第5章

图和树知识的建模与问题求解

本章讨论离散数学基础课程中有关图和树基础知识的问题求解，首先给出我们要求解的问题范围，然后对相关的知识进行面向对象建模，给出主要的类与方法的设计，最后给出问题求解的程序实现与运行效果。

5.1 图和树问题求解的需求建模

作为离散数学基础课程的教材，在主教材图与树这一部分，主要是在介绍图和树的基本概念和基本性质的基础上，重点讨论了有关图和树的一些基本算法，具体算法如下。

（1）图的遍历算法，包括图的**深度优先**和**广度优先**遍历算法。

（2）根树的遍历算法，包括根树的**前序**、**中序**和**后序**遍历算法。

（3）带权图的最短路径算法，即**Dijsktra算法**。

（4）带权图的最小生成树算法，包括**Kruskal算法**和**Prim算法**。

（5）哈夫曼树构建算法，即**Huffman算法**。

要求学生能掌握这些算法的基本思想，针对小规模的图和树，能用合适的形式给出算法执行的过程和结果。

因此，在我们要开发的辅助教学软件Deedm的这一模块，主要让用户以合适的方式输入图和树的信息，包括带权图的信息，然后再以合适的形式展示算法执行过程中的关键信息，以帮助用户理解算法的基本思想。基于这一点，下面的需求分析与建模主要侧重于明确用例的输入和输出形式，而在设计与实现阶段的建模则侧重如何实现这些图与树相关的算法，以及如何记录算法执行过程中的关键信息。

5.1.1 用例分析与建模

根据要展示的算法，这一模块有5个用例：①图的遍历；②树的遍历；③带权图最短路径计算；④带权图最小生成树计算；⑤哈夫曼树构造。图5.1给出了这一部分的用例图。这些用例图的交互事件流程都比较简单，用户按照预先确定的形式录入图、带权图或根树，选择要做的展示，例如对于图的遍历而言，选择是用先深遍历还是先广遍历，抑或两种都展示等；对于图，特别是带权图，应该还可选择是否要展示图的关联矩阵等，然后单击表示开始展示的按钮，系统就按照要

求，以预先确定的形式展示算法的执行过程以及执行结果。表 5.1给出了用例“带权图最小生成树计算”的交互事件流程作为示例。下面主要确定在用例中如何录入图、带权图或根树的顶点和边，以及边的权的信息，并对每种算法以何种形式展示算法的执行过程进行设计，从而完成用例的分析与建模。

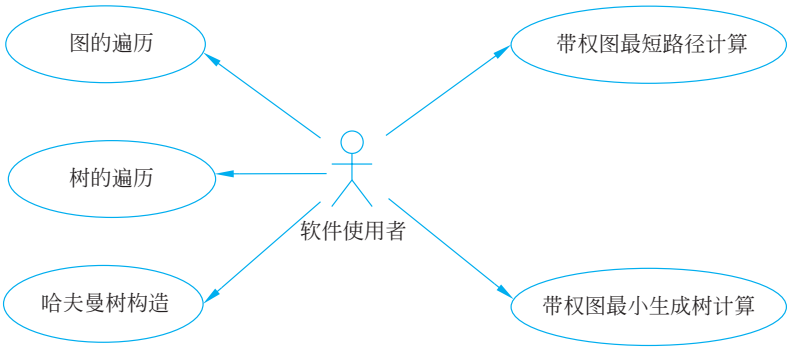

图 5.1　图与树问题求解部分的用例图

表 5.1　用例 4. 带权图最小生成树计算

【用例名称】带权图最小生成树计算
【用例场景】 参与者：用户
【用例价值（求解的离散数学问题）】 展示使用 Kruskal 算法和 Prim 算法计算带权图最小生成的计算过程与计算结果
【用例描述】 （1）用户按照预先确定的形式输入带权图的顶点、边和边的权，或者选择随机生成输入信息 （2）用户选择使用何种算法（Kruskal 算法或/和 Prim 算法）进行展示 （3）用户选择是否给出图的直观形式或/和图的关联矩阵，以及是否展示算法执行过程 （4）用户单击按钮要求进行计算和展示 （5）系统分析输入是否合法，如合法则按照指定的方式展示算法执行过程和结果，否则反馈错误

用例“图的遍历”需要输入有向图或无向图的顶点和边。按照图的定义，顶点是一个集合 $V=\{v_1,v_2,\cdots,v_n\}$，因此可按照输入集合的方式输入顶点集。考虑到可能用一些特殊的符号标记图的顶点，例如在展示表达式的抽象语法树时，可能用运算符 $+,*$ 等标记顶点，但这些特殊符号作为顶点的唯一标识并不合适，因此使用‘id(label)’的形式表示一个顶点，其中 id 是顶点的唯一标识，而 label 用于在展示图的直观形式时标记图的顶点，左右圆括号用于分隔顶点标识 id 与顶点标签 label。

图的每条边关联两个顶点，对于有向图，这两个顶点有一个是起始顶点，一个是终止顶点。两个顶点之间可能有多条边，这些边称为重边。在存在重边的时候就不能只用两个顶点来决定一条边，因此，按照主教材给出的边的描述，我们使用'边 id=（顶点 id, 顶点 id)'的形式表示一条无向边，而用'边 id=〈顶点 id, 顶点 id〉'的形式表示一条有向边，这

里边id用于标识这条边，在图的边集中具有唯一性，顶点id是在录入顶点时确定的顶点标识（注意不是顶点标签label）。

例如，图5.2给出了主教材第9章的例子9.3中的G_1。对这个图，顶点用文本串"{v1, v2, v3, v4, v5, v6}"作为输入，而边用文本串"{e1=(v1,v3), e2=(v1,v3), e3=(v1,v3), e4=(v1,v2), e5=(v1,v4), e6=(v2,v4), e7=(v2,v3),e8=(v3,v4), e9=(v4,v4), e10=(v2,v5)}"作为输入。由于当前的GraphViz在渲染图的时候还不支持在图的顶点或边的标签中使用数学公式，所以目前只能用纯文本形式的v1，而不是数学公式形式的v_1作为顶点的标签。这里在给出顶点时，没有用圆括号给出顶点的标签就意味着顶点的标签与顶点的标识相同。在输入边时也可省略边标识id以及等号=，不过这样在用GraphViz绘制出图形后无法区别重边。

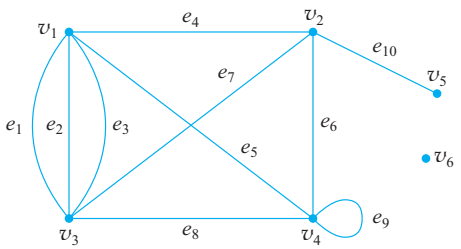

图 5.2 用于说明如何输入图的顶点和边的示例

展示图与树的算法运行过程的最好方式是动画，但动画的实现过于复杂，目前只能选择简单的图文混合的静态形式描述算法的运行过程。对于图的先深遍历，理解算法的关键在于了解算法访问顶点的先后顺序，以及访问每个顶点时栈中的信息，而对于图的先广遍历，则关键在于访问顶点的先后顺序，以及队列中的信息。因此，对于用例“图的遍历”，我们要求给出先深遍历或先广遍历算法的主循环的每一步的已经访问顶点的序列，以及栈或队列的信息即可。例如，对于图5.2给出的图G_1，先深遍历应像下面的形式展示每一步已访问的顶点序列T和相应的栈S的信息。

$$\begin{aligned}
&\text{Step 1} : \quad T = \langle v_1 \rangle \qquad S = \langle v_1 \rangle \\
&\text{Step 2} : \quad T = \langle v_1, v_2 \rangle \qquad S = \langle v_1, v_2 \rangle \\
&\text{Step 3} : \quad T = \langle v_1, v_2, v_3 \rangle \qquad S = \langle v_1, v_2, v_3 \rangle \\
&\qquad\qquad \vdots
\end{aligned}$$

而先广遍历应像下面的形式展示每一步已访问的顶点序列T和相应的队列Q的信息。

$$\begin{aligned}
&\text{Step 1} : \quad T = \langle v_1 \rangle \qquad Q = \langle v_1 \rangle \\
&\text{Step 2} : \quad T = \langle v_1, v_2, v_3, v_4 \rangle \qquad Q = \langle v_2, v_3, v_4 \rangle \\
&\text{Step 3} : \quad T = \langle v_1, v_2, v_3, v_4, v5 \rangle \qquad Q = \langle v_3, v_4, v_5 \rangle \\
&\qquad\qquad \vdots
\end{aligned}$$

对于用例“树的遍历”也可用录入无向图的方式录入根树的顶点和边的信息，只是要进一步给出根是哪个顶点。用例“树的遍历”只需要使用顶点序列展示根树的前序、中序

或后序遍历时访问顶点的先后顺序即可。例如，对主教材的例子9.27给出的表达式的抽象语法树（如图5.3左边所示），顶点和边分别使用下面的文本串输入。

```
顶点：{times(*),plus(+),minus(-),divide(/),three(3),seven(7),anotherthree(3),
      five(5),two(2)}
边：{(times,plus), (times,minus), (plus,three), (plus,divide), (divide,seven),
    (divide,anotherthree), (minus,five), (minus,two)}
```

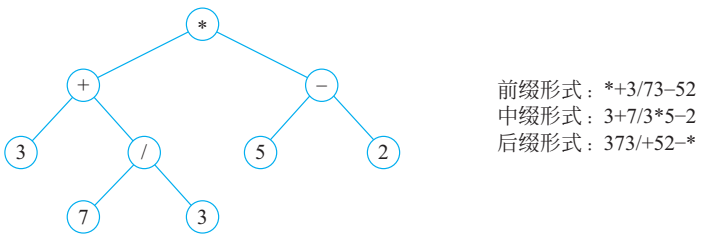

图 5.3　用例“树的遍历”的输入和输出结果示例

这里由于顶点的标签是∗、＋等运算符，所以顶点标识选用更有意义的字符串表示，每条边所关联的两个顶点用顶点标识表示。对于图5.3中的根树，用例“树的遍历”给出的输出应像这个图右边所给出的前序、中序和后序遍历序列以分别表示表达式的前缀、中缀和后缀表示形式。

用例“带权图最短路径计算”和“带权图最小生成树”需要录入带权图的信息，这时顶点信息的录入和普通图顶点信息录入一样，只是在录入边的时候需要加上边的权。我们使用'边id[数值]=（顶点id, 顶点id)'的形式表示一条带权的无向边（有向边则使用角括号括住起始顶点和终止顶点），其中用方括号括住一个实数数值表示边的权。例如，对于主教材的例子9.28给出的带权有向图（如图5.4所示），其顶点、边和边权信息使用下面的文本串输入，这里不需要给出边的标签，所以直接用'[数值]=⟨顶点id,顶点id⟩'的形式给出边的信息。

```
顶点：{v1, v2, v3, v4, v5, v6}
边：{[1]=<v1,v3>, [7]=<v1,v2>, [6]=<v3,v2>, [4]=<v2,v4>, [1]=<v2,v6>, [3]=<v5,v2>,
    [2]=<v3,v5>, [7]=<v3,v6>, [5]=<v5,v4>, [3]=<v6,v5> }
```

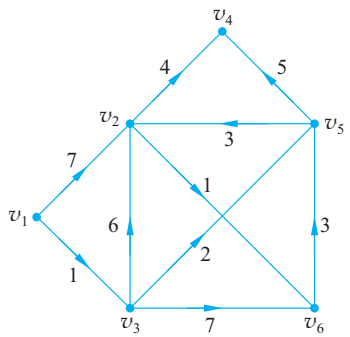

图 5.4　带权图的输入示例

主教材使用表格形式给出Dijkstra算法计算带权图最短路径的过程，所以，针对图5.4给出的带权图，用例“带权图最短路径计算”也使用如表5.2所示的形式展示Dijkstra算法的计算过程，并且进一步给出起始顶点到每个顶点的最短路径和最短距离作为算法的输出结果。

表 5.2 用例“带权图最短路径计算”的输出结果示例

步 骤	v_1	v_2	v_3	v_4	v_5	v_6
第0步	[0]	$7/v_1$	$1/v_1$	∞/v_1	∞/v_1	∞/v_1
第1步		$7/v_1$	[1]$/v_1$	∞/v_1	$3/v_3$	$8/v_3$
第2步		$6/v_5$		$8/v_5$	[3]$/v_3$	$8/v_3$
第3步		[6]$/v_5$		$8/v_5$		$7/v_2$
第4步				$8/v_5$		[7]$/v_2$
第5步				[8]$/v_5$		

带权（无向）图的最小生成树的计算过程也可使用表格展示，若使用Kruskal算法计算，则展示算法主循环体每一次循环所选的边，若使用Prim算法计算，则展示算法主循环体每一次循环所考虑的边，以及选择的顶点和边。无论选择哪个算法，在用表格展示计算过程之后，还给出算法的计算结果，即得到的最小生成树及其权。

例如，对于图5.4给出的带权图，将其看作带权无向图，则用例“带权图最小生成树”中如果选择展示Kruskal算法的计算过程，则应得到如表5.3所示的选边过程，其中给出了按照边权从小到大每一步考虑的边，以及是否因为会有回路而最终是否选中这条边。如果选择展示Prim算法的计算过程，则应得到如表5.4所示的选择顶点和边的过程，其中给出每一步已经在树中的顶点、不在树中的顶点、所考虑的边（即一个顶点在树中，另一顶点不在树中所有的边）、选中的边（即所考虑的边中权最小的顶点），以及选中的顶点（即选中的边中尚不在树中的顶点）。

表 5.3 用例“带权图最小生成树”选用 Kruskal 算法的输出结果示例

步 骤	考 虑 的 边	是 否 选 中
Step 1	[1] = (v1, v3)	✓
Step 2	[1] = (v2, v6)	✓
Step 3	[2] = (v3, v5)	✓
Step 4	[3] = (v5, v2)	✓
Step 5	[3] = (v6, v5)	×
Step 6	[4] = (v2, v4)	✓

表 5.4 用例“带权图最小生成树”选用 Prim 算法的输出结果示例

步骤	树中顶点	不在树中顶点	考虑的边	选中的边	选中的顶点
Step 1	v1	v2,v3,v4,v5,v6	[1]=(v1,v3),[7]=(v1,v2)	[1]=(v1,v3)	v3
Step 2	v1,v3	v2,v4,v5,v6	[7]=(v1,v2),[6]=(v3,v2), [2]=(v3,v5),[7]=(v3,v6)	[2]=(v3,v5)	v5

续表

步骤	树中顶点	不在树中顶点	考虑的边	选中的边	选中的顶点
Step 3	v1,v3,v5	v2,v4,v6	[7]=(v1,v2),[6]=(v3,v2), [3]=(v5,v2),[7]=(v3,v6), [5]=(v5,v4),[3]=(v6,v5)	[3]=(v5,v2)	v2
Step 4	v1,v3,v5,v2	v4,v6	[4]=(v2,v4),[1]=(v2,v6), [7]=(v3,v6),[5]=(v5,v4), [3]=(v6,v5)	[1]=(v2,v6)	v6
Step 5	v1,v3,v5,v2,v6	v4	[4]=(v2,v4),[5]=(v5,v4)	[4]=(v2,v4)	v4

构建哈夫曼树需要录入树的叶子顶点及其权，用'顶点id[数值]'的形式输入一个叶子顶点的标识，用方括号括住的实数数值表示它的权，多个叶子顶点的信息用逗号分隔。对于构造过程的展示，按照主教材的描述，算法主循环体每循环一次后得到一个森林，其中每棵树的根有一个权，按照这些权从小到大排列，所以用例“哈夫曼树构造”的输出应该给出每一次循环后得到的树根顶点及其权，也要绘制出每棵树（只有根顶点的树可省略）。

例如，对于主教材的例9.31，可以文本串"{a[15],b[12],c[25],d[8],e[20],f[6],g[8],h[6]}"作为用例“哈夫曼树构造”的输入，表示叶子的标签分别是字母a,b,c等，叶子的权分别是15,12,25等。对于该输入，用例“哈夫曼树构造”通过给出哈夫曼树构造过程中的根顶点序列以及得到的中间树展示算法执行过程，例如，第三次循环后得到的顶点序列是：

$$20(e) \quad 24(t_3) \quad 25(c) \quad 31(t_4)$$

而得到的中间树如图5.5所示。

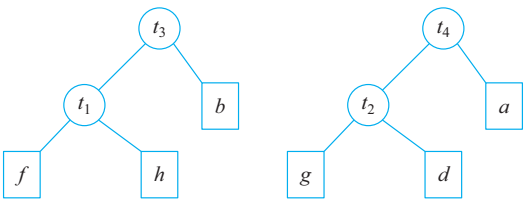

图 5.5 用例“哈夫曼树构造”的输出示例

至此确定了图、根树、带权图，以及哈夫曼树叶子顶点信息的输入形式，也给出了用例“图的遍历”“树的遍历”“带权图最短路径计算”“带权图最小生成树计算”“哈夫曼树构造”的预计输出形式。最后需要指出的是，用户可选择主教材中的例子或问题，也可选择随机生成图、树、带权图以及哈夫曼树叶子及其权的信息作为输入，这时系统自动按照这里确定的输入形式处理输入信息，因此教材例子和随机生成的输入信息也可作为示例，让用户清楚如何输入图、树、带权图和哈夫曼树叶子的信息。

5.1.2 领域实体分析与建模

根据上面的用例分析，图和树的问题求解这一模块主要是图与树的遍历、带权图最短路径与最小生成树计算，以及哈夫曼树构造等经典图和树算法的计算过程和计算结果的展示，涉及的实体主要包括图、树、带权图和哈夫曼树，以及图的顶点、边、带权边、带权顶点、路径和带权路径。图5.6给出了这些实体和它们之间的关系。

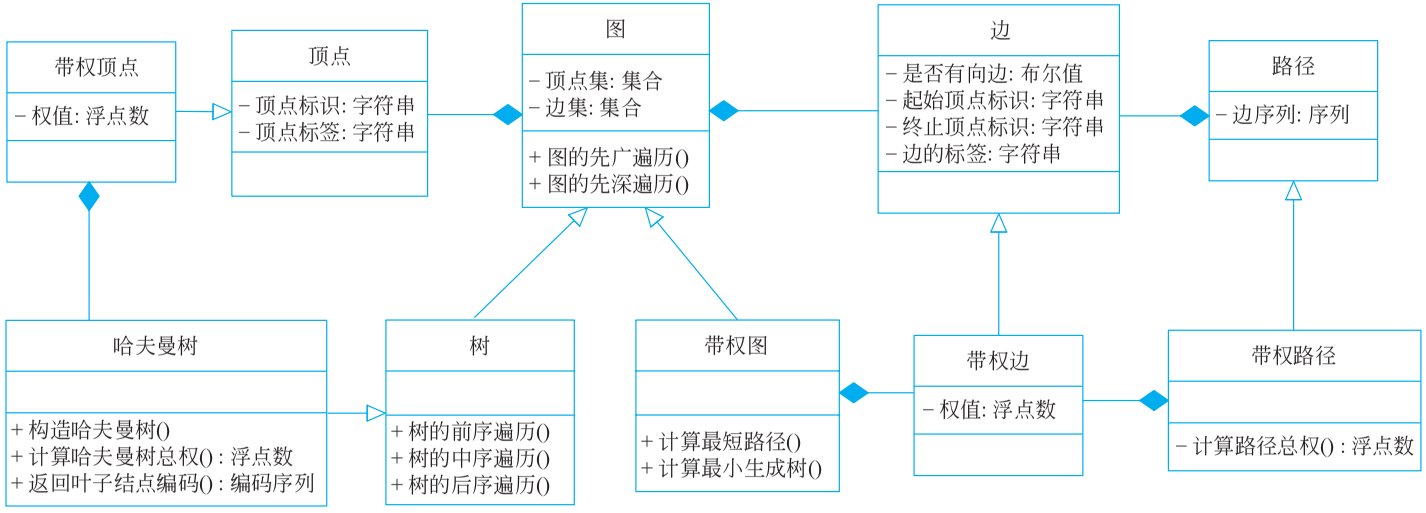

图 5.6　图和树问题求解相关的实体和它们之间的关系

从领域实体建模角度看，图由顶点和边构成，顶点和边与图之间是紧密的聚合关系，即脱离图单独考虑顶点或边是没有意义的。类似地，边和路径之间也是紧密聚合关系，路径就是由一系列边构成，不能脱离边单独考虑路径。树和带权图都是特殊的图，这里带权图由顶点和带权边构成，带权边的序列也构成了带权路径。哈夫曼树是特殊的树，它的顶点是一种带权顶点。

图5.6将图的遍历这个职责分配给了图这个实体，树的遍历分配给了树，最短路径和最小生成树的计算分配给了带权图，而哈夫曼树构造分配给了哈夫曼树，从而通过这些实体可实现上面给出的用例。下面我们将基于这些基本的实体，讨论它们在软件中对应的类，围绕类进行软件设计与实现。

5.2 图和树问题求解的设计

根据确定的图与树问题求解的用例，这一模块涉及的实体主要是图、树、带权图和哈夫曼树等，要实现图与树的遍历、带权图最短路径与最小生成树计算，以及哈夫曼树构造等算法的计算过程和计算结果的展示。本节围绕这些实体在软件中所对应的类，说明这一模块的类的设计，重点在于根据用例分析中给出的预期输出，设计如何展示这些算法的计算过程。

5.2.1 展示图遍历算法的类设计

求解图与树问题的核心实体是图，这也是展示图遍历算法的核心载体。但考虑到图的涵义广泛，在软件中用一个类对应图这个实体，那么这个类可能会有过多的职责，因此我们首先设置了一个类 `AbstractGraph` 表示抽象的图，用于存储图的顶点和边，维持边与顶点之间的关联关系，然后再派生其他表示图的类，承担一些具体的职责。例如，对于展示图的遍历算法，我们设计了类 `DefaultGraph` 作为 `AbstractGraph` 的子类，主要实现图的先深和先广遍历算法，展示这两个算法的执行过程。

我们用 Java 语言的接口 `GraphNode` 表示类 `AbstractGraph` 中存储的图的顶点。之所以使用接口而不是类表示类 `AbstractGraph` 中的顶点，是因为 Java 语言仅支持单继承。一个类可继承其他的类的数据和方法，而只需实现接口 `GraphNode` 定义的一些简单方法就可作为图中的顶点。例如，某些情况下可能需要将某些特殊的子图看作一个大图中的顶点，这样可设计一个类作为 `AbstractGraph` 的子类，并实现接口 `GraphNode`，使得这种表示图的类还可以是顶点。

简单地说，Java 语言的接口提供了这样一种机制：只要某个类实现了一个接口规定的一组特定的方法，那么它就可以当作这个接口表示的实体来用，一个类还可实现多个接口，从而从不同维度来使用这个类。与 C++ 语言的多继承机制相比，Java 语言的接口机制更简单，又不失灵活性。Java 程序中声明的接口往往只有少量且含义简单的方法。例如，这里设计的接口 `GraphNode` 只要求实现它的类提供分别返回顶点标识和顶点标签的方法 `getId()` 和 `getLabel()`。

基于类似的考虑，我们用接口 `GraphEdge` 表示类 `AbstractGraph` 中存储的图的边。这

个接口声明了方法isDirect()返回是不是有向边、getLabel()返回边的标签，以及返回边的起始顶点和终止顶点的方法getStartNode()和getEndNode()，后面这两个方法的返回类型都是GraphNode。

类AbstractGraph的数据成员有nodes和edges，分别是接口GraphNode和GraphEdge的对象实例的列表。这个类提供返回和设置这两个数据成员的方法，因此类的使用者可直接设置AbstractGraph的对象实例的所有顶点和所有边。此外，这个类还提供添加顶点和添加边的方法addNode()和addEdge()，并提供方法adjacentNodes()返回与一个顶点相邻的所有顶点。

类DefaultGraph继承类AbstractGraph，主要用于求解离散数学基础课程中与图相关的问题，其数据成员主要是继承的nodes和edges，分别以类 DefaultGraphNode和类DefaultGraphEdge的对象实例作为顶点列表和边列表的元素。类DefaultGraphNode的数据成员有id和label，分别记录顶点的标识和标签，用于实现接口GraphNode规定的方法。类DefaultGraphEdge的数据成员有边的标签label、边的起始顶点start、终止顶点end和边是否为有向边的标志 directed，用于实现接口GraphEdge规定的方法。

类DefaultGraph的方法getDFSNodeList()和getBFSNodeList()分别完成对图顶点的深度优先遍历（搜索）(Depth First Search, DFS)和广度优先遍历（搜索）(Breadth First Search, BFS)，返回按照遍历顺序构建的顶点列表。主教材给出了这两个算法的描述，这里不再重复给出。为展示遍历的过程，设计一个内部静态类TravelStepRecord记录遍历过程中的信息。类TravelStepRecord可看作一个结构体，它的数据成员包括：step，记录这是遍历的第几步；visitedNodeList，记录到这一步时已经遍历的顶点列表（按遍历顺序）；auxNodeList，记录这一步时先深遍历时栈或先广遍历时的队列中的顶点列表。类DefaultGraph的方法getDFSNodeList()和getBFSNodeList()在遍历图顶点的同时创建类TravelStepRecord的对象实例记录遍历过程中每一步的已遍历的顶点列表和当前栈或队列的情况，并将这些对象实例保存到类DefaultGraph的静态数据成员 travelStepRecords中。通过这个静态数据成员可展示最近一次遍历的执行过程。

在图论中，无向树是连通且无回路的无向图。判断一个图是不是无向树，以及得到一个无向图的生成树是图相关的重要操作，后者也很容易扩充到计算带权图的最小生成树。为得到一个无向图的生成树，我们设计类GraphComponent，它表示一个图的一些顶点构成的连通分支。对于一个用类DefaultGraph的对象实例表示的无向图，使用类GraphComponent的对象实例列表componentList维持该无向图的连通分支。将componentList初始化为空表，然后通过逐一考虑图的每条边对图的连通分支的影响可判断无向图是不是无向树，和得到无向图的生成树。具体来说，对于图的每条边$e=(u,v)$。

（1）如果顶点u和v都不在componentList的任意一个GraphComponent对象实例中，即还不在任意连通分支中，则创建一个只包含这两个顶点的新的GraphComponent对象实例，并添加到componentList中。

（2）如果只有顶点u或v在componentList的一个GraphComponent对象实例表示的连通分支C中，另一个顶点不在任意一个连通分支中，则将这另一个顶点也添加到C中。

（3）如果顶点u在componentList的一个GraphComponent对象实例表示的连通分支

C_1 中，而顶点 v 在另一个连通分支 C_2 中，则将 C_1 和 C_2 中的顶点合并为一个连通分支 C，并在 componentList 中删除连通分支 C_1 和 C_2，而将 C 添加到 componentList 中。

（4）如果顶点 u 和 v 已经在 componentList 的同一个 GraphComponent 对象实例表示的连通分支 C 中，即表示边 e 与构造连通分支 C 时已经考虑过的边构成了回路，这时就可得到整个无向图不是无向树，或者在要得到这个无向图的生成树时，忽略这个边 e 即可（记录前面 (1)(2)(3) 这三种情况考虑过的边则可得到无向图的生成树）。

学过数据结构的同学可知，这就是判断图连通性的查找-合并 (Find-Union) 算法。如果所有的边都考查完之后，列表 componentList 会给出这个无向图的每个连通分支的顶点集。如果列表 componentList 中只有一个元素，那么这个无向图就是连通图。所以，当一个无向图不是连通图时，通过上述方法得到的实际上是每个连通分支的生成树构成的森林。另外可看到，对于带权图，基于边的权从小到大，按照上述方式考查每条边就是计算带权图最小生成树的 Kruskal 算法的一种具体实现方式。

我们在类 DefaultGraph 中设计私有方法 addEdgeToComponentList() 实现考查一条边对图的连通分支的影响，这种考查可理解为将这条边加入已有的连通分支列表，虽然类 GraphComponent 没有实际存储边的信息（因为可从原来的图中获得边信息）。基于该私有方法，类 DefaultGraphd 的公有方法 isUndirectedTree() 可判断图是不是无向树，以及方法 getUndirectedSpanningForest() 可得到当前无向图的生成树（森林）。

为支持图、树、带权图的随机生成，我们设计工具类 GraphUtil，该类仅提供一些静态方法。利用类 AbstractGraph 提供的设置顶点和边的方法，随机生成图、树、带权图，并提供一些构造特殊图（如完全图、圈图、轮图、超立方体图等）的静态方法。另外，用于分析用户输入的图和带权图的字符串是否符合预定格式，并从中提取图的顶点和边信息的方法也在这个工具类中实现。

图5.7给出了展示图遍历算法相关的类设计。这里需要补充的是，类 DefaultGraph 的方法 addEdgeToComponentList() 在考查一条边后返回这条边所在的连通分支，可利用所返回的连通分支类型明确这条边对连通分支列表产生的影响。类 GraphComponent 定义 4 个整数型常量表示 4 种不同的连通分支类型，具体如下。

（1）类型 NEW_COMPONENT，表示这条边的两个端点都不在已有的连通分支中，所以这条边的两个端点放在了一个新创建的连通分支中。

（2）类型 NORMAL_COMPONENT，表示这是一个普通的连通分支，表示这条边的一个端点已经在这个返回的连通分支中，这次只是将另一个顶点也加入这个连通分支中。

（3）类型 COMBINED_COMPONENT，表示这条边的两个端点在两个不同的已有连通分支，返回的连通分支是这两个连通分支合并的结果。

（4）类型 CIRCLED_COMPONENT，表示这条边的两个端点都在这个连通分支中，这条边的加入使得这个连通分支会有回路。因此，如果在考查一条边后返回的连通分支类型是这个类型，则意味着这个图含回路。如果要在判断这条边后进一步判断其他边是否也会得到回路，那么需要将这个连通分支的类型改回为 NORMAL_COMPONENT。

最后，用例“图的遍历”的边界类是 GraphTravelUIManager，它利用类 GraphUtil 提供的方法分析用户的输入，提取图的顶点和边的信息，然后利用类 DefaultGraph 的方法遍历

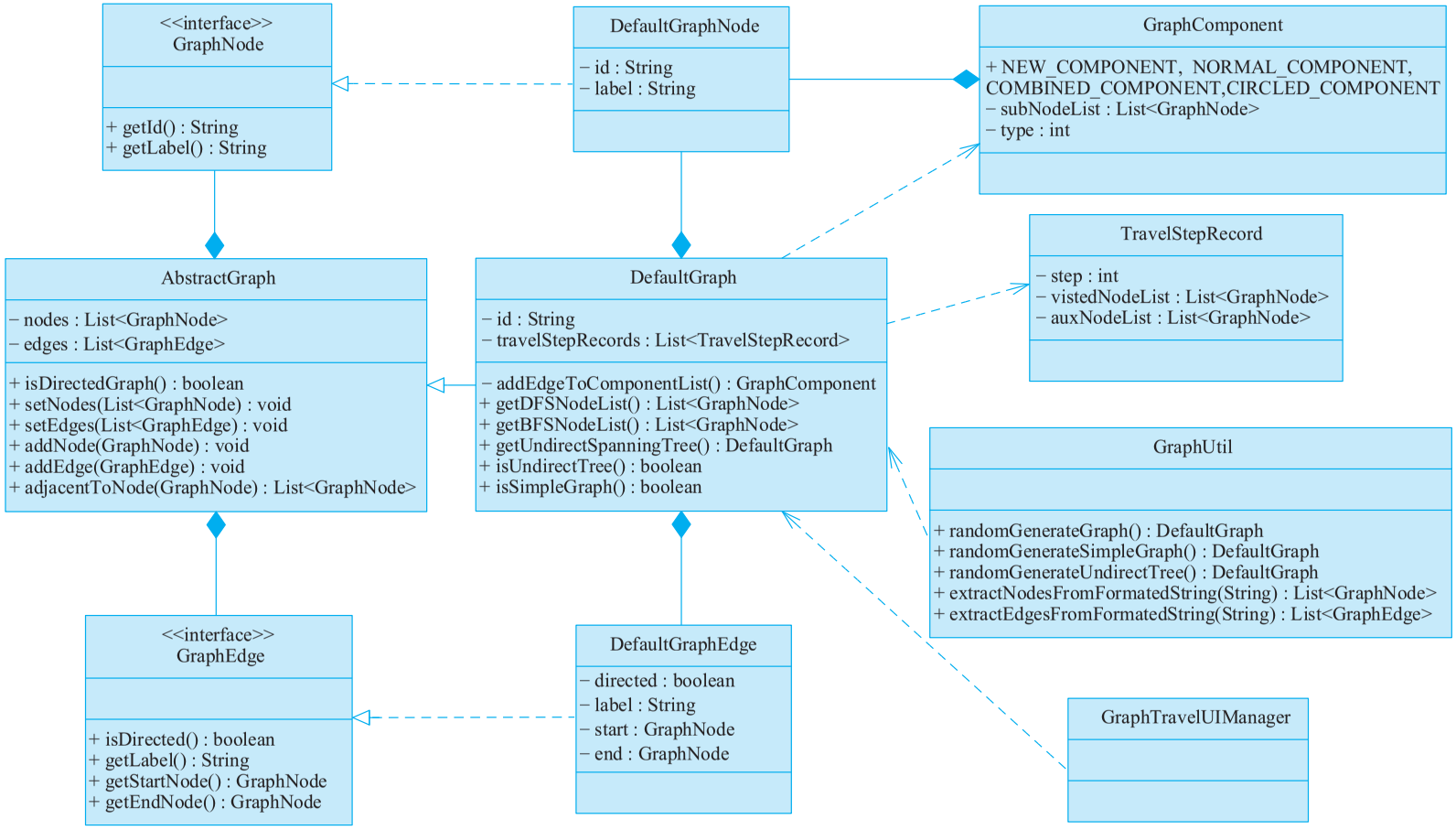

图 5.7　展示图遍历算法的类设计

图的顶点，获得遍历算法的执行过程和执行结果信息，并在软件的输出屏幕区域进行展示。

5.2.2 展示树遍历算法的类设计

展示树遍历算法的核心实体是树，严格来说是根树，而不是无向树。目前在软件中没有设计相应的类表示无向树，而是将无向树用类DefaultGraph存储，并在这个类中提供了判断一个无向图是否为无向树的方法。不过对于根树，我们设计了类RootedTree表示根树，它的主要数据成员是root，给出根树的根。root是类RootedTreeNode的对象实例。虽然从领域实体的角度看，根树也是图，但为了方便展示根树的结构，类RootedTree不再存储树的所有顶点和所有边构成的列表，因此它也不继承类AbstractGraph，而是通过数据成员root，按照根树的结构访问它的所有顶点。类RootedTreeNode表示根树的顶点，它的数据成员中包括这个顶点的儿子顶点构成的列表childList，这个列表的类型是List<RootedTreeNode>，其中的每个元素也是类RootedTreeNode的实例，从而体现根树的结构。

类RootedTree主要实现根树的遍历，其方法getInorderNodeList()、getPreorderNodeList()和getPostorderNodeList()分别给出根树的中序、前序和后序遍历的顶点序列。这些方法目前只给出遍历的结果，没有展示遍历过程中的信息，主教材给出了这些遍历算法的描述。

除实现根树的遍历外，类RootedTree还提供方法convertToDefaultGraph()将其对象实例转换为类DefaultGraph的实例，从而在需要时也可将根树作为有向图看待。这个方法以先广遍历的形式遍历根树，将遍历到的根树顶点以及它的儿子顶点列表childList中的顶点加到类DefaultGraph的对象实例的顶点列表，并为这个顶点以及它的每个儿子顶点构造类DefaultGraphEdge的实例，加到类DefaultGraph的对象实例的边列表，然后递归地对它的每个儿子顶点列表做同样的转换。这种转换直接将根树顶点作为DefaultGraph的顶点列表，因此表示根树顶点的类RootedTreeNode也实现接口GraphNode。

类RootedTree也提供静态方法getAnRootedTree()将一个类DefaultGraph的对象实例转换为根树，这时要指定一个顶点作为根，从这个顶点开始广度优先遍历这个类DefaultGraph的对象实例所表示的图，即将这个根顶点的相邻顶点作为它的儿子顶点。只要在访问相邻顶点时不考虑已经添加到根树中的顶点，那么就得到一棵根树。如果原来的图就是无向树，那么这个方法就将一个类DefaultGraph的对象实例转换为类RootedTree的实例。

图5.8给出了展示树遍历算法相关的类设计，其中类TreeTravelUIManager是用例“树的遍历”的边界类，它利用类GraphUtil提供的方法分析用户的输入，提取顶点和边的信息，创建DefaultGraph的对象实例，在判断其是树的情况下，根据用户指定的根顶点，将该实例转换为类RootedTree的对象实例，并利用类RootedTree的方法遍历树，将遍历结果在软件的输出屏幕区域进行展示。

5.2.3 展示带权图算法的类设计

由于最短路径和最小生成树都是针对带权图进行计算，所以这一小节围绕带权图讨论最短路径和最小生成树算法的展示。这里所说的带权图仅仅是指图的边赋予了一个实数数

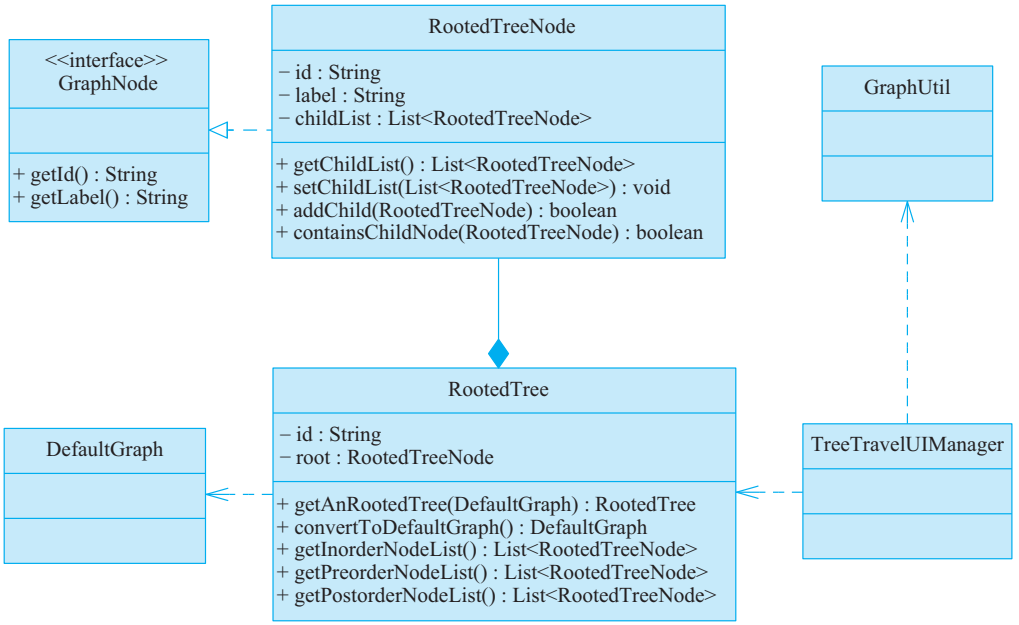

图 5.8 展示树遍历算法的类设计

值作为边的权，因此用类 WeightedGraphEdge 表示带权的边，它继承类 DefaultGraphEdge，除继承自后者的数据成员外，还有双精度 (double) 类型的数据成员 weight 表示边的权，以及设置和访问边的权的方法。

展示带权图算法的核心类是 WeightedGraph，它继承类 DefaultGraph，其顶点列表 nodes 的元素是类 DefaultGraphNode 的对象实例，边列表 edges 的元素是 WeightedGraphEdge 的对象实例。

类 WeightedGraph 的方法 dijkstra() 使用 Dijkstra 算法计算给定起始顶点到其他顶点的最短距离。我们知道，Dijkstra 算法主循环中的关键信息包括尚未计算最短距离的顶点集合、起始顶点到每个顶点的当前最短距离，起始顶点到每个顶点的当前最短路径的直接前驱以及这一次选中的顶点。为展示 Dijkstra 算法的执行过程，需要展示主循环每执行一次这些关键信息的变化。我们设计类 DijkstraRecorder 记录 Dijkstra 算法执行过程这些关键信息的列表，它的数据成员 nodeArray 用数组的形式存储带权图的顶点，使用数组可更容易定位每个顶点。类 DijsktraRecorder 的数据成员 stepList 是类 DisjkstraStepRecorder 的对象实例的列表，每个元素对应 Dijkstra 算法主循环执行一次时的关键信息，即类 DisjkstraStepRecorder 封装算法主循环执行过程中的关键信息。

具体来说，类 DijkstraStepRecorder 的数据成员 selectIndex 表示当前这一步（即这一次循环）选中的顶点在 nodeArray 的下标，而布尔型数组 selectedArray 记录到当前这一步哪些顶点已经得到了最短距离，即若 selectedArray[i] 的值为 true，表明顶点 ndoeArray[i] 的最短距离已经计算好，相应地，双精度型数组 distanceArray 记录对应顶点的当前最短距离，而整数型数组 lastNodeIndexArray 记录对应顶点的当前最短路径中直接前驱的下标。

这样，在根据主教材给出的Dijkstra算法实现方法dijkstra()时，在主循环执行之前创建类DijkstraRecorder的对象实例，然后在主循环体中创建类DijkstraStepRecorder的对象实例保存循环中的关键信息，并将其添加到类DijkstraRecorder的对象实例的数据成员stepList中就可完成算法执行过程的记录。为表示Dijkstra算法的计算结果，设计类GraphPath表示图的路径，它的数据成员edges是边的列表，即其类型是List<GraphEdge>。这个类提供添加边到路径的前端的方法addEdgeFirst()、添加边到路径的后端的方法addEdgeLast()，以及返回路径中的所有顶点的方法getNodes()。进一步，我们用类WeightedGraphPath表示带权图的路径，继承类GraphPath，并用数据成员weight表示带权路径的总权。这个类在添加带权边的同时计算路径的总权。因此，方法dijkstra()的返回类型是List<WeightedGraphPath>，而它记录的算法执行过程则保存在类WeightedGraph的静态数据成员dijkstraRecorder中。

类WeightedGraph的方法kruskal()使用Kruskal算法计算带权图的一棵最小生成树。利用类DefaultGraph的方法addEdgeToComponentList()，按照边权从小到大逐一考查每条边加入连通分支列表所造成的影响，就可按照Kruskal算法得到带权图的一棵最小生成树。Kruskal算法主循环中的关键信息是当前这一步考虑的边，以及这条边是否会与已选择的边构成回路，如果不构成回路就选择，否则就不会选择。因此，我们设计类KruskalRecorder记录这些关键信息的列表，它的数据成员stepList是类KruskalStepRecorder的对象实例的列表，每个元素对应算法主循环执行一次时的关键信息。

具体而言，类KrusklaStepRecorder的数据成员edge记录这一步考虑的边，布尔型数据selected表明这条边是否被选中。方法kruskal()记录的算法执行过程信息保存在类WeightedGraph的静态数据成员kruskalRecorder中，这是类KruskalRecorder的对象实例。方法kruskal()的返回类型是WeightedGraph，即用类WeightedGraph的对象实例表示一棵最小生成树，因为在利用GraphViz展示得到的最小生成树时，只需要将其顶点与边之间的关联展示出来，不需要利用其树状结构，所以不需要特别的类表示无向树和带权无向树。

类WeightedGraph的方法prim()使用Prim算法计算带权图的一棵最小生成树。Prim算法主循环中的关键信息是当前这一步已经在树中的顶点集、尚未在树中的顶点集、当前所考虑的边，即端点分别在上述两个顶点集的边，这一步选中的边，即当前所考虑的边中权最小的边。我们设计类PrimRecorder记录这些关键信息的列表，它的数据成员stepList是类PrimStepRecorder的对象实例的列表，每个元素对应算法主循环执行一次时的关键信息。

具体而言，类PrimStepRecorder的数据成员treeNodeList记录这一步已经在树中的顶点列表，otherNodeList记录尚未在树中的顶点列表，considerEdgeList记录这一步考虑的所有边，selectedEdge记录选中的边，selectedNode记录选中的边的尚未在树中的顶点。方法prim()记录的执行过程信息存储于类WeightedGraph的静态数据成员primRecorder中，这是类PrimRecorder的对象实例。方法prim()的返回类型也是WeightedGraph。

图5.9给出了展示带权图算法的类设计。类ShortestPathUIManager是用例“带权图最

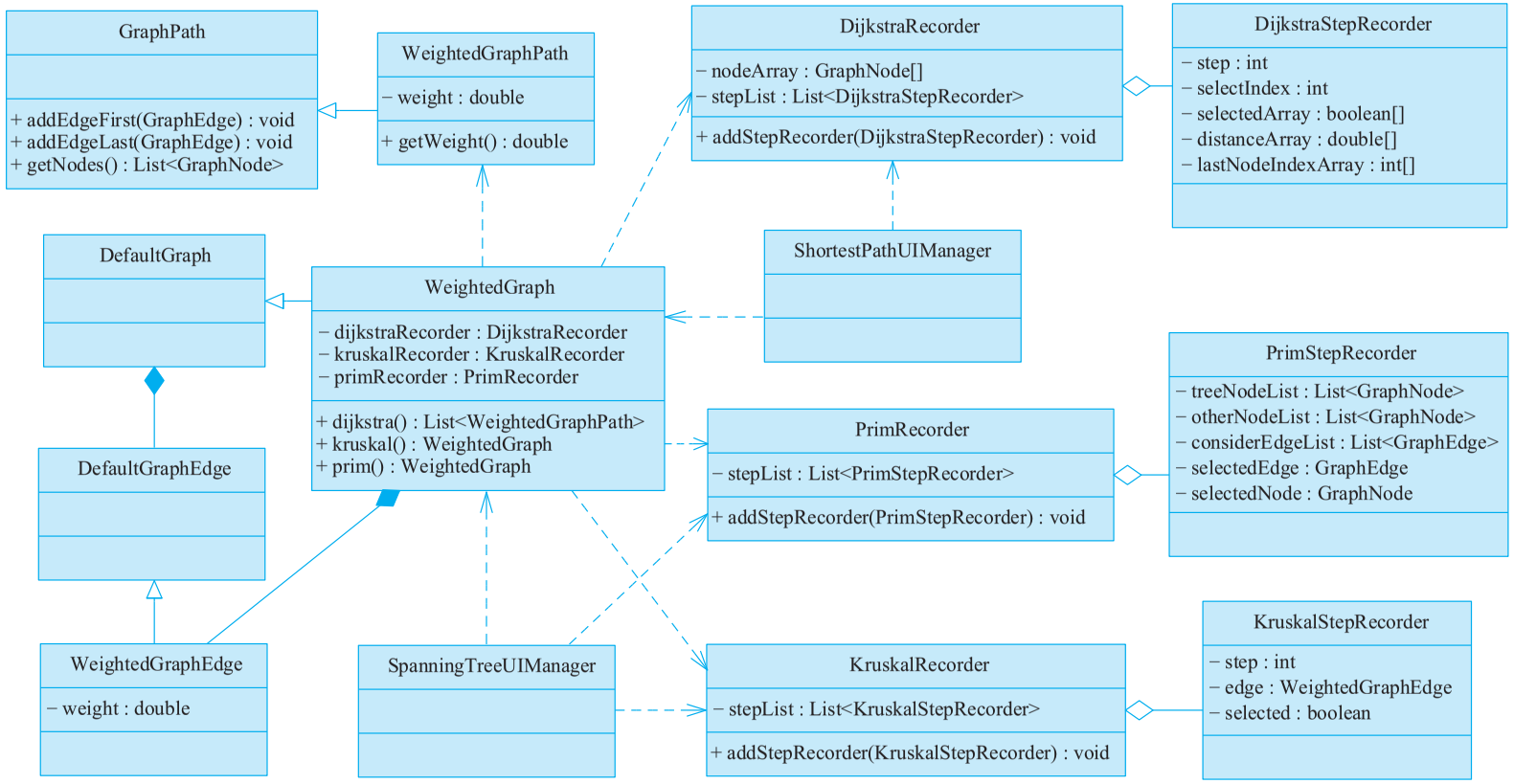

图 5.9　展示带权图算法的类设计

短路径计算”的边界类。它利用类GraphUtil提供的方法获取用户输入的带权图顶点和边的信息，创建类WeightedGraph的对象实例，调用其方法dijkstra()计算带权图的最短路径，获得计算结果，并利用类WeightedGraph的静态数据成员dijkstraRecorder保存的信息展示Dijkstra算法的计算过程和计算结果。

用例“带权图最小生成树计算”的边界类是SpanningTreeUIManager。这个类同样利用类GraphUitl提供的方法获取用户输入的带权图顶点和边的信息，创建类WeightedGraph的对象实例，然后根据用户的选择调用方法kruskal()和prim()计算带权图的最小生成树，获得计算结果，并利用类WeightedGraph的静态数据成员kruskalRecorder或primRecorder保存的信息展示Kruskal算法和Prim算法的计算过程和计算结果。

5.2.4 展示哈夫曼树构造的类设计

类HuffmanTree用于构造哈夫曼树，并展示构造的过程，它是类RootedTree的子类，其主要数据成员是继承自类RootedTree的root，表示树的根。不过哈夫曼树的顶点都带有权，因此用类WeightedTreeNode表示哈夫曼树的顶点。这个类是RootedTreeNode的子类，其数据成员除继承的、存储儿子顶点列表的childList外，还有保存当前顶点权的双精度型数据weight。因此类HuffmanTree的数据成员root指向的是类WeightedTreeNode的对象实例，表示哈夫曼树的根。

类HuffmanTree的对象实例表示一棵构造好的哈夫曼树，所以构造哈夫曼树这个职责不能由类HuffmanTree的非静态方法实现，Java类的非静态方法只能由这个类的对象实例本身调用，不可能在构造这个类的对象实例之前调用。设置另外一个类完成哈夫曼树的构造也是一个好的设计，不过目前我们为简便起见，将哈夫曼树的构造由类HuffmanTree的静态方法createHuffmanTree()完成，这个静态方法按照主教材给出的算法完成哈夫曼树构造，并返回类HuffmanTree的对象实例。

类HuffmanTree的静态数据成员huffmanRecorder用于展示哈夫曼树构造过程。这个数据成员是类HuffmanRecorder的对象实例，它维持列表stepList，列表的元素是类HuffmanStepRecorder的对象实例，每个对象实例保存哈夫曼树构造算法的主循环循环一次得到的中间结果。我们知道，哈夫曼树构造算法主循环执行过程中的中间结果可看作一个森林，森林中每棵树的根按权从小到大排序，然后下一次循环选权最小的两个根，并构造一个新的顶点，以这两个根作为儿子，从而得到一棵新的树。因此类HuffmanStepRecorder的数据成员是数组nodeArray，该数组的每个元素是类WeightedTreeNode的对象实例，表示森林中每棵树的根。方法createHuffmanTree()在构造哈夫曼树的同时，将每次循环得到的森林的每棵树的根用一个类HuffmanStepRecorder的对象实例保存，并添加到huffmanRecorder的数据成员stepList中，从而保存哈夫曼树构造过程信息。

用例“哈夫曼树的构造”需要展示构造得到的哈夫曼树的总权重，以及对叶子的二进制编码结果。类HuffmanTree的方法getTotalWeight()返回构造得到的哈夫曼树的总权重。总权重的计算可使用递归算法，如算法5.1所示。方法getTotalWeight()以构造得到

的哈夫曼树的根root和层次0调用该算法则可得到整棵哈夫曼树的总权重。

算法 5.1：计算哈夫曼树的一棵子树的总权重

```
输入：子树的根subroot，子树根所在层次level
输出：该子树的总权重，即其叶子顶点的权乘以叶子顶点所在层次的总和
if (subroot 已经是叶子顶点) then
    返回subroot的权乘level的值;
else
    令双精度数result为0;
    foreach (subroot的每个儿子child) do
        将以child和level+1为参数递归调用本算法的结果加到result;
    end
    返回result作为以subroot为根的子树的总权重;
end
```

类HuffmanTree的方法getCodeOfLeafNodes()可得到叶子的二进制编码结果，其核心算法仍是递归算法，如算法5.2所示。方法getCodeOfLeafNodes()以构造得到的哈夫曼树的根root和空串为编码前缀串调用算法5.2则可得到哈夫曼树中所有叶子顶点的二进制编码。

算法 5.2：对哈夫曼树的一棵子树的叶子顶点进行二进制编码

```
输入：子树的根subroot、编码前缀串prefix
输出：该子树的叶子顶点编码结果，存储在字符串数组codeArray
if (subroot 已经是叶子顶点) then
    将这时的编码前缀串prefix直接作为subroot的编码添加到codeArray;
else
    设subroot的左儿子为left，右儿子为right;
    以left和前缀串prefix+"0"为参数递归调用本算法编码左儿子left为根的子树的叶子顶点;
    以right和前缀串prefix+"1"为参数递归调用本算法编码右儿子right为根的子树的叶子顶点;
end
```

图5.10给出了展示哈夫曼构造的类设计。为了利用GraphViz将哈夫曼树构造过程中的森林绘制出来，我们将类HuffmanStepRecorder中保存的根顶点数组转换为类RootedForest的对象实例，并进一步转换为类DefaultGraph的对象实例，从而可将图写成DOT文件，调用GraphViz提供的方法生成图片在输出屏幕展示。

用例"哈夫曼树的构造"的边界类是HuffmanTreeUIManager，这个类利用类HuffmanTree提供的静态方法extractWeightedLeafsFromFormatedString()获取用户按照预先给定格式输入的叶子顶点及其权的信息，调用方法createHuffmanTree()构造哈夫曼树，并利用类HuffmanTree的静态数据成员huffmanRecorder保存的信息展示哈夫曼树构造的过程，并展示哈夫曼树的总权和叶子顶点的编码信息。

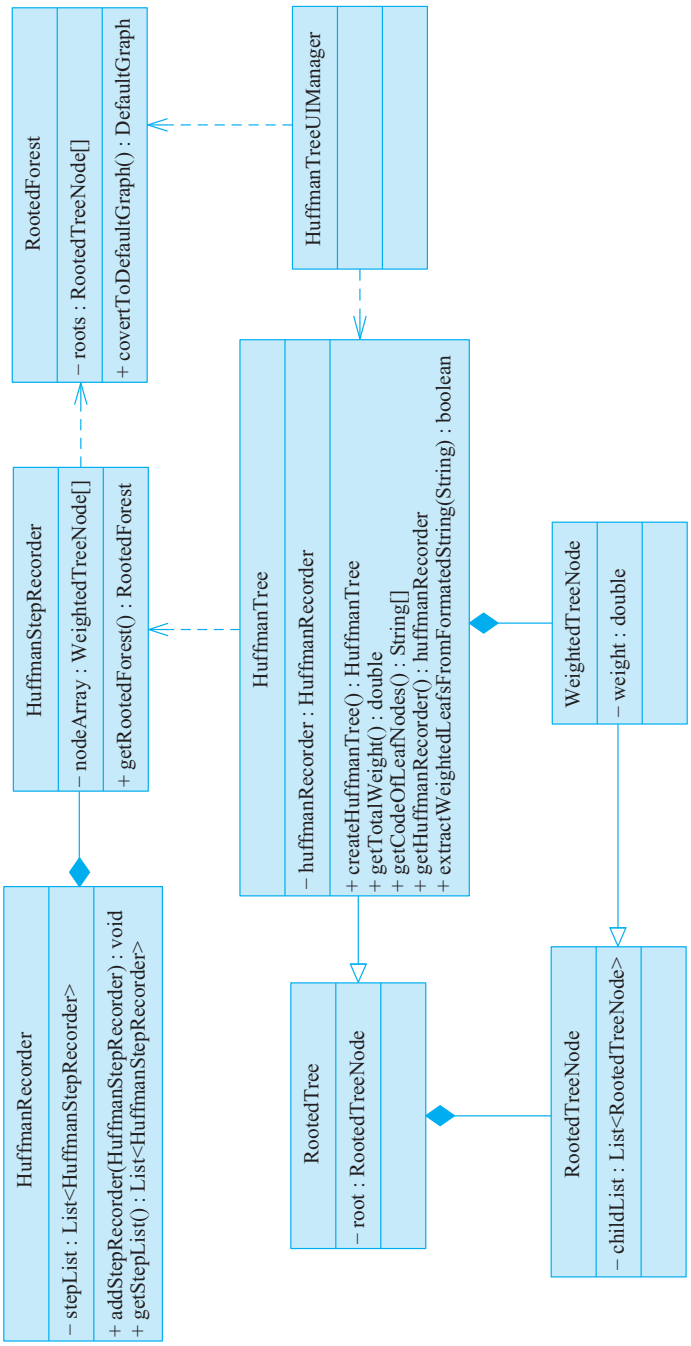

图 5.10 展示哈夫曼树构造算法的类设计

5.3 图和树问题求解的实现

本节给出图和树问题求解的实现，主要是给出表示图、根树、带权图、哈夫曼树的类的主要方法，以及图遍历、树遍历、带权图最短路径和最小生成树，以及哈夫曼树构造算法的实现细节，并给出展示这些算法的输入界面和计算结果，从而让读者对如何实现这些算法，以及如何展示算法的执行过程有更好的了解。

5.3.1 图遍历算法展示的实现

根据上面的设计，类AbstractGraph表示抽象意义上的图，完成图论中图的最基本职责，即维持图中边与顶点之间的关联关系。接口GraphNode和接口GraphEdge分别表示抽象图中的顶点和边。表5.5给出了类AbstractGraph的方法的描述，这些方法主要是维持这个类的两个数据成员，即顶点列表nodes和边列表edges，因此这些方法的实现都比较简单。

表 5.5 类AbstractGraph主要方法的描述

addEdge(GraphEdge):void 往图中添加边
addNode(GraphNode):void 往图中添加顶点
getEdges():List<GraphEdge> 返回图的边列表
getNodes():List<GraphNode> 返回图的顶点列表
hasEdge(GraphEdge):boolean 检查图是否有给定的边
hasEdge(GraphNode, GraphNode):boolean 检查图是否有关联给定两个顶点的边
hasNode(GraphNode):boolean 检查图是否有给定的顶点
isDirectedGraph():boolean 检查图是不是有向图，即是否图的每条边都是有向边
isUndirectedGraph():boolean 检查图是不是无向图，即是否图的每条边都是无向边
setEdges(List<GraphEdge>):void 将图的边列表设定为给定的列表
setNodes(List<GraphNode>):void 将图的顶点列表设定为给定的列表
adjacentNodes(GraphNode):List<GraphNode> 返回与给定顶点相邻的所有顶点构成的列表
adjacentFromNode(GraphNode):List<GraphNode> 返回以邻接自给定顶点的顶点列表，即给定顶点有向边指向该列表中的每个顶点
adjacentToNode(GraphNode):List<GraphNode> 返回以邻接到给定顶点的顶点列表，即该列表中的每个顶点有向边指向该给定顶点
simplyWriteToDotFile(PrintWriter):void 将图的顶点和边的信息写到一个DOT文件，以便使用GraphViz绘制这个图的图形化表示

真正实现图遍历算法的是类DefaultGraph，它继承类AbstractGraph。类DefaultGraphNode和类DefaultGraphEdge分别实现接口GraphNode和GraphEdge，其对象实例作为类DefaultGraph中顶点列表和边列表中的元素。表5.6给出了类DefaultGraph的方法的描述。

表 5.6 类DefaultGraph主要方法的描述

createBFSStepRecord(int, List<GraphNode>, LinkedList<Integer>):TravelStepRecord, 私有方法 创建记录先广遍历某一步的信息的类TravelStepRecord的对象实例，第一个参数是步数，第二个参数是当前步已经遍历的顶点列表，第三个参数是当前在队列中顶点的下标列表。遍历前会将图的顶点列表转换为一个顶点数组
createDFSStepRecord(int, List<GraphNode>, Stack<Integer>):TravelStepRecord, 私有方法 创建记录先深遍历某一步的信息的类TravelStepRecord的对象实例，第一个参数是步数，第二个参数是当前步已经遍历的顶点列表，第三个参数是当前在栈中顶点的下标列表
addEdgeToComponentList(List<GraphComponent>, GraphEdge):GraphComponent, 私有静态方法 添加一条边，或者说考查所要添加的边对图的当前连通分支的影响，返回这条边所影响的连通分支
getTravelStepRecords():List<TravelStepRecord>, 静态方法 返回记录图遍历算法执行过程信息的列表，即返回静态数据成员travelStepRecords的值
calculatePathNumberByUsingAdjacencyMatrix():Matrix[] 利用图的邻接矩阵计算图的路径条数，计算的结果放在一个矩阵中
getAdjacencyMatrix():Matrix; getIncidenceMatrix():Matrix 分别返回图的邻接矩阵和关联矩阵
getBFSNodeList():List<GraphNode> 返回图的一个先广遍历序列，同时记录遍历的过程信息。在此方法后立即调用静态方法getTravelStepRecords()可得到此过程信息
getDFSNodeList():List<GraphNode> 返回图的一个先深遍历序列，同时记录遍历的过程信息。在此方法后立即调用静态方法getTravelStepRecords()可得到此过程信息
getDegree(GraphNode):int; getInDegree(GraphNode):int; getOutDegree(GraphNode):int 分别计算并返回给定顶点的度数、入度和出度
getUndirectSpanningForest():DefaultGraph 将图作为无向图，计算它的生成树，如果有多个连通分支，则每个连通分支得到一个生成树，从而得到的是森林。计算的结果用类DefaultGraph的对象实例表示
isSimpleGraph():boolean 判断图是不是简单图，即判断图是否存在环和重边
isUndirectTree():boolean 判断图是不是无向树，即将图看作无向图，检查它是否连通且没有回路

类DefaultGraph的方法getDFSNodeList()实现图的先深遍历，并将遍历的过程信息记录到类DefaultGraph的静态数据成员travelStepRecords。程序5.1给出了这个方法的源代码，其中第7行开始的循环以顶点列表nodes中下标为startIndex的顶点开始进行遍历，遍历这个顶点所在的连通分支。第8、9行将该顶点添加到遍历结果列表result，并将标志数组visited的相应位置元素置为true，表示该顶点已经遍历。第10行将该顶点添加到栈中。第11、12行创建有关这一步遍历的过程信息，并添加到静态数据成员

travelStepRecords中。

程序 5.1 类DefaultGraph的方法getDFSNodeList()的实现

```
1  public List<GraphNode> getDFSNodeList() {
2      // 初始化记录已遍历的顶点列表result、图的邻接矩阵amatrix、相应顶点是否已遍历的
       // 布尔数组visited
3      // 以及初始化存储顶点下标的栈stack，并初始化静态数据成员travelStepRecords
4      // 变量startIndex表示遍历的起始顶点的下标，初始化为0，即从顶点列表nodes的第一个
       // 顶点开始遍历
5      // step记录步数，初始化为1
6      ... // 此处省略这些完成初始化功能的程序代码
7      while (startIndex < nodes.size()) {
8          GraphNode startNode = nodes.get(startIndex);
9          visited[startIndex] = true; result.add(startNode);
10         stack.push(new Integer(startIndex)); startIndex = startIndex+1;
11         TravelStepRecord record = createDFSStepRecord(step, result, stack);
12         travelStepRecords.add(record); step = step + 1;
13         while (!stack.isEmpty()) {
14             int currentNodeIndex = stack.getTop().intValue();
15             int nextNodeIndex = -1;
16             for (int j = 0; j < nodes.size(); j++) {
17                 if (amatrix.get(currentNodeIndex,j)>0 && visited[j]==false) {
18                     nextNodeIndex = j; break;
19                 }
20             }
21             if (nextNodeIndex >= 0) {
22                 GraphNode nextNode = nodes.get(nextNodeIndex);
23                 visited[nextNodeIndex] = true; result.add(nextNode);
24                 stack.push(new Integer(nextNodeIndex));
25             } else stack.pop();
26             record = createDFSStepRecord(step, result, stack);
27             travelStepRecords.add(record); step = step + 1;
28         }
29         while (startIndex < nodes.size()) {
30             if (visited[startIndex] == false) break;
31             startIndex++;
32         }
33     }
34     return result;
35 }
```

程序5.1中第13行开始的循环以深度优先方式遍历 startIndex这个顶点所在的连通分支。第14行得到栈顶顶点的下标，第16~20行的循环利用邻接矩阵amatrix得到该栈顶顶点的一个尚未遍历的邻接顶点，如果这样的顶点存在，则在第22~24行遍历该邻接顶点，并将其压入堆栈，否则在第25行将栈顶顶点弹出。循环中的第26、27行记录这一步的遍历过程信息。第29行开始的循环是在遍历startIndex所在的连通分支之后，检查是否还有尚未遍历的顶点，如果还有则作为新的起始顶点再进行遍历。

类DefaultGraph的方法getBFSNodeList()实现图的先广遍历，程序5.2给出了这个方

法的源代码，其中第7行开始的循环以顶点列表nodes中下标为startIndex的顶点开始进行遍历，遍历这个顶点所在的连通分支。第8、9行将该顶点添加到遍历结果列表result，并将标志数组 visited的相应位置元素置为true，表示该顶点已经遍历。第10行将该顶点添加到队列的尾部。第11、12行创建有关这一步遍历的过程信息，并添加到静态数据成员travelStepRecords中。

程序5.2中第13行开始的循环以广度优先方式遍历startIndex这个顶点所在的连通分支。第14、15行得到队列头部顶点的下标，并从队列中删除。第16~21行的循环利用邻接矩阵amatrix得到该头部顶点的所有尚未遍历的邻接顶点，将其添加到队列尾部，并标志这些顶点已经被遍历。循环中的第22、23行记录这一步的遍历过程信息。第25行开始的循环是在遍历startIndex所在的连通分支之后，检查是否还有尚未遍历的顶点，如果还有则作为新的起始顶点再进行遍历。

程序 5.2　类DefaultGraph的方法getBFSNodeList()的实现

```
1  public List<GraphNode> getDFSNodeList() {
2      // 初始化记录已遍历的顶点列表result、图的邻接矩阵amatrix、相应顶点是否已遍历的
       // 布尔数组visited
3      // 以及初始化存储顶点下标的队列stack，并初始化静态数据成员travelStepRecords
4      // 变量startIndex表示遍历的起始顶点的下标，初始化为0，即从顶点列表nodes的第一个
       // 顶点开始遍历
5      // step记录步数，初始化为1
6      ... // 此处省略这些完成初始化功能的程序代码
7      while (startIndex < nodes.size()) {
8          GraphNode startNode = nodes.get(startIndex);
9          visited[startIndex] = true; result.add(startNode);
10         queue.addLast(new Integer(startIndex)); startIndex = startIndex+1;
11         TravelStepRecord record = createBFSStepRecord(step, result, queue);
12         travelStepRecords.add(record); step = step + 1;
13         while (!queue.isEmpty()) {
14             int currentNodeIndex = queue.getFirst().intValue();
15             queue.removeFirst();
16             for (int j = 0; j < nodes.size(); j++) {
17                 if (amatrix.get(currentNodeIndex,j)>0 && visited[j]==false) {
18                     GraphNode node = nodes.get(j); visited[j] = true;
19                     result.add(node); queue.addLast(new Integer(j));
20                 }
21             }
22             record = createBFSStepRecord(step, result, queue);
23             travelStepRecords.add(record); step = step + 1;
24         }
25         while (startIndex < nodes.size()) {
26             if (visited[startIndex] == false) break;
27             startIndex++;
28         }
29     }
30     return result;
31 }
```

类DefaultGraph的方法addEdgeToComponentList()基于合并-查找算法考查一条边对图的已有的连通分支的影响，即：①若这条边关联的两个端点都不在已有连通分支中，就创建新的连通分支；②如果只有一个端点在已有连通分支，则将另一端点也添加到该连通分支；③如果两个端点在两个不同连通分支，则合并这两个连通分支；④若两个端点已经在同一个连通分支，则表明这条边与已有的边在该连通分支构成回路。利用这个方法，方法getUndirectSpanningForest()可得到图的生成树（严格地说，对非连通图，得到的是森林）。

程序 5.3 类DefaultGraph的方法getUndirectSpanningForest()的实现

```
1  public DefaultGraph getUndirectSpanningForest () {
2      List<GraphComponent> componentList = new ArrayList<GraphComponent>();
3      List<GraphEdge> selectEdge = new ArrayList<GraphEdge>();
4      for (GraphEdge edge : edges) {
5          if (edge.getStartNode().equals(edge.getEndNode())) continue;
6          GraphComponent component = addEdgeToComponentList(componentList, edge);
7          if (component.getType() != GraphComponent.CIRCLED_COMPONENT) {
8              selectEdge.add(edge);
9          } else component.changeTypeTo(GraphComponent.NORMAL_COMPONENT);
10     }
11     DefaultGraph result = new DefaultGraph("USFof" + id);
12     result.setNodes(nodes); result.setEdges(selectEdge);
13     return result;
14 }
```

程序5.3给出了方法getUndirectSpanningForest()的源代码，其中第4行开始的循环，从一个空的连通分支列表componentList开始逐一考查图的每条边，循环中的第5行忽略图中的环，第6行考查一条边对已有连通分支的影响，只要返回的连通分支的类型不是CIRCLED_COMPONENT，则这条边就不与已选中的边构成回路，就将其添加到选中边（作为生成树的边）的列表selectEdge中。如果返回的连通分支类型是CIRCLED_COMPONENT，则将其类型设置为NORMAL_COMPONENT，以便后续边的考查（因为后续边仍有可能与这个连通分支中的边构成回路）。最后，循环结束后的第11、12行以所考查图的顶点列表为顶点列表，以选中的边列表为边列表，创建表示所考查图的生成森林的类DefaultGraph的对象实例作为方法的返回结果。方法isUndirectTree()也用类似的方式判断图是不是无向树，只要在逐一考查边时遇到返回的连通分支类型是 CIRCLED_COMPONENT，则表示图有回路，而在考查边之后，如果最终的连通分支列表componentList中的元素多于一个，则表示图不是连通图。如果出现这两种情况之一，图就不是无向树，否则就是无向树。

类GraphTravelUIManager是实现用例“图的遍历”的边界类，负责绘制展示图遍历过程的输入界面，如图5.11所示。在这个界面中，用户可按照在用例分析一节所确定的格式输入图的顶点和边的信息。类GraphUtil提供方法分析输入的字符串，提取其中的顶点和边的信息，并创建类DefaultGraph的对象实例表示输入的图。类GraphUtil还提供方法支持图的信息的随机生成。这里使用简单的方法随机生成图，即创建指定个数的顶点，并随机选择两个顶点作为关联的端点而随机得到图的边。

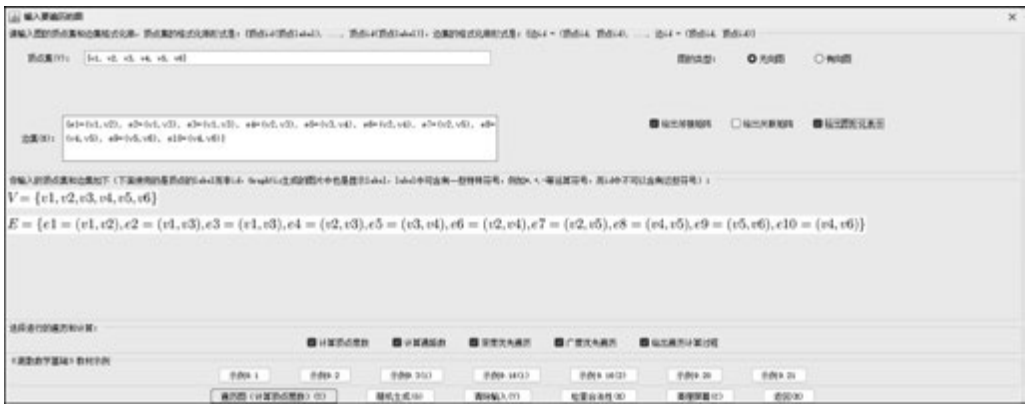

图 5.11　展示图的遍历过程的输入界面

图5.11给出的界面中，用户可选择是否展示图的邻接矩阵和关联矩阵，以及是否给出图形化表示，还可选择计算图的顶点度数、通路条数、遍历图的方法，以及是否展示遍历的过程。在单击“遍历图（计算顶点度数）”按钮之后，类GraphTravelUIManager利用类GraphUtil提供的方法提取信息，得到表示图的类DefaultGraph的对象实例，按照用户指定的方式进行计算和遍历，并在输出屏幕区域展示结果。对于图5.11的输入，将得到图5.12的输出结果。

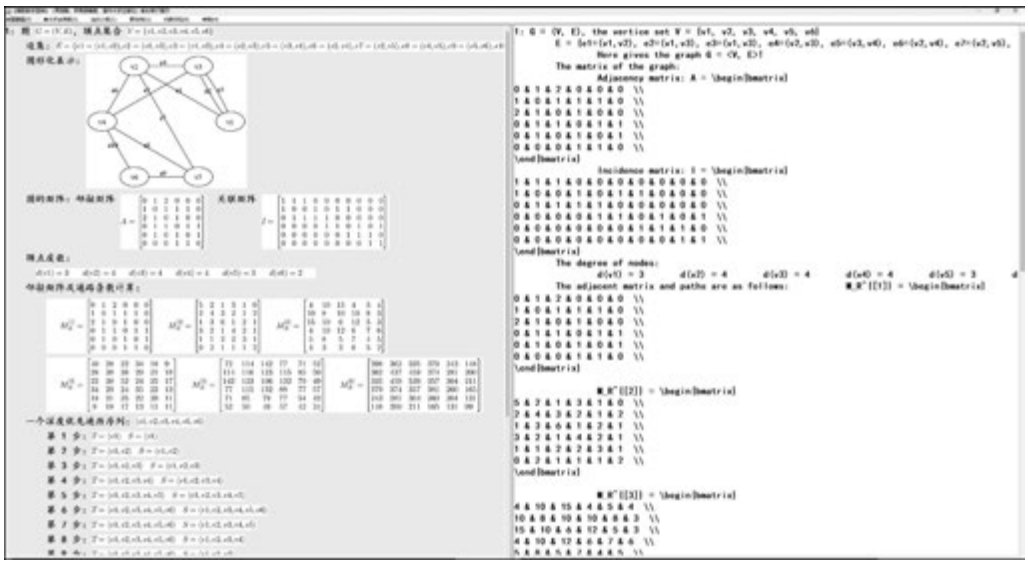

图 5.12　展示图的遍历过程的输出结果

由于给出了图的图形化表示，以及邻接矩阵、顶点度数和通路等信息，因此展示的内容比较多。图5.12只是给出了图的深度优先遍历过程的部分信息。在软件中，通过向下滚动可得到更多信息，包括图深度优先遍历和广度优先遍历的完整过程信息。

5.3.2　树遍历算法展示的实现

展示树遍历算法的核心类是RootedTree，它表示一棵根树，其核心数据成员是表示树

顶点的类RootedTreeNode的对象实例root，给出根树的根。类RootedTreeNode真正体现树状结构，它的数据成员childList给出树顶点的子顶点列表。表5.7给出了类RootedTree的方法描述，其中给出了一些私有方法，这些方法都是递归方法，实现对树结构的递归处理。调用这些私有方法的公有方法则以合适的参数启动这些递归方法，完成类RootedTree的一些职责。

表 5.7 类RootedTree主要方法的描述

getAnRootedTree(DefaultGraph, DefaultGraphNode, boolean):RootedTree, 静态方法 将给定的图，以给定的顶点为根，转换为一棵根树，布尔类型的参数说明给定的图是不是有向图
getAnSubRootedTree(DefaultGraph, RootedTreeNode, DefaultGraphNode, List<DefaultGraphNode>, boolean):void: 静态私有方法 由方法getAnRootedTree()调用的方法，递归地将给定图顶点的相邻顶点进行转换，并将得到的子树作为给定的树顶点的子树
RootedTree(String, RootedTreeNode):Constructor 以给定树顶点为根创建一个对象实例，字符串类型的参数用于在需要时给创建的根树一个名字
convertToDefaultGraph(boolean, boolean):DefaultGraph 将根树转换为类DefaultGraph的对象实例，前一个布尔类型的参数说明是否要转换为有向图，后一个布尔类型的参数指定是否要给转换后得到的图的边设置标签
convertToDefaultGraph(RootedTreeNode, List<GraphNode>, List<GraphEdge>, boolean,boolean):void, 私有方法 被方法convertToDefaultGraph()调用，递归地对以给定树顶点为根的子树进行转换，转换后得到的顶点和边都添加到参数中给定的顶点列表和边列表
getInorderNodeList():List<RootedTreeNode> 返回以中序形式遍历根树的顶点序列
getInorderNodeList(RootedTreeNode):List<RootedTreeNode>, 私有方法 被方法getInorderNodeList()调用，以中序形式递归地遍历以给定树顶点为根的子树
getPostorderNodeList():List<RootedTreeNode> 返回以后序形式遍历根树的顶点序列
getPostorderNodeList(RootedTreeNode):List<RootedTreeNode>, 私有方法 被方法getPostorderNodeList()调用，以后序形式递归地遍历以给定树顶点为根的子树
getPreorderNodeList():List<RootedTreeNode> 返回以前序形式遍历根树的顶点序列
getPreorderNodeList(RootedTreeNode):List<RootedTreeNode>, 私有方法 被方法getPreorderNodeList()调用，以前序形式递归地遍历以给定树顶点为根的子树
getRoot():RootedTreeNode 返回当前根树的根顶点
simplyWriteToDotFile(PrintWriter, boolean, boolean):void 将根的顶点和边的信息写到一个DOT文件，以便使用GraphViz绘制这棵根树的图形化表示

类RootedTree的主要职责是以中序、后序和前序遍历根树，这里以中序遍历的实现为例进行说明。方法getInorderNodeList()返回以中序形式遍历根树的顶点序列，它以根root为参数调用私有方法getInorderNodeList()，后者以中序形式递归地遍历以给定树顶点为根的子树。程序5.4给出这个方法的源代码，其中第4行判断子树的根startNode是不是叶子顶点，如果是则将其添加到结果列表，这相当于遍历该叶子顶点。否则，第7行

获取它的子顶点列表的第一个顶点，即第一个儿子，第8行递归调用本方法，遍历以第一个儿子为根的子树，第9行将遍历的结果添加到结果列表，第10行再添加startNode本身，也即以中序形式遍历该顶点，然后第11行开始的循环遍历startNode的其他儿子为根的子树。

程序 5.4　类RootedTreeNode的私有方法getInorderNodeList()的实现

```
1  private List<RootedTreeNode> getInorderNodeList(RootedTreeNode startNode) {
2      List<RootedTreeNode> result = new ArrayList<RootedTreeNode>();
3      List<RootedTreeNode> childList = startNode.getChildList();
4      if (childList == null) {
5          result.add(startNode); return result;
6      }
7      RootedTreeNode firstChild = childList.get(0);
8      List<RootedTreeNode> childResult = getInorderNodeList(firstChild);
9      result.addAll(childResult);
10     result.add(startNode);
11     for (int i = 1; i < childList.size(); i++) {
12         RootedTreeNode otherChild = childList.get(i);
13         childResult = getInorderNodeList(otherChild);
14         result.addAll(childResult);
15     }
16     return result;
17 }
```

类RootedTree以树结构的形式存储根树，而类DefaultGraph以顶点列表和边列表的形式存储图，这两者之间没有继承关系。类RootedTree提供方法实现这两者的对象实例之间的转换。方法getAnSubRootedTree()从一个图中得到一棵根树，本质上是从给定的一个顶点开始先广遍历这个图，从而得到它的一棵生成树。程序5.5给出了私有方法getAnSubRootedTree()的源代码。这个方法的参数graph给定要得到根树的图，rootNode给定的图顶点作为正在处理的子树的根，参数root与rootNode具有相同的顶点标识和标签，是后者作为根树中顶点的副本。参数addedNodeList是已经有副本在根树中的图顶点，也即这些图顶点已经考虑过，不能再作为后续要考虑的顶点的后代顶点，否则就会构成回路。

程序 5.5　类RootedTreeNode的私有方法getAnSubRootedTree()的实现

```
1  private static void getAnSubRootedTree(DefaultGraph graph,
2          RootedTreeNode root, DefaultGraphNode rootNode,
3          List<DefaultGraphNode> addedNodeList, boolean directedGraph) {
4      List<GraphNode> adjacentNodeList = null;
5      if (directedGraph) adjacentNodeList = graph.adjacentFromNode(rootNode);
6      else adjacentNodeList = graph.adjacentNodes(rootNode);
7      int size = adjacentNodeList.size();
8      if (size == 0) return;
9      DefaultGraphNode[] childGraphNodeList = new DefaultGraphNode[size];
10     RootedTreeNode[] childTreeNodeList = new RootedTreeNode[size];
11     int counter = 0;
12     for (GraphNode adjacentNode : adjacentNodeList) {
13         DefaultGraphNode aNode = (DefaultGraphNode)adjacentNode;
```

```
14          if (!addedNodeList.contains(aNode)) {
15              addedNodeList.add(aNode);
16              RootedTreeNode treeNode =
17                  RootedTreeNode.createRootedTreeNodeFromDefaultGraphNode(aNode);
18              root.addChildNode(treeNode);
19              childGraphNodeList[counter] = aNode;
20              childTreeNodeList[counter] = treeNode;
21              counter = counter+1;
22          }
23      }
24      for (int i = 0; i < counter; i++) {
25          getAnSubRootedTree(graph, childTreeNodeList[i], childGraphNodeList[i],
26              addedNodeList, directedGraph);
27      }
28  }
```

程序5.5的第4~6行获取rootNode的邻接顶点，如果是有向图，则获取rootNode邻接到的顶点。第9、10行创建的图顶点列表和根树顶点列表分别记录新加入的图顶点及其对应在根树中的副本。第12行开始的循环为不在addedNodeList中的图顶点创建相应的根树顶点副本，这些副本都作为root的子顶点。第24~27行的循环递归调用本方法，处理rootNode的新加入addedNodeList的邻接顶点为根的子树。类RootedTree的公有方法getAnSubRootedTree()则为指定为根的图顶点创建一个根树顶点副本，并将这个图顶点作为addedNodeList的唯一元素调用这个私有方法，执行完毕后，最初创建的根树顶点副本就是整棵树的根。显然，若给定的图graph是树，那么调用此方法后，得到的就是用类RootedTree的对象实例表示的这棵树。

类RootedTree的方法convertToDefaultGraph()将根树转换为类DefaultGraph的对象实例。程序5.6给出了私有方法的源代码，其本质是前序遍历根树的顶点，将顶点及其与子顶点间的边添加到图的顶点列表和边列表中。其中，参数root是当前正处理的子树的根，参数nodes和edges是最后得到的图的顶点列表和边列表。第6行开始的循环处理root的每个子顶点，将其添加到顶点列表nodes，并在root和该子顶点之间创建一条边，添加到边列表edges，然后在第14行递归调用本方法，处理以该子顶点为根的子树。公有方法converToDefaultGraph()以根树的根顶点和空的顶点列表和边列表调用此递归版本的私有方法则可将类RootedTree的对象实例转换为类DefaultGraph的对象实例。

程序 5.6　类RootedTreeNode的私有方法convertToDefaultGraph()的实现

```
1   private void convertToDefaultGraph(RootedTreeNode root, List<GraphNode> nodes,
2           List<GraphEdge> edges, boolean directed, boolean labelEdge) {
3       List<RootedTreeNode> childList = root.getChildList();
4       if (childList == null) return;
5       int i = 0;
6       for (RootedTreeNode childNode : childList) {
7           nodes.add(childNode);
8           String label = null;
9           if (labelEdge) label = "" + i;
```

```
10            i = i + 1;
11            DefaultGraphEdge edge = new DefaultGraphEdge(root, childNode,
12                label, directed);
13            edges.add(edge);
14            convertToDefaultGraph(childNode, nodes, edges, directed, labelEdge);
15        }
16    }
```

类TreeTravelUIManager是实现用例“树的遍历”的边界类，负责绘制展示树遍历结果的输入界面，如图5.13所示。在这个界面中，用户输入树信息的方式与输入图信息的方式相同。类GraphUtil也提供方法支持树的信息的随机生成，其生成方式与图生成树的生成方式相同，在创建指定个数的顶点之后，随机生成一些边，并考查这些边添加到图的已有连通分支列表时是否构成回路，如不构成回路，则作为最后生成树的边。

图 5.13　展示树的遍历过程的输入界面

图5.13给出的界面中，用户可选择是否展示树的邻接矩阵和关联矩阵，以及是否给出图形化表示，还可选择遍历树的方法。在单击“遍历树”按钮之后，类TreeTravelUIManager利用类GraphUtil提供的方法提取信息，得到表示图的类DefaultGraph的对象实例，判断它是不是树，如果不是树则会提示输入错误。如果是树则再将其转换为类RootedTree的对象实例，按照用户指定的方式进行遍历，并在输出屏幕区域展示结果。对于图5.13的输入，将得到图5.14的输出结果。由于目前还没有想到合适的方式展示树遍历的过程，因此这里只是简单地给出了根树前序、中序和后序遍历的结果。

5.3.3　带权图最短路径和最小生成树算法展示的实现

展示带权图算法的核心类是WeightedGraph，它继承类DefaultGraph，其顶点列表nodes的元素是类DefaultGraphNode的对象实例，边列表edges的元素是WeightedGraphEdge的对象实例。除继承自类DefaultGraph和类AbstractGraph的数据成员外，这个类还定义了三个静态数据成员，分别用于记录Dijkstra算法、Kruskal算法和Prim算法的运行过程信息。

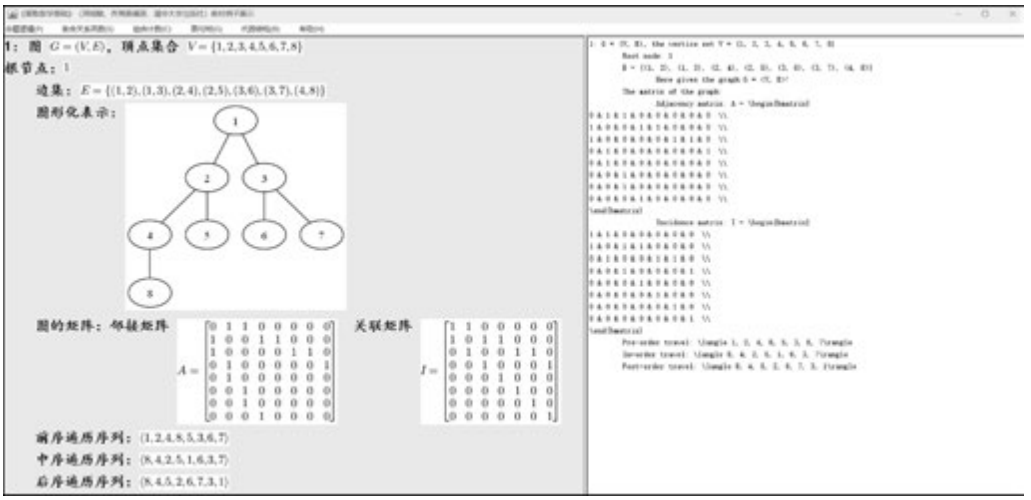

图 5.14 展示树的遍历过程的输出结果

表5.8给出了类WeightedGraph的主要方法的描述。方法dijkstra()使用Dijkstra算法计算带权图中给定起始顶点到其他顶点的最短路径，程序5.7给出了这个方法的核心源代码，其中主要使用了如下4个数组。

表 5.8 类WeightedGraph主要方法的描述

getWeightAdjacencyMatrix():DoubleMatrix 返回带权图的邻接矩阵，也即带权图的距离矩阵
getTotalWeight():double 返回带权图的总权，也即所有边的权的总和
getDijkstraRecorder():DijkstraRecorder，静态方法 返回记录Dijkstra算法执行过程信息的对象实例，也即静态数据成员dijkstraRecorder
dijkstra(GraphNode，GraphNode):List<WeightedGraphPath> 使用Dijkstra算法计算给定起始顶点到其他顶点的最短路径。第一个参数给定起始顶点，第二个参数给定目标顶点，如果是null，则计算起始顶点到其他每个顶点的最短路径
getKruskalRecorder():KruskalRecorder，静态方法 返回记录Kruskal算法执行过程信息的对象实例，也即静态数据成员kruskalRecorder
sortEdgesByWeight():WeightedGraphEdge[]，私有方法 将带权图的所有边按权从小到大进行排序，排序后的结果放到一个数组中
kruskal():WeightedGraph 使用Kruskal算法计算带权图的一棵最小生成树，计算结果用类WeightedGraph的对象实例表示
getPrimRecorder():PrimRecorder: static 返回记录Prim算法执行过程信息的对象实例，也即静态数据成员primRecorder
prim():WeightedGraph 使用Prim算法计算带权图的一棵最小生成树，计算结果用类WeightedGraph的对象实例表示

程序 5.7 类WeightedGraph的方法dijkstra()的实现

```
1  public List<WeightedGraphPath> dijkstra(GraphNode source, GraphNode dest) {
2      // 此处省略对nodeArray、selectedArray、distanceArray、lastNodeIndex和matrix初
       // 始化的代码
3      dijkstraRecorder = new DijkstraRecorder(nodeArray);
```

```
    int stepCounter = 0;
    dijkstraRecorder.addStepRecord(stepCounter, 0, selectedArray,
        distanceArray, lastNodeIndex);
    while (true) {
        boolean hasNode = false;
        for (int i = 0; i < selectedArray.length; i++) {
            if (selectedArray[i] == false) {
                hasNode = true; break;
            }
        }
        if (hasNode == false) break;
        int minIndex = -1;
        for (int i = 0; i < distanceArray.length; i++) {
            if (selectedArray[i] == false) {
                if (minIndex == -1) minIndex = i;
                else if (distanceArray[i]<distanceArray[minIndex]) minIndex = i;
            }
        }
        selectedArray[minIndex] = true;
        for (int i = 0; i < nodeArray.length; i++) {
            if (selectedArray[i] == false) {
                int matrixRow = nodes.indexOf(nodeArray[minIndex]);
                int matrixCol = nodes.indexOf(nodeArray[i]);
                double newDistance = distanceArray[minIndex]+
                matrix.get(matrixRow, matrixCol);
                if (distanceArray[i] > newDistance) {
                    distanceArray[i]=newDistance; lastNodeIndex[i]=minIndex;
                }
            }
        }
        stepCounter++;
        dijkstraRecorder.addStepRecord(stepCounter, minIndex, selectedArray,
            distanceArray, lastNodeIndex);
        if (dest != null) {
            if (nodeArray[minIndex].equals(dest)) break;
        }
    }
    // 此处省略利用lastNodeIndex构造source到已求得最短距离的各顶点的最短路径的构造
}
```

（1）数组nodeArray存储带权图的所有顶点，通过类WeightedGraph保存的顶点列表nodes转换得到，其主要目的是将指定的起始顶点作为数组的第一个元素（即nodeArray[0]），这样可更好地使用表格展示算法的执行过程信息。

（2）数组selectedArray记录nodeArray中对应的顶点是否已经被选中过，一旦被选中过，则就得到了这个顶点的最短距离。初始化时，除了起始顶点对应的第一个元素selectedArray[0]被置为true外，其他元素都被初始化为false。

（3）数组distanceArray记录nodeArray中对应顶点的当前最短距离，即算法主循环

的一次循环后，distanceArray[i] 是顶点 nodeArray[0] 到顶点 nodeArray[i] 的当前最短距离。起始顶点对应的 distanceArray[0] 初始化为0，其他元素都初始化为起始顶点到相应顶点的边的权，如果没有边则初始化为 Java 语言预定义的最大双精度数 Double.MAX_VALUE。

（4）数组 lastNodeIndex 记录 nodeArray 中对应顶点当前最短路径的直接前驱，即算法主循环的一次循环后，lastNodeIndex[i] 是顶点 nodeArray[0] 到顶点 nodeArray[i] 的当前最短路径中，nodeArray[i] 的直接前驱在数组 nodeArray 的下标。初始化时，lastNodeIndex[0] 的值初始化为0。当起始顶点 nodeArray[0] 到顶点 nodeArray[i] 有边时，lastNodeIndex[i] 初始化为0（即起始顶点的下标），否则初始化为 −1。

方法 dijkstra() 一开始对上述数组按照上面所说的方式进行初始化。程序5.7省略了初始化这些数组以及变量 matrix 的源代码。变量 matrix 是当前带权图的距离矩阵，可通过方法 getWeightedAdjacentMatrix() 得到，后者根据类 WeightedGraph 保存的顶点列表和边列表信息构造带权图的距离矩阵。程序5.7的第3~6行，将算法主循环执行之前的初始状态记录到静态数据成员 dijkstraRecorder 中。

程序5.7的第7~40行实现 Dijkstra 算法的主循环，其中第9~13行的循环检查是否还有顶点没有计算得到最短距离。如果所有顶点都已经计算了最短距离，则在第14行终止执行这个主循环。第16~21行的循环在还没有计算最短距离的顶点中，找一个当前距离，即 distanceArray 的值最小的一个顶点，记录其下标 minIndex，该顶点是这一步循环选中的顶点，即在这一步计算得到这个顶点的最短距离。

程序5.7第23~33行的循环修改其他还没有计算得到最短距离的顶点的当前距离和当前路径的直接前驱。第34~36行记录这一步循环中的关键信息，从而记录算法的执行过程信息。第37行判断是否指定了目标顶点，如果指定了，且刚刚选中的顶点就是目标顶点，那么也可终止第7行开始的主循环的执行。主循环执行结束后，不难利用数组 lastNodeIndex 构建起始顶点到各顶点的最短路径。程序5.7省略了这一部分的代码。

用例“带权图最短路径计算”的边界类是类 ShortestPathUIManager，负责绘制展示带权图最短路径计算的输入界面，如图5.15所示。在这个界面中，用户输入按照用例分析一节中预定的格式输入带权图的顶点和边信息，也可选择随机生成这些信息。ShortestPath-

图 5.15 展示带权图最短路径计算的输入界面

UIManager利用类GraphUtil提供的方法提取带权图的顶点和边的信息，以及在需要时随机生成这些输入信息。

图5.15给出的界面中，用户需要指定起始顶点，并可选择是否展示带权图的距离矩阵和是否给出图形化表示，以及展示计算结果的方法。在单击“开始执行”按钮之后，类ShortestPathUIManager利用类GraphUtil提供的方法提取信息，得到表示带权图的类WeightedGraph的对象实例，并调用方法dijkstra()，记录算法的执行过程信息和计算结果。对于图5.15的输入，将得到图5.16的输出结果，其中按照用例分析一节预定的展示形式，使用表格给出算法的执行过程信息，以及起始顶点到各顶点的最短路径，和所选中的边构成的起始顶点到各顶点最短路径的图形化表示形式。

图 5.16　展示带权图最短路径计算的输出结果

类WeightedGraph的方法kruskal()使用Kruskal算法计算带权图的一棵最小生成树，它主要利用类DefaultGraph提供的私有方法addEdgeToComponentList()按照边的权从小到大逐一考查边，根据边对已有连通分支列表的影响确定这条边是否与选中的边构成回路，从而得到最小生成树。程序5.8给出了这个方法的源代码，其中第2行创建一个空的连通分支列表，第3行的列表selectEdge用于记录选中作为树枝的边，第4行调用私有方法sortEdgesByWeight()将带权图的边按从小到大排序后存储在数组sortedEdges中。

程序 5.8　类WeightedGraph的方法kruskal()的实现

```
1  public WeightedGraph kruskal () {
2      List<GraphComponent> componentList = new ArrayList<GraphComponent>();
3      List<GraphEdge> selectEdgeList = new ArrayList<GraphEdge>();
4      WeightedGraphEdge[] sortedEdges = sortEdgesByWeight();
5      int totalNumber = nodes.size();
6      int stepCounter = 1, selectedCounter = 0;
7      kruskalRecorder = new KruskalRecorder();
8      for (int i=0; i<sortedEdges.length && selectedCounter<totalNumber-1; i++) {
9          WeightedGraphEdge edge = sortedEdges[i];
10         boolean selected = false;
```

```
11          if (edge.getStartNode().equals(edge.getEndNode())) continue;
12          GraphComponent component = addEdgeToComponentList(componentList, edge);
13          if (component.getType() != GraphComponent.CIRCLED_COMPONENT) {
14              selectEdgeList.add(edge); selectedCounter++; selected = true;
15          } else component.changeTypeTo(GraphComponent.NORMAL_COMPONENT);
16          kruskalRecorder.addStepRecorder(stepCounter, edge, selected);
17          stepCounter++;
18      }
19      WeightedGraph result = new WeightedGraph("MSTof" + id);
20      result.setNodes(nodes);
21      result.setEdges(selectEdgeList);
22      return result;
23  }
```

程序5.8的第8~18行是实现Kruskal算法的主循环，按照边权从小到大逐一考查带权图的边，调用类DefaultGraph的方法addEdgeToComponent()考查每条边添加到已有连通分支列表产生的影响，只要返回的连通分支类型不是CIRCLED_COMPONENT，则表明当前考查的边不与已选中的边构成回路，从而将其添加到列表selectEdgeList，否则就与已选中的边构成回路，不能作为树枝，这时将返回的连通分支类型改回为NORMAL_COMPONENT，以方便后续的考查。循环中的第16行利用静态数据成员kurskalRecorder记录算法执行的过程信息，即这一步循环考查的边，以及是否选中的信息。程序5.8的主循环后的第19~21行使用带权图原有的顶点列表，以及选中的边列表selectEdgeList创建一个类WeightedGraph的对象实例，作为最小生成树的计算结果返回。

类WeightedGraph的方法prim()使用Prim算法计算带权图的一棵最小生成树，程序5.9给出了它的源代码，其中使用列表selectEdgeList记录选中作为树枝的边，主循环执行之前初始化为空表；列表treeNodeList记录已经在生成树中的顶点，主循环执行之前初始化为只包含顶点列表nodes的第一个顶点的列表；列表otherNodeList记录尚未在生成树中的顶点，主循环之前初始化为包含顶点列表nodes除第一个顶点之外的顶点的列表。

程序 5.9 类WeightedGraph的方法prim()的实现

```
1   public WeightedGraph prim () {
2       // 此处省略初始化selectEdgeList、treeNodeList和otherNodeList的代码
3       primRecorder = new PrimRecorder();
4       int stepCounter = 1;
5       while (!otherNodeList.isEmpty()) {
6           List<GraphEdge> considerEdgeList = new ArrayList<GraphEdge>();
7           WeightedGraphEdge selectEdge = null;
8           GraphNode otherNode = null;
9           for (GraphEdge edge : edges) {
10              GraphNode start = edge.getStartNode(), end = edge.getEndNode();
11              WeightedGraphEdge weightedEdge = (WeightedGraphEdge)edge;
12              if (treeNodeList.contains(start)&&otherNodeList.contains(end)) {
13                  if (selectEdge == null) {
14                      selectEdge = weightedEdge; otherNode = end;
15                  } else if (weightedEdge.weight < selectEdge.weight) {
16                      selectEdge = weightedEdge; otherNode = end;
```

```
17                  }
18                  considerEdgeList.add(weightedEdge);
19              }
20              if (treeNodeList.contains(end)&&otherNodeList.contains(start)) {
21                  if (selectEdge == null) {
22                      selectEdge = weightedEdge; otherNode = start;
23                  } else if (weightedEdge.weight < selectEdge.weight) {
24                      selectEdge = weightedEdge; otherNode = start;
25                  }
26                  considerEdgeList.add(weightedEdge);
27              }
28          }
29          if (selectEdge != null) {
30              primRecorder.addStepRecorder(stepCounter, considerEdgeList,
31                  treeNodeList, otherNodeList, selectEdge, otherNode);
32              stepCounter++; selectEdgeList.add(selectEdge);
33              treeNodeList.add(otherNode); otherNodeList.remove(otherNode);
34          } else {
35              treeNodeList.clear();
36              GraphNode first = otherNodeList.get(0);
37              otherNodeList.remove(0); treeNodeList.add(first);
38          }
39      }
40      // 此处省略创建类WeightedGraph的对象实例的语句，该对象实例作为计算结果返回
41  }
```

程序5.9的第5~39行实现Prim算法的主循环,其中第6行声明的变量considerEdgeList记录某一次循环中考虑的所有边。第9~28行的循环考查带权图的所有边，但只有当边的一个端点在treeNodeList、另外一个端点在otherNodeList时，才会真正被考虑并且添加到列表considerEdgeList中。第9行开始的循环结束后，变量selectEdge记录列表considerEdgeList中权最小的边，变量otherNode记录这条边的那个在otherNodeList中的顶点。程序5.9的第29行的条件语句当selectEdge不为null时，将其添加到选中的边列表中，将otherNode添加到生成树的顶点列表，并将其从otherNodeList中删除。如果selectEdge为null，则表明当前的带权图有多个连通分支，那么就从otherNodeList的第1个顶点开始再得到剩下图的生成树。

注意到类WeightedGraph的方法kruskal()在选择顶点树减1条边后就终止循环，这时如果当前带权图不是连通图，那么选中的边将构成森林，而非一棵树。因此类WeightedGraph的方法kruskal()和方法prim()都在当前带权图不是连通图时得到森林，而在当前带权图是连通图时才得到最小生成树。

用例“带权图最小生成树计算”的边界类是类SpanningTreeUIManager，负责绘制展示带权图最小生成树计算的输入界面，如图5.17所示。在这个界面中，用户输入按照用例分析一节中预定的格式输入带权图的顶点和边信息，也可选择随机生成这些信息。SpanningTreeUIManager利用类GraphUtil提供的方法提取带权图的顶点和边的信息，以及在需要时随机生成这些输入信息。

图5.17给出的界面中，用户可选择是否展示带权图的距离矩阵和是否给出图形化表示，执行的最小生成树算法，以及展示计算结果的方法。在单击“开始执行”按钮之后，类SpanningTreeUIManager利用类GraphUtil提供的方法提取信息，得到表示带权图的类WeightedGraph的对象实例，并调用方法kruskal()或者方法prim()，记录算法的执行过程信息和计算结果。对于图5.17的输入，将得到图5.18的输出结果。由于这里输入的带权图信息与图5.15给出的带权图相同，因此带权图的信息，包括它的顶点和边、距离矩阵以及图形化表示和图5.16相同，因此在图5.18中就没有截取这部分的屏幕，而只是截取了展示最小生成树算法计算过程和结果的部分内容。可看到，图5.18根据用例分析一节预定的形式，使用表格展示了Kruskal算法和Prim算法的执行过程信息。

图 5.17 展示带权图最小生成树计算的输入界面

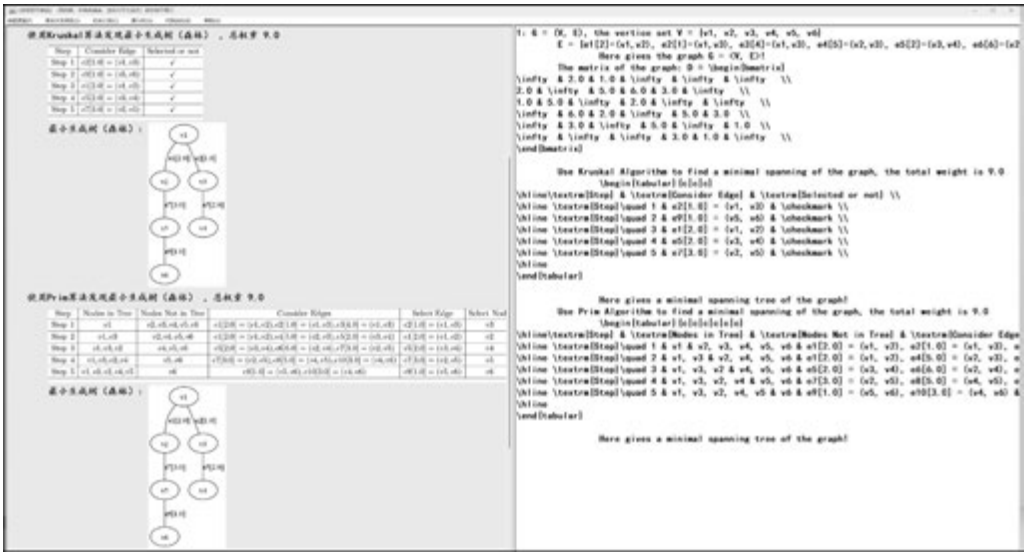

图 5.18 展示带权图最小生成树计算的输出结果

5.3.4 哈夫曼树构造算法展示的实现

展示哈夫曼树构造算法的核心类是HuffmanTree，它表示一棵构造出来的哈夫曼树，也提供静态方法构造哈夫曼树，并记录构造的过程。这个类继承类RootedTree，因此最重要的

数据成员是继承成员root，给出树根顶点。这个类自己还定义了静态数据成员huffmanRecorder，用于记录哈夫曼树构造的过程信息。表5.9给出了类 HuffmanTree 的主要方法的描述。

表 5.9　类 HuffmanTree 主要方法的描述

createHuffmanTree(WeightedTreeNode[]):HuffmanTree，静态方法 以给定的叶子顶点（包括权的信息）构建一棵哈夫曼树
getHuffmanRecorder():HuffmanRecorder: static 返回记录有构建哈夫曼树构造过程信息的对象实例，即返回静态数据成员 huffmanRecoder
extractWeightedLeafsFromFormatedString(String，List<GraphNode>):boolean，静态方法 提取给定字符串中的叶子顶点信息放到类型为 List<GraphNode> 的参数中，如果给定的字符串符合规定的格式，能提取叶子信息，则返回 true，如果格式有误，则返回 false，这时第二个参数中的叶子顶点信息不可用
getErrorMessage():String，静态方法 返回在提取叶子顶点信息时可能的格式错误信息
randomGenerateWeightedLeafNodes(int，int):List<GraphNode>，静态方法 随机生成给定个数的带权叶子顶点，第一个整数参数指定要生成的叶子个数，第二个整数参数指定叶子最大的权值
getWeightedLeafFormatedString(List<GraphNode>):String，静态方法 给定一个带权叶子顶点列表，将其转换为一个包含叶子顶点信息的字符串，这个字符串可作为构建哈夫曼树图形用户界面的输入
HuffmanTree(String，WeightedTreeNode):Constructor 以指定的根顶点创建类的对象实例，第一个参数的字符串可以用于给这棵哈夫曼树命名
getCodeOfLeafNodes(WeightedTreeNode[]):String[] 返回给定的一组树顶点的哈夫曼编码，这组顶点应该在当前这棵哈夫曼树中
codingLeafNode(WeightedTreeNode，String，WeightedTreeNode[]，String[]):void，私有方法 这个方法被方法 getCodeOfLeafNodes() 调用，递归地实现对以第一个参数为根的子树的叶子顶点进行编码
getTotalWeight():double 返回整棵树的总权
calculateTotalWeightOfSubTree(WeightedTreeNode，int):double: 私有方法 这个方法被方法 getTotalWeight() 调用，递归地实现以给定顶点为根的子树的权，整数参数用于指明这个根所在的层数

类HuffmanTree的静态方法createHuffmanTree()构造哈夫曼树，并将构造过程信息记录到静态数据成员huffmanRecorder。程序 5.10给出了这个方法的源代码，其中首先将给定的数组leafs中存储的带权叶子顶点复制到数组nodeArray，复制时根据选择排序的思想，依次选择leafs中权最小的顶点复制到nodeArray，从而使得nodeArray将这些带权叶子顶点按权从小到大的顺序存储。这样做的目的是在不破坏参数leafs的情况下，将需要构造哈夫曼树的带权叶子顶点进行排序。为简便起见，程序5.10中省略了这段源代码。

程序 5.10　类 HuffmanTree 的方法 createHuffmanTree() 的实现

```
1  public static HuffmanTree createHuffmanTree(WeightedTreeNode[] leafs) {
2      // 此处省略初始化按权从小到大存储带权叶子顶点的数组 nodeArray 的代码
3      huffmanRecorder = new HuffmanRecorder();
4      int startIndex = 0;
```

```
5       huffmanRecorder.addStepRecorder(startIndex, nodeArray, startIndex,
6           nodeArray.length);
7       while (startIndex < nodeArray.length-1) {
8           WeightedTreeNode leftChild = nodeArray[startIndex];
9           WeightedTreeNode rightChild = nodeArray[startIndex+1];
10          double weight = leftChild.getWeight() + rightChild.getWeight();
11          WeightedTreeNode newNode = new WeightedTreeNode("t"+startIndex, weight);
12          newNode.addChildNode(leftChild); newNode.addChildNode(rightChild);
13          int insertIndex = startIndex+1;
14          while (insertIndex < nodeArray.length-1) {
15              if (nodeArray[insertIndex+1].getWeight() < weight) {
16                  nodeArray[insertIndex] = nodeArray[insertIndex+1];
17                  insertIndex = insertIndex+1;
18              } else break;
19          }
20          nodeArray[insertIndex] = newNode;
21          startIndex = startIndex + 1;
22          huffmanRecorder.addStepRecorder(startIndex, nodeArray, startIndex,
23              nodeArray.length);
24      }
25      String id = "Huffman" + leafs.length;
26      WeightedTreeNode root = nodeArray[startIndex];
27      HuffmanTree result = new HuffmanTree(id, root);
28      return result;
29  }
```

实现哈夫曼树构造的主循环是程序5.10的第7~24行。在这个循环中，待处理的顶点是数组nodeArray从下标startIndex处开始的顶点，第8、9行取得待处理的顶点中权最小的两个顶点，构造新的顶点newNode，并以这两个顶点作为左右子顶点，以这两个顶点的权的和作为权。第14~19行的循环将新顶点newNode插入nodeArray的合适地方，使得还要处理的顶点从startIndex+1处开始按权从小到大排序。由于每次循环时，权最小的两个顶点在startIndex和startIndex+1处，因此在插入时，可利用startIndex+1处作为缓冲区来移动小于新顶点newNode权的那些顶点，从而将新顶点插入合适位置。

算法主循环的第22行将构造过程信息添加到静态数据成员huffmanRecorder中，其中第一个实际参数startIndex实际上给出的是步数信息，而这次循环后得到的中间结果是以 nodeArray中从下标startIndex处开始的所有顶点为根的多棵树构成的森林。方法addStepRecorder会将这些根顶点复制到类HuffmanRecoder的数据成员stepList中，从而在需要时展示这些树构成的森林。因此当第7~25行的主循环执行完毕时，最终得到的哈夫曼树的根就在nodeArray的 startIndex处，程序5.10的第27行利用这个根顶点构造一个类HuffmanTree的对象实例作为构造的结果返回。

类HuffmanTree的方法getCodeOfLeafNodes()用于给出一组带权叶子顶点的哈夫曼编码，通常这组带权叶子顶点就是用于构造这个哈夫曼树的顶点，这个方法调用递归实现的方法codingLeafNode()，对哈夫曼树叶子顶点进行编码。程序5.11给出了这个私有方法

的源代码，基本上按照算法5.2的思路实现，其中参数subtreeRoot是要编码的子树根顶点，currentCode是当前的编码前缀，leafs是最初要编码的叶子顶点数组，codes存储每个要编码的叶子顶点所得到的编码。当subtreeRoot没有子顶点时，程序5.11的第5~8行在leafs中找到它所在的位置，并设置其对应的编码，否则分别以subtreeRoot的左右子顶点，以及编码前缀分别加上0和1递归调用本方法，从而完成其中叶子顶点的编码。

程序 5.11　类HuffmanTree的私有方法codingLeafNode()的实现

```
1  private void codingLeafNode(WeightedTreeNode subtreeRoot, String currentCode,
2          WeightedTreeNode[] leafs, String[] codes) {
3      List<RootedTreeNode> childList = subtreeRoot.getChildList();
4      if (childList == null) {
5          for (int i = 0; i < leafs.length; i++) {
6              if (leafs[i].equals(subtreeRoot)) {
7                  codes[i] = currentCode; break;
8              }
9          }
10     } else {
11         WeightedTreeNode left = (WeightedTreeNode)childList.get(0);
12         WeightedTreeNode right = (WeightedTreeNode)childList.get(1);
13         codingLeafNode(left, currentCode+"0", leafs, codes);
14         codingLeafNode(right, currentCode+"1", leafs, codes);
15     }
16 }
```

类HuffmanTree的方法getTotalWeigth()用于计算哈夫曼树的总权，通过调用递归实现的方法calculateTotalWeightOfSubTree()来计算子树的权，从而得到整棵树的权。程序5.12给出了这个私有方法的源代码，基本上按照算法5.1的思路实现，其中参数subtreeRoot是要计算权的子树根顶点，level给出这个子树根顶点在整棵树的层数。当subtreeRoot没有子顶点时，程序5.12的第6~9行计算它的权，即等于这个顶点的权乘以它所在的层数，其中用到的变量isFirstLeaf和weightCalLaTeXString是类HuffmanTree的数据成员，相当于这个私有方法的全局变量，用于以LaTeX命令串的形式给出整棵哈夫曼树的总权的计算公式，从而可不仅展示权的计算结果，而且能展示权的计算过程。当subtreeRoot不是叶子顶点时，则以层数加1为参数递归调用本方法计算以它的子顶点为根的子树的权，然后再完成这棵子树的权的计算。

程序 5.12　类HuffmanTree的私有方法calculateTotalWeightOfSubTree()的实现

```
1  private double calculateTotalWeightOfSubTree(WeightedTreeNode subtreeRoot,
2      int level) {
3      double result = 0;
4      List<RootedTreeNode> childList = subtreeRoot.getChildList();
5      if (childList == null) {
6          result = subtreeRoot.getWeight() * level;
7          if (isFirstLeaf) isFirstLeaf = false;
8          else weightCalLaTeXString.append(" + ");
9          weightCalLaTeXString.append(subtreeRoot.getWeight()+"\\times "+  level);
```

```
10      } else {
11          for (RootedTreeNode treeNode : childList) {
12              WeightedTreeNode node = (WeightedTreeNode)treeNode;
13              result = result + calculateTotalWeightOfSubTree(node, level+1);
14          }
15      }
16      return result;
17  }
```

类HuffmanTreeUIManager是实现用例“哈夫曼树构造”的边界类，负责绘制展示哈夫曼树构造的输入界面，如图5.19所示。在这个界面中，用户按照在用例分析一节预定的格式输入叶子顶点及其权的信息。类HuffmanTree提供方法从输入的字符串中提取叶子顶点及其权，也提供方法随机生成一个字符串作为输入。

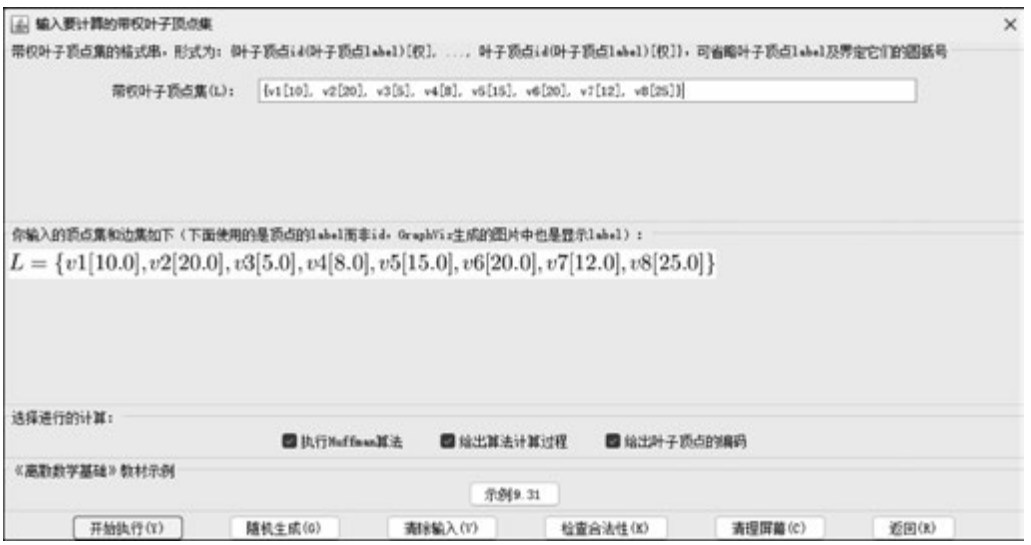

图 5.19 展示哈夫曼树构造的输入界面

图5.19给出的界面中，用户可选择是否给出构造过程，以及是否给出叶子顶点的编码。在单击“开始执行”按钮后，类HuffmanTreeUIManager利用类HuffmanTree提供的方法提取信息，得到叶子顶点及其权的信息，然后调用HuffmanTree的静态方法createHuffmanTree()构造哈夫曼树，并记录构造过程信息，然后在输出屏幕区域展示结果。对于图5.19的输入，将得到图5.20的输出结果，其中展示了算法主循环每执行一次后得到的顶点序列，并用图形化方式给出了以这些顶点为根的森林，最后给出了最终得到的哈夫曼树，以及它的总权和叶子顶点编码。由于步数比较多，我们截取了最开始两步和包含最后一步，以及最终的哈夫曼树及其权和叶子顶点编码等信息的图片。用户在软件中可通过滚动屏幕得到整个构造过程的展示。

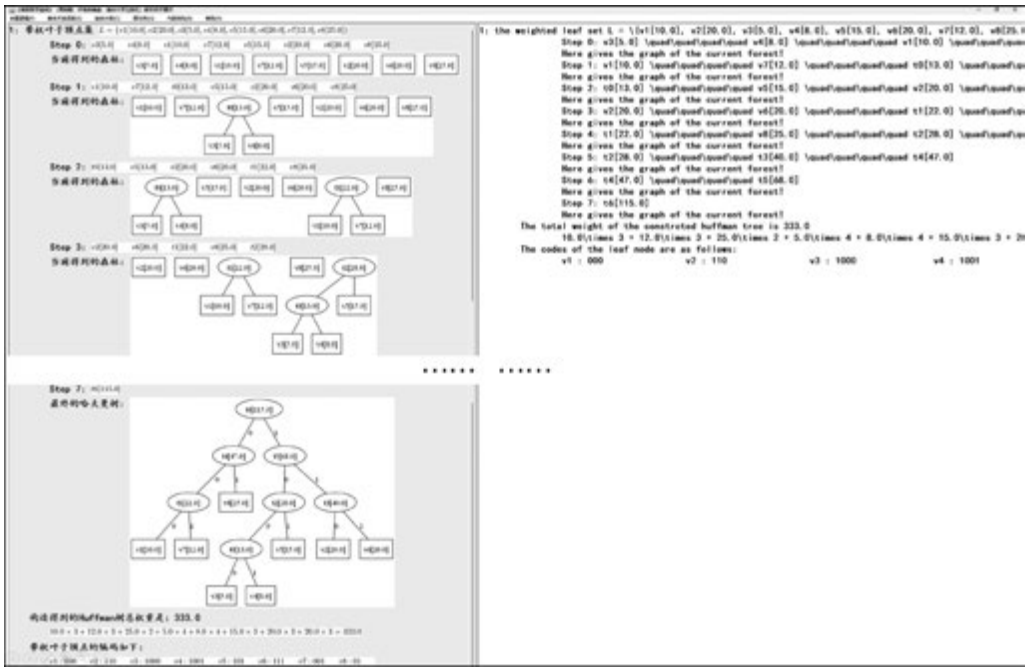

图 5.20　展示哈夫曼树构造的输出结果

5.4　图和树问题建模与求解小结

本章给出了图和树问题的建模与求解，主要是展示图的先深遍历和先广遍历、根树的前序、中序和后序遍历、带权图的最短路径和最小生成树计算，以及哈夫曼树的构造算法，实现这些算法，并以适当的方式展示算法的运行过程与计算结果。为此，我们主要设计和实现了类 AbstractGraph、DefaultGraph、RootedTree、WeightedGraph 和类 HuffmanTree。除类 AbstractGraph 外，其他 4 个类不仅分别是图、根树、带权图和哈夫曼树这些实体在软件中的表示，也是要展示的算法的求解器，即类 DefaultGraph 展示图的遍历算法，类 RootedTree 展示根树的遍历算法，类 WeightedGraph 展示带权图最短路径和最小生成树算法，以及类 HuffmanTree 展示哈夫曼树构造算法。进一步，为展示算法的运行过程信息，我们设计了以后缀 Recorder 命名的记录类，用以记录过程信息，并在展示算法的求解器类使用记录类的对象实例作为静态数据成员，在算法的主循环中保存每一步循环后的关键信息，从而实现算法运行过程信息的展示。

目前图和树这一模块实现的算法展示还比较少，因此我们用表示实体的类同时负责算法的实现与展示。如果要实现和展示更多算法，更好的设计应该是将实体类与算法的展示类分开，并在算法展示类中定义算法运行过程信息的记录类，从而使得软件的结构更加清晰，并且更容易扩充类的功能。例如，目前我们对算法过程的展示，主要是使用表格、文字、图形等静态形式展示算法主循环执行一次后得到的一些关键信息，但显然，对于图和树的算法，同时给出图和树的直观图形，并在图形上以动画的形式展示算法的运行过程，会让学生对相应的算法有更好的理解。下一步，可考虑分开算法的实现与展示，扩充算法运行过程信息记录类，增加算法运行过程动画展示的渲染类，实现图与树算法的动画展示，并且实现更多图论算法的展示。