第3章

图搜索与问题求解

3.1 概述

搜索是人工智能技术中进行问题求解的基本技术，不管是符号智能、连接智能还是计算智能及统计智能和交互智能，不管是解决具体应用问题（如证明、诊断、规划、调度、配置和优化）还是智能行为本身（如学习和识别），最终往往都归结为某种搜索，都要用某种搜索算法来实现。

符号智能中的搜索是运用领域知识，以符号推演的方式，顺序地在问题空间中进行的，其中的问题空间又可表示为某种状态图（空间）或者与或图的形式。所以，这种搜索也被称为图搜索。图搜索技术是人工智能中发展最早的技术，已取得了丰硕成果。图搜索模拟的实际是人脑分析问题、解决问题的过程，它是基于领域知识的问题求解技术。

本章主要介绍状态图（空间）搜索技术及相应的问题求解。

3.2 状态图与状态图搜索

3.2.1 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 3-2 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，或者是在该有向图中寻找从初始节点到目标节点路径的问题。

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，另有一个空格。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

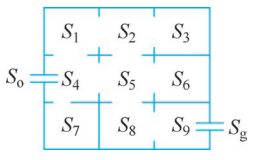

图 3-1 迷宫图

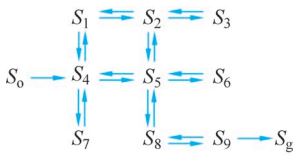

图 3-2 迷宫的有向图表示

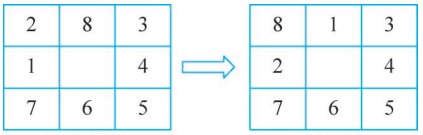

图 3-3　八数码问题示例

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 3-3 所示)，给出数码的移动序列。该问题被称为八数码问题或重排九宫问题。

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，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)对应一次数码移动；反之，一次数码移动也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，于是，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。在人工智能技术中，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。

3.2.2　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)。所以当目标节点找到后，路径也就找到了，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，便会构成一个树状的有向图。这种树状有向图称为**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找出从初始节点到目标节点的路径。为此，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,则搜索成功;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,因此要提高搜索效率(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。常识告诉我们,如果有向导引路,则会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(瞎子爬山法)、最佳图搜索等。

按搜索范围扩展顺序的不同,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,因此在搜索过程中就得随时记录搜索轨迹。为此,用一个称为 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)。

此外,对于树式搜索来说,还得不断地把待考察的节点以某种顺序组织在一起,以便控制搜索的方向和顺序。为此,采用一个称为 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。

CLOSED 表

编号	节点	父节点编号

OPEN 表

节点	父节点编号

图 3-4　CLOSED 表与 OPEN 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，则转步骤(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)也将之删除；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(如图 3-5 所示)。

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。

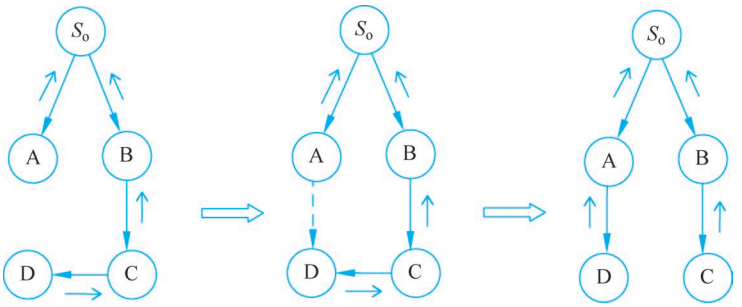

图 3-5　状态图搜索过程中修改返回指针示例

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，转步骤(2)。

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，因为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，当新路径短时自然要走新路径。

(3) 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还要修改相应的代价值，或者不修改返回指针也要修改代价值(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)。

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，则还需要由 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 表即是所找路径。

3.2.3　穷举式搜索

下面先讨论树状结构的状态图搜索，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。

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，则可得到如图 3-6 所示的搜索树。

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)。

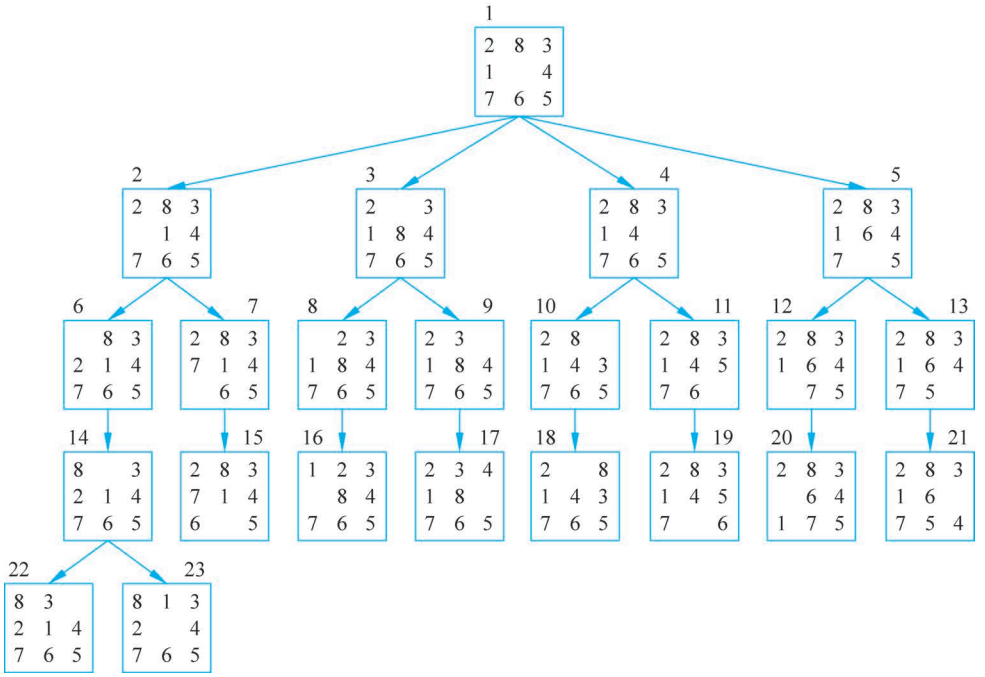

图 3-6 八数码问题的广度优先搜索

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，则用它一定能找到解，且找到的解还是最优解（即最短的路径）。这是广度优先搜索的优点。但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 表为一个栈，这是与广度优先算法的唯一区别。

例 3-2　对于前面的八数码问题，应用深度优先搜索策略，可得如图 3-7 所示的搜索树。

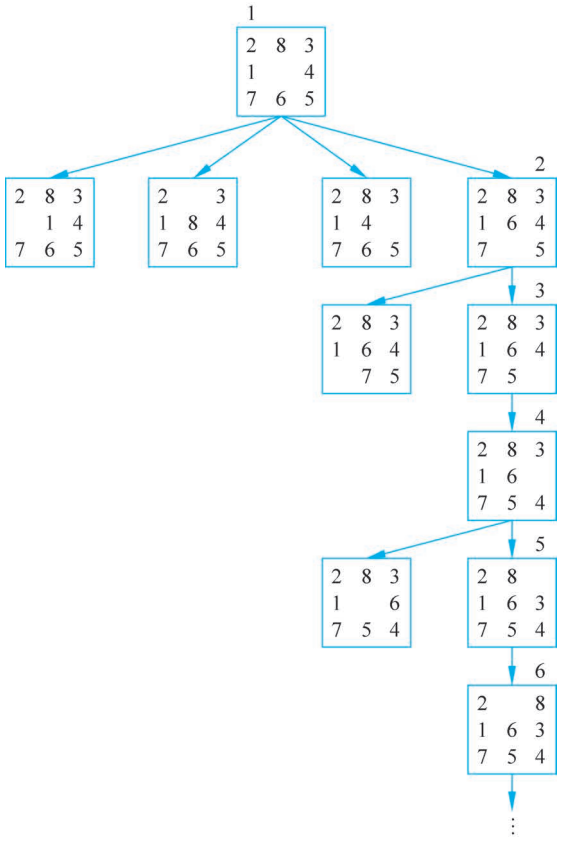

图 3-7　八数码问题的深度优先搜索

深度优先搜索亦称为纵向搜索。由于一个有解的问题树可能含有无穷分支，深度优先搜索如果误入无穷分支（即深度无限），则不可能找到目标节点。因此，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）。

3.2.4　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”。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增加。例如当取阶数为 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，也计算不了的。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} s/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，于是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指导的搜索。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 3 类。

(1) 用于扩展节点的选择，即用于决定应先扩展哪个节点，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 $h(x)$。

启发函数并无固定的模式，如何定义一个启发函数需要根据具体问题来分析。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；根据经验的主观打分等。例如，对于八数码难题，$h(x)$就可定义为节点 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的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。下面给出树状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指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 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如图 3-8 所示。

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。以这个函数指导的搜索树如图 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_o, S_1, S_2, S_3, S_g。

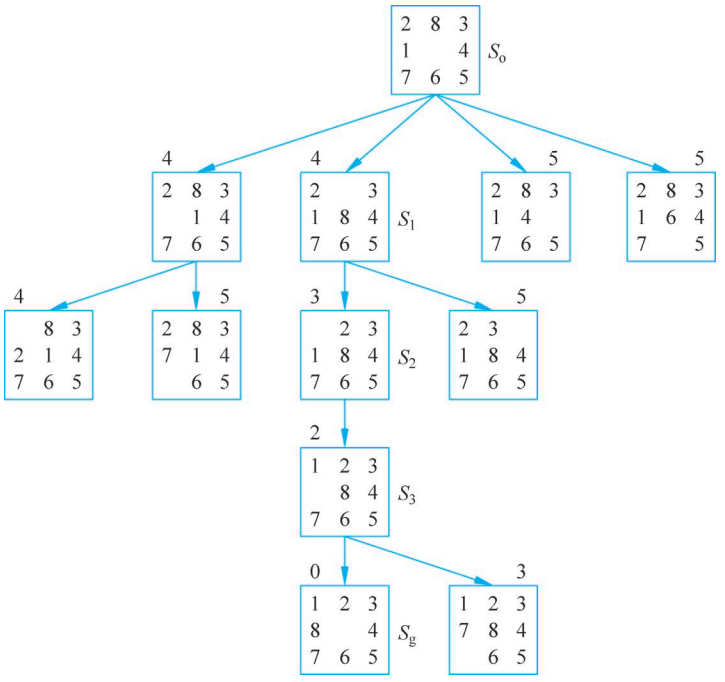

图 3-8　八数码问题的全局择优搜索

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。

3.3　状态图搜索问题求解

状态图搜索可以解决许多没有算法解的实际问题，如规划、设计、诊断、控制、预测、决策、证明等都可以用状态图搜索技术来解决。用状态图搜索技术解决有关的实际问题，首先需要将实际问题表示为状态图搜索问题，然后选用合适的搜索算法进行求解。

3.3.1　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，称为状态空间。所以，状态图也称为状态空间图。

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，该迷宫的状态图问题表示为

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。于是，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

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

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

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

$$r_5:\quad (X_1==0)\wedge(X_2==n)\rightarrow(X_1=n)\wedge(X_2=0)$$
$$r_6:\quad (X_1==0)\wedge(X_8==n)\rightarrow(X_1=n)\wedge(X_8=0)$$

2 组规则

$$r_7:\quad (X_2==0)\wedge(X_1==n)\rightarrow(X_2=n)\wedge(X_1=0)$$
$$r_8:\quad (X_2==0)\wedge(X_3==n)\rightarrow(X_2=n)\wedge(X_3=0)$$
$$r_9:\quad (X_2==0)\wedge(X_0==n)\rightarrow(X_2=n)\wedge(X_0=0)$$
$$\vdots$$

8 组规则

$$r_{22}:\quad (X_8==0)\wedge(X_1==n)\rightarrow(X_8=n)\wedge(X_1=0)$$
$$r_{23}:\quad (X_8==0)\wedge(X_0==n)\rightarrow(X_8=n)\wedge(X_0=0)$$
$$r_{24}:\quad (X_8==0)\wedge(X_7==n)\rightarrow(X_8=n)\wedge(X_7=0)$$

于是，八数码问题可用状态图表示为

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

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，因此上述 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，这时的状态（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。而其余节点需要用状态转换规则来产生。类似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，旁边另外再插上两个宝石杆。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，所以直接考虑原问题，将过于复杂。为了便于分析，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。

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 3-9 所示。

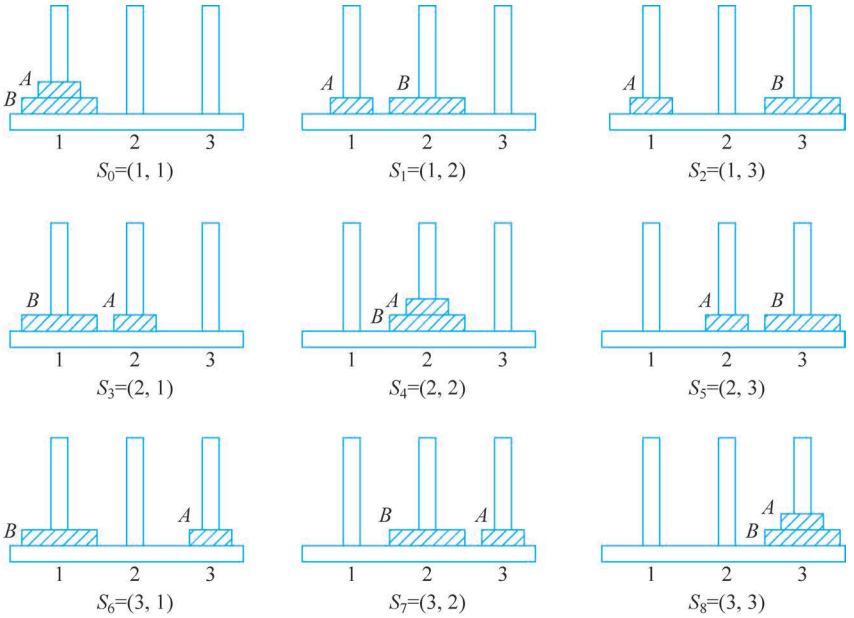

图 3-9 二阶梵塔的全部状态

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

$$A(1,2),\quad A(1,3),\quad A(2,1),\quad A(2,3),\quad A(3,1),\quad A(3,2)$$
$$B(1,2),\quad B(1,3),\quad B(2,1),\quad B(2,3),\quad B(3,1),\quad B(3,2)$$

当然,规则的具体形式应是

$$\text{IF }\langle\text{条件}\rangle\text{THEN } A(i,j)$$
$$\text{IF }\langle\text{条件}\rangle\text{THEN } B(i,j)$$

这样由题意可知,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如图 3-10 所示。

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。下面考虑旅行商问题能否用状态图表示。

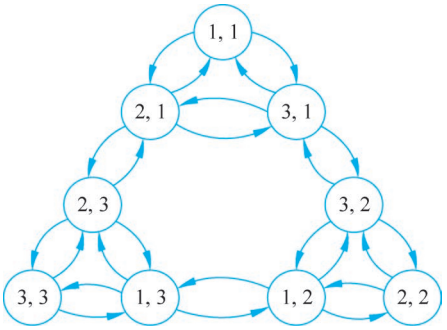

图 3-10　二阶梵塔问题的状态空间图

视频讲解

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，状态转换规则如下。

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 也添在去过的城市名序列后端。

3.3.2　状态图搜索问题求解程序举例

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 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,StateX): -                                  % 步 5 若当前节点不可扩展, 转步 2
        rule(StateX,StateY),                          % 步 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).
```

延伸学习导引

本章介绍了状态图搜索及相应的问题求解方法。关于状态图搜索，进一步还有加权状态图搜索及著名的 A 算法和 A^* 算法等；另外，图搜索技术中还有与或图搜索及博弈树搜索等。这些延伸学习内容，可参见文献[2]或[3]中的第 3 章。

习题 3

1. 何为状态图？状态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，即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，并画出相应的状态变化图。

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。

视频讲解

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. 试述广度优先搜索、深度优先搜索和启发式搜索各有什么特点。