第3章

搜索求解

3.1 搜索概述

搜索是人工智能的一个基本问题，是一种求解问题的一般方法，也是人工智能的一个主要应用领域。

在求解一个问题时，一般涉及两方面：一是如何表示问题，即将问题用合适的方法描述出来；二是如何解决问题，即找到一种可行的问题求解方法。求解问题的基本方法包含搜索法、归纳法、归结法、推理法、产生式法等多种方法。由于大多数需要采用人工智能方法求解的问题缺乏直接的解法，因此，搜索法可以作为问题求解的一般方法。在人工智能领域中，搜索法被广泛应用于下棋等游戏软件中。

本章首先讨论搜索的基本概念，然后着重介绍状态空间知识表示和搜索策略，主要有宽度优先搜索、深度优先搜索等盲目的图搜索策略、A 及 A* 搜索算法等启发式图搜索策略，以及 Alpha-Beta 剪枝算法和蒙特卡罗搜索算法。

3.1.1 搜索的基本问题与主要过程

1. 什么是搜索

人工智能所研究的对象大多属于结构不良或者非结构化的问题，一般很难获得问题的全部信息，也没有现成求解算法使用，只能依靠经验，利用已有知识逐步摸索求解。像这种根据问题的实际情况，不断寻找可利用知识，从而构造一条代价最小的推理路线，使问题得以解决的过程称为搜索。

2. 搜索中需要解决的基本问题

（1）是否一定能找到一个解。

（2）找到的解是否是最佳解。

（3）时间与空间复杂性如何。

（4）是否终止运行或是否会陷入一个死循环。

3. 搜索的主要过程

（1）从初始或目的状态出发，并将它作为当前状态。

（2）扫描操作算子集，将适用当前状态的一些操作算子作用于当前状态而得到新的状态，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，则得到问题的一个

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。

3.1.2　搜索算法分类

搜索算法可根据其是否采用智能方法分为盲目搜索算法和启发式搜索算法。

1. 盲目搜索

指在搜索之前就预定好控制策略，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(依次或随机调用操作算子)进行的搜索。因整个搜索过程中的策略不再改变，盲目搜索算法的灵活性较差，搜索效率较低，不便于复杂问题的求解。

2. 启发式搜索

指可以利用搜索过程得到的中间信息来引导搜索过程向最优方向发展的算法。算法动态地确定调用操作算子的步骤，优先选择较适合的操作算子，尽量减少不必要的搜索，以求尽快地到达结束状态。

3.2　状态空间表示法

状态空间的搜索是人工智能中最基本的求解问题方法，它采用状态空间表示法来表示要求解的问题。状态空间搜索的基本思想是利用“状态”和“操作算子”来表示和求解问题。

3.2.1　状态空间表示的基本概念

首先介绍状态空间表示法，它主要包含以下三个概念。

1. 状态

状态是指问题在任意确定时刻的状况，它用来表征问题的特征、结构等属性。状态一般以一组变量或数组进行表示。在状态空间图中，状态表示为结点。在程序中，状态可以用字符、数字、记录、数组、结构、对象等进行表示。

2. 操作

操作是能够使问题状态发生改变的某种规则、行为、变换、关系、函数、算子、过程等，也被称为状态转换规则。在状态空间图中，操作表示为边。在程序中，操作可以用数据对条件语句、规则、函数、过程等进行表示。

3. 状态空间

状态空间是由一个问题的全部状态，以及这些状态之间的相互关系所构成的集合，可用一个三元组表示：

$$(S,F,G)$$

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

下面以一个具体的例子描述状态空间表示法。

例 3.1　迷宫问题。走迷宫是人们熟悉的一种游戏，图 3.1 是一个迷宫，目标是从迷宫左侧的入口出发，找到一条到达右侧出口的路径。

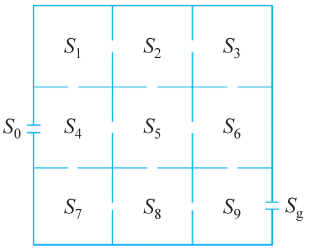

图 3.1　迷宫问题

解：以每个格子作为一个状态，并用其标识符表示。那么两个标识符的序对就是一个状态转换规则，即操作。于是迷宫的状态空间表示为

S：S_0

F：$\{(S_0,S_4),(S_4,S_0),(S_4,S_1),(S_1,S_4),(S_1,S_2),(S_2,S_1),(S_2,S_3),(S_3,S_2),(S_4,S_7),(S_7,S_4),(S_4,S_5),(S_5,S_4),(S_5,S_6),(S_6,S_5),(S_5,S_8),(S_8,S_5),(S_8,S_9),(S_9,S_8),(S_9,S_g)\}$

G：S_g

3.2.2 状态空间的图描述

状态空间可以用有向图来描述，图的结点表示问题的状态，图的弧表示从一个状态转换为另一个状态的操作，状态空间图可以描述问题求解的步骤。初始状态对应于实际问题的已知信息，是图中的根结点。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状态空间中寻找某一路径。

图 3.2 描述了一个有向图表示的状态空间。其中，初始状态为 S_0，针对 S_0 允许使用操作 F_1、F_2 和 F_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转换下去，可以得到目标状态，如 $S_{10}\in G$，则路径 F_2、F_6、F_{10} 就是一个解。

仍以例 3.1 中的迷宫问题为例，如果把迷宫的每个空间和出入口作为一个结点，把通道作为边，则迷宫可以由一个有向图表示，如图 3.3 所示，那么走迷宫其实就是从该有向图的初始结点（入口）出发，寻找目标结点（出口）的问题，或者是寻找通向目标结点的路径的问题。

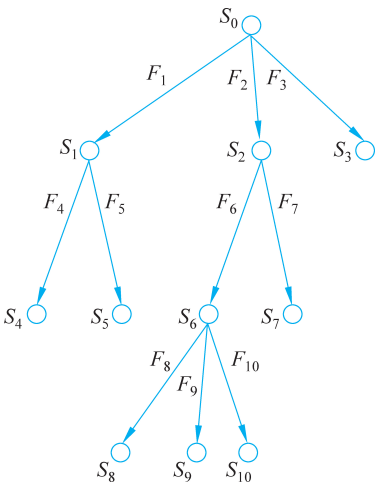

图 3.2 状态空间的有向图描述

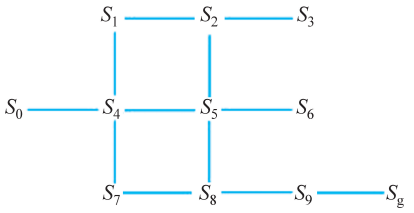

图 3.3 迷宫的有向图表示

可以看出，例 3.1 中的状态转换规则（操作）是迷宫的任意两个格子间的通道，也就是对应状态图中的任一条边，而这个规则正好描述了图中的所有的结点和边。类似于这样罗列出全部的结点和边的状态图称为显示状态图，或者说状态图的显示表示。

3.3　盲目搜索

3.3.1　盲目搜索概述

针对一些通用性较强、较为简单的问题，往往采用盲目搜索策略解决。盲目搜索策略又称非启发式搜索，是一种无信息搜索。一般来说，盲目搜索是按预定的搜索策略进行搜索，而没有利用与问题有关的有利于找到问题解的信息或知识。本节主要介绍两种盲目搜索算法：深度优先搜索和宽度优先搜索。

3.3.2　深度优先搜索算法

在深度优先搜索(Depth First Search，DFS)中，当分析一个结点时，在分析它的任何“兄弟”结点之前分析它的所有“后代”，如图 3.4 所示。深度优先搜索尽可能地向搜索空间的更深层前进，如图 3.4 所示的深度优先搜索顺序为 A、B、D、E、C、F、G。

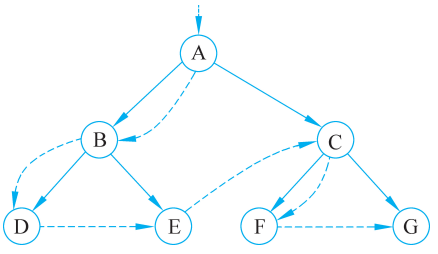

图 3.4　深度优先搜索排序

首先介绍一下扩展的概念。所谓扩展，就是用合适的算符对某个结点进行操作，生成一组后继结点，扩展过程实际上就是求后继结点的过程。所以，对于状态空间图中的某个结点，如果求出了它的后继结点，则此结点为已扩展的结点，而尚未求出后继结点的结点称为未扩展结点。在实际搜索过程中，为了保存状态空间搜索的轨迹，引入了两张表：open 表和 closed 表。open 表保存了未扩展的结点，open 表中的结点排列次序就是搜索次序。closed 表用于存放将要扩展或者已经扩展的结点，它是一个搜索记录器，保存了当前搜索图上的结点。open 表和 closed 表的数据结构分别如表 3.1 和表 3.2 所示。

表 3.1　open 表的结构

状 态 结 点	父　结　点

表 3.2　closed 表的结构

编　　号	状 态 结 点	父　结　点

对于许多问题，深度优先搜索状态空间树的深度可能为无限深，为了避免算法向空间深入时“迷失”(防止搜索过程沿着无用的路径扩展下去)，往往给出一个结点扩展的最大深度——深度界限。

含有深度界限的深度优先搜索算法如下。

(1) 建立一个只含初始结点 S_0 的搜索图 G，把 S_0 放入 open 表。

(2) 如果 open 表是空的，则搜索失败。

(3) 从 open 表中取出第一个结点，置于 closed 表中，给这个结点编号为 n。

(4) 如果 n 是目标结点，则得解，算法成功退出。解路径可从目标结点开始直到初始结点的返回指针中得到。

(5) 扩展结点 n。如果它没有后继结点，则转到步骤(2)；否则，生成 n 的所有后继结点集 $M=\{m_i\}$，把 m_i 作为 n 的后续结点添入 G。为 m_i 添加一个返回到 n 的指针，并把它们放入 open 表的前端。

(6) 返回第(2)步。

注意： 在深度优先搜索中，open 表是一个堆栈结构，即先进后出(FILO)的数据结构。open 表用堆栈实现的方法使得搜索偏向于最后生成的状态。

整个搜索过程产生的结点和指针构成一棵隐式定义的状态空间树的子树，称为搜索树。下面以八数码问题为例，描述搜索树的构建过程。

例 3.2 八数码问题的状态空间：在一个 3×3 的方棋盘上放置着 1,2,3,4,5,6,7,8 八个数码，每个数码占一格，且有一个空格。这些数码可以在棋盘上移动，其移动规则是：与空格相邻的数码方格可以移入空格。需要找到一个移动序列使初始排列布局变为指定的目标排列布局，如图 3.5 所示。

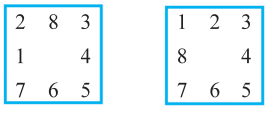

图 3.5 八数码难题

解： 图 3.6 绘出了把深度优先搜索应用于八数码难题的搜索树。搜索树上每个结点旁边的数字表示结点扩展的先后顺序，设置深度优先搜索深度为 5，空格的移动顺序为左、上、右、下。图 3.6 中加粗实线表示的路径是宽度优先搜索得到的解的路径。

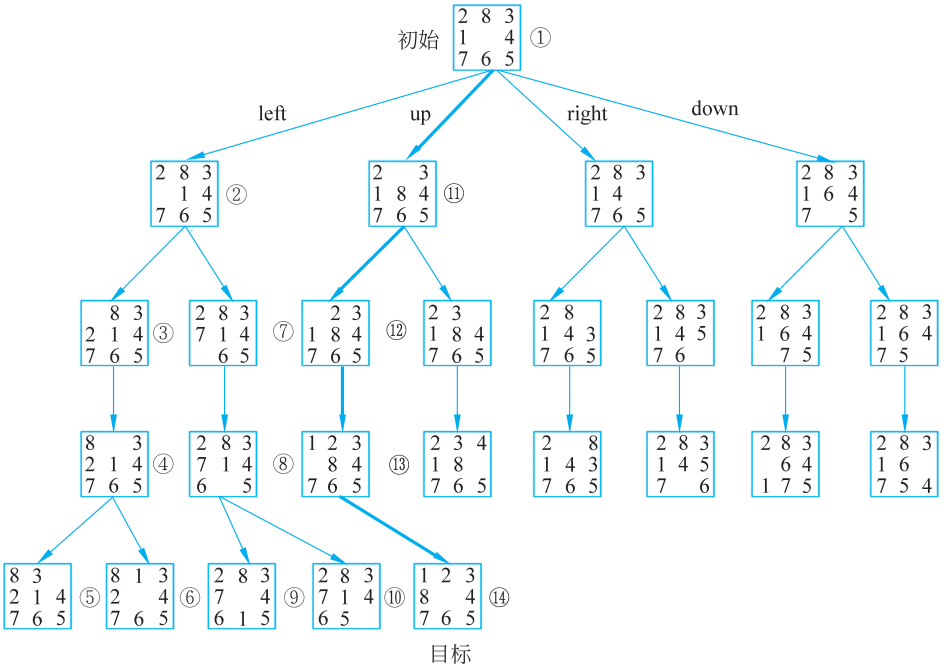

图 3.6 八数码难题的深度优先搜索树

3.3.3 宽度优先搜索算法

如果搜索是以接近起始结点的程度依次扩展结点的，那么这种搜索就叫作宽度优先搜索(Breadth First Search，BFS)，如图 3.7 所示。宽度优先搜索是逐层进行的，在对下一层的任意结点

搜索之前，必须完成本层的所有结点。如图 3.7 所示的宽度优先搜索顺序为 A、B、C、D、E、F、G。

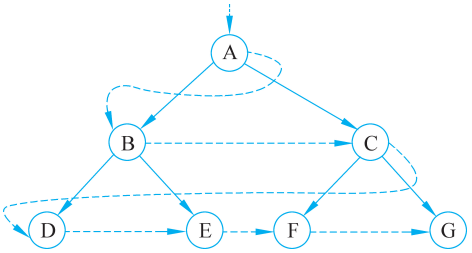

图 3.7　宽度优先搜索排序

宽度优先搜索算法如下。

(1) 建立一个只含初始结点 S_0 的搜索图 G，把 S_0 放入 open 表。

(2) 如果 open 表是空的，则搜索失败。

(3) 从 open 表中取出第一个结点，置于 closed 表中，给这个结点编号为 n。

(4) 如果 n 是目标结点，则得解，算法成功退出。解路径可从目标结点开始直到初始结点的返回指针中得到。

(5) 扩展结点 n。如果它没有后继结点，则转到步骤(2)；否则，生成 n 的所有后继结点集 $M=\{m_i\}$，把 m_i 作为 n 的后续结点添入 G。为 m_i 添加一个返回到 n 的指针，并把它们放入 open 表的末端。

(6) 返回第(2)步。

注意： 宽度优先搜索中，open 表示一个队列结构，即先进先出(FIFO)的数据结构。

显然，宽度优先搜索算法能够保证在搜索树中找到一条通往目标结点的最短路径，图 3.8 绘出了宽度优先搜索应用于八数码难题的搜索树，包含所有存在的路径。

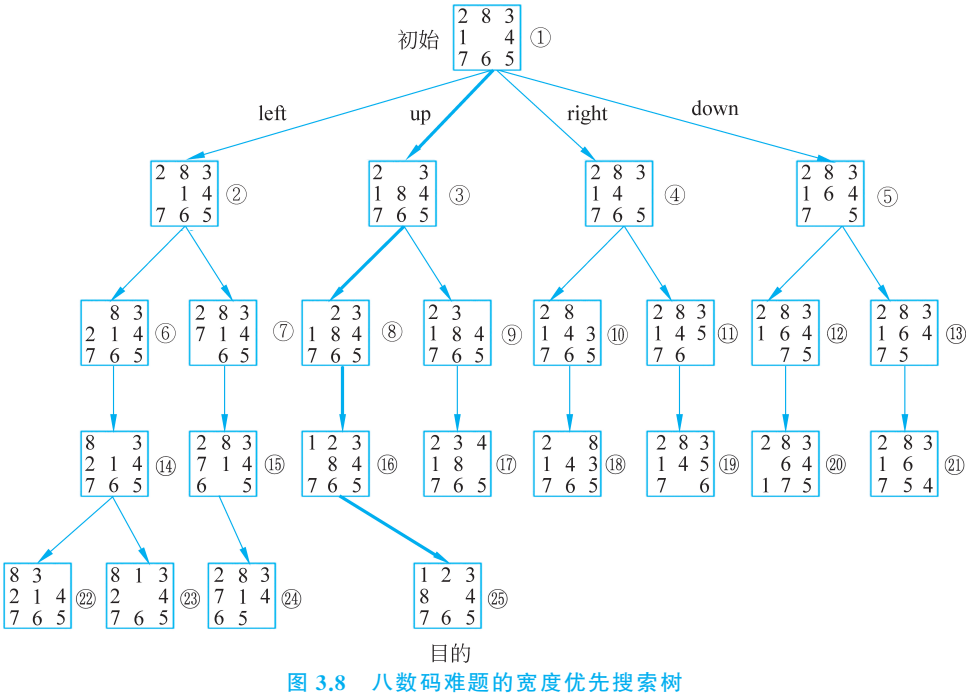

图 3.8　八数码难题的宽度优先搜索树

3.3.4 盲目搜索算法的 Python 实现

1. 深度优先算法 Python 实现

在这个例子里，使用邻接表存储了一个图，然后定义了一个深度优先搜索函数。在函数中，首先将起始结点加入到已访问的集合中，并打印出结点的值。然后对于起始结点的每个邻居结点，如果该结点没有被访问过，则递归调用深度优先搜索函数，并将该结点加入到已访问的集合中。最后，调用深度优先搜索函数，并将起始结点设置为'A'，即从结点'A'开始深度优先搜索整个图，代码如下。

```
#定义一个邻接表存储图
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
#定义深度优先搜索函数
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next_node in graph[start]:
        if next_node not in visited:
            dfs(graph, next_node, visited)
#调用深度优先搜索函数
dfs(graph, 'A')
```

2. 宽度优先算法 Python 实现

在这个例子里，同样使用邻接表存储了一个图，然后定义了一个宽度优先搜索函数。在函数中，首先将起始结点加入到队列中，并将其标记为已访问。然后，进入一个循环，不断从队列中取出队首元素，并打印出结点的值。接着，将该结点的所有未被访问过的邻居结点加入到队列中，并将它们标记为已访问。这个过程将一直持续，直到队列为空。最后，调用宽度优先搜索函数，并将起始结点设置为'A'，即从结点'A'开始宽度优先搜索整个图，代码如下。

```
#定义一个邻接表存储图
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```

```
#定义宽度优先搜索函数
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            print(node)
            queue.extend(graph[node] - visited)
#调用宽度优先搜索函数
bfs(graph, 'A')
```

3.4　启发式搜索

3.4.1　启发式搜索概述

盲目搜索方法需要产生大量的结点才能找到解路径，所以其搜索的复杂性往往是很高的。如果能找到一种搜索算法，充分利用待求解问题自身的某些特性信息，来指导搜索朝着最有利于问题求解的方向发展，即在选择结点进行扩展时，选择那些最有希望的结点加以扩展，那么搜索效率会大大提高。这种利用问题自身特性信息来提高搜索效率的搜索策略，称为启发式搜索。本节首先介绍启发式搜索策略及其所涉及的概念、启发信息、估价函数，然后具体介绍启发式图搜索算法——A 及 A^* 算法。

3.4.2　启发信息和估价函数

在搜索过程中，关键的一步是确定如何选择下一个要被考察的结点，不同的选择方法就是不同的搜索策略。如果在确定要被考察的结点时，能够利用被求解问题的有关特性信息，估计出各结点的重要性，那么就可以选择重要性较高的结点进行扩展，以便提高求解的效率。像这样的可用于指导搜索过程且与具体问题求解有关的控制性信息称为启发信息。

启发信息按作用不同可分为以下三种。

(1) 用于扩展结点的选择，即用于决定应先扩展哪一个结点，以免盲目扩展。

(2) 用于生成结点的选择，即用于决定要生成哪一个或哪几个后继结点，以免盲目生成过多无用的结点。

(3) 用于删除结点的选择，即用于决定删除哪些无用结点，以免造成进一步的时空浪费。

为提高搜索效率就需要利用上述三种启发信息作为搜索的辅助性策略，在搜索过程中需要根据这些启发信息估计各个结点的重要性。本节所描述的启发信息属于第一种启发信息，即决定哪个结点是下一步要扩展的结点，把这一结点称为“最有希望”的结点。那么如何来度量结点的“希望”程度呢？通常可以构造一个函数来度量，称这种函数为估价函数。

估价函数用于估算待搜索结点“希望”程度，并依次给它们排定次序。因此，估价函数 $f(n)$ 定义为从初始结点经过结点 n 到达目标结点的路径的最小代价估计值，其一般形式是

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

其中，$g(n)$是从初始结点到结点 n 的实际代价，而 $h(n)$是从结点 n 到目标结点的最佳路径估计代价，称为启发函数。

$g(n)$的作用一般是不可忽略的，因为它代表了从初始结点经过结点 n 到达目标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，$g(n)$的比重越大，越倾向于宽度优先搜索方式。而 $h(n)$的比重越大，则表示启发性越强。

3.4.3 A 算法

启发式搜索是在搜索路径的控制信息中增加关于被求解问题的相关特征，从而指导搜索向最有希望到达目标结点的方向前进，提高搜索效率。在实际问题求解中，需要用启发信息引导搜索，从而减少搜索量。启发式策略及算法设计一直是人工智能的核心问题之一。它的基本特点是如何寻找并设计一个与问题有关的启发式函数 $h(n)$及构造相应的估价函数 $f(n)$。有了 $f(n)$就可以按照 $f(n)$的大小来安排带扩展结点的次序，选择 $f(n)$最小的结点先进行扩展。

启发式图搜索法使用两张表记录状态信息：在 open 表中保留所有未扩展的结点；在 closed 表中记录已扩展的结点。进入 open 表的状态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。

A 算法是基于估价函数的一种加权启发式图搜索算法，具体步骤如下。

(1) 建立一个只含初始结点 S_0 的搜索图 G，把 S_0 放入 open 表，并计算 $f(S_0)$的值。

(2) 如果 open 表是空的，则搜索失败。

(3) 从 open 表中取出 f 值最小的结点(第一个结点)，置于 closed 表中，给这个结点编号为 n。

(4) 如果 n 是目标结点，则得解，算法成功退出。解路径可从目标结点开始直到初始结点的返回指针中得到。

(5) 扩展结点 n。如果它没有后继结点，则转到步骤(2)；否则，生成 n 的所有后继结点集 $M=\{m_i\}$，把 m_i 作为 n 的后续结点添入 G，并计算 $f(m_i)$。

(6) 若 m_i 未曾在 G 中出现过，即未曾在 open 表或 closed 表中出现过，就将它配上刚计算过的 $f(m_i)$值并添加一个返回到 n 的指针，把它们放入 open 表中。

(7) 若 m_i 已在 open 表中，则需要把原来的 g 值与现在刚计算过的 g 值相比较：若前者不大于后者，则不做任何修改；若前者大于后者，则将 open 表中该结点的 f 值更改为刚计算的 f 值，返回指针更改为 n。

(8) 若 m_i 已在 closed 表中，但 $g(m_i)$小于原先的 g 值，则将表中该结点的 g、f 值及返回指针进行类似第(7)步的修改，并要考虑修改表中通向该结点的后继结点的 g、f 值及返回指针。

(9) 按 f 值自小至大的次序，对 open 表中的结点重新排序。

(10) 返回第(2)步。

例 3.3 用 A 算法求解八数码难题。

解： 图 3.9 给出了利用 A 算法求解八数码难题的搜索树，解的路径为 S(4)→B(4)→E(5)→I(5)→K(5)→L(5)。图 3.9 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

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

式中，$g(n)$代表状态的深度，每步为单位代价；$w(n)$表示以“不在位”的数码作为启发信息的度量。例如，B 的状态深度为 1，不在位的数码数为 3，所以 B 的启发函数值为 4，记为 B(4)，搜索过程如图 3.9 所示。

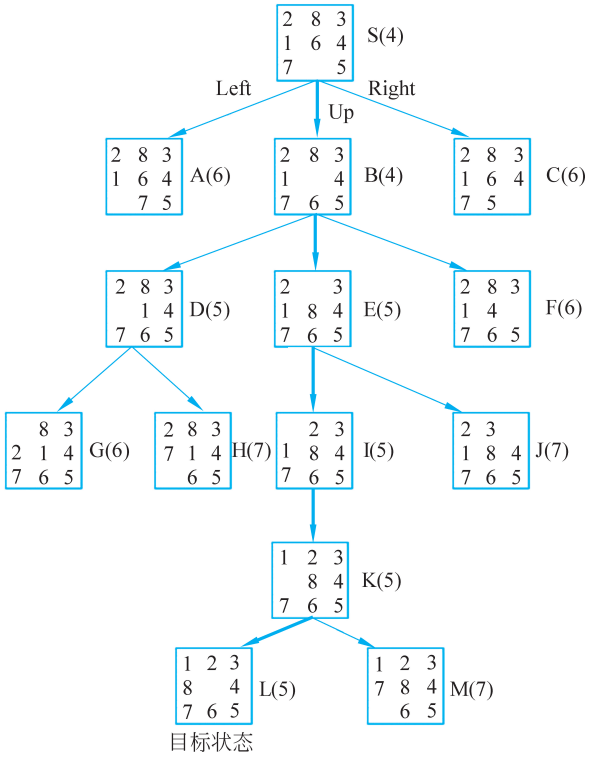

图 3.9　八数码难题的 A 算法搜索树

搜索过程中 open 表和 closed 表内状态排列的变化情况如表 3.3 所示。

表 3.3　open 表和 closed 表状态排列的变化表

Open 表	Closed 表
初始化：(S(4))	(　)
一次循环后： (B(4)，A(6)，C(6))	(S(4))
二次循环后： (D(5)，E(5)，A(6)，C(6)，F(6))	(S(4)，B(4))
三次循环后： (E(5)，A(6)，C(6)，F(6)，G(6)，H(7))	(S(4)，B(4)，D(5))
四次循环后： (I(5)，A(6)，C(6)，F(6)，G(6)，H(7)，J(7))	(S(4)，B(4)，D(5)，E(5))
五次循环后： (K(5)，A(6)，C(6)，F(6)，G(6)，H(7)，J(7))	(S(4)，B(4)，D(5)，E(5)，I(5))

续表

Open 表	Closed 表
六次循环后： (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))
七次循环后： L 为目的状态，则成功退出，结束搜索	(S(4)，B(4)，D(5)，E(5)，I(5)，K(5)，L(5))

3.4.4 A* 搜索算法

A* 搜索算法是由著名的人工智能学者 Nilsson 提出的，它是目前最有影响的启发式图搜索算法，也称为最佳图搜索算法。

定义 $h^*(n)$为状态 n 到目的状态的最优路径的代价，对所有结点 n，当 A 搜索算法的启发函数 $h(n)$小于或等于 $h^*(n)$，即满足 $h(n)\leqslant h^*(n)$时，称为 A* 搜索算法。

如果某一问题有解，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解。

A* 搜索算法有以下三点特性。

1. 可采纳性

如果一个搜索算法对于任何具有解路径的图都能找到一条最佳路径，则称此算法为可采纳的。

定义最优估价函数为

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

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

尽管在大部分实际问题中并不存在 $f^*(n)$这样的先验函数，但可以将 $f(n)$作为 $f^*(n)$的近似值函数。在 A 及 A* 搜索算法中，$g(n)$作为 $g^*(n)$的近似代价，则 $g(n)\geqslant g^*(n)$，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，可以使用 $h(n)$代替 $h^*(n)$作为 n 状态到目的状态的最小代价估计值。如果 A 搜索算法所使用的估价函数 $f(n)$能达到 $f(n)$中的 $h(n)\leqslant h^*(n)$时，则称为 A* 搜索算法。

可以证明，所有的 A* 搜索算法都是可采纳的。

2. 单调性

如果启发函数 h 对任何结点 n_i 和 n_j，只要 n_j 是 n_i 的后继，都有 $h(n_i)-h(n_j)\leqslant c(n_i,n_j)$，其中，$c(n_i,n_j)$是从 n_i 到 n_j 的实际代价，且 $h(t)=0$（t 是目标结点），则称启发函数 h 是单调的。

搜索算法的单调性：在整个搜索空间都是局部可采纳的。一个状态和任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。A* 搜索算法中采用单调性启发函数，可以减少比较代价和调整路径的工作量，从而减少搜索代价。

3. 信息性

在两个 A* 启发策略的 h_1 和 h_2 中，如果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。

如果某一搜索策略的 $h(n)$ 越大，则 A* 算法搜索的信息性越多，所搜索的状态越少，但更多的信息性需要更多的计算时间，可能抵消减少搜索空间所带来的益处。

3.4.5　A* 算法的 Python 实现

下面是使用 Python 实现 A* 算法的一个实例，基本步骤如下。

(1) 定义地图：定义一个地图通常可以使用二维列表来表示。例如，0 代表可以通过的空地，1 代表墙壁或障碍物等。

(2) 定义起点和终点：需要指定起点和终点的坐标。

(3) 定义结点：需要定义一个结点类，包含结点的坐标、父结点、g 值、h 值、f 值等信息。

(4) 实现 A* 算法：根据算法流程，首先从起点开始，计算周围结点的 f 值，并将其加入开放列表中。从开放列表中选取 f 值最小的结点，计算周围结点的 f 值，并将其加入开放列表中。直到找到终点或者开放列表为空。

Python 代码实现：

```
import heapq

class Node:
    def __init__(self, x, y, parent=None):
        self.x = x
        self.y = y
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0

    def __lt__(self, other):
        return self.f < other.f

def astar(start, end, grid):
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, start)

    while open_list:
        current = heapq.heappop(open_list)
        if current.x == end.x and current.y == end.y:
            path = []
            while current:
                path.append((current.x, current.y))
                current = current.parent
            return path[::-1]

        closed_list.add(current)

        for i, j in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
```

```
            x = current.x + i
            y = current.y + j

            if x < 0 or y < 0 or x >= len(grid) or y >= len(grid[0]):
                continue

            if grid[x][y] == 1:
                continue

            neighbor = Node(x, y, current)
            neighbor.g = current.g + 1
            neighbor.h = abs(x - end.x) + abs(y - end.y)
            neighbor.f = neighbor.g + neighbor.h

            if neighbor in closed_list:
                continue

            if neighbor not in open_list:
                heapq.heappush(open_list, neighbor)

    return None
    #创建一个示例的地图
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0]
    ]

    #设置起始结点和目标结点
    start = Node(0, 0)
    end = Node(4, 4)

    #执行 A* 搜索
    path = astar(start, end, grid)
    print("A* 搜索结果:", path)
```

3.5 对抗搜索

3.5.1 博弈概述

博弈是一类富有智能行为的竞争活动，如下棋、打牌、摔跤等。博弈可分为双人完备信息博弈和机遇性博弈。双人完备信息博弈就是两位选手对垒，轮流走步，每方不仅知道对方已经走过的棋步，还能估计出对方未来的走步。对弈的结果是一方赢，另一方输或者双方和局。这类博弈的实例有象棋、围棋等。机遇性博弈是指存在不可预测性的博弈，如掷币等。由于机遇性博弈不具备完备信息，因此不讨论。本节主要讨论双人完备信息博弈问题。

在双人完备信息博弈过程中，双方都希望自己能够获胜，因此当任何一方走步时，都是选择对自己最有利的而对另一方最不利的行动方案。假设博弈的一方为 MAX，另一方为 MIN。在博弈过程的每步，可供 MAX 和 MIN 选择的行动方案都可能有多种，从 MAX 方的观点看，可供自己选择的那些行动方案之间是“或”的关系，原因是主动权掌握在 MAX 手里，选择哪个方案完全是自己决定的；而那些可供对方选择的行动方案之间是“与”的关系，原因是主动权掌握在 MIN 的手里，任何一个方案都有可能被 MIN 选中，MAX 必须防止那种对自己最不利的情况发生。

若把双人完备信息博弈过程用图表示出来，就可以得到一棵与/或树，这种与/或树被称为博弈树。在博弈树中，下一步该 MAX 走步的结点称为 MAX 结点，而下一步该 MIN 走步的结点称为 MIN 结点。博弈树具有如下特点。

(1) 博弈的初始状态是初始结点。

(2) 博弈树中的 MAX 结点和 MIN 结点是逐层交替出现的。

3.5.2　极大极小过程

简单的博弈问题可以生成整个博弈树，找到必胜的策略。但复杂的博弈，如国际象棋，大约有 10^{120} 个结点，要生成整个搜索树是不可能的，一种可行的方法是用当前正在考察的结点生成一棵部分博弈树，由于该博弈树的叶结点一般不是哪一方的获胜结点，因此需要利用估价函数 $f(n)$ 对叶结点进行静态估值。一般来说，那些对 MAX 有利的结点，其估价函数取正值；那些对 MIN 有利的结点，其估价函数取负值；那些使双方利益均等的结点，其估价函数取接近于 0 的值。

为了计算非叶结点的值，必须从叶结点向上倒推。由于 MAX 方总是选择估值最大的走步，因此，MAX 结点的收益应该取其后继结点估值的最大值。由于 MIN 方总是选择使估值最小的走步，因此 MIN 结点的收益应取其后继结点估值的最小值。这样一步一步地计算收益，直至求出初始结点的收益为止。由于我们是站在 MAX 的立场上，因此应选择具有最大收益的走步，这一过程称为极大极小过程。

下面给出一个极大极小过程的例子。

例 3.4　一字棋游戏。设有一个三行三列的棋盘，如图 3.10 所示，两个棋手轮流走，每个棋手走步时往空格上摆一个自己的棋子，谁先使自己的棋子成三子一线为赢。设 MAX 方的棋子用×标记，MIN 方的棋子用○标记，并规定 MAX 方先走步。

解：为了对叶结点进行静态估值，规定估价函数 $e(P)$ 如下。

若 P 是 MAX 的必胜局，则 $e(P)=+\infty$。

若 P 是 MIN 的必胜局，则 $e(P)=-\infty$。

若 P 对 MAX、MIN 都是胜负未定局，则是 $e(P)=e(+P)-e(-P)$。式中，$e(+P)$ 表示棋局 P 上有可能使×成三子一线的数目，$e(-P)$ 表示棋局 P 上有可能使○成三子一线的数目。例如，对如图 3.11 所示的棋局有估价函数值 $e(P)=6-4=2$。

在搜索过程中，具有对称性的棋局认为是同一棋局。例如，如图 3.12 所示的棋局可以认为是同一个棋局，这样可以大大减少搜索空间。图 3.13 给出了第一招走棋以后生成的博弈树。叶结点下面的数字是该结点的静态估值，非叶结点旁边的数字是计算出的收益。可以看出，对 MAX 来说，S_3 是一招最好的走棋，它具有较大的收益。

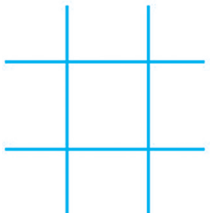

图 3.10 一字棋棋盘

图 3.11 棋局 1

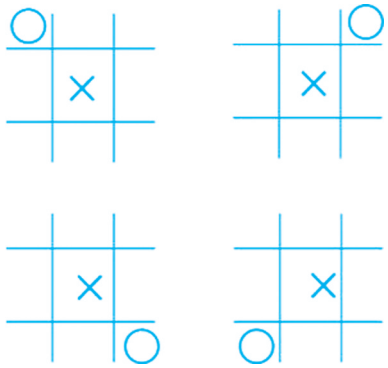

图 3.12 对称棋局的棋子

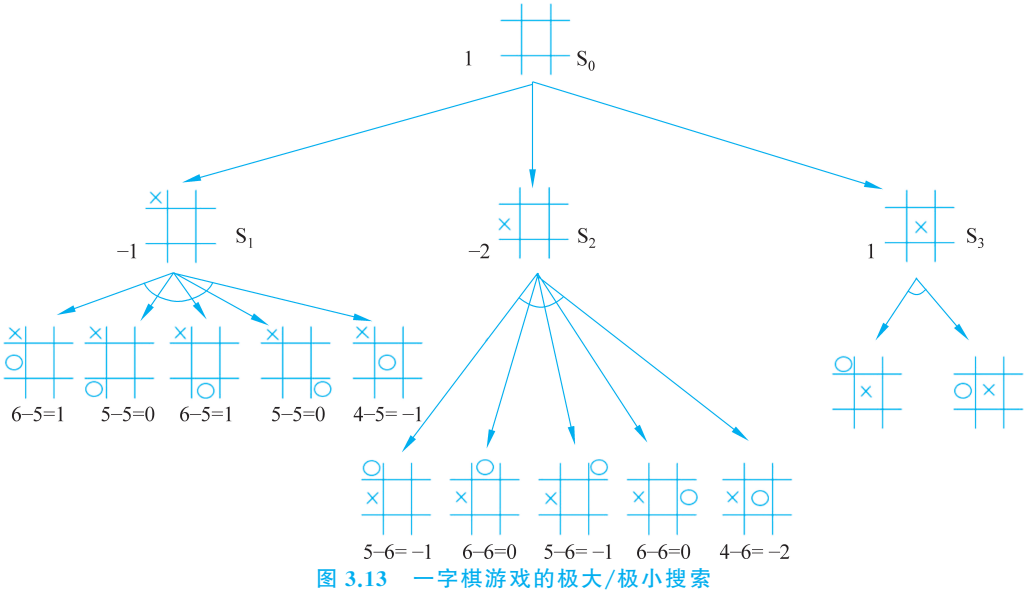

图 3.13 一字棋游戏的极大/极小搜索

3.5.3 Alpha-Beta 剪枝

上述极大/极小过程是先生成与/或树，再计算各结点的布置，这种生成结点和计算估值相分离的搜索方式，需要生成规定深度内的所有结点，因此搜索效率较低。如果能边生成结点边对结点估值，可以剪去一些没用的分支，这种方法称为 Alpha-Beta 剪枝，简写成 α-β 剪枝。通过这种剪枝方法，可以大幅提高搜索效率。

1. α-β 剪枝的方法

（1）MAX 结点的 α 值为当前子结点的最大收益。

（2）MIN 结点的 β 值为当前子结点的最小收益。

2. α-β 剪枝的规则

（1）任何 MAX 结点 n 的 α 值大于或等于它前辈结点的 β 值，则 n 以下的分支可停止搜索，并令结点 n 的收益为 α。这种剪枝称为 β 剪枝。

（2）任何 MIN 结点 n 的 β 值小于或等于它的前辈结点的 α 值，则 n 以下的分支可停止搜索，并令结点 n 的收益为 β。这种剪枝称为 α 剪枝。

假设一棵完整的最小最大搜索树如图 3.14 所示，图 3.15 给出了每个结点 α 值和 β 值的详细变化过程。

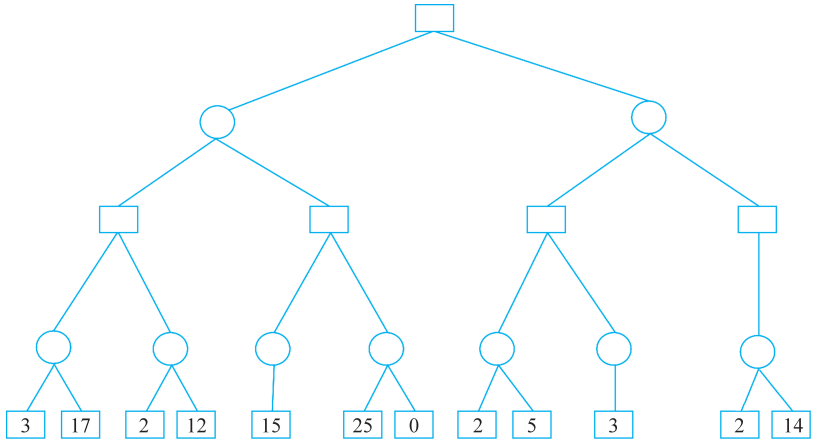

图 3.14　一棵完整的最小最大搜索树

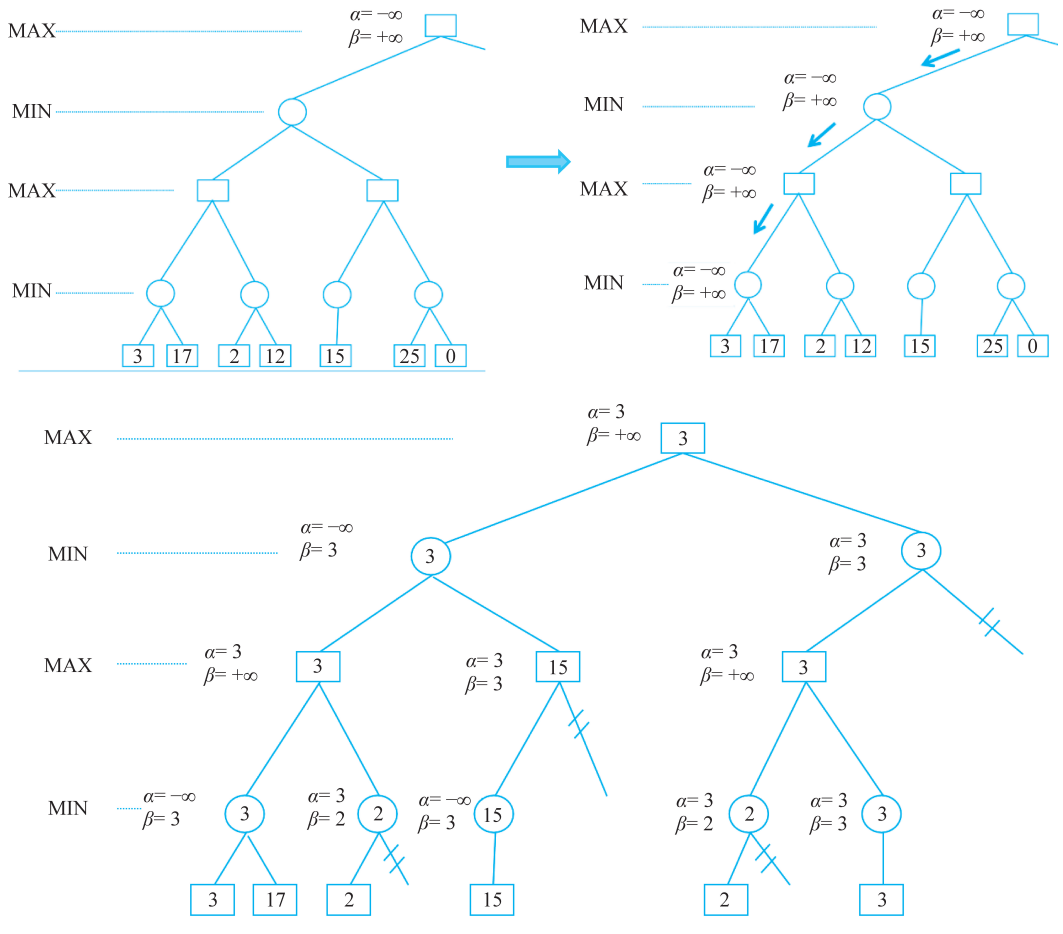

图 3.15　对图 3.14 中搜索树的 α-β 剪枝

图 3.15 中每幅子图对应了扩展一个终局状态或进行剪枝后的搜索树状态，结点上数字表示该结点当前的收益值，结点旁标记了该结点的 α 值和 β 值。原本最小最大搜索树中有 26 个结点，经过 α-β 剪枝后只扩展了其中 17 个结点，可见 α-β 剪枝算法能够有效地减少搜索树中的结点的数目，从而提高了算法效率。

3.5.4 对抗搜索算法的 Python 实现

1. 极大极小算法 Python 实现

假设玩一个简单的决策树游戏，我们和对手轮流选择 1～10 中的一个数字，直到总和达到或超过 100 为止。我们的目标是尽可能使总和接近 100，而对手的目标是尽可能使总和远离 100，代码如下。

```
def play_turn(current_sum, is_our_turn):
    if is_our_turn:
        choice = int(input("Please choose a number between 1 and 10: "))
        current_sum += choice
    else:
        choice = minimax(current_sum, False, 0)
        current_sum += choice
    return current_sum

def minimax(current_sum, is_our_turn, depth):
    if current_sum >= 100:
        return 0
    if is_our_turn:
        best_value = -float('inf')
        for i in range(1, 11):
            value = minimax(current_sum + i, False, depth + 1)
            best_value = max(best_value, value)
        return best_value
    else:
        best_value = float('inf')
        for i in range(1, 11):
            value = minimax(current_sum + i, True, depth + 1)
            best_value = min(best_value, value)
        return best_value

def play_game():
    current_sum = 0
    is_our_turn = True
    while current_sum < 100:
        print("Current sum: ", current_sum)
        current_sum = play_turn(current_sum, is_our_turn)
        is_our_turn = not is_our_turn
    print("Game over!")

play_game()
```

2. Alpha-Beta 剪枝算法 Python 实现

假设需要对一个二叉搜索树进行 Alpha-Beta 剪枝，代码如下。

```
    #定义一个 Node 类表示搜索树中的结点
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

#实现 Alpha-Beta 剪枝算法
def alphabeta(node, alpha, beta, maximizingPlayer):
    if node is None:
        return 0

    if maximizingPlayer:
        value = float('-inf')
        #遍历左子树
        value = max(value, alphabeta(node.left, alpha, beta, False))
        alpha = max(alpha, value)
        #如果 beta 小于或等于 alpha,就剪枝
        if beta <= alpha:
            return value
        #遍历右子树
        value = max(value, alphabeta(node.right, alpha, beta, False))
        alpha = max(alpha, value)
        return value
    else:
        value = float('inf')
        #遍历左子树
        value = min(value, alphabeta(node.left, alpha, beta, True))
        beta = min(beta, value)
        #如果 beta 小于或等于 alpha,就剪枝
        if beta <= alpha:
            return value
        #遍历右子树
        value = min(value, alphabeta(node.right, alpha, beta, True))
        beta = min(beta, value)
        return value
```

3.6　蒙特卡罗搜索

无论是 3.1 节中介绍的一般搜索问题，还是 3.5 节中介绍的对抗搜索问题，在问题特别复杂时，搜索树都有可能会变得十分巨大，以至于搜索算法很难在短时间内搜索整棵搜索树。为了解决这个问题，3.4 节探讨了如何利用辅助信息来找到高效的结点扩展顺序，3.5 节介绍了 α-β 剪枝算法来减少需扩展的结点数量。不难发现，对搜索算法进行优化以提高搜索效率基本上是在解决如下两个问题：优先扩展哪些结点以及放弃扩展哪些结点，综合

来看也可以概括为如何高效地扩展搜索树。

如果将目标稍微降低，改求解一个近似最优解，则上述问题可以看成是如下的探索性问题：算法从根结点开始，每一步动作为选择（在非叶子结点）或扩展（在叶子结点）一个子结点。可以用执行该动作后所收获的收益来判断该动作优劣。收益可以根据从当前结点出发到达目标结点的路径的代价或下棋最终的胜负来定义。算法会倾向于扩展获得收益较高的结点。

算法事先并不知道每个结点将会得到怎样的代价分布，所以只能通过采样式搜索来得到计算收益的样本。由于算法利用的是蒙特卡罗方法来采样估计每个动作的优劣，因此被称为蒙特卡罗搜索算法。下面首先来学习下什么是蒙特卡罗方法。

3.6.1 蒙特卡罗方法

蒙特卡罗方法是一类基于概率方法的统称，它是通过将一个计算问题转换为概率问题后，利用随机性，通过求解概率的方法来求解原始问题的解，最早由冯·诺依曼和乌拉姆等发明。这类方法的特点是，可以在随机采样上计算得到近似结果，随着采样的增多，得到的结果是正确结果的概率逐渐加大，但在（放弃随机采样，而采用类似全采样这样的确定性方法）获得真正的结果之前，无法知道目前得到的结果是不是真正的结果。例如，一个有 1000 个整数的集合，要求其中位数，可以从中抽取 $m<1000$ 个数，把它们的中位数近似地看作这个集合的中位数。随着 m 增大，近似结果是最终结果的概率也在增大，但除非把整个集合全部遍历一遍，否则无法知道近似结果是不是真实结果。

对于简单问题来说，蒙特卡罗是个“笨”办法，例如上面求中位数的例子，可直接用排序算法得出计算结果。但对有些问题来说，蒙特卡罗往往是有效，有时甚至是唯一可行的方法。如 3.5 节介绍的 α-β 剪枝算法在国际象棋中取得了成功，但采用这种方法设计的围棋软件水平却很低。原因主要是国际象棋的棋局局面特征比较明显，通过对每颗棋子单独评分再求和就可以实现对整个局面的评估。但对于围棋来说，棋子之间是紧密联系的，单个棋子一定要与其他棋子联系在一起考虑，才有可能体现出它的作用。棋局评判能力要求更高，上述方法基本不起任何作用。再者，国际象棋的棋盘大小为 64，围棋的棋盘大小为 361。由于棋盘大小不同，每走一步国际象棋和围棋的计算量的要求是不一样的，围棋明显要求更高。另外一个可以说明计算能力要求不同的指标是搜索空间，在该指标上国际象棋和围棋也存在指数级的差异，国际象棋是 10^{50}，而围棋是 10^{171}。宇宙中的原子总数总共大约也才 10^{80}，因此围棋的搜索空间绝对算是天文数字，以目前的运算能力是无法达到的。所以采用蒙特卡罗这种基于概率的方法，通过随机模拟的办法来评判棋局，求解一个近似最优解。

同时因为下棋是甲乙双方一步一步轮流进行的，甲方希望走对自己最有利的棋，乙方也希望走对自己最有利的棋，双方是一个对抗的过程。所以在模拟过程中需要考虑到这种一人一步的对抗性，将搜索树考虑进来。在这样的思想指导下，研究者将蒙特卡罗方法与下棋问题的搜索树相结合，即提出了蒙特卡罗树搜索算法。

3.6.2 蒙特卡罗树搜索算法

蒙特卡罗搜索算法分为以下 4 个步骤。

选择（Selection）：选择指算法从搜索树的根结点开始，向下递归选择子结点，直至到达

叶子结点或者到达具有还未被扩展过的子结点的结点 L，如图 3.16(a)所示。

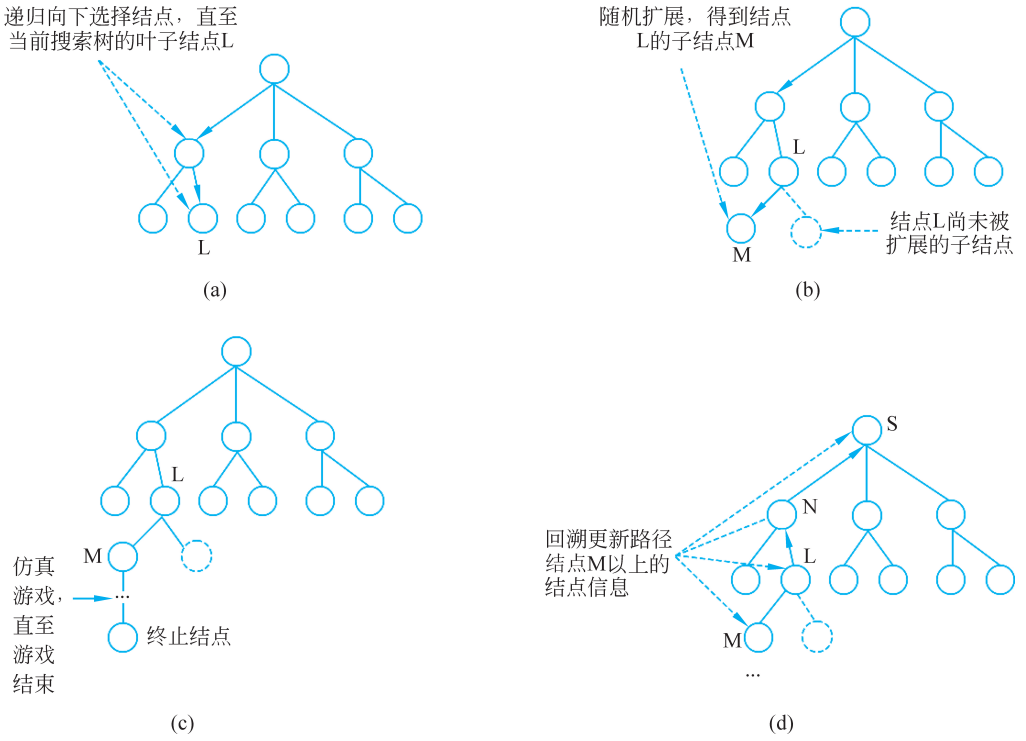

图 3.16　蒙特卡罗搜索算法

扩展(Expansion)：如果结点 L 不是一个终止结点(或对抗搜索的终局结点)，则随机扩展它的一个未被扩展过的后继边缘结点 M，如图 3.16(b)所示。

模拟(Simulation)：从结点 M 出发，模拟扩展搜索树，直到找到一个终止结点，如图 3.16(c)所示。

反向传播(Back Propagation)：用模拟所得结果(终止结点的代价或游戏终局分数)回溯更新模拟路径中 M 以上(含 M)结点的收益均值和被访问次数，如图 3.16(d)所示。

在第一个步骤选择过程中，主要目的就是在有限的时间内选出重点结点来进行模拟，从下围棋的角度来说，就是能挑选出最好的行棋走步。这里的模拟不一定是直接对该结点做随机模拟，也可能是通过对其后辈结点的模拟达到对该结点模拟的目的。在选择过程中，如果一个结点的子结点全部生成完了，则要继续从其子结点中进行选择，直到发现某个结点它还有未生成的子结点为止。

在蒙特卡罗树搜索的过程中，根据到目前为止的模拟结果，搜索树上的每个结点都获得了一定的模拟次数和一个收益值，模拟次数可能有多有少，收益值也有大有小。对于那些模拟次数比较少的结点，由于模拟的次数比较少，不知道它的真实情况，所以无论收益值高低，都应该优先选择以便进一步模拟，了解其真实收益情况。那么收益值大的结点就一定是真的收益值高吗？这也很有可能是因为模拟得不够充分暂时体现出虚假的高分，需要进一步模拟考察。所以在选择结点时应该要同时考虑目前为止结点的收益值和模拟次数，例如，对于某个结点 L，如果它的收益值又高、模拟次数又少，这样的结点肯定要优先选择，以便确认它的收益值的真实性。如果它的收益值很低、模拟次数又多，说明这个低收益值已经比较可