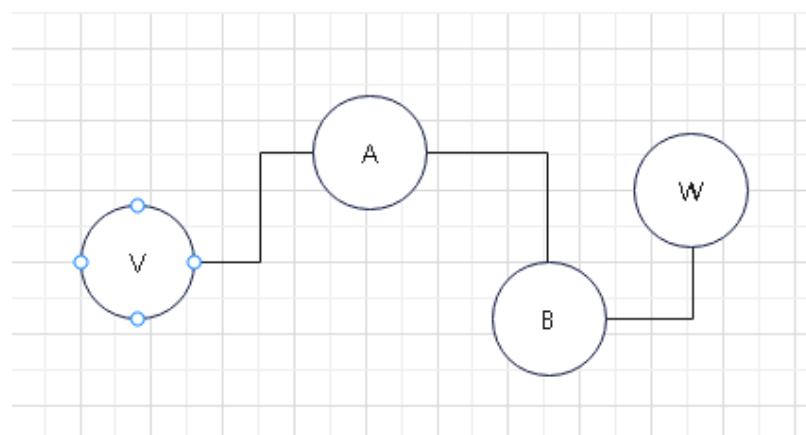


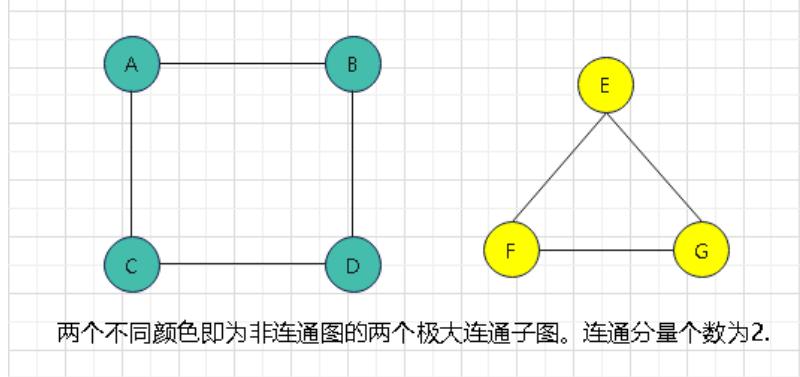
# 第六章 图

## • 一.图的定义

- 有限个顶点和有限条边组成
- $G = (V, E)$ 
  - $|V|$  顶点个数
  - $|E|$  边的条数
- 1.有向图
- 2.无向图
- 3.简单图
  - 图中不存在重复边，不存在顶点到自己的边
- 4.多重图
  - 存在重复边，存在顶点到自己的边
- 5.完全图（简单完全图）
  - 定义：
    - 任意两个顶点之间都存在边
  - 无向完全图
    - 一共有  $n(n-1)/2$  条边
  - 有向完全图
    - 共有  $n(n-1)$  条边
- 6.子图
  - 生成子图：顶点集相同的子图
- 7.连通、连通图和连通分量（无向图）
  - 连通：顶点  $v$  到顶点  $w$  有路径存在，则称  $v$  和  $w$  是连通的。



- 连通图
  - 若图中任意两个顶点都是连通的，则称图G为连通图
- 连通分量
  - 无向图中的极大连通子图称为连通分量



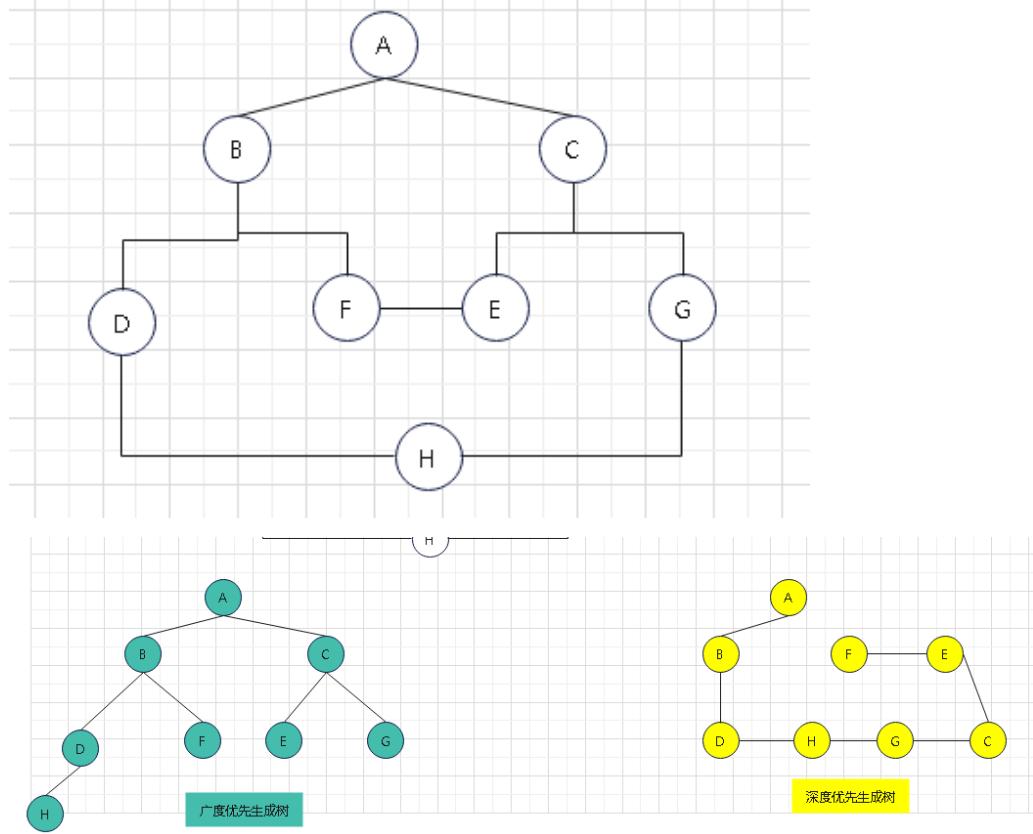
- 极大连通子图：
  - 1. 定义：再加入任何一个顶点，子图都不会再连通
  - 2. 连通图中只有一个极大连通子图，就是它自己
  - 3. 非连通图中有多个极大连通子图。

## • 8. 强连通图、强连通分量（有向图）

- 强连通：如果有一对顶点A,B；A到B和B到A都有路径，则称这两个顶点是强连通的。
- 强连通图：如果图中任何一对顶点都是强连通的，则称此图是强连通图。
- 强连通分量：有向图中的极大强连通子图称为有向图的强连通分量。

## • 9. 生成树、生成森林

- 生成树：
  - 如果连通图中包含所有顶点的一个极小连通子图称为生成树。
  - 图的生成树不唯一。从不同的顶点进行遍历，结果会不一样。
  - 理解：
    - 生成树就是连通所有顶点且不产生回路的所有子图。
    - 连通图中所有顶点数为n，边数为n-1的子图都是生成树。
  -



- 生成森林：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。

## • 10.顶点的度、入度、出度

- 1.顶点的度：以该顶点为一个端点的边的数目
- 2.无向图中：全部顶点的度之和 = 边数 \* 2
- 3.有向图中：一个顶点的度=入度+出度；
- 4.有向图中全部顶点的入度之和=出度之和=边数

## • 11.边的权和网

- 带权图：边上带有权值的图

## • 12.稠密图、稀疏图

- $|E| < |V| \log |V|$  稀疏图，边数很少
- 反之 稠密图 边数很多。

## • 13.路径、路径长度和回路

- 回路：第一个顶点和最后一个顶点相同的路径

## • 14.简单路径、回路

- 简单路径：顶点不重复出现的路径
- 简单回路：除第一个顶点和最后一个顶点，其余顶点不重复出现的回路

## • 15 距离

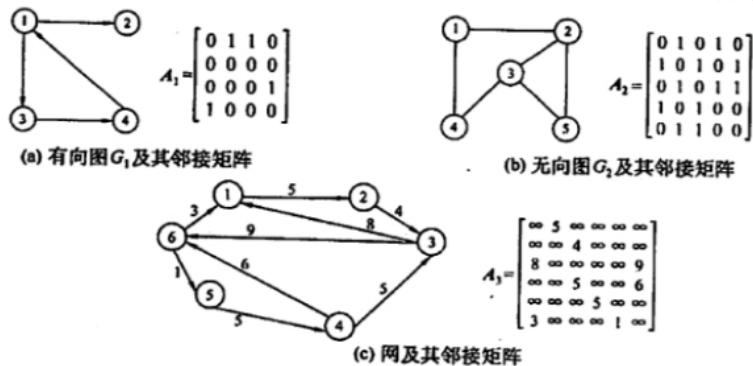
## • 16.有向树

- 一个顶点的入度为0，其余顶点的入度均为1的有向图称为有向树。

## 二、图的存储及基本操作

### 1. 邻接矩阵法

- 1. 定义：一个一维数组存储图中顶点的信息，用一个二维数组存储图中边的信息。



- 2. 存储结构：

```
#define MaxNum 100 //顶点数目的最大值
#define INFINITY 65535 // 表示权值的无穷
typedef char VertexType; // 顶点的数据类型
typedef int EdgeType; // 边的数据类型
typedef struct {
    VertexType Vex[MaxNum]; // 顶点表
    EdgeType Edge[MaxNum][MaxNum]; // 邻接矩阵
    int vexnum, edgenum; // 图当前的顶点数和边数
} MGraph;
```

- 邻接矩阵的创建

```

void createMGraph(MGraph *mGraph){
    cout<<"输入顶点数和边数(使用空格分隔)"<<endl;
    cin>>mGraph->vexnum>>mGraph->edgenum;
    for (int i = 0; i < mGraph->vexnum; ++i) {
        cout<<"请输入第" <<i+1<<"个顶点名"<<endl;
        cin>>mGraph->Vex[i];
    }

    // 邻接矩阵初始化
    for (int i = 0; i < mGraph->vexnum; ++i) {
        for (int j = 0; j < mGraph->vexnum; ++j) {
            if (i == j)
                mGraph->Edge[i][j] = 0;
            mGraph->Edge[i][j] = INFINITY;
        }
    }
    // 对邻接矩阵赋值
    for (int i = 0; i < mGraph->edgenum; ++i) {
        cout<<"请输入边(vi, vj)的i, j和权值w(空格分隔):"<<endl;
        int edge_i,edge_j,weight = 0;
        cin>>edge_i>>edge_j>>weight;
        mGraph->Edge[edge_i][edge_j] = weight;
        mGraph->Edge[edge_j][edge_i] = mGraph->Edge[edge_i][edge_j]; // 无向图 对称
    }
}

```

- 特点：

- 1.无向图的邻接矩阵一定是一个对称矩阵。
- 2.无向图的第*i*行/列非零元素的个数为顶点*i*的度。
- 3.有向图的第*i*行/列非零元素的个数为顶点*i*的出度/入度。
- 4.使用邻接矩阵法存储图，很容易确定图中任意两个顶点之间是否有边相连。
- 5.稠密图适合使用邻接矩阵法进行存储。
- 6. $A^n[i][j] = i$ 到*j*长度为*n*路径的数量

- 2.邻接表法

- 1.定义：对图G每一个顶点都建立一个单链表，单链表中存储的都是与该顶点相关联的顶点。

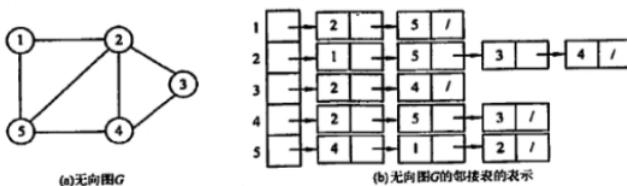
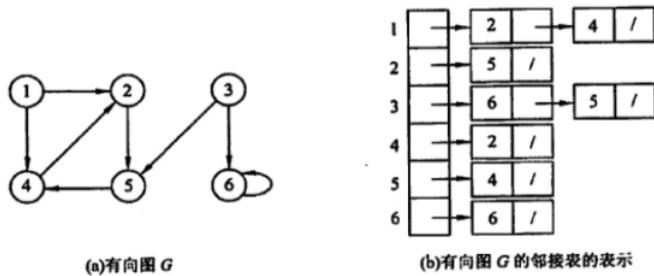


图 6.7 无向图邻接表示法实例



- 2.存储结构

```

#define MaxNum 100 //顶点数目的最大值
#define INFINITY 65535 // 表示权值的无穷
typedef char VertexType; // 顶点的数据类型
typedef int EdgeType; // 边的数据类型

typedef struct ArcNode{ // 边表结点
    int adjvex; // 邻接域, 记录该点的下标, i->j ,也就是记录j的下标
    struct ArcNode *next;
    // EdgeType weight; 存储权值
}ArcNode;
typedef struct VNode{ // 顶点表结点
    VertexType data; //顶点名称
    ArcNode *first; // 第一个指向其他结点的 弧
}VNode,AdjList[MaxNum];
typedef struct{
    AdjList vertices; //邻接表
    int vexnum,arcnum; //图的顶点数和弧数
}ALGraph; // 以邻接表存储的图

```

- 创建邻接表 图

```

// 2022-07-26-14
void createALGraph(ALGraph *alGraph){
    int v_i,v_j = 0;
    ArcNode *arcNode;
    cout<<"输入顶点数 边数"<<endl;
    cin>>alGraph->vexnum>>alGraph->arcnum;
    // 输入顶点 并对 顶点的 first进行初始化
    for (int i = 0; i < alGraph->vexnum; ++i) {
        cout<<"输入第"<<i+1<<"个顶点信息"<<endl;
        cin>>alGraph->vertices[i].data;
        alGraph->vertices[i].first = NULL;
    }
}

```

```

    // 建立边表
    for (int i = 0; i < alGraph->arcnum; ++i) {
        cout<<"输入边(vi,vj)的顶点序号i,j(空格分隔) :"<<endl;
        cin>>v_i>>v_j;
        arcNode = (ArcNode *)malloc(sizeof(ArcNode));
        arcNode->adjvex = v_j;
        // 使用头插法 将当前i所指的first -> arcNode , arcNode -> first
        arcNode->next = alGraph->vertices[v_i].first;
        alGraph->vertices[v_i].first = arcNode;

        // 无向图为对称的 i->j j->i, 若为有向图 则可省略以下步骤
        arcNode = (ArcNode *)malloc(sizeof(ArcNode));
        arcNode->adjvex = v_i;
        arcNode->next = alGraph->vertices[v_j].first;
        alGraph->vertices[v_j].first = arcNode;
        free(arcNode);
    }
}

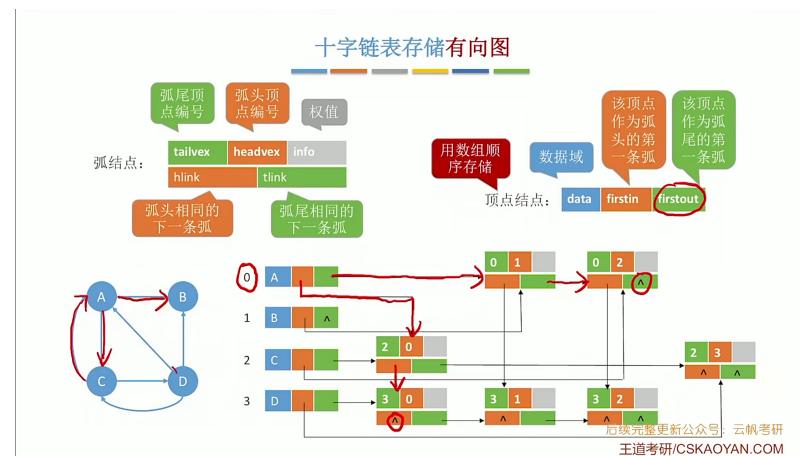
```

- 3.特点

- 1.无向图：存储空间 $O(|V|+2|E|)$ ; 有向图  $O(|V|+|E|)$
- 稀疏图用邻接表更节省空间

- 3.十字链表（有向图）

- 十字链表是为了便于求得图中顶点的度（出度和入度）而提出来的。简单的来理解，十字链表就是在邻接表的基础上加了一个前置指针，使得我们可以在表中既能找到某一顶点的出度也能找到该顶点的入度。



在十字链表存储结构中，有向图中的顶点的结构如下所示：

data	firstIn	firstOut
------	---------	----------

其中data表示顶点的具体数据信息，而firstIn则表示指向以该顶点为弧头的第一个弧节点。而firstOut则表示指向以该顶点为弧尾的第一个弧节点。为了表示有向图中所有的顶点，采用一个顶点数组存储每一个结点，如下图所示。

xList[0]	data	firstIn	firstOut	第一个顶点
xList[1]	data	firstIn	firstOut	第二个顶点
...	...	...	...	...
xList[num]	data	firstIn	firstOut	第 num 个顶点

顶点数组 xList[num]

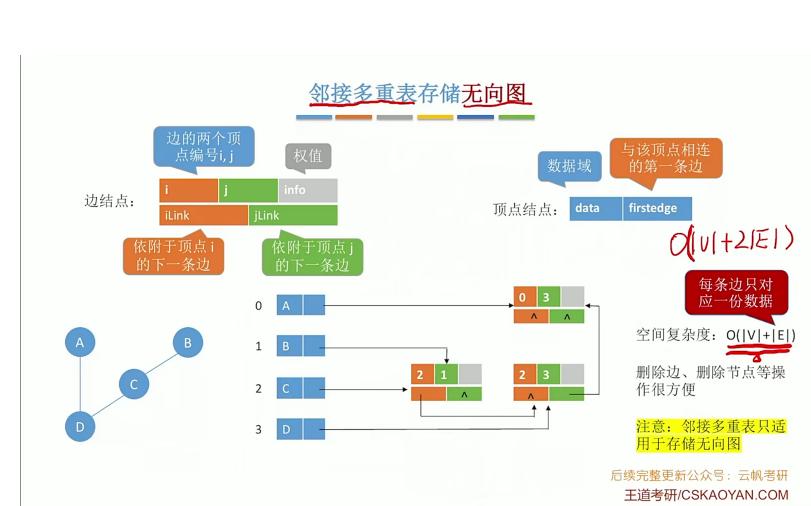
另外，在十字链表存储结构中，有向图中的每一条弧都有一个弧结点与之对应，具体的弧结点结构如下所示：

tailVex	headVex	hLink	tLink	Info
---------	---------	-------	-------	------

其中的tailVex表示该弧的弧尾顶点在顶点数组xList中的位置，headVex表示该弧的弧头顶点在顶点数组中的位置。hLink则表示指向弧头相同的下一条弧，tLink则表示指向弧尾相同的下一条弧。

- 特点：
  - 1. 极容易找到vi为尾的弧，也容易找到vi为头的弧，容易求得顶点出度/入度。
  - 表示不唯一，但一个十字链表确定一个图。

## 4. 邻接多重表（无向图）



## 5. 图的基本操作

图的基本操作主要包括（仅抽象地考虑，故忽略掉各变量的类型）：

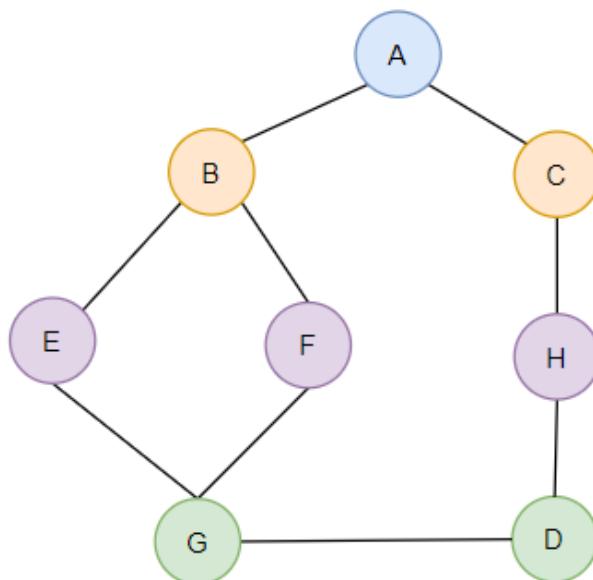
- `Adjacent(G, x, y)`: 判断图  $G$  是否存在边 $\langle x, y \rangle$ 或 $(x, y)$ 。
- `Neighbors(G, x)`: 列出图  $G$  中与结点  $x$  邻接的边。
- `InsertVertex(G, x)`: 在图  $G$  中插入顶点  $x$ 。
- `DeleteVertex(G, x)`: 从图  $G$  中删除顶点  $x$ 。
- `AddEdge(G, x, y)`: 若无向边 $\langle x, y \rangle$ 或有向边 $\langle x, y \rangle$ 不存在，则向图  $G$  中添加该边。
- `RemoveEdge(G, x, y)`: 若无向边 $\langle x, y \rangle$ 或有向边 $\langle x, y \rangle$ 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  $y$  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 $\langle x, y \rangle$ 或 $\langle x, y \rangle$ 对应的权值。
- `Set_edge_value(G, x, y, v)`: 设置图  $G$  中边 $\langle x, y \rangle$ 或 $\langle x, y \rangle$ 对应的权值为  $v$ 。

## • 三、图的遍历

### • 1. 广度优先搜索 (BFS)

#### • 1. 定义：

- 类似二叉树的层次遍历算法，是一种分层的查找过程。
- 广度优先算法可以解决两类问题：
  - 1. 从顶点A出发有前往顶点B的路径吗？
  - 2. 从顶点A出发前往顶点B的路径哪条最短？
- 



图的广度优先搜索就如同树的层次遍历一般，将最近的顶点入队，访问后出队并将出队顶点最近的顶点进队。

#### • 2. 代码逻辑

-

```

void bfs(起始点) {
    将起始点放入队列中;
    标记起点访问;
    while (如果队列不为空) { // 一般采用while，当然也可以使用递归
        访问队列中队首元素x;
        删除队首元素;
        for (x 所有相邻点) {
            if (该点未被访问过且合法) {
                将该点加入队列末尾;
                if (该结点是目标状态) { // 达到目标，提前结束终止循环
                    置 flag= true;
                    break;
                }
            }
        }
    }
    队列为空，广搜结束;
}

```

### ● 3.代码实现

```

●
void BFS_fun(MGraph graph){
    for (int i = 0; i < graph.verNum; ++i) {
        visited[i] = false;
    }
    InitQueue(& queue);
    // 从第一个顶点进行广度优先遍历
    for (int i = 0; i < graph.verNum; ++i) {
        if (!visited[i]){
            // 如果没有被访问过
            BFS_core(graph,i); // 进行访问 并将邻接点放入队列中
        }
    }
}

void BFS_core(MGraph graph,int v){
    visit(v);
    visited[v] = true;
    EnQueue(& queue,v); // 顶点入队
    while (!isEmpty(queue)){ // 队列非空 就一直进行循环
        DeQueue(& queue, & v); // 先进入的顶点出队
        for (int w =FirstNeighbor(graph,v);w>=0;w = NextNeighbor(graph,v,w)){
            if (!visited[w]){// 如果邻接点w没有被访问过
                visit(w); // 访问w
                visited[w] = true;
                EnQueue(& queue,w); // 入队
            }
        }
    }
}

```

### ● 4.性能分析

- 1.空间复杂度  $O(|V|)$  (借助辅助队列，顶点都入队一次)
- 2.时间复杂度

- 邻接矩阵:  $O(|V|^2)$
- 邻接表:  $O(|V| + |E|)$

## • 2.深度优先搜索 (DFS)

### • 1.定义

- DFS类似于树的先序遍历。我们选定一个开始顶点后，就从该顶点开始，顺着一个邻接顶点一条路走到黑。走到最后再往回走，从而达到全图遍历。

### • 2.代码实现

#### • 1.递归

```
void DFSTraverse(MGraph graph){
    for (int i = 0; i < graph.verNum; ++i) {
        visited[i] = false;
    }
    for (int i = 0; i < graph.verNum; ++i) {
        if (!visited[i]){
            DFS(graph,i);
        }
    }
}

void DFS(MGraph graph,int v){
    visit(v);
    visited[v] = true;
    for (int w= FirstNeighbor(graph,v);w>=0;w = NextNeighbor(graph,v,w)){
        if (!visited[w]){
            DFS(graph,w);
        }
    }
}
```

#### • 2.非递归

```
void DFS_Non_Recursion(MGraph graph,int v){
    int k = 0;
    for (int i = 0; i < graph.verNum; ++i) {
        visited[i] = false;      // visited 数组 初始化
    }
    InitStack(stack);

    Push(stack, &v);
    visit(v);
    visited[v] = true;
    while (!isEmpty(stack)){
        Pop(stack, &k);    // 栈中顶点退出
        visit(k);
        visited[k] = true;
```

```

    3     for (int w = FirstNeighbor(graph,k); w>=0; w = NextNeighbor(graph,k,w)) {
    3         if (!visited[w]){
    3             visit(w);
    3             visited[w] = true;
    3             Push(stack, &w);
    3         }
    3     }
    3 }
    3 Pop(stack, &v);
}

```

- 3.性能分析

- 空间复杂度:  $O(|V|)$
- 时间复杂度
  - 邻接矩阵:  $O(|V|^2)$
  - 邻接表:  $O(|V| + |E|)$

- 3.课后代码题(P217)

- 1.以图判树

- 解题思路

```

/*
 * 以图判树可以从两个方面入手:
 * 1. 连通图、没有回路    -- 使用DFS 可以判断是否存在回路
 * 2. 图为最小生成树 即 n个顶点 n-1条边
 * 树需要是连通的图, 所以需要一次DFS就能遍历完所有顶点
 * 我们使用DFS |进行遍历操作, 在基础的DFS函数上 增加两个参数 vexCount,arcCount来统计顶点个数和边的个数
 * 由于在递归中会存在回溯 也就会产生 1 -> 2 && 2 -> 1 , 也就是每条边我们都算了两次
 * 最终, 我们需要判断的条件也就是 alGraph.vexNum == vexCount && alGraph.arcNum == 2*(alGraph.vexNum-1)
 * @param alGraph 邻接表的图
 * @return 是否为树
 */


```

- 实现代码

```

bool isTree(ALGraph alGraph){
    int vexCount = 0,arcCount = 0;
    for (int i = 0; i < alGraph.vexNum; ++i) {
        visited[i] = false;
    }
    DFS(alGraph, v: 0, &vexCount, &arcCount,visited); // DFS 进行统计 顶点个数 和边的个数
    // DFS 结束后 进行最终判断
    if (alGraph.vexNum == vexCount && arcCount == 2*(alGraph.vexNum - 1))
        return true;
    return false;
}


```

```

void DFS(ALGraph graph, int v, int &vexCount, int &arcCount, int visitedList[]) {
    // visit(v);
    visited[v] = true;
    // 成功访问一个顶点，不会再有第二次访问，vexCount++;
    vexCount++;
    for (int w = FirstNeighbor(graph, v); w >= 0; w = NextNeighbor(graph, v, w)) {
        // 不管 邻接点是否被访问 都进行了 边的访问
        arcCount++;
        if (!visited[w]) {
            DFS(graph, w, &vexCount, &arcCount, visited);
        }
    }
}

```

- 2. 分别采用深度优先遍历和广度优先遍历判断以邻接表方式存储的有向图是否存在有顶点  $v_i$  到  $v_j$  的路径。

- 1.BFS

- 思路：使用一次层次遍历，如果可以找到  $j$  的坐标则说明  $i \rightarrow j$  之间有路径 否则没有
- 代码：

```

bool BFS_Path(ALGraph alGraph, int i, int j) {
    int v = 0;
    for (int k = 0; k < alGraph.vexNum; ++k) {
        visited[k] = false;
    }
    InitQueue(queue); // 初始化队列
    EnQueue(queue, i); // 顶点入队
    while (!isEmpty(queue)) {
        DeQueue(queue, &v); // 顶点出队
        visited[v] = true;
        if (v == j)
            return true;

        // 找v 的邻接点
        for (int w = FirstNeighbor(alGraph, v); w >= 0; w = NextNeighbor(alGraph, v, w)) {
            // 邻接点 下标 == j
            if (w == j)
                return true;
            if (!visited[w]) {
                EnQueue(queue, w);
                visited[w] = true;
            }
        }
    }
    // 没有找到
    return false;
}

```

- 2.DFS

- 代码

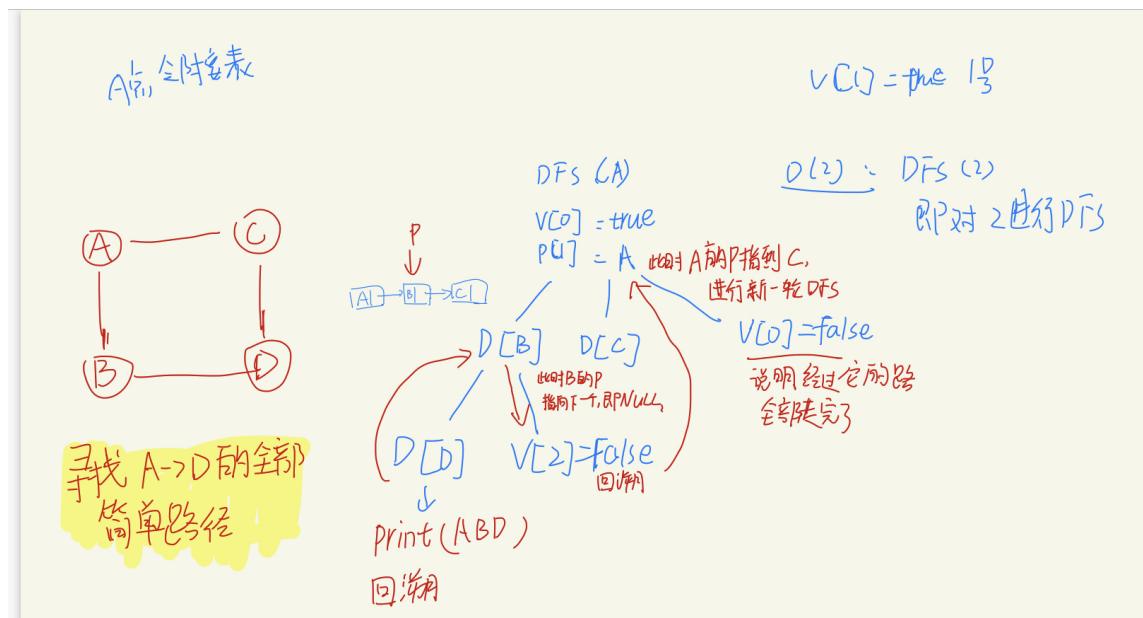
```

bool DFS_Path(ALGraph alGraph,int i,int j){
    bool isFind = false;
    for (int k = 0; k < alGraph.vexNum ; ++k) {
        visited[i] = false;
    }
    DFS(alGraph,i,j, & isFind);
    return isFind;
}

void DFS(ALGraph param, int v,int j,bool &isFind) {
    visited[v] = true;
    if (v == j){
        isFind = true;
    }
    for (int w = FirstNeighbor(param,v); w >= 0 ; w = NextNeighbor(param,v,w)) {
        if (!visited[w]){
            DFS(param,w,j, & isFind);
        }
    }
}

```

- 3.图用邻接表表示，设计一个算法，输出从顶点 $v_i$ 到顶点 $v_j$ 的所有简单路径
  - 思路：很明显我们需要使用DFS来做，可以看做是第二题的进一步操作，也就是当我们找到一条路径之后需要将其输出而不是直接return。其中，涉及到递归的回溯过程。
  - 回溯流程图 -- 能否看懂 看天赋了



- 代码：

```

void DFS_AllSimple_Path(ALGraph alGraph,int v,int j,int path[],int d){
    for (int i = 0; i < alGraph.vexNum; ++i) {
        visited[i] = false;
    }
    ArcNode *p;
    path[++d] = v;// 输入的d为 -1
    visited[v] = true;
    // 放入path数组之后 进行判断
    if (v == j){ // 如果已经找到了 打印path数组
        for (int i = 0; i <=d ; ++i) {
            cout<<path[i]<<" ";
        }
        cout<<endl;
    }

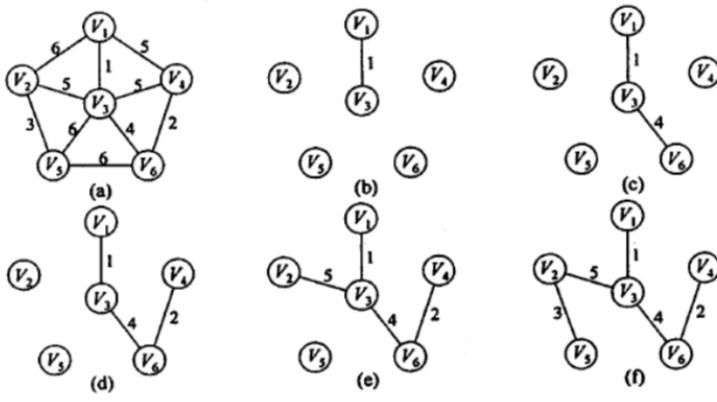
    p = alGraph.vertices[v].first; // p 为v的第一个邻接顶点 弧
    while (p != NULL){
        int w = p->adjvex; // p 指向的 邻接顶点下标
        if (!visited[w]){ // w 没有被访问
            DFS_AllSimple_Path(alGraph,w,j,path,d); // DFS递归
        }
        // 如果这个邻接顶点被访问过了 说明与这个邻接顶点相关的所有边都已经找过了
        // 就要换下一个邻接顶点
        p = p->next;
    }
    // 如果 while循环 退出 就说明 p == NULL 也就是顶点 V的所有邻接顶点都被找过了
    // 把它设置为最初的状态
    visited[v] = false;
}

```

## • 四、图的应用

- 1.最小生成树

- 1.定义：在连通网的所有生成树中，所有边的代价和最小的生成树，称为最小生成树。
- 2.特点：
  - 1、树形不唯一，边的权值唯一
  - 2、边数=顶点数-1
- 3.两种求最小生成数的算法
  - 1.Prim（普里姆）算法
    - 实例



- 1、思想（从顶点扩展）

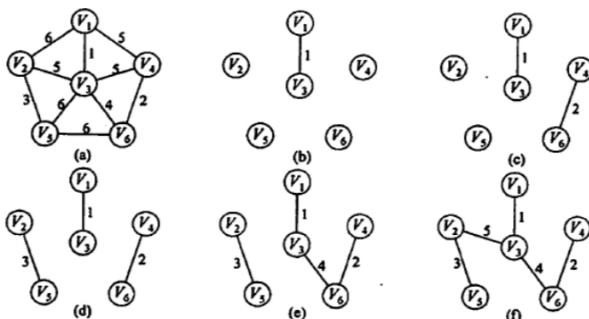
- 1、初始化：先任选一个顶点作为初始顶点
- 2、循环（直到包含所有顶点）；再选择这个顶点的邻边中权值最小的边并且不会构成回路
- 3、再选择这两个顶点的邻边中权值最小的边并且不会构成回路

- 2、特点：

- 1、时间复杂度： $O(|V|^2)$ ，不依赖 $|E|$ ，适合边稠密的图

- 2.Kruskal（克鲁斯卡尔）算法

- 



- 1、思想（权值递增）

- 1、初始化：先包含所有的顶点，没有边
- 2、循环（直到成为一棵树）：按权值递增的顺序选择边且不构成回路，直到包含 $n-1$ 条边

- 2、特点：

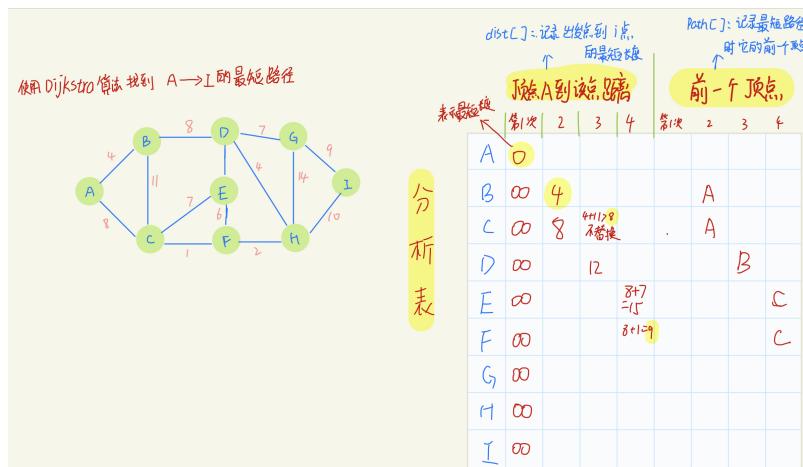
- 1、采用堆存放边集合，时间复杂度 $O(|E|\log|E|)$ ，适合边稀疏而顶点多的图

- 2.最短路径

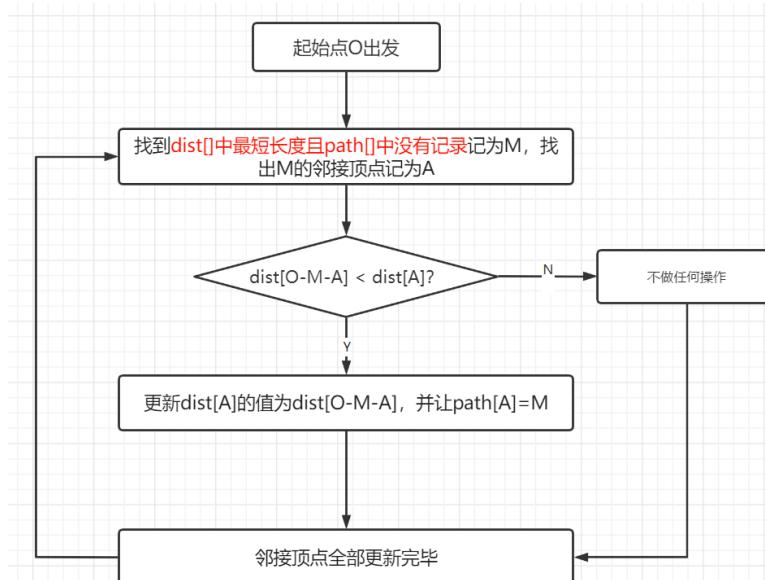
- 1.定义：从图中的某个顶点出发到达另外一个顶点的所经过的边的权重和最小的一条路径，称为最短路径。带权有向图的最短路径问题一般可分为两类：
  - 一是单源最短路径，即求图中某一顶点到其他各顶点的最短路径（Dijkstra算法）
  - 二是求每对顶点间的最短路径（Floyd算法）

- 2. Dijkstra算法

- 1、定义：Dijkstra算法使用了广度优先搜索解决赋权有向图或者无向图的单源最短路径问题，算法最终得到一个最短路径树。时间复杂度 $O(V^2)$
- 2、例题以及分析



### 3、算法流程图



### 4、代码实现

```

void ShortPath_Dijkstra(MGraph mGraph, int start){
    VertexType path[MaxSize]; // 存放上一顶点的数组
    EdgeType dict[MaxSize]; // 存放最短路径的数组
    EdgeType final[MaxSize]; // 记录顶点是否已经被采用 0: 未被采用 1: 被采用
    int min = 0; // 最小值
    int min_position; // 记录最小值的下标
    // 初始化
    for (int i = 0; i < mGraph.vexNum; ++i) {
        final[i] = 0;
        dict[i] = mGraph.Edge[start][i]; // 第一次对dict数组进行更新
        if (dict[i] != INFINITY) // 如果 i 是 start的邻接顶点
            path[i] = start;
        else // 不是
            path[i] = -1;
    }
}

```

```

// 开始进行循环 使用 n-1 轮循环即可将所有的最短路径找到。
for (int i = 1; i < mGraph.vexNum; ++i) {
    // 进行核心操作，找到 最短长度的点 && 没有被采用过的顶点
    min = INFINITY;
    for (int j = 0; j < mGraph.vexNum; ++j) { // 我们从头遍历到尾 一定能找到
        if (final[j]==0 && dict[j] < min){
            min_position = j;
            min = dict[j];
        }
    }
    // 找到了我们need的 最短长度点
    final[min_position] = 1; // 标记被采用过了
    // 检查是否可以更新dict[] 以及 path[]
    // 若 min + min_position -> i 的距离 < dict[i] 而且 final[i] = 0
    // 则可以进行修改
    for (int k = 0; k < mGraph.vexNum; ++k) {
        if (final[k] == 0 && (min + mGraph.Edge[min_position][k]) < dict[k]){
            dict[k] = min + mGraph.Edge[min_position][k]; // 修改当前路径长度
            path[k] = min_position; // 修改path数组
        }
    }
}
printDijkstraResult(mGraph,path,start);

```

- 5.样例以及实现效果

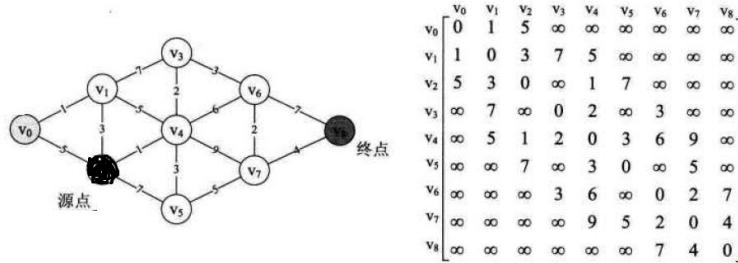


图 7-7-7

V2 → V1 → V0  
 V2 → V1  
 V2  
 V2 → V4 → V3  
 V2 → V4  
 V2 → V4 → V5  
 V2 → V4 → V3 → V6  
 V2 → V4 → V3 → V6 → V7  
 V2 → V4 → V3 → V6 → V7 → V8

- 3.Floyd算法

- 1、定义
  - Floyd（弗洛伊德）算法相对于Dijkstra算法来说，可以解决多源最短路径问题（即可以从任意一个点到任意一个点），可应用于地图导航走最短路径、为各城

市修建最短路径的通信网（节省成本）等问题，时间复杂度是 $O(V^3)$

- 2、代码实现

```
void ShortPath_Floyd(MGraph graph) {
    // 初始化
    int path[MaxSize][MaxSize];
    int A[MaxSize][MaxSize];           // 带权长度
    for (int i = 0; i < graph.vexNum; ++i) {
        for (int j = 0; j < graph.vexNum; ++j) {
            A[i][j] = graph.Edge[i][j];
            path[i][j] = -1; // 表示直达
        }
    }

    // 核心代码
    for (int v = 0; v < graph.vexNum; ++v) {
        for (int i = 0; i < graph.vexNum; ++i) {
            for (int j = 0; j < graph.vexNum; ++j) {
                if (A[i][j] > A[i][v] + A[v][j]) { // 加入中间点 距离更短
                    A[i][j] = A[i][v] + A[v][j]; // 更新距离
                    path[i][j] = v;           // 记录中间顶点
                }
            }
        }
    }

    for (int i = 0; i < graph.vexNum; ++i) {
        for (int j = i+1; j < graph.vexNum; ++j) {
            printFloyd(i, j, path);
            cout<<endl;
        }
        cout<<"\n";
    }
}

void printFloyd(int i, int j, int p[100][100]) {
    if (p[i][j] == -1)
        cout<<"V"<<i<<" -> "<<"V"<<j<<" ";
    else{
        int mid = p[i][j];
        printFloyd(i,mid,p);
        printFloyd(mid,j,p);
    }
}
```

- 4. 拓扑排序

- 1、定义：

- 拓扑排序是针对DAG(Directed acyclic graph，有向无环图),找到一个可以执行的顺序序列。DAG图表示一个工程，顶点表示活动，有向边 $<v_i, v_j>$ 表示活动*i*必须先

于活动j

- 2、特点：
  - 如果这个图不是 DAG，那么它是没有拓扑序的；
  - 如果是 DAG，那么它至少有一个拓扑序；
  - 反之，如果它存在一个拓扑序，那么这个图必定是 DAG.
- 3、思路
  - 对AOV网进行拓扑排序的基本思路是：从AOV网中选择一个入度为0的顶点输出，然后删去此顶点，并删除以此顶点为尾的弧，继续重复此步骤，直到输出全部顶点或者AOV网中不存在入度为0的顶点为止。