

第 3 章



搜索技术

从工程应用的角度来看,开发人工智能技术的一个主要目的就是解决非平凡问题,即难以用常规技术(数值计算、数据库应用等)直接解决的问题。这些问题的求解依赖于问题本身的描述和应用领域相关知识的应用方式。广义地说,人工智能问题都可以看作是一个问题求解的过程。因此问题求解是人工智能的核心问题,其要求是在给定条件下寻求一个能解决某类问题且能在有限步内完成的算法。

按解决问题所需的领域的特有知识的多寡,问题求解系统可以划分为两大类,即知识贫乏系统和知识丰富系统。前者必须依靠搜索技术去解决问题,后者则需要求助于推理技术。

搜索直接关系到智能系统的性能与运行效率,因此美国人工智能专家尼尔逊在 1974 年把它列为人工智能研究的 5 个核心问题(知识的模型化和表示;常识性推理;演绎和问题求解;启发式搜索;人工智能系统和语言)之一。

现在,搜索技术渗透在各种人工智能系统中,可以说没有哪一种人工智能系统应用不到搜索方法。在专家系统、自然语言理解、自动程序设计、模式识别、机器人学、信息检索和博弈等领域都广泛使用了搜索技术。

本章首先讨论搜索的有关概念,然后着重介绍盲目搜索、启发式搜索和与或图的搜索,最后讨论博弈问题的智能搜索算法以及高级搜索技术。

3.1 搜索原理概述

3.1.1 搜索的基本概念及其方法分类

人工智能所要解决的问题大部分是结构不良或非结构化的问题,对这样的问题一般不存在成熟的求解算法可供利用,而只能是利用已有的知识一步步地摸索着前进,在此过

程中,存在着如何寻找可用知识的问题,即如何确定推理路线,使其付出的代价尽可能少,而问题又能得到较好的解决。如在正向演绎推理中,对已知的初始事实,需要在知识库中寻找可使用的知识,这就存在按何种路线进行寻找的问题。另外,可能存在多条路线都可实现对问题的求解,这就又存在按哪一条路线进行求解以获得较高的运行效率的问题。像这样根据问题的实际情况不断寻找可利用的知识,从而构造一条代价较少的推理路线,使问题得到圆满解决的过程称为搜索。

在人工智能中,即使对于结构性能较好,理论上有算法可依的问题,由于问题本身的复杂性以及计算机在时间、空间上的局限性,有时也需要通过搜索来求解。例如在博弈问题中,计算机为了取得胜利,需要在每走一步棋之前,考虑所有的可能性,然后选择最佳走步。找到这样的算法并不困难,但计算机却要付出惊人的时空代价。

搜索分为盲目搜索和启发式搜索。盲目搜索是按预定的控制策略进行搜索,在搜索过程中获得的中间信息不用来改进控制策略。由于搜索总是按预先规定的路线进行,没有考虑到问题本身的特性,所以这种搜索具有盲目性,效率不高,不便于复杂问题的求解。启发式搜索是在搜索中加入了与问题有关的启发性信息,用以指导搜索朝着最有希望的方向前进,加速问题的求解过程并找到最优解。

显然,启发式搜索优于盲目搜索。但由于启发式搜索需要具有与问题本身特性有关的信息,而这并非对每一类问题都可方便地抽取出来,因此盲目搜索仍不失为一种应用较多的搜索策略。

3.1.2 搜索空间和搜索算法

在人工智能中,搜索问题一般包括两个重要问题:搜索什么,在哪里搜索。搜索什么通常指的就是目标,而在哪里搜索就是指搜索空间。搜索空间通常是指一系列状态的集合,因此也称为状态空间。与通常的搜索空间不同,人工智能中大多数问题的状态空间在问题求解之前不是全部知道的。所以,人工智能中的搜索可以分成两个阶段:状态空间的生成阶段和在该状态空间中对目标状态的搜索。由于一个问题的整个状态空间可能会非常大,在搜索之前生成整个空间会占用太大的存储空间。为此,状态空间一般是逐渐扩展的,“目标”状态是在每次扩展的时候进行判断的。

一般来说,搜索方法可以根据是否使用启发式信息分为盲目搜索和启发式搜索方法,也可以根据搜索空间的表示方式分为状态空间搜索方法和与或图搜索方法。状态空间搜索是用状态空间法来求解问题所进行的搜索,与或图搜索是指用问题归约方法来求解问题时所进行的搜索。状态空间法和问题归约法是人工智能中最基本的两种问题表示方法。

在搜索问题中,主要的工作是找到正确的搜索算法。搜索算法一般可以通过下面4个标准来评价。

- (1) 完备性: 如果存在一个解答,该策略是否保证能够找到?
- (2) 时间复杂性: 需要多长时间可以找到解答?
- (3) 空间复杂性: 执行搜索需要多少存储空间?
- (4) 最优性: 如果存在不同的几个解,该算法是否可以发现最高质量的解?

搜索算法制定了状态空间或问题空间扩展的方法,也决定了状态或问题的访问顺序。不同的搜索算法在人工智能领域的命名也不同。例如,在状态空间为一棵树的问题上有两种基本的搜索算法。如果首先扩展根结点,然后扩展根结点生成的所有结点,再扩展这些结点的后继,如此反复下去,则这种算法称为宽度优先搜索,另一种方法是,在树的最深一层的结点中扩展一个结点,只有当搜索遇到一个死亡结点(非目标结点并且是无法扩展的结点)的时候,才返回上一层选择其他的结点搜索。这种算法称为深度优先搜索。然而,无论是宽度优先搜索还是深度优先搜索,结点的遍历顺序都是固定的,即一旦搜索空间给定,结点遍历的顺序就固定了。这种类型的遍历称为“确定”的,也就是盲目搜索。而对于启发式搜索,在计算每个结点的参数之前无法确定先选择哪个结点扩展,这种搜索一般称为“非确定”的。

3.2 盲目搜索策略

下面介绍几种常用的盲目搜索方法,首先介绍生成再测试法这个一般的算法框架,然后介绍性能较好的迭代加深算法,最后介绍等代价搜索算法。

3.2.1 生成再测试法

最简单的盲目搜索方法是“生成再测试”方法。该方法的算法描述如下。

```
Procedure Generate&Test
  Begin
    Repeat
      生成一个新的状态,称为当前状态;
    Until 当前状态 = 目标;
  End
```

显然,上述算法在每次 Repeat-Until 循环中都生成一个新的状态,并且只有当新的状态等于目标状态的时候才退出。在该算法中最重要的部分是新状态的生成。如果生成的新状态不可扩展,则该算法应该停止,为了简单起见,在上述算法中省略了这一部分。

宽度优先搜索算法和深度优先搜索算法可以看做是生成再测试方法的两个具体版本。它们的区别是生成新状态的顺序不同。假设问题空间是一棵树,则深度优先搜索总是优先生成并测试深度增加的结点,而宽度优先搜索则总是优先搜索同一深度的结点。深度优先搜索和宽度优先搜索具有两个主要的特点:①它们只能用于求解搜索空间为树的问题,如果用于处理存在环的搜索空间,则它们都有可能陷入无限循环而无法停止;②宽度优先搜索能够保证找到路径长度最短的解(最优解),而深度优先搜索无法保证这一点。第一个特点是我们不希望的,最好优先搜索算法可以弥补此缺点。对于第二个特点,应该认识到宽度优先搜索的优势是以巨大的存储为代价的。假设问题空间中每个结点平均有 b 个子结点,目标结点的深度为 d ,则宽度优先搜索在最坏情况下需要存储 $O(bd)$ 个结点;相对而言,深度优先搜索则仅需存储 $O(d)$ 个结点。那么,能否设计一种搜索方法结合宽度优先搜索保证最优性的优势与深度优先搜索在存储上的优势?下面介

绍的迭代加深搜索方法就具有此性质。

3.2.2 迭代加深搜索

对于深度 d 比较大的情况,深度优先搜索可能沿着一个不含目标结点的分枝探寻很长时间,在找不到解的同时还浪费了资源。一种较好的方法是对搜索的深度进行控制,这就是有界深度优先搜索方法的主要思想。有界深度优先搜索过程总体上按深度优先算法方法进行,但对搜索深度给出一个深度限制 d_m ,当深度达到了 d_m 的时候,如果还没有找到解答,就停止对该分枝的搜索,换到另外一个分枝进行搜索。

对于有界深度优先搜索策略,有以下几点需要说明。

(1) 在有界深度优先搜索算法中,深度限制 d_m 是一个很重要的参数。当问题有解,且解的路径长度小于或等于 d_m 时,则该算法一定能够找到解。但是和深度优先搜索一样,这并不能保证最先找到的是最优解,此情况下的有界深度而搜索是完备的但不是最优的。但是当 d_m 取得太小,而解的路径长度大于 d_m 时,则搜索过程中就找不到解,此情况下的搜索过程甚至是不完备的。

(2) 深度限制 d_m 不能太大。当 d_m 太大时,搜索过程会产生过多的无用结点,既浪费了计算机资源,又降低了搜索效率。

(3) 有界深度优先搜索的主要问题是深度限制 d_m 的选取。该值也被称为状态空间的直径,如果该值设置的比较合适,则会得到比较有效的有界深度优先搜索。但是对于很多问题,预先无法知道该值到底为多少,只有在该问题求解完成后才能确定出深度限制 d_m ,而那是确定的 d_m 对搜索算法没有意义。为了解决上述问题,可采用如下的改进办法:先任意设定一个较小的数作为 d_m ,然后按有界深度算法搜索,若在此深度限制内找到了解,则算法结束;如在此限制内没有找到问题的解,则增大深度限制 d_m ,继续搜索。此方法称为迭代加深搜索。

迭代加深搜索是一种回避选择选择最优深度限制问题的策略,它是试图尝试所有可能的深度限制:深度首先为 0,然后为 1,然后为 2……一直进行下去。如果初始深度为 0,则该算法只生成根结点,并检测它。如果根结点不是目标,则深度加 1,通过典型的深度优先搜索算法,生成深度为 1 的树。同样,当深度限制为 m 时,它将生成深度为 m 的树。

迭代加深搜索过程描述如下:

```
Procedure - Iterative - deeping
Begin
  For d = 1 to ∞ Do
    Begin
      从初始结点执行深度限制为 d 的有界深度优先搜索;
      如果找到解,则过程结束;
      如果本次迭代中访问的所有结点的深度都小于 d,则过程结束;
    End
  End
```

通过分析可以发现,迭代加深搜索看起来会很浪费资源,因为它在深度限制为 $d+1$

的迭代过程中将重复搜索深度限制为 d 的迭代访问过的结点。然而对于很多问题,这种多次的扩展负担实际上很小,直觉上可以想象,如果一棵树的分枝系数很大,几乎所有的结点都在最底层上,则对于上面各层结点扩展多次对整个系统的影响不是很大。

宽度优先搜索,深度优先搜索,深度优先搜索和迭代加深搜索都是生成再测试算法的具体版本。迭代加深搜索结合了宽度优先搜索和深度优先搜索的优点。表 3.1 总结了宽度优先搜索、深度优先搜索、有界深度搜索和迭代加深搜索的主要特点。

表 3.1 几个盲目搜索算法的特点对比

标 准	宽度优先搜索	深度优先搜索	有界深度搜索	迭代加深搜索
时间	b^d	b^m	b^l	b^d
空间	b^d	b^m	b^l	b^d
最优	是	否	否	是
完备	是	否	如果 $l > d$, 是	是

注: b 是分枝系数, d 是解答的深度, m 是搜索树的最大深度, l 是深度限制。

3.3 启发式搜索

前文讨论的各种搜索方法都是按事先规定的路线进行搜索的,没有用到问题本身的特征信息,具有较大的盲目性,产生的无用结点较多,搜索空间较大,效率不高。如果能够利用问题自身的一些特征信息来指导搜索过程,则可以缩小搜索范围,提高搜索效率。

启发式搜索通常用于两种不同类型的问题,正向推理和反向推理。正向推理一般用于状态空间的搜索。在正向推理中,推理是从预先定义的初始状态出发向目标状态执行。反向推理一般用于问题归约中。在反向推理中,推理是从给定的目标状态向初始状态执行。在前一类的使用启发式函数的搜索算法中,包括通常所谓的 OR 图算法和最好优先算法,以及根据启发式函数的不同而得到的其他的一些算法,如 A^* 算法等。另一方面,启发式反向推理算法通常称为 AND-OR 图搜索算法, AO^* 算法就是其中一种算法。

3.3.1 启发性信息和评估函数

如果在选择结点时能充分利用与问题有关的特征信息,估计结点的重要性,就能在搜索时选择重要性较高的结点,以利于求得最优解。一般把这个过程称为启发式搜索。“启发式”实际上代表了“大拇指准则”,即在大多数情况下是成功的,但不能保证一定成功的准则。

与被解问题的某些特征有关的控制信息(如解的出现规律、解的结构特征等)称为搜索的启发信息,它反映在评估函数中。评估函数的作用是估计待扩展的各结点在问题求解中的价值,即评估结点的重要性。

评估函数 $f(x)$ 的定义为从初始结点 S_0 出发,约束地经过结点 x 到达目标结点 S_g 的所有路径中最小路径代价的估计值。其一般形式为 $f(x) = g(x) + h(x)$, 其中, $g(x)$ 表示从初始结点 S_0 到结点 x 的实际代价, $h(x)$ 表示从 x 到目标结点 S_g 的最优路径的

评估代价,它体现了问题的启发式信息,其形式要根据问题的特性确定, $h(x)$ 被称为启发式函数。启发式方法把问题状态的描述转换成了对问题解决程度的描述,并用评估函数的值来表示。

3.3.2 最好优先搜索算法

下面介绍一种称为最好优先搜索算法(Best-First Search)的算法框架,该方法能处理图。为了处理环,最好优先搜索算法用 OPEN 表和 CLOSED 表记录状态空间中那些被访问过的所有状态。这两个表中的结点及它们关联的边构成了状态空间的一个子图,称为搜索图。OPEN 表存储一些结点,其中每个结点 n 的启发式函数值已经计算出来,但是 n 还没有被“扩展”。CLOSED 表存储一些结点,其中每个结点已经被扩展。该类算法每次迭代从 OPEN 表中取出一个较优的结点 n 进行扩展,将 n 的每个子结点根据情况放入 OPEN 表。算法循环直到发现目标结点或者 OPEN 表为空。算法中的每个结点带有一个父指针,该指针用于合成解路径。

最好优先搜索算法的具体描述如下:

```

Procedure Graph - Search
Begin
    建立只含初始结点  $S_0$  的搜索图  $G$ , 计算  $f(S_0)$ ; 将  $S_0$  放入 OPEN 表; 将 CLOSED 表初始化为空
    While OPEN 表不空 Do
        Begin
            从 OPEN 表中取出  $f(n)$  值最小的结点  $n$ , 将  $n$  从 OPEN 表中删除并放入 CLOSED 表
            If  $n$  是目标结点 Then 根据  $n$  的父指针指出从  $S_0$  到  $n$  的路径, 算法停止
        Else
            Begin
                扩展结点  $n$ 
                If 结点  $n$  有子结点
                Then
                    Begin
                        (1) 生成  $n$  的子结点集合  $\{m_i\}$  把  $m_i$  作为  $n$  的子结点加入到  $G$  中, 并计算  $f(m_i)$ 
                        (2) If  $m_i$  未曾在 OPEN 和 CLOSED 表中出现, Then 将它们配上刚计算过的  $f$  值, 将  $m_i$  的父指针指向  $n$ , 并把它放入 OPEN 表
                        (3) If  $m_i$  已经在 OPEN 表中, Then 该结点一定有多个父结点, 在这种情况下, 比较  $m_i$  相对于  $n$  的  $f$  值和  $m_i$  相对于其原父指针指向的结点的  $f$  值, 若前者不小于后者, 则不做任何更改; 否则将  $m_i$  的  $f$  值更改为  $m_i$  相对于  $n$  的  $f$  值,  $m_i$  的父指针更改为  $n$ 
                        (4) If  $m_i$  已经在 CLOSED 表中, Then 该结点同样也有多个父结点, 在这种情况下, 比较  $m_i$  相对于  $n$  的  $f$  值和  $m_i$  相对于其原父指针指向的结点的  $f$  值, 如果前者不小于后者, 则不作任何更改; 否则将  $m_i$  从 CLOSED 表移到 OPEN 表, 置  $m_i$  的父指针指向  $n$ 
                        (5) 按  $f$  值从小到大的次序, 对 OPEN 表中的结点进行重新排序
                    End
                End
            End
        End
    End
End

```

上述搜索算法生成一个明确的图 G (称为搜索图) 和一个 G 的子集 T (称为搜索树), 图 G 中的每一个结点也在树 T 上, 搜索树是由结点的父指针来确定的。 G 中的每一个结

点(除了初始结点 S_0)都有一个指向 G 中一个父辈结点的指针。该父辈结点就是那个结点在 T 中的唯一父辈结点。算法中(3)(4)步保证对每一个扩展的新结点,其父指针的指向是已经产生的路径中代价最小的。

3.3.3 贪婪优先搜索算法

最好优先搜索算法是一个通用的算法框架。如果将该框架中的 $f(n)$ 实例化为 $f(n)=h(n)$,则得到一个具体的算法,称为贪婪最好优先搜索算法。可以看出,GBFS 算法在判断是否优先扩展一个结点 n 时仅以 n 的启发值为依据。 n 的启发值越小,表明了从 n 到目标结点的代价越小,因而 GBFS 算法沿着 n 所在的分枝搜索就越可能发现目标结点。因此,GBFS 算法一般可以较快地计算出问题的解。

但是,GBFS 算法得出的解是否是最优的?考虑如下情况,OPEN 表中有两个结点 n 和 n' ,其中 $g(n)=5, h(n)=0, g(n')=3, h(n')=1$,而且 n 和 n' 的 h 值分别是它们与目标结点的真实距离,在此情况下,GBFS 将扩展 n 而不是 n' 。显然,经过 n 发现的解的代价高于经过 n' 发现的解的代价,所以 GBFS 返回的不是最优解。仔细分析最好优先算法的流程可以发现,当 $f(n)=h(n)$ 时,其中的步骤(3)和(4)将不会对 n 的信息做改变。与 GBFS 算法相对,假如最好优先搜索算法中的 $f(n)$ 被实例化为 $f(n)=g(n)$,则得到宽度优先搜索算法。读者可以在图的最短路径问题上将 $g(n)$ 定义为源结点到 n 的路径长度,分析此命题的正确性。

可以看出, $h(n)$ 影响算法发现解的速度, $g(n)$ 影响得到解的最优性,下面介绍的 A 算法和 A^* 是使用 $f(n)=g(n)+h(n)$ 的最好优先搜索算法,它们综合考虑了时间效率和解质量。其中, A^* 算法使用的 $h(n)$ 具有更严格的性质。

3.3.4 启发式搜索中的 A^* 算法及应用

如果最好优先搜索算法中的 $f(n)$ 被实例化为 $f(n)=g(n)+h(n)$,则称之为 A 算法。进一步细化,如果启发函数 h 满足对于任一结点 $n, h(n)$ 的值都不大于 n 到目标结点的最优代价,则称此类 A 算法为 A^* 算法。 A^* 算法在一些条件下能够保证找到最优解,即 A^* 算法具有最优性。下面首先以九宫图为例(图 3.1)介绍 A 算法的运行过程,然后对 A^* 算法最优性进行分析。

A 算法采用评估函数判断每个结点的重要性。在该算法运行的初始时刻,OPEN 表中只有初始结点,因此我们扩展它,得到图 3.1 中的第二层结点,将这些结点全部放入 OPEN 表。在第二次迭代过程中,A 算法选择 OPEN 表中具有最小 f 值为 $1+3=4$ 的结点扩展,得到第三层的 3 个结点,并将它们放入 OPEN 表。在第三次迭代中,A 算法选择 OPEN 表中 f 值为 $2+3=5$ 的结点进行扩展。在第四次迭代中,A 算法选择 OPEN 表中 f 值为 $2+3=5$ 的另一个结点进行扩展。在第五次迭代中,A 算法选择 OPEN 表中 f 值为 $3+2=5$ 的结点进行扩展。在第六次迭代中,A 算法选择 OPEN 表中 f 值为 $4+1=5$ 的结点进行扩展。在第七次迭代中,A 算法选择 OPEN 表中 f 值为 $5+0=5$ 的结点进行扩展。通过此例可以发现,A 算法相对于宽度优先搜索和深度优先搜索都具有优势。

但是,由于对启发函数 h 没有任何限制,A 算法不能保证找到最优解。经研究发现,

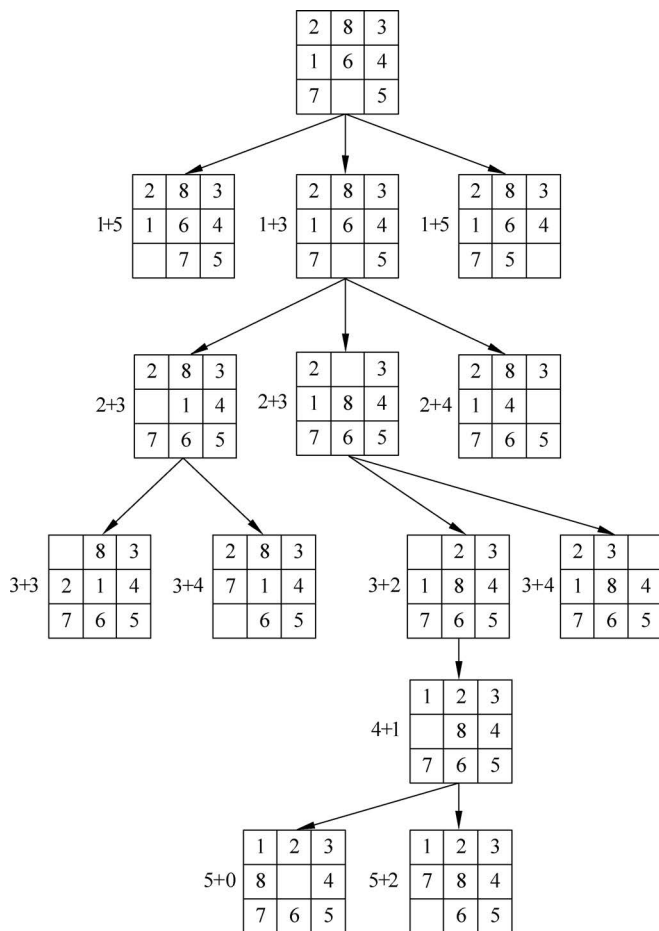


图 3.1 九宫格问题的全局择优搜索树

A 算法在如下 3 个条件成立时能够保证得到最优解：

- (1) 启发函数 h 对任一结点 n 都满足 $h(n)$ 不大于 n 到目标的最优代价。
- (2) 搜索空间中的每个结点具有有限个后继。
- (3) 搜索空间中每个有向边的代价均为正值。

为了表明此类 A 算法的重要性,将此类 A 算法称为 A^* 算法,称上述 3 个条件为 A^* 算法的运行条件。

对 h 的限制可以更为正式地表述如下: 令 h^* 是能计算出任意结点 n 到目标的最优代价的函数,称之为“完美启发函数”。如果 $\forall n: h(n) \leq h^*(n)$, 则称 h 为可采纳的启发函数,或者称 h 是可采纳的,或者简称为可纳的。此外,也引入函数 g^* , 它能计算从开始结点到任意结点的最优代价。定义估价函数: $f^*(n) = g^*(n) + h^*(n)$ 。这样 $f^*(n)$ 就是从起始结点出发经过结点 n 到达目标结点的最佳路径的总代价。

把估价函数 $f(n)$ 和 $f^*(n)$ 相比较, $g(n)$ 是对 $g^*(n)$ 的估价。 $h(n)$ 是对 $h^*(n)$ 的估价。在这两个估价中,尽管 $g(n)$ 容易计算,但它不一定就是从起始结点 S_0 到结点 n 的真正的最短路径的代价,很可能从初始结点 S_0 到结点 n 的真正最短路径还没有找到,所

以一般都有 $g(n) \geq g^*(n)$ 。但应注意, A^* 算法的步骤(3)和(4)保证了如果发现 n 的更好的 $g(n)$ 值, 则以此值作为 n 的最新的 $g(n)$, 并相应地修改 n 的父指针, 步骤(4)还在 n 已被扩展的情况下将 n 移回 OPEN 表, 使得 n 会被再次扩展。

在图 3.1 所示的九宫图问题中, 尽管并不知道 $h^*(n)$ 具体为多少, 但在定义 $h(n) = w(n)$ 时保证了 h 的可采纳性。这是因为 $w(n)$ 统计的是“不在目标状态中相应位置的数字卡片个数”, 这相当于假定把不在目标位置的一个数字卡片移动到它的目标位置仅需要一步, 而实际情况下把一个数字卡片移到目标位置应该需要一步以上。所以 $w(n)$ 必然不大于 $h^*(n)$ 。应当指出, 同一问题启发函数 $h(n)$ 可以有多种设计方法。在九宫图问题中, 还可以定义启发函数 $h(n) = p(n)$, 其中 $p(n)$ 为结点 n 的每一数字卡片与其目标位置之间的欧几里得距离总和。显然有 $p(n) \leq h^*(n)$, 相应的搜索过程也是 A^* 算法。然而 $p(n)$ 比 $w(n)$ 有更强的启发性信息, 因为由 $h(n) = p(n)$ 构造的启发式搜索树比 $h(n) = w(n)$ 构造的启发式搜索树结点数要少。

现在给出一些关于算法性质的定义, 为了叙述方便, 将一个算法记作 M 。

完备性: 如果存在解, 则 M 一定能找到该解并停止, 则称 M 是完备的。

可纳性: 如果存在解, 则 M 一定能够找到最优的解, 则称 M 是可纳的。

优越性: 一个算法 M_1 称为优越于另一个算法 M_2 , 指的是如果一个结点由 M_1 扩展, 则它也会被 M_2 扩展, 即 M_1 扩展的结点集是 M_2 扩展的结点集的子集。

最优性: 在一组算法中如果 M 比其他算法都优越, 则算法 M 称为最优的。

下面的定理 3.1 说明了 A^* 算法的完备性和可纳性。为了证明该定理, 我们首先介绍引理 3.1。

引理 3.1 在 A^* 算法停止之前的每次结点扩展前, 在 OPEN 表上总是存在具有如下性质的结点 n^* :

- (a) n^* 位于一条解路径上。
- (b) A^* 算法已得出从初始结点 S_0 到 n^* 的最优路径。
- (c) $f(n^*) \leq f^*(S_0)$ 。

证明: 为证明此引理在 A^* 算法的每次结点扩展前都成立, 只需证明: ①本引理在 A^* 算法初始执行时成立; ②若本引理在一个结点被扩展之前成立, 则在该结点被扩展之后本引理同样成立。按照此思路, 将采用归纳法进行证明。为叙述方便, 以下简称 A^* 算法为 A^* 。

归纳基础: 在 A^* 算法的第 1 次结点扩展前(即 S_0 被选择进行扩展之前), S_0 在 OPEN 表中, S_0 位于一条最优解路径上(因为所有的解路径都以 S_0 为起点), 并且 A^* 已得知从 S_0 到 S_0 的最优路径。此外, 根据 f 的定义,

$$f(S_0) = g(S_0) + h(S_0) = h(S_0) \leq h^*(S_0) = g^*(S_0) + h^*(S_0) = f^*(S_0)$$

因此, 在第 1 次结点扩展前, S_0 就是满足引理结论的 n^* 。

归纳步骤: 假设引理在第 m 次($m \geq 0$)结点扩展后成立, 证明本引理在第 $m+1$ 次结点扩展后仍成立。

假定 A^* 算法在扩展 m 个结点后, OPEN 表中存在一个结点 n^* , A^* 算法已知从 S_0 到 n^* 的最优路径。那么, 若 n^* 在第 $m+1$ 次扩展中未被选择, 则它在第 $m+1$ 次扩展后

是满足引理要求的结点 n^* , 在此情况下引理得证。另一方面, 若 n^* 在第 $m+1$ 次扩展时被选择, 则 n^* 的每一个未在 OPEN 表和 CLOSED 表中出现的子结点都将被放入 OPEN 表, 而且, 这些新的子结点中必然存在一个结点 (记为 n_p) 位于最优解路径上 (因为经过 n^* 的最优解路径必然在经过 n^* 后再经过 n^* 的某个子结点, 所以 n_p 必然存在)。 n_p 也满足性质 (b), 即 A^* 已得出从 S_0 到 n_p 的最优路径, 该路径记为 P1: 由到达 n^* 的最优路径再连接上 n^* 到 n_p 的有向边而组成。如果从 S_0 到达 n_p 的最优路径不同于 P1, 则 P1 不构成最优解路径, 从而与 n^* 在最优解路径上的假设相矛盾。因此, n_p 满足性质 (a) 和 (b)。下面还需证明性质 (c) 在所有归纳步骤中成立, 即证明性质 (c) 在 A^* 停止前的 0 到 m 次扩展时都成立。

对于任一结点 n^* (n^* 在最优解路径上; 且 A^* 算法已得出从 S_0 到 n^* 的最优路径, 即 $g(n^*) = g^*(n^*)$), 它满足如下不等式:

$f(n^*) = g(n^*) + h(n^*) \leq g^*(n^*) + h(n^*) \leq f^*(n^*) \leq f^*(S_0)$ 。因此, 性质 (c) 成立。至此, 本引理得证。

定理 3.1 若 A^* 算法的运行条件成立, 并且搜索空间中存在从初始结点 S_0 到目标结点的代价有穷的路径, 则 A^* 算法保证停止并得出 S_0 到目标结点的最优代价路径。

证明: 在引理 3.1 的基础上, 证明本定理。首先证明如果搜索空间存在目标结点, 则 A^* 必然停止, 然后证明 A^* 在停止时已找到最优解路径。

首先证明 A^* 必然停止: 假设它不停止, 则它将不断扩展 OPEN 表中的结点。我们已假定搜索空间的分枝因子 (每个结点的平均子结点数目) 为一个有穷值, 且每条有向边的权值为正数。所以, 随着 OPEN 表上的结点在搜索树中的深度增加, 它们的 g 值将无限增长。这种增长必然导致 A^* 在未来的一次结点扩展时 OPEN 表中所有结点的 g 值都大于 $f(S_0)$, 此情况与引理 3.1 矛盾。因此 A^* 算法必然停止。

其次证明 A^* 算法停止时已找到一条最优的解路径。 A^* 只有在 OPEN 表为空或者当前扩展的结点为目标结点时才停止。前一个停止条件在不存在目标结点的搜索空间上发生。而本定理要求搜索空间存在目标结点。因此 A^* 必然在扩展一个目标结点时停止。那么, 现在只需说明该目标结点是否是最优的。假设 A^* 算法在停止时扩展的目标结点不是最优的, 并记此结点为 n_{g2} , 而最优目标结点为 n_{g1} 。易知, 在此情况下, $f^*(n_{g2}) < f^*(S_0)$, $f^*(n_{g1}) = f^*(S_0)$ 。此假设与引理 3.1 矛盾。因为引理 3.1 说明: 在 A^* 选择 n_{g2} 之前, OPEN 表上必然存在一个结点 n^* 满足 $f^*(n^*) \leq f^*(S_0)$ 。由于 $f^*(n^*) \leq f^*(S_0)$, 所以 A^* 在考察 n_{g2} 和 n^* 时必然选择 n^* 而不是 n_{g2} , 这与假设选择了 n_{g2} 相矛盾。

至此, 定理 3.1 得证。

从以上分析可见, 启发函数 h 的性质影响 A^* 算法的可纳性。实际上, h 还影响 A^* 算法的结点扩展数目和实现细节。对于两个可纳的启发函数 h_1 和 h_2 , 如果对于任一结点 n 满足 $h_1(n) \leq h_2(n)$, 则称 h_2 的信息量大于 h_1 。当 A^* 算法使用信息量大的启发函数时, 其扩展的结点数目要少, 表现出“优越性”。另一方面, 如果启发函数具有“单调性”, 则 A^* 算法不必在重复访问一个结点时修改该结点的父指针。

在 A^* 算法中, 计算时间不是主要的限制。由于 A^* 算法把所有生成的结点保存在内

存中,所以 A^* 算法在耗尽计算时间之前一般早已经把存储空间耗尽了。因此,后来开发了一些新的算法,它们的目的是为了克服存储空间问题,但一般不满足最优性或完备性,如迭代加深 A^* 算法(IDA *)、简化内存受限 A^* 算法(SMA *)等。下面简单介绍 IDA * 算法。

3.3.5 迭代加深 A^* 算法

前面已经讨论了迭代加深搜索算法,它以深度优先的方式在有限制的深度内搜索目标结点。该算法在每个深度上检查目标结点是否出现,如果出现则停止,否则深度加 1 继续搜索。而 A^* 算法是选择具有最小评估函数值的结点扩展。下面给出的迭代加深 A^* 搜索算法是上述两种算法的结合。这里启发式函数用作深度的限制,而不是选择扩展结点的排序。IDA * 算法如下:

```

Procedure IDA $^*$ 
Begin
    初始化当前的深度限制,  $c = 1$ ;
    把初始结点压入栈,并假定  $c' = \infty$ ;
    While 栈不空 Do
        Begin
            弹出栈顶元素  $n$ 
            If  $n = \text{goal}$ , Then 结束, 返回  $n$  以及从初始结点到  $n$  的路径
            Else
                Begin
                    For  $n$  的每个子结点  $n'$  Do
                        Begin
                            If  $f(n') \leq c$ , Then 把  $n'$  压入栈
                            Else  $c' = \min(c', f(n'))$ 
                        End
                    End
                End
            End
        End
    End
    If 栈为空并且  $c' = \infty$ , Then 停止并退出;
    If 栈为空并且  $c' \neq \infty$ , Then  $c = c'$ , 并返回 2。
End

```

上述算法涉及两个深度限制。如果栈中所含结点的所有子结点的 f 值小于限制值 c , 则把这些子结点压入栈中以满足迭代加深算法的深度优先准则。而如果不是这样,即结点 n 的一个或多个子结点 n' 的 f 值大于限制值 c , 则结点 n 的 c 值设置为 $\min(c, f(n'))$ 。

该算法停止的条件为: ①找到目标结点(成功结束); ②栈为空并且限制值 $c' = \infty$ 。

IDA * 算法和 A^* 算法相比,主要优点是对于内存的需求。 A^* 算法需要指数级数量的存储空间,因为没有限制搜索深度。而 IDA * 算法只有当结点 n 的所有子结点 n' 的 $f(n')$ 小于限制值 c 时才扩展它,这样就可以节省大量的内存。另一个问题是当启发式函数是最优的时候,IDA * 算法和 A^* 算法扩展相同的结点,并且可以找到最优路径。

3.4 问题归约和 AND-OR 图启发式搜索

问题归约法是不同于状态空间法的另一种问题描述和求解的方法。归约法把复杂的问题变换为若干需要同时处理的较为简单的子问题后再加以分别求解: 只有当这些子问

题全部解决时,问题才算解决,问题的解答由子问题的解答联合构成。

3.4.1 问题归约的概述

问题归约也可用一个三元组(S_0, O, P)来描述,其中 S_0 为初始问题,即要求解的问题; P 是本原问题集,其中的每一个问题是不证明的,自然成立的,如公理、已知事实等,或已证明过的; O 是操作算子集,通过一个操作算子把一个问题化成若干个子问题。

这种表示方法是由问题出发,运用操作算子产生一些子问题,对子问题再运用操作算子产生子问题,这样一直进行到产生的问题均为本原问题,则问题得解。

所有问题归约的最终目的是产生本原问题。问题归约法比状态空间法更适用的问题求解方法,如果在归约法中,每运用一次操作算子,只产生一个子问题,则就是状态空间法。

3.4.2 AND-OR 图的问题表示和应用

用与或图可以方便地把问题归约为子问题替换集合。例如,假设问题 A 既可通过问题 C_1 与 C_2 ,又可通过问题 C_3 、 C_4 和 C_5 ,或者由单独求解问题 C_6 来解决,如图 3.2 所示。图中结点表示要求解问题或子问题。

问题 C_1 和 C_2 构成后继问题的一个集合,问题 C_3 、 C_4 和 C_5 构成另一个后继问题集合;而问题 C_6 则为第 3 个集合。对应于某个给定集合的各结点,用一个连接它们的圆弧来标记。图 3.3 中连接 C_1 与 C_2 、 C_3 、 C_4 和 C_5 的圆弧称为 2 连接弧和 3 连接弧。一般而言,这种弧称为 K 连接弧,表示对问题 A 作用某个操作算子后产生 K 个子问题。

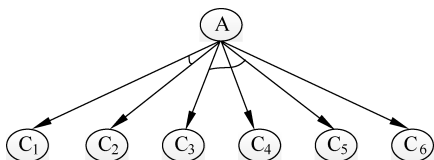


图 3.2 子问题替换集合

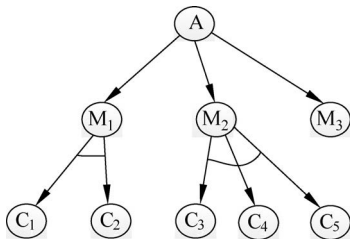


图 3.3 各结点后继只含一个 K 连接弧的与或图

由结点及 K 连接弧组成的图,称为与或图,当所有 K 均为 1 时,就变为普通图。

可以引进某些附加结点,以便使含有一个以上后继问题的每个集合能够聚集在它们各自的父辈结点之下。这样图 3.2 就变为图 3.3 所示的结构了,每个结点的后继只包含一个 K 连接弧。弧连接的子结点叫作与结点,如 C_1 与 C_2 及 C_3 、 C_4 、 C_5 。 $K=1$ 连接的子结点叫做或结点。

当用与或图表示问题归约方法时,其始结点对应于初始问题,第一个 K 连接弧对应于使用了某个操作算子,相应的子结点就表示用该算子后产生的子问题,与或图中的叶结点表示本原问题。

在下面的讨论中,假设与或图中每个结点只包含一个 K 连接弧连接的子结点。

在与或图上执行的搜索过程,其目的在于表明起始结点是有解的。与或图中一个可

解结点可递归地定义如下。

- (1) 终止结点(即本原问题结点)是可解结点。
- (2) 如果某结点有或子结点,那么该结点可解当且仅当至少有一个结点为可解结点。
- (3) 如果某结点有与子结点,那么该结点可解当且仅当所有子结点均为可解结点。

不可解结点可递归定义如下。

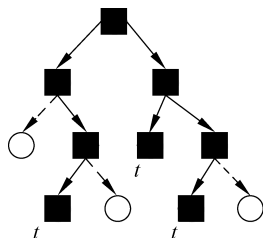


图 3.4 与或图及其解图

- (1) 没有后继结点的非终止结点是不可解结点。
- (2) 或结点是不可解结点,当且仅当它的所有子结点都是不可解结点。
- (3) 一结点是不可解结点,当且仅当它的子结点中至少有一个是不可解结点。

在图 3.4 的例子中,可解结点用黑圆点表示,不可解结点用圆圈表示。能导致初始结点可解的那些可解结点及有关连线组成的子图称为该与或图的解图。

例 3.1 三盘片梵塔问题

原问题可以表示为

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

利用归约方法,在该与或树中,有 7 个终止结点,它们分别对应着 7 个本原问题。如果把这些本原问题从左至右排列起来,即得到了原始问题的解

$$\begin{aligned} (1,1,1) &\rightarrow (1,1,3) \quad (1,1,3) \rightarrow (1,2,3) \\ (1,2,3) &\rightarrow (1,2,2) \quad (1,2,2) \rightarrow (3,2,2) \\ (3,2,2) &\rightarrow (3,2,1) \quad (3,2,1) \rightarrow (3,3,1) \\ (3,3,1) &\rightarrow (3,3,3) \end{aligned}$$

与或树如图 3.5 所示。

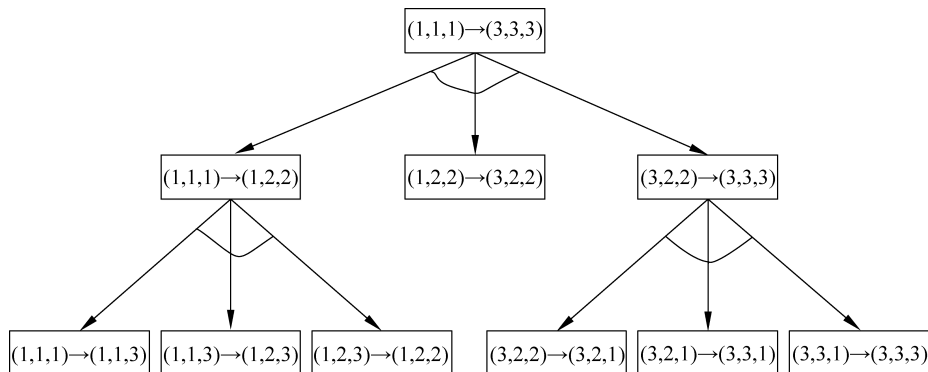


图 3.5 三阶梵塔的与或树

3.4.3 AO* 算法及应用

为了在与或图中找到解,需要一个类似于 A* 的算法,Nilsson 将它称为 AO* 算法。它和 A* 算法是不同的,其主要区别如下。

(1) AO* 算法要能处理与图,它应找出一条路径,即从该图的开始结点出发到达代表解状态的一组结点。

为了弄清为什么 A* 算法不足以搜索与或图,可以考察图 3.6(a)所示的与或图,扩展顶点 A 产生两个子结点集合,一个为结点 B,另一个由结点 C 和 D 组成。在每个结点旁边的数表示该结点 f 值。为简单起见,假定每一操作的耗费是一致的,设带一个后继结点的耗费为 1。若查看结点并从中挑选一个带最低 f 值的结点扩展,则要挑选 C。但根据现有信息,最好开发穿过 B 的那条路径,因扩展 C 也得扩展 D,其总耗费为 9,即 $D+C+2$;而穿过 B 的耗费为 6。问题在于下一步要扩展结点的选择不仅依赖于那一结点的 f 值,而且取决于那一结点是否属于从初始结点出发的当前最短路径的一部分。对此,图 3.6(b)所示的与或图更加清楚。按 A* 算法,最有希望的结点是 G,其 f 值为 3。G 结点是 C 的后继,C 也是 B,C,D 中最有希望的结点,其总耗费为 9。但 C 不是当前最短路径的一部分,因用 C 需用 D,而 D 的耗费为 27。因此不应扩展 G,而应考虑 E 和 F。

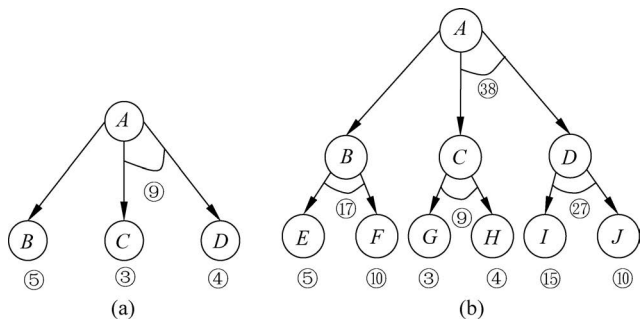


图 3.6 与或图

由此可见,在扩展搜索一个与或图时,每步需做 3 件事。

① 遍历图,从初始结点开始,顺延当前最短路径,积累在此路径上但未扩展的结点集。

② 从这些未扩展结点中选择一个并扩展之。将其后继结点加入图中,计算每一后继结点的 f 值(只需计算 h ,不管 g)。

③ 改变最新扩展结点的 f 估值,以反映由其后继结点提供的新信息。将这种改变往后回传至整个图。在图中往后回走时,每到一结点就判断其后继路径中哪一条最有希望,并将它标记为目前最短路径的一部分。这样可能引起目前最短路的变动。这种图的往后回走传播修正耗费估计的工作在 A* 算法中是不必要的,因为只需考察未扩展结点。但现在必须考察已扩展结点以便挑选目前最短路径。于是,其值是目前最佳估计这一点很重要。

下面通过图 3.7 所示搜索过程中的例子来说明此过程。

步骤 1 A 是唯一结点,因此它在目前最短路径的末端。

步骤 2 扩展 A 后得结点 B、C 和 D,因为 B 和 C 的耗费为 9,到 D 的耗费为 6,所以把到 D 的路径标志为出自 A 的最有希望的路径(被标志的路径在图中用箭头指出)。

步骤 3 选择 D 扩展,从而得到一新路径,即得到 E 和 F 的与弧,其复合耗费估计为

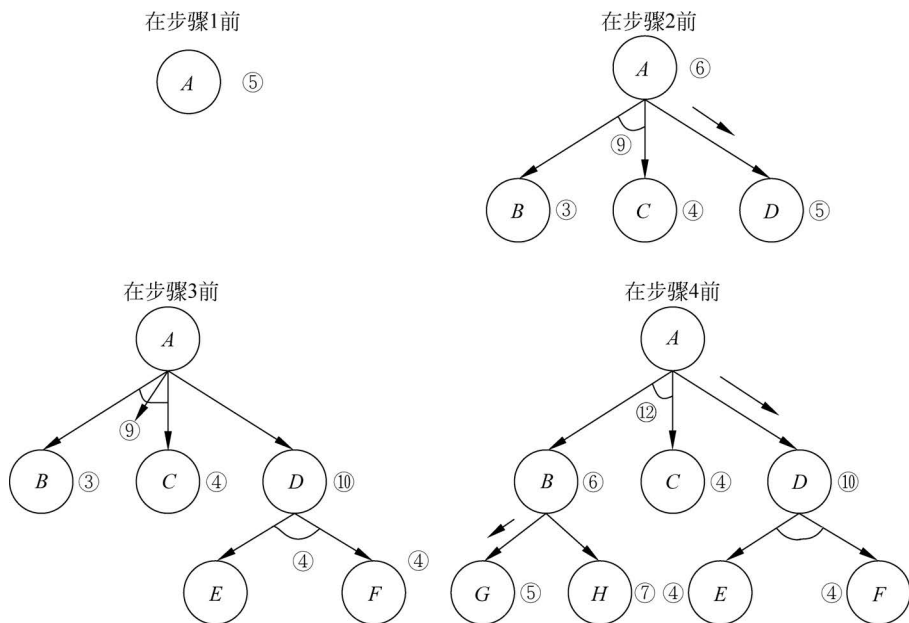


图 3.7 一个与或图的搜索过程

10,故将 D 的 f 值修改为 10。往回退一层发现, A 到 B 、 C 与结点集的耗费为 9,所以,从 A 到 B 、 C 是当前最有希望的路径。

步骤 4 扩展 B 结点,得结点 G 、 H ,且它们的耗费分别为 5、7。往回传其 f 值后, B 的 f 值改为 6(因为 G 的弧最佳)。往回上一层后, A 到 B 、 C 与结点集的耗费改为 12,即 $6+4+2$ 。此后, D 的路径再次成为更好的路径,所以将它作为目前最短路径。

最后求得的耗费为 $f(A) = \min(12, 4+4+2+1) = 11$ 。

从以上分析可以看出,与或图搜索由两个过程组成。

- ① 自顶向下,沿着最优路径产生后继结点,判断结点是否可解。
- ② 自底向上,传播结点是否可解,做估值修正,重新选择最优路径。

(2) 如果有些路径通往的结点是其他路径上的“与”结点扩展出来的结点,那么不能像“或”结点那样只考虑从结点到结点的个别路径,有时候路径长一些可能会更好。

(3) AO^* 算法仅对保证不含任何回路的图进行操作。作这种保证是因为存储一条回路路径绝无必要,这样的路径代表了一条循环推理链。

3.5 博弈树及搜索技术

机器博弈被认为是人工智能领域具挑战性的研究课题之一,具有相当长的历史,从 1956 年塞缪尔的跳棋程序到 1997 年“深蓝”的胜利,计算机与人类经历了一场场波澜壮阔的“搏杀”,给人类留下了难以忘怀的记忆。国际象棋的计算机博弈取得了举世瞩目的成就,积累了一套过程建模、状态表示、走法生成、棋局评估、博弈树搜索、开局库与残局库开发、系统测试与参数优化等成熟的核心技术。然而,由于不同博弈问题的难度和特点差

异很大,目前为止,并非所有的博弈问题都能迎刃而解。中国象棋计算机博弈的难度不亚于国际象棋,研究者还较少且参考资料不多,依然存在着巨大的研究空间。因此,博弈至今仍然是人工智能领域非常重要的一个研究热点。

在实际博弈游戏中,各种搜索技术和启发性知识得到了广泛的应用。除了“深蓝”在国际象棋中的成功之外,人工智能在跳棋、黑白棋、西洋双陆棋、围棋、桥棋和其他扑克牌游戏中都有成功的应用。虽然为围棋一类复杂的游戏建模还存在一定的困难,但近年来,人工智能在视频游戏、战略游戏中,及其对经济现象(自动协商、网络拍卖)的研究中又有了更加广泛和深入的应用。

现实中的博弈游戏花样繁多,但开发人工智能程序所使用的技术却有着一定的相似之处。通常包括以下几个方面的设计:

1. 棋盘的表示

游戏的棋子通常分布在平面的棋盘上,且棋子只能按移动规则在一些固定的位置上移动。所以,棋盘通常用二进制数组表示,数组每个元素的不同取值代表不同棋子。另外,一些改进的棋盘表示法会在四周增加一些辅助单元,以判定棋子走法的合法性,即越界检查等。

2. 开局库

在很多游戏中,开局几步棋对于占据有利的棋局是非常重要的,所以许多游戏程序都使用了开局库。游戏开始时,从对应于某给定策略的开局库中选择一组走棋方案,然后使用静态估值函数推导后面的走棋方案。

3. 静态估值函数

棋局的评价使用静态估值函数。它通常是一个加权的数值型特征向量的函数,具体的特征包括双方棋子的个数、位置、双方棋子的数量差、关键棋子(如王或帅等)的位置和数量信息等,当然权值的选择和静态估值函数的设计体现着玩家的智慧,或者说决定博弈程序的水平高低。

3.5.1 博弈树

博弈,广义地理解为由若干主体参加的斗智、斗勇的竞争过程。在对弈、军事、经济、商业、游戏等领域广泛地存在。常常狭义地理解为对弈类游戏。

1. 博弈问题的状态空间

博弈问题中,棋局用状态表示,对弈双方任意合法地走一步,都使棋局从一个状态变到另一个状态,所以,所有合法的走步就是操作。开局是初始状态,终局可能有多种,站在某一方的角度,有胜局、和局和负局之分。有了状态和操作的描述,就可以构造博弈问题的状态空间。

值得注意的是,对弈与迷宫类(如重排九宫)问题不同,对于迷宫问题,探索者有完全

的选择权,每一步都可以沿着他认为有利的路线走下去。而对弈问题中对弈双方都只有一半选择权,双方的利益是完全冲突的。因此,讨论对弈问题时,评价一个棋局的好坏与胜负,一定要站在某方的立场上。另外,还需要假定双方都是精通博弈者,即每步选择都是理智的。

博弈树: 博弈问题的状态空间就是以状态为结点,以合法走步为边的一个树形图,称为博弈树。

2. 博弈树的特点

假设博弈过程是由双方来完成的,称我方为 MAX 方,对方为 MIN 方,以下总是站在 MAX 方的立场讨论问题。由于双方的利益是完全冲突的,在博弈树中存在与、或两种结点。

与结点: 博弈过程中,对手(MIN)每走一着棋(半个回合),都力图干扰 MAX 的选择,使其偏离取胜的目标。轮到 MIN 走棋时,由于它掌握着出棋的主动权,此时只要全部走法中有一个能导致对方(MAX)败局,或者说,有一个走法能让对手(MAX)取得更低的得分,它就会毫不犹豫地选择这一走法,因此,站在我方(MAX)的立场上,由 MIN 出棋的结点具有与结点的性质。

或结点: 博弈过程中,我方(MAX)每走一着棋(半个回合),都力图使自己通往取胜的目标。轮到 MAX 走棋时,由于它掌握着出棋的主动权,此时只要各种走法中有一个能通向胜局,或者说,有一个走法能让它取得更高的得分,它就会毫不犹豫地选择这一走法,因此,站在我方(MAX)的立场上,MAX 出棋的结点具有或结点的性质。

博弈的过程是双方轮流走步,因此,博弈树中的与、或结点就会按层交替出现。这就是博弈树的特点。

3.5.2 博弈树的搜索及应用实例

常见的博弈树包含大量的结点(状态),从初局开始,随着对弈步数增加,所产生的状态数目会以指数规模增加,即出现所谓的棋局组合爆炸现象。因此,博弈树的搜索也必须是在隐式图的基础上进行的启发式搜索。

为简单起见,这里仅讨论二人零和、全信息、非偶然性博弈。

“二人零和”是指对弈双方的利益是完全冲突的,利益之和永远为零;博弈的结果有三种情况: MAX 方胜; MIN 方胜; 双方平局。很多博弈问题都满足“二人零和”这一条件,如国际象棋。但囚徒困境是经典的非零和博弈例子。

“全信息”是指在对弈过程中,当前的格局及双方对弈的历史是公开的。

“非偶然性”是指对弈双方的每一步选择都是“理智”的,即在采取行动前,都要根据自己的静态估值函数(启发函数)进行得失分析,选取对自己最有利而对对方最不利的对策,不存在“碰运气”式的随机因素。

在对弈问题中,为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案,就需要对当前的情况以及将要发生的情况进行分析,从中选出最佳走步。最常用的方法为极小极大分析及 α - β 剪枝技术,以下将进行简要介绍。

1. 极小极大分析

先介绍两个术语：

静态估值：为了找到当前的最佳走步，一般要向前看多个回合。为此，需要根据问题的特性信息（来自博弈者的经验）定义一个静态估值函数，估价当前博弈树末端结点的得分，称为格局的静态估值。

倒推值：当末端结点的估值计算出来后，再向上推算出各结点的得分，称为倒推值。

极小极大分析方法就是根据当前棋局的静态估值，计算先辈结点倒推值，为其中的一方（例如 MAX）寻找一个最佳走步。其基本思想可通俗地概括为“立足于最坏，争取最好”。

极小极大分析法：对与结点求极小值、或对结点求极大值计算各先辈结点倒推值的方法。

对与结点，选其子结点中一个极小得分作为父结点的得分（倒推值），因为与结点是对方（MIN）结点，站在我方（MAX）的观点，对方的选择一定设法使我方取得极小分，这就是我方考虑与结点得分时所谓的“立足于最坏”。

对或结点，选其子结点中一个极大得分作为父结点的得分（倒推值），因为或结点是 my 方的结点，站在我方的观点，我方的选择一定要争取得到极大分，这就是我方考虑或结点时所谓的“争取最好”。

用极小极大分析法求最佳走步的具体过程是：

首先，按扩展深度限制（回合数）扩展结点，对末端结点求静态估值；然后，对内部结点按极小极大分析方法求倒推值，最后，根据结点的倒推值决定一个最佳走步。

每扩展一次，对内部结点都用新的倒推值代替原来的静态估值或原来的倒推值。如果一个行动方案能获得较大的倒推值，则它就是当前最好的行动方案。

例 3.2 用极小极大分析方法求最佳走步。设某博弈过程对应的博弈树如图 3.8 所示。我方当前格局为 S，考虑扩展深度为 4（两个回合）。

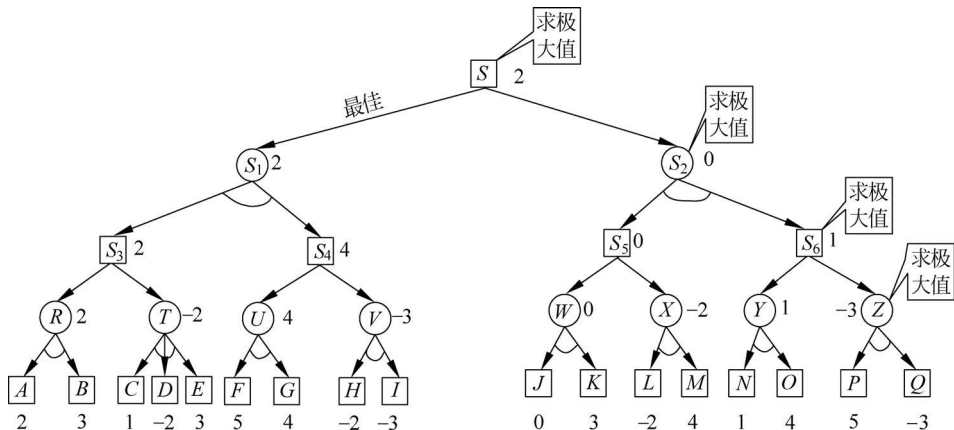


图 3.8 用极小极大分析方法求最佳走步

解：现在要求出我方的下一个最佳走步。首先，对末端结点使用静态估值函数计算，得到的静态估值标识在结点下面；然后，使用极小极大分析过程依次求先辈结点的倒推值，标识在结点右侧；最后，分析的结果是我方会选择得分为2的 S_1 格局。事实上，如果这一估计是准确的，图中 S 、 S_1 、 S_3 、 R 、 A 所代表的路径对双方都是最有利的走步。

例 3.3 井字棋(一字棋)游戏。井字棋的棋盘如图 3.9 的根结点所示。由 MAX 和 MIN 二人对弈，双方轮流往棋盘的空格上放一枚自己的棋子，MAX 放“×”，MIN 放“○”，谁先使自己的棋子构成“三子一线”谁就取得了胜利。

解：棋盘上的整行、整列和对角线称为路(线)。如果一条路上只有 i 方的棋子，则称该路为 i 方的路。如果一条路上只有 i 方的 k 个棋子，则称为该路为 i 方的 k 阶路。

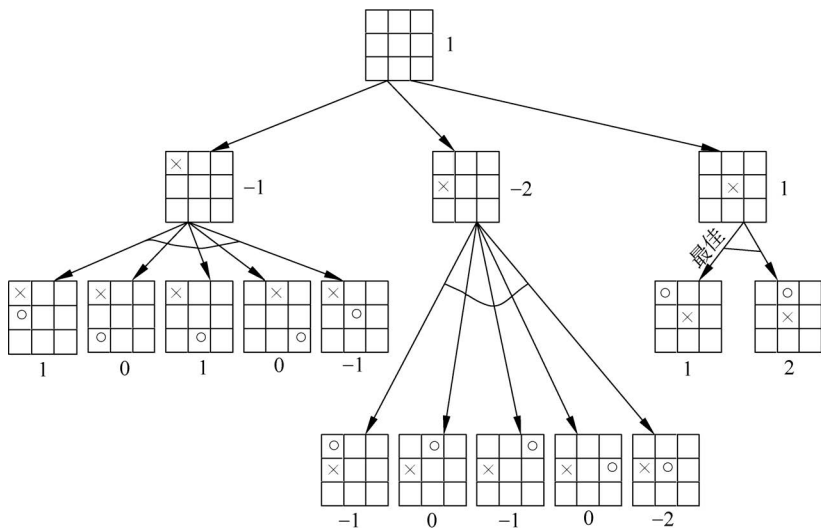


图 3.9 用极小极大分析方法求解井字游戏的第一步

设棋局为 x ，静态估值函数 $f_1(x)=h_1(x)$ 定义如下：

- (1) 若 x 是 MAX 必胜的棋局，则 $h_1(x)=+\infty$ 。
- (2) 若 x 是 MIN 必胜的棋局，则 $h_1(x)=-\infty$ 。
- (3) 若 x 是胜负未定的棋局，则 $h_1(x)=$ “×”方的路数—“○”方的路数。

为了不至于生成太大的博弈树，从开局仅扩展两层(一个回合)，得到的博弈树如图 3.9 所示。按 $h_1(x)$ 计算末端结点的静态估值在结点下方，使用极小极大分析过程，求得“×”方的第一步的最佳走步是将“×”放在正中心位置。

2. α - β 剪枝

极小极大分析法需要按深度限制生成全部端结点，并通过计算全部结点的静态估值或倒推值选择当前结点的一个最佳走步，得到当前格局的后继格局。这种方法将“生成后继”和“估计棋局”两个过程分开考虑。然而，博弈问题往往随着向前看步数的增加，要估计的端结点的个数以指数级增加。因此，极小极大分析法的缺点是效率低。

为此，考虑把“生成后继”和“估计棋局”两个过程结合起来，边生成博弈树边计算各结

点的倒推值,并且根据已知倒推值的范围,及时停止那些不必要结点的生成,即相当于对这些结点(及其以下分枝)“剪枝”,从而减少搜索结点的数目,提高搜索效率, α - β 剪枝是一种典型的在极小极大分析中使用的剪枝技术。

α 剪枝: 对于一个与结点 MIN,若能估计出其倒推值的上确界 β ,并且 β 不大于它的父结点(或结点)的倒推值的下确界 α ,即 $\alpha \geq \beta$,则不必生成该结点的其余子结点(因为这些子结点的估值不会提高 MIN 的父结点的倒推值),这一过程称为 α 剪枝。

如图 3.10(a)所示部分博弈树中,对于与结点 T 来讲,根据结点 C 可确定 T 的取值范围为小于或等于 1,其中 1 即是 T 的上确界 β , T 的父结点 S_3 的取值范围通过 R 可确定为大于或等于 2,2 为结点 S_3 的下确界 α ,即 $\alpha \geq \beta$,结点 T 的取值将不会影响 S_3 的倒推值,因此 D 结点和 E 结点没有考察的必要,即可以进行如图所示的 α 剪枝。结点旁边的数字是相应结点的静态估值或倒推值,圆圈中的数字代表的是结点计算的顺序(下同)。

β 剪枝: 对于一个或结点 MAX,若能估计出其倒推值的下确界 α ,并且 α 不小于 MAX 的父结点(与结点)的倒推值上确界 β ,即 $\alpha \leq \beta$,则不必生成该结点的其余子结点(因为这些子结点的估值不会降低 MAX 的父结点的倒推值),这一过程称为 β 剪枝。 β 剪枝过程的例子如图 3.10(b)所示。

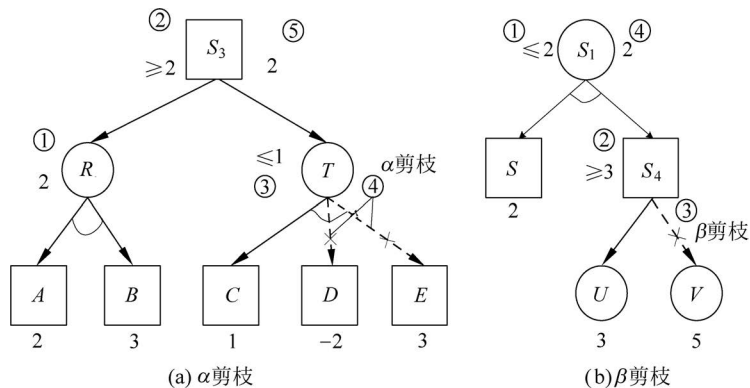


图 3.10 α - β 剪枝过程示意图

需要说明的是,对与结点求上确界 β ,对其子结点(或结点)考虑是否可以进行 α 剪枝;对或结点求下确界 α ,对其子结点(与结点)考虑是否可以进行 β 剪枝。

对于与结点,如果它的父结点(或结点)的 α 值已求出,就逐个用它的子结点去判断该与结点的 β 值,一旦发现 $\beta \leq \alpha$,对其余子结点立即进行 α 剪枝。但如果一个与结点的父结点(或结点)的 α 值还没有求出,就要用该与结点的全部子结点求出极小值,再倒推出父结点的 α 值。对或结点情况类似。对于不能剪枝的结点仍要使用极小极大分析方法。

3. 改进 α - β 剪枝算法

1) 窗口原则

在 α - β 剪枝过程中,初始的搜索窗口往往是从 $-\infty$ (即初始的 α 值)到 $+\infty$ (初始的 β 值),在搜索进行中再不断缩小窗口,加大剪枝效果,这种估计是可靠的,但却不是高效的。

如果我们一开始就使用一个较小的窗口,那么我们就有可能提高剪枝效率,这就是窗口原则。

使用窗口原则的算法有: Falphabeta 算法,即 Failsoft-Alphabeta 算法;渴望搜索(Aspiration Search);极小窗口搜索(Minimal Window Search/PVS)。

2) 置换表

置换表基本思想:在搜索进行中,用一张表把搜索过的结点结果(包括搜索深度,估值类型:准确还是上下边界)记录下来,在后继的搜索过程中,查看表中记录。如果搜索的结点已经有记录(子树的深度大于或者等于当前的新结点要求的搜索深度),它的信息就可以直接运用了,这样我们可以避免重复搜索很多子树。置换表是一种内存增强技术,以空间换时间。

3) 历史启发

历史启发是为了迎合 α - β 剪枝搜索对结点排列顺序敏感的特点来提高剪枝效率的,即根据历史走法对当前搜索的结点集进行排序,从而优先搜索好的走法。

4) 迭代深化

迭代深化是一个不断求精的过程,对博弈树进行多次遍历,并不断加深其深度,用于控制搜索的时间。

在实用中迭代深化和前面提到的算法结合使用具有很好的效果,如 PVS 算法,上几层迭代得到的最佳走法可以帮助下一层提高剪枝效率;迭代过程中把前面局面的历史得分存入置换表,最佳走法存入历史启发表可以提高剪枝效率。

5) 实验数据分析

各种增强策略都能提高 α - β 剪枝的效率,其中空窗口探测(PVS)从第五层开始平均需估计的结点数减少为一半,而效率提高一倍。单纯的迭代深化由于再迭代需要耗费时间,从效率上看提高不大。置换表在前三层没什么表现,这是因为置换表操作也要耗费时间,且当其命中率低时效果不佳,但层数较多命中率高时优势越来越明显。历史启发是这几种增强策略中最好的,在第五层效率就能提高十倍以上,越往后效果更好,这也证实了 α - β 剪枝对顺序的极度敏感。

MTD(f)算法在实验中的前几层稍优于 PVS 算法,但它层数大于六时很不稳定且本身带置换表,因此在把各种增强策略融合时不如 PVS 算法。融合各种增强策略的 PVS 算法在第六层就比基本的 α - β 剪枝快两百多倍。

4. α - β 截断

在深度优先的最小最大法中,我们可以看到,博弈树的某些部分并不会产生任何有意义的值,因而也根本用不着去扩展博弈树的这一部分。识别博弈树中这些可忽略部分的技术,称之为 α - β 截断。之所以叫这个名字,是由于历史原因造成的。

我们可以看出,在轮到棋手下棋的结点上,其部分回溯值是 10。而它的当前计算出来的子结点的部分回溯值是 8。现在,由于该子结点是轮到对手下棋的结点,而对手总是要走那个具有最小值的棋局,故进一步探索的结果只会小于这个值。无论最后的确定值是多少,它总是小于或等于 8。

从另一方面来看,该结点本身的部分回溯值是 10。因为这时轮到棋手下棋,所以只有大于 10 的子结点的值才能改变这个部分回溯值。

所以我们得出的结论是:不需要去进一步扩展其子结点或其他任意后续结点。这是因为进一步的扩展至多只能减少其子结点的回溯值,而其目前的值已经足够小到不能影响其亲结点的部分回溯值了。这种情况就是所谓的 α 截断。

现在,我们对一般的原则叙述如下:

在考虑轮到棋手下棋的一个亲结点及轮到对手下棋的一个子结点时,如果该子结点的数值已经小于或等于其亲结点的回溯值,那么就不需要对该结点或者其后续结点做更多的处理了。计算的过程可以直接返回到亲结点上。

当亲结点是轮到对手下棋的一个结点时,该原则作相应的改动:

在考虑轮到对手下棋的一个亲结点及轮到棋手下棋的一个子结点时,如果该子结点的部分回溯值已经大于或等于其亲结点的部分回溯值,那么就不需要对该子结点或者其后裔结点做更多的处理了。计算过程可以直接返回到亲结点上。这就是所谓的 β 截断。

截断这一技术允许我们有时可以不去考虑某结点的某些子结点的情况。然而,由于非终结点的每一个子结点又是其后续结点所组成的整个博弈树的根,所以,如果我能忽略掉那些子结点的话,不仅仅是忽略了它们本身,还忽略了它们所有的后续结点。因此,这一技术可以删去数量相当大的结点,因而也就大大节省了搜索博弈树所需要的时间。

5. 威佐夫博弈

有两堆各若干个物品,两个人轮流从某一堆或同时从两堆中取同样多的物品,规定每次至少取一个,多者不限,最后取光者得胜。

我们用 $(a[k], b[k])$, $(a[k] < b[k])$ ($k \in 0, 1, 2, 3, \dots$) 来表示两堆物品的数量。

首先我们从简单的情况分析:

如果现在的局势为 $(0, 0)$ 则可以看出肯定是之前的人在上一局中取完了。

假设,现在的局势为 $(1, 2)$, 那么先手只有四种取法了。

(1) 如果,先手取走第一堆中的 1 个,变为 $(0, 2)$, 则后手只需要取走第二堆所有的就变成奇异局势 $(0, 0)$, 所以,结果就是先手输,后手赢。

(2) 如果,先手取走第二堆中的 1 个,变为 $(1, 1)$, 则后手只需取走两堆中的 1 个就变成奇异局势 $(0, 0)$, 所以,结果就是先手输,后手赢。

(3) 如果,先手取走第二堆中的 2 个,变为 $(1, 0)$, 则后手只需要取走第一堆所有的就变成奇异局势 $(0, 0)$, 所以,结果就是先手输,后手赢。

(4) 如果,先手在两堆中各取“1”变成 $(0, 1)$, 则后手取走第二堆的 1 就变成奇异局势 $(0, 0)$ 。所以,结果就是先手输,后手赢。

所以由此可得,先手必输。

假设现在的局势是 $(3, 5)$, 首先根据上面分析的经验,我们知道先手肯定不能把任意一堆物品取完,这是因为每次可以从任意一堆取走任意个物品,那么后手就可以直接把另一堆取完,所以后手获胜。

所以我们这里就不分析那些情况,来分析其他的情况。

先看在一堆中取的情况：

(1) 假设先手在“3”中取 1 个,后手就可以在“5”中取走 4 个,这样就变成了(1,2)的局势,根据上面的分析,我们知道是先手输,后手获胜。

(2) 假设先手在“3”中取 2 个,后手就可以在“5”中取走 3 个,这样也变成了(1,2)的局势了,还是先手输,后手获胜。

(3) 假设先手在“5”中取 1 个,后手就在“3”和“5”中各取走 2 个,这样又成了(1,2)的局势了,先手输,后手赢。

(4) 假设先手在“5”中取 2 个,后手就在“3”和“5”中各取走 3 个,这样变成了(0,0)的局势,先手输,后手赢。

(5) 假设先手在“5”中取 3 个,后手就在“3”和“5”中各取走 1 个,也变成了(1,2)的局势,先手输,后手胜利。

(6) 假设先手在“5”中取 4 个,后手在“3”中取走 1 个,还是(1,2)的局势,先手输,后手赢。

我们发现上面列举的这几种局势,无论先手怎么取都是后手赢。

我们可以来找找那些先手必输局势的规律

第一个(0,0)

第二个(1,2)

第三个(3,5)

第四个(4,7)

第五个(6,10)

第六个(8,13)

第七个(9,15)

第八个(11,18)

第 n 个($a[k]$, $b[k]$)

我们把这些局势称为“奇异局势”。

我们会发现它们的差值是递增的,分别是 0,1,2,3,4,5,6,7, $\dots$, n 。

用数学方法分析发现这些局势的第一个值是未在前面出现过的最小的自然数。

继续分析会发现,每种奇异局势的第一个值总是等于当前局势的差值乘上 1.618。

我们都知道 0.618 是黄金分割率。而威佐夫博弈正好是 1.618,这就是博弈的奇妙之处!

即 $a[k] = (\text{int})((b[k] - a[k]) * 1.618)$ 注:这里的 int 是强制类型转换,注意这不是简单的四舍五入,假如后面的值是 3.9,转换以后得到的不是 4 而是 3,也就是说,强制 int 类型转换得到的是不大于这个数值的最大整数。

在编程题中,有些题目要求精度较高,可以用下述式子来表示这个值:

$$1.618 = (\text{sqrt}(5.0) + 1) / 2$$

6. 巴什博弈

只有一堆 n 个物品,两个人轮流从这堆物品中取物,规定每次至少取一个,最多取 m

个。最后取光者得胜。

显然,如果 $n=m+1$,那么由于一次最多只能取 m 个,所以,无论先取者拿走多少个,后取者都能够一次拿走剩余的物品,后者取胜。因此我们发现了如何取胜的法则:如果 $n=(m+1)r+s$, (r 为任意自然数, $s \leq m$),那么先取者要拿走 s 个物品,如果后取者拿走 k ($\leq m$) 个,那么先取者再拿走 $m+1-k$ 个,结果剩下 $(m+1)(r-1)$ 个,以后保持这样的取法,那么先取者肯定获胜。总之,要保持给对手留下 $(m+1)$ 的倍数,就能最后获胜。

这个游戏还可以有一种变相的玩法:两个人轮流报数,每次至少报一个,最多报 10 个,谁能报到 100 者胜。

7. 取火柴问题

题目 1: 今有若干堆火柴,两人依次从中拿取,规定每次只能从一堆中取若干根,可将一堆全取走,但不可不取,最后取完者为胜,求必胜的方法。

题目 2: 今有若干堆火柴,两人依次从中拿取,规定每次只能从一堆中取若干根,可将一堆全取走,但不可不取,最后取完者为负,求必胜的方法。

先解决第一个问题。

定义: 若所有火柴数异或为 0,则该状态被称为利他态,用字母 T 表示; 否则,为利己态,用 S 表示。

定理 3.2: 对于任何一个 S 态,总能从一堆火柴中取出若干个使之成为 T 态。

证明: 若有 n 堆火柴,每堆火柴有 $A(i)$ 根火柴数,那么既然现在处于 S 态, $c = A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(n) > 0$; 把 c 表示成二进制,记它的二进制数的最高位为第 p 位,则必然存在一个 $A(t)$,它二进制的第 p 位也是 1。(否则,若所有的 $A(i)$ 的第 p 位都是 0,这与 c 的第 p 位就也为 0 矛盾)。那么我们把 $x = A(t) \text{ xor } c$,则得到 $x < A(t)$ 。这是因为既然 $A(t)$ 的第 p 位与 c 的第 p 位同为 1,那么 x 的第 p 位变为 0,而高于 p 的位并没有改变。所以 $x < A(t)$ 。而

$$\begin{aligned} & A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } x \text{ xor } \cdots \text{ xor } A(n) \\ &= A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(t) \text{ xor } c \text{ xor } \cdots \text{ xor } A(n) \\ &= A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(n) \text{ xor } A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(n) \\ &= 0 \end{aligned}$$

这就是说从 $A(t)$ 堆中取出 $A(t) - x$ 根火柴后状态就会从 S 态变为 T 态。证毕。

定理 3.3: T 态,取任何一堆的若干根,都将变为 S 态。

证明: 用反证法试试。

若

$$\begin{aligned} c &= A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(i) \text{ xor } \cdots \text{ xor } A(n) = 0; \\ c' &= A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(i') \text{ xor } c \text{ xor } \cdots \text{ xor } A(n) = 0; \end{aligned}$$

$$\begin{aligned} \text{则有 } c \text{ xor } c' &= A(1) \text{ xor } A(2) \text{ xor } \cdots \text{ xor } A(i) \text{ xor } \cdots \text{ xor } A(n) \text{ xor } A(1) \text{ xor } \\ & A(2) \text{ xor } \cdots \text{ xor } A(i') \text{ xor } c \text{ xor } \cdots \text{ xor } A(n) = A(i) \text{ xor } A(i') = 0 \end{aligned}$$

进而推出 $A(i) = A(i')$,这与已知矛盾。所以命题得证。

定理 3.4: S 态,只要方法正确,必赢。

最终胜利即由 S 态转变为 T 态,任何一个 S 态,只要把它变为 T 态,(由定理 3.2,可以把它变成 T 态。)对方只能把 T 态转变为 S 态(定理 3.3)。这样,所有 S 态向 T 态的转变都可以由己方控制,对方只能被动地实现由 T 态转变为 S 态。故 S 态必赢。

定理 3.5: T 态,只要方法正确,必败。

由定理 3.4 易得。

接着来解决第二个问题。

定义:若一堆中仅有 1 根火柴,则被称为孤单堆。若大于 1 根,则称为充裕堆。

定义: T 态中,若充裕堆的堆数大于等于 2,则称为完全利他态,用 T2 表示;若充裕堆的堆数等于 0,则称为部分利他态,用 T0 表示。

孤单堆的根数异或只会影响二进制的最后一位,但充裕堆会影响高位(非最后一位)。一个充裕堆,高位必有一位不为 0,则所有根数异或不为 0。故不会是 T 态。

定理 3.6: S0 态,即仅有奇数个孤单堆,必败。T0 态必胜。

证明: S0 态,其实就是每次只能取一根。每次第奇数根都由己取,第偶数根都由对方取,所以最后一根必己取。败。同理,T0 态必胜。

定理 3.7: S1 态,只要方法正确,必胜。

证明: 若此时孤单堆堆数为奇数,把充裕堆取完;否则,取成一根。这样,就变成奇数个孤单堆,由对方取。由定理 3.5 可知,对方必输。己必胜。

定理 3.8: S2 态不可转一次变为 T0 态。

证明: 充裕堆数不可能一次由 2 变为 0。得证。

定理 3.9: S2 态可一次转变为 T2 态。

证明: 由定理 3.1, S 态可转变为 T 态, S 态可一次转变为 T 态,又由定理 3.6, S2 态不可转一次变为 T0 态,所以转变的 T 态为 T2 态。

定理 3.10: T2 态,只能转变为 S2 态或 S1 态。

证明: 由定理 3.2, T 态必然变为 S 态。由于充裕堆数不可能一次由 2 变为 0,所以此时的 S 态不可能为 S0 态。命题得证。

定理 3.11: S2 态,只要方法正确,必胜。

证明: 方法如下。

(1) S2 态,就把它变为 T2 态(由定理 3.9)。

(2) 对方只能 T2 转变成 S2 态或 S1 态(定理 3.10)(若转变为 S2,转向 1)若转变为 S1,自己必胜(定理 3.6)。

定理 3.12: T2 态必输。

证明: 同定理 3.11。

综上所述,必输态有: T2, S0

必胜态: S2, S1, T0。

两题比较: 第一题的全过程其实如下:

S2→T2→S2→T2→…→T2→S1→T0→S0→T0→…→S0→T0(全 0)

第二题的全过程其实如下:

S2→T2→S2→T2→…→T2→S1→S0→T0→S0→…→S0→T0(全 0)

是否发现了它们的惊人相似之处?

不难发现, S1 态可以转变为 S0 态(第二题做法), 也可以转变为 T0(第一题做法)。哪一方控制了 S1 态, 即可以有办法使自己得到最后一根(转变为 T0), 也可以使对方得到最后一根(转变为 S0)。

所以, 抢夺 S1 是制胜的关键!

为此, 始终把 T2 态让给对方, 将使对方处于被动状态, 因为他早晚将把状态变为 S1。

3.6 高级搜索

前面介绍的搜索算法都是用来探索搜索空间的, 它们在内存中保留一条或多条路径并且记录哪些是已经探索过的, 哪些是还没有探索过的。当找到目标时, 到达目标的路径同时也构成了这个问题的一个解。然而在许多问题中, 问题的解与到达目标的路径是无关的。例如, 在八皇后问题中, 重要的是最终皇后的布局, 而不是加入皇后的次序。这一类问题包括了许多重要的应用, 例如集成电路设计、工厂场地布局、作业车间调度、自动程序设计、电信网络优化、车辆寻径以及文件夹管理等。

局部搜索算法从一个当前状态出发, 移动到与之相邻的状态。搜索的路径通常是不保留的。其优点是: ①它们只用很少的内存, 通常需要的存销量是一个常数; ②它们通常能在不适合系统化算法的很大或无限的(连续的)状态空间中找到合理的解。

除了找到目标, 局部搜索算法对于解决纯粹的最优化问题是很有用的, 其目标是根据一个目标函数找到最佳状态。许多最优化问题不适合“标准的”搜索模型。例如, 自然界提供了一个目标函数——繁殖适应性——达尔文的进化论可以被视为优化的尝试, 但是这个问题没有“目标测试”和“路径耗散”。为了更好地理解局部搜索, 类比地考虑一个地形图。地形图既有“位置”(用状态定义), 又有“高度”(由启发式耗散函数或目标函数的值定义)。如果高度对应于耗散, 那么目标是找到最低谷, 即一个全局最小值; 如果高度对应于目标函数, 那么目标是找到最高峰, 即一个全局最大值(当然可以通过插入一个负号使两者相互转换)。局部搜索算法就像对地形图的探索, 如果存在解, 那么完备的局部搜索算法总能找到解, 最优的局部搜索算法总能找到全局最小值/最大值。

3.6.1 爬山算法原理

爬山法搜索是一种最基本的局部搜索。它像在地形图上登高一样, 一直向值增加的方向持续移动, 将会在到达一个“峰顶”时终止, 并且在相邻状态中没有比它更高的值。爬山法是深度优先搜索的改进算法。在这种算法中, 使用某种贪心算法来决定在搜索空间中向哪个方向搜索。由于爬山法总是选择在局部最优的方向搜索, 所以可能会有“无解”的风险, 而且找到的解不一定是最优解, 但是它比深度优先搜索的效率要高很多。

该算法不维护搜索树, 当前结点的数据结构只需要记录当前状态和它的目标函数值。爬山法不会预测与当前状态不直接相邻的那些状态的值。这就像健忘的人在大雾中试图登珠穆朗玛峰一样。爬山法的具体算法如下:

Function HILL-CLIMBING(problem) returns a state that is a local maximum

```

Inputs: a problem
Local variables: current, a node
neighbor, a node
current ← MAKE - NODE (INITIAL - STATE [problem])
Loop do
neighbor ← a highest - valued successor of current
if VALUE [neighbor] ≤ VALUE [current] then return STATE [current]
current ← neighbor
End Loop

```

利用八皇后问题说明爬山法算法。局部搜索算法通常使用完全状态形式化,即每个状态都表示为在棋盘上放 8 个皇后,每列一个。后继函数返回的是移动一个皇后到和它同一列的另一个方格中的所有可能的状态(因此每个状态有 $8 \times 7 = 56$ 个后继)。启发式耗散函数 h 是可以彼此攻击的皇后对的数量,不管中间是否有障碍。该函数的全局最小值是 0,仅在找到完美解时才能得到这个值。图 3.11(a)显示了一个 $h=17$ 的状态。图中还显示了它的所有后继的值,最好的后继是 $h=12$ 。爬山法算法通常在最佳后继的集合中随机选择一个进行扩展,如果这样的后继多于一个的话。

爬山法有时称为贪婪局部搜索,因为它只是选择邻居状态中最好的一个,而事先不考虑之后的下一步。尽管贪婪算法是盲目的,但往往是有效的。爬山法能很快朝着解的方向发展,因为它通常很容易改变一个坏的状态。例如,从图 3.11(a)中的状态,它只需要 5 步就能到达图 3.11(b)中的状态,它的 $h=1$,这基本上很接近于解了。可是,爬山法经常会遇到下面的问题:

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	Q	13	16	13	16
Q	14	17	15	Q	14	16	16
17	Q	16	18	15	Q	15	Q
18	14	Q	15	15	14	Q	16
14	14	13	17	12	14	12	18

(a) $h=17$

						Q	
				Q			
	Q						
			Q				
					Q		
							Q
		Q					
Q							

(b) $h=1$

图 3.11 八皇后问题的爬山搜索示意图

1. 局部极大值

局部极大值是一个比它的每个邻居状态都高的峰顶,但是比全局最大值要低。爬山法算法到达局部极大值附近就会被拉向峰顶,然后被卡在局部极大值处无处可走。更具体地,图 3.11(b)中的状态事实上是一个局部极大值(即耗散 h 的局部极小值);不管移动哪个皇后,得到的情况都不会比原来差。

2. 山脊

山脊造成的是一系列的局部极大值,贪婪算法处理这种情况是很难的。

3. 平顶区

平顶区是在状态空间地形图上估价函数值平坦的一块区域。它可能是一块平的局部极大值,不存在上山的出路,或者是一个山肩,从山肩还有可能取得进展。爬山法搜索可能无法找到离开高原的道路。

在各种情况下,爬山法算法都会达到无法取得进展的状态。从一个随机生成的八皇后问题的状态开始,最陡上升的爬山法 86% 的情况下会被卡住,只有 14% 的问题实例能求解。这个算法速度很快,成功找到最优解的平均步数是 4 步,被卡住的平均步数是 3 步,对于包含 88 个状态的状态空间,这已经是不错的结果了。

前面描述的算法中,如果到达一个平顶区,最佳后继的状态值和当前状态值相等时将会停止。如果平顶区其实是山肩,继续前进(即侧向移动)通常是一种好方法。注意,如果在没有上山移动的情况下总是允许侧向移动,那么当到达一个平坦的局部极大值而不是山肩的时候,算法会陷入无限循环。一种常规的解决办法是设置允许连续侧向移动的次數限制。例如,在八皇后问题中允许最多连续侧向移动 100 次。这使问题实例的解决率从 14% 提高到了 94%。成功的代价是:算法对于每个成功搜索实例的平均步数为大约 21 步,每个失败实例的平均步数为大约 64 步。

针对爬山法的不足,有许多变化的形式。例如,随机爬山法,它在上山移动中随机地选择下一步,选择的概率随着上山移动的陡峭程度而变化。这种算法通常比最陡上升算法的收敛速度慢不少,但是在某些状态空间地形图上能找到更好的解。再如,首选爬山法,它在实现随机爬山法的基础上,采用的方式是随机地生成后继结点,直到生成一个优于当前结点的后继。这个算法在有很多后继结点的情况下有很好的效果。

到现在为止所描述的爬山法算法还是不完备的,它们经常会在目标存在的情况下因为被局部极大值卡住而找不到该目标。一种改进方法是随机重新开始的爬山法,它通过随机生成的初始状态来进行一系列的爬山法搜索,找到目标时停止搜索。这个算法是完备的概率接近于 1,原因是它最终会生成一个目标状态作为初始状态。如果每次爬山法搜索成功的概率为 p ,那么需要重新开始搜索的期望次数为 $1/p$ 。对于不允许侧向移动的八皇后问题实例, $p \approx 0.14$,因此大概需要 7 次迭代就能找到目标(6 次失败,1 次成功)。所需步数的期望值为一次成功迭代的搜索步数加上失败的搜索步数与 $(1-p)/p$ 的乘积,大约是 22 步。如果允许侧向移动,则平均需要迭代约 $1/0.94 \approx 1.06$ 次,平均步数为 $1 \times 21 + 0.06/0.94 \times 64 \approx 25$ 步。对于八皇后问题,随机重新开始的爬山法是非常有效的,甚至对于三百万个皇后,这个方法也用不了一分钟就可以找到解。

爬山法算法成功与否在很大程度上取决于状态空间地形图的形状。如果在图中几乎没有局部极大值和高原,随机重新开始的爬山法将会很快地找到好的解。当然,许多实际问题的地形图存在着大量的局部极值。NP 难题通常有指数级数量的局部极大值。尽管如此,经过少数随机重新开始的搜索之后还是能找到一个合理的、较好的局部极大值的。

3.6.2 遗传算法原理

遗传算法(Genetic Algorithm, GA)是由密歇根大学的约翰·亨利·霍兰德(John Henry Holland)和他的同事于20世纪60年代在对细胞自动机进行研究时率先提出的。在20世纪80年代中期之前,对于遗传算法的研究还仅仅限于理论方面,直到在匹兹堡召开了第一届世界遗传算法大会。随着计算机的计算能力的发展和实际应用需求的增多,遗传算法逐渐进入实际应用阶段。1989年,《纽约时报》作者约翰·马科夫(John Markoff)写了一篇文章描述第一个用于商业用途的遗传算法——进化者。之后,越来越多的遗传算法出现并被用于许多领域中,《财富》杂志评出的500强企业中大多数都用它进行时间表安排、数据分析、未来趋势预测、预算以及很多其他组合优化问题的解决。

1. 遗传算法的基本原理

在遗传算法里,优化问题的解被称为个体,它被表示为一个参数列表,叫做染色体或者基因串。染色体一般被表示为简单的数字串,不过也有其他的表示方法,这一过程称为编码。一开始,算法随机生成一定数量的个体,有时候操作者也可以对这个随机产生过程进行干预,播下已经部分优化的种子。在每一代中,每个个体都被单独评价,并通过计算适应度的函数得到一个适应度数值。种群中的个体被按照适应度排序,适应度高的在前面。

下一步是产生下一代个体并组成种群。这个过程是通过选择、交叉、变异完成的。选择是根据新个体的适应度进行的,适应度越高,被选择的机会越高,而适应度低的,被选择的机会越低。初始的数据可以通过这样的选择过程组成一个相对优化的群体。之后,被选择的个体进行交叉,一般的遗传算法都有一个交叉概率,每两个个体通过交叉产生两个“新”个体,代替原来的“老”个体,而不交叉的个体则保持不变。

再下一步是变异,通过变异产生新的“子”个体。一般遗传算法都有一个固定的变异常数,通常是0.1或更小,这代表变异发生的概率。根据这个概率,新个体的染色体随机地突变,通常就是改变染色体的一个位(0变到1,或者1变到0)。

经过这一系列的过程(选择、交叉和变异),产生不同于初始一代的新一代个体,并一代一代地向增加整体适应度的方向发展,因为最好的个体总是更多地被选择去产生下一代,而适应度低的个体逐渐被淘汰掉。这样的过程不断地重复,直到满足终止条件为止。

基本遗传算法的流程如下。

```
initialize the population
loop until the termination criteria is satisfied
  for all individuals of population
    sum of fitness + = fitness of this individual
  end for
  forall individuals of population
    probability = sum of probabilities + (fitness/sum of fitness)
    sum of probabilities + = (fitness/sum of fitness)
  end for
loop until new population is full
```

```

do this twice
    for all members of population
        if rand(0,1)> probability but less than next probability
            then you have been selected
        end for
    end
    if rand(0,1)< crossover probability
        then crossover the two selected individuals and return the new two
    end loop
for all individuals of population
    if rand(0,1)< mutate probability
        then mutate the individual
    end for
endloop
now we get the optimum solution

```

2. 遗传算法的应用示例

接下来将使用遗传算法进行函数极值的求解,以此演示遗传算法的基本流程。

例 3.4 设目标函数是 $f(x)=x^2$, 约束条件为 $x=0,1,2,\dots,30,31$, 求解 $f(x)$ 的最大值。

此问题很容易用其他方法求解,但用此简单问题为例的意义在于说明遗传算法的流程,而且其足够简单。

1) 编码

遗传算法的工作对象是字符串,因此编码是一项基本性的工作。从生物学角度看,编码相当于选择遗传物质,每个字符串对应一个染色体。遗传算法大多采用二进制的 0/1 字符编码。当问题比较简单时,每一位 0/1 变量就代表一个性质。当问题的性质要用数值进行描述时,就涉及二进制数与十进制数的转换。对于长度(位数)为 L 的 0/1 字符串,按数学的排列组合计算,它可以表达 2^L 个数,十进制数与二进制数有如下关系。

$$x = x_{\min} + \frac{x_{\max} - x_{\min}}{2^L - 1} \text{Dec}(y)$$

其中, $x_{\min}, x_{\max}$ 为最小及最大的十进制数, y 为对应于 x 的二进制数, Dec 表示将二进制数转化为十进制数。在这种换算关系下,二进制表示法的精度 δ 为:

$$\delta = \frac{x_{\max} - x_{\min}}{2^L - 1}$$

由上式可算得所需的位数,进而可得知两个相邻十进制的间隔。例如,如果 $x_{\min} = -1, x_{\max} = 1, \delta = 0.5$, 则可得 $L = 3$, 间隔为 $\frac{2}{7}$, 求解空间中有 8 个实数,即 $-1, -1 + \frac{2}{7}, \dots, -1 + \frac{12}{7}, 1$ 。

对于兼有多种性质的问题,可以将描述各种性质的字符串组合在一起,用一长字符串表示。例如,可选 25 位 0/1 字符串表示物体的体积、重量及材质,其中前 10 位数表示体

积,中间 10 位表示重量,后 5 位表示材质。

上述都是针对二进制编码的,遗传算法也可以采用实数编码,即不需要将原始数据变化为二进制数,以原始数据表示染色体即可,最简单的染色体就可仅用一个实数表示。二进制编码的缺陷是在限定码长的情况下所能表示的精度不够,容易导致进化不收敛;而如果要满足一定的精度约束,则必须增加编码长度,搜索空间也将相应增大,从而影响整个进化过程的速度。实数编码的优点是直观,且克服了二进制编码的弊端,而这样做的代价是需要重新设计遗传操作,因为原来针对二进制的交叉、变异策略不再适用。

本节提出的问题是求 $f(x)=x^2$ 的最大值,其编码较为简单,需要 5 个二进制位来表示自变量。

2) 产生初始群体

初始群体是遗传搜索寻优的出发点。群体规模 M 越大,搜索的范围越广,但是每代的遗传操作时间越长。反之, M 越小,每代的运算时间越短,然而搜索空间也越小。初始群体中的每个个体都是按随机的方法产生的。根据串的长度 L ,随机产生 L 个 0/1 字符组成初始个体。在此问题中可以令 $M=4$,一个可能的初始种群是 01101,11000,01000,10011。

3) 计算适应度

适应度是衡量个体优劣的标志,它是执行遗传算法“优胜劣汰”的依据。因此,适应度也是驱使遗传算法向前发展的动力。通常,遗传算法中个体的适应度也就是所研究问题的目标函数,但是,有时适应度是目标函数转换后的结果。

为了方便讨论,此处遗传算法只研究目标变量 x 大于零的最大值问题。对于最小值问题,其适应度可以按下面的式子转换。

$$f(x)=\begin{cases} C_{\max}-g(x) & g(x)<C_{\max} \\ 0 & \text{其他} \end{cases}$$

其中, $f(x)$ 为转换后的适应度, $g(x)$ 为原适应度, $C_{\max}$ 为足够大的常数。

本节的问题中,由于是二进制编码,所以首先要有一个解码的过程,即将二进制串解码为十进制的实数,这也被称为从基因型到表现型的转化,01101→13,11000→24,01000→8,10011→19。根据目标函数 $f(x)=x^2$,可以计算出种群中 4 个个体的适应度为 13→169,24→576,8→64,19→361。

4) 选择

在遗传算法中,通过选择操作将优良个体插入下一代新群体,体现“优胜劣汰”的原则。选择优良个体的方法通常采用轮盘法。轮盘法的基本精神是个体被选中的概率取决于个体的相对适应度。

$$p_i = f_i / \sum f_i$$

其中, p_i 为个体 i 被选中的概率, f_i 为个体 i 的适应度。

显然,个体适应度越高,被选中的概率就越大。但是,适应度小的个体也有可能被选中,以便增加下一代群体的多样性。从统计的意义讲,适应度越大的个体,其刻度长,被选中的可能性就大。

5) 交叉

在遗传算法中,交叉是产生新个体的主要手段。它类似于生物学的杂交,使不同个体的基因互相交换,从而产生新个体。单点交叉操作如下所述。

01101,11000→01100,11001

10011,11000→10000,11011

分别为两对染色体的交叉,第一对的随机交叉位置为4,第二对为2。一对染色体之间是否进行交叉操作,取决于交叉概率。

除了单点交叉之外,还有多点交叉和均匀交叉,单点交叉可看作多点交叉的特例。上述交叉算子同时适用于二进制编码和实数编码。除此之外,针对实数编码的还有中间交叉、启发式交叉等。

6) 变异

变异是遗传算法产生新个体的另一种方法,对于二进制编码来说就意味着某位由1变为0或由0变为1。变异有局部变异和全局变异之分,局部变异是指从种群中随机选取一个位进行取反操作,全局变异则指种群的每一位都有一个取反的几率,或者是每个个体随机选择一个位置进行变异。

7) 终止

算法在迭代若干次后终止,一般终止条件有:进化次数限制;计算耗费的时间限制;一个个体已经满足最优值的条件,即最优值已经找到;适应度已经达到饱和,继续进化不会产生适应度更好的个体;人为干预;以上两种或更多种的组合。

3.6.3 遗传算法的应用实例

这是一个简单的遗传算法实现函数优化问题描述:假设平面 xOy 中有3个点位于 $O(0,0), P(12,0), Q(8,6)$,请找到点 R 使 $W=5|RO|+4|RP|+3|RQ|$ 达到最低值。

```
1    #include<iostream>
2    #include<cstdio>
3    #include<cstdlib>
4    #include<cmath>
5    #include<ctime>
6    using namespace std;
7    /* 参数设置: 目标规模、最大代数、没有问题变量、交叉的顺从性、突变的概率、
8    最好的个体、输出文件 */
9    #define POPSIZE 200
10   #define MAXGENS 1000
11   #define NVAR 2
12   #define PXOVER 0.75
13   #define PMUTATION 0.15
14   #define TRUE 1
15   #define FALSE 0
16   #define LBOUND 0
```

```
17  #define UBOUND 12
18  #define STOP 0.001
19  int generation;
20  int cur_best;
21  double diff;
22  FILE * galog;
23  /* 结构基因型: 定义一串基因变量并确定基因变量取值上确界和下确界, 定义个体适
24  应值、个体适应值占种群适应值以及个体适应值的累加比例 */
25  struct genotype{
26      double gene[NVARS];
27      double upper[NVARS];
28      double lower[NVARS];
29      double fitness;
30      double rfitness;
31      double cfitness;
32  };
33  /* population 当前种群, population[POPSIZE]用于存放个体最优值并假设最优个体能存
34  活下去 */
35  /* 在某些遗传算法中最优值个体并不一定能够存活下去 */
36  struct genotype population[POPSIZE + 1];
37  /* 新一代取代了老一代得到子种群 */
38  struct genotype newpopulation[POPSIZE + 1];
39  /* 遗传算法使用的程序声明: 初始化函数、随机函数、目标函数、评价函数、保留最优个
40  体、当前种群与子代种群最优值比较、基因重组函数、交换函数、基因突变函数、数据记录
41  函数 */
42  void initialize(void);
43  double randval(double, double);
44  double funtion(double x1, double x2);
45  void evaluate(void);
46  void keep_the_best(void);
47  void elitist(void);
48  void select(void);
49  void crossover(void);
50  void swap(double *, double *);
51  void mutate(void);
52  double report(void);
53
54
55
56  /* 初始化函数: 初始化变量边界内的基因值。它还初始化每个数量的点的所有适
57  应值 */
58  void initialize(void){
59      int i, j;
60      for(i = 0; i < NVARS; i++){
61          for(j = 0; j < POPSIZE + 1; j++){
62              if(!i){
63                  population[j].fitness = 0;
64                  population[j].rfitness = 0;
```

```
65         population[j].cfitness = 0;
66     }
67     population[j].lower[i] = LBOUND;
68     population[j].upper[i] = UBOUND;
69     population[j].gene[i] = randval(population[j].lower[i], population[j].upper[i]);
70 }
71 }
72 }
73 /* 随机值生成器: 在边界内生成一个值 */
74 double randval(double low, double high){
75     double val;
76     val = ((double)(rand() % 10000)/10000) * (high - low) + low;
77     return val;
78 }
79 double funtion(double x, double y) {
80     double result1 = sqrt(x * x + y * y) + sqrt((x - 12) * (x - 12) + y * y) + sqrt((x - 8) *
81 (x - 8) + (y - 6) * (y - 6)); //目标函数
82     return result1;
83 }
84 /* 评价函数给出个体适应值, 每次更改函数时, 代码都必须重新进行 */
85 void evaluate(void){
86     int mem;
87     int i;
88     double x[NVARS];
89     for(mem = 0; mem < POPSIZE; mem++){
90         for(i = 0; i < NVARS; i++){
91             x[i] = population[mem].gene[i];
92         }
93         /* 将目标函数值作为适应值 */
94         population[mem].fitness = funtion(x[0], x[1]);
95     }
96     /* 找出种群中的个体最优值并将其移动到最后一位 */
97     void keep_the_best(){
98         int mem;
99         int i;
100         cur_best = 0;
101         /* 找出最高适应值个体并将最优个体复制至 population[POPSIZE] */
102         for(mem = 0; mem < POPSIZE; mem++){
103             if(population[mem].fitness < population[cur_best].fitness){
104                 cur_best = mem;
105             }
106         }
107         /* 防止出现种群基因退化故保留历史最优个体
108         * / if(population[cur_best].fitness <= population[POPSIZE].fitness ||
109         population[POPSIZE].fitness < 1){
110             population[POPSIZE].fitness = population[cur_best].fitness;
```

```
111         for(i = 0; i < NVAR; i++)
112             population[POPSIZE].gene[i] = population[cur_best].gene[i];
113     }
114 }
115 /* 精英功能: 上一代中最好的成员存储在数组中的最后一个成员。如果来自新普
116 通人的最佳个体优于以前人口中的最佳个体, 则从新人群中复制最佳成员; 否则
117 用上一代人中最好的人替换当前人口中最差的人, 防止种群最优值退化 */
118 void elitist(){
119     int i;
120     double best, worst;           //适应值
121     int best_mem, worst_mem;      //序号
122     best_mem = worst_mem = 0;
123     best = population[best_mem].fitness; //最高适应值初始化
124     worst = population[worst_mem].fitness; //最低适应值初始化
125     for(i = 1; i < POPSIZE; i++) //找出最高和最低适应值算法有待改进{
126         if(population[i].fitness < best){
127             best = population[i].fitness;
128             best_mem = i;
129         }
130         if(population[i].fitness > worst){
131             worst = population[i].fitness;
132             worst_mem = i;
133         }
134     }
135     if(best <= population[POPSIZE].fitness){
136         for(i = 0; i < NVAR; i++)
137             population[POPSIZE].gene[i] = population[best_mem].gene[i];
138     population[POPSIZE].fitness = population[best_mem].fitness;
139     }
140     else{
141         for(i = 0; i < NVAR; i++)
142             population[worst_mem].gene[i] = population[POPSIZE].gene[i];
143     population[worst_mem].fitness = population[POPSIZE].fitness;
144     }
145 }
146 /* 选择功能: 标准比例选择, 用于结合精英模型的最大化问题, 确保最佳成员幸
147 存, 筛选函数并产生子代 */
148 void select(void){
149     int mem, i, j;
150     double sum = 0;
151     double p;
152     for(mem = 0; mem < POPSIZE; mem++){
153         sum += population[mem].fitness;
154     }
155     for(mem = 0; mem < POPSIZE; mem++){
156         population[mem].rfitness = population[mem].fitness/sum;
157     }
158     population[0].cfitness = population[0].rfitness;
```

```
159     for(mem = 1; mem < POPSIZE; mem++){
160         population[mem].cfitness = population[mem - 1].cfitness + population[mem].rfit
161     ness;
162     }
163     for(i = 0; i < POPSIZE; i++){
164         p = rand() % 1000/1000.0;
165         if(p < population[0].cfitness){
166             newpopulation[i] = population[0];
167         }
168     else{
169         for(j = 0; j < POPSIZE; j++){
170             if(p >= population[j].cfitness && p < population[j + 1].cfitness)
171 newpopulation[i] = population[j + 1];
172         }
173     }
174     for(i = 0; i < POPSIZE; i++)//子代变父代
175         population[i] = newpopulation[i];
176 }
177 /* Crossover: 执行所选父项的交叉, 基因重组函数 */
178 void Xover(int one, int two){
179     int i;
180     int point;
181     if(NVARS > 1){
182         if(NVARS == 2)
183             point = 1;
184         else
185             point = (rand() % (NVARS - 1)) + 1;
186         for(i = 0; i < point; i++) //只有第一个基因发生重组
187             swap(&population[one].gene[i], &population[two].gene[i]);188
189     }
190 /* 交换: 交换过程有助于交换 2 个变量 */
191 void swap(double *x, double *y){
192     double temp;
193     temp = *x;
194     *x = *y;
195     *y = temp;
196 }
197 /* 交叉功能: 选择参与交叉的两个父母, 实现单点交叉, 杂交函数 */
198 void crossover(void){
199     int mem, one;
200     int first = 0;
201     double x;
202     for(mem = 0; mem < POPSIZE; ++mem){
203         x = rand() % 1000/1000.0;
204         if(x < PXOVER){
205             ++first;
206 /* 选择杂交的个体对, 杂交有待改进, 事实上往往是强者与强者杂交, 这里没有考
207 虑雌雄与杂交对象的选择 */
```

```

208             if(first % 2 == 0)
209                 Xover(one, mem);
210             else
211                 one = mem;
212         }
213     }
214 }
215 /* 突变功能: 随机均匀突变。选择用于突变的变量被变量的下限和上限之间的随
216 机值替换。变异函数: 事实基因的变异往往具有某种局部性 */
217 void mutate(void){
218     int i, j;
219     double lbound, hbound;
220     double x;
221     for(i = 0; i < POPSIZE; i++){
222         for(j = 0; j < NVAR; j++){
223             x = rand() % 1000/1000.0;
224             if(x < PMUTATION){
225                 lbound = population[i].lower[j];
226                 hbound = population[i].upper[j];
227                 population[i].gene[j] = randval(lbound, hbound);
228             }
229         }
230     }
231 /* 报告功能: 报告模拟的进度。双重报告(无效) */
232 double report(void){
233     int i;
234     double best_val;                //种群内最优适应值
235     double avg;                    //平均个体适应值
236     //double stddev;
237     double sum_square;             //种群内个体适应值平方和
238     //double square_sum;
239     double sum;                   //种群适应值
240     sum = 0.0;
241     sum_square = 0.0;
242     for(i = 0; i < POPSIZE; i++){
243         sum += population[i].fitness;
244         sum_square += population[i].fitness * population[i].fitness;
245     }
246     avg = sum/(double)POPSIZE;
247     //square_sum = avg * avg * (double)POPSIZE;
248     //stddev = sqrt((sum_square - square_sum)/(POPSIZE - 1));
249     best_val = population[POPSIZE].fitness;
250     fprintf(galog, "%6d %6.3f %6.3f %6.3f %6.3f %6.3f\n", generation, best_
251     val, population[POPSIZE].gene[0], population[POPSIZE].gene[1], avg, sum);
252     return avg;
253 }
254 /* 主要功能: 每一代都涉及选择最佳成员, 执行交叉和变异, 然后评估结果总体,
255 直到满足终止条件 */

```

```

256 void main(void){
257     int i;
258     double temp;
259     double temp1;
260     if((galog = fopen("data.txt", "w")) == NULL){
261         exit(1);
262     }
263     generation = 1;
264     srand(time(NULL));
265     fprintf(galog, "number value x1 x2 avg sum_value\n");
266     printf("generation best average standard\n");
267     initialize();
268     evaluate();
269     keep_the_best();
270     temp = report();           //记录, 暂存上一代个体平均适应值
271     do{
272         select();             //筛选
273         crossover();          //杂交
274         mutate();             //变异
275         evaluate();           //评价
276         keep_the_best();
277         elitist();
278         temp1 = report();
279         diff = fabs(temp - temp1);
280         temp = temp1;
281         generation++;
282     }
283     while(generation < MAXGENS && diff >= STOP);
284     //fprintf(galog, "\n\n Simulation completed\n");
285     //fprintf(galog, "\n Best member:\n");
286     printf("\nBest member:\n generation: %d\n", generation);
287     for(i = 0; i < NVAR; i++){
288         //fprintf(galog, "\n var( %d) = %3.3f", i, population[POPSIZE].gene[i]);
289         printf("X %d = %3.3f\n", i, population[POPSIZE].gene[i]);
290     }
291     //fprintf(galog, "\n\n bestfitness = %3.3f", population[POPSIZE].fitness);
292     fclose(galog);
293     printf("\nBest fitness = %3.3f\n", population[POPSIZE].fitness);
294 }

```

实验结果：如图 3.12 所示为实验得出的一组解，遗传算法进行了 677 次，找到 R 点坐标为(7.673, 3.251)，距离最小值为 16.514。

```

generation best average standard
Best member:
generation:677
X0=7.673
X1=3.251

Best fitness=16.514
Press any key to continue

```

图 3.12 实验解

3.7 习题

(1) 理解一般图搜索算法, OPEN 表和 CLOSE 表的作用是什么? 为何要标记从子结点到父结点的指针? 举例说明对 3 类子结点处理方式的差异。

(2) 什么是问题归约? 问题归约的操作算子与一般图的搜索有何不同? 为什么应用问题归约得到的状态空间可表示为与或图?

(3) 一个农夫带着一只狼、一只羊和一筐菜, 欲从河的左岸坐船到右岸, 由于船太小, 农夫每次只能带一样东西过河, 并且, 没有农夫看管的话, 狼会吃羊, 羊会吃菜。设计一个方案, 使农夫可以无损失地渡过河。

(4) 设计一个解决旅行商问题的爬山算法。

(5) 设计一个解决旅行商问题的遗传算法。

(6) 对上面 3 题的算法和结果进行比较分析。