

### 4.4.3 最短路径算法的理论基础

边的放松操作是一项非常容易实现的重要操作，它是实现最短路径算法的基础。同时，它也是理解这个算法的理论基础并使我们能够完整地证明算法的正确性。

#### 4.4.3.1 最优性条件

以下命题证明了判断路径是否为最短路径的全局条件与在放松一条边时所检测的局部条件是等价的。

**命题 P (最短路径的最优性条件)**。令  $G$  为一幅加权有向图，顶点  $s$  是  $G$  中的起点， $\text{distTo}[]$  是一个由顶点索引的数组，保存的是  $G$  中路径的长度。对于从  $s$  可达的所有顶点  $v$ ， $\text{distTo}[v]$  的值是从  $s$  到  $v$  的某条路径的长度，对于从  $s$  不可达的所有顶点  $v$ ，该值为无穷大。当且仅当对于从  $v$  到  $w$  的任意一条边  $e$ ，这些值都满足  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$  时（换句话说，不存在有效边时），它们是最短路径的长度。

**证明。**假设  $\text{distTo}[w]$  是从  $s$  到  $w$  的最短路径。如果对于某条从  $v$  到  $w$  的边  $e$  有  $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ ，那么从  $s$  到  $w$  ( 经过  $v$  ) 且经过  $e$  的路径的长度必然小于  $\text{distTo}[w]$ ，矛盾。因此最优性条件是必要的。

要证明最优性条件是充分的，假设  $w$  是从  $s$  可达的且  $s=v_0 \rightarrow v_1 \rightarrow v_2 \dots \rightarrow v_k=w$  是从  $s$  到  $w$  的最短路径，其权重为  $\text{OPT}_{sw}$ 。对于 1 到  $k$  之间的  $i$ ，令  $e_i$  表示  $v_{i-1}$  到  $v_i$  的边。根据最优性条件，可以得到以下不等式：

$$\begin{aligned}\text{distTo}[w] &= \text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}() \\ \text{distTo}[v_{k-1}] &\leq \text{distTo}[v_{k-2}] + e_{k-1}.\text{weight}() \\ &\dots \\ \text{distTo}[v_2] &\leq \text{distTo}[v_1] + e_2.\text{weight}() \\ \text{distTo}[v_1] &\leq \text{distTo}[s] + e_1.\text{weight}()\end{aligned}$$

综合这些不等式并去掉  $\text{distTo}[s]=0.0$ ，得到：

$$\text{distTo}[w] \leq e_1.\text{weight}() + \dots + e_k.\text{weight}() = \text{OPT}_{sw}.$$

现在， $\text{distTo}[w]$  为从  $s$  到  $w$  的某条边的长度，因此它不可能比最短路径更短。所以以下等式必然成立。

$$\text{OPT}_{sw} \leq \text{distTo}[w] \leq \text{OPT}_{sw}$$

#### 4.4.3.2 验证

命题 P 的一个重要的实际应用是最短路径的验证。无论一种算法会如何计算  $\text{distTo}[]$ ，都只需要遍历图中的所有边一遍并检查最优性条件是否满足就能够知道该数组中的值是否是最短路径的长度。最短路径的算法可能会很复杂，因此能够快速验证计算的结果就变得很重要。为此，我们在本书的网站上的实现中包含了一个 `check()` 方法。该方法还会检查 `edgeTo[]` 指明的路径并验证它与 `distTo[]` 是否一致。

#### 4.4.3.3 通用算法

由最优性条件马上可以得到一个能够涵盖已经学习过的所有最短路径算法的通用算法。现在，我们暂时只研究非负权重的情况。

**命题 Q (通用最短路径算法)**。将  $\text{distTo}[s]$  初始化为 0，其他  $\text{distTo}[]$  元素初始化为无穷大，继续如下操作：

放松  $G$  中的任意边，直到不存在有效边为止。

对于任意从  $s$  可达的顶点  $w$ ，在进行这些操作之后， $\text{distTo}[w]$  的值即为从  $s$  到  $w$  的最短路径的长度（且  $\text{edgeTo}[w]$  的值即为该路径上的最后一条边）。

**证明。** 放松边  $v \rightarrow w$  必然会将  $\text{distTo}[w]$  的值设为从  $s$  到  $w$  的某条路径的长度（且将  $\text{edgeTo}[w]$  设为该路径上的最后一条边）。对于从  $s$  可达的任意顶点  $w$ ，只要  $\text{distTo}[w]$  仍然是无穷大，到达  $w$  的最短路径上的某条边肯定仍然是有效的，因此算法的操作会不断继续，直到由  $s$  可达的每个顶点的  $\text{distTo}[]$  值均变为到达该顶点的某条路径的长度。对于已经找到最短路径的任意顶点  $v$ ，在算法的计算过程中  $\text{distTo}[v]$  的值都是从  $s$  到  $v$  的某条（简单）路径的长度且必然是单调递减的。因此，它递减的次数必然是有限的（每切换一条  $s$  到  $v$  简单路径就递减一次）。当不存在有效边的时候，命题 P 就成立了。

将最优性条件和通用算法放在一起学习的关键原因是，通用算法并没有指定边的放松顺序。因此，要证明这些算法都能通过计算得到最短路径，只需证明它们都会放松所有的边直到所有边都失效即可。

651

#### 4.4.4 Dijkstra 算法

在 4.3 节中，我们讨论了寻找加权无向图中的最小生成树的 Prim 算法：构造最小生成树的每一步都向这棵树中添加一条新的边。Dijkstra 算法采用了类似的方法来计算最短路径树。首先将  $\text{distTo}[s]$  初始化为 0， $\text{distTo}[]$  中的其他元素初始化为正无穷。然后将  $\text{distTo}[]$  最小的非树顶点放松并加入树中，如此这般，直到所有的顶点都在树中或者所有的非树顶点的  $\text{distTo}[]$  值均为无穷大。

**命题 R。** Dijkstra 算法能够解决边权重非负的加权有向图的单起点最短路径问题。

**证明。** 如果  $v$  是从起点可达的，那么所有  $v \rightarrow w$  的边都只会被放松一次。当  $v$  被放松时，必有  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$ 。该不等式在算法结束前都会成立，因此  $\text{distTo}[w]$  只会变小（放松操作只会减小  $\text{distTo}[]$  的值）而  $\text{distTo}[v]$  则不会改变（因为边的权重非负且在每一步中算法都会选择  $\text{distTo}[]$  最小的顶点，之后的放松操作不可能使任何  $\text{distTo}[]$  的值小于  $\text{distTo}[v]$ ）。因此，在所有从  $s$  可达的顶点均被添加到树中之后，最短路径的最优性条件成立，即命题 P 成立。

##### 4.4.4.1 数据结构

要实现 Dijkstra 算法，除了  $\text{distTo}[]$  和  $\text{edgeTo}[]$  数组之外还需要一条索引优先队列  $\text{pq}$ ，以保存需要被放松的顶点并确认下一个被放松的顶点。我们知道 `IndexMinPQ` 可以将索引和键（优先级）关联起来并且可以删除并返回优先级最低的索引。在这里，只要将顶点  $v$  和  $\text{distTo}[v]$  关联起来就立即可以得到 Dijkstra 算法的实现。另外，稍加推导也可以知道， $\text{edgeTo}[]$  中的元素所对应的可达顶点构成了一棵最短路径树。

#### 4.4.4.2 换一个角度看问题

根据算法的证明，我们可以从另一个角度来理解它，如图 4.4.9 所示，已知树结点所对应的 `distTo[]` 值均为最短路径的长度。对于优先队列中的任意顶点  $w$ , `distTo[w]` 是从  $s$  到  $w$  的最短路径的长度，该路径上的所有顶点均在树中且路径上的最后一条边为 `edgeTo[w]`。优先级最小的顶点的 `distTo[]` 值就是最短路径的权重，它不会小于已经被放松过的任意顶点的最短路径的权重，也不会大于还未被放松过的任意顶点的最短路径的权重。这个顶点就是下一个要被放松的顶点。所有从  $s$  可达的顶点都会按照最短路径的权重顺序被放松。

图 4.4.10 是算法处理样图 `tinyEWD.txt` 时的轨迹。在这个例子中，算法构造最短路径树的过程如下所述。

- 将顶点 0 添加到树中，将顶点 2 和 4 加入优先队列。
- 从优先队列中删除顶点 2，将  $0 \rightarrow 2$  添加到树中，将顶点 7 加入优先队列。
- 从优先队列中删除顶点 4，将  $0 \rightarrow 4$  添加到树中，将顶点 5 加入优先队列，边  $4 \rightarrow 7$  失效。
- 从优先队列中删除顶点 7，将  $2 \rightarrow 7$  添加到树中，将顶点 3 加入优先队列，边  $7 \rightarrow 5$  失效。
- 从优先队列中删除顶点 5，将  $4 \rightarrow 5$  添加到树中，将顶点 1 加入优先队列，边  $5 \rightarrow 7$  失效。
- 从优先队列中删除顶点 3，将  $7 \rightarrow 3$  添加到树中，将顶点 6 加入优先队列。
- 从优先队列中删除顶点 1，将  $5 \rightarrow 1$  添加到树中，边  $1 \rightarrow 3$  失效。
- 从优先队列中删除顶点 6，将  $3 \rightarrow 6$  添加到树中。

算法按照顶点到起点的最短路径的长度的增序将它们添加到最短路径树中，如图 4.4.10 右侧的红色箭头所示。

Dijkstra 算法的实现 `DijkstraSP` (算法 4.9) 只是用代码复述了算法的描述，还在 `relax()` 方法中添加了一行语句来处理以下两种情况：要么边的 `to()` 得到的顶点还不在优先队列中，此时需要使用 `insert()` 方法将它加入到优先队列中；要么它已经在优先队列中且优先级需要被降低，此时可以用 `change()` 方法实现。

**命题 R (续)**。在一幅含有  $V$  个顶点和  $E$  条边的加权有向图中，使用 Dijkstra 算法计算根结点为给定起点的最短路径树所需的空间与  $V$  成正比，时间与  $E \log V$  成正比（最坏情况下）。

**证明。** 同 Prim 算法的证明（请见命题 N）

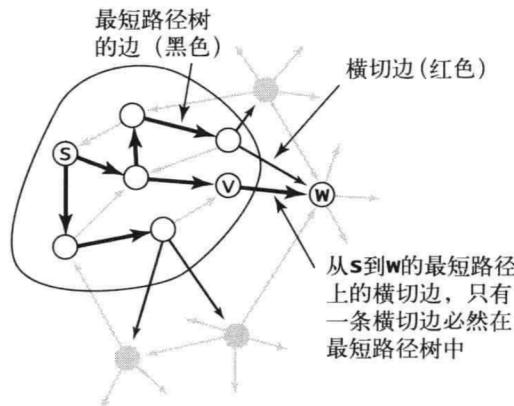


图 4.4.9 Dijkstra 的最短路径算法（另见彩插）

如前所述，思考 Dijkstra 算法的另一种方式就是将它和 4.3 节的 Prim 算法（算法 4.7）相比较。两种算法都会用添加边的方式构造一棵树：Prim 算法每次添加的都是离树最近的非树顶点，Dijkstra 算法每次添加的都是离起点最近的非树顶点。它们都不需要 `marked[]` 数组，因为条件 `!marked[w]` 等价于条件 `distTo[w]` 为无穷大。换句话说，将算法 4.9 中的有向图换成无向图并忽略 `relax()` 方法中 `distTo[v]` 部分的代码，就会得到算法 4.7，也就是 Prim 算法的即时版本（！）。同样，根

据 LazyPrimMST (4.3.4节框注“最小生成树的 Prim 算法的延时实现”) 实现 Dijkstra 算法的延时版本也并不困难。

#### 4.4.4.3 变种

我们只需对 Dijkstra 算法的实现稍作适当的修改就能够解决这个问题的其他版本, 例如, 加权无向图中的单点最短路径。给定一幅加权无向图和一个起点  $s$ , 回答“是否存在一条从  $s$  到给定的顶点  $v$  的路径? 如果有, 找出最短(总权重最小)的那条路径。”等类似问题。

如果将无向图看作有向图, 这个问题的答案就很简单了。也就是说, 对于给定的加权无向图, 创建一幅由相同顶点构成的加权有向图, 且对于无向图中的每条边, 相应地创建两条(方向不同)有向边。有向图中的路径和无向图中的路径存在着一一对应的关系, 路径的权重也是相同的——最短路径的问题是等价的。

#### 算法 4.9 最短路径的 Dijkstra 算法

```
public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty())
            relax(G, pq.delMin());
    }

    private void
    relax(EdgeWeightedDigraph G, int v)
    {
        for(DirectedEdge e : G.adj(v))
        {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight())
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                pq.update(w, distTo[w]);
        }
    }
}
```

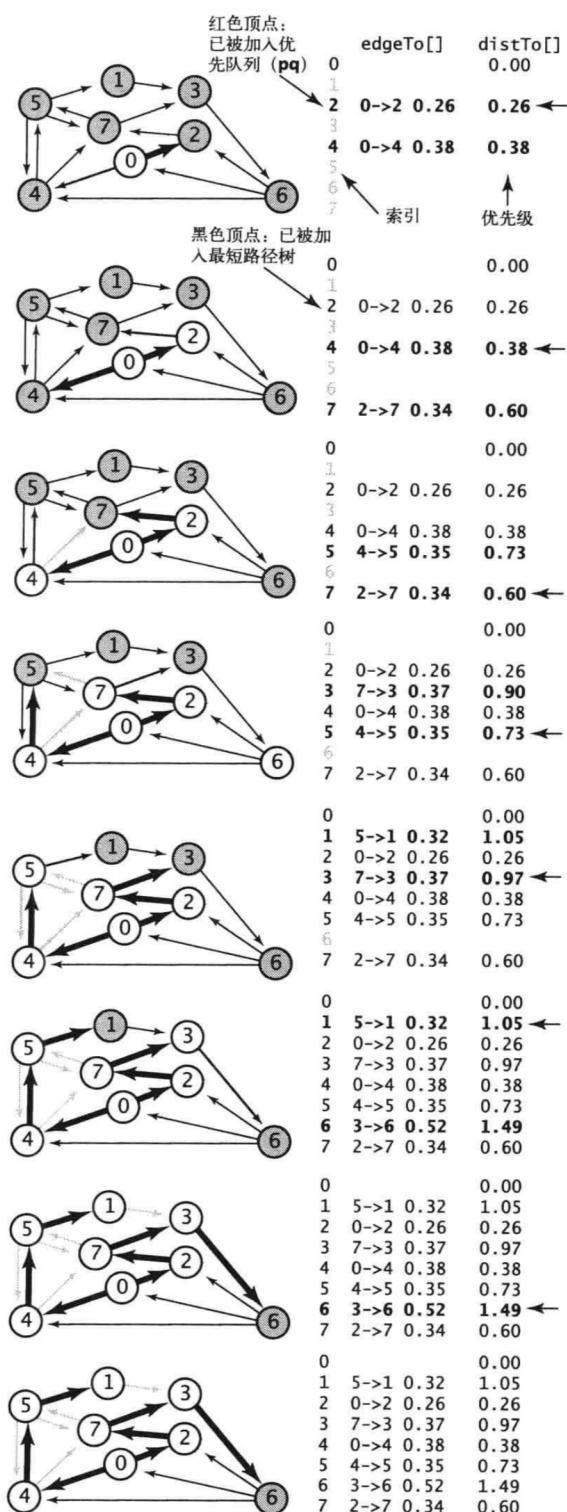


图 4.4.10 Dijkstra 算法的轨迹 (另见彩插)

```

    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.change(w, distTo[w]);
        else                  pq.insert(w, distTo[w]);
    }
}

public double distTo(int v)           // 最短路径树实现中的标准查询算法
public boolean hasPathTo(int v)       // (请见4.4.2.6节框注“最短路径
public Iterable<Edge> pathTo(int v) // API中的查询方法)
}

```

655

Dijkstra 算法的实现每次都会为最短路径树添加一条边，该边由一个树中的顶点指向一个非树顶点 w 且它是到 s 最近的顶点。

**给定两点的最短路径。**给定一幅加权有向图以及一个起点 s 和一个终点 t，找到从 s 到 t 的最短路径。

要解决这个问题，你可以使用 Dijkstra 算法并在从优先队列中取到 t 之后终止搜索。

**任意顶点对之间的最短路径。**给定一幅加权有向图，回答“给定一个起点 s 和一个终点 t，是否存在一条从 s 到 t 的路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

右边框注中短小精悍的代码解决了任意顶点对之间的最短路径问题，所需的时间和空间都与  $EV\log V$  成正比。它构造了 DijkstraSP 对象的数组，每个元素都将相应的顶点作为起点。在用例进行查询时，代码会访问起点所对应的单点最短路径对象并将目的顶点作为参数进行查询。

**欧拉图中的最短路径。**在顶点为平面上的点且边的权重与顶点欧拉间距成正比的图中，解决单点、给定两点和任意顶点对之间的最短路径问题。

在这种情况下，有一个小小的改动可以大幅提高 Dijkstra 算法的运行速度（请见练习 4.4.27）。

图 4.4.11 显示的是 Dijkstra 算法在处理测试文件 mediumEWD.txt（请见 4.4.2.2 节）所定义的欧拉图时用若干不同的起点产生最短路径树的过程。和之前一样，这幅图中的线段都表示双向的有向边。这些图片展示了一段引人入胜的动态过程。

下面，我们将会考虑加权无环图中的最短路径算法并且将在线性时间内解决该问题（比 Dijkstra 算法要快）。然后是负权重的加权有向图中的最短路径问题，Dijkstra 算法不适用于这种情况。

```

public class DijkstraAllPairsSP
{
    private DijkstraSP[] all;

    DijkstraAllPairsSP(EdgeWeightedDigraph G)
    {
        all = new DijkstraSP[G.V()]
        for (int v = 0; v < G.V(); v++)
            all[v] = new DijkstraSP(G, v);
    }

    Iterable<Edge> path(int s, int t)
    { return all[s].pathTo(t); }

    double dist(int s, int t)
    { return all[s].distTo(t); }
}

```

任意顶点对之间的最短路径

656

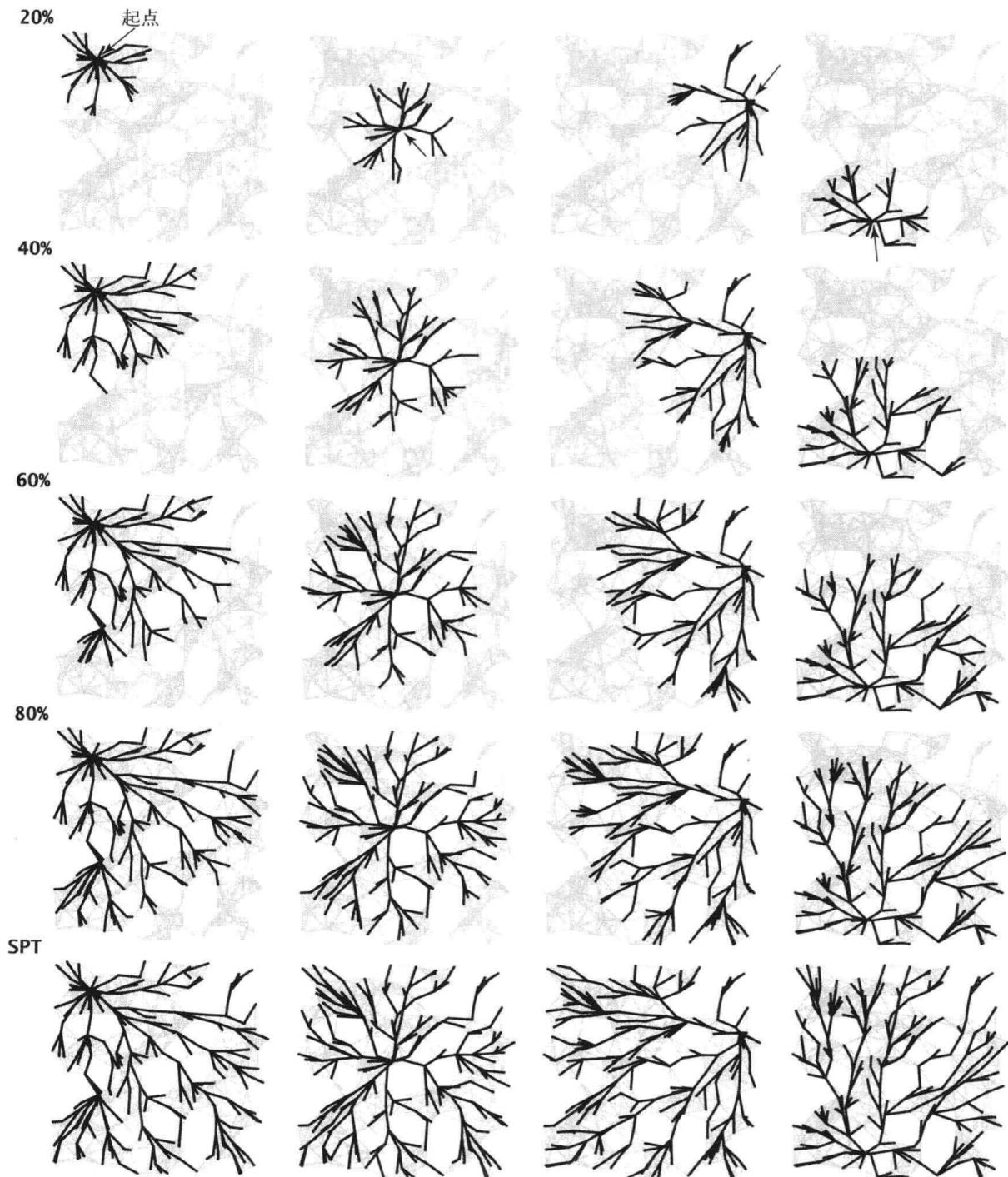


图 4.4.11 Dijkstra 算法 (250 个顶点, 不同的起点)

657

#### 4.4.5 无环加权有向图中的最短路径算法

许多应用中的加权有向图都是不含有向环的。我们现在来学习一种比 Dijkstra 算法更快、更简单的在无环加权有向图中找出最短路径的算法，如图 4.4.12 所示。它的特点是：

- 能够在线性时间内解决单点最短路径问题；
- 能够处理负权重的边；
- 能够解决相关的问题，例如找出最长的路径。

这些算法都是在4.2节中学过的无环有向图的拓扑排序算法的简单扩展。

特别的是，只要将顶点的放松和拓扑排序结合起来，马上就能够得到一种解决无环加权有向图中的最短路径问题的算法。首先，将 `distTo[s]` 初始化为0，其他 `distTo[]` 元素初始化为无穷大，然后一个一个地按照拓扑顺序放松所有顶点。我们可以用与 Dijkstra 算法的证明（命题 R）类似的方法证明这个方法的正确性。

**命题 S。**按照拓扑顺序放松顶点，就能在和  $E+V$  成正比的时间内解决无环加权有向图的单点最短路径问题。

**证明。**每条边  $v \rightarrow w$  都只会被放松一次。当  $v$  被放松时，得到： $\text{distTo}[w] \leq \text{distTo}[v] + e.$  weight()。在算法结束前该不等式都成立，因为 `distTo[v]` 是不会变化的（因为是按照拓扑顺序放松顶点，在  $v$  被放松之后算法不会再处理任何指向  $v$  的边）而 `distTo[w]` 只会变小（任何放松操作都只会减小 `distTo[]` 中的元素的值）。因此，在所有从  $s$  可达的顶点都被加入到树中后，最短路径的最优性条件成立，命题 Q 也就成立了。时间上限很容易得到：命题 G 告诉我们拓扑排序所需的时间与  $E+V$  成正比，而在第二次遍历中每条边都只会被放松一次，因此算法总耗时与  $E+V$  成正比。

[658]

图 4.4.13 是算法处理无环加权有向样图 `tinyEWDAG.txt` 的轨迹。在这个例子中，算法由顶点 5 开始按照以下步骤构建了一棵最短路径树：

- 用深度优先搜索得到图的顶点的拓扑排序  $5 \ 1 \ 3 \ 6 \ 4 \ 7 \ 0 \ 2$ ；
- 将顶点 5 和从它指出的所有边添加到树中；
- 将顶点 1 和边  $1 \rightarrow 3$  添加到树中；
- 将顶点 3 和边  $3 \rightarrow 6$  添加到树中，边  $3 \rightarrow 7$  已经失效；
- 将顶点 6 和边  $6 \rightarrow 2$ 、 $6 \rightarrow 0$  添加到树中，边  $6 \rightarrow 4$  已经失效；
- 将顶点 4 和边  $4 \rightarrow 0$  添加到树中，边  $4 \rightarrow 7$  和  $6 \rightarrow 0$  已经失效；
- 将顶点 7 和边  $7 \rightarrow 2$  添加到树中，边  $6 \rightarrow 2$  已经失效；
- 将顶点 0 添加到树中，边  $0 \rightarrow 2$  已经失效；
- 将顶点 2 添加到树中。

图中没有画出将 2 添加到树中的一步，拓扑序列中的最后一个顶点没有指出的边。

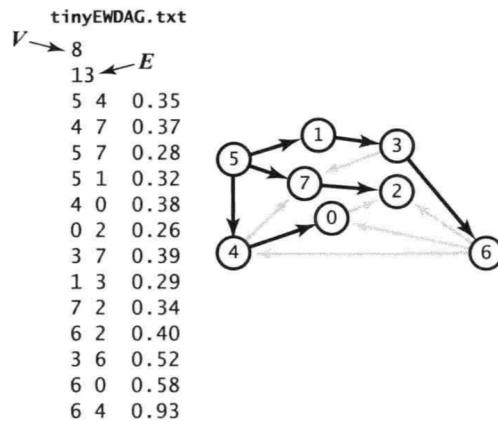


图 4.4.12 一幅无环加权有向图和它的一棵最短路径树

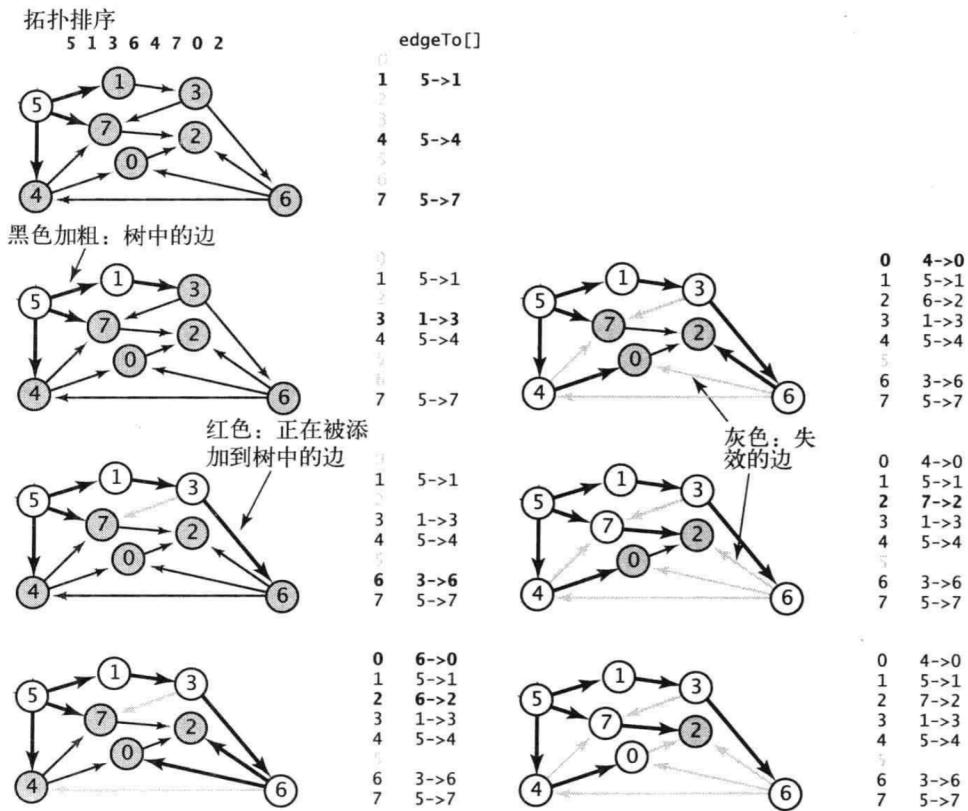


图 4.4.13 寻找无环加权有向图中的最短路径的算法轨迹（另见彩插）

算法 4.10 在实现中直接使用了已学习过的许多代码。它假设 `Topological` 类使用本节中介绍的 `EdgeWeightedDigraph` 类和 `DirectedEdge` 类的 API (请见练习 4.4.12) 重载了拓扑排序的方法。注意，该实现中不需要布尔数组 `marked[]`：因为是按照拓扑顺序处理无环有向图中的顶点，所以不可能再次遇到已经被放松过的顶点。算法 4.10 的效率几乎已经没有提高的空间了：在拓扑排序后，构造函数会扫描整幅图并将每条边放松一次。在已知加权图是无环的情况下，它是找出最短路径的最好方法。

659

#### 算法 4.10 无环加权有向图的最短路径算法

```
public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        Topological top = new Topological(G);
    }
}
```

```

        for (int v : top.order())
            relax(G, v);
    }

    private void relax(EdgeWeightedDigraph G, int v)
        // 请见4.4.1.5框注“顶点的松弛”

    public double distTo(int v)           // 最短路径树实现中的标准查询算法（请见4.4.1.6
                                         // 框注“最短路径API的查询方法”）

    public boolean hasPathTo(int v)
    public Iterable<Directed Edge> pathTo(int v)
}

```

无环加权有向图的最短路径算法使用了拓扑排序（算法 4.5，重载了 `EdgeWeightedDigraph` 类和 `DirectedEdge` 类）来按照拓扑顺序放松所有顶点，这对于计算出图中的最短路径已经足够了。

```

% java AcyclicSP tinyEWDAG.txt 5
5 to 0 (0.73): 5->4 0.35  4->0 0.38
5 to 1 (0.32): 5->1 0.32
5 to 2 (0.62): 5->7 0.28  7->2 0.34
5 to 3 (0.62): 5->1 0.32  1->3 0.29
5 to 4 (0.35): 5->4 0.35
5 to 5 (0.00):
5 to 6 (1.13): 5->1 0.32  1->3 0.29  3->6 0.52
5 to 7 (0.28): 5->7 0.28

```

660

命题 S 很重要，因为它的“无环”能够极大地简化问题的论断。对于最短路径问题，基于拓扑排序的方法比 Dijkstra 算法快的倍数与 Dijkstra 算法中所有优先队列操作的总成本成正比。另外，命题 S 的证明和边的权重是否非负无关，因此无环加权有向图不会受到任何限制。下面用这个特点解决边的负权重问题。我们会考虑使用这个最短路径模型来解决另外两个问题，其中之一乍一看甚至和图的处理似乎没有任何关系。

#### 4.4.5.1 最长路径

考虑在无环加权有向图中寻找最长路径的问题，边的权重可正可负。

**无环加权有向图中的单点最长路径。**给定一幅无环加权有向图（边的权重可能为负）和一个起点  $s$ ，回答“是否存在一条从  $s$  到给定的顶点  $v$  的路径？如果有，找出最长（总权重最大）的那条路径。”等类似问题。

我们刚刚学习过的算法能够快速地解决这个问题。

**命题 T。**解决无环加权有向图中的最长路径问题所需的时间与  $E+V$  成正比。

**证明。**给定一个最长路径问题，复制原始无环加权有向图得到一个副本并将副本中的所有边的权重变为负值。这样，副本中的最短路径即为原图中的最长路径。要将最短路径问题的答案转换为最长路径问题的答案，只需将方案中的权重变为正值即可。根据命题 S 立即可以得到算法所需的时间。

根据这种转换实现 `AcyclicLP` 类来寻找一幅无环加权有向图中的最长路径就十分简单了。实现该类的一个更简单的方法是修改 `AcyclicSP`, 将 `distTo[]` 的初始值变为 `Double.NEGATIVE_INFINITY` 并改变 `relax()` 方法中的不等式的方向。无论使用哪种方法, 都能得到无环加权有向图中的最长路径问题的一种高效的解决方案。和它形成鲜明对比的是, 在一般的加权有向图(边的权重可能为负)中寻找最长简单路径的已知最好算法在最坏情况下所需的时间是指数级别的(请见第 6 章)! 出现环的可能性似乎使这个问题的难度以指教级别增长。

图 4.4.14 是算法在无环加权有向样图 `tinyEWDAG.txt` 中寻找最长路径的轨迹, 你可以将它与图 4.4.13 相比较。在这个例子中, 算法由顶点 5 按照以下步骤构建了一棵最长路径树:

- 用深度优先搜索得到图的顶点的拓扑排序 `5 1 3 6 4 7 0 2;`
- 将顶点 5 和从它指出的所有边添加到树中;
- 将顶点 1 和边  $1 \rightarrow 3$  添加到树中;
- 将顶点 3 和边  $3 \rightarrow 6$ 、 $3 \rightarrow 7$  添加到树中, 边  $5 \rightarrow 7$  已经失效;
- 将顶点 6 和边  $6 \rightarrow 2$ 、 $6 \rightarrow 4$  和  $6 \rightarrow 0$  添加到树中;
- 将顶点 4 和边  $4 \rightarrow 0$ 、 $4 \rightarrow 7$  添加到树中, 边  $6 \rightarrow 0$  和  $3 \rightarrow 7$  已经失效;
- 将顶点 7 和边  $7 \rightarrow 2$  添加到树中, 边  $6 \rightarrow 2$  已经失效;
- 将顶点 0 添加到树中, 边  $0 \rightarrow 2$  已经失效;
- 将顶点 2 添加到树中(未画出)。

最长路径算法处理顶点的顺序和最短路径算法一样, 但产生的结果却完全不同。

#### 4.4.5.2 平行任务调度

作为算法应用的示例, 我们再次考虑在 4.2 节中出现过的任务调度类的问题。这次需要解决以下调度问题(楷体部分为与 4.2.4.1 节的问题描述的不同之处)。

**优先级限制下的并行任务调度。**给定一组需要完成的特定任务, 以及一组关于任务完成的先后次序的优先级限制。在满足限制条件的前提下应该如何在若干相同的处理器上(数量不限)安排任务并在最短的时间内完成所有任务?

4.2 节的模型默认只有单个处理器: 将任务按照拓扑顺序排序, 完成任务的总耗时就是所有任务所需要的总时间。现在假设有足够的处理器并能够同时处理任意多的任务, 受到的只有优先级

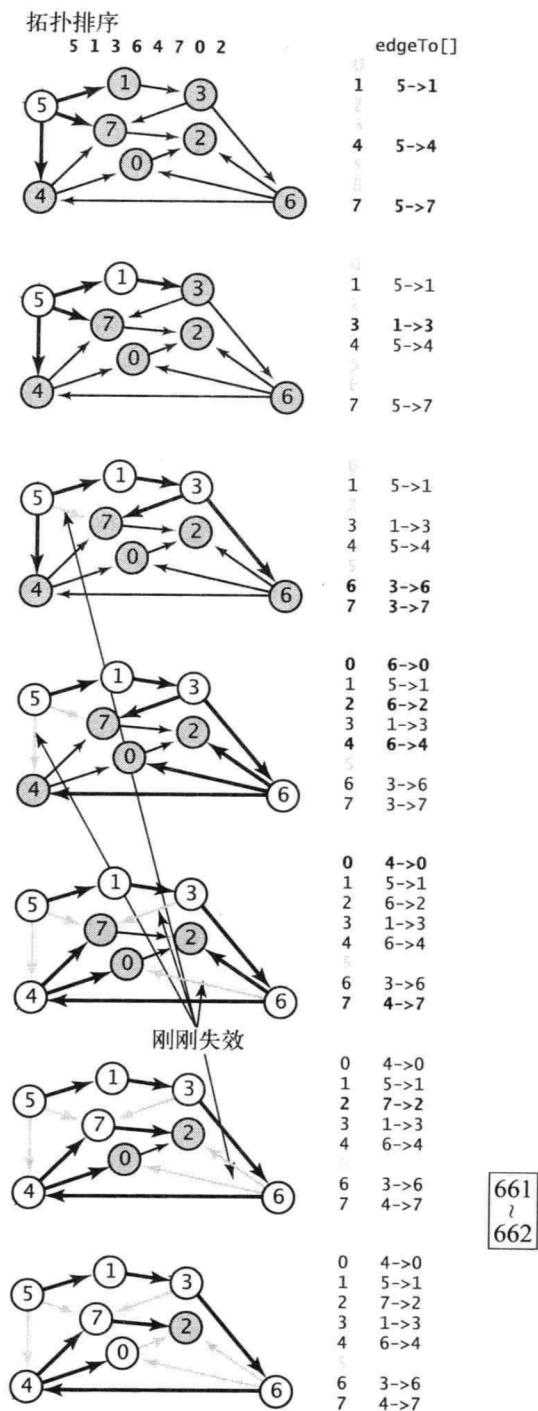


图 4.4.14 无环图中的最长路径算法  
(另见彩插)

的限制。和以前一样，需要处理的任务可能上百万甚至上亿，因此需要一个高效的算法。令人兴奋的是，正好存在一种线性时间的算法——一种叫做“关键路径”的方法能够证明这个问题与无环加权有向图中的最长路径问题是等价的。这个方法已成功应用于无数的工业软件之中。

假设任意可用的处理器都能在任务所需的时间内完成它，那么我们的重点就是尽早安排每一个任务。例如，表 4.4.5 给出了一个任务调度问题，图 4.4.15 给出的解决方案显示了这个问题所需的最短时间为 173.0。这份调度方案满足了所有限制条件，没有其他调度方案能比它耗时更少，因为任务必须按照  $0 \rightarrow 9 \rightarrow 6 \rightarrow 8 \rightarrow 2$  的顺序完成。这个顺序就是这个问题的关键路径。由优先级限制指定的每一列任务都代表了调度方案的一种可能的时间下限。如果将一系列任务的长度定义为完成所有任务的最早可能时间，那么最长的任务序列就是问题的关键路径，因为在这份任务序列中任何任务的启动延迟都会影响到整个项目的完成时间。

**定义。**解决并行任务调度问题的关键路径方法的步骤如下：创建一幅无环加权有向图，其中包括一个起点  $s$  和一个终点  $t$  且每个任务都对应着两个顶点（一个起始顶点和一个结束顶点）。对于每个任务都有一条从它的起始顶点指向结束顶点的边，边的权重为任务所需的时间。对于每条优先级限制  $v \rightarrow w$ ，添加一条从  $v$  的结束顶点指向  $w$  的起始顶点的权重为零的边。我们还需要为每个任务添加一条从起点指向该任务的起始顶点的权重为零的边以及一条从该任务的结束顶点到终点的权重为零的边。这样，每个任务预计的开始时间即为从起点到它的起始顶点的最长距离。

表 4.4.5 一个任务调度问题

任务	时耗	必须在以下任务之前完成			
0	41.0	1	7	9	2
1	51.0		2		
2	50.0				
3	36.0				
4	38.0				
5	45.0				
6	21.0	3	8		
7	32.0	3	8		
8	32.0		2		
9	29.0	4	6		

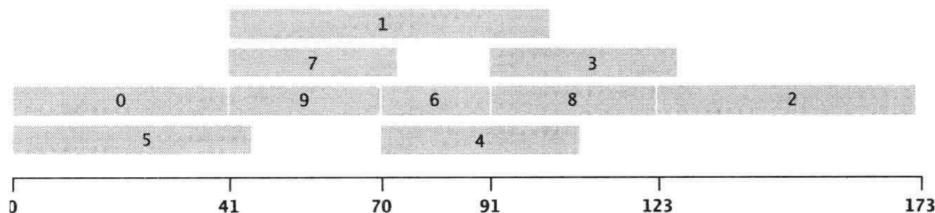


图 4.4.15 并行任务调度问题的解决方案

图 4.4.16 显示的是示例任务所对应的图，图 4.4.17 则显示的是最长路径的答案。如定义所述，在图中每个任务都对应着三条边（从起点到起始顶点、从结束顶点到终点的权重为零的边，以及一条从起始顶点到结束顶点的边），每个优先级限制条件都对应着一条边。后面框注“优先级限制下的并行任务调度问题的关键路径方法”中的 CPM 类简洁明了地实现了关键路径方法。它能够将任意任务调度问题转化为无环加权有向图中的一个最长路径问题，用 AcyclicLP 解决它并打印出每个任务的开始时间以及调度方案的结束时间。

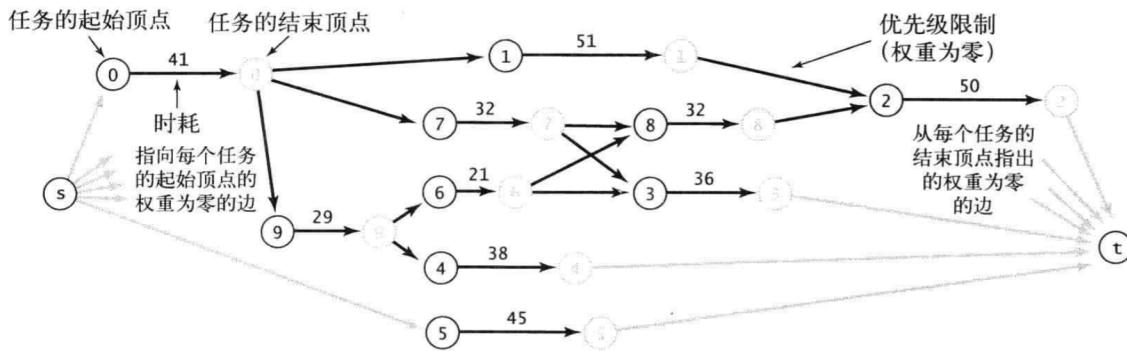


图 4.4.16 任务调度问题的无环加权有向图表示

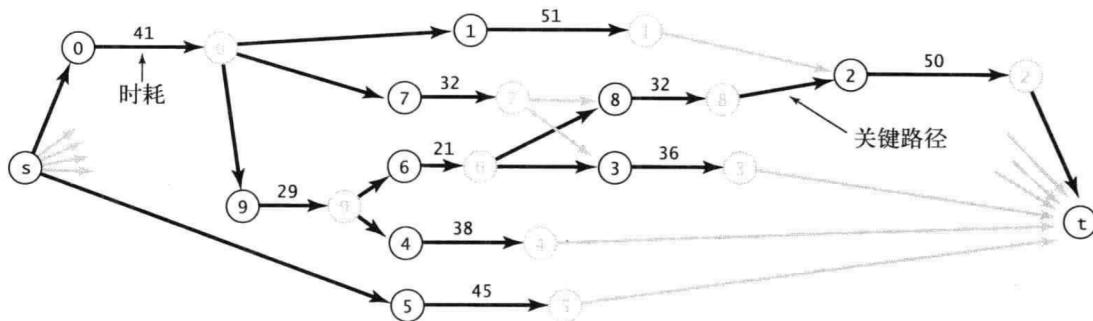


图 4.4.17 任务调度示例问题的最长路径解决方案

663  
664

### 优先级限制下的并行任务调度问题的关键路径方法

```
public class CPM
{
    public static void main(String[] args)
    {
        int N = StdIn.readInt(); StdIn.readLine();
        EdgeWeightedDigraph G;
        G = new EdgeWeightedDigraph(2*N+2);
        int s = 2*N, t = 2*N+1;
        for (int i = 0; i < N; i++)
        {
            String[] a = StdIn.readLine().split("\s+");
            double duration = Double.parseDouble(a[0]);
            G.addEdge(new DirectedEdge(i, i+N, duration));
            G.addEdge(new DirectedEdge(s, i, 0.0));

            G.addEdge(new DirectedEdge(i+N, t, 0.0));
            for (int j = 1; j < a.length; j++)
            {
                int successor = Integer.parseInt(a[j]);
                G.addEdge(new DirectedEdge(i+N, successor, 0.0));
            }
        }
    }
}
```

```
% more jobsPC.txt
10
41.0 1 7 9
51.0 2
50.0
36.0
38.0
45.0
21.0 3 8
32.0 3 8
32.0 2
29.0 4 6
```

```

AcyclicLP lp = new AcyclicLP(G, s);
StdOut.println("Start times:");
for (int i = 0; i < N; i++)
    StdOut.printf("%4d: %5.1f\n", i, lp.distTo(i));
StdOut.printf("Finish time: %5.1f\n", lp.distTo(t));
}
}

```

```

% java CPM <
jobsPC.txt
Start times:
0: 0.0
1: 41.0
2: 123.0
3: 91.0
4: 70.0
5: 0.0
6: 70.0
7: 41.0
8: 91.0
9: 41.0
Finish time:
173.0

```

这里实现的任务调度问题的关键路径方法将问题归约为寻找无环加权有向图的最长路径问题。它会根据任务调度问题的描述用关键路径的方法构造一幅加权有向图（且必然是无环的），然后使用 `AcyclicLP`（请见命题 T）找到图中的最长路径树，最后打印出各条最长路径的长度，也就正好是每个任务的开始时间。

665

**命题 U。**解决优先级限制下的并行任务调度问题的关键路径法所需的时间为线性级别。

**证明。**为什么 `CPM` 类能够解决问题？算法的正确性依赖于两个因素。首先，在相应的有向无环图中，每条路径都是由任务的起始顶点和结束顶点组成的并由权重为零的优先级限制条件的边分隔——从起点  $s$  到任意顶点  $v$  的任意路径的长度都是任务  $v$  的开始 / 结束时间的下限，因为这已经在同一台处理器上顺序完成这些任务的最优的排列顺序了。因此，从起点  $s$  到终点  $t$  的最长路径就是所有任务的完成时间的下限。第二，由最长路径得到的所有开始和结束时间都是可行的——每个任务都只能在优先级限制指定的先导任务完成之后开始，因为它的开始时间就是顶点到它的起始顶点的最长路径的长度。因此，从起点  $s$  到终点  $t$  的最长路径长度就是所有任务完成时间的上限。由命题 T 很容易得到算法所需的时间是线性的。

#### 4.4.5.3 相对最后期限限制下的并行任务调度

一般的最后期限 (deadline) 都是相对于第一个任务的开始时间而言的。假设在任务调度问题中加入一种新类型的限制，需要某个任务必须在指定的时间点之前开始，即指定和另一个任务的开始时间的相对时间。这种类型的限制条件在争分夺秒的生产线上以及许多其他应用中都很常见，但它也会使得任务调度问题更难解决。例如，如表 4.4.6 所示，假设要在前面的示例中加入一个限制条件，使 2 号任务必须在 4 号任务启动后的 12 个时间单位之内开始。实际上，在这里最后期限限制的是 4 号任务的开始时间：它的开始时间不能早于 2 号任务开始 12 个时间单位。在示例中，调度表中有足够的空档来满足这个最后期限限制：我们可以令 4 号任务开始于 111 时间，即 2 号任务计划开始时间前的 12 个时间单位处。需要注意的是，如果 4 号任务耗时很长，这个修改可能会延长整个调度计划的完成时间。同理，如果再添加一个最后期限的限制条件，令 2 号任务必须在 7 号任务启动后的 70 个时间单位内开始，还可以将 7 号任务的开始时间调整到 53，这样就不用修改 3 号任务和 8 号任务的计划开始时间。但是如果继续限制 4 号任务必须在零号任务启动后的 80 个时间单位内开始，那么就不存在可行的调度计划了：限制条件 4 号任务必须在 0 号任务启动后的 80 个时间单位内开始以及 2 号任务必须在 4 号任务启动后的 12 个时间单位之内开始，意味着 2 号任务必须在 0 号任务启动后的 93 个时间单位之内开始，但因为存在任务链 0 (41 个时间单位) → 9 (29 个时间单位) → 6 (21 个时间单位) → 8 (32 个时间单位) → 2，2 号任务最早也只能在 0 号任务

启动后的 123 个时间单位之内开始如表 4.4.7 所示。最后期限的限制越多，调度的可能性也就越多，简单的问题也会变得越困难。

表 4.4.7 向任务调度问题中添加的最后期限限制

任务	相对最后期限	相对于任务
2	12.0	4
2	70.0	7
4	80.0	0

**命题 V。** 相对最后期限限制下的并行任务调度问题是一个加权有向图中的最短路径问题（可能存在环和负权重边）。

**证明。** 与命题 U 一样根据任务调度的描述构造相同的加权有向图，为每条最后期限限制添加一条边：如果任务 v 必须在任务 w 启动后的 d 个时间单位内开始，则添加一条从 v 指向 w 的负权重为 d 的边。将所有边的权重取反即可将该问题转化为一个最短路径问题。如果存在可行的调度方案，证明也就完成了。你将会看到，判断一个调度方案是否可行也是计算的一部分。

这个示例说明了负权重的边在实际应用的模型中也能起到重要的作用。它说明，如果能够有效解决负权重边的最短路径问题，那就能够找到相对最后期限限制下的并行任务调度问题的解决方案。我们已经学习过的算法都无法完成这个任务：Dijkstra 算法只适用于正（或零）权重的边，算法 4.10 要求有向图是无环的。下面我们来看看如何解决含有负权重且不一定是无环的有向图中的最短路径问题（请见图 4.4.18）。

#### 4.4.6 一般加权有向图中的最短路径问题

刚才讨论过的最后期限限制下的任务调度问题告诉我们负权重的边并不仅仅是一个数学问题。相反，它能够极大地扩展解决最短路径问题的模型的应用范围。接下来，考虑既可能含有环也可能含有负权重的边的加权有向图中的最短路径算法。

在开始之前，先来学习一下这种有向图的基本性质以更新我们对最短路径的认识。图 4.4.19 是一个小小的示例，展示的是负权重的边对有向图中的最短路径的影响。也许最明显的改变就是当存在负权重的边时，权重较小的路径含有的边可能会比权重较大的路径更多。在只存在正权重的边时，我们的重点在于寻找近路；但当存在负权重的边时，我们可能会为了经过负权重的边而绕弯。这种效应使得我们要将查找“最短”路径的感觉转变为对算法本质的理解。因此需要抛弃直觉并在一个简单、抽象的层面上考虑这个问题。

表 4.4.6 相对最后期限限制下的任务调度

原始问题	
任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	70.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2号任务必须在4  
号任务启动后的12  
个时间单位之内开始

任务调度	
任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2号任务必须在7  
号任务启动后的70  
个时间单位之内开始

调度方案不存在	
任务	开始时间
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	53.0
8	91.0
9	41.0

4号任务必须在0  
号任务启动后的80  
个时间单位之内开始  
调度方案不存在

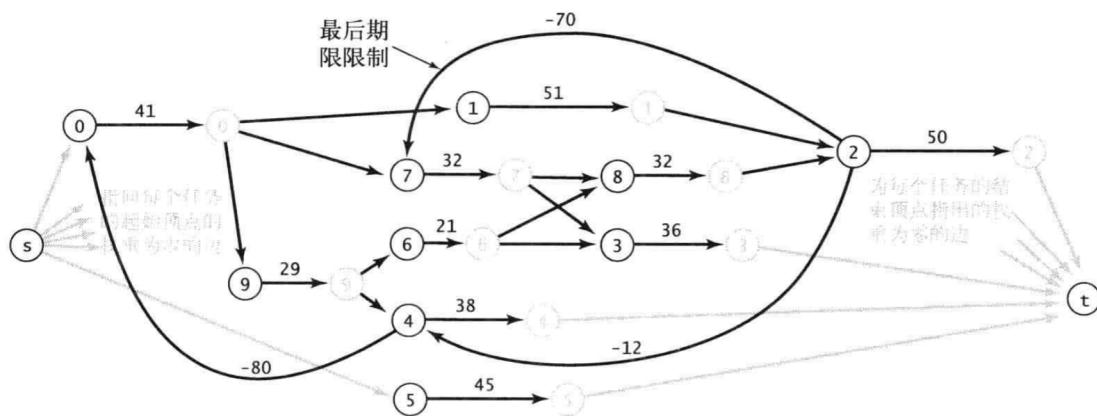


图 4.4.18 相对最后期限限制和优先级限制下的并行任务调度问题的加权有向图表示

#### 4.4.6.1 尝试 I

第一个想法是先找到权重最小（最小的负值）的边，然后将所有边的权重加上这个负值的绝对值，这样原有向图就转变称为了一幅不含有负权重边的有向图。这种天真的做法不会解决任何问题，因为新图中的最短路径和原图中的最短路径毫无关系。路径中的边越多，这种变换产生的危害越大（请见练习 4.4.14）。

#### 4.4.6.2 尝试 II

第二个想法是尝试改造 Dijkstra 算法。这种方法最根本的缺陷在于原算法的基础在于根据距离起点的远近依次检查路径。命题 R 对算法正确性的证明是基于添加一条边会使的路径变得更长的假设。但添加任意负权重的边只会使得路径更短，因此这个假设是不成立的（请见练习 4.4.14）。

#### 4.4.6.3 负权重的环

当我们在研究含有负权重边的有向图时，如果该图中含有一个权重为负的环，那么最短路径的概念就失去意义了。例如图 4.4.20，除了边  $5 \rightarrow 4$  的权重为  $-0.66$  外，它和第一个示例完全相同。这里，环  $4 \rightarrow 7 \rightarrow 5 \rightarrow 4$  的权重为：

$$0.37 + 0.28 - 0.66 = -0.01$$

我们只要围着这个环兜圈子就能得到权重任意短的路径！注意，有向环的所有边的权重并不一定都必须是负的，只要权重之和是负的即可。

**定义。** 加权有向图中的负权重环是一个总权重（环上的所有边的权重之和）为负的有向环。

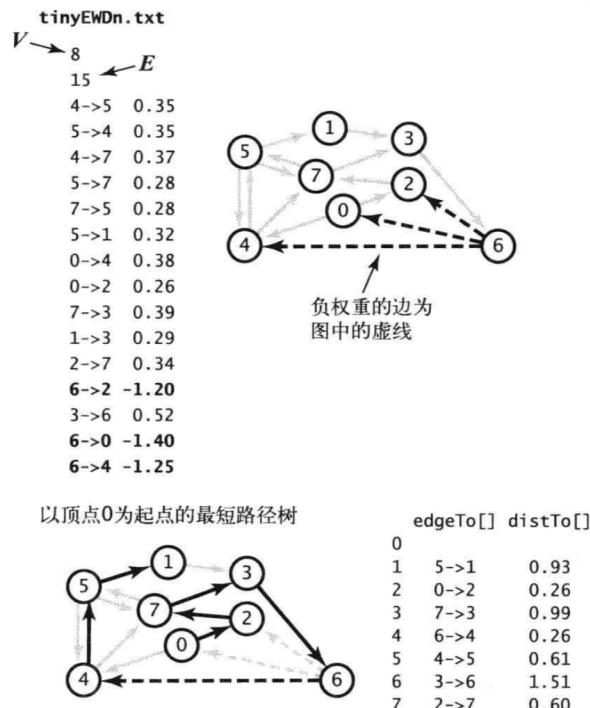


图 4.4.19 含有负权重的边的加权有向图

现在，假设从  $s$  到可达的某个顶点  $v$  的路径上的某个顶点在一个负权重环上。在这种情况下，从  $s$  到  $v$  的最短路径是不可能存在的，因为可以用这个负权重环构造权重任意小的路径。换句话说，在负权重环存在的情况下，最短路径问题是没有意义的，如图 4.4.21 所示。

**命题 W。** 当且仅当加权有向图中至少存在一条从  $s$  到  $v$  的有向路径且所有从  $s$  到  $v$  的有向路径上的任意顶点都不存在于任何负权重环中时， $s$  到  $v$  的最短路径才是存在的。

**证明。** 请见以上讨论以及练习 4.4.29。

注意，要求最短路径上的任意顶点都不存在负权重环意味着最短路径是简单的，而且与正权重边的图一样都能够得到此类顶点的最短路径树。

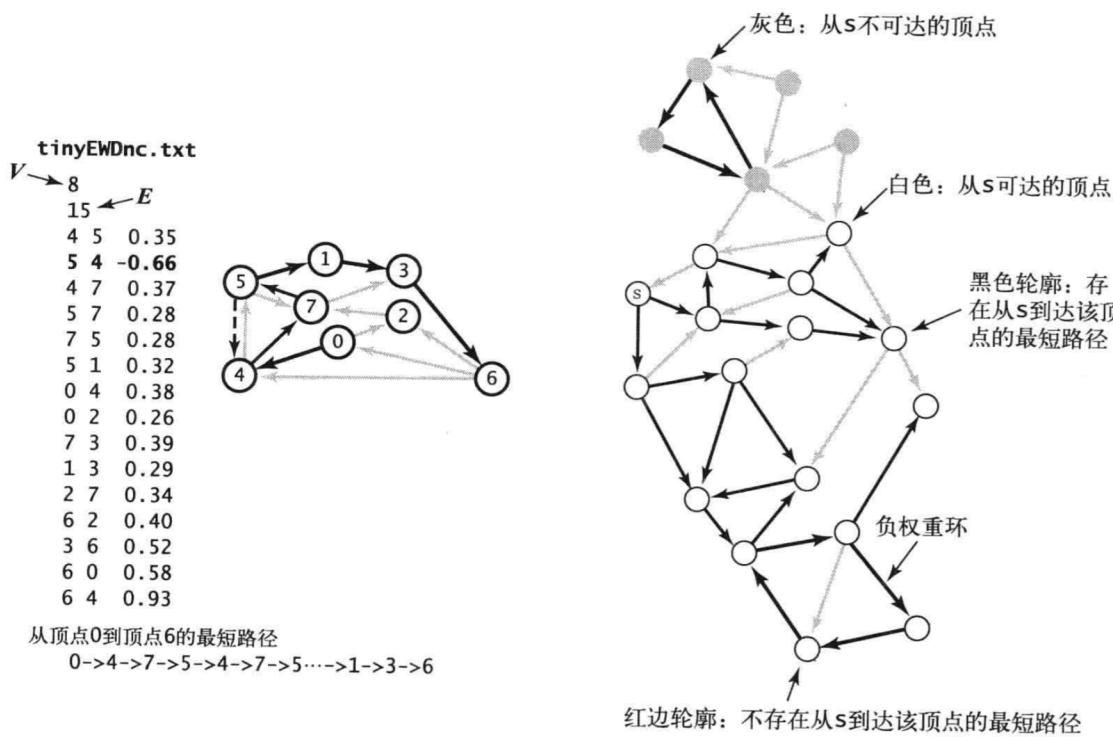


图 4.4.20 含有负权重环的加权有向图（另见彩插） 图 4.4.21 最短路径问题的各种可能性（另见彩插）

#### 4.4.6.4 尝试III

无论是否存在负权重环，从  $s$  到可达的其他顶点的一条最短的简单路径都是存在的。为什么不定义最短路径以方便寻找呢？不幸的是，已知解决这个问题的最好算法在最坏情况下所需的时间是指数级别的（请见第 6 章）。一般来说，我们认为这种问题“太难了”，只会研究它的简单版本。

因此，一个定义明确且可以解决加权有向图最短路径问题的算法要能够：

- 对于从起点不可达的顶点，最短路径为正无穷 ( $+\infty$ )；
- 对于从起点可达但路径上的某个顶点属于一个负权重环的顶点，最短路径为负无穷 ( $-\infty$ )；
- 对于其他所有顶点，计算最短路径的权重（以及最短路径树）。

从本节的开始，我们会不断为最短路径问题加上各种限制并找到解决相应问题的办法。首先，

我们不允许负权重边的存在；其次不接受有向环。现在我们放宽所有这些条件并重点解决一般有向图中的以下问题。

负权重环的检测。给定的加权有向图中含有负权重环吗？如果有，找到它。

负权重环不可达时的单点最短路径。给定一幅加权有向图和一个起点  $s$  且从  $s$  无法到达任何负权重环，回答“是否存在一条从  $s$  到给定的顶点  $v$  的有向路径？如果有，找出最短（总权重最小）的那条路径。”等类似问题。

总结。尽管在含有环的有向图中最短路径是一个没有意义的问题，而且也无法有效解决在这种有向图中高效找出最短简单路径的问题，在实际应用中仍然需要能够识别其中的负权重环。例如，在最后期限限制下的任务调度问题中，负权重环的出现可能相对较少：限制条件和最后期限都是从现实世界中的实际限制得来的，因此负权重环大多可能来自于问题陈述中的错误。找出负权重环，改正相应的错误，找到没有负权重环问题的调度方案才是解决问题的正确方式。在其他情况下，找到负权重环就是计算的目标。下面这个由 R.Bellman 和 L.Ford 在 20 世纪 50 年代末期发明的算法能够简明、有效地解决这些问题并且同样适用于正权重边的有向图。

670

**命题 X (Bellman-Ford 算法)**。在任意含有  $V$  个顶点的加权有向图中给定起点  $s$ ，从  $s$  无法到达任何负权重环，以下算法能够解决其中的单点最短路径问题：将  $\text{distTo}[s]$  初始化为 0，其他  $\text{distTo}[]$  元素初始化为无穷大。以任意顺序放松有向图的所有边，重复  $V$  轮。

**证明。**对于从  $s$  可达的任意顶点  $t$ ，考虑从  $s$  到  $t$  的一条最短路径： $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ ，其中  $v_0$  等于  $s$ ， $v_k$  等于  $t$ 。因为负权重环是不可达的，这样的路径是存在的且  $k$  不会大于  $V-1$ 。我们会通过归纳法证明算法在第  $i$  轮之后能够得到  $s$  到  $v_i$  的最短路径。最简单的情况 ( $i=0$ ) 很容易。假设对于  $i$  命题成立，那么  $s$  到  $v_i$  的最短路径即为  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i$ ， $\text{distTo}[v_i]$  就是这条路径的长度。现在，我们在第  $i$  轮中放松所有的顶点，包括  $v_i$ ，因此  $\text{distTo}[v_{i+1}]$  不会大于  $\text{distTo}[v_i]$  与边  $v_i \rightarrow v_{i+1}$  的权重之和。在第  $i$  轮放松之后， $\text{distTo}[v_{i+1}]$  必然等于  $\text{distTo}[v_i]$  与边  $v_i \rightarrow v_{i+1}$  的权重之和。它不可能更大，因为在第  $i$  轮中放松了所有顶点，包括  $v_i$ ；它也不可能更小，因为它就是路径  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_{i+1}$  的长度，也就是最短路径了。因此，在  $i+1$  轮之后算法能够得到从  $s$  到  $v_{i+1}$  的最短路径。

**命题 W (续)**。Bellman-Ford 算法所需的时间和  $EV$  成正比，空间和  $V$  成正比。

**证明。**在每一轮中算法都会放松  $E$  条边，共重复  $V$  轮。

这个方法非常通用，因为它没有指定边的放松顺序。下面将注意力集中在一个通用性稍逊的方法上，其中只放松从任意顶点指出的所有边（顺序任意），以下代码说明了这种方法的简洁性：

```
for (int pass = 0; pass < G.V(); pass++)
    for (v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

我们不会仔细研究这个版本，因为它总是会放松  $VE$  条边且只需稍作修改即可使算法在一般的应用场景中更加高效。

671

#### 4.4.6.5 基于队列的 Bellman-Ford 算法

其实，根据经验我们很容易知道在任意一轮中许多边的放松都不会成功：只有上一轮中的 `distTo[]` 值发生变化的顶点指出的边才能够改变其他 `distTo[]` 元素的值。为了记录这样的顶点，我们使用了一条 FIFO 队列。算法在处理正权重标准样图中进行的操作如图 4.4.22 所示。在示意图 4.4.22 左侧是每一轮中队列中的有效顶点（红色），紧接着是下一轮中的有效顶点（黑色）。首先将起点加入队列，然后按照以下步骤计算最短路径树。

- 放松边  $1 \rightarrow 3$  并将顶点 3 加入队列。
- 放松边  $3 \rightarrow 6$  并将顶点 6 加入队列。
- 放松边  $6 \rightarrow 4$ 、 $6 \rightarrow 0$  和  $6 \rightarrow 2$  并将顶点 4、0 和 2 加入队列。
- 放松边  $4 \rightarrow 7$ 、 $4 \rightarrow 5$  并将顶点 7 和 4 加入队列。放松已经失效的边  $0 \rightarrow 4$  和  $0 \rightarrow 2$ 。然后再放松边  $2 \rightarrow 7$ （并重新为  $4 \rightarrow 7$  着色）。
- 放松边  $7 \rightarrow 5$ （并重新为  $4 \rightarrow 5$  着色）但不将顶点 5 加入队列（它已经在队列之中了）。放松已经失效的边  $7 \rightarrow 3$ 。然后放松已经失效的边  $5 \rightarrow 1$ 、 $5 \rightarrow 4$  和  $5 \rightarrow 7$ 。此时队列为空。

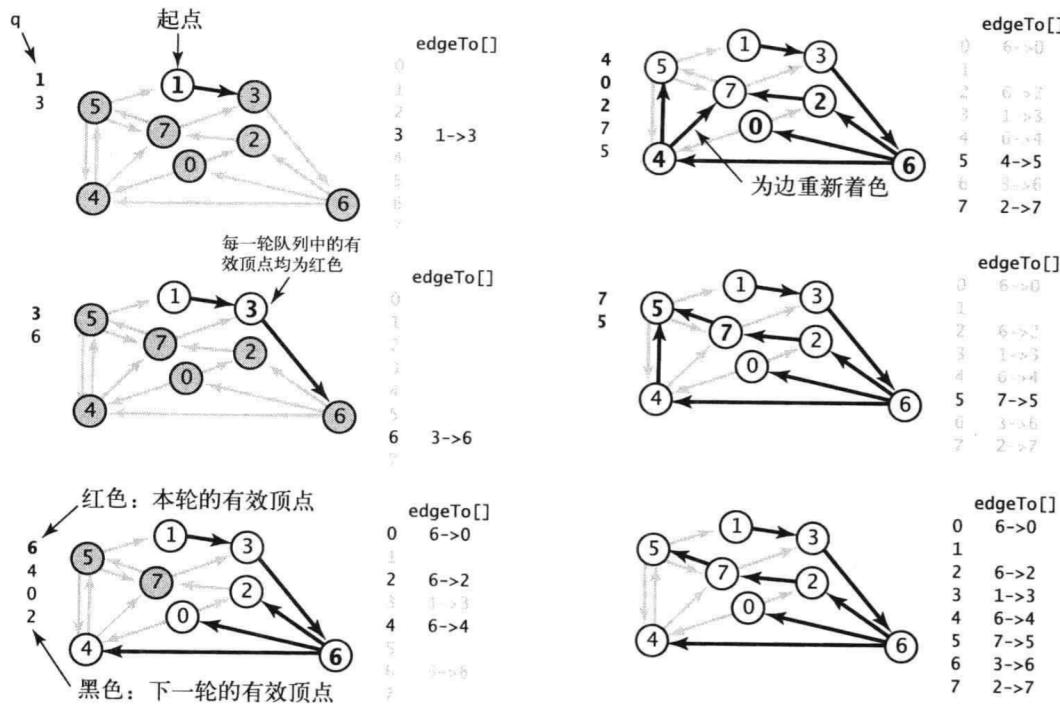


图 4.4.22 Bellman-Ford 算法的轨迹（另见彩插）

#### 4.4.6.6 实现

根据这些描述实现 Bellman-Ford 算法所需的代码非常少，如算法 4.11 所示。它基于以下两种其他的数据结构：

- 一条用来保存即将被放松的顶点的队列 `q`；
- 一个由顶点索引的 `boolean` 数组 `onQ[]`，用来指示顶点是否已经存在于队列中，以防止将顶点重复插入队列。[672]

首先，将起点  $s$  加入队列中，然后进入一个循环，其中每次都从队列中取出一个顶点并将其放松。要将一个顶点插入队列，需要修改 4.4.2.4 节框注“边的松弛”中 `relax()` 方法的实现，以便将被成功放松的边所指向的顶点加入队列中，如右边框注“Bellman-Ford 算法中的放松操作”所示。这些数据结构能够保证：

- 队列中不出现重复的顶点；
- 在某一轮中，改变了 `edgeTo[]` 和 `distTo[]` 的值的所有顶点都会在下一轮中处理。

要完整地实现该算法，我们就要保证在  $V$  轮后算法能够终止。实现它的一种方法是显式记录放松的轮数。我们的实现 `BellmanFordSP`（算法 4.11）使用了另一种方法，将会在 4.4.6.8 节详述：它会在有向图的 `edgeTo[]` 中检测是否存在负权重环，如果找到则结束运行。

**命题 Y。** 对于任意含有  $V$  个顶点的加权有向图和给定的起点  $s$ ，在最坏情况下基于队列的 Bellman-Ford 算法解决最短路径问题（或者找到从  $s$  可达的负权重环）所需的时间与  $EV$  成正比，空间和  $V$  成正比。

**证明。** 如果不存在从  $s$  可达的负权重环，算法会根据命题 X 在进行  $V-1$  轮放松操作后结束（因为所有最短路径含有的边数都小于  $V-1$ ）。如果的确存在一个从  $s$  可达的负权重环，那么队列永远不可能为空。根据命题 X，在第  $V$  轮放松之后，`edgeTo[]` 数组必然会包含一条含有一个环的路径（从某个顶点  $w$  回到它自己）且该环的权重必然是负的。因为  $w$  会在路径上出现两次且  $s$  到  $w$  的第二次出现处的路径长度小于  $s$  到  $w$  的第一次出现的路径长度。在最坏情况下，该算法的行为和通用算法相似并会将所有的  $E$  条边全部放松  $V$  轮。

673

```
private void relax(EdgeWeightedDigraph G, int v)
{
    for (DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQ[w])
            {
                queue.enqueue(w);
                onQ[w] = true;
            }
        }
        if (cost++ % G.V() == 0)
            findNegativeCycle();
    }
}
```

Bellman-Ford 算法中的放松操作

#### 算法 4.11 基于队列的 Bellman-Ford 算法

```
public class BellmanFordSP
{
    private double[] distTo; // 从起点到某个顶点的路径长度
    private DirectedEdge[] edgeTo; // 从起点到某个顶点的最后一条边
    private boolean[] onQ; // 该顶点是否存在于队列中
    private Queue<Integer> queue; // 正在被放松的顶点
    private int cost; // relax() 的调用次数
    private Iterable<DirectedEdge> cycle; // edgeTo[] 中的是否有负权重环
```

```

public BellmanFordSP(EdgeWeightedDigraph G, int s)
{
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    onQ = new boolean[G.V()];
    queue = new Queue<Integer>();
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;
    queue.enqueue(s);
    onQ[s] = true;
    while (!queue.isEmpty() && !this.hasNegativeCycle())
    {
        int v = queue.dequeue();
        onQ[v] = false;
        relax(G, v);
    }
}

private void relax(EdgeWeightedDigraph G, v)
// 4.4.6.6节框注“Bellman-Ford算法的放松操作”

public double distTo(int v) // 最短路径树实现中的标准查询算法（请见4.4.2.6节“最短路径API的查询方法”）
public boolean hasPathTo(int v)
public Iterable<Edge> pathTo(int v) // 4.4.6.8节框注“Bellman-Ford”的负权重检测方法

private void findNegativeCycle()
public boolean hasNegativeCycle()
public Iterable<Edge> negativeCycle()

// 请见6.4.6.8节 .
}

```

Bellman-Ford 算法的实现修改了 `relax()` 方法，将被成功放松的边指向的所有顶点加入到一条 FIFO 队列中（以避免出现重复顶点）并周期性地检查 `edgeTo[]` 表示的子图中是否存在负权重环（请见正文）。

674

基于队列的 Bellman-Ford 算法能够准确有效地解决最短路径问题并且在实际中被广泛应用，甚至包括正权重的情况。例如，如图 4.4.23 所示，在含有 250 个顶点的样图中，算法进行了 14 轮操作且对于相同的问题比较路径长度的次数少于 Dijkstra 算法。

#### 4.4.6.7 负权重的边

图 4.4.24 显示了 Bellman-Ford 算法在处理含有负权重边的有向图的轨迹。首先将起点加入队列 `q`，然后按照以下步骤计算最短路径树。

- 放松边  $0 \rightarrow 2$  和  $0 \rightarrow 4$  并将顶点 2、4 加入队列。
- 放松边  $2 \rightarrow 7$  并将顶点 7 加入队列。放松边  $4 \rightarrow 5$  并将顶点 5 加入队列。然后放松失效的边  $4 \rightarrow 7$ 。
- 放松边  $7 \rightarrow 3$  和  $5 \rightarrow 1$  并将顶点 3 和 1 加入队列。放松失效的边  $5 \rightarrow 4$  和  $5 \rightarrow 7$ 。
- 放松边  $3 \rightarrow 6$  并将顶点 6 加入队列。放松失效的边  $1 \rightarrow 3$ 。
- 放松边  $6 \rightarrow 4$  并将顶点 4 加入队列。这条负权重边使得到顶点 4 的路径变短，因此它的边需要被再次放松（它们在第二轮中已经被放松过）。从起点到顶点 5 和 1 的距离已经失效并会在下一轮中修正。

- 放松边  $4 \rightarrow 5$  并将顶点 5 加入队列。放松失效的边  $4 \rightarrow 7$ 。
- 放松边  $5 \rightarrow 1$  并将顶点 1 加入队列。放松失效的边  $5 \rightarrow 4$  和  $5 \rightarrow 7$ 。
- 放松无效的边  $1 \rightarrow 3$ 。队列为空。

在这个例子中，最短路径树就是一条从顶点 0 到顶点 1 的路径。从顶点 4、5 和 1 指出的所有边都被放松了两次。对照这个例子重读命题 X 的证明能够帮助你更好的理解这个算法。

#### 4.4.6.8 负权重环的检测

实现 BellmanFordSP 会检测负权重环来避免陷入无限的循环中。我们也可以将这段检测代码独立出来使得用例可以检查并得到负权重环。因此我们为表 4.4.4 中的 API 添加以下方法请见表 4.4.8。

表 4.4.8 为处理负权重环扩展最短路径的 API

<code>boolean hasNegativeCycle()</code>	是否含有负权重环
<code>Iterable&lt;DirectedEdge&gt; negativeCycle()</code>	得到负权重环 (如果没有则返回 null)

实现这些方法并不困难，如以下代码所示。在 BellmanFordSP 的构造函数运行之后，命题 Y 说明在将所有边放松  $V$  轮之后当且仅当队列非空时有向图中才存在从起点可达的负权重环。如果是这样，`edgeTo[]` 数组所表示的子图中必然含有这个负权重环。因此，要实现 `negativeCycle()`，会根据 `edgeTo[]` 中的边构造一幅加权有向图并在该图中检测环。我们会使用并修改 4.2 节中的 `DirectedCycle` 类来在加权有向图中寻找环（请见练习 4.4.12）。这种检查的成本分为以下几个部分。

- 添加一个变量 `cycle` 和一个私有函数 `findNegativeCycle()`。如果找到负权重环，该方法会将 `cycle` 的值设为含有环中所有边的一个迭代器（如果没有找到则设为 `null`）。
- 每调用  $V$  次 `relax()` 方法后即调用 `findNegativeCycle()` 方法。

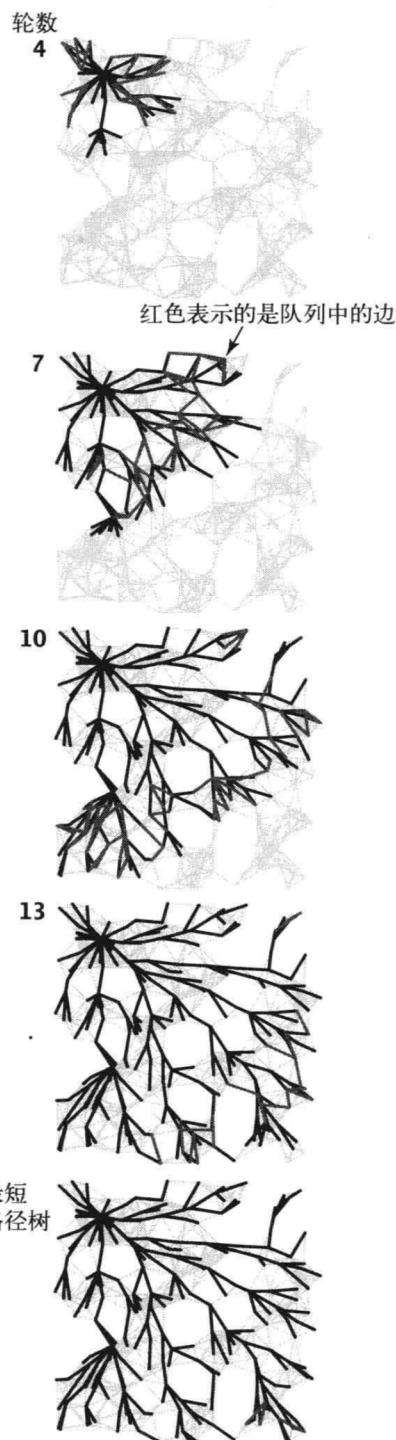


图 4.4.23 Bellman-Ford 算法 (250 个顶点) (另见彩插)

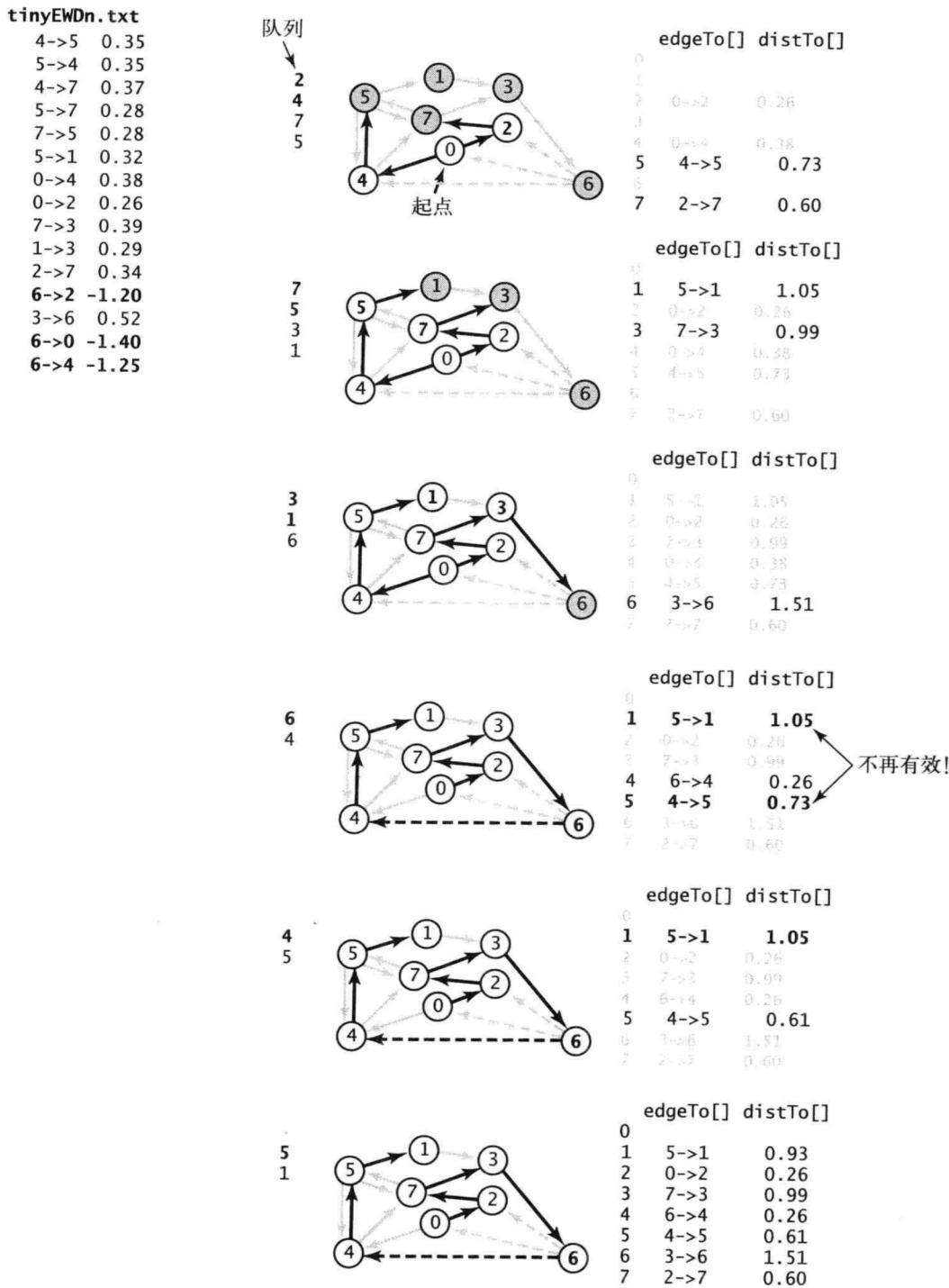


图 4.4.24 Bellman-Ford 算法的轨迹（图中含有负权重边）（另见彩插）

这种方法能够保证构造函数中的循环必然会终止。另外，用例可以调用 `hasNegativeCycle()` 来判断是否存在从起点可达的负权重环（并用 `negativeCycle()` 来获取这个环）。要在任意有向图中检测负权重环的存在只需稍作扩展即可（请见练习 4.4.43）。

图 4.4.25 是 Bellman-Ford 算法在一幅含有负权重环的有向图中的运行轨迹。头两轮放松操作与处理 tinyEWDn.txt 时是一样的。在第三轮中，算法在放松了边  $7 \rightarrow 3$  和  $5 \rightarrow 1$  并将顶点 3 和 1 加入队列后开始放松负权重边  $5 \rightarrow 4$ 。在这次放松操作中算法发现了一个负权重环  $4 \rightarrow 5 \rightarrow 4$ 。它将  $5 \rightarrow 4$  加入最短路径树中并在 edgeTo[] 中将环和起点隔离开来。从这时开始，算法沿着环继续运行并会减少到达所遇到的所有顶点的距离，直至检测到环的存在，此时队列非空。环被保存在 edgeTo[] 中，findNegativeCycle() 会在其中找到它。

```

private void findNegativeCycle()
{
    int V = edgeTo.length;
    EdgeWeightedDigraph spt;
    spt = new EdgeWeightedDigraph(V);
    for (int v = 0; v < V; v++)
        if (edgeTo[v] != null)
            spt.addEdge(edgeTo[v]);
    EdgeWeightedCycleFinder cf;
    cf = new EdgeWeightedCycleFinder(spt);
    cycle = cf.cycle();
}

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

public Iterable<Edge> negativeCycle()
{ return cycle; }

```

Bellman-Ford 算法的负权重环检测方法

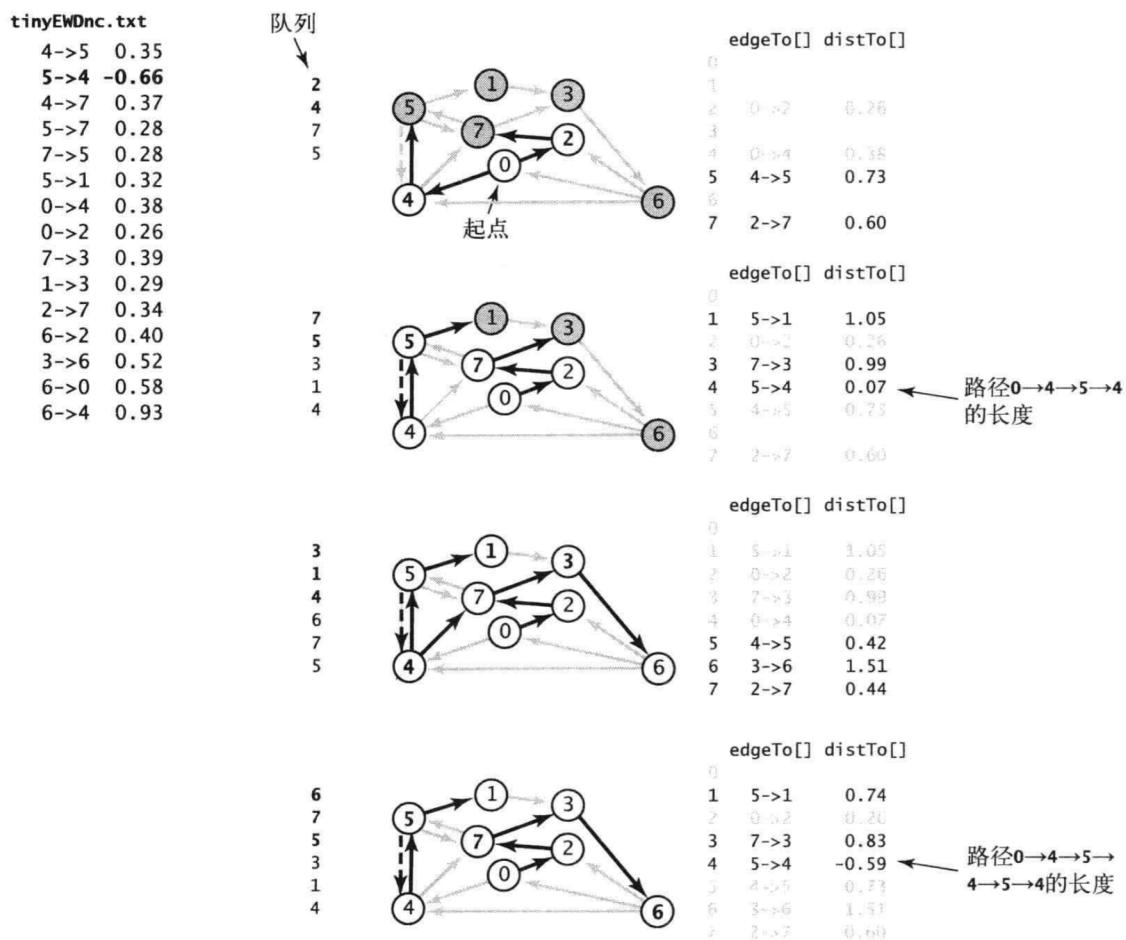


图 4.4.25 Bellman-Ford 算法的轨迹（含有负权重环的图，另见彩插）

#### 4.4.6.9 套汇

假设有一个基于商品贸易的金融交易市场。以下框注显示的是示例文件 rates.txt 的内容，你可以在任意货币兑换比例的表格中找到类似的内容。文件的第一行是货币的种类数  $V$ ，接下来的每一行都对应一种货币，开头是该货币的名称，紧接着是它和其他货币兑换的汇率。简单起见，这个例子中只包含了能够在现代市场中进行交易的数百种货币中的五种：美元（USD）、欧元（EUR）、英镑（GBP）、瑞士法郎（CHF）和加元（CAD）。第  $s$  行的第  $t$  个数字表示一个汇率，即购买一个单位的第  $s$  行的货币需要多少个单位的第  $t$  行的货币。例如，这张表告诉我们，1000 美元能够购买 741 欧元。这张表格等价于一幅完全的加权有向图，顶点对应着货币，边则对应着汇率。权重为  $x$  的边  $s \rightarrow t$  表示从货币  $s$  到货币  $t$  的汇率为  $x$ 。这张图中的路径则表示多次兑换。例如，将权重为  $y$  的边  $t \rightarrow u$  和刚才的边结合起来就得到了一条路径  $s \rightarrow t \rightarrow u$ ，即一个单位的货币  $s$  可以兑换为  $xy$  个单位的货币  $u$ 。比如，欧元可以兑换得到  $1012.206 = 741 * 1.366$  加元。注意，这比直接用美元兑换的汇率更高。你可能会以为  $xy$  总是应该等于边  $s \rightarrow u$  的权重，但这张表格所表示的金融系统非常复杂，并不总是能够保证这种一致性。因此，找到所有从  $s$  到  $u$  的路径中所有边的权重之积最大者就是我们最感兴趣的问题。一种更有趣的情况是，所有边的权重之积小于从终点指向起点的边的权重。在这个示例中，假设边  $u \rightarrow s$  的权重为  $z$  且  $xyz > 1$ 。那么环  $s \rightarrow t \rightarrow u \rightarrow s$  就能够用一个单位的货币  $s$  得到多于一个单位 ( $xyz$ ) 的货币  $s$ 。换句话说，将货币  $s$  兑换为  $t$ 、 $u$  并最后再兑换为  $s$  就可以得到  $100(xyz - 1)$  的利润。例如，如果将 1012.206 加元重新兑换为美元，可以得到  $1012.206 * 0.995 = 1007.14497$  美元，也就是得到了 7.14497 美元的利润。这看起来似乎不多，但一个外汇交易商可能会用一百万美元并在每分钟都进行一遍这样的交易，也就是说他每分钟的利润将超过 7000 美元，或者说每小时的利润超过 420 000 美元！这就是套汇交易的一个例子，请见图 4.4.26。如果没有外力的限制，

比如手续费或是交易金额上限，交易商可以从其中获取无限的利润。即使是在现实世界中的这些限制下，套汇的利润仍然是非常高的。这个问题和最短路径问题有什么关系呢？要回答这个问题非常简单。

% more rates.txt						
5						
USD	1	<b>0.741</b>	0.657	1.061	1.005	
EUR	1.349	1	0.888	1.433	<b>1.366</b>	
GBP	1.521	1.126	1	1.614	1.538	
CHF	0.942	0.698	0.619	1	0.953	
CAD	<b>0.995</b>	0.732	0.650	1.049	1	

**命题 Z。** 套汇问题等价于加权有向图中的负权重环的检测问题。

**证明。** 取每条边权重的自然对数并取反，这样在原始问题中所有边的权重之积的计算就转化为了新图中所有边的权重之和的计算。任意权重之积  $w_1 w_2 \dots w_k$  即对应  $-\ln(w_1) - \ln(w_2) - \dots - \ln(w_k)$  之和。转换后边的权重可能为正也可能为负。一条从  $v$  到  $w$  的路径表示将货币  $v$  兑换为货币  $w$ ，图中的任意负权重环都是一次套汇的好机会（请见图 4.4.27）。

在这个示例中，货币可以任意兑换，因此有向图是完全的，任意负权重环都是从任意顶点可达的。在一般的商品交易中，有些边可能并不存在，因此需要练习 4.4.43 所述的只有一个参数的构造函数。目前没有已知的寻找最佳套汇机会（图中负权重最小的环）的高效算法（图的规模不需要很大就能使所需的计算量超过计算机的承受能力），但找出任意套汇机会的最快算法仍然是很重要的——在第二快的算法找到任何套汇机会之前，使用这种算法的商人很可能已经可以系统地排除许多不佳的

套汇机会了。

$$0.741 * 1.366 * .995 = 1.00714497$$

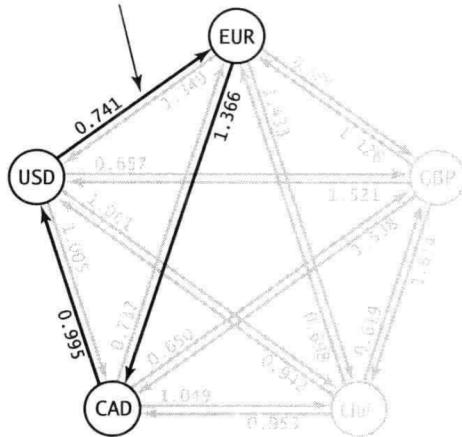


图 4.4.26 一次套汇机会

$$\begin{aligned} -\ln(0.741) &= -.2998 \\ -\ln(1.366) &= -.3119 \\ -\ln(0.995) &= .0050 \\ -.2998 - .3119 + .0050 &= -.0071 \end{aligned}$$

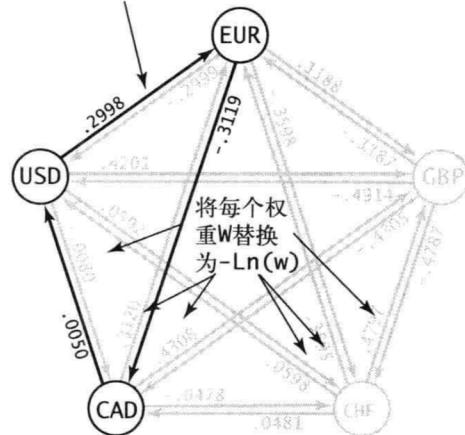


图 4.4.27 一个负权重环就表示了一次套汇的机会

### 货币兑换中的套汇

```
public class Arbitrage
{
    public static void main(String[] args)
    {
        int V = StdIn.readInt();
        String[] name = new String[V];
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(V);
        for (int v = 0; v < V; v++)
        {
            name[v] = StdIn.readString();
            for (int w = 0; w < V; w++)
            {
                double rate = StdIn.readDouble();
                DirectedEdge e = new DirectedEdge(v, w, -Math.log(rate));
                G.addEdge(e);
            }
        }
        BellmanFordSP spt = new BellmanFordSP(G, 0);
        if (spt.hasNegativeCycle())
        {
            double stake = 1000.0;
            for (DirectedEdge e : spt.negativeCycle())
            {
                StdOut.printf("%10.5f %s", stake, name[e.from()]);
                stake *= Math.exp(-e.weight());
                StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
            }
        }
        else StdOut.println("No arbitrage opportunity");
    }
}
```

这段代码调用了 BellmanFordSP 类来寻找汇率表中的套汇机会。它首先使用完全有向图表示汇率表，然后用 Bellman-Ford 算法来找图中的负权重环。

```
% java Arbitrage < rates.txt
1000.00000 USD = 741.00000 EUR
741.00000 EUR = 1012.20600 CAD
1012.20600 CAD = 1007.14497 USD
```

命题 Z 的证明即使在没有套汇机会的情况下仍然有用，因为它将货币兑换问题转化为了一个最短路径问题。因为对数函数是单调的（且会对计算的结果取反），当边的权重之和最小时汇率之积正好最大。尽管边的权重可正可负，从  $v$  到  $w$  的最短路径仍然是将货币  $v$  兑换为货币  $w$  的最好方法。

679  
l  
681

#### 4.4.7 展望

表 4.4.9 总结了本节中我们所学习到的各种最短路径算法的重要性质。在这些算法中进行选择的第一个条件是问题所涉及的有向图的基本性质。它含有负权重的边吗？它含有环吗？它含有负权重的环吗？除了这些基本性质之外，加权有向图的特性多种多样，因此在有多个合适的选择时就需要通过实验找出最佳的算法。

表 4.4.9 最短路径算法的性能特点

算 法	局 限	路径长度的比较次数 (增长的数量级)		所需空间	优 势
		一般情况	最坏情况		
Dijkstra 算法 (即时版本)	边的权重必须为正	$E \log V$	$E \log V$	$V$	最坏情况下仍有较好的性能
拓扑排序	只适用于无环加权有向图	$E + V$	$E + V$	$V$	是无环图中的最优算法
Bellman-Ford 算法(基于队列)	不能存在负权重环	$E + V$	$VE$	$V$	适用领域广泛

#### 历史资料

自 20 世纪 50 年代以来，最短路径算法就已经被深入地研究并被广泛应用了。计算最短路径的 Dijkstra 算法的历史和计算最小生成树的 Prim 算法的历史背景相似（并且也相关）。Dijkstra 算法既指的是按照顶点距离起点的远近顺序构造最短路径树的算法，也指的是该算法的实现，（它也是最适合用邻接矩阵表示的算法。），因为 Dijkstra 在 1959 年的一篇论文中发表了上述观点（并且证明了这种方法同样也可以用来计算最小生成树）。稀疏图算法的性能改进来自于之后对优先队列实现的改进，不仅仅针对最短路径问题。这其中最重要的是 Dijkstra 算法性能的改进。（例如，使用斐波那契堆后最坏情况下的复杂度可以提高到  $E + V \log V$ ）。实践证明 Bellman-Ford 算法十分有效并且应用领域广泛，特别是处理一般性的加权有向图。由于 Bellman-Ford 算法计算普通应用的运行时间常常是线性的，因此在最坏情况下它的运行时间是  $VE$ 。最坏情况下的运行时间为线性级别的稀疏图的最短路径算法是一个仍在研究之中的问题。Bellman-Ford 算法最早由 L.Ford 和 R.Bellman 发表于 20 世纪 50 年代。尽管我们已经看到许多其他的图算法性能得到了大幅改进，但是处理含有负权重边（但不含负权重环）的且在最坏情况下性能更好的有向图算法还没有出现。

682  
l  
683