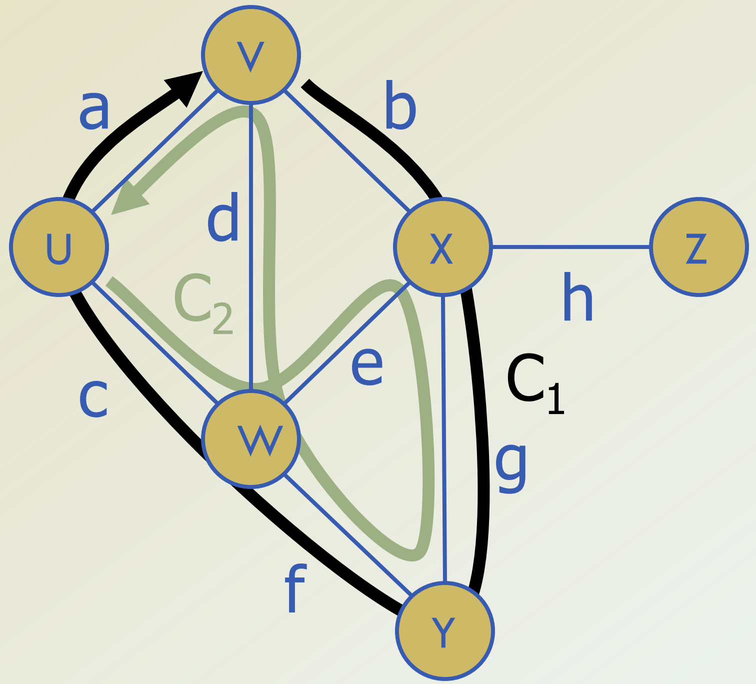
Terminology (cont.)

- **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

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

Proof: each edge is counted twice.

Property 2

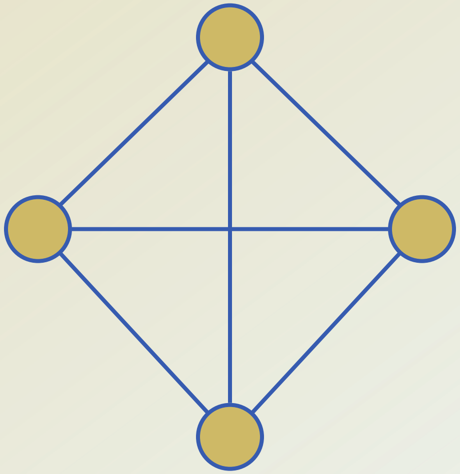
In an undirected graph with no self-loops and no multiple edges

$m \le n\,(n-1)/2$

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

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

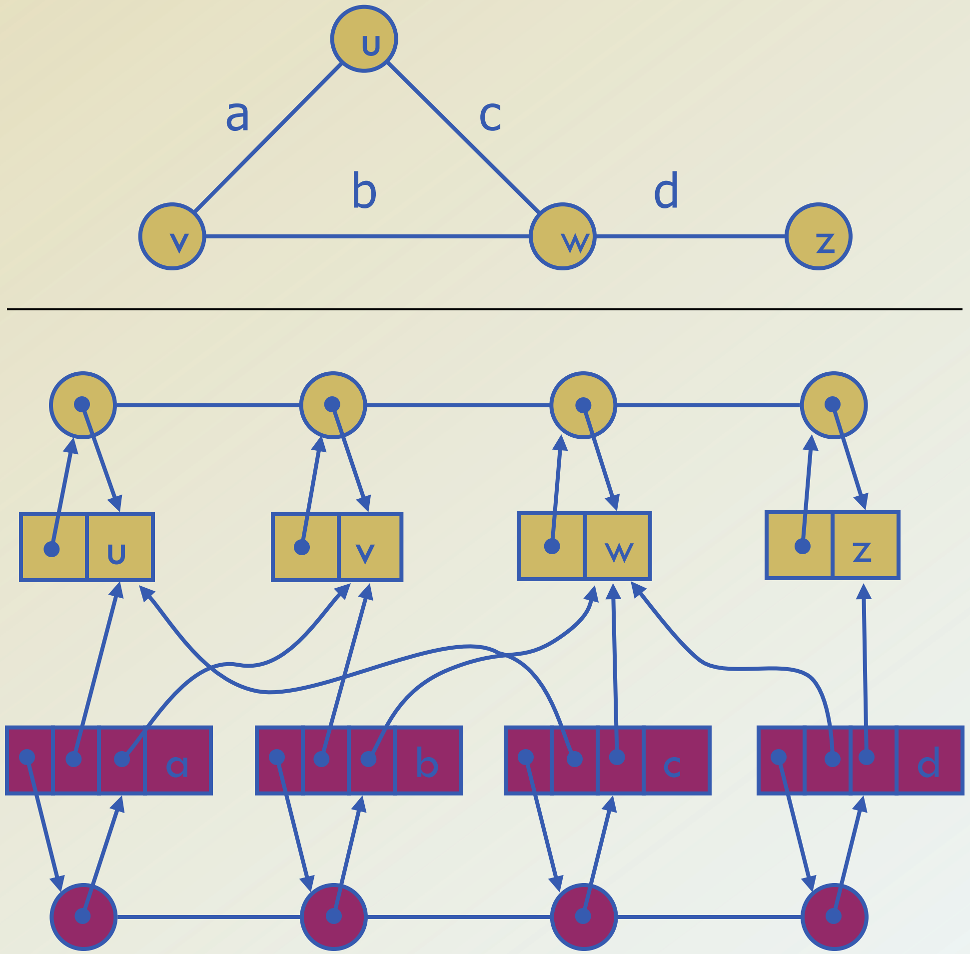
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Main Methods of the Graph ADT

- **Vertices and edges:**
  - are positions
  - store elements
- **Accessor methods:**
  - **endVertices(e):** an array of the two endvertices of e.
  - **opposite(v, e):** the vertex opposite of v on e.
  - **areAdjacent(v, w):** true iff v and w are adjacent.
  - **replace(v, x):** replace element at vertex v with x.
  - **replace(e, x):** replace element at edge e with x.
- **Update methods:**
  - **insertVertex(o):** insert a vertex storing element o.
  - **insertEdge(v, w, o):** insert an edge (v,w) storing element o.
  - **removeVertex(v):** remove vertex v (and its incident edges).
  - **removeEdge(e):** remove edge e.
- **Iterable collection methods:**
  - **incidentEdges(v):** edges incident to v.
  - **vertices( ):** all vertices in the graph.
  - **edges( ):** all edges in the graph.

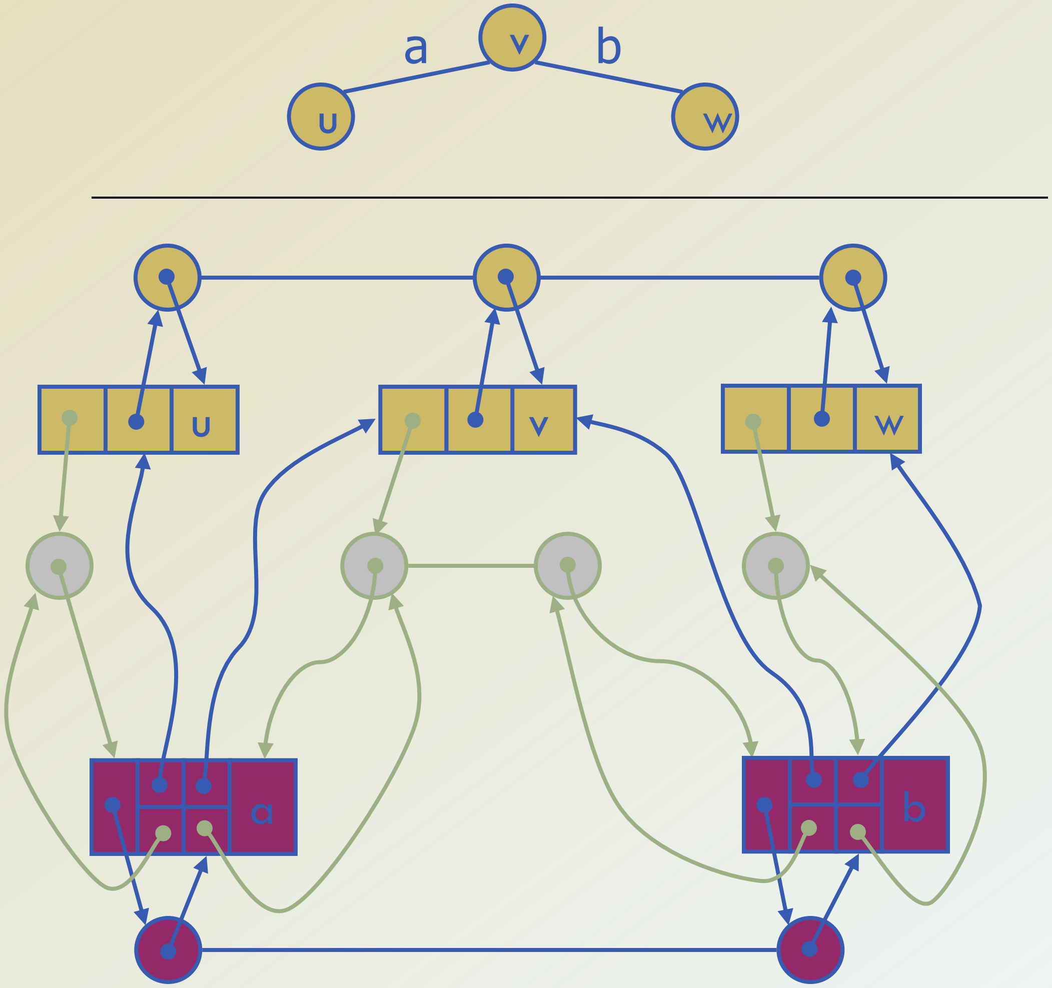
Edge List Structure

- **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.



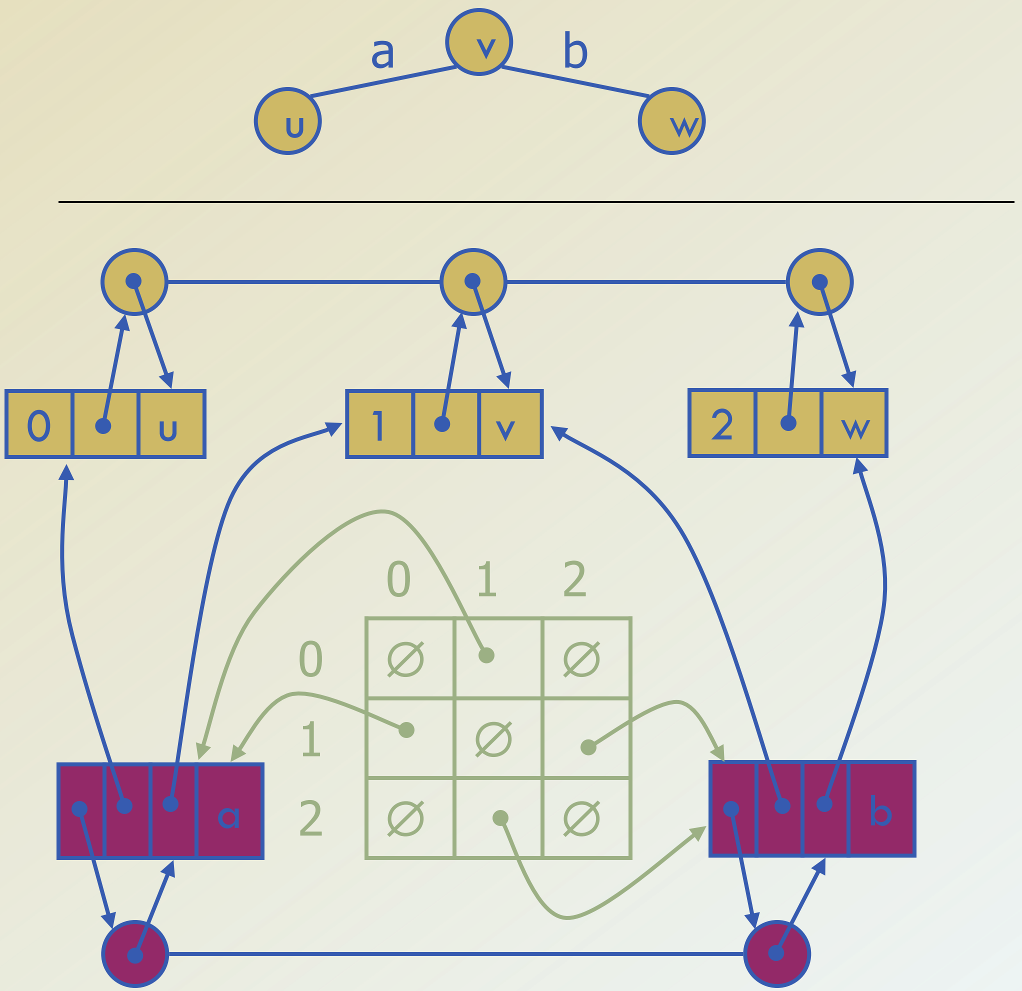
Adjacency List Structure

- Edge 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.



Adjacency Matrix Structure

- Edge list structure.
- Augmented vertex objects
  - Integer key (index) associated with vertex.
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices.
  - Null for non nonadjacent vertices.
- The “old fashioned” version just has 0 for no edge and 1 for edge.

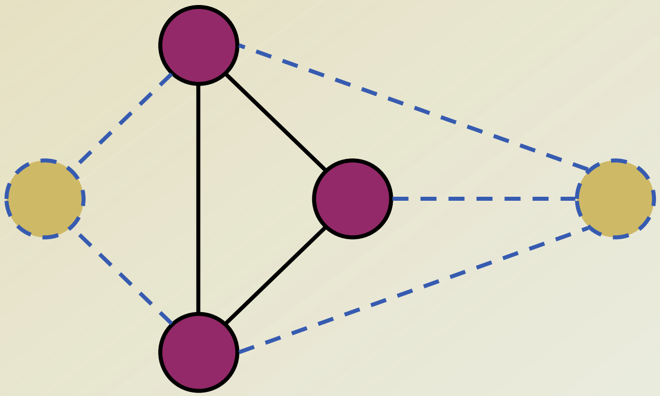


Performance

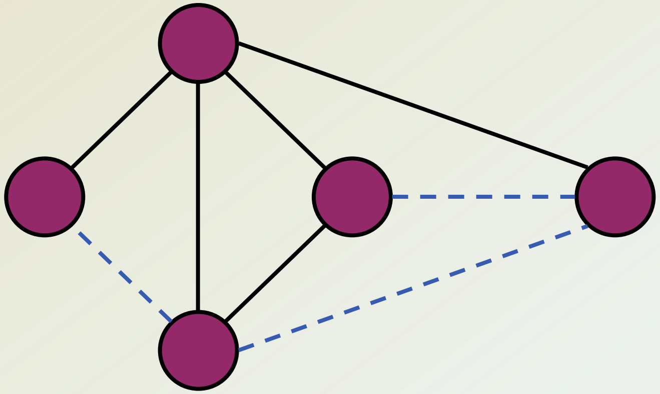
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

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.



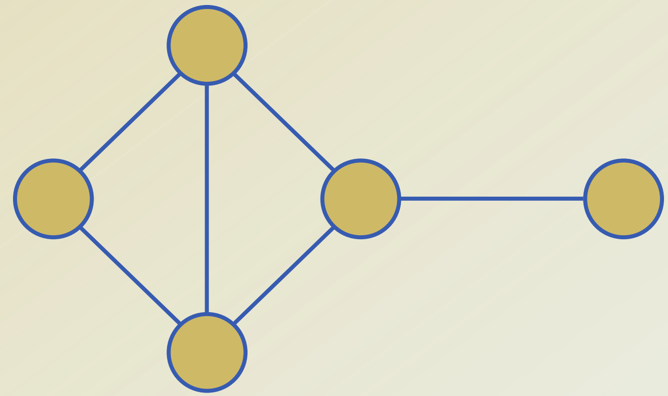
Subgraph



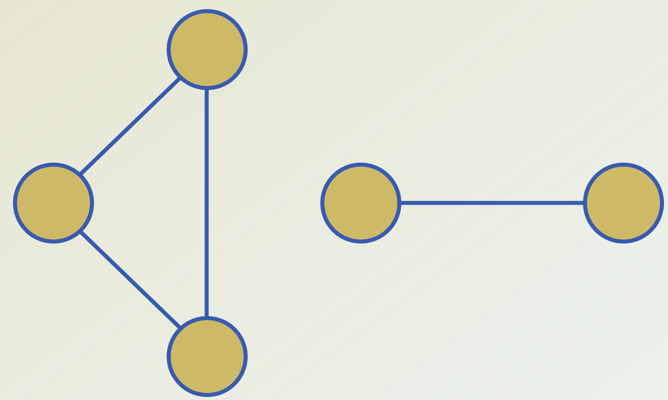
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph G is a maximal connected subgraph of G.



Connected graph

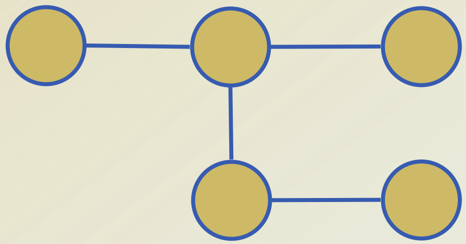


Non connected graph with two connected components

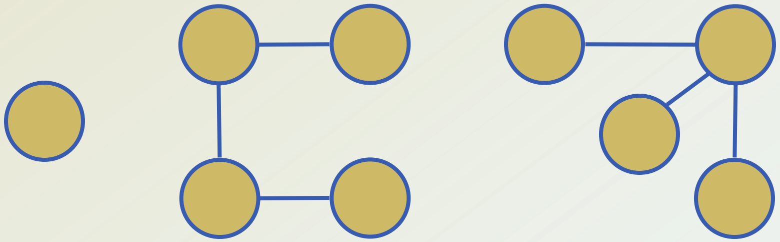
Trees and Forests

- A (free) 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 an undirected graph without cycles.
- The connected components of a forest are trees



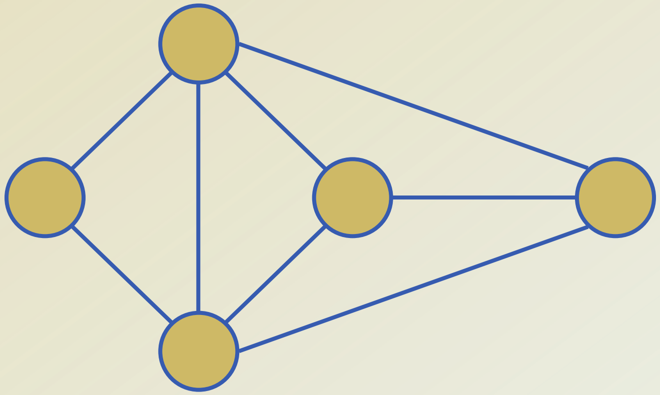
Tree



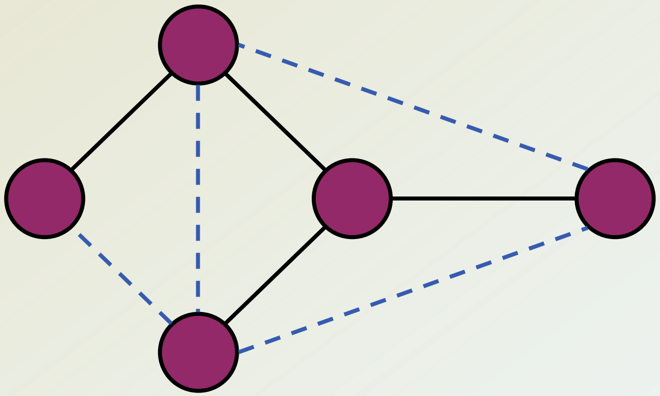
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
- A spanning tree is not unique unless the graph is a tree.
- Spanning trees have applications to the design of communication networks.
- A spanning forest of a graph is a spanning subgraph that is a forest.



Graph



Spanning tree

Depth-First Search

- **Depth-first search (DFS)** is a general technique for **traversing a graph**.
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G.
  - Determines whether G is connected.
  - Computes the connected components of G.
  - Computes a spanning forest of G.
- DFS on a graph with $\boldsymbol{n}$ vertices and $\boldsymbol{m}$ edges takes $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m})$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices.
  - Find a cycle in the graph.

DFS Algorithm

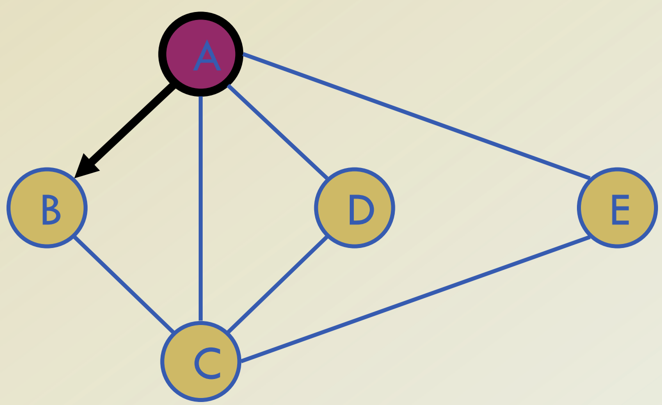
- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

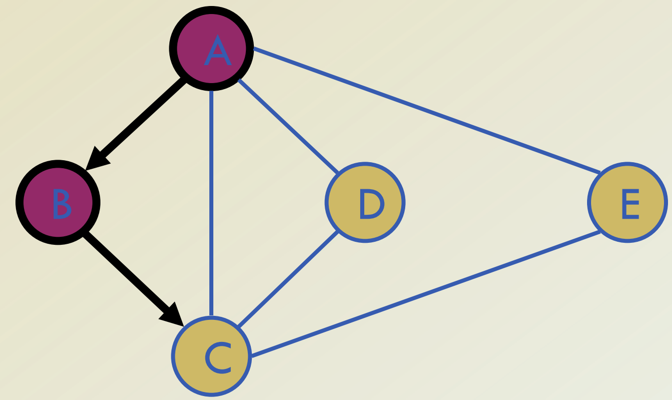
```
Algorithm DFS(G)
   Input graph G
   Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

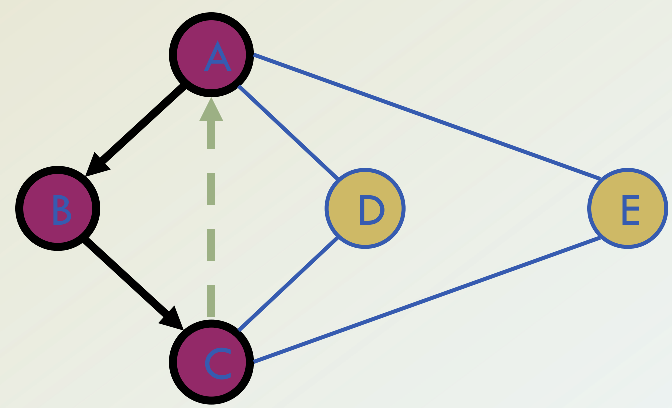
```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
     in the connected component of v
     as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
     if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
           setLabel(e, DISCOVERY)
           DFS(G, w)
        else
           setLabel(e, BACK)
```

Example

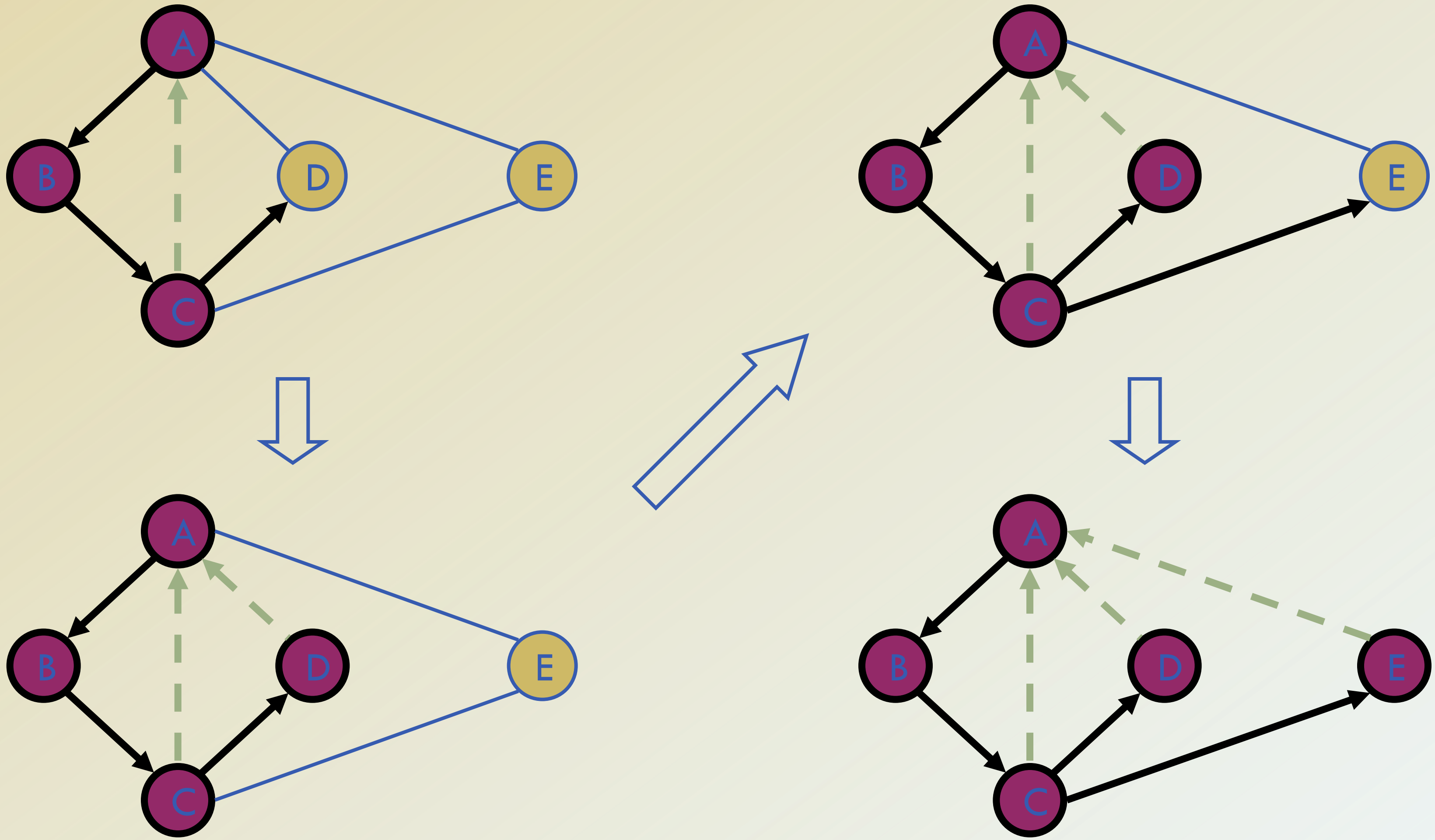
- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- back edge







Example (cont.)



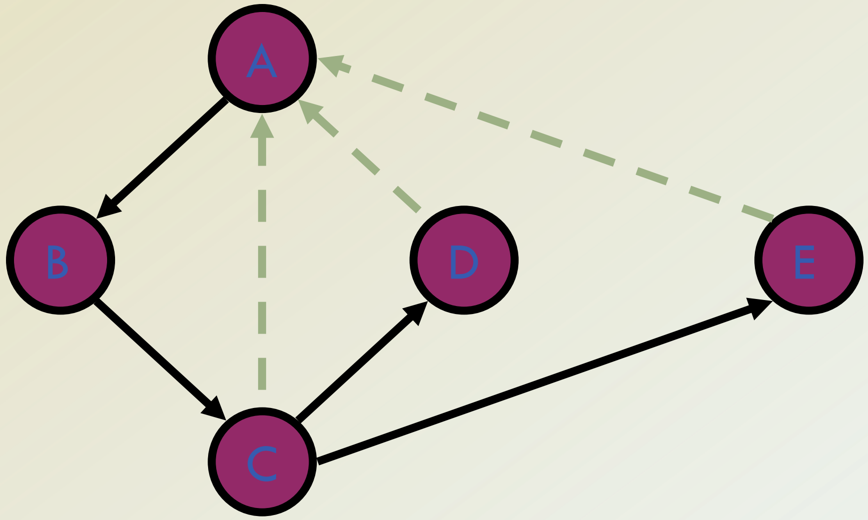
Properties of DFS

Property 1

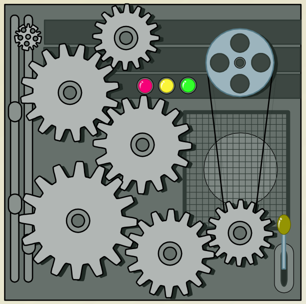
DFS(*G, v*) visits all the vertices and edges in the connected component of ***v***

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

Breadth-First Search

- **Breadth-first search (BFS)** is a general technique for **traversing a graph**.
- A **BFS** traversal of a graph G
  - Visits all the vertices and edges of G.
  - Determines whether G is connected.
  - Computes the connected components of G.
  - Computes a spanning forest of G.
- **BFS** on a graph with n vertices and m edges takes $O(n + m)$ time
- **BFS** can be further extended to solve other graph problems:
  - Find and report a path with the minimum number of edges between two given vertices.
  - Find a simple cycle, if there is one.

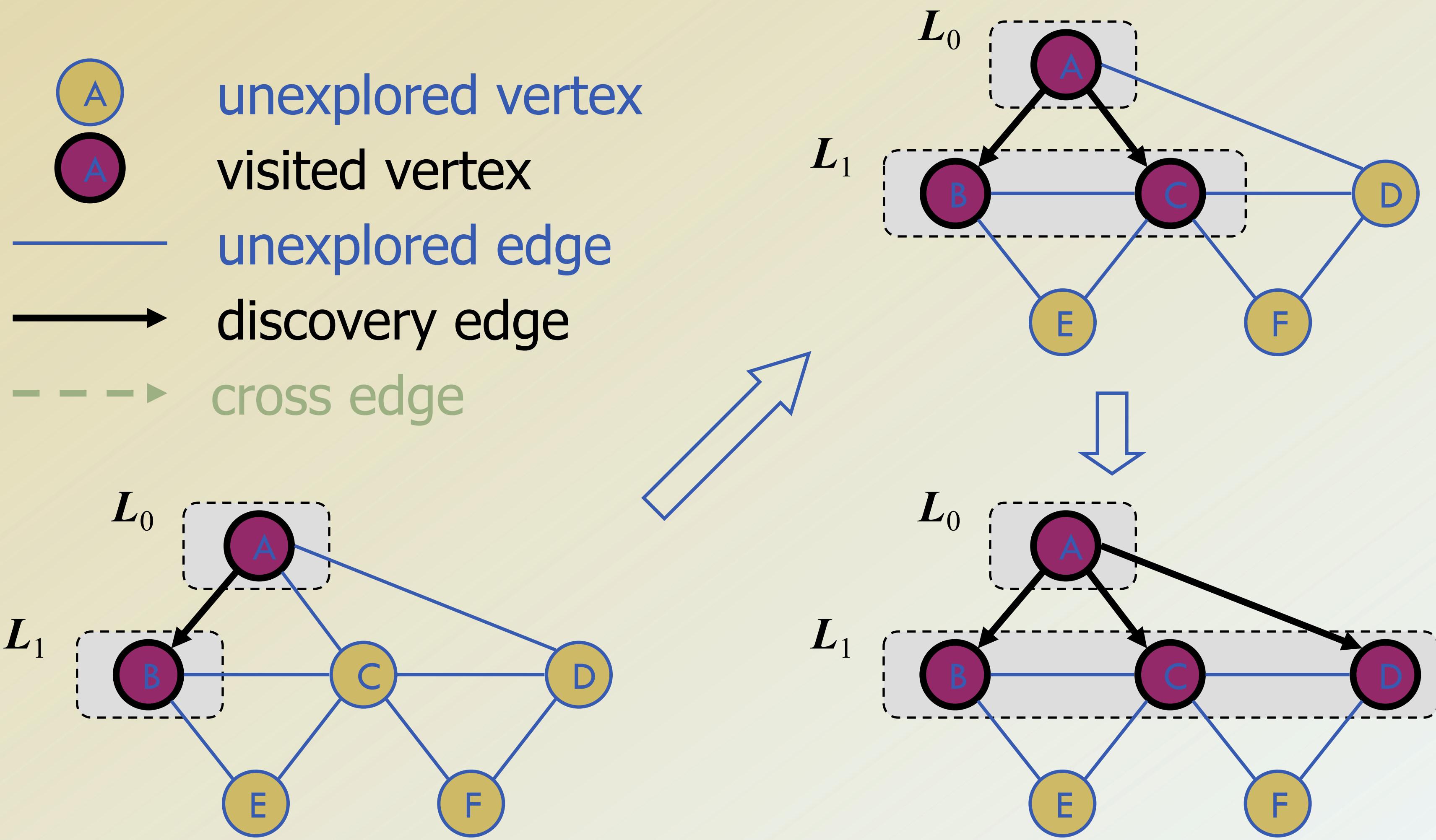
BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

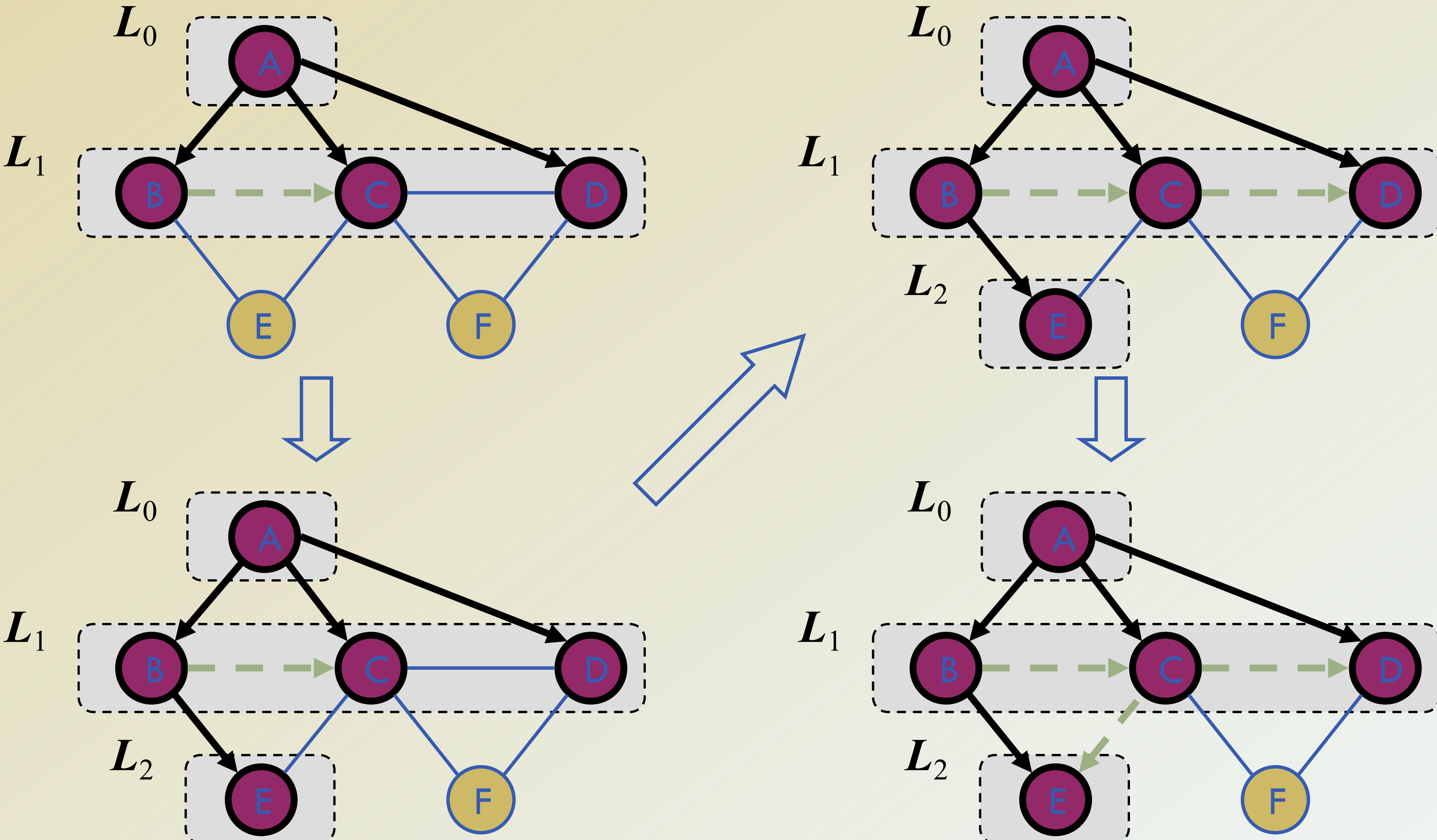
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

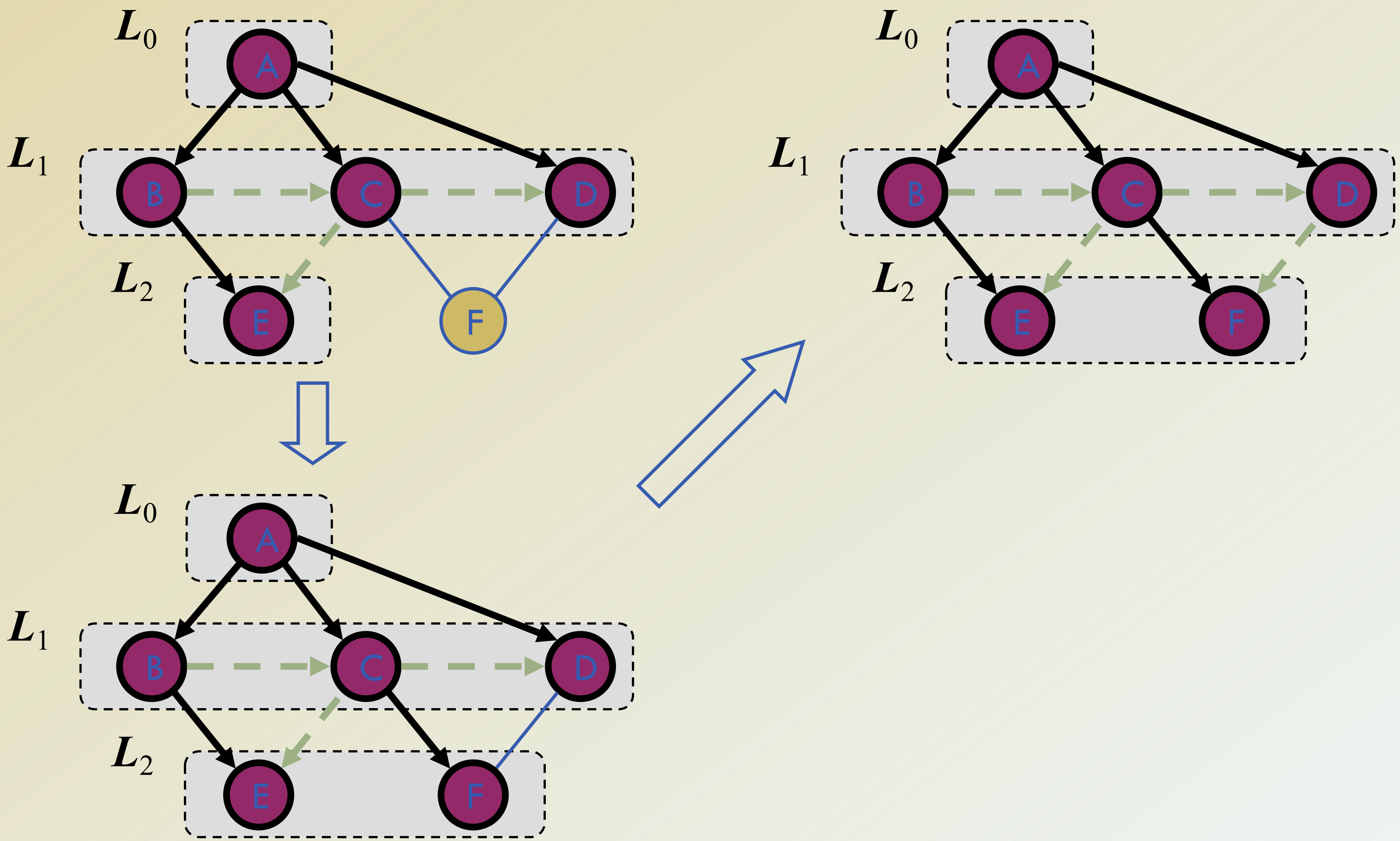
Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F

Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

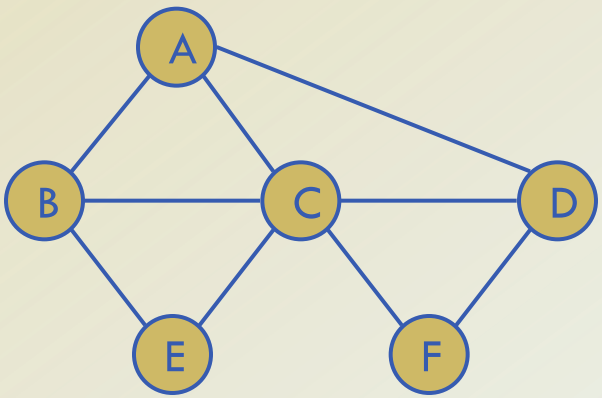
Property 2

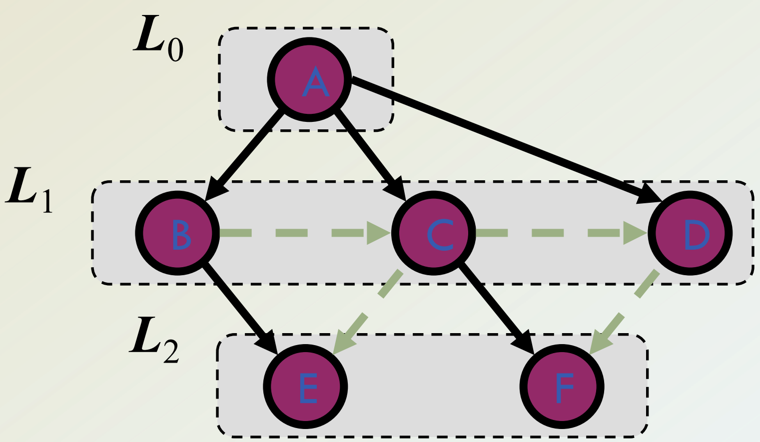
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges.
- Every path from s to v in G_s has at least i edges.



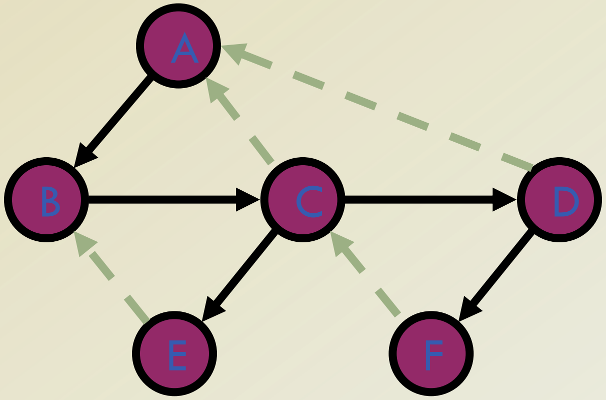


Analysis

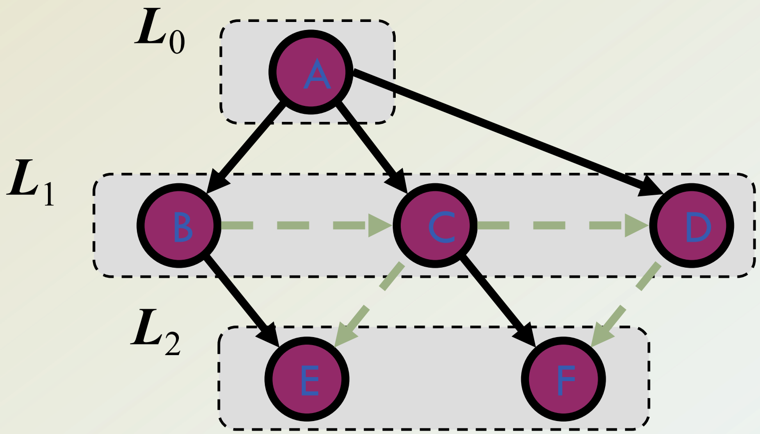
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice :
  - once as UNEXPLORED.
  - once as VISITED.
- Each edge is labeled twice:
  - once as UNEXPLORED.
  - once as DISCOVERY or CROSS.
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called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

DFS vs. BFS

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



DFS



BFS

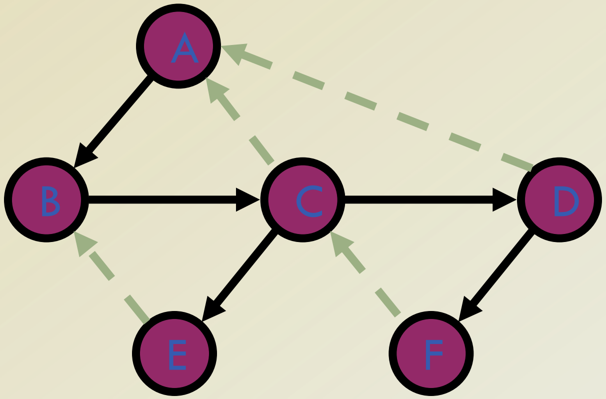
DFS vs. BFS (cont.)

Back edge (*v*,*w*)

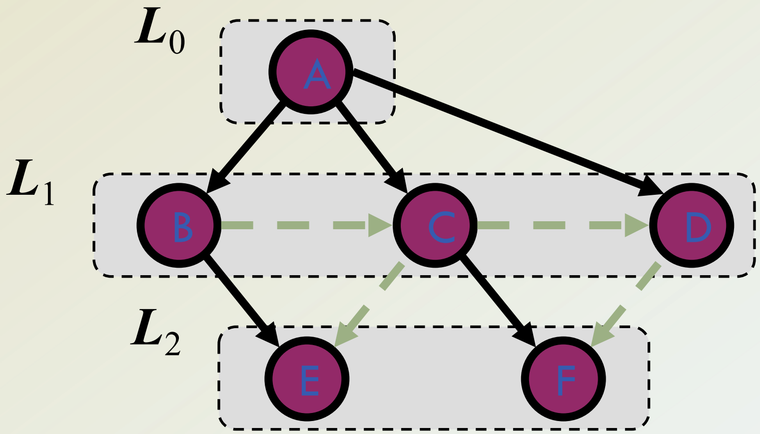
- *w* is an ancestor of *v* in the tree of discovery edges

Cross edge (*v*,*w*)

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



DFS



BFS

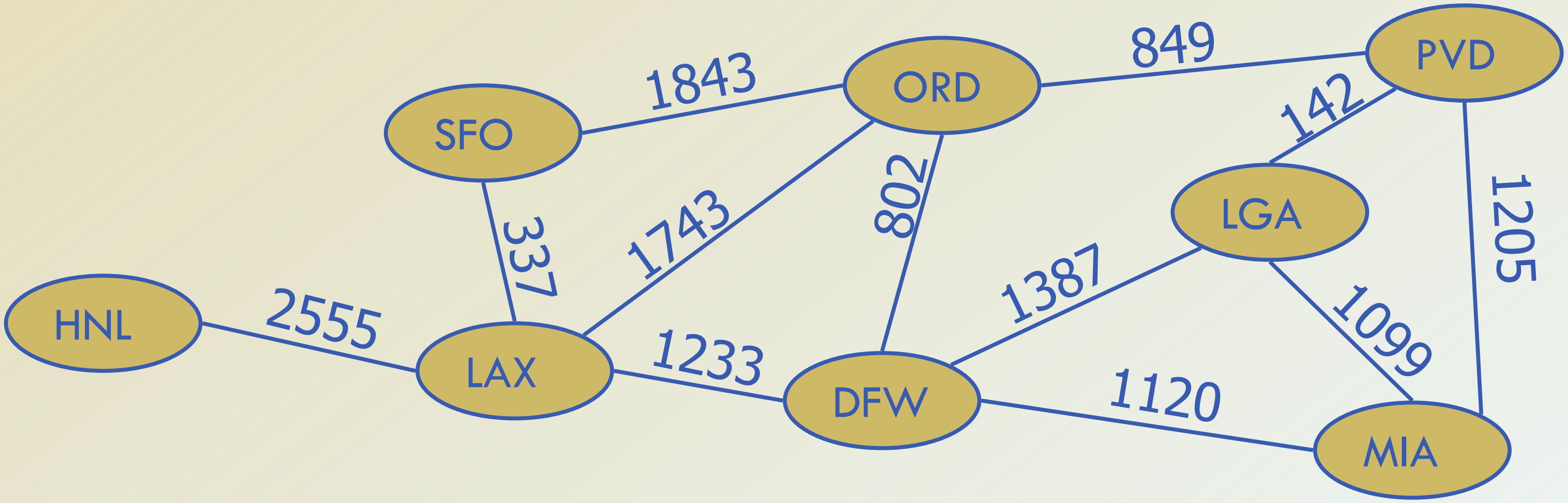
Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices ***u*** and ***z*** using the template method pattern
- We call ***DFS***(***G, u***) with ***u*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***z*** is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

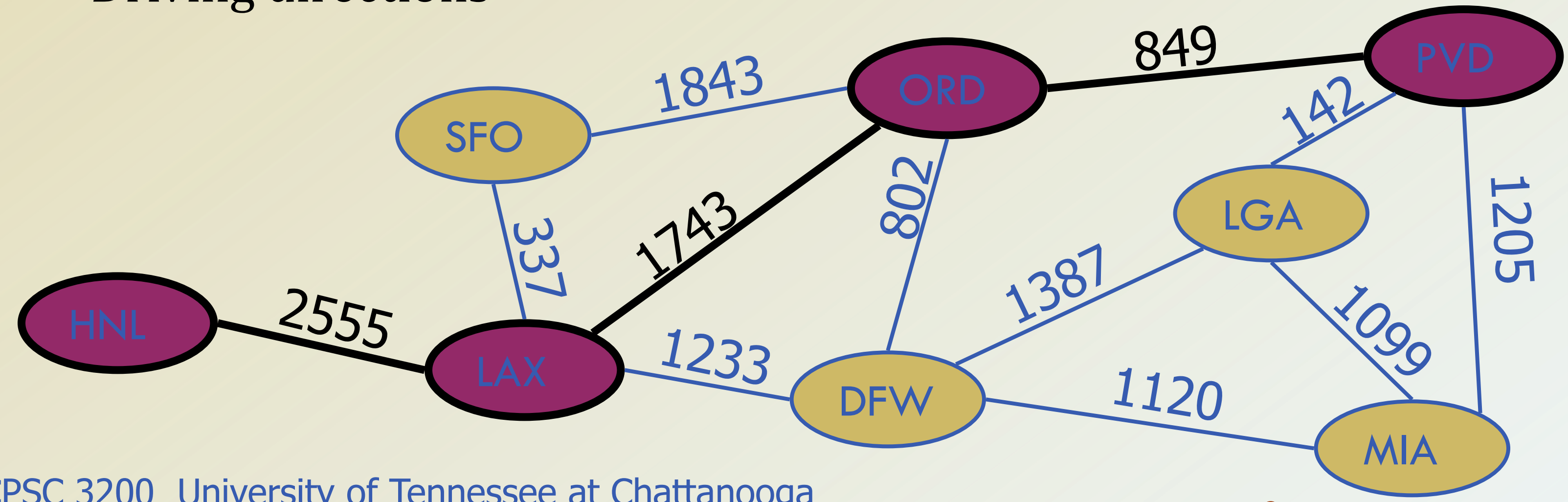
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v***.
  - Length of a path is the sum of the weights of its edges.
- **Example:**
  - Shortest path between Providence and Honolulu
- **Applications**
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path Properties

Property 1:

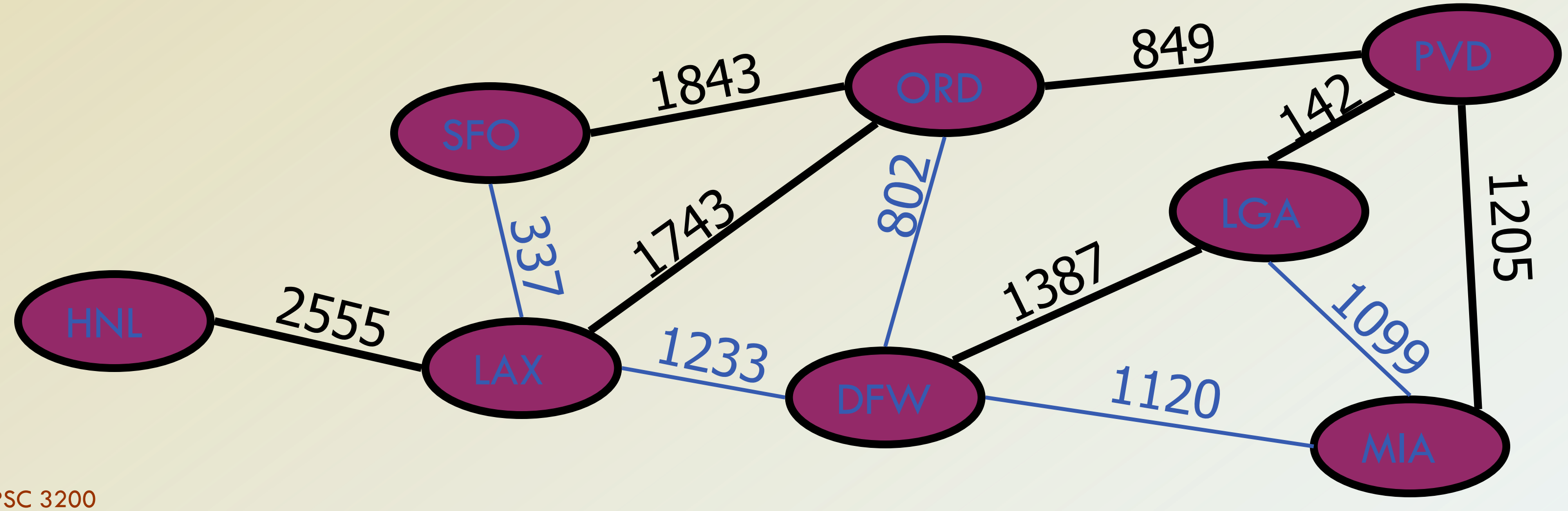
A subpath of a shortest path is itself a shortest path.

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices.

Example:

Tree of shortest paths from Providence.



Dijkstra's Algorithm

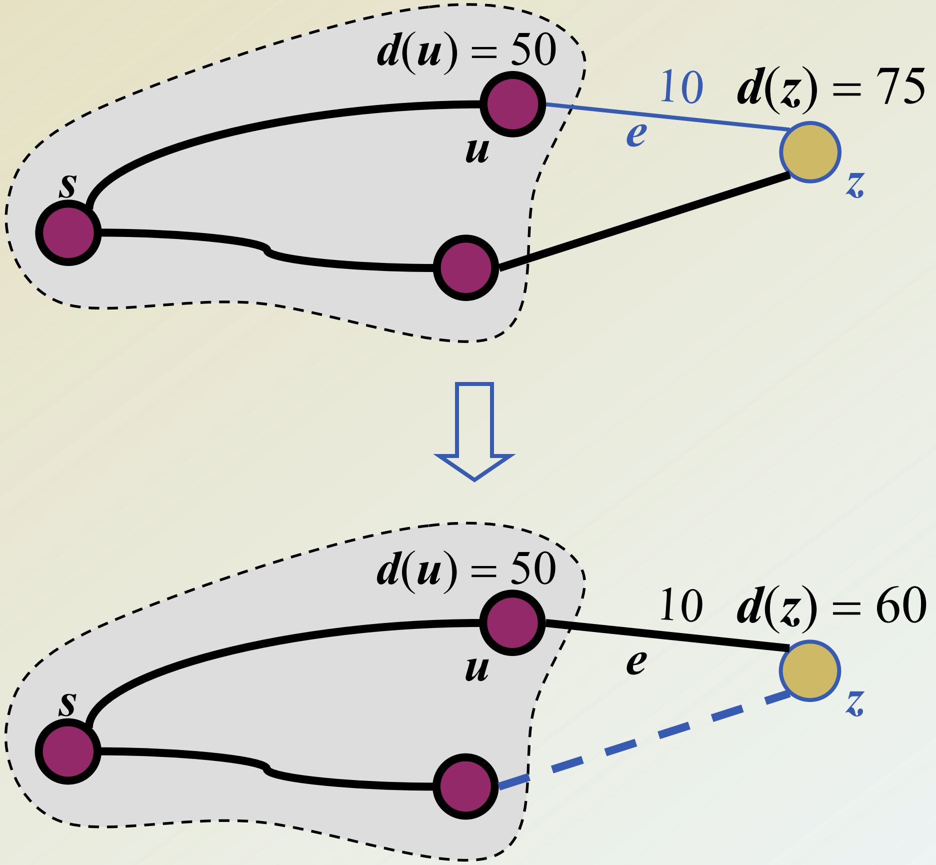
- The distance of a vertex ***v*** from a vertex ***s*** is the length of a shortest path between ***s*** and ***v.***
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex ***s.***
- **Assumptions:**
  - the graph is connected.
  - the edges are undirected.
  - the edge weights are nonnegative.
- We grow a "cloud" of vertices, beginning with ***s*** and eventually covering all the vertices.
- We store with each vertex ***v*** a label ***d***(***v***) representing the distance of ***v*** from ***s*** in the subgraph consisting of the cloud and its adjacent vertices.
- At each step
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d***(***u***).
  - We update the labels of the vertices adjacent to ***u.***

Edge Relaxation

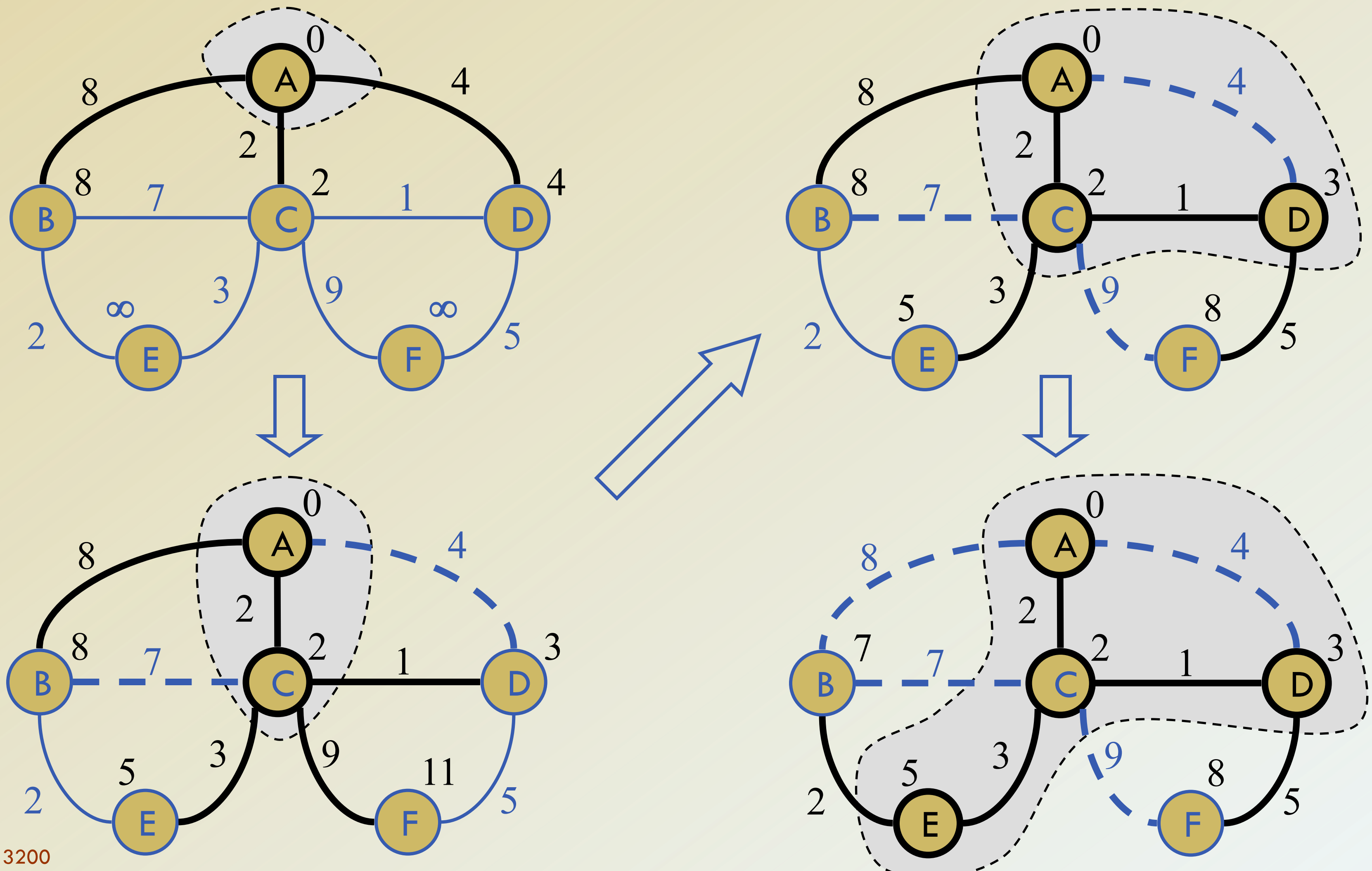
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud

- The relaxation of edge e updates distance $d(z)$ as follows:

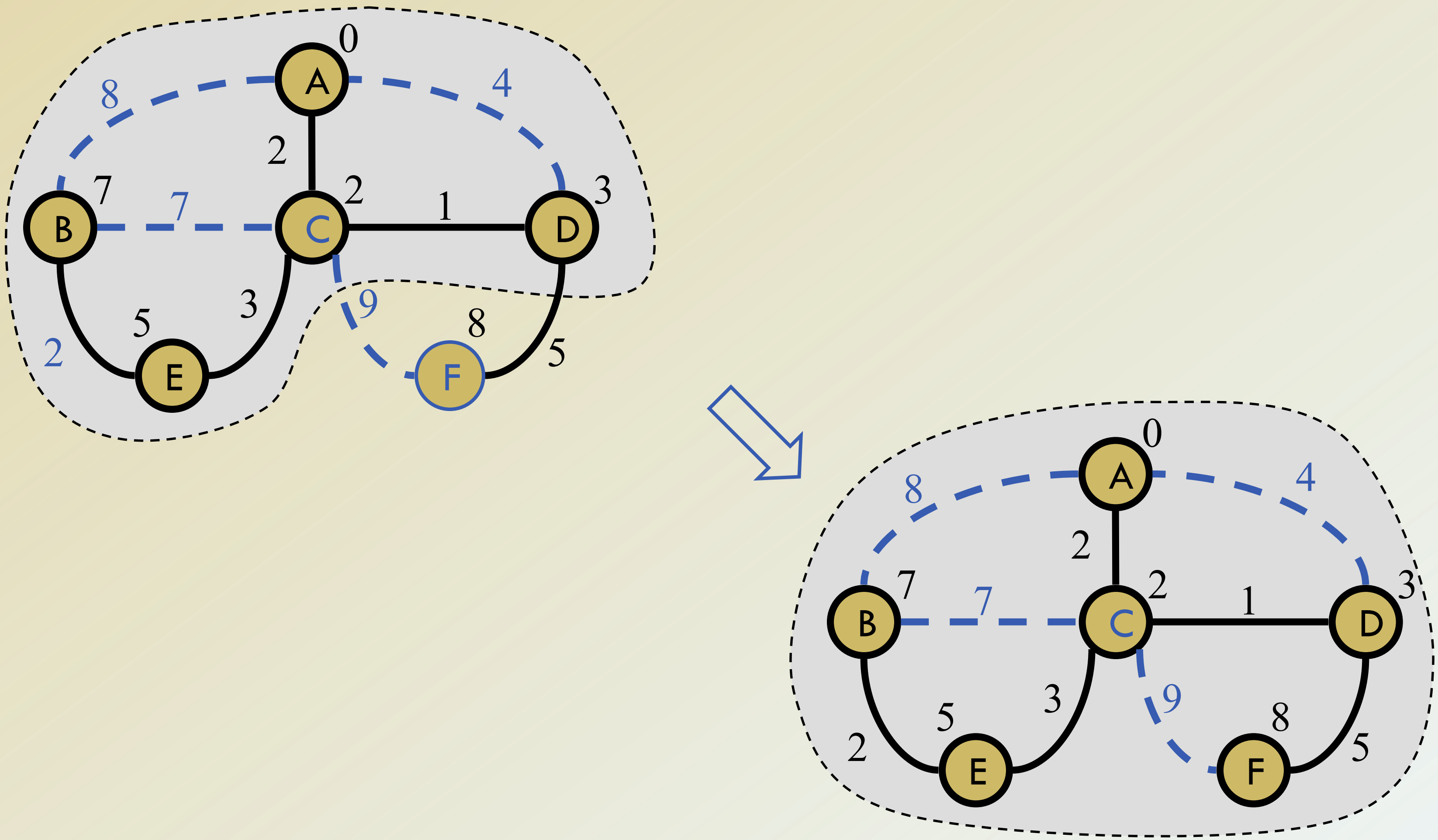
  $d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



Example


A B C D E F
0 8 4 2 8 7 2 1 4 ∞ 3 9 ∞ 2 5
0 8 4 2 8 7 2 1 3 5 3 9 11 2 5
0 8 4 2 8 7 2 1 3 5 3 9 8 2 5
0 8 4 2 7 7 2 1 3 5 3 9 8 2 5

Example (cont.)



Dijkstra's Algorithm

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
  - Key: distance
  - Value: vertex
  - Recall that method ***replaceKey***(***l***,***k***) changes the key of entry ***l***
- We store two labels with each vertex:
  - Distance
  - Entry in priority queue

```
Algorithm DijkstraDistances(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setEntry(v, l)
  while ¬Q.isEmpty()
    l ← Q.removeMin()
    u ← l.getValue()
    for all e ∈ G.incidentEdges(u) { relax e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        Q.replaceKey(getEntry(z), r)
```

Analysis of Dijkstra's Algorithm

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

End of Chapter 13