第3章 搜索策略

从工程应用的角度，开发人工智能技术的一个主要目的就是解决非平凡问题，即难以用常规（数值计算、数据库应用等）技术直接解决的问题。这些问题的求解依赖于问题本身的描述和应用领域相关知识的应用。从广义上说，人工智能问题都可以看作是一个问题求解的过程。因此问题求解是人工智能的核心问题，其要求在给定条件下寻求一个能解决某类问题且能在有限步内完成的算法。

按解决问题所需的领域特有知识的多寡，问题求解系统可以划分为两大类：知识贫乏系统和知识丰富系统。前者必须依靠搜索技术去解决问题，后者则求助于推理技术。

搜索直接关系到智能系统的性能与运行效率，因而美国人工智能专家尼尔逊（N. J. Nilsson，1974 年）把它列为人工智能研究的四个核心问题（知识的模型化和表示；常识性推理、演绎和问题求解；启发式搜索；人工智能系统和语言）之一。

现在，搜索技术渗透在各种人工智能系统中，可以说没有哪一种人工智能系统应用不到搜索方法。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。

本章首先讨论搜索的有关概念，然后着重介绍状态空间的知识表示和搜索策略，主要有一般图的搜索、盲目搜索、启发式搜索和与或图的搜索，最后讨论博弈问题的智能搜索算法。

3.1 引言

智能系统所要解决的问题是各种各样的，其中，大部分是结构不良或非结构化的问题，对这样的问题一般没有算法可以求解，而只能是利用已有的知识一步步地摸索前进。在此过程中，存在着如何寻找可用知识的问题。即如何确定推理路线，使其付出的代价尽可能的少，而问题又能得到较好的解决。例如在推理中，可能存在多条路线都可实现对问题的求解，这就又存在按哪一条路线进行求解以获得较高的运行效率的问题。

因此，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。搜索就是找到智能系统的动作序列的过程。

在智能系统中，即使对于结构性能较好、理论上有算法可依的问题，由于问题本身的复杂性以及计算机在时间、空间上的局限性，有时也需要通过搜索来求解。

在人工智能中，搜索问题一般包括两个重要的问题：搜索什么以及在哪里搜索。前者通常指的是搜索目标，而后者通常指的是“搜索空间”。搜索空间通常是指一系列状态的汇

集，因此也称为状态空间。和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该状态空间中对所求问题状态的搜索阶段。

根据在问题求解过程中是否运用启发性知识，搜索被分为盲目搜索和启发式搜索两种方法。

1. 盲目搜索（又称为非启发式搜索）

盲目搜索是指在问题的求解过程中，不运用启发性知识，只按照一般的逻辑法则或控制性知识，在预定的控制策略下进行搜索，在搜索过程中获得的中间信息不用来改进控制策略。由于搜索总是按预先规定的路线进行，没有考虑到问题本身的特性，这种方法缺乏对求解问题的针对性，需要进行全方位的搜索，而没有选择最优的搜索途径。因此，这种搜索具有盲目性，效率较低，容易出现“组合爆炸”问题。

典型的盲目搜索有深度优先搜索和宽度优先搜索。

2. 启发式搜索

启发式搜索是指在问题的求解过程中，为了提高搜索效率，运用与问题有关的启发性知识，即解决问题的策略、技巧、窍门等实践经验和知识，来指导搜索朝着最有希望的方向前进，加速问题求解过程并找到最优解。

典型的启发式搜索有 A 算法和 A^* 算法。

在搜索问题中，主要的工作是找到正确的搜索策略。一般搜索策略可以通过下面 4 个准则来评价。

(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答，该策略是否可以发现最高质量的解答。

搜索策略反映了状态空间或问题空间的扩展方法，也决定了状态或问题的访问顺序。搜索策略不同，人工智能中搜索问题的命名也不同。例如，考虑一个问题的状态空间为一棵树的形式。如果首先扩展根节点，然后扩展根节点生成的所有节点，然后是这些节点的后继，如此反复下去。这就是宽度优先搜索。另一种方法是，在树的最深一层的节点中扩展一个节点。只有当搜索遇到一个死亡节点(非目标节点且无法扩展)的时候，才返回上一次选择其他节点搜索。这就是深度优先搜索。无论是宽度优先搜索还是深度优先搜索，节点遍历的顺序一般都是固定的，即一旦搜索空间给定，节点遍历的顺序就固定了。这类遍历称为“确定性”的，也就是盲目搜索。而对于启发式搜索，在计算每个节点的参数之前无法确定先选择哪个节点扩展，这种搜索一般也称为非确定的。

3.2 基于状态空间图的搜索技术

搜索最适合于设计基于一个操作算子集的问题求解任务，每个操作算子的执行均可使问题求解更接近于目标状态，搜索路径将由实际选用的操作算子的序列构成。本节将从状

态空间这一概念入手,介绍一般图搜索算法、最常见的两种盲目搜索和启发式搜索算法,最后讨论提高状态图搜索效率的方法。

3.2.1 图搜索的基本概念

1. 显式图与隐式图

为了求解问题,需要把有关的知识存储在计算机的知识库中,包括以下两种存储方式。

(1) 显式存储。把与问题有关的全部状态空间图,即相应的全部有关知识(叙述性知识、过程性知识和控制性知识)都直接存入知识库,这种存储方式称为显式存储或显式图。

(2) 隐式存储。只存储与问题求解有关的部分知识(即部分状态空间)。这种存储方式称为隐式存储。在求解过程中,由初始状态出发,运用相应的知识,逐步生成所需的部分状态空间图,通过搜索推理,逐步转移到要求的目标状态,只需在知识库中存储局部状态空间图,这种图称为隐式图。

为了节约计算机的存储容量,提高搜索推理效率,通常采用隐式存储方式,进行隐式图搜索推理。

2. 图搜索的基本思想

图搜索就是一种在图中寻找路径的方法。这种寻找过程从图中的初始节点开始,至目标节点为止。其中,初始节点和目标节点分别代表产生式系统的初始数据库和满足终止条件的数据库。方法是先把问题的初始状态作为当前状态,选择适用的算符对其进行操作,生成一组子状态,检查目标状态是否在其中出现。若出现,则搜索成功,找到了问题的解;若不出现,则按某种搜索策略从已生成的状态中再选一个状态作为当前状态。重复上述过程,直到目标状态出现或者不再有可供操作的状态及算符时为止。

3.2.2 状态空间搜索

用搜索技术来求解问题的系统均定义一个状态空间,并通过适当的搜索算法在状态空间中搜索解答或解答路径。状态空间搜索的研究焦点在于设计高效的搜索算法,以降低搜索代价并解决组合爆炸问题。

1. 什么是状态空间图

先看下面的例子。

例 3.1 钱币翻转问题。设有 3 个钱币,其初始状态为(反、正、反),欲得的目标状态为(正、正、正)或(反、反、反)。问题是允许每次只能且必须翻转一个钱币,连翻 3 次。问能否达到目标状态。

要求解这个问题,可以通过引入一个三维变量将问题表示出来。设三维变量为:$Q=[q_1,q_2,q_3]$,式中 $q_i=0(i=1,2,3)$表示钱币为正面,$q_i=1(i=1,2,3)$表示钱币为反面。则 3 个钱币可能出现的状态有 8 种组合:

$Q_0=(0,0,0)$,$Q_1=(0,0,1)$,$Q_2=(0,1,0)$,$Q_3=(0,1,1)$,$Q_4=(1,0,0)$,$Q_5=(1,0,1)$,$Q_6=(1,1,0)$,$Q_7=(1,1,1)$

这时，问题可以表示为图 3.1。在此图中，表示了全部可能的 8 种组合状态及其相互关系，其中每个组合状态可认为是一个节点，节点间的连线表示了两节点的相互关系（例如从 Q_5 节点到 Q_4 节点间的连线表示要将 $q_3=1$ 翻成 $q_3=0$，或反之）。现在的问题就是要从初始状态 Q_5，在图中经过适当的路径（即连线），找到目标状态 Q_0 或 Q_7。从图中可以清楚地看出，从 Q_5 不可能经过 3 步到达 Q_0，即不存在从 Q_5 到达 Q_0 的解。但从 Q_5 出发到达 Q_7 的解有 7 个，它们是 *aab*、*aba*、*baa*、*bbb*、*bcc*、*cbc* 和 *ccb*。

从这个问题的求解过程可看到，对某个具体问题，可经过抽象变为某个有向图中寻找目标或路径的问题。人工智能科学中，把这种描述问题的有向图称为状态空间图，简称状态图。其中状态图中的节点代表问题的一种格局，一般称为问题的一个状态；边表示两节点之间的某种联系，如它可以是某种操作、规则、变换、算子或关系等。在状态图中，从初始节点到目标节点的一条路径，或者所找的目标节点，就是相应问题的一个解。其一般描述如图 3.2 所示。

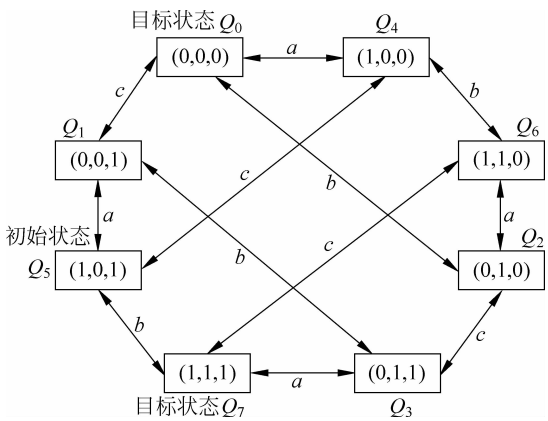

图 3.1　3 枚钱币问题的状态空间图

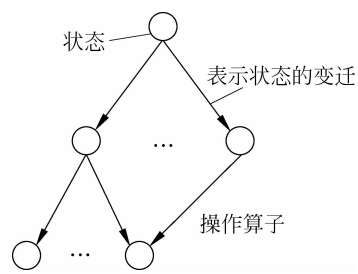

图 3.2　状态空间图的一般描述

在现实生活中，不论是智力问题（如梵塔问题、旅行商问题、八皇后问题、传教士过河问题等），还是实际问题（如定理证明、演绎推理，机器人行动规划等）都可以归结为在某一状态图中寻找目标或路径的问题。所以，状态图是一类问题的抽象表示。

2. 问题的状态空间表示法

状态空间表示法是指用“状态”和“操作”组成的“状态空间”来表示问题求解的一种方法。

（1）状态（state）。状态是指为了描述问题求解过程中不同时刻下状况（例如初始状况、事实等叙述性知识）间的差异，而引入的最少的一组变量的有序组合。它常用矢量形式表示

$$\boldsymbol{S}=[s_0,\ s_1,\ s_2,\cdots]^{\mathrm{T}}$$

其中，$\boldsymbol{s}_i$（$i=0,1,2\cdots$）叫分量。当给定每个分量的值 s_{ki}（$i=0,1,2,\cdots$）时，就得到一个具体的状态 $\boldsymbol{s}_k$

$$\boldsymbol{s}_k=[s_{k0},s_{k1},s_{k2},\cdots]^{\mathrm{T}}$$

状态的维数可以是有限的，也可以是无限的。另外，状态还可以表示成多元数组或其他

形式。状态主要用于表示叙述性知识。

(2) 操作(operator)。操作也称为运算符或算符,它引起状态中的某些分量发生改变,从而使问题由一个具体状态改变到另一个具体状态。操作可以是一个机械的步骤、过程、规则或算子,指出了状态之间的关系。操作用于反映过程性知识。

(3) 状态空间(state space)。状态空间是指一个由问题的全部可能状态及其相互关系(即操作)所构成的有限集合。

状态空间常记为二元组:

$$(S,O)$$

其中,S 为问题求解(即搜索)过程中所有可能到达的合法状态构成的集合;O 为操作算子的集合,操作算子的执行会导致问题状态的变迁。

这样,在状态空间表示法中,问题求解过程就转化为在图中寻找从初始状态 S_0 出发到达目标状态 S_g 的路径问题,也就是寻找操作序列 α 的问题。

作为状态空间表示的经典例,观察经典的"传教士和野人问题"。设 N 个传教士带领 N 个野人划船渡河,且为安全起见,渡河需遵从三个约束:

(1) 船上人数不得超过载重限量,设为 K 个人;

(2) 为预防野人攻击,任何时刻(包括两岸、船上)野人数目不得超过传教士;

(3) 允许在河的某一岸或者在船上只有野人而没有传教士。

为便于理解状态空间表示方法,将该问题简化为一个特例:$N=3$,$K=2$;并以变量 m 和 c 分别指示传教士和野人在左岸或船上的实际人数,变量 b 指示船是否在左岸(值 1 指示船在左岸,否则为 0)。从而上述约束条件转变为 $m+c\leqslant 2$,$m\geqslant c$。

考虑到在这个渡河问题中,左岸的状态描述(m、c 和 b)可以决定右岸的状态,所以整个问题状态就可以左岸的状态来描述,以简化问题的表示。设初始状态下传教士、野人和船都在左岸,目标状态下这三者均在右岸,问题状态以三元组(m,c,b)表示,则问题求解任务可描述为:

$$(3,3,1)\rightarrow(0,0,0)$$

在这个简单问题中,状态空间可能的状态总数为 $4\times4\times2=32$,但由于要遵守安全约束,只有 20 个状态是合法的。下面是几个不合法状态的例子:

$$(1,0,1),(1,2,1),(2,3,1)$$

鉴于存在不合法状态,还会导致某些合法状态不可达,例如状态(0,0,1)、(0,3,0)。结果,这个问题总共只有 16 个可达的合法状态。

渡河问题中的操作算子可以定义两类:$L(m,c)$、$R(m,c)$,分别指示从左岸到右岸的划船操作和从右岸回到左岸的划船操作。由于 m 和 c 取值的可能组合只有 5 个:10,20,11,01,02,故而总共有 10 个操作算子。

可以画出相应于渡河问题状态空间的有向图,如图 3.3 所示。

由于划船操作是可逆的,所以节点间的弧线有双向箭头,弧标签指示船上传教士和野人的人数,显然每个节点只能取 L 和 R 操作之一,这取决于状态变量 b 的值。

由此例可以看出:

(1) 用状态空间方法表示问题时,首先必须定义状态的描述形式,通过使用这种描述形式可把问题的一切状态都表示出来。另外,还要定义一组操作,通过使用这些操作可把问题

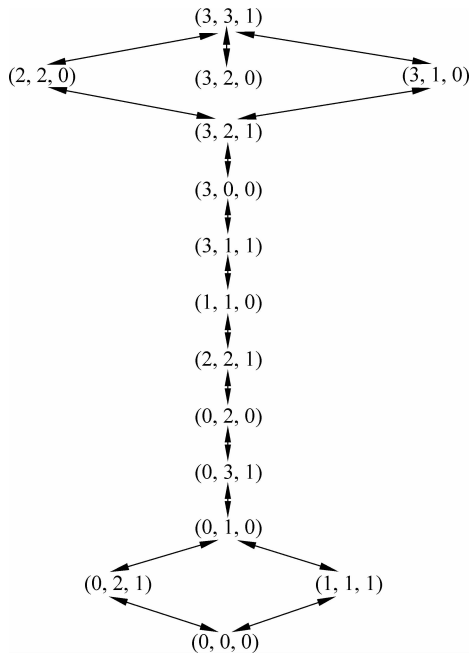

图 3.3　渡河问题的状态空间图

由一种状态转变为另一种状态。

(2) 问题的求解过程是一个不断把操作作用于状态的过程。如果在使用某个操作后得到的新状态是目标状态,就得到了问题的一个解。这个解是从初始状态到目标状态所用操作构成的序列。

(3) 要使问题由一种状态转变到另一种状态时,就必须使用一次操作。这样,在从初始状态转变到目标状态时,就可能存在多个操作序列(即得到多个解)。那么,其中使用操作最少或较少的解才为最优解(因为只有在使用操作时所付出的代价为最小的解才是最优解)。

(4) 对其中的某一个状态,可能存在多个操作,使该状态变到几个不同的后继状态。那么到底用哪个操作进行搜索呢? 这就要依赖于搜索策略了。不同的搜索策略有不同的顺序,这就是本章后面要讨论的问题。

在智能系统中,为了进行问题求解,首先必须用某种形式把问题表示出来,其表示是否适当,将直接影响到求解效率。状态空间表示法就是用来表示问题及其搜索过程的一种方法。它是人工智能科学中最基本的形式化方法,也是问题求解技术的基础。

3. 状态空间搜索的基本思想

状态空间搜索的基本思想就是通过搜索引擎寻找一个操作算子的调用序列,使问题从初始状态变迁到目标状态之一,而变迁过程中的状态序列或相应的操作算子调用序列称为从初始状态到目标状态的解答路径。搜索引擎可以设计为任意实现搜索算法的控制系统。

通常,状态空间的解答路径有多条,但最短的只有 1 条或少数几条。上述渡河问题就有无数条解答路径(因为划船操作可逆),但只有 4 条是最短的,都包含 11 个操作算子的调用。由于一个状态可以有多个可供选择的操作算子,导致了多个待搜索的解答路径。例如图 3.3

中初始状态节点就有 3 个操作算子供选用。这种选择在逻辑上称为“或”关系,意指只要其中有一条路径通往目标状态,就能获得成功解答。由此,这样的有向图称为或图,常见的状态空间一般都表示为或图,因而也称一般图。

除了少数像渡河这样的简单问题外,描述状态空间的一般图都很大,无法直观地画出,只能将其视为隐含图,在搜索解答路径的过程中只画出搜索时直接涉及的节点和弧线,构成所谓的搜索图。作为一般图搜索的再一个例子,下面分析智力游戏八数码问题。

八数码游戏在由 3 行和 3 列构成的九宫棋盘上进行,棋盘上放置数码为 1～8 的 8 个棋牌,剩下一个空格,游戏者只能通过棋牌向空格的移动来不断改变棋盘的布局。这种游戏求解的问题是:给定初始布局(即初始状态)和目标布局(即目标状态),如何移动棋牌才能从初始布局到达目标布局,如图 3.4 所示。显然解答路径实际上就是一个合法的走步序列。

为用一般图搜索方法解决该问题,首先应为问题状态的表示建立数据结构,再制定操作算子集。就以 3×3 的一个矩阵来表示问题状态,每个矩阵元素 $S_{ij} \in \{0,1,\cdots,8\}$;其中 $1 \leqslant i,j \leqslant 3$,数字 0 指示空格,数字 1～8 指示数码。于是图 3.4 中的八数码问题就可表示为矩阵形式,如图 3.5 所示。

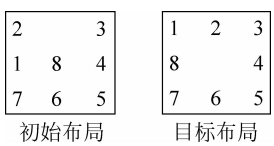

图 3.4 八数码游戏实例

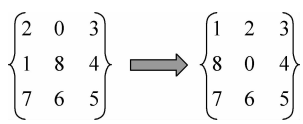

图 3.5 八数码问题的矩阵表示

定义操作算子的直观方法是为每个棋牌制定一套可能的走步:左、上、右、下 4 种移动。这样就需 32 个操作算子。简单易行的方法是仅为空格制定这 4 种走步,因为只有紧靠空格的棋牌才能移动。空格移动的唯一约束是不能移出棋盘。假设在搜索过程的每一步都能选择最合适的操作算子,则图 3.4 中的八数码问题解决时,一次搜索过程涉及的状态所构成的搜索图(这里实际是搜索树)如图 3.6 所示,其中粗线代表解决路径。

对于八数码游戏可能的棋盘布局(问题状态)总共 $9!=362\,880$ 个,由于棋盘的对称性,实际只有这个总数的一半。显然,无法直观地画出整个状态空间的一般图,但搜索图则小得多,可以图示。所以,尽管状态空间可以很大(例如国际象棋),但只要确保搜索空间足够小,就能在合理的时间范围内搜索到问题解答。

搜索空间的压缩程度主要取决于搜索引擎采用的搜索算法。换言之,当问题有解时,使用不同的搜索策略,找到解答路径时,搜索图的大小是有区别的。一般来说,对于状态空间很大的问题,设计搜索策略的关键考虑是解决组合爆炸问题。复杂的问题求解任务往往涉及许多解题因素,问题状态就可以通过解题因素的特别组来加以表示(解题因素可设计为状态变量,如传教士和野人问题中的 m、c 和 b)。所谓组合爆炸是指解题因素多时,因素的可能组合个数会爆炸性(指数级)增长,引起状态空间的急剧膨胀。例如某问题有 4 个因素,且每个因素有 3 个可选值,则因素的组合(即问题状态)有 $3^4=81$ 个;但若因素增加到 10 个,则组合的个数达 $3^{10}=3^4\times3^6=81\times729$,即状态空间扩大到 729 倍。解决组合爆炸问题的方法实际上就是选用好的搜索策略,使得只要搜索状态空间的很小部分就能找到解答。

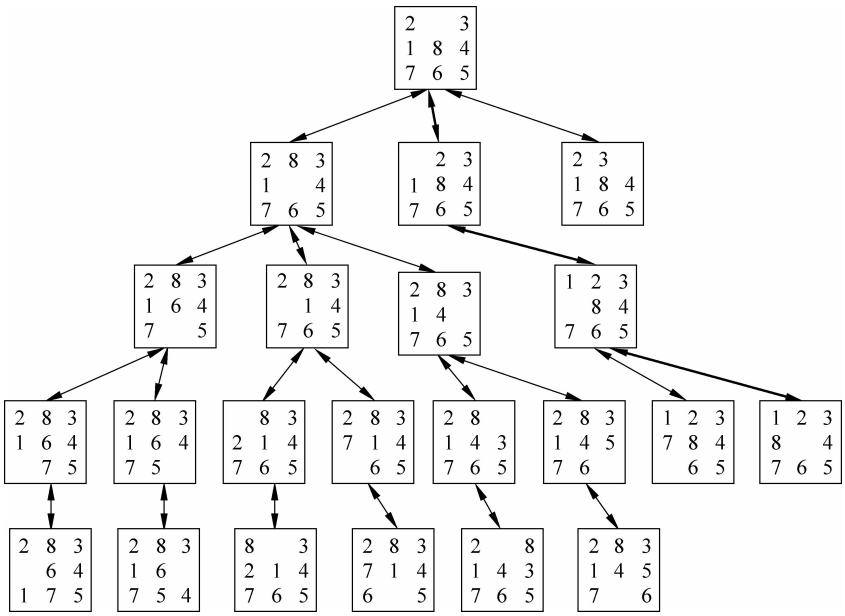

图 3.6　八数码问题的一次搜索图

3.2.3　一般图的搜索算法

下面给出图搜索的一般搜索过程。它是由尼尔逊(N. J. Nilsson)提出的一个著名的图搜索过程 GRAPH SEARCH,它是表达能力很强的一个搜索策略框架。在此过程中要用到 OPEN 表和 CLOSE 表。其中,OPEN 表用于待扩展的节点。节点进入 OPEN 表中的排列顺序是由搜索策略决定; CLOSE 表用于存放已经扩展的节点,当前节点进入 CLOSE 表的最后。

图搜索(GRAPH SEARCH)的一般过程如下:

(1) 建立一个只含有初始节点 S_0 的搜索图 G,把 S_0 放到 OPEN 表中。

(2) 建立 CLOSE 表,其初始为空表。

(3) LOOP: 若 OPEN 表是空表,则问题无解,失败并退出。

(4) 把 OPEN 表上的第一个节点移出并放进 CLOSE 表的后面,标记此节点为节点 n(在 CLOSE 表的编号栏标记)。

(5) 若 n 为一目标节点,则有解并成功退出,此解是追踪图 G 中沿着指针从 n 到 S_0 这条路径而得到的(指针将在步骤(7)中设置)。

(6) 扩展节点 n,同时生成不是 n 的祖先的那些后继节点的集合 M。把 M 的这些成员作为 n 的后继节点添入图 G 中。

(7) 对那些未曾在 G 中出现过的(即未曾在 OPEN 表上或 CLOSE 表上出现过的)M 成员设置一个通向 n 的指针。把 M 的这些成员加进 OPEN 表。对已经在 OPEN 表或 CLOSE 表上的每一个 M 成员,确定是否需要更改通到 n 的指针方向。对已在 CLOSE 表上的每一个 M 成员,确定是否需要更改图 G 中通向它的每一个后继节点的指针方向。

(8) 按某种搜索策略或按某个探试值,重排 OPEN 表。

(9) 转 LOOP。

上述过程生成一个显式图 G(称为搜索图),由返回指针确定 G 的子图 T(称为搜索树),OPEN 表中的节点是 T 的叶节点。

在第(7)步中改变返回指针的原则是:节点有多个父节点时。返回指针指向代价最小的父节点。

通过循环地执行该算法,搜索图会因不断有新节点加入而逐步长大,直到搜索到目标节点。

在搜索图中标记从子节点到父节点的指针,方便了在搜索到目标状态时快速返回解答路径:自初始状态 s 到目标状态的一个节点序列。为说明搜索算法中子节点分类和指针修改的作用,观察一下图 3.7 中的示意例。当前被扩展的节点为 n_i,它扩展出第 1 类子节点 n_1 和 n_2,第 2 类子节点 n_4 和第 3 类子节点 n_3。假设节点 n_3 和 n_4 经由新父节点 n_i 到初始状态节点 s 的路径代价比经由老父节点 n_j 的要小,则节点 n_3 和 n_4 原指向节点 n_j 的指针都移走,改为指向节点 n_i。由于 n_3 自身已扩展出子节点 n_{31} 和 n_{32},而 n_{32} 又有 2 个父节点,所以也应修改 n_{32} 指向父节点的指针(从原先指向 n_j 改为指向 n_3)。但鉴于 n_3 或许并不在最终得到的解答路径上,故这种指针修改并不值得进行。简单地把节点 n_3 放回到 OPEN 表,而不修改其子节点指针,起到了推迟修改的作用。以后一旦节点 n_3 被从 OPEN 表中取出重新扩展时,会重新扩展出 n_{32},这时 n_{32} 成为第 2 类子节点,此时再修改指针也不迟。

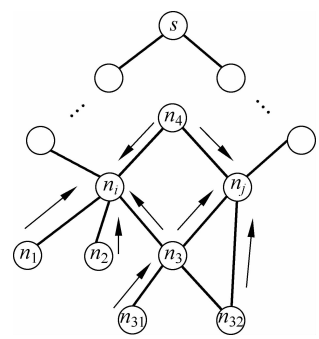

图 3.7 搜索过程中的指针修改

3.3 盲目搜索

一般图搜索算法中,提高搜索效率的关键在于优化 OPEN 表中节点的排序方式,若每次排在表首的节点都在最终搜索到的解答路径上,则算法不会扩展任何多余的节点就可快速结束搜索。所以排序方式成为研究搜索算法的焦点,并由此形成了多种搜索策略。

一种简单的排序策略就是按预先确定的顺序或随机地排序新加入到 OPEN 表中的节点,常用的方式是深度优先和宽度优先。

深度优先、宽度优先及其改进算法的共同缺点是节点排序的盲目性,由于不采用领域专门知识去指导排序,往往会在白白搜索了大量无关的状态节点后才碰到解答,所以这类搜索也称为盲目搜索。

3.3.1 宽度优先搜索

宽度优先搜索(Breadth-First Search)又称为广度优先搜索。

1. 宽度优先搜索的基本思想

宽度优先搜索是指从初始节点 S_0 开始,向下逐层搜索,在 n 层节点未搜索完之前,不进入 $n+1$ 层搜索。同层节点的搜索次序可以任意排列。即先按生成规则生成第一层节点,在

该层全部节点中沿宽(广)度进行横向扫描,检查目标节点 S_g 是否在这些子节点中。若没有,则再将所有第一层节点逐一扩展,得到第二层节点。并逐一检查第二层节点中是否包含有 S_g,如此依次按照先生成、先检查、先扩展的原则进行下去,直到发现 S_g 为止。

2. 宽度优先搜索的流程

宽度优先搜索的搜索过程如下所示:

(1) 把初始节点 S_0,放入 OPEN 表。

(2) 如果 OPEN 表为空。则问题无解,失败并退出。

(3) 把 OPEN 表中的第一个节点取出放入 CLOSE 表中,并按顺序冠以编号 n。

(4) 考察节点 n 是否为目标节点。若是,则求得了问题的解,成功并退出。

(5) 若节点 n 不可扩展,则转第(2)步。

(6) 扩展节点 n,将其子节点放到 OPEN 表的尾部,并为每一个子节点都配置指向父节点的指针,然后转第(2)步。

在宽度优先搜索下,如果问题有解,OPEN 表中必然出现目标节点 S_g,算法一定在第(4)步停止。这表示解已找到,从该目标节点 S_g 根据返回指针往回追溯,直到初始节点 S_0,所得到的一条路径就是问题的解。

例 3.2 如图 3.8 所示,通过挪动积木块,希望从初始状态达到一个目的状态,即三块积木堆叠在一起。积木 A 在顶部,积木 B 在顶中间,积木 C 在底部。请画出按照宽度优先搜索策略所产生的搜索树。

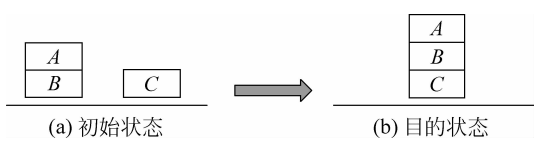

图 3.8 积木问题

这个问题的唯一操作算子为 MOVE(X,Y),即积木 X 搬到 Y(积木或桌面)上面。如挪动积木 A 到桌面上表示为 MOVE(A,Table)。该操作算子可运用的先决条件是:

(1) 被挪动积木的顶部必须为空。

(2) 如 Y 是积木(不是桌面),则积木 Y 的顶部也必须为空。

(3) 同一状态下,运用操作算子的次数不得多于一次。

因此经分析,该宽度优先搜索树如图 3.9 所示。

3. 宽度优先搜索的时间复杂度

为了便于分析,考虑一棵树,其每个节点的分支系数为 b,最大深度为 d。其中分支系数是指一个节点可以扩展产生的新的节点数目。因此搜索树的根节点在第一层会产生 b 个节点,每个节点又产生 b 个新节点,这样在第二层会有 b^2 个节点。因此目标不会出现在深度为($d-1$)层,失败搜索的最小节点数目为:

$$1+b+b^2+\cdots+b^{d-1}=(b^{d-1})/(b-1) \quad b>>1$$

而在找到目标节点之前可能扩展的最大节点数目为:

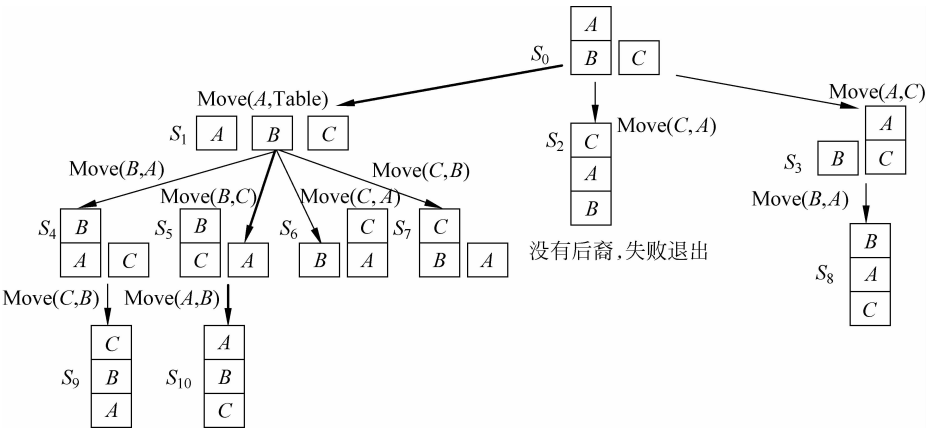

图 3.9 积木问题的宽度优先搜索树

$$1+b+b^2+\cdots+b^{d-1}+b^d=(b^{d+1}-1)/(b-1)$$

对于 d 层,目标节点可能是第一个状态,也可能是最后一个状态。因此,平均需要访问的 d 层节点数目为:

$$(1+b^d)/2$$

所以平均总的搜索的节点数目为:

$$(b^d-1)/(b-1)+(1+b^d)/2\approx b^d(b+1)/2(b-1)$$

因此,宽度优先搜索的时间复杂度和搜索的节点数目成正比。

4. 宽度优先搜索的空间复杂度

宽度优先搜索中,空间复杂度和时间复杂度一样,需要很大的空间,这是因为树的所有叶节点都同时需要储存起来。根节点扩展后,队列中有 b 个节点。第 1 层的最左边节点扩展后,队列中有$(2b-1)$个节点。而当 d 层最左边的节点正在检查是否是目标节点的时候,在队列中的节点数目最多,为(b^d)。该算法的空间复杂度和列对长度有关,在最坏的情况下约为指数级 $O(b^d)$。

表 3.1 给出了宽度优先搜索的时间和空间需求情况,其中分支系数 $b=10$,每秒钟处理 1000 个节点,每个节点需要 100 个字节。

表 3.1 宽度优先搜索的时间和空间需求

深 度	节点数	时 间	空 间
0	1	1μs	100B
2	111	1s	11KB
4	11111	11s	1MB
6	10^6	18min	111MB
8	10^8	31h	11GB
10	10^{10}	128d	1TB
12	10^{12}	35a	111TB
14	10^{14}	3500a	11 111TB

5. 宽度优先搜索的优缺点

宽度优先搜索是一种盲目搜索，时间和空间复杂度都比较高，当目标节点距离初始节点较远时会产生许多无用的节点，搜索效率低。从表 3.1 可以看出，宽度优先搜索中，时间需求是一个很大的问题，特别是当搜索的深度比较大时，尤为严重，但是空间需求是比执行时间更严重的问题。

但是宽度优先搜索也有其优点：由于宽度优先搜索总是在生成扩展完 N 层的节点之后才转向 $N+1$ 层，所以目标节点如果存在，用宽度优先搜索算法总可以找到该目标节点，而且是最小(即最短路径)的节点。但实际意义不大，当状态的后裔数的平均值较大，这种组合爆炸就会使算法耗尽资源，在可利用的空间中找不到解。

3.3.2 深度优先搜索

1. 深度优先搜索的基本思想

深度优先搜索(Depth-First Search)是一种一直向下的搜索策略。它是从初始节点 S_0 开始，按生成规则生成下一级各子节点，检查是否出现目标节点 S_g；若未出现，则按“最晚生成的子节点优先扩展”的原则，再用生成规则生成再下一级的子节点，再检查是否出现 S_g；若仍未出现，则再扩展最晚生成的子节点。如此下去，沿着最晚生成的子节点分枝，逐级“纵向”深入搜索。

由于一个有解的问题常常含有无穷分枝，深度优先搜索过程如果误入无穷分枝，就不可能找到目标节点，所以它是不完备的。与宽度优先搜索不同，深度优先搜索找到的解也不一定是最佳的。

2. 深度优先搜索的流程

深度优先搜索的搜索过程如下所示：

(1) 把初始节点 S_0 放入 OPEN 表。

(2) 如果 OPEN 表为空，则问题无解，失败并退出。

(3) 把 OPEN 表中的第一个节点取出放入 CLOSE 表中，并按顺序冠以编号 n。

(4) 考察节点 n 是否为目标节点。若是，则求得了问题的解，成功并退出。

(5) 若节点 n 不可扩展，则转第(2)步。

(6) 扩展节点 n，将其余节点放到 OPEN 表的首部，并为其配置指向父节点的指针，然后转第(2)步。

如果问题有解，且问题树没有无穷分枝，则必在 OPEN 表中出现 S_g，过程必将在第(4)步停止，搜索成功。所以深度优先搜索法仅对有限状态空间类问题来说具有算法性，但无可采纳性。一般说来它仅是一个过程。需要进一步改进。

例 3.3 卒子穿阵问题。要求一卒子从顶部通过如图 3.10 所示的列阵到达底部。要求卒子行进中不可进入到代表敌兵驻守的区域内(标注 1)，并不准后退。

行	1	2	3	4	列
1	1	0	0	0	
2	0	0	1	0	
3	0	1	0	0	
4	1	0	0	0	

图 3.10 卒子穿阵问题

假定限制值为5,请画出按照深度优先搜索策略所产生的搜索树。

经分析,该深度优先搜索树如图3.11所示。

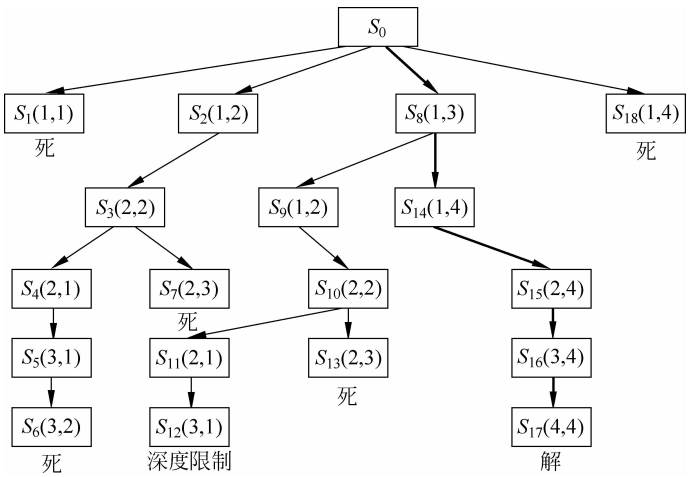

图3.11 卒子穿阵的深度优先搜索树

3. 深度优先搜索的时间复杂度

如果搜索在d层最左边的位置找到了目标,则检查的节点数为$(d+1)$。另一方面,如果只是搜索到d层,而在d层的最右边找到了目标,则检查的节点包括了树中所有节点,其数量为:

$$1+b+b^2+\cdots+b^d=(b^{d+1}-1)/(b-1)$$

所以,平均来说,检查的节点数量为:

$$(b^{d+1}-1)/2(b-1)+(1+d)/2\approx b(b^d+d)/2(b-1)$$

上式就是深度优先搜索的平均时间复杂度。

4. 深度优先搜索的空间复杂度

深度优先搜索对内存的需求是比较适中的。它只需要保存从根到叶的单条路径,包括在这条路径上每个节点的未扩展的兄弟节点的存储器要求是深度约束的线性函数。当搜索过程到达了最大深度的时候,所需要的内存最大。假设每个节点的分支系数为b,当考虑深度为d的一个节点时,保存在内存中的节点的数量包括到达深度d时所有未扩展的节点以及正在被考虑的节点。因此,在每个层次上都有$(b-1)$个未扩展的节点,总的内存需要量为$d(b-1)+1$。因此深度优先搜索的空间复杂度是b的线性函数$O(bd)$,而宽度优先搜索的空间复杂度是b的指数函数$O(b^d)$。

5. 深度优先搜索的优缺点

深度优先搜索的优点是比宽度优先搜索算法需要较少的空间。该算法只需要保存搜索树的一部分,它由当前正在搜索的路径和该路径上还没有完全展开的节点标志所组成。因此,深度优先搜索的存储器要求是深度约束的线性函数。

但是其主要问题是可能搜索到了错误的路径上。很多问题可能具有很深甚至是无限的搜索树，如果不幸选择了一个错误的路径，则深度优先搜索会一直搜索下去，而不会回到正确的路径上。这样一来对于这些问题，深度优先搜索要么陷入无限的循环而不能给出一个答案，要么最后找到一个答案，但路径很长而且不是最优的答案。这就是说，深度优先搜索既不是完备的，也不是最优的。

3.3.3 有界深度搜索和迭代加深搜索

对于深度 d 比较大的情况，深度优先搜索需要很长的运行时间，而且还可能得不到解答。一种比较好的问题求解方法是对搜索树的深度进行控制，即有界深度优先搜索方法。有界深度优先搜索过程总体上按深度优先算法方法进行，但对搜索深度需要给出一个深度限制 d_m，当深度达到了 d_m 的时候，如果还没有找到解答，就停止对该分支的搜索，换到另外一个分支进行搜索。

有限深度优先搜索的搜索过程如下所示：

(1) 把初始节点 S_0 放入 OPEN 表中，置 S_0 的深度 $d(S_0)=0$。

(2) 如果 OPEN 表为空，则问题无解，失败并退出。

(3) 把 OPEN 表中的第一个节点取出放入 CLOSE 表中。并按顺序冠以编号 n。

(4) 考察节点 n 是否为目标节点。若是，则求得了问题的解，成功并退出。

(5) 如果节点 n 的深度 $d(n)=d_m$，则转第(2)步。

(6) 如果节点 n 不可扩展，则转第(2)步。

(7) 扩展节点 n。将其子节点放入 OPEN 表的首部，并为其配置指向父节点的指针。然后转第(2)步。

对于有界深度搜索策略，下面有几点需要说明：

(1) 深度限制 d_m 很重要。当问题有解，且解的路径长度小于或等于 d_m 时，则搜索过程一定能够找到解，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：即这时有界深度搜索是完备的但不是最优的。但是当 d_m 取得太小，解的路径长度大于 d_m 时，则搜索过程中就找不到解：即这时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节点，既浪费了计算机资源，又降低了搜索效率。有界深度搜索的时间和空间复杂度与深度优先搜索类似，空间是线性复杂度 $O(bd_m)$，时间是指数复杂度为 $O(b^{d_m})$。

(3) 有界深度搜索的主要问题是深度限制值 d_m 的选取。该值也被称为状态空间的直径，如果该值设置的比较合适，则会得到比较有效的有界深度搜索。但是对很多问题，我们并不知道该值到底为多少，直到该问题求解完成了，才可以确定出深度限制值 d_m。为了解决上述问题，可采用如下的改进方法：先任意给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。这就是迭代加深搜索的基本思想。

迭代加深搜索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：首先深度为 0，然后深度为 1，然后为 2，等等，一直进行下去。如果初始深度为 0，则该算法只生成根节点，并检测它。如果根节点不是目标，则深度加 1，通过典型的深度优先算法，生成深度为 1 的树。同样当深度限制为 m 时，树的深度也为 m。

迭代加深搜索算法描述如下：

Procedure Iterative-deepening

Begin

(1) 设置当前深度限制＝1；

(2) 把初始节点压入栈，并设置栈顶指针；

(3) While 栈不空并且深度在给定的深度限制之内 do

Begin

弹出栈顶元素；

If 栈顶元素＝goal，返回并结束；

Else 以任意的顺序把栈顶元素的子女压入栈中；

End

End while

(4) 深度限制加 1，并返回(2)；

End.

3.3.4 搜索最优策略的比较

宽度优先搜索、深度优先搜索和迭代加深搜索都可以用于生成和测试算法。然而宽度优先搜索需要指数数量的空间，深度优先搜索的空间复杂度和最大搜索深度呈线性关系。迭代加深搜索对一棵深度受控的树采用深度优先的搜索。它结合了宽度优先和深度优先搜索的优点。和宽度优先搜索一样，它是最优的，也是完备的。但对空间要求和深度优先搜索一样是适中的。表 3.2 给出了这四种搜索策略的比较。其中 b 是分支系数，d 是解答的深度，m 是搜索树的最大深度，l 是深度限制。

表 3.2 四种搜索策略的比较

标准	宽度优先	深度优先	有界深度	迭代加深
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

3.4 启发式搜索

前面讨论的各种搜索方法都是按事先规定的路线进行搜索，没有用到问题本身的特征信息，具有较大的盲目性，产生的无用节点较多，搜索空间较大，效率不高。如果能够利用问题自身的一些特征信息来指导搜索过程，则可以缩小搜索范围，提高搜索效率。

启发式搜索通常用于两种不同类型的问题：正向推理和反向推理。正向推理一般用于状态空间的搜索。在正向推理中，推理是从预选定义的初始状态出发向目标状态方向执行。

反向推理一般用于问题规约中。在反向推理中，推理是从给定的目标状态向初始状态执行。在前一类使用启发式函数的搜索算法中，包括通常所谓的 OR 图算法或者最好优先算法，以及根据启发式函数的不同而得到的其他的一些算法，如 A^* 算法等。另一方面，启发式反向推理算法通常称为 AND-OR 图搜索算法，AO^* 算法就是其中一种算法。

3.4.1 启发性信息和评估函数

如果在选择节点时能充分利用与问题有关的特征信息，估计出节点的重要性，就能在搜索时选择重要性较高的节点，以利于求得最优解。把这个过程称为启发式搜索。“启发式”实际上代表了“大拇指准则(Thumb Rules)”：在大多数情况下是成功的，但不能保证一定成功的准则。

与被解问题的某些特征有关的控制信息(如解的出现规律、解的结构特征等)称为搜索的启发信息。它反映在评估函数中。评估函数的作用是估计待扩展各节点在问题求解中的价值，即评估节点的重要性。

评估函数 $f(x)$定义为从初始节点 S_0出发，约束地经过节点 x 到达目标节点 S_g 的所有路径中最小路径代价的估计值。其一般形式为：

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

其中，$g(x)$表示从初始节点 S_0到节点 x 的实际代价；$h(x)$表示从 x 到目标节点 S_g的最优路径的评估代价，它体现了问题的启发式信息，其形式要根据问题的特性确定，$h(x)$称为启发式函数。启发式方法把问题状态的描述转换成了对问题解决程度的描述，这一程度用评估函数的值来表示。

3.4.2 启发式搜索算法 A

在一般图的算法中(见 3.2.3 节)中，如果第(8)步的重排 OPEN 表是依据 $f(x)=g(x)+h(x)$进行的，则称该过程为 A 算法。

启发式算法 A 按 $f(x)$排序 OPEN 表中的节点，$f(n)$值最小者排在首位，优先加以扩展，体现了最佳优先(best-first)搜索策略的思想。下面以八数码游戏为例，观察算法 A 的应用。

对于八数码问题，评估函数可表示为：

$$f(x) = d(x) + w(x)$$

其中 $d(x)$是当前被考察和扩展的节点 n 在搜索图中的节点深度，作为对 $g(x)$的量度；而启发式函数 $h(x)$则设计为 $w(x)$，其值是节点 x 与目标状态节点 S_g相比较，错位的棋牌个数。一般来说某节点的 $w(x)$越大，即“不在目标位”的数码个数越多，说明它离目标节点越远。

设想当前要解决的八数码问题如图 3.12 所示，初始状态节点的评价函数值 $f(x)=0+4=4$，则应用算法 A 搜索解答路径十分快捷，除了个别走步判断失误外(节点 d 的选用和扩展)，其他的走步选择全部正确。

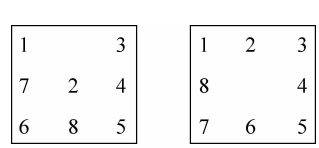

图 3.12 要解决的八数码问题

图 3.13 给出了搜索图，并以字母标识每个节点，字母后的括号中给出评价函数 f 的值，且每次循环从 OPEN 选取扩

展的节点用方框框出,最终的 g 即目标节点;OPEN 和 CLOSE 表中的节点变化如表 3.3 所示。

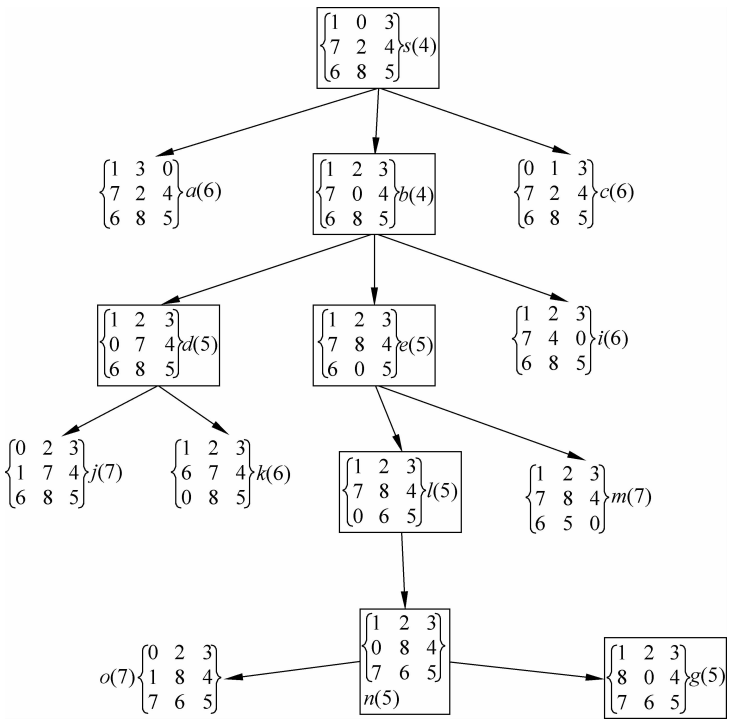

图 3.13 应用算法 A 的八数码搜索图

表 3.3 每个搜索循环结束时 OPEN 表和 CLOSE 表中的节点

循环	OPEN 表	CLOSE 表
初始化	(S)	( )
1	(b c a)	(S)
2	(d e a c i)	(S b)
3	(e a c i k j)	(S b d)
4	(l a c i k j m)	(S b d e)
5	(n a c i k j m)	(S b d e l)
6	(g a c i k j m o)	(S b d e l n)
7	成功结束	

3.4.3 实现启发式搜索的关键因素和 A* 算法

鉴于启发式搜索在提高搜索效率和解决组合爆炸问题中的作用,相关的研究成为人工智能形成和成长期的重要议题之一,也产生了许多成熟的研究成果,并且至今启发式搜索仍是一个活跃的研究领域。下面就实现启发式搜索应考虑的关键因素做一些讨论。

1. 搜索算法的可采纳性(Admissibility)和 A* 算法

在搜索图存在从初始状态节点到目标状态节点解答路径的情况下,若一个搜索法总能找到最短(代价最小)的解答路径,则称算法具有可采纳性。例如,宽度优先的搜索算法就是可采纳的,只是其搜索效率不高。

为考察启发式搜索算法 A 的可纳性,首先引入评价函数 f^*:

$$f^*(x) = g^*(x) + h^*(x)$$

$f^*(x)$、$g^*(x)$、$h^*(x)$分别指示当经由节点 x 的最短(代价最小)解答路径找到时实际的路径代价(长度)、该路径前段(自初始状态节点到节点 x)代价和后段(自节点 x 到目标状态节点)代价。在存在多个目标状态的情况下,$h^*(x)$取 $h^*(x, x_{gi})$中最小者($i=1,2,\cdots$)。

将评价函数 f 与 f^* 相比较,实际上,$f(x)$、$g(x)$和 $h(x)$分别是 $f^*(x)$、$g^*(x)$和 $h^*(x)$的近似值。在理想的情况下,设计评价函数 f 时可以让 $g(x)=g^*(x)$,$h(x)=h^*(x)$,则应用该评价函数的算法 A 就能在搜索过程中,每次都正确地选择下一个从 OPEN 表中取出加以扩展的节点,从而不会扩展任何无关的节点,就可顺利地获取解答路径。然而 $g^*(x)$和 $h^*(x)$在最短解答路径找到前是未知的,故而几乎不可能设计出这种理想的评价函数;而且对于复杂的应用领域,即便是要设计接近于 f^* 的 f 往往也是困难的。一般来讲,$g(x)$的值容易从迄今已生成的搜索树中计算出来,不必专门定义计算公式。例如就以节点深度 $d(x)$作为 $g(x)$,并有 $g(x) \geqslant g^*(x)$。然而 $h(x)$的设计依赖于启发式知识的应用,所以如何挖掘贴切的启发式知识是设计评价函数乃至算法 A 的关键。

前述八数码游戏例(参见图 3.13)使用的启发式函数 $w(x)$就不够贴切,从而在搜索过程中错误地选用了节点 d 加以扩展。其实可以设计更接近于 $h^*(x)$的 $h(x)$,如 $p(x)$,其值是节点 x 与目标状态节点相比较,每个错位棋牌在假设不受阻拦的情况下,移动到目标状态相应位置所需走步(移动次数)的总和。显然 $p(x)$比 $w(x)$更接近于 $h^*(x)$,因为 $p(x)$不仅考虑了错位因素,还考虑了错位的距离(移动次数)。

返回到图 3.13,若启发式函数 $h(x)$采用 $p(x)$,则初始状态节点的评价函数值 $f(s)=5$。在第二个搜索循环结束时,OPEN 表中的节点排序为:$e\ a\ c\ d\ i$,这些节点的评价函数值依次为:5 7 7 7 7;而在使用 $w(x)$情况下,OPEN 表中的节点排序为 $d\ e\ a\ c\ i$,评价函数值依次为:5 5 6 6 6。显然,$p(x)$使得用 $w(x)$不能区分的节点 $d(5)$和 $e(5)$区分为 $d(7)$和 $e(5)$,从而搜索过程不再会错误选择节点 d,而是一下子就选对了节点 e 加以扩展,如图 3.14 所示。

可以证明,若确保对于搜索图中的节点 x,总是有 $h(x) \leqslant h^*(x)$,则算法 A 具有可采纳性,即总能搜索到最短(代价最小)的解答路径。我们称满足 $h(x) \leqslant h^*(x)$的算法 A 为 A*。

A* 算法的步骤为:

(1) 把 S 放入 OPEN 表,记 $f=h$,令 CLOSE 为空表。

(2) 若 OPEN 为空表,则宣告失败并退出。

(3) 选取 OPEN 表中未设置过的具有最小 f 值的节点为最佳节点 BESTNODE,并把它放入 CLOSE 表。

(4) 若 BESTNODE 为一目标节点,则成功求得一解并退出。

(5) 若 BESTNODE 不是目标节点,则扩展之,产生后继节点 SUCCSSOR。

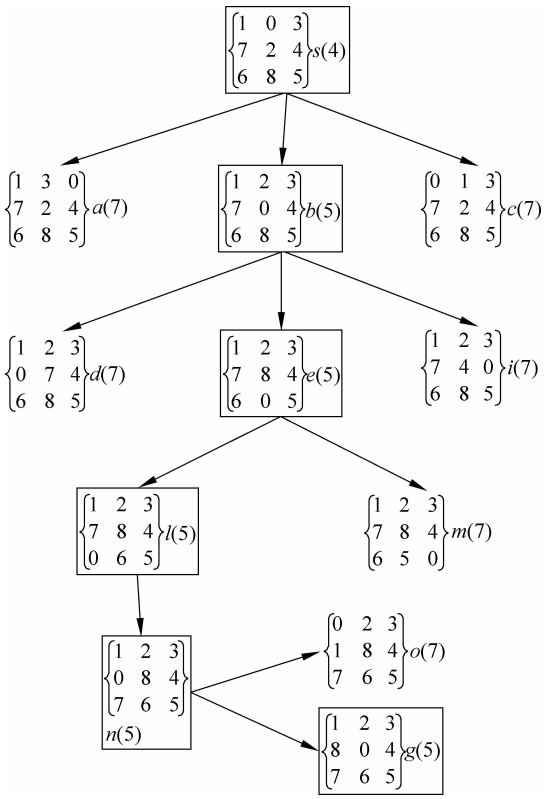

图 3.14 应用启发式函数 $p(x)$的八数码问题搜索图

(6) 对每个 SUCCSSOR 进行下列过程:

① 建立从 SUCCSSOR 返回 BESTNODE 的指针;

② 计算 $g(\text{SUC})=g(\text{BES})+g(\text{BES},\text{SUC})$;

③ 如果 SUCCSSOR∈OPEN,则此节点为 OLD,并把它添至 BESTNODE 的后继节点表中;

④ 比较新旧路径代价。如果 $g(\text{SUC})<g(\text{OLD})$,则重新确定 OLD 的父辈节点为 BESTNODE,记下较小代价 $g(\text{OLD})$,并修正 $f(\text{OLD})$值;

⑤ 若至 OLD 节点的代价较低或一样,则停止扩展节点;

⑥ 若 SUCCSSOR 不在 OPEN 表中,则看其是否在 CLOSE 表中;

⑦ 若 SUCCSSOR 在 CLOSE 表中,则转向③步骤;

⑧ 若 SUCCSSOR 既不在 OPEN 表中,又不在 CLOSE 表中,则把它放入 OPEN 表中,并添入 BESTNODE 后裔表,然后转向第(7)步;

(7) 计算 f 值,然后转入第(2)步。

A* 算法一定是可采纳的。其证明方式如下:

(1) 如果存在一条从初始状态到目标状态的解答路径,则一定存在一条最短解答通路;

(2) 设状态 x'是最短解答路径上的一个状态,那么经过有限步后,x'必然会成为 OPEN 表的第一个节点;

(3) 因为最短解答路径只有有限个节点 x'，所以有限步后算法必然因到达目标状态 x_g。这就是最优解；

(4) 证明完毕。

对于八数码游戏，从当前被扩展节点 x 到目标状态节点的最短路径——棋牌移动的最少次数必定不少于错位棋牌的个数(因为有些棋牌可能需移动多于一次才能到达目标状态的相应位置)，也必定不会少于错位棋牌不受阻挡情况下移动到目标状态相应位置的移动次数总和(因为有些棋牌的移动可能受阻挡)，从而采用 $w(x)$ 和 $p(x)$ 作为评价函数时，算法 A 都是可纳的。

2. 启发式函数的强弱及其影响

可以用 $h(x)$ 接近 $h^*(x)$ 的程度去衡量启发式函数的强弱。当 $h(x) < h^*(x)$ 且两者差距较大时，$h(x)$ 过弱，从而导致 OPEN 表中节点排序的误差较大，易于产生较大的搜索图；反之，当 $h(x) > h^*(x)$，则 $h(x)$ 过强，使算法 A 失去可采纳性，从而不能确保找到最短解答路径。显然，设计恒等于 $h^*(x)$ 的 $h(x)$ 是最为理想的，其确保产生最小的搜索图(因为 OPEN 表中节点的排序正确)，且搜索到的解答路径是最短的。

正如前述，对于复杂的问题求解任务，设计恒等于 $h^*(x)$ 的 $h(x)$ 是不可能的。为此，取消恒等约束，设计接近，又总是$\leqslant h^*(x)$ 的 $h(x)$ 成为应用 A^* 算法搜索问题解答的关键，以压缩搜索图，提高搜索效率。可以证明，对于解决同一问题的两个算法 A1 和 A2，若总有 $h1(x) \leqslant h2(x) \leqslant h^*(x)$，则 $t(A1) \geqslant t(A2)$。其中，$h1$、$h2$ 分别是算法 A1、A2 的启发式函数，t 指示相应算法达到目标状态时搜索图含的节点总数。再以八数码游戏为例，正因为 $w(x) \leqslant p(x) \leqslant h^*(x)$，所以采用 $p(x)$ 扩展出的节点总数不会比采用 $w(x)$ 时多。更明显的例子是采用宽度优先法解决八数码问题，其相当于 $h(x) \equiv 0$，图 3.14 中的搜索树会变得比采用 $w(x)$ 时庞大得多。

3. 设计 $h(x)$ 的实用考虑

随着问题求解任务复杂程度的增加，即便是设计接近，又总是$\leqslant h^*(x)$ 的 $h(x)$ 也变得更困难，而且往往会导致在 $h(x)$ 上的繁重计算工作量。若 $h(x)$ 的计算开销过大，即使最短路径找到，实际的搜索代价也会高居不下，因为路径选择代价随 $h(x)$ 的计算开销而大增。删除 $h(x) \leqslant h^*(x)$ 的约束，将会使 $h(x)$ 的设计容易得多，但却由此也丢失了可采纳性(可能丢失最短解答路径)。不过在许多实用场合，人们并不要求找到最优解答(最短解答路径)，通过牺牲可采纳性来换取 $h(x)$ 设计的简化和减少计算 $h(x)$ 的工作量还是可行的。

从评价函数 $f(x)=g(x)+h(x)$ 可以看出，若 $h(x) \equiv 0$，则意味着先进入 OPEN 表的节点会优先被考察和扩展，因为即使不以 $d(x)$ 作为 $g(x)$，通常先进入 OPEN 表的节点 x 也具有较小的 $g(x)$ 值，从而使搜索过程接近于宽度优先的搜索策略；反之若 $g(x) \equiv 0$，则导致后进入 OPEN 表的节点会优先被考察和扩展，因为后进入 OPEN 表的节点 x 往往更接近于目标状态，即 $h(x)$ 值较小，从而使搜索过程接近于深度优先的搜索策略。

为更有效地搜索解答，可使用评价函数 $f(x)=g(x)+wh(x)$，w 用作加权。在搜索图的浅层(上部)，可让 w 取较大值，以使 $g(x)$ 所占比例很小，从而突出启发式函数的作用，加速向纵深方向搜索；一旦搜索到较深的层次，又让 w 取较小值，以使 $g(x)$ 所占比例很大，并

确保 $wh(x) \leqslant h^*(x)$,从而引导搜索向横广方向发展,寻找到较短的解答路径。

3.4.4 迭代加深 A* 算法

前面讨论了迭代加深搜索算法,它以深度优先的方式在有限制的深度内搜索目标节点。该算法在每个深度上检查目标节点是否出现,如果出现则停止,否则深度加 1 继续搜索。而 A* 算法是选择具有最小估价函数值的节点扩展。由于 A* 算法把所有生成的节点保存在内存中,所以 A* 算法在耗尽计算时间之前一般早已经把空间耗尽了。目前开发了一些新的算法,它们的目的是为了克服空间问题。但一般不满足最优性或完备性,如迭代加深 A* 算法 IDA*、简化内存受限 A* 算法 SMA* 等。

迭代加深 A* 搜索算法 IDA* 是上述两种算法的结合。这里启发式函数用做深度的限制,而不是选择扩展节点的排序。下面简单介绍 IDA* 算法。

Procedure IDA* 算法

```
Begin
  (1) 初始化当前的深度限制 c = 1
  (2) 把初始节点压入栈;
  (3) While 栈不空 do
        Begin
          弹出栈顶元素 n
            If n = goal, Then 结束, 返回 n 以及从初始节点到 n 的路径
            Else do
            Begin
                 For n 的每个子节点 n'
                    If f(n')≤c , Then 把 n'压入栈
                    Else c' = min(c', f(n'))
                 End for
              End
            End While
  (4) If 栈为空并且 c' = ∞, Then 停止并退出
  (5) If 栈为空并且 c'≠∞, Then c = c', 并返回 2
End
```

上述算法涉及了两个深度限制。如果栈中所含节点的所有子节点的 f 值小于限制值 c,则把这些子节点压入栈中以满足迭代加深算法的深度优先准则。然而,如果不这样,即节点 n 的一个或多个子节点 n' 的 f 值大于限制值 c,则节点 n 的 c' 设置为 $\min(c', f(n'))$。该算法终止的条件为:

(1) 找到目标节点(成功结束);

(2) 栈为空并且限制值 $c'=\infty$。

IDA* 算法和 A* 算法相比,主要优点是对于内存的需求。A* 算法需要指数级数量的存储空间,因为没有深度方面的限制。而 ID A* 算法只有当节点 n 的所有子节点 n' 的 $f(n')$ 小于限制值 c 时才扩展它,这样就可以节省大量的内存。

另一问题是当启发式函数是最优的时候,ID A* 算法和 A* 算法扩展相同的节点,并且可以找到最优路径。

3.4.5 回溯策略和爬山法

在 $g(x)\equiv 0$ 的情况下，若限制只用评价函数 $f(x)=h(x)$ 去排序新扩展出来的子节点，即局部排序，就可实现较为简单的搜索策略：回溯策略和爬山法。由于简单易行，在不要求最优解答的问题求解任务中，回溯策略得到广泛的应用。爬山法则适用于能逐步求精的问题。

1. 爬山法

爬山法是实现启发式搜索的最简单方法。人们在登山时，总是设法快速登上顶峰，所以只要好爬，总是选取最陡处，以求快速登顶。爬山实际上就是求函数极大值问题，不过这里不是用数值解法，而是依赖于启发式知识，试探性地逐步向顶峰逼近（广义地，逐步求精），直到登上顶峰。

在爬山法中，限制只能向山顶爬去，即向目标状态逼近，不准后退，从而简化了搜索算法；即不需设置 OPEN 和 CLOSE 表，因为没有必要保存任何待扩展节点；仅从当前状态节点扩展出子节点（相当于找到上爬的路径），并将 $h(x)$ 最小的子节点（对应于到顶峰最近的上爬路径）作为下一次考察和扩展的节点，其余子节点全部丢弃。

爬山法对于单一极值问题（登单一山峰）十分有效而又简便，但对于具有多极值的问题就无能为力了，因为很可能会因错登高峰而失败——不能到达最高峰。

2. 回溯策略

回溯策略可以有效地克服爬山法面临的困难，其保存了每次扩展出的子节点，并按 $h(x)$ 值从小到大排列。如此，相当于爬山的过程中记住了途经的岔路口，只要当前路径搜索失败就回溯（退回）到时序上最近的岔路口，向另一路径方向搜索，从而可以确保最后到达最高峰（即目标状态）。

实现回溯策略的有效方式是应用的递归过程去支持搜索和回溯。令 PATH、SXL、x、x' 为局部变量：

- PATH——节点列表，指示解答路径；
- SXL——当前节点扩展出的子节点列表；
- MOVE-FIRST(SXL)——把 SXL 表首的节点移出，作为下一次要加以扩展的节点；
- x、x'——分别指示当前考察和下一次考察的节点。

该递归过程的算法就取名为 BACKTRACK(x)，参数 x 为当前被扩展的节点，算法的初次调用式是 BACKTRACK(s)，s 即为初始状态节点。算法的步骤如下：

(1) 若 x 是目标状态节点，则算法的本次调用成功结束，返回空表；

(2) 若 x 是失败状态，则算法的本次调用失败结束，返回'FAIL'；

(3) 扩展节点 x，将生成的子节点置于列表 SXL，并按评价函数 $f(k)=h(k)$ 的值从小到大排序（k 指示子节点）；

(4) 若 SXL 为空，则算法的本次调用失败结束，返回'FAIL'；

(5) x'=MOVE-FIRST(SXL)；

(6) PATH=BACKTRACK(x')；

(7) 若 PATH='FAIL', 返回到第(4)步;

(8) 将 x'加到 PATH 表首,算法的本次调用成功结束,返回 PATH。

该递归回溯算法中,失败状态通常意指三种情况:

(1) 不合法状态(如传教士和野人问题中所述的那样)。

(2) 旧状态重现(如八数码游戏中某一棋盘布局的重现,会导致搜索算法死循环)。

(3) 状态节点深度超过预定限度(例如,在八数码游戏中,指示解答路径不超过 6 步)。

失败状态实际上定义了搜索过程回溯的条件;另一种回溯条件是搜索进入"死胡同",由该算法的第(4)步定义。由于回溯是递归算法,解答路径的生成是从算法到达目标状态后逆向进行的,首先产生空表,然后每回到算法的上一次调用就在表首加入节点 x',直到顶层调用返回不包含初始状态节点 s 的解答路径。

影响回溯算法效率的关键因素是回溯次数。鉴于回溯是搜索到失败状态时的一种弥补行为,只要能准确地选择下一步搜索考察的节点,就能大幅度减少甚至避免回溯。所以,设计好的启发式函数 $h(x)$是至关重要的。

3.5 问题规约和与/或图启发式搜索

3.5.1 问题规约

问题规约是人求解问题常用的策略,其把复杂的问题变换为若干需要同时处理的较为简单的子问题后再加以分别求解。只有当这些子问题全部解决时,问题才算解决,问题的解答就由子问题的解答联合构成。问题规约可以递归地进行,直到把问题变换为本原问题的集合。所谓本原问题就是不可或不需再通过变换化简的"原子"问题,本原问题的解可以直接得到或通过一个"黑箱"操作得到。

问题规约是一种广义的状态空间搜索技术,其状态空间可表示为三元组:

$$SP = (S_0, O, P)$$

其中,S_0是初始问题,即要求解的问题;P 是本原问题集,其中的每一个问题是不用证明的,自然成立的,如公理、已知事实等,或已证明过的问题;O 是操作算子集,它是一组变换规则,通过一个操作算子把一个问题化成若干个子问题。

变换可区分为以下三种情况:

(1) 状态变迁——导致问题从上一状态变迁到下一状态,这就是一般图搜索技术中操作算子的作用。

(2) 问题分解——分解问题为需同时处理的子问题,但不改变问题状态。

(3) 基于状态变迁的问题分解——先导致状态变迁,再实现问题分解,实际上就是前两个操作的联合执行。

作为问题规约的例子,观察下面的符号积分求解问题:初始状态为$\int f(x)\mathrm{d}x$,目标状态为可直接求原函数和积分的本原问题,如$\int \sin(x)\mathrm{d}x$、$\int \cos(x)\mathrm{d}x$ 等。而操作算子就是积分变换规则。

下面就是一次典型的积分变换：

$$\begin{aligned}&\int(\sin 3x+x^4/(x^2+1))\mathrm{d}x\\&=\int\sin 3x\mathrm{d}x+\int(x^4/(x^2+1))\mathrm{d}x\\&=\int-(1-\cos^2 x)\mathrm{d}\cos x+\int(x^2-1+1/(1+x^2))\mathrm{d}x\\&=\left(\int-\mathrm{d}\cos x+\cos^2 x\mathrm{d}\cos x\right)+\left(\int x^2\mathrm{d}x-\int\mathrm{d}x+\int(1/(1+x^2))\mathrm{d}x\right)\\&=-\cos x+\cos^3 x/3+x^3/3-x+\mathrm{arctg}x\end{aligned}$$

通过上面的变换可以看出，问题规约的实质是从目标（要解决的问题）出发逆向推理，建立子问题以及子问题的子问题，直至最后把初始问题规约为一个平凡的本原问题集合。

在简单问题的规约过程中，各子问题相互独立，所以子问题的进一步规约和本原问题的求解无交互作用，可按任意次序进行。然而对于许多复杂问题，子问题仅相对独立，之间仍存在一定的交互作用。在这种情况下，正确安排子问题求解的先后次序，甚至孙子问题求解的次序是重要的。

比如下面的梵塔问题，其问题描述如下：

有编号为1、2、3的三个柱子和标识为A、B、C的尺寸依次为小、中、大的三个有中心孔的圆盘。初始状态下三个盘按A、B、C顺序堆放在1号柱子上，目标状态下三个盘以同样次序顺序堆放在3号柱子上，盘子的搬移须遵守以下规则：每次只能搬一个盘子，且较大盘不能压放在较小盘之上，如图3.15所示。

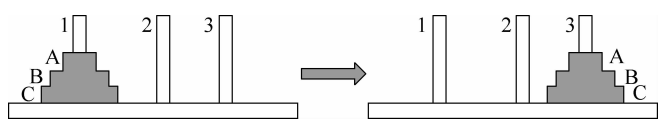

图3.15　梵塔问题

以三元素列表作为数据结构描述问题状态，三个元素依次指示盘子A、B、C所在的柱子编号。如此梵塔问题描述为(1,1,1)⇒(3,3,3)。可以把该问题规约为三个子问题(1,1,1)⇒(2,2,1)、(2,2,1)⇒(2,2,3)和(2,2,3)⇒(3,3,3)，即先把A、B盘搬到柱子2，再把C盘搬到柱子3，最后把A、B盘搬到柱子3。第1、3二个子问题再分别规约为子子问题如下：(1,1,1)⇒(3,1,1)、(3,1,1)⇒(3,2,1)、(3,2,1)⇒(2,2,1)，即依次搬A盘到柱子3，B盘到柱子2，A盘到柱子2；(2,2,3)⇒(1,2,3)，(1,2,3)⇒(1,3,3)、(1,3,3)⇒(3,3,3)，即依次搬A盘到柱子1、B盘到柱子3、A盘到柱子3。现在所有子问题均为本原问题，只要依次解决就可到达目标状态。梵塔问题的子问题间和子子问题间有交互作用，必须注意正确的排序。其状态空间图如图3.16表示，在图中已标出正确的节点生成顺序。

通过上述例子可以看出，应用问题规约策略求解问题的原理简单，方法也有效，所以得到广泛和深入的研究和应用。

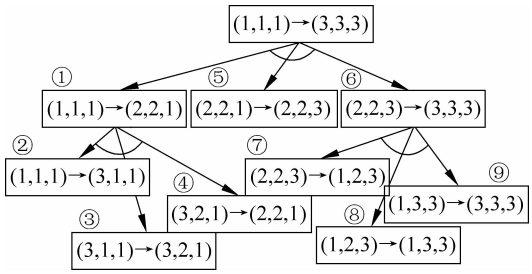

图 3.16 梵塔问题的状态空间表示

3.5.2 与/或图表示

通过问题规约可以看到,对于一个复杂的问题,常常把此问题分解成若干个子问题。如果把每个子问题都解决了,整个问题也就解决了。如果子问题不容易解决,还可以再分成子问题,直至所有的子问题都解决了,则这些子问题的解的组合就构成了整个问题的解。与/或图(AND/OR graph)就是用于表示此类求解过程的一种方法,它是一种树图的形式,是基于人们在求解问题时的一种思维方法。

(1) 分解:“与”树。把一个复杂的问题 P 分解为与之等价的一组简单的子问题 P_1,P_2,…,P_n,而子问题还可分为更小更简单的子问题,如此类推。当这些子问题全都解决时,原问题 P 也就解决了;任何一个子问题 $P_i(i=1,2,\cdots,n)$ 无解,都将导致原问题 P 无解。这样的问题与这一组子问题之间形成了“与”的逻辑关系。这一分解过程可用一个有向图来表示;问题和子问题都用相应的节点表示,从问题 P 到每个子问题 P_i 都只用一个有向边连接,然后用一段弧将这些有向边连起来,以标明它们之间存在的“与”的关系。这种有向图称为“与”图或者“与”树。

(2) 等价变换:“或”树。把一个复杂的问题 P 经过等价变换转变为与之等价的一组简单的子问题 P_1,P_2,…,P_n,而子问题还可再等价变换为若干更小更简单的子问题,如此下去。当这些子问题中有任何一个子问题 $P_i(i=1,2,\cdots,n)$ 有解时,原问题 P 也就解决了;只有当全部子问题都无解时,原问题 P 才无解。这样的问题与这一组子问题之间形成了“或”的逻辑关系。这一等价变换同样可用一个有向图来表示,这种有向图称为“或”图或者“或”树。表示方法类似与图的表示,只是在或图中不用弧将有向边连起来。

(3) 与/或图。在实际问题求解过程中,常常是既有分解又有等价变换,因而常将两种图结合起来一同用于表示问题的求解过程。此时,所形成的图就称为“与/或图”或“与/或树”。

可以把与或图视为对一般图(或图)的扩展;或反之,把一般图视为与或图的特例,即一般图不允许节点间具有“与”关系,所以又可把一般图称为或图。与一般图类似,与或图也有根节点,用于指示初始状态。由于同父子节点间可以存在“与”关系,父、子节点间不能简单地以弧线关联,需要引入超链接概念。因为同样的原因,在典型的与或图中,解答路径往往不复存在,代之以广义的解路径——解图。

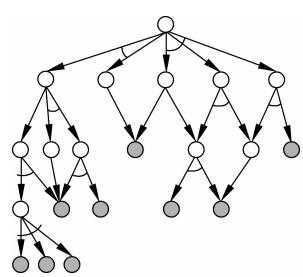

图 3.17 与或图简例

图 3.17 给出一个抽象的与或图简例,节点的状态描述不再显式给出。下面就基于该简例引入和解释与或图搜索的基

本概念。

(1) K-连接——用于表示从父节点到子节点间的连接，也称为父节点的外向连接，并以圆弧指示同父子节点间的“与”关系，K 为这些子节点的个数。一个父节点可以有多个外向的 K-连接。例如，根节点 n_0 就有 2 个 K-连接：一个 2-连接指向子节点 n_1 和 n_2，另一 3-连接指向子节点 n_3、n_4 和 n_5。K 大于 1 的连接也称为超链接，K 等于 1 时超链接蜕化为普通连接，而当所有超链接的 K 都等于 1 时，与或图蜕化为一般图。

(2) 根、叶、终节点——无父节点的节点称为根节点，用于指示问题的初始状态；无子节点的节点称为叶节点。由于问题规约伴随着问题分解，所以目标状态不再由单一节点表示，而是应由一组节点联合表示。能用于联合表示目标状态的节点称为终节点；终节点必定是叶节点，反之不然；非终节点的叶节点往往指示了解答搜索的失败。

(3) 解图的生成——在与或图搜索过程中，可以这样建立解图：自根节点开始选一外向连接，并从该连接指向的每个子节点出发，再选一外向连接，如此反复进行，直到所有外向连接都指向终节点为止。例如，从图 3.17 与或图根节点 n_0 开始，选左边的 K 等于 2 的外向连接，指向节点 n_1 和 n_2，再从 n_1、n_2 分别选外向连接；从 n_1，选左边的 K 等于 1 的外向连接，指向 n_6，依次进行，直到终节点 n_{14}、n_{18}、n_{19} 和 n_{20}；从 n_2 只有一个 K 等于 1 的外向连接指向终节点 n_9。如此，生成如图 3.18(a) 所示的一个解图。注意，解图是遵从问题规约策略而搜索到的，解图中不存在节点或节点组之间的“或”关系；换言之，解图纯粹是一种“与”图。另外，正因为与或图中存在“或”关系，所以往往会搜索到多个解图，本例中就有 4 个(见图 3.18)。

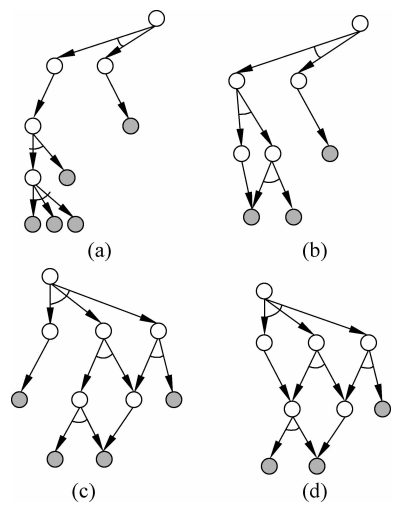

图 3.18　4 个可能的解图

为确保在与或图中搜索解图的有效性，要求解图是无环的，即任何节点的外向连接均不得指向自己或自己的先辈，否则会使搜索陷入死循环。换言之，会导致解图有环的外向连接不能选用。下面给出关于解图、解图代价、能解节点和不能解节点的定义。

(1) 解图。与或图(记为 G)任一节点(记为 n)到终节点集合的解图(记为 G')是 G 的子图。

① 若 n 是终节点，则 G' 就由单一节点 n 构成；

② 若 n 有一外向 K-连接指向子节点 $n_1, n_2, \cdots, n_k$，且这些子节点每个都有到终节点集合的解图，则 G 由该 K-连接和所有这些相应于子节点的解图构成；

③ 否则不存在 n 到终节点集合的解图。

(2) 解图代价。以 $C(n)$ 指示节点 n 到终节点集合解图的代价，并令 K-连接的代价就为 K，则有：

① 若 n 是终节点，则 $C(n)=0$；

② 若 n 有一外向 K-连接指向子节点 $n_1, n_2, \cdots, n_k$，且这些子节点每个都有到终节点集合的解图，则

$$C(n) = K + C(n_1) + C(n_2) + \cdots + C(n_k)$$

(3) 能解节点。

① 终节点是能解节点;

② 若节点 n 有一外向 K-连接指向子节点 $n_1,n_2,\cdots,n_k$,且这些子节点都是能解节点,则 n 是能解节点;

(4) 不能解节点。

① 非终节点的叶节点是不能解节点;

② 若节点 n 的每一个外向连接都至少指向一个不能解节点,则 n 是不能解节点。

关于能解节点和不能解节点如图 3.19 所示。

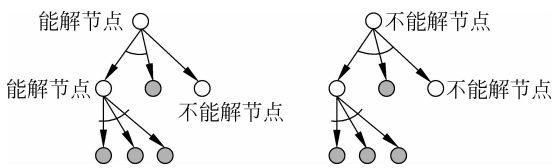

图 3.19 能解节点和不能解节点

3.5.3 与/或图的启发式搜索

与一般图(或图)的搜索过程类似,引入应用领域的启发式知识去引导搜索过程,可以显著提高搜索的有效性,加速搜索算法的收敛。考虑到与或图中搜索的是解图,非由相邻节点间路径连接成的解路径,所以估算评价函数 $f(n)$ 的第 1 分量 $g(n)$ 没有意义,只须估算第 2 分量 $h(n)$。注意,$h(n)$ 也非对于最小路径代价的估计,而是对于最小解图代价的估计。另外,由于与或图中子节点或子节点组间可以存在“或”关系,所在搜索过程中会同时出现多个候选的待扩展局部解图,应估计所有这些局部解图的可能代价,并从中选择一个可能代价最小的用于下一步搜索。由于解图以递归方式生成,解图的代价也以递归方式计算,所以一旦某父节点 n 的由外向 K-连接指向的子节点($n_1,n_2,\cdots,n_k$)每个都估算了其 $h(n_i)$ 的值,($i=1,2,\cdots,k$),则从父节点 n 到终节点集合解图的可能代价 $f(n)$ 可以用公式:

$$f(n) = K + h(n_1) + h(n_2) + \cdots + h(n_k)$$

计算,并用于取代原先在扩展出节点 n 时直接基于 $h(n)$ 估算而得出的 $f(n)$ 值。显然,基于子节点 $h(n_i)$ 算出的 $f(n)$ 更为准确。如此递归计算,可以计算出更为准确的 $f(n_0)$,即从初始状态节点到终节点集合的解图的可能代价。

下面就给出实现与或图启发式搜索的算法 AO*,然后再讨论该算法应用的若干问题。

1. AO* 算法

设:

G——指示搜索图;

G'——被选中的待扩展局部解图;

LGS——候选的待扩展局部解图集;

n_0——指示根节点,即初始状态节点;

n——被选中的待扩展节点;

$f_i(n_0)$——第 i 个候选的待扩展局部解图的可能代价。

该算法的实现过程如下：

(1) $G:=n_0$，LGS 为空集。

(2) 若 n_0 是终节点，则标记 n_0 为能解节点；否则计算 $f(n_0)=h(n_0)$，并把 G 作为 0 号候选局部解图加进 LGS。

(3) 若 n_0 标记为能解节点，则算法成功返回。

(4) 若 LGS 为空集，则搜索失败返回；否则从 LGS 选择 $f_i(n_0)$ 最小的待扩展局部解图作为 G'。

(5) G' 中选择一个非终节点的外端节点(尚未用于扩展出子节点的节点)作为 n。

(6) 扩展 n，生成其子节点集，并从中删去导致有环的子节点以及和它们有“与”关系的子节点；若子节点集为空，则 n 是不能解节点，从 LGS 删去 G'(因为 G' 不可能再扩展为解图)；否则，计算每个子节点 n_i 的 $f(n_i)$，并通过建立外向 K-连接将所有子节点加到 G 中。

(7) 若存在 j 个($j>1$)外向 K-连接，则从 LGS 删去 G'，并将 j 个新局部解图加进 LGS。

(8) G' 中或在取代 G' 的 j 个新局部解图中用公式 $f(n)=K+h(n_1)+h(n_2)+\cdots+h(n_k)$ 的计算结果取代原先的 $f(n)$，并传递这种精化的作用到 $f_i(n_0)(i=1,2,\cdots,j)$；同时将作为终节点的子节点标记为能解节点，并传递节点的能解性。

(9) 返回语句(3)。

下面就以如图 3.17 所示与或图为简例，观察如何应用上述 AO* 算法来搜索解图。为了方便描述，给图中的每个节点进行编号，从 n_0 到 n_{20}，如图 3.20 所示。

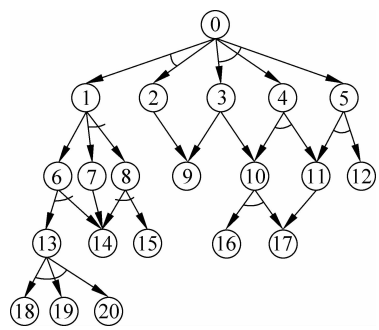

图 3.20　节点编号后的与或图

假定在搜索过程中扩展出来的某些节点的启发式函数 $h(n_i)$ 的估算如下：

$$h(n_0)=3,\quad h(n_1)=2,\quad h(n_2)=1,\quad h(n_3)=1,\quad h(n_4)=4,$$
$$h(n_5)=2,\quad h(n_6)=2,\quad h(n_7)=1,\quad h(n_8)=1,\quad h(n_{13})=3。$$

AO* 算法工作的第 1 个循环扩展根节点 n_0，产生 2 个候选的局部解图，编号为 1(对应于 2-连接)和 2(对应于 3-连接)，加入 LGS 并删去 0 号局部解图。鉴于 $f_1(n_0)=5$ 而 $f_2(n_0)=10$，第 2 个循环就选中 1 号局部解图作为 G'(见图 3.21(a))。随机选中 n_1 加以扩展，建立 2 个外向连接，从 LGS 删去 1 号局部解图，将 2 个扩展出的新局部解图，编号为 3(对应于 1-连接)和 4(对应于 2-连接)，加入 LGS。鉴于 $f_3(n_0)=6$ 而 $f_4(n_0)=7$，第 3 个循环就选中 3 号局部解图作为 G'。随机选中 n_6 加以扩展，建立 1 个外向连接(并由此扩展了 3 号局部解图)，并使 $f_3(n_0)=9$。第 4 个循环就选中 4 号局部解图作为 G'(此时 $f_4(n_0)=7$，最小)，随机选中 n_7 加以扩展，建立 1 个外向连接，并维持 $f_4(n_0)$ 不变。由于新扩展出的节点 n_{14} 是终节点，标记其为能解节点，并递归地标记节点 n_7 为能解节点。第 5 个循环仍选中 4 号局部解图作为 G'，随机选中 n_8 加以扩展，建立 1 个外向连接，并使 $f_4(n_0)=8$。标记新扩展出的终节点 n_{15} 为能解节点，递归地标记节点 n_8 和 n_1 为能解节点。第 6 个循环仍选中 4 号局部解图作为 G'，扩展节点 n_2，标记新扩展出的终节点 n_9 为能解节点，递归地标记节点 n_2 和 n_0 为解节点。至此算法 AO* 成功搜索到解图，且解图代价为 8。

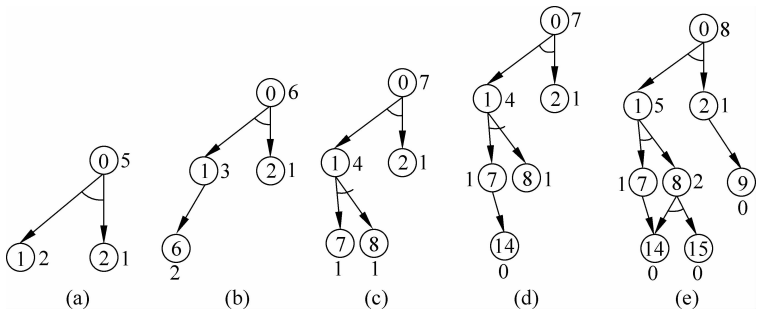

图 3.21 算法 AO* 搜索过程中解图的形成

2. 算法应用的若干问题

(1) 从局部解图中选择加以扩展的节点。

鉴于与或图搜索的是解图而非解路径,所以选择 $f(n)=h(n)$的值最小的节点加以扩展并不一定会加速搜索过程。倒是应选择导致解图代价发生较大变化的节点优先加以扩展,以使搜索的注意力快速地聚焦到实际代价较小的候选解图上。然而,这种选择需要附加的启发式知识。若应用领域挖掘不出这样的启发式知识,可随机选择加以扩展的节点。

(2) AO* 算法的可采纳性。

AO* 算法的应用要求遵从以下约束:总能满足 $h(n)\leqslant h^*(n)$,且确保 $h(n)$满足单调限制条件。只有遵从该约束,AO* 算法是可采纳的,即当某与或图存在解图时,应用 AO* 算法一定能找出代价最小的解图。

类似于算法 A*,$h^*(n)$是实际的代价最小解图找到时解图的代价,我们通常只能设计接近于 $h^*(n)$的 $h(n)$。单调限制条件表示为:

$$h(n)\leqslant K+h(n_1)+h(n_2)+\cdots+h(n_k)$$

$n_1,n_2,\cdots,n_k$ 是节点 n 通过 K-连接指向的子节点。若将 $h(n)$的值视为粗略的估计,$K+h(n_1)+h(n_2)+\cdots+h(n_k)$的值视为细致的计算,则单调限制可理解为:粗略的估计总是不超过细致的计算。

(3) 搜索算法 AO* 与 A* 的比较。

① AO* 应用于与或图搜索,且搜索的是解图;而 AO* 则应用于一般图(或图)搜索,且搜索的是解答路径。

② AO* 选择估算代价最小的局部解图加以优先扩展;而 A* 选择估算代价最小的路径加以优先扩展。

③ AO* 不需考虑评价函数 $f(n)$的分量 $g(n)$,只需对新扩展出的节点 n 计算 $h(n)$,以用于修正 $f_i(n_0)$;而 A* 则需同时计算分量 $g(n)$和 $h(n)$,以评价节点 n 是否在代价最小的路径上。

④ AO* 应用 LGS 存放候选的待扩展局部解图,并依据 $f_i(n_0)$值排序;而 A* 则应用 OPEN 表和 CLOSE 表分别存放待扩展节点和已扩展节点,并依据 $f(n)$值排序。

(4) 解图代价的重复计算。

解图中某些子节点可能会有多个父节点,或者说多个节点的外向连接符可能指向同一

个子节点。依据前述解图代价的递归计算方式，显然这种子节点到终节点集合解图的代价在计算自根节点 n_0 出发的解图时被重复累计了。为正确计算解图的代价，必须删除重复的累计。例如图 3.21(d)中的解图节点 n_{10} 和 n_{11} 到终节点 n_{16} 和 n_{17} 的解图代价被分别重复累计了 2 次，如此整个从根节点 n_0 到终节点集的解图代价为 14，若删除重复的累计，实际解图代价为 11。

3.6 博弈

广义的博弈涉及人类各方面的对策问题，如军事冲突、政治斗争、经济竞争等。博弈提供了一个可构造的任务领域，在这个领域中，具有明确的胜利和失败。同样，博弈问题对人工智能研究提出了严峻的挑战。例如，如何表示博弈问题的状态、博弈过程和博弈知识等。所以，在人工智能中，通过计算机下棋等研究博弈的规律、策略和方法，是有实用意义的。

机器博弈的研究广泛而深入。早在 20 世纪 50 年代，就有人设想利用机器智能来实现机器与人的博弈。国内外许多知名学者和知名科研机构都曾经涉足这方面的研究，历经半个多世纪，到目前为止已经取得了许多惊人的成就。1997 年 IBM 的“深蓝”战胜了国际象棋世界冠军卡斯帕罗夫，震惊了世界。除此之外，加拿大阿尔伯塔大学的奥赛罗程序 Logistello 和西洋跳棋程序 Chinook 也相继成为信息游戏世界冠军，而西洋双陆棋这样的存在非确定因素的棋类也有了美国卡内基·梅隆大学的西洋双陆棋程序 BKG 这样的世界冠军。对围棋、中国象棋、桥牌、扑克等许多种其他种类游戏博弈的研究也正在进行中。

这里讲的博弈是二人博弈，“二人零和、全信息、非偶然”博弈，博弈双方的利益是完全对立的。

所谓“二人零和”，是指在博弈中只有“敌、我”二方。且双方的利益完全对立，其赢得函数之和为零，即

$$\phi_1 + \phi_2 = 0$$

式中，ϕ_1 为我方赢得(利益)；ϕ_2 为敌方赢得(利益)。

即，博弈的双方有三种结局：

(1) 我胜：$\phi_1 > 0$；敌负：$\phi_2 = -\phi_1 < 0$。

(2) 我负：$\phi_1 = -\phi_2 < 0$；敌胜：$\phi_2 > 0$。

(3) 平局：$\phi_1 = 0, \phi_2 = 0$。

通常，在博弈过程中，任何一方都希望自己胜利。双方都采用保险的博弈策略，在最不利的情况下，争取最有利的结果。因此，在某一方当前有多个行动方案可供选择时，总是挑选对自己最为有利而对对方最为不利的那个行动方案。

所谓“全信息”，是指博弈双方都了解当前的格局及过去的历史。

所谓“非偶然”，是指博弈双方都可根据得失大小进行分析，选取我方赢得最大，敌方赢得最小的对策，而不是偶然的随机对策。

另外一种博弈是机遇性博弈，是指不可预测性的博弈，如掷硬币游戏等。这种博弈不在本节讨论的范围。

先来看一个例子，假设有七个钱币，任一选手只能将已分好的一堆钱币分成两堆个数不

等的钱币,两位选手轮流进行,直到每一堆都只有一个或两个钱币,不再能分为止,哪个遇到不能再分的情况,则就为输。

用数字序列加上一个说明表示一个状态,其中数字表示不同堆中钱币的个数,说明表示下一步由谁来分,如(7,MIN)表示只有一个由七个钱币组成的堆,由 MIN 走,MIN 有三种可供选择的分法,即(6,1,MAX),(5,2,MAX),(4,3,MAX)其中 MAX 表示另一对手走,不论哪一种方法,MAX 在它基础上再作符合要求的再分,整个过程如图 3.22 所示。在图中已将双方可能的分法完全表示出来了,而且从中可以看出,无论 MIN 开始时怎么走法,MAX 总可以获胜,取胜的策略用双箭头表示。

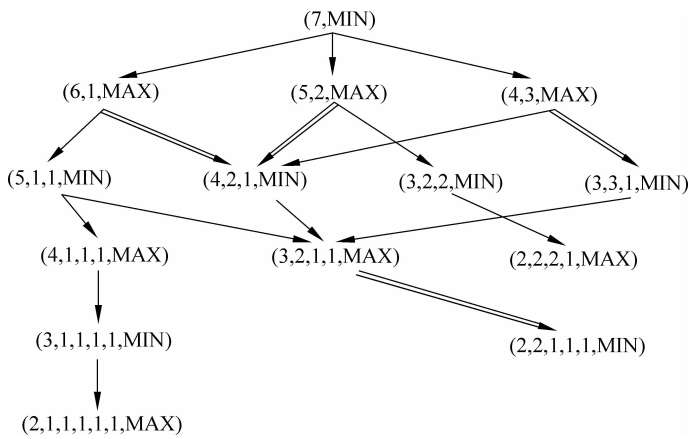

图 3.22 分钱币的博弈

实际的情况没有这么简单,任何一种棋都不可能将所有情况列尽,因此,只能模拟人“向前看几步”,然后做出决策,决定自己走哪一步最有利。也就是说,只能给出几层走法,然后按照一定的估算方法,决定走哪一步棋。

在双人完备信息博弈过程中,双方都希望自己能够获胜。因此当一方走步时,都是选择对自己最有利,而对对方最不利的走法。假设博弈双方为 MAX 和 MIN。在博弈的每一步,可供他们选择的方案都有很多种。从 MAX 的观点看,可供自己选择的方案之间是“或”的关系,原因是主动权在自己手里,选择哪个方案完全由自己决定,而对那些可供 MIN 选择的方案之间是“与”的关系,这是因为主动权在 MIN 手中,任何一个方案都可能被 MIN 选中,MAX 必须防止那种对自己最不利的情况出现。

通过上面的例子可以看出,博弈过程也可以采用与/或树进行知识表达,这种表达形式称为博弈树。由于博弈是敌我双方的智能活动,任何一方不能单独控制博弈过程,而是双方轮流实施其控制对策的过程。因此,博弈树是一种特殊的与/或树。其中,不同级别(深度)的节点,分别交替属于敌我双方,在博弈树生成过程中,由敌我双方轮流进行扩展的,新生成的子节点,是双方交替出现的。

经过分析,博弈树的特点如下:

(1) 与节点、或节点逐级交替出现,敌方、我方逐级轮流扩展其所属节点。

(2) 从我方观点,所有敌方节点部是与节点。因敌方必然选取最不利于我方的一着,扩展其子节点。只要其中有一着(棋步)对我方不利,该节点就对我方不利。换言之,只有该节

点的所有棋步(所有的子节点)皆对我方有利,该节点才对我方有利,故为与节点。

(3) 从我方的观点,所有属于我方的节点都是或节点。因为,扩展我方节点的主动权在我方,可以选取最有利于我方的一着,只要可走的棋步中有一着是有利的,该节点对我方就是有利的。即其子节点中任何一个对我方有利,则该节点对我方有利,故为或节点。

(4) 所有能使我方获胜的终局,都是本原问题,相应的端节点是可解节点;所有使敌方获胜的终局,对我方而言,是不可解节点。

(5) 先走步的一方(我方或敌方)的初始状态相应于根节点。

在人工智能中可以采用搜索方法来求解博弈问题,下面就来讨论博弈中两种最基本的搜索方法。

3.6.1 极大极小过程

极大极小过程是考虑双方博弈若干步之后,从可能的走法中选一步相对好的走法来走,即在有限的搜索深度范围内进行求解。

为此需要定义一个静态估价函数 $e(x)$,以便对棋局的态势做出评估。这个函数可以根据棋局的态势特征进行定义。假定博弈双方分别为 MAX 和 MIN,其中 p 代表棋局。规定:

对于有利于 MAX 方的态势,$e(p)$取正值;对于有利于 MIN 方的态势,$e(p)$取负值;在态势均衡的时候:$e(p)$取零。

MINMAX 的基本思想:

(1) 当轮到 MAX 走步的节点时,MAX 应考虑最好的情况(即 $e(p)$取极大值)。

(2) 当轮到 MIN 走步的节点时,MIN 应考虑最坏的情况(即 $e(p)$取极小值)。

(3) 评价往回倒推时,相应于两位棋手的对抗策略,交替使用(1)和(2)两种方法传递倒推值,直至求出初始节点的倒推值为止。由于我们是站在 MAX 立场上,因此应选择具有最大倒推值的走步。这一过程称为极大极小过程。

例 3.4 如图 3.23 所示是向前看两步,共四层的博弈树,用□表示 MAX,用○表示 MIN,端节点上的数字表示它对应的估价函数的值,在 MIN 处用圆弧连接。

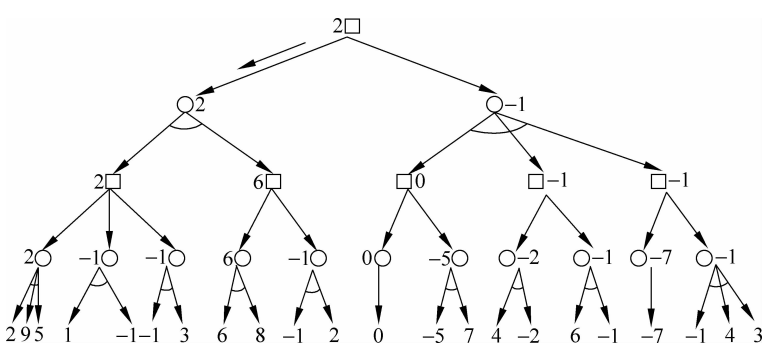

图 3.23 4 层博弈树

图中节点处的数字,在端节点是估价函数的值,称它为静态值,在 MIN 处取最小值,在 MAX 处取最大值,最后 MAX 选择箭头方向的走步。

例 3.5 一字棋游戏。设有一个三行三列的棋盘,如图 3.24 所示,两个棋手轮流走步,每个棋手走步时往空格上摆一个自己的棋子,谁先使自己的棋子成三子一线为赢。设 MAX 方的棋子用×标记,MIN 方的棋子用○标记,并规定 MAX 方先走步。

为了不致于生成太大的博弈树,假设每次仅扩展两层。估价函数定义如下:

设棋局为 p,估价函数为 $e(p)$。

(1) 若 p 是 MAX 必胜的棋局,则 $e(p)=+\infty$。

(2) 若 p 是 MIN 必胜的棋局,则 $e(p)=-\infty$。

(3) 若 p 是胜负未定的棋局,则 $e(p)=e(+p)-e(-p)$。

其中 $e(+p)$表示棋局 p 上有可能使 MAX 成为三子成一线的数目;$e(-p)$表示棋局 p 上有可能使 MIN 成为三子成一线的数目,且具有对称性的两个棋局算作一个棋局。例如,对棋局 1 状态如图 3.25 所示。

其估价函数为

$$e(p) = e(+p) - e(-p) = 6 - 4 = 2$$

在搜索过程中,具有对称性的棋局认为是同一棋局。例如,如图 3.26 所示的棋局可以认为是同一个棋局,这样可以大大减少搜索空间。

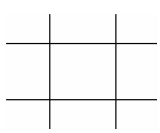

图 3.24 一字棋棋盘

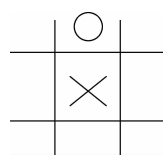

图 3.25 棋局 1

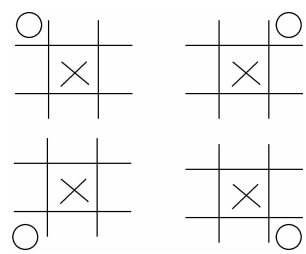

图 3.26 一字棋的棋局状态

假设由 MAX 先走棋,且我们站在 MAX 立场上。图 3.27 给出了 MAX 的第一着走棋生成的博弈树。图中节点旁的数字分别表示相应节点的静态估值或倒推值。由图可以看出,对于 MAX 来说最好的一着棋是 S_3,因为 S_3 比 S_1 和 S_2 有较大的估值。

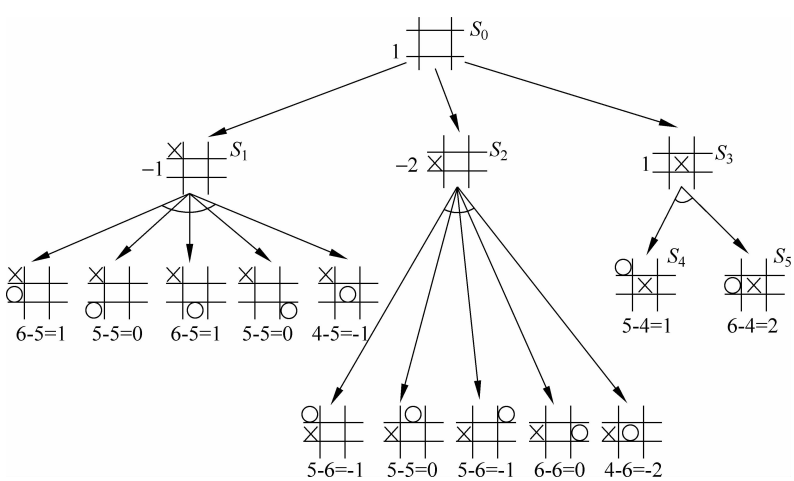

图 3.27 一字棋极大极小搜索

3.6.2 α-β过程

上面讨论的极大极小过程先生成一棵博弈搜索树，而且会生成规定深度内的所有节点，然后再进行估值的倒推计算，这样使得生成博弈树和估计值的倒推计算两个过程完全分离，因此搜索效率较低。如果能边生成博弈树，边进行估值的计算，则可能不必生成规定深度内的所有节点，以减少搜索的次数，这就是下面要讨论的α-β过程。

α-β过程就是把生成后继和倒推值估计结合起来，及时剪掉一些无用分支，以此来提高算法的效率。具体的剪枝方法如下：

(1) 对于一个与节点MIN，若能估计出其倒推值的上确界β，并且这个β值不大于MIN的父节点(一定是或节点)的估计倒推值的下确界α，即α≥β，则就不必再扩展该MIN节点的其余子节点了(因为这些节点的估值对MIN父节点的倒推值已无任何影响了)。这一过程称为α剪枝。

(2) 对于一个或节点MAX，若能估计出其倒推值的下确界α，并且这个α值不小于MAX的父节点(一定是与节点)的估计倒推值的上确界β，即α≥β，则就不必再扩展该MAX节点的其余子节点了(因为这些节点的估值对MAX父节点的倒推值已无任何影响了)。这一过程称为β剪枝。

一个α-β剪枝的具体例子，如图3.28所示。其中最下面一层端节点旁边的数字是假设的估值。

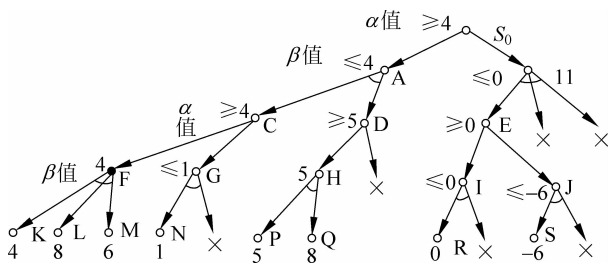

图3.28 一个α-β剪枝的具体例子

在图3.28中，K、L、M的估值推出节点F的倒推值为4，即F的β值为4，由此可推出节点C的倒推值≥4。记C的倒推值的下界为4，不可能再比4小，故C的α值为4。

由节点N的估值推知节点G的倒推值小于≤1，无论G的其他子节点的估值是多少，G的倒推值都不可能比1大。因此，1是G的倒推值的上界，所以G的值≤1。另已知C的倒推值≥4，G的其他子节点又不可能使C的倒推值增大。因此对G的其他分支不必再搜索，相当于把这些分枝剪去。由F、G的倒推值可推出节点C的倒推值≥4，再由C可推出节点A的倒推值≤4，即A的β值为4。另外，由节点P、Q推出的节点H的倒推值为5，因此D的倒推值≥5，即D的α值为5。此时，D的其他子节点的倒推值无论是多少都不能使D及A的倒推值减少或增大，所以D的其他分枝被剪去，并可确定A的倒推值为4。以此类推，最终推出S_0的倒推值为4。

通过上面的讨论可以看出，α-β过程首先使搜索树的某一部分达到最大深度，这时计算出某些MAX节点的α值，或者是某些MIN节点的β值。随着搜索的继续，不断修改个别

节点的 α 或 β 值。对任一节点,当其某一后继节点的最终值给定时,就可以确定该节点的 α 或 β 值。当该节点的其他后继节点的最终值给定时,就可以对该节点的 α 或 β 值进行修正。注意:α、β 值修改有如下规律:

(1) MAX 节点的 α 值永不下降;

(2) MIN 节点的 β 值永不增加。

因此可以利用上述规律进行剪枝,一般可以停止对某个节点搜索,即剪枝的规则表述如下:

(1) 若任何 MIN 节点的 β 值小于或等于任何它的先辈 MAX 节点的 α 值,则可停止该 MIN 节点以下的搜索,然后这个 MIN 节点的最终倒推值即为它已得到的 β 值。该值与真正的极大极小值的搜索结果的倒推值可能不相同,但是对开始节点而言,倒推值是相同的,使用它选择的走步也是相同的。

(2) 若任何 MAX 节点的 α 值大于或等于它的 MIN 先辈节点的 β 值,则可以停止该 MAX 节点以下的搜索,然后这个 MAX 节点处的倒推值即为它已得到的 α 值。

当满足规律(1)而减少了搜索时,进行了 α 剪枝;而当满足规律(2)而减少了搜索时,进行了 β 剪枝。保存 α 和 β 值,并且一旦可能就进行剪枝的整个过程通常称为 α-β 过程,当初始节点的全体后继节点的最终倒推值全部给出时,上述过程便结束。在搜索深度相同的条件下,采用这个过程所获得的走步总跟简单的极大极小过程的结果是相同的,区别只在于 α-β 过程通常只用少得多的搜索便可以找到一个理想的走步。

3.7 小结

本章所讨论的知识的搜索策略是人工智能研究的一个核心问题。搜索是人工智能的一种问题求解方法,搜索策略决定着问题求解的一个推理步骤中知识被使用的优先关系。在搜索中知识利用得越充分,求解问题的搜索空间就越小。对这一问题的研究曾经十分活跃,而且至今仍不乏高层次的研究课题。正如知识表示一样,知识的搜索与推理也有众多的方法,同一问题可能采用不同的搜索策略,而其中有的比较有效,有的不大适合具体问题。本章介绍的几种搜索策略主要适于解决几种不太复杂的问题。

本章首先介绍了基于状态空间图的搜索技术,给出了图搜索的基本概念,分析了状态空间搜索和一般图搜索算法。在应用盲目搜索进行求解过程中,一般是“盲目”穷举的,即不运用特别信息的。盲目搜索中最具代表的算法是宽度优先搜索和深度优先搜索。当状态空间比较大的时候,由于宽度优先需要很大的存储空间,所以宽度优先是不合适的。在很多典型的人工智能问题中,深度优先有着很多的应用。但深度优先不是一种完备的方法,而且,人们常常也不是要沿着某一分支不断地扩展下去,因此迭代加深搜索在这种情况下更适合一些。和 A^* 算法结合,迭代加深搜索得到 IDA^* 算法,该算法已经有了很多的研究,并且在并行结构上实现。

在本章给出的启发式搜索中,最流行的是 A^* 算法和 AO^* 算法。A^* 算法用于或(OR)图,而 AO^* 算法用于与或图。A^* 算法用于状态空间中寻找目标,以及从起始节点到目标节点的最优路径问题。AO^* 算法用于确定实现目标的最优路径。最近,有人通过机器学习的

方法增强状态空间中节点的启发式信息，对 A* 算法算法进行扩展。

另外本章还介绍了博弈问题，这可以看作是一种特殊的与或搜索问题。我们给出了极大极小方法和 α-β 剪枝技术。这两类技术在现在的一些博弈类游戏软件中是必不可少的。

习题

3.1 理解一般图搜索算法，OPEN 表和 CLOSE 表的作用是什么？为何要标记从子节点到父节点的指针？举例说明对三类子节点处理方式的差异。

3.2 对比深度优先和宽度优先的搜索方法，为何说它们都是盲目搜索方法？启发式知识对搜索的指导作用体现在哪些方面？用启发式知识排序 OPEN 表中的节点有什么优点？启发式排序又分哪两种方式？

3.3 理解启发式搜索算法 A*、其使用的评价函数，并就对后继子节点的处理，比较算法 A* 与一般图搜索算法的差别。

3.4 举例说明启发式函数 $h(n)$的强弱对搜索效率的影响。实用上，如何使图搜索更为有效？

3.5 什么是问题规约？问题规约的操作算子与一般图搜索有何不同？为什么应用问题规约得到的状态空间可表示为与或图？

3.6 举例说明与或图搜索的基本概念：K-连接，根、叶、终节点，解图，解图代价，能解节点和不能解节点。

3.7 阐述与或图启发式搜索的算法 AO*，AO* 的可采纳性条件是什么？为什么扩展局部解图时，不必选择 $h(n)$值最小的节点加以扩展？

3.8 比较搜索算法 AO* 和 A*，并说明两者差异的理由。

3.9 有一农夫带一只狐狸、一只小羊和一篮菜过河(从左岸到右岸)。假设船太小，农夫每次只能带一样东西过河。考虑到安全，无农夫看管时，狐狸和小羊不能在一起，小羊和那篮菜也不能在一起。请为该问题的解决设计状态空间，并画出状态空间图。

3.10 修改一般图搜索算法，使其分别刚好能实现深度优先的搜索策略和爬山法。

3.11 某扩展中的搜索图如图 3.29 所示，已被扩展的节点涂黑，待扩展的节点表示为空心圆圈，当前被扩展节点表示为双圆圈，并以虚线连到其生成的后继节点。设相邻节点间路径等长，请依据一般图搜索算法作指针设置和修改工作(并将虚线改为实线)。

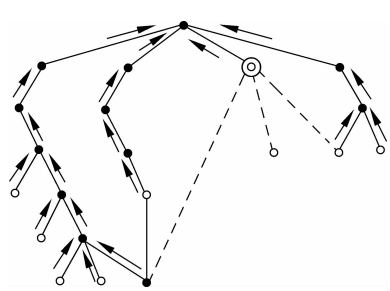

图 3.29 某扩展中的搜索图

3.12 应用启发式搜索算法 A 解决如图 3.30 所示的八数码问题。

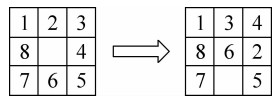

图 3.30 八数码问题

设评价函数 $f(n)=d(n)+p(n)$,画出搜索图,并给出各搜索循环结束时 OPEN 和 CLOSE 表的内容。

3.13 滑动积木游戏具有一个如图 3.31 所示的初始结构。

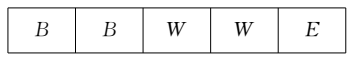

图 3.31 滑动积木游戏初始结构

其中,B 表示黑色将牌,W 表示白色将牌,E 表示空格。游戏的规定走法是:

(1) 任意一个将牌可移入相邻的空格,规定其代价为 1;

(2) 任何一个将牌可相隔 1 个其他的将牌跳入空格,其代价为跳过将牌的数目加 1。

游戏要达到的目标是把所有 W 都移到 B 的左边。对这个问题,定义一个启发函数 $h(n)$,并给出用这个启发函数产生的搜索树。

3.14 巡回售货员问题如图 3.32 所示。图中圆圈表示城市,城市之间垂线旁的数字为城市之间路程的费用。要求从 A 城出发,经过其他各城一次且仅一次,最后回到 A 城,请用 A* 算法求解最小费用的路线,给出估价函数和搜索图。

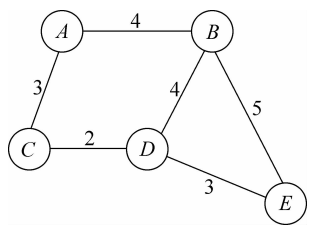

图 3.32 巡回售货员问题

3.15 对于图 3.24 中的一字棋应用 α-β 剪枝算法画出其搜索过程。