第 3 章　问题求解方法

在求解一个问题时会涉及两方面。一方面是该问题的表示，如果一个问题找不到一个合适的表示方法，对它的求解也就无从谈起。第 2 章介绍了各种知识表示技术，这些技术可以用来表示问题。另一方面则是针对该问题，分析其特征，选择一种相对较合适的方法来进行具体求解。在人工智能中，问题求解的基本方法有搜索法、归约法、归结法、推理法、约束满足法、规划和产生式等，现在常用的有模拟退火法、遗传算法等。本章将主要介绍状态空间搜索及其他一些当前常用的问题求解方法。

3.1　状态空间搜索概述

3.1.1　状态图

走迷宫是一种较为常见的游戏，如果把迷宫的每个格子及入口和出口都作为一个结点，把通道作为边，那么迷宫可以由一个有向图来表示。走迷宫其实是从有向图的初始结点（入口）出发，寻找目标结点（出口）的问题；或者寻找通向目标结点（出口）的路径问题。

八数码问题的 3×3 方格棋盘上放置 1～8 的 8 个数码，每个数码占一格，且有一个空格。这些数码在棋盘上可向相邻的空格移动。如果把每个棋局（数码的放置）作为一个结点，把移动作为边，那么该问题可以由一个有向图来表示。该问题就是由该有向图的初始棋局开始，给出形成目标棋局的数码移动的序列。

抽象地看，上述两个问题都是在某个有向图中寻找目标或路径的问题。在人工智能中，这种描述问题的有向图称为状态空间图，简称状态图。

状态图中的结点代表问题中的一种格局，称为问题的一个状态；边表示结点之间的某种联系，如某种操作、规则、变换、算子、通道或关系等。在状态图中，从初始结点到目标结点的一条路径，或者所找的目标结点，就是相应问题的一个解。根据解题需要，路径解可以表示为边的序列或结点的序列。如走迷宫的解可以是结点序列，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，许多实际问题（如路径规划、定理证明、演绎推理、机器人行动规划等）和智力问题（如梵塔问题、旅行商问题、八皇后问题、传教士和野人问题等）都可以抽象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。

3.1.2 问题在状态空间中的图描述

第 2 章介绍了状态空间的知识表示方法，现在介绍用图的概念来描述状态空间。在问题的状态空间描述中，图是最直观的，属于显式描述。

状态空间可以用有向图描述。如图 3-1 所示，其结点表示状态，结点间的线表示允许使用的操作算子，表示对状态 S_0 允许使用操作算子 O_1、O_2 和 O_3，并分别使 S_0 转换为 S_1、S_2 和 S_3。这样一步步利用操作算子转换，如 $S_{10} \in G$，则 O_2、O_6、O_{10} 就是一个解。

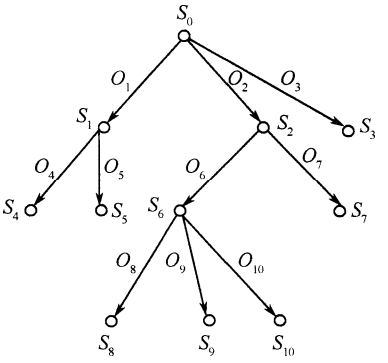

图 3-1　状态空间的有向图描述

这是较为形式化的说明，下面讨论具体问题的状态空间的图描述。

图的结点表示问题的状态，图的线表示状态之间的关系，有向图就是求解问题的步骤。初始状态对应实际问题的已知信息，是图中的根结点。图还可定义目标条件。目标可以描述成一种状态，如一字棋问题中的赢棋或八数码问题中的终局；也可以用解路径本身的特点来描述，如在旅行推销员问题中，当经过图中所有结点，即城市的最短路径找到时，搜索便结束。再如在语法分析问题中，找到了一条对句子进行成功分析的路径便意味着搜索结束。

在问题的状态空间描述中，寻找从一种状态转换为另一种状态的某个操作算子序列就等价于在一个图中寻找某一路径。

在某些问题中，各种操作算子的执行费用是不同的。如在旅行商问题中，两两城市之间的距离通常是不相等的，那么在图中只需给各线标注距离和费用即可。

下面再以八数码问题为例说明简单的状态空间的图描述，其终止条件是某个布局。

【例 3.1】 对于八数码问题，如果给出问题的初始状态，就可以用图来描述其状态空间。其中的线可用表明空格的 4 种可能移动的 4 个操作算子来标注，即将空格向上移（Up）、向左移（Left）、向下移（Down）、向右移（Right），其部分描述如图 3-2 所示。八数码的状态空间图中有回路（因为许多状态有多个父结点）。

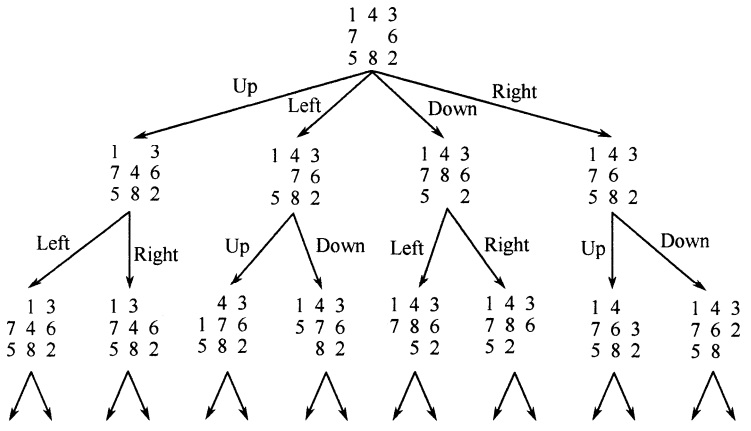

图 3-2　八数码状态空间图（部分）

下面以旅行商为例来说明另一类状态空间的图描述，其终止条件用解路径本身的特点来描述，即经过图中所有城市的最短路径找到时搜索便结束。

【例 3.2】 一个旅行商从家出发，要到 5 个城市去推销产品，然后回家。问题是找到一条最好的路径，使得旅行商访问每个城市后回到出发地全程所走过的路径最短。图 3-3 是这个问题的一个实例，图中结点代表城市，弧上标注的数值表示路径的长度。假定旅行商的家在 *A* 城。

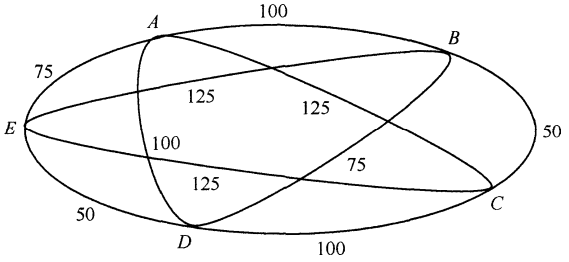

图 3-3　旅行商的一个实例

花费为 475 的路径 *ABDCEA* 就是一个可能的旅行路径，但目标是最短路径。注意，这里对目的的描述是关注整个路径的特性，而不是关注单个状态的特性。

图 3-4 是该问题的部分状态空间。

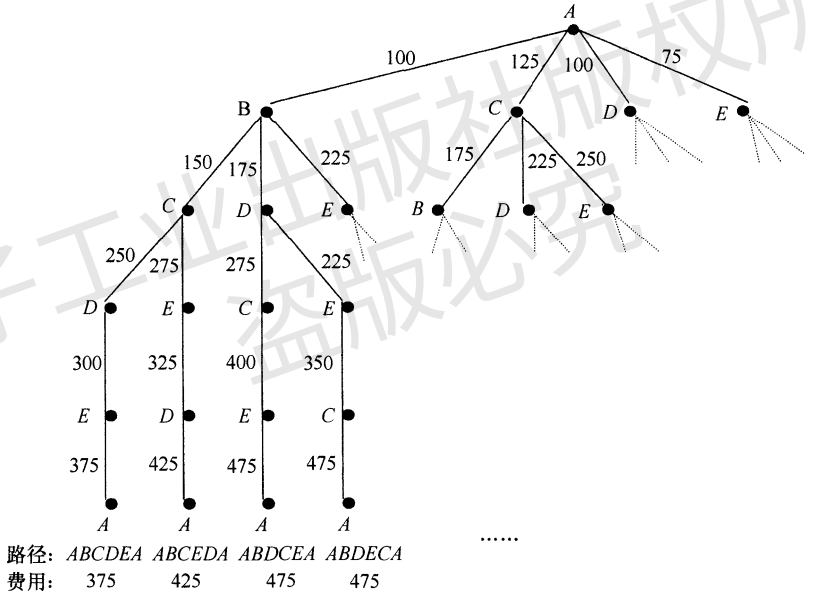

图 3-4　旅行商状态空间图（部分）

上面两个例子中只绘出了问题的部分状态空间图。当然，完全可以绘出问题的全部状态空间图，但对较实际的问题而言，如 80 个城市，要在有限时间内绘出问题的全部状态空间图是不可能的。因此，这类显式描述对于大型问题是不切实际的，而对于具有无限结点集合的问题则是不可能的。注意：一个问题的状态空间是客观存在的，只不过是用 *n* 元组之类的隐式描述而已。

3.1.3　将问题求解定义为状态空间搜索

由于绝大部分人工智能问题缺乏直接求解的方法，因此搜索不失为一种求解问题的一般

机制，构成了人工智能系统的一种框架，其状态空间法构成了人工智能方法的基础。它的结构在如下两方面与问题的求解结构相对应。

① 状态空间法将一个问题形式地定义为用四元组中的操作将某个给定的状态转换成所要求的状态。

② 状态空间法将一个待定问题的求解过程定义为若干算子与搜索的有机结合体。算子则定义了问题空间的一个操作；搜索是考察一个问题的一般技术，其目的在于找出从当前状态到某个目标状态的一条路径或某些目标状态的一组路径。

为提供对问题的形式化描述，需要定义和规定如下内容：

① 定义一个状态空间，包含一个问题相关对象各种可能的排列。当然，只需定义此空间而不必枚举该空间中的所有状态（事实上，对用人工智能方法求解的绝大部分问题来说，枚举其状态空间是不可能或不现实的）。

② 规定一个或多个属于该空间的开始状态，用以描述问题求解过程开始时的情况。

③ 规定一个或多个属于该空间的目标状态，用以描述问题的解。

④ 规定一组规则，用以描述可使用的操作或算子。

将非形式化的问题描述转换成状态空间形式化描述后，便可利用状态空间的搜索方法。应用这一组规则和相应的控制策略，在问题状态空间内进行搜索，直到找出从一开始状态到一目标状态的某条路径或到某些目标状态的一组路径。

下面举例说明如何将一个实际问题的求解转换成状态空间搜索。

【例 3.3】 给定两个水壶，容量分别是 3 升和 4 升，但没有刻度。有一个水龙头用来往水壶中装水，怎样才能在 4 升的水壶中得到 2 升水？

此问题的状态空间可以描述为一个整数序对(X, Y)，其中 X=0,1,2,3,4，Y=0,1,2,3；X 表示在 4 升的水壶中装 X 升水，Y 表示在 3 升的水壶中装 Y 升水。此问题的状态空间大小为 5×4=20。

明显，问题的初始状态是$(0, 0)$，而目标状态是$(2, n)$，其中 n 为任意值。

用于搜索的规则可总结为如下 10 条：

① $(X,Y \mid X<4)\rightarrow(4,Y)$ ；将 4 升水壶灌满

② $(X,Y \mid Y<3)\rightarrow(X,3)$ ；将 3 升水壶灌满

③ $(X,Y \mid X>0)\rightarrow(X-d,Y)$ ；从 4 升水壶中倒出 $d(d\rightarrow X)$ 升

④ $(X,Y \mid Y>0)\rightarrow(X,Y-d)$ ；从 3 升水壶中倒出 $d(d\rightarrow Y)$ 升

⑤ $(X,Y \mid X>0)\rightarrow(0,Y)$ ；将 4 升水壶的水全部倒空

⑥ $(X,Y \mid Y>0)\rightarrow(X,0)$ ；将 3 升水壶的水全部倒空

⑦ $(X,Y \mid X+Y\geqslant 4\wedge Y\rightarrow 0)$ ；将 3 升水壶的水往 4 升水壶中倒
$\rightarrow(4,Y-(4-X))$ ；直到 4 升水壶灌满为止

⑧ $(X,Y \mid X+Y\geqslant 3\wedge X\rightarrow 0)$ ；将 4 升水壶的水往 3 升水壶中倒
$\rightarrow(X-(3-Y),3)$ ；直到 3 升水壶灌满为止

⑨ $(X,Y \mid X+Y\leqslant 3\wedge X\rightarrow 0)$ ；将 4 升水壶的水全部倒入 3 升水壶中
$\rightarrow(0,X+Y)$

⑩ $(X,Y \mid X+Y\leqslant 4\wedge Y\rightarrow 0)$ ；将 3 升水壶的水全部倒入 4 升水壶中
$\rightarrow(X+Y,0)$

这里的$\rightarrow$表示规则的分隔符，它的左部被称为执行该操作或算子的前提，而它的右部被

称为该操作或算子的具体动作。

为了求解水壶问题，除了上面给出的问题描述，还需要某种控制方法，即搜索策略。它选择其左部能匹配当前状态的某条规则，并按照该规则右部的操作对此状态进行相应的状态变换——倒水；然后，检查这一新状态是否为$(2,n)$目标状态，如果不是，则继续选择匹配的规则；如果是，则求得其解。

一般地，在状态空间中，通向目标的路径不止一条，因此会存在若干条有关解的状态变换的操作序列。如下面是水壶问题的一个解。

4升水壶中水的升数	3升水壶中水的升数	应用的规则号数
0	0	②
0	3	⑩
3	0	②
3	3	⑦
4	2	⑤
0	2	⑩
2	0	得到一解

上述10条规则列举了水壶问题所有可能的状态变换描述。这样的状态空间搜索法具有一般性和灵活性。但是具有一般性和灵活性的不一定具有有效性和高效性。如在水壶问题中，对规则③和④来说，尽管它们是合理的，但不一定是必须的，甚至是无益的。直观上便可看出，它们不能使搜索更接近解，去掉它们会使搜索更有效。

3.1.4 搜索的基本概念

在状态空间中，问题的求解过程就是搜索。搜索某个状态空间以求得操作算子序列的一个解的过程，也就对应于使一个隐式图的足够大的一部分变为显式地包含目的结点的过程。这种搜索是状态空间问题求解的主要基础。

搜索的基本问题如下：

① 搜索过程是否一定能找到一个解？

② 搜索过程是否能终止运行，或者是否会陷入一个死循环？

③ 当搜索过程找到解时，找到的是否是最佳解？

④ 搜索过程的时间与空间复杂性如何？

搜索过程的要点如下：

① 从初始状态或目的状态出发，并将它作为当前状态。

② 扫描操作算子集，将适用于当前状态的一些操作算子作用在其上而得到新的状态，并建立指向其父结点的指针。

③ 检查所生成的新状态是否满足结束状态，如果满足，则得解，并可沿着有关指针从结束状态反向到达开始状态，给出一解答路径；否则，将新状态作为当前状态，返回第②步再进行搜索。

搜索可按两个方向进行：① 从初始状态出发的正向搜索，也称为数据驱动；② 从目的状态出发的逆向搜索，也称为目的驱动。

正向搜索是从问题给出的条件和一个用于状态转换的操作算子集合出发的。搜索的过程为应用操作算子从给定的条件中产生新条件，再用操作算子从新条件产生更多的新条件，这个过程持续到有一条满足目的要求的路径产生为止。数据驱动就是用问题给定数据中的约束知识指导搜索，使其沿着那些已知是正确的路线前进。

逆向搜索则是先从欲达到的目的入手，看哪些操作算子能产生该目的，以及应用这些操作算子产生目的时需要哪些条件，这些条件就成为要达到的新目的，即子目的，搜索就通过反向的、连续的子目的不断地进行，一直找到问题给定的条件为止。这样就找到了一条从数据到目的的操作算子组成的链，尽管搜索方向与它正好相反。目的驱动就是利用目的来指导相关操作算子的搜索和空间中的剪枝。

可将正向与逆向搜索看成对称过程。

究竟哪个方向为好，可考察如下 3 个因素：

① 开始状态与目的状态中，哪个状态多？往往从小的状态集出发朝大的状态集搜索，这样求解更容易。

② 哪个方向的分枝因素小？所谓分枝因素，是指从一结点出发可直接到达的平均结点数。一般都是朝着分枝因素低的方向进行搜索。

③ 考虑操作算子的多少及复杂性、状态空间的形状和是否符合人们的思考方法等。

例如，从一个陌生的地方开车回家，要比从家里开车到一个陌生的地方要容易一些。为什么呢？如不考虑存在单行道，两个方向的分枝因素是相同的。但为了在周围寻找通往目的地的路径，若以家作为目标可考虑的参照物较之以陌生地点为目标可考虑的参照物要多些。若知道从某些地方有路可通往家里，就可以把所有这些地点都当作家来考虑。因为只要能到达这些地点中的任意一个，就相当于找到了通往家的路径；而要确认出一条通往这个陌生目的地的路径，就等于已到达了这个陌生地方，所以朝陌生目标行进时，可供选择的通路较之以回家为目标的要少得多。

再如数学定理的机器证明问题。开始状态往往是一组公理集，目的状态是待证定理加上此公理集，两者大小相差无几，于是需要考虑两个方向的分枝因素。从一个小的公理集出发可推导出大量定理；反之，大量的定理又可追溯到小的公理集。因此，从公理出发正向推理到定理的分枝因素要大于从定理出发逆向推理到公理的分枝因素，所以在定理的机器证明时，采用逆向搜索为好。

当然，可以结合这两种搜索，即从开始状态出发做正向搜索，同时从目的状态出发做逆向搜索，直到两条路径在中间的某处汇合为止，这称为双向搜索。

在状态空间搜索中，需要考察哪些操作算子可作用在当前状态上。经操作算子作用后，对其所生成的新状态，需要判别是否到达结束状态，这里就涉及“匹配”概念。

选择可应用操作算子对所有的状态做穷尽扫描，用每个操作算子的可使用条件比较当前状态，并将能够匹配的操作算子组成可用操作算子集（包含的操作算子数往往不止一个）。但这种方法有两个问题：

① 在一个实际问题中，操作算子的总数很大，搜索一步就需要扫描并比较一遍全部操作算子，其效率是较低的。

② 并不是总能明显地判别出某特定状态是否匹配某一操作算子，可能还会涉及变量的取值问题，而具体取值有时又要求大面积的扫描。

因此可采用索引匹配：不通过操作算子进行搜索，而用当前状态作为对操作算子的一个索引，并选择立即匹配的那些操作算子，从数学角度和特定问题考虑对变量施加合适的约束，有时可采用近似匹配，如在语音理解问题中即是。

3.2 盲目的图搜索

通常，搜索策略的主要任务是确定如何选取操作算子的方式，有两种基本方式：盲目搜索和启发式搜索。本节将主要介绍几种盲目搜索的策略。

3.2.1 搜索策略概述

所谓盲目搜索，是指在不具有对特定问题的任何有关信息的条件下，按固定的步骤（依次或随即调用操作算子）进行，能快速地运用一个操作算子。

所谓启发式搜索，则是考虑特定问题领域可应用的知识，动态地确定调用操作算子的步骤，优先选取较合适的操作算子，尽量减少不必要的搜索，以求尽快地到达结束状态，提高搜索效率。

启发式搜索一般要优于盲目搜索，但不可过于追求更多的甚至完整的启发信息，图 3-5 可说明这个问题。

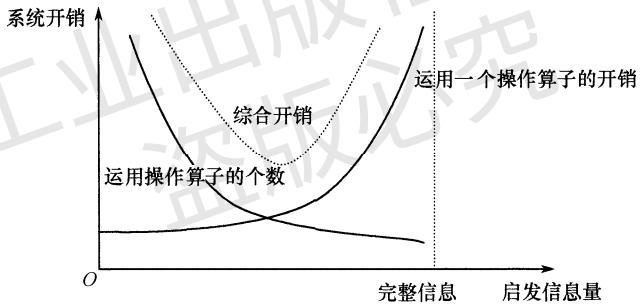

图 3-5　搜索系统的总开销

在盲目搜索中，由于没有可参考的信息，因此只要能匹配的操作算子都需要运用，这会搜索出更多的状态，生成较大的状态空间显式图；而启发式搜索中，如具有较多甚至完整的启发信息，虽然只需要运用少量的操作算子，只生成较小的状态空间显式图，能将搜索引向一个解答，但是每使用一个操作算子便需要做更多的计算与判断。这一关系类似短跑比赛中步幅与步频的关系，所以应综合考虑，尽量使搜索系统的总开销较小（最小极值点附近）。很遗憾，对某个特定问题来说，最小极值点的位置往往是模糊的，甚至是未知的。

人工智能领域中已提出许多具体的搜索策略。

① 求任一解路的搜索策略。主要包括：爬山法（Hill Climbing）、深度优先法（Depth-first）、限定范围搜索法（Beam Search）、回溯法（Back-tracking）、最好优先法（Best-first）。

② 求最佳解路的搜索策略。主要包括：大英博物馆法（British Museum）、广度优先法（Breadth-first）、分枝界限法（Branch and Bound）、动态规划法（Dynamic Programming）、最佳图搜索法（A*）、模拟退火法（Simulated-Annealing）、遗传算法（Genetic Algorithm）。

③ 求与或关系解图的搜索法。主要包括：一般的与/或图搜索法（AO*）、极大极小法（Minimax）、α-β剪枝法（Alpha-beta Pruning）、启发式剪枝法（Heuristic Pruning）。

3.2.2 回溯策略

不管是正向搜索还是逆向搜索，其求解问题都要在状态空间图中找到从初始状态到目的状态的路径。路径上弧的序列便对应解题的步骤。如问题求解在选择操作算子时，能给出绝对可靠的预测或有绝对正确的选择策略，那就不需要搜索了，求解时会一次性成功穿过状态空间而到达目的状态，构造出一条解题路径。但事实上，对一个实际问题不可能存在绝对可靠的预测，求解时必须尝试多条路径直到找到目的状态为止。回溯策略是一种系统地尝试状态空间中各种不同路径的技术。

带回溯策略的搜索是从初始状态出发，不停地、试探性地寻找路径直到它到达目的状态或“不可解结点”——“死胡同”。如果遇到不可解结点就回溯到路径中最近的父结点上，查看该结点是否还有其他子结点未被扩展，如有，则沿这些子结点继续搜索；如找到目标状态，就成功退出搜索，返回解题路径。

可以看出，这种求解过程呈现出递归过程的性质，求解过程在每个结点上的检查遵循着递归方式，下面给出递归过程。

递归过程 StepTrack(DataList):

```
Data:=First(DataList);
If Member(Data,Tail(DataList))
Then Return Fail;                                  // 回老路退回
If Goal(Data) Then Return Nil;                     // 到达目的状态，成功返回
If DeadEnd(Data) Then Return Fail;                 // 到达不合理状态，退回
If Length(DataList)＞Bound Then Return Fail;       // 已到深度限制，退回
Rules:=AppRules(Data);                             // 得出可应用的规则集
If Null(Rules) Then Return Fail;                   // 进入死胡同，退回
```

Loop:

```
R:=First(Rules);                                   // 取出第一条可运用规则
Rules:=Tail(Rules);
Newdata:=Gen(R,Data);                              // 运用规则，生成新状态
NewDataList:=Cons(Newdata,DataList);
Path:=BackTrack(NewDataList);                      // 递归
If Path:=Fail Then Go Loop Else Return Cons(R,Path);
```

说明：变量符号 Data、DataList、Rules、R、Newdata、Path 分别表示当前状态、状态序列表、规则集序列表、当前运用的规则、生成的新状态、当前解路径表；常量符号 Fail、Nil、Bound、Loop 分别表示回溯点标记、空表、深度限制值、循环标号；函数 First(x, y)表示在 y 表中取第一个元素 x；函数 Tail(x)是取除了第一个元素 x 外的表的其余部分；函数 Member(x, y)表示变量 x 是否为 y 表的一个元素；函数 Goal(x)表示变量状态 x 是否为目的状态；函数 DeadEnd(x)表示变量状态 x 是否不可能为解路上的状态；函数 Length(x)表示取 x 表的表长度；函数 AppRules(x)表示求出在变量状态 x 上可运用的规则集；函数 Null(x)表示 x 表是否为空表；函数 Gen(x, y)表示求 x 规则运用在 y 状态后所生成的新状态；函数 Cons(x, y)表示将变量状态 x 加在 y 表的前部。

从递归过程中可以看出，如当前状态 S 未达到目的状态的要求，就对它的第一个子状态 S_{child1} 递归调用回溯过程。如果在以 S_{child1} 为根的子图中未找到目的状态，就对它的兄弟子状态 S_{child2} 递归调用回溯过程。像这样重复进行直到某个子状态的后裔是目的状态或者所有子状态都搜索完为止。如 S 的子状态中没有一个能达到目的，便回溯到 S 的父状态，这样算法就可以对 S 的兄弟状态进行搜索。算法就以这种方式搜索，直到找到目的状态或遍历了状态空间为止。图 3-6 给出了一个状态空间中应用回溯搜索的示意过程，虚线箭头的方向表明了搜索的轨迹，结点旁的数字表明了被搜索到的次序。

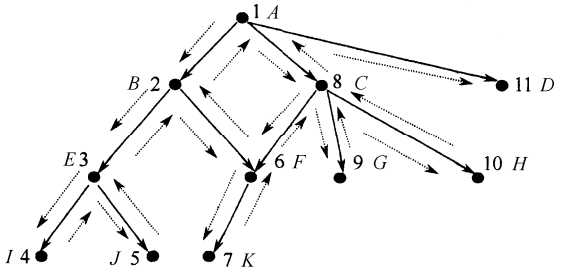

图 3-6　回溯搜索

至此可以定义一个回溯搜索算法。算法用如下 3 张表来保存状态空间中不同性质的结点。

① PS（PathStates）表：保存当前搜索路径上的状态。如果找到了目的状态，PS 就是解题路径上的状态有序集。

② NPS（NewPathStates）表：新的路径状态表，包含了等待搜索的状态，其后裔状态还未被搜索到即未被生成扩展。

③ NSS（NoSolvableStates）表：不可解状态集，列出了找不到解题路径的状态。如果在搜索中扩展出的状态是它的元素，则可立刻将它排除，不必沿该状态继续搜索。

为了使所定义的回溯算法具有普遍性（状态空间是图而不必是树），需要检测并删除多次出现的某些状态，以避免造成无穷循环搜索。具体检测过程可通过判断每个新生成的状态是否在 PS、NPS、NSS 三张表中来实现，如果它属于其中一张表，就说明它已被搜索过而不必再考虑。

当前正在被检测的状态，记为 CS（Current State），总是等于最近加入 PS 中的状态，是当前正在探寻解题路径的“前锋”。各种合适的推理规则或其他问题求解操作都可应用于 PS，应用后一般便得到一些新状态，即 PS 的子状态的有序集，再将该集合中的第一个子状态作为当前状态 CS，并加入 PS，其余的则按序放入 NPS 中，用于以后的搜索；如应用后 CS 没有子状态，就要从 PS、NPS 中删除它，同时将其加入 DE，之后回溯查找 NPS 中表首位置的状态。

该回溯算法描述如下：

```
Function backtrack;
  Begin
    PS:=[Start];    NPS:=[Start];    NSS:=[];    CS:=Strart;          // 初始化
    While NPS≠[] Do
      Begin
        If CS=目的状态 Then Return(PS);                                 // 成功，返回解路径
        If CS 没有子状态(不包括 PS、NPS 和 NSS 中已有的状态)
        Then
```

```
        Begin
          While(PS 非空) and (CS=PS 中第一个元素)) Do
            Begin
              将 CS 加入 NSS;                                  // 标明此状态不可解
              从 PS 中删除第一个元素 CS;                       // 回溯
              从 NPS 中删除第一个元素 CS;
              CS:=NPS 中第一个元素;
            End;
          将 CS 加入 PS;
        End
      Else
        Begin
          将 CS 子状态（不包括 PS、NPS 和 NSS 中已有的）加入 NPS;
          CS:=NPS 中第一个元素;
          将 CS 加入 PS;
        End
    End;
  Return FALL;                                                 // 整个空间搜索完
End
```

图 3-6 的回溯轨迹如下。

初值:	CS=*A*	PS=[*A*]	NPS=[*A*]	NSS=[]
重复	CS	PS	NPS	NSS
0	*A*	[*A*]	[*A*]	[]
1	*B*	[*BA*]	[*BCDA*]	[]
2	*E*	[*EBA*]	[*EFBDCA*]	[]
3	*I*	[*IEBA*]	[*IJEFBCDA*]	[]
4	*J*	[*JEBA*]	[*JEFBCDA*]	[*I*]
5	*F*	[*FBA*]	[*FBCDA*]	[*EJI*]
6	*K*	[*KFBA*]	[*KFBCDA*]	[*EJI*]
7	*C*	[*CA*]	[*CDA*]	[*BFKEJI*]
8	*G*	[*GCA*]	[*GHCDA*]	[*BFKEJI*]

上面搜索的过程显示出回溯是状态空间中的一个正向搜索，将初始条件作为初始状态，对其子状态进行搜索以寻找目的状态。如将目的状态作为搜索图的根即初始状态，本算法便可看作逆向搜索。如对算法中“成功，返回解路径”的判别条件“CS=目的状态”修改为“搜索路径的性质优劣”，那么算法必须通过检查 PS 中的路径来确定是否到达目的状态。

回溯算法是状态空间搜索的一个基本算法，各种图搜索算法，包括深度优先、广度优先、最好优先搜索等，都有回溯的思想：

① 用来处理状态表（NPS）使算法能返回（回溯）到其中的任一状态。

② 用一张“死胡同”状态表（NSS）来避免算法重新搜索无解的路径。

③ 在 PS 表中记录当前搜索路径的状态，当满足目的状态时可以将它作为结果返回。

④ 为避免陷入死循环必须对新生成的子状态进行检查，看它是否在这三张表中。

下面介绍一些与回溯算法类似的用表来保存搜索空间中状态轨迹的搜索算法。与回溯算法不同的是，它们实现了其他搜索策略，因而为解题提供了一个更灵活的手段。

3.2.3 宽度优先搜索

一个搜索算法的策略就是要决定树或图中状态的搜索次序。宽度、深度优先搜索是状态空间的最基本的搜索策略。

宽度优先搜索法是按如图 3-7 所示的次序来搜索状态的。也就是说，由 S_0 生成状态 1、2；然后扩展状态 1，生成状态 3、4、5；接着扩展状态 2，生成状态 6、7、8；该层扩展完后，再进入下一层，对状态 3 进行扩展，如此一层一层地扩展，直到搜索到目的状态（如果目的状态存在）。

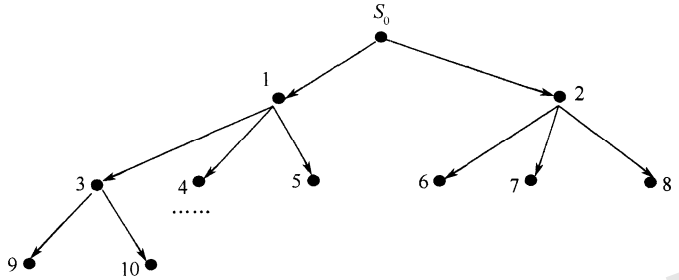

图 3-7　宽度优先搜索法中状态的搜索次序

在实现宽度优先搜索时，为了保存状态空间搜索的轨迹，用到了两个表：open 表和 closed 表。open 表与回溯算法中的 NPS 表相似，包含了已经生成出来但其子状态未被搜索的状态。open 表中状态的排列次序就是搜索的次序。closed 表记录了已被生成扩展过的状态，相当于 backtrack 算法中 PS 和 NSS 的合并。下面是宽度优先搜索过程。

```
Procedure breadth_first_search
  Begin
    Open:=[start];    closed:=[];                              // 初始化
    While open≠[] do
      Begin
        从 open 表中删除第一个状态，称为 n;
        将 n 放入 closed 表中;
        If n=目的状态     Then Return(success);
        生成 n 的所有子状态;
        从 n 的子状态中删除已在 open 或 closed 表中出现的状态;        // 避免循环搜索
        将 n 的其余子状态按生成的次序加入 open 表的后端;
      End;
  End.
```

注意：open 表是一个队列结构，即先进先出（FIFO）的数据结构；在 open 表或 closed 表中出现过的子状态要删去。

如果过程因 While 循环条件（open≠[]）不满足而结束，则表明过程已搜索完整个状态空间而仍未搜索到目的状态，也就是说，搜索失败了。

如果整个状态空间是无限的，并不能满足 While 的循环条件（即无解），则过程便会一直搜索下去，所以在过程中应增加“搜索超时而结束”的终止部分。

下面举一个宽度优先搜索的例子。

【例 3.4】 如图 3-8 所示，通过搬动积木块，希望从初始状态达到一个目的状态，即三块

积木堆叠在一起，积木 *A* 在顶部，积木 *B* 在中间，而积木 *C* 在底部。

图 3-8　积木问题

这个问题的唯一操作算子为 MOVE(*X*, *Y*)，即把积木 *X* 搬到 *Y*（积木或桌面）上面。如“搬动积木 *A* 到桌面上”表示为 MOVE(*A*, Table)。该操作算子可运用的先决条件是：

① 被搬动积木的顶部必须为空。

② 如 *Y* 是积木（不是桌面），则积木 *Y* 的顶部也必须为空。

③ 同一状态下，运用操作算子的次数不得多于一次（可利用 open 表和 closed 表加以检查）。

图 3-9 表示了由宽度优先搜索所产生的搜索树。各结点是以产生和扩展的先后次序编写下标的。当搜索到 S_{10} 目的状态时，过程便结束，此时 open 表包含 S_0～S_5，而 closed 表包含 S_6～S_{10}。

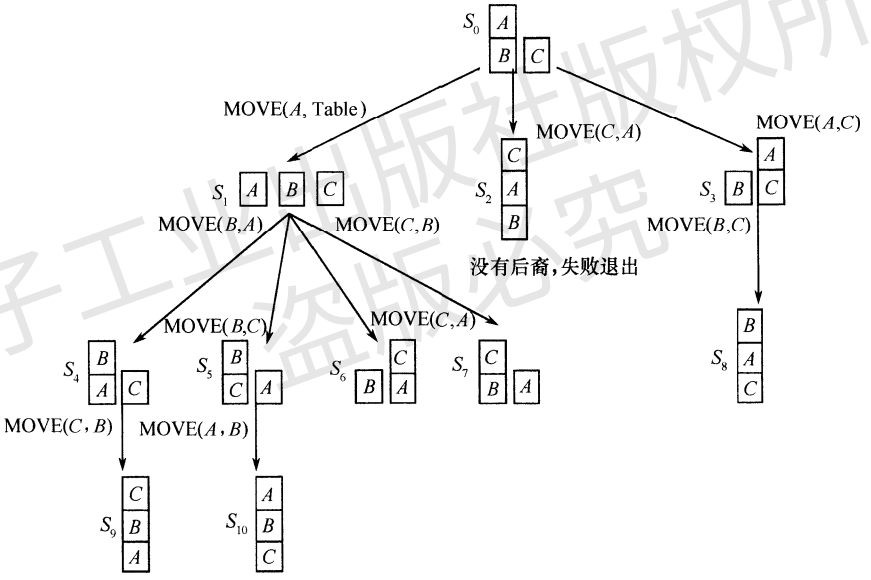

图 3-9　积木问题的宽度优先搜索树

由于宽度优先搜索总是生成扩展完 *N* 层的结点之后才转向 *N*+1 层，因此它总能找到最短的解题路径（如有解）。但实用意义不大，如图的分枝数太多，即状态后裔数的平均值较大，这种组合爆炸就会使算法耗尽资源，在可利用的空间中找不到解。这是由于每层搜索中所有生成的未扩展的结点都要保存到 open 表中，若解题路径较长，则这个数目会大得使过程无法进行搜索。

3.2.4　深度优先搜索

深度优先搜索法是按如图 3-10 所示的次序来搜索状态的。

搜索从 S_0 出发，沿一个方向一直扩展下去：状态 1、2、3，直到达到一定的深度（假定为 3 层），如未找到目的状态或无法再扩展时，便回溯到另一条路径（状态 4）继续搜索，如还未找到目的状态或无法再扩展时，再回溯到另一条路径（状态 5、6）搜索……

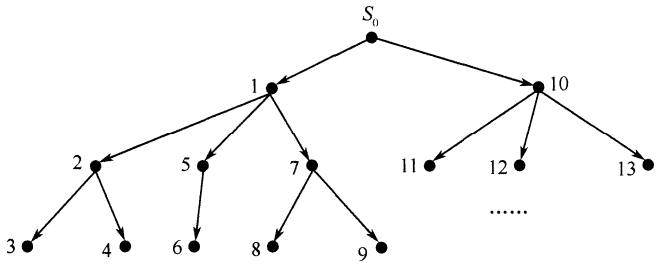

图 3-10　深度优先搜索法中状态的搜索次序

在深度优先搜索中，当搜索到某一个状态时，它所有的子状态和子状态的后裔状态都必须先于该状态的兄弟状态被搜索。深度优先搜索在搜索状态空间时应尽量往深处去，只有再也找不出某状态的后裔状态时，才考虑它的兄弟状态。

显然，深度优先搜索法不一定能找到最优解，并且可能由于深度的限制，甚至会找不到解（待求问题存在着解），然而如不加深度限制，则可能会沿着一条路径无限扩展，这当然不是我们所希望的。为了保证找到解，就应选择合适的深度限制值，或采取不断加大深度限制值的办法，反复搜索，直到找到解。

修改宽度优先搜索过程便可得到深度优先搜索过程：

```
Procedure depth_first_search
  Begin
    open:=[start];    closed:=[];    d:=深度限制值                    // 初始化
    While open≠[] do
     Begin
       从 open 表中删除第一个状态，称为 n;
       将 n 放入 closed 表;
       If n=目的状态  Then Return(success);
       If n 的深度>d  Then Continue;                                  // 回溯
       生成 n 的所有子状态;
       从 n 的子状态中删除已在 open 表或 closed 表中出现的状态;      // 避免循环搜索
       将 n 的其余子状态按生成的次序加入 open 表的前端;
     End;
  End.
```

注意：open 表是一个堆栈结构，即先进后出（FILO）的数据结构，这种将 open 表用堆栈实现的方法使得搜索偏向于最后生成的状态，曾在 open 表或 closed 表中出现过的子状态要删去。

与宽度优先搜索中一样，此处 open 表列出了所有已生成但未做扩展的状态（搜索的“前锋”），closed 表记录了已扩展过的状态。与宽度优先搜索一样，两个算法都可以把每个结点同它的父结点一起保存，以便构造一条从起始状态到目的状态的路径。

与宽度优先搜索不同的是，深度优先搜索并不能保证第一次搜索到某个状态时的路径是到这个状态的最短路径。对任何状态而言，以后的搜索有可能找到另一条通向它的路径。如

果路径的长度对解题很关键，那么当算法多次搜索到同一个状态时，它应该保留最短路径。具体地，可以把每个状态用一个三元组(状态, 父状态, 路径长度)来保存。当生成子状态时，将路径长度加 1，与子状态一起保存。当有多条路径可到达某个子状态时，这些信息可帮助选择最优的路径。必须指出，在深度优先搜索中，简单保存这些信息不能保证算法能得到最优的解题路径。

下面举一个深度优先搜索的例子。

【例 3.5】 求解卒子穿阵问题。要求卒从顶部通过如图 3-11 所示的阵列到达底部。卒在行进中时不可进入代表敌兵驻守的区域内（值为 1），也不准后退。假定深度限制值为 5。

行 \ 列	1	2	3	4
1	1	0	0	0
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0

图 3-11　阵列图

由深度优先搜索法产生的搜索树如图 3-12 所示。在结点 S_0，卒还没有进入阵列，在其他结点，其所处的阵列位置用一对数字(行号, 列号)表示，结点的编号代表搜索的次序。

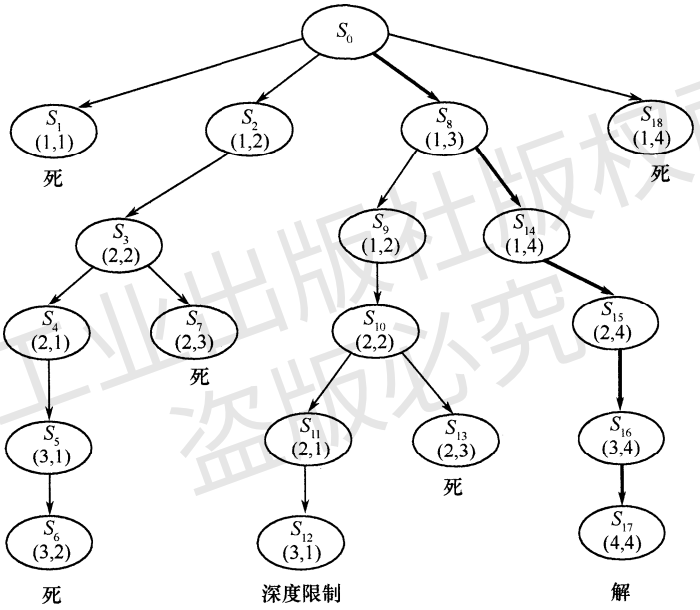

图 3-12　卒穿阵的深度优先搜索树

当搜索过程终止时，open 表含有结点 S_{17}（为一目的结点）和 S_{18}，而其他结点（S_0～S_{16}）都在 closed 表中。明显地，求得的解路径（粗线条部分）比最优路径，即从(1, 4)进入，多走 1 步。

由于该算法把状态空间作为搜索树来考虑，而不是当作一般的搜索图考虑，因此忽略了两个不同结点 S_2 和 S_9 实际上代表了同一个状态，故从结点 S_9 向下的搜索实际是在重复从结点 S_2 向下的搜索。

深度优先搜索能尽快地深入，如果已知解题路径很长，深度优先搜索就不会在开始状态的周围即“浅层”上浪费时间；另外，深度优先搜索会在搜索的深处“迷失方向”，找不到通向目的状态的更短路径或陷入一个不通往目的状态的无限长路径。深度优先搜索在搜索有大量分枝的状态空间时有相当高的效率，因为它不需要把某一层上的所有结点都进行扩展。

3.2.5 图搜索

上述策略的搜索都含有回溯的思想，都是从初始状态出发，试探性地（可退回）寻找路径直到它到达目的状态或“不可解结点”——“死胡同”。但在搜索过程中，当它遇到不可解结点时，只是回溯到路径中“最近的”父结点上查看该结点是否还有其他子结点未被扩展，因此这种回溯试探是局部的、低效率的。

假如搜索的一开始就进入无解的路径，待到发现时，已是对该部分搜索殆尽的时候了，也只得一步步地回溯，其搜索效率可想而知。

在宽度优先、代价优先等搜索方法中，用表来保存搜索状态空间中的轨迹，实现了另一些搜索策略，因而为解题提供了一种比纯粹的回溯算法更灵活的手段。虽然这些策略设置了 open 表和 closed 表来保存已扩展的结点及其扩展出的子结点，但结点之间的联系没有很好地描述出来（对纯粹的回溯算法甚至根本没有描述，除了父子结点关系），只是按照“队列”的纯数学的方法来排列 open 表，因此此类搜索策略仍有很大的搜索复杂性。

在深度优先、有界深度优先等搜索方法中，也用表示保存搜索状态空间中的轨迹，实现了其他一些搜索策略，因而为解题提供了一种比纯粹的回溯算法更灵活的手段。与宽度优先、代价优先等搜索方法类似，虽然这些策略设置了 open 表和 closed 表来保存已扩展的结点及其扩展出的子结点，但结点之间的联系没有很好地描述出来，只是按照“堆栈”的纯数学的方法来排列 open 表。因此，此类搜索策略虽能降低搜索复杂性，但仍有指数级别的搜索复杂性且不一定能找到最优解，甚至可能由于深度的限制，会找不到解（而待求问题存在着解）。

当然，除了上面介绍的方法，还有多种技术可以减少搜索的复杂性，如分枝界限法和最好优先法等。

分枝界限法的搜索策略是：每次生成一条分枝路径，并保留目前最好的回路，其开销作为界限，以检验以后所生成的候选路径；每次把一结点加到路径中时，就对这个未完成的路径加以检查，若发现扩展它的最好结果的开销都比界限值大，就不需要再沿这条路径的方向继续搜索了。这种方法可减少相当多的搜索量，但其搜索复杂性仍然有指数级。

最好优先法的搜索策略是：按照“最小的（代价）、最近的（地点）、最快的（速度）、最短的（距离）等所谓最好的”规则去搜索周围结点而构筑路径。这种方法效率非常高，因为它只需要搜索一条路径！但是这种搜索方式往往找到的不是最短路径，因为它难免有错误，所谓的最好只是局部的。然而，如果时间要求使得遍历搜索不可能实现时，此方法是一个可能的折中方案。

如果用 open 表和 closed 表能较好地描述结点之间的联系，完整地记录已搜索过的状态子图，而不是仅仅保存当前搜索的路径，就形成了所谓的图搜索策略。

在图搜索策略中，在决定搜索结点的次序时，能充分利用已搜索过的状态子图的信息，从中选择当前最佳的结点进行扩展。当然，最佳结点仍是局部的一个状态子图，但比纯粹回溯算法中的局部（父子结点对）的范围大得多，因而其搜索复杂性常会大大降低。

3.3 启发式图搜索

前面介绍的大部分搜索方法都可认为是盲目搜索方法，其搜索的复杂性是很高的。为了

提高算法的效率，必须放弃利用纯数学的方法来决定搜索结点的次序，而需要对具体问题做具体分析，利用与问题有关的信息，从中得到启发来引导搜索，以达到减少搜索量的目的，这就是启发式搜索。

本节先对启发及启发式策略涉及的问题给出简述，再将具体介绍启发式图搜索算法——A 和 A*算法，最后对启发式图搜索算法的性质进行讨论。其他启发式搜索算法，如 AO 及 AO*算法、爬山法、模拟退火法、遗传算法等将在后续几节中介绍。

3.3.1 启发式策略

启发式策略就是利用与问题有关的启发信息进行搜索。

“启发”（Heuristic）是对发现和发明操作算子及搜索方法的研究。在状态空间搜索中，启发式被定义成一系列操作算子，并能从状态空间中选择最有希望到达问题解的路径。

问题求解系统可在两种基本情况下运用启发式策略。

① 由于一个问题在问题陈述和数据获取方面固有的模糊性，可能使它没有一个确定的解，即它是一个模糊系统。如在医疗诊断中，一个病人表现出来的一系列症状会由多个原因引起，医生需要运用启发式搜索来推断出最有可能的诊断，即病因；再如视觉问题，观察者看到的景物经常是模糊的，各物体在其连接、范围和方向上可以有多个解释，光照、背景、观察者自身等造成的幻觉会加大这些模糊性，这就要求视觉系统能运用启发式策略做出给定景象的最有可能的解释。在客观世界中，绝大部分问题属于模糊系统。

② 虽然一个问题可能有确定解，但是求解过程中的搜索代价将令人难以接受。在很多问题（如国际象棋）中，其状态空间特别大，搜索中生成扩展的状态数会随着搜索的深度呈指数级增长。在这种情况下，穷尽式搜索策略，如宽度优先搜索或深度优先搜索，在一个给定的较实际的时空内很可能得不到最终的解，启发式策略则通过引导搜索向最有希望的方向进行而降低搜索复杂性。通过仔细考虑，删除某些状态及其后裔，启发式策略可以消除组合爆炸，并得到令人能接受的解。这是本节所讨论的情况。

但是启发式策略也是极易出错的。在解决问题过程中，启发仅仅是下一步将要采取措施的一个猜想，常常根据经验和直觉来判断。由于启发式搜索只利用特定问题的有限信息（假如已了解了问题的完整信息，那该问题也就不需求解了），如 open 表中状态的描述及其之间的关系，要想准确地预测下一步在状态空间中采取的具体的搜索行为是很难办到的。启发式搜索可能得到一个次最佳解，也可能一无所获，这是启发式搜索固有的局限性，而这种局限性不可能由所谓更好的启发式策略或更有效的搜索算法来彻底消除。

启发式策略及算法设计一直是人工智能的核心问题，并且启发式搜索具有实际意义。问题求解中需要启发式知识来剪枝以减少状态空间的大小，否则只能求解一些模拟的小问题。

启发式搜索通常由两部分组成：启发方法和使用该方法搜索状态空间的算法。

【例 3.6】 一字棋问题可以描述为在九宫棋盘上，从空棋盘开始，双方轮流在棋盘上摆各自的棋子×或○（每次一枚），谁先取得三子一线（一行、一列或一条对角线）的结果就取胜。

×和○能够在棋盘中摆成的各种不同棋局就是问题空间中的不同状态，在 9 个位置上摆放{空, ×, ○}有 39 种棋局。当然，其中大多数不会在实际对局中出现。任意一方的摆棋就是状态空间中的一条弧。由于第三层及更低层的某些状态可以通过不同路径到达，因此其状态

空间是图而不是树，但图中不会出现回路，因为弧具有方向性（下棋时不允许悔棋）。这样搜索路径时就不必检测是否有回路。

在一字棋问题中，第 1 步有 9 个空格，便有 9 种可能的走法，第 2 步 8 种，第 3 步 7 种……如此递减，所以有 9×8×7×…×2×1=9!种不同的棋局状态，其状态空间较大，穷尽搜索的组合数较大。

利用启发方法来剪枝可以减少状态空间的大小。根据棋盘的对称性可以减少搜索空间的大小。棋盘上很多棋局是等价的，如第 1 步实际上只有 3 种走法：角、边的中央和棋盘正中，这时状态空间的大小为 3×8!。在状态空间的第 2 层上，由对称性可进一步减少到 3+12×7!种，还可进一步减少状态空间的大小。

此外，使用启发方法进行搜索，几乎可以整个地消除复杂的搜索过程。设想将棋子走到棋盘上×最多的赢线的格子上，若两种状态有相等的赢的机率，取其中的一个，最初的 3 种状态显示在图 3-13 中。这样可设计一种算法（完全实现启发式搜索），选择具有最高启发值的棋局状态下棋。在本例中，只需要搜索×占据棋盘正中位置的棋局状态，而其他各种棋局状态连同它们的延伸棋局状态都不必再考虑了，如图 3-14 所示，2/3 的状态空间就不必再搜索了。

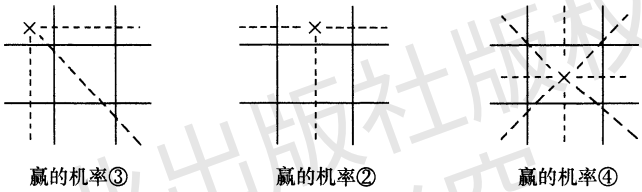

图 3-13　一字棋问题中启发式策略的运用

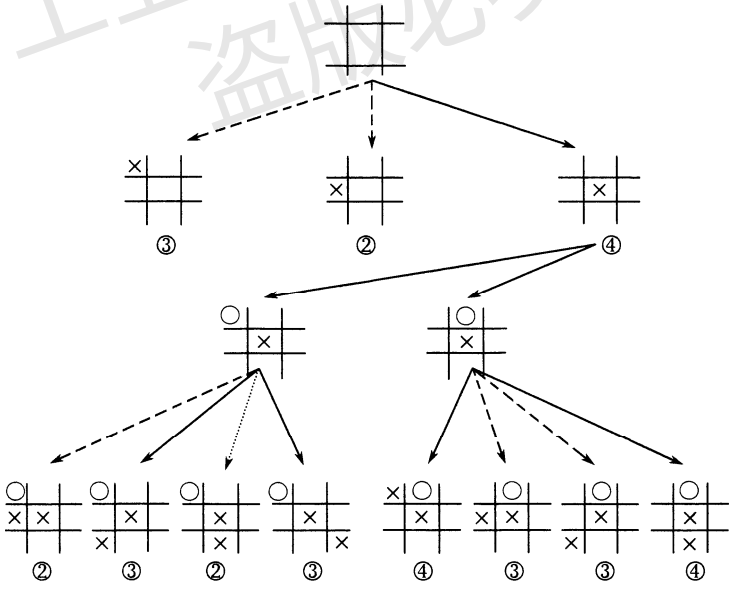

图 3-14　一字棋问题中启发式搜索下缩减的状态空间

第 1 步棋下完后，对方只能有两种走法。无论选择哪种走法，我方均可以通过启发式搜索来选择下一步可能的走法。在搜索过程中，每一步只需要估计单个结点的子结点便可决定下哪一步棋。图 3-14 显示了游戏前 3 步简化了的搜索过程。每种状态都标记了它的启发值。

要精确地计算待搜索的状态数目比较困难，但可以大致计算它的上限。一盘棋最多走 9 步，每步的下一步平均有 4～5 种走法，这样大约是 4.5×9 约 40 种状态，这比原来 9!大小的状态空间缩小了很多。

应该指出，现实中有许多复杂问题是阶乘或指数级别的，其规模很大。如国际象棋有 10^{120} 种可能的状态，西洋跳棋则有 10^{40} 种。这样的状态空间难以或者不可能遍历搜索，则必须采用启发式策略，以减少搜索的复杂程度。

3.3.2 启发信息和估价函数

在实际解决一个具体问题时，人们常常把一个具有复杂联系的实际问题抽象化，保留某些主要因素，忽略大量次要因素，从而将这个实际问题转化成具有明确结构的有限或无限的状态空间问题，并且这个状态空间中的状态和变换规律都是已知的集合，因此可以找到一个求解该问题的算法。

在具体求解时，能够利用与该问题有关的信息来简化搜索过程。此类信息被称为启发信息，利用启发信息的搜索过程被称为启发式搜索。

然而，在求解问题时能利用的大多不是具有完备性的启发信息，而是非完备的启发信息。其原因有以下两点：

① 在大多数情况下，求解问题系统不可能知道与实际问题有关的全部信息，因而无法知道该问题的全部状态空间，也不可能用一套算法来求解所有的问题，这样只能依靠部分状态空间、一些特殊的经验和有关信息来求解其中的部分问题。

② 有些问题在理论上虽然存在着求解算法，但是在工程实践中，这些算法不是效率太低，就是根本无法实现。为了提高求解问题的效率，不得不放弃使用这些“完美的”算法，而求助于一些启发信息来进行启发式搜索。

如在博弈问题中，计算机为了保证最后胜利，可以将所有可能的走法都试一遍，然后选择最佳走步。这样的算法是可以找到的，但计算所需的时空代价十分惊人。就可能有的棋局数来讲，一字棋是 $9!\approx3.6\times10^5$，西洋跳棋是 10^{78}，国际象棋是 10^{120}，围棋是 10^{761}。假设每步可以搜索一个棋局，用极限并行速度（10^{-104} 年/步）来处理，搜索一遍国际象棋的全部棋局也得 10^{16} 年（即 1 亿亿年）才可以算完，而我们已知的宇宙寿命才 100 多亿年！

由此看来，启发式的问题求解不仅在实践上是需要的，在理论上也是必不可少的。

启发信息按运用的方法可分为以下 3 种。

① 陈述性启发信息：一般被用于更准确、更精练的描述状态，使问题的状态空间缩小，如待求问题的特定状况等属于此类信息。

② 过程性启发信息：一般被用于构造操作算子，使操作算子少而精，如一些规律性知识等属于此类信息。

③ 控制性启发信息：表示控制策略的知识，包括协调整个问题求解过程中所使用的各种处理方法、搜索策略、控制结构等有关知识。

为提高搜索效率就需要利用上述 3 种启发信息作为搜索的辅助性策略。这里主要介绍控制性启发信息。

利用控制性启发信息有两种极端的情况：一种是没有任何这种控制性知识作为搜索的依

据，因而搜索的每一步完全是随意的，如随机搜索、宽度搜索、深度搜索等；另一种是有充分的控制性知识作为依据，因而搜索的每一步选择都是正确的，这是“乌托邦”式的。一般情况是介于二者之间的。

这些控制性启发信息往往反映在估价函数中。估价函数的任务是估计待搜索结点的“有希望”程序，并依此给它们排定次序（在 open 表）。估价函数 $f(x)$ 可以是任意一种函数，如有的定义为结点 x 处于最佳路径上的概率，或是 x 结点和目的结点之间的距离或差异，或是 x 格局的得分等。一般，估价一个结点的价值必须综合考虑两方面的因素：已经付出的代价和将要付出的代价。在此，把估价函数 $f(n)$ 定义为从初始结点经过 n 结点到达目的结点的路径的最小代价估计值，其一般形式为

$$f(n)=g(n)+h(n) \tag{3.1}$$

其中，$g(n)$ 是从初始结点到 n 结点的实际代价；$h(n)$ 是从 n 结点到目的结点的最佳路径的估计代价。因为实际代价 $g(n)$ 可以根据已生成的搜索树实际计算出来，而估计代价 $h(n)$ 是对未生成的搜索路径做某种经验性的估计。这种估计来源于对问题解的某些特性的认识，希望依靠这些特性来更快地找到问题的解，因此主要是 $h(n)$ 体现了搜索的启发信息。

一般地，在 $f(n)$ 中，$g(n)$ 的比重越大，越倾向于宽度优先搜索方式，$h(n)$ 的比重越大，表示启发性能越强。

$g(n)$ 的作用一般是不可忽略的，因为它代表了从初始结点经过 n 结点到达目的结点的总代价估值中实际已付出的那一部分。保持 $g(n)$ 项就保持了搜索的宽度优先成分，这有利于搜索的完备性，但会影响搜索的效率。在特殊情况下，如果只希望找到达到目的结点的路径而不关心会付出什么代价，则 $g(n)$ 的作用可以忽略。另外，当 $h(n)\gg g(n)$ 时，也可忽略 $g(n)$，这时有 $f(n)=h(n)$，这有利于搜索的效率，但影响搜索的完备性。

给定一个问题后，根据该问题的特性和解的特性，可以有多种方法定义估价函数，用不同的估价函数指导搜索，其效果可以相差很远。因此，必须尽可能地选择最能体现问题特性的、最佳的估价函数。

【例 3.7】 八数码问题的估价函数设计方法有多种，并且不同的估价函数对求解八数码问题有不同的影响。

最简单的估价函数是，与目的格局相比，某格局位置不符的将牌数目。人们直观地认为，这种估价函数很有效，因为在其他条件相同的情况下，某格局位置不符的将牌数目越少，它与最终目的格局越近，因而它是下一个搜索格局。但是这种估价函数并没有充分利用所能获得的信息，它没有考虑将牌所需移动的距离。一个较好的估价函数则是各将牌移到目的位置所需移动的距离的总和。然而，这两种估价函数都没有考虑将牌逆转（与目的格局中将牌排列的先后顺序）的情况。如果两块将牌相邻但与目标格局相比位置相反，则至少需要移动 3 次才能将它们移到正确的位置上，考虑这种情况的估价函数是对每对逆转将牌乘以一个倍数（如 3）。

第 4 种估价函数可以是综合的，弥补了仅计算将牌逆转数目策略的局限，将位置不符的将牌数目的总和与 3 倍将牌逆转数目相加。

由于上述估价函数忽略了一些重要的信息，故还能加以改进。

这个例子说明，设计一个好的估价函数具有相当的难度。设计估价函数的目标就是利用有限的信息做出一个具有较精确的估价函数。好的估价函数的设计是一个经验问题，判断和直觉是很重要的因素，但是衡量其好坏的最终标准是在具体应用时的搜索效果。

3.3.3 启发式图搜索法——A 及 A*搜索算法

启发式图搜索法的基本特点是，如何寻找并设计一个与问题有关的 $h(n)$ 及构造出 $f(n)=g(n)+h(n)$，然后以 $f(n)$ 的大小来排列待扩展状态的次序，每次选择 $f(n)$ 值最小者进行扩展。

与前面提到的宽度优先和深度优先搜索算法一样，启发式图搜索法也使用了两张表来记录状态信息：在 open 表中保留所有已生成而未扩展的状态，在 closed 表中记录已扩展的状态。算法中有一步是根据某些启发信息来排列 open 表，既不同于宽度优先所使用的“队列”，也不同于深度优先所使用的“堆栈”，而是一个按状态的启发估价函数值的大小排列的一个“表”。进入 open 表的状态不是简单地排在队尾（或队首），而是根据其估值的大小插入到表中合适的位置，每次从表中优先取出启发估价函数值最小的状态加以扩展。

一般启发式图搜索算法（简记为 A）描述如下：

```
Procedure heuristic_search
  Begin
    open:=[start];    closed:=[];    f(s):=g(s)+h(s);          // 初始化
    While open≠[]do
      Begin
        从 open 表中删除第一个状态，称为 n;
        If n=目的状态  Then Return(success);
        生成 n 的所有子状态;
        If n 没有任何子状态 Then Continue;
        For n 的每个子状态 Do
            Case 子状态 is not already on open 表 or closed 表:
              Begin
                计算该子状态的估价函数值;
                将该子状态加到 open 表中;
              End;
            Case 子状态 is already on open 表:
                If 该子状态是沿着一条比在 open 表已有的更短路径而到达
                Then 记录更短路径走向及其估价函数值;
            Case 子状态 is already on closed 表:
                If 该子状态是沿着一条比在 closed 表已有的更短路径而到达 Then
                  Begin
                    将该子状态从 closed 表移到 open 表中;
                    记录更短路径走向及其估价函数值;
                  End;
            CaseEnd;
            将 n 放入 closed 表中;
            根据估价函数值，从小到大重新排列 open 表;
      End;
    Return(failure);                                          // open 表中结点已耗尽
  End.
```

A 搜索算法在循环中每次从 open 表中取出第一个状态。如果该状态满足目的条件，那么算法返回到达该状态的搜索路径，每个状态都保留了其父状态的信息，以保证能返回完整的搜索路径。

如果 open 表的第一个状态不是目的状态，那么算法应用与之相匹配的一系列操作算子进行相应的操作来产生它的子状态。如果某个子状态已在 open（或 closed）表中出现过，即该状态再次被发现时，那么通过刷新它的祖先状态的历史记录，使算法极有可能找到到达目的状态的更短的路径。

接着，A 搜索算法估算 open 表中每个状态的估价函数值，按照值的大小重新排序，将值最小的状态放在表头，使其第一个被扩展。

图 3-15 是一个层次式状态空间，有些状态还标注了相应的估价函数值。标上值的那些状态都是在 A 搜索中实际生成的。在这个图中，A 搜索算法扩展的状态都已显示，可以看出 A 算法不需要搜索所有的状态空间。A 搜索算法的目标是尽可能地减小搜索空间而得到解。一般地，启发信息给得越多即估价函数值越大，需要搜索处理的状态数就越少。

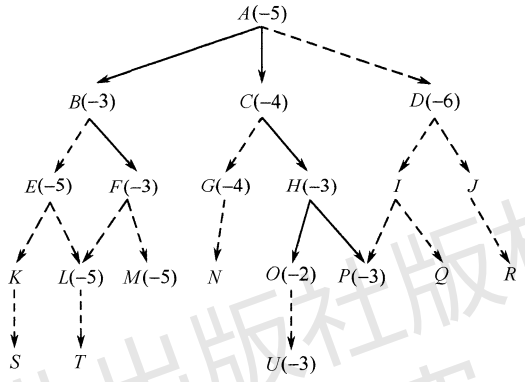

图 3-15　一个启发式图搜索的层次式状态空间

A 搜索算法总是从 open 表中选取估价函数值最小的状态进行扩展。但是在图 3-15 中，假定 P 是目标状态，而到 P 的路径上的状态有较低的估价函数值。可以看出，启发信息难免会有错误：状态 O 比 P 的估价函数值小而先被搜索扩展。然而 A 算法本身具有纠错功能，能从不理想的状态跳转到正确的状态，即从 B 跳转到 C 上来进行搜索。

因此，A 算法并不丢弃其他生成状态，而把它们保留在 open 表中。当某个启发信息将搜索导向错误路径时，算法可以从 open 表中检索出先前生成的“次最好”状态，并且将搜索方向转向状态空间的另一部分。当算法发现状态 F 的子状态有很差的估价函数值时，搜索便转移到 C，但 F 的子状态 L 和 M 都保留在 open 表中，以防算法在未来的某一步再次转向它们（见图 3-15）。

【例 3.8】 图 3-16 给出了利用 A 搜索算法求解八数码问题的搜索树，粗箭头线为解路，图中状态旁括号内的数字表示该状态的估价函数值，其估价函数定义为

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

其中，$d(n)$ 代表状态的深度，每步为单位代价；$w(n)$ 表示以“不在位”的将牌数作为启发信息的度量。

搜索过程中 open 表和 closed 表的状态排列变化情况如表 3.1 所示。

前面已提到，启发信息给得越多即估价函数值越大，则 A 搜索算法需搜索处理的状态数就越少，其效率就越高。但也不是估价函数值越大越好，因为估价函数值太大会使 A 搜索算法不一定能搜索到最优解。

表 3.1　open 表和 closed 表的状态排列变化情况

步　骤	open 表状态排列	closed 表状态排列
初始化	(*s*(4))	(　)
1 次循环后	(*B*(4)*A*(6)*C*(6))	(*s*(4))
2 次循环后	(*D*(5)*E*(5)*A*(6)*C*(6)*F*(6))	(*s*(4)*B*(4))
3 次循环后	(*E*(5)*A*(6)*C*(6)*F*(6)*G*(6)*H*(7))	(*s*(4)*B*(4)*D*(5))
4 次循环后	(*I*(5)*A*(6)*C*(6)*F*(6)*G*(6)*H*(7)*J*(7))	(*s*(4)*B*(4)*D*(5)*E*(5))
5 次循环后	(*K*(5)*A*(6)*C*(6)*F*(6)*G*(6)*H*(7)*J*(7))	(*s*(4)*B*(4)*D*(5)*E*(5)*I*(5))
6 次循环后	(*L*(5)*A*(6)*C*(6)*F*(6)*G*(6)*H*(7)*J*(7)*M*(7))	(*s*(4)*B*(4)*D*(5)*E*(5)*I*(5)*K*(5))
7 次循环后	*L* 为目的状态，则成功退出，结果搜索	(*s*(4)*B*(4)*D*(5)*E*(5)*I*(5)*K*(5)*L*(5))

定义 $h^*(n)$ 为状态 n 到目的状态的最优路径的代价。对一具体问题，只要有解，则一定存在 $h^*(n)$。于是，当要求估价函数 $f(n)$ 中的 $h(n) \leqslant h^*(n)$ 时，A 搜索算法就称为 A*搜索算法。

可以这么说，如某一问题有解，那么利用 A*搜索算法对该问题进行搜索则一定能搜索到解，并且一定能搜索到最优的解而结束。在例 3.8 中的八数码问题中，$w(n)$ 即为 $h(n)$，表示“不在位”的将牌数，$w(n)$ 满足了 $h(n) \leqslant h^*(n)$ 的条件，因此图 3-16 所示的八数码问题的 A 搜索树也是 A*搜索树，所得的解路(*s*, *B*, *E*, *I*, *K*, *L*)为最优解路，其步数为状态 L 上所标注的 5。

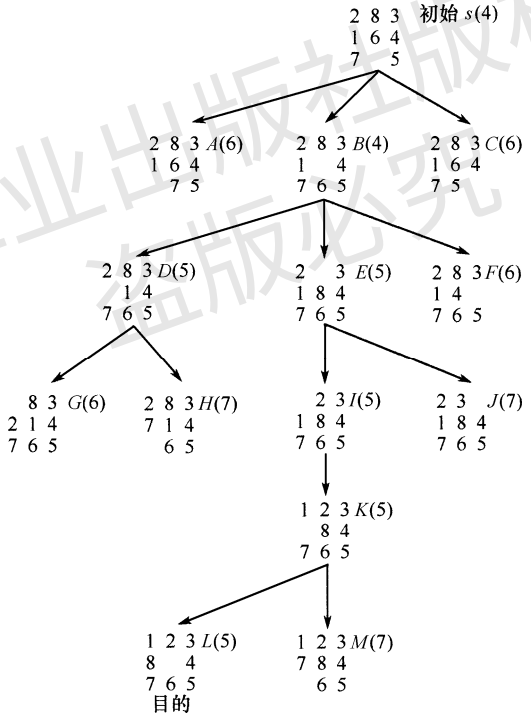

图 3-16　八数码问题的 A 搜索树

3.3.4　A*搜索算法的讨论

A*搜索算法比 A 搜索算法要好，不仅能得到目标解，还一定能找到最优解（只要问题有解）。

在一些问题求解中，只要搜索到任意解，就会想得到最优解。因此，其搜索效率显得更重要。那么，是否还有更好的启发策略，在什么意义上称一个启发策略比另一个要好？

另外，当通过启发式搜索得到某一状态的路径代价时，能否保证在以后的搜索中不会出现到达该状态有更小的代价？就上面的问题讨论以下 3 点。

1．可采纳性

当一个搜索算法在最短路径存在时能保证找到它，就称它是可采纳的。

通过估价函数 $f(n)=g(n)+h(n)$，可归纳出一类可采纳性的启发搜索策略的特征。若 n 是状态空间图中的一个状态，$g(n)$ 是某一状态在图中的深度，$h(n)$ 是 n 到目的状态代价的估计值，此时 $f(n)$ 则是从起点出发，通过 n 到达目标状态的路径的总代价的估计值。

定义最优估价函数

$$f^*(n)=g^*(n)+h^*(n) \tag{3.2}$$

其中，$g^*(n)$ 为起点到 n 状态的最短路径代价值，$h^*(n)$ 是 n 状态到目的状态的最短路径的代价值。这样，$f^*(n)$ 就是从起点出发通过 n 状态而到达目的状态的最佳路径的总代价值。

尽管在绝大部分实际问题中并不存在 $f^*(n)$ 这样的先验函数，也可以将 $f(n)$ 作为 $f^*(n)$ 的一个近似估计函数。在 A 及 A*搜索算法中，$g(n)$ 作为 $g^*(n)$ 的近似估价（可能并不相等，但 $g(n)\geqslant g^*(n)$）。仅当搜索过程已发现了到达 n 状态的最佳路径时，它们才相等）。

同样可以用 $h(n)$ 代替 $h^*(n)$，以此作为 n 状态到目的状态的最小代价估计值。虽然在绝大多数情况下无法计算 $h^*(n)$，但是要判别一个 $h(n)$ 是否大于 $h^*(n)$ 还是可能的。如果 A 搜索算法所使用的估计函数 $f(n)$ 中的 $h(n)\leqslant h^*(n)$，则称它为 A*搜索算法。

可以证明，所有的 A*搜索算法都是可采纳的。

宽度优先算法是 A*搜索算法的一个特例，它是一个可采纳的搜索算法。该算法相当于 A*搜索算法中取 $h(n)=0$ 和 $f(n)=g(n)+0$。宽度优先搜索时，对某一状态只考虑它同起始状态的距离代价。这是由于该算法在考虑 $n+1$ 层状态之前，已考察了 n 层中的任意一种状态，因此每个目的状态都是沿着最短的可能路径而找到的。遗憾的是，宽度优先搜索算法的搜索效率太低。

2．单调性

在 A*搜索算法中并不要求 $g(n)=g^*(n)$，意味着要采纳的启发算法可能先沿着一条非最佳路径搜索到某一中间状态。从 A 搜索算法可看出，在这种情况下，算法需要比较代价、调整路径等，使搜索的效率大大降低。因此，我们会问：是否有这样的启发算法，它是“局部可采纳的”，也就是该算法总能找到到达搜索中所遇到的每个状态的最短路径。幸运的是，只要函数 $g(n)$ 满足单调性便能保证这一点。那么，什么是函数的单调性呢？

如果一个启发函数 $h(n)$ 满足：

① 对所有状态 n_i 和 n_j，其中 n_j 是 n_i 的后裔，满足 $h(n_i)-h(n_j)\leqslant \text{cost}(n_i,n_j)$，其中 $\text{cost}(n_i,n_j)$ 是从 n_i 到 n_j 的实际代价。

② 目的状态的启发函数值为 0 或 $h(\text{Goal})=0$。

则称该启发函数 $h(n)$ 是单调的。

搜索算法的单调性可这样描述：在整个搜索空间都是局部可采纳的。一个状态和任意一

个子状态之间的差由该状态与其子状态之间的实际代价所限定。也就是说，启发策略无论何处都是可采纳的，总是从祖先状态沿着最佳路径到达任一状态。

于是，由于算法总是在第一次发现该结点时就已经发现了到达该点状态的最短路径，因此当某一状态被重新搜索时，就不需要检验新的路径是否更短了，这意味着，当某一状态被重新搜索到时，可以将其立即从 open 表或 closed 表中删除，而不需要修改路径的信息。

可以容易证明，单调性启发策略是可采纳的。这意味着，单调性策略中的 $h(n)$ 满足 A* 搜索策略的下界要求，算法是可采纳的。

3．信息性

何时一个启发策略要比另一个启发策略好，也就是具有更多的启发知识，使其容易找到最短路径，搜索较少的状态，特别是当两种策略都是 A*搜索算法时。

在两个 A*启发策略的 h_1 和 h_2 中，如对搜索空间中的任一状态 n 都有 $h_1(n) \leqslant h_2(n)$，就称策略 h_2 比 h_1 具有更多的信息性。如某一搜索策略的 $h(n)$ 越大，则它所搜索的状态数越少。

如果启发策略 h_2 的信息性比 h_1 要多，则用 h_2 所搜索的状态集合是 h_1 的一个子集。即算法 A*的信息性越多，它所搜索的状态数就越少。需要注意的是，更多的信息性需要更多的计算时间，从而有可能抵消减少搜索空间所带来的益处。

3.4 与/或图搜索

自然界中事物之间的关系不仅是“或”，还存在着“与”。即使某问题内的状态之间的关系似乎是“或”，但可以用问题归约等方法来求解，这就会涉及“与”关系。本节将讨论有关“与/或”关系的图描述、图搜索——AO*搜索算法和博弈树搜索。

3.4.1 与/或图的概念

第 2 章简单介绍了与/或图表示法，这里再对与/或图进行较深入的描述。在客观世界中，基本的逻辑有与、或、非，前面介绍的只有 $(Q \vee R) \Rightarrow P$ 的或逻辑关系的图表示，如图 3-17 所示；而对 $(Q \wedge R) \Rightarrow P$ 的与逻辑关系则需要用图 3-18 所示的形式表示；至于非逻辑关系，如 $\neg Q \Rightarrow P$ 可直接用其对立物 $(\neg Q \equiv Q')$ 转换为 $Q' \Rightarrow P$ 来表示。

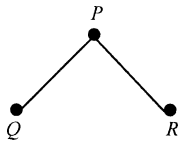

图 3-17　$(Q \vee R) \Rightarrow P$ 的或图

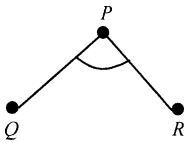

图 3-18　$(Q \vee R) \Rightarrow P$ 的与图

问题归约由问题出发，运用操作算子产生一些子问题，对子问题再运用操作算子产生子问题的一些子问题……一直进行到产生的子问题都为本原问题（不必证明的、自然成立的，如已知的事实、公理等），则问题得解。由于一个问题产生的若干子问题内的关系是并列的、同时的，因此用与/或图便能表示问题归约的状态空间。

与/或图中有“与”结点和“或”结点两种不同的结点。

若由与运算连接，它们在图中就被称为与结点，如图 3-18 中的 Q 和 R，并用一条弧将相关的边连接起来，这种弧及相关的边被称为超弧。与/或图中超弧表示 Q 和 R 都为真时，则 P 才为真。

若由或运算连接，它们在图中就被称为或结点，如图 3-17 中的 Q 和 R，或结点与父结点之间的边就不用弧线连接，表示只要 Q 或 R 任意一个为真时，P 就为真。

与/或图是一种普遍图，这种图被称为超图。也就是说，超图是存在超弧的图。与超弧相关的边数（K）被称为该超弧的度，或实现 K-连接。

前几节的普通图实际上是超图的特殊情况，此时所有超弧的度都为 1，全部实现 1-连接。也就是说，当与/或图中的所有超弧的度都为 1，即全部实现 1-连接时，该与/或图就成了普通图。

一个与/或图可能有如图 3-19 所示的结构。当引入附加结点后，图 3-19 可转换成图 3-20，其与/或关系保持不变，并且会使与/或关系变得更单纯、更清晰。

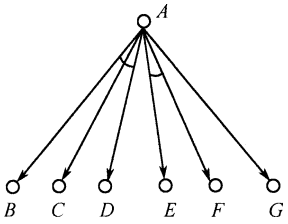

图 3-19　可能的与/或图

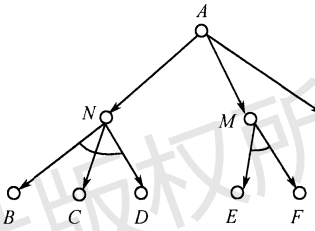

图 3-20　转换后的与/或图

在与/或图上执行搜索过程，其目的在于表明起始结点是有解的。

一个结点被称为可解结点，其递归定义为：

① 终叶结点是可解结点（直接与本原问题相关联）。

② 如果某非终叶结点含有或的子结点，那么只要子结点中有一个是可解结点，该非终叶结点便为可解结点。

③ 如果某非终叶结点含有与的子结点，那么只有当所有子结点全为可解结点时，该非终叶结点才是可解结点。

类似地，一个结点被称为不可解结点，其递归定义为：

① 没有子结点的非终叶结点是不可解结点。

② 如果某非终叶结点含有或的子结点，那么只有当所有子结点全为不可解结点时，该非终叶结点才是不可解结点。

③ 如果某非终叶结点含有与的子结点，那么只要子结点中有一个是不可解结点，该非终叶结点便为不可解结点。

于是，一个解图是那些可解结点的子图，是包含结点到目的结点集的、连通的可解结点的子图。解图 G'（从结点 n 到目的结点集合 N 的图）递归定义为：

① 如果 n 是 N 的一个元素，则 G' 由单个结点 n 组成。

② 如果 n 有一个扩展出结点 $\{n_1, n_2, \cdots, n_k\}$ 的 K-连接符，使得从每个 n_i（$i = 1, 2, \cdots, k$）到 N 有一解图，则 G' 由结点、n、K-连接符和 $\{n_1, n_2, \cdots, n_k\}$ 中的每个结点到 N 的解图所组成。

③ 否则，n 到 N 不存在解图。

如果 $n = s$ 为初始结点，则该解图即为所求解问题的解图，其值为 $h^*(s)$。

对普通图来说，在搜索中需要计算或估计其解路的代价；同样，对与/或图，也需要计算

或估计其解图的代价，但要复杂些。

设 K-连接符的代价为 C_k，如解图的代价记为 $K(n,N)$，则可递归计算：

① 如果 n 是 N 的一个元素，则 $K(n,N)=0$。

② 如果 n 有一 K-连接符扩展出结点集 $\{n_1,n_2,\cdots,n_k\}$，则

$$K(n,N)=C_n+K(n_1,N)+K(n_2,N)+\cdots+K(n_k,N)$$

具有最小代价的解图被称为最优解图，其值也用 $h^*(n)$ 表示。求解问题的解图值为 $h^*(s)$。

图 3-21 是 n_0 到 $N=\{n_7,n_8\}$ 的一个解图，其中没圈的数字表示规定的代价值，有圈的数字表示计算的代价值。该解图的代价值为 7。

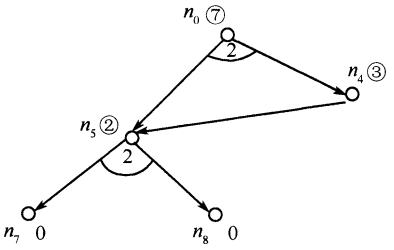

图 3-21　解图代价值的计算

注意，可能需要不止一次地计算解图中的某些代价值，如 n_5 的代价值②被计算了两次。

由于在与/或图中出现了与结点，其结构与普通图结构大为不同。与/或图需要有其特有的搜索技术，而且是否存在与结点也就成为区别两种问题求解方法的主要依据。

与/或图搜索与一般图搜索相比，要求保存更多的结点记录。对属于或结点的后裔结点的检查与一般图搜索法中一样：一旦找到一条从初始状态沿或结点通向目的状态的路径，问题就解决了，除非这条路径被证明是失败的，而必须搜索另一路径。注意，检查与结点时，要证明父结点为真，则必须证明它的所有与后裔结点都为真。

3.4.2　AO 及 AO*搜索算法

在普通图搜索中，是寻找目的结点，而在与/或图中，则是寻找一个解图。

寻找一个解图就是生成一个足以证明初始结点是可解的与/或图，也就是在问题的完整的隐含图中扩展生成包含初始结点和目的结点集合的连通的显式子图。

寻找解图时，需要检查起始结点是否为可解结点，而这种检查只有当扩展生成出可解结点时才需要进行。在做这种检查之前，首先必须对当前已生成的与/或图中的所有结点实施每结点是否为可解结点的标注过程。如果起始结点被标注为可解的，那么搜索过程可成功地结束；如果起始结点还不能被标注为可解的，那么应当继续扩展生成结点（尽可能地记录在所有生成的结点中，哪些结点被标注为可解的，以便减少下一次标注过程的工作量）。同样，当某个被选为扩展的结点不是终叶结点并且没有任何子结点时，该结点就是不可解结点。接着将不可解结点的标注过程运用于当前已生成的与/或图上，以便查看初始结点是否为不可解结点。如果起始结点被标注为不可解的，那么即失败地结束搜索过程；如果起始结点还不能被标注为不可解的，那么应当继续扩展生成结点（应尽可能地记录在所有生成的结点中，哪些结点被标注为不可解的，以便减少下一次标注过程的工作量）。

由于存在结点的可解或不可解性，因此能从搜索图中删去可解结点的任何不可解的子结点；同样，能删去不可解结点的所有的子结点，因为搜索这些被删除的结点是没有意义的，只会降低搜索的效率。

与普通图搜索算法类似，与/或图搜索算法有盲目搜索，如宽度优先搜索法、深度优先搜索法等；也有启发式搜索，如 AO 及 AO*搜索算法。这里讨论 AO 及 AO*搜索算法。

与普通图搜索算法一样，由于盲目搜索过程没有考虑有关问题域的信息，故对于人工智能应用来说，它往往不是很有效，因此需要采用启发式搜索。于是我们要问，怎样设计与/或图的启发式搜索算法呢？

下面描述这样的一个启发式搜索过程：它使用一个启发函数 $h(n)$，$h(n)$ 是 $h^*(n)$ 的一个估计，而 $h^*(n)$ 则是从结点 n 到一个终叶结点集合的一个最优解图。正如普通图搜索一样，如果 $h(n)$ 满足一定的限制，那么搜索过程是可以得到简化的。

如对 $h(n)$ 加以单调限制，即对隐含图中从结点 n 指向其后继结点 $n_1,\cdots,n_k$ 的每个连接符施加限制。假设：

$$h(n)\leqslant c(n,n_1,\cdots,n_k)+h(n_1)+\cdots+h(n_k) \tag{3.3}$$

其中，c 为连接符的代价。这一限制类似普通图中对启发函数的单调限制。对于 n 在终叶结点集合中的情况，若有 $h(n)=0$，则单调限制意味着 $h(n)$ 是 $h^*(n)$ 的一个下界，即对于所有结点 n，都有 $h(n)\leqslant h^*(n)$。

可以将 AO* 搜索过程描述如下。

```
建立仅仅包含起始结点 S 的图 G;
建立图 G'(=G);
估价函数 q(s)=h(s);
If 结点 S 为终叶结点  Then 标注结点 S 为 SOLVED;
Until 结点 S 已标注 SOLVED Do
  Begin
    （图生长）
    通过跟踪图 G 中从结点 S 出发的有标记的连接符（图 G 的连接符在下一步中标记），计算图 G 中的一个候选局部解图 G';
    Select 图 G' 的任意一个非终叶结点 n（将在以后说明如何选择）;
      扩展结点 n，生成其全部与/或后继结点，并把它们作为结点 n 的后继结点增添到图 G 中;
      对于未曾在图 G 中出现过的每个后继结点 n_j，相应的估价函数 g(n_j)=h(n_j);
      对这些后继结点中属于终叶结点者，标记 SOLVED，并赋估价函数 g(n_j)=h(n_j) 为 0;
      （修正与标注）
      建立一个仅包含结点 n 的单一结点集合 N;
    Until 集合 N 为空 Do
      Begin
        从集合 N 移出一结点 m，要求这个结点 m 在图 G 中的后裔不出现在集合 N 中;
        根据下面方法修正结点 m 的估价函数 q(m);
        （1）对于从结点 m 指向结点集 {n_1i,...,n_ki} 的第 i 个连接符，计算估价函数 q_i(m)=c_i+q(n_1i)+...+q(n_ki);
            注：对式中估价函数 q(n_ji)（其中 j=1,2,...,k）的值，或者是在这一内循环处理中的某次运算中
                刚刚计算过，或者这是第一次运算;
        （2）令估价函数 q(m) 为结点 m 的所有外向连接符所对应的估价函数 q_i(m) 中的最小值，并对这个具有最小值的
            连接符加以标记（如与以前的标记情况不同，则删掉以前的标记）;
        （3）If 该连接符的全部后继结点都已标注 SOLVED  Then 标记结点 m 为 SOLVED;
        （4）If 结点 m 已标注为 SOLVED.OR.结点 m 的估价函数发生了修正  Then 将结点 m 的所有那样的父结点（这些
              父结点的通过某个有标记连接符的后继结点之一就是结点 m）都添加到集合 N 中;
      End;
End.
```

说明：G 为当前扩展生成的与/或图；G' 为一候选局部解图，它是 G 的一个子图；$h(n)$ 是结点 n 的启发函数；而 $q(n)$ 是结点 n 的估价函数，它是从结点 n 到一组终叶结点的一个最优解图的代价估计；N 是结点的集合，用在对连接符的标记操作中。

AO*搜索算法可以理解为下面两个主要过程的反复。

首先，一个自上而下的“图生长”过程，并通过跟踪有标记的连接符寻找一个候选局部解图（这些标记表明在搜索图中离开每个结点的当前最好局部解图。在该搜索算法终止之前，最好的局部解图尚未扩展生成它的全部终叶结点，所以称它为局部的）。

AO*搜索算法中的第二个主要过程是一个自下而上的“修正和标记”过程，即估价函数值的修正、连接符的标记和 SOLVED 的标注过程。从刚被扩展的结点开始，此过程修正其估价函数值（利用其后继结点最新计算的费用），并把标记标注在估计可达到终叶结点的最好途径的外向连接符上。在搜索图中，这个估价函数修正值的估计是向上传递的（图的无环性保证了这种向上传递过程不会遇到循环）。仅仅那些经过估价函数值修正的结点，其祖先结点才有可能需要修正它们的估价函数值。由于已假设 $h(n)$ 是有单调限制的，估价函数值的修正只可能是数值的增大，因此并非所有的祖先都需要进行估价函数值的修正，只有那些在局部解图中含有修正过估价函数的后裔结点的祖先结点才需要进行估价函数值的修正（所以才有第（4）步）。

当与/或图是一棵与或树时，自下而上的过程可以稍微简单一些（因为这时每个结点只有一个父结点）。

为了避免使 AO*搜索算法进一步复杂化，被选来加以扩展的结点可能没有后继结点这一情况被忽略了。这种情况在修正结点估价函数中是容易处理的，实现的方法是把一个极高的估价函数值与没有后继结点的任一结点 m（或者，更一般地，是被认为不属于任何一个解图的任一结点）联系起来；然后，自下而上地操作将向上传递这个高估价函数值，从而排除了包含该结点的一个图被选作候选解图的任何机会。

假设结点 n 在隐含的与/或图中具有有限个后裔结点，而且它们并没有组成一个从结点 n 到一组终叶结点的解图，则结点 n 的估价函数 $q(n)$ 值最终将会修正得很高。所以，如果起始结点具有很高的估价函数值，就可能预示着这个起始结点没有解图存在。

如果从某个给定结点到一组终叶结点存在一个解图，而且对所有结点有 $h(n) \leqslant h^*(n)$，$h(n)$ 满足单调性，则可以证明 AO*搜索算法最终总能得到一个最优的解图（这个最优解图可以通过从结点 S 经过有标记的连接符跟踪到终叶结点而得到。搜索结束时这个解图的代价就等于结点 S 的估价函数值）。因此，按照 3.3 节所讨论的，具有这些限制的 AO*搜索算法是可采纳的。

当规定 $h(n) \equiv 0$ 时，则可以从 AO*搜索算法获得一个宽度优先的搜索算法，此时因为这个 $h(n)$ 函数满足了单调性（且是 $h^*(n)$ 的下界），所以利用 $h(n) \equiv 0$ 的宽度优先搜索算法也是可采纳的。

如果 $h(n) \neq 0$ 但不满足单调性，也不满足 $h(n)$ 是 $h^*(n)$ 的下界，则 AO*搜索算法便蜕变为 AO 搜索算法，与普通图搜索类似，AO 搜索算法是不可采纳的。也就是说，利用 AO 搜索算法所得的解图不一定是问题状态空间的最优解图。

下面通过一个例子进一步解释 AO*搜索算法。

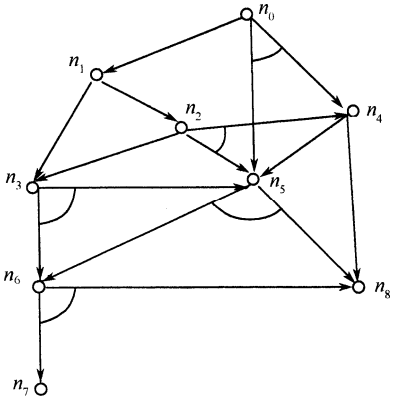

图 3-22　与/或图的搜索举例

【例 3.9】 对于如图 3-22 所示的与/或图，利用 AO 及 AO*搜索算法来求 n_0 到 $\{n_7, n_8\}$ 的解图。

假设该与/或图中各结点的启发函数 $h(n)$ 的值分别为

$$h(n_0)=0 \quad h(n_1)=2 \quad h(n_2)=4$$
$$h(n_3)=4 \quad h(n_4)=1 \quad h(n_5)=1$$
$$h(n_6)=2 \quad h(n_7)=0 \quad h(n_8)=0$$

令 *K*-连接符的代价为 *K*。注意，$h(n)$ 函数满足了 $h^*(n)$ 的一个下界，且满足单调性，因此这是一个 AO*搜索算法，其搜索过程（各次外循环）如图 3-23 所示。

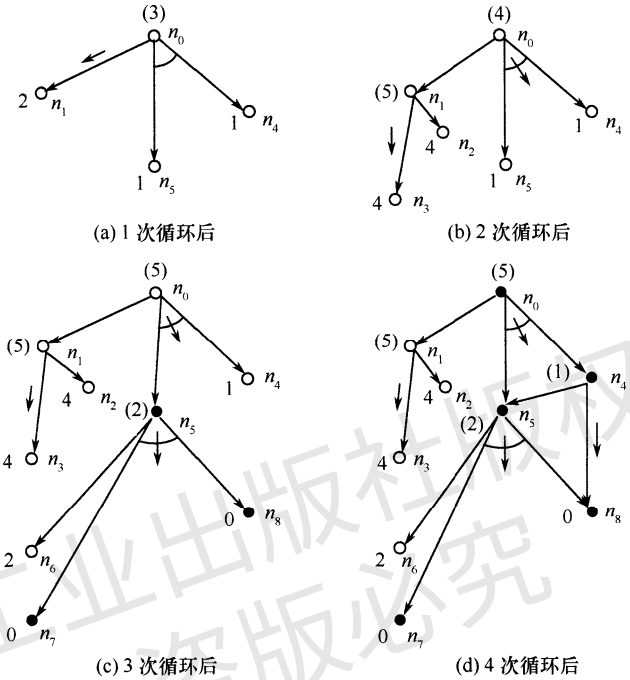

图 3-23　AO*搜索过程

在图 3-23 的每个分图中，每个结点旁边标记的是启发函数 $h(n)$ 的值（不带括号）或估价函数 $q(n)$ 的修正值（带括号）；短箭头用来标记连接符，标明候选局部解图；已标注 SOLVED 的结点用实心圆点来表示。

在第 1 次循环扩展结点 n_0，下一次循环扩展结点 n_1，接着扩展结点 n_5，最后扩展结点 n_4。在结点 n_4 被扩展后，结点 n_0 便被标注 SOLVED，此时通过向下跟踪有标记的连接符，便获得了此状态空间的解图（其代价为 5，最优解）。

AO*搜索算法是怎样选择一个估计为最好的候选局部解图的非终叶结点来加以扩展的，这个问题尚未加以讨论。也许，选择最有可能改变的局部解图最好估计的叶结点是有效的。如果候选局部解图的估计永远不变，那么无论如何 AO*搜索算法最终必须扩展这个图的全部非终叶结点。但是，如果能估计到最终会改变到某个更接近于最优解图的情况，那么 AO*搜索算法越早进行这种改变越好，也许尽早扩展具有最高估价函数值的那个叶结点将会提高搜索效率。

与 A*搜索算法一样，AO*搜索算法也可以用不同的方式加以修正，以便在特殊情况下更加实用。首先，不是在每个结点扩展后就重新计算一个新估计的候选局部解图，而是一次扩

展多个有关系的叶结点（至少对与关系的若干子结点应一起扩展）及其一定数量的后裔结点，再自下而上地计算启发函数和估价函数，从而找出一个候选局部解图。这一方式减少了频繁的自下而上运算的总开销。但有些冒险的是，某些结点的扩展可能不在最优解图上，会多扩展出一些无意义的结点。还有分段搜索方式也可应用于与/或图搜索，就是每经过一段时期的搜索（若干次外循环），对扩展生成的明确与/或图进行修剪枝节，保留少数几个有较大希望是最优解图的候选局部解图。当然，这也有风险，被修剪掉的局部解图可能就是一个最优解图的不可缺少的部分。

3.4.3 博弈树搜索

博弈一直是启发式搜索的一个重要应用领域，20 世纪 60 年代已经出现了若干博弈系统，如美国 IBM 公司的“深蓝”系统已达到了国际特级大师的水平。

这里讨论的是双人的（两人严格地轮流走步）、完备信息的（任何一方能完全知道对方已走过的步和以后可以走的所有步）博弈问题。由于在决定自己走步时只需考虑对自己有利的一步——或；而考察对方时，则应考虑对方所有可能的走步——与。因此，能利用与/或图搜索算法来解决博弈问题。另外，由于两人严格地轮流走步，使博弈状态图呈现出严格的与和或的交替层次，因此可设计特殊的与/或图搜索算法——博弈树搜索算法，使搜索更有效。

这里，通过例子来说明两种重要的博弈树搜索算法：MAX/MIN 搜索法和 α-β 剪枝过程。

首先，考虑状态空间较小的博弈，这些博弈问题能够用盲目搜索法（全部搜索）找到赢的状态；然后，考虑那些无法或难以完整地表示的状态空间及搜索整个状态空间的博弈，在这种情况下，博弈者只能运用启发信息来引导自己朝赢的方向发展。

【例 3.10】 Grundy 博弈问题。假设桌上放有一堆（7 个）钱币，两个人轮流把某堆钱币分成不相等的两堆，最后不能分下去的人为负。由于钱币数较少，整个博弈的状态空间可盲目搜索。

在博弈过程中，不能预测对手的行动，就假定对手和自己运用同样的方法力争赢得比赛。尽管这种假定有一定的局限性，但它提供了一个预测对方行动的合理的基础。本例就是在这种假设下进行状态搜索的。

博弈中的两个人分别被称为 MIN 方和 MAX 方。各自名称的含义是显而易见的：MAX 方代表努力赢得的一方，或尽力将其赢得的概率最大化的一方；而 MIN 方代表力图使 MAX 方偏离取胜目标的另一方。博弈双方 MAX 方和 MIN 方总是移向最有利于自己的状态，反之，MAX 方和 MIN 方总是移向最不利于对方的状态。

图 3-24 给出了 Grundy 博弈的全部状态空间，在搜索状态空间的每层上都标注 MIN 或 MAX，代表在此层上的行动方。在 Grundy 博弈中，MIN 方先走，每个叶结点（决定胜负的结点）都赋予值 0 或 1，用来表示是哪一方取胜（1 为 MAX 方取胜，而 0 为 MIN 方取胜），MAX/MIN 法根据下面的规则倒推出搜索图中各结点的值：

若父结点是 MAX 结点，则该结点的值为它的所有子结点估值的最大值；若父结点是 MIN 结点，则该结点的值为它的所有子结点估值的最小值。

这样，状态图中的每个状态都具有一个值（结点的倒推值带括号），表示所代表的一方达到的最好状态。图 3-24 也标出了博弈的整个 MAX/MIN 过程。

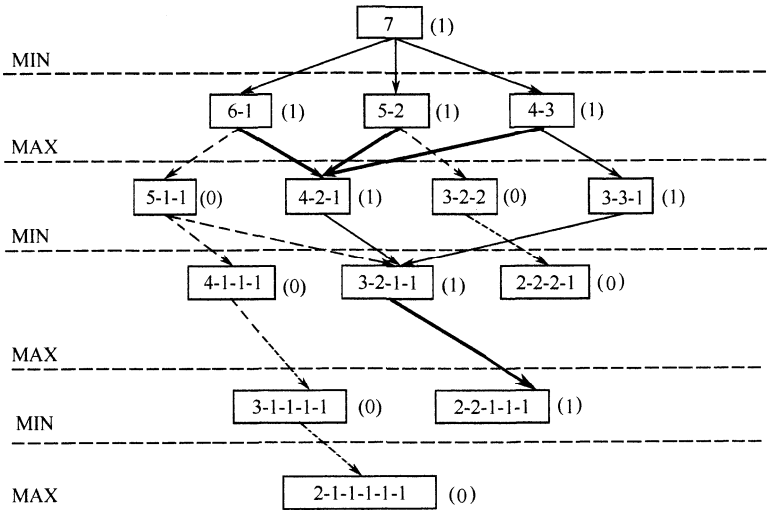

图 3-24　Grundy 博弈问题搜索图

由于 MIN 方的所有的第一步行动都指向具有倒推值为 1 的结点，故 MAX 方总能取胜；而只有当 MAX 方出错时，MIN 方才能取胜。在图 3-24 中，MIN 方不管选取哪一种行动，MAX 方都可沿着标出的粗线取胜。

尽管有许多博弈可以搜索整个状态空间，但最令人感兴趣的博弈问题往往不允许使用盲目搜索法（全部搜索），下面讨论启发式 MAX/MIN 搜索法。

在更复杂的博弈中，很难运用 MAX/MIN 搜索法来扩展整个状态空间，而是根据一定的时间和空间代价，将搜索扩展到某一层为止。由于这样扩展的明确子图的叶结点并不是博弈的最终决定胜负的状态，也就无法给出到底是输还是赢的确定值。因此，一定要根据一些能表明胜负概率的启发函数给叶结点赋值。然后，根据 MAX/MIN 法的倒推规则自下而上地倒推出搜索图中各结点的值，包括根结点。但是，这样倒推到根结点的值并不能表明谁就会赢，而只是考虑在这有限的几步（搜索图中描述的是与/或的层数）中，能达到最好的一种状态所对应的启发函数值。

在博弈中，每个人都想取胜，因此用一些启发知识直接估算一方相对于另一方的优势。在国际象棋中，残局优势是很重要的，因而一个简单的启发策略总是计算 MAX 方和 MIN 方之间的残局优势差，并尽量将这种差异极大化。更复杂的启发策略是根据残局的不同而赋予不同的启发函数值。绝大多数博弈都会有可方便利用的许多启发信息。

博弈图是一层一层地搜索，就像图 3-24 中所看到的，MAX 方和 MIN 方轮流走步。轮到任意一方的所有可能的走步构成了图中新的一层。过程一般搜索到一个固定的深度（这个深度由机器的空间/时间比所限定），从而扩展生成到该深度的所有状态的一个明确与/或图。最低一层状态的值根据启发函数计算而得，再根据 MAX/MIN 搜索法的倒推规则一直倒推赋值到根状态的子状态，接着根状态便可依据子状态的这些值，在其中选一条路径（取胜概率大的），图 3-25 是一个具有 4 层的根据启发式 MAX/MIN 搜索法得到的模拟状态空间图。

对启发式 MAX/MIN 搜索过程要注意以下两个问题：

① 对任一固定层数的限定可能产生严重的误导。如对一个层数有限的明确图，当采用某一启发策略，对最低层的状态进行计算时，该启发策略可能难以发现所选择的走步路径在

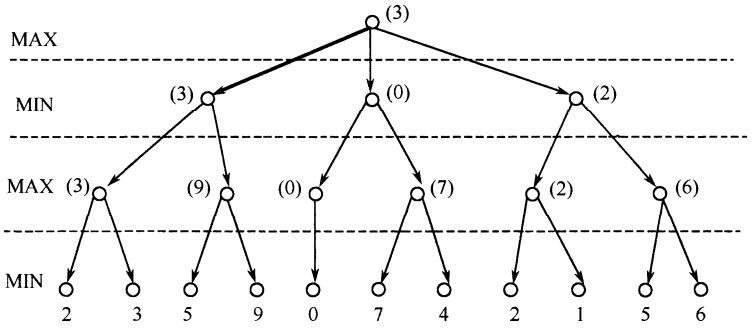

图 3-25　启发式 MAX/MIN 搜索

以后会导致不利局面的情况。例如，对手为吃你的皇后故意设个骗局作为诱惑，而搜索又恰好到设置骗局这一层为止，则过程将向这个状态走棋，从而导致全盘失利，这种情况称为“水平线效果”。当沿着看上去很好的状态继续深入搜索几层时常会碰到这种情况，仍无法避免“水平线效果”的产生，因为无论层数多深，过程总会在最深一层停止，而无法考察该层以下的状态。

② 在用 MAX/MIN 搜索法倒推赋值给各中间状态（包括根状态）时还会出现下面的情况。利用启发式 MAX/MIN 搜索法搜索状态空间时，并不是搜索的层数越深越好，因为根据启发函数计算而得的某个状态的值与该状态所处的深度是无关的，也因为不会随着深度的增加而使启发函数对状态的取胜概率的计算更准确（博弈如战争，战况瞬息万变而难以预料）。

为进一步了解启发式 MAX/MIN 搜索法及其应用，再考察一字棋游戏。

【例 3.11】 在一字棋游戏中，规定 MAX 方为×方，MIN 方为○方，MAX 方先走。

这个问题的整个状态空间有 9!个结点，即使除去同构的棋局，仍是一个很大的数字。显然盲目（全部）搜索在这里是行不通的。因此必须考虑启发式搜索方法，本例使用启发式 MAX/MIN 搜索法。尽管启发信息不够完备，但对求解这个问题还是非常有效的，一旦给定了启发函数 $h_1(n)$（在此令估价函数 $f(n)=h_1(n)$），就可以利用前面介绍的启发式 MAX/MIN 搜索法求出最优走步。

为了便于讨论，将棋盘的整行、整列或整条对角线称为赢线。如果一条赢线上只有×（○）方的棋子或为空，而没有○（×）方的棋子，那么这条赢线称为×（○）方的赢线。

这样，可以定义×方的估价函数 $h_1(n)$ 为：

① 若 n 为非终局结点，则 $h_1(n)=$ ×方的赢线数−○方的赢线数，如图 3-26 所示，×方的赢线数为 4，○方的赢线数为 2，则 $h_1(n)=4-2=2$。

② 若 n 为和局，则 $h_1(n)=0$，如图 3-27(a)所示。

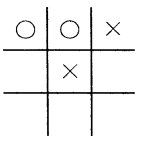

图 3-26　一字棋游戏中启发函数 $h_1(n)$ 的定义

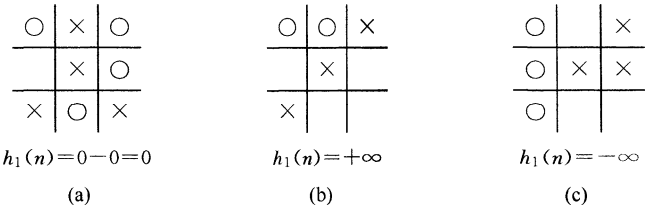

图 3-27　特殊 $h_1(n)$ 值

③ 若 n 为×方取胜的终局结点，则 $h_1(n)=+\infty$，如图 3-27(b)所示。

④ 若 n 为×方失败的终局结点，则 $h_1(n)=-\infty$，如图 3-27(c)所示。

利用上述 $h_1(n)$ 的定义，就可以计算出各种棋局的估价函数值。

显然，如果用○方的观点来分析同一个棋局的估价，得到估价函数 $h'(n)$，那么必然有 $h_1'(n)=-h_1(n)$。

利用 $h_1(n)$ 可以得到第 1 步的启发式 MAX/MIN 搜索图，如图 3-28 所示。搜索图的扩展深度为 2，它是一个似乎合理的部分博弈树，图中用粗线标出了双方的最优走步。

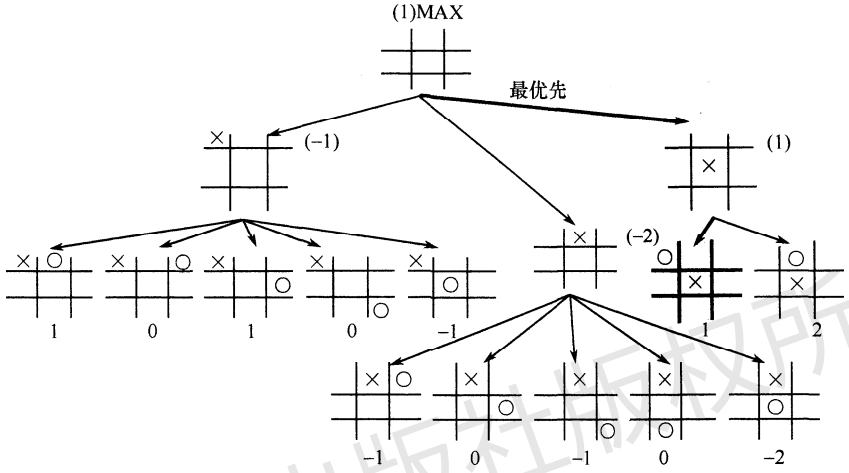

图 3-28　第 1 步的一字棋游戏搜索图

类似地，可以得到以后各步的启发式 MAX/MIN 搜索图。

然后，在图 3-29（图中以 $h_1(n)/h_2(n)$ 形式标注）所示的第 2 步 MAX/MIN 搜索图中就暴露了 $h_1(n)$ 的缺陷，它对指导中局的搜索不够准确。

从图 3-29 可以判断出×方的最优走步按 $h_1(n)$ 的估价，如图 3-30(a)和(b)所示，因为它们的估价为 1，而对×方不利的走步是图 3-30(c)和(d)，因为它们的估价为 0。

如果×方按这一估价选择图 3-30(a)的走法，则○方似乎有 4 种估价为 1 的“最优选择”（图 3-31(a)～(d)），但实际上，除了选择图 3-31(d)的走步，当○方选择其他任何一种“最优选择”，如选择图 3-31(a)、(b)或(c)，就会立即出现×方的胜局（如图 3-31(e)所示）。

可见，○方此时应该首先考虑的是占领×方的已有两个棋子的赢线，而不能只考虑增加自己的赢线。但 $h_1(n)$ 的定义中没有将一条赢线上已有几个棋子占位的情况加以区分，导致估价产生了偏差。下面介绍一种相当精确的估价函数 $h_2(n)$。

首先，引入如下概念。

① 如果一条赢线上没有任何棋子，则称为 0 阶赢线。0 阶赢线既可看作属于×方，也可看作属于○方，它们对估价没有影响。

② 如果一条赢线上仅有 1 个×（○）方的棋子，则被称为×（○）方的 1 阶赢线。

③ 如果一条赢线上已有 2 个×（○）方的棋子，则被称为×（○）方的 2 阶赢线。

于是，可以定义 $h_2(n)$ 为：

① 如果结点 n 为终局结点，那么 $h_2(n)=h_1(n)$。

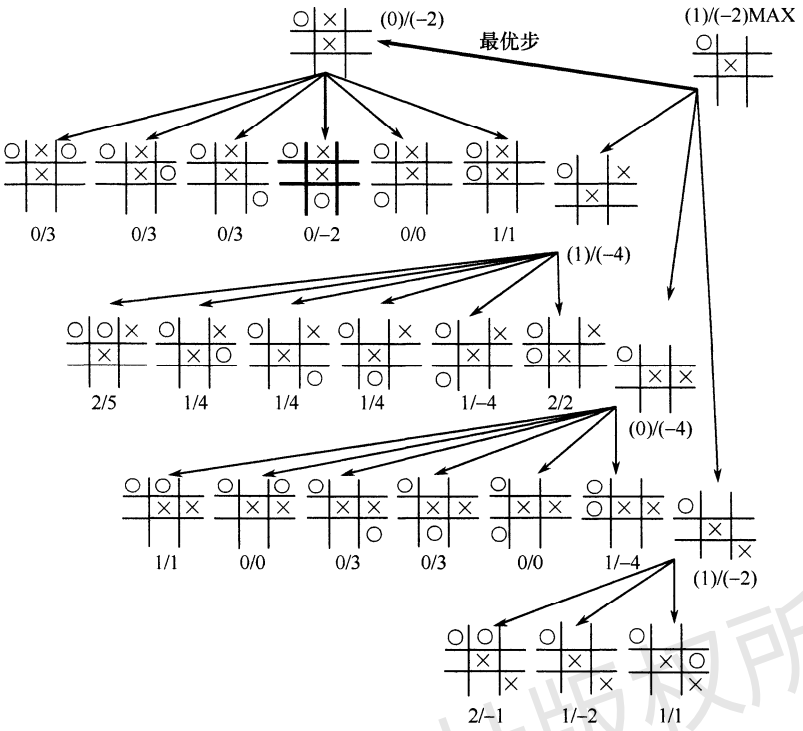

图 3-29　第 2 步的一字棋游戏搜索图

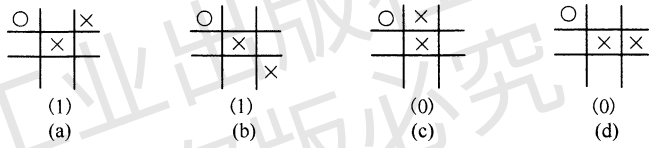

图 3-30　×方走进的可能选择

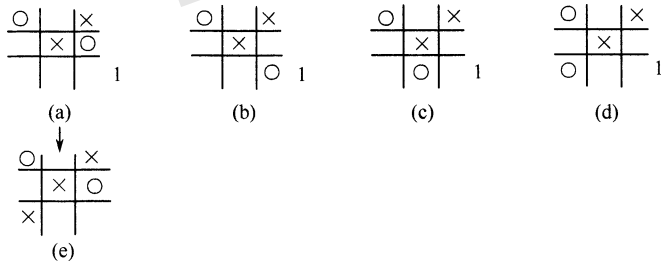

图 3-31　估价函数 $h_1(n)$ 的误导

② 如果结点 n 为非终局结点，则×方的估价函数是：

$$h_2(n)=(\times\text{方1阶赢线数}-\bigcirc\text{方1阶赢线数})+4\times(\times\text{方2阶赢线数}-\bigcirc\text{方2阶赢线数})+a$$

其中

$$a=\begin{cases}+2, & \text{若}\times\text{方出子可占领}\bigcirc\text{方的2阶赢线}\\ -2, & \text{若}\bigcirc\text{方出子可占领}\times\text{方的2阶赢线}\\ 0, & \text{其他}\end{cases}$$

利用 $h_2(n)$，可以对第 2 步 MAX/MIN 搜索图重新进行估价，其值写在 $h_1(n)$ 的斜杠下面，

得到的最优走步在图 3-29 中用粗线标注。可见，$h_2(n)$ 要比 $h_1(n)$ 精确得多。利用 $h_2(n)$ 得到第 3 步和第 4 步 MAX/MIN 搜索图分别如图 3-32 和图 3-33 所示，选中的走步都是最优的（用粗线进行了标注）。

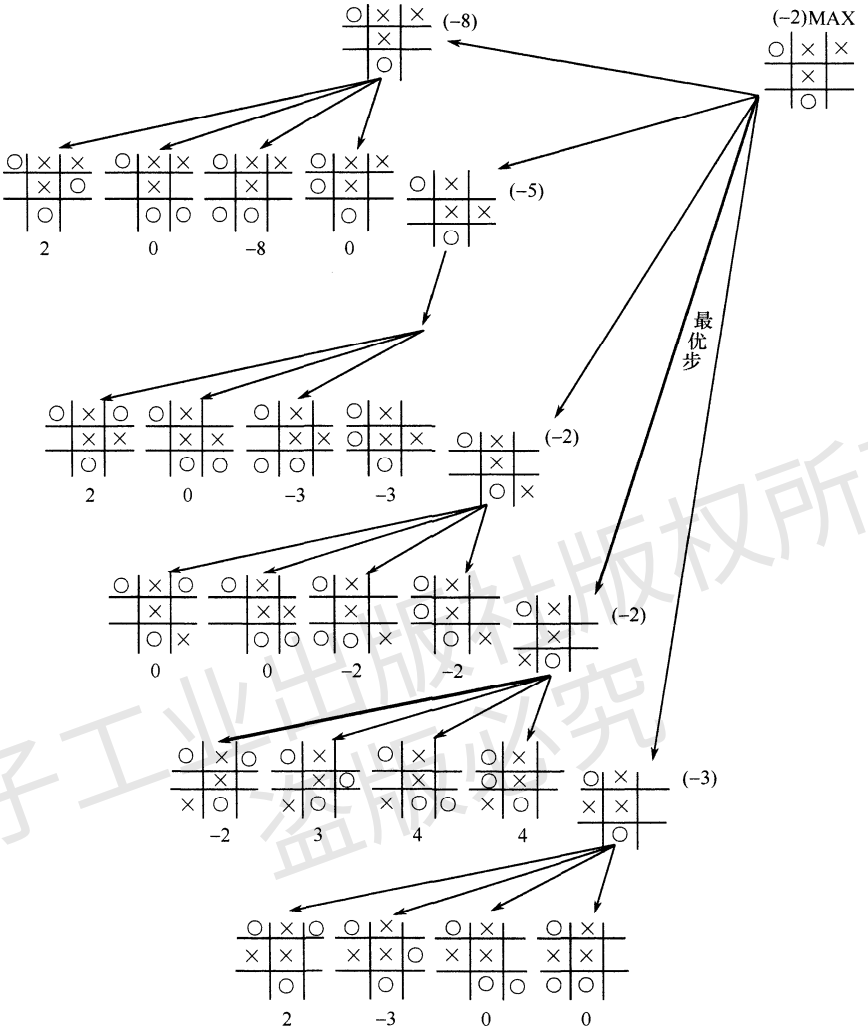

图 3-32　第 3 步的一字棋游戏搜索图

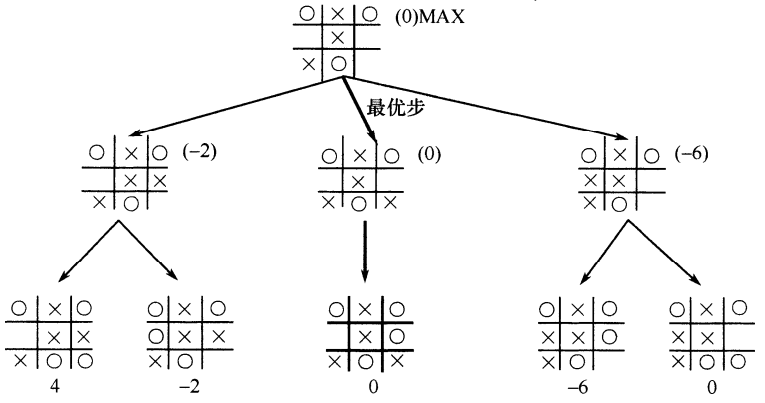

图 3-33　第 4 步的一字棋游戏搜索图

从一字棋的全部 4 步 MAX/MIN 搜索图看出，博弈双方都按 $h_2(n)$ 来指导搜索，那么结局为和局，任意一方的失误就会“引火烧身而自取灭亡”。

这种启发式 MAX/MIN 搜索法是把扩展生成博弈树的过程与计算、估价及确定最优步的过程完全分开进行，只有当全部生成规定深度的博弈树后，才开始进行计算、估价及确定最优步，这一分离导致搜索效率较低。如果在树生长的同时完成叶结点的估价函数的计算和中间结点的倒推值的倒推操作，即边生成博弈树边计算估价，那么可以减少许多生成和计算的工作量，这种技术被称为 α-β 剪枝技术。下面举一个例子加以说明。

【例 3.12】 α-β 剪枝过程，以图 3-25 中的局部博弈树为例。

如图 3-34 所示，首先根据结点 K、L 的启发函数值推出结点 E 的倒推值 BACK(E) = 3，则结点 B 的候选倒推值 PRO-BACK(B) ≤ 3。由结点 M 的启发函数值可知 PRO-BACK(F) ≥ 5，它不可能改变结点 B 的候选倒推值 PRO-BACK(B)，因此可以不搜索结点 N 即可以剪枝。至此可确定 BACK(B) = 3，从而 PRO-BACK(A) ≥ 3。由结点 O 的启发函数值可得 BACK(G) = 0，从而 PRO-BACK(C) ≤ 0。显然，它不会改变结点 A 的候选倒推值 PRO-BACK(A)，因此结点 C 的右分枝可以停止搜索——剪枝。同理，结点 J 及其以下的分枝也可以停止搜索——剪枝。最后确定了 BACK(A) = 3，并由此得出最优走步：到结点 B（粗箭头）。从图 3-34 中可统计出，原来 21 个结点的搜索任务可以减少到 14 个（实心点部分）。

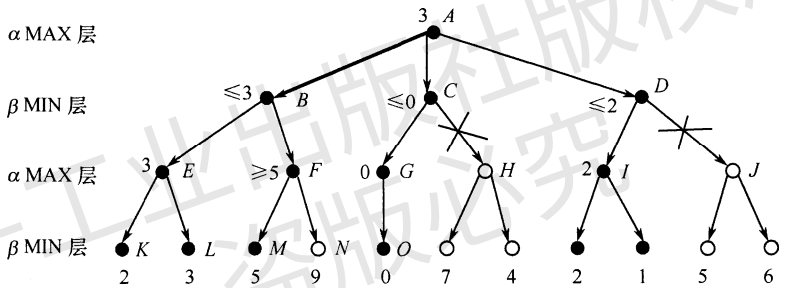

图 3-34　α-β 剪枝过程

正如图 3-34 中左边所示，把搜索过程中得到的或结点 X 的候选倒推值 PRO-BACK(X)称为 α 值，它是 BACK(X)的下界。以后无论出现怎样的搜索情况，α 不可能再下降了。相对应地，把与结点 Y 的候选倒推值 PRO-BACK(Y)叫做 β 值，它是 BACK(Y)的上界。同样地，以后无论出现怎样的搜索情况，β 不可能再上升了。

在搜索过程中，α 和 β 值按以下规律变化：① 令或结点的 α 值等于它的当前子结点中的最大倒推值；② 令与结点的 β 值等于它的当前子结点中的最小倒推值。

在搜索过程中，α-β 剪枝规则如下：

① 任何或结点 X 的 α 值如果不能降低其祖先结点的 β 值时，对结点 X 以下的分枝可以停止搜索，并令 BACK(A)= α。这种剪枝叫 β 剪枝。图 3-34 中的结点 F 下面的剪枝就是 β 剪枝。

② 任何与结点 Y 的 β 值如果不能升高其祖先结点的 α 值时，对结点 Y 以下的分枝可以停止搜索，并令 BACK(Y)= β。这种剪枝叫 α 剪枝。如图 3-34 中结点 C 和结点 D 下面的剪枝就是 α 剪枝。

在实际操作时，只要一个祖先结点和一个后裔结点之间满足 $\alpha \geq \beta$ 的条件，便可以进行剪枝（不管是 α 剪枝还是 β 剪枝）。

α-β 搜索的效率与结点的排列顺序有关。如果搜索过程中子结点产生的顺序正好满足，或结点中估值最高的子结点为最先产生，与结点中估值最低的子结点为最先产生，那么剪掉的结点数量最多，比 MAX/MIN 搜索法的效率约高 1 倍，也就是如果产生同样数量的结点，则其深度可加大 1 倍。假如子结点产生的顺序很差，那么 α-β 搜索的效率与 MAX/MIN 搜索法的一样。不过，在搜索问题过程中完全按序排列是不可能实现的（如果可能，就根本不需要搜索了），但这也较好地潜在说明了采用现有的最好排序的重要性。

前几节介绍了 A^*及 AO^*搜索算法等常规的搜索算法。深度优先、宽度优先等盲目搜索算法就不用说了，即便是 A^*搜索算法，在一般情况下，其算法复杂性仍然是指数时间级的。因此，当问题的规模大到一定程度后，这些常规的搜索算法就显得无能为力了。从 3.5 节开始将介绍一些相对比较新的搜索方法，如局部搜索算法、模拟退火算法和遗传算法等。这些算法的一个共同特点是引入随机因素，每次运行并不能保证求得问题的最优解，但经过多次运行后，一般总能得到一个与最优解相差不太大的满意解。这些算法以放弃每次必然找到最佳解的目标，来换取算法时间复杂度的降低，以适合求解大规模的优化问题。

3.5 局部搜索算法

局部搜索算法是从爬山法改进而来的。设想要爬一座自己从未爬过的高山，目标是爬到山顶，那么如何设计一种策略使得人们可以最快地达到山顶呢？在一般情况下，如果没有任何有关山顶的其他信息，沿着最陡的山坡向上爬，应该是一种不错的选择。这就是局部搜索算法最基本的思想，即在搜索过程中，始终向着离目标最近的方向搜索。当然，最优解可以是求最大值，也可以是求最小值，二者的思想是一样的。在下面的讨论中，如果没有特殊说明，均假定最优解求解的是最小值。

下面首先给出局部搜索的最一般的算法过程：

```
Procedure local_search;
  Begin
    随机 Select 一初始的可能解 x0∈D,xb=x0,P=N(xb);
      // D 是问题的定义域，xb 用于记录到目标位置得到的最优解，P 为 xb 的领域
      While 不满足结束条件（包括到达了规定的循环次数，P 为空等） Do
        Begin
          Select P 的一个子集 P′，xn 为 P′ 中的最优解;
          If f(xn)<f(xb)  Then xb=xn , P=N(xb)  Else  P=P−P′;        //  f(x) 为指标函数
        End;
    Print(xb,f(xb));
End.
```

在算法中选择 P 的一个子集 P'，可以根据问题的特点选择适当大小的子集。在极端情况下，可以选择 P' 为 P 本身，或者是 P 的一个元素。后者可以采用随机选择的方法从 P 中得到一个元素。

【例 3.13】 5 个城市旅行商问题见图 3-3，试用局部搜索方法求解该问题。

假设从城市 A 出发，用城市的序列表示该问题的一个可能解。设初始生成的可能解为

$$x_0=(A,B,C,E,D)$$

则根据各城市间的距离计算得到旅行商的旅行距离

$$f(x_b)=f(x_0)=425$$

首先选择两个城市间的位置交换方式来得到一个可以解的领域，并在 P 中随机选择一个元素（这里取第一个元素）的方法，那么算法的执行过程如下。

$$P=\{(A,C,B,E,D),(A,E,C,B,D),(A,D,C,E,B),(A,B,E,C,D),\\(A,B,D,E,C),(A,B,C,D,E)\}$$

第 1 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,C,B,E,D),\ f(x_n)=450,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,E,C,B,D),(A,D,C,E,B),(A,B,E,C,D),\\(A,B,D,E,C),(A,B,C,D,E)\}$$

第 2 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,E,C,B,D),\ f(x_n)=425,\ f(x_n)=f(x_b)$$
$$P=P-\{x_n\}=\{(A,D,C,E,B),(A,B,E,C,D),(A,B,D,E,C),(A,B,C,D,E)\}$$

第 3 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,D,C,E,B),\ f(x_n)=550,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,B,E,C,D),(A,B,D,E,C),(A,B,C,D,E)\}$$

第 4 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,E,C,D),\ f(x_n)=550,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,B,D,E,C),(A,B,C,D,E)\}$$

第 5 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,D,E,C),\ f(x_n)=475,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,B,C,D,E)\}$$

第 6 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,C,D,E),\ f(x_n)=375,\ f(x_n)<f(x_b),\ x_b=(A,B,C,D,E)$$
$$P=\{(A,C,B,D,E),(A,D,C,B,E),(A,E,C,D,B),(A,B,D,D,E),\\(A,B,E,D,C),(A,B,C,E,D)\}$$

第 7 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,C,B,D,E),\ f(x_n)=375,\ f(x_n)=f(x_b)$$
$$P=P-\{x_n\}=\{(A,D,C,B,E),(A,E,C,D,B),(A,B,D,C,E),\\(A,B,E,D,C),(A,B,C,E,D)\}$$

第 8 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,D,C,B,E),\ f(x_n)=450,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,E,C,D,B),(A,B,D,C,E),(A,B,E,D,C),(A,B,C,E,D)\}$$

第 9 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,E,C,D,B),\ f(x_n)=475,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,B,D,C,E),(A,B,E,D,C),(A,B,C,E,D)\}$$

第 10 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,D,C,E),\ f(x_n)=475,\ f(x_n)>f(x_b)$$
$$P=P-\{x_n\}=\{(A,B,E,D,C),(A,B,C,E,D)\}$$

第 11 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,E,D,C),\ f(x_n)=500,\ f(x_n)>f(x_b)$$

$$P=P-\{x_n\}=\{(A,B,C,E,D)\}$$

第 12 次循环：从 P 中随机选择一个元素，假设

$$x_n=(A,B,C,E,D),\ f(x_n)=425,\ f(x_n)>f(x_b),\ P=P-\{x_n\}=\{\ \}$$

P 等于空，算法结束，得到结果为 $x_b=(A,B,C,D,E),\ f(x_n)=375$。

在该问题中，由于初始值 (A,B,C,E,D) 的指标函数为 425，已经是一个比较不错的结果了，在 12 次循环的搜索过程中，指标函数只下降了一次，最终的结果指标函数为 375，这刚好是该问题的最优解。

从局部搜索的一般算法可以看出，该方法非常简单，也存在一些问题。在一般情况下，并不能像例 3.13 那样幸运地得到问题的最优解。

一般的局部搜索算法主要有以下 3 个问题。

1．局部最优问题

如果指标函数 f 在定义域 D 上只有一个极值点时，一般的局部搜索算法可以找到该极值点。但现实中，其指标函数 f 在定义域 D 上往往有多个局部的极值点，就如同一座群山中往往有多个小山峰一样。按照局部搜索的一般方法，一旦陷入了局部极值点，算法就将在该点处结束，这时得到的可能是一个非常糟糕的结果。解决的方法是在算法的“Select P 的一个子集 P'，x_n 为 P' 中的最优解”步中，每次并不一定选择领域内最优的点，而是依据一定的概率，从领域内选择一个点。指标函数值优的点，被选中的概率比较大，而值较差的点，被选中的概率比较小，并考虑归一化的问题，使得领域内所有点被选中的概率之和为 1。

当前点的一个邻居被选中的概率可以由领域中所有邻居的指标函数值计算得到。设 x 为当前点，其邻域为 $N(x)$，当求解的最优解为极大值时，$x_i\in N(x_i)$ 被选中的概率 $P_{\max}(x_i)$ 可定义为

$$P_{\max}(x_i)=\frac{f(x_i)}{\sum\limits_{x_j\in N(x)}f(x_j)} \tag{3.4}$$

其中，$f(x_i)$ 是 x_i 的指标函数。

这样的概率定义既符合概率和为 1 的归一化条件，又与“指标函数优的点，被选中的概率大，而指标函数差的点，被选中的概率小”的思想一致。

当求解的最优解为最小值时，也可以使用类似的思想定义概率值。如用 $1-P_{\max}(x_i)$ 作为 x_i 被选中的概率，进行归一化处理后表示为 $P_{\min}(x_i)$。$P_{\min}(x_i)$ 与 $P_{\max}(x_i)$ 的关系有

$$P_{\min}(x_i)=\frac{1}{|N(x)-1|}(1-P_{\max}(x_i)) \tag{3.5}$$

通过引入上述概率计算及随机选择机制，局部搜索算法有可能从局部的“最优解”处跳出，但由于该算法会随机地选择一些不太好的点，因此有些情况下得到的结果可能不令人满意，但总体上效果会比一般的局部搜索算法好。

2．步长问题

在距离空间中，邻域可以简单地定义为距离当前点为固定距离的点，这个固定距离称为

步长。如果步长选择不合适，即便是单极值的指标函数，一般的局部搜索算法也可能找不到一个可以接受的解。

一个可行的方法是将固定步长的搜索方法变为动态步长，开始时选择比较大的步长，随着搜索的进行，逐步减小步长。这样既解决了固定步长所带来的问题，又在一定程度上解决了小步长搜索耗时的问题。

对于组合优化问题，虽然有时距离的概念并不适用，但仍存在“步长”问题。这里的“步长”不是一般意义下的步长概念，而是指一个点到它的邻居的变化程度。以旅行商问题为例，邻居可以通过交换 2 个城市获得，也可以通过交换 3 个或者更多的城市获得。显然，交换 3 个城市比交换 2 个城市的变化大，可以认为交换 3 个城市比交换 2 个城市的“步长”长。

3．初始点位置问题

一般的局部搜索算法是否能找到全局最优解，与初始点的位置有很大的依赖关系。

一种改进的方法是随机地生成一些初始点，从每个初始点出发进行搜索，找到各自的最优解，再从这些最优解中选择一个最好的结果作为最终的结果。

解决上述 3 个问题的改进方法往往不是孤立使用的，而是综合运用。如同时考虑局部最优问题和步长问题，就形成了模拟退火算法。

3.6 模拟退火算法

模拟退火算法是局部搜索算法的一种扩展，其思想最早由 Metropolis 在 1953 年提出，Kirkpatrick 等人在 1983 年成功地将模拟退火算法用于求解组合优化问题。作为求解复杂组合优化问题的一种有效的方法，模拟退火算法已经在许多工程和科学领域得到广泛应用。

3.6.1 固体退火过程

模拟退火算法是根据复杂组合优化问题与固体的退火过程之间的相似之处，在它们之间建立联系而提出来的。

固体的退火过程是一种物理现象，属于热力学和统计物理学的研究范畴。当对一个固体进行加热时，粒子的热运动不断增加，随着温度的不断上升，粒子逐渐脱离其平衡位置，变得越来越自由，直到达到固体的熔解温度，粒子排列从原来的有序状态变为完全的无序状态。这就是固体的熔解过程。退火过程与熔解过程刚好相反。随着温度的下降，粒子的热运动逐渐减弱，粒子逐渐停留在不同的状态，其排列也从无序向有序方向发展，直到温度很低时，粒子重新以一定的结构排列。粒子不同的排列结构对应不同的能量水平。如果退火过程是缓慢进行的，也就是说，如果温度的下降非常缓慢，使得在每个温度下，粒子的排列都能达到一种平衡态，那么当温度趋于 0K（绝对温度）时，系统的能量将趋于最小值。

由此可知，系统落入能量较低状态的概率是随温度单调下降的，而系统落入高能量状态的概率是随温度单调上升的。也就是说，系统落入低能量状态的概率随着温度的下降单调上升，而系统落入高能量状态的概率随着温度的下降单调下降。

在高温条件下，系统基本处于无序的状态，基本以等概率落入各状态。在给定的温度下，

系统落入低能量状态的概率大于系统落入高能量状态的概率。这样在同一温度下，如果系统交换得足够充分，那么系统会趋向于落入低能量的状态。随着温度的缓慢下降，系统落入低能量状态的概率逐步增加，而落入高能量状态的概率逐步减小，使得系统各状态能量的期望值随温度的下降单调下降，而只有那些能量小于期望值的状态，其概率才会随温度下降而增加，其他状态均随温度下降而下降。因此，随着能量期望值的逐步下降，能量低于期望值的状态逐步减小，当温度趋于 0 时，只剩下那些具有最小能量的状态，系统处于其他状态的概率趋近子 0。因此最终系统将以概率 1 处于具有最小能量的一个状态。

固体退火过程能最终达到最小能量的一个状态，从理论上来说，必须满足 4 个条件：① 初始温度必须足够高；② 在每个温度下，状态的交换必须足够充分；③ 温度的下降必须足够缓慢；④ 最终温度必须足够低。

3.6.2 模拟退火算法

组合优化问题与固体退火过程有其类似性，将组合优化问题类比为固体的退火过程（如表 3.2 所示），便提出了求解组合优化问题的模拟退火算法。

表 3.2 组合优化问题与固体退火过程的类比

组合优化问题	固体退火过程
组合优化问题的解 $i(i \in D)$	物理系统中的一个状态 $s(s \in S)$
解的指标函数 $f(i)$	状态的能量 $E(s)$
解在邻域 N 中的交换 X	粒子的热运动 M
最优解 $f_{\min}(i)$	能量最低状态 $E_{\min}(s)$
控制参数 t	温度 T

在求解组合优化问题时，首先给定一个比较大的 t 值，相当于一个比较高的温度 T。随机给定一个问题的解 i，作为问题的初始解。在给定的 t 下，随机产生一个问题的解 j，$j \in N(i)$，其中 $N(i)$ 是 i 的邻域。从解 i 到新解 j 的转移概率，按照 Metropolis 准则确定。

$$P_t(i \Rightarrow j) = \begin{cases} 1, & f(j) < f(i) \\ \mathrm{e}^{-\frac{f(j)-f(i)}{t}}, & \text{其他} \end{cases} \tag{3.6}$$

如果新解 j 被接受，那么以解 j 代替解 i，否则继续保持解 i。重复该过程，直到在该控制参数 t 下达到平衡。与退火过程中的温度 T 缓慢下降相对应，在进行足够多的状态转移后，控制参数 t 要缓慢下降，并在每个参数 t 下，重复以上过程，直到控制参数 t 降低到足够小为止。最终得到的是该组合优化问题的一个最优解。

下面给出模拟退火算法的描述。

```
Procedure Simulated-Annealing
  Begin
    k = 0; t0 = Tmax （设定初始温度）;
    随机 Select 一个解 i;          计算指标函数 f(i);
    While 不满足结束条件（即温度不够低） Do
      Begin
        While 在该温度内未达到平衡条件（即状态交换不够充分） Do
```

```
        Begin
          从 i 的邻域 N(i) 中随机选择一个解 j;
          计算指标函数 f(j);
          If  f(j) < f(i)   Then  i = j; f(i) = f(j);
          Else
            Begin
              计算 P_t(i ⇒ j) = e^{-(f(j)-f(i))/t_k};
              If  P_t(i ⇒ j) >Random(0,1)  Then  i = j; f(i) = f(j);
            End;
        End;
      t_{k+1} = Drop(t_k); k = k + 1;
    End;
  Print (i, f(i));
End.
```

该算法有内外两层循环。内循环模拟的是在给定温度下系统达到热平衡的过程。每次循环随机地产生一个新解 j（可能是一个劣解），然后按照 Metropolis 准则，随机地接受该解 j。算法中的 Random( )函数是一个在[0, 1]间均匀分布的随机数发生器，与从解 i 到解 j 的转移概率相结合，模拟系统是否接受该解 j。外循环模拟的是温度下降过程，控制参数 t_k 起到与温度 T 相类似的作用，表示第 k 次循环时系统所处的温度。算法中的 Drop( )是一个温度下降函数，按照一定的原则实施温度的缓慢下降。

模拟退火算法与局部搜索算法很相似，二者最大的不同是模拟退火算法按照 Metropolis 准则随机地接受一些劣解，即指标函数值大的解。当温度比较高时，接受劣解的概率比较大，在初始温度下，几乎以接近 100%的概率接受劣解。随着温度的下降，接受劣解的概率逐渐减小，直到当温度趋于 0 时，接受劣解的概率也趋于 0。这样有利于算法从局部最优解中跳出，求得问题的全局的最优解。

上述模拟退火算法只是给出了一个算法框架，其中 4 个重要的条件——初始温度的选取、内循环结束条件、Drop( )和外循环结束条件—在算法中都没有提及，而这正是模拟退火算法的关键所在（正如固体退火过程的必须满足 4 个条件）。

与固体退火过程一样，为了使模拟退火算法以概率 1 求解到问题的最优解，至少要满足这 4 个条件。然而“初始温度必须足够高，状态交换必须足够充分，温度下降必须足够缓慢，最终温度必须足够低”这样的条件是与人们试图给出求解组合最优化问题的低复杂度算法的初衷相违背的。如果模拟退火算法仍然是一个指数复杂度的算法，那么对于求解复杂组合优化问题不会带来任何帮助。现在的问题是，如何弱化一些条件，使得人们能够在一个多项式时间复杂度内，求得一个组合优化问题的满意解。所幸的是，只要适当地构造状态 i 产生 j 的概率及接受状态 j 的概率和邻域 $N(i)$，就可以从理论上（基于马尔可夫链的平稳分布和全局最优问题）证明模拟退火算法一定能以概率 1 找到全局最优解。下面将给出一些确定初始温度及内、外循环结束条件的基本方法。

3.6.3 参数的确定

初始温度，在每个温度下状态的交换次数和温度的下降方法，以及温度下降到什么程度

算法结束等，成为模拟退火算法求解问题时必须考虑的问题。对于很多实际问题，求解问题最优解的意义并不大，一个满意解就足够了。能否在一个多项式时间内得到问题的满意解则是我们最关心的问题。

并不是任何一组参数都能够保证模拟退火算法收敛于某个近似解。大量的实验表明，解的质量与算法的运行时间是呈正比例关系的，很难做到两全其美。下面给出模拟退火中一些参数或准则的确定方法，并试图在求解时间与解的质量之间做一个折中的选择。这些参数或准则包括：① 初始温度参数 t_0；② 温度 t_k 的衰减函数 Drop(t_k)，即温度的下降方法；③ 每一温度 t_k 下的停止准则，即马尔可夫链的长度 L_k；④ 算法的终止准则（用终止温度 t_f 或终止条件给出）。

1．起始温度 t_0 的选取

模拟退火算法要求初始温度足够高，这样才能在初始温度下，使问题的求解以等概率处于任何一个状态。多高的温度才算“足够高”呢？这显然与具体的问题有关，就像金属材料中，不同的材料具有不同的熔解温度一样。因此初始温度应根据具体的问题而定。

一个合适的初始温度应保证平稳分布中每个状态的概率基本相等，也就是接受概率 P_0 近似等于 1。

仿照固体的升温过程，可以通过逐渐升温的方法得到一个合适的初始温度。具体方法如下：

① 给定一个希望的初始接受概率 P_0，给定一个较低的初始温度 t_0，如 t_0=1。

② 随机产生一个状态序列，并按式(3.7)计算该序列的接受概率：

$$\text{接受概率}=\frac{\text{接受的状态数}}{\text{产生的状态总数}} \tag{3.7}$$

如果接受概率小于给定的初始接受概率 P_0，那么提高温度，更新 t_0；然后，重新计算接受概率并比较，直到接受概率大于 P_0。

其中，更新 t_0 可以采用每次加倍的方法，即 t_0=2×t_0；也可以采用每次加固定值的方法，即 t_0=t_0+T（T 为一个事先给定的常量）。

2．温度的下降方法

退火过程要求下降温度足够缓慢，常用的温度下降方法有以下 3 种。

（1）等比例下降

该方法通过设置一个衰减系数，使得温度每次以相同的比率下降。

$$t_{k+1}=\alpha t_k \quad (k=0,1,\cdots) \tag{3.8}$$

其中，t_k 是当前温度，t_{k+1} 是下一个时刻的温度，α 是一个常数，$0<\alpha<1$，α 越接近于 1，温度下降得越慢，一般可以选取 0.8～0.95。该方法简单实用，是一种常用的温度下降方法。

（2）等值下降

该方法使得温度每次下降的幅度是一个固定值。

$$t_{k+1}=t_k-\Delta t \tag{3.9}$$

设 K 是希望的温度下降总次数，则 $\Delta t=\frac{t_0}{K}$，其中 t_0 是初始温度。该方法的好处是可以控制总的温度下降次数，但由于每次温度下降一个固定值，如果温度设置得过小，那么在高温时温度下降太慢；如果设置得过大，那么在低温下温度下降得又过快。

（3）基于距离参数的下降方法

如果温度每次下降的幅度比较小，那么相邻温度下的平稳分布应该变化不大，也就是说，对于任意一个状态 $i \in S$，相邻温度下的平稳分布应满足：

$$\frac{1}{1+\delta} < \frac{P_{t_k}(i)}{P_{t_{k+1}}(i)} < 1+\delta \quad (k=0,1,\cdots) \tag{3.10}$$

其中，δ 是一个较小的正数，被称为距离参数。于是，温度的衰减函数为：

$$t_{k+1} = \frac{t_k}{1+\dfrac{t_k \ln(1+\delta)}{3\sigma_{t_k}}} \quad (k=0,1,\cdots) \tag{3.11}$$

其中，σ_{t_k} 为温度 t_k 下所产生的状态的指标函数值的标准差 $(f(i)-f_m)$。实际计算时，可通过样本值近似计算得到。

方法（1）和方法（2）独立于具体的问题，而方法（3）是与具体问题有关的温度下降方法。

3．每个温度下的停止准则

在每个温度下，模拟退火算法都要求产生足够的状态交换，如果用 L_k 表示在温度 t_k 下的迭代次数，那么 L_k 应使得在这一温度下的马尔可夫链基本达到平稳状态。

一般有以下几种常用的停止准则：

（1）固定长度方法

这是最简单的一种方法，在每个温度下，都使用相同的 L_k。L_k 的选取与具体的问题相关，一般与邻域的大小直接关联，通常选择为问题规模 n 的一个多项式函数。如在 n 城市的旅行商问题中，如果采用交换两个城市的方法产生邻域，邻域的大小为 $n(n-1)/2$，那么 L_k 可以选取为 C_n，其中 C 为常数。

（2）基于接受概率的停止准则

由前面对退火过程的分析知道，在比较高的温度时，系统处于每个状态的概率基本相同，而且每个状态的接受概率也接近于 1。因此在高温时，即使比较小的迭代数，也可以基本达到平稳状态。而随着温度的下降，被拒绝的状态数随之增加，因此在低温下迭代数应增加，以免由于迭代数太少，而过早地陷入局部最优状态。因此，一个直观的想法是随着温度的下降适当地增加迭代次数。

一种方法是，规定一个接受次数 R，在某一温度下只有被接受的状态数达到 R 时，在该温度下的迭代才停止，并转入下一个温度。由于随着温度的下降，状态被接受的概率随之下降，因此这一准则满足随着温度的下降而适当地增加迭代次数的要求。但由于温度比较低时接受概率很低，为了防止出现过多的迭代次数，一般设置一个迭代次数的上限。当迭代次数达到上限时，即使不满足接受次数 R，也停止这一温度的迭代过程。

与上一种方法相似，可以规定一个状态接受概率 R。如果实际接受概率达到了 R，那么停止该温度下的迭代，转入下一个温度。为了防止迭代次数过少或者过多，一般定义一个迭代次数的下限和上限，只有当迭代次数达到了下限并且满足所要求的接受概率 R 时，或者达到了迭代次数的上限时，才停止这一温度的迭代。

还可以通过引入“代”的概念来定义停止准则。在迭代过程中，若干相邻的状态称为“一代”，如果相邻两代的解的指标函数差值小于规定的值，则停止该温度下的迭代。

在某一温度下的迭代次数与温度的下降紧密相关。如果温度下降的幅度比较小，那么相邻两个温度之间的平稳分布相差也应该比较小。一些研究表明，过大的迭代次数对提高解的质量关系不大，只会增加系统的运算时间。因此一般选取比较小的温度衰减值，只要迭代次数适当大就可以了。

4．算法的终止准则

模拟退火算法从初始温度 t_0 开始逐步下降温度，最终当温度下降到一定值时，算法结束。合理的结束条件应使算法收敛于问题的某一个近似解，同时保证该解具有一定的质量，且应在一个可以接受的有限时间内停止求解。一般有下面 6 种确定算法终止的方法。

（1）零度法

从理论上讲，当温度接近 0 时，模拟退火算法才结束。因此可以设定一个正常数 ε，当 $t_k < \varepsilon$ 时，算法才结束。

（2）循环总控制法

给定一个温度下降次数 K，当温度的迭代次数达到 K 次时，则算法停止。这要求有一个合适的 K，如果 K 的值选择的不合适，对于小规模的问题将导致增加算法无谓的运行时间，而对于大规模的问题，那么可能难以得到高质量的解。

（3）无变化控制法

随着温度的下降，虽然由于模拟退火算法会随机地接受一些不好的解，但从总体上说，得到的解的质量应该逐步提高，在温度比较低时更是如此。如果在相邻的 n 个温度中，得到的解的指标函数值无任何变化，那么说明算法已经收敛，即收敛于局部最优解。又由于在低温下跳出局部最优解的可能性很小，因此算法可以终止。

（4）接受概率控制法

给定一个小的概率值 p，如果在当前温度下除了局部最优状态，其他状态的接受概率均小于 p 值，那么算法可结束。

（5）邻域平均概率控制法

设一个邻域的大小为 N，在邻域内一个状态被接受的平均概率为 1/N。如果接受概率值 $\mathrm{e}^{-\frac{f_1-f_0}{t}}$（其中 f_0 和 f_1 为该邻域中的局部最优值和局部次最优值）小于平均值 1/N 时，则可以认为从局部最优解跳出的可能性已经很小了，因此可以终止算法。

（6）相对误差估计法

设温度 t 的指标函数的期望值为 $< f(t_f) >= \sum_{i \in S} f(i) P_t(i)$，当终止温度 $t_f \ll 1$ 时，由泰勒级数展开近似有

$$< f(t_f) > - f_m \approx t_f \left. \frac{\mathrm{d} < f(t) >}{\mathrm{d}t} \right|_{t=t_f} \tag{3.12}$$

其中，f_m 是最优解的指标函数值。所以，当 $t_f \ll 1$ 时，可以用式(3.12)估算当前解与最优解之间的误差。

为了消除指标函数值大小的影响，可以用 $< f(\infty) >$ 的相对误差来表示。当相对误差满足如下条件时，算法便可终止。

$$\frac{t_f}{<f(\infty)>}\left.\frac{\mathrm{d}<f(t)>}{\mathrm{d}t}\right|_{t=t_f}=\frac{\sigma_{t_f}^2}{t_f<f(\infty)>}<\varepsilon \tag{3.13}$$

其中，ε是一个给定的小正数，称为停止参数；σ_{t_f}是在温度t_f下的指标函数的标准差。

实际计算时，$<f(\infty)>$用$<f(t_0)>$代替，指标函数的期望值、标准差等均通过样本值近似计算得到，于是算法的终止准则可以表达为

$$\frac{\overline{f^2(t_f)}-\overline{f(t_f)}^2}{t_f\overline{f(t_0)}}<\varepsilon \tag{3.14}$$

其中，$\overline{f(t_0)}=\frac{1}{n}\sum_{i=1}^{n}f(X_i)$，$\overline{f^2(t)}=\frac{1}{n}\sum_{i=1}^{n}f^2(X_i)$，$X_i(i=1,2,\cdots,n)$是温度为$t$时状态的$n$个样本。

以上给出了用模拟退火算法求解组合优化问题时确定算法参数的一些方法，这些方法基本上是基于经验的，并没有太多的理论指导，需要根据具体的问题具体分析。

3.6.4 应用举例

下面通过旅行商问题具体说明如何应用模拟退火方法来求解组合优化问题。

设有n个城市，城市间的距离用矩阵$\boldsymbol{D}=[d_{ij}](i,j=1,2,\cdots,n)$表示，其中$d_{ij}$表示城市$i$与城市$j$之间的距离，当问题对称时，有$d_{ij}$=$d_{ji}$。

（1）解空间

n个城市的任何一种排列$\pi_1,\pi_2,\cdots,\pi_n$[$\pi_i=j(j=1,2,\cdots,n)$表示第i个到达的城市是j，且$\pi_{n+1}=\pi_1$]均是问题的一个可能的解，其空间规模为$(n-1)!$。

（2）指标函数

由于问题本身要求解最短长度的旅行距离，于是将访问所有城市的路径长度定义为问题的指标函数，即

$$f(\pi_1,\pi_2,\cdots\pi_n)=\sum_{i=1}^{n}d_{\pi_i\pi_{i+1}}(\text{注意}\pi_{n+1}=\pi_1)$$

（3）新解的产生

采用两个城市间的逆序交换方式得到问题的一个新解。设当前解是$(\pi_1,\pi_2,\cdots,\pi_n)$，被选中要逆序交换的城市是第$u$和第$v$个访问的城市（其中$u<v$，$1\leqslant u,v<n+1$），则得到的新解是

$$(\pi_1,\pi_2,\cdots,\pi_{u-1},\pi_u,\pi_{v-1},\pi_{v-2},\cdots,\pi_{u+1},\pi_v,\pi_{v+1},\cdots,\pi_n) \tag{3.15}$$

（4）指标函数差

考虑城市间距离的对称性，式(3.15)所示的新解与原解之间的指标函数差只涉及两对城市间的变化，因此指标函数差为

$$\Delta f=(d_{\pi_u\pi_{v-1}}+d_{\pi_{u+1}\pi_v})-(d_{\pi_u\pi_{u+1}}+d_{\pi_{v-1}\pi_v}) \tag{3.16}$$

（5）新解的接受准则

按式(3.17)计算接受条件：

$$A_t=\begin{cases}1, & \Delta f<0\\ \mathrm{e}^{-\frac{\Delta f}{t}}, & \text{其他}\end{cases} \tag{3.17}$$

（6）参数确定

康立山等人在其著作中对模拟退火算法求解旅行商问题进行了大量的试验分析，确定了一组比较合适的参数，该组参数权衡了运行时间和解的质量等因素。下面给出这组参数，以使大家有一个感性的认识。

初始温度 $t_0=280$，在每个温度下采用固定的迭代次数，$L_k=100n$，n 为城市数；温度的衰减系数 $\alpha=0.92$，即 $t_{k+1}=0.92\times t_k$；算法的终止准则为当相邻两个温度得到的解无任何变化时算法停止。

Nirwan Ansari 和 Edwin Hou 二人在他们的著作中采用了另一组参数。初始温度 t_0 是这样确定的：从 $t_0=1$ 出发，并以 $t_0=1.05\times t_0$ 对 t_0 进行更新，直到接受概率大于等于 0.9 时为止，此时得到的温度为初始温度；在每个温度下采用固定的迭代次数，$L_k=10n$，n 为城市数；温度的衰减系数 $\alpha=0.95$，即 $t_{k+1}=0.95\times t_k$；算法的终止准则为温度低于 0.01，或者没有任何新解生成。

模拟退火算法求解旅行商问题还是非常有效的。在大多数情况下都能找到最优解并结束，即使不是最优解，也是一个可以接受的满意解。即使是最差的结果，与最优解的差距也不是很大。

3.7 遗传算法

遗传算法是根据自然界的“物竞天择，适者生存”现象而提出的一种随机搜索算法，20 世纪 70 年代由美国密执根大学的 Holland 教授首先提出。遗传算法将优化问题看作自然界中生物的进化过程，通过模拟大自然中生物进化过程的遗传规律来达到寻优的目的。近年来，遗传算法作为一种有效的工具，已广泛地应用于最优化问题求解之中。

3.7.1 生物进化和遗传算法

虽然人们对生物进化的一些细节还不是很清楚，但一些进化理论的特性已经普遍被研究者所接受。生物进化的特性总结如下：

① 进化过程发生在染色体上，而不是发生在它们所编码的生物体上。

② 自然选择把染色体及其所编码的结构的表现联系在一起，那些适应性好的个体的染色体经常比适应性差的个体染色体有更多的繁殖机会。

③ 繁殖过程是进化发生的时刻。变异可以使生物体子代的染色体不同于它们父代的染色体。通过结合两个父代染色体中的物质，重组过程可以在子代中产生有很大差异的染色体。

④ 生物进化没有记忆。有关产生个体的信息包含在个体携带的染色体的集合及染色体的编码结构中，这些个体会很好地适应它们的生存环境。

总之，生物在进化过程中，经过优胜劣汰的自然选择，会使种群逐步优化。经过长期的演化，优良物种得以保留。不同的环境，不同物种的基因结构，导致了最终的物种群的不同，

但它们都有一个共同的特征：最能适应自己所处的生存环境。

受达尔文进化论“物竞天择，适者生存”思想的启发，Holland 教授把优化问题求解与生物进化过程对应起来，将生物进化的思想引到复杂问题求解中，提出了求解优化问题的遗传算法。

遗传算法首先对优化问题的解进行编码，编码后的一个解称为一个染色体，组成染色体的元素称为基因。一个群体由若干染色体组成，染色体的个数称为群体的规模。与自然界中的生存环境相对应，遗传算法用适应函数表示环境，它是已编码的解的函数，是一个适应环境程度的评价。适应函数的构造一般与优化问题的指标函数相关。在简单情况下，直接用指标函数或者指标函数经过简单的变换后作为适应函数使用。在一般情况下，适应函数值表示所对应的染色体适应环境的能力强弱，适应函数起着自然界中环境的作用。当适应函数确定后，自然选择规律将以适应函数值的大小来决定一个染色体是否继续生存的概率。生存的染色体称为种群，它们中的部分或者全部以一定的概率进行交配、繁衍，从而得到下一代的群体。交配是一个生殖过程，发生在两个染色体之间。作为双亲的两个染色体在交换部分基因后，繁殖出两个新的染色体，即问题的新解。交配是遗传算法区别于其他优化算法的最主要特征。在进化过程中，染色体的某些基因可能发生变异，即表示染色体的编码发生了某些变化。一个群体的进化需要染色体的多样性，而变异对保持群体的多样性具有一定的作用。

表 3.3 给出了遗传算法与生物进化之间的对应关系。

表 3.3　遗传算法与生物进化之间的对应关系

遗传算法	生物进化	遗传算法	生物进化
适应函数	环境	编码的元素	基因
适应值函数	适应性	被选定的一组解	群体
适应函数值最大的解被保留的概率最大	选择	根据适应函数选择的一组解（以编码形式表示）	种群
问题的一个解	个体	以一定的方式由双亲产生后代的过程	交配
解的编码	染色体	编码的某此分量发生变化的过程	变异

选择、交配和变异是遗传算法的三个主要操作。

依据适应函数值的大小，选择操作从规模为 N 的群体中随机地选择若干染色体构成种群，种群的规模可以与原来群体的规模一致，也可以不一致。这里假设二者的规模是一致的，即从群体中选择 N 个染色体构成种群。虽然种群的规模与群体的规模是一致的，但二者并不完全一样，因为适应函数值大的染色体可能多次从群体中被选出，而适应值小的染色体可能失去被选中的机会。因此，一些适应函数值大的染色体可能重复出现在种群中，而一些适应函数值小的染色体则可能被淘汰。这体现的正是自然界中的“优胜劣汰，适者生存”选择规律。

从群体中选择存活的染色体有多种方法，常用的有“赌轮盘”（适应函数值大的对应较宽格子）和“确定法”等。

交配发生在两个染色体之间，由两个称为双亲的父代染色体经杂交以后，产生两个具有双亲的部分基因的新染色体。当染色体采用二进制形式编码时，交配过程是以这样一种形式进行的：设 a 和 b 是两个交配的染色体，即

$$a: a_1 a_2 \cdots a_i a_{i+1} \cdots a_n$$

$$b: b_1 b_2 \cdots b_i b_{i+1} \cdots b_n$$

其中，$a_i, b_i \in \{0,1\}$。随机产生一个交配位设为 i，则 a 和 b 两个染色体从 $i+1$ 以后的基因进行交换，得到两个新的染色体，如图 3-35 所示。

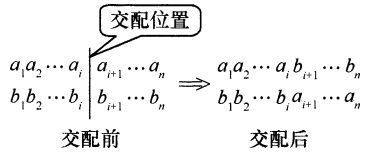

图 3-35　两个染色体交配

例如，对于 x_1=11001 和 x_2=01111 两个二进制编码的染色体，当交配位等于 2 时，则产生 y_1 和 y_2 两个子代染色体

$$\left.\begin{aligned} x_1 &= 11001 \\ x_2 &= 01111 \end{aligned}\right\} \Rightarrow \begin{aligned} y_1 &= 11111 \\ y_2 &= 01001 \end{aligned}$$

在进化过程中，通常交配是以一定的概率发生的，而不是 100%地发生。

变异发生在染色体的某一个基因上，当以二进制编码时，变异的基因由 0 变成 1，或者由 1 变成 0。如对于染色体 $x=11001$，若变异位发生在第 3 位，则变异后的染色体变成了 $y=11101$。变异对于一个群体保持多样性是有好处的，但也有很强的破坏作用，因此变异总是以一个很小的概率来控制。

遗传算法的控制参数包括群体规模 N、算法停止准则及交配概率 p_c 和变异概率 p_m。

下面具体描述遗传算法。其中，每代中群体的规模是固定的，变量 t 表示当前的代数，以适应函数值最大者为最优解。

```
Procedure Genetic_Algorithm
  Begin
    设群体规模 N;
    随机生成 N 个染色体 x_i ( i=1,2,…,N) 作为初始群体;
    设交配概率 p_c 和变异概率 p_m ;    t=0;
    While 不满足停止准则 Do
      Begin
        对 x_i(i=1,2,…,N) 分别计算其适应函数值 F(x_i) ;
        对 x_i(i=1,2,…,N) 按 P(x_i)=F(x_i)/(Σ_{j=1}^{N} F(x_j)) 计算概率 P(x_i) ;
        根据计算概率 P(x_i) 从群体中随机地选择 N 个染色体，得到种群;
        根据交配概率 p_c 对种群中的染色体进行交配，其子代进入新的群体；原种群中未交配的染色体，则
        直接复制到新群体中;
        根据变异概率 p_m 对新群体中的染色体进行变异，并用变异后的染色体代替新群体中的原染色体;
        用新群体代替旧群体，t=t+1;
      End;
    Select 适应值最大的染色体，经解码后作为最优解输出;
End.
```

算法的停止准则一般可以通过设置进化的最大代数来定义，或者定义为当经过连续的几代进化后，得到的最优解没有任何变化时即停止。

所以，遗传算法具有如下特点：

① 遗传算法是一个随机搜索算法，适用于数值求解具有多参数、多变量、多目标的复杂最优化问题。

② 遗传算法对待求解问题的指标函数没有什么特殊的要求，如不要求连续性、导数存在、单峰值等假设，甚至不需要显式地写出指标函数。

③ 经过编码后，遗传算法几乎不需要任何与问题有关的知识，唯一需要的信息是适应函

数值的计算。它不需要使用者对问题有很深入的了解，也不需更多的求解技巧，只需要通过选择、交配和变异等简单操作便可求解复杂的问题，是一个比较通用的优化算法。

④ 遗传算法具有天然的并行性，适用于并行化求解。

遗传算法是一个随机搜索算法，程序的每次运行得到的结果可能是不一样的。那么，遗传算法得到的解的质量是否有保证呢？得到最优解的可能性又如何呢？

理论上，只要进化的代数足够多，遗传算法找到最优解的可能性非常大。实际中由于要考虑在可接受的有限时间内停止算法的问题，因此解的质量与算法的控制参数（如群体的规模、交配概率、变异概率和进化代数等）有很大关系。这些参数的选取问题将在后面讨论。

3.7.2 遗传算法的实现问题

1．编码问题

在用遗传算法求解问题时首先遇到的是编码问题。将问题的解以适合遗传算法求解的形式进行编码，称为遗传算法的表示。而交配、变异等操作与编码形式有关。因此在对问题进行编码时，需要考虑交配和变异问题。最简单的编码是二进制形式，还有整数编码、实数编码、树编码等。采用什么样的编码形式与具体问题有关，下面给出采用二进制编码形式进行编码的例子。

【例 3.14】 旅行商问题的遗传算法编码。

对于 n 个城市的旅行商问题，可以用一个矩阵表示一个可能的解。如对于 4 个城市的旅行商问题，如下矩阵可以表示一个可能解 $BADCB$：

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

其中，行表示不同的城市，列表示城市的访问顺序。若第 i 行第 j 列为 1，则表示第 j 个访问的城市是城市 i，并默认最后一个城市之后访问第一个城市。若按行展开该矩阵，则该问题可能的解可以用一个 4×4 的二进制向量表示：0100100000010010。n 城市的旅行商问题可以用 $n\times n$ 位二进制向量表示一个可能的旅行路线。

$n\times n$ 位二进制向量所有可能的编码空间为 $2^{n\times n}$，而一个对称的 n 城市旅行商问题的可能解的个数为 $n!/2$，只占编码空间非常小的比例。以 $n=10$ 为例，编码空间为可能解空间的 7.0×10^{23} 倍，可能解在整个状态空间中是非常稀疏的，交配和变异所产生的是大量的非可能解。可以想象，对于旅行商问题来说，这样的编码方式将导致求解效率低下。

对于 n 城市旅行商问题，一种很自然的想法是，对城市进行编号，每个城市分别用 1～n 之间不同的整数表示，n 个整数的一个排列就代表了旅行商问题的一个可能解。这就是所谓的整数编码问题。整数编码要解决的是如何交配和变异的问题，这将在后面介绍。

2．遗传算法的评价

理论上，当进化的代数趋于无穷时，遗传算法找到最优解的概率为 1，即保证了遗传算

法的收敛性。但在实际计算时，希望随时了解遗传算法的发展情况，监视算法的变化趋势，常用的算法如下。

① 当前最好法：在每代的进化过程中记录得到的当前最好解，通过最好解的变化了解算法的变化趋势。不同算法之间也可以通过最好解的变化情况进行横向比较。

② 在线比较法：用当前代中染色体的平均指标函数值来刻画算法的变化趋势。计算方法如下：

$$v_{\text{on\_line}} = \frac{1}{T}\sum_{t=1}^{T} f(t) \tag{3.18}$$

其中，T 为当前代中染色体的个数，$f(t)$ 为第 t 个染色体的指标函数值。

在以最大化为问题的优化目标时，在进化过程中，每代的值可能出现一些波动，但总的趋势应该是上升的，并逐渐趋于稳定。

③ 离线比较法：与在线比较法有些类似，但是用进化过程中每代的最好解的指标函数值的平均值来评价算法的进化过程。计算方法如下：

$$v_{\text{off\_line}} = \frac{1}{T}\sum_{t=1}^{T} f^*(t) \tag{3.19}$$

其中，T 是到目前为止的进化代数，$f^*(t)$ 是第 t 代中染色体的最好指标函数值。在以最大化为问题的优化目标时，随着算法的进化，该值具有上升的趋势。

以上每种方法都可以监控算法的进化趋势，掌握进化情况，从而决定算法是否停止。

3．适应函数

由于任何一个最小化优化问题都可以转化为最大化优化问题，因此在下面的讨论中均假定以最大化为问题的优化目标。

一般可以直接选取问题的指标函数作为适应函数，如求函数 $f(x)$ 的最大值就可以直接采用 $f(x)$ 为适应函数。但有时，函数 $f(x)$ 在最大值附近的变化可能非常小，以至于它们的适应值非常接近，很难区分出哪个染色体占优。此时希望定义新的适应函数，要求该适应函数与问题的指标函数具有相同的变化趋势，但变化的速度更快。加速方法一般有 2 种。

① 非线性加速适应函数：利用已有的信息构造适应函数，即

$$f'(x) = \begin{cases} \dfrac{1}{f_{\max} - f(x)}, & f(x) < f_{\max} \\ M, & \text{其他} \end{cases} \tag{3.20}$$

其中，$f(x)$ 为问题的指标函数，$f_{\max}$ 是当前得到的最优指标函数值；M 是一个充分大的数，M 值的大小将影响到算法以怎样的概率选取种群。M 不一定是一个常量，可以随着算法的进行而变化，开始时可以相对小一些，以保证种群的多样性，然后逐步增大。

② 线性加速适应函数：将当前得到的最优指标函数值放大为指标函数值平均值的 M 倍。其中，平均值为 $\dfrac{\sum_{i=1}^{m} f(x_i)}{m}$，并在进行这种变换时保持变换前后的不变性。

另一种定义适应函数的方法是利用染色体指标函数值从小到大的排列序号作为适应函数值，利用该值采用“赌轮盘”的方法得到每个染色体被选中的概率。

4．二进制编码的交配规则

在遗传算法中，常用的交配规则如下。

① 双亲双子法：即前面已经介绍过的最常用的交配方法，见图 3-35。

② 变化交配法：对双亲双子法的一种改进。在有些情况下，采用双亲双子法交配得到的两个染色体与其双亲完全一致，这样的交配起不到任何作用。如两个染色体的前 3 位完全一致，因此当交配位选择在前 3 位时，其子染色体将与两个父染色体完全一致。变化交配法就是在随机产生交配位时排除掉这样的交配位。

③ 多交配位法：不只产生一个交配位，而是产生多个交配位进行交配，在交配时采用交配区间交替进行。对于长度为 n 的染色体，设有 m 个交配位，形成 m+1 交配区间（为叙述方便而加 1 的，其中 $m+1 \leqslant n$ ）。在交配的时候，两个父染色体交换奇数（或偶数）的交配区间内的基因，而不交换偶数（或奇数）的交配位之间的基因。图 3-36 给出了 3 个交配位（分别为 2、4、6）的情况（$m=3$），因 $n=7$，便形成 4 个交配区间，于是第 1、3 区间不交换，而第 2、4 区间交换。

$$\begin{array}{l} 1\ 1 \mid 0\ 1 \mid 0\ 0 \mid 1 \\ 1\ 1 \mid 0\ 0 \mid 0\ 1 \mid 0 \end{array} \Rightarrow \begin{array}{l} 1\ 1 \mid 0\ 0 \mid 0\ 0 \mid 0 \\ 1\ 1 \mid 0\ 1 \mid 0\ 1 \mid 1 \end{array}$$

图 3-36　多交配位法

④ 双亲单子法：两个染色体交配后只产生一个子染色体，一般是从普通交配法得到的两个子染色体中随机地选择一个，或者选择适应函数值大的那个子染色体。

5．整数编码的交配规则

如果采取交换两个父染色体的部分基因的方法进行交配，所产生的子染色体就不一定刚好是 1～n 的一个排列，从而产生无效解。为此必须选择能够保持编码有效性的交配规则。下面以旅行商问题为例，介绍 4 种整数编码的交配规则。

① 常规交配法：与二进制编码的双亲双子法类似。设有父代 1 和父代 2，交配后产生子代 1 和子代 2。随机选取一个交配位，子代 1 交配位之前的基因选自父代 1 交配位之前的基因。交配位后的基因从父代 2 中按顺序选取那些没有出现过的基因。子代 2 也进行类似处理，如图 3-37 所示。

$$\begin{array}{l} \\ \text{父代 1} \\ \text{父代 2} \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & \vert & 5 & 6 & 7 & 8 \\ 5 & 2 & 1 & 7 & \vert & 3 & 8 & 3 & 6 \end{bmatrix} \Rightarrow \begin{array}{l} \\ \text{子代 1} \\ \text{子代 2} \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & \vert & 5 & 7 & 8 & 6 \\ 5 & 2 & 1 & 7 & \vert & 3 & 4 & 6 & 8 \end{bmatrix}$$

交配位　　交配位

图 3-37　常规交配法

② 基于次序的交配法。对于两个选定的染色体父代 1 和父代 2，首先随机地选定一组位置。设父代 1 中与所选位置相对应的数字从左到右依次为 $x_1, x_2, \cdots, x_k$，然后从父代 2 中也找到这 k 个数字，并从父代 2 中把它们去除，这样在父代 2 中就有了 k 个空位置。将 $x_1, x_2, \cdots, x_k$ 依次填入父代 2 的空位置，就得到了交配后的一个染色体子代 1。采用同样的方法可以得到另一个染色体子代 2。例如：

父代 1：	1	2	3	4	5	6	7	8	9	10
父代 2：	5	9	2	4	6	1	10	7	3	8
所选位置：	*	*		*			*			

父代 1 中与所选位置相对应的数字为 2、3、5、8。从父代 2 中找出这些数字，并删除它

们，其中 *B* 表示空位置：

父代 2：　*B*　9　*B*　4　6　1　10　7　*B*　*B*

将 2、3、5、8 按序填入父代 2 的空位置，得到子代 1：

子代 1：　2　9　3　4　6　1　10　7　5　8

同理可得，子代 2：

子代 2：　1　9　3　4　5　2　6　8　7　10

③ 基于位置的交配法。同方法②一样，对于两个选定的染色体父代 1 和父代 2，首先随机产生一组位置。子代 1 在这些位置上的基因从父代 2 中直接得到，子代 1 的其他位置的基因按顺序从父代 1 中选取那些不相重的基因。子代 2 也进行类似处理。父代 1 和父代 2 同②时，有：

子代 1：　1　9　2　3　6　4　5　7　8　10

子代 2：　9　2　3　4　5　6　1　8　10　7

④ 基于部分映射的交配法。对于两个选定的染色体父代 1 和父代 2，随机产生两个位置，两个父代在这两个位置之间的基因形成对应对集，然后分别在两个父代内用每个对应对各自地去交换基因对，从而产生两个子代。如：

父代 1：　2　6　4　3　8　1　5　7　9

父代 2：　8　5　1　7　6　2　4　3　9

所选位置：　　*　　*

按所选位置可得对应对集{3:7,8:6,1:2}，然后按每个对应对各自地交换父代 1 中的基因对，得到子代 1：

子代 1：　*1*　8　4　7　6　*2*　5　3　9

同理，按每个对应对各自地交换父代 2 中的基因对，得到子代 2：

子代 2：　6　5　*2*　3　8　*1*　4　7　9

6．变异规则

变异发生在某个染色体的某个基因上，将可变性引入群体中，增强了群体的多样性，从而提供了从局部最优中逃脱出来的一种手段。

当问题以二进制编码形式表示时，随机地产生一个变异位，被选中的基因由“0”变为“1”，或者由“1”变为“0”。

当问题以整数形式编码时，被选中的基因可以由一个整数随机地变为另一个整数，但必须考虑染色体的合理性。这时要根据问题本身的性质，合理地定义变异方法。以旅行商问题为例，有以下 3 种变异方法。

① 基于位置的变异：随机地产生两个变异位，然后将第二个变异位上的基因移动到第一个变异位之前。如假定两个变异位分别为 2 和 5，则对于整数编码 2 136 457，变异后为 2 413657。

② 基于次序的变异：随机地产生两个变异位，然后交换这两个变异位上的基因。如假定两个变异位分别为 2 和 5，则对于整数编码 2 136 457，变异后为 2 436 157。

③ 打乱变异：随机地选取染色体上的一段，然后打乱在该段内的基因顺序。如假定选取段为第 2～5 位，则对于整数编码 2 136457，其可能的一种变异结果为 2 463157。

变异虽然可以带来群体的多样性，但因其具有很强的破坏性，因此一般通过一个很小的变异概率来控制它的使用。

对遗传算法的性能评价问题是一件很困难的事情，这里不再展开讨论。

3.8 约束满足法

人工智能中的许多问题可看作约束满足问题，这类问题的求解任务是寻找某问题的状态，使它满足一个给定的约束集。许多设计任务可看作约束满足问题，因为必须在固定的时间、耗费及材料的限制条件下进行设计。密码算术问题就是这类问题的一个例子。

【例 3.15】 求解密码算术问题。考虑以字母表示的算术问题，给每个字母赋予一个十进制数字，使得它们满足：

$$\begin{array}{r} \text{s e t d} \\ +\text{m o r e} \\ \hline \text{m o t e y} \end{array}$$

其中，若同一字母出现多次，则每次必须赋予同一个数字，且任意两个不同字母不能赋予同一个数字。

按约束满足法，此密码算术问题描述如下。

（1）问题

$$\begin{array}{r} \text{s e t d} \\ +\text{m o r e} \\ \hline \text{m o t e y} \end{array}$$

（2）约束集

① 任意两个不同的字母不能有相同的数值。

② 按普通算术运算（加、进位等）规则约束。

（3）初始问题状态

$$\begin{array}{ll} s=? & c_1=? \\ e=? & c_2=? \\ & c_3=? \\ & c_4=? \\ \vdots & \\ r=? & \\ y=? & \end{array}$$

其中，c_1,c_2,c_3,c_4 表示算式从低位到高位的进位，初始问题状态表示所有字母数值均未知。

设最初的约束条件以表形式存储，该初始约束表与初始问题状态同时存在。

（4）目标问题状态

所有字母赋予一个数字且所有约束都得到满足。

这里的问题求解过程不仅包含相应状态的变换，还有一个随着得到的局部解而变化的约束表。问题求解过程是通过反复循环来完成的。每循环一次，要做两件事：① 运用约束推论规则生成相关的新的约束；② 运用字母赋值规则去生成由目前的约束集要求的所有赋值，再

挑选另一规则去生成一个假定赋值，接着生成下一循环的附加约束。

为了执行这一循环，设置约束生成过程和问题求解过程来共同未完成。约束生成过程的主要功能是产生新的约束；而问题求解过程的主要功能是在约束条件下，对字母赋值（包括确定的或不确定的）。

图 3-38 表示了解决该问题的前几次循环的结果，每一层只列出了增加的约束。

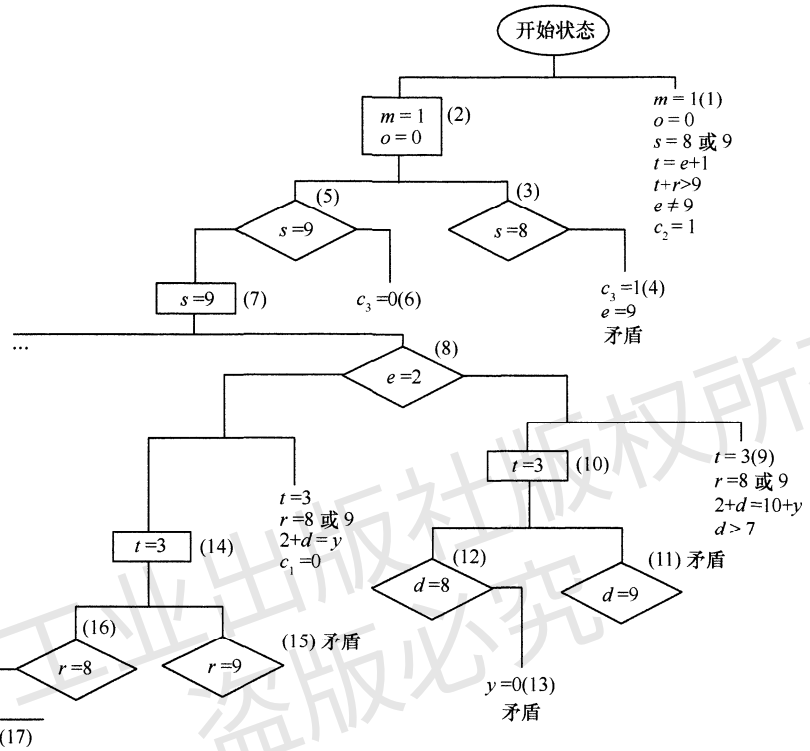

图 3-38　密码算术问题的约束满足法求解过程

图 3-38 中的棱形框表示猜测，这种猜测是在约束生成过程中产生新的约束条件下由问题求解过程对能确定的字母赋值后所做出的猜测。如问题不能继续解下去，可再做出猜测。当对某一字母赋予猜测值后，便可开始执行约束生成过程，进入新的循环。矩形框则表示赋值。另外，未加框的为约束条件。

下面针对该密码算术问题看一下约束满足法的求解过程。

开始时，约束生成过程运行，此时可得到如下约束条件：

① m=1 。

② s=8 或 9。因为要产生进位，$s+m+c_3$ 至少为 10，且最多为 11，但 m=1，而不同的字母代表的数字不同，所以 o 必为 0。

同时，约束条件 $t=e+1$，$e\neq 9$，…。

约束条件产生后，问题求解过程根据约束条件对 m 和 o 赋值，即 m=1，o=0，至此，第一次循环运行完毕。

为了将问题继续做下去，现需要猜测，即选择一个规则产生一个假定赋值，如赋予 s 值为 8，然后开始下一次循环。

约束生成过程再运行，产生约束 c_3=1，e=9，这与先前的约束条件 $e \neq 9$ 矛盾，于是回溯处理 s=9 的情况，再运行问题求解过程，将 s 赋值为 9，然后运行约束生成过程，产生约束条件 c_3=0，为了继续搜索，又要猜测，如赋予 e 值为 2。

约束生成过程运行，产生新约束：

① t=3。因为 t=e+o+c_2 必为 1，否则 t=e。

② r=8 或 9。因为 $r+t+c_1=2$ 或 12，但 3 个正整数之和不可能为 2，所以 $r+3+0$（或 1）=12，即 r=8 或 9。

③ $2+d$=y 或 $2+d$=10+y。

至此，约束生成过程可先选 $2+d$=10+y，由此产生一额外约束，即 $d>7$，因为 $2+d$ 至少为 10 才能产生进位，所以 $d>7$。

再将控制转向问题求解过程，立即得 $t=3$。现在必须猜测，如先猜测约束的字母 d，它为 8 或 9。

又一次循环，再启动约束生成过程。当 d=9 时，出现了 s 和 d 都为 9，有矛盾，然后回溯，处理 d=8，导出 y=0，这就出现了 o 和 y 都为 0，有矛盾的情况。从而使整个过程回溯到上一个选择点，即转向 $2+d=y$ 处，循环从那里继续……

直到该密码算术问题达到目标问题状态（有些问题可能会无解）。图 3-38 中括号内的数字为约束满足法求解的顺序。

本问题的解为

$$\begin{array}{r} 9567 \\ +1085 \\ \hline 10652 \end{array}$$

从上述过程可以看出，为了提高约束满足法求解问题的正确性和效率，有些步骤的处理是很重要的：

① 对约束生成过程而言，它产生约束的规则必须使约束避免产生不合逻辑的结果，也不需生成所有合法的约束，具有包含关系的约束条件不必都生成出来。

② 先对哪些项进行赋值猜测，也就是先对哪些变量进行猜测，将会影响搜索的效率，如对约束条件多的变量先做猜测是有利的。

③ 在每次循环时，选择哪个结点进行（回溯点的选择）也是很讲究的。因此，在解决实际问题时必须考虑如何选择策略。本例实际上用的是深度优先策略。

当然，上面只是解决约束满足问题的一种方法，还可以有其他途径，感兴趣的读者可参阅有关文献。

本章对重要的状态空间搜索法，如局部搜索算法、模拟退火算法和遗传算法等分别进行了介绍，使大家对这些算法有简单了解。如果想用这些方法求解复杂问题，还需要进行更深入的学习和研究。

习 题 3

3.1 分析深度和广度优先算法的优缺点。你能举出它们的正例和反例吗？

3.2 在深度优先搜索中，每个结点的子结点是按某种次序生成和扩展的，在决定生成子状态

的最优次序时，应该用什么标准来衡量？

3.3　下面的问题应使用广度优先搜索还是深度优先搜索？为什么？

（1）国际象棋程序

（2）医疗诊断程序

（3）寻找使机器人从 A 点到 B 点的路径的规划程序。

（4）一个决定从原料到最终产品的生产步骤的最优次序的程序。

（5）用于判断两个命题演算表达式是否等同的程序。

3.4　在四皇后问题中，设应用每一次摆放规则的代价值均为 1，试描述这个问题的 $h^*(x)$ 启发函数的一般特征。你是否认为，任何 h 启发函数对引导搜索都是有用的？

3.5　对传教士与野人过河（$M=C=5$，一船可载 3 人）问题，定义两个 h 启发函数（非零），并给出这两个函数的 A 算法搜索图。讨论用这两个启发函数求解该问题时是否能得到最优解。

3.6　讨论一个 h 启发函数在搜索期间可以得到改善的几种方法。

3.7　求解旅行商问题。给出如下城市表，两城市之间的距离用 d 表示。请用 A 或 A*算法找出一条旅行商由城市 A 出发，最后回到城市 A 的最短路线。

$$d(A,B)=6,\ d(B,C)=6,\ d(A,E)=7,\ d(A,C)=1,\ d(A,D)=5$$
$$d(B,D)=4,\ d(B,E)=3,\ d(C,D)=8,\ d(C,E)=2,\ d(D,E)=5$$

3.8　滑板游戏由三块黑色板和三块白色板组成，中间有一块为空（如图 3-39 所示），每一块板可移到邻近的位置上，代价为 1；也可跳过一块或两块到空位置上，代价为跳过的滑板数（1 或 2）。要求通过合法的转位，使所有的白色板都在黑色板的左侧（不考虑空位置）。

图 3-39　滑板游戏示意图

（1）用状态空间表示法来描述这个问题。

（2）分析该状态空间的复杂度。

（3）设计一个解决此问题的启发策略，并分析它的可采纳性、单调性及信息性。

3.9　用状态空间搜索法证明以下字符串 (((),()),(),((),())) 是由下列重写规则规定的语法中的一个句子 S。

（1）$S \leftarrow (\ )$　　（2）$A \leftarrow S$

（3）$A \leftarrow A,A$　　（4）$S \leftarrow (A)$

在这些重写规则中，规定箭号左边的符号可以代替箭号右边的子字符串，此子字符串可出现在该字符串中的任何位置上。

3.10　选择一个你所熟悉的领域，设计一个状态搜索系统（非形式化）。

3.11　什么是与/或图的解图？试说明与/或图的可采纳启发式搜索算法 AO*的要点。如图 3-40 所示，结点处标的是启发函数 $h(n)$ 的值，两结点之间的代价均设为 1，与结点处的代价值由累加计算而得，试画出求解的搜索过程。

3.12　数字重写问题的变换规则如下。

$$6 \rightarrow 3,3 \qquad 4 \rightarrow 3,1$$
$$6 \rightarrow 4,2 \qquad 3 \rightarrow 2,1$$
$$4 \rightarrow 2,2 \qquad 2 \rightarrow 1,1$$

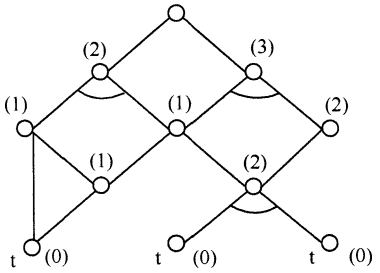

图 3-40　与/或图搜索图

如何用这些规则把数字 6 变换成一个由若干数字 1 组成的数字串。试用 AO*搜索算法求解，并给出搜索图。设 k-连接符的代价是 k 个单位，$h(x)$ 启发函数值规定为 $h(1)=0, h(n)=n(n\neq 1)$。

3.13　试证明如下定理：平行四边形的两对角线彼此相交后相互等分。应用与/或树画出搜索求证步骤，并指明构成此定理证明的解树。

3.14　博弈搜索过程为何总是由当前状态向目标状态搜索，而不是由目标状态回溯到当前状态？什么样的游戏可采用回溯策略？

3.15　图 3-41 中运用了极大极小化搜索，请指出其最优走步。

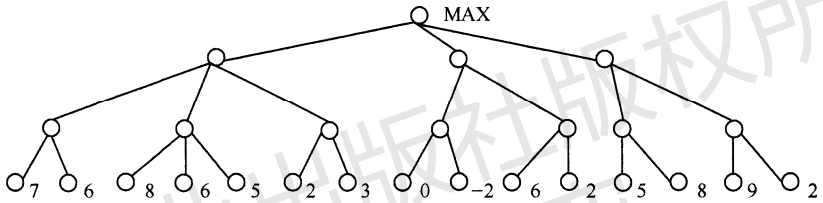

图 3-41　博弈隐含图之一

3.16　在如图 3-41 所示的博弈树中，说明 α-β 过程是如何剪枝的，并给出相应的搜索树（从左到右）。

3.17　在图 3-42 中运用从左到右的 α-β 剪枝技术和从右到左的 α-β 剪枝技术。试讨论两者之间的不同。

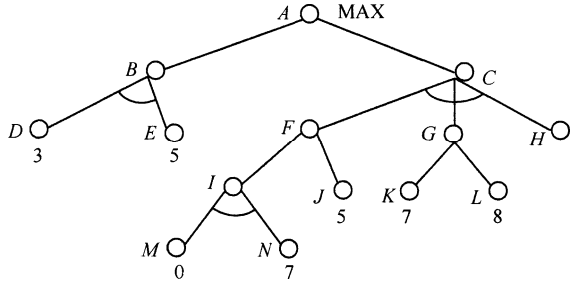

图 3-42　博弈隐含图之二

3.18　编写一个 α-β 搜索算法。

3.19　试分析模拟退火算法对爬山法有何改进？

3.20　分别用回溯方法和局部搜索方法实现一个求解 N 皇后问题的程序，当 N 比较大时，比较两种方法所用的求解时间。

3.21　用模拟退火算法求解旅行商问题，并讨论其求解效率问题。

3.22　用遗传算法求解旅行商问题，并与模拟退火算法进行比较。