第3章

搜索技术

搜索技术是人工智能的基本求解技术之一，在人工智能各领域中被广泛应用。早期的人工智能程序与搜索技术的联系更为紧密，几乎所有的早期人工智能程序都以搜索为基础。例如，A. Newell 和 H. A. Simon 等人编写的 LT(Logic Theorist)程序，J. Slagle 编写的符号积分程序 SAINT，A. Newell 和 H. A. Simon 编写的 GPS(General Problem Solver)程序，H. Gelernter 编写的 Geometry theorem-proving machine 程序，R. Fikes 和 N. Nilsson 编写的 STRIPS(Stanford Research Institute Problem Solver)程序以及 A. Samuel 编写的 Chechers 程序等，都使用了各种搜索技术。

现在，搜索技术渗透在人工智能各领域中，例如，在专家系统、自然语言理解、自动程序设计、模式识别、机器人学、信息检索和博弈等领域都广泛使用搜索技术。搜索技术具有如此丰富的应用领域的原因在于：广义地讲，人工智能的大多数问题都可以看作是搜索问题。

搜索问题的求解一直是人工智能的核心问题，它通常是先将问题转换为某个可供搜索的空间(简称“搜索空间”)，然后在该空间内寻找一个解。问题一般由初始条件、目标和操作集合这 3 部分组成。根据问题类型不同，问题可被转换为状态空间图、And-Or 图(与或图)等不同类型的搜索空间。问题的解在搜索空间中也相应地具有不同的形式。本章将了解根据问题的目标判断出搜索算法是否已经发现解的具体方法。根据问题的目标，搜索技术用于处理如何在搜索空间上以较短的时间发现较优的解。由于所面临的大多数现实问题对应的问题空间巨大，人们往往无法指导搜索算法沿着确定的方向搜索，所以，通常需要用搜索算法的不断试探来解决。本章讨论搜索空间的形式和最优解的确定，以及如何从问题空间的初始结点经过变换得到解等内容。因此，本章首先讨论一些基于状态空间图的通用算法，包括蛮力搜索法和启发式搜索法，然后讨论适用于问题归约的搜索方法，最后讨论博弈问题的智能搜索方法。

3.1 概述

人工智能所要解决的问题大部分是结构不良或非结构化问题,对这样的问题一般不存在成熟的求解算法。对于给定的问题,智能系统的行为一般是找到能够达到所希望目标的动作序列,并使其所付出的代价最小、性能最好。基于给定的问题,问题求解的第一步是问题的建模。搜索就是找到智能系统的动作序列的过程。搜索算法的输入是问题的实例,输出是表示为动作序列的方案。一旦有了方案,系统就可以执行该方案所给出的动作了。这一阶段称为执行阶段。因此,求解一个问题主要包括3个阶段:问题建模、搜索和执行。本章主要讨论搜索的问题,而其他两个阶段的问题会在其他章节中讨论。

一般给定一个问题就是确定该问题的一些基本信息,由以下4个部分组成。

(1) 初始条件集合:定义了问题的初始状态。

(2) 操作符集合:把一个问题从一个状态变换为另一个状态的动作集合。

(3) 目标检测函数:用来确定一个状态是不是目标。

(4) 路径费用函数:对每条路径赋予一定费用的函数。

其中,初始条件集合和操作符集合定义了搜索空间。

在人工智能中,搜索问题一般包括两个重要的问题:搜索什么,在哪里搜索。搜索什么通常指的就是目标,而在哪里搜索就是指搜索空间。搜索空间通常是指一系列状态的集合,因此也称为状态空间。与通常的搜索空间不同,人工智能中大多数问题的状态空间在问题求解之前不是全部知道的。所以,人工智能中的搜索可以分成两个阶段:状态空间的生成阶段和在该状态空间中对目标状态的搜索。由于一个问题的整个状态空间可能会非常大,在搜索之前生成整个空间会占用太大的存储空间。为此,状态空间一般是逐渐扩展的,"目标"状态是在每次扩展的时候进行判断的。

一般地讲,搜索方法可以根据是否使用启发式信息分为盲目搜索方法和启发式搜索方法,也可以根据搜索空间的表示方式分为状态空间搜索方法和与或图搜索方法。状态空间搜索是用状态空间法来求解问题所进行的搜索,与或图搜索是指用问题规约方法来求解问题时所进行的搜索。状态空间法和问题规约法是人工智能中最基本的两种问题表示方法。

盲目搜索方法一般是指从当前的状态到目标状态需要走多少步或者每条路径的花费并不知道,所能做的只是可以区分出哪个是目标状态,因此,它一般是按预定的搜索策略进行搜索。由于这种搜索总是按预定的路径进行,没有考虑到问题本身的特性,所以这种搜索具有很大的盲目性,效率不高,不便于复杂问题的求解。启发式搜索方法是在搜索过程中加入了与问题有关的启发式信息,用于指导搜索朝着最有希望发现目标状态的方向前进,加速问题的求解并找到最优解。显然盲目搜索不如启发式搜索效率高,但是由于启发式搜索需要和问题本身特性有关的信息,而对于很多问题这些信息很少,或者根本就没有,或者很难抽取,所以盲目搜索仍然是很重要的一类搜索方法。

在搜索问题中,主要的工作是找到正确的搜索算法。搜索算法一般可以通过下面4个标准来评价。

(1) 完备性:如果存在一个解答,该策略是否保证能够找到?

(2) 时间复杂性：需要多长时间可以找到解答？

(3) 空间复杂性：执行搜索需要多少存储空间？

(4) 最优性：如果存在不同的几个解，该算法是否可以发现最高质量的解？

搜索算法制定了状态空间或问题空间扩展的方法，也决定了状态或问题的访问顺序。不同的搜索算法在人工智能领域的命名也不同。例如，在状态空间为一棵树的问题上有两种基本的搜索算法。如果首先扩展根结点，然后扩展根结点生成的所有结点，再扩展这些结点的后继，如此反复下去，则这种算法称为宽度优先搜索，另一种方法是，在树的最深一层的结点中扩展一个结点，只有当搜索遇到一个死亡结点(非目标结点并且是无法扩展的结点)的时候，才返回上一层选择其他的结点搜索。这种算法称为深度优先搜索。然而，无论是宽度优先搜索还是深度优先搜索，结点的遍历顺序都是固定的，即一旦搜索空间给定，结点遍历的顺序就固定了。这种类型的遍历称为“确定”的，也就是盲目搜索。而对于启发式搜索，在计算每个结点的参数之前无法确定先选择哪个结点扩展，这种搜索一般称为“非确定”的。

3.2 盲目搜索方法

下面介绍几种常用的盲目搜索方法，首先介绍生成再测试法这个一般的算法框架，然后介绍性能较好的迭代加深算法。

3.2.1 生成再测试法

最简单的盲目搜索方法是“生成再测试”(generate and test)方法。该方法的算法描述如下。

```
Procedure Generate&Test
    Begin
        Repeat
            生成一个新的状态,称为当前状态;
        Until 当前状态=目标;
    End
```

显然，上述算法在每次 Repeat-Until 循环中都生成一个新的状态，并且只有当新的状态等于目标状态的时候才退出。在该算法中最重要的部分是新状态的生成。如果生成的新状态不可扩展，则该算法应该停止，为了简单起见，在上述算法中省略了这一部分。

宽度优先搜索算法和深度优先搜索算法可以看做是生成再测试方法的两个具体版本。它们的区别是生成新状态的顺序不同。假设问题空间是一棵树，则深度优先搜索总是优先生成并测试深度增加的结点，而宽度优先搜索则总是优先搜索同一深度的结点。深度优先搜索和宽度优先搜索具有两个主要的特点：(1)它们只能用于求解搜索空间为树的问题，如果用于处理存在环的搜索空间，则它们都有可能陷入无限循环而无法停止；(2)宽度优先搜索能够保证找到路径长度最短的解(最优解)，而深度优先搜索无法保证这一点。第一个特点是我们不希望的，3.3.2 节介绍的最好优先搜索算法可以弥补此缺点。

对于第二个特点，应该认识到宽度优先搜索的优势是以巨大的存储为代价的。假设问题空间中每个结点平均有 b 个子结点，目标结点的深度为 d，则宽度优先搜索在最坏情况下需要存储 $O(b^d)$ 个结点；相对而言，深度优先搜索则仅需存储 $O(d)$ 个结点。那么，能否设计一种搜索方法结合宽度优先搜索保证最优性的优势与深度优先搜索在存储上的优势？下面介绍的迭代加深搜索方法就具有此性质。

3.2.2 迭代加深搜索

对于深度 d 比较大的情况，深度优先搜索可能沿着一个不含目标结点的分枝探寻很长时间，在找不到解的同时还浪费了资源。一种较好的方法是对搜索的深度进行控制，这就是有界深度优先搜索方法的主要思想。有界深度优先搜索过程总体上按深度优先算法方法进行，但对搜索深度给出一个深度限制 d_m，当深度达到了 d_m 的时候，如果还没有找到解答，就停止对该分枝的搜索，换到另外一个分枝进行搜索。

对于有界深度优先搜索策略，有以下几点需要说明。

(1) 在有界深度优先搜索算法中，深度限制 d_m 是一个很重要的参数。当问题有解，且解的路径长度小于或等于 d_m 时，则该算法一定能够找到解。但是和深度优先搜索一样，这并不能保证最先找到的是最优解，此情况下的有界深度而搜索是完备的但不是最优的。但是当 d_m 取得太小，而解的路径长度大于 d_m 时，则搜索过程中就找不到解，此情况下的搜索过程甚至是不完备的。

(2) 深度限制 d_m 不能太大。当 d_m 太大时，搜索过程会产生过多的无用结点，既浪费了计算机资源，又降低了搜索效率。

(3) 有界深度优先搜索的主要问题是深度限制 d_m 的选取。该值也被称为状态空间的直径，如果该值设置得比较合适，则会得到比较有效的有界深度优先搜索。但是对很多问题，预先无法知道该值到底为多少，只有在该问题求解完成后才能确定出深度限制 d_m，而那时确定的 d_m 对搜索算法没有意义。为了解决上述问题，可采用如下的改进方法：先任意设定一个较小的数作为 d_m，然后按有界深度算法搜索，若在此深度限制内找到了解，则算法结束；如在此限制内没有找到问题的解，则增大深度限制 d_m，继续搜索。此方法称为迭代加深搜索。

迭代加深搜索(iterative deepening search)是一种回避选择最优深度限制问题的策略，它是试图尝试所有可能的深度限制：深度首先为 0，然后为 1，然后为 2，……，一直进行下去。如果初始深度为 0，则该算法只生成根结点，并检测它。如果根结点不是目标，则深度加 1，通过典型的深度优先搜索算法，生成深度为 1 的树。同样，当深度限制为 m 时，它将生成深度为 m 的树。

迭代加深搜索过程描述如下：

```
Procedure Iterative-deeping
Begin
    For d=1 to∞Do
    Begin
        从初始结点执行深度限制为 d 的有界深度优先搜索;
```

```
        如果找到解,则过程结束;
        如果本次迭代中访问的所有结点的深度都小于 d,则过程结束;
    End
End
```

通过分析可以发现,迭代加深搜索看起来会很浪费资源,因为它在深度限制为 $d+1$ 的迭代过程中将重复搜索深度限制为 d 的迭代访问过的结点。然而对于很多问题,这种多次的扩展负担实际上很小,直觉上可以想象,如果一棵树的分枝系数很大,几乎所有的结点都在最底层上,则对于上面各层结点扩展多次对整个系统的影响不是很大。

宽度优先搜索、深度优先搜索和迭代加深搜索都是生成再测试算法的具体版本。迭代加深搜索结合了宽度优先搜索和深度优先搜索的优点。表 3-1 总结了宽度优先搜索、深度优先搜索、有界深度搜索和迭代加深搜索的主要特点。

表 3-1 几个盲目搜索算法的特点对比

标准	宽度优先搜索	深度优先搜索	有界深度搜索	迭代加深搜索
时间	b^d	b^m	b^l	b^d
空间	b^d	b^m	b^l	b^d
最优	是	否	否	是
完备	是	否	如果 $l>d$,是	是

注:b 是分枝系数,d 是解答的深度,m 是搜索树的最大深度,l 是深度限制。

3.3 启发式搜索

前面讨论的搜索方法都是按事先规定的、根据结点的深度制定的路线进行搜索,搜索过程机械,具有较大的盲目性,生成的无用结点较多,搜索空间较大,因而效率不高。除了结点的深度信息之外,如果能够利用结点暗含的与问题相关的一些特征信息来预测目标结点的存在方向,并沿着该方向搜索,则有希望缩小搜索范围,提高搜索效率。利用结点的特征信息来引导搜索过程的一类方法称为启发式搜索。

任何一种启发式搜索算法在生成一个结点的全部子结点之前,都将使用算法设计者提供的评估函数判断这个“生成”过程是否值得进行。评估函数通常为每个结点计算一个整数值,称为该结点的评估函数值。通常,评估函数值小的结点被认为是值得进行“生成”过程。按照惯例,将“生成结点 n 的全部子结点”称为“扩展结点 n”。启发式搜索可以用于两种不同方向的搜索:前向搜索和反向搜索。前向搜索一般用于状态空间的搜索,从初始状态出发向目标状态方向进行;反向搜索一般用于问题规约中,从给定的目标状态向初始状态进行。为这两种搜索方法设计评估函数时应采用不同的思路,3.3.1 节和 3.4.3 节分别解释这个特点。

3.3.1 启发性信息和评估函数

在搜索过程中,关键的是在下一步选择哪个结点进行扩展,选择的方法不同就形成了

不同的搜索策略。如果在选择结点时能充分利用它与问题有关的特征信息估计出它对尽快找到目标结点的重要性，就能在搜索时选择重要性较高的结点，以便快速找到解或者最优解。称这样的过程为启发式搜索。“启发式”实际上是一种“大拇指准则”(thumb rules)：即在大多数情况下是成功的，但不能保证一定成功的准则。

用来评估结点重要性的函数称为评估函数。评估函数 $f(n)$ 对从初始结点 S_0 出发、经过结点 n 到达目标结点 S_g 的路径代价进行估计。其一般形式为

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

其中，$g(n)$ 表示从初始结点 S_0 到结点 n 的已获知的最小代价；$h(n)$ 表示从 n 到目标结点 S_g 的最优路径代价的估计值，它体现了问题的启发式信息。所以，$h(n)$ 被称为启发式函数。$g(n)$ 和 $h(n)$ 的定义都要依据当前处理的问题的特性，$h(n)$ 的定义更需要算法设计者的创造力。下面介绍在九宫图问题上 $g(n)$ 和 $h(n)$ 的定义方法。

在九宫图问题中，有一个 3×3 的棋盘，其中 8 个格子上放着带数字的卡片，1 个格子空白，每张卡片可以被移动到与它相邻的空白格子，求解的目标是将棋盘上卡片的初始格局通过一系列移动卡片的操作变换到目标格局。图 3-1 是九宫图问题一个实例，其中 S_0 表示初始格局，S_g 表示目标格局。评估函数可以表示为

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

其中，$g(n)=d(n)$ 定义为结点在 n 搜索树中的深度；$h(n)=w(n)$ 定义为“结点 n 中不在目标状态中相应位置的数字卡片个数”，$h(n)$ 包含了问题的启发式信息。可以看出，一般来说某结点 n 的 $h(n)$ 越大，即“不在目标位”的数字卡片个数越多，说明目标结点离 n 越远，进而可以认为“扩展”n 就相对不重要。在图 3-1 中，对于初始结点 S_0，由于 $g(S_0)=0$，$h(S_0)=5$，因此 $f(S_0)=5$。

S_0=

2	8	3
1	6	4
7		5

S_g=

1	2	3
8		4
7	6	5

图 3-1　九宫图问题

$f(n)$ 由 $g(n)$ 和 $h(n)$ 两部分组成，启发式搜索算法可以使用 $f(n)$ 的不同组合，进而表现出不同的特性。例如，有的算法使用 $f(n)=g(n)$，有的算法使用 $f(n)=h(n)$，有的算法使用 $f(n)=g(n)+h(n)$。下面将介绍最好优先搜索(best-first search)算法使用不同形式的 f 所表现出来的特点。

3.3.2　最好优先搜索算法

在 3.2 节已经提及宽度优先搜索和深度优先搜索不适用于状态空间存在环的情况，下面介绍一种称为最好优先搜索算法(best-first search)的算法框架，该方法能处理图。为了处理环，最好优先搜索算法用 OPEN 表和 CLOSED 表记录状态空间中那些被访问过的所有状态。这两个表中的结点及它们关联的边构成了状态空间的一个子图，称为搜索图。OPEN 表存储一些结点，其中每个结点 n 的启发式函数值已经计算出来，但是 n 还没有被“扩展”。CLOSED 表存储一些结点，其中每个结点已经被扩展。该类算法每次迭代从 OPEN 表中取出一个较优的结点 n 进行扩展，将 n 的每个子结点根据情况放入 OPEN 表。算法循环直到发现目标结点或者 OPEN 表为空。算法中的每个节点带有一个父指针，该指针用于合成解路径。

最好优先搜索算法的具体描述如下：

```
Procedure Graph-Search
Begin
    建立只含初始结点 S0 的搜索图 G,计算 f(S0);将 S0 放入 OPEN 表;将 CLOSED 表初始化为空。
    While OPEN 表不空 Do
    Begin
        从 OPEN 表中取出 f(n)值最小的结点 n,将 n 从 OPEN 表中删除并放入 CLOSED 表。
        If n 是目标结点 Then 根据 n 的父指针指出从 S0 到 n 的路径,算法停止。
        Else
        Begin
            扩展结点 n
            If 结点 n 有子结点
            Then
            Begin
                (1) 生成 n 的子结点集合{mi}把 mi 作为 n 的子结点加入到 G 中,并计算 f(mi)。
                (2) If mi 未曾在 OPEN 和 CLOSED 表中出现,Then 将它们配上刚计算过的 f
                    值,将 mi 的父指针指向 n,并把它们放入 OPEN 表。
                (3) If mi 已经在 OPEN 表中,Then 该结点一定有多个父结点。在这种情况下,
                    比较 mi 相对于 n 的 f 值和 mi 相对于其原父指针指向的结点的 f 值,若前
                    者不小于后者,则不做任何更改;否则将 mi 的 f 值更改为 mi 相对于 n 的 f
                    值,mi 的父指针更改为 n。
                (4) If mi 已经在 CLOSED 表中,Then 该结点同样也有多个父结点。在这种情
                    况下,比较 mi 相对于 n 的 f 值和 mi 相对于其原父指针指向的结点的 f 值。
                    如果前者不小于后者,则不作任何更改;否则将 mi 从 CLOSED 表移到 OPEN
                    表,置 mi 的父指针指向 n。
                (5) 按 f 值从小到大的次序,对 OPEN 表中的结点进行重新排序。
            End
        End
    End
End
```

上述搜索算法生成一个明确的图 G(称为搜索图)和一个 G 的子集 T(称为搜索树),图 G 中的每一个结点也在树 T 上。搜索树是由结点的父指针来确定的。G 中的每一个结点(除了初始结点 S_0)都有一个指向 G 中一个父辈结点的指针。该父辈结点就是那个结点在 T 中的唯一父辈结点。算法中(3)、(4)步保证对每一个扩展的新结点,其父指针的指向是已经产生的路径中代价最小的。

3.3.3 贪婪最好优先搜索算法

最好优先搜索算法是一个通用的算法框架。如果将该框架中的 $f(n)$实例化为 $f(n)=h(n)$,则得到一个具体的算法,称为贪婪最好优先搜索(Greedy Best-First Search,GBFS)算法。可以看出,GBFS 算法在判断是否优先扩展一个结点 n 时仅以 n 的启发值为依据。n 的启发值越小,表明了从 n 到目标结点的代价越小,因而 GBFS 算法沿着 n 所在的分枝搜索就越可能发现目标结点。因此,GBFS 算法一般可以较快地计算出问题的解。

但是，GBFS 算法得出的解是否是最优的？考虑如下情况，OPEN 表中有两个结点 n 和 n'，其中 $g(n)=5$，$h(n)=0$，$g(n')=3$，$h(n')=1$，而且 n 和 n' 的 h 值分别是它们与目标结点的真实距离，在此情况下，GBFS 将扩展 n 而不是 n'。显然，经过 n 发现的解的代价高于经过 n' 发现的解的代价，所以 GBFS 返回的不是最优解。仔细分析最好优先算法的流程可以发现，当 $f(n)=h(n)$ 时，其中的步骤(3)和(4)将不会对 n 的信息做改变。3.3.4 节内容将说明步骤(3)和(4)是为了保证算法的最优性而设置的。与 GBFS 算法相对，假如最好优先搜索算法中的 $f(n)$ 被实例化为 $f(n)=g(n)$，则得到宽度优先搜索算法。读者可以在图的最短路径问题上将 $g(n)$ 定义为源结点到 n 的路径长度，分析此命题的正确性。

可以看出，$h(n)$ 影响算法发现解的速度，$g(n)$ 影响得到解的最优性，下面介绍的 A 算法和 A^* 是使用 $f(n)=g(n)+h(n)$ 的最好优先搜索算法，它们综合考虑了时间效率和解的质量。其中，A^* 算法使用的 $h(n)$ 具有更严格的性质。

3.3.4　A 算法和 A^* 算法

如果最好优先搜索算法中的 $f(n)$ 被实例化为 $f(n)=g(n)+h(n)$，则称之为 A 算法。进一步细化，如果启发函数 h 满足对于任一结点 n，$h(n)$ 的值都不大于 n 到目标结点的最优代价，则称此类 A 算法为 A^* 算法。A^* 算法在一些条件下能够保证找到最优解，即 A^* 算法具有最优性。下面首先以九宫图为例(图 3-2)介绍 A 算法的运行过程，然后介绍文献[34]对 A^* 算法最优性的分析。

A 算法采用 3.3.1 节定义的评估函数判断每个结点的重要性。在该算法运行的初始时刻，OPEN 表中只有初始结点，因此我们扩展它，得到图 3-2 中的第二层结点，将这些结点全部放入 OPEN 表。在第二次迭代过程中，A 算法选择 OPEN 表中具有最小 f 值为 $1+3=4$ 的结点扩展，得到第三层的 3 个结点，并将它们放入 OPEN 表。在第三次迭代中，A 算法选择 OPEN 表中 f 值为 $2+3=5$ 的结点进行扩展。在第四次迭代中，A 算法选择 OPEN 表中 f 值为 $2+3=5$ 的另一个结点进行扩展。在第五次迭代中，A 算法选择 OPEN 表中 f 值为 $3+2=5$ 的结点进行扩展。在第六次迭代中，A 算法选择 OPEN 表中 f 值为 $4+1=5$ 的结点进行扩展。在第七次迭代中，A 算法选择 OPEN 表中 f 值为 $5+0=5$ 的结点进行扩展。通过此例可以发现，A 算法相对于宽度优先搜索和深度优先搜索都具有优势。

但是，由于对启发函数 h 没有任何限制，A 算法不能保证找到最优解。经研究发现，A 算法在如下 3 个条件成立时能够保证得到最优解：

(1) 启发函数 h 对任一结点 n 都满足 $h(n)$ 不大于 n 到目标的最优代价。

(2) 搜索空间中的每个结点具有有限个后继。

(3) 搜索空间中每个有向边的代价均为正值。

为了表明此类 A 算法的重要性，将此类 A 算法称为 A^* 算法，称上述 3 个条件为 A^* 算法的运行条件。

对 h 的限制可以更为正式地表述如下：令 $h*$ 是能计算出任意结点 n 到目标的最优代价的函数，称之为“完美启发函数”。如果 $\forall n: h(n) \leqslant h*(n)$，则称 h 为可采纳的启发

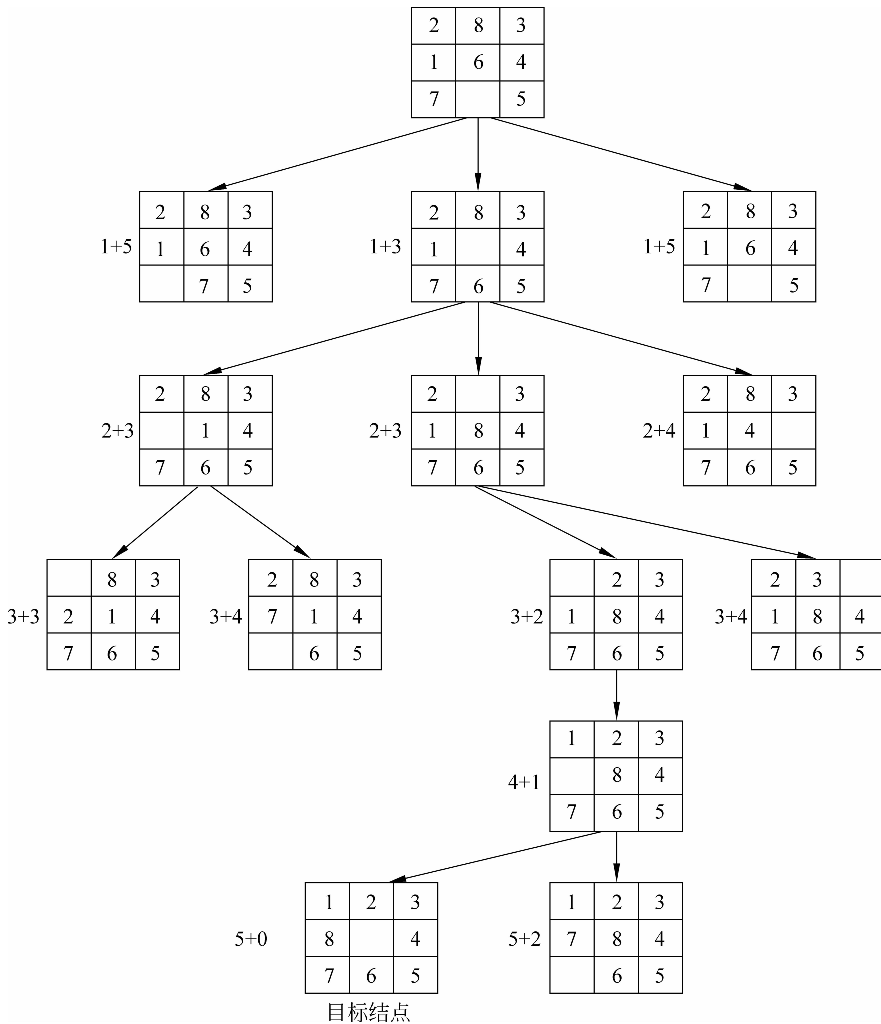

图 3-2　九宫图问题的全局择优搜索树

函数(admissible heuristic function),或者称 h 是可采纳的,或者简称为可纳的。此外,也引入函数 $g*$,它能计算从开始结点到任意结点的最优代价。定义估价函数:$f*(n)=g*(n)+h*(n)$。这样 $f*(n)$就是从起始结点出发经过结点 n 到达目标结点的最佳路径的总代价。

把估价函数 $f(n)$和 $f*(n)$相比较,$g(n)$是对 $g*(n)$的估价。$h(n)$是对 $h*(n)$的估价。在这两个估价中,尽管 $g(n)$容易计算,但它不一定就是从起始结点 S_0 到结点 n 的真正的最短路径的代价,很可能从初始结点 S_0 到结点 n 的真正最短路径还没有找到,所以一般都有 $g(n)\geq g*(n)$。但应注意,A^* 算法的步骤(3)和(4)保证了如果发现 n 的更好的 $g(n)$值,则以此值作为 n 的最新的 $g(n)$,并相应地修改 n 的父指针,步骤(4)还在 n 已被扩展的情况下将 n 移回 OPEN 表,使得 n 会被再次扩展。

在图 3-1 所示的九宫图问题中,尽管并不知道 $h*(n)$具体为多少,但在定义 $h(n)=w(n)$时保证了 h 的可采纳性。这是因为 $w(n)$统计的是“不在目标状态中相应位置的数

字卡片个数”，这相当于假定把不在目标位置的一个数字卡片移动到它的目标位置仅需要一步，而实际情况下把一个数字卡片移到目标位置应该需要一步以上。所以 $w(n)$ 必然不大于 $h*(n)$。应当指出，同一问题启发函数 $h(n)$ 可以有多种设计方法。在九宫图问题中，还可以定义启发函数 $h(n)=p(n)$，其中 $p(n)$ 为结点 n 的每一数字卡片与其目标位置之间的欧几里得距离总和。显然有 $p(n)\leqslant h*(n)$，相应的搜索过程也是 A^* 算法。然而 $p(n)$ 比 $w(n)$ 有更强的启发性信息，因为由 $h(n)=p(n)$ 构造的启发式搜索树比 $h(n)=w(n)$ 构造的启发式搜索树结点数要少。这一结论在后面关于 A^* 算法特性的讨论中说明。

现在给出一些关于算法性质的定义，为了叙述方便，将一个算法记作 M。

完备性：如果存在解，则 M 一定能找到该解并停止，则称 M 是完备的。

可纳性：如果存在解，则 M 一定能够找到最优的解，则称 M 是可纳的。

优越性(Dominance)：一个算法 M_1 称为优越于另一个算法 M_2，指的是如果一个结点由 M_1 扩展，则它也会被 M_2 扩展，即 M_1 扩展的结点集是 M_2 扩展的结点集的子集。

最优性：在一组算法中一个算法 M 称为最优的，如果 M 比其他算法都优越。

下面的定理 3.1 说明了 A^* 算法的完备性和可纳性。为了证明该定理，我们首先介绍引理 3.1。

引理 3.1　在 A^* 算法停止之前的每次结点扩展前，在 OPEN 表上总是存在具有如下性质的结点 $n*$：

(a) $n*$ 位于一条解路径上。

(b) A^* 算法已得出从初始结点 S_0 到 $n*$ 的最优路径。

(c) $f(n*)\leqslant f*(S_0)$。

证明：为证明此引理在 A^* 算法的每次结点扩展前都成立，只需证明：(1)本引理在 A^* 算法初始执行时成立；(2)若本引理在一个结点被扩展之前成立，则在该结点被扩展之后本引理同样成立。按照此思路，将采用归纳法进行证明。为叙述方便，以下简称 A^* 算法为 A^*。

归纳基础：在 A^* 算法在第 1 次结点扩展前(即，S_0 被选择进行扩展之前)，S_0 在 OPEN 表中，S_0 位于一条最优解路径上(因为所有的解路径都以为 S_0 起点)，并且 $A*$ 已得知从 S_0 到 S_0 的最优路径。此外，根据 f 的定义，

$$f(S_0)=g(S_0)+h(S_0)=h(S_0)\leqslant h*(S_0)=g*(S_0)+h*(S_0)=f*(S_0)$$

因此，在第 1 次结点扩展前，S_0 就是满足引理结论的 $n*$。

归纳步骤：假设引理在第 m 次($m\geqslant 0$)结点扩展后成立，证明本引理在第 $m+1$ 次结点扩展后仍成立。

假定 A^* 算法在扩展 m 个结点后，OPEN 表中存在一个结点 $n*$，A^* 算法已知从 S_0 到 $n*$ 的最优路径。那么，若 $n*$ 在第 $m+1$ 次扩展中未被选择，则它在第 $m+1$ 次扩展后是满足引理要求的结点 $n*$，在此情况下引理得证。另一方面，若 $n*$ 在第 $m+1$ 次扩展时被选择，则 $n*$ 的每一个未在 OPEN 表和 CLOSED 表中出现的子结点都将被放入 OPEN 表，而且，这些新的子结点中必然存在一个结点(记为 n_p)位于最优解路径上(因为经过 $n*$ 的最优解路径必然在经过 $n*$ 后再经过 $n*$ 的某个子结点，所以 n_p 必然存在)。n_p 也满足性质(b)，即 A^* 已得出从 S_0 到 n_p 的最优路径，该路径记为 P_1：由到达 $n*$ 的最