图

7.1 图的基本概念

1.图：由结点和有穷集合V和边的结合E组成。将结点称为顶点，边是顶点的有序偶对。若两个顶点之间存在一条边，则表示这两个顶点具有相邻关系。
2.

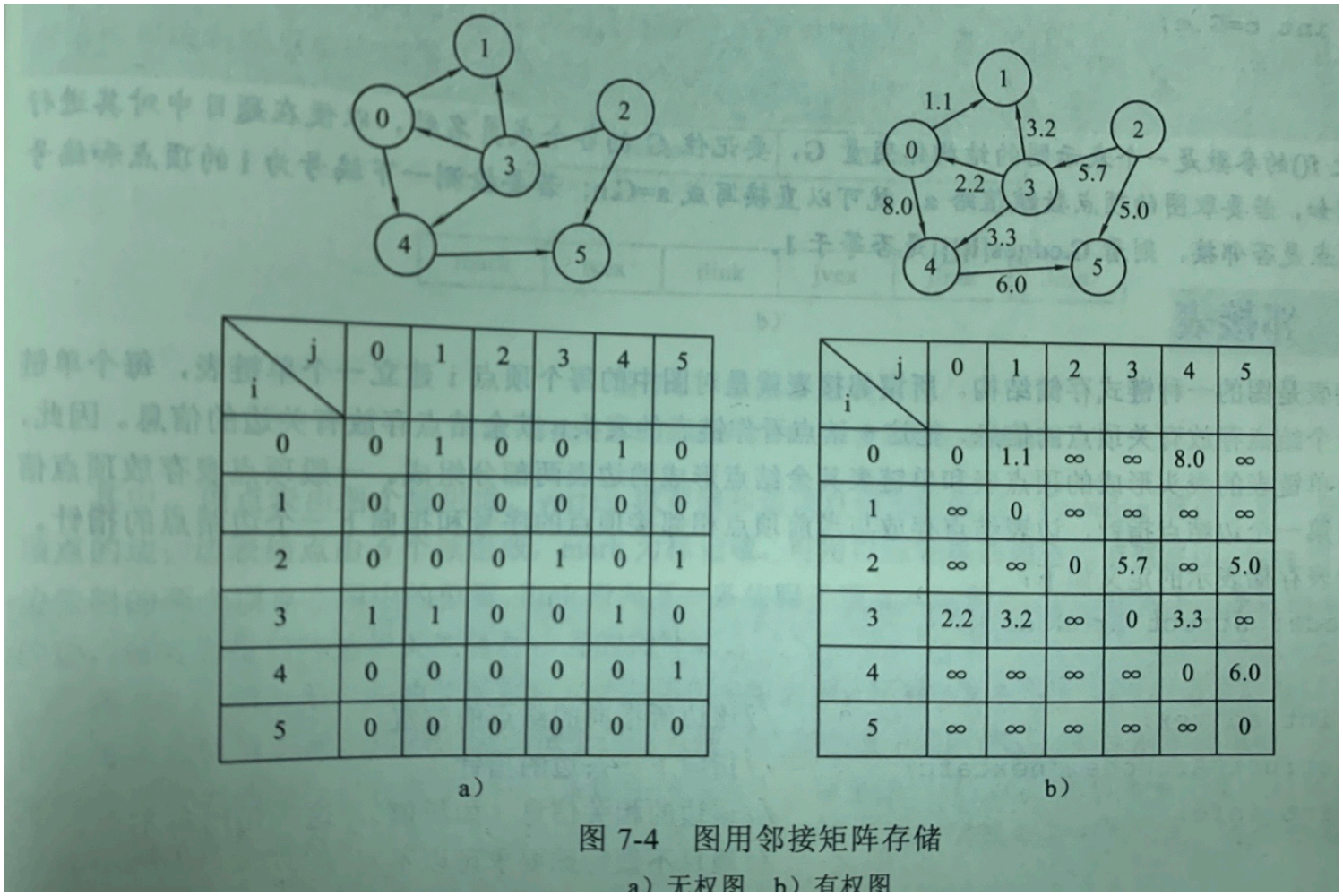
有向图	无向图
每条边都没有方向	每条边都有方向

3.弧：在有向图中，通常将边称为弧，含箭头的成为弧头，另一端称为弧尾，记作<vi,vj>，它表示从顶点vi到顶点vj有一条边。
4.顶点的度、入度和出度
5.有向完全图和无向完全图：若有向图中有n个顶点，则最多有n(n−1)条边，这样的有向图称为有向完全图。若无向图中有n个顶点，则最多有n(n−1)/2条边，这样的无向图称为无向完全图。
6.路径和路径长度：路径为相邻顶点序偶所构成的序列。路径长度是指路径上边的数目。
7.简单路径：序列中顶点不重复出现的路径。
8.回路：若一条路径中第一个顶点和最后一个顶点相同，则这条路径是一条回路。
9.连通、连通图和连通分量：若vi到vj有路径，则称vi和vj连通。如果图中任意两个顶点之间都连通，则称该图为**连通图**；否则图中的**极大连通子图称为连通分量。**
10.强连通图和强连通分量：如果对于每一对顶点vi和vj，从vi到vj和从vj到vi都有路径，称为强连通图；否则将其中的极大强连通子图称为强连通分量。
11.权和网：途中每条边都有一个对应的数（权），可以表示从一个顶点到另一个顶点的距离或花费的代价。边上带有权的图称为带权图，也称为网。

7.2 图的存储结构

7.2.1 邻接矩阵

A[i][j]=1，代表顶点i和j邻接，即i和j存在边或弧；
A[i][j]=0，代表顶点i和j不邻接。（0≤i，j≤n−1）。



i \ j	0	1	2	3	4	5
0	0	1	0	0	1	0
1	0	0	0	0	0	0
2	0	0	0	1	0	1
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	0	0	0	0	0

a)

i \ j	0	1	2	3	4	5
0	0	1.1	∞	∞	8.0	∞
1	∞	0	∞	∞	∞	∞
2	∞	∞	0	5.7	∞	5.0
3	2.2	3.2	∞	0	3.3	∞
4	∞	∞	∞	∞	0	6.0
5	∞	∞	∞	∞	∞	0

b)

图 7-4　图用邻接矩阵存储

a）无权图　b）有权图

邻接矩阵的结构型定义如下：

```
typedef struct {
    int no;
    char info;
} VertexType;

typedef struct {
    int edges[maxSize][maxSize];
    int n, e;
    VertexType vex[maxSize];
} MGraph;
```

7.2.2 邻接表

邻接表是图的一种链式存储结构。

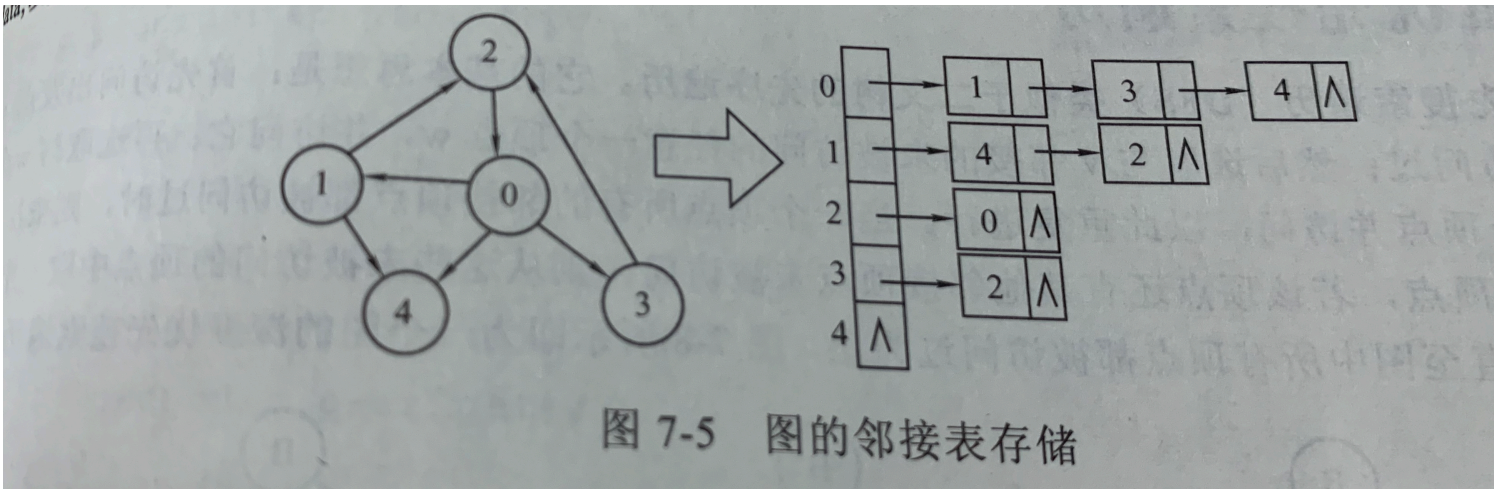
对途中的每个顶点i建立一个单链表，每个单链表的第一个结点存放有关顶点的信息，把这一结点看作链表的表头，其余结点存放有关边的信息。邻接表由单链表的表头形成的顶点表和单链表其余结点形成的边表两部分组成。顶点表存放顶点信息和指向第一个边结点指针，边表结点存放于当前顶点相邻接顶点的序号和指向下一个边结点的指针。

邻接表存储表示的定义如下：

```
typedef struct ArcNode {
    int adjvex;                 //该边所指向的结点的位置
    struct ArcNode *nextarc;    //指向下一条边的指针
    int info;                   //该边的相关信息（如权值）
} ArcNode;

typedef struct {
    char data;                  //顶点信息
    ArcNode *firstarc;          //指向第一条边的指针
} VNode;

typedef struct {
    VNode adjlist[maxSize];     //邻接表
    int n, e;                   //顶点数和边数
} AGraph;                       //图的邻接表类型
```



7.2.3 邻接多重表

邻接多重表和十字链表相似，有顶点表和边表组成，每一条边用一个结点表示。

顶点表结构：

Vertex	firstedge

vertex：和该顶点相关的信息

firstedge：第一条以富裕该顶点的边

边表结点结构：

mark	ivex	ilink	jvex	jlink	info

mark：标记域，记录是否被搜索过

ivex，jvex：该边衣服的两个顶点在图中的位置

ilink：指向下一条依附于顶点ivex的边

jlink：指向下一条依附于顶点jvex的边

info：指向与边相关的各种信息的指针域。

7.3 图的遍历算法操作

7.3.1 深度优先搜索遍历（DFS）

类似于二叉树的先序遍历

基本思想

1.访问出发结点v，并将其标记为已访问；

2.选取与v邻接的未被访问的任意一个顶点我，并访问；

3.在选取与w邻接的未被访问的任一顶点并访问；

4.当一个顶点的所有邻接顶点都被访问过时，则一次退回到最近被访问的顶点；若该顶点还有其他邻接顶点未被访问，则从这些未被访问的顶点中去一个饼重复上述过程。

以邻接表为存储结构的图的深度优先搜索遍历算法如下：

```
int visit[maxSize];

void DFS(AGraph *G, int v) {
    ArcNode *p;
    visit[v] = 1;                       //设置已访问标记
    Visit(v);
    p = G->adjlist[v].firstarc;         //指向顶点v的第一条边
    while(p != NULL) {
        if(visit[p->adjvex] == 0)       //若顶点未访问，则递归访问它
            DFS(G, p->adjvex);
            p = p->nextarc;
    }
}
```

7.3.2 广度优先搜索遍历（BFS）

类似于树的层次访问

执行过程：

1）任取图中一个顶点访问，入队，并将这个顶点标记为已访问；

2）队列不空时循环：出队，依次检查出队顶点的所有邻接顶点，访问没有被访问过的邻接顶点并将其入队；

3）当队列为空时，跳出循环。

算法：

```
void BFS(AGraph *G, int v, int visit[maxSize]) {
    ArcNode *p;
    int que[maxSize], front = 0, rear = 0;
    int j;
    Visit(v);
    rear = (rear+1) % maxSize;              //当前顶点v进队
    que[rear] = v;
    while(front != rear) {
        front = (front + 1) % maxSize;      //顶点出队
        j = que[front];
        p = G->adjlist[j].firstarc;         //p指向出队顶点j的
                                            //第一条边
        while(p != NULL) {
            if(visit[p->adjvex] == 0) {
                Visit(p->adjvex);
                visit[p->adjvex] = 1;
                rear = (rear+1) % maxSize;  //该顶点进队
                que[rear] = p->adjvex;
            }
            p = p->nextarc;                 //p指向j的下一条边
        }
    }
}
```

以上两种方法只适用于连通图

对于非连通图：

(1)深度优先遍历：

```
void dfs(AGraph *p) {
    int i;
    for(i = 0; i < g->n; ++i) {
        if(visit[i] == 0)
            DFS(g, i);
    }
}
```

(2)广度优先遍历：

```
void bfs(AGraph *p) {
    int i;
    for(i = 0; i < g->n; ++i) {
```

```
        if(visit[i] == 0)
            BFS(g, i, visit);
    }
}
```

7.4 最小（代价）生成树

7.4.1 普里姆算法和克鲁斯卡尔算法

1.普里姆算法

（1）主要思想

从图中任取一个顶点，把它当作一棵树，然后从**与这棵树相接**的边中选取一条最短（**权值最小**）的边，并将边与所连接的顶点并入，然后从与这棵树想接的边中选取一条最短的边，再将边与所连顶点并入，以此类推。

在此过程中，建立两个数组vest[]和lowcost[]。vest[i]=1表示顶点i已经并入生成树中，vest[i]=0反之；lowcost[]存放当前**生成树**到剩余各顶点**最短边的权值**。

（2）执行过程

从树中某一顶点v0开始：

1）将v0倒其他顶点的所有边当作候选边；

2）重复以下步骤n−1次：

- 从候选边中挑选出权值最小的边输出，并将与该边另一端相接的顶点v并入生成树。
- 考察所有剩余顶点vi，如果(v,vi)权值比lowcost[vi]小，则用(v,vi)的权值更新lowcost[vi]。

代码如下：

```
void Prim(MGraph g, int v0, int &sum) {
    int lowcost[maxSize], vest[maxSize], v;
    int i, j, k, min;
    v = v0;
    for(i = 0; i < g.n; ++i) {
        lowcost[i] = g.edges[v0][i];
        vest[i] = 0;
    }
    vest[v0] = 1;
    sum = 0;
    for(i = 0; i < g.n; ++i) {
        min = INF;          //INF是一个已经定义的比图中
                            //所有边权值都大的常量
        for(j = 0; j < g.n; ++j) {
            if(vest[j] == 0 && lowcost[j] < min) {
                min = lowcost[j];       //选出最短的边
                k = j;
```

```
            }
        }
        vest[k] = 1;
        v = k;
        sum += min;                 //记录最小生成树的权值
        for(j = 0; j < g.n; ++j) {
            if(vest[j] == 0 && g.edges[v][j] < lowcost[j]) {
                lowcost[j] = g.edges[v][j];
            }
        }
    }
}
```

（3）时间复杂度：$O(n^2)$

2.克鲁斯卡尔算法

（1）基本思想

每次找出候选边中**权值最小**的边，就将该边并入生成树中，重复此过程。

（2）执行过程

将途中的边按权值从小到大排列，从最小边开始扫描，并检测当前边是否为候选边，即是否并入，如不构成回路，则将改变并入当前生成树中，知道所有边都被检测完。

代码如下：

```
typedef struct {
    int a, b;               //一条边相连的两个顶点
    int w;                  //权值
} Road;

Road road[maxSize];         //定义并查集数组
int v[maxSize];             //在并查集中查找根结点的函数

int getRoot(int a) {
    while(a != v[a]) a = v[a];
    return a;
}

void Kruskal(Magraph g, int &sum, Road road[]) {
    int i;
    int N, E, a, b;
    N = g.n;
    E = g.e;
    sum = 0;
    for(i = 0; i < N; ++i)
        v[i] = i;
```

```
    sort(raod, E);                  //对road数组中的E条件边按其
                                    //权值从小到大排列
    for(i = 0; i < E; ++i) {
        a = getRoot(road[i].a);
        b = getRoot(road[i].b);
        if(a != b) {
            v[a] = b;
            sum += road[i].w;
        }
    }
}
```

（3）时间复杂度

时间主要消耗在sort()和单层循环上。时间复杂度主要由选取的排序算法决定。排序算法所处理数据的规模由变数e决定，与顶点无关，**该算法适用于稀疏图。**

注：普里姆算法和克鲁斯卡尔算法都针对于无向图。

7.5 最短路径

7.5.1 迪杰斯特拉

通常采用该算法求图中**某一顶点到其余个顶点的最短路径**。

1.主要思想：

设有两个顶点集合S和T，集合S中存放图中已找到最短路径的顶点，集合T存放图中剩余顶点。初始状态时，集合S中只包含源点v0，然后不断从集合T中选取到顶点v0路径长度最短的顶点vu并入S集合中。**集合S每并入一个新结点vu，都要修改顶点v0到集合T中顶点的最短路径长度值**。不断从父此过程，知道T中顶点全部并入到S为止。

在vu被选入S中，vu被确定为最短路径上的顶点，此时vu就想v0到达T中顶点的中转站，多一个中转站，到达T中的路径就增加了，而这些**新的路径可能比之前的路径要短**，因此要修改v0到T中的路径长度。此时对于T中的一个顶点vk，有两种情况：

1.v0不经过vu到达vk的路径长度为a（旧路径长度）

2.v0经过vu到达vk的路径长度为b（新路径长途）

若a≤b，什么都不做；若a>b，则用b代替a。

当T中所有顶点都被处理完后，会出现一组新的v0到T中各顶点的路径，其中有一条最短的，对应了T中一个顶点，就是新的vu，并入S。重复上述过程

2.执行过程：

见书P199

打印路径函数：

```
void printPath(int path[], int a) {
```

```
    int stack[maxSize], top = -1;
    while(path[a] != -1) {
        stack[++top] = a;
        a = path[a];
    }
    stack[++top] = a;
    while(top != -1)
        cout<<stack[top--]<<" ";
    cout<<endl;
}
```

迪杰斯特拉算法代码：

```
void Dijkstra(MGraph g, int v, int dist[], int path[]) {
    int set[maxSize];
    int min, i, j, u;
    for(i = 0; i < g.n; ++i) {
        dist[i] = g.edges[v][i];
        set[i] = 0;
        if(g.edges[v][i] < INF)
            path[i] = v;
        else
            path[i] = -1;
    }
    set[v] = 1;
    path[v] = -1;
    for(i = 0; i < g.n-1; ++i) {
        min = INF;
        for(j = 0; j < g.n; ++j) {
            if(set[j] == 0 && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }
    }
    set[u] = 1;
    for(j = 0; j < g.n; ++j) {
        if(set[j] == 0 && dist[u] + g.edges[u][j] < dist[j]) {
            dist[j] = dist[u] + g.edges[u][j];
            path[j] = u;
        }
    }
}
```

3.时间复杂度：$O(n^2)$

7.5.2 弗洛伊德算法

书P205

算法代码：

```
void Floyd(MGraph g, int Path[][maxSize]) {
    int i, j, k;
    int A[maxSize][maxSize];
    for(i = 0; i < g.n; ++i) {
        for(j = 0; j < g.n; ++j) {
            A[i][j] = g.edges[i][j];
            Path[i][j] = -1;
        }
    }
    for(k = 0; k < g.n; ++k) {
        for(i = 0; i < g.n; ++i) {
            for(j = 0; j < g.n; ++j) {
                if(A[i][j] > A[i][k] + A[K][J]) {
                    A[i][j] = A[i][k] + A[k][j];
                    Path[i][j] = k;
                }
            }
        }
    }
}
```

时间复杂度:$O(n^3)$

7.6 拓扑排序

7.6.1 拓扑排序核心算法

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，若存在由u到v的路径，则在拓扑排序序列中一定是u出现在v的前边。

在一个有向图中找到一个拓扑排序序列的过程：

1）从有向图中找到一个没有前驱（入度为0）的顶点输出；

2）删除1）中的顶点，并删除从该顶点发出的全部边；

3）重复步骤，到剩余的图中不存在没有前驱的顶点为止。

结构体：

```
typedef struct {
    char data;
    int count;
    ArcNode *firstarc;
} VNode;
```

算法：

算法开始时置n为0，扫描所有顶点，将入度为0的顶点入栈。然后再栈不空的时候循环执行：出栈，将出栈顶点输出，执行++n，并且将由此顶点引出的边所持想的顶点入度都减1，并且将入度变为0的顶点入栈；出栈，...，栈空时循环推出，排序结束。循环推出后判断n是否等于图中顶点个数。相等返回1，拓扑成功；反之返回0，失败。

代码：

```
int TopSort(AGraph *G) {
    int i, j, n = 0;
    int stack[maxSize], top = -1;
    ArcNode *p;
    for(i = 0; i< G->n; ++i) {
        if(G->adjlist[i].count == 0) {
            stack[++top] = i;
        }
    }
    while(top != -1) {
        i = stack[top--];
        ++n;
        cout<<i<<" ";
        p = G->adjlist[i].firstarc;
        while(p != NULL) {
            j = p->adjves;
            --(G->adjlist[j].count);
            if(G->adjlist[j].count == 0) {
                stack[++top] = j;
            }
            p = p->nextarc;
        }
    }
    if(n == G->n)
        return 1;
    else
        return 0;
}
```

若AOV图中考察各顶点的出度并以下列步骤进行排序，则将这种排序成为拟拓扑排序，输出结果为拟拓扑有序序列。

1）在网中选择一个没有后继的顶点（出度为0）输出；

2）在网中删除该结点，并删除所有到达该结点的边；
3）重复上两部，知道AOV网中已无出度为0的顶点为止。

7.7 关键路径

7.7.1 AOE网

与AOV图对比

相同点	都是有向无环图
不同点	AOE网的边表示活动，边有权值，代表活动持续时间。顶点表示时间，事件是图中新活动开始或酒活动结束的标志。 AOV网的顶点表示活动，边无权值，边代表活动之间的先后关系。

7.7.2 关键路径核心算法

见书P210