Data Structure and Algorithms

Chapter 8 Graph

Dr. Zhiqiang Liu

School of Software and Microelectronics, Northwest Polytechnical University

Outline

Examples

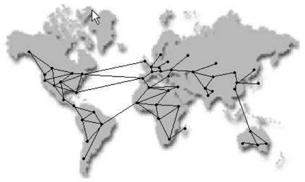
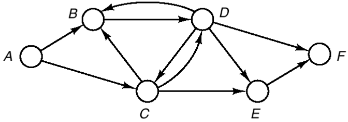
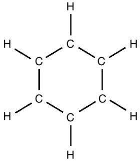


Figure 1 *Flights connecting cities as a graph*



Message transmission in a network



Benzene molecule

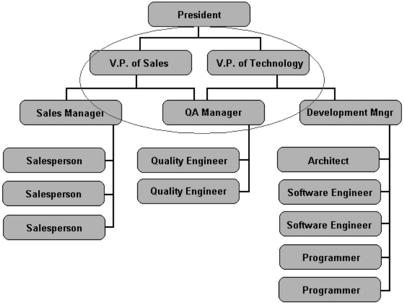


Figure 2 *An organizational chart as a graph*

Next Section

8.1 Graph Definition and Concepts

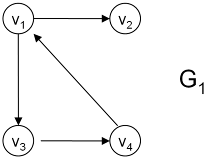
- Graph
  - Consists of a set V of vertices and a set E of edges.
  - A graph is a collection of vertices, V, and associated edges, E, given by the pair.

$$G = (V, E)$$

$$E = \{(x,y) | x,y \in V\}$$
$$E = \{< x,y > | x,y \in V \&\& Path(x,y)\}$$

Concepts

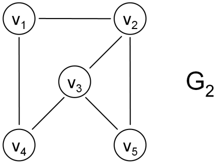
- Vertex
- Edge



$G1 = (V1, A1)$

$V1 = \{v1, v2, v3, v4\}$

$A1 = \{< v1, v2 >, < v1, v3 >, < v3, v4 >, < v4, v1 >\}$



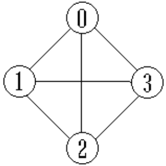
$G2 = (V2, E2)$

$V2 = \{v1, v2, v3, v4, v5\}$

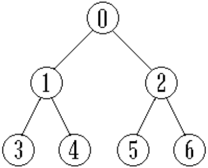
$E2 = \{(v1, v2), (v1, v4), (v2, v3), (v2, v5), (v3, v4), (v3, v5)\}$

8.1 Graph Definition and Concepts

- **Directed graph**
  - Ordered pair <x,y> has the sense of direction
- **Undirected graph**
  - Unordered pair (x,y) has not the sense of direction

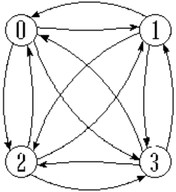


(a) G1



(b) G2

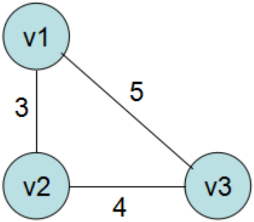
(c) G3



(d) G4

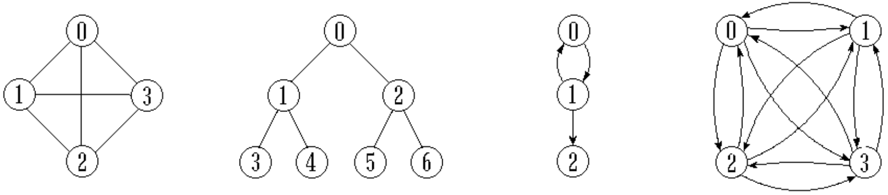
8.1 Graph Definition and Concepts

- **Network**
  - Graph with weight
  - Weight: the numbers relate to edges of the graph
  - Weights represent some meaning such as times, distance, price and so on
  - Weighted pair <x,y> or (x,y)



8.1 Graph Definition and Concepts

- **Completed graph**
  - Completed undirected graph: Graph consists of n vertices and $n(n-1)/2$ edges.
  - Completed directed graph : Graph consists of n vertices and $n(n-1)$ edges.

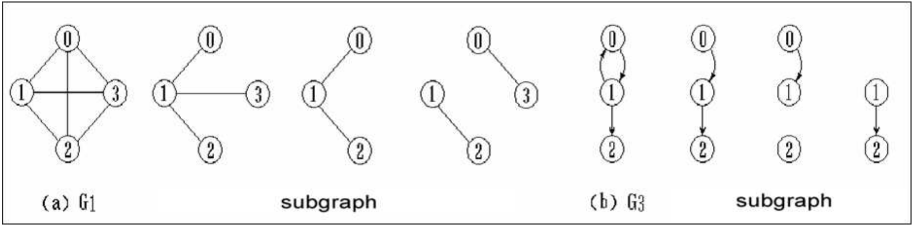


(a) G1 (b) G2 (c) G3 (d) G4

8.1 Graph Definition and Concepts

- **Sub-graph:**

  There are two graph, G=(V, E) and G'=(V',E'), if V' $\subseteq$ V and E' $\subseteq$ E, so G' is sub-graph of G.



8.1 Graph Definition and Concepts

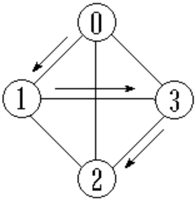
- **Adjacent vertex**
  - If (u, v) is one of E(G) , u and v is a pair of adjacent vertices
- **Degree of vertex, TD(v)**
  - Degree of a vertex represent number of edges those related to the vertex
  - In Degree ID(v)
  - Out Degree OD(v)
  - TD(v)=ID(v)+OD(v)

$$e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$$

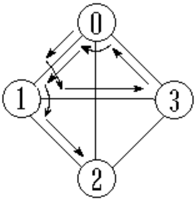
$$e = \sum_{i=1}^{n} ID(v_i) = \sum_{i=1}^{n} OD(v_i)$$

8.1 Graph Definition and Concepts

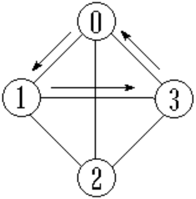
- **Path**
  - In $G = (V, E)$, a path from v_1 to vn is a sequence of edges edge(v_1, v_2), edge(v_2, v_3),..... edge(v_{n-1}, v_n) and is denoted as path v_1, v_2, v_3, ... v_{n-1}, v_n
- **Cycle (Circuit)**
  - In path from v_1 to v_n, if $v_1 = v_n$ and no edge is repeated.
- **Simple path**



(a) Simple path



(b) No Simple path

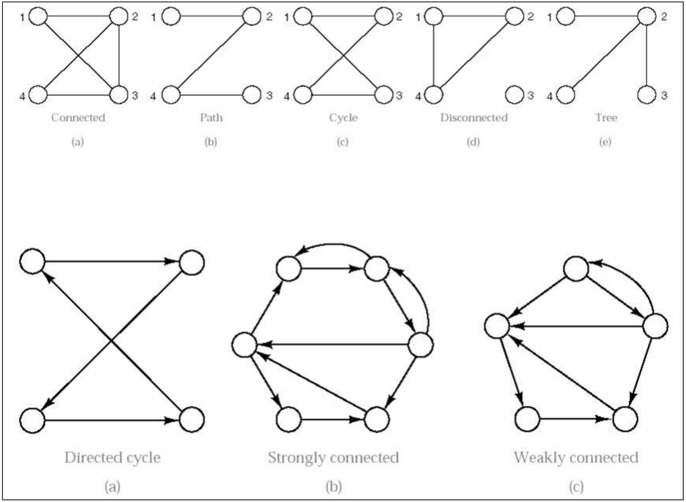


(c) Cycle

8.1 Graph Definition and Concepts

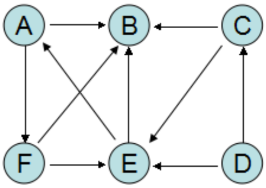
- **Path Length**
  - Generic length
  - Weighted length
- **Connected graph**
- **Connected component**
  - A connected component of an UNDIRECTED graph is a BIGGEST subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.
- **Strongly connected component**
  - A strongly connected component of a DIRECTED graph G is a BIGGEST subgraph that is strongly connected

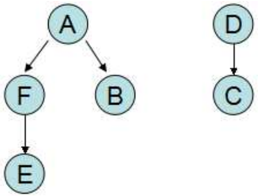
8.1 Graph Definition and Concepts



8.1 Graph Definition and Concepts

- **Spanning tree**
  - Connected graph
  - Mini connected sub-graph
  - Have all the vertices of graph (n)
  - Only $(n-1)$ edges
  - Without cycle
- **Spanning forest**
  - Unconnected graph or directed graph
  - Directed spanning trees

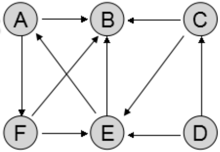




Next Section

8.2 ADT and Storage of Graph

```
class Graph {
public:
    Graph ( );
    void InsertVertex ( Type & vertex );
    void InsertEdge( int v1, int v2, int weight );
    void RemoveVertex ( int v );
    void RemoveEdge ( int v1, int v2 );
    int IsEmpty ( );
    Type GetWeight ( int v1, int v2 );
    int GetFirstNeighbor ( int v );
    int GetNextNeighbor ( int v1,  int v2 );
}
```



8.2 ADT and Storage of Graph

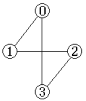
Adjacency Matrix

- Graph $A = (V, E)$
  - Connected graph
  - Mini connected sub-graph

$$A.Edge[i][j] = \begin{cases} 1, & if < i,j > \in E or (i,j) \in E \\ 0, & otherwise \end{cases}$$

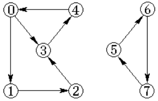
- Undirected graph is stored in symmetric matrix
- Directed graph may be stored in asymmetric matrix

8.2 ADT and Storage of Graph



G8

G3



G7

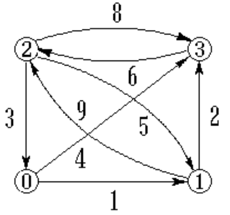
$$
\text{A.}\boldsymbol{Edge} = \begin{array}{c c}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\
\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}
\quad
\text{A.}\boldsymbol{Edge} = \begin{array}{c c}
\begin{array}{ccc} 0 & 1 & 2 \end{array} & \\
\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \end{array}
\end{array}
\quad
\text{A.}\boldsymbol{Edge} = \begin{array}{c c}
\begin{array}{ccccc:ccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} & \\
\left(\begin{array}{ccccc:ccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\end{array}
$$

- **In directed graph, out-degree of vertex *i* is the sum of row *i*, in-degree of vertex *i* is the sum of colum *i*.**
- **In undirected graph, degree of vertex *i* is the sum of row *i*.**

8.2 ADT and Storage of Graph

- **Network**

$$A.Edge[i][j]=\begin{cases} W(i,j) & \text{if } i\,!=j \text{ and } <i,j>\ \in E \text{ or } (i,j)\in E \\ \infty & \text{otherwise and } i\,!=j \\ 0 & \text{if } i==j \end{cases}$$



$$A.Edge=\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\ \begin{pmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \end{array}$$

```
// Graph class by adjacency matrix
const int MaxEdges = 50;
const int MaxVertices = 10;

template <class NameType, class DistType>
class Graph
{
private:
    SeqList<NameType> VerticesList (MaxVertices);
    DistType Edge[MaxVertices][MaxVertices];
    int CurrentEdges;
    int FindVertex ( SeqList<NameType> & L;
                     const NameType &vertex )
    {
        return L.Find (vertex);
    }

    int GetVertexPos ( Const NameType &vertex )
    {
        return FindVertex (VerticesList, vertex );
    }
```

```
public:
    Graph ( int sz = MaxNumEdges );
    int GraphEmpty ( ) const
    {
        return VerticesList.IsEmpty ( );
    }
    int GraphFull( )  const
    {
        return VerticesList.IsFull( ) ||
               CurrentEdges ==MaxEdges;
    }
    int NumberOfVertices ( )
    {
        return VerticesList.last +1;
    }
    int NumberOfEdges ( )
    {
        return CurrentEdges;
    }
    NameType GetValue ( int i )
    {
        return i >= 0 && i <= VerticesList.last
```

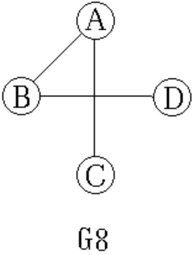
```
                ? VerticesList.data[i] : NULL;
    }
    DistType GetWeight ( int v1,  int v2 );
    int GetFirstNeighbor ( int v );
    int GetNextNeighbor ( int v1,  int v2 );
    void InsertVertex ( NameType & vertex );
    void InsertEdge(int v1,  int v2, DistType weight);
    void RemoveVertex ( int v );
    void RemoveEdge ( int v1,  int v2 );
}

//Implementation of functions
//constructor
template <class NameType, class DistType>
Graph<NameType, DistType> :: Graph( int sz)
{
    for ( int i = 0; i < sz; i++ )
        for ( int j = 0; j < sz; j++ )
            Edge[i][j] = 0;
    CurrentEdges = 0;
}

```

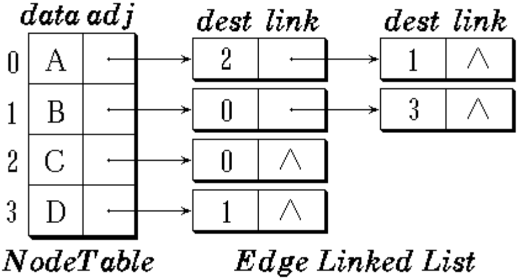
```
//get the weight of arc (v1, v2)
template <class NameType, class DistType>
DistType Graph<NameType, DistType> ::
GetWeight( int v1,  int v2 )
{
    if ( v1 != -1 && v2 != -1 )
        return Edge[v1][v2];
    else return 0;
}
//Get the first adjacent node
template <class NameType, class DistType>
int Graph<NameType, DistType>::
GetFirstNeighbor(const int v)
{
    if ( v != -1 ){
        for( int col = 0; col < VerticesList.last; col++ )
            if( Edge[v][col] > 0 && Edge[v][col] < max)
                return col;
    }
    return -1;
}
```

8.2 ADT and Storage of Graph

Adjacency List

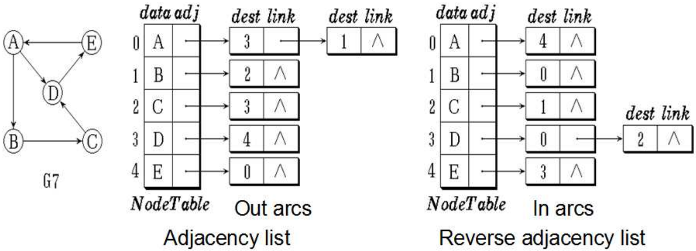
- **Undirected graph**





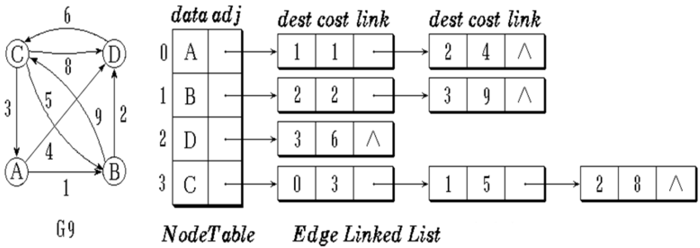
8.2 ADT and Storage of Graph

- **Directed graph**
  - Adjacency list
  - Reverse adjacency list



8.2 ADT and Storage of Graph

- **Network**
  - Adjacency list
  - Reverse adjacency list



```
//Graph class by adjacency list
const int DefaultSize = 10;
template <class DistType> class Graph;
template <class DistType> struct Edge
{
    friend class Graph<NameType, DistType>;
    int dest;
    DistType cost;
    Edge<DistType> *link;
    Edge ( ) { }
    Edge ( int D, DistType C ) :  dest (D), cost (C),
        link (NULL) { }
    int operator != ( Edge<DistType>& E )const
    {
        return dest != E.dest;
    }
}

template <class NameType, class DistType>
struct Vertex
{
```

```
    friend class Graph<NameType, DistType>;
    NameType data;
    Edge<DistType> *adj;
}

template <class NameType, class DistType>
class Graph
{
private:
    Vertex<NameType, DistType> *NodeTable;
    int NumVertices;
    int MaxVertices;
    int NumEdges;
    int GetVertexPos ( NameType & vertex );
public:
    Graph ( int sz );
    ~Graph ( );
    int GraphEmpty ( )const
    {
        return NumVertices == 0;
    }
    int GraphFull ( ) const
```

```
    {
        return NumVertices == MaxVertices;
    }
    NameType GetValue ( int i )
    {
        return i >= 0 && i < NumVertices ?
               NodeTable[i].data : NULL;
    }
    int NumberOfVertices ( )
    {
        return NumVertices;
    }
    int NumberOfEdges ( )
    {
        return NumEdges;
    }
    void InsertVertex ( NameType & vertex );
    void RemoveVertex ( int v );
    void InsertEdge(int v1,  int v2,  DistType weight);
    void RemoveEdge ( int v1,  int v2 );
    DistType GetWeight ( int v1,  int v2 );
    int GetFirstNeighbor ( int v );
```

```
    int GetNextNeighbor ( int v1,  int v2 );
}

//Constructor
template <class NameType, class DistType>
Graph<NameType, DistType> :: Graph ( int sz =
    DefaultSize )
    : NumVertices (0), MaxVertices (sz), NumEdges (0)
{
    int n, e, k, j;
    NameType name, tail, head;
    DistType weight;
    NodeTable =  new Vertex<Nametype>[MaxVertices];
    cin >> n;
    for ( int i = 0; i < n; i++)
    {
        cin >> name;
        InsertVertex ( name );
    }
    cin >> e;
    for ( i = 0; i < e; i++)
    {
```

```
            cin >> tail >> head >> weight;
            k = GetVertexPos ( tail );
            j = GetVertexPos ( head );
            InsertEdge ( k, j, weight );
        }
}

//Destructor
template <class NameType, class DistType>
Graph<NameType, DistType> ::
~Graph ( )
{
    for ( int i = 0; i < NumVertices; i++ )
    {
        Edge<DistType> *p = NodeTable[i].adj;
        while ( p != NULL )
        {
            NodeTable[i].adj = p->link;
            delete p;
            p = NodeTable[i].adj;
        }
    }
```

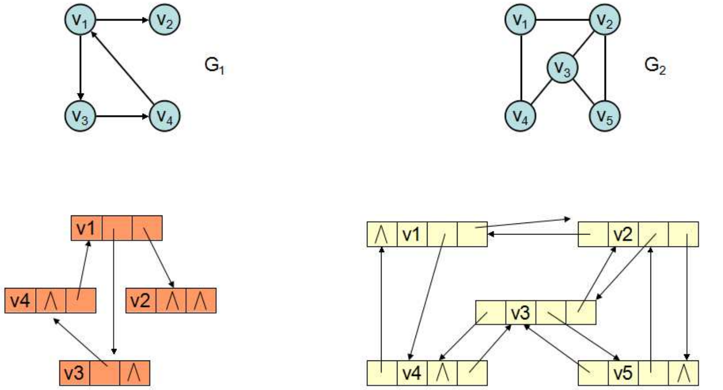
```
    delete [ ] NodeTable;
}

//Implementation of functions
//get the serial number of vertex
template <class NameType, class DistType>
int Graph<NameType, DistType> ::
GetVertexPos ( const NameType & vertex )
{
    for ( int i =0; i < NumVertices; i++ )
        if ( NodeTable[i].data == vertex ) return i;
    return -1;
}

//Get the first adjacent of v
template <Class NameType, class DistType>
int Graph<NameType, DistType> ::
GetFirstNeighbor ( int v )
{
    if ( v != -1 )
    {
        Edge<DistType> *p = NodeTable[v].adj;
```

```
            if ( p != NULL ) return p->dest;
        }
        return -1;
}

//Get the next adjacent of v1 after v2
template <Class NameType, class DistTypeType>
int Graph<NameType, DistType> ::
GetNextNeighbor ( int v1,  int v2 )
{
        if ( v1 != -1 )
        {
            Edge<DistType> *p = NodeTable[v1].adj;
            while ( p != NULL )
            {
                if ( p->dest == v2 && p->link != NULL )
                    return p->link->dest;
                else p = p->link;
            }
        }
        return -1;
}
```

```
//Get the cost between  v1 and v2
template <Class NameType, class DistType>
DistType Graph<NameType, DistType> ::
GetWeight ( int v1,  int v2)
{
    if ( v1 != -1 && v2 != -1 )
    {
        Edge<DistType> *p = NodeTable[v1].adj;
        while ( p != NULL )
        {
            if ( p->dest == v2 )
                return p->cost;
            else
                p = p->link;
        }
    }
    return 0;
}
```

8.2 ADT and Storage of Graph

Multi-linked list



8.2 ADT and Storage of Graph

Adjacency Multi-list(邻接多重表)

- **Undirected graph**
  - Edge

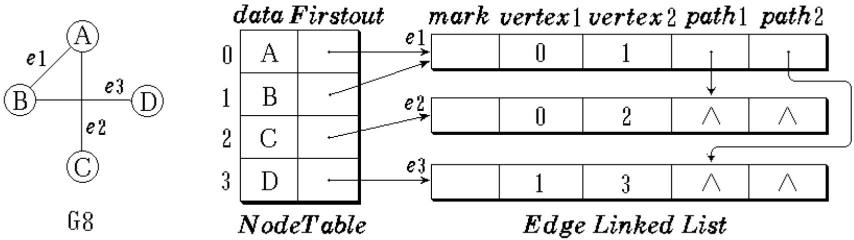
mark	vertex1	vertex2	path1	path2

    - **Mark (visit sign)**
    - **Vertex1, Vertex2**
    - **Path1, Path2 (pointers to next edge incident with vertex1 and vertex2)**
  - Vertex

data	Firstout

8.2 ADT and Storage of Graph

Example1 of AML



8.2 ADT and Storage of Graph

- **Directed graph**
  - Edge

mark	vertex1	vertex2	path1	path2

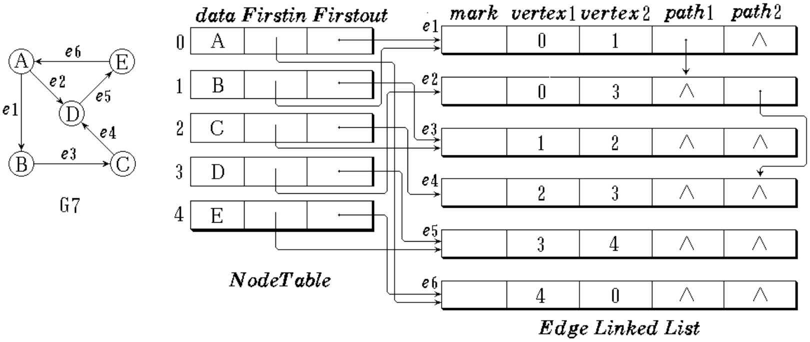
    - **Mark**
    - **Vertex1 (initial node)**
    - **Vertex2 (terminal node)**
    - **Path1 (link to next arc beginning from vertex1)**
    - **Path2 (link to next arc ending at vertex2)**
  - Vertex

data	Firstin	Firstout

    - **Data**
    - **Firstin (point to the first arc starting from this vertex)**
    - **Firstout (point to the first arc ending at this vertex)**

8.2 ADT and Storage of Graph

Example2 of AML



Next Section

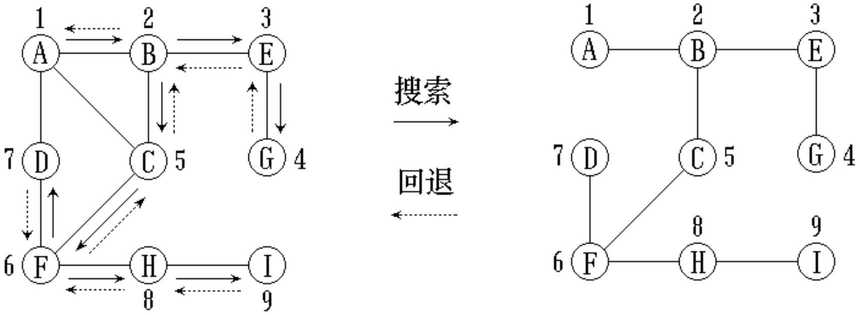
8.3 Traversal and Connectivity

Traversal

- Requirement: Visit each vertex one time and only one time
- Approaches
  - **Depth-first traversal**
  - **Breadth-first traversal**
- Auxiliary Boolean array
  - **Visited[]**

8.3 Traversal and Connectivity

Depth First Search (DFS)



- **Steps**
  - Searching
  - Backtracking (recursion involved)

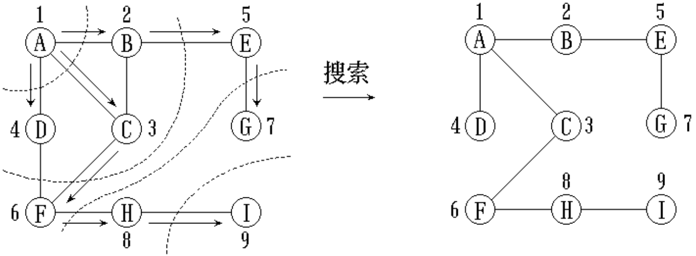
```
// DFS algorithm
template <class NameType, class DistType>
void Graph <NameType, DistType> :: DFS ( )
{
    int * visited = new int [NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
        visited [i] = 0;
    DFS (0, visited );
    delete [ ] visited;
}

template<class NameType, class DistType>
void Graph<NameType, DistType> ::
DFS ( const int v, int visited [ ] )
{
    cout << GetValue (v) << ‘’ ;  //visit v
    visited[v] = 1;           //marking v
    int w = GetFirstNeighbor (v);
    //Get the first adjacent of v : w
    while ( w != -1 )
    {
```

```
            if ( !visited[w] ) DFS ( w, visited );
            //if  w has not been visited, DFS(w)
            w = GetNextNeighbor ( v, w );
            //Get the next adjacent of v after w
        }
}
```

- **Time Analysis**
  - **O(n+e) for adjacency list**
  - **O(n^2) for adjacency matrix**

8.3 Traversal and Connectivity

Breadth First Search (BFS)



- **Steps**
  - Search
  - Search in next level (queue involved)

```
//BFS algorithm

template<class NameType, class DistType>
void Graph <NameType, DistType> ::
BFS ( int v )
{
    int * visited = new int[NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
        visited[i] = 0;
    cout << GetValue (v) << ' ';
    visited[v] = 1;
    Queue<int> q;
    q.EnQueue (v);
    while ( !q.IsEmpty ( ) )
    {
        v = q.DeQueue ( );
        int w = GetFirstNeighbor (v);
        while ( w != -1 )
        {
            if ( !visited[w] )
            {
```

```
                cout << GetValue (w) << ‘’ ;
                visited[w] = 1;
                q.EnQueue (w);
            }
            w = GetNextNeighbor (v, w);
        }
    }
    delete [ ] visited;
}
```

- **Time Analysis**

  - $d_0 + d_1 + \ldots + d_{n-1}$ **= O(e) for adjacency list,** d_i **is the degree of node i**
  - **O(n^2) for adjacency matrix**

8.3 Traversal and Connectivity

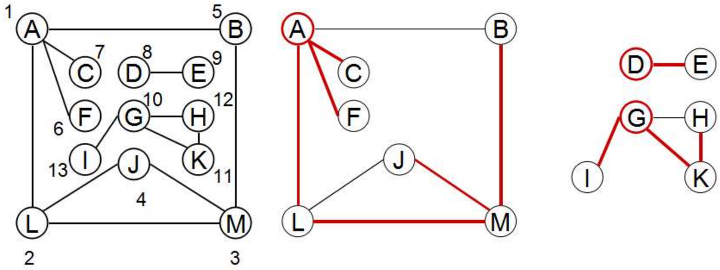
Connected component

- **Connected graph**
  - Visit all the nodes within one searching
  - Spanning tree
    - **Depth first spanning tree**
    - **Breadth first spanning tree**
- **Unconnected graph**
  - Call DFS or BFS several times until all the nodes have been visited
  - Spanning forest

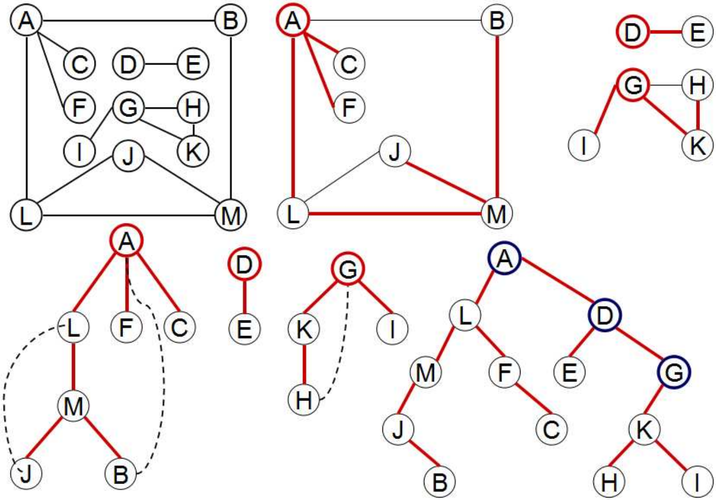
8.3 Traversal and Connectivity

对于非连通图，其中的每一个连通分量都可以通过遍历得到一棵生成树，所有这些连通分量的生成树就构成了非连通图生成森林。

如果以孩子兄弟链表作生成森林的存储结构，则可以形成非连通图的生成森林。算法如下：



8.3 Traversal and Connectivity



```
void DFSForest (MGraph g, CSTree *T)
{
    CSTree   p, q;
    int  v;
    *T = NULL;
    for (v = 0; v < g.vexNum; v++)  visited[v] = FALSE;
    q = *T;
    for (v=0; v < g.vexNum; v++)
    {
        if (!visited[v])
        {
            p = (CSNode *) malloc(sizeof(CSNode));
            assert(p);
            p->data = GetVex (g, v);
            p->firstChild = NULL;
            p->nextSibling = NULL;
            if (!(*T))  *T = p;
            else  q->nextSibling = p;
            q = p;
            DFSTree (g, v, &p);
        }   /* if */
```

```
    }   /* for */
}   /* End of DFSForest() */
void DFSTree (MGraph g, int v, CSTree *T)
{
    BOOL    first;
    CSTree   p, q;
    int  w;
    visited[v] = TRUE;
    first = TRUE;
    q = *T;
    for (w = FirstAdjVex (g, v); w!=-1; w = NextAdjVex (
        g, v, w))
    {
        if (!visited[w])
        {
            p = (CSTree) malloc(sizeof(CSNode));
            p->data = GetVex (g, w);
            p->firstChild = NULL;
            p->nextSibling = NULL;
            if (first)
            {
                (*T)->firstChild = p;
```

```
                first = FALSE;
            }
            else  q->nextSibling = p;
            q = p;
            DFSTree (g, w, &q);
        }   /* if */
    }   /* for */
}
```

Next Section

8.4 Minimum Cost Spanning Tree

- 最小（代价）生成树: 一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。砍去一条边就使生成树变成非连通图；增加一条边，就会出现一个回路。
- 许多应用问题都是一个求无向连通图的最小生成树问题。例如：要在n个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信；但铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，另一个目标是要使铺设光缆的总费用最低。这就需要找到带权的最小生成树。

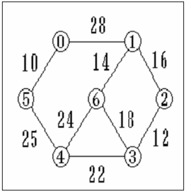
8.4 Minimum Cost Spanning Tree

- 最小（代价）生成树: 一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。砍去一条边就使生成树变成非连通图；增加一条边，就会出现一个回路。
- 许多应用问题都是一个求无向连通图的最小生成树问题。例如：要在n个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信；但铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，另一个目标是要使铺设光缆的总费用最低。这就需要找到带权的最小生成树。
- **Undirected graph**
  - DFS or BFS based spanning tree
  - n vertices and n-1 edges
  - Dependent on the starting vertex
- **Undirected network**
  - n-1 lowest cost edges, and
  - no cycle
  - Independent with starting vertex

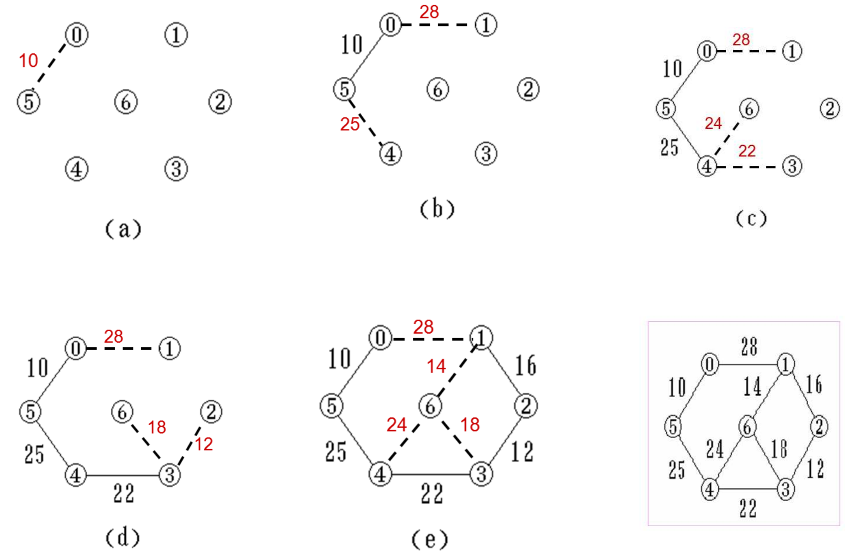
8.4 Minimum Cost Spanning Tree

Prim algorithm

- **Connected network**
  - $N = \{V, E\}$, starting at u_0
- **Initialization**
  - $U = \{u_0\}$
- **Iteration**
  - $U = U + \{v\}, (u, v)$ is the lowest cost edge $u \in U$, $v \in V - U$
- **Termination**
  - $U == V$



8.4 Minimum Cost Spanning Tree



8.4 Minimum Cost Spanning Tree

Implementation of Prim Algorithm

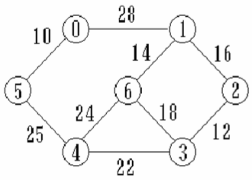
- **Auxiliary arrays**
  - lowcost[]

    lowest cost (u, v), $u \in U$, $v \in V - U$
  - nearvex[]

    the serial number of vertex u for $v \in V - U$

8.4 Minimum Cost Spanning Tree

Example



(a) Network

$$
\begin{array}{c}
\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\left(\begin{array}{ccccccc}
0 & 28 & \infty & \infty & \infty & 10 & \infty \\
28 & 0 & 16 & \infty & \infty & \infty & 14 \\
\infty & 16 & 0 & 12 & \infty & \infty & \infty \\
\infty & \infty & 12 & 0 & 22 & \infty & 18 \\
\infty & \infty & \infty & 22 & 0 & 25 & 24 \\
10 & \infty & \infty & \infty & 25 & 0 & \infty \\
\infty & 14 & \infty & 18 & 24 & \infty & 0
\end{array}\right)
\end{array}
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
$$

(b) Adjacency matrix

Initialization

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞

	0	1	2	3	4	5	6
nearvex	-1	0	0	0	0	0	0

Lowest cost is (0,5), v=5, mark vertex 5 after selecting edge (0,5)

8.4 Minimum Cost Spanning Tree

- **Initialization**
  - lowcost[]: From adjacency matrix
  - nearver[]: Set to 0 except the starting vertex (-1)
- **Iteration**
  - lowcost[]
    v = nearvex[i] $\neq$ -1 && min(lowcost[i])
    selected edge is (nearvex[v], v), cost is lowcost[v]
    modify the lowcost array
  - nearvex[]
    set nearvex[v] = -1, add (nearvex[v], v, lowcost[v])
    into spanning tree

8.4 Minimum Cost Spanning Tree

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞

	0	1	2	3	4	5	6
nearvex	−1	0	0	0	0	0	0

$v = 5$

Select vertex 5 and add in spanning tree (Select line 5 match to lowcast)

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞

	0	1	2	3	4	5	6
nearvex	−1	0	0	0	5	−1	0

$v = 4$

repeat

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
lowcost	0	28	12	22	25	10	18
lowcost	0	16	12	22	25	10	18
lowcost	0	16	12	22	25	10	14
lowcost	0	16	12	22	25	10	14

	0	1	2	3	4	5	6
nearvex	−1	0	0	4	−1	−1	4
nearvex	−1	0	3	−1	−1	−1	3
nearvex	−1	2	−1	−1	−1	−1	3
nearvex	−1	−1	−1	−1	−1	−1	1
nearvex	−1	−1	−1	−1	−1	−1	−1

```
Final result
    (0, 5, 10), (5, 4, 25), (4, 3, 22),
    (3, 2, 12), (2, 1, 16), (1, 6, 14)
//Prim algorithm
void Graph<string, float>  :: Prim ( MinSpanTree &T )
{
    int NumVertices = VerticesList.last;
    float * lowcost = new float[NumVertices];
    int * nearvex = new int[NumVertices];
    for ( int i = 1; i < NumVertices; i++ )
    {
        lowcost[i] = Edge[0][i];
        nearvex[i] = 0;
    }
    nearvex[0] = -1;
    MSTEdgeNode e;
    for ( i = 1; i < NumVertices; i++ )
    {
        float min = MAXNUM;
        int v = 0;
        for ( int j = 0; j < NumVertices; j++ )
```

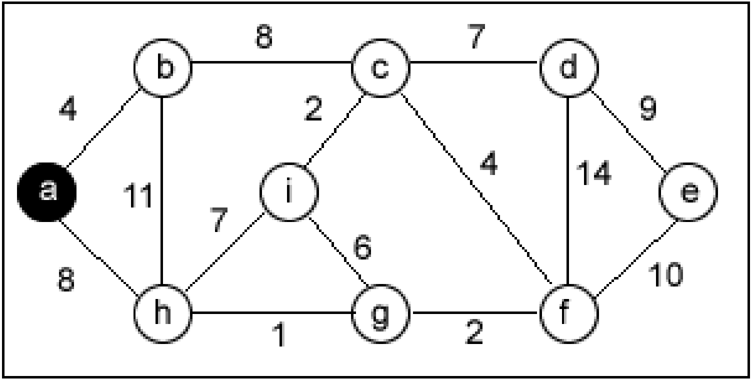
```
                if ( nearvex[j] != -1 && lowcost[j] < min )
                {
                    v = j;
                    min = lowcost[j];
                }
            if ( v )
            {
                e.tail = nearvex[v];
                e.head = v;
                e.cost = lowcost[v];
                T.Insert (e);
                nearvex[v] = -1;
                for ( j = 1; j < NumVertices; j++ )
                    if ( nearvex[j] != -1 && Edge[v][j] <
                        lowcost[j] )
                    {
                        lowcost[j] = Edge[v][j];
                        nearvex[j] = v;
                    }
            }
        }
}
```

8.4 Minimum Cost Spanning Tree

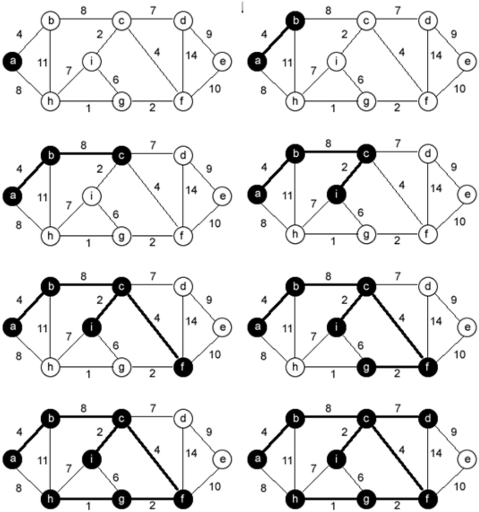
- **Time Analysis**
  - $O(n^2)$ for adjacency matrix

8.4 Minimum Cost Spanning Tree

- Get Minimum cost spanning tree with PRIM

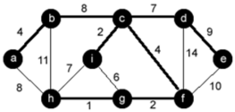


8.4 Minimum Cost Spanning Tree



1	2
3	4
5	6
7	8

Result



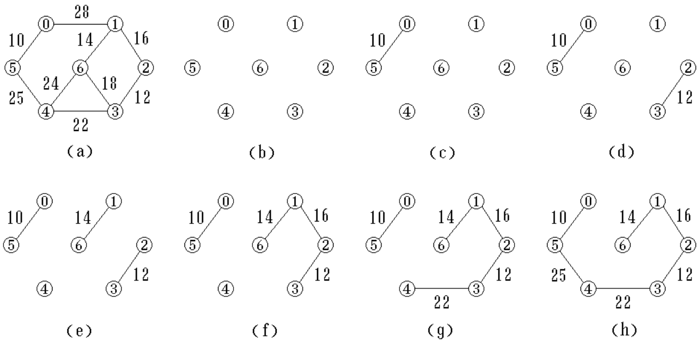
8.4 Minimum Cost Spanning Tree

Kruskal algorithm

- **Connected network** $N = \{V, E\}$
  - n vertices
- **Initialization**
  - $T = \{V, \phi\}$
  - n connected components
- **Iteration**
  - Select lowest cost edge (u, v)
  - Number of components decrease by 1

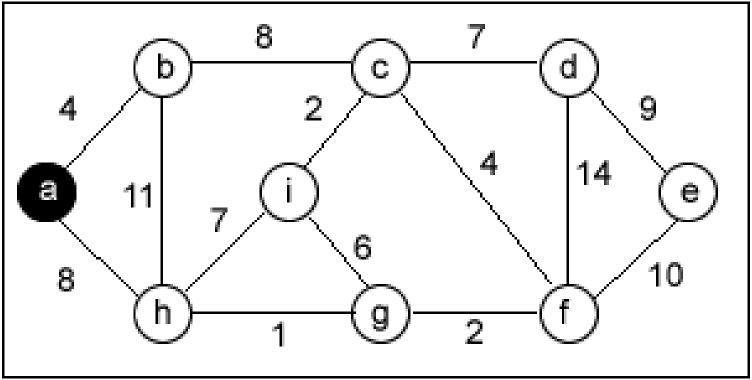
Greedy Strategy

8.4 Minimum Cost Spanning Tree

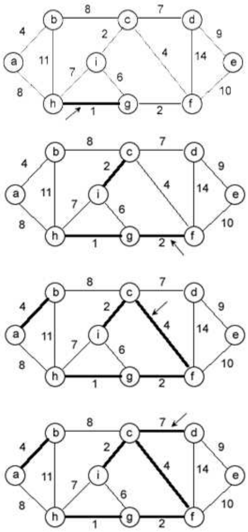


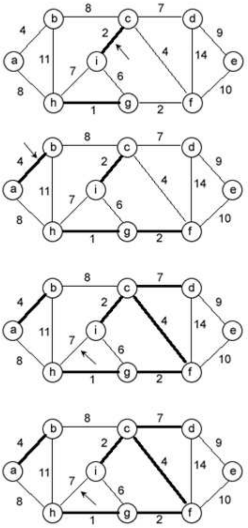
8.4 Minimum Cost Spanning Tree

- Get Minimum cost spanning tree with Kruskal



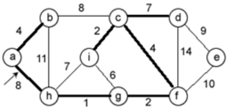
8.4 Minimum Cost Spanning Tree



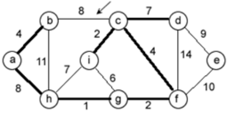


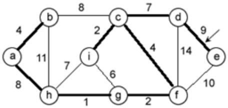
1	2
3	4
5	6
7	8

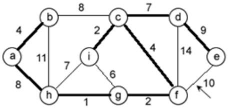
8.4 Minimum Cost Spanning Tree

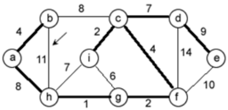


9	10
11	12
13	14

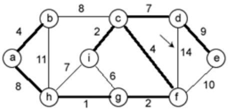








Result



Next Section

8.5 Shortest Path

- **Background**
  - Minimum transfer number
  - Shortest weighted path length
- **Solutions**
  - BFS traversal (unweighted shortest path)
  - Dijkstra algorithm
  - Floyd algorithm

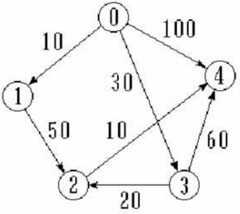
8.5 Shortest Path

Dijkstra algorithm

- **Problem**
  - Given a directed graph in which each edge has a non-negative weight or cost, find a path of least total weight from a given vertex,called the source, to <u>every other vertex</u> in the graph.
- **Rule**
  - Greedy criterion

8.5 Shortest Path

Example



(a) Directed netrwork

$$
\begin{array}{c}
\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\left(\begin{array}{ccccc}
0 & 10 & \infty & 30 & 100 \\
\infty & 0 & 50 & \infty & \infty \\
\infty & \infty & 0 & \infty & 10 \\
\infty & \infty & 20 & 0 & 60 \\
\infty & \infty & \infty & \infty & 0
\end{array}\right)
\end{array}
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
$$

(b) Adjacency matrix

Source	Dest	Shortest path			Path length
v_0	v_1	(v_0, v_1)			10
	v_2	—	(v_0, v_1, v_2)	(v_0, v_3, v_2)	— 60 50
	v_3	(v_0, v_3)			30
	v_4	(v_0, v_4)	(v_0, v_3, v_4)	(v_0, v_3, v_2, v_4)	100 90 60

8.5 Shortest Path

Implementation

- **Initialization**
  - $S \leftarrow \{v0\}$;
    $dist[j] \leftarrow Edge[0][j], j = 1, 2, \ldots, n-1$;
- **Iteration**
  - $dist[k] \leftarrow min\{dist[i]\}, i \in V - S$;
    $S \leftarrow S \bigcup \{k\}$;
  - $dist[i] \leftarrow min\{dist[i], dist[k] + Edge[k][i]\}$ for each $i \in V - S$
- **Termination**
  - $S = V$

```
// class of graph for shortest path
const int NumVertices = 6;
class Graph
{
private:
    float Edge[NumVertices][NumVertices];
    float dist[NumVertices];
    int path[NumVertices];
    int S[NumVertices];
public:
    void ShortestPath ( int,  int );
    int choose ( int );
};
// Dijkstra algorithm

void Graph :: ShortestPath ( int n,  int v )
{
//是一个具有Graph n 个顶点的带权有向图, 各边
//上的权值由Edge[i][j给出。本算法建立起一个]
//数组: dist[j], 0 □ j < n, 是当前求到的从顶点 v
//到顶点 j 的最短路径长度, 同时用数组path[j],
```

```
// 0 □ j < n, 存放求到的最短路径。
    for ( int i = 0; i < n; i++)
    {
        dist[i] = Edge[v][i];       //dist initialization
        S[i] = 0;
        if ( i != v && dist[i] < MAXNUM) path[i] = v;
        else path[i] = -1;                 //path initialization
    }
    S[v] = 1;
    dist[v] = 0;              //add source v into S
    //find out the shortest path (vertex u)
    for ( i = 0; i < n-1; i++ )
    {
        float min = MAXNUM;
        int u = v;
        for ( int j = 0; j < n; j++ )
            if ( !S[j] && dist[j] < min )
            {
                u = j;
                min = dist[j];
```

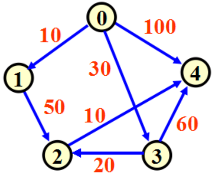
```
            }
        S[u] = 1;                          //add u into
            S
        for ( int w = 0; w < n; w++ )      //modify
            dist
            if ( !S[w] && Edge[u][w] < MAXNUM &&
                    dist[u] + Edge[u][w] < dist[w] )
            {
                dist[w] = dist[u] + Edge[u][w];
                path[w] = u;
            }
    }
}
```

8.5 Shortest Path

Demo of Dijkstra algorithm

	V1			V2			V3			V4		
	S[1]	*d*[1]	*p*[1]	*S*[2]	*d*[2]	*p*[2]	*S*[3]	*d*[3]	*p*[3]	*S*[4]	*d*[4]	*p*[4]
0	0	10	0	0	∞	0	0	30	0	0	100	0
1	1	10	0	0	60	1	0	30	0	0	100	0
3	1	10	0	0	50	3	1	30	0	0	90	3
2	1	10	0	1	50	3	1	30	0	0	60	2
4	1	10	0	1	50	3	1	30	0	1	60	2

- How to get the path from vertex 0 (source) to *i* (dest)?
  path[4] = 2 → path[2] = 3 → path[3] = 0,
  The path is 0, 3, 2, 4， (from 0 to 4)



8.5 Shortest Path

Floyd algorithm

- **Problem**
  - Given a directed network, compute the shortest path and path length between arbitrary vertex v_i and v_j
- **Solution**

An array of matrix

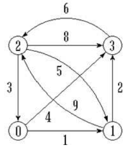
$$A^{(-1)}, A^{(0)}, \ldots, A^{(n-1)}.$$

where $A^{(-1)}[i][j] = Edge[i][j];$

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

$$k = 0, 1, \ldots, n-1$$

Example of Floyd Algorithm



$$
\begin{array}{ccccc}
0 & 1 & 2 & 3 & \\
\left(\begin{array}{cccc} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{array}\right) & & & & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}
$$

	$A^{(-1)}$				$A^{(0)}$				$A^{(1)}$				$A^{(2)}$				$A^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
2	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	$Path^{(-1)}$				$Path^{(0)}$				$Path^{(1)}$				$Path^{(2)}$				$Path^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	3	1
1	0	0	1	1	0	0	1	1	0	0	1	1	2	0	1	1	2	0	3	1
2	2	2	0	2	2	0	0	0	2	0	0	1	2	0	0	1	2	0	0	1
3	0	0	3	0	0	0	3	0	0	0	3	0	2	0	3	0	2	0	3	0

The path is 0 if the path doesn't exist

```
void Graph :: AllLengths ( int n ) {
    for ( int i = 0; i < n; i++ )       //Initialization of a and path
        for ( int j = 0; j < n; j++ ) {
            a[i][j] = Edge[i][j];
            if ( i <> j && a[i][j] < MAXINT )
                path[i][j] = i;          // if i -> j exists
            else path[i][j] = 0;
        }
    for ( int k = 0; k < n; k++ )   //compute a(k) and path(k)
        for ( i = 0; i < n; i++ )
        for ( j = 0; j < n; j++ )
            if ( a[i][k] + a[k][j] < a[i][j] ) {
                a[i][j] = a[i][k] + a[k][j];
                path[i][j] = path[k][j];
            }
}
```

8.5 Shortest Path

Analysis

- **Path**
  - 1⇒0, a[1][0]=11,
    path[1][0]=2⇒1->...->2->0
    path[1][2]=3⇒1->...->3 ->2->0
    path[1][3]=1⇒1->3

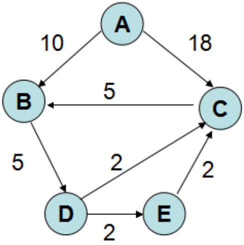
  Conclusion: path:1->3->2->0, length=11

- **Time complexity**
  - O(n^3)

- **Adjacency matrix used**

Quiz

- Based on Dijkstra algorithm, compute the shortest path from A to other vertices.
- Based on Floyd algorithm, compute the shortest paths and lengths between these vertices.



Next Section

8.6 Topological Sorting and Critical Path

- **Acyclic directed graph**
  - Characteristics
  - Applications
- **AOV network**
  - Topologic sorting
- **AOE network**
  - Project management
  - Critical path

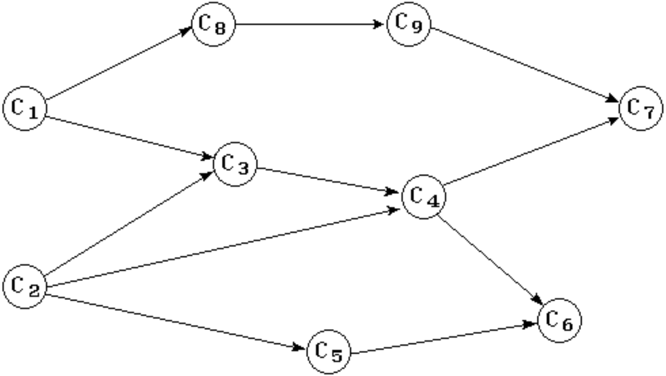
8.6 Topological Sorting and Critical Path

Example

Course No.	Name	preliminary courses
C_1	Mathematics	
C_2	Programming language	
C_3	Set and Graph	C1,C2
C_4	Data structure	C3,C2
C_5	Advanced Programming	C2
C_6	Compiler	C5,C4
C_7	Operating System	C4,C9
C_8	Generic Physics	C1
C_9	Computer Principle	C8

8.6 Topological Sorting and Critical Path

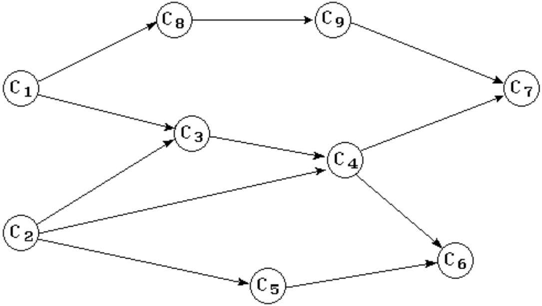
AOV network



Activity On Vertex Network

8.6 Topological Sorting and Critical Path

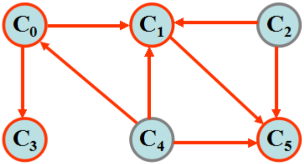
Topologic sorting



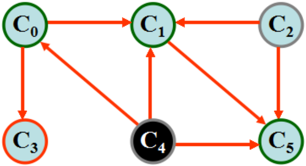
- $C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$
- $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$
- $\ldots$

8.6 Topological Sorting and Critical Path

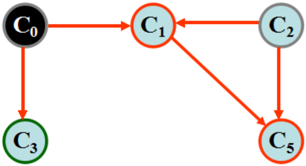
Procedure of topologic sorting



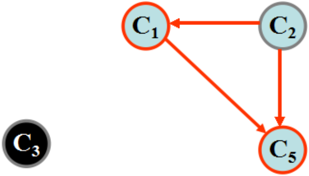
(a) Acyclic DG



(b) Output C4

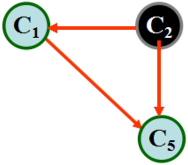


(c) Output C0

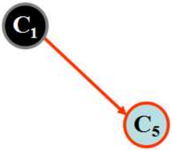


(d) Output C3

8.6 Topological Sorting and Critical Path



(e) Output C2



(f) Output C1

(g) Output C5

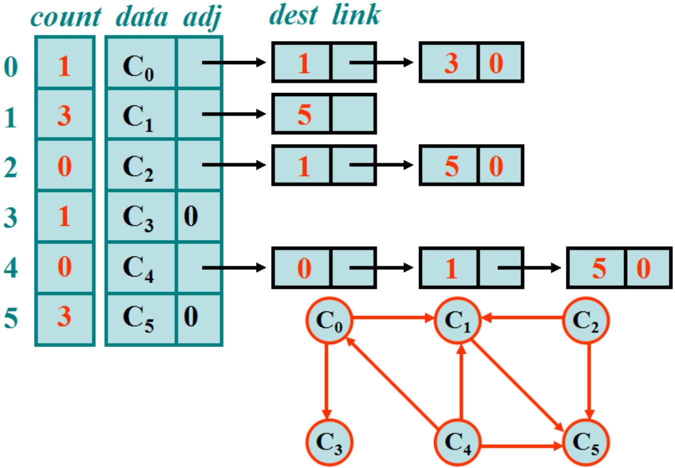
(h) Termination

Topologic sequence is

$$C_4, C_0, C_3, C_2, C_1, C_5$$

8.6 Topological Sorting and Critical Path

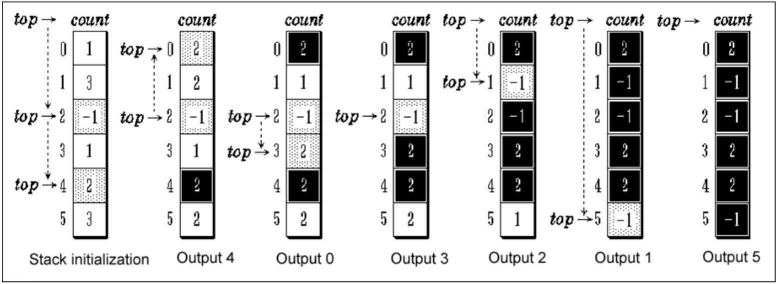
AOV network and adjacency list

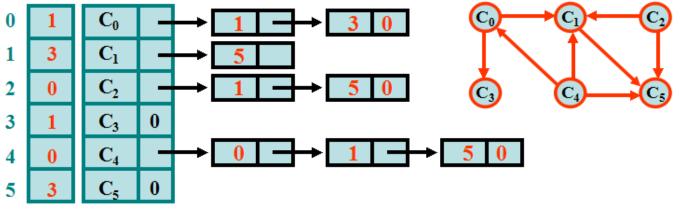


```
class Graph                                  //Graph Class
{
    friend class <int, float>Vertex;
    friend class <float>Edge;
private:
    Vertex<int, float> *NodeTable;
    int *count;
    int n;
public:
    Graph ( const int vertices = 0 ) : n ( vertices )
    {
        NodeTable = new vertex<int, float>[n];
        count = new int[n];
    };
    void TopologicalOrder ();
}
```

8.6 Topological Sorting and Critical Path

Stack involved





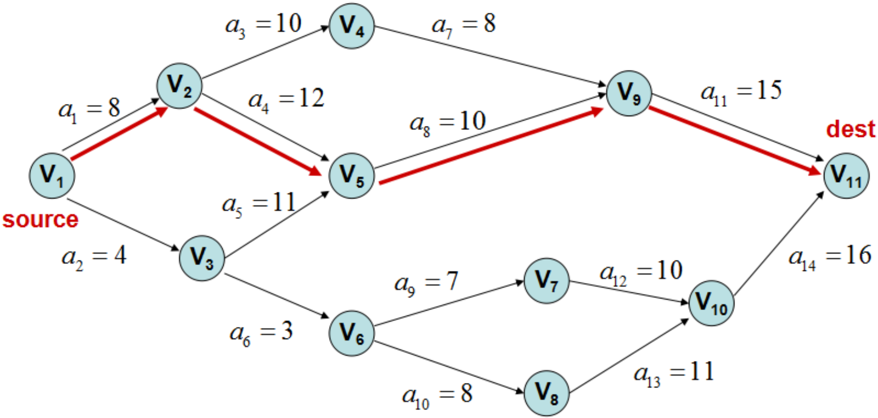
```
//Topologic sorting
void Graph :: TopologicalSort ( )
{
    int top = -1;
    for ( int i = 0; i < n; i++ )
        if ( count[i] == 0 )
        {
            count[i] = top;
            top = i;
        }
    for ( i = 0; i < n; i++ )
        if ( top == -1 )
        {
            cout << “Exit cycle ! ”Error << endl;
            return;
        }
        else
        {
            int j = top;
            top = count[top];
            cout << j << endl;                //output
```

```
            Edge<float> * l = NodeTable[j].adj;
            while (l)
            {
                int k = l->dest;
                if ( --count[k] == 0 )
                {
                    count[k] = top;
                    top = k;
                }
                l = l->link;
            }
        }
}
```

Time complexity analysis: $O(n+e)$

AOE network

Activity on edge network



8.6 Topological Sorting and Critical Path

- **Acyclic directed graph**
  - Edge – activity
  - Cost on edge – duration of activity
  - Vertex – event
- **The above mentioned graph is called as AOE network**

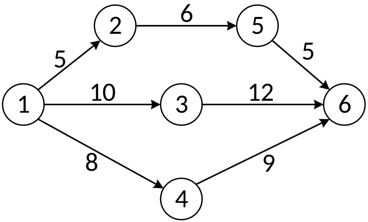
8.6 Topological Sorting and Critical Path

Application of AOE network

- **In project management**
  - How much time can the project be finished at least?
  - In order to reduce the whole duration of the project, how to prompt the specific activities?

Remarks and Concepts

- 在AOE网络中，有些活动必须顺序进行，有些活动则可以并行进行。
- 从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。
- 因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)。



8.6 Topological Sorting and Critical Path

Remarks and Concepts

- 事件V_i的最早可能开始时间$V_e(i)$
  是从源点V_0 到顶点V_i 的最长路径长度。
- 事件V_i 的最迟必须开始时间$V_l[i]$
  是在保证汇点V_{n-1} 在$V_e[n-1]$时刻完成的前提下，事件V_i的允许的最迟开始时间。
- 活动a_k 的最早可能开始时间 e[k]
  设活动a_k在边$< V_i, V_j >$ 上，则e[k] 是从源点V_0到顶点V_i的最长路径长度。因此, e[k] = $V_e[i]$。
- 活动a_k 的最迟必须开始时间$l[k]$
  $l[k]$是在不会引起时间延误的前提下，该活动允许的最迟开始时间。

$$l[k] = V_l[j] - dur(< i, j >)$$

其中，$dur(< i, j >)$是完成a_k 所需的时间。

8.6 Topological Sorting and Critical Path

- 时间余量 l[k] - e[k]
  表示活动 a_k 的最早可能开始时间和最迟必须开始时间的时间余量。$l[k] == e[k]$ 表示活动 a_k 是没有时间余量的关键活动。
  - 为找出关键活动，需要求各个活动的 e[k] 与 l[k]，以判别是否 l[k] == e[k].
  - 为求得 e[k]与 l[k]，需要先求得从源点 V0 到各个顶点 Vi 的 Ve[i] 和 Vl[i]。
  - 求Ve[i]的递推公式

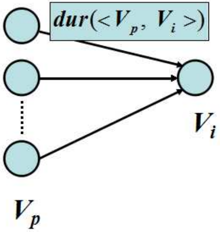
8.6 Topological Sorting and Critical Path

- 从 $V_e[0] = 0$开始，向前递推

$$V_e[i] = \max_p\{V_e[j] + dur(< V_p, V_i >)\},$$

$$< V_p, V_i > \in S_2, i = 1, 2, \ldots, n-1$$

其中, S_2是所有指向顶点V_i 的有向边$< V_p, V_i >$的集合。



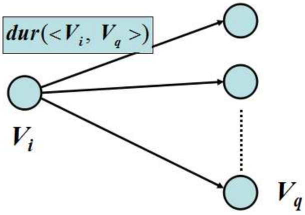
8.6 Topological Sorting and Critical Path

- 从$V_l[n-1] = V_e[n-1]$开始，反向递推

$$V_l[i] = \min_q \{V_l[j] - dur(< V_i, V_q >)\},$$
$$< V_i, V_q > \in S_1, i = n-2, n-3, \ldots, 0$$

其中, S_1是所有从顶点V_i 发出的有向边$< V_i, V_j >$的集合。

- 这两个递推公式的计算必须分别在拓扑有序及 逆拓扑有序的前提下进行。



8.6 Topological Sorting and Critical Path

- 设活动a_k $(k = 1, 2, \ldots, e)$在带权有向边$< V_i, V_j >$上, 它的持续时间用dur $(< V_i, V_j >)$ 表示，则有

$$e[k] = V_e[i];$$
$$l[k] = V_l[j] - dur(< V_i, V_j >); k = 1, 2, \ldots, e.$$

这样就得到计算关键路径的算法。

思考题：计算关键路径的方法

8.6 Topological Sorting and Critical Path

- 设活动a_k $(k=1,2,\ldots,e)$在带权有向边$<V_i,V_j>$上，它的持续时间用dur $(<V_i,V_j>)$ 表示，则有

  $$e[k]=V_e[i];$$
  $$l[k]=V_l[j]-dur(<V_i,V_j>);k=1,2,\ldots,e.$$

  这样就得到计算关键路径的算法。

- 计算关键路径时，可以一边进行拓扑排序一边计算各顶点的$V_e[i]$。为了简化算法，假定在求关键路径之前已经对各顶点实现了拓扑排序，并按拓扑有序的顺序对各顶点重新进行了编号。
  算法在求$V_e[i],i=0,1,\ldots,n-1$时按拓扑有序的顺序计算，
  在求$V_l[i],i=n-1,n-2,\ldots,0$时按逆拓扑有序的顺序计算。

8.6 Topological Sorting and Critical Path

How to get $V_e(i)$ and $V_l(i)$

- **Topologic sorting for** $V_e(i)$

  $V_e[i] = \max_p \{V_e[j] + dur(<V_p, V_i>)\},$

  $$<V_p, V_i> \in S_2, i = 1, 2, \ldots, n-1$$

- **Reverse topologic sorting for** $V_l(i)$

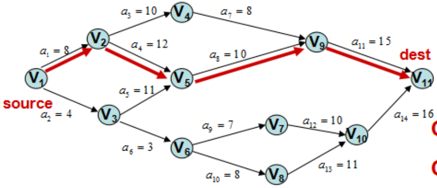
  $V_l[i] = \min_q \{V_l[j] - dur(<V_i, V_q>)\},$

  $$<V_i, V_q> \in S_1, i = n-2, n-3, \ldots, 0$$

8.6 Topological Sorting and Critical Path

events	Ve(i)	Vl(i)
V_1	0	0
V_2	8	8
V_3	4	7
V_4	18	22
V_5	20	20
V_6	7	10
V_7	14	19
V_8	15	18
V_9	30	30
V_{10}	26	29
V_{11}	45	45

activity	e(k)	l(k)	e(k)-l(k)
a_1	0	0	0
a_2	0	3	
a_3	8	12	
a_4	8	8	0
a_5	4	9	
a_6	4	7	
a_7	18	22	
a_8	20	20	0
a_9	7	12	
a_{10}	7	10	
a_{11}	30	30	0
a_{12}	14	19	
a_{13}	15	18	
a_{14}	26	29	



Critical path: a1->a4->a8->a11

Critical activities: a1, a4, a8, a11

```
//Algorithm implementation
void graph :: CriticalPath ( )
{
    int i, j;
    int p, q, k;
    float e, l;
    float * Ve = new float[n], * Vl = new float[n];
    for ( i = 0; i < n; i++ ) Ve[i] = 0;
    for ( i = 0; i < n; i++ )
    {
        Edge<float> *p = NodeTable[i].adj;
        while ( p != NULL )
        {
            k = pp->dest;
            if ( Ve[i] + pp->cost > Ve[k] )
                Ve[k] = Ve[i] + p->cost;
            p = p->link;
        }
    }

```

```
    //在此算法中需要对邻接表中单链表的结点加以
    //修改, 在各结点中增加一个域int cost, 记录该结
    //点所表示的边上的权值。

    for ( i = 0; i < n; i++ )
        Vl[i] = Ve[n-1];  //initialization

    for ( i = n-2; i; i-- )
    {
        q = NodeTable[i].adj;
        while ( q != NULL )
        {
            k = ->dest;
            if ( Vl[k] - q->cost < Vl[i])
                Vl[i] = Vl[k] - q->cost;
            q = q->link;
        }
    }

    for ( i = 0; i < n; i++ )
    {
        p = NodeTable[i].adj;
```

```
        while ( p != NULL )
        {
            k = →pdest;
            e = Ve[i];
            l = Vl[k] - p->cost;
            if ( l == e )
                cout << "<" << i << "," << k << ">" “<<
                    are critical activities”. << endl;
            p = p->link;
        }
    }
}
```

8.6 Topological Sorting and Critical Path

Analysis

- **Time complexity**
- **Topologic sorting for** $V_e(j)$
  - O(n+e)
- **Reverse topologic sorting for** $V_l(j)$
  - O(n+e)
- **Compute e(i) and l(i) for each activity**
  - O(e)
- **In total**
  - O(n+e)

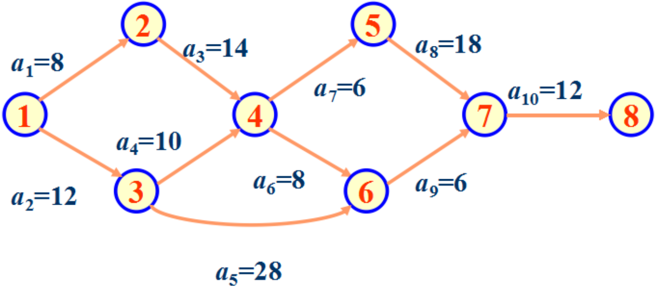
8.6 Topological Sorting and Critical Path

Problems

- **Critical activity**
  - Unique?
  - Parallel and Sequential activity?
- **Critical path**
  - Length?
  - Unique?

8.6 Topological Sorting and Critical Path

Quiz

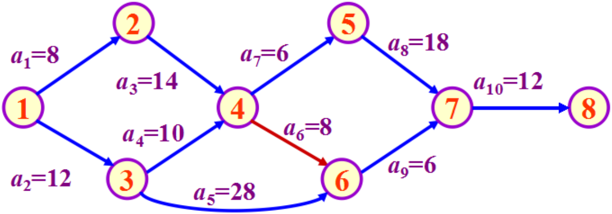


8.6 Topological Sorting and Critical Path

Answer

	1	2	3	4	5	6	7	8
V_e	0	8	12	22	28	40	46	58
V_l	0	8	12	22	28	40	46	58

	1	2	3	4	5	6	7	8	9	10
e	0	0	8	12	12	22	22	28	40	46
l	0	0	8	12	12	32	22	28	40	46



Next Section

8.7 Summary

8.7 Summary

Points

- 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 application of graphs in the problem-solving and algorithm-development phases are well understood.

8.7 Summary

- 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.

8.7 Summary

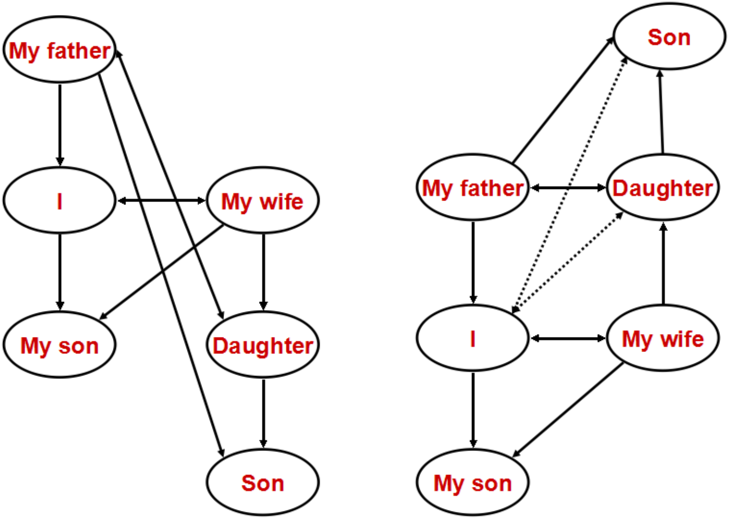
A funny story

I married a widow who had a grown-up daughter. My father, who visited us quite often, fell in love with my step-daughter and married her. Hence, my father became my son-in-law, and my step-daughter became my mother.

Some months later, my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson.

Some months later, my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson.

8.7 Summary


My father
I
My wife
My son
Daughter
Son
Son
My father
Daughter
I
My wife
My son

8.7 Summary

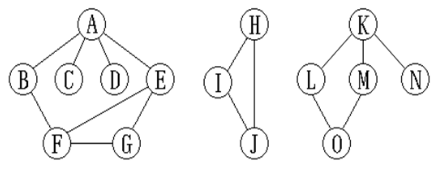
My wife is my grandmother, since she is my mother's mother. Hence, I am my wife's husband and at the same time her step-grandson; in other words,

Conclusion:

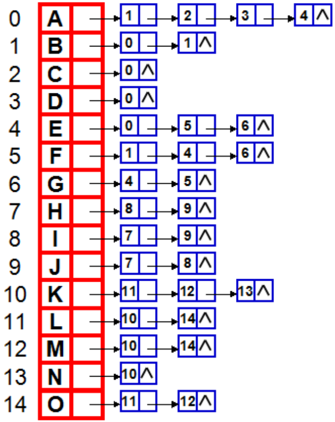
I am my own grandfather.

8.7 Summary

Quiz



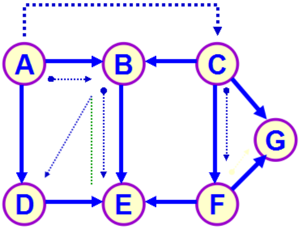
Unconnected graph



1. Draw the DFS spanning tree and corresponding binary tree.
2. Design the algorithm to store the tree into BT.

8.7 Summary

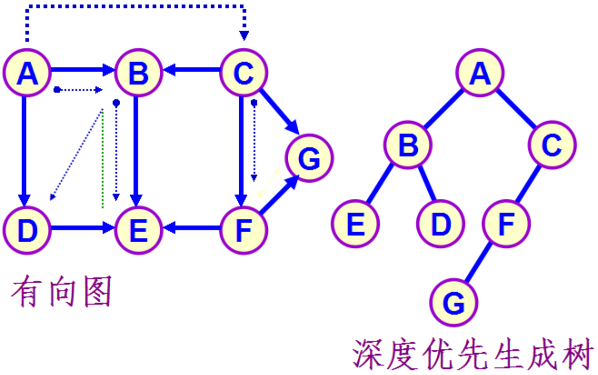
【例1】以深度优先搜索方法从出发遍历图，建立深度优先生成森林。



有向图

8.7 Summary

【例1】以深度优先搜索方法从出发遍历图，建立深度优先生成森林。



```
template<class Type>
void Graph<Type> ::
DFS_Forest ( Tree<Type> &T )
{
    TreeNode<Type> *rt, *subT;

    //创建访问标志数组
    int *visited = new int[n];

    for ( int i = 0; i < n; i++ )
        //初始化都未访问过,
        visited [i] = 0;

    //遍历所有顶点
    for ( i = 0; i < n; i++ )
        //顶点 i 未访问过
        if (!visited[i])
        {
            //原为空森林建根,
            if ( T.IsEmpty ( ) )
                //顶点 i 的值成为根 rt 的值
```

```
                subT = rt = T.BuildRoot (GetValue(i));
            else
                //顶点 i 的值成为 subT 右兄弟的值
                subT = T.InsertRightSibling(subT,
                    GetValue(i));

            //从顶点 i 出发深度优先遍历
            //建立以 为根的subT T 的子树
            DFS_Tree ( T, subT, i, visited );
        }
}

template<class Type>
void Graph<Type> :: DFS_Tree( Tree<Type> &T, TreeNode<
    Type>*RT,int i,  int visited[] )
{
    TreeNode<Type> *p;
    //顶点 i 作访问过标志
    visited [i] = 1;

    //取顶点 i 的第一个邻接顶点 w
    int w = FirstAdjvertex(i);
```

```
    // 第一个未访问子女应是i i 的左子女
    int FirstChild = 1;

    //邻接顶点 w 存在
    while(w)
    {
        if ( ! visited [w] )
        {
            // 未访问过w, 将成为 i 的子女
            if ( FirstChild )
            {
                // p 插入为 RT 的左子女
                p = T.InsertLeftChild (RT,GetValue(w));

                //建右兄弟
                FirstChild = 0;
            }
            else
                // p 插入为 p 的左子女
                p = T.InsertRightSibling( p, GetValue(w)
                    );
```

```
            //递归建立 w 的以 p 为根的子树
            DFS_Tree ( T, p, w )
        }
        //邻接顶点 w 处理完
        //取 i 的下一个邻接顶点 w
        w = NextAdjVertex ( i, w );
    }
    //回到 while 判邻接顶点 w 存在
}
```

8.7 Summary

Assignments and Experiments

- **Assignments**

- **Experiments**

  Based on the following input, try to create an adjacency list with in degree info.

  - **N vertices**
    - v_i
  - **E edges**
    - $< v_i, v_j >$

8.7 Summary

Experiments

- 设有一个有向图存储在邻接表中。试设计一个算法，按深度优先搜索策略对其进行拓扑排序。并以下图为例检验你的算法的正确性。