## 第9章 Edsger及其朋友们从A到B的旅程

*两点之间最短的距离尚在建设中。*

— 诺艾利·阿尔蒂托（Noelie Altito）

现在，我们可以回顾一下序言部分的第二个问题：[[1]](#footnote-1)如何确定从喀什到宁波的最短路线？如果把这个问题交给电子地图软件，不到一秒钟就可以得到答案。现在这个问题似乎没有一开始那么神秘了，我们甚至还可以在工具的帮助下编写相应的程序。解题思路是很清楚的，如果道路的各条分岔长度相等，使用广度优先搜索（BFS）算法就可以找到最短路径。而只要图中没有环，还可以使用DAG最短路径算法。然而，中国的公路地图既有环，道路也不等长。不过值得庆幸的是，本章会给出解决这个问题的有效算法！

为免读者误会本章只能供编写电子地图软件一用，我们可以思考一下最短路径抽象化适用的其他场合。例如，我们可以用这种思想实现网络的高效浏览，互联网传输采用了各种各样的数据包路由方法。事实上，网络中*充斥*着这样的路由算法，它们都工作在幕后。不过，这样的算法还可用于完成较为隐晦的图算法寻路任务，比如让计算机游戏角色有策略地移动。再比如，你也许想用最少的步数走通某种形式的迷宫？这等同于在状态空间中确定最短路径，状态空间是表示迷宫状态（节点）和移动轨迹（边）的抽象图。又或许，你想利用汇率差异创收？本章介绍的一种算法可以助你一臂之力（见练习9-1）。

寻找最短路径也是其它非图类算法中一个重要的子程序。例如，确定*n*个人和*n*个职位[[2]](#footnote-2)之间最佳匹配的一种常见算法就需要反复寻找最短路径。我曾编写过一个修复XML文件的程序，插入开始和结束标签，以满足某个简单的XML Schema的规定（比如“列表的项目需要嵌套在列表标签中”）。后来我发现，本章介绍的一个算法可以轻松解决这样的问题。其他的应用领域还有运筹学、集成电路制造、机器人技术，不一而足。寻找最短路径绝对是一个不容回避的问题。幸运的是，虽然算法的某些部分有一定难度，但我们已经在前几章对其中困难的内容做了详尽阐释。

最短路径问题可分为以下几种形式。例如，在有向图和无向图中找出最短路径，二者最重要的区别在于起点和目的地。你是想找到从一个节点到其他所有节点的最短路径（源节点唯一），从一个节点到另一节点的最短路径（单对节点，一对一，点对点），从其他所有节点到一个节点的最短路径（目标节点唯一），还是从所有节点到所有其他节点的最短路径（所有节点两两组合）？其中，源节点唯一和所有节点两两组合这两种情况可能是最重要的。我们虽然有一些诀窍可以解决单对节点问题（见下文的“中途相遇”和“把握未来走向”），但无法保证解题速度比一般的源节点唯一的情况快。目标节点唯一问题当然等同于源节点唯一（有向图只需将边翻转）。对于所有节点两两组合的问题，可以将每个节点分别视为源节点唯一的情况（下文会做进一步介绍），不过也有专门的算法可以解决这个问题。

### 扩展知识

本书第4章引入了松弛和逐步改进的思想，第8章则介绍了如何将这一思想应用于求解DAG图的最短路径问题。事实上，针对DAG图（程序清单8-4）的迭代最短路径算法不仅是动态编程的一个典型案例，它也阐明了本章算法的基本结构：对图的边采用松弛技术，将有关最短路径的知识扩展开去。

下面，我们回顾一下其中的工作原理：把图表示为字典的字典，用字典D存放距离值估计（上界值）。此外，还要增加一个前导节点字典P， 这和第5章介绍的很多遍历算法的做法相同。这些前导指针构成了所谓的最短路径树，可以帮助我们重建与D中距离对应的实际路径。然后，如程序清单9-1所示，松弛技术就可以作为公共代码relax函数分解出来。需要注意的是，我把D中不存在的项目值都视为无穷大。（当然，也可以在主算法中将它们初始化定义为无穷大。）

***清单9-1.*** 松弛技术

inf = float('inf')

def relax(W, u, v, D, P):

d = D.get(u,inf) + W[u][v] # 可能的最短路径估计

if d < D.get(v,inf): # 确定为最短路径？

D[v], P[v] = d, u # 更新距离值估计和父节点信息

return True # 字典有更新！

这其中蕴含的思想是，我们通过途经u，看看是否可以缩短路程，从而改进目前已知的到达v点的最短路径。如果事实证明不是最短路径，没有关系，我们不再理会。如果确实是最短路径，就记下新的距离值，记住这条路径经过了哪些节点（将P [v]设为u）。我还额外增加了一个小功能：返回值表明是否确实发生了改变，这个功能稍后会派上用场（不过，并非所有的算法都需要这个功能）。

下面，我们看看程序的运行情况：

>>> D[u]

7

>>> D[v]

13

>>> W[u][v]

3

>>> relax(W, u, v, D, P)

True

>>> D[v]

10

>>> D[v] = 8

>>> relax(W, u, v, D, P)

>>> D[v]

8

如上所示，relax函数的首次调用后，D [v]从13改进为10，这是因为找到了一条途经u的最短路径，从起点到u的距离为7，而u到v的距离仅为3。然后，我又发现可以通过一条长为8的路径到达v，再次运行relax函数，但这时并未发现小于8的最短路径，所以字典保持不变。

我们可以这样推测，如果设D [u]为4，再次运行relax函数，D [v]这次会更新为7，改进的距离值估计从u扩展到v。这种路径知识的扩展就是relax函数的核心功能。如果随机松弛各边，对距离所做的任何改进（及其对应的路径）最终将扩展到全图各点。所以，如果永远随机地使用松弛技术，最终一定会得到正确答案。不过，永远意味着相当漫长的等待......

这正是松弛策略游戏（第4章曾简要涉及）的用武之地：我们要尽可能少地调用relax函数，获得准确的结果。我们需要付出的代价多少取决于待解问题的具体性质。例如，对于DAG图，我们需要*对每边都调用一次relax函数*，这显然是可以想到的最好办法。下面我们还会看到，对于比较常见的图形，实际付出的代价是很小的（不过总运行时间较长，而且不允许边权值为负）。不过在此之前，我们先来看看后面会用到的一些重要事实。下列规则的假设前提是从节点s出发，D[s]初始值为零，所有其他节点的距离值估计设为无穷大。设d(u,v)是从u到v的最短路径的长度。

* d(s,v) <= d(s,u) + W[u,v]. 这是*三角不等式*的一个例子。
* d(s,v) <= D[v]. 对于除s以外的节点v，D[v]的初始值为无穷大，一旦发现最短路径，数值才会减小。我们绝对不会”作弊”，所以它始终表示上界值。
* 如果节点v无路可达，松弛技术就不可能改变D[v]的无穷大状态，因为找不到可以改进D[v]的最短路径。
* 假设到v的最短路径由从s到u的路径和从u到v的一条边构成。如果在对从u到v的边采用松弛技术之前，D[u]的值始终是正确的（即D[u] == d(s,u)），那么采用松弛技术后，D[v]的值始终是正确的，P[v]定义的路径也是正确的。
* 假设 [s, a, b, ... , z, v]是从s到v的最短路径。假定对该路径的各边(s,a), (a,b), ... , (z,v)依次使用了松弛技术，那么不论其他边在此期间是否进行过松弛操作，D[v]和P[v]的值都是正确的。

继续阅读下文前，读者需要确保自己理解上述声明的正确性，这样便于理解本章的剩余内容。

### 松弛可“疯狂”

随机松弛是有点疯狂，但疯狂地“松弛”就不一定了。比方说，对*所有*边都采用松弛技术，顺序可以随机，没有关系，只要保证照应到所有的边。然后，再做一次，也许换种次序，但一定要照应到所有的边。然后，再来一次，一次又一次，直到不再发生任何变化为止。

|  |
| --- |
| **提示** 可以想象一下，每个节点都在不断吆喝自己的方案，根据己方目前掌握的最短路径，向相邻节点推销最短路径。如果节点得到的报价比自己目前掌握的更优惠，它就会更换路径供应商，并相应降低自己的报价。 |

这似乎并没有什么不合理之处，至少对于初次尝试而言。不过，这里遇到两个问题：需要多长时间才能等到没有任何变化发生（如果能够等到的话）？到了那时，如何保证答案的正确性？

我们先来看一个简单的例子。假设所有的边权值非负且相同。这意味着松弛操作只有找到较少边组成的路径，才能找到最短路径。那么，如果对所有的边都做了一次松弛操作，将会发生什么？最起码，节点s的所有邻居都会有正确答案，并在最短路径树中将s定为自己的的父节点。根据各边的松弛次序不同，最短路径树也许会进一步扩展开去，但我们无法保证这一点。如果对所有边再做一次松弛操作呢？最起码，最短路径树会至少扩展一层。事实上，在最坏的情况下，最短路径树将像某种效率极其低下的BFS算法那样，层层扩展下去。对于有*n*个节点的图而言，路径最多可以有*n*-1条边，因此所需迭代次数的最大值为*n*-1。

不过，通常而言，对边所做的假定没有这么多（假如可以这样的话，我们还不如使用BFS算法，它的表现会很出色）。由于边的权值各有不同（甚至可能为负数），后几轮的松弛操作可能会修改前几轮设定的前导指针。例如，一轮松弛操作后，节点s的相邻节点v会将P [v]值设为s，但我们不能确定这是正确的！也许之后又发现了一条通过其他节点到达v更短的路径，P [v]值将被覆盖。那么，对所有边都做了一轮松弛操作后，我们可以掌握哪些情况呢？

请回想一下前一节列出的一条原则：如果对所有的边沿着从s到v最短路径的顺序进行松弛操作，路径答案（含D和P）就是正确的。具体来说，在这种情况下，我们对所有由一条边组成的最短路径上的所有边都做了松弛操作。请注意，我们不知道这些路径的位置，因为我们还不知道各种最优路径包含哪些边。不过，虽然将s与其相邻节点连在一起的一些P边很可能不是最终答案，但我们知道，正确的边一定都已包含在其中了。

解题还在继续，经过对图中各边的*k*轮松弛操作后，现已获得由*k*条边组成的所有最短路径。根据之前的推理，对于有*n*个节点和*m*条边的图而言，找到正确答案最多需要进行*n*-1轮，程序运行时间为Q(*nm*)。当然，这只是最坏情况下的运行时间。如果我们增加一个检查步骤：上一轮之后字典是否有变化？如果没有，就不用再继续操作下去。我们甚至还可以抛开*n*-1，只看这一项检查。毕竟，之前的推理认为，最多不会超过*n*-1轮，所以通过检查就可以判断是否停止运行算法。对还是不对？不对。这里还有一个问题：负环。

负环是最短路径算法的天敌。如果没有负环，凭借“没有变化”的条件判定终止程序是没有问题的，可一旦出现负环，距离值估计就要永远改进下去。所以，只要允许负边的存在（为什么不呢？），就需要用迭代次数作为保障。好消息是，我们可以用迭代次数来*检测*负环的存在：运行*n*轮而不是*n*-1轮，看看前一次迭代是否引发变化。如果确实获得改进（本不该出现的），就可以得出结论：“这是负环干的！”我们可以宣布答案无效，放弃查找。

■ **备注** 请不要有什么误解：即便有负环，找到最短路径也是完全有可能的。答案不允许包含环，所以负环不会影响答案。我只是说，在允许负环存在的情况下*找到*最短路径是一个尚未解决的问题（见第11章）。

现在，我们谈一谈本章第一个正式算法——Bellman-Ford算法（参见程序清单9-2）。这是一个适用于任意有向或无向图的单源最短路径算法。如果图包含负环，算法将报告这一事实，并放弃查找。

***清单9-2.*** Bellman-Ford算法

def bellman\_ford(G, s):

D, P = {s:0}, {} # 到s的起始距离为0；无父节点

for rnd in G: # n = len(G) （总轮数）

changed = False # 本轮目前无变化

for u in G: # 遍历出节点...

for v in G[u]: # ... 及其入节点...

if relax(G, u, v, D, P): # 确定从u到v的最短路径?

changed = True # 确定! 有变化发生

if not changed: break # 本轮无变化：结束

else: # 第n轮前未结束?

raise ValueError('negative cycle') # 检测到负环

return D, P # 否则：D和P正确

请注意，Bellman-Ford算法实现的与众不同之处，恰恰在于它包含了检查变化的changed。这种检查机制带来两个好处。首先，如果不需要多次迭次，程序可以提前终止；第二，可以通过判断前次“多余的”迭代是否带来变化，检测负环的存在与否。（如果没有这样的检查，更常见的做法是添加一段单独的代码，实现这最后一次迭代，再配上变化检查机制。）

由于该算法是其他几种算法的基础，我们需要弄清楚它的工作原理。回顾第2章加权图的例子，我们可以将其类型指定为字典的字典，如下所示：

a, b, c, d, e, f, g, h = range(8)

G = {

a: {b:2, c:1, d:3, e:9, f:4},

b: {c:4, e:3},

c: {d:8},

d: {e:7},

e: {f:5},

f: {c:2, g:2, h:2},

g: {f:1, h:6},

h: {f:9, g:8}

}

图9-1呈现了该图的全貌，我们不妨称之为bellman\_ford(G, A)。程序运行时究竟发生了什么？要了解详情，我们可以使用调试器，或者跟踪或运行记录包。为简单起见，我们不妨增加两行打印命令，显示进行松弛操作的边，以及分配给D的值。我们还可以按照排序顺序（使用排序命令）遍历节点及其相邻节点，以获得确定的结果。

（图）

***图 9-1.*** *一个权重图的例子*

得到的打印显示结果应该是这样的：

(a,b) D[b] = 2

(a,c) D[c] = 1

(a,d) D[d] = 3

(a,e) D[e] = 9

(a,f) D[f] = 4

(b,c)

(b,e) D[e] = 5

(c,d)

(d,e)

(e,f)

(f,c)

(f,g) D[g] = 6

(f,h) D[h] = 6

(g,f)

(g,h)

(h,f)

(h,g)

这是Bellman-Ford算法第一轮运行的结果；可以看出，算法遍历各边一次。打印输出又进行了一轮，但没有值分配给D，函数返回。其中有些草率：距离值估计D[e]先设定为9，即从a至e的直达距离。当对(a, b)和(b, e)都进行松弛操作后，我们发现了更好的选择，即长为5的路径a, b, e。不过，我们的运气相当不错，因为只需要遍历各边一次就找到了正确答案。下面我们试试更有意思的点子，强迫算法又进行一轮后再稳定下来。你有什么办法吗？一种办法是：

G[a][b] = 3

G[a][c] = 7

G[c][d] = -4

现在我们发现了一条通过f到达d的最短路径，但第一轮时没有找到。

(a,b) D[b] = 3

(a,c) D[c] = 7

(a,d) D[d] = 3

(a,e) D[e] = 9

(a,f) D[f] = 4

(b,c)

(b,e) D[e] = 6

(c,d)

(d,e)

(e,f)

(f,c) D[c] = 6

(f,g) D[g] = 6

(f,h) D[h] = 6

(g,f)

(g,h)

(h,f)

(h,g)

我们在第一轮把D[c]降到了6，不过那时我们已经对边(c,d)做了松弛操作，那条边再无改进空间，因为D[c]是7，D[d]是3。不过，到了第二轮，你会发现

(c,d) D[d] = 2

而到了第三轮，距离值估计已经稳定。

最后，我们在这个例子中引入负环。保持原先的各边权值不变，只做一点改动如下：

G[g][h] = -9

我们不去管不改变D的松弛操作，在打印结果中加入一些轮数编号，可得到如下结果：

# 第1轮：

(a,b) D[b] = 2

(a,c) D[c] = 1

(a,d) D[d] = 3

(a,e) D[e] = 9

(a,f) D[f] = 4

(b,e) D[e] = 5

(f,g) D[g] = 6

(f,h) D[h] = 6

(g,h) D[h] = -3

(h,g) D[g] = 5

# 第2轮：

(g,h) D[h] = -4

(h,g) D[g] = 4

# 第3轮：

(g,h) D[h] = -5

(h,g) D[g] = 3

# 第4轮：

(g,h) D[h] = -6

(h,f) D[f] = 3

(h,g) D[g] = 2

...

# 第8轮：

(g,h) D[h] = -10

(h,f) D[f] = -1

(h,g) D[g] = -2

Traceback (most recent call last):

...

ValueError: negative cycle

我删去了其中几轮，不过你肯定已经发现了规律：第3轮后，g、h 和 f的距离值估计不断减1。即使到了第8轮还有减1的情况，考虑到一共只有8个节点，这就提示了负环的存在。这并不意味着此题无解，只是说明持续的松弛操作无法找到解决方案，我们就此提出一个异常情况。

当然，如果最近路径不经过负环，就不成问题。比方说，我们可以用del G[f][g]去掉边(f,g)。现在，最起码f不会参与构成环了，但g和h还会无休止地”改进”彼此的距离值估计。但是，如果我们把边(f,h)也去掉，负环问题不复存在！

(a,b) D[b] = 2

(a,c) D[c] = 1

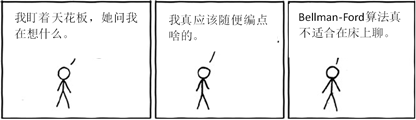
(a,d) D[d] = 3

(a,e) D[e] = 9

(a,f) D[f] = 4

(b,e) D[e] = 5

各节点的互连关系未变，负环依然存在，但遍历过程不再经过负环。如果这种调整让你觉得怪怪的，放心好了：到g和到h的距离都是正确的，都是无穷大，因为理应如此。不过，如果调用bellman\_ford(G, g)或bellman\_ford(G, h)，负环再次变成可达，就会得到混乱的操作结果，每轮都有若干更新，最终是负环异常。



***枕边情话。****或许我该试试Wexler算法的? (*[*http://xkcd.com/69*](http://xkcd.com/69)*)*

### 找到隐藏的DAG图

Bellman-Ford算法表现十分出色。从许多方面看，它都是本章最容易理解的算法：只要对所有的边反复进行松弛操作，直到确定一切答案正确为止。对于任意图而言，这都是一个很好的算法，但如果可以事先做一些假定，就能（通常如此）收到更好的效果。前文曾提过，单源DAG图最短路径问题可以用线性时间解决。不过在本节中，我们需要处理一个不同的约束条件。环还是可以有的，但*边的权值不能为负*。（其实，这是实际生活的常态，比如引言中讨论的各种情况。）这不仅意味着我们可以抛开负环的烦恼，还能给出关于各种距离是否正确的结论，从而显著改进运行时间。

下面要构建的算法，是算法学泰斗艾兹赫尔·戴克斯特拉（Edsger W. Dijkstra）在1959年设计的，可以用多种方式解释。理解这种算法的正确性可能有点棘手。我觉得可以把它看作是DAG最短路径算法的近亲，二者的重要区别在于这种算法的目标是*揭示一个隐藏的DAG图*。

尽管求解路径的图结构不限，我们可以认为其中一些边与解题无关。首先，我们可以假定，从源节点到其他各个节点的距离*已知*。当然，我们实际上并不知道，但这种假想情形有助于我们的推理。再设想各节点按距离大小从左到右排序。排序的重要性不大，但假设边的权值没有负数，这一点至关重要。

因为所有的边权都是正值，假想的节点排序结果，能够帮助我们确定，一个节点的下一步解只能位于其*左侧*。在右侧找到构成最短路径的节点是不可能的，因为右侧节点到该节点的距离较远，除非它有一条后向边为负值，否则无法构成最短路径。正的后向边对我们根本没用，与求解问题无关。这样一来，我们要解决的就是一个DAG图，用到的拓扑排序就是一开始假设的顺序——依据实际距离大小的节点排序。

这种求解结构的图示请参阅图9-2。（稍后解释其中的问号含义。）

（图）

***图 9-2.*** *逐步揭示隐藏的DAG图。节点上标记着最终距离。因为权值为正，后向边（虚线）无法影响结果，因此与待解问题无关。*

不出所料，现在我们遇到了解题的重要关口：路径有环。我们原本以为，通过揭示基本问题结构（分解成子问题或找到隐藏的DAG图），就可以解决这个问题。不过，推理过程还是有用的，因为我们现在有了具体的目标。我们要找到节点的排序，就需要依靠值得信赖的伙伴——归纳法！

让我们再回顾一下图9-2。假设突出显示的节点就是我们的逐步归纳所需要找到的（即之前的节点已经确定，并已定好正确的距离值估计）。和解决普通的DAG最短路径问题一样，一旦确认节点构成最短路径，并且确定其正确距离，我们就对这个节点的出边都进行松弛操作。这意味着，我们对它之前的所有节点的出边都进行了松弛操作。我们还没有对它*之后*的节点的出边进行松弛操作，但如前所述，这无关紧要，因为这些之后的节点的距离值估计都是上界值，而且回边的权值为正，所以它们不可能构成最短路径。

这意味着（根据早先的松弛技术特性或参见第8章有关DAG最短路径算法的讨论），下个节点的*距离值估计一定是正确的*。换言之，图9-2中突出显示的节点现在一定已经更新了正确的距离值估计，因为我们对前三个节点的所有出边都做了松弛操作。这是非常好的消息了，剩下的工作就是要弄清楚*到底哪个节点*符合这样的情况。记得吗？我们还不知道排序是怎样的呢。我们要一步步计算出拓扑排序。

当然，只有一个节点有可能成为最短路径的下个节点：[[3]](#footnote-3)这个节点拥有*最短的距离值估计*。我们知道根据排序，它就是下个节点。我们还知道它的距离值估计是正确的，因为这些估计都是上界值，之后的节点不可能有更小的估计。很精彩吧？至此，我们通过归纳解决了这个问题。我们按照距离排序，对每个节点的所有出边都做了松弛操作，这意味着始终选取距离值估计最低的节点作为下个节点。

这种结构和Prim算法相当类似：使用优先级队列进行遍历。正如Prim算法所示，我们知道，遍历尚未发现的节点不可能做过松弛操作，因此我们（到目前为止）对它们是不感兴趣的。而对于那些我们*已经*发现（并已做过松弛操作）的节点，我们最感兴趣的是其中优先级最低的。在Prim算法中，优先级是向后连回遍历树的边的权值；在Dijkstra算法中，优先级是距离值估计。当然，在我们寻找最短路径的过程中，优先级会发生变化（就像新出现的可能的生成树的边会降低Prim算法中的优先级），但就像程序清单7-5所说明的那样，我们可以反复向堆添加同一个节点（而不是修改堆条目的优先级），而不会影响正确性或运行时间。运行结果见程序清单9-3。其运行时间为对数线性函数，即Q((*m*+*n*) lg *n*)，其中*m*为边数，*n*为节点数。由此可见，对于（1）从队列中提取出的每个节点；和（2）要接受松弛操作的每条边都需要进行（对数）堆操作。[[4]](#footnote-4)如果从起始节点出发可以到达Q(*n*)个节点，只要图有W(*n*)条边，运行时间就可以简化为Q(*m* lg *n*)。

***清单 9-3.*** Dijkstra算法

from heapq import heappush, heappop

def dijkstra(G, s):

D, P, Q, S = {s:0}, {}, [(0,s)], set()

#距离值估计, 最短路径树，优先级队列，已访问节点集

while Q: # 节点尚未处理?

\_, u = heappop(Q) # 最小距离估计的节点

if u in S: continue # 节点已访问？ 跳过

S.add(u) # 现已访问

for v in G[u]: # 遍历其所有相邻节点

relax(G, u, v, D, P) # 对出边进行放松操作

heappush(Q, (D[v], v)) # 添加到队列中，以距离值估计作为优先级

return D, P # 返回D和P的最终结果

Dijkstra算法虽与Prim算法类似（队列优先级规定不同），不过和另一个经典算法同样关系密切——广度优先搜索（BFS）算法。我们可以考虑一下边权值为正整数的情况。将权值为*w* 的一条边替换为*w*-1 条无权边, 连成一条由虚拟节点组成的路径（见图9-3）。我们损失了找到*高效*算法的机会（见练习9-3），但我们知道BFS 算法会找到*正确*的解决方案。事实上，它解决问题的方式与Dijkstra算法十分相似：在各条（原始）边上花费的时间与边权值成正比，所以会按距离大小从起始节点依序到达各（原始）节点。



长度=3 3条边

***图 9-3.*** *虚拟节点模拟一条边的权值或长度*

这有点像沿各边立起一系列多米诺骨牌（骨牌数与权值成正比），然后推倒起始节点的第一张骨牌。一个节点有多个方向可达，但我们通过观察哪张骨牌被压在下面，就可以判断哪个方向是最终赢家。

假如我们一开始就使用这种方法，那么Dijkstra算法就可以看做是一种通过“模拟”BFS算法或多米诺骨牌（或者流水、声波扩散，等等）获得性能的方法，无需分别处理各个虚拟节点（或多米诺骨牌）。其实，可以把优先级队列看做是一条时间轴，用来记录沿各种路径到达节点的不同时刻。我们观察新发现的边的长度，思考这样一个问题，“如果走这条边，多米诺骨牌何时可以到达目标节点？”我们将这条边花费的时间（边的权值）加入到当前时间（到当前节点的距离），把结果记录在时间轴（堆）上。我们对首次到达的节点都进行如此操作（说到底，我们只关心*最短的*路径），继续沿时间轴移动，到达其他节点。之后，如果我们在时间轴上再次到达相同节点，就不再管它。[[5]](#footnote-5)

上文已清楚说明了Dijkstra算法与DAG最短路径算法的相近之处。其实就是动态编程的应用，不过递归分解没有DAG那么明显。为获得解决方案，Dijkstra算法也使用了贪婪算法，因为它总是移动到拥有当前距离值估计最小的节点。用二叉堆作为优先级队列，还会涉及分治算法。总体而言，这是一个漂亮的算法，用到了本书介绍过的很多内容，值得花些时间充分理解。

### 多对多问题

在下一节中，我们将介绍一个非常精彩的求解所有节点对之间最短距离的算法。即使图有很多条边，这种专用算法也可以有效解决问题。不过，本节将简要谈谈将前两个算法——Bellman-Ford算法和Dijkstra算法——结合在一起的方法，适用于求解稀疏图（即边数相对较少的图）。这就是Johnson算法，很多算法设计的课程和书籍似乎都忽视了它的存在，但它确实是一个聪明的算法，凭借已有的知识储备，我们可以轻松掌握。

Johnson算法的动机如下：解决稀疏图所有节点对之间的最短路径问题，对从各个节点出发的情况使用Dijkstra算法，这其实是一个很好的解决办法，它本身并不足以激发新的算法产生......但问题是，Dijkstra算法不允许负权边的存在。对于单源最短路径问题，除了改用Bellman-Ford算法，我们没有其他办法。不过，对于全节点对问题，我们可以进行初步的预处理，让所有的边权值为正。

我们的想法是，增加一个新的节点*s*，它到所有现有节点的边权值为零，然后对从*s*出发的情况运行Bellman-Ford算法。这样可以计算出从*s*到图中每个节点的距离，我们称之为*h* (*v*)。然后我们可以用*h*调整各边的权值，定义新的权值如下：*w* ’(*u*,*v*) = *w*(*u*,*v*) + *h*(*u*) - *h*(*v*)。这个定义有两个非常有用的特点。首先，它保证每个新的权值*w*’(*u*,*v*)非负（服从本章之前讨论过的三角不等式，另见练习9-5）。其次，我们并不会为待解问题引入干扰项！也就是说，如果找到了有新权值的最短路径，它就是有原始权值的最短路径（不过边数所代表的长度有所不同）。那么，为什么会这样呢？

有一个可爱的思想可以对此做出解释，它叫*套筒式求和*：比如，（(*a* - *b*) + (*b* - *c*) + ... + (*y* - *z*)这样的加和算式就像望远镜的套筒结构一样，层层套叠，最后得出*a* – *z*。这是因为，其他变量都是既在加号之前带着负号出现一次，又在加号之后带着正号出现一次，所以，它们的总和为零。Johnson算法修正后的各边所构成的每条路径也是同样的情况。对于路径中的任意边(*u*,*v*)而言，权值的修正方法是先加上*h*(*u*)，再减去*h* (*v*)。下一条边将把*v*作为第一个节点，加上*h* (*v*)，就此从总和中消去。同样地，之前的边把*h*(*u*)减去，也是两两抵消。

只有两条边稍有不同（任何路径都是如此），即第一条和最后一条。第一条是没有问题的，因为*h* (*s*)是零，而所有节点的*w* (*s*,*v*)都是零。那么最后一条呢？也不是问题。最后一个节点*v*.对应的*h* (*v*)是带负号的，但这对终止于该节点的所有路径都是如此——得到的最短路径还是最短的。

这种变换也没有舍弃任何信息，所以只要我们使用Dijkstra算法发现最短路径，就可以逆向变换所有的路径长度。使用类似的套筒理论，我们可以通过对基于变换后权值的答案加*h*(*v*)再减*h*(*u*)，得到从*u*到*v*的最短路径的实际长度，这就是程序清单9-4实现的算法。[[6]](#footnote-6)

***清单 9-4.*** Johnson算法

from copy import deepcopy

def johnson(G): # 所有节点对的最短路径

G = deepcopy(G) # 以免破坏原始节点

s = object() # 确定未被使用的节点

G[s] = {v:0 for v in G} # 从s出发的边权值为零

h, \_ = bellman\_ford(G, s) # h[v]: 从s出发的最短距离

del G[s] # 不再需要s

for u in G: # 从u出发...

for v in G[u]: # ... 到v的边权值 ...

G[u][v] += h[u] - h[v] # ... 边权调整（非负）

D, P = {}, {} # D[u][v]和P[u][v]

for u in G: # 遍历起始节点...

D[u], P[u] = dijkstra(G, u) # ... 找到最短路径

for v in G: # 对每个目标节点 ...

D[u][v] += h[v] - h[u] # ... 重新调整距离

return D, P # 结果为二维

|  |
| --- |
| **请注意：**无需检查bellman\_ford函数调用是否成功和是否找到负环（在有负环的情况下，Johnson算法无效），因为如果图中有负环，bellman\_ford函数会返回异常。 |

假设Dijkstra算法的运行时间为Q(*m* lg *n*)，Johnson算法速度较慢，运行时间是其n倍，即Q(mn lg n)，优于Floyd-Warshall算法三次方的运行时间（我们稍后讨论该算法），适用于稀疏图（即边相对较少的图）。[[7]](#footnote-7)

Johnson算法使用的变换与A \*算法的势函数密切相关（参见本章稍后的“把握未来走向”），它类似于第10章最小费用二分匹配问题使用的变换。二者的目标都是确保边权值为正，但应用场合略有不同（边权值因逐次迭代而发生变化）。

### “牵强”的子问题

Dijkstra算法肯定是基于动态编程的原理，但确定其中子问题的排序（或相互间依赖关系）的需要遮蔽了这一事实的光彩。本节讨论的算法由Roy、Floyd和Warshall三人独立发现，是一个典型的动态编程案例。它基于缓存式递归分解，通常实现过程具有迭代性。它形式貌似简单，但设计极其精巧。从某些方面看，它基于第8章讨论的“非入即出”原理，但初看之下，由此产生的子问题却似乎具有很高的人工化程度，与该原理相去甚远。

在解决许多动态编程问题的过程中，我们都需要寻找一组递归相关的子问题，可一旦发现，这些子问题往往显得十分自然。例如，DAG最短路径中的节点，或者最长公共子序列（LCS）问题的前缀对。后者例证了可以将有用的原理推广到不太明显的结构：对允许处理的元素加以限制。例如，在LCS问题中，我们限制前缀的长度。而背包问题的人工意味更为浓厚：我们对对象进行排序，自行限制处理前*k*个。子问题就被这个“允许对象集”和背包容量参数化了。

在全节点对的最短路径问题中，我们可以使用这种形式的限制，以及“非入即出”原理，*设计*一组隐式子问题：我们随意对节点进行排序，并限制允许用于构成最短路径的中间节点的数量，即前*k*个。这样，我们就使用三个参数，对子问题进行了参数化：

* 起始节点
* 终止节点
* 允许经过的最大节点编号

如果你不清楚这样做的目的，也许会觉得第三个参数没有任何作用——它怎么可能帮助我们对允许完成的工作加以限制？这其中蕴含的思想是*分解*解空间，把问题分解为子问题，再将子问题互相连接，构成子问题图。根据“非入即出”的思想（节点*k*，入还是出？），创建递归的依赖关系，从而完成连接工作。

如果只允许使用前*k*个节点作为中间节点，设从节点*u*到节点*v*的最短路径的长度为*d*(*u*, *v*, *k*)。我们可以分解问题如下：

*d*(*u*, *v*, *k*) = min(*d*(*u*, *v*, *k*−1), *d*(*u*, *k*, *k*−1) + *d*(*k*, *v*, *k*−1))

和背包问题一样，我们要考虑是否包括节点*ķ*。如果不包括它，我们只需使用现有的解决方案，可以找到不使用*k*的最短路径，即*d*(*u*, *v*, *k*−1)。如果包括它，我们必须使用到达*K*的最短路径（即*d*(*u*, *k*, *k*−1)）以及从*k*出来的最短路径（即*d*(*k*, *v*, *k*−1)）。请注意，在这三个子问题中，我们都要处理前*k-1*个节点，因为我们要么排除节点*k，*要么明确地把它作为终点而不是中间节点。这可以保证我们对子问题进行规模排序（即拓扑排序）——无环。

程序清单9-5提供了实现这种思想的算法。（程序使用第8章的缓存装饰器。）请注意，这里假设节点为1到*n*范围内的整数。如果使用其他的节点对象，就需要用列表V以任意顺序存放节点，另外min函数的参数要用V[k-1]和V[k-2]，而不是k和k-1。还要注意的是，返回的D图的形式是D[u,v]，而不是D[u][v]。我还假设这是一个完整的权值矩阵，因此如果从*u*到*v*无边，D[u][v]为无穷大（inf），如果有需要的话，修改这些假定条件是很方便的。

***清单9-5.*** Floyd-Warshall算法的缓存式递归实现

def rec\_floyd\_warshall(G): # 所有的最短路径

@memo # 存储子解

def d(u, v, k): # 从u经1..k到v

if k==0: return G[u][v] # 假设v在G[u]中

return min(d(u, v, k-1), d(u, k, k-1) + d(k, v, k-1)) # 是否使用k?

return {(u, v): d(u, v, len(G)) for u in G for v in G}

# D[u, v] = d(u, v, n)

下面我们来试试迭代的版本。假定有三个子问题参数（*u*、*v*和*k*），我们需要三个for循环来迭代处理所有的子问题。那么，我们似乎有理由认为，要存储所有的子解，就需要按三次方增长的内存开销，但和LCS问题一样，我们可以减少内存开销。[[8]](#footnote-8)递归分解只能将处于*k*阶段的问题与处于*k*−1阶段的问题关联在一起。这就是说，我们只需要*两*个距离图，一个用于本次迭代，一个用于前次迭代。不过，我们还可以有更好的性能表现...

和使用松弛技术的情况一样，此题也在寻找最短路径。处于*k*阶段的问题是“途经节点*k*是否会提供比目前掌握的路径更好的解决方案？”设当前的距离图是D，先前的距离图是C，如下所示：

D[u][v] = min(D[u][v], C[u][k] + C[k][v])

现在想一想，如果全程只用一个距离图，会是什么情况：

D[u][v] = min(D[u][v], D[u][k] + D[k][v])

现在含义略嫌隐晦，有点循环定义的感觉，不过没有问题。我们要找的是最短路径，对吧？D[u][k]和D[k][v]的值就是实际路径的长度（因此是最短距离的上界），所以我们没有耍花样作弊。此外，二者均不大于C[u][k]和C[k][v]，因为我们从未*提高*过图上的权值。因此，只有一种情形可以解释，那就是D[u][v]以更快的速度向正确答案移动，这是毫无疑问的。因此，我们只需要一个二维距离图（即二次而不是三次方内存空间），我们将通过不断更新内存空间寻找最短路径。从很多方面看，这一结果很像是Bellman-Ford算法的二维版本（但并不完全一致，参见程序清单9-6）。

***清单9-6.*** Floyd-Warshall算法，仅考虑距离

def floyd\_warshall(G):

D = deepcopy(G) # 尚无中间节点

for k in G: # 寻找含节点k的最短路径

for u in G:

for v in G:

D[u][v] = min(D[u][v], D[u][k] + D[k][v])

return D

我一开始使用的是这个图的副本作为候选的距离图。这是因为我们尚未尝试通过任何中间节点，所以唯一的可能性是原始权值给出的直连边。还要注意的是，顶点为数字的假设已不复存在，因为我们不再需要对所处阶段进行明确的参数化。只要依据之前的结果，努力为每个可能的中间节点创建最短路径，得到的解决方案都是一样的。由此产生的算法超级简单，但其背后的原理可不简单，但愿你也能赞同这一点。

不过，像Johnson算法那样有个P矩阵也很不错。和众多的动态编程算法一样，构建实际的解决方案可以轻松计算出最优值，只需记录下每次所做的选择。在这种情况下，如果发现一条途经k的最短路径，P[u][v]记录的前导结果就要被替换为P[k][v]，即最短捷径后“半”部分的前导结果。最终算法见程序清单9-7。对于有边相连的所有节点对，原来的P都有对应的前导结果。之后，P随着D的更新而更新。

***清单9-7.*** Floyd-Warshall算法

def floyd\_warshall(G):

D, P = deepcopy(G), {}

for u in G:

for v in G:

if u == v or G[u][v] == inf:

P[u,v] = None

else:

P[u,v] = u

for k in G:

for u in G:

for v in G:

shortcut = D[u][k] + D[k][v]

if shortcut < D[u][v]:

D[u][v] = shortcut

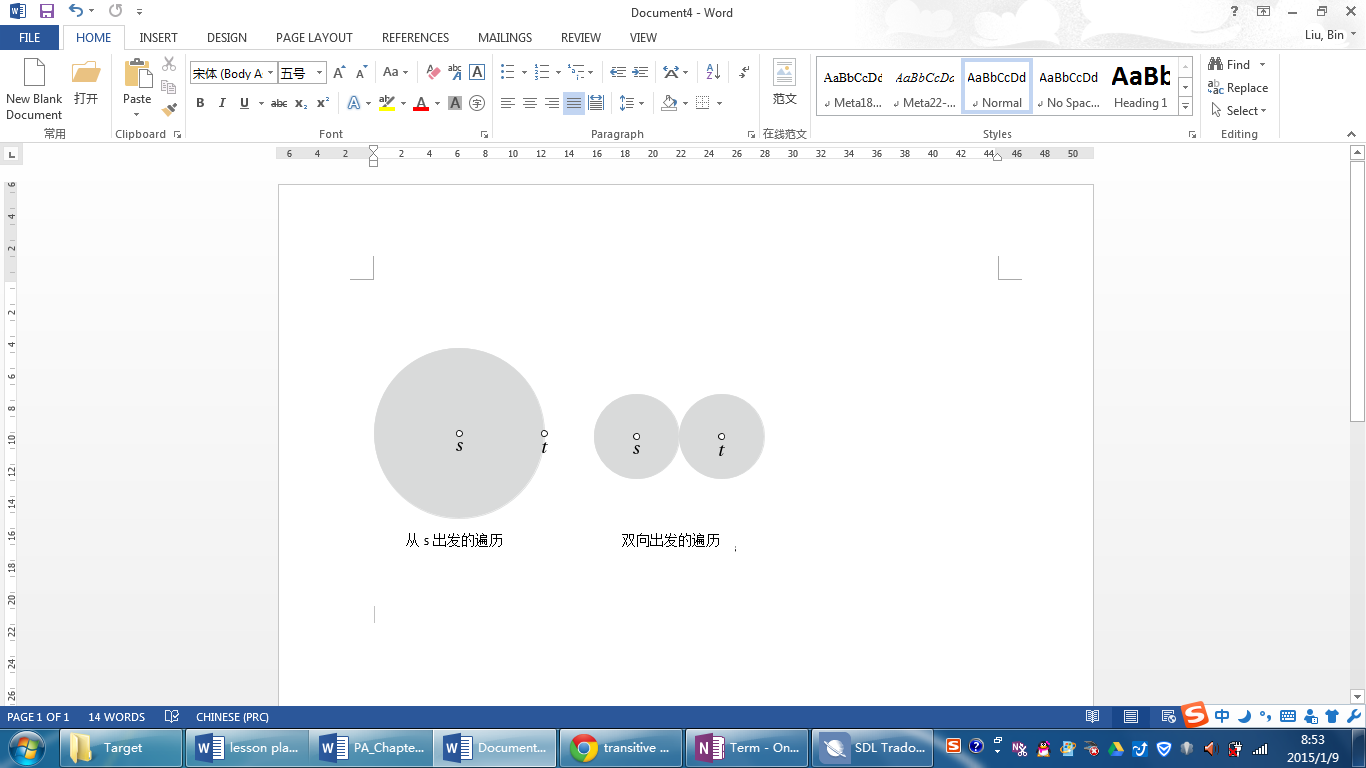
P[u,v] = P[k,v]

return D, P

请注意，在这里使用shortcut < D[u][v]是很重要的，而不是shortcut <= D[u][v]。尽管后者得到的距离也是正确的，但在有些情况下，最后一步会是D[v][v]，这将导致P[u,v] = None。把Floyd-Warshall算法改为计算图的*传递闭包*（即Warshall算法）是很容易的。见练习9-9。

**中途相遇**

Dijkstra算法（还有BFS算法的无权特殊情况）的子问题的解，在图上的向外扩展情形就像是池塘的涟漪。如果最终目标是从A出发到达B，或者使用习惯的节点名称，从*s*到*t*，这意味着“涟漪”必须经过许多你并不真正感兴趣的节点，如图9-4中的左图所示。另一方面，如果一开始就从起点和终点同时出发，展开遍历（假设允许反向遍历边），这样的两组涟漪会在某些情况下中途相遇，从而节省大量工作，如右图所示。



***图 9-4.*** *单向和双向“涟漪”，表示通过遍历找到从s出发的最短路径所需开销*

请注意，虽然图9-4中的“图形化证据”很有说服力，但它肯定不是正规的推论，并未提供任何保证。实际上，虽然本节和下节的算法针对单源单目标的最短路径问题给出了实用的改进办法，但目前还没有任何点到点算法有优于普通单源问题的渐近最坏情况表现。诚然，半径是大圆一半的两个圆只占到其面积的一半，但图的具体表现不一定会像欧几里德平面那样。我们当然希望获得运行时间的改善，不过这就是所谓的*启发式算法*。这种算法基于学术性猜测，往往受到先验主义的评价。我们可以肯定的是，它的渐近性性能表现不会比Dijkstra算法差，改进实际运行时间才是关键。

为实现这种Dijkstra算法的双向图版本，我们先对原版稍作修改，把它变成一个子解生成器，让我们可以提取尽可能多的子解用来“相会”。这类似于第5章介绍的一些遍历函数，如iter\_dfs（程序清单5-5）。这种迭代行为意味着我们可以彻底抛开距离表，仅仅依靠优先级队列中保存的距离值。为简单起见，这里没有引入前导节点的信息，但我们很容易就可以通过向堆元组中添加前导节点来扩展解决方案。要获得距离表（像原版Dijkstra算法那样），只需调用字典dict(idijkstra(G, s))。

代码参见程序清单9-8。

***清单 9-8.*** Dijkstra算法作为解决方案生成器的实现

def idijkstra(G, s):

Q, S = [(0,s)], set() # 已访问距离的队列

while Q: # 尚无处理的节点？

d, u = heappop(Q) # 距离值估计最小的节点

if u in S: continue # 节点已访问？跳过

S.add(u) # 节点现已访问

yield u, d # 产生子解/节点

for v in G[u]: # 遍历其所有相邻节点

heappush(Q, (d+G[u][v], v)) # 添加到队列，距离值估计作为优先级

请注意，我并未使用松弛技术，它隐含在堆中。或者可以这么说，heappush就是新的松弛技术。再次添加一个有更好距离值估计的节点意味着它的优先级高于原有条目，这相当于用松弛操作覆盖原有条目。这类似于第7章介绍的Prim算法的实现。

既然我们已经可以分步实现Dijkstra算法，构建一个双向的版本就不会太难了。

我们在原有算法的入节点和出节点实例之间来回往复，各组涟漪一次扩展一个节点。假如一直进行到底，会得到两组完整的答案：从*s*到*t*的距离和回溯遍历情况下的从*t*到*s*的距离。当然，这两个答案是相同的，这使得整个活动失去了意义。关键的思想是一旦涟漪相遇，立即停止遍历。一旦idijkstra函数的两个实例产生相同的节点，就跳出循环，这似乎是个不错的设想。

至此，我们遭遇了这个算法唯一真正的精要所在：同时从两个节点*s*和*t*出发进行遍历操作，不断移动到下个距离最近的节点，所以一旦两个算法移动到（即产生）相同的节点，就意味着二者沿最短路径相遇，这个推论是合理的，对吗？毕竟如果只从*s*出发进行遍历，只要到达*t*就终止了（即idijkstra函数调用产生了节点*t*）。遗憾的是，直觉欺骗了我们（至少是我）。图9-5中的简单例子应该可以厘清这种可能的误解。那么最短路径到底在哪里？我们如何确定遍历停止的最佳时机？

（图）

***图 9-5.*** *首次相遇节点（突出显示的节点）并不一定位于最短路径上（最短路径为突出显示的边）。*

事实上，一旦两个实例相遇即结束遍历是没错的。但是，要找到最短路径，我们需要在算法执行的时候保持高度警觉。我们需要维护至今为止发现的最佳距离，每次对一条边(*u*,*v*)进行松弛操作时，我们已经掌握了从*u*到*s*的距离（通过向前遍历），和从*v*到*t*的距离（通过向后遍历），就需要仔细检查，确定将现有路径与边(*u*,*v*)连接起来能否改进我们的最佳解决方案。

事实上，我们可以对终止条件稍作限定（见练习9-10）。不必再等待两个实例访问同一节点，只需查看它们已经走了多远，也就是说，它们目前为止获得的最新距离。最新距离不可能减小，因此，如果二者总和至少和我们目前为止发现的最佳路径相等，那就不可能再找到更好的方案，这就大功告成了。

不过，还是一个疑问挥之不去。之前的推论可能让你相信，继续遍历不可能再找到更好的路径，可如何才能确定没有错过更好的解决方案吗？比方说，我们目前发现的最佳路径长度为*m*。导致遍历终止的两个距离为*l*和*r*，所以我们知道*l + r ≥ m*为终止条件。假定有一条从*s*到*t*的路径比*m*短。为满足这一假定条件，该路径必须包含一条边(*u*,*v*)，使得*d*(*s*,*u*) < *l*且*d*(*v*,*t*) < *r*（见练习9-11）。这意味着，与各自对应的当前节点相比，*u*到*s*的距离更短，*v*到*t*的距离更短，所以两者一定已经被算法访问（产生）过。既然算法产生过这两个节点，我们维护的最佳解决方案到目前为止应该已经发现了这条路径，这就是一组矛盾。换言之，这个算法是正确的。

密切关注到目前为止的最佳路径，这个理念要求我们深入Dijkstra算法的内部。我更青睐idijkstra算法的抽象化，所以我要坚持使用这个算法最简单的版本：一旦发现两组遍历返回相同节点，即终止算法运行，然后查找最佳路径，检查将两组遍历连在一起的所有边。如果你使用的数据集适合双向搜索，这种查找不会存在太大的瓶颈，不过你可以打破限定，自行调整算法。最终的代码见程序清单9-9。Itertools的循环功能提供了一个迭代器，可以不断给出其他迭代器提供的值，而且自始至终不断自行产生数据。在这种情况下，这意味着我们在前向遍历和后向遍历之间不断交替循环。

***清单9-9.*** Dijkstra算法的双向图版本

from itertools import cycle

def bidir\_dijkstra(G, s, t):

Ds, Dt = {}, {} # 分别从s和t出发遍历的D

forw, back = idijkstra(G,s), idijkstra(G,t) # 两个“Dijkstras算法实例”

dirs = (Ds, Dt, forw), (Dt, Ds, back) # 交替往复的情形

try: # 直至前向/后向遍历终止

for D, other, step in cycle(dirs): # 在二者之间切换

v, d = next(step) # 确定节点的下一节点/距离

D[v] = d # 记住距离

if v in other: break # 该节点是否也被另一遍历访问?

except StopIteration: return inf # 其中一组遍历未相遇已完成

m = inf # 二者相遇；然后查找最短路径

for u in Ds: # 对于每个前向遍历访问过的节点

for v in G[u]: # ... 遍历其相邻节点

if not v in Dt: continue # 节点是否也被后向遍历访问？

m = min(m, Ds[u] + G[u][v] + Dt[v]) # 该路径是否优于当前路径？

return m # 返回最佳路径

请注意，此代码假定G为无向图（即所有边均允许有两个方向），并且对于任意节点u满足G[u][u] = 0。你可以轻松扩展算法，不需要用到这些假定条件（练习9-12）。

### 把握未来走向

如前所述，遍历的基本思想是非常灵活的，只要使用不同的队列，就可以获得不少有用的算法。例如，使用FIFO和LIFO队列，可以获得BFS和DFS算法；使用适当的优先级队列，可以获得Prim和Dijkstra算法的核心。本节介绍的A\*算法同样通过调整优先级，扩展Dijkstra算法。

如前所述，A\*算法的思想类似Johnson算法，但目的有所不同。Johnson算法对所有的边权值加以变换，确保边权为正，并且最短路径依然最短。A\*算法以类似的方式对边进行修改，但这次的目标不是保证正边权，我们已经假定边的权值为正了（因为构建A\*算法的基础是Dijkstra算法）。我们的目标是通过使用未来走向信息，引导遍历沿正确方向进行：我们要让远离目标节点的边权值大于那些离目标节点较近的边。

|  |
| --- |
| **请注意：**这类似于第11章介绍的分支定界策略使用的最佳优先搜索。 |

当然，如果真的知道走哪条边可以离目标节点更近，我们就可以用贪婪算法解决问题，直接沿最短路径移动，不用理会其他旁支路线。A \*算法的好处是，它弥补了对未来走向一无所知的Dijkstra算法和假设明确知道未来走向的理想情况之间的差距，引入了一个*潜在势函数*，或*启发式函数h*(*v*)，即我们对剩余距离*d*(*v*,*t*)的最佳猜测。稍后将会看到，Dijkstra算法是A\*的一种特殊情况，即*h*(*v*) = 0。此外，如果我们可以指定*h*(*v*) = *d*(*v*,*t*)，该算法将直接从*s*一路行进到*t*。

那么，其中的具体作用机制是什么？我们定义了修正后的边权，得到套筒式加和，就像Johnson算法那样（虽然应该注意的是，这里的符号有所变化）：*w*’(*u*,*v*) = *w*(*u*,*v*) - *h*(*u*) + *h*(*v*)。套筒式加和确保了最短路径依然最短（同Johnson算法），因为所有的路径长度改变量相同，即*h*(*t*) - *h*(*s*)。如果把启发式函数设为零（或任何常数），权值保持不变。

显而易见，这种调整体现了我们的意图：奖励方向正确的边，惩罚方向不正确的边。我们给各边权增加了*下降势能*（启发式函数），这类似于重力势能的作用机制。如果往一张表面不平的桌上丢一颗鹅卵石，它会沿势能减小的方向运动。在本例中，该算法将沿导致剩余距离下降的方向发展，这正是我们想要的结果。

A\*算法相当于针对修正图的Dijkstra算法，所以如果*h*是可行的，算法就是正确的，即任意节点*u*和*v*满足*w*’(*u*,*v*)非负。按照*D*[*v*] - *h*(*s*) + *h*(*v*) 的递增顺序（而不仅仅是*D*[*v*]）对节点进行遍历。由于*h*(*s*)是一个普通常数，我们可以忽略它，只需将*h*(*v*)添加到现有的优先级队列中。这一加和是我们对从*s*经*v*到*t*的最短路径所做的最佳估计。如果*w*’(*u*,*v*)可行，那么*h*(*v*)也会是*d*(*v*,*t*)的下界（见练习9-14）。

实现所有这一切的一个（常见）方法是使用原始的idijkstra函数，当一个节点入堆时，将*h*(*v*)添加到优先级队列中。原始的距离值估计依然保存在D中。不过，简化的情况可以只使用堆（同idijkstra函数），我们需要对权值进行调整，把边(u,v)的结果减去*h*(*u*)，程序清单9-10就采用了这种方法。如下所示，在返回距离前，去除了多余的*h*(*t*)。 （考虑到a\_star函数所包含的庞大内容，这确实是个短小精悍的程序，你说呢？）

***清单9-10.*** A\*算法

from heapq import heappush, heappop

def a\_star(G, s, t, h):

P, Q = {}, [(h(s), None, s)] # 含下降势能的前导节点和队列

while Q: # 尚未处理的节点？

d, p, u = heappop(Q) # 下降势能最低的节点

if u in P: continue # 节点已访问？跳过

P[u] = p # 设定路径前导节点

if u == t: return d - h(t), P # 已到达！返回距离和前导节点

for v in G[u]: # 遍历所有相邻节点

w = G[u][v] - h(u) + h(v) # 修改含下降势能的边权值

heappush(Q, (d + w, u, v)) # 添加到队列中，以下降势能作为优先级

return inf, None # 未到达t

如上所示，除了新增对u == t的检查，该算法与Dijkstra算法唯一的不同就是权值调整。换言之，只要你愿意，也可以使用Dijkstra算法的点到点版本（即包含u == t检查的版本），对权值修改过的图进行操作，而不必单独使用A \*算法。

当然，为了从A\*算法获益，我们需要有很好的启发式函数。函数的内容在很大程度上取决于要解决的具体问题。例如，在一张道路交通图上查找路线，从某个节点到目的地的直线欧氏距离必须是一个有效的启发式函数（下界值）。这个函数事实上适用于任意平面运动，例如计算机游戏里的怪物行动轨迹。但是，如果存在大量的死胡同和迂回曲折，这个下界值可能不会非常准确。（替代方法请参阅“如果你希望了解更多......”一节）。

A \*算法也用于搜索解空间，我们可以视之为抽象图（或隐式图）。例如，我们也许需要解决魔方问题或者刘易斯·卡罗尔（Lewis Carroll）所谓的*单词梯*问题。我们在此不妨讨论一下后一个问题。

单词梯是从一个起始单词开始构建的，比如lead，以另一个单词作为结束，比如gold。单词梯的每一步搭建都要用到实际单词。从一个单词推进到下一单词，只能更换一个字母。（单词梯还有其他的版本，增删字母或字母交换位置。）因此，本题的一种解法是，通过load和goad这两个单词，从lead到达gold。如果把每个单词都看做是图中的一个节点，我们可以为彼此相差一个字母的所有单词之间加上边。我们可能没必要真的建立这样一个结构，但不妨“假设”一下，如程序清单9-11所示。

***清单 9-11.*** 单词梯路径的隐式图

from string import ascii\_lowercase as chars

class WordSpace: # 含功能封装函数的隐式图

def \_\_init\_\_(self, words): # 以单词作为节点创建图

self.words = words

self.M = M = dict() # 可达单词

def variants(self, wd, words): # 产生所有单词变体

wasl = list(wd) # 将单词作为列表

for i, c in enumerate(wasl): # 遍历位置和字符

for oc in chars: # 所有可能的字符

if c == oc: continue # 不用相同的字符替换

wasl[i] = oc # 替换字符

ow = ''.join(wasl) # 排出字符串作为候选单词

if ow in words: # 是否为有效单词？

yield ow # 生成单词

wasl[i] = c # 重置字符

def \_\_getitem\_\_(self, wd): # 毗邻关系图界面

if wd not in self.M: # 缓存相邻节点

self.M[wd] = dict.fromkeys(self.variants(wd, self.words), 1)

return self.M[wd]

def heuristic(self, u, v): # 默认的启发式函数

return sum(a!=b for a, b in zip(u, v)) # 有多少个字符不同？

def ladder(self, s, t, h=None): # a\_star的功能封装函数

if h is None: # 允许存在其他启发式函数

def h(v):

return self.heuristic(v, t)

\_, P = a\_star(self, s, t, h) # 获得前导节点图

if P is None:

return [s, None, t] # 如无最佳路径存在

u, p = t, []

while u is not None: # 从t开始向后退

p.append(u) # 附上每个前导节点

u = P[u] # 再走一步

p.reverse() # 该路径为向后遍历

return p

WordSpace类的主要思想是，它可以作为加权图使用，这样可以配合a\_star一起工作。如果G是WordSpace对象，G['lead']就是字典，其他单词（如“load”和“mead”）是键值，各边权值为1。这里使用的默认启发式函数只是统计单词不同的字符位数。

只要有某种形式的单词列表，使用WordSpace类是很容易的。许多UNIX系统都有一个名为/usr/share/dict/words或/usr/dict/words的文件，文件里每行一个单词。如果没有这样的文件，你可以从<http://ftp.gnu.org/gnu/aspell/dict/en>下载，还可以在网上找到（或类似文件）。然后，就可以构建如下所示的WordSpace（去掉空格，所有字符统一为小写）：

>>> words = set(line.strip().lower() for line in open("/usr/share/dict/words"))

>>> G = WordSpace(words)

如果你对得到的单词梯不满意，当然可以随意去掉其中一些单词。构建好WordSpace后，就可以展开正式工作了：[[9]](#footnote-9)

>>> G.ladder('lead', 'gold')

['lead', 'load', 'goad', 'gold']

相当清楚，但也许还不够精彩。那么让我们试试下面这个函数：

>>> G.ladder('lead', 'gold', h=lambda v: 0)

我只是用毫无信息量的部分替代了启发式函数，等于是把A \*算法变成了BFS算法（或者说，在无权图上运行Dijkstra算法）。在我的电脑上（并且用我的单词列表），运行时间的差异是相当明显的。实际上，使用第一个（默认）启发式函数加速了近100倍！[[10]](#footnote-10)

### 本章小结

本章比前几章的关注点更为集中，主要介绍在网络状结构和空间中寻找最佳路线，换言之，求解图的最短路径问题。本章算法使用的一些基本思路和机制，在本书先前章节已有所涉及，这样可以帮助我们逐步构建解决方案。所有最短路径算法寻找最短路径共同使用的基本策略是，要么使用relax函数或等效方法确定路径上新的可能的下个或最后一个节点（大部分算法的做法），要么考虑一条包含两条子路径的最段路径，以某个中间节点作为出节点或入节点（Floyd-Warshall策略）。基于松弛的算法解题方式有所不同，具体情况由对图所作的假设而定。Bellman-Ford算法只是通过依次遍历每一条边构建最短路径，最多重复n-1次迭代过程（如果还有改进可能，就报告负环）。

在第8章可以看到，比Bellman-Ford算法更高效是有可能的；对于DAG图而言，只要按照拓扑排序访问节点，就可以对每边只做一次松弛操作。拓扑排序不适用于一般的图，但如果规定边权非负，我们可以找到对重要的边加以尊重的拓扑排序方法，即按照节点到起始节点的距离排序。当然，我们一开始并不知道排序结果，但可以通过在剩余节点中始终选取距离值估计最小的节点来逐步明确，就像Dijkstra算法做的那样。这么做是很有必要的，因为我们已经对所有可能的前导节点的出边都做了松弛操作，所以根据排序结果，下个节点的路径估计一定是正确的，这样的节点也只有一个，它拥有最小上界。要查找所有节点对之间的距离，我们有几种选择。例如，我们可以对每个可能的起始节点运行Dijkstra算法。对于稀疏程度高的图，这么做效果相当不错，而且事实上，即使有的边权为负，我们也可以用这个方法！我们首先运行Bellman-Ford算法，再调整所有的边权，以满足（1）路径的长度等级不变（最短路径依然最短）（2）边权为正。另一个选择是使用动态编程，就像Floyd-Warshall算法那样，每个子问题都是由起始节点、目标节点和最短路径允许通过的其他节点（以某种预先确定的顺序）共同定义的。

要找到从一个节点到另一个节点的最短路径，再没有比找到从起始节点到其他所有节点最短路径更好的渐近方法了。尽管如此，还是有一些启发式方法可以获得实际的改善。其中之一就是*双向*搜索，“同时”从起始节点和目标节点出发进行遍历，一旦二者中途相遇即终止遍历，从而减少了需要访问的节点数（至少我们是这样希望的）。另一种方法是使用启发式“最佳优先”方法，用启发式函数引导我们暂时抛开希望不大的节点，优先移动到更有希望的节点，就像A \*算法做的那样。

### 如果您感兴趣......

大多数算法书都会介绍寻找最短路径的基本算法。不过，有些比较高级的启发式算法，如A\*算法，通常可以在人工智能方面的书中找到。在这类书中，你还可以找到使用这种算法（及其他相关算法）搜索复杂解空间的详细说明，那些例子看起来一点也不像本章讨论的显式图结构。要充分了解人工智能的相关内容，我衷心推荐Russell和Norvig的书。关于A\*算法的启发式思想，你可以在网上搜索关键词“shortest path（最短路径）”和“landmarks（地标）”或“ALT（基于地标和三角不等式的A\*算法搜索）”。

如果你想考察Dijkstra算法的渐近性，可以研究一下斐波那契堆。如果用斐波那契堆替换二叉堆，Dijkstra算法的渐近运行时间将得到改善，但性能表现仍有可能受到影响，除非你用的实例非常大，因为Python的堆实现速度是非常快的，而Python的斐波那契堆实现（情况相当复杂）有可能很慢。不过，值得一试。

最后，你也许想把Dijkstra算法的双向版本和A \*算法的启发式机制结合在一起。不过，在行动之前，你应该先研究一下，有些陷阱可能会导致你设计的算法无效。关于这一点和使用基于地标的启发式算法（以及随时间变化的图所带来的挑战）的更多信息（略高级）请参见Nannicini等人的论文。（见“参考文献”）。

### 练习题

1. 在某些情况下，由于不同货币汇率之间的差异，人们可以将一种货币换成另一种，反复兑换直到获利，再换回到原始货币类型。如何使用Bellman-Ford算法检测获利情况的出现？
2. 如果一个以上的节点到起始节点的距离相同，运行Dijkstra算法会发生什么？结果是否仍然正确？
3. 如图9-3所示，使用虚拟节点表示边长，有什么坏处？
4. 如果用无序列表而不是二叉堆来实现Dijkstra算法，运行时间有何变化？
5. 为什么可以肯定，Johnson算法调整后的边权都是非负值？有出错的可能吗？
6. Johnson算法的*h*函数基于Bellman-Ford算法设计。在套筒式求和中这个函数会被消去，那为什么不能使用其他函数？
7. 实现Floyd-Warshall算法的缓存式版本，保存记忆的方法与迭代法相同。
8. 扩展Floyd-Warshall算法的缓存式版本，让其可以像迭代法那样计算P表。
9. 如何对Floyd-Warshall算法进行修改，让它可以检测到路径的*存在*，而不是找出*最短*的路径（即Warshall算法）？
10. Dijkstra算法双向版本较为严格的终止标准是正确的，为什么能说明原始版本就是正确的？
11. 为证明Dijkstra算法双向版本是正确的，我提出了一条假设路径，要优于到目前为止发现的最佳路径，该路径必须包含一条边(*u*,*v*)，使得*d*(*s*,*u*) < *l*且*d*(*v*,*t*) < *r*。为什么会有这样的要求？
12. 改写bidir\_dijkstra函数，令其不需要输入图是对称的且拥有权值为零的自反边（或循环）。
13. 实现双向版本的BFS算法。
14. 如果*w*’可行，为什么说*h*(*v*)是*d*(*v*,*t*)的下界？

### 参考资料

* Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269-271.
* Nannicini, G., Delling, D., Liberti, L., and Schultes, D. (2008). Bidirectional A\* search for time-dependent fast paths. In *Proceedings of the 7th international conference on Experimental algorithms*, Lecture Notes in Computer Science, pages 334-346.
* Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*, third edition. Prentice Hall.

1. 第11章会回顾“瑞典之旅”的寻路问题。 [↑](#footnote-ref-1)
2. 即最小费用二分匹配问题，将在第10章探讨。 [↑](#footnote-ref-2)
3. 这里假定节点的距离各不相同。如果出现距离相同的情况，候选节点就不止一个了。练习9-2要求对这种情况进行讨论。 [↑](#footnote-ref-3)
4. 你也许会注意到，出于简化代码的目的，回到S的边也都接受过松弛操作。这不影响正确性和渐近运行时间，不过你可以修改代码，跳过这些节点。 [↑](#footnote-ref-4)
5. Dijkstra算法比较常规的做法是每个节点只能添加一次，对应的距离值估计在堆中更新，也就是说，如果发现更好的距离值估计，旧的结果被覆盖，对旧的路径不再理会。 [↑](#footnote-ref-5)
6. 节点s的创建采用了object对象实例化的方法，每个这样的实例都是独一无二的（也就是说，它们并非==意义上的相等），因此适用于添加虚拟节点以及需要区别于所有合法值的其他形式的哨兵对象。 [↑](#footnote-ref-6)
7. 判定一个图为稀疏图的常见标准是*m* = *O*(*n*)。但是在本例中，Johnson算法的性能在*m* = *O*(*n*2/lg *n*)的情况下（适用边数较多的情形）可与Floyd-Warshall算法匹敌（渐近）。另一方面，Floyd-Warshall算法的恒定开销很小。 [↑](#footnote-ref-7)
8. 在缓存型版本的算法中，也同样可以节省内存开销。参见练习 9-7. [↑](#footnote-ref-8)
9. 比方说，我用的是有关炼金术的单词，我去掉了像*algedo*和*dola*这样的单词。 [↑](#footnote-ref-9)
10. 是整100，而不是100的因子。（当然也不是100的因子的11次方。） [↑](#footnote-ref-10)