



Chapter 8

Graphs



- 
- **Definition of graphs and some terminology**
 - **Graph representations**
 - **Some algorithms**

9.1 Definition and terminologies

1. Definition of graphs:

Graph=(V, E)

V: nonempty finite vertice set

E: edge set

- **Undirected Graph:**

If the tuple denoting an edge is unordered, then (v_1, v_2) and (v_2, v_1) are the same edge.

9.1 Definition and terminologies

- **Directed graph:**

If the tuple denoting an edge is ordered, then

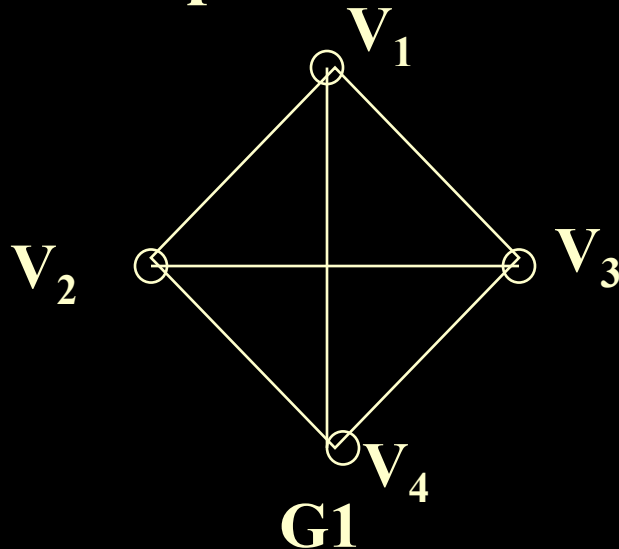
$\langle v1, v2 \rangle$ and $\langle v2, v1 \rangle$ are different edges.

$v1$:始点

$v2$:终点

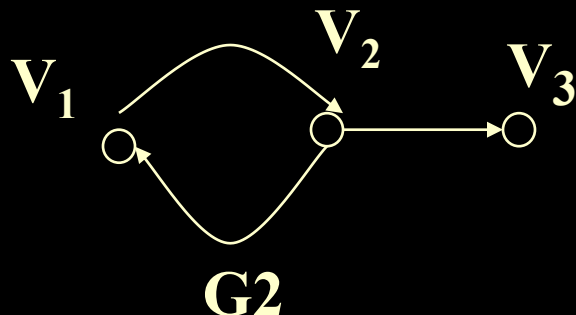
9.1 Definition and terminologies

Example:



$$V(G_1) = \{V_1, V_2, V_3, V_4\}$$

$$E(G_1) = \{(V_1, V_2), (V_1, V_3), (V_1, V_4), (V_2, V_3), (V_2, V_4), (V_3, V_4)\}$$

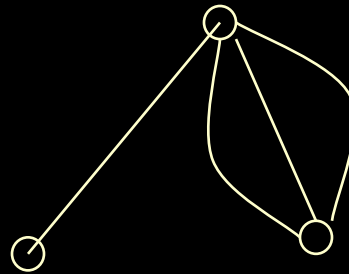
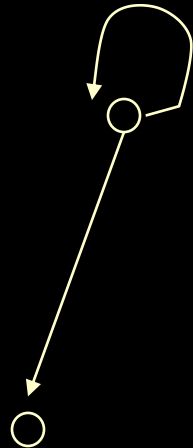


$$V(G_2) = \{V_1, V_2, V_3\}$$

$$E(G_2) = \{\langle V_1, V_2 \rangle, \langle V_2, V_1 \rangle, \langle V_2, V_3 \rangle\}$$

9.1 Definition and terminologies

We will not discuss graphs of the following types

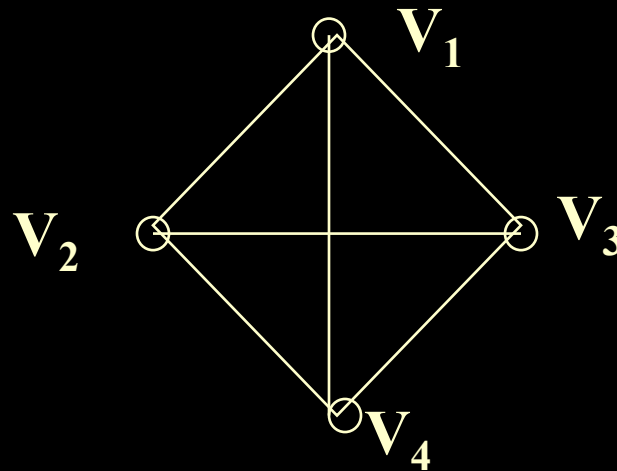


多重图

9.1 Definition and terminologies

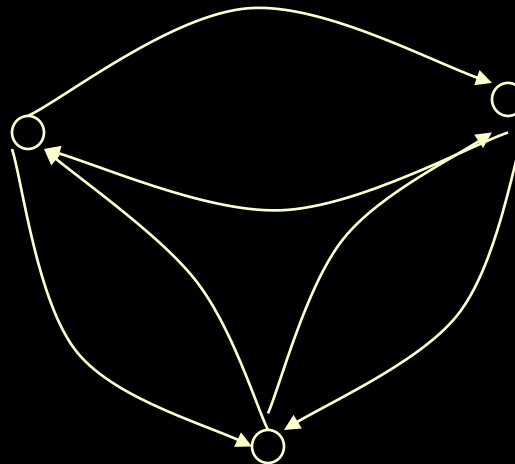
2. Complete graph

In an undirected graph with n nodes, the number of edges $\leq n*(n-1)/2$. If “=” is satisfied, then it is called a complete undirect graph.



9.1 Definition and terminologies

In a directed graph with n nodes, the number of edges $\leq n*(n-1)$. If “=” is satisfied, then it is called a complete directed graph.



9.1 Definition and terminologies

3. degree d_v of vertex v , $TD(v)$:

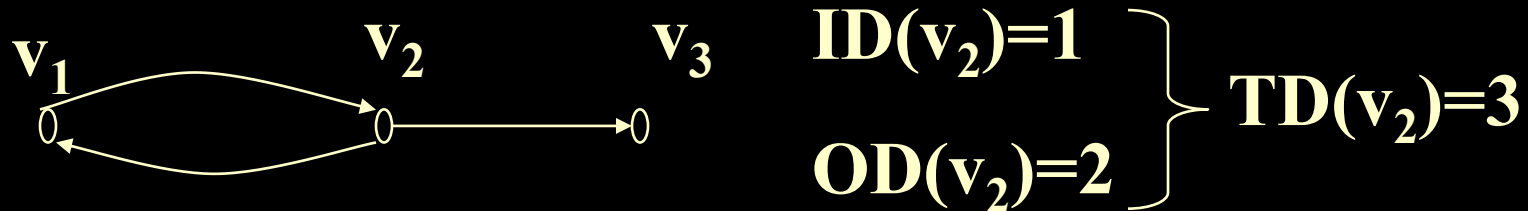
is the number of edges incident on vertex v .

In a directed graph :

- in-degree of vertex v is the number of edges incident to v , $ID(v)$.
- out-degree of vertex v is the number of edges incident from the v , $OD(v)$.

9.1 Definition and terminologies

$$TD(v) = ID(v) + OD(v)$$



Generally, if there are n vertices and e edges in a graph, then

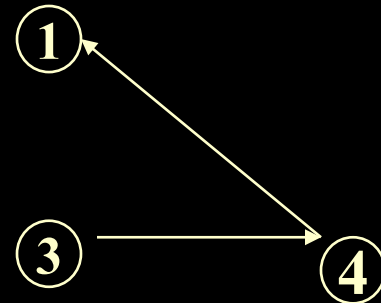
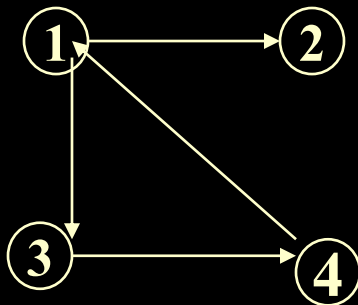
$$e = \left(\sum_{i=1}^n TD(v_i) \right) / 2$$

9.1 Definition and terminologies

4. Subgraph

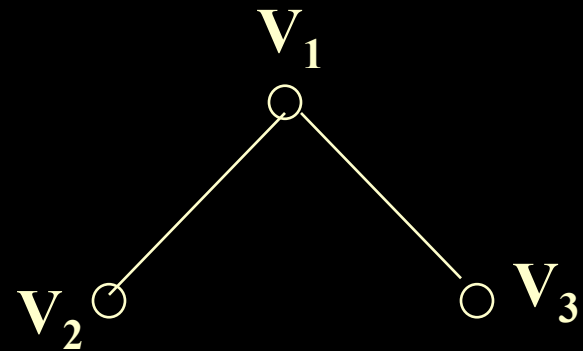
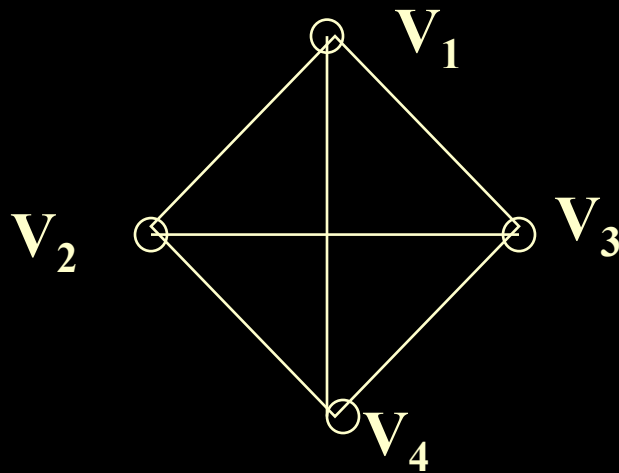
Graph $G=(V,E), G'=(V',E')$, if $V' \subseteq V, E' \subseteq E$, and the vertices incident on the edges in E' are in V' , then G' is the subgraph of G .

For example:



9.1 Definition and terminologies

Another example:



9.1 Definition and terminologies

5. path.

A sequence of vertices $P=i_1, i_2, \dots, i_k$ is an i_1 to i_k path in the graph of graph $G=(V, E)$ iff the $\text{edge}(i_j, i_{j+1})$ is in E for every j , $1 \leq j < k$.

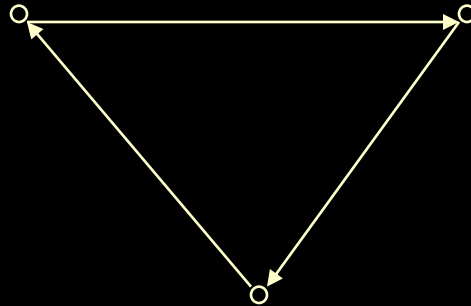
6. Simple path and cycle

A Simple path is a path in which all vertices except possibly the first and last, are different.

A Simple cycle is a simple path with the same start and end vertex.

9.1 Definition and terminologies

An example of a cycle



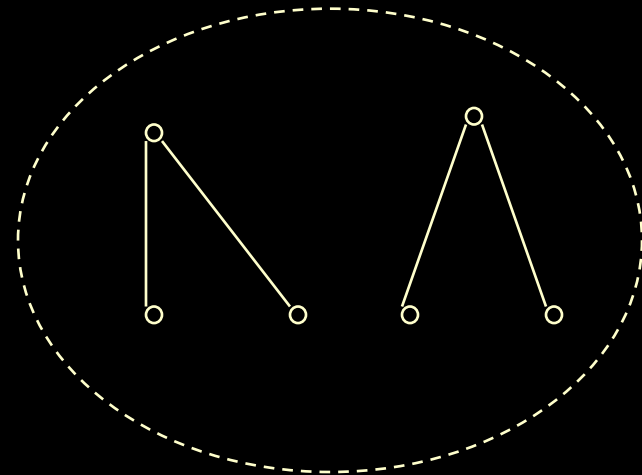
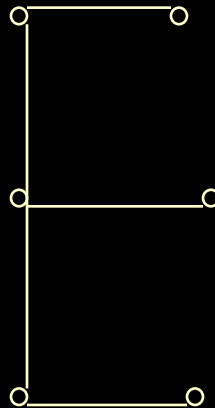
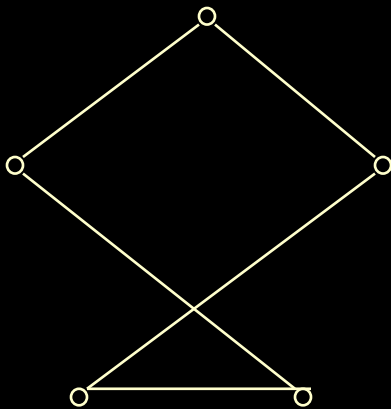
9.1 Definition and terminologies

7. Connected graph & Connected component

- In a undirected graph, if there is a path from vertex v_1 to v_2 , then v_1 and v_2 are connected.
- In a undirected graph ,if two arbitrary vertices are connected, then the graph is a connected graph.

9.1 Definition and terminologies

Examples :



Non-connected graph

Maximum connected subgraph(极大连通子图) is called connected component.

9.1 Definition and terminologies

8. Strong connected graph and strongly connected component

- A digraph is strongly connected iff it contains a directed path from i to j and from j to i for every pair of distinct vertices i and j .
- The maximum strong connected subgraph (极大强连通子图) of a non-strongly connected graph is called strongly connected component (强连通分量).

9.1 Definition and terminologies

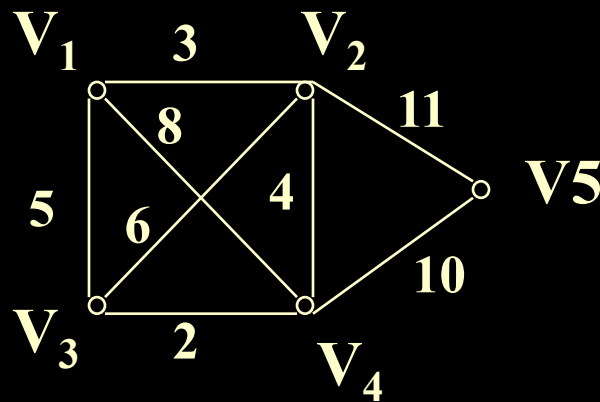


9. Network

- When weights and costs are assigned to edges, the resulting data object is called weighted graph and weighted digraph.
- The term network refers to weighted connected graph and weighted connected digraph.

9.1 Definition and terminologies

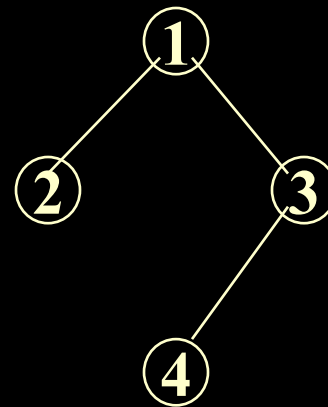
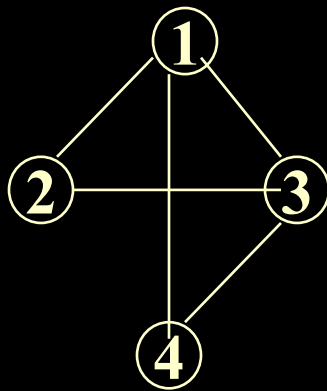
Example of a network



9.1 Definition and terminologies

10. Spanning tree

- A spanning tree of a connected graph is its minimum connected subgraph(极小连通子图). An n -vertex spanning tree has $n-1$ edges.
- For example:



9.2 ADT Graph and Digraph

AbstractDataType Graph

{

instances

a set V of vertices and a set E of edges

operations

Create(n):create an undirected graph with n vertices and no edges

Exist(i,j): return true if edge(i,j)exists; false otherwise

Edges():return the number of edges in the graph

Vertices():return the number of vertices in the graph

Add(i,j): add the edge(i,j) to the graph

Delete(i,j):delete the edge (i,j)

Degree(i): return the degree of vertex i

InDegree(i): synonym for degree

OutDegree(i): synonym for degree

}

9.3 Representation of graphs and digraphs

1. Adjacency Matrix

$$G=(V,E), V=\{V_1, V_2, \dots, V_n\}$$

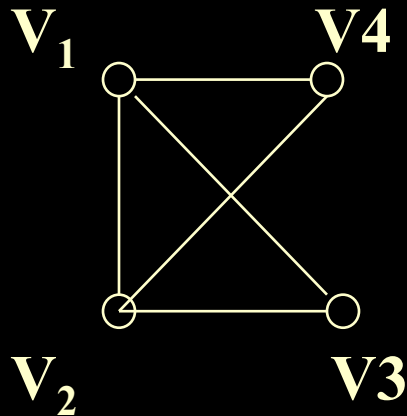
then the adjacency matrix of graph G :

$$A(i,j)=\begin{cases} 1 & \text{if } \langle i,j \rangle, \langle j,i \rangle \in E \text{ or } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

9.3 Representation of graphs and digraphs

For example:

graph



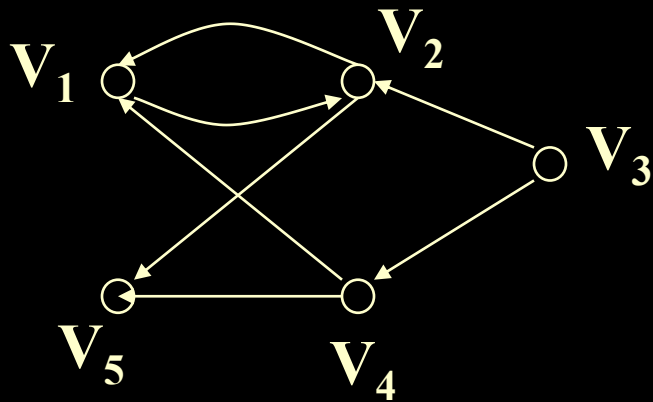
$$A(i,j) = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

1) Adjacency matrix of graph is a symmetric matrix

2) $\sum_{j=1}^n A(i,j) = \sum_{j=1}^n A(j,i) = d_i$ (degree of vertex i)

9.3 Representation of graphs and digraphs

Digraph



$A(i,j)=$

0	1	0	0	0
1	0	0	0	1
0	1	0	1	0
1	0	0	0	1
0	0	0	0	0

$$\sum_{j=1}^n A(i,j) = d_i^{\text{out}}$$

$$\sum_{j=1}^n A(j,i) = d_i^{\text{in}}$$

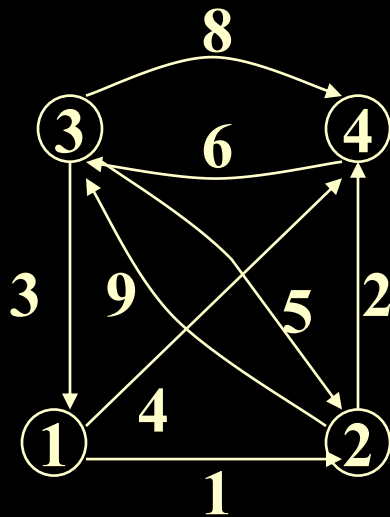
9.3 Representation of graphs and digraphs

Representation of networks, replace 1 with weights, others with ∞

$$A(i,j)=\begin{cases} W(i,j) & \text{if } i \neq j \text{ and } \langle i,j \rangle, \langle j,i \rangle \in E \text{ or } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

9.3 Representation of graphs and digraphs

For example:



$A(i,j)=$

∞	1	∞	4
∞	∞	9	2
3	5	∞	8
∞	∞	6	∞

除以上的邻接矩阵外，还要
一个记录各顶点信息的表，
一个当前的边数

其类说明：

```
const int MaxNumEdges=50 // 最大边数
```

```
const int MaxNumVertices=10 //最大顶点数
```

```
template<class NameType, class DistType> class Graph
```

```
{ private:
```

```
    SeqList<NameType> VerticesList(MaxNumVertices) //顶点表
```

```
    DistType Edge [MaxNumVertices] [MaxNumVertices] //邻接矩阵
```

```
    int CurrentEdges; //当前边数
```

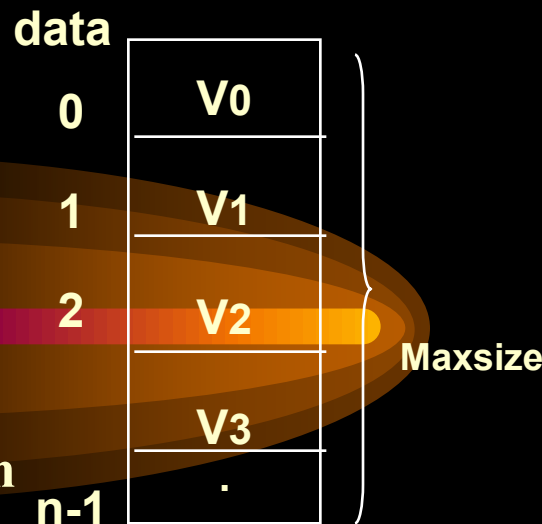
```
    int FindVertex (SeqList <NameType> &L; const NameType &Vertex)
```

```
        {return L.Find(Vertex);}
```

```
    int GetVertexPos (const NameType &Vertex)
```

```
        {return FindVertex(VerticesList);}
```

```
        // 给出了顶点Vertex在图中的位置
```



public:

```
Graph (const int sz=MaxNumEdges);  
int GraphEmpty() const {return VerticesList.IsEmpty();}  
int GraphFull() const{return VerticesList.IsFull() ||  
CurrentEdges==MaxNumEdges;}  
  
int NumberofVertices() {return VerticesList.last;}  
int NumberofEdges() {return CurrentEdges;}  
NameType Getvalue(const int i)  
{return i>=0 && i<VerticesList.last ? VerticesList.data[i] : NULL;}  
DistType Getweight (const int v1,const int v2);  
  
int GetFirstNeighbor(const int v);  
int GetNextNeighbor(const int v1,const int v2);  
  
void InsertVertex(const NameType & Vertex);  
void InsertEdge(const int v1,const int v2, DistType weight);  
void removeVertex(const int v);  
void removeEdge(cosnt int v1,const int v2);  
}
```

- **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;
    currentEdge=0;
}
```

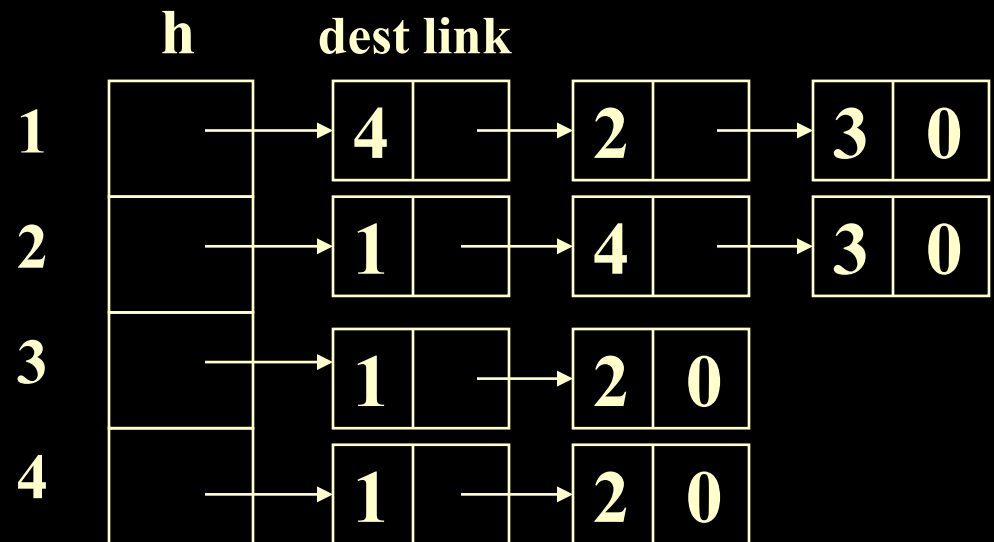
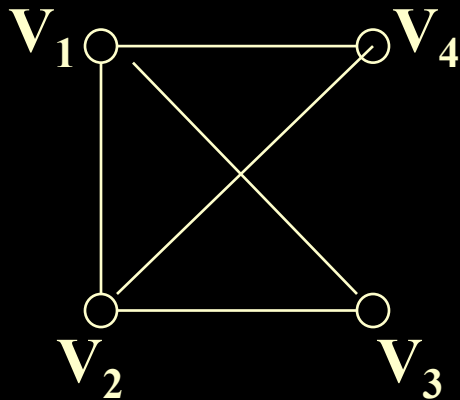
- **求顶点V的第一个邻接顶点的位置**

```
template<class NameType, class DistType>
    int Graph<NameType, DistType> :: GetFirstNeighbor(int v)
{   if (v!=-1)
        {   for( int col=0; col<CurrentVertices; col++)
                if (Edge[v][col]>0 && Edge[v][col]<max) return col;
        }
    return -1;
}
```

9.3 Representation of graphs and digraphs

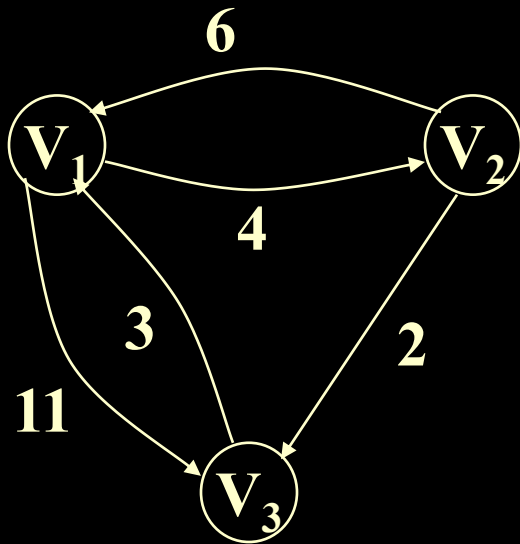
2. Linked-adjacency Lists

reduce the storage requirement if the number of edges in the graph is small.



9.3 Representation of graphs and digraphs

Digraph:



	h	dest	cost	link
1		2	4	3 11 0
2		1	6	3 2 0
3		1	3	0

	h	dest	cost	link
1		2	6	3 3 0
2		1	4	0
3		1	11	2 2 0

又称逆邻接表

关于邻接表的声明

```
const int Defaultsize=10;  
template <class NameType, class DistType> class Graph;
```

```
template <class DistType> struct Edge //边的定义
```

```
{ friend class Graph <NameType,DistType>;
```

边的三个数据成员 {

```
    int dest;           //边的另一顶点在顶点表中的位置  
    DistType cost;      //边上的权  
    Edge<DistType> *link; //下一条边的链指针
```

```
    Edge() {}
```

```
    Edge(int D,DistType C):dest(D),cost(C),link(NULL){}
```

```
    int operate != (const Edge<DistType> &E) const  
        {return dest != E.dest; }
```

```
}
```

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

```
{ friend class Edge<DistType>;
```

```
    friend class Graph<NameType, DistType>;
```

```
    NameType data;           //顶点名字
```

```
    Edge<DistType> *adj;     //出边表头指针
```

```
}
```


Node Table:

data	adj

dest cost link

--	--	--

图的类定义

```
template<class NameType, class DistType> class Graph
```

```
{ private:
```

图的私有
数据成员

```
    Vertex<NameType, DistType> *NodeTable; //顶点表
```

```
    int NumVertices; //当前顶点数
```

```
    int MaxNumVertices; //最大顶点个数
```

```
    int NumEdges; //当前边数
```

```
    int GetVertexpos(const Type &Vertex);
```

```
public:
```

```
    Graph(int sz);
```

```
    ~Graph();
```

```
    int GraphEmpty() const {return NumVertices==0;}
```

```
    int GraphFull() const
```

```
        {return NumVertices==MaxNumVertices;}
```

```
    int NumberOfVertices() {return NumVertices;}
```

```
    int NumberOfEdges() {return NumEdges;}
```

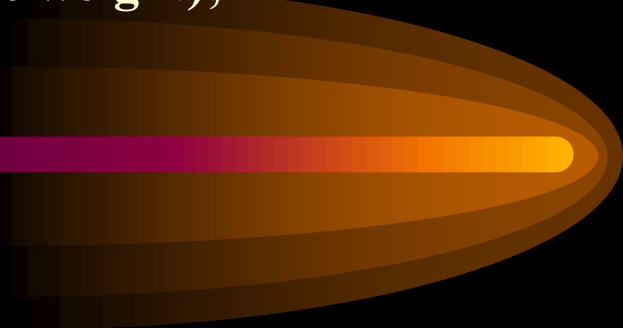
```
    NameType GetValue(const int i)
```

```
        {return i>=0 && i<NumVertices ? NodeTable[i].data: NULL;}
```

```
    void InsertVertex(const 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 GetFristNeighbor(int v);  
int GetNextNeighbor(int v1,int v2);  
}
```



1) Constructor

```
template<class NameType,class DistType>
Graph<NameType,DistType> ::
Graph(int sz=Defaultsize) :
    NumVertices(0),MaxNumVertices(sz),NumEdge(0)
{
    int NumVertices, NumEdges, k, j;
    NameType name,tail,head;
    DistType weight;
    NodeTable=newVertex<NameType>[MaxNumVertices]

    cin>>NumVertices;    //输入顶点数
    for( int i=0; i<NumVertices; i++ )
    {
        cin>>name;
        InsertVertex(name);
    }
    cin>>NumEdges; //输入边数
    for( i=0; i<NumEdges; i++)
    {
        cin>>tail>>head>>weight;
        k=GetVertexpos(tail);
        j=GetVertexpos(head);
        InsertEdge(k,j,weight);
    }
}
```

- **int Graph<NameType,DistType> ::
 GetVertexpos(const NameType &Vertex)
{ for(int i=0; i<NumVertices; i++)
 if (NodeTable[i].data==Vertex) return i;
 return -1;
}**

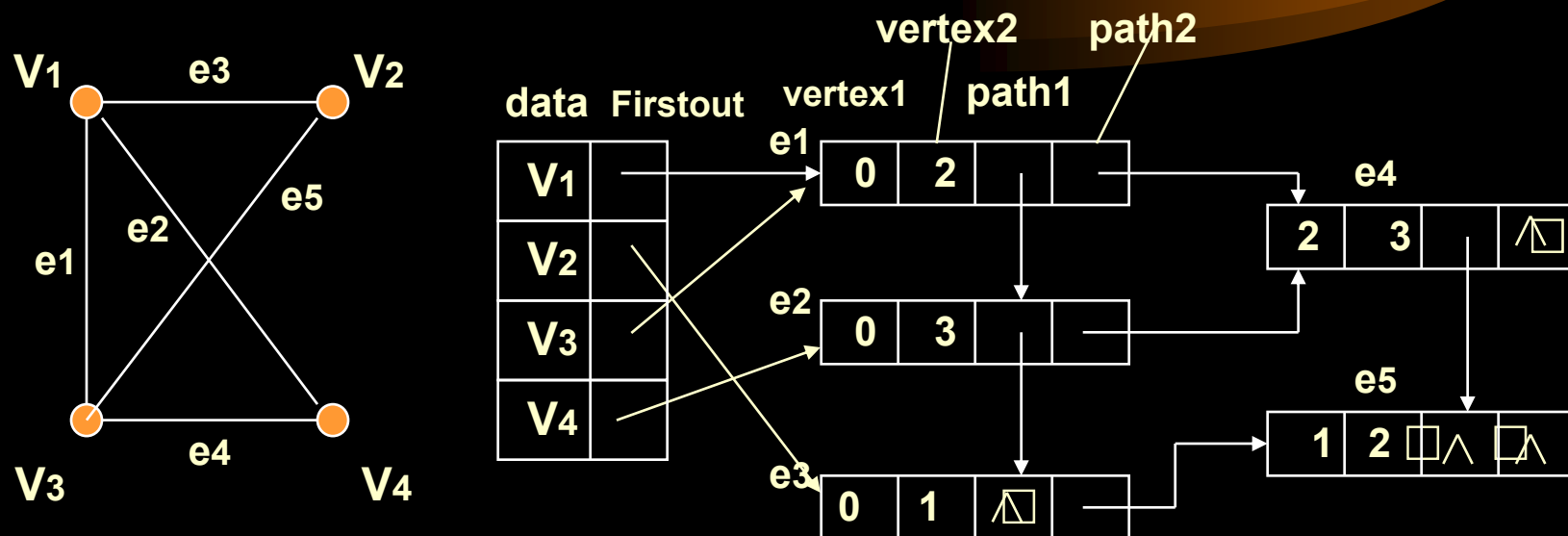
2) 给出顶点V的第一个邻接顶点的位置

```
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
}
```

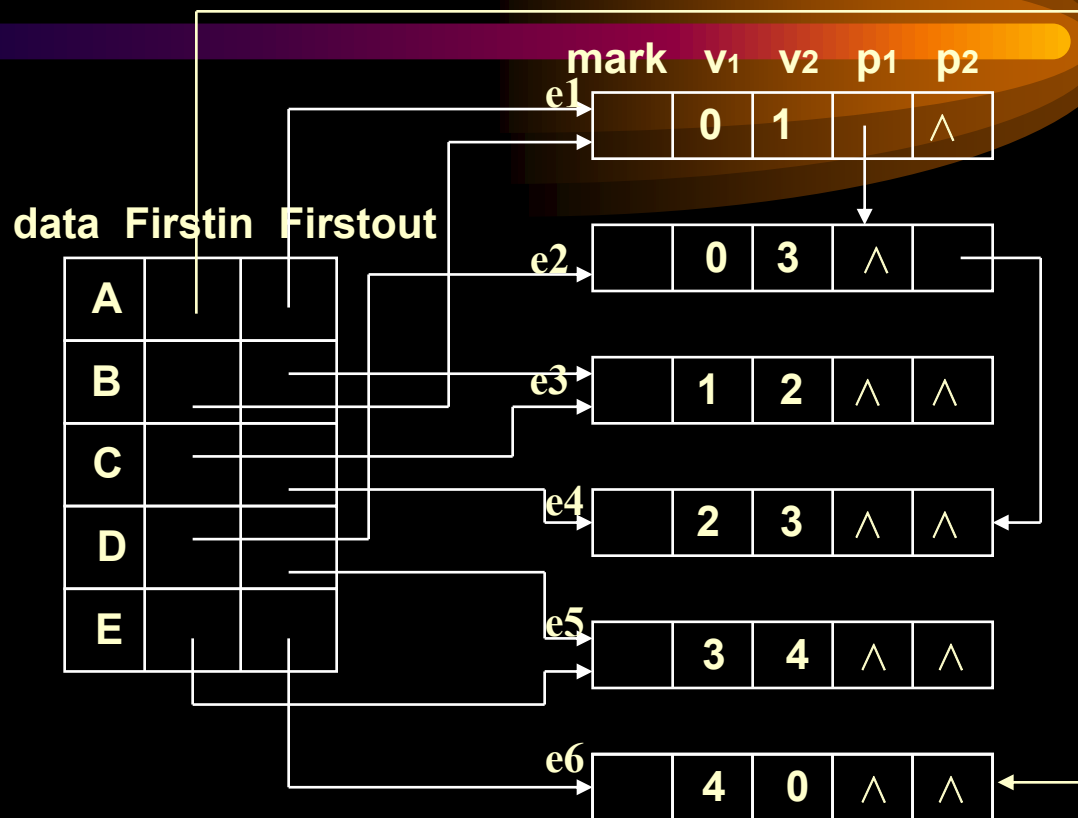
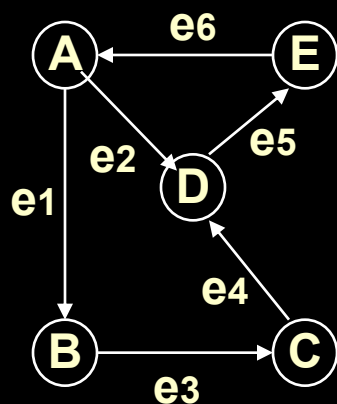
- **template<class NameType,class DistType>**
int Graph<NameType,DistType>::GetNextNeighbor
(int v1,int v2)
{ if (v1!=-1)
{ Edge<DistType> *p=NodeTable[v1].adj;
while (p!=NULL)
{ if(p→dest==v2 && p→link!=NULL)
return p→link→dest;
else p=p→link;
}
}
return -1;
}

3. 邻接多重表 (adjacency multilist)

在无向图中, 如果边数为 m , 则在邻接表表示中需 $2m$ 个单位来存储. 为了克服这一缺点, 采用邻接多重表, 每条边用一个结点表示.



对有向图而言，需用邻接表和逆邻接表，如果把这两个表结合起来用有向图的邻接多重表(也称为十字链表)来表示一个有向图。



9.4 图的遍历与连通性

图的遍历 (Graph Traversal):

从图中某一顶点出发访问图中其余顶点,且使每个顶点仅被访问一次.

遍历二叉树: 前序, 中序, 后序.

遍历树: 深度优先遍历 (先根,后根)
广度优先遍历

遍历森林: 深度优先遍历 (先根,中根,后根)
广度优先遍历

遍历图: 深度优先遍历 DFS (Depth First Search)
广度优先遍历 BFS (Breadth First Search)

1.深度优先搜索 (depth-first-search)

思想：从图中某个顶点 V_0 出发,访问它,然后选择一个

V_0 邻接到的未被访问的一个邻接点 V_1 出发深度优先遍

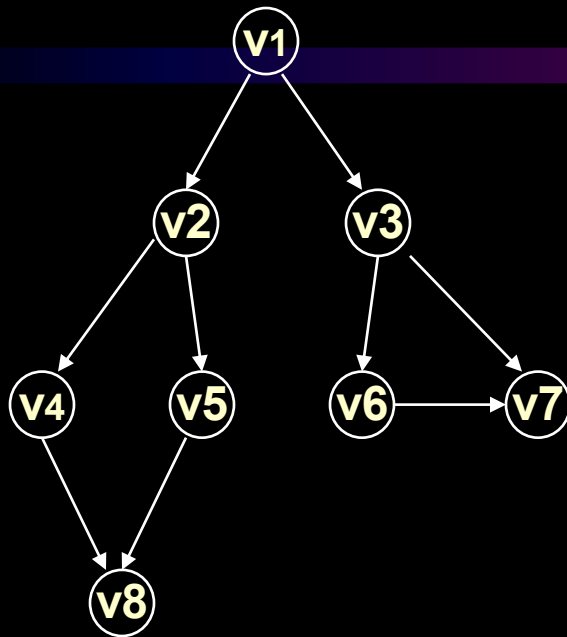
历图,当遇到一个所有邻接于它的结点都被访问过了的

结点 U 时,回退到前一次刚被访问过的拥有未被访问的

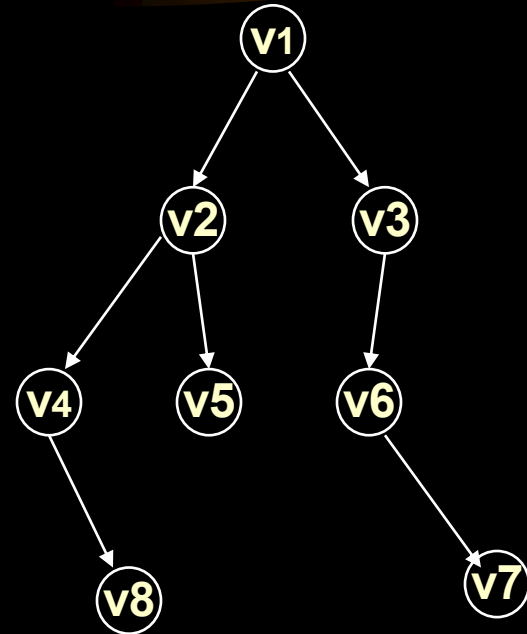
邻接点 W ,再从 W 出发深度遍历,.....直到连通图中的所

有顶点都被访问过为止.

以有向图为例：



$V1 \rightarrow V2 \rightarrow V4 \rightarrow V8 \rightarrow V5 \rightarrow V3 \rightarrow V6 \rightarrow V7$



深度优先生成树

递归方法实现

算法中用一个辅助数组visited[]:

{ 0:未访问
1:访问过了

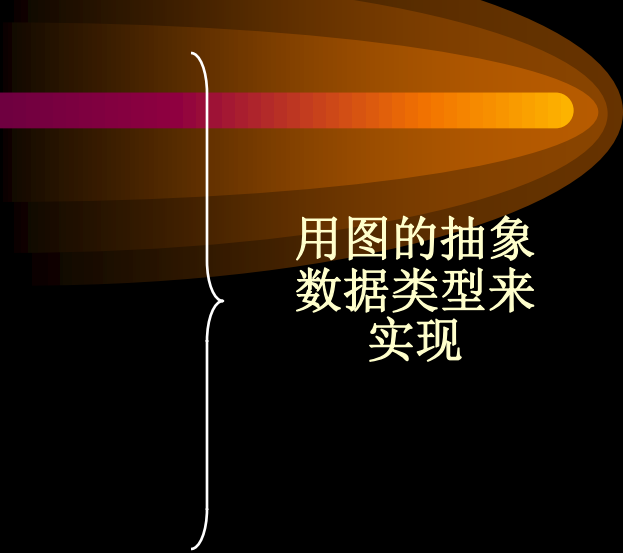
假设图为连通图

主过程:

```
template<NameType,DistType>
    void Graph<NameType,DistType> :: DFS( )
{   int *visited=new int[NumVertices];
    for ( int i=0; i<NumVertices; i++) visited[i]=0;
    DFS(0,visited);    //从顶点0开始深度优先搜索
    delete[] visited;
}
```

子过程:

```
template<NameType,DistType> void Graph<NameType,DistType> ::  
    DFS(int v, visited[])  
{    cout<<GetValue(v)<<"";  
    visited[v]=1;  
    int w=GetFirstNeighbor(v);  
    while (w!=-1)  
    {    if(!visited[w]) DFS(w,visited);  
        w=GetNextNeighbor(v,w);  
    }  
}
```



用图的抽象
数据类型来
实现

算法分析:

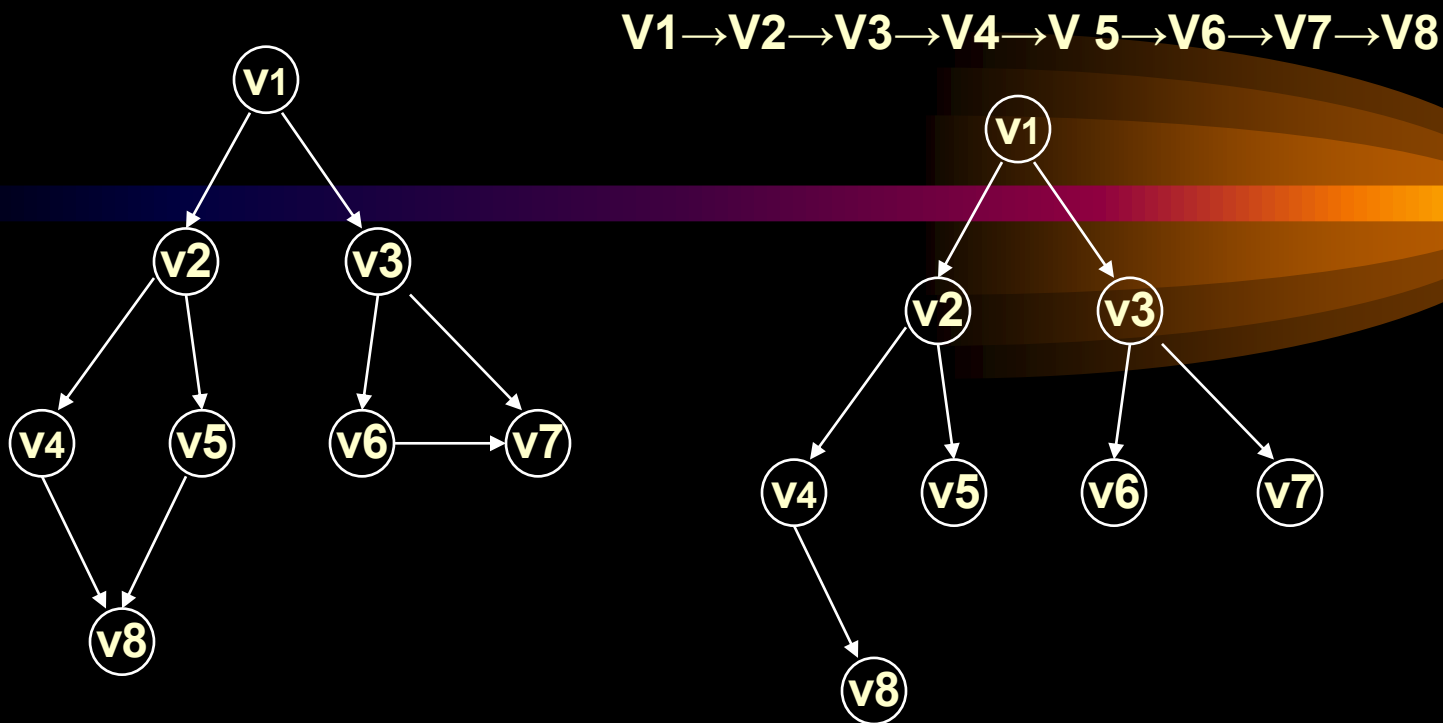
用邻接表表示 $O(n+e)$

用邻接矩阵表示 $O(n^2)$

2. 广度优先遍历 (breadth search)

思想：从图中某顶点 v_0 出发，在访问了 v_0 之后依次访问 v_0 的各个未曾访问过的邻接点，然后分别从这些邻接点出发广度优先遍历图，直至图中所有顶点都被访问到为止。

例子



广度优先生成树

算法同样需要一个辅助数组 **visited[]** 表示顶点是否被访问过。
还需要一个队列,记正在访问的这一层和上一层的顶点。
算法显然是非递归的。


```
template<NameType,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;
}
```

算法分析： 每个顶点进队列一次且只进一次， \therefore 算法中循环语句至多执行 n 次。

从具体图的存储结构来看：

1) 如果用邻接表： $O(n+e)$

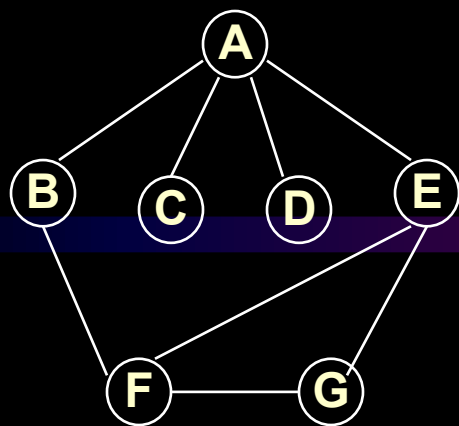
2) 如果用邻接矩阵：对每个被访问过的顶点，循环检测矩阵中 n 个元素， \therefore 时间代价为 $O(n^2)$

3. 连通分量

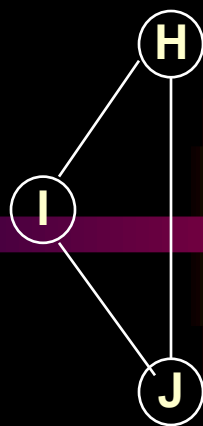
以上讨论的是对一个无向的连通图或一个强连通图的有向图进行遍历，得到一棵深度优先或广度优先生成树。

但当无向图(以无向图为例)为非连通图时，从图的某一顶点出发进行遍历(深度，广度)只能访问到该顶点所在的最大连通子图(即连通分量)的所有顶点。

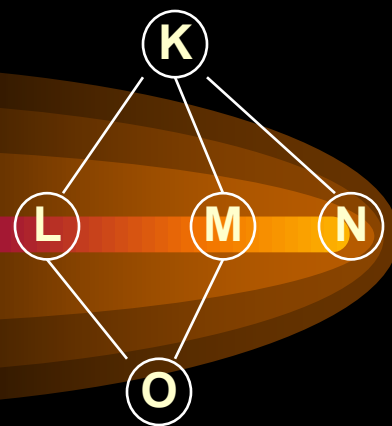
例子:



A,B,F,G,E,C,D

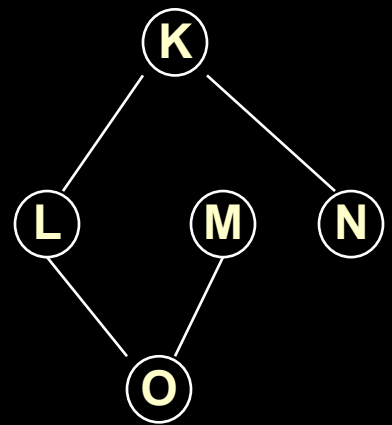
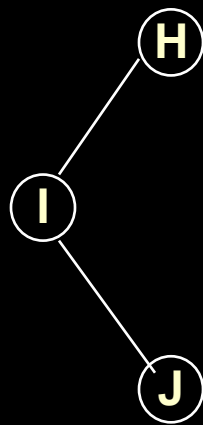
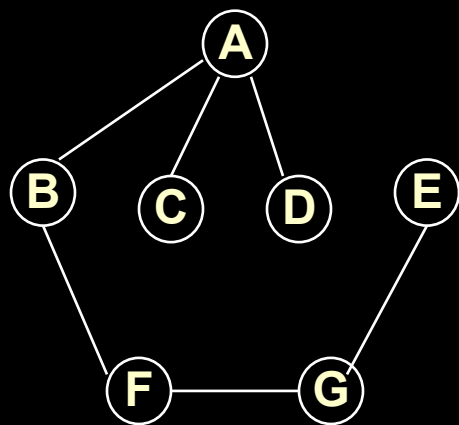


H,I,J



K,L,O,M,N

非连通无向图



非连通图的连通分量

下面是利用深度优先搜索求非连通图的连通分量算法
实际上只要加一个循环语句就行了.

```
Template<NameType,DistType>
    void Graph<NameType,DistType> :: components()
{   int* visited=new int[NumVertices];
    for (int i=0; i<NumVertices; i++) visited[i]=0;
    for (i=0; i<NumVertices; i++)
        if (!visited[i])
        { DFS(i,visited);
          outputNewComponent();
        }
    delete[] visited;
}
```

9.5 最小生成树

1. 生成树的有关概念

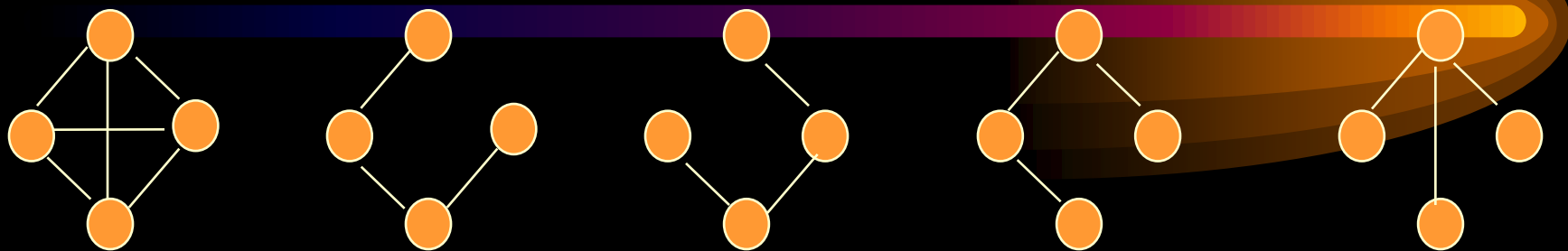
- 生成树的定义

设 $G=(V, E)$ 是一个连通的无向图（或是强连通有向图）从图 G 中的任一顶点出发作遍历图的操作，把遍历走过的边的集合记为 $TE(G)$ ，显然 $G'=(V, TE)$ 是 G 之子图， G' 被称为 G 的生成树（spanning tree），也称为一个连通图。

n 个结点的生成树有 $n-1$ 条边。

生成树的代价： $TE(G)$ 上诸边的代价之和

- 生成树不唯一

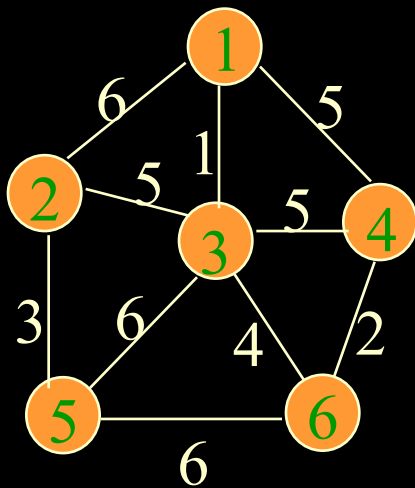


- 最小代价生成树（minimun-cost spanning tree）

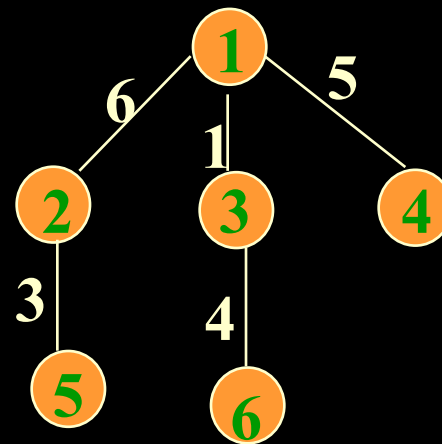
问题的提出：如何找到一个网络的最小生成树，即各边权的总和为最小的生成树

实际例子：6个城市已固定，现从一个城市发出信息到每一个城市如何选择或铺设通信线路，使花费（造价）最低。

前提：每条边有代价

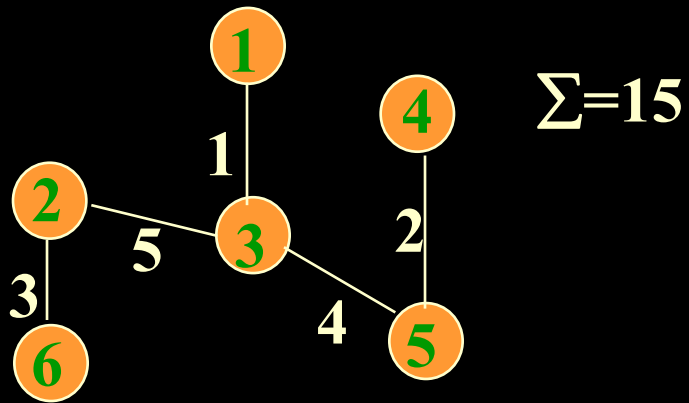


如果用广度优先，则



$$\Sigma=19$$

最小代价生成树



两个算法: Prim,
Kruskal.

它们都采用了逐步求解 (Greedy) 的策略。

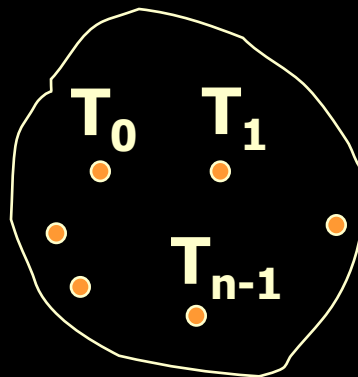
Grandy策略:

设: 连通网络 $N=\{V,E\}$, V 中有 n 个顶点。

- 1) 先构造 n 个顶点, 0条边的森林 $F=\{T_0, T_1, \dots, T_{n-1}\}$
- 2) 每次向 F 中加入一条边。该边是一端在 F 的某棵树 T_i 上而另一端不在 T_i 上的所有边中具有最小权值的边。

这样使 F 中两棵树合并为一棵, 树的棵数-1

3) 重复 $n-1$ 次



最小生成树的类声明:

```
const int MAXINT=机器可表示的, 问题中不可能出现的大数
class MinSpanTree;
class MSTEdgeNode
{ friend class MinSpanTree;
  private :
    int tail ,head;
    int cost;
};
```

边的结构:

tail	head	cost
------	------	------

```
class MinSpanTree
```

```
{ public :
```

```
    MinSpanTree(int SZ = NumVertices-1): MaxSize(SZ), n(0)
```

```
        {edgevalue = new MSTEdgeNode[MaxSize];}
```

```
protected:
```

```
    MSTEdgeNode* edgevalue;
```

//边值数组

```
    int MaxSize, n;
```

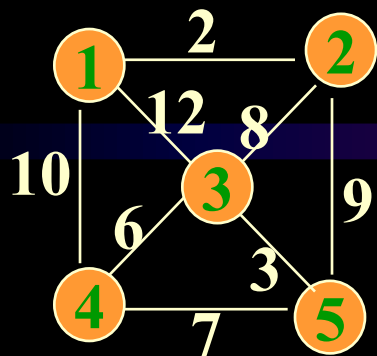
//数组的最大元素个数和

//当前个数

```
};
```

Kruskal算法

1)把无向图中的所有边排序



2 3 6 7 8 9 10 12
(1,2)(3,5)(3,4)(4,5)(2,3)(2,5)(1,4)(1,3)

2)一开始的最小生成树为

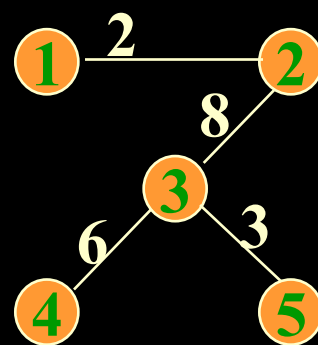
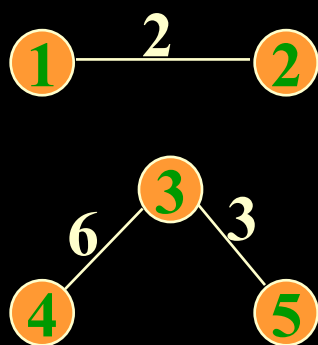
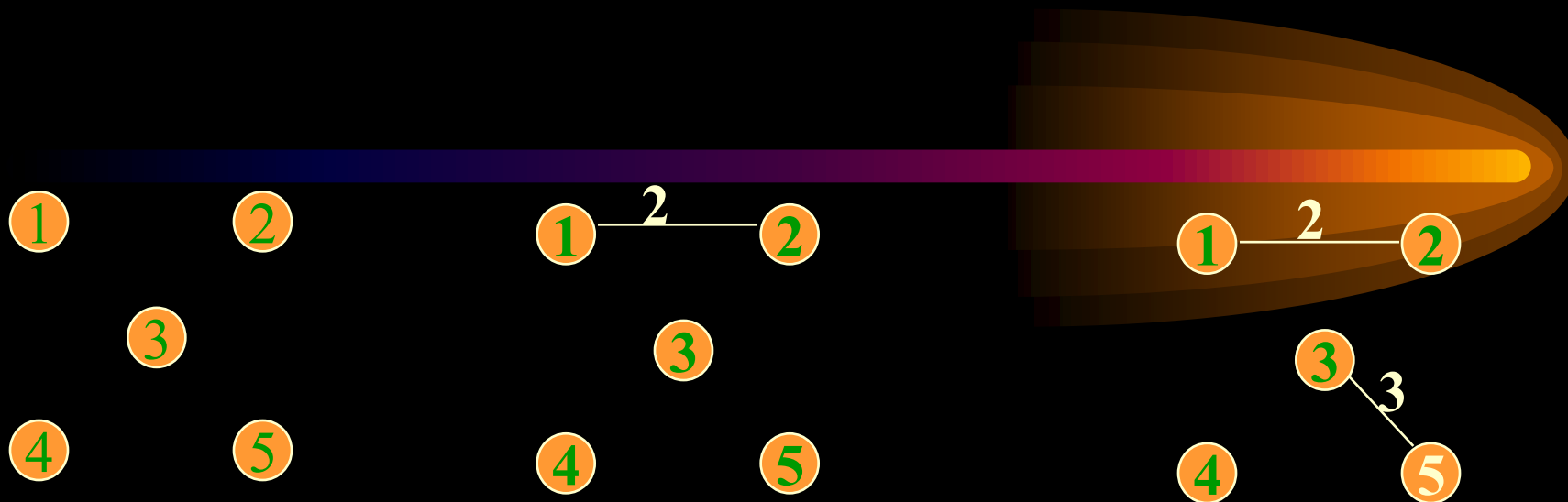
$$T \leftarrow (V, TE) \\ TE = \emptyset$$

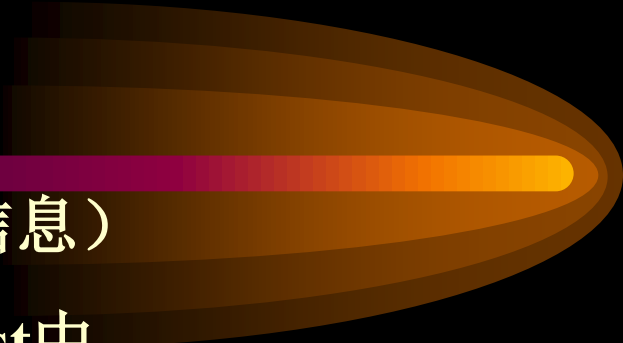
即由 n 个不同的连通分量组成

3)在 E 中选一条代价最小的边 (u,v) 加入 T ，一定要满足 (u,v) 不和 TE 中已有的边构成回路，

$$E \leftarrow E - \{(u,v)\}, TE = TE \cup \{(u,v)\}$$

4)一直到 TE 中加满 $n-1$ 条边为止。



- 
- 图用邻接矩阵表示, `edge`(边的信息)
 - 图的顶点信息在顶点表 `Verticelist`中
 - 边的条数为`CurrentEdges`
 - 取最小的边以及判别是否构成回路,

取最小的边利用: 最小堆 (`MinHeap`)

克鲁斯卡尔算法

```
void Graph<string , float>::Kruskal(MinSpanTree&T)
```

```
{ MSTEdgeNode e;
```

```
  MinHeap<MSTEdgeNode>H(currentEdges);
```

```
  int NumVertices=VerticesList.Last , u , v ;
```

```
  Ufsets F(NumVertices);    //建立n个单元素的连通分量
```

```
  for(u=0;u<NumVertices;u++)
```

```
    for (v=u+1;v<NumVertices;v++)
```

```
      if(Edge[u][v]!=MAXINT)
```

```
        { e.tail=u;e.head=v;e.cost=Edge[u][v];
```

```
          H.insert(e);
```

```
        }
```

```
  int count=1; //生成树边计数
```

```
  while(count<NumVertices)
```

```
    { H.RemoveMin(e);
```

```
      u=F.Find(e.tail); v=F.Find(e.head);
```

```
      if(u!=v)
```

```
        {F.union(u,v);T.Insert(e);count++;}
```

```
    }
```

```
}
```


java(伪代码)

public void kruskal()

**{ int edgesAccepted; DisjSet s; priorityQueue h;
Vertex u, v ; SetType uset, vset; Edge e;**

h = readGraphIntoHeapArray();

h.buildHeap() ;

s = new DisjSet(NUM_VERTICES);

edgesAccepted = 0 ;

while(edgesAccepted < NUM_VERTICES – 1)

{ e = h.deleteMin() ; //Edge e = (u, v)

uset = s. find(u);

vset = s.find(v);

if(uset != vset)

{ edgesAccepted++;

s.union(uset, vset);

}

}

}

算法分析:

1) 建立 e 条边的最小堆。

检测邻接矩阵 $O(n^2)$

每插入一条边，执行一次

`fiterup()` 算法: $\log_2 e$

所以，总的建堆时间为 $O(e \log_2 e)$

2) 构造最小生成树时:

e 次出堆操作: 每一次出堆，执行一次`filterdown()`,
总时间为 $O(e \log_2 e)$

$2e$ 次`find`操作: $O(e \log_2 n)$

$n-1$ 次`union`操作: $O(n)$

所以，总的计算时间为 $O(e \log_2 e + e \log_2 n + n^2 + n)$

3.Prim算法

设：原图的顶点集合 V (有 n 个)

生成树的顶点集合 U (最后也有 n 个)，一开始为空

TE 集合为 $\{ \}$

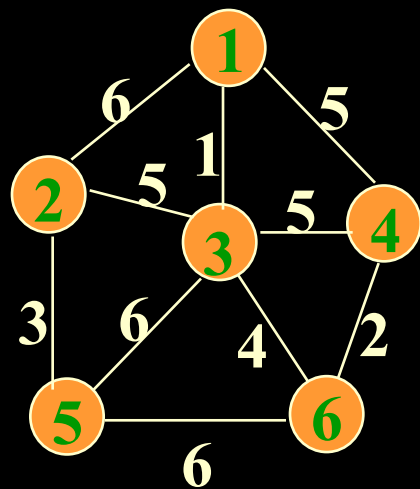
步骤：

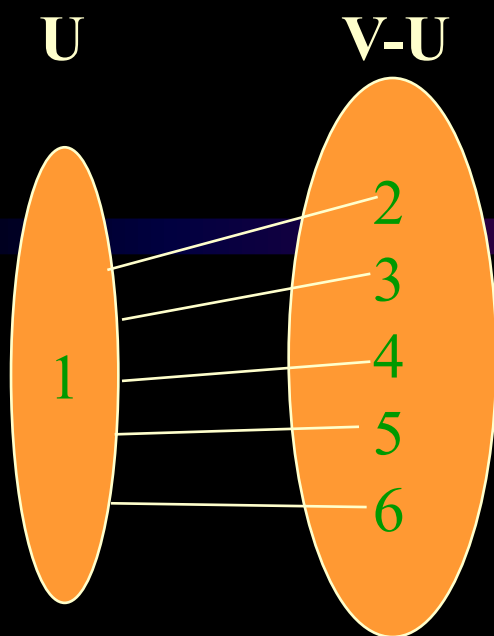
1) $U=\{1\}$ 任何起始顶点， $TE=\{ \}$

→ 2) 每次生成（选择）一条边。这条边是所有边 (u,v)
中代价（权）最小的边，
 $u \in U, v \in V-U$
 $TE=TE+[(u,v)]$;
 $U=U+[v]$

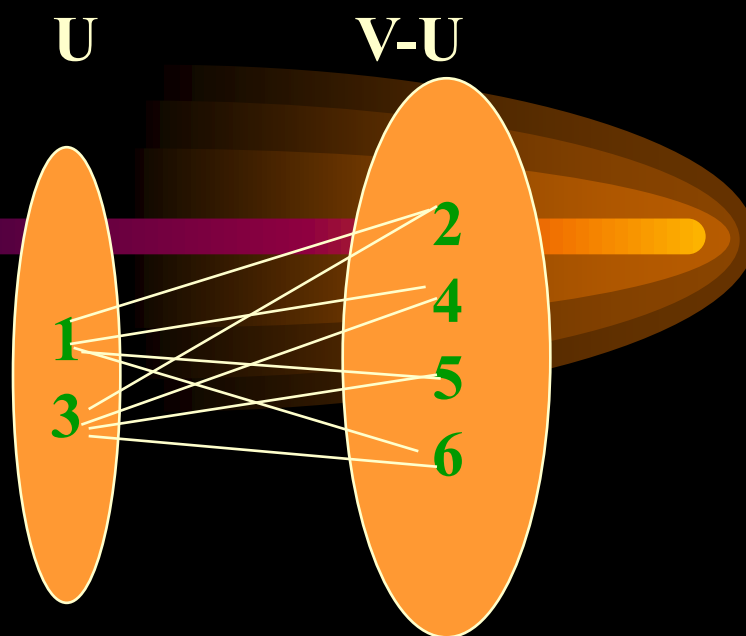
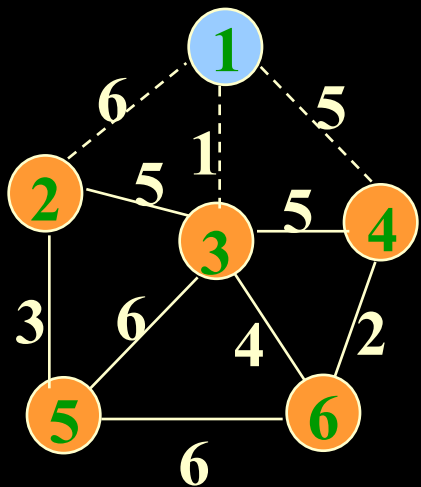
3) 当 $U \neq V$

用一开始的例子：

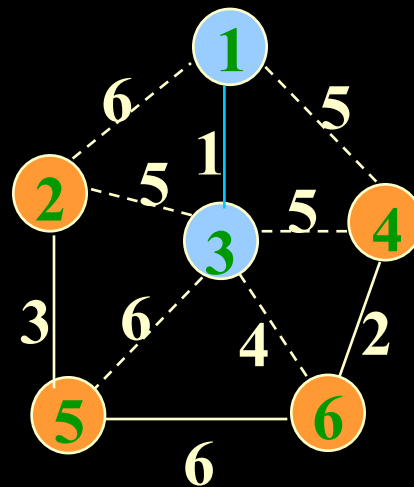


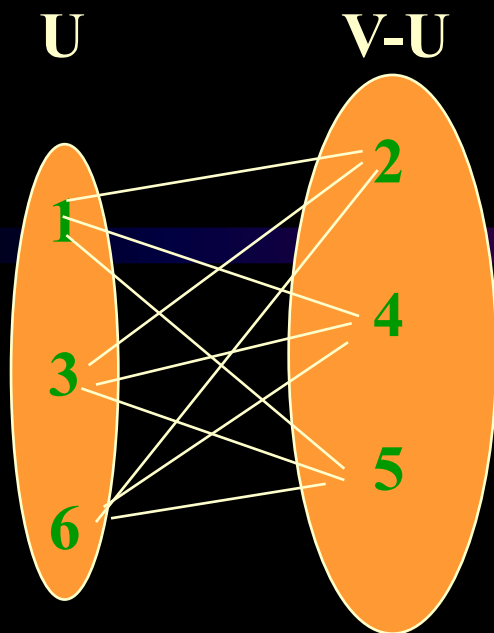


5条中找最小

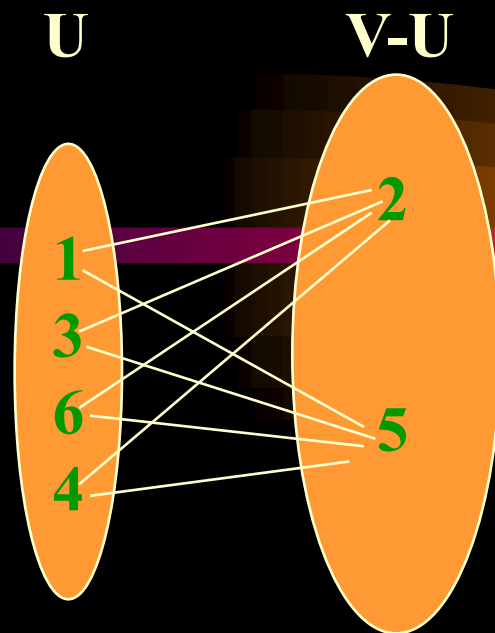
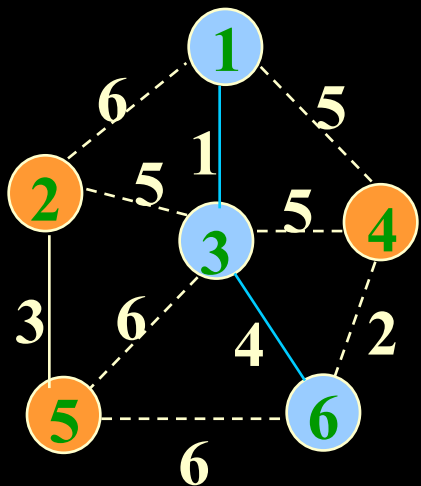


8条中找最小

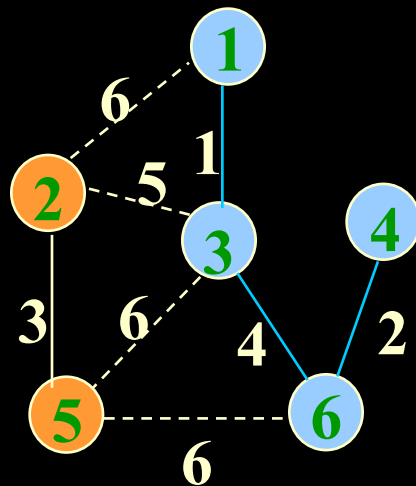


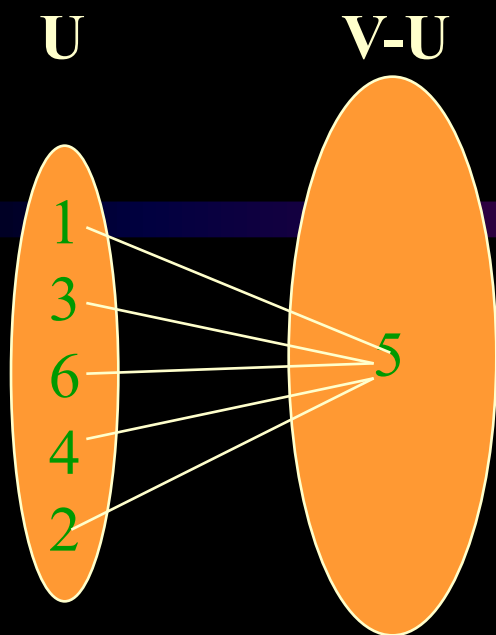


9条中找最小

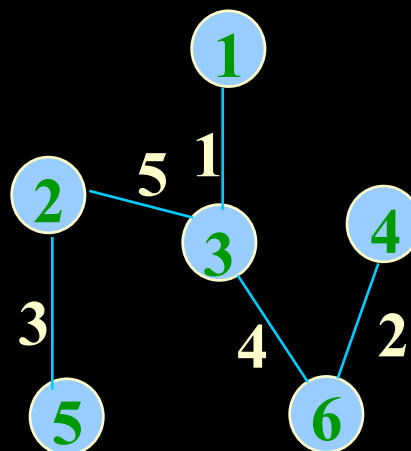
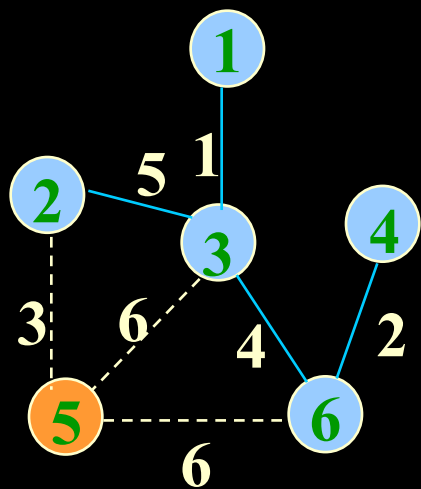


8条中找最小





5条中找最小



$\Sigma=15$

最小生成树不是唯一的. 为什么?

算法分析: $1*(n-1)+2*(n-2)+\dots+(n-1)*(n-(n-1))$

$$=n-1+2n-4+3n-9+\dots+(n-1)*n-(n-1)^2$$

$$=n+2n+3n+\dots+(n-1)*n-1^2-2^2-3^2-\dots-(n-1)^2$$

$$=n*(1+2+3+\dots+(n-1))-(1^2+2^2+3^2+\dots+(n-1)^2)$$

$$=n*n*(n-1)/2-n*(n-1)*(2n-1)/6$$

$$=n*(n-1)*(n+1)/6$$

$$=O(n^3)$$

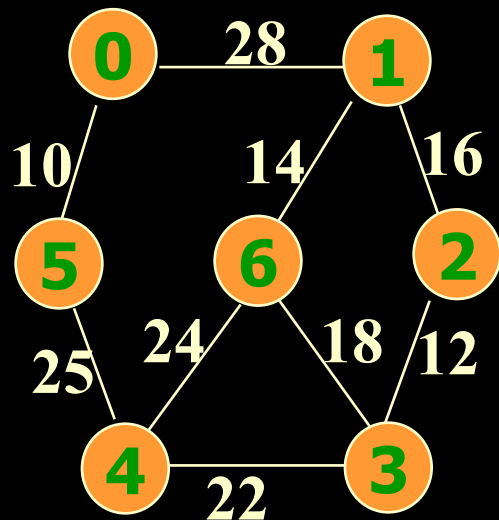
具体实现:

- 1) 图采用邻接矩阵
- 2) 改进了实现效率 ($O(n^2)$).

Lowcost[]:存放生成树顶点集合内顶点到生成树外各顶点的边上的当前最小权值;

nearvex[]:记录生成树顶点集合外各顶点, 距离集合内那个顶点最近。

从顶点0出发



初始状态

Lowcost:

0	1	2	3	4	5	6
0	28	∞	∞	∞	10	∞

nearvex:

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

↑
如果为-1则表示已加入生成树顶点集合

反复做以下工作：

- 1) 在Lowcost[]中选择nearvex[i]≠-1,且lowcost[i] 最小的边用v标记它。，则选中的权值最小的边为 (nearvex[v],v), 相应的权值为lowcost[v]。

例如在上面图中第一次选中的v=5;则边 (0, 5) , 是选中的权值最小的边, 相应的权值为lowcost[5]=10。

- 2) 将nearvex[v] 改为-1, 表示它已加入生成树顶点集合。
将边(nearvex[v],v,lowcost[v])加入生成树的边集合。

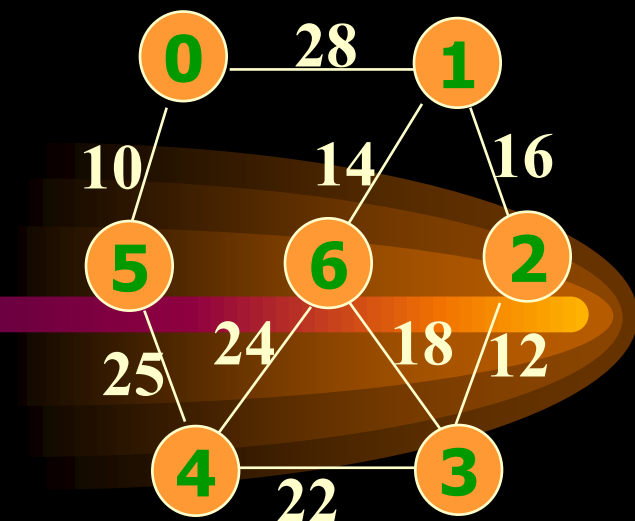
- 3) 修改。

取lowcost[i]=min{lowcost[i],Edge[v][i]},即用生成树顶点集合外各顶点i到刚加入该集合的新顶点 v的距离(Edge[v][i])与原来它所到生成树顶点集合中顶点的最短距离lowcost[i]做比较, 取距离近的, 作为这些集合外顶点到生成树顶点集合内顶点的最短距离。

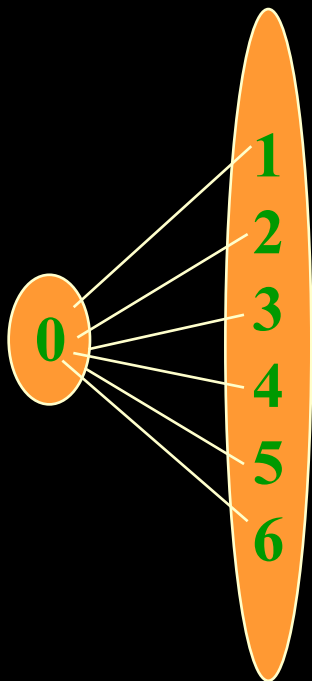
如果生成树顶点集合外的顶点 i到刚加入该集合新顶点v的距离比原来它到生成树顶点集合中顶点的最短距离还要近, 则修改nearvex[i]:

nearvex[i]=v

表示生成树外顶点i到生成树的内顶点v 当前距离最短。



U V-U



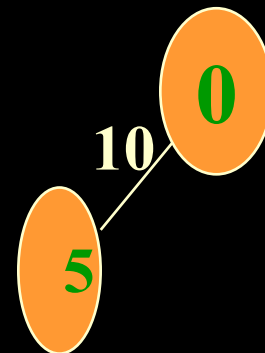
nearvex

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

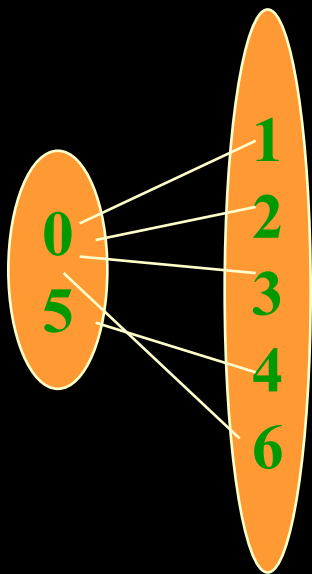
lowcost

0	0
1	28
2	∞
3	∞
4	∞
5	10
6	∞

(0,5,10)

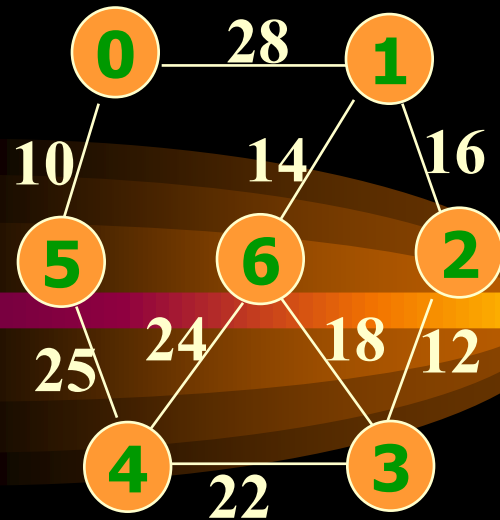
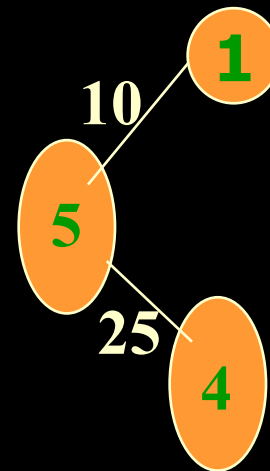


U V-U

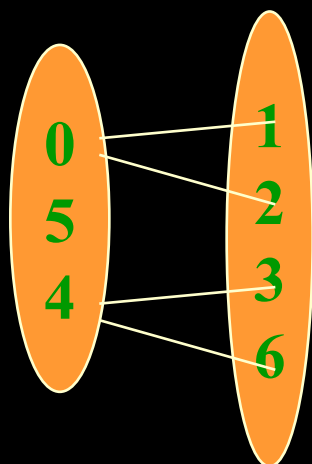


	nearvex	lowcost
0	-1	0
1	0	28
2	0	∞
3	0	∞
4	5	25
5	-1	10
6	0	∞

(5,4,25)



U V-U



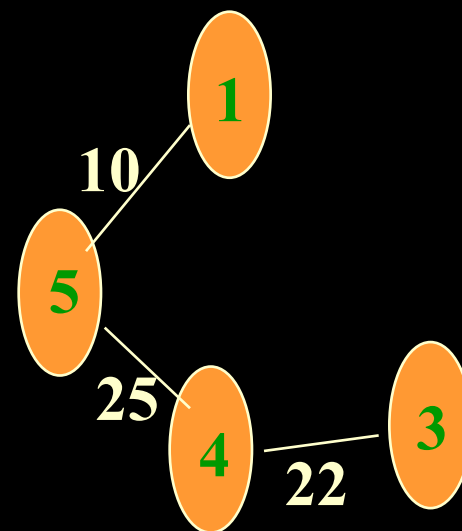
nearvex

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

lowcost

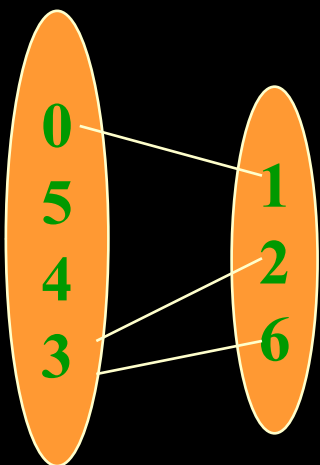
0	0
1	28
2	∞
3	22
4	25
5	10
6	24

(4,3,22)



U

V-U



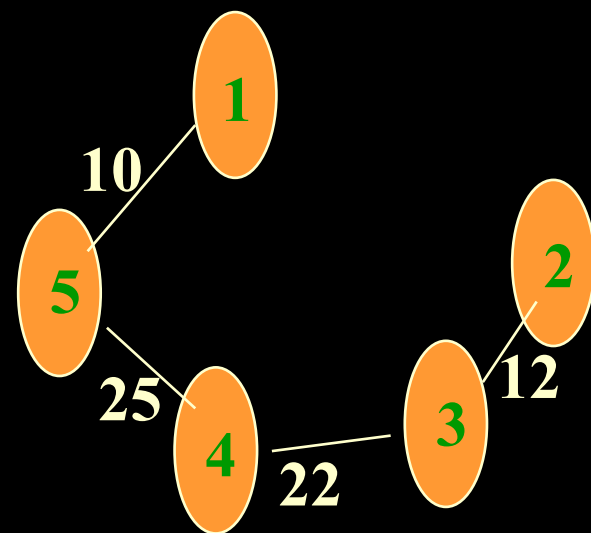
nearvex

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

lowcost

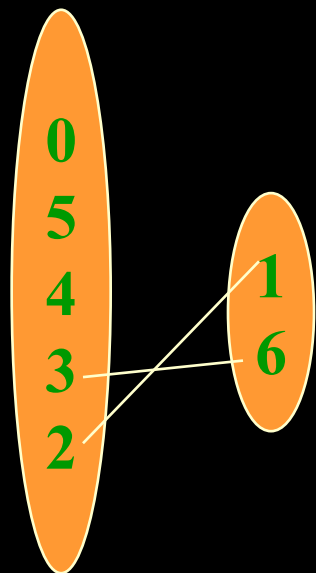
0	0
1	28
2	12
3	22
4	25
5	10
6	18

(3,2,12)



U

V-U



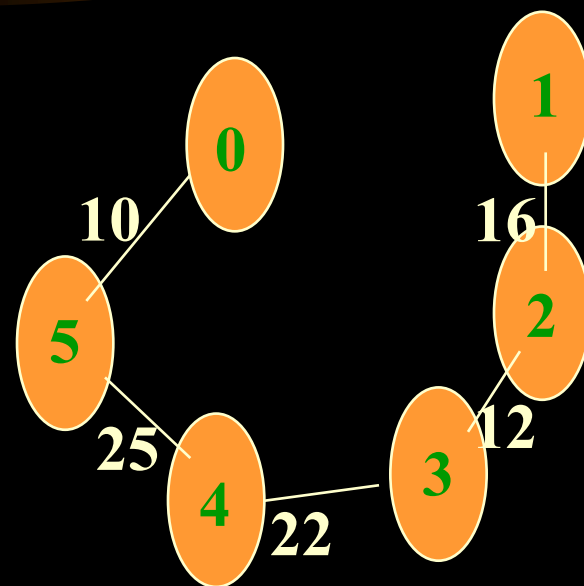
nearvex

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

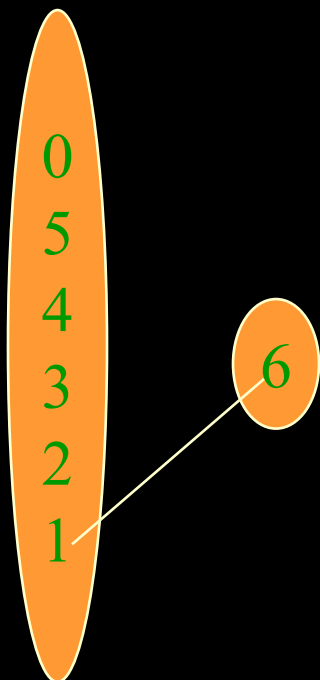
lowcost

0	0
1	16
2	12
3	22
4	25
5	10
6	18

(2,1,16)

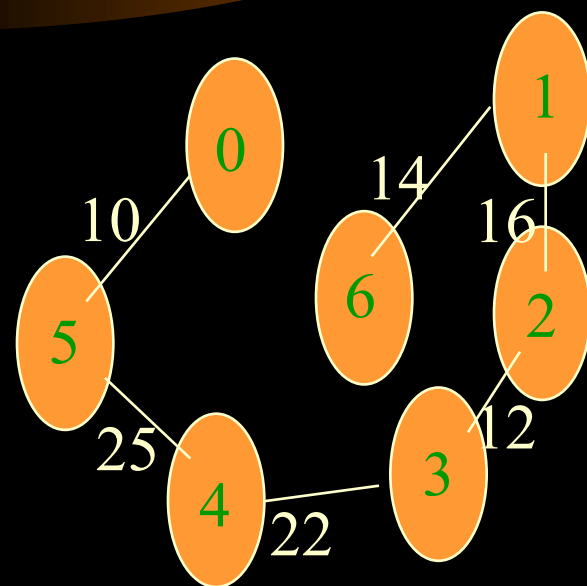


U V-U



	nearvex	lowcost
0	-1	0
1	-1	16
2	-1	12
3	-1	22
4	-1	25
5	-1	10
6	1	14

(1,6,14)



Prim（普里姆）算法

```
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;
```

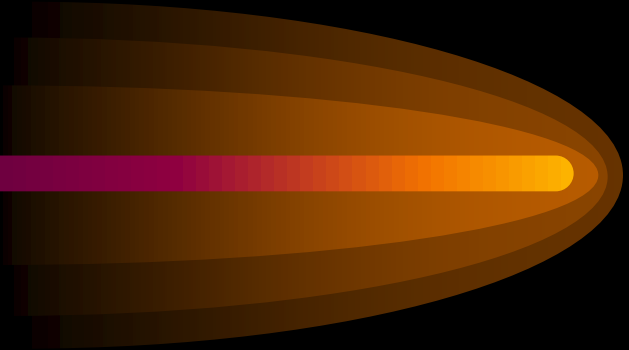
```
  MSTEdgeNode e;
```

```
  for (int i=1; i< NumVertices; i++)  
  { 1. float min=MAXINT; int v=0;  
    2. for ( int j=1; j< NumVertices; j++)  
      if(nearvex[j]!=-1&&lowcost[j]<min)  
        {v=j; min=lowcost[j];} //for j, 选择最小的边
```

```
3. if (v)
    { e.tail=nearvex[v];
      e.head=v;
      e.cost=lowcost[v];

      T.Insert(e);
      nearvex[v]=-1;

      for(int j=1; j< NumVertices; j++)
          if( nearvex[j]!=-1 && Edge[v][j]<lowcost[j] )
              { lowcost[j]=Edge[v][j]; nearvex[j]=v;}
          } //if
      } //for i
    }
```



算法结构为：

[n
[n
[求最小的 n
[修改 n
[

时间复杂度： $O(n^2)$

9.6 最短路径 (shortest path)

设 $G=(V,E)$ 是一个带权图（有向，无向），如果从顶点 v 到顶点 w 的一条路径为 (v, v_1, v_2, \dots, w) ，其路径长度不大于从 v 到 w 的所有其它路径的长度，则该路径为从 v 到 w 的最短路径。

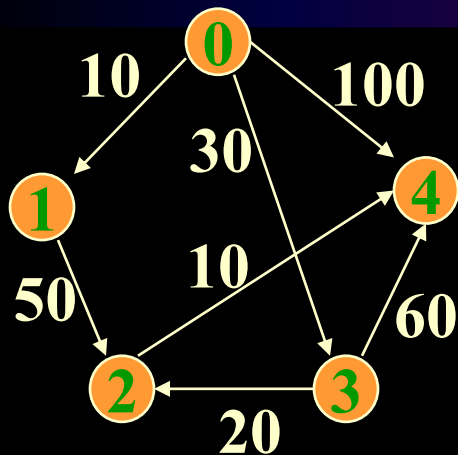
背景：交通网络中，求各城镇间的最短路径。

三种算法：

- 1) 边上权值为非负情况的从一个结点到其它各结点的最短路径（单源最短路径）（Dijkstra算法）
- 2) 边上权值为任意值的单源最短路径
- 3) 边上权值为非负情况的所有顶点之间的最短路径

1.含非负权值的单源最短路径 (Dijkstra)

• 问题



V_0		V_1	10	
V_0	V_3	V_2	50	
V_0		V_3	30	
V_0	V_3	V_2	V_4	60

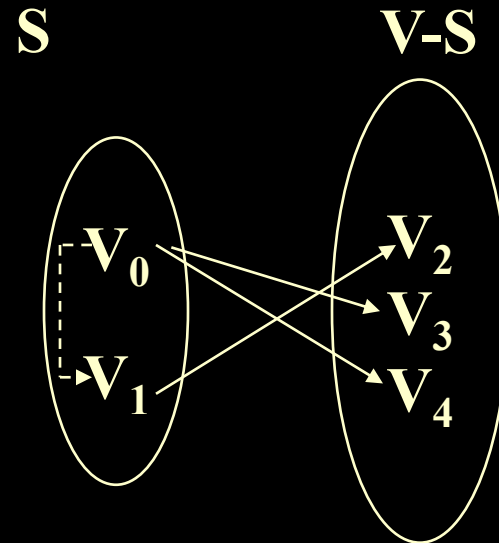
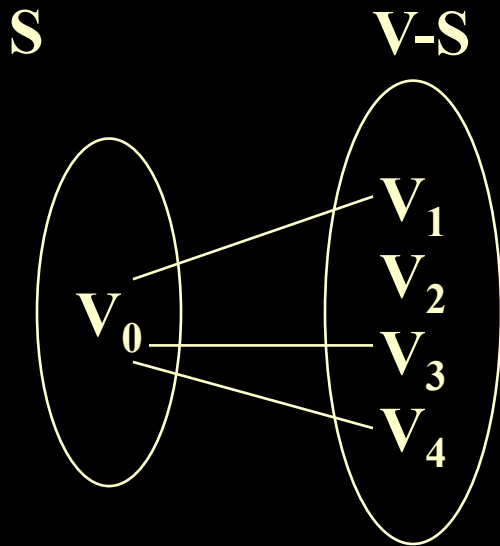
如果按距离递增的顺序重新排列一下

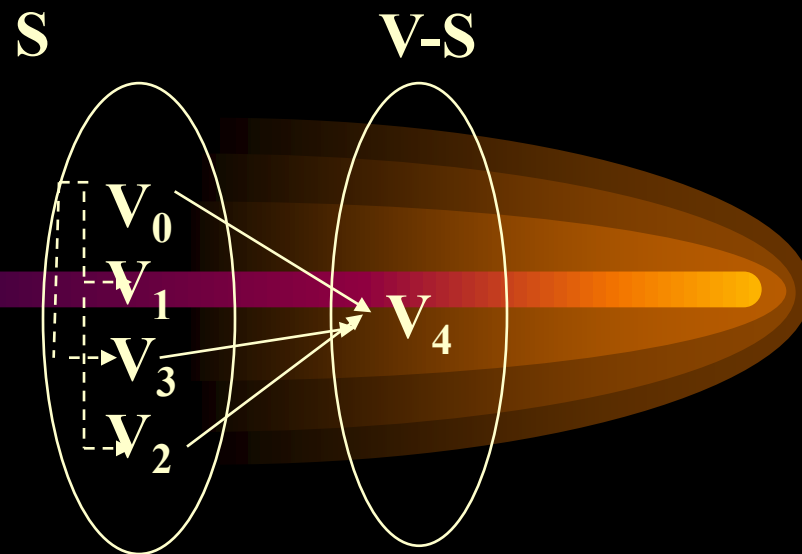
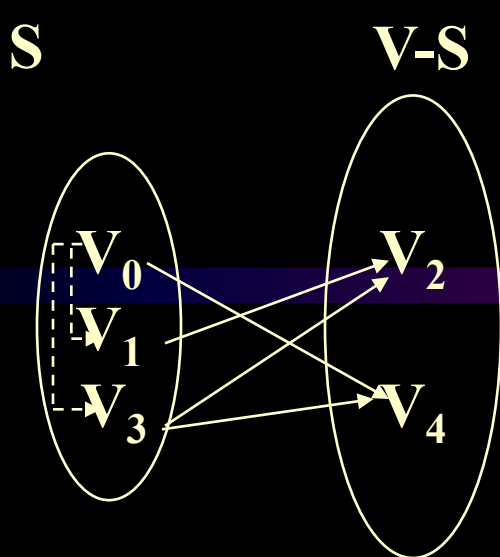
	经过	终止	距离	
V_0		V_1	10	
V_0		V_3	30	
V_0	V_3	V_2	50	
V_0	V_3	V_2	V_4	60

	0	1	2	3	4
0	0	10	∞	30	100
1	∞	0	50	∞	∞
2	∞	∞	0	∞	10
3	∞	∞	20	0	60
4	∞	∞	∞	∞	0

- 思想：按最短路径长度递增的次序产生最短路径。
一开始，在源点到直接有连线的诸顶点的 path 中找最小的，去掉该点，然后找从源点到余下点中最短的 path (这里可以不是直接连线，可以是经过前面已找到的最短 path 的顶点)

算法思想：





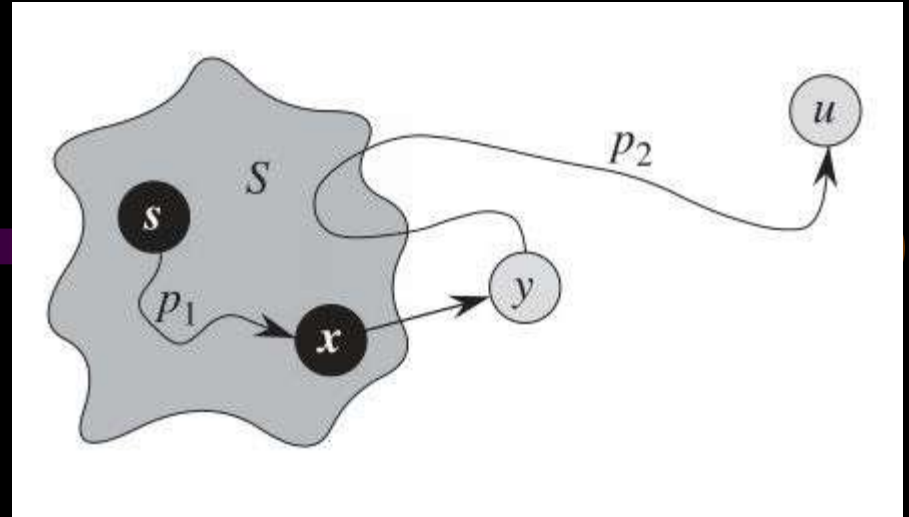
距离值数组:dist

0	0			
1	10 0-1			
2	∞ 0-2	60 0-1-2	50 0-3-2	
3	30 0-3	30 0-3		
4	100 0-4	100 0-4	90 0-3-4	60 0-3-2-4

路径path

0				
1	0	0	0	0
2	-1	1	3	3
3	0	0	0	0
4	0	0	3	2

每次放由 v_0 到达该顶点的前一顶点



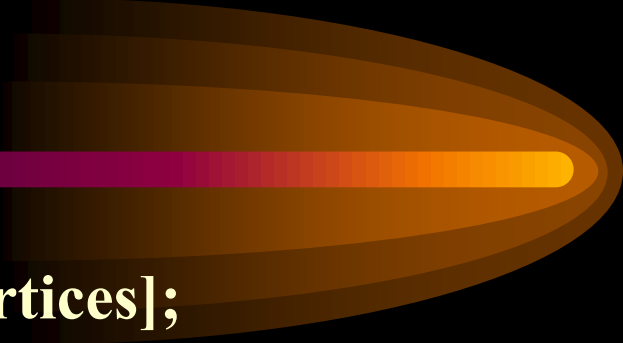
正确性证明，待证问题：

当把顶点 u 加入到集合 S 的时候，辅助距离数组 $\text{dist}[u]$ 的值必然是源点 s 到 u 的最短路径长度。

证明方法：数学归纳法

Dijkstra 算法

```
const int NumVertices=6;  
class graph  
{ private:  
    int Edge[NumVertices][NumVertices];  
    int dist[NumVertices];  
    int path[NumVertices];  
    int S[NumVertices];  
public:  
    void shortestpath(int,int);  
};
```



```

void Graph::shortestpath(int n,int v)
{   for( int i=0; i<n; i++)
    { 1. dist[i]=Edge[v][i]; s[i]=0;
      2. if( i!=v && dist[i]< MAXNUM ) path[i]=v;
        else path[i]=-1;
    }

    s[v]=1; dist[v]=0;
    for( i=0; i<n-1; i++)
    { 1. float min=MAXNUM; int u=v;
      2. for( int j=0; j<n; j++)
          if( !s[j] && dist[j]<min ) { u=j; min=dist[j];}

      3. s[u]=1;
      4. for ( int w=0; w<n; w++)
          if( !s[w] && Edge[u][w] < MAXNUM &&
              dist[u]+Edge[u][w] < dist[w])
              { dist[w]=dist[u]+Edge[u][w]; path[w]=u;}
    }//for
}

```

算法分析:

[n

[n

[求最小的 n

[修改 n

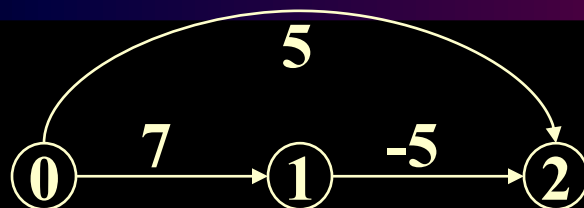
$O(n^2)$

2.边上权值为任意值的单源最短路径（贝尔曼—福特）

dijkstra算法在边上权值为任意值的图上是不能正常工作的。

为什么？问题出在程序的哪里？

例子：



$v_0 \rightarrow v_2$ 5

$v_0 \rightarrow v_1$ 7

贝尔曼—福特提出了一个改进的算法：

构造一个最短路径长度数组

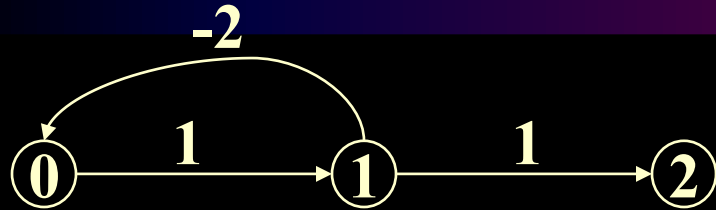
$\text{dist}^1[u], \text{dist}^2[u], \dots, \text{dist}^{n-1}[u]$

其中： $\text{dist}^1[u]$ ：是从源点 v 到终点 u 的只经过一条边的
的最短路径长度，

$\text{dist}^1[u] = \text{Edge}[v][u]$

$\text{dist}^2[u]$ ：是从源点 v 最多经过两条边到达终点 u
的最短路径长度；

$\text{dist}^3[u]$: 是从源点 v 最多经过不构成带负长度边回路的三条边的最短路径长度;



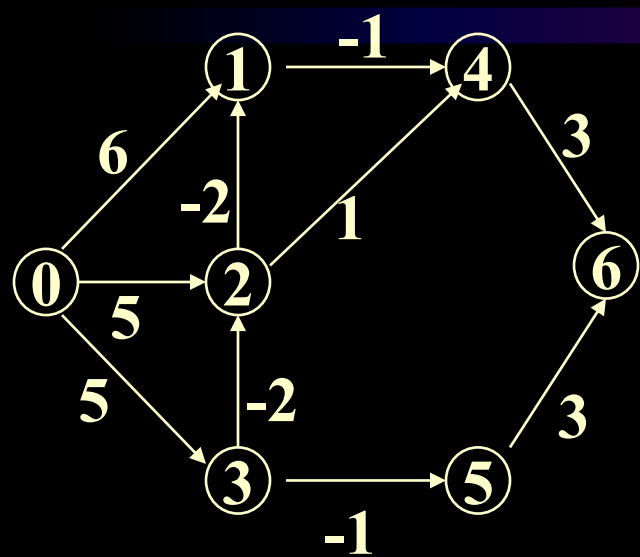
不允许带负权值的边组成回路

...

$\text{dist}^{n-1}[u]$: 是从源点 v 最多经过不构成带负长度边回路的 $n-1$ 条边的最短路径长度;

递推公式: $\text{dist}^1[u] = \text{Edge}[v][u];$
 $\text{dist}^k[u] = \min\{\text{dist}^{k-1}[u], \min_{j=0,1,2,\dots,n-1}\{\text{dist}^{k-1}[j] + \text{Edge}[j][u]\}\}$

例子：



	1	2	3	4	5	6
$\text{dist}^k[0]$	0	0	0	0		
$\text{dist}^k[1]$	6 0-1	3 0-2-1	1 0-3-2-1	1		
$\text{dist}^k[2]$	5 0-2	3 0-3-2	3	3		
$\text{dist}^k[3]$	5 0-3	5 0-3	5	5		
$\text{dist}^k[4]$	∞ 0-4	5 0-1-4	2 0-2-1-4	0 0-3-2-1-4		
$\text{dist}^k[5]$	∞ 0-5	4 0-3-5	4	4		
$\text{dist}^k[6]$	∞ 0-6	∞ 0-6	7 0-3-5-6	5 0-2-1-4-6		

```
void Graph::BellmanFord(int n, int v)
{ for(int i=0;i<n;i++)
    { dist[i]=Edge[v][i];
      if(i!=v&&dist[i]<MAXNUM)path[i]=v;
      else path[i]=-1;
    }

    for (int k=2;k<n;k++)
        for(int u=0;u<n;u++)
            if(u!=v)
                for(i=0;i<n;i++)
                    if (Edge[i][u]<>0 && Edge[i][u]<MAXNUM &&
                        dist[u]>dist[i]+Edge[i][u])
                        {dist[u]=dist[i]+Edge[i][u];path[u]=i;}
                }
}
```

算法分析: $O(n^3)$

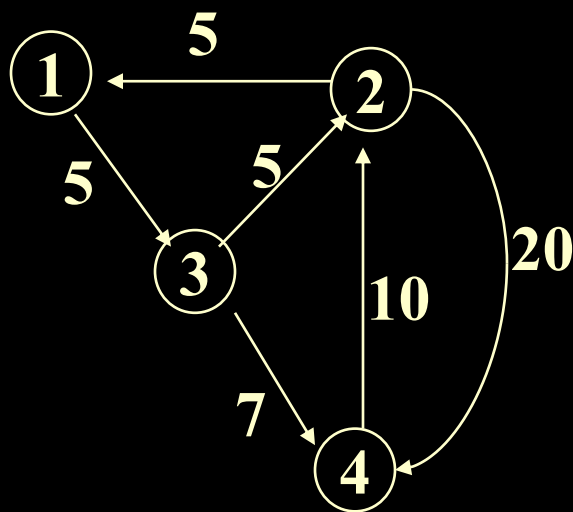
3.所有顶点之间的最短路径 (floyed)

前提：各边权值均大于0的带权有向图。

方法：1)把有向图的每一个顶点作为源点，重复执行Dijkstra算法n次，
执行时间为 $O(n^3)$

2)Floyed方法，算法形式更简单些，但是时间仍然是 $O(n^3)$

例子：



$$A = \begin{bmatrix} 0 & \infty & 5 & \infty \\ 5 & 0 & \infty & 20 \\ \infty & 5 & 0 & 7 \\ \infty & 10 & \infty & 0 \end{bmatrix}$$

floyed算法：在矩阵A上作n-1次迭代，设每次迭代结果分别为

$$A^{(0)}, A^{(1)}, A^{(2)}, \dots, A^{(n)}$$

$A^{(0)}$ =源矩阵，认为 $v_i \rightarrow v_j$ 的直接弧为它们的min路径

$$A^{(1)} = A^{(1)}[i,j] = \min(A^{(0)}[i,j], A^{(0)}[i,1] + A^{(0)}[1,j])$$

此时 $A^{(1)}[i,j]$ 可能已换成 $v_i - v_1 - v_j$

$$A^{(2)} = A^{(2)}[i,j] = \min(A^{(1)}[i,j], A^{(1)}[i,2] + A^{(1)}[2,j])$$

即考虑经过顶点2，它可能是 $v_i - v_j$, $v_i - v_1 - v_j$, $v_i - v_1 - v_2 - v_j$,
 $v_i - v_2 - v_1 - v_j$, $v_i - v_2 - v_j$ 的min者

⋮

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

⋮

如上例

$$A^{(0)} = \begin{bmatrix} 0 & \infty & 5 & \infty \\ 5 & 0 & \infty & 20 \\ \infty & 5 & 0 & 7 \\ \infty & 10 & \infty & 0 \end{bmatrix}$$

$$A^{(1)} = \begin{bmatrix} 0 & \infty & 5 & \infty \\ 5 & 0 & 10 & 20 \\ \infty & 5 & 0 & 7 \\ \infty & 10 & \infty & 0 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 0 & \infty & 5 & \infty \\ 5 & 0 & 10 & 20 \\ 10 & 5 & 0 & 7 \\ 15 & 10 & 20 & 0 \end{bmatrix}$$

$$A^{(3)} = \begin{bmatrix} 0 & 10 & 5 & 12 \\ 5 & 0 & 10 & 17 \\ 10 & 5 & 0 & 7 \\ 15 & 10 & 20 & 0 \end{bmatrix}$$

$$A^{(4)} = \begin{bmatrix} 0 & 10 & 5 & 12 \\ 5 & 0 & 10 & 17 \\ 10 & 5 & 0 & 7 \\ 15 & 10 & 20 & 0 \end{bmatrix}$$

$$\text{path}^{(4)} = \begin{bmatrix} 0 & 3 & 1 & 3 \\ 2 & 0 & 1 & 3 \\ 2 & 3 & 0 & 3 \\ 2 & 4 & 1 & 0 \end{bmatrix}$$

具体算法:

```
void Graph::Alllength(int n)
```

```
{ 1. for(int i=0; i<n; i++)  
    for(int j=0; j<n; j++)  
    { a[i][j]=Edge[i][j];  
      if(i<>j&& a[i][j]<MAXNUM) path[i][j]=i;  
      else path[i][j]=0;  
    }  
2. for(int k=0; k<n; k++)  
    for(int i=0; i<n; i++)  
        for(int 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];  
                }  
}
```

三重循环 $O(n^3)$

9.7 活动网络 (Activity Network)

本节介绍两个算法

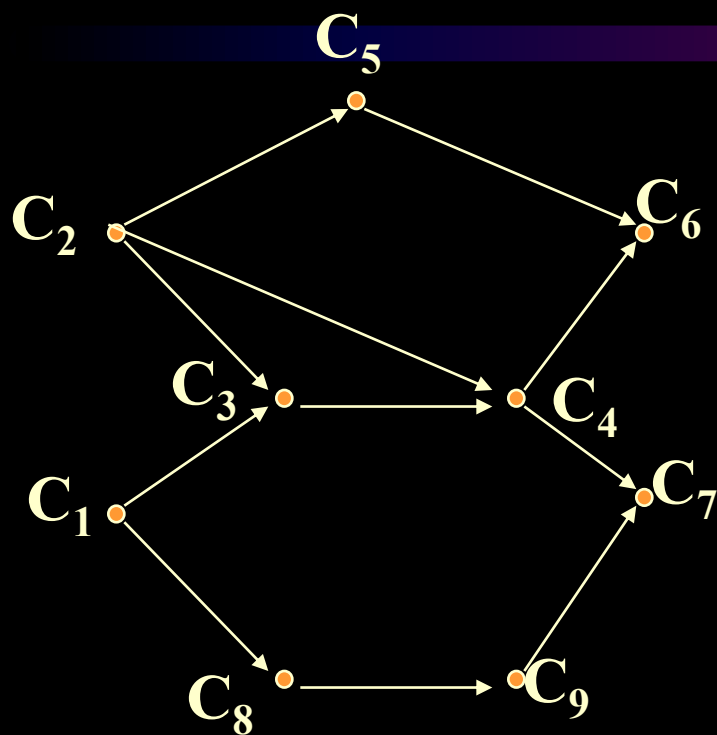
- 用顶点表示活动的网络 (AOV网络)
- 用边表示活动的网络 (AOE网络)

1.用顶点表示活动的网络
(拓扑排序—topological sort)

- 例子：假设计算机系的开课情况为：

课程代号	课程名称	先修课程
C ₁	高等数学	
C ₂	程序设计基础	
C ₃	离散数学	C ₁ , C ₂
C ₄	数据结构	C ₂ , C ₃
C ₅	高级语言程序设计	C ₂
C ₆	编译方法	C ₄ , C ₅
C ₇	操作系统	C ₄ , C ₉
C ₈	普通物理	C ₁
C ₉	计算机原理	C ₈

这说明要学 C_3 必须要学 C_1, C_2, \dots , 所以, 整个课程间的优先关系可以用一个有向图来表示。



一般来讲集合上的偏序关系用符号
“ \prec ”表示。用无环有向图来实现
先于

图中顶点表示课程（活动），有向边（弧）表示先决条件。

若课程*i*是课程*j*的预修课程，则图中有弧*<i,j>*

- **AOV网 (Activity On Vertex network)**

用顶点表示活动，用弧表示活动间的优先关系的有向图称为AOV网。

直接前驱，直接后继：*<i,j>*是网中一条弧，则*i*是*j*的直接前驱，*j*是*i*的直接后继。

前驱，后继：从顶点*i*→顶点*j*有一条有向路径，则称*i*是*j*的前驱，*j*是*i*的后继。

***AOV网中，不应该出现有向环**

- 拓扑排序 (topological sort)

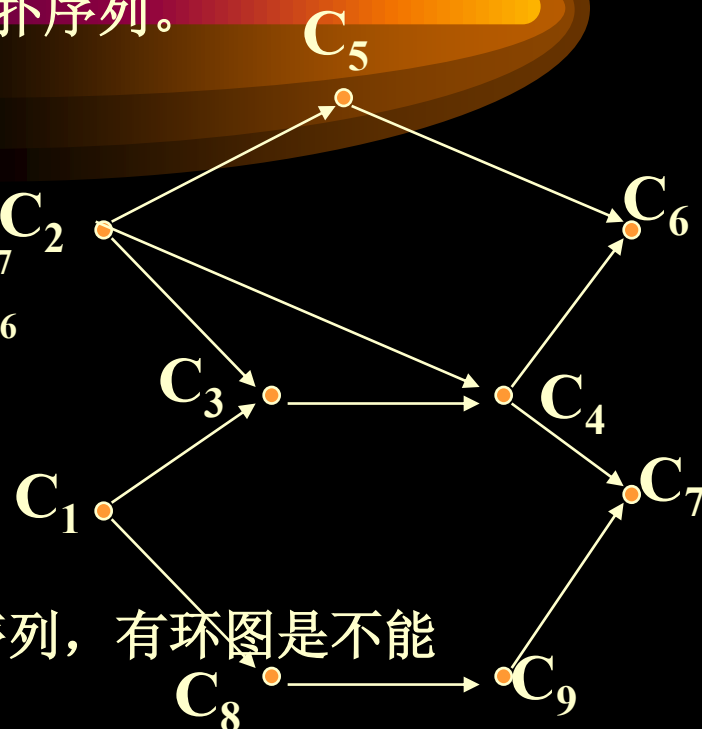
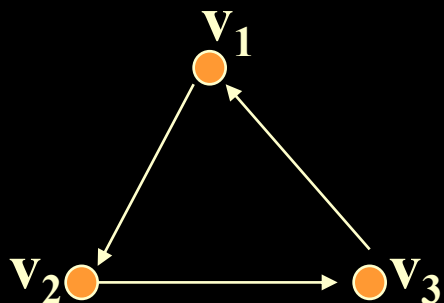
有向图 $G=(V,E)$, V 里结点的线性序列 $(v_{i1}, v_{i2}, \dots, v_{in})$,
如果满足: 在 G 中从结点 v_i 到 v_j 有一条路径, 则序列中结点
 v_i 必先于结点 v_j , 称这样的线性序列为一拓扑序列。

如上例, 学生选课有

$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$
或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$

拓扑序列不是唯一的

不是任何有向图的结点都可以排成拓扑序列, 有环图是不能排的。



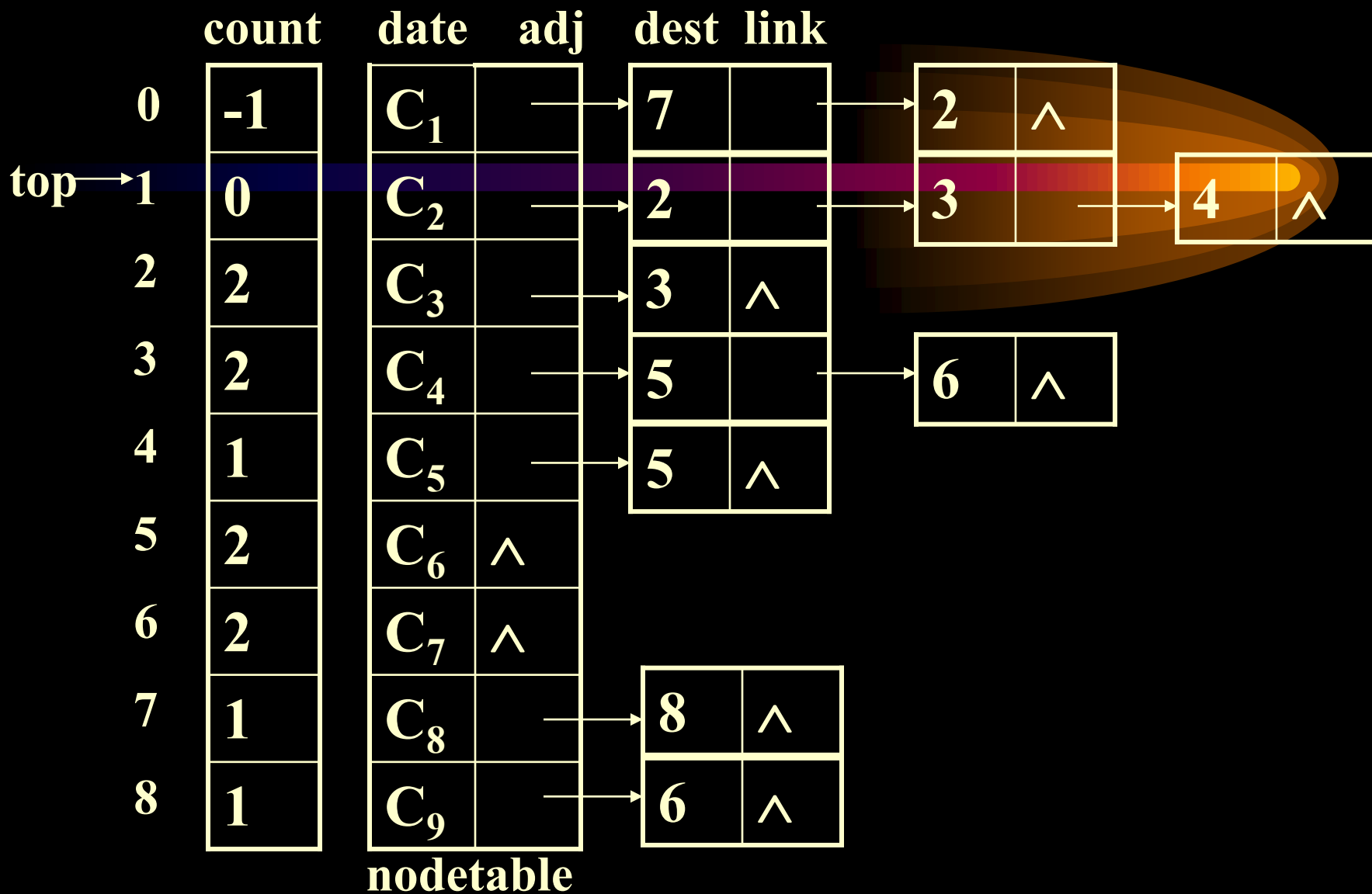
算法思想：

- 1) 从图中选择一个入度为0的结点输出之。
(如果一个图中，同时存在多个入度为0的结点，则随便输出那一个结点)
- 2) 从图中删掉此结点及其所有的出边。
- 3) 反复执行以上步骤：
 - a) 直到所有结点都输出了，则算法结束
 - b) 如果图中还有结点，但入度不为0，则说明有环路

具体实现算法：

AOV网用邻接表来实现
数组count存放各顶点的入度

并且为了避免每次从头到尾查找入度为0的顶点，建立入度为0的顶点栈，栈顶指针为top，初始化时为-1.



AOV网的声明

```
class Graph  
{ 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 ( ) ;  
};
```

void Graph :: Topologicalsort ()

{ 1. int top=-1;

2. for (int i=0; i<n ;i++)

if (count[i]==0) {count[i]=top ; top =i;}

3. for (int i=0 ; i<n ; i++)

if (top== -1){cout <<“Network has a cycle”<< endl; return;}

else

{ 1) int j=top; top=count[top];

2) cout<<j<<endl;

3)Edge<float> *l = NodeTable[j].adj;

4) while(l)

{ int k = l.dest;

if (--connt[k] == 0){ count[k] = top; top = k; }

l = l->link;

}

}

}

java

void topsort() throws CycleFound

{ Queue q;

int counter = 0;

Vertex v, w;

q = new Queue();

for each vertex v

if(v.indegree == 0)

q.enqueue(v);

while(!q.isEmpty())

{ v = q.dequeue();

v.topNum = ++counter; //Assign next number

for each w adjacent to v

if(--w.indegree == 0) q.enqueue;

}

if(counter != NUM_VERTICES)

throw new CycleFound();

}

算法分析: n 个顶点, e 条边

建立链式栈 $O(n)$

每个结点输出一遍, 每条边被检查一遍 $O(n+e)$

所以为: $O(n+n+e)$

2.用边表示活动的网络（AOE网络, Activity On Edge Network）

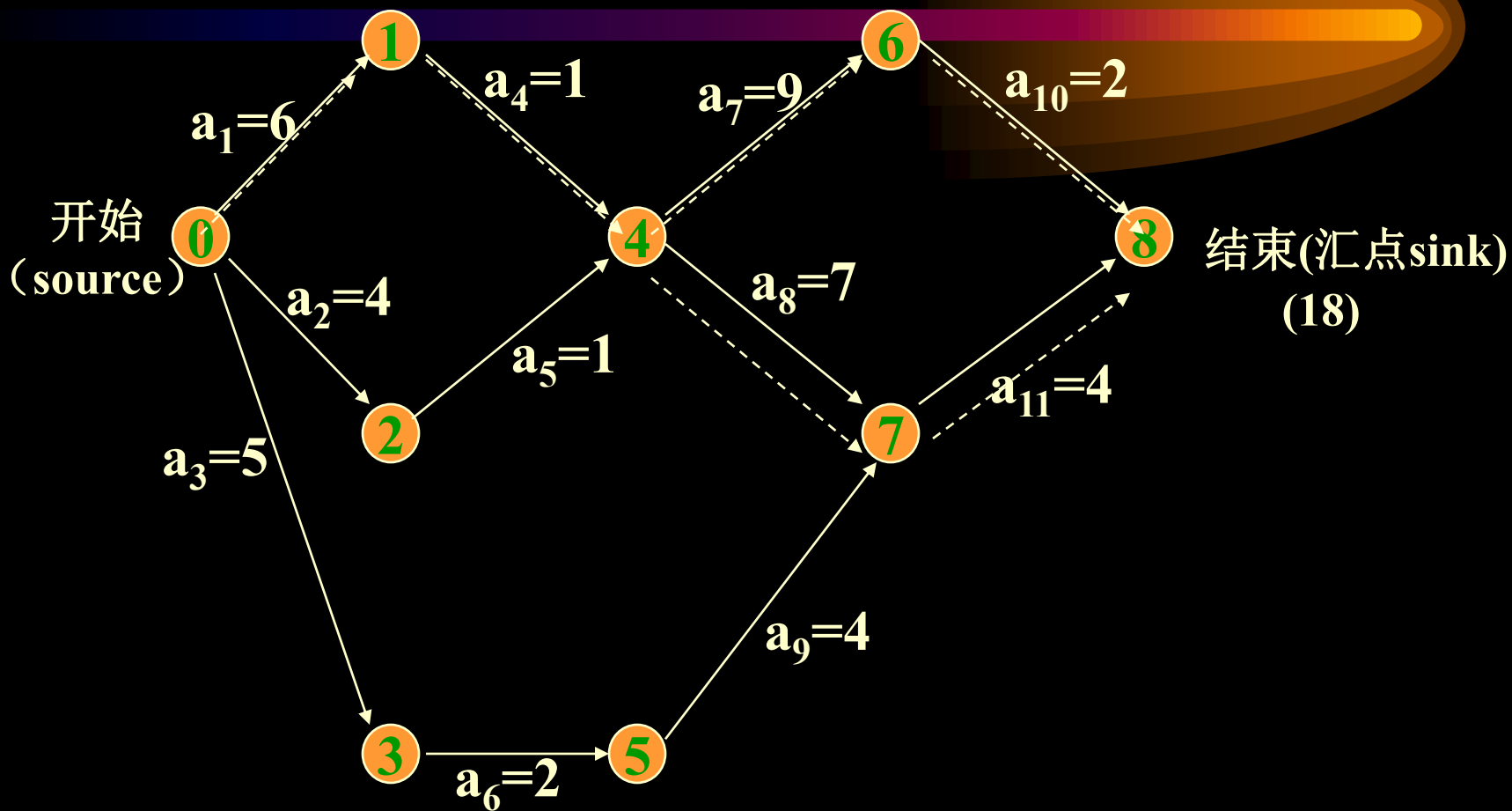
又称为事件顶点网络

- 顶点：表示事件（event）

事件——状态。表示它的入边代表的活动已完成，它的出边代表的活动可以开始，如下图 v_0 表示整个工程开始， v_4 表示 a_4 ， a_5 活动已完成 a_7 ， a_8 活动可开始。

有向边：表示活动。

边上的权——表示完成一项活动需要的时间



图中有11项活动: a_1, a_2, \dots, a_{11} ;

9个事件或者称9个状态: v_0, v_1, \dots, v_8 ,

v_0 表示整个工程开始,

v_8 表示整个工程结束,

边上的权表示活动完成所需的天数(这些天数仅仅是估计值),
假设图为无环有向图。

有唯一的入度为0的开始结点
唯一的出度为0的完成结点 } 与AOV网不同

- 关键路径 (critical path)

1)目的: 利用事件顶点网络, 研究完成整个工程需要多少时间
加快那些活动的速度后, 可使整个工程提前完成。

2)关键路径: 具有从开始顶点(源点) \rightarrow 完成顶点(汇点)的
最长的路径

3) 找关键活动的算法:

(1) 定义几个量

对事件而言

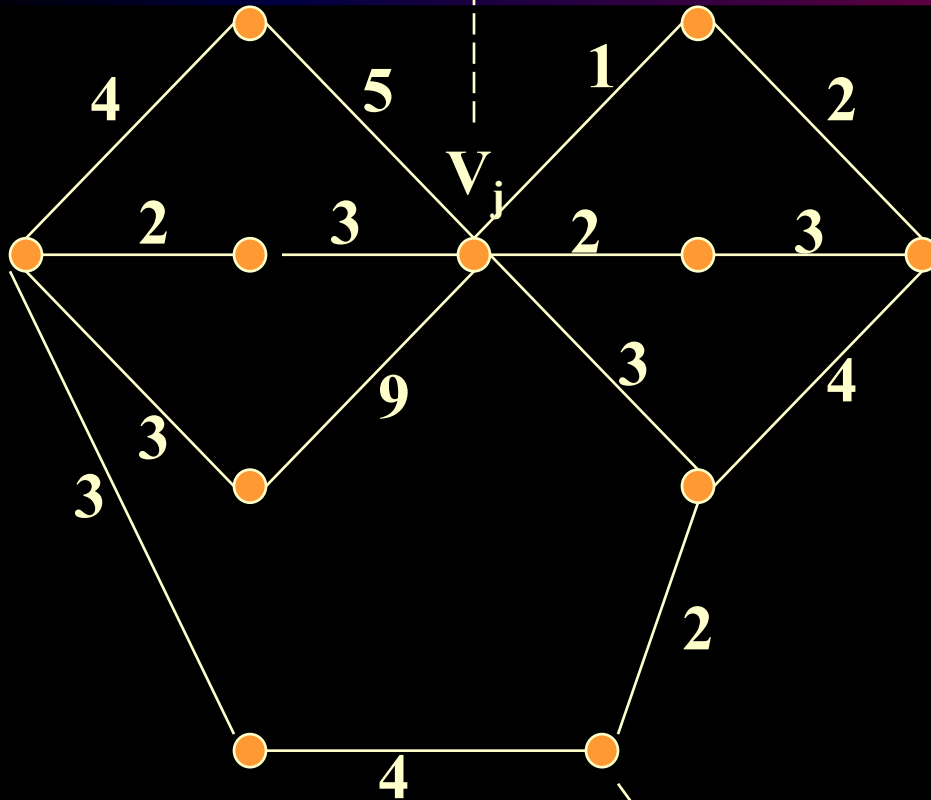
$Ve[i]$ —表示事件 V_i 的可能最早发生时间

定义为从源点 $V_0 \rightarrow V_i$ 的最长路径长度, 如 $Ve[4]=7$ 天

$Vl[i]$ —表示事件 V_i 的允许的最晚发生时间。

是在保证汇点 V_{n-1} 在 $Ve[n-1]$ 时刻(18)完成的前提下,
事件 V_i 允许发生的最晚时间 = $Ve[n-1] - V_i \rightarrow V_{n-1}$ 的最长路径长度。

(19)



最早发生时间是7
最迟发生时间是13

对活动而言：

$e[k]$ —表示活动 $a_k = \langle V_i, V_j \rangle$ 的可能的最早开始时间。
即等于事件 V_i 的可能最早发生时间。

$$e[k] = Ve[i]$$

$l[k]$ —表示活动 $a_k = \langle V_i, V_j \rangle$ 的允许的最迟开始时间
 $l[k] = Vl[j] - \text{dur}(\langle i, j \rangle)$;

$l[k] - e[k]$ —表示活动 a_k 的最早可能开始时间和最迟
允许开始时间的时间余量。也称为松弛时间。
(slack time)

$l[k] = e[k]$ —表示活动 a_k 是没有时间余量的关键活动

一开始的例子中： a_8 的最早可能开始时间 $e[8] = Ve[4] = 7$
最迟允许开始时间 $l[8] = Vl[7] - \text{dur}(\langle 4, 7 \rangle)$
 $= 14 - 7 = 7$

$\therefore a_8$ 是关键路径上的关键活动

a_9 的最早可能开始时间 $e[9] = Ve[5] = 7$

最迟允许开始时间 $l[9] = Vl[7] - \text{dur}(<5, 7>) = 14 - 4 = 10$

$\therefore l[9] - e[9] = 3$, 该活动的时间余量为3, 即推迟3天或延迟3天完成都不影响整个工程的完成, 它不是关键活动

(2)步骤:

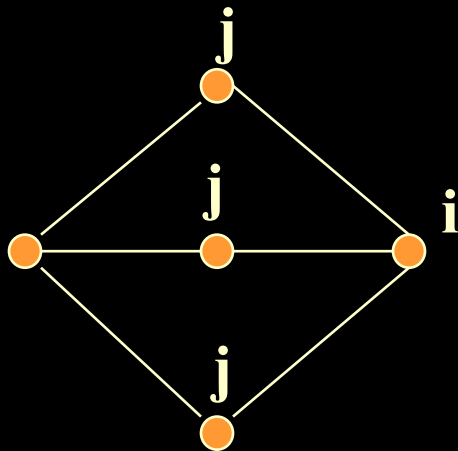


求各事件的可能最早发生时间

从 $Ve[0]=0$ 开始, 向前推进求其它事件的 Ve

$$Ve[i] = \max_j \{Ve[j] + \text{dur}(\langle V_j, V_i \rangle)\}, \langle V_j, V_i \rangle \in S_2, i=1, 2, \dots, n-1$$

S_2 是所有指向顶点 V_i 的有向边 $\langle V_j, V_i \rangle$ 的集合



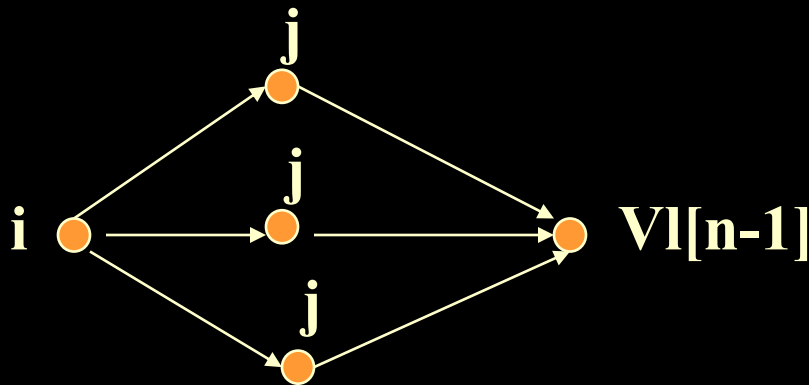
例如图中 $Ve[4]$ 为:
$$\left. \begin{array}{l} Ve[1] + \langle V_1, V_4 \rangle \text{的权} 1 = 6 + 1 = 7 \\ Ve[2] + \langle V_2, V_4 \rangle \text{的权} 1 = 4 + 1 = 5 \end{array} \right\} \text{取max}$$

■ 求各事件的允许最晚发生时间

从 $VI[n-1]=Ve[n-1]$ 开始, 反向递推

$$VI[i] = \min_j \{VI[j] - \text{dur}(<V_i, V_j>)\}, <V_i, V_j> \in S_1, i = n-2, n-3, \dots, 0$$

S_1 是所有从顶点 V_i 出发的有向边 $<V_i, V_j>$ 的集合



又如图中 $VI[4]$ 为:
$$\left. \begin{array}{l} VI[6] - \text{权} 9 = 16 - 9 = 7 \\ VI[7] - \text{权} 7 = 14 - 7 = 7 \end{array} \right\} \min$$

以上的计算必须在拓扑有序及逆拓扑有序的前提下进行
求 $Ve[i]$ 必须使 V_i 的所有前驱结点的 Ve 都求得
求 $VI[i]$ 必须使 V_i 的所有后继结点最晚发生时间都求得

- 求每条边（活动） $a_k = \langle V_i, V_j \rangle$ 的 $e[k], l[k]$
 $e[k] = Ve[i];$
 $l[k] = VI[j] - \text{dur}(\langle V_i, V_j \rangle), k=1, 2, \dots, e$
- 如果 $e[k] == l[k]$ 则 a_k 是关键活动

AOE网用邻接表来表示，并且假设顶点序列已按拓扑有序与逆拓扑有序排好。如上例：

逆拓扑次序

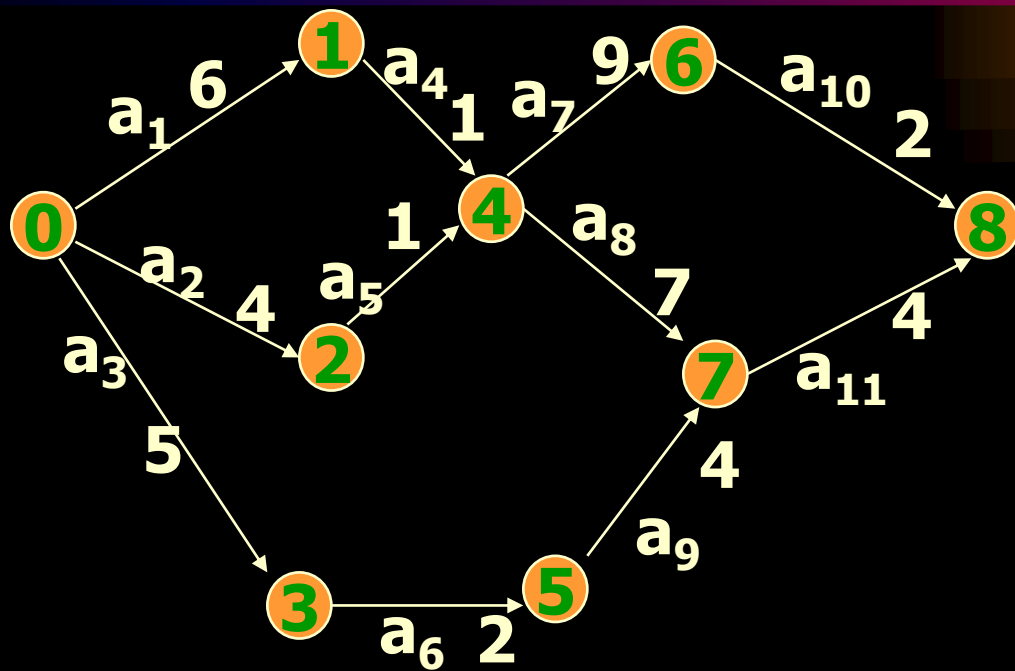
拓扑次序

count adj

dest dur link

0	0		→	1	6		→	2	4		→	3	5	^
1	1		→	4	1	^								
2	1		→	4	1	^								
3	1		→	5	2	^								
4	2		→	6	9		→	7	7	^				
5	1		→	7	4	^								
6	1		→	8	2	^								
7	2		→	8	4	^								
8	2	^												

NodeTable



void Graph ::CriticalPath ()

{ int i , j ; int p, k ; float e, l ;

1. float * Ve=new float [n]; float * Vl=new float[n];

2. for (i=0; i<n ; i++) Ve[i]=0;

3. for (i=0; i<n ; i++)

{ Edge <float> * p=NodeTable[i].adj;

while (p!=NULL)

{ k = p. dest;

if (Ve[i]+p. cost > Ve[k]) Ve[k]=Ve[i]+p. cost ;

p=p. link;

}

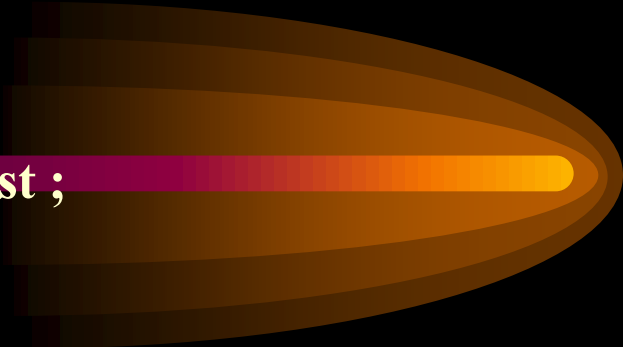
}

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

```

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

```



```

6. for (i=0; i<n ;i++)
{ p=NodeTable[i].adj;
  while (p!=NULL)
  { k= p. dest;
    e=Ve[i]; l=Vl[k]-p. cost ;
    if(l= =e)
      cout<<"<<i<<"", "<<k<<">"<<"is critical
      Activity"<<endl
    p=p. link;
  }
}
}

```



算法分析: 拓扑排序 $Ve[i]$
逆拓扑排序求 $VI[i]$
求各活动 $e[k]$ 和 $l[k]$

	}	$O(n+e)$	}	$O(n+e)$
		$O(e)$		

第9章 图的小结

一 图的基本概念

二 图的存储表示 邻接矩阵，邻接表

- 图的若干算法以及时间复杂性分析

- 1 图的遍历

- 深度优先搜索

- 广度优先搜索

- 2 最小生成树—Kruscal, Prim

- 3 最短路径 —Dijkstra, *Bellman—Ford, floyed

- 4 活动网络 —AOV, AOE

Chapter 9

2009年统考题(单选题):

10. 下列关于无向连通图特性的叙述中, 正确的是

a. 所有顶点的度之和为偶数 b. 边数大于顶点个数减1

c. 至少有一个顶点的度为 1

A. 只有a B. 只有b C. a和b D. a和c

2009年统考题(综合应用题):

(10分) 带权图 (权值非负, 表示边连接的两顶点间的距离) 的最短路径问题是找出从初始顶点到目标顶点之间的一条最短路径. 假设从初始顶点到目标顶点之间存在路径, 现有一种解决该问题的方法:

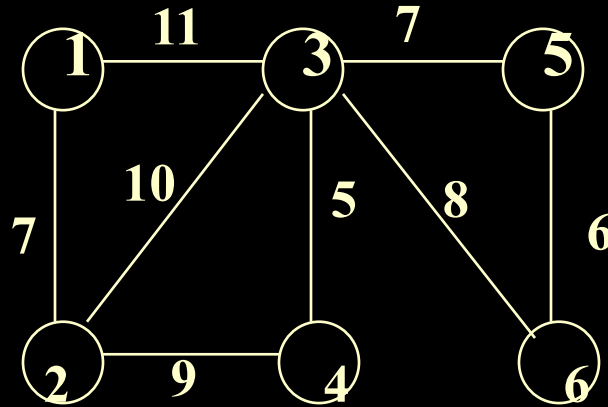
- 1) 设最短路径初始时仅包含初始顶点, 令当前顶点 u 为初始顶点;
- 2) 选择离 u 最近且尚未在最短路径中的一个顶点 v , 加入到最短路径中, 修改当前顶点 $u = v$;
- 3) 重复步骤2), 直到 u 是目标顶点时为止.

请问上述方法能否求得最短路径? 若该方法可行, 请证明之; 否则, 请举例说明.

Chapter 9

Exercise:

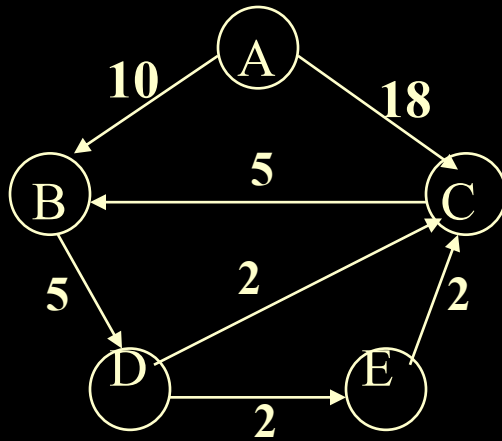
1. 对下列无向图:



分别用Prim算法与Kruscal算法，求出最小代价生成树（要求写出构造生成树的每一步）。

Chapter 9

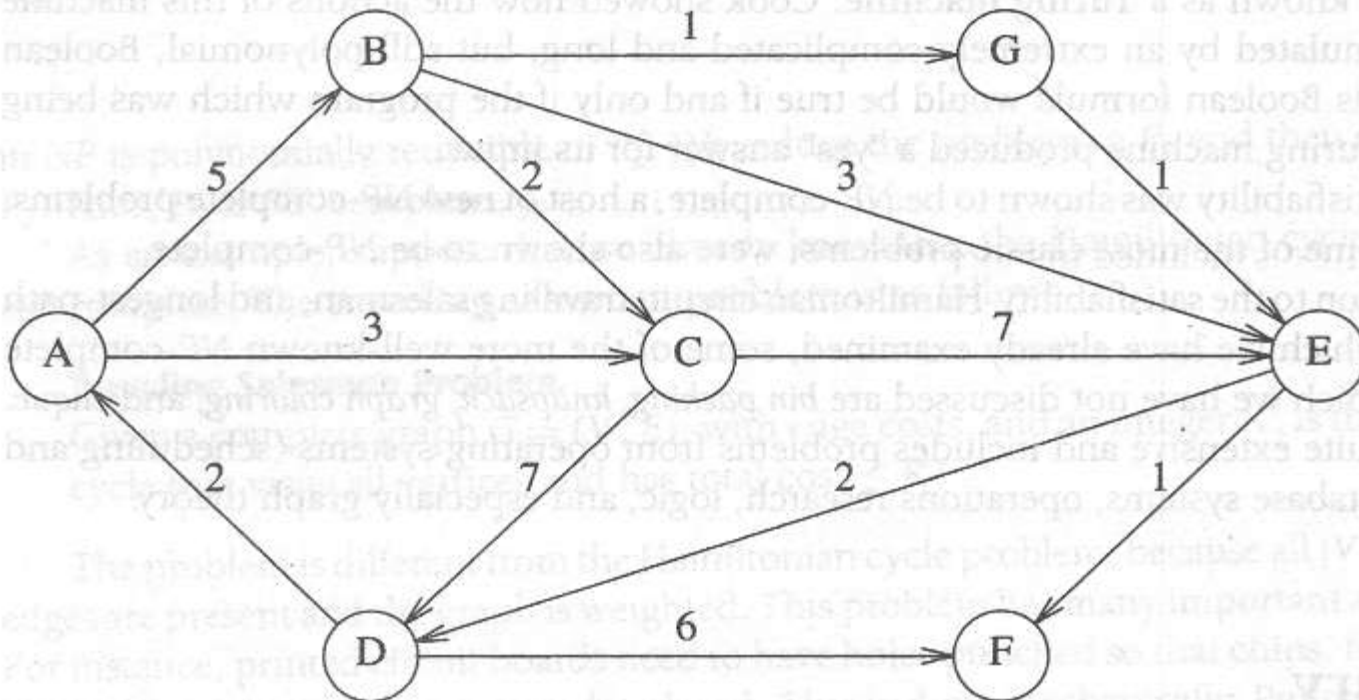
2. 对下列有向图:



用Dijkstra算法求从顶点A到其它各顶点的最短路径。

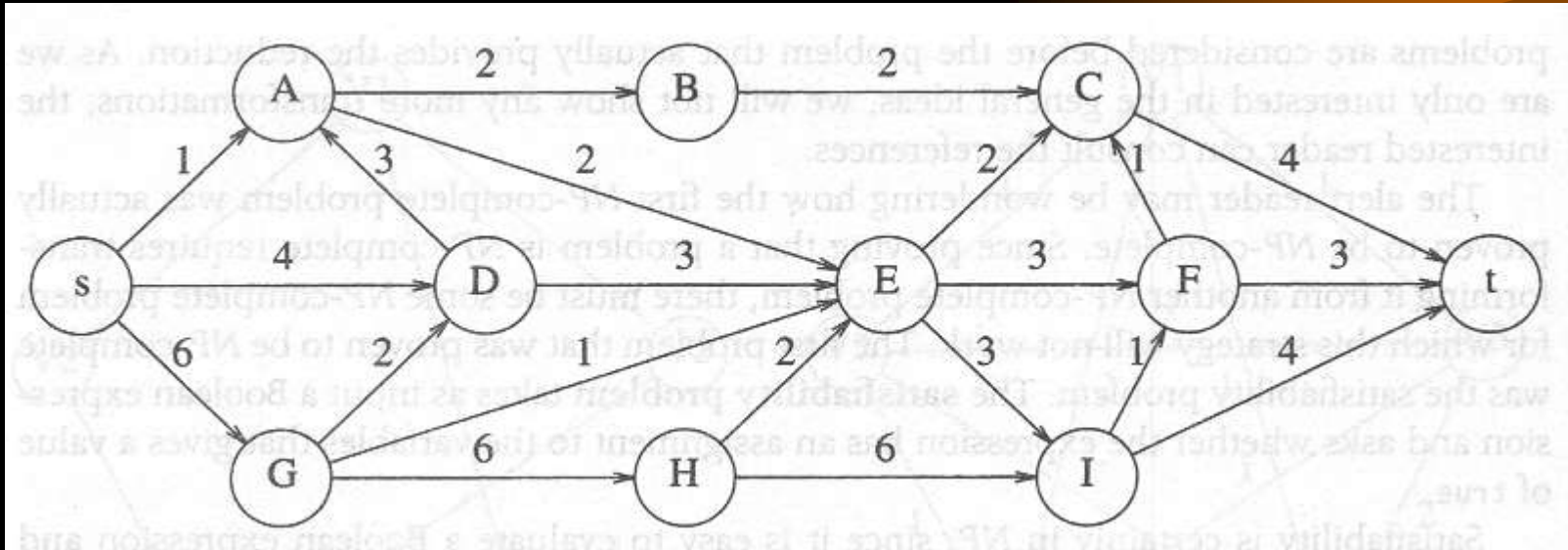
Chapter 9

3. a. Find the shortest path from A to all other vertices for the graph in 下图
b. Find the shortest unweighted path from B to all other vertices for the graph in 下图



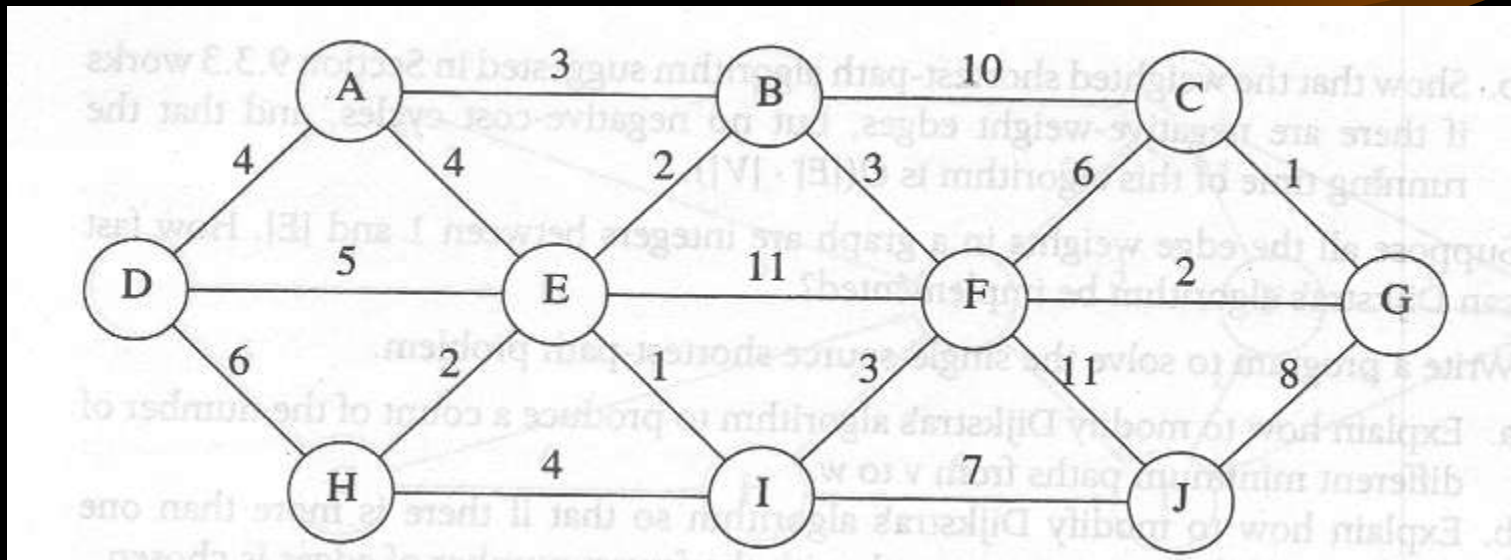
Chapter 9

4. Find a topological ordering for the graph in 下图



Chapter 9

- *5. a. Find a minimum spanning tree for the graph in 下图 using both Prim's and Kruskal's algorithms.
- b. Is this minimum spanning tree unique? Why?



实习题:

7. 判别一个有向图中是否有环路，并把所有环路打印出来。