The 6th Course

GRAPHS

Graphs: Definitions

- A ***graph G*** consists of a set V , whose members are called the ***vertices*** of G, together with a set E of pairs of distinct vertices from V .
- The pairs in E are called the ***edges*** of G.
- If e =(v, w) is an edge with vertices v and w, then v and w are said to ***lie on*** e, and e is said to be ***incident*** with v and w.
- If the pairs are unordered, G is called an ***undirected graph***.

Graphs: Definitions

- If the pairs are ordered, G is called a ***directed graph***. The term directed graph is often shortened to ***digraph***, and the unqualified term graph usually means undirected graph.
- Two vertices in an undirected graph are called ***adjacent*** if there is an edge from the first to the second.
- A ***path*** is a sequence of **distinct** vertices, each adjacent to the next.

Graphs: Definitions

- A ***cycle*** is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.
- A graph is called ***connected*** if there is a path from any vertex to any other vertex.
- A ***free tree*** is defined as a connected undirected graph with no cycles.
- In a directed graph a path or a cycle means always moving in the direction indicated by the arrows. Such a path (cycle) is called a directed path (cycle).

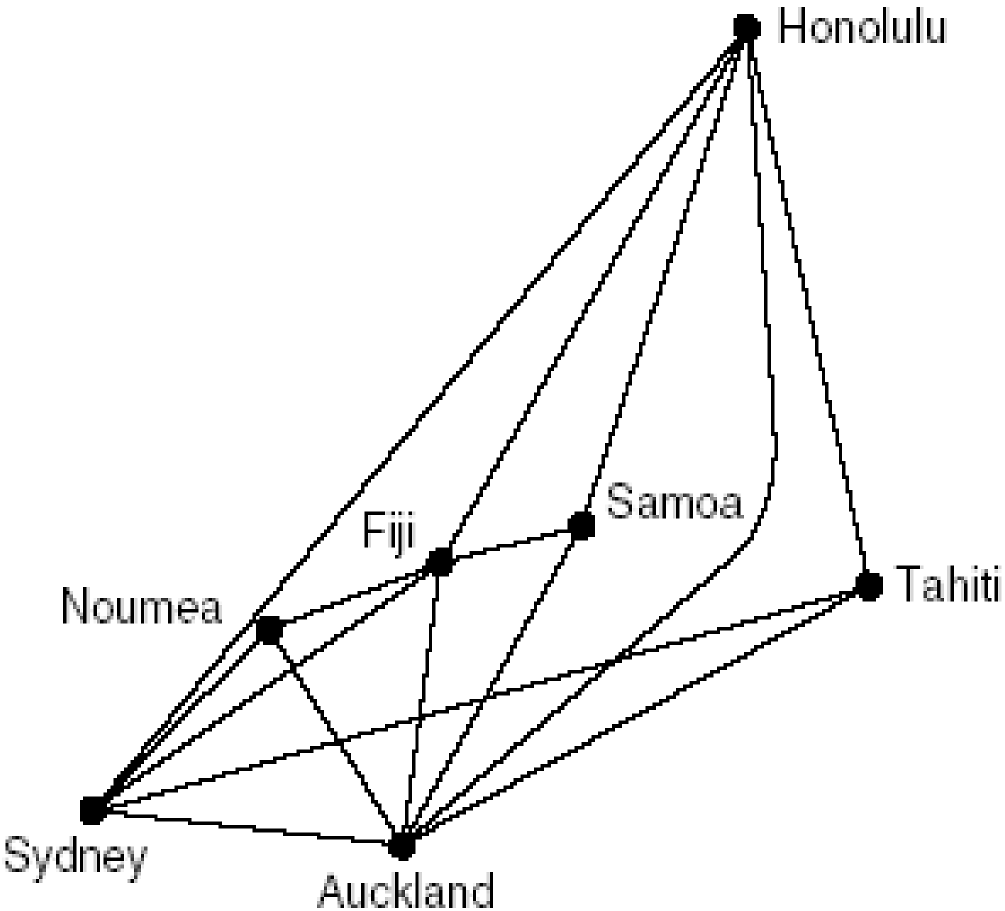
Graphs: Definitions

- *subgraph*.
- The maximal connected subgraphs of an undirected graph are called *connected components*.
- A graph without cycles is *acyclic*.
- A directed graph without cycles is a directed *acyclic graph* or *DAG*.

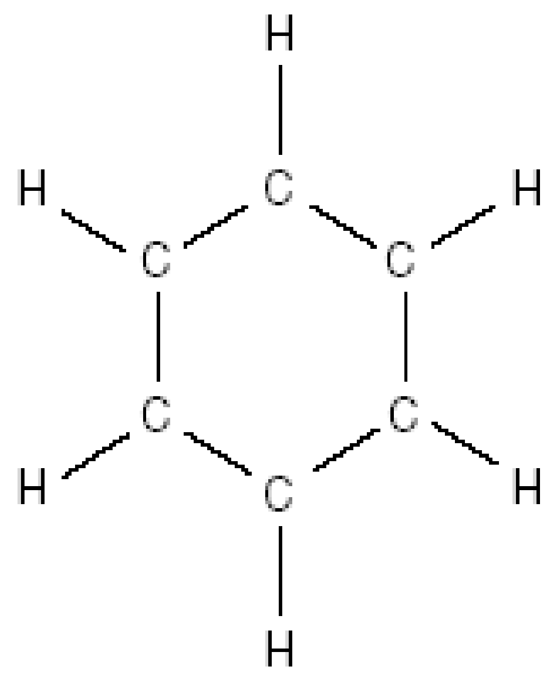
Graphs: Definitions

- A directed graph is called ***strongly connected*** if there is a directed path from any vertex to any other vertex. If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph ***weakly connected***.
- The ***valence*** of a vertex is the number of edges on which it lies, hence also the number of vertices adjacent to it.

Examples of Graphs

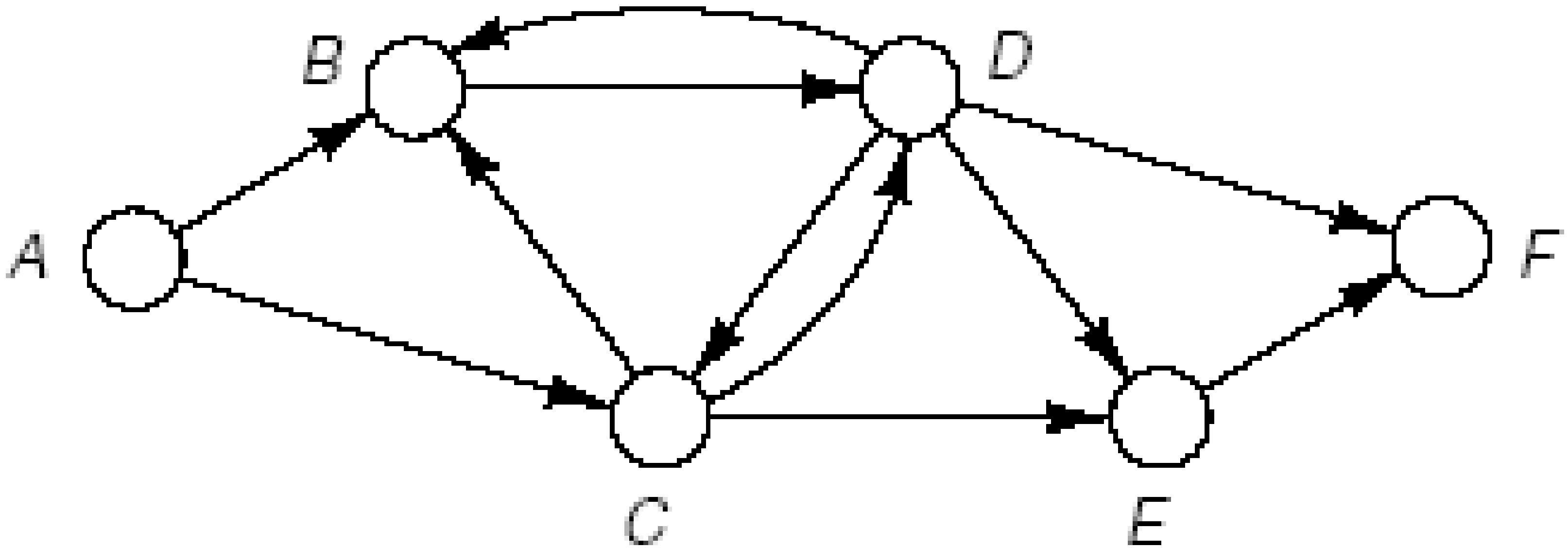


Selected South Pacific air routes



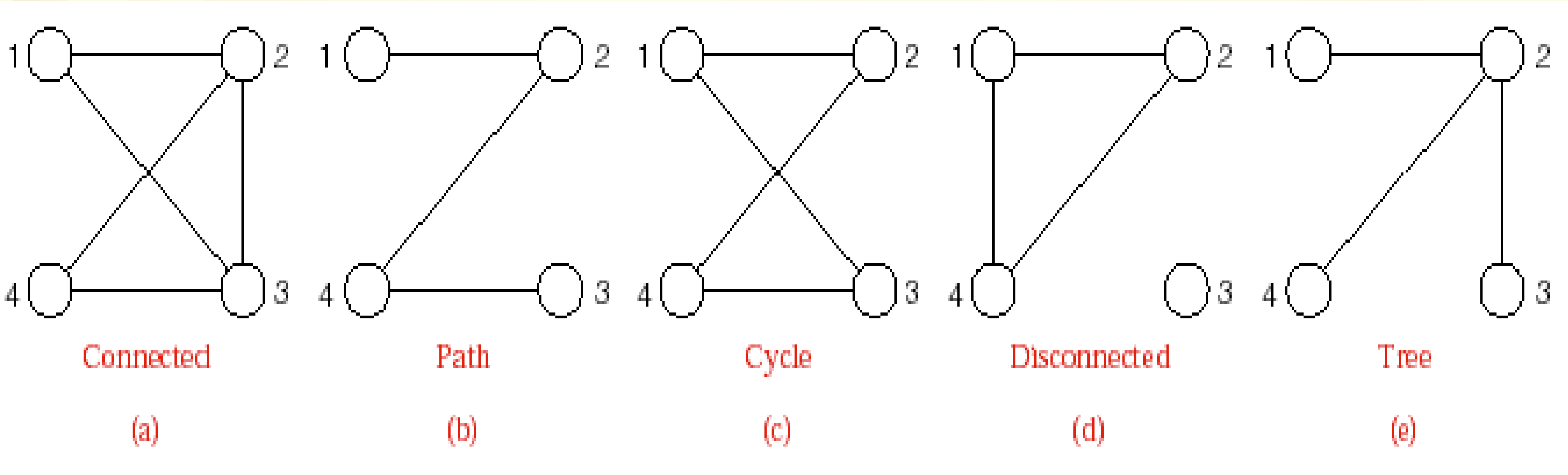
Benzene molecule

Examples of Graphs

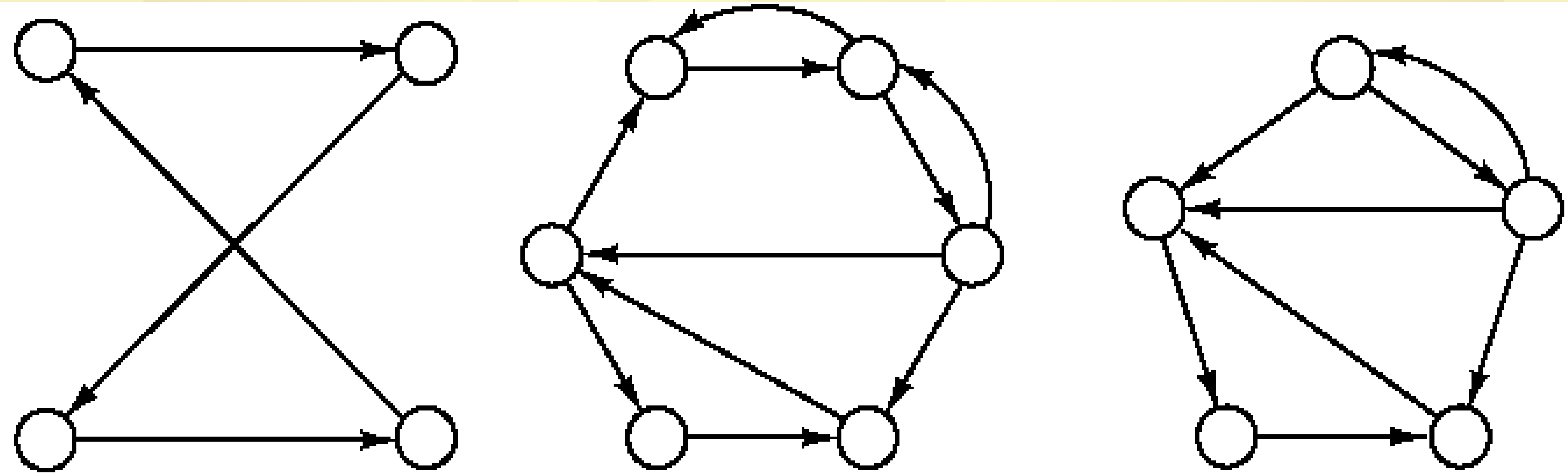


Message transmission in a network

Examples of Graphs



Examples of Graphs



Directed cycle

(a)

Strongly connected

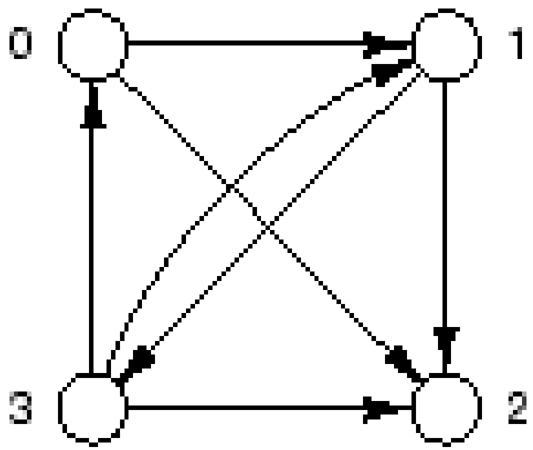
(b)

Weakly connected

(c)

List Implementation of Digraphs

Directed graph

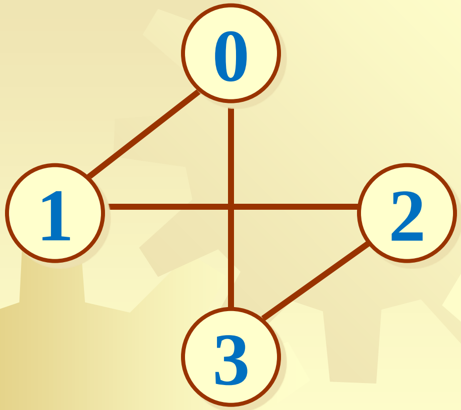


Adjacency sets

vertex	Set
0	{ 1, 2 }
1	{ 2, 3 }
2	Ø
3	{ 0, 1, 2 }

Adjacency table

	0	1	2	3
0	F	T	T	F
1	F	F	T	T
2	F	F	F	F
3	T	T	T	F



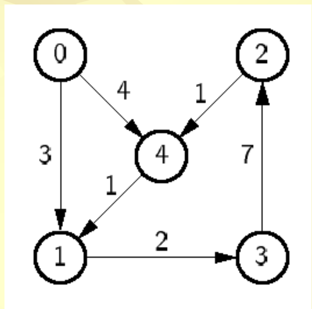
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

undirected graph_Symmetric matrix

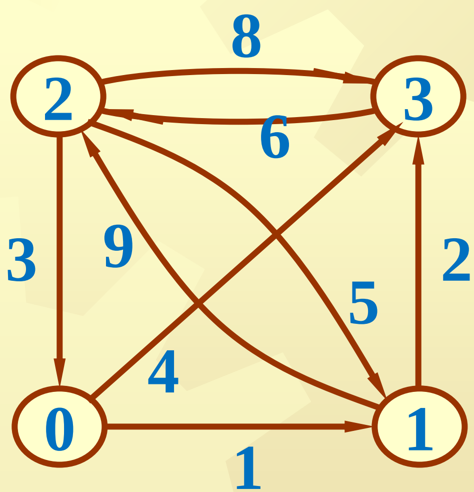
$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

directed graph_Non Symmetric matrix

- Instead of bits, the graph could store edge, weights.

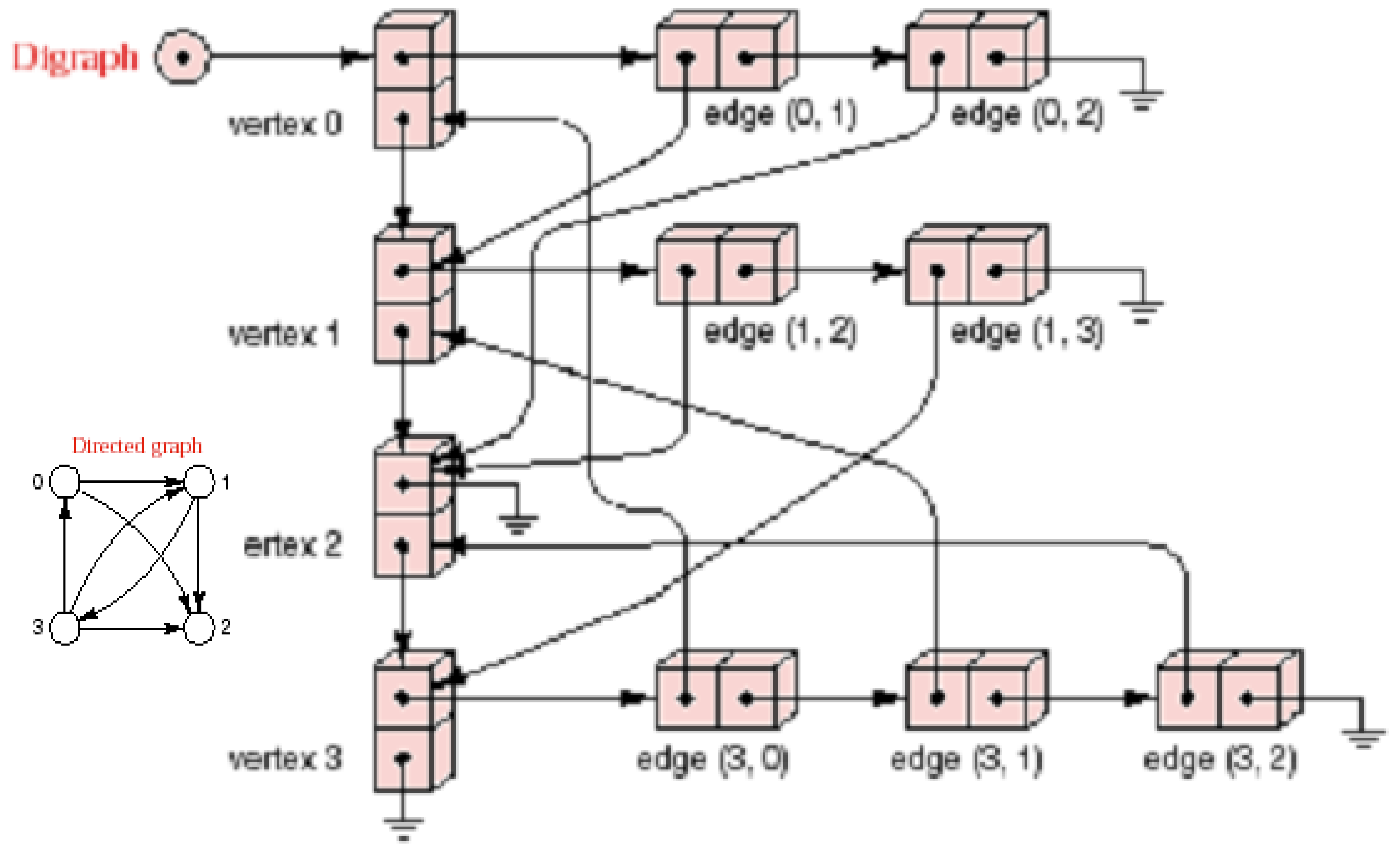


$$
\begin{array}{c}
0\\ 1\\ 2\\ 3\\ 4
\end{array}
\begin{bmatrix}
 & 3 & & & 4 \\
 & & & 2 & \\
 & & & & 1 \\
 & & 7 & & \\
 & 1 & & &
\end{bmatrix}
$$



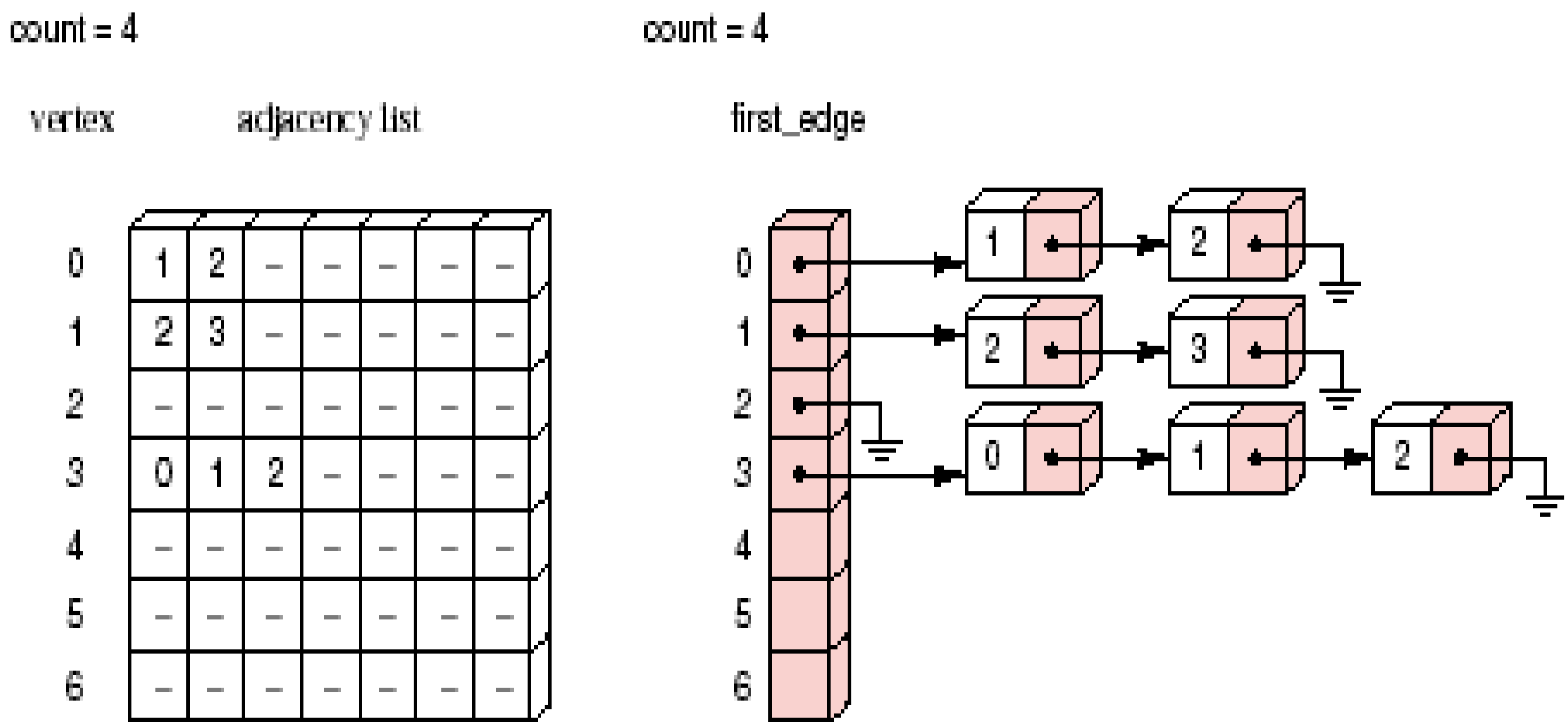
$$
A = \begin{bmatrix}
\infty & 1 & \infty & 4 \\
\infty & \infty & 9 & 2 \\
3 & 5 & \infty & 8 \\
\infty & \infty & 6 & \infty
\end{bmatrix}
$$

List Implementation of Digraphs



(a) Linked lists

List Implementation of Digraphs

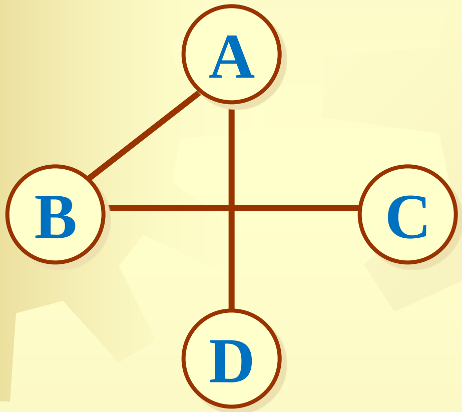


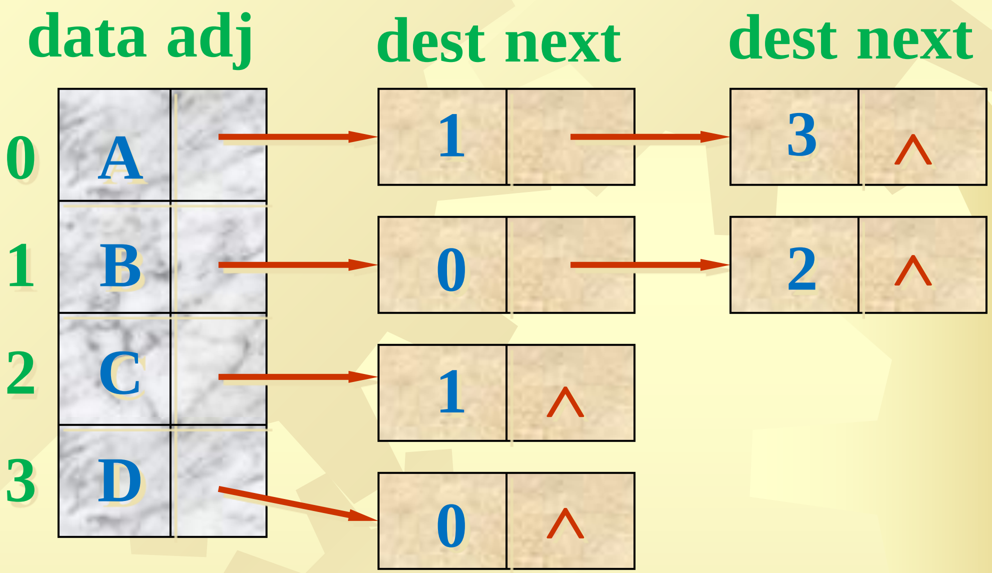
(b) Contiguous lists

(c) Mixed

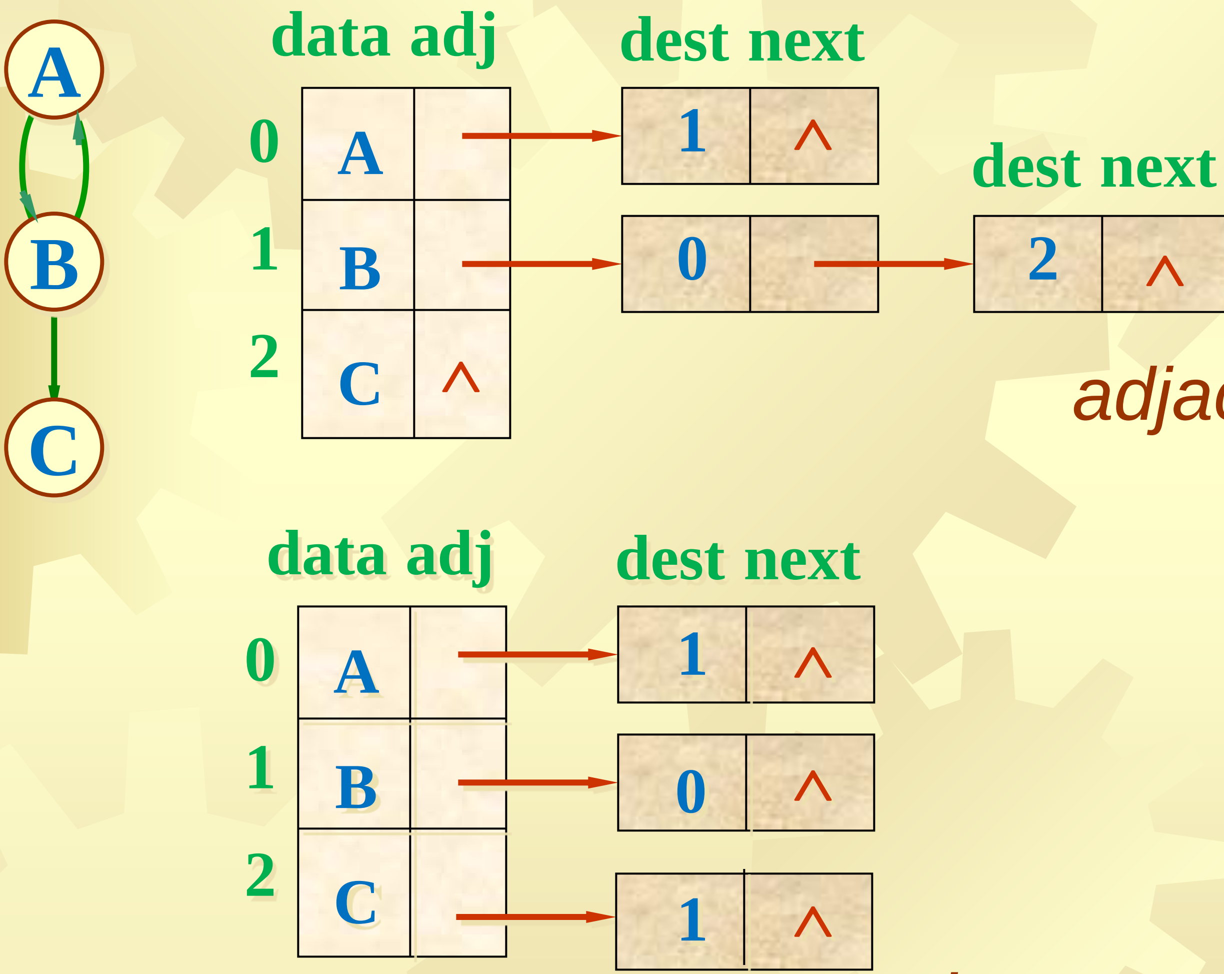
Adjacency List

- undirected graph





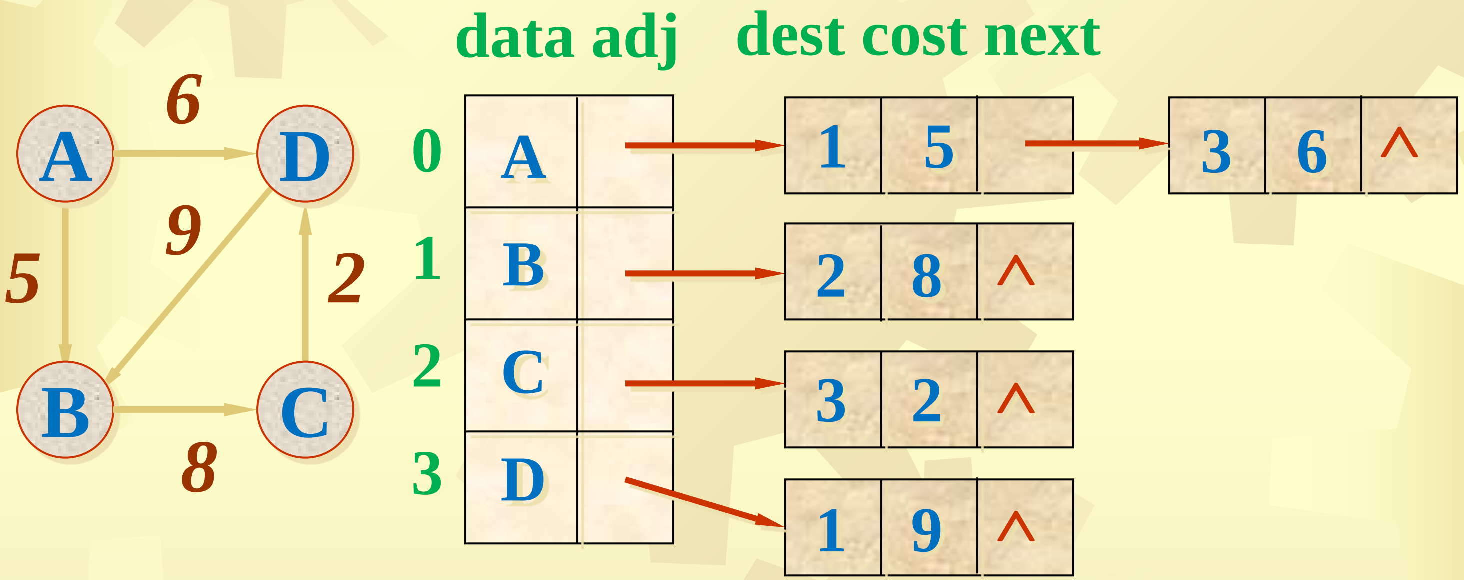
Adjacency List of directed graph



adjacency list

Inverse adjacency list

Adjacency List of Net



邻接多重表 (Adjacency Multilist)

- undirected graph
  - Edge

mark	vertex1	vertex2	path1	path2

    - mark 是处理标记；
    - vertex1和vertex2是该边两顶点位置；
    - path1 指向下一条依附 vertex1的边；
    - path2 指向下一条依附 vertex2 的边。
  - Network

mark	cost	vertex1	vertex2	path1	path2

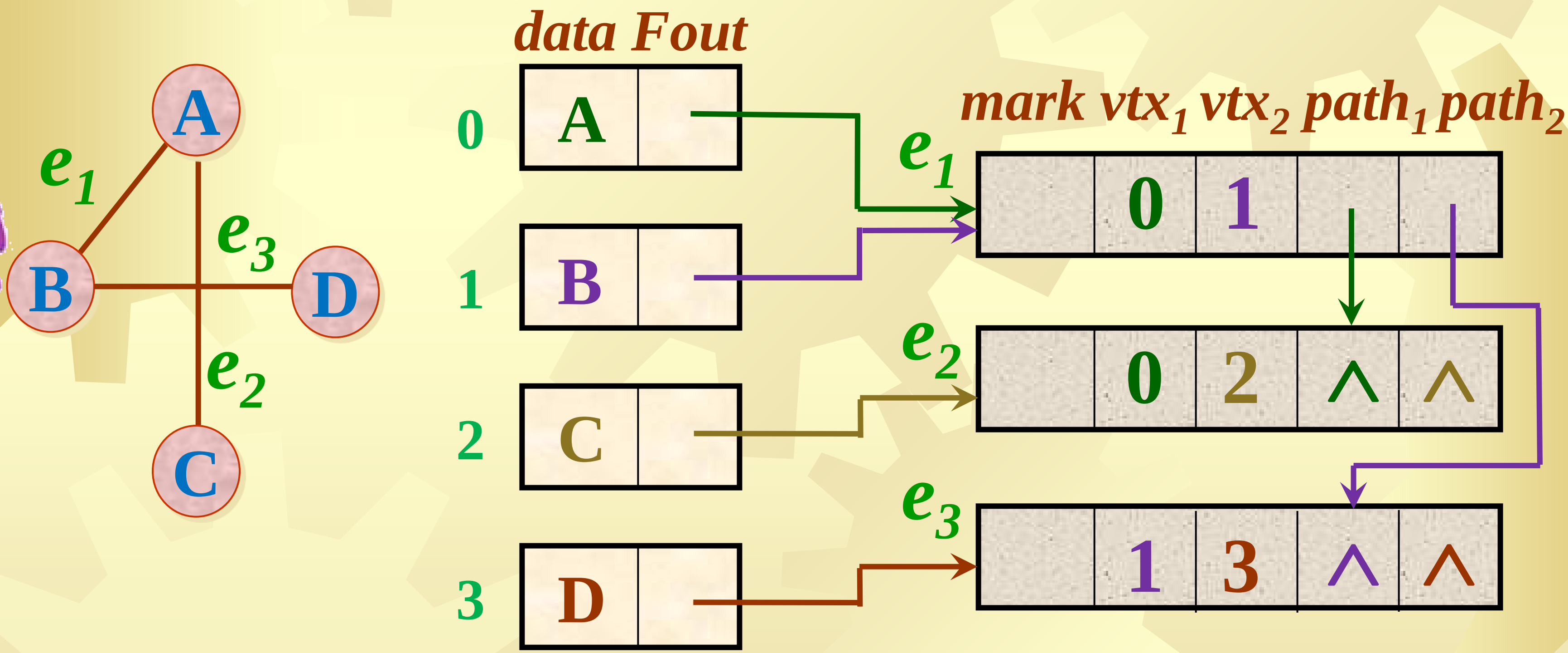
Adjacency Multilist

- **Node**

data	*Firstout*

  - data 存放与该顶点相关的信息；
  - Firstout 是指示第一条依附该顶点的边的指针。
- 在邻接多重表中，所有依附同一个顶点的边都链接在同一个单链表中。

Adjacency Multilist



Adjacency Multilist

- directed graph
  - Edge

mark	*vertex1*	*vertex2*	*path1*	*path2*

  - 其中，mark 是处理标记；vertex1 和 vertex2 指明该有向边始顶点和终顶点的位置。Path1 指向同一顶点发出的下一条边的边结点；path2 指向进入同一顶点的下一条边的边结点。
  - 需要时还可有权值域 cost。

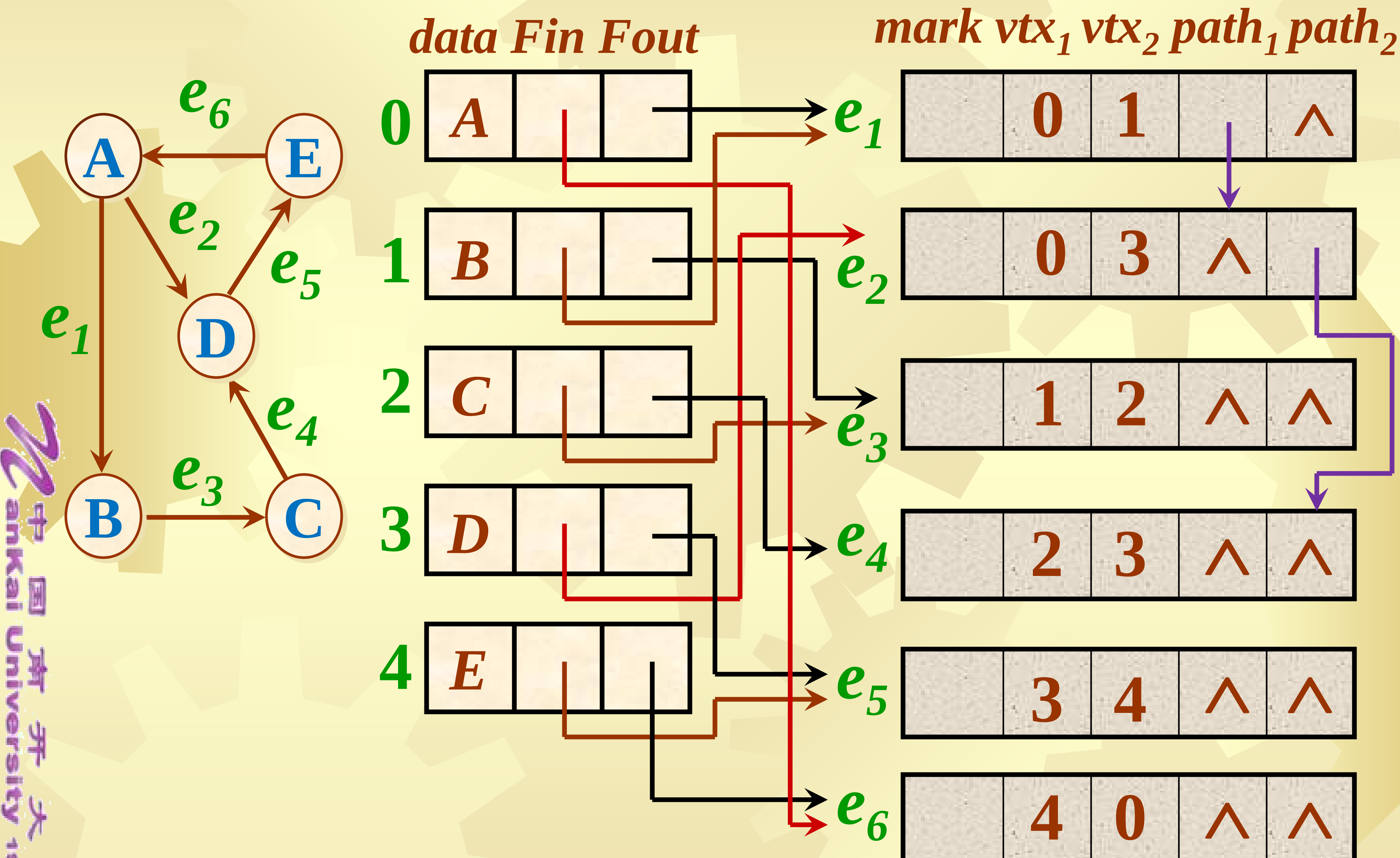
Adjacency Multilist of Degraph

- **Node**

data	*Firstin*	*Firstout*

- **data** 存放与该顶点相关的信息；
- **Firstout** 指示以该顶点为始顶点的出边表的第一条边；
- **Firstin** 指示以该顶点为终顶点的入边表的第一条边。

Adjacency Multilist

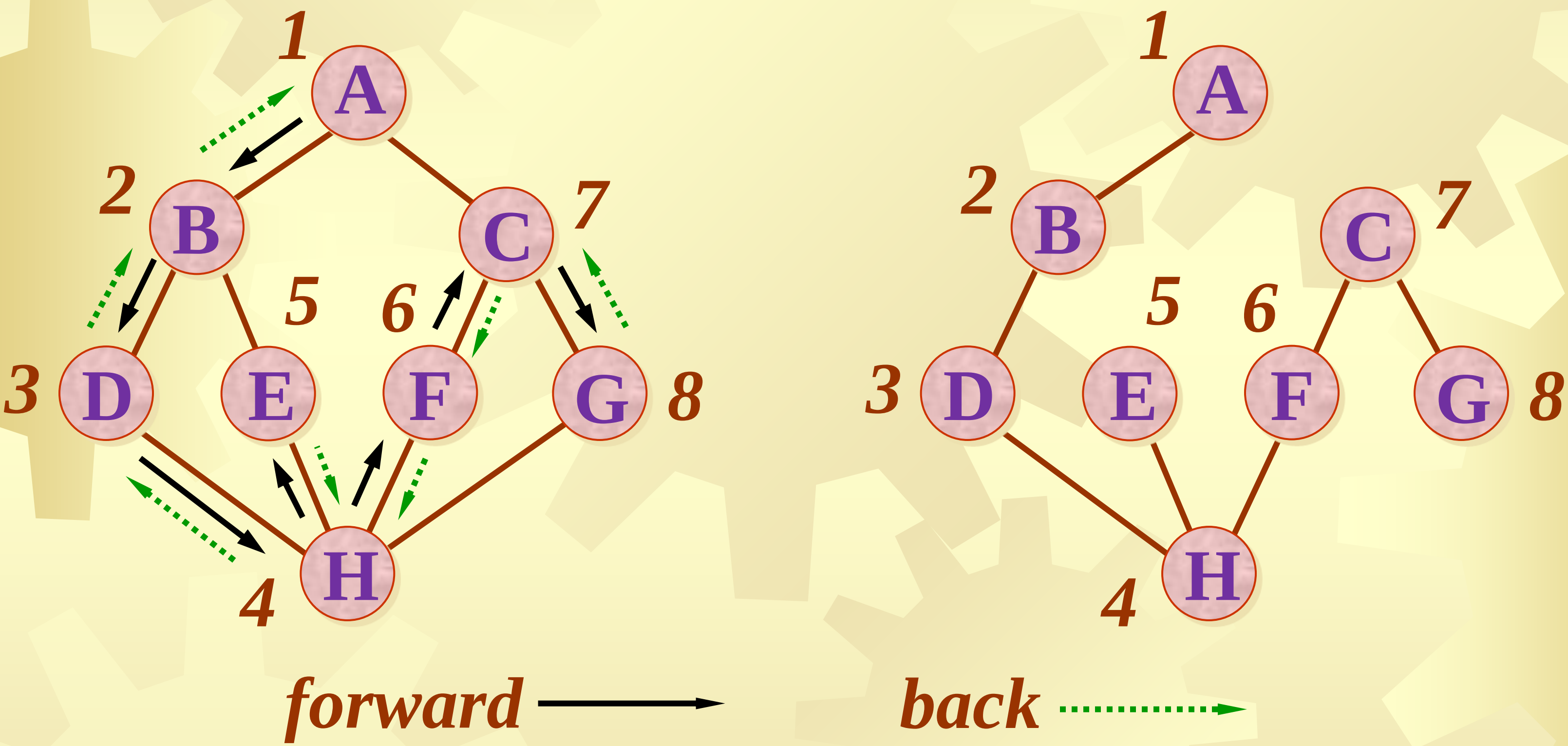


Graph Traversal

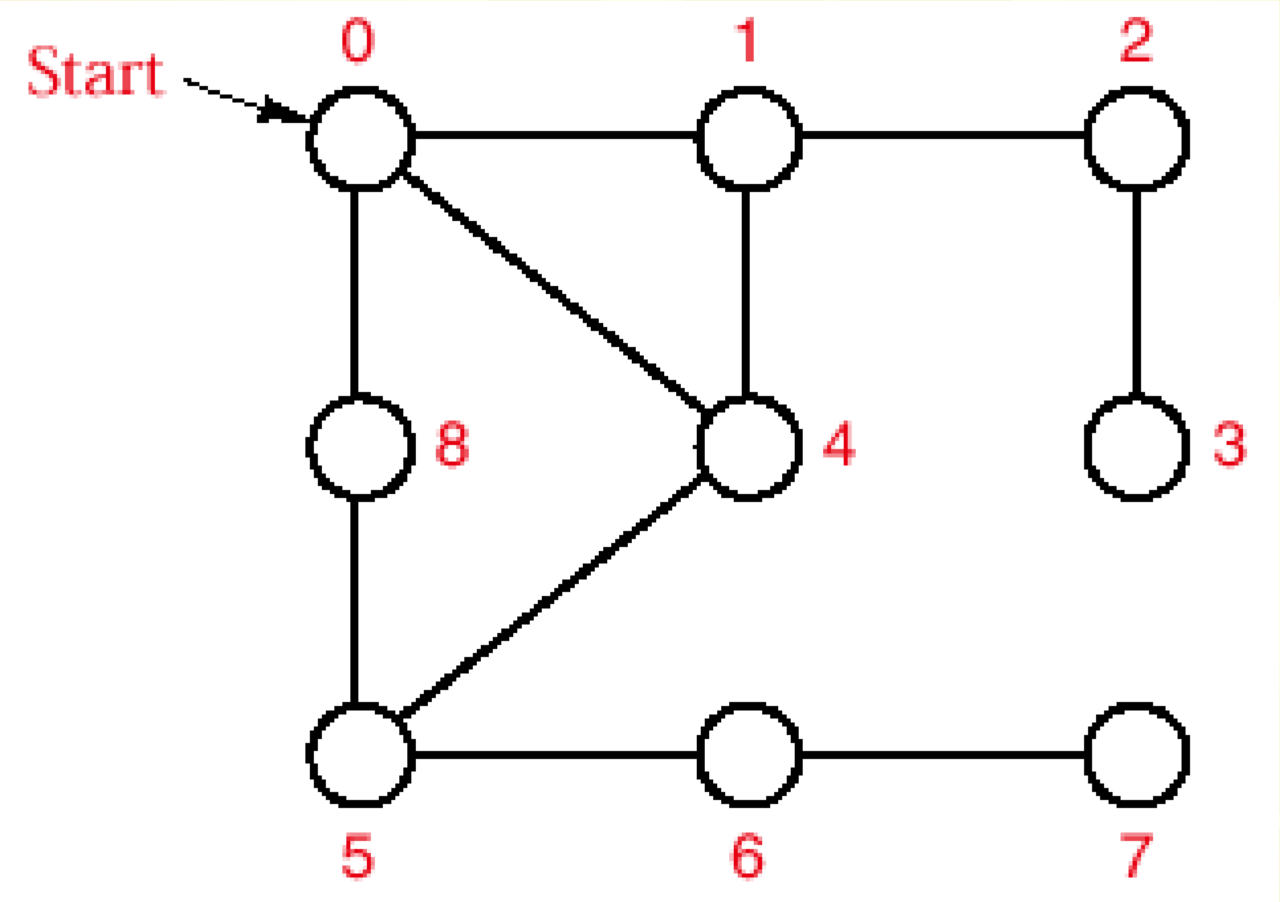
- ***Depth-first traversal*** **of a graph is roughly analogous to preorder traversal of an ordered tree. Suppose that the traversal has just visited a vertex v, and let $w_1, w_2, \ldots, w_k$ be the vertices adjacent to v. Then we shall next visit w_1 and keep w_2，…，w_k waiting. After visiting w_1 , we traverse all the vertices to which it is adjacent before returning to traverse w_2，…，w_k .**

DFS (Depth First Search)

- DFS



Graph Traversal-DFT



Depth-First Algorithm

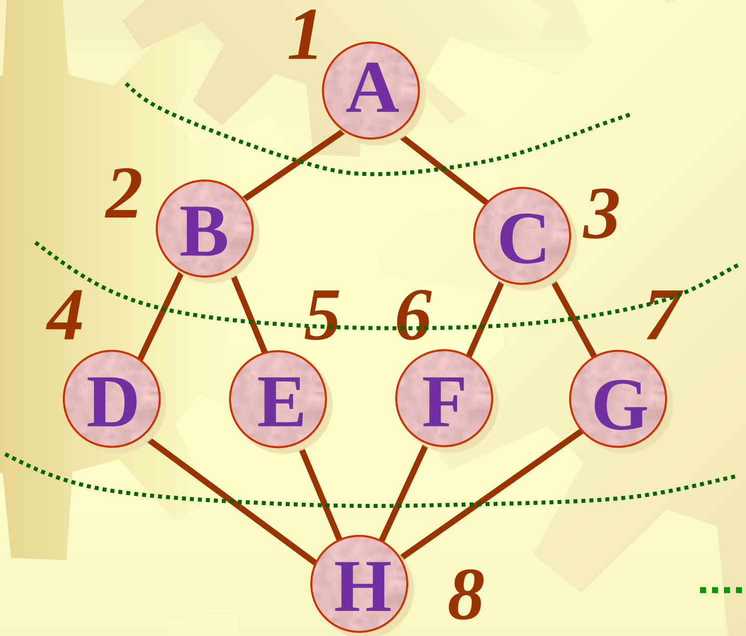
- bool visited[max_size]
- The recursion is performed
- using stack
- get connected components

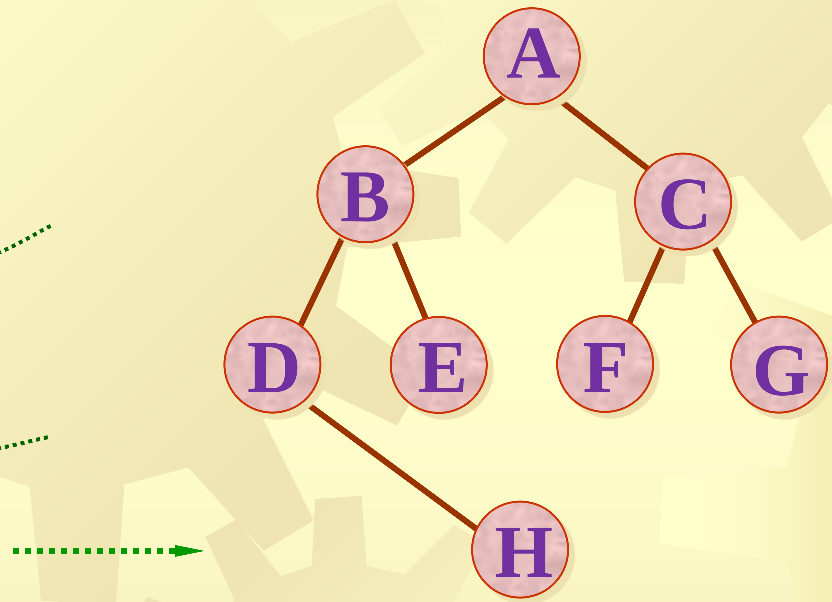
Graph Traversal

- ***Breadth-first traversal*** **of a graph is roughly analogous to level-by-level traversal of an ordered tree. If the traversal has just visited a vertex v, then it next visits all the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to v have been visited.**

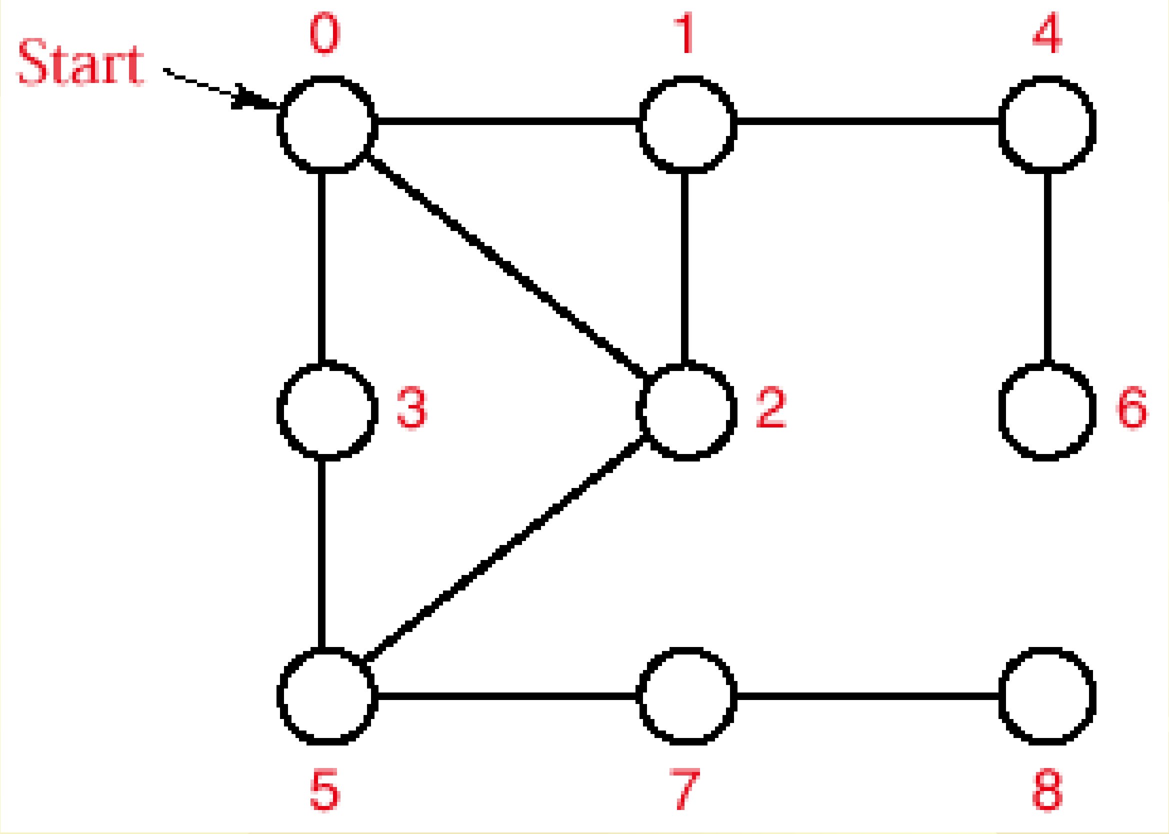
BFS (Breadth First Search)

- **BFS**





Graph Traversal-BFT

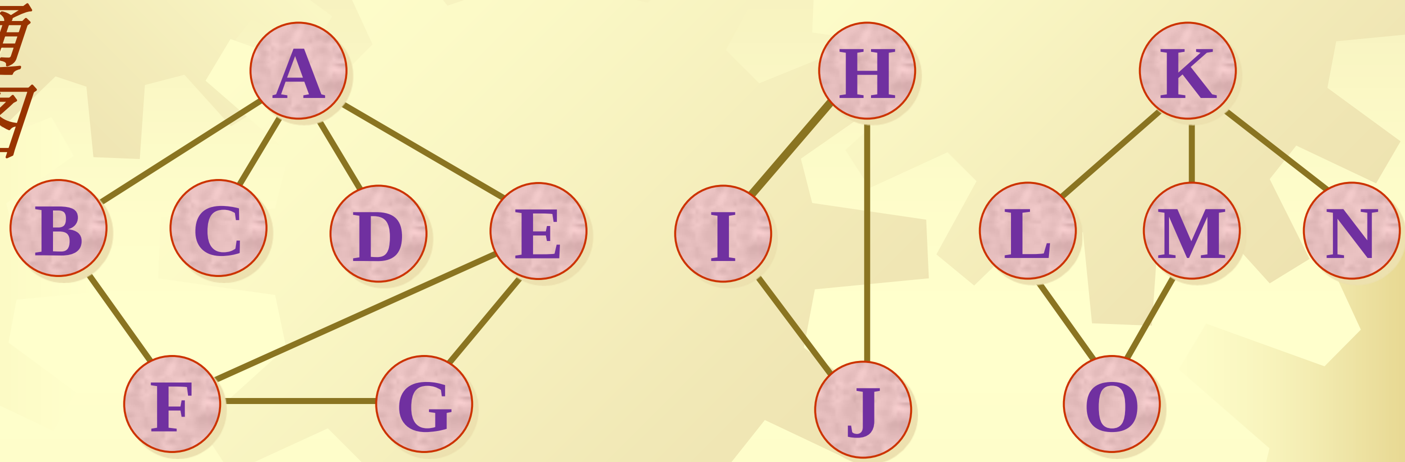


Breadth-First Algorithm

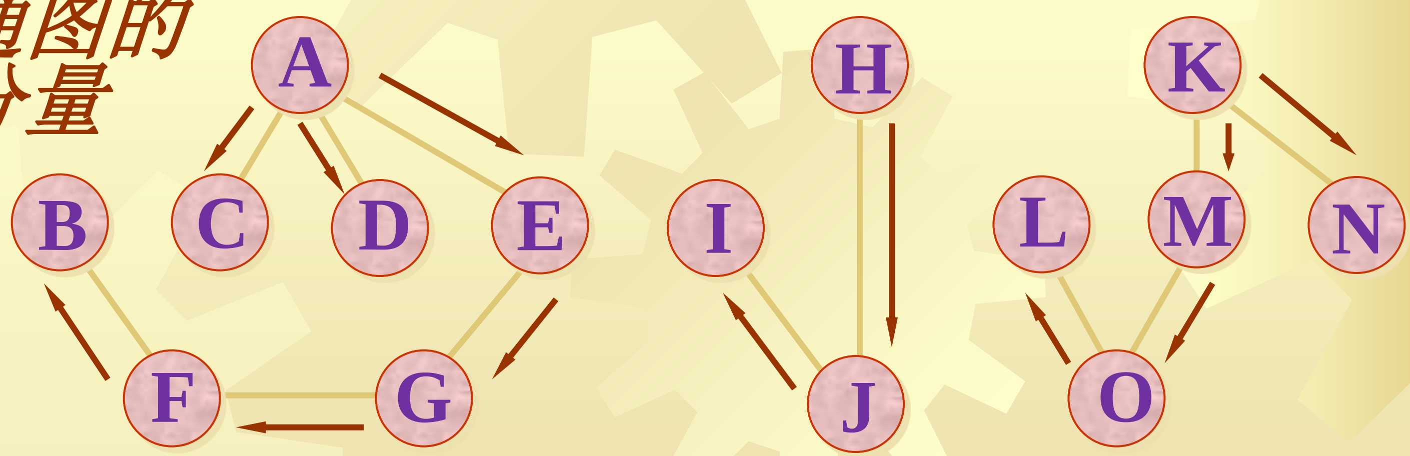
- bool visited[max_size]
- Is a iteration function
- using queue
- get connected components

- 对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。

非连通无向图



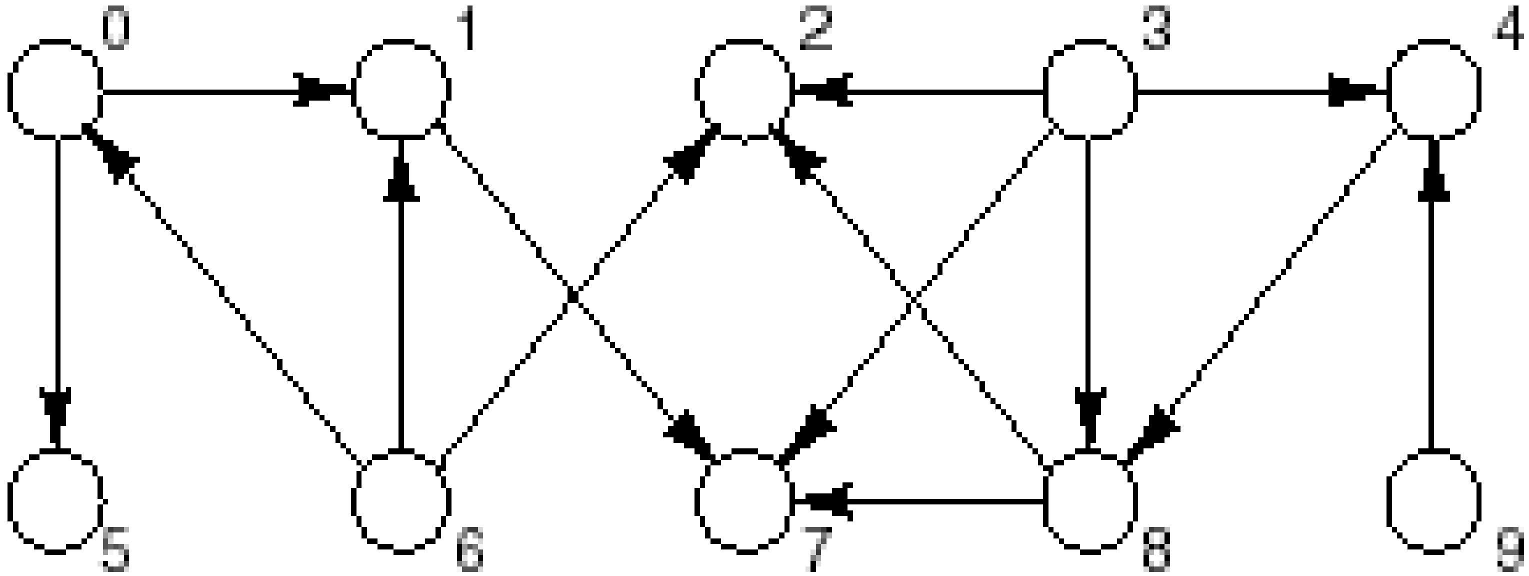
非连通图的连通分量



Topological Sort

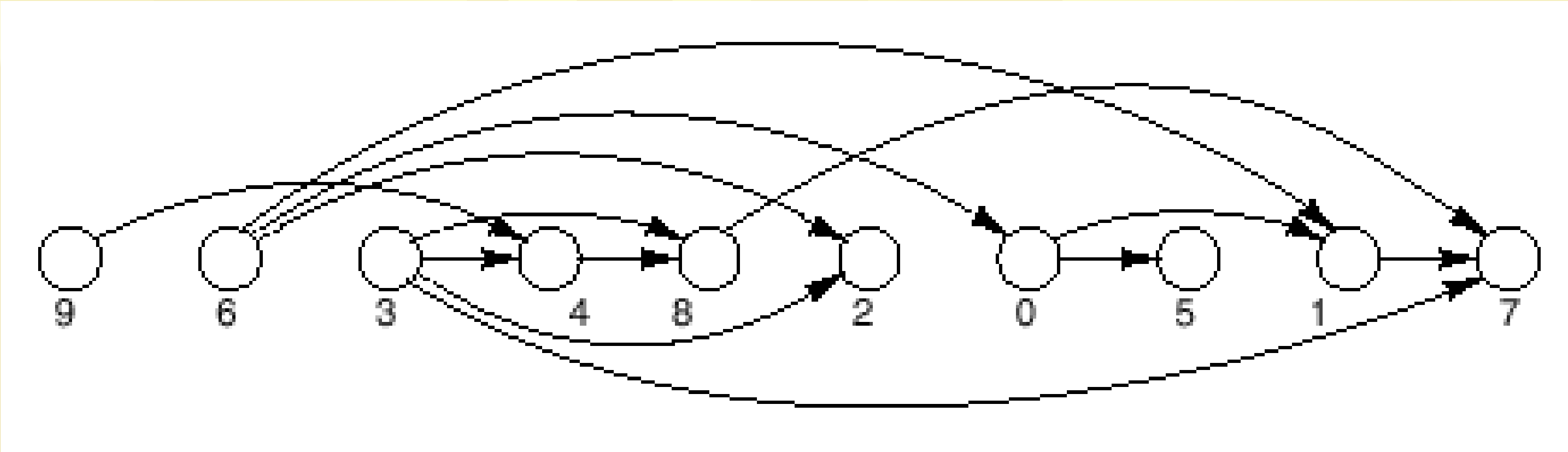
- Let G be a directed graph with no cycles. A ***topological order*** for G is a sequential listing of all the vertices in G such that, for all vertices v , w $\in$ G, if there is an edge from v to w, then v precedes w in the sequential listing.

Topological Sort



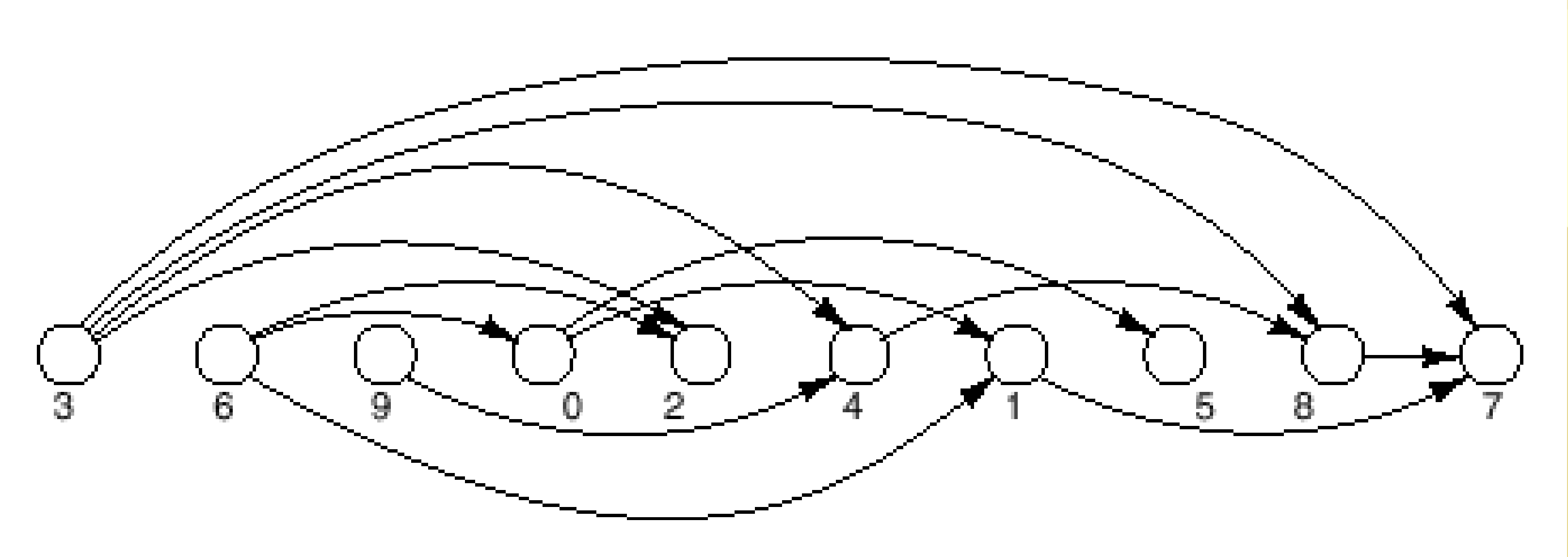
Directed graph with no directed cycles

Depth-First Ordering



- Using DFT;
- Output all successors, then output itself;
- Reverse the sequential listing

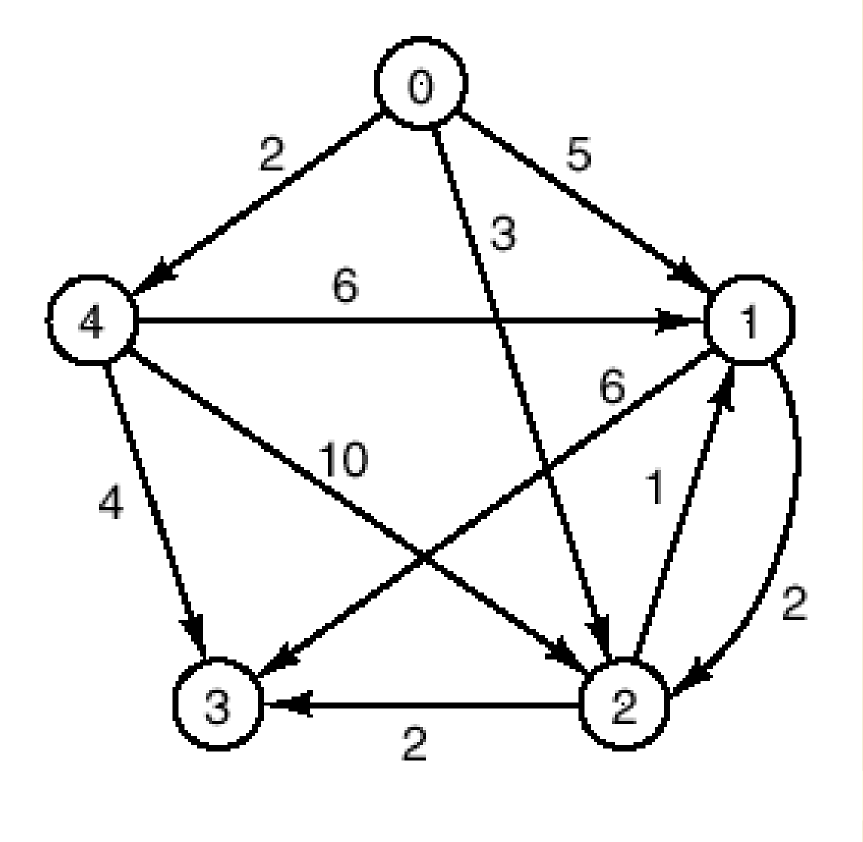
Breadth-First Ordering



- Using predecessor_count
- If it equals to 0, then output the vertice

A Greedy Algorithm: Shortest Paths

- The problem of ***shortest paths***
  - Given a directed graph in which each edge has a nonnegative weight or cost, and a path of least total weight from a given vertex, called the ***source***, to every other vertex in the graph.
- Single Source Shortest Paths
  - Dijkstra's Algorithm
- All Pairs Shortest Paths
  - Floyd's Algorithm



- 0→1: 4 0 →2 →1
- 0→2: 3 0 →2
- 0→3: 5 0 →2 →3
- 0→4: 2 0 →4

Finding a Shortest Path

- We keep a set S of vertices whose closest distances to the source, vertex 0, are known and add one vertex to S at each stage.
- We maintain a table distance(dist[v]) that gives, for each vertex v, the distance from 0 to v along a path all of whose vertices are in S, except possibly the last one.
  - dist[v] = distance(0 to v) along a special path by now
- Initially
  - dist[i] = cost[v_0][v_i] $v_i \in V$
  - cost is the adjacency matrix of the G

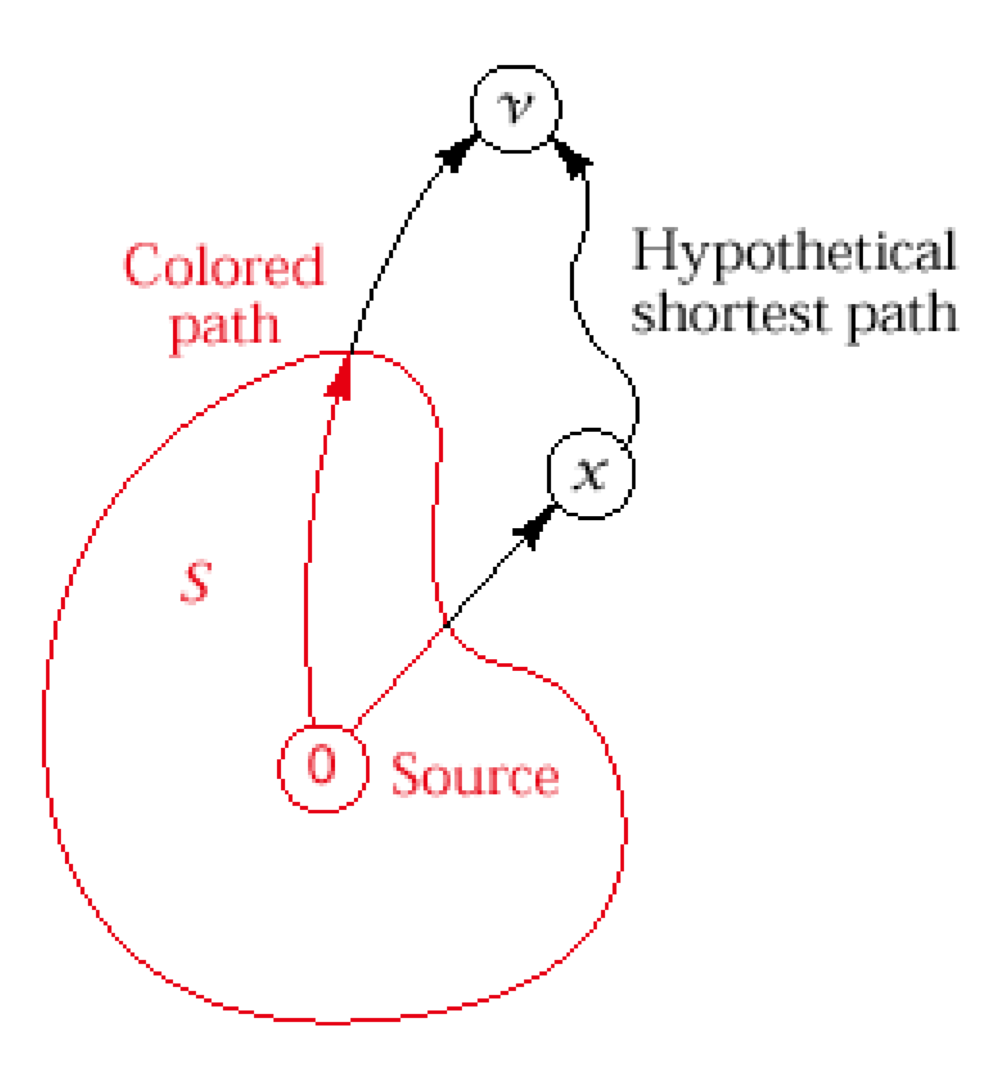
Finding a Shortest Path

- **To determine what vertex to add to S at each step, we apply the greedy criterion of choosing the vertex v with the smallest distance recorded in the table distance, such that v is not already in distance.**
- **Choose the vertex v with the smallest distance recorded in the table distance, such that v is not already in distance.**

Finding a Shortest Path

- Prove that the distance recorded in distance really is the length of the shortest path from source to v.
- For suppose that there were a shorter path from source to v, such as shown below. This path first leaves S to go to some vertex x, then goes on to v (possibly even reentering S along the way). But if this path is shorter than the colored path to v, then its initial segment from source to x is also shorter, so that the greedy criterion would have chosen x rather than v as the next vertex to add to S, since we would have had distance[x] < distance[v]

Finding a Shortest Path



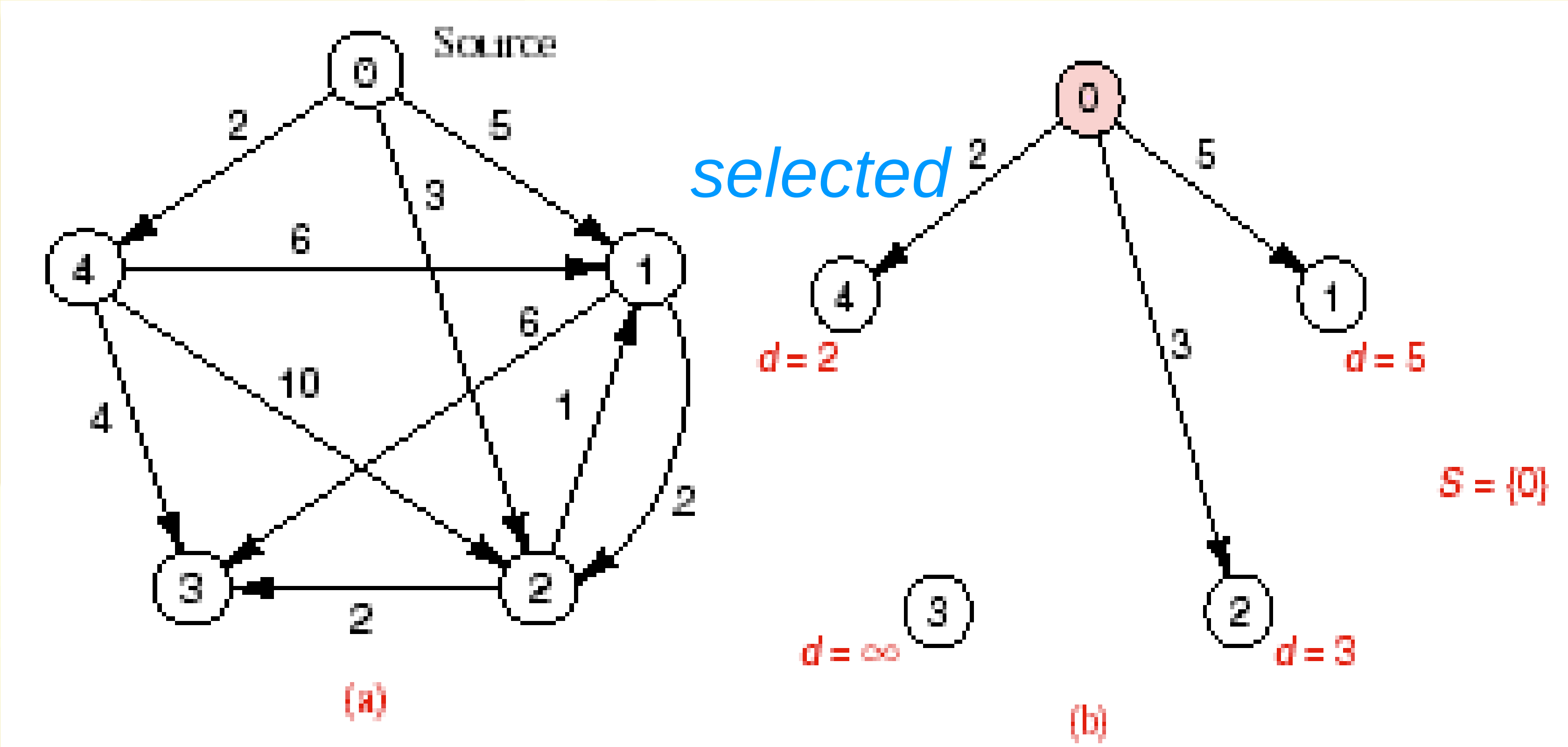
Finding a Shortest Path

- When we add v to S, we think of v as now colored and also color the shortest path from source to v.
- Next, we update the entries of distance by checking, for each vertex w not in S, whether a path through v and then directly to w is shorter than the previously recorded distance to w.

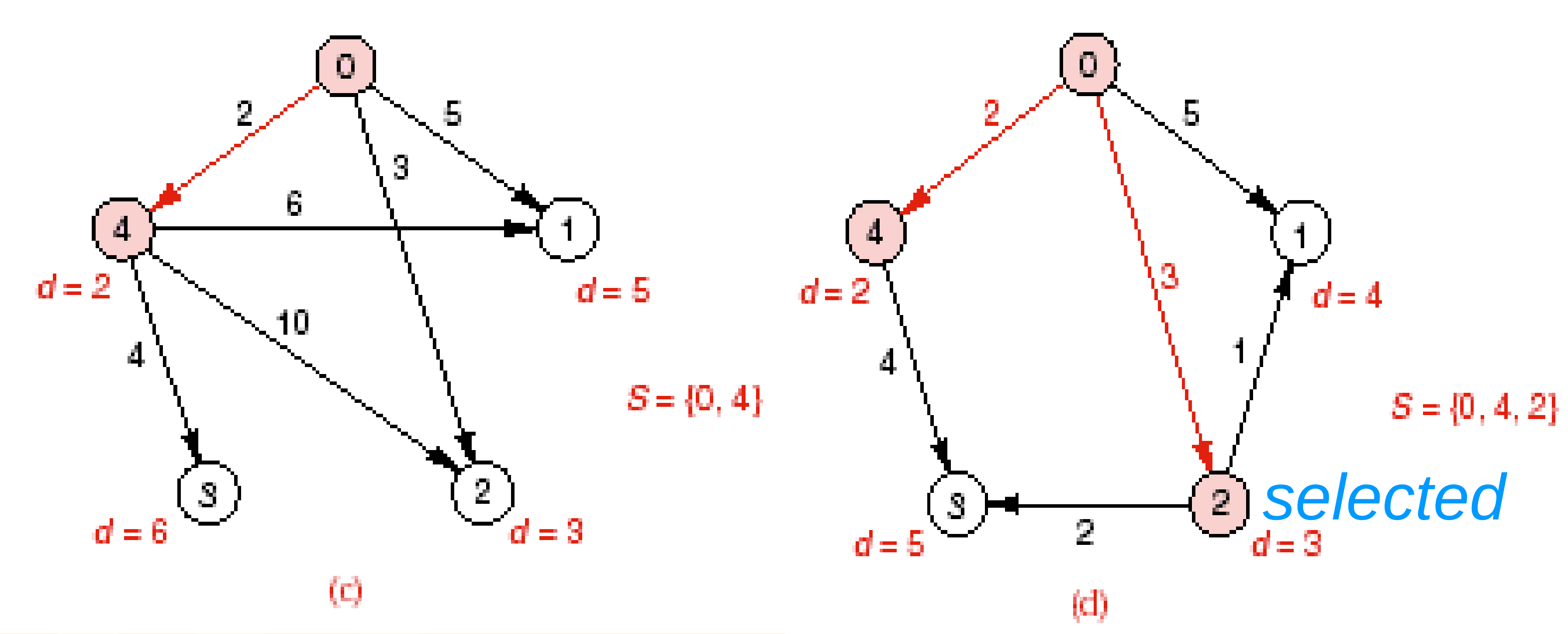
if dist[j] + cost[j][k] < dist[k]

then dist[k] = dist[j] + cost[j][k]

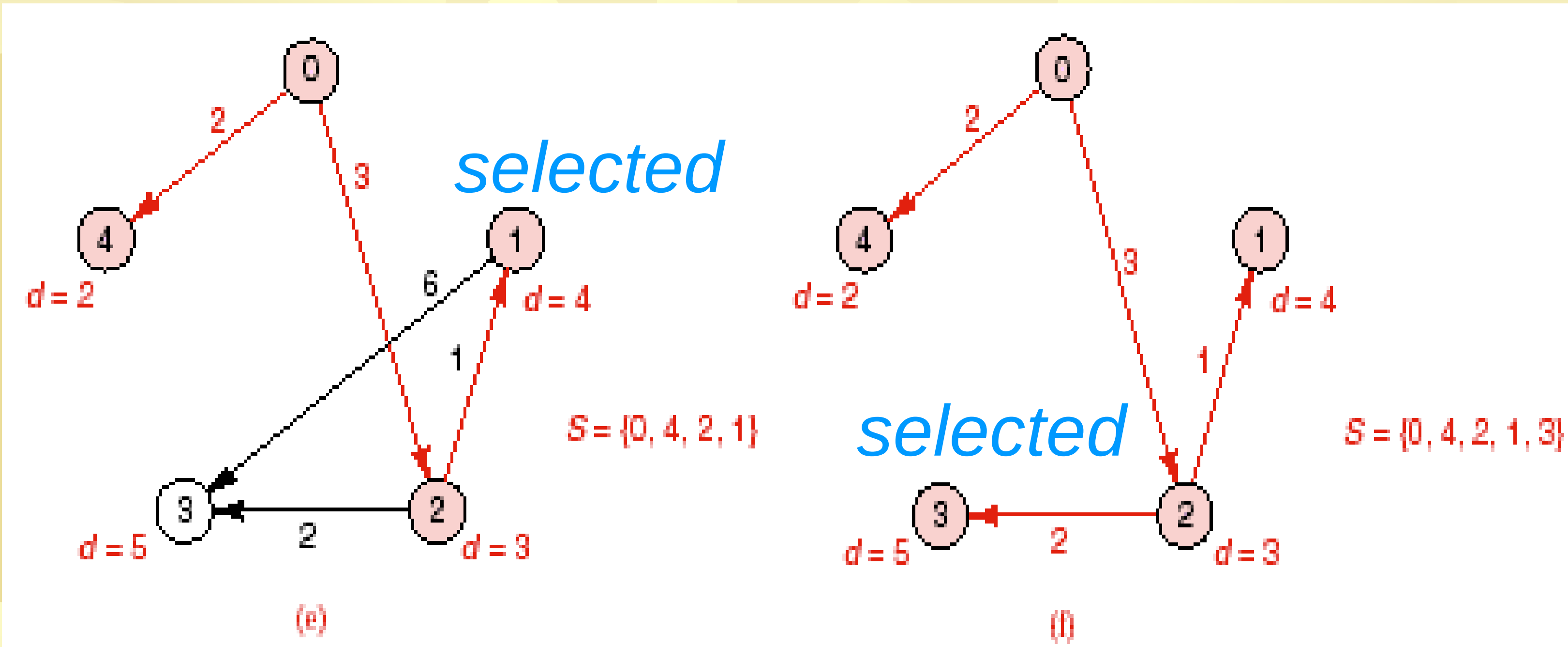
Example of Shortest Path



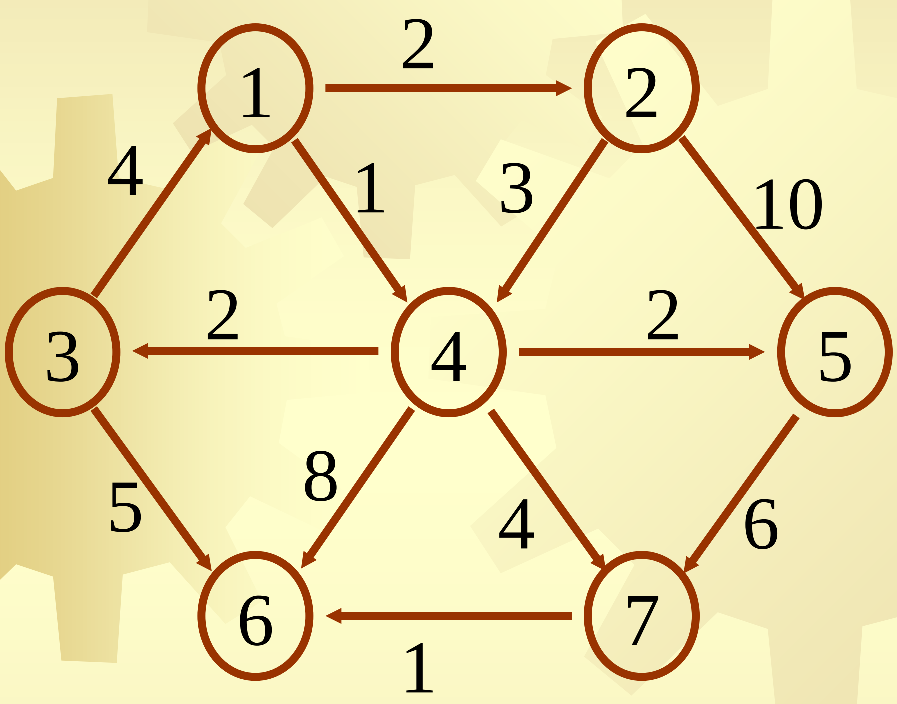
Example of Shortest Path



Example of Shortest Path

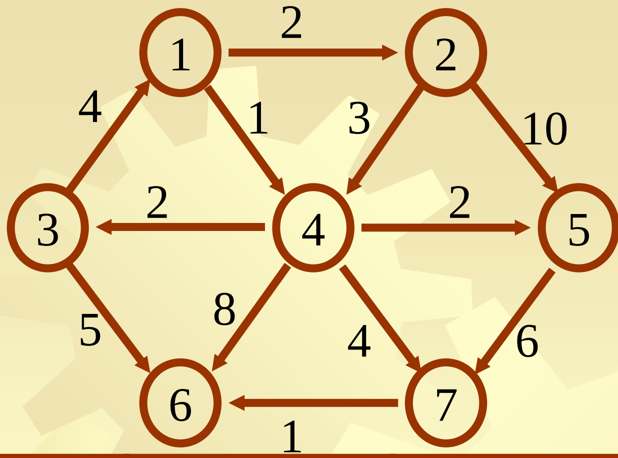


Example of Shortest Path



Adj matrix with costs

$$\begin{bmatrix} \infty & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 3 & 10 & \infty & \infty \\ 4 & \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & 2 & \infty & 2 & 8 & 4 \\ \infty & \infty & \infty & \infty & \infty & \infty & 6 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty \end{bmatrix}$$



v_2	2 (v_1, v_2)	2 (v_1, v_2)				
v_3	∞	3 (v_1,v_4,v_3)	3 (v_1,v_4,v_3)			
v_4	1 (v_1,v_4)					
v_5	∞	3 (v_1,v_4,v_5)	3 (v_1,v_4,v_5)	3 (v_1,v_4, v_5)		
v_6	∞	9 (v_1,v_4,v_6)	9 (v_1,v_4,v_6)	8 (v_1,v_4,v_3,v_6)	8 (v_1,v_4,v_3,v_6)	6 (v_1,v_4,v_7,v_6)
v_7	∞	5 (v_1,v_4,v_7)	5 (v_1,v_4,v_7)	5 (v_1,v_4, v_7)	5 (v_1,v_4,v_7)	
v_j	v_4	v_2	v_3	v_5	v_7	v_6

Algorithm of Shortest Path

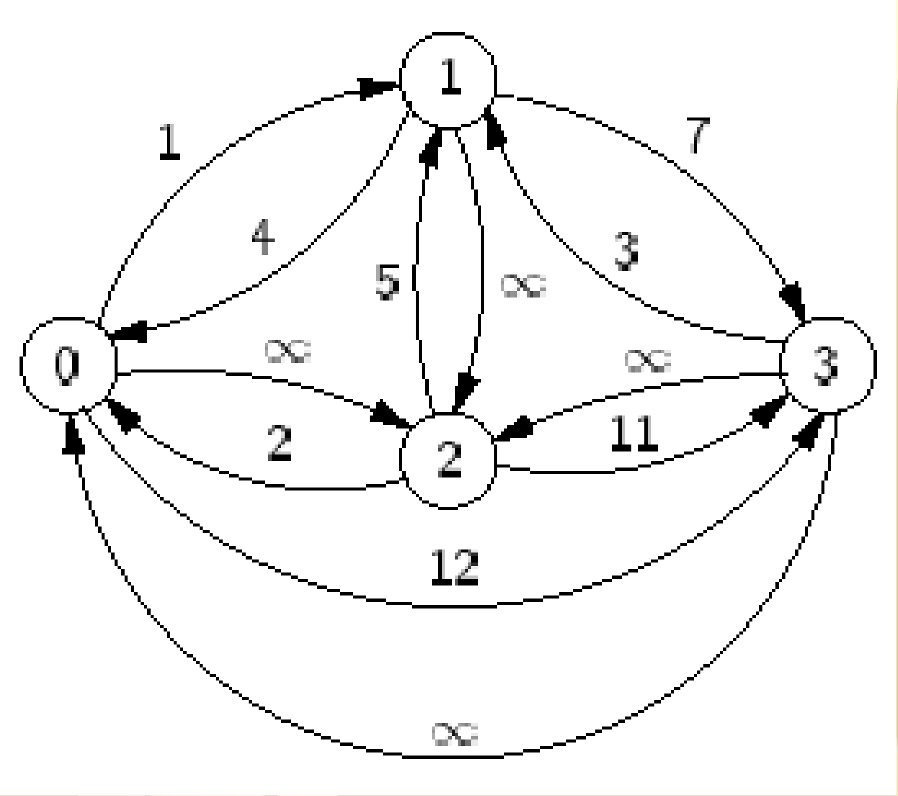
```
distance[v] = adjacency[source][v];
min = distance[w];

for (w = 0; w < count; w++)
   if (!found[w])
     if (min +adjacency[v][w] < distance[w])
        distance[w] = min + adjacency[v][w];
```

All Pairs Shortest Paths

- **For every vertex u, v∈V, calculate d(u, v).**
- **Could run Dijkstra's Algorithm |V| times.**
- **Better is Floyd's Algorithm.**

- Define a ***k-path*** from u to v to be any path whose intermediate vertices all have indices less than k.

- 0,3 is a 0-path.
- 2,0,3 is a 1-path.
- 0,2,3 is a 3-path, but not a 2- or 1-path.
- Everything is a 4-path.



- **Definition:**
  - $A^{(k)}[i][j]$ is the distance of k-path from v_i to v_j
- **iterative:**
  - $A^{(0)}[i][j] = cost[i][j]$
  - $A^{(k)}[i][j] =$

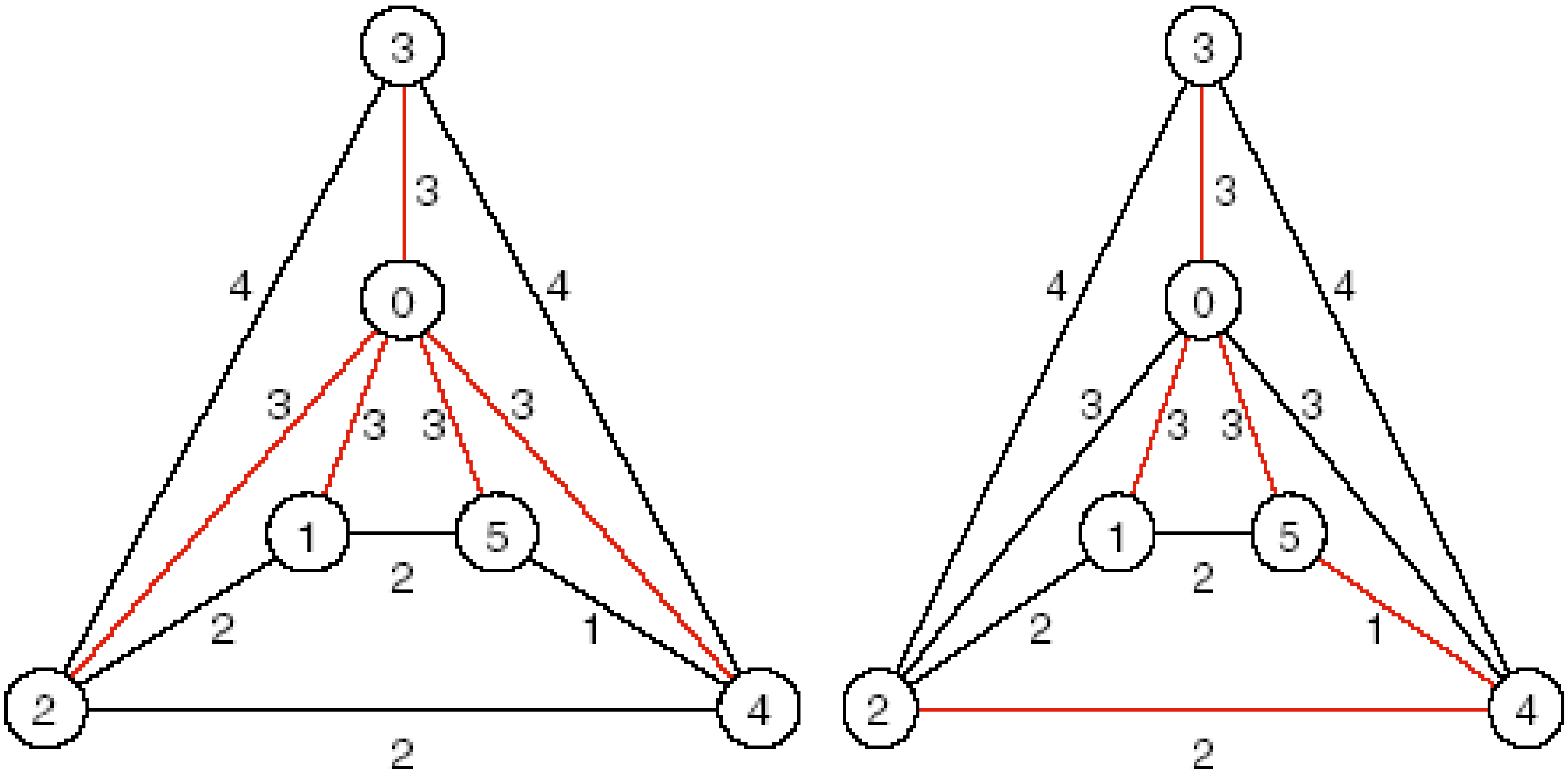
    $$\min(A^{(k-1)}[i][j],\ A^{(k-1)}[i][k] + A^{(k-1)}[k][j])$$

    $$(\text{for } 1 \le k \le n)$$
- **result:**
  - $A^{(n)}[i][j]$ is the distance of shortest path from v_i to v_j

Minimal Spanning Trees

- If the original network is based on a connected graph G, then the shortest paths from a particular source vertex to all other vertices in G form a tree that links up all the vertices of G.
- A (connected) tree that is build up out of all the vertices and some of the edges of G is called a ***spanning tree*** of G.

Two Spanning Trees

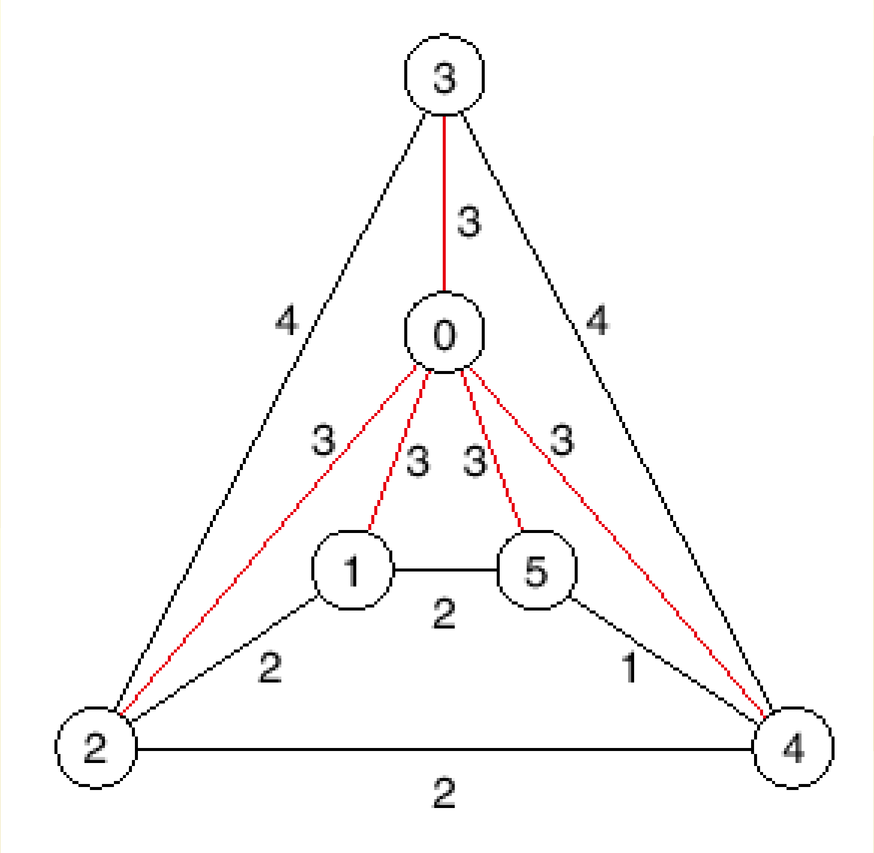


Weight sum of tree = 15
(a)

Weight sum of tree = 12
(b)

Minimal Spanning Trees

- **Shortest paths from source 0 to all vertices in a network:**



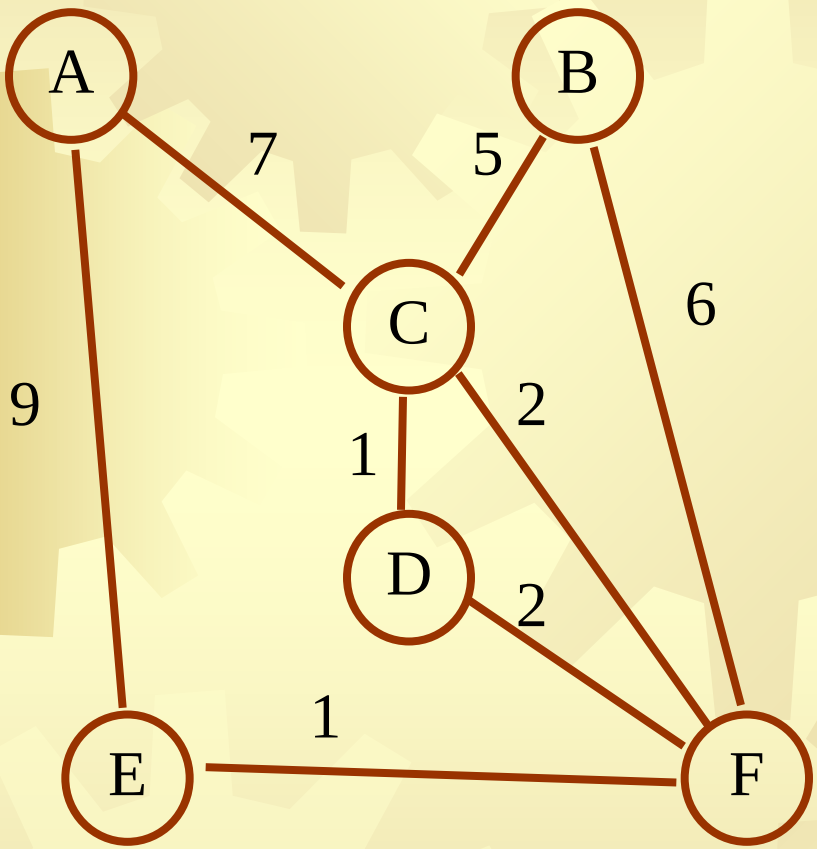
Minimum Cost Spanning Trees

- **Minimum Cost Spanning Tree (MST) Problem:**
  - **Input: An undirected, connected graph G.**
  - **Output: The subgraph of G that 1) has minimum total cost as measured by summing the values for all of the edges in the subset, and 2) keeps the vertices connected.**

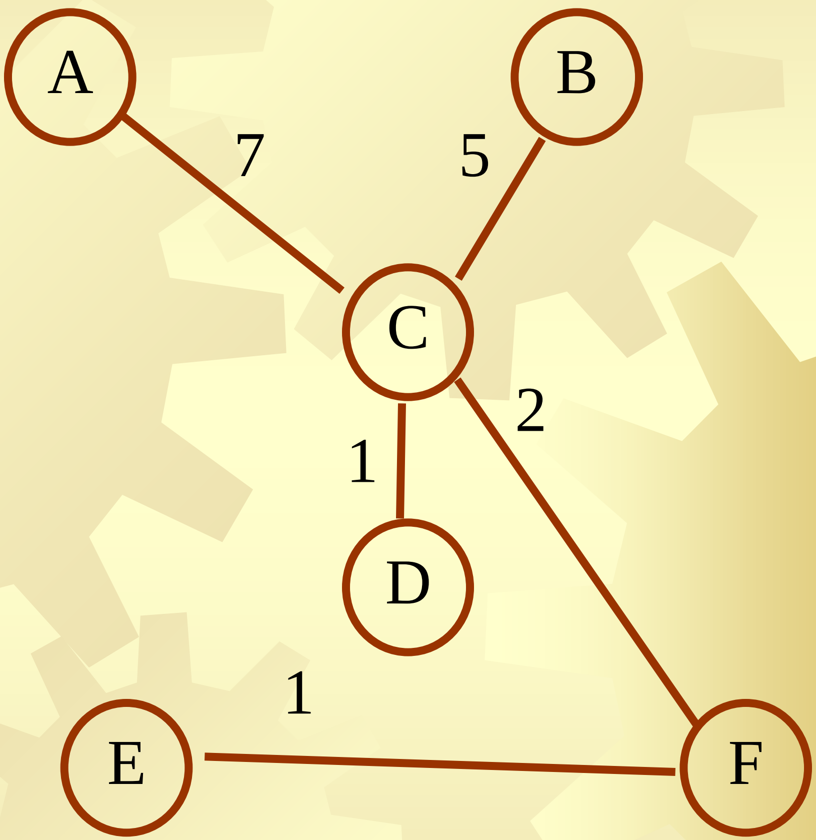
Minimal Spanning Trees

- DEFINITION: A ***minimal spanning tree*** of a connected network is a spanning tree such that the sum of the weights of its edges is as small as possible.

MST



(a)



(b)

Prim's Algorithm for MST

- Start with a source vertex.
- Keep a set X of those vertices whose paths to source in the minimal spanning tree that we are building have been found.
- Keep the set Y of edges that link the vertices in X in the tree under construction.
- The vertices in X and edges in Y make up a small tree that grows to become our final spanning tree.

Prim's Algorithm for MST

- Initially, source is the only vertex in X, and Y is empty. At each step, we add an additional vertex to X: This vertex is chosen so that an edge back to X has as small as possible a weight. This minimal edge back to X is added to Y .
- For implementation, we shall keep the vertices in X as the entries of a Boolean array component. We keep the edges in Y as the edges of a graph that will grow to give the output tree from our program.

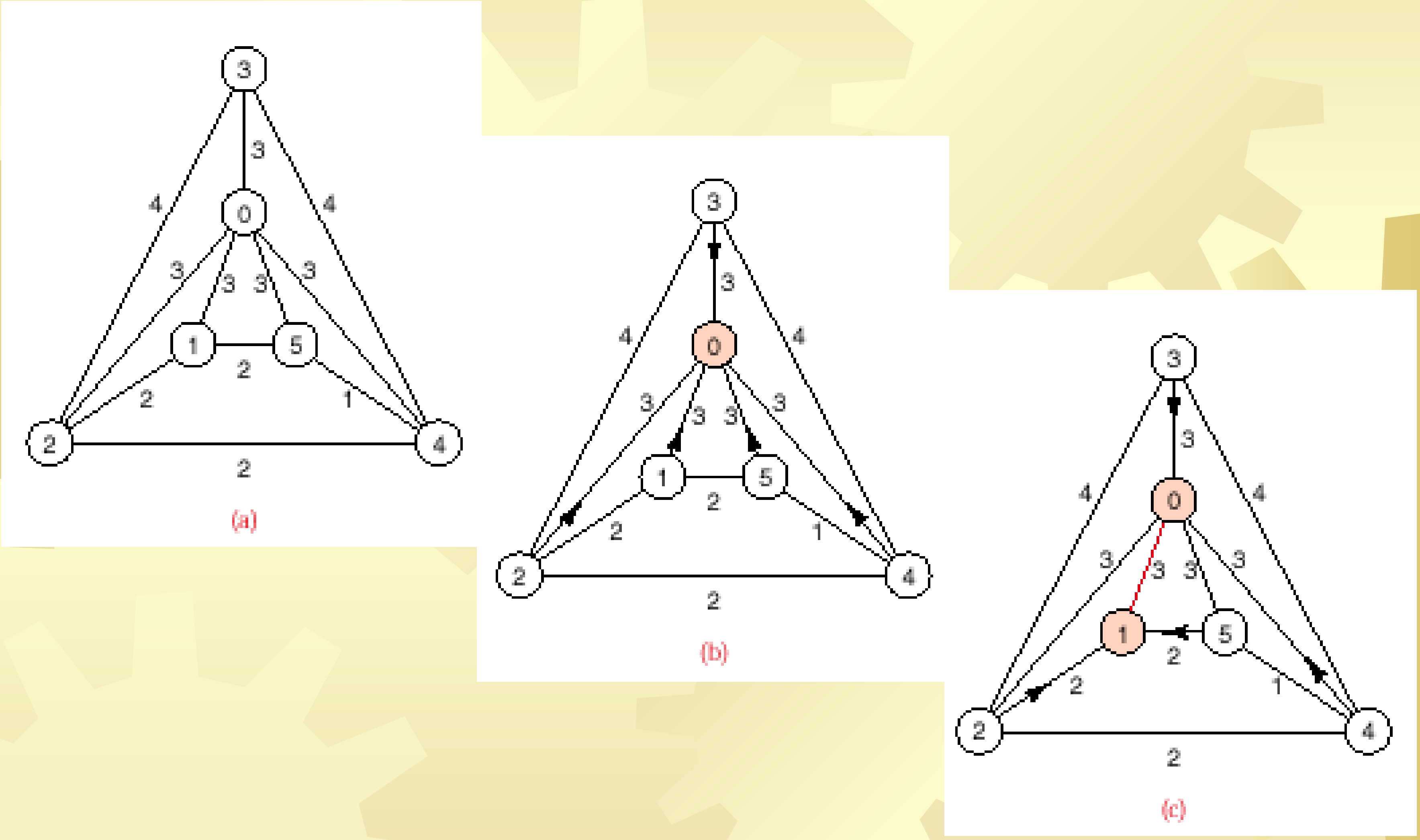
Prim's Algorithm for MST

- We maintain an auxiliary table neighbor that gives, for each vertex v, the vertex of X whose edge to v has minimal cost.
- We also maintain a second table distance that records these minimal costs. If a vertex v is not joined by an edge to X we shall record its distance as the value infinity. The table neighbor is initialized by setting neighbor[v] to source for all vertices v, and distance is initialized by setting distance[v] to the weight of the edge from source to v if it exists and to infinity if not.

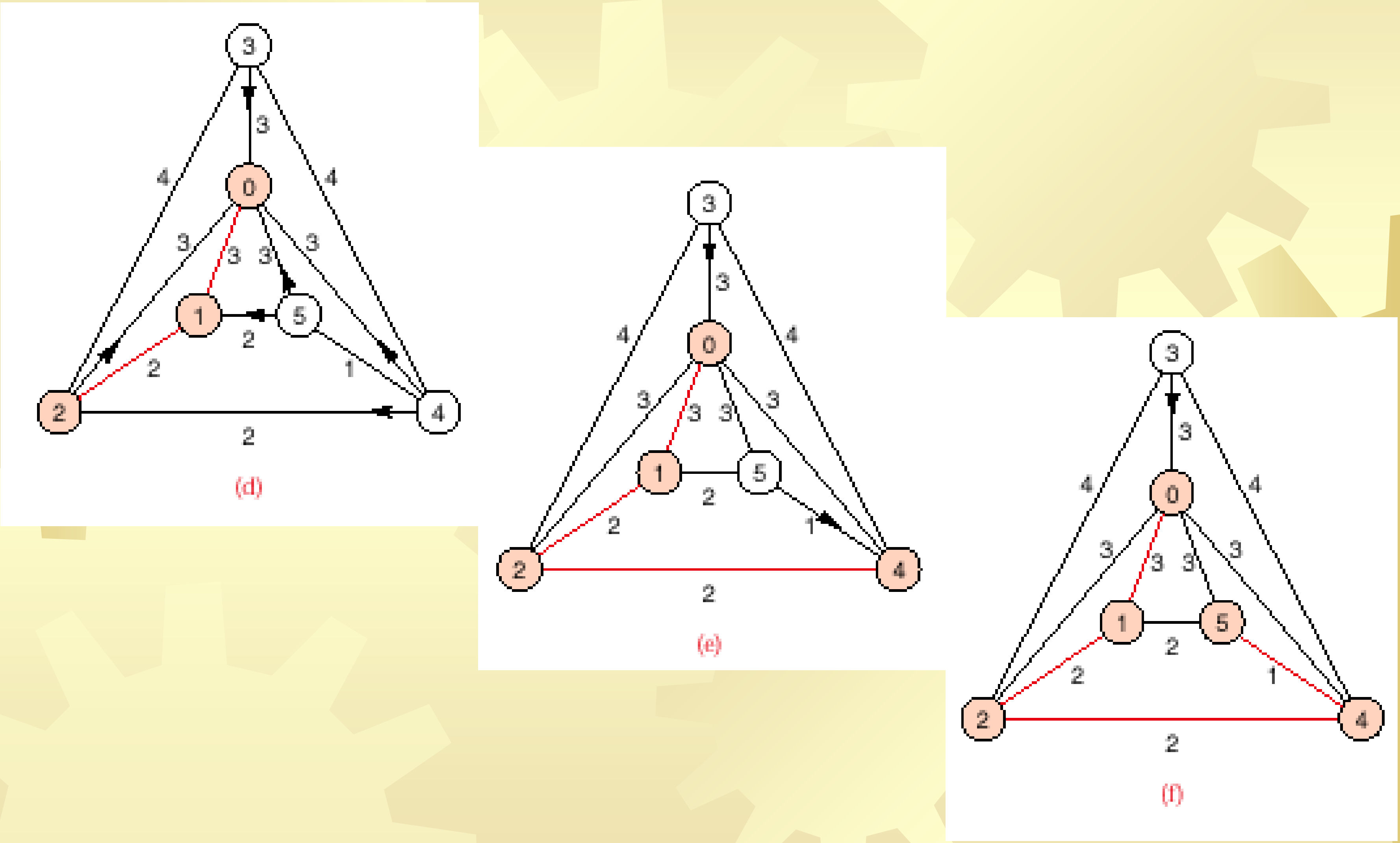
Prim's Algorithm for MST

- To determine what vertex to add to X at each step, we choose the vertex v with the smallest value recorded in the table distance, such that v is not already in X. After this we must update our tables to reflect the change that we have made to X. We do this by checking, for each vertex w not in X, whether there is an edge linking v and w, and if so, whether this edge has a weight less than distance[w]. In case there is an edge(v,w) with this property, we reset neighbor[w] to v and distance[w] to the weight of the edge.

Example of Prim's Algorithm

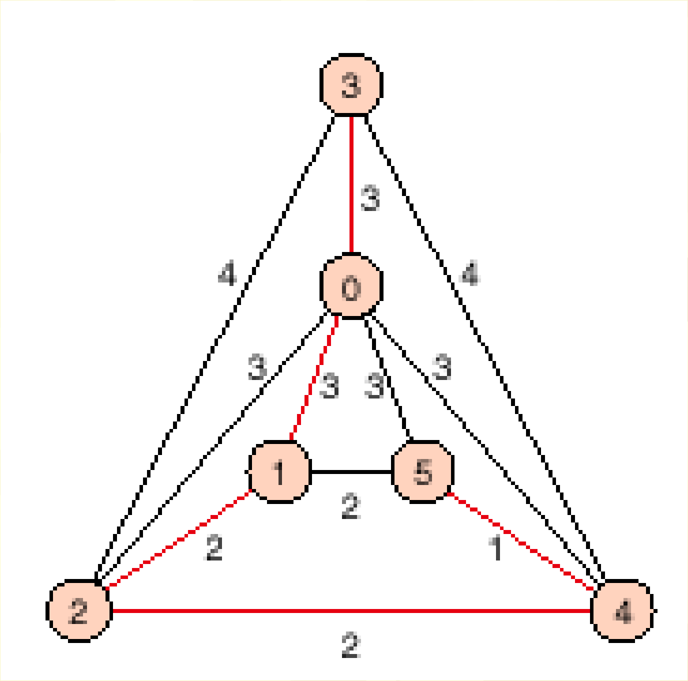


Example of Prim's Algorithm

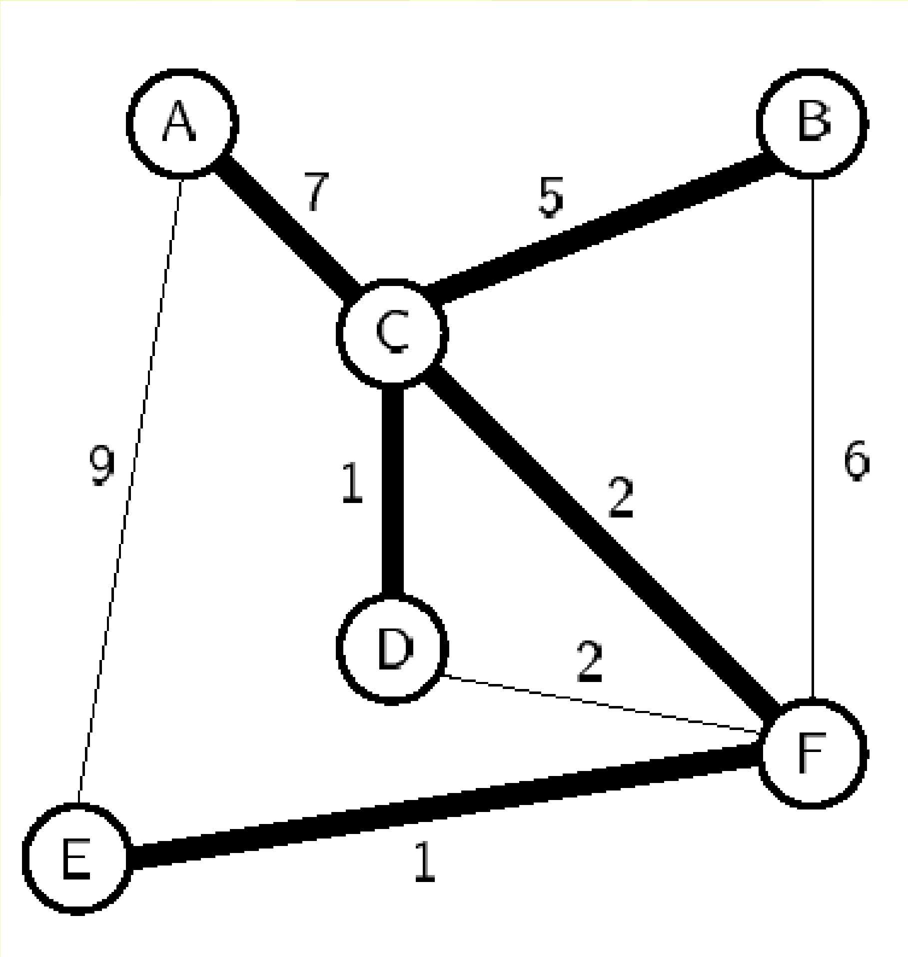


(d) (e) (f)

Example of Prim's Algorithm

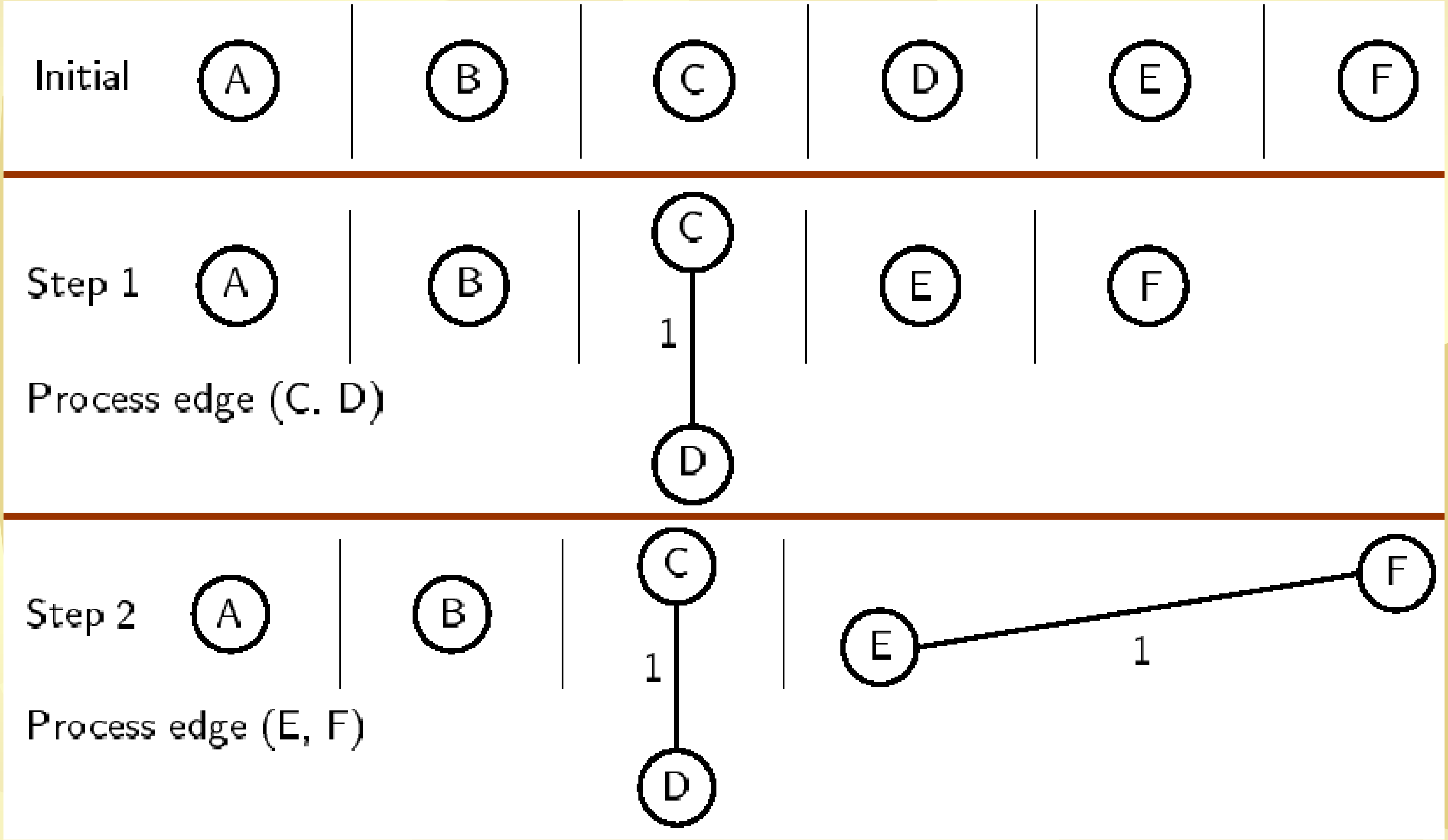


Kruskal's Algorithm

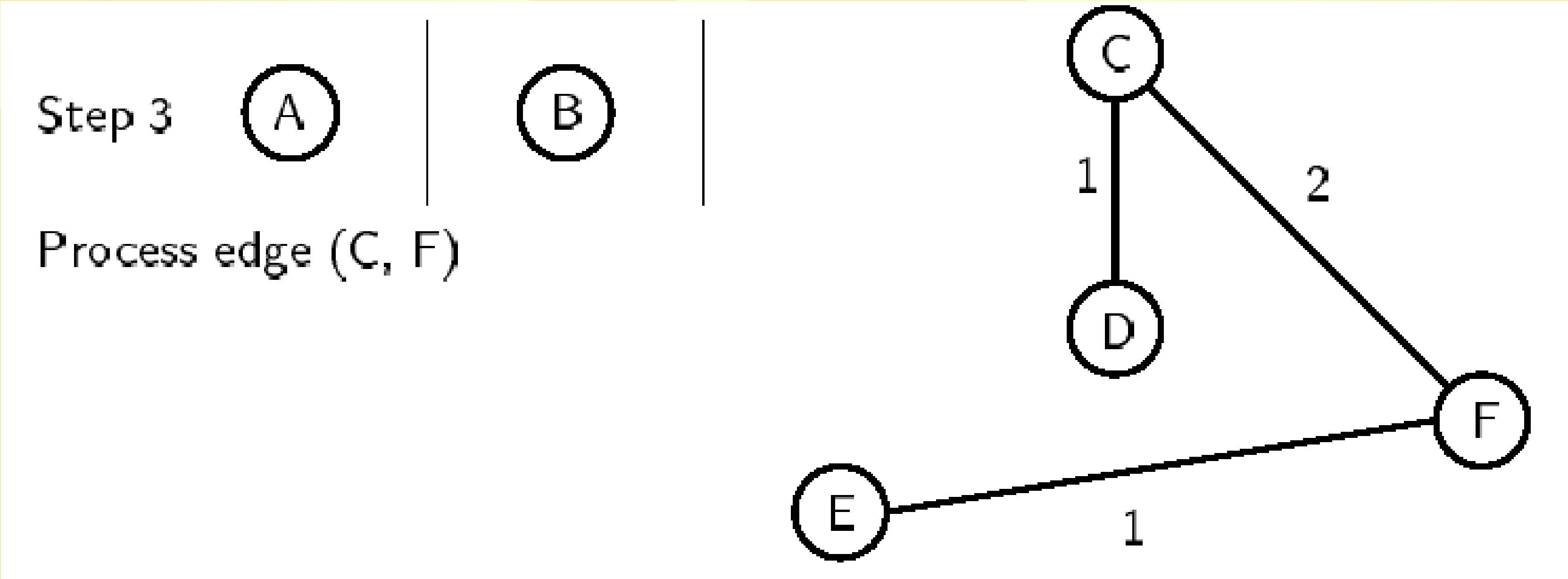


- **initial:**
  - **E1 = Φ, T = $\{v_1, v_2, \ldots, v_n\}$, | V | = n;**
- **while |E1|<n-1 do**
  - **chose one edge e= $(u_i, v_j) \in$ E,**
    cost(e) = min{ cost of (u_i, v_j) | $(u_i, v_j) \in$ E }
    delete e from E
  - **if (u_i and v_j belong to different connected components)**
    E1 = E1 $\cup$ {e}

Kruskal's Algorithm



Kruskal's Algorithm



Pointers and Pitfalls

- Graphs provide an excellent way to describe the essential features of many applications, thereby facilitating specification of the underlying problems and formulation of algorithms for their solution. Graphs sometimes appear as data structures but more often as mathematical abstractions useful for problem solving.
- Graphs may be implemented in many ways by the use of different kinds of data structures. Postpone implementation decisions until the applications of graphs in the problem-solving and algorithm-development phases are well understood.

Pointers and Pitfalls

- Many applications require graph traversal. Let the application determine the traversal method: depth first, breadth first, or some other order. Depth-first traversal is naturally recursive(or can use a stack). Breadth-first traversal normally uses a queue.
- Greedy algorithms represent only a sample of the many paradigms useful in developing graph algorithms. For further methods and examples, consult the references.