第七章 图

宋国杰

gjsong@pku.edu.cn

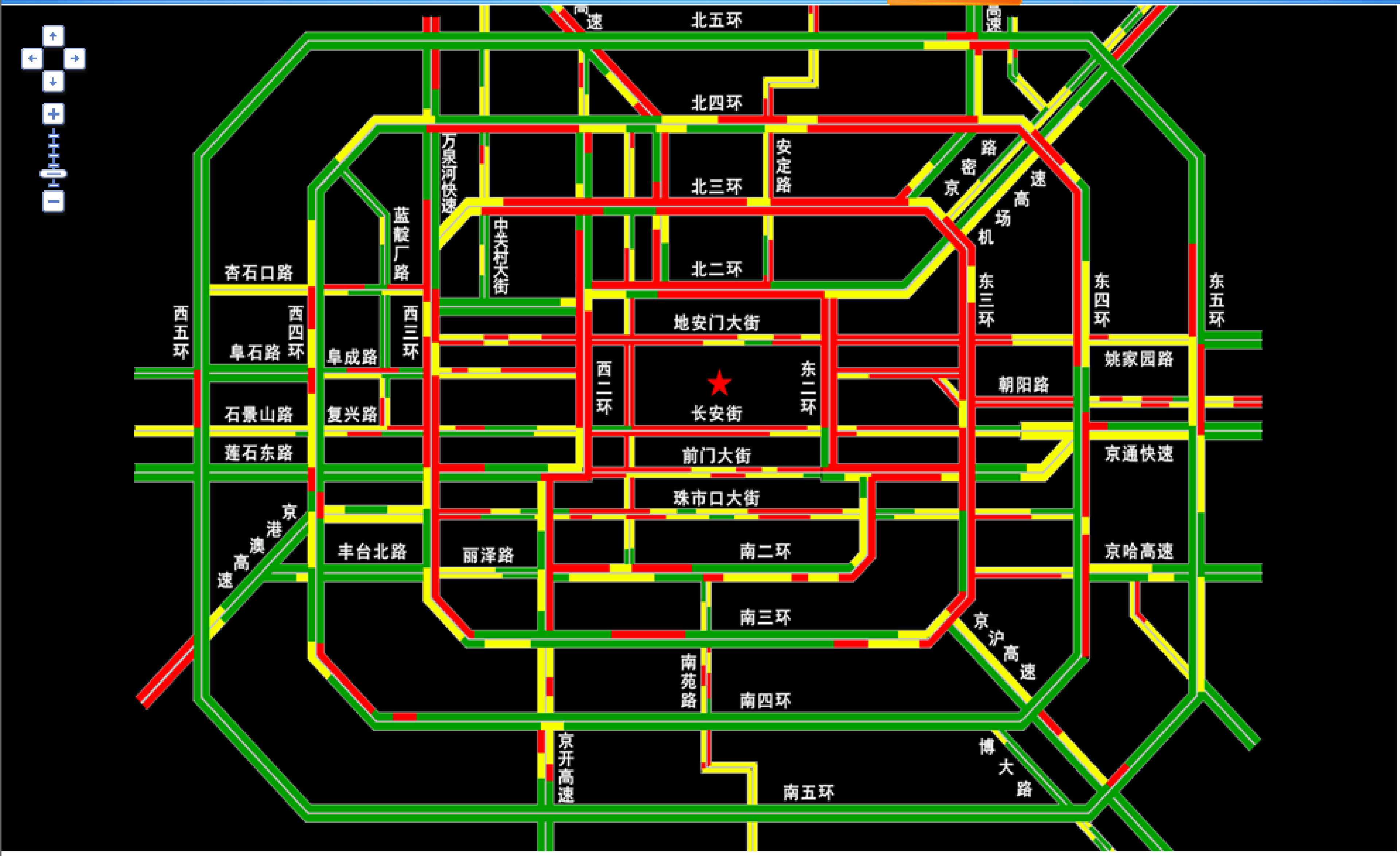
北京大学信息科学技术学院

课程内容

- 7.1 图的基本概念
- 7.2 图的抽象数据类型
- 7.3 图的存储结构
- 7.4 图的周游（深度、广度、拓扑）
- 7.5 最短路径问题
- 7.6 最小支撑树

让我们先来看看身边各种各样的网络

Road Network in Beijing

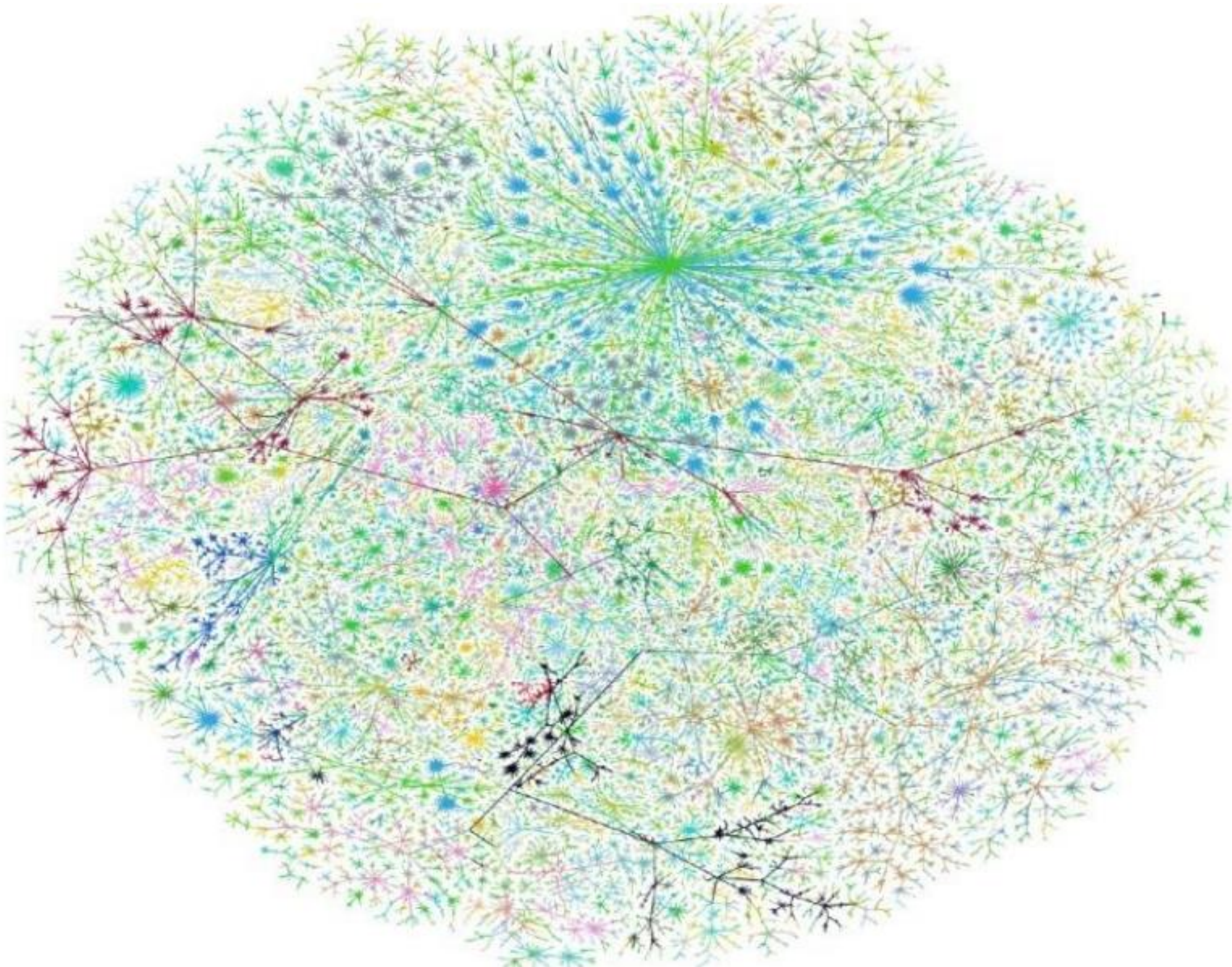


Friends & Family

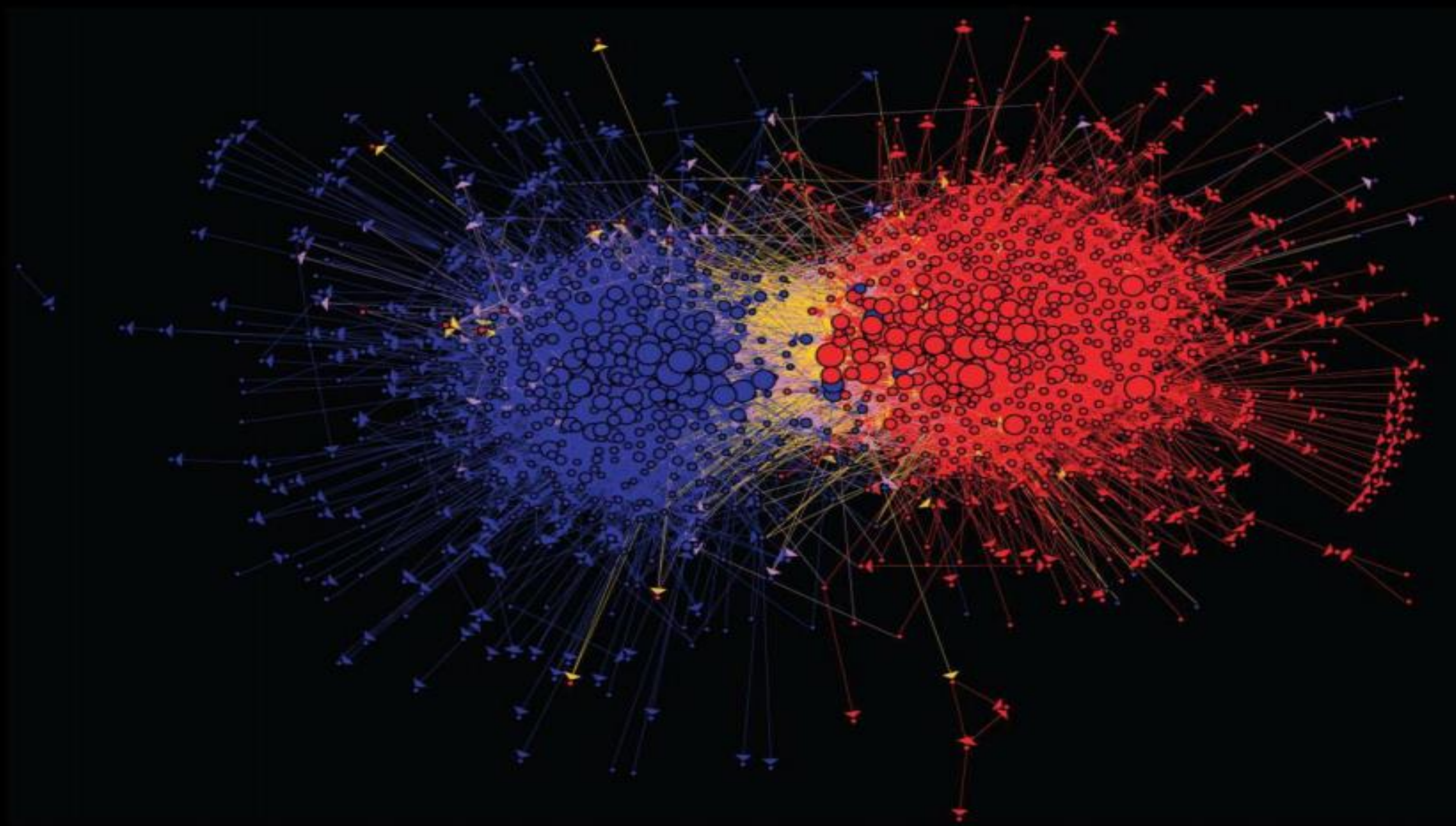
Brain

Network: Facebook Social Graph

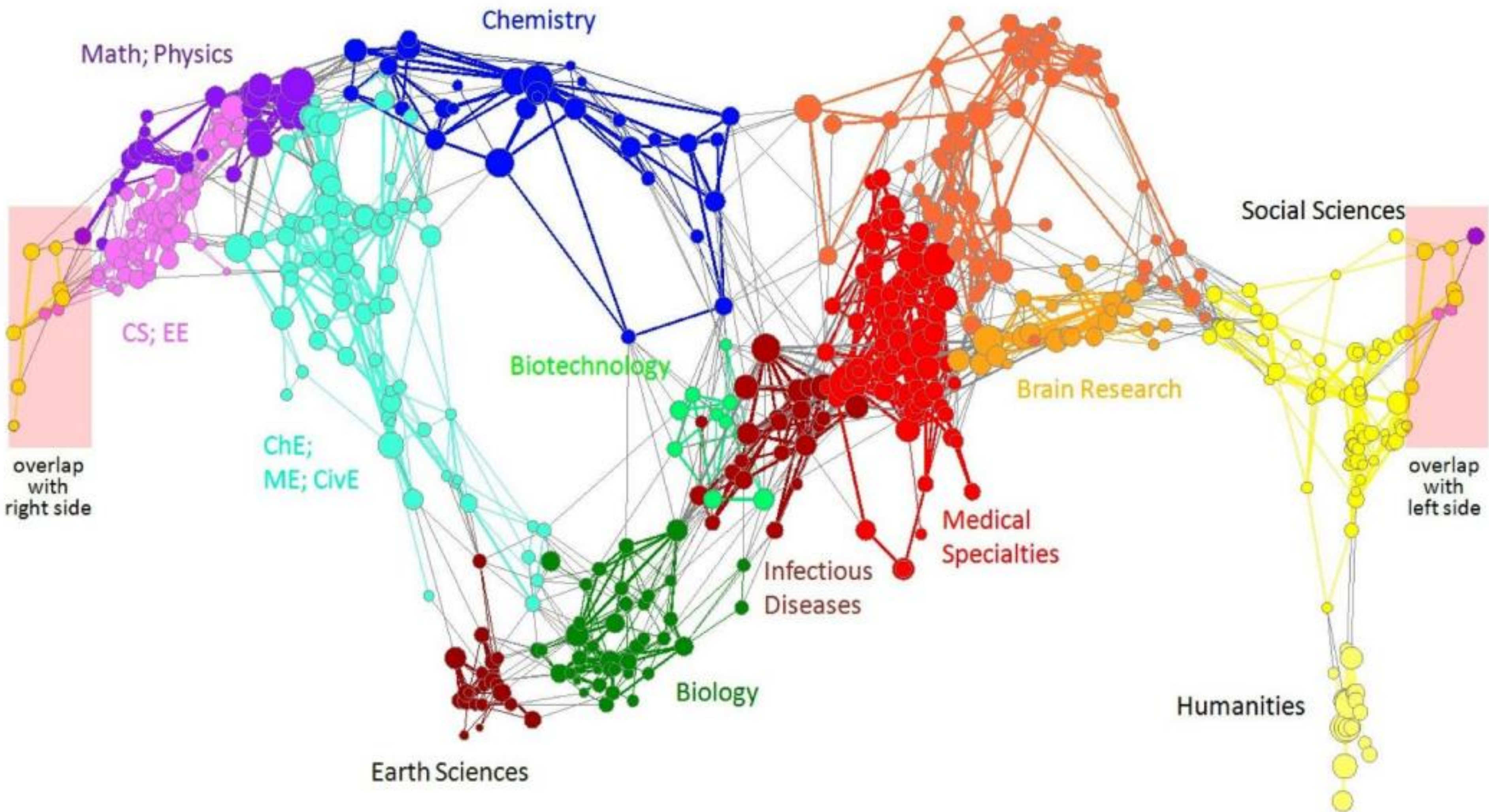
Network: Graph of the Internet



Network: Connections between political blogs

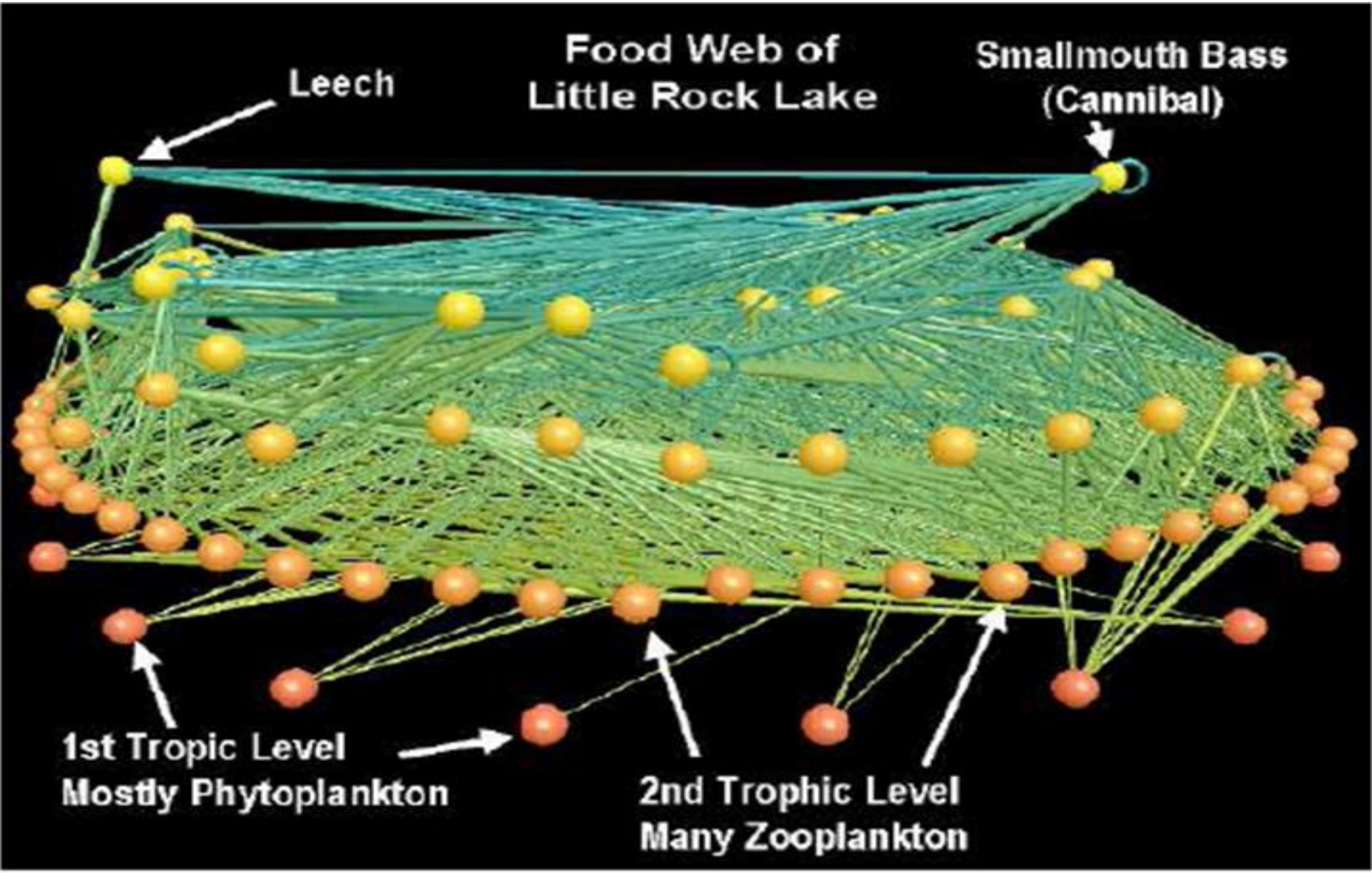


Network: Information network



Citation networks and Maps of science

Network: Food Webs

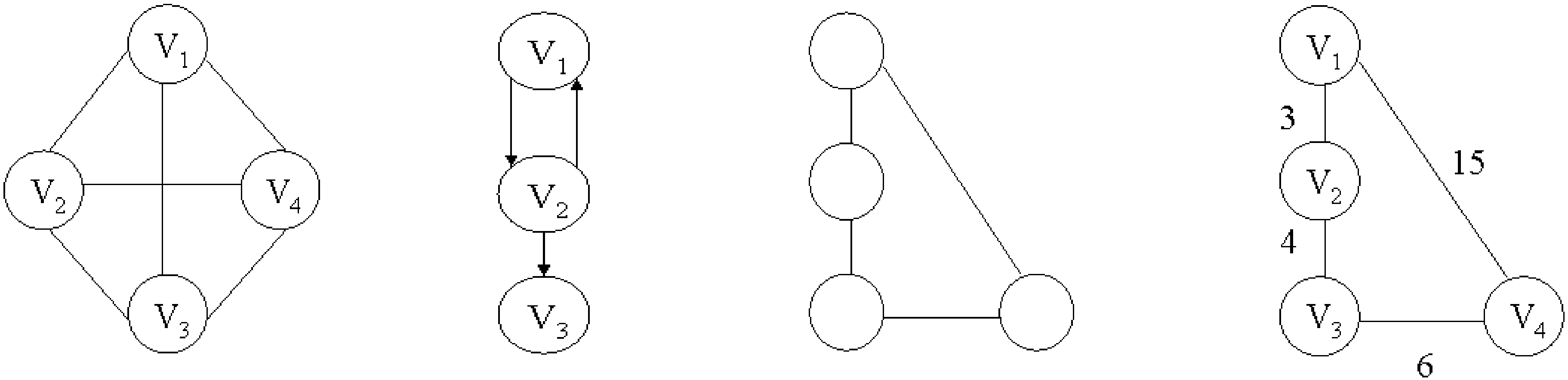


World Economy

7.1 图的基本概念

- 图的定义
  - 用G=（V，E）代表一个图
    - V表示有限的顶点集合
    - E表示边集合，是顶点的偶对(边的始点,边的终点)
    - |V|表示顶点的总数， |E|表示边的总数
- 图的不同分类
  - 稀疏图（sparse graph）：边数相对较少的图
  - 密集图（dense graph）：边数相对较多的图
  - 完全图（complete graph）：包括所有可能边的图
  - 无向图（undirected graph）：顶点偶对无序的图
  - 有向图（directed graph）：顶点偶对有序的图
  - 标号图（labeled graph）：各顶点均带有标号的图
  - 带权图（weighted graph）：边上标有权的图

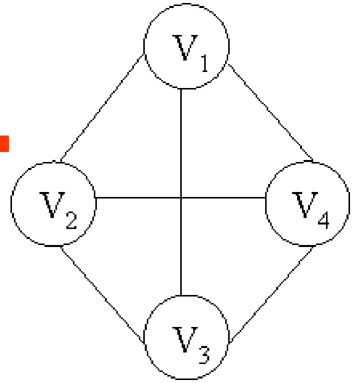
图示



G1 **G2** **G3** **G4**

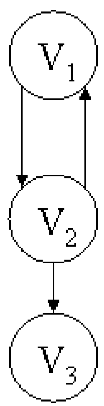
- 在上述四图中
  - **G1,G3,G4**是无向图
  - **G2**是有向图
  - **G1**是完全图：任何具有**n**个顶点的无向图，边数小于等于**n(n-1)/2**
  - **G1,G2,G4**是标号图
  - **G4**是带权图

图的集合表示



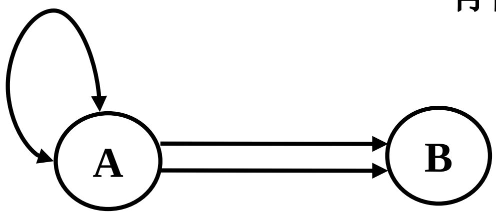
无向图G1

- 无向图G1的集合表示
  - G1=(V,E)
  - V(G1)={V_1，V_2，V_3，V_4}
  - E(G1)={(V_1,V_2),(V_1,V_3) ,(V_1,V_4) ,(V_2,V_3) ,(V_2,V_4) ,(V_3,V_4)}　　//圆括弧
- 有向图G2的集合表示
  - G2=(V,E)
  - V(G2)={V_1，V_2，V_3}
  - E(G2)={<V_1,V_2>,<V_2,V_1> ,<V_2,V_3>}　　//尖括弧



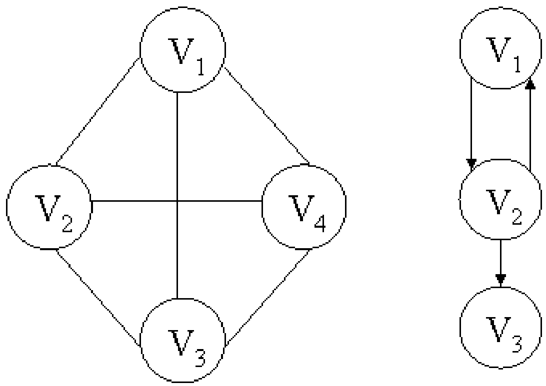
有向图G2

- 本章图，有约定
  - 不考虑顶点到自身的边
  - 不允许一条边在图中重复出现



图的相关概念

- 相邻顶点，或邻接点（neighbors）
  - 一条边所连接的两个顶点，称为邻接点
  - 这条边称为相关联的边

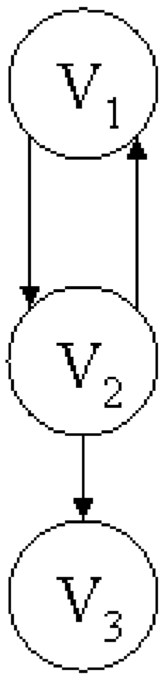


- 顶点的度（degree）
  - 定义为与该顶点相关联边的数目
  - 对于有向图G＝（V,E）而言
    - 入度(in degree)：以顶点V为终点的边的数目
    - 出度(out degree)：以顶点V为始点的边的数目
    - 终端结点(叶子)：出度为0的顶点
  - 若图G有n个顶点，*e*条边，d_i为顶点Vi的度数，则

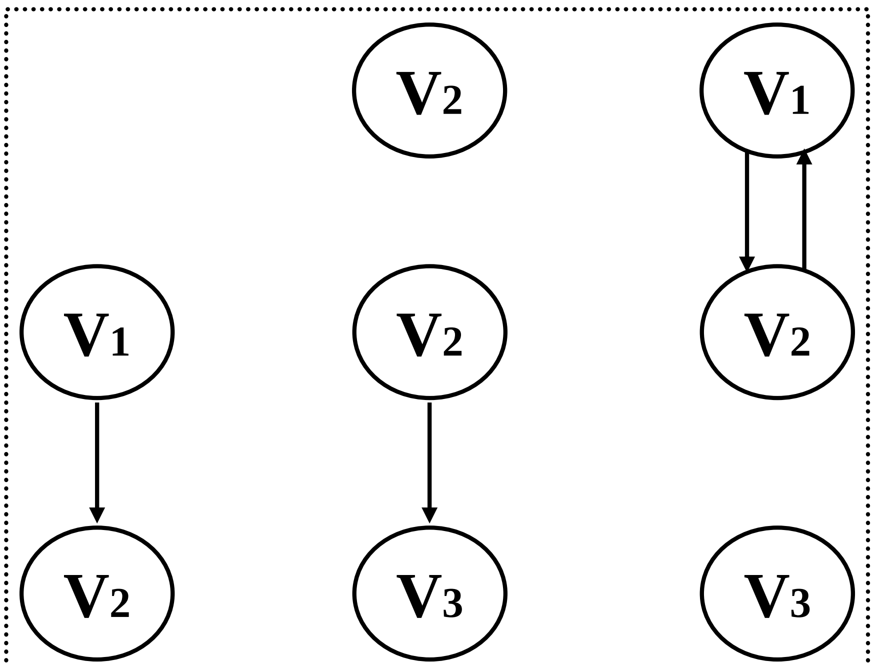
$$e = \frac{1}{2}\sum_{i=1}^{n} d_i$$

子图

- 子图（subgraph）
  - 图G=(V，E)，G’=(V’，E’)中，若V’⊆V，E’⊆E，并且E’中的边所关联的顶点都在V’中，则称图G’是图G的子图



(G2)



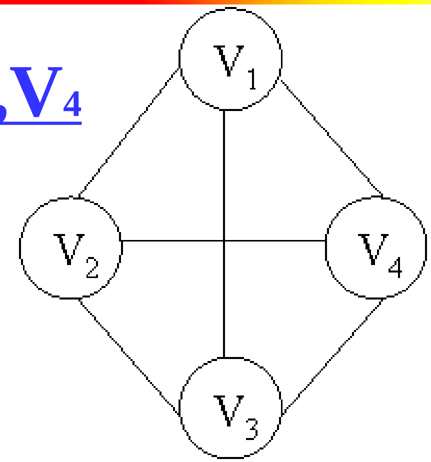
(G2的子图)

图的路径

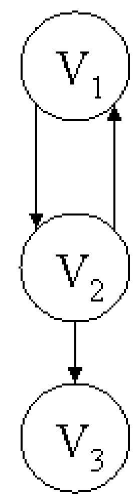
- 路径（path）
  - 在图G=(V，E)中，如果存在顶点序列V_p，V_{i1}，V_{i2}，…，V_{in}，V_q，使得（V_p，V_{i1}），(V_{i1}，V_{i2})，…，(V_{in}，V_q)（若对有向图，则使得<V_p，V_{i1}>，<V_{i1}，V_{i2}>，…，<V_{in}，V_q>）都在E中，则称从顶点Vp到顶点Vq存在一条路径。
- 简单路径（simple path）
  - 路径上除了V_p和V_q可以相同外，其它顶点都不相同
- 路径长度
  - 路径上边的条数
- 回路（也称环）：路经上某个顶点与自身连接
  - 简单回路：首尾顶点相同的简单路径
- 无环图（acyclic graph）
  - 有向无环图（directed acyclic graph，简写为DAG）

图示

- $(V_1,V_2)(V_2,V_3)(V_3,V_4)$可缩写成$V_1,V_2,V_3,V_4$
- 在图G1中
  - V_1,V_2,V_3是一条简单路径
  - V_1,V_3,V_4,V_1,V_3不是一条简单路径
  - V_1, V_2,V_3,V_4,V_2是路径，而且是回路，但不是简单回路
  - V_1, V_2,V_3,V_1是简单路径，而且是简单回路
- 在图G2中
  - V_1,V_2,V_3是一条简单的有向路径
  - V_1,V_2,V_3,V_2则不是路径



(G1)



(G2)

图论的起源

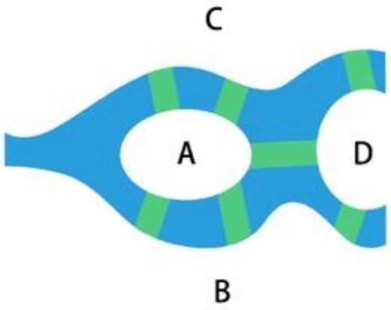
哥尼斯堡七桥问题：“一笔画”问题

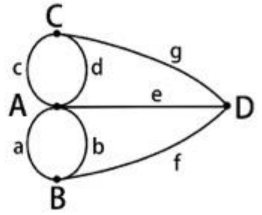
起源

- 18世纪哥尼斯堡城的普莱格尔河上有七座桥，能否一次走遍七座桥，并且每座桥只允许通过一次，最后仍可以回到起始地点?
- 1736年，欧拉运用数学抽象法将其转换为一笔画问题，并论述和证明了该过程是绝对不可能实现的

图论是计算机科学的数据结构和算法中最重要的框架（没有之一）。

哥尼斯堡七桥图

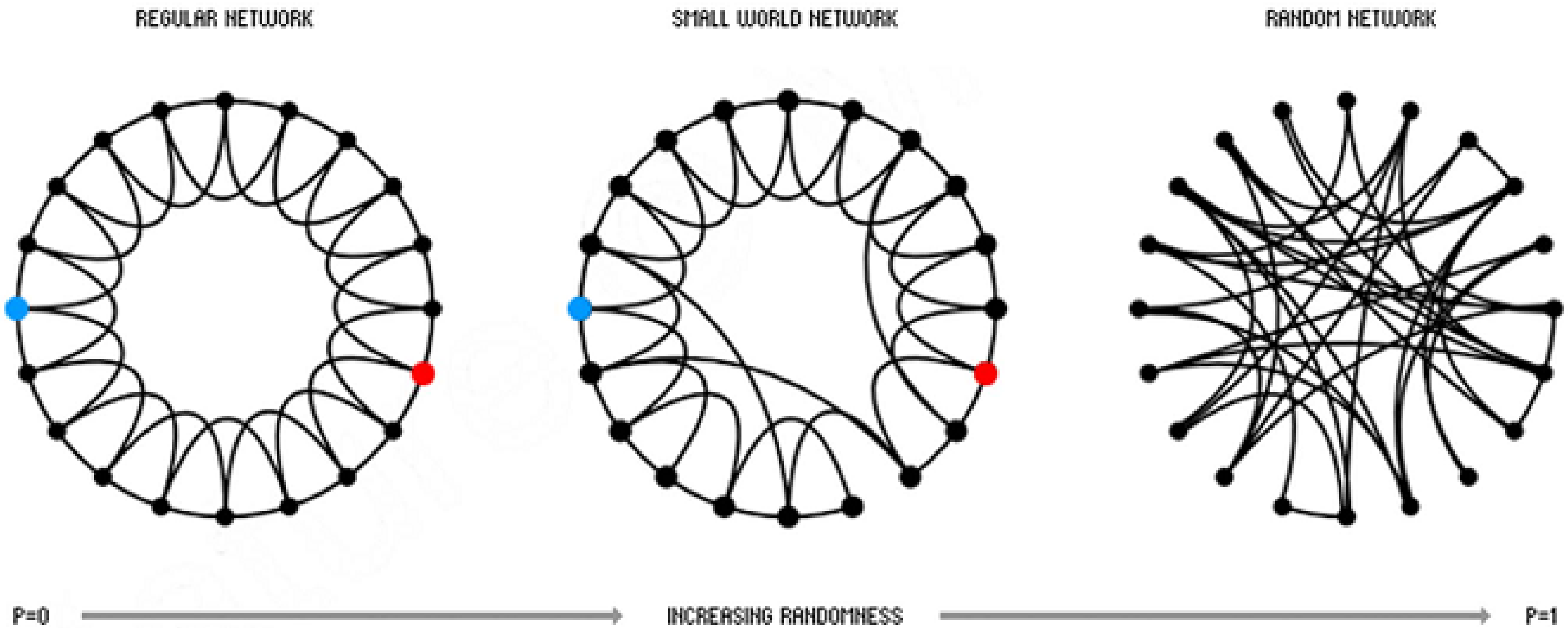




欧拉

奇点的数目不是0个就是2个

几种典型的图



High clustering
High diameter

$h = \frac{N}{2\bar{k}}$ $C = \frac{3}{4}$

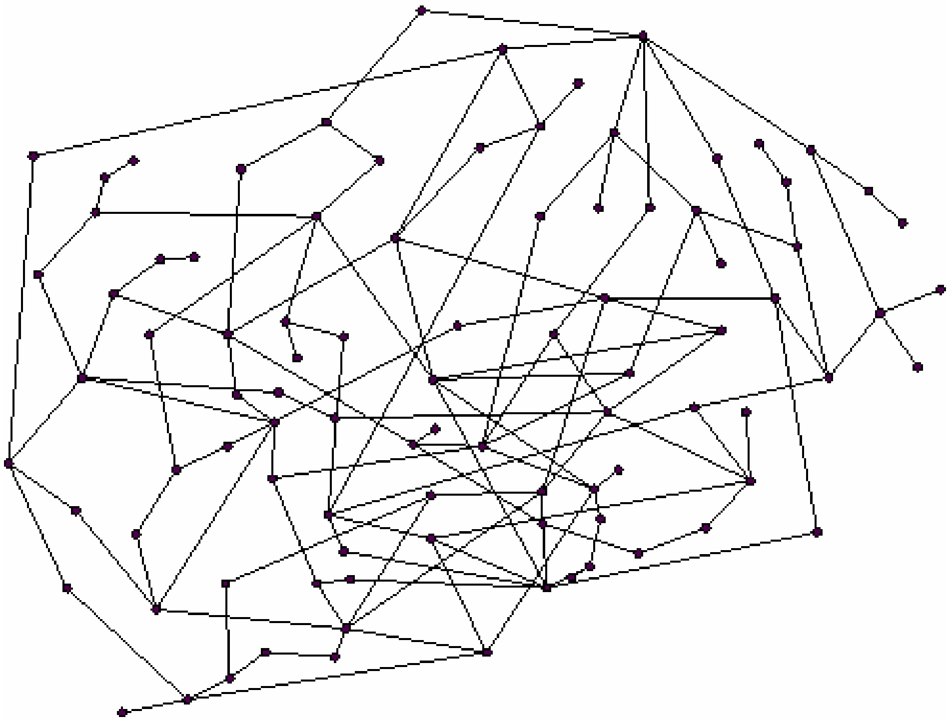
High clustering
Low diameter

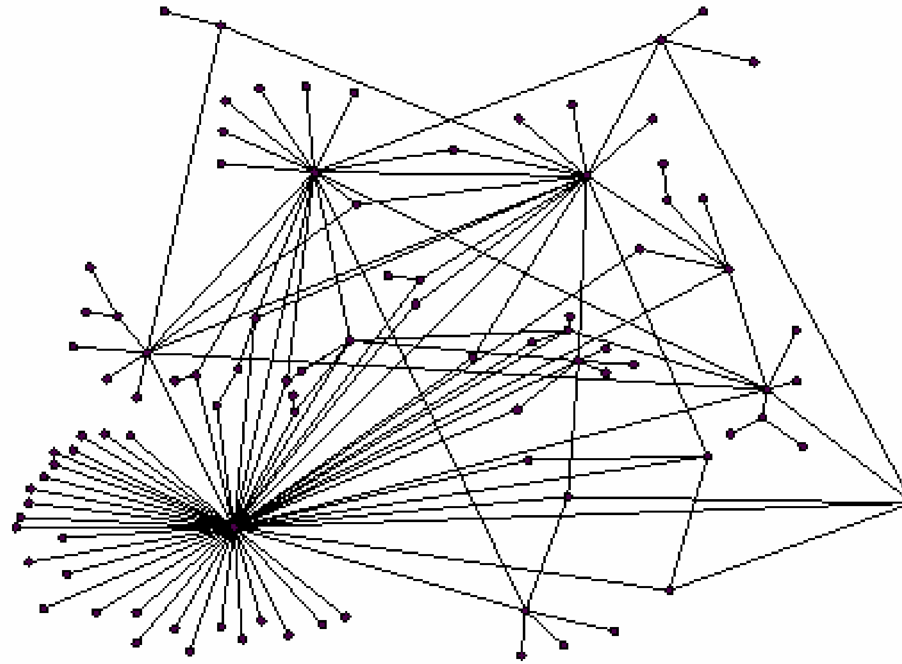
Low clustering
Low diameter

$h = \frac{\log N}{\log \alpha}$ $C = \frac{\bar{k}}{N}$

Rewiring allows us to “interpolate” between
a regular lattice and a random graph

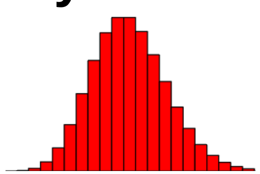
Scale-free network





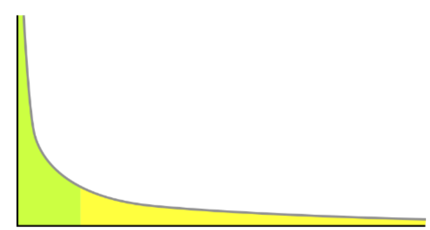
Poisson network

(Erdos-Renyi random graph)



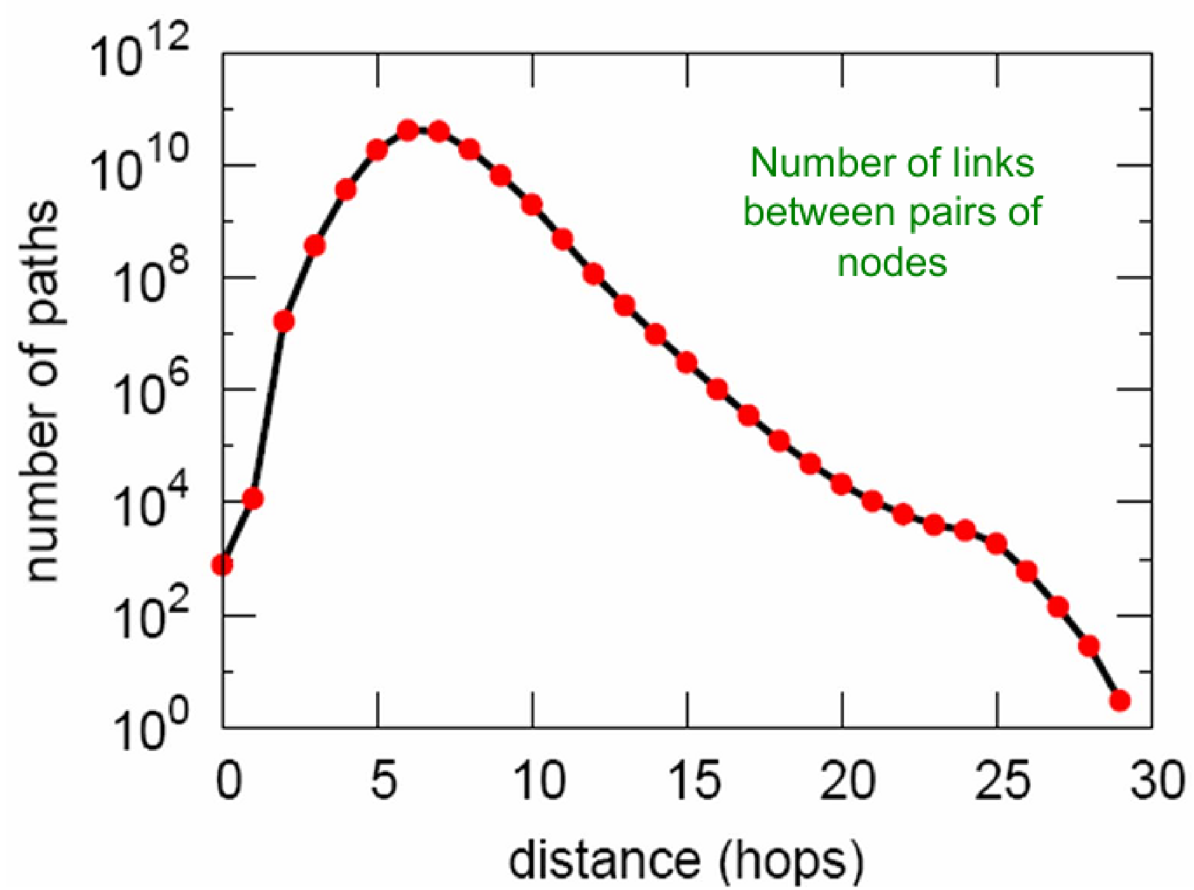
Degree distribution is Poisson

Scale-free (power-law) network



Degree distribution is Power-law

小世界现象



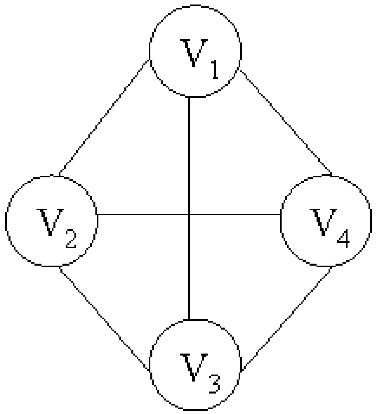
Avg. path length **6.6**
90% of the people can be reached in < 8 hops

图的连通性

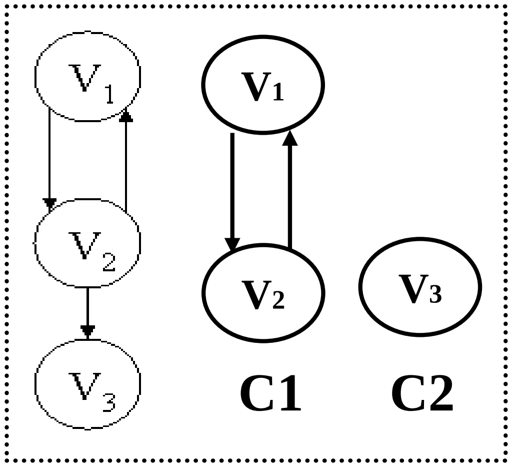
- 有根图
  - 有向图中，若存在顶点V_0，从此顶点有路径可以到达图中其它所有顶点，则称此有向图为有根图，V_0称作图的根
- 连通图
  - 对无向图G=(V, E)而言，如果从V_1到V_2有一条路径（从V_2到V_1也一定有一条路径），则称V_1和V_2是连通的
  - 若图G中任意两个顶点都是连通的，则无向图G是连通的
- 连通分量（连通分支）
  - 指无向图的**最大连通子图**

- 强连通性
  - 对有向图G=(V,E)而言，若对于G中任意两个顶点V_i和V_j（$V_i \neq V_j$），都有一条从V_i到V_j的（有向路径），同时还有一条从V_j到V_i的(有向)路径，则称有向图G是强连通的
- 强连通分量
  - 有向图强连通的最大子图
- 网络
  - 带权的连通图
- 自由树（**free tree**）
  - 不带有简单回路的无向图，它是连通的，且具有|V|-1条边

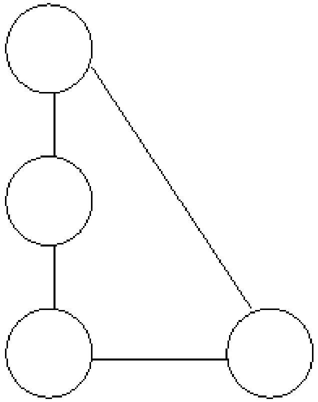
图示



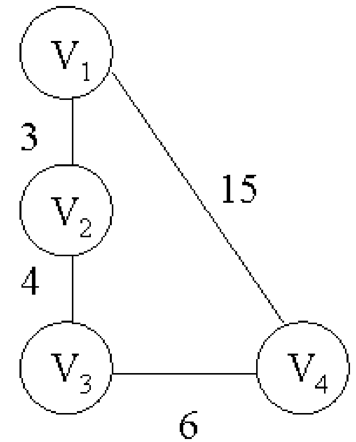
G1



G2

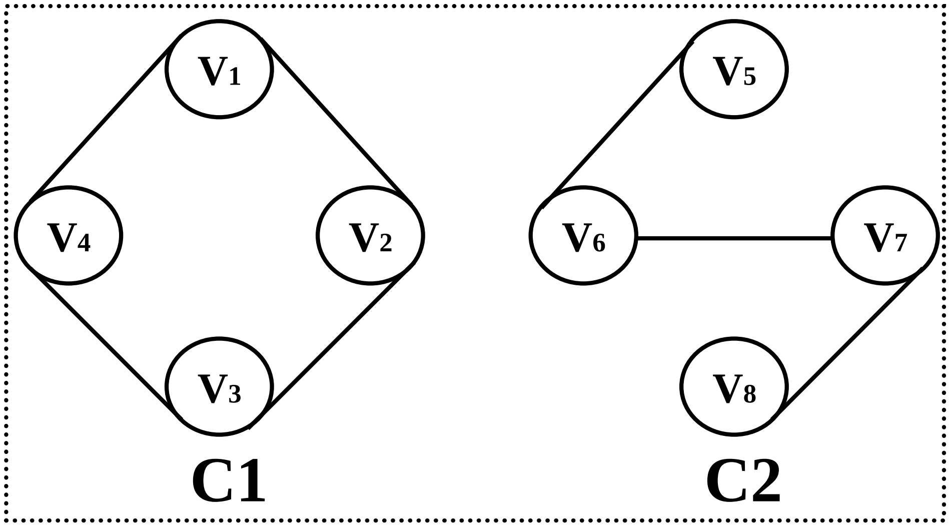


G3



G4

- 在上述四图中
  - G1,G3,G4是都是连通的
  - G5有两个连通分量C1和C2
  - G2不是强连通的
  - G2的两个强连通分量C1和C2
  - G4是网络

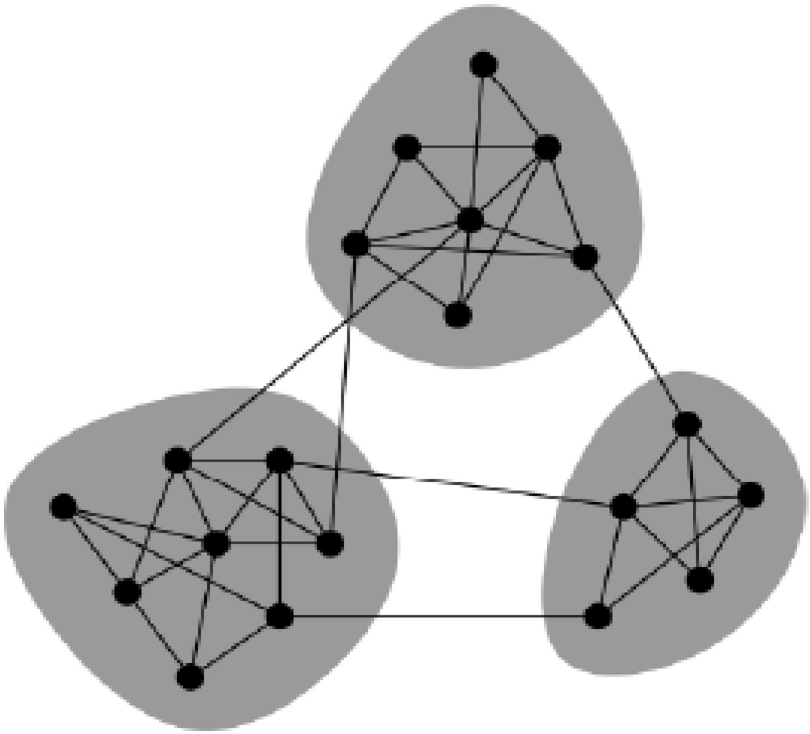


G5

Network communities

- Networks of **tightly connected groups**

- **Network communities:**
  - Sets of nodes with **lots** of connections **inside** and **few** to **outside** (the rest of the network)



Communities, clusters, groups, modules

7.2 图的抽象数据类型

```
class Graph{                        //图的ADT
public:
    int VerticesNum();              //返回图的顶点个数
    int EdgesNum();                 //返回图的边数
   //返回与顶点oneVertex相关联的第一条边
    Edge FirstEdge(int oneVertex);
    //返回与边PreEdge有相同关联顶点oneVertex的下一条边
    Edge NextEdge(Edge preEdge);
    //添加一条边
    bool setEdge(int fromVertex,int toVertex,int weight);
```

```
//删一条边
bool delEdge(int fromVertex,int toVertex);
//如果oneEdge是边则返回TRUE，否则返回FALSE
bool IsEdge(Edge oneEdge);
//返回边oneEdge的始点
int FromVertex(Edge oneEdge);
 //返回边oneEdge的终点
int ToVertex(Edge oneEdge);
//返回边oneEdge的权
int Weight(Edge oneEdge);
};
```

7.3 图的存储结构

- **7.3.1 相邻矩阵**
- **7.3.2 邻接表**
- **7.3.3 十字链表**

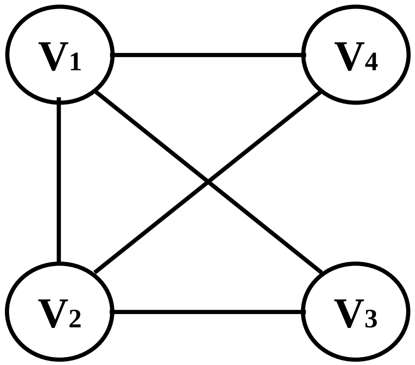
1、图的相邻矩阵表示法

- 相邻矩阵
  - 表示顶点间相邻关系的矩阵
  - 若G是一个具有n个顶点的图，则G的相邻矩阵是如下定义的n×n矩阵

$$A[i,j]=\begin{cases}1, & \text{若}(V_i,V_j)\text{或}<V_i,V_j>\text{是图G的边}\\ 0, & \text{若}(V_i,V_j)\text{或}<V_i,V_j>\text{不是图G的边}\end{cases}$$

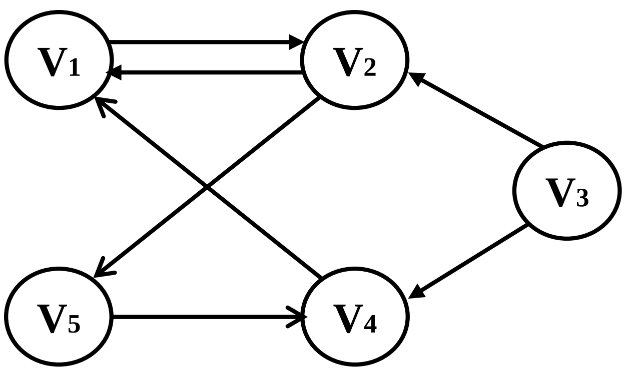
- 相邻矩阵的空间代价为O(n²)，与边数无关

图示



G6

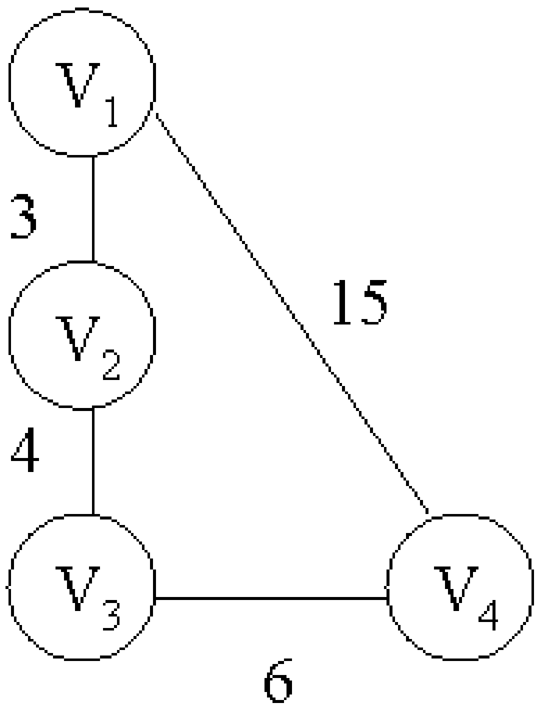
$$A_6 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$



G7

$$A_7 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

➢ 加权矩阵

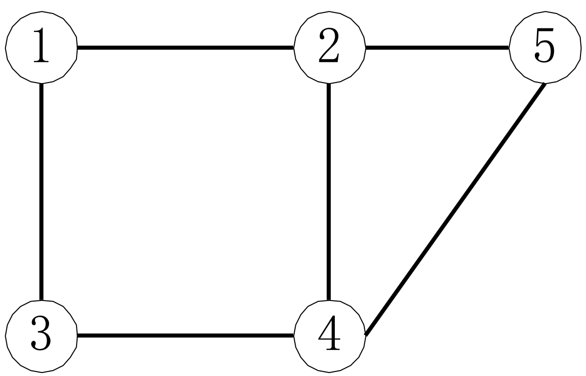


G4

$$A_4 = \begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

性质1

- 从无向图的邻接矩阵可以得出如下结论:
  - 矩阵对称;
  - 第i行或第i列中1的个数为顶点i的度;
  - 矩阵中1的个数的一半为图中边的数目;
  - 容易判断顶点i和顶点j之间是否有边相连



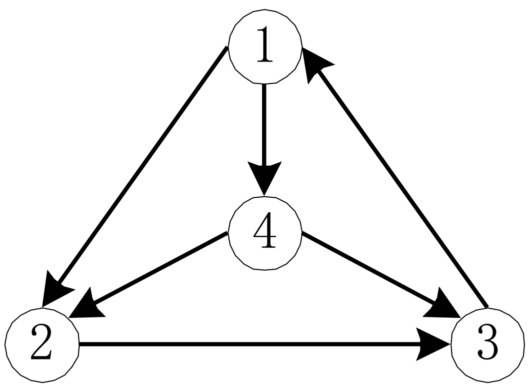
(a)

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

(b)

性质2

- 从有向图的邻接矩阵可以得出如下结论：
  - 矩阵不一定对称;
  - 第i行中1的个数为顶点i的出度;
  - 第i列中1的个数为顶点i的入度;
  - 矩阵中1的个数为图的边数;
  - 容易判断顶点i和顶点j是否有边相连



(a)

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

(b)

边的基类

```
Class Edge{                                  //边的基类
Public:
    int from, to, weight;
    Edge(){                                  //构造函数
        from=-1; to=-1; weight=0;
    }
    Edge(int f, int t, int w){               //构造函数
        from=f; to=t; weight=w;
    }
    bool operator > (Edge oneEdge){          //定义边比较运算符 “>”
        return  weight>oneEdge.weight;
    }
    bool operator < (Edge oneEdge){          //定义边比较运算符 “<”
        return  weight<oneEdge.weight;
    }
}
```

图的基类

```
Class Graph{                          //图的基类
Public:
    int numVertex,  numEdge;          //分别表示图的顶点和边的数目
    int *indegree, *Mark;             //顶点入度数组和是否被访问过的标记数组

    Graph(int numVert){                               //构造函数
        numVertex= numVert;
        numEdge=0;
        indegree = new int[numVertex];                //入度数组;
        Mark = new int[numVertex];                    //顶点访问标志数组;
        for (int i=0; i<numVertex;i++)
        {
            Mark[i]= UNVISITED;
            Indegree[i]=0;
        }
    }
```

```
~Graph(){                          //析构函数
    delete[] mark;
    delete[] indegree;
}
virtual Edge FirstEdge (int oneVertex)=0;       //返回顶点的第一条边;
virtual Edge NextEdge (Edge preVertex)=0;       //返回当前边的下一条边;
int verticesNum() {return  numVertex};          //返回顶点数;
int EdgesNum() {return  numEdge};               //返回边数;
bool isEdge(Edge  oneEdge){                     //判断oneEdge是否为边;
    if (oneEdge.weight>0 && oneEdge.weight<INFINITY &&
        oneEdge.to>=0)     return true;
    else   return false;
}
```

```
    int FromVertex(Edge oneEdge)                    //返回边oneEdge的始点
        {return oneEdge.from;}
    int ToVertex(Edge oneEdge)                      //返回边oneEdge的终点
        {return oneEdge.to;}
    int Weight(Edge oneEdge)                        //返回边oneEdge的权
        {return oneEdge.weight;}
    virtual void setEdge(int from, int to, int weight)=0;       //设置边
    virtual void delEdge(int from, int to)=0;                   //删除边
};
```

基于邻接矩阵的类表示

```
Class Graphm: public Graph{
private:
    int ** matrix;
Public:
    Graphm(int numVert) : Graph(numVert) {        //构造函数
        int i, j;
        matrix = (int **) new int* [numVertex];   //声明一个相邻矩阵；
        for (int i=0; i<numVertex; i++)           //构造一个相邻矩阵；
            matrix[i]= new int[numVertex];
        for (int i=0; i<numVertex; i++)           //初始化相邻矩阵；
            for (int j=0; j<numVertex; j++)
                matrix[i][j]=0;
    }
```

```
~Graph(){                        //析构函数
    for (int i=0; i<numVertex; i++)
        delete[] matrix[i];
    delete[] matrix;
}
Edge FirstEdge(int oneVertex) {        //返回顶点oneVertex的第一条边;
    Edge  myEdge;
    myEdge.from = oneVertex;  myEdge.to = -1;
    for (int i=0; i<numVertex; i++) {
        if (matrix[oneVertex][i]!=0){
            myEdge.to=i;     myEdge.weight = matrix[oneVertex][i];
            break;
        }
    }
    return myEdge;
}
```

```
Edge NextEdge(Edge preEdge) { //返回与边preEdge有相同顶点的下一条边
        Edge  myEdge;
        myEdge.from = preEdge.from;
        myEdge.to = -1;                    //-1可以判断是非边
        for (int i=preEdge.to+1; i<numVertex; i++) {
            if (matrix[preEdge.from][i]!=0){
                myEdge.to=i;
                myEdge.weight = matrix[oneVertex][i];
                break;
            }
        }
        return myEdge;
    }
```

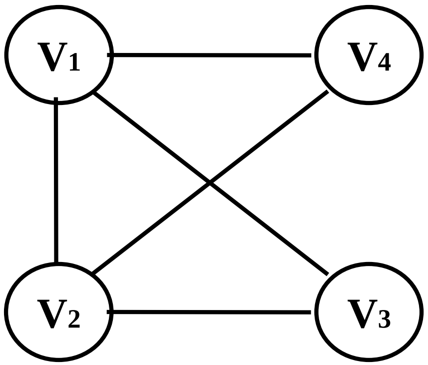
```
void setEdge(int from, int to, int weight) {        //为图设定一条边;
        if (matrix[from][to]<=0){
                numEdge++;
                indegree[to]++;
        }
        matrix[from][to]=weight;
}
void delEdge(int from, int to) {                    //删掉图的一条边;
        if (matrix[from][to]>0){
                numEdge--;
                indegree[to]--;
        }
        matrix[from][to]=0; //0可以判断是非边
}
```

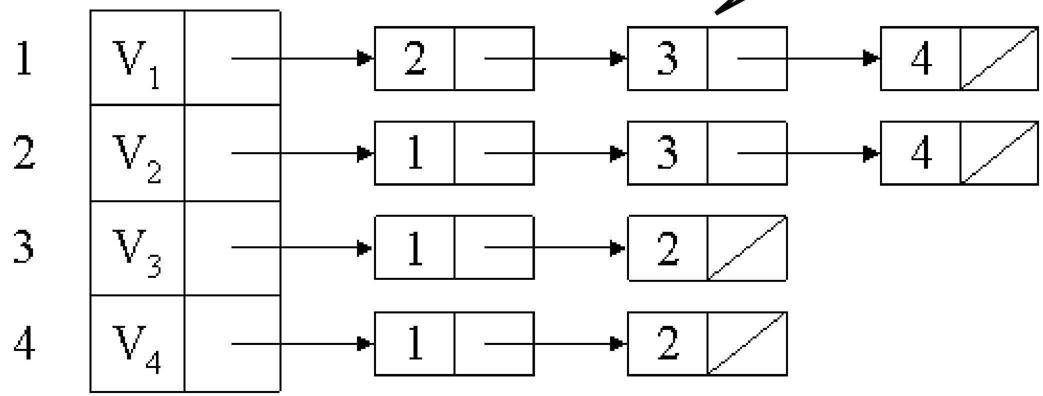
2、图的邻接表表示法

- 邻接矩阵表示法
  - 矩阵的规模只与顶点的个数n有关（n^2）
  - 与边无关
  - 由于大量的边不存在，造成空间浪费
- 邻接表表示法
  - 即与顶点有关，又与边有关
  - 顶点表：n个顶点，包括顶点数据和指向边表的指针
  - 边链表：m条边，包括顶点序号和指向下一表目的指针

图示1

边链表





无向图G6

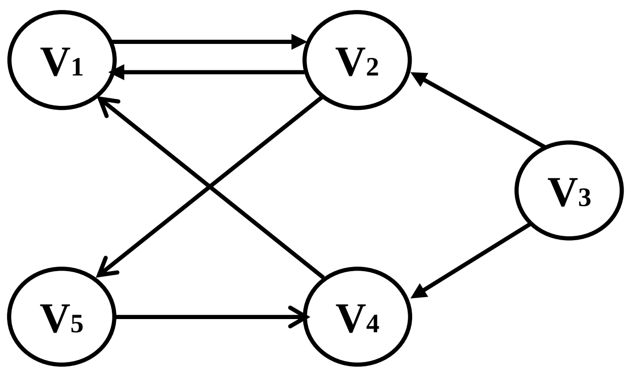
顶点表

无向图G6的邻接表表示

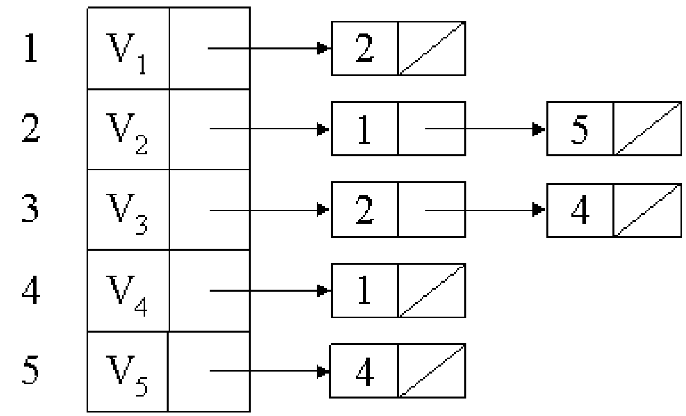
需要|V|＋2 |E|个存储单元

边链表中表目顺序往往按照顶点编号从小到大排列。

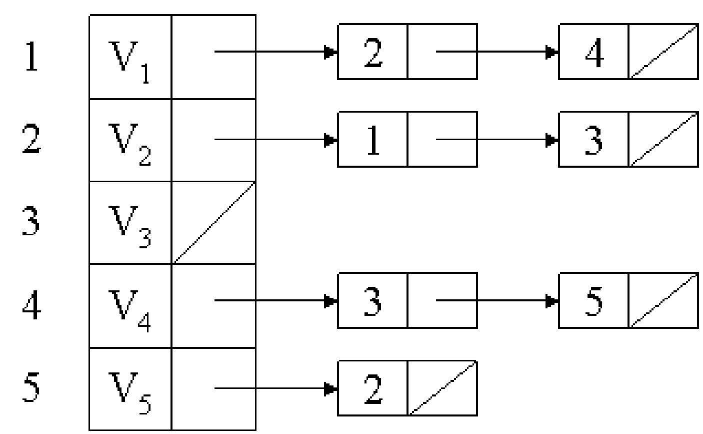
图示2



有向图G7



（a）有向图G7的出边表



（b）有向图G7的入边表

1、保存出边表和出边表之一即可

2、需要|V| + |E|个存储单元

基于邻接表的类表示

```
Struct listUnit{                          //邻接表表目中数据部分的结构定义
    int vertex;                                                  //边的终点;
    int weight;                                                  //边的权;
};
Template <class Elem>                                            //链表元素
class link{
Public:
    Elem  element;                                               //顶点元素;
    link *next;                                                  //指针元素;
    link (const Elem& elemval, Link *nextval=NULL){              //构造函数
        element  = elemval;                                      //初始化顶点值;
        next = nextval;                                          //初始化指针值;
    }
    link (link  * nextval = NULL) {next = nextval;}              //构造函数
};
```

```
template <class Elem>                          //链表
class LList{
public:
    Link <Elem> *head;                         //定义头指针head，只为操作方便
    LList(){                                   //构造函数；
        head = new Link<Elem>();               //定义一个头结点；
    }
    void removeall(){                          //释放边表所有表目占据的空间；
        Link<Elem> * fence;
        while(head!=NULL){
            fence =head;  head=head->next; delete fence;
        }
    }
    ~LList(){removeall();}                     //析构函数
};
```

```
class Graphl:  public Graph{
    private:
    LList <listunit>  * graList;        //保存所有边表的数组;
    Public:
    Graphl(int numVert):Graph(numVert){         //构造函数
        graList =  new LList <listUnit>[numVertex];
    }
    ~Graphl(){                                  //析构函数
        delete[] graList;
    }
```

```
Edge firstEdge(int oneVertex)            //返回顶点oneVertex的第一条边
{
    Edge myEdge;
    myEdge.from = oneVertex;
    myEdge.to = -1;
    Link<listUnit> *temp = graList[oneVertex].head; //取对应链表的头
    if (temp->next != NULL)
    {
        myEdge.to = temp->next->element.vertex;
        myEdge.weight = temp->next->element.weight;
    }
    return  MyEdge;
}
```

Edge nextEdge(Edge preVertex) { //返回与preEdge有相同顶点的下一条边

```
    Edge myEdge;
    myEdge.from = preEdge.from;
    myEdge.to = -1;
    Link<listUnit> *temp = graList[preEdge.from].head;
    while (temp->next != NULL && temp->next->element.vertex <=preEdge.to)
            temp=temp->next;
    if (temp->next!=NULL)
    {
            myEdge.to = temp->next->element.vertex;
            myEdge.weight = temp->next->element.weight;
            return  MyEdge;
    }
}
```

Void setEdge(int from, int to, int weight){ //为图设定一条边;

```
Link <listUnit> * temp = graList[from].head;
while (temp->next!=NULL && temp->next->element.vertex<to)
        temp=temp->next;                          //确定要插入边表的位置
if (temp->next==NULL){                            //为空，在尾部插上这条边
        temp->next=new Link<listUnit>;
        temp->next->element.vertex = to;
        temp->next->element.weight = weight;
        numEdge++;
        indegree[to]++;
        return;
}
if (temp->next->element.vertex == to){ //边已经在图中存在，只改权值;
        temp->next->element.weight = weight;
        return;
}
```

```
    if (temp->next->element.vertex > to){  //不存在，在边表中插入该条边；
        Link <listUnit> * other = temp->next;
        temp->next=new Link<listUnit>;
        temp->next->element.vertex = to;
        temp->next->element.weight = weight;
        temp->next->next = other;
        numEdge++;
        indegree[to]++;
        return;
    }
}//end setEdge;
```

Void delEdge(int from, int to){ //删除图的一条边；

```
    Link <listUnit> * temp = graList[from].head;
    while (temp->next!=NULL && temp->next->element.vertex<to)
        temp=temp->next;                            //确定要插入边表的位置
    if (temp->next==NULL)                           //边在图中不存在；
        return;
    if (temp->next->element.vertex > to)            //边在图中不存在；
        return;
    if (temp->next->element.vertex == to){          //边在图中存在；
        Link <listUnit> * other = temp->next->next;
        delete temp->next;
        temp->next = other;
        numEdge--;
        indegree[to]--;
        return;
    }
  }
}
```

3、十字链表

- 另一种链式存储结构，可看成是邻接表和逆邻接表的结合
- 顶点表：对应图的顶点，由3个域组成
  - data域
  - firstinarc指针指向第一条以该顶点为终点的边
  - firstoutarc指针指向第一条以该顶点为起点的边
- 边链表：对应有向图的每一条边，共5个域：
  - 起点(fromvex)和终点(tovex)的顶点序号；边权值的info域；
  - fromnextarc指针指向下一个以fromvex为起点的边；
  - tonextarc指针指向下一条以tovex为终点的边；

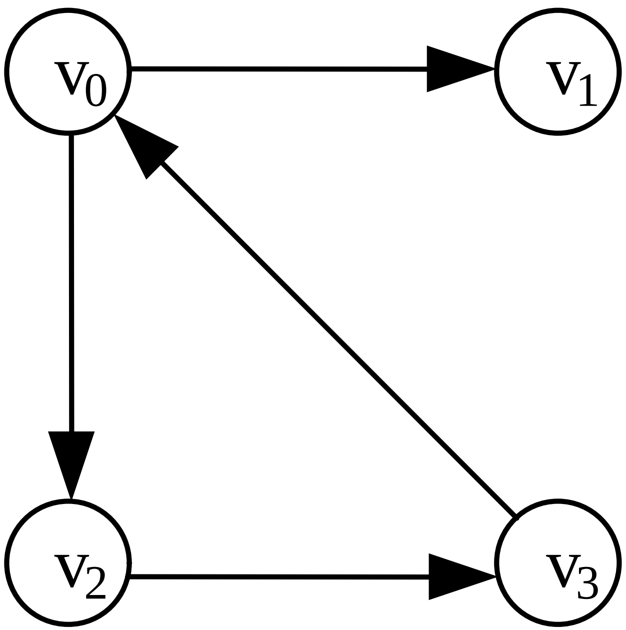
data	firstinarc	
	firstoutarc	

顶点结点

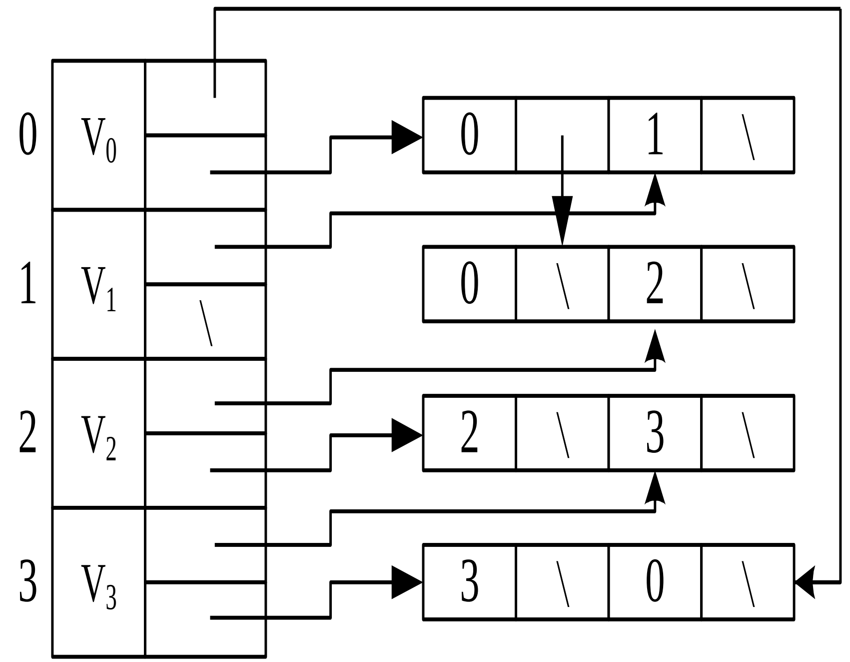
fromvex	fromnextarc	tovex	tonextarc	info

弧(有向边)结点

图示



有向图



顶点表 边表

7.4 图的周游

- 图周游（Graph Traversal）
  - 给定图G和任一顶点V_0，从V_0出发系统地访问G中所有的顶点，每个顶点访问一次，称为图周游
- 图周游的典型算法
  - 从一个顶点出发，访问其余顶点，需考虑下列情况：
    - 非连通图：从一顶点出发，可能不能到达所有其它的顶点
    - 存在回路的图：也有可能会陷入死循环
  - 解决办法
    - 顶点保留一标志位，初始时标志位置未访问（UNVISITED）
    - 在周游过程中，当顶点被访问时，标志位置已访问（VISITED）

周游算法

```
void graph_traverse(Graph& G)
{  //图的周游算法的实现
    for(int i=0;i<G.VerticesNum();i++)  //初始化标志位
        G.Mark[i]=UNVISITED;
    for(int i=0;i<G.VerticesNum();i++)
        if(G.Mark[i]== UNVISITED)
            do_traverse(G, i); //do_traverse函数
}
```

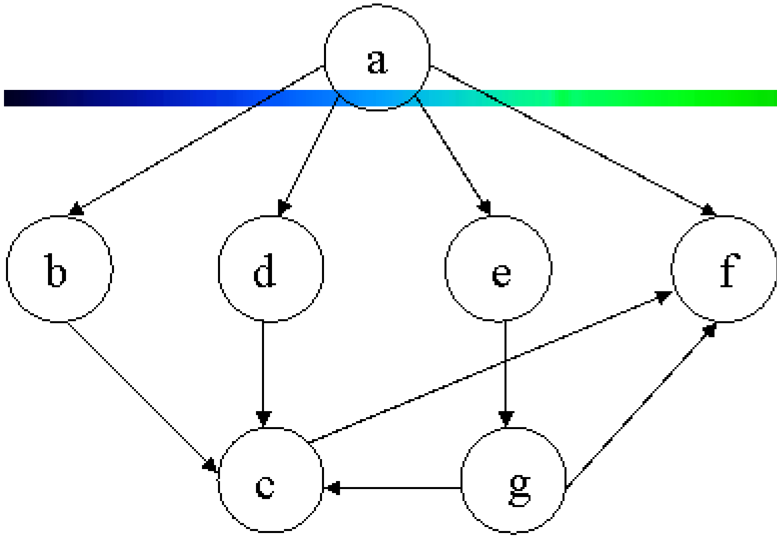
- 周游是求解图的连通性、拓扑排序和关键路径等问题的基础。

- 两类主要方式
  - 深度优先
  - 广度优先

1、深度优先搜索（DFS）

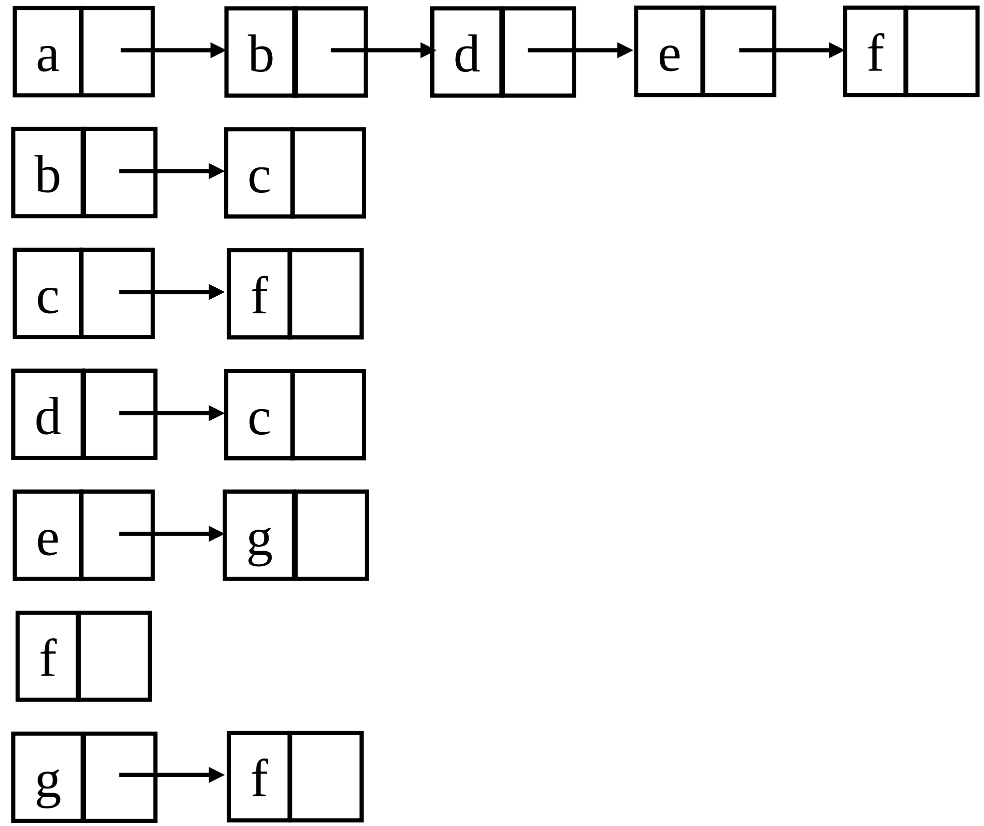
- 基本思想
  - 访问顶点V，然后访问该顶点未被访问过的邻接顶点V’
  - 从V’出发递归地按照深度优先的方式周游下去
  - 直到当一所有邻接点都被访问过的顶点U时，则回到已访问顶点序列中最后一个拥有未被访问顶点的下一邻接点W
  - 再从W出发递归地按照深度优先的方式周游
  - 最后，当任何已被访问过的顶点都没有未被访问的相邻顶点时，则周游结束
- 深度优先搜索树（Depth-First Search Tree）

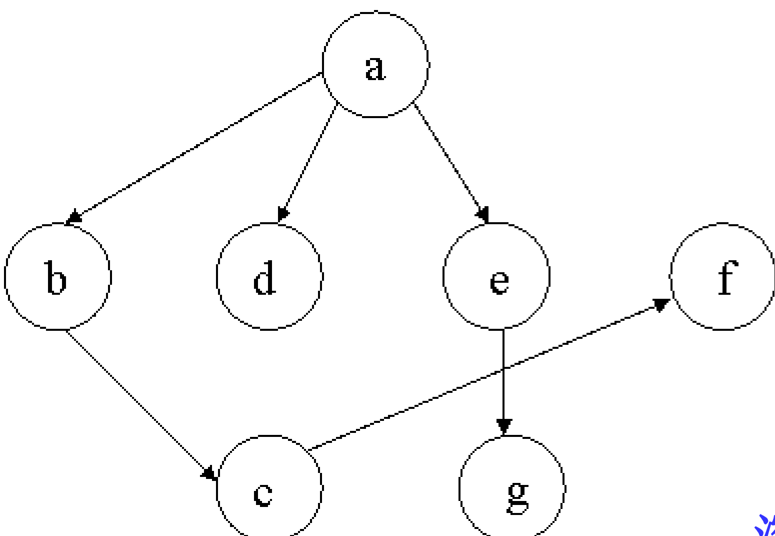
图示



(a)有向图

(b)邻接表





(c)深度优先搜索树

深度优先搜索的顺序：a，b，c，f，d，e，g

深度优先搜索算法

```
void DFS(Graph& G, int V)
{
    Visit(G, V);                          //访问V
    G.Mark[V]= VISITED;                   //访问顶点V，并标记其标志位
    for(Edge e=G. FirstEdge(V); G.IsEdge(e); e=G. NextEdge(e))
        //递归地按照深度优先的方式访问V邻接的未被访问的顶点
        if(G.Mark[G. ToVertices(e)]== UNVISITED)
            DFS(G, G. ToVertices(e));
}
```

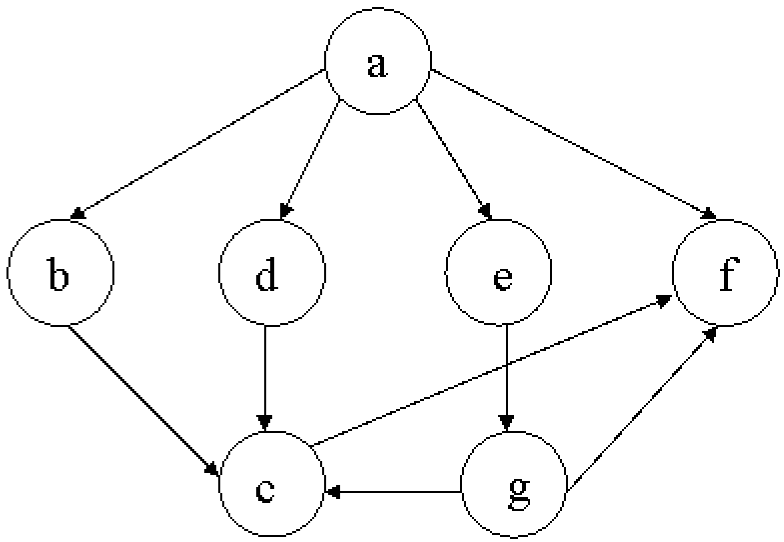
复杂性分析

- 时间复杂度
  - **DFS**对每一条边处理一次（无向图的每条边从两个方向处理），每个顶点访问一次。
  - 采用邻接表表示时，有向图总代价为$\Theta(|V|+|E|)$，无向图为$\Theta(|V|+2|E|)$
  - 采用相邻矩阵表示时，处理所有的边需要$\Theta(|V|^2)$的时间，所以总代价为$\Theta(|V|+|V|^2)= \Theta(|V|^2)$。

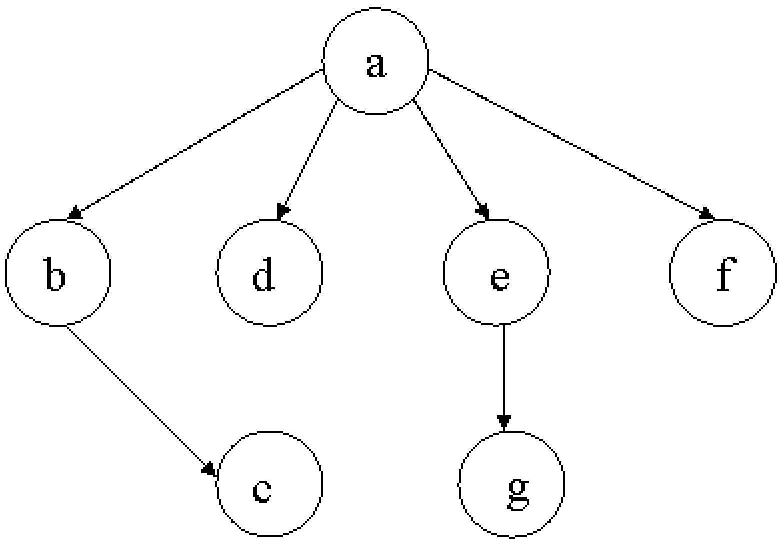
2、广度优先搜索（BFS）

- 基本思想
  - 访问顶点V_0
  - 然后访问V_0邻接到的所有未被访问过的邻居顶点V_{01}，V_{02}，…V_{0i}
  - 再依次访问V_{01}，V_{02}，…V_{0i}邻接到的所有未被访问的邻居顶点
  - 如此进行下去，直到访问遍所有的顶点。
- 广度优先搜索树（breadth-first search tree）

图示



(a)有向图



(b)广度优先搜索树

广度优先搜索的顺序：**a，b，d，e，f，c，g**

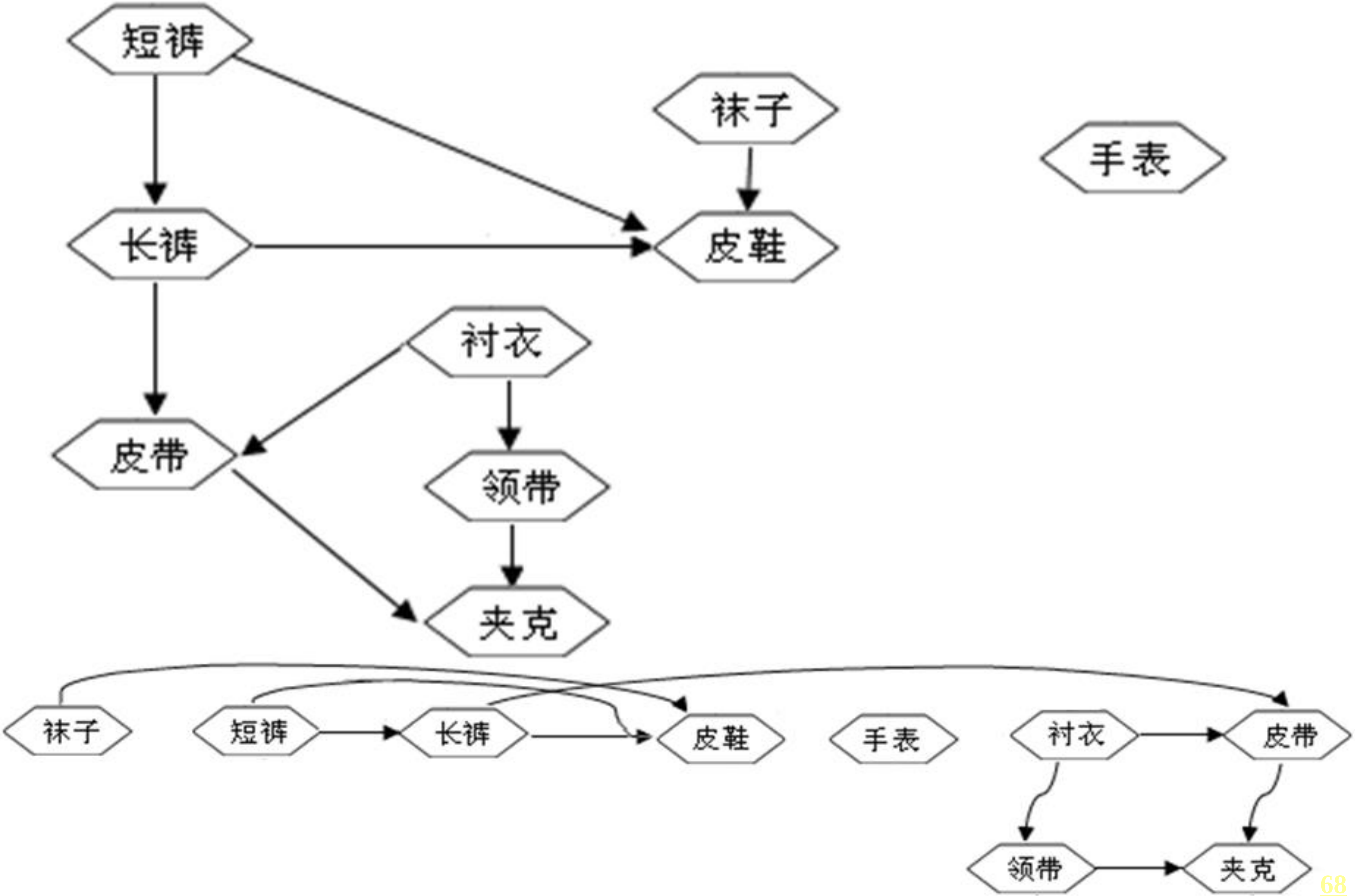
广度优先搜索算法

```
void BFS(Graph& G, int V){
    using std::queue;                    //初始化广度优先周游要用到的队列
    queue<int> Q;
    G.Mark[V]= VISITED;                  //访问顶点V，并标记其标志位， V入队
    Visit(G, V); Q.push(V);
    while(!Q.empty()) {                  //如果队列仍然有元素
        int V=Q.front(); Q.pop();   //取顶部元素，并出队
        //将与该点相邻的每一个未访问的顶点都入队
        for(Edge e=G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e)){
            if(G.Mark[G.ToVertex(e)]== UNVISITED) {
                G.Mark[G.ToVertex(e)]=VISITED;
                Visit(G, G.ToVertex(e));
                Q.push(G.ToVertex(e)); //入队
            }
        }
    }
}
```

复杂度分析

- 广度优先搜索实质上与深度优先相同，只是访问顺序不同而已。
- 二者时间复杂度也相同

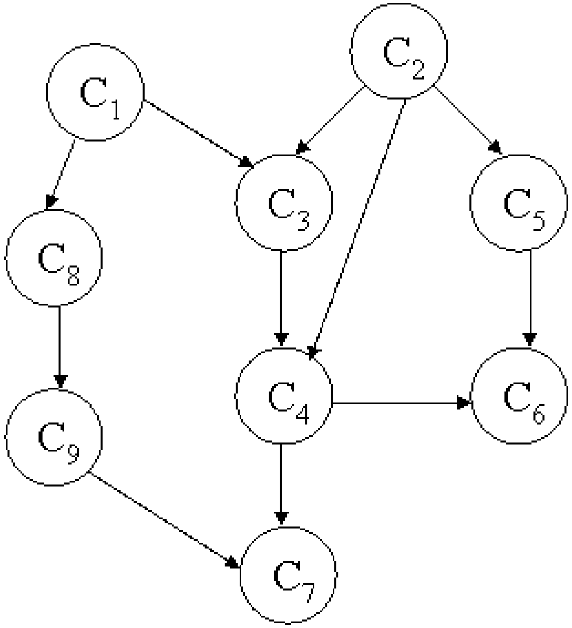
3、拓扑排序



➢ 问题定义

- 先决条件：是指以某种线性顺序来组织多项任务，以便能够在满足先决条件的情况下逐个完成各项任务
- 有向无环图能够模拟先决条件

先修课程	课程代号	课程名称
	高等数学	C1
	程序设计	C2
C1，C2	离散数学	C3
C2，C3	数据结构	C4
C2	算法语言	C5
C4，C5	编译技术	C6
C4，C9	操作系统	C7
C1	普通物理	C8
C8	计算机原理	C9



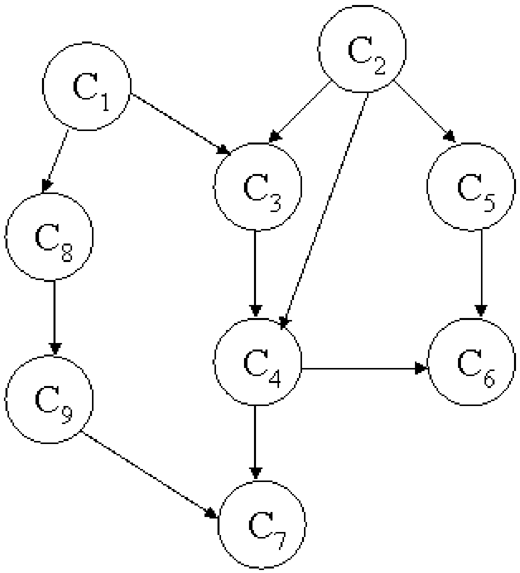
学生课程安排图

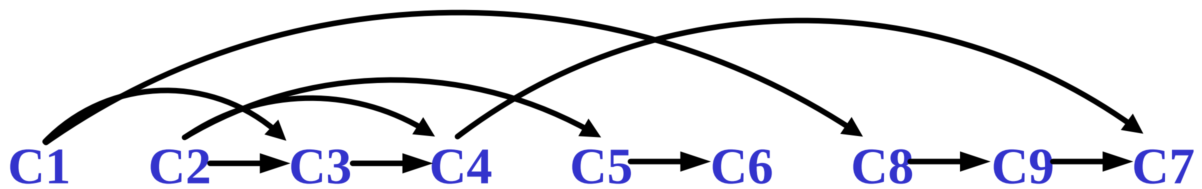
➢ 拓扑排序（topological sort）

- 将一个**有向无环图**中所有顶点在不违反**先决条件关系**的前提下排成线性序列的过程称为拓扑排序
- 对一个有向无环图**G**进行拓扑排序，是将**G**中所有顶点排成一个线性序列，使得图中任意一对顶点**u**和**v**，若<**u**，**v**>∈**E(G)**，则**u**在线性序列中出现在**v**之前
- 拓扑排序形成的序列称作拓扑序列

性质1

- 若将图中顶点按拓扑次序排成一行，则图中所有的有向边均是从左指向右的





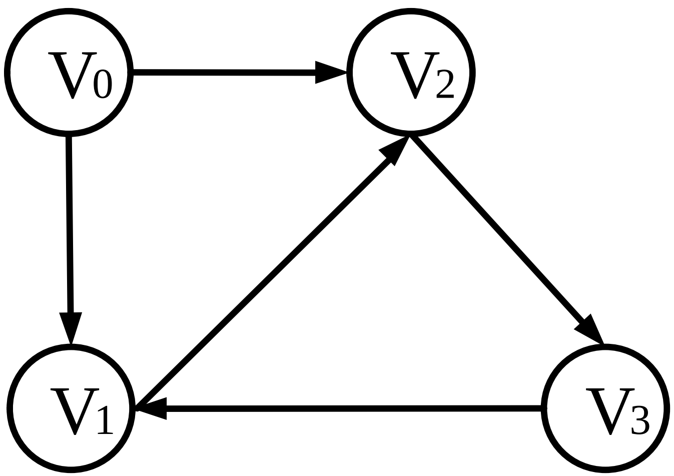
- **拓扑序列不唯一**
  - 例如，上图至少可以有如下两个拓扑序列：<u>C1C2C3C4C5C6C8C9C7</u> 和<u>C1C8C9C2C5C3C4C7C6</u>

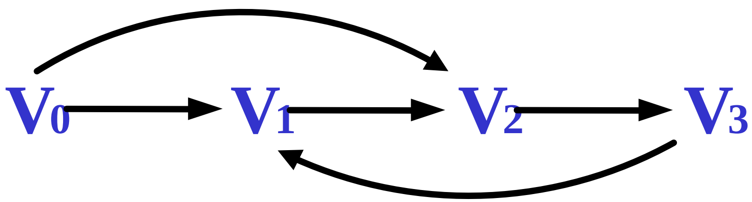
思考题

给定一有向无环图（DAG），请输出不同拓扑排序的个数！

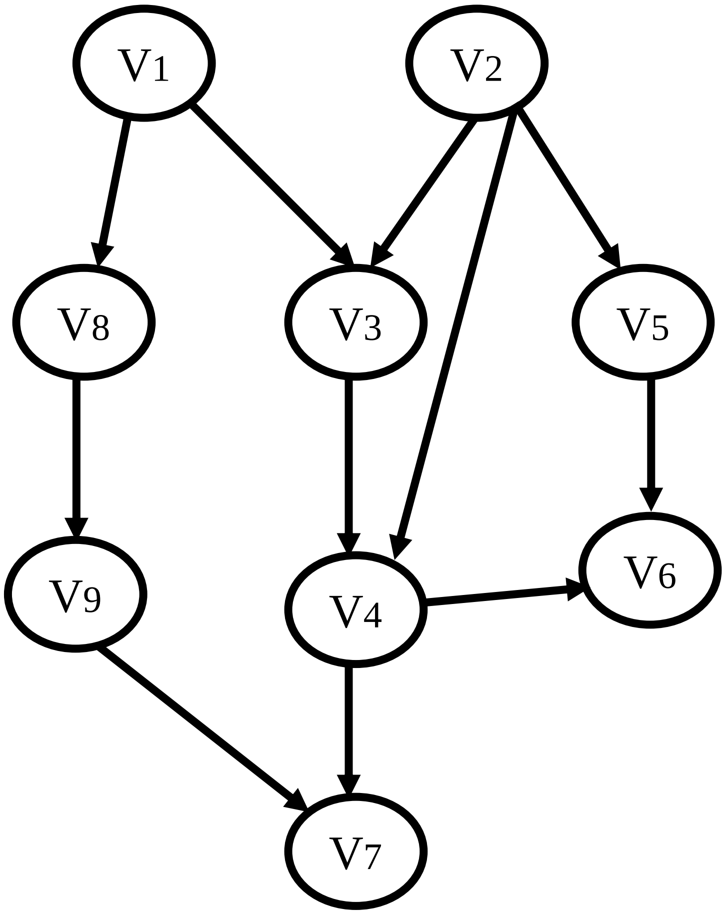
性质2

- 环存在时不存在拓扑序列





拓扑排序的基本思想



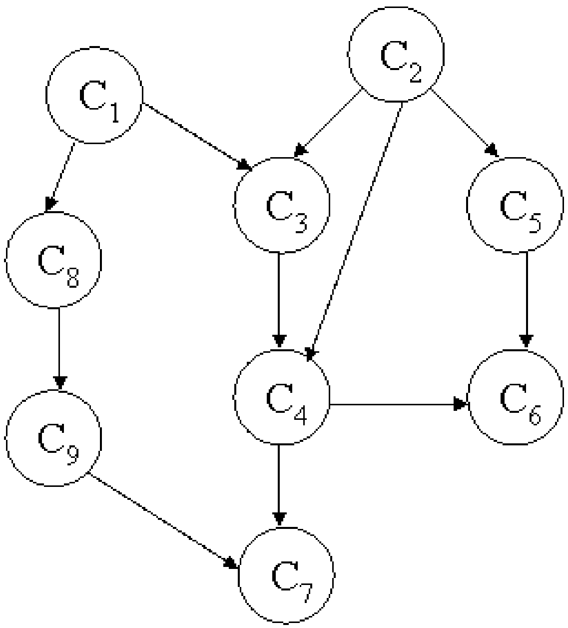
- 限定是有向无环图
- 拓扑排序方法
  - 从图中选择一入度为0的顶点并输出
  - 从图中删掉此顶点及其所有的出边（出边关联顶点的入度减1）
  - 回到第（1）步继续执行。
- 环路存在时
  - 排序结束，仍有顶点没有被输出
  - 但在剩下的图中找不到入度为0的顶点

拓扑排序算法

- 基于邻接矩阵的实现

- 基于邻接表的实现
  - 广度优先排序(BFS-TopSort)
  - 深度优先排序(DFS-TopSort)

基于邻接矩阵表示的拓扑排序



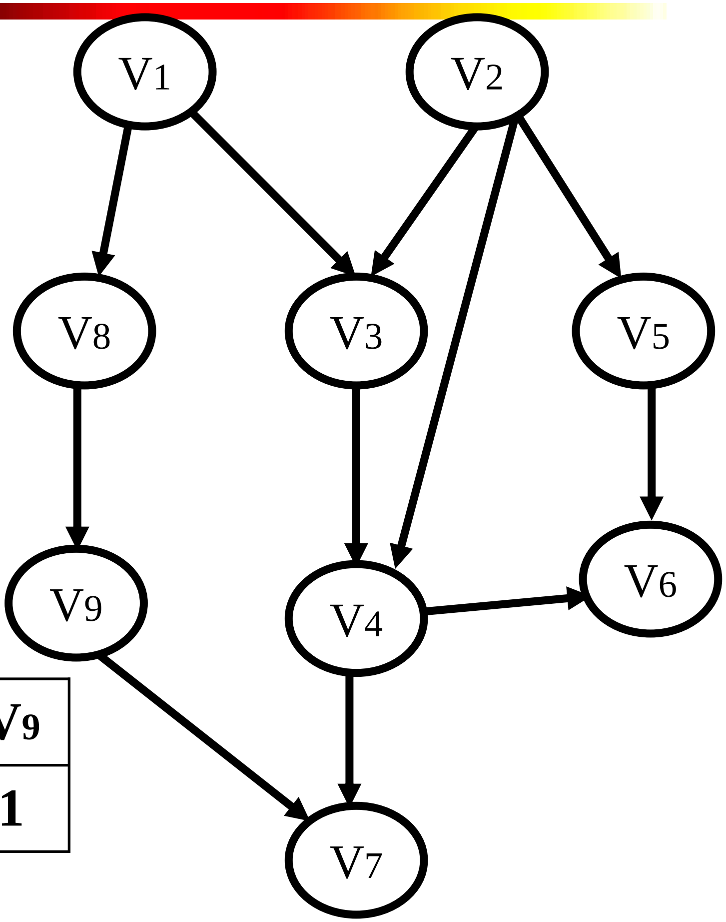
新序号	1	2	3	4	5	6	9	7	8
行/列号	1	2	3	4	5	6	7	8	9
1	0	0	1	0	0	0	0	1	0
2	0	0	1	1	1	0	0	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	1	1	0	0
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	1	0	0

图G的邻接矩阵A

- G中入度为0的是A中全0的列
- 删除某个顶点的出边就是把邻接矩阵中对应的行清0

基于邻接表的拓扑排序：BFS方法

- **BFS-TopSort**
  - 为每个顶点设置一个表示该结点入度字段(indegree)，入度表
    - 不用检查n*n的矩阵
    - 直接检查数组就可确定入度为0的顶点



入度表

顶点	V1	V2	V3	V4	V5	V6	V7	V8	V9
入度	0	0	2	2	1	2	2	1	1

队列

BFS-TopSort算法

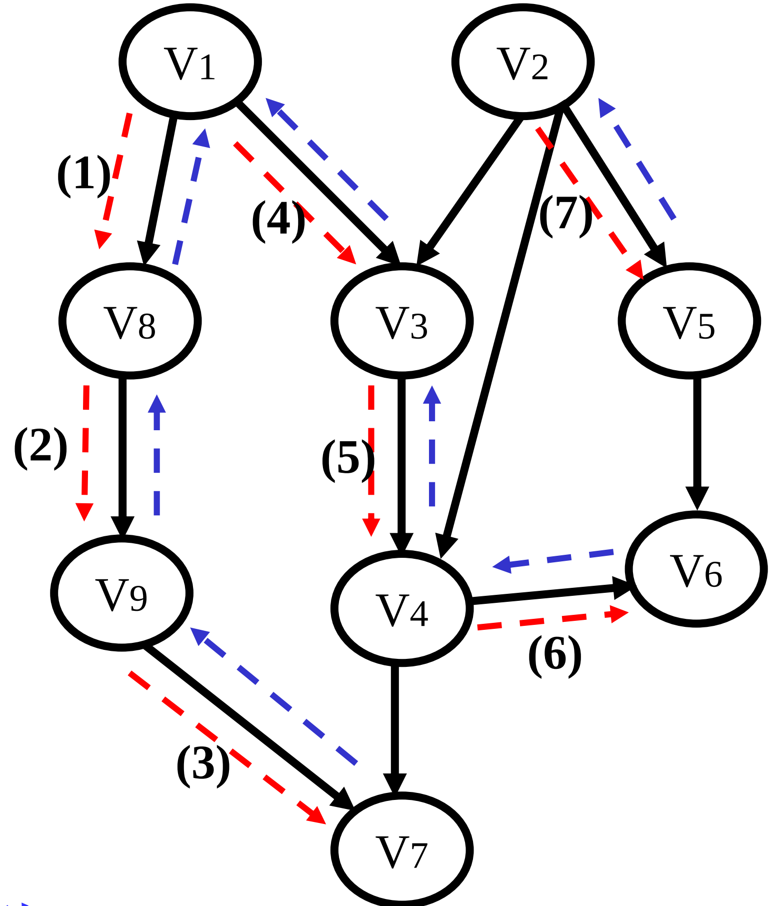
```
void TopsortbyQueue(Graph& G)  {                    //队列方式实现的拓扑排序
    for(int i=0;i<G.VerticesNum();i++)
            G.Mark[i]=UNVISITED;                    //初始化标记数组
    using std::queue;
    queue<int> Q;                                   //初始化队列
    for(i=0; i<G.VerticesNum(); i++)  {             //图中入度为0的顶点入队
        if(G.Indegree[i]==0)
            Q.enqueue(i);
    }
    while(!Q.empty()){                              //如果队列中还有图的顶点
            V=Q.dequeue();                          //一个顶点出队
            Visit(G, V);                            //访问该顶点
            G.Mark[V]=VISITED;
```

```
        //边e的终点的入度值减1
        for(Edge e= G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e))  {
                G.Indegree[G.ToVertex(e)]--;
                if(G.Indegree[G.ToVertex(e)]==0)
                        Q.enqueue(G.ToVertex(e)); //入度为0的顶点入队
        }//end for
    }//end while
    for(i=0; i<G.VerticesNum(); i++)  {
        if (G.Mark[i]==UNVISITED){
            Print("图有环");                    //图有环
            break;
        }
    }
}   //注：在有环的情况下会提前退出，从而可能没处理完所有的边和顶点
```

广度优先排序可以判定有环存在~~

基于邻接表的拓扑排序：DFS

- **DFS-TopSort**
  - 栈的使用
  - 逆序序列



逆序队列

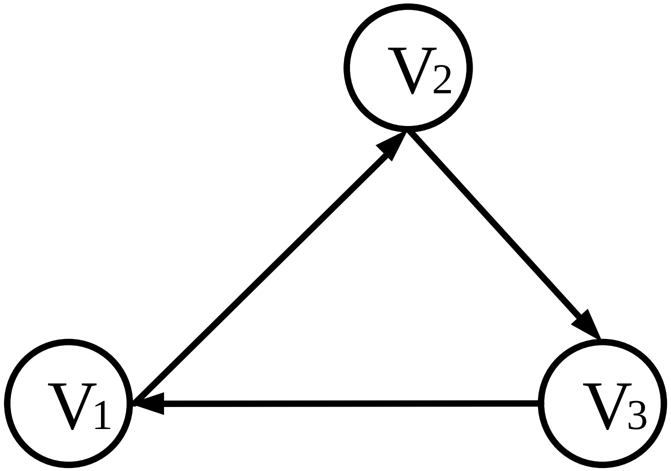
DFS-TopSort算法

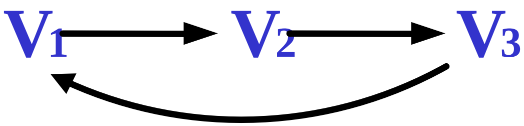
```
void TopsortbyDFS(Graph& G){                    //深度优先拓扑排序,结果逆序
    for(int i=0; i<G.VerticesNum(); i++)        //初始化标志位
        G.Mark[i]=UNVISITED;
    int *result=new int[G.VerticesNum()];       //最终输出的逆序结果
    int tag=0;
    for(i=0; i<G.VerticesNum(); i++)            //对图的所有顶点进行处理
        if(G.Mark[i]== UNVISITED)
            Do_topsort(G,i,result,tag);         //调用递归函数
    for(i=G.VerticesNum()-1;i>=0;i--) {         //逆序输出
        Visit(G, result[i]);
    }
}
```

```
//深度优先搜索实现的拓扑排序
void Do_topsort(Graph& G, int V, int *result, int& tag){
    G.Mark[V]= VISITED;
    //访问V邻接到的所有未被访问过的顶点
    for(Edge e= G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e))
        if(G.Mark[G.ToVertex(e)]== UNVISITED)
            Do_topsort(G, G.ToVertex(e), result, tag); //递归调用
    result[tag++]=V;
}
```

环的判断

- **深度优先拓扑排序不能判断环的存在**





思考题

如何判断深度优先拓扑排序环的存在？

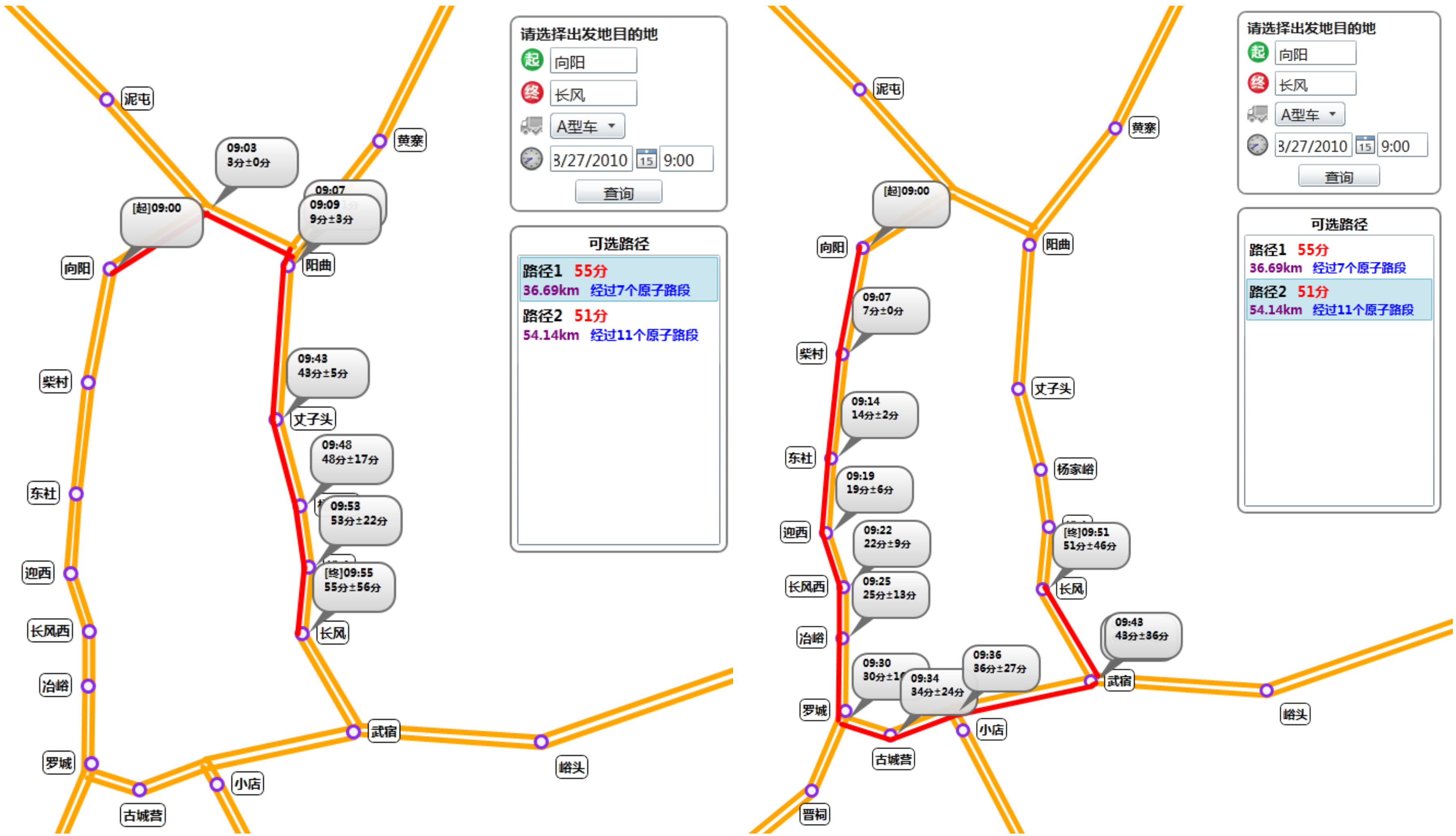
复杂性分析

- 拓扑排序的时间复杂度

  - 采用相邻矩阵时，每次算法需要找所有入度为0的顶点，需要 $\Theta(|V|^2)$的时间，那么对|V|个顶点而言，总代价为$\Theta(|V|^3)$

  - 当采用邻接表时，因为在顶点表的每个顶点中可以有一个字段来存储入度，所以只需$\Theta(|V|)$的时间，加上处理边、顶点的时间，总代价为$\Theta(2|V|+|E|)$

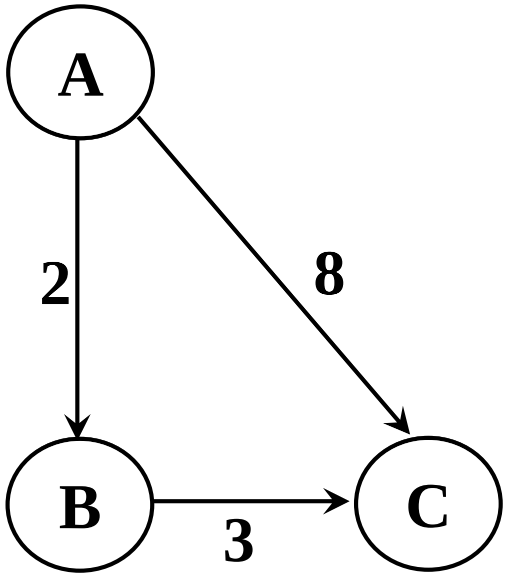
7.5 最短路径问题



请选择出发地目的地
向阳
长风
A型车
3/27/2010
9:00
查询
可选路径
路径1 55分
36.69km 经过7个原子路段
路径2 51分
54.14km 经过11个原子路段
泥屯
黄寨
向阳
阳曲
柴村
丈子头
东社
迎西
长风西
长风
冶峪
罗城
古城营
小店
武宿
峪头
晋祠
杨家峪
[起]09:00
09:03
3分±0分
09:09
9分±3分
09:43
43分±5分
09:48
48分±17分
09:53
53分±22分
[终]09:55
55分±56分
09:07
7分±0分
09:14
14分±2分
09:19
19分±6分
09:22
22分±9分
09:25
25分±13分
09:34
34分±24分
09:36
36分±27分
09:43
43分±36分
[终]09:51
51分±46分

问题定义

- **带权图的最短路径问题**
  - 即求两个顶点间长度最短的路径
    - – 路径上各边权值的总和
- 管线铺设， 出行线路选择等应用



带权图

- **广度优先遍历本质上就是单位权重图的最短路径搜索问题**

最短路径问题求解分类

- **单源最短路径**
  - 图G中，给定源点s∈V，找出s到图中其它各顶点的最短路径
  - 代表性算法：Dijkstra算法（贪心思路）
- **每对顶点间的最短路径**
  - 图G中，任意的顶点V_i，V_j∈V，找出从V_i到V_j的最短路径
  - 代表性算法：Floyd算法（动态规划思路）

Dijkstra（迪杰斯特拉）

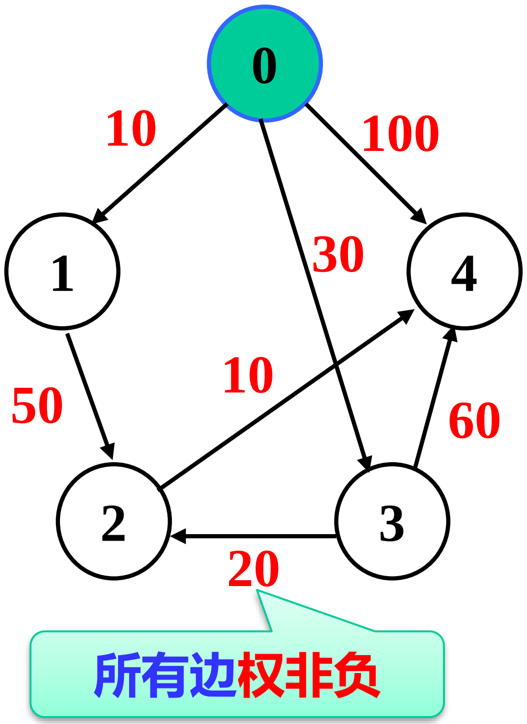
- Edsger Wybe Dijkstra，荷兰人
- 20世纪最伟大的计算机科学家之一，曾获1972年图灵奖，与D. E. Knuth并称为20世纪最伟大的计算机科学家
- 主要贡献有如下几个方面：
  - 1)提出“goto有害论”；
  - 2)解决了有趣的“哲学家聚餐”问题；
  - 3)最短路径算法(SPF)的创造者；
  - 4)第一个Algol 60编译器的设计者和实现者；
  - 5)THE操作系统的设计者和开发者。

(1930/5/11~2002/8/6)

1、Dijkstra算法

- 1959年提出，是**边权非负**情况下的经典算法
- 基本思想
  - 每次从距离**已生成最短路径的节点集**“**一步之遥**”的节点中，选择距离原点V_0最近的边进行延伸
  - 结果由近及远生成以起始点V_0为根的有向树。
- 是一类贪心算法

路径长度递增序



图G中从源点0到其余各点的最短路径

源点	中间结点	终点	路径长度
0		1	10
0		3	30
0	3	2	50
0	3，2	4	60

- 按<u>路径长度递增序</u>产生各顶点最短路径
  - 若按长度递增的次序生成从源点s到其它顶点的最短路径，则当前正在生成的最短路径上除终点以外，其余顶点的最短路径均已生成

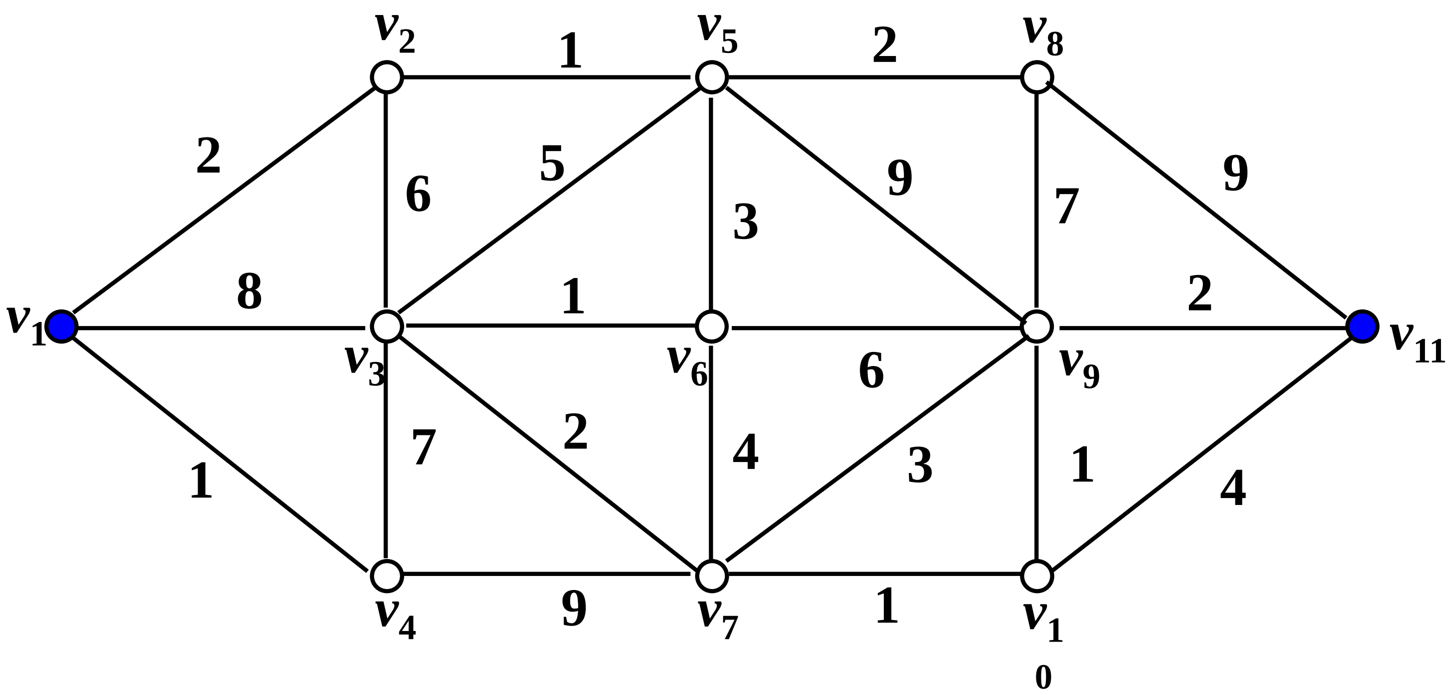
实现策略

- 把图中顶点分成两组
  - 第一组：已确定最短路径的顶点
  - 第二组：尚未确定最短路径的顶点
- 按最短路径长度递增顺序逐个把第二组的顶点加到第一组中
  - 直至从s出发可以到达的所有顶点都包括进第一组
- 在合并过程中，保持s到第一组各顶点的最短路径长度都不大于从s到第二组各顶点的最短路径长度
  - 第一组顶点对应的距离值：从s到该顶点的最短路径长度
  - 第二组顶点对应的距离值：从s到该顶点的值包括以第一组中顶点为中间顶点的最短路径长度

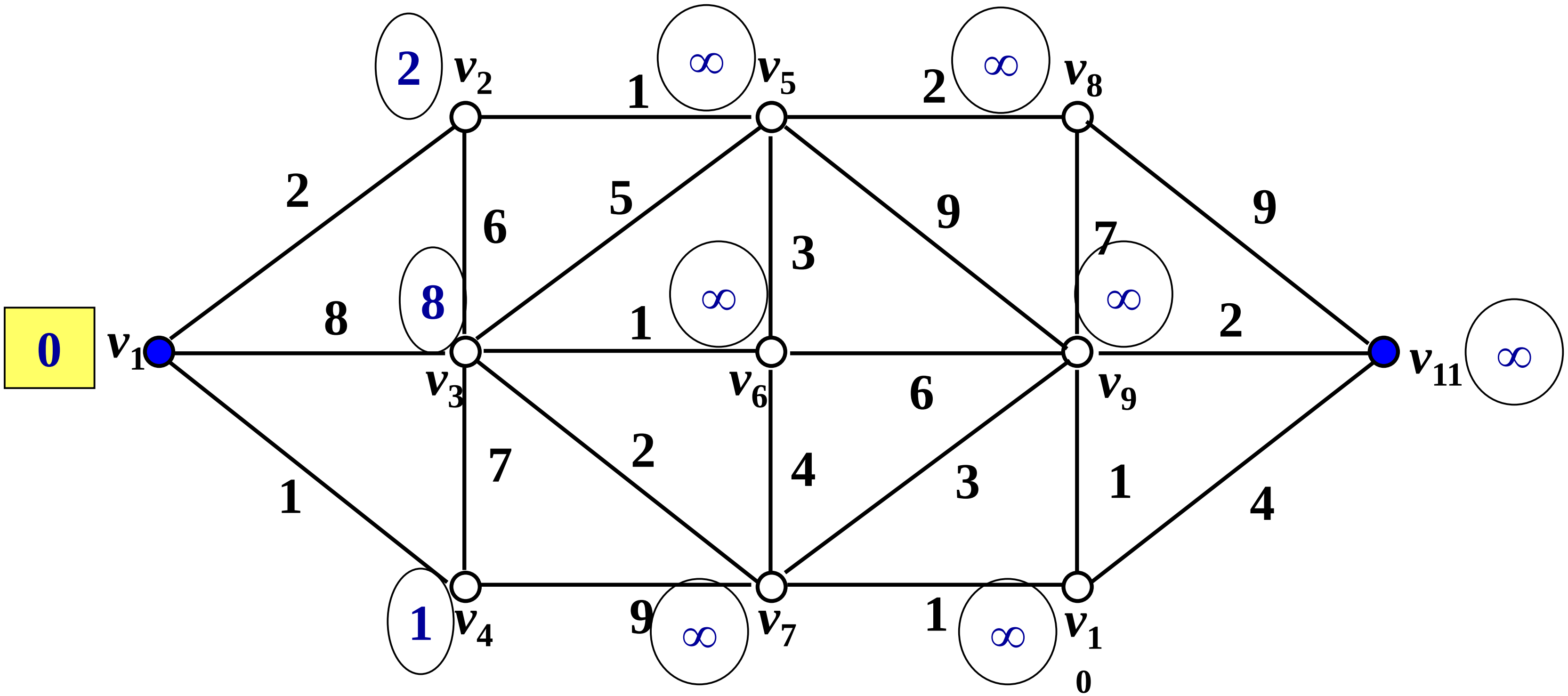
具体过程

- 初始化：第一组只包括源点s，第二组包括其它所有顶点
  - s距离值为0，第二组顶点的距离值确定如下：
    - 若有边<s，V_i>或（s，V_i），则Vi的距离值为边所带的权，否则为∞
- 过程：每次从第二组的顶点中选一个其距离值为最小的顶点V_m加入到第一组中
  - 每往第一组加入顶点V_m，要对第二组各顶点的距离值进行一次修正
    - 若加进V_m做中间顶点，使从s到V_i的最短路径比不加V_m的短，则需要修改V_i的距离值
  - 修改后再选距离值最小的顶点加入到第一组中，重复上述过程
  - 结束条件：直到图的所有顶点都包括在第一组中或者再也没有可加入到第一组的顶点存在

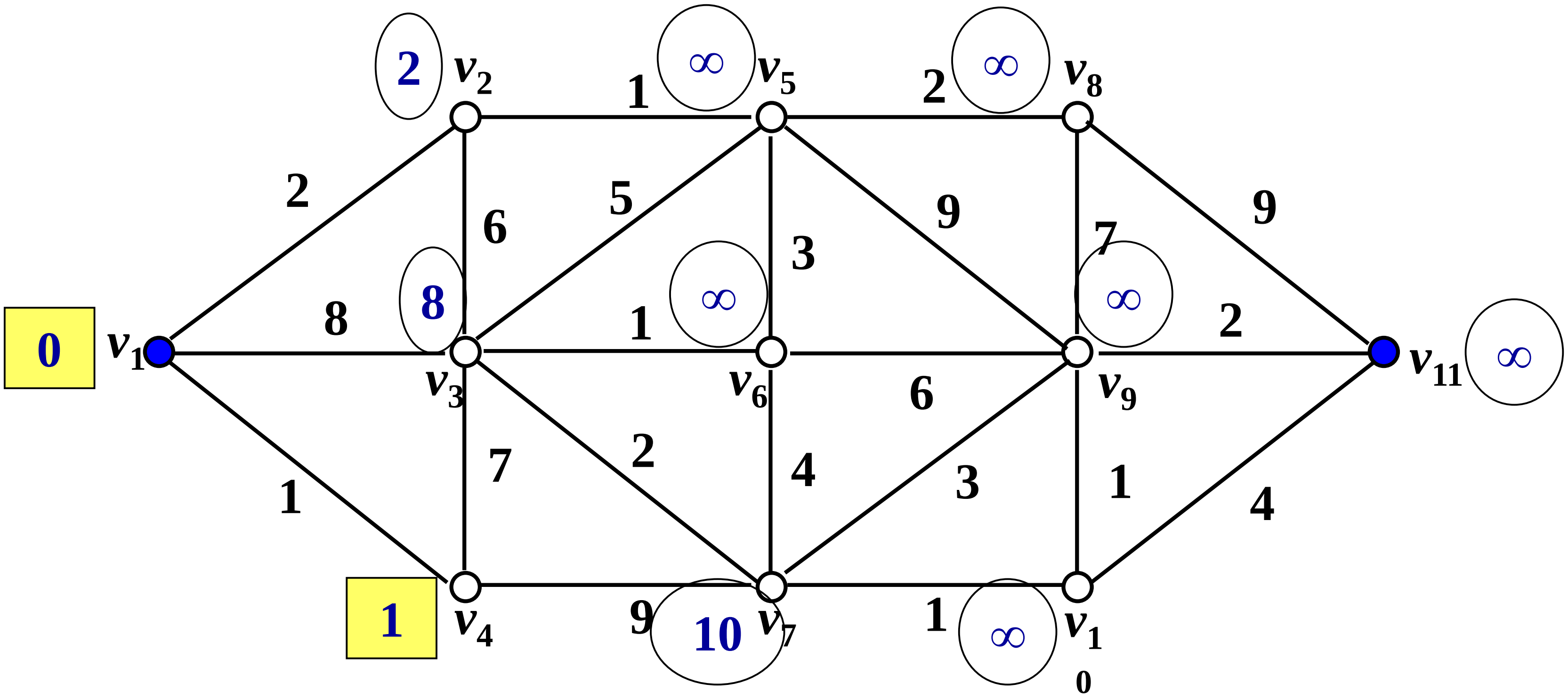
示例



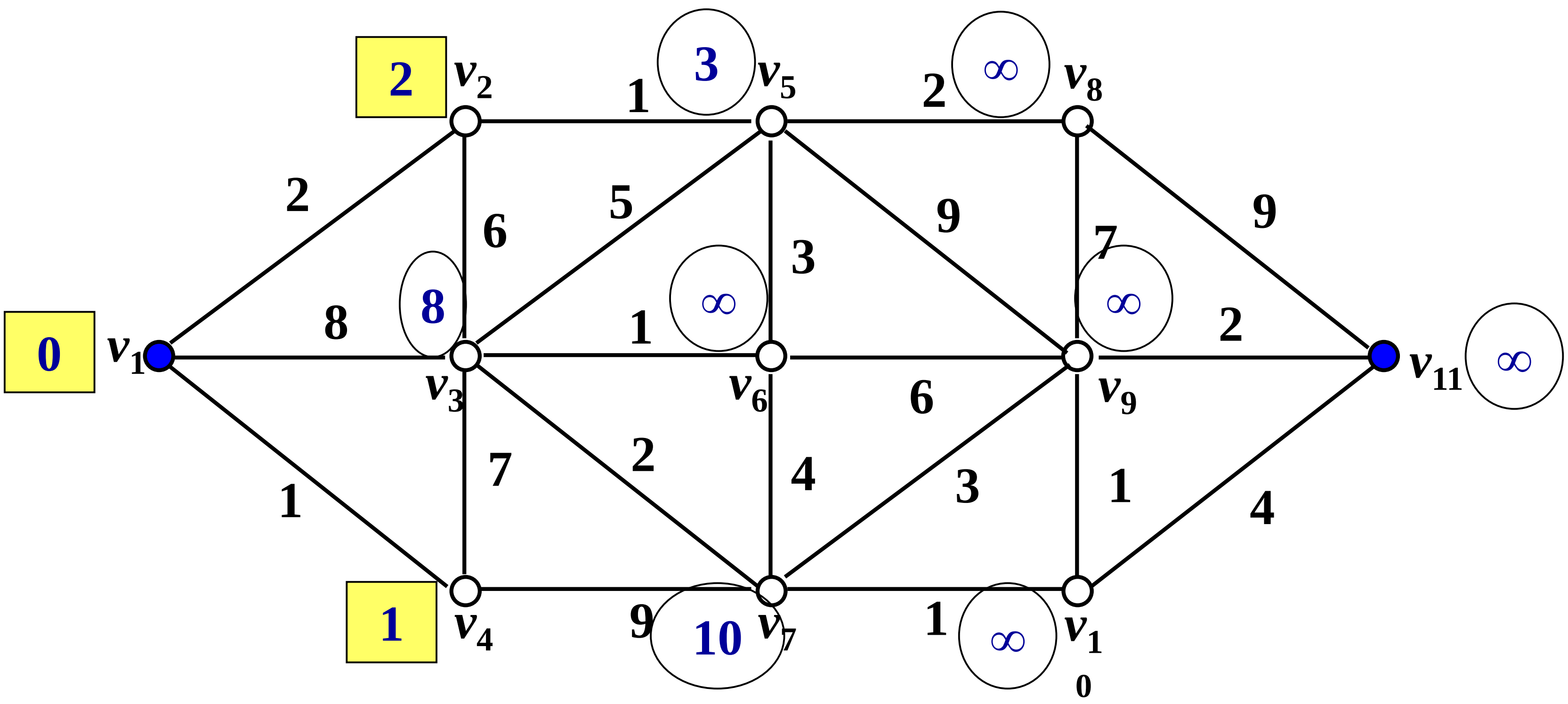
示例


0
v_1
2
v_2
1
∞
v_5
2
∞
v_8
2
6
5
9
9
3
7
8
8
1
∞
∞
2
v_{11}
∞
v_3
v_6
6
v_9
1
7
2
4
3
1
4
1
v_4
9
∞
v_7
1
∞
v_{10}

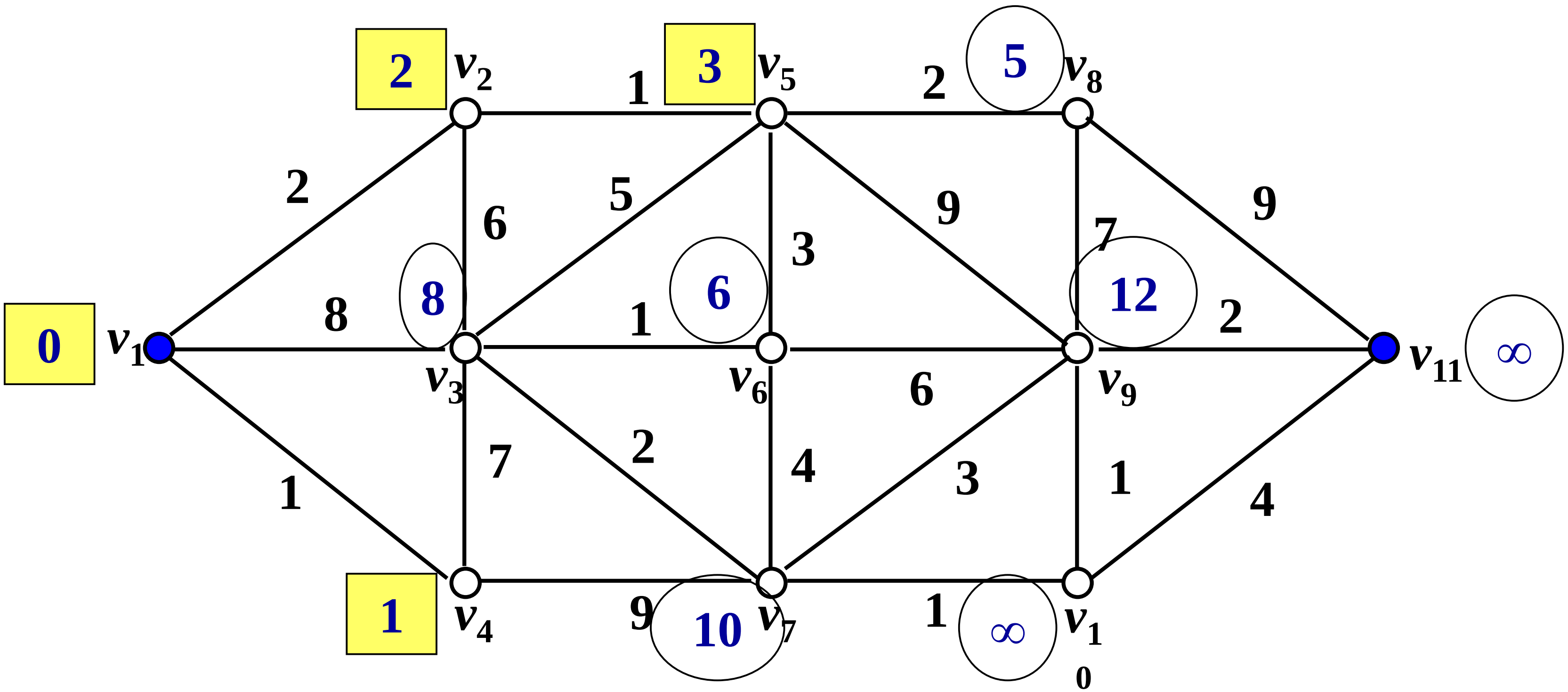
示例


0
v1
2
v2
8
v3
1
v4
1
v5
v6
10
v7
2
v8
v9
v10
v11
∞
1
2
9
9
6
5
3
7
2
8
1
7
2
4
6
3
1
9
1
4
2

示例

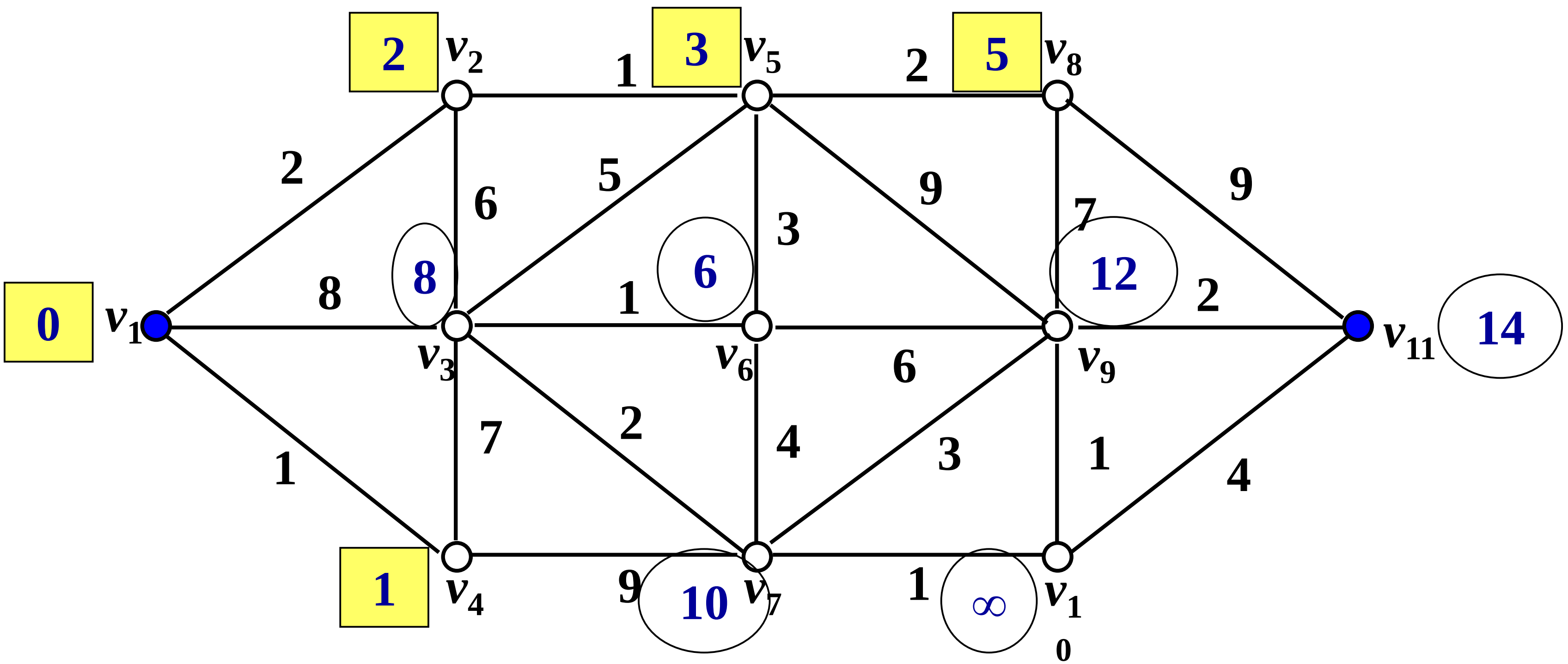

0
v1
2
v2
1
v4
8
3
v5
10
v7
∞
v8
∞
v6
∞
v9
∞
v10
∞
v11
2
1
8
6
7
1
5
1
2
9
3
4
9
2
7
6
3
1
1
9
2
4
v3

示例



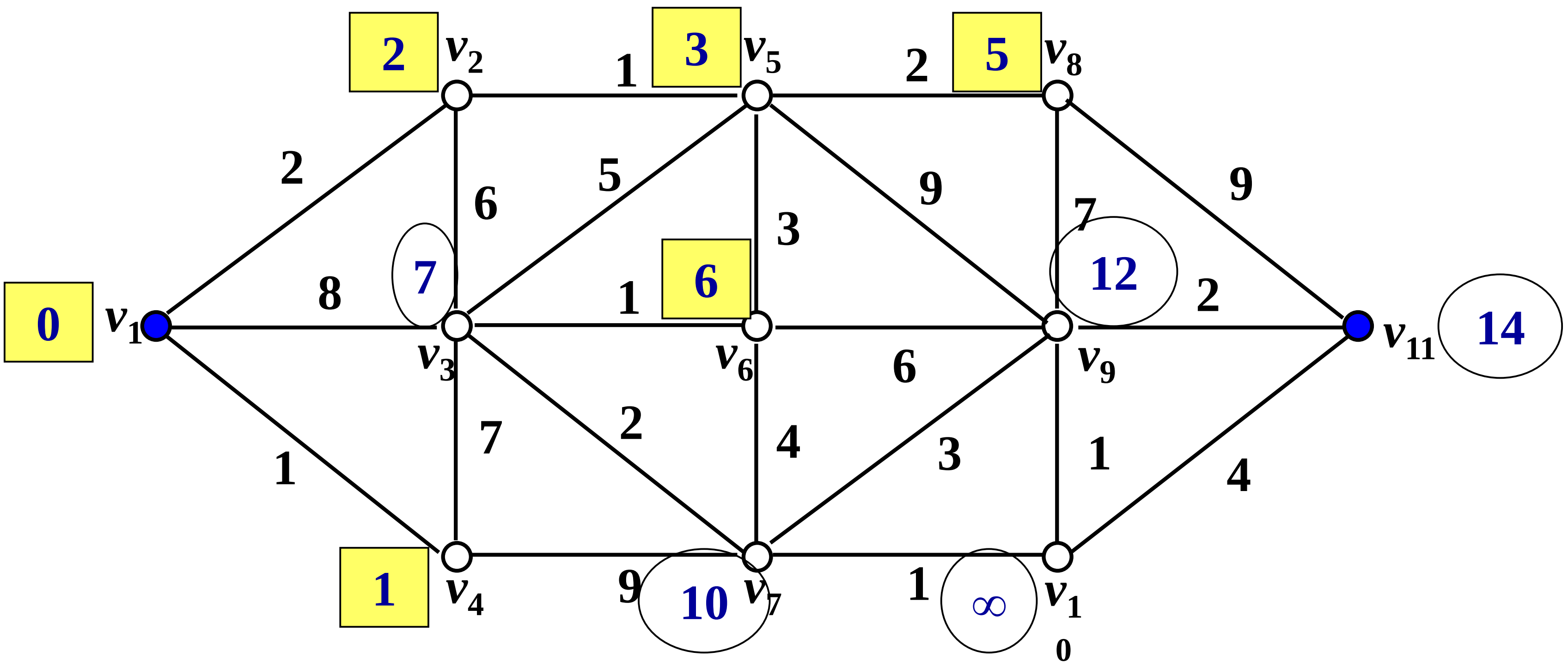
2
v2
1
3
v5
2
5
v8
2
6
5
3
9
7
9
8
6
12
0
v1
8
1
2
v11
∞
v3
v6
6
v9
7
2
4
3
1
1
4
1
v4
9
10
v7
1
∞
v10

示例



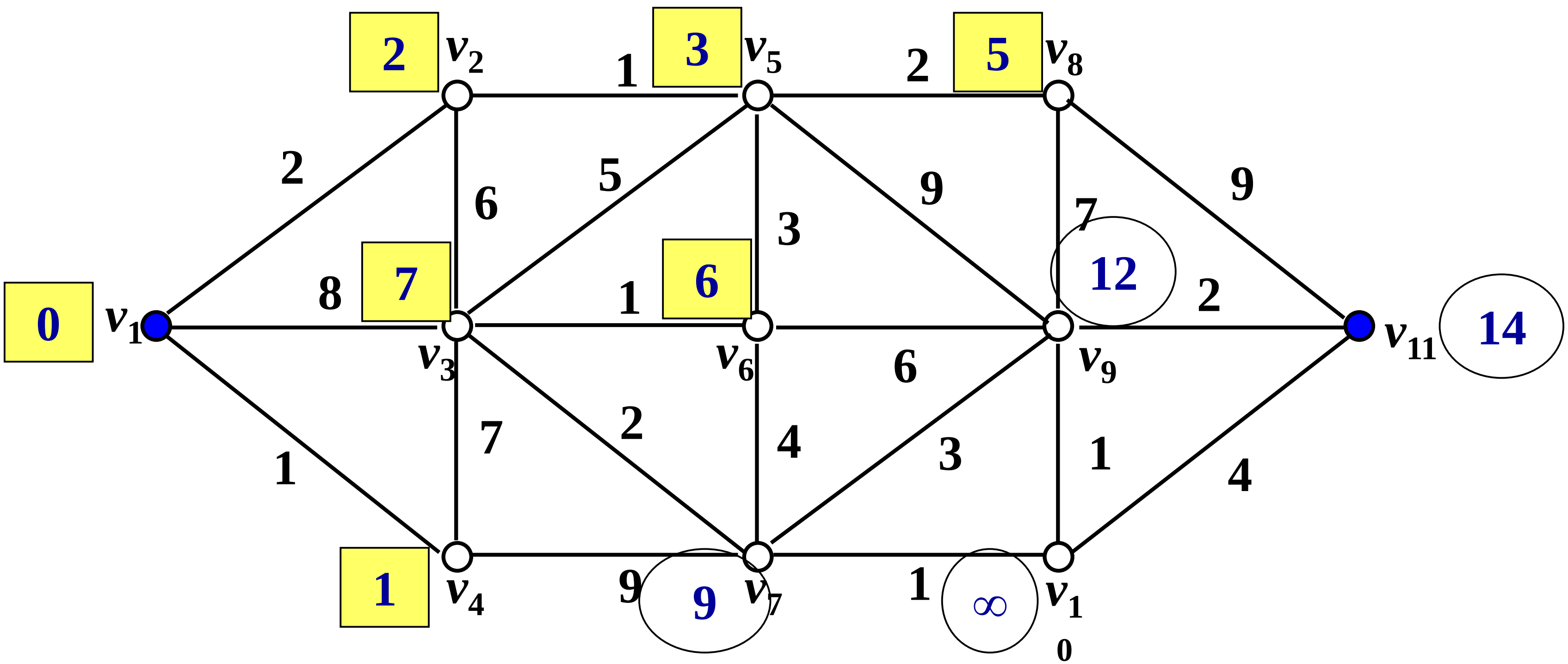
2
v2
1
3
v5
2
5
v8
2
6
5
3
9
7
9
0
v1
8
8
1
6
12
2
14
v11
v3
v6
6
v9
7
2
4
3
1
1
4
1
v4
9
10
v7
1
∞
v10

示例



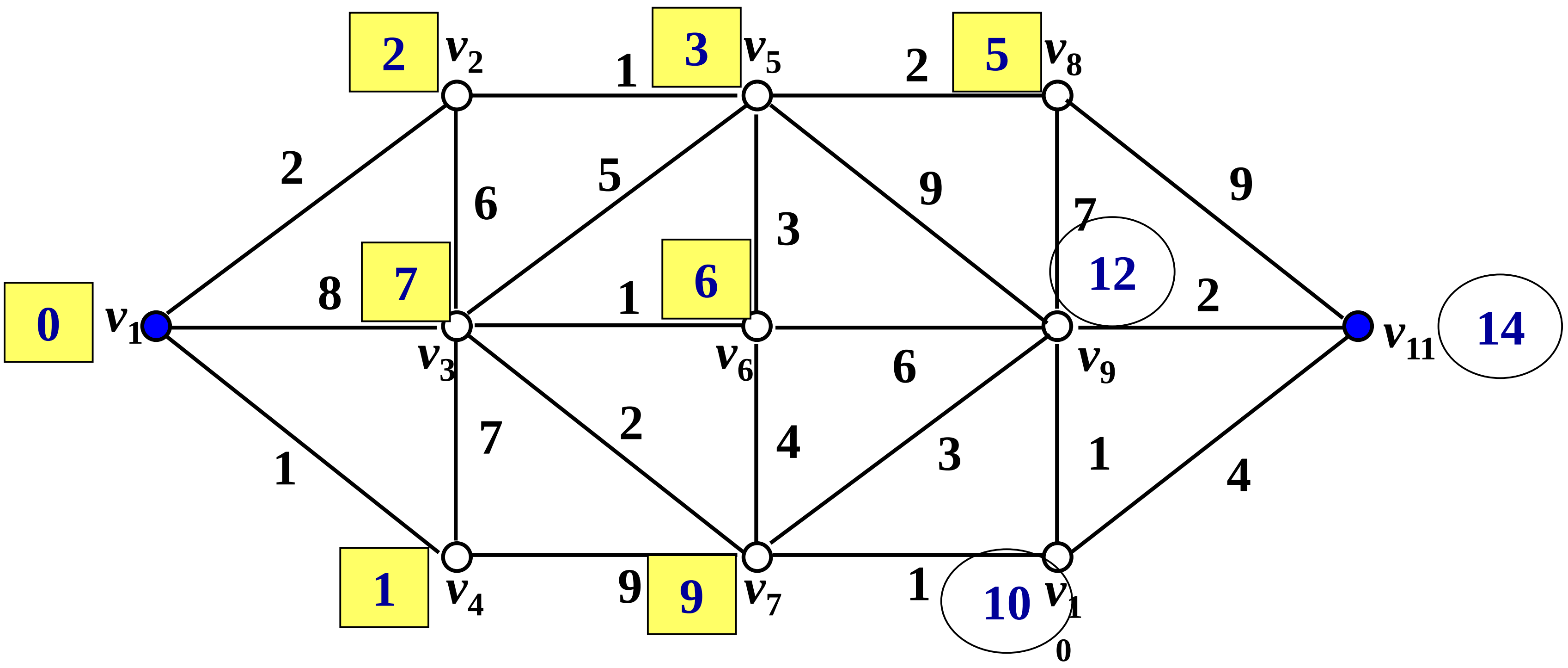
2
v_2
1
3
v_5
2
5
v_8
2
6
5
9
3
7
9
0
v_1
8
7
1
6
12
2
v_3
v_6
6
v_9
v_{11}
14
1
7
2
4
3
1
4
1
v_4
9
10
v_7
1
∞
v_{10}

示例



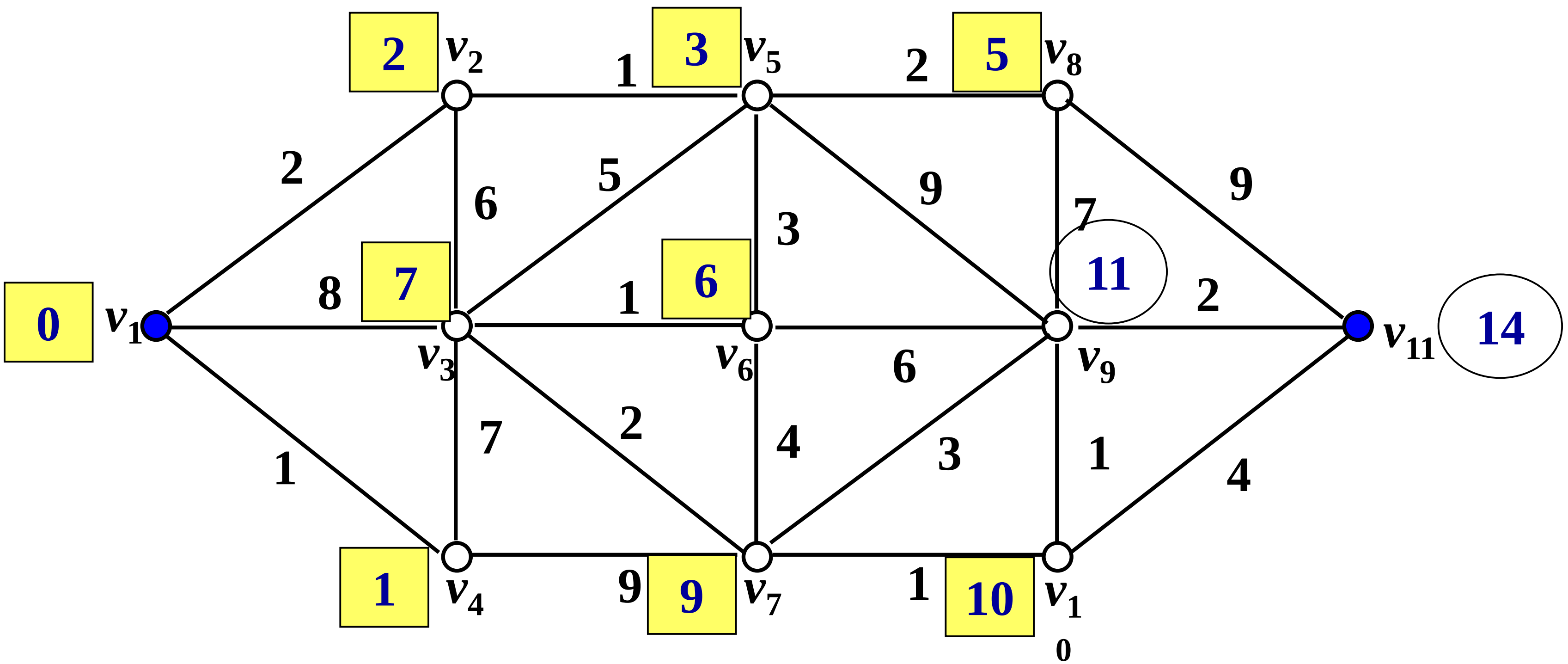
0
v1
2
v2
7
v3
1
v4
3
v5
6
v6
9
v7
5
v8
12
v9
∞
v10
14
v11
2
8
1
6
7
1
5
1
2
9
3
4
2
9
6
3
7
1
1
9
2
4

示例



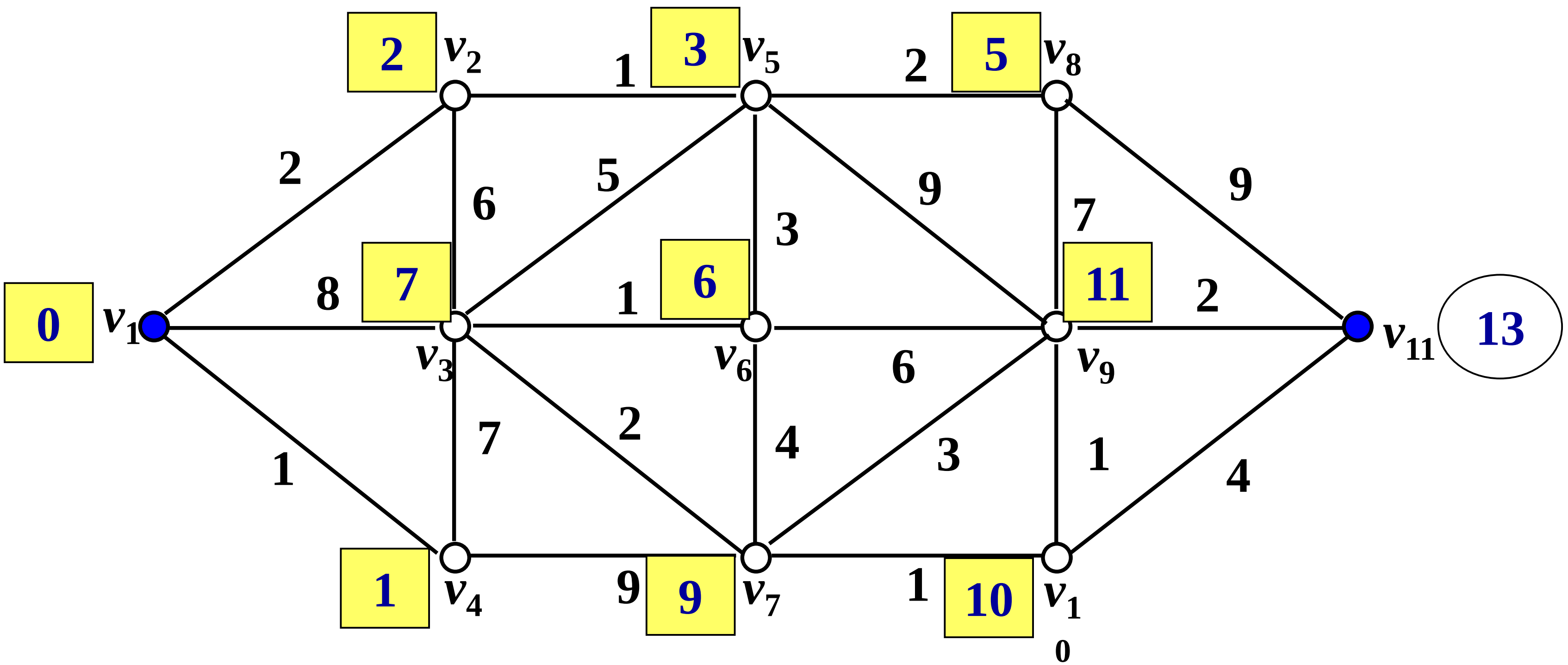
0
v1
2
v2
7
v3
1
v4
3
v5
6
v6
9
v7
5
v8
12
v9
10
v10
14
v11
2
1
8
6
7
1
5
1
2
9
2
3
4
9
6
3
1
7
9
2
1
4

示例



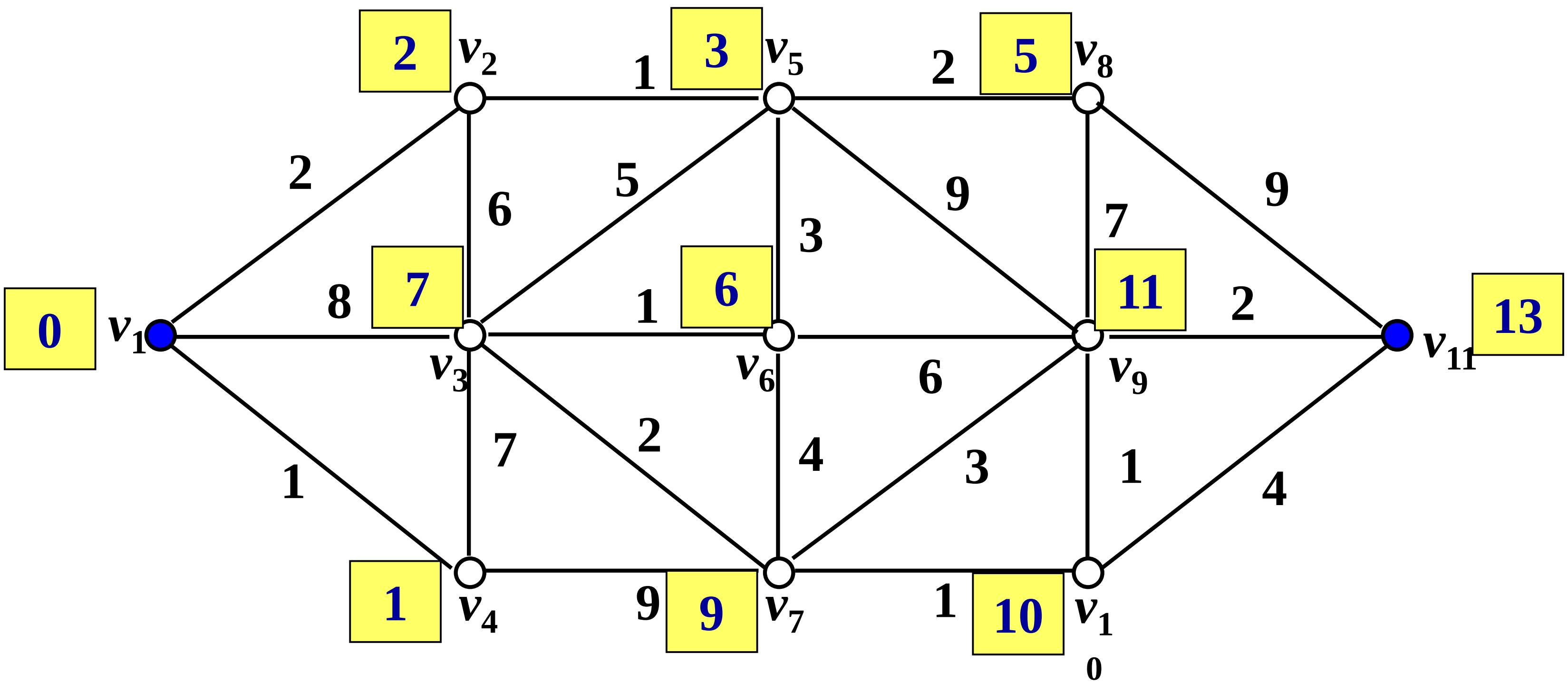
0
v1
2
v2
7
v3
1
v4
3
v5
6
v6
9
v7
5
v8
11
v9
10
v10
14
v11
2
8
1
6
7
1
5
2
9
1
3
4
9
2
6
3
7
1
9
2
4

示例



0
2
7
1
3
6
9
5
11
10
13
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
v_9
v_{10}
v_{11}
2
8
1
6
7
1
5
1
2
9
3
4
2
9
6
3
7
1
1
9
2
4

示例

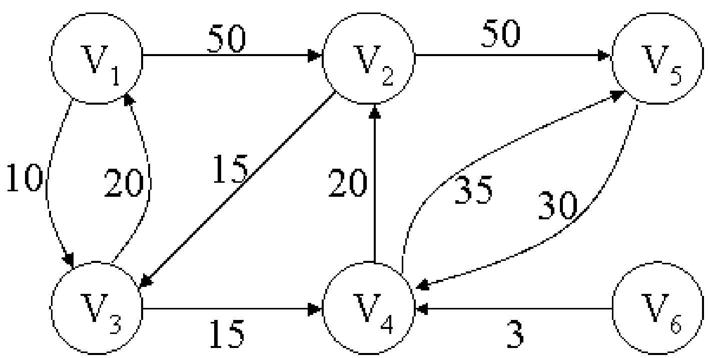


至此，从v1到其他所有点的路径就全部求出来了。

举例

	V_1	V_2	V_3	V_4	V_5	V_6
初始状态	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1
V_1进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1
V_3进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length:25 pre:3	length:∞ pre:1	length:∞ pre:1
V_4进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1
V_2进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1
V_5进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1

最短路径的表示方法



借助一个长度为N的数组，包括：

1）源点到当前节点的路径长度（length）

2）当前节点的前驱节点（pre）

Dijkstra算法描述

```
class Dist  {        // Dist类，Dijkstra和Floyd算法用于保存最短路径信息
public:
    int index;       // 顶点的索引值，仅Dijkstra算法用到
    int length;      // 当前最短路径长度
    int pre;         // 路径最后经过的顶点
};
void Dijkstra(Graph& G,int s, Dist* &D) {
    D=new Dist[G.VerticesNum()];
    for(int i=0;i<G.VerticesNum();i++){         //初始化Mark数组、D数组
        G.Mark[i]=UNVISITED;
        D[i].length= INFINITY;
        D[i].index=i;                           //顶点的索引值；
        D[i].pre=s;                             //路径最后经过的顶点
    }
```

初始化

```
D[s].length=0;                                    //源点为s;
MinHeap<Dist> H(G.EdgesNum());                    //声明一个最小值堆;
H.Insert(D[s]);                                   //以源点s初始化堆;
for(i=0;i<G.VerticesNum();i++){
    Bool FOUND =  false;
    Dist d;
    while (! H.empty()){
        d = H.RemoveMin();
        if( G.Mark[d.index]==UNVISITED)
            FOUND = true;    break                //把该点加入已访问组;
    }
```

找下一最短路径顶点

```
        if (! FOUND)   break;
        int v =d.index;   G.Mark[v] = VISITED;  Visit(v);    //打印输出;
        for(Edge e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e)){
            if(D[G.ToVertex(e)].length>(D[v].length + G.Weight(e))){
                D[G.ToVertex(e)].length=D[v].length + G.Weight(e);
                D[G.ToVertex(e)].pre=v;
                H.insert(D[G.ToVertex(e)]);  //入堆;
            }
        }
    }//end for;
}
```

更新权值等信息（松弛技术）

时间复杂性分析

- 如果不采用最小堆的方式，而是通过两两比较来扫描D数组
  - 每次寻找权值最小结点，需要进行|V|次扫描，每次扫描|V|个顶点（$|V|^2$），而在更新D值处总共扫描|E|次
  - 总时间代价为$\Theta(|V|^2 + |E|) = \Theta(|V|^2)$
- 如果采用最小堆的方式
  - 每次改变D[i].length，通过先删除再重新插入的方法来改变顶点i在堆中的位置
  - 或者仅为某个顶点添加一个新值(更小的)，作为堆中新元素(而不作删除旧值的操作，因为旧值被找到时，该顶点一定被标记为VISITED，从而被忽略)。
  - 不作删除旧值的缺点是，在最差情况下，它将使堆中元素数目由$\Theta(|V|)$增加到$\Theta(|E|)$，此时总的时间代价为$\Theta((|V|+|E|)\log|E|)$，因为处理每条边时都必须对堆进行一次重排

2、Floyd算法

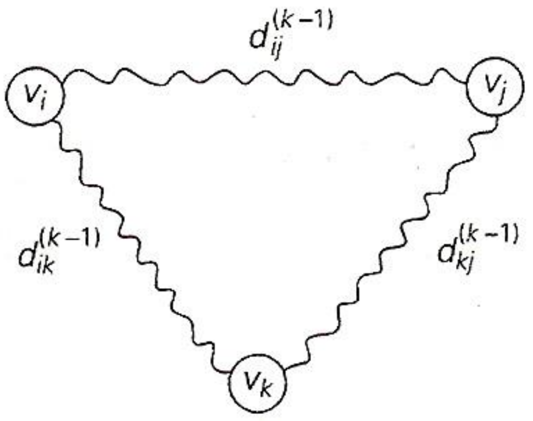
- 求每对顶点间的最短路径
  - 方法1：反复执行Dijkstra算法
  - 方法2：Floyd算法
- Floyd算法
  - 该算法名称以创始人之一、1978年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名。
  - 又称插点法，是一种用于寻找给定的加权图中任意节点对之间的最短路径算法。
  - 是一类动态规划的方法

算法过程

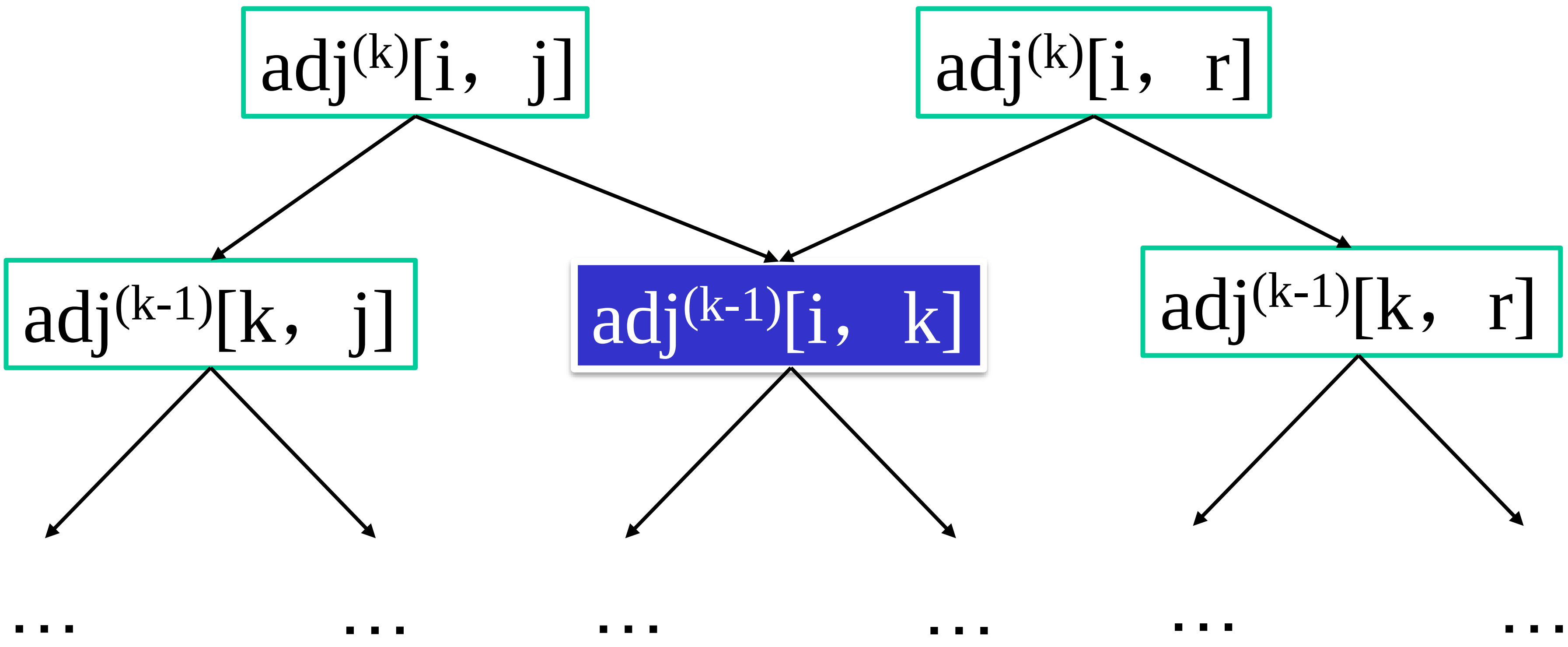
- 假设用相邻矩阵adj表示图
  - 任意两点间距离是边的权，如果两点间没有边直接相连，则权为无穷大（∞）
- 在原图的相邻矩阵$adj^{(0)}$上做n次迭代，递归地产生一个矩阵序列$adj^{(1)}$, $adj^{(2)}$,…, $adj^{(n)}$
  - $adj^{(k)}[i, j]$等于从顶点V_i到顶点V_j中间顶点序号不大于k的最短路径长度
- $adj^{(n)}$包括了所有最终的最短路径

递推公式

- 假设已求得矩阵$adj^{(k-1)}$，那么从顶点V_i到顶点V_j中间顶点的序号不大于k的最短路径有两种情况：
  - 中间不经过顶点V_k，那么就有$adj^{(k)}[i，j]=adj^{(k-1)}[i，j]$
  - 中间经过顶点V_k，那么$adj^{(k)}[i,j] < adj^{(k-1)}[i,j]$，且$adj^{(k)}[i,j]= adj^{(k-1)}[i,k] + adj^{(k-1)}[k,j]$



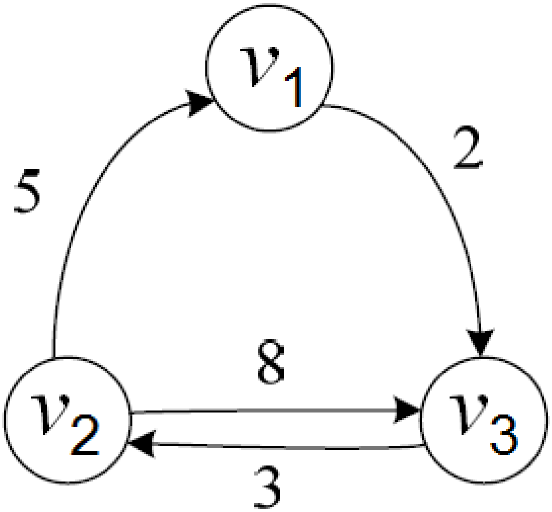
动态规划递推过程



最短路径确定

- 确定最短路径
  - 设置一个n×n的矩阵path，path[i，j]是由顶点v_i到顶点v_j的最短路径上排在顶点v_j前面的那个顶点，即当*k*是使得$adj^{(k)}[i,j]$达到最小值，那么就置path[i,j]=path[k,j]
  - 如果当前没有最短路径时，就将path[i，j]置为-1。

示例



$$adj=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 8\\ \infty & 3 & 0\end{bmatrix} \qquad path=\begin{bmatrix}1 & -1 & 1\\ 2 & 2 & 2\\ -1 & 3 & 3\end{bmatrix}$$

$$adj^{(1)}=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 7\\ \infty & 3 & 0\end{bmatrix} \qquad path=\begin{bmatrix}1 & -1 & 1\\ 2 & 2 & 1\\ -1 & 3 & 3\end{bmatrix}$$

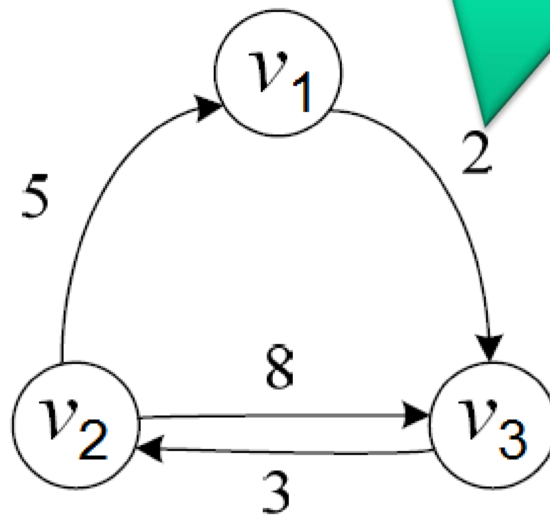
$$adj^{(2)}=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{bmatrix} \qquad path=\begin{bmatrix}1 & -1 & 1\\ 2 & 2 & 1\\ 2 & 3 & 3\end{bmatrix}$$

$$adj^{(3)}=\begin{bmatrix}0 & 5 & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{bmatrix} \qquad path=\begin{bmatrix}1 & 3 & 1\\ 2 & 2 & 1\\ 2 & 3 & 3\end{bmatrix}$$

示例

- 例如，想知道v1到v2的最短路径：
  - 由D[1][2]可知v1到v2的最短路径长度为5
  - 路径由path[1][2]=3可知顶点v2的前一顶点为v3，
  - 由path[1][3]=1可知v3的前一顶点为v1
  - 因此从v1到v2的最短路径为v1→v3→v2

允许边权值为负数！



$$adj^{(3)} = \begin{bmatrix} 0 & 5 & 2 \\ 5 & 0 & 7 \\ 8 & 3 & 0 \end{bmatrix} \qquad path = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \end{bmatrix}$$

Floyd算法实现

初始化

```
void Floyd(Graph& G, Dist** &D){
    int i,j,v;                                        //i, j, v作为计数器
    D=new Dist*[G.VerticesNum()];
    for(i=0; ;i<G.VerticesNum();i++)                  //申请数据D的空间
        D[i]=new Dist[G.VerticesNum()];
    for(i=0;i<G.VerticesNum();i++)                    //初始化D数组
        for(j=0;j<G.VerticesNum();j++)
            if(i==j){
                D[i][j].length=0;                     //权值矩阵
                D[i][j].pre=i;                        //path矩阵
            }else {
                D[i][j]=INFINITY;
                D[i][j].pre=-1;
            }
```

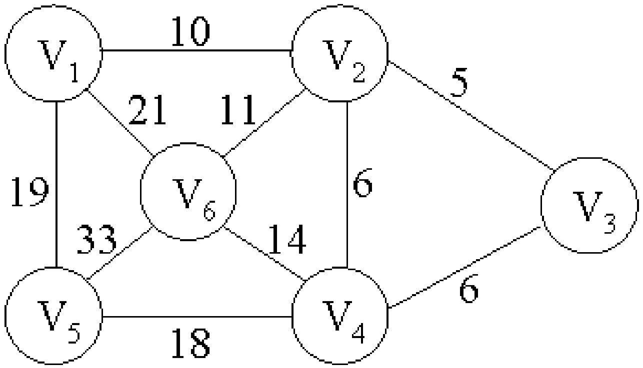
```
for(v=0;v<G.VerticesNum();v++)        //矩阵初始化，仅初始化邻接顶点
    for(Edge e=G.FirstEdge(v); G.IsEdge(e); e=G.NextEdge(e)){
        D[v][G.ToVertex(e)].length=G.Weight(e);
        D[v][G.ToVertex(e)].pre=v;
    }
//如果两顶点间最短路径经过顶点v，则有权值进行更新！
for(v=0;v<G.VerticesNum();v++)
    for(i=0;i<G.VerticesNum();i++)
        for(j=0;j<G.VerticesNum();j++)
            if((D[i][v].length +D[v][j].length) < D[i][j].length){
                D[i][j].length =D[i][v].length +D[v][j].length;
                D[i][j].pre=D[v][j].pre;
            }
}
```

权值、路径更新

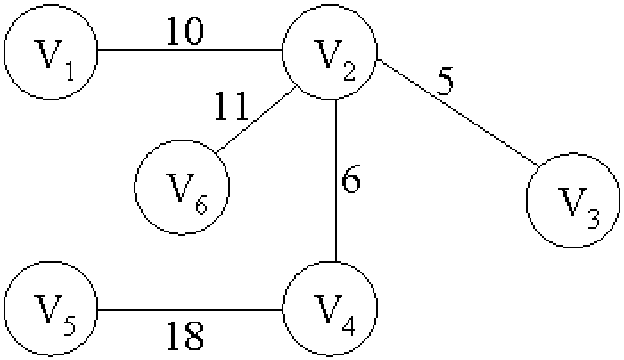
➢ **时间复杂性：三重for循环，复杂度是O(n^3)，适合稠密图**

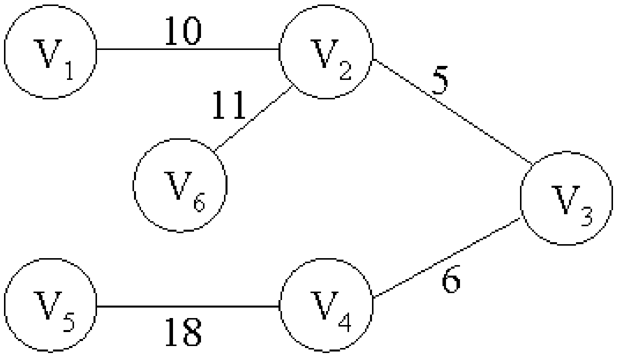
7.6 最小支撑(生成)树

➤ 公路网的造价问题



(a)交通网络





(b) 最小支撑树 (c) 最小支撑树

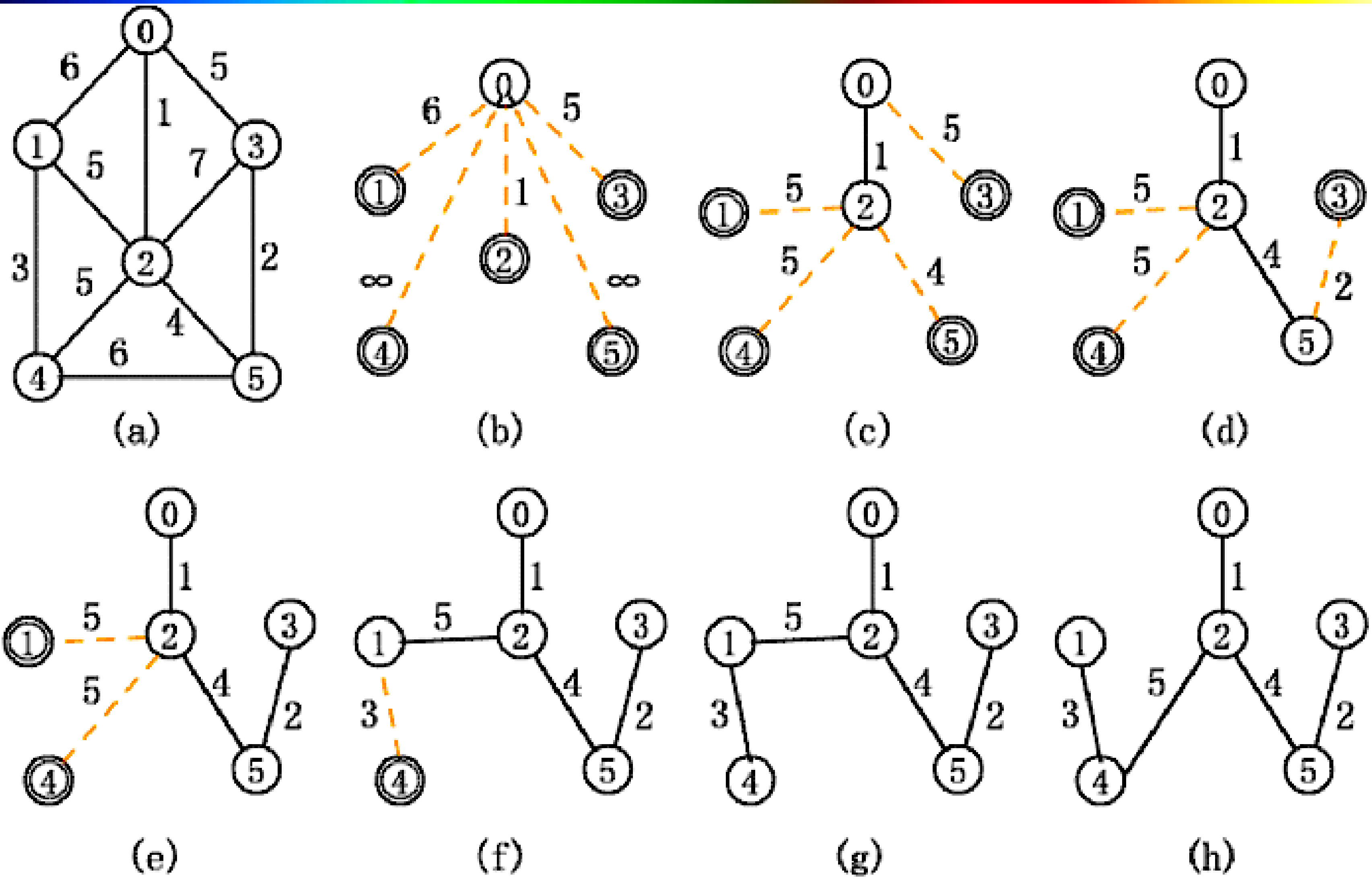
最小支撑树

- 最小支撑树（**M**inimum-cost **S**panning **T**ree，**MST**）
  - 对于带权的**连通无向图G**，其最小支撑树是一个包括G的所有顶点和部分边的图，这部分的边满足下列条件：
    - 保证图的连通性
    - 边权值总和最小
- 代表算法
  - **Prim**算法
  - **Kruskal**算法

1、Prim算法

- 具体操作
  - 从图中任意一个顶点开始，把这个顶点包括在MST里
  - 然后，在那些其一个端点已在MST里，另一个端点还未在MST里的边中，找权最小的一条边（相同边存在，任选择其一），并把这条边和其不在MST里的那个端点包括进MST里
  - 如此进行下去，每次往MST里加一个顶点和一条权最小的边，直到把所有的顶点都包括进MST里
- **MST不唯一，但是最小权值是确定的**

举例



Prim算法构造最小生成树的过程

Prim算法实现

```
void Prim(Graph& G, int s, Edge* &MST ) {
    int MSTtag=0;                                  //最小支撑树边的标号
    Edge *MST=new Edge[G.VerticesNum()-1];
    MinHeap<Edge> H(G.EdgesNum());
    for(int i=0;i<G.VerticesNum();i++)             //初始化Mark数组、距离数组
        G.Mark[i]=UNVISITED;
    int v=s;                                       //开始顶点
    G.Mark[v]=VISITED;                             //开始顶点首先被标记
    do{  //将以v为顶点，另一顶点未被标记的边插入最小值堆H
        for(Edge e= G. FirstEdge(v);G.IsEdge(e);e=G. NextEdge(e))
            if(G. Mark[G. ToVertex(e)]==UNVISITED)
                H.Insert(e);
        bool Found=false;
        while(!H.empty()) {                        //寻找下一条权最小的边
            e=H.RemoveMin();
```

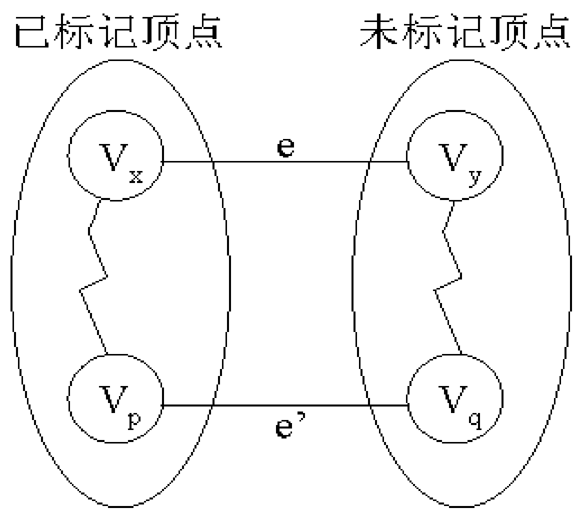
```
            if(G. Mark[G. ToVertex(e)]==UNVISITED){
                Found=true;
                break;
            }
        } //end while
        if(!Found){
            Print("不存在最小支撑树。");
            delete [] MST;                          //释放空间
            MST=NULL;                               //MST是空数组
            return;
        }
        v= G. ToVertex(e);
        G.Mark[v]=VISITED;       //在顶点v的标志位上做已访问的标记
        AddEdgetoMST(e,MST,MSTtag++); //将边e加到MST中
    }  while(MSTtag < (G. VerticesNum()-1))
}
```

证明

- **引理：用Prim算法构造的生成树是MST**
- 首先证明这样一个结论：
  - 设T(V*，E*)是连通无向图G=(V，E)的一棵正在构造的生成树，又E中有边e=(V_x，V_y)，其中$V_x \in$V*，V_y不属于V*，且e的权W(e)是所有一个端点在V*里，另一端不在V*里的边的权中最小者，则一定存在G的一棵包括T的MST包括边e=(V_x，V_y)。

反证法

1. 设G中任何包括T的MST都不包括e=(V_x，V_y)，且设T’是一棵这样的MST
2. 由于T’是连通的，因此有从V_x到V_y的路径V_x，…，V_y
3. 把边e=(V_x，V_y)加进树T’，得到一个回路V_x，…，V_y，V_x
4. 上述路径V_x，…，V_y中必有边e’=(V_p，V_q)，其中$V_p \in V^*$，V_q不属于V*，由条件知边的权W(e’)≥W(e)，从回路中去掉边e’，回路打开，成为另一棵生成树T”，T”包括边e=(V_x，V_y)，且各边权的总和不大于T’各边权总和
5. 因此T”是一棵包括边e的MST，与假设矛盾，即证明了我们的结论

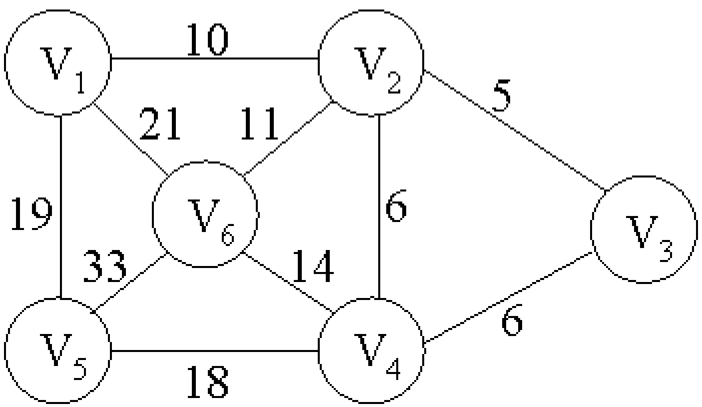


Prim算法与Dijkstra算法的区别

- 相同点：都是贪心的思路
- 不同点
  - Prim算法要寻找的是离已加入顶点距离最近的点
  - 而Dijkstra是寻找离源点距离最近的点
- 其时间复杂度分析与Dijkstra算法相同

2、Kruskal算法

- **Kruskal算法的基本思想**
  - 对于图G=（V，E），开始时，将顶点集分为|V|个等价类，每个等价类包括一个顶点
  - 然后，以权的大小为顺序处理各条边，如果某条边连接两个不同等价类的顶点，则这条边被添加到MST，两个等价类被合并为一个；
  - 反复执行此过程，直到只剩下一个等价类


V1
V2
V3
V4
V5
V6
10
5
21
11
19
6
33
14
6
18


初始状态：
步骤1：
处理边(V2, V3)
步骤2：
处理边(V3, V4)
步骤3：
处理边(V1, V2)
步骤4：
处理边(V2, V6)
步骤5：
处理边(V4, V5)

Kruskal算法描述

```
void Kruskal(Graph& G, Edge* &MST{
    Partree A(G.VerticesNum());                    //等价类
    MinHeap<Edge> H(G.EdgesNum());                 //声明一个最小堆
    MST=new Edge[G.VerticesNum()-1];               //最小支撑树
    int MSTtag=0;                                  //最小支撑树边的标号
    for(int i=0; i<G.VerticesNum(); i++) {         //将图的所有边插入最小值堆H中
        for(Edge e= G.FirstEdge(i); G.IsEdge(e);e=G. NextEdge(e))
            if(G.FromVertex(e)< G.ToVertex(e))
                H.Insert(e);
    int EquNum=G.VerticesNum();                    //开始时有|V|个等价类
    while(EquNum>1) {                              //合并等价类
        Edge e=H.RemoveMin();                      //获得下一条权最小的边
```

边集入堆

```
        int from=G.FromVertex(e);          //记录该条边的信息
        int to= G.ToVertex(e);
        if(A.differ(from,to)) {      //如果边e的两个顶点不在一个等价类
            //将边e的两个顶点所在的两个等价类合并为一个
            A.UNION(from,to);
            AddEdgetoMST(e,MST,MSTtag++); //将边e加到MST
            EquNum--;                              //将等价类的个数减1
        }
    }//end while
}
```

性能分析

- 使用了路径压缩，**differ**和**UNION**函数几乎是常数
- 假设可能对几乎所有边都判断过了，则最坏情况下算法时间代价为 **O(|E|log |E|)**，即堆排序的时间
- 适合于稀疏图