第六节：图（上）

6.1 图

1.关于图

图表示的是“多对多”的关系。它包含：

（1）一组顶点：通常用 V（Vertex）表示顶点集合。

（2）一组边：通常用 E（Edge）表示边的集合，表示顶点与顶点的关系：

①边是顶点对：(v,w)∈E，其中 v,w∈V。这是一个双向的。

②有向边：<v,w>，表示从 v 指向 w 的边（单行线）。

③不考虑重边和自回路。

其抽象数据类型为：

类型名称：图（Graph） **数据对象集：一非空的顶点集合 Vertex 和一个边集合 Edge，每条边用对应的一对顶点表示。** **操作集：对于任意的图 G ∈ Graph，顶点 v、v_1 和 v_2 ∈ ertex，以及任一访问顶点的函数 visit()，操作举例：** **Graph Create( )：构造并返回一个空图；** **void Destroy(Graph G)：释放图 G 占用的存储空间；** **Graph InsertVertex(Graph G, Vertex v)：返回一个在 G 中增加了新顶点 v 的图** **Graph InsertEdge(Graph G, Vertex v_1, Vertex v_2)：返回一个在 G 中增加了新边 (v_1, v_2) 的图；** **Graph DeleteVertex(Graph G, Vertex v)：删除 G 中顶点 v 及其相关边，将结果图返回；** **Graph DFS（ Graph G, Vertex v, visit())：在图 G 中，从顶点 v 出发进行深度优先遍历；**

图的一些分类：

（1）**无向图**。边（v, w）等同于边（w, v）。用圆括号“（）”表示无向边。

（2）**有向图**（Directed Graphs）：边<v, w>不同于边<w, v>。用尖括号“< >”表示有向边；有向边也称“弧（Arc）”。

（3）**简单图**（Simple Graphs）:没有重边和自回路的图。

（4）**邻接点**：如果（v, w）或 < v, w >是图中任意一条边，那么称 v 和 w 互为“邻接点（Adjacent Vertices）”。

（5）**路径、简单路径、回路、无环图**：

图 G 中从 v_p 到 v_q 的路径 $= \{ v_p, v_{i1}, v_{i2}, \cdots, v_{in}, v_q \}$ 使得$(v_p, v_{i1}), (v_{i1}, v_{i2}), \cdots, (v_{in}, v_q)$ 或 $< v_p, v_{i1} >, \cdots, < v_{in}, v_q >$ 都属于 E(G)。

路径长度：路径中边的数量。

简单路径：$v_{i1}, v_{i2}, \cdots, v_{in}$ 都是不同顶点。

回路：起点和终点相同（$v_p = v_q$ ）的路径。

无环图：不存在任何回路的图。

有向无环图：不存在回路的有向图，也称 DAG（Directed Acyclic Graph）。

（6）**完全图**：分为有向完全图（n(n-1)条边）和无向完全图（n(n-1)/2 条边）。

（7）**度**：与顶点 v 相关的边数。从该点发出的边数为“出度”，指向该点的边数为“入度”。

（8）**稠密图、稀疏图**：是否满足 |E| > |V|log2|V|，作为稠密图和稀疏图的分界条件。

（9）**权（Cost）、网络（Network）**。

（10）**图 G 的子图 G'：V(G') ⊆ V(G) && E(G') ⊆ E(G)。**

（11）无向图的顶点连通、连通图、连通分量：如果无向图从一个顶点 v_i 到另一个顶点 v_j $(i \neq j)$ 有路径，则称顶点 v_i 和 v_j 是“连通的（Connected）”；无向图中任意两顶点都是连通的，则称该图是“连通图（Connected Graph）”；无向图的极大连通子图称为“连通分量（Connected Component）”。连通分量的概念包含以下 4 个要点：*子图、连通、极大顶点数、极大边数*

（12）有向图的强连通图、连通分量：有向图中任意一对顶点 v_i 和 v_j $(i \neq j)$ 均既有从 v_i 到 v_j 的路径，也有从 v_j 到 v_i 的路径，则称该有向图是“强连通图（Strongly Connected Graph）”。有向图的极大强连通子图称为“强连通分量（Strongly Connected Component）”。连通分量的概念也包含前面 4 个要点。

（13）树、生成树：树是图的特例：无环的无向图。

所谓连通图 G 的“生成树（Spanning Tree）”，是 G 的包含其全部 n 个顶点的一个极小连通子图。它必定包含且仅包含 G 的 n-1 条边。

生成树有可能不唯一。

当且仅当 G 满足下面 4 个条件之一（完全等价）：

① G 有 n-1 条边，且没有环；

② G 有 n-1 条边，且是连通的；

③ G 中的每一对顶点有且只有一条路径相连；

④ G 是连通的，但删除任何一条边就会使它不连通。

2.表示图——邻接矩阵

邻接矩阵 G[N][N]-N 个顶点从 0 到 N-1 编号。其中 G[i][j]=1（若<v_i,v_j>是 G 中的边）或 0（不是 G 中的边）。可以看出邻接矩阵有以下特点：

（1）主对角线全为 0；

（2）这是一个对称矩阵。

那么，对于无向图来说，怎样可以节省一半空间？

可以用一个长度为 N(N+1)/2 的 1 维数组 A 存储{G00,G10,…,G(n-1)0,…,G(n-1)(n-1)}。则 Gij 在 A 中对应的下标是：(i*(i+1)/2+j)。对于网络，只要把 G[i][j]的值定义为边<vi,vj>的权重即可。

问题：vi 和 vj 在网络中它们之间没有边该如何表示？

邻接矩阵有以下优点：

（1）直观、简单、好理解；

（2）方便检查任意一对顶点间是否存在边；

（3）方便找任一顶点的所有“邻接点”（有边直接相连的顶点）；

（4）方便计算任一顶点的“度”：

对于无向图来说，对应行（列）非 0 的元素的个数。对于有向图来说，对应行非 0 元素的个数是“出度”，对应列非 0 元素的个数是“入度”。

邻接矩阵的缺点：

（1）浪费空间：存稀疏图有大量无效元素。但是对于稠密图（特别是完全图）还是很合算。

（2）浪费时间：统计系数图中一共有多少条边。

3.表示图——邻接表

邻接表：G[N]为指针数组，对应矩阵每行一个链表，只存非 0 元素。对于图 G 中的每

个顶点 v_i，将所有邻接于 v_i 的顶点 v_j 链成一个单链表，这个单链表就称为顶点 v_i 的邻接表，再将所有点的邻接表表头放到一个数组中，就构成了图的邻接表。

一定要够稀疏用邻接表才合算!!!!

邻接表的特点：

（1）方便找任一顶点的所有“邻接点”。

（2）节约稀疏图的空间：需要 N 个头指针+2E 个结点。

（3）方便计算任一顶点的“度”：**对于无向图来说是如此，对于有向图来说，这只能计算“出度”；需要构造“逆邻接表”（存指向自己的边）来方便计算“入度”。**

6.2 图的遍历

1.深度优先搜索（DFS）

代码实现：

```
void DFS( Graph G,   int V )
{ /* 从第 V 个顶点出发递归地深度优先遍历图 G */
      VertexType W;
        Visited[V] = TRUE;
      VisitFunc(V);          /* 访问第 V 个顶点 */
      for( W = FirstAdjV(G, V);   W;   W = NextAdjV (G, V, W) )
           if( !Visited[W] )
               DFS(G, W); /* 对 V 的尚未访问的邻接顶点 W 递归调用 DFS */
}
```

这段代码的意思是，首先我们开始访问，然后每访问一个节点，都将其标记为 True，然后开始访问 V 的邻接点，如果没访问，那就去访问并且置为 True，当看到邻接点都是 True 时，我们原路返回，若发现没访问过的邻接点，立即去访问；如果没有发现，继续原路返回，直到返回到第一个结点。

若由 N 个顶点，E 跳边，时间复杂度是：

（1）用邻接表存储图：O(N+E);

（2）用邻接矩阵，有 $O(N^2)$。

2.广度优先搜索（BFS）

相当于层序遍历。

```
void BFS(Graph G)
{    /* 按广度优先遍历图 G。使用辅助队列 Q 和访问标志数组 Visited */
          Queue *Q;     VertexType U, V, W;
          for ( U = 0; U < G.n; ++U )   Visited[U] = FALSE;
          Q = CreatQueue( MaxSize ); /* 创建空队列 Q */
          for ( U = 0; U<G.n; ++U )
       if (!Visited[U] ) {                /* 若 U 尚未访问 */
                Visited[U] = TRUE;
                          VisitFunc(U);           /* 访问 U */
                AddQ (Q, U);            /* U 入队列 */
                while ( ! IsEmptyQ(Q) ) {
                     V = DeleteQ( Q );   /*   队头元素出队并置为 V */
                     for( W = FirstAdjV(G, V);   W;   W = NextAdjV(G, V, W) )
```

```
                    if ( !Visited[W] ) {
            Visited[W] = TRUE;
                            VisitFunc (W);      /* 访问 W */
                        AddQ (Q, W);
        }
            } /* while 结束*/
              } /* 结束从 U 开始的 BFS */
}
```

若由 N 个顶点，E 跳边，时间复杂度是：

（1）用邻接表存储图：O(N+E)；

（2）用邻接矩阵，有 $O(N^2)$。

6.3 图的应用

1.应用实例：拯救 007

如图 1 所示，如何让 007 通过这一个个结点（鳄鱼），一步一步跳到岸边呢？这里用深度优先算法更为合适。

原来的总体算法（伪代码）：

```
void ListComponents(Graph G)
{
    for(each V in G)
        if(!visited[V]){
            DFS(V);
        }
}
```

这里的总体算法（伪代码）：

```
void Save007(Graph G)
{
    for(each V in G)
        if(!visited[V]&& FirstJump(V))/*判断是否踩过这个鳄鱼还有是否能够得着*/
        {
            answer = DFS(V);
            if(answer ==YES) break;
        }
        if(answer ==YES) output("YES");
        else output("BYEBYE");
}
```

DFS 部分的伪代码：

```
int DFS(Vertex V)
{
    visited[V] = True;
    if(IsSafe(V)) answer = YES;
    else{
        for(V 的每个邻接点 W)
```

```
            if(!visited[W]&& Jump(V,W))
                /*Jump 计算两个鳄鱼之间距离是否小于 007 最大跳动距离*/
            {
                answer=DFS(W);
                DFS(W);
            }
    }
    return answer;
}
```

6.4 图的建立

1.用邻接矩阵表示图

```
typedef struct GNode *PtrToGNode;
struct GNode{
    int Nv;/*顶点数*/
    int Ne;/*边数*/
    WeightType G[MaxVertexNum][MaxVertexNum];
    DataType Data[MaxVertexNum];/*存顶点的数据*/
};
typedef PtrToGNode MGraph;/*以邻接矩阵存储的图类型*/
```

2.初始化图

```
typedef int Vertex;/*用顶点下标表示顶点，为整型*/
MGraph CreateGraph(int VertexNum)
{
    Vertex V,W;
    MGraph Graph;
    Graph = (MGraph)malloc(sizeof(struct GNode));
    Graph->Nv = VertexNum;
    Graph->Ne = 0;

    /*这里默认顶点标号从 0 开始，到(Graph->Nv-1)*/
    for(V=0;V<Graph->Nv;V++)
        for(W=0;W<Graph->Nv;W++)
            Graph->G[V][W] = 0;/*有向图是 INFINITY*/
    return Graph;
}
```

3.插入边

```
typedef struct ENode *PtrToGNode;
struct ENode
{
    Vertex V1,V2;/*有向边<V1,V2>*/
    WeightType Weight;/*权重*/
};
```

```
typedef PtrToGNode Edge;

void InsertEdge( MGraph Graph, Edge E)
{
	/*插入边<V1,V2>*/
	Graph->G[E->V1][E->V2]=E->Weight;
	/*无向图*/
	Graph->G[E->V2][E->V1]=E->Weight;

}
```

4.建立图

（1）输入格式：Nv Ne

V1 V2 Weight

代码如下：

```
MGraph BuildGraph()
{
	Edge E;
	MGraph Graph;
	Vertex V;
	int Nv,i;
	scanf("%d",&Nv);
	Graph = CreateGraph(Nv);
	scanf("%d",&(Graph->Ne));
	if(Graph->Ne!=0)
	{
		E = (Edge)malloc(sizeof(struct ENode));
		for(i=0;i<Graph->Ne;i++){
			scanf("%d %d %d",&E->V1,&E->V2,&E->Weight);
			InsertEdge(Graph, E);
		}
	}
	/*如果顶点有数据的话，读入数据*/
	for(V=0;V<Graph->Nv;V++)
		scanf("%c",&(Graph->Data[V]));

	return Graph;
}
```

但是这样来说太麻烦了，如果不要这么麻烦，一个**整体的程序（不再需要子程序）**可以如下：

```
int G[MAXN][MAXN],Nv,Ne;
void BuildGraph()
{
	int i,j,v1,v2,w;
	scanf("%d",&Nv);
```

```
	/*CreateGraph*/
	for(i=0;i<Nv;i++)
		for(j=0;j<Nv;j++)
			G[i][j]=0;/*初始化，有向图为无穷*/
	scanf("%d", &Ne);/*输入有多少个结点*/
	for(i=0;i<Ne;i++)
	{
		scanf("%d %d %d", &v1,&v2,&w);
		/*将边插入*/
		G[v1][v2]=w;
		G[v2][v1]=w;
	}
}
```

一气呵成。

5.邻接表表示的图结点的结构

邻接表：G[N]为指针数组，对应矩阵每行一个链表，只存非 0 元素。

```
typedef struct GNode *PtrToGNode;
struct GNode{
	int Nv;/*顶点数*/
	int Ne;/*边数*/
	AdjList G;/*邻接表*/
};
typedef PtrToGNode LGraph;

typedef struct Vnode{
	PtrToAdjVNode FirstEdge;
	DataType Data;/*存顶点的数据*/
}AdjList[MaxVertexNum];
/*AdjList 是邻接表类型*/

typedef struct AdjVNode *PtrToAdjVNode;
struct AdjVNode{
	Vertex AdjV;/*邻接点下标*/
	WeightType Weight;
	PtrToAdjVNode Next;/*指向下一个邻接点的指针*/
};
```

6.邻接表表示的图-建立图

（1）初始化一个由 VertexNum 个顶点但没有边的图

```
typedef int Vertex;/*用顶点下标表示顶点，为整型*/
LGraph CreateGraph(int VertexNum)
{
	Vertex V,W;
	LGraph Graph;
```

```
    Graph = (LGraph)malloc(sizeof(struct GNode));
    Graph->Nv = VertexNum;
    Graph->Ne =0;
    /*每一个顶点跟着的链表都是空的为没有边*/
    for(V=0;V<Graph->Nv;V++)
        Graph->G[V].FirstEdge = NULL;
    return Graph;
}
```

（2）向 LGraph 中插入边

```
void InsertEdge(LGraph Graph, Edge E)
{
    PtrToAdjVNode NewNode;

    /*插入边<v1,v2>*/
    NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));
    NewNode->AdjV=E->V2;
    NewNode->Weight = E->Weight;
    /*将 V2 插入 V1 的表头*/
    NewNode-Next = Graph->G[E->V1].FirstEdge;
    Graph->G[E->V1].FirstEdge = NewNode;

    /*若是无向图，还要插入边<V2,V1>*/
    /*为 V1 建立新的邻接点*/
    NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));
    NewNode->AdjV=E->V1;
    NewNode->Weight = E->Weight;
    /*将 V1 插入 V2 的表头*/
    NewNode-Next = Graph->G[E->V2].FirstEdge;
    Graph->G[E->V2].FirstEdge = NewNode;
}
```

（3）完整建立 LGraph

```
LGraph BuildGraph()
{
    LGraph Graph;
/*剩余与邻接矩阵完整版类似*/
}
```