第 3 篇

计算机是如何找到最优路径的

艾博士导读

最优路径问题是人工智能研究中的一个重要问题，很多问题都可以转化为最优路径求解问题，如语音识别、汉字识别后处理、拼音输入法等。

A* 算法是求解最优路径的一个重要算法，在人工智能中具有重要地位，曾经被列为计算机领域最重要的 32 个算法之一。

那么都有哪些最优路径搜索算法呢？这些算法各自的特点是什么？每种算法是否可以找到最优路径？或者在什么条件下可以找到最优路径？本篇将逐一介绍，解开这些谜团。

秋天到了，正是看红叶的季节。北京香山的红叶最为著名，小明计划周末和同学们一起骑车去香山看红叶，线路图如图 3.1 所示。小明虽然以前多次和父母去过香山，但是每次都是爸爸开车去的，小明并不熟悉如何到达香山。不过这也难不倒小明，手机上有导航软件，输入目的地香山后，很容易就可以找到到达香山的路线。现在的导航软件做的是真好，可以提供好几条路线供选择：有距离最短，有花费时间最短，还有经过的红绿灯最少等，给出的是不同条件下的最优路径。这引起了小明强烈的好奇心：计算机是如何找到最优路径的呢？

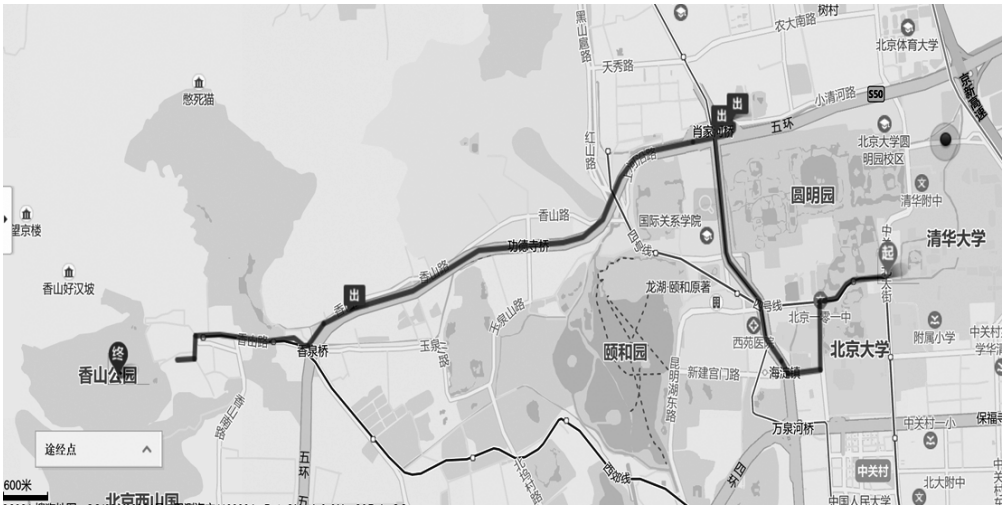

图 3.1　从清华大学到香山公园路线

游玩香山回来之后，带着这个问题，小明又来找艾博士请教。

3.1 路径搜索问题

明白了小明的来意之后，艾博士从书柜里找出了一张很久未用的纸质地图。

艾博士指着地图对小明说：现在真是太方便了，无论想去哪里，导航软件都能很快给出路径。以前我们都是依靠这种地图，在地图上查找半天，才能找到一条合适的路线。

艾博士让小明尝试着找到一条去香山的路。小明由于是第一次使用这种纸质地图，在地图上探索半天，才好不容易找到了一条到达香山的路线。

艾博士问小明：小明，你刚才是怎么找到去香山的路线的呢？

小明回答说：刚开始我有些不得要领，完全是无规律地乱找。后来我发现主要是找路口，重要的是在哪个路口应该向哪个方向走，因为相邻的两个路口之间只有一条路，是不需要考虑如何走的。

艾博士夸奖道：小明你真聪明！其实所谓的路线，就是将经过的路口——包括道路的入口和出口——连接在一起。所以找路线，重要的就是找到这些必须经过的路口。

为此，我们可以将路口看作是一个状态，相邻的路口用称作“边”的连线连接在一起，这样地图就可以用这种状态连接图表示了。如图 3.2 所示，就表示了一个简单的地图，图中 A、B、C 等表示的是路口，两个相邻路口用边连接在一起。边旁边的数字表示两个路口之间的距离。比如 S 到 A 的距离为 6。这里的状态，在图中又称作是节点。

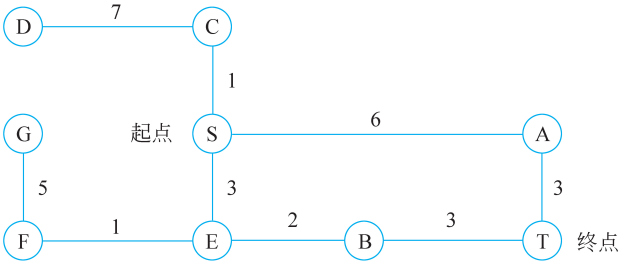

图 3.2 状态连接图示意

艾博士总结说：在地图上找出从起始地点 S 到目标地点 T 的路线，就是在这样的状态连接图上寻找出几个状态，这些状态连接在一起，可以从起始点 S 到达目标地点 T。这就是路径搜索问题。如果找到的路径满足一定的最优条件，则是最优路径搜索问题。由于这种搜索是在状态连接图上进行的，所以又可以称为状态搜索问题。

小明着急地问道：艾博士，那么如何通过状态连接图搜索到路径呢？

艾博士：小明，我们先不着急介绍具体的搜索算法，先来看看这里的关键问题是什么。以图 3.2 为例，假设 S 是所在的起点，T 是要去的终点。从 S 出发可以到达 A、C、E，我们用图 3.3 所示的搜索图表示。图中由于 A、C、E 3 个节点是从 S 生成出来的，所以这 3 个节点称作 S 的子节点，S 称作这几个节点的父节点。父节点产生子节点的过程称作扩展。

由于连接 S 的只有 A、C、E 这 3 个路口，所以要到达目标 T 的话，必须经过这 3 个路口中的一个，那么下一步应该选择哪个节点扩展呢？假设选择了节点 E 扩展，则生成子节点 B 和 F，得到的搜索树如图 3.4 所示。

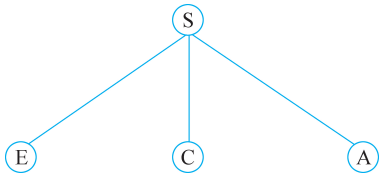

图 3.3　从 S 扩展出 3 个子节点 A、C、E

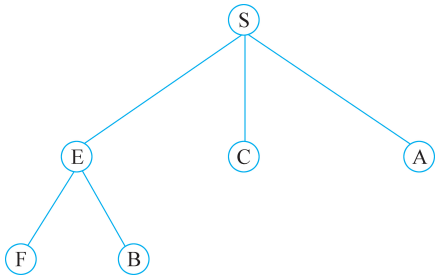

图 3.4　扩展节点 E 生成出子节点 B、F

到这一步之后，又面临选择哪个节点扩展的问题，从可能经过的 A、B、C、F 4 个路口中选择一个节点扩展。

讲到这里，艾博士问小明：小明你说说看，这里的关键问题是什么？

小明想了想回答道：这里的关键问题应该是如何选择哪个节点优先扩展。

听了小明的回答，艾博士非常高兴：小明，你说得很对。如何选择节点优先扩展是状态搜索问题的关键所在。事实上，不同的选择方法，就构成了不同的搜索算法。不同的算法具有不同的性质，下面我们分别介绍几个常用的方法。

另外我们需要强调的是，通过状态搜索不只可以求解路径问题，其他很多问题也都可以转化为状态搜索问题进行求解。比如八数码问题。

小明不解地问道：八数码问题是个什么问题呢？

艾博士解释说：八数码问题是一个智力游戏问题。在 3×3 组成的九宫格棋盘上，摆有 8 个将牌，每一个将牌都刻有 1～8 数码中的某一个数码。棋盘中留有一个空格，允许其周围的某一个将牌向空格移动，这样通过移动将牌就可以不断改变将牌的布局。这种游戏求解的问题是：给定一种初始的将牌布局（初始状态）和一个目标布局（目标状态），问如何通过移动将牌，实现从初始状态到达目标状态的转变。图 3.5 给出了一个八数码问题的示意图。

小明：我明白什么是八数码问题了，但是这与路径搜索问题有什么关系呢？

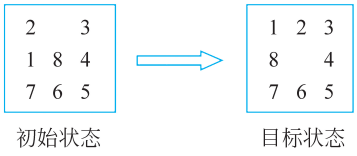

图 3.5　八数码问题示例

艾博士解释说：这个问题实际上跟路径搜索问题是一样的。初始状态可以认为是出发点，在这个状态下走一个将牌形成的新状态可以看作是与初始状态相邻的“路口”。八数码问题的解就是找到若干个相邻的“路口”（状态），可以从初始状态一步步达到目标状态。

小明恍然大悟道：经您这么一解释，八数码问题还真是和路径搜索问题是一样的。

艾博士：还有定理证明问题，从搜索的角度来说，也可以认为是寻找路径的过程。

小明：定理证明也跟路径搜索问题有关？

艾博士：你学过几何吧？几何证明题一般是怎么描述的？又是怎么证明的？

小明想了想回答说：一般都是先给几个已知条件，然后要求证明某个结论，比如证明两条直线平行、两个角相等等。证明过程一般是用某个定理，从已知条件推理出某个结论，然后再反复运用已知的定理得出更多的结论，直到最终得出要求证明的结论。

艾博士：这里的已知条件就相当于路径搜索问题中的起始状态，而要证明的结论就相

当于目标状态。运用定理推导出的一些中间结果可以认为是搜索过程中出现的中间状态，从一个状态推导出另一个状态所用的定理，就相当于连接两个状态的边。所以从搜索的角度描述定理证明的话，就是找到一条从初始条件出发，通过一系列定理连接的、到达要证明的结论的路径。

小明认真思考后说：仔细想想还真是这么一回事。

艾博士：很多问题都可以转化为路径搜索问题，在下面的讲解中，除非特殊说明，我们均以状态连接图上的路径搜索为例做介绍。

小明读书笔记

路径搜索问题就是在一个状态图上，如何选择被扩展的节点，不同的选择方法就构成了不同的搜索算法。

路径搜索方法不仅适用于求解地图上查找路线问题，这里的“路径”是广义的路径概念，很多问题可以转化为路径搜索问题来求解。

3.2 宽度优先搜索算法

艾博士：小明，我们首先从宽度优先算法开始介绍。所谓宽度优先算法，就是选择节点深度最浅的节点优先扩展。

小明：什么是节点深度呢？

艾博士解释说：在搜索图中，第一个节点称作根节点，根节点的深度为0，其他节点的深度为其父节点的深度加1。简单地说，根节点深度为0，根节点的子节点深度为1，再下一层子节点深度为2，……

艾博士：小明，你说说看，图3.4中几个节点的深度分别多少？

小明回答说：S是根节点，深度为0；A、C、E 3个节点均为S的子节点，所以它们的深度均为1；B、F均为E的子节点，节点深度均为2。

艾博士接着说：宽度优先搜索就是从根节点开始，每次选择一个节点深度最浅的节点扩展，直到生成出目标节点为止。

小明问道：艾博士，如果深度最浅的节点存在多个时如何选择呢？

艾博士：这种情况下可以随机选择一个深度最浅的节点进行扩展。

下面我们以图3.2为例介绍如何采用宽度优先搜索求解从起点S到终点T的路径。图3.6给出了该问题的搜索图，其中红圈数字表示节点被扩展的次序，这里假定当深度一样时，排在左边的节点优先扩展。

该搜索过程首先选择深度最浅的根节点S进行扩展，产生了3个子节点A、C、E；由于这时这3个节点的深度都是1，我们根据假定优先扩展左边的节点E，产生节点B、F；这时A、C的节点深度最浅，我们选择C扩展，产生节点D；这时A的深度最浅，扩展后产生节点T。而这时T就是终点节点，搜索结束，得到路径S-A-T。

这个问题比较简单，很快就找到了达到终点也就是目标状态的路径。对于复杂一些的情况，需要继续寻找节点深度最浅的节点扩展，直到找到目标为止。

小明问艾博士：宽度优先搜索如何求解其他问题呢？比如前面提到过的八数码问题。

艾博士：当然可以求解八数码问题，方法是一样的。对于图 3.7 所示的八数码问题，图 3.8 给出了用宽度优先搜索求解该八数码问题的搜索图，其中红圈中的数字表示的是该节点被扩展的次序。

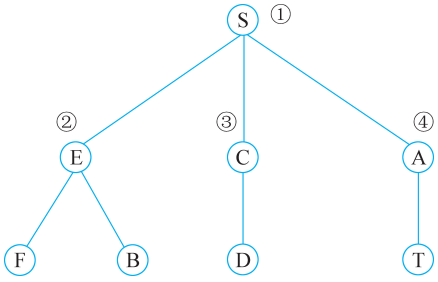

图 3.6　宽度优先搜索求解示意图

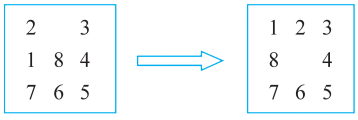

图 3.7　八数码问题

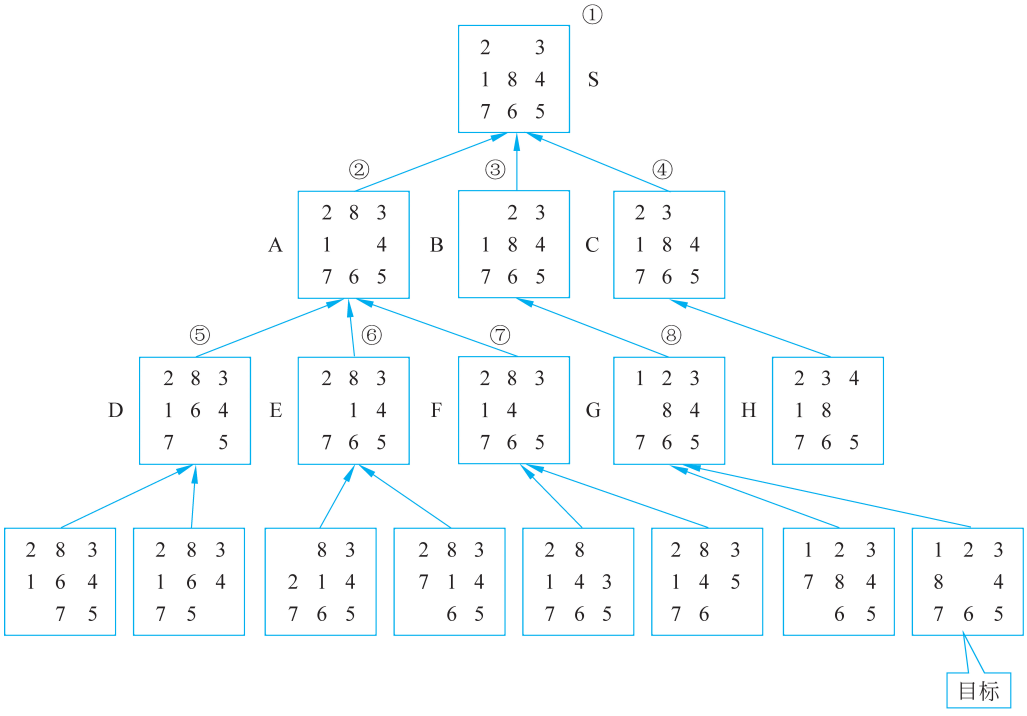

图 3.8　宽度优先搜索求解八数码问题搜索图

艾博士对照着图 3.8 给出的八数码问题搜索图说：我们来看看用宽度优先搜索求解八数码问题的搜索过程。开始只有初始节点 S，对 S 进行扩展，生成 A、B、C 3 个子节点。由于 A、B、C 3 个节点的深度是一样的，深度均为 1，按照约定选择最左边的节点 A 扩展，生成 D、E、F 3 个子节点。这时节点 B、C 的深度最浅，同样选择排在左边的节点 B 扩展，生成出子节点 G。再选择深度最浅的节点 C 扩展，生成出子节点 H。此时深度最浅的节点有 D、E、F、G、H 5 个节点，深度均为 2。同样依次扩展节点 D、E、F、G，当扩展到节点 G 时，其子节点中出现了目标节点，搜索到此结束。通过搜索图，可以得到该八数码问题的解，也就是为达到目标状态将牌的移动方法：将牌 2 右移，将牌 1 上移，将牌 8 左移。

小明听着艾博士的讲解，有些不太明白地问道：艾博士，这种方法为什么叫宽度优先搜索呢？

艾博士解释说：小明你看图 3.8 中所示的节点扩展次序，是沿着“横向”进行的，先优先扩展第一层节点，再扩展第二层、第三层……这就是宽度优先名称的由来。

小明：明白了。那么宽度优先搜索有什么特点呢？

艾博士：宽度优先搜索有一个重要的结论，就是在单位代价下，当问题有解时，可以找到问题的最优解，也就是路径总代价最小的解。

小明：单位代价是什么意思呢？

艾博士：我们先解释一下什么是代价。所谓代价就是父节点到子节点的广义“距离”。比如从路口 A 到路口 B 距离多少千米可以是代价，需要用多长时间也是代价，花费多少钱也是代价。而单位代价指的是任何两个父子节点间的代价总是为 1。比如在上面的八数码问题中，如果移动一个将牌的代价为 1 的话，则该问题就是单位代价的。在单位代价下，我们用宽度优先搜索得到的八数码问题的解，就是总代价最小的，也就是将牌的移动次数最少的解。

对于地图上求两个地点间的一条路径，如果认为两个相邻路口的代价为 1，则找到的是经过的路口最少的路径。当每个路口都有红绿灯时，实际上就是经过的红绿灯最少的路径。在城市中，寻找一条红绿灯最少的路径也是很有意义的。

小明问：为什么在单位代价下宽度优先搜索得到的就是代价最小的路径呢？

艾博士解释说：如同前面说过的，宽度优先搜索在搜索图上体现的是“横着”走，先扩展深度为 1 的节点，再扩展深度为 2 的节点……逐步加深扩展节点的深度。假设从初始节点到目标节点存在不同的路径，通过这些路径到达目标节点的深度也不相同。宽度优先搜索优先选择深度最浅的节点扩展，所以当第一次出现目标节点时，一定是经过这条路径计算的目标节点深度最浅。在单位代价下，节点深度就是初始节点到目标节点的总代价，所以宽度优先搜索可以找到总代价最小的路径。

小明：经您这么一解释就明白了。在单位代价条件下，从初始节点到达一个节点的总代价等于该节点所处的深度，宽度优先搜索算法的思想是选择深度最浅的节点扩展，也就是选择总代价最小的节点优先扩展，所以当第一次遇到目标节点时，一定就找到了到达目标节点的最短路径。

小明读书笔记

宽度优先搜索算法是一种常用的路径搜索算法，其特点是每次选择节点深度最浅的节点优先扩展。当问题是单位代价时，也就是任何相邻节点之间的代价均为 1 时，可以找到从初始节点到目标节点代价最小的最优路径。

3.3 迪杰斯特拉算法

小明：在单位代价下，宽度优先搜索算法可以找到代价最小的路径，但是很多问题并不是单位代价的。比如说对于八数码问题，如果移动将牌的代价为将牌的数码，如数码为 5 的将牌移动一次的代价为 5，每个将牌的数码不一样，移动的代价也不相同。再比如，如果步

行或者骑车去某个地方，相对于红绿灯最少来说，可能距离最短更重要。即便是开车，距离也是一个重要的影响因素。那么是否有办法求解一般情况下总代价最小路径的方法呢？

艾博士反问小明：小明，宽度优先搜索是如何选择被扩展节点的？

小明马上回答说：优先选择深度最浅的节点扩展。

艾博士：小明回答得很好。如果我们用 $g(n)$ 表示从初始节点到节点 n 的一条路径的代价，节点 n 的深度就相当于单位代价下的 $g(n)$。所以，宽度优先搜索优先选择深度浅的节点，就相当于单位代价条件下优先选择 $g(n)$ 小的节点扩展。我们修改一下宽度优先搜索算法，优先选择 $g(n)$ 小的节点扩展，宽度优先搜索算法就变成了迪杰斯特拉算法。但是需要修改一下结束条件：当目标节点的 $g(n)$ 值最小时，算法才结束，而不是只要目标一出现就马上结束。

小明问：为什么要加上这样的条件呢？

艾博士：这个条件很关键，我们后面再讲为什么要有这样的条件。

我们还是通过图 3.2 的例子介绍迪杰斯特拉算法，为了看起来方便，我们将图 3.2 的状态连接图重画在图 3.9 中，图 3.10 给出了该问题的搜索图。同样，红圈里的数字表示节点被扩展的顺序，连接线旁边的数字表示的是两个节点间的距离。

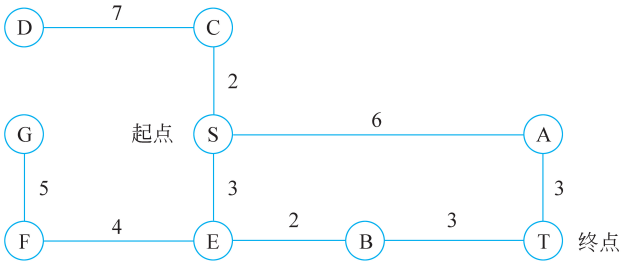

图 3.9　状态连接图示意

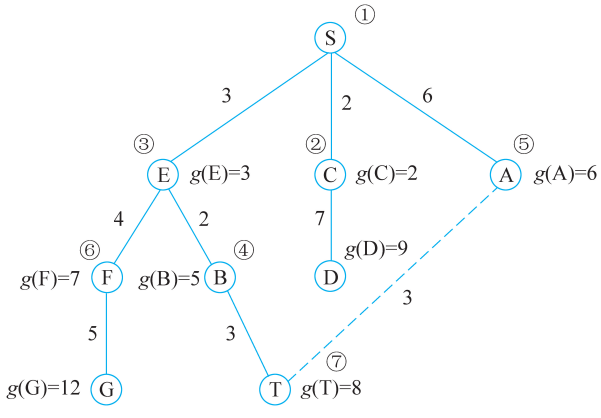

图 3.10　迪杰斯特拉算法搜索示意图

在图 3.10 中，首先扩展初始节点 S，生成出节点 A、E、C，按照 S 到这 3 个节点的距离，分别得到：

$$g(\mathrm{A})=6$$

$$g(\mathrm{C})=2$$

$$g(E)=3$$

由于 $g(C)$最小，所以第二次选择 C 扩展，生成出节点 D，并得到 $g(D)=g(C)+7=9$。

接下来从 A、D、E 中选择一个 g 值最小的节点，$g(E)=3$ 被选中第三个扩展，产生节点 B、F，分别计算出：

$$g(B)=5$$

$$g(F)=7$$

在 A、D、B、F 几个节点中，$g(B)=5$ 最小，成为第四个被选择扩展的节点，生成出节点 T，经计算有：

$$g(T)=8$$

这时虽然已经找到一条从初始节点 S 到目标节点 T 的路径 S-E-B-T，但是由于 $g(T)$ 并不是最小的($g(A)=6$、$g(F)=7$，均小于 $g(T)=8$)，所以算法并不停止，继续选择 g 值最小的节点扩展。

接下来第五个被选择扩展的节点是 A，再一次生成出节点 T。这时找到了两条到达 T 的路径，一条是之前找到的 S-E-B-T，另一条是刚找到的 S-A-T。这种情况下，需要做一次选择，保留代价最小的路径，由于通过路径 S-E-B-T 计算 $g(T)$为 8，而通过路径 S-A-T 计算 $g(T)$为 9，所以保留前者，而忽略后者，图中用虚线表示。

这时 $g(F)$又成为了最小的，所以第六次选择节点 F 扩展，生成出节点 G，并计算 g 值为：

$$g(G)=12$$

这时，已经生成但还未被扩展的节点有 F、G、T，3 个节点中 T 的 g 值最小，而 T 又是目标节点，算法结束。

这样就找到了从初始节点 S 到达目标节点 T 的路径 S-E-B-T。

小明： 艾博士，迪杰斯特拉算法每次选择 g 值最小的节点扩展，当问题是单位代价时，与宽度优选搜索算法是等价的。那么在一般情况下，迪杰斯特拉算法一定能够找到最小代价的路径吗？

艾博士： 可以证明，迪杰斯特拉算法可以找到代价最小的路径。即便不满足单位代价条件，找到的也一定是代价最小的路径。

但是一定要记住前面我们提到过的迪杰斯特拉算法的结束条件，当目标节点的 g 值最小时，算法才结束。这是迪杰斯特拉算法能找到最小代价路径的一个重要条件。因为迪杰斯特拉算法总是从搜索图的叶节点(即搜索图中已经产生但是还没有被扩展的节点)中选择一个节点扩展，当目标节点的 g 值最小时，其他节点还没有到达目标节点，其 g 值就已经比目标节点的 g 值大了，所以通过其他节点到达目标节点的路径代价肯定不会比目前得到的这条路径小，从而找到的一定是代价最小的路径。不过这里也需要做一个补充说明，即代价都是大于 0 的，不存在负的代价。

小明还是有些不太明白地问道：在上面的例子中，最初就找到了路径 S-E-B-T 为什么不能马上结束呢？最终找到的最短路径也是这一条路径啊。

艾博士解释说：小明你看图 3.10 所示的搜索图，图中虚线部分的代价如果不是 3 而是 1，会出现什么情况？这时从 S 到 T 的最短路径应该是 S-A-T，而非 S-E-B-T，如果开始找到了 S-E-B-T 就马上停止，还能找到最短路径吗？

小明想了想明白了：确实是这样的，所以迪杰斯特拉算法的这个结束条件是必须有的，否则就不能保证找到最优路径。

艾博士进一步补充说：我们还需要强调一下，一般介绍迪杰斯特拉算法时，叙述方法可能与我们这里介绍的不太一样，但是其核心思想是一样的。我们之所以这样讲解，主要是为了从宽度优先搜索算法扩展到迪杰斯特拉算法，后面还将再次扩展到启发式搜索方法，将这几个方法采用同一个框架连贯、统一起来。事实上，宽度优先搜索算法没有利用两个节点间的代价信息，所以只能在单位代价下找到最优路径。而迪杰斯特拉算法利用了两个节点间的代价信息，适应面更加广泛，在非单位代价情况下，也同样能找到最优路径。

小明读书笔记

迪杰斯特拉算法充分利用了到达每个节点的代价信息，优先选择代价最小的节点扩展，即便问题是非单位代价时，也可以找到最优路径。需要强调的是，当被选择扩展的节点是目标节点时算法才结束，而不是一发现目标节点马上就结束，只有这样才能保证算法找到最优解。

3.4　启发式搜索

3.4.1　A 算法

艾博士：迪杰斯特拉算法具有很好的性质，可以找到最小代价的路径。但是该算法也存在明显的不足。小明你想想看，有什么不足？

小明仔细看着图 3.10 所示的搜索图思考了一下说：艾博士，我一时还说不出来有什么不足。

艾博士启发小明说：你把图 3.10 所示的搜索图中节点的扩展次序标注到图 3.9 所示的状态连接图上，是否会发现点什么问题？

小明思考了一下，按照艾博士的要求在状态连接图上标识出节点的扩展次序，如图 3.11 所示。

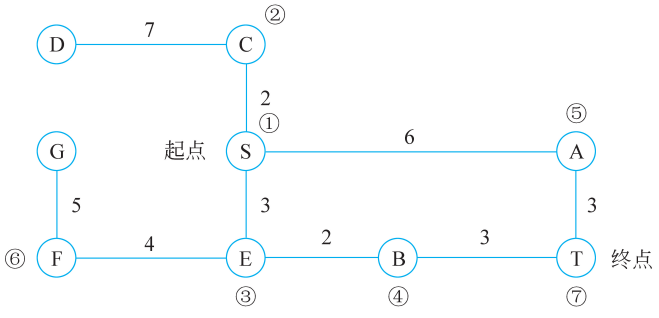

图 3.11　标注了扩展次序的状态连接图

然后，小明看着图思考起来。不一会儿小明就发现了问题，对艾博士说：我知道了迪杰斯特拉算法存在的不足了，我试着说一下，请艾博士看看是否正确。

在图 3.11 中，状态 C 是远离目标状态的，却第二个就被扩展，比距离目标更近且方向也

正确的E、B先扩展。而状态F不仅远离目标状态，方向也是完全相反的，却也要尝试进行扩展。这是不是就是该算法存在的问题？

艾博士高兴地说：小明你分析得非常正确，这正是该算法存在的不足。算法只利用了节点的 g 值，优先扩展 g 值小的节点，而完全没有考虑这个节点是否距离目标更近。这样会造成很多不必要的节点扩展，严重降低算法的求解效率。

小明忍不住问艾博士：那么怎么克服这一不足呢？

艾博士：一种解决办法就是在选择待扩展节点时，不只是考虑该节点的 g 值，同时还要考虑该节点到目标节点的距离。假设我们用 $h(n)$ 表示节点 n 到目标节点的距离，那么：

$$f(n)=g(n)+h(n)$$

就反映了从初始节点S经过节点 n 到达目标节点T的路径距离，也就是这条路径的代价。比如在前面的例子中，由于C和F都是远离目标状态的，它们的 h 值就会比较大，从而导致 f 值比较大，就有可能避免对这两个节点的扩展，从而提高了算法的效率。

听到这里，小明很高兴地说：对啊，这样的话就可以提高搜索效率了。

刚说到这里，小明刚刚还在微笑的面孔突然又凝固起来：可是，一个节点的 h 值怎么计算呢？我们如何知道一个节点到达目标的距离呢？

艾博士说：你的疑虑是对的，一个节点的 h 值确实无法事先知道，但是我们可以大概估计，如果可以估计出一个节点到达目标的大概距离，哪怕是不太准确，那么对于搜索路径也是很有帮助的。

小明：想一想是这样的道理。艾博士，公式中的 $f(n)$、$g(n)$ 和 $h(n)$ 这3个函数具有什么物理含义吗？

艾博士：这3个函数都具有明确的物理含义，我们具体总结一下。$g(n)$ 表示的是从初始节点S到达节点 n 最短路径代价的估计值，通常为搜索图中已经找到的从S到 n 的路径代价。$h(n)$ 称作启发函数，表示的是从节点 n 到达目标节点T最短路径代价的估计值，该值的计算与具体问题有关。$f(n)$ 称作评价函数，表示的是从S出发、经过节点 n 到达目标节点T最短路径代价的估计值，该值通过累加 $g(n)$ 和 $h(n)$ 获得。$f(n)$ 是对节点 n 的总体评价，既考虑了从初始节点S到节点 n 的代价，又考虑从节点 n 到目标节点T的代价，是对通过节点 n 到达目标节点最佳路径代价的估计，体现了节点 n 是否在到达目标节点最佳路径上的可能性。$f(n)$ 越小说明节点 n 在最佳路径上的可能性越大，因此优先扩展 $f(n)$ 小的节点，是一个可行的搜索策略。

采用这样的搜索策略，每次优先选取 f 值最小的节点扩展，直到目标节点的 f 值最小为止，这样的搜索算法称作A算法。A算法与迪杰斯特拉算法的区别就是用节点的 f 值代替 g 值，每次选取 f 值最小的节点优先扩展。

小明有些着急地问道：我觉得A算法中最重要的就是启发函数 $h(n)$ 的计算，那么如何估计节点的 h 值呢？

看着小明的样子，艾博士安慰说：先不用着急，这个问题我们后面再详细讲解。先假定已经给出了节点的 h 值，看看A算法是如何工作的。

还是以上面说的问题为例，但是这次我们给出了每个节点的 h 值，如图3.12所示。

艾博士对小明说：这次你尝试着用A算法求解一下试试。

小明马上找来一张纸画了起来，得到了图3.13所示的搜索图。

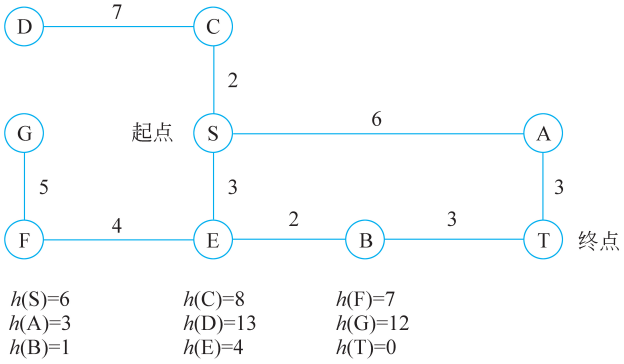

图 3.12　给出了节点 *h* 值的示意图

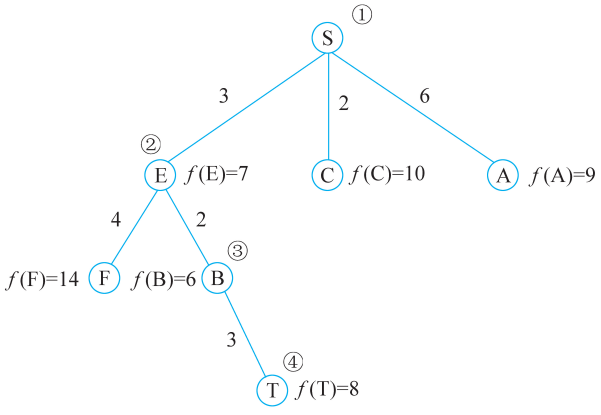

图 3.13　A 算法搜索示意图

艾博士对小明说：请你解释一下做的过程。

小明对照着图 3.13 解释说：首先以初始节点 S 建立根节点，计算 S 的 f 值：

$$f(S)=g(S)+h(S)$$

由于 S 是初始节点，所以 $g(S)=0$，而 $h(S)$图 3.12 中给出的结果是 6，所以 $f(S)=6$。这时待扩展节点只有一个节点 S，选择 S 扩展，生成子节点 A、C、E。分别计算这 3 个节点的 f 值：

$$f(A)=g(A)+h(A)=6+3=9$$
$$f(C)=g(C)+h(C)=2+8=10$$
$$f(E)=g(E)+h(E)=3+4=7$$

从 A、C、E 3 个待扩展节点中，选择 f 值最小的节点 E 优先扩展，生成子节点 B、F。同样计算这两个节点的 f 值：

$$f(B)=g(B)+h(B)=5+1=6$$
$$f(F)=g(F)+h(F)=7+7=14$$

这样待扩展节点为 A、C、B、F 这 4 个节点。从中选择 f 值最小的 B 节点进行扩展，生成出节点 T，计算 T 的 f 值：

$$f(T)=g(T)+h(T)=8+0=8$$

此时待扩展节点为 A、C、F、T 4 个节点，其中 f 值最小的是节点 T，同时 T 也是目标节

点，所以算法结束，得到路径 S-E-B-T。

看到小明的求解结果，艾博士夸奖道：小明讲解得非常清楚，尤其最后特意指明 $f(T)$ 最小才结束，这一点是非常重要的。

在 A 算法中，还有一些细节需要处理，为此我们先给出 A 算法的详细描述，以便说清楚这些细节。

在介绍算法之前，先介绍几个算法中用到的符号。S 为给定的初始节点。OPEN 是一个表，当前的待扩展节点放在该表中，并按照 f 值从小到大排列。CLOSED 也是一个表，所有扩展过的节点放在该表中。

A 算法。

1 初始化：OPEN=(S)，CLOSED=( )，计算 $f(S)$；
2 循环做以下步骤直到 OPEN 为空结束：
3 循环开始
4 从 OPEN 中取出第一个节点，用 n 表示该节点；
5 如果 n 就是目标节点，算法结束，输出节点 n，算法成功结束；
6 否则将 n 从 OPEN 中删除，放到 CLOSED 中；
7 扩展节点 n，生成出 n 的所有子节点，用 m_i 表示这些子节点；
8 计算节点 m_i 的 f 值，由于可能存在多个路径到达 m_i，用 $f(n,m_i)$ 表示经过节点 n 到达 m_i 计算出的 f 值，不同的到达路径其 $g(m_i)$ 值可能不同，但是 $h(m_i)$ 是一样的，因为 $h(m_i)$ 是从 m_i 到目标节点路径代价的估计值，与如何从初始节点到达 m_i 无关；
9 如果 m_i 既不在 OPEN 中，也不在 CLOSED 中，说明这是一个新出现的节点，则将 m_i 加入 OPEN 中，并标记 m_i 的父节点为 n；
10 如果 m_i 在 OPEN 中，并且 $f(n,m_i)<f(m_i)$，则 $f(m_i)=f(n,m_i)$，并标记 m_i 的父节点为 n；
11 如果 m_i 在 CLOSED 中，并且 $f(n,m_i)<f(m_i)$，则 $f(m_i)=f(n,m_i)$，并标记 m_i 的父节点为 n，将 m_i 从 CLOSED 中删除并重新加入 OPEN 中；
12 对 OPEN 中的节点按照 f 值从小到大排序；
13 循环结束
14 没有找到解，算法以失败结束

下面我们对 A 算法做具体解释。

A 算法的基本思想是从待扩展节点中，选一个 f 值最小的节点扩展。为了表述方便，我们将待扩展节点放在 OPEN 中，并对 OPEN 中的节点按照 f 值从小到大排序，这样 OPEN 中的第一个节点就是 f 值最小的节点。而被扩展的节点放在 CLOSED 中。最开始，待扩展节点只有初始节点 S 在 OPEN 中，CLOSED 节点为空。

然后就是循环操作，每次从 OPEN 中取出第一个节点 n，也就是 f 值最小的节点，首先判断 n 是否目标节点，如果是目标节点，输出该目标节点，A 算法结束。否则就扩展节点 n，产生 n 的所有子节点 m_i，然后将 n 从 OPEN 中删除，加入 CLOSED 中。由于从 S 到达 m_i

的路径可能有多个，通过不同路径计算的 $g(m_i)$ 也可能不同，计算出的 f 值也会不同。我们只需要保留一个最小的 f 值即可。由于这次是通过节点 n 扩展出的 m_i，所以我们用 $f(n,m_i)$ 表示经过节点 n 到达 m_i 计算出的 f 值。如果 m_i 既不在 OPEN 中，也不在 CLOSED 中，说明 m_i 是第一次出现的节点，直接将 m_i 加入 OPEN 中，并标记 m_i 的父节点为 n，表示 m_i 是从 n 节点产生的。如果 m_i 在 OPEN 中，说明到达 m_i 至少有两条路径，之前已经从其他路径产生过节点 m_i，并且 m_i 还没有被扩展过。通过之前路径计算得到的 $f(m_i)$ 与通过 n 节点这条新路径计算得到的 $f(n,m_i)$ 进行比较，哪个路径计算的 f 值小就保留哪条路径。如果以前路径计算的 $f(m_i)$ 小，就保持不变，忽略从 n 产生 m_i 这条路径。如果新路径的 $f(n,m_i)$ 小，就标记 m_i 的父节点为 n，并用 $f(n,m_i)$ 作为节点 m_i 的 f 值。如果 m_i 在 CLOSED 中，同样说明到达 m_i 至少有两条路径。如果 m_i 之前的 f 值小于新路径的 f 值 $f(n,m_i)$，则保持不变，忽略从 n 产生 m_i 这条路径。如果新路径的 $f(n,m_i)$ 小，就标记 m_i 的父节点为 n，并用 $f(n,m_i)$ 作为节点 m_i 的 f 值。但是由于 m_i 是在 CLOSED 中的，说明 m_i 已经被扩展过，其子节点已经生成，如果 m_i 的父节点被修改了，则到达 m_i 的子节点及其后续节点（如果已经产生的话）的路径也被改变了，g 值将有所变化，所以 m_i 的子节点及其后续节点的 f 值也应该有所改变。为了处理这种情况，A 算法采用了一种简化处理的方法，先将 m_i 从 CLOSED 中删除，然后将 m_i 重新放回到 OPEN 中，当以后该节点排在 OPEN 中的第一位、m_i 节点再次被选择进行扩展时，重新生成其子节点，这时按照 A 算法自然就修改了 m_i 的子节点的 f 值，简化了 A 算法的处理过程。最后对 OPEN 中的节点按照 f 值大小进行排序，再次循环进行以上操作，直到算法结束。

A 算法有两个结束出口：第一个出口是，当 OPEN 的第一个节点是目标节点时，说明找到了从初始节点到目标节点的路径，算法找到解成功结束；第二个出口是，当 OPEN 中不含有任何节点，也就是 OPEN 为空时，算法退出循环结束，此时表示算法已经遍历了所有的可能，但是仍然没有找到解，算法失败结束。

介绍完 A 算法之后，艾博士再次强调说：请注意 A 算法的结束条件，必须是当目标节点的 f 值在 OPEN 中最小，也就是目标节点排在 OPEN 表的第一个位置时，算法才成功结束，而不是目标一出现就结束算法。这是初学者最容易犯的错误，一定要牢记这一点。该结束条件与迪杰斯特拉算法是一样的。

听了艾博士的讲解，小明说：谢谢艾博士的讲解，我一定牢记住这个结束条件。能否再举一个 A 算法求解的例子呢？

艾博士： 好的，我们再举一个用 A 算法求解八数码问题的例子。该八数码问题如图 3.14 所示。

艾博士： 为了用 A 算法求解八数码问题，需要先定义 h 函数，以便计算 f 值。我们定义 h 函数为“不在位的将牌数”。也就是说，看一个状态的所有将牌所在的位置，与目标状态将牌所在位置进行比较，看有多少个将牌位置与目标状态不一致。不在位将牌的数量就是该状态的 h 值。比如图 3.15 所示的状态，与目标相比 1、2、6、8 四个将牌不在目标位置，所以该状态的 h 值就是 4。由于八数码问题一次只能移动一个将牌，在将牌移动一次的代价为 1 的情况下，不在位将牌数一定程度上体现了一个状态与目标状态的距离，可以用来估计其到达目标所需要的代价。

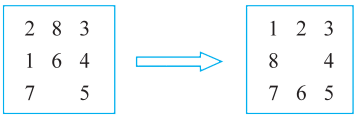

图 3.14 八数码问题举例

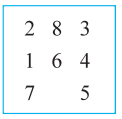

图 3.15 不在位将牌数作为 h 函数计算举例

图 3.16 给出了采用 A 算法求解该八数码问题的搜索图。图中红圈中的数字表示节点被扩展的次序，英文字母旁括号中的数字表示计算出的该节点的 f 值。

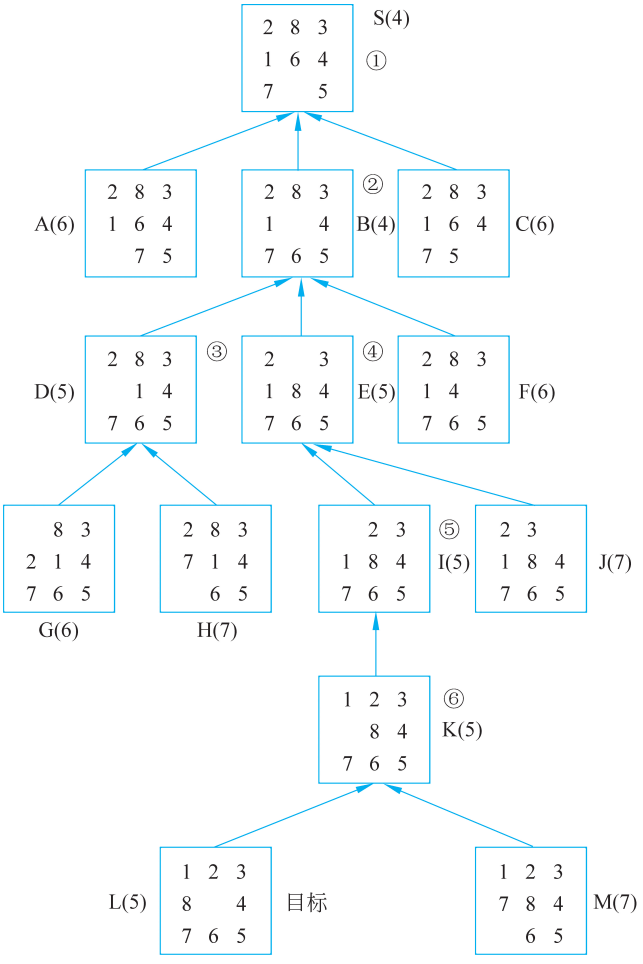

图 3.16 A 算法求解八数码问题的搜索图

按照 A 算法，S 首先被扩展，产生 A、B、C 3 个子节点，并标注这 3 个节点的父节点为 S，图中用指向 S 的箭头表示，S 被放入 CLOSED 中。由于这 3 个节点都是经过一步产生的，所以 g 值都是 1。节点 A 有 1、2、6、7、8 五个将牌不在位，所以 $h(A)=5$，加上 A 的 g 值 1，有 $f(A)=1+5=6$，同理可以得到 $f(B)=4$、$f(C)=6$。这样 OPEN 和 CLOSED 分别如下：

$$\text{OPEN}=(B(4),A(6),C(6))$$

$$\text{CLOSED}=(S(4))$$

从 OPEN 中取出第一个节点 B(4) 放入 CLOSED 中，扩展 B(4) 生成子节点 D、E、F，

并标注这 3 个节点的父节点为 B。这 3 个节点均是经过两步走成的，所以 g 值均为 2。节点 D、E 都是 1、2、8 三个将牌不在位，所以 h 值都是 3。节点 F 是 1、2、4、8 四个将牌不在位，其 h 值为 4。这样得到 $f(D)=f(E)=2+3=5$，$f(F)=2+4=6$。D(5)、E(5)、F(6) 分别放入 OPEN 中并按照 f 值排序，有：

OPEN＝(D(5)，E(5)，A(6)，C(6)，F(6))

CLOSED＝(S(4)，B(4))

从 OPEN 表中取出第一个节点 D(5)放入 CLOSED 中，扩展 D(5)生成子节点 G、H，并标注这两个节点的父节点为 D，计算有 $f(G)=6$、$f(H)=7$，分别放入 OPEN 中并排序，有：

OPEN＝(E(5)，A(6)，C(6)，F(6)，G(6)，H(7))

CLOSED＝(S(4)，B(4)，D(5))

从 OPEN 表中取出第一个节点 E(5)放入 CLOSED 中，扩展 E(5)生成子节点 I、J，并标注这两个节点的父节点为 E，计算有 $f(I)=5$、$f(J)=7$，分别放入 OPEN 中并排序，有：

OPEN＝(I(5)，A(6)，C(6)，F(6)，G(6)，H(7)，J(7))

CLOSED＝(S(4)，B(4)，D(5)，E(5))

从 OPEN 表中取出第一个节点 I(5) 放入 CLOSED 中，扩展 I(5) 生成子节点 K，并标注 K 的父节点为 I，计算有 $f(K)=5$，放入 OPEN 中并排序，有：

OPEN＝(K(5)，A(6)，C(6)，F(6)，G(6)，H(7)，J(7))

CLOSED＝(S(4)，B(4)，D(5)，E(5)，I(5))

从 OPEN 表中取出第一个节点 K(5)放入 CLOSED 中，扩展 K(5)生成子节点 L、M，并标注这两个节点的父节点为 K，计算有 $f(L)=5$、$f(M)=7$，分别放入 OPEN 中并排序，有：

OPEN＝(L(5)，A(6)，C(6)，F(6)，G(6)，H(7)，J(7)，M(7))

CLOSED＝(S(4)，B(4)，D(5)，E(5)，I(5)，K(5))

从 OPEN 中取出第一个节点 L(5)，发现 L 就是目标节点，算法结束输出目标节点 L。

小明听着艾博士的讲解，对照着图 3.16 所示的搜索图说：不在位的将牌数虽然看起来是一个并不太准确的路径代价估计值，但是效果还是挺好的，看来 A 算法是一个不错的算法。但是我还有个问题，虽然 A 算法最终找到了目标节点，但是算法并没有说如何得到这条解路径，如何获得从初始节点到达目标节点的解路径呢？对于八数码问题来说，我们更关心的应该是如何移动将牌到达目标，而不是目标本身。

艾博士说：小明你提了一个很好的问题。虽然 A 算法并没有说如何得到解路径，但是保留了相关的信息。比如每个产生的节点都有一个父节点标记，算法成功结束时输出的是找到的目标节点，这样从目标节点开始，一步步沿着父节点标志反向寻找到初始节点，就得到了需要的解路径。

小明：我明白如何得到解路径了。对于图 3.16 所示的搜索图来说，算法输出的是目标节点 L，由 L 知道其父节点是 K，而 K 的父节点是 I，以此类推，我们可以得到该问题解路径的逆序表示为 L-K-I-E-B-S，将其反转过来就是 S-B-E-I-K-L，就得到了该问题的解。也就是将牌 6 下走一步，将牌 8 下走一步，将牌 2 右走一步，将牌 1 上走一步，将牌 8 左走一步，就实现了从给定的初始状态通过将牌的移动达到目标状态。

艾博士：正是这样的。

3.4.2 A* 算法

艾博士：小明，你发现没有，我们一直没有说 A 算法是否能够找到最优路径？

小明：确实啊。从前面的八数码问题例子看，A 算法的求解效率还是很高的，但是是否能保证找到最优解呢？

艾博士：我们在解释一个节点的 h 值时，只说了是该节点到达目标节点路径代价的一个估计值，并没有对 h 函数做具体的限制。所以在对 h 函数没有任何限制的情况下，不好讨论 A 算法的性质。

可以证明，对于任何一个节点 n，如果 $h(n)\leqslant h^*(n)$的话，则 A 算法一定可以在有解的情况下找到最优解，也就是代价最小的路径。其中 $h^*(n)$表示从节点 n 到达目标节点最优路径的代价。

当满足条件 $h(n)\leqslant h^*(n)$时，A 算法称作 A* 算法，其中 $h(n)\leqslant h^*(n)$称作 A* 条件。

小明有些不太明白地问：当 h 函数满足 A* 条件时，A 算法就是 A* 算法吗？

艾博士肯定地回答说：是的，从算法描述的角度来说，A* 算法与 A 算法是完全一样的，只是对 h 函数加上了 A* 条件限制。

小明又有些疑惑地问道：一般来说，在求解之前我们并不知道 $h^*(n)$的具体大小，那么如何判断 h 函数是否满足 A* 条件呢？

艾博士：这又是一个很好的问题。我们需要根据具体问题具体判断。比如说我们在地图上用 A* 算法求给定两点的最短路径，假设我们知道每个路口，也就是节点的坐标的话，就可以将 h 函数定义为每个路口到达终点的欧氏距离(欧几里得度量)。因为欧氏距离是两点间的最短距离，所以一个路口到达终点的实际最短距离不可能小于欧氏距离，所以这样的 h 函数是满足 A* 条件的。

再比如，在前面的八数码问题例子中，我们定义 h 函数为不在位的将牌数。由于八数码问题每次只能移动一步将牌，而移动一步将牌理想情况下最多也是把一个不在位的将牌移动到位，所以当有 m 个不在位将牌时，至少需要 m 步才可能把所有将牌移动到位，所以这样的 h 函数也是满足 A* 条件的。

小明：我明白了，判断一个 h 函数是否满足 A* 条件，需要根据具体问题具体分析，根据问题和定义的 h 函数判断是否满足 A* 条件。

艾博士肯定地说：就是这样的，这里并不存在一般的方法，只能具体问题具体分析，根据实际问题判断是否满足 A* 条件。

3.4.3 定义 h 函数的一般原则

小明又问艾博士：A* 算法最重要的就是一个满足 A* 条件的 h 函数。那么应该如何定义 h 函数呢？

艾博士：如何定义 h 函数是 A* 算法应用中最重要的内容，只能根据具体问题具体讨论，但还是有一般原则的。

小明忙问道：有哪些一般原则呢？

艾博士：一个重要的原则可以总结为，放宽原问题的限制条件，在宽条件下求解一个状

态到目标状态的最优路径代价，以此代价作为该状态的 h 值。在宽条件下求解的最小代价，不会比严格条件下的最小代价大，所以这样得到的 h 函数肯定可以满足 A^* 条件。

小明：应该怎样放宽条件呢？

艾博士：这个问题比较重要，而且与具体问题有关，我们通过几个例子加以说明，从简单问题到复杂一些的问题。

1. 地图上求解两点间的最优路径问题

艾博士：地图上求解两点间的最优路径问题，其限制条件是必须沿着道路前进，道路不一定是笔直的，可能有弯曲，甚至在某段路可能会向相反的方向行进。我们可以放宽这个条件，比如任何两点间都可以直行，不必沿着道路行进。这样放宽条件后，任何一个路口到终点的路径最小代价都可以用欧氏距离计算，该距离一定不会大于严格条件下最优路径的代价，所以用欧氏距离定义 h 函数一定可以满足 A^* 条件。

小明：这里例子比较容易理解，欧氏距离肯定是两点间的最短距离。

2. 八数码问题

艾博士：八数码问题的限制条件是将牌每次只能移动一步，而且只能移动到空格位置。我们把这一限制条件放宽，假定每个将牌可以“跳跃”，从当前位置跳跃到目标位置，而且不管目标位置是否有其他将牌存在，都可以跳跃过去。这样当有 m 个将牌不在位时，采用这种跳跃的方式，最多经过 m 次跳跃，就达到了目标状态。因此，这种宽条件下所移动的将牌次数不会多于原问题严格条件下的将牌移动次数，所以用“不在位的将牌数”定义 h 函数，可以满足 A^* 条件，而且不在位将牌的多少，也一定程度上反映了该状态到达目标状态的距离。

小明恍然大悟道：原来用不在位将牌数作为 h 函数值是这样定义出来的。但是对于八数码问题来说，将限制放宽到可以跳跃，似乎条件放得太宽了，是否可以收紧一些呢？

艾博士：这正是我想要说的，其实条件可以再收紧一些。八数码问题将牌每次只能移动一步，我们可以继续保留这个限制，但是放宽只能移动到空格位置这个条件，也就是不管旁边位置是否有将牌，都可以移动过去。在这样的宽松条件下，每个不在位的将牌需要移动 k 步到达目标位置，而 k 就是该将牌到达目标位置的距离。所以我们可以用“每个不在位将牌到其目标位置的距离和”来定义 h 函数。这是在宽松条件下达到目标状态所需要的最小代价，所以一定不会多于严格条件下所需要的代价，满足 A^* 条件。

下面我们举例说明这种情况下如何计算一个状态的 h 值。在图 3.17 给出的例子中，1、2、6、8 四个将牌不在目标位置，其中 1、2、6 三个将牌距离目标位置均为 1，也就是经过 1 步移动就可以到达目标位置，将牌 8 需要移动两步才能到达目标位置，所以将牌 8 到达目标位置的距离为 2，这样四个不在位将牌距离目标位置的距离和为 $1+1+1+2=5$，则该状态的 h 值为 5。

小明：八数码问题这两个例子很好地说明了如何通过放宽条件，在宽松条件下定义 h 函数的问题，很受启发，大概了解了如何定义 h 函数的思路了。

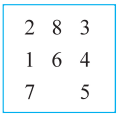

图 3.17 用不在位将牌距离目标位置距离和作为 h 函数计算举例

3. 传教士与野人问题

艾博士：下面我们再举一个稍微复杂一点的例子——传教士与野人问题。

传教士与野人问题描述如下。

有 5 个传教士和 5 个野人来到河边准备渡河，河岸有一条船，每次至多可供 3 人乘渡。问如何规划摆渡方案，使得任何时刻，在河的两岸以及船上的野人数目总是不超过传教士的数目(但允许在河的某一岸或者船上只有野人而没有传教士)。假设传教士和野人都会划船，而没有传教士和野人以外的其他划船人。

在这里我们主要讨论如何设计 h 函数，以便可以用 A^* 算法求解该问题。

我们假设开始时传教士和野人在河的左岸，目标是乘船到达河的右岸。由于总人数是固定的，所以我们可以考虑用左岸传教士和野人的人数，以及船在河的哪边表示一个状态。比如表达为一个如下三元组：

$$(M, C, B)$$

其中，M、C 分别表示在左岸的传教士和野人人数；B 表示船在河的哪一边，船在左岸时 $B=1$，船在右岸时 $B=0$。

该问题的一个特点是船在河的两岸转换，如果这个状态船在左岸，则下一个状态船在右岸；如果这个状态船在右岸，则下一个状态船在左岸。所以在定义 h 函数时要考虑这种情况，需要分别考虑船在不同的岸边的情况。同时我们假设，摆渡一次船的代价为 1，而不考虑船上具体有多少人。

如何通过放宽条件的方法得到该问题的 h 函数呢？

对于传教士和野人问题，主要的限制条件是在摆渡的过程中“在河的两岸以及船上的野人数目总是不超过传教士的数目”，我们将这个限制条件去掉，只考虑船每次最多可供 3 人乘渡这一个条件。

先考虑船在左岸的情况。如果不考虑限制条件，也就是说，船一次可以将 3 人从左岸运到右岸，然后再有 1 人将船送回来。这样，船一个来回可以将 2 人运过河，而船仍然在左岸。而最后剩下的 3 人，则可以一次将他们全部从左岸运到右岸。所以，在不考虑限制条件的情况下，至少需要摆渡$\left\lceil \frac{M+C-3}{2} \right\rceil \times 2+1$ 次。其中分子上的“-3”表示剩下 3 人留待最后一次运过去。除以 2 是因为一个来回可以运过去 2 人，需要$\frac{M+C-3}{2}$个来回把除了最后 3 人外的其他所有人运送过去，因为“来回”数不能是小数，需要向上取整，这个用符号$\lceil\ \rceil$表示。乘以 2 是因为一个来回相当于两次摆渡，总代价要乘以 2。最后的“$+1$”，则表示将剩下的 3 人运过去，需要一次摆渡。可以说，$\left\lceil \frac{M+C-3}{2} \right\rceil \times 2+1$ 是在宽松条件下，把所有的

传教士和野人全部从左岸摆渡到右岸所需要的最小摆渡次数。

化简有：

$$\left\lceil \frac{M+C-3}{2} \right\rceil \times 2+1 \geqslant \frac{M+C-3}{2} \times 2+1=M+C-3+1=M+C-2$$

所以，当状态是船在左岸时，需要的摆渡次数至少为 $M+C-2$ 次，其中 M、C 分别为当前状态在左岸的传教士和野人人数。

再考虑船在右岸的情况。同样不考虑限制条件。船在右岸，需要一个人将船运到左岸。因此对于状态$(M,C,0)$来说，其下一个状态是$(M+1,C,1)$或者是$(M,C+1,1)$，具体看是传教士将船运到左岸，还是野人将船运到左岸，在宽松条件下我们并不需要具体区分这两种情况，我们假设下一个状态为$(M+1,C,1)$，这时就相当于船在左岸的情况了。按照前面的分析，我们将此时左岸的传教士人数 $M+1$ 和野人人数 C 代入上面分析的船在左岸的情况，可以得到对于状态$(M+1,C,1)$至少需要$(M+1)+C-2$ 次摆渡。但是我们需要计算的是船在右岸时状态$(M,C,0)$所需要的最少摆渡次数，而状态$(M+1,C,1)$是状态$(M,C,0)$经过一次摆渡达到的，所以$(M+1,C,1)$所需要的最少摆渡次数加 1，就是$(M,C,0)$所需要的最少摆渡次数。所以有$(M,C,0)$需要的最少摆渡次数为$(M+1)+C-2+1$，化简有 $M+C$。

总结以上情况有：

对于船在左岸时的状态$(M,C,1)$，需要的最少摆渡次数为 $M+C-2$。

对于船在右岸时的状态$(M,C,0)$，需要的最少摆渡次数为 $M+C$。

考虑到船在左岸时 $B=1$，船在右岸时 $B=0$。两种情况可以综合在一起，状态(M,C,B)需要的最少摆渡次数为 $M+C-2B$。

由于该摆渡次数是在不考虑限制条件下推出的最少所需要的摆渡次数。因此，当有限制条件时，最优的摆渡次数不可能小于该摆渡次数。所以这样定义的 h 函数一定满足 A^* 条件。

小明：这个例子确实有点复杂，但也可以很好地体现通过放宽限制条件定义 h 函数的思想。如果不采用降低条件的方法，都不知道从何处着手。

3.5　深度优先搜索算法

小明：艾博士，我听说还有一种叫作深度优先的搜索算法？

艾博士：是的，下面我们就介绍一下深度优先搜索算法。小明，你说说宽度优先搜索算法是如何选择被扩展节点的？

小明想了一下回答说：宽度优先搜索算法优先选择节点深度最浅的节点扩展。

艾博士：小明说得非常正确。与宽度优先搜索算法相反，深度优先搜索算法优先选择深度最深的节点扩展。

小明问道：深度优先搜索算法具体是如何实现的呢？

艾博士：深度优先搜索算法有不同的实现方法，一种最常用的实现方法是利用回溯方法实现的。

小明：什么是回溯方法呢？

艾博士：假如我们走迷宫。我们事先规定好某种策略，比如每次遇到路口时，优先选择最左边的路口进入，如果遇到死胡同，则退回来试探第二个路口。当然很多情况下并不是直接就遇到死胡同，而是探索了若干个可能的走法之后才发现进入这个路口是不可行的，这样也相当于遇到了死胡同，退回来再选择其他可能的路口进行试探。这种遇到死胡同就退回的方法称为回溯方法。

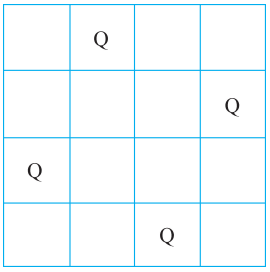

图 3.18　四皇后问题

为了介绍如何用回溯方法实现深度优先搜索，我们通过四皇后问题的求解，介绍深度优先搜索方法。

小明：请艾博士介绍一下什么是四皇后问题。

艾博士：所谓四皇后问题，就是在一个 4×4 的棋盘上，如何摆放 4 个皇后，使得任何两个皇后都不在一条直线上，包括横线、纵线和斜线。比如图 3.18 所示的就是四皇后问题的一个解，其中 Q 表示皇后。

为了叙述方便，我们设棋盘从上到下为第一行、第二行……从左到右为第一列、第二列……我们用棋盘上皇后所在的坐标组成的表表示四皇后问题的一个状态。比如图 3.18 所示的状态可以表示为：

((1,2),(2,4),(3,1),(4,3))

图 3.19 给出了采用深度优先搜索算法求解四皇后问题的搜索图。在求解过程中，我们从第一行开始，一行一行地进行由上到下探索。而在每一行，我们从第一列开始，一列一列地从左到右进行探索。具体过程如下。

图中最开始是一个空表，表示棋盘上没有皇后。然后在(1,1)位置放置一个皇后，得到状态((1,1))。

然后在第二行从左到右按列探索。第二行的一、二列都不能再放置皇后了，所以只能在第三列放置第二个皇后，得到状态((1,1),(2,3))。此时我们得到的皇后摆放情况如图 3.20 所示。

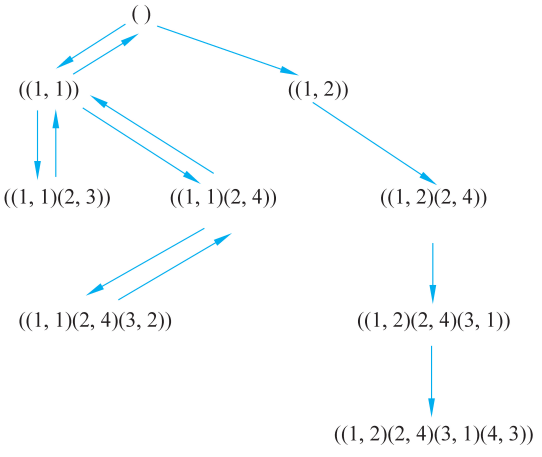

图 3.19　四皇后问题搜索图

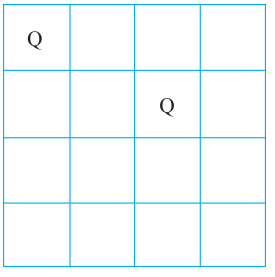

图 3.20　皇后的摆放情况示意图