第七章 图

刘杰
人工智能学院
南开大学 Nankai University

目录

图的基本概念

图(Graph)是一种比线性表和树更为复杂的数据结构。

线性结构：是研究数据元素之间的一对一关系。在这种结构中，除第一个和最后一个元素外，任何一个元素都有唯一的一个直接前驱和直接后继。

树结构：是研究数据元素之间的一对多的关系。在这种结构中，每个元素对下(层)可以有0个或多个元素相联系，对上(层)只有唯一的一个元素相关，数据元素之间有明显的层次关系。

图的基本概念

图结构：是研究数据元素之间的多对多的关系。在这种结构中，任意两个元素之间可能存在关系。即结点之间的关系可以是任意的，图中任意元素之间都可能相关。

图的应用极为广泛，已渗入到诸如语言学、逻辑学、物理、化学、电讯、计算机科学以及数学的其它分支。

图的定义

图由结点的有穷集合V和边的集合E组成。

其中，为和树形结构区别，通常将结点称为顶点，边是顶点的有序偶对，若两个顶点之间存在一条边，就表示这两个顶点具有相邻关系。其形式化定义为：

$$G = (V, E)$$

$$V = \{v_i | v_i \in data\ object\}$$

$$E = \{< v_i, v_j > | v_i, v_j \in V \wedge p(v_i, v_j)\}$$

其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合，$P(v_i, v_j)$表示从顶点v_i到顶点v_j有一条边。

有向图与无向图

弧(Arc)：表示两个顶点v和w之间存在一个关系，用顶点偶对$< v, w >$表示。通常根据图的顶点偶对将图分为有向图和无向图。

有向图(Digraph)：若图G的关系集合E(G)中，顶点偶对$< v, w >$的v和w之间是有序的，称图G是有向图。

在有向图中，若$< v, w > \in E(G)$，表示从顶点v到顶点w有一条弧。其中：v称为弧尾(tail)或始点(initial node)，w称为弧头(head)或终点(terminal node)。

无向图(Undigraph)：若图G的关系集合$E(G)$中，顶点偶对$< v, w >$的v和w之间是无序的，称图G是无向图。

有向图与无向图

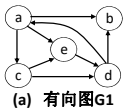
在无向图中，若$< v,w > \in E(G)$，有$< w,v > \in E(G)$，即$E(G)$是对称，则用无序对(v,w)表示v和w之间的一条边(Edge)，因此(v,w)和(w,v)代表的是同一条边。

例1：设有有向图G1和无向图G2，形式化定义分别是：

$G1 = (V1,\ E1)\ V1 = \{a,b,c,d,e\}$

$E1 = \{< a,b >,< a,c >,< a,e >,< c,d >,< c,e >,< d,a >,< d,b >,< e,d >\}$

$G2 = (V2,\ E2)\ V2 = \{a,b,c,d\}\ E2 = \{(a,b),(a,c),(a,d),(b,d),(b,c),(c,d)\}$



(a) 有向图G1

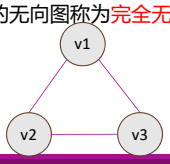
(b) 无向图G2

完全无向图

完全无向图：对于无向图，若图中顶点数为n，用e表示边的数目，则$e \in [0,\ n(n-1)/2]$。具有$n(n-1)/2$条边的无向图称为完全无向图。

完全无向图另外的定义是：

对于无向图$G = (V,\ E)$，若$v_i,\ v_j \in V$，当$v_i \neq v_j$时，有$(v_i, v_j) \in E$，即图中任意两个不同的顶点间都有一条无向边，这样的无向图称为完全无向图。

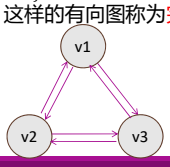


完全有向图

完全有向图：对于有向图，若图中顶点数为n，用e表示弧的数目，则$e \in [0,\ n(n-1)]$。具有$n(n-1)$条边的有向图称为完全有向图。

完全有向图另外的定义是：

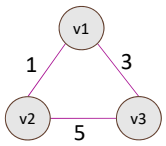
对于有向图$G = (V,\ E)$，若任意$v_i, v_j \in V$，当$v_i \neq v_j$时，有$< v_i, v_j > \in E$且$< v_j, v_i > \in E$，即图中任意两个不同的顶点间都有一条弧，这样的有向图称为完全有向图。



图的稀疏性与权重

有很少边的图$(e < nlogn)$的图称为稀疏图，反之称为稠密图。

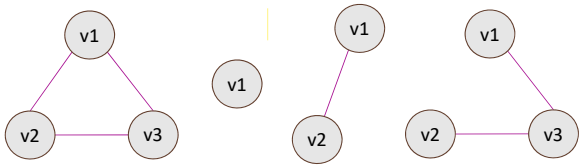
权(Weight)：与图的边和弧相关的数。权可以表示从一个顶点到另一个顶点的距离或耗费。



带权无向图

子图和生成子图

子图和生成子图：设有图$G = (V,E)$和$G' = (V',E')$，若$V' \subset V$且$E' \subset E$，则称图G'是G的子图；若$V' = V$且$E' \subset E$，则称图G'是G的一个生成子图。



(a) (b) (c)

(a)(b)为子图，(c)为生成子图

顶点的邻接与度

顶点的邻接(Adjacent)：对于无向图$G = (V,E)$，若边$(v,w) \in E$，则称顶点v和w互为邻接点，即v和w相邻接。边(v,w)依附(incident)于顶点v和w。

对于有向图$G = (V,E)$，若有向弧$< v,w > \in E$，则称顶点v邻接到顶点w，顶点w邻接自顶点v，弧$< v,w >$与顶点v和w相关联。

顶点的度、入度、出度：对于无向图$G = (V,E)$，$v_i \in V$，图G中依附于v_i的边的数目称为顶点v_i的度(degree)，记为$TD(vi)$。

易知，无向图中，所有顶点度的和是图中边的2倍。

顶点的邻接与度

对有向图$G=(V,E)$，若$v_i \in V$，图G中以v_i作为起点的有向边(弧)的数目称为顶点v_i的出度(Outdegree)，记为$OD(v_i)$；以v_i作为终点的有向边(弧)的数目称为顶点v_i的入度(Indegree)，记为$ID(v_i)$。顶点v_i的出度与入度之和称为v_i的度，记为$TD(v_i)$。即

$$TD(v_i)=OD(v_i)+ID(v_i)$$

路径(Path)、路径长度、回路(Cycle)：对无向图$G=(V,E)$，若从顶点v_i经过若干条边能到达v_j，称顶点v_i和v_j是连通的，又称顶点v_i到v_j有路径。

对有向图$G=(V,E)$，从顶点v_i到v_j有有向路径，指的是从顶点v_i经过若干条有向边(弧)能到达v_j。

路径与回路

路径也可以定义为图G中连接两顶点之间所经过的顶点序列。即

$Path=v_{i_0}v_{i1}\ldots v_{im}$，$v_{ij}\in V$且$(v_{ij-1},v_{ij})\in E \quad j=1,2,\ldots,m$对有向图：

$Path=v_{i_0}v_{i1}\ldots v_{im}$，$v_{ij}\in V$且$<v_{ij-1},v_{ij}>\in E\ j=1,2,\ldots,m$

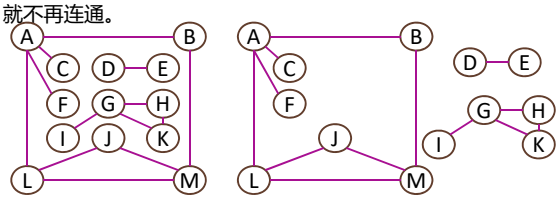
路径上边或有向边(弧)的数目称为该路径的长度。

在一条路径中，若没有重复相同的顶点，该路径称为简单路径；第一个顶点和最后一个顶点相同的路径称为回路(环)；在一个回路中，若除第一个与最后一个顶点外，其余顶点不重复出现的回路称为简单回路(简单环)。

连通图与连通分量

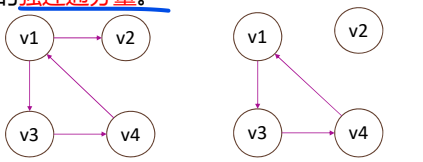
连通图、图的连通分量：对无向图$G=(V,E)$，若任意v_i，$v_j\in V$，v_i和v_j都是连通的，则称图G是连通图，否则称为非连通图。若G是非连通图，则极大的连通子图称为G的连通分量。

“极大”的含义：指的是对子图再增加图G中的其它顶点，子图就不再连通。



连通图与连通分量

对有向图$G=(V,E)$，若$\forall v_i$，$v_j\in V$，都有以v_i为起点，v_j为终点以及以v_j为起点，v_i为终点的有向路径，称图G是强连通图，否则称为非强连通图。若G是非强连通图，则极大的强连通子图称为G的强连通分量。



(a)非强连通图　(b)两个强连通分量

生成树与生成森林

生成树、生成森林：一个连通图(无向图)的生成树是一个极小连通子图，它含有图中全部n个顶点和只有足以构成一棵树的n-1条边，称为图的生成树。

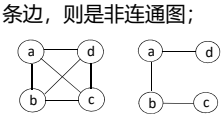
关于无向图的生成树的几个结论：

(1)一棵有n个顶点的生成树有且仅有n-1条边；

(2)如果一个图有n个顶点和小于n-1条边，则是非连通图；

(3)如果多于n-1条边，则一定有环；

(4)有n-1条边的图不一定是生成树。



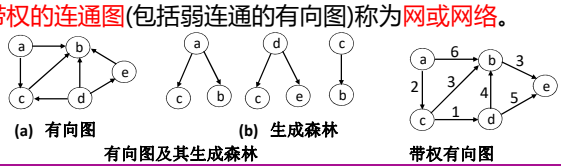
图的一棵生成树

生成树与生成森林

有向图的生成森林是这样一个子图，由若干棵有向树组成，含有图中全部顶点。

有向树是只有一个顶点的入度为0，其余顶点的入度均为1的有向图。

网：每个边(或弧)都附加一个权值的图，称为带权图。带权的连通图(包括弱连通的有向图)称为网或网络。



(a) 有向图　(b) 生成森林　带权有向图

有向图及其生成森林

图的存储结构

图的存储结构比较复杂，其复杂性主要表现在：（1）任意顶点之间可能存在联系，无法以数据元素在存储区中的物理位置来表示元素之间的关系。

（2）图中顶点的度不一样，有的可能相差很大，若按度数最大的顶点设计结构，则会浪费很多存储单元，反之按每个顶点自己的度设计不同的结构，又会影响操作。

图的常用的存储结构有：邻接矩阵、邻接链表、十字链表、邻接多重表和边表。

邻接矩阵(数组)表示法

基本思想：

对于有n个顶点的图，

(1)用一维数组$vexs[n]$存储顶点信息，

(2)用二维数组$A[n][n]$存储顶点之间关系的信息。

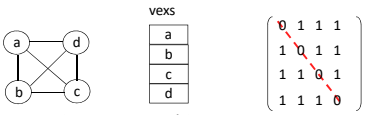
该二维数组称为邻接矩阵。在邻接矩阵中，以顶点在$vexs$数组中的下标代表顶点，邻接矩阵中的元素$A[i][j]$存放的是顶点i到顶点j之间关系的信息。

无权图的邻接矩阵

无向无权图$G=(V,\ E)$有$n(n\geq 1)$个顶点，其邻接矩阵是n阶对称方阵，如图所示。

其元素的定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}(vi,v_j)\in E\text{，即}v_i,v_j\text{邻接}\\ 0 & \text{若}(vi,v_j)\notin E\text{，即}v_i,v_j\text{不邻接}\end{cases}$$



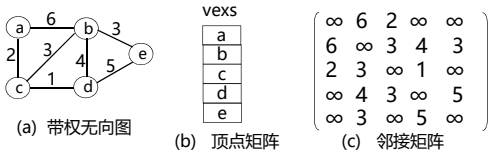
(a) 无向图 (b) 顶点矩阵 (c) 邻接矩阵

无向无权图的数组存储

无向带权图的邻接矩阵

无向带权图$G=(V,\ E)$的邻接矩阵如图所示。其元素的定义如下：

$$A[i][j]=\begin{cases}W_{ij} & \text{若}(vi,v_j)\in E\text{，即}v_i,v_j\text{邻接，权值为}w_{ij}\\ \infty & \text{若}(vi,v_j)\notin E\text{，即}v_i,v_j\text{不邻接时}\end{cases}$$



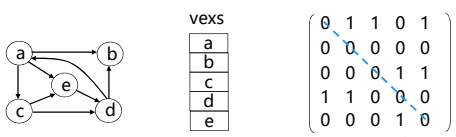
(a) 带权无向图 (b) 顶点矩阵 (c) 邻接矩阵

无向带权图的数组存储

有向无权图的邻接矩阵

有向无权图$G=(V,\ E)$邻接矩阵如图所示。元素定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}<vi,v_j>\in E\text{，从}v_i\text{到}v_j\text{有弧}\\ 0 & \text{若}<vi,v_j>\notin E\text{，从}v_i\text{到}v_j\text{没有弧}\end{cases}$$



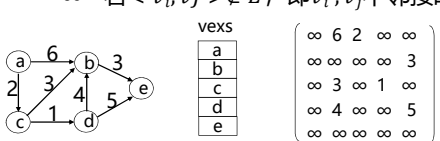
(a) 有向图 (b) 顶点矩阵 (c) 邻接矩阵

有向无权图的数组存储

有向带权图的邻接矩阵

有向带权图$G=(V,\ E)$的邻接矩阵如图所示。其元素的定义如下：

$$A[i][j]=\begin{cases}w_{ij} & \text{若}<v_i,vj>\in E\text{，即}v_i,v_j\text{邻接,权值为}w_{ij}\\ \infty & \text{若}<v_i,vj>\notin E\text{，即}v_i,v_j\text{不邻接时}\end{cases}$$



(a) 带权有向图 (b) 顶点矩阵 (c) 邻接矩阵

带权有向图的数组存储

邻接矩阵的特性

对无向图邻接矩阵：

(1)邻接矩阵是对称方阵；

(2)对于顶点v_i，其度数是第i行的非0元素的个数；

(3)无向图的边数是上(或下)三角形矩阵中非0元素个数。

对有向图，

(1)对于顶点v_i，第i行的非0元素的个数是其出度$OD(v_i)$；第i列的非0元素的个数是其入度$ID(v_i)$。

(2)邻接矩阵中非0元素的个数就是图的弧的数目。

邻接矩阵存储的实现

```
typedef int VRType; //顶点关系类型定义
typedef char InfoType; //顶点信息类型定义
typedef char VertexType[MAX_NAME];
typedef enum { DG, DN, AG, AN }GraphKind; /* {有向图,有向网,无向图,无向网} */

typedef struct {
    VRType adj; /*顶点关系类型。对无权图，用1(是)或0(否)表示相邻否 */
                /* 对带权图，则为权值类型 */
    InfoType *info; /* 该弧相关信息的指针(可无) */
}ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERTEX_NUM]; /* 顶点数组 */
    AdjMatrix arcs; /* 邻接矩阵 */
    int vexnum, arcnum; /* 图的当前顶点数和弧数 */
    GraphKind kind; /* 图的种类标志 */
}MGraph;
```


示例—有向图的构建

```
void CreateDG(MGraph &G)
{ /* 采用数组(邻接矩阵)表示法,构造有向图G */
    int i, j, k, l, IncInfo;
    char s[MAX_NAME], *info;
    VertexType va, vb;
    printf("请输入有向图G的顶点数,弧数,弧是否含其它信息(是:1,否:0): ");
    scanf("%d,%d,%d", G.vexnum, G.arcnum, &IncInfo);
    printf("请输入%d个顶点的值(<%d个字符):\n", G.vexnum, MAX_NAME);
    for (i = 0; i<G.vexnum; ++i) /* 构造顶点向量 */
        scanf("%s", G.vexs[i]);
    for (i = 0; i<G.vexnum; ++i) /* 初始化邻接矩阵 */
        for (j = 0; j<G.vexnum; ++j)
        {
            G.arcs[i][j].adj = 0; /* 图 */
            G.arcs[i][j].info = NULL;
        }
    printf("请输入%d条弧的弧尾 弧头(以空格作为间隔): \n", G.arcnum);
```


示例—有向图的构建

```
    for (k = 0; k<G.arcnum; ++k)
    {
        scanf("%s%s%*c", va, vb);  /* %*c忽略回车符 */
        i = LocateVex(G, va);
        j = LocateVex(G, vb);
        G.arcs[i][j].adj = 1; /* 有向图 */
        if (IncInfo) {
            printf("请输入该弧的相关信息(<%d个字符): ", MAX_NAME);
            gets_s(s);
            l = strlen(s);
            if (l) {
                info = (char*)malloc((l + 1) * sizeof(char));
                strcpy(info, s);
                G.arcs[i][j].info = info; /* 有向 */
            }
        }
    }
    G.kind = DG;
}
```


示例—图的顶点定位

图的顶点定位操作实际上是确定一个顶点在vexs数组中的位置(下标)，其过程完全等同于在顺序存储的线性表中查找一个数据元素。

```
int LocateVex(MGraph &G, VertexType u)
{ /* 初始条件:图G存在,u和G中顶点有相同特征 */
  /* 操作结果:若G中存在顶点u,则返回该顶点在图中位置;否则返回-1 */
    int i;
    for (i = 0; i<G.vexnum; ++i)
        if (strcmp(u, G.vexs[i]) == 0)
            return i;
    return -1;
}
```


示例—增加顶点

```
void InsertVex(MGraph &G, VertexType v)
{  /* 在图G中增添新顶点v(不增添与顶点相关的弧)，类似在顺序存储的线性表的末尾增加一个数据元素。*/
    int i;
    strcpy(G.vexs[G.vexnum], v); /* 构造新顶点向量 */
    for (i = 0; i <= G.vexnum; i++) {
        if (G.kind % 2) /* 网 */{
            G.arcs[G.vexnum][i].adj = INFINITY; //初始化该行邻接矩阵的值
            G.arcs[i][G.vexnum].adj = INFINITY; //初始化该列邻接矩阵的值
        }
        else /* 图 */{
            G.arcs[G.vexnum][i].adj = 0; /* 初始化该行邻接矩阵的值*/
            G.arcs[i][G.vexnum].adj = 0; /* 初始化该列邻接矩阵的值*/
        }
        G.arcs[G.vexnum][i].info = NULL; /* 初始化相关信息指针 */
        G.arcs[i][G.vexnum].info = NULL;
    }
    G.vexnum += 1; /* 图G的顶点数加1 */
}
```

示例—增加弧

根据给定的弧或边所依附的顶点，修改邻接矩阵中所对应的数组元素。

```
void InsertArc(MGraph &G, VertexType v, VertexType w)
{ /* 初始条件：图G存在,v和W是G中两个顶点 */
  /* 操作结果：在G中增添弧<v,w>,若G是无向的,则还增添对称弧<w,v> */
    int i, l, v1, w1;
    char *info, s[MAX_NAME];
    v1 = LocateVex(G, v); /* 尾 */
    w1 = LocateVex(G, w); /* 头 */
    if (v1 < 0 || w1 < 0)
        exit(-1);
    G.arcnum++; /* 弧或边数加1 */
     if (G.kind % 2) /* 网 */ {
        printf("请输入此弧或边的权值：");
        scanf("%d", &G.arcs[v1][w1].adj);
    }
```


示例—增加弧

```
    else /* 图 */
        G.arcs[v1][w1].adj = 1;
    printf("是否有该弧或边的相关信息(0:无 1:有)：");
    scanf("%d%*c", &i);
    if (i) {
        printf("请输入该弧或边的相关信息(<%d个字符)：", MAX_NAME);
        gets_s(s);
        l = strlen(s);
        if (l) {
            info = (char*)malloc((l + 1) * sizeof(char));
            strcpy(info, s);
            G.arcs[v1][w1].info = info;
        }
    }
    if (G.kind>1) /* 无向 */ {
        G.arcs[w1][v1].adj = G.arcs[v1][w1].adj;
        G.arcs[w1][v1].info = G.arcs[v1][w1].info; //指向同一个相关信息
    }
}
```


邻接链表法

基本思想：

对图的每个顶点建立一个单链表，存储该顶点所有邻接顶点及其相关信息。每一个单链表设一个表头结点。

第i个单链表表示依附于顶点v_i的边(对有向图是以顶点v_i为头或尾的弧)。

邻接链表结点结构

每个链表设一个**表头结点**(称为顶点结点)，由两个域组成，**链域**(firstarc)指向链表中的第一个结点，**数据域**(data) 存储顶点名或其他信息。

链表中的结点称为表结点，每个结点由三个域组成，其中**邻接点域**(adjvex)指示与顶点v_i邻接的顶点在图中的位置(顶点编号)，**链域**(nextarc)指向下一个与顶点v_i邻接的表结点，**数据域**(info)存储和边或弧相关的信息，如权值等。对于无权图，如果没有与边相关的其他信息，可省略此域。

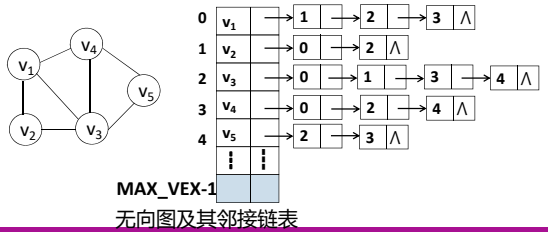
顶点结点：| data | firstarc |　　表结点：| Adjvex | info | nextarc |

邻接链表结点结构

邻接链表示例

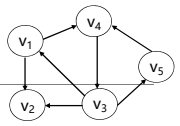
在图的邻接链表表示中，所有顶点结点以顺序结构形式存储，以便随机访问任意顶点的链表。

对无向图使用邻接链表存储图：

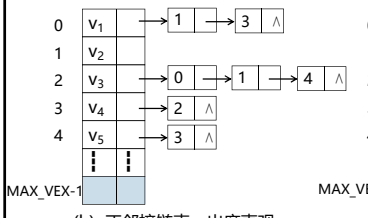


无向图及其邻接链表

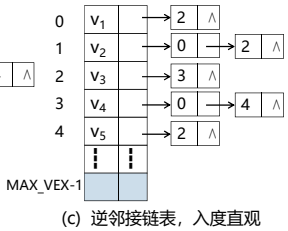
有向图的邻接链表



(a) 有向图



(b) 正邻接链表，出度直观



(c) 逆邻接链表，入度直观

邻接表法的特点

- 表头向量中每个分量就是一个单链表的头结点，分量个数就是图中的顶点数目；
- 在边或弧稀疏的条件下，用邻接表表示比用邻接矩阵表示节省存储空间；
- 在无向图，顶点v_i的度是第i个链表的结点数；
- 对有向图可以建立正邻接表或逆邻接表。正邻接表是以顶点v_i为出度(即为弧的起点)而建立的邻接表；逆邻接表是以顶点v_i为入度(即为弧的终点)而建立的邻接表；
- 在有向图中，第i个链表中的结点数是顶点v_i的出 (或入)度；求入 (或出)度，须遍历整个邻接表；
- 在邻接表上容易找出任一顶点的第一个邻接点和下一个邻接点；

邻接表法类型定义

顶点结点：| data | firstarc |　　表结点：| Adjvex | info | nextarc |

邻接链表结点结构

```
#define MAX_VERTEX_NUM 20
#define MAX_NAME 5 /* 顶点字符串的最大长度+1 */
typedef int VRType;
typedef char InfoType;
typedef char VertexType[MAX_NAME];
typedef enum { DG, DN, AG, AN }GraphKind; /* {有向图,有向网,无向图,
无向网} */
typedef struct ArcNode
{
    int adjvex; /* 该弧所指向的顶点的位置 */
    struct ArcNode *nextarc; /* 指向下一条弧的指针 */
    InfoType *info; /* 网的权值指针） */
}ArcNode; /* 表结点 */
```


邻接表法类型定义

顶点结点：| data | firstarc |　　表结点：| Adjvex | info | nextarc |

邻接链表结点结构

```
typedef struct
{
    VertexType data; /* 顶点信息 */
    ArcNode *firstarc; /* 第一个表结点的地址,指向第一条依附该顶点的
弧的指针 */
}VNode, AdjList[MAX_VERTEX_NUM]; /* 头结点 */

typedef struct
{
    AdjList vertices;
    int vexnum, arcnum; /* 图的当前顶点数和弧数 */
    int kind; /* 图的种类标志 */
}ALGraph;
```


图的遍历

图的遍历(Travering Graph)：从图的某一顶点出发，访遍图中的其余顶点，且每个顶点仅被访问一次。图的遍历算法是各种图的操作的基础。

复杂性：图的任意顶点可能和其余的顶点相邻接，可能在访问了某个顶点后，沿某条路径搜索后又回到原顶点。

解决办法：在遍历过程中记下已被访问过的顶点。设置一个辅助向量$Visited[1\ldots n](n$为顶点数)，其初值为0，一旦访问了顶点v_i后，使$Visited[i]$为1或为访问的次序号。

图的遍历算法有深度优先搜索算法和广度优先搜索算法。采用的数据结构是(正)邻接链表。

深度优先搜索

深度优先搜索(Depth First Search--DFS)遍历类似树的先序遍历，是树的先序遍历的推广。

算法思想：设初始状态时图中的所有顶点未被访问，则：

(1)：从图中某个顶点v_i出发，访问v_i；然后找到v_i的一个邻接顶点v_{i1}；

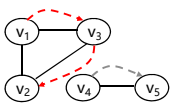
(2)：从v_{i1}出发，深度优先搜索访问和v_{i1}相邻接且未被访问的所有顶点；

(3)：转(1)，直到和v_i相邻接的所有顶点都被访问为止

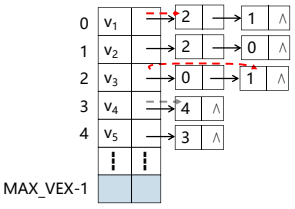
(4)：继续选取图中未被访问顶点v_j作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

DFS示例

无向图的深度优先搜索遍历示例(红色箭头)。某种DFS次序是：$v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$



(a) 无向图G



(b) G的邻接链表

无向图深度优先搜索遍历

算法实现

由算法思想知，这是一个递归过程。因此，先设计一个从某个顶点(编号)为v_0开始深度优先搜索的函数，便于调用。

```
bool visited[MAX_VERTEX_NUM]; /* 访问标志数组(全局量) */
void(*VisitFunc)(VertexType); /* 函数变量 */
void DFS(MGraph G, int v)
{ /* 从第v个顶点出发递归地深度优先遍历图G。算法7.5 */
    VertexType w1, v1;
    int w;
    visited[v] = 1; /* 设置访问标志为TRUE(已访问) */
    VisitFunc(G.vexs[v]); /* 访问第v个顶点 */
    strcpy(v1, *GetVex(G, v));
    for (w = FirstAdjVex(G, v1); w >= 0; w = NextAdjVex(G, v1,
strcpy(w1, *GetVex(G, w))))
        if (!visited[w])
            DFS(G, w); //对v的尚未访问的序号为w的邻接顶点递归调用DFS
}
```


算法实现

```
void DFSTraverse(MGraph G, void(*Visit)(VertexType))
{ /* 初始条件: 图G存在,Visit是顶点的应用函数。算法7.4 */
  /* 操作结果: 从第1个顶点起,深度优先遍历图G,并对每个顶点调用
函数Visit */
  /*           一次且仅一次。一旦Visit()失败,则操作失败 */
    int v;
    VisitFunc = Visit; /* 使用全局变量VisitFunc,使DFS不必设
函数指针参数 */
    for (v = 0; v<G.vexnum; v++)
        visited[v] = 0; /* 访问标志数组初始化(未被访问) */
    for (v = 0; v<G.vexnum; v++)
        if (!visited[v])
            DFS(G, v); /* 对尚未访问的顶点调用DFS */
    printf("\n");
}
```

遍历时，对图的每个顶点至多调用一次DFS函数。其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。当图有e条边，其时间复杂度为O(e)，总时间复杂度为O(n+e)。

广度优先搜索 波纹

广度优先搜索(Breadth First Search--BFS)遍历类似树的按层次遍历的过程。

算法思想：设初始状态时图中的所有顶点未被访问，则：

(1)：从图中某个顶点v_i出发，访问v_i；

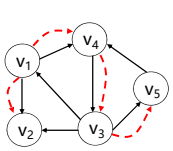
(2)：访问v_i的所有相邻接且未被访问的所有顶点$v_{i1}, v_{i2} \ldots, v_{im}$；

(3)：以v_{i1}，v_{i2}，…，v_{im}的次序，以$v_{ij}(1 \le j \le m)$依次作为v_i，转(1)；

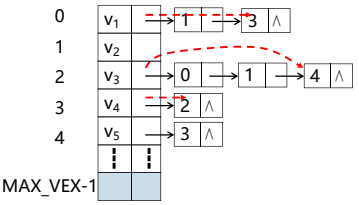
(4)：继续选取图中未被访问顶点v_k作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

广度优先搜索示例

有向图的广度优先搜索遍历示例(红色箭头)。上述图的BFS次序是：$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_5$



(a) 有向图G'



(b) G'的正邻接链表

有向图广度优先搜索遍历

算法实现

为了标记图中顶点是否被访问过，同样需要一个访问标记数组；其次，为了依此访问与v_i相邻接的各个顶点，需要附加一个队列来保存访问v_i的相邻接的顶点。

```
void BFSTraverse(ALGraph G, void(*Visit)(char*))
{/*按广度优先非递归遍历图G。使用辅助队列Q和访问标志
数组visited */
    int v, u, w;
    VertexType u1, w1;
    LinkQueue Q;
    for (v = 0; v<G.vexnum; ++v)
        visited[v] = 0; /* 置初值 */
    InitQueue(&Q); /* 置空的辅助队列Q */
```



```
    for (v = 0; v<G.vexnum; v++) /* 如果是连通图,只v=0就遍历全图
*/
        if (!visited[v]) /* v尚未访问 */ {
            visited[v] = 1;
            Visit(G.vertices[v].data);
            EnQueue(&Q, v); /* v入队列 */
            while (!QueueEmpty(Q)) /* 队列不空 */ {
                DeQueue(&Q, &u); /* 队头元素出队并置为u */
                strcpy(u1, *GetVex(G, u));
                for (w = FirstAdjVex(G, u1); w >= 0; w =
NextAdjVex(G, u1, strcpy(w1, *GetVex(G, w))))
                    if (!visited[w]) /* w为u的尚未访问的邻接
顶点 */ {
                        visited[w] = 1;
                        Visit(G.vertices[w].data);
                        EnQueue(&Q, w); /* w入队 */
                    }
            }
        }
    printf("\n");
}
```

算法分析

用广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是邻接点搜索次序不同，因此对于邻接表的图存储结构来说，深度/广度优先搜索算法遍历图的总时间复杂度为$O(n+e)$。

图的遍历可以系统地访问图中的每个顶点，因此，图的遍历算法是图的最基本、最重要的算法，许多有关图的操作都是在图的遍历基础之上加以变化来实现的。

图的连通性问题

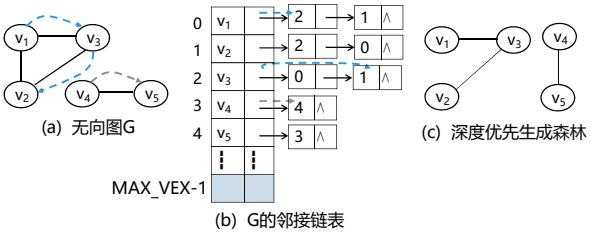
对于无向图，对其进行遍历时：

若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；

若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

图的连通性问题

如图所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，2次调用DFS所得到的顶点访问序列集是：{ v1,v3,v2}和{ v4,v5 }



(a) 无向图G

(b) G的邻接链表

(c) 深度优先生成森林

图的连通性问题

(1) 若$G=(V,E)$是无向连通图，顶点集和边集分别是$V(G)$，$E(G)$。若从G中任意点出发遍历时，$E(G)$被分成两个互不相交的集合：

$T(G)$：遍历过程中所经过的边的集合；

$B(G)$：遍历过程中未经过的边的集合；

显然$E(G)=T(G)\cup B(G)$，$T(G)\cap B(G)=\emptyset$

图$G'=(V,T(G))$是G的极小连通子图，且G'是一棵树。G'称为图G的一棵生成树。

从任意点出发按DFS算法得到生成树G'称为深度优先生成树；按BFS算法得到的G'称为广度优先生成树。

图的连通性问题

(2) 若$G=(V,E)$是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集：$V_1(G),V_2(G),\ldots,V_n(G)$和相应所经过的边集：$T_1(G),T_2(G),\ldots,T_n(G)$。

则对应的顶点集和边集的二元组：

$$G_i=(Vi(G),T_i(G))(1\le i\le n)$$

是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。

说明：当给定无向图要求画出其对应的生成树或生成森林时，必须先给出相应的邻接表，然后才能根据邻接表画出其对应的生成树或生成森林。

最小生成树

如果连通图是一个带权图，则其生成树中的边也带权，生成树中所有边的权值之和称为生成树的代价。

最小生成树(Minimum Spanning Tree)：带权连通图中代价最小的生成树称为最小生成树。

最小生成树在实际中具有重要用途，如设计通信网。设图的顶点表示城市，边表示两个城市之间的通信线路，边的权值表示建造通信线路的费用。n个城市之间最多可以建$n\times(n-1)/2$条线路，如何选择其中的$n-1$条，使总的建造费用最低?

最小生成树

构造最小生成树的算法有许多，基本原则是：

尽可能选取权值最小的边，但不能构成回路；

选择$n-1$条边构成最小生成树。

以上的基本原则是基于MST的如下性质：

设$G=(V,E)$是一个带权连通图，U是顶点集V的一个非空子集。若$u\in U$，$v\in V-U$，且(u,v)是U中顶点到$V-U$中顶点之间权值最小的边，则必存在一棵包含边(u,v)的最小生成树。

最小生成树

用反证法证明。

设图G的任何一棵最小生成树都不包含边(u,v)。设T是G的一棵最小生成树，则T是连通的，从u到v必有一条路径$(u,\dots,v)$，当将边(u,v)加入到T中时就构成了回路。则路径$(u,\dots,v)$中必有一条边(u',v')，满足$u'\in U$，$v'\in V-U$。删去边(u',v')便可消除回路，同时得到另一棵生成树T'。

由于(u,v)是U中顶点到$V-U$中顶点之间权值最小的边故(u,v)的权值不会高于(u',v')的权值，T'的代价也不会高于T，T'是包含(u,v)的一棵最小生成树，与假设矛盾。

Prim算法　从顶点入手，找最小边

从连通图$N=(U,E)$中找最小生成树$T=(U,TE)$。

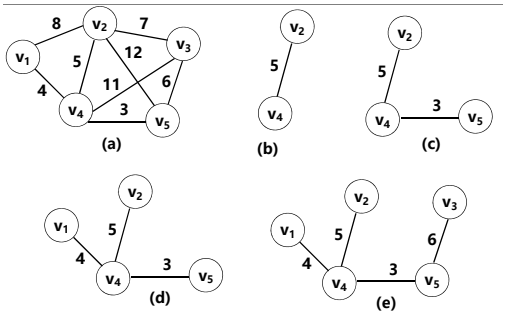
算法思想：

(1) 若从顶点v_0出发构造，$U=\{v_0\}$，$TE=\{\}$；

(2) 先找权值最小的边(u,v)，其中$u\in U$且$v\in V-U$，并且子图不构成环，则$U=U\cup\{v\}$，$TE=TE\cup\{(u,v)\}$；

(3) 重复(2)，直到$U=V$为止。则TE中必有$n-1$条边，$T=(U,TE)$就是最小生成树。

Prim算法示例



按Prim算法从v_2出发构造最小生成树的过程

算法实现

使用辅助数组closeedge来记录从顶点集U到V-U的代价最小的边。

```
typedef struct
{ /* 记录从顶点集U到V-U的代价最小的边的辅助数组定义 */
    VertexType adjvex;
    VRType lowcost;
}minside[MAX_VERTEX_NUM];
int minimum(minside SZ, MGraph G) { // 求closedge.lowcost的最小正值
   int i = 0, j, k, min;
   while (!SZ[i].lowcost) i++;
   min = SZ[i].lowcost; k = i; /* min为第一个不为0的值 */
   for (j = i + 1; j < G.vexnum; j++)
       if (SZ[j].lowcost>0)
           if (min > SZ[j].lowcost) {
               min = SZ[j].lowcost;
               k = j;}
   return k;
}
```



```
void MiniSpanTree_PRIM(MGraph G, VertexType u)
{ /* 用普里姆算法从第u个顶点出发构造网G的最小生成树T,输出T的各条边 */
    int i, j, k;
    minside closedge;
    k = LocateVex(G, u);
    for (j = 0; j < G.vexnum; ++j) /* 辅助数组初始化 */{
        if (j != k) {
            strcpy(closedge[j].adjvex, u);
            closedge[j].lowcost = G.arcs[k][j].adj;
        }
    }
    closedge[k].lowcost = 0; /* 初始,U={u} */
    printf("最小代价生成树的各条边为:\n");
    for (i = 1; i < G.vexnum; ++i) { /* 选择其余G.vexnum-1个顶点 */
        k = minimum(closedge, G); /* 求出T的下一个结点：第K顶点 */
        printf("(%s-%s)\n", closedge[k].adjvex, G.vexs[k]);
        closedge[k].lowcost = 0; /* 第K顶点并入U集 */
        for (j = 0; j < G.vexnum; ++j)
            if (G.arcs[k][j].adj < closedge[j].lowcost)
            { /* 新顶点并入U集后重新选择最小边 */
                strcpy(closedge[j].adjvex, G.vexs[k]);
                closedge[j].lowcost = G.arcs[k][j].adj;
            }
    }
}
```

克鲁斯卡尔(Kruskal)算法

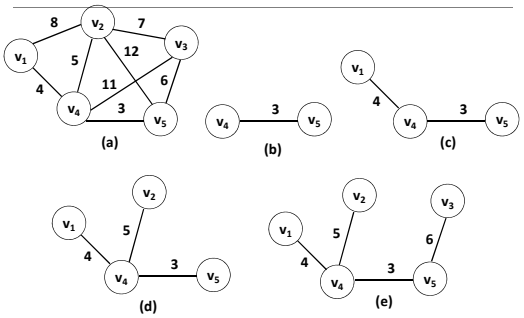
设$G=(V,E)$是具有n个顶点的连通网，$T=(U,TE)$是其最小生成树。初值：$U=V$，$TE=\{\}$。

算法思想：对G中的边按权值大小从小到大依次选取。

(1)选取权值最小的边(vi, v_j)，若边(vi, v_j)加入到TE后形成回路，则舍弃边(vi, v_j)；否则，将该边并入到TE中，即$TE=TE\cup\{(vi, v_j)\}$。

(2) 重复(1)，直到TE中包含有$n-1$条边为止。

Kruskal算法示例



按kruskal算法构造最小生成树的过程

算法分析

Prim算法包含两个内循环，时间复杂度为$O(n^2)$，与图的边数无关，因此适用于求边稠密的图的最小生成树。

而Kruskal算法时间复杂度为$O(|E|\log|E|)$，其中$|E|$为图中的边数，相较于Prim算法更适用于边稀疏而顶点较多的图。

最短路径

若用带权图表示交通网，图中顶点表示地点，边代表两地之间有直接道路，边上的权值表示路程(或所花费用或时间)。从一个地方到另一个地方的路径长度表示该路径上各边的权值之和。问题：

两地之间是否有通路?

在有多条通路的情况下，哪条最短?

考虑到交通网的有向性，直接讨论的是带权有向图的最短路径问题，但解决问题的算法也适用于无向图。

将一个路径的起始顶点称为**源点**，最后一个顶点称为**终点**。

Dijkstra算法

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。

Dijkstra算法的主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法。

注意该算法要求图中不存在负权回路。

算法思想

(1)初始时，集合S只包含源点，即$S=\{v\}$，$D(v)=0$。集合U包含除v外的其它顶点，即: $U=\{$其余顶点$\}$，若v与U中顶点u有边，则$<u,v>$正常有权值$D(u)=W(v,u)$，否则权值为∞。($W(v,u)$表示v到u的权值，$D(u)$表示最短路径值)

(2)从U中选取一个距离最小的顶点k，把k加入集合S中（该选定距离$D(k)$就是v到k的最短路径长度）。

$$k=argmin(D(i), i\in U)$$

算法思想

(3)以k为新考虑的中间点，修改U中各顶点的距离：若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值。

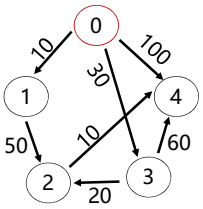
$$D(u) = \min\{D(u), D(k) + W(k,u)\}$$

(4)重复步骤(2)和(3)直到所有顶点都包含在 S 中。

67

示例

从顶点0出发，$S = \{0\}$,寻找到其他顶点的最短路径



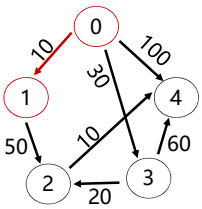
循环	S	k	D(0)	D(1)	D(2)	D(3)	D(4)
初始化	{0}	-	0	10	∞	30	100
1	{0,1}	1					
2	{0,1,3}	3					
3	{0,1,3,2}	2					
4	{0,1,3,2,4}	4					

$U = \{1,2,3,4\}$

从U中选择距离最小的顶点$k = 1$加入S中。

68

示例



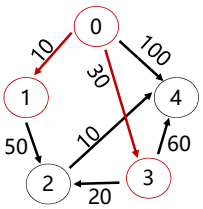
循环	S	k	D(0)	D(1)	D(2)	D(3)	D(4)
初始化	{0}	-	0	10	∞	30	100
1	{0,1}	1	0	10	60	30	100
2	{0,1,3}	3	0				
3	{0,1,3,2}	2	0				
4	{0,1,3,2,4}	4	0				

$U = \{2,3,4\}$

从U中选择距离最小的顶点$k = 3$加入S中。

69

示例



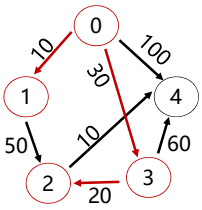
循环	S	k	D(0)	D(1)	D(2)	D(3)	D(4)
初始化	{0}	-	0	10	∞	30	100
1	{0,1}	1	0	10	60	30	100
2	{0,1,3}	3	0	10	50	30	90
3	{0,1,3,2}	2	0				
4	{0,1,3,2,4}	4	0				

$U = \{2,4\}$

从U中选择距离最小的顶点$k = 2$加入S中。

70

示例



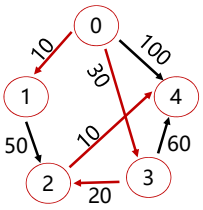
循环	S	k	D(0)	D(1)	D(2)	D(3)	D(4)
初始化	{0}	-	0	10	∞	30	100
1	{0,1}	1	0	10	60	30	100
2	{0,1,3}	3	0	10	50	30	90
3	{0,1,3,2}	2	0	10	50	30	60
4	{0,1,3,2,4}	4	0				

$U = \{4\}$

从U中选择距离最小的顶点$k = 4$加入S中。

71

示例



循环	S	k	D(0)	D(1)	D(2)	D(3)	D(4)
初始化	{0}	-	0	10	∞	30	100
1	{0,1}	1	0	10	60	30	100
2	{0,1,3}	3	0	10	50	30	90
3	{0,1,3,2}	2	0	10	50	30	60
4	{0,1,3,2,4}	4	0	10	50	30	60

$U = \emptyset$

算法完成。

72

算法实现

```
typedef int VRType;
typedef char InfoType;
typedef char VertexType[MAX_NAME];
typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef int ShortPathTable[MAX_VERTEX_NUM];
void ShortestPath_DIJ(MGraph G,int v0,PathMatrix &P,ShortPathTable &D)
{ /* 用Dijkstra算法求有向网G的v0顶点到其余顶点v的最短路径P[v]及带权长度 */
  /* D[v]。若P[v][w]为TRUE,则w是从v0到v当前求得最短路径上的顶点。 */
    int v, w, i, j, min;
    bool final[MAX_VERTEX_NUM];
    for (v = 0; v < G.vexnum; ++v) {
        final[v] = 0;
        D[v] = G.arcs[v0][v].adj;
        for (w = 0; w < G.vexnum; ++w) P[v][w] = 0; /* 设空路径 */
        if (D[v] < INFINITY) {
            P[v][v0] = 1;P[v][v] = 1;} }
    D[v0] = 0;
    final[v0] = 1; /* 初始化,v0顶点属于S集 */
```



```
    for (i = 1; i < G.vexnum; ++i) /* 其余G.vexnum-1个顶点 */
        { /* 开始主循环,每次求得v0到某个v顶点的最短路径,并加v到S集 */
            min = INFINITY; /* 当前所知离v0顶点的最近距离 */
            for (w = 0; w < G.vexnum; ++w)
                if (!final[w]) /* w顶点在V-S中 */
                    if (D[w] < min)
                    {
                        v = w;
                        min = D[w];
                    } /* w顶点离v0顶点更近 */
            final[v] = 1; /* 离v0顶点最近的v加入S集 */
            for (w = 0; w < G.vexnum; ++w) /* 更新当前最短路径及距离 */
            {
                if (!final[w] && min < INFINITY&&G.arcs[v][w].adj <
INFINITY && (min + G.arcs[v][w].adj < (*D)[w]))
                { /* 修改D[w]和P[w],w∈V-S */
                    D[w] = min + G.arcs[v][w].adj;
                    for (j = 0; j < G.vexnum; ++j)
                        P[w][j] = P[v][j];
                    P[w][w] = 1;
                }
            }
        }
    }
```


Floyd算法

问题：已知一个各边权值均大于0的带权有向图，对每一对顶点 $v_i \neq v_j$，要求求出v_i与v_j之间的最短路径和最短路径长度。

算法思想：若$(v_i, \dots, v_k)$和$(v_k, \dots, v_j)$分别是从v_i到v_k和从v_k到v_j的中间顶点的序号不大于$k-1$的最短路径，则将$(v_i, \dots, v_k, \dots, v_j)$和已经得到的从$v_i$到$v_j$且中间顶点的序号不大于$k-1$的最短路径相比较，其长度较短者便是从$v_i$到$v_j$的中间顶点序号不大于$k$的最短路径。这样，经过$n$次比较后，最后求得的必是从$v_i$到$v_j$的最短路径。按此方法可以同时求得各对顶点间的最短路径。

Floyd算法

定义一个n阶方阵序列：

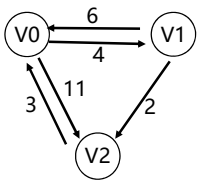
$$A^{(-1)},\ A^{(0)},\ \dots, A^{(n-1)}.$$

其中$A^{(-1)}[i][j] = Edge[i][j]$;

$A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}$, $k = 0,1,\dots,n-1$

- $A^{(-1)}[i][j]$是从顶点v_i到v_j,无中间顶点的最短路径长度;
- $A^{(k)}[i][j]$是从顶点v_i 到v_j，中间顶点的序号不大于k的最短路径的长度;
- $A^{(n-1)}[i][j]$是从顶点v_i到v_j的最短路径长度。

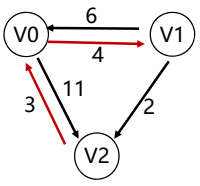
示例



$$D^{(-1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

递推求解$D^{(n-1)}$矩阵

示例

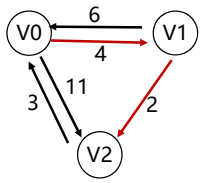


$$D^{(-1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

递推求解$D^{(n-1)}$矩阵

示例

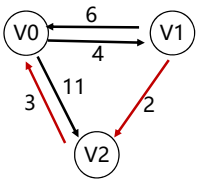


$$D^{(0)}=\begin{bmatrix}0 & 4 & 11\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

$$D^{(1)}=\begin{bmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

递推求解$D^{(n-1)}$矩阵

示例



$$D^{(1)}=\begin{bmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

$$D^{(2)}=\begin{bmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

$D^{(2)}$为图任意两个顶点的最短距离矩阵

算法实现

```
typedef int VRType;
typedef char VertexType[MAX_NAME];
typedef char InfoType;
typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef int DistancMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
void ShortestPath_FLOYD(MGraph G, PathMatrix &P, DistancMatrix &D)
{ /* 用Floyd算法求有向网G中各对顶点v和w之间的最短路径P[v][w]及其 */
  /* 带权长度D[v][w]。
    int u, v, w, i;
    for (v = 0; v < G.vexnum; v++) /* 各对结点之间初始已知路径及距离 */
        for (w = 0; w < G.vexnum; w++) {
            D[v][w] = G.arcs[v][w].adj;
            for (u = 0; u < G.vexnum; u++)
                P[v][w][u] = 0;
            if (D[v][w] < INFINITY) /* 从v到w有直接路径 */ {
                P[v][w][v] = 1;
                P[v][w][w] = 1;
            }
        }
```


算法实现

```
 for (u = 0; u < G.vexnum; u++)
        for (v = 0; v < G.vexnum; v++)
            for (w = 0; w < G.vexnum; w++)
                if (D[v][u] + D[u][w] < D[v][w]) /* 从v经u到w的一
条路径更短 */
                {
                    D[v][w] = D[v][u] + D[u][w];
                    for (i = 0; i < G.vexnum; i++)
                        P[v][w][i] = P[v][u][i] || P[u][w][i];
                }
}
```