第5章

搜索策略


讲解视频


人物介绍

第4章给出了求解问题的方法。但是在求解过程中，具体的每一步往往有多种选择。例如，有多条知识可以用，或者有多种操作可以用。哪一个是最佳选择呢？不同的选择方案首先会影响求解问题的效率，其次可能会影响是否可得到解（或者最优解）。搜索策略决定从起点到终点的每一步如何走，特别是面对“岔路”时如何选择。在具体求解问题（推理）时，运用合适的搜索策略至关重要。本章主要介绍几种常用的搜索策略。

5.1 搜索的基本概念

根据问题的实际情况寻找可用知识，并以此构造出一条代价较小的推理路线，使得问题获得圆满解决的过程叫作搜索。简单地说，搜索就是利用已知条件（知识）寻求解决问题的办法的过程。人工智能所研究的大多是结构不良或非结构化的问题。对于这些问题，一般很难获得其全部信息，更没有现成的算法可供求解使用。因此，只能依靠经验，利用已有知识逐步摸索求解。搜索是人工智能中的一个基本问题。理论上有解的问题，在现实世界中由于各种约束（主要是时空资源的约束）而未必能得到解（或者最优解）。搜索策略最关心的问题就是能否尽可能快地得到（有效或者最优）解。搜索策略合适与否直接关系到智能系统的性能和运行效率，尼尔逊把它列为人工智能研究中的4个核心问题之一。通常把常规算法无法解决的问题分为两类：一类是结构不良或非结构化问题；另一类是结构比较好，理论上也有算法可依，但问题本身的复杂性超过了计算机在时间、空间上的局限。对于这两类问题，我们往往无法用某些巧妙的算法来获取它们的精确解，而只能利用已有的知识一步步摸索着前进。在这个过程中就存在着如何寻找可用知识，确定出开销尽可能少的一条推理路线的问题。

对那些结构比较好，理论上有算法可依的问题，如果问题或算法的复杂性较高（如按指数形式增长），由于受计算机在时间和空间上的限制，也无法付诸实用。这就是人们常说的组合爆炸问题。例如，64阶汉诺塔问题有 3^{64} 种状态，一共需要移动约1800亿亿步

(18446744073709551615)，才能最终完成整个过程。这是一个天文数字，没有人能够在有生之年通过手动的方式来完成它。因此仅从空间上来看，如此巨大的状态数量是一个当今任何一台普通计算机都无法存储的问题。可见，理论上有算法可依的问题实际上不一定可解。像这类问题，也需要采用搜索的方法来进行求解。

对于搜索的类型，可根据搜索过程是否使用启发式信息分为盲目搜索和启发式搜索，也可根据问题的表示方式分为状态空间搜索和基于树的搜索。

盲目搜索是按预定的控制策略进行搜索，在搜索过程中获得的中间信息并不改变控制策略。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，因此这种搜索具有盲目性，效率不高，不便于复杂问题的求解。启发式搜索是在搜索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的方向前进，加速问题的求解过程，并找到最优解。

状态空间搜索是指用状态空间法来求解问题所进行的搜索。基于树的搜索通常指的是与或树搜索和博弈树搜索，它们是用问题归约法来求解问题时所进行的搜索。状态空间法与问题归约法是人工智能领域最基本的两种问题求解方法，因此，接下来将从盲目搜索和启发式搜索两个方面介绍基于状态空间的搜索，然后再以相同的角度介绍使用问题归约法的基于树的搜索。

5.2　基于状态空间的盲目搜索

虽然总体上看盲目搜索不如启发式搜索那么高效，但由于启发式搜索需要抽取与问题本身有关的一些特别难以提取的特征信息，因此有时盲目搜索也是一种直截了当的搜索策略。前文已说过，在人工智能中是通过搜索来生成状态空间对问题进行求解的。其基本思想是：首先把问题的初始状态(即初始节点)作为当前状态，选择适合的算符对其进行操作并生成一组子状态(或称后继状态、后继节点、子节点)，然后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为止。下面我们首先看一下状态空间的搜索过程。

5.2.1　状态空间的搜索过程

下面列出状态空间的一般搜索过程。在此之前先对搜索过程中要用到的两个数据结构(OPEN 表与 CLOSED 表)做些简单说明。

状态节点在 OPEN 表中的排列顺序是不同的。OPEN 表用于存放刚生成的节点，其形式见表 5-1。对于不同的搜索策略，其节点的排列顺序也不同。例如，对广度优先搜索而言，节点按生成的顺序排列，先生成的节点排在前面，后生成的节点排在后面。

表 5-1　OPEN 表

状态节点	父节点

CLOSED 表用于存放将要扩展或者已扩展的节点，其形式见表 5-2。所谓对一个节点进行“扩展”，是指用合适的算符对该节点进行操作，生成一组子节点。

表 5-2 CLOSED 表

编号	状态节点	父节点

搜索的一般过程如下：

(1) 把初始节点 S_0 放入 OPEN 表，并建立目前只包含 S_0 的图，记为 G。

(2) 检查 OPEN 表是否为空，若为空则问题无解，退出。

(3) 把 OPEN 表的第一个节点取出放入 CLOSED 表，并记该节点为节点 n。

(4) 判断节点 n 是否为目标节点。若是，则求得了问题的解，退出。

(5) 考察节点 n，生成一组子节点。把其中不是节点 n 父节点的那些子节点记作集合 M，并把这些子节点作为节点 n 的子节点加入 G。

(6) 针对 M 中子节点的不同情况，分别进行如下处理：

① 对那些未曾在 G 中出现过的 M 的成员设置一个指向父节点(即节点 n)的指针，并将它们放入 OPEN 表。

② 对那些先前已在 G 中出现过的 M 的成员，确定是否需要修改它的指向父节点的指针。

③ 对那些先前已在 G 中出现并且已经扩展了的 M 的成员，确定是否需要修改其后继节点指向父节点的指针。

(7) 按某种搜索策略对 OPEN 表中的节点进行排序。

(8) 执行步骤(2)。

下面对上述过程作一些说明。

上述过程是状态空间的一般搜索过程，具有通用性。在此之后讨论的各种搜索策略都可看作是它的一个特例。各种搜索策略的主要区别是对 OPEN 表中节点排序的准则不同。例如，广度优先搜索把先生成的子节点排在前面，而深度优先搜索则把后生成的子节点排在前面。

一个节点经一个算符操作后一般只生成一个子节点。但适用于一个节点的算符可能有多个，此时就会生成一组子节点。在这些子节点中可能有些是当前扩展节点(即节点 n)的父节点、祖父节点等，此时不能把这些先辈节点作为当前扩展节点的子节点。余下的子节点记作集合 M，并加入图 G 中。这就是步骤(5)的操作。

一个新生成的节点，可能是第一次被生成的节点，也可能是先前已作为其他节点的后继节点被生成过，当前又被作为另外一个节点的后继节点再次生成。此时，它究竟应作为哪个节点的后继节点呢？一般由原始节点到该节点路径上所付出的代价来决定。哪条路径付出的代价小，相应路径上的上一个节点就作为它的父节点。

通过搜索所得到的图称为搜索图。由搜索图中的所有节点及反向指针(在步骤(6)形成的指向父节点的指针)所构成的集合是一棵树，称为搜索树。

在搜索过程中，一旦某个被考察的节点是目标节点(步骤(4))，就得到了一个解。该解由从初始节点到该目标节点所形成的路径的算符构成，而路径则由步骤(6)形成的反向指针指定。

如果在搜索中一直找不到目标节点，而且 OPEN 表中不再有可供扩展的节点，则搜索失败，执行步骤(2)退出。

由于盲目搜索一般适用于其状态空间是树状结构的问题，因此对盲目搜索而言，通常不会出现一般搜索过程步骤(6)中后两点的问题。每个节点经过扩展后生成的子节点都是第一次出现的节点，不必检查并修改指针方向。

由上述搜索过程可以看出，问题的求解过程实际上就是搜索过程。问题求解的状态空间图是通过搜索逐步形成的，边搜索边形成；搜索每前进一步，就要检查是否到达了目标状态，这样可尽量少生成与问题求解无关的状态，既节省了存储空间，又提高了效率。

5.2.2　状态空间的广度优先搜索

广度优先搜索又称为宽度优先搜索。如图 5-1 所示(标号代表搜索次序)，这种搜索是逐层进行的，在对下一层的任意节点进行考察之前，必须完成本层所有节点的搜索。

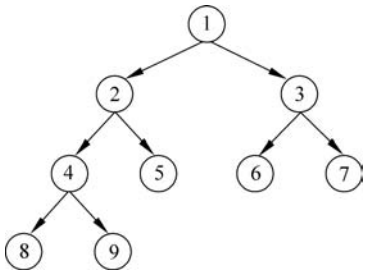

图 5-1　广度优先搜索

广度优先搜索的基本思想是：从初始节点 S_0 开始，逐层地对节点进行扩展并考察它是否为目标节点，在第 n 层的节点没有全部扩展并考察之前，不对第 $n+1$ 层的节点进行扩展。OPEN 表中的节点总是按进入的先后顺序排列，先进入的节点排在前面，后进入的节点排在后面。其搜索过程如下：

(1) 把初始节点 S_0 放入 OPEN 表。

(2) 如果 OPEN 表为空，则问题无解，退出。

(3) 把 OPEN 表的第一个节点(记为节点 n)取出并放入 CLOSED 表。

(4) 考察节点 n 是否为目标节点。若是，则求得了问题的解，退出。

(5) 若节点 n 不可扩展，则执行步骤(2)。

(6) 扩展节点 n，将其子节点放入 OPEN 表的尾部，并为每一个子节点都配置指向父节点的指针，然后执行步骤(2)。广度优先搜索流程如图 5-2 所示。

例 5.1　重排九宫格问题。在 3×3 的方格棋盘上放置分别标有数字 1、2、3、4、5、6、7、8 的 8 张牌，初始状态为 S_0，目标状态为 S_g，如图 5-3 所示。其中，图 5-3(a)是初始状态，图 5-3(b)是目标状态。

可使用的算符有空格左移、空格上移、空格右移和空格下移，即它们只允许把位于空格左、上、右、下边的牌移入空格。要求寻找从初始状态到目标状态的路径。

解：应用广度优先搜索，可得到如图 5-4 所示的搜索树。由图 5-4 可以看出，解路径是 $1(S_0)\rightarrow 3\rightarrow 8\rightarrow 16\rightarrow 26(S_g)$。

广度优先搜索的盲目性较大。当目标节点距离初始节点较远时将会产生许多无用节点，搜索效率低，这是它的缺点。但是，只要问题有解，用广度优先搜索总可以得到解，而且得到的是路径最短的解，这是它的优点。

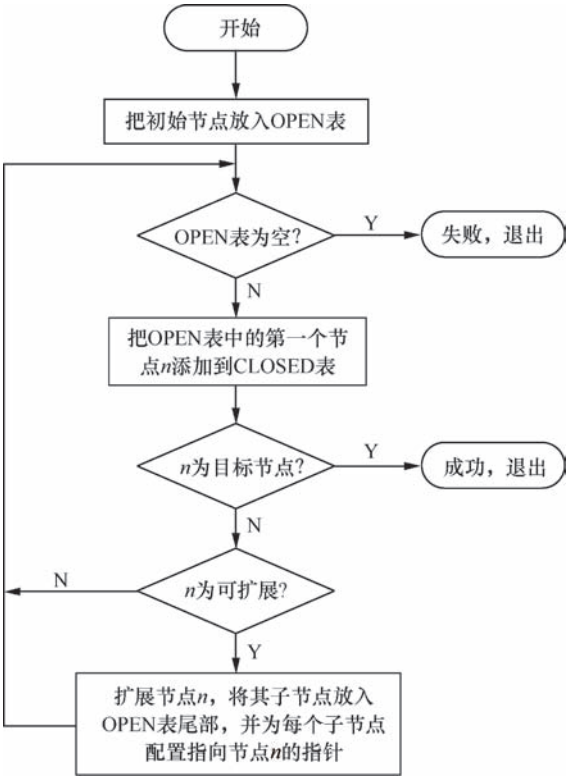

图 5-2　广度优先搜索流程

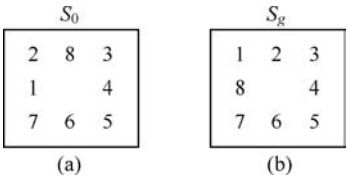

图 5-3　重排九宫格问题的初始状态与目标状态

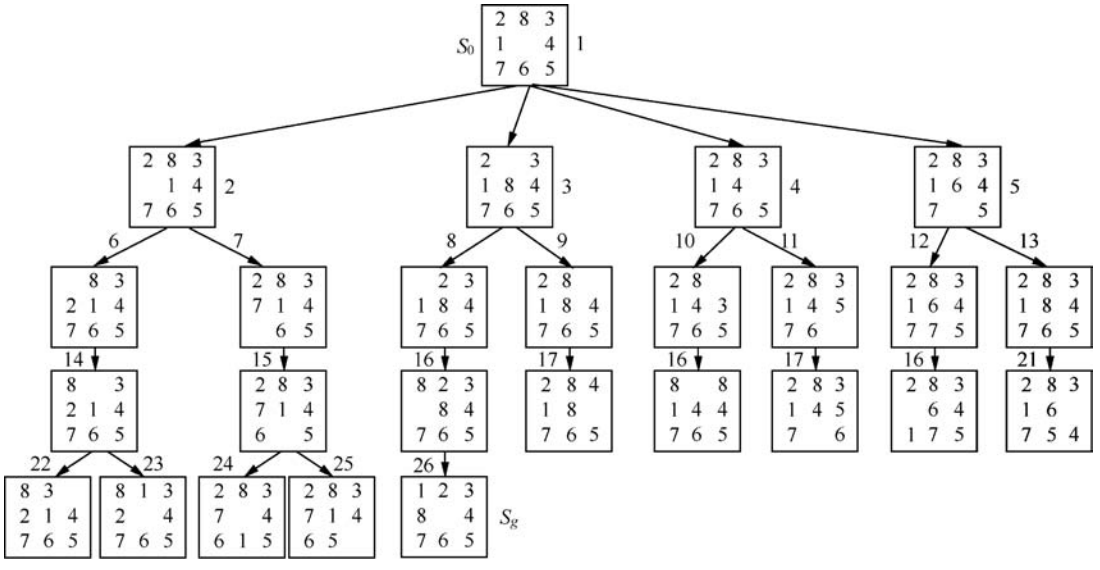

图 5-4　重排九宫格问题的广度优先搜索树

5.2.3 状态空间的深度优先搜索

深度优先搜索是一种后生成的节点先扩展的搜索策略。这种搜索策略的搜索过程是：从初始节点 S_0 开始，在其余节点中选择一个最新生成的节点进行考察，如果该子节点不是目标节点且可以扩展，则扩展该子节点，然后再在此子节点的子节点中选择一个最新生成的节点进行考察，依次向下搜索，直到某个子节点既不是目标节点又不能继续扩展时，才选择其兄弟节点进行考察。OPEN 表是一种栈式存储结构，最先进入的节点排在最后面，最后进入的节点排在最前面。

深度优先搜索算法过程如下：

(1) 把初始节点 S_0 放入 OPEN 表中。

(2) 如果 OPEN 表为空，则问题无解，失败退出。

(3) 把 OPEN 表的第一个节点取出，放入 CLOSED 表，并记该节点为 n。

(4) 考察节点 n 是否为目标节点。若是，则得到问题的解，成功退出。

(5) 若节点 n 不可扩展，则执行步骤(2)。

(6) 扩展节点 n，将其子节点放入 OPEN 表的首部，并为每一个子节点设置指向父节点的指针，然后执行步骤(2)。

深度优先搜索与广度优先搜索的唯一区别是：广度优先搜索是将节点 n 的子节点放入 OPEN 表的尾部；而深度优先搜索是把节点 n 的子节点放入 OPEN 表的首部。仅此一点不同，却使得搜索的路线完全不一样。

我们仍以重排九宫格问题为例，使用深度优先搜索的方式求解。如图 5-5 所示，是用深

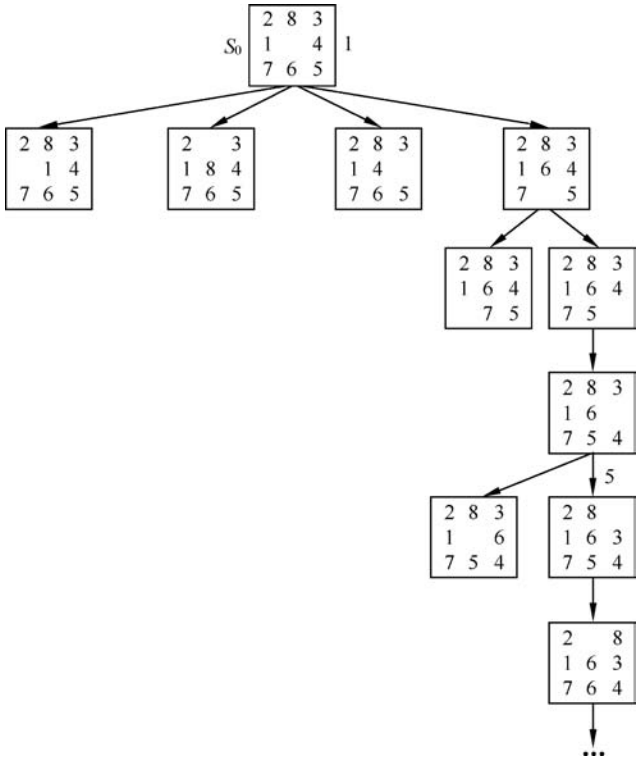

图 5-5 重排九宫格的深度优先搜索树

度优先搜索得到的搜索树的一部分(还可继续往下搜索,因版面原因暂未画出)。可以明显地看出在深度优先搜索中,搜索一旦进入某个分支,就将沿着该分支一直向下搜索。如果目标节点恰好在此分支上,则可较快地得到解。但是,如果目标节点不在该分支上,而该分支又是一个无穷分支,就不可能得到解了。可见,深度优先搜索是不完备的,即使问题有解,通过它也不一定能求得解。

5.3 基于状态空间的启发式搜索

基于状态空间的启发式搜索是一种能够利用搜索过程中得到的问题本身的某些特征信息来引导搜索过程,以使得搜索可以尽快达到目标的一种搜索方式。由于启发式搜索具有针对性,故它可以非常有效地缩小搜索范围,提高搜索的效率。

5.3.1 动态规划

由理查德·贝尔曼(Richard Bellman)创建的动态规划有时称为“正反向”算法,当使用概率时称为维特比算法。动态规划致力于解决由若干交互和交联子问题构成的问题中的受限存储搜索问题。动态规划保存且重用问题求解中已搜索、已求解的子问题轨迹。为了重用子问题轨迹,存储子问题技术有时称为存储部分子目标的解。动态规则是一种常用于串匹配、拼写检查,以及自然语言处理相关领域的重要算法。下面我们用一个简单的例子来说明它。

动态规划需要使用数据结构保存与当前处理状态有关的子问题轨迹,这里使用数组。因为初始化的需要,数组的维数比各串的长度都多 1,在本例中取 8 行 12 列,如图 5-6 所示,数组各元素(x,y)的值反映匹配过程中在该位置全局对准成功。

		B	A	A	D	D	C	A	B	D	D	A
	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0										
B	2											
A	3											
D	4											
C	5											
B	6											
A	7											

图 5-6 初始化阶段使用动态规划完成字符对准数组的第一步

在建立的当前状态中有 3 种可能代价：若两个字符相同则说明成功对准,则向前移动较短串中的一个字符,代价为 1,由数组的列记分；若插入一个新字符,则代价为 1,反映在

数组的行得分中。若要对准的字符是不同的，则代价为 2(移动和插入)；若它们是相同的，则代价为 0，反映在数组的“对角线”中。如图 5-7 所示，表示初始状态，第 1 行和第 1 列渐增 1 表示不断移动或插入字符到空位或空串上。在动态规划算法的正向阶段中，考虑到求解的当前位置部分匹配成功，因此从左上角填充数组。即 x 行和 y 列的交叉点(x,y)的值是 $x-1$ 行 y 列、$x-1$ 行 $y-1$ 列或 x 行 $y-1$ 列中 3 个值之一的函数(相对最小对准问题是最小代价)。这 3 个数组位置具有直到求解完毕的当前位置的对准信息。若在位置(x,y)上有一匹配的字符，则把 0 加到位置$(x-1,y-1)$上；若无字符匹配，则加 2(移动和插入)。移动较短字符串或插入一个字符则加 1，前者增加 y 列前元素的值，后者增加 x 行之上元素的值。持续该过程，直到产生如图 5-8 所示的已填充数组。可以看出，最小代价匹配常接近数组左上到右下“对角线”的位置。

		B	A	A	D	D	C	A	B	D	D	A
	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0	1	2	3	4	5	6	7	8	9	10
B	2	1	2	3	4	5	6	7	6	7	8	9
A	3	2	1	2	3	4	5	6	7	8	9	8
D	4	3	2	3	2	3	4	5	6	7	8	9
C	5	4	3	4	3	4	3	4	5	6	7	8
B	6	5	6	5	4	5	4	5	4	5	6	7
A	7	6	5	4	5	6	5	4	5	6	7	6

图 5-7　反映两个串最大对准信息的完全数组

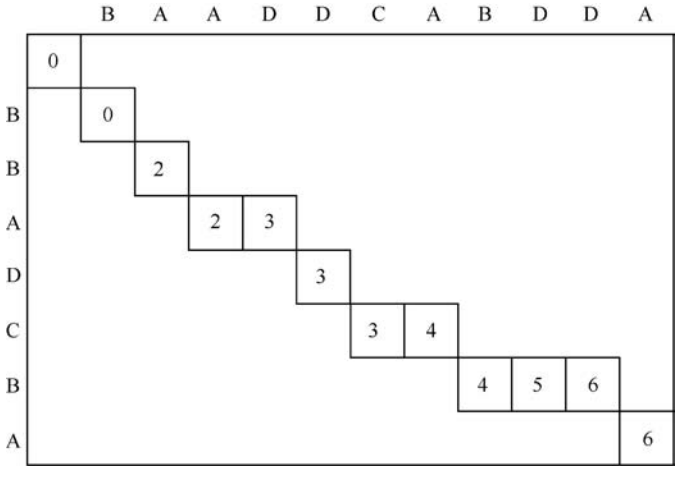

图 5-8　产生一种串对准的动态规划完成的反向部分

一旦数组被填充，便开始算法的反向阶段以产生具体解。即由尽可能好的对准开始，穿过数组追溯，选出一个具体的解对准。我们在最大行列的值上开始该过程，在本例中是 8 行

12 列中的 6。由 6 穿过数组反向移动，每移动一步选出一个产生当前状态的直接状态前驱（由正向阶段），该前驱或为产生该状态的对角线、行或列之一。每当出现横向渐减差时，如接近追溯开端的 6 和 5，我们便选择上一对角线作为匹配的来源；否则使用前一行或列的值。图 5-8 的追溯是几种可能性之一，它产生前面给出的最优串对准。

5.3.2 A* 算法

满足以下条件的搜索过程称为 A* 算法。

（1）把 OPEN 表中的节点按估价函数

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

的值从小到大进行排序（搜索的一般过程的步骤⑦）。

（2）$g(x)$是对 $g^*(x)$的估计，且 $g(x)>0$。

（3）$h(x)$是 $h^*(x)$的下界，即对所有的节点 x 均有

$$h(x)\leqslant h^*(x)$$

其中，$g^*(x)$是从初始节点 S_0 到节点 x 的最小代价；$h^*(x)$是从节点 x 到目标节点的最小代价。若有多个目标节点，则为其中最小的一个。

在 A* 算法中，$g(x)$比较容易得到，它实际上就是从初始节点 S_0 到节点 x 的路径代价，恒有 $g(x)\geqslant g^*(x)$。而且在算法执行过程中随着更多搜索信息的获得，$g(x)$的值呈下降趋势。例如，在图 5-9 中，从节点 S_0 开始，经扩展得到 x_1 与 x_2，且

$$g(x_1)=3,\quad g(x_2)=7$$

对 x_1 进行扩展后得到 x_2 与 x_3，此时

$$g(x_2)=6,\quad g(x_3)=5$$

显然，后来算出的 $g(x_2)$比先前算出的小。

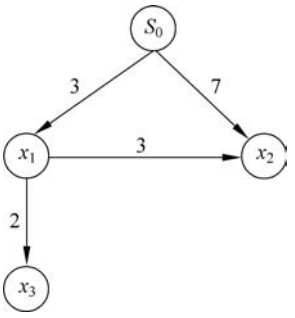

图 5-9 $g(x)$的计算

启发式函数 $h(x)$的确定依赖于具体问题领域的启发式信息。其中，$h(x)\leqslant h^*(x)$的限制是十分重要的，它可保证 A* 算法能找到最优解。

下面来讨论 A* 算法的可纳性。

对于可解状态空间（即从初始节点到目标节点有路径存在）来说，如果一个搜索算法能在有限步内终止，并且能找到最优解，则称该搜索算法是可纳的。

A* 算法是可纳的，即它能在有限步内终止并找到最优解。下面分两个方面来证明这一结论。

对于有限图，其节点个数是有限的，A* 算法一定会在有限步内终止。可见，A* 算法在经过若干次循环之后只可能出现两种情况：一种由于搜索到了目标节点而终止，另一种由于 OPEN 表中的节点被取完而终止。不管发生哪种情况，A* 算法都在有限步内终止。

对于无限图，只要从初始节点到目标节点有路径存在，则 A* 算法也必然会终止。该证明分两步进行。第 1 步先证明在 A* 算法结束之前，OPEN 表中总存在节点 x'。该节点是最优路径上的一个节点，且满足

$$f(x')\leqslant f^*(S_0)$$

设最优路径是 $S_0, x_1, x_2, \cdots, x_m, S_g^*$。由于 A* 算法中的 $h(x)$ 满足 $h(x) \leqslant h^*(x)$，所以 $f(S_0), f(x_1), f(x_2), \cdots, f(x_m)$ 均不大于 $f(S_g^*)$，且 $f(S_g^*) = f^*(S_0)$。

又因为 A* 算法是全局择优的，所以在它结束之前，OPEN 表中一定含有 $S_0, x_1, x_2, \cdots, x_m, S_g^*$ 中的一些节点。设 x' 是最前面的一个，则它必然满足

$$f(x') \leqslant f^*(S_0)$$

至此，第 1 步证明结束。

现在来进行第 2 步的证明。这一步用反证法，即假设 A* 算法不终止，则会得出与第 1 步矛盾的结论，从而说明 A* 算法一定会终止。

假设 A* 算法不终止，并设 e 是图中各条边的最小代价，$d^*(x_n)$ 是从节点 S_0 到节点 x_n 的最短路径长度，则显然有

$$g^*(x_n) \geqslant d^*(x_n) \times e$$

又因为

$$g(x_n) \geqslant g^*(x_n)$$

所以有

$$g(x_n) \geqslant d^*(x_n) \times e$$

因为

$$h(x_n) \geqslant 0, \quad f(x_n) \geqslant g(x_n)$$

故得到

$$f(x_n) \geqslant d^*(x_n) \times e$$

由于 A* 算法不终止，随着搜索的进行，$d^*(x_n)$ 会无限增长，从而使 $f(x_n)$ 也无限增长，这就与第 1 步证明得出的结论矛盾。因为对可解状态空间来说，$f^*(S_0)$ 一定是有限值。所以，只要从初始节点到目标节点有路径存在，即使对于无限图，A* 算法也一定会终止。

5.3.3 爬山法

实现启发式搜索的最简单方法是“爬山法”。爬山法扩展搜索的当前状态，产生该状态的各子节点，并且估价这些子节点。然后选出“最好的”子节点进一步扩展，不保留它的同辈节点和父节点。爬山法类似性急、鲁莽的爬山者使用的策略：沿尽可能陡峭的山路向上爬，直到不能再爬高。因为不保存历史信息，所以算法不可能由失败中恢复。爬山法的典型例子是井字棋博弈中使用的“选择有最多可能获胜途径的状态”。

爬山法的一个主要问题是容易陷在“局部极大值”上。若达到某一状态，该状态有相比它的任何子节点都好的估价，则算法终止。若该状态不是目标，只是局部极大值，则算法不能求出最优解。即在受限的情况下爬山法性能会很好，但由于不能把握全部空间的情况，所以它不能达到全局最好状态。局部极大值问题也表现在重排九宫格问题中。其中，为把一张牌移到目标位置，需把已在目标位置上的其他牌移开。在求解重排九宫格问题中这是必要的，但只是暂时使牌的布局状态变坏。因为从宏观角度看，“较好的”不一定是“最好的”，没有回溯或其他恢复机制的搜索方法不能辨别局部与全局极大值。

图 5-10 是局部极大值问题的示意。假定探测该搜索空间，到达状态 X。X 的各子节点、子子节点等的估价说明爬山法即使向前看多层也会出错。有方法可避免该问题，如使用随机估价函数，但该函数通常无法保证爬山法的最优性能。塞缪尔的西洋跳棋程序给出了爬山法的一种有趣变种。该程序在当时是杰出的，特别是在 20 世纪 50 年代计算机性能受到限制的情况下。该程序不仅把启发式搜索用于西洋跳棋，而且还实现了最优使用受限存储器的算法，以及实现了一种简单形式的机器学习。的确，塞缪尔的许多先进思想现在仍用于博弈和机器学习。

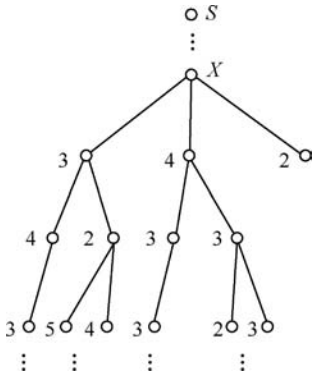

图 5-10　局部极大值问题示意

塞缪尔的程序用几种不同启发式测度的加权求和来估价棋盘状态。和式中的 x_1 表示棋盘的特征，如棋子的优势、棋子的位置、棋盘中心的控制、牺牲棋子获得优势的机会等，甚至可以表示某弈手的诸棋子关于棋盘某轴线的惯性矩。x_1 的系数 c 是可调整的加权，试图模拟该特征在棋盘总估价中的重要性。因此，若棋子的优势比中心的控制更重要，则将反映在棋子的优势系数上。该程序在搜索空间中采用向前看 k 步策略（k 的选择受计算机时、空资源的限制），并且按上述加权公式估价第 k 层上的所有状态。

若加权公式导致一系列着法失败，则程序要调整它的系数 c 以改进性能。大系数的项要为失败负更多责任，并且减小它们的系数，增大较小的系数，使它们对估价有更大影响。若程序获胜，则做相反工作。程序在与人或它自己的另一版本的对弈中得到训练。塞缪尔的程序采用爬山法进行学习，试图通过局部改进加权公式来改善性能。

塞缪尔的西洋跳棋程序能不断改善性能，直到自身达到极高水平。塞缪尔通过检查各加权启发测度的效果并替换不太有效的测度，减少爬山法的某些限制。但该程序仍有某些令人感兴趣的限制。例如，因于受限全局策略，使用加权公式易误入陷阱，此时程序将不足以解决对手前后矛盾的着法。若对手使用广泛变化的策略，或者甚至采用愚笨的着法，则加权公式中的权值会取“杂乱”值，致使程序性能全面下降。

5.3.4　模拟退火算法

模拟退火算法也是基于状态空间的一种启发式搜索算法，它由固体退火原理发展而来。把固体加温到充分高的温度，然后让它慢慢冷却，等再次加温时，固体内部的粒子随温度的升高变成无序状，固体的内能得以增加，慢慢冷却时粒子也渐渐变为有序，此时每个粒子的温度都达到了平衡状态，最后在常温时达到基态，即内能减为最小的状态。由 Metropolis 接受准则可以得知：在温度为 T 时，粒子将达到平衡的概率是 $e^{-\Delta E/(kT)}$。其中 E 是温度为 T 时的内能，ΔE 是它的改变量，k 叫作玻耳兹曼常数。为了进一步说明模拟退火算法，我们可以看如图 5-11 所示的模拟退火算法的流程示意，用固体退火模拟组合优化问题，把内能 E 模拟成目标函数 f，将温度 T 改为控制参数 t，这样我们就能得到解组合优化问题的模拟退火算法。即从初始解 i 和控制参数 t 开始，将当前解进行如下循环：获得新解→计算目标函数的差值→接受或舍弃。与此同时逐步衰减 t 的值，算法收敛后的解即为所求的最优

近似解。以上是在蒙特卡罗迭代求解法的基础上衍生出的一种启发式随机搜索算法。冷却进度表控制着退火过程，该表包括控制参数的初始值 t、每个 t 值所对应的迭代次数 L、终止条件 S 和每次的衰减因子 Δt 这 4 项。

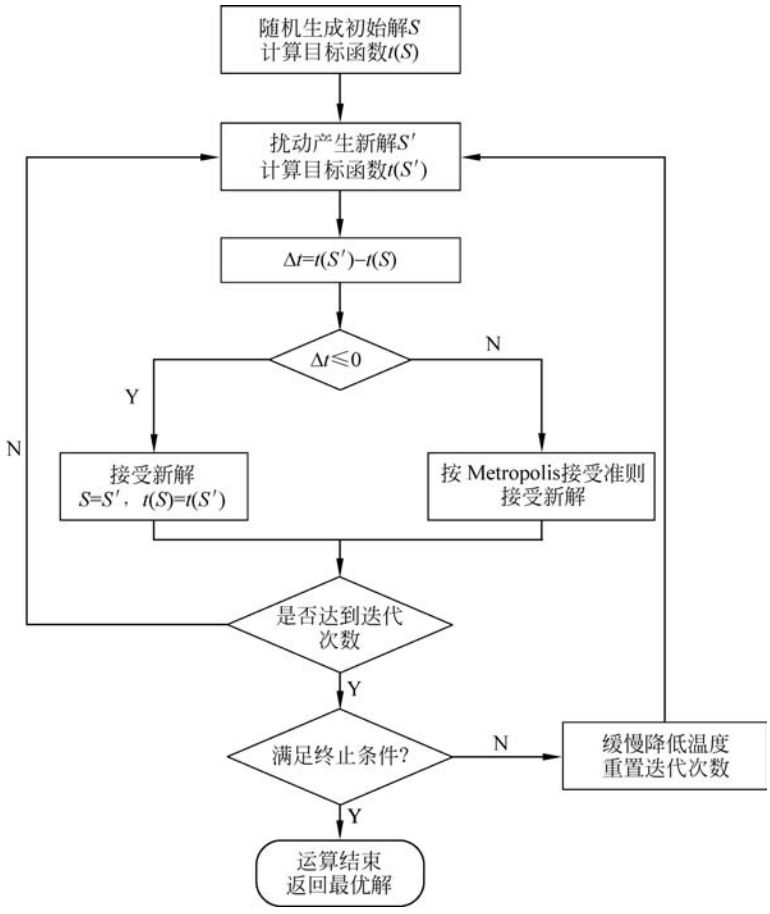

图 5-11　模拟退火算法流程示意

初始解、目标函数和解空间构成了模拟退火算法，下面阐述它的基本思想。

(1) 初始化：将初始温度 T 设为充分大，将初始解状态 S 设置为循环的起点，将每个 T 值所对应的循环次数设为 L。

(2) 对 $k=1,2,\cdots,L$ 重复步骤(3)～(6)。

(3) 得到新解 S'。

(4) 按式 $\Delta t'=C(S')-C(S)$ 计算得到增量，其中评价函数记作 $C(S)$。

(5) 如果 $\Delta t'>0$，那么以概率 $e^{-\Delta t'/T}$ 接受 S' 当作当前新的解；反之，接受 S' 当作当前新的解。

(6) 结束条件通常是连续若干个新解都没有被接受。一旦满足终止条件，那么输出当前解作为找到的最优解，然后结束程序。

(7) 慢慢减小温度 T，直到 $T\to 0$，之后执行步骤(2)。

下面从 4 个阶段来阐述模拟退火算法新解的接受与产生。

（1）新解的产生。具体过程依托于一个产生函数，并由当前解产生一个存在于解空间内的新解。为了便于后续的计算和接受，减少算法耗时，一般选择将当前产生的解只经简单变换而产生新解的简捷方法，例如对构成新解的全部或部分元素进行置换、互换等操作。显然，当前新解的邻域结构是由产生新解的变换方法决定的，这在很大程度上影响了冷却进度表的选取。

（2）计算与新解所对应的相关目标函数差值。由于目标函数差值只由变换部分产生，因此目标函数差值往往按其增量进行计算。事实表明，对于大多数应用来说，此方法是求目标函数差值的最快方法。

（3）判断产生的新解有没有被接受。具体依据是 Metropolis 接受准则，具体表述是：如果 $\Delta t'>0$，则按概率 $e^{\frac{\Delta t'}{T}}$ 接受 S' 作为新的当前解 S，反之直接接受 S' 作为新的当前解 S。

（4）当确认新解被接受的情况下，将当前解替换为新解，具体操作是，将当前解与产生的新解相比较，仅实现其改变的部分即可，并且要同步修正目标函数值。此时，当前解进行了一次循环迭代，我们在此基础上进行下一轮操作。若新解被判定为舍弃，则直接在当前解的基础上进行下一轮操作。

可以看到，模拟退火算法和初始值的选取是没有关系的，此算法求出的解和初始解状态 S（是算法迭代的起点）也没有关系。具备渐近收敛性是模拟退火算法的一大特点，因此模拟退火算法是按概率 l 收敛于全局最优解的一种全局优化算法，这一点在理论上已经得到论证。

至此，我们已经对各种启发式搜索有了一定的了解和认识。在基于状态空间的搜索问题中我们可以将任何搜索算法都概括为两个部分：开发和探索。开发采用了一个准则，即好的解决方案可能彼此接近。一旦找到了一个好的解决方案，就可以检查其周围，确定是否存在更好的解决方案。除此之外，我们还要谨记“没有冒险就没有收获”，意思是更好的解决方案可能存在于状态空间的位置探索区域，因此不要将搜索限制在一个小区域内。理想的搜索算法必须在这两种冲突策略之间取得适当的平衡。

5.4 基于树的盲目搜索

基于树的搜索策略也分为盲目搜索和启发式搜索两大类。本节仅讨论盲目搜索策略，启发式搜索策略将在 5.5 节讨论。基于树的盲目搜索主要是指与或树形式的 3 种搜索，它主要用于复杂问题的简化，其求解过程与状态空间法类似，也是通过搜索来实现对问题求解。下面分别介绍这 3 种搜索。

5.4.1 与或树的一般性搜索

与或树表示法是分而治之思想的最好诠释，通常用于复杂问题的简化。接下来介绍与或树的基本概念。

1）本原问题

不能再分解或变换，而且直接可求解的子问题叫作本原问题。

2) 与节点和或节点

若节点 P 的子节点全部可解时,P 才可解,则称 P 为与节点;若节点 P 的子节点只要有一个可解,P 就可解,则称 P 为或节点。

3) 端节点与终止节点

在与或树中,没有子节点的节点称为端节点;本原问题所对应的节点称为终止节点。显然,终止节点一定是端节点,但端节点不一定是终止节点。

4) 可解节点

在与或树中,满足下列条件之一者,称为可解节点:

(1) 它是一个终止节点。

(2) 它是一个或节点,且其子节点中至少有一个是可解节点。

(3) 它是一个与节点,且其子节点全部是可解节点。

5) 不可解节点

关于可解节点的三个条件均不满足的节点是不可解节点。

使用与或树解决问题时,首先要定义问题的描述方法及分解或变换问题的算符。然后就可用它们通过搜索树生成与或树,从而求得原始问题的解。

由此可以看出,一个节点是否为可解节点是由它的子节点确定的。对于一个“与”节点,只有当其子节点全部为可解节点时,它才为可解节点;只要子节点中有一个为不可解节点,它就是不可解节点。对于一个“或”节点,只要子节点中有一个是可解节点,它就是可解节点;只有全部子节点都是不可解节点时,它才是不可解节点。像这样由可解子节点来确定父节点、祖父节点等为可解节点的过程称为可解标示过程;由不可解子节点来确定父节点、祖父节点等为不可解节点的过程称为不可解标示过程。在与或树的搜索中将反复使用这两个过程,直到初始节点(即原始问题)被标示为可解或不可解节点为止。

下面给出与或树的一般搜索过程。

(1) 把原始问题作为初始节点 S_0,并把它作为当前节点。

(2) 应用分解或等价变换算符对当前节点进行扩展。实际上就是把原始问题变换为等价问题或者分解成几个子问题。

(3) 为每个子节点设置指向父节点的指针。

(4) 选择合适的子节点作为当前节点,反复执行步骤(2)和步骤(3)。在此期间要多次调用可解标示和不可解标示过程,直到初始节点被标示为可解节点或不可解节点为止。

由这个搜索过程所形成的节点和指针结构称为搜索树。

与或树搜索的目标是寻找解树,从而求得原始问题的解。如果在搜索的某一时刻,通过可解标示过程可确定初始节点是可解的,则由此初始节点及其下属的可解节点就构成了解树。如果在某一时刻被选为扩展的节点不可扩展,并且它不是终止节点,则此节点就是不可解节点。此时可应用不可解标示过程确定初始节点是否为不可解节点,如果可以肯定初始节点是不可解的,则搜索失败,否则继续扩展节点。

可解与不可解标示过程都是自下而上进行的,即由子节点的可解性确定父节点的可解性。由于与或树搜索的目标是寻找解树,因此,如果已确定某个节点为可解节点,则其不可解的后继节点就不再有用,可从搜索树中删去。同样,如果已确定某个节点是不可解节点,则其全部后继节点都不再有用,可从搜索树中删去。但当前这个不可解节点还不能删去,因

为在判断其先辈节点的可解性时还要用到它。这是与或树搜索的两个特有性质，可用来提高搜索效率。

5.4.2 与或树的广度优先搜索

与或树的广度优先搜索同状态空间搜索中的广度优先搜索非常相似，搜索过程同样是按照“先扩展早产生的节点”进行的，唯一的区别在于其搜索过程中要多次调用可解标示过程和不可解标示过程。下面看一下广度优先搜索的具体步骤。

(1) 将初始节点 S_0 放入 OPEN 表中。

(2) 将 OPEN 表中的首节点(记作节点 n)取出并放进 CLOSED 表中。

(3) 若节点 n 可以扩展，那么需接着执行以下步骤：

① 将节点 n 进行扩展，把它的子节点放进 OPEN 表的尾部，同时为每个子节点匹配一个可能在标示过程中使用的指向父节点的指针。

② 判断这些子节点中是否有终止节点。如果有，就标示这些终止节点，将它们记作可解节点，并使用可解标示过程对其先辈节点(父节点、祖父节点等)中的可解节点进行标示。若初始节点 S_0 也被标示为可解节点，那么就说明解树已获得，说明搜索成功，即可退出搜索过程；反之，若不能确定可解节点是 S_0 的话，那么则需执行把 OPEN 表中具有可解先辈的节点删除的操作。

③ 执行步骤(3)第②步继续判断。

(4) 若确定节点 n 是不可扩展的，那么需要执行以下步骤：

① 标示节点 n 为不可扩展节点。

② 对节点 n 的先辈节点中的所有不可解节点使用不可解标示过程进行标示。若 S_0(初始节点)也被标示为不可解节点，那么说明搜索失败，即原始问题没有解，此时退出搜索即可；反之，若不能确定 S_0 是不可解节点，则需对 OPEN 表中的节点(即那些具有不可解先辈的节点)进行删除操作。

③ 执行步骤(4)第②步继续判断。

下面看一个与或树广度优先搜索的示例。

例 5.2 现有与或树如图 5-12 所示，树的各节点按图中标注的顺序从左至右扩展。(注：终止节点为标有 t_1、t_2、t_3、t_4 的节点。不可解的端节点为标有 A 和 B 节点。)

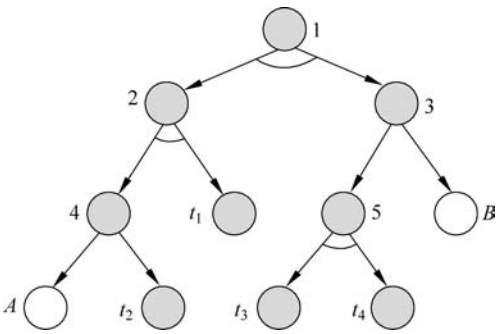

图 5-12 与或树的广度优先搜索示意

解：首先扩展 1 号节点，从而 2 号节点与 3 号节点也可相继得出。因为 2、3 号节点都不是终止节点，故我们继续扩展 2 号节点。可以看到直至现在，OPEN 表中仅剩 3 号节点。

当把 2 号节点进行扩展后，我们可以获得 t_1 节点与 4 号节点，这时 OPEN 表中的节点有 3 号、4 号与 t_1。由上述过程可知，我们可标示终止节点 t_1 为可解节点，同时也要对它的先辈节点中的可解节点使用可解标示过程进行标示。本例中，t_1 的父节点是一个“与”节点，所以只由 t_1 可解还无法确定 2 号节点是不是可解节点，还需进一步搜索。接下来对 3 号节点进行扩展。

当把 3 号节点进行扩展后，可以得到 5 号节点和 B 节点，由于它们都不是终止节点，所以我们继续对 4 号节点进行扩展。

当把 4 号节点进行扩展后，可以得到 A 节点和 t_2 节点。此时可以看到 t_2 是终止节点，因此标示其为可解节点，同时标示 4 号节点和 2 号节点均为可解节点(使用可解标示过程)。直至现在，我们仍不能确定 1 号节点是不是可解节点。目前，由于 OPEN 表中的第一个待考察的节点是 5 号节点，所以我们将对 5 号节点进行扩展。

当把 5 号节点进行扩展后，可以得到 t_3 和 t_4 节点。此时可以看到 t_3 和 t_4 两节点都是终止节点，因此标示这两个节点为可解节点，再使用可解标示过程就能得到 1、3 和 5 号节点都是可解节点的结论。

此时完成搜索过程，成功获得了由 1、2、3、4、5 号节点和 t_1、t_2、t_3、t_4、t_5 节点构成的解树。

5.4.3 与或树的深度优先搜索

与或树的深度优先搜索过程和与或树的广度优先搜索过程基本相同。只是要把步骤(3)的第①步改为“扩展节点 n，将其子节点放入 OPEN 表的首部，并为每个子节点配置指向父节点的指针，以备标示过程使用”，这样就可使后产生的节点先被扩展。

也可以像状态空间的有界深度优先搜索那样为与或树的深度优先搜索规定一个深度界限，使搜索在规定的范围内进行。它的搜索过程如下：

(1) 将初始节点 S_0 放进 OPEN 表中。

(2) 将 OPEN 表中的节点 n(首节点)取出并放进 CLOSED 表中。

(3) 若深度界限小于节点 n 的深度，那么跳转执行步骤(5)第①步。

(4) 若节点 n 确定是可扩展的，那么执行以下步骤：

① 对节点 n 进行扩展操作，把它的子节点放到 OPEN 表的最开始位置，同时为每个子节点匹配一个可能在标示过程中使用的指向父节点的指针。

② 判断这些子节点中是否有终止节点。如果有，就标示这些终止节点，将它们记作可解节点，并使用可解标示过程对其先辈节点(父节点、祖父节点等)中的可解节点进行标示。若初始节点 S_0 同样被标示成可解节点，那么就说明解树已获得，说明搜索成功，即可退出搜索过程。反之，若不能确定可解节点是 S_0，那么则需执行把 OPEN 表中具有可解先辈的节点删除的操作。

③ 执行步骤(4)第②步继续判断。

(5) 若节点 n 是不可扩展的，那么执行如下步骤：

① 标示节点 n 为不可解节点。

② 判断这些子节点中是否有终止节点。如果有，就标示这些终止节点，将它们记作可解节点，并使用可解标示过程对其先辈节点（父节点、祖父节点等）中的可解节点进行标示。若初始节点 S_0 也被标示为可解节点，那么就说明解树已获得，说明搜索成功，即可退出搜索过程。反之，若不能确定可解节点是 S_0，那么则需执行把 OPEN 表中具有可解先辈的节点删除的操作。

③ 执行步骤(5)第②步继续判断。

如果对如图 5-12 所示的与或树在限定深度界限为 4 时执行有界深度优先搜索，那么扩展节点的顺序就是 1、3、B、5、2、4。

5.5 基于树的启发式搜索

5.5.1 与或树的有序搜索

求取代价最小的解树有很多搜索方法，与或树的有序搜索就是其中之一，它是一种启发式搜索。要想得到代价最小的解树，就必须做到首先往前多看几步，然后再确定想要扩展的节点。与或树的有序搜索的最大特点就是根据代价来决定具体的搜索路线。在搜索过程中，首先计算扩展各个节点要付出的代价，然后选择出代价最小的节点后再进行下一步的扩展。

下面我们就与或树的有序搜索的概念和它的搜索过程两方面进行阐述。由前文可知，计算出解树的代价是进行有序搜索的前提。由于计算解树的代价可以利用计算解树中节点的代价来实现，因此我们的行文逻辑是先介绍计算节点代价的方法，再进一步讨论如何求解树的代价。

假设 $C(x,y)$ 表示节点 x 到它的子节点 y 的代价，那么计算节点 x 的代价的方法如下：

(1) 若 x 是终止节点，那么定义节点 x 的代价是 $h(x)=0$。

(2) 若 x 是“或”节点，$y_1,y_2,\cdots,y_n$ 是它的子节点，那么节点 x 的代价是

$$h(x)=\min\{C(x,y_i)+h(y_i)\}$$

(3) 若 x 是“与”节点，那么有两种计算节点 x 的代价的方法：和代价法与最大代价法。

按和代价法计算，那么可以得出

$$h(x)=\sum_{i=1}^{n}(C(x,y_i)+h(y_i))$$

按最大代价法计算，那么可以得出

$$h(x)=\max\{C(x,y_i)+h(y_i)\}$$

(4) 若节点 x 不可扩展，且又不是终止节点，那么定义 $h(x)=\infty$。

从上述计算节点代价的方法能够看出：若是可解问题，那么我们从子节点的代价就能够推算出父节点的代价；继续逐层上推，最终就能求出初始节点 S_0 的代价，即解树的代价。

例 5.3 图 5-13 是一棵包含两棵解树的与或树：节点 S_0、A、t_1 和 t_2 组成左解树；节点 S_0、B、D、G、t_4 和 t_5 组成另一棵解树。节点 t_1、t_2、t_3、t_4、t_5 是这棵与或树的终止节点，

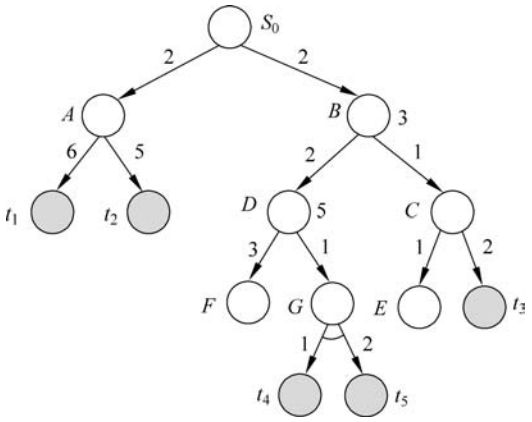

图 5-13　与或树的代价树示意

E、F 是端节点，两节点之间的数字为它们之间的代价，请确定初始节点 S_0 的代价。

解：从左侧解树入手。

由和代价法计算：$h(A)=11$，$h(S_0)=13$。

由最大代价法计算：$h(A)=6$，$h(S_0)=8$。

从右侧解树入手。

由和代价法计算：$h(G)=3$，$h(D)=4$，$h(B)=6$，$h(S_0)=8$。

由最大代价法计算：$h(G)=2$，$h(D)=3$，$h(B)=5$，$h(S_0)=7$。

由上面的计算结果可以看出：若依和代价法计算，右侧解树的代价最小是 8，那么右侧解树即为最优解树；若依最大代价法计算，此时右侧解树的代价是 7，故最优解树仍然是右侧解树。一般来说，使用不一样的计算代价的方法得到的最优解树也往往是不一样的。

不管是用和代价法求解，还是用最大代价法求解，求解已知子节点 y_i 的代价 $h(y_i)$ 均为计算其父节点 x 的代价 $h(x)$ 的前提。不过，由于搜索是先有父节点、后有子节点的自上而下的过程，因此，除非父节点 x 的所有子节点均为不可扩展节点，否则我们无法得知其子节点的代价。那么此时我们就要间接地求出节点 x 的代价 $h(x)$。首先根据问题本身提供的启发式信息定义一个启发式函数，再由这个启发式函数入手来估算出其子节点 y_i 的代价 $h(y_i)$，然后再利用前文所述的和代价法或最大代价法求得节点 x 的代价 $h(x)$，最后就可以自下而上地逐层推出节点 x 的父节点、祖父节点和初始节点 S_0 的各先辈节点的代价了。

在节点 y_i 被扩展后，也是先用启发式函数估算出其子节点的代价，然后再算出 $h(y_i)$。此时算出的 $h(y_i)$ 可能与原先估算出的 $h(y_i)$ 不相同。这时应该用后得到的 $h(y_i)$ 将原先估算出的 $h(y_i)$ 取代，与此同时按照相应的 $h(y_i)$ 自下而上地将各先辈节点的代价值进行重新计算。只要节点 y_i 的子节点被扩展，以上步骤就要重复进行一遍。简而言之，一旦有新节点生成时，都要自下而上地对它们的先辈节点的代价重新计算。这是一个循环迭代的过程，即自上而下地生成新节点，然后又自下而上地计算代价。

求得最优解树（代价最小的解树）是有序搜索的最终目的。为实现这个目的，我们要保证在搜索过程中的每一时刻都尽力使求出的部分解树的代价为当前最小代价。所以当选择想要扩展的节点时，我们都要先挑选出“最有希望”成为最优解树的节点，然后再进行扩展。因为这些节点和它们的先辈节点所组成的与或树很可能是最优解树的一部分，故我们称其

为“希望树”。

在搜索过程中，随着新节点的不断生成，节点的代价是在不断变化的，希望树也是在不断变化的。在某一时刻，这一部分节点构成希望树；但到另一时刻，可能是另一些节点构成希望树，随当时的情况而定。但不管如何变化，任一时刻的希望树都必须包含初始节点 S_0，而且它是对最优解树近根部分的某种估计。

希望树 T 的定义如下：

(1) 希望树 T 包含初始节点 S_0。

(2) 若希望树 T 包含节点 x，那么一定可以得出如下结论：

① 若 x 为具有子节点 $y_1, y_2, \cdots, y_n$ 的“或”节点，那么具有 $\min\{C(x, y_i)+h(y_i)\}$ 值的那个子节点 y_i 也是希望树的一部分。

② 若 x 为“与”节点，那么它的所有子节点均为希望树的一部分。

与或树的有序搜索是一种在选择希望树的同时不断修复希望树的迭代搜索方法。若问题存在最终解，那么经过有序搜索我们一定能得出最优解树，下面将具体讨论其搜索过程。

(1) 将初始节点 S_0 放进 OPEN 表里。

(2) 根据当前搜索树中节点的代价求出以 S_0 作为根节点的希望树 T。

(3) 依次选出 OPEN 表中希望树的端节点 N，然后把它放进 CLOSED 表里。

(4) 若确定 N 为终止节点，那么执行以下步骤：

① 将节点 N 表示成可解节点。

② 对希望树 T 执行可解标示过程，即标示节点 N 对应的先辈节点中的所有可解节点都为可解节点。

③ 如果初始节点 S_0 能被标示成可解节点，那么就说明 T 一定为最优解树，搜索成功。

④ 如果初始节点未能被标示成可解节点，则将 OPEN 表里具有可解先辈的全部节点删去。

(5) 若节点 N 并非是终止节点，同时它也不能扩展，那么执行以下步骤：

① 将节点 N 标示成不可解节点。

② 对希望树 T 执行不可解标示过程，即标示节点 N 对应的先辈节点中的所有不可解节点为不可解节点。

③ 如果初始节点 S_0 同样被标示成不可解节点，即说明搜索失败。

④ 如果初始节点没有被标示成不可解节点，那么把 OPEN 表中所有具备不可解先辈的节点删除。

(6) 若节点 N 可扩展且它不是终止节点，那么执行以下步骤：

① 对节点 N 进行拓展，得到节点 N 的全部子节点。

② 将上一步得到的子节点放进 OPEN 表里，同时给每个子节点匹配一个可能在标示过程中使用的指向父节点的指针。

③ 算出以上子节点与其对应的所有先辈节点的代价。

(7) 执行步骤(2)。

例 5.4 如图 5-14 所示是初始节点 S_0 经扩展两层后得到的与或树。拓展规则是每次扩展“与”节点和“或”节点各一层。假设每个节点到相应子节点的代价是 1，请求出该树的最优解树。

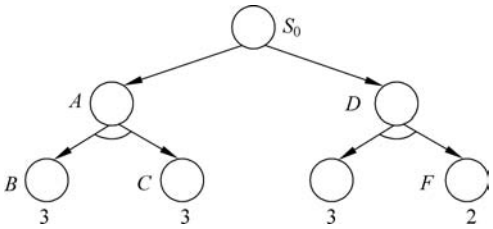

图 5-14　扩展两层后的与或树示意

解：由图 5-14 可知，在用启发式函数进行估算时，已知节点 B 的代价是 $h(B)=3$、节点 C 的代价是 $h(C)=3$、节点 E 的代价是 $h(E)=3$、节点 F 的代价是 $h(F)=2$，如果按照和代价法计算，那么可以有

$$h(A)=8,\quad h(D)=7,\quad h(S_0)=8$$

显然，这时希望树就为 S_0 的右子树，接下来扩展这棵希望树的端节点。

图 5-15 所示是对节点 E 扩展后得出的与或树。使用启发式函数估算出节点的代价如节点最下方的数字所示。

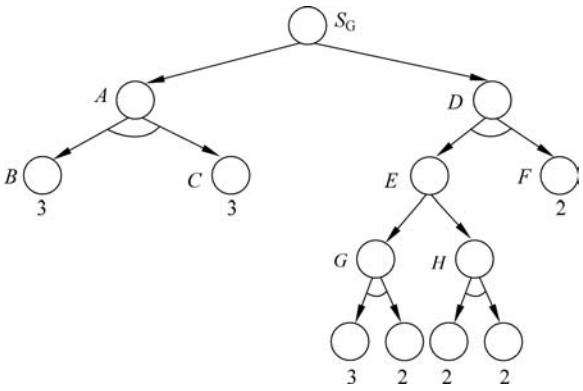

图 5-15　扩展节点 *E* 后的与或树示意

如果按照和代价法计算的话，那么可以有

$$h(G)=7,\quad h(H)=6,\quad h(E)=7,\quad h(D)=11$$

这种情况下，很容易就能在 S_0 的左子树中求出 $h(S_0)=9$，从 S_0 的右子树中求出 $h(S_0)=12$。可以看出，左子树的代价是小于右子树的，因此将此时的希望树替换为左子树。

图 5-16 所示是对节点 B 与节点 C 扩展后得出的与或树。使用启发式函数估算出节点的代价如节点最下方的数字所示。

由图 5-16 可知，终止节点是拓展节点 L 的两个子节点，由代价和法可得

$$h(L)=2,\quad h(M)=6,\quad h(B)=3,\quad h(A)=8$$

从左子树能够得出 $h(S_0)=9$。除此之外，因为节点 L 的左右子节点均为终止节点，故节点 L 与节点 B 均为可解节点。由于目前仍无法确定节点 C 是否为可解节点，因此节点 A 与节点 S 同样无法被确定是否为可解节点。

对节点 C 进行扩展后得到的与或树仍为图 5-16，由和代价法可得

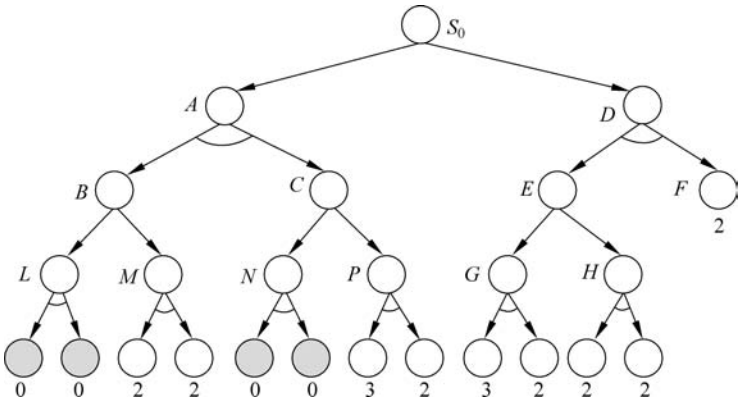

图 5-16　扩展 *C* 后的与或树示意图

$$h(N)=2,\quad h(P)=7,\quad h(C)=3,\quad h(A)=8$$

从左子树能够得出 $h(S_0)=9$。除此之外，因为节点 N 的左右子节点均为终止节点，故节点 N 与节点 C 均为可解节点。又因为节点 B 是可解节点，那么就能推出节点 A 与 S_0 节点均为可解节点。至此，我们就用和代价法求出了本题的最优解树，即最小代价为 9 的解树如图 5-16 所示。

5.5.2　博弈树搜索

博弈是人们生活中常见的一种活动。例如下棋、打牌等活动均是博弈活动。博弈活动中隐藏着深刻的优化理论。博弈活动中一般有对立的几个方面，每一方都试图使自己的利益最大化。博弈活动的整个过程其实就是一个动态的搜索过程。

不妨假设 A 和 B 正在比赛象棋。从规则上容易得出：

（1）比赛采取轮流制。

（2）比赛的结果只有 3 种：A 胜、B 胜和双方打和。

（3）对战双方了解当前形势和历史信息。

（4）对战双方都是绝对理性的，都选取对自己最为有利的对策。

博弈活动中对战的双方都希望自己能获得胜利。对于对战的任一方，比如我们站在 A 方的立场，当比赛轮到 A 方落子时，A 方可以有多种落子方案。具体落哪个子，完全由 A 自己决定，这可以看作是与或树中的“或”关系。为了获得胜利，A 总是会选择对自己最为有利的落子方案，这就相当于 A 在一棵或树中选择了最优路径。如果比赛轮到了 B 方落子，那么对于 A 来说，就必须考虑 B 所有可能的落子方案，这就相当于与或树中的“与”关系。因为主动权掌握在 B 手中，所以任何落子方案都是有可能的。

把上述博弈过程用图表示出来，就可得到一棵与或树。这里要强调的是，该与或树始终站在某一方（例如 A 方）的立场上，绝不可一会儿站在这一方的立场上，一会儿又站在另一方的立场上。

把描述博弈过程的与或树称为博弈树，它有如下特点：

（1）博弈的初始格局是初始节点。

(2) 在博弈树中,或节点和与节点是逐层交替出现的。自己一方扩展的节点之间是或关系,对方扩展的节点之间是与关系。双方轮流地扩展节点。

(3) 所有能使自己一方获胜的终局都是本原问题,相应的节点是可解节点;所有能使对方获胜的终局都是不可解节点。

在博弈问题中,为了从众多可供选择的方案中选出一个对自己有利的行动方案,就要对当前情况以及将要发生的情况进行分析,从中选出最优方案。最常用的分析方法是极大极小分析法。其基本思想如下:

(1) 目的是为博弈双方中的一方寻找一个最优行动方案。

(2) 要寻找这个最优方案,就要通过计算当前所有可能的方案来进行比较。

(3) 方案的比较是根据问题的特性定义一个估价函数,用来估算当前博弈树端节点的得分。此时估算出来的得分称为静态估值。

(4) 当计算出端节点的估值后,再推算出父节点的得分。推算的方法是:对"或"节点,选其子节点中的一个最大的得分作为父节点的得分,这是为了使自己能在可供选择的方案中选出一个对自己最有利的方案;对"与"节点,选其子节点中的一个最小的得分作为父节点的得分,这是为了考虑最坏的情况。这样计算出的父节点的得分称为倒推值。

(5) 如果一个行动方案能获得较大的倒推值,则它就是当前最好的行动方案。

图 5-17 给出了计算倒推值的示例。

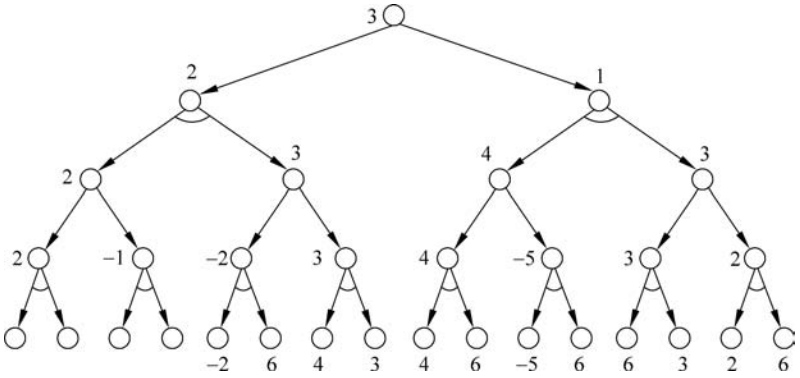

图 5-17　计算倒推值示例

在博弈问题中,每一个格局可供选择的行动方案都有很多,这会生成十分庞大的博弈树。

据统计,西洋跳棋完整的博弈树约有 10^{40} 个节点。试图利用完整的博弈树来进行极大极小分析是困难的。可行的办法是只生成一定深度的博弈树,然后进行极大极小分析,找出当前最好的行动方案。在此之后,还可在已经选定的分支上再扩展一定深度,选出最好的行动方案。如此进行下去,直到取得胜负为止。至于每次生成博弈树的深度,当然是越大越好,但由于受到计算机存储空间的限制,深度只能根据实际情况而定。

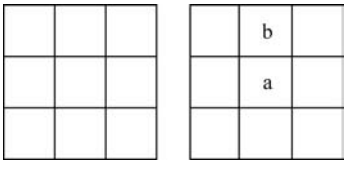

图 5-18　一字棋

例 5.5　一字棋游戏。设有如图 5-18 所示的 9 个空格。有 A、B 二人对弈,轮到谁走棋谁就往空格上放自己的一只棋子。谁先使自己的 3 个棋子串成一条直线,谁就取

得了胜利。

解：设 A 的棋子用“a”表示，B 的棋子用“b”表示。为了不至于生成太大的博弈树，假设每次仅扩展两层。设棋局为 p，估价函数为 $e(p)$，且满足如下条件：

（1）若 p 为 A 必胜的棋局，则 $e(p)=+\infty$。

（2）若 p 为 B 必胜的棋局，则 $e(p)=-\infty$。

（3）若 p 为胜负未定的棋局，则 $e(p)=e(+p)-(-p)$。

其中，$e(+p)$表示棋局 p 上有可能使 a 成为 3 子一线的方案数目；$e(-p)$表示棋局 p 上有可能使 b 成为 3 子一线的方案数目。例如，如图 5-18 所示的棋局，则 $e(p)=6-4=2$。另外还需假定具有对称性的两个棋局算作一个棋局，并且假定 A 先走棋，我们站在 A 的立场上。

图 5-19 给出了 A 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 5-19 可以看出，对于 A 来说最好的一着棋是 S_3，因为 S_3 比 S_1 和 S_2 有更大的倒推值。

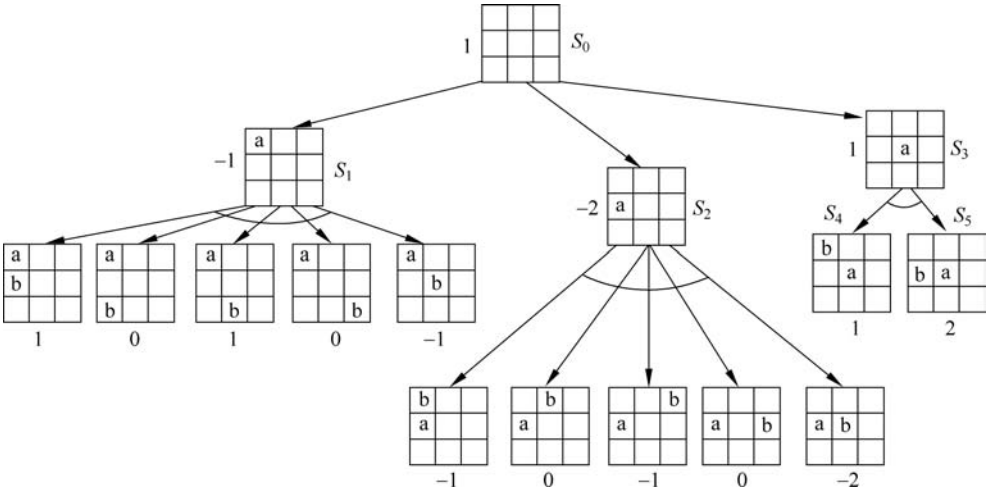

图 5-19　一字棋的极大极小搜索示意

在 A 走 S_3 这一着棋后，B 的最优选择是 S_4。因为这一着棋的静态估值较小，对 A 不利。

不管 B 选择 S_4 还是选择 S_5，A 都要再次运用极大极小分析法产生深度为 2 的博弈树，以决定下一步应该如何走棋。其过程与上面类似，这里不再重复。

5.5.3　博弈树的剪枝优化

前面讨论的极大极小分析过程先得到一棵博弈树，然后再进行估值的倒推计算。两个过程完全分离，效率很低。鉴于博弈树具有“与”节点和“或”节点逐层交替出现的特点，如果可以边生成节点边计算估值和倒推值，就可以删除一些不必要的节点以提高效率。这就是下面要讨论的 α-β 剪枝技术。

各端节点的估值如图 5-20 中的博弈树所示，其中对于 G 尚未计算估值。由 D 与 E 的

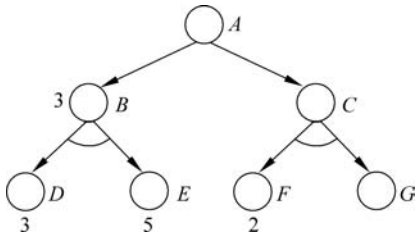

图 5-20　α-β 剪枝技术示意

估值得到 B 的倒推值为 3，这表示 A 的倒推值最小为 3。另外，由 F 的估值得知 C 的倒推值最大为 2，因此 A 的倒推值为 3。这里虽然没有计算 G 的估值，但是仍然不影响对上层节点倒推值的推算，这表示这个分枝可以从博弈树中剪去。

对于一个“与”节点来说，它取当前子节点中的最小倒推值作为它倒推值的上界，称此值为 β 值。对于一个“或”节点来说，它取当前子节点中的最大倒推值作为它倒推值的下界，称此值为 α 值。

下面给出 α-β 剪枝技术的一般规律。

(1) 任何“或”节点 x 的 α 值如果不能降低其父节点的 β 值，则对节点 x 以下的分枝可停止搜索，并使 x 的倒推值为 α。这种剪枝技术称为 β 剪枝。

(2) 任何“与”节点 x 的 β 值如果不能增大其父节点的 α 值，则对节点 x 以下的分枝可停止搜索，并使 x 的倒推值为 β。这种剪枝技术称为 α 剪枝。

在 α-β 剪枝技术中，一个节点的第一个子节点的倒推值(或估值)是很重要的。对于一个“或”节点来说，估值最高的子节点最先生成；对于一个“与”节点来说，估值最低的子节点最先生成。被剪除的节点数越多，搜索的效率越高。这称为最优 α-β 剪枝技术。

5.6　案例：无人驾驶中的搜索策略

本节以无人驾驶为背景，介绍 A* 算法在其中的作用。A* 算法是一种在路网上求解最短路径的直接搜索算法，原理是引入估价函数，加快搜索速度，提高局部择优算法搜索的精度，已成为当前较为流行的最短路径算法。车辆路径规划寻路算法有很多，百度 Apollo 的路径规划模块使用的是启发式搜索算法 A* 进行路径的查找与处理。以图 5-21 的网格图为例，将该网格图中的每个单元格当作一个节点，就能从任何一个节点移动到其任意相邻节点。这个特殊网格图包含一些阻挡潜在路径的“墙壁”。

估价函数用公式表示为

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

其中，$f(n)$是从初始节点到目标节点的最短路径的估计代价，$g(n)$是从初始节点到节点 n 的代价，$h(n)$是从节点 n 到目标节点的估计代价。

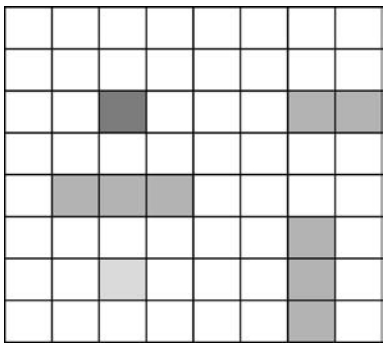

图 5-21　A* 算法的网格搜索示意

要保证找到最短路径(最优解)，关键在于估价函数 $f(n)$的选取(或者说 $h(n)$的选取)。很显然，距离估计与实际值越接近，估价函数取得就越好。例如，对于路网来说，可以取两节点间的曼哈顿距离作为距离估计，即 $f(n)=g(n)+(|dx-nx|+|dy-ny|)$。这样估价函数 $f(n)$在 $g(n)$一定的情况下，会或多或少地受距离估计值 $h(n)$的制约，节点距目标节点越近，h 值越

小，f 值就越小，就能保证最短路径的搜索向目标节点方向进行。

A* 算法保持着两个表，即 OPEN 表和 CLOSED 表。OPEN 表由未考察的节点组成，而 CLOSED 表由已考察的节点组成。当算法已经检查过与某个节点相连的所有节点，计算出它们的 f、g 和 h 值，并把它们放入 OPEN 表中已待考察，则称这个节点为已考察的。

算法过程如下：

（1）令 s 为初始节点。

（2）计算 s 的 f、g 和 h 值。

（3）将 s 加入 OPEN 表，此时 s 是 OPEN 表里唯一的节点。

（4）令 b 为 OPEN 表中的最佳节点（最佳的意思是该节点的 f 值最小）。如果 b 是目标节点，则退出，此时已找到一条路径；如果 OPEN 表为空，则退出，此时没有找到路径。

（5）令 c 等于一个与 b 相连的有效节点，计算 c 的 f、g 和 h 值。检查 c 是在 OPEN 表里还是在 CLOSED 表里。若在 CLOSED 表里，则检查新路径是否比原先更好（f 更小）。若是，则采用新路径，否则把 c 放入 OPEN 表。对所有 b 的有效子节点重复步骤（5）。

（6）重复步骤（4）。

对于每个候选节点，我们添加 g 值和 h 值来计算总和，即 f 值。最佳候选节点是 f 值最小的节点，每当抵达新节点时，通过重复此过程来选择下一个候选节点。总是选择尚未访问过且具有最小 f 值的节点，这就是 A* 算法，它能建立一条稳定前往目标节点的路径。关于 A* 算法在 Apollo 规划模块的实际应用案例，读者们可以登录 Apollo 官方网站查阅更详细的内容。

Python 的 networkx 包支持了 A* 算法，可自定义距离估计函数。如代码清单 5-1 所示。

代码清单 5-1　使用 networkx 包的 A* 算法

```
>>> import networkx as nx
# 一维路径
>>> G = nx.path_graph(5)
>>> print(nx.astar_path(G,0,4))
[0, 1, 2, 3, 4]

# 二维图
>>> G = nx.grid_graph(dim = [3,3]) # nodes are two - tuples (x,y)
>>> def dist(a, b): # 定义距离估计函数
...     (x1, y1) = a
...     (x2, y2) = b
...     return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
>>> print(nx.astar_path(G,(0,0),(2,2),dist))
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)]
```

代码清单 5-2 展示了如何使用 Python 从头实现一个 A* 算法。

代码清单 5-2　基于 Python 实现的 A* 算法

```
class AStar:
    """A* (A - Star) 算法主类"""
```

```
    def __init__(self, map):
        self.map = map
        self.open_set = []
        self.close_set = []

    def base_cost(self, p):
        """节点到起点的移动代价,对应了 g(n)"""
        x_dis = p.x
        y_dis = p.y
        # 到起点的距离
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis, y_dis)

    def heuristic_cost(self, p):
        """节点到终点的启发函数,对应 h(n)
        由于是基于网格的图形,所以本函数和 base_cost 函数均使用的是对角距离"""
        x_dis = self.map.size - 1 - p.x
        y_dis = self.map.size - 1 - p.y
        # 到终点的距离
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis, y_dis)

    def total_cost(self, p):
        """代价总和,即对应了 f(n)"""
        return self.base_cost(p) + self.heuristic_cost(p)

    def is_valid_point(self, x, y):
        """ 判断点是否有效,不在地图内部或者障碍物所在点都是无效的"""
        if x < 0 or y < 0:
            return False
        if x >= self.map.size or y >= self.map.size:
            return False
        return not self.map.is_obstacle(x, y)

    def process_point(self, x, y, parent):
        """针对每一个节点进行处理:
        如果是没有处理过的节点,则计算优先级设置父节点,并且添加到 open_set 中"""
        if not self.is_valid_point(x, y):
            return  # 无效的点直接返回
        p = point.Point(x, y)
        if self.is_in_close_list(p):
            return  # 在 close_set 中的点直接返回
        print('Process Point [', p.x, ',', p.y, ']', ', cost: ', p.cost)
        if not self.is_in_open_list(p):
            p.parent = parent
            p.cost = self.total_cost(p)
            self.open_set.append(p)

    def select_point_in_open_list(self):
        """从 open_set 中找到优先级最高的节点,返回其索引"""
```

```
            index = 0
            selected_index = -1
            min_cost = sys.maxsize
            for p in self.open_set:
                cost = self.total_cost(p)
                if cost < min_cost:
                    min_cost = cost
                    selected_index = index
                index += 1
            return selected_index

        def build_path(self, p, ax, plt, start_time):
            """从终点往回沿着 parent 构造结果路径
            然后从起点开始绘制结果,结果使用绿色方块,每次绘制一步便保存一张图片"""
            path = []
            while True:
                path.insert(0, p)  # 先插入
                if self.is_start_point(p):
                    break
                else:
                    p = p.parent
            for p in path:
                rec = Rectangle((p.x, p.y), 1, 1, color='g')
                ax.add_patch(rec)
                plt.draw()
                self.save_image(plt)
            end_time = time.time()
            print('算法运行时间为', int(
                end_time - start_time), '(秒)')

        def run_and_save_image(self, ax, plt):
            """A* 算法的主逻辑"""
            start_time = time.time()

            start_point = point.Point(0, 0)
            start_point.cost = 0
            self.open_set.append(start_point)

            while True:
                index = self.select_point_in_open_list()
                if index < 0:
                    print('没有找到任何一条可行的路径')
                    return
                p = self.open_set[index]
                rec = Rectangle((p.x, p.y), 1, 1, color='c')
                ax.add_patch(rec)
                self.save_image(plt)

                if self.is_end_point(p):
```

```
                return self.build_path(p, ax, plt, start_time)

            del self.open_set[index]
            self.close_set.append(p)

            # 处理所有的邻居节点
            x = p.x
            y = p.y
            self.process_point(x - 1, y + 1, p)
            self.process_point(x - 1, y, p)
            self.process_point(x - 1, y - 1, p)
            self.process_point(x, y - 1, p)
            self.process_point(x + 1, y - 1, p)
            self.process_point(x + 1, y, p)
            self.process_point(x + 1, y + 1, p)
            self.process_point(x, y + 1, p)
```

习题

一、选择题

1. 以下(　　)不是启发式搜索算法。

A. A* 算法　　B. 模拟退火算法

C. 深度优先搜索　　D. 爬山法

2. 以下(　　)是广度优先算法的优点。

A. 总能找到路径最短的解(若有)　　B. 占用资源少

C. 搜索效率高　　D. 针对性强

3. 一般情况下,基于状态空间的深度优先搜索中,OPEN 表采用(　　)数据结构。

A. 队列　　B. 栈　　C. 堆　　D. 图

4. 以下(　　)是深度优先搜索的缺点。

A. 实现代码较复杂　　B. 执行过程中会产生大量无用节点

C. 不保证能求得解　　D. 迭代速度慢

5. 在与或树的一般性搜索中,若问题较复杂,通常采取(　　)思想来处理。

A. 动态规划　　B. 贪心

C. 回溯　　D. 分治

6. 爬山法的主要问题是(　　)。

A. 容易陷在“局部极大值”上　　B. 系统资源开销大

C. 搜索效率低　　D. 盲目性强

7. 在一个无人驾驶路径搜索项目中,要求能得到稳定的前往目的地的路径,并要求其效率尽可能高,应该选用以下(　　)算法。

A. 爬山法　　B. 模拟退火法

C. A* 算法　　D. 广度优先搜索

8. 以下（　　）不是模拟退火算法的基本组成部分。

A. 解空间　　B. 冷却进度表　　C. 目标函数　　D. 初始解

9. 关于希望树，以下说法不正确的是（　　）。

A. 任何一时刻的希望树一定包含初始节点 S_0

B. 希望树有可能成为最优解树的一部分

C. 随着新节点的生成，希望树会被不断修改、迭代

D. 希望树一般不会被拓展

10. 搜索过程中，以下（　　）一般会被放在最后考虑。

A. 代码复杂度与长度

B. 时间复杂度与空间复杂度的平衡

C. 好的解决方案可能彼此接近，也可能位于状态空间的位置探索区域

D. 面对复杂问题时，如何将其抽象成算法

二、判断题

1. 搜索策略一般分为盲目式搜索和启发式搜索。（　　）

2. 启发式搜索采用问题自身的特性信息，以指导搜索朝着最有希望的方向前进。（　　）

3. 模拟退火法保存且重用问题求解中已搜索、已求解的子问题轨迹。（　　）

4. 启发式搜索的效率一定比盲目式搜索高。（　　）

5. Samuel 提出了爬山法的优化算法，解决了“局部极大值”问题。（　　）

6. 模拟退火法已在理论上被证明，将以概率 1 收敛于全局最优解。（　　）

7. 回溯机制可以使算法辨别局部和全局最大值。（　　）

8. A^* 算法不具有可纳性。（　　）

9. 复杂问题分解得到“与”树，等价变化得到“或”树。（　　）

10. 如果一个问题存在解决算法，则一定可以搜索得到它的解。（　　）

三、问答题

1. 什么是搜索？有哪两大类不同的搜索方法？二者的区别是什么？

2. 深度优先搜索与广度优先搜索的区别是什么？

3. 在什么情况下深度优先搜索优于广度优先搜索，为什么？请加以说明。

4. 局部择优搜索与全局择优搜索的相同之处是什么？区别是什么？

5. 设有图 5-22 所示的与或树，请分别按和代价法及最大代价法求解树的代价。

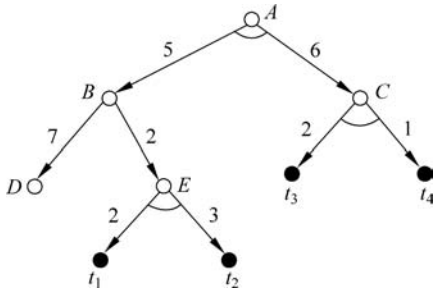

图 5-22　与或树

6. 有一农夫带一只狼、一只羊和一筐菜从河的左岸乘船到右岸，但受下列条件限制：

(1) 船太小，农夫每次只能带一样东西过河；

(2) 如果没有农夫看管，则狼要吃羊、羊要吃菜。

请设计一个过河方案，使得农夫、狼和羊都能不受损失地过河。

提示：

(1) 用四元组(农夫，狼，羊，菜)表示状态，其中每个元素都为 0 或 1，用 0 表示在左岸，用 1 表示在右岸；

(2) 把每次过河的一种方案作为一种操作，每次过河都必须有农夫，因为只有他可以划船。