第3章

图搜索算法

搜索算法的搜索路径是用称为图的数据结构保存的。图由节点和边构成，节点代表搜索过程中的某一个状态，边是从一个节点过渡到另一个节点的算子。前面讨论了用搜索求解问题的方法是在一个图(一般是隐式图)中进行搜索，而搜索的策略是在图中寻找最佳解路径。下面首先讨论图在搜索过程中是以怎样的形式存储的。

抽象地来看，搜索图可以有两种结构。

(1) 树结构，允许搜索图中有相同状态多次出现，每出现一次就用一个新的节点表示，这样的图没有回路出现，即没有圈。

(2) 图结构，不允许搜索图中有相同状态多次出现，即相同的状态只能用一个节点表示。这样，如果有两条路径都生成了某个状态，这个状态只能用一个节点表示，这就在图中形成了圈。

本章讨论用图来存储问题的搜索空间，而保存解路径则用树结构。根据图对应的实际问题背景可分为“或图”和“与/或图”两种。或图对应的背景为搜索扩展时，可在若干分支中选择一个(这就是“或”的意思)；与/或图则是在搜索扩展时，有可能同时搜索若干分支(这就是“与”的含义)，也有可能在若干分支选择其中之一(这就是“或”的含义)。这两种图都可能存在回路，以下讨论的算法中，都通过一定的手段避开在回路中循环搜索。

3.1 或图搜索

或图对应的背景为搜索扩展时，可在若干分支中选择一个。例如编写一个下棋程序，其搜索过程的每一步就是在若干下棋的走步中选择一个，这就是典型的或图中的搜索。

下面首先讨论一般或图上的搜索算法，然后讨论建立在这个一般算法之上的若干更高效的搜索算法。

3.1.1 或图搜索算法

图搜索算法只记录状态空间那些被搜索过的状态，它们组成一个搜索图 G。G 由两张存放节点的表(List)，再加上一个反向树组成。

(1) Open 表。用于存放已经生成,且已用启发式函数作过估计或评价,但尚未产生它们的后继节点的那些节点,也称未考察节点。

(2) Closed 表。用于存放已经生成,且已考察过的节点。

(3) 反向边组成的 Tree。用来存放当前已生成的搜索树,该树也是一个表,该表的元素是搜索过程中的反向边(反向指针)。

例 3.1 在如图 3.1 所示的搜索图中,假如由节点 A 搜索到节点 B,再搜索到节点 E,而且已扩展了它的两个后继节点。此时用上述方法表示的隐式图 G 为:

```
Open = (D,H,I)          //还未考察节点
Closed = (A,B,E)        //已考察过的节点
Tree = ((E,B),(B,A))    //反向边组成的表
```

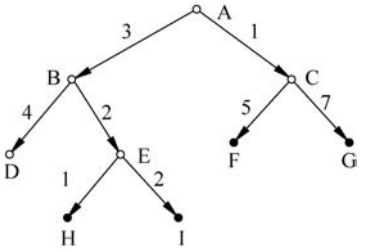

图 3.1 一个简单的搜索图

图 3.1 中,假如 I 是目标节点,搜索到 I 节点后,搜索成功,(I,E)加入到了 Tree。这时,根据这个 Tree,做反向查找,很快就可以找到解路径是(A,B)(B,E)(E,I)。

下面讨论或图通用搜索算法,通用的含义是,只要搜索问题能表示成这样一个或图,这个算法就可以用于求解这个问题,即算法具有通用性。

1. 或图通用搜索算法

设 S_0 为初始状态,S_g 为目标状态。

(1) 产生仅由 S_0 组成的 Open 表,即 Open=(S_0)。

(2) 产生一个空的 Closed 表。

(3) 如果 Open 为空,则失败退出。

(4) 在 Open 表上按某一原则选出一个节点,称为 n,将 n 放到 Closed 表中,并从 Open 表中去掉 n。

(5) 若 $n \in S_g$,则成功退出,此时,解为在 Tree 中沿指针从 n 到 S_0 的路径,或 n 本身。(例如八皇后问题给出到达的目标状态 n 即可,八数码问题要给出到达目标状态的路径)。

(6) 产生 n 的一切后继,将后继中不是 n 的先辈点的一切点构成集合 M,装入 G 作为 n 的后继。(这就剔除了既是 n 的先辈又是 n 的后继的结点,从而避免了回路。)

(7) 对 M 中的元素 P,分别作两类处理:

① 若 P∉G,即 P 不在 Open 表中也不在 Closed 表中,则 P 根据一定原则加入 Open 表,同时对 P 进行评价,把指向 P 的边反向后,加入 Tree 中。

② 若 P∈G,则决定是否更改 Tree 中 P 到 n 的指针。

(8) 转 3。

2. 算法说明

以上算法中有两点需要说明。

说明一：在算法的第(4)步，每次取出 Open 表的第一个节点，然后在第(7)步的①中，若生成的后继节点放于：

(1) Open 表的尾部，算法相当于广度优先(Breadth-first-search)。

(2) Open 表的首部，算法相当于深度优先(Depth-first-search)。

(3) 根据启发式函数 f 的估计值确定最佳者，放于 Open 表的首部，算法相当于最佳优先(Best-first-search)。

例 3.2 我们不妨对上述第(1)种情况进行算法步骤的跟踪，还是假定目标是图 3.1 中的节点 G。

(1) 产生仅由 S_0 组成的 Open 表，即 Open=(A)。

(2) 产生一个空的 Closed 表。

(3) Open=(A)，不为空。

(4) 从 Open 表中取出第一个节点 A，称为 n，将 n 放到 Closed 表中，并从 Open 表中去掉 A，此时 Open=( )，Closed=(A)。

(5) n=A，不是目标。

(6) 产生 A 的后继 B 和 C，此时 M=(B,C)。

(7) 考察 M 中的元素 B、C，它们不在 Open 表中也不在 Closed 表中，加入 Open 表，Open=(B,C)，再将反向边(B,A)、(C,A)加入 Tree 中。

(8) 转(3)。

(9) Open=(B,C)，不为空。

(10) 从 Open 表中取出第一个结点 B，放到 Closed 表中，并从 Open 表中去掉 B，此时 Open=(C)，Closed=(A,B)。

(11) n=B，不是目标。

(12) 产生 B 的后继 D、E，此时 M=(D,E)。

(13) 考察 M 中的元素 D、E，它们不在 Open 表中也不在 Closed 表中，加入 Open 表，Open=(C,D,E)，反向边(D,B)、(E,B)加入 Tree 中。

(14) 转(3)。

(15) Open=(C,D,E)，不为空。

(16) 从 Open 表中取出第一个节点 C，放到 Closed 表中，并从 Open 表中去掉 C，此时 Open=(D,E)，Closed=(A,B,C)。

(17) n=C，不是目标。

(18) 产生 C 的后继 F、G，此时 M=(F,G)。

(19) 考察 M 中的元素 F、G，它们不在 Open 表中也不在 Closed 表中，加入 Open 表，Open=(D,E,F,G)，反向边(F,C)、(G,C)加入 Tree 中。

(20) 转(3)。

继续跟踪下去，不难看出，按照这种策略，访问的次序是 A、B、C、D、E、F、G。这正是广度优先搜索的次序。

若生成的后继节点放于 Open 表的首部，读者用上述同样的步骤进行跟踪，不难看出，访问的次序是 A、B、D、E、H、I、C、F、G。这正是深度优先搜索的次序。

若将启发式函数 f 的估计值的最佳者放于 Open 表的首部，再对或图通用的搜索算法进行跟踪。对于图 3.1，若将达到某个节点的边的权值作为这个节点启发式函数 f 的值，即 $f(B)=3$，$f(C)=1$，$f(D)=4$，$f(E)=5$，$f(F)=3$，$f(G)=7$，$f(H)=3$，$f(I)=2$，在这个假定下，搜索算法访问的次序分别是 A、C、B、E、H、I、D、F、G。读者不妨跟踪算法，记录一下 Open 表中节点的编号变化的情况。

说明二：在算法的第②步中，若 p∈G，则决定是否更改 Tree 中 p 到 n 的指针。这是因为 p 已经在 G 中，说明以前访问过 p 节点，那么原来的访问路径与当前的访问路径就要进行比较，看哪一条路径更好。

例如，如图 3.2 所示，p∈M 且在 Open 表中，这说明 p 在作为 n 的后继之前已是某一节点 m 的后继，但本身尚未被考察（未生成 p 的后继）。

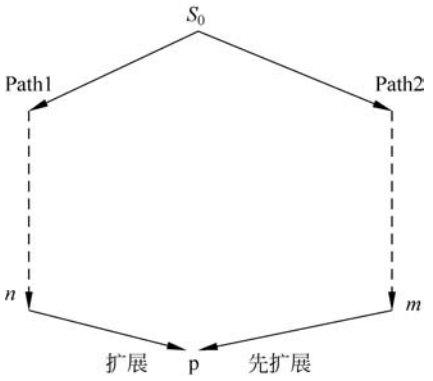

图 3.2　p 在 n 之前已是某一节点 m 的后继

这说明 $S_0 \rightarrow$p 至少有两条路径，这时有两种情况：

- 若 Path1 的代价＜Path2 的代价时，当前路径较好，要修改 p 的指针，使其指向 n，即标出搜索之后的最好路径；
- 若 Path1 的代价≥Path2 的代价时，原路径较好，不改变 p 的指针。

或图通用搜索算法第②步中是否需要更改 Tree 中 p 到 n 的指针，还有更复杂的情况，在这里不再详细讨论。

3.1.2　A 算法与 A* 算法

1. A 算法与 A* 算法定义

在或图通用搜索算法第④步“在 Open 表上按某一原则选出一个节点”定义为：按启发式函数 f 的值从小到大排列，然后取出第一个节点，并将启发式函数的形式定义为：$f(n)=g(n)+h(n)$。具有这样的启发式函数的或图通用搜索算法就称为 A 算法。

A 算法的启发式函数中，$g(n)$表示从 S_0 到 n 点的搜索费用的估计，因为 n 为当前节点，搜索已达到 n 点，所以可计算出 $g(n)$。$h(n)$表示从 n 到 S_g 接近程度的估计，因为尚未找到解路径，所以 $h(n)$仅仅是估计值。

在 A 算法中，若令 $h(n)\equiv 0$，则 A 算法相当于广度优先，因为上一层节点的搜索费用一般比下一层的小，所以优先搜索本层节点(向广度发展)。

$g(n)\equiv h(n)\equiv 0$，则相当于随机算法。

$g(n)\equiv 0$，则相当于最佳优先算法。

在 A 算法中若进一步规定 $h(n)\geqslant 0$，并且定义：

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

其中，$f^*(n)$表示 S_0 经点 n 到 S_g 最优路径的搜索费用，也有人将 $f^*(n)$定义为实际最小搜索费用；$g^*(n)$为 S_0 到 n 的实际最小费用；$h^*(n)$为 n 到 S_g 的实际最小费用的估计。

特别是要求 $h(n)\leqslant h^*(n)$，就称这种 A 算法为 **A^* 算法**。

例 3.3　在图 3.3 中寻找城市 A 到城市 B 的最短路径，实线表示从 S_0 到某节点 n_i 所经过的路径，虚线表示 n_i 与 S_g 的可选择路径，双虚线表示 n_i 与 S_g 的直线距离(可以从地图上量出)，但并不一定有实际的道路，则实线表示的路径为 $g(n)$，双虚线和虚线表示的路径都可作为 $h(n)$。以 n_3 为例，$g(n)=\{S_0\rightarrow n_1\rightarrow n_3\}$，$h(n)$可以是$\{n_3\rightarrow n_4\rightarrow S_g\}$、$\{n_3\rightarrow n_4\rightarrow n_5\rightarrow S_g\}$或$\{n_3\rightarrow S_g$(双虚线)$\}$这些路径中的一个，即 $h(n)$是 n_3 到 S_g 的各种可能的路径。

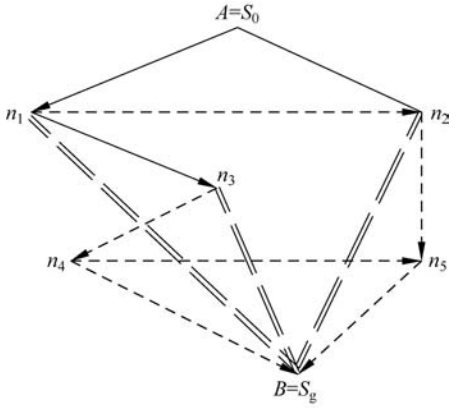

图 3.3　在地图上寻找城市 A 到城市 B 的最短路径

在本例中，如果将双虚线表示的路径定义为 $h(n)$，显然有 $h(n)\leqslant h^*(n)$，$g(n)\geqslant g^*(n)$。所以，本例使用这样的 $h(n)$(双虚线)作为搜索算法的估计函数，这个算法就是 A^* 算法。

不难看出，A^* 算法是从 A 算法约束而来，而 A 算法又是从或图通用搜索算法进行限制得到的，因此很容易将或图通用搜索算法改造成为如下所示的 A^* 算法。

2. A^* 算法

设 S_0 为初始状态，S_g 为目标状态。

(1) Open=$\{S_0\}$。

(2) Closed=$\{\ \}$。

(3) 如果 Open=$\{\}$,失败退出。

(4) 从 Open 表中取出 $f(n)$值最小的节点 n,放到 Closed 表中。

$$f(n)=g(n)+h(n),\quad h\leqslant h^*$$

(5) 若 $n\in S_g$,则成功退出。

(6) 产生 n 的一切后继,将后继中不是 n 的先辈点的一切点构成集合 M。

(7) 考察 M 中的元素 P,分别作两类处理:

① 若 P$\notin$G,则对 P 进行估计加入 Open 表,记入 G 和 Tree。

② 若 P$\in$G,则决定更改 Tree 中 P 到 n 的指针,并且更改 P 的子节点 n 的指针和费用。

(8) 转(3)。

3. A* 算法的性质

A* 算法与一般的最佳优先算法比较,有其特有的优良性质:如果问题有解,即 $S_0\rightarrow S_g$ 存在一条路径,A* 算法一定能找到最优解。这一性质称为**可采纳性**。

例 3.4 继续讨论八数码问题的求解,如图 3.4 所示。

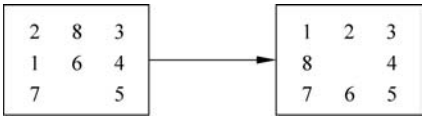

图 3.4 八数码问题求解

以前采用的估计函数为:

$$f=\text{放错位置的数字的个数}$$

现在采用:

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

其中,$d(n)$为搜索树的深度;$w(n)$为放错位置的数字的个数。这里 $g(n)=d(n)$,$h(n)=w(n)$,并且满足 $w(n)\leqslant h^*(n)$,因为 $w(n)$只估计放错位置的数字的个数,而这些数码从放错位置到正确位置要移动的步数,比这些数码的个数显然要大得多。用这样定义的 $f(n)$所得到的搜索图如图 3.5 所示。

在搜索第二层时,若 f 值相等,需要再规定一下如何优先选择后继:

(1) 后生成的节点优先,如图 3.5 中双线路径所示。

(2) 先生成的节点优先,则如图 3.5 中的粗线路径,但生成 3 个后继(C_1,C_2,C_3)之后,下次在算法的第 4 步仍会在 Open 中找最小,即在 Open 表中含有图 3.5 的如下 3 层的节点:

一层	二层	三层
($A_1=6$,$A_3=6$)	($B_2=5$,$B_3=6$)	($C_1=6$,$C_2=7$,$C_3=6$)

其中 B_2 最优,则仍可找到 B_2 继续扩展。我们通过跟踪 Open 表中元素可以看到这一点。

例 3.5 野人与传教士过河问题。有 N 个传教士和 N 个野人,要从左岸向右岸渡河,有一条船,每次最多可供 k 人乘渡。问为了传教士的安全起见,应如何规划摆渡方案,使得

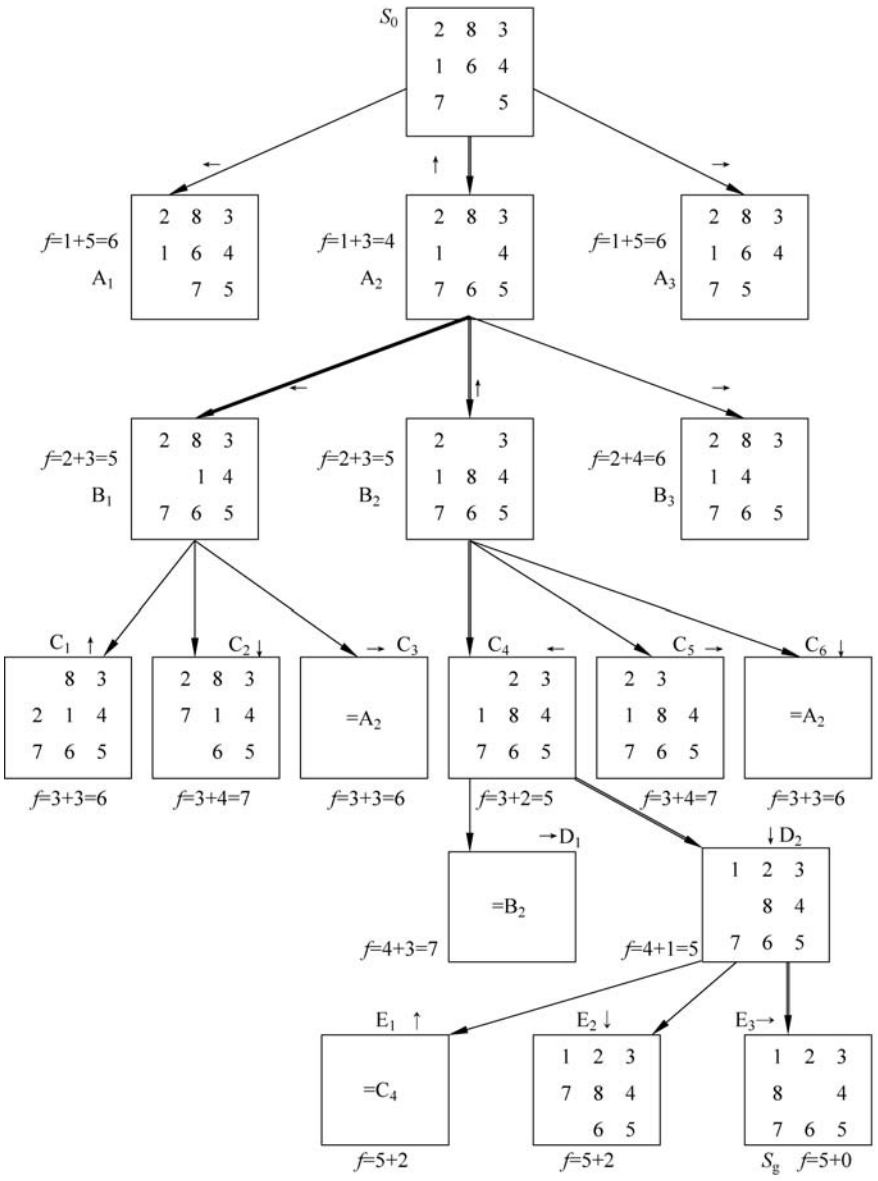

图 3.5　八数码问题使用 A* 算法的搜索图

任何时刻，传教士和野人在一起时，河两岸以及船上的野人数目总是不超过传教士的数目（否则传教士有可能被野人吃掉）。

这个问题要求在求解传教士和野人从左岸全部摆渡到右岸的过程中，任何时刻传教士和野人在一起时，满足传教士数大于等于野人数，并且两者人数之和小于或等于 k 的摆渡方案。该问题的求解过程如下。

设 M、C 为某一时刻传教士人数和野人人数，k 为船能够承载的人数。过河时，要求 $M \geqslant C$ 且 $M+C \leqslant k$。又设 L 表示船在左岸，R 表示船在右岸，并假设传教士和野人都为 5 个人，船一次可载 3 人时，即 $N=5$，$k=3$，该问题的状态空间可表示为：

初始状态

	L	R
M	5	0
C	5	0
B	1	0

目标状态

	L	R
M	0	5
C	0	5
B	0	1

其中，$B=1$ 代表在左岸，$B=0$ 代表不在左岸。由于野人和传教士的总数是一定的，可以只考虑左岸或右岸就可以了(这就减少了状态变量)，因此可以用如下方式考虑状态空间。

1) 定义状态空间

用三元组 $S_k=(ML,CL,BL)$来表示传教士、野人和船是否在左岸的状态，其中 ML 表示传教士在左岸的实际人数，CL 表示野人在左岸的实际人数，BL 用来指示船是否在左岸。

条件：$ML\geqslant 0, CL\leqslant 5, BL\in\{0,1\}$

问题即转化为：初始状态为 (5,5,1)，目标状态为 (0,0,0)。

为了表述的简洁，按每次渡河的人数分别写出每一个规则，共有(3,0)、(0,3)、(2,1)、(1,1)、(1,0)、(0,1)、(2,0)、(0,2)8 种渡河可能的组合(其中(x,y)表示 x 个传教士和 y 个野人一起上船渡河)，因此共有 16 个规则(从左岸到右岸、从右岸到左岸各 8 个)。注意，这里没有出现(1,2)，因为该组合在船上的传教士人数少于野人人数。

规则集如下：

P_{01} if$(ML,CL,BL=1)$ then$(ML,CL-1,BL-1)$

P_{02} if$(ML,CL,BL=1)$ then$(ML,CL-2,BL-1)$

P_{03} if$(ML,CL,BL=1)$ then$(ML,CL-3,BL-1)$

P_{10} if$(ML,CL,BL=1)$ then$(ML-1,CL,BL-1)$

P_{11} if$(ML,CL,BL=1)$ then$(ML-1,CL-1,BL-1)$

P_{20} if$(ML,CL,BL=1)$ then$(ML-2,CL,BL-1)$

P_{21} if$(ML,CL,BL=1)$ then$(ML-2,CL-1,BL-1)$

P_{30} if$(ML,CL,BL=1)$ then$(ML-3,CL,BL-1)$

Q_{01} if$(ML,CL,BL=0)$ then$(ML,CL+1,BL+1)$

Q_{02} if$(ML,CL,BL=0)$ then$(ML,CL+2,BL+1)$

Q_{03} if$(ML,CL,BL=0)$ then$(ML,CL+3,BL+1)$

Q_{10} if$(ML,CL,BL=0)$ then$(ML+1,CL,BL+1)$

Q_{11} if$(ML,CL,BL=0)$ then$(ML+1,CL+1,BL+1)$

Q_{20} if$(ML,CL,BL=0)$ then$(ML+2,CL,BL+1)$

Q_{21} if$(ML,CL,BL=0)$ then$(ML+2,CL+1,BL+1)$

Q_{30} if$(ML,CL,BL=0)$ then$(ML+3,CL,BL+1)$

为了理解以上规则的含义，通过规则 P_{21} 来说明：

P_{21} if$(ML,CL,BL=1)$ then $(ML-2,CL-1,BL-1)$

表示从左岸渡了两个传教士和一个野人到右岸，因此，左岸的传教士和野人的人数减少了，分别为 $ML-2$ 和 $CL-1$，而此时 $BL-1=0$，表示船在右岸。

Q_{21} 表示的意义正好相反，即从右岸渡了两个传教士和一个野人到左岸，因此左岸的传教士和野人的人数增加了。

2）启发式搜索策略

这里的启发式搜索策略使用如下 A 算法中的启发式函数表示：

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

其中，$g(n)$是从开始节点搜索到节点 n 时已经花费的代价；$h(n)$指启发式函数中的启发式部分，是从节点 n 到目标节点的最优路径的估计代价，搜索的启发信息主要由 $h(n)$来体现。

（1）启发式函数 1：

$$f(n)=\begin{cases} g(n), & \text{两岸的传教士数目} \geqslant \text{野人数目} \\ \infty, & \text{其他} \end{cases}$$

其中，$g(n)$表示搜索树的深度，也是船只来回的次数。f 函数中启发函数为零，即 $h(n)=0$，此时对应的搜索图如图 3.6 所示，图中还是针对 $N=5$、$k=3$ 时的搜索图。

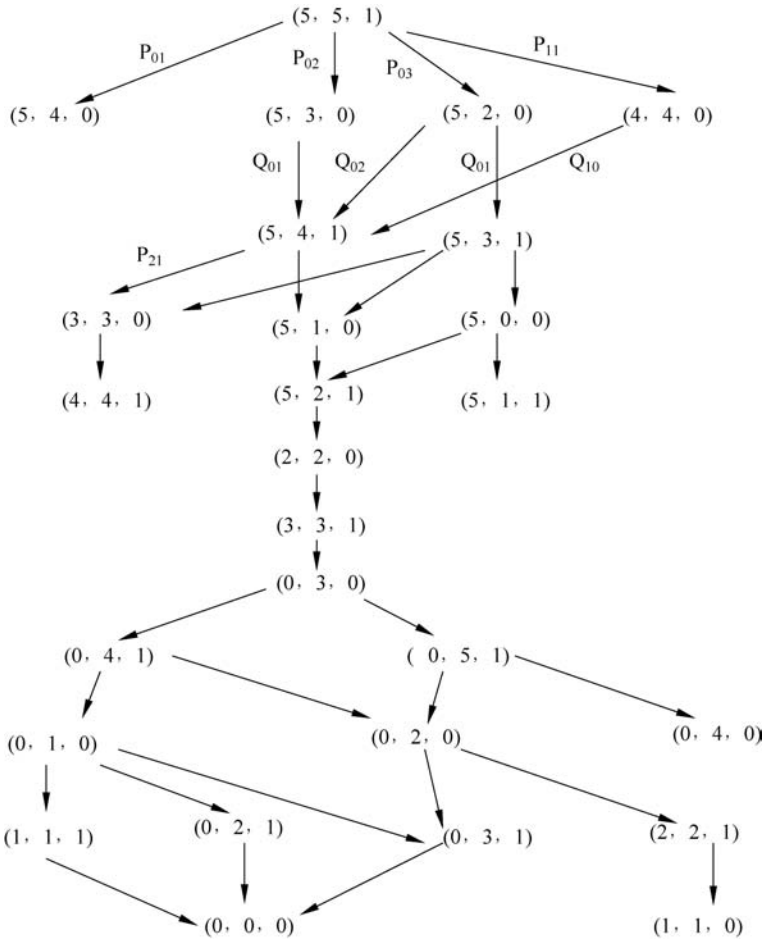

图 3.6　启发式函数 1 对应的搜索图

（2）启发式函数 2：

$$f(n)=\begin{cases} g(n)+M+C, & \text{两岸的传教士数目分别大于等于野人数目(当野人与传教士在一起时)} \\ \infty, & \text{其他} \end{cases}$$

即启发函数为 $h(n)=M+C$，此时对应的搜索图如图 3.7 所示。

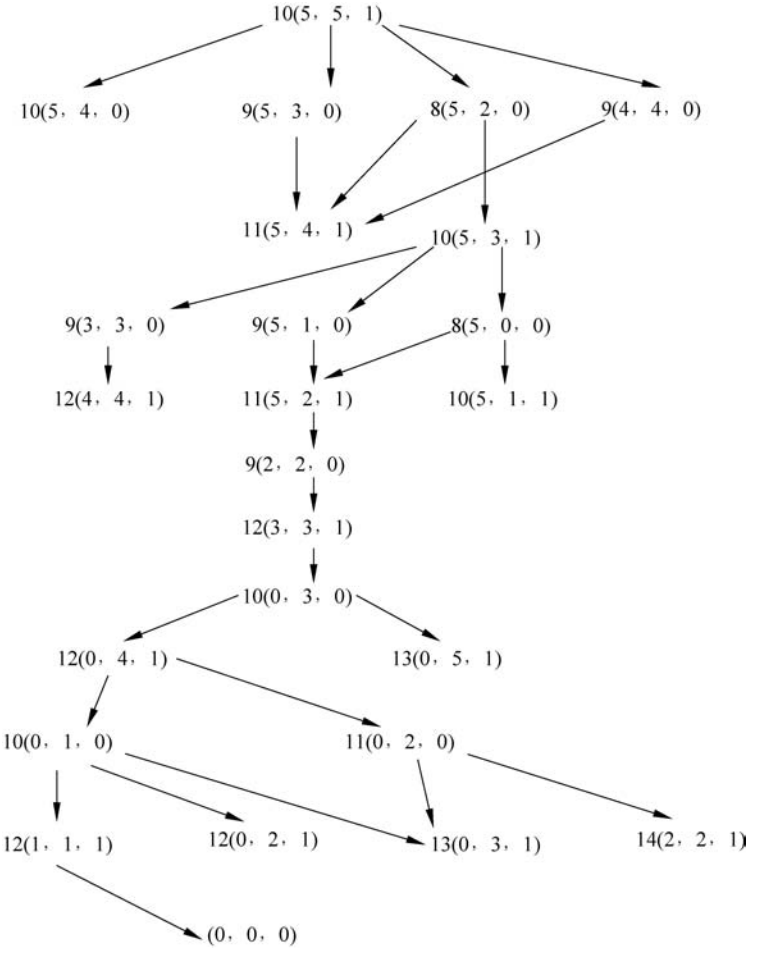

图 3.7　启发式函数 2 对应的搜索图

搜索图中的初始节点 10(5,5,1)表示 f 值为 $10 f(n)=g(n)+M+C=0+5+5=10$)，(5,5,1)表示有 5 个传教士、5 个野人在左岸，船在左岸。

（3）启发式函数 3：

$$f(n)=\begin{cases} g(n)+ML+CL-2BL, & \text{两岸的传教士数目} \geqslant \text{野人数目(当传教士与野人在一起时)} \\ \infty, & \text{其他} \end{cases}$$

其中，$g(n)$表示搜索树的深度，也是船只往返的次数。启发函数为 $h(n)=M+C-2B$，此时对应的搜索图如图 3.8 所示。

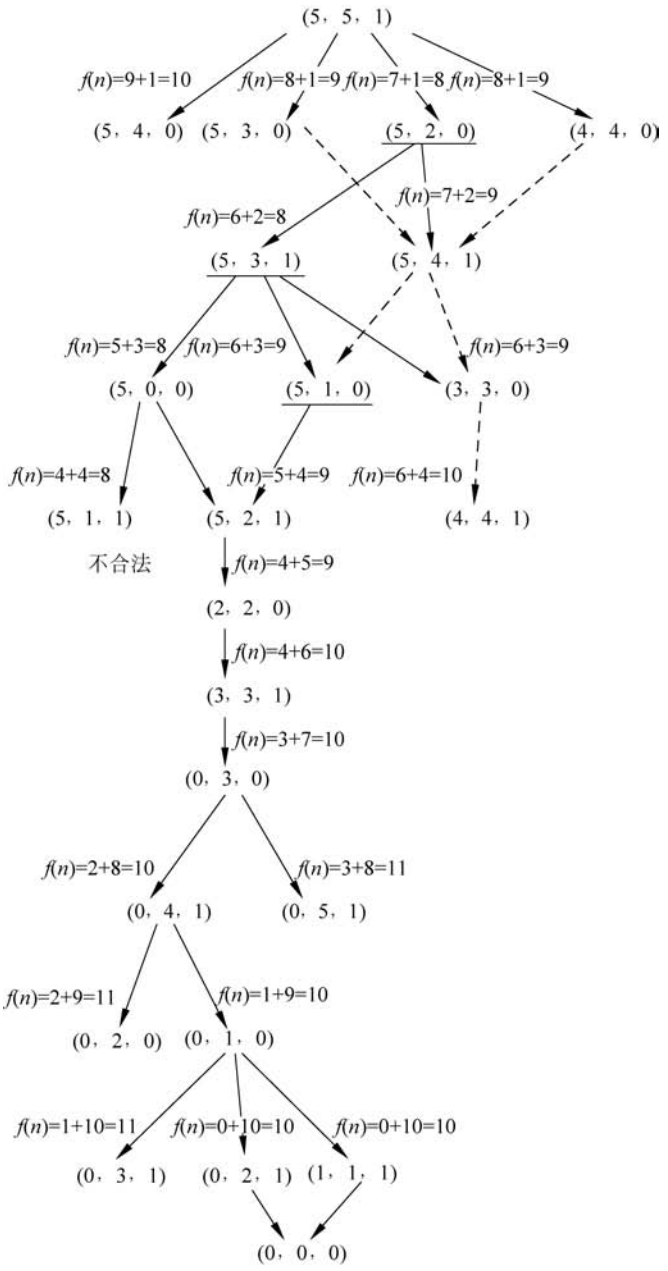

图 3.8　启发式函数 3 对应的搜索图

3.2　与/或图搜索

在问题求解过程中，往往需要将一个大的问题变换成若干个子问题，子问题又可分解成更小的子问题，这样一直分解到可以直接求解为止，全部子问题的解构成了整个问题的解，这样的问题求解过程称为问题的规约(Problem Reduction)。这也是一种搜索过程，不过现

在在图中搜索到的解不只是一个节点或一条路径，而是一个搜索树。因为在问题分解成子问题后，对于解决原来的问题有 3 种可能：

(1) 解决其中一个子问题就相当于解决原来的问题(分支之间是“或”关系)。

(2) 解决全部子问题才算解决原来的问题(分支之间是“与”关系)。

(3) 解决其中一些子问题就相当于解决原来的问题(分支之间“与”“或”关系都有)。

这样从原问题到子问题之间就存在 AND(全部解决)和 OR(部分解决)的关系。这就是“与/或(AND/OR)图”的来由。

3.2.1 问题归约求解方法和“与/或图”

在与/或图中，要求解的大问题称为初始问题，可直接求解的问题为本原问题。一般来说，使用归约方法求解问题需要具备三大要素：

(1) 初始问题的描述。

(2) 一组将问题变换成子问题的变换规则。

(3) 一组本原问题的描述。

例 3.6 符号积分问题。

(1) 初始问题描述：$\int f(x)\mathrm{d}x$。

(2) 变换规则：积分规则。

(3) 本原问题：可直接求原函数的积分，如$\int \sin(x)\mathrm{d}x$、$\int \mathrm{e}^{x}\mathrm{d}x$。

求解高等数学中的积分问题是典型的与/或图的搜索，一个积分问题可能有不同的求解方法，这些不同的解法之间就是“或”的关系；利用某些积分规则，例如，使用和式分解规则：$\int (f(x)+g(x))\mathrm{d}x \rightarrow \int f(x)\mathrm{d}x+\int g(x)\mathrm{d}x$，将一个积分问题分解成两个积分问题，这两个积分问题都要求解出来，才能算得上原积分问题求解完成了，这两个积分问题之间就是“与”的关系。

从初始问题出发，分解子问题以及子问题的子问题，直至把初始问题归约成为一个本原问题的集合，这就是问题规约方法求解问题的基本途径。

3.2.2 与/或图的构造方法

将问题求解归约为与/或图的搜索时，作如下规定：

(1) 与/或图中对应于原始问题描述的节点为初始节点；与/或图中对应于本原问题的节点叫终叶节点。

(2) 可解节点的可递归定义为：

① 终叶节点是可解节点；

② 若 n 为一非终叶节点，且含有“或”后继节点，则只有当后继节点中至少有一个是可解节点时，n 才可解；

③ 若 n 为一非终叶节点，且含有“与”后继节点，则只有当后继节点全部可解时，n 才可解。

(3) 不可解节点的可递归定义为：

① 没有后继节点的非终叶节点为不可解；

② 若 n 为一非终叶节点，且含有“或”后继节点，则仅当全部后继节点为不可解时，n 不可解；

③ 若 n 为一非终叶节点，且含有“与”后继节点，则至少有一个后继节点为不可解时，n 为不可解。

(4) 与或图搜索费用的计算：设从当前节点 n 到目标集 S_g 费用估计为 $h(n)$。

① 若 $n \in S_g$，则 $h(n)=0$；

② 若 n 有一组由“与”弧连接的后继节点 $\{n_1, n_2, \cdots, n_i\}$，则

$$h(n)=c_1+c_2+\cdots+c_i+h(n_1)+h(n_2)+\cdots+h(n_i)$$

其中，c_k 为 n 到 n_k 弧的费用；

③ 若 n 既有“与”弧又有“或”弧连接的后继，则一个“与”弧算作一个“或” 后继，再取各“或”弧所连接的后继中费用最小者为 n 的费用。

3.2.3　与/或图的搜索过程

对或图搜索，若搜索到某个节点时，则无论 n 是否生成了后继节点，n 的费用都是由本身的状态决定的。但对与/或图则不同，其费用计算的规则是：

(1) n 未生成后继节点时，费用由 n 本身的估计值决定，这个费用是给定的一个估计值；

(2) n 已生成后继节点时，费用由 n 的后继节点的费用决定，即利用 3.2.2 中搜索费用计算的第(1)～(4)步的方法进行计算。

因为后继节点代表分解的子问题，子问题的难易程度决定原问题求解的难易程度，所以不再考虑 n 在之前估计的难易程度。因此当决定了某个路径时，要将后继节点的估计值往回传送。

下面举例说明这个过程。

例 3.7　图 3.9 为一个与/或图的搜索过程。

(1) A 是唯一节点。

(2) 扩展 A 后，得到节点 B、C 和 D，因为 B、C 的耗费为 9，D 的耗费为 6，所以把到 D 的边标志为出自 A 最有希望的边。

(3) 选择对 D 的扩展，得到 E 和 F 的与弧，其耗费估计值为 10。此时回退一步后，发现与弧 BC 比 D 更好，所以将弧 BC 标志为目前最优路径。

(4) 在扩展 B 后，再回传值发现弧 BC 的耗费为 12(6＋4＋2)，所以 D 再次成为当前最优路径。

最后求得的耗费为：$f(\mathrm{A})=\min(12, 4+4+2+1)=11$。

以上搜索过程由两个步骤组成：

(1) 自顶向下，沿当前最优路径产生后继节点。

(2) 自底向上，作估计值修正，再重新选择最优路径。

与/或图搜索仅对不含循环路径的图进行操作，因为循环路径表示循环推理，它不可能对问题进行规约。

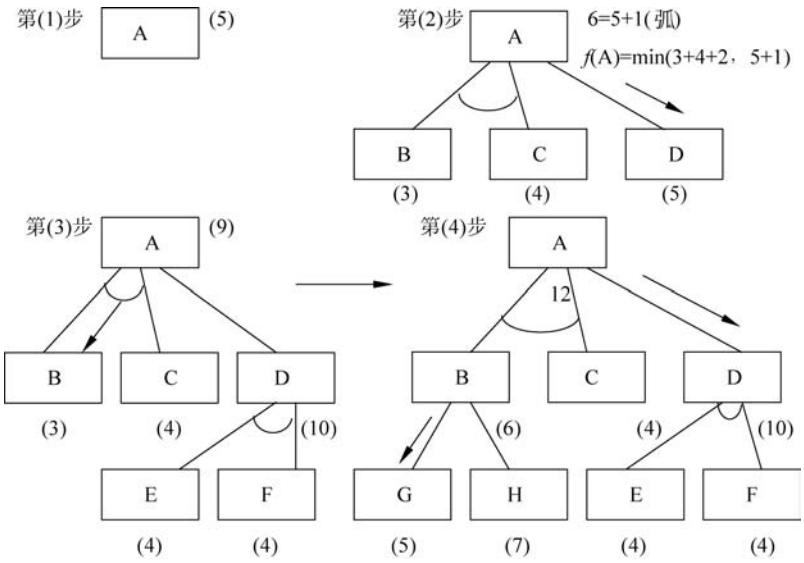

图 3.9　一个与/或图的搜索过程

例如：

表示求解 x 可以归结为求 y，求 y 又可以归结为求 x，因而两者都不可能求解。

3.2.4　与/或图搜索算法 AO*

AO* 算法用一个阈值 Futility 作为不可解节点的标志，用 h' 作为静态估计函数，用 mark 作为当前最优路径的标记。

AO* 算法步骤为：

(1) 令 G 仅由初始状态节点组成，称为 Init，计算 h'(Init)。

(2) 在 Init 标志 solved 之前或 h'(Init)变成大于 Futility 之前，执行以下步骤：

① 沿始于 Init 的已带标志的弧，选出当前沿标志路上未扩展的节点之一扩展(即求后继节点)，此节点称为 node。

② 生成 node 的后继节点。

若无后继节点，则令 h'(node)＝Futility，说明该节点不可解。

若有后继节点，称为 successor，对每个不是 node 祖先的后继节点(避免回路)，执行以下步骤：

i 将 successor 加入 G。

ii 若 successor$\in S_g$，则标志 successor 为 solved，且令 h'(successor)＝0。

iii 若 successor$\notin S_g$，则求 h'(successor)。

③ 自底向上作评价值修正，重新挑选最优路径。

令 S 为一节点集。

S＝{已标志为 solved 的点，或 h' 值已改变，需回传至其先辈节点的节点}

令 S 初值＝{node}，重复以下过程，直到 S 为空时停止。

i 从 S 中挑选一节点，该节点的后辈点均不在 S 中(保证挑选出的要处理的点都在其先辈节点之前作处理)，此节点称为 current，并从 S 中删除。

ii 计算始于 current 的每条弧及其后继节点的费用，即每条弧本身的费用加上弧末端节点 h' 的值(注意按与或图搜索费用的计算规则，区分与弧和或弧的计算方法)，并从中选出极小费用的弧作为 h'(current)的新值。

iii 将费用最小弧标志为出自 current 的最优路径。

iv 若 current 与新的带标志的弧所连接的点均标志为 solved，则 current 标志为 solved。

v 若 current 已标志为 solved 或 current 的费用已改变，则需要往回传，因此要将 current 的所有先辈节点加入 S 中。

3.2.5　用 AO* 算法求解一个智力问题

有这样一个智力问题：有 12 枚硬币，凡轻于或重于真币者，即为假币(只有一枚假币)，要设计一个搜索算法来识别假币并指出它是轻于还是重于真币，且利用天平的次数不多于 3 次。

该问题的困难之处在于问题要求只称 3 次就要找到假币，否则就承认失败。如果称法不得当，使得留下的未知币太多，就不可能在 3 次内称出假币。因此，每称一次，我们希望尽可能地得到关于假币的信息。

利用人工智能的求解方法解决这个问题首先必须解决下面两个问题：

(1) 问题表示方法，记录和描述问题的状态；

(2) 求解程序如何对某种称法进行评价。

下面就对使用 AO* 算法求解这个智力难题进行讨论。

1. 问题的表示

要把这个问题在计算机中表示出来，就要分析构成该问题状态的因素有哪些：首先是硬币可能有哪些状态；然后是使用天平每称一次后，有关硬币的状态会发生什么样的变化；最后是每称一次后，必须保存所剩的使用天平的次数。

可将硬币的重量状态分为 4 种类型：

(1) 标准型(Standard)，标记为 S；

(2) 轻标型(Light or Standard)，标记为 LS；

(3) 重标型(Heavy or Standard)，标记为 HS；

(4) 轻重标准型(Light or Heavy or Standard)，标记为 LHS。

一个硬币为 LHS 状态，那是我们对它一无所知；LS 和 HS 状态是有可能为轻的或有可能为重的，当然也可能是标准的；S 状态是已知为标准的。

例如，一次称两个硬币，如果天平偏向左边，则天平左盘中的硬币属于重标型，而右盘中的硬币属于轻标型，其余不在天平上的属于标准型(因为只有一个假币)。每称一次，硬币的重量状态可能会从一种类型转变为另一种类型。问题处于初始状态时，所有的硬币均属于 LHS 型。

综上所述，问题的状态空间可表示成一个五元组：

```
(lhs,ls,hs,s,t)
```

其中，前 4 个元素表示当前这 4 种类型硬币的个数，t 表示所剩称硬币的次数。在这样的状态空间表示下，有：

初始状态：$(l2,0,0,0,3)$

目标状态：sg_1：$(0,1,0,11,0)$和 sg_2：$(0,0,1,11,0)$

其中，sg_1 和 sg_2 分别表示最后找到一个轻的或找到一个重的硬币，其余 11 个为标准硬币。

2. 如何利用 AO* 算法求解

利用 AO* 算法求解问题需要找出如下要素：

(1) 初始问题的描述；

(2) 一组将问题变换成子问题的变换规则；

(3) 一组本原问题描述。

该问题的初始问题前面已经表示出来了，本原问题就是两个目标状态。下面要定义一组转换规则。这里的转换规则就是每称一次，要考虑如何取硬币放到天平上，然后称完后，根据天平的状态，硬币的重量状态可能会从一种类型转变为另一种类型。

首先考虑如何取硬币的问题。设当前的状态为(lhs,ls,hs,s,t)，用函数 PICKUP$([lhs_1,ls_1,hs_1,s_1],[lhs_2,ls_2,hs_2,s_2])$表示本次分别从$(lhs,ls,hs,s)$中取出 lhs_1、ls_1、hs_1、s_1 个硬币放到天平的左边，取出 lhs_2、ls_2、hs_2、s_2 个硬币放到天平的右边。对于 PICKUP 应默认有如下性质成立：

(1) $0<lhs_1+ls_1+hs_1+s_1=lhs_2+ls_2+hs_2+s_2\leqslant 6$，即天平两边的硬币数相等且小于等于 6；

(2) $lhs_1+lhs_2\leqslant lhs \wedge ls_1+ls_2\leqslant ls \wedge hs_1+hs_2\leqslant hs \wedge s_1+s_2\leqslant s$，即取出的硬币数小于等于相应类型原有的硬币数。

然后令 PICKUP()等于-1、0、1 分别表示天平左倾斜、平衡和右倾斜。在这个定义下，有如下转换规则。

(1) 左倾斜规则：

if PICKUP $([lhs_1,ls_1,hs_1,s_1],[lhs_2,ls_2,hs_2,s_2])=-1 \wedge (lhs,ls,hs,s,t)$

then $(lhs',ls',hs',s',t-1)$；

其中，$s'=s+ls-ls_2+hs-hs_1+lhs-(lhs_1+lhs_2)$；$ls'=ls_2+lhs_2$；$hs'=lhs_1+hs_1$；$lhs'=0$。

这 4 个公式的含义分别是：若天平左倾，则在左天平的状态为 LS 的硬币、在右天平的状态为 HS 的硬币和未放到天平上的硬币都是标准的，即 $hs_2+ls_1+(lhs-lhs_1-lhs_2)+(ls-ls_1-ls_2)+(hs-hs_1-hs_2)$ 个硬币的状态都改变为标准型；右天平原有的 ls_2 个轻标准型的硬币仍然为轻标准型，右天平的 lhs_2 个轻重标准型硬币改变为轻标准型；左天平原有的 hs_1 个重标准型的硬币仍然为重标准型，左天平的 lhs_2 个轻重标准型硬币改变为重标准型；只要不平衡，就不存在 LHS 型的硬币，天平上的硬币可以确定为 LS 或 HS 型，天平下的硬币可以确定为 S 型，这时 $lhs'=0$。

(2) 平衡规则：

if PICKUP $([lhs_1,ls_1,hs_1,s_1],[lhs_2,ls_2,hs_2,s_2])=0 \wedge (lhs,ls,hs,s,t)$

then $(lhs',ls',hs',s',t-1)$；

其中，$s'=s+ls_1+ls_2+hs_1+hs_2+lhs_1+lhs_2$；$ls'=ls-ls_1-ls_2$；$hs'=hs-hs_1-hs_2$；$lhs'=lhs-lhs_1-lhs_2$。

这4个公式的含义分别是：若天平平衡，则所有在天平上的硬币都是标准的，即有 $ls_1+ls_2+hs_1+hs_2+lhs_1+lhs_2$ 个硬币的状态都改变为标准型；左、右天平原有的轻标准型的硬币改变为标准型，所以从 ls 中减去 ls_1 和 ls_2；左、右天平原有的重标准型的硬币改变为标准型，所以也要从 hs 中减去 hs_1 和 hs_2；在平衡情况下，lsh 型的硬币要减去左、右天平原有的轻重标准型的硬币，即 $lhs'=lhs-lhs_1-lhs_2$。

(3) 右倾斜规则：

if PICKUP $([lhs_1,ls_1,hs_1,s_1],[lhs_2,ls_2,hs_2,s_2])=1 \wedge (lhs,ls,hs,s,t)$

then $(lhs',ls',hs',s',t-1)$；

其中，$s'=s+ls-ls_1+hs-hs_2+lhs-(lhs_1+lhs_2)$；$ls'=ls_1+lhs_1$；$hs'=lhs_2+hs_2$；$lhs'=0$。

这4个公式的含义与L规则的含义类似：若天平右倾，则在左天平上状态为HS的硬币和右天平上状态为LS的硬币以及未放到天平上的硬币都是标准的，即 $hs_1+ls_2+(lhs-lhs_1-lhs_2)+(ls-ls_1-ls_2)+(hs-hs_1-hs_2)$ 个硬币的状态都改变为标准型；左天平原有的 ls_1 个轻标准型的硬币仍然为轻标准型，左天平的 lhs_1 个轻重标准型硬币改变为轻标准型；右天平原有的重标准型的 hs_2 个硬币仍然为重标准型，右天平的 lhs_2 个轻重标准型硬币改变为重标准型；因为不平衡，$lhs'=0$。

以上3个规则中 $t-1$ 表示所剩使用天平的次数减少了一次。

3. 如何在问题空间中搜索

该问题可以用 AO* 算法求解，主要是基于这样的背景：用PICKUP()填入不同的参数表示一种选取方法，不同的选取方法之间是或的关系，当选定一组PICKUP的参数后，就必须考虑它的值为 −1、0、1 时下层的节点都可解，则3种情况之间的关系为“与”的关系。这说明该问题的搜索图是与或图，图3.10给出了这个问题的搜索图。

用于该算法的评价函数可设置为：

$$h(lhs,ls,hs,s,t)=ls+hs+lhs-1$$

显然有：

$$h(0,1,0,11,0)=0 \text{ 和 } h(0,0,1,11,0)=0$$

即 $h(sg_1)=h(sg_2)=0$。

由于问题的本原问题对应的节点是可解节点，因此 sg_1、sg_2 为可解节点。不可解节点可定义为：如果节点 $n=(lhs,ls,hs,s,t)$ 中 t=0 且 (lhs,ls,hs,s) 不属于{(0,1,0,11),(0,0,1,11)}，则 n 为不可解节点。

作好上述准备工作后，就可用 AO* 算法进行求解，图3.10就是用 AO* 算法得到的一个解图。

用 AO* 算法可以很快地接近目标，因为对于某种选法，它的节点层数不超过三层，且某个节点只要有一个下层的 and 节点为不可解节点，则马上推出该节点为不可解节点。例如，考虑 PICKUP([6,0,0,0],[6,0,0,0])，它的值只可能是1或−1，它的下层节点实际只有一个为(0,6,6,0,2)的节点。对于它的下两层的节点的搜索可很快导出都是不可解节点。因而导出 PICKUP([6,0,0,0],[6,0,0,0])不是一种正确的取法。同样的道理也可以很快

导出 PICKUP([1,0,0,0],[1,0,0,0])、PICKUP([2,0,0,0],[2,0,0,0])等都不是正确的取法。因为使用 AO* 算法抛去了大量的不可解的分支,所以可以很快地找出所有解。

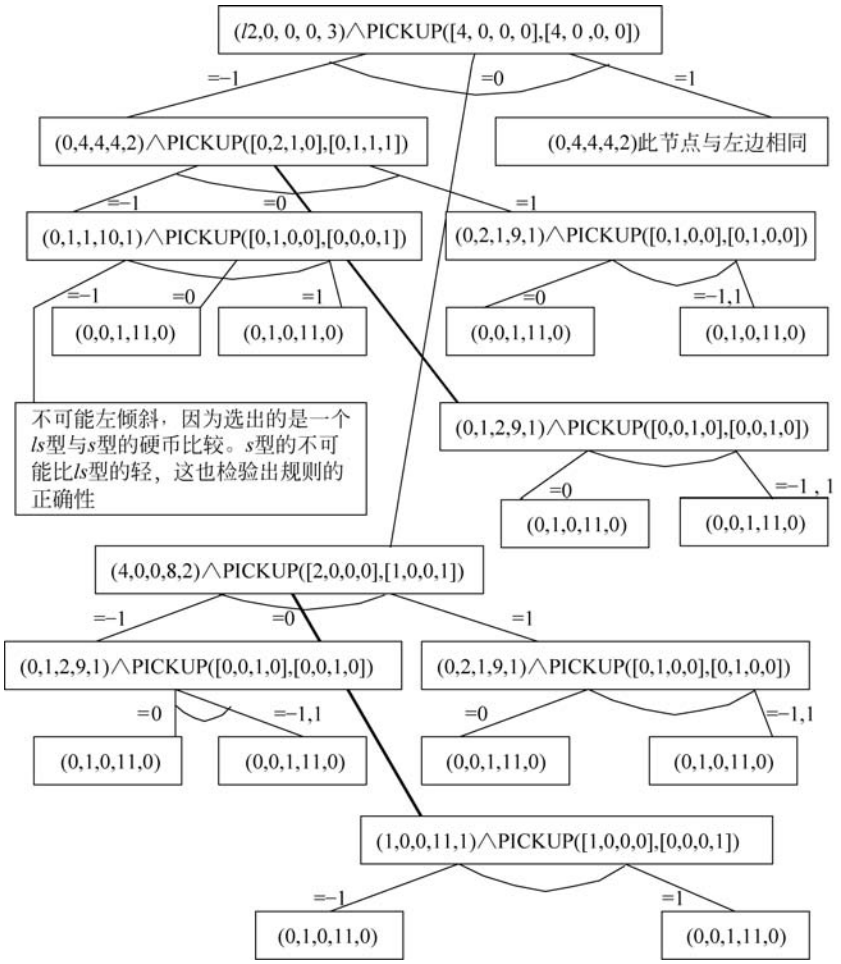

图 3.10　利用 AO* 算法求解 12 硬币问题的解图

习题 3

3.1　对 $N=5$、$k\leqslant 3$ 的传教士和野人问题,定义两个 h 函数(非零),并给出用这两个启发函数的 A 算法搜索图。讨论用这两个启发函数求解该问题时是否得到最优解。

3.2　什么是图搜索过程? 其中,重排 Open 表意味着什么,重排的原则是什么?

3.3　什么是 A* 算法? 它的评价函数如何确定,它与 A 算法有什么区别?

3.4　证明 Open 表上具有 $f(n)<f^*(s)$的任何节点 n,最终都将被 A* 选择去扩展。

3.5　怎么用一架天平 3 次称出 13 个硬币中唯一的然而未知轻重的假币(已知有标准的硬币)?

3.6　A* 算法有哪些性质?

3.7　请给出通用图搜索算法中,Open 表和 Closed 表所表示的一般的含义。

3.8　A^* 算法在什么条件下，执行效果最好？为什么？

3.9　“或图”和“与或图”各对应什么样的实际背景？所对应的最优搜索算法是什么？

3.10　请给出使用天平 4 次从 39 个硬币中找出唯一的未知轻重的假币的方案。

3.11　在问题的分析过程中，将一个大的复杂问题分解为一组简单的问题，这组简单的问题解决了，则大的复杂问题解决了，这是什么逻辑关系？应该分解成什么样的搜索树？如果将一个较难的问题变换为容易的、等价的或等效的问题，变换后的问题解决了则原来的问题解决了，这是什么逻辑关系？应该分解成什么样的搜索树？

3.12　请写出可解和不可解节点的递归定义。