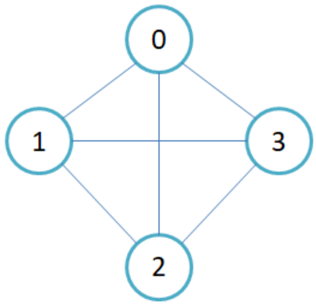
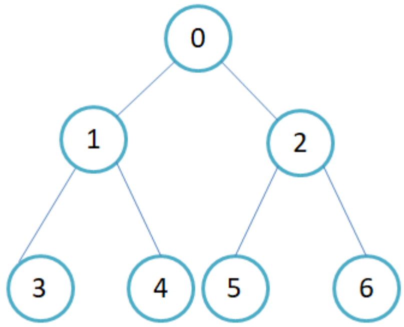
Lesson02---图

1. 图的基本概念

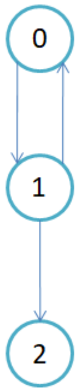
> **图是由顶点集合及顶点间的关系组成的一种数据结构：G = (V， E)**，其中：
>
> **顶点集合V = {x|x属于某个数据对象集}是有穷非空集合**；
>
> **E = {(x,y)|x,y属于V}或者E = {<x, y>|x,y属于V && Path(x, y)}是顶点间关系的有穷集合，也叫做边的集合。**
>
> (x, y)表示x到y的一条双向通路，即(x, y)是无方向的；Path(x, y)表示从x到y的一条单向通路，即Path(x, y)是有方向的。
>
> 顶点和边：**图中结点称为顶点**，第i个顶点记作vi。**两个顶点vi和vj相关联称作顶点vi和顶点vj之间有一条边**，图中的第k条边记作ek，ek = (vi，vj)或<vi，vj>。
>
> 有向图和无向图：**在有向图中，顶点对<x, y>是有序的，顶点对<x，y>称为顶点x到顶点y的一条边(弧)，<x, y>和<y, x>是两条不同的边**，比如下图G3和G4为有向图。在**无向图中，顶点对(x, y)是无序的，顶点对(x,y)称为顶点x和顶点y相关联的一条边，这条边没有特定方向，(x, y)和(y，x)是同一条边**，比如下图G1和G2为无向图。注意：**无向边(x, y)等于有向边<x, y>和<y, x>。**



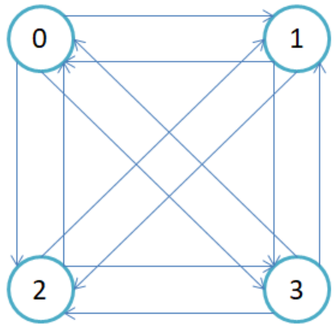
G1



G2



G3



G4

完全图：在**有n个顶点的无向图中**，若**有n * (n-1)/2条边**，即**任意两个顶点之间有且仅有一条边**，则称此图为**无向完全图**，比如上图G1；在**n个顶点的有向图**中，若**有n * (n-1)条边**，即**任意两个顶点之间有且仅有方向相反的边**，则称此图为**有向完全图**，比如上图G4。

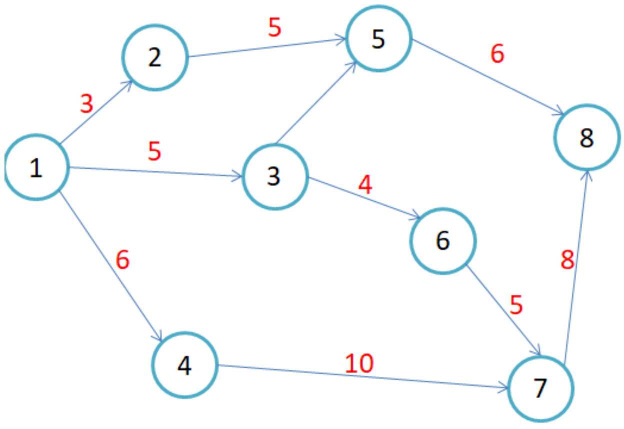
邻接顶点：在**无向图中G中，若(u, v)是E(G)中的一条边，则称u和v互为邻接顶点**，并称**边(u,v)依附于顶点u和v**；在**有向图G中，若<u, v>是E(G)中的一条边，则称顶点u邻接到v，顶点v邻接自顶点u，并称边<u, v>与顶点u和顶点v相关联。**

顶点的度：**顶点v的度是指与它相关联的边的条数，记作deg(v)**。在有向图中，**顶点的度等于该顶点的入度与出度之和**，其中顶点v的**入度是以v为终点的有向边的条数**，记作indev(v);顶点v的**出度是以v为起始点的有向边的条数**，记作outdev(v)。因此：**dev(v) = indev(v) + outdev(v)**。注意：对于**无向图，顶点的度等于该顶点的入度和出度**，即dev(v) = indev(v) = outdev(v)。

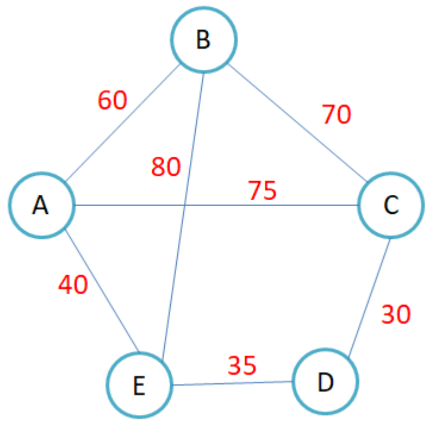
路径：在图G = (V， E)中，若**从顶点vi出发有一组边使其可到达顶点vj，则称顶点vi到顶点vj的顶点序列为从顶点vi到顶点vj的路径。**

路径长度：对于**不带权的图，一条路径的路径长度是指该路径上的边的条数**；对于**带权的图，一条路 径的路径长度是指该路径上各个边权值的总和。**

权值：边附带的数据信息

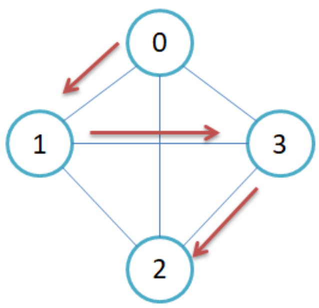


施工进度图

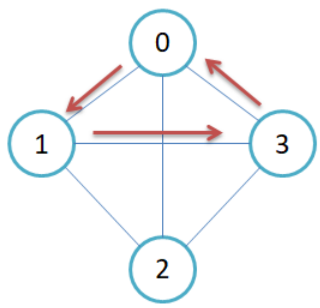


交通网络图

简单路径与回路：**若路径上各顶点v1，v2，v3，...，vm均不重复，则称这样的路径为简单路径。若路 径上第一个顶点v1和最后一个顶点vm重合，则称这样的路径为回路或环。**

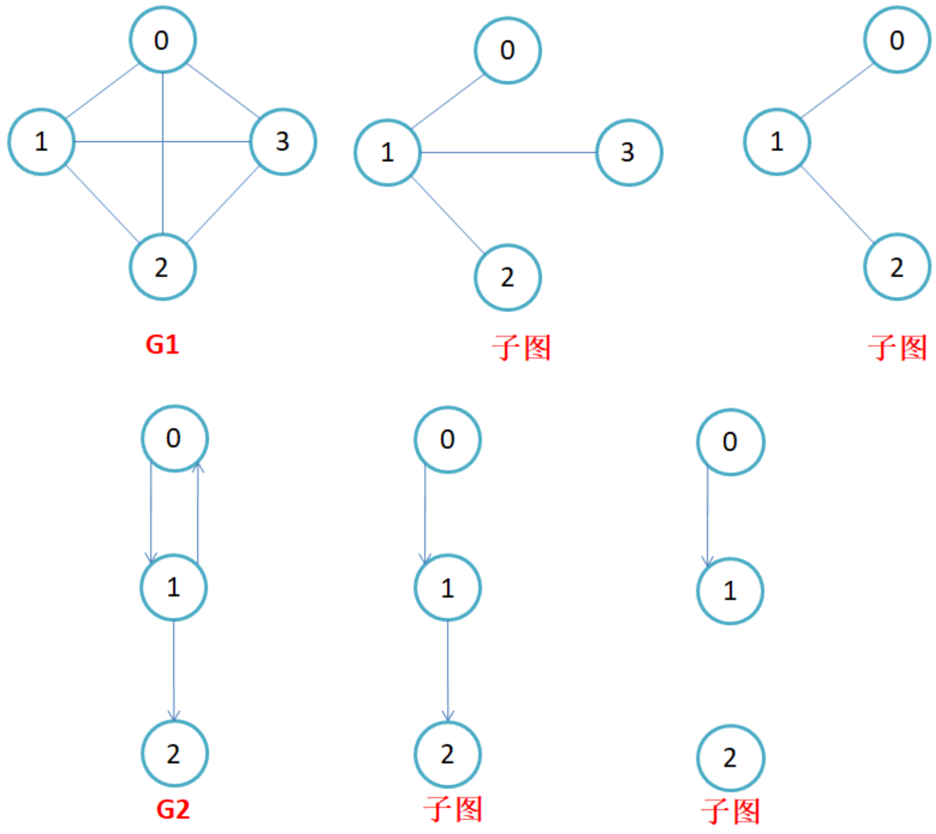


简单路径



回路

子图：**设图G = {V, E}和图G1 = {V1，E1}，若V1属于V且E1属于E，则称G1是G的子图。**



连通图：在**无向图**中，若从顶点v1到顶点v2有路径，则称顶点v1与顶点v2是连通的。**如果图中任意一 对顶点都是连通的，则称此图为连通图。**

强连通图：在**有向图**中，若在**每一对顶点vi和vj之间都存在一条从vi到vj的路径，也存在一条从vj到 vi的路径，则称此图是强连通图。**

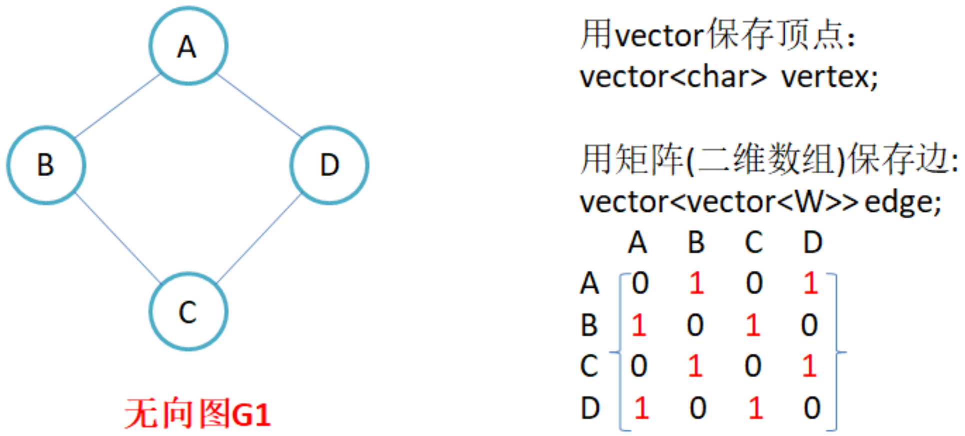
生成树：一个**连通图的最小连通子图**称作该图的生成树。**有n个顶点的连通图的生成树有n个顶点和n- 1条边。**

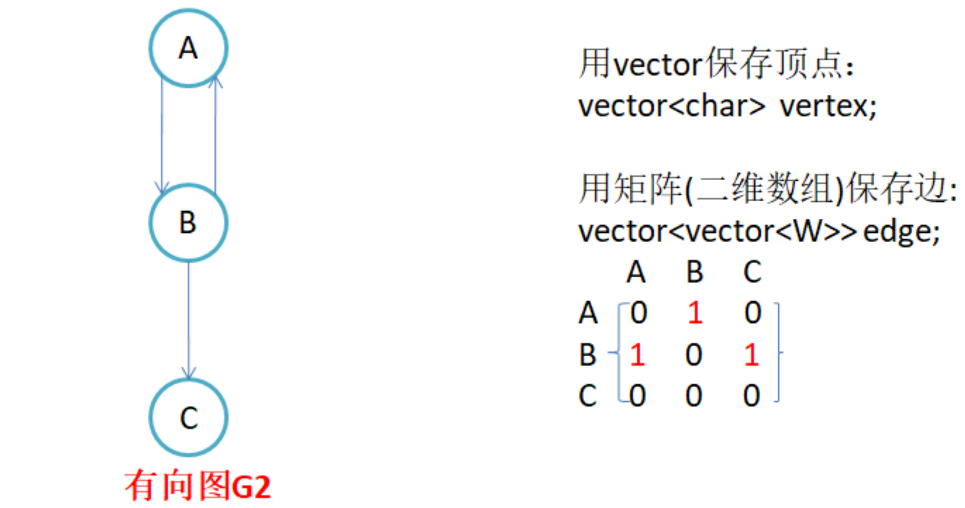
2. 图的存储结构

因为图中既有节点，又有边(节点与节点之间的关系)，因此，**在图的存储中，只需要保存：节点和边关系即可。**节点保存比较简单，只需要一段连续空间即可，那边关系该怎么保存呢?

2.1 邻接矩阵

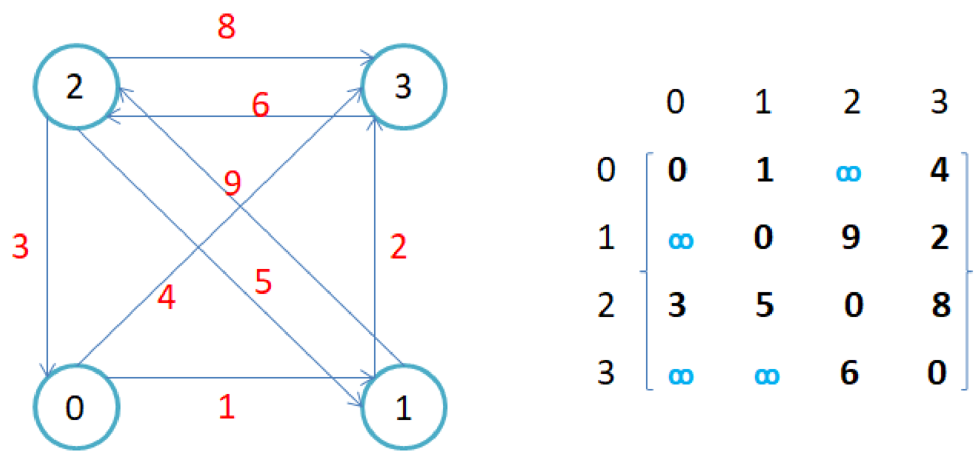
因为节点与节点之间的关系就是连通与否，即为0或者1，因此**邻接矩阵(二维数组)即是：先用一个数组将定点保存，然后采用矩阵来表示节点与节点之间的关系。**





注意:

1. **无向图的邻接矩阵是对称的，第i行(列)元素之和，就是顶点i的度。有向图的邻接矩阵则不一定是对称的，第i行(列)元素之后就是顶点i 的出(入)度。**
2. 如果边带有权值，并且两个节点之间是连通的，上图中的边的关系就用权值代替，如果两个顶点不通，则使用无穷大代替。



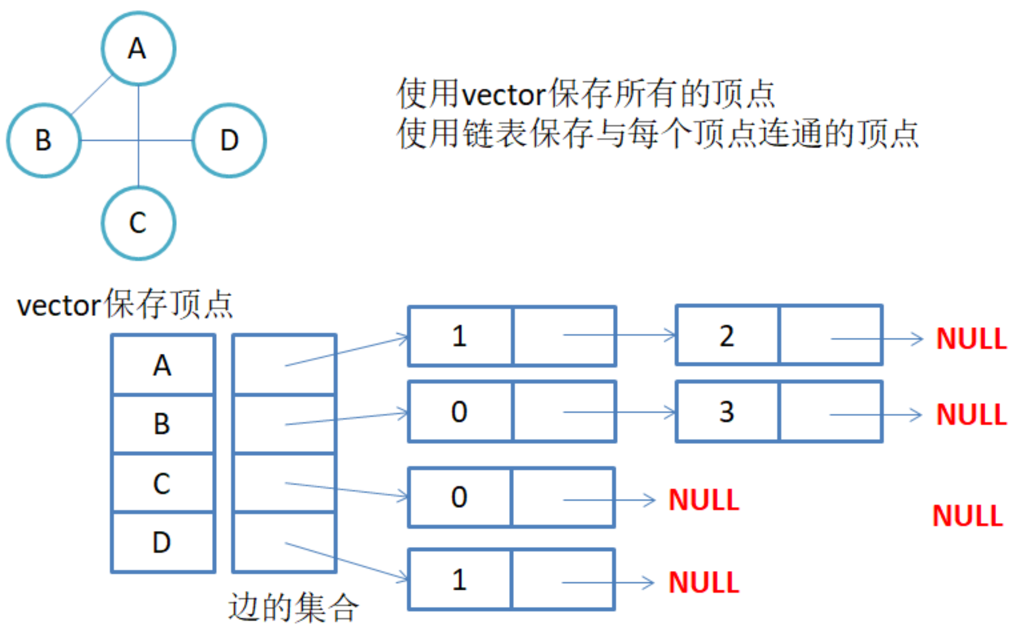
3. 用邻接矩阵存储图的有点是能够快速知道两个顶点是否连通，缺陷是如果顶点比较多，边比较少时，矩阵中存储了大量的0成为系数矩阵，比较浪费空间，并且要求两个节点之间的路径不是很好求。

```
template<class V, class W, bool IsDirect = false>
class Graph
{
public:
```

```
    Graph(V* array, size_t size)
        : _v(array, array+size)
    {
        _edges.resize(size);
        for(size_t i = 0; i < size; ++i)
            _edges[i].resize(size);
    }

    // 获取顶点元素在其数组中的下标
    size_t GetIndexOfV(const V& v)
    {
        for(size_t i = 0; i < _v.size(); ++i)
        {
            if(v == _v[i])
                return i;
        }

        assert(false);
        return -1;
    }

    size_t GetDevOfV(const V& v)
    {
        size_t index = GetIndexOfV(v);
        size_t N = _v.size();
        size_t count = 0;

        // 出度
        for(size_t i = 0; i < N; ++i)
        {
            if(_edges[index][i])
                count++;
        }

        if(IsDirect)
        {
            //有向图---> 入度
            for(size_t i = 0; i < N; ++i)
            {
                if(_edges[i][index])
                    count++;
            }
        }

        return count;
    }

    void AddEdge(const V& v1, const V& v2, const W& weight)
    {
        size_t index1 = GetIndexOfV(v1);
        size_t index2 = GetIndexOfV(v2);

        _edges[index1][index2] = weight;
        if(!IsDirect)
            _edges[index2][index1] = weight;
    }

    void PrintGraph()
```

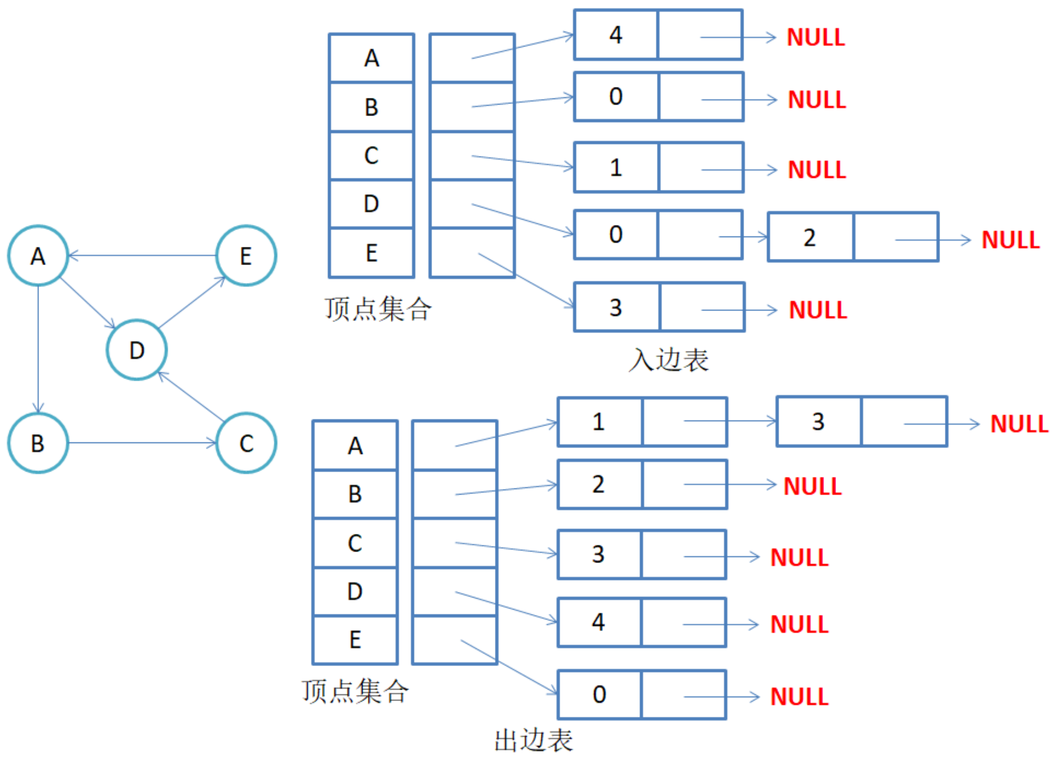
```
    {
        size_t N = _v.size();
        for(size_t i = 0; i < N; ++i)
            cout<<_v[i]<<" ";
        cout<<endl;

        for(size_t i = 0; i < N; ++i)
        {
            for(size_t j = 0; j < N; ++j)
            {
                printf("%2d ", _edges[i][j]);
            }

            cout<<endl;
        }
    }

private:
    vector<V> _v;
    vector<vector<W>> _edges;
};

/*
测试用例：
"ABCDE"
'A', 'D' , 10
'A', 'E' , 20
'B', 'C' , 10
'B', 'D' , 20
'B', 'E' , 30
'C', 'E' , 40
*/
```

2.2 邻接表

> 邻接表：使用数组表示顶点的集合，**使用链表表示边的关系。**
>
> 1. 无向图邻接表存储
>
> 
>

注意：**无向图中同一条边在邻接表中出现了两次。如果想知道顶点vi的度，只需要知道顶点vi边链表集合中结点的数目即可。**

2. 有向图邻接表存储



注意：有向图中每条边在邻接表中只出现一次，与顶点vi对应的邻接表所含结点的个数，就是该顶点的出度，也称出度表，要得到vi顶点的入度，必须检测其他所有顶点对应的边链表，看有多少边顶点的dst取值是i。

```
template<class W>
struct LinkEdge
{
    LinkEdge<W>* _pNext;
    W _weight;      // 节点的权值
    size_t _src;    // 起点的下标(后面使用)
    size_t _dst;    // 终点的下标

    LinkEdge(size_t src, size_t dst, const W& weight)
        : _src(src)
        , _dst(dst)
        , _weight(weight)
        , _pNext(NULL)
    {}
};

template<class V, class W, bool IsDirect = false>
class Graph
{
    typedef LinkEdge<W> LinkEdge;
    typedef Graph<V, W, IsDirect> Self;
public:
    Graph(const V* array, size_t size)
        : _v(array, array+size)
    {
        _linkEdges.resize(size);
    }
```

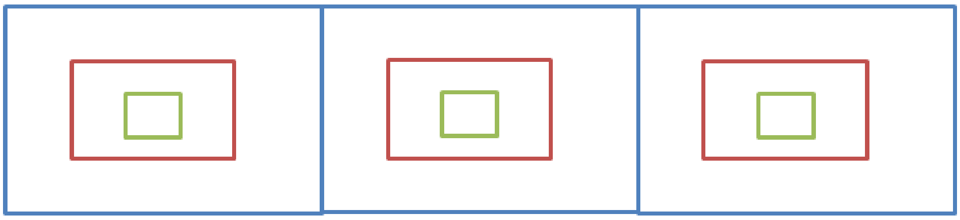
```
    // g.AddEdge('A', 'D', 10);
    void AddEdge(const V& v1, const V& v2, const W& weight)
    {
        size_t src = GetIndexOfV(v1);
        size_t dst = GetIndexOfV(v2);

        _AddEdge(src, dst, weight);
        if(!IsDirect)
            _AddEdge(dst, src, weight);
    }

    // 获取顶点元素在其数组中的下标
    size_t GetIndexOfV(const V& v)
    {
        for(size_t i = 0; i < _v.size(); ++i)
        {
            if(v == _v[i])
                return i;
        }

        assert(false);
        return -1;
    }

    void PrintGraph()
    {
        for(size_t index = 0; index < _v.size(); ++index)
        {
            cout<<"v["<<_v[index]<<"]--->";

            LinkEdge* pCur = _linkEdges[index];
            while(pCur)
            {
                cout<<"v["<<_v[pCur->_dst]<<"]--->";
                pCur = pCur->_pNext;
            }
            cout<<"NULL"<<endl;
        }
    }

    int GetVDev(const V& v)
    {
        size_t index = GetIndexOfV(v);
        LinkEdge* pCur =_linkEdges[index];
        size_t count = 0;

        // 出度
        while(pCur)
        {
            count++;
            pCur = pCur->_pNext;
        }

        if(IsDirect)
        {
            // 入度
            int dst = index;
            for(size_t src = 0; src < _v.size(); ++src)
```

```cpp
            {
                if(src == dst)
                    continue;
                else
                {
                    LinkEdge* pCur = _linkEdges[src];
                    while(pCur)
                    {
                        if(pCur->_dst == dst)
                            count++;

                        pCur = pCur->_pNext;
                    }
                }
            }
        }

        return count;
    }

private:
    void _AddEdge(size_t src, size_t dst, const W& weight)
    {
        LinkEdge* pCur = _linkEdges[src];
        // 检测当前边是否存在
        while(pCur)
        {
            if(pCur->_dst == dst)
                return;

            pCur = pCur->_pNext;
        }

        LinkEdge* pNewNode = new LinkEdge(src, dst, weight);
        pNewNode->_pNext = _linkEdges[src];
        _linkEdges[src] = pNewNode;
    }

private:
    vector<V> _v;
    vector<LinkEdge*> _linkEdges;
};
```

3. 图的遍历

> **给定一个图G和其中任意一个顶点v0，从v0出发，沿着图中各边访问图中的所有顶点，且每个顶点仅被遍历一次。"遍历"即对结点进行某种操作的意思。** 请思考树以前是怎么遍历的，此处可以直接用来遍历图吗？为什么？

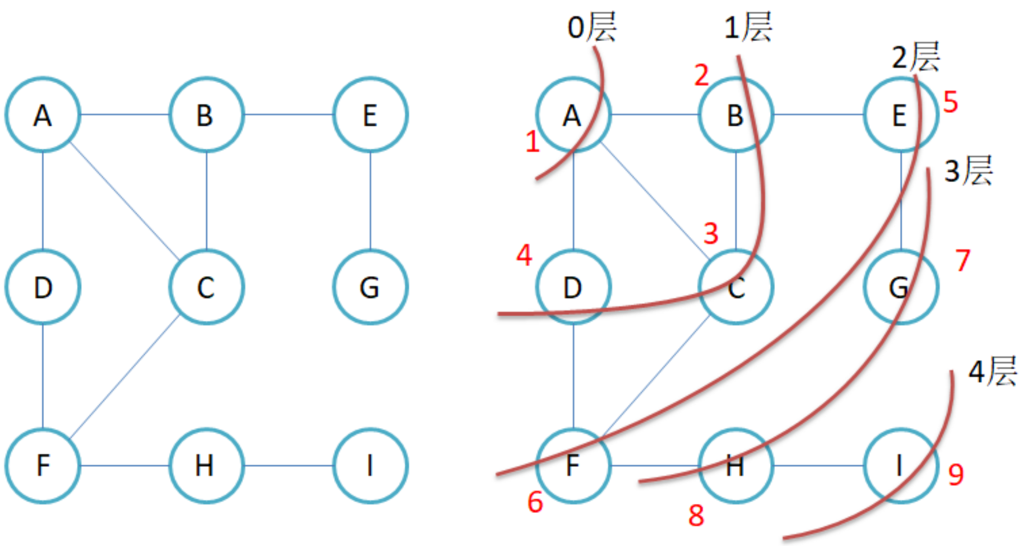
3.1 图的广度优先遍历



比如现在要找东西，假设有三个抽屉，东西在那个抽屉不清楚，现在要将其找到，广度优先遍历的做法是：

1. 先将三个抽屉打开，在最外层找一遍
2. 将每个抽屉中红色的盒子打开，再找一遍
3. 将红色盒子中绿色盒子打开，再找一遍

直到找完所有的盒子，注意：每个盒子只能找一次，不能重复找

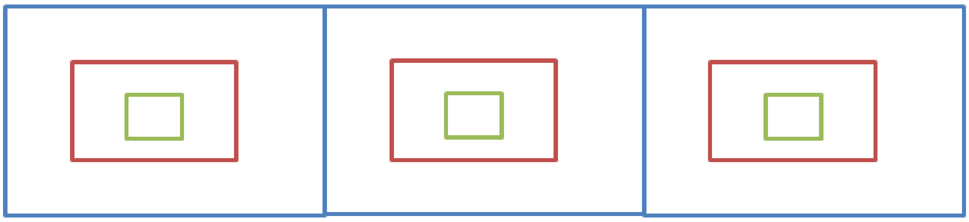


问题：如何防止节点被重复遍历

```
void _BFS(queue<int>& q, vector<bool>& visited)
{
    while(!q.empty())
    {
        size_t index = q.front();
        if(!visited[index])
        {
            cout<<_v[index]<<" ";
            visited[index] = true;
            LinkEdge* pCur = _linkEdges[index];
            while(pCur)
            {
                q.push(pCur->_dst);
                pCur = pCur->_pNext;
            }
        }

     q.pop();
    }

    cout<<endl;
}
```

```
void BFS(const V& v)
{
    queue<int> q;
    vector<bool> visited(_v.size(), false);
    q.push(GetIndexOfV(v));
    _BFS(q, visited);

 // 处理非连通图
    for(size_t index = 0; index < _v.size(); ++index)
    {
        if(visited[index])
            continue;

        q.push(index);
        _BFS(q, visited);
    }
}
```

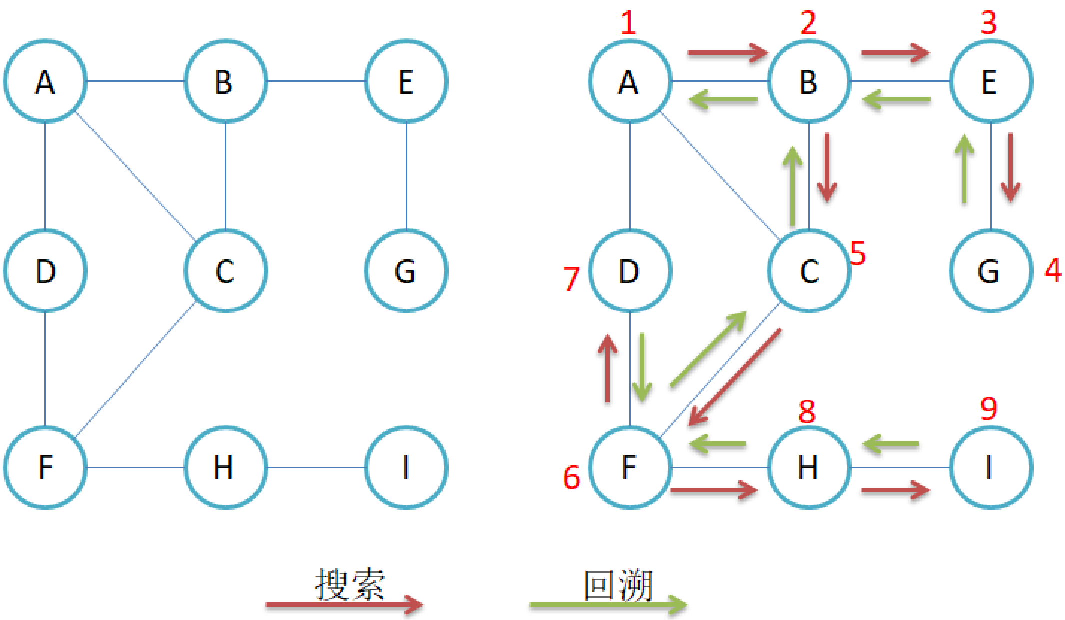
3.2 图的深度优先遍历



比如现在要找东西，假设有三个抽屉，东西在那个抽屉不清楚，现在要将其找到，广度优先遍历的做法是：

1. 先将第一个抽屉打开，在最外层找一遍
2. 将第一个抽屉中红盒子打开，在红盒子中找一遍
3. 将红盒子中绿盒子打开，在绿盒子中找一遍
4. 递归查找剩余的两个盒子

深度优先遍历：将一个抽屉一次性遍历完(包括该抽屉中包含的小盒子)，再去递归遍历其他盒子




```
void _DFS(int index, vector<bool>& visited)
```

```
{
    if(!visited[index])
    {
        cout<<_v[index]<<" ";
        visited[index] = true;

        LinkEdge* pCur = _linkEdges[index];
        while(pCur)
        {
            _DFS(pCur->_dst, visited);
            pCur = pCur->_pNext;
        }
    }
}

void DFS(const V& v)
{
    cout<<"DFS:";
    vector<bool> visited(_v.size(), false);
    _DFS(GetIndexOfV(v), visited);

    for(size_t index = 0; index < _v.size(); ++index)
        _DFS(index, visited);

    cout<<endl;
}
```

4. 最小生成树

> 连通图中的每一棵生成树，都是原图的一个极大无环子图，即：**从其中删去任何一条边，生成树就不在连通；反之，在其中引入任何一条新边，都会形成一条回路。**
>
> **若连通图由n个顶点组成，则其生成树必含n个顶点和n-1条边**。因此构造最小生成树的准则有三条：
>
> 1. **只能使用图中的边来构造最小生成树**
> 2. **只能使用恰好n-1条边来连接图中的n个顶点**
> 3. **选用的n-1条边不能构成回路**
>
> 构造最小生成树的方法：**Kruskal算法**和**Prim算法**。这两个算法都采用了**逐步求解的贪心策略。**
>
> 贪心算法：是指在问题求解时，总是做出当前看起来最好的选择。也就是说贪心算法做出的不是整体 最优的的选择，而是某种意义上的局部最优解。贪心算法不是对所有的问题都能得到整体最优解。

4.1 Kruskal算法

> 任**给一个有n个顶点的连通网络N={V,E}**，
>
> **首先构造一个由这n个顶点组成、不含任何边的图G={V,NULL}**，其中每个顶点自成一个连通分量，
>
> **其次不断从E中取出权值最小的一条边(若有多条任取其一)，若该边的两个顶点来自不同的连通分量，则将此边加入到G中。**
>
> 如此重复，直到所有顶点在同一个连通分量上为止。
>
> **核心：每次迭代时，选出一条具有最小权值，且两端点不在同一连通分量上的边，加入生成树。**

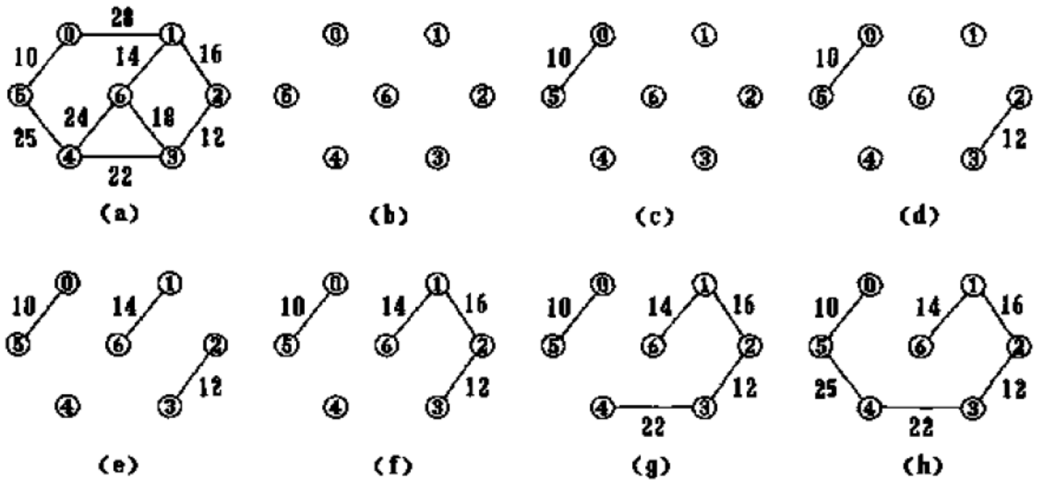


图 8.17 应用克鲁斯卡尔算法构造最小生成树的过程

4.2 Prime算法

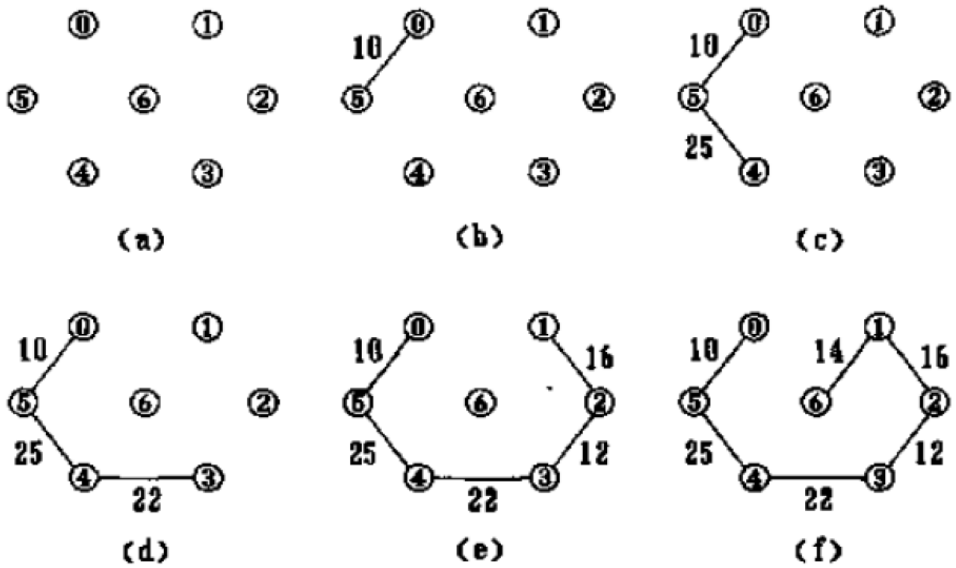
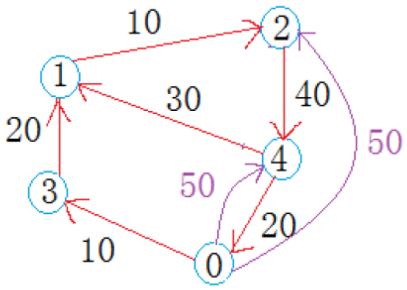


图 8.19 用普里姆算法构造最小生成树的过程

5. 单元最短路径

> 最短路径问题：从在带权图的某一顶点出发，找出一条通往另一顶点的最短路径，最短也就是沿路径各边的权值总和达到最小。

A	B	C	D	E
0	1	2	3	4



A[0]->50[4]->50[2]->10[3]->NULL
B[1]->10[2]->NULL
C[2]->40[4]->NULL
D[3]->20[1]->NULL
E[4]->30[1]->20[0]->NULL

	0	1	2	3	4
dist	#	#	50	10	50
path	0	0	0	0	0

0,1的路径为:0->3->1->路径长度为：30
0,2的路径为:0->3->1->2->路径长度为：40
0,3的路径为:0->3->路径长度为：10
0,4的路径为:0->4->路径长度为：50