第 1 章　搜索问题

人类的思维过程，可以看作是一个搜索的过程。从小学到现在，你一定遇到过很多种的智力游戏问题，比如说传教士和野人问题。有 3 个传教士和 3 个野人来到河边准备渡河，河岸有一条船，每次至多可供 2 人乘渡。问传教士为了安全起见，应如何规划摆渡方案，使得任何时刻，在河的两岸以及船上的野人数目总是不超过传教士的数目（但允许在河的某一岸或者船上只有野人而没有传教士）。如果你来做这个智力游戏的话，在每一次渡河之后，都会有几种渡河方案供你选择，究竟哪种方案才有利于在满足题目所规定的约束条件下顺利过河呢？这就是搜索问题。经过反复努力和试探，你终于找到了一种解决办法。在高兴之余，你马上可能又会想到，这个方案所用的步骤是否最少？也就是说它是最优的吗？如果不是，如何才能找到最优的方案？在计算机上又如何实现这样的搜索？这些问题就是本章我们要介绍的搜索问题。

一般地，很多问题可以转化为状态空间的搜索问题。

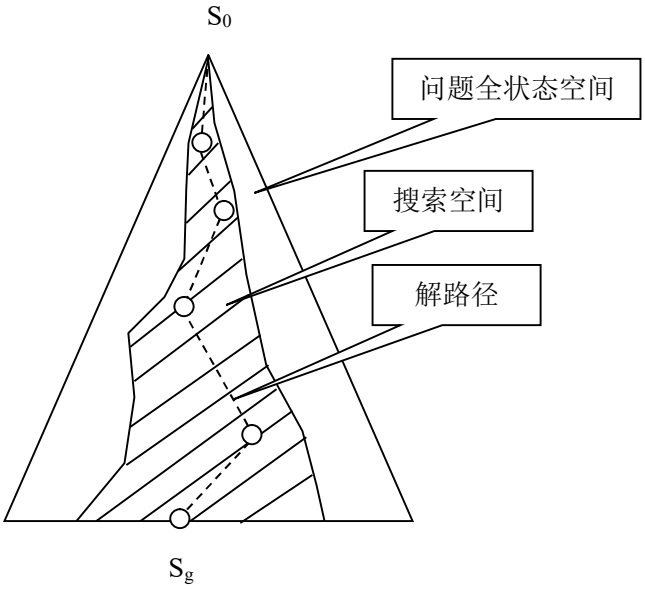


图 1.1 搜索空间示意图

比如传教士和野人问题，当我们用在河的左岸的传教士人数、野人人数和船的情况表示问题时，该问题的初始状态可以用三元组表示为（3，3，1），结束状态可以表

示为（0，0，0），而中间状态则可以表示为（2，2，0）、（3，2，1）、（3，0，0）……等，每个三元组对应了三维空间上的一个点，而问题的解，则是一个合法状态的序列，其中序列的第一个状态是问题的初始状态，而最后的一个状态则是问题的结束状态。介于初始状态和结束状态之间的则是中间状态。除了第一个状态外，该序列中任何一个状态，都可以通过一条规则，由与他相邻的前一个状态转换得到。

图 1.1 给出了一个搜索问题的示意图。其含义是如何在一个比较大的问题空间中，只通过搜索比较小的范围，就找到问题的解。使用不同的搜索策略，找到解的搜索空间范围是有区别的。一般来说，对大空间问题，搜索策略是要解决组合爆炸的问题。

通常搜索策略的主要任务是确定如何选取规则的方式。有两种基本方式：一种是不考虑给定问题所具有的特定知识，系统根据事先确定好的某种固定排序，依次调用规则或随机调用规则，这实际上是盲目搜索的方法，一般统称为无信息引导的搜索策略。另一种是考虑问题领域可应用的知识，动态地确定规则的排序，优先调用较合适的规则使用，这就是通常称为启发式搜索策略或有信息引导的搜索策略。

本章及下一章将介绍几个常用的搜索策略。

1.1 深度优先搜索（DEPTH-FIRST-SEARCH）

深度优先策略属于盲目搜索的一种。所谓深度优先搜索，简单地说是这样一种策略：每次优先搜索深度最深的节点。一般是先将规则给定一个固定的排序，在搜索时，对当前状态（搜索开始时，当前状态是初始状态）依次检测每一条规则，在当前状态未使用过的规则中找到第一条可应用规则，应用于当前状态，得到的新状态重新设置为当前状态，并重复以上搜索。如果当前状态无规则可用，或者所有规则已经被试探过仍未找到问题的解，则将当前状态的前一个状态（即直接生成该状态的状态）设置为当前状态。重复以上搜索，直到找到问题的解，或者试探了所有可能后仍找不到问题的解为止。

这样，由于每次都优先扩展新生成的状态，刚好体现了深度优先这一思想。

下面我们以一个简单的迷宫问题为例，说明深度优先搜索策略的基本思想。

迷宫问题如图 x 所示，迷宫的入口坐标为（0,0），出口坐标为（5,4），图中的粗实线是可以通行的道路。问如何找到一条从入口到出口的路线。

图中箭头给出了按照深度优先搜索策略进行搜索的过程。这里的规则，假定在任何一个路口，按照“上、右、下、左”的次序前进，当无路可走或者试探过所有前进方式后则回到当前深度最深的前一个路口。从（0,0）出发，向上前进到（0,1），前进到（1,1），再一次前进到（1,2）、（1,3）。由于（1,3）无路可走，则回到它的前一个路

口（1,2）。在（1,2）处已经尝试过向上前进，这次则向右前进到（2,2），然后到达（3,2）。在（3,2）处先向上走到达（3,3），由于再无路可走，又回到（3,2），向右到达（4,2）。（4,2）无路可走再次回到（3,2）。到此为止（3,2）处已经尝试了所有可能的前进方式，只能再回到（2,2）。在（2,2）处尝试还没有走过的向下，进入到（2,1）。由（2,1）逐次前进到（3,1）、（4,1）、（5,1）、（5,2）、（5,3），最后到达出口（5,4）。到此为止，搜索结束。那么，如何得到从迷宫入口到出口的路线呢？在搜索过程中，每前进一步都要记录所走的方向（向可以走的方向前进一步，可以认为是一条规则），如果发生了回溯到前一个状态的情况，则删除该方向的记录。对照图上标示的箭头，如果一个路段旁边有两个方向相反的箭头，则这两个箭头都需要删除，最后保留的箭头所指示的路线，就是从入口到出口的路线。在这个例子中，得到路线就是（0,0）、（0,1）、（1,1）、（1,2）、（2,2）、（2,1）、（3,1）、（4,1）、（5,1）、（5,2）、（5,3）、（5,4）。

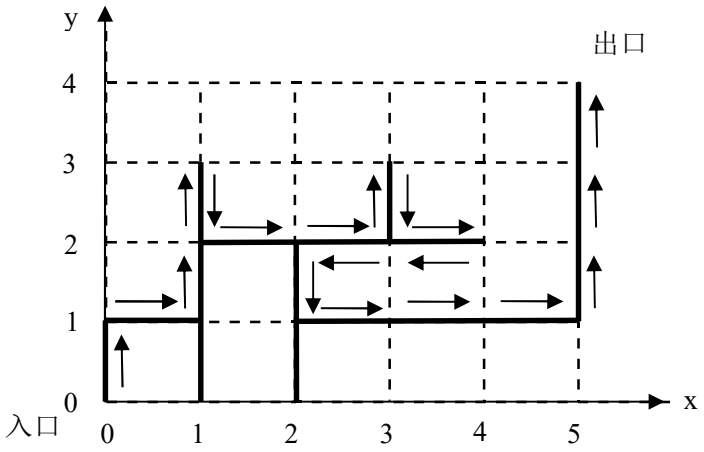


图 x，迷宫问题

深度优先搜索策略可以有多种实现的方法，其中递归法是一种最直接的实现方法。下面定义一个递归过程 DEPTH-FIRST-SEARCH，实现深度优先搜索策略。其功能是：如果从当前状态 DATA 到目标状态有路径存在，则返回以规则序列表示的从 DATA 到目标状态的路径；如果从当前状态 DATA 到目标状态没有路径存在，则返回 FAIL。

下面用类 LISP 语言的形式给出深度优先搜索算法，并通过四皇后问题说明算法的运行过程。在下面的算法中，“；”号后面的内容表示注释。

1．递归过程 DEPTH-FIRST-SEARCH(DATA)

递归过程 DEPTH-FIRST-SEARCH(DATA)

①　IF TERM(DATA)，RETURN NIL；TERM 取真即找到了目标，则过程返回空表 NIL。也就是说，如果当前状态就是目标，则不需再搜索就结束。

②IF DEADEND(DATA)，RETURN FAIL；DEADEND 取真，表示该状态不合法，则过程返回 FAIL，必须回溯到前一个状态。

③RULES：＝APPRULES(DATA)；APPRULES 计算 DATA 的可应用规则集，依某种原则（任意排列或按启发式准则）排列后赋给 RULES。

④LOOP：IF NULL(RULES)，RETURN FAIL；如果 NULL 取真，则表示规则已用完仍未找到目标，过程返回 FAIL，必须回溯到前一个状态。

⑤R：＝FIRST(RULES)；取第一个可应用规则。

⑥RULES：＝TAIL(RULES)；从可应用规则表中删除第一个规则。

⑦RDATA：＝GEN(R，DATA)；将规则 R 作用于当前状态 DATA，生成 DATA 的下一个状态 RDATA。

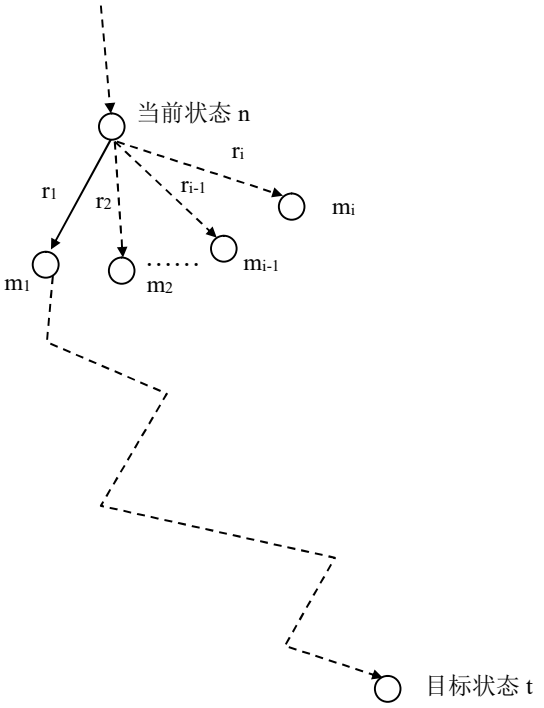
⑧PATH：＝DEPTH-FIRST-SEARCH (RDATA)；对 RDATA 递归调用本过程。

⑨IF PATH＝FAIL，GO LOOP；当 PATH＝FAIL 时，表示递归调用未能找到从 RDATA 到达目标的路径，则转移尝试调用 DATA 的下一条规则进行新的尝试。

⑩RETURN CONS(R，PATH)；到达这一步说明递归调用找到了从 RDATA 到目标的路径 PATH。这样将从 DATA 产生 RDATA 的规则 R 从前插入到 PATH，就得到了从 DATA 到达目标的路径。过程返回表示该解路径的规则表（或局部解路径子表）。

递归过程 DEPTH-FIRST-SEARCH 是将循环与递归结合在一起的。下面通过一个示意图，说明一下该算法的思想。

如下图所示，当前状态 n 相当于算法中的 DATA，(r_1，r_2，…，r_{i-1}，r_i)是 n 的可应用规则集 RULES，m_1、m_2、…、m_{i-1}、m_i 是 n 的 i 个子状态，它们分别可以通过 r_1、r_2、…、r_{i-1}、r_i 这 i 个规则作用于状态 n 得到。t 是目标状态。



为了得到从当前状态 n 到目标状态 t 的路径，可以从两个方面考虑。一是要探索 n 的 i 个子状态 m_1、m_2、…、m_{i-1}、m_i 中，通过哪一个状态可以到达目标状态 t。这一点，算法是通过循环实现的，每次从 n 的可应用规则集中，选取一个规则作用于 n。所体现的是“横向”探索。二是“纵向”探索。为了探索 n 的某一个子状态 m_k (k=1，2，…，i)是否可以达到目标状态 t，算法通过递归来完成这个试探。

整个算法的思想就是：要想求得从 n 到 t 的路径，首先查看 m_1 到 t 是否有路径存在。如果从 m_1 到 t 有路径存在，则在该路径前加上 r_1，就得到了从 n 到 t 的路径（这里的路径是用规则的集合表示的）。在试探 m_1 到 t 有没有路径的过程中，m_1 又成为了当前状态，又要探索 m_1 的子状态到 t 是否有路径存在，如此进行下去。递归所起的正是这样的作用。如果从 m_1 到 t 没有路径存在，算法则试探 m_2 到 t 是否存在路径，它同样也要试探 m_2 的子状态到 t 有无路径存在，等等。所以算法中循环和递归是交叉进行的，一方面是“横向”探索，一方面是“纵向”探索。

下面对这个算法作几点说明。首先解释一下其中的变量、常量、谓词、函数等符号的意义。变量符号 DATA、RULES、R、RDATA、PATH 分别表示当前状态、（某个状态的）可应用规则集、当前调用规则、新生成的状态（即 R 作用于 DATA 产生的状态）、当前解路径表。常量符号 NIL、FAIL、LOOP 分别表示空表、回溯点标记、

循环标号。当状态变量 DATA 满足结束条件时，TERM(DATA)取真；当 DATA 为非法状态时，DEADEND(DATA)取真；当规则集变量表 RULES 取空时，NULL 取真。函数 RETURN、APPRULES、FIRST、TAIL、GEN、GO、CONS 的作用是：RETURN 返回其自变量值；APPRULES 求可应用规则集；FIRST 和 TAIL 分别取可应用规则集的第一个规则和取除第一个规则以外的所有规则；GEN 调用规则 R 生成一个新状态；GO 执行转移；CONS 构造新表，把其第一个自变量加到一个表（第二个自变量）的前面。DEPTH-FIRST-SEARCH (RDATA)表示调用递归过程作用于新自变量上。

这里再说明一下该过程的运行情况。当某一个状态 S_g 满足结束条件时，算法才在第 1 步结束并返回 NIL，此时 DEPTH-FIRST-SEARCH (S_g)＝NIL。失败退出发生在第 2、4 步，第 2 步是处理不合法状态的回溯点，而第 4 步是处理全部规则应用均失败时的回溯点。若处在递归调用过程期间失败，过程会回溯到上一层继续运行，而在最高层失败则整个过程宣告失败退出。构造成功结束时的规则表在第 10 步完成。算法第 3 步实现可应用规则的排序，可以是固定排序或根据启发信息排序。

DEPTH-FIRST-SEARCH 过程只设置了两个回溯点，可用于求解 N－皇后这类性质的问题，下面以四皇后问题为例进一步说明算法的运行过程。

2．四皇后问题

在一个 4×4 的国际象棋棋盘上，一次一个地摆放四枚皇后棋子，摆好后要满足每行、每列和对角线上只允许出现一枚棋子，即棋子间不许相互俘获。

图 1.2 给出棋盘的几个状态，其中 a，b 满足目标条件，c，d，e 为不合法状态，即不可能构成满足目标条件的中间状态。

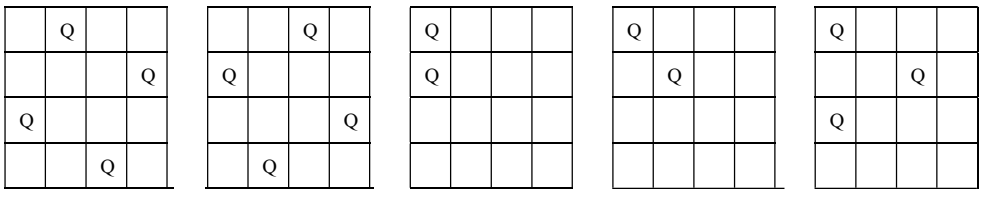


图 1.2 四皇后问题棋盘的几个状态

四皇后问题的一个状态可以用一个表表示，表的元素为 ij(1≤i, j≤4)，表示棋子所在的行和列。因最多只有 4 个棋子，故表元素个数最多为 4。

图 1.2 中 a，b 分别可以记为(12　24　31　43)和(13　21　34　42)，c，d，e 分别可以记为(11　21)，(11　22)，(11　23　31)等等。

可以简单地定义规则 R_{ij}(1≤i, j≤4)，表示在满足前提条件下，在第 i 行第 j 列摆放一个棋子。

当规则序列以 R_{11}，R_{12}，R_{13}，R_{14}，R_{21}，…，R_{44} 这种固定排序方式调用 DEPTH-FIRST-SEARCH 时，四皇后问题的搜索示意图如图 1.3 所示（为简单起见，每个状态只写出其增量部分）。可以看出，总共回溯 22 次，主过程结束时返回规则表 (R_{12}　R_{24}　R_{31}　R_{43})。

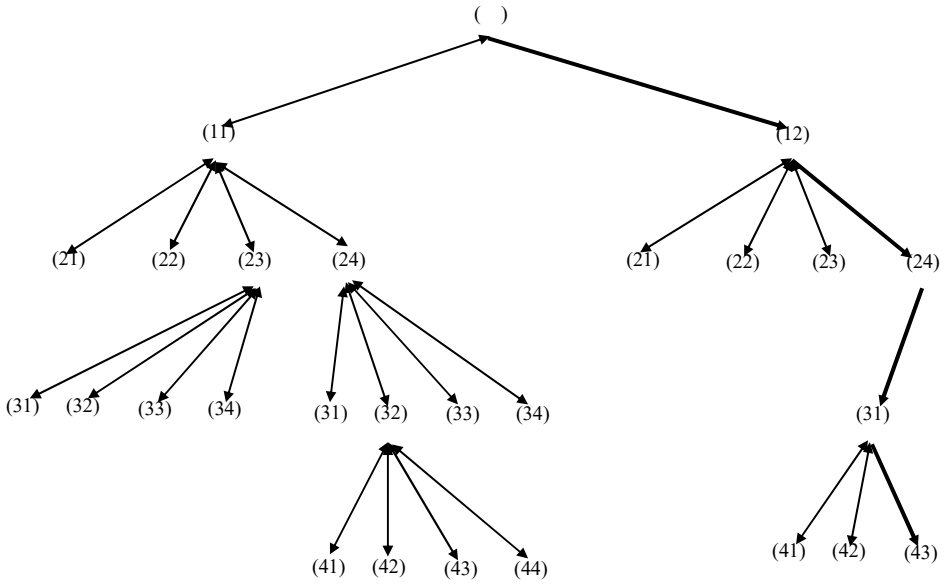
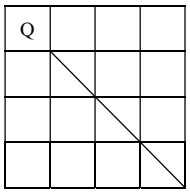
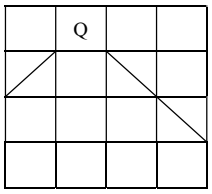


图 1.3 固定排序搜索树

在深度优先搜索策略中，也可以通过引入一些与问题有关的信息来加快搜索到解的速度。对于皇后问题来说，由于每一行、每一列和每一个对角线，都只能放一个皇后，当一个皇后放到棋盘上后，不管它放在棋盘的什么位置，它所影响的行和列方向上的棋盘位置数是固定的，因此在行、列方面没有什么信息可以利用。但在不同的位置，在对角线方向所影响的棋盘位置数则是不同的，可以想象，如果当把一个皇后放在棋盘的某个位置后，它所影响的棋盘位置数少，那么给以后放皇后留有的余地就大，找到解的可能行也大；反之留有的余地就小，找到解的可能行也小。如下图所示，（a）图皇后所影响的最长对角线是 3，而（b）图皇后所影响的最长对角线是 2，显然在（b）图位置放置皇后找到解的可能性大于（a）图位置。利用这样的信息对可应用规则集进行动态排序，可以加快找到解的速度。图 1.4 给出采用这种方法后四皇后问题的搜索树，比较图 1.3 和图 1.4，可以说明这种方法对于加快找到解的速度是很有效的，只回溯了 2 次就找到了问题的解。



（a）



（b）

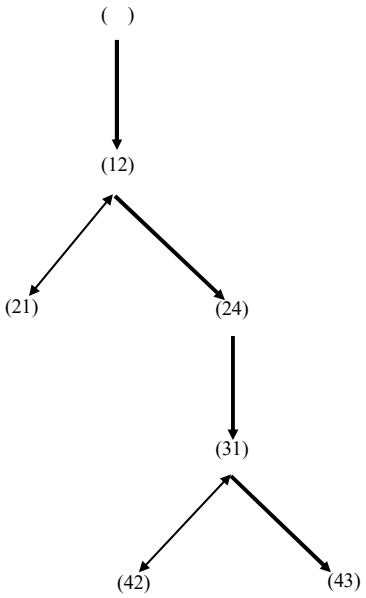
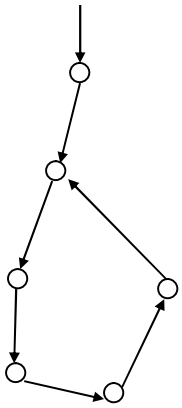


图 1.4 动态排序搜索树

3．递归过程 DEPTH-FIRST-SEARCH1(DATALIST，BOUND)

在前面的深度优先算法 DEPTH-FIRST-SEARCH 中，设置了两个回溯点，一个是当遇到非法状态时回溯，一个是当试探了一个状态的所有子状态后，仍然找不到解时回溯。对于某些问题，可能会遇到这样的问题：一个是问题的某一个（或者某些）分支具有无穷个状态，算法可能会落入某一个“深渊”，永远也回溯不回来，这样，就不能找到问题的解。另一个问题是，在某一个分支上具有环路，搜索会在这个环路中一直进行下去，同样也回溯不出来，从而找不到问题的解。如下图所示。



为了解决这两个问题，下面将给出深度优先搜索算法 DEPTH-FIRST-SEARCH1，该算法比前面的深度优先搜索算法又增加了两个回溯点：一个是用一个参数 BOUND 来限制算法所能够搜索的最大深度，当当前状态的深度达到了限制深度时，算法将强制进行回溯，从而可以避免落入“深渊”；另一个是将过程的参数用从初始状态到当前状态的表来替代原来的当前状态，当新的状态产生时，查看是否已经在该表中出现过了，如果出现过，则表明有环路存在，算法也将进行回溯，从而解决了环路问题。

改进后的算法如下：

递归过程 DEPTH-FIRST-SEARCH1(DATALIST，BOUND)；DATALIST 为从初始状态到当前状态的逆序表，即表的第一个元素为当前状态，最后一个状态为初始状态。BOUND 为给定的最大深度限制。

⑵ DATA：＝FIRST(DATALIST)；设置 DATA 为当前状态

⑵IF MEMBER(DATA，TAIL(DATALIST))，RETURN FAIL；TAIL 是取尾操作，表示取表 DATALIST 中除了第一个元素以外的所有元素。如果 DATA 在 TAIL(DATALIST)中存在，则表示有环路出现，过程返回 FAIL，必须回到前一个状态。

⑶IF TERM(DATA)，RETURN NIL；TERM 取真即找到了目标，则过程返回空表 NIL。也就是说，如果当前状态就是目标，则不需再搜索就结束。

⑷IF DEADEND(DATA)，RETURN FAIL；DEADEND 取真，表示该状态不合法，则过程返回 FAIL，必须回到前一个状态。

⑸IF LENGTH(DATALIST)＞BOUND，RETURN FAIL；LENGTH 计算 DATALIST 的长度，即搜索的深度，当搜索深度大于给定值 BOUND 时，则过程返回 FAIL，必须回到前一个状态。

⑹RULES：＝APPRULES(DATA)；APPRULES 计算 DATA 的可应用规则集，依某种原则（任意排列或按启发式准则排列）排列后赋给 RULES。

⑺LOOP：IF NULL(RULES)，RETURN FAIL；NULL 取真，即规则用完仍未找到目标，过程返回 FAIL，必须回到前一个状态。

⑻R：＝FIRST(RULES)；取第一个可应用规则。

⑼RULES：＝TAIL(RULES)；从可应用规则表中删去第一个规则。

⑽RDATA：＝GEN(R，DATA)；将规则 R 作用于当前状态 DATA，生成 DATA 的下一个状态 RDATA。

⑾RDATALIST：＝CONS(RDATA，DATALIST)；将下一个状态 RDATA 从前插入到表 DATALIST 中。

⑿PATH：＝DEPTH-FIRST-SEARCH1(RDATALIST，BOUND)；递归调用本过程。

⒀IF PATH＝FAIL，GO LO0P；当 PATH＝FAIL 时，表示递归调用未能找到从 RDATA 到达目标的路径，则转移尝试调用 DATA 的下一条规则进行新的尝试。

⒁RETURN CONS(R，PATH)；到达这一步说明递归调用找到了从 RDATA 到目标的路径 PATH。这样将从 DATA 产生 RDATA 的规则 R 从前插入加入到 PATH，就得到了从 DATA 到达目标的路径。过程返回表示该解路径的规则表（或局部解路径子表）。

这个算法与前一个算法的差别是递归过程自变量是状态的链表。在过程 DEPTH-FIRST-SEARCH1 中，形参 DATALIST 是从初始状态到当前状态的逆序表，即初始状态排在表的最后，而当前状态排在表的最前面。返回值同前面的算法一样，是以规则序列表示的路径表（当求解成功时），或者是 FAIL（当求解失败时）。此外在第 2、5 步增设了两个回溯点以检验是否重访已出现过的状态和限定搜索深度范围，这分别由谓词 MEMBER 和＞，函数 LENGTH，参数 BOUND 实现。

另外，上述两个算法得到的解路径均是以规则表表示的，也可以修改为用状态表表示路径，只需对算法进行简单修改就可完成，这里不再赘述。

1.2　图搜索策略

前面讲的深度优先搜索策略的一个特点就是只保留了从初始状态到当前状态的一条路径（无论是 DEPTH-FIRST-SEARCH 还是 DEPTH-FIRST-SEARCH1 都是如此），当回溯出现时，回溯点处进行的搜索将被算法“忘记”，其好处是节省了存储空间，而不足是这些被回溯掉的已经搜索过的部分，不能被以后使用。与此相对应的，将所

有搜索过的状态都记录下来的搜索方法称为“图搜索”，搜索过的路径除了可以重复利用外，其最大的优点是可以更有效地利用与问题有关的一些知识，从而达到启发式搜索的目的。

实际上图搜索策略是实现从一个隐含图中，生成出一部分确实含有一个目标节点的显式表示子图的搜索过程。图搜索策略在人工智能系统中广泛被使用，本节将对算法及涉及的基本理论问题进行讨论。

首先回顾一下图论中几个术语的含义。

节点深度：根节点的深度为 0，其他节点的深度规定为父节点深度加 1，即 $d_{n+1}=d_n+1$。

路径：设一节点序列为$(n_0，n_1，\cdots，n_i，\cdots，n_k)$，对 i＝1，2，…，k，若任一节点 n_{i-1} 都具有一个后继节点 n_i，则该节点序列称为从节点 n_0 到节点 n_k 长度为 k 的一条路径。

路径耗散值：令 $C(n_i，n_j)$为节点 n_i 到 n_j 这段路径（或弧线）的耗散值，一条路径的耗散值等于连接这条路径各节点间所有弧线耗散值的总和。路径耗散值可按如下递归公式计算：

$C(n_i，t)=C(n_i，n_j)+C(n_j，t)$

$C(n_j，t)$为 $n_i \rightarrow t$ 这条路径的耗散值。

耗散值是一个抽象的概念，反应的是一个路径的代价。可以是一段路程的长度，也可以是走完该路程所需要花费的时间，或者花费的金额等。

扩展一个节点：后继节点操作符（相当于可应用规则）作用到节点（对应于某一状态描述）上，生成出其所有后继节点（新状态），并给出连接弧线的耗散值（相当于使用规则的代价），这个过程叫做扩展一个节点。扩展节点可使定义的隐含图生成为显式表示的状态空间图。

下面首先给出一般的图搜索算法框架，该框架的一些内容具体化后，就可以得到不同的图搜索算法。

一般图搜索算法框架

①G:＝G_0(G_0＝s)，OPEN:＝(s)；G 表示图，s 为初始节点，设置 OPEN 表，最初只含初始节点。

②CLOSED:＝(　)；设置 CLOSED 表，起始设置为空表。

③LOOP: IF OPEN＝(　)，THEN　EXIT(FAIL)；

④n:＝FIRST(OPEN)，REMOVE(n，OPEN)，ADD(n，CLOSED)；称 n 为当前节点。

⑤IF GOAL(n)，THEN EXIT(SUCCESS)；由 n 返回到 s 路径上的指针，可给出解

路径。

⑥EXPAND(n)→{m_i}，G:＝ADD(m_i，G)；子节点集{m_i}中不包含 n 的父辈节点。

⑦标记和修改指针

• ADD(m_j，OPEN)，并标记 m_j 连接到 n 的指针；m_j 为 OPEN 和 CLOSED 中未出现过的子节点，{m_i}＝{m_j}∪{m_k}∪{m_l}。

• 计算是否要修改 m_k，m_l 到 n 的指针；m_k 为已出现在 OPEN 中的子节点，m_l 为已出现在 CLOSED 中的子节点。

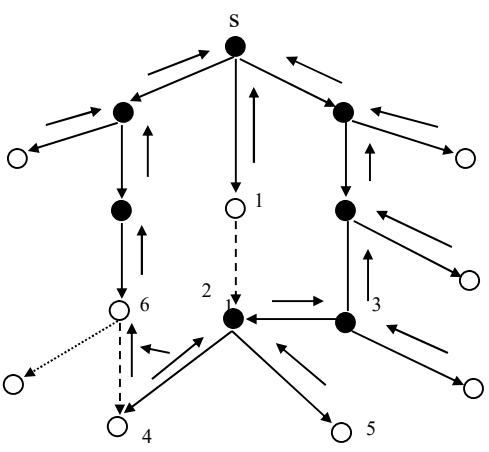
• 计算是否要修改 m_l 到其后继节点的指针；

⑧对 OPEN 中的节点按某种原则重新排序；

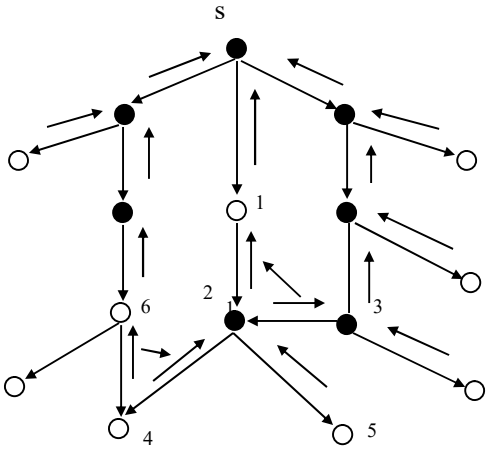
⑨GO LOOP；

这是一个一般的图搜索过程，通过不断的循环，过程便生成出一个显式表示的图 G（搜索图）和一个 G 的子集 T（搜索树）。该搜索树 T 是由第 7 步中标记的指针决定，除根节点 s 外，G 中每个节点只有一个指针指向 G 中的一个父节点，显然树中的每一个节点都处在 G 中。由于图 G 是无环的，因此可根据树定义任一条特殊的路径。可以看出，OPEN 表上的节点，都是搜索树的端节点，即那些已经生成出来但至今尚未被选作为扩展的节点；而 CLOSED 表上的节点，可以是已被扩展而不能生成后继节点的那些端节点，也可以是树中的非端节点。

以上的图搜索过程，在第 8 步要对 OPEN 表上的节点按照某种原则进行排序，以便在第 4 步能选出一个“最好”的节点优先扩展。不同的排序方法便可构成形式多样的专门搜索算法，这在后面还要进一步讨论。如果选出待扩展的节点是目标节点，则算法在第 5 步成功结束，并可通过一步步追踪到 s 的指针给出解路径。如果在该过程的某个循环中，搜索树不再剩有待选的节点，即 OPEN 表变空时，则过程失败结束，问题找不到解。



（a）



（b）

图 1.5 扩展节点 6 和节点 1 得到的搜索图

现在再说明一下第 7 步中标记和修改指针的问题。如果要搜索的隐含图是一棵树，则可肯定第 6 步生成的后继节点不可能是以前生成过的，这时搜索图就是搜索树，不存在 m_k、m_1 这种类型的子节点，因此不必进行修改指针的操作。如果要搜索的隐含图不是一棵树，则有可能出现 m_k 这样的子节点，就是说这时又发现了到达 m_k 的新通路，这样就要比较不同路径的耗散值，把指针修改到具有较小耗散值的路径上。例如，图 1.5 所示的两个搜索图中，实心圆点在 CLOSED 表中（已扩展过的节点），空心圆点则在 OPEN 表中（待扩展点）。先设下一步轮到要扩展节点 6，并设生成的两个子节点，其中有一个节点 4 已在 OPEN 中，那么原先路径(s→3→2→4)耗散值为 5（设每段路径均为单位耗散，即相邻两个节点间的耗散值为 1），新路径(s→6→4)的耗散值为 4，所以节点 4 的指针应由指向节点 2 修正到指向节点 6，如图 1.5(a)所示。接着设下一循环要扩展节点 1，若节点 1 只生成一个子节点 2（它已在 CLOSED 中），显然这时节点 2 原先指向节点 3 的指针，要修改到指向节点 1，由此又引起子节点 3 的指针又修改为指向节点 2，如图 1.5(b)所示。

1.3 无信息图搜索过程

无信息图搜索过程是在算法的第 8 步中使用任意排列 OPEN 表节点的顺序，通常有两种排列方式，可以分别构成基于图的深度优先搜索（为了与上一节介绍的基于递归的深度优先搜索算法相区别，我们称其为图深度优先搜索算法，并用 DEPTH－FIRST－SEARCH－GS 表示算法过程）和宽度优先搜索。

1．图深度优先搜索

所谓图深度优选搜索，就是每次循环中，优选扩展 OPEN 表中深度最深的节点。下面先给出图深度优先算法，后面再给出对算法的说明。

过程 DEPTH－FIRST－SEARCH－GS

①G:=G_0(G_0=s)，OPEN:=(s)，CLOSED:=(　)；

②LOOP：IF OPEN＝(　)THEN EXIT(FAIL)；

③n:＝FIRST(OPEN)；

④IF GOAL(n) THEN EXIT(SUCCESS)；

⑤REMOVE(n，OPEN)，ADD(n，CLOSED)；

⑥EXPAND(n)→{m_i}；

G:＝ADD(m_i，G)；

⑦ADD(m_j，OPEN)，并标记 m_j 到 n 的指针；把不在 OPEN 或 CLOSED 中的节点放在 OPEN 表的最前面，使深度大的节点可优先扩展。

⑧GO LOOP；

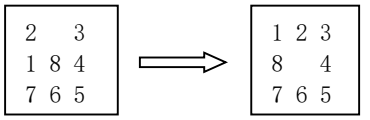
该算法是根据一般的图搜索算法改变而成的。所谓图深度优先搜索，就是在每次扩展一个节点时，选择到目前为止深度最深的节点优先扩展。这一点是在算法的第 7 步体现出来的。第 7 步中的 ADD(m_j，OPEN)表示将被扩展节点 n 的所有新子节点 m_j 加到 OPEN 表的前面。开始时，OPEN 表中只有一个初始节点 s，s 被扩展，其子节点被放入 OPEN 表中。在算法的第 3 步，OPEN 表的第一个元素——设为 n——被取出扩展，这时节点 n 的深度在 OPEN 表中是最大的，OPEN 表中的其他节点的深度都不会超过 n 的深度。n 的子节点被放到 OPEN 表的最前面。由于子节点的深度要大于父节点的深度，实际上 OPEN 表是按照节点的深度进行排序的，深度深的节点被排在了前面，而深度浅的节点被放在了后面。这样当下一个循环再次取出 OPEN 表的第一个元素时，实际上选择的就是到目前为止深度最深的节点，从而实现了深度优先的搜索策略。

一般情况下，当问题有解时，图深度优先搜索不但不能保证找到最优解，也不能保证一定能找到解。如果问题的状态空间是有限的，则可以保证找到解，当问题的状态空间是无限的时，则可能陷入“深渊”，而找不到解。为此，像上一节讲过的基于递归算法的深度搜索策略一样，可以加上对搜索的深度限制。其方法是在算法的第 7 步，当节点的深度达到限制深度时，则不将其子节点加入到 OPEN 表中，从而实现对搜索深度的限制。当然，这个深度限制应该设置的合适，深度过深影响搜索的效率，而深度过浅，则可能影响找到问题的解。

下面以八数码问题为例，说明图深度优先搜索是如何求解问题的。

八数码问题（Eight－Puzzle）描述如下：

在 3×3 组成的九宫格棋盘上，摆有八个将牌，每一个将牌都刻有 1—8 数码中的某一个数码。棋盘中留有一个空格，允许其周围的某一个将牌向空格移动，这样通过移动将牌就可以不断改变将牌的布局。这种游戏求解的问题是：给定一种初始的将牌布局或结构（称初始状态）和一个目标的布局（称目标状态），问如何移动将牌，实现从初始状态到目标状态的转变。问题的解答其实就是给出一个合法的走步序列。



初始状态　　目标状态

图 八数码问题

图 x 给出了用深度优先方法求解八数码问题的搜索树，其中深度限制设置为 4。图中圆圈中的序号表示节点的扩展顺序。除了初始节点外，每个节点用箭头指向其父节点，当搜索到目标节点后，沿着箭头所指反向追踪到初始节点，即可得到问题的解答。

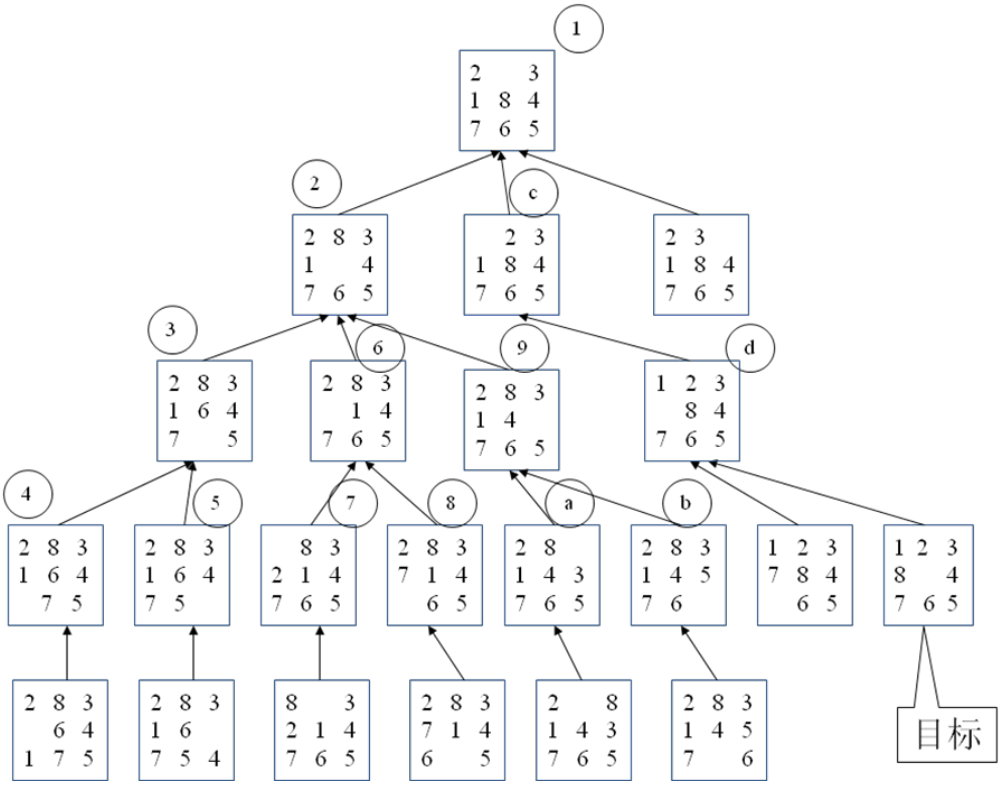


图 x 用深度优先搜索求解八数码问题

图深度优先搜索算法和基于递归的深度优先算法有什么区别呢（为了方便，下面的说明中，深度优先算法表示基于递归的深度优先算法）？

前面已经说过，深度优先搜索算法在搜索过程中只保留初始节点到当前节点的一条路径，比较节省内存。但是由于回溯的存在，一些已经搜索过测路径会被抛弃掉，不能再被利用。比如，如果算法中深度限制设置的过浅，找不到解，需要再增加搜索深度时，就需要全部从头再来，重新搜索。另外如果问题表示为一个图，到某些节点存在多个路径，而这些节点一下的部分可能会造成多次重复搜索，影响了搜索效率。而图深度优先搜索算法，由于保留了所有的已经搜索过测路径，则不会存在这样的问题。当然其不足是占用空间比较多，需要比较大的存储空间。二者各有优缺点，可以

根据实际情况选择使用哪种算法。

2．宽度优先

同图深度优先算法一样，宽度优先算法也是从一般的图搜索算法变化而成。在图深度优先搜索中，每次选择深度最深的节点首先扩展，而宽度优先搜索则正好相反，每次选择深度最浅的节点优先扩展。与图深度优先算法不同的只是第 7 步，这里 ADD(OPEN，m_j)表示将 m_j 类子节点放到 OPEN 表的后边，从而实现了对 OPEN 表中的元素按节点深度排序，只不过这次将深度浅的节点放在 OPEN 表的前面了，而深度深的节点被放在了 OPEN 表的后边。当问题有解时，宽度优先算法一定能找到解，并且在单位耗散的情况下，可以保证找到最优解。

过程 BREADTH－FIRST－SEARCH

①G:=G_0(G_0=s)，OPEN:=(s)，CLOSED:=(　)；

②LOOP: IF OPEN＝(　) THEN EXIT(FAIL)；

③n:＝FIRST(OPEN)；

④IF GOAL(n) THEN EXIT(SUCCESS)；

⑤REMOVE(n，OPEN)，ADD(n，CLOSED)；

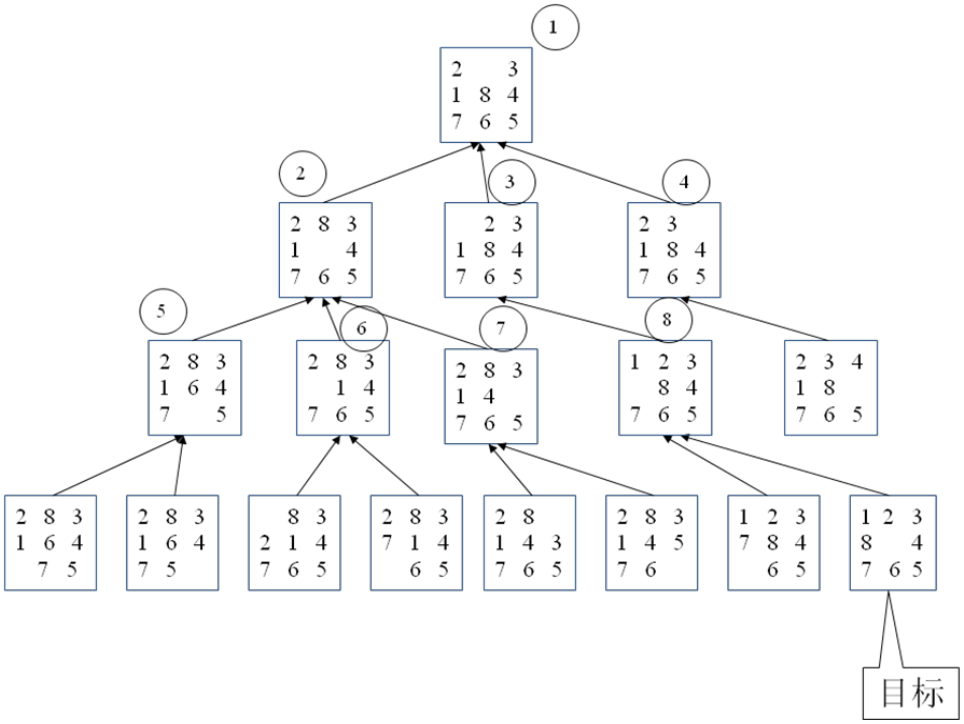
⑥EXPAND(n)→{m_i}，

　G:＝ADD(m_i，G)；

⑦ADD(OPEN，m_j)，并标记 m_j 到 n 的指针；把不在 OPEN 或 CLOSED 中的节点放在 OPEN 表的后面，使深度浅的节点可优先扩展。

⑧GO LOOP；

对于前面的八数码问题，同样可以用宽度优先进行求解，搜索树如图 x 所示。



3．迭代加深搜索策略

宽度优先搜索具有的优势是，在单位耗散值的情况下，当问题有解时必能找到最优解。其不足是产生的节点数随目标所在的深度呈几何增长，当最优解路径比较长时，会消耗大量的内存空间，以至于因占用空间过大而不能求解。（基于递归的）深度优先搜索策略虽然不一定找到最优解，且生成的节点数并不少，但由于其仅保留从初始状态到当前状态的一条路径，在设置了最大搜索深度限制 BOUND 的情况下，需要保留的节点数最多仅为 BOUND+1 个，具有占用空间小的优势。能否把深度优先搜索策略与宽度优先的思想结合起来，在小空间下实现最优路径的求解呢？

我们首先分析一下为什么宽度优先可以找到最优解。

宽度优先搜索，每次优先扩展深度浅的节点，在扩展了初始节点之后，首先扩展深度为 1 的节点，然后再扩展深度为 2 的节点，扩展的节点深度一层层逐渐加深，直到找到目标节点为止。在搜索树上体现的是一种“横着走”的形式，逐步加深搜索深度。如果问题是单位耗散值的话，其节点深度就是从初始节点到该节点路径的耗散值，

所以一旦扩展出了目标节点，就找到了从初始节点到目标节点的最优路径，即耗散值最小的路径。

在 1.1 节给出的深度优先搜索过程 DEPTH－FIRST－SEARCH1 中，有一个控制搜索深度的参数 BOUND，在搜索深度达到该值时将强制进行回溯。如果目标节点的深度大于 BOUND 的话，深度优先搜索过程将搜索到所有深度小于等于 BOUND 的节点，而不会搜索任何深度大于 BOUND 的节点。如果我们设置 BOUND 的初始值为 1，然后逐渐加大 BOUND 值，循环调用 DEPTH－FIRST－SEARCH1 过程，直到 BOUND 值等于目标节点的深度时，搜索到目标结束。这样的搜索效果将等同于宽度优先搜索，可以找到问题的最优解，并且具有深度优先搜索策略占用空间小的性质，仅使用目标节点深度量级的存储空间。

我们把这种搜索策略称作为迭代加深搜索算法（ITERATIVE-DEEPENING-DEPTH-FIRST），下面给出该算法的描述。

过程 ITERATIVE-DEEPENING-DEPTH-FIRST(DATA0)；DATA0 为初始状态

① BOUND := 1；
② PATH := FAIL;
③ WHILE (PATH != FAIL)
④ DATALIST := LIST(DATA0)；用初始状态组成一个表
⑤ PATH := DEPTH-FIRST-SEARCH1(DATALIST，BOUND)；调用深度优先过程，如果 PATH 等于 FAIL 表示没有搜索到目标，否则 PATH 得到一条从初始节点到目标节点的路径
⑥ BOUND := BOUND + 1；加大一层搜索深度
⑦ END WHILE
⑧ RETURN PATH；返回解路径

从以上算法可以看出，如果最佳解的深度为 d 时，算法要调用 d 次 DEPTH-FIRST-SEARCH1，会不会使得算法的求解时间大幅度增加呢？下面就分析一下宽度优先算法和迭代加深算法之间的时间关系。

为了分析上的方便，我们假设在一个满 b 叉树上搜索，最佳解的深度为 d。有理由认为算法的时间与所产生的节点数成正比，这样，我们只需分析两个算法所生成的节点数的关系即可。

设宽度优先产生的节点数记为 N_{BF}，则：

$$N_{BF} = 1 + b + b^2 + \cdots + b^d$$

$$= (b^{d+1} - 1)/(b - 1)$$

设迭代加深搜索策略产生的节点数记为 N_{IDBT}，则[1]：

$$N_{IDBT} = (d+1) + db + (d-1)b^2 + \cdots + 2b^{d-1} + b^d$$

$$= \frac{b(b^{d+1} - 1)}{(b-1)^2} - \frac{d+1}{b-1}$$

$$= \frac{b}{b-1}\frac{b^{d+1} - 1}{b-1} - \frac{d+1}{b-1}$$

$$= \frac{b}{b-1}N_{BF} - \frac{d+1}{b-1}$$

由于分叉数 b ≥ 2，所以有：

$$N_{IDBT} < \frac{b}{b-1}N_{BF}$$

如果假设搜索所用的时间与产生的节点数成正比，则迭代加深搜索策略所用时间不大于宽度优先搜索所用时间的 b/(b-1)倍，分支数越大，二者所用时间越接近。下表给出了不同 b 值时所用时间比 k，当 b=2 时 k 值最大为 2。也就是说，迭代加深搜索策略所用时间不会超过宽度优先搜索所用时间的 2 倍。

b	2	3	4	5	6	7	8	9	10
时间倍数 k	2	1.5	1.33	1.25	1.2	1.17	1.14	1.13	1.11

迭代加深搜索策略除了为了节省空间外，还可以应用到一些在规定的时间内，尽可能做出最优选择的场合。比如计算机下棋程序，一般来说搜索的深度越深，性能越好，但是一般下棋每一步会有时间限制，不能超时。这时就可以采用迭代加深搜索的思想，逐步加深搜索范围，在限时快结束时，用当前得到的最好结果作为决策。否则如果开始设置的搜索深度过深，可能限时结束时还得不到结果。反之如果设置的搜索深度过浅，则没有充分利用时间。

[1]在下面的公式推导中，省略了从第一行推导出第二行的过程。其基本思想是：$b*N_{IDBT}-N_{IDBT}$ 可以得到一个等比数列，然后通过求和公式求解出 N_{IDBT} 的计算公式。

1.4　启发式图搜索过程

启发式搜索是利用问题拥有的启发信息来引导搜索，达到减少搜索范围，降低问题复杂度的目的。这种利用启发信息的搜索过程称为启发式搜索方法。在研究启发式搜索方法时，先说明一下启发信息应用、启发能力度量及如何获得启发信息这几个问题，然后再来讨论算法及一些理论问题。

一般来说启发信息过弱，搜索算法在找到一条路径之前将扩展过多的节点，即求得解路径所需搜索的耗散值（搜索花费的工作量）较大；相反引入强的启发信息，有可能大大降低搜索工作量，但不能保证找到最小耗散值的解路径（最佳路径），因此实际应用中，希望最好能引入降低搜索工作量的启发信息而不牺牲找到最佳路径的保证。这是一个重要而又困难的问题，从理论上要研究启发信息和最佳路径的关系，从实际上则要解决获取启发信息方法的问题。

比较不同搜索方法的效果可用启发能力的强弱来度量。在大多数实际问题中，人们感兴趣的是使路径的耗散值和求得路径所需搜索的耗散值两者的某种组合最小，更一般的情况是考虑搜索方法对求解所有可能遇见的问题，其平均的组合耗散值最小。如果搜索方法 1 的平均组合耗散值比方法 2 的平均组合耗散值低，则认为方法 1 比方法 2 有更强的启发能力。实际上很难给出一个计算平均组合耗散值的方法，因此启发能力的度量也只能根据使用方法的实际经验作出判断，没有必要去追求严格的比较结果。

启发式搜索过程中，要对 OPEN 表进行排序，这就需要有一种方法来计算待扩展节点有希望通向目标节点的不同程度，我们总是希望能找到最有希望通向目标节点的待扩展节点优先扩展。一种最常用的方法是定义一个评价函数 f（Evaluation function）对各个子节点进行计算，其目的就是用来估算出“有希望”的节点来。如何定义一个评价函数呢？通常可以参考的原则有：一个节点处在最佳路径上的概率；求出任意一个节点与目标节点集之间的距离度量或差异度量；根据格局（博弈问题）或状态的特点来打分。即根据问题的启发信息，从概率角度、差异角度或记分法给出计算评价函数的方法。

1．启发式搜索算法 A

启发式搜索算法 A，一般简称为 A 算法，是一种典型的启发式搜索算法。其基本思想是：定义一个评价函数 f，对当前的搜索状态进行评估，找出一个最有希望的节点来扩展。

评价函数的形式如下：

f(n)＝g(n)＋h(n)

其中 n 是被评价的节点。

f(n)、g(n)和 h(n)各自表述什么含义呢？我们先来定义下面几个函数的含义，它们与 f(n)、g(n)和 h(n)的差别是都带有一个“*”号。

g*(n)：表示从初始节点 s 到节点 n 的最短路径的耗散值；

h*(n)：表示从节点 n 到目标节点 g 的最短路径的耗散值；

f*(n)=g*(n)+h*(n)：表示从初始节点 s 经过节点 n 到目标节点 g 的最短路径的耗散值。

而 f(n)、g(n)和 h(n)则分别表示是对 f*(n)、g*(n)和 h*(n)三个函数值的的估计值。是一种预测。A 算法就是利用这种预测，来达到有效搜索的目的的。它每次按照 f(n)值的大小对 OPEN 表中的元素进行排序，f 值小的节点放在前面，而 f 值大的节点则被放在 OPEN 表的后面，这样每次扩展节点时，总是选择当前 f 值最小的节点来优先扩展。

过程 A

①OPEN:＝(s)，f(s):＝g(s)+h(s);

②LOOP: IF OPEN＝(　) THEN EXIT(FAIL)；

③n:＝FIRST(OPEN)；

④IF GOAL(n) THEN EXIT(SUCCESS)；

⑤REMOVE(n，OPEN)，ADD(n，CLOSED)；

⑥EXPAND(n)→{m_i}，计算 f(n，m_i)＝g(n，m_i)+h(m_i)；g(n，m_i)是从 s 通过 n 到 m_i 的耗散值，f(n，m_i)是从 s 通过 n、m_i 到目标节点耗散值的估计。

• ADD(m_j，OPEN)，标记 m_i 到 n 的指针。

• IF f(n，m_k)<f(m_k) THEN f(m_k):＝f(n，m_k)，标记 m_k 到 n 的指针；比较 f(n，m_k)和 f(m_k)，f(m_k)是扩展 n 之前计算的耗散值。

•IF f(n，m_l)<f(m_l) THEN f(m_l):＝f(n，m_l)，标记 m_l 到 n 的指针，ADD(m_l，OPEN)；当 f(n，m_l)<f(m_l)时，把 m_l 重放回 OPEN 中，不必考虑修改到其子节点的指针。

⑦OPEN 中的节点按 f 值从小到大排序；

⑧GO　LOOP；

A 算法同样由一般的图搜索算法改变而成。在算法的第 7 步，按照 f 值从小到大对 OPEN 表中的节点进行排序，体现了 A 算法的含义。

算法要计算 f(n)、g(n)和 h(n)的值，g(n)根据已经搜索的结果，按照从初始节点 s 到节点 n 的路径，计算这条路径的耗散值就可以了。而 h(n)则依赖于启发信息，是与问题有关的，需要根据具体的问题来定义。通常称 h(n)为启发函数，是对未来扩展的

方向作出估计。

算法 A 是按 f(n)递增的顺序来排列 OPEN 表的节点，因而优先扩展 f(n)值小的节点，体现了好的优先搜索思想，所以算法 A 是一个好的优先的搜索策略。图 1.6 表示出当前要扩展节点 n 之前的搜索图，扩展 n 后新生成的子节点 $m_1(\in\{m_j\})$、$m_2(\in\{m_k\})$、$m_3(\in\{m_l\})$要分别计算其评价函数值：

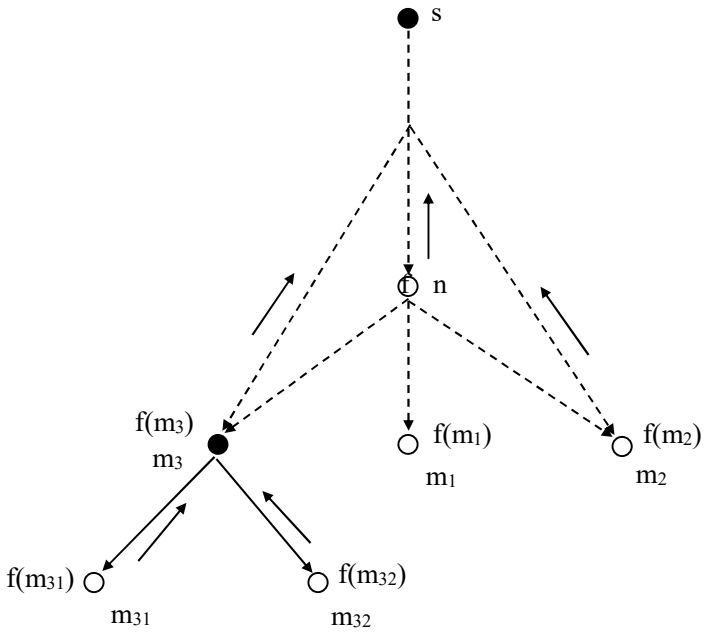


图 1.6 搜索示意图

$f(m_1)=g(m_1)+h(m_1)$

$f(n，m_2)=g(n，m_2)+h(m_2)$

$f(n，m_3)=g(n，m_3)+h(m_3)$

然后按第 6 步条件进行指针设置和第 7 步重排 OPEN 表节点顺序，以便确定下一次要扩展的节点。

下面再以八数码问题为例说明 A 算法的搜索过程。

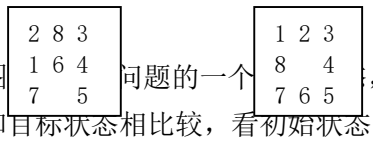
设评价函数 f(n)形式如下：

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

其中 d(n)代表节点的深度，取 g(n)=d(n)表示讨论单位耗散的情况；取 h(n)=W(n)表示以“不在位”的将牌个数作为启发函数的度量，这时 f(n)可估计出通向目标节点的希望程度。

“不在位的将牌数”计算方法如下：

我们来看下面的两个图。



其中左边的图是8数码问题的一个状态，右边的图是8数码问题的目标状态。我们拿初始状态和目标状态相比较，看初始状态的哪些将牌不在目标状态的位置上，这些将牌的数目之和，就是“不在位的将牌数”。比较上面两个图，发现 1、2、6 和 8 四个将牌不在目标状态的位置上，所以初始状态的“不在位的将牌数”就是 4，也就是初始状态的 h 值。其他状态的 h 值，也按照此方法计算。

图 1.7 表示使用这种评价函数时的搜索树，图中括弧中的数字表示该节点的评价函数值 f。算法每一循环结束时，其 OPEN 表和 CLOSED 表的排列如下：

OPEN 表	CLODED 表
初始化 (s(4))	( )
第一循环结束 (B(4) A(6) C(6))	(s(4))
第二循环结束 (D(5) E(5) A(6) C(6) F(6))	(s(4) B(4))
第三循环结束 (E(5) A(6) C(6) F(6) G(6) H(7))	(s(4) B(4) D(5))
第四循环结束 (I(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5))
第五循环结束 (K(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5) I(5))
第六循环结束 (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))
第七循环结束 第四步成功退出	

根据目标节点 L 返回到 s 的指针，可得解路径 S(4)，B(5)，E(5)，I(5)，K(5)，L(5)

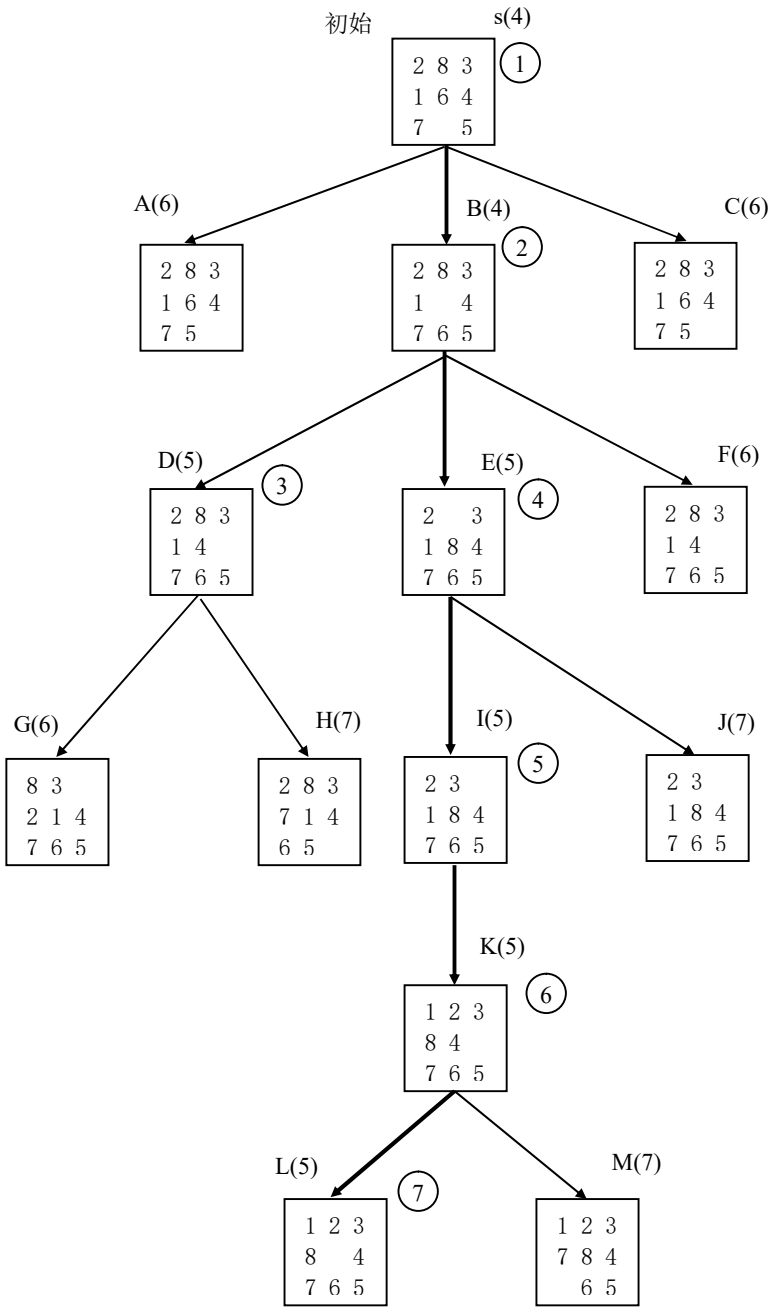

初始
s(4)
2 8 3
1 6 4
7 5
1
A(6)
B(4)
C(6)
2 8 3
1 6 4
7 5
2 8 3
1 4
7 6 5
2
2 8 3
1 6 4
7 5
D(5)
E(5)
F(6)
2 8 3
1 4
7 6 5
3
2 3
1 8 4
7 6 5
4
2 8 3
1 4
7 6 5
G(6)
H(7)
I(5)
J(7)
8 3
2 1 4
7 6 5
2 8 3
7 1 4
6 5
2 3
1 8 4
7 6 5
5
2 3
1 8 4
7 6 5
K(5)
1 2 3
8 4
7 6 5
6
L(5)
M(7)
1 2 3
8 4
7 6 5
7
1 2 3
7 8 4
6 5

图 1.7 八数码问题的搜索树

2．爬山法

过程 Hill-climbing

①n:=s；s 为初始节点

②LOOP：IF GOAL(n) THEN EXIT(SUCCESS)；

③EXPAND(n)→{m_i}，计算 h(m_i)，nextn:=m(min h(m_i)的节点)；

④IF h(n)<h(nextn) THEN EXIT(FAIL);

⑤n:=nextn；

⑥GO LOOP；

显然如果将山顶作为目标，h(n)表示山顶与当前位置 n 之间高度之差，则该算法相当于总是登向山顶，在单峰的条件下，必能到达山峰。

3．分支界限法

分支界限法是优先扩展当前具有最小耗散值分支路径的端节点 n，其评价函数为 f(n)=g(n)。该算法的基本思想很简单，实际上是建立一个局部路径（或分支）的队列表，每次都选耗散值最小的那个分支上的端节点来扩展，直到生成出含有目标节点的路径为止。

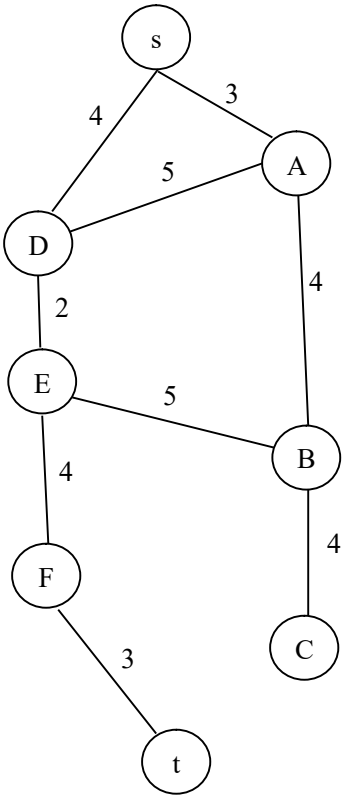


图 1.8 八城市地图示意图

过程 Branch-Bound

①QUEUE:=(s-s)，g(s)=0;

②LOOP：IF QUEUE=(　　)　THEN EXIT(FAIL);

③PATH:=FIRST(QUEUE)，n:=LAST(PATH);

④IF GOAL(n) THEN EXIT(SUCCESS);

⑤EXPAND(n)→{m_i}，计算 g(m_i)=g(n，m_i)，REMOVE(s - n，QUEUE)，ADD(s-m_i，QUEUE);

⑥QUEUE 中局部路径 g 值最小者排在前面；

⑦GO LOOP；

应用该算法求解图 1.8 的最短路径问题，其搜索图如图 1.9 所示，求解过程中 QUEUE 的结果简记如下（D(4)代表耗散值为 4 的 s-D 分支，其余类推）：

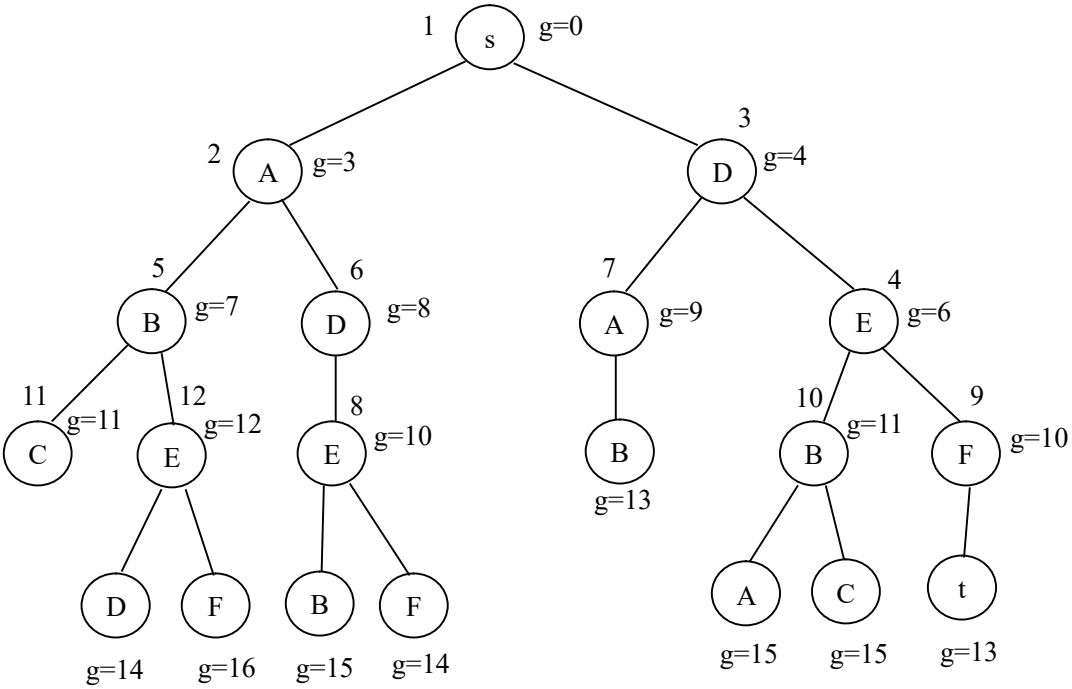


图 1.9 分支界限搜索树

初始 (s(0))

1）(A(3) D(4))

2）(D(4) B(7) D(8))

3）(E(6) B(7) D(8) A(9))

4）(B(7) D(8) A(9) F(10) B(11))

5）(D(8) A(9) F(10) B(11) C(11) E(12))

6）(A(9) E(10) F(10) B(11) C(11) E(12))

7）(E(10) F(10) B(11) C(11) E(12) B(13))

8）(F(10) B(11) C(11) E(12) B(13) F(14) B(15))

9）(B(11) C(11) E(12) t(13) B(13) F(14) B(15))

10）(C(11) E(12) t(13) B(13) F(14) A(15) B(15) C(15))

11）(E(12) t(13) B(13) F(14) A(15) B(15) C(15))

12）(t(13) B(13) F(14) D(14) A(15) B(15) C(15) F(16))

13）结束。

4．动态规划法

在 A 算法中，当 h(n)≡0 时，则 A 算法演变为动态规划算法。由于在 A 算法中，很多问题的启发函数 h 难于定义，因此动态规划算法仍然是至今一直被经常使用的算法。在其他方面的一些书中——如运筹学方面的书——讲到的动态规划方法，在形式上可能与这里介绍的有所不同，但其性质是一样的，而且这里所介绍的动态规划方法，具有更多的灵活性。

动态规划法实际上是对分支界限法的改进。从图 1.9 看出，第二循环扩展 A(3)后生成的 D(8)节点（D(4)已在 QUEUE 上）和第三循环扩展 D(4)之后生成的 A(9)节点（A(3)已以 QUEUE 上）都是多余的分支，因为由 s－D 到达目标的路径显然要比 s－A－D 到达目标的路径要好。因此删去类似于 s－A－D 或 s－D－A 这样一些多余的路径将会大大提高搜索效率。动态规划原理指出，求 s→t 的最佳路径时，对某一个中间节点 I，只要考虑 s 到 I 中最小耗散值这一条局部路径就可以，其余 s 到 I 的路径是多余的，不必加以考虑。下面给出具有动态规划原理的分支界限算法。

过程 Dynamic-Programming

①QUEUE:=(s-s)，g(s)=0;

②LOOP: IF QUEUE=(　) THEN EXIT(FAIL);

③PATH:=FIRST(QUEUE)，n:=LAST(PATH);

④IF GOAL(n) THEN EXIT(SUCCESS);

⑤EXPAND(n)→{m_i}，计算 $g(m_i)=g(n, m_i)$，REMOVE(s-n，QUEUE)，ADD(s-m_i，QUEUE);

⑥若 QUEUE 中有多条到达某一公共节点的路径，则只保留耗散值最小的那条路径，其余删去，并重新排序，g 值最小者排在前面；

⑦GO LOOP；

对图 1.8 的例子应用该算法，其搜索图如图 1.10 所示，实际只剩下两条搜索路径，改善了搜索效率。

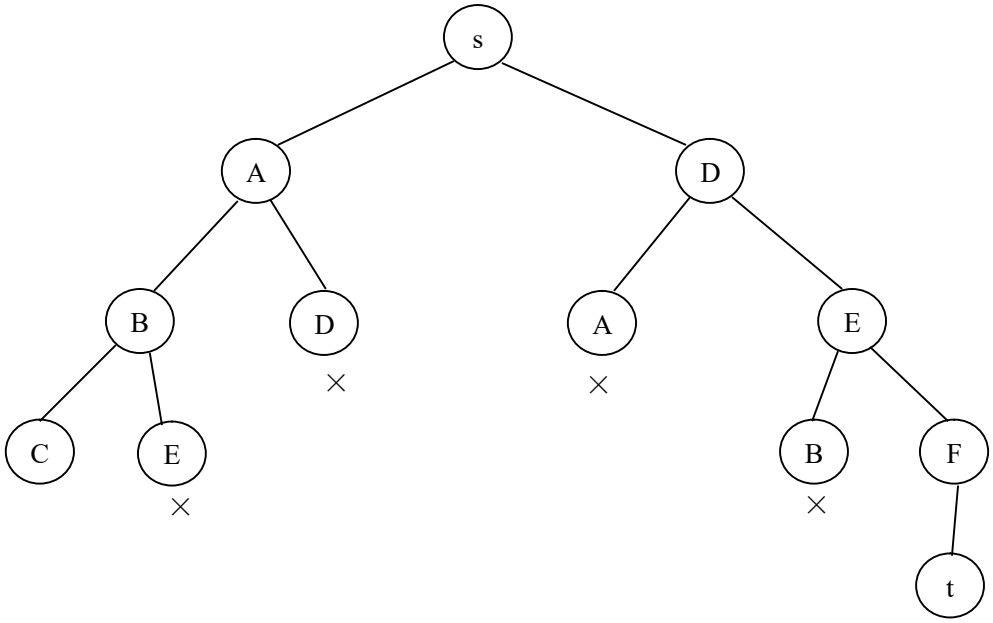


图 1.10 动态规划原理的搜索树

5．最佳图搜索算法 A＊（Optimal Search）

当在算法 A 的评价函数中，使用的启发函数 h(n)是处在 h*(n)的下界范围，即满足 $h(n) \leq h^*(n)$时，则我们把这个算法称为算法 A*。A*算法实际上是分支界限和动态规划原理及使用下界范围的 h 相结合的算法。当问题有解时，A*一定能找到一条到达目标节点的最佳路径。例如在极端情况下，若 $h(n)\equiv 0$（肯定满足下界范围条件），因而一定能找到最佳路径。此时若 $g\equiv d$，则算法等同于宽度优先算法。前面已提到过，宽度优先算法能保证找到一条到达目标节点最小长度的路径，因而这个特例从直观上就验证了 A*的一般结论。

一般地说对任意一个图，当 s 到目标节点有一条路径存在时，如果搜索算法总是在找到一条从 s 到目标节点的最佳路径上结束，则称该搜索算法是可采纳的

（Admissibility）。A*就具有可采纳性。

下面来证明 A * 的可采纳性及若干重要性质。

定理 1：对有限图，如果从初始节点 s 到目标节点 t 有路径存在，则算法 A 一定成功结束。

证明：设 A 搜索失败，则算法在第 2 步结束，OPEN 表变空，而 CLOSED 表中的节点是在结束之前被扩展过的节点。由于图有解，令（n_0=s，n_1，n_2，…，n_k=t）表示某一解路径，我们从 n_k 开始逆向逐个检查该序列的节点，找到出现在 CLOSED 表中的节点 n_i，即 n_i∈CLOSED，n_{i+1}∉CLOSED（n_i 一定能找到，因为 n_0∈DLOSED，n_k∉CLOSED）。由于 n_i 在 CLOSES 中，必定在第 6 步被扩展，且 n_{i+1} 被加到 OPEN 中，因此在 OPEN 表空之前，n_{i+1} 已被处理过。若 n_{i+1} 是目标节点，则搜索成功，否则它被加入到 CLOSED 中，这两种情况都与搜索失败的假设矛盾，因此对有限图不失败则成功。[证毕]

因为 A * 是 A 的特例，因此它具有 A 的所有性质。这样对有限图如果有解，则 A * 一定能在找到到达目标的路径结束，下面要证明即使是无限图，A * 也能找到最佳解结束。我们先证两个引理：

引理 2.1：对无限图，若有从初始节点 s 到目标点 t 的一条路径，则 A * 不结束时，在 OPEN 中即使最小的一个 f 值也将增到任意大，或有 f(n)>f * (s)。

在如下的证明中，隐含了两个假设：（1）任何两个节点之间的耗散值都大于某个给定的大于零的常量；（2）h(n)对于任何 n 来说，都大于等于零。

证明：设 d * (n)是 A * 生成的搜索树中，从 s 到任一节点 n 最短路径长度的值（设每个弧的长度均为 1），搜索图上每个弧的耗散值为 $C(n_i, n_{i+1})$（C 取正）。令 e=min $C(n_i, n_{i+1})$，则 g * (n)≥d * (n)e。而 g(n)≥g * (n)≥d * (n)e，故有：

f(n)=g(n)+h(n)≥g(n)≥d * (n)e（设 h(n)≥0）

若 A * 不结束，d * (n)ψ∞，f 值将增到任意大。

设 $M = \frac{f^*(s)}{e}$，M 是一个定数，所以搜索进行到一定程度会有 d * (n)>M，或 $\frac{d^*(n)}{M} > 1$，则

$$f(n) \geq d^*(n)e = d^*(n)\frac{f^*(s)}{M} = f^*(s)\frac{d^*(n)}{M} > f^*(s)。$$

[证毕]

引理 2.2：A*结束前，OPEN 表中必存在 f(n)≤f * (s)的节点（n 是在最佳路径上的节点）。

证明：设从初始节点 s 到目标节点 t 的一条最佳路径序列为：

(n_0=s，n_1，…，n_k=t)

算法初始化时，s 在 OPEN 中，由于 A * 没有结束，在 OPEN 中存在最佳路径上的节点。设 OPEN 表中的某节点 n 是处在最佳路径序列中（至少有一个这样的节点，因 s 一开始是在 OPEN 上），显然 n 的先辈节点 n_p 已在 CLOSED 中，因此能找到 s 到 n_p 的最佳路径，而 n 也在最佳路径上，因而 s 到 n 的最佳路径也能找到，因此有

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

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

而最佳路径上任一节点均有 $f^*(n)=f^*(s)$ ($f^*(s)$是最佳路径的耗散值)，所以 $f(n)\leqslant f^*(s)$。[证毕]

定理 2：对无限图，若从初始节点 s 到目标节点 t 有路径存在，则 A*也一定成功结束。

证明：假定 A*不结束，由引理 2.1 有 $f(n)>f^*(s)$，或 OPEN 表中最小的一个 f 值也变成无界，这与引理 2.2 的结论矛盾，所以 A*只能成功结束。[证毕]

推论 2.1：OPEN 表上任一具有 $f(n)<f^*(s)$的节点 n，最终都将被 A*选作为扩展的节点。

定理 3：若存在初始节点 s 到目标节点 t 的路径，则 A*必能找到最佳解结束。

证明：

（1）由定理 1、2 知 A*一定会找到一个目标节点结束。

（2）设找到一个目标节点 t 结束，而 sψt 不是一条最佳路径，即：

$$f(t)=g(t)>f^*(s)$$

而根据引理 2.2 知结束前 OPEN 表上有节点 n，且处在最佳路径上，并有 $f(n)\leqslant f^*(s)$，所以

$$f(n)\leqslant f^*(s)<f(t)$$

这时算法 A*应选 n 作为当前节点扩展，不可能选 t，从而也不会去测试目标节点 t，即这与假定 A*选 t 结束矛盾，所以 A*只能结束在最佳路径上。[证毕]

推论 3.1：A*选作扩展的任一节点 n，有 $f(n)\leqslant f^*(s)$。

证明：令 n 是由 A*选作扩展的任一节点，因此 n 不会是目标节点，且搜索没有结束，由引理 2.2 而知在 OPEN 中有满足 $f(n')\leq f^*(s)$ 的节点 n′。若 n=n′，则 $f(n)\leqslant f^*(s)$，否则选 n 扩展，必有 $f(n)\leq f(n')$，所以 $f(n)\leqslant f^*(s)$成立。[证毕]

6．启发函数与 A*算法的关系

应用 A*的过程中，如果选作扩展的节点 n，其评价函数值 $f(n)=f^*(n)$，则不会去扩展多余的节点就可找到解。可以想象到 f(n)越接近于 $f^*(n)$，扩展的节点数就会越少，即启发函数中，应用的启发信息（问题知识）愈多，扩展的节点数就越少。

定理 4：有两个 A*算法 A_1 和 A_2，若 A_2 比 A_1 有较多的启发信息（即对所有非目标节点均有 $h_2(n)>h_1(n)$），则在具有一条从 s 到 t 的隐含图上，搜索结束时，由 A_2 所扩展的每一个节点，也必定由 A_1 所扩展，即 A_1 扩展的节点至少和 A_2 一样多。

证明：使用数学归纳法，对节点的深度应用归纳法。

（1）对深度 d(n)=0 的节点（即初始节点 s），定理结论成立，即若 s 为目标节点，则 A_1 和 A_2 都不扩展 s，否则 A_1 和 A_2 都扩展了 s（归纳法前提）。

（2）设深度 $d(n)\leqslant=k$，对所有路径的端节点，定理结论都成立（归纳法假设）。

（3）要证明 d(n)＝k+1 时，所有路径的端节点，结论成立，我们用反证法证明。

设 A_2 搜索树上有一个节点 n（d(n)=k+1）被 A_2 扩展了，而对应于 A_1 搜索树上的这个节点 n，没有被 A_1 扩展。根据归纳法假设条件，A_1 扩展了 n 的父节点，n 是在 A_1 搜索树上，因此 A_1 结束时，n 必定保留在其 OPEN 表上，n 没有被 A_1 选择扩展，有

$f_1(n)\geqslant f^*(s)$，即 $g_1(n)+h_1(n)\geqslant f^*(s)$

所以 $h_1(n)\geqslant f^*(s) - g_1(n)$ （1）

另一方面 A_2 扩展了 n，有

$f_2(n)\leqslant f^*(s)$，即 $g_2(n)+h_2(n)\leqslant f^*(s)$

所以 $h_2(n)\leqslant f^*(s)-g_2(n)$ （2）

由于 d=k 时，A_2 扩展的节点，A_1 也一定扩展，故有

$g_1(n)\leqslant g_2(n)$（因 A_1 扩展的节点数可能较多）

所以 $h_1(n)\geqslant f^*(s) - g_1(n)\geqslant f^*(s) - g_2(n)$ （3）

比较式（2）、（3）可得：至少在节点 n 上有 $h_1(n)\geqslant h_2(n)$，这与定理的前提条件矛盾，因此存在节点 n 的假设不成立。[证毕]

在定理 4 中所说的“有两个 A*算法 A_1 和 A_2”，指的是对于同一个问题，分别定义了两个启发函数 h_1 和 h_2。这里要强调几点：首先，这里的 A_1 和 A_2 都是 A*的，也就是说定义的 h_1 和 h_2 都要满足 A*算法的条件。第二，只有当对于任何一个节点 n，都有 $h_2(n)>h_1(n)$时，定理才能保证用 A_2 搜索所扩展的节点数≤用 A_1 搜索所扩展的节点数。而如果仅是 $h_2(n)\geqslant h_1(n)$时（比定理的条件多了一个“等于”，而不只是单纯的“大于”），定理并不能保证用 A_2 搜索所扩展的节点数≤用 A_1 搜索所扩展的节点数。也就是说，如果仅是 $h_2(n)\geqslant h_1(n)$，有等于的情况出现，可能会有用 A_2 搜索所扩展的节点数反而多于用 A_1 搜索所扩展的节点数的情况。第三，这里所说的“扩展的节点数”，是这样来计算的，同一个节点不管它被扩展多少次（在 A 算法的第六步，对于 m_l 类节点，存在重新放回到 OPEN 表的可能，因此一个节点有可能被反复扩展多次，在后面我们会看到这样的例子），在计算“扩展的节点数”时，都只计算一次，而不

管它被重复扩展了多少次。

该定理的意义在于，在使用 A*算法求解问题时，定义的启发函数 h，在满足 A*的条件下，应尽可能地大一些，使其接近于 h*，这样才能使得搜索的效率高。

由这个定理可知，使用启发函数 h(n)的 A*算法，比不使用 h(n) (h(n)≡0)的算法，求得最佳路径时扩展的节点数要少，图 1.11 的搜索图例子可看出比较的结果。当 h≡0 时，求得最佳解路（s，C，J，t_7），其 f*(s)=8，但除 t_1~t_8 外所有节点都扩展了，即求出所有解路后，才找到耗散值最小的路径。而引入启发函数（设其函数值如图中节点旁边所示）后，除了最佳路径上的节点 s，C，J 被扩展外，其余的节点都没有被扩展。当然一般情况下，并不一定都能达到这种效果，主要在于获取完备的启发信息较为困难。

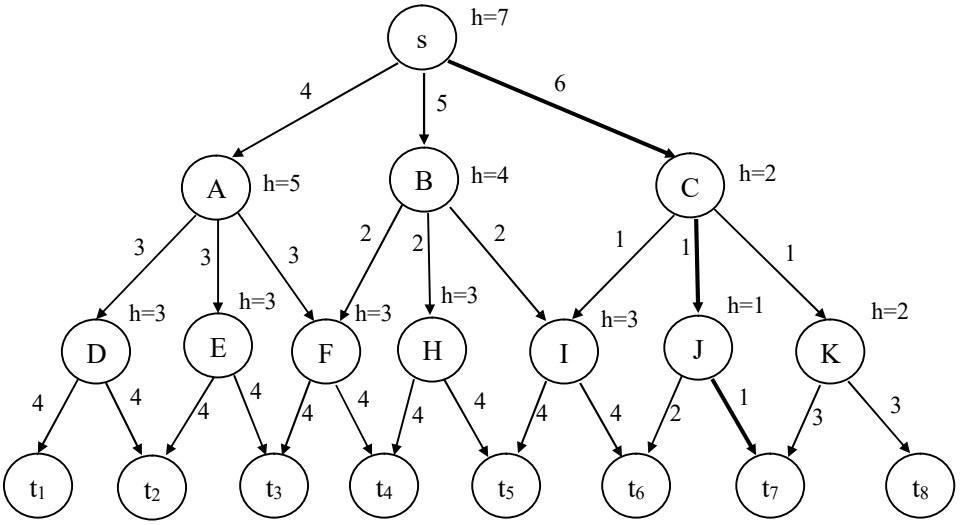


图 1.11 启发函数 h(n)的效果比较

7．A*算法的改进

上一节讨论了启发函数对扩展节点数所起的作用。如果用扩展节点数作为评价搜索效率的准则，那么可以发现 A 算法第 6 步中，对 m_1 类节点要重新放回 OPEN 表中的操作，将引起多次扩展同一个节点的可能，因而即使扩展的节点数少，但重复扩展某些节点，也将导致搜索效率下降。图 1.12 给出同一节点多次扩展的例子，并列出了调用算法 A*过程时 OPEN 和 CLOSED 表的状态。从 CLOSED 表可以看出，在修改 m_1 类节点指针过程中，节点 A、B、C 重复扩展，次数分别为 8、4、2，总共扩展 16 次节点。

从这个例子看出，如果不使用启发函数，则每个节点仅扩展一次，虽然扩展的节点数相同，但 A*扩展的次数多。如果对启发函数施加一定的限制——单调限制，则当 A*算法选某一个节点扩展时，就已经找到到该节点的最佳路径。下面就来证明这个结论。

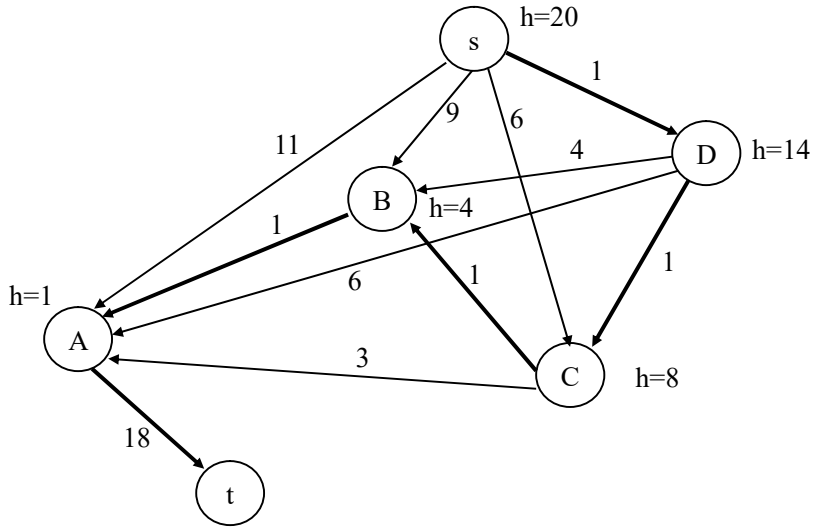


图 1.12　A*算法多次扩展同一节点搜索图例子

OPEN 表					CLOSED 表				
初始化 (s(20))					( )				
1 (A(12)	B(13)	C(14)	D(15))		(s(20))				
2 (	B(13)	C(14)	D(15)	t(29))	(s(20)	A(12))			
3 (A(11)		C(14)	D(15)	t(29))	(s(20		B(13))		
4 (		C(14)	D(15)	t(28))	(s(20)	A(11)	B(13))		
5 (A(10)	B(11)		D(15)	t(28))	(s(20			C(14))	
6 (	B(11)		D(15)	t(27))	(s(20)	A(10)		C(14))	
7 (A(9)			D(15)	t(27))	(s(20		B(11)	C(14))	
8 (			D(15)	t(26))	(s(20)	A(9)	B(11)	C(14))	
9 (A(8)	B(9)	C(10)		t(26))	(s(20				D(15))
10 (	B(9)	C(10)		t(25))	(s(20)	A(8)			D(15))
11 (A(7)		C(10)		t(25))	(s(20		B(9)		D(15))
12 (		C(10)		t(24))	(s(20)	A(7)	B(9)		D(15))
13 (A(6)	B(7)			t(24))	(s(20			C(10)	D(15))
14 (	B(7)			t(23))	(s(20)	A(6)		C(10)	D(15))
15 (A(5)				t(23))	(s(20		B(7)	C(10)	D(15))
16 (				t(22))	(s(20)	A(5)	B(7)	C(10)	D(15))
17 成功结束									

一个启发函数 h，如果对所有节点 n_i 和 n_j（n_j 是 n_i 的子节点），都有 $h(n_i) - h(n_j) \leqslant C(n_i, n_j)$或 $h(n_i) \leqslant C(n_i, n_j) + h(n_j)$且 $h(t_i)=0$，则称该 h 函数满足单调限制条件。其意义是从 n_i 到目标节点，最佳路径耗散值估计 $h(n_i)$不大于 n_j 到目标节点最佳路径耗散值估计 $h(n_j)$与 n_i 到 n_j 孤线耗散值两者之和。

对八数码问题，$h(n)=W(n)$是满足单调限制的条件。证明如下：

当用“不在位的奖牌数”来定义八数码问题的 h 时（即 $h(n)=W(n)$），其目标的 h 值显然等于 0。第二个条件成立。

由于八数码问题一次只能移动一个将牌，因此当移动完一个将牌后，会有以下三种情况：

（1）一个将牌从“在位”移动到了“不在位”，其结果是使得 h 值增加 1，这时 $h(n_i) - h(n_j)=-1$；

（2）一个原来就“不在位”的将牌，移动后，还是“不在位”，其结果是使得 h 值不变，这时 $h(n_i) - h(n_j)=0$；

（3）一个将牌从“不在位”移动到了“在位”，其结果使得 h 值减少 1，这时 $h(n_i) - h(n_j)=1$。

综合以上三种情况，均有 $h(n_i) - h(n_j) \leqslant 1$。而由于八数码问题每走一定耗散值为 1，所以有 $C(n_i, n_j)=1$。所以有：$h(n_i) - h(n_j) \leqslant C(n_i, n_j)$，单调条件的第一条也被满足。所以当用“不在位的奖牌数”来定义八数码问题的 h 时，该 h 是单调的。

定理 5：若 h(n)满足单调限制条件，则 A*扩展了节点 n 之后，就已经找到了到达节点 n 的最佳路径。即若 A*选 n 来扩展，在单调限制条件下有 $g(n)=g^*(n)$。

证明：设 n 是 A*选作扩展的任一节点，若 $n=s$，显然有 $g(s)=g^*(s)=0$，因此考虑 $n \neq s$ 的情况。

我们用序列 $P=(n_0=s, n_1, \cdots, n_k=n)$表达到达 n 的最佳路径。现在从 OPEN 中取出非初始节点 n 扩展时，假定没有找到 P，这时 CLOSED 中一定会有 P 中的节点（至少 s 是在 CLOSED 中，n 刚被选作扩展，不在 CLOSED 中），把 P 序列中（依顺序检查）最后一个出现在 CLOSED 中的节点称为 n_j，那么 n_{j+1} 是在 OPEN 中（$n_{j+1} \neq n$），由单调限制条件，对任意 i 有

$$g^*(n_i) + h(n_i) \leqslant g^*(n_i) + C(n_i, n_{i+1}) + h(n_{i+1}) \qquad (1)$$

因为 n_i 和 n_{i+1} 在最佳路径上，所以有

$$g^*(n_{i+1}) = g^*(n_i) + C(n_i, n_{i+1})$$

代入（1）式后有 $g^*(n_i) + h(n_i) \leqslant g^*(n_{i+1}) + h(n_{i+1})$

这个不等式对 P 上所有相邻的节点都合适，若从 $i=j$ 到 $i=k-1$ 应用该不等式，并利用传递性有

$g^*(n_{l+1})+h(n_{l+1}) \leqslant g^*(n_k)+h(n_k)$

即　　$f(n_{l+1}) \leqslant g^*(n)+h(n)$　　（2）

另一方面，A*选 n 来扩展，必有

$f(n)=g(n)+h(n) \leqslant f(n_{j+1})$　　（3）

比较（2）、（3）得 $g(n) \leqslant g^*(n)$，但已知 $g(n) \geqslant g^*(n)$，因此选 n 扩展时必有 $g(n)=g^*(n)$，即找到了到达 n 的最佳路径。

[证毕]

定理 6：若 h(n)满足单调限制，则由 A*所扩展的节点序列，其 f 值是非递减的，即 $f(n_i) \leqslant f(n_j)$

证明：由单调限制条件：

$h(n_i) - h(n_j) \leqslant C(n_i, n_j)$

即　　$f(n_i) - g(n_i) - f(n_j) + g(n_j) \leqslant C(n_i, n_j)$

$f(n_i) - g(n_i) - f(n_j) + g(n_i) + C(n_i, n_j) \leqslant C(n_i, n_j)$

$f(n_i) - f(n_j) \leqslant 0$。[证毕]

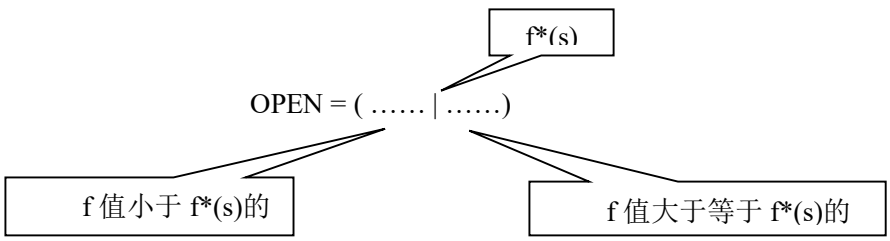
根据定理 5，在应用 A*算法时，在第 6 步可不必进行节点的指针修正操作，因而改善了 A*的效率。

另一方面我们从图 1.12 的例子看出，因 h 不满足单调限制条件，在扩展节点 n 时，有可能还没有找到到达 n 的最佳路径，因此该节点还会再次被放入 OPEN 中，从而造成了该节点被重复扩展。

定义满足单调条件的 h，是避免重复扩展节点的好办法。但是对于实际问题来说，定义一个单调的 h 并不是一件很容易的事，那么能否通过修改算法，来达到避免或者减少重复节点扩展的问题呢？答案是肯定的。只要适当地修改 A 算法，就可以达到这样的目的。

在改进 A*算法的时候，一是要保持 A*算法的可采纳性，二是不能增加过多的计算工作量。由推论 2.1 我们知道，OPEN 表上任一具有 $f(n) < f^*(s)$的节点 n 定会被扩展。由推论 3.1 我们知道，A*选作扩展的任一节点，定有 $f(n) \leqslant f^*(s)$。这两个推论正是我们改进 A*算法的理论基础。

算法改进的基本思路是：如下图所示，我们仍像 A*算法那样，按照节点的 f 值从小到大排序 OPEN 表中的节点，我们以 f*(s)为界将 OPEN 表划分为两部分，一部分由那些 f 值小于 f*(s)的节点组成，我们称其为 NEST，其他的节点属于另一个部分。



由推论 2.1 我们知道，OPEN 表上任一具有 f(n) < f*(s)的节点 n 定会被扩展。所以，NEST 中的节点，不管先扩展，还是后扩展，在 A*结束前总归要被扩展，如果我们改变一下他们的扩展顺序，不会影响到算法扩展节点的个数。我们知道，如果 h 是单调的，就可以避免重复节点扩展问题。而如果 h≡0 的话，由于任何两个节点间的耗散值是大于 0 的，因此对于任何节点 n_i 和 n_j，其中 n_j 是 n_i 的后继节点，有 $h(n_i) - h(n_j) = 0 - 0 < C(n_i, n_j)$，且 h(t)＝0（t 是目标节点），所以恒等于 0 的 h 是单调的。因此，如果对于 NEST 中的节点，我们令其 h 值为 0 的话，至少在它们之间不会引起重复扩展节点问题。由以上分析，如果我们这样改变算法，对于在 NEST 中出现的节点，我们令其 h 值为 0，按照 f(n)=g(n)进行扩展的话，既可以避免重复扩展节点问题，又不会增加扩展节点的个数，而且也没有增加什么计算工作量，是对 A*算法一种很好的改进。当然这种改进并不是彻底的，它只是可能减少重复扩展节点问题，并不能保证完全避免，最坏情况下，与 A*完全一样，在避免重复扩展节点这一点上没有任何改进。

那么，由于在问题得到解决之前，f*(s)的值是未知的，如何得到 NEST 呢？我们可以用一种近似的方法，来得到一个 NEST 的子集。由推论 3.1 我们知道，A*选作扩展的任一节点，定有 f(n)≤f*(s)，所以我们可以用到目前为止扩展过的节点中，最大的 f 值作为 f*(s)的近似值，记做 f_m。f_m 是动态改变的，随着搜索的进行，越来越接近 f*(s)值。这样，在实际使用时，不是直接用 f*(s)对 OPEN 表进行划分，而是用 f_m 对 OPEN 进行划分，从而得到 NEST 的子集。这样得到的 NEST 的子集，我们仍然称其为 NEST。

下面我们给出修正的 A 算法。

修正过程 A

①OPEN:＝(s)，f(s)=g(s)+h(s)=h(s)，f_m:=0；

②LOOP: IF OPEN＝(　)THEN EXIT(FAIL)；

③NEST：＝{n_i｜f(n_i)＜f_m＝；NEST 给出 OPEN 中满足 f＜f_m 的节点集合。

IF NEST≠(　) THEN n:＝n(min g_i)

ELSE n:＝FIRST(OPEN)，f_m:=f(n)；NEST 不空时，取其中 g 最小者作为当前节点，否则取 OPEN 的第一个当前节点。

④－⑧同过程 A

现在看一下用修正 A*搜索图 1.12 的情况：

OPEN	fm	CLOSED
初始化 s(0+20)	0	
1 (A(11+1) B(9+4) C(6+8) D(1+14))	20	(s(0+20))
2 (A(7+1) B(5+4) C(2+8))	20	(s(0+20) D(1+14))
3 (A(5+1) B(3+4))	20	(s(0+20) C(2+8) D(1+14))
4 (A(4+1))	20	(s(0+20) B(3+4) C(2+8) D(1+14))
5 (t(22+0))	20	(s(0+20) A(4+1) B(3+4) C(2+8) D(1+14))
成功结束		

由 OPEN 表和 CLOSED 表中的状态看出，修正后的算法减少了重复扩展的次数。一般情况下，它比 A*算法扩展节点的次数要少或相等。

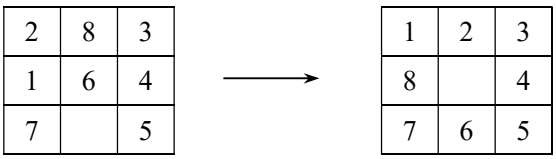
8．迭代加深 A*算法

A*算法虽然可以有效地减少搜索范围，但是当问题规模比较大时，会生成比较多的节点，同样会带来占用过多的内存的问题。如同宽度优先搜索算法一样，也可以用回溯搜索策略，通过逐步加深的方法模拟 A*算法，实现用较少的内存空间，达到 A*算法一样的效果，搜索到问题的最优解。只是这里逐步加深的不是搜索节点的深度，而是 A*算法的 f 值。其基本思想是：在深度优先搜索算法中增加一个当某个节点的 f 值大于给定的 f_0 时就进行回溯的条件，循环调用深度优先搜索策略，逐渐加大 f_0 值，直到找到最优解为止，或者找不到解结束。如何加大 f_0 值呢？一种简单的方法就是用前一次产生回溯的节点中最小的 f 值代替 f_0。这样就得到了迭代加深 A*算法，用深度优先搜索的空间，得到 A*算法的效果。

9．A*算法应用举例

A*算法的理论意义在于给出了求解最佳解的条件 $h(n) \leqslant h^*(n)$。对给定的问题，函数 $h^*(n)$（n 是变量）在问题有解的条件下客观上是存在的，但在问题求解过程中不可能明确知道，因此对实际问题，能不能使所定义的启发函数满足下界范围条件？如果困难很大，那么 A*算法的实际应用就会受到限制。下面将通过几个应用实例来说明这个问题。

（1）八数码问题



初始状态　　目标状态

对八数码问题，如果启发函数根据任意节点与目标之间的差异来定义，例如取h(n)＝W(n)，那么很容易看出，尽管我们对具体的h*(n)是多少很难确切知道，但根据“不在位”将牌个数这个估计，就能得出至少要移动 W(n)步才能达到目标，显然有W(n)≤h*(n)（假定为单位耗散的情况）。如果启发函数进一步考虑任意节点与目标之间距离的信息，例如取 h(n)＝P(n)，P(n)定义为每一个将牌与其目标位置之间距离（不考虑夹在其间的将牌）的总和，那么同样能断定至少也要移动 P(n)步才能达到目标，因此有 P(n)≤h*(n)。图 1.13 给出 h(n)＝P(n)时的搜索图，节点旁边还标出 W(n)和 P(n)启发函数值。由解路可给出 g*(n)和 h*(n)的值，由此可得最佳路径上的节点有 f*＝g*+h*＝5。

下面给出该八数码问题取不同启发函数，应用 A*算法求得最佳解时所扩展和生成的节点数。

启发函数	h(n)＝0	h(n)＝W(n)	h(n)＝P(n)
扩展节点数	26	6	5
生成节点数	46	13	11

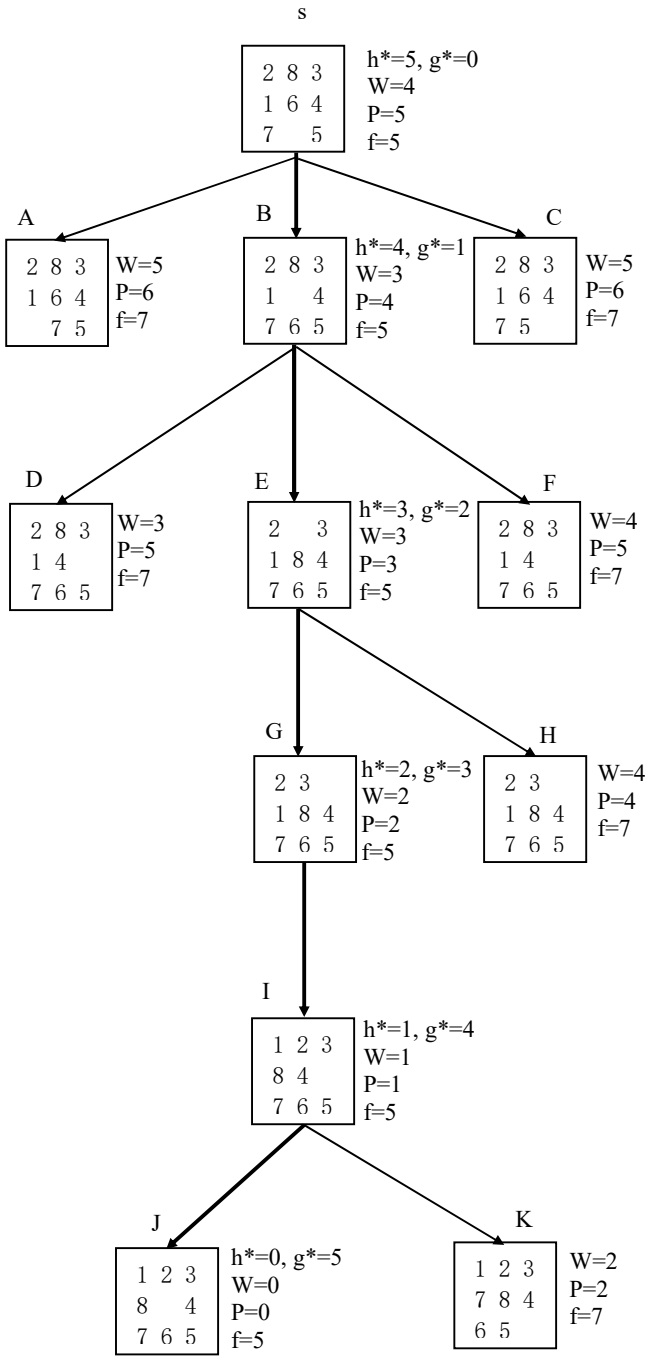
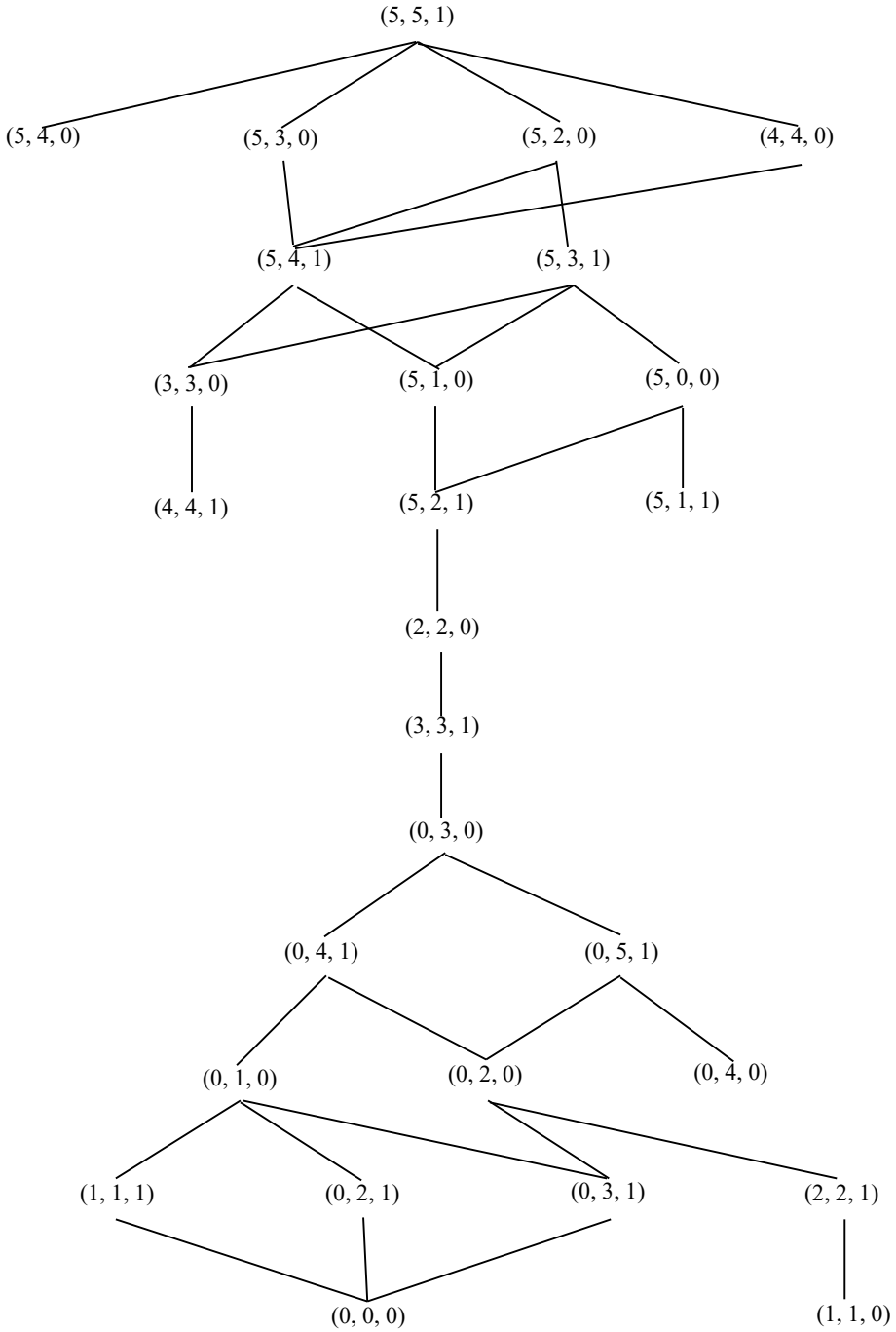
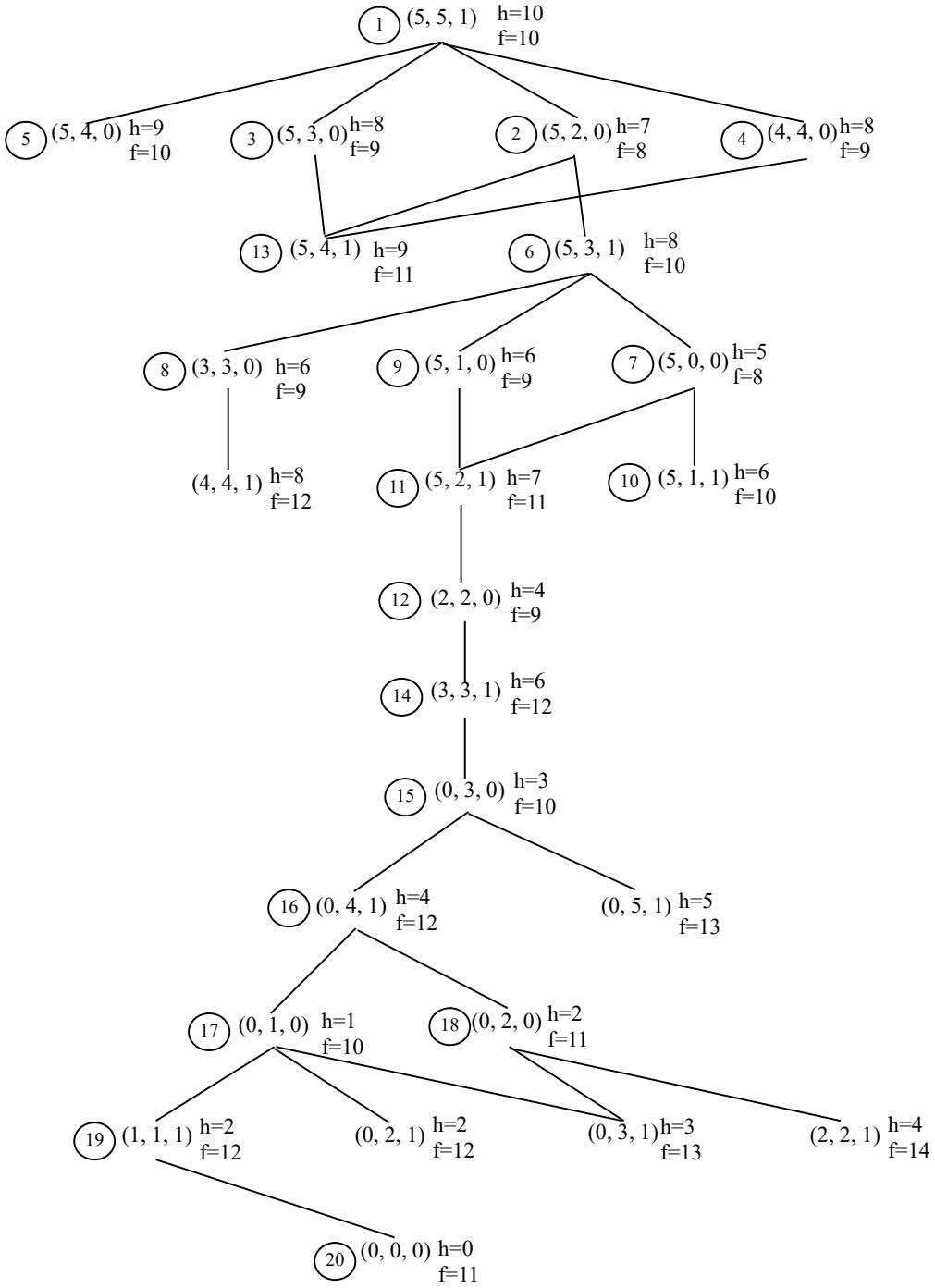


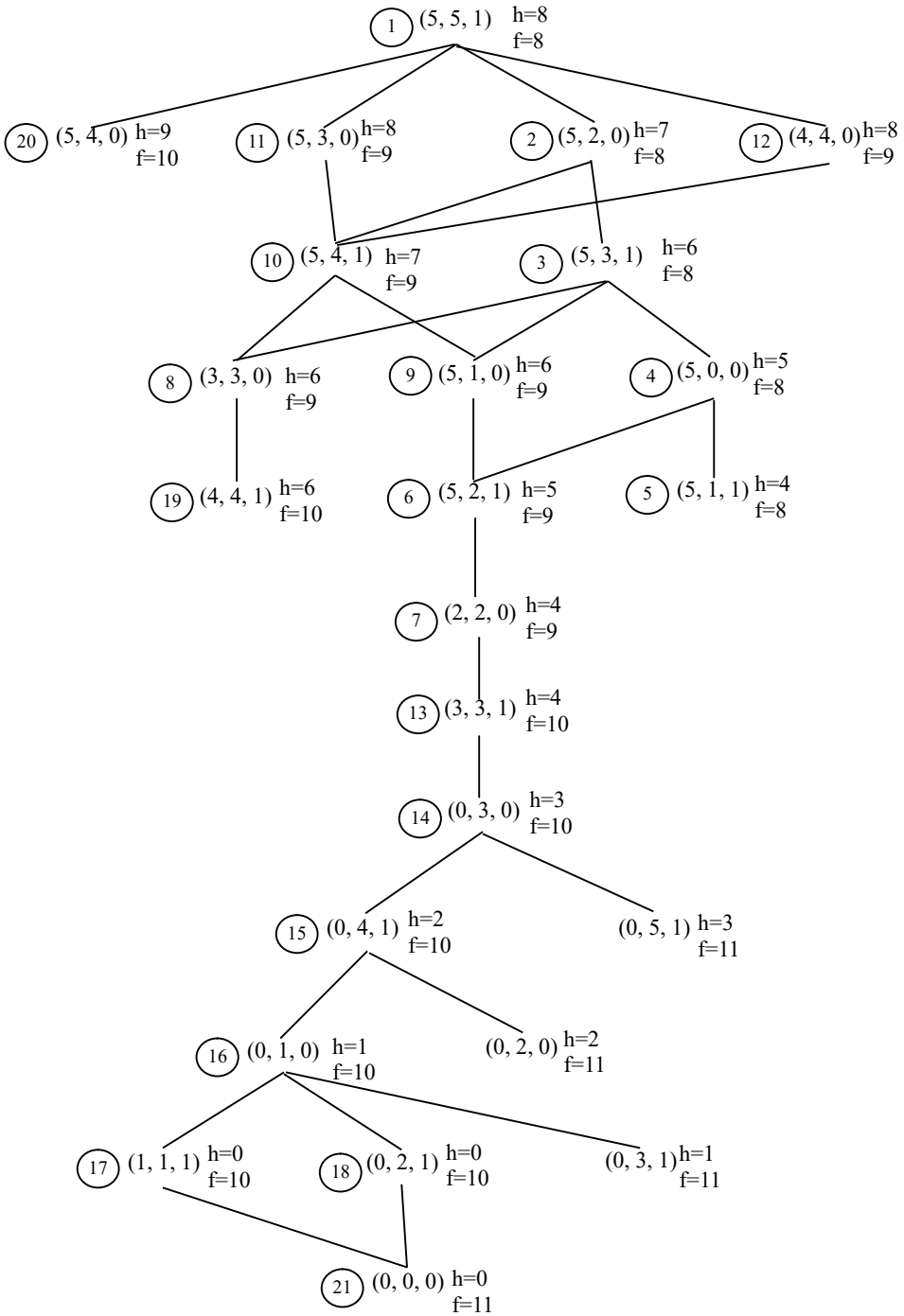
图 1.13　h(n)=P(n)的搜索树


(5, 5, 1)
(5, 4, 0)
(5, 3, 0)
(5, 2, 0)
(4, 4, 0)
(5, 4, 1)
(5, 3, 1)
(3, 3, 0)
(5, 1, 0)
(5, 0, 0)
(4, 4, 1)
(5, 2, 1)
(5, 1, 1)
(2, 2, 0)
(3, 3, 1)
(0, 3, 0)
(0, 4, 1)
(0, 5, 1)
(0, 1, 0)
(0, 2, 0)
(0, 4, 0)
(1, 1, 1)
(0, 2, 1)
(0, 3, 1)
(2, 2, 1)
(0, 0, 0)
(1, 1, 0)

h(n) = 0


1 (5, 5, 1) h=10 f=10
5 (5, 4, 0) h=9 f=10
3 (5, 3, 0) h=8 f=9
2 (5, 2, 0) h=7 f=8
4 (4, 4, 0) h=8 f=9
13 (5, 4, 1) h=9 f=11
6 (5, 3, 1) h=8 f=10
8 (3, 3, 0) h=6 f=9
9 (5, 1, 0) h=6 f=9
7 (5, 0, 0) h=5 f=8
(4, 4, 1) h=8 f=12
11 (5, 2, 1) h=7 f=11
10 (5, 1, 1) h=6 f=10
12 (2, 2, 0) h=4 f=9
14 (3, 3, 1) h=6 f=12
15 (0, 3, 0) h=3 f=10
16 (0, 4, 1) h=4 f=12
(0, 5, 1) h=5 f=13
17 (0, 1, 0) h=1 f=10
18 (0, 2, 0) h=2 f=11
19 (1, 1, 1) h=2 f=12
(0, 2, 1) h=2 f=12
(0, 3, 1) h=3 f=13
(2, 2, 1) h=4 f=14
20 (0, 0, 0) h=0 f=11

h(n) = M+C


1 (5, 5, 1) h=8 f=8
20 (5, 4, 0) h=9 f=10
11 (5, 3, 0) h=8 f=9
2 (5, 2, 0) h=7 f=8
12 (4, 4, 0) h=8 f=9
10 (5, 4, 1) h=7 f=9
3 (5, 3, 1) h=6 f=8
8 (3, 3, 0) h=6 f=9
9 (5, 1, 0) h=6 f=9
4 (5, 0, 0) h=5 f=8
19 (4, 4, 1) h=6 f=10
6 (5, 2, 1) h=5 f=9
5 (5, 1, 1) h=4 f=8
7 (2, 2, 0) h=4 f=9
13 (3, 3, 1) h=4 f=10
14 (0, 3, 0) h=3 f=10
15 (0, 4, 1) h=2 f=10
(0, 5, 1) h=3 f=11
16 (0, 1, 0) h=1 f=10
(0, 2, 0) h=2 f=11
17 (1, 1, 1) h=0 f=10
18 (0, 2, 1) h=0 f=10
(0, 3, 1) h=1 f=11
21 (0, 0, 0) h=0 f=11

h(n) = M+C-2B

图 1.14 M－C 问题搜索图

（2）传教士和野人问题（Missionaries and Cannibals，M–C 问题）

一般的传教士和野人问题描述如下：

有 N 个传教士和 N 个野人来到河边准备渡河，河岸有一条船，每次至多可供 k 人乘渡。问传教士为了安全起见，应如何规划摆渡方案，使得任何时刻，在河的两岸以及船上的野人数目总是不超过传教士的数目（但允许在河的某一岸只有野人而没有传教士）。

设 M＝C＝5，K≤3，由分析可知至少要往返 10 次才可能把全部 M 和 C 摆渡到右岸，因此 M 和 C 的线性组合可作为启发函数的基本分量，此外还可以考虑有船与否对摆渡的影响。图 1.14 给出 h(n)＝0、h(n) =M+C、h(n)＝M+C–2B 三种启发函数的搜索图（假定为单位耗散）。可以看出 h(n)＝0 的搜索图就是该问题的状态空间图；取 h(n)＝M+C 不满足 h(n)≤h*(n)条件；只有 h(n)＝M+C–2B 可满足 h(n)≤h*(n)。

要说明 h(n)＝M+C 不满足 A*条件是很容易的，只需要给出一个反例就可以了。比如状态(1, 1, 1)，h(n)=M+C=1+1=2，而实际上只要一次摆渡就可以达到目标状态，其最优路径的耗散值为 1。所以不满足 A*的条件。

下面我们来证明 h(n)＝M+C–2B 是满足 A*条件的。

我们分两种情况考虑。先考虑船在左岸的情况。如果不考虑限制条件，也就是说，船一次可以将三人从左岸运到右岸，然后再有一个人将船送回来。这样，船一个来回可以运过河 2 人，而船仍然在左岸。而最后剩下的三个人，则可以一次将他们全部从左岸运到右岸。所以，在不考虑限制条件的情况下，也至少需要摆渡 $\left\lceil \frac{M+C-3}{2} \right\rceil \times 2+1$ 次。其中分子上的“－3”表示剩下三个留待最后一次运过去。除以“2”是因为一个来回可以运过去 2 人，需要 $\frac{M+C-3}{2}$ 个来回，而“来回”数不能是小数，需要向上取整，这个用符号 $\lceil\ \rceil$ 表示。而乘以“2”是因为一个来回相当于两次摆渡，所以要乘以 2。而最后的“＋1”，则表示将剩下的 3 个运过去，需要一次摆渡。

化简有：

$$\left\lceil \frac{M+C-3}{2} \right\rceil \times 2+1 \geq \frac{M+C-3}{2} \times 2+1 = M+C-3+1 = M+C-2$$

再考虑船在右岸的情况。同样不考虑限制条件。船在右岸，需要一个人将船运到左岸。因此对于状态(M，C，0)来说，其所需要的最少摆渡数，相当于船在左岸时状态(M+1，C，1)或(M，C+1，1)所需要的最少摆渡数，再加上第一次将船从右岸送到左岸的一次摆渡数。因此所需要的最少摆渡数为：$(M+C+1)-2+1$。其中(M+C+1)的“＋1”表示送船回到左岸的那个人，而最后边的“＋1”，表示送船到左岸时的一次摆渡。

化简有：$(M+C+1)-2+1=M+C$。

综合船在左岸和船在右岸两种情况下，所需要的最少摆渡次数用一个式子表示为：$M+C-2B$。其中 B＝1 表示船在左岸，B＝0 表示船在右岸。

由于该摆渡次数是在不考虑限制条件下，推出的最少所需要的摆渡次数。因此，当有限制条件时，最优的摆渡次数只能大于等于该摆渡次数。所以该启发函数 h 是满足 A*条件的。

在图 1.14 所示的几个搜索图中，圆圈中的数字表示节点扩展的顺序。当出现 f 值相同的节点时，其扩展顺序是任意的，所以节点的扩展的顺序并不是唯一的。

（3）迷宫问题

迷宫图从入口到出口有若干条通路，求从入口到出口处最短路径的走法。

图 1.15 为一简单迷宫示意图及其平面坐标表示。以平面坐标图来表示迷宫的通路时，问题的状态以所处的坐标位置来表示，即综合数据库定义为(x，y)，1≤x，y≤N，则迷宫问题归结为求(1，1)到(4，4)的最短路径问题。

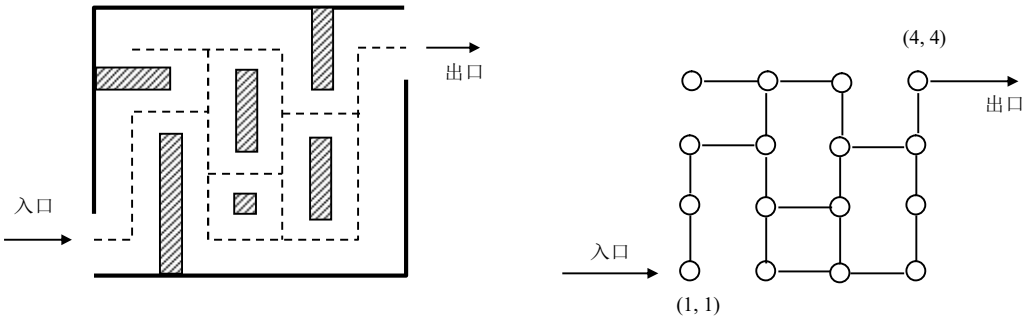


图 1.15 迷宫问题及其表示

迷宫走法规定为向东、南、西、北前进一步，由此可得规则集简化形式如下：

R_1：if (x，y) then (x+1，y)

R_2：if (x，y) then (x，y–1)

R_3：if (x，y) then (x–1，y)

R_4：if (x，y) then (x，y+1)

对于这个简单例子，可给出状态空间如图 1.16 所示。

为求得最佳路径，可使用 A*算法。假定搜索一步取单位耗散，则可定义：

$$h(n)=|X_G - x_n| + |Y_G - y_n|$$

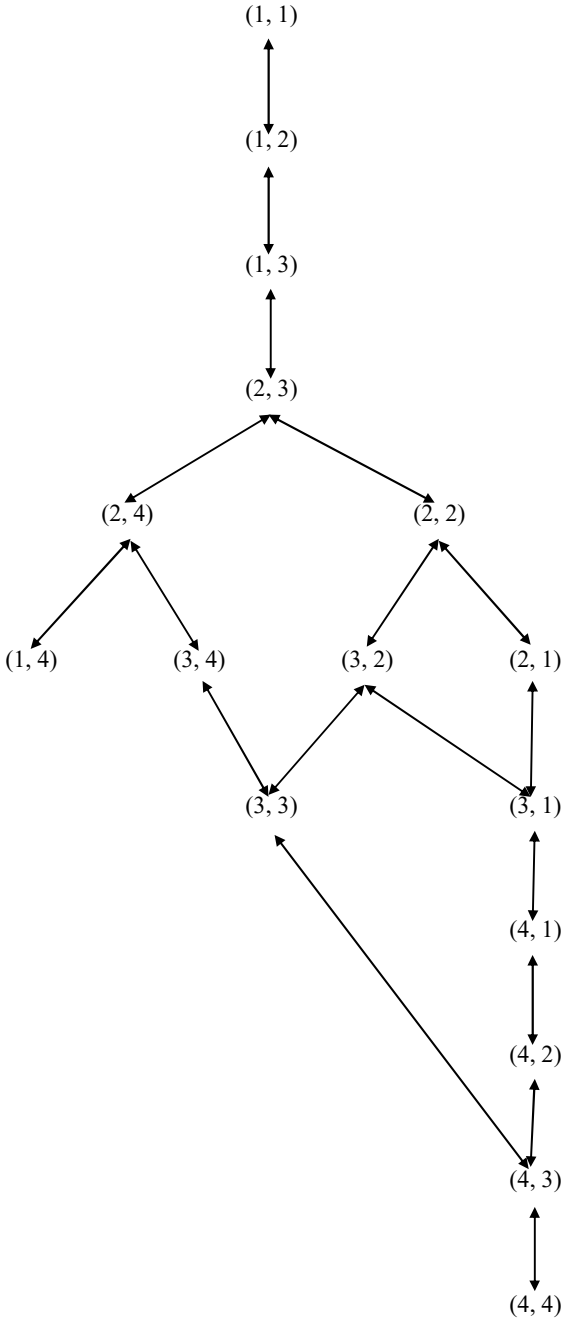


图 1.16 状态空间图

其中(X_G，Y_G)为目标点坐标，(x_n，y_n)为节点 n 的坐标。由于该迷宫问题所有路都是水平或垂直的，没有斜路，因此，$h(n)=|X_{G-Xn}| + |Y_G - y_n|$显然可以满足 A*的条件，即 h(n)≤h*(n)。取 g(n)＝d(n)，有 f(n)＝d(n)+h(n)。再设当不同节点的 f 值相等时，以深度优先排序，则搜索图如图 1.17 所示。最短路径为((1，1)，(2，3)，(2，4)，(3，4)，(3，3)，(4，3)，(4，4))。

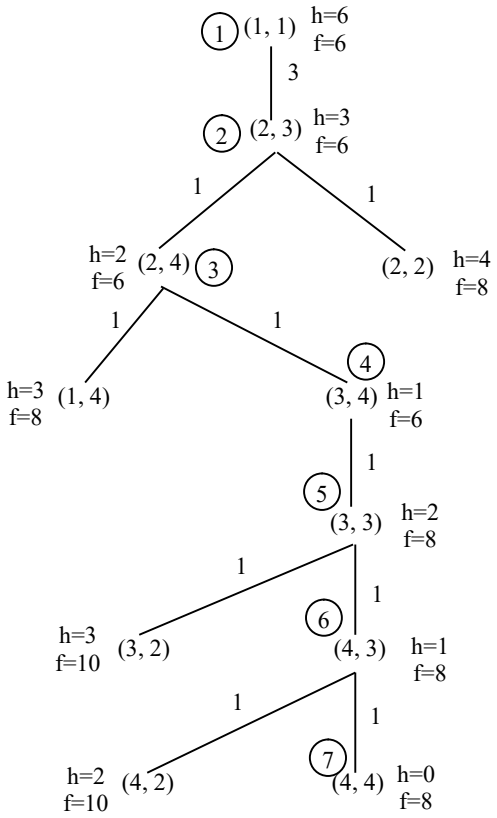
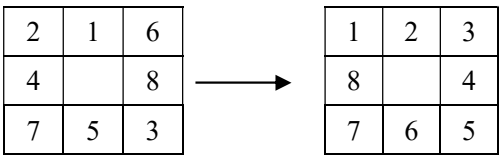


图 1.17　迷宫问题启发式搜索图

在该搜索图中，目标节点的 f 是 8，有几个节点的 f 值也是 8，那么这几个 f 值为 8 的节点，也有被扩展的可能，就看他们在 OPEN 表中的具体排列次序了。这里假定了 f 值相等时，以深度优先排序。

10．评价函数的启发能力



首先我们通过八数码问题为例说明 A 算法的启发能力与选择启发函数 h(n)的关系。一般来说启发能力强，则搜索效率较高。有时选用不是 h*(n)下界范围的 h(n)时，虽然会牺牲找到最佳解的性能，但可使启发能力得到改善，从而有利于求解一些较难的问题。

求解这个八数码问题，使用启发函数 h(n)＝P(n)时，仍不能估计出交换相邻两个将牌位置难易程序的影响，为此可再引入 S(n)分量。S(n)是对节点 n 中将牌排列顺序的计分值，规定对非中心位置的将牌，顺某一方向检查，若某一将牌后面跟的后继者和目标状态相应将牌的顺序相比不一致时，则该将牌估分取 2，一致时则估分取 0；对中心位置有将牌时估分取 1，无将牌时估分值取 0；所有非中心位置每个将牌估分总和加上中心位置的估分值定义为 S(n)。依据这些启发信息，取 h(n)＝P(n)+3S(n)时，就是用 f(n)＝g(n)+P(n)+3S(n)来估计最佳路径的耗散值。f(n)值小的节点，确能反映该节点愈有希望处于到达目标节点的最佳路径上。图 1.18 给出了该问题的搜索树，图中给出各节点的 f 值，圆圈中的数字表示扩展顺序。由图看出 h(n)函数不满足下界范围，但在该问题中，刚好找到了最小长度（18 步）的解路径。

该例子给了一个不满足 A*条件的 h 函数。从图上可以看出，启发效果非常的好，对于需要 18 步才能完成的 8 数码问题，几乎没有扩展什么多余的节点，就找到了解路径。这里所用的方法一是组合两个不同的启发函数；二是采取加权的方法（这里对 S(n)加权为 3），来加大 S(n)的作用。这样得到的启发函数由于不满足 A*条件，因此不能保证找到问题的最佳解，但往往可以提高搜索效率，加快找到解的速度。由于这样的启发函数还是反映了被评估节点到目标节点路径耗散值的多少，算法虽然不能一定找到最优解，但一般来说，找到的也是一个可以被接受的满意解。很多情况下，满意解就足够了，最优解并没有什么特殊的意义，二者可能相差很少，但却使得问题简单了很多。

还有一个决定搜索算法启发能力的因素是涉及到计算启发函数的工作量，从被扩展的节点数最少的角度看，h≡h*最优，但这可能导致繁重的计算工作量。有时候一个不是 h*下界范围的 h 函数可能比起下界范围的 h 函数更容易计算，而且被扩展节点的总数可以减少，使启发能力加倍得到改善，虽然牺牲了可采纳性，但从启发能力的角度看仍是可取的。

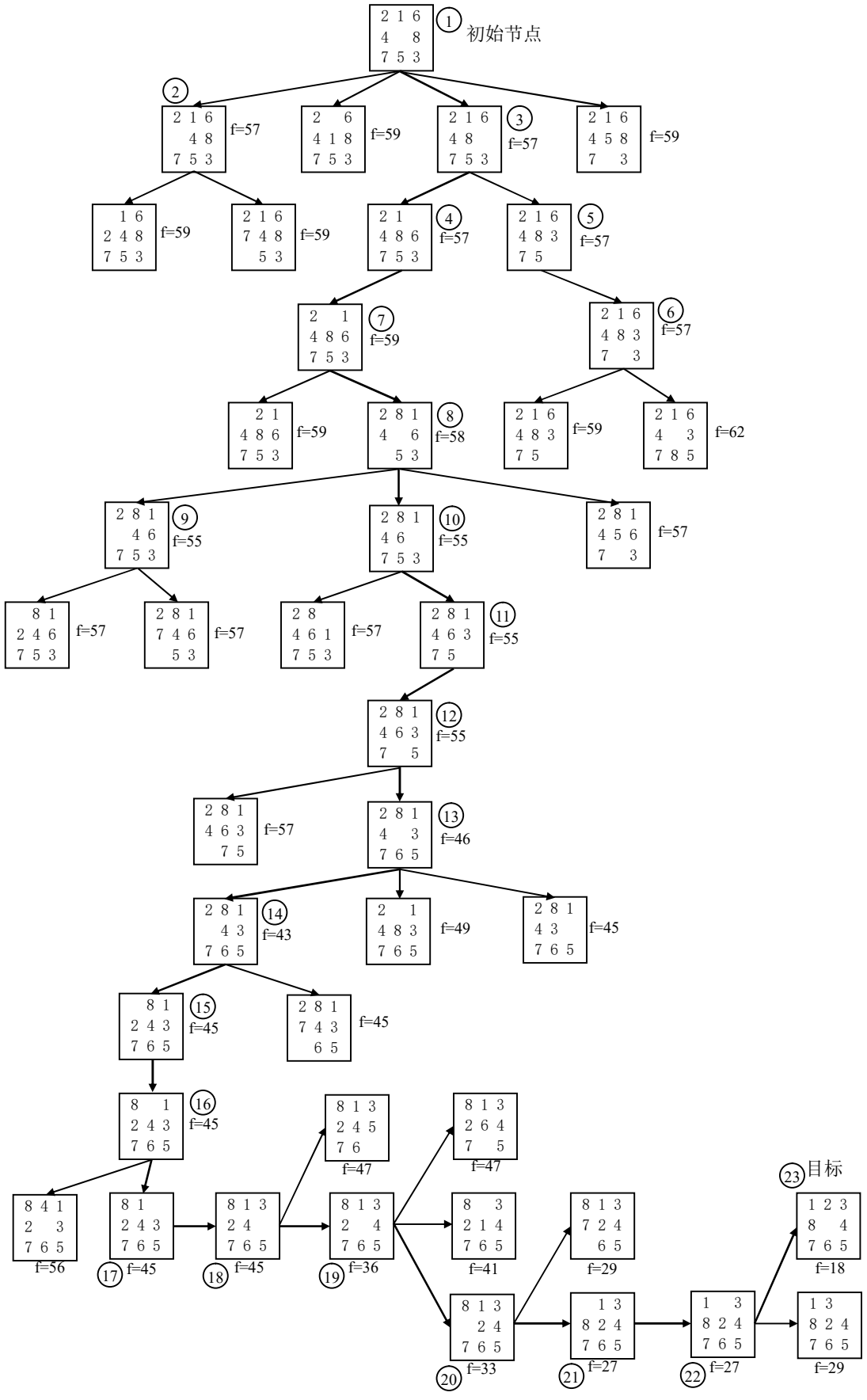

2 1 6 / 4 8 / 7 5 3
1
初始节点
2
2 1 6 / 4 8 / 7 5 3
f=57
2 6 / 4 1 8 / 7 5 3
f=59
2 1 6 / 4 8 / 7 5 3
3
f=57
2 1 6 / 4 5 8 / 7 3
f=59
1 6 / 2 4 8 / 7 5 3
f=59
2 1 6 / 7 4 8 / 5 3
f=59
2 1 / 4 8 6 / 7 5 3
4
f=57
2 1 6 / 4 8 3 / 7 5
5
f=57
2 1 / 4 8 6 / 7 5 3
7
f=59
2 1 6 / 4 8 3 / 7 3
6
f=57
2 1 / 4 8 6 / 7 5 3
f=59
2 8 1 / 4 6 / 5 3
8
f=58
2 1 6 / 4 8 3 / 7 5
f=59
2 1 6 / 4 3 / 7 8 5
f=62
2 8 1 / 4 6 / 7 5 3
9
f=55
2 8 1 / 4 6 / 7 5 3
10
f=55
2 8 1 / 4 5 6 / 7 3
f=57
8 1 / 2 4 6 / 7 5 3
f=57
2 8 1 / 7 4 6 / 5 3
f=57
2 8 / 4 6 1 / 7 5 3
f=57
2 8 1 / 4 6 3 / 7 5
11
f=55
2 8 1 / 4 6 3 / 7 5
12
f=55
2 8 1 / 4 6 3 / 7 5
f=57
2 8 1 / 4 3 / 7 6 5
13
f=46
2 8 1 / 4 3 / 7 6 5
14
f=43
2 1 / 4 8 3 / 7 6 5
f=49
2 8 1 / 4 3 / 7 6 5
f=45
8 1 / 2 4 3 / 7 6 5
15
f=45
2 8 1 / 7 4 3 / 6 5
f=45
8 1 / 2 4 3 / 7 6 5
16
f=45
8 1 3 / 2 4 5 / 7 6
f=47
8 1 3 / 2 6 4 / 7 5
f=47
8 4 1 / 2 3 / 7 6 5
f=56
8 1 / 2 4 3 / 7 6 5
17
f=45
8 1 3 / 2 4 / 7 6 5
18
f=45
8 1 3 / 2 4 / 7 6 5
19
f=36
8 3 / 2 1 4 / 7 6 5
f=41
8 1 3 / 7 2 4 / 6 5
f=29
23
目标
1 2 3 / 8 4 / 7 6 5
f=18
8 1 3 / 2 4 / 7 6 5
20
f=33
1 3 / 8 2 4 / 7 6 5
21
f=27
1 3 / 8 2 4 / 7 6 5
22
f=27
1 3 / 8 2 4 / 7 6 5
f=29

图 1.18　h(n)＝P(n)+3S(n)的搜索树

在某些情况下，要改变启发能力，可以通过对 h 函数乘以加权系数的简单方法实现。当加权系数很大时，g 分量的作用相对减弱，因而可略去不计，这相当于在搜索期间任何阶段上，我们不在乎到目前为止所得到的路径耗散值，而只关心找到目标节点所需的剩余工作量，即可以使用 f≡h 作为评价函数来对 OPEN 表上的节点排序。但是另一方面为了保证最终能找到通向目标节点的路径，即便并不要求寻找一条最小耗散的路径，也还是应当考虑 g 的作用。特别是当 h 不是一个理想的估计时，在 f 中把 g 考虑进去就是在搜索中添加一个宽度优先分量，从而保证了隐含图中不会有某些部分不被搜索到。而只扩展 h 值最小的节点，则会引起搜索过程扩展了一些靠不住的节点。

评价函数中，g 和 h 相对比例可通过选择 f＝g+wh 中 w 的大小加以控制。w 是一个正数，很大的 w 值则会过份强调启发分量，而过小的 w 值则突出宽度优先的特征。经验证明使 w 值随搜索树中节点深度成反比变化，可提高搜索效率。即在深度浅的地方，搜索主要依赖于启发分量；而较深的地方，搜索逐渐变成宽度优先，以保证到达目标的某一条路径最终被找到。

总结以上讨论，得出影响算法 A 启发能力的三个重要因素是：

（1）路径的耗散值；

（2）求解路径时所扩展的节点数；

（3）计算 h 所需的工作量。

因此选择 h 函数时，应综合考虑这些因素以便使启发能力最大。

1.5　搜索算法讨论

1．弱方法

人工智能范畴的一些问题都比较复杂，一般无法用直接求解的方法找到解答，因此通常都要藉助于搜索技术。前面几节讨论的搜索方法，其描述均与问题领域无关，如果把这些方法应用于特定问题的领域时，其效率依赖于该领域知识应用的情况，从我们举过的几个例子就可说明这个问题。但由于这些方法难于克服搜索过程的组合爆

炸问题，因此在人工智能领域中，把这些方法统称为“弱方法”。这些搜索方法可用来求解不存在确定求解算法的某一类问题，或者虽然有某种求解算法，但复杂性很高，有不少均属 NP－完全类的问题。为避免求解过程的组合爆炸，在搜索算法中引入启发性信息，多数情况能以较少的代价找到解，但并不能保证任何情况下都能获得解，这就是所谓“弱方法”的含义。当然如果引入强有力的启发信息，则求解过程就能显示出“强”的作用。下面我们来讨论用优选法求解极值这类问题时搜索过程的特点。

设状态是实数域[a，b]上实值连续函数 f，求该目标函数在何处取得极值及其大小。

在几何学中黄金分割法的思想是在区间[0，1]间取 0.618 和 0.382 两个特殊点来考虑问题：若 f(0.382)较优，则剪去[0.618，1]区段；若 f(0.618)较优，则剪去[0，0.382]区段；然后在新区间依此规则继续下去，直至函数 f 在某一点取得极值，这就是优先法的要点。显然目标函数 f 中包含了启发信息，下面给出该算法（应用该算法时先把[a，b]通过变换转换成[0，1]）：

黄金分割法

①x_1:＝0，x_2:＝1，x_3:＝0.382，x_4:＝0.618；赋变量初值

②LOOP1: IF $X_3 \approx X_4$ THEN EXIT(SUCCESS)；

IF $f(x_3)>f(x_4)$ THEN GO LOOP2;

IF $f(x_3)<f(x_4)$ THEN GO LOOP3;

IF $f(x_3)=f(x_4)$ THEN GO LOOP2∨LOOP3;可根据某种原则决定

③LOOP2: x_2:＝x_4，x_4:＝x_3，x_3:＝$x_1+x_2-x_4$；

④GO LOOP1；

⑤LOOP3: x_1:＝x_3，x_3:＝x_4，x_4:＝$x_1+x_2-x_3$；

⑥GO LOOP1；

由于计算中引入了极强的启发信息，因而获得最佳的搜索效果，可以证明 f 在[0，1]间具有单极值时，$f(x_3)$或 $f(x_4)$即为求得的极值点，而且求解过程搜索的点数是最少的。

前面我们讨论的几个搜索算法都属弱方法，实际上人工智能研究中提出属于这类的算法很多（如约束满足法，手段目的分析法等等），都可以用来求解某一特定问题，至于具体选择哪一种策略，很大程度上取决于问题的特征和实际要求。另外这些方法只提供了一种框架，对复杂问题只要能较好地运用特定问题的启发信息，就可能获得较好的搜索效果。

2．搜索算法分析

算法分析主要要回答这些方法执行的效果怎样，找到的解其优劣程度如何。在一般的计算机科学领域中，主要强调对算法进行数学分析，如严格数学分析遇到困难，则采取运行一组经过精心挑选的问题，再对其执行情况作统计分析。而人工智能领域研究这个问题的途径则采取将算法用某种语言具体实现，然后运行某个智能问题的典型实例，并观察其表现行为来进行分析比较。这主要是由于人工智能问题比较复杂，通常不容易对一过程是否可行作出令人信服的分析证明。此外，有时甚至无法把问题的值域描述清楚，因而也难于对程序行为作统计分析。再一点就是人工智能是一门实验科学，实践是目前主要的研究途径。

搜索过程最基本的一个分析是对深度为 D、分支因数为 B 的一棵完全树的节点数（为 B^D）进行讨论。显然如果一过程执行的时间随问题的规模变大而指数增长时，则该过程无实用意义。因此要研究改进穷举搜索的各种方法，并通过所得到的搜索时间上界，来和穷举法比较改善的程度。目前优于穷举法的若干方法有三种情况。

（1）能保证找到的解与穷举法所得结果一样好，但耗时较少。这类方法的问题是能否给出某一方法具体有多快。

（2）对问题的一些实例，耗费时间和穷举法一样，但对另一些实例则较穷举法好得多。这类方法的问题是运行一组一般问题，期望有多快。

（3）得到的解比穷举法结果较差。问题是要在给定时间内找到解，这个解与最佳解之间有多大差别。

此外对 NP－完全类问题，已知若干非确定型（即每次可得到任意数目的路径数）多项式时间的算法，但所有已知的确定性算法都是指数型的。可以证明，NP－完全类中的若干问题在下述意义上彼此等价：如果能找到其中一个问题的一种确定型多项式时间算法，则该算法可应用于所有的问题。

综上所述，求解人工智能问题的一个途径是企图以多项式时间求解 NP－完全类问题。实现这一点可有两种选择：一是寻找平均角度看执行较快的一些算法，即使在最坏的情况下不慢也行；另一是寻找近似算法，使能在可接受的时间限度内获得满意的解答。

分支界限法实际上是第一种选择的实例，对这种策略有如下评述：

（1）各种选择按完美排序进行时，最好情况下其性能有多好。

（2）各种选择按不良排序（从最坏到最好）进行时，最差情况下其性能有多好。结果显然，其搜索节点数同穷举法，此外还必须附加跟踪当前约束及进行无畏的比较工作量。

（3）各种选择按某种随机过程的排序进行时，平均情况下其性能有多好。

（4）各种选择按应用于一组特定问题的启发函数排序进行时，在实际世界中，

平均角度看其性能有多好。通常这个结果优于真正随机世界的平均情况。

总的来说，人们愿意接受良好的平均时间性能，但即使如此也未必能做得满意，就连分支界限法的平均时间也是指数型的（~$1.2e^N$，N 为问题的规模）。因此有时也得牺牲完美解的要求并接受近似的解法。一种进一步改进的分支界限法，用于求解旅行商问题称为最近邻居算法就是求满意解的一个实例，通过分析得到所需的时间与 N^2 成比例。

至于 A 算法既是“寻找平均较快”策略的例子，又是求满意解策略的例子，关键是启发函数 h 的选取问题。至于更深入的讨论可参阅有关文献。

3．数字魔方问题求解的搜索策略

数字魔方游戏已有悠久的历史，是古代数学家、哲学家、神学家、占星家等探索的问题。现在来看一个 1750 年 Benjamin Franklin 编造的一个数字魔方。在 8×8 的棋盘方格上，填入 1－64 的数字，其结果如图 1.19 所示。

52	61	4	13	20	29	36	45
14	3	62	51	46	35	30	19
53	60	5	12	21	28	37	44
11	6	59	54	43	38	27	22
55	58	7	10	23	26	39	42
9	8	57	56	41	40	25	24
50	63	2	15	18	31	34	47
16	1	64	49	48	33	32	17

图 1.19　Benjamin Franklin 数字魔方

其主要特性是每行或每列 8 个数字的总和为 260。再进一步分析发现任一半行或半列 4 个数字的总和为 130；任一拐弯对角线 8 个数字的总和也是 260（如 16，63，57，10，23，40，34，17 或 50，8，7，54，43，26，25，47 两条拐弯对角线）；4 角的 4 个数字(52，45，16，17)与中心的 4 个数字(54，43，10，23)也是 260；任意一个 2×2 的子魔方 4 个数字之和是 130（如 52，61，14，3 或 23，26，41，40），还可给出几个类似的性质。Benjamin Franklin 还研究了魔圆的构造问题，9 个同心圆等分为 8 个扇区，将 12－75 共 64 个数字填入空格中，如图 2.20 所示。其性质是：任一同心圆周格子上的 8 个数字之和加上中心圆的数字（为起始数 12）总和为 360；任一径向

8 个数字之和加上中心圆的数字总和亦为 360（圆周的度数）。更为惊讶的是任一径向半列的 4 个相邻格子数字和加上 6，其总和为 180；由水平直径分割后任一半圆数字和加上 6，其总和仍为 180。由此看出构造这种数字魔方或魔圆并满足若干性质是一个极为复杂的搜索问题。

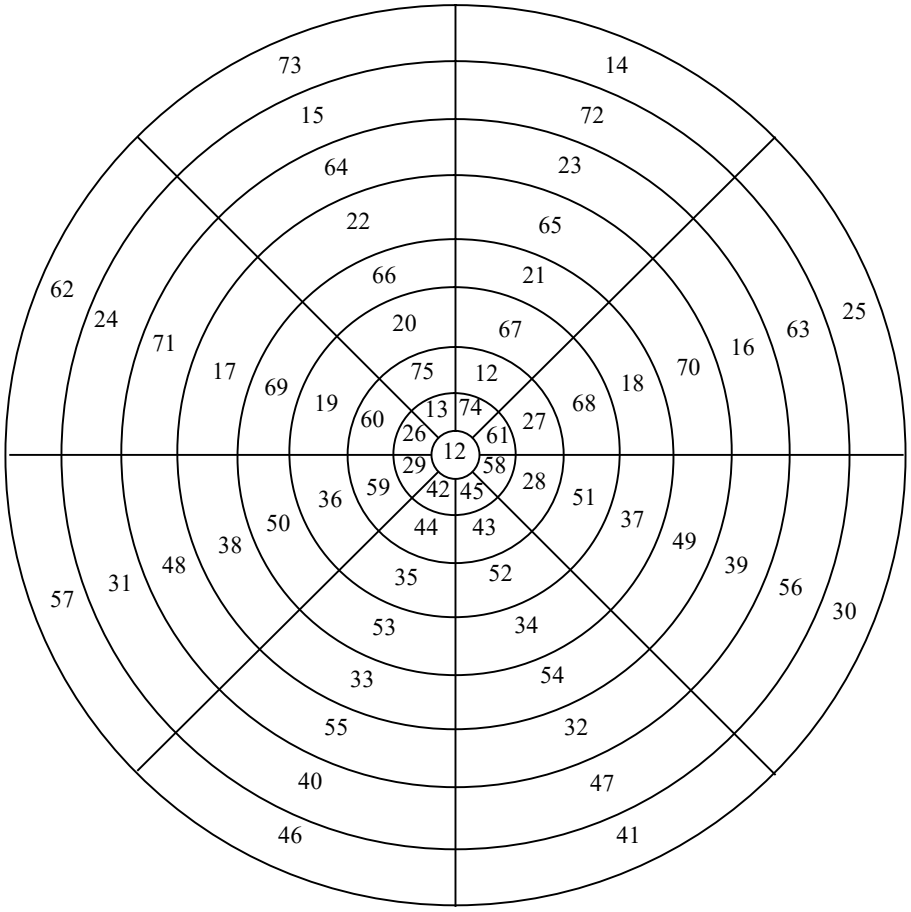


图 2.20　Benjamin Franklin 数字魔园

下面来讨论一个最简单的 3×3 数字魔方问题。将 1－9 共 9 个数码填入魔方格使行、列和对角线数码总和相等。对于奇数阶魔问题，数学家们已构造出一个极为巧妙的算法，不花费任何多余的搜索就可以直接找到问题的解。

这个算法的要点如下：

N 阶（奇次）魔方算法：

①$D:=1,P(D,(x=\frac{\lceil N\rceil}{2},y=N))$；函数 P 把最小数字置于顶行中心处，每一方格用坐标标记，如图 2.21。

②LOOP: D:＝D＋1，x:＝x＋1，y:＝y＋1；

③IF D＝N^2　THEN　EXIT(SUCCESS)；

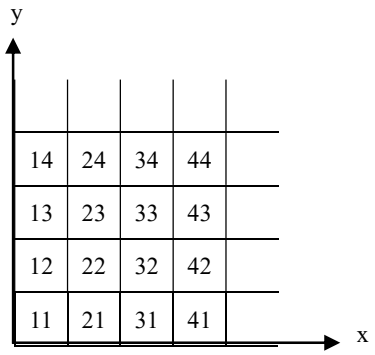


图 1.21 魔方坐标图

④IF (x＜N) ∧ (y＜N) THEN IF (x，y)=NIL
THEN P(D，(x，y))
ELSE P(D，(x-1，y-2))
IF (x＜N) ∧ (y＞N) THEN P(D，(x，y-N))
IF (x＞N) ∧ (y＜N) THEN P(D，(x-N，y))
IF (x＞N) ∧ (y＞N) THEN IF(x-N，y-N)=NIL
THEN P(D，(x-N，y-N))
ELSE P(D，(x-1，y-2))；

⑤GO LOOP；

该算法应用于 3×3 数字魔方问题，其搜索图如图 1.22 所示，从搜索过程看出，构造算法的基本启发信息是把数码等分成 N 组，每一组 N 个数码放置原则是每一个数码必须处在不同行不同列的方格上，这样搭配就可能使行、列和对角线的数字和相接近乃至完全相等。这是最有希望获得目标要求的搜索方向，这样就把许多没有希望的路径删弃，从而大大提高了搜索效果。

进一步分析数字魔方的要求，还可以给出若干有用的启发信息，例如 N（奇次）阶魔方，行、列或对角的总和值$S=N(\frac{1+N^2}{2})$，在$C_{N^2}^{N}$种的数字组合中，只有满足数字总和为 S 的 2N＋2 组排法才可能构成解。其中具有公共元的 4 组数字应排列在中

心行、中心列和对角线上，且公共元的那个数码必定处在中心格位置上，还有 3 组共有的那些数码必定处在四角位置上等等。利用信息引导搜索，可求解任意阶数字魔方问题。

对于任意偶次阶的数字魔方问题，也有简单的求解算法，读者可从程序设计的参考书找到，这里不再介绍。

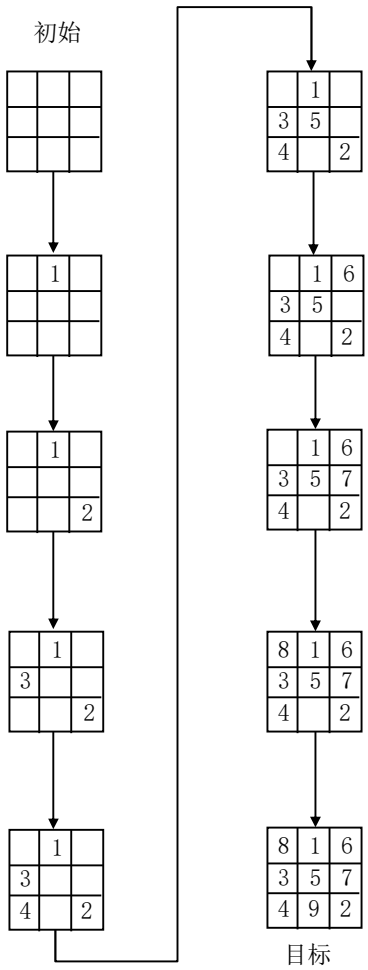


图 1.22 3×3 数字魔方搜索图

4．搜索算法的研究工作

A*算法是 N.J.Nilsson70 年代初的研究成果，是从函数的观点来讨论搜索问题，并在理论上取得若干结果。但 A*算法不能完全克服“指数爆炸”的困难。

70年代末J.Pearl从概率观点研究了启发式估计的精度同A*算法平均复杂性的关系。Pearl 假设如下的概率搜索空间：一个一致的 m-枝树 G，在深度 d 处有一个唯一的目标节点 G_d，其位置事先并不知道。

设估计量 h(n)是在[0，h*(n)]区间中的随机应量，由分布函数 $F_{h(n)}(x)=P[h(n)\leqslant x]$ 来描述；E(Z)表示用 A*算法求到目标 G_d 时所展开的节点平均个数，并称之为 A*的平均复杂性。若 h(n)满足

$$P\left[\frac{h^*(n)-h(n)}{h^*(n)}>\varepsilon\right]>\frac{1}{m},\varepsilon>0$$

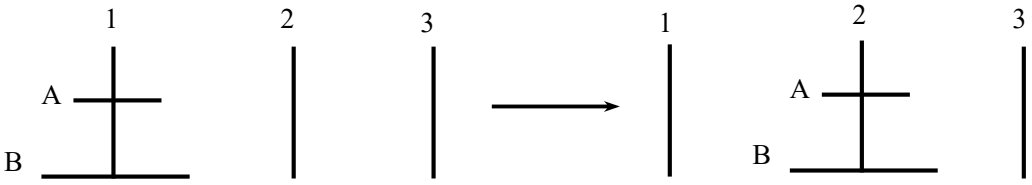
则有 $E(Z)\sim O(e^{cd})$，$c>0$

80 年代初，张钹、张铃提出把启发式搜索看成某种随机取样的过程，从而将统计推断引入启发式搜索。把各种统计推断方法，如序贯概率比检验法（SPRT），均值固定宽度信度区间渐近序贯法（ASM）等，同启发式搜索算法相结合，得到一种称为 SA（统计启发式）的搜索算法。该算法在一定假设条件下，能以概率为一找到目标，且其平均复杂性为 O(dlnd)或 $O(d\ln^2 d)$，而且证明其所需的条件比使 A*搜索为多项复杂性的条件为弱。

总之搜索策略是人工智能研究的核心问题之一，已有许多成熟的结果，并在解决人工智能的有关问题中得到广泛应用。但目前仍有若干深入的问题有待发展，特别是结合实际问题，探索有效实用的策略仍是一个研究和开发的工作，还应当给予足够的重视。

习题

1.1 用深度优先策略求解如下所示二阶梵塔问题，画出搜索过程的状态变化示意图。



对每个状态规定的操作顺序为：先搬 1 柱的盘，放的顺序是先 2 柱后 3 柱；再搬 2 柱的盘，放的顺序是先 3 柱后 1 柱；最后搬 3 柱的盘，放的顺序是先 1 柱后 2 柱。

1.2 滑动积木块游戏的棋盘结构及某一种将牌的初始排列结构如下：

B	B	B	W	W	W	E

其中 B 表示黑色将牌，W 表示白色将牌，E 表示空格。游戏的规定走法是：

（1）任意一个将牌可以移入相邻的空格，规定其耗散值为 1；

（2）任意一个将牌可相隔 1 个或 2 个其他的将牌跳入空格，规定其耗散值等于跳过将牌的数目；游戏要达到的目标是使所有白将牌都处在黑将牌的左边（左边有无空格均可）。对这个问题，定义一个启发函数 h(n)，并给出利用这个启发函数用算法 A 求解时所产生的搜索树。你能否辨别这个 h(n)是否满足下界范围？在你的搜索树中，对所有的节点满足不满足单调限制？

1.3 对 1.4 节中的旅行商问题，定义两个 h 函数（非零），并给出利用这两个启发函数用算法 A 求解 1.4 节中的五城市问题。讨论这两个函数是否都在 h*的下界范围及求解结果。

1.4 2.1 节四皇后问题表述中，设应用每一条规则的耗散值均为 1，试描述这个问题 h*函数的一般特征。你是否认为任何 h 函数对引导搜索都是有用的？

1.5 对 N＝5，k≤3 的 M－C 问题，定义两个 h 函数（非零），并给出用这两个启发函数的 A 算法搜索图。讨论用这两个启发函数求解该问题时是否得到最佳解。

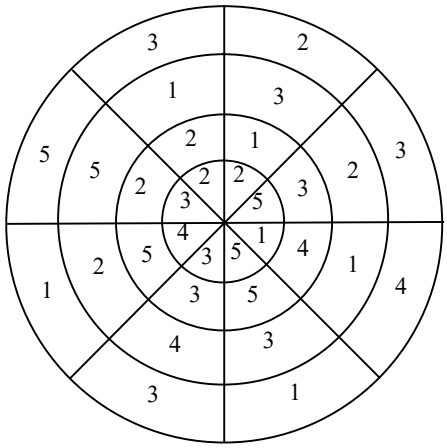
1.6 证明 OPEN 表上具有 f(n)＜f*(s)的任何节点 n，最终都将被 A*选择去扩展。

1.7 如果算法 A*从 OPEN 表中去掉任一节点 n，对 n 有 f(n)＞F（F＞f*(s)），试说明为什么算法 A*仍然是可采纳的。

1.8 用算法 A 逆向求解图 2.7 中的八数码问题，评价函数仍定义为 f(n)=d(n)+w(n)。逆向搜索在什么地方和正向搜索相会。

1.9 讨论一个 h 函数在搜索期间可以得到改善的几种方法。

1.10 四个同心圆盘的扇区数字如图所示，每个圆盘可单独转动。问如何转动圆盘使得八个径向的 4 个数字和均为 12。



1.11 在 3×3 的九宫格内，用 1，2，…，9 的九个数字填入九宫内，使得每行

数字组成的十进制数平方根为整数。

试用启发式搜索算法求解，分析问题空间的规模和有用的启发信息，给出求解的搜索简图。

1.12　一个数码管由七段组成，用七段中某些段的亮与不亮可分别显示 0－9 这十个数字。问能否对这十个数字给出一种排列，使得每相邻两个数字之间的转换，只能是打开几个亮段或关闭几个亮段，而不能同时有打开的亮段，又有关闭的亮段。试用产生式系统求解该问题。