

## 14.5 有向非循环图

没有有向循环的有向图在许多应用中都能遇到。这样的有向图经常被叫作有向非循环图，或者简称为 DAG。这样的图的应用主要包括以下几个方面：

- 学士学位课程之间的选修课程。
- 面向对象程序的类之间的继承。
- 工程任务之间的行程安排的约束条件。

我们在下面的例子中对最后一个应用进行了更深的探讨。

**例题 14-20:** 为了管理一个巨大的工程，将它分解为更小的任务的集合是非常方便的。然而，任务之间很少是独立的，因为任务之间存在行程安排的约束条件。(例如，在房屋建筑的工程中，订购钉子的任务明显在订购露天平台屋顶的瓦片之前。) 明显地，行程计划的约束条件没有循环，因为那将会使得工程变得不可能。(例如，为了获得工作你需要去获得工作经验，但是为了获得工作经验你又必须去找到工作。) 行程安排的限制条件在任务能够被履行的命令下强加了限制条件。也就是说，如果限制规定任务  $a$  必须在任务  $b$  开始之前完成，那么在任务执行的顺序中， $a$  必须在  $b$  之前。因此，如果我们将任务的可行的集合建模为一个有向图的顶点，那么无论是否对  $u$  的任务必须在对  $v$  的任务之前执行，我们都要放置一个有向边，然后定义一个有向非循环图。

### 拓扑排序

上述例子产生了下面的定义。定义  $\vec{G}$  是有  $n$  个顶点的有向图。 $\vec{G}$  的拓扑排序是对  $\vec{G}$  的每条边  $(v_i, v_j)$  来说  $\vec{G}$  的顶点的顺序  $v_1, \dots, v_n$ ，这种情况下  $i < j$ 。也就是说，拓扑排序是一种排序，使得  $\vec{G}$  的任何有向路径以增加的顺序遍历顶点。需要注意的是一个有向图可能不止有一个拓扑排序（见图 14-12）。

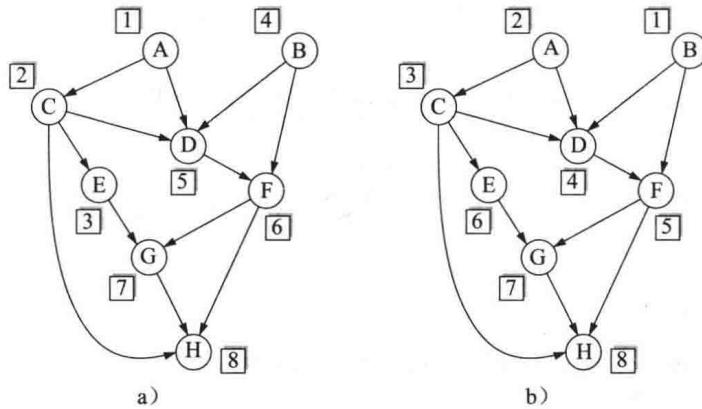


图 14-12 相同非循环有向图的两种拓扑排序

**命题 14-21:**  $\vec{G}$  有一个拓扑排序当且仅当它是非循环的。

**证明:** 这个必要性（声明中的“当且仅当”部分）非常容易论证。假设  $\vec{G}$  具有拓扑排序。假设（为了反驳） $\vec{G}$  由边  $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_{k-1}}, v_{i_0})$  的组合能构成循环。因为拓扑排序，我们必须有  $i_0 < i_1 < \dots < i_{k-1} < i_0$ ，这明显是不可能的。因此， $\vec{G}$  必须是非循环的。

我们现在考虑条件的充分性（“如果”部分）。假设  $\vec{G}$  是非循环的。我们将会为如何为  $\vec{G}$  建立拓扑排序给出算法说明。因为  $\vec{G}$  是非循环的， $G$  必须有一个没有输入边的顶点（即入度为 0）。定义  $v_l$  为这样的一个顶点。事实上，如果  $v_l$  不存在，那么从任意的开始顶点追踪一

个有向路径，我们最终会遇到先前经过的顶点，因此否定了 $\vec{G}$ 的无循环性。如果我们从 $\vec{G}$ 中移除 $v_1$ 和它的传出边，产生的有向图依旧是无循环的。因此，产生的有向图同样有没有传入边的顶点，然后我们让 $v_2$ 成为这样的顶点。通过重复这个进程直到有向图变成空的，我们获得了 $\vec{G}$ 的顶点的顺序 $v_1, \dots, v_n$ 。因为上述解释，如果 $(v_i, v_j)$ 是 $\vec{G}$ 的一条边，那么在 $v_j$ 可以被删除之前， $v_i$ 必须被删除，因此， $i < j$ 。所以， $v_1, \dots, v_n$ 是一个拓扑排序。■

命题 14-21 的证明为有向图计算拓扑顺序提供了一种算法，我们称为拓扑排序。在代码段 14-11 中展示了这项技术的 Python 实现，图 14-13 展示了该算法执行的例子。我们使用名为 incount 的字典来实现，将每一个顶点 $v$ 映射到展示了 $v$ 的输入边的当前数目的计数器上，不包括那些先前被加到拓扑顺序的顶点。专门地，一个 Python 字典提供  $O(1)$  的预期时间去使用每一项，而不是最坏情况下的时间。这和图的遍历一样，如果顶点从 0 到  $n - 1$  被索引，或者如果我们存储计数器作为顶点的一个元素，这将会转换成最坏情况下的时间。

#### 代码段 14-11 拓扑排序算法的 Python 实现（我们在图 14-13 中展示了该算法的例子）

```

1 def topological_sort(g):
2     """Return a list of vertices of directed acyclic graph g in topological order.
3
4     If graph g has a cycle, the result will be incomplete.
5     """
6     topo = []          # a list of vertices placed in topological order
7     ready = []          # list of vertices that have no remaining constraints
8     incount = {}        # keep track of in-degree for each vertex
9     for u in g.vertices():
10         incount[u] = g.degree(u, False)    # parameter requests incoming degree
11         if incount[u] == 0:               # if u has no incoming edges,
12             ready.append(u)              # it is free of constraints
13     while len(ready) > 0:
14         u = ready.pop()                # u is free of constraints
15         topo.append(u)                # add u to the topological order
16         for e in g.incident_edges(u):   # consider all outgoing neighbors of u
17             v = e.opposite(u)
18             incount[v] -= 1            # v has one less constraint without u
19             if incount[v] == 0:
20                 ready.append(v)
21
22 return topo

```

作为一种副作用，代码段 14-11 的拓扑排序算法同样测试是否已知的有向图 $\vec{G}$ 是非循环的。事实上，如果算法没有对所有顶点进行排序就结束了，那么没有被排序的顶点的子图必须包含一个有向循环。

#### 拓扑排序的性能

**命题 14-22：**定义 $\vec{G}$ 是一个有 $n$ 个顶点和 $m$ 条边的使用邻接列表结构表示的有向图。拓扑排序算法使用了 $O(n)$ 的辅助空间，运行时间是 $O(n + m)$ ，并且计算 $\vec{G}$ 的拓扑顺序或者加入一些顶点失败，表明 $\vec{G}$ 中存在有向循环。

**证明：**入度为 $n$ 的原始记录基于 degree 算法使用了 $O(n)$ 的时间。也就是说当 $u$ 从 ready 列表中移除的时候，顶点 $u$ 被拓扑排序访问了。顶点 $u$ 仅仅当 $\text{incount}(u)$ 为 0 时被访问，并且任何其他的顶点恰好被访问一次。该算法遍历了每次访问的每个顶点的所有传出边，因此它的运行时间和被访问的顶点的传出边的数目成正比。和命题 14-9 一致，运行时间是 $(n + m)$ 。至于空间的使用，观察到容器 topo、ready 和 incount 的每个顶点至多有一项，因此使用了 $O(n)$ 的空间。■

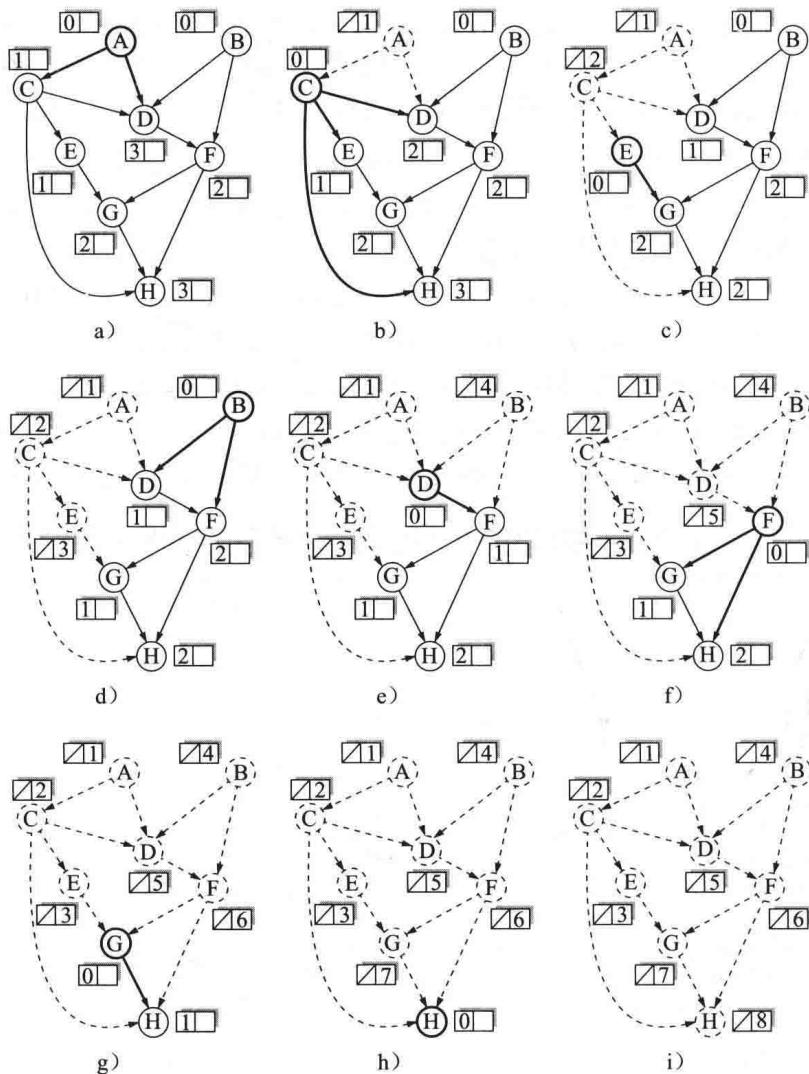


图 14-13 `topological_sort` (代码段 14-11) 运行的例子。顶点附近的标签展示了它当前的 `incount` 值，以及在产生的拓扑顺序中的最终排名。突出的顶点是将会变成拓扑顺序中的下一个顶点的 `incount` 等于 0 的顶点。虚线表示已经被检查过并且不再反映在 `incount` 值中的边。

## 14.6 最短路径

正如我们在 14.3.3 节看到的，广度优先搜索策略可以用来在连通图中从一些开始顶点到每一个其他顶点寻找最短路径。这个途径在每条边和其他任何一条边一样好的情况下有意义，但是也有许多这个途径并不恰当的情况。

例如，我们可能想使用图去表示城市间的路，我们可能对找到旅行穿越城市的最快路径很感兴趣。在这种情况下，所有的边彼此相等可能并不合适，因为一些城际的距离可能比其他的大许多。同样，我们可以使用图来表示网络通信（例如互联网），我们可能对在两台计算机之间找到最快的路径并按该路线发送数据包很感兴趣。在这种情况下，所有的边彼此相等可能就不是很正确了，因为计算机网络中的一些连接通常比其他（例如，一些边可能代表低带宽的连接，而其他可能代表高速、光纤的连接）连接快很多。因此，考虑那些边的权重并不相等的图是很自然的。

### 14.6.1 加权图

加权图是一种有和每条边  $e$  相关联的数值的（例如，数字）标签的图，这个数字标签称为边  $e$  的权重。对于  $e = (u, v)$ ，记  $w(u, v) = w(e)$ 。我们在图 14-14 中展示了一个加权图的例子。

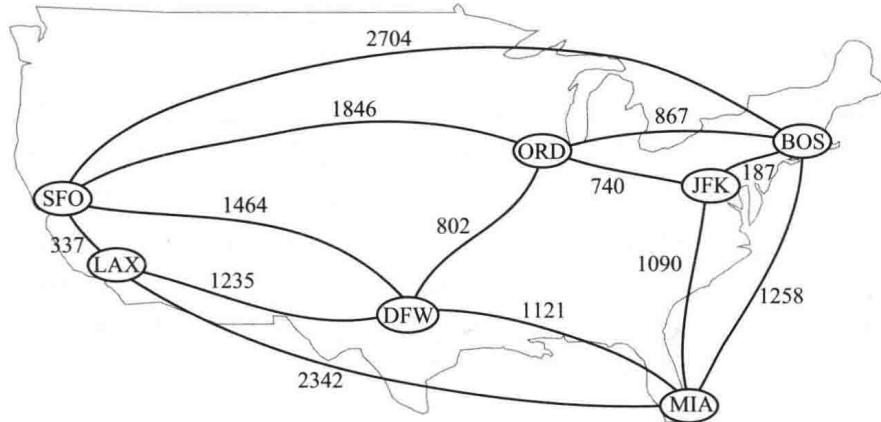


图 14-14 加权图示例，其中，顶点代表主要美国机场，边权重代表以英里为单位的距离。这个图在 JFK 到 LAX 之间有一条总权重为 2777 英里（经过 ORD 和 DFW）的路径。这是 JFK 到 LAX 在图中最小权重的路径

#### 在加权图中定义最短路径

定义  $G$  为加权图。路径的长度（或者权重）是  $P$  的边的权重的总和。即如果  $P = ((v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k))$ ，那么  $P$  的长度（表示为  $w(P)$ ）被定义为

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

在图中顶点  $u$  到顶点  $v$  之间的距离表示为  $d(u, v)$ ，是从  $u$  到  $v$  之间长度最短的路径（也称为最短路径），如果这样的路径存在的话。

人们经常使用约定：如果在  $G$  中从  $u$  到  $v$  之间没有任何路径，则  $d(u, v) = \infty$ 。即使在  $G$  中从  $u$  到  $v$  有路径，然而，如果在  $G$  中有总权重为负的循环，则  $u$  到  $v$  的距离可能没有定义。例如，假设顶点在  $G$  中表示城市， $G$  中边的权重表示从一个城市去另一个城市需要花费多少钱。如果有人愿意实际支付从 JFK 到 ORD 的费用，那么边 (JFK, ORD) 的“费用”是负的。如果另外一些人愿意支付从 ORD 去 JFK 的费用，那么在  $G$  中会有一个负权重的循环并且距离不会再被定义。即任何人现在都可在图中从任何城市 A 到另一个城市 B 之间建一条路径（有循环）：首先去 JFK，并且在去 B 之前，循环和他想从 JFK 到 ORD 然后回来的次数一样多。这样的路径允许我们建立任意低的负消费的路径（并且在过程中获得收益）。但是距离不可以是任意低的负的数字。因此，我们随时可以使用边的权重去表示距离，必须小心不要引入任何负权重的循环。

假设给定了一个加权图  $G$ ，我们要求寻找从一些顶点  $s$  到  $G$  中的其他顶点的最短路径，将边的权重看作距离。在本节，我们探索寻找所有这样的最短路径的高效方式（如果它们存在的话）。我们讨论的第一个算法非常简单，并且很常见，假设当  $G$  中所有的边的权重是非负的（即对每一个  $G$  中的边  $e$  都有  $w(e) \geq 0$ ），因此，我们可以提前知道在  $G$  中没有负权重的循环。当所有的权重和呈现在 14.3.3 节的 BFS 遍历算法解决的一样的时候，即得计算最短路径的特殊情况。

这对解决依赖于贪心算法设计模式（13.4.2 节）的唯一来源问题是一个有趣的进展。记得在这个模式中我们通过重复地在每个可用的迭代中做出最好的选择来解决该问题。这个范例经常用于当我们试图在一些对象的集合中优化代价函数的情况。我们可以在集合中添加目标，一次添加一个，并且总是选择下一个优化那些尚未被选择的目标。

### 14.6.2 Dijkstra 算法

将贪心算法模式应用于单源最短路径的主要思想是从源顶点开始执行“加权”广度优先算法。特别地，我们可以使用贪心算法去开发一个算法，该算法迭代地从  $s$  中增加顶点的“云”，其中顶点按照它们与  $s$  的距离的顺序进入云。因此，在每次迭代中，下一个被选择的顶点是和  $s$  很接近的云之外的顶点。当不再有顶点在云之外（或者云之外的顶点不再和云之内的有连接），并且从  $s$  到  $G$  的每一个从  $s$  开始可达的顶点都有最短的路径的时候，该算法就结束了。这个方法非常简单，但是很强大，是贪心设计模式的例子。对单源应用贪心算法时，最短路径问题产生了 Dijkstra 算法。

#### 边的逐次近似

我们对  $V$  中的每个顶点  $v$  定义一个标签  $D[v]$ ，用来在  $G$  中对从  $s$  到  $v$  的距离做近似估算。这些标签的意思是  $D[v]$  将会存储我们到目前为止从  $s$  到  $v$  找到的最好的路径的长度。首先，对每个  $v \neq s$ ， $D[s] = 0$  并且  $D[v] = \infty$ ，然后我们定义  $C$  集合，它是顶点的“云”，初始状态下是空集合。在算法的每次迭代中，我们选择了不在  $C$  中有最短的  $D[u]$  标签的顶点  $u$ ，然后将  $u$  放进  $C$ 。（一般来说，我们将使用优先级队列来选择云外的顶点。）在第一次迭代中，将  $s$  放进  $C$  中。一旦新顶点  $u$  被放进  $C$  中，接下来更新每个邻近  $u$  并且在  $C$  之外的顶点  $v$  的标签  $D[v]$ ，以反映这样的事实——有新的更好的方式通过  $u$  到  $v$ 。这个更新操作被称为松弛过程，因为它需要一个旧的估计并检查是否可以改进以接近其真实值。特定的边松弛操作如下：

边的逐次近似： if  $D[u] + w(u, v) < D[v]$  then  

$$D[v] = D[u] + w(u, v)$$

#### 算法的说明和例子

我们在代码段 14-12 中给出了 Dijkstra 算法的伪代码，并且在图 14-15 ~ 图 14-17 中说明了 Dijkstra 算法的一些迭代。

#### 代码段 14-12 Dijkstra 算法的伪代码，解决了单源最短路径问题

**Algorithm** ShortestPath( $G, s$ ):

**Input:** A weighted graph  $G$  with nonnegative edge weights, and a distinguished vertex  $s$  of  $G$ .

**Output:** The length of a shortest path from  $s$  to  $v$  for each vertex  $v$  of  $G$ .

Initialize  $D[s] = 0$  and  $D[v] = \infty$  for each vertex  $v \neq s$ .

Let a priority queue  $Q$  contain all the vertices of  $G$  using the  $D$  labels as keys.

**while**  $Q$  is not empty **do**

{pull a new vertex  $u$  into the cloud}

$u$  = value returned by  $Q.\text{remove\_min}()$

**for** each vertex  $v$  adjacent to  $u$  such that  $v$  is in  $Q$  **do**

{perform the *relaxation* procedure on edge  $(u, v)$ }

**if**  $D[u] + w(u, v) < D[v]$  **then**

$D[v] = D[u] + w(u, v)$

Change to  $D[v]$  the key of vertex  $v$  in  $Q$ .

**return** the label  $D[v]$  of each vertex  $v$

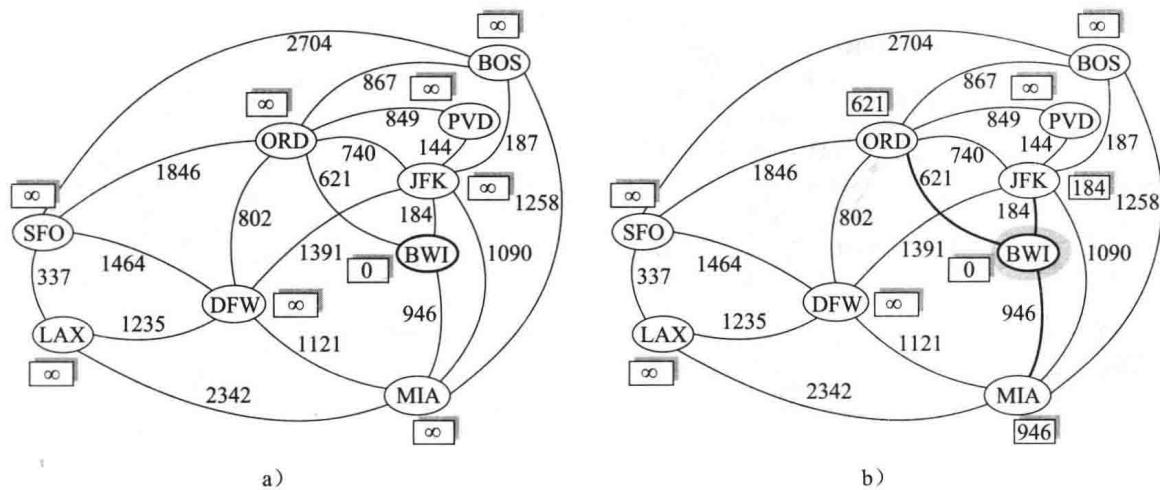


图 14-15 加权图的 Dijkstra 算法的执行。开始顶点是 BWI。每个顶点  $v$  旁边的框存储标签  $D[v]$ 。最短路径树的边被画成了粗箭头，对每个“云”之外的顶点  $u$ ，我们用粗线表示将  $u$  拉进其中的当前最好的边（下接图 14-16）

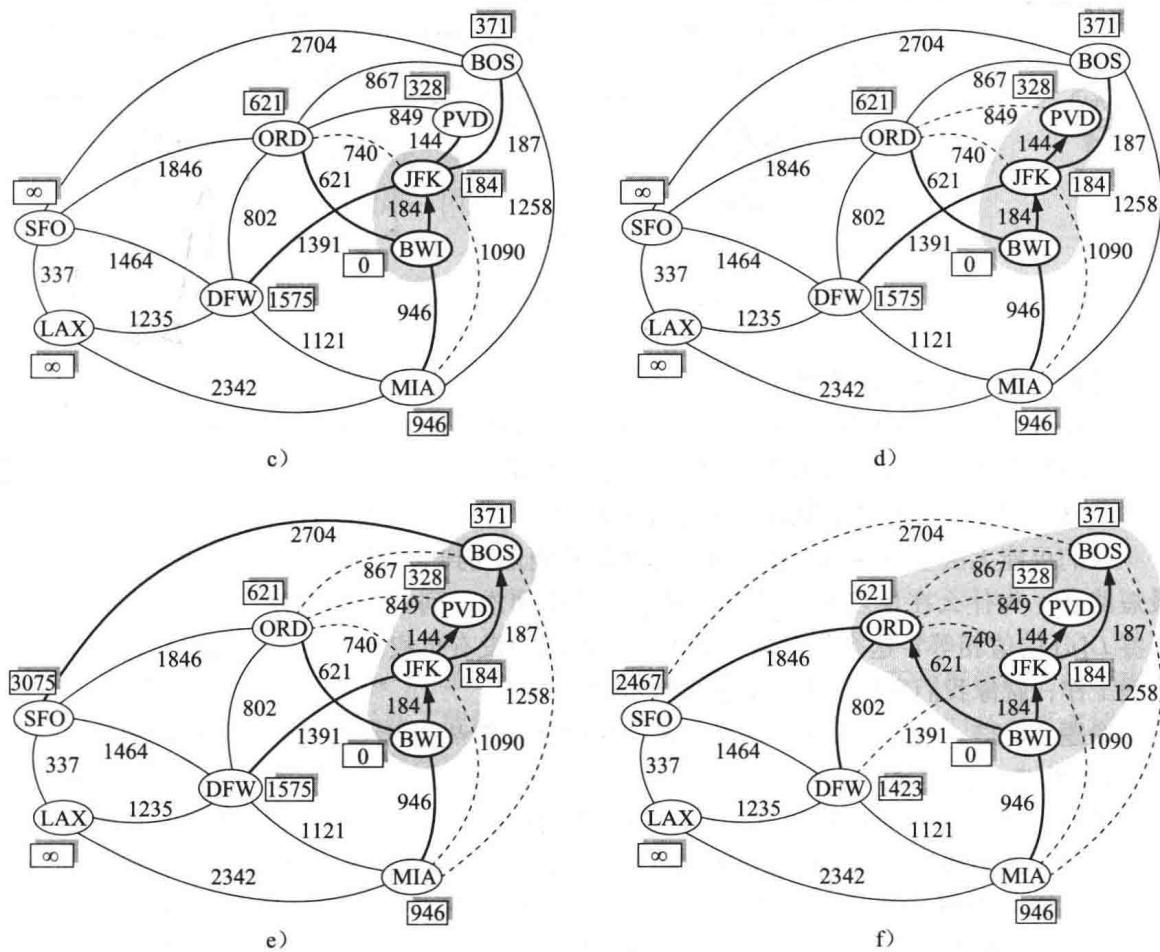


图 14-16 Dijkstra 算法的例子 (上接图 14-15, 下接图 14-17)

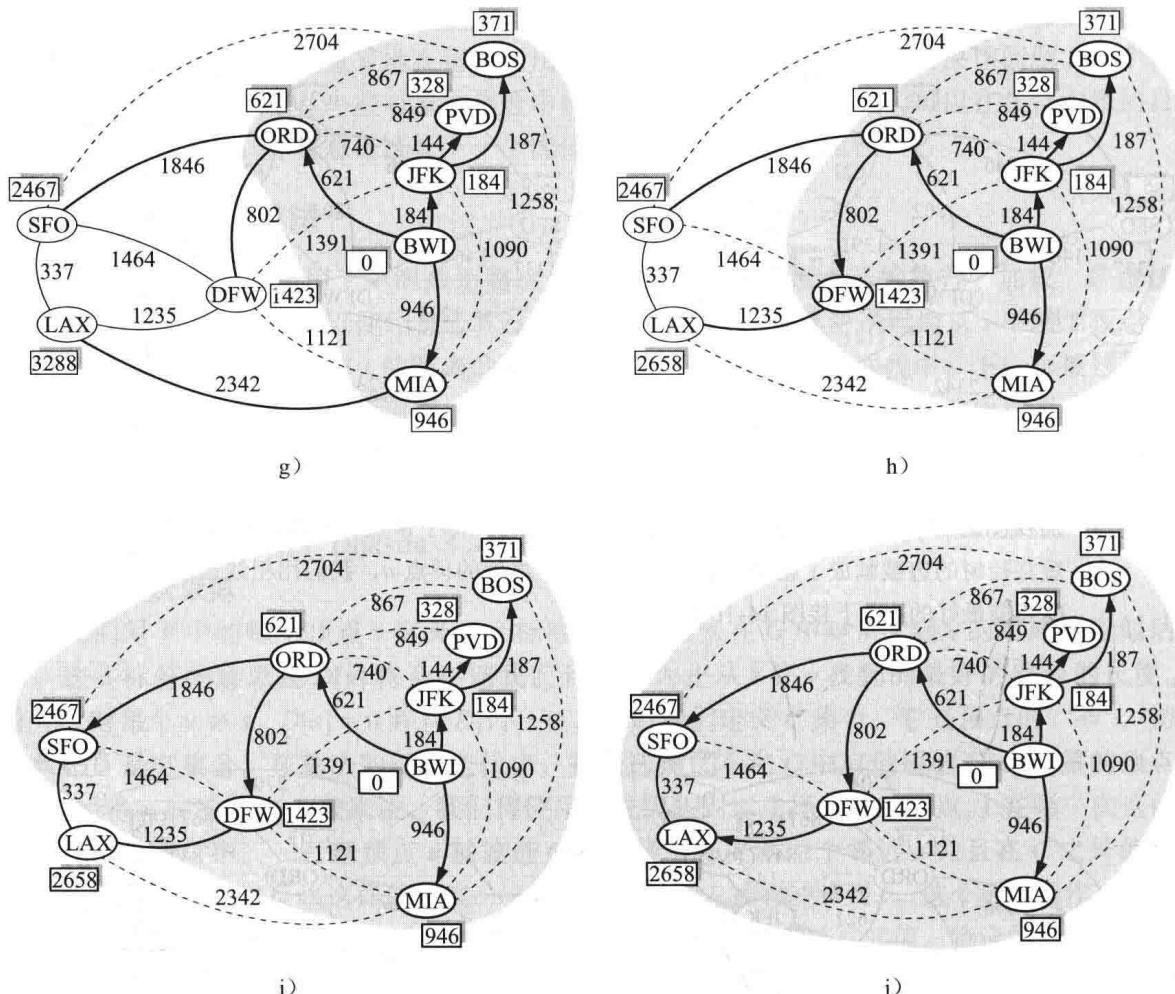


图 14-17 Dijkstra 算法的例子 (上接图 14-16)

### 为什么这种算法会起作用

Dijkstra 算法有意思的地方是，此刻一个顶点  $u$  被拉进  $C$ ，它的标签  $D[u]$  存储了从  $u$  到  $v$  的最短路径的正确长度。因此，当算法结束的时候，它计算了从  $s$  到  $G$  的每个顶点最短路径的距离。即它将会解决单源最短路径的问题。

我们可能无法立刻明白为什么这种算法能正确地找到从开始顶点  $s$  到其他每个顶点  $u$  的最短路径。为什么在顶点  $u$  从优先队列  $Q$  中移除和添加到云  $C$  的时候，从  $s$  到  $u$  的距离和标签  $D[u]$  的值相等？这个问题的答案取决于在图中没有负权重的边，因为它允许贪心算法正确工作，就像我们在接下来的命题中展示的一样。

**命题 14-23：**在 Dijkstra 算法中，无论任何时候顶点  $v$  被拉进云中，标签  $D[v]$  和从  $s$  到  $v$  的最短路径的长度  $d(s, v)$  相等。

**证明：**对在  $V$  中的一些顶点  $v$ ，假设  $D[v] > d(s, v)$ ，然后令  $z$  为算法拉进云  $C$  的第一个顶点（即从  $Q$  中移除），例如  $D[z] > d(s, z)$ 。从  $s$  到  $z$  有最短路径  $P$ （否则  $d(s, z) = \infty = D[z]$ ）。因此我们要考虑当  $z$  被拉进  $C$  的时刻，并且在这个时刻让  $y$  成为  $P$ （从  $s$  到  $z$  时）中而不是  $C$  中的第一个顶点。令  $x$  为路径  $P$  中  $y$  的前驱（注意  $x = s$ ）（见图 14-18）。我们知道，对于我们选择的  $y$ ， $x$  此时已经在  $C$  中了。

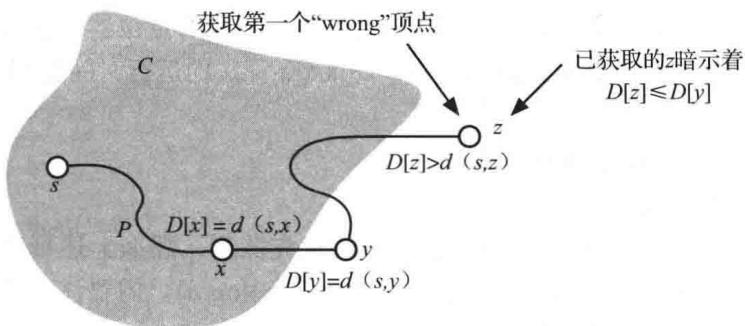


图 14-18 命题 14-23 的证明的原理图

此外,  $D[x] = d(s, x)$ , 因为  $z$  是第一个不正确的顶点。当  $x$  被拉进  $C$  时, 我们测试了  $D[y]$ , 因此目前可以得到

$$D[y] \leq D[x] + w(x, y) = d(s, x) + w(x, y)$$

但是因为  $y$  是从  $s$  到  $z$  的最短路径中的下一个顶点, 这意味着

$$D[y] = d(s, y)$$

但是我们现在处于正在获取  $z$  (不是  $y$ ) 并将其加入  $C$  的时刻, 因此

$$D[z] \leq D[y]$$

最短路径的子路径是它自己的最短路径是非常明确的。因此, 因为  $y$  在从  $s$  到  $z$  的最短路径上,

$$d(s, y) + d(y, z) = d(s, z)$$

此外, 因为没有负权重的边,  $d(y, z) \geq 0$ 。所以,

$$D[z] \leq D[y] = d(s, y) \leq d(s, y) + d(y, z) = d(s, z)$$

但是这个与  $z$  的定义相矛盾, 因此, 不可能有这样的顶点  $z$ 。 ■

### Dijkstra 算法的运行时间

在本节, 我们分析 Dijkstra 算法的时间复杂度。分别用  $n$  和  $m$  表示输出图  $G$  的顶点和边的数目。假设边的权重可以在恒定的时间内相加和比较。我们在代码段 14-12 中给出了 Dijkstra 算法的概要描述, 而分析它的运行时间需要给出更多执行细节。特别地, 我们应该指出使用的数据结构和它们是如何实现的。

我们首先假设用邻接列表或者邻接图结构表示图  $G$ 。这个数据结构允许我们在松弛步骤和它们的数量成正比期间单步调试邻近  $u$  的顶点。因此, 时间花费在嵌入的 for 循环的管理上, 该循环的迭代的次数是

$$\sum_{u \in V_G} \text{out deg}(u)$$

命题 14-9 的时间是  $O(m)$ 。外部 while 循环执行了  $O(n)$  次, 因为在每次迭代的过程中一个新的顶点被加到云里。这仍然不能解决算法分析的所有细节, 然而, 我们必须更多地说明如何实现算法中的其他主要数据结构——优先队列  $Q$ 。

回顾代码段 14-12 中搜寻优先队列的操作, 我们发现  $n$  个顶点最初就被插入优先队列里。因为这些是唯一的插入元素, 所以队列的最大长度为  $n$ 。在 while 循环的  $n$  次迭代的每次迭代中, 对 `remove_min` 的使用是为了提取具有  $Q$  中最小标签  $D$  的顶点  $u$ 。然后, 对  $u$  的每个邻居  $v$ , 我们执行边的逐次近似, 然后可以潜在地在队列中更新  $v$  的值。因此, 我们实际上需要一个适应性优先级队列的实现 (见 9.5 节), 在这种情况下, 使用方法 `update( $l, k$ )`

改变顶点  $v$  的值，其中， $l$  是与顶点  $v$  相关的优先队列条目的定位器。在最坏的情况下，需要对图的每条边进行这样的更新。总的来说，Dijkstra 算法的运行时间受到下面几项的限制：

- $n$  插入  $Q$ 。
- $n$  在  $Q$  上调用 `remove_min` 方法。
- $m$  在  $Q$  上调用 `update` 方法。

如果  $Q$  是一个被当作堆来实现的适应性强的优先队列，那么上述每个操作的运行时间为  $O(\log n)$ ，所以 Dijkstra 的全部运行时间为  $O((n + m)\log n)$ 。需要注意的是，如果我们希望将运行时间仅仅表达为  $n$  的函数，那么在最坏的情况下是  $O(n^2 \log n)$ 。

现在我们对使用未排好顺序的适应性强的优先队列考虑一个可替代的实现（见练习 P-9.58）。这当然需要我们花费  $O(n)$  的时间去提取最小元素，但是它提供了非常快速的主键更新，提供的  $Q$  支持位置感知的项（9.5.1 节）。特别地，我们可以在  $O(1)$  时间内在松弛步骤实现每个主键值的更新——一旦在  $Q$  中定位的条目更新了，就可以很容易地改变主键值。因此，这个实现产生了  $O(n^2 + m)$  的运行时间，因为  $G$  很是简单的，所以可以简化到  $O(n^2)$ 。

### 两种实现方式的比较

在 Dijkstra 算法中，我们有两种选择去实现有位置感知项的适应性强的优先队列：堆实现，它的运行时间是  $O((n + m) \log n)$ ；未排序的序列的实现，产生了  $O(n^2)$  的运行时间。这两种实现的编码相对简单，在编程复杂度方面的需求而言是相等的。这两种实现就最坏情况下的运行时间的常数因子而言同样是相等的。仅仅看这些最坏情况下的时间，当图中边的数量很小的时候（当  $m < n^2/\log n$  的时候），我们更喜欢堆实现，而当边的数量非常大的时候（ $m > n^2/\log n$ ）我们更喜欢序列实现。

**命题 14-24：**已知有  $n$  个顶点和  $m$  条边的有权图，每条边的权重是非负的，还有  $G$  的顶点  $s$ ，Dijkstra 算法计算从  $s$  到所有其他顶点的距离时最好的情况是  $O(n^2)$  或者  $O((n + m) \log n)$ 。

我们注意到一个高级优先级队列实现，称为斐波那契堆，它可以用于在  $O(m + n \log n)$  时间内实现 Dijkstra 算法。

### 用 Python 对 Dijkstra 算法进行编程

Dijkstra 算法的伪代码描述已经给出，现在我们展示执行 Dijkstra 算法的 Python 代码，假设我们给出一个边元素是非负数字权重的图。算法的实现是以函数 `shortest_path_lengths` 的形式，它把图和指定的源顶点作为参数（见代码段 14-13）。它返回一个名为 `cloud` 的字典，映射每一个从源可达的顶点  $v$  到它的最短路径距离  $d(s, v)$ 。我们依赖在 9.5.2 节开发的 `AdaptableHeapPriorityQueue` 作为一个适用性强的优先队列。

**代码段 14-13** Dijkstra 算法对从单源计算最短路径距离的 Python 实现。我们假设边  $e$  的 `e.element()` 代表那条边的权重

```

1 def shortest_path_lengths(g, src):
2     """Compute shortest-path distances from src to reachable vertices of g.
3
4     Graph g can be undirected or directed, but must be weighted such that
5     e.element() returns a numeric weight for each edge e.
6
7     Return dictionary mapping each reachable vertex to its distance from src.
8     """
9     d = {}                      # d[v] is upper bound from s to v
10    cloud = {}                   # map reachable v to its d[v] value
11    pq = AdaptableHeapPriorityQueue() # vertex v will have key d[v]
12    pqlocator = {}                # map from vertex to its pq locator

```

```

13
14    # for each vertex v of the graph, add an entry to the priority queue, with
15    # the source having distance 0 and all others having infinite distance
16    for v in g.vertices():
17        if v is src:
18            d[v] = 0
19        else:
20            d[v] = float('inf')           # syntax for positive infinity
21            pqlocator[v] = pq.add(d[v], v) # save locator for future updates
22
23    while not pq.is_empty():
24        key, u = pq.remove_min()
25        cloud[u] = key              # its correct d[u] value
26        del pqlocator[u]           # u is no longer in pq
27        for e in g.incident_edges(u):
28            v = e.opposite(u)
29            if v not in cloud:
30                # perform relaxation step on edge (u,v)
31                wgt = e.element()
32                if d[u] + wgt < d[v]:      # better path to v?
33                    d[v] = d[u] + wgt       # update the distance
34                    pq.update(pqlocator[v], d[v], v) # update the pq entry
35
36    return cloud                  # only includes reachable vertices

```

就像我们在本章用其他算法完成时一样，用字典去映射顶点到相关的数据（在这种情况下，映射  $v$  到它的距离界限  $D[v]$  和它的适应性强的优先队列定位器）。这些字典的元素期望的存取时间  $O(1)$  可以被转换成最坏情况的界限，或者通过对顶点从 0 到  $n - 1$  进行编号作为列表的索引来实现，或者通过在每个顶点元素中存储信息来实现。

Dijkstra 算法通过对除了源之外的每个  $v$  设定  $d[v] = \infty$  开始。我们用 Python 中的特殊值 `float('inf')` 来提供表示正无穷大的数值。然而，我们在通过函数返回的结果云中避免包括这个“无穷”距离的顶点。可以通过等待向优先级队列添加顶点直到到达其的边缘被放宽之后，再完全避免使用该数字限制（见练习 C-14.64）。

### 重建最短路径树

代码段 14-12 是 Dijkstra 算法的伪代码描述，代码段 14-13 是我们的实现，对每个顶点  $v$  计算值  $d[v]$ ，那是从源顶点  $s$  到  $v$  的最短路径的长度。然而，这些算法的形式不能明确地计算获得的那些距离的实际路径。从源顶点  $s$  产生的所有最短路径的集合可以被简洁地表示为最短路径树。这个路径形成了一个有根的树，因为如果从  $s$  到  $v$  的最短路径经过中间顶点  $u$ ，它必须以从  $s$  到  $u$  的最短路径开始。

在本节，我们描述了以源  $s$  为根的最短路径树可以在  $O(n + m)$  的时间内被重建，给出的  $d[v]$  值的集合由使用  $s$  作为源的 Dijkstra 引入。当我们表示 DFS 和 BFS 树的时候，将会映射每个顶点  $v \neq s$  到根  $u$ （可能  $u = s$ ），这样  $v$  在从  $s$  到  $v$  的最短路径上之前， $u$  立刻变成顶点。如果  $u$  是在从  $s$  到  $v$  的最短路径上在  $v$  之前的顶点，则必须

$$d[u] + w(u, v) = d[v]$$

相反，如果满足上述公式，那么从  $s$  到  $u$  的最短路径——跟随在边  $(u, v)$  之后的——是到  $v$  的最短路径。

在代码段 14-14 中对重建树的实现便依赖于这个逻辑，对每一个顶点  $v$  检测输入边，寻找一个  $(u, v)$  满足关键方程。运行时间是  $O(n + m)$ ，此时我们考虑每个顶点和这些边的所有输入边（见命题 14-9）。

## 代码段 14-14 重建最短路径的 Python 函数，依赖于单源距离的知识

```

1 def shortest_path_tree(g, s, d):
2     """Reconstruct shortest-path tree rooted at vertex s, given distance map d.
3
4     Return tree as a map from each reachable vertex v (other than s) to the
5     edge e=(u,v) that is used to reach v from its parent u in the tree.
6     """
7     tree = { }
8     for v in d:
9         if v is not s:
10            for e in g.incident_edges(v, False):      # consider INCOMING edges
11                u = e.opposite(v)
12                wgt = e.element()
13                if d[v] == d[u] + wgt:
14                    tree[v] = e                         # edge e is used to reach v
15
16    return tree

```

## 14.7 最小生成树

假设我们希望在一个新的建筑物中使用最少数量的电缆来连通所有的计算机。我们可以使用无向的加权图  $G$  来建模这个问题，其顶点表示计算机，边  $(u, v)$  的权重  $w(u, v)$  与需要连接计算机  $u$  和计算机  $v$  的电缆的数量相等。除了计算从一些特别的顶点  $v$  的最短路径，我们还对寻找包含  $G$  的所有顶点的树  $T$  和在所有这样的树中的最小总权重感兴趣。找到这样的树的算法是本节的焦点。

## 问题定义

已知一个无向的、有权重的图  $G$ ，我们有兴趣找到一棵树  $T$ ，它包含  $G$  中的所有顶点，并最小化总和

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

这样的包括连通图  $G$  的每个顶点的树被称为生成树，并且计算一棵有最小总权重的生成树  $T$  的问题称为最小生成树（MST）问题。最小生成树问题的高效算法的发展在时间上早于现代计算机科学本身的概念。在本节，我们讨论了两种解决 MST 问题的经典算法。这些算法都是贪心算法的应用，在前面的章节简短地讨论过，依赖于通过迭代地获得最小化一些代价函数的对象去选择目标，从而加入一个不断增长的集合。我们讨论的第一个算法是 Prim-Jarník 法，从单个根节点生成 MST，它和 Dijkstra 算法的最短路径算法有很多相似的地方。我们讨论的第二个算法是 Kruskal 算法，通过按照边的权重的非递减顺序去考虑边来成群地“生成” MST。

为了简化算法的描述，我们假设输入图  $G$  是无向（即它的所有边都是无向的）且简单的（即它没有自循环和平行边）。因此，我们将  $G$  的边表示为无序的顶点对  $(u, v)$ 。

在我们讨论这些算法的细节之前，先得出一个关于形成算法的基础的最小生成树的重要事实。

## 最小生成树的重要的事实

我们讨论的两个 MST 算法都基于贪心算法，在这种情况下依赖于下面的至关重要的事实（见图 14-19）。

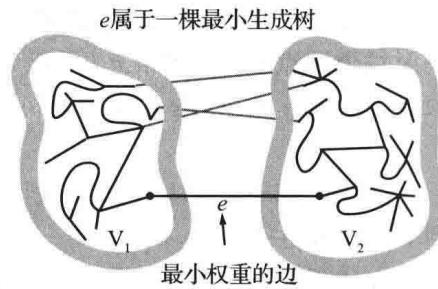


图 14-19 关于最小生成树的重要事实的说明

**命题 14-25：**定义  $G$  是一个有权重的连通的图，令  $V_1$  和  $V_2$  是两个不相交的非空集合的  $G$  的顶点的一部分。此外，令  $e$  是那些一个端点在  $V_1$ 、另一个端点在  $V_2$  的有最小权重的  $G$  的边。这就是一棵最小生成树， $e$  是它的一条边。

**证明：**令  $T$  是  $G$  的最小生成树。如果  $T$  不包含边  $e$ ，则将  $e$  添加到  $T$  必须创建一个循环。因此，循环中的一些边  $f \neq e$  有一个端点在  $V_1$ ，另一个端点在  $V_2$ 。此外，通过  $e$  的选择， $w(e) \leq w(f)$ 。如果我们从  $T \cup \{e\}$  中移除  $f$ ，便获得了一棵总权重不比以前多的生成树。因为  $T$  是最小生成树，所以新的树同样必须是最小生成树。■

事实上，如果  $G$  的权重是不同的，那么最小生成树是唯一的，我们将这个不是特别重要的事实的证明留作练习 C-14.65。另外，注意即使图  $G$  包含负权重的边或者负权重的循环，命题 14-25 都是有效的，不像我们提出的最短路径算法。

### 14.7.1 Prim-Jarník 算法

在 Prim-Jarník 算法中，我们从一些“根”顶点  $s$  开始的单个集群生成一棵最小生成树。其主要思想和 Dijkstra 算法类似。我们以一些顶点  $s$  开始，定义顶点  $C$  的初始“云”。然后，在每次迭代中，我们选择一个最小权重的边  $e = (u, v)$ ，将云  $C$  中的顶点  $u$  连接到  $C$  之外的顶点  $v$ 。之后将顶点  $v$  放到云  $C$  中，并且这个进程一直重复直到生成树形成。再一次，最小生成树的重要事实发挥作用，因为一直选择最小权重的边，一个顶点在  $C$  内，另一个在  $C$  外，所以我们可以确保一直在添加有效的边到 MST。

为了高效地实现这个方法，我们可以从 Dijkstra 算法中得到另一个线索。我们为云  $C$  之外的每个顶点  $v$  维持标签  $D[v]$ ，因此将  $v$  加入云  $C$ ， $D[v]$  存储了被观察到的最小边的权重。(在 Dijkstra 算法中，这个标签测量了从开始顶点  $s$  到  $v$  的全部路径长度，包括边  $(u, v)$ 。) 这些标签用作优先级队列中的键，用于决定哪个顶点在下一行中加入云。我们在代码段 14-15 中给出了伪代码。

#### 代码段 14-15 MST 问题的 PrimJarník 算法

**Algorithm** PrimJarník( $G$ ):

**Input:** An undirected, weighted, connected graph  $G$  with  $n$  vertices and  $m$  edges

**Output:** A minimum spanning tree  $T$  for  $G$

Pick any vertex  $s$  of  $G$

$D[s] = 0$

**for** each vertex  $v \neq s$  **do**

$D[v] = \infty$

Initialize  $T = \emptyset$ .

Initialize a priority queue  $Q$  with an entry  $(D[v], (v, \text{None}))$  for each vertex  $v$ , where  $D[v]$  is the key in the priority queue, and  $(v, \text{None})$  is the associated value.

**while**  $Q$  is not empty **do**

$(u, e) = \text{value returned by } Q.\text{remove\_min}()$

    Connect vertex  $u$  to  $T$  using edge  $e$ .

**for** each edge  $e' = (u, v)$  such that  $v$  is in  $Q$  **do**

        {check if edge  $(u, v)$  better connects  $v$  to  $T$ }

**if**  $w(u, v) < D[v]$  **then**

$D[v] = w(u, v)$

            Change the key of vertex  $v$  in  $Q$  to  $D[v]$ .

            Change the value of vertex  $v$  in  $Q$  to  $(v, e')$ .

**return** the tree  $T$

#### PrimJarník 算法的分析

PrimJarník 算法实现中的问题和 Dijkstra 算法类似，它们均依赖于一个适应性强的优先队

列  $Q$  (见 9.5.1 节)。我们最初将  $n$  插入  $Q$  中, 后来执行  $n$  的取出操作, 并且可能更新全部  $m$  的优先权作为算法的一部分。这些步骤是全部运行时间中主要的花费。有一个基于堆的优先队列, 每个操作运行时间为  $O(\log n)$ , 算法全部运行时间是  $O((n + m) \log n)$ , 对于一个连通图来说是  $O(m \log n)$ 。或者, 我们可以通过使用未排序的列表作为优先队列来达到  $O(n^2)$  的运行时间。

### PrimJarník 算法的图解

我们在图 14-20 和图 14-21 中对 PrimJarník 算法进行了图解说明。

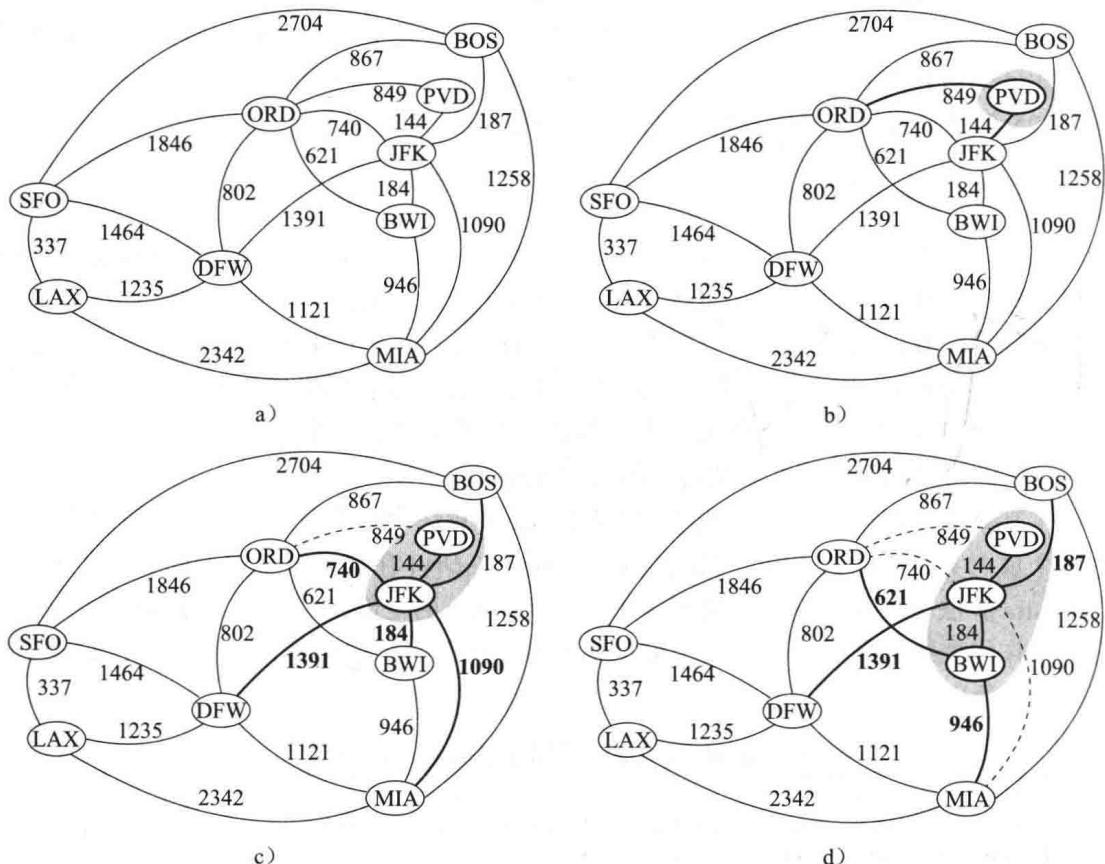


图 14-20 PrimJarník MST 算法的图解说明, 以顶点 PVD 开始 (下接图 14-21)

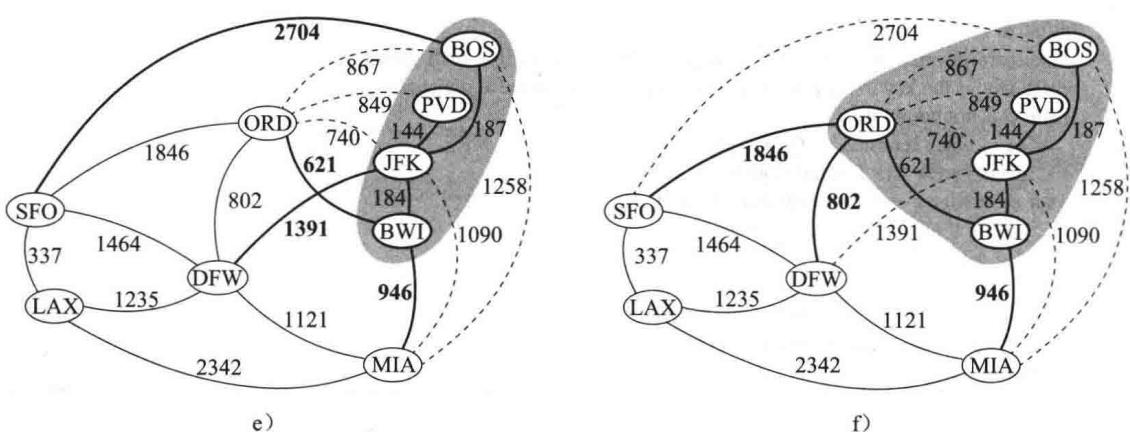


图 14-21 PrimJarník MST 算法的图解说明 (上接图 14-20)

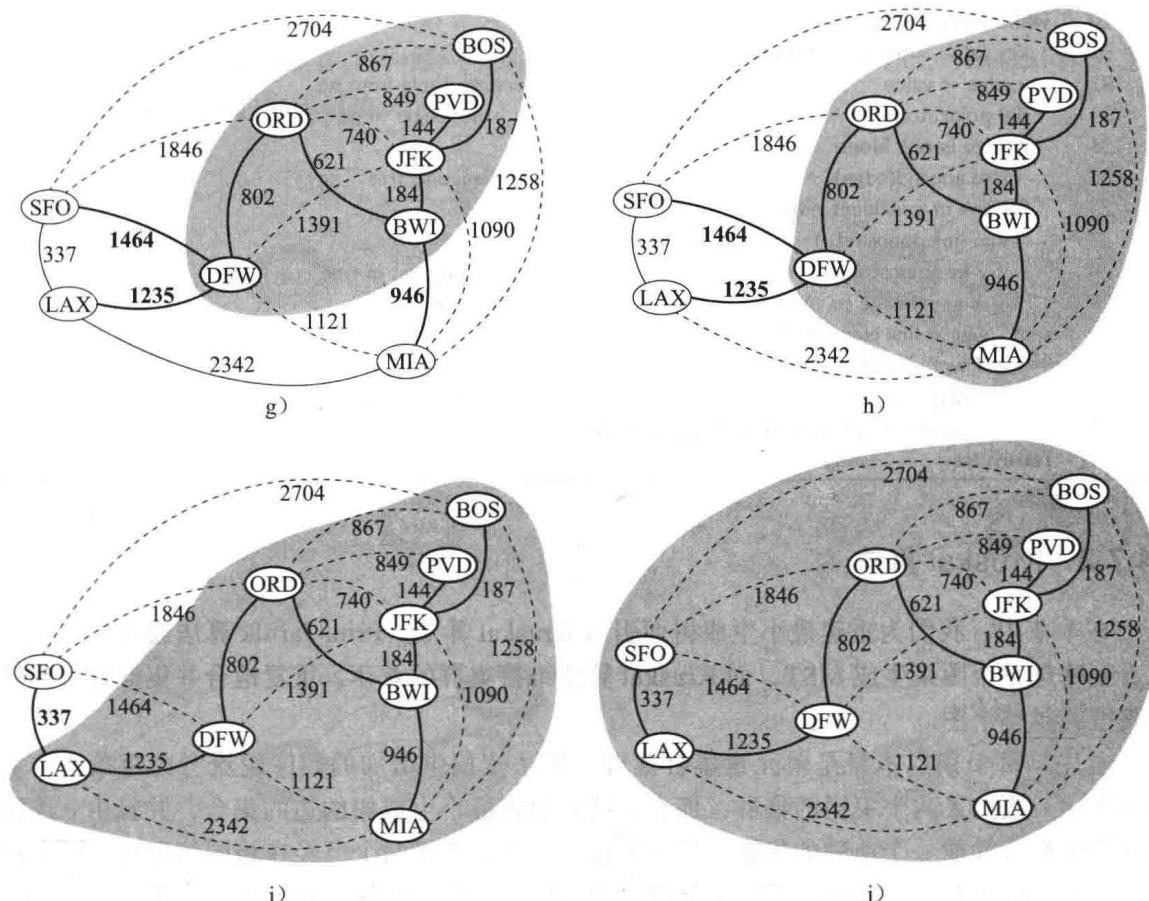


图 14-21 (续)

### Python 实现

代码段 14-16 展示了 PrimJarník 算法的 Python 实现。MST 被作为一个边的无序列表返回。

#### 代码段 14-16 最小生成树问题的 PrimJarník 算法的 Python 实现

```

1 def MST_PrimJarnik(g):
2     """Compute a minimum spanning tree of weighted graph g.
3
4     Return a list of edges that comprise the MST (in arbitrary order).
5     """
6     d = { }                      # d[v] is bound on distance to tree
7     tree = [ ]                    # list of edges in spanning tree
8     pq = AdaptableHeapPriorityQueue() # d[v] maps to value (v, e=(u,v))
9     pqlocator = { }               # map from vertex to its pq locator
10
11    # for each vertex v of the graph, add an entry to the priority queue, with
12    # the source having distance 0 and all others having infinite distance
13    for v in g.vertices():
14        if len(d) == 0:            # this is the first node
15            d[v] = 0              # make it the root
16        else:
17            d[v] = float('inf')   # positive infinity
18        pqlocator[v] = pq.add(d[v], (v, None))
19

```

```

20 while not pq.is_empty():
21     key,value = pq.remove_min()
22     u,value = value
23     del pqlocator[u]
24     if edge is not None:
25         tree.append(edge)                                # add edge to tree
26     for link in g.incident_edges(u):
27         v = link.opposite(u)
28         if v in pqlocator:                            # thus v not yet in tree
29             # see if edge (u,v) better connects v to the growing tree
30             wgt = link.element()
31             if wgt < d[v]:                            # better edge to v?
32                 d[v] = wgt
33                 pq.update(pqlocator[v], d[v], (v, link)) # update the pq entry
34

```

### 14.7.2 Kruskal 算法

在本节中，我们为重建最小生成树而引入 Kruskal 算法。Prim-Jarník 算法通过生成单个树直到跨越整个图来生成 MST，而 Kruskal 算法维持集群的森林，重复地合并集群对直到单个集群跨越整个图。

首先，每个顶点本身是单元素集合集群。算法按权重增加的顺序轮流考虑每条边。如果一条边  $e$  连接了两个不同的集群，那么  $e$  被添加到最小生成树的边的集合，并且由  $e$  连接的两个集群合并成一个单独的集群。另一方面，如果  $e$  连接两个已经在相同的集群的两个顶点，那么  $e$  被丢弃。一旦算法添加了足够的边去形成一棵生成树，算法就结束了，并且这棵树作为最小生成树输出。

我们在代码段 14-17 中给出了 Kruskal 的 MST 算法的伪代码，并且在图 14-22 ~ 图 14-24 中展示了这个算法的例子。

代码段 14-17 MST 问题的 Kruskal 算法

**Algorithm Kruskal( $G$ ):**

**Input:** A simple connected weighted graph  $G$  with  $n$  vertices and  $m$  edges

**Output:** A minimum spanning tree  $T$  for  $G$

**for each vertex  $v$  in  $G$  do**

Define an elementary cluster  $C(v) = \{v\}$ .

Initialize a priority queue  $Q$  to contain all edges in  $G$ , using the weights as keys.

$T = \emptyset$  { $T$  will ultimately contain the edges of the MST}

**while**  $T$  has fewer than  $n - 1$  edges **do**

$(u, v)$  = value returned by  $Q.\text{remove\_min}()$

Let  $C(u)$  be the cluster containing  $u$ , and let  $C(v)$  be the cluster containing  $v$ .

**if**  $C(u) \neq C(v)$  **then**

Add edge  $(u, v)$  to  $T$ .

Merge  $C(u)$  and  $C(v)$  into one cluster.

**return** tree  $T$

就像 Prim-Jarník 算法的情况，Kruskal 算法的正确性基于命题 14-25 中最小生成树的重要事实。每次 Kruskal 算法添加一条边  $(u, v)$  到最小生成树  $T$  中，我们可以通过让  $V_1$  成为包含  $v$  的集群，并让  $V_2$  包含  $V$  中的剩余顶点来定义顶点  $V$  的集合的一个分区。这可以明确

地定义一个  $V$  的顶点不相交的分区，并且更重要的是，因为我们按权重的顺序来从  $Q$  中提取边， $e$  必须是一个顶点在  $V_1$ 、另一个顶点在  $V_2$  的最小权重的边。因此，Kruskal 算法总是能添加有效的最小的生成树边。

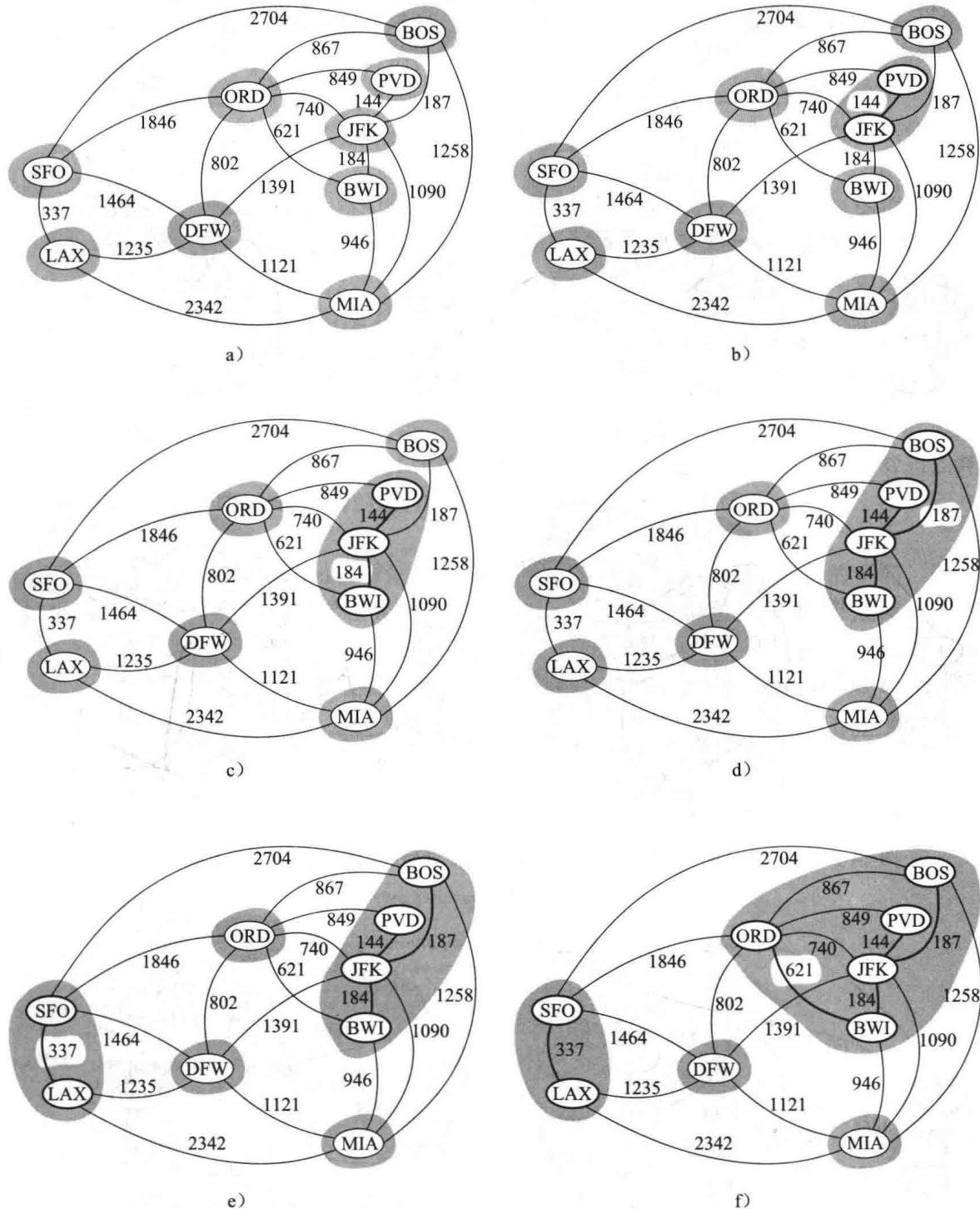
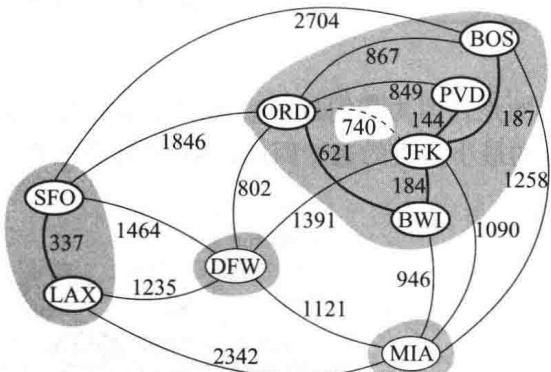
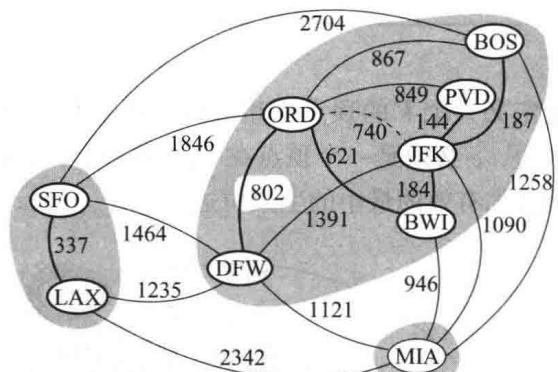


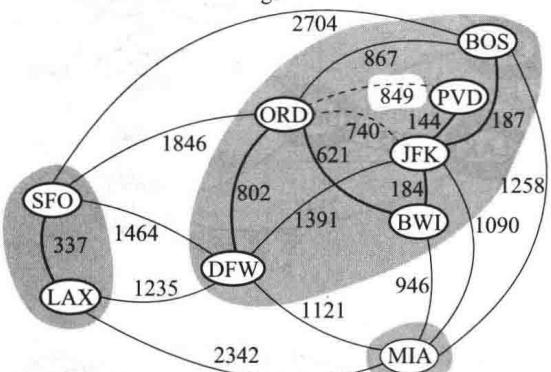
图 14-22 有数字权重的图的 Kruskal 算法执行的例子。我们将集群作为阴影区域展示，并且突出显示在每个迭代中考虑的边（下接图 14-23）



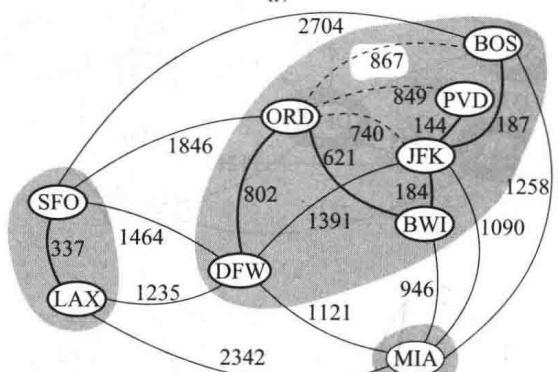
g)



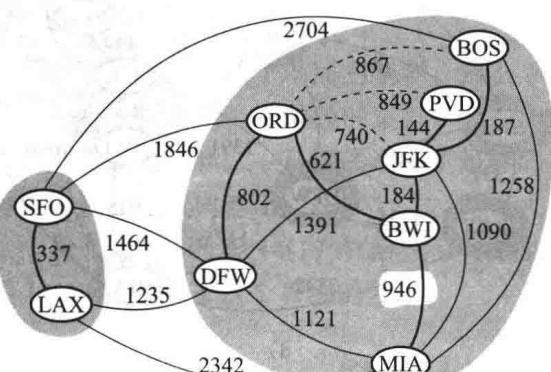
h)



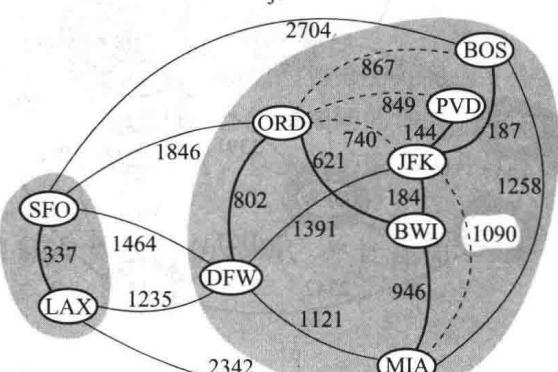
i)



j)

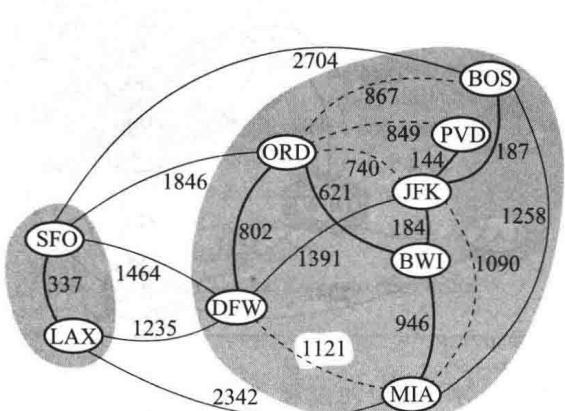


k)

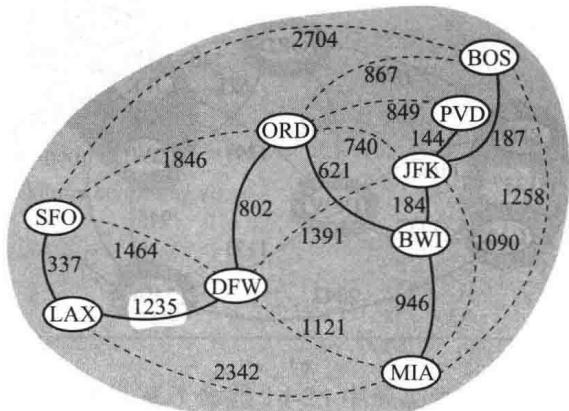


l)

图 14-23 Kruskal 的 MST 算法执行的例子。不合格的边用虚线显示 (上接图 14-24)



m)



n)

图 14-24 Kruskal 的 MST 算法执行的例子。我们所考虑的边合并了最后两个集群，总结了 Kruskal 算法的执行 (上接图 14-23)

### Kruskal 算法的运行时间

Kruskal 算法的运行时间主要花费在两个方面。第一个是需要考虑权重的非递减顺序的边，第二个是集群分区的管理。分析运行时间时，我们需要在实现中给出更多的细节。

按权重的边的顺序可以在  $O(m \log m)$  的时间内实现，或者通过排序算法，或者通过使用优先队列  $Q$ 。如果那个队列是用堆实现的，我们可以通过进行重复的插入操作在  $O(m \log m)$  的时间内初始化  $Q$ ，或者在  $O(m)$  时间内使用自下而上的堆来建造（见 9.3.6 节），后来每次 `remove_min` 调用的运行时间为  $O(\log m)$ ，因为队列的大小是  $O(m)$ 。我们注意到因为对一个简单图来说  $m$  是  $O(n^2)$ ，所以  $O(\log m)$  和  $O(\log n)$  是一样的。因此，由于边的顺序导致运行时间为  $O(m \log n)$ 。

剩下的任务是集群的管理，为了实现 Kruskal 算法，我们必须能够找到边  $e$  的顶点  $u$  和  $v$  的集群，并测试这些集群是否是不同的，如果不同，就将这两个集群合并成一个。我们迄今为止学习的数据结构没有能很好地适合这个任务的。然而，我们通过形式化管理不相交分区的问题来总结本章，并且引入了高效的联合查找数据结构。在 Kruskal 算法中，我们执行了至多  $2m$  个查找操作和  $n - 1$  个并集操作。可以看到，一个简单的联合查找结构可以在  $O(m + n \log n)$  的时间内执行组合操作（见命题 14-26），而且更先进的结构可以支持更快的时间。

对于一个连通图， $m \geq n - 1$ ，并且对边排序  $O(m \log n)$  的时间的界限限制了管理集群的时间。综上所述，Kruskal 算法的运行时间为  $O(m \log n)$ 。

### Python 实现

代码段 14-18 展示了 Kruskal 算法的 Python 实现。随着 Prim-Jarník 算法的实现，最小生成树以边的列表的形式返回。在 Kruskal 算法中，这些边将会以它们的权重非递减的顺序被报告。

我们的实现过程假设为了管理集群分区而使用 Partition 类。Partition 类的实现见 14.7.3 节。

#### 代码段 14-18 最小生成树问题的 Kruskal 算法的 Python 实现

```

1 def MST_Kruskal(g):
2     """Compute a minimum spanning tree of a graph using Kruskal's algorithm.
3
4     Return a list of edges that comprise the MST.
5
6     The elements of the graph's edges are assumed to be weights.
7     """
8     tree = []                      # list of edges in spanning tree
9     pq = HeapPriorityQueue()        # entries are edges in G, with weights as key
10    forest = Partition()          # keeps track of forest clusters
11    position = {}                 # map each node to its Partition entry
12
13    for v in g.vertices():
14        position[v] = forest.make_group(v)
15
16    for e in g.edges():
17        pq.add(e.element(), e)      # edge's element is assumed to be its weight
18
19    size = g.vertex_count()
20    while len(tree) != size - 1 and not pq.is_empty():
21        # tree not spanning and unprocessed edges remain
22        weight,edge = pq.remove_min()
23        u,v = edge.endpoints()

```

```

24     a = forest.find(position[u])
25     b = forest.find(position[v])
26     if a != b:
27         tree.append(edge)
28         forest.union(a,b)
29
30     return tree

```

### 14.7.3 不相交分区和联合查找结构

在本节中，我们考虑用于管理的分区为不相交集合的元素集合的数据结构。我们的原始动机是以 Kruskal 的最小生成树算法为支持，保持了不相交的树的森林，偶尔有邻近树的合并。更一般地说，不相交分区问题可以被应用于各种模型的离散增长。

我们用下面的模型来形式化问题。分区数据结构管理了被组织在不相交集合中元素的全集（即元素属于这些集合中的一个且仅一个集合）。和 Set ADT 或者 Python 的 set 集合不同，我们不期望能够遍历集合的内容，也不能有效地测试给定集合是否包括给定的元素。为了避免这样的观念混淆，我们称分区的集群为组。然而，对每一组将不需要一个明确的结构，取而代之的是允许组的组织变得含蓄。为了区别一个组和另一个组，我们假设在任何时候，每个组都有指定的条目，我们称之为组的领导。

我们使用位置目标来定义分区 ADT 的方法，每个位置目标存储了一个元素  $x$ 。分区 ADT 支持以下方法。

- `make_group(x)`: 创建一个包含新元素  $x$  的不相交的组并且返回存储  $x$  的位置。
- `union(p, q)`: 合并包含位置  $p$  和  $q$  的组。
- `find(p)`: 返回包含位置  $p$  的组的领导的位置。

#### 序列的实现

总共有  $n$  个元素的分区的简单实现使用了序列的集合，对每个组都有一个序列，其中组  $A$  的序列存储了元素位置。每个位置对象存储了一个变量 `element`，它引用其相关联的元素  $x$  并且允许 `element()` 方法在  $O(1)$  的时间内执行。此外，每个位置存储了一个变量 `group`，引用存储  $p$  的序列，因为这个序列代表包含  $p$  元素的组（见图 14-25）。

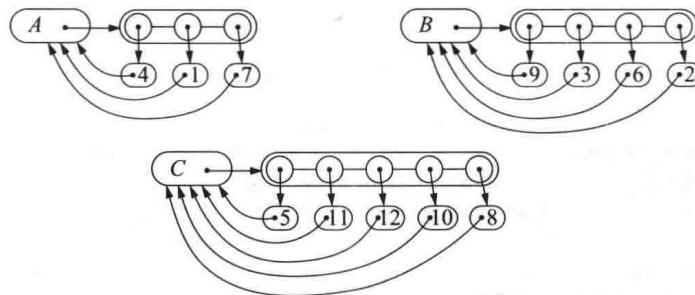


图 14-25 由三个组组成的分区基于序列的实现，三个组是： $A = \{1, 4, 7\}$ ， $B = \{2, 3, 6, 9\}$ ， $C = \{5, 8, 10, 11, 12\}$

使用此方法，我们可以很容易在  $O(1)$  时间内执行 `make_group(x)` 和 `find(p)` 操作，允许序列的第一个位置作为“领导”。`union(p, q)` 操作需要将两个序列联合成一个并且更新其中一个的位置的组引用。我们通过移除有更小尺寸的序列的所有位置来选择实现这种操作，并且在有更大尺寸的序列中插入它们。

每次我们从较小的组  $a$  得到位置并且将它插入更大的组  $b$  时，都要更新组的引用，因为位置现在指向  $b$ 。因此，操作  $\text{union}(p, q)$  花费了  $O(\min(n_p, n_q))$ ，其中  $n_p$ (resp. $n_q$ ) 是包含位置  $p$ (resp. $q$ ) 的组的基数。显然，如果在分区全集中有  $n$  个元素，则时间是  $O(n)$ 。然而，我们接下来进行摊销分析，它展示了这个实现比最坏情况下的分析好很多。

**命题 14-26：** 使用上述基于序列的分区实现时，对涉及最多  $n$  个元素的最初空分区执行一系列关于  $k$  的  $\text{make\_group}$ 、 $\text{union}$  和  $\text{find}$  操作需要  $O(k + n \log n)$  的时间。

**证明：** 我们使用统计的方法并且假设一美元可以支付执行一个  $\text{find}$  操作、一个  $\text{make\_group}$  操作或者在一个  $\text{union}$  操作中从一个序列到另一个序列的位置目标的移动时间。在  $\text{find}$  操作或者  $\text{make\_group}$  操作的情况下，我们为操作本身花费 1 美元。在  $\text{union}$  操作的情况下，我们假设 1 美元可以为比较两个序列大小的固定时间的工作支付，并且我们为从较小的组移动到较大的组的每一个位置花费 1 美元。显然，为每一个  $\text{find}$  和  $\text{make\_group}$  操作支付的 1 美元，和为每一个  $\text{union}$  操作收集的第一个美元，合计是全部的  $k$  美元。

那么，为位置而支出的花费是为了  $\text{union}$  操作。重要的是，每次我们从一个组到另一个组移动一个位置，位置组的大小至少是两倍。因此，每个位置至多在  $\log n$  的时间内从一个组移动到另一个组；因此，每个位置至多被支付  $O(\log n)$  次。因为我们假设原始分区是空的，在给定的操作序列中有  $O(n)$  个不同的被引用的元素，这暗示着在  $\text{union}$  操作期间，移动元素的总时间是  $O(n \log n)$ 。  
■

#### 基于树的分区实现 \*

表示分区的其他数据结构使用了树的集合去存储  $n$  个元素，其中每棵树和不同的组相关联（见图 14-26）。特别地，我们用链接的数据结构实现每棵树，其本身也是组位置对象。我们视每个位置  $p$  是一个有实例变量的节点  $\text{element}$ ，指向它的元素  $x$ ，以及一个实例变量  $\text{parent}$ ，指向它的父节点。按照惯例，如果  $p$  是树的根，我们设置  $p$  的父节点指向它自己。

使用这种分区数据结构，操作  $\text{find}(p)$  通过从位置  $p$  向上走到树的根来执行，在最坏的情况下它花费了  $O(n)$  的时间。操作  $\text{union}(p, q)$  可以通过使一棵树变成另一棵树的子树来实现。这个可以通过定位两个树根，然后在  $O(1)$  的附加时间内通过设置一个树根的父节点的引用指向另一个树根来实现。这两种操作的例子在图 14-27 中给出。

首先，这个实现可能不比基于序列的数据结构好，但是我们可以添加以下两个简单的启发式方法让它运行得更快。

- 基于大小的  $\text{union}$  操作：对每一个位置  $p$ ，在  $p$  位置的子树根存储元素的

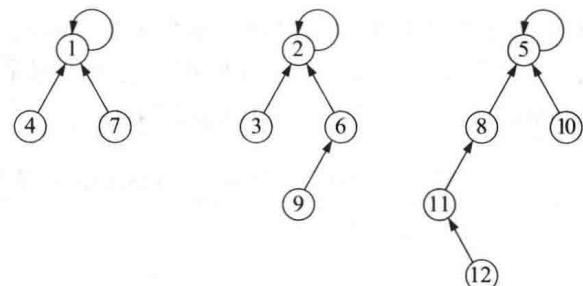


图 14-26 包含三组的分区基于树的实现，三组为： $A=\{1, 4, 7\}$ ， $B=\{2, 3, 6, 9\}$ ， $C=\{5, 8, 10, 11, 12\}$

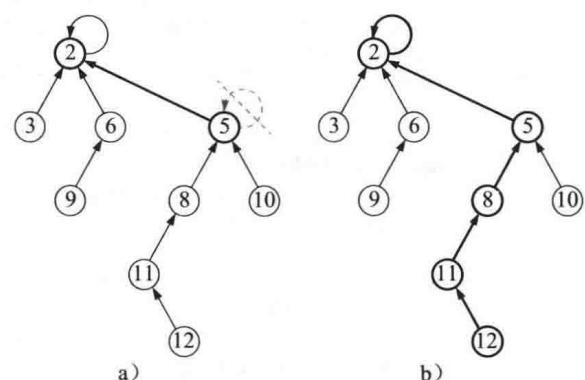


图 14-27 一个分区基于树的实现：a) 操作  $\text{union}(p, q)$ ；b) 操作  $\text{find}(p)$ ，其中  $p$  指示了元素 12 的位置对象

数目。在 union 操作中，让较小的组的树根作为一个孩子或者另一个树根，然后更新较大树根的大小区域。

- 路径压缩：在 find 操作中，对于每个 find 函数访问过的位置  $q$ ，对根重置  $q$  的父节点（见图 14-28）。

这个数据结构令人惊奇的特性是，当使用上述两种启发式方法时，执行了一系列包含花费  $O(k \log^* n)$  时间的  $n$  个元素的  $k$  次操作，其中  $\log^* n$  是 log-star 操作，即 tower-of-twos 函数的倒数。直觉上， $\log^* n$  是在获得比 2 更小的数字之前可以迭代地得到一个数字的对数的次数。表 14-4 展示了一些样本值。表 14-4 展示了一些样本值。

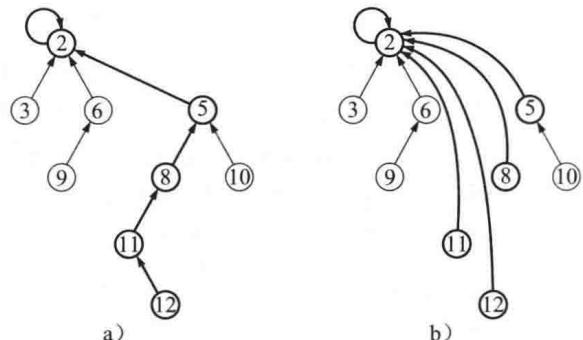


图 14-28 启发式的路径压缩：a) 对元素 12 通过 find 操作的路径遍历；b) 重建树

表 14-4 一些  $\log^* n$  的值和它的倒数的临界值

最小值 $n$	$2$	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65\,536$	$2^{2^{2^{2^2}}} = 2^{65,536}$
$\log^* n$	1	2	3	4	5

**命题 14-27：**在使用基于树的分区表示的同时按大小和路径压缩的情况下，对最多包含  $n$  个元素的初始空分区执行一系列  $k$  个 make、union 和 find 操作需要  $O(k \log^* n)$  时间。

尽管这个数据结构的分析相当复杂，但是它的实现却非常直白。我们用这个结构的完整 Python 代码来作总结，见代码段 14-19。

代码段 14-19 使用基于大小的 union 操作和路径压缩的 Partition 类的 Python 实现

```

1 class Partition:
2     """Union-find structure for maintaining disjoint sets."""
3
4     #----- nested Position class -----
5     class Position:
6         __slots__ = '_container', '_element', '_size', '_parent'
7
8         def __init__(self, container, e):
9             """Create a new position that is the leader of its own group."""
10            self._container = container      # reference to Partition instance
11            self._element = e
12            self._size = 1
13            self._parent = self           # convention for a group leader
14
15        def element(self):
16            """Return element stored at this position."""
17            return self._element
18
19        #----- public Partition methods -----
20        def make_group(self, e):
21            """Makes a new group containing element e, and returns its Position."""
22            return self.Position(self, e)
23
24        def find(self, p):
25            """Finds the group containing p and return the position of its leader."""
26            if p._parent != p:
27                p._parent = self.find(p._parent) # overwrite p._parent after recursion

```

```

28     return p._parent
29
30 def union(self, p, q):
31     """Merges the groups containing elements p and q (if distinct)."""
32     a = self.find(p)
33     b = self.find(q)
34     if a is not b:           # only merge if different groups
35         if a._size > b._size:
36             b._parent = a
37             a._size += b._size
38     else:
39         a._parent = b
40         b._size += a._size

```

## 14.8 练习

请访问 [www.wiley.com/college/goodrich](http://www.wiley.com/college/goodrich) 以获得练习帮助。

### 巩固

- R-14.1 画出一个有 12 个顶点、18 条边和 3 条连通分支的简单无向图。
- R-14.2 如果  $G$  是有 12 个顶点和 3 条连通分支的简单无向图，它的边可能的最大数目是多少？
- R-14.3 画出在图 14-1 中的无向图的邻接矩阵表示。
- R-14.4 画出在图 14-1 中的无向图的邻接列表表示。
- R-14.5 画出一个有 8 个顶点和 16 条边的简单连通的有向图，并且每个顶点的入度和出度为 2。说明有一个单独（不简单）的循环包含图的所有边，即你可以在不拿开铅笔的情况下在边的各自方向上追踪所有的边。（这样的循环被称为欧拉路径。）
- R-14.6 假设我们表示了一个有  $n$  个顶点和  $m$  条边的用边列表结构表示的图  $G$ 。为什么在这种情况下 `insert_vertex` 方法运行时间为  $O(1)$ ，而 `remove_vertex` 方法运行时间为  $O(n)$ ？
- R-14.7 给出使用邻接矩阵表示的在  $O(1)$  时间内执行 `insert_edge(u, v, x)` 操作的伪代码。
- R-14.8 就像在本章中描述的一样，用邻接列表的表示方式重做练习 R-14.7。
- R-14.9 边的列表  $E$  从邻接矩阵的表示中省略后还能达到在表 14-1 中给出的时间界限吗？为什么或者为什么不能？
- R-14.10 边的列表  $E$  从邻接列表的表示中省略后还能达到在表 14-3 中给出的时间界限吗？为什么或者为什么不能？
- R-14.11 在下列每个情况中你会使用邻接矩阵结构还是邻接列表结构？证明你的选择是合理的。
  - a) 有 10 000 个顶点和 20 000 条边的图，并且尽可能少使用空间很重要。
  - b) 有 10 000 个顶点和 20 000 000 条边的图，并且尽可能少使用空间很重要。
  - c) 你需要尽可能快地回答 `get_edge(u, v)` 查询，无论你使用了多少空间。
- R-14.12 解释为什么在用邻接矩阵结构表示的有  $n$  个顶点的简单图上进行 DFS 遍历的运行时间为  $O(n^2)$ 。
- R-14.13 为了证实非树的边都是 back 边，重新画图 14-8b，DFS 树的边用实线并且面向下，就像树的标准描述一样，并且所有的非树的边用虚线画。
- R-14.14 如果一个简单无向图包含每一对不同顶点之间的边，那么这个简单无向图是完全的。一个完全图的深度优先搜索树形状如何？
- R-14.15 从练习 R-14.14 中回想一个完全图的定义，一个完全图的广度优先搜索树形状如何？
- R-14.16 定义  $G$  是顶点从数字 1 到 8 之间的无向图，并且每个顶点的邻接顶点由下表给出：

顶点	邻接顶点
1	( 2, 3, 4 )
2	( 1, 3, 4 )
3	( 1, 2, 4 )
4	( 1, 2, 3, 6 )
5	( 6, 7, 8 )
6	( 4, 5, 7 )
7	( 5, 6, 8 )
8	( 5, 7 )

假设在  $G$  的遍历中，一个已知顶点的邻接顶点以和它们在上表列出的一样的顺序被返回。

- a) 画出  $G$ 。
- b) 给出使用以顶点 1 开始的 DFS 遍历被访问的  $G$  的顶点的序列。
- c) 给出使用以顶点 1 开始的 BFS 遍历被访问的顶点的序列。

R-14.17 画出图 14-2 中的有向图的传递闭包。

R-14.18 如果图 14-11 中图的顶点被编号为 ( $v_1 = \text{JFK}$ ,  $v_2 = \text{LAX}$ ,  $v_3 = \text{MIA}$ ,  $v_4 = \text{BOS}$ ,  $v_5 = \text{ORD}$ ,  $v_6 = \text{SFO}$ ,  $v_7 = \text{DFW}$ )，在 Floyd-Warshall 算法中边会以什么顺序加入传递闭包？

R-14.19 包含  $n$  个顶点的简单有向路径组成的图的传递闭包中有多少边？

R-14.20 已知一个有  $n$  个顶点的完全二叉树  $T$ ，以给定的位置为根，考虑一个以  $T$  中的节点作为它的顶点的有向图  $\vec{G}$ 。对  $T$  中的每一对父子节点，创建一条在  $\vec{G}$  中的从父亲到孩子的有向路径。展示有  $O(n \log n)$  条边的  $\vec{G}$  的传递闭包。

R-14.21 为在图 14-3d 中用实边画的有向图计算一个拓扑排序。

R-14.22 Bob 喜欢外语并且想在接下来的几年计划他的课程安排。他对下面九种语言课程感兴趣：LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169。课程的先决条件是：

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

在尊重先决条件的情况下，按照什么顺序 Bob 可以修习这些课程？

R-14.23 画一个有 8 个顶点和 16 条边的简单的连通的加权图，每条边的权重都唯一。确定一个顶点作为“开始”顶点并且说明 Dijkstra 算法在这个图上的运行时间。

R-14.24 若图是有向的，并且我们想计算从一个源顶点到所有其他顶点的最短有向路径，展示如何为 Dijkstra 算法修改伪代码。

R-14.25 画出一个有 8 个顶点和 16 条边的简单的连通的无向加权图，且每条边的权重都唯一。说明 Prim-Jarník 算法为这个图计算最小生成树的执行过程。

R-14.26 问题同上，重复表述 Kruskal 算法。

R-14.27 在一个湖中有 8 个小岛，有一个国家想建造 7 座桥去连通它们，使得每个岛通过一座桥或者

更多的桥能够到达任何其他的岛。建造一座桥的花费和它的长度成正比。各个岛之间的距离在下面的表中给出。

1	2	3	4	5	6	7	8
1	- 240	210	340	280	200	345	120
2	-	265	175	215	180	185	155
3	-	-	260	115	350	435	195
4	-	-	-	160	330	295	230
5	-	-	-	-	360	400	170
6	-	-	-	-	-	175	205
7	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-

如何建桥可以得到最小的总建造花费？

- R-14.28 描述在说明了 DFS 遍历的图 14-9 中的图形化约定的含义。线粗细的含义是什么？箭头的含义是什么？虚线的含义是什么？
- R-14.29 对说明有向 DFS 遍历的图 14-8 重复练习 R-14.28。
- R-14.30 对说明 BFS 遍历的图 14-10 重复练习 R-14.28.
- R-14.31 对说明 Floyd-Warshall 算法的图 14-11 重复练习 R-14.28。
- R-14.32 对说明拓扑排序算法的图 14-13 重复练习 R-14.28。
- R-14.33 对说明 Dijkstra 算法的图 14-15 和图 14-16 重复练习 R-14.28。
- R-14.34 对说明 Prim-Jarník 算法的图 14-20 和图 14-21 重复练习 R-14.28。
- R-14.35 对说明 Kruskal 算法的图 14-22 ~ 图 14-2 重复练习 R-14.28。
- R-14.36 Goerge 声明他在从位置  $p$  开始的分区结构中有一个路径压缩的最快方式。就把  $p$  放进列表  $L$  中，然后开始接下来的父类指针。每次他遇见一个新的位置  $q$ ，就把  $q$  加到  $L$  中并且更新每个  $L$  中的节点的父类指针以指向  $q$  的父类。展示 Goerge 在长度为  $n$  的路径上运行时间为  $\Omega(n^2)$  的算法。

## 创新

- C-14.37 给出 14.2.5 节的邻接图实现的 `remove_vertex(v)` 方法的 Python 实现，确保你的实现工作对有向图和无向图都能进行。你的方法的运行时间应该为  $O(\deg(v))$ 。
- C-14.38 给出 14.2.5 节的邻接图实现的 `remove_edge(e)` 方法的 Python 实现，确保你的实现工作对有向图和无向图都能进行。你的方法的运行时间应该为  $O(1)$ 。
- C-14.39 假设我们希望用边列表结构表示一个有  $n$  个顶点的图  $G$ ，假设我们用集合  $\{0, 1, \dots, n - 1\}$  中的数字标识顶点。描述如何实现集合  $E$ ，使得 `get_edge(u, v)` 方法支持  $O(\log n)$  的性能。在这种情况下你怎么实现这个方法？
- C-14.40 令  $T$  是以由一个连通的无向图  $G$  的深度优先搜索产生的开始顶点为根的生成树。讨论为什么  $G$  中不在  $T$  中的每一条边可从  $T$  中的一个顶点走向它的一个祖先，即它是一条 back 边。
- C-14.41 假设一旦  $v$  被发现时 DFS 进程就结束了，那么在代码段 14-6 中报告从  $u$  到  $v$  的路径的解决方案应该在实际中变得更高效。修改代码以实现这个优化。
- C-14.42 定义  $G$  是一个有  $n$  个顶点和  $m$  条边的无向图  $G$ 。为每次迭代正确地遍历每个  $G$  的边描述一个时间为  $O(n + m)$  的算法。
- C-14.43 实现一个返回有向图  $G$  中的一个循环的算法（如果有这样的循环存在）。
- C-14.44 为无向图  $g$  写一个函数 `components(g)`，该函数能返回一个字典，将每个顶点映射到一个整数，该整数作为其连通分支的标识符。即两个顶点应该被映射到相同的标识符，当且仅当它们

们在相同的连通分支中。

- C-14.45 如果从开始到结束有一条路径，则称迷宫是正确的，整个迷宫是从开始可达的，并且在迷宫周围的任何一部分没有循环。已知一个  $n \times n$  网格画的迷宫，如何判定它是否是正确建造的？该算法的运行时间是多少？
- C-14.46 计算机网络应该避免单个点的失败，即如果网络节点失败的话可以断开网络。我们称一个无向的连通的图  $G$  是双连通的，如果它不包含删除这个顶点会将  $G$  分成两个或者更多的连通分支的顶点。给出一个为添加至多  $n$  条边到有  $n \geq 3$  个顶点和  $m \geq n - 1$  条边的连通图  $G$  中的算法，以保证  $G$  是双连通的。你的算法的运行时间应该是  $O(n + m)$ 。
- C-14.47 解释对于一个用无向图建造的 BFS 树来说，为什么所有的非树边是 cross 边。
- C-14.48 解释对于一个有向图建造的 BFS 树来说，为什么没有非树的 forward 边。
- C-14.49 说明如果  $T$  是一个由连通图  $G$  建造的 BFS 树，那么，对于每个在  $i$  阶层的顶点  $v$ ，在  $s$  和  $v$  之间的路径有  $i$  条边，并且  $G$  的在  $s$  和  $v$  之间的其他路径至少有  $i$  条边。
- C-14.50 证明命题 14-16。
- C-14.51 提供一个使用 FIFO 队列而不是分级规划的 BFS 算法的实现，用于管理已经被发现的顶点，直到考虑它们的邻居的时候。
- C-14.52 如果一个图  $G$  的顶点可以被分为两个集合  $X$  和  $Y$ ，使得在  $G$  中的每条边在  $X$  中有一个结束顶点并且另一个在  $Y$  中，则该图是双向的。为判定无向图  $G$  是否是双向的设计和分析一个高效的算法（在提前不知道集合  $X$  和  $Y$  的情况下）。
- C-14.53 一个有  $n$  个顶点和  $m$  条边的有向图  $\vec{G}$  的欧拉路径是一个根据它的方向每次正确遍历  $\vec{G}$  的每条边的循环。如果  $\vec{G}$  是连通的并且  $\vec{G}$  中的每个顶点的入度等于出度，则这样的循环总是存在的。为找到这样一个有向图  $\vec{G}$  的欧拉路径描述一个时间为  $O(n + m)$  的算法。
- C-14.54 名为 RT&T 的公司有  $n$  个被高速通信线路连接的开关站的网络。每个顾客的手机直接连接到在其领域内的一个站。RT&T 的工程师开发了一个电视电话原型系统，该系统允许两个客户在电话通信期间看见彼此。为了具有可接受的图像质量，被用来在两个客户之间传送视频信号的链接的数量不能超过 4 个。假设 RT&T 的网络是用图来表示的。设计一个算法，在每个站，计算使用不超过 4 个链接可以实现的站的集合。
- C-14.55 长距离电话的时间延迟可以通过用呼叫者和被呼叫者之间的电话网络通信线路的数目乘以一个很小固定常量来决定。假设名为 RT&T 的公司的电话网络是一棵树。RT&T 的工程师想计算在长途电话中可能的最大时间延迟。已知一棵树  $T$ ， $T$  的直径是  $T$  的两个节点的最长路径的长度。给出一个计算  $T$  的直径的高效算法。
- C-14.56 亚塔马林多大学和许多世界各地的其他学校一样在开展多媒体工程。计算机网络的建造是用来连接那些使用通信线路的学校，这形成了一棵树。学校决定在其中一个学校安装一个文件服务器以便在所有学校之间分享数据。因为一个线路的传输时间由链路的设置和同步控制，数据传输的花费和使用链路的数目成正比。因此，需要为文件服务器选择一个“中心”位置。已知一棵树  $T$  和  $T$  的一个节点  $v$ ， $v$  的离心率是从  $v$  到  $T$  的任何其他节点的最长路径的长度。有最小离心率的  $T$  的节点被称为  $T$  的中心。
- 设计一个计算  $T$  的中心的高效算法，其中，已知一棵  $n$  个节点的树  $T$ 。
  - 这个中心是唯一的吗？如果不是，一棵树可以有多少个不同的中心？
- C-14.57 用从 0 到  $n - 1$  的数字对  $\vec{G}$  的顶点进行编号，若有一些方式使得  $\vec{G}$  包含边  $(i, j)$  当且仅当对所有在  $[0, n - 1]$  中的  $i$  有  $i < j$ ，则称一个有  $n$  个顶点的有向非循环图  $\vec{G}$  是压缩的。给出一个

探索 $\vec{G}$ 是否是压缩的时间为 $O(n^2)$ 的算法。

- C-14.58 定义 $\vec{G}$ 是一个有 $n$ 个顶点的加权有向图。为计算从每个顶点到每个其他顶点的最短路径的长度设计一个在 $O(n^3)$ 时间内的 Floyd-Warshall 算法的变形。
- C-14.59 为寻找从一个非循环加权有向图 $\vec{G}$ 的一个顶点 $s$ 到一个顶点 $t$ 的最长有向路径设计一个高效的算法。详细说明使用的表示方式和使用的辅助数据结构。同样，分析该算法的时间复杂度。
- C-14.60 无向图 $G = (V, E)$ 的独立集合是 $V$ 的子集 $I$ ，使得 $I$ 中的两个顶点都不是相邻的。即如果 $u$ 和 $v$ 在 $I$ 中，那么 $(u, v)$ 不在 $E$ 中。极大无关组 $M$ 是一个独立的集合，使得如果我们添加任何额外的顶点到 $M$ 中，那么它将不再独立。每个图都有一个极大无关组。(你知道为什么吗？这个问题不是练习的一部分，但是一个值得思考的问题。)给出一个计算图 $G$ 的极大无关组的高效算法。这个方法的运行时间是多少？
- C-14.61 给出当一个有 $n$ 个顶点的简单图 $G$ 用堆实现的时候使得 Dijkstra 算法的运行时间为 $\Omega(n^2 \log n)$ 的例子。
- C-14.62 给出这样的例子：具有负权重的加权有向图 $\vec{G}$ ，但是没有负权重循环，使得 Dijkstra 算法错误地计算从一些开始顶点 $s$ 开始的最短路径。
- C-14.63 为在已知的连通图中从 start 顶点到 goal 顶点找到一条最短路径。考虑下面的贪心策略：
- 1) 初始化到 start 的路径。
  - 2) 将 visited 集合初始化为 {start}。
  - 3) 如果 start = goal，返回路径并且退出，否则继续。
  - 4) 找到最小权重的边 (start, v)，使得 v 和 start 相邻但是 v 没有被访问。
  - 5) 添加 v 到路径中。
  - 6) 添加 v 到被访问中。
  - 7) 设置 start 和 v 相等并且进行第 3 步。
- 这个贪心策略总是能找到从 start 到 goal 的最短路径吗？或者直观地解释为什么它会工作，或者举出一个反例。
- C-14.64 在代码段 14-13 中的 shortest\_path\_lengths 的实现依赖于使用“无穷大”作为一个数值，以表示从源（未知）可达的顶点的距离界限。在没有这样的标记的情况下重新实现，此时顶点（除了源顶点）不会添加到优先队列中，直到它们显然是可到达的。
- C-14.65 说明如果在连通的加权图 $G$ 中的所有权重都是不同的，那么 $G$ 会有一棵正确的最小生成树。
- C-14.66 一种旧的 MST 方法称为 Baruvka 算法，在有 $n$ 个顶点和 $m$ 条不同权重的边的图 $G$ 上按照下面这样工作：

Let  $T$  be a subgraph of  $G$  initially containing just the vertices in  $V$ .

**while**  $T$  has fewer than  $n - 1$  edges **do**

**for** each connected component  $C_i$  of  $T$  **do**

    Find the lowest-weight edge  $(u, v)$  in  $E$  with  $u$  in  $C_i$  and  $v$  not in  $C_i$ 。

    Add  $(u, v)$  to  $T$  (unless it is already in  $T$ ).

**return**  $T$

证明这个算法是正确的并且它的运行时间为 $O(m \log n)$ 。

- C-14.67 定义 $G$ 是一个有 $n$ 个顶点和 $m$ 条边的图， $G$ 的所有边的权重是在 $[1, n]$ 范围内的数字。为寻找 $G$ 的最小生成树给出一个运行时间为 $O(m \log^* n)$ 的算法。
- C-14.68 考虑一个电话网络的图解，这个电话网络的顶点代表转换中心，并且它的边代表连接一对中心的通信线路。边用它们的带宽标出，并且一条路径的带宽和路径的边之间的最低带宽相等。给出一个算法，已知一个网络和两个转换中心 $a$ 和 $b$ ，输出 $a$ 和 $b$ 之间的路径的最大带宽。

C-14.69 NASA 想使用传播渠道链接遍布城市的  $n$  个站。每一对站有不同的可用带宽，称为 priori。NASA 想以所有的站被渠道链接并且总带宽（定义为渠道的单个带宽的总和）是最大的方式去选择  $n - 1$  个渠道。对这个问题给出一个高效的算法并且判定它在最坏情况下的时间复杂度。考虑加权图  $G = (V, E)$ ，其中  $V$  是站的集合并且  $E$  是站之间的渠道的集合。定义  $E$  的每条边  $e$  的权重  $w(e)$  为通信渠道的带宽。

C-14.70 Asymptopia 的城堡里有一个迷宫，沿着迷宫的每个走廊有一包金币。每包的金币数量都是不同的。贵族骑士 Paul 得到了一个穿越迷宫并捡拾金币包的机会。他只能从被标记为“ENTER”的门进入迷宫并且从标记为“EXIT”的门出去。然而在迷宫中他可能不会重走路径。迷宫的每个走廊有一个在墙上喷绘的箭头。在迷宫中没有办法去遍历“循环”。已知迷宫的地图，包括每条走廊金币的数量，描述一个算法去帮助 Paul 捡到最多的金币。

C-14.71 假设你已经得到一个时间表，它包括：

- $n$  个机场的集合  $A$ ，并且对  $A$  中的每个机场  $a$ ，有最小的连接时间  $c(a)$ 。
- $m$  个航班的集合  $F$ ，对每个  $F$  中的航班，有下面的说明：
  - $A$  的起始机场  $a_1(f)$
  - $A$  的目的机场  $a_2(f)$
  - 出发时间  $t_1(f)$
  - 到达时间  $t_2(f)$

为航班的行程安排问题描述一个高效的算法。在这个问题中，我们已知机场  $a$  和  $b$  以及时间  $t$ ，并且希望计算航班的序列，该航班允许在时间  $t$  或者在时间  $t$  之后离开  $a$  时，能在最早的可能时间到达  $b$ 。在中间的机场的最小连通时间必须被考虑在内。该算法作为一个参数为  $n$  或者  $m$  的函数的运行时间是多少？

C-14.72 假设我们已知一个有  $n$  个顶点的有向图  $G$ ，并且令  $M$  是与  $G$  相一致的  $n \times n$  的邻接矩阵。

a) 定义  $M$  和它本身的乘积 ( $M^2$ )，对于  $1 \leq i, j \leq n$ ，有：

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \cdots \oplus M(i, n) \odot M(n, j)$$

其中  $\oplus$  是布尔型 or 运算符， $\odot$  是布尔型 and 运算符。已知这个定义，关于顶点  $i$  和  $j$ ， $M^2(i, j) = 1$  暗示了什么？如果是  $M^2(i, j) = 0$  呢？

b) 假设  $M^4$  是  $M^2$  和它本身的乘积。 $M^4$  中的项意味着什么？ $M^5 = (M^4)M$  呢？一般来说，包含在矩阵  $M^p$  中的信息是什么？

c) 现在假设  $G$  是有权重的并且假设有下面的说明：

- 1)  $1 \leq i \leq n, M(i, i) = 0$ 。
- 2)  $1 \leq i, j \leq n, M(i, j) = \text{weight}(i, j)$ ，如果  $(i, j)$  在  $E$  中。
- 3)  $1 \leq i, j \leq n, M(i, j) = \infty$ ，如果  $(i, j)$  不在  $E$  中。

同样，定义  $M^2$ ，对于  $1 \leq i, j \leq n$ ，有：

$$M^2(i, j) = \min \{M(i, 1) + M(1, j), \dots, M(i, n) + M(n, j)\}$$

如果  $M^2(i, j) = k$ ，我们从顶点  $i$  和  $j$  的关系中能总结出什么？

C-14.73 Karen 有新的方式在从位置  $p$  开始的基于树的并集 / 查找分区数据结构上进行路径压缩。她把从  $p$  到根的路径上的所有位置放进集合  $S$ 。然后扫描  $S$ ，并且将  $S$  中的每一个位置的父指针指向它的父类的父节点（记得根指向它自己的父节点）。如果这个过程改变了任何位置父节点的值，那么就重复这个过程，并且继续重复这个过程直到扫描完  $S$  且没有改变任何位置的父节点。说明 Karen 的算法是正确的并且分析长度为  $h$  的路径的运行时间。

## 项目

- P-14.74 使用邻接矩阵去实现支持简化的不包含 update 方法的 Graph ADT。该类应该包括有两个集合的构造函数方法，两个集合分别为顶点元素的集合  $V$  和顶点元素对的集合  $E$ ，并且产生了用这两个集合代表的图  $G$ 。
- P-14.75 使用边列表结构实现在 P-14.74 中描述的简化的 Graph ADT。
- P-14.76 使用邻接列表结构实现在 P-14.74 中描述的简化的 Graph ADT。
- P-14.77 继承 P-14.74 的类以支持 Graph ADT 的更新方法。
- P-14.78 设计重复的 DFS 遍历的实验，与用于计算有向图的传递闭包的 Floyd-Warshall 算法比较。
- P-14.79 对在本章讨论的两个最小生成树算法（Kruskal 和 Prim-Jarník）执行实验性的比较。设计一个实验的大量的集合去测试这些使用随机生成的图表的算法的运行时间。
- P-14.80 建造迷宫的一种方式是以  $n \times n$  网格开始的，其中每个网格单元以四个单位长度的墙壁为界限。然后移除两个界限单位长度的墙，表示开始和结束。对每一个剩余的单位长度的墙来说，若不在边界上，我们就指定一个随机数并且创建一个名为 dual 的图  $G$ ，因此每个网格单元都是  $G$  的一个顶点并且有一条连接两个单元顶点的边当且仅当单元分享共同的墙。每条边的权重是相应墙的权重。我们通过找到  $G$  的一棵最小生成树  $T$  和移除  $T$  中所有与边对应的墙来建造这个迷宫。使用这个算法写一个程序，生成迷宫然后解决它们。最低要求是，你的程序应该画出迷宫，并且在理想的情况下，它同样应该设想解决方案。
- P-14.81 写一个程序，基于最短路径路由为计算机网络中的节点建立一个路由表，其中路径距离用跳跃总数来测量，即路径中边的数量。这个问题的输入对所有在网络中的节点来说是连接信息，就像下面的例子一样：

241.12.31.14: 241.12.31.15 241.12.31.18 241.12.31.19

这暗示三个连接到 241.12.31.14 的网络节点，即三个节点是一跳。在地址  $A$  的节点的路由表是  $(B, C)$  对的集合，这暗示着，按从  $A$  到  $B$  的路线发送信息，下一个被送到（按照从  $A$  到  $B$  的最短路径）的节点是  $C$ 。你的程序应该对网络中的每个节点输出路由表，已知节点连通性列表的输入列表，每个输入列表像上述语法一样输入，一行一个。

## 扩展阅读

深度优先搜索方法是计算机科学发展的一部分，但是 Hopcroft 和 Tarjan<sup>[52, 94]</sup> 展示了该算法对解决一些不同图的问题是多么有用。Knuth<sup>[64]</sup> 讨论了拓扑排序问题。我们描述的简单线性时间算法是为了判定一个有向图是否是强连通的，这归功于 Kosaraju。Floyd-Warshall 算法被 Floyd<sup>[38]</sup> 呈现在书中并且基于 Warshall<sup>[102]</sup> 的原理。

第一个著名的最小生成树算法归功于 Baruvka<sup>[9]</sup> 并于 1926 年发布。Prim-Jarník 算法首先在 1930 年由 Jarník<sup>[55]</sup> 在捷克发布，并且英文版在 1957 年由 Prim<sup>[85]</sup> 发布。Kruskal 在 1956<sup>[67]</sup> 年发布了他的最小生成树算法。对最小生成树问题的更多历史感兴趣的读者可以看 Graham 和 Hell 的书<sup>[47]</sup>。目前渐近的最快最小生成树算法是 Karger、Klein 和 Tarjan<sup>[57]</sup> 在预期  $O(m)$  时间内运行的随机算法。Dijkstra<sup>[35]</sup> 在 1959 年发布了他的单源最短路径算法。Prim-Jarník 算法的运行时间，还有 Dijkstra 的运行时间，事实上可以通过用更精细的数据结构“斐波那契堆”<sup>[40]</sup> 或者“松驰的堆”<sup>[37]</sup>，通过实现队列  $Q$  以改良到  $O(n \log n + m)$ 。

为了学习不同的画图算法，请看 Tamassia 和 Liotta 的书<sup>[92]</sup>，以及 Di Battista、Eades、Tamassia 和 Tollis 的书<sup>[34]</sup>。对图的算法的深层学习感兴趣的读者可以看 Ahuja、Magnanti 和 Orlin 的书<sup>[7]</sup>，Cormen、Leiserson、Rivest 和 Stein 的书<sup>[29]</sup>，Mehlhorn 和 Tarjan 的书<sup>[77][95]</sup>，还有 van Leeuwen 的书<sup>[98]</sup>。

# 内存管理和 B 树

迄今为止，我们对数据结构的研究主要关注计算效率——通过 CPU 执行基本操作的数量来衡量。实际上，计算机系统的性能也会受计算机存储系统的管理所影响。在对数据结构的分析中，我们根据数据结构所使用的内存总量给出渐近边界。在本章中，我们考虑更多的是关于计算机存储系统的使用。

首先讨论计算机程序在执行期间内存的分配和释放，以及这对程序性能的影响。其次讨论目前计算机系统中多级存储结构的复杂性。虽然我们经常将计算机的存储器抽象为自由互换位置的池，实际上，运行程序所使用的数据是在物理存储器的组合中进行存储和传输的（如 CPU 的寄存器、高速缓存、内部存储器和外部存储器）。我们考虑使用经典的管理内存的数据结构算法，以及存储器层次结构是如何影响数据结构和算法的选择的，如查找和排序等经典问题。

## 15.1 内存管理

为了在实际计算机中实现任何数据结构，我们需要使用计算机的内存。计算机存储器被组织成字序列，其中每一个序列通常包含 4、8 或 16 个字节（取决于计算机）。这些内存字编号从 0 到  $N - 1$ ，其中  $N$  是计算机可获得的内存字节的数量。与每个内存字节相关联的数字称为内存地址。因此，计算机的存储器基本上可被视为一个巨大的内存字节的矩阵。如 5.2 节的图 5-1 所示，我们所描绘的计算机的部分内存如下图所示。



为了运行程序和存储信息，必须对计算机的内存进行管理，以便确定什么样的数据被存储在哪个存储器单元。在这一节中，我们将讨论内存管理的基本知识，特别描述存储新对象时怎样分配内存，当对象不再需要时怎样将分配的内存进行释放和回收，以及 Python 解释器怎样使用内存来完成任务。

### 15.1.1 内存分配

在 Python 中，所有对象都存储在内存池中，该内存池称为内存堆或 Python 堆（不要与第 9 章中提出的“堆”数据结构相混淆）。当运行如下的命令时，

```
w = Widget()
```

假定 `Widget` 是一个类名，该类的一个新实例被创建并存储在内存堆中的某个地方。当执行 Python 程序时，Python 解释器负责协调操作系统空间的使用和管理内存堆的使用。

内存堆存储空间被分成连续的块，类似于矩阵，块的大小可以是变量或固定值。系统必须实现该功能，才可以迅速为新对象分配内存。一种常用的方法是将连续空间的可用内存连

接到链表上，称为空闲表。只要它们的内存未被使用，这些空间就会被连接到链表中。随着内存的分配和释放，空闲表中的空间集就会发生变化，那些未使用的内存空间被已用的内存块分离成不相连的空间。未使用的内存分离成单独的空间，也被称为碎片。现在的问题是找到大的连续内存块将会变得越来越难，即使使用等量的未被使用的内存（但碎片化）。因此，我们希望尽可能地使碎片最小化。

可能产生两种类型的碎片。当所分配的内存块的一部分未使用时，可能产生内部碎片，例如，程序可以请求大小 1000 的矩阵，但仅使用该矩阵的前 100 个内存单元。没有太多的运行时环境可以做到降低内部碎片。此外，当几个已分配内存的连续块之间有未使用的内存空间时，可能会产生外部碎片。由于运行时环境可以控制当请求发生时在哪里分配内存，故运行时环境应该以尽量减少外部碎片的方式来分配内存。

为了最小化外部碎片，建议用几种启发式方法来从堆中分配内存。最佳适应算法是搜索整个空闲列表以查找其大小最接近所请求内存的空间。首次适应算法是从空闲列表的首部开始搜索，直至搜索到第一个足够大的空间。循环首次适应算法与首次适应算法类似，它也是搜索空闲列表中第一个足够大的空间，但它每次搜索都从以前中断的地方开始，将空闲列表视为循环链表并开始搜索（7.2 节）。最差适应算法搜索空闲列表以找到最大的可用内存空间，如果该列表保存为一个优先级队列（第 9 章），会比搜索整个空闲列表的速度更快。在每一个算法中，从所选的内存空间减去所请求的内存量后，剩余的内存空间会返回到空闲列表中。

尽管最佳适应算法听起来可能不错，但由于所选择空间的剩余部分偏小，故最易产生外部碎片。首次适应算法快，但它往往在空闲列表前面产生很多的外部碎片，这将降低之后的搜索速率。循环首次适应算法使碎片更均匀地分布在内存堆，从而降低了搜索时间，但很难分配大的内存块。最差适应算法试图通过保留尽可能大的连续内存空间来避免这种问题。

### 15.1.2 垃圾回收

有些语言（如 C 和 C ++），明确规定对象的存储空间由程序员释放，而这是初级程序员经常忽略的任务，甚至对有经验的程序员来说也是令人沮丧的编程错误的根源。与此相反，Python 的设计者将内存管理的负担完全交给解释器。解释器负责检测“陈旧”对象的进程，释放用于这些对象的空间，并返回回收空间到空闲列表，这一过程称为垃圾回收。

要执行自动垃圾回收，首先必须有方法来检测到那些不再需要的对象。由于解释器不能有效分析任意 Python 程序的语义，它依赖于以下用于回收对象的保护规则。要访问程序中的一个对象，它必须有该对象的直接或间接引用。我们将这种对象定义为活动对象。在定义活动对象时，对象的直接引用是以标识符的形式存在于活跃的命名空间（即全局命名空间，或任何函数的本地命名空间）。例如，执行命令 `w = Widget()` 后，标识符 `w` 将作为新的 `widget` 对象的引用在当前的命名空间定义。我们只能直接引用的这种对象为根对象。对象的间接引用是发生在一些其他活动对象的状态中的引用。例如，如果前面例子中的 `Widget` 实例包含一个列表属性，该列表也是一个活动对象（因为它可以通过使用标识符 `w` 来间接达到）。这组活动对象是递归定义的，因此由 `Widget` 引用的列表中的任何对象也归为活动对象。

Python 解释器假设活动对象是正在运行的程序中使用的活跃对象，这些对象不应该被释放。其他的对象可以被垃圾回收。Python 通过以下两个策略来确定哪些是活动对象。

## 引用计数

每个 Python 对象的状态都是一个整数，称为引用计数，即计算机系统中任何地方的对象有多少次引用。每一次引用赋给这个对象时，该对象的引用计数递增，每一次的引用被重新分配给其他对象时，原对象的引用计数递减。每个对象的引用计数的维护增加了  $O(1)$  空间，并且每次引用计数的递增和递减操作都会给  $O(1)$  空间增加额外的计算时间。

Python 解释器允许运行程序来检测一个对象的引用计数。系统模块中有一个 `getrefcount` 函数，返回一个等于对象的引用计数的整数并作为一个参数传递。值得注意的是，因为该函数的形参要赋给调用方的实参，所以当报告计数时，在函数的本地命名空间中有该对象的附加引用。

引用计数的优点是，如果一个对象的计数减到零，那么该对象不可能是活动对象，因此该系统能够立即释放该对象（或将其放置在准备释放的对象的队列中）。

## 周期检测

若对象的引用计数为零，显然意味着它不可能是活动对象，但重要的是要辨别一个有非零引用计数的对象是否仍没资格作为活动对象。有可能存在一组对象，这些对象互相引用，即使这些对象到根对象都是不可达的。

例如，正在运行的 Python 程序有一个标识符 `data`，它是使用双链表实现序列的一个引用。在这种情况下，由 `data` 引用的列表是一个根对象，作为列表的属性存储的首部和尾部节点是活动对象，因为列表的所有中间节点都是间接引用，并且所有元素作为这些节点的元素引用。如果标识符 `data` 离开了该范围或将被重新分配给其他对象，对于列表实例的引用计数可能变为零，成为垃圾回收，但所有节点的引用计数仍为非零，由上面的简单规则将阻止其垃圾回收。

几乎每隔一段时间，特别是当内存堆中的可用空间正在变得越来越稀缺时，Python 解释器就会使用垃圾回收的更高级形式来收回不可达的对象，尽管它们的引用计数非零。有不同的用于实现周期检测的算法（Python 中的 GC 模块的垃圾回收机制是抽象的，并依赖于解释器的实现方式）。接下来讨论垃圾回收的经典算法：标记 – 清除算法。

在标记 – 清除垃圾回收算法中，我们设置一个“标记”位来标识每个对象是否是活动对象。当确定在某些时候需要垃圾回收，我们暂停所有其他活动，并清除当前在内存堆中分配的所有对象的标志位，然后通过跟踪活跃的命名空间来标记所有根对象为活动对象。我们必须确定所有其他活动对象——从根对象可达的对象。为了有效地做到这一点，我们就可以（见 14.3.1 节）由对象引用其他对象所定义的有向图进行深度优先搜索。在这种情况下，内存堆中的每个对象是一个有向图顶点，并且从一个对象到另一个对象的引用是一条有向边。通过从每个根对象进行深度优先搜索，我们可以正确识别并标记每个活动对象，这一过程被称为“标记”阶段。一旦这个过程完成，再通过内存堆扫描并回收未被标记的对象正在使用的任何空间。这时，还可以有选择地将内存堆中的分配空间合并成一个单独的块，从而暂时消除外部碎片。该扫描和回收过程被称为“清除”阶段。当清除完成时，恢复运行暂停的程序。因此，标记 – 清除垃圾回收算法会按照活动对象的数量和其引用的数量加上内存堆的大小的比例，及时回收未使用的空间。

## 就地执行 DFS

标记 – 清除算法能正确回收内存堆中未使用的空间，但在“标记”阶段面临一个重要问题。由于我们是在可用内存不足时回收内存空间，因此必须注意在垃圾回收期间不要使用额

外的空间。麻烦的是，14.3.1 节中是以递归形式描述 DFS 算法，可以使用的空间正比于图的顶点数。在垃圾回收的情况下，图中的顶点是在内存堆中的对象，因此可能没有这么多内存可以使用。所以，唯一的选择就是找到一种方法来就地执行 DFS 而不是递归执行，也就是说，必须用固定的额外空间来执行 DFS。

就地执行 DFS 的主要思想是，模拟递归堆栈使用图的边（在垃圾收集的情况下相当于对象引用）。从访问过的顶点  $v$  到一个新的顶点  $w$  进行遍历时，修改边  $(v, w)$  存储在  $v$  的邻接表来指向在 DFS 树中  $v$  的父母节点。返回到  $v$  时（模拟从  $w$  上的“递归”调用返回），假设有方法来确定哪些边需要改变，我们可以切换到指向修改的边  $w$ 。

### 15.1.3 Python 解释器使用的额外内存

15.1.1 节已经讨论过 Python 解释器如何在内存堆中为对象分配内存，然而并非只有在运行 Python 程序时需要使用内存。我们将在本节讨论内存的其他一些重要用途。

#### 运行时调用栈

栈在 Python 程序的运行时环境中有很多重要的应用。运行中的 Python 程序有一个私有栈，称为调用栈或 Python 解释器栈，该栈用于跟踪函数调用的当前活跃（即未结束的）的嵌套序列。堆栈的每个条目都是一个被称为活动记录或框架的结构，存储了函数调用的重要信息。

在调用栈的顶部是正在调用的活动记录，也就是当前控制执行的函数活动。栈的其余元素是挂起等待调用的活动记录，也就是函数已经调用另一个函数，目前等待另一个函数结束时返回控制给前函数。堆栈中元素的顺序对应于当前函数调用的链。当一个新函数被调用时，调用该函数的活动记录被压入栈。调用结束后，它的活动记录从栈中弹出并且 Python 解释器恢复先前暂停的调用过程。

每个活动记录包含代表函数调用的本地命名空间的字典（参考 1.10 节和 2.5 节对命名空间的进一步讨论）。命名空间将作为参数和局部变量的标识符映射到对象的值，但被引用的对象仍然驻留在内存堆。函数调用的活动记录还包括函数定义本身的引用以及一个特殊变量（称为程序计数器），包含当前正在执行的函数语句的地址。当一个函数返回控制到另一个函数时，该挂起函数存储的程序计数器使得解释器正确继续该函数的运行。

#### 递归实现

使用堆栈实现函数嵌套调用的好处是允许程序使用递归。如第 4 章中讨论的，函数可以调用其本身。本章隐式地描述了调用栈的概念和用递归跟踪地叙述活动记录的使用。有趣的是，早期的编程语言（比如 COBOL 和 Fortran）最初没有使用调用栈来实现函数调用。但由于递归的优雅和效率，几乎所有现代编程语言都使用调用栈来实现函数调用，包括经典语言的当前版本。

递归跟踪的每一层对应于在递归函数的执行过程中放置在调用堆栈上的活动记录。在任何时间点，调用栈的内容对应地从初始函数调用到当前函数的所有层。为了更好地说明调用栈如何使用递归函数，我们回顾 Python 阶乘

$$n! = n(n-1)(n-2)\cdots 1$$

函数的经典递归定义的实现，代码段 4-1 给出了原始代码，图 4-1 给出了递归跟踪。第一次调用 factorial 函数，它的活动记录包括存储参数值  $n$  的命名空间。该函数递归调用函数本身用来计算  $(n - 1)!$ ，产生了新的活动记录，有自己的命名空间和参数，然后压入调用栈。接

下来，再调用自身来计算  $(n - 2)$ ，等等。递归调用链和调用堆栈的大小长为  $n + 1$ ，最深层的嵌套调用是 `factorial(0)`，只是返回 1，不再进一步递归。运行时堆栈允许阶乘函数的几个调用同时存在。每个活动记录存储着其参数的值和最终被返回的值。当第一递归调用最终终止时将返回  $(n - 1)!$  的值，然后在 `factorial` 函数的初始调用中将返回结果乘以  $n$  从而计算出  $n!$ 。

### 操作数栈

有趣的是，Python 解释器在另一个地方也使用栈。例如，算术表达式  $((a + b)*(c + d))/e$  就是解释器通过使用操作数栈进行计算。8.5 节介绍了如何使用表达式树的后序遍历来计算算术表达式。我们是以递归方式描述该算法，然而这种递归描述可以通过非递归的过程来实现，只需包含一个操作数堆栈。一个简单的二进制操作，如  $a + b$ ，通过将  $a$  压栈、 $b$  压栈，然后调用指令从堆栈中弹出顶部的两个数，执行相应的二进制操作，并且将计算结果返回到堆栈。同样，从内存中写元素或读元素的指令涉及操作多个栈的进栈和出栈方法的使用。

## 15.2 存储器层次结构和缓存

随着社会上计算使用量的与日俱增，应用软件必须管理非常大的数据集。这样的应用包括在线金融交易、数据库的组织和维护以及客户的购买记录和偏好分析。数据的数量可以如此之大，算法和数据结构的整体性能有时更多地取决于访问数据的时间而不是处理器的速度。

### 15.2.1 存储器系统

为了容纳大数据集，计算机有不同类型的存储器层次结构，它们的大小和到 CPU 的距离有所不同。最接近 CPU 的是在 CPU 本身使用的内部寄存器。访问这些位置非常快，但这样的空间也相对较少。层次结构中的第二层是一个或多个高速缓冲存储器。这种存储空间比 CPU 的寄存器集大得多，但是访问它需要更长的时间。层次结构中的第三层是内部存储器，也称为主存储器或核心存储器。内部存储器比高速缓冲存储器大得多，但也需要更多的访问时间。层次结构中的外一层是外部存储器，它通常由磁盘、CD 驱动器、DVD 驱动器或磁带组成。这个存储器是非常大的，但也很慢。通过外部网络存储数据可以被看作该层次结构的又一级别，它有更大的存储容量，但访问速度更慢。因此，可以将计算机存储器层次结构看作包含五层或更多的层，其中每一层比前一层的存储容量更大，但访问速度更慢（见图 15-1）。在程序的执行过程中，数据定期从一层复制给相邻层，这些传输也成为计算的瓶颈。

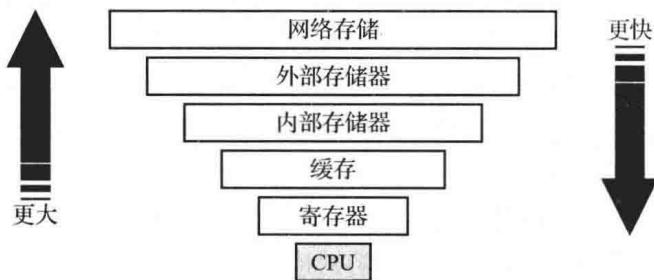


图 15-1 内存分层

### 15.2.2 高速缓存策略

存储器层次结构对程序性能的影响很大程度上取决于所要解决的问题的大小和计算机系统的物理特性。通常情况下，瓶颈发生在两个层次的存储器层次结构中，可以容纳所有的数

据项层次和一个低于该层的一层。对于在主存储器中完全匹配的问题，最重要的两个层次是高速缓冲存储器和内部存储器。访问内部内存的时间可能是高速缓冲存储器的 10 到 100 倍。因此，能够在高速缓冲存储器中执行大多数的存储器访问。此外，对于在主存储器中不完全匹配的问题，两个最重要的层次是内部存储器和外部存储器。这里的差异更大，通常对于外部存储设备如磁盘的访问时间是内部存储器的  $100\,000 \sim 1\,000\,000$  倍。

换个角度看这个数字，想象有位在巴尔的摩的学生想发送一条要钱的消息给他在芝加哥的父母。如果学生给他的父母发送电子邮件，大约 5 秒消息可以到达他们的家用电脑。将这种模式下的通信对应于访问 CPU 的内部存储器。另一种通信方式对应于访问外部存储器，慢 500 000 倍，就是该学生亲自步行到芝加哥传递消息，如果他可以平均每天 20 英里，这将需要一个月的时间，因此我们应该要尽可能少地访问外部存储器。

尽管在不同层的访问存在巨大差异，但大多数算法设计时并没有考虑到存储层次。事实上到目前为止，在这本书中描述的所有算法假定所有的存储访问都是平等的。这种假设似乎起初是一个巨大的疏忽，而且我们只是在最后一章提出，但有很好的理由进行这个合理的假设。

假定所有存储器的访问需要相同时间的理由之一是，因为有些特定设备的内存的大小信息往往很难得到。事实上，关于存储器大小的信息可能很难得到。例如，很难在某一个特定的计算机体系结构配置中定义一个在许多不同的计算机平台上运行的 Python 程序。当然，可以使用特定的架构信息，如果我们使用它的话（我们将在本章后面说明如何开发这些信息）。但是，一旦在一定的架构配置中优化了软件，软件将不再是设备无关的。幸运的是，这样的优化不总是必要的，第二个理由是假设所有存储器的访问需要相同时间。

### 缓存与分块

内存访问平等假设的另一个理由是，操作系统的设计师已经开发了通用的机制，允许更快访问内存。这些机制是基于大多数软件所具备的两个重要的局部参考特性：

- **时间局部性。**如果程序访问一个特定的内存位置，那么在不久的将来它再次访问相同位置的可能性会增加。例如，在几个不同的表达式使用计数器变量的值是很常见的，包括递增计数器的值。事实上，计算机架构师共同的格言是，在一个程序所花费的 90% 的时间在其 10% 的代码上。
- **空间局部性。**如果程序访问某个内存位置，它不久之后访问附近的其他位置的可能性会增加。例如，程序使用矩阵时，可能会以连续或近乎连续的方式访问矩阵的位置。

计算机科学家和工程师们进行了大量的软件分析实验证明，大多数软件都具备这类局部参考特性。例如，嵌套 for 循环重复扫描矩阵将显示出这两种局部性。

反过来，时间局部性和空间局部性为多层计算机存储器系统提供了两个基本设计选择（其实存在于高速缓冲存储器和内部存储器之间的接口，以及在内部存储器和外部存储器之间的接口）。

第一种设计选择称为虚拟内存。这个概念包括提供和二级存储器容量一样大的地址空间，只有当被寻址时，才将位于第二层的数据传送到第一层。虚拟存储器不限制程序员对内部存储器容量的约束。将数据存到主存储器的概念称为高速缓存，它是由时间局部性的特性促使的。通过将数据存入主存储器中，我们希望它会很快再次访问，并且在不久的将来将能

够快速响应所有这些数据的请求。

第二种设计选择是由空间局部性促使的。具体来讲，如果要访问存储在第二级存储器的数据，那么将一个大的连续空间包括要访问的位置的块存入第一级存储器（见图 15-2）。这个概念称为分块，并且它是由很快会访问第二级存储器相邻位置的期望所促使的。在高速缓冲存储器和内部存储器间的接口中，这种块通常称为高速缓存行，并且在内部存储器和外部存储器间的接口中，这种块通常称为页。

缓存和分块实现的虚拟内存往往让我们察觉到两级存储器的速度比实际上的快得多。但是，还有一个问题，一级存储器比二级存储器内存小得多。此外，由于存储系统使用分块，当一些程序可能达到它从二级存储器请求数据的点时，但一级存储器中的块有可能满了。为了满足该请求，并保持使用缓存和分块，在这种情况下我们必须从一级存储器取出一些块，以腾出空间给从二级存储器取出的新块。

### 浏览器中的缓存

决定依次取出哪些块给数据结构和算法设计带来了一些有趣的问题。为此，我们考虑当再次访问网页时出现的相关问题。根据时间局部性来看，在缓存中存储网页副本是有好处的，当请求再次发生时，它能够快速检索到这些页面。这有效地创建了一个以缓存作为更小、更快的内部存储器和网络作为外部存储器的两级存储器层次结构。特别是，假设有一个  $m$  个“插槽”的缓冲存储器，可以包含 Web 页面，假设一个网页可放置在高速缓存中的任何一个插槽中。这称为全相联高速缓冲存储器。

浏览器会运行不同的网页。每次浏览器请求这样一个网页  $p$ ，浏览器确定（使用快速测试）网页  $p$  是否改变且当前是否在高速缓存中。如果网页  $p$  在高速缓存中，则该浏览器使用缓存副本就能满足请求。如果网页  $p$  不在高速缓存中，对于网页  $p$  的页面请求将要搜索整个因特网，并传输到缓存中。如果高速缓存中的  $m$  个插槽中的一个是可用的，则浏览器将网页  $p$  分配到任一个空槽中。但是，如果高速缓冲存储器的  $m$  个单元都被占用时，计算机必须确定取出哪些先前浏览过的网页然后驱逐，并由网页  $p$  代替。当然，有很多不同的策略用来确定网页的取出。

### 页面置换策略

一些较知名的页面替换策略（见图 15-3）如下：

- 先进先出策略（FIFO）。置换在主存中停留时间最长的页面，也就是在最远的过去传输到缓存中的页面。
  - 最近最久未使用策略（LRU）。置换在过去最远一次请求的页面。
- 此外，我们可以考虑一个简单的、纯随机的策略。
- 随机策略。在缓存中随机置换一个页面。

随机策略是最容易实施的策略之一，因为它仅需要一个随机或伪随机数生成器。参与实施这一策略的开销是每一个页面置换所需的额外空间  $O(1)$ 。此外，对于每个页面请求都没有额外的开销，除了确定该页是否是在缓存中。不过，这一策略并没有试图采取根据用户的浏览表现出任何时间局部性的优势。

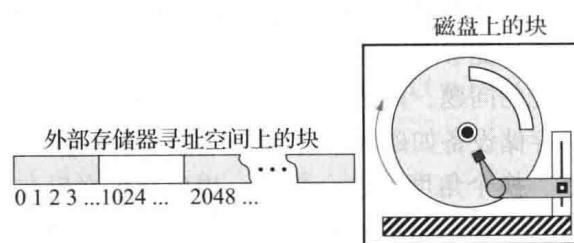


图 15-2 外部存储器中的块

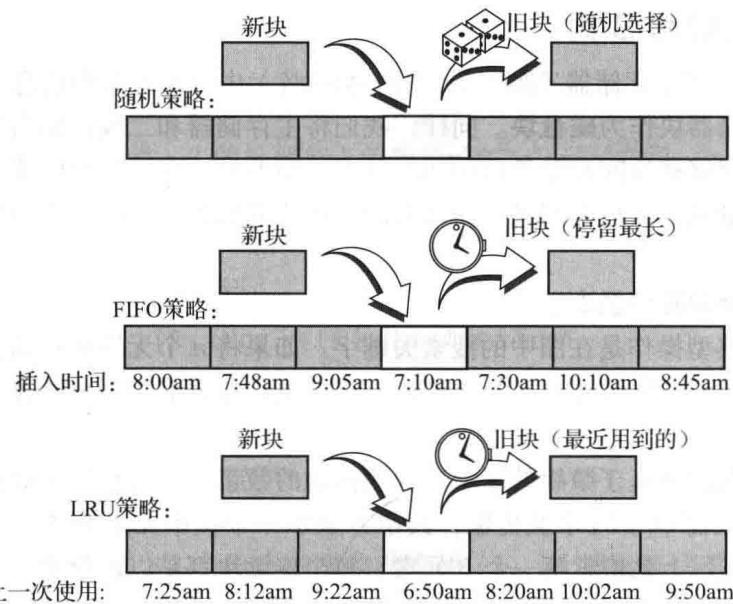


图 15-3 随机策略、FIFO 策略和 LRU 页面置换策略

先进先出策略的实现相当简单，因为它仅需要一个队列  $Q$  来存储缓存中引用的页面。当被浏览器引用时，页面会进入队列  $Q$ ，并且会存入到高速缓存。当页面需要被取出时，计算机简单地执行队列  $Q$  的出队操作来确定置换哪个页面。因此，这一策略还需提供每个页面替换所需的  $O(1)$  空间。同时，FIFO 策略对页面请求没有额外的开销，而且它试图展示一些时间局部性的优势。

LRU 策略比 FIFO 策略更深一步，总是通过取出最近最少使用的页面来尽可能多地展示时间局部性的优势。从策略的角度来看，这是一个很好的方法，但从实现成本的角度看花费大。也就是说，它优化时间和空间局部性的方式是相当昂贵的。实施 LRU 策略需要使用支持更新现有网页的优先级的可适应优先级队列  $Q$ 。如果  $Q$  是基于链表来实现排序序列，则每个页面请求和页面置换的开销需要  $O(1)$  空间。当  $Q$  中插入一个页面或更新其优先级时，该页面在  $Q$  中被赋予最高级，并且放置在链表的末尾，这需要在  $O(1)$  时间内完成。虽然 LRU 策略具有固定的时间开销，但使用上述的实施方式所涉及的常量条件包含额外时间的开销和用于优先队列  $Q$  上的额外的空间，从实用角度来看这个策略缺少吸引力。

由于这些不同的页面替换策略有不同的实施难度和展示局部性优势的程度，自然，应对比分析这些方法，来看看哪一种方法是最好的。

从最坏情况的角度看，FIFO 和 LRU 策略是相当没有吸引的行为。例如，假设在高速缓存有  $m$  个页面，对于一个循环请求  $m+1$  个页面的程序考虑用 FIFO 和 LRU 策略进行页面置换。无论是 FIFO 还是 LRU 策略，对这样序列的页面请求实现效果都很差，因为它们对于每一个页面请求都要进行页面替换。因此，从最坏的情况来看，我们可以想象这些策略几乎是最糟糕的——对于每一个页面请求，它们都需要页面替换。

这种最坏情况的分析是有点过于悲观，但是它集中于每个协议的页面请求的一个坏序列的行为。理想的分析是在所有可能的页面请求序列比较这些方法。当然，不可能做到详尽，但已经有大量的实验模拟来自真实程序的页面请求序列。基于这些实验的比较，LRU 策略已被证明是优于 FIFO 策略，当然通常比随机策略更好。

### 15.3 外部搜索和 B 树

考虑到不适合在主存储器（如一个典型的数据库）中维护大集合信息。在这方面，我们指的是将二级存储器块作为磁盘块。同样，我们将主存储器和二级存储器间块的传输作为磁盘传输。回顾主存储器访问和磁盘访问间的巨大时间差异，在外部存储器中维护大集合信息的主要目标是尽量减少执行查询或更新所需的磁盘传输的数量。我们指的是该算法所涉及的 I/O 复杂性。

#### 一些低效的外部存储器表示

我们支持的典型操作是在图中的搜索关键字。如果将  $n$  个无序项存储在双向链表中，在链表中搜索特定键在最坏情况下需  $n$  次传输，因为执行链表上的每个链表节点可能会访问存储器中的不同块。

我们可以通过使用基于数组的序列减少块传输的数量。因为空间局部性原理，执行一个数组的有序搜索只需  $O(n/B)$  个块传输，其中  $B$  表示一个块中元素的数目。这是因为访问数组的第一个元素实际上是检索第一个  $B$  元素，每个连续块都是以此类推。值得一提的是，仅使用紧凑数组表示时才能达到  $O(n/B)$  的块传输（参见 5.2.2 节）。标准的 Python 列表类是一个引用容器，所以即使按引用序列存储在数组中，在搜索期间被检查的实际元素一般不按顺序存储在存储器中，从而导致在最坏的情况下需要  $n$  个块传输。

我们可以用一个有序数组存储序列。在这种情况下，通过二分搜索，只需执行  $O(\log_2 n)$  次传输，这是一个很好的改进。但是，不能从块传输得到显著的好处，因为二分搜索过程中每个查询可能会在不同的块中进行。通常，更新操作对有序数组来说成本很高。

由于这些简单的实现 I/O 效率低下，我们应该考虑对数时间内部存储策略，即使用平衡二叉树（如 AVL 树或红黑树）或对数平均情况下查询和更新的其他搜索结构（如跳转表或伸展树）。通常，在这些结构中查询或更新所访问的每个节点将是在不同的块中进行。因此，这些方法在最坏的情况下执行查询或更新操作都需要  $O(\log_2)$  次传输。但是，我们可以做得更好！——可以执行批量查询和更新只用  $O(\log_B n) = O(\log n / \log B)$  次传输。

#### 15.3.1 (a, b) 树

为了减少搜索时外部存储器访问的次数，可以使用多路搜索树（见 11.5.1 节）来表示图。这种方法产生了  $(2, 4)$  树数据结构，也称为  $(a, b)$  树。

$(a, b)$  树是一棵多路搜索树，它的每个节点具有  $a \sim b$  个孩子节点，存储着  $(a-1) \sim (b-1)$  个记录。在  $(a, b)$  树中搜索，插入和删除记录的算法是  $(2, 4)$  树直接缩影。 $(2, 4)$  树缩影到  $(a, b)$  树的优点在于，一棵广义类树提供了一个灵活的搜索结构，其中节点的多少和各种映射操作的运行时间取决于参数  $a$  和参数  $b$ 。通过设置参数  $a$  和  $b$  来处理磁盘块的大小，我们可以根据该数据结构取得良好的外部存储性能。

#### (a, b) 树的定义

$(a, b)$  树的参数  $a$  和  $b$  是整数且，满足  $2 \leq a \leq (b+1)/2$ 。 $(a, b)$  树是一棵多路搜索树，具有以下附加限制：

- 大小属性：每个内部节点至少有  $a$  个孩子节点，至多有  $b$  个孩子节点，根节点除外。
- 深度属性：所有外部节点具有相同的深度。

**命题 15-1：** 存储  $n$  个记录的  $(a, b)$  树的高度是  $O(\log n / \log b)$  到  $O(\log n / \log a)$  之间。

**证明：** 设  $T$  是存储  $n$  个记录的  $(a, b)$  树， $h$  是  $T$  的高度。我们通过建立如下等式来证

明这个命题。

$$\frac{1}{\log b} \log(n+1) \leq h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1$$

根据大小属性和深度属性， $T$  的外部节点的数量  $n^h$  在  $2a^{h-1}$  到  $b^h$  之间。

根据命题 11-7 可知， $n^h = n + 1$  因此，

$$2a^{h-1} \leq n + 1 \leq b^h$$

再同时取从 2 为底的对数，得到：

$$(h-1)\log a + 1 \leq \log(n+1) \leq h \log b$$

通过不等式运算完成以上证明。 ■

### 搜索和更新操作

回顾在多路搜索树  $T$  中， $T$  的各节点  $v$  持有二级结构  $M(v)$ ，这本身就是一个图（见 11.5.1 节）。如果  $T$  是  $(a, b)$  树，那么  $M(v)$  最多存储  $b$  条记录。令  $f(b)$  表示在图  $M(v)$  中执行搜索中的时间。这与在 11.5.1 节给出的多路搜索树  $(a, b)$  搜索算法是完全一样。因此，一棵有  $n$  条记录的  $(a, b)$  树  $T$  需要  $O(f(b)/\log a * \log n)$  的时间。注意，如果  $b$  为常数（并且  $a$  也是），那么搜索时间为  $O(\log n)$ 。

$(a, b)$  树主要用于存储在外部存储器的映射。也就是说，要尽量减少磁盘访问，我们选择参数  $a$  和  $b$ ，使每个树节点占用一个磁盘块（如果我们想简单地计算块传输，则令  $f(b) = 1$ ）。在这种情况下提供合适的  $a$  和  $b$  值会产生一个数据结构，我们简述为 B 树。在我们描述这种结构前，但是，让我们来讨论如何在  $(a, b)$  树中进行插入和删除。

$(a, b)$  树的插入算法类似于  $(2, 4)$  树。当在  $b$  节点  $w$  中插入记录时，就会成为非法的  $(b+1)$  节点，此时发生上溢。（一个多路树中的一个节点如果它有  $d$  个孩子，就是  $d - 1$  节点。）为了补救上溢，我们移动  $w$  的一半记录给其父母节点，并将  $w$  替换为  $[(b+1)/2]$  节点  $w'$  和  $[(b+1)/2]$  节点  $w''$ 。现在我们明白了在  $(a, b)$  树的定义中为什么需要  $a \leq (b+1)/2$ 。注意到分散的结果，我们需要构建两个二级结构  $M(w')$  和  $M(w'')$ 。

从  $(a, b)$  树中删除记录的算法也类似于  $(2, 4)$  树。当在  $a - 1$  节点  $w$  中删除一条记录时，就会成为非法的  $(a-1) - 1$  节点，此时发生下溢，根节点除外。为了补救下溢，我们通过将  $w$  的兄弟节点转换成非  $a$  节点，或将  $w$  与其兄弟节点融合成  $a$  节点，合成的新节点是  $(2a-1)$  节点。这是需要  $a \leq (b+1)/2$  的另一个理由。表 15-1 显示了  $(a, b)$  树的性能。

表 15-1 由  $(a, b)$  树  $T$  实现的  $n - 1$  节点的时间界限。假定  $T$  节点的二级结构对  $f(b)$  函数和  $g(b)$  函数支持在  $f(b)$  时间内搜索，在  $g(b)$  时间内分开和合成。当只计算磁盘传输时，时间复杂度能达到  $O(1)$

操作	运行时间
$M[k]$	$O\left(\frac{f(b)}{\log a} \log n\right)$
$M[k] = v$	$O\left(\frac{g(b)}{\log a} \log n\right)$
$\text{del } M[k]$	$O\left(\frac{g(b)}{\log a} \log n\right)$

### 15.3.2 B树

(a, b) 树数据结构中的一个版本，也是外部存储器维护信息常用的方法，称为 B 树（见图 15-4）。一个  $d$  阶 B 树，满足  $a = \lceil d/2 \rceil$  和  $b = d$ 。既然已经讨论了 (a, b) 树的标准查询和更新方法，这里只讨论 B 树的 I/O 复杂性。

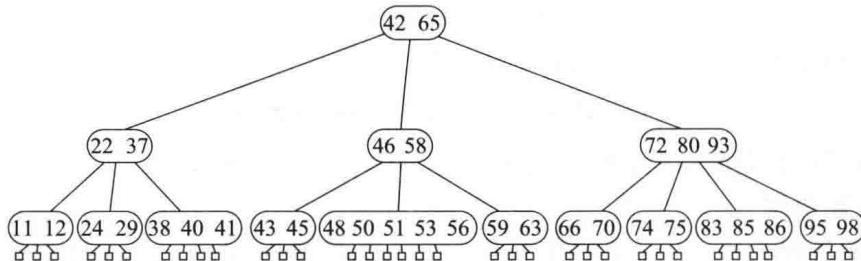


图 15-4 一个 6 阶的 B 树

B 树的一个重要属性是可以选择  $d$ ，使得一个节点中存储的  $d - 1$  个键和  $d$  个孩子的引用可以紧凑地装入一个磁盘块，这意味着  $d$  与  $B$  成正比。这种选择允许我们在 (a, b) 树中的搜索和更新操作的分析中假设  $a$  和  $b$  正比于  $B$ 。因此，每次访问一个节点来执行搜索或更新操作时， $f(b)$  和  $g(b)$  的时间复杂度都是  $O(1)$ ，只需要执行一个块传输。

通过上述观察，检测到树的每一次执行搜索或更新操作最多需要  $O(1)$  节点，因此对 B 树的任意搜索或更新仅需要  $O(\log \lceil d/2 \rceil n)$ ，也就是  $O(\log n / \log B)$  次块传输。例如，对 B 树完成一次操作就是在节点中插入新记录，如果由于此操作，节点上溢（有  $d + 1$  个孩子节点），那么该节点会分成两个节点，分别有  $\lfloor (d + 1)/2 \rfloor$  和  $\lceil (d + 1)/2 \rceil$  个孩子节点。该过程在接下来的每一层都重复此操作，直至到达  $O(\log_B n)$  层。

同样，如果删除操作导致一个节点下溢（有  $\lceil d/2 \rceil - 1$  个孩子节点），那么使用至少有  $\lceil d/2 \rceil + 1$  个孩子节点的兄弟节点或将这个节点与其兄弟节点融合（父母节点重复此操作）。同插入操作一样，这将向上继续执行至多  $O(\log_B n)$  层。每个内部节点至少具有  $\lceil d/2 \rceil$  个孩子节点意味着用于支持 B 树的每个磁盘块至少有一半空间是满的。因此，有以下结论：

**命题 15-2：**  $n$  个记录的 B 树的搜索和更新操作的 I/O 复杂度为  $O(\log_B n)$ ，并且使用  $O(n/B)$  个块，其中  $B$  是块的大小。

## 15.4 外部存储器中的排序

除了数据结构（例如映射）需要在外部存储器实现，还有许多算法也必须在输入集合上操作，它们太大了，以至于不能完全适用于内存。在这种情况下，对象尽可能少使用块传输来解决算法问题。这种使用外部存储器的最典型的算法是排序问题。

### 多路归并排序

在外部存储器上对有  $N$  个对象的集合  $S$  进行排序是一个有效的方法，相当于我们熟悉的归并分类算法上一个简单的外部存储变量。这种变量背后的主要思想是同时递归地合并排序列表，从而减少递归的次数。具体来说，这种多路归并排序（multiway merge-sort）方法的一个高层次的描述是把  $S$  分为规模大致相当的  $d$  个子集  $S_1, S_2, \dots, S_d$ ，递归地排序每一个子集  $S_i$ ，然后同时将所有  $d$  个已经排好序的列表合并为一个  $S$  的排过序的形式。如果我们可以只使用  $O(n/B)$  次磁盘传输执行合并过程，那么对于足够大的  $n$ ，由算法执行的传输总量满

足如下递归：

$$t(n) = d \cdot t(n/d) + cn / B$$

对于一些常数  $c \geq 1$ , 当  $n \leq B$  时可以停止递归, 因为在这一节点上我们可以执行单个块传输, 使所有的对象到内存中, 然后用一个高效的内部存储算法对这些集合排序。因此,  $t(n)$  的停止准则是:

$$t(n) = 1, \text{ 如果 } n / B \leq 1$$

这意味着一个闭合解, 其中  $t(n)$  是  $O((n/B)\log_d(n/B))$ , 这是

$$O((n/B)\log(n/B)/\log d)$$

因此, 如果我们可以选择  $d$  作为  $\Theta(M/B)$ , 其中  $M$  是内存的大小, 然后最坏情况下这种多路归并算法执行块传输的数量将会变得非常少。基于在下一节中将给出的原因, 我们选择

$$d = (M/B) - 1$$

该算法留给我们的唯一选择是如何只使用  $O(n/B)$  次块传输来执行  $d$  路合并。

## 多路合并

在一个标准的合并排序中(见 12.2 节), 合并过程通过在两个序列各自开头反复提取最小项来将两个已经排过序的序列合并为一个序列。在  $d$  路合并中, 在  $d$  个序列开头我们反复寻找最小项, 并将其作为合并序列的下一个元素, 直到所有的元素都包括在内才停止。

在外部存储排序算法的背景下, 如果内存的大小是  $M$ , 并且每一块的大小为  $B$ , 在任意的给定时间, 我们在主存中可以存储多达  $M/B$  的块。我们专门选择  $d = (M/B) - 1$ , 使得在任意的给定时间内主存中的每个输入序列能保留一块, 并有一个额外的块用作合并序列的缓冲, 如图 15-5 所示。

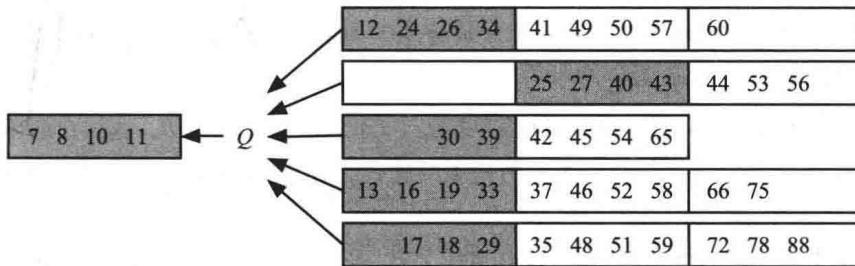


图 15-5  $d = 5, B = 4$  的  $d$  路合并。块在主存中用灰色表示

我们保持内存中每个输入序列中最小的未加工的元素, 当前一块用完时, 从一个序列中请求下一块。同样, 我们使用内存的一块来缓冲合并序列, 当缓冲满了的时候刷新外存的块。通过这种方式, 单一  $d$  路合并中执行的传输总数是  $O(n/B)$ , 因为我们每扫描列表  $S_i$  一次, 就写合并列表  $S'$  一次。根据计算时间, 选择可以使用  $O(d)$  次操作执行的最小  $d$  值。如果愿意使用  $O(d)$  的内存, 可以在每个队列中保持一个优先队列以识别最小的元素, 从而在  $O(\log d)$  时间内通过删除最小的元素并用同一序列的下一个元素取代它来进一步合并。因此,  $d$  路合并的内部时间是  $O(n \log d)$ 。

**命题 15-3:** 给定一个紧密地存储在外存中  $n$  个元素的基础数组序列  $S$ , 我们可以用  $O((n/B)\log(n/B)/\log(M/B))$  块传输和  $O(n \log n)$  的内部计算对  $S$  排序, 其中  $M$  是内存的大小,  $B$  是一个块的大小。

## 15.5 练习

请访问 [www.wiley.com/college/goodrich](http://www.wiley.com/college/goodrich) 以获得练习帮助。

### 巩固

- R-15.1 Julia 刚买了一台新的计算机，使用 64 位整数来处理内存单元。解释为什么 Julia 在她的生活中将永远不会更新她的电脑的内存，这可能将会是她的电脑内存的最大尺寸。假设你需要有不同的公式来代表不同的比特。
- R-15.2 详细地描述从一个  $(a, b)$  树上添加或删除一项的算法。
- R-15.3 假设  $T$  是一棵多叉树，其中每个内部节点至少有 5 个、最多有 8 个孩子。当  $a$  和  $b$  的值为多少的时候  $T$  是一棵有效的  $(a, b)$  树？
- R-15.4 当  $d$  的值为多少时，上一题中的树  $T$  是一个  $d$  阶的 B 树。
- R-15.5 考虑一个由四页组成的初始为空的内存缓存。LRU 算法会导致页面请求序列  $(2,3,4,1,2,5,1,3,5,4,1,2,3)$  有多少缺页？
- R-15.6 考虑一个由四页组成的初始为空的内存缓存。FIFO 算法会导致页面请求序列  $(2,3,4,1,2,5,1,3,5,4,1,2,3)$  有多少缺页？
- R-15.7 考虑一个由四页组成的初始为空的内存缓存。随机算法会导致页面请求序列  $(2,3,4,1,2,5,1,3,5,4,1,2,3)$  的最大缺页数是多少？演示在这种情况下所有算法产生的随机选择。
- R-15.8 画出插入到初始为空的 7 阶 B 树的结果，条目的键为  $(4,40,23,50,11,34,62,78,66,22,90,59,25,7,2,64,77,39,12)$ 。

### 创新

- C-15.9 描述一个有效的外部存储算法，用于删除大小为  $n$  的数组列表中的所有副本项目。
- C-15.10 描述一个外部存储数据结构来实现堆栈 ADT，使需要处理一个  $k$  队列的 push 和 pop 操作的磁盘总数是  $O(k/B)$ 。
- C-15.11 描述一个外部存储数据结构来实现队列 ADT，使需要处理一个  $k$  队列的 enqueue 和 dequeue 操作的磁盘总数是  $O(k/B)$ 。
- C-15.12 描述一个 PositionalList ADT 的外部存储版（7.4 节），块大小为  $B$ ，这样在最坏的情况下使用  $O(n/B)$  传输来完成一个长度为  $n$  的列表的迭代，并且 ADT 的所有其他方法只需要  $O(1)$  传输。
- C-15.13 改变定义红黑树的规则，使每一棵红黑树  $T$  有一个相应的  $(4,8)$  树，反之亦然。
- C-15.14 描述一个 B 树插入算法的改进版本，使我们每次因为节点  $w$  的分裂创建溢出时，对所有  $w$  的兄弟再分配键，让每个兄弟拥有大致相同的键值（可能是  $w$  父亲的级联分裂）。使用这种方案填充每块的最小部分是多少？
- C-15.15 另一个可能的外部存储映射的实现是使用跳跃表，但是在跳跃表的任意层级上，要在单个块中收集  $O(B)$  节点的连续的组。特别地，我们定义一个  $d$  阶的 B 跳跃表来表示表结构，其中每块包含至少  $d/2$  个列表节点和最多  $d$  个列表节点。在这种情况下，也选择  $d$  作为一个可以容纳块的跳跃表级别中列表节点的最大数。描述对于一个 B 跳跃表，我们如何修改插入和删除算法以使结构的预期高度为  $O(\log n/\log B)$ 。
- C-15.16 描述如何使用 B 树实现分区（合并 - 查找）ADT（见 14.7.3 节），使合并和查找操作每次最多使用  $O(\log n/\log B)$  的磁盘传输。
- C-15.17 假设我们给定一个有  $n$  个整数键元素的队列  $S$ ，使在  $S$  中的一些元素是“蓝色”，一些是“红

色”。此外，如果键值相同，则一个红色元素  $e$  匹配一个蓝色元素  $f$ 。为寻找  $S$  中所有的红 - 蓝对，请描述一个有效的外部存储算法。你的算法有多少磁盘传输需要执行？

- C-15.18 考虑页面缓存问题，内存缓存可以容纳  $m$  页，并且我们给定一个有  $n$  个请求的序列  $p$  取自  $m + 1$  个可能的页面池。为脱机算法描述最佳策略，并显示它一共能导致最多  $m + n / m$  的缺页，从一个空的缓存开始。
- C-15.19 描述一个有效的外部存储算法，该算法可以确定是否一个大小为  $n$  的整型数组包含一个出现次数大于  $n/2$  的值。
- C-15.20 考虑到页面缓存策略，基于最不经常使用 (LFU) 规则，当请求新的页面时淘汰最不经常进入缓存中的页面。如果有相同使用频率的页面，LFU 淘汰最不常使用的且缓存时间最长的页面。现有一个  $n$  个请求的序列  $P$ ，证明对于  $m$  页的缓存 LFU 可引起  $\Omega(n)$  次缺页，然而最优算法将只引起  $O(m)$  次缺页。
- C-15.21 假设在  $d$  阶 B 树  $T$  中有节点搜索函数  $f(d) = 1$ ,  $f(d) = \log d$ 。在  $T$  中执行搜索的渐近运行时间现在变成了多少？

## 项目

- P-15.22 写一个 Python 类，该类模拟内存管理的最佳适应、最坏适应、首次适应和循环首次适应算法。用实验的方法确定在请求各种内存序列的情况下哪种方法是最好的。
- P-15.23 写一个 Python 类，借助  $(a, b)$  树实现所有有序映射 ADT 方法，其中  $a$  和  $b$  是作为参数传递给构造函数的整型常量。
- P-15.24 实现 B 树数据结构，假设一个块的大小为 1024 个整型键。测试“磁盘传输”所需的数量来处理一个映射操作序列。

## 拓展阅读

对层次存储器体系结构系统研究感兴趣的读者可以参考 Burger<sup>[21]</sup> 等人的书或者 Hennessy 和 Patterson<sup>[50]</sup> 的书。我们描述的标记 - 清除算法这种垃圾收集方法是执行垃圾收集的许多不同算法之一。我们鼓励对进一步研究垃圾收集感兴趣的读者研究 Jones 和 Lins<sup>[62]</sup> 的书。Knuth<sup>[62]</sup> 对于外部存储器分类和搜索有非常好的论述。Ullman<sup>[97]</sup> 讨论了数据库系统的外存结构。Gonnet 和 Baeza Yates<sup>[44]</sup> 的手册比较了多个不同的排序算法的性能，其中有许多是外部存储器算法。B 树是由 Bayer 和 McCreight<sup>[11]</sup> 和 Comer<sup>[28]</sup> 发明的，并对该数据结构提供了非常好的概述。Mehlhorn<sup>[76]</sup> 和 Samet<sup>[87]</sup> 的书对于 B 树和它们的变形也有很好的论述。Aggarwal 和 Vitter<sup>[3]</sup> 研究分类的 I/O 复杂性及相关问题，建立了上界和下界。Goodrich 等人<sup>[46]</sup> 研究几种计算几何问题的 I/O 复杂性。鼓励有兴趣进一步研究 I/O 算法的读者研究 Vitter<sup>[99]</sup> 的调查论文。

## 附录 A |

Data Structures and Algorithms in Python

# Python 中的字符串

字符串是来自字母表的一些字符序列。在 Python 中，内置的 str 类表示基于 Unicode 国际字符集的字符串、一个 16 位的字符编码，涵盖了大多数书面的语言。Unicode 是包括基本拉丁文字母、数字和常见符号的 7 位 ASCII 字符集的扩展。字符串在多数编程应用中特别重要，因为文本通常用于输入和输出。

1.2.3 节提供了关于 str 类的基本介绍，包括使用的字符串，如 'hello' 和用于构造一个典型的对象的字符串表示形式的语法 str(obj) 等。1.3 节进一步讨论了常用的运算符支持的字符串，例如使用 “+” 进行连接。本附录作为更详细的参考，描述字符串支持文本处理的快捷操作。为了描述 str 类的行为，我们将它分为以下几大类的功能。

### 搜索子串

操作符语法 s 中的方法，可以确定给定的模式是否是字符串 s 的子串。表 A-1 描述了几种相关的方法，确定搜索的数量和索引是从最左边或最右边开始。表中的每个函数接收两个可选的参数，分别为 start 和 end，可以有效地将搜索限制到 start 和 end 之间，即 s[start:end]。例如，调用 s.find (pattern, 5) 可以将搜索限制到 s[5:]。

表 A-1 搜索子串的方法

调用语法	描    述
s.count(pattern)	返回与 pattern 不重叠的匹配项数目
s.find(pattern)	返回索引最左边以 pattern 开始；否则返回 -1
s.index(pattern)	和 find 方法类似，但是如果没有找到，会提高 ValueError
s.rfind(pattern)	返回索引最右边以 pattern 开始；否则返回 -1
s.rindex(pattern)	和 rfind 方法类似，但是如果没有找到，会提高 ValueError

### 构建相关的字符串

在 Python 中，字符串是不可变的，所以它们的方法都不修改现有的字符串实例。然而，许多方法返回一个新建的字符串，它与一个现有的字符串密切相关。表 A-2 总结了这类方法，其中包括用新的字符串替换当前字符串，更改字母的大小写，根据需要产生一个宽度固定字符串，产生从任一端剥离无关字符字符串的备份。

表 A-2 相关字符串的方法

调用语法	描    述
s.replace(old, new)	返回一用新匹配项替代所有旧匹配项的 s 的备份
s.capitalize()	返回其拥有的第一个字符大写的 s 一个备份
s.upper()	返回所有字符都大写的 s 的一个备份
s.lower()	返回所有字符都小写的 s 的一个备份
s.center(width)	返回 s 的一个拷贝，中间用空格填充相应的宽度
s.ljust(width)	返回 s 的一个拷贝，结尾用空格填充相应的宽度

(续)

调用语法	描述
s.rjust(width)	返回 s 的一个拷贝，开头用空格填充相应的宽度
s.zfill(width)	返回 s 的一个拷贝，开头用 0 填充相应的宽度
s.strip()	返回 s 的一个拷贝，删除开头和结尾无用的空白
s.lstrip()	返回 s 的一个拷贝，删除开头无用的空白
s.rstrip()	返回 s 的一个拷贝，删除结尾无用的空白

表中几个函数接收的可选参数没有详细的说明。例如，replace() 方法默认情况下替换所有不重叠的旧有模式，但可选参数可以限制进行替换的数量。居中或两端对齐处理文本的方法使用空格作为默认填充字符进行填充，但是可选填充字符可以被指定为一个可选参数。

同样，所有删除字符的变形默认情况下都是删除开头和结尾的空格，但是一个可选的参数可以选定应从两端开始删除的字符。

### 测试布尔条件

表 A-3 包括测试一个字符串的布尔属性，例如是否它某一种方式开始或结束，或其字符是否由字母、数字、空白等的组成的方法。标准 ASCII 字符集是由字母字符即大写 A ~ Z 和小写 a ~ z，数字即 0 ~ 9，空白（包括空格、制表符、换行符和回车）组成的，被视为字母和数字的字符代码推广到更一般的 Unicode 字符集合。

表 A-3 测试布尔条件的方法

调用语法	描述
s.startswith(pattern)	如果 pattern 是字符串 s 的前缀，返回 True
s.endswith(pattern)	如果 pattern 是字符串 s 的后缀，返回 True
s.isspace()	如果非空字符串的所有字符是空白，返回 True
s.isalpha()	如果非空字符串的所有字符是字母，返回 True
s.islower()	如果所有字母都是小写的，返回 True
s.isupper()	如果所有字母都是大写的，返回 True
s.isdigit()	如果非空字符串的所有字符都是在 0 和 9 之间，返回 True
s.isdecimal()	如果非空字符串的所有字符代表是数字 0 ~ 9 包括 Unicode 等价物，返回 True
s.isnumeric()	如果非空字符串的所有字符都是数字包括 Unicode 字符（例如，0 ~ 9、等价物、分数字符），则返回 True
s.isalnum()	如果非空字符串的所有字符都是字母或数字（根据上述定义），返回 True

### 拆分和连接字符串

表 A-4 介绍了 Python 中 string 类的几种重要方法，用来将一系列字符串序列连接起来——通过使用分隔符来分隔每对序列或采取现有的字符串，并根据给定的分解模式确定该字符串分解。

表 A-4 拆分和连接字符串的方法

调用方法	描述
sep.join(strings)	返回给定字符串组成的序列，将 sep 作为分隔符插入每对序列之间
s.splitlines()	返回字符串 s 的子串列表，以换行符分隔
s.split(sep, count)	返回字符串 s 的子串列表，以 sep 作为分隔符分隔 count 次。如果不指定 count，则分隔所有；如果不指定 sep，则使用空格作为分隔符

(续)

调用方法	描述
s.rsplit(sep, count)	类似 split() 方法，但是使用最右边出现的 sep
s.partition(sep)	使用最左边出现的 sep 让 s = head + sep + tail, 返回 (head, sep, tail), 否则返回 (s, "", "")
s.rpartition(sep)	使用最右边出现的 sep 让 s = head + sep + tail, 返回 (head, sep, tail), 否则返回 (s, "", "")

join() 方法用于把一系列的字符组合成字符串。例如，'and '.join(['red', 'green', 'blue']), 结果是 'red and green and blue'。注意，分隔符字符串中包含空格。相反，命令 'and'.join(['red', 'green', 'blue']) 会产生 'redandgreenandblue' 的结果。

表 A-4 讨论的其他方法提供了和 join() 方法相反的功能，它们利用给定的分隔符把一个字符串分隔成一个子串的序列。例如，命令 'red and green and blue'.split('and ') 会产生结果 ["red", 'green', 'blue']。如果不指定分隔符或者分隔符是空，就利用空格作为分隔符。因此，'red and green and blue'.split() 的结果是 ['red', 'and', 'green', 'and', 'blue']。

### 字符串格式

str 类的格式方法组成了包含一个或多个格式化的参数的一个字符串。语法 s.format(arg0, arg1, ...) 调用的方法，会生成一个或多个参数被替换的格式化的字符串的预期结果。举一个简单的例子，表达式 '{} had a little {}'.format('Mary', 'lamb') 会产生结果 'Mary had a little lamb'。表达式中成对的花括号是在结果中被替代的字段的占位符。默认情况下，传递到该函数的参数替换按照先后顺序，因此 Mary 替换第一个花括号， lamb 替换第二个花括号。然而，替代模式可以被显式的编号改变顺序，或者可以在多个位置使用同一个参数。比如，表达式 '{0}, {0}, {0} your {1}'.format('row', 'boat') 会产生结果 'row, row, row your boat'。

所有替代模式允许使用填充字符和对齐模式来填充参数到一个特定的宽度。比如，'{:-^20}'.format('hello')。在这个例子中，连字符 (-) 作为填充字符插入字符 (^) 选定所需的字符串居中，20 是参数所需的宽度。本示例的结果是字符串 '-----hello-----'。默认情况下，空格作为填充字符并且默认从右边开始填充。

对于数值类型，有额外的格式选项。如果其宽度说明开头是 0，很多会用 0 填充而不是空格填充。比如，日期可以由 '{}/{:02}/{:02}'.format(year, month, day) 转化为传统格式 "YYYY/MM/DD"。整数可以转化二进制、八进制或十六进制分别通过添加字符 b、o 或 x 作为数值的后缀。一个浮点数的精度被小数点和小数点后所需的位数指定。比如，表达式 '{:.3}'.format(2/3) 产生的结果是字符串 '0.667'，精确到小数点后三位。一个程序员可以显示指定使用定点表示法（例如 0.667）通过添加字符 f 作为后缀，或者科学计数法（例如，6.667e - 01）通过添加字符 e 作为后缀来表示小数。

## 有用的数学定理

在这个附录中，我们会给出一些有用的数学定理。先从一些组合的定义和定理开始。

### 对数和指数

对数函数定义为

$$\log_b a = c, \quad a = b^c$$

下面是对数和指数的运算法则：

- 1 )  $\log_b ac = \log_b a + \log_b c$
- 2 )  $\log_b a/c = \log_b a - \log_b c$
- 3 )  $\log_b a^c = c \log_b a$
- 4 )  $\log_b a = (\log_c a) / \log_c b$
- 5 )  $b^{\log_c a} = a^{\log_c b}$
- 6 )  $(b^a)^c = b^{ac}$
- 7 )  $b^a b^c = b^{a+c}$
- 8 )  $b^a / b^c = b^{a-c}$

另外，还有下列规则。

**命题 B-1：**如果  $a > 0, b > 0$ , 并且  $c > a + b$ , 有

$$\log a + \log b < 2 \log c - 2$$

**证明：**这足以显示  $ab < c^2/4$ , 可以证明：

$$\begin{aligned} ab &= \frac{a^2 + 2ab + b^2 - a^2 + 2ab - b^2}{4} \\ &= \frac{(a+b)^2 - (a-b)^2}{4} \leq \frac{(a+b)^2}{4} < \frac{c^2}{4} \end{aligned}$$

自然对数函数  $\ln x = \log_e x$ , 其中  $e = 2.71828\cdots$ , 可以使用下面的表达式来表示：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

另外，

$$\begin{aligned} e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ \ln(1+x) &= x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \dots \end{aligned}$$

有很多有用的不等式和这些函数有关（源于这些函数的定义）。

**命题 B-2：**如果  $x > -1$ ,

$$\frac{x}{1+x} \leq \ln(1+x) \leq x$$

**命题 B-3：**当  $0 \leq x < 1$  时,

$$1+x \leq e^x \leq \frac{1}{1-x}$$

**命题 B-4：**对于任何两个正实数  $x$  和  $n$ ,

$$\left(1 + \frac{x}{n}\right)^n \leq e^x \leq \left(1 + \frac{x}{n}\right)^{n+x/2}$$

### 取整函数和联系

floor 和 ceiling 函数分别定义如下:

1)  $\lfloor x \rfloor$  小于等于  $x$  的最大整数。

2)  $\lceil x \rceil$  大于等于  $x$  的最小整数。

当整数  $a \geq 0$ ,  $b > 0$  时, 取模运算定义为

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$$

阶乘函数定义为

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1)n$$

二项式系数为

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

这是相当于一个定义为从  $n$  项的集合中选择  $k$  个不同项目的不同组合的数目 (和顺序无关)。“二项式系数”一名源于二项式展开:

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^k b^{n-k}$$

也有以下的关系。

**命题 B-5：**如果  $0 \leq k \leq n$ , 那么

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \frac{n^k}{k!}$$

**命题 B-6 (斯特林公式):**

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \varepsilon(n)\right)$$

其中  $\varepsilon(n)$  是  $1/n^2$  的高阶无穷小。

斐波纳契级数是一些数值的迭代, 比如当  $n \geq 2$  时,  $F_0 = 0, F_1 = 1$ , 有  $F_n = F_{n-1} + F_{n-2}$ 。

**命题 B-7：**如果  $F_n$  由斐波纳契级数定义, 则  $F_n \Theta(g^n)$ , 其中  $g = (1 + \sqrt{5})/2$ , 也被称作黄金分割率。

### 求和

这里有很多有用的求和公式。

**命题 B-8：因数求和**

$$\sum_{i=1}^n af(i) = a \sum_{i=1}^n f(i)$$

提供了一个不取决于  $i$  的变形。

**命题 B-9:** 变换顺序:

$$\sum_{i=1}^n \sum_{j=1}^m f(i, j) = \sum_{j=1}^m \sum_{i=1}^n f(i, j)$$

其中有一个特殊的伸缩和公式

$$\sum_{i=1}^n (f(i) - f(i-1)) = f(n) - f(0)$$

经常出现在数据结构或算法的分部分析中。

以下是其他一些经常出现在数据结构和算法分析中的求和公式。

**命题 B-10:**  $\sum_{i=1}^n i = n(n+1)/2$ 。

**命题 B-11:**  $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$ 。

**命题 B-12:** 如果  $k \geq 1$  且是一个整数常量, 那么

$$\sum_{i=1}^n i^k = \Theta(n^{k+1})$$

另一个常见的求和公式是几何求和,  $\sum_{i=0}^n a^i$ , 对于任意的实数  $0 < a \neq 1$ 。

**命题 B-13:** 对于任意实数  $0 < a \neq 1$ , 有

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

**命题 B-14:** 对于任意实数  $0 < a < 1$ , 有

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

此外, 还有两个常见公式的组合, 被称为线性指数总和, 它有以下扩展。

**命题 B-15:** 对于  $0 < a \neq 1$  且  $n \geq 2$ , 有

$$\sum_{i=1}^n ia^i = \frac{a - (n+1)a^{(n+1)} + na^{(n+2)}}{(1-a)^2}$$

第  $n$  个的谐波数  $H_n$  被定义为

$$H_n = \sum_{i=1}^n \frac{1}{i}$$

**命题 B-16:** 如果  $H_n$  是第  $n$  个谐波数, 则  $H_n$  等于  $\ln n + \Theta(1)$ 。

**基本概率**

回顾概率论中的一些基本公式。最基本的是关于概率的任何语句都是定义在样本空间  $S$  上的。样本空间是指从一些实验中可能出现的所有结果组。我们没有从正式意义上定义术语“outcomes”和“experiment”。

**例题 B-17:** 考虑一个实验, 包括五次投掷硬币的结果。这个样本空间有  $2^5$  的结果, 对于每种不同的结果都有可能出现。

样本空间也可以是无限的，如例 B-2 所示。

**例题 B-18：**考虑这样一个实验，投掷一枚硬币，直到出现正面朝上为止。这个实验中样本空间是无限的，每个结果都是  $i$  次反面朝上后接着出现一次正面朝上， $i = 1, 2, 3, \dots$ 。

概率空间是一个样本空间  $S$  和概率函数  $\Pr$ ，把  $S$  的子集映射到实数区间  $[0,1]$  之间的结果。在数学上概率的概念是一些“事件”发生的可能性。实际上， $S$  的每个子集  $A$  称作一个事件，概率函数  $\Pr$  被认为有以下基本属性，当事件从  $S$  定义时：

- 1)  $\Pr(\emptyset) = 0$ 。
- 2)  $\Pr(S) = 1$ 。
- 3)  $0 \leq \Pr(A) \leq 1$ , 对于任意  $A \subseteq S$ 。
- 4) 如果  $A, B \subseteq S$  且  $A \cap B = \emptyset$ , 则  $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ 。

如果存在下式的关系，则两个事件  $A$  和  $B$  相互独立：

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$$

如果存在下式的关系，则一个事件的集合  $\{A_1, A_2, \dots, A_n\}$  相互独立：

$$\Pr(A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}) = \Pr(A_{i_1}) \Pr(A_{i_2}) \dots \Pr(A_{i_k})$$

对于任意子集  $\{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$ 。

条件概率表示为  $\Pr(A|B)$  是指在事件  $B$  发生的前提下事件  $A$  发生的概率，被定义为一个比率

$$\frac{\Pr(A \cap B)}{\Pr(B)}$$

假定  $\Pr(B) > 0$ 。

使用随机变量来处理事件是一种比较好的方法。直观地说，随机变量是取决于一些实验结果的变量的值。实际上，随机变量是函数  $X$  把一些样本空间  $S$  映射到实数上的结果。随机指示变量是随机变量把结果映射到集合  $\{0, 1\}$  上。在数据结构和算法分析中，经常使用随机变量  $X$  以描述随机算法的运行时间。在这种情况下，样本空间  $S$  被定义为在算法中使用的随机源可能出现的所有结果。

我们对一个随机变量的典型值、平均值或者“期望值”最感兴趣。随机变量  $X$  的期望值定义为

$$E(X) = \sum_x x \Pr(X = x)$$

其中求和函数定义在  $X$  的定义域上（在这种情况下假定为离散的）。

**命题 B-19 (期望的线性运算)：**假设  $X$  和  $Y$  是随机变量， $c$  是一个数字，那么

$$E(X + Y) = E(X) + E(Y) \quad \text{且} \quad E(cX) = cE(X)$$

**例题 B-20：**假设  $X$  是随机变量，表示两个骰子投掷出的点数的总和，那么  $E(X)=7$ 。

**证明：**要证明这个结论，假设  $X_1$  和  $X_2$  是随机变量分别对应于每个骰子的点数。因此， $X_1 = X_2$  (即它们是两个功能相同的实例) 并且  $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$ 。每个结果中每个点数出现的概率都是  $1/6$ 。因此，

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} + \frac{7}{2}$$

其中  $i = 1, 2$ 。因此， $E(X) = 7$ 。

两个随机变量  $X$  和  $Y$  是独立的，如果对任意实数  $x$  和  $y$  有

$$\Pr(X = x | Y = y) = \Pr(X = x)$$

**命题 B-21：**如果两个随机变量  $X$  和  $Y$  是独立的，那么

$$E(XY) = E(X)E(Y).$$

**例题 B-22：**假设  $X$  是一个随机变量，表示随机投掷两枚骰子出现的点数的积，那么  $E(X) = 49/4$ 。

**证明：**假设  $X_1$  和  $X_2$  分别表示两个骰子投掷出的点数。变量  $X_1$  和  $X_2$  明显是独立的，因此

$$E(X) = E(X_1X_2) = E(X_1)E(X_2) = (7/2)^2 = 49/4$$

下面的定理和从它推导出的推论被称为切诺夫界限。■

**命题 B-23：**假设  $X$  是在独立 0/1 有限数字的随机变量的和，并且  $X$  的期望  $\mu > 0$ ，那么，对于  $\delta > 0$ ，

$$\Pr(X > (1 + \delta)\mu) < \left[ \frac{e^\delta}{(1 + \delta)^{(1+\delta)}} \right]^\mu$$

### 有用的数学技术

为了比较不同函数的增长率，有时候可以运用以下规则。

**命题 B-24 (洛必达法则)：**如果有  $\lim_{n \rightarrow \infty} f(n) = +\infty$  并且  $\lim_{n \rightarrow \infty} g(n) = +\infty$ ，那么  $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$ ，其中  $f'(n)$  和  $g'(n)$  分别是  $f(n)$  和  $g(n)$  的导数。

在给定上限和下限进行求和时，会经常用到拆分求和，如下所示：

$$\sum_{i=1}^n f(i) = \sum_{i=1}^j f(i) + \sum_{i=j+1}^n f(i)$$

另一个有用的技术是由积分约束的求和。如果  $f$  是一个非减的函数，那么，假定以下术语有定义：

$$\int_{a-1}^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x) dx$$

以下是出现在分而治之算法分析中的递推关系的一般形式：

$$T(n) = aT(n/b) + f(n)$$

其中常数  $a \geq 1$  并且  $b > 1$ 。

**命题 B-25：**假设  $T(n)$  由上述定义，那么

- 1 ) 如果对某些常数  $\varepsilon > 0$ ， $f(n)$  是  $O(n^{\log_b a - \varepsilon})$ ，那么  $T(n)$  是  $\Theta(n^{\log_b a})$ 。
- 2 ) 如果对固定的非负常数  $k \geq 0$ ， $f(n)$  是  $\Theta(n^{\log_b a} \log^k n)$ ，那么  $T(n)$  是  $\Theta(n^{\log_b a} \log^{k+1} n)$ 。
- 3 ) 如果对某些常数  $\varepsilon > 0$ ， $f(n)$  是  $\Omega(n^{\log_b a + \varepsilon})$ ，并且如果  $af(n/b) \leq cf(n)$ ，那么  $T(n)$  是  $\Theta(f(n))$ 。

这一命题是渐近地表征分而治之算法递推关系的主方法。

# 参考文献

- [1] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 2nd ed., 1996.
- [2] G. M. Adel'son-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263–266, 1962. English translation in *Soviet Math. Dokl.*, 3, 1259–1262.
- [3] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, 1988.
- [4] A. V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255–300, Amsterdam: Elsevier, 1990.
- [5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
- [6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
- [7] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [8] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, MA: Addison-Wesley, 1999.
- [9] O. Barůvka, "O jistem problemu minimalním," *Práce Moravské Přírodovedecké Společnosti*, vol. 3, pp. 37–58, 1926. (in Czech).
- [10] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.
- [11] R. Bayer and McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173–189, 1972.
- [12] D. M. Beazley, *Python Essential Reference*. Addison-Wesley Professional, 4th ed., 2009.
- [13] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.
- [14] J. L. Bentley, "Programming pearls: Writing correct programs," *Communications of the ACM*, vol. 26, pp. 1040–1045, 1983.
- [15] J. L. Bentley, "Programming pearls: Thanks, heaps," *Communications of the ACM*, vol. 28, pp. 245–250, 1985.
- [16] J. L. Bentley and M. D. McIlroy, "Engineering a sort function," *Software—Practice and Experience*, vol. 23, no. 11, pp. 1249–1265, 1993.
- [17] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.
- [18] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
- [19] G. Brassard, "Crusade for a better notation," *SIGACT News*, vol. 17, no. 1, pp. 60–64, 1985.
- [20] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1991.
- [21] D. Burger, J. R. Goodman, and G. S. Sohi, "Memory systems," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 18, pp. 447–461, CRC Press, 1997.
- [22] J. Campbell, P. Gries, J. Montojo, and G. Wilson, *Practical Programming: An Introduction to Computer Science*. Pragmatic Bookshelf, 2009.
- [23] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.
- [24] S. Carlsson, "Average case results on heapsort," *BIT*, vol. 27, pp. 2–17, 1987.

- [25] V. Cedar, *The Quick Python Book*. Manning Publications, 2nd ed., 2010.
- [26] K. L. Clarkson, “Linear programming in  $O(n3^d)$  time,” *Inform. Process. Lett.*, vol. 22, pp. 21–24, 1986.
- [27] R. Cole, “Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm,” *SIAM J. Comput.*, vol. 23, no. 5, pp. 1075–1091, 1994.
- [28] D. Comer, “The ubiquitous B-tree,” *ACM Comput. Surv.*, vol. 11, pp. 121–137, 1979.
- [29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 3rd ed., 2009.
- [30] M. Crochemore and T. Lecroq, “Pattern matching and text compression algorithms,” in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 8, pp. 162–202, CRC Press, 1997.
- [31] S. Crosby and D. Wallach, “Denial of service via algorithmic complexity attacks,” in *Proc. 12th Usenix Security Symp.*, pp. 29–44, 2003.
- [32] M. Dawson, *Python Programming for the Absolute Beginner*. Course Technology PTR, 3rd ed., 2010.
- [33] S. A. Demurjian, Sr., “Software design,” in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 108, pp. 2323–2351, CRC Press, 1997.
- [34] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing*. Upper Saddle River, NJ: Prentice Hall, 1999.
- [35] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [36] E. W. Dijkstra, “Recursive programming,” *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.
- [37] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan, “Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation,” *Commun. ACM*, vol. 31, pp. 1343–1354, 1988.
- [38] R. W. Floyd, “Algorithm 97: Shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
- [39] R. W. Floyd, “Algorithm 245: Treesort 3,” *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.
- [40] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *J. ACM*, vol. 34, pp. 596–615, 1987.
- [41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [42] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley, 1989.
- [43] M. H. Goldwasser and D. Letscher, *Object-Oriented Programming in Python*. Upper Saddle River, NJ: Prentice Hall, 2008.
- [44] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*. Reading, MA: Addison-Wesley, 1991.
- [45] G. H. Gonnet and J. I. Munro, “Heaps on heaps,” *SIAM J. Comput.*, vol. 15, no. 4, pp. 964–971, 1986.
- [46] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, “External-memory computational geometry,” in *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 714–723, 1993.
- [47] R. L. Graham and P. Hell, “On the history of the minimum spanning tree problem,” *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985.
- [48] L. J. Guibas and R. Sedgewick, “A dichromatic framework for balanced trees,” in *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pp. 8–21, Springer-Verlag, 1978.
- [49] Y. Gurevich, “What does  $O(n)$  mean?,” *SIGACT News*, vol. 17, no. 4, pp. 61–63, 1986.
- [50] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2nd ed., 1996.
- [51] C. A. R. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, pp. 10–15, 1962.
- [52] J. E. Hopcroft and R. E. Tarjan, “Efficient algorithms for graph manipulation,” *Com-*

- munications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.
- [53] B.-C. Huang and M. Langston, “Practical in-place merging,” *Communications of the ACM*, vol. 31, no. 3, pp. 348–352, 1988.
- [54] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, MA: Addison-Wesley, 1992.
- [55] V. Jarník, “O jistem problemu minimalním,” *Praca Moravské Prirodovedecké Společnosti*, vol. 6, pp. 57–63, 1930. (in Czech).
- [56] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.
- [57] D. R. Karger, P. Klein, and R. E. Tarjan, “A randomized linear-time algorithm to find minimum spanning trees,” *Journal of the ACM*, vol. 42, pp. 321–328, 1995.
- [58] R. M. Karp and V. Ramachandran, “Parallel algorithms for shared memory machines,” in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 869–941, Amsterdam: Elsevier/The MIT Press, 1990.
- [59] P. Kirschenhofer and H. Prodinger, “The path length of random skip lists,” *Acta Informatica*, vol. 31, pp. 775–792, 1994.
- [60] J. Kleinberg and É. Tardos, *Algorithm Design*. Reading, MA: Addison-Wesley, 2006.
- [61] A. Klink and J. Wälde, “Efficient denial of service attacks on web application platforms.” 2011.
- [62] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.
- [63] D. E. Knuth, “Big omicron and big omega and big theta,” in *SIGACT News*, vol. 8, pp. 18–24, 1976.
- [64] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 3rd ed., 1997.
- [65] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1998.
- [66] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, “Fast pattern matching in strings,” *SIAM J. Comput.*, vol. 6, no. 1, pp. 323–350, 1977.
- [67] J. B. Kruskal, Jr., “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proc. Amer. Math. Soc.*, vol. 7, pp. 48–50, 1956.
- [68] R. Lesuisse, “Some lessons drawn from the history of the binary search algorithm,” *The Computer Journal*, vol. 26, pp. 154–163, 1983.
- [69] N. G. Leveson and C. S. Turner, “An investigation of the Therac-25 accidents,” *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [70] A. Levitin, “Do we teach the right algorithm design techniques?,” in *30th ACM SIGCSE Symp. on Computer Science Education*, pp. 179–183, 1999.
- [71] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, MA/New York: The MIT Press/McGraw-Hill, 1986.
- [72] M. Lutz, *Programming Python*. O'Reilly Media, 4th ed., 2011.
- [73] E. M. McCreight, “A space-economical suffix tree construction algorithm,” *Journal of Algorithms*, vol. 23, no. 2, pp. 262–272, 1976.
- [74] C. J. H. McDiarmid and B. A. Reed, “Building heaps fast,” *Journal of Algorithms*, vol. 10, no. 3, pp. 352–365, 1989.
- [75] N. Megiddo, “Linear programming in linear time when the dimension is fixed,” *J. ACM*, vol. 31, pp. 114–127, 1984.
- [76] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [77] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, vol. 2 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [78] K. Mehlhorn and A. Tsakalidis, “Data structures,” in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301–341, Amsterdam: Elsevier, 1990.

- [79] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.
- [80] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.
- [81] T. Papadakis, J. I. Munro, and P. V. Poblete, "Average search and update costs in skip lists," *BIT*, vol. 32, pp. 316–332, 1992.
- [82] L. Perkovic, *Introduction to Computing Using Python: An Application Development Focus*. Wiley, 2011.
- [83] D. Phillips, *Python 3: Object Oriented Programming*. Packt Publishing, 2010.
- [84] P. V. Poblete, J. I. Munro, and T. Papadakis, "The binomial transform and its application to the analysis of skip lists," in *Proceedings of the European Symposium on Algorithms (ESA)*, pp. 554–569, 1995.
- [85] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, vol. 36, pp. 1389–1401, 1957.
- [86] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
- [87] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.
- [88] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76–100, 1993.
- [89] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.
- [90] G. A. Stephen, *String Searching Algorithms*. World Scientific Press, 1994.
- [91] M. Summerfield, *Programming in Python 3: A Complete Introduction to the Python Language*. Addison-Wesley Professional, 2nd ed., 2009.
- [92] R. Tamassia and G. Liotta, "Graph drawing," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 52, pp. 1163–1186, CRC Press LLC, 2nd ed., 2004.
- [93] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, pp. 862–874, 1985.
- [94] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.
- [95] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.
- [96] A. B. Tucker, Jr., *The Computer Science and Engineering Handbook*. CRC Press, 1997.
- [97] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1983.
- [98] J. van Leeuwen, "Graph algorithms," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525–632, Amsterdam: Elsevier, 1990.
- [99] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proc. 8th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci., Springer-Verlag, 1991.
- [100] J. S. Vitter and W. C. Chen, *Design and Analysis of Coalesced Hashing*. New York: Oxford University Press, 1987.
- [101] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Algorithms and Complexity* (J. van Leeuwen, ed.), vol. A of *Handbook of Theoretical Computer Science*, pp. 431–524, Amsterdam: Elsevier, 1990.
- [102] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.
- [103] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.
- [104] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, MA: Addison-Wesley, 1993.
- [105] J. Zelle, *Python Programming: An Introduciton to Computer Science*. Franklin, Beedle & Associates Inc., 2nd ed., 2010.

[General Information]

书名=14490879\_数据结构与算法 PYTHON语言实现

页数=477

SS号=14490879