第3章

启发式搜索和退火

对盲搜的讨论使我们了解了人工智能搜索的基本原理。尽管如此，我们必须承认，基本技术只在简单的领域有用。当涉及更贴近现实的问题时，它们就会因为其盲目性而受挫。在创建了子状态之后，不考虑它们各自的特性优势，而只是将它们随机地放在列表 L 中。对于实际工程而言，这是不够的。

人类在寻求解决问题的方案时，不会那么机械。我们倾向于通过启发式、经验法则或指导方针对“子状态”进行排序，这有助于将解决方案的过程引向更具可行性的途径。在上下文搜索中，启发式采用评价函数的形式，为每个状态返回其值。例如，该状态与最终状态的接近度。优先选择能产生更高值的子状态的操作可以提高搜索过程的效率。

这就是构成下面所讨论的整个启发式搜索算法系列的主干思想。此外，本章还简要地提到了一种主要的技术选择，即模拟退火。

3.1 爬山算法和最佳优先搜索

下面从两个基本算法开始。两者都依赖于估计搜索的状态的评价价值的能力。

3.1.1 评价函数

工程师向机器传达状态质量的概念的方式是评价函数。输入是对状态的描述，输出是一个有助于估计该状态与给定问题解决方案的接近程度的值。

3.1.2 数值举例：滑方块

衡量与最终状态相似度的最简单方法是计算错误方块的数量——这些方块与最终状态规定的方块不同。因此，在如图 2.3 所示的情况下，初始状态的一些数字(1、3、5、6 和 7)位于“正确的”方块上；剩下的 3 个数字(2、4 和 8)都在“错误的”方格上。因此，初始状态与最终状态的距离为 $d=3$。

诚然，这是一个简单的标准。更好的方法还会考虑错位方块与正确位置的距离。换句话说，该标准将询问在不受其他方块阻碍的情况下，每个方块需要走多少步才能到达正确的

位置。对于数字 2、4 和 8，其各自的距离分别为 1、2 和 2。因此，该状态与最终状态的距离为 $d=1+2+2=5$。

即使这样也不够完美。一个方块离正确位置的距离并不等同于把它送到那里的难度。一个更好的标准应该试图量化这个方面。当然，设计这样的函数不是一件容易的事；程序员如果没有一定的知识、经验和创造力，是不可能成功的。从这个意义上来说，评价函数向程序传达了人类专家的某些洞察力。

3.1.3　复杂的评价函数

2.3 节解释了最终状态可能不仅仅是由一个精确的模式定义的，而是由最终状态所要满足的条件来定义的。因此，在幻方的情况下，我们可能被要求找到一个“至少一列只包含偶数或至少一行只包含奇数的状态”。在这种情况下，开发有用的评价函数可能需要合适的创造力。

3.1.4　最大化或最小化

评价函数有两种基本方法。在刚才介绍的那个问题中，评价函数依赖于当前状态和最终状态之间的距离。该值要最小化：距离越小，状态越接近最终解。另一种可能性是量化两种状态之间的相似性。这是要最大化的：相似性越大越好。

无论工程师选择哪种方法，都必须前后保持一致。忘记在开发的软件中使用的评价函数是最大化还是最小化是愚蠢的。

3.1.5　爬山算法

深度优先搜索的一个缺点是，它以随机顺序将生成的子状态放在列表 L 中。评价函数可以让我们做得更好。一种被称为爬山的改进，将子状态按照评价函数确定的顺序放在 L 的开头，这样最可行的状态就会出现在列表前面。原理由表 3.1 的伪代码给出。

表 3.1　爬山算法的伪代码

输入：包含初始状态的列表 L，之前访问过的状态的空列表 L_{seen}、程序员定义的评价函数。

1. 设 s 是 L 中的第一个状态，如果 s 是一个最终状态，则成功终止。
2. 应用状态 s 所有合法的搜索操作符，从而获得其子状态。
3. 忽略那些已经在 L_{seen} 中的子状态。
4. 根据评价函数返回的值对剩余的子状态进行排序，并将其插入 L 的开头。
5. 把 s 从 L 中去掉，放在 L_{seen} 的末尾。
6. 若 $L=\varnothing$，以失败终止，否则返回步骤 1。

同样，我们不能忘记，如果评价函数度量的是输入状态与最终状态的相似性，那么第一个子状态应该是值最高的那一个。如果函数度量距离，那么第一个子状态应该是具有最小值的那一个。

3.1.6　最佳优先搜索

爬山法使搜索总是由父状态到子状态；只有在每个子状态都考察完毕之后，搜索才被允许返回到之前的状态。然而，在某些应用程序中，工程师会意识到被拒绝状态的所有子状

态的值都低于先前状态的值。

在这种情况下，放弃当前的路径（它没有提供任何直接的改善），并回到先前那个值更高的状态是有意义的。

这就是表 3.2 中伪代码所总结的最佳优先搜索算法的原理。读者会注意到其中的区别：每次要检查一个新的状态是否为最终状态时，最佳优先搜索都会从整个列表 L 中选择具有最佳值的状态，而不管这个状态在搜索树中位于什么位置。相比之下，爬山法只选择刚被拒绝的状态的最佳子状态。

3.1.7 实现最佳优先搜索的两种方法

一种简单的实现总是将新创建的子状态附加在 L 中的某个预定义位置（开头或结尾），并将从评价函数收到的值联系到每个子状态上。原则上，列表不必是有序的。当需要验证下一个状态是否为最终状态时，程序根据最优的函数值选择该状态。这是表 3.2 中的伪代码给出的方法。

表 3.2 最佳优先搜索算法的伪代码

输入：包含初始状态的列表 L，之前访问过的状态的空列表 L_{seen}，程序员定义的评价函数。

1. 设 s 是 L 中具有最高评价函数返回值的状态。如果 s 是一个最终状态，则成功终止。
2. 应用状态 s 所有合法的搜索操作符，从而获得其子状态。
3. 忽略那些已经在 L_{seen} 中的子状态，把剩下的放到 L 中。
4. 把 s 从 L 中去掉，放在 L_{seen} 的末尾。
5. 若 $L=\varnothing$，以失败终止，否则返回步骤 1。

另一种实现采取了不同的方法。每次创建并评价新的子状态时，程序都会将它们插入 L 的适当位置，以便列表始终按照从最佳状态到最劣状态排序。[①] 当要选择下一个状态时，程序选择 L 的第一个元素。

具体的解决方案将取决于具体情况，以及工程师的个人偏好和常识。

3.1.8 两种方法的比较

图 3.1 和图 3.2 说明了爬山算法和最佳优先搜索算法的主要区别。在爬山算法中，下一个要考察的状态总是从当前状态的子状态中选择的。最佳优先搜索则更加灵活：它继续搜索整个列表 L 中的最佳状态。这个最佳状态不必是当前状态的子状态。

更高的灵活性能够保证更快的搜索吗？大多数情况下确实如此。然而，并不是总是这样。在某些领域，子状态总是比父状态更好，在这种情况下，爬山算法是正确的，而最佳优先搜索的灵活性，只会因为必须始终识别最佳的下一个状态而产生额外的开销，进而减慢进程。

3.1.9 人类的搜索方式

一个小学生在面对滑方块谜题时，会每次移动一个方块，将当前的状态转换成另一个状态，然后再转换成另一个状态——这意味着一个总是从父状态到子状态的过程。

① 为此，通常使用链表数据结构。

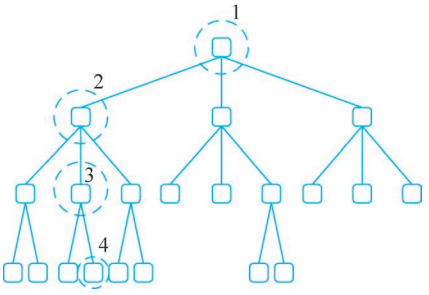

图 3.1　圈出的状态已被验证是否为最终状态。整数表示爬山的顺序。注意，这个算法总是从父状态到子状态进行

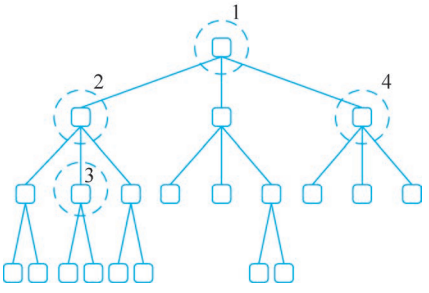

图 3.2　圈出的状态已被验证是最终状态。整数表示最佳优先搜索顺序。注意，这个算法并不总是从父状态到子状态进行

换句话说，他遵守爬山算法的原则。同时，小学生会经常重访同一状态，甚至也许都没注意到这一事实，因为他缺乏列出 L_{seen} 清单的能力。我们可能会说，人类解决问题遵循某种不完美的爬山原则。

控制问题

如果你在回答下列任何问题时遇到困难，那么请返回阅读前文的相应部分。

- 启发式这个词是什么意思？评价函数在哪些方面起到启发式的作用？
- 解释爬山算法和最佳优先搜索算法的原理。并就与具体实现有关的问题发表评论。
- 比较爬山算法和最佳优先搜索算法的行为。在什么情况下，其中一种会优于另一种？

3.2　评价函数的实践方面

每个程序员都知道，仅掌握算法的原理是不够的。要想成功，就必须准备好处理许多情况，而这些情况是课堂上的例子所避免的，因为出于教育目的，例子必须是简单的。然而，忽视这些复杂的情况可能会导致一个看似深思熟虑的项目惨遭失败。因此，让我们看看工程师可能遇到的情况。

3.2.1　状态值的时间恶化

让我们回到滑方块谜题。先假设初始状态和最终状态如图 3.3 所示，然后进一步假设评价函数定义如下：创建一个计数器，初始化为 0。对于每个偏离最终状态位置的方块，将

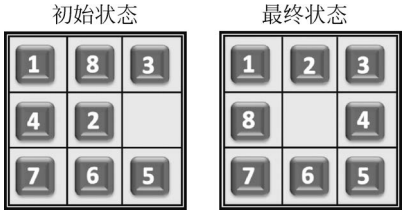

图 3.3 在初始状态下移动任何一个方块都会使其状态与最终状态的距离增加 1

其所需移动的步数添加到计数器中，以使其到达空棋盘上的正确位置。在如图 3.3 所示的例子中，因为放错位置的方块，所以距离是 $d=1+2+2=5$，2、4 和 8 需要分别移动 1、2 和 2 步。读者应该还记得，这个公式是在前面章节提出的。

现在看看这个评价函数是如何影响求解过程的。在初始状态下，有 3 个操作是可能的。其中一个操作是将 3 号方块滑下去。由于这个方块已经在正确的位置上，所以这个操作只会使它与正确位置的距离增加 1，而评价函数对新状态返回的值也相应增加。如果把 5 号移上去，也会发生同样的情况。最后一个选择是将 2 号向右移动。这一个棋子并不在正确的位置上；然而，滑动它将使其与最终状态的距离从 1 变到 2。

我们已经得出了一个重要的观察结果：在寻找问题的解的过程中，程序可能被迫接受一个中间状态，其值似乎更加偏离我们想要的结果，就像游客在攀登山峰之前可能不得不穿越中间的山谷一样。

3.2.2 多个状态可以有相同的值

同一个示例说明了另一种经常遇到的情况：3 个搜索操作符中的每一个都增加了与最终状态相同的距离。

在这种情况下，爬坡算法无法决定（除了随机选择）应用 3 个搜索操作符中的哪一个。

不可否认，我们所使用的评价函数是相当粗糙的。如果搜索程序依靠的是一个更精妙的公式，可能会给每个状态的子状态一个不同的值，那么它就不太可能面临刚刚的困难。另一方面，我们不能忘记评价函数只不过是工程师直觉上的近似值，因此可能是不可靠的；它们甚至可能把搜索带向错误的方向。

3.2.3 前瞻性评价策略

前面看到，智能体有时必须在有同等价值的状态中做出选择。我们现在还了解到，搜索过程可能要克服局部的高峰（或低谷），而不知道后面会出现什么。这些观察启发了所谓的前瞻性评价策略。

这个想法是，智能体不应该基于对状态的直接子状态的评价来做出选择，而应该基于对更远的后代的子状态："孙子"状态、"曾孙"状态等来进行选择。因此，在如图 3.4 所示的搜索树中，状态 b 的值为 2.5，但深度为 2 的前瞻性评价策略将使用值 3.5 进行操作，这是有最大值的"孙子"状态 m 的值。

智能体应该前瞻多深是一个用户设置的参数。当然，更大的深度往往会缓解之前提到的复杂问题的严重性（如评价函数的局部极值）。但是，评价大量"后代"状态的成本可能很高。

3.2.4 集束搜索

处理上述困难的另一种方法是并行运行两个或多个搜索。这一原则被称为集束搜索，

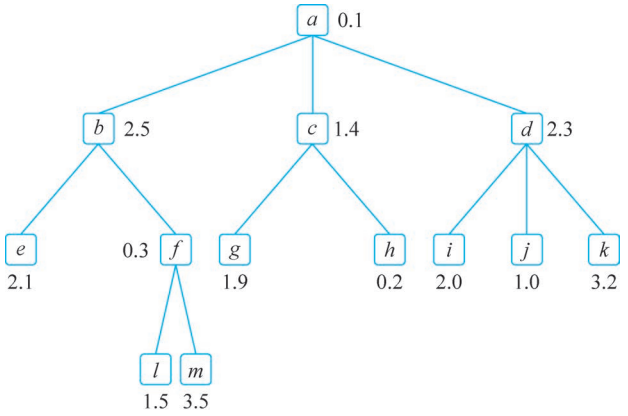

图 3.4　一个搜索树的例子。每个节点代表一种状态。对于每个状态，都提供了评价函数返回的值

参见表 3.3 中的伪代码。我们的想法是，不仅要挑选一个单一的最佳状态（例如，在爬山算法中那个最佳的子状态），还要挑选 N 个最佳状态。如果这些状态中没有一个是最终状态的，那么这 N 个状态就会被立即从 L 转移到 L_{seen}，并在 L 中被其所有的子状态取代，然后适当地排序。当然，同样的原则也可以在最佳优先搜索的背景下使用。

表 3.3　集束搜索的伪代码，最佳优先搜索的“并行版本”

输入：包含初始状态的列表 L，之前访问过的状态的空列表 L_{seen}，由程序员定义的评价函数、用户指定的 N 值。

1. 依据评价函数对列表 L 进行排序。若 $L=\varnothing$，以失败终止。
2. 设 $N'=\min\{N,\text{length}(L)\}$。如果 L 中的前 N' 个元素中至少有一个是最终状态，则成功终止。
3. 找出这 N 个状态的所有子状态。忽略那些已经在 L_{seen} 中的，将剩下的插入 L 中。
4. 将前 N 个状态从 L 移到 L_{seen} 中。
5. 返回步骤 1。

读者可能会觉得伪代码的第 2 步不太直观。下面是“奇怪”术语 $N'=\min\{N,\text{length}(L)\}$ 背后的动机。有时，用户指定的 N 值超过列表 L 当时的长度。例如，如果用户指定 $N=3$，但 L 只包含两种状态，则会发生这种情况，因此 $\text{length}(L)=2$。在这种情况下，智能体只能评价 $\min\{3,2\}=2$ 个状态。

3.2.5　N 在集束搜索中的作用

我们建议 N 取什么值？答案一如既往地取决于某些权衡。较高的 N 值有助于智能体越过评价函数的局部极值，也有助于减少忽略浅解（从初始状态出发路径较短的解）的危险。然而，这些好处可能会被高内存成本抵消。

通过对该算法的研究，读者会意识到，N 值非常大的集束搜索算法可能会退化为广度优先搜索算法，不再是启发式的搜索，而成为盲搜。

3.2.6　数值举例

考虑如图 3.4 所示的搜索树所代表的简单问题。树中的每个节点代表一个状态。每个

状态旁边都有一个由评价函数返回的值。例如，状态 b 的值是 2.5。然后，表 3.4 详细列出了目前为止，我们看到的 3 种启发式搜索技术的行为。对于每一种，我们都提供了两个列表 L 和 L_{seen} 的逐步演变过程。

表 3.4　图 3.4 中启发式搜索技术应用于搜索树的几个步骤。状态值是要最大化的

BFS 算法	
L	L_{seen}
$\binom{a}{0.1}$	$\varnothing$
$\binom{b}{2.5}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}$
$\binom{d}{2.3}\binom{e}{2.1}\binom{c}{1.4}\binom{f}{0.3}$	$\binom{a}{0.1}\binom{b}{2.5}$
$\binom{k}{3.2}\binom{e}{2.1}\binom{i}{2.0}\binom{c}{1.4}\binom{j}{1.0}\binom{f}{0.3}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{d}{2.3}$
$\binom{e}{2.1}\binom{i}{2.0}\binom{e}{1.4}\binom{j}{1.0}\binom{f}{0.3}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{d}{2.3}\binom{k}{3.2}$
爬山算法	
L	L_{seen}
$\binom{a}{0.1}$	$\varnothing$
$\binom{b}{2.5}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}$
$\binom{e}{2.1}\binom{f}{0.3}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}\binom{b}{2.5}$
$\binom{f}{0.3}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{e}{2.1}$
$\binom{m}{3.5}\binom{l}{1.5}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{e}{2.1}\binom{f}{0.3}$
$\binom{l}{1.5}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{e}{2.1}\binom{f}{0.3}\binom{m}{3.5}$
最佳优先搜索的集束版本，$N=2$	
L	L_{seen}
$\binom{a}{0.1}$	$\varnothing$
$\binom{b}{2.5}\binom{d}{2.3}\binom{c}{1.4}$	$\binom{a}{0.1}$
$\binom{k}{3.2}\binom{e}{2.1}\binom{i}{2.0}\binom{g}{1.9}\binom{c}{1.4}\binom{j}{1.0}\binom{f}{0.3}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{d}{2.3}$
$\binom{i}{2.0}\binom{g}{1.9}\binom{c}{1.4}\binom{j}{1.0}\binom{f}{0.3}$	$\binom{a}{0.1}\binom{b}{2.5}\binom{d}{2.3}\binom{k}{3.2}\binom{e}{2.1}$

按照表 3.4 给出的方式，用纸笔手动模拟算法的流程是一个很好的练习。我们还建议读者仔细研究一下，找出这些算法的行为有哪些不同，以及为什么不同。

3.2.7　昂贵的评价

在实际应用中，评价任何状态与最终状态的距离可能并不容易。很多时候，仅仅依靠一些数学公式是不可能做到的。即使可以设计出这样的公式，工程师也可能选择前瞻性策略；为了获得单个状态的值，程序可能需要对后代的数百个子状态进行重复计算。最后，在某些情况下，评价状态的唯一方法是通过系统实验，而这可能会更加昂贵。

这些状态评价的高昂成本可能会成为工程师考虑的另一个关键因素，他们希望为手头的任务确定最合适的搜索算法。

控制问题

如果你在回答下列任何问题时遇到困难，那么请返回阅读前文的相应部分。

- 讨论了评价函数的主要困难：局部峰值、多态等价、主观性、高成本。
- 前瞻性策略是什么意思？它的优点和缺点是什么？
- 解释集束搜索的原理，以及它的宽度（即参数 N）对其行为的实际影响。

3.3　A* 和 IDA*

人们总是缺乏耐心的。每当解决问题的过程在经过一系列漫长的步骤后似乎毫无进展时，一般人就会对当前的方法失去兴趣并放弃它，转而采用其他方法。这种缺乏一致性的情况并不像道德家想象得那么糟糕。类似的行为已被证明在某最强大的搜索算法中是有用的。

3.3.1　动机

考虑如图 3.5 所示的迷宫。从左边进入入口后（见图 3.5 中的箭头），智能体将以最少的步骤到达右边的出口。

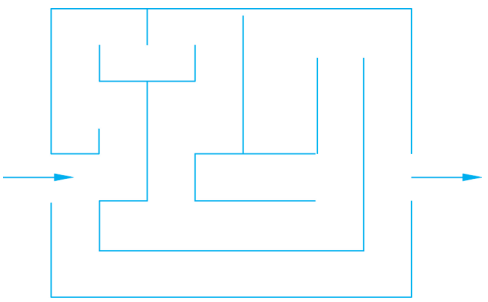

图 3.5　仅有到最终状态的距离还不够。同样重要的是，到达任一中间状态所积累的代价是多少

很容易看出，最短的路径是智能体在进入后马上转弯的那条。然而，这不是启发式搜索推荐的第一步，启发式搜索的评价函数（要最小化）会度量智能体到目标的距离。这样的搜索首先会向右，而不是向下，因为向右移动会缩短距离，而向下移动会增加距离。不幸的是，向右移动将使智能体走上一条不可避免的曲折而漫长道路，至少可以说是一种低效的尝试。我们意识到爬山算法和最佳优先搜索算法在这里不会有很好的效果。

人类解决问题的方式是不同的。在一条毫无希望的道路上走了一段时间后，我们失去

了信心，迟早会回到某个更早的地方，因为在那里我们可以选择其他方向转弯。这可能会让人想起最佳优先搜索，但这种相似性只是表面上的。如果某个较早的状态比当前状态的任何子状态更接近出口，则最佳优先搜索将回溯。然而，人类的动机是首先考虑迄今为止搜索所产生的代价。

3.3.2 代价函数

本节中介绍的方法的主要创新之处在于，不仅要最小化状态 s 与目标的距离，而且要最小化从初始状态到 s 中所累积的代价。在一个简单的程序中，比如前文中提到的迷宫，这些代价可以用起点到 s 的过程中执行的操作（搜索操作符）的数量来确定。稍后将讨论评价确定这些代价的更复杂方法。

3.3.3 A* 算法

被称为 A* （读作“A 星”）的技术依赖于评价函数和代价函数。如果评价函数返回 $g(s)$，而代价函数返回 $h(s)$；如果两者具有相同的权重，那么搜索智能体会希望最小化 $g(s)+h(s)$。注意，在 A* 中，评价函数返回的值将被最小化（而在前面提过的搜索技术中，工程师可以在最小化和最大化之间进行选择）。表 3.5 中的伪代码总结了刚才描述的原理。

表 3.5 A* 算法的伪代码：最佳优先搜索版本

输入：包含初始状态的列表 L，之前访问过的状态的空列表 L_{seen}。
评价函数 $g(s)$，衡量状态 s 到目标的距离。
代价函数 $h(s)$，衡量在达到状态 s 之前所产生的代价。
1. 设 s 是 L 中具有 $g(s)+h(s)$ 最低值的状态。
2. 如果 s 是一个最终状态，则成功终止。
3. 应用状态 s 所有合法的搜索操作符，从而获得其子状态。
4. 忽略那些已经在 L_{seen} 中的子状态，将剩余的放在 L 中。
5. 将 s 从 L 中去掉，并将其放在 L_{seen} 中。
6. 若 $L=\varnothing$，以失败终止，否则返回步骤 1。

在某些应用场景中，不能假设这两个因素（搜索代价和状态到最终状态的距离）是同等重要的。在这种情况下，程序员可能更喜欢下面这个更加通用的公式：$c_g g(s)+c_h h(s)$。这里系数 c_g 和 c_h 的值反映了这两个组成部分的相对重要性。

3.3.4 数值举例

图 3.6 显示了我们从图 3.4 中已经知道的同一个搜索树。然而，这次得到的各个状态的值为 $g(s)+h(s)$；这意味着我们把从初始状态到当前状态 s 所需的步数添加到评价函数中。

例如，在图 3.4 中，状态 g 的值是 $g(g)=0.9$。可以看到，从树的根节点经过两步到达 g，$h(g)=2$，我们确定这个状态的值为 $0.9+2=2.9$。这就是图 3.6 中所示的数值。所有其他状态的值都是这样得出的，这很容易验证。

3.3.5 A* 的两个版本

注意，表 3.5 中的算法只是最佳优先搜索算法的一个小改进。这实际上是工程实践中

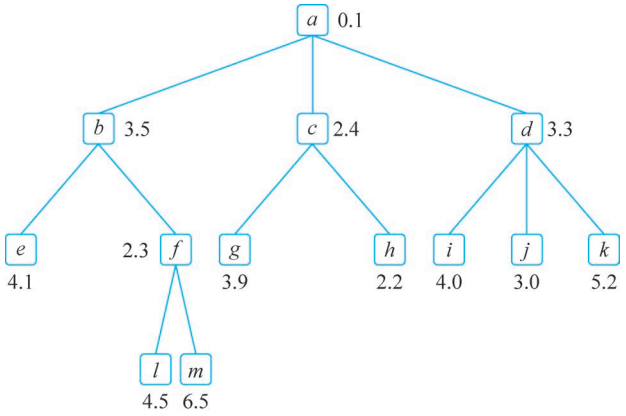

图 3.6　这与图 3.4 中的搜索树几乎相同。唯一的区别是状态值，由评价函数提供的状态值随着每个状态的深度增加而增加

最常见的 A* 的版本。

可以发现，很容易将同样的算法修改（在评价函数返回的值的基础上增加搜索代价）应用于爬山算法上，因此有时也会使用这个版本的算法。

3.3.6　更复杂的代价函数

为简单起见，本节到目前为止假设搜索代价可以通过从初始状态到当前状态的过程中所执行的操作数来很好地近似——换句话说，可以通过搜索树中遍历过的边的数量来估算。然而在现实世界中，事情可能没有这么简单。

在某些应用场景中，每个操作都带有不同代价。在这种情况下，给搜索树中的每条边标注一个数字是有意义的，因为这个数字可以量化这个边所代表的搜索操作的代价。然后，把从根节点到代表 s 的节点所经过的所有边的代价相加，就可以得到与状态 s 相关的总代价。

3.3.7　跳跃式技术

有时，工程师发现不可能设计评价函数，因为给定状态和最终状态之间的任何相似性都只是推测的，而量化它会产生误导。在这种情况下，更安全的做法可能是完全忽视评价函数 $g(s)=0$，而只寻求最小化代价 $h(s)$。由于历史原因，这种方法有时称为跳跃式技术（leaps and bounds）。

3.3.8　IDA*

最终的启发式搜索技术是 A* 的迭代深化版本。其动机与前面盲搜中的情况相同（参见第 2 章）。

我们的想法是将 A* 的爬山版本运行到一定深度上。如果没有找到解，则进行重复搜索且最大深度增加 1。虽然爬山版本的 A* 在这里似乎更自然，但我们也可以将迭代深化应用于最佳优先搜索版本的搜索。

控制问题

如果你在回答下列任何问题时遇到困难，那么请返回阅读前文的相应部分。

- 在什么情况下，搜索代价是一个不可忽视的重要标准？
- 解释 A^* 的基本原理，并讨论它的简单变动，比如灵活的代价函数，以及确定搜索代价相对于评价函数的相对重要性的方法。
- 讨论跳跃式技术背后的动机和原理。你将如何实现 IDA^*？

3.4 模拟退火

细心的读者已经注意到，到目前为止讨论的启发式搜索算法只不过是经典盲搜系列算法的变体。不过，还有一种替代方法：一种受到冶金学中某些已知物理过程启发的技术。

3.4.1 生长无缺陷晶体

假设你决定制造完美的，或者至少是近乎完美的硅晶体。为此，你需要硅原子处在它们的最低能量排列中——即众所周知的，以晶格为主体的方式排列硅晶体。这种排列可以通过所谓的退火来获得。

这个过程包括两个阶段。在第一个阶段，材料被加热到很高的初始温度。在第二个阶段，这种被加热的材料缓慢冷却。量子力学解释了在这个冷却阶段发生了什么。在每一个时刻，这组原子发现自己处于一种称为状态 s 的特定排列中。这个状态会受到随机影响的改变，每次重新排列原子，就会产生一个新的状态 s_n。有时，新的状态是稳定的；有时则不稳定，那么这种排列可能会恢复到之前的状态 s。

3.4.2 正式视图

新状态能否保持，或者系统是否能恢复到以前的状态，取决于总能量 e。我们知道，自然中更倾向于低能量状态。这意味着，如果 $E(s_n)<E(s)$，那么系统将保持新的状态 s_n，直到下一次随机变化产生影响。在相反的情况下，新状态的能量高于前一状态的能量，即 $E(s_n)>E(s)$，那么事情就更有趣了。在大多数情况下，s_n 是不稳定的，系统会恢复到之前较低的能量状态 s。但是，也有一定的概率，即系统会保持较高的能量状态，直到下一次随机变化的影响到来。

系统保持在高能态的概率取决于温度。温度越高，系统保持在高能态的概率就越大。在初始阶段，当温度很高时，系统是非常不稳定的。随着温度的下降，系统停留在高能态的概率降低，这意味着系统的大多数随机改变而导致低能态。物理学家已经阐明了以下原理：如果温度下降得很慢，那么原子几乎总是排列成完美的晶体。

总之，经过一段加热期之后，再经过一段精心控制的冷却期，就会产生大且几乎没有缺陷的晶体。

3.4.3 AI 视角

刚刚描述的过程让我们想起了我们在讨论启发式搜索时的经验。在那里，任务是到达一个定义明确的最终状态（这里是完美晶体的最低能量状态），但是为了达到这个状态，问题解决者必须经过许多次优状态。

这一点在滑方块谜题的例子中得到了说明，这也是最佳优先搜索、集束搜索和前瞻性评

价等策略被引入的原因。这些技术的威力在于它们能够在次优状态下进行问题解决。

制造完美硅晶体的雄伟目标似乎与人工智能搜索的目标没有多大关系。不过，其基本过程的高效相当耐人寻味。难怪计算机科学家试图将这些原理转化为一种算法——而且他们成功了！

3.4.4　简化视角下的模拟退火

这项技术的完整版可以在一些更高级的介绍性文章中找到。然而，即使是一个大大简化的模拟退火公式，也将有助于工程师处理具有挑战性的问题。

让我们按照以下方式修改爬山算法：在选择下一个状态时，智能体大部分时间会选择指向具有最佳值子状态的搜索操作符。然而，智能体有时会优先选择次优的子状态。假设评价函数度量给定状态与最终状态的距离，这意味着程序应该最小化它（就像大自然会寻求最小化实际物理状态的能量一样）。用 v 表示当前状态的值，用 v_n 表示所选子状态的值。设 k 为常数，T 为温度，这些参数的具体作用将在后面详细说明。采取选择该子状态的操作的概率由以下公式计算：

$$P=\begin{cases} e^{-\frac{v_n-v}{kT}}, & v_n \geqslant v \\ 1, & \text{其他} \end{cases} \tag{3.1}$$

3.4.5　状态值的影响

式(3.1)规定，如果新状态的值小于当前状态的值（$v_n<v$），则接受新状态的概率为100%。如果新状态的值大于当前状态的值（$v_n>v$），则仍然可以接受新状态。该情况发生的概率取决于状态值之间的差 v_n-v：状态值的增加程度越大，接受新状态的概率越低。当然，工程师还必须指示计算机程序在 $v_n=v$ 的情况下该怎么做。然而，从本节的角度来看这无关紧要。

3.4.6　温度影响

较劣状态被接受的概率是由系统的温度 t 控制的。开始时温度很高，随着时间的推移逐渐降低。仔细看一下式(3.1)就会发现，温度越高，接受较劣状态的可能性越大。随着温度下降，甚至可能接近零度时，更劣的状态很少会被接受。还要注意指数分母中的常数 k。它的作用是控制公式对 T 变化的敏感性。[①]

模拟退火成功与否取决于温度降低的速度，冶金学家知道冷却过程应该是缓慢的。

但是，多慢才算慢呢？另外，这个过程开始的初始温度应该是多少呢？

3.4.7　冷却

理论家已经推导出了一个保证成功的公式，但温度的降低是如此缓慢，以至于这个过程似乎永远都不会结束，因此我们必须做出妥协。如果能在合理的时间范围内完成搜索，注重

① 在原物理公式中，k 是玻耳兹曼常数。

实际的工程师会接受次优操作(即,过程可能只找到“近乎最优”的解)。虽然有人已经提出了一些基于严格分析的非常复杂的公式,但经验表明,即使用以下简单公式(T_k 表示第 k 个时间步长的温度,另外 $\alpha \in [0.90, 0.99]$),通常也能获得良好的结果:

$$T_{k+1} = \alpha T_k \tag{3.2}$$

在基准程序中,每次状态变化后温度都会降低。在这种情况下,工程师更喜欢较高的 α 值(接近 1)。其他场景中可能要求温度只在预定的步骤数之后降低,例如,在每 10 个状态变化之后。在这种情况下使用较低的 α 值,可能接近 0.9。

3.4.8 初始温度

初始温度 T_0 必须足够高,以便系统在早期做出许多“错误的决定”。这基本上是作者在这里能提供的唯一建议。对于初始温度具体应该是多少,似乎并没有什么非常固定的普遍规定。在实际应用中,可以通过预实验来指导选择。

控制问题

如果你在回答下列任何问题时遇到困难,那么请返回阅读前文的相应部分。

- 解释冶金学中用来生长无缺陷硅晶体的退火过程的原理。
- 描述实现模拟退火的计算方法。如何生成随机的状态变化?一个新状态什么时候会被保持,什么时候不会被接受?
- 关于温度的初始化你能说些什么,以及该如何降低温度?

3.5 背景知识的作用

前面章节中描述的搜索技术解决了一些有趣的难题,并且在许多重要任务中被证明是有用的。但我们必须时刻保持警惕。

早期的成功往往会阻止批评,它在某种程度上影响了我们正确看待事物的能力。当启发式搜索的思想第一次被提出时,一些狂热者甚至声称这种方法掌握着所有计算智能的关键。

今天,我们有了更好的认识。人们对搜索技术无所不能的信心已经动摇,专家现在一致认为,还有很多工作要做。下面的例子旨在说明,当面对真正的智能时,搜索的思想在哪些方面受到了限制。

3.5.1 AI 搜索解决的幻方问题

图 3.7 是流行的幻方谜题。任务是用 1～9 的 9 个整数填充 9 个方格,使每个方格包含不同的数字,并且所有列、行和主对角线的数字和都相等。这就是最终状态的定义。初始状态是图片左边的 3×3 的空方块;搜索操作符将一个整数放在其中一个空方块中。在学习了前面的章节后,读者会发现很容易编写一个程序来处理这个任务。

这样的计划是否有效?让我们来看看。假设搜索从在左上角放置一个随机的整数开始。有 9 种可能性。对于其中的每一个,下一个方格(例如,上面第一行中间的那一个)将接受其余 8 个整数中的一个。这产生了 9×8=72 种组合。这样继续下去,我们就会发现,盲搜可能要面对多达 9! 种填充幻方的不同方式!这显然太多了。诚然,如果找到处理各种对

称性的方法(例如,将棋盘旋转 90°,或将其水平倒置),则可以大大减小这个数字。此外,一个精心设计的评价函数将帮助我们采用更有效的启发式搜索,这可能会进一步加快这一过程。尽管如此,成本仍然很高;在这种场景中,搜索算法的机械式表现似乎是几乎毫无希望的。

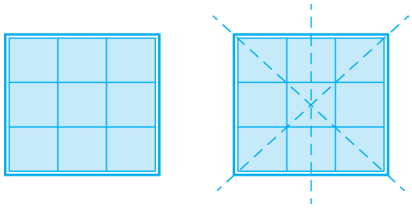

图 3.7　幻方:任务是用 1～9 的整数填充 9 个方格,使所有列、所有行和主对角线上的数字和都相同

3.5.2　数学家解决幻方问题

人类会用不同方法解决问题。数学家首先观察到 1 ～ 9 的整数的和是 $\sum_{i=1}^{9} i = 45$。如果 3 列中的每一列都有相等的和,那么这个和必为 45/3 = 15。这显然减少了填充幻方的方案数量。

数学家推理的下一步如图 3.7 的右侧所示。假设在中间行、中间列和两个主对角线中填充方格。这包含了所有的 9 个方格,但是正方形中心的方格(让我们用 x 表示它的值)被填充了 4 次。如果减去这个方格出现的 4 次中的 3 次($3x$),那么每个方格只被用到一次,从前面讨论我们知道,总和是 45。

如果不减去中心方格的 3 次多余填充,则有 4 个三元组(列、行和两条对角线)。因为每个的总和是 15,所以总数必须是 $4 \times 15 = 60$。再次用 x 表示中心方格的值。减去它的 3 次“多余”出现得到 $60 - 3x = 45$。这里可得到 $x = 5$。最后得出结论,5 就是要放在中心方格中的整数。

剩下的就很简单了。在左上角可以放置其余 8 个整数中的任何一个。它们中的每一个都决定了右下角的数值。例如,如果左上角填充 9,那么右下角就必须是 15－9－5＝1。这样一来,如何填充上面第一行中间的方格就只有 6 种可能。假设选择 2。那么右上角得到 15－9－2＝4;幻方的其余部分也将以类似方式填充,始终遵守行、列和主对角线之和为 15 的规则。

3.5.3　课程:背景知识的好处

上一段的分析将可能性的数量减少到 $8 \times 6 = 48$。这比原来的 9! 少多了!以至于我们甚至不需要计算机;这个问题现在可以用纸和笔在几分钟内解决。根据这一经验,我们开始怀疑传统的搜索方法确实过于机械,效率低下。我们需要更好的东西。

由于背景知识的引入,使搜索空间在相当程度上变小了。从表面上看,幻方可能是一个典型的搜索问题。然而,我们注意到,一个更优雅的方法会考虑到幻方的几何形状,进而一个更有效的解决方案会依赖于数学家的想法。我们开始意识到,解决问题的计算机应该以某种方式被赋予知识,并且有能力以有意义的方式处理这些知识。粗暴的数字计算是相当浪费的。

3.5.4 数独中的分支因子

另一个值得我们注意的方面是分支因子。大致看一下图 3.8 中的数独问题，就会发现最上面一行只使用了 9 个整数中的 3 个。一个粗心的初学者可能会假设，比如说，对于数字 3 右边的方格，可以填充剩下的 6 个整数中的任何一个。当然，这不是一种合理的处理方式。首先，8 是不合格的，因为它已经出现在同一列（第三行）中。同样，6 和 9 也是不合格的，因为它们已经出现在给定的 3×3 正方形中。这样就只剩下 3 种可能了，而不是 6 种。

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

图 3.8 数独的分支因子比看起来要小

然而，这还没完。看到有 3 种可能性可供选择，却没有任何迹象表明应该选择哪一种，人类玩家会找一个选项数量较少的方格。通常，我们可以找到一个只允许一种可能性的方格，这相当于分支因子减少到 $b=1$，这确实是一个显著的减少！我们再一次观察到，这种令人印象深刻的冗余可能性减少是由于我们对数独本质的理解，而不是计算机的蛮力。

3.5.5 斑马谜题

在其他谜题中，对知识表示和知识利用的需求更加明显。以第 1 章中的斑马谜题为例。已知一系列关于房子、人、动物等的事实，并被要求回答一个只能从这些事实中推断出来的问题。试图通过传统的搜索来做到这一点是徒劳的，除非我们找到一种方法，它能将任何理性的人都可以获得的一些知识传递给机器。这可能要采取“如果-那么”规则的形式通过计算机程序成功地进行操作，这就可以被称为自动推理。

斑马谜题的存在为我们提供了另一个证据，使我们确信仅靠搜索是不够的。在许多有意义的领域，搜索必须得到某种知识的帮助。如何对这些知识进行编码，以及如何在解决问题的环境中利用这些知识，这些问题将在第 9 章开始讨论。然而在此之前，关于搜索本身还有许多问题值得讨论。

控制问题

如果你在回答下列任何问题时遇到困难，那么请返回阅读前文的相应部分。

- 讨论数学家解决幻方谜题的方法与搜索技术的区别。
- 熟练的程序员会如何减小数独游戏中的平均分支因子？你能编写一个程序来实现这种简化吗？
- 提出一个通过经典搜索方法解决斑马谜题的方法。该如何表示问题中的事实？什么将构成搜索状态？
- 什么是背景知识？它如何帮助解决问题？

3.6 连续域

早期的 AI 主要关注每个状态允许有限个操作的领域，并且状态由离散变量描述。然而，在许多实际应用中，描述和操作都只能用来自连续域的数字来表征。让我们来看看这对搜索意味着什么。

3.6.1　连续域举例

图 3.9 展示了一个名为 Mexican Hat 的函数。平面上的任一点[x,y]定义了一种状态,纵坐标给出了其对应的评价函数的值。我们可以看到状态分布于一个连续域。

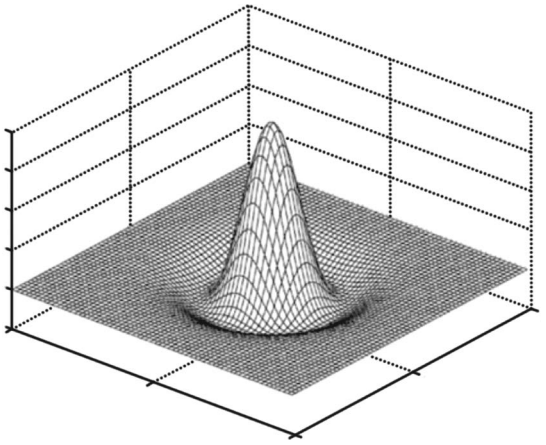

图 3.9　一个连续域。平面中的每一对[x,y]代表一种状态。评价函数映射在纵轴上

假设从平面上随机生成的初始状态开始。目标是找到使评价函数达到最大值的状态。[①] 假设采用爬山算法:在每个状态下,智能体希望沿着其最陡的梯度以一种保证函数增长最快的方式来修改 x 和 y 的值。因此,操作是连续的。状态的定义域不一定是二维的;状态可以用任意数量的变量来描述。它们中的一些可以是离散的,但如果至少有一个是连续的,则认为该域是连续的。

3.6.2　离散化

到目前为止,我们所看到的搜索技术都期望一个进行离散操作的智能体:在每个状态下,操作(搜索操作符)都是从有限的备选项集合中选择的。相比之下,连续域中的操作数量是无限的,因为描述状态的变量有无穷多的可能变化。

处理这种情况最简单的方法是离散化。其思想是将每个变量的定义域划分为有限的区间。因此,在如图 3.9 所示的例子中,可以用 10×10 的正方形网格覆盖平面。在每个时刻,智能体发现自己处于由这些方格之一定义的状态。然后可以通过命令定义操作,“沿着 x 轴移动 N 个方格,沿着 y 轴移动 M 个方格。”

离散化允许我们使用已经熟悉的启发式搜索。不可避免地,这种便利性是以准确性的损失为代价的:给定方格内的所有状态都被认为是相等的。通常,这种限制是可以接受的,因为工程师可能不会坚持寻找最优解,而满足于“做得相当好”。这就是追求完美的数学家和追求实际解决方案的工程师的世界观的区别。

3.6.3　梯度上升与神经网络

除了离散化之外,还可以使用数学上简洁且不会丢失信息的方法。如果定义评价函数

① 在本例中,函数最大值对应的状态为[0,0],即坐标系的原点。

的公式是已知的，那么一种可选方法是通过微积分找到函数的最陡梯度。

通过将函数的一阶导数设为 0 来识别局部极值；剩下要做的就是决定极值点代表的是最大值还是最小值。这种方法被称为梯度上升，如果想要函数的最小值则为梯度下降。

在不深入讨论细节的情况下，让我们顺便提一下，梯度上升技术是神经网络的典型技术。由于该技术的深度学习大获成功，这项技术最近变得流行起来。然而，关于神经网络和深度学习的讨论属于机器学习领域，而不是核心 AI 的教科书应该过多讨论的。如果读者知道连续域可以通过爬山算法来解决，并且相关的技术涉及一些微积分，那就足够了。

3.6.4 群体智能算法

不过，这里还有另一种选择。第 8 章将介绍一系列强大的技术，统称为群体智能。它们功能强大，不需要微积分，而且与深度学习一样受欢迎和流行。

控制问题

如果你在回答下列任何问题时遇到困难，那么请返回阅读前文的相应部分。

- 操作的离散域和连续域之间的主要区别是什么？
- 离散化如何帮助我们使用经典的人工智能技术？离散化的主要缺点是什么？
- 本节简要介绍了处理连续域而不需要离散化的两种方法。你还记得它们的名字吗？

3.7 熟能生巧

为了加深理解，不妨尝试以下练习、思考题和计算机作业。

- 对于该章中的每个谜题，至少设计两个不同的评价函数。例如，南岸传教士的数量会是一个好的指标吗？如果不是，为什么？关键是要习惯这样的想法：通常存在许多替代方案，有些很简单，有些很复杂；但很少有哪个方案在指导搜索过程的方式上是令人完全满意的。
- 假设一个谜题如图 3.10 中的搜索树所示。展示爬山算法、最佳优先搜索算法、A^* 算法和集束搜索算法($N=2$)访问状态的顺序。
- 已经实现了一些盲搜算法的读者会发现很容易修改它，使其变成爬山算法和最佳优先搜索算法。其中最困难的部分是确定其评价函数。尝试通过一个简单的编程练习来做到这一点。
- 在已经实现爬山算法和最佳优先搜索算法的程序后，在各种测试平台上运行实验来比较它们的行为。写一篇两页的文章，总结你的观察。
- 另一个有用的研究将比较由不同评价函数所指导的启发式搜索的效率。重点是观察到设计良好的评价函数可以显著加快求解过程。
- 将之前练习中开发的最佳优先搜索程序转换为 A^* 技术。唯一需要增加的是一个代价函数，该函数把达到给定状态前累积的代价相加。在最简单的实现中，这意味着监控状态的深度——从初始状态到当前状态的路径上的操作数。在更复杂的实现中，程序员还将考虑路径上各个操作可能产生的不同代价。
- 如何将模拟退火应用到本章的玩具场景？怎样确定初始温度？
- 编写一个计算机程序实现模拟退火。进行一项实验研究，将程序的行为与更传统的

启发式搜索技术进行比较。探讨式(3.1)中常数 k 的实际影响。

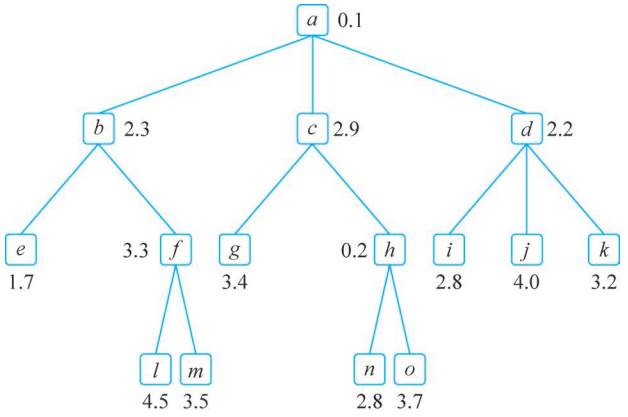

图 3.10　一个用于练习的搜索树示例。这些数字表示各个状态与最终状态的评价相似度(要最大化)

3.8　结语

启发式搜索的历史始于 Minsky(1961 年)提出的爬山算法。代价函数的加入(在 A* 中),传统上被认为是 Hart、Nilsson 和 Raphael(1968 年)提出的。模拟退火的基本思想是由多位作者提出的,但 Kirpatrick、Gelatt 和 Vechi(1983 年)通常被认为是通过成功地将该技术应用于推销员问题[①]而取得主要突破的人。他们似乎也是第一次使用模拟退火这个术语的人。

一段时间以来,盲搜和启发式搜索技术被认为是解决计算机智能之谜的可靠答案。其应用成功的范围远远超出了玩具场景和初级谜题,这些成果进一步增强了人们对其的乐观与信心。其中,令人印象最深刻的也许就是自动规划系统,这是第 5 章讨论的主题。另一个令人兴奋的成果是模拟退火技术的引入:如果一种原理足够好,可以用其生长出无缺陷的晶体,那么为什么不能将其应用于人工智能问题呢?

然而,科学家和工程师逐渐意识它们的某些局限性,如 3.5 节中提到的那些。首先,评价函数并不能真正传达给机器足够的相当于人类的理解;另外还需要其他方法来减小分支因子并指导求解过程。数独的例子说明了这一点,而幻方的例子更能说明这一点,我们看到一个数学家遵循着与搜索方法本质无关的推理线来很好地解决问题。最后一个例子——斑马谜题——进一步加深了我们的怀疑,即智能程序应该能够用明确编码的知识进行推理。

在本书的后面,有几章将专门讨论如何表示知识,以及如何将其应用于自动推理的问题。我们将特别关注处理不精确性、不确定性和未知性的方法,所有这些都与人类的思维方式密不可分。

然而,目前我们还没有解决问题的全部。首先,关于帮助计算机解决游戏问题的算法有很多值得讨论。正如我们将在第 4 章看到的,这些技术与迄今为止看到的技术有很大的不

① 有关推销员问题的更多信息,可参见 5.5 节。

同。然后，为了让我们相信经典搜索尽管有种种局限性，但可以成功地解决现实问题，第 5 章将讨论在自动规划领域中使用经典搜索技术的一些简单方法。

之后，介绍了经典搜索的主要替代方法。第 6 章解释了流行的遗传算法的特性与使用，第 7 章介绍了人工生命中涌现性的思想，第 8 章在解释一些称为群体智能的技术时，以这些问题为基础。