图

7.1 图的基本概念

一. 图的基本术语

1. 图的定义

记为 $G=(V,E)$

- $V=\text{vertex}$：顶点集合，是有穷非空集；
- $E=\text{edge}$：边集合，是有穷集。

问：当 $E(G)$ 为空时，图 G 存在否?

答：还存在！但此时图 G 只有顶点而没有边。

2. 有向图与无向图

- **有向图**：图 G 中的每条边都是有方向的；
- **无向图**：图 G 中的每条边都是无方向的；

3. 完全图

- 图 G 任意两个顶点都有一条边相连接；
  - 若 n 个顶点的无向图有 $\frac{n(n-1)}{2}$ 条边，称为 **无向完全图**；
  - 若 n 个顶点的有向图有 $n(n-1)$ 条边，称为 **有向完全图**。

4. 稀疏图与稠密图

- **稀疏图**：边较少的图。通常边数远少于 $n\log n$；
- **稠密图**：
  - **无向图中**，边数接近 $\frac{n(n-1)}{2}$；
  - **有向图中**，边数接近 $n(n-1)$。

5. 子图

设有两个图 $G=(V,E)$ 和 $G'=(V',E')$。若 $V'\subseteq V$ 且 $E'\subseteq E$，则称图 G' 是图 G 的子图。

6. 带权图与网络

- **带权图**：即边上带权的图。其中权是指每条边可以标上具有某种含义的数值（即与边相关的数）；
- **网络**：$\rightarrow$ 带权图。

7. 连通图与强连通图

- **连通图**：
  - 在无向图中，若从顶点 v_1 到顶点 v_2 有路径，则称顶点 v_1 与 v_2 是 **连通** 的；
  - 如果图中任意一对顶点都是连通的，则称此图是 **连通图**；
  - 非连通图的极大连通子图叫做 **连通分量**。
- **强连通图**：
  - 在有向图中，若对于每一对顶点 v_i 和 v_j，都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径，则称此图是 **强连通图**；
  - 非强连通图的极大强连通子图叫做 **强连通分量**。

8. 生成树与生成森林

- **生成树**：
  - 是一个极小连通子图，它含有图中全部 n 个顶点，但只有 $n-1$ 条边；
  - 如果在生成树上添加 1 条边，必定构成一个环；
  - 若图中有 n 个顶点，却少于 $n-1$ 条边，必为非连通图。
- **生成森林**：
  - 若干棵生成树的集合，包含全部顶点，但构成这些树的边或弧是最少的。

9. 邻接点、弧头与弧尾

- **邻接点**：若 (u,v) 是 $E(G)$ 中的一条边，则称 u 与 v 互为邻接顶点；
- **弧头和弧尾**：有向边 (u,v) 称为弧，边的始点 u 叫做弧尾，终点 v 叫做弧头。

10. 度、入度与出度

- **度**：
  - 顶点 v 的度是与它相关联的边的条数，记作 $\mathrm{TD}(v)$；
  - 在有向图中，顶点的度等于该顶点的入度与出度之和。
- **入度与出度**：
  - 顶点 v 的入度是以 v 为终点的有向边的条数，记作 $\mathrm{ID}(v)$；
  - 顶点 v 的出度是以 v 为始点的有向边的条数，记作 $\mathrm{OD}(v)$。
- 问：当有向图中仅 1 个顶点的入度为 0，其余顶点的入度均为 1，此时是何形状？
- 答：是树！而且是一棵有向树！

11. 路径、路径长度、简单路径与回路

- **路径**：
  - 在图 $G=(V,E)$ 中，若从顶点 v_i 出发，沿一些边经过一些顶点 $v_{p_1},v_{p_2},\dots,v_{p_m}$ 到达顶点 v_j，则称顶点序列 $(v_i,v_{p_1},v_{p_2},\dots,v_{p_m},v_j)$ 为从顶点 v_i 到顶点 v_j 的路径。
  - 它经过的边 $(v_i,v_{p_1}),(v_{p_1},v_{p_2}),\dots,(v_{p_m},v_j)$ 应当是属于 E 的边。
- **路径长度**：

    - 非带权图的路径长度是指此路径上边的条数；
    - 带权图的路径长度是指路径上各边的权之和。
- **简单路径**：路径上各顶点 $v_1, v_2, \ldots, v_m$ 均不互相重复。
- **回路**：若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。

二. 图的基本操作

CreateGraph(&G, num);

- **操作结果**：构造一个顶点数为 `num`，边数为 0 的图。

DestroyGraph(&G);

- **初始条件**：图 G 存在。
- **操作结果**：销毁图 G。

FirstAdjVex(G, v);

- **初始条件**：图 G 存在，v 是 G 中某个顶点。
- **操作结果**：返回 v 的第一个邻接点。若该顶点在 G 中没有邻接点，则返回“空”。
- 若 $\langle v, w \rangle \in G$，则称 w 为 v 的邻接点；
- 若 $(v, w) \in G$，则称 w 和 v 互为邻接点。

NextAdjVex(G, v, w);

- **初始条件**：图 G 存在，v 是 G 中某个顶点，w 是 v 的邻接顶点。
- **操作结果**：返回 v 的（相对于 w 的）下一个邻接点。若 w 是 v 的最后一个邻接点，则返回“空”。

InsertEdge(&G, v, 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 中增添以 v, w 为顶点的边。

DeleteEdge(&G, v, 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 中删除以 v, w 为顶点的边。

7.2 图的存储结构

一. 邻接矩阵（数组）表示法

1. 基本思想

建立一个顶点表和一个邻接矩阵：

- **顶点表**：记录各个顶点信息；
- **邻接矩阵**：表示各个顶点之间关系。

设图 $A=(V,E)$ 有 n 个顶点，则图的邻接矩阵是一个二维数组 `A.Edge[n][n]`，定义为：

$$\mathrm{A.Edge}[i][j]=\begin{cases}1, & \text{如果 } <i,j>\in E \text{ 或者 } (i,j)\in E\\ 0, & \text{否则}\end{cases}$$

2. 示例

- **顶点表**：(v_1,v_2,v_3,v_4,v_5)
- **邻接矩阵**：

$$\mathrm{A.Edge}=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&1\\0&1&1&1&0\end{bmatrix}$$

分析：

1. 无向图的邻接矩阵是对称的；
2. 顶点 i 的度 = 第 i 行（列）中 1 的个数；
3. 特别：完全图的邻接矩阵中，对角元素为 0，其余全 1。

3. 有权图（即网络）的邻接矩阵

定义：

$$\mathrm{N.Edge}[i][j]=\begin{cases}W_{ij} & \text{如果 } <v_i,v_j> \text{ 或 } (v_i,v_j)\in VR\\ \infty & \text{无边（弧）}\end{cases}$$

示例：

- **顶点表**：$(v_1,v_2,v_3,v_4,v_5,v_6)$
- **邻接矩阵**：

$$\mathrm{N.Edge}=\begin{bmatrix}\infty&5&\infty&7&\infty&\infty\\\infty&\infty&4&\infty&\infty&\infty\\8&\infty&\infty&\infty&9&\infty\\\infty&\infty&5&\infty&\infty&6\\\infty&\infty&5&\infty&\infty&\infty\\3&\infty&\infty&\infty&1&\infty\end{bmatrix}$$

优点：

- 容易实现图的操作，如：求某顶点的度、判断顶点之间是否有边（弧）、找顶点的邻接点等等。

缺点：

- n 个顶点需要 n^2 个单元存储边（弧），空间效率为 $O(n^2)$。对稀疏图而言尤其浪费空间。

二. 邻接表（链式）表示法

1. 基本思想

- 用链表的结点来表示边
- 对每个顶点 v_i 建立一个单链表，把与 v_i 有关联的边的信息（即度或出度边）链接起来，表中每个结点都设为 3 个域：
  - **data**：数据域，存储顶点 v_i 信息；
  - **firstarc**：链域，指向单链表的第一个结点；
  - **adjvex**：邻接点域，表示 v_i 邻接点的位置；
  - **nextarc**：链域，指向下一个边/弧；
  - **info**：数据域，与边有关的信息（如权值）。
- 每个单链表还应当附设一个头结点（设为 2 个域），存 v_i 信息；
- 每个单链表的头结点另外用顺序存储结构存储。

2. 分析

- 对于 n 个顶点、e 条边的无向图，邻接表中除了 n 个头结点外，只有 $2e$ 个表结点，空间效率为 $O(n+2e)$；
- 若是稀疏图（$e \ll n^2$），则比邻接矩阵表示法 $O(n^2)$ 省空间。

计算方法：

- **无向图顶点的度**：

$$TD(V_i) = \text{单链表中链接的结点个数}$$

- **有向图顶点的出度**：

$$OD(V_i) = \text{单链出边表中链接的结点数}$$

- **有向图顶点的入度**：

$$ID(V_i) = \text{邻接点为 } V_i \text{ 的弧个数}$$

- **有向图顶点 V_i 的度**：

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

（需遍历全表）

3. 定义

```c
#define MAX_VERTEX_NUM 20  // 假设的最大顶点数
typedef struct ArcNode {   // 弧结构
    int adjvex;            // 该弧所指向的顶点位置
    struct ArcNode *nextarc; // 指向下一条弧的指针
```

```
    InfoArc *info;          // 该弧相关信息的指针
} ArcNode;
typedef struct VNode {      // 顶点结构
    VertexType data;        // 顶点信息
    ArcNode *firstarc;      // 指向依附该顶点的第一条弧的指针
} VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {            // 图结构
    AdjList vertices;       // 应包含邻接表
    int vexnum, arcnum;     // 应包含顶点总数和弧总数
    int kind;               // 还应说明图的种类（用标志）
} ALGraph;
```

空间效率：

- $O(n+2e)$ 或 $O(n+e)$

时间效率：

- $O(n+e\cdot n)$

7.3 图的遍历

深度优先搜索

- 基本思想: 仿树的**先序遍历**过程

1. 深度优先搜索的步骤

- **起始点选择**：从图中的某个起始顶点 v 开始。
- **访问邻接顶点**：
  - 访问顶点 v 的任一邻接顶点 w_1。
  - 再从 w_1 出发，访问与 w_1 邻接但还未被访问过的顶点 w_2。
  - 继续从 w_2 出发，重复上述过程，直至到达所有邻接顶点都被访问过的顶点 u。
- **回溯**：
  - 如果当前顶点的所有邻接顶点都已被访问，则回退到前一次刚访问过的顶点。
  - 如果该顶点还有未访问的邻接顶点，则访问这些顶点；如果没有，则继续回溯。
- **重复过程**：
  - 重复上述步骤，直到图中所有顶点都被访问过为止。

2. 代码实现

全局变量与函数声明

```
Boolean visited[MAX]; // 记录顶点是否被访问的标志数组
Status (*VisitFunc)(int v); // 全局函数变量，用于定义访问顶点的操作
```

主函数：`DFSTraverse`

```
void DFSTraverse(Graph G, Status (*Visit)(int v)) {
    /* 对图 G 作深度优先遍历 */
    VisitFunc = Visit; // 设置访问函数
    for (v=0; v<G.vexnum; ++v) {
        visited[v] = FALSE; // 初始化访问标志数组
    }
    for (v=0; v<G.vexnum; ++v) {
        if (visited[v] == 0) // 对尚未访问的顶点调用 DFS
            DFS(G, v);
    }
}
```

递归函数：`DFS`

```
void DFS(Graph G, int v) {
    /* 从顶点 v 出发，深度优先搜索遍历连通图 G */
    visited[v] = TRUE; // 标记顶点 v 为已访问
    VisitFunc(v); // 访问顶点 v
    for (w=FirstAdjVex(G,v); w!=0; w=NextAdjVex(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
        }
    }
}
```

5. 关键点总结

- **递归机制**：DFS 使用递归实现，每次访问一个顶点后，递归地访问其未访问的邻接顶点，直到无法继续为止。
- **处理非连通图**：在主函数 `DFSTraverse` 中，通过遍历所有顶点并调用 `DFS`，确保即使***图不连通***，也能访问到所有的连通分量。

6. 链表实现

```
void dfs (g, v) {
    visit(v);
    visited[v] = 1;
    p = g[v]->next_adj;
    while(p != NULL){
        w = p->node_index;
        if(visited[w] == 0)
            dfs(g, w);
        p = p->next_adj;
    }
}
```

7. 时间复杂度分析

(设图中有 n 个顶点，e 条边)

- 如果用邻接矩阵来表示图，遍历图中每一个顶点都要从头扫描该顶点所在行，因此遍历全部顶点所需的时间为 $O(n^2)$。
- 如果用邻接表来表示图，遍历图的时间复杂度为 $O(n+e)$。

结论：

- 稠密图适于在邻接矩阵上进行深度遍历；
- 稀疏图适于在邻接表上进行深度遍历。

广度优先搜索(BFS, Breadth_First Search)

- 基本思想: 仿树的**层次遍历**过程
- 代码实现

```
void BFSTraverse(Graph G, Status (*Visit)(int v)) {
    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;                          // 初始化辅助数组
    InitQueue(Q);                                    // 置空辅助队列Q
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v]) {                           // v 尚未访问
            visited[v] = TRUE;
            visit(v);
            EnQueue(Q, v);                           // 入队列
            while (!QueueEmpty(Q)) {                 // 队列头元素出队，记为u
                DeQueue(Q, u);
                for (w=FirstAdjVex(G,u); w!=0; w=NextAdjVex(G,u,w)) {
                    if (!visited[w]) {               // w 为 u 的尚未访问的邻接顶点
                        visited[w] = TRUE;
                        Visit(w);
                        EnQueue(Q, w);
                    }
                }
            }
        }
}//BFSTraverse
```

总结

- 对于非连通图的遍历过程中每调用一次DFS算法/循环都得到该图的一个连通分量

7.4 图的连通性

基本概念

- 一次搜索可以获得一个连通分量。
- 1. **非连通图的遍历限制**：
     当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法只能访问到该顶点所在的最大连通子图（连通分量）的所有顶点，而无法遍历到图中的所有顶点。
  2. **求解所有连通分量的方法**：
     若从无向图的每一个连通分量中的一个顶点出发进行遍历，可以求得无向图的所有连通分量。
  3. **算法中的顶点检测**：
     在算法中，需要对图的每一个顶点进行检测：
     - 如果已被访问过，则该顶点一定是落在图中已求得的连通分量上；
     - 如果还未被访问，则从该顶点出发遍历图，可以求得图的另一个连通分量。
- 对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。
- **生成树的定义**：
  - 按照生成树的定义，n 个顶点的连通网络的生成树有 n 个顶点和 $n-1$ 条边。
- **构造生成树的准则**：
  - 必须只使用该网络中的边来构造生成树；
  - 必须使用且仅使用 $n-1$ 条边来联结网络中的 n 个顶点；
  - 不能使用产生回路的边。

Prim算法

- 设置一个辅助数组，对当前 $V-U$ 集中的每个顶点，记录和顶点集 U 中顶点相连接的代价最小的边
- **Prim 算法的核心步骤**
  - **取图中任意一个顶点 v 作为生成树的根**：
    - 之后往生成树上添加新的顶点 w。
    - 在添加的顶点 w 和已经在生成树上的顶点 v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。
    - 之后继续往生成树上添加顶点，直至生成树上含有 $n-1$ 个顶点为止。

```
struct {
    VertexType adjvex;  // U 集中的顶点序号
    VRType     lowcost; // 边的权值
} closedge[MAX_VERTEX_NUM];*

void MiniSpanTree_PRIM(MGraph G, VertexType u)
{
    // 用 PRIM 算法从第 u 个顶点出发构造网 G 的最小生成树 T。
    k = LocateVex(G, u);
    // 辅助数组初始化
    for (j = 0; j < G.vexnum; ++j)
```

```
        if (j != k) // adjvex, lowcost
            closedge[j] = {u, G.arcs[k][j].adj};
    closedge[k].lowcost = 0;
    for (i = 1; i < G.vexnum; ++i) { // 选择其余 G.vexnum-1 个顶点
        k = minimum(closedge); // 求出 T 的下一个节点
        // 此时 closedge[k].lowcost =
        // MIN{closedge[v_i].lowcost | closedge[v_i].lowcost > 0, v_i ∈ V-U}.
        printf("%c -> %c, %d\n", closedge[k].adjvex, G.vexs[k], closedge[k].lowcost);
// 输出生成树的边
        closedge[k].lowcost = 0; // 第 k 顶点并入 U 集
        for (j = 0; j < G.vexnum; ++j) { // 新顶点并入 U 后重新选择最小边
            if (G.arcs[k][j].adj < closedge[j].lowcost) {
                closedge[j] = {G.vexs[k], G.arcs[k][j].adj};
            }
        }
    }// MiniSpanTree
```

Kruskal算法

- **考虑问题的出发点**：
  - 为了使生成树上边的权值之和达到最小，应使生成树中每一条边的权值尽可能小。
- **具体做法**：
  - 先构造一个只含 n 个顶点的子图 SG。
  - 从权值最小的边开始，若它的添加不使 SG 中产生回路，则在 SG 上加上这条边。
  - 如此重复，直至加上 $n-1$ 条边为止。

Prim 算法与 Kruskal 算法的对比

算法名	Prim 算法	Kruskal 算法
时间复杂度	$O(n^2)$	$O(e\log e)$
适应范围	稠密图	稀疏图

- **Prim 算法**：
  - 时间复杂度为 $O(n^2)$，适用于稠密图。
- **Kruskal 算法**：
  - 时间复杂度为 $O(e\log e)$，适用于稀疏图。

图的拓扑排序

拓扑排序的关键点

1. **没有前驱的顶点**：
    - 没有前驱的顶点 = 入度为零的顶点。
2. **删除顶点及以它为尾的弧**：
    - 删除顶点及以它为尾的弧 = 弧头顶点的入度减 1。

拓扑排序的实现

算法描述

采用邻接表作为有向图的存储结构，并在头结点中增加一个存放顶点入度的数组（`indegree`）。为避免每次都要搜索入度为零的顶点，在算法中设置一个“栈”，以保存“入度为零”的顶点。

```
Status TopologicalSort(ALGraph G) {
    // 有向图G采用邻接表存储结构。
    // 若G无回路，则输出G的顶点的一个拓扑序列并返回OK，
    // 否则ERROR。

    FindInDegree(G, indegree);  // 对各顶点求入度
    InitStack(S);               // 建零入度顶点栈S
    for (i = 0; i < G.vexnum; ++i)
        if (!indegree[i])
            Push(S, i);  // 入度为零的顶点入栈

    count = 0;                  // 对输出顶点计数
    while (!EmptyStack(S)) {
        // 输出i号顶点并计数。
        Pop(S, v);
        ++count;
        printf("%d ", G.vertices[v].data);

        for (p = G.vertices[v].Firstarc; p; p = p->Nextarc) {
            k = p->adjvex;      // 对i号顶点的每个邻接点的入度减1
            if (!(--indegree[k])) // 若入度为零，则入栈
                Push(S, k);
        }
    }

    if (count < G.vexnum)
        return ERROR;  // 该图中有回路
    else
        return OK;
}//TopologicalSort
```

算法复杂度分析

对有 (n) 个顶点和 (e) 条弧的有向图，建立求各顶点的入度的时间复杂度为 (O(e))；建零入度顶点栈的时间复杂度为 (O(n))。在拓扑排序中，若有向图无环，则每个顶点进一次栈，出一次栈，入度减 1 的操作在 `while` 语句中共执行 (e) 次。所以，总的时间复杂度为 (O(n + e))。