3.3.34 所有的 2-3 树。编写一段代码来生成高度为 2、3 和 4 的所有结构不同的 2-3 树，分别共有 2、7 和 122 种（提示：使用符号表）。

3.3.35 2-3 树。编写一段程序 TwoThreeST.java，使用两种结点类型来直接表示和实现 2-3 查找树。

3.3.36 2-3-4-5-6-7-8 树。说明平衡的 2-3-4-5-6-7-8 树中的查找和插入算法。

3.3.37 无记忆性。请证明红黑树不是没有记忆的。例如，如果你向树中插入一个小于所有键的新键，然后立即删除树的最小键，你可能得到一棵不同的树。

3.3.38 旋转的基础定理。请证明，使用一系列左旋转或者右旋转可以将一棵二叉查找树转化为由同一
452 组键生成的其他任意一棵二叉查找树。

3.3.39 删除最小键。实现红黑树的 deleteMin() 方法，在沿着树的最左路径向下的过程中实现正文所述的变换，保证当前结点不是 2- 结点。

解答：

```
private Node moveRedLeft(Node h)
{  // 假设结点 h 为红色，h.left 和 h.left.left 都是黑色，
   // 将 h.left 或者 h.left 的子结点之一变红
   flipColors(h);
   if (isRed(h.right.left))
   {
      h.right = rotateRight(h.right);
      h = rotateLeft(h);
   }
   return h;
}
public void deleteMin()
{
   if (!isRed(root.left) && !isRed(root.right))
      root.color = RED;
   root = deleteMin(root);
   if (!isEmpty()) root.color = BLACK;
}
private Node deleteMin(Node h)
{
   if (h.left == null)
      return null;
   if (!isRed(h.left) && !isRed(h.left.left))
      h = moveRedLeft(h);
   h.left = deleteMin(h.left);
   return balance(h);
}
```

其中的 balance() 方法由下一行代码和算法 3.4 的递归 put() 方法中的最后 5 行代码组成：

453

```
if (isRed(h.right)) h = rotateLeft(h);
```

这里的 flipColors() 方法将会补全三条链接的颜色，而不是正文中实现插入操作时实现的 flipColors() 方法。对于删除，我们会将父结点设为 BLACK（黑）而将两个子结点设为 RED（红）。

3.3.40 删除最大键。实现红黑树的 deleteMax() 方法。需要注意的是因为红链接都是左链接，所以这里用到的变换和上一道练习中的稍有不同。

解答：

```
private Node moveRedRight(Node h)
{  // 假设结点 h 为红色，h.right 和 h.right.left 都是黑色，
   // 将 h.right 或者 h.right 的子结点之一变红
   flipColors(h)
```

```
      if (!isRed(h.left.left))
         h = rotateRight(h);
      return h;
   }
   public void deleteMax()
   {
      if (!isRed(root.left) && !isRed(root.right))
         root.color = RED;
      root = deleteMax(root);
      if (!isEmpty()) root.color = BLACK;
   }
   private Node deleteMax(Node h)
   {
      if (isRed(h.left))
          h = rotateRight(h);
      if (h.right == null)
         return null;
      if (!isRed(h.right) && !isRed(h.right.left))
         h = moveRedRight(h);
      h.right = deleteMax(h.right);
      return balance(h);
   }
```

454

3.3.41 删除操作。将上两题中的方法和二叉查找树的 delete() 方法结合起来，实现红黑树的删除操作。

解答：

```
public void delete(Key key)
{
   if (!isRed(root.left) && !isRed(root.right))
      root.color = RED;
   root = delete(root, key);
   if (!isEmpty()) root.color = BLACK;
}
private Node delete(Node h, Key key)
{
   if (key.compareTo(h.key) < 0)
   {
      if (!isRed(h.left) && !isRed(h.left.left))
         h = moveRedLeft(h);
      h.left =  delete(h.left, key);
   }
   else
   {
      if (isRed(h.left))
         h = rotateRight(h);
      if (key.compareTo(h.key) == 0 && (h.right == null))
         return null;
      if (!isRed(h.right) && !isRed(h.right.left))
         h = moveRedRight(h);
      if (key.compareTo(h.key) == 0)
      {
         h.val = get(h.right, min(h.right).key);
         h.key = min(h.right).key;
         h.right = deleteMin(h.right);
      }
      else h.right = delete(h.right, key);
   }
   return balance(h);
}
```

455

实验题

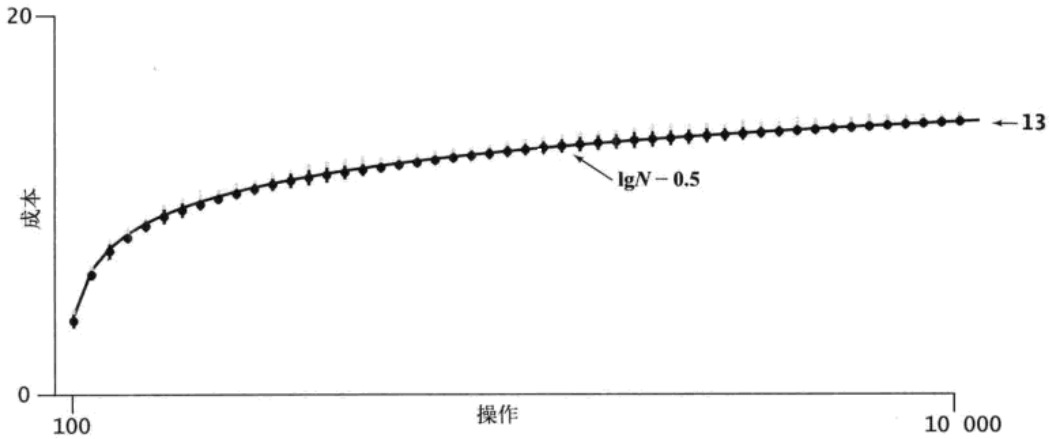
3.3.42 统计红色结点。编写一段程序，统计给定的红黑树中红色结点所占的比例。对于 $N=10^4$、10^5 和 10^6，用你的程序统计至少 100 棵随机构造的大小为 N 的红黑树并得出一个猜想。

3.3.43 成本图。改造 RedBlackBST 的实现来绘制本节中能够显示计算中每次 put() 操作的成本的图（请参考练习 3.1.38）。

3.3.44 平均查找用时。用实验研究和计算在一棵由 N 个随机结点构造的红黑树中到达一个随机结点的平均路径长度（内部路径长度除以 N 再加 1）的平均差和标准差，对于 1 到 10 000 之间的每个 N 至少重复实验 1000 遍。将结果绘制成和图 3.3.30 相似的 Tufte 图，并画上函数 $\lg N-0.5$ 的曲线。

3.3.45 统计旋转。改进你为练习 3.3.43 给出的程序，用图像绘制出在构造红黑树的过程中旋转和分解结点的次数并讨论结果。

3.3.46 红黑树的高度。改进你为练习 3.3.43 给出的程序，用图像绘制出所有红黑树的高度并讨论结果。



456

图 3.3.30 随机构造的红黑树中到达一个随机结点的平均路径长度（另见彩插）

3.4 散列表

如果所有的键都是小整数，我们可以用一个数组来实现无序的符号表，将键作为数组的索引而数组中键 i 处储存的就是它对应的值。这样我们就可以快速访问任意键的值。在本节中我们将要学习*散列表*。它是这种简易方法的扩展并能够处理更加复杂的类型的键。我们需要用算术操作将键转化为数组的索引来访问数组中的键值对。

使用散列的查找算法分为两步。第一步是用*散列函数*将被查找的键转化为数组的一个索引。理想情况下，不同的键都能转化为不同的索引值。当然，这只是理想情况，所以我们需要面对两个或者多个键都会散列到相同的索引值的情况。因此，散列查找的第二步就是一个*处理碰撞冲突*的过程，如图 3.4.1 所示。在描述了多种散列函数的计算后，我们会学习两种解决碰撞的方法：*拉链法*和*线性探测法*。

散列表是算法在*时间和空间上作出权衡*的经典例子。如果没有内存限制，我们可以直接将键作为（可能是一个超大的）数组的索引，那么所有查找操作只需要访问内存一次即可完成。但这种理想情况不会经常出现，因为当键很多时需要的内存太大。另一方面，如果没有时间限制，我们可以使用无序数组并进行顺序查找，这样就只需要很少的内存。而散列表则使用了适度的空间和时间并在这两个极端之间找到了一种平衡。事实上，我们不必重写代码，只需要调整散列算法的参数就可以在空间和时间之间作出取舍。我们会使用概率论的经典结论来帮助我们选择适当的参数。

概率论是数学分析的重大成果。虽然它不在本书的讨论范围之内，但我们将要学习的散列算法利用了这些知识，这些算法虽然简单但应用广泛。使用散列表，你可以实现在一般应用中拥有（均摊后）*常数级别*的查找和插入操作的符号表。这使得它在很多情况下成为实现简单符号表的最佳选择。

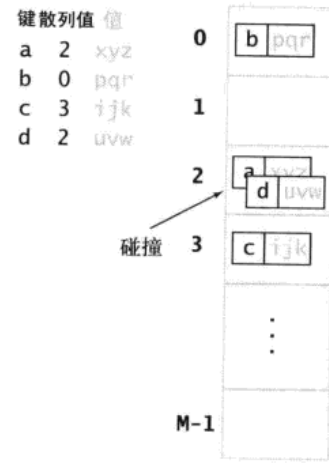


图 3.4.1 散列表的核心问题

457 ~ 458

3.4.1 散列函数

我们面对的第一个问题就是散列函数的计算，这个过程会将键转化为数组的索引。如果我们有一个能够保存 M 个键值对的数组，那么我们就需要一个能够将任意键转化为该数组范围内的索引（[0, M–1] 范围内的整数）的*散列函数*。我们要找的散列函数应该易于计算并且能够均匀分布所有的键，即对于任意键，0 到 M–1 之间的每个整数都有相等的可能性与之对应（与键无关）。这个要求似乎有些难以理解。那么要理解散列，就首先要仔细思考如何去实现这样一个函数。

散列函数和键的类型有关。*严格地说，对于每种类型的键都我们都需要一个与之对应的散列函数*。如果键是一个数，比如社会保险号，我们就可以直接使用这个数；如果键是一个字符串，比如一个人的名字，我们就需要将这个字符串转化为一个数；如果键含有多个部分，比如邮件地址，我们需要用某种方法将这些部分结合起来。对于许多常见类型的键，我们可以利用 Java 提供的默认实现。我们会简略讨论多种数据类型的散列函数。你应该看看它们是如何实现的，因为你也需要为自定义的类型实现散列函数。

3.4.1.1 典型的例子

假设在我们的应用中，键是美国的社会保险号。一个社会保险号含有 9 位数字并被分为三个部

分，例如 123-45-6789。第一组数字表示该号码签发的地区（例如，第一组号码为 035 的社会保险号来自罗得岛州，214 则来自马里兰州），另两组数字表示个人身份。社会保险号共有 10 亿（10^9）个，但假设我们的应用程序只需要处理几百个，我们可以使用一个大小 M=1000 的散列表。散列函数的一种实现方法是使用键（社会保险号）中的三个数字。用第三组中的三个数字似乎比用第一组中的三个数字更好（因为我们的客户不太可能完全平均地分布在各个地区），但下面会讲到，更好的方法是用所有 9 个数字得到一个整数，然后再考虑整数的散列函数。

键	散列值 (M = 100)	散列值 (M = 97)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	**1**	25
900	0	27
413	13	25
701	**1**	22
418	18	30
601	**1**	19

除留余数法

3.4.1.2 正整数

将整数散列最常用方法是*除留余数法*。我们选择大小为素数 M 的数组，对于任意正整数 k，计算 k 除以 M 的余数。这个函数的计算非常容易（在 Java 中为 k%M）并能够有效地 将键散布在 0 到 M–1 的范围内。如果 M 不是素数，我们可能无法利用键中包含的所有信息，这可能导致我们无法均匀地散列散列值。例如，如果键是十进制数而 M 为 10^k，那么我们只能利用键的后 k 位，这可能会产生一些问题。举个简单的例子，假设键为电话号码的区号且 M=100。由于历史原因，美国的大部分区号中间位都是 0 或者 1，因此这种方法会将大量的键散列为小于 20 的索引，但如果使用素数 97，散列值的分布显然会更好（一个离 100 更远的素数会更好），如右侧所示。与之类似，互联网中使用的 IP 地址也不是随机的，所以如果我们想用除留余数法将其散列就需要用素数（2 的幂除外）大小的数组。

3.4.1.3 浮点数

如果键是 0 到 1 之间的实数，我们可以将它乘以 M 并四舍五入得到一个 0 至 M–1 之间的索引值。尽管这个方法很容易理解，但它是有缺陷的，因为这种情况下键的高位起的作用更大，最低位对散列的结果没有影响。修正这个问题的办法是将键表示为二进制数然后再使用除留余数法（Java 就是这么做的）。

3.4.1.4 字符串

除留余数法也可以处理较长的键，例如字符串，我们只需将它们当作大整数即可。例如，右侧的代码就能够用除留余数法计算 String S 的散列值：

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
  hash = (R * hash + s.charAt(i)) % M;
```

散列字符串键

Java 的 charAt() 函数能够返回一个 char 值，即一个非负 16 位整数。如果 R 比任何字符的值都大，这种计算相当于将字符串当作一个 N 位的 R 进制值，将它除以 M 并取余。一种叫 Horner *方法*的经典算法用 N 次乘法、加法和取余来计算一个字符串的散列值。只要 R 足够小，不造成溢出，那么结果就能够如我们所愿，落在 0 至 M–1 之内。使用一个较小的素数，例如 31，可以保证字符串中的所有字符都能发挥作用。Java 的 String 的默认实现使用了一个类似的方法。

3.4.1.5 组合键

如果键的类型含有多个整型变量，我们可以和 String 类型一样将它们混合起来。例如，假设被查找的键的类型是 Date，其中含有几个整型的域：day（两个数字表示的日），month（两个数字表示的月）和 year（4 个数字表示的年）。我们可以这样计算它的散列值：

```
int hash = (((day * R + month) % M ) * R + year) % M;
```

只要 R 足够小不造成溢出，也可以得到一个 0 至 M–1 之间的散列值。在这种情况下我们可以通过选择一个适当的 M，比如 31，来省去括号内的 %M 计算。和字符串的散列算法一样，这个方法也能处理有任意多整型变量的类型。

表 3.4.1 所有例子中的键的散列值

键	S	E	A	R	C	H	X	M	P	L
散列值 (M = 5)	2	0	0	4	4	4	2	4	3	3
散列值 (M = 16)	6	10	4	14	5	4	15	1	14	6

3.4.1.6 Java 的约定

每种数据类型都需要相应的散列函数，于是 Java 令所有数据类型都继承了一个能够返回一个 32 位整数的 hashCode() 方法。每一种数据类型的 hashCode() 方法都必须和 equals() 方法一致。也就是说，如果 a.equals(b) 返回 true，那么 a.hashCode() 的返回值必然和 b.hashCode() 的返回值相同。相反，如果两个对象的 hashCode() 方法的返回值不同，那么我们就知道这两个对象是不同的。但如果两个对象的 hashCode() 方法的返回值相同，这两个对象也有可能不同，我们还需要用 equals() 方法进行判断。请注意，这说明如果你要为自定义的数据类型定义散列函数，你需要同时重写 hashCode() 和 equals() 两个方法。默认散列函数会返回对象的内存地址，但这只适用于很少的情况。Java 为很多常用的数据类型重写了 hashCode() 方法（包括 String、Integer、Double、File 和 URL）。

3.4.1.7 将 hashCode() 的返回值转化为一个数组索引

因为我们需要的是数组的索引而不是一个 32 位的整数，我们在实现中会将默认的 hashCode() 方法和除留余数法结合起来产生一个 0 到 M–1 的整数，方法如下：

```
private int hash(Key x)
{  return (x.hashCode() & 0x7fffffff) % M;  }
```

这段代码会将符号位屏蔽（将一个 32 位整数变为一个 31 位非负整数），然后用除留余数法计算它除以 M 的余数。在使用这样的代码时我们一般会将数组的大小 M 取为*素数*以充分利用原散列值的所有位。*注意*：为了避免混乱，我们在例子中不会使用这种计算方法而是使用表 3.4.1 所示的散列值作为替代。

3.4.1.8 自定义的 hashCode() 方法

散列表的用例希望 hashCode() 方法能够将键平均地散布为所有可能的 32 位整数。也就是说，对于任意对象 x，你可以调用 x.hashCode() 并认为有均等的机会得到 2^{32} 中的任意一个 32 位整数值。Java 中的 String、Integer、Double、File 和 URL 对象的 hashCode() 方法都能实现这一点。而对于自己定义的数据类型，你必须试着自己实现这一点。3.4.1.5 节中的 Date 例子展示了一种可行的方案：用实例变量的整数值和除留余数法得到散列值。在 Java 中，所有的数据类型都继承了 hashCode() 方法，因此还有一个更

```
public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;

   public int hashCode()
   {
      int hash = 17;
      hash = 31 * hash + who.hashCode();
      hash = 31 * hash + when.hashCode();
      hash = 31 * hash
         + ((Double) amount).hashCode();
      return hash;
   }
   ...
}
```

自定义类型中hashCode()方法的实现

简单的做法：将对象中的每个变量的 hashCode() 返回值转化为 32 位整数并计算得到散列值，如 Transaction 类所示。

对于原始类型的对象，可以将其转化为对应的数据类型然后再调用 hashCode() 方法。和以前一样，系数的具体值（这里是 31）并不是很重要。

3.4.1.9 软缓存

如果散列值的计算很耗时，那么我们或许可以将每个键的散列值缓存起来，即在每个键中使用一个 hash 变量来保存它的 hashCode() 的返回值（请见练习 3.4.25）。第一次调用 hashCode() 方法时，我们需要计算对象的散列值，但之后对 hashCode() 方法的调用会直接返回 hash 变量的值。Java 的 String 对象的 hashCode() 方法就使用了这种方法来减少计算量。

总的来说，要为一个数据类型实现一个优秀的散列方法需要满足三个条件：

- 一致性——等价的键必然产生相等的散列值；
- 高效性——计算简便；
- 均匀性——均匀地散列所有的键。

设计同时满足这三个条件的散列函数是专家们的事。有了各种内置函数，Java 程序员在使用散列时只需要调用 hashCode() 方法即可，我们没有理由不信任它们。

但是，在有性能要求时应该谨慎使用散列，因为糟糕的散列函数经常是性能问题的罪魁祸首：程序可以工作但比预想的慢得多。保证均匀性的最好办法也许就是保证键的每一位都在散列值的计算中起到了相同的作用；实现散列函数最常见的错误也许就是忽略了键的高位。无论散列函数的实现是什么，当性能很重要时你应该测试所使用的所有散列函数。计算散列函数和比较两个键，哪个耗时更多？你的散列函数能够将一组键均匀地散布在 0 到 M–1 之间吗？用简单的实现测试这些问题能够预防未来的悲剧。例如，图 3.4.2 就显示出，对于《双城记》我们的 hash() 方法在使用了 Java 的 String 类型的 hashCode() 方法后能够得到一个合理的分布。

459
↓
462

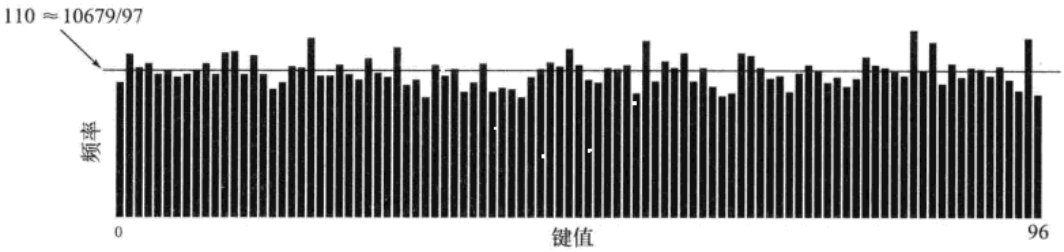


图 3.4.2 《双城记》中每个单词的散列值的出现频率（10 679 个键，即单词，M=97）（另见彩插）

这些讨论的背后是我们在使用散列时作出的一个重要假设。这个假设是一个我们实际上无法达到的理想模型，但它是我们实现散列函数时的指导思想。

假设 J（均匀散列假设）。我们使用的散列函数能够均匀并独立地将所有的键散布于 0 到 M–1 之间。

讨论。我们在实现散列函数时随意指定了很多参数，这显然无法实现一个能够在数学意义上均匀并独立地散布所有键的散列函数。坚深的理论研究告诉我们想要找到一个计算简单但又拥有一致性和均匀性的散列函数是不太可能的。在实际应用中，和使用 Math.random() 生成随机数一样，大多数程序员都会满足于随机数生成器类的散列函数。很少有人会去检验独立性，而这个性质一般都不会满足。

尽管验证这个假设很困难，假设 J 仍然是考察散列函数的重要方式，原因有两点。首先，设计散列函数时尽量避免随意指定参数以防止大量的碰撞，这是我们的重要目标；其次，尽管我们可能无法验证假设本身，它提示我们使用数学分析来预测散列算法的性能并在实验中进行验证。 463

3.4.2 基于拉链法的散列表

一个散列函数能够将键转化为数组索引。散列算法的第二步是*碰撞处理*，也就是处理两个或多个键的散列值相同的情况。一种直接的办法是将大小为 M 的数组中的每个元素指向一条链表，链表中的每个结点都存储了散列值为该元素的索引的键值对。这种方法被称为*拉链法*，因为发生冲突的元素都被存储在链表中。这个方法的基本思想就是选择足够大的 M，使得所有链表都尽可能短以保证高效的查找。查找分两步：首先根据散列值找到对应的链表，然后沿着链表顺序查找相应的键。

拉链法的一种实现方法是使用原始的链表数据类型（请见练习 3.4.2）来扩展 SequentialSearchST（算法 3.1）。另一种更简单的方法（但效率稍低）是采用一般性的策略，为 M 个元素分别构建符号表来保存散列到这里的键，这样也可以重用我们之前的代码。算法 3.5 实现的 SeparateChainingHashST 使用了一个 SequentialSearchST 对象的数组，在 put() 和 get() 的实现中先计算散列函数来选定被查找的 SequantialSearchST 对象，然后使用符号表的 put() 和 get() 方法来完成相应的任务。

因为我们要用 M 条链表保存 N 个键，无论键在各个链表中的分布如何，链表的平均长度肯定是 N/M。例如，假设所有的键都落在了第一条链表上，所有链表的平均长度仍然是 $(N+0+0+\cdots+0)/M=N/M$。拉链法在实际情况中很有用，因为每条链表确实都大约含有 N/M 个键值对。在一般情况中，我们能够由它验证假设 J 并且可以依赖这种高效的查找和插入实现。

在标准索引用例中使用基于拉链法的散列表如图 3.4.3 所示。

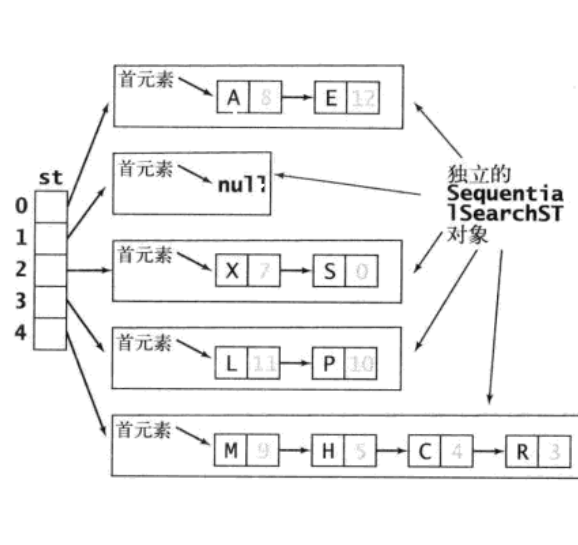


图 3.4.3 标准索引用例使用基于拉链法的散列表 464

算法 3.5 基于拉链法的散列表

```
public class SeparateChainingHashST<Key, Value>
{
   private int N;                                  // 键值对总数
   private int M;                                  // 散列表的大小
   private SequentialSearchST<Key, Value>[] st;  // 存放链表对象的数组

   public SeparateChainingHashST()
   {  this(997);  }
```

```
   public SeparateChainingHashST(int M)
   {  // 创建M条链表
      this.M = M;
      st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
      for (int i = 0; i < M; i++)
         st[i] = new SequentialSearchST();
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M; }

   public Value get(Key key)
   {  return (Value) st[hash(key)].get(key);  }

   public void put(Key key, Value val)
   {  st[hash(key)].put(key, val);  }

   public Iterable<Key> keys()
   // 请见练习3.4.19

}
```

这段简单的符号表实现维护着一条链表的数组，用散列函数来为每个键选择一条链表。简单起见，我们使用了 SequentialSearchST。在创建 st[] 时需要进行类型转换，因为 Java 不允许泛型的数组。默认的构造函数会使用 997 条链表，因此对于较大的符号表，这种实现比 SequentialSearchST 大约会快 1000 倍。当你能够预知所需要的符号表的大小时，这段短小精悍的方案能够得到不错的性能。一种更可靠的方案是动态调整链表数组的大小，这样无论在符号表中有多少键值对都能保证链表较短（请见 3.4.4 节及练习 3.4.18）。

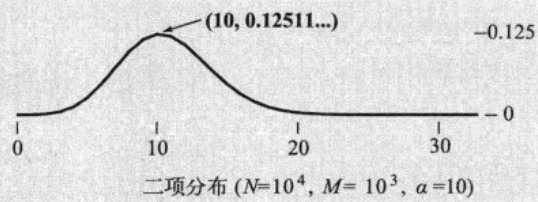
465

命题 K。在一张含有 M 条链表和 N 个键的的散列表中，（在假设 J 成立的前提下）任意一条链表中的键的数量均在 N/M 的常数因子范围内的概率无限趋向于 1。

简略的证明。有了假设 J，这个问题就变成了一个经典的概率论问题。在这里我们为有一些概率论基础知识的读者给出一个简要的证明。

由二项分布可知，一条给定的链表正好含有 k 个键的概率为：

$$\binom{N}{k}\left(\frac{1}{M}\right)^{k}\left(\frac{M-1}{M}\right)^{N-k}$$



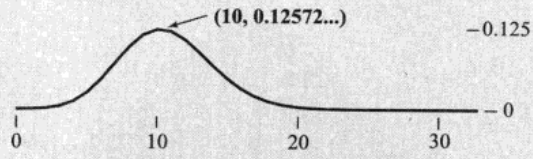
二项分布 (N=10^4, M= 10^3, α =10)

因为我们实际上是从 N 个键中取了其中 k 个。这 k 个键被散列到给定的链表的概率均为 $1/M$，而剩下的 $(N–k)$ 个键不被散列到给定的链表中的概率均为 $(1–1/M)$。令 $\alpha=N/M$，这个公式可以写为：

$$\binom{N}{k}\left(\frac{\alpha}{N}\right)^{k}\left(1-\frac{\alpha}{N}\right)^{N-k}$$

对于较小的 α，经典的**泊松分布**可以非常近似地表示它：

$$\frac{\alpha^k e^{-\alpha}}{k!}$$



泊松分布 ($N = 10^4, M = 10^3, \alpha = 10$)

由此可得，一条链表中含有超过 $t\alpha$ 个键的概率不会超过 $(\alpha e/t)^t e^{-\alpha}$。对于实际应用来说，这个数字非常小。例如，如果平均链表长度为 10，那么一个键的散列值落在一条长度超过 20 的链表的概率不超过 $(10e/2)^2 e^{-10} \approx 0.0084$；如果平均链表长度为 20，那么一个键的散列值落在一条长度超过 40 的链表的概率不超过 $(20\,e/2)^2 e^{-20} \approx 0.000\,001\,6$。这个结果并不能保证每条链表都很短，但我们可以知道当 α 一定时，最长链表的平均长度的增长速度为 $\log N/\log\log N$。 466

这段数学分析非常有力，但需要注意的是它完全依赖于假设 J。如果散列函数不是均匀和独立的，那么查找和插入的成本就可能和 N 成正比，也就是和顺序查找类似。假设 J 比我们见过的其他和概率有关的算法中相应的假设都有效，但也更加难以验证。在计算散列值时，我们假设每个键都有均等的机会被散列到 M 个索引中的任意一个，无论键有多复杂。我们没法用实验来验证所有可能的数据类型，所以我们会进行更复杂的实验，在实际应用中可能出现的一组键中随机取样进行验证，然后统计结果并分析。好消息是我们在测试中仍然可以使用这个算法来验证假设 J 和由它得出的数学推论。

性质 L。在一张含有 M 条链表和 N 个键的的散列表中，未命中查找和插入操作所需的比较次数为 $\sim N/M$。

例证。在实际应用中，散列表算法的高性能并不需要散列函数完全符合假设 J 意义上的均匀性。自 20 世纪 50 年代以来，无数程序员都见证了命题 K 所预言的性能改进，即使有些散列函数不是均匀的，命题也成立。例如，图 3.4.4 所示的 `FrequencyCounter` 使用的散列表（其中的 `hash()` 方法是基于 Java 的 `String` 类型的 `hashCode()` 方法）中的链表长度和理论模型完全一致。这条性质的例外之一是在许多情况下散列函数未能使用键的所有信息而造成的性能低下。除此之外，大量经验丰富的程序员给出的应用实例令我们确信，在基于拉链法的散列表中使用大小为 M 的数组能够将查找和插入操作的效率提高 M 倍。

3.4.2.1 散列表的大小

在实现基于拉链法的散列表时，我们的目标是选择适当的数组大小 M，既不会因为空链表而浪费大量内存，也不会因为链表太长而在查找上浪费太多时间。而拉链法的一个好处就是这并不是关键性的选择。如果存入的键多于预期，查找所需的时间只会比选择更大的数组稍长；如果少于预期，虽然有些空间浪费但查找会非常快。当内存不是很紧张时，可以选择一个足够大的 M，使得查找需要的时间变为常数；当内存紧张时，选择尽量大的 M 仍然能够将性能提高 M 倍。例如对于 `FrequencyCounter`，从图 3.4.4 可以看出，每次操作所需要的比较次数从使用 `SequentialSearchST` 时的上千次降低到了使用 `SeparateChainingHashST` 时的若干次，正如我 467

们所料。另一种方法是动态调整数组的大小以保持短小的链表（请见练习 3.4.18）。

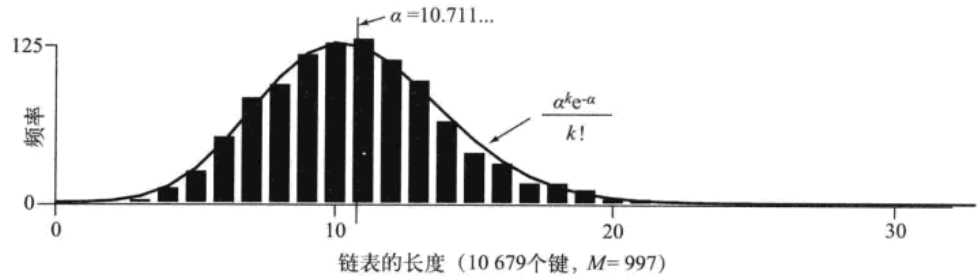


图 3.4.4 使用 SeparateChainingHashST，运行 java FrequencyCounter 8 < tale.txt 时所有链表的长度（另见彩插）

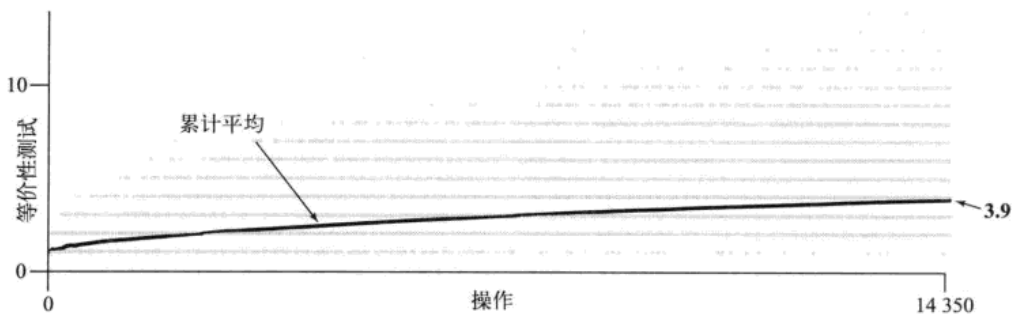
3.4.2.2 删除操作

要删除一个键值对，先用散列值找到含有该键的 SequentialSearchST 对象，然后调用该对象的 delete() 方法（请见练习 3.1.5）。这种重用已有代码的方式比重新实现链表的删除更好。

3.4.2.3 有序性相关的操作

散列最主要的目的在于均匀地将键散布开来，因此在计算散列后键的顺序信息就丢失了，如图3.4.5所示。如果你需要快速找到最大或者最小的键，或是查找某个范围内的键，或是实现表 3.1.4 中有序符号表 API 中的其他任何方法，散列表都不是合适的选择，因为这些操作的运行时间都将会是线性的。

基于拉链法的散列表的实现简单。在键的顺序并不重要的应用中，它可能是最快的（也是使用最广泛的）符号表实现。当使用 Java 的内置数据类型作为键，或是在使用含有经过完善测试的 hashCode() 方法的自定义类型作为键时，算法 3.5 能够提供快速而方便的查找和插入操作。下面，我们会介绍另一种解决碰撞冲突的有效方法。



468 图 3.4.5 使用 SeparateChainingHashST，运行 java FrequencyCounter 8 < tale.txt 的成本（M=997）

3.4.3 基于线性探测法的散列表

实现散列表的另一种方式就是用大小为 M 的数组保存 N 个键值对，其中 $M>N$。我们需要依靠数组中的空位解决碰撞冲突。基于这种策略的所有方法被统称为*开放地址散列表*。

开放地址散列表中最简单的方法叫做*线性探测法*：当碰撞发生时（当一个键的散列值已经被另一个不同的键占用），我们直接检查散列表中的下一个位置（将索引值加 1）。这样的线性探测可能会产生三种结果：

- 命中，该位置的键和被查找的键相同；
- 未命中，键为空（该位置没有键）；
- 继续查找，该位置的键和被查找的键不同。

我们用散列函数找到键在数组中的索引，检查其中的键和被查找的键是否相同。如果不同则继续查找（将索引增大，到达数组结尾时折回数组的开头），直到找到该键或者遇到一个空元素，如图 3.4.6 所示。我们习惯将检查一个数组位置是否含有被查找的键的操作称作探测。在这里它可以等价于我们一直使用的比较，不过有些探测实际上是在测试键是否为空。

开放地址类的散列表的核心思想是与其将内存用作链表，不如将它们作为在散列表的空元素。这些空元素可以作为查找结束的标志。在 LinearProbingHashST 中可以看到（算法 3.6），使用这种思想来实现符号表的 API 是十分简单的。我们在实现中使用了并行数组，一条保存键，一条保存值，并像前面讨论的那样使用散列函数产生访问数据所需的数组索引。

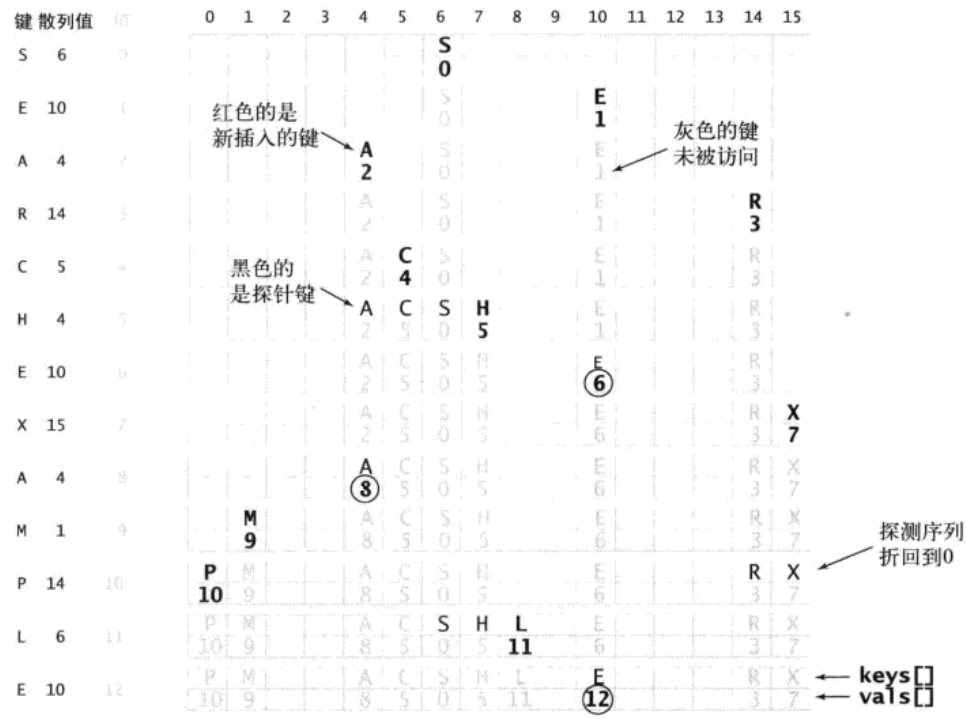


图 3.4.6 标准索引用例使用的基于线性探测的符号表的轨迹（另见彩插） 469

算法 3.6 基于线性探测的符号表

```
public class LinearProbingHashST<Key, Value>
{
   private int N;          // 符号表中键值对的总数
   private int M = 16;     // 线性探测表的大小
   private Key[] keys;     // 键
   private Value[] vals;   // 值
   public LinearProbingHashST()
   {
      keys = (Key[])   new Object[M];
```

```
      vals = (Value[]) new Object[M];
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M; }

   private void resize()         // 请见3.4.4节

   public void put(Key key, Value val)
   {
      if (N >= M/2) resize(2*M);  // 将M加倍（请见正文）

      int i;
      for (i = hash(key); keys[i] != null; i = (i + 1) % M)
         if (keys[i].equals(key)) { vals[i] = val; return; }
      keys[i] = key;
      vals[i] = val;
       N++;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
         if (keys[i].equals(key))
             return vals[i];
      return null;
   }
}
```

这段符号表的实现将键和值分别保存在两个数组（BinarySearchST类型）中，使用空（标记为null）来表示一簇键的结束。如果一个新键的散列值是一个空元素，那么就将它保存在那里；如果不是，我们就顺序查找一个空元素来保存它。要查找一个键，我们从它的散列值开始顺序查找，如果找到则命中，
470 如果遇到空元素则未命中。keys() 方法的实现请见练习 3.4.19。

3.4.3.1 删除操作

如何从基于线性探测的散列表中删除一个键？仔细想一想，你会发现直接将该键所在的位置设为null是不行的，因为这会使得在此位置之后的元素无法被查找。例如，假设在轨迹图的例子中（图3.4.6）我们需要用这种方法删除键C，然后查找H。H的散列值是4，但它实际存储在这一簇键的结尾，即7号位置。如果我们将5号位置设为null，get() 方法将无法找到H。因此，我们需要将簇中被删除键的右侧的所有键重新插入散列表。这个过程比想象的要复杂，所以你最好以练习（请见练习3.4.17）为例跟踪右侧这段代码的运行全过程。

和拉链法一样，开放地址类的散列表的性能也依赖于 $\alpha = N/M$ 的比值，但意义有所不同。我们将 α 称为散列表的使用率。对于基于拉链法的散列表，α 是每条链表的长度，因此一般大于1；

```
public void delete(Key key)
{
   if (!contains(key)) return;
   int i = hash(key);
   while (!key.equals(keys[i]))
      i = (i + 1) % M;
   keys[i] = null;
   vals[i] = null;
   i = (i + 1) % M;
   while (keys[i] != null)
   {
      Key   keyToRedo = keys[i];
      Value valToRedo = vals[i];
      keys[i] = null;
      vals[i] = null;
      N--;
      put(keyToRedo, valToRedo);
      i = (i + 1) % M;
   }
   N--;
   if (N > 0 && N == M/8) resize(M/2);
}
```

基于线性探测的散列表的删除操作

对于基于线性探测的散列表，α 是表中已被占用的空间的比例，它是不可能大于 1 的。事实上，在 LinearProbingHashST 中我们不允许 α 达到 1（散列表被占满），因为此时未命中的查找会导致无限循环。为了保证性能，我们会动态调整数组的大小来保证使用率在 1/8 到 1/2 之间。这个策略是基于数学上的分析，我们会在讨论实现的细节之前介绍。 471

3.4.3.2 键簇

线性探测的平均成本取决于元素在插入数组后聚集成的一组连续的条目，也叫做键簇，如图 3.4.7 所示。例如，在示例中插入键 C 会产生一个长度为 3 的键簇（A C S）。这意味着插入 H 需要探测 4 次，因为 H 的散列值为该键簇的第一个位置。显然，短小的键簇才能保证较高的效率。随着插入的键越来越多，这个要求很难满足，较长的键簇会越来越多，如图 3.4.8 所示。另外，因为（基于均匀性假设）数组的每个位置都有相同的可能性被插入一个新键，长键簇更长的可能性比短键簇更大，因为新键的散列值无论落在簇中的任何位置都会使簇的长度加 1（甚至更多，如果这个簇和相邻的簇之间只有一个空元素相隔的话）。下面我们要将键簇的影响量化来预测线性探测法的性能，并使用这些信息在我们的实现中设置适当的参数值。

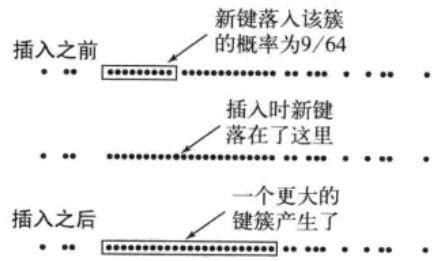


图 3.4.7 线性探测法中的键簇（M=64）

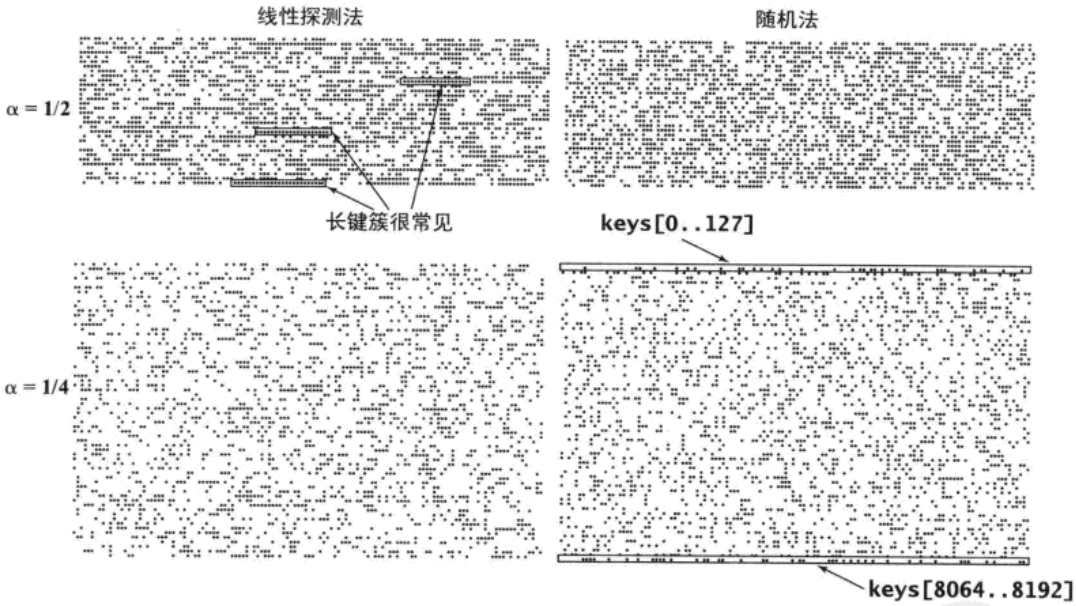


图 3.4.8 数组的使用模式（2048 个键，每行 128 个） 472

3.4.3.3 线性探测法的性能分析

尽管最后的结果的形式相对简单，准确分析线性探测法的性能是非常有难度的。Knuth 在 1962 年作出的以下推导是算法分析史上的一个里程碑。

命题 M。在一张大小为 M 并含有 $N=\alpha M$ 个键的基于线性探测的散列表中，基于假设 J，命中和未命中的查找所需的探测次数分别为：

$$\sim\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)\text{和}\sim\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

特别是当 α 约为 1/2 时，查找命中所需要的探测次数约为 3/2，未命中所需要的约为 5/2。当 α 趋近于 1 时，这些估计值的精确度会下降，但不需要担心这些情况，因为我们会保证散列表的使用率小于 1/2。

讨论。要计算平均值，首先要计算在散列表中每个位置上出现查找未命中所需要的探测次数，然后将所有探测次数之和除以 M。所有查找未命中都至少需要一次探测，因此我们从第一次探测之后开始计数。考虑在一张半满的（M=2N）线性探测散列表中可能出现的以下两种极端情况：在最好的情况下，偶数位置的数组元素都是空的，奇数位置的数组元素都是满的；在最坏的情况下，前半张表是空的，后半张表是满的。键簇的平均长度在两种情况下都是 $N/(2N)=1/2$，但未命中的查找所需的探测次数在最好情况下为 1（所有的查找都至少需要一次探测）加上 $(0+1+0+1+\cdots)/(2N)=1/2$，在最坏情况下为 $1+(N+(N-1)+\cdots)/(2N)\sim N/4$。将这段证明一般化可得未命中的查找平均所需的比较次数和键簇长度的*平方*成正比。如果一个键簇的长度为 t，那么 $(t+(t-1)+\cdots+2+1)/M=t(t+1)/(2M)$ 就是在这段键簇中查找未命中所需的平均探测次数。因为所有键簇的总长度肯定为 N，所以将表中所有键簇所得的平均探测次数相加可以得到，一次未命中的查找的平均成本为 $1+N/(2M)+$（每个键簇的长度的平方之和），再除以 $2M$。因此，给定一张散列表，我们就可以快速计算该表中一次未命中查找的平均成本（请见练习 3.4.21）。一般情况下，键簇的形成需要一个复杂的动态过程（也就是线性探测算法），很难分析并找出特点，而且这也远远超出了本书的讨论范围。

473

命题 M 告诉我们（在假设 J 的前提下）当散列表快满的时候查找所需的探测次数是巨大的（α 越趋近于 1，由公式可知探测的次数也越来越大），但当使用率 α 小于 1/2 时探测的预计次数只在 1.5 到 2.5 之间。下面，我们为此来考虑动态调整散列表数组的大小。

3.4.4 调整数组大小

我们可以使用第 1 章中介绍的调整数组大小的方法来保证散列表的使用率永远都不会超过 1/2。首先，我们的 LinearProbingHashST 需要一个新的构造函数，它接受一个固定的容量作为参数（在算法 3.6 的构造函数中加入一行代码就可以在创建数组之前将 M 设为给定的值）。然后，我们需要右边给出的 resize() 方法。

```
private void resize(int cap)
{
   LinearProbingHashST<Key, Value> t;
   t = new LinearProbingHashST<Key, Value>(cap);
   for (int i = 0; i < M; i++)
      if (keys[i] != null)
         t.put(keys[i], vals[i]);
   keys = t.keys;
   vals = t.vals;
   M    = t.M;
}
```

调整线性探测散列表

它会创建一个新的给定大小的 LinearProbingHashST，保存原表中的 keys 和 values 变量，然后将原表中所有的键重新散列并插入到新表中。这使我们可以将数组的长度加倍。put() 方法中的第一条语句会调用 resize() 来保证散列表最多为半满状态。这段代码构造的散列表比原来大一倍，因此 α 的值就会减半。和其他需要调整数组大小的应用场景一样，我们也需要在 delete() 方法的最后加上：

```
if (N > 0 && N <= M/8) resize(M/2);
```

以保证所使用的内存量和表中的键值对数量的比例总在一定范围之内。动态调整数组大小可以为我们保证 α 不大于 1/2。

3.4.4.1 拉链法

我们可以用相同的方法在拉链法中保持较短的链表（平均长度在 2 到 8 之间）：在 resize() 中将 LinearProbingHashST 替换为 SeparateChainingHashST，当 N >= M/2 时调用 resize(2*M)，并在 delete() 中（在 N > 0 &&N <= M/8 时）调用 resize(M/2)。对于拉链法，如果你能准确地估计用例所需的散列表的大小 N，调整数组的工作并不是必需的，只需要根据查找耗时和（$1+N/M$）成正比来选取一个适当的 M 即可。而对于线性探测法，调整数组的大小是必需的，因为当用例插入的键值对数量超过预期时它的查找时间不仅会变得非常长，还会在散列表被填满时进入无限循环。 474

3.4.4.2 均摊分析

从理论角度来说，当我们动态调整数组大小时，需要找出均摊成本的上限，因为我们知道使散列表长度加倍的插入操作需要大量的探测。

命题 N。假设一张散列表能够自己调整数组的大小，初始为空。基于假设 J，执行任意顺序的 t 次查找、插入和删除操作所需的时间和 t 成正比，所使用的内存量总是在表中的键的总数的常数因子范围内。

证明。对于拉链法和线性探测法，结合命题 K 和命题 M 可知，这个命题只是对我们在第 1 章中第一次讨论过的数组增长的均摊分析的简单重复而已。

如图 3.4.9 和图 3.4.10 所示，在 FrequencyCounter 的例子中，累计平均的曲线很好地显示出散列表中调整数组大小的动态行为。每次数组长度加倍之后，累计平均值都会增加约 1，因为表中的每个键都需要重新计算散列值。然后该值慢慢下降，因为半数左右的键被重新分配到了表中的不同位置。随着表中的键的增加，该值下降的速度也慢慢降低。

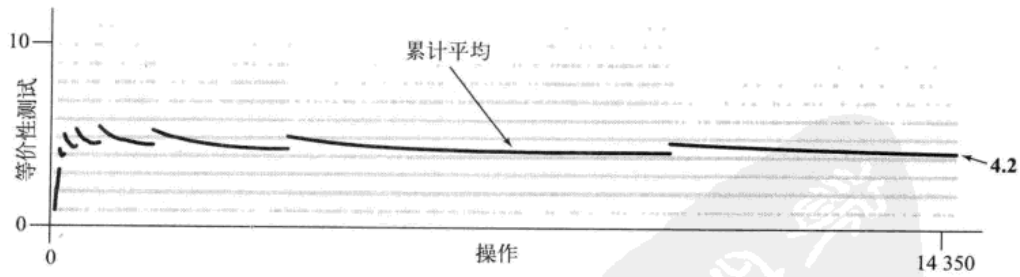
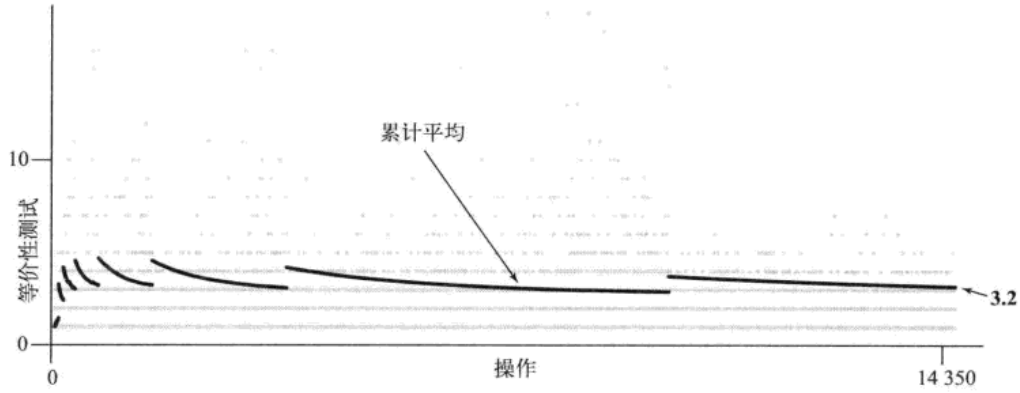


图 3.4.9 使用能够自动调整数组大小的 SeparateChainingHashST，运行 java FrequencyCounter 8< tale.txt 的成本



475 图 3.4.10 使用能够自动调整数组大小的 LinearProbingHashST，运行 java FrequencyCounter 8 < tale.txt 的成本

3.4.5 内存使用

我们说过，如果我们希望将散列表的性能调整到最优，理解它的内存使用情况是非常重要的。虽然这种调整是专家们的事儿，但通过估计引用的使用数量来粗略计算所需的内存量仍然是很好的练习。方法如下：除了存储键和值所需的空间之外，我们实现的 SeparateChainingHashST 保存了 M 个 SequentialSearchST 对象和它们的引用。每个 SequentialSearchST 对象需要 16 字节，它的每个引用需要 8 字节。另外还有 N 个 node 对象，每个都需要 24 字节以及 3 个引用（key、value 和 next），比二叉查找树的每个结点还多需要一个引用。在使用动态调整数组大小来保证表的使用率在 1/8 到 1/2 之间的情况下，线性探测使用 $4N$ 到 $16N$ 个引用。可以看出，根据内存用量来选择散列表的实现并不容易。对于原始数据类型，这些计算又有所不同（请见练习 3.4.24）。

符号表的内存使用如表 3.4.2 所示。

表 3.4.2 符号表的内存使用

方 法	N 个元素所需的内存（引用类型）
基于拉链法的散列表	~48N+32M
基于线性探测的散列表	在 ~32N 和 ~128N 之间
476 各种二叉查找树	~56N

自计算机发展的伊始，研究人员就研究了（并且现在仍在继续研究）散列表并找到了很多方法来改进我们所讨论过的几种基本算法。你能找到大量关于这个主题的文献。大多数改进都能降低时间 - 空间的曲线：在查找耗时相同的情况下使用更少的空间，或使在使用相同空间的情况下进行更快的查找。其他方法包括提供更好的性能保证，如最坏情况下的查找成本；改进散列函数的设计等。我们会在练习中讨论其中的部分方法。

拉链法和线性探测法的详细比较取决于实现的细节和用例对空间和时间的要求。即使基于性能考虑，选择拉链法而非线性探测法也不一定是合理的（请见练习 3.5.31）。在实践中，两种方法的性能差别主要是因为拉链法为每个键值对都分配了一小块内存而线性探测则为整张表使用了两个很大的数组。对于非常大的散列表，这些做法对内存管理系统的要求也很不相同。在现代系统中，在性能优先的情景下，最好由专家去把握这种平衡。

有了这些假设，期望散列表能够支持和数组大小无关的常数级别的查找和插入操作是可能的。对于任意的符号表实现，这个期望都是理论上的最优性能。但散列表并非包治百病的灵丹妙药，因为：

- 每种类型的键都需要一个优秀的散列函数；
- 性能保证来自于散列函数的质量；
- 散列函数的计算可能复杂而且昂贵；
- 难以支持有序性相关的符号表操作。

在考察了这些基本问题之后，我们会在 3.5 节的开头将散列表和我们学习过的其他符号表的实现方法进行比较。

477

答疑

问 Java 的 `Integer`、`Double` 和 `Long` 类型的 `hashCode()` 方法是如何实现的？

答 `Integer` 类型会直接返回该整数的 32 位值。对于 `Double` 和 `Long` 类型，Java 会返回值的机器表示的前 32 位和后 32 位异或的结果。这些方法可能不够随机，但它们的确能够将值散列。

问 当能够动态调整数组大小时，散列表的大小总是 2 的幂，这不是个问题吗？这样 `hash()` 方法就只使用了 `hashCode()` 返回值的低位。

答 是的，这个问题在默认实现中特别明显。解决这个问题的一种方法是先用一个大于 `M` 的素数来散列键值对，例如：

```
private int hash(Key x)
{
   int t = x.hashCode() & 0x7fffffff;
   if (lgM < 26) t = t % primes[lgM+5];
   return t % M;
}
```

这段代码假设我们使用了一个变量 lg$M$，它的值等于 lg$M$（直接初始化为该值，并在将数组长度加倍或者减半时增大或者减小它），以及一个数组 `primes[]`，其中含有大于各个 2 的幂的最小素数（请见右表[①]）。代码中的常数 5 是随意取的一个值——我们希望第一次取余操作（%）能够将所有值散列在小于该素数的范围之内，而第二次取余操作则将其中的 5 个值映射到小于 `M` 的所有值中。请注意，对于很大的 `M` 这是没有意义的。

k	δ_k	`primes[k]` ($2^k - \delta_k$)
5	1	31
6	3	61
7	1	127
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859
27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

将散列表大小设为素数

478

问 我忘记了，为什么不将 `hash(x)` 实现为 `x.hashCode() % M`？

答 散列值必须在 0 到 `M`–1 之间，而在 Java 中，取余（`%`）的结果可能是负数。

问 那为什么不将 `hash(x)` 实现为 `Math.abs(x.hashCode()) % M`？

答 问得好，不幸的是对于最大的整数 `Math.abs()` 会返回一个负值。对于许多典型情况，这种溢出不会造成什么问题，但对于散列表这可能使你的程序在几十亿次插入之后崩溃，这很难说。例如，Java 中字符串 `"polygenelubricants"` 的散列值为 -2^{31}。找出散列值为这个数（以及为 0）的其他字符串已经变成了一种有趣的算法谜题。

① 这里似乎和表的内容不相符，表中prime[k]的值是小于2^k的最大素数。——译者注

问 在算法 3.5 中为什么使用 SequentialSearchST 而非 BinarySearchST 或者 RedBlackBST？

答 一般来说，我们希望散列到每个索引值上的键越少越好，而对于小规模符号表初级实现的性能一般更好。在某些情况下，使用这些复杂的实现也许能够稍稍将性能提高，但最好让专家来进行这种调优。

问 散列表的查找比红黑树更快吗？

答 这取决于键的类型，它决定了 hashCode() 的计算成本是否大于 compareTo() 的比较成本。对于常见的键类型以及 Java 的默认实现，这两者的成本是近似的，因此散列表会比红黑树快得多，因为它所需的操作次数是固定的。但需要注意的是，如果要进行有序性相关的操作，这个问题就没有意义了，因为散列表无法高效地支持这些操作。进一步的讨论请见 3.5 节。

问 为什么不能让基于线性探测的散列表充满四分之三？

答 没什么特别的原因。你可以选择任意的 α 值并用命题 M 来估计相应的查找成本。对于 α=3/4，查找命中的平均成本为 2.5，未命中的为 8.5。但如果你允许 α 增长到 7/8，查找未命中的平均成本就会达到 32.5，这可能已经超出了你的承受能力。随着 α 趋近于 1，命题 M 得出的估计值的准确度会下降，但你不应该使散列表的占有率达到那种程度。

479

练习

3.4.1 将键 E A S Y Q U T I O N 依次插入一张初始为空且含有 M=5 条链表的基于拉链法的散列表中。使用散列函数 11 k % M 将第 k 个字母散列到某个数组索引上。

3.4.2 重新实现 SeparateChainingHashST，直接使用 SequentialSearchST 中链表部分的代码。

3.4.3 修改你为上一道练习给出的实现，为每个键值对添加一个整型变量，将其值设为插入该键值对时散列表中元素的数量。实现一个方法，将该变量的值大于给定整数 k 的键（及其相应的值）全部删除。注意：这个额外的功能在为编译器实现符号表时很有用。

3.4.4 使用散列函数 (a * k) % M 将 S E A R C H X M P L 中的第 k 个键散列为一个数组索引。编写一段程序找出 a 和最小的 M，使得该散列函数得到的每个索引都不相同（没有碰撞）。这样的函数也被称为*完美散列函数*。

3.4.5 下面这段 hashCode() 的实现合法吗？

```
public int hashCode()
{ return 17; }
```

如果合法，请描述它的使用效果，否则请解释原因。

3.4.6 假设键为 t 位整数。对于一个使用素数 M 的除留余数法的散列函数，请证明对于键的每一位，都存不同的两个键，它们的散列值只有该位不同。

3.4.7 考虑对于整型的键将除留余数法的散列函数实现为 (a * k) % M，其中 a 为一个任意的固定素数。这样是否足以利用键的所有位使得我们可以使用一个非素数 M 了呢？

3.4.8 对于 N=10、10^2、10^3、10^4、10^5 和 10^6，请估计将 N 个键插入一张 SeparateChainingHashST 的散列表后还剩多少空链表？提示：参考练习 2.5.31。

480

3.4.9 为 SeparateChainingHashST 实现一个即时的 delete() 方法。

3.4.10 将键 E A S Y Q U T I O N 依次插入一张初始为空且大小为 M=16 的基于线性探测法的散列表中。使用散列函数 11 k % M 将第 K 个字母散列到某个数组索引上。对于 M=10 将本题重新完成一遍。

3.4.11 将键 E A S Y Q U T I O N 依次插入一张初始为空大小为 M=4 的基于线性探测法的散列表中，数组只要达到半满即自动将 K 长度加倍。使用散列函数 11 k % M 将第 k 个字母散列到某个数组索引上。给出得到的散列表的内容。

3.4.12 设有键 A 到 G，散列值如下所示。将它们按照一定顺序插入到一张初始为空大小为 7 的基于线性探测的散列表中（这里数组的大小不会动态调整）。下面哪个选项是不可能由插入这些键产生的？给出这些键在构造散列表时可能所需的最大和最小探测次数，并给出相应的插入顺序来证明你的答案。

a. E F G A C B D
b. C E B G F D A
c. B D F A C E G
d. C G B A D E F
e. F G B D A C E
f. G E C A D B F

键	A	B	C	D	E	F	G
散列值（M=7）	2	0	0	4	4	4	2

3.4.13 在下面哪些情况中基于线性探测的散列表中的一次随机的命中查找所需的时间是线性的？

a. 所有键均被散列到同一个索引上
b. 所有键均被散列到不同的索引上
c. 所有键均被散列到同一个偶数索引上
d. 所有键均被散列到不同的偶数索引上

3.4.14 对于未命中的查找回答上一道练习的问题，假设被查找的键被散列到表中任意位置的可能性均等。

3.4.15 在最坏情况下，向一张初始为空、基于线性探测法并能够动态调整数组大小的散列表中插入 N 个键需要多少次比较？

3.4.16 假设有一张大小为 10^6 的基于线性探测的散列表已经半满了，被占用的元素随机分布。请估计所有索引值中能够被 100 整除的位置都被占用的概率。 481

3.4.17 使用 3.4.3.1 节的 delete() 方法从标准索引测试用例使用的 LinearProbingHashST 中删除键 C 并给出结果散列表的内容。

3.4.18 为 SeparateChainingHashST 添加一个构造函数，使用例能够指定查找操作可以接受的在链表中进行的平均探测次数。动态调整数组的大小以保证链表的平均长度小于该值，并使用答疑中所述的方法来保证 hash() 方法的系数总是素数。

3.4.19 为 SeparateChainingHashST 和 LinearProbingHashST 实现 keys() 方法。

3.4.20 为 LinearProbingHashST 添加一个方法来计算一次命中查找的平均成本，假设表中每个键被查找的可能性相同。

3.4.21 为 LinearProbingHashST 添加一个方法来计算一次未命中查找的平均成本，假设使用了一个随机的散列函数。请注意：要解决这个问题并不一定要计算所有的散列函数。

3.4.22 为下列数据类型实现 hashCode() 方法：Point2D、Interval、Interval2D 和 Date。

3.4.23 对于字符串类型的键，考虑 R = 256 和 M = 255 的除留余数法的散列函数。请证明这是一个糟糕的选择，因为任意排列的字母所得字符串的散列值均相同。

3.4.24 对于 double 类型，分析拉链法、线性探测法和二叉查找树的内存使用情况。将结果整理成类似于表 3.4.2 的表格。

482

提高题

3.4.25 *散列值的缓存*。修改 3.4.19 节的 Transaction 类并维护一个变量 hash，在 hashCode() 方法第一次为一个对象计算散列值后将值保存在 hash 中，这样随后的调用就不必重新计算了。*请注意*：这种方法仅适用于不可变的数据类型。

3.4.26 *线性探测法中的延时删除*。为 LinearProbingHashST 添加一个 delete() 方法，在删除一个键值对时将其值设为 null，并在调用 resize() 方法时将键值对从表中删除。这种方法的主要难点在于决定何时应该调用 resize() 方法。*请注意*：如果后来的 put() 方法为该键指定了一个新的值，你应该用新值将 null 覆盖掉。你的程序在决定扩张或者收缩数组时不但要考虑到数组的空元素，也要考虑到这种*死掉*的元素。

3.4.27 *二次探测*。修改 SeparateChainingHashST，进行二次散列并选择两条链表中的较短者。将键 E A S Y Q U T I O N 依次插入一张初始为空且大小为 M=3 的基于拉链法的散列表中，以 11 k % M 作为第一个散列函数，17 k % M 作为第二个散列函数来将第 k 个字母散列到某个数组索引上。给出插入过程的轨迹以及随机的命中查找和未命中查找在该符号表中所需的平均探测次数。

3.4.28 *二次散列*。修改 LinearProbingHashST，进行二次散列以得到探测起始点。确切地说，是将（所有的）(i + 1) % M 替换为 (i + k) % M，其中 k 是一个非零、和 M 互质且和键相关的整数。*提示*：可以令 M 为素数来满足互质的条件。使用上一道练习中给出的两个散列函数，将键 E A S Y Q U T I O N 依次插入一张初始为空且大小为 M=11 的基于线性探测的散列表中。给出插入过程的轨迹以及随机的命中查找和未命中查找所需的平均探测次数。

3.4.29 *删除操作*。分别为前两题中所述的散列表实现即时的 delete() 方法。

3.4.30 *卡方值*（chi—square statistic）。为 SeparateChainingHashST 添加一个方法来计算散列表的 χ^2。对于大小为 M 并含有 N 个元素的散列表，这个值的定义为：

483

$$\chi^2 = (M/N)((f_0-N/M)^2+(f_1-N/M)^2+\cdots+(f_{M-1}-N/M)^2)$$

其中，f_i 为散列值为 i 的键的数量。这个统计数据是检测我们的散列函数产生的随机值是否满足假设的一种方法。如果满足，对于 $N>cM$，这个值落在 $M-\sqrt{M}$ 和 $M+\sqrt{M}$ 之间的概率为 $1-1/c$。

3.4.31 *Cuckoo 散列函数*。实现一个符号表，在其中维护两张散列表和两个散列函数。一个给定的键只能存在于一张散列表之中。在插入一个新键时，在其中一张散列表中插入该键。如果这张表中该键的位置已经被占用了，就用新键替代老键并将老键插入到另一张散列表中（如果在这张表中该键的位置也被占用了，那么就将这个占用者重新插入第一张散列表，把位置腾给被插入的键），如此循环往复。动态调整数组大小以保持两张表都不到半满。这种实现中查找所需的比较次数在最坏情况下是一个常数，插入操作所需的时间在均摊后也是常数。

3.4.32 *散列攻击*。找出 2^N 个 hashCode() 方法返回值均相同且长度均为 2^N 的字符串。假设 String 类型的 hashCode() 方法的实现如下：

```
public int hashCode()
{
   int hash = 0;
   for (int i = 0; i < length(); i ++)
      hash = (hash * 31) + charAt(i);
   return hash;
}
```

重要提示：Aa 和 BB 的散列值相同。

3.4.33 糟糕的散列函数。考虑 Java 的早期版本中 String 类型的 hashCode() 方法的实现，如下所示：

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length()/8);
   for (int i = 0; i < length(); i += skip)
      hash = (hash * 37) + charAt(i);
   return hash;
}
```

说明你认为设计者选择这种实现的原因以及为什么它被替换成了上一道练习中的实现。 484

实验题

3.4.34 散列的成本。用各种常见的数据类型进行实验以得到 hash() 方法和 compareTo() 方法的耗时比的经验数据。

3.4.35 卡方检验。使用你为练习 3.4.30 给出的答案验证常用数据类型的散列函数产生的值是否随机。

3.4.36 链表长度的范围。编写一段程序，向一张长度为 $N/100$ 的基于拉链法的散列表中插入 N 个随机的 int 键，找出表中最长和最短的链表的长度，其中 $N=10^3$、10^4、10^5 和 10^6。

3.4.37 混合使用。用实验研究在 SeparateChainingHashST 中使用正 RedBlackBST 代替 SequentialSearchST 来处理碰撞的性能。这种方案的优点是即使散列函数很糟糕它仍然能够保证对数级别的性能，缺点是需要维护两种不同的符号表实现。实际效果如何呢？

3.4.38 拉链法的分布。编写一段程序，向一张大小为 10^5 的基于线性探测法的散列表中插入 10^5 个小于 10^6 的随机非负整数并在每 10^3 次插入后打印出当前探测的总次数。讨论你的结果在何种程度上验证了命题 K。[①]

3.4.39 线性探测法的分布。向一张大小为 N 的基于线性探测法的散列表中插入 $N/2$ 个随机非负整数并根据表中的键簇计算一次未命中查找的平均成本，其中 $N=10^3$、10^4、10^5 和 10^6。讨论你的结果在何种程度上验证了命题 M。

3.4.40 绘图。改进 LinearProbingHashST 和 SeparateChainingHashST 的实现，使之绘出和正文中类似的图表。

3.4.41 二次探测。用实验研究来评估二次探测法的效果（请见练习 3.4.27）。

3.4.42 二次散列。用实验研究来评估二次散列法的效果（请见练习 3.4.28）。

3.4.43 停车问题（D. Knuth）。用实验研究来验证一个猜想：向一张大小为 M 的基于线性探测法的散列表中插入 M 个随机键所需的比较次数为 ~ $cM^{3/2}$，其中 c= $\sqrt{\pi/2}$。 485

① 这个题目和拉链无关，是原书的bug。——译者注

3.5 应用

在计算机发展的早期，符号表帮助程序员从使用机器语言的数字地址进化到在汇编语言中使用符号名称；在现代应用程序中，符号名称的含义能够通行于跨越全球的计算机网络。快速查找算法曾经并继续在计算机领域中扮演着重要角色。符号表的现代应用包括科学数据的组织，例如在基因组数据中寻找分子标记或模式从而绘制全基因组图谱；网络信息的组织，从搜索在线贸易到数字图书馆；以及互联网基础构架的实现，例如包在网络结点中的路由、共享文件系统和流媒体等。高效的查找算法确保了这些以及无数其他重要的应用程序成为可能。在本节中我们会考察几个有代表性的例子。

- 能够快速并灵活地从文件中提取由逗号分隔的信息的一个字典程序和一个索引程序。逗号分隔的格式（及类似格式）常用于存储网络信息。
- 为一组文件构建逆向索引的一个程序。
- 一个表示稀疏矩阵的数据类型。它用符号表处理的问题规模能够远远大于这种数据类型的标准实现。

在第 6 章中，我们会学习一种适合于数据库或者文件系统的符号表，它能够保存的数据量超过你的想象。

符号表在本书其他章节的算法中也会起到关键的作用。例如，我们会使用符号表来表示图（第 4 章）以及处理字符串（第 5 章）。

在本章中我们已经看到，实现能够快速进行各种操作的符号表是一项很有挑战性的任务。另一方面，我们学习过的实现都经过了仔细研究，应用广泛并且在许多环境中都可用（包括 Java 的标准
486 库）。从现在开始，符号表就将成为你的编程工具箱中的一件重要武器。

3.5.1 我应该使用符号表的哪种实现

表 3.5.1 总结了由本章中多个命题和性质得到的各种符号表算法的性能特点（散列表的最坏情况除外，它的结果来自于研究文献并且也不太可能在实际应用中遇到）。从表中显然可以知道，对于典型的应用程序，应该在散列表和二叉查找树之间进行选择。

相对二叉查找树，散列表的优点在于代码更简单，且查找时间最优（常数级别，只要键的数据类型是标准的或者简单到我们可以为它写出满足（或者近似满足）均匀性假设的高效散列函数即可）。二叉查找树相对于散列表的优点在于抽象结构更简单（不需要设计散列函数），红黑树可以保证最坏情况下的性能且它能够支持的操作更多（如排名、选择、排序和范围查找）。大多数程序员的第一选择都是散列表，在其他因素更重要时才会选择红黑树。在第 5 章中我们会遇到这个“第一选择”的例外：当键都是长字符串时，我们可以构造出比红黑树更灵活而又比散列表更高效的数据结构。

表 3.5.1 各种符号表实现的渐进性能的总结

算法（数据结构）	最坏情况下的运行时间的增长数量级（N 次插入之后）		平均情况下的运行时间的增长数量级（N 次插入之后）		关键接口	内存使用（字节）
	查找	插入	查找命中	插入		
顺序查询（无序链表）	N	N	$N/2$	N	`equals()`	$48N$
二分查找（有序数组）	$\lg N$	N	$\lg N$	$N/2$	`compareTo()`	$16N$
二叉树查找（二叉查找树）	N	N	$1.39\lg N$	$1.39\lg N$	`compareTo()`	$64N$
2-3 树查找（红黑树）	$2\lg N$	$2\lg N$	$1.00\lg N$	$1.00\lg N$	`compareTo()`	$64N$
拉链法*（链表数组）	$<\lg N$	$<\lg N$	$N/(2M)$	N/M	`equals()` `hashCode()`	$48N+64M$
487 线性探测法*（并行数组）	$c\lg N$	$c\lg N$	<1.5	<2.5	`equals()` `hashCode()`	在 $32N$ 和 $128N$ 之间

* 需要均匀并独立的散列函数。

我们的符号表实现已经可以广泛应用于各种应用程序，但经过简单的修改后这些算法还可以适应并支持其他一些使用广泛的场景，有必要在这里提一下。

3.5.1.1 原始数据类型

假设我们有一张符号表，其中整型的键对应着浮点型的值。如果使用我们的标准实现，键和值会被储存在 `Integer` 和 `Double` 类中，因此我们需要两个额外的引用来访问每个键值对。如果应用程序只会使用几千个键进行几千次查找，那么这些引用可能没什么问题。但如果是对几十亿个键进行几十亿次查找，那么这些引用就会造成巨大的额外开销。使用原始数据类型代替 `Key` 类型可以为每个键值对节省一个引用。当键的值也是原始数据类型时我们又可以节约另外一个引用。图 3.5.1 显示了在拉链法中使用原始数据类型的情况，这种交换也适用于符号表的其他实现。对于性能优先的应用程序，这种改进并不困难并且值得一试（请见练习 3.5.4）。

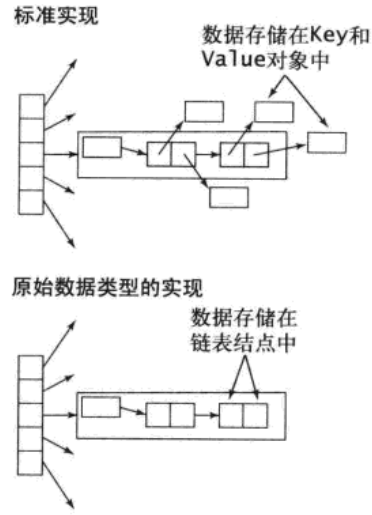


图 3.5.1 拉链法的内存使用情况

3.5.1.2 重复键

符号表的实现有时需要专门考虑重复键的可能性。许多应用都希望能够为同一个键绑定多个值。例如在一个交易处理系统中，多笔交易的客户属性都是相同的。符号表不允许重复键，因此用例只能自己管理重复键。本节稍后我们会遇到一个这样的示例程序。我们可以考虑在实现中允许数据结构保存重复的键值对，并在查找时返回给定的键所对应的任意值之一。我们也可以加入一个方法来返回给定的键对应的所有值。修改我们实现的二叉查找树和散列表来在数据结构中保存重复的键并不困难。修改红黑树可能会稍有挑战（请见练习 3.5.9 和练习 3.5.10）。这种实现在许多文献中都可以找到（包括本书以前的版本）。

488

3.5.1.3 Java 标准库

Java 的 java.util.TreeMap 和 java.util.HashMap 分别是基于红黑树和拉链法的散列表的符号表实现。TreeMap 没有直接支持 `rank()`、`select()` 和我们的有序符号表 API 中的一些其他方法，但它支持一些能够高效实现这些方法的操作。HashMap 和我们的 `LinearProbingHashST` 的实现基本相同——它也会动态调整数组的大小来保持使用率大约不超过 75%。

为了保持前后一致，我们在本书中一般会使用 3.3 节中基于红黑树的符号表或是 3.4 节中基于线性探测法的符号表。为了节省篇幅并保证符号表的用例和具体实现的独立性，我们在调用代码中将使用 `ST` 来代替有序符号表 `RedBlackBST`，用 `HashST` 来代替有序性操作无关紧要且拥有散列函数的 `LinearProbingHashST`。尽管我们知道某些应用可能需要改变或者扩展这些算法和数据结构，我们仍然要这样约定。你应该使用哪种符号表？随便，只要记得测试你的选择是否能够提供所需要的性能就好。

3.5.2 集合的 API

某些符号表的用例不需要处理值，它们只需要能够将键插入表中并检测一个键在表中是否存在。因为我们不允许重复的键，这些操作对应着下面这组 API（表 3.5.2），它们只处理表中所有键的集合，和相应的值无关。

表 3.5.2 集合数据类型的一组基本 API

public class	SET<Key>	
	SET()	创建一个空的集合
void	add(Key key)	将键 key 加入集合
void	delete(Key key)	从集合中删除键 key
boolean	contains(Key key)	键 key 是否在集合之中
boolean	isEmpty()	集合是否为空
int	size()	集合中键的数量
String	toString()	对象的字符串表示

489

只要忽略键关联的值或者使用一个简单的类进行封装，你就可以将任何符号表的实现变成一个 SET 类的实现（请见练习 3.5.1 至练习 3.5.3）。

用并（union）、交（intersection）、补（complement）和其他数学集合的操作扩展 SET 类需要的 API 更复杂（例如，complement 操作需要先定义所有可能的键的集合），使用的算法也更有趣，练习 3.5.17 会讨论它们。

基于符号表 ST，SET 类分有序和无序两个版本。如果键都是 Comparable 的，我们可以为有序的键定义 min()、max()、floor()、ceiling()、deleteMin()、deleteMax()、rank()、select() 以及需要两个参数的 size() 和 get() 方法来构成一组完整的 API。为了遵守我们关于符号表 ST 的约定，我们在用例中用 SET 表示有序的集合，用 HashSET 表示无序的集合。

```
public class DeDup
{
   public static void main(String[] args)
   {
      HashSET<String> set;
      set = new HashSET<String>();
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!set.contains(key))
         {
            set.add(key);
            StdOut.println(key);
         }
      }
   }
}
```

Dedup 过滤器

为了演示 SET 的使用方法，我们来看一组过滤器（filter）程序。它会从标准输入读取一组字符串并将其中一些写入标准输出。这种程序源自于早期内存很小无法容纳所有数据的计算机系统。它们在今天仍有用武之地，那就是当你的程序需要从网络中获取输入时。在例子中我们使用 tinyTale.txt（请见表 3.1.7）作为输入。为了保证可读性，我们将输入中的换行符保留到了输出中，不过代码并没有这么做。

3.5.2.1 dedup

过滤器例子的原型是一个调用 SET 或者 HashSET 来去掉输入流中的重复项的程序，一般叫做 dedup（如右侧代码所示）。我们会保存一个已知字符串的集合。如果下一个键已经存在于集合中，忽略之；如果不在，将它加入集合并打印它。标准输出中键的顺序和它们在标准输入中的顺序相同，只是去掉了重复项。这个过程需要的空间和输入中不同的键的数量成正比（一般比键的总量要小得多）。

```
% java DeDup < tinyTale.txt
it was the best of times worst
age wisdom foolishness
epoch belief incredulity
season light darkness
spring hope winter despair
```

490

3.5.2.2 白名单和黑名单

过滤器的另一个经典应用是用一个文件中保存的键来判定输入流中的哪些键可以被传递到输出流。这个通用程序有许多天然的应用，最简单的例子就是*白名单*。其中，文件中的键被定义为*好键*。用例可以选择将所有不在白名单上的键传递到标准输出并忽略所有白名单上的键（就像第1章中我们的第一个程序处理的那个例子一样），也可以选择只将所有在白名单上的键传递到标准输出并忽略所有不在白名单上的键（如右侧这段代码所示，使用 HashSET 实现的 WhiteFilter）。例如，电子邮件程序可能会允许用户通过这样一个过滤器指定朋友的邮件地址并将所有来自其他人的邮件当成垃圾邮件。我们根据指定的列表构造一个 HashSET，然后从标准输入中读取所有键。如果下个键存在于集合之中则打印它，否则就忽略它。*黑名单*则与之相反，名单上的所有键都被定义为*坏键*。同样，黑名单过滤器也有两种自然的应用。在电子邮件的例子中，用户可能会指定一些已知的垃圾邮件发送者的地址并要求程序放过所有不是由这些地址发来的邮件。我们可以用 HashSET 实现一个 BlackFilter，过滤条件只需要和 WhiteFilter 相反即可。实际应用中，信用卡公司用黑名单过滤被盗用的信用卡号，路由器用白名单来实现防火墙。它们使用的名单可能非常巨大，输入无限并且响应时间要求非常严格。我们已经学习过的符号表实现能够很好地满足这些需求。 491

```
public class WhiteFilter
{
   public static void main(String[] args)
   {
      HashSET<String> set;
      set = new HashSET<String>();
      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))
            StdOut.println(word);
      }
   }
}
```

白名单过滤器

```
% more list.txt
was it the of

% java WhiteFilter list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackFilter list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

3.5.3 字典类用例

符号表使用最简单的情况就是用连续的 put() 操作构造一张符号表以备 get() 查询。许多应用程序都将符号表看做一个可以方便地查询并更新其中信息的*动态字典*。以下列出了这类用例中的一些常见例子。

- *电话黄页*。当符号表中的键是人名而值是电话号码时，这张符号表就成了一个电话本。但和一本纸质印刷的电话黄页的一个重大不同是我们可以向其中添加新的名字或者更新其中的电话号码。我们也可以将电话号码作为键而将人名作为值——如果你从来没这么做过，试着在浏览器的搜索栏中输入你的电话（包括区号）并搜索一下。
- *字典*。将一个单词和它的含义关联起来就得到了“字典”。几个世纪以来人们都会在家里和

办公室里放一本纸质的字典以查找单词（键）的定义和拼写（值）。现在，有了优秀的符号表实现，人们在电脑上可以使用内置的拼写检查器并快速查到单词的意义。

- 账户信息。如今股民们都会在网上实时获取股票的价格信息。这些网络服务会关联股票名称（键）和当前价格（值）以及丰富的其他信息。类似的商业应用非常多，比如金融机构会将名字或者账号与账户信息关联，学校会将学生的姓名或者学号与他的成绩关联，等等。
- 基因组学。在现代基因组学中符号的作用非常重要。最简单的例子就是A、C、T和G这几个字母代表了活体组织中DNA的四种核苷酸。另一个比较简单的例子是密码子（核苷酸三联体）和氨基酸的对应关系（TTA表示亮氨酸，TCT表示丝氨酸，等等），以及氨基酸序列和蛋白质之间的对应关系。基因组学的研究者每天都需要使用各种符号表来组织这些信息。
- 实验数据。从天体物理学到动物学，现代科学家被各种实验数据包围着。有效的组织和访问这些信息才能理解它们的含义，而符号表正是一个关键的入手点。基于符号表的高级数据结构和算法如今已经成为科学研究的一个重要部分。
- 编译器。符号表最早期的应用之一就是组织程序代码的信息。最初，计算机程序只是一串简单的数字，但程序员们很快发现使用符号来表示操作和内存地址（变量名）要方便得多。将名称和数字关联起来就需要一张符号表。随着程序的增长，符号表操作的性能逐渐变成了程序开发效率的瓶颈，为此而开发的数据结构和算法就是我们在本章中学习的内容。

492

- 文件系统。我们都在使用符号表定期整理计算机系统中的数据。也许其中最明显的例子就是文件系统了，因为是它将文件名（键）和文件内容的地址（值）关联起来。音乐播放器同样使用文件系统关联了歌曲名（键）和歌曲的位置（值）。
- 互联网DNS。域名系统（DNS）是互联网信息组织的基础，它可以将人类能够理解的URL（键，如www.princeton.edu或是www.wikipedia.org）和计算机网络中路由器能够理解的IP地址（值，如208.216.181.15或是207.142.131.206）关联起来。这个系统被称为下一代“电话黄页”。有了它，人们就可以使用便于记忆的域名，而机器也可以高效地处理对应的数字。为此，全球互联网的路由器中每秒钟进行的符号表查找次数是个天文数字，所以性能显然非常重要。每年，互联网上都会新增上百万台电脑和其他设备，因此互联网路由器中的符号表也需要能够动态地适应它们。

将以上几个典型应用总结一下，如表3.5.3所示。

表3.5.3 典型的字典类应用

应用领域	键	值
电话黄页	人名	电话号码
字典	单词	定义
账户信息	账号	余额
基因组	密码子	氨基酸
实验数据	数据/时间	实验结果
编译器	变量名	内存地址
文件共享	歌曲名	计算机
DNS	网站	IP地址

尽管已经涉及了许多领域，表3.5.3中选取的仍然只是几个有代表性的例子来说明符号表应用的广泛程度。每当使用一个名称来指代某种东西时，都用到了符号表。也许你只是用到了计算机的

文件系统或是互联网，但在某个角落肯定有一张符号表在默默工作。

作为一个具体的例子，我们来看看一个从文件或者网页中提取由逗号分隔的信息（.csv 文件格式）的程序。这种格式存储的列表的信息不需要任何专用的程序就可以读取：数据都是文本，每行中各项均由逗号隔开。在本书的网站上你会找到很多 .csv 文件，都和我们刚才提到过的应用领域相关，包括 amino.csv（密码子和氨基酸的编码关系）、DJIA.csv（道琼斯工业平均指数历史上每天的开盘价、成交量和收盘价）、ip.csv（DNS 数据库中的一部分条目）和 upc.csv（广泛用于识别商品的 Uniform Product Code 条形码），如右侧代码框所示。电子表格等数据处理应用程序都能读写 .csv 文件，我们的例子程序说明你也能够编写 Java 程序来根据需要处理这些数据。

下页的 LookupCSV 根据命令行指定的文件中的数据构建了一组键值对，并会打印出由标准输入读取的键对应的值。命令行参数包括一个文件名和两个整数，分别用来指定键和值所在的位置。

这个例子的目的在于展示符号表的作用和灵活性。哪个网站的 IP 地址是 128.112.136.35？ www.cs.princeton.edu；哪种氨基酸对应着密码子 TCA？丝氨酸；DJIA 在 1929 年 10 月 29 号的价格是多少？ 252.38；哪种商品的条形码是 0002100001086？卡夫芝士粉（Kraft Parmesan）。有了 LookupCSV 和合适的 .csv 文件，可以轻易查到这类问题的答案。

在处理交互性的查询时，性能一般都不是问题（因为你的计算机在你打字的工夫就能检索上百万条信息），所以在使用 LookupCSV 时符号表的高效性并不明显。但是当程序需要进行（大量的）查找时，符号表的性能就很重要了。例如，互联网上的一

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
...
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine

% more DJIA.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...

% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...

% more UPC.csv
...
0002058102040,,"1 1/4"" STANDARD STORM DOOR"
0002058102057,,"1 1/4"" STANDARD STORM DOOR"
0002058102125,,"DELUXE STORM DOOR UNIT"
0002082012728,"100/ per box","12 gauge shells"
0002083110812,"Classical CD","'Bits and Pieces'"
002083142882,CD,"Garth Brooks - Ropin' The Wind"
0002094000003,LB,"PATE PARISIEN"
0002098000009,LB,"PATE TRUFFLE COGNAC-M&H 8Z RW"
0002100001086,"16 oz","Kraft Parmesan"
0002100002090,"15 pieces","Wrigley's Gum"
0002100002434,"One pint","Trader Joe's milk"
...
```

典型的含有由逗号分隔的值的文件（.csv）

台路由器每秒钟可能需要查找上百万个 IP 地址。在本书中，我们已经通过 FrequencyCounter 看到了高性能的必要性，在本节中你还会看到其他几个例子。

练习里有几个更加复杂的处理 .csv 文件的测试用例。例如，我们可以将一个字典动态化，允许它接受从标准输入中得到的指令来改变一个键的值，或是为它添加范围查找的功能，或者我们可以为同一个文件构造多个字典。

493 ~ 494

字典的查找

```
public class LookupCSV
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);
      ST<String, String> st = new ST<String, String>();
      while (in.hasNextLine())
      {
         String line = in.readLine();
         String[] tokens = line.split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }
      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         if (st.contains(query))
            StdOut.println(st.get(query));
      }
   }
}
```

这段数据驱动的符号表用例会从一个文件中读取键值对并根据标准输入中的键打印出相应的值。其中键和值都是字符串，分隔符由命令行参数指定。

```
% java LookupCSV ip.csv 1 0
128.112.136.35
www.cs.princeton.edu

% java LookupCSV DJIA.csv 0 3
29-Oct-29
230.07

% java LookupCSV amino.csv 0 3
TCC
Serine

% java LookupCSV UPC.csv 0 2
0002100001086
Kraft Parmesan
```

495

3.5.4 索引类用例

字典的主要特点是每个键都有一个与之关联的值，因此基于关联型抽象数组来为一个键指定一个值的符号表数据类型正合适。每个账号都唯一地表示一个客户，每个条码都唯一地表示一种商品，等等。但一般说来，一个给定的键当然有可能和多个值相关联。例如，在我们的 amino.csv 的例子中，每个密码子都对应着一种氨基酸，但一种氨基酸有可能对应着多个密码子。如下页的 aminoI.txt 所示，

文件的每一行都包含一个氨基酸和它对应的多个密码子。

我们使用*索引*来描述一个键和多个值相关联的符号表，下面是更多的例子。

- 商业交易。公司使用客户账户来跟踪一天内所有交易的一种方法是为当日所有交易建立一个索引，其中键是客户的账号，值是和该账号有关的所有交易。
- 网络搜索。当你输入一个关键字并得到一系列含有这个关键字的网站时，你就是在使用网络搜索引擎创建的索引。每个键（查询）都关联着一个值（一组网页），当然实际情况会更加复杂，因为我们经常会指定多个关键字。
- 电影和演员。本书网站上的 movies.txt 来自于 IMDB（互联网电影数据库）。每一行都含有一部电影的名称（键），随后是在其中出演的演员列表（值），用斜杠分隔，如图 3.5.2 所示。

aminoI.txt

```
Alanine,AAT,AAC,GCT,GCC,GCA,GCG
Arginine,CGT,CGC,CGA,CGG,AGA,AGG
Aspartic Acid,GAT,GAC
Cysteine,TGT,TGC
Glutamic Acid,GAA,GAG
Glutamine,CAA,CAG
Glycine,GGT,GGC,GGA,GGG
Histidine,CAT,CAC
Isoleucine,ATT,ATC,ATA
Leucine,TTA,TTG,CTT,CTC,CTA,CTG
Lysine,AAA,AAG
Methionine,ATG
Phenylalanine,TTT,TTC
Proline,CCT,CCC,CCA,CCG
Serine,TCT,TCA,TCG,AGT,AGC
Stop,TAA,TAG,TGA
Threonine,ACT,ACC,ACA,ACG
Tyrosine,TAT,TAC
Tryptophan,TGG
Valine,GTT,GTC,GTA,GTG
```

"," 分隔符

键　　多个值

一个小型索引文件（20行）

将每个键关联的所有值都放入一个数据结构中（比如一个 Queue）并用它作为值就可以轻松构造一个索引。根据这一点来扩展 LookupCSV 很简单，我们将它留作一道练习（请见练习 3.5.12）。这里我们看一下 LookupIndex，它能够从一个文件，例如 aminoI.txt 或 movies.txt（分隔符不一定和 .csv 文件一样必须是逗号，但需要能够从命令行指定），构造一个索引。构造完成后 LookupIndex 能够接受查询并打印出键对应的所有值。更有意思的是 LookupIndex 也会为每个文件构造一个*反向索引*，也就是将键和值的角色互换。在氨基酸的例子中，它的功能相当于 LookupCSV（找到给定密码子所对应的氨基酸）。在电影和演员的例子中，它使我们能够找到一个演员出演过的所有电影。这项信息隐藏于数据当中，但没有符号表我们就很难获取它。*请仔细研究这个例子*，因为它深刻地揭示了符号表的本质特征。

表 3.5.4 总结了典型的索引类应用的符号表中键值的对应情况。

表 3.5.4　典型的索引类应用

应用领域	键	值	应用领域	键	值
基因组学	氨基酸	一系列密码子	网络搜索	关键字	一系列网页
商业交易	账号	一系列交易	IMDB	电影	一系列演员

movies.txt

```
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/...
Tirez sur le pianiste (1960)/Heymann, Claude/...
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/...
To Be or Not to Be (1942)/Verebes, Ernö (I)/...
To Be or Not to Be (1983)/.../Brooks, Mel (I)/...
To Catch a Thief (1955)/Paris, Manuel/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/...
...
```

"/"分隔符

键　　多个值

图 3.5.2　一个巨型索引文件（250 000 多行）的一小部分

反向索引

反向索引一般是指用值来查找键的操作，比如我们有大量的数据并且希望知道某个键都在哪些地方出现过。这是另一种符号表的典型用例，它会进行一系列 get() 和 put() 的混合调用。和以前一样，我们将每个键和一个 SET 类型的值关联起来，这个值中包含了该键出现的所有位置。位置信息的性质和用途取决于应用场景：在一本书中，位置可能是书的页码；在一段程序中，位置可能是代码的行号；在基因组中，位置可能是一段基因序列的某个位点，等等。

- 互联网电影数据库（IMDB）。在上文的例子中，输入是将每部电影和它的演员关联起来的一个索引。它的反向索引则会将每个演员和他出演过的所有电影相关联。
- 图书索引。每本教科书都会有一个索引。你能在其中查找到一个术语和它出现过的所有页码。创建优秀的索引当然需要作者的努力来去掉常见和无关的词语，但文档处理系统能够使用符号表将整个过程自动化。一种有趣的特殊情况叫做对照索引（concordance），它会给出每个单词在书中出现的所有位置（请见练习 3.5.20）。
- 编译器。在一个使用了许多符号的庞大程序中，能够知道每个名称的使用位置很有帮助。在以前，一张打印的以追踪各个符号在程序中使用位置的符号表曾经是程序员最重要的工具之一。在现代计算机系统中，符号表是程序员用来管理各种名称的工具软件的基础。
- 文件搜索。现代操作系统都提供了根据关键字搜索文件的功能。对于这个索引，键就是关键字，值则是含有该关键字的所有文件的集合。
- 基因组学。基因组学研究中的一个典型（或许有些过于简化了）情况是科学家希望知道一个给定的核苷酸序列在一个基因或者一组基因中的位置。某些特定序列或者近似序列的存在也许都有重大的意义。这种研究首先就需要一个序列和基因的对照索引，但也需要一些修改，因为基因是无法像句子一样被切分为单词的（请见练习 3.5.15）。

常见反向索引用例的符号表的键值对应情况如表 3.5.5 所示。

表 3.5.5 典型的反向索引

应用领域	键	值
IMDB	演员	一系列电影
图书	术语	一系列页码
编译器	标识语	一系列使用位置
文件搜索	关键字	文件集合
基因组学	基因片段	一系列位置

498

索引（以及反向索引）的查找

```
public class LookupIndex
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);    // (索引数据库)
      String sp = args[1];        // (分隔符)
      ST<String, Queue<String>> st = new ST<String, Queue<String>>();
      ST<String, Queue<String>> ts = new ST<String, Queue<String>>();
      while (in.hasNextLine())
```

```
        {
           String[] a = in.readLine().split(sp);
           String key = a[0];
           for (int i = 1; i < a.length; i++)
           {
              String val = a[i];
                    if (!st.contains(key))
st.put(key, new Queue<String>());
                    if (!ts.contains(val))
ts.put(val, new Queue<String>());
              st.get(key).enqueue(val);
              ts.get(val).enqueue(key);
           }
        }
        while (!StdIn.isEmpty())
        {
           String query = StdIn.readLine();
           if (st.contains(query))
             for (String s : st.get(query))
                StdOut.println(" " + s);
           if (ts.contains(query))
             for (String s : ts.get(query))
                StdOut.println(" " + s);
      }
    }
  }
```

```
% java LookupIndex aminoI.txt ","
Serine
  TCT
  TCA
  TCG
  AGT
  AGC
TCG
  Serine

% java LookupIndex movies.txt "/"
Bacon, Kevin
  Mystic River (2003)
  Friday the 13th (1980)
  Flatliners (1990)
  Few Good Men, A (1992)
  ...
Tin Men (1987)
  Blumenfeld, Alan
  DeBoy, David
  ...
```

这段数据驱动的符号表用例会从一个文件中读取键值对并根据标准输入中的键打印出相应的值。其中键为字符串，值为一列字符串，分隔符由命令行参数指定。

499

下面的 FileIndex 从命令行接受多个文件名并使用一张符号表来构造一个反向索引，它能够将任意文件中的任意一个单词和一个出现过这个单词的所有文件的文件名构成的 SET 对象关联起来。在接受标准输入的查询时，输出单词对应的文件列表。这个过程与工具软件在网络上或是在你的计算机上查找信息的过程类似，即根据输入的关键字得到所有该关键字出现过的位置。这类工具的开发者一般会在下面几点上下工夫来改进这个过程：

- 查询形式；
- 被索引的文件或网页的集合；
- 文件或网页在结果中的排列顺序。

例如，你肯定已经习惯了在网络搜索引擎（它们的基础都是将网络上的大部分页面进行索引）的查询中输入多个关键字进行查找，并得到一组按照相关性或者重要性（对于你或是对于广告商而言）由高到低排序的结果。本节最后的练习中讨论了这里的一些改进。我们会在以后学习和网络搜索有关的各种算法，但符号表仍然会是整个过程的核心工具。

和 LookupIndex 一样，你也应该从本书的网站上下载 FileIndex 并用它来为你的电脑上的一些文件或是你感兴趣的一些网站建立索引，从而更好地理解符号表的使用。你将会发现即使是根据巨型文件构造庞大的索引，这个工具的耗时也不多，因为每个 put() 操作和 get() 请求的处理都非常快。确保巨型的动态索引实现即时响应是算法技术的重要胜利之一。

500

文件索引

```
import java.io.File;
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();
      for (String filename : args)
      {
         File file = new File(filename);
         In in = new In(file);
         while (!in.isEmpty())
         {
            String word = in.readString();
            if (!st.contains(word)) st.put(word, new SET<File>());
            SET<File> set = st.get(word);
            set.add(file);
         }
      }
      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         if (st.contains(query))
             for (File file : st.get(query))
               StdOut.println("   " + file.getName());
      }
   }
}
```

这段符号表用例能够为一组文件创建索引。我们将每个文件中的每个单词都记录在符号表中并维护一个 SET 对象来保存出现过该单词的文件。In 对象接受的名称也可以是网页，因此这段代码也可以用来为一组网页创建反向索引。

```
% more ex1.txt
it was the best of times

% more ex2.txt
it was the worst of times

% more ex3.txt
it was the age of wisdom

% more ex4.txt
it was the age of foolishness
```

```
% java FileIndex ex*.txt
age
  ex3.txt
  ex4.txt
best
  ex1.txt
was
  ex1.txt
  ex2.txt
  ex3.txt
  ex4.txt
```

501

3.5.5 稀疏向量

下面这个例子展示的是符号表在科学和数学计算领域所起到的重要作用。我们会考察一种重要而常见的计算，它在典型的实际应用中常常是性能的瓶颈，然后我们会演示符号表如何解决这个瓶颈并能够处理规模大得多的问题。实际上，这个计算正是 S. Brin 和 L. Page 发明的 PageRank 算法的核心，这个算法在 2000 年左右造就了 Google（它同时也是一个著名的数学抽象模型，在很多其

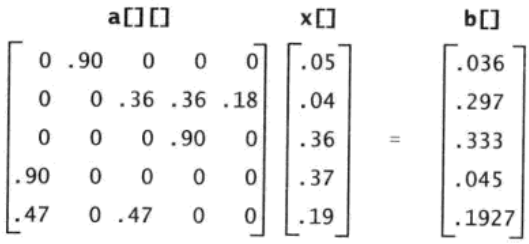


图 3.5.3 矩阵和向量的乘法

他场景中都会用到）。

我们要考察的简单计算就是矩阵和向量的乘法（如图 3.5.3 所示）：给定一个矩阵和一个向量并计算结果向量，其中第 i 项的值为矩阵的第 i 行和给定的向量的点乘。为了简化问题，我们只考虑 N 行 N 列的方阵，向量的大小也为 N。在 Java 中，用代码实现这种操作非常简单，但所需的时间和 N^2 成正比，因为 N 维结果向量中的每一项都需要计算 N 次乘法。因为需要存储整个矩阵，计算所需的空间也和 N^2 成正比。实现代码如下所示。

```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// 初始化a[][]和x[]
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

矩阵和向量相乘的标准实现

在实际应用中，N 往往非常巨大。例如，在刚才提到的 Google 的应用中，N 等于互联网中所有网页的总数。在 PageRank 算法发明的时候，这个数字大概在百亿到千亿之间，但之后一直在暴增。因此，N^2 的值应该远远大于 10^{20}。没人能够负担起这么多内存和时间来进行这种计算，所以我们需要更好的算法。

幸好，这里的矩阵常常是*稀疏的*，即其中大多数项都是 0。实际上，在 Google 的应用中，每行中的非零项的数量是一个较小的常数：每个网页中指向其他页面的链接其实都很少（相比互联网中所有网页的总数而言）。因此，我们可以将这个矩阵表示为由稀疏向量组成的一个数组，使用 HashST 的稀疏向量实现如下面的 SparseVector 所示。

502

能够完成点乘的稀疏向量

```
public class SparseVector
{
   private HashST<Integer, Double> st;
   public SparseVector()
    {  st = new HashST<Integer, Double>();  }
   public int size()
   {  return st.size();  }
   public void put(int i, double x)
    {  st.put(i, x);  }
   public double get(int i)
   {
      if (!st.contains(i)) return 0.0;
      else return st.get(i);
   }
   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : st.keys())
            sum += that[i]*this.get(i);
       return sum;
   }
}
```

这个符号表的用例实现了稀疏向量的主要功能并高效完成了点乘操作。我们将一个向量中的每一项
503 和另一个向量中对应项相乘并将所有结果相加，所需的乘法操作数量等于稀疏向量中的非零项的数目。

稀疏矩阵的表示如图 3.5.4 所示。

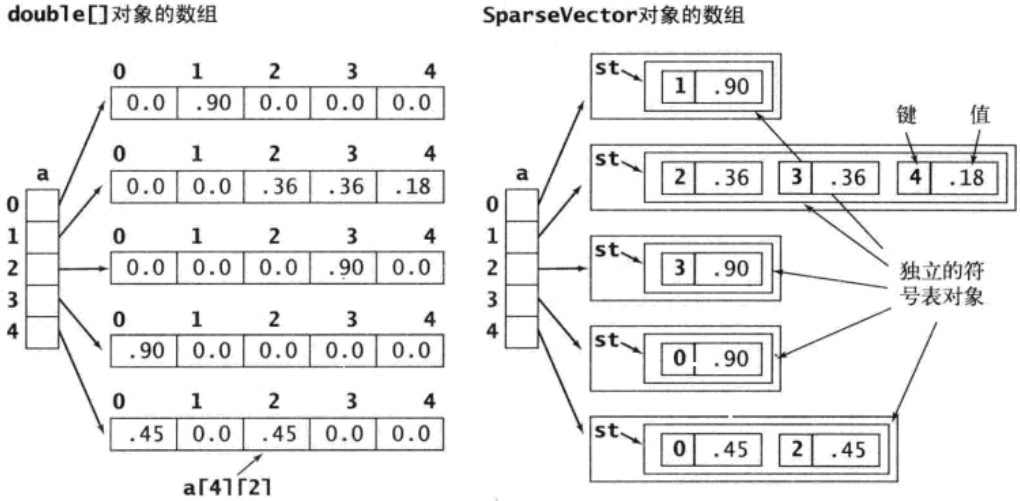


图 3.5.4 稀疏矩阵的表示

这里我们不再使用 a[i][j] 来访问矩阵中第 i 行第 j 列的元素，而是使用 a[i].put(j, val) 来表示矩阵中的值并使用 a[i].get(j) 来获取它。从下面这段代码可以看到，用这种方式实现的矩阵和向量的乘法比数组表示法的实现更简单（也能更清晰地描述乘法的过程）。更重要的是，它所需的时间仅和 N 加上矩阵中的非零元素的数量成正比。

虽然对于较小或是不那么稀疏的矩阵，使用符号表的代价可能会非常高昂，但你应该理解它对于巨型稀疏矩阵的意义。为了更好地说明这一点，设想一个超大的应用（就像 Brin 和 Page 面对的问题一样），N 可能超过 100 亿或者 1000 亿而平均每行中的非零元素小于 10。对于这种应用，使用符号表能够将矩阵和向量乘法的速度提升 10 亿倍甚至更多。这种应用虽然简单但非常重要，不愿意挖掘其中省时省力的潜力的程序员解决实际问题能力的潜力也必然是有限的，能够将运行速度提升几十亿倍的程序员勇于面对看似无法解决的问题。

构造 Google 所使用的矩阵是一种图的应用（当然也是符号表的一种应用），尽管是一个巨型的稀疏矩阵。有了这个矩阵，PageRank 算法的计算就变成了简单的矩阵和向量之间的乘法运算，不断用结果向量取代计算所使用的向量，重复这个迭代过程直到收敛（这一点是由概率论的基础定理所保证的）。因此，使用一个类似于 SparseVector 的类能够将这种应用程序所需的空间和时间改进几百或者几千亿倍，甚至更多。

在许多科学计算中类似的改进都是可能的，因此稀疏向量和矩阵的应用十分广泛，并且一般都会被集成到科学计算专用的库中。在处理庞大的向量或矩阵的时候，你最好用一些简单的性能测试来保证不会错过类似的改进机会。另外，大多数编程语言都拥有处理原始数据类型数组的能力，因此像例子中那样用数组来保存密集的向量也许能提供更好的性能。对于这些应用，有必要深入了解它们的运行瓶颈从而选择合适的数据类型实现。

符号表之所以是算法技术为现代计算机基础设施建设的一大重要贡献，是因为在很多实际应用中它都能够节省大量的运行成本，使得各个领域内许多原来完全无法想象的问题的解决成为可能。科学或是工程领域能够将运行效率提升一千亿倍的发明极少——我们已经在几个例子中看到，符号表做到了，并且这些改进的影响非常深远。但我们学习过的数据结构和算法的演化并没有结束：它们才出现了几十年，我们也并没有完全了解它们的性质。鉴于它们的重要性，符号表的各种实现仍然是全球学者的研究热点。随着它的应用范围不断扩展，我们会在更多领域看到它的新发展。

```
...
SparseVector[] a;
a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];

...
// 初始化 a[] 和 x[]
...
for (int i = 0; i < N; i++)
   b[i] = a[i].dot(x);
```

稀疏矩阵和向量的乘法

504 ≀ 505

答疑

问 SET 能够包含 null 吗？

答 不行。和符号表一样，键必须是非空的对象。

问 SET 可以是 null 吗？

答 不行。一个 SET 集合可以是空的（不包含任何对象），但不能为 null。和 Java 的其他数据类型一样，一个 SET 类型的变量的值可以是 null，但这仅仅意味着它没有指向任何 SET 对象。对 SET 使用 new 的结果必然是一个非空的对象。

问 如果能够将所有数据都存储在内存中，那就没有必要使用过滤器了，对吗？

答 是的。过滤器最大的用处在于处理输入数据量未知的情况。在其他情况下，它可能会是一种有用的思维方式，但也不是万能的。

问 我在一张电子表格中保存了一些数据。我需要开发一个类似于 LookupCSV 的程序查找这些数据吗？

答 你的电子表格程序应该能够将它们导出为 .csv 的文件，这样你就可以直接使用 LookupCSV 了。

问 FileIndex 程序有什么用？操作系统不能解决这个问题吗？

答 如果操作系统能够满足你的需求，当然应该直接使用它的解决方案。和我们的许多例子程序一样，FileIndex 也是为了向你展示这些应用程序的基本原理并为你提供其他的可能性。

问 为什么 SparseVector 的 dot() 方法不接受一个 SparseVector 对象作为参数并返回一个 SparseVector 对象？

答 这也是一个不错的设计，它所需的代码比我们的设计稍稍复杂一些，因此也是一道不错的编程练习（请见练习 3.5.16）。对于普通矩阵的处理，我们也许还应该再增加一个 SparseMatrix 数据类型。

506

练习

3.5.1 分别使用 ST 和 HashST 来实现 SET 和 HashSET（为键关联虚拟值并忽略它们）。

3.5.2 删除 SequentialSearchST 中和值相关的所有代码来实现 SequentialSearchSET。

3.5.3 删除 BinarySearchST 中和值相关的所有代码来实现 BinarySearchSET。

3.5.4 分别为 int 和 double 两种原始数据类型的键实现 HashSTint 类和 HashSTdouble 类（将 LinearProbingHashST 中的泛型改为原始数据类型）。

3.5.5 分别为 int 和 double 两种原始数据类型的键实现 STint 类和 STdouble 类（将 RedBlackBST 中的泛型改为原始数据类型）。用经过修改的 SparseVector 作为用例测试你的答案。

3.5.6 分别为 int 和 double 两种原始数据类型的键实现 HashSETint 类和 HashSETdouble 类（删去你为练习 3.5.4 给出的答案中所有关于值的代码）。

3.5.7 分别为 int 和 double 两种原始数据类型的键实现 SETint 类和 SETdouble 类（删去你为练习 3.5.5 给出的答案中所有关于值的代码）。

3.5.8 修改 LinearProbingHashST，允许在表中保存重复的键。对于 get() 方法，返回给定键所关联的任意值；对于 delete() 方法，删除表中所有和给定键相等的键值对。

3.5.9 修改二叉查找树 BST，允许在树中保存重复的键。对于 get() 方法，返回给定键所关联的任意值；对于 delete() 方法，删除树中所有和给定键相等的结点。

3.5.10 修改红黑树 RedBlackBST，允许在树中保存重复的键。对于 get() 方法，返回给定键所关联的
507 任意值；对于 delete() 方法，删除树中所有和给定键相等的结点。

3.5.11 开发一个和 SET 相似的类 MultiSET，允许出现相等的键，也就是实现了数学上的多重集合。

3.5.12 修改 LookupCSV，将每个键和输入中与该键对应的所有值相关联（而非和关联型抽象数组的一样，仅关联最近出现的那个值）。

3.5.13 修改 LookupCSV 为 RangeLookupCSV，从标准输入接受两个键并打印出 .csv 文件中所有在该范围之内的键值对。

3.5.14 编写并测试方法 invert()，它接受参数 ST<String, Bag<String>> 并返回给定符号表的反向索引（一个相同类型的符号表）。

3.5.15 编写一个程序，从标准输入接受一个字符串和一个整数 k 作为参数，在标准输出中有序打印出在字符串中找到的 k 元文法（k-gram），以及每个 k-gram 在字符串中的位置。

3.5.16 为 SparseVector 添加一个 sum() 方法，接受一个 SparseVector 对象作为参数并将两者相加的结果返回为一个 SparseVector 对象。请注意：你需要使用 delete() 方法来处理向量中的一
508 项变为 0 的情况（请特别注意精度）。

提高题

3.5.17 数学集合。你的目标是实现表 3.5.6 中 MathSET 的 API 来处理（可变的）数学集合。

表 3.5.6 一种简单的集合数据类型的 API

Public class	**MathSET<Key>**	
	MathSET(Key[] universe)	创建一个集合
void	add(Key key)	将 key 加入集合
MathSET<Key>	complement()	所有不在该集合中的键的集合
void	union(MathSET<Key> a)	将 a 中所有不在该集合中的键加入该集合（并集）
void	intersection(MathSET<Key> a)	将该集合中所有不在 a 中的键删除（交集）
void	delete(Key key)	将 key 从集合中删去
boolean	contains(Key key)	集合中是否存在键 key
boolean	isEmpty()	集合是否为空
int	size()	集合中键的总数

请使用符号表来实现它。附加题：使用 boolean 类型的数组来表示集合。

3.5.18 多重集合。请参考练习 3.5.2、练习 3.5.3 以及前面的练习，为无序和有序的多重集合（可以含有相同的键的集合）给出 MultiHashSET 和 MultiSET 的 API，并分别用 SeparateChainingMultiSET 和 BinarySearchMultiSET 实现它们。

3.5.19 符号表中的等值键。（有序的和无序的）MultiST 的 API 分别和表 3.1.2 以及表 3.1.4 中定义的符号表 API 相同，只是允许存在等值的键。因此，get() 方法的行为是返回给定键所关联的任意值。另外，我们还需要添加一个新方法来返回和给定键关联的所有值：

```
Iterable<Value> getAll(Key key)
```

根据我们的 SeparateChainingHashST 和 BinarySearchST 的代码来实现 SeparateChainingMultiST 和 BinarySearchMultiST 的 API。 509

3.5.20 对照索引。编写一个 ST 的用例 Concordance，为从标准输入得到的字符串构建对照索引并打印出来（请见表 3.5.5）。

3.5.21 反向对照索引。编写一个程序 InvertedConcordance，从标准输入接受一个对照索引并在标准输出中打印出原始的字符串。注意：这个计算和著名的“死海卷轴”故事有关。最早发现原始石板的团队仅公开了用一种不为人知的方式生成的对照索引。一段时间之后其他研究者才找到了如何将这种索引还原的方法，并最终将石板上的全文公之于众。

3.5.22 完全索引的 CSV 文件。编写一个 ST 的用例 FullLookupCSV，构造一个 ST 对象的数组（每列一个），以及一个允许使用者指定键和值的列的测试用例。

3.5.23 稀疏矩阵。为稀疏二维矩阵设计一组 API 并将它实现，支持矩阵的加法和乘法操作。包含分别能够指定行和列向量的构造函数。

3.5.24 不重叠的区间查找。给定对象的一组互不重叠的区间，编写一个函数接受一个对象作为参数并判断它是否存在于其中任何一个区间之内。例如，如果对象是整数而区间为 1643–2033，5532–7643，8999–10332，5666653–5669321，那么查询 9122 的结果为第三个区间，而 8122 的结果是不在任何区间。

3.5.25 登记员的日程安排。东北部某著名大学的注册主任最近作出的安排中有一位老师需要在同一时间为两个不同的班级授课。请用一种方法来检查类似的冲突，帮助这位主任不要再犯同样的错误。简单起见，假设每节课的时间为 50 分钟，分别从 9:00、10:00、11:00、1:00、2:00 和 3:00 开始。

3.5.26 LRU 缓存。创建一个支持以下操作的数据结构：访问和删除。访问操作会将不存在于数据结构中的元素插入。删除操作会删除并返回最近访问过的元素。提示：将元素按照访问的先后顺序保存在一条双向链表之中，并保存指向开头和结尾元素的指针。将元素和元素在链表中的位置分别作为键和相应的值保存在一张符号表中。当你访问一个元素时，将它从链表中删除并重新插入链表的头部。当你删除一个元素时，将它从链表的尾部和符号表中删除。 510

3.5.27 列表。实现表 3.5.7 中的 API：

表 3.5.7 列表数据类型的 API

Public class List<Item> implements Iterable<Item>	
List()	创建一个列表
void addFront(Item item)	将 item 添加到列表的头部
void addBack(Item item)	将 item 添加到列表的尾部
Item deleteFront()	删除列表头部的元素
Item deleteBack()	删除列表尾部的元素

（续）

Public class List<Item> implements Iterable<Item>	
void delete(Item item)	从列表中删除 item
void add(int i, Item item)	将 item 添加为列表的第 i 个元素
Item delete(int i)	从列表中删除第 i 个元素
boolean contains(Item item)	列表中是否存在元素 item
boolean isEmpty()	列表是否为空
int size()	列表中元素的总数

提示：使用两个符号表，一个用来快速定位列表中的第 i 个元素，另一个用来快速根据元素查找。（Java 的 java.util.List 包含类似的方法，但它的实现的操作并不都是高效的。）

3.5.28 uniQueue。创建一个类似于队列的数据类型，但每个元素只能插入队列一次。用一个符号表来记录所有已经被插入的元素并忽略所有将它们重新插入的请求。

511

3.5.29 支持随机访问的符号表。创建一个数据结构，能够向其中插入键值对，查找一个键并返回相应的值以及删除并返回一个随机的键。提示：将一个符号表和一个随机队列结合起来实现该数据结构。

512

实验题

3.5.30 重复元素（续）。使用3.5.2.1节的dedup过滤器重新完成练习2.5.31。比较两种解决方法的运行时间。然后使用 dedup 运行试验，其中 $N=10^7$、10^8 和 10^9。使用随机的 long 值重新完成试验并讨论结果。

3.5.31 拼写检查。将本书网站上的 dictionary.txt 文件作为命令行参数，用 3.5.2.2 节的 BlackFilter 程序打印出从标准输入接受的文本文件中所有拼写错误的单词。在这个测试中分别使用 RedBlackBST、SeparateChainingHashST 和 LinearProbingHashST 处理 WarAndPeace.txt（本书网站提供）并讨论结果。

3.5.32 字典。在一个性能优先的场景中研究类似于 LookupCSV 用例的性能。请设计一个查询生成器来代替标准输入并用大量的输入和查询来测试用例的性能。

3.5.33 索引。在一个性能优先的场景中研究类似于 LookupIndex 用例的性能。请设计一个查询生成器来代替标准输入并用大量的输入和查询来测试用例的性能。

3.5.34 稀疏向量。用实验来比较使用稀疏矩阵和使用标准数组实现矩阵向量乘法的性能。

3.5.35 原始数据类型。对于 LinearProbingHashST 和 RedBlackBST，评估使用原始数据类型来表示 Integer 和 Double 值的情况。如果在一张巨型的符号表中进行大量的查找，这么做能节省多少空间和时间？

513

第4章 图

在许多计算机应用中，由相连的结点所表示的模型起到了关键的作用。这些结点之间的连接很自然地会让人们产生一连串的疑问：沿着这些连接能否从一个结点到达另一个结点？有多少个结点和指定的结点相连？两个结点之间最短的连接是哪一条？

要描述这些问题，我们要使用一种抽象的数学对象，叫做图。本章中，我们会详细研究图的基本性质，为学习各种算法并回答这种类型的疑问作好准备。这些算法是解决许多重要的实际问题的基础，没有优秀的算法，这些问题的解决无法想象。

图论作为数学领域中的一个重要分支已经有数百年的历史了。人们发现了图的许多重要而实用的性质，发明了许多重要的算法，其中许多困难问题的研究仍然十分活跃。本章中，我们会介绍一系列基础的图算法，它们在各种应用中都十分重要。

和我们已经研究过的许多其他问题域一样，关于图的算法研究相对来说才开始不久。尽管有些基础的算法在几个世纪前就已发现了，但大多数有趣的结论都是近几十年才被发现。得益于我们已经学习过的那些算法，即使是由最简单的图论算法得到的程序也是很有用的，而那些我们将要学习的复杂算法则都是已知的最优美和最有意思的算法的一部分。

514
~
515

为了展示图论应用的广泛领域，在探索这片富饶之地之前，我们先来看以下几个示例。

地图。正在计划旅行的人也许想知道“从普罗维登斯到普林斯顿的最短路线”。对最短路径上经历过交通堵塞的旅行者可能会问：“从普罗维登斯到普林斯顿的哪条路线最快？”要回答这些问题，我们都要处理有关结点（十字路口）之间多条连接（公路）的信息。

网页信息。当我们在浏览网页时，页面上都会包含其他网页的引用（链接）。通过单击链接，我们可以从一个页面跳到另一个页面。整个互联网就是一张图，结点是网页，连接就是超链接。图算法是帮助我们在网络上定位信息的搜索引擎的关键组件。

电路。在一块电路板上，晶体管、电阻、电容等各种元件是精密连接在一起的。我们使用计算机来控制制造电路板的机器并检查电路板的功能是否正常。我们既要检查短路这类简单问题，也要检查这幅电路图中的导线在蚀刻到芯片上时是否会出现交叉等复杂问题。第一类问题的答案仅取决于连接（导线）的属性，而第二个问题则会涉及导线、各种元件以及芯片的物理特性等详细信息。

任务调度。商品的生产过程包含了许多工序以及一些限制条件，这些条件会决定某些任务的先后次序。如何安排才能在满足限制条件的情况下用最少的时间完成这些生产工序呢？

商业交易。零售商和金融机构都会跟踪市场中的买卖信息。在这种情形下，一条连接可以表示现金和商品在买方和卖方之间的转移。在此情况下，理解图的连接结构原理可能有助于增强人们对市场的理解。

配对。学生可以申请加入各种机构，例如社交俱乐部、大学或是医学院等。这里结点就对应学生和机构，而连接则对应递交的申请。我们希望找到申请者与他们感兴趣的空位之间配对的方法。

计算机网络。计算机网络是由能够发送、转发和接收各种消息的站点互相连接组成的。我们感
516 兴趣的是这种互联结构的性质，因为我们希望网络中的线路和交换设备能够高效率地处理网络流量。

软件。编译器会使用图来表示大型软件系统中各个模块之间的关系。图中的结点即构成整个系统的各种类和模块，连接则为类的方法之间的可能调用关系（静态分析），或是系统运行时的实际调用关系（动态分析）。我们需要分析这幅图来决定如何以最优的方式为程序分配资源。

社交网络。当你在使用社交网站时，会和你的朋友之间建立起明确的关系。这里，结点对应人而连接则联系着你和你的朋友或是关注者。分析这些社交网络的性质是当前图算法的一个重要应用。对它感兴趣的不止是社交网络的公司，还包括政治、外交、娱乐、教育、市场等许多其他机构（参见表 4.0.1）。

表 4.0.1 图的典型应用

应 用	结 点	连 接
地图	十字路口	公路
网络内容	网页	超链接
电路	元器件	导线
任务调度	任务	限制条件
商业交易	客户	交易
配对	学生	申请
计算机网络	网站	物理连接
软件	方法	调用关系
社交网络	人	友谊关系

这些示例展示了图作为一种抽象模型的应用范围以及我们在处理图时可能会遇到的各种计算问题。人们研究过的关于图的问题数以千计，但它们大多数都能用一些简单的图模型解决——本章我们将会学习几个最重要的模型。在实际应用中，处理庞大的数据是很常见的，因此解决方法是否可行完全取决于算法的效率。

在本章中，我们会依次学习 4 种最重要的图模型：*无向图*（简单连接）、*有向图*（连接有方向
517 性）、*加权图*（连接带有权值）和*加权有向图*（连接既有方向性又带有权值）。

4.1 无向图

在我们首先要学习的这种图模型中，边（edge）仅仅是两个顶点（vertex）之间的连接。为了和其他图模型相区别，我们将它称为无向图。这是一种最简单的图模型，我们先来看一下它的定义。

定义。图是由一组顶点和一组能够将两个顶点相连的边组成的。

就定义而言，顶点叫什么名字并不重要，但我们需要一个方法来指代这些顶点。一般使用0至V-1来表示一张含有V个顶点的图中的各个顶点。这样约定是为了方便使用数组的索引来编写能够高效访问各个顶点中信息的代码。用一张符号表来为顶点的名字和0到V-1的整数值建立一一对应的关系并不困难（请见4.1.7节），因此直接使用数组索引作为结点的名称更方便且不失一般性（也不会损失什么效率）。我们用v-w的记法来表示连接v和w的边，w-v是这条边的另一种表示方法。

在绘制一幅图时，用圆圈表示顶点，用连接两个顶点的线段表示边，这样就能直观地看出图的结构。但这种直觉有时也可能会误导我们，因为图的定义和绘出的图像是无关的。例如，图4.1.1中的两组图表示的是同一幅图，因为图的构成只有（无序的）顶点和边（顶点对）。

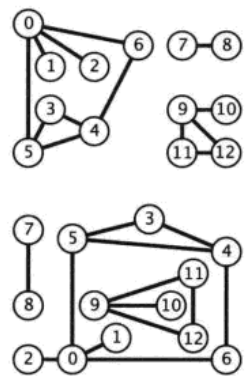


图4.1.1 同一幅图的两种表示

特殊的图。我们的定义允许出现两种简单而特殊的情况，参见图4.1.2：

- 自环，即一条连接一个顶点和其自身的边；
- 连接同一对顶点的两条边称为平行边。

数学家常常将含有平行边的图称为多重图，而将没有平行边或自环的图称为简单图。一般来说，实现允许出现自环和平行边（因为它们会在实际应用中出现），但我们不会将它们作为示例。因此，我们用两个顶点就可以指代一条边了。

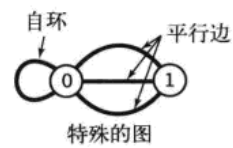


图4.1.2 特殊的图

518

4.1.1 术语表

和图有关的术语非常多，其中大多数定义都很简单，我们在这里集中介绍。

当两个顶点通过一条边相连时，我们称这两个顶点是*相邻的*，并称该连接*依附于*这两个顶点。某个顶点的*度数*即为依附于它的边的总数。*子图*是由一幅图的所有边的一个子集（以及它们所依附的所有顶点）组成的图。许多计算问题都需要识别各种类型的子图，特别是由能够顺序连接一系列顶点的边所组成的子图。

定义。在图中，*路径*是由边顺序连接的一系列顶点。*简单路径*是一条没有重复顶点的路径。*环*是一条至少含有一条边且起点和终点相同的路径。*简单环*是一条（除了起点和终点必须相同之外）不含有重复顶点和边的环。路径或者环的*长度*为其中所包含的边数。

大多数情况下，我们研究的都是简单环和简单路径并会省略掉简单二字。当允许重复的顶点时，我们指的都是一般的路径和环。当两个顶点之间存在一条连接双方的路径时，我们称一个顶点和另

一个顶点是连通的。我们用类似 u-v-w-x 的记法来表示 u 到 x 的一条路径，用 u-v-w-x-u 表示从 u 到 v 到 w 到 x 再回到 u 的一条环。我们会学习几种查找路径和环的算法。另外，路径和环也会帮我们从整体上考虑一幅图的性质，参见图 4.1.3。

定义。如果从任意一个顶点都存在一条路径到达另一个任意顶点，我们称这幅图是**连通图**。一幅非连通的图由若干连通的部分组成，它们都是其**极大连通子图**。

直观上来说，如果顶点是物理存在的对象，例如绳节或是念珠，而边也是物理存在的对象，例如绳子或是电线，那么将任意顶点提起，连通图都将是一个整体，而非连通图则会变成两个或多个部分。一般来说，要处理一张图就需要一个个地处理它的连通分量（子图）。

*无环图*是一种不包含环的图。我们将要学习的几个算法就是要找出一幅图中满足一定条件的无环子图。我们还需要一些术语来表示这些结构。

定义。树是一幅无环连通图。互不相连的树组成的集合称为*森林*。连通图的*生成树*是它的一幅子图，它含有图中的所有顶点且是一棵树。图的*生成树森林*是它的所有连通子图的生成树的集合，参见图 4.1.4 和图 4.1.5。

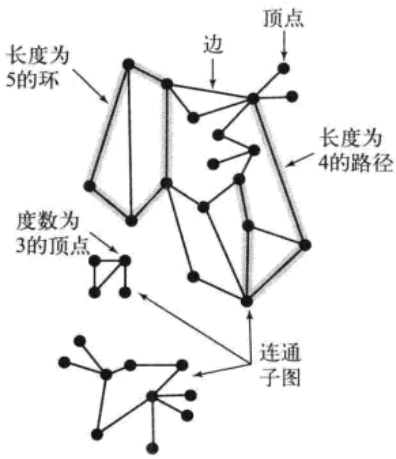


图 4.1.3 图的详解

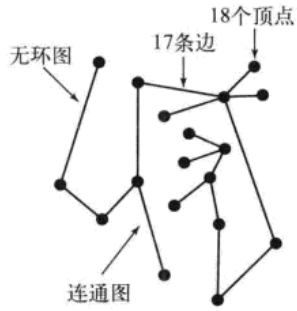


图 4.1.4 一棵树

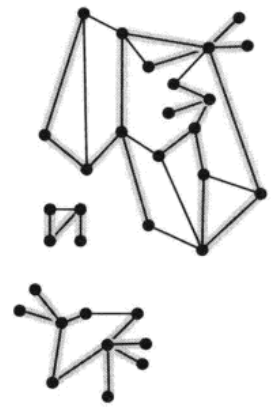


图 4.1.5 生成树森林

树的定义非常通用，稍做改动就可以变成用来描述程序行为的（函数调用层次）模型和数据结构（二叉查找树、2-3 树等）。树的数学性质很直观并且已被系统地研究过，因此我们就不给出它们的证明了。例如，当且仅当一幅含有 V 个结点的图 G 满足下列 5 个条件之一时，它就是一棵树：

- G 有 V–1 条边且不含有环；
- G 有 V–1 条边且是连通的；
- G 是连通的，但删除任意一条边都会使它不再连通；
- G 是无环图，但添加任意一条边都会产生一条环；
- G 中的任意一对顶点之间仅存在一条简单路径。

我们会学习几种寻找生成树和森林的算法，以上这些性质在分析和实现这些算法的过程中扮演着重要的角色。

图的密度是指已经连接的顶点对占所有可能被连接的顶点对的比例。在稀疏图中，被连接的顶点对很少；而在稠密图中，只有少部分顶点对之间没有边连接。一般来说，如果一幅图中不同的边的数量只占顶点总数 V 的一小部分，那么我们就认为这幅图是稀疏的，否则则是稠密的，参见图 4.1.6。这条经验规律虽然会留下一片灰色地带（比如当边的数量为 ~ $cV^{3/2}$ 时），但实际应用中稀疏图和稠密图之间的区别是十分明显的。我们将会遇到的应用使用的几乎都是稀疏图。

二分图是一种能够将所有结点分为两部分的图，其中图的每条边所连接的两个顶点都分别属于不同的部分。图 4.1.7 即为一幅二分图的示例，其中红色的结点是一个集合，黑色的结点是另一个集合。二分图会出现在许多场景中，我们会在本节的最后详细研究其中的一个场景。

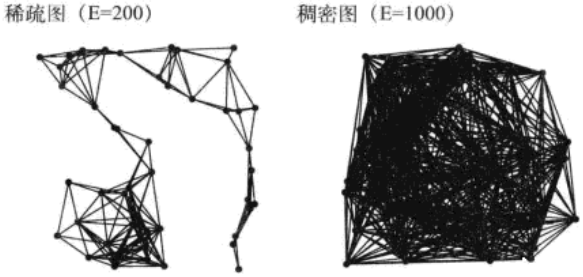


图 4.1.6 两幅图 (V=50)

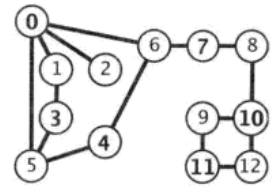


图 4.1.7 二分图（另见彩插）

现在，我们已经做好了学习图处理算法的准备。我们首先会研究一种表示图的数据类型的 API 及其实现，然后会学习一些查找图和鉴别连通分量的经典算法。最后，我们会考虑真实世界中的一些图的应用，它们的顶点的名字可能不是整数并且会含有数目庞大的顶点和边。

519 ≀ 521

4.1.2 表示无向图的数据类型

要开发处理图问题的各种算法，我们首先来看一份定义了图的基本操作的 API，参见表 4.1.1。有了它我们才能完成从简单的基本操作到解决复杂问题的各种任务。

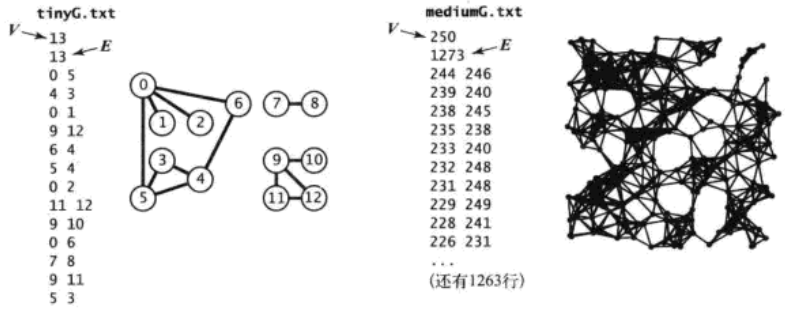
表 4.1.1 无向图的 API

public class	**Graph**	
	Graph(int V)	创建一个含有 V 个顶点但不含有边的图
	Graph(In in)	从标准输入流 in 读入一幅图
int	V()	顶点数
int	E()	边数
void	addEdge(int v, int w)	向图中添加一条边 v-w
Iterable<Integer>	adj(int v)	和 v 相邻的所有顶点
String	toString()	对象的字符串表示

这份 API 含有两个构造函数，有两个方法用来分别返回图中的顶点数和边数，有一个方法用来添加一条边，toString() 方法和 adj() 方法用来允许用例遍历给定顶点的所有相邻顶点（遍历顺序不确定）。值得注意的是，本节将学习的所有算法都基于 adj() 方法所抽象的基本操作。

第二个构造函数接受的输入由 $2E+2$ 个整数组成：首先是 V，然后是 E，再然后是 E 对 0 到 $V-1$ 之间的整数，每个整数对都表示一条边。例如，我们使用了由图 4.1.8 中的 tinyG.txt 和 mediumG.txt 所描述的两个示例。

调用 Graph 的几段用例代码请见表 4.1.2。



522

图 4.1.8 Graph 的构造函数的输入格式（两个示例）

表 4.1.2 最常用的图处理代码

任 务	实 现
计算 v 的度数	`public static int degree(Graph G, int v)` `{` `  int degree = 0;` `  for (int w : G.adj(v)) degree++;` `  return degree;` `}`
计算所有顶点的最大度数	`public static int maxDegree(Graph G)` `{` `  int max = 0;` `  for (int v = 0; v < G.V(); v++)` `  if (degree(G, v) > max)` `  max = degree(G, v);` `  return max;` `}`
计算所有顶点的平均度数	`public static double avgDegree(Graph G)` `{ return 2 * G.E() / G.V(); }`
计算自环的个数	`public static int numberOfSelfLoops(Graph G)` `{` `  int count = 0;` `  for (int v = 0; v < G.V(); v++)` `  for (int w : G.adj(v))` `  if (v == w) count++;` `  return count/2;  // 每条边都被记过两次` `}`
图的邻接表的字符串表示（Graph 的实例方法）	`public String toString()` `{` `  String s = V + " vertices, " + E + " edges\n";` `  for (int v = 0; v < V; v++)` `  {` `  s += v + ": ";` `  for (int w : this.adj(v))` `  s += w + " ";` `  s += "\n";` `  }` `  return s;` `}`

523

4.1.2.1 图的几种表示方法

我们要面对的下一个图处理问题就是用哪种方式（数据结构）来表示图并实现这份 API，这包含以下两个要求：

- 它必须为可能在应用中碰到的各种类型的图预留出足够的空间；
- Graph 的实例方法的实现一定要快——它们是开发处理图的各种用例的基础。

这些要求比较模糊，但它们仍然能够帮助我们在三种图的表示方法中进行选择。

- *邻接矩阵*。我们可以使用一个 V 乘 V 的布尔矩阵。当顶点 v 和顶点 w 之间有相连接的边时，定义 v 行 w 列的元素值为 true，否则为 false。这种表示方法不符合第一个条件——含有上百万个顶点的图是很常见的，V^2 个布尔值所需的空间是不能满足的。
- *边的数组*。我们可以使用一个 Edge 类，它含有两个 int 实例变量。这种表示方法很简洁但不满足第二个条件——要实现 adj() 需要检查图中的所有边。
- *邻接表数组*。我们可以使用一个以顶点为索引的列表数组，其中的每个元素都是和该顶点相邻的顶点列表，参见图 4.1.9。这种数据结构能够同时满足典型应用所需的以上两个条件，我们会在本章中一直使用它。

除了这些性能目标之外，经过缜密的检查，我们还发现了另一些在某些应用中可能会很重要的东西。例如，允许存在平行边相当于排除了邻接矩阵，因为邻接矩阵无法表示它们。

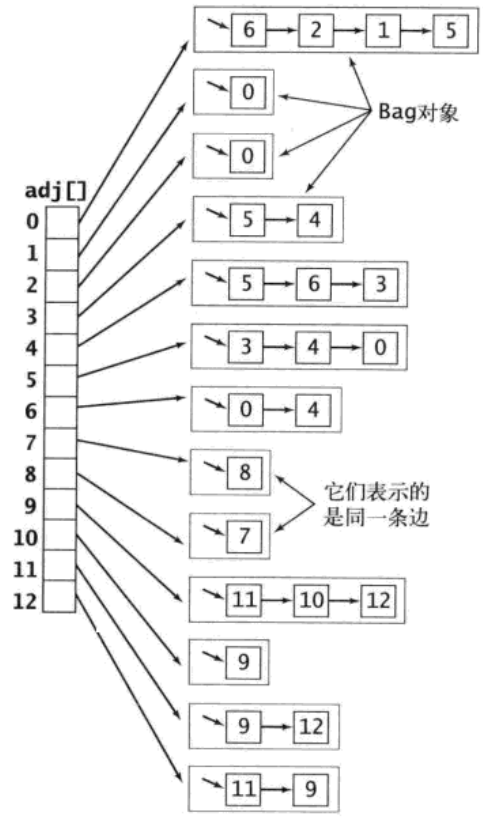


图 4.1.9 邻接表数组示意（无向图）

4.1.2.2 邻接表的数据结构

非稠密图的标准表示称为*邻接表*的数据结构，它将每个顶点的所有相邻顶点都保存在该顶点对应的元素所指向的一张链表中。我们使用这个数组就是为了快速访问给定顶点的邻接顶点列表。这里使用 1.3 节中的 Bag 抽象数据类型来实现这个链表，这样我们就可以在常数时间内添加新的边或遍历任意顶点的所有相邻顶点。后面框注“Graph 数据类型”中的 Graph 类的实现就是基于这种方法，而图 4.1.9 中所示的正是用这种方法处理 tinyG.txt 所得到的数据结构。要添加一条连接 v 与 w 的边，我们将 w 添加到 v 的邻接表中并把 v 添加到 w 的邻接表中。因此，在这个数据结构中每条边都会出现两次。这种 Graph 的实现的性能有如下特点：

- 使用的空间和 $V+E$ 成正比；

524

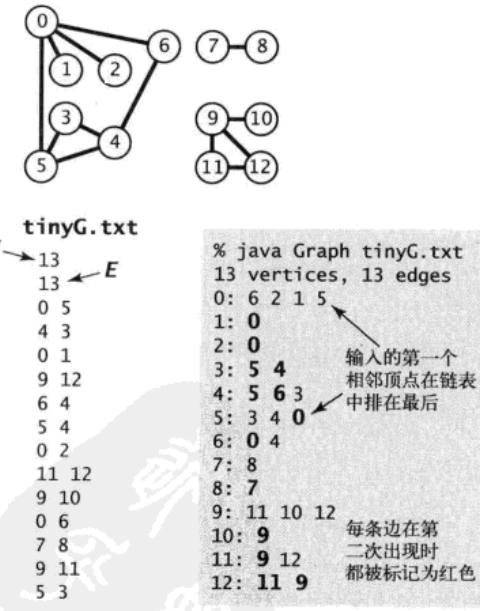


图 4.1.10 由边得到的邻接表（另见彩插）

- 添加一条边所需的时间为常数；
- 遍历顶点 v 的所有相邻顶点所需的时间和 v 的度数成正比（处理每个相邻顶点所需的时间为常数）。

对于这些操作，这样的特性已经是最优的了，这已经可以满足图处理应用的需要，而且支持平行边和自环（我们不会检测它们）。注意，边的插入顺序决定了 Graph 的邻接表中顶点的出现顺序，参见图 4.1.10。多个不同的邻接表可能表示着同一幅图。当使用构造函数从标准输入中读入一幅图时，这就意味着输入的格式和边的顺序决定了 Graph 的邻接表数组中顶点的出现顺序。因为算法在使用 adj() 来处理所有相邻的顶点时不会考虑它们在邻接表中的出现顺序，这种差异不会影响算法的正确性，但在调试或是跟踪邻接表的轨迹时我们还是需要注意这一点。为了简化操作，假设 Graph 有一个测试用例来从命令行参数指定的文件中读取一幅图并将它打印出来（参见表 4.1.2 中的 toString()
525 方法的实现），以显示邻接表中的各个顶点的出现顺序，这也是算法处理它们的顺序（请见练习 4.1.7）。

Graph 数据类型

```
public class Graph
{
   private final int V;           // 顶点数目
   private int E;                 // 边的数目
   private Bag<Integer>[] adj;    // 邻接表
   public Graph(int V)
   {
      this.V = V; this.E = 0;
      adj = (Bag<Integer>[]) new Bag[V];       // 创建邻接表
      for (int v = 0; v < V; v++)              // 将所有链表初始化为空
         adj[v] = new Bag<Integer>();
   }
   public Graph(In in)
   {
      this(in.readInt());           // 读取V并将图初始化
      int E = in.readInt();         // 读取E
      for (int i = 0; i < E; i++)
      {  // 添加一条边
         int v = in.readInt();      // 读取一个顶点
         int w = in.readInt();         // 读取另一个顶点
         addEdge(v, w);                // 添加一条连接它们的边
      }
   }
   public int V()  {  return V;  }
   public int E()  {  return E;  }
   public void addEdge(int v, int w)
   {
      adj[v].add(w);                             // 将w添加到v的链表中
      adj[w].add(v);                             // 将v添加到W的链表中
      E++;
    }
   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

这份 Graph 的实现使用了一个由顶点索引的整型链表数组。每条边都会出现两次，即当存在一条连接 v 与 w 的边时，w 会出现在 v 的链表中，v 也会出现在 w 的链表中。第二个构造函数从输入流中读取一
526 幅图，开头是 V，然后是 E，再然后是一列整数对，大小在 0 到 V–1 之间。toString() 方法请见表 4.1.2。

在实际应用中还有一些操作可能是很有用的，例如：

- 添加一个顶点；
- 删除一个顶点。

实现这些操作的一种方法是扩展之前的API，使用符号表（ST）来代替由顶点索引构成的数组（这样修改之后就不需要约定顶点名必须是整数了）。我们可能还需要：

- 删除一条边；
- 检查图是否含有边 v-w。

要实现这些方法（不允许存在平行边），我们可能需要使用 SET 代替 Bag 来实现邻接表。我们称这种方法为*邻接集*。本书中不会使用这些数据结构，因为：

- 用例代码不需要添加顶点、删除顶点和边或是检查一条边是否存在；
- 当用例代码需要进行上述操作时，由于频率很低或者相关的邻接链表很短，因此可以直接使用穷举法遍历链表来实现；
- 使用 SET 和 ST 会令算法的实现变得更加复杂，分散了读者对算法本身的注意力；
- 在某些情况下，它们会使性能损失 $\log V$。

使我们的算法适应其他设计（例如，不允许出现平行边或是自环）并避免不必要的性能损失并不困难。表 4.1.3 总结了之前提到过的所有其他实现方法的性能特点。常见的应用场景都需要处理庞大的稀疏图，因此我们会一直使用邻接表。

表 4.1.3 典型 Graph 实现的性能复杂度

数据结构	所需空间	添加一条边 v-w	检查 w 和 v 是否相邻	遍历 v 的所有相邻顶点
边的列表	E	1	E	E
邻接矩阵	V^2	1	1	V
邻接表	$E+V$	1	*degree*(v)	*degree*(v)
邻接集	$E+V$	$\log V$	$\log V$	$\log V$+*degree*(v)

527

4.1.2.3 图的处理算法的设计模式

因为我们会讨论大量关于图处理的算法，所以设计的首要目标是将图的表示和实现分离开来。为此，我们会为每个任务创建一个相应的类，用例可以创建相应的对象来完成任务。类的构造函数一般会在预处理中构造各种数据结构，以有效地响应用例的请求。典型的用例程序会构造一幅图，将图传递给实现了某个算法的类（作为构造函数的参数），然后调用用例的方法来获取图的各种性质。作为热身，我们先来看看这份 API，参见表 4.1.4。

表 4.1.4 图处理算法的 API（热身）

public class	Search	
	Search(Graph G, int s)	找到和起点 s 连通的所有顶点
boolean	marked(int v)	v 和 s 是连通的吗
int	count()	与 s 连通的顶点总数

我们用起点（source）区分作为参数传递给构造函数的顶点与图中的其他顶点。在这份 API 中，构造函数的任务是找到图中与起点连通的其他顶点。用例可以调用 marked() 方法和 count() 方

法来了解图的性质。方法名 marked() 指的是这种基本算法使用的一种实现方式，本章中会一直使用到这种算法：在图中从起点开始沿着路径到达其他顶点并标记每个路过的顶点。后面框注中的图处理用例 TestSearch 接受由命令行得到的一个输入流的名称和起始结点的编号，从输入流中读取一幅图（使用 Graph 的第二个构造函数），用这幅图和给定的起始结点创建一个 Search 对象，然后用 marked() 打印出图中和起点连通的所有顶点。它也调用了 count() 并打印了图是否是连通的（当且仅当搜索能够标记图中的所有顶点时图才是连通的）。

528

```
public class TestSearch
{
   public static void main(String[] args)
   {
      Graph G = new Graph(new In(args[0]));
      int s = Integer.parseInt(args[1]);
      Search search = new Search(G, s);

      for (int v = 0; v < G.V(); v++)
         if (search.marked(v))
            StdOut.print(v + " ");
      StdOut.println();

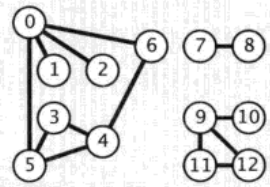
      if (search.count() != G.V())
         StdOut.print("NOT ");
      StdOut.println("connected");
   }
}
```

图处理的用例（热身）

```
% java TestSearch tinyG.txt 0
0 1 2 3 4 5 6
NOT connected

% java TestSearch tinyG.txt 9
9 10 11 12
NOT connected
```

```
tinyG.txt
V
13
13          E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



我们已经见过 Search API 的一种实现：第 1 章中的 union-find 算法。它的构造函数会创建一个 UF 对象，对图中的每一条边进行一次 union() 操作并调用 connected(s,v) 来实现 marked(v) 方法。实现 count() 方法需要一个加权的 UF 实现并扩展它的 API，以便使用 count() 方法返回 wt[find(v)]（请见练习 4.1.8）。这种实现简单而高效，但下面我们要学习的实现还可以更进一步。它基于的是*深度优先搜索*（DFS）的。这是一种重要的递归方法，它会沿着图的边寻找和起点连通的所有顶点。深度优先搜索是本章中将学习的好几种关于图的算法的基础。

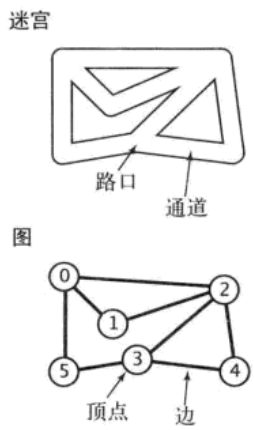


图 4.1.11 等价的迷宫模型

4.1.3 深度优先搜索

我们常常通过系统地检查每一个顶点和每一条边来获取图的各种性质。要得到图的一些简单性质（比如，计算所有顶点的度数）很容易，只要检查每一条边即可（任意顺序）。但图的许多其他性质和路径有关，因此一种很自然的想法是沿着图的边从一个顶点移动到另一个顶点。尽管存在各种各样的处理策略，但后面将要学习的几乎所有与图有关的算法都使用了这个简单的抽象模型，其中最

简单的就是下面介绍的这种经典的方法。

4.1.3.1 走迷宫

思考图的搜索过程的一种有益的方法是，考虑另一个和它等价但历史悠久而又特别的问题——在一个由各种通道和路口组成的迷宫中找到出路。有些迷宫的规则很简单，但大多数迷宫则需要很复杂的策略才行。用迷宫代替图、通道代替边、路口代替顶点仅仅只是一些文字游戏，但就目前来说，这么做可以帮助我们直观地认识问题，参见图 4.1.11。探索迷宫而不迷路的一种古老办法(至少可以追溯到忒修斯和米诺陶的传说)叫做 Tremaux 搜索，参见图 4.1.12。要探索迷宫中的所有通道，我们需要：

- 选择一条没有标记过的通道，在你走过的路上铺一条绳子；
- 标记所有你第一次路过的路口和通道；
- 当来到一个标记过的路口时(用绳子)回退到上个路口；
- 当回退到的路口已没有可走的通道时继续回退。

绳子可以保证你总能找到一条出路，标记则能保证你不会两次经过同一条通道或者同一个路口。要知道是否完全探索了整个迷宫需要的证明更复杂，只有用图搜索才能够更好地处理问题。Tremaux 搜索很直接，但它与完全搜索一张图仍然稍有不同，因此我们接下来看看图的搜索方法。

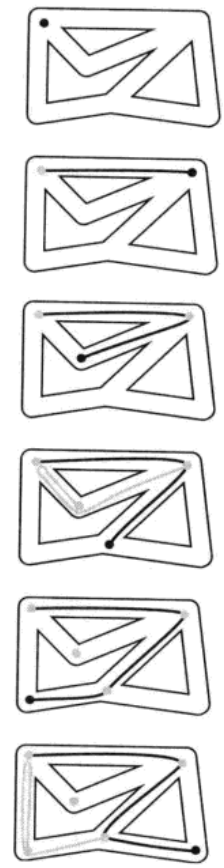


图 4.1.12 Tremaux 搜索(另见彩插)

```
public class DepthFirstSearch
{
   private boolean[] marked;
   private int count;

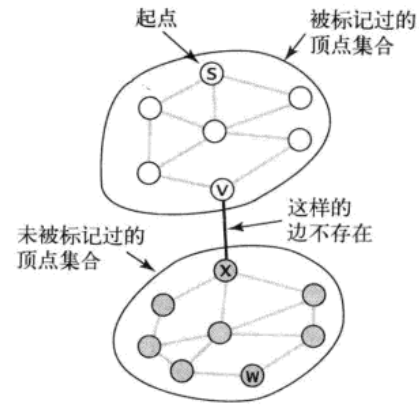
   public DepthFirstSearch(Graph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      count++;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean marked(int w)
   {  return marked[w];  }

   public int count()
   {  return count;  }
}
```

深度优先搜索



4.1.3.2 热身

搜索连通图的经典递归算法（遍历所有的顶点和边）和 Tremaux 搜索类似，但描述起来更简单。要搜索一幅图，只需用一个递归方法来遍历所有顶点。在访问其中一个顶点时：

- 将它标记为已访问；
- 递归地访问它的所有没有被标记过的邻居顶点。

这种方法称为深度优先搜索（DFS）。Search API 的一种实现使用了这种方法，如深度优先搜索框注所示。它使用一个 boolean 数组来记录和起点连通的所有顶点。递归方法会标记给定的顶点并调用自己来访问该顶点的相邻顶点列表中所有没有被标记过的顶点。如果图是连通的，每个邻接链表中的元素都会被检查到。

命题 A。深度优先搜索标记与起点连通的所有顶点所需的时间和顶点的度数之和成正比。

证明。首先，我们要证明这个算法能够标记与起点 s 连通的所有顶点（且不会标记其他顶点）。因为算法仅通过边来寻找顶点，所以每个被标记过的顶点都与 s 连通。现在，假设某个没有被标记过的顶点 w 与 s 连通。因为 s 本身是被标记过的，由 s 到 w 的任意一条路径中至少有一条边连接的两个顶点分别是被标记过的和没有被标记过的，例如 v-x。根据算法，在标记了 v 之后必然会发现 x，因此这样的边是不存在的。前后矛盾。每个顶点都只会被访问一次保证了时间上限（检查标记的耗时和度数成正比）。

529 ~ 531

4.1.3.3 单向通道

代码中方法的调用和返回机制对应迷宫中绳子的作用：当已经处理过依附于一个顶点的所有边时（搜索了路口连接的所有通道），我们就只能“返回”（return，两者的意义相同）。为了更好地与迷宫的 Tremaux 搜索对应起来，我们可以想象一座完全由单向通道构造的迷宫（每个方向都有一个通道）。和在迷宫中会经过一条通道两次（方向不同）一样，在图中我们也会路过每条边两次（在它的两个端点各一次）。在 Tremaux 搜索中，要么是第一次访问一条边，要么是沿着它从一个被标记过的顶点退回。在无向图的深度优先搜索中，在碰到边 v-w 时，要么进行递归调用（w 没有被标记过），要么跳过这条边（w 已经被标记过）。第二次从另一个方向 w-v 遇到这条边时，总是会忽略它，因为它的另一端 v 肯定已经被访问过了（在第一次遇到这条边的时候）。

4.1.3.4 跟踪深度优先搜索

通常，理解算法的最好方法是在一个简单的例子中跟踪它的行为。深度优先算法尤其是这样。在跟踪它的轨迹时，首先要注意的是，算法遍历边和访问顶点的顺序与图的表示是有关的，而不只是与

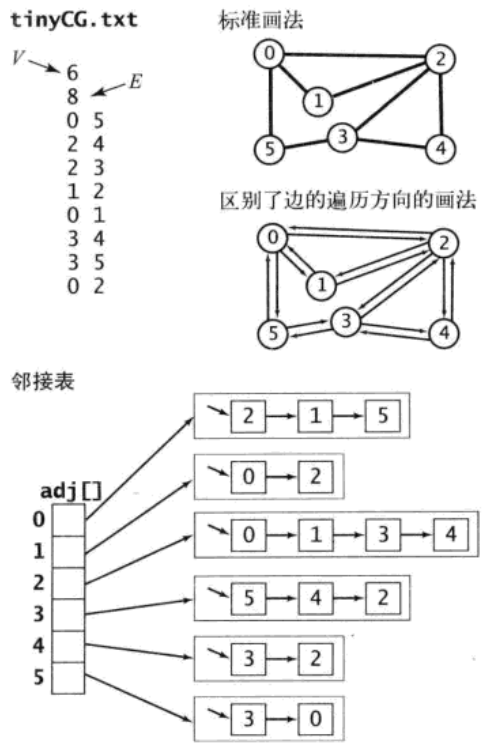


图 4.1.13 一幅连通的无向图

图的结构或是算法有关。因为深度优先搜索只会访问和起点连通的顶点，所以使用图 4.1.13 所示的一幅小型连通图为例。在示例中，顶点 2 是顶点 0 之后第一个被访问的顶点，因为它正好是 0 的邻接表的第一个元素。要注意的第二点是，如前文所述，深度优先搜索中每条边都会被访问两次，且在第二次时总会发现这个顶点已经被标记过。这意味着深度优先搜索的轨迹可能会比你想象的长一倍！示例图仅含有 8 条边，但需要追踪算法在邻接表的 16 个元素上的操作。

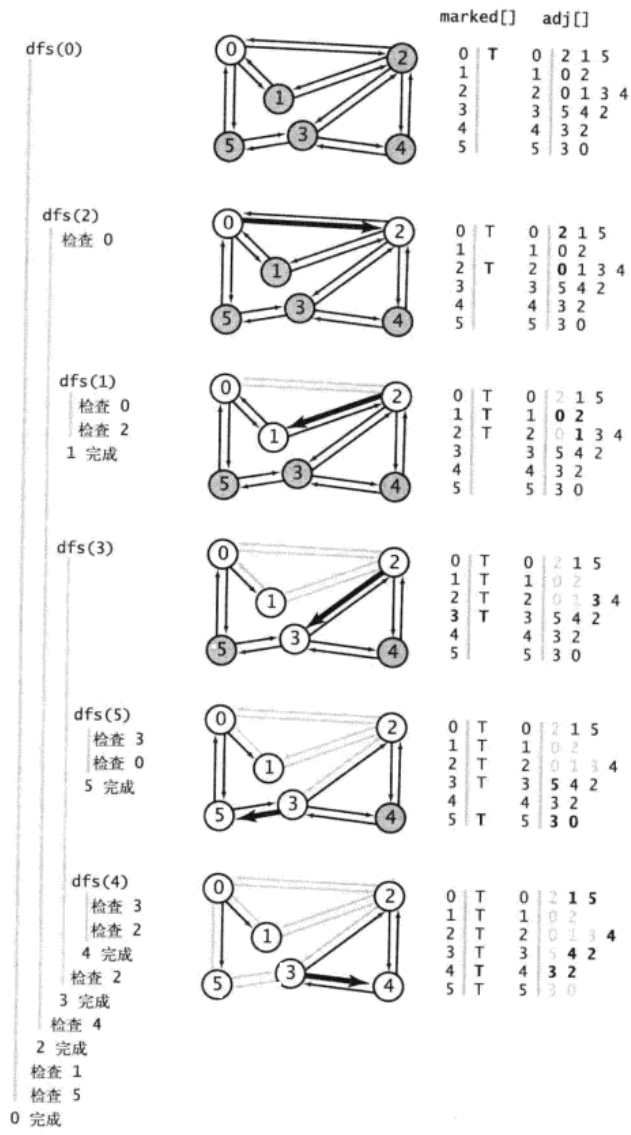


图 4.1.14 使用深度优先搜索的轨迹，寻找所有和顶点 0 连通的顶点（另见彩插）

4.1.3.5 深度优先搜索的详细轨迹

图 4.1.14 显示的是示例中每个顶点被标记后算法使用的数据结构，起点为顶点 0。查找开始于构造函数调用递归的 dfs() 来标记和访问顶点 0，后续处理如下所述。

- 因为顶点 2 是 0 的邻接表的第一个元素且没有被标记过，dfs() 递归调用自己来标记并访问顶点 2（效果是系统会将顶点 0 和 0 的邻接表的当前位置压入栈中）。
- 现在，顶点 0 是 2 的邻接表的第一个元素且已经被标记过了，因此 dfs() 跳过了它。接下来，顶点 1 是 2 的邻接表的第二个元素且没有被标记，dfs() 递归调用自己来标记并访问顶点 1。
- 对顶点 1 的访问和前面有所不同：因为它的邻接表中的所有顶点（0 和 2）都已经被标记过了，因此不需要再进行递归，方法从 dfs(1) 中返回。下一条被检查的边是 2-3（在 2 的邻接表中顶点 1 之后的顶点是 3），因此 dfs() 递归调用自己来标记并访问顶点 3。
- 顶点 5 是 3 的邻接表的第一个元素且没有被标记，因此 dfs() 递归调用自己来标记并访问顶点 5。
- 顶点 5 的邻接表中的所有顶点（3 和 0）都已经被标记过了，因此不需要再进行递归。
- 顶点 4 是 3 的邻接表的下一个元素且没有被标记过，因此 dfs() 递归调用自己来标记并访问顶点 4。这是最后一个需要被标记的顶点。

532 ~ 533

- 在顶点4被标记了之后，dfs()会检查它的邻接表，然后再检查3的邻接表，然后是2的邻接表，然后是0的，最后发现不需要再进行任何递归调用，因为所有的顶点都已经被标记过了。

这种简单的递归模式只是一个开始——深度优先搜索能够有效处理许多和图有关的任务。例如，本节中，我们已经可以用深度优先搜索来解决在第1章首次提到的一个问题。

连通性。给定一幅图，回答“两个给定的顶点是否连通？”或者“图中有多少个连通子图？”等类似问题。

我们可以轻易地用处理图问题的标准设计模式给出这些问题的答案，还要将这些解答与在1.5节中学习的union-find算法进行比较。

问题“两个给定的顶点是否连通？”等价于“两个给定的顶点之间是否存在一条路径？”，也许也可以叫做路径检测问题。但是，在1.5节学习的union-find算法的数据结构并不能解决找出这样一条路径的问题。深度优先搜索是我们已经学习过的几种方法中第一个能够解决这个问题的算法。它能够解决的另一个问题如下所述。

单点路径。给定一幅图和一个起点s，回答“从s到给定目的顶点v是否存在一条路径？如果有，找出这条路径。”等类似问题。

深度优先搜索算法之所以极为简单，是因为它所基于的概念为人所熟知并且非常容易实现。事实上，它是一个既小巧而又强大的算法，研究人员用它解决了无数困难的问题。上述两个问题只是
534 我们将要研究的许多问题的开始。

4.1.4 寻找路径

单点路径问题在图的处理领域中十分重要。根据标准设计模式，我们将使用如下API（请见表4.1.5）。

表4.1.5 路径的API

public class	Paths	
	Paths(Graph G, int s)	在G中找出所有起点为s的路径
boolean	hasPathTo(int v)	是否存在从s到v的路径
Iterable<Integer>	pathTo(int v)	s到v的路径，如果不存在则返回null

构造函数接受一个起点s作为参数，计算s到与s连通的每个顶点之间的路径。在为起点s创建了Paths对象后，用例可以调用pathTo()实例方法来遍历从s到任意和s连通的顶点的路径上的所有顶点。现在暂时查找所有路径，以后会实现只查找具有某些属性的路径。

```
% java Paths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-2-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-3-4
0 to 5: 0-2-3-5
```

535

```
public static void main(String[] args)
{
   Graph G = new Graph(new In(args[0]));
   int s = Integer.parseInt(args[1]);
   Paths search = new Paths(G, s);
   for (int v = 0; v < G.V(); v++)
   {
      StdOut.print(s + " to " + v + ": ");
      if (search.hasPathTo(v))
         for (int x : search.pathTo(v))
            if (x == s) StdOut.print(x);
            else StdOut.print("-" + x);
      StdOut.println();
   }
}
```

Paths实现的测试用例

上一页右下角框注中的用例从输入流中读取了一个图并从命令行得到一个起点，然后打印出从起点到与它连通的每个顶点之间的一条路径。

4.1.4.1 实现

算法 4.1 基于深度优先搜索实现了 Paths。它扩展了 4.1.3.2 节中的热身代码 DepthFirstSearch，添加了一个实例变量 edgeTo[] 整型数组来起到 Tremaux 搜索中绳子的作用。这个数组可以找到从每个与 s 连通的顶点回到 s 的路径。它会记住每个顶点到起点的路径，而不是记录当前顶点到起点的路径。为了做到这一点，在由边 v-w 第一次访问任意 w 时，将 edgeTo[w] 设为 v 来记住这条路径。换句话说，v-w 是从 s 到 w 的路径上的最后一条已知的边。这样，搜索的结果是一棵以起点为根结点的树，edgeTo[] 是一棵由父链接表示的树。算法 4.1 的代码的右侧是一个小示例。要找出 s 到任意顶点 v 的路径，算法 4.1 实现的 pathTo() 方法用变量 x 遍历整棵树，将 x 设为 edgeTo[x]，就像 1.5 节中的 union-find 算法一样，然后在到达 s 之前，将遇到的所有顶点都压入栈中。将这个栈返回为一个 Iterable 对象帮助用例遍历 s 到 v 的路径。

算法 4.1 使用深度优先搜索查找图中的路径

```
public class DepthFirstPaths
{
   private boolean[] marked; // 这个顶点上调用过dfs()了吗?
   private int[] edgeTo;     // 从起点到一个顶点的已知路径上的最后一个顶点
   private final int s;      // 起点

   public DepthFirstPaths(Graph G, int s)
   {
      marked = new boolean[G.V()];
      edgeTo = new int[G.V()];
      this.s = s;
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))

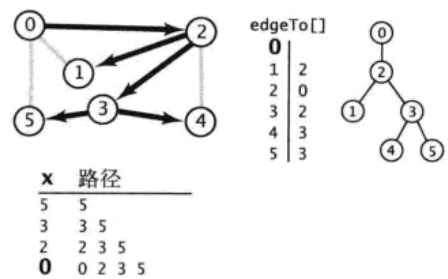
         if (!marked[w])

         {

            edgeTo[w] = v;
            dfs(G, w);
         }
   }

   public boolean hasPathTo(int v)
   {  return marked[v];  }

   public Iterable<Integer> pathTo(int v)
   {
      if (!hasPathTo(v)) return null;
      Stack<Integer> path = new Stack<Integer>();
      for (int x = v; x != s; x = edgeTo[x])
         path.push(x);
      path.push(s);
      return path;
   }
}
```



x	路径
5	5
3	3 5
2	2 3 5
0	0 2 3 5

pathTo(5) 的计算轨迹

这段 Graph 的用例使用了深度优先搜索，以找出图中从给定的起点 s 到它连通的所有顶点的路径。

来自 DepthFirstSearch（4.1.3.2 节）的代码均为灰色。为了保存到达每个顶点的已知路径，这段代码使用了一个以顶点编号为索引的数组 edgeTo[]，edgeTo[w]=v 表示 v-w 是第一次访问 w 时经过的边。

536 edgeTo[] 数组是一棵用父链接表示的以 s 为根且含有所有与 s 连通的顶点的树。

4.1.4.2 详细轨迹

图 4.1.15 显示的是示例中每个顶点被标记后 edgeTo[] 的内容，起点为顶点 0。marked[] 和 adj[] 的内容与 4.1.3.5 节中的 DepthFirstSearch 的轨迹相同，递归调用和边检查的详细描述也完全一样，这里不再赘述。深度优先搜索向 edgeTo[] 数组中顺序添加了 0-2、2-1、2-3、3-5 和 3-4。这些边构成了一棵以起点为根结点的树并提供了 pathTo() 方法所需的信息，使得调用者可以按照前文所述的方法找到从 0 到顶点 1、2、3、4、5 的路径。

DepthFirstPaths 与 DepthFirstSearch 的构造函数仅有几条赋值语句不同，因此 4.1.3.2 节中的命题 A 仍然适用。另外，我们还有以下命题。

命题 A（续）。使用深度优先搜索得到从给定起点到任意标记顶点的路径所需的时间与路径的长度成正比。

证明。根据对已经访问过的顶点数量的归纳可得，DepthFirstPaths 中的 edgeTo[] 数组表示了一棵以起点为根结点的树。pathTo() 方法构造路径所需的时间和路径的长度成正比。

537

4.1.5 广度优先搜索

深度优先搜索得到的路径不仅取决于图的结构，还取决于图的表示和递归调用的性质。我们很自然地还经常对下面这些问题感兴趣。

单点最短路径。给定一幅图和一个起点 s，回答“从 s 到给定目的顶点 v 是否存在一条路径？如果有，找出其中最短的那条（所含边数最少）。”等类似问题。

解决这个问题的经典方法叫做广度优先搜索（BFS）。它也是许多图算法的基石，因此我们会

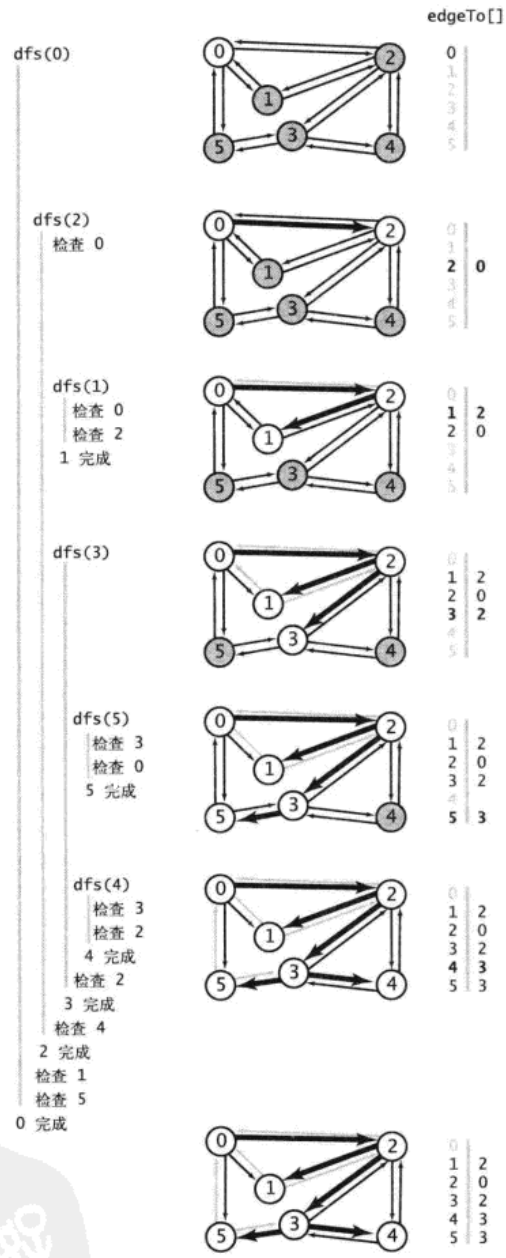


图 4.1.15 使用深度优先搜索的轨迹，寻找所有起点为 0 的路径（另见彩插）

在本节中详细学习。深度优先搜索在这个问题上没有什么作为，因为它遍历整个图的顺序和找出最短路径的目标没有任何关系。相比之下，广度优先搜索正是为了这个目标才出现的。要找到从 s 到 v 的最短路径，从 s 开始，在所有由一条边就可以到达的顶点中寻找 v，如果找不到我们就继续在与 s 距离两条边的所有顶点中查找 v，如此一直进行。深度优先搜索就好像是一个人在走迷宫，广度优先搜索则好像是一组人在一起朝各个方向走这座迷宫，每个人都有自己的绳子。当出现新的叉路时，可以假设一个探索者可以分裂为更多的人来搜索它们，当两个探索者相遇时，会合二为一（并继续使用先到达者的绳子），参见图 4.1.16。

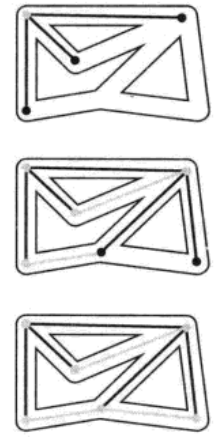


图 4.1.16 广度优先的迷宫搜索

在程序中，在搜索一幅图时遇到有多条边需要遍历的情况时，我们会选择其中一条并将其他通道留到以后再继续搜索。在深度优先搜索中，我们用了一个可以下压的栈（这是由系统管理的，以支持递归搜索方法）。使用 LIFO（后进先出）的规则来描述压栈和走迷宫时先探索相邻的通道类似。从有待搜索的通道中选择最晚遇到过的那条。在广度优先搜索中，我们希望按照与起点的距离的顺序来遍历所有顶点，看起来这种顺序很容易实现：使用（FIFO，先进先出）队列来代替栈（LIFO，后进先出）即可。我们将从有待搜索的通道中选择最早遇到的那条。

实现

算法 4.2 实现了广度优先搜索算法。它使用了一个队列来保存所有已经被标记过但其邻接表还未被检查过的顶点。先将起点加入队列，然后重复以下步骤直到队列为空：

- 取队列中的下一个顶点 v 并标记它；
- 将与 v 相邻的所有未被标记过的顶点加入队列。

538

算法 4.2 中的 `bfs()` 方法不是递归的。不像递归中隐式使用的栈，它显式地使用了一个队列。和深度优先搜索一样，它的结果也是一个数组 `edgeTo[]`，也是一棵用父链接表示的根结点为 s 的树。它表示了 s 到每个与 s 连通的顶点的最短路径。用例也可以使用算法 4.1 中为深度优先搜索实现的相同的 `pathTo()` 方法得到这些路径。

图 4.1.17 和图 4.1.18 显示了用广度优先搜索处理样图时，算法使用的数据结构在每次循环的迭代开始时的内容。首先，顶点 0 被加入队列，然后循环开始搜索。

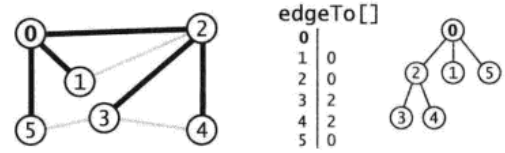
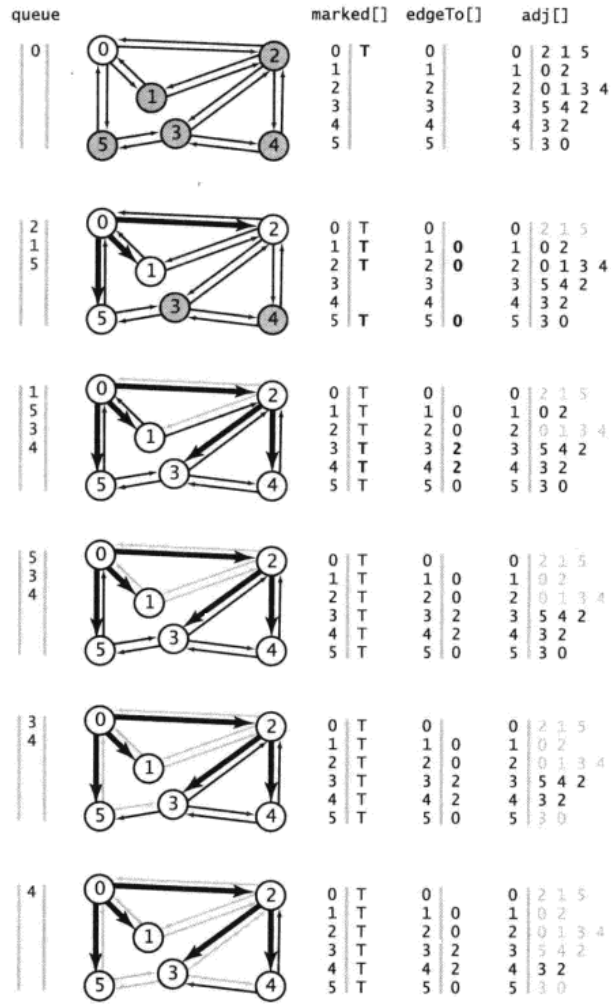


图 4.1.17 使用广度优先搜索寻找所有起点为 0 的路径的结果

- 从队列中删去顶点 0 并将它的相邻顶点 2、1 和 5 加入队列中，标记它们并分别将它们在 `edgeTo[]` 中的值设为 0。
- 从队列中删去顶点 2 并检查它的相邻顶点 0 和 1，发现两者都已经被标记。将相邻的顶点 3 和 4 加入队列，标记它们并分别将它们在 `edgeTo[]` 中的值设为 2。
- 从队列中删去顶点 1 并检查它的相邻顶点 0 和 2，发现它们都已经被标记了。
- 从队列中删去顶点 5 并检查它的相邻顶点 3 和 0，发现它们都已经被标记了。
- 从队列中删去顶点 3 并检查它的相邻顶点 5、4 和 2，发现它们都已经被标记了。
- 从队列中删去顶点 4 并检查它的相邻顶点 3 和 2，发现它们都已经被标记了。



539 图 4.1.18 使用广度优先搜索的轨迹，寻找所有起点为 0 的路径（另见彩插）

算法 4.2 使用广度优先搜索查找图中的路径

```
public class BreadthFirstPaths

{
   private boolean[] marked; // 到达该顶点的最短路径已知吗?
   private int[] edgeTo;     // 到达该顶点的已知路径上的最后一个顶点
   private final int s;      // 起点

   public BreadthFirstPaths(Graph G, int s)
   {
      marked = new boolean[G.V()];
      edgeTo = new int[G.V()];
      this.s = s;
      bfs(G, s);
   }
```

```
   private void bfs(Graph G, int s)
   {
      Queue<Integer> queue = new Queue<Integer>();
      marked[s] = true;           // 标记起点
      queue.enqueue(s);           // 将它加入队列
      while (!queue.isEmpty())
      {
         int v = queue.dequeue(); // 从队列中删去下一顶点
         for (int w : G.adj(v))
            if (!marked[w])       // 对于每个未被标记的相邻顶点
            {
               edgeTo[w] = v;     // 保存最短路径的最后一条边
               marked[w] = true;  // 标记它，因为最短路径已知
               queue.enqueue(w);  // 并将它添加到队列中
            }
      }
   }

   public boolean hasPathTo(int v)
   {  return marked[v];  }

   public Iterable<Integer> pathTo(int v)
   // 和深度优先搜索中的实现相同（请见算法4.1）
}
```

```
% java BreadthFirstPaths
tinyCG.txt 0
0 to 0: 0
0 to 1: 0-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-4
0 to 5: 0-5
```

这段 Graph 的用例使用了广度优先搜索，以找出图中从构造函数得到的起点 s 到与其他所有顶点的最短路径。bfs() 方法会标记所有与 s 连通的顶点，因此用例可以调用 hasPathTo() 来判定一个顶点与 s 是否连通并使用 pathTo() 得到一条从 s 到 v 的路径，确保没有其他从 s 到 v 的路径所含的边比这条路径更少。

540

对于这个例子来说，edgeTo[] 数组在第二步之后就已经完成了。和深度优先搜索一样，一旦所有的顶点都已经被标记，余下的计算工作就只是在检查连接到各个已被标记的顶点的边而已。

命题 B。对于从 s 可达的任意顶点 v，广度优先搜索都能找到一条从 s 到 v 的最短路径（没有其他从 s 到 v 的路径所含的边比这条路径更少）。

证明。由归纳易得队列总是包含零个或多个到起点的距离为 k 的顶点，之后是零个或多个到起点的距离为 k+1 的顶点，其中 k 为整数，起始值为 0。这意味着顶点是按照它们和 s 的距离的顺序加入或者离开队列的。从顶点 v 加入队列到它离开队列之前，不可能找出到 v 的更短的路径，而在 v 离开队列之后发现的所有能够到达 v 的路径都不可能短于 v 在树中的路径长度。

命题 B（续）。广度优先搜索所需的时间在最坏情况下和 $V+E$ 成正比。

证明。和命题 A 一样（请见 4.1.3.2 节），广度优先搜索标记所有与 s 连通的顶点所需的时间也与它们的度数之和成正比。如果图是连通的，这个和就是所有顶点的度数之和，也就是 $2E$。

注意，我们也可以用广度优先搜索来实现已经用深度优先搜索实现的 Search API，因为它检查所有与起点连通的顶点和边的方法只取决于查找的能力。

我们在本章开头说过，深度优先搜索和广度优先搜索是我们首先学习的几种通用的图搜索的算

法之一。在搜索中我们都会先将起点存入数据结构中，然后重复以下步骤直到数据结构被清空：

- 取其中的下一个顶点并标记它；
- 将 v 的所有相邻而又未被标记的顶点加入数据结构。

这两个算法的不同之处仅在于从数据结构中获取下一个顶点的规则（对于广度优先搜索来说是最早加入的顶点，对于深度优先搜索来说是最晚加入的顶点）。这种差异得到了处理图的两种完全不同的视角，尽管无论使用哪种规则，所有与起点连通的顶点和边都会被检查到。

图 4.1.19 和图 4.1.20 显示了深度优先搜索和广度优先搜索处理样图 mediumG.txt 的过程，它们清晰地展示了两种方法中搜索路径的不同。深度优先搜索不断深入图中并在栈中保存了所有分叉的顶点；广度优先搜索则像扇面一般扫描图，用一个队列保存访问过的最前端的顶点。深度优先搜索探索一幅图的方式是寻找离起点更远的顶点，只在碰到死胡同时才访问近处的顶点；广度优先搜索则会首先覆盖起点附近的顶点，只在临近的所有顶点都被访问了之后才向前进。深度优先搜索的路径通常较长而且曲折，广度优先搜索的路径则短而直接。根据应用的不同，所需要的性质也会有所不同（也许路径的性质也会变得无关紧要）。在 4.4 节中，我们会学习 Paths 的 API 的其他实现来寻找有特定属性的路径。

541 ~ 542

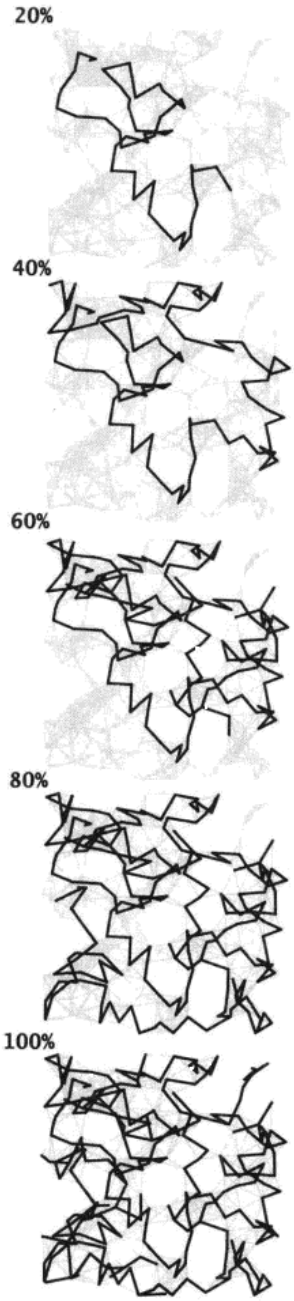


图 4.1.19 使用深度优先搜索查找路径（250 个顶点）

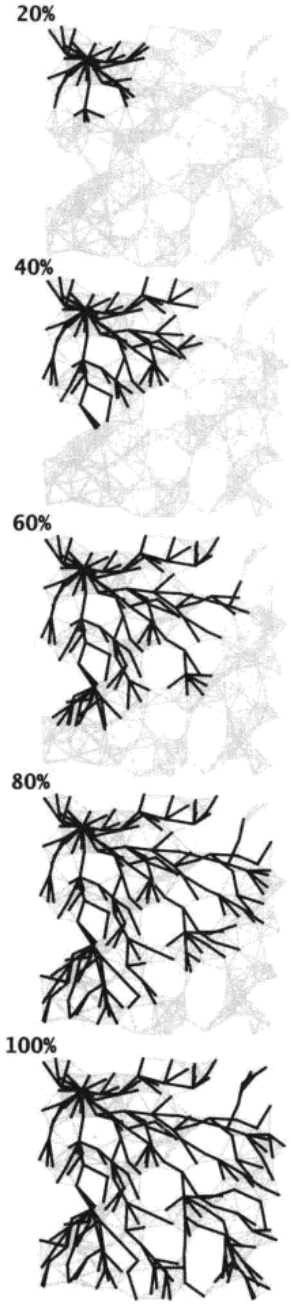


图 4.1.20 使用广度优先搜索查找最短路径（250 个顶点）

4.1.6 连通分量

深度优先搜索的下一个直接应用就是找出一幅图的所有连通分量。回忆 1.5 节中“与……连通”是一种等价关系，它能够将所有顶点切分为等价类（连通分量）。对于这个常见的任务，我们定义如下 API（请见表 4.1.6）。

表 4.1.6 连通分量的 API

public class	CC	
	CC(Graph G)	预处理构造函数
boolean	connected(int v, int w)	v 和 w 连通吗
int	count()	连通分量数
int	id(int v)	v 所在的连通分量的标识符（0 ~ count()-1）

用例可以用 id() 方法将连通分量用数组保存，如框注中的用例所示。它能够从标准输入中读取一幅图并打印其中的连通分量数，其后是每个子图中的所有顶点，每行一个子图。为了实现这些，它使用了一个 Bag 对象数组，然后用每个顶点所在的子图的标识符作为数组的索引，以将所有顶点加入相应的 Bag 对象中。当我们希望独立处理每个连通分量时这个用例就是一个模型。

4.1.6.1 实现

CC 的实现（请见算法 4.3）使用了 marked[] 数组来寻找一个顶点作为每个连通分量中深度优先搜索的起点。递归的深度优先搜索第一次调用的参数是顶点 0——它会标记所有与 0 连通的顶点。然后构造函数中的 for 循环会查找每个没有被标记的顶点并递归调用 dfs() 来标记和它相邻的所有顶点。另外，它还使用了一个以顶点作为索引的数组 id[]，将同一个连通分量中的顶点和连通分量的标识符关联起来（int 值）。这个数组使得 connected() 方法的实现变得十分简单，和 1.5 节中的 connected() 方法完全相同（只需检查标识符是否相同）。这里，标识符 0 会被赋予第一个连通分量中的所有顶点，1 会被赋予第二个连通分量中的所有顶点，依此类推。这样所有的标识符都会如 API 中指定的那样在 0 到 count()-1 之间。这个约定使得以子图作为索引的数组成为可能，如右侧框注用例所示。

```
public static void main(String[] args)
{
   Graph G = new Graph(new In(args[0]));
   CC cc = new CC(G);

   int M = cc.count();
   StdOut.println(M + " components");

   Bag<Integer>[] components;
   components = (Bag<Integer>[]) new Bag[M];
   for (int i = 0; i < M; i++)
      components[i] = new Bag<Integer>();
   for (int v = 0; v < G.V(); v++)
      components[cc.id(v)].add(v);
   for (int i = 0; i < M; i++)
   {
      for (int v: components[i])
         StdOut.print(v + " ");
      StdOut.println();
   }
}
```

查找连通分量API的测试用例

543

算法 4.3 使用深度优先搜索找出图中的所有连通分量

```
public class CC
{
   private boolean[] marked;
   private int[] id;
   private int count;
```

```
public CC(Graph G)
{
   marked = new boolean[G.V()];
   id = new int[G.V()];
   for (int s = 0; s < G.V(); s++)
      if (!marked[s])
      {
          dfs(G, s);
          count++;
      }
}

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
          dfs(G, w);
}

public boolean connected(int v, int w)
{  return id[v] == id[w];  }

public int id(int v)
{  return id[v];  }

public int count()
{  return count;  }
}
```

```
% more tinyG.txt
13 vertices, 13 edges
0: 6 2 1 5
1: 0
2: 0
3: 5 4
4: 5 6 3
5: 3 4 0
6: 0 4
7: 8
8: 7
9: 11 10 12
10: 9
11: 9 12
12: 11 9

% java CC tinyG.txt
3 components
6 5 4 3 2 1 0
8 7
12 11 10 9
```

这段 Graph 的用例使得它的用例可以独立处理一幅图中的每个连通分量。来自 DepthfirstSearch（请见 4.1.3.2 节）的代码均为灰色。这里的实现是基于一个由顶点索引的数组 id[]。如果 v 属于第 i 个连通分量，则 id[v] 的值为 i。构造函数会找出一个未被标记的顶点并调用递归函数 dfs() 来标记并区分出所有和它连通的顶点，如此重复直到所有的顶点都被标记并区分。connected()、count() 和 id() 方法的实现非常简单（另见图 4.1.21）。

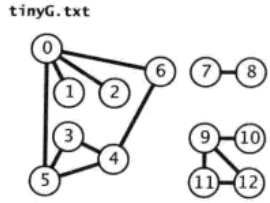
544

命题 C。深度优先搜索的预处理使用的时间和空间与 $V+E$ 成正比且可以在常数时间内处理关于图的连通性查询。

证明。由代码可以知道每个邻接表的元素都只会被检查一次，共有 $2E$ 个元素（每条边两个）。实例方法会检查或者返回一个或两个变量。

4.1.6.2 union-find 算法

CC 中基于深度优先搜索来解决图连通性问题的方法与第 1 章中的 union-find 算法相比孰优孰劣？理论上，深度优先搜索比 union-find 法快，因为它能保证所需的时间是常数而 union-find 算法不行；但在实际应用中，这点差异微不足道。union-find 算法其实更快，因为它不需要完整地构造并表示一幅图。更重要的是，union-find 算法是一种动态算法（我们在任何时候都能用接近常数的时间检查两个顶点是否连通，甚至是在添加一条边的时候），但深度优先搜索则必须要对图进行预处理。因此，我们在完成只需要判断连通性或是需要完成有大量连通性查询和插入操作混合等类似的任务时，更倾向使用 union-find 算法，而深度优先搜索则更适合实现图的抽象数据类型，因为它能更有效地利用已有的数据结构。



	count	marked[] 0 1 2 3 4 5 6 7 8 9 10 11 12	id[] 0 1 2 3 4 5 6 7 8 9 10 11 12
dfs(0)	0	**T**	**0**
dfs(6)	0	T **T**	0 **0**
检查 0			
dfs(4)	0	T **T** T	0 **0** 0
dfs(5)	0	T T **T** T	0 0 **0** 0
dfs(3)	0	T **T** T T	0 **0** 0 0
检查 5			
检查 4			
3 完成			
检查 4			
检查 0			
5 完成			
检查 6			
检查 3			
4 完成			
6 完成			
dfs(2)	0	T **T** T T T	0 **0** 0 0 0
检查 0			
2 完成			
dfs(1)	0	T **T** T T T T	0 **0** 0 0 0 0
检查 0			
1 完成			
检查 5			
0 完成			
dfs(7)	1	T T T T T T T **T**	0 0 0 0 0 0 0 **1**
dfs(8)	1	T T T T T T T T **T**	0 0 0 0 0 0 0 1 **1**
检查 7			
8 完成			
7 完成			
dfs(9)	2	T T T T T T T T **T**	0 0 0 0 0 0 0 1 1 **2**
dfs(11)	2	T T T T T T T T T T **T**	0 0 0 **9** 0 0 0 1 1 2 **2**
检查 9			
dfs(12)	2	T T T T T T T T T T **T**	0 0 0 0 0 0 0 1 1 2 **2**
检查 11			
检查 9			
12 完成			
11 完成			
dfs(10)	2	T T T T T T T T T T **T**	0 0 0 0 0 0 0 1 1 2 **2**
检查 9			
10 完成			
检查 12			
9 完成			

图 4.1.21 使用深度优先搜索的轨迹，寻找所有连通分量

我们已经用深度优先搜索解决了几个非常基础的问题。这种方法很简单，递归实现使我们能够进行复杂的运算并为一些图的处理问题给出简洁的解决方法。在表 4.1.7 中，我们为下面两个问题作出了解答。

检测环。给定的图是无环图吗？

双色问题。能够用两种颜色将图的所有顶点着色，使得任意一条边的两个端点的颜色都不相同吗？这个问题也等价于：这是一幅二分图吗？

深度优先搜索和已学习过的其他算法一样，它简洁的代码下隐藏着复杂的计算。因此，研究这些例子、在样图中跟踪算法的轨迹并加以扩展、用算法来解决环和着色的问题都是非常值得的（留作练习）。

545 ≀ 546

表 4.1.7 使用深度优先搜索处理图的其他示例

任 务	实 现
G 是无环图吗？（假设不存在自环或平行边）	`public class Cycle` `{` `  private boolean[] marked;` `  private boolean hasCycle;` `  public Cycle(Graph G)` `  {` `  marked = new boolean[G.V()];` `  for (int s = 0; s < G.V(); s++)` `  if (!marked[s])` `  dfs(G, s, s);` `  }` `  private void dfs(Graph G, int v, int u)` `  {` `  marked[v] = true;` `  for (int w : G.adj(v))` `  if (!marked[w])` `  dfs(G, w, v);` `  else if (w != u) hasCycle = true;` `  }` `  public boolean hasCycle()` `  { return hasCycle; }` `}`
G 是二分图吗？（双色问题）	`public class TwoColor` `{` `  private boolean[] marked;` `  private boolean[] color;` `  private boolean isTwoColorable = true;` `  public TwoColor(Graph G)` `  {` `  marked = new boolean[G.V()];` `  color = new boolean[G.V()];` `  for (int s = 0; s < G.V(); s++)` `  if (!marked[s])` `  dfs(G, s);` `  }` `  private void dfs(Graph G, int v)` `  {` `  marked[v] = true;` `  for (int w : G.adj(v))` `  if (!marked[w])` `  {` `  color[w] = !color[v];` `  dfs(G, w);` `  }` `  else if (color[w] == color[v]) isTwoColorable = false;` `  }` `  public boolean isBipartite()` `  { return isTwoColorable; }` `}`

547

4.1.7 符号图

在典型应用中，图都是通过文件或者网页定义的，使用的是字符串而非整数来表示和指代顶点。为了适应这样的应用，我们定义了拥有以下性质的输入格式：

- 顶点名为字符串；

- 用指定的分隔符来隔开顶点名（允许顶点名中含有空格）；
- 每一行都表示一组边的集合，每一条边都连接着这一行的第一个名称表示的顶点和其他名称所表示的顶点；
- 顶点总数 V 和边的总数 E 都是隐式定义的。

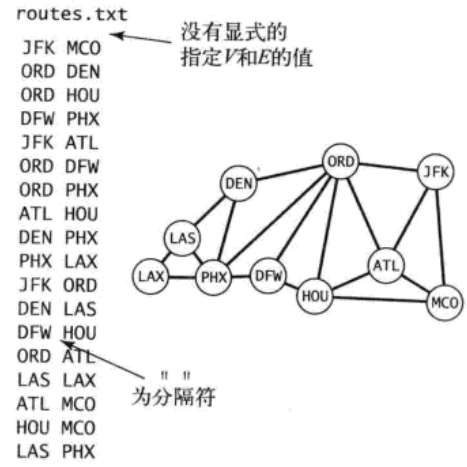


图 4.1.22 符号图示例（边的列表）

图 4.1.22 是一个简单的示例。Routes.txt 文件表示的是一个小型运输系统的模型，其中表示每个顶点的是美国机场的代码，连接它们的边则表示顶点之间的航线。文件只是一组边的列表。图 4.1.23 所示的是一个更庞大的例子，取自 movies.txt，即 3.5 节中介绍的*互联网电影数据库*。还记得吗？这个文件的每一行都列出了一个电影名以及出演该部电影的一系列演员。从图的角度来说，我们可以将它看作一幅图的定义，电影和演员都是顶点，而邻接表中的每一条边都将电影和它的表演者联系起来。注意，这是一幅二分图——电影顶点之间或者演员结点之间都没有边相连。

4.1.7.1 API

表 4.1.8 中，API 定义的 Graph 用例可以直接使用已有的图算法来处理这种文件定义的图。

表 4.1.8 用符号作为顶点名的图的 API

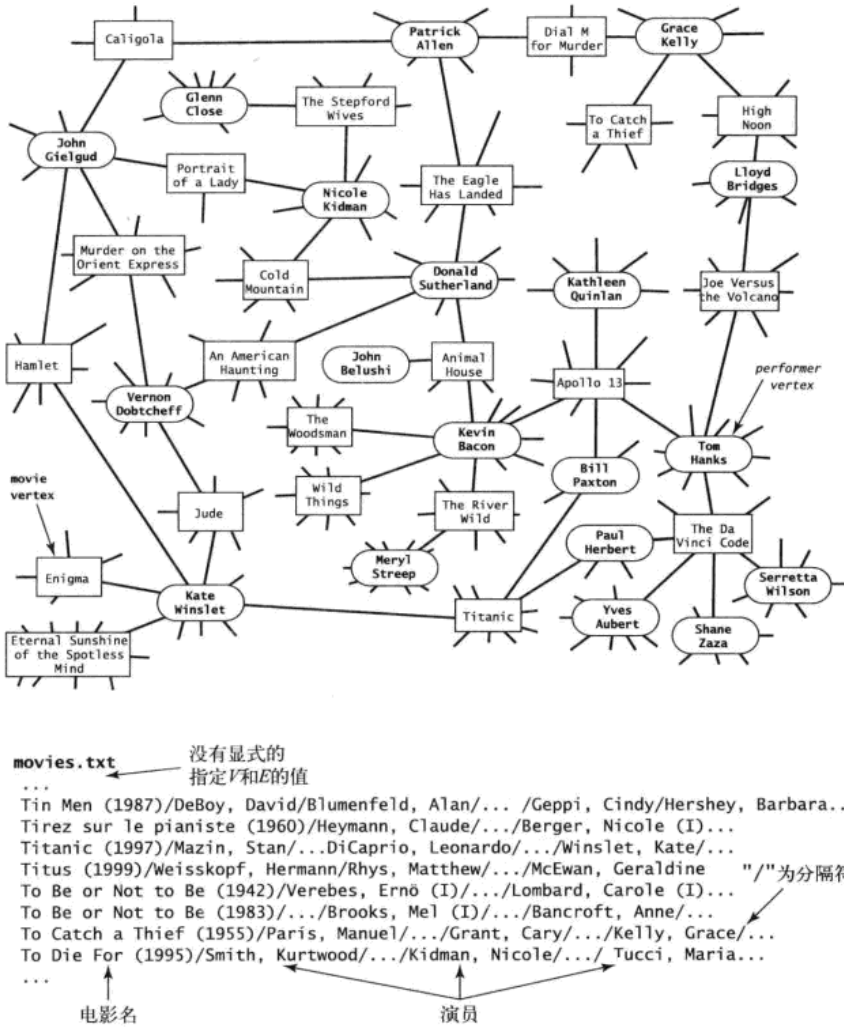
public class **SymbolGraph**		
	SymbolGraph(String filename, String delim)	根据 filename 指定的文件构造图，使用 delim 来分隔顶点名
boolean	contains(String key)	key 是一个顶点吗
int	index(String key)	Key 的索引
String	name(int v)	索引 v 的顶点名
Graph	G()	隐藏的 Graph 对象

548

这份 API 定义了一个构造函数来读取并构造图，用 name() 方法和 index() 方法将输入流中的顶点名和图算法使用的顶点索引对应起来。

4.1.7.2 测试用例

下一页框注所示的是符号图的测试用例，它用第一个命令行参数指定的文件（第二个命令行参数指定了分隔符）来构造一幅图并从标准输入接受查询。用户可以输入一个顶点名并得到该顶点的相邻结点的列表。这个用例提供的正好是 3.5 节中研究过的反向索引的功能。以 routes.txt 为例，你可以输入一个机场的代码来查找能从该机场直飞到达的城市，但这些信息并不是直接就能从文件中得到的。对于 movies.txt，你可以输入一个演员的名字来查看数据库中他所出演的影片列表。输入一部电影的名字来得到它的演员列表，这不过是在照搬文件中对应行数据，但输入演员的名字来得到影片的列表则相当于查找反向索引。尽管数据库的构造是为了将电影名连接到演员，二分图模型同时也意味着将演员连接到电影名。二分图的性质自动完成了反向索引。以后我们将会看到，这将成为处理更复杂的和图有关的问题的基础。




```
movies.txt          ← 没有显式的指定V和E的值
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara...
Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)...
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine    "/"为分隔符
To Be or Not to Be (1942)/Verebes, Ernö (I)/.../Lombard, Carole (I)...
To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...
To Catch a Thief (1955)/Paris, Manuel/.../Grant, Cary/.../Kelly, Grace/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../Tucci, Maria...
...
```

电影名　　演员

图 4.1.23　符号图示例（邻接表）

```
public static void main(String[] args)
{
   String filename = args[0];
   String delim = args[1];
   SymbolGraph sg = new SymbolGraph(filename, delim);

   Graph G = sg.G();

   while (StdIn.hasNextLine())
   {
      String source = StdIn.readLine();
      for (int w : G.adj(sg.index(source)))
         StdOut.println("   " + sg.name(w));
   }
}
```

符号图API的测试用例

```
% java SymbolGraph routes.txt " "
JFK
   ORD
   ATL
   MCO
LAX
   LAS
   PHX
```

```
% java SymbolGraph movies.txt "/"
Tin Men (1987)
   DeBoy, David
   Blumenfeld, Alan
   ...
   Geppi, Cindy
   Hershey, Barbara
   ...
Bacon, Kevin
   Mystic River (2003)
   Friday the 13th (1980)
   Flatliners (1990)
   Few Good Men, A (1992)
   ...
```

549 ~ 550

很显然，这种方法适用于我们遇到过的所有图算法：用例可以用 index() 将顶点名转化为索引并在图的处理算法中使用，然后将处理结果用 name() 转化为顶点名以方便在实际应用中使用。

4.1.7.3 实现

SymbolGraph 的完整实现请见下面的框注“符号图的数据类型”。它用到了以下 3 种数据结构，参见图 4.1.24。

- 一个符号表 st，键的类型为 String（顶点名），值的类型为 int（索引）；
- 一个数组 keys[]，用作反向索引，保存每个顶点索引所对应的顶点名；
- 一个 Graph 对象 G，它使用索引来引用图中顶点。

SymbolGraph 会遍历两遍数据来构造以上数据结构，这主要是因为构造 Graph 对象需要顶点总数 V。在典型的实际应用中，在定义图的文件中指明 V 和 E（见本节开头 Graph 的构造函数）可能会有些不便，而有了 SymbolGraph，我们就可以方便地在 routes.txt 或者 movies.txt 中添加或者删除条目而不用担心需要维护边或顶点的总数。

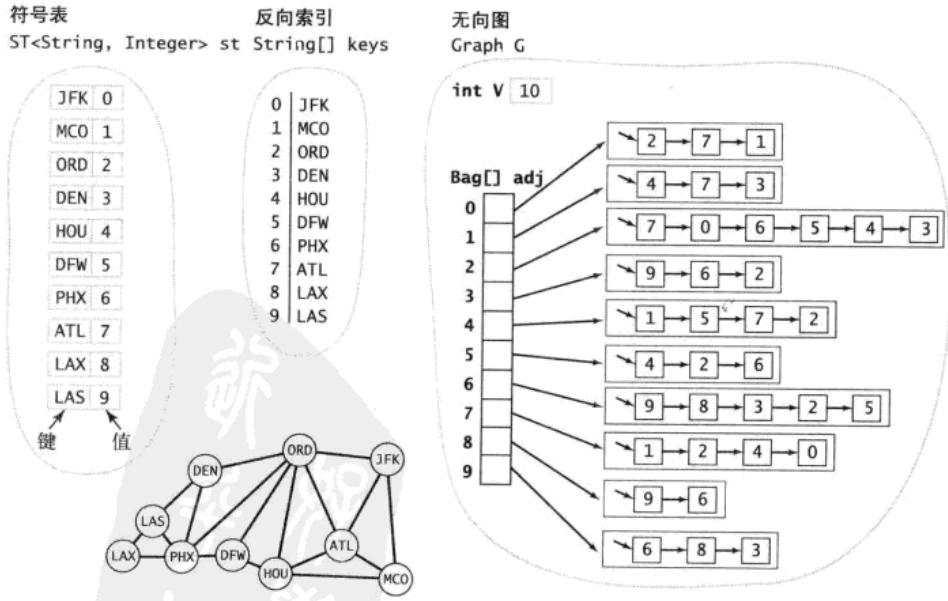


图 4.1.24 符号图中用到的数据结构

551

符号图的数据类型

```
public class SymbolGraph
{
   private ST<String, Integer> st;                  // 符号名 → 索引
   private String[] keys;                           // 索引 → 符号名
   private Graph G;                                 // 图

   public SymbolGraph(String stream, String sp)
   {
      st = new ST<String, Integer>();
      In in = new In(stream);                       // 第一遍
      while (in.hasNextLine())                      // 构造索引
      {
         String[] a = in.readLine().split(sp);      // 读取字符串

         for (int i = 0; i < a.length; i++)         // 为每个不同的字符串关联一个索引
            if (!st.contains(a[i]))

               st.put(a[i], st.size());
      }
      keys = new String[st.size()];                 // 用来获得顶点名的反向索引是一个数组

      for (String name : st.keys())
         keys[st.get(name)] = name;

      G = new Graph(st.size());
      in = new In(stream);                          // 第二遍
      while (in.hasNextLine())                      // 构造图
      {
         String[] a = in.readLine().split(sp);      // 将每一行的顶点和该行的其他顶点相连
         int v = st.get(a[0]);
         for (int i = 1; i < a.length; i++)
             G.addEdge(v, st.get(a[i]));
      }
   }

   public boolean contains(String s) {  return st.contains(s);  }
   public int index(String s)        {  return st.get(s);  }
   public String name(int v)         {  return keys[v];  }
   public Graph G()                  {  return G;  }
}
```

这个 Graph 实现允许用例用字符串代替数字索引来表示图中的顶点。它维护了实例变量 st（符号表用来映射顶点名和索引）、keys（数组用来映射索引和顶点名）和 G（使用索引表示顶点的图）。为了构造这些数据结构，代码会将图的定义处理两遍（定义的每一行都包含一个顶点及它的相邻顶点列表，用分隔符 sp 隔开）。

552

4.1.7.4 间隔的度数

图处理的一个经典问题就是，找到一个社交网络之中两个人间隔的度数。为了弄清楚概念，我们用一个最近很流行的名为 *Kevin Bacon* 的游戏来说明这个问题。这个游戏用到了刚才讨论的“电影 – 演员”图。Kevin Bacon 是一个活跃的演员，曾出演过许多电影。我们为图中的每个演员赋一个 Kevin Bacon 数：Bacon 本人为 0，所有和 Kevin Bacon 出演过同一部电影的人的值为 1，所有（除了 Kevin Bacon）和 Kevin Bacon 数为 1 的演员出演过同一部电影的其他演员的值为 2，依次类推。例如，Meryl Streep 的 Kevin Bacon 数为 1，因为她和 Kevin Bacon 一同出演过 *The River Wild*。

Nicole Kidman 的值为 2，因为她虽然没有和 Kevin Bacon 同台演出过任何电影，但她和 Tom Cruise 一起演过 *Days of Thunder*，而 Tom Cruise 和 Kevin Bacon 一起演过 *A Few Good Men*。给定一个演员的名字，游戏最简单的玩法就是找出一系列的电影和演员来回溯到 Kevin Bacon。例如，有些影迷可能知道 Tom Hanks 和 Lloyd Bridges 一起演过 *Joe Versus the Volcano*，而 Bridges 和 Grace Kelly 一起演过 *High Noon*，Kelly 又和 Patrick Allen 一起演过 *Dial M for Murder*，Allen 和 Donald Sutherland 一起演过 *The Eagle has Landed*，Sutherland 和 Kevin Bacon 一起出演了 *Animal House*。但知道这些也并不足以确定 Tom Hanks 的 Kevin Bacon 数。（他的值实际上应该是 1，因为他和 Kevin Bacon 在 *Apollo 13* 中合作过）。你可以看到 Kevin Bacon 数必须定义为最短电影链的长度，因此如果不用计算机，人们很难知道游戏中到底谁赢了。当然，如后面框注“间隔的度数”中 `SymbolGraph` 的用例 `DegreesOfSeparation` 所示，`BreadthFirstPaths` 才是我们所要的程序，它通过最短路径来找出 movies.txt 中任意演员的 Kevin Bacon 数。这个程序从命令行得到一个起点，从标准输入中接受查询并打印出一条从起点到被查询顶点的最短路径。因为 movies.txt 所构造的是一幅二分图，每条路径上都会交替出现电影和演员的顶点。打出的结果可以证明这样的路径是存在的（但并不能证明它是最短的——你需要向你的朋友证明命题 B 才行）。`DegreesOfSeparation` 也能够在非二分图中找到最短路径。例如，在 routes.txt 中，它能够用最少的边找到一种从一个机场到达另一个机场的方法。

```
% java DegreesOfSeparation movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
   Bacon, Kevin
   Few Good Men, A (1992)
   Cruise, Tom
   Days of Thunder (1990)
   Kidman, Nicole
Grant, Cary
   Bacon, Kevin
   Mystic River (2003)
   Willis, Susan
   Majestic, The (2001)
   Landau, Martin
   North by Northwest (1959)
   Grant, Cary
```

553

你可能会发现用 `DegreesOfSeparation` 来回答一些关于电影行业的问题很有趣。例如，你不但可以找到演员和演员之间的间隔，还可以找到电影和电影之间的间隔。更重要的是，间隔的概念在其他许多领域也被广泛研究。例如，数学家也会玩这个游戏，但他们的图是用一些论文的作者到 P.Erdös（20 世纪的一位多产的数学家）的距离来定义的。类似地，似乎新泽西州的每个人的 Bruce Springsteen 数都为 2，因为每个人都声称自己认识某个认识 Bruce 的人。要玩 Erdös 的游戏，你需要一个包含所有数学论文的数据库；要玩 Sprintsteen 的游戏还要困难一些。从更严肃的角度来说，间隔度数的理论在计算机网络的设计以及理解各个科学领域中的自然网络中都能起到重要的作用。

554

```
% java DegreesOfSeparation movies.txt "/" "Animal House (1978)"
Titanic (1997)
   Animal House (1978)
   Allen, Karen (I)
   Raiders of the Lost Ark (1981)
   Taylor, Rocky (I)
   Titanic (1997)
To Catch a Thief (1955)
   Animal House (1978)
   Vernon, John (I)
   Topaz (1969)
   Hitchcock, Alfred (I)
   To Catch a Thief (1955)
```

间隔的度数

```
public class DegreesOfSeparation
{
   public static void main(String[] args)
   {
      SymbolGraph sg = new SymbolGraph(args[0], args[1]);

      Graph G = sg.G();

      String source = args[2];
      if (!sg.contains(source))
      {  StdOut.println(source + "not in database."); return;  }

      int s = sg.index(source);
      BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

      while (!StdIn.isEmpty())
      {
         String sink = StdIn.readLine();
         if (sg.contains(sink))
         {
            int t = sg.index(sink);
            if (bfs.hasPathTo(t))
               for (int v : bfs.pathTo(t))
                  StdOut.println("   " + sg.name(v));
            else StdOut.println("Not connected");
         }
         else StdOut.println("Not in database.");
      }
   }
}
```

```
% java
DegreesOfSeparation
routes.txt " " JFK
LAS
   JFK
   ORD
   PHX
   LAS
DFW
   JFK
   ORD
   DFW
```

这段代码使用了 SymbolGraph 和 BreadthFirstPath 来查找图中的最短路径。对于 movies.txt，可以用它来玩 Kevin Bacon 游戏。

555

4.1.8 总结

在本节中，我们介绍了几个基本的概念，本章的其余部分会继续扩展并研究：

- 图的术语；
- 一种图的表示方法，能够处理大型而稀疏的图；
- 和图处理相关的类的设计模式，其实现算法通过在相关的类的构造函数中对图进行预处理、构造所需的数据结构来高效支持用例对图的查询；

- 深度优先搜索和广度优先搜索；
- 支持使用符号作为图的顶点名的类。

表 4.1.9 总结了我们已经学习过的所有图算法的实现。这些算法非常适合作为图处理的入门学习。随后学习更加复杂类型的图以及处理更加困难的问题时，我们还会用到这些代码的变种。在考虑了边的方向以及权重之后，同样的问题会变得困难得多，但同样的算法仍然奏效并将成为解决更加复杂问题的起点。

表 4.1.9 本节中得到解决的无向图处理问题

问 题	解决方法	参 阅
单点连通性	`DepthFirstSearch`	4.1.3.2 节
单点路径	`DepthFirstPaths`	算法 4.1
单点最短路径	`BreadthFirstPaths`	算法 4.2
连通性	`CC`	算法 4.3
检测环	`Cycle`	表 4.1.7
双色问题（图的二分性）	`TwoColor`	表 4.1.7

556

答疑

问 为什么不把所有的算法都实现在 Graph.java 中？

答 可以这么做，可以向基本的 Graph 抽象数据类型的定义中添加查询方法（以及它们需要的私有变量和方法等）。尽管这种方式可以用到一些我们所使用的数据结构的优点，它还是有一些严重的缺陷，因为图处理的成本比 1.3 节中遇到那些基本数据结构要高得多。这些缺点主要有：

- 在图处理中，需要实现的操作还有很多，我们无法在一份 API 中全部精确地定义它们；
- 简单任务的 API 和复杂任务所使用的 API 是相同的；
- 一个方法将可以访问另外一个方法专用的变量，这有悖我们需要遵守的封装原则。

这种情况并不罕见：这种 API 被称为宽接口（请见 1.2.5.2 节）。本章包含如此众多的图算法，将导致这种 API 变得非常宽。

问 SymbolGraph 真需要将图的定义遍历两遍吗？

答 不，你也可以将用时增加 lg*N* 并直接用 ST 而非 Bag 来实现 `adj()`。我们的另一本书 *An Introduction to Programming in Java: An Interdisciplinary Approach* 中含有使用这种方法的一个实现。

557

练习

4.1.1 一幅含有 *V* 个顶点且不含有平行边的图中至多含有多少条边？一幅含有 *V* 个顶点的连通图中至少含有多少条边？

4.1.2 按照正文中示意图的样式（请见图 4.1.9）画出 Graph 的构造函数在处理图 4.1.25 的 tinyGex2.txt 时构造的邻接表。

4.1.3 为 Graph 添加一个复制构造函数，它接受一幅图 G 然后创建并初始化这幅图的一个副本。G 的用例对它作出的任何改动都不应该影响到它的副本。

4.1.4 为 Graph 添加一个方法 `hasEdge()`，它接受两个整型参数 v 和 w。如果图含有边 v-w，方法返回 `true`，否则返回 `false`。

4.1.5 修改 Graph，不允许存在平行边和自环。

4.1.6 有一张含有四个顶点的图，其中的边为 0-1、1-2、2-3 和 3-0。给出一种邻接表数组，无论以任何顺序调用 addEdge() 来添加这些边都无法创建它。

4.1.7 为 Graph 编写一个测试用例，用命令行参数命名并从输入流中接受一幅图，然后用 toString() 方法将其打印出来。

4.1.8 按照正文中的要求，用 union-find 算法实现 4.1.2.3 中搜索的 API。

4.1.9 使用 dfs(0) 处理由 Graph 的构造函数从 tinyGex2.txt（请见练习 4.1.2）得到的图并按照 4.1.3.5 节的图 4.1.14 的样式给出详细的轨迹。同时，画出 edgeTo[] 所表示的树。

4.1.10 证明在任意一幅连通图中都存在一个顶点，删去它（以及和它相连的所有边）不会影响到图的连通性，编写一个深度优先搜索的方法找出这样一个顶点。提示：留心那些相邻顶点全部都被标记过的顶点。

4.1.11 使用算法 4.2 中的 bfs(G,0) 处理由 Graph 的构造函数从 tinyGex2.txt（请见练习 4.1.2）得到的图并画出 edgeTo[] 所表示的树。

4.1.12 如果 v 和 w 都不是根结点，能够由广度优先搜索得到的树中计算它们之间的距离吗？

4.1.13 为 BreadthFirstPaths 的 API 添加并实现一个方法 distTo()，返回从起点到给定的顶点的最短路径的长度，它所需的时间应该为常数。

4.1.14 如果用栈代替队列来实现广度优先搜索，我们还能得到最短路径吗？

4.1.15 修改 Graph 的输入流构造函数，允许从标准输入读入图的邻接表（方法类似于 SymbolGraph），如图 4.1.26 的 tinyGadj.txt 所示。在顶点和边的总数之后，每一行由一个顶点和它的所有相邻顶点组成。

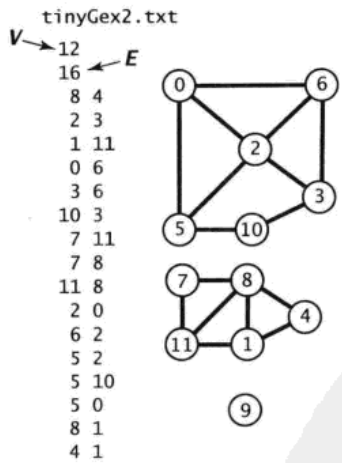


图 4.1.25

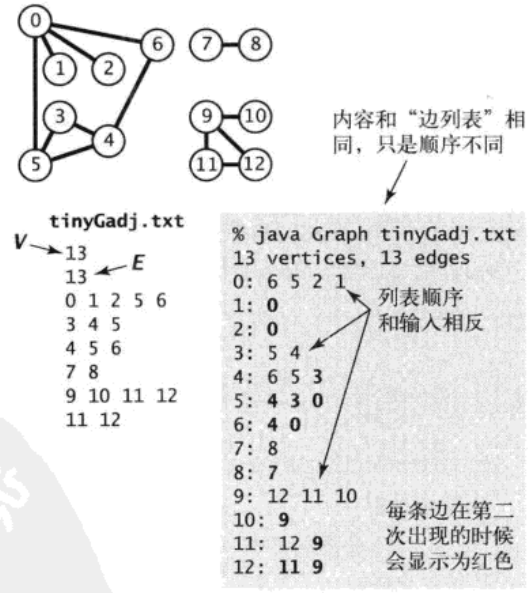


图 4.1.26

4.1.16 顶点 v 的离心率是它和离它最远的顶点的最短距离。图的直径即所有顶点的最大离心率，半径为所有顶点的最小离心率，中点为离心率和半径相等的顶点。实现以下 API，如表 4.1.10 所示。

表 4.1.10

public class	GraphProperties	
	GraphProperties(Graph G)	构造函数（如果 G 不是连通的，抛出异常）
int	eccentricity(int v)	v 的离心率
int	diameter()	G 的直径
int	radius()	G 的半径
int	center()	G 的某个中点

4.1.17 图的周长为图中最短环的长度。如果是无环图，则它的周长为无穷大。为 GraphProper-ties 添加一个方法 girth()，返回图的周长。提示：在每个顶点都进行广度优先搜索。含有 s 的最小环为 s 到某个顶点 v 的最短距离加上 v 到 s 的最短距离。 558 ≀ 559

4.1.18 使用 CC 找出由 Graph 的输入流构造函数从 tinyGex2.txt（请见练习 4.1.2）得到的图中的所有连通分量并按照图 4.1.21 的样式给出详细的轨迹。

4.1.19 使用 Cycle 在由 Graph 的输入流构造函数从 tinyGex2.txt（请见练习 4.1.2）得到的图中找到的一个环并按照本节示意图的样式给出详细的轨迹。在最坏情况下，Cycle 构造函数的运行时间的增长数量级是多少？

4.1.20 使用 TwoColor 给出由 Graph 的构造函数从 tinyGex2.txt（请见练习 4.1.2）得到的图的一个着色方案并按照本节示意图的样式给出详细的轨迹。在最坏情况下，TwoColor 构造函数的运行时间的增长数量级是多少？

4.1.21 用 SymbolGraph 和 movie.txt 找到今年获得奥斯卡奖提名的演员的 Kevin Bacon 数。

4.1.22 编写一段程序 BaconHistogram，打印一幅 Kevin Bacon 数的柱状图，显示 movies.txt 中 Kevin Bacon 数为 0、1、2、3……的演员分别有多少。将值为无穷大的人归为一类（不与 Kevin Bacon 连通）。

4.1.23 计算由 movies.txt 得到的图的连通分量的数量和包含的顶点数小于 10 的连通分量的数量。计算最大的连通分量的离心率、直径、半径和中点。Kevin Bacon 在最大的连通分量之中吗？

4.1.24 修改 DegreesOfSeparation，从命令行接受一个整型参数 y，忽略上映年数超过 y 的电影。

4.1.25 编写一个类似于 DegreesOfSeparation 的 SymbolGraph 用例，使用深度优先搜索代替广度优先搜索来查找两个演员之间的路径，输出类似如右侧框注所示的数据格式。

```
% java DegreesOfSeparationDFS movies.txt
Source: Bacon, Kevin
Query:  Kidman, Nicole
   Bacon, Kevin
   Mystic River (2003)
   O' Hara, Jenny
   Matchstick Men (2003)
   Grant, Beth
  ... [123 movies] (!)
   Law, Jude
   Sky Captain... (2004)
   Jolie, Angelina
   Playing by Heart (1998)
   Anderson, Gillian (I)
   Cock and Bull Story, A (2005)
   Henderson, Shirley (I)
   24 Hour Party People (2002)
   Eccleston, Christopher
   Gone in Sixty Seconds (2000)
   Balahoutis, Alexandra
   Days of Thunder (1990)
   Kidman, Nicole
```

4.1.26 使用 1.4 节中的内存使用模型评估用 Graph 表示一幅含有 V 个顶点和 E 条边的图所需的内存。

4.1.27 如果重命名一幅图中的顶点就能够使之变得和另一幅图完全相同，这两幅图就是同构的。画出含有 2、3、4、5 个顶点的所有非同构的图。

4.1.28 修改 Cycle，允许图含有自环和平行边。 560 ≀ 561

提高题

4.1.29 欧拉环和汉密尔顿环。考虑以下4组边定义的图：

```
0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7
```

哪几幅图含有欧拉环（恰好包含了所有的边且没有重复的环）？哪几幅图含有汉密尔顿环（恰好包含了所有的顶点且没有重复的环）？

4.1.30 图的枚举。含有 V 个顶点和 E 条边（不含平行边）的不同的无向图共有多少种？

4.1.31 检测平行边。设计一个线性时间的算法来统计图中的平行边的总数。

4.1.32 奇环。证明一幅图能够用两种颜色着色（二分图）当且仅当它不含有长度为奇数的环。

4.1.33 符号图。实现一个 SymbolGraph（不一定必须使用 Graph），只需要遍历一遍图的定义数据。由于需要查找符号表，实现中图的各种操作时耗可能会变为原来的 $\log V$ 倍。

4.1.34 双向连通性。如果任意一对顶点都能由两条不同（没有重叠的边或顶点）的路径连通则图就是双向连通的。在一幅连通图中，如果一个顶点被删掉后图不再连通，该顶点就被称为关节点。证明没有关节点的图是双向连通的。提示：给定任意一对顶点 s 和 t 和一条连接两点的路径，由于路径上没有任何顶点为关节点，构造另一条不同的路径连接 s 和 t。

4.1.35 边的连通性。在一幅连通图中，如果一条边被删除后图会被分为两个独立的连通分量，这条边就被称为桥。没有桥的图称为边连通图。开发一种基于深度优先搜索算法的数据类型，判断一个图是否是边连通图。

4.1.36 欧拉图。为平面上的图设计并实现一份叫做 EulideanGraph 的 API，其中图所有顶点均有坐标。实现一个 show() 方法，用 StdDraw 将图绘出。

562 ~ 563

4.1.37 图像处理。在一幅图像中将所有相邻的、颜色相同的点相连就可以得到一幅图，为这种隐式定义的图实现填充（flood fill）操作。

实验题

4.1.38 随机图。编写一个程序 ErdosRenyiGraph，从命令行接受整数 V 和 E，随机生成 E 对 0 到 V–1 之间的整数来构造一幅图。注意：生成器可能会产生自环和平行边。

4.1.39 随机简单图。编写一个程序 RandomSimpleGraph，从命令行接受整数 V 和 E，用均等的几率生成含有 V 个顶点和 E 条边的所有可能的简单图。

4.1.40 随机稀疏图。编写一个程序 RandomSparseGraph，根据精心选择的一组 V 和 E 的值生成随机的稀疏图，以便用它对由 Erdös-Renyi 模型得到的图进行有意义的经验性测试。

4.1.41 随机欧拉图。编写一个 EulideanGraph 的用例（请见练习 4.1.36）RandomEulideanGraph，用随机在平面上生成 V 个点的方式生成随机图，然后将每个点和在以该点为中心半径为 d 的圆内的其他点相连。注意：如果 d 大于阈值 $\sqrt{\lg V / \pi V}$，那么得到的图几乎必然是连通的，否则得到的图几乎必然是不连通的。

4.1.42 随机网格图。编写一个 EulideanGraph 的用例 RandomGridGraph，将 $\sqrt{V}$ 乘 $\sqrt{V}$ 的网格中的所有顶点和它们的相邻顶点相连（参考练习 1.5.18）。修改代码为图额外添加 R 条随机的边。对于较大的 R，缩小网格使得总边数保持在 V 个左右。添加一个选项，使得出现一条从顶点 s 到顶

点 v 的边的概率与 s 到 t 的欧拉距离成反比。

4.1.43 *真实世界中的图*。从网上找出一幅巨型加权图——可以是一张标记了距离的地图，或者是标明了费用的电话连接，或是航班价目表。编写一段程序 RandomRealGraph，从这些顶点构成的子图中随机选取 V 个顶点，然后再从这些顶点构成的子图中随机选取 E 条边来构造一幅图。

4.1.44 *随机区间图*。考虑数轴上的 V 个区间的集合。这样的一个集合定义了一幅区间图，图中的每个顶点都对应一个区间，而边则对应两个区间的交集（大小不限）。编写一段程序，随机生成大小均为 d 的 V 个区间，然后构造相应的区间图。*提示*：使用二分查找树。 564

4.1.45 *随机运输图*。定义运输系统的一种方法是定义一个顶点链的集合，每条顶点链都表示一条连接了多个顶点的路径。例如，链 0-9-3-2 定义了边 0-9、9-3 和 3-2。编写一个 EulideanGraph 的用例 RandomTransportation，从一个输入文件中构造一幅图，文件的每行均为一条链，使用符号名。编辑一份合适的输入使得程序能够从中构造一幅和巴黎地铁系统相对应的图。

测试所有的算法并研究所有图模型的所有参数是不现实的。请为下面的每一道题都编写一段程序来处理从输入得到的任意图。这段程序可以调用上面的任意生成器并对相应的图模型进行实验。可以根据上次实验的结果自己作出判断来选择不同实验。陈述结果以及由此得出的任何结论。

4.1.46 *深度优先搜索中的路径长度*。对于各种图的模型，运行实验并根据经验判断 DepthFirstPaths 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。

4.1.47 *广度优先搜索中的路径长度*。对于各种图的模型，运行实验并根据经验判断 BreadthFirstPaths 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。

4.1.48 *连通分量*。运行实验随机生成大量的图并画出柱状图，根据经验判断各种类型的随机图中连通分量的数量的分布情况。

4.1.49 *双色问题*。大多数的图都无法用两种颜色着色，深度优先搜索能够很快发现这一点。对于各种图模型，使用经验性的测试来研究 TwoColor 检查的边的数量。 565

4.2 有向图

在有向图中，边是单向的：每条边所连接的两个顶点都是一个有序对，它们的邻接性是单向的（表4.2.1）。许多应用（比如表示网络、任务调度条件或是电话的图）都是天然的有向图。为实现添加这种单向性的限制很容易也很自然，看起来没什么坏处。但实际上这种组合性的结构对算法有深刻的影响，使得有向图和无向图的处理大有不同。本节中，我们会学习搜索和处理有向图的一些经典算法。

表4.2.1 实际生活中的典型有向图

应　用	顶　点	边
食物链	物种	捕食关系
互联网连接	网页	超链接
程序	模块	外部引用
手机	电话	呼叫
学术研究	论文	引用
金融	股票	交易
网络	计算机	网络连接

4.2.1 术语

虽然我们为有向图的定义和无向图几乎相同（将使用的部分算法和代码也是），但仍然需要在这里重复一遍。为了说明边的方向性而产生的细小文字差异所代表的结构特性正是本节的重点。

定义。一幅有方向性的图（或有向图）是由一组顶点和一组有方向的边组成的，每条有方向的边都连接着有序的一对顶点。

我们称一条有向边由第一个顶点指出并指向第二个顶点。在一幅有向图中，一个顶点的出度为由该顶点指出的边的总数；一个顶点的入度为指向该顶点的边的总数（请见图4.2.1）。当上下文的意义明确时，我们在提到有向图中的边时会省略有向二字。一条有向边的第一个顶点称为它的头，第二个顶点则被称为它的尾。将有向边画为由头指向尾的一个箭头。用 v→w 来表示有向图中一条由 v 指向 w 的边。和无向图一样，本节的代码也能处理自环和平行边，但它们不会出现在例子中，
566 在正文中一般也不会提到它们。除了特殊的图，一幅有向图中的两个顶点的关系可能有4种：没有边相连；存在从 v 到 w 的边 v→w；存在从 w 到 v 的边 w→v；既存在 v→w 也存在 w→v，即双向的连接。

定义。在一幅有向图中，有向路径由一系列顶点组成，对于其中的每个顶点都存在一条有向边从它指向序列中的下一个顶点。有向环为一条至少含有一条边且起点和终点相同的有向路径。简单有向环是一条（除了起点和终点必须相同之外）不含有重复的顶点和边的环。路径或者环的长度即为其中所包含的边数。

和无环图一样，我们假设有向路径都是简单的，除非我们明确指出了某个重复了的顶点（像有向环的定义中那样）或是指明是一般性的有向图。当存在从 v 到 w 的路径时，称顶点 w 能够由顶点

v 达到。我们约定，每个顶点都能够达到它自己。除了这种情况之外，在有向图中由 v 能够到达 w 并不意味着由 w 也能到达 v。这个不同虽然很明显但非常重要，后面将会看到这一点。

要理解本节中的算法，你就必须要理解有向图中的可达性和无向图中的连通性的区别。理解这种区别可能比你想象得更困难。例如，尽管你可能一眼就能看出一小幅无向图中的两个顶点之间是否连通，但是在一小幅有向图中快速找出一条有向路径就不那么容易了，比如图 4.2.2 所示的例子。处理有向图就如同在一座只有单行道的城市中穿梭，而且这些单行道的方向是杂乱无章的。在这种情况下，想从一处到达另一处会是一件很麻烦的事。但与直觉相反，我们用来表示有向图的标准数据结构甚至比无向图的表示更加简单！

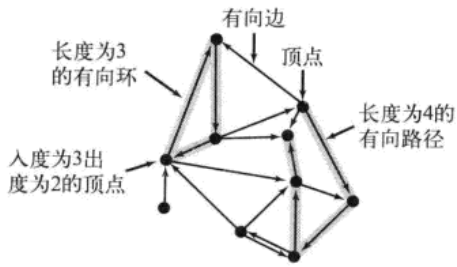


图 4.2.1 有向图详解

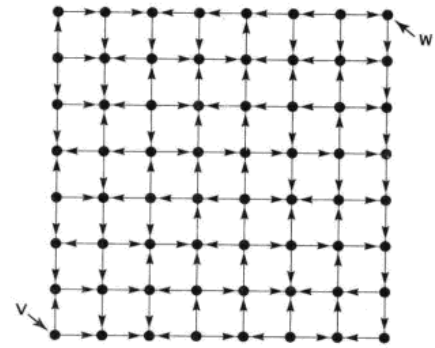


图 4.2.2 在这幅有向图中，从 v 能够到达 w 吗

567

4.2.2 有向图的数据类型

以下这份 API 以及下一页中的 Digraph 类和 Graph 类本质上是相同的（请见 4.1.2.2 节框注“Graph 数据类型”）。

表 4.2.2 有向图的 API

public class	**Digraph**	
	Digraph(int V)	创建一幅含有 *V* 个顶点但没有边的有向图
	Digraph(In in)	从输入流 in 中读取一幅有向图
int	V()	顶点总数
int	E()	边的总数
void	addEdge(int v, int w)	向有向图中添加一条边 v→w
Iterable<Integer>	adj(int v)	由 v 指出的边所连接的所有顶点
Digraph	reverse()	该图的反向图
String	toString()	对象的字符串表示

4.2.2.1 有向图的表示

我们使用邻接表来表示有向图，其中边 v→w 表示为顶点 v 所对应的邻接链表中包含一个 w 顶点。这种表示方法和无向图几乎相同而且更明晰，因为每条边都只会出现一次，如后面框注“有向图（diagraph）的数据类型”所示。

4.2.2.2 输入格式

由输入流读取有向图的构造函数的代码与 Graph 类中相应构造函数的代码完全相同——因为两者的输入格式是一样的，但所有的边都是有向边。在边列表的格式中，一对顶点 v 和 w 表示边 v→w。

4.2.2.3 有向图取反

Digraph 的 API 中还添加了一个方法 reverse()。它返回该有向图的一个副本，但将其中所有边的方向反转。在处理有向图时这个方法有时很有用，因为这样用例就可以找出“指向”每个顶点的所有边，而 adj() 给出的是由每个顶点指出的边所连接的所有顶点。

4.2.2.4 顶点的符号名

在有向图中，允许用例使用符号作为顶点名也更加简单。要实现与 SymbolGraph 类似的 SymbolDigraph 类，只需要将其中的 Graph 字样都替换成 Digraph 即可。

花一点时间对比一下后面框注中的代码和示意图与 4.1.2.1 节及 4.1.2.2 节的框注“Graph 数据类型”中无向图的代码是非常有价值的。在用邻接表表示无向图时，如果 v 在 w 的链表中，那么 w 必
568 然也在 v 的链表中。但在有向图中这种对称性是不存在的。这个区别在有向图的处理中影响深远。

Digraph 数据类型

```
public class Digraph
{
   private final int V;
   private int E;
   private Bag<Integer>[] adj;

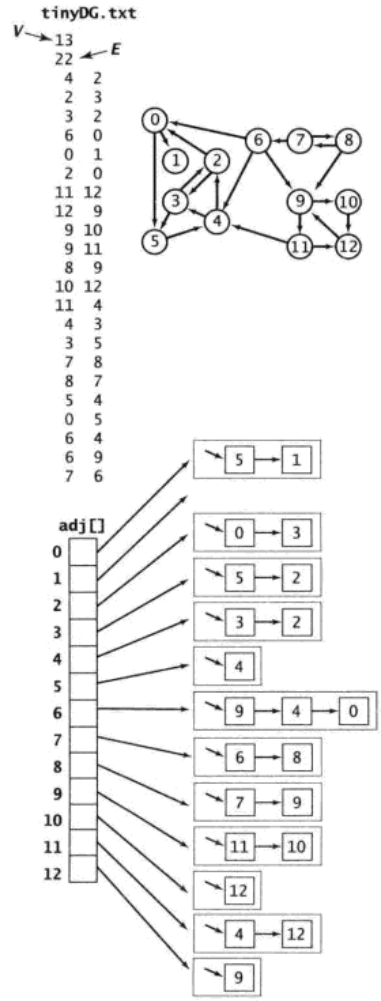
   public Digraph(int V)
   {
      this.V = V;
      this.E = 0;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public int V()  {  return V;  }
   public int E()  {  return E;  }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      E++;
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }

   public Digraph reverse()
   {
      Digraph R = new Digraph(V);
      for (int v = 0; v < V; v++)
         for (int w : adj(v))
            R.addEdge(w, v);
      return R;
   }
}
```



有向图的输入格式和邻接表的表示

Digraph 数据类型与 Graph 数据类型（请见 4.1.2.2 框注“Graph 数据类型”）基本相同，区别是 addEdge() 只调用了一次 add()，而且它还有一个 reverse() 方法来返回图的反向图。因为两者的代码非常相似，所以省略了 toString() 方法（请见
569 表 4.1.2）和从输入流中读取图的构造函数。

4.2.3 有向图中的可达性

在无向图中介绍的第一个算法就是 4.1.3.2 节中的 DepthFirstSearch，它解决了单点连通性的问题，使得用例可以判定其他顶点和给定的起点是否连通。使用完全相同的代码，将其中的 Graph 替换为 Digraph，也可以解决一个有向图中的类似问题。

单点可达性给定一幅有向图和一个起点 s，回答“是否存在一条从 s 到达给定顶点 v 的有向路径？”等类似问题。

算法 4.4 中的 DirectedDFS 类将 DepthFirstSearch 稍加润色并实现了以下 API。

表 4.2.3 有向图的可达性 API

public class	DirectedDFS	
	DirectedDFS(Digraph G, int s)	在 G 中找到从 s 可达的所有顶点
	DirectedDFS(Digraph G, Iterable<Integer> sources)	在 G 中找到从 sources 中的所有顶点可达的所有顶点
boolean	marked(int v)	v 是可达的吗

在添加了一个接受多个顶点的构造函数之后，这份 API 使得用例能够解决一个更加一般的问题。

多点可达性给定一幅有向图和顶点的集合，回答“是否存在一条从集合中的任意顶点到达给定顶点 v 的有向路径？”等类似问题。

我们在 5.4 节中解决经典的字符串处理问题时会再次遇到这个问题。

DirectedDFS 使用了解决图处理的标准范例和标准的深度优先搜索来解决这些问题。它对每个起点调用递归方法 dfs()，以标记遇到的任意顶点。

命题 D。在有向图中，深度优先搜索标记由一个集合的顶点可达的所有顶点所需的时间与被标记的所有顶点的出度之和成正比。

证明。同 4.1.3.2 节的命题 A。

图 4.2.3 显示了这个算法在处理有向样图时的操作轨迹。这份轨迹比相应的无向图算法的轨迹稍稍简单些，因为深度优先搜索本质上是一种适用于处理有向图的算法，每条边都只会被表示一次。研究这些轨迹有助于巩固你对有向图中深度优先搜索的理解。 570

算法 4.4 有向图的可达性

```
public class DirectedDFS
{
   private boolean[] marked;

   public DirectedDFS(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   public DirectedDFS(Digraph G, Iterable<Integer> sources)
   {
```

```
        marked = new boolean[G.V()];
        for (int s : sources)
           if (!marked[s]) dfs(G, s);

     }

     private void dfs(Digraph G, int v)
     {
        marked[v] = true;
        for (int w : G.adj(v))
           if (!marked[w]) dfs(G, w);
     }

     public boolean marked(int v)
     {  return marked[v];  }

     public static void main(String[] args)
     {
        Digraph G = new Digraph(new In(args[0]));

        Bag<Integer> sources = new Bag<Integer>();
        for (int i = 1; i < args.length; i++)
            sources.add(Integer.parseInt(args[i]));

        DirectedDFS reachable = new DirectedDFS(G, sources);

        for (int v = 0; v < G.V(); v++)
           if (reachable.marked(v)) StdOut.print(v + " ");
        StdOut.println();
     }
  }
```

```
% java DirectedDFS tinyDG.txt 1
1

% java DirectedDFS tinyDG.txt 2
0 1 2 3 4 5

% java DirectedDFS tinyDG.txt 1 2 6
0 1 2 3 4 5 6 9 10 11 12
```

571 这份深度优先搜索的实现使得用例能够判断从给定的一个或者一组顶点能到达哪些其他顶点。

4.2.3.1 标记 – 清除的垃圾收集

多点可达性的一个重要的实际应用是在典型的内存管理系统中，包括许多 Java 的实现。在一幅有向图中，一个顶点表示一个对象，一条边则表示一个对象对另一个对象的引用。这个模型很好地表现了运行中的 Java 程序的内存使用状况。在程序执行的任何时候都有某些对象是可以被直接访问的，而不能通过这些对象访问到的所有对象都应该被回收以便释放内存（请见图 4.2.4）。标记 – 清除的垃圾回收策略会为每个对象保留一个位做垃圾收集之用。它会周期性地运行一个类似于 DirectedDFS 的有向图可达性算法来标记所有可以被访问到的对象，然后清理所有对象，回收没有被标记的对象，以腾出内存供新的对象使用。

4.2.3.2 有向图的寻路

DepthFirstPaths（4.1.4.1 节算法 4.1）和 BreadthFirstPaths（4.1.5 节算法 4.2）也都是有向图处理中的重要算法。和刚才一样，同样的 API 和代码（仅将 Graph 替换为 Digraph）也能够高效地解决以下问题。

单点有向路径给定一幅有向图和一个起点 s，回答“从 s 到给定目的顶点 v 是否存在一条有向路径？如果有，找出这条路径。”等类似问题。

单点最短有向路径给定一幅有向图和一个起点 s，回答“从 s 到给定目的顶点 v 是否存在一条有向路径？如果有，找出其中最短的那条（所含边数最少）。”等类似问题。

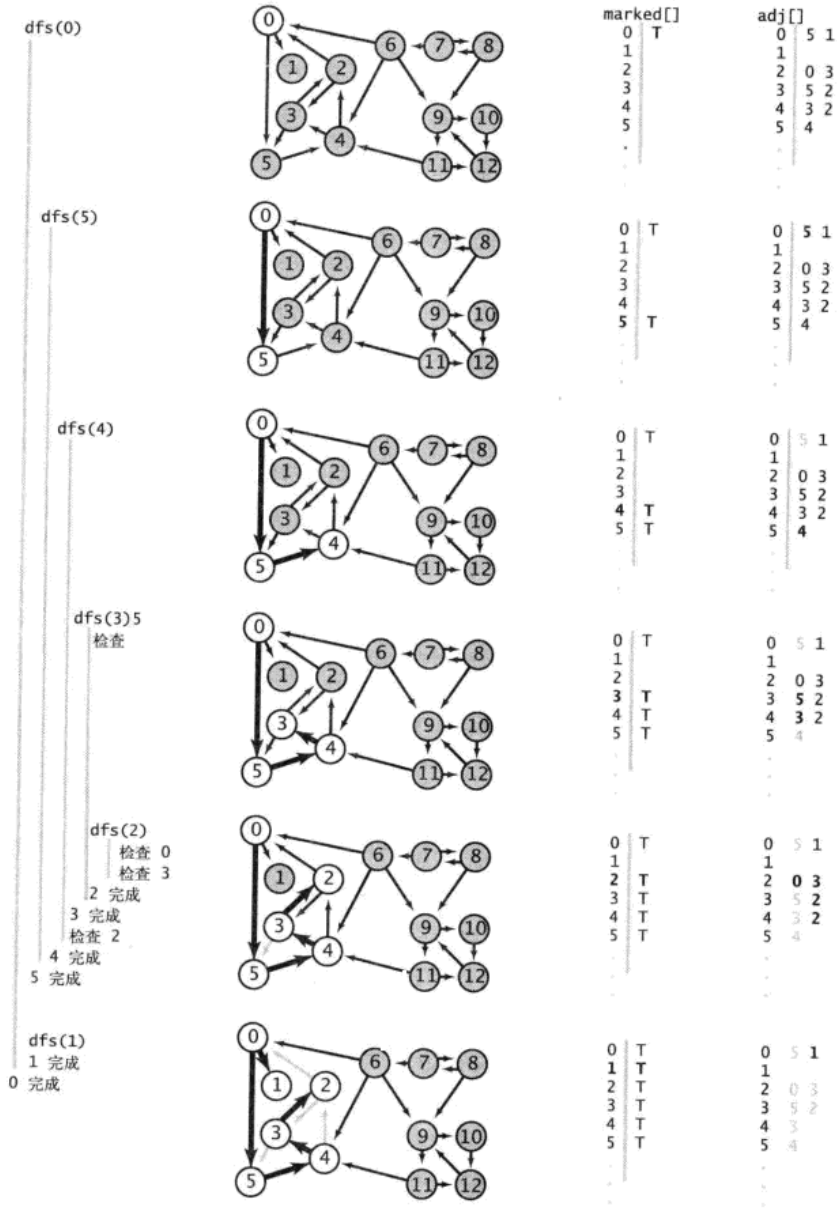


图 4.2.3 使用深度优先搜索在一幅有向图中寻找能够从顶点 0 到达的所有顶点的轨迹

在本书的网站上以及本节最后的练习中，我们将以上问题的答案分别命名为 DepthFirstDirectedPaths 和 BreadthFirstDirectedPaths。

572
~
573

4.2.4 环和有向无环图

在和有向图相关的实际应用中，有向环特别的重要。没有计算机的帮助，在一幅普通的有向图中找出有向环可能会很困难。从原则上来说，一幅有向图可能含有大量的环；在实际应用中，我们一般只会重点关注其中一小部分，或者只想知道它们是否存在（请见图 4.2.5）。

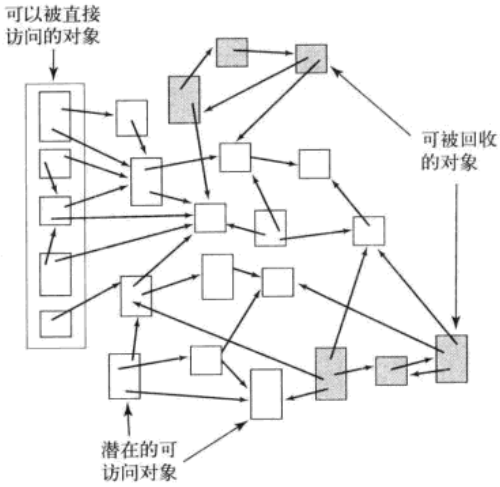


图 4.2.4 垃圾回收示意图

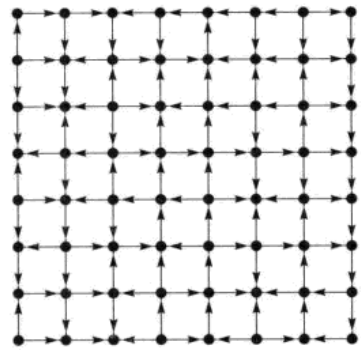


图 4.2.5 这幅有向图含有有向环吗

为了在有向图处理中研究有向环的作用更加有趣，我们来看看下面这个有向图模型的原型应用。

4.2.4.1 调度问题

一种应用广泛的模型是给定一组任务并安排它们的执行顺序，限制条件是这些任务的执行方法和起始时间。限制条件还可能包括任务的时耗以及消耗的其他资源。最重要的一种限制条件叫做*优先级限制*，它指明了哪些任务必须在哪些任务之前完成。不同类型的限制条件会产生不同类型不同难度的调度问题。研究者已经解决了上千种不同的此类问题，而且还在为其中许多寻找更好的算法。以一个正在安排课程的大学生为例，有些课程是其他课程的先导课程，如图 4.2.6 所示。

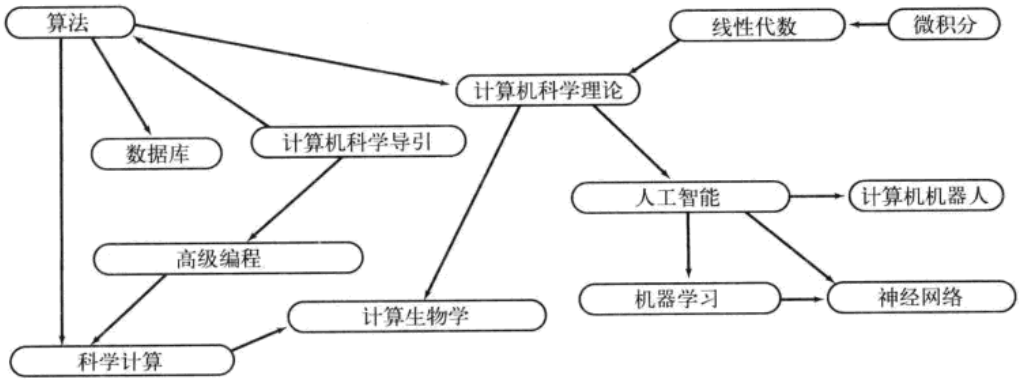


图 4.2.6 有优先级限制的调度问题

如果再假设该学生一次只能修一门课，实际上就遇到了下面这个问题。

优先级限制下的调度问题。给定一组需要完成的任务，以及一组关于任务完成的先后次序的优先级限制。在满足限制条件的前提下应该如何安排并完成所有任务？

对于任意一个这样的问题，我们都可以马上画出一张有向图，其中顶点对应任务，有向边对应优先级顺序。为了简化问题，我们以使用整数为顶点编号的标准模型来表示这个示例，如图4.2.7所示。

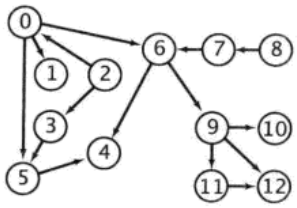


图 4.2.7 标准有向图模型

在有向图中，优先级限制下的调度问题等价于下面这个基本的问题。

拓扑排序。给定一幅有向图，将所有的顶点排序，使得所有的有向边均从排在前面的元素指向排在后面的元素（或者说明无法做到这一点）。

图 4.2.8 为示例的拓扑排序。所有的边都是向下的，所以它清晰地表示了这幅有向图模型所代表的有优先级限制的调度问题的一个解决方法：按照这个顺序，该同学可以在满足先导课程限制的条件下修完所有课程。这个应用是很典型的——表 4.2.4 列举了其他一些有代表性的应用。

表 4.2.4 拓扑排序的典型应用

应　用	顶　点	边
任务调度	任务	优先级限制
课程安排	课程	先导课程限制
继承	Java 类	`extends` 关系
电子表格	单元格（cell）	公式
符号链接	文件名	链接

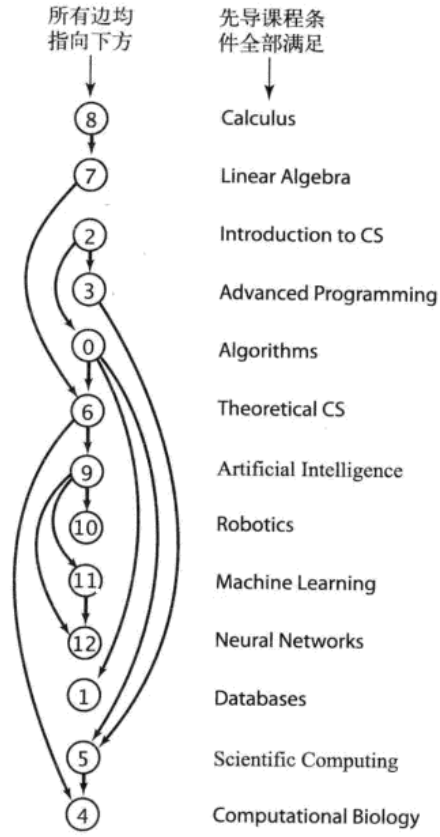


图 4.2.8 拓扑排序

4.2.4.2 有向图中的环

如果任务 x 必须在任务 y 之前完成，而任务 y 必须在任务 z 之前完成，但任务 z 又必须在任务 x 之前完成，那肯定是有人搞错了，因为这三个限制条件是不可能被同时满足的。一般来说，如果一个有优先级限制的问题中存在有向环，那么这个问题肯定是无解的。要检查这种错误，需要解决下面这个问题。

有向环检测。给定的有向图中包含有向环吗？如果有，按照路径的方向从某个顶点并返回自己来找到环上的所有顶点。

一幅有向图中含有的环的数量可能是图的大小的指数级别（请见练习 4.2.11），因此我们只需要找出一个环即可，而不是所有环。在任务调度和其他许多实际问题中不允许出现有向环，因此不含有环的有向图就变得很特殊。

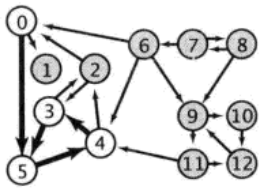
定义。*有向无环图*（DAG）就是一幅不含有环的有向图。

因此，解决有向图检测的问题可以回答下面这个问题：一幅有向图是有向无环图吗？基于深度优先搜索来解决这个问题并不困难，因为由系统维护的递归调用的栈表示的正是“当前”正在遍历的有向路径（就好像用 Tremaux 方法探索迷宫时的那条绳子一样）。一旦我们找到了一条边 v → w 且 w 已经存在于栈中，就找到了一个环，因为栈表示的是一条由 w 到 v 的有向路径，而 v → w 正好

补全了这个环。同时，如果没有找到这样的边，那就意味着这幅有向图是无环的，见图 4.2.9。请见后面框注“寻找有向环”中的 DirectedCycle 基于这个思想实现了表 4.2.5 中的 API。

表 4.2.5 有向环的 API

public class	DirectedCycle	
	DirectedCycle(Digraph G)	寻找有向环的构造函数
boolean	hasCycle()	G 是否含有有向环
Iterable<Integer>	cycle()	有向环中的所有顶点（如果存在的话）



	marked[] 0 1 2 3 4 5 ...	edgeTo[] 0 1 2 3 4 5 ...	onStack[] 0 1 2 3 4 5 ...
dfs(0)			
dfs(5)	1	0	1
dfs(4)	1 1	5 0	1 1
dfs(3)	1 1 1	4 5 0	1 1 1
检查5号顶点	1 0 0 1 1 1	- - - 4 5 0	1 0 0 1 1 1

575 ≀ 576

图 4.2.9 在一幅有向图中寻找环

寻找有向环

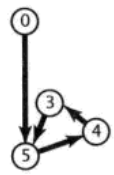
```
public class DirectedCycle
{
   private boolean[] marked;
   private int[] edgeTo;
   private Stack<Integer> cycle;    // 有向环中的所有顶点（如果存在）
   private boolean[] onStack;       // 递归调用的栈上的所有顶点

   public DirectedCycle (Digraph G)
   {
      onStack = new boolean[G.V()];
      edgeTo  = new int[G.V()];
      marked  = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }
   private void dfs(Digraph G, int v)
   {
      onStack[v] = true;
      marked[v] = true;
      for (int w : G.adj(v))
         if (this.hasCycle()) return;
         else if (!marked[w])
         {  edgeTo[w] = v; dfs(G, w);  }
         else if (onStack[w])

         {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x])
               cycle.push(x);

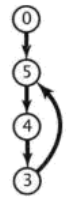
            cycle.push(w);
            cycle.push(v);

         }
      onStack[v] = false;
   }
```



edgeTo[]

0	
1	
2	
3	4
4	5
5	0



v	w	x	有向环
5	5	3	3
3	5	4	4 3
4	5	5	5 4 3
3	5	5	3 5 4 3

有向环检测的轨迹

```
    public boolean hasCycle()
    {  return cycle != null;  }

    public Iterable<Integer> cycle()
    {  return cycle;  }
}
```

该类为标准的递归 dfs() 方法添加了一个布尔类型的数组 onStack[] 来保存递归调用期间栈上的所有顶点。当它找到一条边 v → w 且 w 在栈中时，它就找到了一个有向环。环上的所有顶点可以通过 edgeTo[] 中的链接得到。 577

在执行 dfs(G,v) 时，查找的是一条由起点到 v 的有向路径。要保存这条路径，DirectedCycle 维护了一个由顶点索引的数组 onStack[]，以标记递归调用的栈上的所有顶点（在调用 dfs(G,v) 时将 onStack[v] 设为 true，在调用结束时将其设为 false）。DirectedCycle 同时也使用了一个 edgeTo[] 数组，在找到有向环时返回环中的所有顶点，方法和 DepthFirstPaths（请见算法 4.1）以及 BreadthFirstPaths（请见算法 4.2）相同。

4.2.4.3 顶点的深度优先次序与拓扑排序

优先级限制下的调度问题等价于计算有向无环图中的所有顶点的拓扑排序，因此有表 4.2.6 所示的 API。

表 4.2.6 拓扑排序的 API

public class	**Topological**	
	Topological(Digraph G)	拓扑排序的构造函数
boolean	isDAG()	G 是有向无环图吗
Iterable<Integer>	order()	拓扑有序的所有顶点

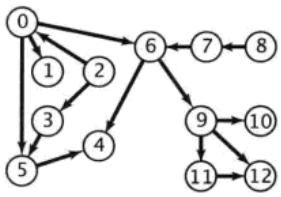
命题 E。 当且仅当一幅有向图是无环图时它才能进行拓扑排序。

证明。 如果一幅有向图含有一个环，它就不可能是拓扑有序的。与此相反，我们将要学习的算法能够计算任意有向无环图的拓扑顺序。

值得注意的是，实际上我们已经见过一种拓扑排序的算法：只要添加一行代码，标准深度优先搜索程序就能完成这项任务！要做到这一点，我们先来看看后面框注“有向图中基于深度优先搜索的顶点排序”的 DepthFirstOrder 类。它的基本思想是深度优先搜索正好只会访问每个顶点一次。如果将 dfs() 的参数顶点保存在一个数据结构中，遍历这个数据结构实际上就能访问图中的所有顶点，遍历的顺序取决于这个数据结构的性质以及是在递归调用之前还是之后进行保存。在典型的应用中，人们感兴趣的是顶点的以下 3 种排列顺序。

- 前序：在递归调用之前将顶点加入队列。
- 后序：在递归调用之后将顶点加入队列。
- 逆后序：在递归调用之后将顶点压入栈。

图 4.2.10 所示的是用 DepthFirstOrder 处理有序无环样图所产生的轨迹。它的实现简单，支持在图的高级处理算法中十分有用的 pre()、post() 和 reversePost() 方法。例如，Topological 类中的 order() 方法就调用了 reversePost() 方法。 578



前序就是dfs()的调用顺序

后序就是顶点遍历完成的顺序

	pre	post	reversePost
dfs(0)	**0**		
dfs(5)	0 **5**		
dfs(4)	0 5 **4**		
4 完成		**4**	**4**
5 完成		4 **5**	5**4**
dfs(1)	0 5 4 **1**		
1 完成		4 5 **1**	**1**5 4
dfs(6)	0 5 4 1 **6**		
dfs(9)	0 5 4 1 6 **9**		
dfs(11)	0 5 4 1 6 9 **11**		
dfs(12)	0 5 4 1 6 9 11**12**		
12 完成		4 5 1 **12**	**12** 1 5 4
11 完成		4 5 1 12 **11**	**11** 12 1 5 4
dfs(10)	0 5 4 1 6 9 11 12 **10**		
10 完成		4 5 1 12 11 **10**	**10** 11 12 1 5 4
检查 12			
9 完成		4 5 1 12 11 10 **9**	**9** 10 11 12 1 5 4
检查 4			
6 完成		4 5 1 12 11 10 9 **6**	**6** 9 10 11 12 1 5 4
0 完成		4 5 1 12 11 10 9 6 **0**	**0** 6 9 10 11 12 1 5 4
检查 1			
dfs(2)	0 5 4 1 6 9 11 12 10**2**		
检查 0			
dfs(3)	0 5 4 1 6 9 11 12 10 2 **3**		
检查 5			
3 完成		4 5 1 12 11 10 9 6 0 **3**	**3** 0 6 9 10 11 12 1 5 4
2 完成			
检查 3		4 5 1 12 11 10 9 6 0 3 **2**	**2** 3 0 6 9 10 11 12 1 5 4
检查 4			
检查 5			
检查 6			
dfs(7)	0 5 4 1 6 9 11 12 10 2 3 **7**		
检查 6			
7 完成		4 5 1 12 11 10 9 6 0 3 2 **7**	**7**2 3 0 6 9 10 11 12 1 5 4
dfs(8)	0 5 4 1 6 9 11 12 10 2 3 7 **8**		
检查 7			
8 完成		4 5 1 12 11 10 9 6 0 3 2 7 **8**	**8**7 2 3 0 6 9 10 11 12 1 5 4
检查 9			
检查 10			
检查 11			
检查 12			

队列　　队列　　栈

逆后序

579

图 4.2.10 计算有向图中顶点的深度优先次序（前序、后序和逆后序）

有向图中基于深度优先搜索的顶点排序

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Queue<Integer> pre;          // 所有顶点的前序排列
   private Queue<Integer> post;         // 所有顶点的后序排列
   private Stack<Integer> reversePost;  // 所有顶点的逆后序排列
```

```
   public DepthFirstOrder(Digraph G)
   {
      pre         = new Queue<Integer>();
      post        = new Queue<Integer>();
      reversePost = new Stack<Integer>();
      marked  = new boolean[G.V()];

      for (int v = 0; v < G.V(); v++)

         if (!marked[v]) dfs(G, v);

   }

   private void dfs(Digraph G, int v)
   {
      pre.enqueue(v);

      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
            dfs(G, w);

      post.enqueue(v);
      reversePost.push(v);
   }

   public Iterable<Integer> pre()
   {  return pre;  }
   public Iterable<Integer> post()
   {  return post;  }
   public Iterable<Integer> reversePost()
   {  return reversePost;  }
}
```

该类允许用例用各种顺序遍历深度优先搜索经过的所有顶点。这在高级的有向图处理算法中非常有用，因为搜索的递归性使得我们能够证明这段计算的许多性质（例如命题 F）。 580

命题 4.5 拓扑排序

```
public class Topological
{
   private Iterable<Integer> order;       // 顶点的拓扑顺序

   public Topological(Digraph G)
   {
      DirectedCycle cyclefinder = new DirectedCycle(G);
      if (!cyclefinder.hasCycle())

      {
         DepthFirstOrder dfs = new DepthFirstOrder(G);

         order = dfs.reversePost();
      }
   }

   public Iterable<Integer> order()
   {  return order;  }
   public boolean isDAG()
   {  return order != null;  }
```

```
   public static void main(String[] args)
   {
      String filename = args[0];
      String separator = args[1];
      SymbolDigraph sg = new SymbolDigraph(filename, separator);

      Topological top = new Topological(sg.G());

      for (int v : top.order())
         StdOut.println(sg.name(v));
   }
}
```

这段代码使用了 DepthFirstOrder 类和 DirectedCycle 类来返回一幅有向无环图的拓扑排序。其中的测试代码解决了一幅 SymbolDigraph 中有优先级限制的调度问题。在给定的有向图包含环时，order() 方法会返回 null，否则会返回一个能够给出拓扑有序的所有顶点的迭代器。这里省略了关于 SymbolDigraph 的代码，因为它和 SymbolGraph（请见表 4.1.1）的代码几乎完全相同，只需把所有的 Graph 替换为 Digraph 即可。

581

命题 F。一幅有向无环图的拓扑排序即为所有顶点的逆后序排列。

证明。对于任意边 v→w，在调用 dfs(v) 时，下面三种情况必有其一成立（请见图 4.2.11）。

- dfs(w) 已经被调用过且已经返回了（w 已经被标记）。
- dfs(w) 还没有被调用（w 还未被标记），因此 v→w 会直接或间接调用并返回 dfs(w)，且 dfs(w) 会在 dfs(v) 返回前返回。
- dfs(w) 已经被调用但还未返回。证明的关键在于，在有向无环图中这种情况是不可能出现的，这是由于递归调用链意味着存在从 w 到 v 的路径，但存在 v→w 则表示存在一个环。

在两种可能的情况中，dfs(w) 都会在 dfs(v) 之前完成，因此在后序排列中 w 排在 v 之前而在逆后序中 w 排在 v 之后。因此任意一条边 v→w 都如我们所愿地从排名较前顶点指向排名较后的顶点。

```
% more jobs.txt
Algorithms/Theoretical CS/Databases/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra

Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks
```

```
% java Topological jobs.txt "/"
Calculus
Linear Algebra
Introduction to CS

Advanced Programming
Algorithms
Theoretical CS
Artificial Intelligence
Robotics
Machine Learning
Neural Networks

Databases
Scientific Computing
Computational Biology
```

Topological 类（请见算法 4.5）的实现使用了深度优先搜索来对有向无环图进行拓扑排序。图 4.2.11 为排序的轨迹。

命题 G。使用深度优先搜索对有向无环图进行拓扑排序所需的时间和 $V+E$ 成正比。

证明。由代码可知，第一遍深度优先搜索保证了不存在有向环，第二遍深度优先搜索产生了顶点的逆后序排序。两次搜索都访问了所有的顶点和所有的边，因此它所需的时间和 $V+E$ 成正比。

尽管算法很简单，但是它被忽略了很多年，比它更流行的是一种使用队列储存顶点的更加直观的算法。（请见练习 4.2.30）

在实际应用中，拓扑排序和有向环的检测总会一起出现，因为有向环的检测是排序的前提。例如，在一个任务调度应用中，无论计划如何安排，其背后的有向图中包含的环意味着存在一个必须被纠正的严重错误。因此，解决任务调度类应用通常需要以下 3 步：

- 指明任务和优先级条件；
- 不断检测并去除有向图中的所有环，以确保存在可行方案的；
- 使用拓扑排序解决调度问题。

类似地，调度方案的任何变动之后都需要再次检查是否存在环（使用 DirectedCycle 类），然后在计算新的调度安排（使用 Topological 类）。

582

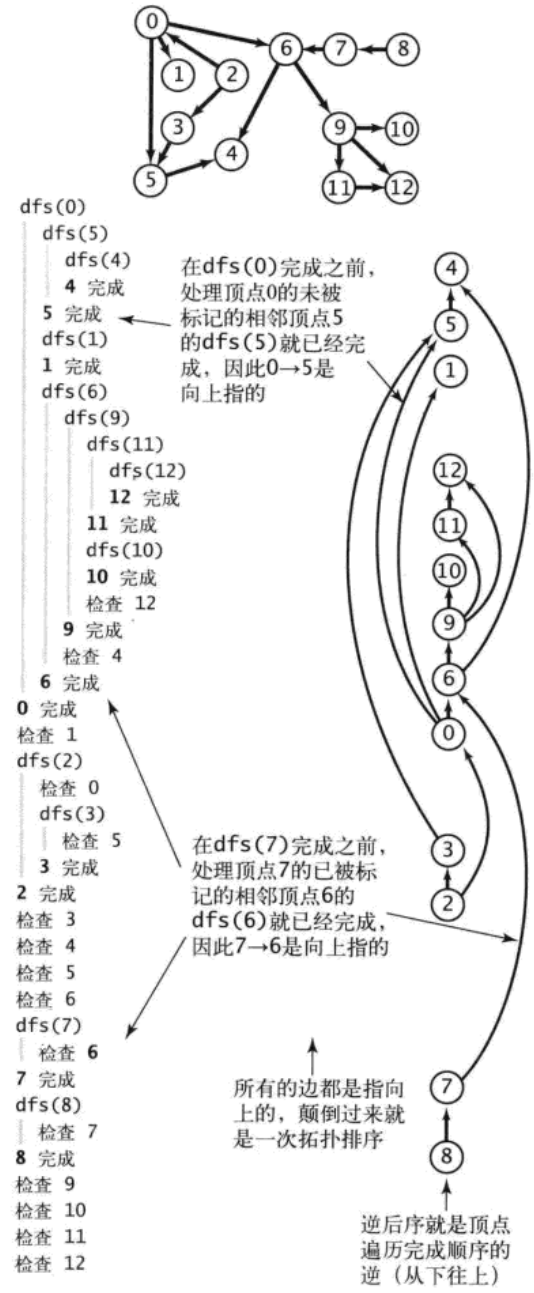


图 4.2.11 有向无环图的逆后序是拓扑排序

582 ~ 583

4.2.5 有向图中的强连通性

在前文中，我们仔细区别了有向图中的可达性和无向图中的连通性。在一幅无向图中，如果有一条路径连接顶点 v 和 w，则它们就是连通的——既可以由这条路径从 w 到达 v，也可以从 v 到达 w。相反，在一幅有向图中，如果从顶点 v 有一条有向路径到达 w，则顶点 w 是从顶点 v 可达的，但从 w 到达 v 的路径可能存在也可能不存在。在对有向图的研究中，我们也会考虑与无向图中的连通性类似的一个问题。

定义。如果两个顶点 v 和 w 是互相可达的，则称它们为**强连通的**。也就是说，既存在一条从 v 到 w 的有向路径，也存在一条从 w 到 v 的有向路径。如果一幅有向图中的任意两个顶点都是强连通的，则称这幅有向图也是**强连通的**。

图 4.2.12 给出了几个强连通图的例子。从这些例子中你可以看到，环在强连通性的理解上起着重要的作用。事实上，回忆一下一条普通的有向环可能含有重复的顶点就很容易知道，两个顶点是强连通的当且仅当它们都在一个普通的有向环中（证明：画出从 v 到 w 和从 w 到 v 的路径即可）。

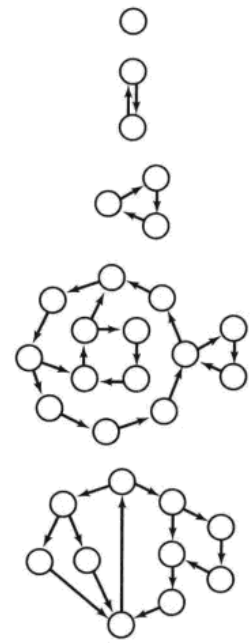


图 4.2.12 强连通的有向图

4.2.5.1 强连通分量

和无向图中的连通性一样，有向图中的强连通性也是一种顶点之间平等关系，因为它有着以下性质。

- 自反性：任意顶点 v 和自己都是强连通的。
- 对称性：如果 v 和 w 是强连通的，那么 w 和 v 也是强连通的。
- 传递性：如果 v 和 w 是强连通的且 w 和 x 也是强连通的，那么 v 和 x 也是强连通的。

作为一种平等关系，强连通性将所有顶点分为了一些平等的部分，每个部分都是由相互均为强连通的顶点的最大子集组成的。我们将这些子集称为**强连通分量**，请见图 4.2.13。样图 tinyDG.txt 含有 5 个强连通分量。一个含有 V 个顶点的有向图含有 1 ~ V 个强连通分量——一个强连通图只含有一个强连通分量，而一个有向无环图中则含有 V 个强连通分量。需要注意的是强连通分量的定义是基于顶点的，而非边。有些边连接的两个顶点都在同一个强连通分量中，而有些边连接的两个顶点则在不同的强连通分量中。后者不会出现在任何有向环之中。与识别连通分量在无向图中的重要性一样，在有向图的处理中识别强连通分量也是非常重要的。

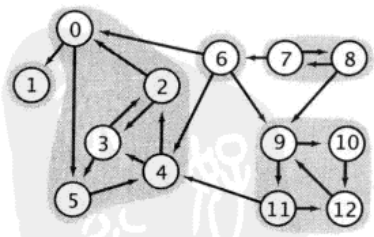


图 4.2.13 一幅有向图和它的强连通分量

4.2.5.2 应用举例

在理解有向图的结构时，强连通性是一种非常重要的抽象，它突出了相互关联的几组顶点（强连通分量）。例如，强连通分量能够帮助教科书的作者决定哪些话题应该被归为一类，或帮助程序员组织程序的模块（请见表 4.2.7）。图 4.2.14 是一个生态学的例子。这幅有向图描绘的是各种生物之间的食物链模型，其中顶点表示物

种，而从一个顶点指向另一个顶点的一条边则表示指向顶点的物种对指出顶点的物种的捕食关系。这些有向图（其中物种和捕食关系都是经过仔细选择和研究的）的科学研究有效地帮助了生态学家解决生态系统中的一些基本问题。这种有向图中的强连通分量能够帮助生态学家理解食物链中能量的流动。图 4.2.17 所示的是一张表示网络内容的有向图，其中顶点表示网页，而边表示从一个页面指向另一个页面的超链接。在这样一幅有向图中，强连通分量能够帮助网络工程师将网络中数量庞大的网页分为多个大小可以接受的部分分别进行处理。练习和本书的网站会涉及这些应用和其他例子的更多性质。

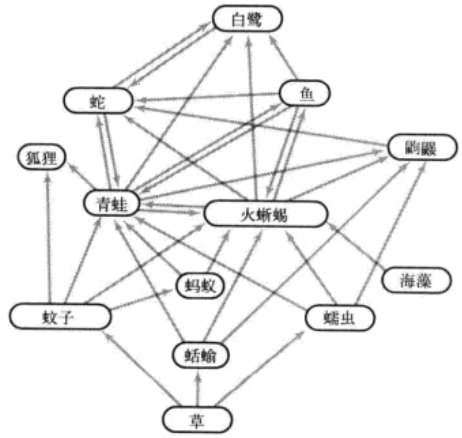


图 4.2.14 一幅表示食物链的有向图的一小部分

表 4.2.7 强连通分量的典型应用

应 用	顶 点	边
网络	网页	超链接
教科书	话题	引用
软件	模块	调用
食物链	物种	捕食关系

584 ~ 585

因此，在有向图中我们也需要表 4.2.8 所列的这份和 CC（请见表 4.1.6）类似的 API。

表 4.2.8 强连通分量的 API

public class	SCC	
	SCC(Digraph G)	预处理构造函数
boolean	stronglyConnected(int v, int w)	v 和 w 是强连通的吗
int	count()	图中的强连通分量的总数
int	id(int v)	v 所在的强连通分量的标识符（在 0 至 count()-1 之间）

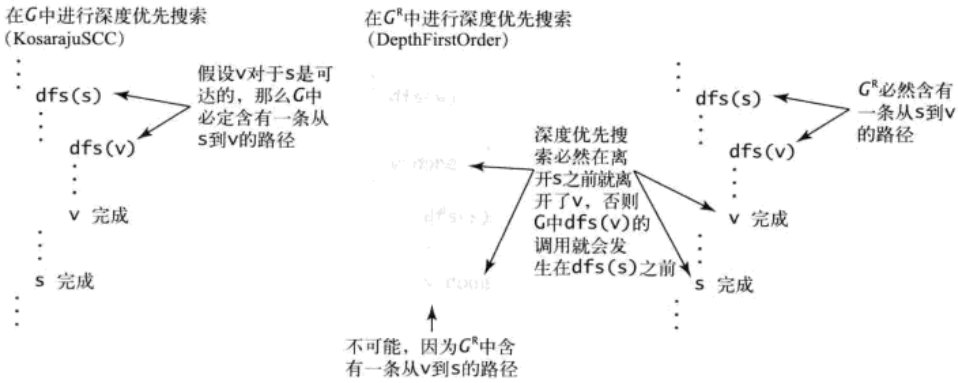
设计一种平方级别的算法来计算强连通分量（请见练习 4.2.23）并不困难，但（和以前一样）对于处理在实际应用中经常遇到的像刚才示例所示的大型有向图来说，平方级别的时间和空间需求是不可接受的。

4.2.5.3 Kosaraju 算法

我们在 CC（请见算法 4.3）中看到过，计算无向图中的连通分量只是深度优先搜索的一个简单应用。那么在有向图中应该如何高效地计算强连通分量呢？令人惊讶的是，算法 4.6 中的 KosarajuCC 的实现只为 CC 添加了几行代码就做到了这一点，它将会完成以下任务（请见图 4.2.15）。

- 在给定的一幅有向图 G 中，使用 DepthFirstOrder 来计算它的反向图 G^R 的逆后序排列。
- 在 G 中进行标准的深度优先搜索，但是要按照刚才计算得到的顺序而非标准的顺序来访问所有未被标记的顶点。

- 在构造函数中，所有在同一个递归 dfs() 调用中被访问到的顶点都在同一个强连通分量中，将它们按照和 CC 相同的方式识别出来。



586

图 4.2.15 Kosaraju 算法的正确性证明

算法 4.6 计算强连通分量的 Kosaraju 算法

```
public class KosarajuSCC
{
   private boolean[] marked;    // 已访问过的顶点
   private int[] id;            // 强连通分量的标识符
   private int count;           // 强连通分量的数量

   public KosarajuSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder order = new DepthFirstOrder(G.reverse());
      for (int s : order.reversePost())
         if (!marked[s])
         {  dfs(G, s); count++;  }
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      id[v] = count;
      for (int w : G.adj(v))
         if (!marked[w])
            dfs(G, w);
   }

   public boolean stronglyConnected(int v, int w)
   {  return id[v] == id[w];  }

   public int id(int v)
   {  return id[v];  }

   public int count()
```

```
% java KosarajuSCC tinyDG.txt
5 components
1
0 5 4 3 2
11 12 9 10
6
8 7
```

```
    { return count; }
}
```

突出显示的代码是这份实现和 CC（请见算法 4.3）仅有的不同之处（还需要将 4.1.6.1 节中用到的 main() 函数中的 Graph 替换为 Digraph，CC 替换为 KosarajuSCC）。为了找到所有强连通分量，它会在反向图中进行深度优先搜索来将顶点排序（搜索顺序的逆后序），在给定有向图中用这个顺序再进行一次深度优先搜索。

587

Kosaraju 算法是一个典型示例，这个方法容易实现但难以理解。尽管它有些神秘，但如果你能一步一步地理解下面这个命题的证明并参考图 4.2.4，那你一定可以理解这个算法的正确性。

命题 H。使用深度优先搜索查找给定有向图 G 的反向图 G^R，根据由此得到的所有顶点的逆后序再次用深度优先搜索处理有向图 G（Kosaraju 算法），其构造函数中的每一次递归调用所标记的顶点都在同一个强连通分量之中。

证明。首先要用反证法证明“*每个和 s 强连通的顶点 v 都会在构造函数调用的 dfs(G,s) 中被访问到*”。假设有一个和 s 强连通的顶点 v 不会在构造函数调用的 dfs(G,s) 中被访问到。因为存在从 s 到 v 的路径，所以 v 肯定在之前就已经被标记过了。但是，因为也存在从 v 到 s 的路径，在 dfs(G,v) 调用中 s 肯定会被标记，因此构造函数应该是不会调用 dfs(G,s) 的。矛盾。

其次，要证明“*构造函数调用的 dfs(G,s) 所到达的任意顶点 v 都必然是和 s 强连通的*”。设 v 为 dfs(G,s) 到达的某个顶点。那么，G 中必然存在一条从 s 到 v 的路径，因此只需要证明 G 中还存在一条从 v 到 s 的路径即可。这也等价于 G^R 中存在一条从 s 到 v 的路径，因此只需要证明在 G^R 中存在一条从 s 到 v 的路径即可。

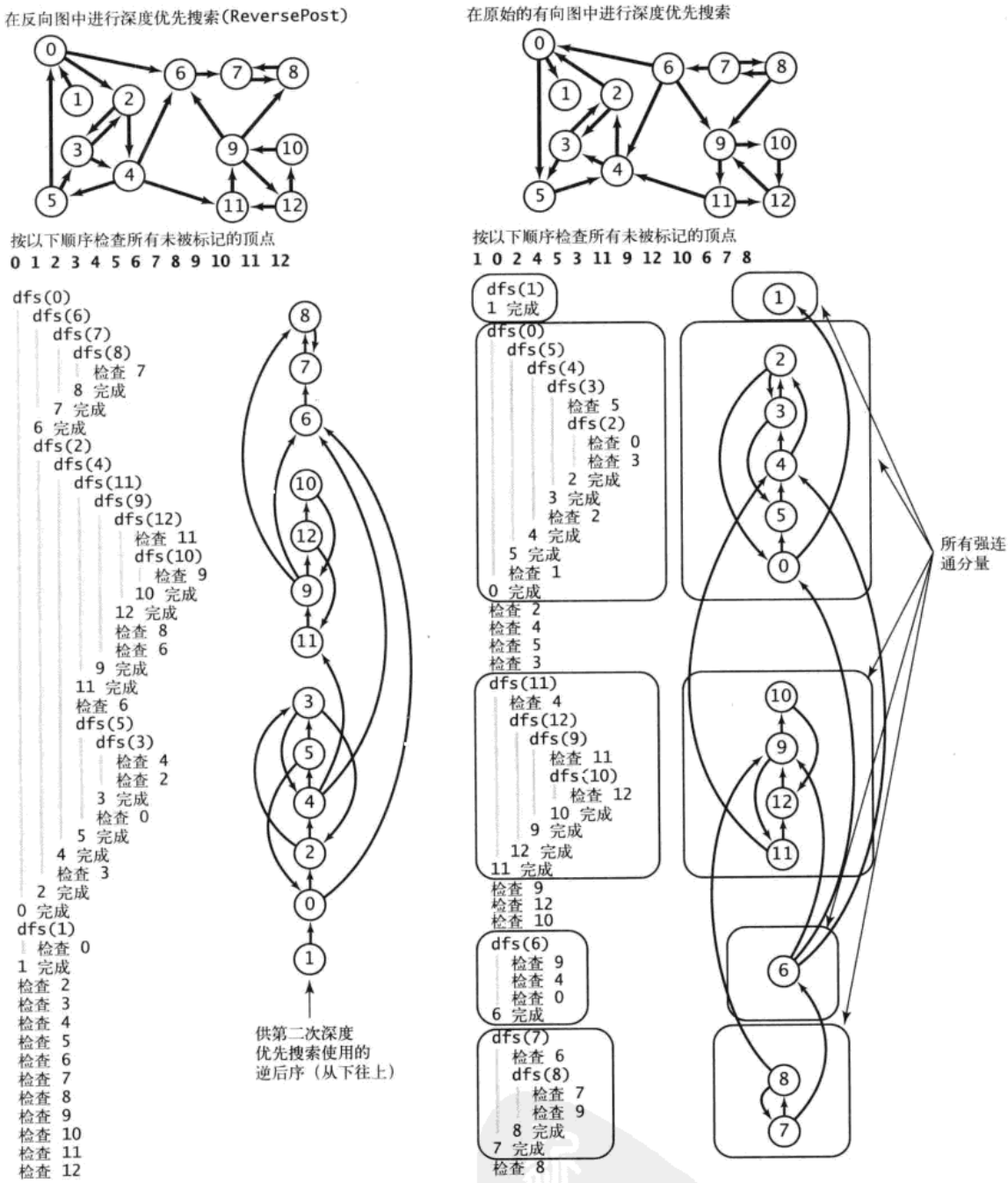
证明的核心在于，按照后逆序进行的深度优先搜索意味着，在 G^R 中进行的深度优先搜索中，dfs(G,v) 必然在 dfs(G,s) 之前就已经结束了，这样 dfs(G,v) 的调用就只会出现两种情况：

- 调用在 dfs(G,s) 的调用之前（并且也在 dfs(G,s) 的调用之前结束）；
- 调用在 dfs(G,s) 的调用之后（并且也在 dfs(G,s) 的结束之前结束）。

第一种情况是不可能出现的，因为在 G^R 中存在一条从 v 到 s 的路径；而第二种情况则说明 G^R 中存在一条从 s 到 v 的路径。证毕。

图 4.2.16 所示为 Kosaraju 算法处理 tinyDG.txt 时的轨迹。在每次 dfs() 调用轨迹的右侧都是有向图的一部分，顶点按照搜索结束的顺序排列。因此，从下往上来看左侧这幅有向图的反向图得到的就是所有顶点的逆后序，也就是在原始的有向图中进行深度优先搜索时所有未被标记的顶点被检查的顺序。你可以从图中看到，在第二遍深度优先搜索中，首先调用的是 dfs(1)（标记顶点 1），然后调用的是 dfs(0)（标记顶点 5、4、3 和 2），然后检查了顶点 2、4、5 和 3，再调用 dfs(11)（标记顶点 11、12、9 和 10），在检查了 9、12 和 10 之后调用 dfs(6)（标记顶点 6），最后调用 dfs(7) 标记了顶点 7 和 8。

588



589

图 4.2.16　在有向图中寻找强连通分量的 Kosaraju 算法

图 4.2.17 中所示的是一个更大的示例，也是 Web 的有向图模型的一个非常小的部分。

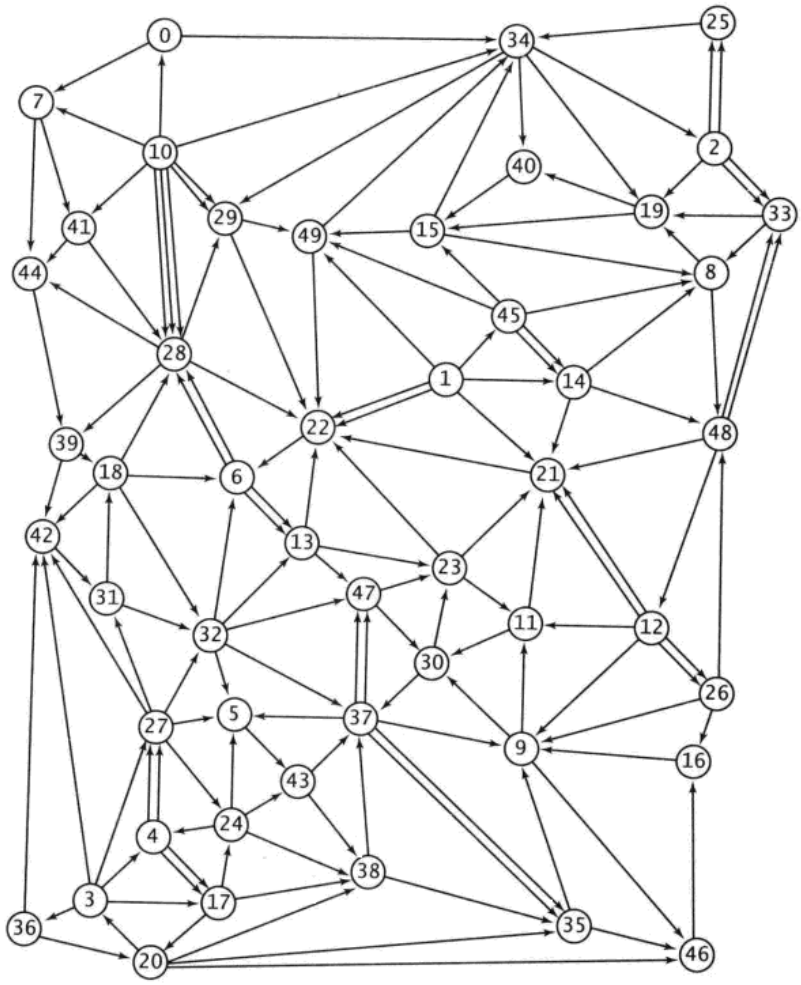


图 4.2.17 这幅有向图中含有多少个强连通分量

我们在第 1 章已经介绍过 kosaraju 算法并在 4.1 节中再次使用该算法解决了无向图的连通性问题。Kosaraju 算法也解决了有向图中的类似问题。

强连通性。给定一幅有向图，回答“给定的两个顶点是强连通的吗？这幅有向图中含有多少个强连通分量？”等类似问题。

我们能否用和无向图相同的效率解决有向图的连通性问题？这个问题已经被研究了很长时间了（R.E.Tarjan 在 20 世纪 70 年代末解决了这个问题）。这样一个简单的解决方法实在令人惊讶。

命题 I。Kosaraju 算法的预处理所需的时间和空间与 $V+E$ 成正比且支持常数时间的有向图强连通性的查询。

证明。该算法会处理有向图的反向图并进行两次深度优先搜索。这 3 步所需的时间都与 $V+E$ 成正比。反向复制一幅有向图所需的空间与 $V+E$ 成正比。

4.2.5.4　再谈可达性

根据 CC 类我们可以知道，在无向图中如果两个顶点 v 和 w 是连通的，那么就既存在一条从 v 到 w 的路径也存在一条从 w 到 v 的路径。根据 KosarajuCC 类可知，在有向图中如果两个顶点 v 和 w 是强连通的，那么也既存在一条从 v 到 w 的路径也存在（另）一条从 w 到 v 的路径。但对于一对非强连通的顶点呢？也许存在一条从 v 到 w 的路径，也许存在一条从 w 到 v 的路径，也许两条都不存在，但两条不可能都存在。

顶点对的可达性。给定一幅有向图，回答“是否存在一条从一个给定的顶点 v 到另一个给定的顶点 w 的路径？”等类似问题。

对于无向图，这个问题等价于连通性问题；对于有向图，它和强连通性的问题有很大区别。CC 实现需要线性级别的预处理时间才能支持常数时间的查询操作。我们能够在有向图的相应实现中达到这样的性能吗？这个看似简单的问题困扰了专家数十年。为了更好地理解这个问题，我们来看看图 4.2.18。它展示了下面这个基本的概念。

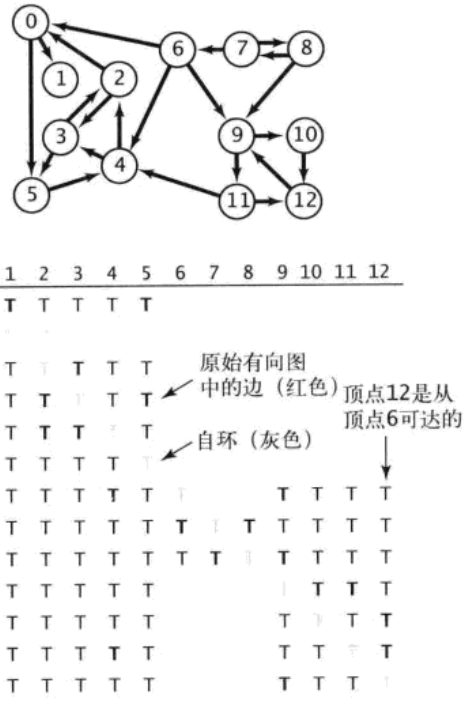


图 4.2.18　传递闭包（另见彩插）

定义。有向图 G 的传递闭包是由相同的一组顶点组成的另一幅有向图，在传递闭包中存在一条从 v 指向 w 的边当且仅当在 G 中 w 是从 v 可达的。

根据约定，每个顶点对于自己都是可达的，因此传递闭包会含有 V 个自环。样图只有 13 条边，但它的传递闭包含有可能的 169 条边中的 102 条。一般来说，一幅有向图的传递闭包中所含的边都比原图中多得多，一幅稀疏图的传递闭包却是一幅稠密图也是很常见的。例如，含有 V 个顶点和 V 条边的有向环的传递闭包是一幅含有 V^2 条边的有向完全图。因为传递闭包一般都很稠密，我们通常都将它们表示为一个布尔值矩阵，其中 v 行 w 列的值为 true 当且仅当 w 是从 v 可达的。与其明确计算一幅有向图的传递闭包，不如使用深度优先搜索来实现表 4.2.9 中的 API。

表 4.2.9　顶点对可达性的 API

public class	TransitiveClosure	
	TransitiveClosure(Digraph G)	预处理的构造函数
boolean	reachable(int v, int w)	w 是从 v 可达的吗

下页框注中的代码使用 DirectedDFS（请见算法 4.4）简单明了地实现了它。无论对于稀疏还是稠密的图，它都是理想解决方案，但它不适用于在实际应用中可能遇到的大型有向图，因为

```
public class TransitiveClosure
{
   private DirectedDFS[] all;
   TransitiveClosure(Digraph G)
   {
      all = new DirectedDFS[G.V()];
      for (int v = 0; v < G.V(); v++)
         all[v] = new DirectedDFS(G, v);
   }

   boolean reachable(int v, int w)
   {  return all[v].marked(w);  }
}
```

顶点对的可达性

构造函数所需的空间和 V^2 成正比，所需的时间和 $V(V+E)$ 成正比：共有 V 个 DirectedDFS 对象，每个所需的空间都与 V 成正比（它们都含有大小为 V 的 marked[] 数组并会检查 E 条边来计算标记）。本质上，TransitiveClosure 通过计算 G 的传递闭包来支持常数时间的查询——传递闭包矩阵中的第 v 行就是 TransitiveClosure 类中的 DirectedDFS[] 数组的第 v 个元素的 marked[] 数组。我们能够大幅度减少预处理所需的时间和空间同时又保证常数时间的查询吗？用远小于平方级别的空间支持常数级别的查询的一般解决方案仍然是一个有待解决的研究问题，并且有重要的实际意义：例如，除非这个问题得到解决，对于像代表互联网这样的巨型有向图，否则无法有效解决其中的顶点对可达性问题。

590 ~ 593

4.2.6 总结

在本节中，我们介绍了有向边和有向图并强调了有向图处理算法和无向图处理中相应算法的关系，涵盖了以下几个方面：

- 有向图的术语；
- 有向图的表示和算法在本质上和无向图是相同的，但部分有向图问题更加复杂；
- 有向环、有向无环图、拓扑排序和优先级限制下的调度问题；
- 有向图的可达性、路径和强连通性。

表4.2.10总结了我们已经学过的各种有向图算法的实现（只有一个算法不基于深度优先搜索）。这些问题的描述都很简单，但它们的解决方法有的仅仅简单改造了无环图中的相应问题的处理算法，有的却非常巧妙。我们将在 4.4 节中遇到加权有向图，这些算法将是学习更加复杂的算法的基础。

表 4.2.10 本节中得到解决的有向图处理问题

问　题	解决方法	参　阅
单点和多点的可达性	DirectedDFS	算法 4.4
单点有向路径	DepthFirstDirectedPaths	4.2.3.2
单点最短有向路径	BreadthFirstDirectedPaths	4.2.3.2
有向环检测	DirectedCycle	4.2.4.2 框注“查找有向环”
深度优先的顶点排序	DepthFirstOrder	4.2.4.2 框注“有向图中基于深度优先搜索的顶点排序”
优先级限制下的调度问题	Topological	算法 4.5
拓扑排序	Topological	算法 4.5
强连通性	KosarajuSCC	算法 4.6
顶点对的可达性	TransitiveClosure	4.2.5.4 节

594

答疑

问　自环是一个环吗？

595 答　是的，但没有自环的顶点对于自己也是可达的。

练习

4.2.1　一幅含有 V 个顶点且没有平行边的有向图中最多可能含有多少条边？一幅含有 V 个顶点且没有孤立顶点的有向图中最少需要多少条边？

4.2.2　按照正文中示意图的样式（请见图 4.1.10）画出 Digraph 的构造函数在处理图 4.2.19 的 tinyDGex2.txt 时构造的邻接表。

4.2.3　为 Digraph 添加一个构造函数，它接受一幅有向图 G 然后创建并初始化这幅图的一个副本。G 的用例的对它作出的任何改动都不应该影响到它的副本。

4.2.4　为 Digraph 添加一个方法 hasEdge()，它接受两个整型参数 v 和 w。如果图含有边 v→w，方法返回 true，否则返回 false。

4.2.5　修改 Digraph，不允许存在平行边和自环。

4.2.6　为 Digraph 编写一个测试用例。

tinyDGex2.txt

```
V → 12
16 ← E
8 4
2 3
0 5
0 6
3 6
10 3
7 11
7 8
11 8
2 0
6 2
5 2
5 10
3 10
8 1
4 1
```

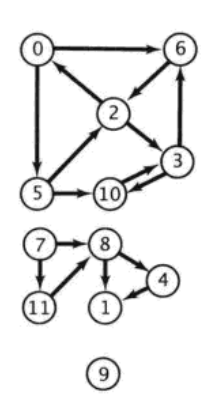


图　4.2.19

4.2.7　顶点的入度为指向该顶点的边的总数。顶点的出度为由该顶点指出的边的总数。从出度为 0 的顶点是不可能达到任何顶点的，这种顶点叫做终点；入度为 0 的顶点是不可能从任何顶点到达的，所以叫做起点。一幅允许出现自环且每个顶点的出度均为 1 的有向图叫做映射（从 0 到 V–1 之间的整数到它们自身的函数）。编写一段程序 Degrees.java，实现下面的 API，如表 4.2.11 所示。

表　4.2.11

public class	**Degrees**	
	Degrees(Digraph G)	构造函数
int	indegree(int v)	v 的入度
int	outdegree(int v)	v 的出度
Iterable<Integer>	sources()	所有起点的集合
Iterable<Integer>	sinks()	所有终点的集合
596 boolean	isMap()	G 是一幅映射吗

4.2.8　画出所有含有 2、3、4 和 5 个顶点的非同构有向无环图。（参考练习 4.1.28）

4.2.9　编写一个方法，检查一幅有向无环图的顶点的给定排列是否就是该图顶点的拓扑排序。

4.2.10　给定一幅有向无环图，是否存在一种无法用基于深度优先搜索算法得到的顶点的拓扑排序？顶点的相邻关系不限。证明你的结论。

4.2.11 描述一组稀疏有向图，其含有的有向环的个数随着顶点增加而呈指数级增长。

4.2.12 一幅含有 V 个顶点和 V–1 条边且为一条简单路径的有向图的传递闭包中含有多少条边？

4.2.13 给出这幅含有 10 个顶点和以下边的有向图的传递闭包：

3→7 1→4 7→8 0→5 5→2 3→8 2→9 0→6 4→9 2→6 6→4

4.2.14 证明 G 和 G^R 中的强连通分量是相同的。

4.2.15 一幅有向无环图的强连通分量是哪些？

4.2.16 用 Kosaraju 算法处理一幅有向无环图的结果是什么？

4.2.17 真假判断：一幅有向图的反向图的顶点的后逆序排列和该有向图的顶点的后序排列相同。

4.2.18 使用 1.4 节中的内存使用模型评估含有 V 个顶点和 E 条边的 `Digraph` 的内存使用情况。 597

提高题

4.2.19 *拓扑排序与广度优先搜索。*解释为何如下算法无法得到一组拓扑排序：运行广度优先搜索并按照所有顶点和起点的距离标记它们。

4.2.20 *有向欧拉环。*欧拉环是一条每条边只出现一次的有向环。编写一个程序 `Euler` 来找出有向图中的欧拉环或者说明它不存在。*提示：*当且仅当有向图 G 是连通的且每个顶点的出度和入度相同时 G 含有一条有向欧拉环。

4.2.21 *有向无环图中的 LCA。*给定一幅有向无环图和两个顶点 `v` 和 `w`，找出 `v` 和 `w` 的 LCA（Lowest Common Ancestor，最近共同祖先）。LCA 的计算在实现编程语言的多重继承、分析家谱数据（找出家族中近亲繁衍的程度）和其他一些应用中很有用。*提示：*将有向无环图中的顶点 `v` 的高度定义为从根结点到 `v` 的最长路径。在所有 `v` 和 `w` 的共同祖先中，高度最大者就是 `v` 和 `w` 的最近共同祖先。

4.2.22 *最短先导路径。*给定一幅有向无环图和两个顶点 `v` 和 `w`，找出 `v` 和 `w` 之间的最短先导路径。设 `v` 和 `w` 的一个共同的祖先顶点为 `x`，先导路径为 `v` 到 `x` 的最短路径和 `w` 到 `x` 的最短路径。`v` 和 `w` 之间的最短先导路径是所有先导路径中的最短者。*热身：*构造一幅有向无环图，使得最短先导路径到达的祖先顶点 `x` 不是 `v` 和 `w` 的最近共同祖先。*提示：*进行两次广度优先搜索，一次从 `v` 开始，一次从 `w` 开始。

4.2.23 *强连通分量。*设计一种线性时间的算法来计算给定顶点 `v` 所在的强连通分量。在这个算法的基础上设计一种平方时间的算法来计算有向图的所有强连通分量。

4.2.24 *有向无环图中的汉密尔顿路径。*设计一种线性时间的算法来判定给定的有向无环图中是否存在一条能够正好只访问每个顶点一次的有向路径。

答案：计算给定图的拓扑排序并顺序检查拓扑排序中每一对相邻的顶点之间是否存在一条边。

4.2.25 *唯一的拓扑排序。*设计一个算法来判定一幅有向图的拓扑排序是否是唯一的。*提示：*当且仅当拓扑排序中每一对相邻的顶点之间都存在一条有向边（即有向图含有一条汉密尔顿路径）时它的拓扑排序才是唯一的。如果一幅有向图的拓扑排序不唯一，另一种拓扑排序可以由交换拓扑排序中的某一对相邻的顶点得到。 598

4.2.26 *2- 可满足性。*给定一个由 M 个子句和 N 个变量的组成的以合取范式形式给出的布尔逻辑命题，每个子句都正好含有两个变量，找到一组使布尔表达式为真的变量赋值（如果存在）。*提示：*构造一幅含有 $2N$ 个顶点的蕴涵有向图（implication graph）（每个变量和它的反都各有一个顶点）。对于每个子句 `x+y`，添加一条从 `y'` 到 `x` 的边和一条从 `x'` 到 `y` 的边。要满足子句 `x+y`，必有（i）

如果 y 是假那么 x 为真，或者 (ii) 如果 x 是假那么 y 为真。说明：当且仅当没有任何顶点 x 和它的反 x′存在于同一个强连通分量中时这个表达式才能被满足。另外，核心有向无环图（将每个强连通分量看作一个顶点）的拓扑排序也能够产生一组可以满足该表达式的变量赋值。

4.2.27 有向图的枚举。证明所有不同的含有 V 个顶点且不含平行边的有向图的总数为 2^{V^2} 个。（含有 V 个顶点和 E 条边的不同有向图有多少个？）假设宇宙中每个电子在一纳秒内能够检查一幅有向图，宇宙中的电子总数不超过 10^{80} 个，宇宙的寿命小于 10^{20} 年。对于所有含有 20 个顶点的不同有向图，计算机最多能够检查它们的百分之几？

4.2.28 有向无环图的枚举。给出一个公式，计算含有 V 个顶点和 E 条边的所有有向无环图的数量。

4.2.29 算术表达式。编写一个类来计算由有向无环图表示的算术表达式。使用一个由顶点索引的数组来保存每个顶点所对应的值。假设叶子结点中的值是常数。描述一组算术表达式，使得它所对应的表达式树（expression tree）的大小是相应的有向无环图的大小的指数级别。（因此程序处理有向无环图所需的时间将和处理表达式树所需的时间的对数成正比。）

4.2.30 基于队列的拓扑排序。实现一种拓扑排序，使用由顶点索引的数组来保存每个顶点的入度。遍历一遍所有边并使用练习 4.2.7 给出的 Degrees 类来初始化数组以及一条含有所有顶点的队列。然后，重复以下操作直到起点队列为空：

- 从队列中删去一个顶点并将其标记；
- 遍历由被删除顶点指出的所有边，将所有被指向的顶点的入度减一；
- 如果顶点的入度变为 0，将它插入顶点队列。

599

4.2.31 有向欧拉图。修改你为 4.1.37 给出的解答，为平面图设计一份 API 名为 EulideanDigraph，这样你就能够处理用图形表示的图了。

600

实验题

4.2.32 随机有向图。编写一个程序 ErdosRenyiDigraph，从命令行接受整数 V 和 E，随机生成 E 对 0 到 V–1 之间的整数来构造一幅有向图。注意：生成器可能会产生自环和平行边。

4.2.33 随机简单有向图。编写一个程序 RandomSimpleDigraph，从命令行接受整数 V 和 E，用均等的几率生成含有 V 个顶点和 E 条边的所有可能的简单有向图。

4.2.34 随机稀疏有向图。将你为练习 4.1.41 给出的解答修改为 RandomSparseDigraph，根据精心选择的一组 V 和 E 的值生成随机的稀疏有向图，使得我们可以用它进行有意义的经验性测试。

4.2.35 随机欧拉图。将你为练习 4.1.42 给出的解答修改为 EulideanDigraph 的用例 RandomEulideanDigraph，随机指定每条边的方向。

4.2.36 随机网格图。将你为练习 4.1.43 给出的解答修改为 EulideanDigraph 的用例 RandomGridDigraph，随机指定每条边的方向。

4.2.37 真实世界中的有向图。从互联网上找出一幅巨型有向图——可以是某个在线商业系统的交易图，或是由网页和链接得到的有向图。编写一段程序 RandomRealDigraph，从这些顶点构成的子图中随机选取 V 个顶点，然后再从这些顶点构成的子图中随机选取 E 条边来构造一幅图。

4.2.38 真实世界中的有向无环图。从互联网上找出一幅巨型有向无环图——可以是大型软件系统中的类依赖关系，或是大型文件系统中的目录结构。编写一段程序 RandomRealDAG，从这些顶点构成的子图中随机选取 V 个顶点，然后再从这些顶点构成的子图中随机选取 E 条边来构造一幅图。

601

测试所有的算法并研究所有图模型的所有参数是不现实的。请为下面的每一道题都编写一

段程序来处理从输入得到的任意图。这段程序可以调用上面的任意生成器并对相应的图模型进行实验。你可以根据上次实验的结果自己作出判断来选择不同实验。陈述结果以及由此得出的任何结论。

4.2.39 可达性。对于各种有向图的模型，运行实验并根据经验判断从一个随机选定的顶点可以到达的顶点数量的平均值。

4.2.40 深度优先搜索中的路径长度。对于各种有向图的模型，运行实验并根据经验判断 DepthFirstDirectedPaths 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。

4.2.41 广度优先搜索中的路径长度。对于各种有向图的模型，运行实验并根据经验判断 BreadthFirstDirectedPaths 在两个随机选定的顶点之间找到一条路径的概率并计算找到的路径的平均长度。

4.2.42 强连通分量。运行实验随机生成大量有向图并画出柱状图，根据经验判断各种类型的随机有向图中强连通分量的数量的分布情况。 602

4.3 最小生成树

加权图是一种为每条边关联一个权值或是成本的图模型。这种图能够自然地表示许多应用。在一幅航空图中，边表示航线，权值则可以表示距离或是费用。在一幅电路图中，边表示导线，权值则可能表示导线的长度即成本，或是信号通过这条线路所需的时间。在这些情形中，最令人感兴趣的自然是将成本最小化。在本节中，我们将学习加权无向图模型并用算法回答下面这个问题。

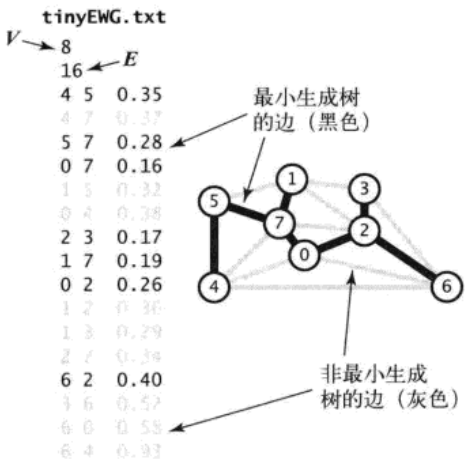


图 4.3.1 一幅加权无向图和它的最小生成树

最小生成树。给定一幅加权无向图，找到它的一棵最小生成树。

定义。图的**生成树**是它的一棵含有其所有顶点的无环连通子图。一幅加权无向图的**最小生成树**（MST）是它的一棵权值（树中所有边的权值之和）最小的生成树。（请见图 4.3.1）。

在本节中，我们会学习计算最小生成树的两种经典算法：Prim 算法和 Kruskal 算法。这些算法理解容易，实现简单。它们是本书中最古老和最知名的算法之一，但它们也根据现代数据结构得到了改进。因为最小生成树的重要应用领域太多，对解决这个问题的算法的研究至少从 20 世纪 20 年代在设计电力分配网络时就开始了。现在，最小生成树算法在设计各种类型的网络（通信、电子、水利、计算机、公路、铁路、航空等）以及自然界中的生物、化学和物理网络等各个领域的研究中都起到了重要的作用，请见表 4.3.1。

表 4.3.1 最小生成树的典型应用

应用领域	顶 点	边
电路	元器件	导线
航空	机场	航线
电力分配	电站	输电线
图像分析	面部容貌	相似关系

603
~
604

一些约定

在计算最小生成树的过程中可能会出现各种特殊情况。虽然它们大多数都很容易处理，但为了行文的流畅，我们约定如下。

- 只考虑连通图。我们对生成树的定义意味着最小生成树只可能存在于连通图中，请见图 4.3.2a。从另一个角度来说，请回想 4.1 节所述的树的基本性质，我们要找的就是一个由 V–1 条边组成的集合，它们既连通了图中的所有顶点而权值之和又最小。如果一幅图是非连通的，我们只能使用这个算法来计算它的所有连通分量的最小生成树，合并在一起称其为最小生成森林（请见练习 4.3.22）。
- 边的权重不一定表示距离。有时你对几何学的直觉能够帮助你理解算法，因此在示例中，顶点都表示是平面上的点，而权重都表示是两点之间的距离，比如图 4.3.2b。但需要注意的是，

权重也可能表示时间、费用或是其他完全不同的变量，而且也完全不一定会和距离成正比。

- 边的权重可能是 0 或者负数。如果边的权重都是正的，将最小生成树定义为连接所有顶点且总权重最小的子图就足够了，这样的一幅子图必然是一棵生成树。定义中的生成树条件说明图也可以含有权重为 0 或是负数的边，请见图 4.3.2c。
- 所有边的权重都各不相同。如果不同边的权重可以相同，最小生成树就不一定唯一了（请见练习 4.3.2）。存在多棵最小生成树的可能性会使部分算法的证明变得更加复杂，因此我们在表示中排除了这种可能性。事实上这个假设并没有限制算法的适用范围，因为不做修改它们也能处理存在等值权重的情况，请见图 4.3.2d 。

总之，在学习最小生成树相关算法的过程中我们假设任务的目标是在一幅加权（但权值各不相同的）连通无向图中找到它的最小生成树。

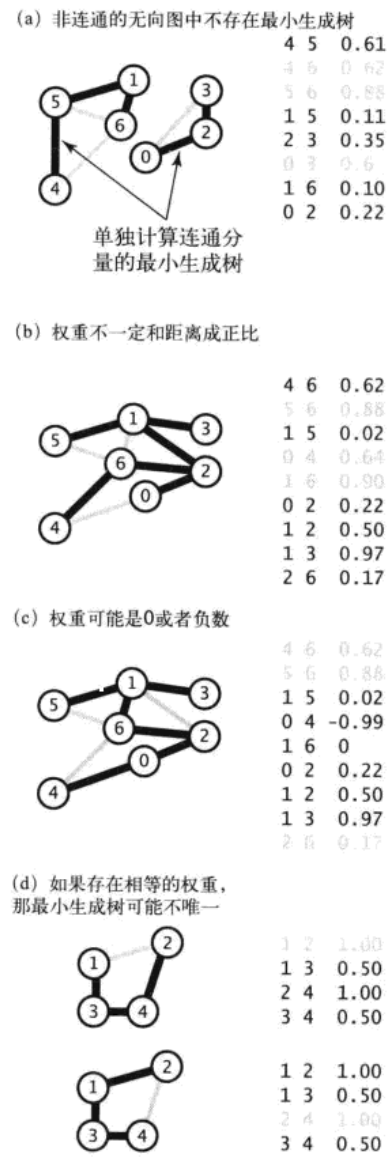


图 4.3.2 计算最小生成树时可能遇到的各种特殊情况

4.3.1 原理

首先，我们回顾一下 4.1 节中给出的树的两个最重要的性质，另见图 4.3.3：

- 用一条边连接树中的任意两个顶点都会产生一个新的环；
- 从树中删去一条边将会得到两棵独立的树。

这两条性质是证明最小生成树的另一条基本性质的基础，而由这条基本性质就能够得到本节中的最小生成树算法。

4.3.1.1 切分定理

我们称之为*切分定理*的这条性质将会把加权图中的所有顶点分为两个集合、检查横跨两个集合的所有边并识别哪条边应属于图的最小生成树。

定义。图的一种**切分**是将图的所有顶点分为两个非空且不重复的两个集合。横切边是一条连接两个属于不同集合的顶点的边。

通常，我们通过指定一个顶点集并隐式地认为它的补集为另一个顶点集来指定一个切分。这样，一条横切边就是连接该集合的一个顶点和不在该集合中的另一个顶点的一条边。如图 4.3.4 所示，我们将切分中一个集合的顶点都画为了灰色，另一个集合的顶点则为白色。

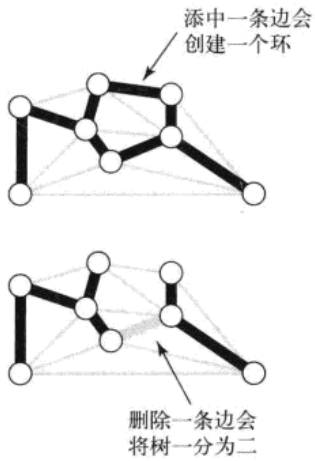


图 4.3.3 树的基本性质

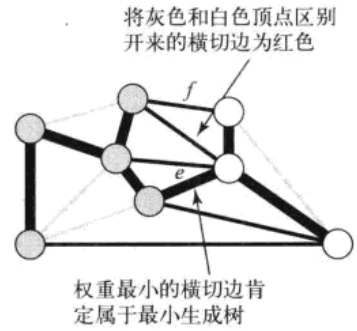


图 4.3.4 切分定理（另见彩插）

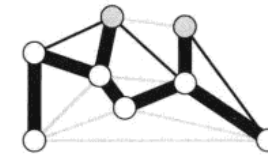


图 4.3.5 产生了两条属于最小生成树的横切边的一种切分

命题 J（切分定理）。在一幅加权图中，给定任意的切分，它的横切边中的权重最小者必然属于图的最小生成树。

证明。另 e 为权重最小的横切边，T 为图的最小生成树。我们采用反证法：假设 T 不包含 e。那么如果将 e 加入 T，得到的图必然含有一条经过 e 的环，且这个环至少含有另一条横切边——设为 f，f 的权重必然大于 e（因为 e 的权重是最小的且图中所有边的权重均不同）。那么我们删掉 f 而保留 e 就可以得到一棵权重更小的生成树。这和我们的假设 T 矛盾。

在假设所有的边的权重均不相同的前提下，每幅连通图都只有一棵唯一的最小生成树（请见练习 4.3.3），切分定理也表明了对于每一种切分，权重最小的横切边必然属于最小生成树。

图 4.3.4 是切分定理的示意图。注意，权重最小的横切边并不一定是所有横切边中唯一属于图的最小生成树的边。实际上，许多切分都会产生若干条属于最小生成树的横切边，如图 4.3.5 所示。

4.3.1.2 贪心算法

切分定理是解决最小生成树问题的所有算法的基础。更确切的说，这些算法都是一种贪心算法的特殊情况：使用切分定理找到最小生成树的一条边，不断重复直到找到最小生成树的所有边。这些算法相互之间的不同之处在于保存切分和判定权重最小的横切边的方式，但它们都是以下性质的特殊情况。

命题 K（最小生成树的贪心算法）。下面这种方法会将含有 V 个顶点的任意加权连通图中属于最小生成树的边标记为黑色：初始状态下所有边均为灰色，找到一种切分，它产生的横切边均不为黑色。将它权重最小的横切边标记为黑色。反复，直到标记了 V–1 条黑色边为止。

证明。为了简单，我们假设所有边的权重均不相同，尽管没有这个假设该命题同样成立（请见练习 4.3.5）。根据切分定理，所有被标记为黑色的边均属于最小生成树。如果黑色边的数量小于 V–1，必然还存在不会产生黑色横切边的切分（因为我们假设图是连通的）。只要找到了 V–1 条黑色的边，这些边所组成的就是一棵最小生成树。

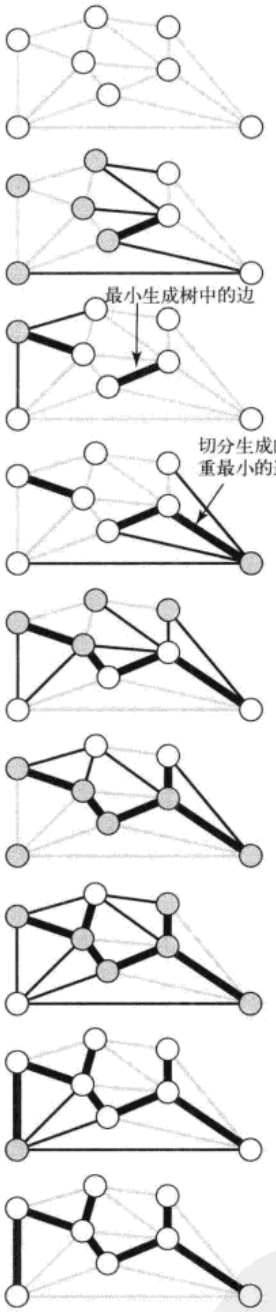


图 4.3.6 贪心最小生成树算法（另见彩插）

图 4.3.6 所示的是这个贪心算法运行的典型轨迹。每一幅图表现的都是一次切分，其中算法识别了一条权重最小的横切边（红色加粗）并将它加入最小生成树之中。

4.3.2 加权无向图的数据类型

加权无向图应该如何表示？也许最简单的方法就是扩展 4.1 节中对无向图的表示方法：在邻接矩阵的表示中，可以用边的权重代替布尔值来作为矩阵的元素；在邻接表的表示中，可以在链表的结点中增加一个权重域。（和以前一样，我们把重点放在稀疏图上，将邻接矩阵的表示方法留作练习。）这种经典的方法很有吸引力，但我们会使用另外一种并不太复杂的表示方式。它需要一个更加通用的 API 来处理 Edge 对象，能够使程序适用于更加常见的场景，请见表 4.3.2。

表 4.3.2 加权边的 API

public class **Edge** implements Comparable<Edge>		
	Edge(int v, int w, double weight)	用于初始化的构造函数
double	weight()	边的权重
int	either()	边两端的顶点之一
int	other(int v)	另一个顶点
int	compareTo(Edge that)	将这条边 e 与 that 比较
String	toString()	对象的字符串表示

访问边的端点的 either() 和 other() 方法乍一看会有些奇怪——在看到调用它们的代码时就会清楚了为什么会有这样的需要了。Edge 的实现请见框注“带权重的边的数据类型”，它是 EdgeWeightedGraph 的 API 的基础。加权无向图的实现很自然地使用了 Edge 对象，请见表 4.3.3。

表 4.3.3 加权无向图的 API

public class **EdgeWeightedGraph**		
	EdgeWeightedGraph(int V)	创建一幅含有 V 个顶点的空图
	EdgeWeightedGraph(In in)	从输入流中读取图
int	V()	图的顶点数
int	E()	图的边数
void	addEdge(Edge e)	向图中添加一条边 e
Iterable<Edge>	adj(int v)	和 v 相关联的所有边
Iterable<Edge>	edges()	图的所有边
String	toString()	对象的字符串表示

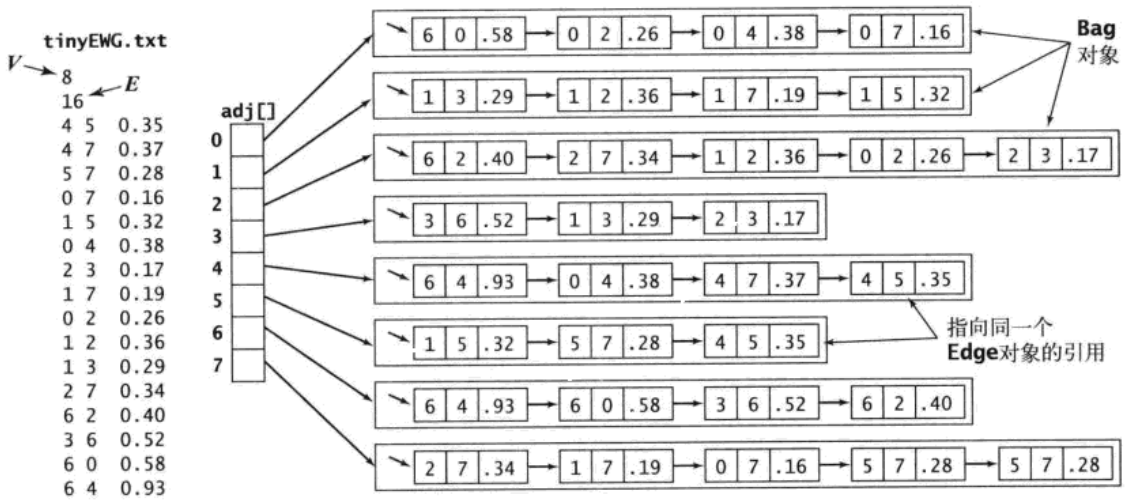
605 ≀ 608

这份 API 和 Graph 的 API（请见表 4.1.1）非常相似。两者的两个重要的不同之处在于本节 API 的基础是 Edge 且添加了一个 edges() 方法（请见框注“返回加权无向图中的所有边”）来遍历图的所有边（忽略自环）。后面框注“加权无向图的数据类型”中 EdgeWeightedGraph 的实现的其他部分与 4.1 节的无环图的实现基本相同，只是在邻接表中用 Edge 对象替代了 Graph 中的整数来作为链表的结点。

```
public Iterable<Edge> edges()
{
   Bag<Edge> b = new Bag<Edge>();
   for (int v = 0; v < V; v++)
      for (Edge e : adj[v])
         if (e.other(v) > v) b.add(e);
   return b;
}
```

返回加权无向图中的所有边

图 4.3.7 显示的是在处理样例文件 tinyEWG.txt 时用 EdgeWeightedGraph 对象表示的加权无向图。它按照 1.3 节中的标准实现显示了链表中每个 Bag 对象的内容。为了整洁，用一对 int 值和一个 double 值表示每个 Edge 对象。实际的数据结构是一个链表，其中每个元素都是一个指向含有这些值的对象的指针。需要特别注意的是，虽然每个 Edge 对象都有两个引用（每个顶点的链表中都有一个），但图中的每条边所对应的 Edge 对象只有一个。在示意图中，边在链表中的出现顺序和处理它们的顺序是相反的，这是由于标准链表实现和栈的相似性所导致的。和 Graph 一样，使用 Bag 对象可以保证用例的代码和链表中对象的顺序是无关的。



609

图 4.3.7 加权无向图的表示

带权重的边的数据类型

```
public class Edge implements Comparable<Edge>
{
   private final int v;                          // 顶点之一
   private final int w;                          // 另一个顶点
   private final double weight;                  // 边的权重

   public Edge(int v, int w, double weight)
   {
      this.v = v;
```

```
        this.w = w;
        this.weight = weight;
    }

    public double weight()
    {  return weight;  }

    public int either()
    {  return v;  }

    public int other(int vertex)
    {
       if      (vertex == v) return w;
       else if (vertex == w) return v;
       else throw new RuntimeException( "Inconsistent edge" );
    }

    public int compareTo(Edge that)
    {
       if      (this.weight() < that.weight()) return -1;
       else if (this.weight() > that.weight()) return +1;
       else                                    return  0;
    }

    public String toString()
    {  return String.format( "%d-%d %.2f" , v, w, weight);  }
}
```

该数据结构提供了 either() 和 other() 两个方法。在已知一个顶点 v 时，用例可以使用 other(v) 来得到边的另一个顶点。当两个顶点都是未知的时候，用例可以使用惯用代码 v=e.either(), w=e.other(v); 来访问一个 Edge 对象 e 的两个顶点。

610

加权无向图的数据类型

```
public class EdgeWeightedGraph
{
   private final int V;                  // 顶点总数
   private int E;                        // 边的总数
   private Bag<Edge>[] adj;              // 邻接表

   public EdgeWeightedGraph(int V)
   {
      this.V = V;
      this.E = 0;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Edge>();
   }

   public EdgeWeightedGraph(In in)
   // 见练习4.3.9

   public int V() {  return V;  }
   public int E() {  return E;  }

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
```

```
        adj[v].add(e);
        adj[w].add(e);
        E++;
      }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }

   public Iterable<Edge> edges()
   // 请见4.3.2节框注“返回加权无向图中的所有边”

}
```

该实现使用了一个由顶点索引的邻接表。与 Graph（请见 4.1.2.2 节框注“Graph 数据类型”）一样，每条边都会出现两次：如果一条边连接了顶点 v 和 w，那么它既会出现在 v 的链表中也会出现在 w 的链表中。edges() 方法将所有边放在一个 Bag 对象中（请见 4.3.2 节框注“返回加权无向图中的所有边”）。toString() 方法的实现留作练习。

611

4.3.2.1 用权重来比较边

API 说明 Edge 类必须实现 Comparable 接口并包含一个 compareTo() 方法。一幅加权无向图中的边的自然次序就是按权重排序，相应的 compareTo() 方法的实现也就很简单了。

4.3.2.2 平行边

和无环图的实现一样，这里也允许存在平行边。我们也可以用更复杂的方式实现 EdgeWeightedGraph 类来消除平行边，比如只保留平行的边中的权重最小者。

4.3.3.3 自环

允许存在自环。尽管自环可能的确存在于输入或是数据结构之中，但是 EdgeWeightedGraph 中 edge() 的实现并没有统计它们。这对最小生成树算法没有影响，因为最小生成树肯定不会含有自环。如果在应用中自环很重要，那你或许需要根据应用场景修改代码。

你会看到，有了 Edge 对象之后用例的代码就可以变得更加干净整洁。这也有个小小的代价：每个邻接表的结点都是一个指向 Edge 对象的引用，它们含有一些冗余的信息（v 的邻接链表中的每个结点都会用一个变量保存 v）。使用对象也会带来一些开销。虽然每条边的 Edge 对象都只有一个，但邻接表中还是会含有两个指向同一 Edge 对象的引用。另一种广泛使用的方案是与 Graph 一样，用两个结点对象来表示一条边，每个结点对象都会保存顶点的信息和边的权重。这种方法也是有代价的——需要两个结点，每条边的权重都会被保存两遍。

612

4.3.3 最小生成树的 API 和测试用例

按照惯例，在 API 中会定义一个接受加权无向图为参数的构造函数并且支持能够为用例返回图的最小生成树和其权重的方法。那么我们应该如何表示最小生成树呢？由于图 G 的最小生成树是 G 的一幅子图并且同时也是一棵树，因此我们有很多选择，最主要的几种表示方法为：

- 一组边的列表；
- 一幅加权无向图；
- 一个以顶点为索引且含有父结点链接的数组。

在为各种应用选择这些表示方法时，我们希望尽量给予最小生成树的实现以最大的灵活性，因此我们采用了表 4.3.4 所示的 API。

表 4.3.4 最小生成树的 API

public class MST		
	MST(EdgeWeightedGraph G)	构造函数
Iterable<Edge>	edges()	最小生成树的所有边
double	weight()	最小生成树的权重

4.3.3.1 测试用例

和以前一样，我们会创建样图并开发一个测试用例来测试最小生成树的实现。右侧框注就是一个示例。它从输入流中读取图的所有边并构造一幅加权无向图，然后计算该图的最小生成树并打印树的所有边和权重之和。

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   EdgeWeightedGraph G;
   G = new EdgeWeightedGraph(in);

   MST mst = new MST(G);
   for (Edge e : mst.edges())
      StdOut.println(e);
   StdOut.println(mst.weight());
}
```

最小生成树的测试用例

4.3.3.2 测试数据

你可以在本书的网站上找到 tinyEWG.txt 文件，它定义了我们用来展示最小生成树算法的轨迹样图（请见图 4.3.1）。在网站上你还能找到 mediumEWG.txt，它定义了一幅含有 250 个顶点的加权无向图，如图 4.3.8 所示。它也是一幅欧拉图的示例，它的顶点都是平面上的点，边为连接它们的线段且权重为两点之间的欧拉距离。这样的图有助于我们理解最小生成树算法的行为，同时也是我们提到过的许多典型实际问题的模型，例如公路地图和电路图。在本书的网站上你还能找到一幅较大的样图 largeEWG.txt，它是一幅含有一百万个顶点的欧拉图。我们的目标就是在合理的时间范围内通过计算得到这种规模的图的最小生成树。

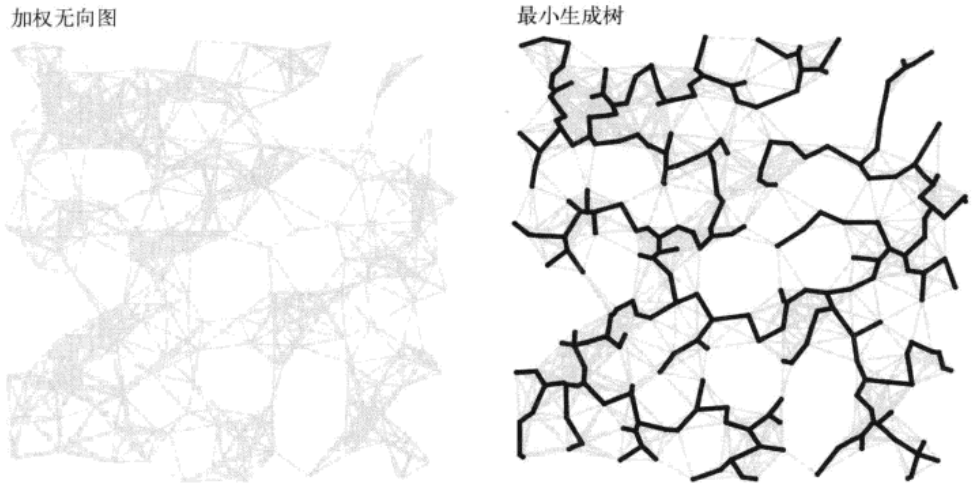
```
% more tinyEWG.txt
8 16
4 5 .35
4 7 .37
5 7 .28
0 7 .16
1 5 .32
0 4 .38
2 3 .17
1 7 .19
0 2 .26
1 2 .36
1 3 .29
2 7 .34
6 2 .40
3 6 .52
6 0 .58
6 4 .93

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

```
% more mediumEWG.txt
250 1273
244 246 0.11712
239 240 0.10616
238 245 0.06142
235 238 0.07048
233 240 0.07634
232 248 0.10223
231 248 0.10699
229 249 0.10098
228 241 0.01473
226 231 0.07638
... [还有1263条边]

% java MST mediumEWG.txt
  0 225 0.02383
 49 225 0.03314
 44  49 0.02107
 44 204 0.01774
 49  97 0.03121
202 204 0.04207
176 202 0.04299
176 191 0.02089
 68 176 0.04396
 58  68 0.04795
... [还有293条边]
10.46351
```

613



614
~
615

图 4.3.8 一幅含有 250 个顶点的无向加权欧拉图（共含有 1273 条边）和它的最小生成树

4.3.4 Prim 算法

我们要学习的第一种计算最小生成树的方法叫做 Prim 算法，它的每一步都会为一棵生长中的树添加一条边。一开始这棵树只有一个顶点，然后会向它添加 $V-1$ 条边，每次总是将下一条连接树中的顶点与不在树中的顶点且权重最小的边（黑色表示）加入树中（即由树中的顶点所定义的切分中的一条横切边），如图 4.3.9 所示。

命题 L。 Prim 算法能够得到任意加权无向图的最小生成树。

证明。 由命题 K 可知，这棵不断生长的树定义了一个切分且不存在黑色的横切边。该算法会选取权重最小的横切边并根据贪心算法不断将它们标记为黑色。

以上我们对 Prim 算法的简单描述没有回答一个关键的问题：如何才能（有效地）找到最小权重的横切边呢？人们提出了很多方法——在用一种特别简单的方法解决这个问题之后我们会讨论其中的一部分方法。

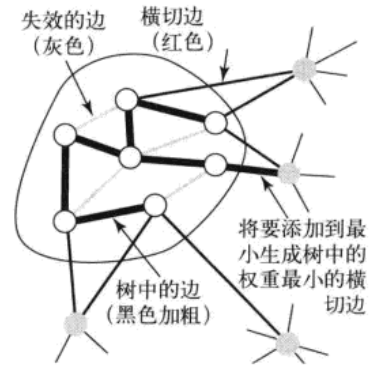


图 4.3.9 最小生成树的 Prim 算法（另见彩插）

4.3.4.1 数据结构

实现 Prim 算法需要用到一些简单常见的数据结构。具体来说，我们会用以下方法表示树中的顶点、边和横切边。

- 顶点。使用一个由顶点索引的布尔数组 `marked[]`，如果顶点 `v` 在树中，那么 `marked[v]` 的值为 `true`。
- 边。选择以下两种数据结构之一：一条队列 `mst` 来保存最小生成树中的边，或者一个由顶点索引的 `Edge` 对象的数组 `edgeTo[]`，其中 `edgeTo[v]` 为将 `v` 连接到树中的 `Edge` 对象。

- 横切边：使用一条优先队列 MinPQ<Edge> 来根据权重比较所有边（请见 4.3.2 节框注“带权重的边的数据类型”）。

有了这些数据结构我们就可以回答“哪条边的权重最小？”这个基本的问题了。

4.3.4.2 维护横切边的集合

每当我们向树中添加了一条边之后，也向树中添加了一个顶点。要维护一个包含所有横切边的集合，就要将连接这个顶点和其他所有不在树中的顶点的边加入优先队列（用 marked[] 来识别这样的边）。但还有一点：连接新加入树中的顶点与其他已经在树中顶点的所有边都失效了。（这样的边都已经不是横切边了，因为它的两个顶点都在树中。）Prim 算法的即时实现可以将这样的边从优先队列中删掉，但我们先来学习这个算法的一种延时实现，将这些边先留在优先队列中，等到要删除它们的时候再检查边的有效性。

图 4.3.10 是处理样图 tinyEWG.txt 的轨迹。每一张图片都是算法访问过一个顶点之后（被添加到树中，邻接链表中的边也已经被处理完成）图和优先队列的状态。优先队列的内容被按照顺序显示在一侧，树中的新顶点旁边有个星号。算法构造最小生成树的过程如下所述。

- 将顶点 0 添加到最小生成树之中，将它的邻接链表中的所有边添加到优先队列之中。
- 将顶点 7 和边 0-7 添加到最小生成树之中，将顶点的邻接链表中的所有边添加到优先队列之中。
- 将顶点 1 和边 1-7 添加到最小生成树之中，将顶点的邻接链表中的所有边添加到优先队列之中。
- 将顶点 2 和边 0-2 添加到最小生成树之中，将边 2-3 和 6-2 添加到优先队列之中。边 2-7 和 1-2 失效。
- 将顶点 3 和边 2-3 添加到最小生成树之中，将边 3-6 添加到优先队列之中。边 1-3 失效。
- 将顶点 5 和边 5-7 添加到最小生成树之中，将边 4-5 添加到优先队列之中。边 1-5 失效。

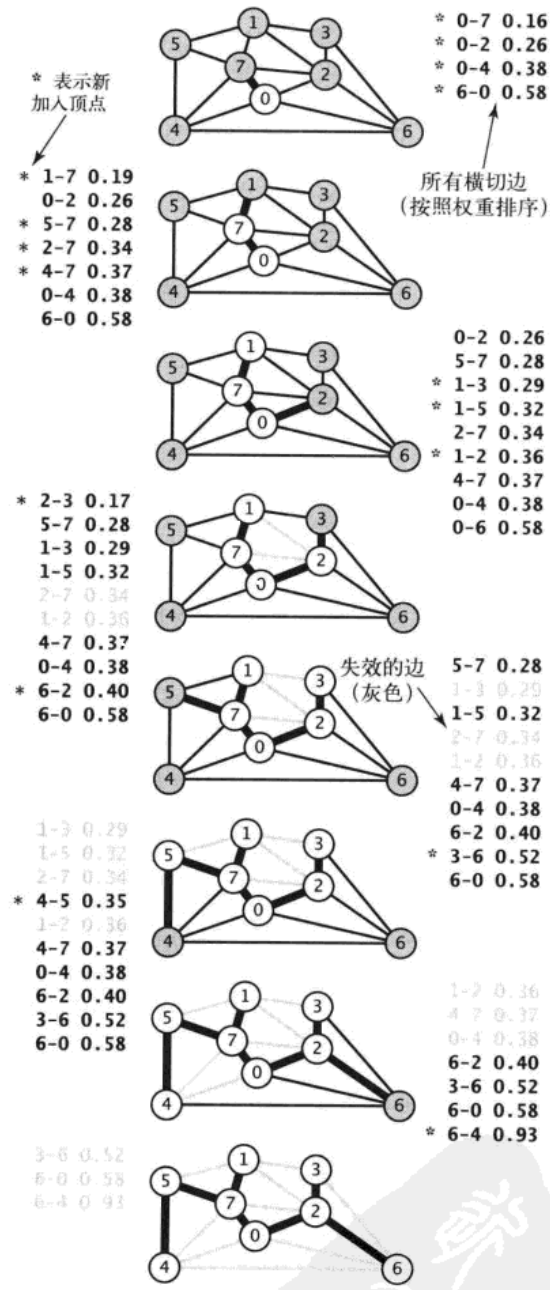


图 4.3.10 Prim 算法的轨迹（延时实现，另见彩插）

- 从优先队列中删除失效的边 1-3、1-5 和 2-7。
- 将顶点 4 和边 4-5 添加到最小生成树之中，将边 6-4 添加到优先队列之中。边 4-7 和 0-4 失效。
- 从优先队列中删除失效的边 1-2、4-7 和 0-4。
- 将顶点 6 和边 6-2 添加到最小生成树之中，和顶点 6 相关联的其他边均失效。

616 ~ 617

在添加了 V 个顶点（以及 V–1 条边）之后，最小生成树就完成了。优先队列中的余下的边都是无效的，不需要再去检查它们。

4.3.4.3 实现

有了这些预备知识，Prim 算法的实现就很简单了，请见后面框注“最小生成树的 Prim 算法的延时实现”中的 LazyPrimMST 类。和前两节实现深度优先搜索和广度优先搜索一样，实现会在构造函数中计算图的最小生成树，这样用例方法就可以用查询类方法获得最小生成树的各种属性。我们使用了一个私有方法 visit() 来为树添加一个顶点、将它标记为“已访问”并将与它关联的所有未失效的边加入优先队列，以保证队列含有所有连接树顶点和非树顶点的边（也可能含有一些已经失效的边）。代码的内循环是算法的具体实现：我们从优先队列中取出一条边并将它添加到树中（如果它还没有失效的话），再把这条边的另一个顶点也添加到树中，然后用新顶点作为参数调用 visit() 方法来更新横切边的集合。weight() 方法可以遍历树的所有边并得到它们的权重之和（延时实现）或是用一个运行时的变量统计总权重（即时实现），这一点留作练习 4.3.31。

4.3.4.4 运行时间

Prim 算法有多快？我们已经知道优先队列的性质，所以要回答这个问题并不困难。

命题 M。 Prim 算法的延时实现计算一幅含有 V 个顶点和 E 条边的连通加权无向图的最小生成树所需的空间与 E 成正比，所需的时间与 $E\log E$ 成正比（最坏情况）。

证明。 算法的瓶颈在于优先队列的 insert() 和 delMin() 方法中比较边的权重的次数。优先队列中最多可能有 E 条边，这就是空间需求的上限。在最坏情况下，一次插入的成本为 ~ $\lg E$，删除最小元素的成本为 ~ $2\lg E$（请见第 2 章的命题 O）。因为最多只能插入 E 条边，删除 E 次最小元素，时间上限显而易见。

在实际中，估计的运行时间上限是比较保守的，因为一般情况下优先队列中的边都远小于 E。这么困难的任务，解决方法却如此的简单、高效而实用，实在令人佩服。下面，我们会简要讨论一些改进算法的方法。和以前一样，在性能优先的应用场景中仔细评估这些改进的工作应该留给专家。

618

最小生成树的 Prim 算法的延时实现

```
public class LazyPrimMST
{
   private boolean[] marked;          // 最小生成树的顶点
   private Queue<Edge> mst;           // 最小生成树的边
   private MinPQ<Edge> pq;            // 横切边（包括失效的边）

   public LazyPrimMST(EdgeWeightedGraph G)
   {
      pq = new MinPQ<Edge>();
      marked = new boolean[G.V()];
      mst = new Queue<Edge>();

      visit(G, 0);   // 假设G是连通的（请见练习4.3.22）
      while (!pq.isEmpty())
```

```
        {
            Edge e = pq.delMin();                       // 从pq中得到权重最小的边
            int v = e.either(), w = e.other(v);         // 跳过失效的边
            if (marked[v] && marked[w]) continue;
            mst.enqueue(e);                             // 将边添加到树中
            if (!marked[v]) visit(G, v);                // 将顶点（v或w）添加到树中
            if (!marked[w]) visit(G, w);
        }
    }

    private void visit(EdgeWeightedGraph G, int v)
    {  // 标记顶点v并将所有连接v和未被标记顶点的边加入pq
       marked[v] = true;
       for (Edge e : G.adj(v))
          if (!marked[e.other(v)]) pq.insert(e);
    }

    public Iterable<Edge> edges()
    {  return mst;  }

    public double weight()    // 请见练习4.3.31

}
```

Prim 算法的这种实现使用了一条优先队列来保存所有的横切边、一个由顶点索引的队列来标记树的顶点以及一条队列来保存最小生成树的边。这种延时实现会在优先队列中保留失效的边。

619

4.3.5 Prim 算法的即时实现

要改进 LazyPrimMST，可以尝试从优先队列中删除失效的边，这样优先队列就只含有树顶点和非树顶点之间的横切边，但其实还可以删除更多的边。关键在于，我们感兴趣的只是连接树顶点和非树顶点中权重最小的边。当我们将顶点 v 添加到树中时，对于每个非树顶点 w 产生的变化只可能使得 w 到最小生成树的距离更近了，如图 4.3.11 所示。简而言之，我们不需要在优先队列中保存所有从 w 到树顶点的边——而只需要保存其中权重最小的那条，在将 v 添加到树中后检查是否需要更新这条权重最小的边（因为 v-w 的权重可能更小）。我们只需遍历 v 的邻接链表就可以完成这个任务。换句话说，我们只会在优先队列中保存每个非树顶点 w 的一条边：将它与树中的顶点连接起来的权重最小的那条边。将 w 和树的顶点连接起来的其他权重较大的边迟早都会失效，所以没必要在优先队列中保存它们。

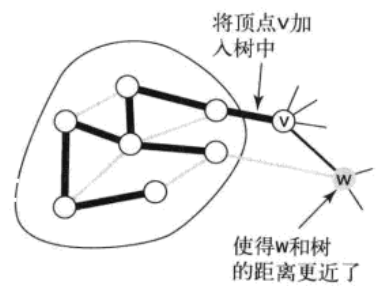


图 4.3.11 Prim 算法的即时实现

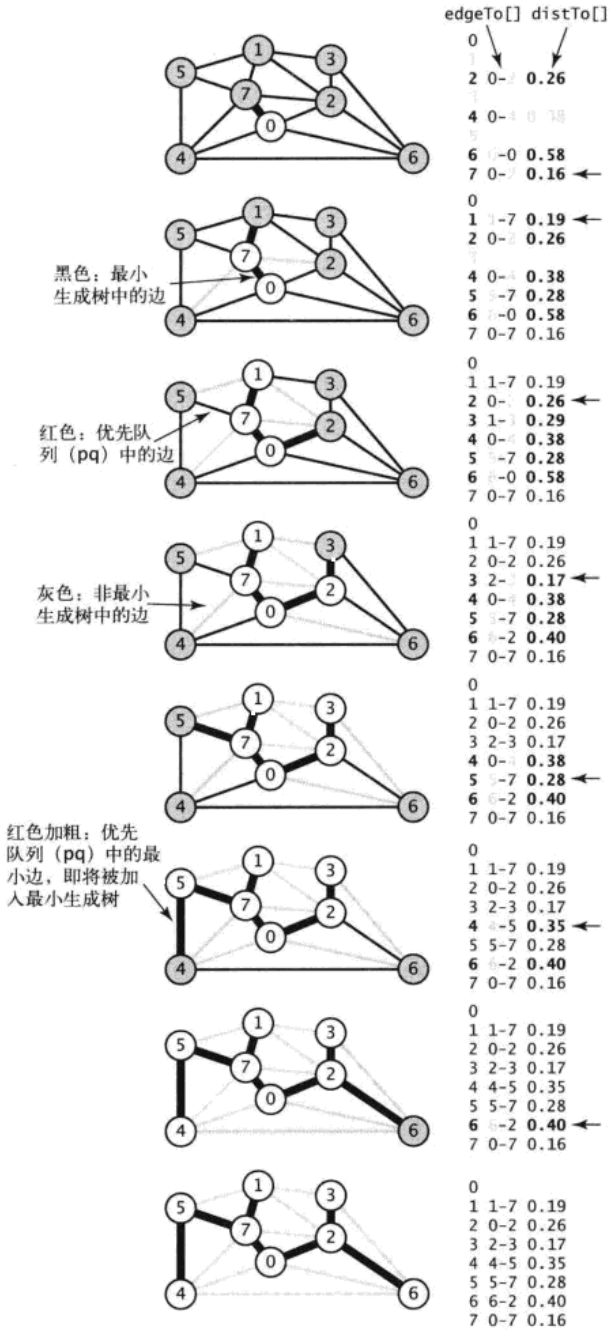
PrimMST 类（请见算法 4.7）使用了 2.4 节中介绍的索引优先队列实现的 Prim 算法。它将 LazyPrimMST 中的 marked[] 和 mst[] 替换为两个顶点索引的数组 edgeTo[] 和 distTo[]，它们具有如下性质。

- 如果顶点 v 不在树中但至少含有一条边和树相连，那么 edgeTo[v] 是将 v 和树连接的最短边，distTo[v] 为这条边的权重。
- 所有这类顶点 v 都保存在一条索引优先队列中，索引 v 关联的值是 edgeTo[v] 的边的权重。

这些性质的关键在于优先队列中的最小键即是权重最小的横切边的权重，而和它相关联的顶点 v 就是下一个将被添加到树中的顶点。marked[] 数组已经没有必要了，因为判断条件 !marked[w] 等价于 distTo[w] 是无穷的（且 edgeTo[w] 为 null）。要维护这些数据结构，

PrimMST 会从优先队列中取出一条边 v 并检查它的邻接链表中的每条边 v-w。如果 w 已经被标记过，那么这条边就已经失效了；如果 w 不在优先队列中或者 v-w 的权重小于目前已知的最小值 edgeTo[w]，代码会更新数组，将 v-w 作为将 v 和树连接的最佳选择。

图 4.3.12 所示的是 PrimMST 在处理样图 tinyEWG.txt 过程中的轨迹。将每个顶点加入最小生成树之后，edgeTo[] 和 distTo[] 的内容显示在右侧，不同的颜色显示了最小生成树中的顶点（索引为黑色）、非最小生成树的顶点（索引为灰色）、最小生成树的边（黑色）和优先队列中的索引值对（红色）。在示意图中，将每个非最小生成树顶点连接到树的最短边为红色。该算法向最小生成树中添加的边的顺序和延时版本相同，不同之处在于优先队列的操作。它构造最小生成树的过程如下所述。



4.3.12 Prim 算法的轨迹（即时版本，另见彩插）

- 将顶点 0 添加到最小生成树之中，将它的邻接链表中的所有边添加到优先队列之中，因为这些边都是目前（唯一）已知的连接非树顶点和树顶点的最短边。
- 将顶点 7 和边 0-7 添加到最小生成树之中，将边 1-7 和 5-7 添加到优先队列之中。边 4-7 和 2-7 不会影响到优先队列，因为它们的权重分别都大于连接顶点 2 和 4 与最小生成树的最小边。
- 将顶点 1 和边 1-7 添加到最小生成树之中，将边 1-3 添加到优先队列之中。
- 将顶点 2 和边 0-2 添加到最小生成树之中，将连接顶点 6 与树的最小边由 0-6 替换为 2-6，将连接顶点 3 与树的最小边由 1-3 替换为 2-3。
- 将顶点 3 和边 2-3 添加到最小生成树之中。
- 将顶点 5 和边 5-7 添加到最小生成树之中，将连接顶点 4 与树的最小边由 0-4 替换为 4-5。
- 将顶点 4 和边 4-5 添加到最小生成树之中。

❑ 将顶点 6 和边 6-2 添加到最小生成树之中。

添加了 $V-1$ 条边之后，最小生成树完成且优先队列为空。

620 | 621

算法 4.7 最小生成树的 Prim 算法（即时版本）

```
public class PrimMST
{
   private Edge[] edgeTo;          // 距离树最近的边
   private double[] distTo;        // distTo[w]=edgeTo[w].weight()
   private boolean[] marked;       // 如果v在树中则为true
   private IndexMinPQ<Double> pq;  // 有效的横切边

   public PrimMST(EdgeWeightedGraph G)
   {
      edgeTo = new Edge[G.V()];
      distTo = new double[G.V()];
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      pq = new IndexMinPQ<Double>(G.V());

      distTo[0] = 0.0;
      pq.insert(0, 0.0);              // 用顶点0和权重0初始化pq
      while (!pq.isEmpty())
         visit(G, pq.delMin());       // 将最近的顶点添加到树中
   }

   private void visit(EdgeWeightedGraph G, int v)
   {  // 将顶点v添加到树中，更新数据
      marked[v] = true;
      for (Edge e : G.adj(v))

      {
         int w = e.other(v);

         if (marked[w]) continue;     // v-w失效
         if (e.weight() < distTo[w])

         {  // 连接w和树的最佳边Edge变为e
            edgeTo[w] = e;

            distTo[w] = e.weight();
            if (pq.contains(w)) pq.change(w, distTo[w]);
            else                pq.insert(w, distTo[w]);

         }
      }
   }

   public Iterable<Edge> edges()    // 请见练习4.3.21
   public double weight()           // 请见练习4.3.31
}
```

这份 Prim 算法的实现将所有有效的横切边保存在了一条索引优先队列中。

622

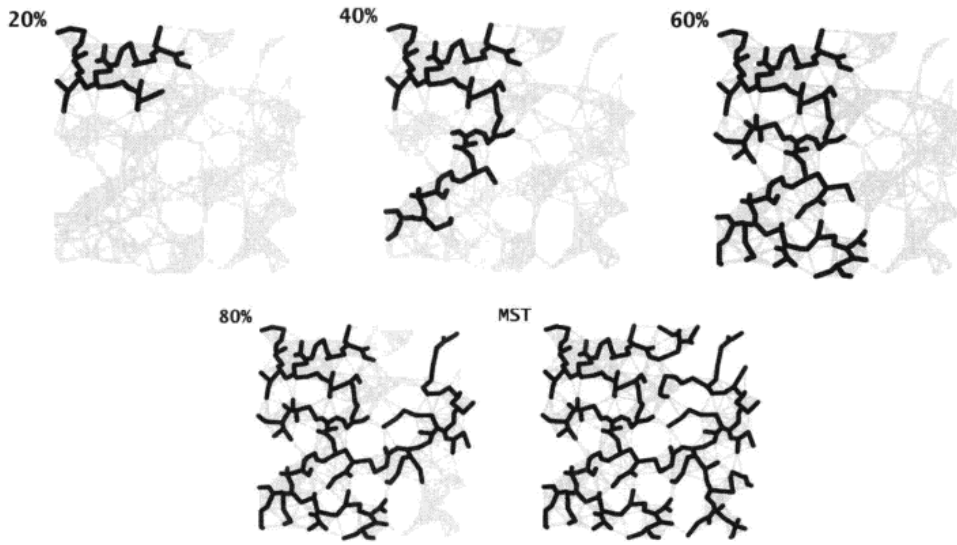
该算法的证明与命题 M 的证明本质上相同，Prim 算法的即时版本可以找到一幅连通的加权无向图的最小生成树，所需时间和 $E\log V$ 成正比，空间和 V 成正比（请见命题 N）。对于实际应用中经常出现的巨型稀疏图，两者在时间上限上没有什么区别（因为对于稀疏图来说是 $\lg E \sim \lg V$），

但空间上限变为了原来的一个常数因子（但很显著）。在性能优先的应用场景中，更加深入的分析和实验最好还是留给专家吧，因为相关的因素有很多，例如 MinPQ 和 IndexPQ 的实现、图的表示方法、应用场景所使用的图模型等。按照惯例，我们需要仔细研究这些改进，因为只有当这种常数因子的性能改进非常必要时，它所带来的代码复杂性才是值得的。在复杂的现代系统中有时这样做甚至会得不偿失。

命题 N。 Prim 算法的即时实现计算一幅含有 V 个顶点和 E 条边的连通加权无向图的最小生成树所需的空间和 V 成正比，所需的时间和 $E\log V$ 成正比（最坏情况）。

证明。 因为优先队列中的边数最多为 V，且使用了三条由顶点索引的数组，所以所需空间的上限和 V 成正比。算法会进行 V 次插入操作，V 次删除最小元素的操作和（在最坏情况下）E 次改变优先级的操作。已知在基于堆实现中的索引优先队列中所有这些操作的增长数量级为 $\log V$（请见第 2 章命题 Q），所以将所有这些加起来可知算法所需时间和 $E\log V$ 成正比。

图 4.3.13 展示了 Prim 算法是如何处理含有 250 个顶点的欧拉图 mediumEWG.txt 的。这是一个很有意思的动态过程（请见练习 4.3.27）。大多数情况下，树的生长都是通过连接一个和新加入的顶点相邻的顶点。当新加入的顶点周围没有非树顶点时，树的生长又会从另一部分开始。



623

图 4.3.13 Prim 算法（250 个顶点）

4.3.6 Kruskal 算法

我们要仔细学习的第二种最小生成树算法的主要思想是按照边的权重顺序（从小到大）处理它们，将边加入最小生成树中（图中的黑色边），加入的边不会与已经加入的边构成环，直到树中含有 $V-1$ 条边为止。这些黑色的边逐渐由一片森林合并为一棵树，也就是图的最小生成树。这种计算方法被称为 Kruskal 算法。

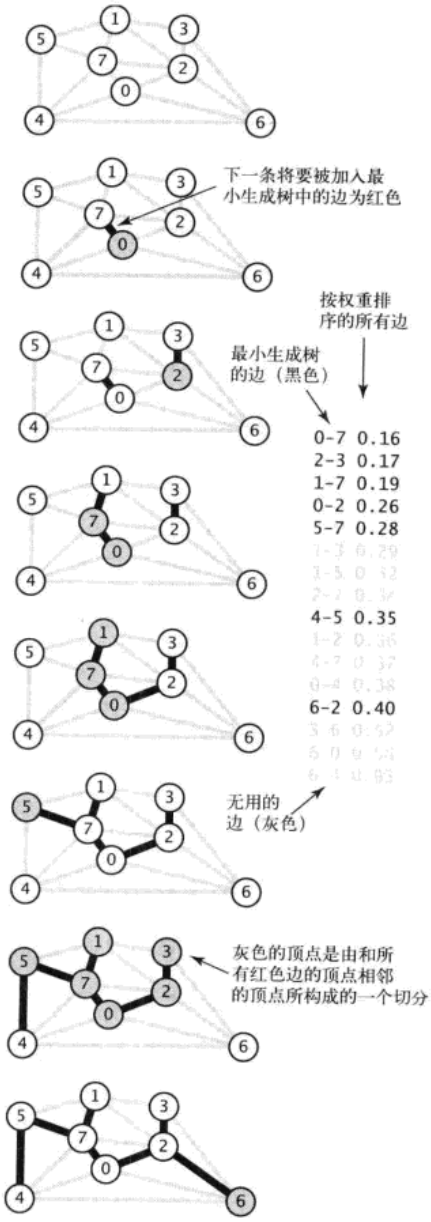


图 4.3.14 Kruskal 算法的轨迹（另见彩插）

命题 O。Kruskal 算法能够计算任意加权无向图的最小生成树。

证明。由命题 K 可知，如果下一条将被加入最小生成树中的边不会和已有的黑色边构成环，那么它就跨越了由所有和树顶点相邻的顶点组成的集合以及它们的补集所构成的一个切分。因为加入的这条边不会形成环、它是目前已知的唯一一条横切边且是按照权重顺序选择的边，所以它必然是权重最小的横切边。因此，该算法能够连续选择权重最小的横切边，和贪心算法一致。

Prim 算法是一条边一条边地来构造最小生成树，每一步都为一棵树添加一条边。Kruskal 算法构造最小生成树的时候也是一条边一条边地构造，但不同的是它寻找的边会连接一片森林中的两棵树。我们从一片由 V 棵单顶点的树构成的森林开始并不断将两棵树合并（用可以找到的最短边）直到只剩下一棵树，它就是最小生成树。

图 4.3.14 显示的是 Kruskal 算法处理 tinyEWG.txt 时的每一个步骤。首先，权重最小的条边都被加入到了最小生成树中，之后算法判断出 1-3、1-5 和 2-7 已经失效并将 4-5 加入最小生成树。最后 1-2、4-7 和 0-4 失效，6-2 被加入最小生成树。

有了本书中我们已经学习过的许多工具，Kruskal 算法的实现并不困难：我们将会使用一条优先队列（请见 2.4 节）来将边按照权重排序，用一个 union-find 数据结构（请见 1.5 节）来识别会形成环的边，以及一条队列（请见 1.3 节）来保存最小生成树的所有边。算法 4.8 实现了以上设想。注意，使用队列来保存最小生成树的所有边意味着用例在遍历时将会按照权重的升序得到这些边。`weight()` 方法需要遍历所有边来取得权重之和（或是使用一个变量动态统计权重之和），它的实现留作练习（请见练习 4.3.31）。

分析 Kruskal 算法所需的运行时间很简单，因为我们已经知道它的操作所需的时间。

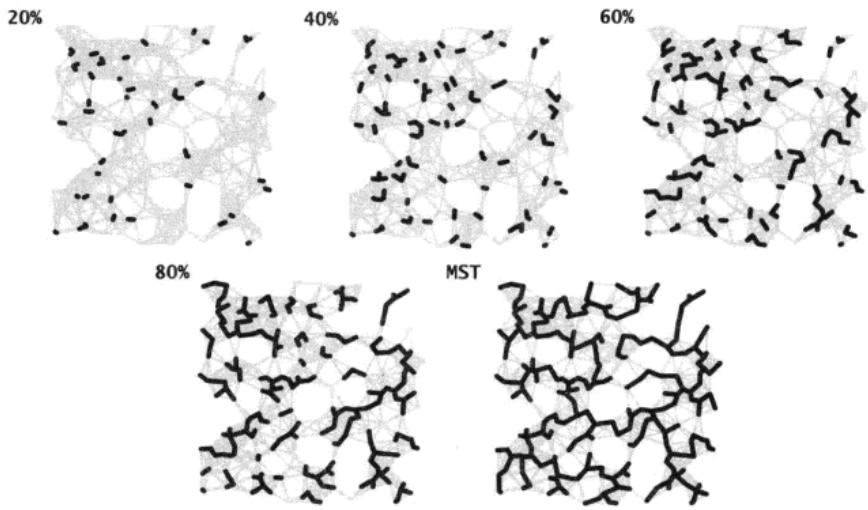
命题 N（续）。Kruskal 算法的计算一幅含有 V 个顶点和 E 条边的连通加权无向图的最小生成树所需的空间和 E 成正比，所需的时间和 $E\log E$ 成正比（最坏情况）。

证明。算法的实现在构造函数中使用所有边初始化优先队列，成本最多为 E 次比较（请见2.4节）。优先队列构造完成后，其余的部分和Prim算法完全相同。优先队列中最多可能含有 E 条边，即所需空间的上限。每次操作的成本最多为 $2\lg E$ 次比较，这就是时间上限的由来。Kruskal算法最多还会进行 E 次 `Connected()` 和 V 次 `union()` 操作，但这些成本相比 $E\log E$ 的总时间的增长数量级可以忽略不计（请见1.5节）。

与Prim算法一样，这个估计是比较保守的，因为算法在找到 $V-1$ 条边之后就会终止。实际的成本应该与 $E+E_0\log E$ 成正比，其中 E_0 是权重小于最小生成树中权重最大的边的所有边的总数。尽管拥有这个优势，Kruskal算法一般还是比Prim算法要慢，因为在处理每条边时除了两种算法都要完成的优先队列操作之外，它还需要进行一次 `connect()` 操作（请见练习4.3.39）。

624 ~ 625

图4.3.15所示为Kruskal算法在处理较大的样图mediumEWG.txt时的动态情况。很显然，边是按照权重顺序被添加到森林中的。



626

图4.3.15 Kruskal算法（250个顶点）

算法4.8 最小生成树的Kruskal算法

```
public class KruskalMST
{
   private Queue<Edge> mst;

   public KruskalMST(EdgeWeightedGraph G)
   {
      mst = new Queue<Edge>();
      MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
      UF uf = new UF(G.V());

      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();                  // 从pq得到权重最小的边和它的顶点
         int v = e.either(), w = e.other(v);
         if (uf.connected(v, w)) continue;      // 忽略失效的边
```

```
            uf.union(v, w);                    // 合并分量
            mst.enqueue(e);                    // 将边添加到最小生成树中
        }
    }

    public Iterable<Edge> edges()
    {  return mst;  }

    public double weight()          // 请见练习4.3.31

}
```

这份 Kruskal 算法的实现使用了一条队列来保存最小生成树中的所有边、一条优先队列来保存还未被检查的边和一个 union-find 的数据结构来判断无效的边。最小生成树的所有边会按照权重的升序返回给用例。`weight()` 方法的实现留作练习。

```
% java KruskalMST tinyEWG.txt
0-7 0.16
2-3 0.17
1-7 0.19
0-2 0.26
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

627

4.3.7 展望

最小生成树问题是本书中的被研究的最多的几个问题之一。解决这个问题的基本方法在现代数据结构和算法性能分析手段的发明之前就已经问世了。在当时，计算一幅含有上千条边的图的最小生成树还是一项令人望而生畏的任务。我们学习的最小生成树算法和这些老式方法的不同之处主要在于运用了现代的数据结构来完成一些基本的操作，这（再加上现代的计算能力）使得我们可以计算含有上百万甚至数十亿条边的图的最小生成树。

4.3.7.1 历史资料

计算稠密图的最小生成树算法（请见练习 4.3.29）最早是由 R.Prim 在 1961 年发明的，随后 E.W.Dijkstra 也独自发明了它。尽管 Dijkstra 的描述更为通用，但这个算法通常被称为 Prim 算法。其实算法的基本思想是 V.Jarnik 在 1939 年发明的，所以一些人也将这种方法称为 Jarnik 算法并认为 Prim 的（或是 Dijkstra）的贡献在于为稠密图找到了高效的实现算法。在 20 世纪 70 年代优先队列发明之后，它直接被应用在了寻找稀疏图中的最小生成树上。计算稀疏图中的最小生成树所需的时间和 $E\log E$ 成正比很快广为人知且并没有将此归功于任何一位研究者。在 1984 年，M.L.Fredman 和 R.E.Tarjan 发明了数据结构斐波纳契堆，将 Prim 算法所需的运行时间在理论上改进到了 $E+V\log V$。J.Kruskal 在 1956 年就发表了他的算法，但同样，相关的抽象数据结构在很多年中都没有被仔细研究。有趣的是，Kruskal 的论文中提到了 Prim 算法的一个变种，而 O.Boruvka 在 1926 年（！）的论文中就已经提到了这两种不同的方法。Boruvka 的论文要解决的是一个电力分配的问题并介绍了另外一种用现代数据结构可以轻易实现的方法（请见练习 4.3.43 和练习 4.3.44）。M.Sollin 在 1961 年重新发现了这个方法。该方法随后引起了其他人的注意并成为实现较好的渐进性能的最小生成树算法和平行最小生成树算法的基础。各种最小生成树算法的特点请见表 4.3.5。

表 4.3.5 各种最小生成树算法的性能特点

算 法	V 个顶点 E 条边，最坏情况下的增长数量级	
	空 间	时 间
延时的 Prim 算法	E	$E\log E$
即时的 Prim 算法	V	$E\log V$
Kruskal	E	$E\log E$
Fredman-Tarjan	V	$E+V\log V$
Chazelle	V	非常接近但还没有达到 E
628 理想情况	V	E?

4.3.7.2 线性的最小生成树算法？

一方面，目前还没有理论能够证明，不存在能在线性时间内得到任意图的最小生成树的算法。另一方面，发明能够在线性时间内计算稀疏图的最小生成树的算法仍然没有进展。自从 20 世纪 70 年代将 union-find 数据结构应用于 Kruskal 算法以及将优先队列应用于 Prim 算法之后，更好的实现这些抽象数据结构就成了许多研究者的主要目标。许多研究者都将寻找高效的优先队列的实现作为找到稀疏图的高效的最小生成树算法的关键，而其他一些人则研究了 Boruvka 算法的一些变种并将它们作为近似于线性级别的稀疏图的最小生成树算法的基础。这些研究仍然有希望最终为我们带来一个实用的线性最小生成树算法，它们甚至已经显示了一个线性时间的随机化算法的存在性。研究者距离线性时间的目标已经很近了：B.Chazelle 在 1997 年发表了一个算法，它在实际应用中和线性时间的算法的差距已经小到了无法区别的程度（尽管可以证明它并不是线性的），但它非常复杂以至于无法实用。尽管此类研究得到的算法大都十分复杂，其中一些的简化版也许可以进入实际应用。同时，在大多数应用场景中，我们都可以使用已经学过的基本方法在线性时间内得到图的最小生成树，只是对于一些稀疏图所需的时间要乘以 $\log V$。

总的来说，我们可以认为在实际应用中最小生成树问题已经被“解决”了。对于大多数的图来说，找到它的最小生成树的成本只比遍历图的所有边稍高一点。除了极为稀疏的图，这一点都能成立，但即使是在这种情况下，使用最好的算法所能得到的性能提升也不过是一个很小的常数因子，可能最多 10 倍。人们已经在许多图的模型中证明了这些结论，而很多实践者则已经使用 Prim 算法
629 和 Kruskal 算法计算大型图中的最小生成树数十年之久了。

答疑

问 Prim 和 Kruskal 算法能够处理有向图吗？

630 答 不行，不可能。那是一个更加困难的有向图处理问题，叫做最小树形图问题。

练习

4.3.1 证明可以将图中的所有边的权重都加上一个正常数或是都乘以一个正常数，图的最小生成树不会受到影响。

4.3.2 画出图 4.3.16 中的所有最小生成树（所有边的权重均相等）。

4.3.3 证明当图中所有边的权重均不相同时图的最小生成树是唯一的。

4.3.4 证明或给出反例：仅当加权无向图中所有边的权重均不相同时图的最小生成树是唯一的。

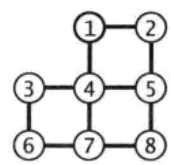


图 4.3.16

4.3.5 证明即使存在权重相同的边贪心算法仍然有效。

4.3.6 从 tinyEWG.txt 中（请见图 4.3.1）删去顶点 7 并给出加权图的最小生成树。

4.3.7 如何得到一幅加权图的最大生成树？

4.3.8 证明环的性质：任取一幅加权图中的一个环（边的权重各不相同），环中权重最大的边必然不属于图的最小生成树。

4.3.9 根据 Graph 中的构造函数（请见 4.1.2.2 框注“Graph 数据类型”）为 EdgeWeighted Graph 实现一个相应构造函数，从输入流中读取一幅图。

4.3.10 为稠密图实现 EdgeWeightedGraph，使用邻接矩阵（存储权重的二维数组），不允许存在平行边。

4.3.11 使用 1.4 节中的内存使用模型评估用 EdgeWeightedGraph 表示一幅含有 V 个顶点和 E 条边的图所需的内存。

4.3.12 假设加权图中的所有边的权重都不相同，其中权重最小的边一定属于图的最小生成树吗？权重最大的边可能属于图的最小生成树吗？任意环中的权重最小边都属于图的最小生成树吗？证明你的每个回答或者给出相应的反例。

4.3.13 给出一个反例证明以下策略不一定能够找到图的最小生成树：首先以任意顶点作为图的最小生成树，然后向树中添加 V–1 条边，每次总是添加依附于最近加入最小生成树的顶点的所有边中的权重最小者。 631

4.3.14 给定一幅加权图 G 以及它的最小生成树。从 G 中删去一条边且 G 仍然是连通的，如何在与 E 成正比的时间内找到新图的最小生成树。

4.3.15 给定一幅加权图 G 以及它的最小生成树。向 G 中添加一条边 e，如何在与 V 成正比的时间内找到新图的最小生成树。

4.3.16 给定一幅加权图 G 以及它的最小生成树。向 G 中添加一条边 e，编写一段程序找到 e 的权重在什么范围之内才会被加入最小生成树。

4.3.17 为 EdgeWeightedGraph 类实现 toString() 方法。

4.3.18 给出使用延时 Prim 算法、即时 Prim 算法和 Kruskal 算法在计算练习 4.3.6 中的图的最小生成树过程中的轨迹。

4.3.19 假设你使用的优先队列的实现会维护一条有序链表。在最坏情况下，用 Prim 算法和 Kruskal 算法处理一幅含有 V 个顶点和 E 条边的加权图的时间增长数量级是多少？这种方法适用于什么情况？证明你的结论。

4.3.20 真假判断：在 Kruskal 算法的执行过程中，最小生成树中的每个顶点到它的子树中的某个顶点的距离比到非子树中的任意顶点都近。证明你的结论。

4.3.21 为 PrimMST 类（请见算法 4.7）实现 edges() 方法。

解答：

```
public Iterable<Edge> edges()
{
   Bag<Edge> mst = new Bag<Edge>();
   for (int v = 1; v < edgeTo.length; v++)
      mst.add(edgeTo[v]);
   return mst;
}
```

632

提高题

4.3.22 *最小生成森林*。开发新版本的 Prim 算法和 Kruskal 算法来计算一幅加权图的最小生成森林，图不一定是连通的。使用 4.1 节中连通分量的 API 并找到每个连通分量的最小生成树。

4.3.23 *Vyssotsky 算法*。开发一种不断使用环的性质（请见练习 4.3.8）来计算最小生成树的算法：每次将一条边添加到假设的最小生成树中，如果形成了一个环则删去环中权重最大的边。*注意*：这个算法不如我们学过的几种方法引人注意，因为很难找到一种数据结构能够有效支持“删除环中权重最大的边”的操作。

4.3.24 *逆向删除算法*。实现以下计算最小生成树的算法：开始时图含有原图的所有边，然后按照权重大小的降序排列遍历所有的边。对于每条边，如果删除它图仍然是连通的，那就删掉它。证明这种方法可以得到图的最小生成树。实现中加权边的比较次数增长的数量级是多少？

4.3.25 *最坏情况生成器*。开发一个加权图生成器，图中含有 V 个顶点和 E 条边，使得延时的 Prim 算法所需的运行时间是非线性的。对于即时的 Prim 算法回答相同的问题。

4.3.26 *关键边*。关键边指的是图的最小生成树中的某一条边，如果删除它，新图的最小生成树的总权重将会大于原最小生成树的总权重。找到在 $E\log E$ 时间内找出图的关键边的算法。注意：这个问题中边的权重并不一定各不相同（否则最小生成树中的所有边都是关键边）。

4.3.27 *动画*。编写一段程序将最小生成树算法用动画表现出来。用程序处理 mediumEWG.txt 来产生类似于图 4.3.12 和图 4.3.14 的示意图。

4.3.28 *空间最优的数据结构*。实现另一个版本的延时 Prim 算法，在 `EdgeWeightedGraph` 和 `MinPQ` 中使用低级数据结构代替 `Bag` 和 `Edge` 来节省空间。根据 1.4 节中的内存使用模型用一个 V 和 E 的
633 函数评估节省的内存总量（参考练习 4.3.11）。

4.3.29 *稠密图*。实现另一个版本的 Prim 算法，即时（但不使用优先队列）且能够在 V^2 次加权边比较之内得到最小生成树。

4.3.30 *欧拉加权图*。修改你为练习 4.1.37 给出的解答，为平面图创建一份 API——`Euclidean EdgeWeightedGraph`，这样你就能够处理用图形表示的图了。

4.3.31 *最小生成树的权重*。为 `LazyPrimMST`、`PrimMST` 和 `KruskalMST` 实现 `weight()` 方法，使用*延时*策略，只在被调用时才遍历最小生成树的所有边来计算总权重。然后用*即时*策略再次实现这个方法，在计算最小生成树的过程中维护一个动态的总权重。

4.3.32 *指定的集合*。给定一幅连通的加权图 G 和一个边的集合 S（不含环），给出一种算法得到含有 S 中的所有边的最小加权生成树。

4.3.33 *验证*。编写一个使用最小生成树算法以及 `EdgeWeightedGraph` 类的方法 `check()`，使用以下根据命题 J 得到的*最优切分条件*来验证给定的一组边就是一棵最小生成树：如果给定的一组边是一棵最小生成树，且删除树中的任意边得到的切分中权重最小的横切边正是被删除的那条边，则
634 这最小生成一组边就是图的最小生成树。你的方法的运行时间的增长数量级是多少？

实验题

4.3.34 *随机稀疏加权图*。基于你为练习 4.1.41 给出的解答编写一个随机稀疏加权图生成器。在赋予边的权重时，定义一个随机加权有向图的抽象数据结构并给出两种实现：一种按均匀分布生成权重，另一种按高斯分布生成权重。编写用例程序，用两种权重分布和一组精心挑选过的 V 和 E 的值

生成随机的稀疏加权图，使得我们可以用它对权重的各种分布进行有意义的经验性测试。

4.3.35 随机欧拉加权图。修改你为练习 4.1.42 给出的解答，将每条边的权重设为顶点之间的距离。

4.3.36 随机网格加权图。修改你为练习 4.1.43 给出的解答，将每条边的权重设为 0 到 1 之间的随机值。

4.3.37 真实世界中的加权图。从网上找出一幅巨型加权无向图——可以是标注了距离的地图，或是标明了资费的电话黄页，或是航线的价目表。编写一段程序 RandomRealEdgeWeightedGraph，从这些顶点构成的子图中随机选取 V 个顶点，然后再从这些顶点构成的子图中随机选取 E 条边来构造一幅图。

测试所有的算法并研究所有图的模型的所有参数是不现实的。请为下面的每一道题都编写一段程序来处理从输入得到的任意图。这段程序可以调用上面的任意生成器并对相应的图模型进行实验。你可以根据上次实验的结果自己作出判断来选择不同实验。陈述结果以及由此得出的任何结论。

4.3.38 延时的代价。对于各种图的模型，运行实验并根据经验比较 Prim 算法的延时版本和即时版本的性能差异。

4.3.39 对比 Prim 算法与 Kruskal 算法。运行实验并根据经验比较 Prim 算法的延时版本和即时版本与 Kruskal 算法的性能差异。

4.3.40 减少开销。运行实验并根据经验判断练习 4.3.28 中在 EdgeWeightedGraph 类中使用原始数据类型代替 Edge 所带来的效果。 635

4.3.41 最小生成树中的最长边。运行实验并根据经验分析最小生成树中最长边的长度以及图中不长于该边的边的总数。

4.3.42 切分。根据快速排序的切分思想（而非使用优先队列）实现一种新方法，检查 Kruskal 算法中的当前边是否属于最小生成树。

4.3.43 Boruvka 算法。实现 Boruvka 算法：和 Kruskal 算法类似，只是分阶段地向一组森林中逐渐添加边来构造一棵最小生成树。在每个阶段中，找出所有连接两棵不同的树的权重最小的边，并将它们全部加入最小生成树。为了避免出现环，假设所有边的权重均不相同。提示：维护一个由顶点索引的数组来辨别连接每棵树和它最近的邻居的边。记得用上 union-find 数据结构。

4.3.44 改进的 Boruvka 算法。给出 Boruvka 算法的另一种实现，用双向环形链表表示最小生成树的子树，使得子树可以被合并或改名，每个阶段所需的时间与 E 成正比（这样就不需要 union-find 数据结构了）。

4.3.45 外部最小生成树。如果一幅图非常大，内存最多只能存储 V 条边，如何计算它的最小生成树？

4.3.46 Johnson 算法。使用一个 d 向堆实现优先队列（请见练习 2.4.41）。对于各种图的模型，找到 d 的最优值。 636

4.4 最短路径

也许最直观的图处理问题就是你常常需要使用某种地图软件或者导航系统来获取从一个地方到达另一个地方的路径。我们立即可以得到与之对应的图模型：顶点对应交叉路口，边对应公路，边的权重对应经过该路段的成本，可以是时间或者距离。如果有单行线，那就意味着还需要考虑加权有向图。在这个模型中，问题很容易就可以被归纳为：

找到从一个顶点到达另一个顶点的成本最小的路径。

除了这类问题的直接应用，最短路径模型还适用于一系列其他问题（请见表 4.4.1），其中有一些看起来似乎和图的处理毫无关系。举个例子，我们会在本节的最后考虑金融学领域的套汇问题。

表 4.4.1 最短路径的典型应用

应 用	顶 点	边
地图	交叉路口	公路
网络	路由器	网络连接
任务调度	任务	优先级限制
套汇	货币	汇率

我们采用了一个一般性的模型，即加权有向图（它是 4.2 节和 4.3 节的模型的结合）。在 4.2 节中我们希望知道从一个顶点是否可以到达另一个顶点。在本节中，我们会把权重考虑进来，就像在 4.3 节中研究的加权无向图那样。在加权有向图中，每条有向路径都有一个与之关联的路径权重，它是路径中的所有边的权重之和。这种重要的度量方式使得我们能够将这个问题归纳为“找到从一个顶点到达另一个顶点的权重最小的有向路径”，也就是本节的主题。图 4.4.1 就是一个示例。

定义。在一幅加权有向图中，从顶点 s 到顶点 t 的最短路径是所有从 s 到 t 的路径中的权重最小者。

本节中，我们将会学习解决下面这个问题的经典算法。

单点最短路径。给定一幅加权有向图和一个起点 s，回答“从 s 到给定的目的顶点 v 是否存在一条有向路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

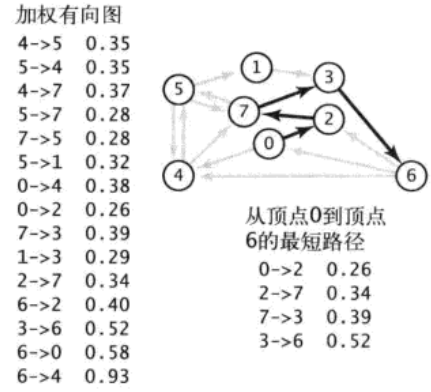


图 4.4.1 一幅加权有向图和其中的一条最短路径

我们计划在本节中讨论下列问题：

- 加权有向图的 API 和实现以及单点最短路径的 API；
- 解决边的权重非负的最短路径问题的经典 Dijkstra 算法；
- 在无环加权有向图中解决该问题的一种快速算法，边的权重甚至可以是负值；
- 适用于一般情况的经典 Bellman-Ford 算法，其中图可以含有环，边的权重也可以是负值。我们还需要算法来找出负权重的环，以及不含有这种环的加权有向图中的最短路径。

在学习了这些算法之后，我们还会考虑它们的应用。

4.4.1 最短路径的性质

最短路径问题的基本定义是很简单的，但这种简洁也隐藏了一些在学习相关的算法和数据结构之前需要解决的问题。

- 路径是有向的。最短路径需要考虑到各条边的方向。
- 权重不一定等价于距离。几何上的直觉可以帮助你理解算法，因此示例中的顶点都在平面上且权重为顶点之间的欧拉距离，例如图 4.4.1 所示的那幅有向图。但权重也可以表示时间、花费或是某种完全无关的东西，也不一定会和距离的远近成正比。我们使用了双关性的术语来强调这一点，指的是权重或是成本最短的路径。
- 并不是所有顶点都是可达的。如果 t 并不是从 s 可达的，那么就不存在任何路径，也就不存在 s 到 t 的最短路径。为了简化问题，我们的样图都是强连通的（每个顶点从另外任意一个顶点都是可达的）。
- 负权重会使问题更复杂。我们暂时假设边的权重都是正的（或零）。负权重所带来的意外效应是本节最后部分的重点。
- 最短路径一般都是简单的。我们的算法会忽略构成环的零权重边，因此找到的最短路径都不会含有环。
- 最短路径不一定是唯一的。从一个顶点到达另一个顶点的权重最小的路径可能有多条，我们只要找到其中一条即可。
- 可能存在平行边和自环：平行边中的权重最小者才会被选中，最短路径也不可能包含自环（除非自环的权重为零，但我们会忽略它）。在正文中，为了避免歧义我们隐式地假设平行边不存在，用 v→w 来表示从 v 到 w 的边，本节的代码处理它们并没有困难。

最短路径树

我们的重点是单点最短路径问题，其中给出了起点 s，计算的结果是一棵最短路径树 (SPT)，它包含了顶点 s 到所有可达的顶点的最短路径。如图 4.4.2 所示。

定义。给定一幅加权有向图和一个顶点 s，以 s 为起点的一棵最短路径树是图的一幅子图，它包含 s 和从 s 可达的所有顶点。这棵有向树的根结点为 s，树的每条路径都是有向图中的一条最短路径。

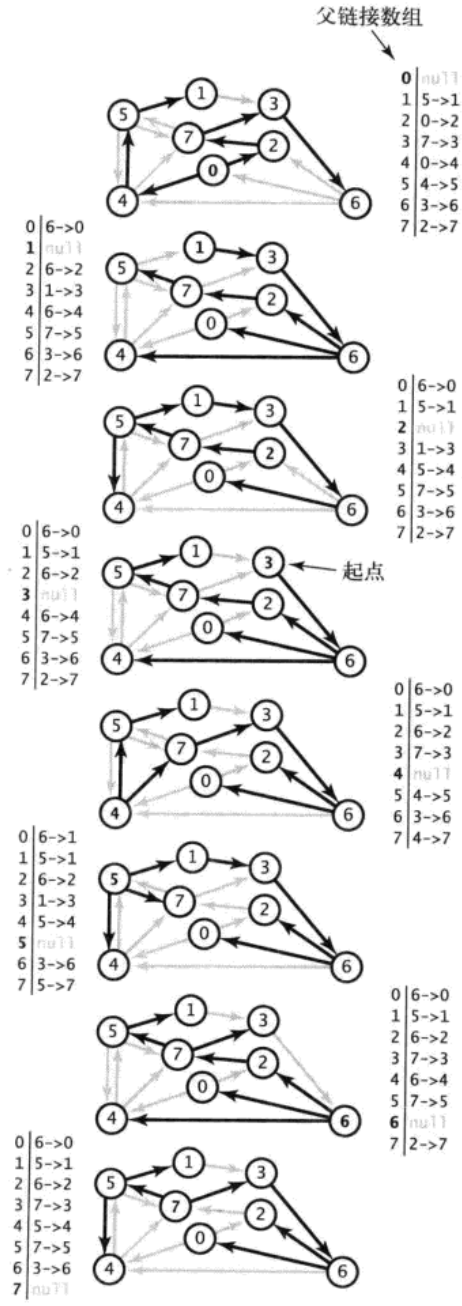


图 4.4.2 最短路径树（另见彩插）

这样一棵树是一定存在的：一般来说，从 s 到一个顶点有可能存在两条长度相等的路径。如果出现这种情况，可以删除其中一条路径的最后一条边。如此这般，直到从起点到每个顶点都只有一条路径相连（即一棵树，请见图 4.4.3）。通过构造这棵最短路径树，可以为用例提供从 s 到图中任何顶点的最短路径，表示方法为一组指向父结点的链接，和 4.1 节中表示路径的方法完全一样。



图 4.4.3 一棵含有 250 个顶点的最短路径树

637 ~ 640

4.4.2 加权有向图的数据结构

加权有向图的数据结构比加权无向图的数据结构更加简单，因为有向边只有一个方向。与 Edge 类中的 either() 和 other() 方法不同，这里定义了 from() 和 to() 方法，请见表 4.4.2。

表 4.4.2 加权有向边的 API

public class	DirectedEdge	
	DirectedEdge(int v, int w, double weight)	
double	weight()	边的权重
int	from()	指出这条边的顶点
int	to()	这条边指向的顶点
String	toString()	对象的字符串表示

从 4.1 节到 4.3 节，从 Graph 类过渡到了 EdgeWeightedGraph 类。与以前一样，我们在这里添加了 edges() 方法并使用 DirectedEdge 类代替了整型变量，请见表 4.4.3。

表 4.4.3 加权有向图的 API

public class	EdgeWeightedDigraph	
	EdgeWeightedDigraph(int V)	含有 V 个顶点的空有向图
	EdgeWeightedDigraph(In in)	从输入流中读取图的构造函数
int	V()	顶点总数
int	E()	边的总数
void	addEdge(DirectedEdge e)	将 e 添加到该有向图中
Iterable<DirectedEdge>	adj(int v)	从 v 指出的边
Iterable<DirectedEdge>	edges()	该有向图中的所有边
String	toString()	对象的字符串表示

这两份 API 的实现请见后面的框注“加权有向边的数据结构”和“加权有向图的数据结构”。它们很自然地扩展了 4.2 节和 4.3 节中相应的类的实现。Digraph 类中的邻接表使用的是整数，在 EdgeWeightedDigraph 的邻接表中使用的是 WeightedEdge 对象。与从 4.1 节到 4.2 节中 Graph 类到 Digraph 类的转换一样，从 4.3 节的 EdgeWeightedGraph 类到本节中的 EdgeWeightedDigraph 类的转换代码也变得简单了，因为在数据结构中每条边只会出现一次。

641

加权有向边的数据类型

```
public class DirectedEdge
{
   private final int v;                          // 边的起点
   private final int w;                          // 边的终点
   private final double weight;                  // 边的权重

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public double weight()
   {  return weight;  }

   public int from()

   {  return v;  }

   public int to()
   {  return w;  }

   public String toString()
   {  return String.format("%d->%d %.2f", v, w, weight);  }
}
```

DirectedEdge 类的实现比 4.3 节中无向边的数据类型 Edge 类（请见 4.3.2 节框注“带权重的边的数据类型”）更简单，因为边的两个端点是有区别的。用例可以使用惯用代码 `int v=e.to(), w=e.from();` 来访问 DirectedEdge 的两个端点。

642

加权有向图的数据类型

```
public class EdgeWeightedDigraph
{
   private final int V;                          // 顶点总数
   private int E;                                // 边的总数
   private Bag<DirectedEdge>[] adj;              // 邻接表

   public EdgeWeightedDigraph(int V)
   {
      this.V = V;
      this.E = 0;
      adj = (Bag<DirectedEdge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<DirectedEdge>();
   }

   public EdgeWeightedDigraph(In in)
   // 请见练习4.4.2

   public int V() {  return V;  }
   public int E() {  return E;  }
   public void addEdge(DirectedEdge e)
   {
      adj[e.from()].add(e);
      E++;
   }

   public Iterable<Edge> adj(int v)
```

```
    { return adj[v]; }

    public Iterable<DirectedEdge> edges()
    {
       Bag<DirectedEdge> bag = new Bag<DirectedEdge>();
       for (int v = 0; v < V; v++)
          for (DirectedEdge e : adj[v])
             bag.add(e);
       return bag;
    }

}
```

EdgeWeightedDigraph类的实现混合了EdgeWeightedGraph类和Digraph类。它维护了一个由顶点索引的Bag对象的数组，Bag对象的内容为DirectedEdge对象。与Digraph类一样，每条边在邻接表中只会出现一次：如果一条边从v指向w，那么它只会出现在v的邻接链表中。这个类可以处理自环和平行边。toString()方法的实现留作练习4.4.2。

643

图4.4.4所示的是用EdgeWeightedDigraph表示左侧的加权有向图时所构造的数据结构，在构造的过程中边被按照顺序一条一条地加入图中。与以前一样，我们使用了Bag类来表示邻接表并在图中按照标准方式将它们表示为链表。与4.2节中普通的有向图一样，每条边在数据结构中都只出现了一次。

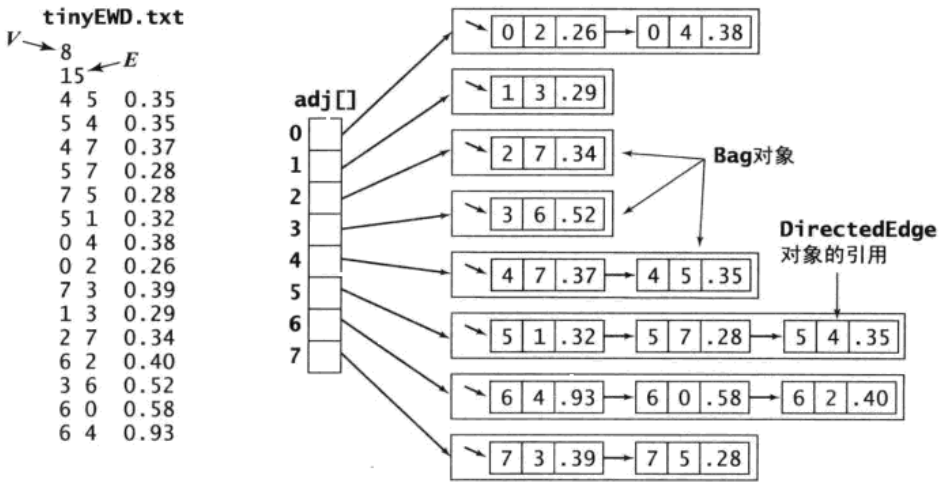


图4.4.4 加权有向图的表示

4.4.2.1 最短路径的API

对于最短路径的API，我们的设计思路与4.1节中的DepthFirstPaths和BreadthFirstPaths的API是一样的。算法将会实现表4.4.4所示的API来为用例提供图中的最短路径和其长度。

表4.4.4 最短路径的API

public class	SP	
	SP(EdgeWeightedDigraph G, int s)	构造函数
double	distTo(int v)	从顶点s到v的距离，如果不存在则路径为无穷大
boolean	hasPathTo(int v)	是否存在从顶点s到v的路径
Iterable<Direc-tedEdge>	pathTo(int v)	从顶点s到v的路径，如果不存在则为null

构造函数会创建最短路径树并计算最短路径的长度，其他查询方法则会使用这些数据结构为用例提供路径的长度以及路径的 Iterable 对象。

4.4.2.2 测试用例

右侧框注是一个简单测试用例。它接受一个输入流和一个起点作为命令行参数，从输入流中读取加权有向图，根据起点来计算有向图的最短路径树并打印从起点到其他所有顶点的最短路径。我们约定，所有的最短路径实现都使用该测试用例进行测试。在下面的框注中使用了 tinyEWD.txt 文件，它定义了一幅较小的样图中所有的边和权重，会用来显示最短路径算法的详细轨迹。它的文件格式与最小生成树算法中使用的样图相同：首先是顶点总数 V 和边的总数 E，随后是 E 行数据，每一行为两个顶点的索引和一个权重。在本书的网站上，你可以找到一些定义了更大的加权有向图的文件，包括 mediumEWG.txt。它定义了一幅含有 250 个顶点的加权有向图，如图 4.4.3 所示。在这幅图的图像中，每一行数据都表示方向相反的两条边，因此这个文件所含有的边数是在学习最小生成树时所使用的 mediumEWG.txt 的 2 倍。在最短路径树的图像中，每一行都表示一条从顶点指出的有向边。

```
public static void main(String[] args)
{
   EdgeWeightedDigraph G;
   G = new EdgeWeightedDigraph(new In(args[0]));
   int s = Integer.parseInt(args[1]);
   SP sp = new SP(G, s);

   for (int t = 0; t < G.V(); t++)
   {
      StdOut.print(s + " to " + t);
      StdOut.printf(" (%4.2f): ", sp.distTo(t));
      if (sp.hasPathTo(t))
         for (DirectedEdge e : sp.pathTo(t))
            StdOut.print(e + "   ");
      StdOut.println();
   }
}
```

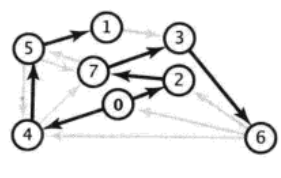
```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6
0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

最短路径的测试用例

644 ~ 645

4.4.2.3 最短路径的数据结构

表示最短路径所需的数据结构很简单，如图 4.4.5 所示。



	edgeTo[]	distTo[]
0	null	0
1	5->1	1.05
2	0->2	0.26
3	7->3	0.97
4	0->4	0.38
5	4->5	0.73
6	3->6	1.49
7	2->7	0.60

图 4.4.5 最短路径的数据结构

- 最短路径树中的边。和深度优先搜索、广度优先搜索和 Prim 算法一样，使用一个由顶点索引的 DirectedEdge 对象的父链接数组 edgeTo[]，其中 edgeTo[v] 的值为树中连接 v 和它的父结点的边（也是从 s 到 v 的最短路径上的最后一条边）。
- 到达起点的距离。我们需要一个由顶点索引的数组 distTo[]，其中 distTo[v] 为从 s 到 v 的已知最短路径的长度。

我们约定，edgeTo[s] 的值为 null，distTo[s] 的值为 0。同时还约定，从起点到不可达的顶点的距离均为 Double.POSITIVE_INFINITY。和以前一样，我们会实现使用这些数据结构的数据类型并支持用例调用方法来查询最短路径和它们的长度。

4.4.2.4 边的松弛

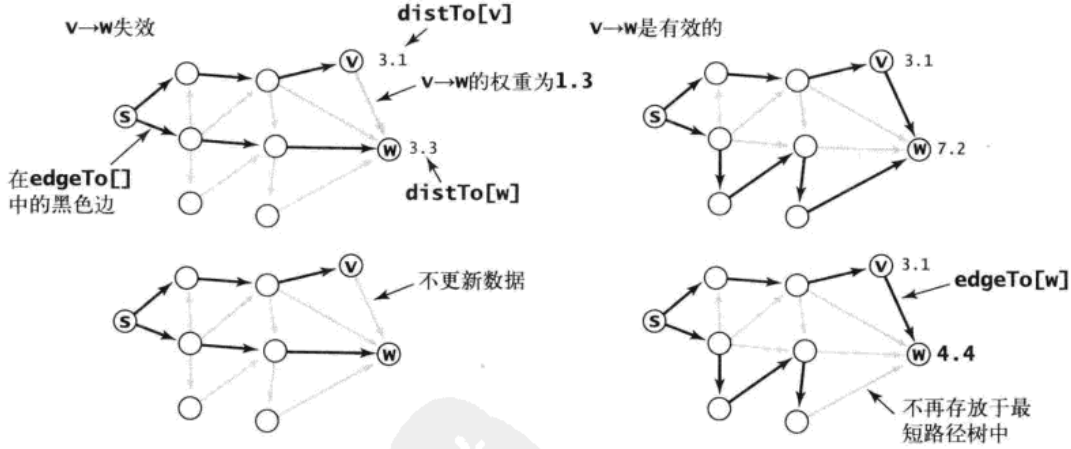
我们的最短路径 API 的实现都基于一个被称为*松弛*(relaxation)的简单操作。一开始我们只知道图的边和它们的权重，distTo[] 中只有起点所对应的元素的值为 0，其余元素的值均被初始化为 Double.POSITIVE_INFINITY。随着算法的执行，它将起点到其他顶点的最短路径信息存入了 edgeTo[] 和 distTo[] 数组中。在遇到新的边时，通过更新这些信息就可以得到新的最短路径。特别是，我们在其中会用到*边的松弛*技术，定义如下：*放松*边 v→w 意味着检查从 s 到 w 的最短路径是否是先从 s 到 v，然后再由 v 到 w。如果是，则根据这个情况更新数据结构的内容。上边框注中的代码实现了这个操作。

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
   }
}
```

边的松弛

646 由 v 到达 w 的最短路径是 distTo[v] 与 e.weight() 之和——如果这个值不小于 distTo[w]，称这条边*失效*了并将它忽略；如果这个值更小，就更新数据。

图 4.4.6 显示的是边的放松操作之后可能出现的两种情况。一种情况是边失效（左边的例子），不更新任何数据；另一种情况是 v→w 就是到达 w 的最短路径（右边的例子），这将会更新 edgeTo[w] 和 distTo[w]（这可能会使另一些边失效，但也可能产生一些新的有效边）。*松弛*这个术语来自于用一根橡皮筋沿着连接两个顶点的路径紧紧展开的比喻：放松一条边就类似于将橡皮筋转移到一条更短的路径上，从而缓解了橡皮筋的压力。如果 relax() 改变了和边 e 相关的顶点的 distTo[e.to()] 和 edgeTo[e.to()] 的值，就称 e 的*放松是成功的*。



647 图 4.4.6 边的松弛的两种情况（另见彩插）

4.4.2.5 顶点的松弛

实际上，实现会放松从一个给定顶点指出的所有边，如下页框注中（被重载的）relax() 的实现所示。注意，从任意 distTo[v] 为有限值的顶点 v 指向任意 distT[] 为无穷的顶点的边都是有效的。如果 v 被放松，那么这些有效边都会被添加到 edgeTo[] 中。某条从起点指出的边将会是第一条被加入 edgeTo[] 中的边。算法会谨慎选择顶点，使得每次顶点松弛操作都能得出到达某个顶点的更短的路径，最后逐渐找出到达每个顶点的最短路径。如图 4.4.7 所示。

```
private void relax(EdgeWeightedDigraph G, int v)
{
   for (DirectedEdge e : G.adj(v))
   {
      int w = e.to();
      if (distTo[w] > distTo[v] + e.weight())
      {
         distTo[w] = distTo[v] + e.weight();
         edgeTo[w] = e;
      }
   }
}
```

顶点的松弛

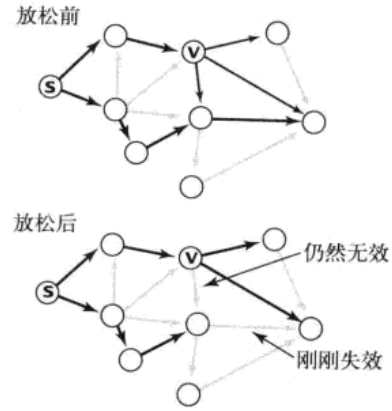


图 4.4.7 顶点的松弛

648

4.4.2.6 为用例准备的查询方法

与 4.1 节（以及练习 4.1.13）中实现路径查找的 API 相似，edgeTo[] 和 distTo[] 数组直接支持 pathTo()、hasPathTo() 和 distTo() 查询方法，如下方框注所示。默认所有最短路径的实现都包含这段代码。前面已经提到过，只有在 v 是从 s 可达的情况下，distTo[v] 才是有意义的，还已经约定，对于从 s 不可达的顶点，distTo() 方法都应该返回无穷大。在实现这个约定时，将 distTo[] 中的所有元素都初始化为 Double.POSITIVE_INFINITY，distTo[s] 则为 0。最短路径算法会将从起点可达的顶点 v 的 distTo[v] 设为一个有限值，这样就不必再用 marked[] 数组来在图的搜索中标记可达的顶点，而是通过检测 distTo[v] 是否为 Double.POSITIVE_INFINITY 来实现 hasPathTo(v)。对于 pathTo() 方法，我们约定如果 v 不是从起点可达的则返回 null，如果 v 等于起点则返回一条不含任何边的路径。对于可达的顶点，我们会遍历最短路径树并返回栈上的所有边，这和 DepthFirstPaths 以及 BreadthFirstPaths 的做法完全一样。图 4.4.8 显示了在示例中路径 0 → 2 → 7 → 3 → 6 是如何被找到的。

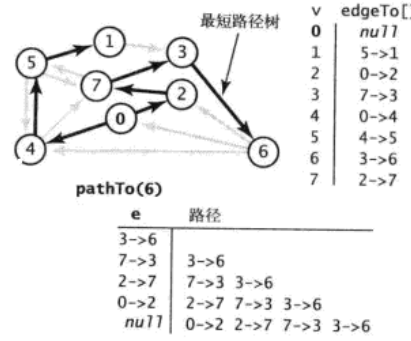


图 4.4.8 pathTo() 方法的计算轨迹

```
public double distTo(int v)
{   return distTo[v];   }

public boolean hasPathTo(int v)
{   return distTo[v] < Double.POSITIVE_INFINITY;  }

public Iterable<DirectedEdge> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<DirectedEdge> path = new Stack<DirectedEdge>();
   for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

最短路径API中的查询方法

649

4.4.3 最短路径算法的理论基础

边的放松操作是一项非常容易实现的重要操作，它是实现最短路径算法的基础。同时，它也是理解这个算法的理论基础并使我们能够完整地证明算法的正确性。

4.4.3.1 最优性条件

以下命题证明了判断路径是否为最短路径的全局条件与在放松一条边时所检测的局部条件是等价的。

命题P（最短路径的最优性条件）。令G为一幅加权有向图，顶点s是G中的起点，distTo[]是一个由顶点索引的数组，保存的是G中路径的长度。对于从s可达的所有顶点v，distTo[v]的值是从s到v的某条路径的长度，对于从s不可达的所有顶点v，该值为无穷大。当且仅当对于从v到w的任意一条边e，这些值都满足distTo[w]<=distTo[v]+e.weight()时（换句话说，不存在有效边时），它们是最短路径的长度。

证明。假设distTo[w]是从s到w的最短路径。如果对于某条从v到w的边e有distTo[w]>distTo[v]+e.weight()，那么从s到w（经过v）且经过e的路径的长度必然小于distTo[w]，矛盾。因此最优性条件是必要的。

要证明最优性条件是充分的，假设w是从s可达的且$s=v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k=w$是从s到w的最短路径，其权重为$OPT_{sw}$。对于1到$k$之间的$i$，令$e_i$表示$v_{i-1}$到$v_i$的边。根据最优性条件，可以得到以下不等式：

$$
\begin{aligned}
\text{distTo}[w] = \text{distTo}[v_k] &<= \text{distTo}[v_{k-1}] + e_k\text{.weight()} \\
\text{distTo}[v_{k-1}] &<= \text{distTo}[v_{k-2}] + e_{k-1}\text{.weight()} \\
&\ldots \\
\text{distTo}[v_2] &<= \text{distTo}[v_1] + e_2\text{.weight()} \\
\text{distTo}[v_1] &<= \text{distTo}[s] + e_1\text{.weight()}
\end{aligned}
$$

综合这些不等式并去掉distTo[s]=0.0，得到：

$$\text{distTo}[w] <= e_1\text{.weight()} + \ldots + e_k\text{.weight()} = OPT_{sw}.$$

现在，distTo[w]为从s到w的某条边的长度，因此它不可能比最短路径更短。所以以下等式必然成立。

$$OPT_{sw} <= \text{distTo}[w] <= OPT_{sw}$$

650

4.4.3.2 验证

命题P的一个重要的实际应用是最短路径的验证。无论一种算法会如何计算distTo[]，都只需要遍历图中的所有边一遍并检查最优性条件是否满足就能够知道该数组中的值是否是最短路径的长度。最短路径的算法可能会很复杂，因此能够快速验证计算的结果就变得很重要。为此，我们在本书的网站上的实现中包含了一个check()方法。该方法还会检查edgeTo[]指明的路径并验证它与distTo[]是否一致。

4.4.3.3 通用算法

由最优性条件马上可以得到一个能够涵盖已经学习过的所有最短路径算法的通用算法。现在，我们暂时只研究非负权重的情况。

命题 Q（通用最短路径算法）。将 distTo[s] 初始化为 0，其他 distTo[] 元素初始化为无穷大，继续如下操作：

放松 G 中的任意边，直到不存在有效边为止。

对于任意从 s 可达的顶点 w，在进行这些操作之后，distTo[w] 的值即为从 s 到 w 的最短路径的长度（且 edgeTo[w] 的值即为该路径上的最后一条边）。

证明。放松边 v→w 必然会将 distTo[w] 的值设为从 s 到 w 的某条路径的长度（且将 edgeTo[w] 设为该路径上的最后一条边）。对于从 s 可达的任意顶点 w，只要 distTo[w] 仍然是无穷大，到达 w 的最短路径上的某条边肯定仍然是有效的，因此算法的操作会不断继续，直到由 s 可达的每个顶点的 distTo[] 值均变为到达该顶点的某条路径的长度。对于已经找到最短路径的任意顶点 v，在算法的计算过程中 distTo[v] 的值都是从 s 到 v 的某条（简单）路径的长度且必然是单调递减的。因此，它递减的次数必然是有限的（每切换一条 s 到 v 简单路径就递减一次）。当不存在有效边的时候，命题 P 就成立了。

将最优性条件和通用算法放在一起学习的关键原因是，通用算法并没有指定边的放松顺序。因此，要证明这些算法都能通过计算得到最短路径，只需证明它们都会放松所有的边直到所有边都失效即可。

651

4.4.4 Dijkstra 算法

在 4.3 节中，我们讨论了寻找加权无向图中的最小生成树的 Prim 算法：构造最小生成树的每一步都向这棵树中添加一条新的边。Dijkstra 算法采用了类似的方法来计算最短路径树。首先将 distTo[s] 初始化为 0，distTo[] 中的其他元素初始化为正无穷。然后将 distTo[] 最小的非树顶点放松并加入树中，如此这般，直到所有的顶点都在树中或者所有的非树顶点的 distTo[] 值均为无穷大。

命题 R。Dijkstra 算法能够解决边权重非负的加权有向图的单起点最短路径问题。

证明。如果 v 是从起点可达的，那么所有 v→w 的边都只会被放松一次。当 v 被放松时，必有 distTo[w]<=distTo[v]+e.weight()。该不等式在算法结束前都会成立，因此 distTo[w] 只会变小（放松操作只会减小 distTo[] 的值）而 distTo[v] 则不会改变（因为边的权重非负且在每一步中算法都会选择 distTo[] 最小的顶点，之后的放松操作不可能使任何 distTo[] 的值小于 distTo[v]）。因此，在所有从 s 可达的顶点均被添加到树中之后，最短路径的最优性条件成立，即命题 P 成立。

4.4.4.1 数据结构

要实现 Dijkstra 算法，除了 distTo[] 和 edgeTo[] 数组之外还需要一条索引优先队列 pq，以保存需要被放松的顶点并确认下一个被放松的顶点。我们知道 IndexMinPQ 可以将索引和键（优先级）关联起来并且可以删除并返回优先级最低的索引。在这里，只要将顶点 v 和 distTo[v] 关联起来就立即可以得到 Dijkstra 算法的实现。另外，稍加推导也可以知道，edgeTo[] 中的元素所对应的可达顶点构成了一棵最短路径树。

4.4.4.2 换一个角度看问题

根据算法的证明，我们可以从另一个角度来理解它，如图 4.4.9 所示，已知树结点所对应的 distTo[] 值均为最短路径的长度。对于优先队列中的任意顶点 w，distTo[w] 是从 s 到 w 的最短路径的长度，该路径上的所有顶点均在树中且路径上的最后一条边为 edgeTo[w]。优先级最小的顶点的 distTo[] 值就是最短路径的权重，它不会小于已经被放松过的任意顶点的最短路径的权重，也不会大于还未被放松过的任意顶点的最短路径的权重。这个顶点就是下一个要被放松的顶点。所有从 s 可达的顶点都会按照最短路径的权重顺序被放松。

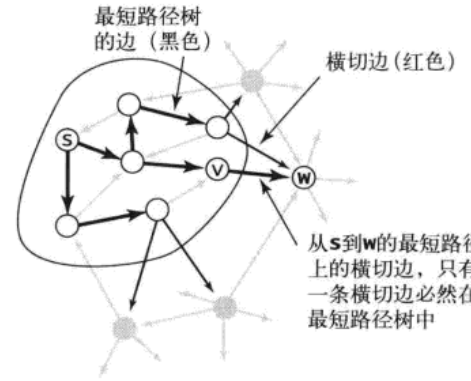


图 4.4.9 Dijkstra 的最短路径算法（另见彩插）

图 4.4.10 是算法处理样图 tinyEWD.txt 时的轨迹。在这个例子中，算法构造最短路径树的过程如下所述。

- 将顶点 0 添加到树中，将顶点 2 和 4 加入优先队列。
- 从优先队列中删除顶点 2，将 0→2 添加到树中，将顶点 7 加入优先队列。
- 从优先队列中删除顶点 4，将 0→4 添加到树中，将顶点 5 加入优先队列，边 4→7 失效。
- 从优先队列中删除顶点 7，将 2→7 添加到树中，将顶点 3 加入优先队列，边 7→5 失效。
- 从优先队列中删除顶点 5，将 4→5 添加到树中，将顶点 1 加入优先队列，边 5→7 失效。
- 从优先队列中删除顶点 3，将 7→3 添加到树中，将顶点 6 加入优先队列。
- 从优先队列中删除顶点 1，将 5→1 添加到树中，边 1→3 失效。
- 从优先队列中删除顶点 6，将 3→6 添加到树中。

算法按照顶点到起点的最短路径的长度的增序将它们添加到最短路径树中，如图 4.4.10 右侧的红色箭头所示。

Dijkstra 算法的实现 DijkstraSP（算法 4.9）只是用代码复述了算法的描述，还在 relax() 方法中添加了一行语句来处理以下两种情况：要么边的 to() 得到的顶点还不在优先队列中，此时需要使用 insert() 方法将它加入到优先队列中；要么它已经在优先队列中且优先级需要被降低，此时可以用 change() 方法实现。

命题 R（续）。在一幅含有 V 个顶点和 E 条边的加权有向图中，使用 Dijkstra 算法计算根结点为给定起点的最短路径树所需的空间与 V 成正比，时间与 $E\log V$ 成正比（最坏情况下）。

证明。同 Prim 算法的证明（请见命题 N）

如前所述，思考 Dijkstra 算法的另一种方式就是将它和 4.3 节的 Prim 算法（算法 4.7）相比较。两种算法都会用添加边的方式构造一棵树：Prim 算法每次添加的都是离树最近的非树顶点，Dijkstra 算法每次添加的都是离起点最近的非树顶点。它们都不需要 marked[] 数组，因为条件 !marked[w] 等价于条件 distTo[w] 为无穷大。换句话说，将算法 4.9 中的有向图换成无向图并忽略 relax() 方法中 distTo[v] 部分的代码，就会得到算法 4.7，也就是 Prim 算法的即时版本（！）。同样，根

据 LazyPrimMST（4.3.4 节框注“最小生成树的 Prim 算法的延时实现”）实现 Dijkstra 算法的延时版本也并不困难。

4.4.4.3 变种

我们只需对 Dijkstra 算法的实现稍作适当的修改就能够解决这个问题的其他版本，例如，加权无向图中的单点最短路径。给定一幅加权无向图和一个起点 s，回答“是否存在一条从 s 到给定的顶点 v 的路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

如果将无向图看作有向图，这个问题的答案就很简单了。也就是说，对于给定的加权无向图，创建一幅由相同顶点构成的加权有向图，且对于无向图中的每条边，相应地创建两条（方向不同）有向边。有向图中的路径和无向图中的路径存在着一一对应的关系，路径的权重也是相同的——最短路径的问题是等价的。

算法 4.9 最短路径的 Dijkstra 算法

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph
G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new
IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_
INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
         relax(G, pq.delMin())
   }

   private void
relax(EdgeWeightedDigraph G, int v)
   {
      for(DirectedEdge e : G.adj(v))
      {
         int w = e.to();
         if (distTo[w] > distTo[v] + e.
weight())
```

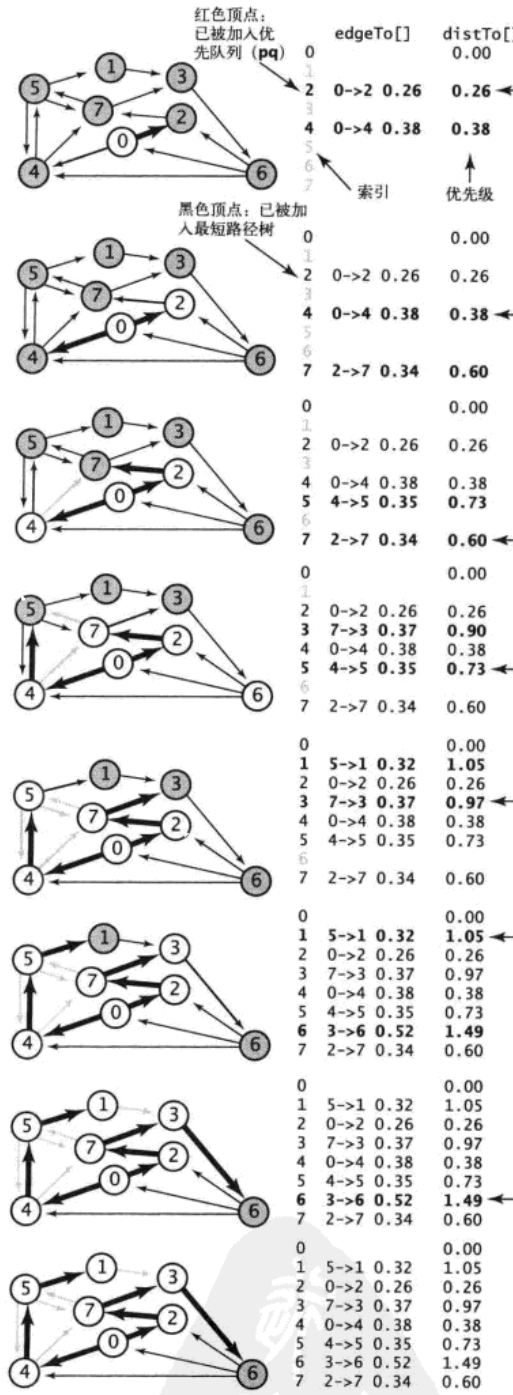


图 4.4.10 Dijkstra 算法的轨迹（另见彩插）

652
~
654

```
            {
               distTo[w] = distTo[v] + e.weight();
               edgeTo[w] = e;
               if (pq.contains(w)) pq.change(w, distTo[w]);
               else                pq.insert(w, distTo[w]);
            }
         }
      }

   public double distTo(int v)          // 最短路径树实现中的标准查询算法
   public boolean hasPathTo(int v)      // （请见4.4.2.6节框注“最短路径
   public Iterable<Edge> pathTo(int v) // API中的查询方法）
}
```

Dijkstra 算法的实现每次都会为最短路径树添加一条边，该边由一个树中的顶点指向一个非树顶点 w 且它是到 s 最近的顶点。

给定两点的最短路径。给定一幅加权有向图以及一个起点 s 和一个终点 t，找到从 s 到 t 的最短路径。

要解决这个问题，你可以使用 Dijkstra 算法并在从优先队列中取到 t 之后终止搜索。

任意顶点对之间的最短路径。给定一幅加权有向图，回答“给定一个起点 s 和一个终点 t，是否存在一条从 s 到 t 的路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

```
public class DijkstraAllPairsSP
{
   private DijkstraSP[] all;

   DijkstraAllPairsSP(EdgeWeightedDigraph G)
   {
      all = new DijkstraSP[G.V()]
      for (int v = 0; v < G.V(); v++)
         all[v] = new DijkstraSP(G, v);
   }

   Iterable<Edge> path(int s, int t)
   {  return all[s].pathTo(t);  }

   double dist(int s, int t)
   {  return all[s].distTo(t);  }

}
```

任意顶点对之间的最短路径

右边框注中短小精悍的代码解决了任意顶点对之间的最短路径问题，所需的时间和空间都与 $E V \log V$ 成正比。它构造了 DijkstraSP 对象的数组，每个元素都将相应的顶点作为起点。在用例进行查询时，代码会访问起点所对应的单点最短路径对象并将目的顶点作为参数进行查询。

欧拉图中的最短路径。在顶点为平面上的点且边的权重与顶点欧拉间距成正比的图中，解决单点、给定两点和任意顶点对之间的最短路径问题。

在这种情况下，有一个小小的改动可以大幅提高 Dijkstra 算法的运行速度（请见练习 4.4.27）。

图 4.4.11 显示的是 Dijkstra 算法在处理测试文件 mediumEWD.txt（请见 4.4.2.2 节）所定义的欧拉图时用若干不同的起点产生最短路径树的过程。和之前一样，这幅图中的线段都表示双向的有向边。这些图片展示了一段引人入胜的动态过程。

下面，我们将会考虑加权无环图中的最短路径算法并且将在线性时间内解决该问题（比 Dijkstra 算法要快）。然后是负权重的加权有向图中的最短路径问题，Dijkstra 算法不适用于这种情况。

655

656

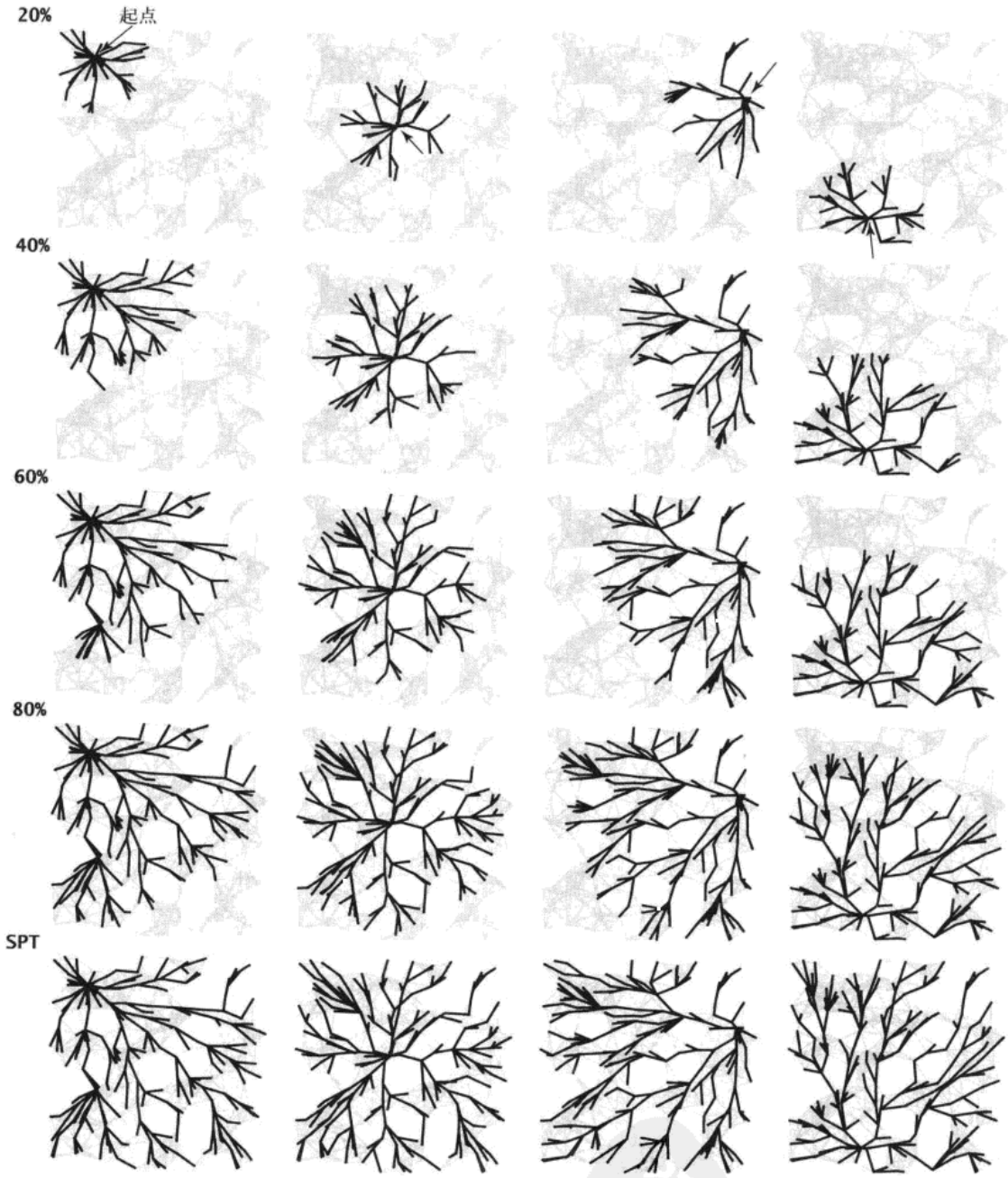


图 4.4.11 Dijkstra 算法（250 个顶点，不同的起点）

657

4.4.5 无环加权有向图中的最短路径算法

许多应用中的加权有向图都是不含有有向环的。我们现在来学习一种比 Dijkstra 算法更快、更简单的在无环加权有向图中找出最短路径的算法，如图 4.4.12 所示。它的特点是：

- ❑ 能够在线性时间内解决单点最短路径问题；
- ❑ 能够处理负权重的边；
- ❑ 能够解决相关的问题，例如找出最长的路径。

这些算法都是在4.2节中学过的无环有向图的拓扑排序算法的简单扩展。

特别的是，只要将顶点的放松和拓扑排序结合起来，马上就能够得到一种解决无环加权有向图中的最短路径问题的算法。首先，将 distTo[s] 初始化为0，其他 distTo[] 元素初始化为无穷大，然后一个一个地按照拓扑顺序放松所有顶点。我们可以用与Dijkstra算法的证明（命题R）类似的方法证明这个方法的正确性。

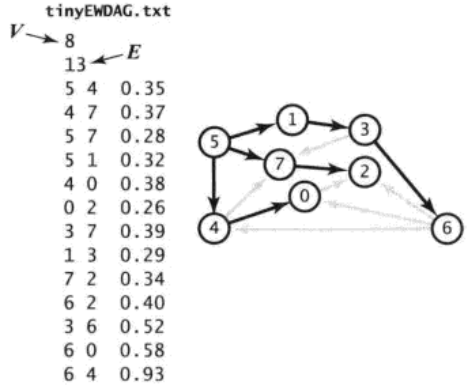


图 4.4.12 一幅无环加权有向图和它的一棵最短路径树

命题S。按照拓扑顺序放松顶点，就能在和 $E+V$ 成正比的时间内解决无环加权有向图的单点最短路径问题。

证明。每条边 v→w 都只会被放松一次。当 v 被放松时，得到：distTo[w]<= distTo[v]+e.weight()。在算法结束前该不等式都成立，因为 distTo[v] 是不会变化的（因为是按照拓扑顺序放松顶点，在 v 被放松之后算法不会再处理任何指向 v 的边）而 distTo[w] 只会变小（任何放松操作都只会减小 distTo[] 中的元素的值）。因此，在所有从 s 可达的顶点都被加入到树中后，最短路径的最优性条件成立，命题Q也就成立了。时间上限很容易得到：命题G告诉我们拓扑排序所需的时间与 $E+V$ 成正比，而在第二次遍历中每条边都只会被放松一次，因此算法总耗时与 $E+V$ 成正比。

658

图4.4.13是算法处理无环加权有向样图 tinyEWDAG.txt 的轨迹。在这个例子中，算法由顶点5开始按照以下步骤构建了一棵最短路径树：

- ❑ 用深度优先搜索得到图的顶点的拓扑排序 5 1 3 6 4 7 0 2；
- ❑ 将顶点5和从它指出的所有边添加到树中；
- ❑ 将顶点1和边1→3添加到树中；
- ❑ 将顶点3和边3→6添加到树中，边3→7已经失效；
- ❑ 将顶点6和边6→2、6→0添加到树中，边6→4已经失效；
- ❑ 将顶点4和边4→0添加到树中，边4→7和6→0已经失效；
- ❑ 将顶点7和边7→2添加到树中，边6→2已经失效；
- ❑ 将顶点0添加到树中，边0→2已经失效；
- ❑ 将顶点2添加到树中。

图中没有画出将2添加到树中的一步，拓扑序列中的最后一个顶点没有指出的边。

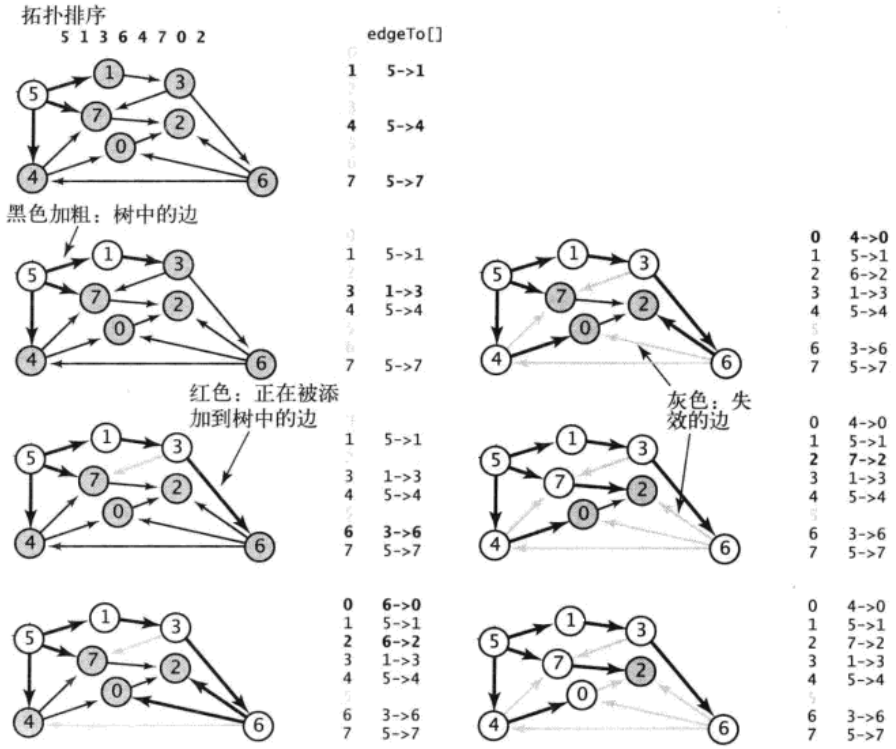


图 4.4.13 寻找无环加权有向图中的最短路径的算法轨迹（另见彩插）

算法 4.10 在实现中直接使用了已学习过的许多代码。它假设 Topological 类使用本节中介绍的 EdgeWeightedDigraph 类和 DirectedEdge 类的 API（请见练习 4.4.12）重载了拓扑排序的方法。注意，该实现中不需要布尔数组 marked[]：因为是按照拓扑顺序处理无环有向图中的顶点，所以不可能再次遇到已经被放松过的顶点。算法 4.10 的效率几乎已经没有提高的空间了：在拓扑排序后，构造函数会扫描整幅图并将每条边放松一次。在已知加权图是无环的情况下，它是找出最短路径的最好方法。

659

算法 4.10 无环加权有向图的最短路径算法

```
public class AcyclicSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;

   public AcyclicSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      Topological top = new Topological(G);
```

```
        for (int v : top.order())
            relax(G, v);
    }

    private void relax(EdgeWeightedDigraph G, int v)
    // 请见4.4.1.5框注“顶点的松驰”

    public double distTo(int v)          // 最短路径树实现中的标准查询算法（请见4.4.1.6
                                            框注“最短路径API的查询方法”）
    public boolean hasPathTo(int v)
    public Iterable<DirectedEdge> pathTo(int v)
}
```

无环加权有向图的最短路径算法使用了拓扑排序（算法 4.5，重载了 EdgeWeightedDigraph 类和 DirectedEdge 类）来按照拓扑顺序放松所有顶点，这对于计算出图中的最短路径已经足够了。

```
% java AcyclicSP tinyEWDAG.txt 5
5 to 0 (0.73): 5->4 0.35  4->0 0.38
5 to 1 (0.32): 5->1 0.32
5 to 2 (0.62): 5->7 0.28  7->2 0.34
5 to 3 (0.62): 5->1 0.32  1->3 0.29
5 to 4 (0.35): 5->4 0.35
5 to 5 (0.00):
5 to 6 (1.13): 5->1 0.32  1->3 0.29  3->6 0.52
5 to 7 (0.28): 5->7 0.28
```

660

命题 S 很重要，因为它的“无环”能够极大地简化问题的论断。对于最短路径问题，基于拓扑排序的方法比 Dijkstra 算法快的倍数与 Dijkstra 算法中所有优先队列操作的总成本成正比。另外，命题 S 的证明和边的权重是否非负无关，因此无环加权有向图不会受到任何限制。下面用这个特点解决边的负权重问题。我们会考虑使用这个最短路径模型来解决另外两个问题，其中之一乍一看甚至和图的处理似乎没有任何关系。

4.4.5.1 最长路径

考虑在无环加权有向图中寻找最长路径的问题，边的权重可正可负。

无环加权有向图中的单点最长路径。给定一幅无环加权有向图（边的权重可能为负）和一个起点 s，回答“是否存在一条从 s 到给定的顶点 v 的路径？如果有，找出最长（总权重最大）的那条路径。”等类似问题。

我们刚刚学习过的算法能够快速地解决这个问题。

命题 T。解决无环加权有向图中的最长路径问题所需的时间与 $E+V$ 成正比。

证明。给定一个最长路径问题，复制原始无环加权有向图得到一个副本并将副本中的所有边的权重变为负值。这样，副本中的最短路径即为原图中的最长路径。要将最短路径问题的答案转换为最长路径问题的答案，只需将方案中的权重变为正值即可。根据命题 S 立即可以得到算法所需的时间。

根据这种转换实现 AcyclicLP 类来寻找一幅无环加权有向图中的最长路径就十分简单了。实现该类的一个更简单的方法是修改 AcyclicSP，将 distTo[] 的初始值变为 Double.NEGATIVE_INFINITY 并改变 relax() 方法中的不等式的方向。无论使用哪种方法，都能得到无环加权有向图中的最长路径问题的一种高效的解决方案。和它形成鲜明对比的是，在一般的加权有向图（边的权重可能为负）中寻找最长简单路径的已知最好算法在最坏情况下所需的时间是指数级别的（请见第 6 章）！出现环的可能性似乎使这个问题的难度以指数级别增长。

图 4.4.14 是算法在无环加权有向样图 tinyEWDAG.txt 中寻找最长路径的轨迹，你可以将它与图 4.4.13 相比较。在这个例子中，算法由顶点 5 按照以下步骤构建了一棵最长路径树：

- 用深度优先搜索得到图的顶点的拓扑排序 5 1 3 6 4 7 0 2；
- 将顶点 5 和从它指出的所有边添加到树中；
- 将顶点 1 和边 1→3 添加到树中；
- 将顶点 3 和边 3→6、3→7 添加到树中，边 5→7 已经失效；
- 将顶点 6 和边 6→2、6→4 和 6→0 添加到树中；
- 将顶点 4 和边 4→0、4→7 添加到树中，边 6→0 和 3→7 已经失效；
- 将顶点 7 和边 7→2 添加到树中，边 6→2 已经失效；
- 将顶点 0 添加到树中，边 0→2 已经失效；
- 将顶点 2 添加到树中（未画出）。

最长路径算法处理顶点的顺序和最短路径算法一样，但产生的结果却完全不同。

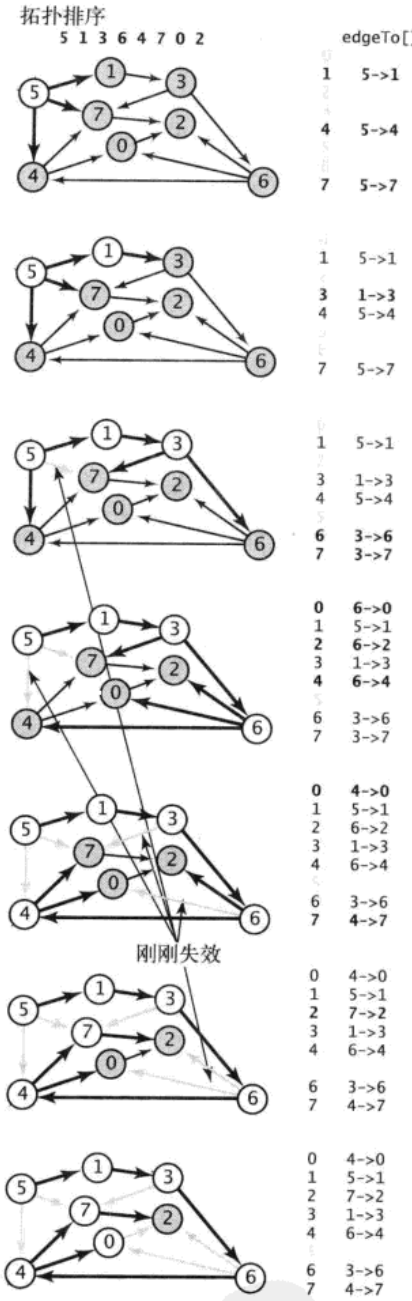


图 4.4.14 无环图中的最长路径算法（另见彩插）

661 ~ 662

4.4.5.2 平行任务调度

作为算法应用的示例，我们再次考虑在 4.2 节中出现过的任务调度类的问题。这次需要解决以下调度问题（楷体部分为与 4.2.4.1 节的问题描述的不同之处）。

优先级限制下的并行任务调度。给定一组需要完成的特定任务，以及一组关于任务完成的先后次序的优先级限制。在满足限制条件的前提下应该如何在*若干相同的处理器上（数量不限）安排任务并在最短的时间内完成所有任务*？

4.2 节的模型默认只有单个处理器：将任务按照拓扑顺序排序，完成任务的总耗时就是所有任务所需要的总时间。现在假设有足够多的处理器并能够同时处理任意多的任务，受到的只有优先级

的限制。和以前一样，需要处理的任务可能上百万甚至上亿，因此需要一个高效的算法。令人兴奋的是，正好存在一种线性时间的算法——一种叫做“关键路径“的方法能够证明这个问题与无环加权有向图中的最长路径问题是等价的。这个方法已成功应用于无数的工业软件之中。

假设任意可用的处理器都能在任务所需的时间内完成它，那么我们的重点就是尽早安排每一个任务。例如，表 4.4.5 给出了一个任务调度问题，图 4.4.15 给出的解决方案显示了这个问题所需的最短时间为 173.0。这份调度方案满足了所有限制条件，没有其他调度方案能比它耗时更少，因为任务必须按照 0→9→6→8→2 的顺序完成。这个顺序就是这个问题的关键路径。由优先级限制指定的每一列任务都代表了调度方案的一种可能的时间下限。如果将一系列任务的长度定义为完成所有任务的最早可能时间，那么最长的任务序列就是问题的关键路径，因为在这份任务序列中任何任务的启动延迟都会影响到整个项目的完成时间。

表 4.4.5 一个任务调度问题

任务	时耗	必须在以下任务之前完成
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6

定义。解决并行任务调度问题的关键路径方法的步骤如下：创建一幅无环加权有向图，其中包含一个起点 s 和一个终点 t 且每个任务都对应着两个顶点（一个起始顶点和一个结束顶点）。对于每个任务都有一条从它的起始顶点指向结束顶点的边，边的权重为任务所需的时间。对于每条优先级限制 v→w，添加一条从 v 的结束顶点指向 w 的起始顶点的权重为零的边。我们还需要为每个任务添加一条从起点指向该任务的起始顶点的权重为零的边以及一条从该任务的结束顶点到终点的权重为零的边。这样，每个任务预计的开始时间即为从起点到它的起始顶点的最长距离。

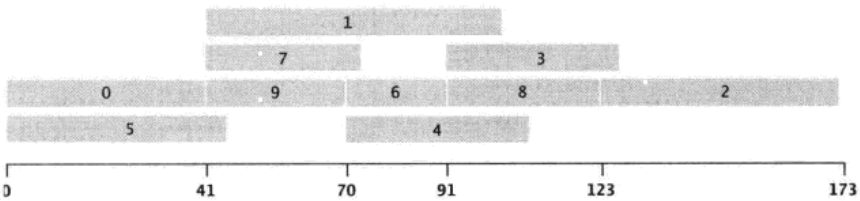


图 4.4.15 并行任务调度问题的解决方案

图 4.4.16 显示的是示例任务所对应的图，图 4.4.17 则显示的是最长路径的答案。如定义所述，在图中每个任务都对应着三条边（从起点到起始顶点、从结束顶点到终点的权重为零的边，以及一条从起始顶点到结束顶点的边），每个优先级限制条件都对应着一条边。后面框注“优先级限制下的并行任务调度问题的关键路径方法”中的 CPM 类简洁明了地实现了关键路径方法。它能够将任意任务调度问题转化为无环加权有向图中的一个最长路径问题，用 AcyclicLP 解决它并打印出每个任务的开始时间以及调度方案的结束时间。

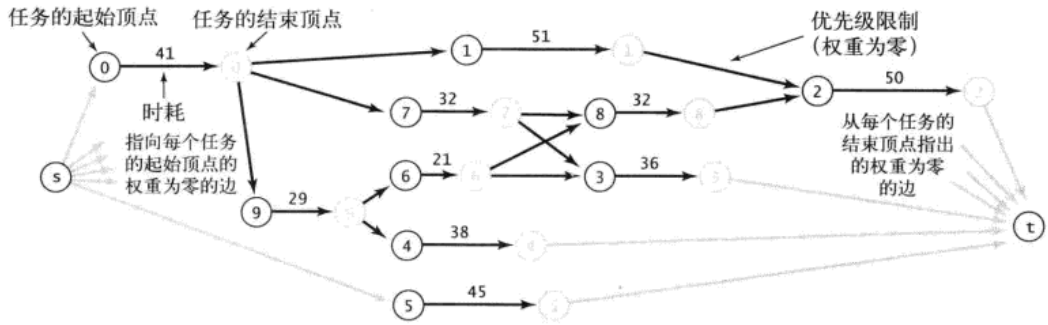


图 4.4.16 任务调度问题的无环加权有向图表示

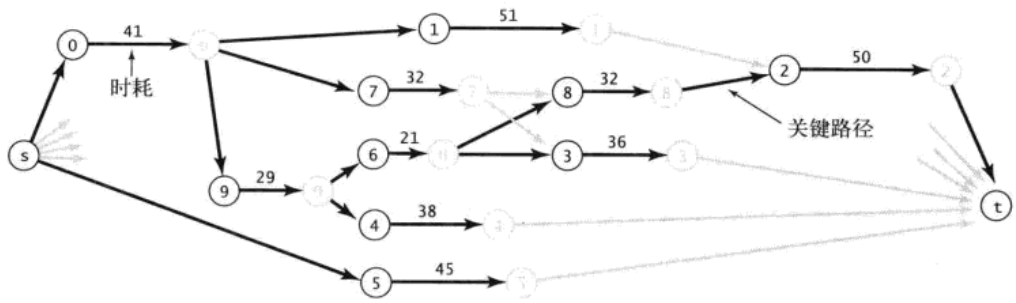


图 4.4.17 任务调度示例问题的最长路径解决方案

663
~
664

优先级限制下的并行任务调度问题的关键路径方法

```
public class CPM
{
   public static void main(String[] args)
   {
      int N = StdIn.readInt(); StdIn.readLine();
      EdgeWeightedDigraph G;
      G = new EdgeWeightedDigraph(2*N+2);
      int s = 2*N, t = 2*N+1;
      for (int i = 0; i < N; i++)
      {
         String[] a = StdIn.readLine().split("\\s+");
         double duration = Double.parseDouble(a[0]);
         G.addEdge(new DirectedEdge(i, i+N, duration));
         G.addEdge(new DirectedEdge(s, i, 0.0));

         G.addEdge(new DirectedEdge(i+N, t, 0.0));
         for (int j = 1; j < a.length; j++)
         {
            int successor = Integer.parseInt(a[j]);
            G.addEdge(new DirectedEdge(i+N, successor, 0.0));
         }
      }
```

```
% more jobsPC.
txt
10
41.0  1 7 9
51.0  2
50.0
36.0
38.0
45.0
21.0  3 8
32.0  3 8
32.0  2
29.0  4 6
```

```
        AcyclicLP lp = new AcyclicLP(G, s);

        StdOut.println("Start times:");
        for (int i = 0; i < N; i++)
           StdOut.printf("%4d: %5.1f\n", i, lp.distTo(i));
        StdOut.printf("Finish time: %5.1f\n", lp.distTo(t));
    }

}
```

```
% java CPM <
jobsPC.txt
Start times:
   0:   0.0
   1:  41.0
   2: 123.0
   3:  91.0
   4:  70.0
   5:   0.0
   6:  70.0
   7:  41.0
   8:  91.0
   9:  41.0
Finish time:
173.0
```

这里实现的任务调度问题的关键路径方法将问题归约为寻找无环加权有向图的最长路径问题。它会根据任务调度问题的描述用关键路径的方法构造一幅加权有向图（且必然是无环的），然后使用 AcyclicLP（请见命题 T）找到图中的最长路径树，最后打印出各条最长路径的长度，也就正好是每个任务的开始时间。

665

命题 U。解决优先级限制下的并行任务调度问题的关键路径法所需的时间为线性级别。

证明。为什么 CPM 类能够解决问题？算法的正确性依赖于两个因素。首先，在相应的有向无环图中，每条路径都是由任务的起始顶点和结束顶点组成的并由权重为零的优先级限制条件的边分隔——从起点 s 到任意顶点 v 的任意路径的长度都是任务 v 的开始 / 结束时间的下限，因为这已经是在同一台处理器上顺序完成这些任务的最优的排列顺序了。因此，从起点 s 到终点 t 的最长路径就是所有任务的完成时间的下限。第二，由最长路径得到的所有开始和结束时间都是可行的——每个任务都只能在优先级限制指定的先导任务完成之后开始，因为它的开始时间就是顶点到它的起始顶点的最长路径的长度。因此，从起点 s 到终点 t 的最长路径长度就是所有任务完成时间的上限。由命题 T 很容易得到算法所需的时间是线性的。

4.4.5.3 相对最后期限限制下的并行任务调度

一般的最后期限（deadline）都是相对于第一个任务的开始时间而言的。假设在任务调度问题中加入一种新类型的限制，需要某个任务必须在指定的时间点之前开始，即指定和另一个任务的开始时间的相对时间。这种类型的限制条件在争分夺秒的生产线上以及许多其他应用中都很常见，但它也会使得任务调度问题更难解决。例如，如表 4.4.6 所示，假设要在前面的示例中加入一个限制条件，使 2 号任务必须在 4 号任务启动后的 12 个时间单位之内开始。实际上，在这里最后期限限制的是 4 号任务的开始时间：它的开始时间不能早于 2 号任务开始 12 个时间单位。在示例中，调度表中有足够的空档来满足这个最后期限限制：我们可以令 4 号任务开始于 111 时间，即 2 号任务计划开始时间前的 12 个时间单位处。需要注意的是，如果 4 号任务耗时很长，这个修改可能会延长整个调度计划的完成时间。同理，如果再添加一个最后期限的限制条件，令 2 号任务必须在 7 号任务启动后的 70 个时间单位内开始，还可以将 7 号任务的开始时间调整到 53，这样就不用修改 3 号任务和 8 号任务的计划开始时间。但是如果继续限制 4 号任务必须在零号任务启动后的 80 个时间单位内开始，那么就不存在可行的调度计划了：限制条件 4 号任务必须在 0 号任务启动后的 80 个时间单位内开始以及 2 号任务必须在 4 号任务启动后的 12 个时间单位之内开始，意味着 2 号任务必须在 0 号任务启动后的 93 个时间单位之内开始，但因为存在任务链 0（41 个时间单位）→ 9（29 个时间单位）→ 6（21 个时间单位）→ 8（32 个时间单位）→ 2，2 号任务最早也只能在 0 号任务

启动后的 123 个时间单位之内开始如表 4.4.7 所示。最后期限的限制越多，调度的可能性也就越多，简单的问题也会变得越困难。

表 4.4.7 向任务调度问题中添加的最后期限限制

任务	相对最后期限	相对于任务
2	12.0	4
2	70.0	7
4	80.0	0

命题 V。相对最后期限限制下的并行任务调度问题是一个加权有向图中的最短路径问题（可能存在环和负权重边）。

证明。与命题 U 一样根据任务调度的描述构造相同的加权有向图，为每条最后期限限制添加一条边：如果任务 v 必须在任务 w 启动后的 d 个时间单位内开始，则添加一条从 v 指向 w 的负权重为 d 的边。将所有边的权重取反即可将该问题转化为一个最短路径问题。如果存在可行的调度方案，证明也就完成了。你将会看到，判断一个调度方案是否可行也是计算的一部分。

这个示例说明了负权重的边在实际应用的模型中也能起到重要的作用。它说明，如果能够有效解决负权重边的最短路径问题，那就能够找到相对最后期限限制下的并行任务调度问题的解决方案。我们已经学习过的算法都无法完成这个任务：Dijkstra 算法只适用于正（或零）权重的边，算法 4.10 要求有向图是无环的。下面我们来看看如何解决含有负权重且不一定是无环的有向图中的最短路径问题（请见图 4.4.18）。

表 4.4.6 相对最后期限限制下的任务调度

原始问题

任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	70.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2号任务必须在4号任务启动后的12个时间单位之内开始

任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	**111.0**
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2号任务必须在7号任务启动后的70个时间单位之内开始

任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	**53.0**
8	91.0
9	41.0

4号任务必须在0号任务启动后的80个时间单位之内开始

调度方案不存在

4.4.6 一般加权有向图中的最短路径问题

刚才讨论过的最后期限限制下的任务调度问题告诉我们负权重的边并不仅仅是一个数学问题。相反，它能够极大地扩展解决最短路径问题的模型的应用范围。接下来，考虑既可能含有环也可能含有负权重的边的加权有向图中的最短路径算法。在开始之前，先来学习一下这种有向图的基本性质以更新我们对最短路径的认识。图 4.4.19 是一个小小的示例，展示的是负权重的边对有向图中的最短路径的影响。也许最明显的改变就是当存在负权重的边时，权重较小的路径含有的边可能会比权重较大的路径更多。在只存在正权重的边时，我们的重点在于寻找近路；但当存在负权重的边时，我们可能会为了经过负权重的边而绕弯。这种效应使得我们要将查找“最短”路径的感觉转变为对算法本质的理解。因此需要抛弃直觉并在一个简单、抽象的层面上考虑这个问题。

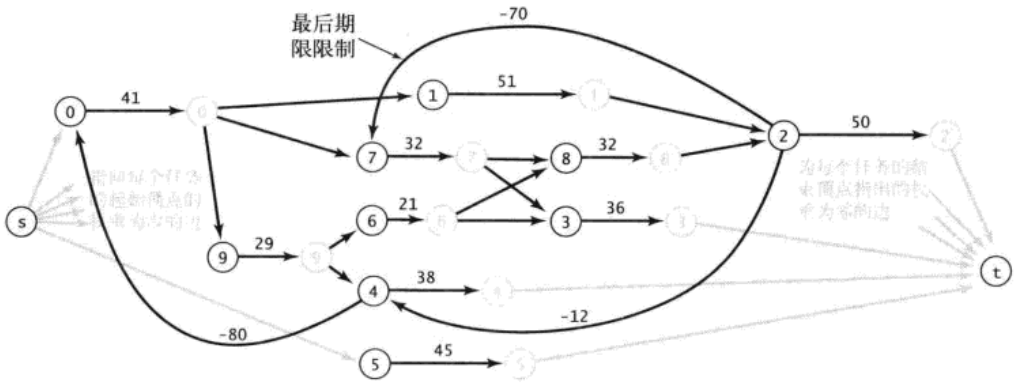


图 4.4.18　相对最后期限限制和优先级限制下的并行任务调度问题的加权有向图表示

4.4.6.1　尝试 I

第一个想法是先找到权重最小（最小的负值）的边，然后将所有边的权重加上这个负值的绝对值，这样原有向图就转变称为了一幅不含有负权重边的有向图。这种天真的做法不会解决任何问题，因为新图中的最短路径和原图中的最短路径毫无关系。路径中的边越多，这种变换产生的危害越大（请见练习 4.4.14）。

4.4.6.2　尝试 II

第二个想法是尝试改造 Dijkstra 算法。这种方法最根本的缺陷在于原算法的基础在于根据距离起点的远近依次检查路径。命题 R 对算法正确性的证明是基于添加一条边会使的路径变得更长的假设。但添加任意负权重的边只会使得路径更短，因此这个假设是不成立的（请见练习 4.4.14）。

4.4.6.3　负权重的环

666 ~ 668

当我们在研究含有负权重边的有向图时，如果该图中含有一个权重为负的环，那么最短路径的概念就失去意义了。例如图 4.4.20，除了边 5→4 的权重为 −0.66 外，它和第一个示例完全相同。这里，环 4→7→5→4 的权重为：

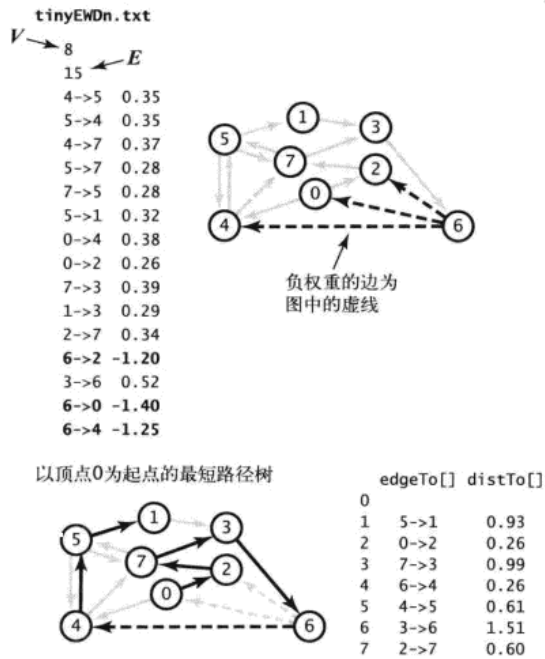


图 4.4.19　含有负权重的边的加权有向图

$$0.37+0.28-0.66=-0.01$$

我们只要围着这个环兜圈子就能得到权重任意短的路径！注意，有向环的所有边的权重并不一定都必须是负的，只要权重之和是负的即可。

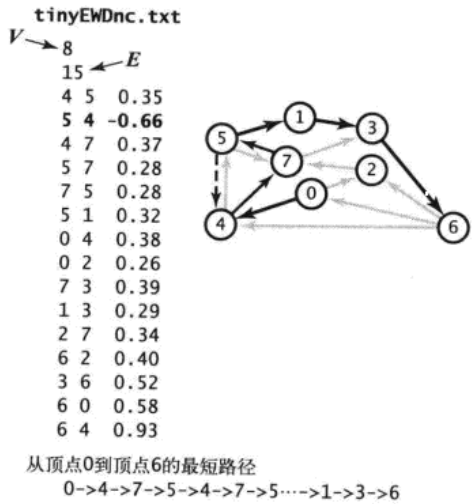
定义。加权有向图中的**负权重环**是一个总权重（环上的所有边的权重之和）为负的有向环。

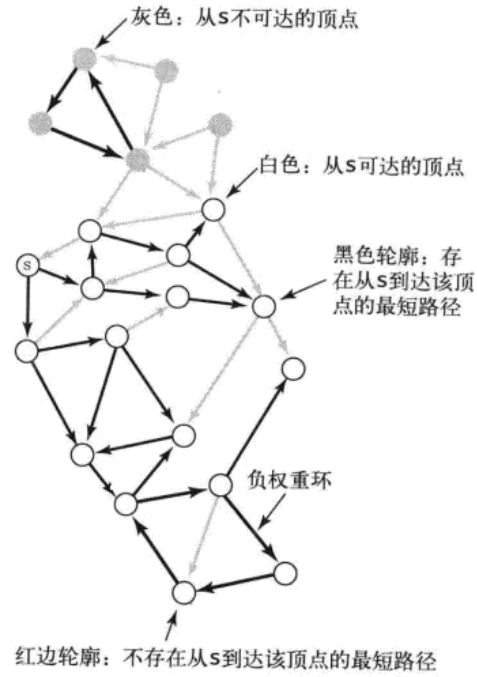
现在，假设从 s 到可达的某个顶点 v 的路径上的某个顶点在一个负权重环上。在这种情况下，从 s 到 v 的最短路径是不可能存在的，因为可以用这个负权重环构造权重任意小的路径。换句话说，在负权重环存在的情况下，最短路径问题是没有意义的，如图 4.4.21 所示。

命题 W。当且仅当加权有向图中至少存在一条从 s 到 v 的有向路径且所有从 s 到 v 的有向路径上的任意顶点都不存在于任何负权重环中时，s 到 v 的最短路径才是存在的。

证明。请见以上讨论以及练习 4.4.29。

注意，要求最短路径上的任意顶点都不存在负权重环意味着最短路径是简单的，而且与正权重边的图一样都能够得到此类顶点的最短路径树。





669

图 4.4.20 含有负权重环的加权有向图（另见彩插） 图 4.4.21 最短路径问题的各种可能性（另见彩插）

4.4.6.4 尝试III

无论是否存在负权重环，从 s 到可达的其他顶点的一条最短的简单路径都是存在的。为什么不定义最短路径以方便寻找呢？不幸的是，已知解决这个问题的最好算法在最坏情况下所需的时间是指数级别的（请见第 6 章）。一般来说，我们认为这种问题“太难了”，只会研究它的简单版本。

因此，一个定义明确且可以解决加权有向图最短路径问题的算法要能够：

- 对于从起点不可达的顶点，最短路径为正无穷（$+\infty$）；
- 对于从起点可达但路径上的某个顶点属于一个负权重环的顶点，最短路径为负无穷（$-\infty$）；
- 对于其他所有顶点，计算最短路径的权重（以及最短路径树）。

从本节的开始，我们会不断为最短路径问题加上各种限制并找到解决相应问题的办法。首先，

我们不允许负权重边的存在；其次不接受有向环。现在我们放宽所有这些条件并重点解决一般有向图中的以下问题。

负权重环的检测。给定的加权有向图中含有负权重环吗？如果有，找到它。

负权重环不可达时的单点最短路径。给定一幅加权有向图和一个起点 s 且从 s 无法到达任何负权重环，回答“是否存在一条从 s 到给定的顶点 v 的有向路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

总结。尽管在含有环的有向图中最短路径是一个没有意义的问题，而且也无法有效解决在这种有向图中高效找出最短简单路径的问题，在实际应用中仍然需要能够识别其中的负权重环。例如，在最后期限限制下的任务调度问题中，负权重环的出现可能相对较少：限制条件和最后期限都是从现实世界中的实际限制得来的，因此负权重环大多可能来自于问题陈述中的错误。找出负权重环，改正相应的错误，找到没有负权重环问题的调度方案才是解决问题的正确方式。在其他情况下，找到负权重环就是计算的目标。下面这个由 R.Bellman 和 L.Ford 在 20 世纪 50 年代末期发明的算法能
670 够简明、有效地解决这些问题并且同样适用于正权重边的有向图。

命题 X（Bellman-Ford 算法）。在任意含有 V 个顶点的加权有向图中给定起点 s，从 s 无法到达任何负权重环，以下算法能够解决其中的单点最短路径问题：将 distTo[s] 初始化为 0，其他 distTo[] 元素初始化为无穷大。以任意顺序放松有向图的所有边，重复 V 轮。

证明。对于从 s 可达的任意顶点 t，考虑从 s 到 t 的一条最短路径：$v_0 \to v_1 \to \cdots \to v_k$，其中 v_0 等于 s，v_k 等于 t。因为负权重环是不可达的，这样的路径是存在的且 k 不会大于 $V-1$。我们会通过归纳法证明算法在第 i 轮之后能够得到 s 到 v_i 的最短路径。最简单的情况（$i=0$）很容易。假设对于 i 命题成立，那么 s 到 v_i 的最短路径即为 $v_0 \to v_1 \to \cdots \to v_i$，distTo[$v_i$] 就是这条路径的长度。现在，我们在第 i 轮中放松所有的顶点，包括 v_i，因此 distTo[v_{i+1}] 不会大于 distTo[v_i] 与边 $v_i \to v_{i+1}$ 的权重之和。在第 i 轮放松之后，distTo[v_{i+1}] 必然等于 distTo[v_i] 与边 $v_i \to v_{i+1}$ 的权重之和。它不可能更大，因为在第 i 轮中放松了所有顶点，包括 v_i；它也不可能更小，因为它就是路径 $v_0 \to v_1 \to \cdots \to v_{i+1}$ 的长度，也就是最短路径了。因此，在 $i+1$ 轮之后算法能够得到从 s 到 v_{i+1} 的最短路径。

命题 W（续）。Bellman-Ford 算法所需的时间和 EV 成正比，空间和 V 成正比。

证明。在每一轮中算法都会放松 E 条边，共重复 V 轮。

这个方法非常通用，因为它没有指定边的放松顺序。下面将注意力集中在一个通用性稍逊的方法上，其中只放松从任意顶点指出的所有边（顺序任意），以下代码说明了这种方法的简洁性：

```
for (int pass = 0; pass < G.V(); pass++)
   for (v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

我们不会仔细研究这个版本，因为它总是会放松 VE 条边且只需稍作修改即可使算法在一般的
671 应用场景中更加高效。

4.4.6.5 基于队列的 Bellman-Ford 算法

其实，根据经验我们很容易知道在任意一轮中许多边的放松都不会成功：只有上一轮中的 distTo[] 值发生变化的顶点指出的边才能够改变其他 distTo[] 元素的值。为了记录这样的顶点，我们使用了一条 FIFO 队列。算法在处理正权重标准样图中进行的操作如图 4.4.22 所示。在示意图 4.4.22 左侧是每一轮中队列中的有效顶点（红色），紧接着是下一轮中的有效顶点（黑色）。首先将起点加入队列，然后按照以下步骤计算最短路径树。

- 放松边 1→3 并将顶点 3 加入队列。
- 放松边 3→6 并将顶点 6 加入队列。
- 放松边 6→4、6→0 和 6→2 并将顶点 4、0 和 2 加入队列。
- 放松边 4→7、4→5 并将顶点 7 和 4 加入队列。放松已经失效的边 0→4 和 0→2。然后再放松边 2→7（并重新为 4→7 着色）。
- 放松边 7→5（并重新为 4→5 着色）但不将顶点 5 加入队列（它已经在队列之中了）。放松已经失效的边 7→3。然后放松已经失效的边 5→1、5→4 和 5→7。此时队列为空。

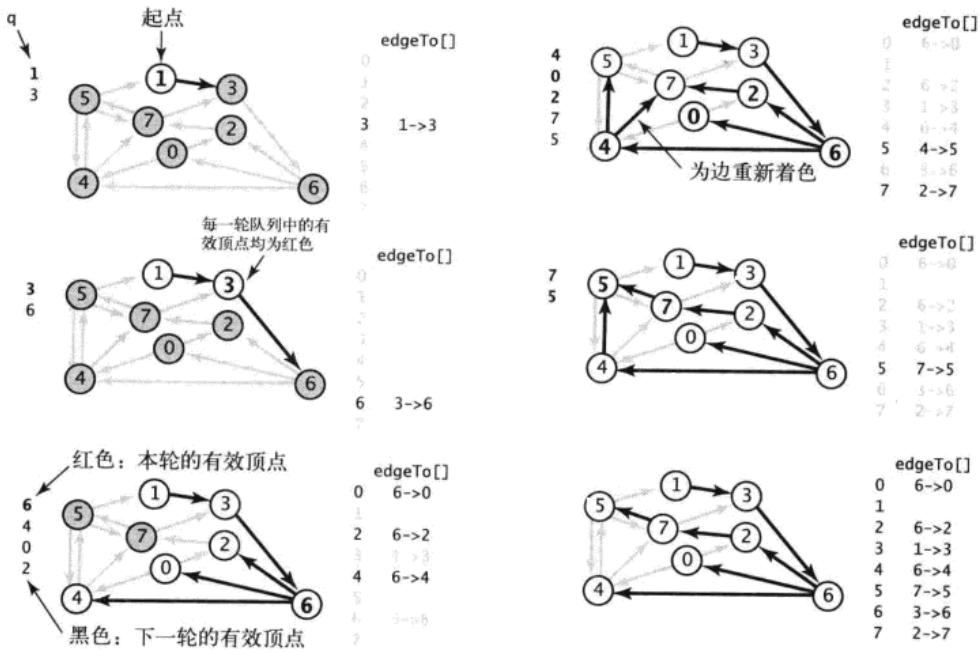


图 4.4.22 Bellman-Ford 算法的轨迹（另见彩插）

4.4.6.6 实现

根据这些描述实现 Bellman-Ford 算法所需的代码非常少，如算法 4.11 所示。它基于以下两种其他的数据结构：

- 一条用来保存即将被放松的顶点的队列 q；
- 一个由顶点索引的 boolean 数组 onQ[]，用来指示顶点是否已经存在于队列中，以防止将顶点重复插入队列。

672

首先，将起点s加入队列中，然后进入一个循环，其中每次都从队列中取出一个顶点并将其放松。要将一个顶点插入队列，需要修改4.4.2.4节框注“边的松驰”中relax()方法的实现，以便将被成功放松的边所指向的顶点加入队列中，如右边框注“Bellman-Ford算法中的放松操作”所示。这些数据结构能够保证：

- 队列中不出现重复的顶点；
- 在某一轮中，改变了edgeTo[]和distTo[]的值的所有顶点都会在下一轮中处理。

```
private void relax(EdgeWeightedDigraph G, int v)
{
   for (DirectedEdge e : G.adj(v))
   {
      int w = e.to();
      if (distTo[w] > distTo[v] + e.weight())
      {
         distTo[w] = distTo[v] + e.weight();
         edgeTo[w] = e;
         if (!onQ[w])
         {
            queue.enqueue(w);
            onQ[w] = true;
         }
      }
      if (cost++ % G.V() == 0)
         findNegativeCycle();
   }
}
```

Bellman-Ford算法中的放松操作

要完整地实现该算法，我们就需要保证在V轮后算法能够终止。实现它的一种方法是显式记录放松的轮数。我们的实现BellmanFordSP（算法4.11）使用了另一种方法，将会在4.4.6.8节详述：它会在有向图的edgeTo[]中检测是否存在负权重环，如果找到则结束运行。

命题Y。 对于任意含有V个顶点的加权有向图和给定的起点s，在最坏情况下基于队列的Bellman-Ford算法解决最短路径问题（或者找到从s可达的负权重环）所需的时间与EV成正比，空间和V成正比。

证明。如果不存在从s可达的负权重环，算法会根据命题X在进行V–1轮放松操作后结束（因为所有最短路径含有的边数都小于V–1）。如果的确存在一个从s可达的负权重环，那么队列永远不可能为空。根据命题X，在第V轮放松之后，edgeTo[]数组必然会包含一条含有一个环的路径（从某个顶点w回到它自己）且该环的权重必然是负的。因为w会在路径上出现两次且s到w的第二次出现处的路径长度小于s到w的第一次出现的路径长度。在最坏情况下，该
673 算法的行为和通用算法相似并会将所有的E条边全部放松V轮。

算法4.11 基于队列的Bellman-Ford算法

```
public class BellmanFordSP
{
   private double[] distTo;                  // 从起点到某个顶点的路径长度
   private DirectedEdge[] edgeTo;            // 从起点到某个顶点的最后一条边
   private boolean[] onQ;                    // 该顶点是否存在于队列中
   private Queue<Integer> queue;             // 正在被放松的顶点
   private int cost;                         // relax()的调用次数
   private Iterable<DirectedEdge> cycle;     // edgeTo[]中的是否有负权重环
```

```
public BellmanFordSP(EdgeWeightedDigraph G, int s)
{
   distTo = new double[G.V()];
   edgeTo = new DirectedEdge[G.V()];
   onQ = new boolean[G.V()];
   queue = new Queue<Integer>();
   for (int v = 0; v < G.V(); v++)
      distTo[v] = Double.POSITIVE_INFINITY;
   distTo[s] = 0.0;
   queue.enqueue(s);
   onQ[s] = true;
   while (!queue.isEmpty() && !this.hasNegativeCycle())
   {
      int v = queue.dequeue();
      onQ[v] = false;
      relax(G, v);
   }
}

private void relax(EdgeWeightedDigraph G,v)
// 4.4.6.6节框注 "Bellman-Ford算法的放松操作"

public double distTo(int v)          // 最短路径树实现中的标准查询算法（请见4.4.2.6节“最短路径API的查询方法”）

public boolean hasPathTo(int v)
public Iterable<Edge> pathTo(int v)  // 4.4.6.8节框注 “Bellman-Ford” 的负权重检测方法

private void findNegativeCycle()
public boolean hasNegativeCycle()
public Iterable<Edge> negativeCycle()

// 请见6.4.6.8节 .
}
```

Bellman-Ford 算法的实现修改了 relax() 方法，将被成功放松的边指向的所有顶点加入到一条 FIFO 队列中（以避免出现重复顶点）并周期性地检查 edgeTo[] 表示的子图中是否存在负权重环（请见正文）。

674

基于队列的 Bellman-Ford 算法能够准确有效地解决最短路径问题并且在实际中被广泛应用，甚至包括正权重的情况。例如，如图 4.4.23 所示，在含有 250 个顶点的样图中，算法进行了 14 轮操作且对于相同的问题比较路径长度的次数少于 Dijkstra 算法。

4.4.6.7 负权重的边

图 4.4.24 显示了 Bellman-Ford 算法在处理含有负权重边的有向图的轨迹。首先将起点加入队列 q，然后按照以下步骤计算最短路径树。

- 放松边 0→2 和 0→4 并将顶点 2、4 加入队列。
- 放松边 2→7 并将顶点 7 加入队列。放松边 4→5 并将顶点 5 加入队列。然后放松失效的边 4→7。
- 放松边 7→3 和 5→1 并将顶点 3 和 1 加入队列。放松失效的边 5→4 和 5→7。
- 放松边 3→6 并将顶点 6 加入队列。放松失效的边 1→3。
- 放松边 6→4 并将顶点 4 加入队列。这条负权重边使得到顶点 4 的路径变短，因此它的边需要被再次放松（它们在第二轮中已经被放松过）。从起点到顶点 5 和 1 的距离已经失效并会在下一轮中修正。