第 3 章

图搜索与问题求解

图搜索是人工智能中发展最早的技术，现已取得了不少成果。本章主要介绍传统的图搜索技术[包括状态图(空间)搜索、与或图搜索和博弈树搜索]及其问题求解。

3.1 状态图与状态图搜索

3.1.1 状态图

先看几个智力问题及其求解。

迷宫问题 走迷宫是人们熟悉的一种游戏，图 3-1 所示的就是一个迷宫。如果把该迷宫的每一个格子以及入口和出口作为节点，把通道作为边，则该迷宫可以由一个有向图表示(见图 3-2)。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，或者是在该有向图中寻找从初始节点到目标节点路径的问题。

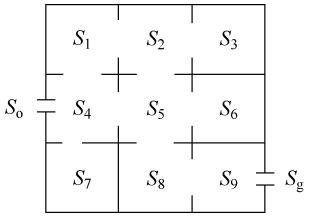

图 3-1 迷宫图

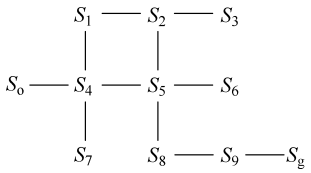

图 3-2 迷宫的有向图表示

八数码问题 在一个 3×3 的方格棋盘上放置 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(见图 3-3)，给出数码的移

动序列。该问题称为八数码问题或重排九宫问题。

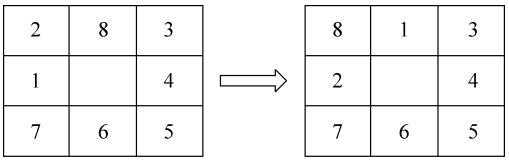

图 3-3　八数码问题示例

可以想象，如果把整个棋盘所表示的一个棋局作为一个节点，则相邻的节点就可以通过移动数码一个一个地产生出来。这样，所有节点按相邻关系就可以连成一个有向图。可以看出，图中的一条边（即相邻两个节点的连线）就对应一次数码移动；反之，一次数码移动也就对应着图中的一条边。数码移动是按移动规则进行的。所以，图中的一条边也就代表一个移动规则或者移动规则的一次执行。那么，这个八数码问题也就是从该有向图的初始节点（初始棋局）出发寻找目标节点（目标棋局）的问题，或者是在该有向图中寻找一条从初始节点到目标节点的路径问题。

以上两个问题虽然内容不同，但抽象地看，它们都是在某个有向图中寻找目标或者路径的问题。在人工智能中，把这种描述问题的有向图称为状态空间图，简称**状态图**（state graph）。之所以称为状态图，是因为图中的节点代表问题中的一种格局，一般称之为问题的一个状态；边表示两节点之间的某种联系，如可以是某种操作、规则、变换、算子、通道或关系等。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。根据实际需要，路径解可以表示为边的序列或节点的序列。例如，迷宫问题的解可以是节点序列，而八数码问题的解可以是边（即棋步）序列。

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者路径的问题。在状态图中寻找目标或者路径的基本方法就是搜索。因此，研究状态图搜索具有普遍意义。

3.1.2　状态图搜索

所谓搜索，顾名思义，就是从初始节点出发，在图中试探地前进，寻找目标节点的过程（也可以反向进行）。那么，当目标节点找到后，路径也就找到了。所以，寻找目标和寻找路径是一致的。可以想象，由于图中有许多节点和边，因此，搜索过程中经过的节点和边，按连接关系，便会构成一个树形的有向图。这种树形有向图称为**搜索树**。随着搜索的进行，搜索树会不断地生长，直到当搜索树中出现目标节点时，搜索便停止。这时从搜索树中就可以容易地找出从初始节点到目标节点的路径来。所以，在搜索过程中应当随时记录搜索轨迹。

上面仅是对搜索的通俗描述。现在我们考虑如何用计算机来实现上述搜索。

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，形象地说就是以“画树”的方式进行搜索。即从树根（初始节点）出发，一笔一笔地描绘出一棵树来。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所产生的搜索树。

所谓线式搜索，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分为不回溯的和可回溯的两种。不回溯的线式搜索就是每到一个“岔路口”仅沿一条路继续前进，即对每一个节点始终都仅生成一个子节点（如果有子节点的话）。生成一个节点的子节点也称对该节点进行扩展。这样，如果扩展到某一个节点，该节点恰好就是目标节点，则搜索成功；如果直到不能再扩展时，还未找到目标节点，则搜索失败。可回溯的线式搜索也是对每一个节点都仅扩展一条边，但当不能再扩展时，则退回一个节点，然后再扩展另一条边（如果有的话）。这样，要么最终找到了目标节点，搜索成功；要么一直回溯到初始节点也未找到目标节点，则搜索失败。

由上所述可以看出，树式搜索成功后，还需再从搜索树中找出所求路径，而线式搜索只要搜索成功，则“搜索线”就是所找的路径，即问题的解。

那么，又怎样从搜索树中找出所求路径呢？这只需在扩展节点时记住节点间的关系即可。这样，当搜索成功时，从目标节点反向沿搜索树按所做标记追溯回去一直到初始节点，便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，大体可分为盲目搜索和启发式（heuristic）搜索两大类。

通俗地讲，盲目搜索就是无“向导”的搜索，启发式搜索就是有“向导”的搜索。那么，树式盲目搜索就是穷举式搜索，即从初始节点出发，沿连接边逐一考察各个节点（看是否为目标节点），或者反向进行；而线式盲目搜索，对于不回溯的就是随机碰撞式搜索，对于回溯的则也是穷举式的搜索。

启发式搜索则是利用“启发性信息”引导的搜索。所谓“启发性信息”就是与问题有关的有利于尽快找到问题解的信息或知识。例如“欲速则不达”“知己知彼，百战不殆”“学如逆水行舟，不进则退”等格言，就是指导人们行为的启发性信息。常识告诉人们，如果有向导引路，就会因少走弯路而事半功倍。所以，启发式搜索往往会提高搜索效率，而且可能找到问题的最优解。根据启发性信息的内容和使用方式的不同，启发式搜索又可分为许多不同的策略，如全局择优、局部择优、最佳图搜索等。

按搜索范围的扩展顺序的不同，搜索又可分为广度优先和深度优先两种类型。树式搜索既可深度优先进行，也可广度优先进行；不回溯的线式搜索，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，我们用一个名为 CLOSED 表（如图 3-4 所示）的动态数据结构来专门记录考察过的节点。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，可能它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，还得不断地把待考察的节点组织在一起，并做某种排序，以便控制搜索的方向和顺序。为此，我们采用一个名为 OPEN 表（如图 3-4 所示）的动态数据结构，来专门登记当前待考察的节点。

OPEN 表

节点	父节点编号

CLOSED 表

编号	节点	父节点编号

图 3-4　OPEN 表与 CLOSED 表示例

下面给出树式搜索和线式搜索的一般算法。

树式搜索算法

（1）把初始节点 S_0 放入 OPEN 表中。

（2）若 OPEN 表为空，则搜索失败，退出。

（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。

（4）若目标节点 $S_g = N$，则搜索成功，结束。

（5）若 N 不可扩展，则转（2）。

（6）扩展 N，生成一组子节点，对这组子节点做如下处理：

① 删除 N 的先辈节点（如果有的话）。

② 对已存在于 OPEN 表的节点（如果有的话）也删除之；但删除之前要比较其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 表中的原返回指针，使其沿新路径返回（见图 3-5）。

③ 对已存在于 CLOSED 表的节点（如果有的话），做与②同样的处理，并且再将其移出 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 的返回指针后放入 OPEN 表中某处，或对 OPEN 表进行重新排序，转（2）。

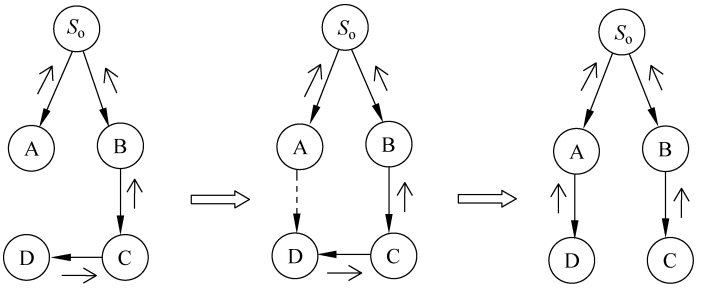

图 3-5　状态图搜索过程中修改返回指针示例

说明：

(1) 这里的返回指针也就是父节点在 CLOSED 表中的编号。

(2) 步骤(6)中修改返回指针的原因是，这些节点又被第二次生成，所以它们返回初始节点的路径已有两条，但这两条路径的“长度”可能不同。当新路径短时便会走新路径。

(3) 这里路径的长短是按路径上的节点数来衡量的，后面将会看到路径的长短也可以按其“代价”(如距离、费用、时间等)衡量。若按其代价衡量，则在需修改返回指针的同时修改相应的代价值，或者不修改返回指针但要修改代价值(为了实现代价小者优先扩展)。

线式搜索算法分为不回溯的线式搜索和可回溯的线式搜索两种。

不回溯的线式搜索

(1) 把初始节点 S_0 放入 CLOSED 表中。

(2) 令 $N=S_0$。

(3) 若 N 是目标节点，则搜索成功，结束。

(4) 若 N 不可扩展，则搜索失败，退出。

(5) 扩展 N，选取其一个未在 CLOSED 表中出现过的子节点 N_1 放入 CLOSED 表中，令 $N=N_1$，转步骤(3)。

可回溯的线式搜索

(1) 把初始节点 S_0 放入 CLOSED 表中。

(2) 令 $N=S_0$。

(3) 若 N 是目标节点，则搜索成功，结束。

(4) 若 N 不可扩展，则移出 CLOSED 表的末端节点 N_e，若 $N_e=S_0$，则搜索失败，退出。否则，以 CLOSED 表新的末端节点 N_e 作为 N，即令 $N=N_e$，转步骤(3)。

(5) 扩展 N，选取其一个未在 CLOSED 表中出现过的子节点 N_1，放入 CLOSED 表中，令 $N=N_1$，转步骤(3)。

需要说明的是，上述算法仅是搜索目标节点的算法，当搜索成功后，如果需要路径，还须由 CLOSED 表再找出路径。找路径的方法是：对于树式搜索，从 CLOSED 表中序号最大的节点起，根据返回指针追溯至初始节点 S_0，所得的节点序列或边序列即为所找路径；对于线式搜索，CLOSED 表即为所找路径。

3.1.3　穷举式搜索

下面先讨论树形结构的状态图搜索，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。

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察，只有当同一级节点考察完之后，才考察下一级节点。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步生成的。

例 3-1 用广度优先搜索策略求解八数码问题。

设初始节点 S_0 和目标节点 S_g 分别如图 3-3 所示的初始棋局和目标棋局，用广度优先搜索策略，即可得到如图 3-6 所示的搜索树。

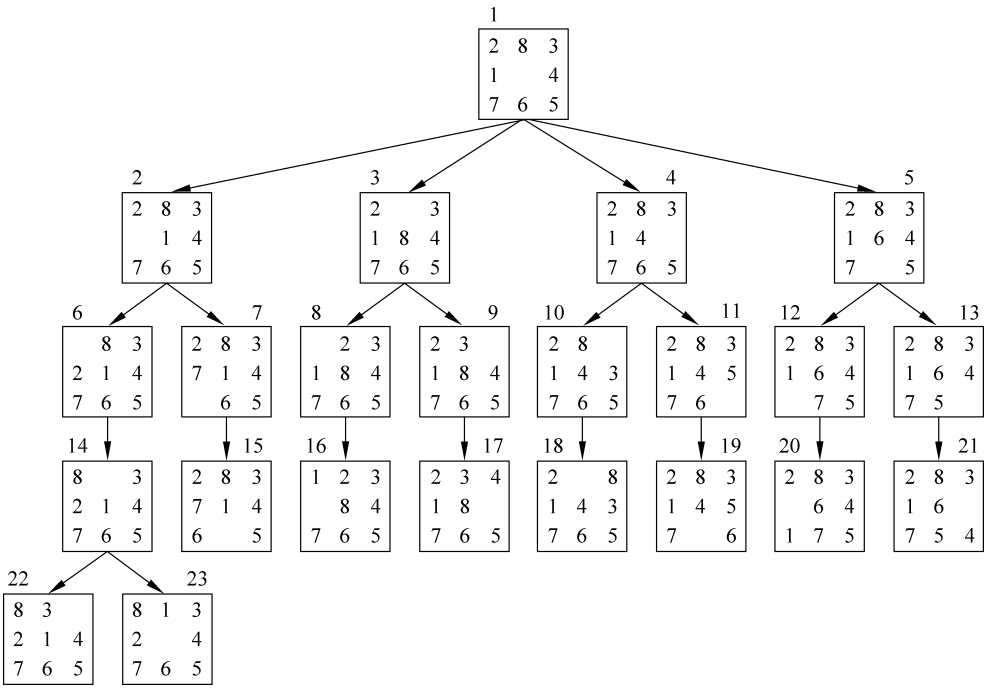

图 3-6 八数码问题的广度优先搜索

广度优先搜索算法

(1) 把初始节点 S_0 放入 OPEN 表中。

(2) 若 OPEN 表为空，则搜索失败，退出。

(3) 取 OPEN 表中前面第一个节点 N 放在 CLOSED 表中，并冠以顺序编号 n。

(4) 若目标节点 $S_g = N$，则搜索成功，结束。

(5) 若 N 不可扩展，则转步骤(2)。

(6) 扩展 N，将其所有子节点配上指向 N 的指针依次放入 OPEN 表尾部，转步骤(2)。

其中 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。

广度优先搜索也被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，则一定能用它找到解，且找到的还是最优解(即最短的路径)。这是广度优先搜索的优点。它的缺点是搜索效率低。

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步形成的。

深度优先搜索算法

(1) 把初始节点 S_0 放入 OPEN 表中。

(2) 若 OPEN 表为空，则搜索失败，退出。

(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。

(4) 若目标节点 $S_g=N$，则搜索成功，结束。

(5) 若 N 不可扩展，则转步骤(2)。

(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，转步骤(2)。

可以看出，这里的 OPEN 表为一个堆栈，这是与广度优先算法的唯一区别。

例 3-2　对于八数码问题，应用深度优先搜索策略，可得如图 3-7 所示的搜索树。

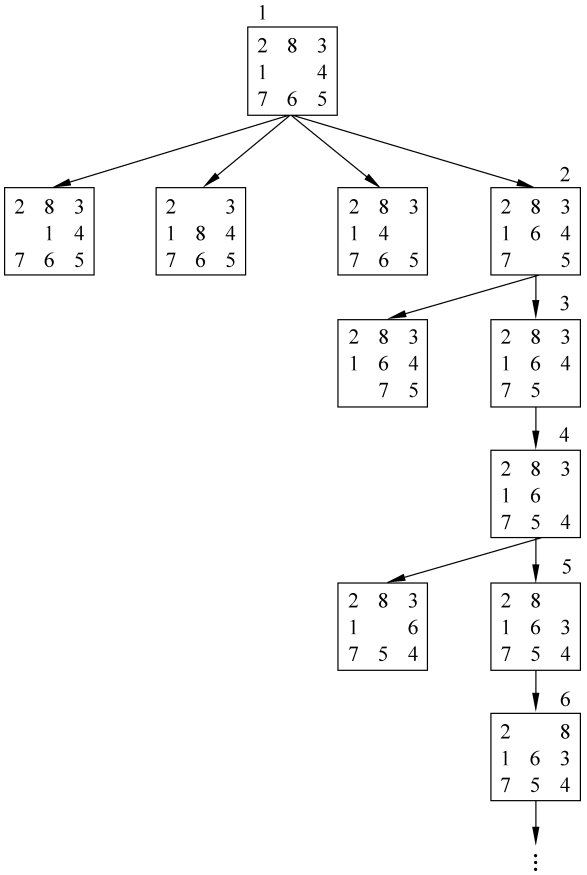

图 3-7　八数码问题的深度优先搜索

深度优先搜索也被称为纵向搜索。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）。

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。

有界深度优先搜索算法

(1) 把 S_0 放入 OPEN 表中，置 S_0 的深度 $d(S_0)=0$。

(2) 若 OPEN 表为空，则失败，退出。

(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。

(4) 若目标节点 $S_g=N$，则成功，结束。

(5) 若 N 的深度 $d(N)=d_m$（深度限制值），或者若 N 无子节点，则转步骤(2)。

(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 $d(N_i)=d(N)+1$，转步骤(2)。

3.1.4 启发式搜索

1. 问题的提出

从理论上讲，穷举搜索法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，当阶数较小（如小于6）时，在计算机上求解并不难，但当阶数再增加时，其时空要求将会急剧地增加。例如当取 64 时，则其状态空间中就有 $3^{64}=0.94\times10^{30}$ 个节点，最短的路径长度（节点数）$=2^{64}-1\approx2\times10^{19}$。这是现有任何计算机都存放不下，也不能计算的。又如博弈问题，计算机为了取胜，它可以将所有走法都试一下，然后选择最佳走步。找到这样的算法并不难，但计算的时空消耗却大得惊人。例如：就可能有的棋局数讲，一字棋是 $9!\approx3.6\times10^5$，西洋棋是 10^{78}，国际象棋是 10^{120}，围棋是 10^{761}。假设每步可以选择一种棋局，用极限并行速度（10^{-104} 秒/步）计算，国际象棋也得算 10^{16} 年。这些困难迫使人们不得不寻找更有效的搜索方法，即提出了启发式搜索策略。

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 3 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。

例如，由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，最后为零。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。可以看出，这种启发性信息属于上面的第一种类型。

需要指出的是，不存在能适合所有问题的万能启发性信息，或者说，不同的问题有不同的启发性信息。

3. 启发函数

在启发式搜索中，通常用所称的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 S_g 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？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，$h(x)$就可以定义为节点 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。下面给出树形图的树式搜索的两种启发式搜索策略及算法。

(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，其基本思想是：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

(1) 把初始节点 S_0 放入 OPEN 表中，计算 $h(S_0)$。

(2) 若 OPEN 表为空，则搜索失败，退出。

(3) 移出 OPEN 表中第一个节点 N，将其放入 CLOSED 表中，并冠以序号 n。

(4) 若目标节点 $S_g=N$，则搜索成功，结束。

(5) 若 N 不可扩展，则转步骤(2)。

(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，转步骤(2)。

例 3-3 用全局择优搜索法解八数码问题。初始棋局和目标棋局同例 3-1。

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 3-8 所示。图中节点旁的数字就是该节点的启发函数值。由图可见,此八数码问题的解为:S_0,S_1,S_2,S_3,S_g。

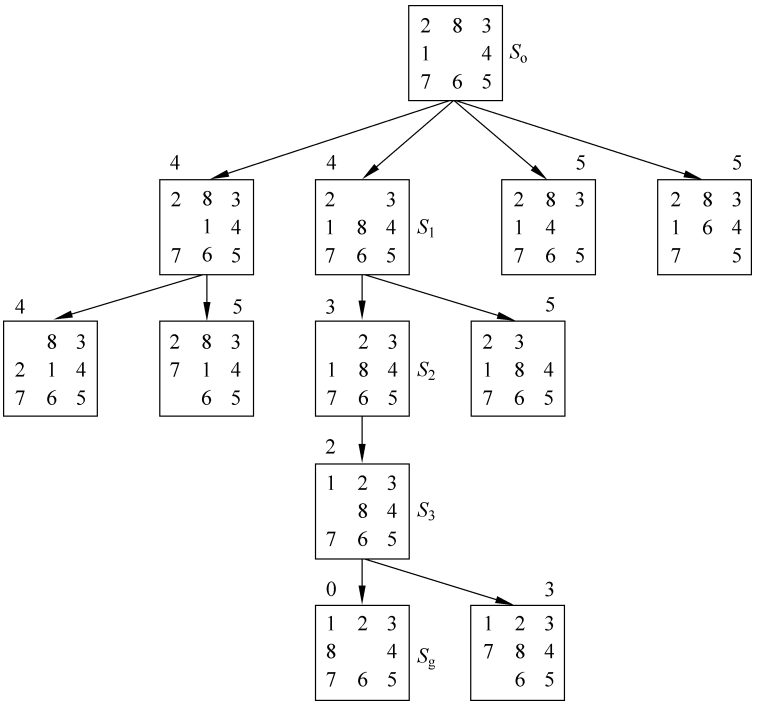

图 3-8 八数码问题的全局择优搜索

视频讲解

(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。

3.1.5 加权状态图搜索

1. 加权状态图与代价树

设图 3-9(a)所示的是一个交通图,而 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。我们考虑:如何找出一条从 A 到 E 费用最少的旅行路线。

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

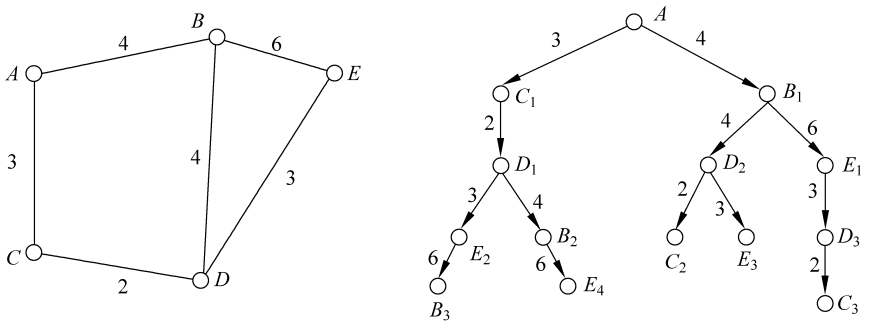

图 3-9　交通图及其代价树

一般加权状态图的搜索比较复杂，但树形加权状态图——代价树的搜索相对容易。所以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；然后对其他节点以此类推，但对其他节点 x，不能将其父节点及祖先作为 x 的子节点。例如，把如图 3-9(a)所示的交通图转换成代价树如图 3-9(b)所示。所谓代价，可以是两点之间的距离、交通费用或所需时间等。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

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

而

$$g(S_0)=0$$

下面介绍两种代价树的搜索策略，即分支界限法和最近择优法。

2. 分支界限法(最小代价优先法)

分支界限法的基本思想是：每次从 OPEN 表中选出 $g(x)$值最小的节点进行考察，而不管这个节点是在搜索树的什么位置上。

可以看出，这种搜索法与前面的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)。这样，把最好优先法算法中的 $h(x)$换成 $g(x)$即可得分支界限法的算法。所以，从算法角度考虑，这两种搜索法实际是一样的。但二者在计算节点的代价值与启发函数值的方法是有差别的。

事实上，一个节点 x 的代价值 $g(x)$是从初始节点 S_0 方向计算而来的，其计算方法为

$$g(S_0)=0$$

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

而启发函数值 $h(x)$则是朝目标节点方向计算的；$g(x)$与 x 的父节点代价有关，与子节点代价无关，而 $h(x)$与 x 的父、子节点的启发值均无关。

3. 最近择优法(盲人爬山法)

同上面的情形一样，这种方法实际上同局部择优法类似，区别也仅是选取扩展节点的

标准不同，一个是代价 $g(x)$值最小，一个是启发函数 $h(x)$值最小。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。

现在基于代价树求解上面那个旅行路线问题。易见采用分支界限法所得路径为

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

这是一条最小费用路径(费用为 8)。

3.1.6 A 算法和 A* 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上是一种深度优先的搜索策略。虽然它很高效，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[注意，$h(x)$也可以用代价来定义]。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(也可以是代价)估计值之总和。有时估价函数还可以表示为

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

其中，$d(x)$表示节点 x 的深度。

由于 $g(x)$或 $d(x)$越小，说明节点 x 越靠近初始节点 S_0，所以，$f(x)$中的 $g(x)$或 $d(x)$有利于搜索的横向发展。因而，可以提高搜索的完备性，但影响搜索效率。由于 $h(x)$越小说明节点 x 越接近目标节点 S_g，所以，$h(x)$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。而 $f(x)$恰好是二者的一个折中，这正是估价函数的优点。但在确定 $f(x)$时，还要权衡利弊，使 $g(x)$(或 $d(x)$)与 $h(x)$的比重适当，才能取得理想的效果。如果只关心到达目标节点的路径，并希望有较高的搜索效率，则 $g(x)$可以忽略。当然，这样会影响搜索的完备性。

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。

---

(1) 把附有 $f(S_0)$的初始节点 S_0 放入 OPEN 表。

(2) 若 OPEN 表为空，则搜索失败，退出。

(3) 移出 OPEN 表中第一个节点 N，并将其放入 CLOSED 表中，再冠以顺序编号 n。

(4) 若目标节点 $S_g=N$，则搜索成功，结束。

(5) 若 N 不可扩展，则转步骤(2)。

(6) 扩展 N，生成一组附有 $f(x)$的子节点，对这组子节点做如下处理：

① 考察是否有已在 OPEN 表或 CLOSED 表中存在的节点；若有，则再考察其中有无 N 的先辈节点，若有则删除之；对于其余节点，也删除，但由于它们又被第二次生成，因而需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 $f(x)$值以升序排序，转步骤(2)。

算法中节点 x 的估价函数 $f(x)$的计算方法是

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

至于 $h(x)$的计算公式则需由具体问题而定。

可以看出，A 算法其实就是对于本节开始给出的图搜索一般算法中的树式搜索算法，再增加了估价函数 $f(x)$的一种启发式搜索算法。

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

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

其中，$h^*(x)$是从节点 x 到目标节点的最小启发函数值(也可以是代价)，即最佳路径上的实际启发函数值(若有多个目标节点则为其中最小的一个)，则这样的 A 算法被称为 A* 算法。

在 A* 算法中，限制 $h(x)\leqslant h^*(x)$的原因是为了保证取得最优解。理论分析证明，如果问题存在最优解，则这样的限制就可以保证能找到最优解，虽然这个限制可能产生无用搜索。实际上，不难想象，当某一节点 x 的 $h(x)>h^*(x)$时，该节点就可能失去优先扩展的机会，因而导致得不到最优解。

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。

3.1.7　状态图搜索策略小结

将上述的状态图搜索策略归纳如下(见图 3-10)。

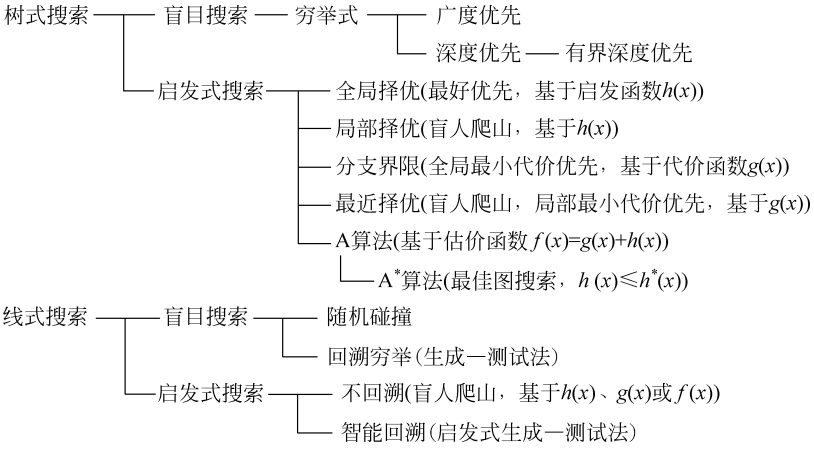

图 3-10　状态图搜索策略归类图

3.2 状态图搜索问题求解

本节我们就用状态图搜索技术解决有关的实际问题。事实上，许多实际问题（如：规划、设计、诊断、控制、预测、决策、证明等）都可以表示为或归结为状态图搜索问题。

3.2.1 问题的状态图表示

1. 状态

状态就是状态图中的节点。状态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，一般用一组数据表示。在程序中，状态用字符、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、法则、行为、变换、关系、函数、算子、过程等。状态转换规则也称为操作，问题的状态也只能被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，在程序中则可用数据对、条件语句、规则、函数、过程等实现。

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，被称为状态空间。所以，状态图也被称为状态空间图。

例 3-4 迷宫问题的状态图表示。

对于迷宫问题，可以每个格子作为一个状态，并用标识符作为其表示。那么，两个标识符组成的序对就是一个状态转换规则。该迷宫的状态图表示如下。

S：S_o

F：$\{(S_o,S_4),(S_4,S_o),(S_4,S_1),(S_1,S_4),(S_1,S_2),(S_2,S_1),$
$(S_2,S_3),(S_3,S_2),(S_4,S_7),(S_7,S_4),(S_4,S_5),(S_5,S_4),(S_5,S_6),$
$(S_6,S_5),(S_5,S_8),(S_8,S_5),(S_8,S_9),(S_9,S_8),(S_9,S_g)\}$

G：S_g

例 3-5 用状态图表示八数码问题。

首先，将棋盘中的 8 个格子分别用变量表示如下：

X_1	X_2	X_3
X_8	X_0	X_4
X_7	X_6	X_5

并用向量

$$\boldsymbol{A}=(X_0,X_1,X_2,X_3,X_4,X_5,X_6,X_7,X_8)$$

表示棋盘。那么，一个棋局就可以表示为变量 $X_i(i=0,1,\cdots,8)$ 的一组取值；反之，变量 X_i 的一组合法取值也就代表了一个棋局。这样，向量 $\boldsymbol{A}$ 就是该问题的状态表达式。

设初始状态和目标状态分别为

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

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

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

$$r_1:\quad (X_0==0)\wedge(X_2==n)\rightarrow(X_0=n)\wedge(X_2=0)$$

$$r_2:\quad (X_0==0)\wedge(X_4==n)\rightarrow(X_0=n)\wedge(X_4=0)$$

$$r_3:\quad (X_0==0)\wedge(X_6==n)\rightarrow(X_0=n)\wedge(X_6=0)$$

$$r_4:\quad (X_0==0)\wedge(X_8==n)\rightarrow(X_0=n)\wedge(X_8=0)$$

1 组规则

$$r_5:\quad (X_1==0)\wedge(X_2==n)\rightarrow(X_1=n)\wedge(X_2=0)$$

$$r_6:\quad (X_0==0)\wedge(X_8==n)\rightarrow(X_0=n)\wedge(X_8=0)$$

2 组规则

$$r_7:\quad (X_2==0)\wedge(X_1==n)\rightarrow(X_2=n)\wedge(X_1=0)$$

$$r_8:\quad (X_2==0)\wedge(X_3==n)\rightarrow(X_2=n)\wedge(X_3=0)$$

$$r_9:\quad (X_2==0)\wedge(X_0==n)\rightarrow(X_2=n)\wedge(X_0=0)$$

……

8 组规则

$$r_{22}:\quad (X_8==0)\wedge(X_1==n)\rightarrow(X_8=n)\wedge(X_1=0)$$

$$r_{23}:\quad (X_8==0)\wedge(X_0==n)\rightarrow(X_8=n)\wedge(X_0=0)$$

$$r_{24}:\quad (X_8==0)\wedge(X_7==n)\rightarrow(X_8=n)\wedge(X_7=0)$$

则八数码问题可用状态图表示为

$$(\{S_o\},\{r_1,r_2,\cdots,r_{24}\},\{S_g\})$$

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，上述 24 条规则也可以简化成 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了。

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被称为隐式状态图，或者说状态图的隐式表示。

例 3-6 梵塔问题的状态图表示。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，他要求僧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。现在考虑梵塔问题是否能用状态图表示。

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。如果直接考虑原问题，则过于复杂。为了便于分析，这里仅考虑二阶梵塔（即只有两个金盘）问题。

设有 3 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。

我们用二元组(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：

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

其实际状态如图 3-11 所示。

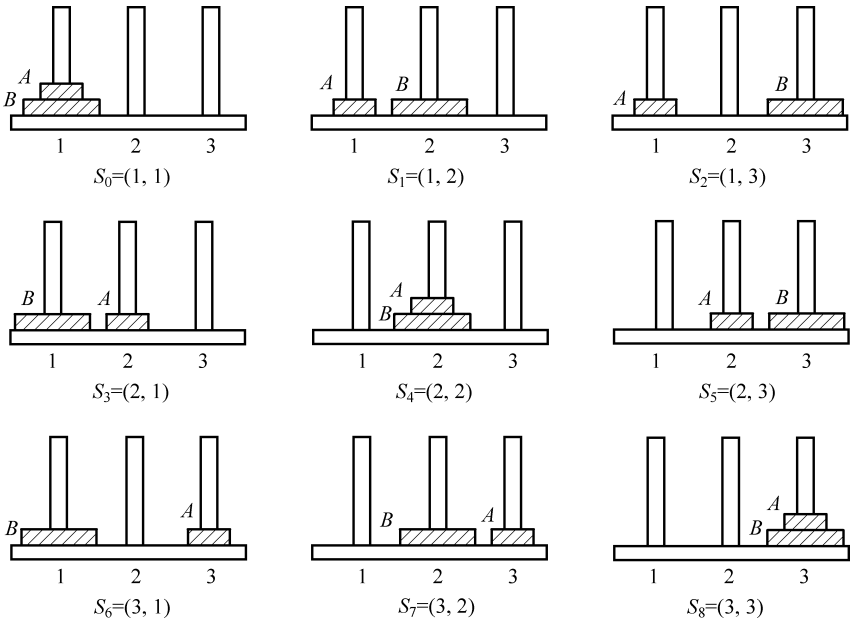

图 3-11 二阶梵塔的全部状态

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〉THEN $A(i,j)$

IF 〈条件〉THEN $B(i,j)$

由题意可知，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：

$$(\{(1,1)\},\{A(1,2),\cdots,B(3,2)\},\{(3,3)\})$$

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可如图 3-12 所示。

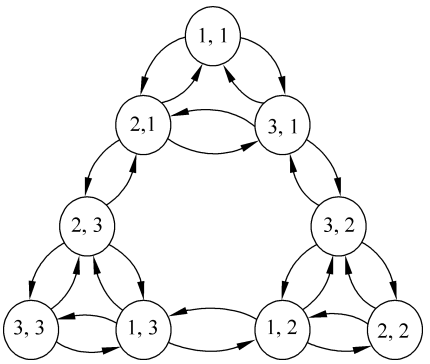

图 3-12　二阶梵塔状态空间图

例 3-7　旅行商问题(Traveling-Salesman Problem，TSP)的状态图表示。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。这就是旅行商问题。下面我们考虑旅行商问题能否用状态图表示。

仔细分析旅行商问题不难发现，虽然问题中的旅行商是在交通图上移动，但该问题的状态并非交通图上的节点，而应为以 A 打头的已访问过的城市序列：$A\cdots$。这样，

$$S_0: A。$$
$$S_g: A,\cdots,A。$$

其中“…”为其余 $n-1$ 个城市的一个序列。

从而，状态转换规则为：

r_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；

r_2：如果所有城市都去过一次，则从当前城市返回 A 城把 A 也添加在去过的城市名序列后端。

3.2.2　状态图问题求解程序举例

例 3-8　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)),                     % 步骤 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),                       % 步骤 2 若 OPEN 表空,则失败,退出
   retract(open(State,Pointer)),              % 步骤 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_).
                                              % 步骤 4 若当前节点为目标节点,则成功
step4(_,State): - mark(End),equal(State,End). % 转步骤 2
step4(No,State): - step56(No,State),!,fail.
step56(No,StateX): -                          % 步骤 5 若当前节点不可扩展,转步骤 2
        rule(StateX,StateY),                  % 步骤 6 扩展当前节点 X 得 Y
        not(open(StateY,_)),                  % 考察 Y 是否已在 OPEN 表中
```

```
        not(closed(_,StateY,_)),              % 考察 Y 是否已在 CLOSED 表中
        assertz(open(StateY,No)),             % 可改变搜索策略
        fail.
step56(_,_): -!.
equal(X,X).
repeat.
repeat: -repeat.
resulting(N): -closed(N,X,M),asserta(res(X)),resulting(M).
resulting(_): -res(X),write(X),nl,fail.
resulting(_): -!.
rule(X,Y): -<问题中的状态转换规则>.          % 例如: rule(X,Y): -road(X,Y).
```

例 3-9　迷宫问题的求解程序。

下面仅给出初始状态、目标状态和状态转换规则集，搜索程序用例 3-8 的通用程序。

```
DOMAINS
   State = symbol
CLAUSES
   solve: - search(a,e),result.
/* 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)集,需并入通用程序后 */
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 3-10　八数码问题的求解程序。

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 3-8 的通用程序，即得八数码问题的求解程序。

```
DOMAINS
state = st(integer,integer,integer,integer,integer,integer,integer,integer,integer)
CLAUSES
solve: -search(st(0,1,2,3,4,5,6,7,8),st(0,2,8,3,4,5,6,7,1)),result.
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)集,需并入通用程序后 */
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 3-11 旅行商问题的求解程序。

```
                /* 旅行商问题 */
DOMAINS
    State=st(lists,integer)
    lists=symbol*
    Gx,Grule,Fx=integer
    city1,city2=symbol
    distance=integer
    StartingCity=symbol
    CitySum=integer
DATABASE-mydatabase
    open(State,integer,Gx,Fx)
    closed(integer,State,integer,Gx)
    open1(State,integer,integer,integer)
    min(State,integer,integer,integer)
    mark(string,integer)
    minD(integer)
    fail_
PREDICATES
    road(city1,city2,distance)
    search(StartingCity,CitySum)
    searching
    step4(integer,State,Gx)
    step56(integer,State,Gx)
    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)
    mindist(integer)
    mindist1
    pa(integer)
    result
GOAL
```

```
    clearwindow,
    write("Please inout starting city name:"),
    readln(Start),
    write("Please input the sum of citys in the map:"),
    readint(Sum),
    search(Start,Sum),
    result.
CLAUSES
search(StartingCity,CitySum): -
    retractall(_,mydatabase),assert(closed(0,st([],0),0,0)),
    assert(open(st([StartingCity],0),0,0,0)),
    assert(mark(StartingCity,CitySum)),
    repeat,
    searching,!.
searching: -
    open(State,BackPointer,Gx,_),
    retract(open(State,_,_,_)),
    closed(No,_,_,_),No2 = No + 1,
    asserta(closed(No2,State,BackPointer,Gx)),
    !,step4(No2,State,Gx).
searching: - assert(fail_).
 result: - not(fail_),closed(_,st(L,_),_,G),write(L,G).
 result: - beep,write("sorry don't find a road!").
 step4(_,st(L,N),_): - mark(_,StateSum),N = StateSum.
 step4(No,State,Gx): - step56(No,State,Gx),!,fail.
 step56(No,st(L,N),Gx): -                    % Gx 为当前节点的代价
     rule(st(L,N),StateY,Grule),             % Grule 为规则的代价(即边代价)
     not(open(StateY,_,_,_)),                % StateY 为扩展得到的子节点
     not(closed(_,StateY,_,_)),
     calculator(N,Gx,Grule,Gy,Fy),
     asserta(open(StateY,No,Gy,Fy)),
     fail.
step56(_,_,_): - sort,!.                     % 按估价函数值对 OPEN 表以升序排序
calculator(N,Gx,Grule,Gy,Fy): -
              Gy = Gx + Grule,               % 计算子节点的代价值 g(y)
              mark(_,CitySum),
              mindist(MinD),
              Hy = (CitySum - N - 1) * MinD,  % 计算子节点的启发函数值 h(y)
              Fy = Gy + Hy,!.                % 计算子节点的估价函数值 f(y) = g(y) + h(y)
mindist(MinD): -
        road(_,_,D1),assert(minD(D1)),mindist1,minD(MinD),!.
mindist1: - road(_,_,D),pa(D),fail.
mindist1: - !.
pa(D): - minD(Do),Do > D,retract(minD(_)),assert(minD(D)),!.
pa(_): - !.
sort: - not(open(_,_,_,_)),!.
sort: - repeat,open(X,N,G,F),assert(min(X,N,G,F)),p1,not(open(_,_,_,_)),p2.
p1: - open(X,N,G,F),p12(X,N,G,F),fail.
p1: - min(X,N,G,F),
```

```
  assertz(open1(X,N,G,F)),retract(open(X,N,G,F)),retract(min(_,_,_,_)),!.
p12(_,_,G,Fn): - min(_,_,_,Fo),Fo<=Fn,!.
p12(X,N,G,Fn): - retract(min(_,_,_,_)),assert(min(X,N,G,Fn)),!.
p2: - open1(X,N,G,F),assertz(open(X,N,G,F)),fail.
p2: - retractall(open1(_,_,_,_)),!.
repeat.
repeat: - repeat.
member(X,[X|_]).
member(X,[_|Y]): - member(X,Y).
append([],L,L).
append([H|T],L,[H|Tn]): - append(T,L,Tn).
rule(st([H|T],IN),st(OL,ON),Grule): -      % 状态变换规则 1
                              mark(StartingCity,StateSum),
                              IN=StateSum-1,
                              road(H,StartingCity,D),
                              append([StartingCity],[H|T],OL),
                              ON=IN+1,
                              Grule=D.
rule(st([H|T],IN),st(OL,ON),Grule): -      % 状态变换规则 2
                              road(H,Y,D),
                              not(member(Y,[H|T])),
                              append([Y],[H|T],OL),
                              ON=IN+1,
                              Grule=D.
/* 交通图 (如)
road(xian,beijing,1165).
road(xian,shanghai,1511).
    …
*/
```

可以看出,该程序与例 3-8 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。其中代价的计算公式为

节点$(A\cdots XY)$ 的代价 = 起始城市到 X 城的距离 + X 城到 Y 城的距离

启发函数值的计算公式为

节点$(A\cdots XY)$ 的启发值 =(城市总数 − 已访问过的城市数 − 1)×
× min{所有两城间的距离}

这里把一个节点的启发函数值定义为该节点到目标节点的距离下限(相当于至少还要花费的代价)。那么,随着访问城市数的增加,启发函数值则在逐渐减少。式中减 1 的原因是每次计算时,总是对刚才扩展到的子节点计算的,而该节点还未计入已扩展数中。

由于这个启发函数值的实际值($h^*(x)$)总不会小于所有城市间最小距离的整倍数($h(x)$),所以,符合 A* 算法的要求。代价值和启发值在搜索过程中的处理差别是,前者要不断进行传递和累加,而后者只是在需要时临时计算,且不进行传递和累计。

该程序实际是一个旅行商问题的通用程序。对于一个具体的旅行路径规划,只需把

具体的“交通图”用谓词 road(City1,City2,Cost)描述出来,并作为事实并入该程序。

该程序还有一个特点——它实际是进行双重搜索:一方面在显式图(交通图)上进行搜索,同时又在由此产生的隐式图(以访问过的城市序列为状态节点的状态图)上进行搜索。而该问题的解,并不是隐式图中的路径,而是路径中的最后一个节点。这个节点恰好是交通图上的一条路径。

3.3　与或图与与或图搜索

3.3.1　与或图

先用一个几何证明问题引入与或图的概念。

问题　如图 3-13 所示,设有四边形 $ABCD$ 和 $A'B'C'D'$,要求证明它们全等。

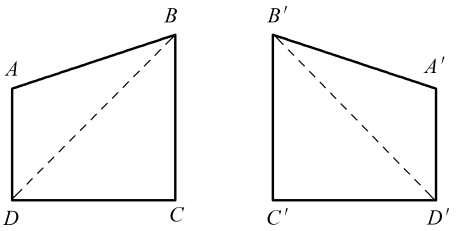

图 3-13　四边形 $ABCD$ 和 $A'B'C'D'$

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$\triangle ABD \cong \triangle A'B'D'$

Q_2:证明$\triangle BCD \cong \triangle B'C'D'$

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。

而问题 Q_1 还可再被分解为

$Q_{1\text{-}1}$:证明 $AB = A'B'$

$Q_{1\text{-}2}$:证明 $AD = A'D'$

$Q_{1\text{-}3}$:证明 $\angle A = \angle A'$

或

$Q'_{1\text{-}1}$:证明 $AB = A'B'$

$Q'_{1\text{-}2}$:证明 $AD = A'D'$

$Q'_{1\text{-}3}$:证明 $BD = B'D'$

问题 Q_2 还可再被分解为

$Q_{2\text{-}1}$:证明 $BC = B'C'$

$Q_{2\text{-}2}$:证明 $CD = C'D'$

$Q_{2\text{-}3}$:证明 $\angle C = \angle C'$

或

$Q'_{2\text{-}1}$:证明 $BC = B'C'$

$Q'_{2\text{-}2}$:证明 $CD = C'D'$

$Q'_{2\text{-}3}$:证明 $BD = B'D'$

现在考虑原问题与这两组子问题的关系,如图 3-14 所示。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**。如图 3-15 所示的则是一个更典型的与或图。

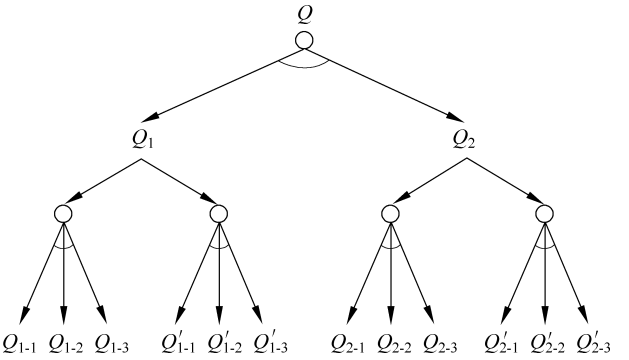

图 3-14　问题的分解与变换

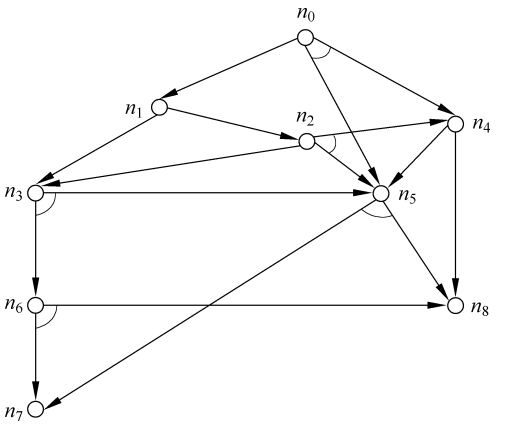

图 3-15　一个典型的与或图

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。

由上面的问题可以看出，与或图可以用来描述一类问题的求解过程。事实上，若把待解的原问题作为初始节点，把由原问题经一系列分解或变换而得到的直接可解的简单问题作为目标节点，那么，问题求解过程也就是在一个与或图中寻找一个从初始节点到目标节点的路径问题。例如，如果把上面的原问题 Q 作为初始节点，把子问题 $Q_{1\text{-}1}$、$Q_{1\text{-}2}$、$Q_{1\text{-}3}$……作为目标节点，则对问题 Q 的求解就是在如图 3-14 所示的与或图中寻找路径的问题。但可以看出，与或图中的路径一般不是状态图中那样的线形路径，而是树形“路径”。因此，一般称这种路径为**解树**或**解图**。所以，求解与或图问题就是在与或图中搜索解树或解图的问题。

同状态图一样，与或图也是问题求解的一种抽象表示。事实上，许多问题的求解过程都可以用与或图搜索来描述。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。所以，研究与或图搜索也具有普遍意义。

用与或图搜索来描述问题的求解过程，首先，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(即归约为)一些直接可解的子问题,或者不可解也不能再分解或变换的子问题为止。然后,根据所得到的搜索树确定原问题的可解性。如果可解,则由搜索树找出解图或解树。

下面再引入与或图搜索中的几个基本概念。直接可解的简单问题被称为**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**,一个节点的子节点间如果是"与"关系,则该节点便称为**与节点**,一个节点的子节点间如果是"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。)

3.3.2　与或图搜索

1. 搜索方式,解树(图)

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。但在与或图中搜索解树(图),不像在或图中那样只是简单地寻找目标节点,而是边扩展节点边进行逻辑判断,以确定初始节点是否可解。一旦能够确定初始节点的可解性,则搜索停止。这时,如果初始节点可解,则根据返回指针便可从搜索树中得到一个解树(图)。所以,准确地说,解树(图)实际上是由可解节点形成的一个子树(图),这个子图(树)的根为初始节点,叶为终止节点,且这个子树(图)还一定是与树(图)。

2. 可解性判别

怎样判断一个节点的可解性呢?下面给出判别准则。

(1) 一个节点是可解节点,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。盲目搜索可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,一般的与或图搜索也涉及一些复杂的处理。因篇幅所限,这里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。

(1) 把初始节点 Q_0 放入 OPEN 表。

(2) 移出 OPEN 表的第一个节点 N，将其放入 CLOSED 表，并冠以序号 n。

(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，然后由它们的可解反向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)，转步骤(2)。

(4) 若 N 不可扩展，则做下列工作：

① 标记 N 为不可解节点，然后由它的不可解反向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)，转步骤(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 3-12 设有与或树如图 3-16 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(优先)搜索策略，搜索过程如下。

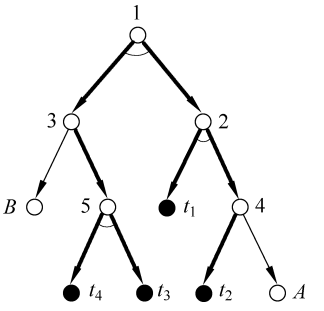

图 3-16 与或树及其解

(1) 扩展 1 号节点得到 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) 扩展 2 号节点后得到 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。由于 t_1 是终止节点，故标记它为可解节点，并将它放入 CLOSED 表，再判断其先辈节点的可解性，但 t_1 的父节点 2 是一个与节点，故仅由 t_1 的可解还不能确定 2 号节点可解。所以，就继续搜索。

(3) 扩展 3 号节点得到 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) 扩展 4 号节点后得到节点 A 和 t_2。t_2 是终止节点，标记为可解节点，放入 CLOSED 表。这时其先辈节点 4 和 2 也为可解节点，但 1 号节点还不能确定。这时从 OPEN 表中删去节点 A，因为其父节点 4 已经可解。

(5) 扩展 5 号节点得到 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。

这时，由 CLOSED 表便得到由节点 1、2、3、4、5 和 t_1、t_2、t_3、t_4 构成的解树，如图 3-16 中的粗线所示。

3.3.3　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。像这样根据代价决定搜索路线的方法被称为与或树的**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 Q_0。也就是说，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，有下面几种代价计算方法。

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，则 $g(x)=0$。

(2) 若 x 是或节点，则 $g(x)=\min\limits_{1\leqslant i\leqslant n}\{c(x,y_i)+g(y_i)\}$，其中 $y_1,y_2,\cdots,y_n$ 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：$g(x)=\sum\limits_{i=1}^{n}\{c(x,y_i)+g(y_i)\}$

② 最大代价法：$g(x)=\max\limits_{1\leqslant i\leqslant n}\{c(x,y_i)+g(y_i)\}$

其中，$y_1,y_2,\cdots,y_n$ 是 x 的子节点。

(4) 对非终止的端节点 x，$g(x)=\infty$。

例 3-13　如图 3-17 所示的与或树，可以发现其中包括两棵解树，一棵解树由 Q_0、A、t_1 和 t_2 组成；另一棵解树由 Q_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 和 F 是非终止的端节点；各边上的数字是相应的代价。

由右边的解树，

按和代价：$g(A)=11$，　$g(Q_0)=13$

按最大代价：$g(A)=6$，　$g(Q_0)=8$

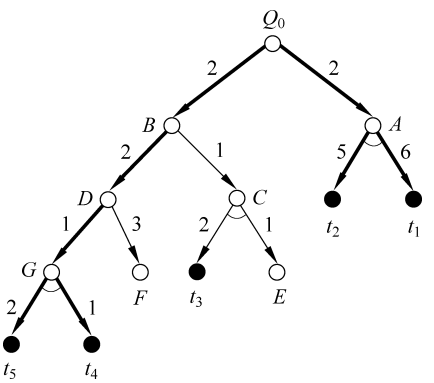

图 3-17　含代价的与或树

由左边的解树，

按和代价：$g(G)=3$，　$g(D)=4$，　$g(B)=6$，　$g(Q_0)=8$

按最大代价：$g(G)=2$，　$g(D)=3$，　$g(B)=5$，　$g(Q_0)=7$

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 x 的代价 $g(x)$如何计算呢？解决的办法是，根据问题本身提供的启发性信息定义一个启发函数，由启发函数估算出子节点 y_i 的代价 $g(y_i)$，然后再按和代价或最大代价算出节点 x 的代价值 $g(x)$。有了 $g(x)$，节点 x 的父节点、祖父节点以及直到初始节点 S_0 的各先辈节点的代价 g 都可自下而上地逐层推算出来。

当节点 y_i 被扩展后，也是先用启发函数估算出其子节点的代价，然后再算出 $g(y_i)$。此时算出的 $g(y_i)$可能与原先估算出的 $g(y_i)$不相同，这时应该用后算出的 $g(y_i)$取代原先估算出的 $g(y_i)$，并且按此 $g(y_i)$自下而上地重新计算各先辈节点的 g 值。当节点 y_i 的子节点又被扩展时，上述过程又要重复进行一遍。总之，每当有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的节点，又自下而上地计算代价 g 的反复进行的过程。

有序搜索的目的是求出最优解树，即代价最小的解树。这就要求搜索过程中任一时刻求出的部分解树其代价都应是最小的。为此，每次选择欲扩展的节点时都应挑选有希望成为最优解树一部分的节点进行扩展。由于这些节点及其先辈节点（包括初始节点 S_0）所构成的与或树有可能成为最优解树的一部分，因此称它为“希望树”。

在搜索过程中随着新节点的不断生成，节点的代价值是在不断变化的，因此希望树也在不断变化。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。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，而且希望树总是对最优解树近根部分的某种估计。

下面是希望树的定义。

(1) 初始节点 Q_0 在希望树 T 中。

(2) 如果节点 x 在希望树 T 中，则一定有：

① 如果 x 是具有子节点 $y_1, y_2, \cdots, y_n$ 的“或”节点，则具有值

$$g(x)=\max_{1\leqslant i\leqslant n}\{c(x,y_i)+g(y_i)\}$$

的那个子节点 y_i 也应在 T 中。

② 如果 x 是“与”节点，则它的全部子节点都应在 T 中。

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 Q_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 Q_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 Q_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 Q_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) 转步骤(2)。

例 3-14　下面举例说明上述搜索过程。

设初始节点为 Q_0，每次扩展两层，并设 Q_0 经扩展后得到如图 3-18(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

$$g(B)=3,\quad g(C)=3,\quad g(E)=3,\quad g(F)=2$$

若按和代价计算，则得

$$g(A)=8,\quad g(D)=7,\quad g(Q_0)=8$$

(注：这里的边代价一律按 1 计算，下同。)

此时，Q_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 3-18(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

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

此时，由 Q_0 的右子树算出的 $g(Q_0)=12$。但是，由左子树算出的 $g(Q_0)=9$。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 3-18(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

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

由此可推算出 $g(Q_0)=9$。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 3-18(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

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

由此可推算出 $g(Q_0)=9$。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。再由已推出的可解节点 B，可推出 A 和 Q_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 3-18(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。

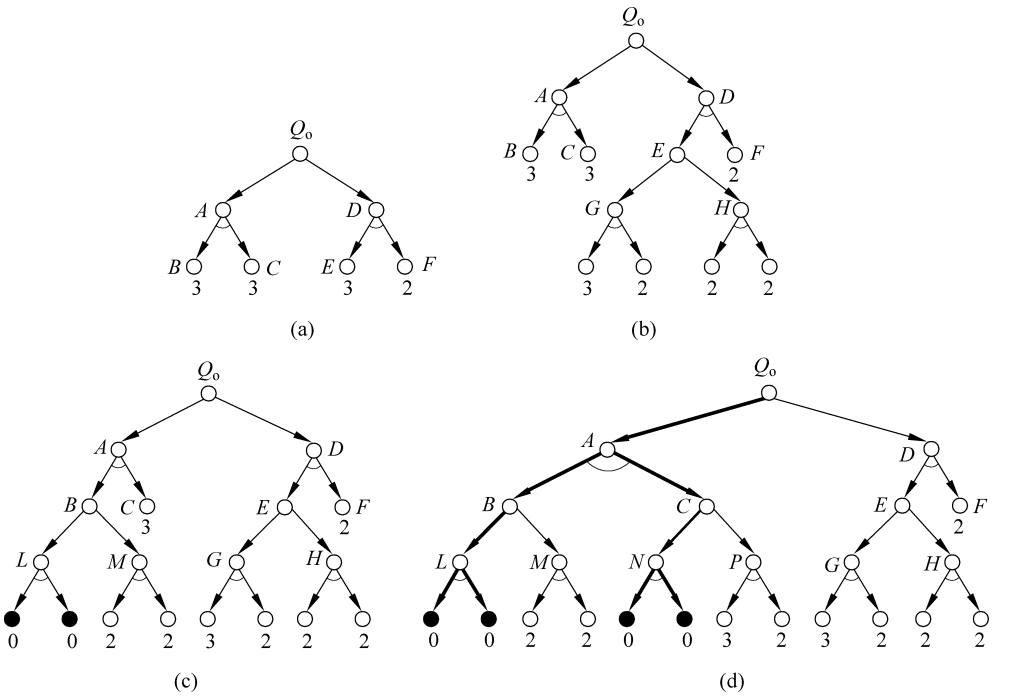

图 3-18　与或树有序搜索示例

3.4　与或图搜索问题求解

3.4.1　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，一般表示问题的变换过程(而不是状态变换过程)。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。所以，与或图的节点一般代表问题，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。

与或图也可表示为一个三元组

$$(Q_0, F, Q_n)$$

在这里 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。其实，一个 PROLOG 程序也就是一个与或图。程序中的询问（即目标）就是初始问题，规则就是问题变换规则，事实就是本原问题。下面再举几个例子。

例 3-15　三阶梵塔问题的与或图表示。

对于梵塔问题，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 3 个子问题。这 3 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。对于第一和第三这两个子问题，可用上面 n 个盘子的方法做同样的处理。根据这一思想，可把三阶梵塔问题分解为下面的 3 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）和（3）又分别可分解为 3 个子问题。

于是，得到三阶梵塔问题的与或树表示，如图 3-19 所示。

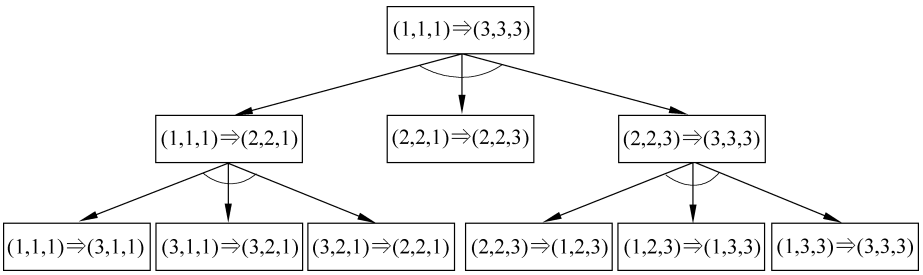

图 3-19　三阶梵塔问题的与或树

图中的三元组 (i, j, k) 中的 i 代表金盘 A 所在的杆号，j 代表金盘 B 所在的杆号，k 代表金盘 C 所在的杆号。这个与或树中，共有 7 个终止节点，对应于 7 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：

$$\begin{aligned}
(1,1,1) &\Rightarrow (3,1,1)\\
(3,1,1) &\Rightarrow (3,2,1)\\
(3,2,1) &\Rightarrow (2,2,1)\\
(2,2,1) &\Rightarrow (2,2,3)\\
(2,2,3) &\Rightarrow (1,2,3)\\
(1,2,3) &\Rightarrow (1,3,3)\\
(1,3,3) &\Rightarrow (3,3,3)
\end{aligned}$$

视频讲解

此例说明,有些问题既可用状态图表示,也可用与或图表示。事实上,任一个状态图都可以转化为一个与或图。读者从上面的两个梵塔问题中不难看出其转化方法。

3.4.2 与或图问题求解程序举例

例 3-16 基于与或图搜索的迷宫问题求解程序。

```
/* 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).
```

可以看出,该程序只给出了问题分解规则,即与或树,而搜索程序则是利用了PROLOG 自身的解释程序。这正是用 PROLOG 解决此类问题的特点。该程序执行时也可回溯,且用 PROLOG 的表记录了搜索路径,所以它又是一种可回溯的线式搜索程序。

例 3-17 梵塔问题求解程序。

基于例 3-15 中对于梵塔问题的分析,可得递归程序如下:

```
/* Hanoi tower */
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。

3.5　博弈树搜索*

诸如下棋、打牌、竞技、战争等竞争性智能活动被称为博弈，而其中最简单的称为“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3种情况，即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。

3.5.1　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。此时，如果站在A方的立场上，则可供A方选择的若干行动方案之间是“或”关系，因为主动权掌握在A方手里，他或者选择这个行动方案，或者选择另一个行动方案，完全由A方自己决定。当A方选取任一方案走了一步后，B方也有若干个可供选择的行动方案，此时这些行动方案对A方来说它们之间则是“与”关系，因为这时主动权掌握在B方手里，这些可供选择的行动方案中的任何一个都可能被B方选中，A方必须应付每一种情况的发生。

这样，如果站在某一方(如A方，即在A方要取胜的意义下)，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。

3.5.2　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。其基本思想如下：

(1) 设博弈的双方中一方为A，另一方为B。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。具体地说，就是要考虑每一方案实施后对方可能采取的所有行动，并计算可能的得分。

(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。此时估算出来的得分被称为静态估值。

(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。这样计算出的父节点的得分被称为**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。

如图 3-20 所示给出了计算倒推值的示例。

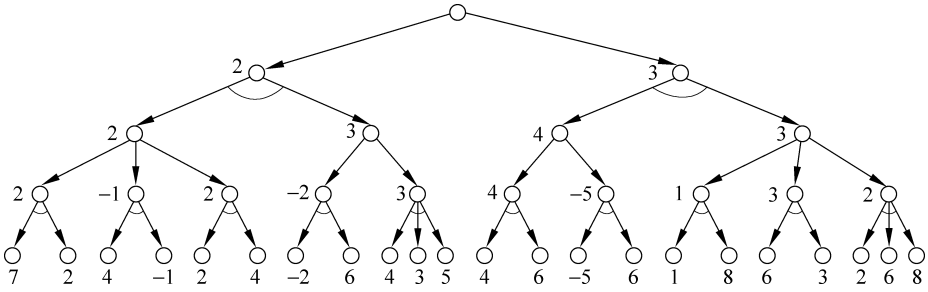

图 3-20 倒推值的计算示例

在博弈问题中,每一个格局可供选择的行动方案都有很多,因此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。所以,试图利用完整的博弈树来进行极小-极大分析是有困难的。可行的办法是只生成一定深度的博弈树,然后进行极小-极大分析,找出当前最好的行动方案。在此之后,在已选定的分支上扩展一定深度,再选最好的行动方案。如此进行下去,直到取得胜败的结果为止。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,需根据实际情况而定。

例 3-18 一字棋游戏。设有如图 3-21 所示的 9 个空格,由 A、B 二人对弈,轮到谁走棋谁就往空格上放一枚自己的棋子,谁先使自己的棋子构成“三子成一线”,谁就取得了胜利。

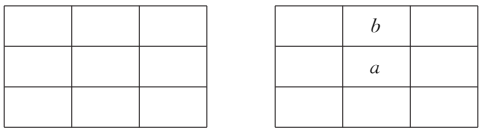

图 3-21 一字棋

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。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 $e(P)=+\infty$。

(2) 若 P 是 B 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 a 成为三子一线的数目;$e(-P)$表示棋局 P 上有可能使 b 成为三子一线的数目。对于如图(b)所示的棋局,则

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

另外,假定具有对称性的两个棋局算作一个棋局且 A 先走棋。图 3-22 就是为了 A 的第一着走棋而生成的博弈树。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 有更大的倒推值。

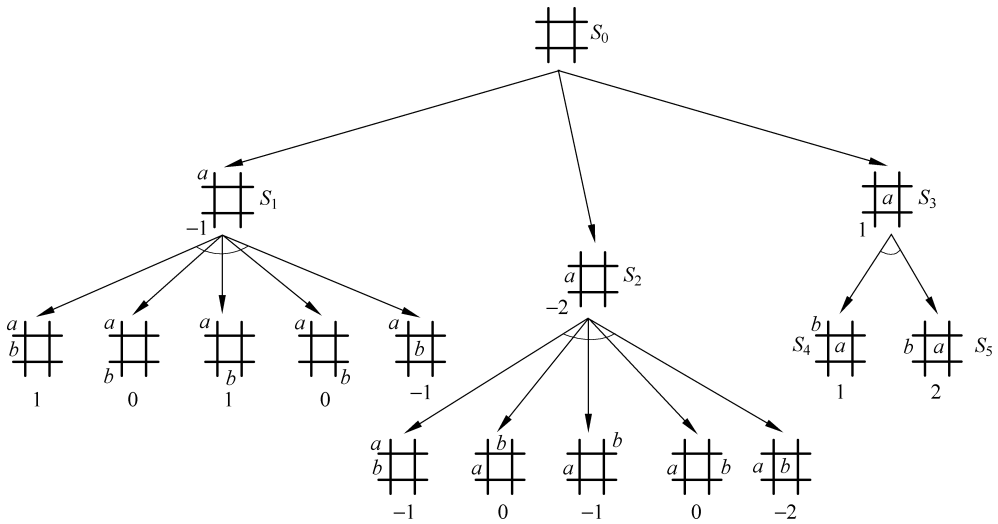

图 3-22　一字棋极小-极大搜索示例

在 A 走了 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 或 S_5,A 都要再次运用极小-极大分析法产生深度为 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,不再赘述。

3.5.3　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。

这一技术的基本思想是,边生成博弈树边计算评估各节点的倒推值,并且根据评估出的倒推值范围,及时停止扩展那些已无必要再扩展的子节点,即相当于剪去了博弈树上的一些分枝,从而节约了机器开销,提高了搜索效率。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 $\alpha \geqslant \beta$,则不必再扩展该 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)。这一过程被称为 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 $\alpha \geqslant \beta$,则不必再扩展该 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)。这一过程被称为 β 剪枝。

例 3-19 图 3-23 所示的博弈树搜索就采用了 α-β 剪枝技术。

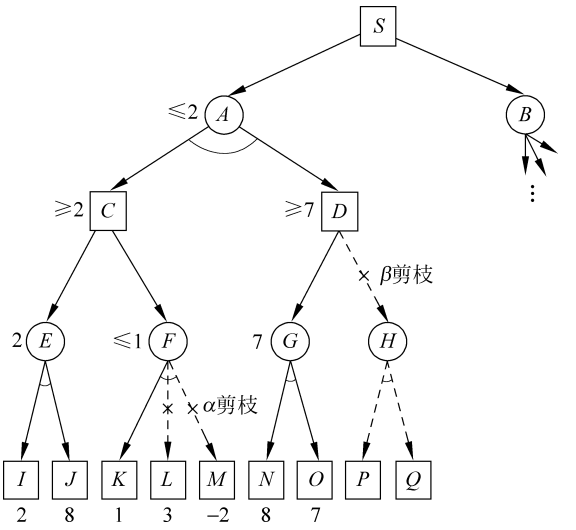

图 3-23 α-β 剪枝示例

习题 3

1. 什么是状态图、与或图? 图搜索与问题求解有什么关系?

2. 什么是状态图问题的解? 什么是最优解?

3. 综述状态图搜索的方式和策略。

4. 设有三只琴键开关一字排开,初始状态为“关、开、关”,问连按三次后是否会出现“开、开、开”或“关、关、关”的状态? 要求每次必须按下一个开关,而且只能按一个开关。另外,画出这个琴键开关的状态空间图。

注:琴键开关有这样的特点,若第一次按下时它为“开”,则第二次按下时它就变成了“关”。

5. 农夫过河问题:有一农夫带一只狼、一只羊和一筐菜欲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。

设计一个过河方案,使得农夫、狼、羊、菜都能不受损失地过河。画出相应的状态变化图。


视频讲解

提示:

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

(2) 把每次过河的一种安排作为一个算符,每次过河都必须有农夫,因为只有他可以划船。

6. 阐述状态空间的一般搜索过程。OPEN 表与 CLOSED 表的作用是什么?

7. 广度优先搜索与深度优先搜索各有什么特点?

8. 图 3-24 是五大城市间的交通示意图,边上的数字是两城市间的距离。用图搜索技术编写程序,求解以下问题:

(1) 任找一条西安到北京的旅行路线,并给出其距离。

(2) 找一条从西安到北京且途经上海的路径。

(3) 找一条从西安到北京且途经上海,但不能去昆明的路径。

9. 什么是估价函数? 在估价函数中,$g(x)$和 $h(x)$各起什么作用?

10. 局部择优搜索与全局择优搜索的相同处与区别各是什么?

11. 对于与或图,什么是与节点? 什么是或节点? 什么是可解节点? 什么是解树?

12. 试用与或树描述下面不定积分的求解过程:

$$\int (x^2 + 5x + \sin^2 x \cos^2 x)\mathrm{d}x$$

13. 什么是与或图问题的解? 什么是最优解?

14. 设有如图 3-25 所示的一棵与或树,请指出解树;分别按和代价及最大代价求解树代价;指出最优解树。

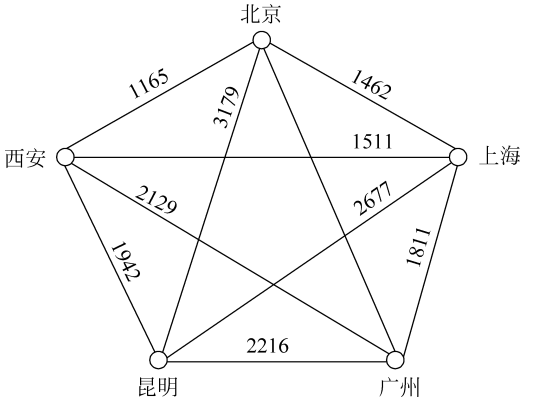

图 3-24　交通图

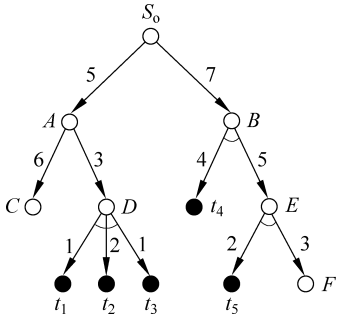

图 3-25　与或树