第六章、图

一、图的基本概念

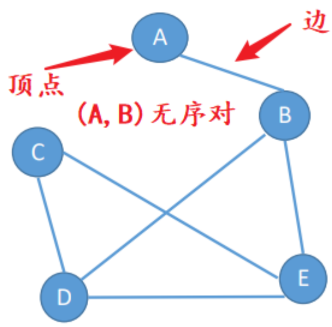
1. 图的定义

    图G由顶点集V和边集E组成，记为G = (V, E)，其中V(G)表示图G中顶点的有限非空集；E(G) 表示图G中顶点之间的关系（边）集合。若V = {v1, v2, ... , vn}，则用|V|表示图G中顶点的个 数，也称图G的阶，E = {(u, v) | uÎV, vÎV}，用|E|表示图G中边的条数。

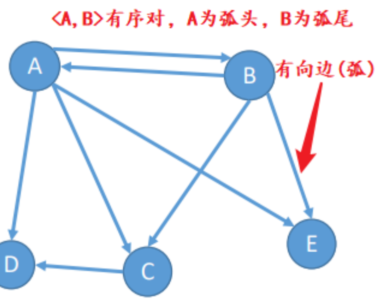
    注意：线性表可以是空表，树可以是空树，但图不可以是空，即V一定是非空集

2. 基本术语

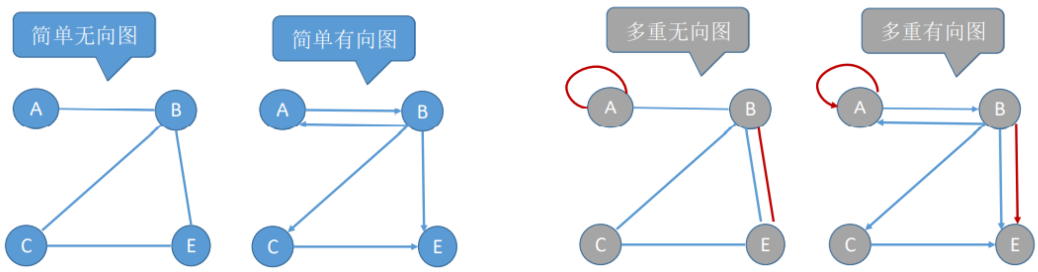
    - 无向图：



    - 有向图：



    - 简单图、多重图、完全图



无完全图边数为$\frac{n(n-1)}{2}$、有完全图边数为$n(n-1)$

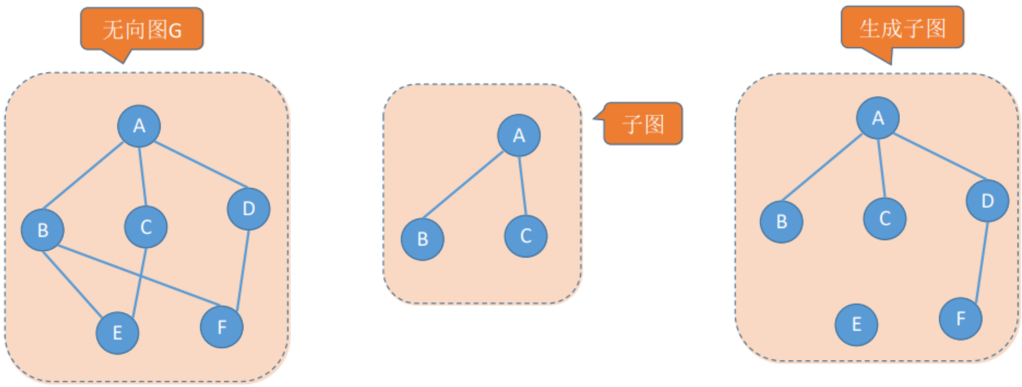
    - 顶点的度、出度、入度

无向图：顶点的度依附于该顶点的边的条数，记为 `TD(v)` 。具有n个顶点、e条边的无向图中、全部顶点的度的和等于边数的 `2e` 。

有向图：入度是以顶点v为终点的有向边的数目，记为 `ID(v)` ，出度是以顶点v为起点的有向边的数目，记为 `OD(v)` 。顶点v的度等于其入度和出度之和，即 `TD(v) = ID(v) + OD(v)` 。

- 子图、生成图

两个图G = (V, E)和G’ = (V’ , E’)，若V’ 是V的子集，且E’ 是 E的子集，则称G’ 是G的子图。若有满足V(G’) = V(G)的子图G’ ，则称G’ 为G的生成子图



- 连通、连通图、连通分量

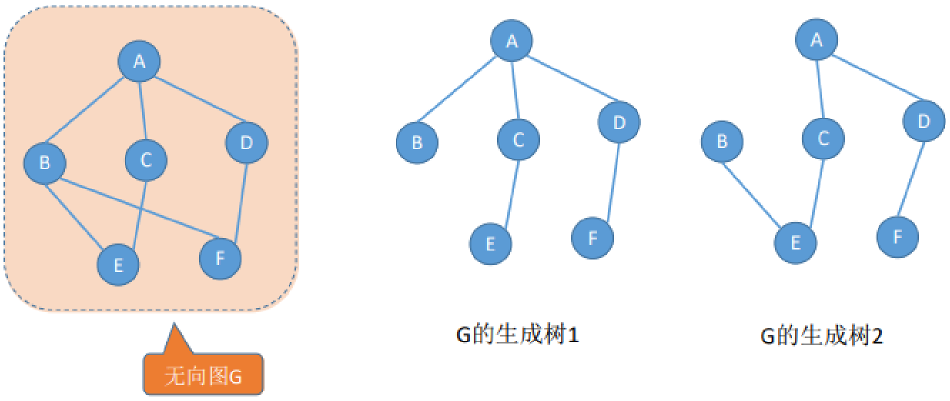
连通：无向图：若从顶点v到顶点w有路径存在，则称v和w是 *连通的* 、有向图：若从顶点v到顶点w和从顶点w到顶点v之间都有路径，则称这两个顶点是 *强连通的* 。

连通图：无向图：若图中 *任意两个顶点都是连通的* ，则称图为连通图，否则称为非连通图。有向图：若图中任何一对顶点都是强连通的，则称此图为 强连通图。

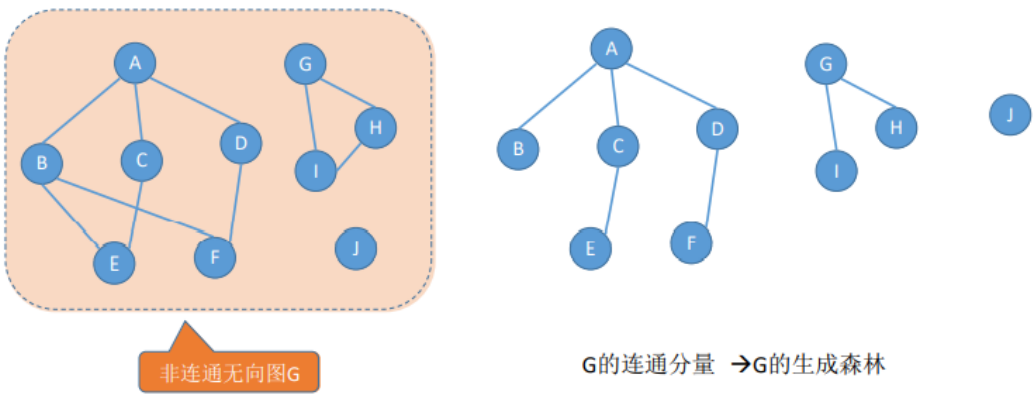
连通分量：无向图：又叫极大连通子图（子图必须连通，且包含 尽可能多的顶点和边）、有向图：极大强连通子图称为有向图的强连通分量。

- 生成树、生成森林

生成树：包含所有顶点的一个极小连通子图（极小连通子图：边尽可能的少， 但要保持连通）。若图中顶点数为n，则它的生成树含有 n-1 条边。



生成森林：在非连通图中，连通分量的生成树构成了非连通图的生成森林。

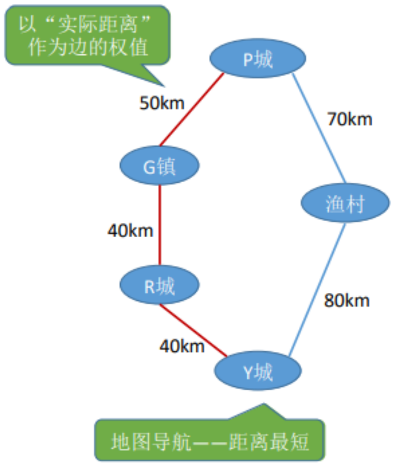


- 边的权、带权的边（网）

  边的权：在一个图中，每条边都可以标上具有某种含义的数值，该数值称为该边的权值。

  网：边上带有权值的图称为带权图。

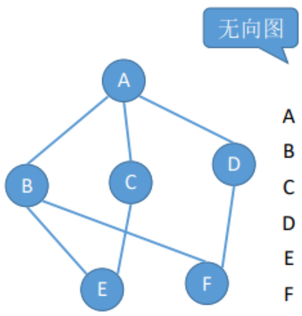
  带权路径长度：一条路径上所有边的权值之和，称为该路径的带权路径长度。



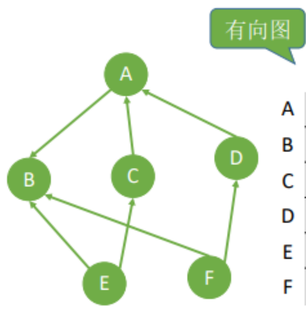
- 路径（顶点v到顶点w之间的一条路径是指顶点序列）、路径长度（路径上边的数目）、回路(第一个顶点和最后一个顶点相同的路径称为回路或环)、点到点的距离（从顶点u出发到顶点v的最短路径若存在，则此路径的长度称为从u到v的距离。 若从u到v根本不存在路径，则记该距离为无穷（∞））

二、图的存储

1.邻接矩阵



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	1	0
D	1	0	0	0	0	1
E	0	1	1	0	0	0
F	0	1	0	1	0	0



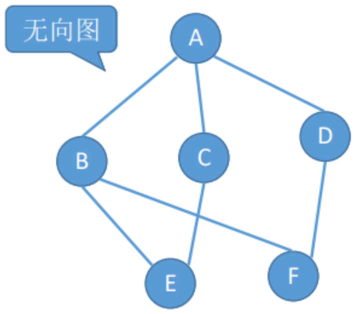
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	0
C	1	0	0	0	0	0
D	1	0	0	0	0	0
E	0	1	1	0	0	0
F	0	1	0	1	0	0

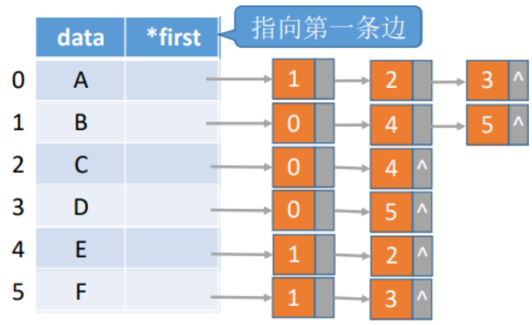
```
#define MaxVertexNum 100     // 顶点数目的最大值
typedef char VertexType;
typedef int EdgeType;
typedef struct{
    VertexType Vex[MaxVertexNum];    // 顶点表（存顶点）
    EdgeType Edge[MaxVertexNum][MaxVertexNum];  // 邻接矩阵、边表
    int vexnum,arcnum;       // 图的当前顶点数和边数/弧数
}MGraph;
```

特性:

- 无向图：第i个结点的度 = 第i行（或第i列）的非零元素个数。
- 有向图：第i个结点的出度 = 第i行的非零元素个数； 第i个结点的入度 = 第i列的非零元素个数； 第i个结点的度 = 第i行、第i列的非零元素个数之和。
- 邻接矩阵法求顶点的度/出度/入度的时间复杂度为 O(|V|)、空间复杂度：O($|V|^2$)（*只和顶点数相关，和实际的边数无关*）。
- 设图G的邻接矩阵为A（矩阵元素为0/1），则A^n 的元素$A^n[i][j]$等于由顶点i到顶点j的长度为n的路径的数目。

2.邻接表





- 顺序+链式存储

```
// 边表
typedef struct ArcNode {
    int adjvex; // 边/弧指向哪个结点
    struct ArcNode *next;   // 指向下一条弧的指针
    // InfoType info;   // 边的权值
}ArcNode;
// 顶点表
typedef struct VNode {
    VertexType data;    // 顶点信息
    ArNode *first;      // 第一条边/弧
}VNode,AdjList[MaxVertexNum];
// 用邻接表存储图
typedef struct {
    AdjList vertices;   // 邻接表
    int vexnum,arcnum;  // 顶点个数，边数
}ALGraph;
```

性质：

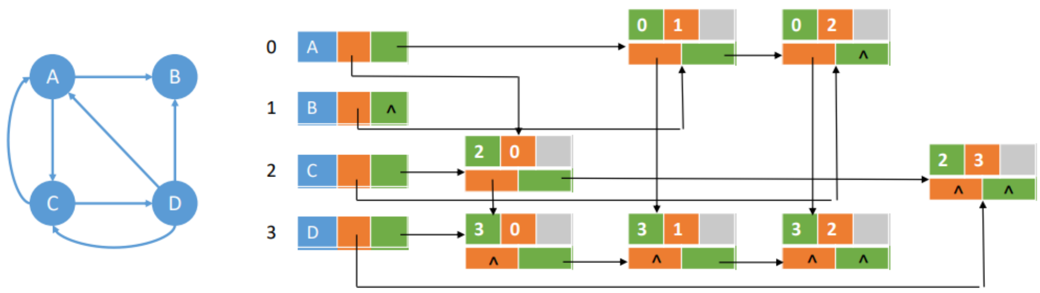
- 边结点的数量是 `2|E|` ， 整体空间复杂度为 `O(|V| + 2|E|)`
- 图的邻接表表示 方式并不唯一

对比：

	邻接表	邻接矩阵
空间复杂度	无向图 O(|V| + 2|E|)；有向图O(|V| + |E|)	O(|V|^2)
适合用于	存储稀疏图	存储稠密图
表示方式	不唯一	唯一
计算度/出度/入度	计算有向图的度、入度不方便，其余很方便	必须遍历对应行或列
找相邻的边	找有向图的入边不方便，其余很方便	必须遍历对应行或列

3.十字链表

十字链表只用于存储有向图

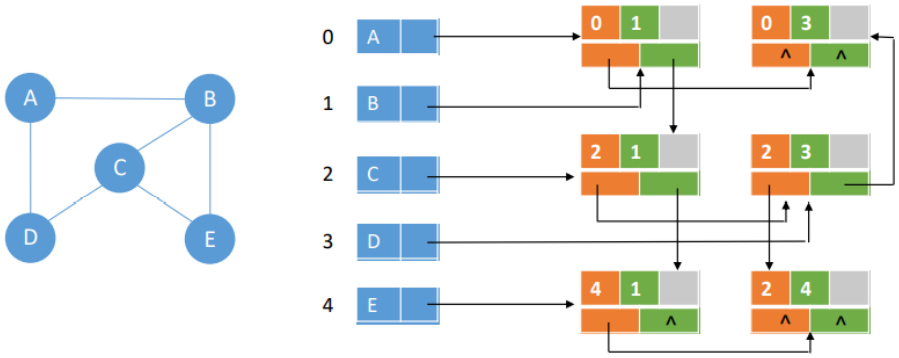



```
// 边表结点
typedef struct ArcBox{
    int tailvex,headvex;    // 弧尾、弧头
    struct ArcBox *hlink,*tlink;    //
    // InfoType *info;  // 弧的信息
}
// 顶点结点
typedef struct VexNode{
    VertexType data;    // 顶点数据
    ArcBox *firstin,*firstout;
}VexNode;

// 十字链表
typedef struct{
    VexNode xlist[MAX_VerTex_NUM];  // 表头向量
    int vexnum,arcnum;  // 有向图的当前顶点数和边数
}OLGraph;
```

4.邻接多重表

邻接多重表只适用于存储无向图




```
typedef enum{unvisited,visited} VisitIf;
// 边结点
typedef struct EBox{
    VisitIf mark;   // 标记变量判断是否访问
    int ivex,jvex;
    struct EBox *ilink,*jlink;
    // InfoType *info;  // 结点信息
}EBox;
// 顶点表
typedef struct VexBox{
    VertexType data;
    EBox *firstedge;    // 指向第一条边
}VexBox;
// 多重表
typedef struct{
    VexBox adjmulist[MAX_VERTEX_NUM];
    int vexnum,edgenum;
}AMLGraph;
```

三、图的操作

Adjacent(G,x,y)：判断图G是否存在边或(x, y)。

Neighbors(G,x)：列出图G中与结点x邻接的边。

InsertVertex(G,x)：在图G中插入顶点x。

DeleteVertex(G,x)：从图G中删除顶点x。

AddEdge(G,x,y)：若无向边(x, y)或有向边不存在，则向图G中添加该边。

RemoveEdge(G,x,y)：若无向边(x, y)或有向边存在，则从图G中删除该边。

FirstNeighbor(G,x)：求图G中顶点x的第一个邻接点，若有则返回顶点号。若x没有邻接点或图中不存在x，则返回-1。

> NextNeighbor(G,x,y)：假设图G中顶点y是顶点x的一个邻接点，返回除y之外顶点x的下一个邻接点的顶点号，若y是x的最后一个邻接点，则返回-1。
>
> Get_edge_value(G,x,y)：获取图G中边(x, y)或对应的权值。
>
> Set_edge_value(G,x,y,v)：设置图G中边(x, y)或对应的权值为v。

四、图的遍历

1.深度优先遍历

```
void DFSTraverse(Graph G){
    for(int i=0;i<G.vexnum;++i)
        visited[i]=FALSE;
    for(int i=0;i<G.vexnum;++i)
        if(!visited[i])
            DFS(G,v);
}
void DFS(Graph G,int v){
    visit(V);
    visited[v]=TRUE;
    for(w=FirstNeighbor(G,v);w>=0;w=Neighbor(G,v,w))
        if(!visited[i]){
            DFS(G,w);
        }
}
```

时间复杂度：

`邻接矩阵：` 访问 |V| 个顶点需要O(|V|)的时间 查找每个顶点的邻接点都需要O(|V|)的时间，而总共有|V|个顶点 时间复杂度= $O(|V|^2)$

`邻接表`： 访问 |V| 个顶点需要O(|V|)的时间 查找各个顶点的邻接点共需要O(|E|)的时间， `时间复杂度= $O(|V|+|E|)$

补充：同一个图的邻接矩阵表示方式唯一，因此深度优先遍历序列唯一 同一个图邻接表表示方式不唯一，因此深度优先遍历序列不唯一。

深度优先生成树： 由深度优先遍历生成的树。

2.广度优先遍历(BFS)

广度优先遍历（Breadth-First-Search, BFS）要点：

1. 找到与一个顶点相邻的所有顶点
2. 标记哪些顶点被访问过
3. 需要一个辅助队列

```
bool visited[MaxVertexNum];
void BFSTraverse(Graph G){
    // 访问标记数组初始化
    for(int i=0;i<G.vexnum;++i)
```

```
        visited[i]=FALSE;
    InitQueue(Q);   // 初始化辅助队列
    for(int i=0;i<G.vexnum;++i)
        if(!visited[i])
            BFS(G,i);
}
void BFS(Graph G,int v){
    visit(v);   // 访问顶点
    visited[v]=TRUE;    // 标记结点以备访问
    Enqueue(Q,v);   // 被访问结点入队
    while(!isEmpty(Q)){
        DeQueue(Q,v);   // 顶点v出队
        // 循环访问邻接点
        for(w=FirstNeighbor(G,v);w>=0;w=Neighbor(G,v,w))
            // 检测v的所有邻接点
            if(!visited[w]){
                visit[w];
                visited[w]=TRUE;
                Enqueue(Q,w);   // 被访问结点入队
            }
    }
}
```

时间复杂度：

`邻接矩阵`：访问|V|个顶点需要O(|V|)的时间、查找每个顶点的邻接点都需要O(|V|)的时间，而总共有|V|个顶点时间复杂度= $O(|V|^2)$

`邻接表`： 访问 |V| 个顶点需要O(|V|)的时间 查找各个顶点的邻接点共需要O(|E|)的时间， `时间复杂度= $O(|V|+|E|)$

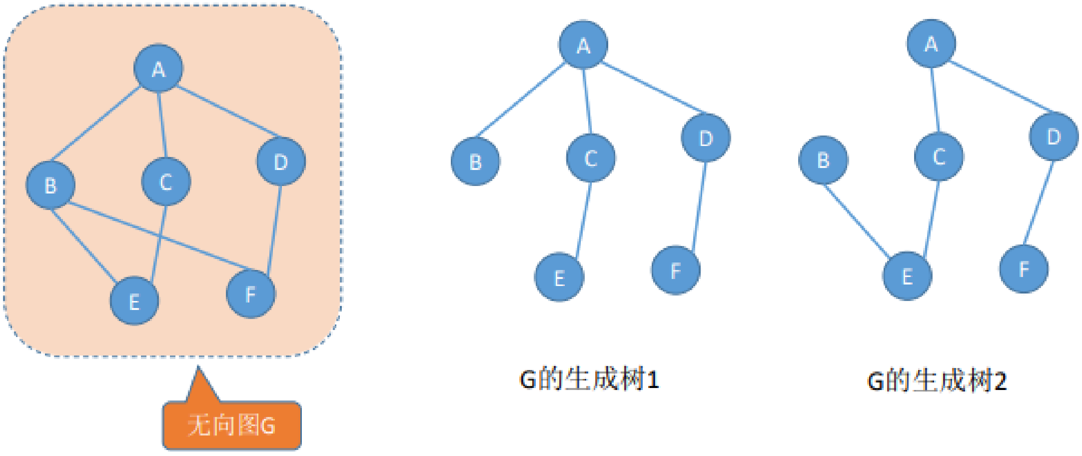
空间复杂度： 最坏情况，辅助队列大小为 $O(|V|)$

图的连通性： 可以看有几个连通分量，要是连通分量为1，那么图就是连通的是单图，要是有多个连通分量那么就是非连通的。

五、最小生成树（最小代价树）

1. 概念：连通图的生成树是包含图中全部顶点的一个极小连通子图。

若图中顶点数为n，则它的生成树含有 n-1 条边。对生成树而言，若砍去它的一条边，则会变成非连通 图，若加上一条边则会形成一个回路。



G的生成树1　　G的生成树2

2. 最小代价树：对于一个带权连通无向图G = (V, E)，生成树不同，每棵树的权（即树中所有边上的权值 之和）也可能不同。设R为G的所有生成树的集合，若T为R中边的权值之和最小的生成 树，则T称为G的最小生成树（Minimum-Spanning-Tree, MST）。

- 特点：
    - 最小生成树可能有多个，但边的权值之和总是唯一且最小的
    - 最小生成树的边数 = 顶点数 - 1。砍掉一条则不连通，增加一条边则会出现回路。
    - 如果一个连通图本身就是一棵树，则其最小生成树就是它本身
    - 只有连通图才有生成树，非连通图只有生成森林

3. 最小生成树求法：

普里姆算法(加点法)： 从某一个顶点开始构建生成树； 每次将代价最小的新顶点纳入生成 树，直到所有顶点都纳入为止。

算法实现思想：

初始状态：V是所有顶点的集合，即V={A,B,C,D,E,F,G}；U和T都是空！
第1步：将顶点A加入到U中。
此时，U={A}。
第2步：将顶点B加入到U中。
上一步操作之后，U={A}, V-U={B,C,D,E,F,G}；因此，边(A,B)的权值最小。将顶点B添加到U中；此时，U={A,B}。
第3步：将顶点F加入到U中。
上一步操作之后，U={A,B}, V-U={C,D,E,F,G}；因此，边(B,F)的权值最小。将顶点F添加到U中；此时，U={A,B,F}。
第4步：将顶点E加入到U中。
上一步操作之后，U={A,B,F}, V-U={C,D,E,G}；因此，边(F,E)的权值最小。将顶点E添加到U中；此时，U={A,B,F,E}。
第5步：将顶点D加入到U中。
上一步操作之后，U={A,B,F,E}, V-U={C,D,G}；因此，边(E,D)的权值最小。将顶点D添加到U中；此时，U={A,B,F,E,D}。
第6步：将顶点C加入到U中。
上一步操作之后，U={A,B,F,E,D}, V-U={C,G}；因此，边(D,C)的权值最小。将顶点C添加到U中；此时，U={A,B,F,E,D,C}。
第7步：将顶点G加入到U中。
上一步操作之后，U={A,B,F,E,D,C}, V-U={G}；因此，边(F,G)的权值最小。将顶点G添加到U中；此时，U=V。

此时，最小生成树构造完成！它包括的顶点依次是：**A B F E D C G**。

```
//普利姆算法：从一个顶点开始形成一棵树，依次添加剩余顶点中到达这棵树权值最小的顶点
```

```
//包含两个辅助数组 lowcost[]数组用于存储当前图中各个顶点到现有树的权值
// adjvex[]数组用于存储各个顶点与树相接的对应“前驱”顶点
void prim(MGraph G) //图以零解矩阵存储便于取顶点间权值
{
    int i,min,k;
    int lowcost[G.vexnum];
    int adjvex[G.vexnum];
    // 初始化部分
    for(i = 0;i < G.vexnum;++i){
        lowcost[i] = G.edges[0][i]; //初始化lowcost[]数组，当前存储图中各个顶点到源点的权值
        adjvex[i] = 0; //初始化adjvex[]数组，此时下一个接入树中的顶点的前驱必然是0号顶点
    }
    // 实现部分
    // 此处为重点，这个for循环执行N-1次，即我们只需添加N-1条边
    for(j = 1;j < G.vexnum;++j){
        min = 65535;   // 每一次循环开始时将min的值重置
        //扫描lowcost[]数组，找出下一个到达树的权值最小的顶点
        for(i = 0;i < G.vexnum;++i){
            if(lowcost[i] != 0 && lowcost[i] < min){
                min = lowcost[i];
                k = i;
        }
    }

    // 数据打印部分
    //将这个到树权值最小的顶点加入到树中，其在lowcost[]数组的值标记为0
    lowcost[k] = 0;
    //打印新选中的顶点到树的边
    printf("%d->%d",adjvex[k],k);
        //更新lowcost[]数组和adjvex[]数组
    for(i = 0;i < G.vexnum;++i){
        if(lowcost[i] != 0 && lowcost[i] > G.edges[k][i]){
            lowcost[i] = G.vexnum[k][i];
            adjvex[i] = k;
        }
    }
}
```

时间复杂度： $O(|V|^2)$ 适合用于边稠密图

克鲁斯卡尔（加边法）： 每次选择一条权值最小的边，使这 条边的两头连通（原本已经连通的 就不选） 直到所有结点都连通。

算法步骤：

第1步：将边<E,F>加入R中。
边<E,F>的权值最小，因此将它加入到最小生成树结果R中。
第2步：将边<C,D>加入R中。
上一步操作之后，边<C,D>的权值最小，因此将它加入到最小生成树结果R中。
第3步：将边<D,E>加入R中。
上一步操作之后，边<D,E>的权值最小，因此将它加入到最小生成树结果R中。

第4步：将边<B,F>加入R中。

上一步操作之后，边<C,E>的权值最小，但<C,E>会和已有的边构成回路；因此，跳过边<C,E>。同理，跳过边<C,F>。将边<B,F>加入到最小生成树结果R中。

第5步：将边<E,G>加入R中。

上一步操作之后，边<E,G>的权值最小，因此将它加入到最小生成树结果R中。

第6步：将边<A,B>加入R中。

上一步操作之后，边<F,G>的权值最小，但<F,G>会和已有的边构成回路；因此，跳过边<F,G>。同理，跳过边<B,C>。将边<A,B>加入到最小生成树结果R中。

时间复杂度： $O(|E|log_2|E|)$ 适合用于边稀疏图

代码实现：

```
/*
 * 克鲁斯卡尔（Kruskal)最小生成树
 */
void kruskal(Graph G)
{
    int i,m,n,p1,p2;
    int length;
    int index = 0;          // rets数组的索引
    int vends[MAX]={0};     // 用于保存"已有最小生成树"中每个顶点在该最小树中的终点。
    EData rets[MAX];        // 结果数组，保存kruskal最小生成树的边
    EData *edges;           // 图对应的所有边
    // 获取"图中所有的边"
    edges = get_edges(G);
    // 将边按照"权"的大小进行排序(从小到大)
    sorted_edges(edges, G.edgnum);
    for (i=0; i<G.edgnum; i++){
        p1 = get_position(G, edges[i].start);   // 获取第i条边的"起点"的序号
        p2 = get_position(G, edges[i].end);     // 获取第i条边的"终点"的序号
        m = get_end(vends, p1);                 // 获取p1在"已有的最小生成树"中的终点
        n = get_end(vends, p2);                 // 获取p2在"已有的最小生成树"中的终点
        // 如果m!=n，意味着"边i"与"已经添加到最小生成树中的顶点"没有形成环路
        if (m != n){
            vends[m] = n;                       // 设置m在"已有的最小生成树"中的终点为n
            rets[index++] = edges[i];           // 保存结果
        }
    }
    free(edges);
    // 统计并打印"kruskal最小生成树"的信息
    length = 0;
    for (i = 0; i < index; i++)
        length += rets[i].weight;
    printf("Kruskal=%d: ", length);
    for (i = 0; i < index; i++)
        printf("(%c,%c) ", rets[i].start, rets[i].end);
    printf("\n");
}
```

六、最短路径

1. **BFS算法**（无权图）：无权图可以视为一种特殊的带权图，只是每条边的权值都为1

代码实现：

```
void BFS_Min_Distance(Graph G,int u){
    // 初始化，d[i]表示从u到i的最短路径
    for(int i=0;i<G.vexnum;++i){
        d[i]=max_num;
        path[i]=1;
    }
    d[u]=0;
    visited[u]=TRUE;    // 标记结点以备访问
    Enqueue(Q,u);    // 被访问结点入队
    while(!isEmpty(Q)){
        DeQueue(Q,u);    // 顶点v出队
        // 循环访问邻接点
        for(w=FirstNeighbor(G,v);w>=0;w=Neighbor(G,v,w))
            // 检测v的所有邻接点
            if(!visited[w]){
                d[w]=d[u]+1;
                path[w]=u;
                visited[w]=TRUE;
                Enqueue(Q,w);    // 被访问结点入队
            }
    }
}
```

补、一定是以2为根 的，高度最小的生成树

2. **迪杰斯特拉算法**

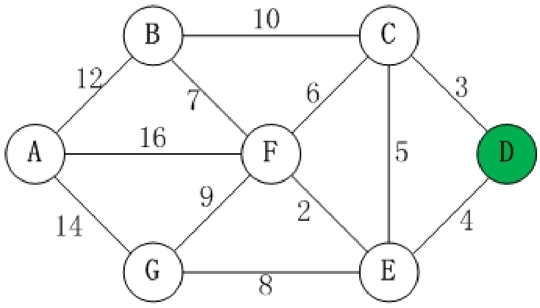
思想：

(1) 初始时，S只包含起点s；U包含除s外的其他顶点，且U中顶点的距离为"起点s到该顶点的距离"[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为∞]。

(2) 从U中选出"距离最短的顶点k"，并将顶点k加入到S中；同时，从U中移除顶点k。

(3) 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。

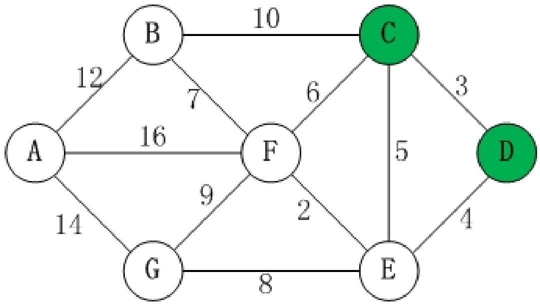
(4)重复步骤(2)和(3)，直到遍历完所有顶点。



第1步：
选取顶点D

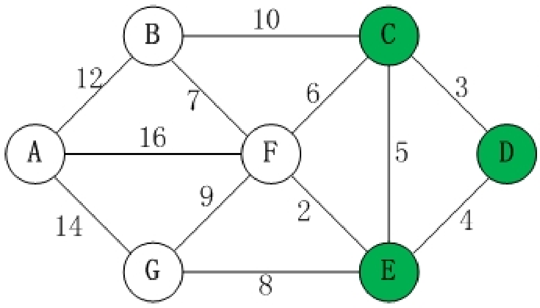
S={D(0)}
U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}

注：
(01) S是已计算出最短路径的定点的集合
(02) U是未计算出最短路径的定点的集合
(03) C(3)表示顶点C到起点D的最短距离是3



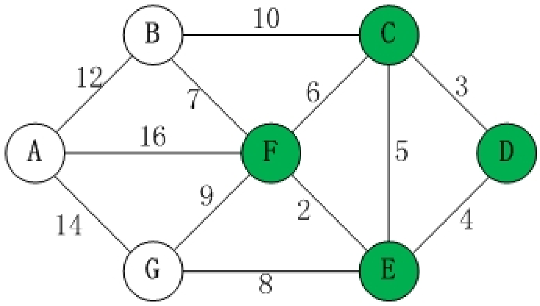
第2步：
选取顶点C

S={D(0),C(3)}
U={A(∞),B(23),E(4),F(9),G(∞)}



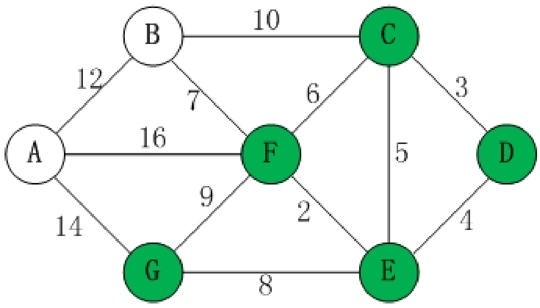
第3步：
选取顶点E

S={D(0),C(3),E(4)}
U={A(∞),B(23),F(6),G(12)}



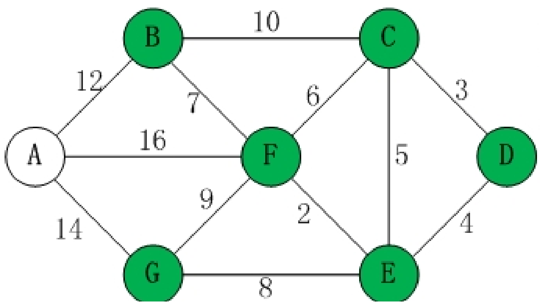
第4步：
选取顶点F

S={D(0),C(3),E(4),F(6)}
U={A(22),B(13),G(12)}



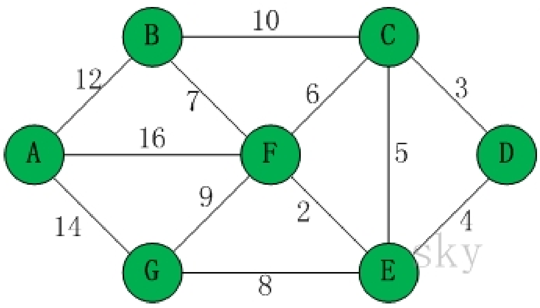
第5步：
选取顶点G

S={D(0),C(3),E(4),F(6),G(12)}
U={A(22),B(13)}



第6步：
选取顶点B

S={D(0),C(3),E(4),F(6),G(12),B(13)}
U={A(22)}



第7步：
选取顶点A

S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}

代码实现:

```
/*
 * Dijkstra最短路径。
 * 即，统计图(G)中"顶点vs"到其它各个顶点的最短路径。
 *
 * 参数说明：
 *        G -- 图
 *       vs -- 起始顶点(start vertex)。即计算"顶点vs"到其它顶点的最短路径。
 *     prev -- 前驱顶点数组。即，prev[i]的值是"顶点vs"到"顶点i"的最短路径所经历的全部顶点中，位于"顶点i"之前的那个顶点。
 *     dist -- 长度数组。即，dist[i]是"顶点vs"到"顶点i"的最短路径的长度。
 */
void dijkstra(Graph G, int vs, int prev[], int dist[]){
    int i,j,k;
    int min;
    int tmp;
    int flag[MAX];      // flag[i]=1表示"顶点vs"到"顶点i"的最短路径已成功获取。
    // 初始化
    for (i = 0; i < G.vexnum; i++){
        flag[i] = 0;              // 顶点i的最短路径还没获取到。
        prev[i] = 0;              // 顶点i的前驱顶点为0。
        dist[i] = G.matrix[vs][i];// 顶点i的最短路径为"顶点vs"到"顶点i"的权。
    }
    // 对"顶点vs"自身进行初始化
    flag[vs] = 1;
    dist[vs] = 0;
    // 遍历G.vexnum-1次；每次找出一个顶点的最短路径。
    for (i = 1; i < G.vexnum; i++){
        // 寻找当前最小的路径；
        // 即，在未获取最短路径的顶点中，找到离vs最近的顶点(k)。
        min = INF;
        for (j = 0; j < G.vexnum; j++){
            if (flag[j]==0 && dist[j]<min){
                min = dist[j];
                k = j;
            }
        }
        // 标记"顶点k"为已经获取到最短路径
        flag[k] = 1;

```

```
        // 修正当前最短路径和前驱顶点
        // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最短路径和前驱顶点"。
        for (j = 0; j < G.vexnum; j++){
            tmp = (G.matrix[k][j]==INF ? INF : (min + G.matrix[k][j])); // 防止溢出
            if (flag[j] == 0 && (tmp  < dist[j])){
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }
    // 打印dijkstra最短路径的结果
    printf("dijkstra(%c): \n", G.vexs[vs]);
    for (i = 0; i < G.vexnum; i++)
        printf("  shortest(%c, %c)=%d\n", G.vexs[vs], G.vexs[i], dist[i]);
}
```

时间复杂度： 总时间复杂度$O(n^2)$，即$O(|V|^2)$。

3. **弗洛伊德算法**

初始状态：S是记录各个顶点间最短路径的矩阵。
第1步：初始化S。
矩阵S中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞。实际上，就是将图的原始矩阵复制到S中。
注:a[i][j]表示矩阵S中顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

第2步：以顶点A(第1个顶点)为中介点，若 `a[i][j]> a[i][0]+a[0][j]`，则设置 `a[i][j]=a[i][0]+a[0][j]`。以顶点 `a[1]`，即顶点B和顶点G之间的距离为例)，上一步操作之后，`a[1][6]` =∞；而将A作为中介点时，(B,A)=12，(A,G)=14，因此B和G之间的距离可以更新为26。

同理，依次将顶点B,C,D,E,F,G作为中介点，并更新a[i][j]的大小。

代码实现：

```
/*
 * floyd最短路径。
 * 即，统计图中各个顶点间的最短路径。
 *
 * 参数说明：
 *        G -- 图
 *     path -- 路径。path[i][j]=k表示，"顶点i"到"顶点j"的最短路径会经过顶点k。
 *     dist -- 长度数组。即，dist[i][j]=sum表示，"顶点i"到"顶点j"的最短路径的长度是sum。
 */
void floyd(Graph G, int path[][MAX], int dist[][MAX]){
    int i,j,k;
    int tmp;
    // 初始化
    for (i = 0; i < G.vexnum; i++){
        for (j = 0; j < G.vexnum; j++){
            dist[i][j] = G.matrix[i][j];    // "顶点i"到"顶点j"的路径长度为"i到j的权值"。
```

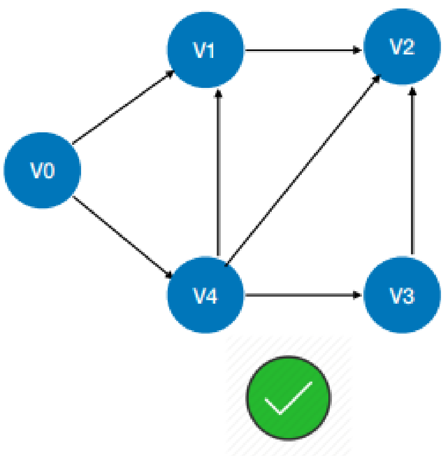
```
                path[i][j] = j;                 // "顶点i"到"顶点j"的最短路径是经过顶点j。
            }
        }
        // 计算最短路径
        for (k = 0; k < G.vexnum; k++){
            for (i = 0; i < G.vexnum; i++){
                for (j = 0; j < G.vexnum; j++){
                    // 如果经过下标为k顶点路径比原两点间路径更短，则更新dist[i][j]和path[i][j]
                    tmp = (dist[i][k]==INF || dist[k][j]==INF) ? INF : (dist[i][k] + dist[k][j]);
                    if (dist[i][j] > tmp){
                        // "i到j最短路径"对应的值设，为更小的一个(即经过k)
                        dist[i][j] = tmp;
                        // "i到j最短路径"对应的路径，经过k
                        path[i][j] = path[i][k];
                    }
                }
            }
        }
        // 打印floyd最短路径的结果
        printf("floyd: \n");
        for (i = 0; i < G.vexnum; i++){
            for (j = 0; j < G.vexnum; j++)
                printf("%2d  ", dist[i][j]);
            printf("\n");
        }
}
```

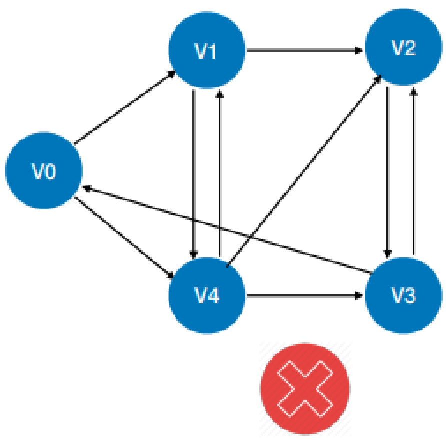
时间复杂度： 总时间复杂度$O(n^3)$，即$O(|V|^3)$。

七、有向无环图

1. 概念：

有向无环图：若一个有向图中不存在环，则称为有向无环图，简称DAG图（Directed Acyclic Graph）





应用1：DAG描述表达式

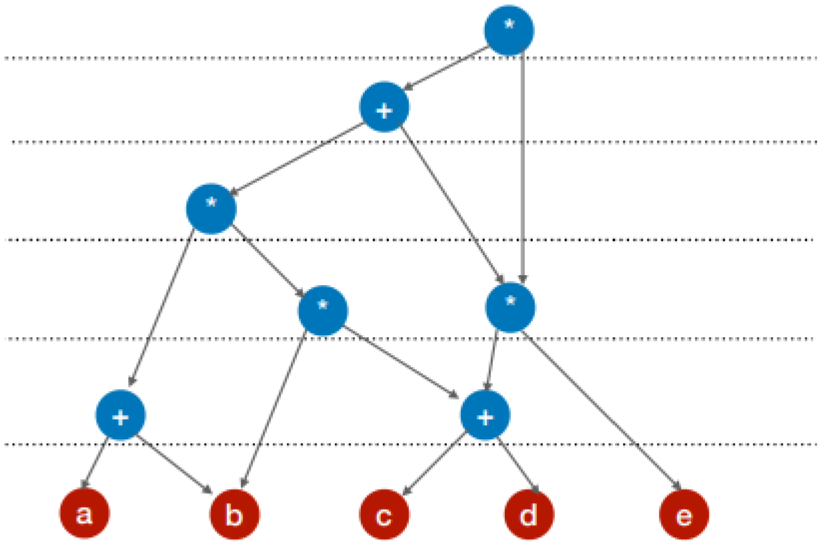
Step 1：把各个操作数不重复地排成一排 a b c d e

Step 2：标出各个运算符的生效顺序（先 后顺序有点出入无所谓）

Step 3：按顺序加入运算符，注意“分层”

Step 4：从底向上逐层检查同层的运算符 是否可以合体

$$((a+b)*(b*(c+d))+(c+d)*e)*((c+d)*e)$$



应用2：拓扑排序

`AOV` 网(Activity On Vertex NetWork，用顶点表示活动的网)： 用DAG图（有向无环图）表示一个工程。顶点表示活动，有向边表示活动Vi必须先于活动Vj进行。

算法步骤:

拓扑排序的实现：

① 从AOV网中选择一个没有前驱（入度为0）的顶点并输出。

② 从网中删除该顶点和所有以它为起点的有向边。

③ 重复①和②直到当前的AOV网为空或当前网中不存在无前驱的顶点为止。

应用3：关键路径