第 3 章

搜索策略

3.1 搜索的基本概念

3.1.1 搜索的含义

如何在大量的结构不良或非结构化的问题中获取对自己有用的信息，是人工智能中非常重要的一部分。对于这些问题，一般很难获得其全部信息，更没有现成的算法可供使用。因此，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，就显得尤为重要。搜索就是要寻找一个操作序列，使问题从初始状态转换到目标状态。这个操作序列就是目标的解。因此，所谓搜索，就是根据问题的实际情况，按照一定的策略或规则，从知识库中寻找可利用的知识，从而构造一条使问题获得解决的推理路线的过程。搜索包含两层含义：一是要找到从初始事实到问题最终答案的一条推理路线，二是找到的这条路线是时间和空间复杂度最小的求解路线。

通常搜索策略的主要任务是确定选取规则的方式。可根据是否使用启发式信息分为盲目搜索和启发式搜索，也可以根据问题的表示方法分为状态空间搜索和与/或树搜索。

盲目搜索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，一般统称为无信息引导的搜索策略。由于搜索总是按照预定的控制策略进行搜索，因此这种搜索策略具有盲目性，效率不高，不便于复杂问题的求解。启发式搜索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，加速问题的求解过程，使搜索朝着最有希望的方向前进，找到最优解。

状态空间搜索是指用状态空间法来求解问题所进行的搜索。与/或树搜索是指用问题归约法来求解问题时进行的搜索。下面分别介绍状态空间法与问题归约法。

3.1.2 状态空间法

在分析了人工智能研究的求解方法之后，就会发现许多问题求解方法采用了试探搜索方法。也就是说，这些方法是通过在某个可能的解空间内寻找一个解来求解问题。这种基于解空间的问题表示和求解方法就是状态空间法。状态空间搜索的研究焦点在于设计高效的搜索算法，以降低搜索代价并解决组合爆炸问题。

1. 状态空间及其搜索的表示

在状态空间表示法中，问题是用“状态”和“操作”来表示的，问题求解的过程使用状态空间来表示。

1）状态

状态(state)是表示问题求解过程中每一步问题状况的数据结构，可以用如下形式表示：

$$S_k=\{S_{k0},S_{k1},\cdots\}$$

在这种表示方式中，当对每一个分量都给予确定的值时，就得到了一个具体的状态。其中，每一个分量称为状态变量。实际上，任何一种类型的数据结构都可以用来描述状态，只要它有利于问题求解，就可以选用。

2）操作

操作(operation)也称为算符，它是把问题从一种状态变换为另一种状态的手段。当对一个问题状态使用某种操作时，它将引起该状态中某些分量值的变化，从而使问题从一个具体状态变为另一个具体状态。操作符可以是过程、规划、数学算子、运算符号或逻辑符号等。操作可理解为状态集合上的一个函数，它描述状态之间的关系。

3）状态空间

状态空间(state space)用来描述一个问题的全部状态及这些状态之间的相互关系。状态空间常用一个三元组(S,F,G)来表示。其中：

- S 为问题的所有初始状态的集合，其中的每个元素表示一种状态。
- F 为操作的集合，用于把一个状态转换为另一个状态。
- G 是 S 的一个非空子集，为目标状态的集合。它可以是若干具体的状态，也可以是对某些状态性质的描述。

状态空间也可以用一个赋值的有向图来表示，该有向图称为状态空间图。在状态空间图中，节点表示问题的状态，有向边(弧)表示操作。

2. 状态空间问题的例子

下面通过具体例子来说明状态空间法。

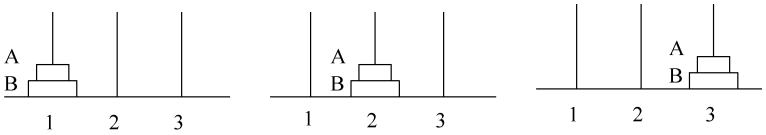
例 3.1 二阶梵塔问题。设有 3 根柱子，它们的编号分别为 1 号、2 号、3 号。在初始情况下，1 号柱子上穿有 A 和 B 两个圆盘，A 比 B 小，A 位于 B 的上面。要求把这两个圆盘全部移到另一根柱子上，而且规定每次只能移动一个圆盘，任何时刻都不能使大圆盘位于小圆盘的上面。

解：设用 $S_k=\{S_{k0},S_{k1}\}$ 表示问题的状态，其中 S_{k0} 表示圆盘 A 所在的柱子号，S_{k1} 表

示圆盘 B 所在的柱子号。全部可能的问题状态共有以下 9 种：

$$S_0=\{1,1\}\quad S_1=\{1,2\}\quad S_2=\{1,3\}\quad S_3=\{2,1\}\quad S_4=\{2,2\}$$
$$S_5=\{2,3\}\quad S_6=\{3,1\}\quad S_7=\{3,2\}\quad S_8=\{3,3\}$$

其中，初始状态 S_0 和目标状态 S_4、S_8 如图 3.1 所示。



(a) 初始状态S_0　(b) 目标状态S_4　(c) 目标状态S_8

图 3.1　二阶梵塔问题的部分状态

问题的初始状态集合为 $S_k=\{S_0\}$，目标状态集合为 $G=\{S_4,S_8\}$。操作分别用 A(i,j)和 B(i,j)表示。其中 A(i,j)表示把圆盘 A 从第 i 号柱子移到第 j 号柱子上，B(i,j)表示把圆盘 B 从第 i 号柱子移到第 j 号柱子上。共有 12 种操作，分别是

A(1,2)　A(1,3)　A(2,1)　A(2,3)　A(3,1)　A(3,2)
B(1,2)　B(1,3)　B(2,1)　B(2,3)　B(3,1)　B(3,2)

根据上述 9 种可能的状态和 12 种操作，可构成二阶梵塔问题的状态空间图，如图 3.2 所示。

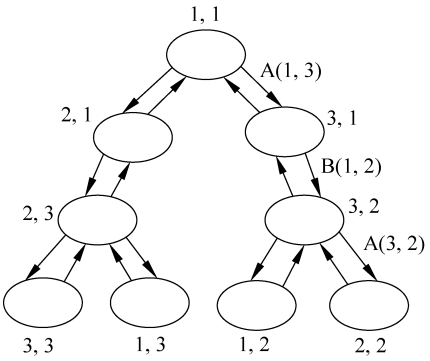


图 3.2　二阶梵塔的状态空间图

在图 3.2 中，从初始节点(1,1)到目标节点(2,2)及(3,3)的任何一条路径都是问题的一个解。其中，最短的路径长度是 3，它由 3 个操作组成。例如，从初始状态(1,1)开始，通过使用操作 A(1,3)、B(1,2)及 A(3,2)，可到达目标状态(2,2)。

例 3.2　作为状态空间的经典例子，我们来观察“传教士与野人问题”。设 N 个传教士带领 N 个野人划船渡河，且为安全起见，渡河需要遵循两个约束：①船上人数不得超过载重限量，设为 K 个人；②为预防野人攻击，任何时刻(包括两岸、船上)野人数目不得超过传教士数目。

为便于理解状态空间表示方法，可以将该问题简化为一个特例：$N=3$，$K=2$；并以变量 m 和 c 分别表示传教士和野人在左岸或者在船上的实际人数，变量 b 表示船是否在左岸(值 1 表示船在左岸，否则为 0)。从而上述约束条件转变成为 $m+c\leqslant 2$，$m\geqslant c$。

考虑到在这个渡河问题中,左岸的状态描述(m,c,b)可以决定右岸的状态,所以整个问题的状态就可以用左岸的状态来描述,以简化问题的表示。设初始状态下传教士、野人和船都在左岸,目标状态下这三者均在右岸,问题状态以三元组(m,c,b)表示,则问题求解任务可以描述为

$$(3,3,1)\rightarrow(0,0,0)$$

在此问题中,状态空间可能的状态总数为$4\times4\times2=32$,但由于要遵守安全约束,只有20种状态是合法的。下面是几个不合法状态的例子:

$$(1,0,1),(1,2,1),(2,3,1)$$

鉴于存在不合法的状态,还会导致某些合法的状态不可达,例如状态(0,0,1)、(0,3,0)。因此,此问题总共只有16种可达的合法状态:

$$S_0=(3,3,1)\quad S_1=(3,2,1)\quad S_2=(3,1,1)\quad S_3=(2,2,1)$$
$$S_4=(1,1,1)\quad S_5=(0,3,1)\quad S_6=(0,2,1)\quad S_7=(0,1,1)$$
$$S_8=(3,2,0)\quad S_9=(3,1,0)\quad S_{10}=(3,0,0)\quad S_{11}=(2,2,0)$$
$$S_{12}=(1,1,0)\quad S_{13}=(0,2,0)\quad S_{14}=(0,1,0)\quad S_{15}=(0,0,0)$$

有了这些状态,还需要考虑可进行的操作。在此问题中,操作是指用船把传教士或野人从河的左岸运到右岸,或者从河的右岸运到左岸,并且每个操作都应该满足条件①、②。因此,操作应该由两部分组成,即条件部分和动作部分。操作只有当其条件具备时才能进行,动作则刻画了应用此操作所产生的结果。

此处用符号P_{ij}表示从左岸到右岸的运人操作,用符号Q_{ij}表示从右岸到左岸的运人操作,其中,i表示船上的传教士数,j表示船上的野人数。通过分析可以知道有10种操作可供选择,其操作集为

$$F=\{P_{01},P_{10},P_{11},P_{02},P_{20},Q_{01},Q_{10},Q_{11},Q_{02},Q_{20}\}$$

下面以P_{01}和Q_{01}为例说明这些操作的条件和动作:

操作符号	条件	动作
P_{01}	$b=1,m=0$ 或 $3,c\geqslant1$	$b=0,c=c-1$
Q_{01}	$b=0,m=0$ 或 $3,c\leqslant2$	$b=1,c=c+1$

从而可画出类似图3.2所示的状态空间图。

例3.3 猴子摘香蕉问题。在前面讨论谓词逻辑知识表示时曾经提到这一问题,现在用状态空间法求解。

解:问题状态可用四元组(w,x,y,z)来表示。其中,w表示猴子的水平位置;x表示箱子的水平位置;y表示猴子是否在箱子上,当猴子在箱子上时y取1,否则y取0;z表示猴子是否拿到香蕉,当拿到香蕉时z取1,否则z取0。

所有可能的状态为

S_0:$(a,b,0,0)$ 初始状态

S_1:$(b,b,0,0)$

S_2:$(c,c,0,0)$

S_3:$(c,c,1,0)$

S_4:$(c,c,1,1)$ 目标状态

允许的操作为

Goto(u)：猴子走到位置 u，即$(w,x,0,0)\rightarrow(u,x,0,0)$。

Pushbox(v)：猴子推着箱子到水平位置 v，即$(x,x,0,0)\rightarrow(v,v,0,0)$。

Climbbox：猴子爬上箱子，即$(x,x,0,0)\rightarrow(x,x,1,0)$。

Grasp：猴子拿到香蕉，即$(c,c,1,0)\rightarrow(c,c,1,1)$。

这个问题的状态空间图如图 3.3 所示。由初始状态变为目标状态的操作序列为

$$\{\text{Goto}(b),\text{Pushbox}(c),\text{Climbbox},\text{Grasp}\}$$

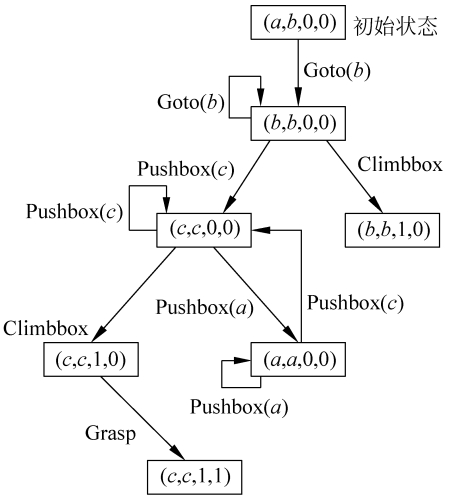


图 3.3　猴子摘香蕉问题的状态空间图

3.1.3　问题归约法

问题归约法是另一种对问题进行描述及求解的办法。其基本思想是：对问题进行分解和变换，将此问题最终变为一个子问题的集合，通过求解子问题达到求解原问题的目的。问题归约法适用于当初始问题比较复杂时，分解后的子问题的解可以直接得到，从而解决了初始问题的情况。

所谓“分解”是指：如果一个问题 P 可以归约为一组子问题 $P_1,P_2,\cdots,P_n$，并且只有当所有子问题 P_i 都有解时原问题 P 才有解，任何一个子问题 P_i 无解都会导致原问题 P 无解，则称此种归约为问题的分解，即分解所得到的子问题的“与”与原问题 P 等价。

所谓“变换”是指：如果一个问题 P 可以归约为一组子问题 $P_1,P_2,\cdots,P_n$，并且子问题 P_i 中只要有一个有解，则原问题 P 就有解，只有当所有子问题 P_i 都无解时原问题 P 才无解，称此种归约为问题的等价变换，简称变换，即变换所得到的子问题的“或”与原问题 P 等价。

1. 问题归约描述

问题归约法由下面 3 个部分组成：

(1) 一个初始问题的描述。

(2) 一套把问题变换成为子问题的操作符。

(3) 一套本原问题的描述。

从目标(要解决的问题)出发逆向推理，建立子问题以及子问题的子问题，直至最后把初

始问题归纳成为一个平凡的本原问题集合，这就是问题归约的实质。

例 3.4 三阶梵塔问题。有 3 个柱子(1,2,3)和 3 个不同尺寸的圆盘(A,B,C)。在每个圆盘的中心有一个孔，所以圆盘可以堆叠在柱子上。最初，全部 3 个圆盘都堆在柱子 1 上，最大的圆盘 C 在底部，最小的圆盘 A 在顶部。要求把所有圆盘都移到柱子 3 上，每次只许移动一个，而且只能先搬动柱子顶部的圆盘，还不许把尺寸较大的圆盘堆放在尺寸较小的圆盘上。

若采用状态空间法来求解这个问题，其状态空间有 27 个节点，每个节点代表柱子上的圆盘的一种正确位置。当然，也可以用问题归约法来求解此问题。

归约过程如下：

(1) 移动圆盘 A 和 B 至柱子 2 的双圆盘移动问题。

(2) 移动圆盘 C 至柱子 3 的单圆盘移动问题。

(3) 移动圆盘 A 和 B 至柱子 3 的双圆盘移动问题。

可以看出简化后的问题每一个都比原问题容易，所以原问题都可以变成易解决的本原问题。而将一个复杂的原问题归约成一系列本原问题的过程可以很方便地用与/或树来表示。下面介绍有关与/或树的相关内容。

2. 与/或树的相关概念

1) 节点与弧线

父节点：是一个初始问题或是可分解为子问题的问题节点。

子节点：是一个初始问题或是子问题分解的子问题节点。

或节点：只要解决某个问题就可解决其父问题的节点集合。

与节点：只有解决所有子问题，才能解决其父问题的节点集合。

端节点：没有子节点的节点。

终止节点：本原问题所对应的节点。由此可见，终止节点一定是端节点，而端节点却不一定是终止节点。

弧线：是父节点指向子节点的圆弧连线。

2) 或树、与/或树和解树

把一个原问题变换成若干个子问题可用一个或树来表示，如图 3.4 所示。

把一个问题分解为若干个子问题可用一个与树来表示，如图 3.5 所示。

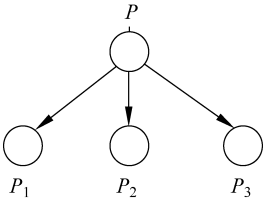


图 3.4 或树

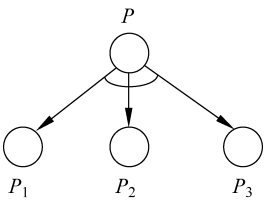


图 3.5 与树

如果一个问题既需要通过分解，又需要通过变换才能得到其本原问题，其归约过程可以用一个与/或树来表示，如图 3.6 所示。

由可解节点构成，并且由这些可解节点可以推出初始节点（它对应着原问题）为可解节点的子树为解树。在解树中一定包含初始节点。例如，在图 3.7 所给的与/或树中，用粗线表示的子树是一个解树。在该图中，节点 P 为原始问题节点，用 t 标出的节点是终止节点。

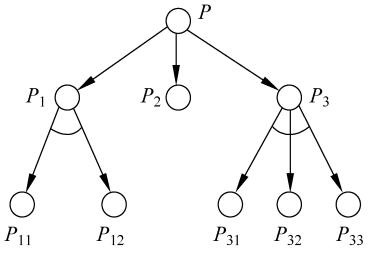


图 3.6　与/或树

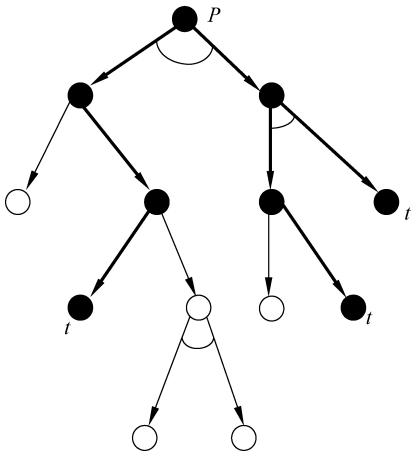


图 3.7　解树

3. 问题归约的例子

如例 3.4 的三阶梵塔问题，此问题也可以用状态空间法来解，不过本例主要用它来说明如何用问题归约法来解决问题。

首先，定义该问题的形式化表示方法。设用三元组(i,j,k)表示问题在任意时刻的状态，用→表示状态的转换。在此三元组中，i 代表圆盘 C 所在的柱子号，j 代表圆盘 B 所在的柱子号，k 代表圆盘 A 所在的柱子号。

前面分解的 3 个子问题可分别表示如下。

(1) 移动圆盘 A 和 B 至柱子 2 的双圆盘问题：

$$(1,1,1) \rightarrow (1,2,2)$$

(2) 移动圆盘 C 至柱子 3 的单圆盘问题：

$$(1,2,2) \rightarrow (3,2,2)$$

(3) 移动圆盘 A 和 B 至柱子 3 的双圆盘问题：

$$(3,2,2) \rightarrow (3,3,3)$$

其中，子问题(1)、(3)都是一个二阶梵塔问题，它们都可以再继续进行分解；子问题(2)是本原问题，它已经不需要再分解。

三阶梵塔问题的分解过程可用图 3.8 所示的与/或树来表示。在该与/或树中，有 7 个终止节点，它们分别对应着 7 个本原问题。得到问题的解为

$$(1,1,1) \rightarrow (1,1,3) \quad (1,1,3) \rightarrow (1,2,3) \quad (1,2,3) \rightarrow (1,2,2)$$
$$(1,2,2) \rightarrow (3,2,2) \quad (3,2,2) \rightarrow (3,2,1) \quad (3,2,1) \rightarrow (3,3,1)$$
$$(3,3,1) \rightarrow (3,3,3)$$

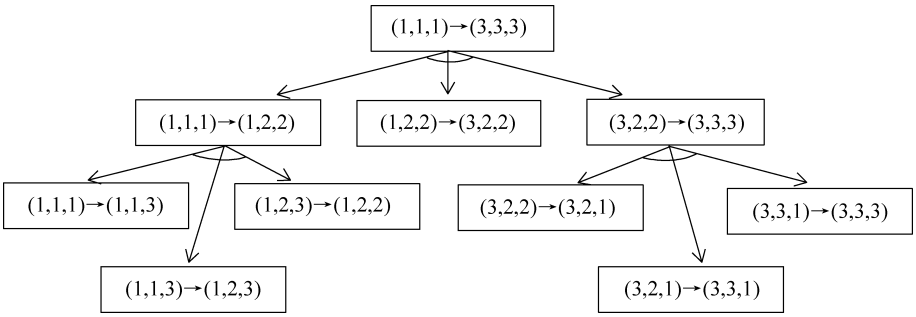


图 3.8　三阶梵塔问题的与/或树表示

3.2　状态空间搜索

状态空间的搜索策略分为盲目搜索和启发式搜索两大类。下面讨论的广度优先搜索、深度优先搜索和有界深度优先搜索都属于盲目搜索策略。

盲目搜索策略的一个共同特点是它们的搜索路线是已经决定好的，没有利用被求解问题的任何特征信息，在决定要被扩展的节点时，并没有考虑该节点到底是否可能出现在解的路径上，也没有考虑它是否有利于问题的求解以及所求的解是否为最优解。

3.2.1　盲目搜索

1. 一般图搜索

一般图搜索是在状态空间中搜索从初始状态到目标状态解答路径的过程。由于问题的状态空间可以用一个有向图来表示，因此状态空间搜索实际上就是对有向图的搜索。从图搜索的角度来看，状态空间搜索的基本思想可以概括为：将问题的初始状态作为当前扩展节点对其进行扩展，生成一组子节点，然后检查目标状态是否出现在这些节点中。如果出现，表明搜索成功，即找到了该问题的解；如果没有出现，则再按照某一种搜索策略从已生成的子节点中选择一个节点作为当前的扩展节点。重复上述过程，直到目标状态出现在子节点中或者没有可供扩展的节点为止。所谓对一个节点进行“扩展”是指对该节点用某个可用操作施加作用，生成该节点的一组子节点。

在开始搜索过程之前，先定义两个数据结构 OPEN 表与 CLOSED 表。OPEN 表用于存放刚生成的节点，对于不同的搜索策略，节点在 OPEN 表中的排列顺序是不同的。例如，对广度优先搜索，节点按生成的顺序排列，先生成的节点排在前面，后生成的节点排在后面。CLOSED 表用于存放将要扩展或已扩展的节点。

搜索步骤如下：

(1) 把初始节点 S_0 放入 OPEN 表，并建立目前只包含 S_0 的图，记为 G。

(2) 检查 OPEN 表是否为空，若为空则问题无解，失败退出。

(3) 把 OPEN 表的第一个节点取出放入 CLOSED 表，并记该节点为节点 n。

(4) 判断节点 n 是否为目标节点。若是，则求得问题的解，成功退出。

(5) 考察节点 n，生成一组子节点。把其中不是节点 n 先辈的那些子节点记作集合 M，并把这些子节点作为节点 n 的子节点加入 G 中。

(6) 针对 M 中子节点的不同情况，分别进行如下处理：

- 对那些未曾在 G 中出现过的 M 成员设置一个指向父节点(即节点 n)的指针，并将它们放入 OPEN 表。
- 对那些先前已在 G 中出现过的 M 成员，确定是否需要修改它指向父节点的指针。
- 对那些先前已经在 G 中出现并且已经扩展了的 M 成员，确定是否需要修改其后继节点指向父节点的指针。

(7) 按某种搜索策略对 OPEN 表中的节点进行排序。

(8) 转(2)。

2. 广度优先搜索

广度优先搜索又称为宽度优先搜索，是一种先生成的节点先扩展的简单策略。从初始节点 S_0 开始逐层向下扩展，只有当同一层的节点全部被搜索完以后，才能进入下一层继续搜索。

在搜索的过程中，要建立两个数据结构：OPEN 表和 CLOSED 表，其形式分别如表 3.1 和表 3.2 所示。

表 3.1　OPEN 表

节　　点	父节点编号

表 3.2　CLOSED 表

编　　号	节　　点	父节点编号

OPEN 表用于存放刚生成的节点，对于不同的策略，节点在此表中的排列顺序是不同的。CLOSED 表用于存放将要扩展或者已扩展的节点(节点 n 的子节点)。

所谓对一个节点进行扩展，是指用合适的算符对该节点进行操作，生成一组子节点。一个节点经一个算符操作后一般只生成一个子节点，但对一个可使用的节点可能有多个，故此时会生成一组子节点。需要注意的是，在这些子节点中，可能有些是当前扩展节点(节点 n)的父节点或者祖父节点等，此时不能把这些先辈节点作为当前扩展节点的子节点。

在广度优先搜索策略中，OPEN 表中的节点是按进入的先后排序，先进入 OPEN 表的节点排在前，后进入的节点排在后。

因此，广度优先搜索的基本思想是：从初始节点 S_0 开始，逐层地对节点进行扩展并考察它是否为目标节点，在第 n 层的节点没有全部扩展并考察之前，不对第 $n+1$ 层的节点进行扩展。其搜索过程如下：

(1) 把初始节点 S_0 放在 OPEN 表中。

(2) 若 OPEN 表为空，则问题无解，退出。

(3) 把 OPEN 表中的第一个节点(记为节点 n)取出放入 CLOSED 表中。

(4) 考察节点 n 是否为目标节点。若是，则得到问题的解，成功退出。

(5) 若节点 n 不可扩展，则转(2)。

(6) 扩展节点 n，将其子节点放入 OPEN 表的尾部，并且为每一个子节点设置指向父节

点的指针，然后转(2)。

例 3.5 重排九宫问题。在 3×3 的方格棋盘上，分别放置了标有数字 1～8 的 8 张牌，初始状态为 S_0，目标状态为 S_g，如图 3.9 所示。可用的操作有空格左移、空格上移、空格右移、空格下移。即只允许把位于空格左、上、右、下的牌移入空格。要求用广度优先搜索策略寻找初始状态到目标状态的路径。

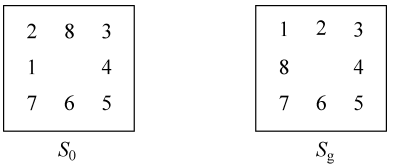


图 3.9 重排九宫问题

解：应用广度优先策略，可以在第四级得到解，搜索树如图 3.10 所示。可以看出，解的路径是

$$S_0 \rightarrow 3 \rightarrow 8 \rightarrow 16 \rightarrow 26$$

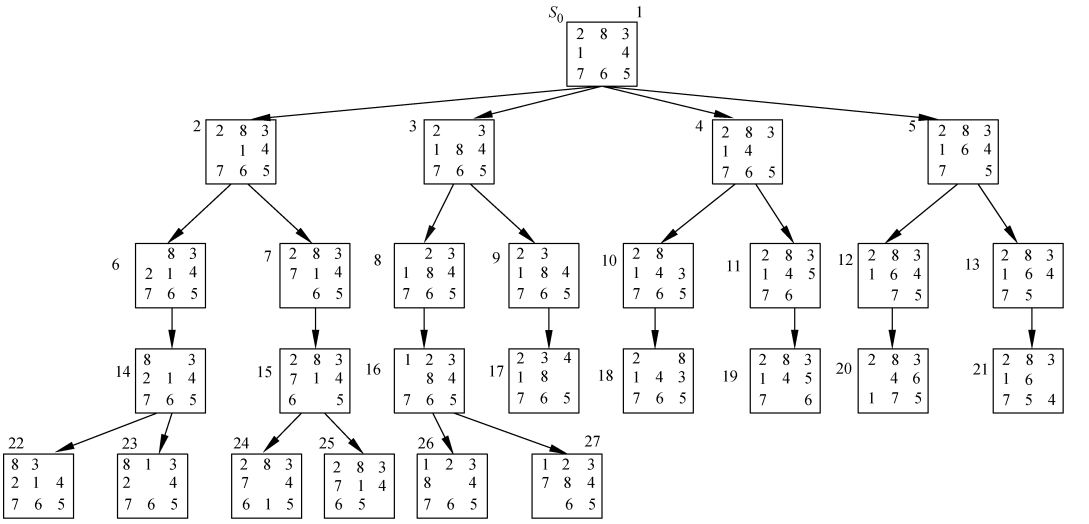


图 3.10 重排九宫的广度优先搜索

由于广度优先搜索总是在生成扩展完第 n 层的节点后才转到第 $n+1$ 层，所以总能找到最优解。但是实用意义不大，广度优先算法的主要缺点是盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用节点，最后导致组合爆炸，尽管耗尽资源，在可利用的空间中也找不到解。

3. 深度优先搜索

深度优先搜索总是先扩展后生成的节点。其基本思想是：从初始节点 S_0 开始，在其子节点中选择一个最新生成的节点进行考察，如果该子节点不是目标节点并且可以扩展，则扩展此子节点，再在此节点的子节点中选择一个最新生成的节点进行扩展，一直如此向下搜索。当到达某一子节点，此子节点既不是目标节点又不能继续扩展时，才选择其兄弟节点进行考察。其搜索过程如下：

(1) 初始节点放入 OPEN 表中。

(2) 若 OPEN 表为空，则问题无解，退出。

(3) 把 OPEN 表中的第一个节点(记为 n)取出放入 CLOSED 表中。

(4) 考察节点 n 是否为目标节点,若是,则问题解求出,退出。

(5) 若节点不可扩展,则转(2)。

(6) 扩展节点 n,将其子节点放入 OPEN 表的首部,并为其配置指向父节点的指针,然后转(2)。

对比广度优先搜索与深度优先搜索,可以看出二者唯一的区别是,广度优先搜索时将节点 n 的子节点放到 OPEN 表的尾部,而深度优先搜索时把节点 n 的子节点放到 OPEN 表的首部。这一不同点使得搜索的路线完全不同。

例 3.6　对例 3.5 的重排九宫问题进行深度优先搜索。

解:用深度优先搜索可得到图 3.11 所示的搜索树。但这只是搜索树的一部分,尚未到达目标节点,仍可继续往下搜索。

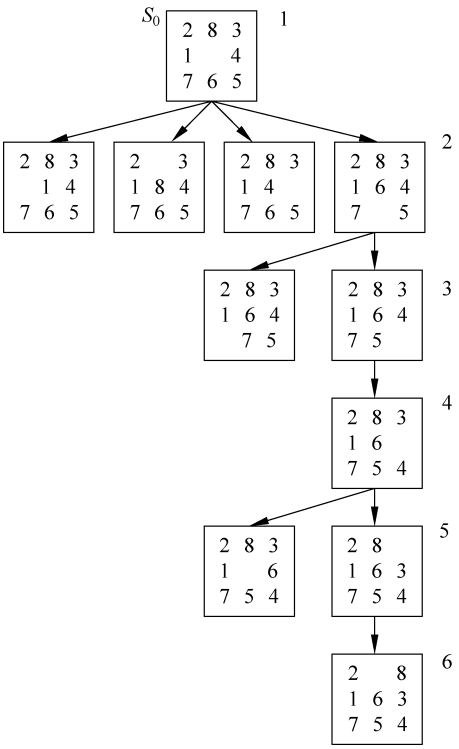


图 3.11　重排九宫的深度优先搜索

从深度优先搜索的算法可以看出,搜索一旦进入某个分支,就将沿着这个分支一直向下进行下去,如果目标恰好在这一个分支上,则可以很快找到解。但是,如果目标不在此分支上,且该分支是一个无穷分支,则搜索过程就不可能找到解。因此,深度优先搜索是一种不完备策略,即使问题有解也不一定能够找到。此外,即使能够找到解,此解也不一定是最短路径的解。

广度优先搜索和深度优先搜索各有不足,为了弥补各自的不足,可以采用有界深度优先算法。顾名思义,这是对深度优先算法给出深度限制 d_m,当搜索深度达到了 d_m,即使没有找到目标,也要停止该分支的搜索,换到另一个分支进行搜索。折中的办法是广度优先和深度优先策略的一种结合。

4. 代价树搜索

当路径的花费与弧有关时，我们常常想到的是花费最小的路径。例如，对于一个投递机器人，花费可能是两个位置之间的距离，需要解出距离最短的那条解路径。花费也可能是机器人按照弧实施动作所需要的各种资源。而在前面讨论的各种搜索策略中，并没有将注意力放在各边的代价上，因为默认各边的代价是相同的，且都为一个单位量。但是，对于许多实际问题，状态空间的各个边的代价不可能完全相同。为此，需要在搜索树中给每一条边都标上代价。这种标有代价的树称为代价树。

1) 代价树的代价表示

$$g(n_2)=g(n_1)+c(n_1,n_2)$$

其中，n_1 与 n_2 分别表示某一父节点与其子节点，$g(n)$表示从初始节点 S_0 到节点 n 的代价，用 $c(n_1,n_2)$表示从父节点 n_1 到其子节点 n_2 的代价。

在代价树中，最小代价的路径和最短路径(即路径长度最短)是有可能不同的。代价树搜索的目的是为了找到最优解，即找到一条代价最小的解路径。

2) 代价树的广度优先搜索

代价树的广度优先搜索每次从 OPEN 表中选择节点以及往 CLOSED 表中存放节点时，总是选择代价最小的节点。即 OPEN 表中节点的顺序是按照其代价由小到大排列的，代价小的节点排在前面，代价大的节点排在后面，与节点在树中的位置无关。

代价树的广度优先搜索算法如下：

(1) 将初始节点 S_0 放入 OPEN 表中，置 S_0 的代价 $g(S_0)=0$。

(2) 如果 OPEN 表为空，则问题无解，失败退出。

(3) 把 OPEN 表的第一个节点取出放入 CLOSED 表，并记该节点为 n。

(4) 考察节点 n 是否为目标节点，若是，则找到了问题的解，成功退出；否则继续。

(5) 若节点 n 不可扩展，则转(2)；否则转(6)。

(6) 扩展节点 n，生成其子节点 $n_i(i=1,2,3,\cdots)$，将这些节点放入 OPEN 表中，并为每一个子节点设置指向父节点的指针。按公式 $g(n_i)=g(n)+c(n,n_i)(i=1,2,3,\cdots)$计算各节点的代价，并根据各节点的代价对 OPEN 表中的全部节点按照从小到大的顺序重新进行排序。

(7) 转(2)。

代价树的广度优先搜索策略是完备的。如果问题有解，上述算法一定能找到它，并且找到的一定是最优解。

例 3.7 城市交通问题。设有 5 个城市，城市之间的交通线路如图 3.12 所示，图中的数字表示两个城市之间的交通费用，即代价。用代价树的广度优先搜索，求出从 A 市出发到 E 市费用最小的交通路线。

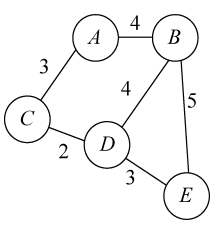


图 3.12 城市交通线路图

解：图 3.12 是一个网络图，不能直接用于搜索算法，需要将其先转换为代价树。

把一个网络图转换为代价树的方法是：从起始节点 A 开始，把与它直接邻接的节点作为其子节点。对其他的节点也做同样的处理。但当一个节点已作为某个节点的直系先辈节点时，就不能再作

为这个节点的子节点。例如，图中与节点 B 直接相邻的节点有节点 A，D，E，但由于 A 已经作为 B 的父节点在代价树中出现过了，因此 A 不能再作为 B 的子节点。此外，图中的节点除初始节点 A 外，其他节点都可能在代价树中出现多次，为区别它们的多次出现，分别用下标 1，2，…标出，但实际上是同一个节点。

转换后的代价树如图 3.13 所示。

对如图 3.13 所示的代价树，按广度优先搜索可得到最优解：

$$A \rightarrow C_1 \rightarrow D_1 \rightarrow E_2$$

其代价为 8。可见，从 A 市到 E 市的最小费用路线为：

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

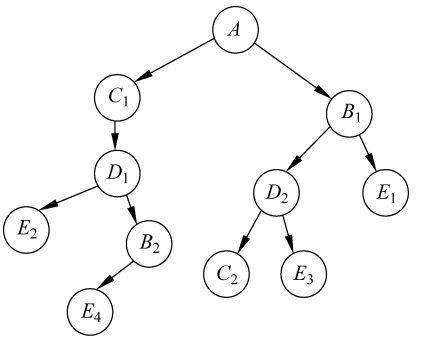


图 3.13　城市交通线路图的代价树

3）代价树的深度优先搜索

代价树的广度优先搜索每次都是从 OPEN 表的全体节点中选择一个代价最小的节点，而深度优先搜索是从刚扩展的子节点中选择一个代价最小的节点。即两种搜索策略的区别在于每次选择最小代价节点的方法不同。

代价树的深度优先搜索算法如下：

（1）将初始节点 S_0 放入 OPEN 表中，置 S_0 的代价 $g(S_0)=0$。

（2）如果 OPEN 表为空，则问题无解，失败退出。

（3）把 OPEN 表的第一个节点取出放入 CLOSED 表，并记该节点为 n。

（4）考察节点 n 是否为目标节点，若是，则找到了问题的解，成功退出。

（5）若节点不可扩展，则转(2)，否则转(6)。

（6）扩展节点 n，生成其子节点 $n_i(i=1,2,3,\cdots)$，将这些子节点按边代价由小到大放入 OPEN 表中的首部，并为每一个子节点设置指向父节点的指针。

（7）转(2)。

对例 3.7 所给出的问题，用代价树的深度优先搜索策略找到的路径为

$$A \rightarrow C_1 \rightarrow D_1 \rightarrow E_1$$

和广度优先搜索相比，深度优先搜索找到的路径是相同的。这只是一种巧合。一般来说，它找到的解不一定是最优解，即代价树的深度优先搜索策略是不完备的，甚至当搜索进入无穷分支时，算法将找不到解。

3.2.2　状态空间的启发式搜索

3.2.1 节介绍了状态空间的盲目搜索策略。例如状态空间的深度优先搜索和宽度优先搜索。这类方法进行的是一种蛮力搜索，因而效率低。本节介绍状态空间的启发式搜索策略，由于此种方法有较强的针对性，因此可以缩小搜索范围，提高搜索效率。

1. 估价函数与启发式信息

3.2.1 节所介绍的所有搜索算法都是无指导信息的，并没有考虑目标节点在哪里。它们没有使用任何指引它们该去哪里的信息，除非它们无意中发现了目标。

在搜索过程中,用于决定要扩展的下一个节点的信息,即用于指导搜索过程且与具体问题求解有关的控制信息称为启发信息;决定下一步要控制的节点称作“最有希望”的节点,其“希望”的程度通常通过构造一个函数来表示,这种函数被称为估价函数。

1) 估价函数

估价函数的任务是估计待搜索节点的重要程度,给它们排定顺序。在这里,把估价函数 $f(n)$定义为从初始节点 S_0 经过节点 n 到达目标节点的最小代价路径的代价估计值,它的一般形式为

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

其中,$g(n)$为初始节点 S_0 到节点 n 已实际付出的代价;$h(n)$是从节点 n 到目标节点 S_g 最优路径的估计代价,搜索的启发式信息主要由 $h(n)$来体现,故把 $h(n)$称为启发函数。对 $g(n)$的值,可以按指向父节点的指针,从节点 n 反向跟踪到初始节点 S_0,得到一条从初始节点 S_0 到节点 n 的最小代价路径,然后把这条路径上的所有有向边的代价相加,就得到 $g(n)$的值。

2) 启发信息

启发信息是指与具体问题求解过程有关的,并可指导搜索过程朝着最有希望的方向前进的控制信息。一般有以下 3 种:

(1) 有效地帮助确定扩展节点的信息。

(2) 有效地帮助决定哪些后继节点应被生成的信息。

(3) 能决定在扩展节点时哪些节点应从搜索树上删除的信息。

一般来说,搜索过程所使用的启发性信息的启发能力越强,扩展的无用节点就越少。

例 3.8 八数码难题。设问题的初始状态 S_0 和目标状态 S_g 如图 3.9 所示,且估价函数为 $f(n)=d(n)+W(n)$,式中,$d(n)$表示节点 n 在搜索树中的深度;$W(n)$表示节点 n 中“不在位”的数码个数,请计算初始状态 S_0 的估价函数值 $f(S_0)$。

解:在本例的估价函数中,取 $g(n)=d(n)$,$h(n)=W(n)$。此处用 S_0 到 n 的路径上的单位代价表示实际代价,用 n 中“不在位”的数码个数作为启发信息。一般来说,某节点中的“不在位”的数码个数越多,说明它离目标节点越远。

对初始节点 S_0,由于 $d(S_0)=0$,$W(S_0)=3$,因此有

$$f(S_0)=0+3=3$$

这个例子仅是为了说明估价函数的含义及估价函数值的计算。在问题搜索过程中,除了需要计算初始节点的估价函数之外,更多的是要计算新生成节点的估价函数值。

2. A 算法

在搜索的每一步都利用估价函数 $f(n)=g(n)+h(n)$对 OPEN 表中的节点进行排序,则该搜索算法称为 A 算法。由于估价函数中带有问题自身的启发性信息,因此,A 算法也称为启发式搜索算法。

根据搜索过程中选择扩展节点的范围,启发式搜索算法可分为全局择优搜索算法和局部择优搜索算法。其中,全局择优搜索算法每当需要扩展节点时,总是从 OPEN 表的所有节点中选择一个估价函数值最小的节点进行扩展。局部择优搜索算法每当需要扩展节点时,总是从刚生成的子节点中选择一个估价函数最小的节点进行扩展。下面主要讨论全局择优搜索算法。

全局择优搜索算法的搜索过程可描述如下：

(1) 把初始节点 S_0 放入 OPEN 表中，$f(S_0)=g(S_0)+h(S_0)$。

(2) 如果 OPEN 表为空，则问题无解，失败退出。

(3) 把 OPEN 表的第一个节点取出放入 CLOSED 表，并标记该节点为 n。

(4) 考察节点 n 是否为目标节点，若是，则找到了问题的解，成功退出。

(5) 若节点不可扩展，则转(2)。

(6) 扩展节点 n，生成其子节点 n_i($i=1,2,3,\cdots$)，计算每一个子节点的估价值 $f(n_i)$($i=1,2,3,\cdots$)，并为每个子节点设置指向父节点的指针，然后将这些子节点放入 OPEN 表中。

(7) 根据各节点的估价函数值，对 OPEN 表中的全部节点按从小到大的顺序重新进行排序。

(8) 转(2)。

由于上述算法的第(7)步要对 OPEN 表中的全部节点按估价函数值从小到大重新进行排序，这样在算法第(3)步取出的节点就一定是 OPEN 表的所有节点中估价函数值最小的一个节点。因此，它是一种全局择优的搜索方式。

对上述算法进一步分析还可以发现：如果取估价函数 $f(n)=g(n)$，则它将退化为代价树的广度优先搜索；如果取估价函数 $f(n)=d(n)$，则它将退化为广度优先搜索。可见，广度优先搜索和代价树的广度优先搜索是全局择优搜索的两个特例。

例 3.9　八数码难题。设问题的初始状态 S_0 和目标状态 S_g 如图 3.9 所示，估价函数与例 3.8 相同，请用全局择优搜索解决该问题。

解：这个问题的全局择优搜索树如图 3.14 所示，在图 3.14 中，每个节点旁边的数字是该节点的估价函数值。例如，对节点 S_2，其估价函数值的计算为

$$f(S_2)=d(S_2)+W(S_2)=2+2=4$$

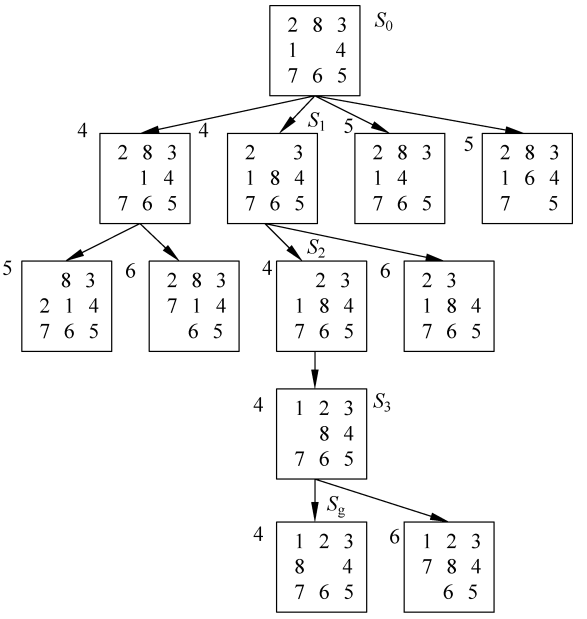


图 3.14　全局择优搜索树

从图 3.14 还可以看出,该问题的解为

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_g$$

3. A* 算法

1) A* 算法概述

A* 算法也是一种启发式搜索方法,它是对扩展节点的选择方法做了一些限制,选用了一个比较特殊的估价函数,这时的估价函数 $f(n)=g(n)+h(n)$ 是对函数 $f^*(n)=g^*(n)+h^*(n)$ 的一种估计或近似,即,$f(n)$ 是对 $f^*(n)$ 的估计,$g(n)$ 是对 $g^*(n)$ 的估计,$h(n)$ 是对 $h^*(n)$ 的估计。

函数 $f^*(n)$ 的定义是这样的:它表示从节点 S_0 到节点 n 的一条最佳路径的实际代价加上从节点 n 到目标节点 S_g 的一条最佳路径的代价之和。而 $g^*(n)$ 就是从节点 S_0 到节点 n 之间最小代价路径的实际代价,$h^*(n)$ 则是从节点 n 到目标节点 S_g 的最小代价路径上的代价。既然 $g(n)$ 是对 $g^*(n)$ 的估计,所以 $g(n)$ 是比较容易求得的,它就是从初始节点 S_0 到节点 n 的路径代价,这可通过由节点 n 到节点 S_0 回溯时把所遇各段弧线的代价加起来而得到,显然恒有 $g(n)\geqslant g^*(n)$。$h(n)$ 是对 $h^*(n)$ 的估计,它依赖于有关问题领域的启发信息,是上述提到的启发函数,其具体形式要根据问题特性进行构造。在 A* 算法中要求启发函数 $h(n)$ 是 $h^*(n)$ 的下界,即对所有的 n 均有 $h(n)\leqslant h^*(n)$。这一要求十分重要,它能保证 A* 算法找到最优解。理论分析表明,若问题存在最优解,则此限制就可能保证找到最优解。虽然,这个限制可能产生无用搜索,但是不难想象,当某一节点 n 的 $h(n)>h^*(n)$,则该节点就有可能失去优先扩展的机会,因而导致得不到最优解。

2) A* 算法性质

A* 算法具有可采纳性、单调性和信息性。

(1) 可采纳性。

所谓可采纳性是指对于可求解的状态空间图(即从状态空间图的初始节点到目标节点有路径存在)来说,如果一个搜索算法能在有限步内终止,并且能找到最优解,则称该算法是可采纳的。分 3 步证明如下:

① 对于有限图,A* 算法一定会在有限步内终止。

对于有限图,其节点个数是有限的。可见 A* 算法在经过若干次循环之后只可能出现两种情况:或者由于搜索到了目标节点而终止;或者由于 OPEN 表中的节点被取完而终止。不管发生哪种情况,A* 算法都在有限步内终止。

② 对于无限图,只要初始节点到目标节点有路径存在,则 A* 算法也必然会终止。

该证明分两步进行。证明在 A* 算法结束之前,OPEN 表中总存在节点 x'。该节点是最优路径上的一个节点,且满足

$$f(x')\leqslant f^*(S_0)$$

设最优路径是 $S_0,x_1,x_2,\cdots,x_m,S_g^*$。由于 A* 算法中的 $h(x)$ 满足 $h(x)\leqslant h^*(x)$,所以 $f(S_0),f(x_1),f(x_2),\cdots,f(x_m)$ 均不大于 $f(S_g^*)$,$f(S_g^*)=f^*(S_0)$。

又因为 A* 算法是全局择优的,所以在它结束之前,OPEN 表中一定含有 $S_0,x_1,x_2,\cdots,x_m,S_g^*$ 中的一些节点。设 x' 是最前面的一个,则它满足

$$f(x') \leqslant f^*(S_0)$$

至此，第一步证明结束。

现在进行第二步的证明。这一步用反证法，即假设 A* 算法不终止，则会得出与上一步矛盾的结论，从而说明 A* 算法一定会终止。

假设 A* 算法不终止，并设 e 是图中各条边的最小代价，$d^*(x_n)$ 是从 S_0 到节点 x_n 的最短路径长度，则显然有

$$g^*(x_n) \geqslant d^*(x_n) \times e$$

又因为

$$g(x_n) \geqslant g^*(x_n)$$

所以有

$$g(x_n) \geqslant d^*(x_n) \times e$$

因为

$$h(x_n) \geqslant 0, \quad f(x_n) \geqslant g(x_n)$$

故得到

$$f(x_n) \geqslant d^*(x_n) \times e$$

由于 A* 算法不终止，随着搜索的进行，$d^*(x_n)$ 会无限增长，从而使 $f(x_n)$ 也无限增长。这就与上一步证明得出的结论矛盾。因为对可解状态空间来说，$f^*(S_0)$ 一定是有限值。

所以，只要从初始节点到目标节点有路径存在，即使对于无限图，A* 算法也一定会终止。

③ A* 算法一定终止在最优路径上。

假设 A* 算法不是在最优路径上终止，而是在某个目标节点 t 处终止，即 A* 算法未能找到一条最优路径，则

$$f(t) = g(t) > f^*(S_0)$$

但由②的证明可知，在 A* 算法结束之前，OPEN 表中存在节点 x'，它在最优路径上，且满足

$$f(x') \leqslant f^*(S_0)$$

此时，A* 算法一定会选择 x' 来扩展而不会选择 t，这就与假设矛盾。显然，A* 算法一定终止在最优路径上。

根据可采纳性的定义及以上证明可知 A* 算法是可采纳的。同时由上面的证明还可知，A* 算法选择扩展的任何一个节点 x' 都满足如下性质：

$$f(x') \leqslant f^*(S_0)$$

(2) 单调性。

在 A* 算法中，若对启发性函数加以适当的单调性条件限制，就可使它对所扩展的一系列节点的估价函数单调递增(或非递减)，从而减少对 OPEN 表或 CLOSED 表的检查和调整，提高搜索效率。

所谓单调性限制是指 $h(x)$ 满足如下两个条件：

① $h(S_g)=0$。

② 设 x_j 是节点 x_i 的任一子节点，则有

$$h(x_i)-h(x_j)\leqslant c(x_i,x_j)$$

其中，S_g 是目标节点，$c(x_i,x_j)$是节点 x_i 到其子节点 x_j 的边代价。

若把上述不等式改写为如下形式：

$$h(x_i)\leqslant h(x_j)+c(x_i,x_j)$$

就可看出节点 x_i 到目标节点最优费用的估价不会超过从 x_i 到其子节点 x_j 的边代价加上从 x_j 到目标节点最优费用的估价。

可以证明，当 A* 算法的启发式函数 $h(x)$满足单调限制时，有如下两个结论：

① 若 A* 算法选择节点 x_n 进行扩展，则

$$g(x_n)=g^*(x_n)$$

② 由 A* 算法所扩展的节点序列的估价值是非递减的。

这两个结论都是在 $h(x)$满足单调限制时才成立的。否则，它们不一定成立。例如，对于第②个结论，当 $h(x)$不满足单调限制时，有可能某个要扩展的节点比以前扩展的节点的估价值小。

(3) 信息性。

A* 算法的搜索效率主要取决于启发函数 $h(n)$，在满足 $h(n)\leqslant h^*(n)$的前提下，$h(n)$的值越大越好。$h(n)$的值越大，表明它携带的与求解问题相关的启发信息越多，搜索过程就会在启发信息指导下朝着目标节点逼近，少走弯路，提高搜索效率。

设 $f_1(x)$与 $f_2(x)$是对同一问题的两个估价函数：

$$f_1(x)=g_1(x)+h_1(x)$$

$$f_2(x)=g_2(x)+h_2(x)$$

A_1^* 与 A_2^* 分别是以 $f_1(x)$与 $f_2(x)$为估价函数的 A* 算法，且设对所有非目标节点 x 均有

$$h_1(x)<h_2(x)$$

在此情况下，将证明 A_1^* 扩展的节点数不会比 A_2^* 扩展的节点数少，即 A_2^* 扩展的节点集是 A_1^* 扩展的节点集的子集。可用归纳法证明如下。

设 K 表示搜索的深度。当 $K=0$ 时，结论显然成立。因为若初始状态就是目标状态，则 A_1^* 与 A_2^* 都无须扩展任何节点。若初始状态不是目标状态，它们都要对初始节点进行扩展，此时 A_1^* 与 A_2^* 扩展的节点数是相同的。

设当搜索树的深度为 $K-1$ 时结论成立，即凡 A_2^* 扩展了的前 $K-1$ 代节点，A_1^* 也都扩展了。此时，只要证明 A_2^* 扩展的第 K 代的任一节点 x_k 也被 A_1^* 扩展就可以了。

由假设可知，A_2^* 扩展的前 $K-1$ 代节点 A_1^* 也都扩展了。因此在 A_1^* 搜索树中有一条从初始节点 S_0 到 x_k 的路径，其费用不会比 A_2^* 搜索树中从 S_0 到 x_k 的费用更大，即

$$g_1(x_k)\leqslant g_2(x_k)$$

假设 A_1^* 不扩展节点 x_k，就表示 A_1^* 能找到另一个具有更小估价值的节点进行扩展并找到最优解。此时有

$$f_1(x_k)\geqslant f^*(S_0)$$

即

$$g_1(x_k)+h_1(x_k)\geqslant f^*(S_0)$$

对上述不等式应用如下关系式

$$g_1(x_k) \leqslant g_2(x_k)$$

得到

$$h_1(x_k) \geqslant f^*(S_0) - g_2(x_k)$$

这与最初的假设 $h_1(x) < h_2(x)$ 矛盾。

由此可得出“A_1^* 扩展的节点数不会比 A_2^* 扩展的节点数少”这一结论是正确的，即启发函数所携带的启发性信息越多，搜索时扩展的节点数越少，搜索效率越高。

3）A* 算法应用举例

例 3.10　传教士和野人问题（简称 MC 问题）。设在河的左岸有三个野人、三个传教士和一条船，传教士想用这条船把所有的野人运到对岸，但是受以下条件的约束：

一是传教士和野人都会划船，但每次船上至多可载两个人。

二是在河的任一岸，如果野人数目超过传教士数目，传教士就会被野人吃掉。

如果野人会服从任何一次过河安排，请规划一个确保传教士和野人都能过河，且没有传教士被野人吃掉的安全过河计划。用 A* 算法解决该问题。

解：用 m 表示左岸的传教士人数，c 表示左岸的野人数，b 表示左岸的船数，用 (m, c, b) 表示问题的状态。

对于 A* 算法，首先需要确定估价函数。设 $g(n)=d(n)$，$h(n)=m+c-2b$，则有

$$f(n) = g(n) + h(n) = d(n) + m + c - 2b$$

式中，$d(n)$ 为节点的深度。通过分析可知，$h(n) \leqslant h^*(n)$，满足 A* 算法的限制条件。

MC 问题的搜索图如图 3.15 所示。在该图中，每个节点旁边还标出了该节点的 h 值和 f 值。

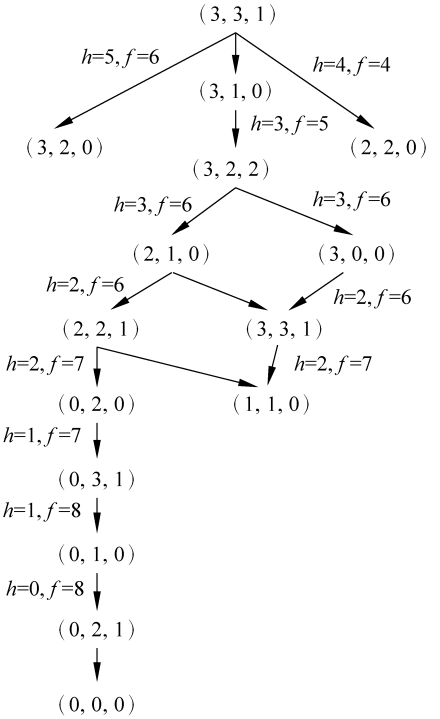


图 3.15　传教士和野人问题的搜索图

3.3 博弈树的启发式搜索

3.3.1 概述

博弈是一类富有智能行为的竞争活动，如下棋、打牌、战争等。博弈可以分为双人完备信息博弈和机遇性博弈。所谓双人完备信息博弈，就是两位选手对垒，轮流走步，每一方不仅知道对方已经走过的棋步，而且还可以估计出对方未来的走步。对弈的结果是一方赢，另一方输，或者是双方和局。这类博弈的实例有象棋、围棋等。所谓机遇性博弈，是指存在不可预测性的博弈，如掷币等。对机遇性博弈，由于不具备完备信息，因此本节不讨论。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的过程中，双方都希望自己能获胜。因此，当任何一方走步时，都是选择对自己最为有利，而对另一方最为不利的行动方案。假设博弈的一方为 MAX，另一方为 MIN。在博弈过程中的每一步，可供 MAX 和 MIN 选择的行动方案都可能有很多种。从 MAX 方的观点来看，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由自己决定的；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，任何一个方案都可能被 MIN 选中，MAX 必须防止那种对自己最为不利的情况的发生。

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/或树称为博弈树。在博弈树中，那些下一步该 MAX 走步的节点称为 MAX 节点，而下一步该 MIN 走步的节点称为 MIN 节点。博弈树具有如下特点：

(1) 博弈的初始状态是初始节点。

(2) 博弈树中的“或”节点和“与”节点逐层交替出现。

(3) 整个博弈过程始终站在某一方的立场上。所有能使自己一方胜利的终局都是本原问题，相应的节点是可解节点；所有使对方获胜的终局都是不可解节点。例如，站在 MAX 方，所有能使 MAX 方获胜的节点都是可解节点，所有能使 MIN 方获胜的节点都是不可解节点。

3.3.2 极大极小过程

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

为了计算非叶节点的值，必须从叶节点向上倒推。对于 MAX 节点，由于 MAX 方总是选择估值最大的走步，因此，MAX 节点的倒推值应该取其后继节点估值的最大值。对于 MIN 节点，由于 MIN 方总是选择使估值最小的走步，因此 MIN 节点的倒推值应取其后继节点估值的最小值。这样一步一步地计算倒推值，直至求出初始节点的倒推值为止。由于我们是站在 MAX 立场上，因此应该选择具有最大倒推值的走步。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 3.11　一字棋游戏。设有一个 3 行 3 列的棋盘，如图 3.16 所示，两个棋手轮流走步，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 MIN 方先走步。

解：为了对叶节点进行静态估值，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 MIN、MAX 都是胜负未定局，则

$$e(P)=e(+P)-e(-P)$$

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，对于如图 3.17 所示的棋局有估价函数值为：

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

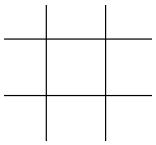


图 3.16　一字棋棋盘

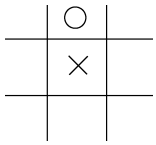


图 3.17　棋局 1

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 3.18 所示的棋局可以认为是同一个棋局，这样能大大减少搜索空间。图 3.19 给出了第一招走棋后生成的博弈树。图中叶节点下面的数字是该节点的静态估值，非叶节点旁边的数字是计算出的倒推值。从图中可以看出，对 MAX 来说，S_3 是一招最好的走棋，它具有较大的倒推值。

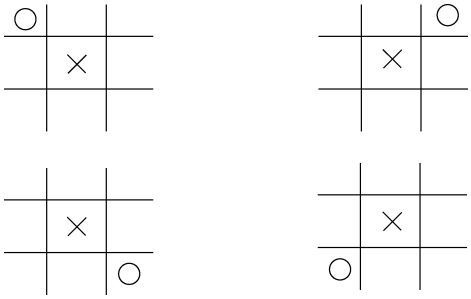


图 3.18　对称棋局的例子

3.3.3　α-β 剪枝

上述极大极小过程是先生成与/或树，然后再计算各节点的估值，这种生成节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节点，因此搜索效率低。如果能在生成节点的同时对节点进行估值，从而可以剪去一些没用的分枝，这种技术称为 α-β 剪枝过程。

α-β 剪枝的方法如下：

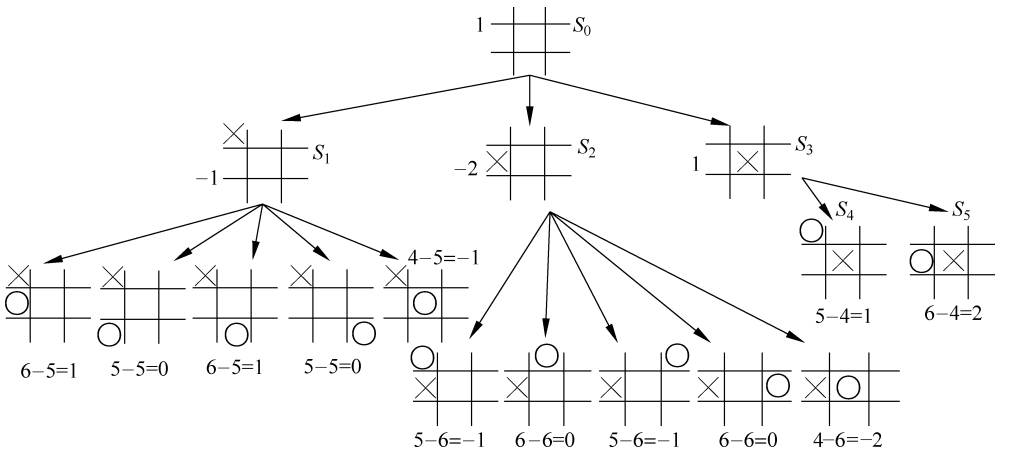


图 3.19　一字棋的极大极小搜索

(1) MAX 节点的 α 值为当前子节点最大倒推值。

(2) MIN 节点的 β 值为当前子节点最小倒推值。

α-β 剪枝的规则如下：

(1) 任何 MAX 节点 n 的 α 值大于或等于它先辈节点的 β 值，则 n 以下的分枝可停止搜索，并令节点 n 的倒推值为 α。这种剪枝称为 β 剪枝。

(2) 任何 MIN 节点 n 的 β 值小于或等于它先辈节点的 α 值，则 n 以下的分枝可停止搜索，并令节点 n 的倒推值为 β。这种剪枝称为 α 剪枝。

下面来看一个 α-β 剪枝的具体例子，如图 3.20 所示。其中，最下面一层端节点下面的数字是假设的估值。

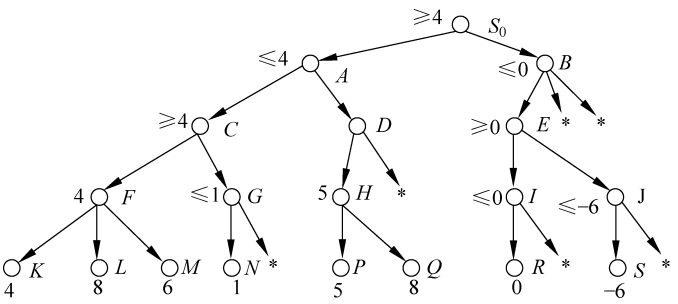


图 3.20　α-β 剪枝的例子

在图 3.20 中，由节点 K、L、M 的估值推出节点 F 的倒推值为 4，即 F 的 β 值为 4，由此可推出节点 C 的倒推值(≥4)。记 C 的倒推值的下界为 4，不可能再比 4 小，故 C 的 α 值为 4。由节点 N 的估值推出节点 G 的倒推值(≤1)，无论 G 的其他子节点的估值是多少，G 的倒推值都不可能比 1 大。事实上，随着子节点的增多，G 的倒推值只可能是越来越小。因此，1 是 G 的倒推值的上界，所以 G 的值为 1。另外，已经知道 C 的倒推值(≥4)，G 的其他子节点又不可能使 C 的倒推值增大。因此，对 G 的其他分枝不必再进行搜索，这就相当于把这些分枝剪去。由 F、G 的倒推值可推出节点 C 的倒推值为 4，再由 C 可推出节点 A 的

倒推值(≤4),即 A 的 β 值为 4。另外,由节点 P、Q 推出的节点 H 的倒推值为 5,此时可推出 D 的倒推值(≥5),即 D 的 α 值为 5。此时,D 的其他子节点的倒推值无论是多少都不能使 D 及 A 的倒推值减少或者增加,所以 D 的其他分枝被剪去,并可确定 A 的倒推值为 4。用同样的方法可推出其他分枝的剪枝情况,最终推出 S_0 的倒推值为 4。

3.4　实践:A* 算法实现最优路径规划


在线视频


本章彩图

启发式探索是利用问题拥有的启发信息来引导搜索,达到减少探索范围、降低问题复杂度的目的。A* 寻路算法是启发式探索的一个典型实践,在寻路搜索的过程中,给每个节点绑定了一个估计值(即启发式),在对节点的遍历过程中采取估计值优先原则,估计值更优的节点会被优先遍历。

3.4.1　A* 算法基本原理

A* 算法是一种有序搜索算法,其特点在于对估价函数的定义上。公式表示为:$f(n)=g(n)+h(n)$,其中,$f(n)$是从初始状态经由状态 n 到目标状态的代价估计,$g(n)$是在状态空间中从初始状态到状态 n 的实际代价,$h(n)$是从状态 n 到目标状态的最佳路径的估计代价。对于路径搜索问题,状态就是图中的节点,代价就是距离。

3.4.2　A* 算法搜索步骤

1. 算法步骤

(1) 设置地图大小,起点 S,终点 E,障碍集合 Blocklist。

(2) 添加起点 S 到 Openlist(待搜索集合)。

(3) 将 S 取出,添加到 Closelist(已搜索集合)。

(4) 查找 S 所有相邻节点,添加到 Openlist,并设置 S 为它们的父节点;以绿色初始节点右侧的灰色节点为例:$f(n)=g(n)+h(n)$。$g(n)=1$,绿色初始节点到该节点的移动步数;$h(n)=3$,灰色节点移动到红色终点的步数(曼哈顿距离),也可以使用欧氏距离。$f(n)=g(n)+h(n)=1+3=4$,其他相邻节点计算相同。如图 3.21 所示,曼哈顿距离向 4 个方向移动,距离公式为:

$$d=(x_2-x_1)+(y_2-y_1)$$

欧氏距离向 8 个方向移动,距离公式为:

$$d=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

(5) 选择 Openlist 中 f 值最小点,有两个分别为绿色右侧节点和绿色下方节点,将右侧节点添加至 Closelist,并设置绿色节点为其父节点,选择下方节点也可以,本例节点顺序右下左上,根据启发式规则相同,结果和搜索效率与选取顺序无关,如图 3.22 所示。

(6) 此时 Openlist 中 f 的最小值是 4,黄色相邻节点中只有上下节点可达,根据计算得到上下节点 f 值,此处 f 值为最小值,采用其他路径计算值均不小于 4,如图 3.23 所示。

因此,选取黄色节点下方 f 值小的节点添加至 Closelist,此时 Openlist 中 f 的最小值为 4,继续选取节点添加至 Openlist,如图 3.24 所示。

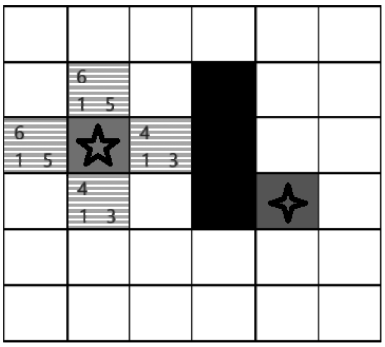


图 3.21 步骤(4)示意图

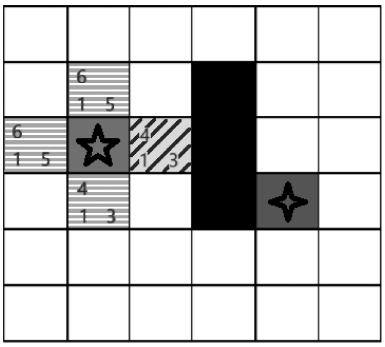


图 3.22 步骤(5)示意图

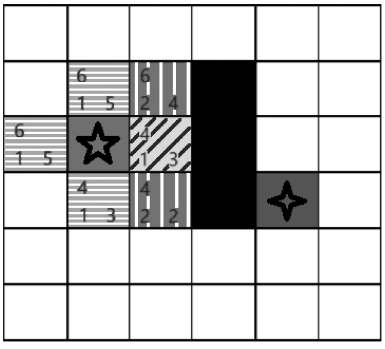


图 3.23 黄色相邻上下节点

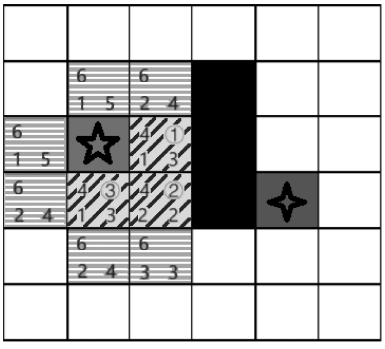


图 3.24 步骤(6)示意图

接下来算法进入相同方式迭代过程,具体步骤参见附录 B。

A* 算法最终执行结果如图 3.25 所示。

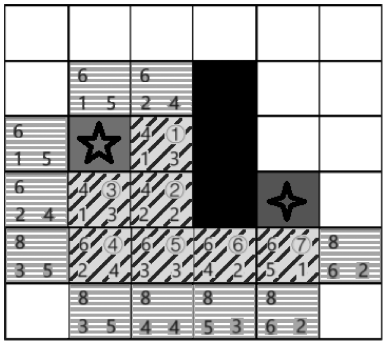


图 3.25 结果示意图

2. 路径搜索

开始从红色节点逆推,红色节点的父节点为⑦号节点,⑦号节点的父节点为⑥号节点,⑥号节点的父节点为⑤号节点,⑤号节点的父节点为②号节点,⑤号节点是搜索到②号节点时添加到 Openlist 中的,并且一直未被更新,②号节点的父节点为①号节点,最终的搜索路径为:起点-①-②-⑤-⑥-⑦-终点。

这里的搜索路径并不是最佳路径的唯一解，其中路径：起点-③-④-⑤-⑥-⑦和路径：起点-③-②-⑤-⑥-⑦都可以通过相应的算法求出，作为搜索的最佳路径，因为这些路径理论上是等同的，这里只以一种最佳路径作为演示。

对于上例演示的情况中存在无论如何重新计算 Openlist 中节点的 f 值都不会更小，也就是无法进行更新操作，因此再举一个例子，演示更新搜索。

现在考虑可以 8 个方向搜索，但是斜向搜索需要步数为 4。

选择 Openlist 中 f 值最小的节点，选择了右侧 f 值为 4 的节点，此时计算右侧 f 值为 4 的节点的相邻节点的 f 值，如图 3.26 所示。其中左侧绿色起始点已经添加进 Closelist 中，右侧三个为黑色节点，因此不考虑这 4 个节点，其他节点均已存在于 Openlist 中，现对其 f 值进行更新。

(1) 先看左上角的相邻节点，通过黄色节点到达该节点，$g(n) = 5$，$h(n)$不变，$f(n)$反而更大了，因此不更新。左下角节点同理。

(2) 上方节点，通过黄色节点计算 $g(n) = 2$，$h(n)$不变，$f(n) = 6 < 8$。所以，更新这个节点的 f 值，并将其父节点修改为黄色节点。下方居中节点同理，如图 3.27 所示。

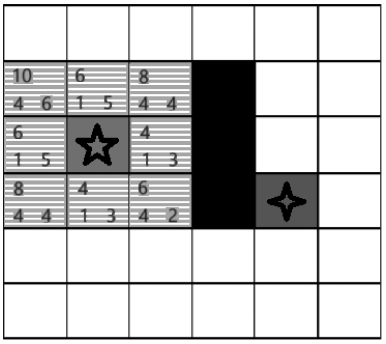


图 3.26　计算相邻节点 f 值

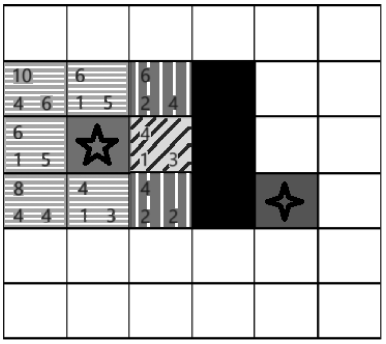


图 3.27　更新父节点

3.4.3　使用 Python 实现上述流程

(1) 绘制地图全貌：起点、终点、障碍和可通行节点。

```
for (x, y) in mymap.generate_cell(CELL_WIDTH, CELL_HEIGHT):
    if (x,y) in bl_pix:
        #绘制黑色的障碍物单元格,并留出 2 个像素的边框
        pygame.draw.rect(screen, Color.BLACK.value, ((x + BORDER_WIDTH,y + BORDER_WIDTH),
(CELL_WIDTH - 2 * BORDER_WIDTH, CELL_HEIGHT - 2 * BORDER_WIDTH)))
    else:
        # 绘制绿色的可通行单元格,并留出 2 个像素的边框
        pygame.draw.rect(screen, Color.GREEN.value, ((x + BORDER_WIDTH,y + BORDER_WIDTH),
(CELL_WIDTH - 2 * BORDER_WIDTH, CELL_HEIGHT - 2 * BORDER_WIDTH)))
#绘制起点和终点
pygame.draw.circle(screen, Color.BLUE.value, (pix_sn[0] + CELL_WIDTH//2, pix_sn[1] + CELL_
HEIGHT//2), CELL_WIDTH//2 - 1)
pygame.draw.circle(screen, Color.RED.value, (pix_en[0] + CELL_WIDTH//2, pix_en[1] + CELL_
HEIGHT//2), CELL_WIDTH//2 - 1)
```

(2) 获得相邻节点。

```
def get_neighbor(self, cnode):
    #offsets = [(-1,1),(0,1),(1,1),(-1,0),(1,0),(-1,-1),(0,-1),(1,-1)]
    offsets = [(-1, 0), (1, 0), (0, 1), (0, -1)]
    nodes_neighbor = []
    x, y = cnode.pos[0], cnode.pos[1]
    for os in offsets:
        x_new, y_new = x + os[0], y + os[1]
        pos_new = (x_new, y_new)
        #判断是否在地图范围内,超出范围跳过
        if x_new < 0 or x_new > self.mapsize[0] - 1 or y_new < 0 or y_new > self.mapsize[1]:
            continue
        nodes_neighbor.append(Node(pos_new))
    return nodes_neighbor
```

(3) 采用曼哈顿或欧氏距离计算 $h(n)$。

```
#gx_f2n = math.sqrt((father.pos[0] - self.pos[0]) ** 2 + (father.pos[1] - self.pos[1])
** 2)
gx_f2n = abs(father.pos[0] - self.pos[0]) + abs(father.pos[1] - self.pos[1])
gvalue = gx_f2n + gx_father
```

(4) 更新 f 值。

```
def update_fx(self, enode, father):
    gvalue, fvalue = self.compute_fx(enode, father)
    if fvalue < self.fvalue:
        self.gvalue, self.fvalue = gvalue, fvalue
        self.father = father
```

(5) 绘制搜索最优路径

```
for (x, y) in mymap.generate_cell(CELL_WIDTH, CELL_HEIGHT):
    if (x,y) in rl_pix and (x,y)!= (pix_sn[0],pix_sn[1]) and (x,y)!= (pix_en[0],pix_en[1]):
        pygame.draw.rect(screen, Color.GREY.value, ((x + BORDER_WIDTH,y + BORDER_WIDTH),
(CELL_WIDTH - 2 * BORDER_WIDTH, CELL_HEIGHT - 2 * BORDER_WIDTH)))
```

算法对应示例程序参照附录 B。

3.4.4 最优路径规划

尝试采用 A* 算法对图 3.28 所示的地图进行最优路径规划。

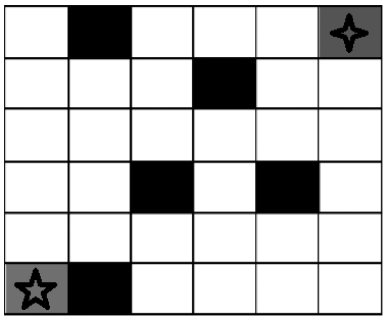


图 3.28　路径规划图

3.5　习题

1. 什么是搜索？有哪两类不同的搜索方法？两者的区别是什么？

2. 什么是状态空间？用状态空间法表示问题时，什么是问题的解？什么是最优解？最优解唯一吗？

3. 试写出状态空间图的一般搜索过程。在搜索过程中 OPEN 表和 CLOSED 表的作用分别是什么？有何区别？

4. 什么是盲目搜索？主要有哪几种盲目搜索策略？

5. 什么是宽度优先搜索？什么是深度优先搜索？有何不同？

6. 什么是与树？什么是或树？什么是与/或树？什么是可解节点？什么是解树？

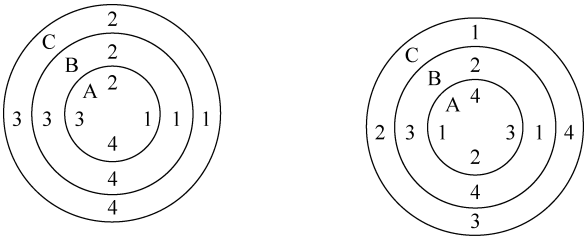
7. 有一个农夫带一只狼、一只羊和一筐青菜从河的左岸乘船到右岸，但受到下列条件的限制：

(1) 船太小，农夫每次只能带一样东西过河。

(2) 如果没有农夫看管，则狼要吃羊，羊要吃菜。

请设计一个过河方案，使得羊和菜都能不受损失地过河，画出相应的状态空间图。

8. 圆盘问题。设有大小不等的 3 个圆盘 A、B、C 套在一根轴上，每个盘上都标有数字 1、2、3、4，并且每个圆盘都可以独立地绕轴做逆时针转动，每次转动 90°，其初始状态 S_0 和目标状态 S_g 如图 3.29 所示，请用广度优先搜索和深度优先搜索求出从 S_0 到 S_g 的路径。



(a) 初始状态S_0　　(b) 目标状态S_g

图 3.29　题 8 示意图

9. 设有如图 3.30 所示的与/或树,请分别用与/或树的广度优先搜索和深度优先搜索求出解树。

10. 设有如图 3.31 所示的与/或树,请分别按和代价法及最大代价法求解树的代价。

11. 设有如图 3.32 所示的博弈树,其中最下面的数字是假设的估值。请对该博弈树做如下工作:

(1) 计算各节点的倒推值。

(2) 利用 α-β 剪枝技术剪去不必要的分枝。

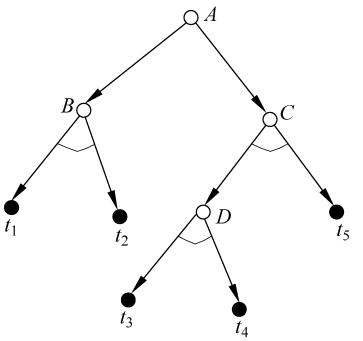


图 3.30 题 9 示意图

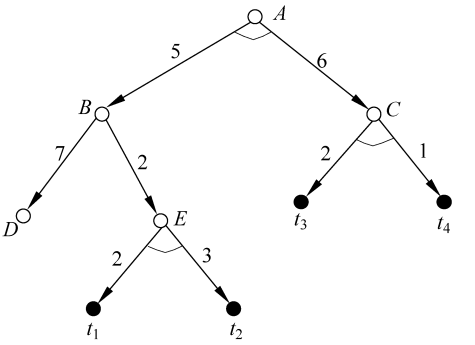


图 3.31 题 10 示意图

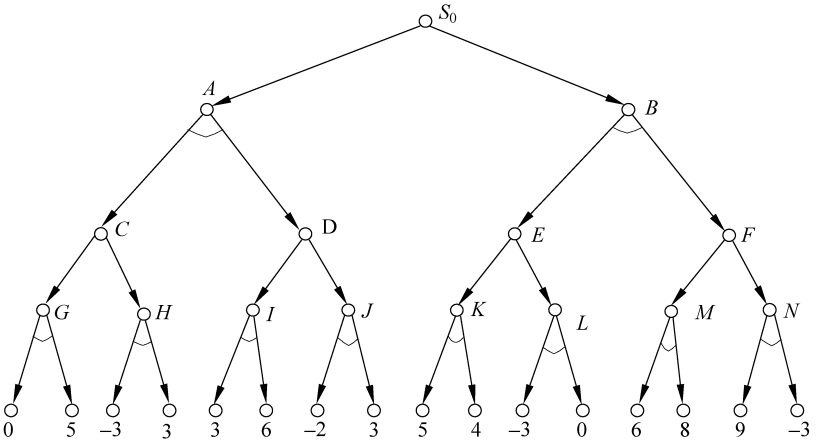


图 3.32 题 11 示意图