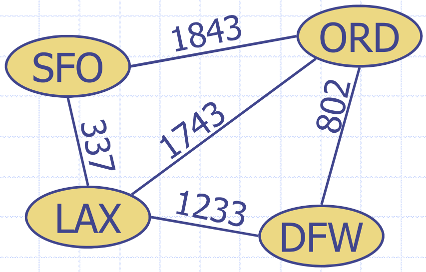
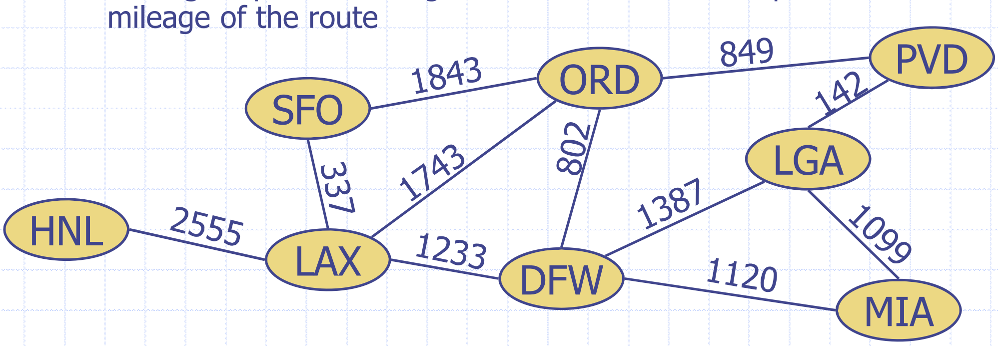
Graphs



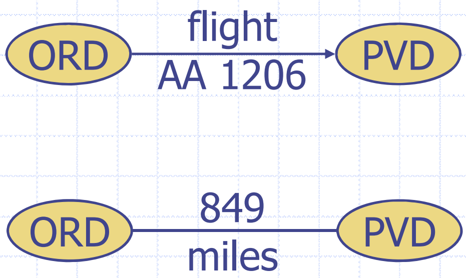
Graphs

- A graph $G = (V, E)$, 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



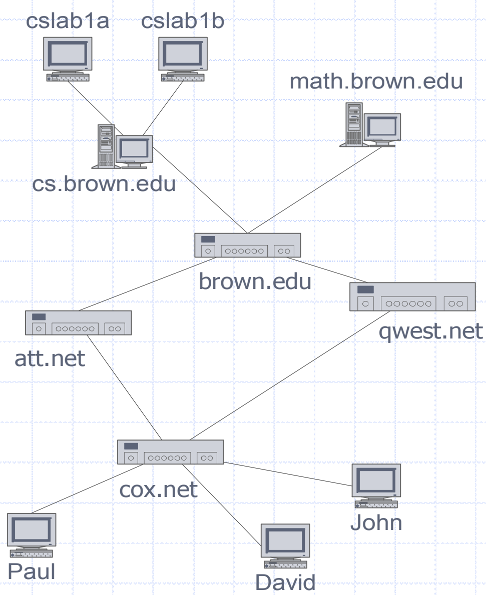
Edge Types

- 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



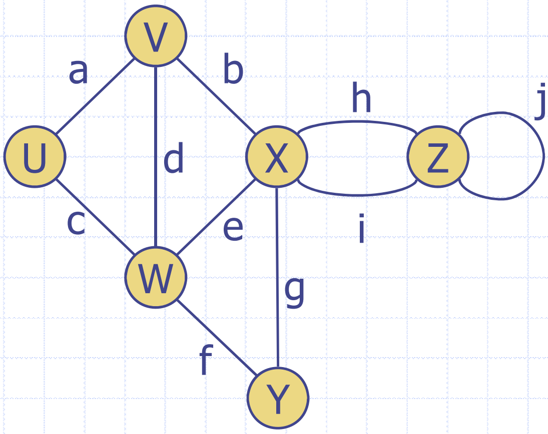
Applications

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



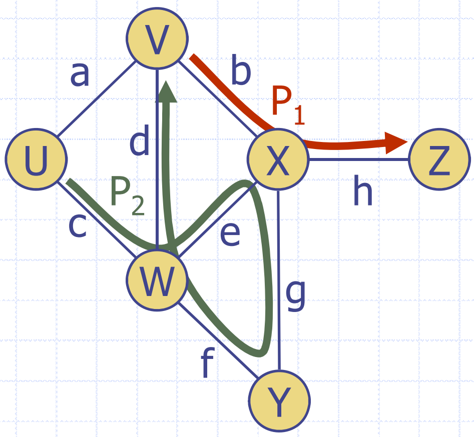
Terminology

- 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



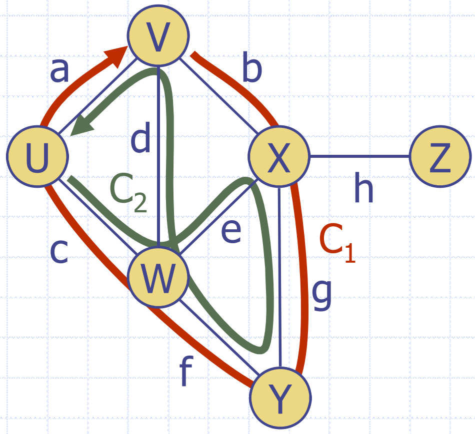
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
  - The *length* of a path is the number of edges in the path.
- 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
  - A cycle that all its vertices and edges are distinct except the first and the last vertices
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) 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

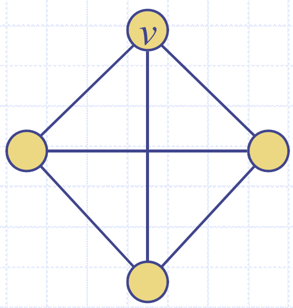
$m \leq n\,(n-1)/2$

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

What is the bound for a directed graph?

Notation

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



Example

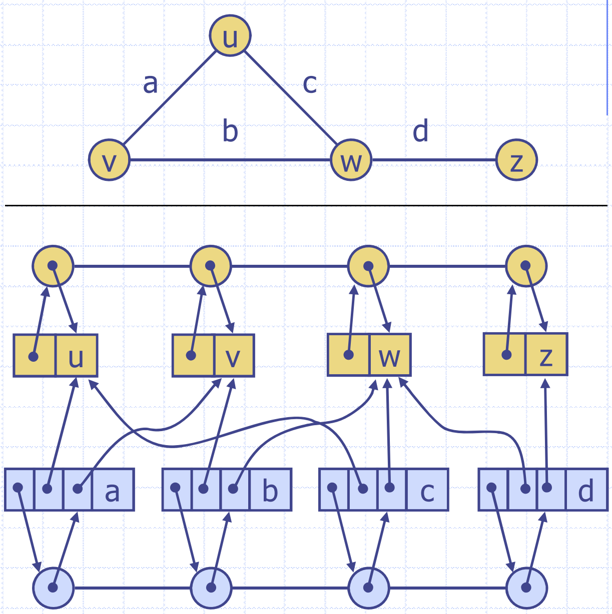
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

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

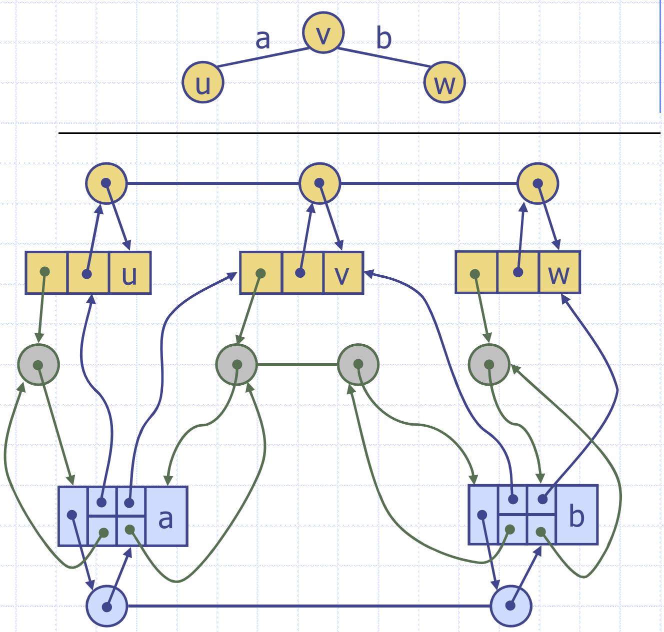
Edge List Structure

- 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



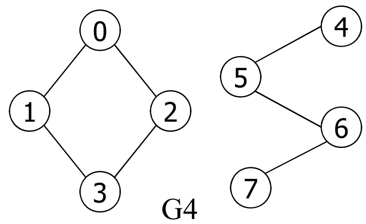
Adjacency List Structure

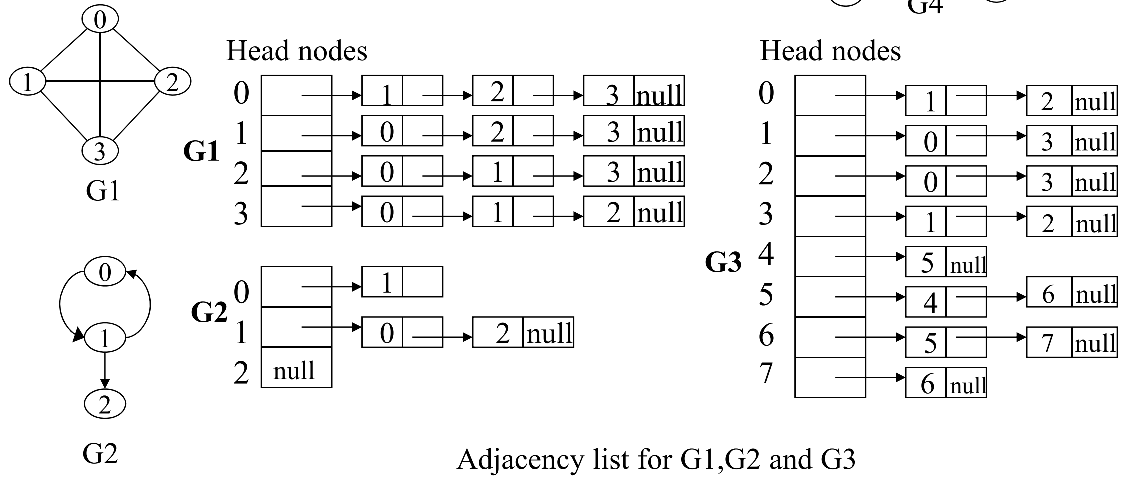
- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



Adjacency List Structure

- Use *n* linked list for each vertex

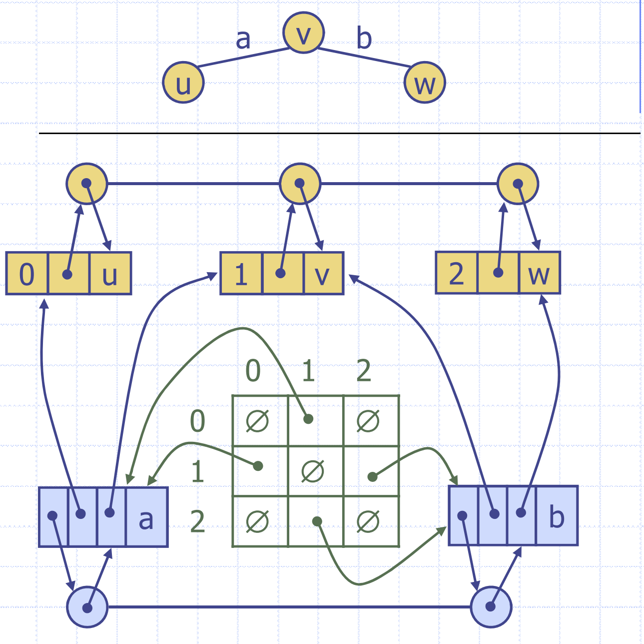




Adjacency list for G1,G2 and G3

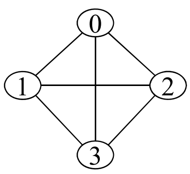
Adjacency Matrix Structure

- 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

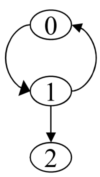


Adjacency Matrix

- Adjacency matrix for G is a two-dimensional n x n array.
- adj_mat[i][j]=1 if edge(v_i,v_j) exists in E(G), otherwise adj_mat[i][j]=0.



G1



G2

$$G1 \quad \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} 0\,1\,2\,3 \\ \begin{pmatrix} 0&1&1&1 \\ 1&0&1&1 \\ 1&1&0&1 \\ 1&1&1&0 \end{pmatrix} \end{array}$$

$$G3 \quad \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} 0\,1\,2 \\ \begin{pmatrix} 0&1&0 \\ 1&0&1 \\ 0&0&0 \end{pmatrix} \end{array}$$

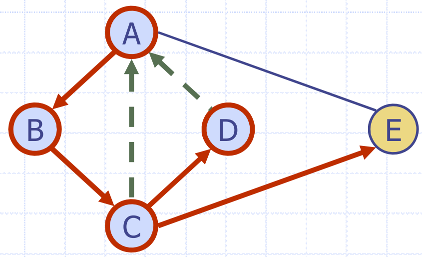
$$G3 \quad \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} 0\,1\,2\,3\,4\,5\,6\,7 \\ \begin{pmatrix} 0&1&1&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0 \\ 1&0&0&1&0&0&0&0 \\ 0&1&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&0 \end{pmatrix} \end{array}$$

Adjacency matrices for G1, G2,and G3

Performance

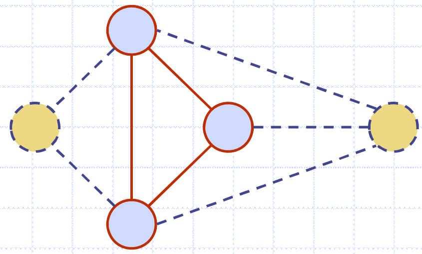
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Adjacency List	Adjacency Matrix
incidentEdges(v)	deg(v)	n
areAdjacent (v, w)	min(deg(v), deg(w))	1
insertVertex(o)	1	n^2
insertEdge(v, w, o)	1	1
removeVertex(v)	deg(v)	n^2
removeEdge(e)	1	1

Depth-First Search



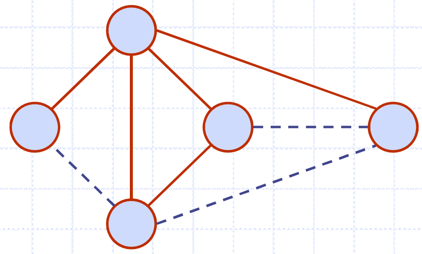
Subgraphs

- G′ is a **subgraph** of G when V(G) ⊇ V(G′) and E(G) ⊇ E(G′)



Subgraph

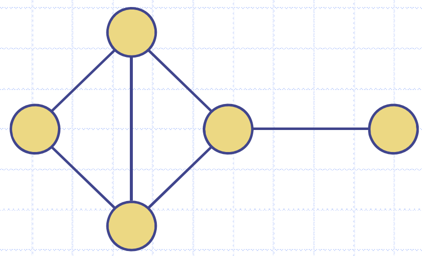
- A **spanning subgraph** of G is a subgraph that contains all the vertices of G



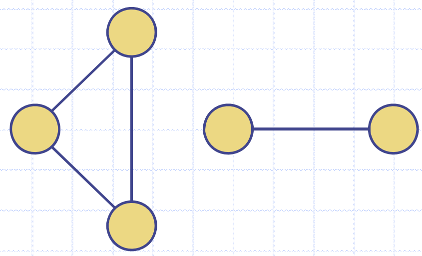
Spanning subgraph

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

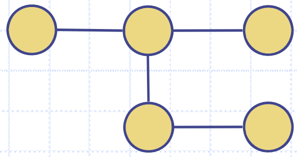


Non connected graph with two connected components

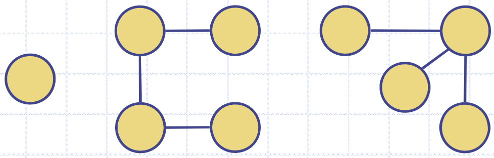
Trees and Forests

- 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



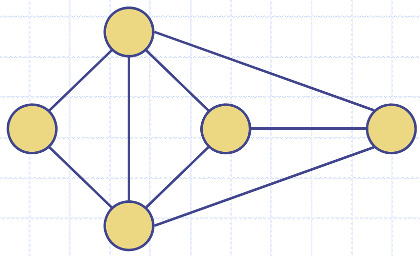
Tree



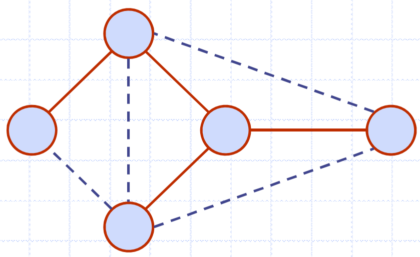
Forest

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



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
- DFS on a graph with n vertices and m edges takes $O(n + m)$ 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
- Depth-first search is similar to Preorder traversal in Trees

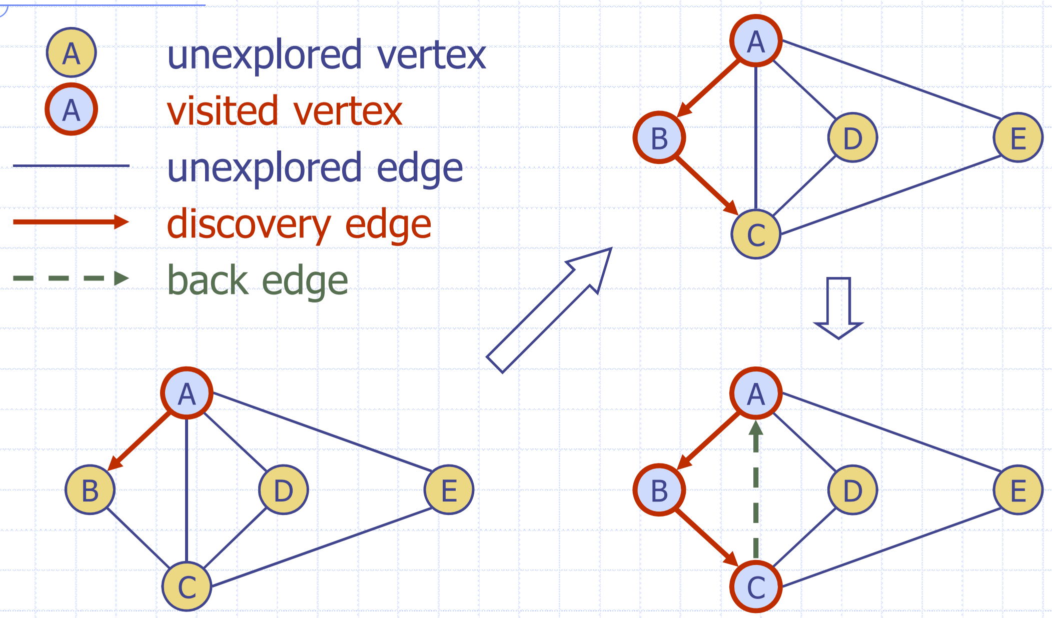
DFS Algorithm

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

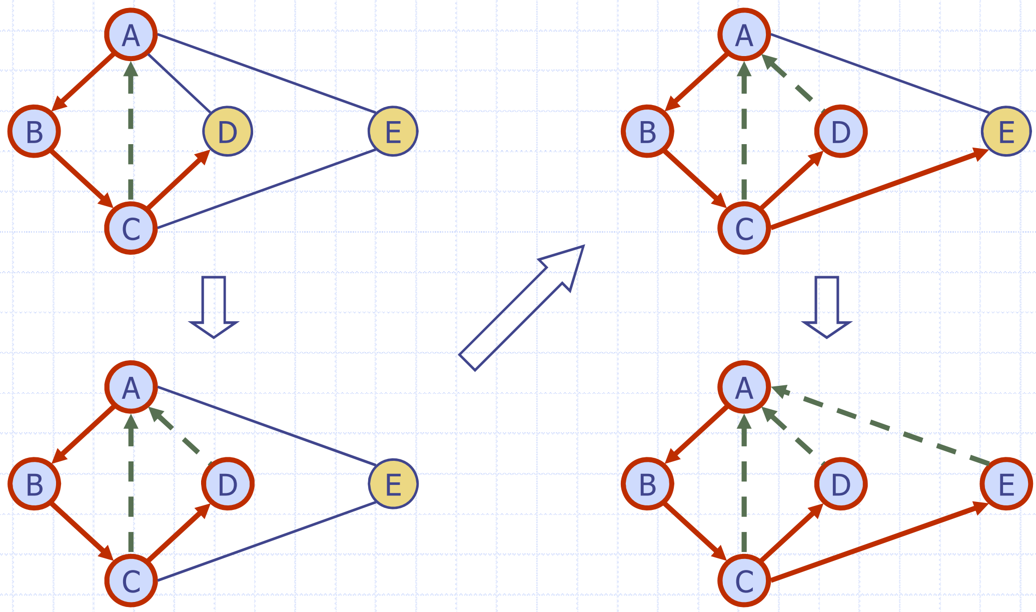
```
Algorithm DFS(G)
  Input graph G
  Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)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

```
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



Example (cont.)



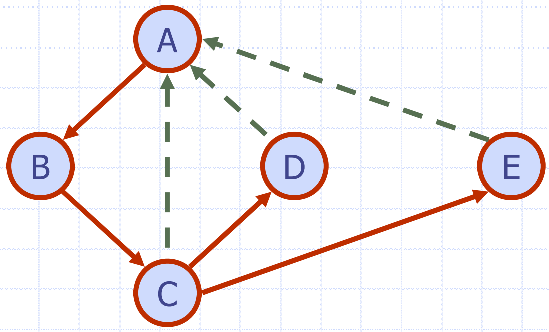
Properties of DFS

Property 1

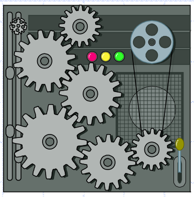
$DFS(G, v)$ visits all the vertices and edges in the connected component of v

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

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z
- 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

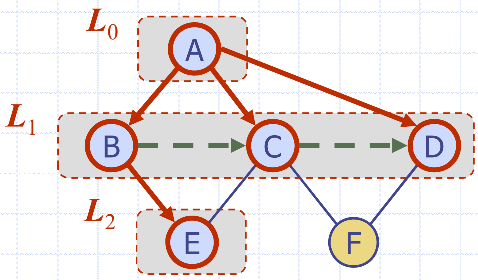
```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
         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


Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                cycleDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Breadth-First Search



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
- 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

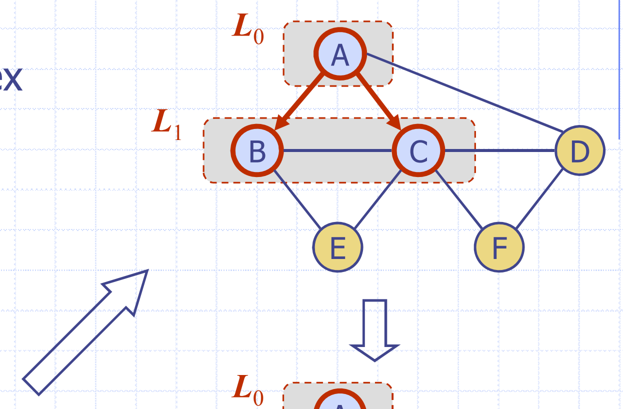
A unexplored vertex

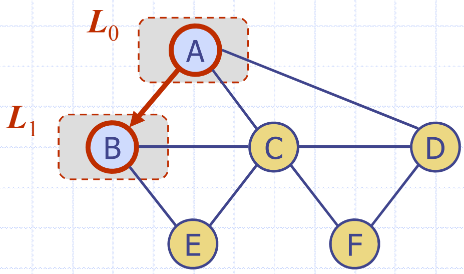
A visited vertex

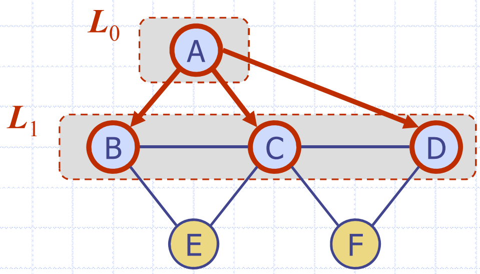
unexplored edge

discovery edge

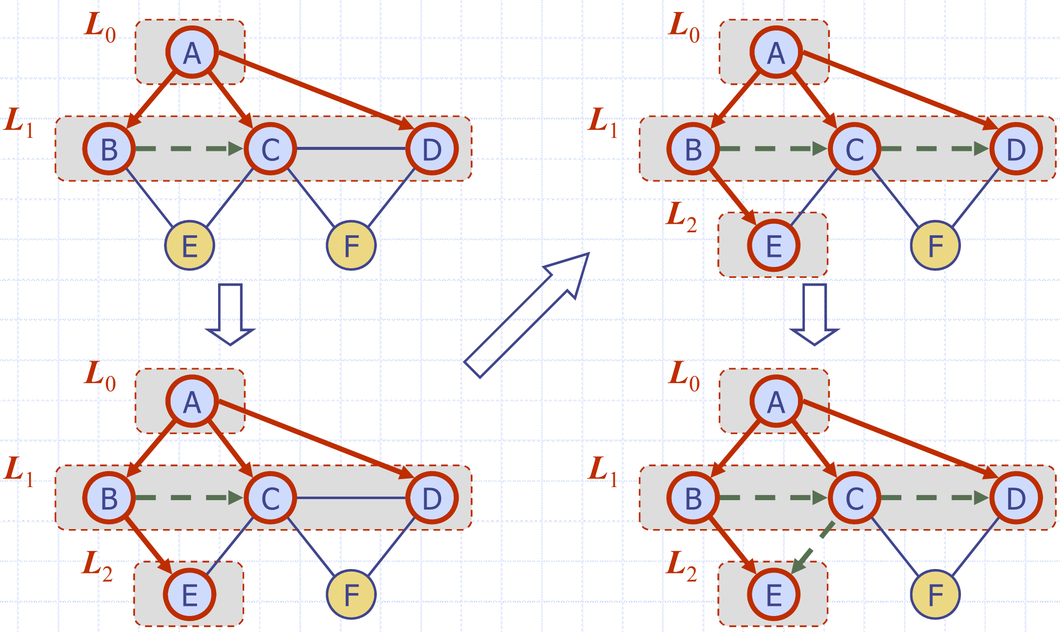
cross edge



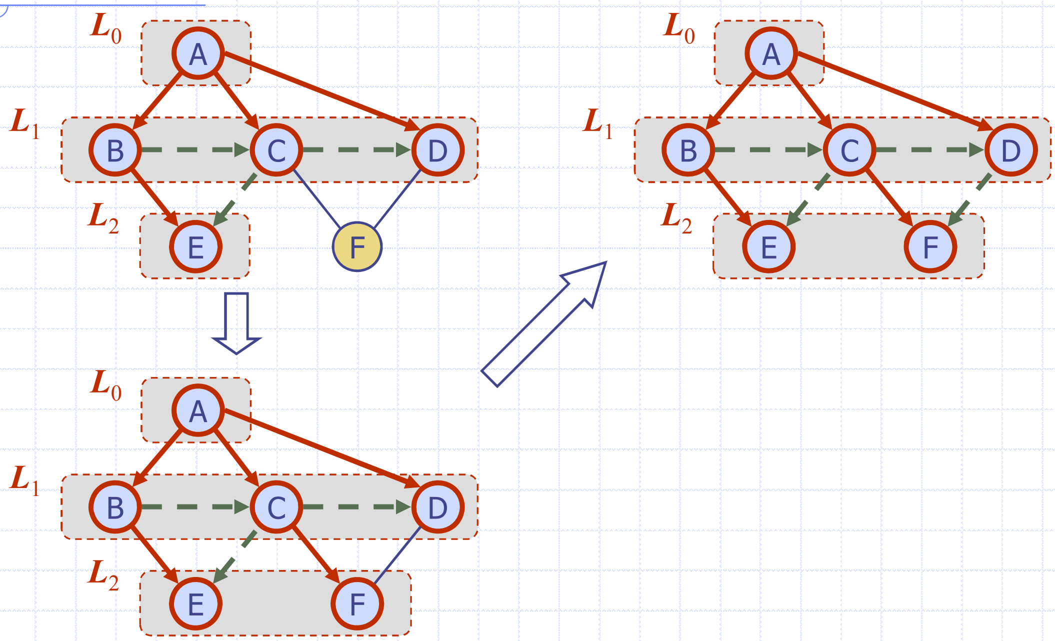




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

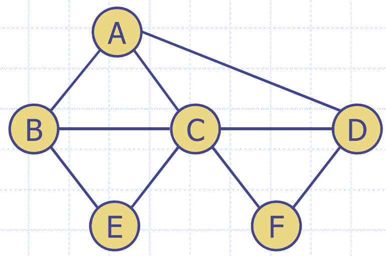
Property 2

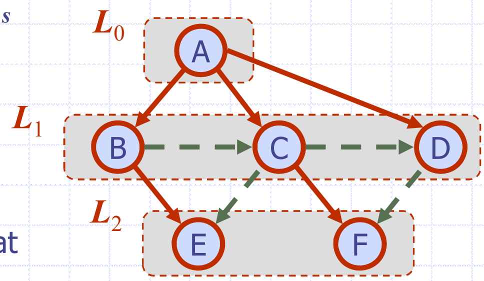
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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

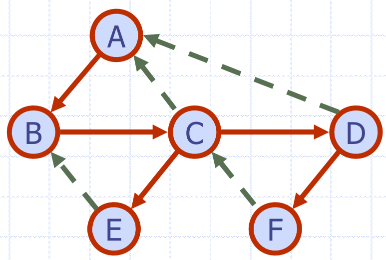
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once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

Applications

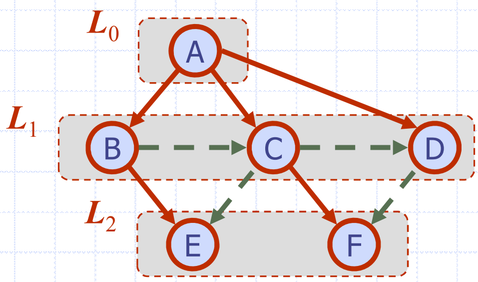
- We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Connected components, paths, cycles	√	√
Shortest paths		√



DFS



BFS

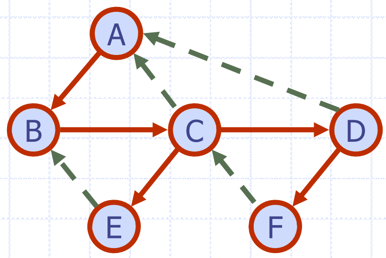
DFS vs. BFS (cont.)

Back edge (v,w)

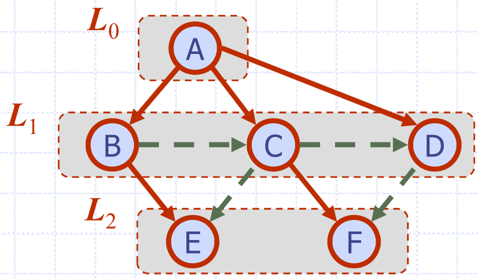
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



DFS



BFS

DFS vs. BFS (cont.)

- ❑ Problem
  For an undirected graph G = (V,E) and a starting vertex v ∈ V(G), visit all the vertices that is reachable from v.
  - o Depth First Search(DFS) : similar to preorder tree traversal
  - o Breath First Search(BFS) : similar to level order tree traversal

- ❑ Depth First Search
  1. Visit start vertex v
  2. Select a vertex w that has not been visited from adjacency list of v and perform Depth First Search on w (Use Recursion)

- ☐ Breadth First Search
  1. Starting from vertex v, visit all the vertices in the adjacency list of v and mark them as "visited" and insert into the queue.
  2. Extract a vertex from queue and visit all the unvisited vertices in that vertex's adjacency list. Repeat this until the queue becomes empty.

Connected Components

Problem

1. Is a given undirected graph connected?
2. Output connected components of a graph.

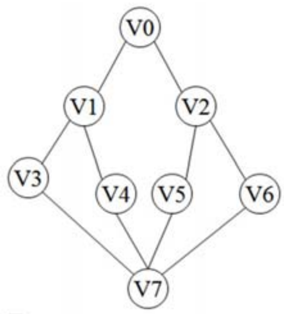
- Pseudo-code

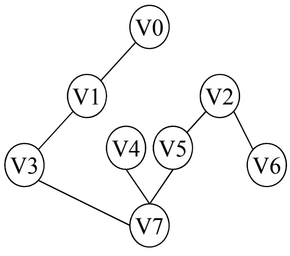
```
void connected ( )
{
  /* determine the connected components of a graph */
    for ( int i = 0 ; i < n ; i++ )
      if  ( !visited [ i ] )  {
          DFS( i );      / * DFS searches all the vertices in a component* /

           //  # of components is the same as the # of  "\n"
          System.out.println ( "" );
        }
}
```


Spanning Trees (Revisited)

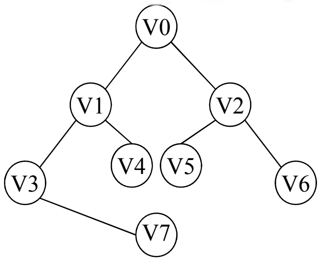
A **spanning tree** of a connected graph is a spanning subgraph that is a tree.

Depth first spanning tree *vs.* Breadth first spanning tree





(a) DFS(0) spanning tree



(b) BFS (0) spanning tree

Minimum Cost Spanning Trees

The cost of spanning tree for a weighted undirected graph is the sum of the cost(weight) of all edges in the spanning tree.

The minimum cost spanning tree has the least cost among all spanning trees.

Minimum Cost Spanning Trees Algorithms

- Kruskal Algorithm
- Prim Algorithm

Overview of Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time.

The algorithm selects the edges for inclusion in T in nondecreasing order of their cost.

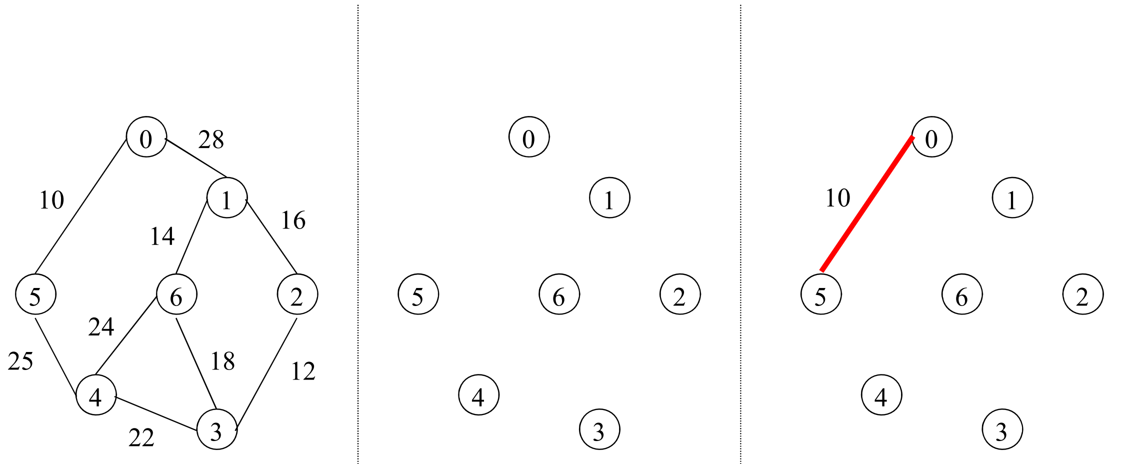
An edge is added to T if it does not form a cycle with the edges that are already in T.

Since G is connected and has $n > 0$ vertices, exactly $n-1$ edges will be selected for inclusion in T.

Kruskal’s Algorithm

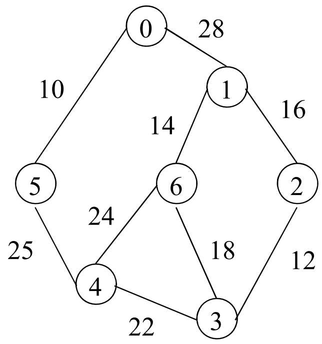
```
T = { } ;

while  ( T contains less than n -1 edges  && E is not empty )
{
      choose  a  least cost edge (v,w) from E ;
      delete (v,w) from E ;

      if ((v,w) does not create a cycle in T )
            add  (v,w) to T;
      else
            discard  (v,w);
   }

   if ( T contains fewer than n -1 edges)
       System.out.println (“No spanning tree”);  /*the graph is not connected*/
```


Kruskal’s Algorithm

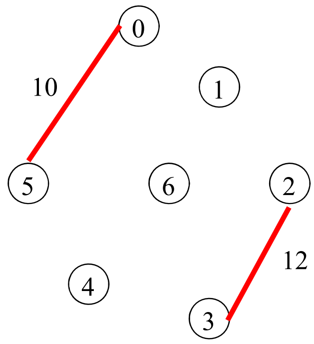


Graph G (a) (b)

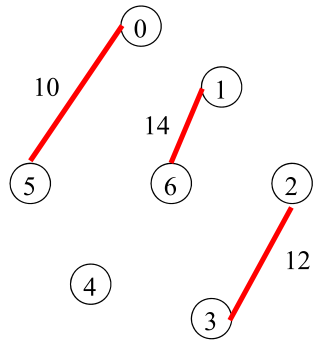
Kruskal's Algorithm



Graph G

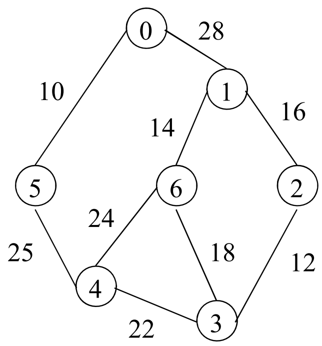


(c)

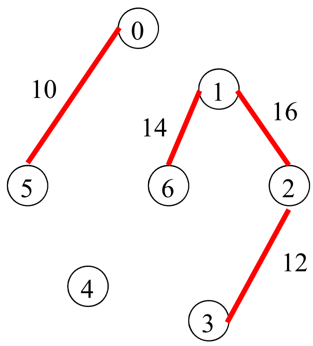


(d)

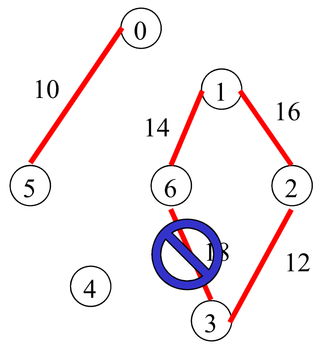
Kruskal's Algorithm



Graph G

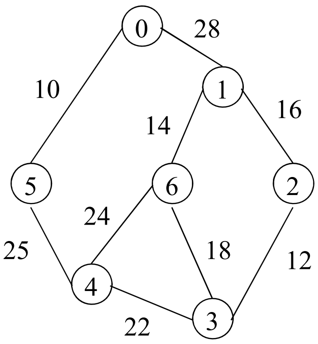


(e)

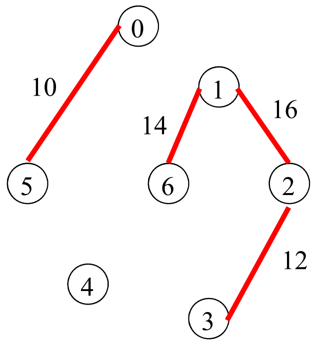


(f)

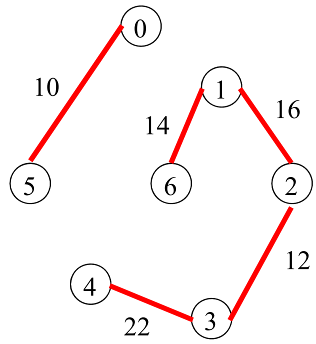
Kruskal’s Algorithm



Graph G

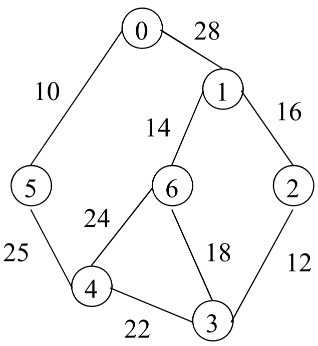


(e)

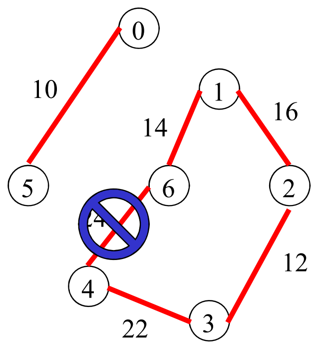


(f)

Kruskal’s Algorithm

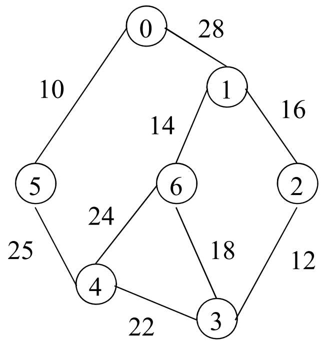


Graph G

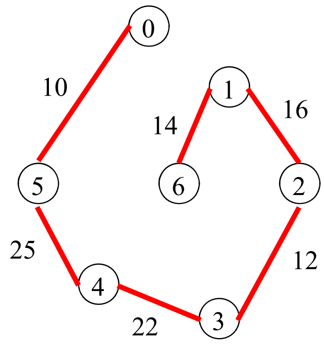


(g)

Kruskal’s Algorithm



Graph G



(g)

Cost = 10+12+14+16+22+25

Overview of Prim’s algorithm

Prim’s algorithm, like Kruskal’s algorithm, constructs a minimum cost spanning tree *T* by adding edges to *T* one at a time.

However, at each stage, the set of selected edges forms a tree.
(By contrast, it forms a forest in Kruskal’s algorithm)

Prim’s algorithm begins with a tree, T, that contains a single vertex.

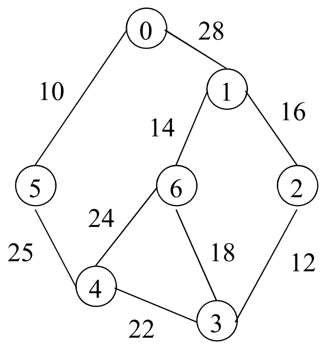
Next, we add a least cost edge (u, v) to T such that $T \cup \{(u,v)\}$ is also a tree.

Repeat this edge addition step until *T* contains n-1 edges.

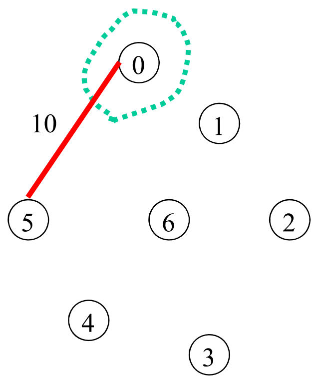
Prim's Algorithm

```
T = { } ;  /* T is set of tree edges */
TV = { 0 } ; /*TV is set of tree vertices */
/* start with vertex 0 and no edges */

while  ( T contains less than n -1 edges )  {
    let (u ,v) be a least cost edge such that u ∈ TV and v ∉ TV;
    if (there is no such edge )
        break ;
    add v to TV;
    add (u , v) to T
}

if  ( T contains fewer than n -1 edges)
    System.out.println (" No spanning tree");
```

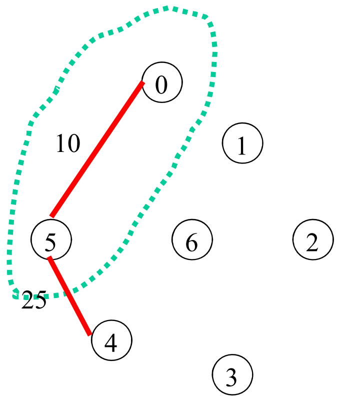

Prim's Algorithm



Graph G

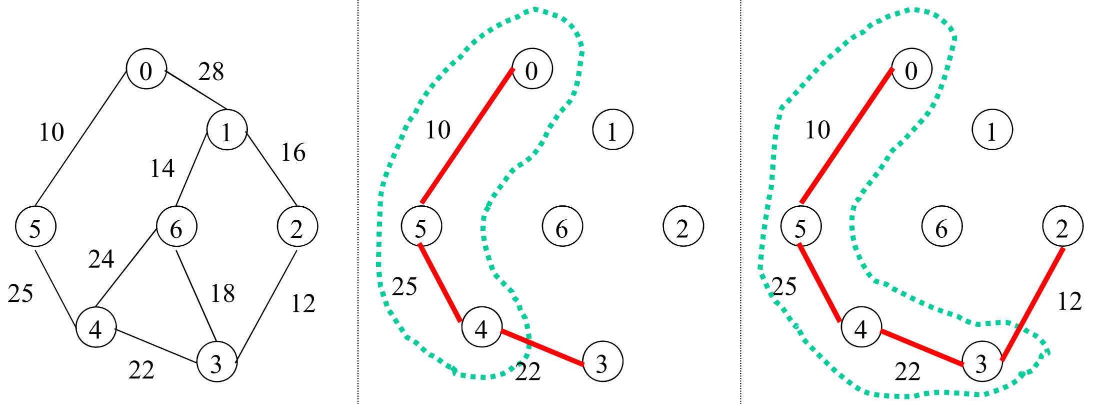


(a)



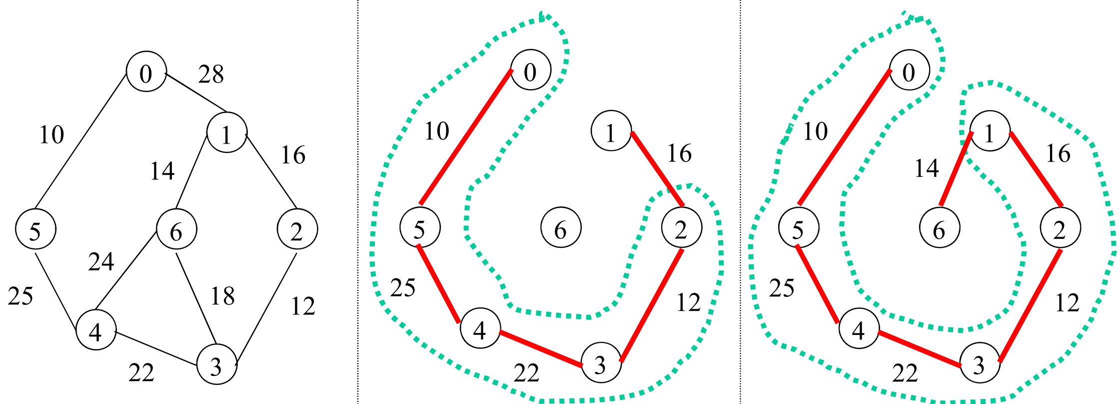
(b)

Prim’s Algorithm



Graph G (c) (d)

Prim’s Algorithm



Graph G (e) (f)

Cost = 10+25+22+12+16+14

Shortest Paths

Let G be a weighted graph.

- The *length* of a path, P, is the sum of the weights of the edges of P.
- The *distance* from a vertex v to a vertex u in G, denoted $d(v, u)$, is the length of a minimum length path (also called *shortest path*) from v to u.

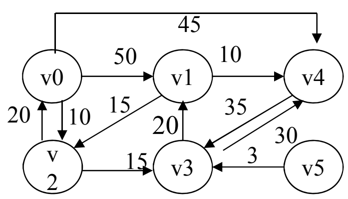
Two Types of Shortest Paths Problems

- Single Source Shortest Paths Problem
  - Dijkstra's Algorithm
- All-Pairs Shortest Paths Problem
  - Floyd-Warshall Algorithm

Assume all the weights in G are non-negative.

Single Source Shortest Paths

In this problem, we are given a directed graph $G=(V,E)$, a weighting function $w(e)$ (>0) for the edges of G, and a source vertex, v_0. We wish to determine a shortest path from v_0 to each of the remaining vertices of G.



Path	length
1) $v_0\ v_2$	10
2) $v_0\ v_2\ v_3$	25
3) $v_0\ v_2\ v_3\ v_1$	45
4) $v_0\ v_4$	45

Graph and its paths from v0

Dijkstra's Algorithm

The algorithm works by maintaining a set *S* of vertices whose shortest distance from the source is already known. Initially *S* contains only the source vertex.

At each step, we add to *S* a remaining vertex *v* whose distance from the source is as short as possible. Since all edges have non-negative costs, we can always find a shortest path from the source to *v* that passes only through vertices in *S*. Call such a path *special*.

At each step, we use an array *D* to record the length of the shortest special path to each vertex.

Once *S* includes all vertices, all paths are special, so *D* will hold the shortest distance from the source to each vertex.

Dijkstra's Algorithm
(Cont'd)

Assume we are given a graph $G = (V, E)$ where $V = \{0, 1, \dots, n\text{-}1\}$ and vertex 0 is the source.

C is a two-dimensional array of costs, where $C[i][j]$ is the cost of edge from vertex i to vertex j. If there is no edge (i,j), then assume $C[i][j] = \infty$.

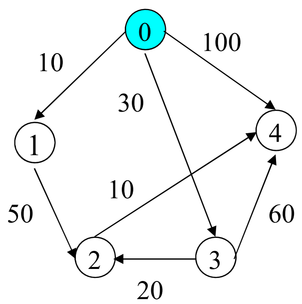
At each step, $D[i]$ contains the length of the current shortest special path to vertex i.

Dijkstra's algorithm

(Cont'd)

```
// Compute the cost of the shortest paths from vertex 0 to every
// vertex of a directed graph
Algorithm Dijkstra() {
   S = {0};
   for (i = 1; i < n; i++)
      D[i] = C[0][i];  // initialize D
   while ( there are remaining vertices in V-S ) {
      choose a vertex w in V-S such that D[w] is a minimum;
      add w to S;
      for ( each vertex v in V-S )
         D[v] = min(D[v], D[w] + C[w][ v]);
   }
}
```

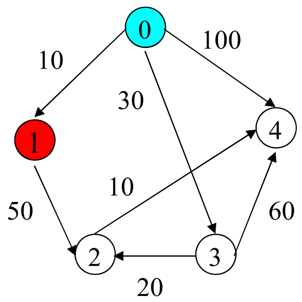

Dijkstra's Algorithm



{0,0,0,0,0}

Iteration	S	w	D[1]	D[2]	D[3]	D[4]
initial	{0}	-	10	∞	30	100

Dijkstra’s Algorithm



{0,0,1,0,0}

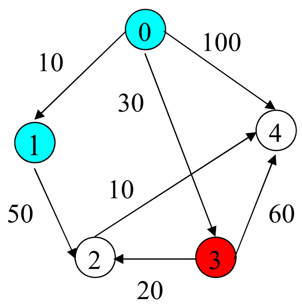
Iteration	S	w	D[1]	D[2]	D[3]	D[4]
initial	{0}	-	10	∞	30	100
1	{0,1}	1	10	60	30	100

D[2] = ∞ > D[1] + C[1][2] = 10+50 = 60

D[3] = 30 < D[1] + C[1][3] = 10+ ∞ = ∞

D[4] = 100 < D[1] + C[1][4] = 10+ ∞ = ∞

Dijkstra’s Algorithm



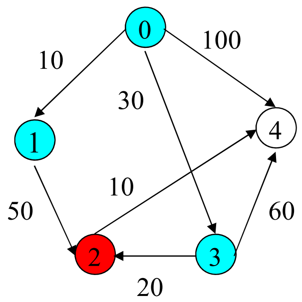
{0,0,3,0,3}

Iteration	S	w	D[1]	D[2]	D[3]	D[4]
initial	{0}	-	10	∞	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90

D[2] = 60 > D[3] + C[3][2] = 30+20 = 50

D[4] = 100 < D[3] + C[3][4] = 30+60 = 90

Dijkstra's Algorithm

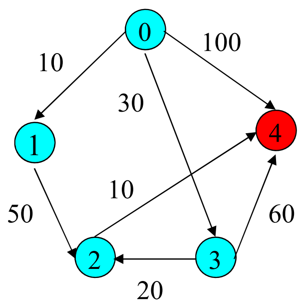


{0,0,3,0,2}

Iteration	S	w	D[1]	D[2]	D[3]	D[4]
initial	{0}	-	10	∞	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90
3	{0,1,3,2}	2	10	50	30	60

D[4] = 90 < D[2] + C[2][4] = 50+10 = 60

Dijkstra's Algorithm

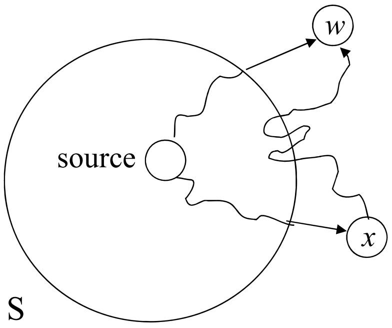


{0,0,3,0,2}

Iteration	S	w	D[1]	D[2]	D[3]	D[4]
initial	{0}	-	10	∞	30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	3	10	50	30	90
3	{0,1,3,2}	2	10	50	30	60
4	{0,1,3,2,4}	4	10	50	30	60

Why Dijkstra's Algorithm Works

There cannot be a shorter nonspecial path from the source to *w*.



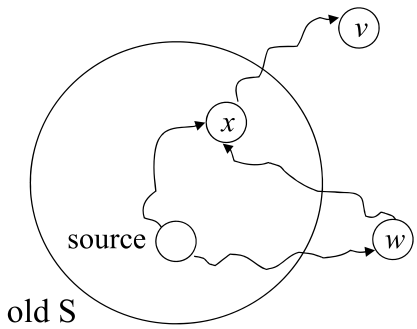
Hypothetical shorter path to *w* via *x*

```
Algorithm Dijkstra() {
   S = {0};
   for (i = 1; i < n; i++)
      D[i] = C[0][i];  // initialize D
   while ( there are remaining vertices in V-S ) {
      choose a vertex w in V-S such that D[w] is a minimum;
      add w to S;
      for ( each vertex v in V-S )
         D[v] = min(D[v], D[w] + C[w][ v]);
   }
}
```


Why Dijkstra's Algorithm Works
(Cont'd)

Whenever a new node *w* is added to *S*,
D[v] is truly the shortest distance of a special path to *v* at all times.

```
Algorithm Dijkstra() {
   S = {0};
   for (i = 1; i < n; i++)
      D[i] = C[0][i];  // initialize D
   while ( there are remaining vertices in V-S ) {
      choose a vertex w in V-S such that D[w] is a minimum;
      add w to S;
      for ( each vertex v in V-S )
         D[v] = min(D[v], D[w] + C[w][ v]);
   }
}
```



Impossible shortest special path

All-Pairs Shortest Paths Problem

- We are given a directed graph $G = (V, E)$ in which each edge $\langle v,w \rangle$ has a non-negative cost $C[v][w]$.
- The *all-pairs shortest paths (APSP)* problem is to find for each ordered pair of vertices $\langle v, w \rangle$ the smallest length of any path from v to w.

Two Alternatives

- Use Dijkstra's algorithm with each vertex in turn as the source.
- Floyd-Warshall Algorithm