第 7 章　图

教学要点

图结构是一种由数据元素集合及元素间的关系集合组成的非线性数据结构。在图结构中，数据元素之间的关系没有限制，任意两个元素之间都可能有某种关系，因而每个数据元素可以有多个前驱元素和多个后继元素。图是表示离散结构的一种有力的工具，可以用来描述现实世界的众多问题。

本章介绍具有非线性关系的图结构，重点讨论图的基本概念及图的存储结构，还将介绍图结构中的常用算法，如遍历算法、图的生成树和最短路径等。

本章在 Visual Studio 中用名为 graph 的类库型项目实现有关数据结构的类型定义，用名为 graphtest 的应用程序型项目实现图结构的测试和应用演示程序。

建议本章授课 6 学时，实验 4 学时。

7.1　图的定义与基本术语

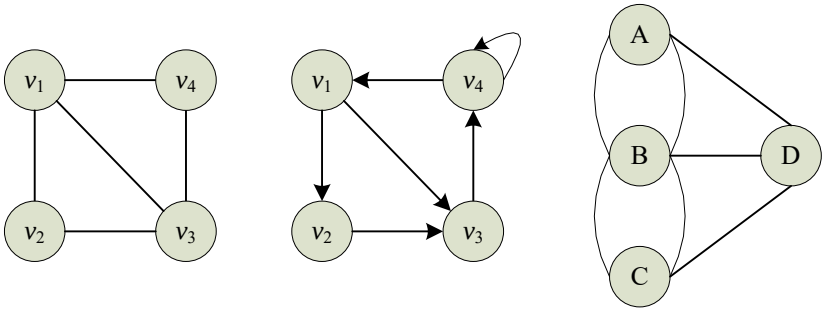
7.1.1　图的定义

图（graph）是一种非线性数据结构，它的数据元素之间的关系没有限制，任意两个元素之间都可能有某种关系。数据元素用结点（node）表示，如果两个元素相关，就用一条边（edge）将相应的结点连接起来，这两个结点称为相邻结点。这样，图就可以定义为由结点集合及结点间的关系集合组成的一种数据结构，图 7.1 显示了几个图的示例。结点又称为顶点（vertex），结点之间的关系称为边。一个图 G 记作 G = (V, E)，其中，V 是结点的有限集合，E 是边的有限集合，即有：

$$V = \{ x \mid x \in \text{某个具有相同特性的数据元素集合} \}$$

$$E = \{ e(x, y) \mid x, y \in V \}$$

在上式中，$e(x, y)$表示结点 x 和结点 y 之间的一条边，是一种无序结点对，无方向性，它表示两个结点之间的相邻关系。



(a) 无向图G_1　　(b) 有向图G_2　　(c) Koenigburg七桥图G_3

图 7.1　图结构示例

如果结点间的关系是有序结点对，可表示为 e<x, y>，它表示从结点 x 到结点 y 的一条单向通路，是有方向的。因而，图中边的集合表示为：

$$E = \{ \mathbf{e}\langle x, y\rangle \mid x, y \in V \}$$

1. 无向图与有向图

在一个图 G 中，如果任一条边 $e(x, y)$仅表示两个结点 x 和 y 之间的相邻关系，无方向性，则称边 $e(x, y)$是无向的；图 G 则被称为无向图（undirected graph）。图 7.1 中的 G_1 和 G_3 都是无向图，G_1 的结点集合 V 和边的集合 E 分别为：

$$V(G_1) = \{v_1, v_2, v_3, v_4\}$$

$$E(G_1) = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$$

一般，用圆括号将一对结点括起来组成的无序偶对表示无向边，如(A, B)和(B, A)表示同一条边。

在一个图 G 中，如果任一条边 $\mathbf{e}\langle x, y\rangle$是两个结点 x 和 y 的有序偶对，表示从结点 x 到 y 的单向通路，有方向性，则称边 $\mathbf{e}\langle x, y\rangle$是有向的。一般，用尖括号将一对结点括起来表示有向边，x 称为有向边的起点（initial node），y 称为有向边的终点（terminal node），所以$\langle x, y\rangle$和$\langle y, x\rangle$分别表示两条不同的有向边；图 G 称为有向图（directed graph，digraph）。图 7.1 中的 G_2 是有向图，它的结点集合 V 与图 G_1 相同，它的边集合 E 为：

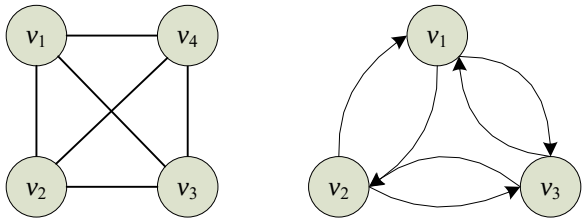
$$E(G_2) = \{\langle v_1, v_2\rangle, \langle v_1, v_3\rangle, \langle v_2, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_4\rangle, \langle v_4, v_1\rangle\}$$

有向图中的边又称为弧（arc）。在图的图形表示中，用箭头表示有向边的方向，箭头从起点指向终点。当存在边的起点和终点是同一个结点的情况，即存在边 $e(v, v)$或 $\mathbf{e}\langle v, v\rangle$时，称该边为环（loop）。例如，$G_2$ 中存在环 $\mathbf{e}\langle v_4, v_4\rangle$。

无环且无重边的无向图称为简单图，如图 7.1 中的 G_1，本章一般讨论的是简单图。

2. 完全图、稀疏图和稠密图

一个有 n 个结点的无向图，可能的最大边数为 $n \times (n - 1) / 2$；而一个有 n 个结点的有向图，其弧的最大数目为 $n \times (n - 1)$。如果一个图的边数达到相同结点集合构成的所有图的边数最大值，则称该图为完全图（complete graph），如图图 7.2 所示。n 个结点的完全图通常记为 K_n。



(a) 无向完全图G_4　　(b) 有向完全图G_5

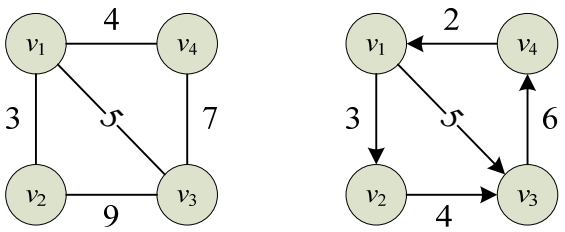
图 7.2 完全图

一个有 n 个结点的图，其边的数目如果远小于 n^2，则称为稀疏图（sparse graph）。一个图的边数如果接近最大数目，则称为稠密图（dense graph）。

3. 带权图

在图中除了用边表示两个结点之间的相邻关系外，有时还需表示它们相关的强度信息，例如从一个结点到另一个结点的距离、花费的代价、所需的时间等，诸如此类的信息可以通过在图的每条

边上加一个称作权（weight）的数来表示。边上加有权值的图称为带权图（weighted graph）或网络（network）。图 7.3 显示了两个带权图，权值标在相应的边上。



(a) 带权的无向图G_6　　(b) 带权的有向图G_7

图 7.3 带权图

7.1.2 结点与边的关系

1. 相邻结点

若 $e(v_i, v_j)$是无向图中的一条边，则称结点 v_i 和 v_j 是相邻结点，边 $e(v_i, v_j)$与结点 v_i 和 v_j 相关联。若 **e**$<v_i, v_j>$是有向图中的一条边，则称结点 v_i 邻接到结点 v_j，结点 v_j 邻接于结点 v_i，边 **e**$< v_i, v_j>$与结点 v_i 和 v_j 相关联。

2. 度、入度、出度

图中与结点 v 相关联的边的数目称为该结点的度（degree），记作 TD(v)。度为 1 的结点称为悬挂点（pendant node）。在图 7.1 的图 G_1 中，结点 v_1 和 v_3 的度都是 3，结点 v_2 和 v_4 的度都是 2。

在有向图中，以结点 v 为终点的有向边的数目称为该结点的入度（in-degree），记作 ID(v)；以结点 v 为起点的有向边的数目称为该结点的出度（out-degree），记作 OD(v)。出度为 0 的结点称为终端结点（或叶子结点）。结点 v 的度是该结点的入度与出度之和，即有

$$\mathrm{TD}(v) = \mathrm{ID}(v) + \mathrm{OD}(v)$$

在图 7.1 的图 G_2 中，结点 v_3 的入度 ID(v_3)=2，出度 OD(v_3)=1，度 TD(v_3)=3。

3. 度与边数的关系

如果有 n 个结点的无向图 G，其结点集合为$\{v_1, v_2, \cdots, v_n\}$，其边数为 e，则

$$e = \frac{1}{2}\sum_{i=1}^{n}\mathrm{TD}(v_i)$$

当 G 为有向图时，它的度与边数的关系可写为

$$\sum_{i=1}^{n}\mathrm{ID}(v_i) = \sum_{i=1}^{n}\mathrm{OD}(v_i) = e$$

$$\sum_{i=1}^{n}\mathrm{TD}(v_i) = \sum_{i=1}^{n}\mathrm{ID}(v_i) + \sum_{i=1}^{n}\mathrm{OD}(v_i) = 2e$$

7.1.3 子图与生成子图

设图 G = (V, E)，G′ = (V′, E′)，若 V′⊆V，E′⊆E，并且 E′中的边所关联的结点都在 V′中，则称图 G′是 G 的子图（subgraph）。任一图 G = (V, E)都是它自己的子图，如果 G 的子图 G′≠G，则称 G′是 G 的真子图。

如果 G′是 G 的子图，且 V′ = V，则称图 G′是 G 的生成子图（spanning subgraph）。

7.1.4 路径、回路及连通性

1. 路径与回路

在图 G = (V, E) 中，若从结点 v_i 出发，经过边 e(v_i, v_{p1})到达结点 v_{p1}，继续经过边 e(v_{p1}，v_{p2})到达结点 v_{p2}，……，最后经过边 e(v_{pm}, v_j)到达结点 v_j，也就是从结点 v_i 出发依次沿边 e(v_i, v_{p1}), e(v_{p1}，v_{p2}), …, e(v_{pm}, v_j)经过结点序列 v_{p1}, v_{p2}, …, v_{pm} 到达结点 v_j，则称边序列{e(v_i, v_{p1}), e(v_{p1}，v_{p2}), …, e(v_{pm}, v_j)}是从结点 v_i 到结点 v_j 的一条路径（path），通常缩写成结点序列(v_i, v_{p1}, v_{p2}, …, v_{pm}, v_j)。

有向图 G 中的路径也是有向的，如果 **e**<*vi*, v_{p1}>, **e**<v_{p1}, v_{p2}>，…，**e**<v_{pm}, v_j>都是有向图 G 中的边，则结点 v_i 和结点 v_j 之间存在路径，v_i 为该路径的起点，v_j 为终点。

一条路径的长度（path length）就是该条路径上边的数目。在带权图中，路径的长度有时指的是加权路径长度，它定义为从起点到终点的路径上各条边的权值之和。例如，图 7.3(a)中从结点 v_1 到 v_3 的一条路径（v_1, v_2, v_3）的加权路径长度为 3+9=12。

如果在一条路径中，除起点和终点外，其他结点都不相同，则此路径称为简单路径（simple path）。起点和终点相同且长度大于 1 的简单路径成为回路（cycle）。例如，图 7.1 G_3 中的路径（B, C, D, B）是一条回路。

2. 图的连通性

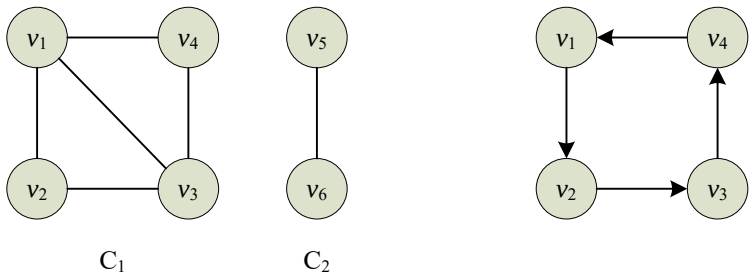
如果无向图 G 中的两个结点 v_i 和 v_j 之间有一条路径，则称结点 v_i 和结点 v_j 是连通的（connected）。如果图 G 中任意两个不同的结点之间都是连通的，则称图 G 为连通图（connected graph）。

非连通图中可能若干对结点之间不是连通的，它的极大连通子图称为该图的连通分量（connected component）。图 7. 1 中的 G_1 是连通图，图 7.4(a)则不是连通图，它有两个连通分量，分别是 C_1 和 C_2。

如果有向图 G 中的某两个不同的结点 v_i 和 v_j 之间有一条从 v_i 到 v_j 的路径，同时还有一条从 v_j 到 v_i 的路径，则称这两个结点是强连通的（strongly connected）。如果有向图 G 中任意两个不同的结点之间都是强连通的，则该有向图是强连通的。图 7.4(b)中的 G_8 是强连通的有向图。

有向图的强连通分量指的是该图的强连通的最大子图。

一个有向图 G 中，若存在一个结点 v_0，从 v_0 有路径可以到达图 G 中其他所有结点，则称结点 v_0 为图 G 的根，称此有向图为有根的图。



(a) 无向图的两个连通分量C_1和C_2　　(b) 强连通有向图G_8

图 7.4 图的连通性

7.1.5 图的基本操作

图结构的基本操作有以下几种：

- Initialize：初始化。建立一个图实例并初始化它的结点集合和边的集合。
- AddNode /AddNodes：在图中设置、添加一个或若干结点。
- Get/Set：访问。获取或设置图中的指定结点。
- Count：求图的结点个数。
- AddEdge/ AddEdges：在图中设置、添加一条或若干条边，即设置、添加结点之间的关联。
- Nodes/Edges：获取结点表或边表。
- Remove：删除。从图中删除一个数据结点及相关联的边。
- Contains/IndexOf：查找。在图中查找满足某种条件的结点（数据元素）。
- Traversal: 遍历。按某种次序访问图中的所有结点，并且每个结点恰好访问一次。
- Copy：复制。复制一个图。

7.2 图的存储结构

图结构是结点和边的集合，图的存储结构要记录这两方面的信息。结点的集合可以用一个称为结点表的线性表来表示；图中的一条边表示某两个结点的邻接关系，图的边集可以用邻接矩阵（adjacency matrix）或邻接表（adjacency list）来表示。邻接矩阵是一种顺序存储结构，而邻接表是一种链式存储结构。

7.2.1 图结构的邻接矩阵表示法

1. 邻接矩阵的定义

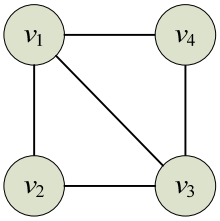
图结构的邻接矩阵用来表示图的边集，即图中结点间的相邻关系集合。设 G=(V, E)是一个具有 n 个结点的图，它的邻接矩阵是一个 n 阶方阵，其中的元素具有下列性质：

$$a_{ij}=\begin{cases}1 & \text{若e}(v_i,v_j)\in E\text{或}\mathbf{e}<v_i,v_j>\in E\\ 0 & \text{若e}(v_i,v_j)\notin E\text{或}\mathbf{e}<v_i,v_j>\notin E\end{cases}$$

邻接矩阵任意元素 $a_{i,j}$ 的值表示两个结点 v_i 和 v_j 之间是否有相邻关系，即这两个结点间是否存在

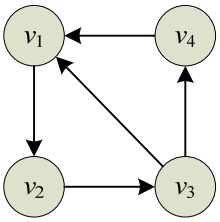
边：如果 $a_{i,j}$=1，则 v_i 和 v_j 之间存在一条边；如果 $a_{i,j}$=0，则 v_i 和 v_j 之间无边相连。邻接矩阵作为其全部元素的整体则表示图的边集。

例如，图 7.5 显示了一个无向图及其对应的邻接矩阵，图 7.6 则显示了一个有向图及其对应的邻接矩阵。



$$\mathbf{A}_1=\begin{bmatrix}0&1&1&1\\1&0&1&0\\1&1&0&1\\1&0&1&0\end{bmatrix}$$

图 7.5 无向图及其邻接矩阵



$$\mathbf{A}_8=\begin{bmatrix}0&1&0&0\\0&0&1&0\\1&0&0&1\\1&0&0&0\end{bmatrix}$$

图 7.6 有向图及其邻接矩阵

从上面的例子中可以看出，无向图的邻接矩阵是对称矩阵，即 $a_{ij}=a_{ji}$，有向图的邻接矩阵则不一定对称。一般地，用邻接矩阵表示一个具有 n 个结点的图结构需要 n^2 个存储单元。对于无向图，则因为其邻接矩阵是对称的，可以只存储邻接矩阵的上三角或下三角数据元素，因而只需 $n^2/2$ 个存储单元。图结构的邻接矩阵表示法的空间复杂度为 $O(n^2)$。

2. 带权图的邻接矩阵

对于带权图，设某边 $e(v_i, v_j)$或 $\mathbf{e}<v_i, v_j>$上的权值为 w_{ij}，则带权图的邻接矩阵定义为：

$$a_{ij}=\begin{cases}w_{ij} & 若v_i\neq v_j且e(v_i,v_j)\in E或\mathbf{e}<v_i,v_j>\in E\\ \infty & 若v_i\neq v_j且e(v_i,v_j)\notin E或\mathbf{e}<v_i,v_j>\notin E\\ 0 & 若v_i=v_j\end{cases}$$

图 7.3 中的两个带权图的邻接矩阵分别为 A_6 和 A_7：

$$\mathbf{A}_6=\begin{bmatrix}0&3&5&4\\3&0&9&\infty\\5&9&0&7\\4&\infty&7&0\end{bmatrix}\quad \mathbf{A}_7=\begin{bmatrix}0&3&5&\infty\\\infty&0&4&\infty\\\infty&\infty&0&6\\2&\infty&\infty&0\end{bmatrix}$$

3. 邻接矩阵与结点的度

根据邻接矩阵容易求得各个结点的度。无向图中某结点 v_i 的度等于邻接矩阵第 i 行上各元素之和，即有：

$$\mathrm{TD}(v_i)=\sum_{j=1}^{n}a_{ij}$$

有向图中的某结点 v_i 的出度等于矩阵第 i 行上各元素之和，结点 v_j 的入度等于第 j 列上各元素

之和，即有：

$$\mathrm{OD}(v_i)=\sum_{j=1}^{n}a_{ij},\ \mathrm{ID}(v_j)=\sum_{i=1}^{n}a_{ij}$$

4. 图的顶点类和邻接矩阵图类的定义

下面定义 Vertex 类表示图中的顶点，成员 data 存储顶点的数据，成员 visited 作为顶点是否被访问过的标志，以后在图的遍历操作中将会用到。

```
public class Vertex<T> {
    private T data;
    private bool visited;
    public Vertex() {
        data = default(T); visited = false;
    }
    public Vertex(T data, bool visited) {
        this.data = data;
        this.visited = visited;
    }
    public Vertex(T data) {
        this.data = data;
        this.visited = false;
    }

    public T Data {
        get { return data; }
        set { data = value; }
    }
    public bool Visited {
        get { return visited; }
        set { visited = value; }
    }

    public void Show() {
        Console.Write("-" + this.data + " ->");
    }
    public override string ToString() {
        return string.Format("-{0}->", this.data);
    }
}
```

下面定义 AdjacencyMatrixGraph 类表示一个以邻接矩阵存储、具有 n 个结点的图。成员变量 count 表示图的结点个数，成员变量 vertexList 是一个线性表（称作结点表），保存图的结点集合。成员变量 AdjMat 是一个二维数组，用来存储图的邻接矩阵。

```
public class AdjacencyMatrixGraph<T> {
    private int count = 0;                          //图的结点个数
    private IList<Vertex<T>> vertexList;            //图的结点表
    private int[,] AdjMat;                          //二维数组存储图的邻接矩阵
    ......
}
```

上面声明的两个类都是泛型类，顶点的数据类型在定义图和顶点类型的实例时决定。Vertex 类和 AdjacencyMatrixGraph 类都声明在名字空间 DSAGL 中。

5. 邻接矩阵图的基本操作

1) 邻接矩阵图的初始化

使用带二维数组参数的构造方法创建图对象，存储指定的邻接矩阵，并设置一个空的结点表；缺省的构造方法则分别构建一个空的邻接矩阵和一个空的结点表。算法如下：

```
public AdjacencyMatrixGraph(int[,] adjmat){
    int n = adjmat.GetLength(0);
    AdjMat = new int[n,n];
    Array.Copy(adjmat, AdjMat, n*n);
    vertexList = new List<Vertex<T>>();
    count = n;
}
public AdjacencyMatrixGraph(){
    AdjMat = new int[MaxVertexCount, MaxVertexCount];
    vertexList = new List<Vertex<T>>();
    count = 0;
}
```

2) 返回或设置图的结点数

该操作告知或设置图的结点个数，以名为 Count 的属性来实现这个功能，编码如下：

```
public int Count {
    get { return count; }
    set { count = value; }
}
```

3) 获取或设置指定结点的值

就像 C#的数组下标从 0 开始一样，我们用从 0 开始的索引参数 *i* 来指示图的第 *i* 个结点，以类的索引器的形式实现这个功能，编码如下：

```
public T this[int i] {
    get {
        if (i >= 0 && i < count)
            return vertexList[i].Data;
        else
            throw new IndexOutOfRangeException(
                "Index Out Of Range Exception in " + this.GetType());
```

```
        }
        set {
            if (i >= 0 && i < count)
                vertexList[i].Data = value ;
            else
                throw new IndexOutOfRangeException(
                    "Index Out Of Range Exception in " + this.GetType());
        }
    }
```

4) 为图设置一组结点

该操作将图的结点表 vertexList 设为参数 nodes 指定的表，编码如下：

```
public void AddNodes(IList<Vertex<T>> nodes) {
    vertexList = nodes;
    count = vertexList.Count;
}
```

5) 查找具有特定值的元素

在图中查找具有特定值 k 的元素的过程为：在图中按结点号顺序检查结点值是否等于 k，若相等则返回结点号；否则继续与下一个结点进行比较，当比较了所有的数据元素后仍未找到，则返回 -1，表示查找不成功。算法实现如下：

```
public int IndexOf(T k){
    int j = 0;
    while (j < count && !k.Equals(vertexList[j].Data) )
        j++;
    if (j >= 0 && j < count)
        return j;
    else return -1;
}
```

7.2.2　图结构的邻接表表示法

1. 用邻接表表示无向图

邻接矩阵表示图的空间复杂度为 $O(n^2)$，它与图中结点的个数有关，而与边的数目无关。对于稀疏图，其边数可能远小于 n^2，图的邻接矩阵中就会有很多零元素，这种存储方式将造成存储空间上的浪费。对于这种情况，可以用结点表和邻接表来表示和存储图结构，其占用的存储空间既与图的结点数有关，也与边数有关。对于 n 个结点的图，如果边数 $m<<n^2$，则需占用的存储空间较为节省。另外，邻接表保存了与一个结点相邻接的所有结点，这也给图的操作提供了方便。

图的结点表用来保存图中的所有结点，它通常是一个顺序存储的线性表，该线性表中的每个元素对应于图的一个结点。线性表元素的类型是一种重新定义的图结点类型（GraphNode 类），它包括两个基本成员：data 和 neighbors。data 表示结点数据元素信息，neighbors 则指向该结点的**邻接结点表**，简称**邻接表**。

图中的每个结点都有一个邻接表 neighbors，它保存与该结点相邻接的若干个结点，因此邻接表中的每个结点对应于与该结点相关联的一条边。图 7.7 显示了一个无向图及其邻接表。

与无向图邻接矩阵将每条边的信息对称地存储两次的情形类似，用邻接表表示无向图，也会将每条边的信息存储两次，每条边分别存储在与该边相关联的两个结点的邻接表中，因此对于 n 个结点 m 条边的无向图，共需要占用 $n+2m$ 个结点单元来存储图结构，即图所占用的存储空间大小既与图的结点数有关，也与图的边数有关。

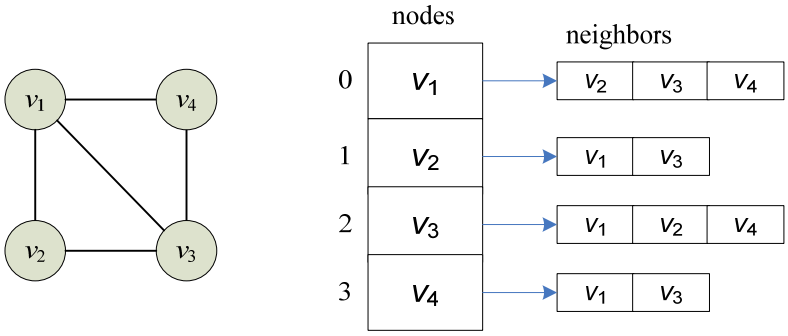


图 7.7 无向图的邻接表

2. 用邻接表表示有向图

对于有向图，一个结点的邻接表可以只存储出边相关联的邻接结点，因此，n 个结点 m 条边的有向图的邻接表需要占用 $n+m$ 个结点存储单元。图 7.8 显示了一个有向图及其出边邻接表。

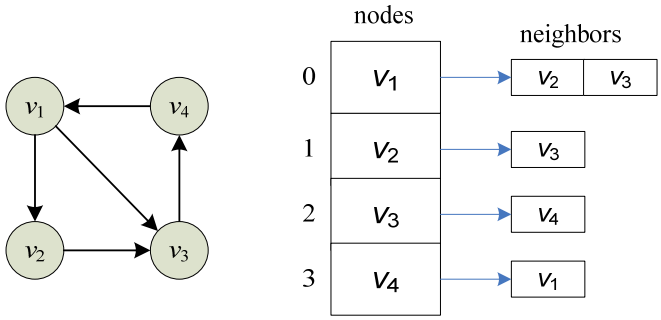


图 7.8 有向图的邻接表

3. 定义图的结点类和邻接表图类

下面定义 GraphNode 类来表示图中的结点，类的成员变量 data 存储结点的数据，成员变量 neighbors 存储结点的邻接表，成员变量 visited 作为结点是否被访问过的标志，成员 costs 留着用以存储边的权值。

```
public class GraphNode<T > {
    private T data;
    private bool visited;
    private List<GraphNode<T>> neighbors = null;
    private List<int> costs;                              // 边的权值
```

```
        public GraphNode(T data, List<GraphNode<T>> neighbors) {
            this.data = data; this.visited = false; this.neighbors = neighbors;
        }
        public GraphNode(T data) : this(data, null) { }

        public T Data { get{return data;} set {data = value;} }

        public bool Visited { get { return visited; } set { visited = value; } }

        public List<GraphNode<T>> Neighbors {
            get {
                if (neighbors == null)
                    neighbors = new List<GraphNode<T>>();
                return neighbors;
            }
            set { neighbors = value; }
        }

        public List<int> Costs {
            get {
                if (costs == null)
                    costs = new List<int>();
                return costs;
            }
            set { costs = value; }
        }
    }
```

下面定义 Graph 类来表示一个以邻接表存储的图，其中成员变量 nodes 表示图的结点表，结点表中每个元素对应于图的一个结点，它的类型为 GraphNode，结点的 neighbors 成员保存了结点的邻接表。

```
    public class Graph<T> {
        private const int Infinity = Int16.MaxValue;
        private IList<GraphNode<T>> nodes;              // 图的结点表
        ……
    }
```

GraphNode 类和 Graph 类都声明在名字空间 DSAGL 中。它们也都设计为泛型类，结点的数据类型在定义图和结点类型的实例时决定。

4. 邻接表图的基本操作

1) 邻接表图的初始化

使用构造方法创建图对象，存储指定的结点表，并根据给定的邻接矩阵建立邻接表。算法如下：

```
    public Graph(IList<GraphNode<T>> nodes, int[,] mat) {    //以邻接矩阵建立图的邻接表
```

```
        this.nodes = nodes;
        int i, j;
        int nOfNodes = mat.GetLength(0);
        for (i = 0; i < nOfNodes; i++) {
            for (j = 0; j < nOfNodes; j++)                    //查找与i相邻的其他结点j
                if (mat[i, j]!=0 && mat[i, j]!=Infinity) {
                    nodes[i].Neighbors.Add(nodes[j]);         //邻接表中添加结点
                }
        }
    }
```

Graph 类的构造方法将邻接矩阵 mat 中表示的边转换成各结点 nodes[i]的邻接表 Neighbors。

2) 返回图的结点数

该操作告知图的结点个数，以名为 Count 的属性来实现这个功能，编码如下：

```
    public int Count {
        get { return nodes.Count; }
    }
```

3) 获取或设置指定结点的值

用从 0 开始的索引参数 *i* 来指示图的第 *i* 个结点，以类的索引器的形式实现这个功能，编码如下：

```
    public T this[int i] {
        get {
            if (i >= 0 && i < Count)
                return nodes[i].Data;
            else
                throw new IndexOutOfRangeException(
                    "Index Out Of Range Exception in " + this.GetType());
        }
        set {
            if (i >= 0 && i < Count)
                nodes[i].Data = value;
            else
                throw new IndexOutOfRangeException(
                    "Index Out Of Range Exception in " + this.GetType());
        }
    }
```

4) 在图中增加结点

将参数指定的新结点添加进图的结点表，编码如下：

```
    // adds a node to the graph
    public void AddNode(T value) {
        nodes.Add( new GraphNode<T>(value) );
```

```
    }
    public void AddNode(GraphNode<T> node) {
        nodes.Add(node);
    }
```

5) 查找具有特定值的元素

在图的结点表中查找具有特定值的结点，如果找到满足条件的结点，就返回该结点或指示结点的序号；如果图中没有满足条件的结点，则返回 null 或返回-1，编码如下：

```
    public GraphNode<T> FindByValue(T k) {
        foreach (GraphNode<T> node in nodes)
            if (node.Data.Equals(k))
                return node;
        return null;
    }

    public int IndexOf(T k) {
        int j = 0;
        while (j < Count && !k.Equals(nodes[j].Data))
            j++;
        if (j >= 0 && j < Count)
            return j;
        else return -1;
    }
    public int IndexOf(GraphNode<T> k) {
        int j = 0;
        while (j < Count && !k.Data.Equals(nodes[j].Data)) j++;
        if (j >= 0 && j < Count) return j;
        else return -1;
    }
```

6) 在图中增加边，即增加结点之间的关联

增加一条有向边的算法如下：

```
    public void AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost) {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
    }
    public void AddDirectedEdge(T from, T to, int cost) {
        GraphNode<T> fromNode = FindByValue(from);
        fromNode.Neighbors.Add(FindByValue(to));
        fromNode.Costs.Add(cost);
    }
```

增加一条无向边的算法如下：

```
public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost) {
    from.Neighbors.Add(to);
    from.Costs.Add(cost);
    to.Neighbors.Add(from);
    to.Costs.Add(cost);
}
public void AddUndirectedEdge(T from, T to, int cost) {
    GraphNode<T> fromNode = FindByValue(from);
    GraphNode<T> toNode = FindByValue(to);
    fromNode.Neighbors.Add(toNode);
    fromNode.Costs.Add(cost);
    toNode.Neighbors.Add(fromNode);
    toNode.Costs.Add(cost);
}
```

7) 输出图的邻接表。ShowAdjacencyList 方法输出各结点的邻接表 Neighbors 中的各个结点数据元素值。

```
public void ShowAdjacencyList() {
    Console.WriteLine("邻接表:");
    for (int i = 0; i < nodes.Count; i++) {
        Console.Write(nodes[i].Data + " -> ");
        for (int j = 0; j < nodes[i].Neighbors.Count; j++) {
            Console.Write(nodes[i].Neighbors[j].Data + " + ");
        }
        Console.WriteLine(".");
    }
}
public void Show() {
    ShowAdjacencyList();
}
```

7.3 图的遍历

图的遍历（traversal）操作指的是，从图的一个结点出发，以某种次序访问图中的每个结点，并且每个结点仅被访问一次。与遍历操作在树结构中的作用类似，遍历也是图的一种基本操作，图的许多其他操作都可以建立在遍历操作的基础之上。

对于图的遍历，存在两种基本策略：深度优先搜索（depth first search）遍历和广度优先搜索（breadth first search）遍历。图的遍历可以从任意结点开始，从图的某一指定结点出发，图的深度优先搜索遍历类似于二叉树的先根遍历，优先从一条路径向更远处访问图的其他结点，逐渐向所有路径扩展；图的广度优先搜索遍历类似于二叉树的层次遍历，优先考虑直接近邻的结点，逐渐向远处扩展。

7.3.1　基于深度优先策略的遍历

1. 基于深度优先策略的遍历算法描述

图的深度优先搜索遍历可以看成是二叉树先根遍历的推广，就是优先从一条路径向更远处访问图的所有结点。图中的一个结点可能与多个结点相邻接，在图的遍历中，访问了一个结点后，可能会沿着某条路径又回到该结点。为了避免同一个结点重复多次被访问，在遍历过程中必须对访问过的结点作标记。前面在图的结点结构（GraphNode 类和 Vertex 类）的定义中，添加数据成员 visited 就是用来记录结点是否被访问过。

深度优先遍历操作的递归算法描述如下：

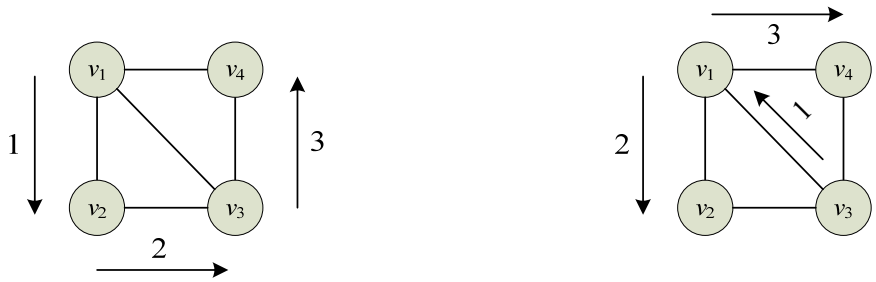
1)　从图中选定的一个结点（设该结点下标为 *m*，结点值为 *s*）出发，访问该结点。

2)　查找与结点 *s* 相邻接且未被访问过的另一个结点（设该结点下标为 *n*，结点值为 *t*）。

3)　若存在这样的结点 *t*，则从结点 *t* 出发继续进行深度优先搜索遍历。

4)　若找不到结点 *t*，说明从 *s* 开始能够到达的所有结点都已被访问过，此条路径遍历结束。

图 7.9 显示从不同的结点进行深度优先搜索遍历。



(a) 从顶点v_1出发的一种深度优先遍历序列$\{v_1, v_2, v_3, v_4\}$　(b) 从顶点v_3出发的一种深度优先遍历序列$\{v_3, v_1, v_2, v_4\}$

图 7.9 无向图的深度优先搜索遍历

按照上述算法，对一个连通的无向图或一个强连通的有向图，从某一个结点出发，一次深度优先搜索遍历可以访问图的每个结点；否则，一次深度优先搜索遍历只能访问图中的一个连通分量。

2. 邻接矩阵图的深度优先遍历操作的算法实现

在以邻接矩阵存储的图 AdjacencyMatrixGraph 类中，增加成员方法 DepthFirstSearch 和 DepthFirstShow 实现图的深度优先遍历算法。

图的结点表 vertexList 中的每个元素对应图的一个结点，结点类型为 Vertex，它记录结点的值及是否被访问过等信息。在一个含有 *n* 个结点的图中进行深度优先遍历，一旦访问一个结点，则该结点被标志为 “已被访问”（其域 Visited 被置为 true），此后便不再访问该结点。

增加成员后的 AdjacencyMatrixGraph 类如下：

```
public class AdjacencyMatrixGraph<T> {
    private const int MaxVertexCount = 10;
    private const int Infinity = Int16.MaxValue;
    private int count = 0;                              //图的结点个数
    private IList<Vertex<T>> vertexList;                //图的结点表
    private int[,] AdjMat;                              //二维数组存储图的邻接矩阵
```

```
        public void ResetVisitFlag() {   //设置未访问标记
            int i;
            for (i = 0; i < count; i++)
                vertexList[i].Visited = false;
        }

        //从结点 m : [0 - count-1] 开始的深度优先遍历，结果放在表或数组sql中
        public void DepthFirstSearch(int m, IList<T> sql){
            int n = 0;
            T k = vertexList[m].Data;
            sql.Add(k);
            vertexList[m].Visited = true;
            while (n < count) {                    //查找与m相邻的且未被访问的其他结点
                if (AdjMat[m, n] != 0 && AdjMat[m, n] != Infinity && !vertexList[n].Visited)
                    DepthFirstSearch(n, sql);     //递归，继续深度优先遍历
                else
                    n++;
            }
        }

        //从结点m :[0 - count-1] 开始的深度优先遍历，结果显示在控制台
        public void DepthFirstShow(int m) {
            vertexList[m].Show();
            vertexList[m].Visited = true;
            int n = 0;
            while (n < count) {                    //查找与m相邻的且未被访问的其他结点
                if (AdjMat[m, n] != 0 && AdjMat[m, n] != Infinity && !vertexList[n].Visited)
                    DepthFirstShow(n);             //递归，继续深度优先遍历
                else
                    n++;
            }
        }
    }
```

从上面的代码可以看出，对于有 n 个结点的图，它的邻接矩阵需要 n^2 个存储单元，处理一行的时间复杂度为 $O(n)$，矩阵共有 n 行，故深度优先遍历算法的时间复杂度为 $O(n^2)$。

【例7.1】　邻接矩阵图的深度优先遍历算法测试。

```
using System;
using System.Collections.Generic;
using DSAGL;
namespace graphtest {
    class AdjacencyMatrixGraphTest {
        static void Main(string[] args) {
```

```
            int[,] adjmat = { { 0, 1, 1, 1 }, { 1, 0, 1, 0 }, { 1, 1, 0, 1 }, { 1, 0, 1, 0 } };
            Vertex<string>[] nodes = new Vertex<string>[4];
            for (int i = 0; i < 4; i++)
                nodes[i] = new Vertex<string>("Vertex" + (i+1) );
            AdjacencyMatrixGraph<string> g = new AdjacencyMatrixGraph<string>(adjmat);
            g.AddNodes(nodes);
            DepthFirstShowTest(g);
        }

        static void DepthFirstShowTest(AdjacencyMatrixGraph<string> g) {
            Console.WriteLine("深度优先遍历:");
            for (int i = 0; i < g.Count; i++) {
                g.DepthFirstShow(i);
                Console.WriteLine();
                g.ResetVisitFlag();
            }
        }
    } }
```

程序运行结果如下：

```
深度优先遍历:
-Vertex1 ->-Vertex2 ->-Vertex3 ->-Vertex4 ->
-Vertex2 ->-Vertex1 ->-Vertex3 ->-Vertex4 ->
-Vertex3 ->-Vertex1 ->-Vertex2 ->-Vertex4 ->
-Vertex4 ->-Vertex1 ->-Vertex2 ->-Vertex3 ->
```

从结点 Vertex1 和 Vertex3 出发得到的深度优先搜索遍历序列分别是< Vertex1，Vertex2，Vertex3，Vertex4>和< Vertex3，Vertex1，Vertex2，Vertex4>，其过程显示在图 7.9 中。

3. 邻接表图的深度优先遍历算法实现

在表示以邻接表存储的图结构 Graph 类中用 DepthFirstShow 成员方法来实现图的深度优先遍历算法。修改后的 Graph 类如下：

```
public class Graph<T> {
    private const int Infinity = Int16.MaxValue;
    private IList<GraphNode<T>> nodes;                    // 图的结点表

   //设置未访问标记
    public void ResetVisitFlag() {
        for (int i = 0; i < Count; i++)
            nodes[i].Visited = false;
    }

     //图的深度优先遍历，从结点号m开始，结果显示在控制台
    public void DepthFirstShow(int m) {
```

```
        int i, j;
        Console.Write("-" + nodes[m].Data + " ->");
        nodes[m].Visited = true;
        for (j = 0; j < nodes[m].Neighbors.Count; j++) {
            if (!nodes[m].Neighbors[j].Visited) {
                i = IndexOf(nodes[m].Neighbors[j]);
                DepthFirstShow(i);                          //递归访问邻接结点
            }
        }
    }

    //从结点k(值)开始的深度优先遍历，结果显示在控制台
    public void DepthFirstShow(T k) {
        int i = IndexOf(k);                                 // 结点k的下标，从0开始
        DepthFirstShow(i);
    }
}
```

从上面的代码可以看出，对于有 n 个结点和 e 条边的图，对于邻接表图结构进行深度优先遍历的时间复杂度为 $O(n+e)$。

【例7.2】　邻接表图的深度优先遍历算法测试。

```
using System; using System.Collections.Generic;
using DSAGL;
namespace graphtest {
    class GraphTraversalTest {
        static void Main(string[] args) {
            int[,] adjmat = { { 0, 1, 1, 1 }, { 1, 0, 1, 0 }, { 1, 1, 0, 1 }, { 1, 0, 1, 0 } };
            GraphNode<string>[] nodes = new GraphNode<string>[4];
            for (int i = 0; i < 4; i++)
                nodes[i] = new GraphNode<string>( "Vertex" + (i + 1) );
            Graph<string> g = new Graph<string>(nodes, adjmat);
            g.ShowAdjacencyList();
            DepthFirstShowTest(g);
        }

        static void DepthFirstShowTest(Graph<string> g) {
            Console.WriteLine("深度优先遍历:");
            for (int i = 0; i < g.Count; i++) {
                g.DepthFirstShow(i);
                Console.WriteLine();
                g.ResetVisitFlag();
            }
        }
```

```
    }
}
```

程序运行结果如下：

```
邻接表:
Vertex1 -> Vertex2 + Vertex3 + Vertex4 + .
Vertex2 -> Vertex1 + Vertex3 + .
Vertex3 -> Vertex1 + Vertex2 + Vertex4 + .
Vertex4 -> Vertex1 + Vertex3 + .
深度优先遍历:
-Vertex1 ->-Vertex2 ->-Vertex3 ->-Vertex4 ->
-Vertex2 ->-Vertex1 ->-Vertex3 ->-Vertex4 ->
-Vertex3 ->-Vertex1 ->-Vertex2 ->-Vertex4 ->
-Vertex4 ->-Vertex1 ->-Vertex2 ->-Vertex3 ->
```

7.3.2　基于广度优先策略的遍历

1. 基于广度优先策略的遍历算法描述

基于广度优先搜索策略遍历图，就是优先考虑直接近邻的结点，逐渐向远处扩展到图的所有结点。类似于二叉树的层次遍历，在基于广度优先策略的图的遍历操作中，通过设立一个队列结构来保存访问过的结点，以便在继续遍历中依次访问它们的尚未被访问过的邻接点。图的广度优先遍历算法描述如下：

1) 从图中选定的一个结点（设该结点下标为 m，结点值为 s）作为出发点，访问该结点。
2) 将访问过的结点 s 送入队列（Enqueue）。
3) 当队列不空时，进入以下的循环：
   a) 队头结点（设该结点值为 k，结点下标为 i）从队列出队（Dequeue），标记为 v_i。
   b) 访问与 v_i 有边相连且未被访问过的所有结点 v_n（结点值为 t，结点下标为 n），访问过的结点 v_n 入队。
4) 当队列空时，循环结束，说明从结点 s 开始能够到达的所有结点都已被访问过。

图 7.10 显示从不同的结点进行广度优先搜索遍历所得到的不同结点序列。

(a) 从顶点v_1出发的一种广度优先遍历序列$\{v_1, v_2, v_3, v_4\}$　(b) 从顶点v_3出发的一种广度优先遍历序列$\{v_3, v_1, v_2, v_4\}$

图 7.10　图的广度优先搜索

用广度优先算法对图进行遍历时，由于使用队列结构保存访问过的结点，若结点 v_1 在结点 v_2

之前被访问，则与结点 v_1 相邻接的结点将会在与结点 v_2 相邻接的结点之前被访问。

对于一个连通的无向图或强连通的有向图，从图的任一结点出发，进行一次广度优先搜索便可遍历全图；否则，只能访问图中的一个连通分量。

对于有向图，每条弧<v_i, v_j>被检测一次，对于无向图，每条边(v_i, v_j)被检测两次。

2. 邻接矩阵图的广度优先遍历算法实现

在以邻接矩阵存储的图 AdjacencyMatrixGraph 类中，增加一个成员方法 BreadthFirstShow 实现基于广度优先策略的遍历算法。该方法编码如下所示：

```
//从结点 m 开始的广度优先遍历, m 为起始结点序号
public void BreadthFirstShow(int m) {
    int i,n;
    Queue<T> q = new Queue<T>();                          //设置空队列
    // Queue<int> qi = new Queue<iny>();                  //设置空队列
    vertexList[m].Show();                                 //访问起始结点
    vertexList[m].Visited = true;                         //设置访问标记
    T k = vertexList[m].Data;
    q.Enqueue(k);                                         //访问过的结点k入队
    // qi.Enqueue(m);                                     //访问过的m结点入队
    while(q.Count!=0) {                                   //队列不空时进入循环
        k = q.Dequeue();                                  //队头元素出队
        i = IndexOf(k);                                   //i是结点k在结点数组中的下标
        // i = qi.Dequeue();                              //i是结点在结点数组中的下标
        n = 0;
        while(n<count){                                   //查找与k相邻且未被访问的结点
            if (AdjMat[i,n]!=0 && AdjMat[i,n]!=Infinity && !vertexList[n].Visited ) {
                vertexList[n].Show();
                vertexList[n].Visited = true;
                q.Enqueue(vertexList[n].Data);
                qi.Enqueue(n);
            } else
                n++;
        }
    }
}
//从值为k的结点开始的广度优先遍历
public void BreadthFirstShow(T k) {
    int i = IndexOf(k);                                   // 结点k的下标，从开始
    BreadthFirstShow(i);
}
```

在【例 7.1】的 Main 方法中，增加调用图的广度优先遍历操作的语句：

```
g.BreadthFirstShow();
```

该代码运行的结果如下：

```
广度优先遍历:
-Vertex1 ->-Vertex2 ->-Vertex3 ->-Vertex4 ->
-Vertex2 ->-Vertex1 ->-Vertex3 ->-Vertex4 ->
-Vertex3 ->-Vertex1 ->-Vertex2 ->-Vertex4 ->
-Vertex4 ->-Vertex1 ->-Vertex3 ->-Vertex2 ->
```

3. 邻接表图的广度优先遍历算法实现

在以邻接表存储的图 Graph 类中，增加一个成员方法 BreadthFirstShow 实现图的广度优先遍历操作。该方法如下所示：

```
//图的广度优先遍历, m 为起始结点序号
public void BreadthFirstShow(int m) {
    int i, j;
    Queue<T> q = new Queue<T>();                          //设置空队列
    // Queue<int> qi = new Queue<int>();                  //设置空队列
    Console.Write("-" + nodes[m].Data + " ->");           //访问起始结点
    nodes[m].Visited = true;                              //设置访问标记
    T k = nodes[m].Data;
    q.Enqueue(k);                                         //访问过的结点k入队
    // qi.Enqueue(m);                                     //访问过的m结点入队
    while (q.Count != 0) {                                //队列不空时
        k = q.Dequeue();                                  //出队
        i = IndexOf(k);                                   //i是结点k在结点数组中的下标
        // i = qi.Dequeue();                              //i是结点在结点数组中的下标
        for (j = 0; j < nodes[i].Neighbors.Count; j++) {
            if (!nodes[i].Neighbors[j].Visited) {     //查找与i相邻且未被访问的结点
                Console.Write("-" + nodes[i].Neighbors[j].Data + " ->");
                nodes[i].Neighbors[j].Visited = true;
                q.Enqueue(nodes[i].Neighbors[j].Data);
                // qi.Enqueue(Index(nodes[i].Neighbors[j].Data));
            }
        }
    }
}
//从值为k的结点开始的广度优先遍历
public void BreadthFirstShow(T k) {
    int i = IndexOf(k);                                   // 获取结点k的下标，从0开始
    BreadthFirstShow(i);
}
```

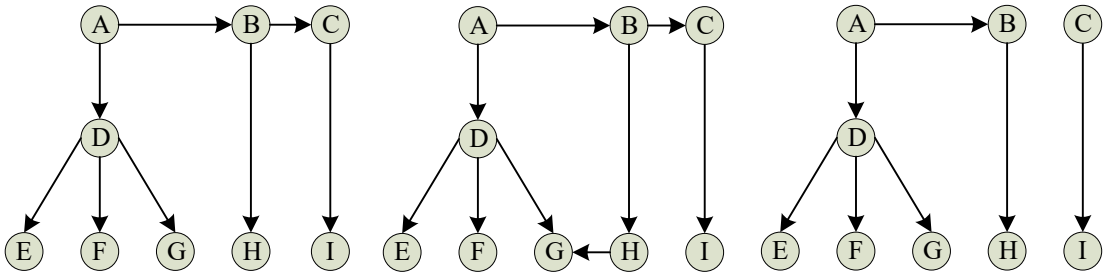
7.4　最小代价生成树

图（graph）可以看成是树（tree）和森林（forest）的推广，树和森林则分别是图的某种特例，下面首先从图的角度来看待树和森林，然后讨论图的生成树、最小代价生成树等概念。

7.4.1 树和森林与图的关系

树和森林都是特殊的图。树是连通的无回路的无向图，树中的悬挂点称为叶子结点，其他的结点称为分支结点。森林则是诸连通分量均为树的图。

树是一种简单图，因为它无环也无重边。若在树中任意一对结点之间加上一条边，则形成图中的一条回路；若去掉树中的任意一条边，则树变为森林，整体是一种非连通图。树和森林与图的关系如图 7.11 所示。



(a) 树是连通图　　(b) 树加上一条边成为有回路的图　　(c) 树去掉一条边成为森林，非连通图

图 7.12 树和森林与图的关系

对一棵树 T 而言，其结点数为 n，边数为 m，那么有 $n - m = 1$。

7.4.2 图的生成树

1. 生成树的定义

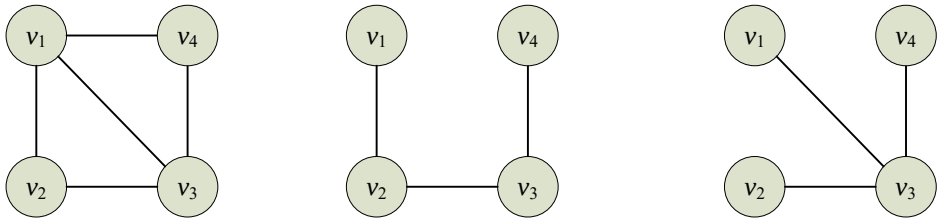
无向图 G 的生成子图 T 如果是一颗树，则树 T 称为图 G 的生成树（spanning tree）。由定义知，图 G 的生成树 T 具有与图 G 相同的结点集合，它包含原图结构中尽可能少的边，但仍然构成连通图。如果在生成树中加入一条边，则产生回路；如果删除生成树中的一条边，生成树将被分成不连通的两棵树。

假设有一个铁路网络图，图中结点表示城市，边表示连接两个城市的铁路线路，则该图的生成树包含图中的所有结点（城市）和尽可能少的边（铁路线路），但任意两个城市仍然是可通达的。

设 G=(V, E)是一个连通的无向图，从图 G 的任意一个结点 v 出发进行一次遍历所经过的边的集合为 TE，则 T=(V, TE)是 G 的一个连通子图，它实际上是图 G 的一棵生成树。可见，任意一个连通图都至少有一棵生成树。

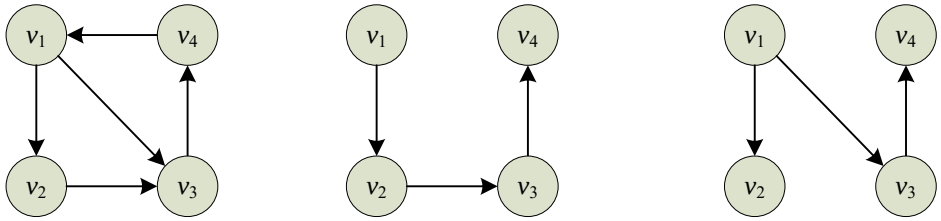
图的生成树不是唯一的，从不同的结点出发遍历图的结点可以得到不同的生成树，采用不同的搜索策略也可以得到不同的生成树。具有 n 个结点的连通无向图的生成树有 n 个结点和 n - 1 条边。对图的任意两个结点 v_i 和 v_j，在生成树中，v_i 和 v_j 之间只有唯一的一条路径。

以深度优先策略遍历图得到的生成树，称为深度优先生成树；以广度优先遍历得到的生成树，称为广度优先生成树。连通无向图的生成树如图 7.13 所示，强连通有向图的生成树如图 7.14 所示。



(a) 无向图　(b) 从v_1出发的深度优先生成树　(c) 从v_3出发的广度优先生成树

图 7.13 无向图及其生成树



(a) 有向图　(b) 从v_1出发的深度优先生成树　(c) 从v_1出发的广度优先生成树

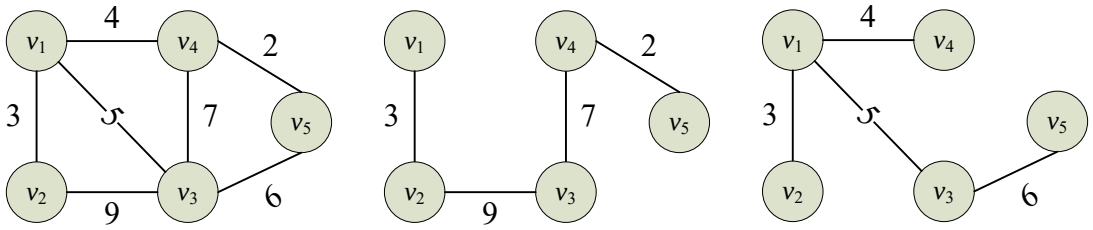
图 7.14 有向图及其生成树

2. 生成森林

如果图 G 是非连通的无向图，它的各连通分量的生成树构成图 G 的生成森林（spanning forest）。生成森林中的每棵树是对非连通图 G 进行一次遍历所能到达的一个连通分量。

3. 带权图的生成树

图 7.15 显示一个带权图及其生成树。因为带权图的权值常用来表示代价、距离等，所以称一个带权图的生成树各边的权值之和为生成树的代价（cost）。一般地，一个连通图的生成树不止一棵，各生成树的代价可能不一样，图 7.15 中图的两棵生成树的代价分别为 21 和 18。



(a) 带权的无向图　(b) 从v_1出发的深度优先生成树　(c) 从v_1出发的广度优先生成树

图 7.15 带权图及其生成树

7.4.3 最小代价生成树

在表示城市之间的铁路网络的图中，结点表示城市，边表示城市之间的铁路线路，边上的权值表

示相应的路程。该图的不同生成子树的边长之和（铁路总长）可能是不同的，其中某些生成树给出连接每个城市的具有最小代价（路程）的铁路布局。许多生产和科研问题都蕴含着与这个例子类似的需求，即需要求取图的最小代价生成树。

若 G 是一个连通的带权图，则它的生成树 T 中各边的权值之和 $w(T)$称为图 G 的生成树的代价（cost），它等于：

$$w(T)=\sum_{e\in T}w(e)$$

其中 $w(e)$为边 e 上的权。代价最小的生成树称为图的最小代价生成树（minimum cost spanning tree，MCST），简称最小生成树（minimal spanning tree，MST）。

按照生成树的定义，具有 n 个结点的连通图的生成树有 n 个结点和 n - 1 条边，其中最小代价生成树具有下列 4 条性质：

- 包含图中的 n 个结点。
- 包含且仅包含图中的 n-1 条边。
- 不包含产生回路的边。
- 最小生成树是各边权值之和最小的生成树。

在图中构造最小生成树有两种典型的算法：克鲁斯卡尔（Kruskal）算法和普里姆（Prim）算法。它们都在逐步求解的过程中利用了最小生成树的一条简称为 MST 的性质：假设 G=(V, E)是一个连通加权图，U 是 V 的一个非空子集。若 e(u, v)是一条具有最小权值的边，其中 $u\in$U，$v\in$V - U，则必存在一颗包含边 e(u, v)的最小生成树。

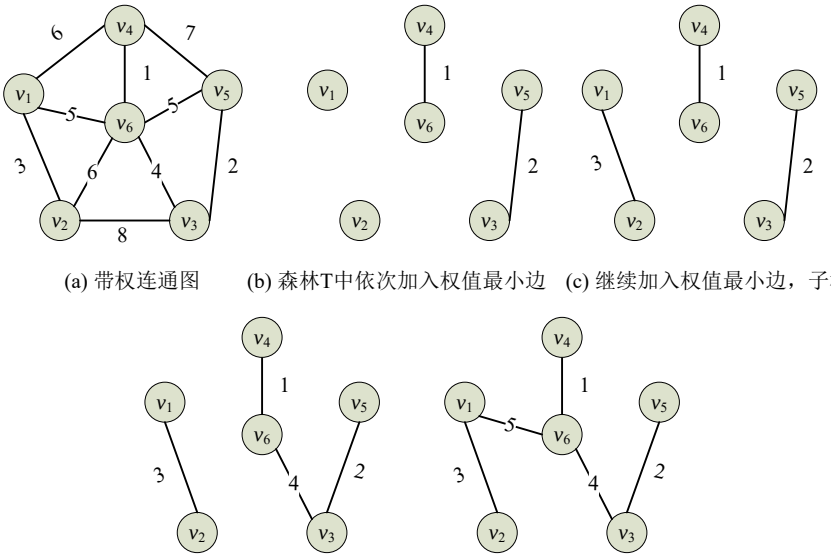
1. Kruskal 算法

设连通带权图 G=(V, E)有 n 个结点和 m 条边。克鲁斯卡尔算法的基本思想是，最初先构造一个包括全部 n 个结点、但无边的森林 $T=\{T_1, T_2, \cdots, T_n\}$；然后依照边的权值从小到大的顺序，逐边将它们放回到所关联的结点上，但删除会生成回路的边；由于边的加入，使 T 中的某两棵树合并为一棵，森林 T 中的树的棵数减 1。经过 n - 1 步，最终得到一棵有 n - 1 条边的最小代价生成树。

Kruskal 算法描述如下：

1) 构造 n 个结点和 0 条边的森林；依照边的权值大小从小到大将边集排序。
2) 进入循环，依次选择权值最小、但其加入不产生回路的边加入森林，直至该森林变成一棵树为止。

以 Kruskal 算法构造连通带权图的最小生成树的过程如图 7.16 所示。



(a) 带权连通图　(b) 森林T中依次加入权值最小边　(c) 继续加入权值最小边，子树合并

(d) 继续加入权值最小边　(e) 最小代价生成树

图 7.16 以 Kruskal 算法构造连通带权图的最小生成树

构造最小生成树时，从权值最小的边开始，选择 n - 1 条权值较小的边构成无回路的生成树；每一步选择权值尽可能小的边，但是，并非每一条当前权值最小的边都可选。这样的逐步求解过程使得生成树的代价最小。

2. Prim 算法

普里姆算法从连通带权图 G=(V, E)的某个结点 s 逐步扩张成一颗生成树。Prim 算法描述如下：

1) 生成树 T=(U, E_T)开始仅包括初始结点 s，即 U ={ s}。
2) 进入循环，选择与 T 相关的具有最小权值的边 e(u, v_i)，u∈U，v_i∈V - U，将该边与结点 v_i 加入到生成树 T 中，直至产生一个 n - 1 条边的生成树。

以 Prim 算法构造连通带权图的最小生成树的过程如图 7.17 所示。

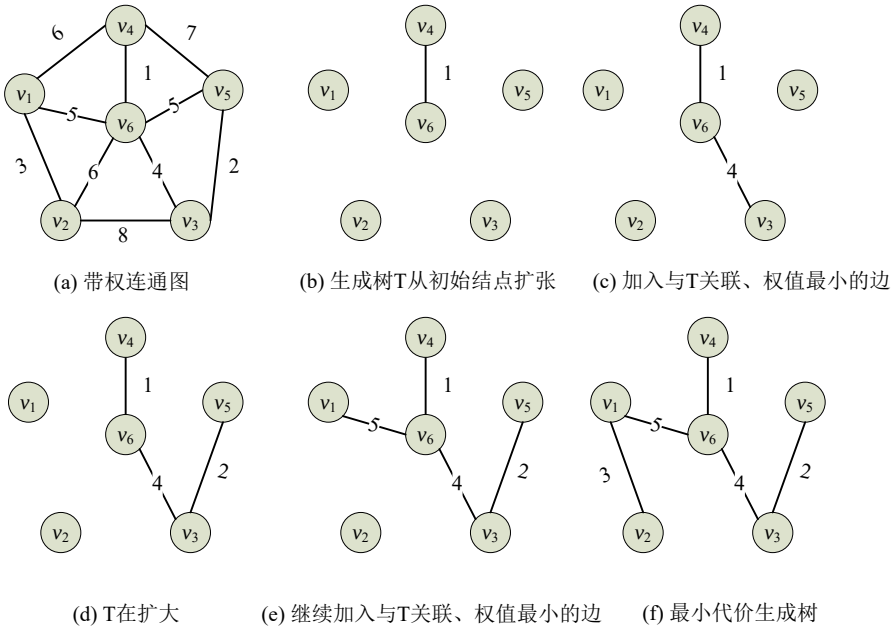


图 7.17 以 Prim 算法构造连通带权图的最小生成树

7.5 最短路径

在城市间的铁路网络图，从一个城市到达另一城市可能存在多条路径，不同路径的长度一般是不同的，其中有一条路径最短，这个例子包含着最短路径的概念。

设有一个带权图 G=(V, E)，如果图 G 中从结点 v_s 到结点 v_n 的一条路径为（v_s, v_1, v_2, …, v_n），其路径长度不大于从 v_s 到 v_n 的所有其他路径的长度，则该路径是从 v_s 到 v_n 的最短路径(shortest path)，v_s 称为源点，v_n 称为终点。

在边上权值非负的带权图 G 中，若给定一个源点 v_s，求从 v_s 到 G 中其他结点的最短路径称为单源最短路径问题。依次将图 G 中的每个结点作为源点，求每个结点的单源最短路径，则可求解所有结点间的最短路径问题。

例如，对于图 7.17 (a)中的带权图，其邻接矩阵为

$$A=\begin{bmatrix} 0 & 3 & \infty & 6 & \infty & 5 \\ 3 & 0 & 8 & \infty & \infty & 6 \\ \infty & 8 & 0 & \infty & 2 & 4 \\ 6 & \infty & \infty & 0 & 7 & 1 \\ \infty & \infty & 2 & 7 & 0 & 5 \\ 5 & 6 & 4 & 1 & 5 & 0 \end{bmatrix}$$

以 v_1 为源点的单源最短路径如表 7-1 所示。

表 7-1 以 v_1 为源点的单源最短路径

源 点	终 点	路 径	路径长度	最短路径
v_1	v_2	(v_1, v_2)	3	✓
		(v_1, v_6, v_2)	11	

	v_3	(v_1, v_6, v_3)	9	✓
		(v_1, v_2, v_3)	11	
	v_4	(v_1, v_4)	6	✓
		(v_1, v_6, v_4)	6	✓
	v_5	(v_1, v_6, v_5)	10	✓
		(v_1, v_6, v_3, v_5)	11	
	v_6	(v_1, v_6)	5	✓
		(v_1, v_4, v_6)	7	

这类最短路径问题可用函数迭代法求解。考虑有 n 个结点的网络，直接用编号 1, 2, …, n 标识结点，需要求解结点 i（i=1, 2, …, n-1）到结点 n 的最小距离。函数迭代法将用到下列基本方程：

$$\begin{cases} f(i) = \min\limits_{1 \le j \le n} \{c_{ij} + f(j)\}, i = 1, 2, \ldots, n-1 \\ f(n) = 0 \end{cases}$$

其中 $f(i)$表示结点 i 到结点 n 的最小距离，$f(j)$表示结点 j 到结点 n 的最小距离，c_{ij} 是连接结点 i 和结点 j 之间的距离（或费用，如果结点 i 和结点 j 之间有边，c_{ij} 就等于边上的权值，否则设为无穷大）。该方程的含义是，为求结点 i 到结点 n 的最小距离，先对每个结点 j，计算结点 i 到结点 j 的距离 c_{ij}，加上结点 j 到结点 n 的最小距离，计算出的若干结果中值最小的就是结点 i 到结点 n 的最小距离。

在上面的方程中，$f(i)$和 $f(j)$都是未知量，需要从已知条件出发，逐步迭代求解出最优解。迭代的基本思想是，先计算各结点经 1 步（即经过一条边）达到结点 n 的最短距离 $f_1(i)$，再计算各结点经 2 步到达结点 n 的最短距离 $f_2(i)$，依次类推，计算出结点 i 经 k 步到达结点 n 的最短距离为 $f_k(i)$。具体步骤如下：

1) 取初始函数 $f_1(i)$的值为各结点 i 经 1 步达到结点 n 的距离 c_{in}，其中 c_{nn}=0。

2) 对于 k=1, 2, …，用上面的方程求 $f_k(i)$：

$$\begin{cases} f(i) = \min\limits_{1 \le j \le n} \{c_{ij} + f(j)\}, i = 1, 2, \ldots, n-1 \\ f(n) = 0 \end{cases}$$

3) 当计算到对所有 i=1, 2, …n，均成立 $f_k(i) = f_{k-1}(i)$时停止。迭代次数不会超过 n-1，理论上可以证明，用函数迭代法确定的值序列$\{f_k(i)\}$是个单调非增序列，并收敛于 $f(i)$。即算法迭代停止时的 $f_k(i)$就是结点 i 到结点 n 的最小距离 $f(i)$。达到最优后，可以根据计算过程回溯出从结点 i 达到结点 n 的最短路线。

习题 7

7.1 简述图的基于邻接矩阵的存储结构和基于邻接表的存储结构。

7.2 对于图 7.中的无向带权图，请给出：

1) 图的邻接矩阵。

2) 图中每个结点的度。

3) 从结点 *a* 出发，进行深度优先和广度优先遍历所得到路径和结点序列。

4) 以结点 *a* 为起点的一棵深度优先生成树和一棵广度优先生成树。

5) 分别以 Kruskal 算法和 Prim 算法构造最小生成树。在 Prim 算法中假设从结点 a 开始。

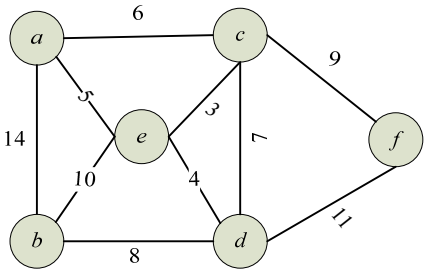


图 7.18 无向图

7.3 对于**图 7.18** 中的有向图，请给出：

1) 图的邻接矩阵。

2) 图的邻接表。

3) 图中每个结点的度、入度和出度。

4) 图的强连通分量。

5) 从结点 v_1 出发，进行深度优先和广度优先遍历所得到的结点序列和边的序列。

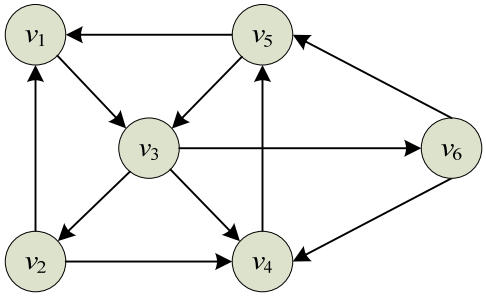


图 7. 18 有向图

7.4 有 6 个城市，任何两个城市之间都有一条道路连接，6 个城市两两之间的距离如下表所示，用有权图来表示这 6 个城市之间的道路连接，计算城市 1 到城市 6 的最短距离。

	城市1	城市2	城市3	城市4	城市5	城市6
城市1	0	2	3	1	12	15
城市2	2	0	2	5	3	12

城市3	3	2	0	3	6	5
城市4	1	5	3	0	7	9
城市5	12	3	6	7	0	2
城市6	15	12	5	9	2	0

7.5 在邻接矩阵图类 AdjacencyMatrixGraph 中实现查找某个特定结点的操作：

```
public int IndexOf(Vertex<T> n);
```

7.6 在邻接表图类 Graph 中实现判断结点间是否存在边的操作：

```
public bool ContainsDirectedEdge(T from, T to);

public bool ContainsUndirectedEdge(T from, T to);
```