```
4    public static void main(String[] args) {
5      // Prompt the user to enter the nine coins' Hs and Ts
6      System.out.print("Enter an initial nine coins' Hs and Ts: ");
7      Scanner input = new Scanner(System.in);
8      String s = input.nextLine();
9      char[] initialNode = s.toCharArray();
10
11     WeightedNineTailModel model = new WeightedNineTailModel();
12     java.util.List<Integer> path =
13       model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15     System.out.println("The steps to flip the coins are ");
16     for (int i = 0; i < path.size(); i++)
17       NineTailModel.printNode(NineTailModel.getNode(path.get(i)));
18
19     System.out.println("The number of flips is " +
20       model.getNumberOfFlips(NineTailModel.getIndex(initialNode)));
21   }
22 }
```

```
Enter an initial nine coins Hs and Ts: HHHTTTHHH Enter

The steps to flip the coins are
HHH
TTT
HHH
HHH
THT
TTT

TTT
TTT
TTT

The number of flips is 8
```

该程序在第 8 行提示用户将一个由 H 和 T 构成的 9 个字母的初始结点作为字符串输入，从该字符串获取一个字符数组（第 9 行），然后创建一个模型（第 11 行），获取从初始结点到目标结点的一个最短路径（第 12 ～ 13 行），显示路径中的结点（第 16 ～ 17 行），最后调用 `getNumberOfFlips` 来获取到达目标结点所需的翻转次数（第 20 行）。

复习题

29.15 为什么程序清单 28-13 中 `NineTailModel` 的 `tree` 数据域定义为受保护的？

29.16 `WeightedNineTailModel` 中是如何创建图的结点的？

29.17 `WeightedNineTailModel` 中是如何创建图的边的？

关键术语

Dijkstra'algorithm（Dijkstra 算法）
edge-weighted graph（边加权图）
minimum spanning tree（最小生成树）
Prim's algorithm（Prim 算法）
shortest path（最短路径）
single-source shortest path（单源最短路径）
vertex-weighted graph（顶点加权图）

本章小结

1. 可以使用邻接矩阵或者线性表来存储图中的加权边。
2. 图的生成树是一个子图，也是一棵树，并连接着图中所有的顶点。
3. Prim 算法找出最小生成树的工作机制如下：算法首先从包含一个任意结点的生成树开始。算法通过

添加和已在树中的顶点具有最小权重边的结点，来扩展这棵树。

4. Dijkstra 算法从源顶点开始搜索，然后一直寻找到源顶点具有最短路径的结点，直到所有结点被找到。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

*29.1 （Kruskal 算法）本书中介绍了找出最小生成树的 Prim 算法。Kruskal 算法是另一种著名的找出最小生成树的算法。该算法重复地找出最小权重边，如果不会形成环，就将它添加到树中。当所有顶点都在树中时，终止这个过程。使用 Kruskal 算法设计和实现一个找出 MST 的算法。

*29.2 （使用邻接矩阵实现 Prim 算法）教材在邻接边上使用线性表实现 Prim 算法。对于加权图，使用邻接矩阵实现该算法。

*29.3 （使用邻接矩阵实现 Dijkstra 算法）教材在邻接边上使用线性表来实现 Dijkstra 算法。对于加权图，使用邻接矩阵实现该算法。

*29.4 （修改 9 枚硬币反面问题中的权值）教材中我们将翻转的次数作为每次移动的权重。假设权值是翻转次数的 3 倍，然后修改这个程序。

*29.5 （证明或反证）猜想 `NineTailModel` 和 `WeightedNineTailModel` 可能会得到相同的最短路径。编写程序去证明或者反证这个观点。（提示：令 `tree1` 和 `tree2` 分别表示从 `NineTailModel` 和 `WeightedNineTailModel` 获取的根结点为 511 的树。如果一个结点 `u` 在 `tree1` 中的深度和在 `tree2` 中的深度一样，那么，从结点 `u` 到目标结点的路径长度是相同的。）

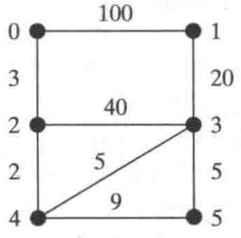
**29.6 （加权 4×4 16 枚硬币反面的模型）教材中加权的 9 枚硬币反面问题使用的是 3×3 的矩阵。假设你有 16 枚放在 4×4 的矩阵中的硬币。创建一个名为 WeightedTailModel16 的新的模型类，然后创建模型的一个实例并且将这个对象存入一个名为 WeightedTailModel16.dat 的文件中。

**29.7 （加权 4×4 16 枚硬币反面问题）为加权 4×4 16 枚硬币反面的问题修改程序清单 29-9。程序应该读取前一个编程练习题创建的模型对象。

**29.8 （旅行商人问题）旅行商人问题（traveling salesman problem, TSP）就是找出往返的最短路径，访问每个城市一次且只能访问一次，最后返回到起始城市。这个问题等价于编程练习题 28.17 中的寻找一条最短的哈密尔顿环。在 WeightedGraph 类中添加下面的方法：

```
// Return a shortest cycle
// Return null if no such cycle exists
public List<Integer> getShortestHamiltonianCycle()
```

*29.9 （找出最小生成树）编写一个程序，从文件中读取一个连通图并且显示它的最小生成树。文件中的第一行是表明顶点个数（`n`）的数字。顶点被标记为 `0,1,…,n-1`。接下来的每一行以 `u1,v1,w1|u2,v2,w2|…` 的形式来描述边。每个三元组描述一条边和它的权重。图 29-24 显示了与图对应的文件的例子。注意，我们假设图是无向的。如果图有一条边 `(u,v)`，那么它也有一条边 `(v,u)`，但文件中只表示了一条边。当构建一个图时，两条边都需要考虑。



a）

```
File
6
0, 1, 100 | 0, 2, 3
1, 3, 20
2, 3, 40 | 2, 4, 2
3, 4, 5 | 3, 5, 5
4, 5, 9
```

b）

图 29-24 加权图的顶点和边可以存储在一个文件中

程序应该提示用户输入文件名，然后从文件中读取数据，建立 WeightedGraph 的一个实例 g，调用 g.printWeightedEdges() 来显示所有的边，调用 getMinimumSpanningTree() 来获取一个 WeightedGraph.MST 的实例 tree，调用 tree.getTotalWeight() 来显示最小生成树的权重，以及调用 tree.printTree() 来显示这棵树。下面是这个程序的运行示例：

```
Enter a file name: c:\exercise\WeightedGraphSample.txt ↵Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
Total weight in MST is 35
Root is: 0
Edges: (3, 1) (0, 2) (4, 3) (2, 4) (3, 5)
```

提示：使用 new WeightedGraph(list,numberOfVertices) 来创建一个图，其中 list 包含一个 WeightedEdge 对象的线性表。使用 new WeightedEdges(u,v,w) 来创建一条边。读取第一行以获取顶点的个数。将接下来的每一行读入一个字符串 s 中，并且使用 s.split("[\\|]") 来提取三元组。对于每个三元组，triplet.split("[,]") 提取顶点和权值。

*29.10 （为图创建文件）修改程序清单 29-3，创建一个表示 graph1 的文件。文件格式在编程练习题 29.9 中描述。从程序清单 29-3 中的第 7 ～ 24 行定义的数组来创建文件。图的顶点个数为 12，它将存储在文件的第一行。如果 u<v，那么存储边 (u,v)。文件的内容如下所示：

```
12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239
```

*29.11 （找出最短路径）编写一个程序，从文件中读取一个连通图。图存储在一个文件中，指定格式与编程练习题 29.9 一样。程序应该提示用户输入文件名、两个顶点，然后显示这两个顶点之间的最短路径。例如，对于图 29-23 中的图，顶点 0 和顶点 1 之间的最短路径可以显示为 0 24 3 1。下面是该程序的一个运行示例：

```
Enter a file name: WeightedGraphSample2.txt ↵Enter
Enter two vertices (integer indexes): 0 1 ↵Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
A path from 0 to 1: 0 2 4 3 1
```

*29.12 （显示加权图）修改程序清单 28-6 中的 GraphView，显示加权图。编写一个程序，显示图 29-1

中的图，如图 29-25 所示。(教师可以要求学生扩展该程序，添加带有正确的边的新的城市到该图中。)

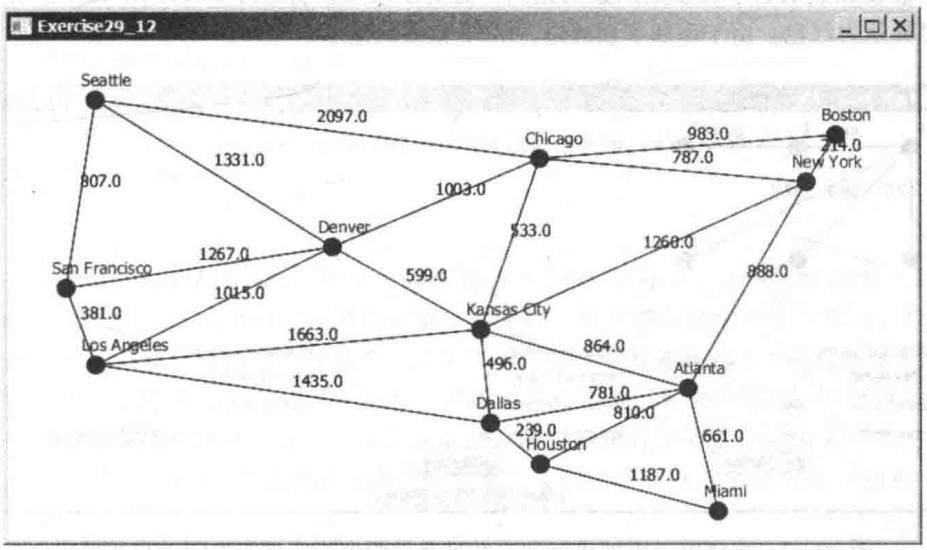


图 29-25 编程练习题 29.12 显示一个加权图

*29.13 (显示最短路径) 修改程序清单 28-6 中的 GraphView，显示一个加权图和两个指定城市之间的最短路径，如图 29-19 所示。需要在 GraphView 中添加一个数据域 path。如果 path 不为空，路径中的边显示为红色。如果输入了一个图中没有的城市，程序显示一个对话框来警告用户。

*29.14 (显示最小生成树) 修改程序清单 28-6 中的 GraphView，为图 29-1 中的图显示其加权图和最小生成树，如图 29-26 所示。MST 的路径显示为红色。

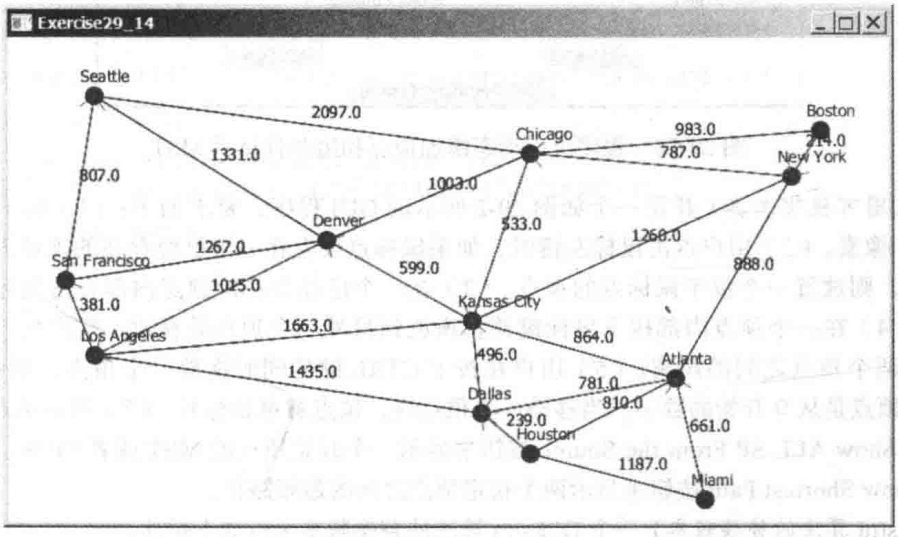


图 29-26 编程练习题 29.14 显示一个 MST

***29.15 (动态图) 编写一个程序，允许用户动态创建加权图。用户通过输入顶点的名字和位置来生成顶点，如图 29-27 所示。用户也可以创建一条边来连接两个顶点。为了简化程序，假设顶点的名字和顶点的索引相同。需要以顶点索引顺序 0,1,…,n 来添加顶点。用户可以指定两个顶点并且让程序以红色来显示它们之间的最短路径。

***29.16 （显示一个动态 MST）编写一个程序，允许用户动态地创建加权图。用户通过输入顶点的名字和位置来生成顶点，如图 29-28 所示。用户也可以创建一条边来连接两个顶点。为了简化程序，假设顶点的名字和顶点的索引相同。需要以顶点索引顺序 0,1,…,n 来添加顶点。MST 的路径显示为红色。由于添加了新的边，MST 被重新显示。

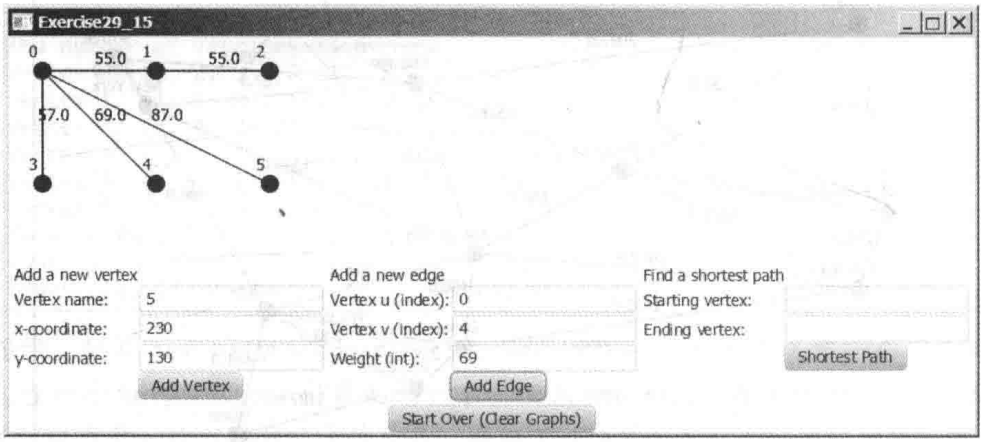


图 29-27 程序可以添加顶点和边并且显示两个指定顶点之间的最短路径

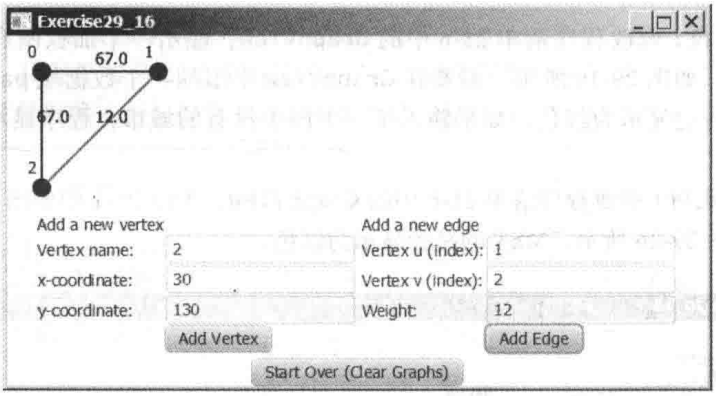


图 29-28 程序可以动态添加顶点和边并且显示 MST

***29.17 （加权图可视化工具）开发一个如图 29-2 所示的 GUI 程序，要求如下：（1）每个顶点的半径为 20 像素。（2）用户点击鼠标左键时，如果鼠标点没有在一个已经存在的顶点内部或者过于接近，则放置一个位于鼠标点的顶点。（3）在一个已经存在的顶点内部右击鼠标来删除该顶点。（4）在一个顶点内部按下鼠标键并且拖放到另外一个顶点处释放，则产生一条边，并且显示两个顶点之间的距离。（5）用户在按下 CTRL 键的同时拖放一个顶点，则移动该顶点。（6）顶点是从 0 开始的数字。当移动一个顶点时，顶点被重新标号。（7）可以单击 Show MST 或者 Show ALL SP From the Source 按钮来显示一个起始顶点的 MST 或者 SP 树。（8）可以单击 Show Shortest Path 按钮来显示两个指定顶点之间的最短路径。

***29.18 （Dijkstra 算法的替换版本）一个 Dijkstra 算法的替换版本可以如下描述：

输入：一个加权图 G = (V,E)，其中权值都为正。

输出：从一个源顶点 s 开始的最短路径树。

```
Input: a weighted graph G = (V, E) with non-negative weights
Output: A shortest path tree from a source vertex s

 1  ShortestPathTree getShortestPath(s) {
```

```
2    Let T be a set that contains the vertices whose
3      paths to s are known;
4    Initially T contains source vertex s with cost[s] = 0;
5    for (each u in V - T)
6       cost[u] = infinity;
7
8    while (size of T < n) {
9      Find v in V - T with the smallest cost[u] + w(u, v) value
10       among all u in T;
11     Add v to T and set cost[v] = cost[u] + w(u, v);
12     parent[v] = u;
13   }
14 }
```

算法使用 cost[v] 来存储从顶点 v 到源顶点 s 的最短路径。cost[s] 为 0。开始时设置 cost[v] 为无穷大，表示没有找到从 v 到 s 的路径。让 V 表示图中的所有顶点，T 表示已经知道开销的顶点集合。开始时，源顶点 s 位于 T 中。算法重复地找到 T 中的顶点 u 和 V-T 中的顶点 v，使得 cost[u] + w(u + v) 最小，并将 v 移至 T 中。教材中给出的最短路径算法不断为 V-T 中的顶点更新开销和父顶点。该算法初始时将每个顶点的开销设置为无穷大，然后在顶点被加入到 T 中的时候仅修改该顶点的开销一次。实现这个算法，并使用程序清单 29-7 来测试你的新算法。

***29.19 （高效找到具有最小 cost[u] 的顶点 u）getShortestPath 方法使用线性搜索，找到具有最小 cost[u] 的顶点 u。这将使用 $O(|V|)$ 的时间。搜索时间可以使用一个 AVL 树缩减为 $O(\log|V|)$。修改该方法，使用一个 AVL 树来存储 V-T 中的顶点。使用程序清单 29-7 来测试你的新实现。

***29.20 （高效测试一个顶点 u 是否在 T 中）在程序清单 29-2 中的方法 getMinimumSpanningTree 和 getShortestPath 中，采用线性表实现了 T。这样通过调用 T.contains(u) 来测试一个顶点 u 是否在 T 中需要 O(n) 的时间。通过引入一个名为 isInT 的数组来修改这两个方法。当一个顶点 u 被加入到 T 中的时候设置 isInT[u] 为 true。测试一个顶点 u 是否在 T 中现在可以在 O(1) 的时间内完成。使用下面的代码编写一个测试程序，其中 graph1 是从图 29-1 中创建的。

```
WeightedGraph<String> graph1 = new WeightedGraph<>(edges, vertices);
WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
System.out.println("Total weight is " + tree1.getTotalWeight());
tree1.printTree();
WeightedGraph<String>.ShortestPathTree tree2 =
  graph1.getShortestPath(graph1.getIndex("Chicago"));
tree2.printAllPaths();
```