第4章　图搜索技术

图搜索是人工智能的核心技术之一，人工智能的许多分支领域都涉及到图搜索。

这里的图是指节点和有向边组成的网络。按连接同一节点的各边间的逻辑关系，图可分为或图（直接图）和与或图两类。

4.1　状态图搜索

4.1.1 状态图

我们通过例子引入状态图的概念。

例4.1　走迷宫是人们熟悉的一种游戏，如图4–1就是一个迷宫。如果我们把该迷宫的每一个格子以及入口和出口都作为节点，把通道作为边，则该迷宫可以由一个有向图表示(如图4—2所示)。那么，走迷宫其实就是从该有向图的初始节点(入口)出发，寻找目标节点(出口)的问题，或者是寻找通向目标节点(出口)的路径的问题。

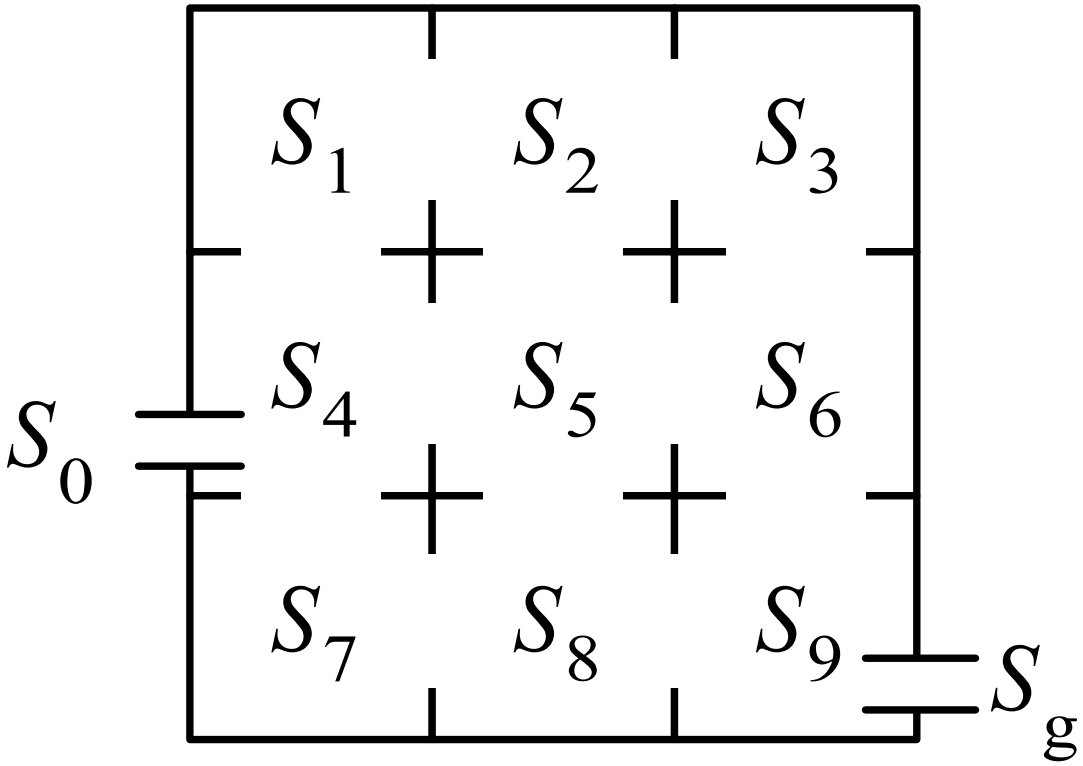

图4—1 迷宫图

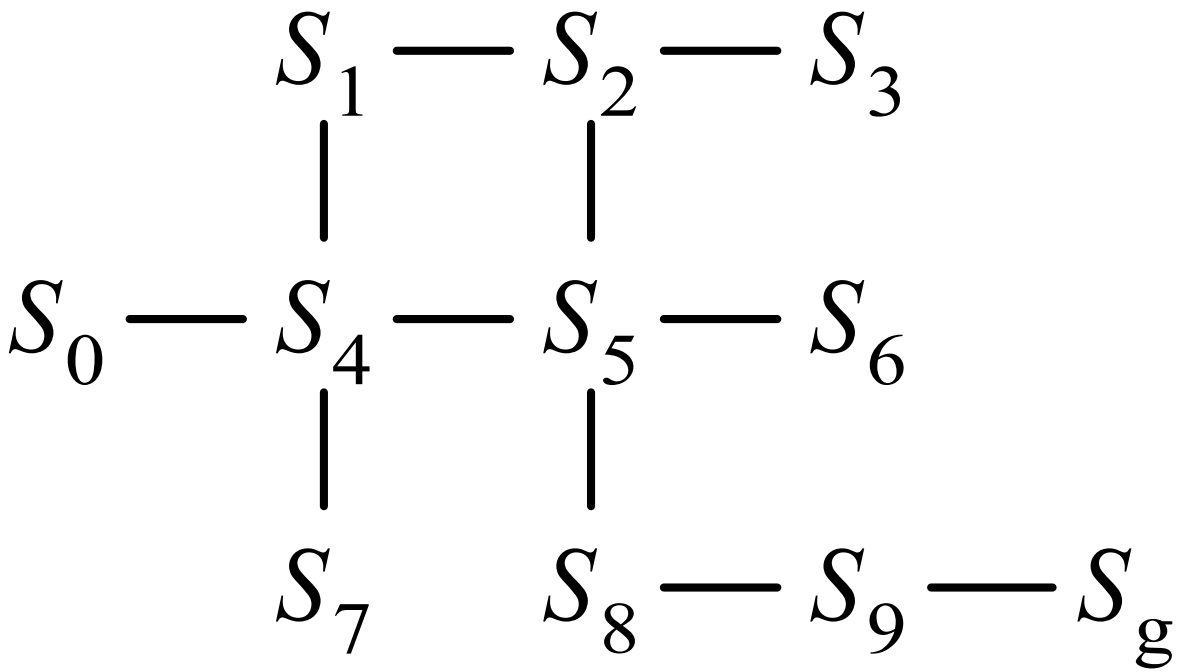

图4—2 迷宫的有向图表示

例 4.2　在一个 3×3 的方格棋盘上放置着 1,2,3,4,5,6,7,8八个数码，每个数码占一格，且有一个空格。这些数码可在棋盘上移动，其移动规则是：与空格相邻的数码方可移入空格。现在的问题是：对于指定的初始棋局和目标棋局(如图4–3所示)，给出数码的移动序列。该问题称为八数码难题或重排九宫问题。

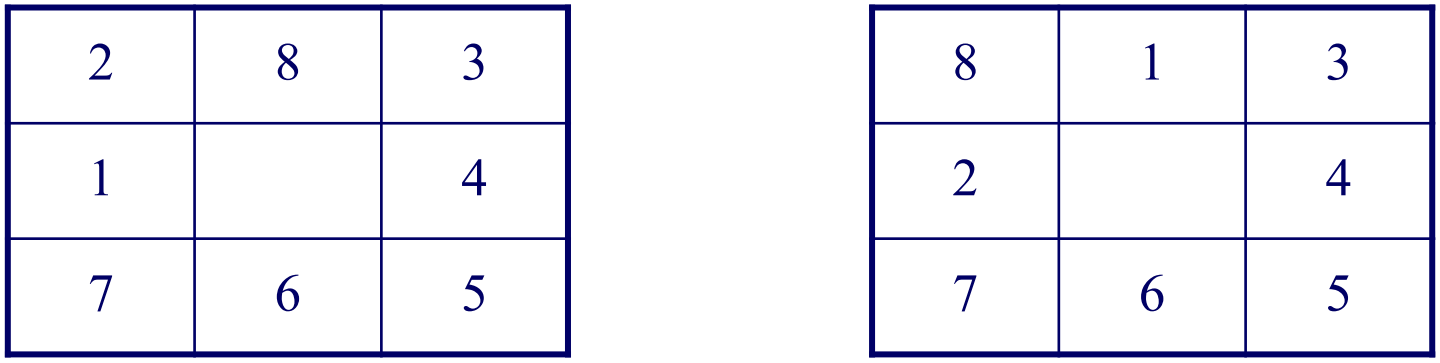

图4—3　八数码问题

2	8	3
1		4
7	6	5

初始棋局

2	8	3
	1	4
7	6	5

中间棋局

2		3
1	8	4
7	6	5

中间棋局

2	8	3
1	4	
7	6	5

中间棋局

图4—3—1　八数码棋局移动规则

可以看出，我们把一个棋局作为一个节点，相邻的节点就可以通过移动数码产生出来。这样，所有节点就可由他们的相邻关系连接成一个有向图。可以看出，图中的一条边（即相邻两个节点的连线）就是对应一次数码移动，反之，一次数码移动也对应图中的一条边。而数码移动是按数码的移动规则进行的。所以，图中的一条边也就代表一个移动规则或移动规则的一次执行。于是，这个把数码问题就是要在该有向图中寻找目标节点，或找一条从初试节点到目标节点的路径问题。

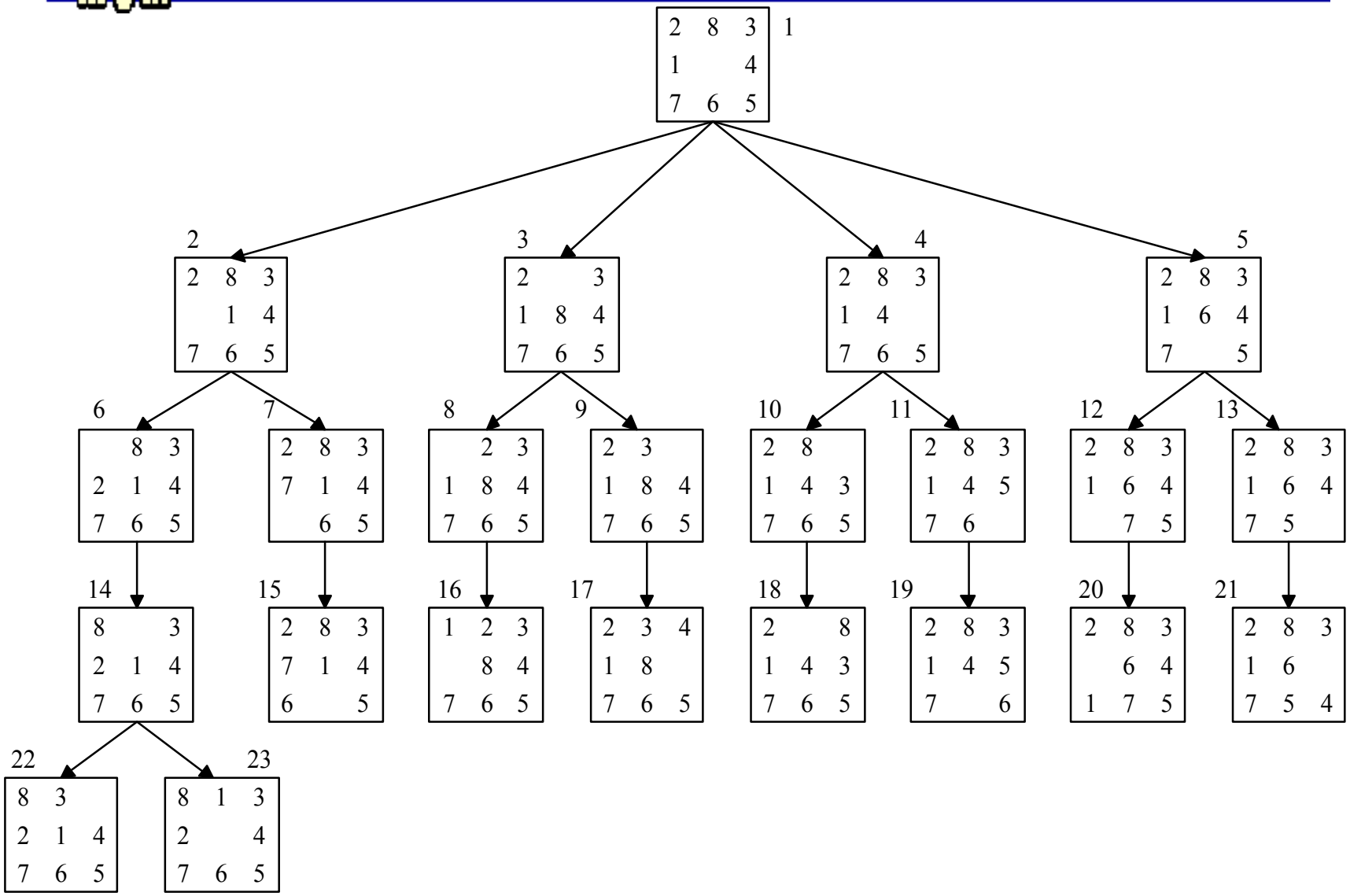

图4—5　八数码问题的广度优先搜索

以上两个问题都是在某个有向图中寻找目标或路径问题，这种问题图搜索问题。把描述问题的有向图称为状态空间图，简称状态图。图中的节点代表问题中的一种格局，一般称为问题的一个状态，边表示两个状态之间的联系。在状态图中，从初始节点到目标节点的一条路径或者所找到的目标节点，就是问题的解（路径解）。

状态图实际上是一类问题的抽象表示。事实上，有许多智力问题(如梵塔问题、旅行商问题、八皇后问题、农夫过河问题等)和实际问题（如路径规划、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。因此，研究状态图搜索具有普遍意义。

4.1.2 状态图搜索

在状态图中寻找目标或路径的基本方法就是搜索。所谓搜索，顾名思义，就是从初始节点出发，沿着与之相连的边试探地前进，寻找目标节点的过程(也可以反向进行)。搜索过程中经过的节点和边，按原图的连接关系，形成树型的有向图，称为搜索树。搜索过程应当随时记录搜索痕迹。

1.搜索方式

用计算机来实现状态图的搜索，有两种最基本的方式：树式搜索和线式搜索。

所谓树式搜索，形象地讲就是以“画树”的方式进行搜索。 即从树根（初始节点）出发一笔一笔地描出来的。

准确地讲，树式搜索就是在搜索过程中记录所经过的所有节点和边。所以，树式搜索所记录的轨迹始终是一棵“树”，这棵树也就是搜索过程中产生的搜索树。

线式搜索，形象地讲就是以“画线”的方式进行搜索。准确地讲，线式搜索就是在搜索过程中只记录那些当前认为是处在所找路径上的节点和边。所以，线式搜索所记录的轨迹始终是一条“线”（折线）。线式搜索的基本方式可以分为不回溯和可回溯两种。

怎样从搜索树中找出所求路径呢？这只需要在扩展节点时记住节点间的父子关系。当搜索成功后从目标节点反向沿搜索树所作标记追溯回去一直到初始节点，便可得到一条从初始节点到目标节点的路径，即问题的一个解。

2.搜索策略

由于搜索具有探索性，所以要提高搜索效率(尽快地找到目标节点)，或要找最佳路径(最佳解)就必须注意搜索策略。对于状态图搜索，已经提出了许多策略，它们大体可分为盲目搜索和启发式（heuristic）搜索两大类。

盲目搜索就是无“向导”搜索。树式盲目搜索就是穷举搜索，线式盲目搜索对不可回溯的是随机碰撞搜索，对可回溯的穷举搜索。

启发式搜索是有“向导”搜索，利用启发性信息引导的搜索。

按搜索范围的扩展顺序不同，搜索又可分为广度优先和深度优先两种搜索。对于树式搜索，既可深度优先进行，也可广度优先进行。对于线式搜索则总是深度优先进行。

3. 搜索算法

由于搜索的目的是为了寻找初始节点到目标节点的路径，所以在搜索过程中就得随时记录搜索轨迹。为此，我们用一个称为CLOSED表的动态数据结构来专门记录考查过的节点。显然，对于树式搜索来说，CLOSED表中存储的正是一棵不断成长的搜索树；而对于线式搜索来说，CLOSED表中存储的是一条不断伸长的折线，它可能本身就是所求的路径(如果能找到目标节点的话)。

另一方面，对于树式搜索来说，还得不断地把待考查的节点组织在一起，并做某种排列，以便控制搜索的方向和顺序。为此，我们采用一个称为OPEN表的动态数据结构，来专门登记当前待考查的节点。OPEN表和CLOSED表的结构如图4—4所示。

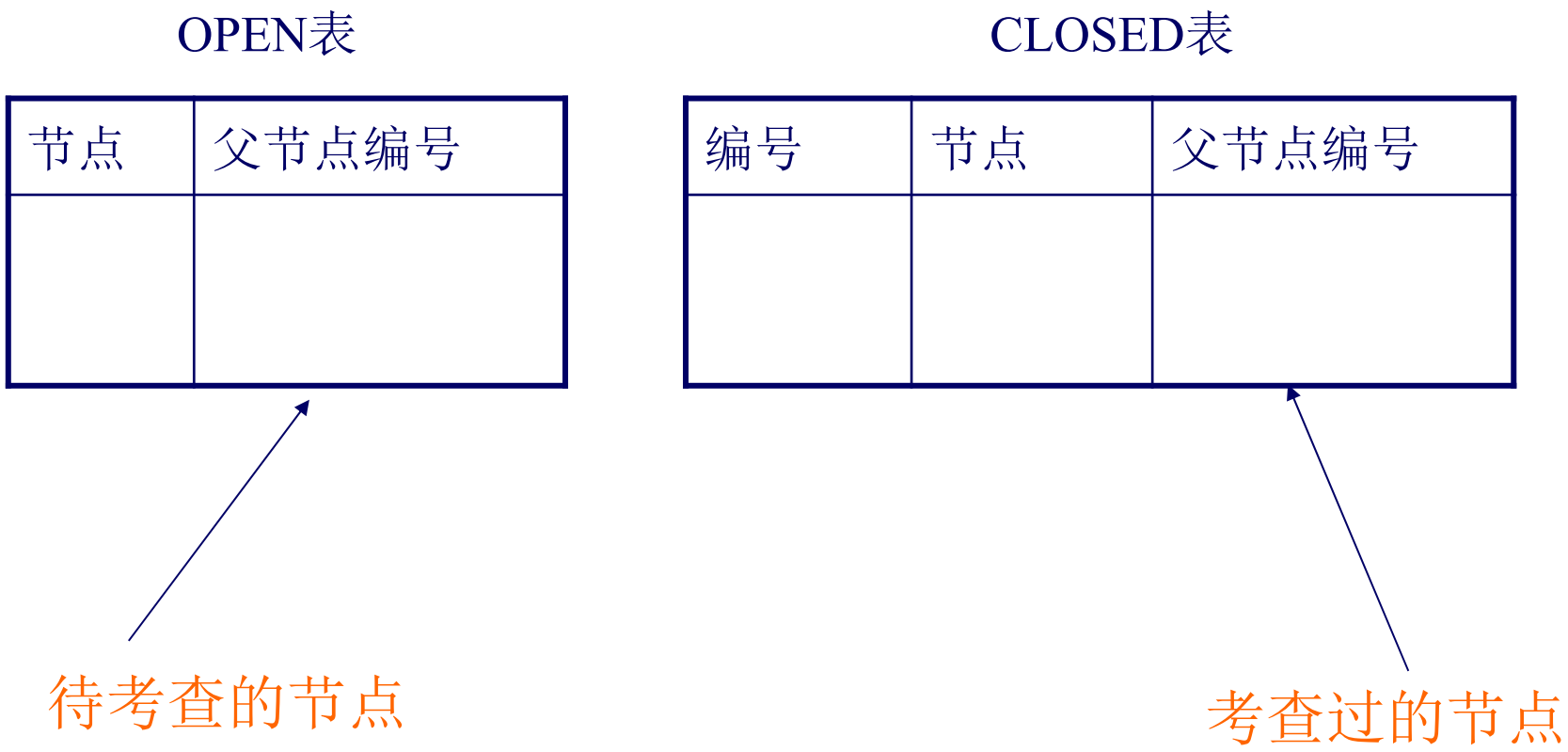

图4—4　OPEN表与CLOSED表示例

树式搜索算法：

步1 把初始节点S_0放入OPEN表中；

步2 若OPEN表为空，则搜索失败，退出。

步3 取OPEN表中前面第一个节点N放入CLOSED表中，并冠以顺序编号n；

步4 若目标节点S_g=N，则搜索成功，结束。

步5 若N不可扩展，则转步2；

步6 扩展N，生成一组子节点，对这组子节点作如下处理：

（1）删除N的先辈节点（如果有的话）；

（2）对已存在于OPEN表中的节点（如果有的话）也删除掉；但删除之前要比较返回初始节点的新路径与原路径，如果新路径“短”，则修改这些节点在OPEN表中的返回指针，使其沿新路返回；

（3）对已存在于CLOSED表中的节点（如果有的话），也作与（2）同样的处理，修改返回指针，并且将其移出CLOSED表，放入OPEN表重新扩展（为了重新计算代价）；

（4）对其余子节点配上指向N的返回指针依次放入OPEN表的某处，或对OPEN表进行重新排序，转步2。

说明：

（1）返回指针是父节点在CLOSED表中的编号；

（2）步6修改返回指针的原因是，因为这些节点又被第二次生成；

（3）这里对路径的长短是按路径上的节点数来痕量的，还可以按代价（如距离、时间、费用）衡量。

4.1.3 穷举式搜索

为简单起见，下面我们先讨论树型结构的状态图搜索，并仅限于树式搜索。

按搜索树生成方式的不同，树式穷举搜索又分为广度优先和深度优先两种搜索方式。这两种方式也是最基本的树式搜索策略，其他搜索策略都是建立在它们之上的。下面先介绍广度优先搜索。

1. 广度优先搜索

广度优先搜索就是始终先在同一级节点中考查，只有当同一级节点考查完之后，才考查下一级节点。或者说，是以初始节点为根节点，向下逐级扩展搜索树。所以，广度优先策略的搜索树是自顶向下一层一层逐渐生成的。

广度优先搜索算法：

步1 把初始节点S_0放入OPEN表中；

步2 若OPEN表为空，则搜索失败，退出。

步3 取OPEN表中前面第一个节点N放入CLOSED表中，并冠以顺序编号n；

步4 若目标节点S_g=N，则搜索成功，结束。

步5 若N不可扩展，则转步2；

步6 扩展N，将其所有子节点配上指向N的返回指针依次放入OPEN表的尾部，转步2。

其中的OPEN表是一个队列，CLOSED表是一个顺序表，表中各节点按顺序编号，正被考察的节点在表中编号最大。如果问题有解，OPEN表中必然出现目标节点S_g，那么，当搜索到目标节点S_g时，算法结束，然后根据返回指针在CLOSED表中往回追溯，直至初始节点，所得的路径即为问题的解。

广度优先搜索亦称为宽度优先或横向搜索。这种策略是完备的，即如果问题的解存在，用它则一定能找到解，且找到的解还是最优解（即最短的路径）。这是优点，缺点是搜索效率低。

例4.3 用广度优先搜索策略解八数码难题。

由于把一个与空格相邻的数码移入空格，等价于把空格向数码方向移动一位。所以，该题中给出的数码走步规则也可以简化为：对空格可施行左移、右移、上移和下移等四种操作。

设初始节点S_0和目标节点S_g分别如图4–3的初始棋局和目标棋局所示，我们用广度优先搜索策略，则可得到如图4–5所示的搜索树。

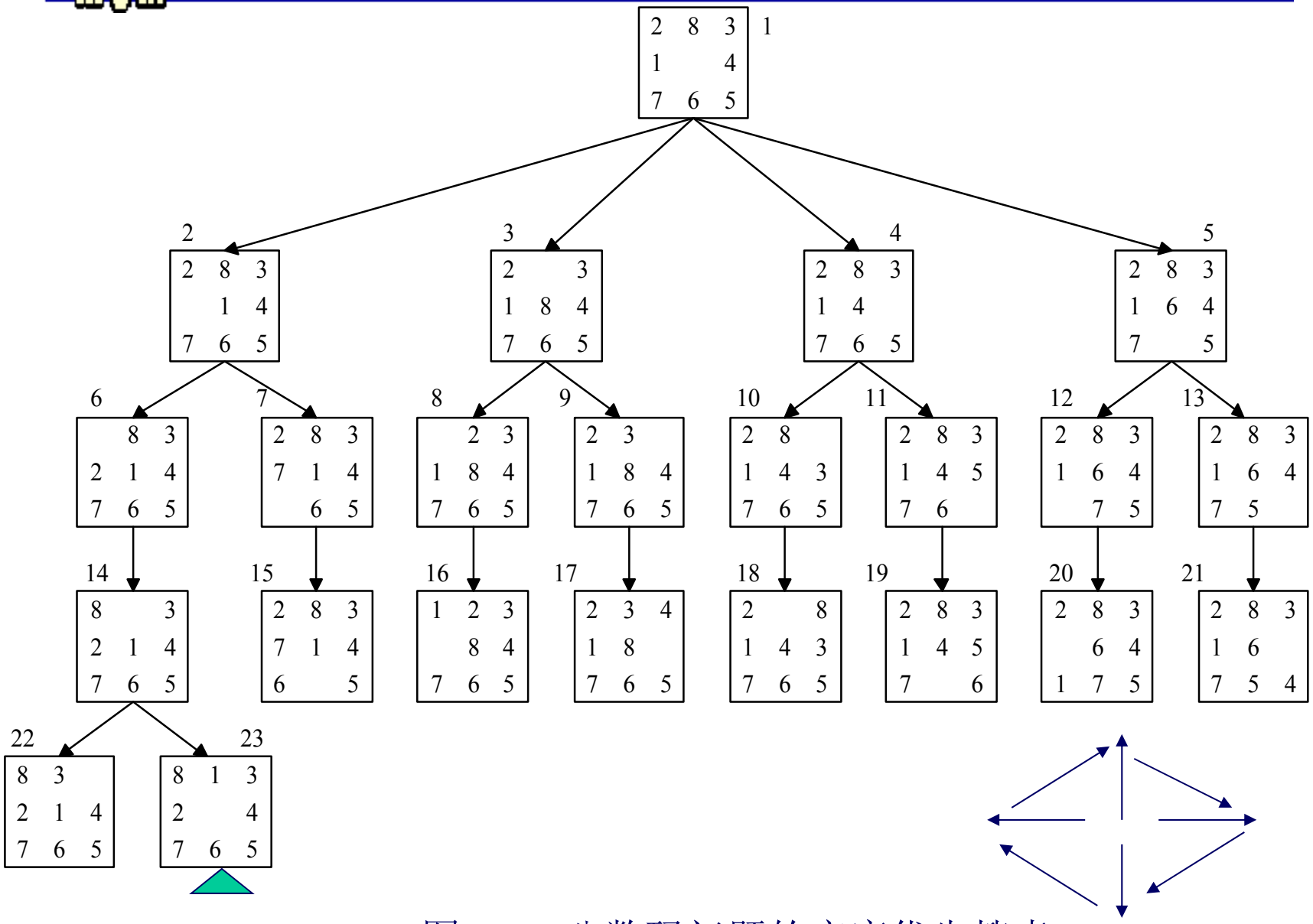

图4—5　八数码问题的广度优先搜索

2.深度优先搜索

深度优先搜索就是在搜索树的每一层始终先只扩展一个子节点，不断地向纵深前进，直到不能再前进(到达叶子节点或受到深度限制)时，才从当前节点返回到上一级节点，沿另一方向又继续前进。这种方法的搜索树是从树根开始一枝一枝逐渐形成的。

深度优先搜索算法：

步1 把初始节点S_0放入OPEN表中；

步2 若OPEN表为空，则搜索失败，退出。

步3　取OPEN表中前面第一个节点N放入CLOSED表中，并冠以顺序编号n；

步4 若目标节点S_g=N，则搜索成功，结束。

步5 若N不可扩展，则转步2；

步6　扩展N，将其所有子节点配上指向N的返回指针依次放入OPEN表的首部，转步2。

其中的OPEN表是一个栈，这是与宽度优先搜索算法的唯一区别。

例4.4 对于八数码问题，应用深度优先搜索策略，可得如图4–6所示的搜索树。

深度优先搜索亦称为纵向搜索。由于一个有解的问题树可能含有无穷分枝，深度优先搜索如果误入无穷分枝(即深度无限)，则不可能找到目标节点。所以，深度优先搜索策略是不完备的。另外，应用此策略得到的解不一定是最佳解(最短路径)。

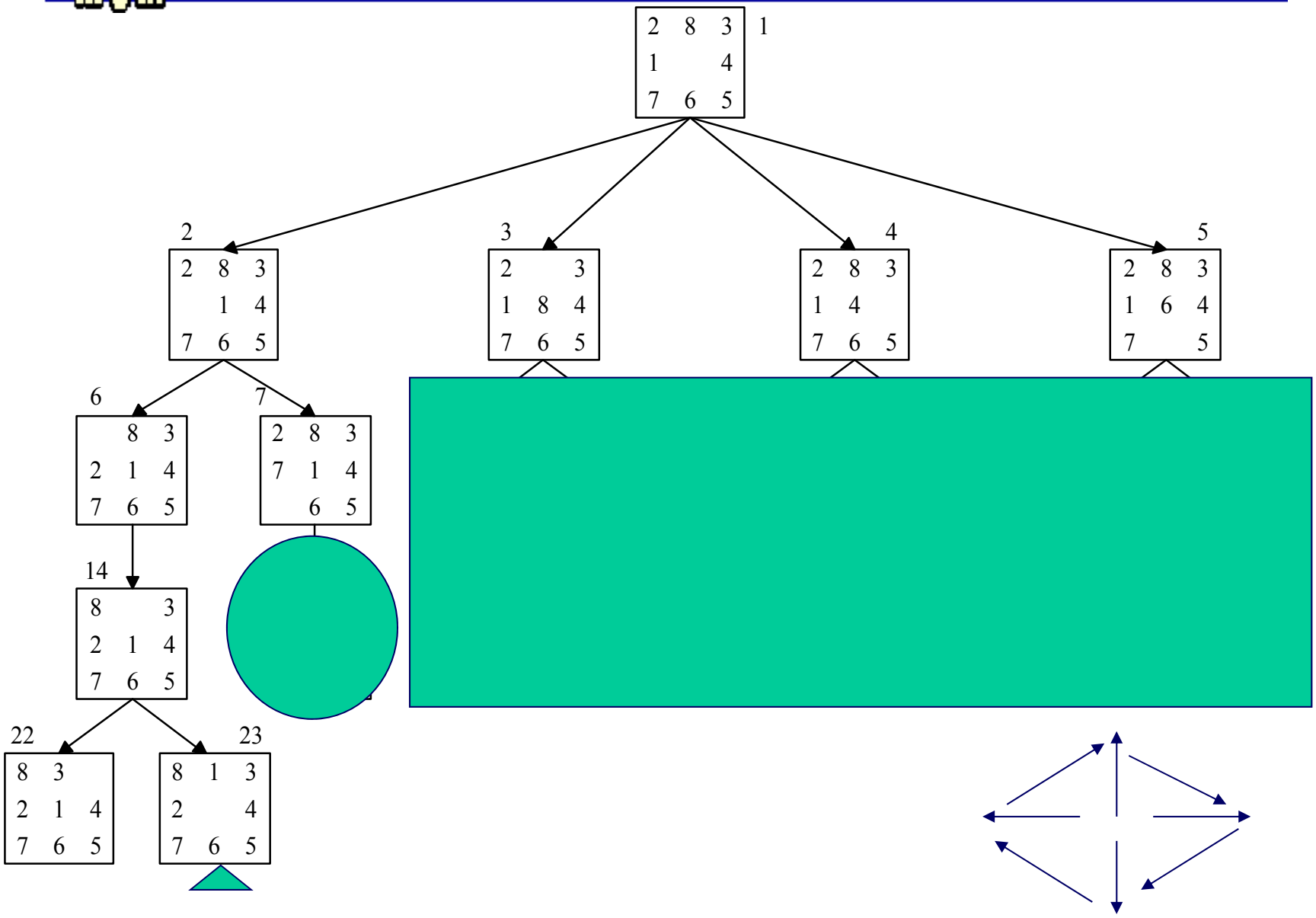

图4—6 —1 八数码问题的深度优先搜索

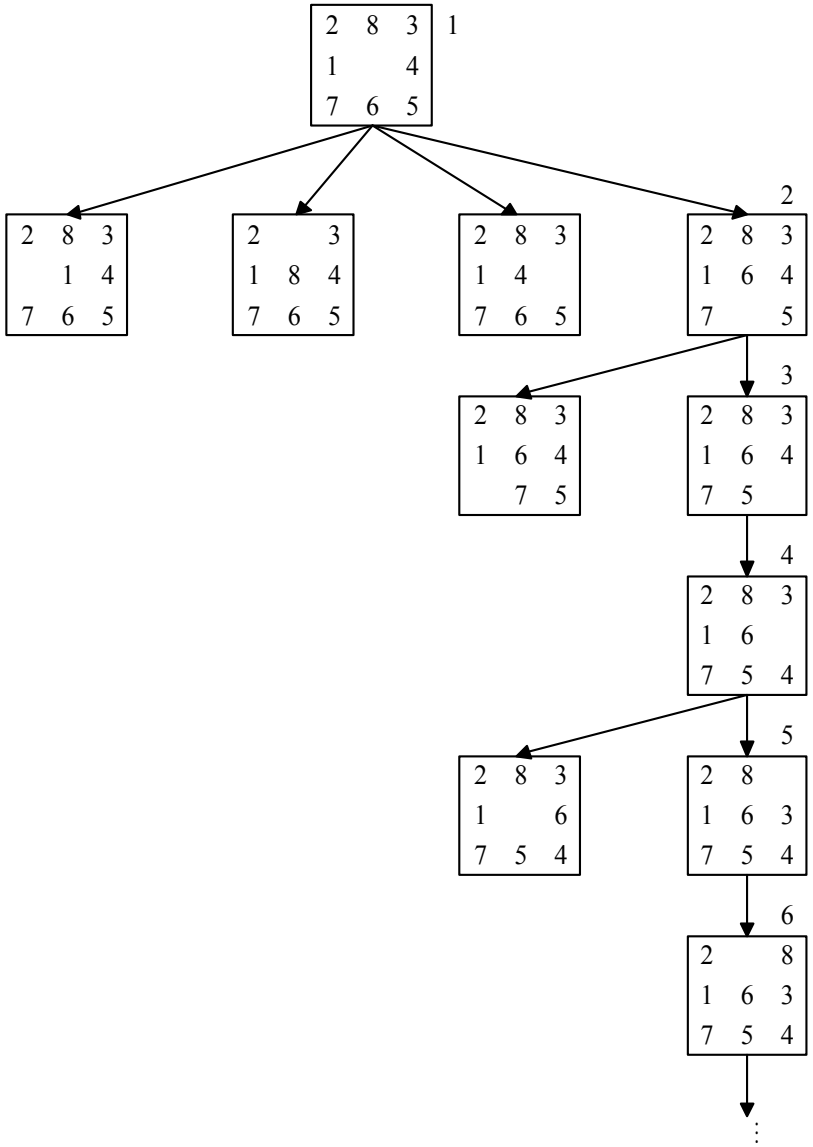

图4—6 —2 八数码问题的深度优先搜索

3.有界深度优先搜索

广度优先和深度优先是两种最基本的穷举搜索方法，在此基础上，根据需要再加上一定的限制条件，便可派生出许多特殊的搜索方法。例如有界深度优先搜索。

有界深度优先搜索就是给出了搜索树深度限制，当从初始节点出发沿某一分枝扩展到一限定深度时，就不能再继续向下扩展，而只能改变方向继续搜索。节点x的深度(即其位于搜索树的层数)通常用d(x)表示，则有界深度优先搜索算法如下：

步1 把S_0放入OPEN表中，置S_0的深度$d(S_0)=0$；

步2 若OPEN表为空，则失败，退出。

步3 取OPEN表中前面第一个节点N，放入CLOSED表中，并冠以顺序编号n；

步4 若目标节点S_g=N，则成功，结束。

步5 若N的深度d(N)=dm(深度限制值)，或者若N无子节点，则转步2；

步6 扩展N，将其所有子节点N_i配上指向N的返回指针后依次放入OPEN表中前部，置$d(N_i)=d(N)+1$，转步2。

4.1.4 启发式搜索

1．问题的提出

前面我们讲的穷举搜索法，从理论上讲，似乎可以解决任何状态空间的搜索问题，但实践表明，穷举搜索只能解决一些状态空间很小的简单问题，而对于那些大状态空间问题，穷举搜索就不能胜任了。因为大空间问题，往往会导致“组合爆炸”。 例如梵塔问题、国际象棋、围棋。梵塔问题中当圆盘数取64时，状态空间有3^{64}个节点，现有的任何计算机已经无法存放路径。可能的棋局，国际象棋是10^{120}，围棋是10^{761}。国际象棋的算法要10^{16}年才能算完。

2. 启发性信息

启发式搜索就是利用启发性信息进行制导的搜索。启发性信息就是有利于尽快找到问题之解的信息。按其用途划分，启发性信息一般可分为以下三类：

(1)用于扩展节点的选择，即用于决定应先扩展哪一个节点，以免盲目扩展。

(2)用于生成节点的选择，即用于决定应生成哪些后续节点，以免盲目地生成过多无用节点。

(3)用于删除节点的选择，即用于决定应删除哪些无用节点，以免造成进一步的时空浪费。

不同的问题有不同的启发性信息。

例如：八数码问题的部分状态图可以看出，从初始节点开始，在通向目标节点的路径上，各节点的数码格局同目标节点相比较，其数码不同的位置个数在逐渐减少，最后为0。所以，这个数码不同的位置个数便是标志一个节点到目标节点距离远近的一个启发性信息，利用这个信息就可以指导搜索。

3. 启发函数

在启发式搜索中，通常用所谓启发函数来表示启发性信息。启发函数是用来估计搜索树上节点x与目标节点Sg接近程度的一种函数，通常记为h(x)。

如何定义一个启发函数呢？启发函数并无固定的模式，需要具体问题具体分析。通常可以参考的思路有：一个节点到目标节点的某种距离或差异的度量；一个节点处在最佳路径上的概率；或者根据经验的主观打分等等。例如，对于八数码难题，用h(x)就可表示节点x的数码格局同目标节点相比，数码不同的位置个数。

4.启发式搜索算法

启发式搜索要用启发函数来导航，其搜索算法就要在状态图一般搜索算法基础上再增加启发函数值的计算与传播过程，并且由启发函数值来确定节点的扩展顺序。为简单起见，下面我们仅给出树型图的树式搜索的两种策略。

1) 全局择优搜索

全局择优搜索就是利用启发函数制导的一种启发式搜索方法。该方法亦称为最好优先搜索法，它的基本思想是：在OPEN表中保留所有已生成而未考察的节点，并用启发函数h(x)对它们全部进行估价，从中选出最优节点进行扩展，而不管这个节点出现在搜索树的什么地方。

全局择优搜索算法如下：

步1 把初始节点S_0放入OPEN表中，计算$h(S_o)$；

步2 若OPEN表为空，则搜索失败，退出；

步3 移出OPEN表中第一个节点N放入CLOSED表中，并冠以序号n；

步4 若目标节点S_g=N，则搜索成功，结束；

步5 若N不可扩展，则转步2；

步6 扩展N，计算每个子节点x的函数值h(x)，并将所有子节点配以指向N的返回指针后放入OPEN表中，再对OPEN表中的所有子节点按其函数值大小以升序排序，转步2。

例4.5 用全局择优搜索法解八数码难题。初始棋局和目标棋局同例3。

解 设启发函数h(x)为节点x的格局与目标格局相比数码不同的位置个数。以这个函数制导的搜索树如图4–7所示。图中节点旁的数字就是该节点的估价值。由图可见此八数问题的解为：

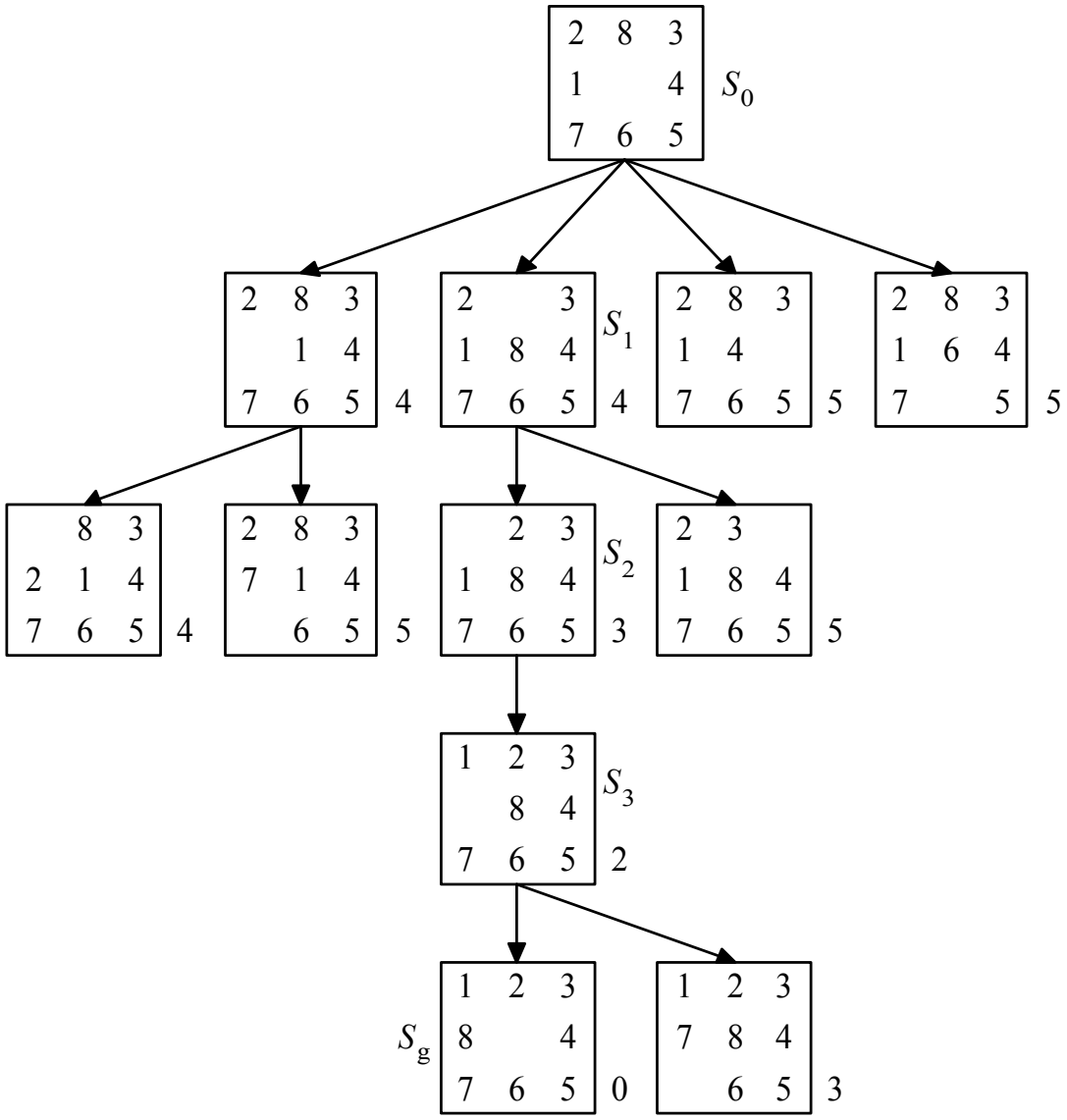

图4—7　八数码问题的全局择优搜索

2) 局部择优搜索

局部择优搜索与全局择优搜索的区别是，扩展节点N后仅对N的子节点按启发函数值大小以升序排序，再将它们依次放入OPEN表的首部。故算法从略。

4.1.5 加权状态图搜索

1. 加权状态图与代价树

例4.6 图4–8(a)是一个交通图，设A城是出发地，E城是目的地，边上的数字代表两城之间的交通费。试求从A到E最小费用的旅行路线。

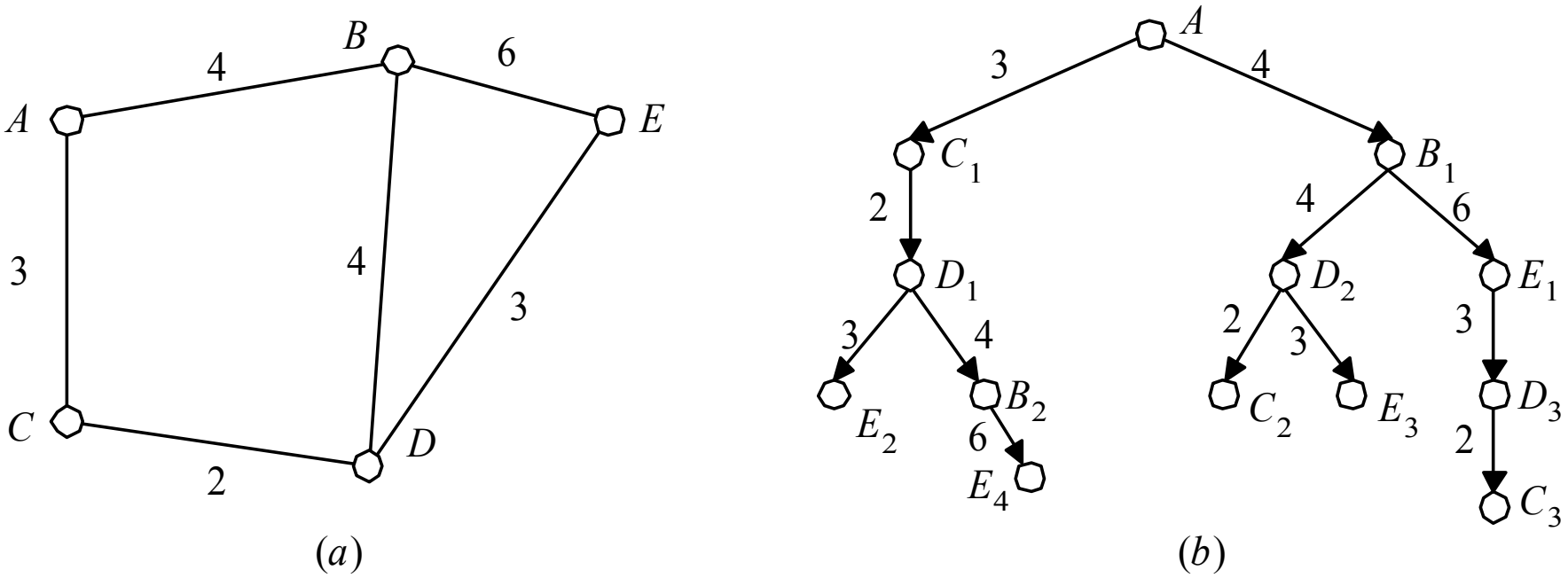

图4—8 交通图及其代价树

可以看出，这个图与前面的状态图不同的地方是边上附有数值。它表示边的一种度量（此例中是交通费，当然也可以是距离）。一般地，称这种数值为**权值**，而把边上附有数值的状态图称之为**加权状态图**或**赋权状态图**。

显然，加权状态图的搜索与权值有关，并且要用权值来导航。具体来讲，加权状态图的搜索算法，要在一般状态图搜索算法基础上再增加权值的计算与传播过程，并且要由权值来确定节点的扩展顺序。

同样，为简单起见，我们先考虑树型的加权状态图——代价树的搜索。所谓**代价**，可以是两点之间的距离、交通费用或所需时间等等。通常用$g(x)$表示从初始节点S_0到节点x的代价，用$c(x_i,x_j)$表示父节点x_i到子节点x_j的代价，即边(x_i,x_j)的代价。从而有

$$g(x_j)=g(x_i)+c(xi,xj)$$

而 $g(S_0)=0$

也可以将加权状态图转换成代价树来搜索，其转换方法是，从初始节点起，先把每一个与初始节点相邻的节点作为该节点的子节点；然后对其他节点依次类推，但对其他节点x，**不能将其父节点及祖先再作为x的子节点**。例如，把图4–8(a)所示的交通图转换成代价树如图4—8(b)所示。

下面介绍两种代价树的搜索策略，即分支界限法和最近优先法。

2.分支界限法(最小代价优先法)

其基本思想是：每次从OPEN表中选出g(x)值最小的节点进行考察，而不管这个节点在搜索树的什么位置上。

可以看出，这种搜索法与前面的最好优先法（即全局择优法）的区别仅是选取扩展节点的标准不同，一个是代价值g(x)（最小），一个是启发函数值h(x)（最小）。这就是说，把最好优先法算法中的h(x)换成g(x)即得分支界限法的算法。

3.最近择优法(瞎子爬山法)

同上面的情形一样，这种方法实际同局部择优法类似，区别也仅是选取扩展节点的标准不同，一个是代价值g(x)（最小），一个是启发函数值h(x)（最小）。这就是说，把局部择优法算法中的h(x)换成g(x)就可得最近择优法的算法。

现在我们用代价树搜索求解例4.6中给出的问题。我们用分支界限法得到的路径为

$$A \rightarrow C \rightarrow D \rightarrow E$$

这是一条最小费用路径(费用为8)。

4.1.6 启发式图搜索的A算法和A*算法

前面我们介绍了图搜索的一般算法，并着重讨论了树型图的各种搜索策略。本节我们给出图搜索的两种典型的启发式搜索算法。

1.估价函数

利用启发函数h(x)制导的启发式搜索，实际是一种深度优先的搜索策略。虽然它是很高效的，但也可能误入歧途。

所以，为了更稳妥一些，人们把启发函数扩充为估价函数。估价函数的一般形式为

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

其中g(x)为从初始节点S_0到节点x已经付出的代价，h(x)是启发函数。即估价函数f(x)是从初始节点S_0到达节点x处已付出的代价与节点x到达目标节点S_g的接近程度估计值之总和。

有时估价函数还可以表示为

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

d(x)表示节点x的深度。

可以看出，　f(x)中的g(x)或d(x)有利于搜索的横向发展，因而可提高搜索的完备性，但影响效率；h(x)则有利于搜索的纵向发展，因而可提高搜索的效率，但影响完备性。所以，f(x)是二者的一个折中。但在确定f(x)时，要权衡利弊，使g(x)（或d(x)）与h(x)比例适当。

如果把h(x)取为节点x到目标节点S_g的估计代价，则f(x)就是已知代价与未知代价之和。这时基于f(x)的搜索就是最小代价搜索。

2. A算法

A算法是基于估价函数f(x)的一种加权状态图启发式搜索算法。其具体步骤如下：

步1 把附有$f(S_0)$的初始节点S_0放入OPEN表；

步2 若OPEN表为空，则搜索失败，退出。

步3 移出OPEN表中第一个节点N放入CLOSED表中，并冠以顺序编号n；

步4 若目标节点S_g=N，则搜索成功，结束。

步5 若N不可扩展，则转步2；

步6 扩展N，生成一组附有f(x)的子节点，对这组节点作如下处理：

（1）考察是否有已在OPEN表或ClOSED表中存在的节点；若有，则再考虑其中有无N的先辈节点，若有则删除之；对于其余节点，也删除之，但由于他们又被第二次生成，因而需要考虑是否修改已经存在于OPEN表或CLOSED表中的这些节点及其后裔的返回指针和f(x)值，修改原则是“抄f(x)值最小的路走”；

（2）对其余子节点配上指向N的返回指针后放入OPEN表中，并对OPEN表按f(x)以升序排序，转步2。

算法中节点x的估价函数f(x)的计算方法是

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

$$=g(x_i)+c(x_i,\ x_j)+h(x_j)\quad (x_j\text{是}x_i\text{的子节点})$$

3. A*算法

如果对上述A算法再限制其估价函数中的启发函数h(x)满足：对所有的节点x均有

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

其中$h^*(x)$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)，则它就称为A*算法。

A*算法中，限制$h(x) \leq h^*(x)$的原因是为了保证取得最优解。理论分析表明，如果问题存在最优解，则这样的限制就可保证找到最优解。

A*算法也称为最佳图搜索算法，它是著名的人工智能学者Nilsson提出的。

4.1.7 状态图搜索策略小结

上述的状态图搜索策略可归纳如下：

- 树式搜索
  - 盲目搜索
    - 穷举式
      - 广度优先
      - 深度优先
        - 有界深度优先
  - 启发式搜索
    - 全局择优（最好优先，基于启发函数h(x)）
    - 局部择优（瞎子爬山，基于h(x)）
    - 分支界限（全局最小代价优先，基于代价函数g(x)）
    - 最近择优（瞎子爬山，局部最小代价优先，基于g(x)）
    - A算法（基于估价函数f(x)＝g(x)＋h(x)）
      - A*算法（最佳图搜索，h(x)≤h*(x)）

- 线式搜索
  - 盲目搜索
    - 随机碰撞
    - 回溯穷举（生成–测试法）
  - 启发式搜索
    - 不回溯（瞎子爬山，基于h(x)、g(x)或f(x)）
    - 智能回溯（启发式生成—测试法）

4.2 状态图问题求解

许多实际问题（如规划、设计、诊断、控制、预测、决策、证明等等） 都可以表示（或归结）为状态图搜索问题。旅行商问题、机器人行动规划、定理证明和故障诊断都是图搜索问题。

4.2.1 问题的状态图表示

1.状态

状态就是问题在任一确定时刻的状况，它表征了问题特征和结构等。状态在状态图中表示为节点。状态一般用一组数据表示。在程序中用字符、数字、记录、数组、结构、对象等表示。

2.状态转换规则

状态转换规则就是能使问题状态改变的某种操作、规则、行为、变换、关系、函数、算子、过程等等。状态转换规则也称为操作，问题的状态也只能经定义在其上的这种操作而改变。状态转换规则在状态图中表示为边。在程序中状态转换规则可用数据对、条件语句、规则、函数、过程等表示。

3. 状态图表示

一个问题的状态图是一个三元组

（S,F,G）

其中S是问题的初始状态集合，F是问题的状态转换规则集合，G是问题的目标状态集合。

一个问题的全体状态及其关系，就构成一个空间，称为状态空间。所以，状态图也称为状态空间图。

例4.7 迷宫问题的状态图表示。

我们仍以例4.1中的迷宫为例。我们以每个格子作为一个状态，并用其标识符作为其表示。那么，两个标识符组成的序对就是一个状态转换规则。于是，该迷宫的状态图表示为

S：S_0

F：{(S_0,S_4),(S_4, S_0)，(S_4,S_1),(S_1, S_4)，(S_1,S_2),(S_2, S_1)，(S_2,S_3),(S_3, S_2),

(S_4,S_7),(S_7, S_4)，(S_4,S_5),(S_5, S_4)，(S_5,S_6),(S_6, S_5)，(S_5,S_8),(S_8, S_5),

(S_8,S_9),(S_9, S_8)，(S_9,S_g)}

G：S_g

这种列出全部节点和边的状态图称为显示状态图，或状态图的显示表示。

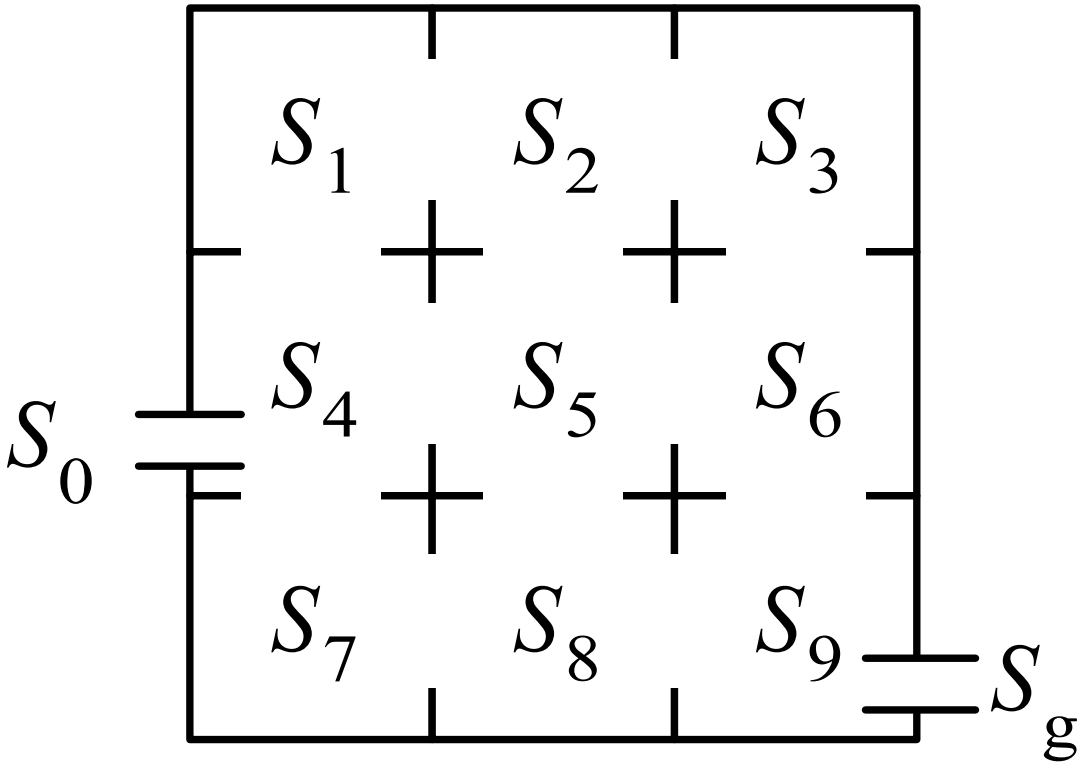

图4—1 迷宫图

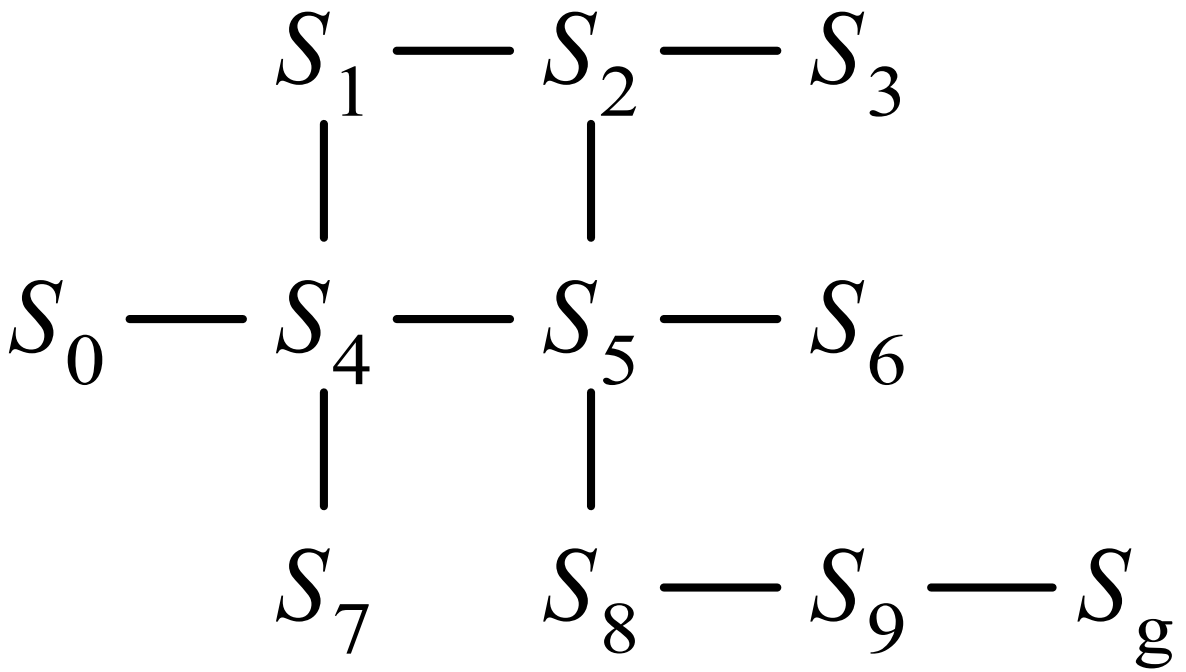

图4—2 迷宫的有向图表示

例4.8　八数码难题的状态图表示。

我们将棋局

X_1	X_2	X_3
X_6	X_0	X_4
X_7	X_6	X_5

用向量

$$A=(X_0,X_1,X_2,X_3,X_4,X_5,X_6,X_7,X_8)$$

表示，X_i为变量，X_i的值就是方格X_i内的数字。于是，向量 A 就是该问题的状态表达式。

设初始状态和目标状态分别为

$$S_0=(0,2,8,3,4,5,6,7,1)$$

$$S_g=(0,1,2,3,4,5,6,7,8)$$

易见，数码的移动规则就是该问题的状态变换规则，即操作。经分析，该问题共有24条移码规则，可分为9组。

0组规则：

$r_1(X_0=0)\wedge(X_2=n)\rightarrow X_0\Leftarrow n\wedge X_2\Leftarrow 0;$

$r_2(X_0=0)\wedge(X_4=n)\rightarrow X_0\Leftarrow n\wedge X_4\Leftarrow 0;$

$r_3(X_0=0)\wedge(X_6=n)\rightarrow X_0\Leftarrow n\wedge X_6\Leftarrow 0;$

$r_4(X_0=0)\wedge(X_8=n)\rightarrow X_0\Leftarrow n\wedge X_8\Leftarrow 0;$

1组规则：

$r_5(X_1=0)\wedge(X_2=n)\rightarrow X_1\Leftarrow n\wedge X_2\Leftarrow 0;$

$r_6(X_1=0)\wedge(X_8=n)\rightarrow X_1\Leftarrow n\wedge X_8\Leftarrow 0;$

2组规则：

$r_7(X_2=0)\wedge(X_1=n)\rightarrow X_2\Leftarrow n\wedge X_1\Leftarrow 0;$

$r_8(X_2=0)\wedge(X_3=n)\rightarrow X_2\Leftarrow n\wedge X_3\Leftarrow 0;$

$r_9(X_2=0)\wedge(X_0=n)\rightarrow X_2\Leftarrow n\wedge X_0\Leftarrow 0;$

...

8组规则：

$r_{22}(X_8=0)\wedge(X_1=n)\rightarrow X_8\Leftarrow n\wedge X_1\Leftarrow 0;$

$r_{23}(X8=0)\wedge(X_0=n)\rightarrow X_8\Leftarrow n\wedge X_0\Leftarrow 0;$

$r_{24}(X8=0)\wedge(X_7=n)\rightarrow X_8\Leftarrow n\wedge X_7\Leftarrow 0;$

于是，八数码问题的状态图可表示为

$$(\{S_0\},\ \{r_1,r_2,...,r_{24}\ \},\{S_g\})$$

当然，上述24条规则也可以简化为4条：即空格上移、下移、左移、右移。不过，这时状态(即棋局)就需要用矩阵来表示。

可以看出，这个状态图中仅给出了初始节点和目标节点，并未给出其余节点。而其余节点需用状态转换规则来产生。类似于这样表示的状态图称为**隐式状态图**，或者说**状态图的隐式表示**。

例4.9　梵塔问题。传说在印度的贝那勒斯的圣庙中，主神梵天做了一个由64个大小不同的金盘组成的“梵塔”，并把它穿在一个宝石杆上。另外，旁边再插上两个宝石杆。然后，他要求憎侣们把穿在第一个宝石杆上的64个金盘全部搬到第三个宝石杆上。搬动金盘的规则是：一次只能搬一个；不允许将较大的盘子放在较小的盘子上。于是，梵天预言：一旦64个盘子都搬到了3号杆上，世界将在一声霹雳中毁灭。这就是梵塔问题。

经计算，把64个盘子全部搬到3号杆上，需要穿插搬动盘子

$2^{64}-1=18\ 446\ 744\ 073\ 709\ 511\ 615$

次。所以直接考虑原问题，将过于复杂。

为了便于分析，我们仅考虑二阶梵塔（即只有两个金盘）问题。设有三根宝石杆，在1号杆上穿有A、B两个金盘，A小于B，A位于B的上面。要求把这两个金盘全部移到另一根杆上，而且规定每次只能移动一个盘子，任何时刻都不能使B位于A的上面。

设用二元组(S_A,S_B)表示问题的状态，S_A表示金盘A所在的杆号，S_B表示金盘B所在的杆号，这样，全部可能的状态有9种，可表示如下：

(1,1),(1,2),(1,3)

(2,1),(2,2),(2,3)

(3,1),(3,2),(3,3)

如图4—9所示。

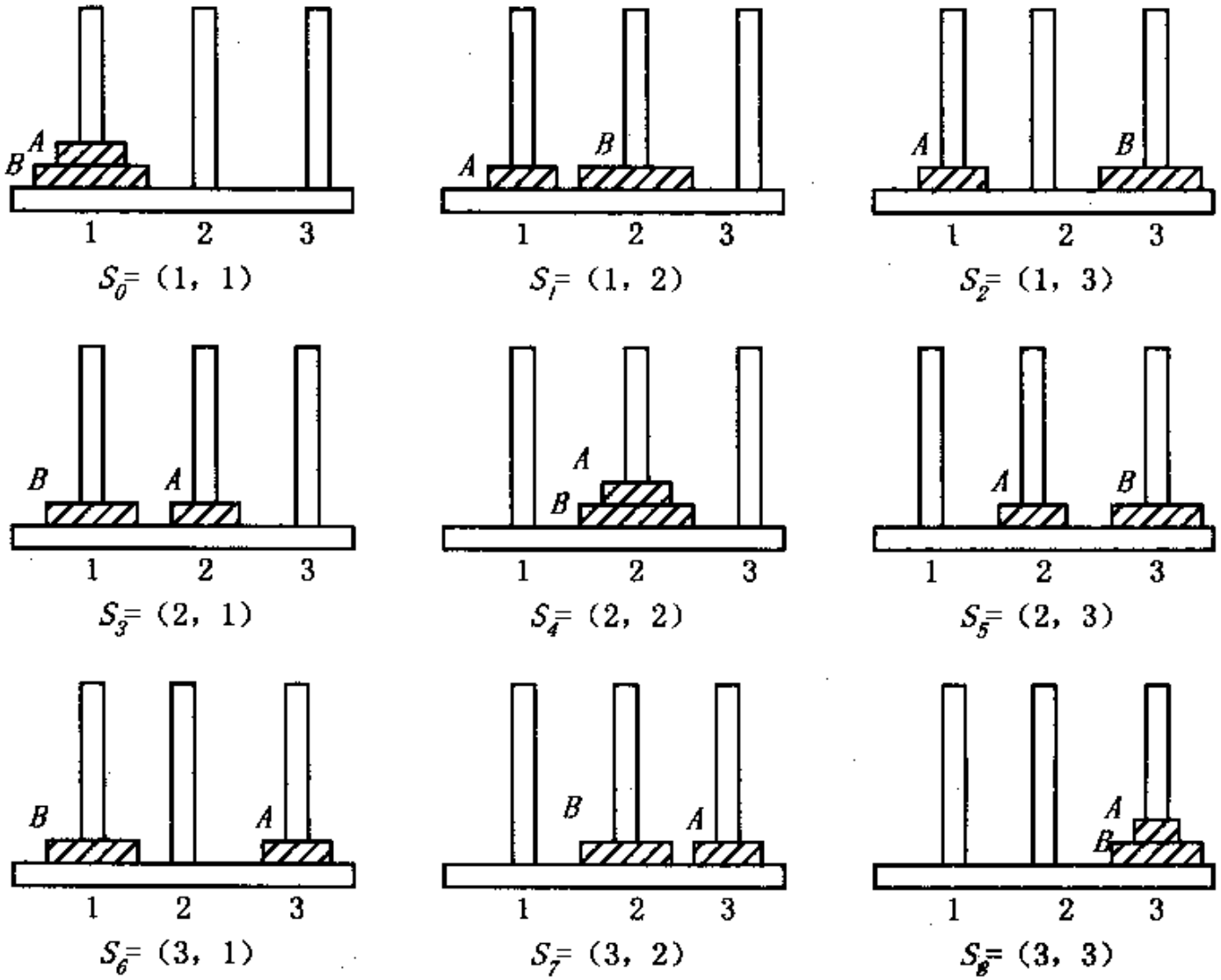

图4–9　二阶梵塔的全部状态

这里的状态转换规则就是金盘的搬动规则，分别用A(*i*,*j*)及B(*i*,*j*)表示：A(*i*,*j*)表示把A盘从第i号杆移到第j号杆上;B(*i*,*j*)表示把B盘从第i号杆移到第j号杆上。经分析，共有12个操作，它们分别是：

A(1,2),A(1,3),A(2,1),A(2,3),A(3,1),A(3,2)

B(1,2),B(1,3),B(2,1),B(2,3),B(3,1),B(3,2)

当然，规则的具体形式应是：

IF<条件>THENA(i,j)

IF<条件>THENB(i,j)

(条件留给读者完成)

这样由题意，问题的初始状态为(1,1)，目标状态为(3,3)。则二阶梵塔问题可用状态图表示为

（{(1,1)},{A(1,2),...,B(3,2)},{(3,3)}）

由这9种可能的状态和12种操作，二阶梵塔问题的状态空间图如图4—10所示。

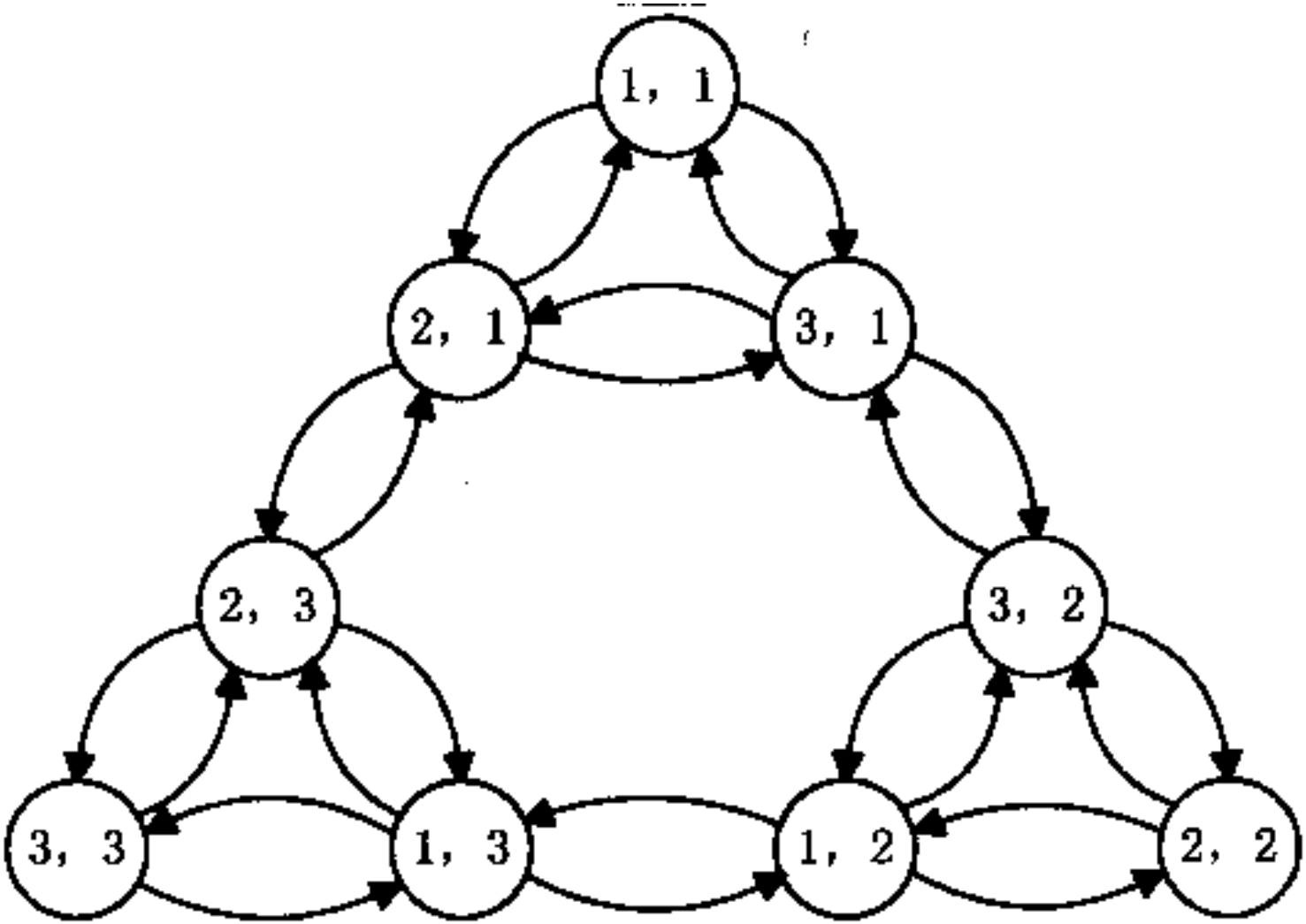

图4—10　二阶梵塔状态空间图

例4.10 旅行商问题(Traveling　Salesman Problem，简称TSP)。设有n个互相可直达的城市，某推销商准备从其中的A城出发，周游各城市一遍，最后又回到A城。要求为该推销商规划一条最短的旅行路线。该问题的状态为以A打头的已访问过的城市序列:A...

S_0:A。

S_g:A，...,A。其中“...”为其余n-1个城市的一个序列。

状态转换规则：

规则1 如果当前城市的下一个城市还未去过，则去该城市，并把该城市名排在已去过的城市名序列后端。

规则2 如果所有城市都去过一次，则从当前城市返回A城，把A也添在去过的城市名序列后端。

4.2.2 状态图问题求解程序举例

例4.11　下面是一个通用的状态图搜索程序。对于求解的具体问题，只需将其状态图的程序表示并入该程序即可。

```
/*状态图搜索通用程序*/
DOMAINS
state=<领域说明> %例如:state=symbol
 DATABASE-mydatabase
```

```
open(state,integer)        %用动态数据库实现OPEN表
closed(integer,state,integer)%和CLOSED表
res(state)
open1(state,integer)
min(state,integer)
mark(state)
fail-
```

PREDICATES

solve

search(state,state)

result

searching

step4(integer,state)

step56(integer,state)

equal(state,state)

repeat

resulting(integer)

rule(state,state)

```
GOAL
 solve.
CLAUSES
 solve：-search(<初始状态>，<目标状态>),result.
/*例如
 solve：-
search(st(0,1,2,3,4,5,6,7,8),st(0,2,8,3,4,5,6,7,1)),result.
 */
search(Begin,End)：-%搜索
 retractall(-,mydatabase),
 assert(closed(0,Begin,0)),
```

```
assert(open(Begin,0)),%步1将初始节点放入OPEN表
 assert(mark(End))，
 repeat，
 searching，!.
result：-%输出解
 not(fail-)，
 retract(closed(0，-,0)),
 closed(M，-，-)，
 resulting(M)，!.
result：-beep,write("sorry don't find a road!").
```

searching：-

open(State,Pointer),%步2若OPEN表空,则失败,退出

retract(open(State,Pointer)),%步3取出OPEN表中第一个节点,给其

closed(No，　-，　-),No2=No+1,%编号

asserta(closed(No2,State,Pointer)),%放入CLOSED表

!,step4(No2,State).

searching：-assert(fail-). %步4若当前节点为目标节点,则成功

step4(-,State)：-mark(End),equal(State,End). %转步2

step4(No,State)：-step56(No,State),!,fail.

step56(No,State X)：-　　%步5若当前节点不可扩展,转步2

rule(State X,State Y)，%步6扩展当前节点X得Y

not(open(State Y，-))，%考察Y是否已在OPEN表中

not(closed(-,State Y，-))，%考察Y是否已在CLOSED表中

assertz (open(State Y,No)),%可改变搜索策略

fail.

```
step56(-，-)：-!.

equal(X,X).

repeat.

repeat：-repeat.

resulting(N)：-closed(N,X,M),assert a (res (X)),resulting(M).

resulting(-)：-res (X),write(X),n l,fail.

resulting(-)：-!.

rule(X,Y)：-<问题中的状态转换规则>.%例如:rule(X,Y):-
    road(X,Y).
```

例4.12 迷宫问题程序。下面仅给出初始状态、目标状态和状态转换规则集，程序用例4.11的通用程序。

```
DOMAINS
    State=symbol
GOAL
    search(a,e).
CLAUSES
/*把该问题的状态转换规则挂接在通用程序的规则上*/
    rule(X,Y)：-road(X,Y).
```

/*下面是该问题的状态转换规则(其实也就是迷宫图)*/

road(a,b).road(a,c).road(b,f).road(f,g).road(f,ff).road(g,h).

road(g,i).road(b,d).road(c,d).road(d,e).road(e,b).

例4.13　八数码问题程序。我们把前面给出的该问题的状态图表示，用PROLOG语言翻译如下，搜索程序用例4.11的通用程序。

```
DOMAINS
state=st (integer,integer,integer,integer,integer,
integer,integer,integer,integer)
GOAL
search(st(0,1,2,3,4,5,6,7,8),st(0,2,8,3,4,5,6,7,1)).
 CLAUSES
rule(X,Y)：-rule1(X,Y)./* 把该问题的状态转换规则挂接在通用程序的规则上*/
```

/*下面是该问题的状态转换规则(即走步规则)*/

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X2,X1,X0,X3,X4,X5,X6,X7,X8))：-X0=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X4,X1,X2,X3,X0,X5,X6,X7,X8))：-X0=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X6,X1,X2,X3,X4,X5,X0,X7,X8))：-X0=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X8,X1,X2,X3,X4,X5,X6,X7,X0))：-X0=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X2,X1,X3,X4,X5,X6,X7,X8))：-X1=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X2,X8,X3,X4,X5,X6,X7,X1))：-X1=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X2,X1,X3,X4,X5,X6,X7,X8))：-X2=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X3,X2,X4,X5,X6,X7,X8))：-X2=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X2,X1,X0,X3,X4,X5,X6,X7,X8))：-X2=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X3,X2,X4,X5,X6,X7,X8))：-X3=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X4,X3,X5,X6,X7,X8))：-X3=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X4,X3,X5,X6,X7,X8))：-X4=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X4,X1,X2,X3,X0,X5,X6,X7,X8))：-X4=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X5,X4,X6,X7,X8))：-X4=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X5,X4,X6,X7,X8))：-X5=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X6,X5,X7,X8))：-X5=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X6,X1,X2,X3,X4,X5,X0,X7,X8))：-X6=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X6,X5,X7,X8))：-X6=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X5,X7,X6,X8))：-X6=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X5,X7,X6,X8))：-X7=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X5,X6,X8,X7))：-X7=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X8,X2,X3,X4,X5,X6,X7,X1))：-X8=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X8,X1,X2,X3,X4,X5,X6,X7,X0))：-X8=0.

rule1(st(X0,X1,X2,X3,X4,X5,X6,X7,X8),st(X0,X1,X2,X3,X4,X5,X6,X8,X7))：-X8=0.

例4.14 旅行商问题程序。

```
/*旅行商问题*/
DOMAINS
  State=st(lists,integer)
  lists=symbol*
  Gx,Grule,Fx=integer
  city1,city2=symbol
  distance=integer
  Starting City=symbol
  City Sum=integer
```

DATABASE-my data base

open(State,integer,G x,F x)

closed(integer,State,integer,G x)

open1(State,integer,integer,integer)

min(State,integer,integer,integer)

mark(string,integer)

min D(integer)

fail-

PREDICATES

road(city1,city2,distance)

search(Starting City,City Sum)

searching

step4(integer,State,G x)

step56(integer,State,G x)

calculator(integer,integer,integer,integer,integer)

repeat

sort

p1

p12(State,integer,integer,integer)

```
p2
rule(State,State,Grule )
member(symbol,lists)
append(lists,lists,lists)
min dist(integer)
mindist1
pa(integer)
result
GOAL
 clear window，
 write("Please in out starting city name：")，
 readln(Start)，
```

可以看出，该程序与例4.11的通用程序基本相同，但这是一个基于A*算法的启发式图搜索程序。估价函数f(x)为代价函数g(x)和启发函数h(x)之和。其中代价函数的计算公式为节点(A...XY)的代价＝起始城市到X城的代价＋X城到Y城的代价　其中的代价可以是距离、费用或时间等（下同）。

4.3 与或图搜索

4.3.1 与或图

我们仍用例子引入与或图的概念。

例4.15 如图4–11所示，设有四边形ABCD和A′B′C′D′，要求证明它们全等。

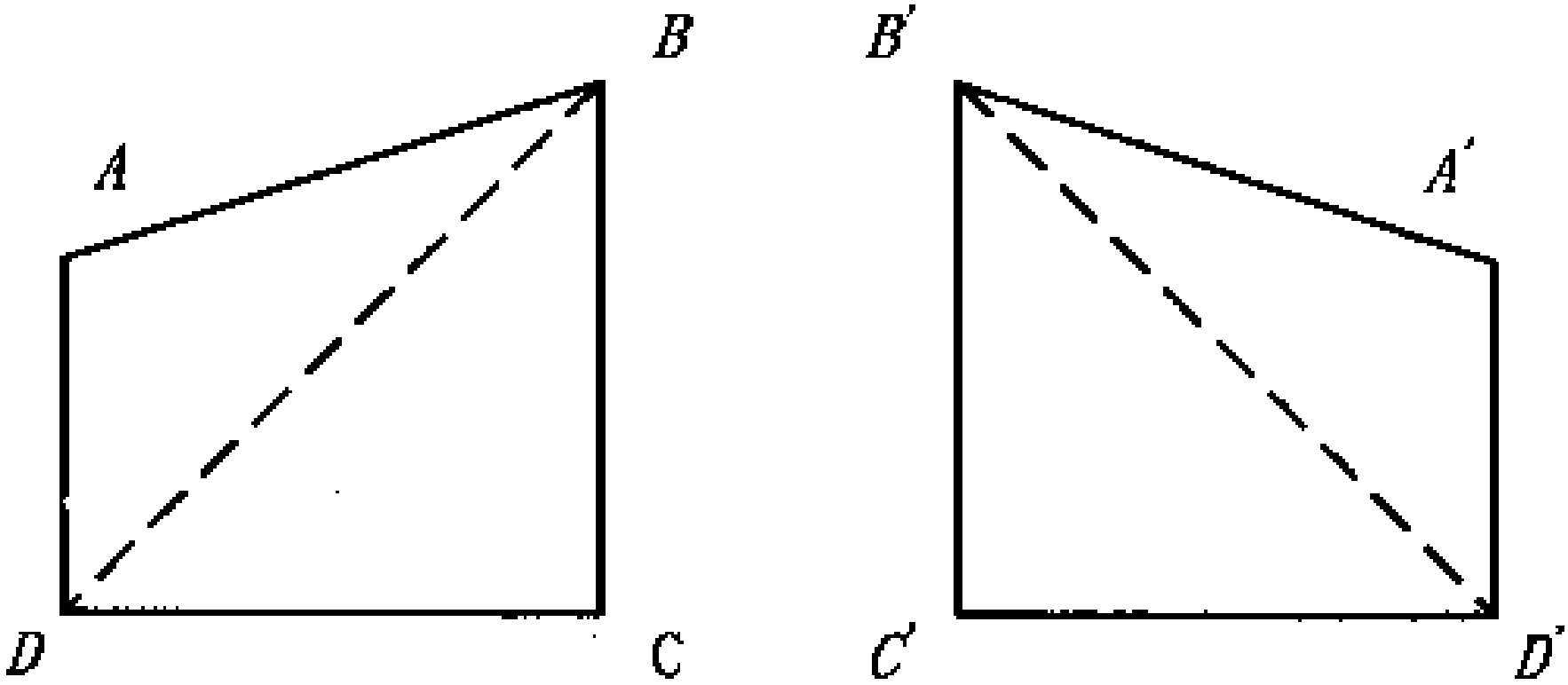

图4—11　四边形ABCD和A′B′C′D′

分析：分别连接B、D和B′、D′，则原问题可分解为两个子问题：

Q_1：证明△ABD≌△A′B′D′

Q_2：证明△BCD≌△B′C′D′

于是，原问题的解决可归结为这两个子问题的解决。换句话说，原问题被解决当且仅当这两个子问题都被解决。进一步，问题Q_1还可再被分解为

Q_{11}：证明AB＝A′B′

Q_{12}：证明AD＝A′D′

Q_{13}：证明∠A＝∠A′

或

$Q_{11'}$：证明AB＝A′B′

$Q_{12'}$：证明AD＝A′D′

$Q_{13'}$：证明BD＝B′D′

问题Q_2还可再被分解为

Q_{21}：证明BC＝B′C′

Q_{22}：证明CD＝C′D′

Q_{23}：证明∠C＝∠C′

或

$Q_{21'}$：证明BC＝B′C′

$Q_{22'}$：证明CD＝C′D′

$Q_{23'}$：证明BD＝B′D′

现在考虑原问题与这两组子问题的关系，我们便得到图4–12。图中的弧线表示所连边为“与”关系，不带弧线的边为或关系。这个图中既有与关系又有或关系，因此被称为**与或图**。但这个与或图是一种特殊的与或图，称为与或树。图4—13所示的则是一个典型的与或图。

可以看出，从与、或关系来看，前面的状态图，实际就是或图。这就是说，与或图是状态图的推广，而状态图是与或图的特例。

与或图可以用来描述许多问题，如梵塔问题、猴子摘香蕉问题、博弈问题、求不定积分问题、定理证明问题等等。所以研究与或图有普遍意义。

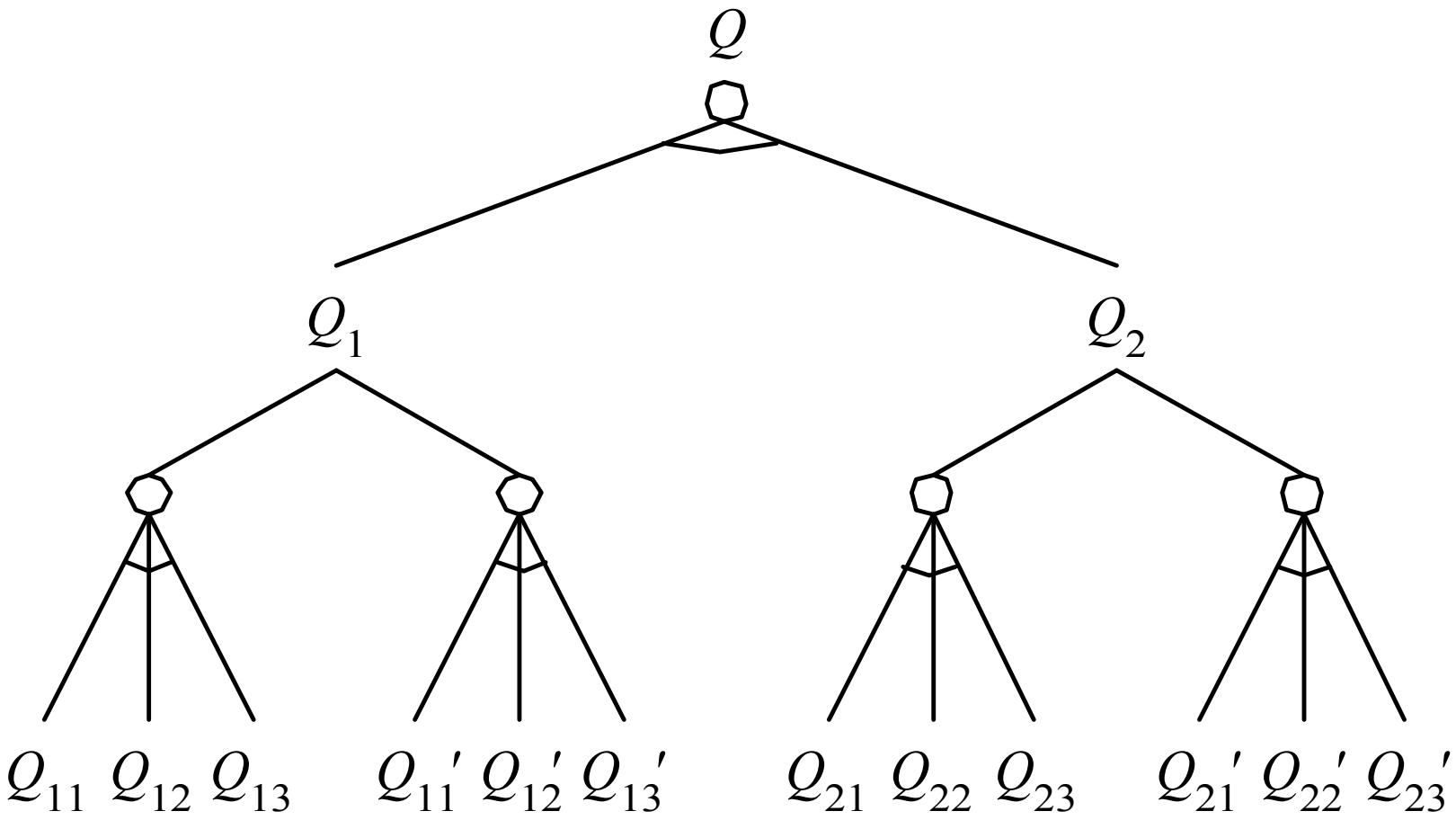

图4—12　问题的分解与变换

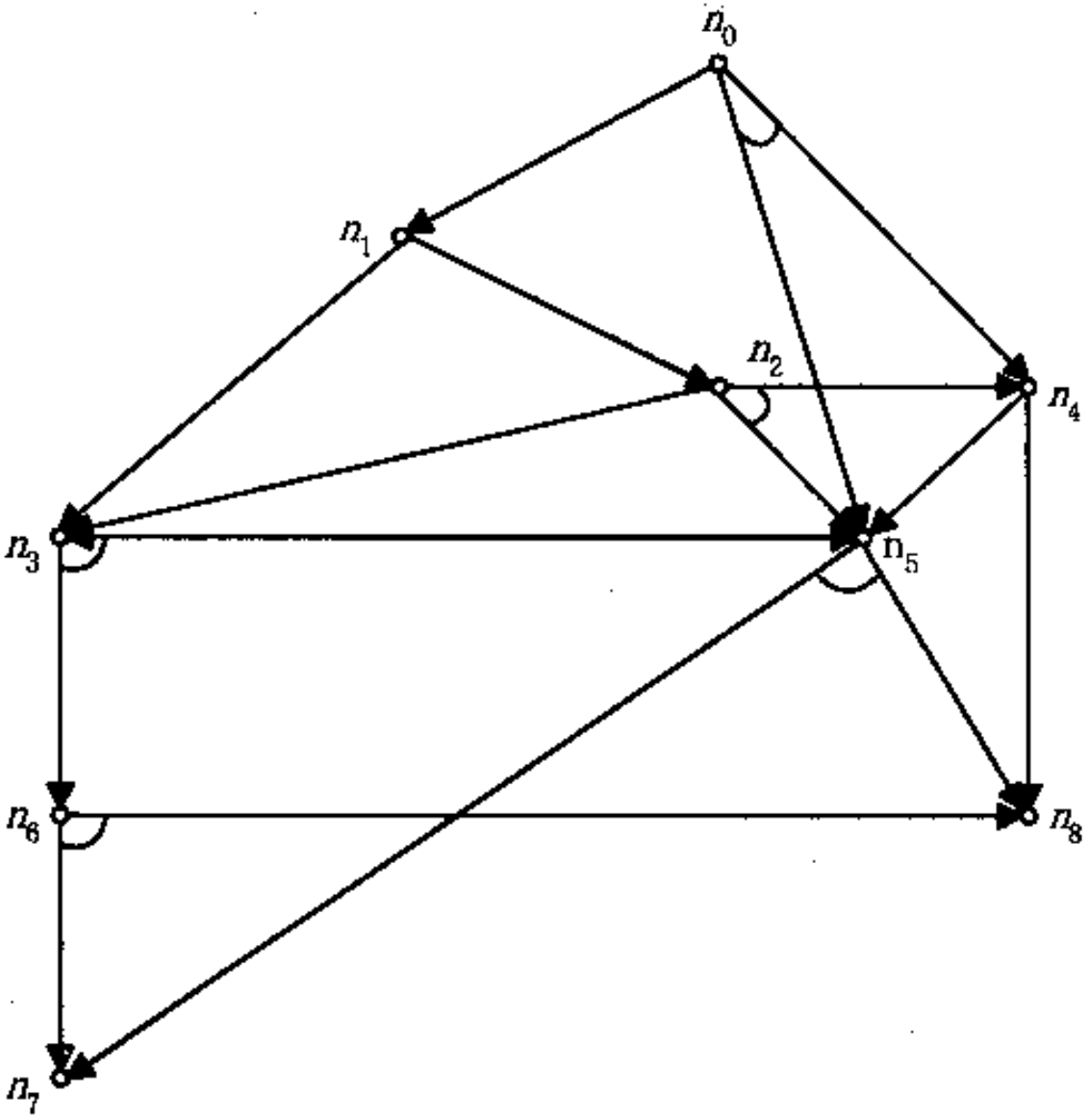

图4—13　一个典型的与或图

用与或图描述问题的求解过程，就是将原问题通过有关变换规则不断分解（为子问题）或变换（为等价问题），直到问题分解或变换为（归约为）一些直接可解的子问题，或者不可解也不能分解或变换为止。

为了叙述方便，下面引入一些新概念：直接可解的简单问题称为**本原问题**，本原问题对应的节点称为**终止节点**，在与或图（树）中无子节点的节点称为**端节点**，一个节点的子节点如果是“与”关系，则该节点便称为**与节点**，一个节点的子节点如果是“或”关系，则该节点便称为**或节点**。注意，终止节点一定是端节点，但端节点不一定是终止节点。

4.3.2 与或图搜索

1. 搜索方式，解图（树）

同状态图(即或图)的搜索一样，与或图搜索也分为树式和“线”式两种类型。对于树式搜索来讲，其搜索过程也是不断地扩展节点，并配以返回指针，而形成一棵不断生长的搜索树。

解图（树）是由可解节点形成的一个子图（树），这个个子图（树）的根是初始节点，叶为终止节点，且这个子图（树）还一定是与图（树）。

2.可解性判别

怎样判断一个节点的可解性呢？下面我们给出判别准则。

(1)一个节点是可解，则节点须满足下列条件之一：

①终止节点是可解节点；

②一个与节点可解，当且仅当其子节点全都可解；

③一个或节点可解，只要其子节点至少有一个可解。

(2)一个节点是不可解，则节点须满足下列条件之一：

①非终止节点的端节点是不可解节点；

②一个与节点不可解，只要其子节点至少有一个不可解；

③一个或节点不可解，当且仅当其子节点全都不可解。

3.搜索策略

与或图搜索也分为盲目搜索和启发式搜索两大类。前者又分为穷举搜索和盲目碰撞搜索。穷举搜索又分为深度优先和广度优先两种基本策略。

4.搜索算法

同一般状态图搜索一样，一般与或图搜索也涉及一些复杂的处理。因篇幅所限，我们仅介绍特殊的与或图—与或树的搜索算法。与或树的树式搜索过程可概括为以下步骤：

步1 把初始节点S_0放入OPEN表；

步2 移出OPEN表的第一个节点N放入CLOSED表，并冠以序号n；

步3 若节点N可扩展，则做下列工作：

(1)扩展N，将其子节点配上指向父节点的指针后放入OPEN表；

(2)考察这些子节点中是否有终止节点。若有，则标记它们为可解节点，并将它们放入CLOSED表，然后由它的可解返回推断其先辈节点的可解性，并对其中的可解节点进行标记。如果初始节点S_0也被标记为可解节点，则搜索成功，结束。

(3) 删去OPEN表中那些具有可解先辈的节点（因为其先辈节点已经可解，故已无再考察该节点的必要），转步2；

步4 若N不可扩展，则做下列工作：

(1)标记N为不可解节点，然后由它的不可解返回推断其先辈节点的可解性，并对其中的不可解节点进行标记。如果初始节点S_0也被标记为不可解节点，则搜索失败，退出。

(2)删去OPEN表中那些具有不可解先辈的节点（因为其先辈节点已不可解，故已无再考察这些节点的必要），转步2；

同状态图搜索一样，搜索成功后，解树已经记录在CLOSED表中。这时需按指向父节点的指针找出整个解树。下面举一个广度优先搜索的例子。

例4.16 设有与或树如图4–14所示，其中1号节点为初始节点，t_1、t_2、t_3、t_4均为终止节点，A和B是不可解的端节点。采用广度搜索策略，搜索过程如下：

(1)扩展1号节点，得2号和3号节点，依次放入OPEN表尾部。由于这两个节点都非终止节点，所以接着扩展2号节点。此时OPEN表中只有3号节点。

(2)2号节点扩展后，得4号节点和t_1节点。此时OPEN表中依次有3号、4号和t_1节点。

(3)扩展3号节点，得5号节点和B节点。两者均非终止节点，所以继续扩展4号节点。

(4) 4号节点扩展后得节点A和t_2。t_2是终止节点，标记为可解节点，放入CLOSED表。

(5) 扩展5号节点得t_3和t_4。由于t_3和t_4都为终止节点(放入CLOSED表)，故可推得节点5、3、1均为可解节点。搜索成功，结束。

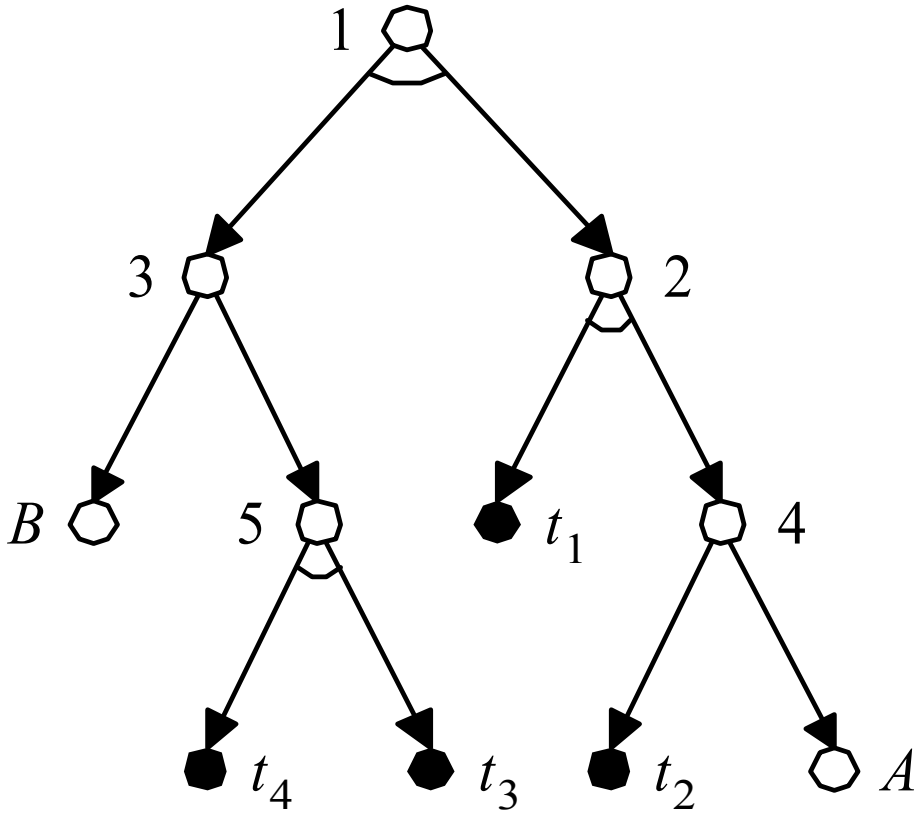

图4—14　与或树及其解树

4.3.3 启发式与或树搜索

广度优先搜索及深度优先搜索都是盲目搜索，其共同点是：

(1)搜索从初始节点开始，先自上而下地进行搜索，寻找终止节点及端节点，然后再自下而上地进行可解性标记，一旦初始节点被标记为可解节点或不可解节点，搜索就不再继续进行。

(2)搜索都是按确定路线进行的，当要选择一个节点进行扩展时，只是根据节点在与或树中所处的位置，而没有考虑要付出的代价，因而求得的解树不一定是代价最小的解树，即不一定是最优解树。

为了求得最优解树，就要在每次确定欲扩展的节点时，先往前多看几步，计算一下扩展这个节点可能要付出的代价，并选择代价最小的节点进行扩展。这种根据代价决定搜索线路的方法称为与或树的**有序搜索**，它是一种重要的启发式搜索策略。

1.解树的代价

解树的代价就是树根的代价。树根的代价是从树叶开始自下而上逐层计算而求得的。而解树的根对应的是初始节点S_0。这就是说，在与或树的搜索过程中，代价的计算方向与搜索树的生长方向相反。这一点是与状态图不同的。具体来讲，有下面的计算方法：

设g(x)表示节点x的代价，c(x,y)表示节点x到其子节点y的代价(即边xy的代价)，则

(1)若x是终止节点，g(x)＝0；

(2)若x是或节点，$g(x)=\min_{1\le i\le n}\{c(x,y_i)+g(y_i)\}$，其中$y_1$，$y_2$，…，$y_n$是x的子节点；

(3)若x是与节点x，则有两种计算公式。

1）和代价法　$g(x)=\sum_{i=n}^{n}\{c(x,y_i)+g(y_i)\}$

2）最大代价法　$g(x)=\max_{1\le i\le n}\{c(x,y_i)+g(y_i)\}$，其中$y_1$，$y_2$，…，$y_n$是x的子节点；

(4)对非终止的端节点x，g(x)＝∞。

例4.17 如图4–15所示的与或树，其中包括两棵解树，一棵解树由S_0,A,t_1和t_2组成；另一棵解树由S_0,B,D,G,t_4和t_5组成。在此与或树中，t_1,t_2,t_3,t_4,t_5为终止节点；E,F是端节点，其代价均为∞；边上的数字是该边的代价。由右边的解树可得：

按和代价：$g(A)=11, g(S_0)=13$

按最大代价：$g(A)=6, g(S_0)=8$

由左边的解树可得：

按和代价：$g(G)=3, g(D)=4, g(B)=6, g(S_0)=8$

按最大代价：$g(G)=2, g(D)=3, g(B)=5, g(S_0)=7$

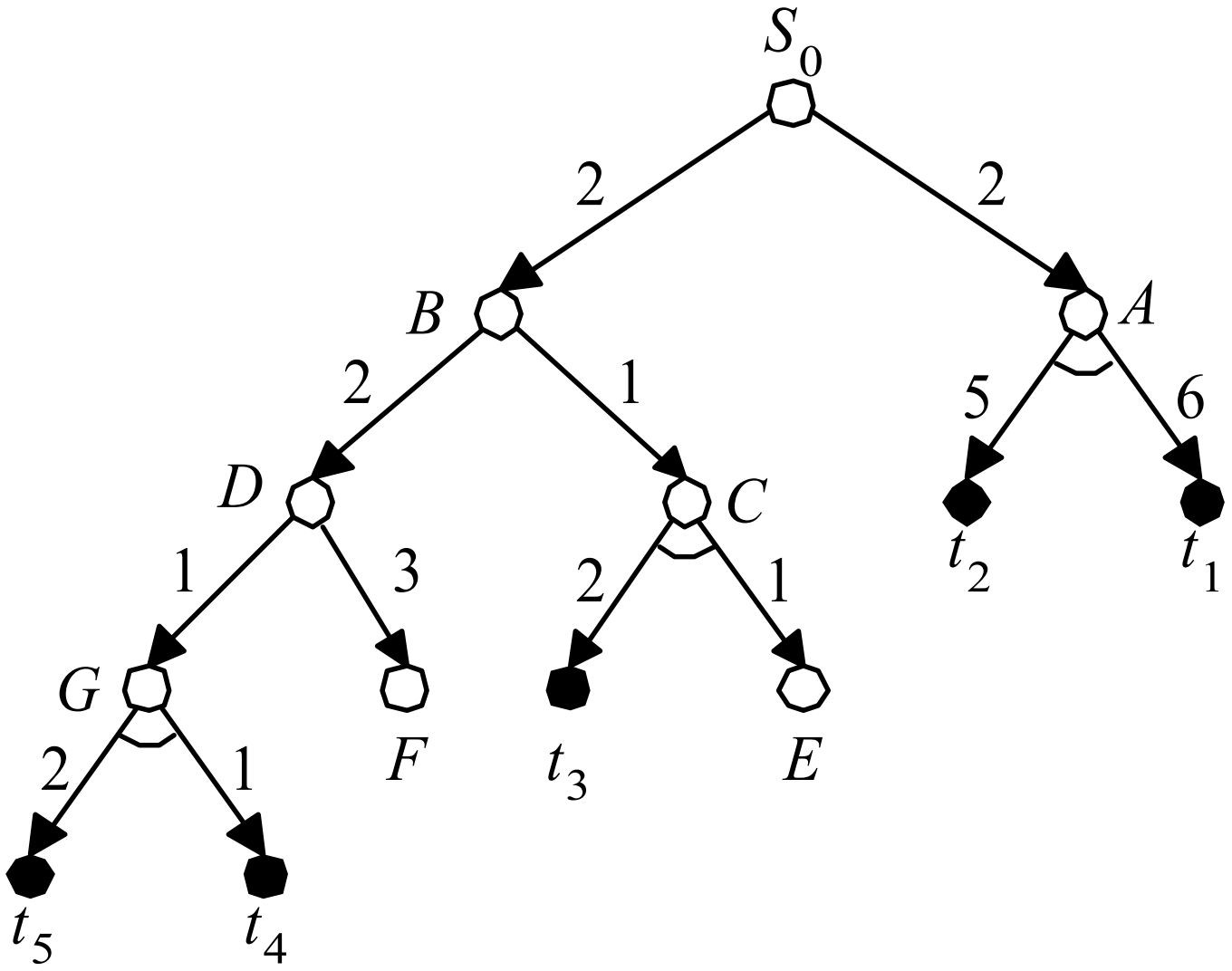

图4—15 含代价的与或树

显然，若按和代价计算，左边的解树是最优解树，其代价为8;若按最大代价计算，左边的解树仍然是最优解树，其代价是7。但有时用不同的计算代价方法得到的最优解树不相同。

2. 希望树

无论是用和代价法还是最大代价法，当要计算任一节点x的代价g(x)时，都要求已知其子节点y_i的代价$g(y_i)$。但是，搜索是自上而下进行的，即先有父节点，后有子节点，除非节点x的全部子节点都是不可扩展节点，否则子节点的代价是不知道的。此时节点的代价如何计算？采用启发式信息。

有序搜索的目的是求出最优解树，即最小代价的解树。因此，在扩展过程中要选最有希望的节点进行扩展，这样由该节点及父辈节点就有可能成为最优解树的一部分，因此称为**希望树**。每当新一代的节点生成时，都要自下而上地重新计算其先辈节点的代价g，这是一个自上而下地生成新节点，又自下而上地计算代价g 的反复进行的过程。

3.与或树的有序搜索过程

与或树的有序搜索过程是一个不断选择、修正希望树的过程。如果问题有解，则经有序搜索将找到最优解树。

其搜索过程如下：

(1)把初始节点S_0放入OPEN表中。

(2)求出希望树T，即根据当前搜索树中节点的代价g求出以S_0为根的希望树T。

(3)依次把OPEN表中T的端节点N选出放入CLOSED表中。

(4)如果节点N是终止节点，则做下列工作：

①标示N为可解节点。

②对T应用可解标记过程，把N的先辈节点中的可解节点都标记为可解节点。

③若初始节点S_0能被标记为可解节点，则T就是最优解树，成功退出。

④否则，从OPEN表中删去具有可解先辈的所有节点。

(5)如果节点N不是终止节点，且它不可扩展，则做下列工作：

①标示N为不可解节点。

②对T应用不可解标记过程，把N的先辈节点中的不可解节点都标记为不可解节点。

③若初始节点S_0也被标记为不可解节点，则失败退出。

④否则，从OPEN表中删去具有不可解先辈的所有节点。

(6)如果节点N不是终止节点，但它可扩展，则可做下列工作：

①扩展节点N，产生N的所有子节点。

②把这些子节点都放入OPEN表中，并为每一个子节点配置指向父节点（节点N）的指针。

③计算这些子节点的g值及其先辈节点的g值。

(7)转(2)。

例4.18 下面我们举例说明上述搜索过程。

设初始节点为S_0，每次扩展两层，并设S_0经扩展后得到如图4–16(a)所示的与或树，其中子节点B，C，E，F用启发函数估算出的g值分别是

g(B)=3，g(C)=3，g(E)=3，g(F)=2

若按和代价计算，则得到

g(A)=8，g(D)=7，$g(S_0)$=8

(注:这里把边代价一律按1计算，下同。)

此时，S_0的右子树是希望树。下面将对此希望树的节点进行扩展。

设对节点E扩展两层后得到如图4–16(b)所示的与或树，节点旁的数字为用启发函数估算出的g值。则按和代价法计算得到

g(G)=7,g(H)=6,g(E)=7,g(D)=11

此时，由S_0的右子树算出的$g(S_0)=12$。但是，由左子树算出的$g(S_0)=9$。显然，左子树的代价小，所以现在改取左子树作为当前的希望树。

假设对节点B扩展两层后得到如图4–16(c)所示的与或树，节点旁的数字是对相应节点的估算值，节点L的两个子节点是终止节点，则按和代价法计算得到

$$g(L)=2,g(M)=6,g(B)=3,g(A)=8$$

由此可推算出$g(S_0)=9$。这时，左子树仍然是希望树，继续对其扩展。该扩展节点C。

假设节点C扩展两层后得到如图4–16(d)所示的与或树，节点旁的数字是对相应节点的估算值，节点N的两个子节点是终止节点。按和代价计算得到

$$g(N)=2,g(P)=7,g(C)=3,g(A)=8$$

由此可推算出$g(S_0)=9$。另外，由于N的两个子节点都是终止节点，所以N和C都是可解节点。再由前面推出的B是可解节点，可推出A和S_0都是可解节点。这样就求出了代价最小的解树，即最优解树——图4–16(d)中粗线部分所示。该最优解树是用和代价法求出来的，解树的代价为9。

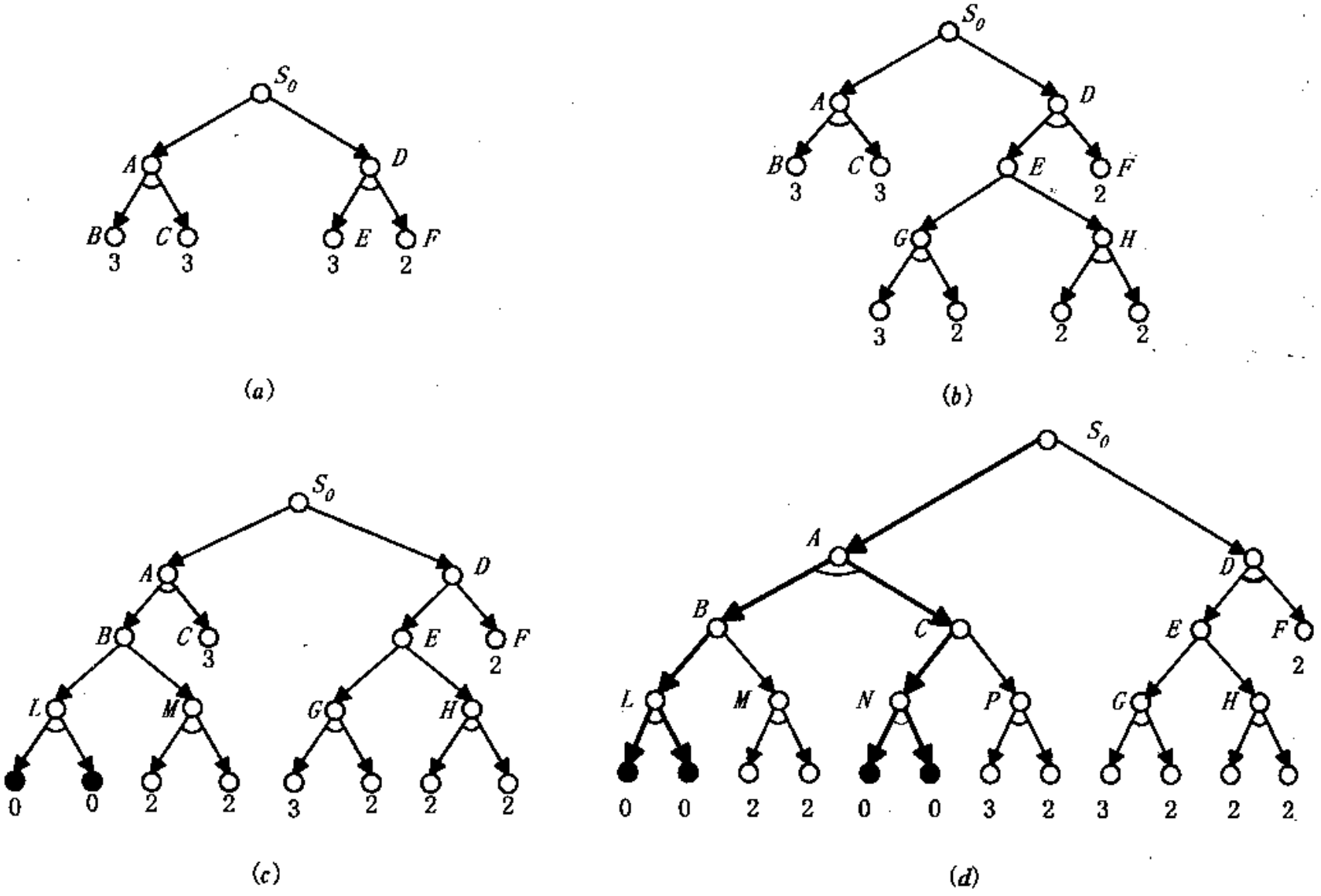

图4—16 与或树有序搜索

4.4 与或图问题求解

4.4.1 问题的与或图表示

与或图是描述问题求解的另一种有向图。与或图一般表示问题的变换过程（而不是状态变换）。具体讲，它是从原问题出发，通过运用某些规则不断进行问题分解（得到与分支）和变换（得到或分支），而得到一个与或图。换句话说，与或图的节点一般代表问题。那么，整个图也就表示问题空间。与或图中的父节点与其子节点之间服从逻辑上的与、或运算关系。所以，与或图表示的问题是否有解，要进行逻辑判断，与或图的搜索也受逻辑的制约。

与或图也是一个三元组

（Q_0，F，Q_n）

这里，Q_0表示初始问题，F表示问题变换规则集，Q_n表示本原问题集。

例如，高等数学中的积分公式，就是一些典型的问题分解和变换规则，所以，一般的求不定积分问题就可用与或图来描述。

例4.19 三阶梵塔问题。

对于梵塔问题，我们也可以这样考虑：为把1号杆上的n个盘子搬到3号杆，可先把上面的n-1个盘子搬到2号杆上；再把剩下的一个大盘子搬到3号杆；然后再将2号杆上的n-1个盘子搬到3号杆。这样，就把原来的一个问题分解为三个子问题。这三个子问题都比原问题简单，其中第二个子问题已是直接可解的问题。对于第一和第三两个子问题，可用上面n个盘子的方法，做同样的处理。根据这一思想，我们可把三阶梵塔问题分解为下面的三个子问题：

(1)把A、B盘从1号杆移到2号杆。

(2)把C盘从1号杆移到3号杆。

(3)把A、B盘从2号杆移到3号杆。

其中子问题(1)、(3)又分别可分解为三个子问题。

于是，我们可得到三阶梵塔问题的与或树表示（见图4—17）：

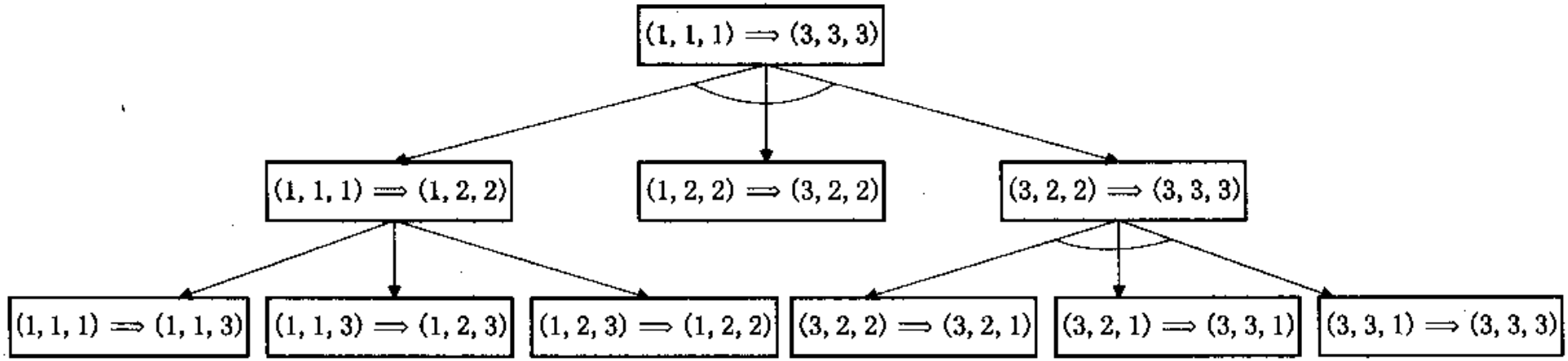

图4–17　三阶梵塔问题的与或树

需说明的是，三元组

(i，　j，　k)

i代表金盘**C**所在的杆号;j代表金盘**B**所在的杆号，k代表金盘**A**所在的杆号。在图4-17所示的与或树中，共有七个终止节点，对应于七个本原问题，它们是通过“分解”得到的。若把这些本原问题的解按从左至右的顺序排列，就得到了原始问题的解：

$$(1,1,1) \Rightarrow (1,1,3)$$

$$(1,1,3) \Rightarrow (1,2,3)$$

$$(1,2,3) \Rightarrow (1,2,2)$$

$$(1,2,2) \Rightarrow (3,2,2)$$

$$(3,2,2) \Rightarrow (3,2,1)$$

$$(3,2,1) \Rightarrow (3,3,1)$$

$$(3,3,1) \Rightarrow (3,3,3)$$

4.4.2 与或图问题求解程序举例

例4.20 基于与或图搜索的迷宫问题程序。

```
        /*puzzle room problem*/
DOMAINS
    room list=room*
    room=symbol
PREDICATES
     road(room,room)
     path(room,room,room list)
     go(room,room)
      member(room,room list)
```

GOAL

go(a,e).

CLAUSES

go(X,Y)：-path(X,Y, ［X］).%首先将入口放入表中，该表用来记录走过的路径

path(X,X,L)：-write(L).%当path中的两个点相同时，表明走到了出口。程序结束

path(X,Y,L)：-%这个语句实际是问题分解规则，它将原问题分解为两个子问题

road(X,Z),%从当前点向前走到下一点Z

```
not(member(Z,L)),
path(Z,Y,［Z|L］).%再找Z到出口Y的路径
member(X，　［X|-］).
member(X，　［-|T］)if member(X,T).
/* 迷宫图 */
road(a,b).road(a,c).road(b,f).road(f,g).road(f,ff).road(g,h).
road(g,i).road(b,d).road(c,d).road(d,e).road(e,b).
```

例4.21 梵塔问题程序。

对于梵塔问题，我们这样考虑：为把1号杆上的n个盘子搬到3号杆，可先把上面的n-1个盘子搬到2号杆上；再把剩下的一个大盘子搬到3号杆；然后再将2号杆上的n-1个盘子搬到3号杆。这样，就把原来的一个问题分解为三个子问题。这三个子问题都比原问题简单，其中第二个子问题已是直接可解的问题。对于第一和第三两个子问题，可用上面n个盘子的方法，做同样的处理。于是，可得递归程序如下：

```
        /*Hanoitower*/
DOMAINS
        disk-amount,pole-No=integer
PREDICATES
        move(disk-amount,pole-No,pole-No,pole-No)
GOAL
        move(5,1,2,3).
```

```
CLAUSES
 move(0，-，-，-)：-!.
 move(N,X,Y,Z)：-  /*move N disks from X to Z*/
  M=N-1，
  move(M,X,Z,Y),write(X,"to",Z),move(M,Y,X,Z).
```

程序中的盘子数取为5。

4.5　博弈树搜索

诸如下棋、打牌、竞技、战争等一类竞争性智能活动称为博弈。其中最简单的一种称为“二人零和、全信息、非偶然”博弈。

所谓“二人零和、全信息、非偶然”博弈是指：

(1)对垒的A，B双方轮流采取行动，博弈的结果只有三种情况：A方胜，B方败；B方胜，A方败；双方战成平局。

(2)在对垒过程中，任何一方都了解当前的格局及过去的历史。

(3)任何一方在采取行动前都要根据当前的实际情况，进行得失分析，选取对自己最为有利而对对方最为不利的对策，不存在“碰运气”的偶然因素。即双方都是很理智地决定自己的行动。

4.5.1 博弈树的概念

在博弈过程中，任何一方都希望自己取得胜利。因此，当某一方当前有多个行动方案可供选择时，他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。

这样，如果站在某一方（如A方，即在A要取胜的意义下)，把上述博弈过程用图表示出来，则得到的是一棵“与或树”。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；所有使对方获胜的终局都是不可解节点。

4.5.2 极小极大分析法

在二人博弈问题中，为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案，就需要对当前的情况以及将要发生的情况进行分析，从中选出最优的走步。最常使用的分析方法是极小极大分析法。其基本思想是：

(1)设博弈的双方中一方为A，另一方为B。然后为其中的一方(例如A)寻找一个最优行动方案。

(2)为了找到当前的最优行动方案，需要对各个可能的方案所产生的后果进行比较。

(3)为计算得分，需要根据问题的特性信息定义一个估价函数，用来估算当前博弈树端节点的得分。

(4)当端节点的估值计算出来后，再推算出父节点的得分，推算的方法是：对“或”节点，选其子节点中一个最大的得分作为父节点的得分，这是为了使自己在可供选择的方案中选一个对自己最有利的方案；对“与”节点，选其子节点中一个最小的得分作为父节点的得分，这是为了立足于最坏的情况。

(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4–18给出了计算倒推值的示例。

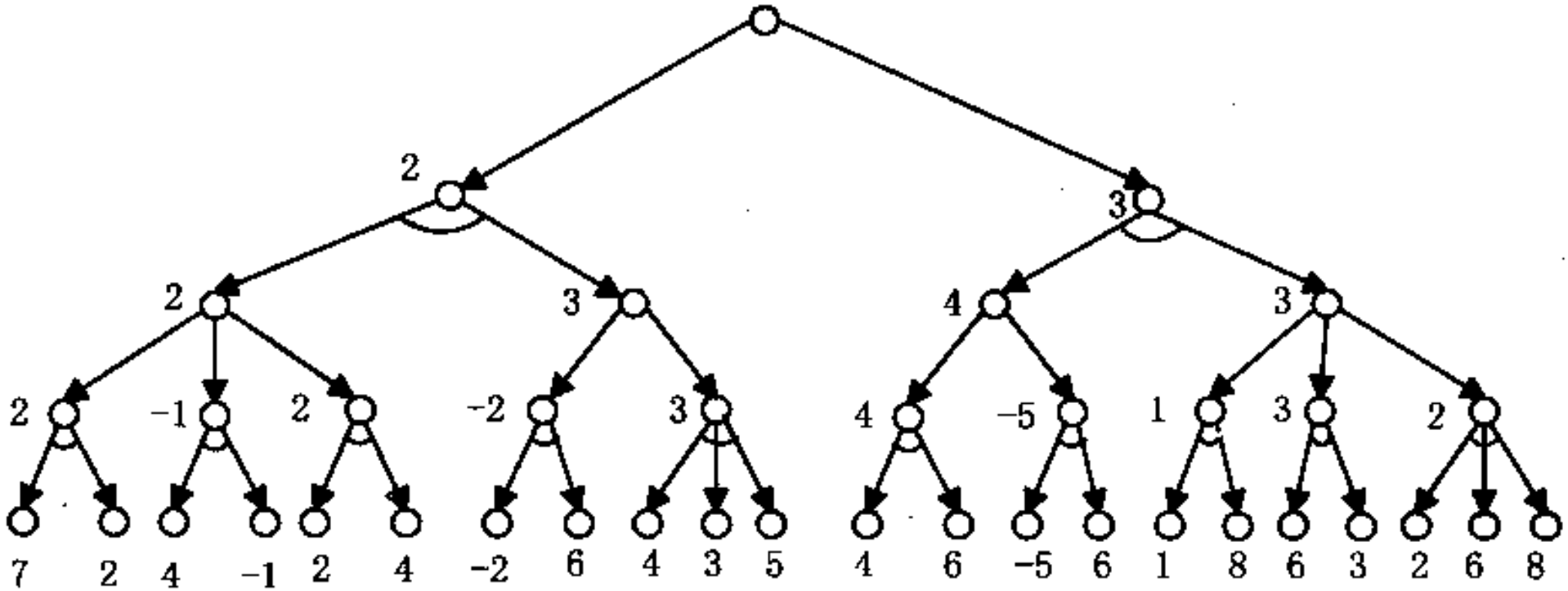

图4—18　倒推值的计算

例4.22　一字棋游戏。设有如图4–19(a)所示的九个空格，由A，B二人对弈，轮到谁走棋谁就往空格上放一只自己的棋子，谁先使自己的棋子构成“三子成一线”谁就取得了胜利。

图4—19　一字棋

设A的棋子用“a″表示，B的棋子用“b″表示。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A必胜的棋局，则e(P)=+∞。

(2)若P是B必胜的棋局，则e(P)=-∞。

(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a成为三子成一线的数目；e(-P)表示棋局P上有可能使b成为三子成一线的数目。例如，对于图4–19(b)所示的棋局，则

$$e(P)=6-4=2$$

另外，我们假定具有对称性的两个棋局算作一个棋局。还假定A先走棋，我们站在A的立场上。

图4–20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有较大的倒推值。

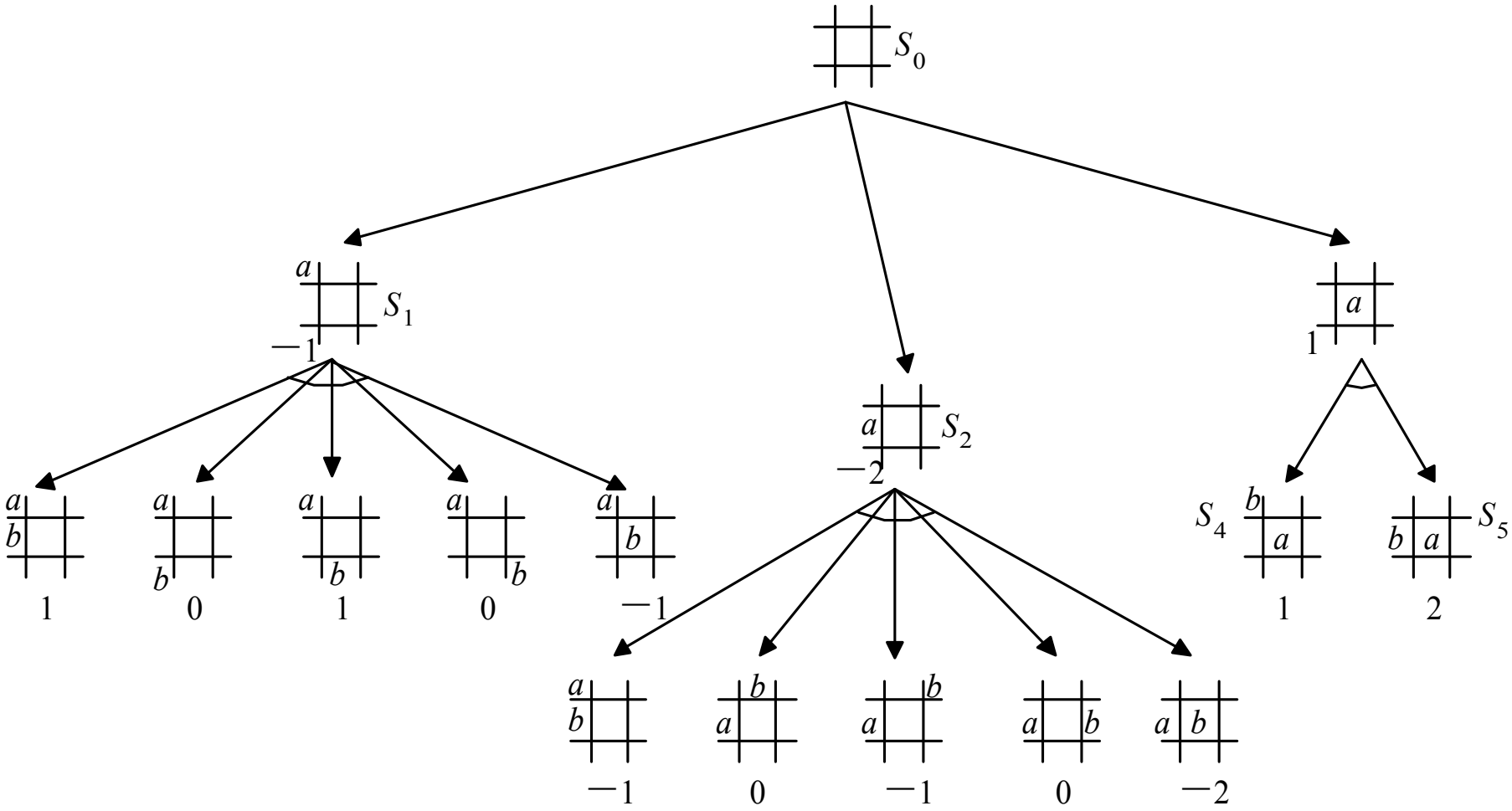

图4—20　一字棋极小极大搜索

4.5.3 α–β剪枝技术

上述的极小极大分析法,实际是先生成一棵博弈树,然后再计算其倒推值。这样做的缺点是效率较低。于是,人们又在极小极大分析法的基础上,提出了α–β剪枝技术。

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。

例4.23　图4–21所示的博弈树搜索,n就采用了α—β剪枝技术。

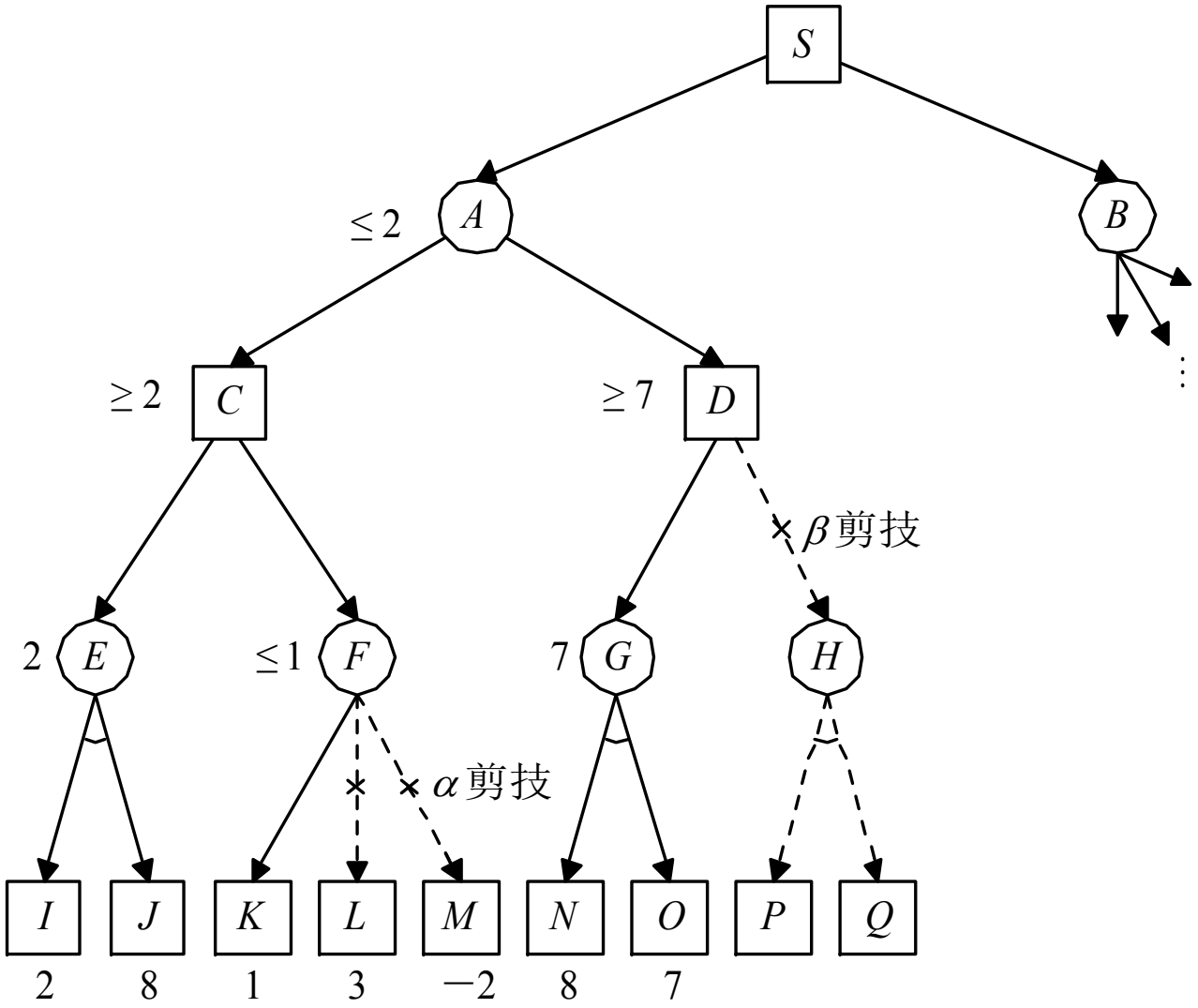

图4—21　α—β剪枝