第 3 章　搜索技术

本章我们学习通过搜索进行问题求解，搜索即是指从问题出发寻找解的过程。

1 搜索技术

通过搜索进行问题求解这章，讨论当问题求解不能通过单个行动一步完成时，Agent 如何找到一组行动序列达到目标。搜索即是指从问题出发寻找解的过程。

最简单的 Agent 是反射 Agent，这类 Agent 存有在何种状态下可采取何种行动的直接映射表，它们的行为就取决于这种映射。有些环境中 Agent 的这种映射表可能非常大，导致占用很多存储空间或者查表消耗的时间太长而无法学习，Agent 在这样的环境中难以运转。另一方面，基于目标的 Agent 会考虑将要采取的行动及行动的可能后果，即与目标还有多远。

本章讨论基于目标的 Agent 中的一种，称为问题求解 Agent（problem-solving Agent）。问题求解 Agent 使用原子表示：世界的状态被视为一个整体，对问题求解算法而言没有可见的内部结构。使用更先进的要素化或结构化表示的基于目标的 Agent，通常被称为规划 Agent。

要进行问题求解，首先要讨论的是对问题及其解的精确定义，我们将通过一些实例来说明如何描述一个问题及其解。接着我们介绍一些求解此类问题的通用的搜索算法。首先讨论无信息的（uninformed）搜索算法——无信息是指算法除了问题定义本身没有任何其他信息。尽管这些算法有的可以用于求解任何问题，但此类算法效率都不好。另一方面，有信息（Informed）的搜索算法，利用给定的知识引导能够更有效地找到解。

1.1 搜索问题的定义

1.1.1 问题定义

我们假设智能 Agent 要最大化其性能度量。如果 Agent 能采纳一个目标（goal）并试图去满足它，最大化性能度量的问题就可能会简化。我们首先讨论 Agent 为何这样做及该如何做。

想象一个 Agent 正在罗马尼亚的 Arad 享受旅游假期。该 Agent 的性能度量包含很多方面：它想要晒黑些，想学习罗马尼亚语，欣赏风景，享受夜生活（诸如此类），还要避免宿醉，等等。这个 Agent 决策问题有些复杂，要权衡许多方面的因素，同时要阅读大量的旅游指南。现在假设该 Agent 有一张第二天飞离 Bucharest 的不能改签也不能退的机票。在这种情况下，Agent 建立合理目标：抵达 Bucharest。导致不能按时到达 Bucharest 的行动方案将不再予以考虑，因此该 Agent 的决策问题被大幅度简化。Agent 的行动能力可能帮它达到各种目的，但是此时的目标限制了这些行动，帮助 Agent 组织行动序列，以达到最终目标。基于当前的情形和 Agent 的性能度量进行目标形式化（goal formulation）是问题求解的第一个步骤。

我们将目标考虑成是世界的一个状态集合——目标被满足的那些状态的集合。Agent 的任务是找出现在和未来如何行动，以使它达到一个目标状态。在 Agent 能做这个之前，它（或是我们代表它）需要确定它能完成的行动种类和行动能带来的状态变化。如果 Agent 试图在诸如“左脚前移 1 英尺”或“将方向盘向左旋转 1 度”的层次上考虑行动，它将可能永远无法找到走出停车场的路，更别说去 Bucharest 了，因为在那样的细节水平上世界的不确定性因素太多，而问题的解也将包含过多的步骤。问题形式化（problem formulation）是在给定目标下确定需要考虑哪些行动和状态的过程。后面我们将详细地讨论这个过程。现在我们假设 Agent 将在开车从一个主要城镇到另一个城镇的层次上考虑行动。因此每个状态表示 Agent 在一个特定的城镇中。

Agent 现在的目标是开车去 Bucharest，正在考虑从 Arad 先开往哪里。从 Arad 开出有三条道路分别前往 Sibiu、Timisoara 和 Zerind。这三条路没有一条能直接到达最终目标，所以除非 Agent 对罗马尼亚非常熟悉，它无法知道应该走哪条路'。换句话说，Agent 不知道这三条路中哪条路或哪个可能的行动是最好的，因为它对由每个行动之后的状态知道得不够多。如果 Agent 没有额外的知识，那么它就只能随机选择一个行动。这种糟糕的情况我们在第 4 章中讨论。

但是假设 Agent 有罗马尼亚的地图。地图上的每个点都可以向 Agent 提供信息：Agent 可以到达哪些状态和它可以采取哪些行动。Agent 可以利用这些信息假想整个旅程，考虑途经上述 3 个城镇后的后继阶段，试图找出最终能到达 Bucharest 的路。一旦 Agernt 在地图上发现从 Arad 到 Bucharest 的路，它就可以完成相应的驾驶行动来达到它的目标。一般来说，一个 Agent 在面临多种未知值的选择时，可以首先检查那些最终导出已知价值的状态的未来行动，然后做出决策。

下面我们解释何为“检查未来行动”，在此之前我们还应具体了解环境的性质，这点在 2.3 节中有定义。现在，我们假定环境是可观察的，所以 Agent 总是知道当前状态。Agent 在罗马尼亚开车，假设在司机到达地图上的每个城市时都会发现有标识标明该城市。我们假设该任务环境是离散的，所以在任一给定状态，可以选择的行动是有限的。这是正确的，因为在罗马尼亚游玩时，每个城市只与其他一小部分城市相邻。我们假设环境是已知的，所以 Agent 知道每个行动达到哪个状态。（拥有一份足够精确的地图以满足游玩问题的要求。）最后，我们假设环境是确定的，每个行动的结果只有一个。在理想条件下，这对罗马尼亚的 Agent 是正确的——这意味着如果 Agent 选择了从 Arad 开车前往 Sibiu，那么它一定会到达 Sibiu。当然，条件并不总是理想化的.

在这些假设下，任何问题的解是一个行动的固定序列。"当然!"，有人会说，"还能是什么?"也许，一般来说，它也可能是分支策略，在感知到抵达城市时会建议不同的行动选择。例如，在不够理想化的条件下，Agent 可能计划从 Arad 开往 Sibiu 接着前往 Rimnicu Vilca，但也可能要做好本来要去 Sibiu 结果却意外抵达 Zerind 的计划。幸运的是，如果 Agent 知道初始状态并且环境是已知的和确定的，它清楚地知道行动之后的状态。因为行动之后只有一个后果，问题求解才有可能继续选择后继的行动。

为达到目标，寻找这样的行动序列的过程被称为搜索。搜索算法的输入是问题，输出是问题的解，以行动序列的形式返回问题的解。解一旦找到，它所建议的行动将会付诸实施。这被称为执行阶段。那么，我们就完成了对 Agent 的简单设计，即“形式化、搜索、执行”，如图 3.1 所示。在完成对目标和对待求解问题的形式化之后，Agent 调用搜索过程进行问题求解。然后 Agent 用得到的解来导引行动，按照问题求解给出的解步骤逐一实施——通常是执行序列中的第一个行动——从序列中删除已完成的步骤。一旦解被执行，Agent 将形式化新的目标。

它首先对目标和问题进行形式化，然后搜索能够解决该问题的行动序列，最后依次执行这些行动。这个过程完成之后，它会形式化另一个目标并重复以上步骤

需要注意的是，Agent 在执行解的行动序列时，它无视它的感知信息，每当它选择了行动它都知道它将到达什么状态。Agent 在执行计划时是闭上了眼睛的，就是说，它十分确定行动后果是什么。控制理论把这称为开环系统，因为无视感知信息打破了 Agent 和环境之间的环路。

1.1.2 良定义的问题及解

一个问题可以用 5 个组成部分形式化地描述：

- Agent 的初始状态。例如，在罗马尼亚问题中 Agent 的初始状态可以描述为 Im（Arad。
- 描述 Agent 的可能行动。给定一个特殊状态 s，ACTIONS（s）返回在状态 s 下可以执行的动作集合。我们称这些行动对状态 s 是可应用的。例如，考虑状态 Im（Arad）

可应用的行动为：{Go(Sibiz),Go(Tmisoara),Go(Zerind)}

- 对每个行动的描述;正式的名称是转移模型，用函数 RESULT（s，a）描述：在状态 s 下执行行动 a 后达到的状态。我们也会使用术语后继状态来表示从一给定状态出发通过单步行动可以到达的状态集合 1。例如，RESuLT(In(Arad,.Go(Zerind)=Im(Zerind)

总之，初始状态、行动和转移模型无疑就定义了问题的状态空间———即从初始状态可以达到的所有状态的集合。状态空间形成一个有向网络或图，其中结点表示状态，结点之间的弧表示行动（图 3.2 中的罗马尼亚地图就可以被解释为一个状态空间图，每条连线视为双向驾驶行动即双向边）。状态空间中的一条路径指的是通过行动连接起来的一个状态序列。

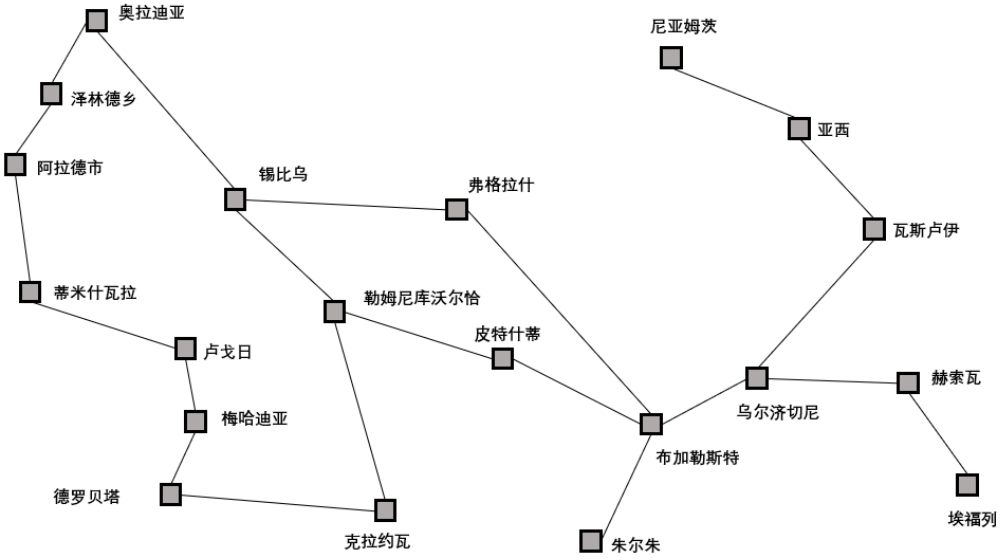


图 1.1 部分罗马尼亚地图的简化版

● 目标测试，确定给定的状态是不是目标状态。有时候目标状态是一个显式集合，测试只需简单检查给定的状态是否在目标状态集合中。在罗马尼亚问题中，目标状态集是一个单元素集合{In（Bucharest）}。有些时候目标状态并不是一个显式可枚举的目标状态集合，而是具备某些特定抽象属性的状态。例如，在国际象棋中，目标状态是指被"将死"的状态，即对方的国王在己方的攻击下已经无路可逃必死无疑。

● 路径耗散函数为每条路径赋一个耗散值，即边加权。问题求解 Agent 选择能反映它自己的性能度量的耗散函数。对于试图前往 Bucharest 的 Agent，时间是基本要素，所以它的路径耗散可以是用公里数表示的路径长度。在本章中，我们假设一条路径的耗散值为该路径上的每个行动（每条边）的耗散值总和 1。采用行动 a 从状态 s 走到状态 s 所需要的单步耗散用 c（s，a，s’）表示。罗马尼亚问题中的单步耗散即单步路程距离，如图 3.2 所示，我们假设单步耗散值是非负的[2]。

由上述元素即可定义一个问题，通常把它们组织在一起成为一个数据结构，并以此作为问题求解算法的输入。问题的解就是从初始状态到目标状态的一组行动序列。解的质量由路径耗散函数度量，所有解里路径耗散值最小的解即为最优解。

1.1.3 问题的形式化

上一节我们给出了罗马尼亚问题的形式化，用初始状态、行动、转移模型、目标测试路径耗散来描述。这种形式化看起来是合理的，不过它依然只是个模型——一种抽象的数学描述——不是真实的事情。比较我们选择的简单状态描述，In（Arad），和实际的越野前行，现实世界状态包括太多事情：同行的旅伴，收音机播放的节目，窗外的景色，附近是否有执法人员，到下一个休息点的距离、路况、天气情况等。我们选择的状态描述中不包括这些信息，因为它们与找到前往 Bucharest 的路径问题不相关。在表示中去除细节的过程被称为抽象。

不仅是状态描述要抽象，我们还需要对行动进行抽象。一个驾驶行动会造成很多影响。驾驶行为不仅改变了车辆和它的乘客的位置，它还花费了时间，消耗了汽油，产生了污染，以及改变了 Agent 自身（就像他们所说的，旅行拓展了视野）。我们的形式化则只考虑了位置的变化。我们同时也忽略了许多其他行动：打开收音机，欣赏窗外的景色，遇到执法人员而减速，等等。我们当然更不会将行动细节到"把方向盘向左转 1 度"这种层次上。

我们是否能够精确地定义合适的抽象层次?考虑对应于现实的世界状态和现实的行动序列，我们在前面选择了抽象化的状态和行动。现在考虑罗马尼亚问题抽象后的一个解：例如，从 Arad 到 Sibiu 到 Rimnicu Vilcea 到 Pitesti 到 Bucharest 的解路径。这个抽象解可以对应大量的更细节的解路径。例如，我们在从 Sibiu

开往 Rimnicu Vilcea 的途中听收音机，然后在剩下的旅途中关掉收音机。如果我们能够把任何抽象解扩展成为更细节的世界中的解，这种抽象就是有效的;一个充分条件是对于每个抽象为"在 Arad"的细节状态都有一条详细路径到达一些如"在 Sibiu"的状态，等等[3]。如果执行解中的每个行动比原始问题中的容易，那么这种抽象是有用的;在这种情况下，解路径中的每个行动要足够容易，以至于对于平均水平的驾驶 Agent 而言不用更进一步地搜索或者规划就能实施了。因此，选择一个好的问题抽象，包括在保持有效抽象的前提下去除尽可能多的细节和确保抽象后的行动容易完成。如果缺乏能力去构建有用的问题抽象，智能 Agent 将会被现实世界完全淹没。

1.2 搜索的基本概念

在对问题进行形式化之后，我们现在需要对问题求解。一个解是一个行动序列，所以搜索算法的工作就是考虑各种可能的行动序列。可能的行动序列从搜索树中根结点的初始状态出发;连线表示行动，结点对应问题的状态空间中的状态。图 3.6 给出了求解罗马尼亚问题画搜索树的最初几步。搜索树的根结点对应于初始状态 ImArad）。第一步检测该结点是否为目标状态。（显然它不是目标状态，但是这步检测很重要，因为这样可以解决如"从 Arad 出发，到达 Arad"的问题。）下面我们就要考虑选择各种行动。这是通过扩展当前状态完成的;即，在当前状态下应用各种合法行动，由此生成了一个新的状态集。在这个问题中，从父结点 In（drad 出发得到三个新的子结点：In（Sbut），Im（Tmisoara）和 Im（Zerind）。现在我们需要从这三种可能性中选择其一继续考虑。

这就是搜索——选择一条路往下走，把其他的选择暂且放在一边，等以后发现第一个选择不能求出问题的解时再考虑。假设我们首先选择 Sibiu。检查它是否为目标状态（不是），然后扩展它得到四个状态：In（drad），Im（Fagaras），In（Oradea）和 In（RimnicuVicea）。现在我们的选择包括这四个状态，以及 Timisoara 和 Zerind。这六个结点都是叶结点，在当前的搜索树中没有子结点。在任一给定时间点，所有待扩展的叶结点的集合称为边缘。（很多作者称之为开结点表，这种说法不容易记忆也不精确，原因是其他的数据结构比表更合适。）在图 3.6 中，搜索树中的边缘包括那些粗实线的结点。

在边缘中选择结点并扩展的过程一直继续，直到找到了解或者已经没有状态可扩展。在图 3.7 中给出了一般的树搜索算法。搜索算法的基本结构大多如此;区别主要在如何选择将要扩展的状态———即搜索策略。

细心的读者可能已经发现图 3.6 中有些特别：它包括了从 Arad 到 Sibiu 然后又回到 Arad 的路径!这时 InArad）是搜索树中的重复状态，生成了一个有环路的路径。考虑这样的有环路径，这意味着罗马尼亚问题的完整搜索树是无限的，因为环路是没有限制的。另一方面，状态空间——如图 3.2 中所示——只有 20 个状态。我们在 3.4 节会讨论，循环会导致算法失败，会导致有解的问题无法求得解。幸运的是，我们无须考虑有环的路径。我们可以依赖直觉这样做：由于路径代价是递增的并且每一步的代价都是非负数，通向某一给定状态的有环路径都不会比去掉那个环路的好。

有环路径是冗余路径的一种特殊情况，在两个状态之间的迁移路径多于一条时这种情况可能就会发生。考虑路径 Arad-Sibiu（路径长度 140 公里）和 Arad-Zerind-Oradea-Sibiu （路径长度 297 公里）。显然，后一条路径是冗余的——是达到同一状态的较差方法。如果你关心最终目标，那么对到达任一给定状态都没有必要记录超过一条路径，因为通过一种途径如果可以到达目标状态，通过其他途径同样也可以。

有些情况下，通过定义问题本身可以减少冗余路径。例如，在我们形式化八皇后问题时每个皇后可以放在任一列中，那么到达 n 后问题的每个状态有 n!个不同的路径;但是如果我们在形式化此问题时，定义每个皇后只能放在最左侧的空列中，那么每个状态就只能通过一条路径抵达。

有些问题中，冗余状态是不可避免的。这里指的是问题中的行动是可逆的，如交通找路问题和滑块问题，这样的情况下冗余状态不可避免。矩形网格中的寻径问题（后面的图 3.9）在计算机游戏中极为重要。在这样的网格中，每个状态有四个后继状态，所以包括重复状态的深度为 d 的搜索树有 4d 个叶结点;但是事实上对任一给定状态 d 步内只有大概 2d2 个确定的状态。设 d=20，这意味着搜索树会有几万亿个结点，但事实上我们只有大约 800 个确定的状态。所以，如果不处理冗余路径会使得可解问题变得不可解。即使算法知道如何避免死循环，处理冗余路径依然重要。

正如前面所说，遗忘历史的算法将会不幸的重复历史。避免探索冗余路径的方法是牢记曾经走过的路。为了做到这一点，我们给 TREE-SEARCH 算法增加一个参数———这个数据结构称为探索集（也被称为 closed 表），用它记录每个已扩展过的结点。新生成的结点若与已经生成的某个结点相匹配的话——即是在探索集中或是边缘集中——那么它将被丢弃而不是被加入边缘集中。新算法叫 GRAPH-SEARCH，如图 3.7 所示。本章中的特定算法都具有这种一般结构。

清楚的是，GRAPH-SEARCH 算法构造的搜索树中每个状态至多只包含一个副本，所以我们可以直接在状态空间图中生长一棵树，如图 3.8 所示。这个算法还有另一个好的特点：边缘将状态空间图分成了已探索区域和未被探索区域，因此从初始状态出发至任一未被探索状态的路径都不得不通过边缘中的结点。这种特点如图 3.9 所示。每个步骤要么将一个状态从边缘变为已探索区域，要么将未探索区域变为边缘，我们看到算法系统地检查状态空间中的每一个状态，直到找到问题的解。

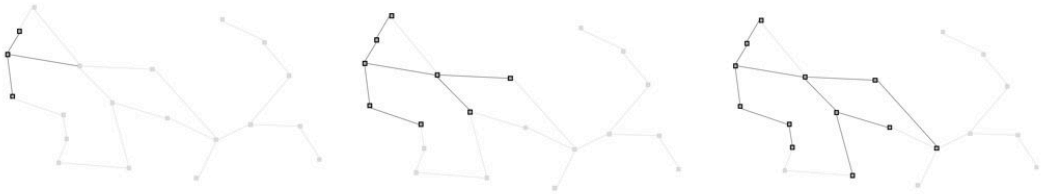
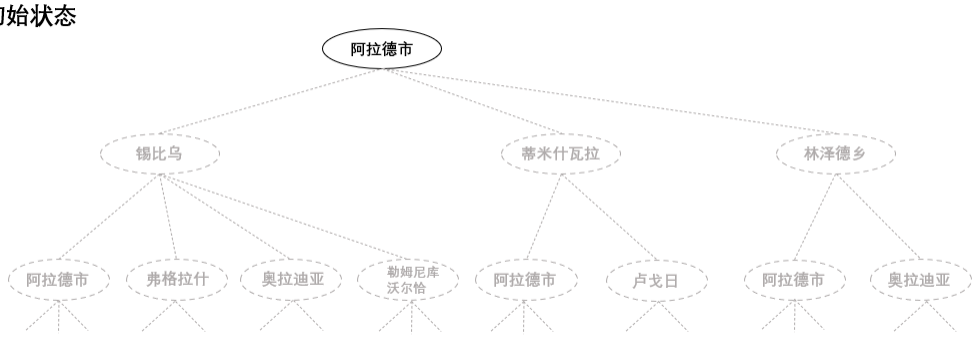
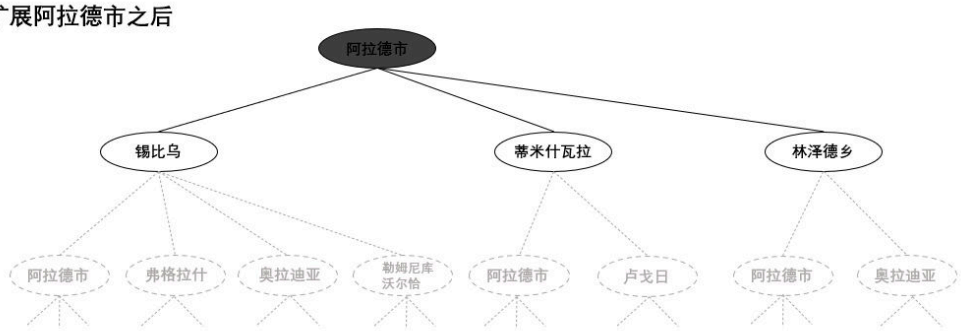


图 1.2 用 GRAPH-SEATCH 求解图 3.1 罗马尼亚问题搜索树的生长顺序，我们逐步扩展。要注意的是第三步，最北部城市奥拉迪亚已到尽头；他的两个后继都已经经由其他路径成为已扩展的

(a)初始状态



(b)扩展阿拉德市之后



(c)扩展锡比乌之后

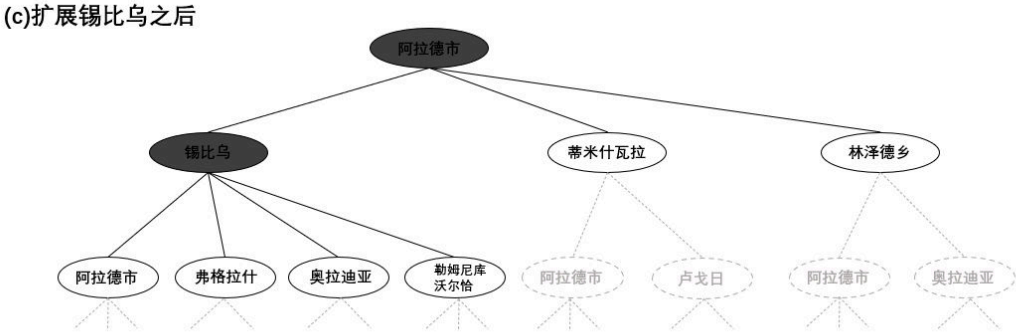


图 1.3　求解罗马尼亚问题的部分搜索树。要注意的是已被扩展过的结点用阴影表示；已经生成但未被扩展的结点用粗实线表示；尚未生成的结点用浅虚线表示

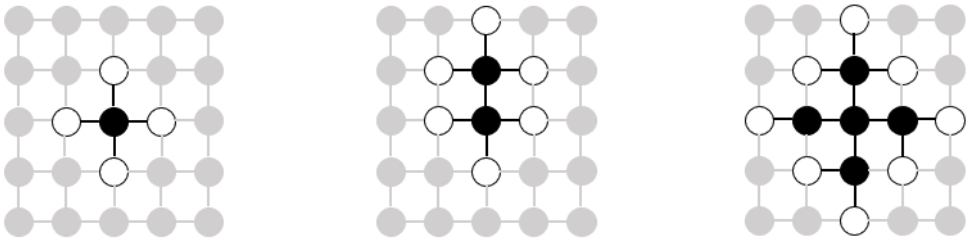


图 1.4 用矩阵网格问题看 GRAPH-SEARCH 算法的分离特点。边缘（白色结点）总是隔开了状态空间的已探索区域（黑色结点）和未被探索区域（灰色结点）。在（a）图中，只有根节点被探索过。在（b）图中一个叶结点被扩展。在（c）图中，根结点的后继以顺时针序被探索

1.3 盲目搜索

盲目搜索方法又叫非启发式搜索，是一种无信息搜索，一般只适用于求解比较简单的问题，盲目搜索通常是按预定的搜索策略进行搜索，而不会考虑到问题本身的特性。常用的盲目搜索有宽度优先搜索和深度优先搜索两种。

1.3.1 宽度优先搜索 (BFS)

宽度优先搜索是简单搜索策略，先扩展根节点，接着扩展根节点所有的后继，然后再扩展它们的后继，以此类推。一般地，在下一层任意节点扩展之前，搜索树上的所有节点都应该已经扩展过。

宽度优先搜索是一般图搜索算法的一个实例，每次总是扩展深度最浅的节点，这可以通过将边缘组织成 FIFO 队列来实现（即，新节点加入到队列尾，浅层的老节点会在深层节点之前被扩展）。

这可以通过将边缘组织成 FIFO 队列来实现。就是说，新结点(结点比其父结点深)加入到队列尾，这意味着浅层的老结点会在深层结点之前被扩展。对一般图搜索算法做简单修改，目标的测试是在结点被生成的时候，而不是结点被选择扩展的时候。我们会在讨论时间复杂度的时候解释这一点。要注意的是，算法具有一般的图搜索框架，忽视所有到边缘结点或已扩展结点的新路径；可以容易地看出，这样的路径至少和已经找到的一样深。所以，宽度优先搜索总是有到每一个边缘结点的最浅路径。

根据上节提到的 4 个标准,宽度优先搜索的性能怎样吧?很容易知道宽度优先搜索是完备，如果最浅的目标结点处于一个有限深度 d，宽度优先搜索在扩展完比它浅的有结点(假设分支因子 b 是有限的)之后最终一定能找到该目标结点。请注意目标结点一经生成,我们就知道它一定是最浅的目标结点,原因是所有比它的浅的结点在此之前已生成并且肯定未能通过目标测试。最的目标结点不一定就是最优的目标结点:从技术看,如果路径代价是基于结点深的非递减函数,宽度优先搜索是最优的。最常见的情况就是当所有的行动要花费相同的代价。

到目前为止我们讨论的宽度优先搜索的性能都是好的方面,但是它在时间和空间耗要上却不好。假设搜索一致树(uniform tree)的状态空间中每个状态都有b个后继。搜案树的根结点生成第一层的b个子结点,每个子结点又生成 b 个子结点,第二层则有b^2个结点这些结点的每一个再生成b个子结点。在第三层则得到b个结点,依此类推。现在假设解的深度为 d 在最环的情况下,解是那一层最后生成的结点。这时的结点总数为:

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

(如果算法是在选择要扩展的结点时而不是在结点生成时进行目标检测,那么在目标被检测到之前深度d上的其他结点已经被扩展,这时时间复杂度应为$O(b^{d+1})$。空间复杂度:对任何类型的图搜索,每个已扩展的结点都保存在探索集中,空间复杂度总是在时间复杂度的b分之一内。特别对于宽度优先图搜索,每个生成的结点都在内存中,那么将有$O(b^{d-1})$个结点在探索集中$O(b^d)$个结点在边缘结点集中,所以空间复杂度为$O(b^d)$即它由边结点集的大小所决定。即使转换为树的搜索问题也节省不了多大的存储空间,如果状态空间有重复路径的话,这种转换会耗费大量时间。指数级的复杂度$O(b^d)$今人担忧。一般而言，指数级别复杂度的搜索

问题不能用无信息的搜索算法求解，除非是规模很小的实例。

```
def  BFS(graph,s):#graph 图  s 指的是开始结点
    #需要一个队列
    queue=[]
    queue.append(s)
    seen=set()#看是否访问过该结点
    seen.add(s)
    while (len(queue)>0):
        vertex=queue.pop(0)#保存第一结点，并弹出，方便把他下面的子节点接入
        nodes=graph[vertex]#子节点的数组
        for w in nodes:
            if w not in seen:#判断是否访问过，使用一个数组
                queue.append(w)
                seen.add(w)
        print(vertex)
```

1.3.2 深度优先搜索（DFS）

深度优先搜索(depth-firstsearch）总是扩展搜索树的当前边缘结点集中最深的结点。搜索很快推进到搜索树的最深层，那里的结点没有后继。当那些结点扩展完之后，就从边缘结点集中去掉，然后搜索算法回溯到下一个还有未扩展后继的深度稍浅的结点。深度优先搜索算法是图搜索算法的实例，宽度优先搜索使用 FIFO 队列，而深度优先搜索使用 LIFO 队列。LIFO 队列指的是最新生成的结点最早被选择扩展。这一定是最深的未被扩展结点，因为它比它的父结点深 1，上一次扩展的则是这个父结点因为当时它最深。

作为一个可行的 TREE-SEARCH 实现，通常使用调用自己的递归函数来实现深度优先搜索算法，可以依次对当前结点的子结点调用该算法。

深度优先搜索算法的效率严重依赖与使用的是图搜索还是树搜索。避免重复状态和冗余路径的图搜索，在有限状态空间是完备的，因为它至多扩展所有结点。而树搜索，则不完备，会陷入死循环。深度优先搜索可以改成无需额外内存耗费，它只检查从根结点到当前结点的新结点；这避免了有限状态空间的死循环，但无法避免冗余路径。在无限状态空间中，如果遭遇了无限的又无法到达目标结点的路径，无论是图搜索还是树搜索都会失败。同样的原因，无论是基于图搜索还是树搜索的深度优先搜索都不是最优的

深度优先搜索的时间复杂度受限于状态空间的规模（当然，也可能是无限的）。另一方面．深度优先的树搜索，可能在搜索树上生成所有$O(b^m)$个结点，其中 m 指的是任一结点的最大深度：这可能比状态空间大很多。要注意的是 m 可能比 d（最浅解的深度）大很多，并且如果树是无界限的．m 可能是无限的。

这样看来，深度优先搜索与宽度优先搜索相比似乎没有任何优势，那我们为什么要考虑它？原因就在于空间复杂度。对图搜索而言，优势在于，深度优先搜索只需要存储一条从根结点到叶结点的路径，以及该路径上每个结点的所有未被扩展的兄弟结点即可。一旦一个结点被扩展，当它的所有后代都被探索过后该结点就从内存中删除。考虑状态空间分支因子为 b 最大深度为 m，深度优先搜索只需要存储$O(bm)$个结点。假设与目标结点在同一深度的结点没有后继，我们发现在深度 d=16 的时候深度优先搜索只需要 156K 字节而不是 10E 字节，节省了大约 7000 亿倍的空间。这使得深度优先搜索在 AI 的很多领域成为工作主力。

深度优先搜索的一种变形称为回溯搜索(backtracking search)，所用的内存空间更少。在回溯搜索中，每次只产生一个后继而不是生成所有后继；每个被部分扩展的结点要记住下一个要生成的结点。这样，内存只需要$O(m)$而不是$O(bm)$。回溯搜索催化了另一个节省内存（和节省时间）的技巧：通过直接修改当前的状态描述而不是先对它进行复制来生成后继。这可以把内存需求减少到只有一个状态描述以及$O(m)$个行动。为了达到这个目的，当我们回溯生成下一个后继时，必须能够撤销每次修改。对于状态描述相当复杂

的问题，例如机器人组装问题，这些技术是成功的关键。

```
def DFS(graph,s):
    #需要一个队列
    stack=[]
    stack.append(s)
    seen=set()#看是否访问过
    seen.add(s)
    while (len(stack)>0):
        #拿出邻接点
        vertex=stack.pop()#这里 pop 参数没有 0 了，最后一个元素
        nodes=graph[vertex]
        for w in nodes:
            if w not in seen:#如何判断是否访问过，使用一个数组
                stack.append(w)
                seen.add(w)
        print(vertex)
```

1.4 启发式搜索：启发函数、A*搜索

1.4.1 启发函数

本节我们将考察八数码问题的启发函数，以此为例探讨启发式的一般性质。八数码问题是最早的启发式搜索问题之一。在 3.2 节我们提到过，这个游戏的目标是把棋子水平或者竖直地滑动到空格中，直到棋盘局面和目标状态一致（图 3.28）。

一个随机产生的八数码问题的平均解步数是 22 步。分支因子约为 3（当空格在棋盘正中间的时候，有四种可能的移动;而当它在四个角上的时候只有两种可能;当在四条边上的时候有三种可能）。这意味着到达深度为 22 的穷举搜索树将考虑大约 322≈ 3.1x1010 个状态。

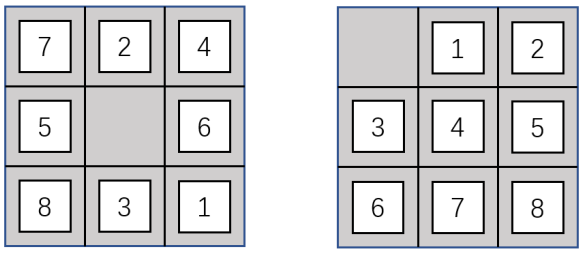


图 1.5 典型的八数码问题实例。它的解路径为 26 步

图搜索可以把这个数目削减大约 17000 倍，因为只有 9!/2=181440 个可达到的不同状态。（参见习题 3.4。）这是一个容易管理的数目，但是考虑 15 数码问题，这个数目是大约 1013，因此我们需要找到好的启发函数。如果想用 A*算法找到最短解路径，我们需要一个绝不会高估到达目标的步数的启发式函数。15 数码问题的启发式函数研究有很长的历史;这里有两个常用的：

● h1=不在位的棋子数。图 3.28 中所有的 8 个棋子都不在正确的位置，因此起始状态的 h=8。h 是一个可采纳的启发式函数，因为要把不在位的棋子都移动到正确位置上，每个错位的棋子至少要移动一次。

●h2=所有棋子到其目标位置的距离和。因为棋子不能斜着移动，计算距离指的是水平和竖直的距离

和。这有时被称为市街区距离或曼哈顿距离。h2 也是可采纳的，因为任何移动能做得最多的是把棋子向目标移近一步。图 3.28 中起始状态的棋子 1～8 得到的曼哈顿距离为

h=3＋1+2+2+2+3+3+2=18

可以看到正如我们所希望的，这两个启发式函数都没有超过实际的解代价 26。

3.6.1 启发式的精确度对性能的影响

一种刻画启发式的方法是有效分支因子 b*。对于某一问题，如果 A*算法生成的总结点数为 N，解的深度为 d，那么 b*就是深度为 d 的标准搜索树为了能够包括 N＋1 个结点所必需的分支因子。即，

$$N+1=1+b^*+(b^*)^2+\cdots+(b^*)^d$$

例如，如果 A*算法用 52 个结点在第 5 层找到了解，那么有效分支因子就是 1.92。有效分支因子可能会因问题实例发生变化，但是在难题中通常它是相当稳定的（前面我们提到过，随着解路径所在深度的增加，A*算法扩展的结点数呈指数级增长，这导致了有效分支因子的存在）。所以，在一小部分问题集合上做实验以测量出 b*的值，有益于探讨启发式的总体实用性。设计良好的启发式会使 b*的值接近于 1，以合理的计算代价对大规模的问题进行求解。

为了测试启发式函数 h 和 h，我们随机地产生了 1200 个八数码问题，解路径长度从 2 到 24 不等（每个偶数值有 100 个例子），分别用迭代加深搜索、使用h_1与h_2的 A*树搜索对这些问题求解。图 3.29 给出了每种搜索策略扩展的平均结点数和有效分支因子。结果说明h_2好于h_1，并且远好于迭代加深搜索。在长度为 12 的解上，用h_2作为启发式函数的 A*算法的效率比无信息的迭代加深搜索高 50000 倍。

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

图 3.29 ITERATIVE-DEEPENING-SEARCH 以及使用 h_1 和 h_2 的 A*算法的搜索代价和有效分支因子的比较。图中的数据是通过八数码问题实例计算的平均值，对于解路径的不同深度 d，每种深度上都选取 100 个问题实例

有人可能会问h_2是否总是比h_1好?答案是肯定的。这从两个启发式的定义很容易看出来，对于任意结点 n，$h_2(n) \geq h_1(n)$。因此称h_2比h_1占优势。优势可以直接转化为效率：使用h_2的 A*算法永远不会比使用h_1的 A*算法扩展更多的结点（除了 f(n)=C*的某些结点）。证明很简单。回忆一下 3.5 节的讨论，每个 f(n)<C*的结点都必将被扩展。还可以这样说，每个 f(n)<C*-g（n）的结点一定会被扩展。但是因为对于所有的结点，它的h_2值都至少和h_1一样大，在使用h_2的 A*搜索中被扩展的结点必定也会被使用h_1的 A*所扩展，而h_1还可能引起其他结点的扩展。所以，一般来讲使用值更大的启发式函数是好的，前提是计算该启发式花费的时间不是太多的话。

1.4.1.1 从松弛问题出发设计可采纳的启发式

我们已经看到h_1（错位棋子数）和h_2（曼哈顿距离）对于八数码问题者是相当好的启发式，而且h_2更好。那么h_2是如何被提出来的?计算机是否有能力机械地设计出这样的启发式?

h_1和h_2估算的是八数码问题中剩余路径的长度，对于该问题的简化版本它们也是相当精确的路径长度。如果游戏的规则改变为每个棋子可以随便移动，而不是只能移动到与其相邻的空位上，那么h_1将给出最短解的确切步数。类似地，如果一个棋子可以向任意方向移动一步，甚至可以移到已经被其他棋子占据的位置上，那么h_2将给出最短解的确切步数。减少了行动限制的问题称为松弛问题。松弛问题的状态空间图是

原有状态空间的超图，原因是减少限制导致图中边的增加。

由于松弛问题增加了状态空间的边，原有问题中的任一最优解同样是松弛问题的最优解;但是松弛问题可能存在更好的解，理由是增加的边可能导致捷径。所以，一个松弛问题的最优解代价是原问题的可采纳的启发式。更进一步，由于得出的启发式是松弛问题的确切代价，那么它一定遵守三角不等式，因而是一致的（参见 3.5 节）。

如果问题定义是用形式语言描述的，那么有可能来自动构造它的松弛问题'。例如，如果八数码问题的行动描述如下：

棋子可以从方格 A 移动到方格 B，如果

A 与 B 水平或竖直相邻而且 B 是空的，

我们可以去掉其中一个或者两个条件，生成三个松弛问题：

（a）棋子可以从方格 A 移动到方格 B，如果 A 和 B 相邻。

（b）棋子可以从方格 A 移动到方格 B，如果 B 是空的。

（c）棋子可以从方格 A 移动到方格 B。

由（a），我们可以得出h_2（曼哈顿距离）。原因是如果我们依次将每个棋子移入其目的位置，h_2就是相应的步数。由（b）得到的启发式将在习题 3.31 中讨论。由（c）我们可以得出h_1（不在位的棋子数），因为如果把不在位的棋子一步移到其目的地，h_1就是相应的步数。要注意的是：用这种技术生成的松弛问题本质上要能够不用搜索就可以求解，因为松弛规则使原问题分解成 8 个独立的子问题。如果松弛问题本身很难求解，使用它的值作为对应的启发式就得不偿失了。

一个名为 ABSOLVER 的程序可以从原始的问题定义出发，使用"松弛问题"技术和各种其他技术自动地生成启发式（Prieditis，1993）。ABSOLVER 为 8 数码游戏找到比以前已有的启发式都好的新启发式，并且为著名的魔方游戏找到了第一个有用的启发式。

生成新的启发式函数的难点在于经常不能找到"无疑最好的"启发式。如果可采纳启发式的集合$h_1 \cdots h_m$。对问题是有效的，并且其中没有哪个比其他的更有优势，我们应该怎样选择呢?其实我们不用选择。我们可以这样定义新的启发式从而得到其中最好的：

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\}$$

这个合成的启发式使用的是对应于问题中结点的更精确的函数。因为它的每个成员启发式都是可采纳的，所以h也是可采纳的：也很容易证明h是一致的。此外，h比所有成员启发式更有优势。

3.6.3 从子问题出发设计可采纳的启发式：模式数据库

可采纳的启发式也可以从考虑给定问题的子问题的解代价得到。例如，图 3.30 给出了图 3.28 所示的八数码问题的一个子问题。这个子问题涉及将棋子 1、2、3、4 移动到正确位置上。显然，这个子问题的最优解的代价是完整问题的解代价的下界。在某些情况下这实际上比曼哈顿距离更准确。

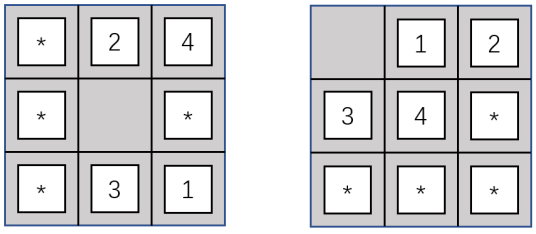


图 1.6 图 1.5 所示八数码问题的一个子问题。任务是将棋子 1、2、3 和 4 移到正确位置上，而不考虑其他棋子的情况

模式数据库（patterm databases）的思想就是对每个可能的子问题实例存储解代价———在我们的例子中，就是 4 个棋子和一个空位组成的可能状态。（其他 4 个棋子的位置与解决这个子问题是无关的，但是移动那四个棋子的代价也要算在总代价里。）接着，对搜索中遇到的每个完备状态计算其可采纳的启发式h_{DB}，计算通过在数据库里查找出相应的子问题进行。数据库本身的构造是通过从目标状态向后 1 搜索并记录下

每个遇到的新模式的代价完成的;搜索的开销分摊到许多子问题实例上。

1-2-3-4 的选择是随机的;同样可以构造 5-6-7-8 或者 2-4-6-8 等的数据库。每个数据库都能产生一个可采纳的启发式，这些启发式可以像前面所讲的那样取最大值的方式组合使用。这种组合的启发式比曼哈顿距离要精确;求解随机的 15 数码问题时所生成的结点数要少 1000 倍。

有人可能会想，1-2-3-4 数据库和 5-6-7-8 数据库的子问题看起来没有重叠，从它们得到的启发式是否可以相加?相加得到的启发式是否还是可采纳的?答案是否定的，因为对于一给定状态，1-2-3-4 子问题的解和 5-6-7-8 子问题的解可能有一些重复的移动———不移动 5-6-7-8，1-2-3-4 也不可能移入正确位置，反之亦然。不过如果我们不计入这些移动又会怎样?就是说，我们记录的不是求解 1-2-3-4 子问题的总代价值，而只是涉及 1-2-3-4 的移动次数。这样很容易得出，两个子问题的代价之和仍然是求解整个问题的代价的下界。这就是不相交的模式数据库的思想。用这样的数据库，我们可以在几毫秒内解决一个随机的 15 数码问题——与使用曼哈顿距离启发式相比生成的结点数减少了 10000 倍。对于 24 数码问题减少的结点数以百万倍计。

无交集的模式数据库在滑动棋子问题上相当可行，因为在问题可以分隔，使得每次移动只影响其中的一个子问题———因为一次只移动一个棋子。对于魔方这样的问题，这种划分相当困难，因为每步移动都会影响到 26 个立方体中的 8 块或 9 块。目前已经提出了更一般的可相加的可采纳启发式应该用魔方问题中（Yang 等，2008），但是还没有证明这种后发式要好于最好的不相加的启发式。

1.4.1.2 从经验中学习启发式

启发函数$h(n)$用来估计从结点 n 开始的解代价。Agent 怎样才能构造这样的函数?上节我们讨论了一个方案——即找出一些很容易找到最优解的松弛问题。另一个方案则是从经验里学习。"经验"在这里意味着求解大量的八数码问题。每个八数码问题的最优解都成为可供$h(n)$学习的实例。每个实例都包括解路径上的一个状态和从这个状态到达解的代价。从这些例子中，一个学习算法可以用来构造$h(n)$，（够幸运的话）它能预测搜索过程中所出现的其他状态的解代价。使用神经网络、决策树还有其他一些方法的学习技术，将在第 18 章中介绍（同样可以使用第 21 章中描述的强化学习方法）。

如果在状态描述外还能刻画给定状态的特征，归纳学习方法则是最可行的。例如，特征"不在位的棋子数"对于估算从一个状态到目标状态的真实距离可能是有用的。我们把这个特征记为$x_1(n)$。选取 100 个随机产生的八数码问题，统计它们实际的解代价。我们会发现当$x_1(n)$是 5 的时候，平均解代价约为 14，等等。有了这些数据，就可以用 x 的值来预测加$h(n)$。当然，我们还可以使用多个特征。第二个特征$x_2(n)$可以是"现在相邻但在目标状态中不相邻的棋子对数"。如何将$x_1(n)$和$x_2(n)$结合起来预测$h(n)$?通常的方法是使用线性 组合;

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

常数c_1和c_2可以调整以符合解代价的实际数据。人们希望c_1和c_2都是正数，原因是错位棋子数和不正确的相邻对使问题求解变得更困难。要注意的是这个启发式确实满足目标状态 $h(n) = 0$的条件，但不能保证可采纳或是一致性。

1.4.2 A*搜索

最佳优先搜索的最广为人知的形式称为 A*搜索（可以读为"A 星搜索"）。它对结点的评估结合了 g（n），即到达此结点已经花费的代价，和 h（n），从该结点到目标结点所花代价：

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

由于 g（n）是从开始结点到结点 n 的路径代价，而 h（n）是从结点 n 到目标结点的最小代价路径的估计值，因此

f(n）=经过结点 n 的最小代价解的估计代价

这样，如果我们想要找到最小代价的解，首先扩展 g（n）＋ h（n）值最小的结点是合理的。可以发现这个策略不仅仅合理：假设启发式函数 h（n）满足特定的条件，A*搜索既是完备的也是最优的。算法与一致代价搜索类似，除了 A*使用 g＋h 而不是 g。

保证最优性的条件：可采纳性和一致性

保障最优性的第一个条件是 h（n）是一个可采纳启发式。可采纳启发式是指它从不会过高估计到达目标的代价。因为 g（n）是当前路径到达结点 n 的实际代价，而 f（n）=g（n）＋h（n），我们可以得到直接结论：f（n）永远不会超过经过结点 n 的解的实际代价。

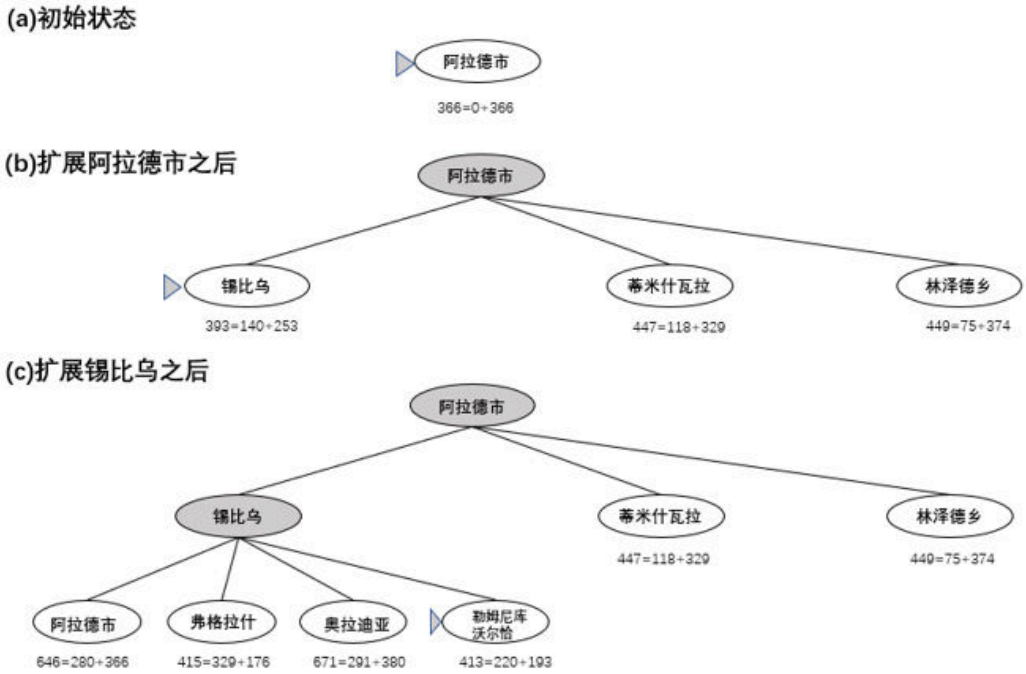
可采纳的启发式自然是乐观的，因为它们认为解决问题所花代价比实际代价小。可采纳启发式的明显例子就是用来寻找到达 Bucharest 的路径的直线距离 hSLD。直线距离是可采纳的启发式，因为两点之间直线最短，所以用直线距离肯定不会高估。图 3.24 给出了通过 A*树搜索求解到达 Bucharest 的过程。g 值从图 3.2 给出的单步代价计算得到，hsp 值在图 3.22 中给出。特别要注意的是，Bucharest 首次在步骤（e）的边缘结点集里出现，但是并没有被选中扩展，因为它的 f 值（450）比 Pitesti 的 f 值（417）高。换个说法就是可能有一个经过 Pitesti 的解的代价低至 417，所以算法将不会满足于代价为 450 的解。

第二个条件，略强于第一个的条件被称为一致性（有时也称为单调性），只作用于在图搜索中使用 A*算法'。我们称启发式 h（n）是一致的，如果对于每个结点 n 和通过任一行动 a 生成的 n 的每个后继结点 n’，从结点 n 到达目标的估计代价不大于从 n 到 n'的单步代价与从 n’到达目标的估计代价之和：

$h(n) \leq c(n,a,n') + h(n')$

这是一般的三角不等式，它保证了三角形中任何一条边的长度不大于另两条边之和。这里，三角形是由 n，n'和离 n 最近的目标结点 Gn 构成的。对于可采纳的启发式，这种不等式有明确意义：如果从 n 经过 n'到 Gn 比 h（n）代价小，就违反了 h(n）的性质：它是到达 Gn 的下界。

很容易证明（习题 3.29）一致的启发式都是可采纳的。虽然一致性的要求比可采纳性更严格，要找到满足可采纳性的但可能不一致的启发式仍然需要艰苦的工作。本章中我们讨论的可采纳的启发式都是一致的。例如，考虑 hSLD。我们知道当每边都用直线距离来度量时是满足一般的三角形不等式的，而且 n 和 n'之间的直线距离不超过 c（n，a,n'）。因此，hSLD 是一致的启发式。



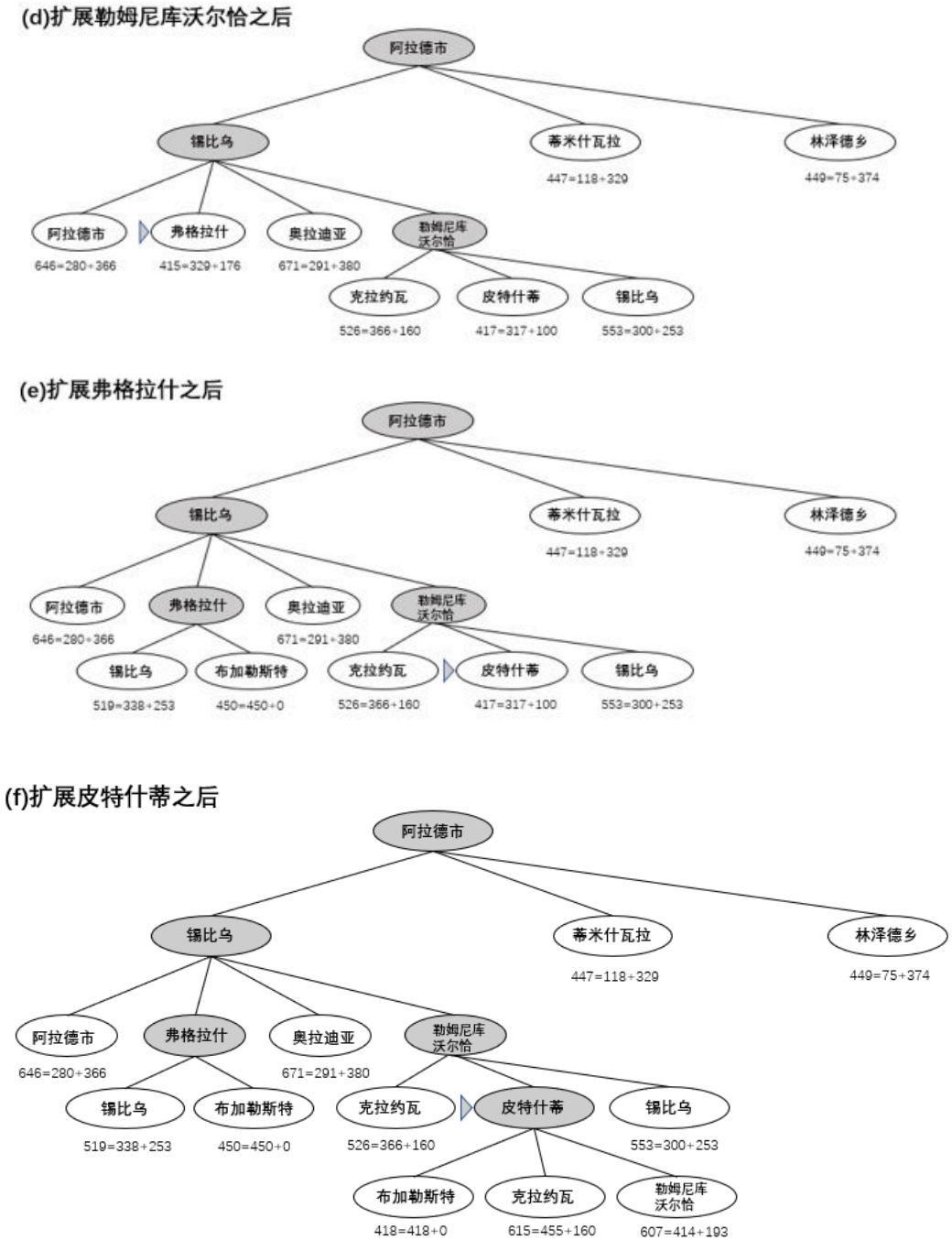


图 1.7 使用 A*搜索求解罗马尼亚问题。结点都用$f = g + h$标明。H 值是图给出的到布加勒斯特的直线距离

1.4.2.1 A*算法的最优性

我们前面提到过，A*有如下性质：如果 h（n）是可采纳的，那么 A*的树搜索版本是最优的;如果 h（n）是一致的，那么图搜索的 A*算法是最优的。

我们讨论上述声明中的后半部分，因为这更有用。一致代价搜索中参数 g 被替换成 f——就像是 A*算法自身。

第一步是证明如下性质：如果 h（n）是一致的，那么沿着任何路径的 f(n）值是非递减的。证明可从一致性的定义直接得到。假设 n'是结点 n 的后继;那么对于某行动 a，有 g（n'）=g（n）+c（n，a.n'），可得到

f（n'）=g（n'）+h（n）=g（n）+c（n，a,n'）+h（n'）≥g（n）+h（n）=f（n）

下一步则需要证明：若 A*选择扩展结点 n 时，就已经找到到达结点 n 的最优路径。否则，在到达结点 n 的最优路径上就会存在另一边缘结点 n'，这可由图 3.9 的图分离性质得到;因为 f 在任何路径上都是非递减的，n'的 f 代价比 n 小，会先被选择。

从上面两个观察可以看出，GRAPH-SEARCH 的 A"算法以 f(n）值的非递减序扩展结点。由于 f 是目标结点的实际代价（目标结点的 h=0），因此，第一个被选择扩展的目标结点一定是最优解，之后扩展的目标结点代价都不会低于它。

f 代价沿着任何路径都是非递减的事实也意味着我们可以在状态空间上绘制等值线。图 3.25 给出了实例。在 400 的等值线内，所有结点的 f(n）值都小于等于 400，其他依此类推。那么，由于 A*算法扩展的是 f 值最小的边缘结点，可以看到 A*搜索由起始结点发散，以 f 值增长同心带状的方式添加结点。

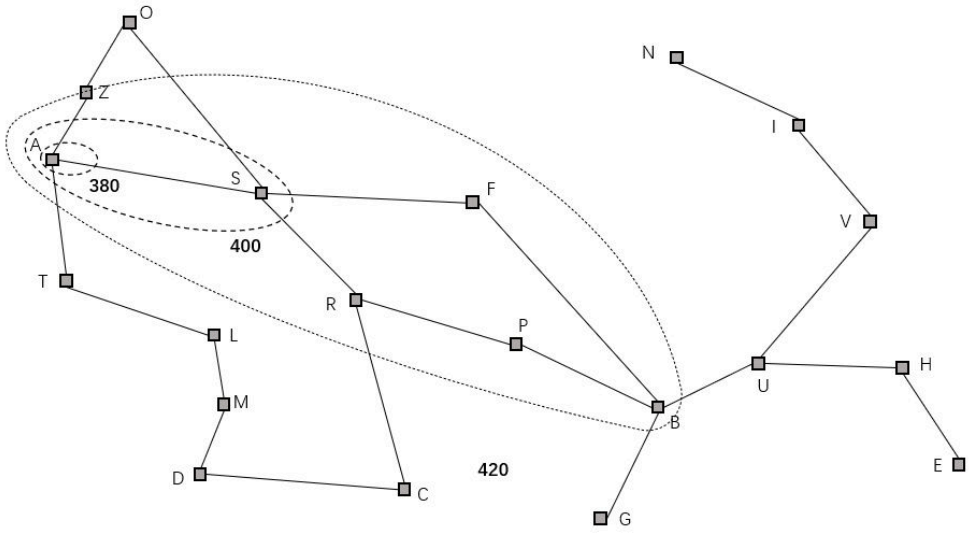


图 3.25　罗马尼亚地图的等值线 *f*=380，*f*=400 和 *f*=420，以阿拉德为初始状态。在给定的等值线内的结点的 *f* 值小于等于线值

对于一致代价搜索（A*搜索中令 h(n）=0），同心带是以起始状态为圆心的"圆"。如果使用更精确的启发式，同心带将向目标结点方向拉伸，并且在最优解路径的周围收敛变窄。如果 C*是最优解路径的代价值，可以得到：

- A*算法扩展所有 f（n）<C*的结点。
- A*算法在扩展目标结点前可能会扩展一些正好处于"目标等值线"（f(n）=C*）上的结点。

完备性要求代价小于等于 C*的结点是有穷的，前提条件是每步代价都超过ε并且 b 是有穷的。

要注意的是 A*算法不会扩展 f（n）>C*的结点——如图 3.24 中，Timisoara 尽管是根结点的子结点，并没有被扩展。可以说 Timisoara 的子树被剪枝了;因为 hSLD 是可采纳的，搜索算法可以在忽略这棵子树的同时确保最优性。剪枝——无需检验就直接把它们从考虑中排除——在 AI 的很多领域中都是很重要的。

最后一个观察到的事实是，在这类最优算法中——从根结点开始扩展搜索解路径的算法——A*算法对于任何给定的一致的启发式函数都是效率最优的。就是说，没有其他的最优算法能保证扩展的结点少于 A*算法（除了在 f(n）=C*的结点上做文章）。这是因为如果算法不扩展所有 f（n）<C*的结点，那么就很有可能会漏掉最优解。

令人满意的是，A*搜索在所有此类算法中是完备的、最优的也是效率最优的。然而，这并不意味着 A"算法是我们所需要的答案。难点在于，对于相当多的问题而言，在搜索空间中处于目标等值线内的结点数量仍然以解路径的长度呈指数级增长。对这个结论的分析超出了本书的范围，但仍有如下基本结论。对于那些每步骤代价为常量的问题，时间复杂度的增长是最优解所在深度 d 的函数，这可以通过启发式的绝对

错误和相对错误来分析。绝对误差定义为Δ =h*-h，其中 h*是从根结点到目标结点的实际代价，相对误差定义为 e=(h*-h)/h*。

复杂度的结论严重依赖于对状态空间所做的假设。最简单的模型是只有一个目标状态的状态空间，本质上是树及行动是可逆的。（八数码问题满足第一、第三个假设。）在这种情况下，A*的时间复杂度在最大绝对误差下是指数级的，为 O（bΔ）。考虑每步骤代价均为常量，我们可以把这记为$O(b^{\varepsilon d})$，其中 d 是解所在深度。考虑绝大多数实用的启发式，绝对误差至少是路径代价 h*的一部分，所以ε是常量或者递增的并且时间复杂度随 d 呈指数级增长。我们还可以看到更精确的启发式的作用：$O(b^{\varepsilon d}) = O((b^{\varepsilon})^{d})$，所以有效的分支因子（下节会给出形式化定义）为$b^{\varepsilon}$。

如果状态空间中包含多个目标状态——特别是接近最佳目标状态时——搜索过程可能会误入歧途，带来的额外代价是目标状态的数目的一部分。最后，考虑图搜索，情况会更坏。即使绝对误差受限于常量，满足 f(n）<C*的结点也是指数级的。例如，吸尘器世界中 Agent 可以以单位代价打扫任一方格却不用访问它：在这样的情况下，方格可以以任何顺序打扫。如果开始时有 N 个脏的方格，则会有 2N 个状态，其中一些子集已被打扫并且这些都在最优解路径上———所以满足 f(n)<C*———尽管启发式的误差是 1。

A*的复杂度使得坚持找到最优解的做法变得不实用。可以使用 A*算法的各种变型快速地找到局部最优解，或者有时可以设计更精确却不是严格满足可采纳性的启发式。无论任何情况下，与无信息搜索相比，使用好的启发式可以节省大量的时间和空间。我们将在第 3.6 节讨论如何设计好的启发式。

然而计算时间还不是 A*算法的主要缺点。因为它在内存中保留了所有已生成的结点（跟算法 GRAPH-SEARCH 一样），A*算法常常在计算完之前就耗尽了它的内存。因此，A*算法对于很多大规模问题，A*算法并不实用。确实有算法通过花费一些执行时间来克服内存问题，同时又不牺牲最优性和完备性。我们将在以后讨论。

1.4.3 简单实战

http://ddrv.cn/a/86090

迷宫寻路问题是人工智能中的有趣问题，如何表示状态空间和搜索路径是寻路问题的重点，本文的主要内容是 A*搜索算法的理解和应用，首先对基本知识和算法思想进行了解，再通过其对迷宫问题求解应用，编写 Python 程序进行深入学习。

我们假设某个人要从 Start 点到达 End 点，存在墙壁把这两个点层层隔开。我们把这一块搜索区域分成了一个一个的方格，使搜索区域简单化，这正是寻找路径的第一步。这种方法将我们的搜索区域简化成了一个普通的二维数组。数组中的每一个元素表示对应的一个方格，该方格的状态被标记为可通过的和不可通过的。通过找出从 Start 点到 End 点所经过的方格，就能得到 Start->End 的路径。

2. Open 和 Closed 表

创建了一个简单的搜索区域后，A*算法有两个重要的数据列表：一个记录下所有被考虑来寻找最短路径的方格（称为 open 列表）和一个记录下不会再被考虑的方格（称为 closed 列表）。

首先在 closed 列表中添加起点位置，然后把所有与它当前位置相邻的可通行小方格添加到 open 列表中。在 A*算法中，我们从起点开始，依次检查它的相邻方格，选取相邻方格然后继续向外扩展直到找到目的地。但是该选哪一个方格呢？我们需要一个评价值。

3. 路径评价

设置路径上的每个方格对应一个 评价值 F = G + H：

G 是从起点沿着已生成的路径到一个给定方格的移动开销，从起点开始到相邻方格的移动量为 1，该值会随着离始点越来越远而增大。

H 是从当前方格到终点的移动估算值，被称为视探，因为我们并不能确定剩余移动开销是多少，它仅仅是一个估算值。移动量估算值离真实值越接近，最终的路径会更加精确。如果估计值停止作用，很可能生成的路径不会是最优的。

4. 算法流程

重复以下步骤，直到遍历到终点 End，找到最短路径：

选取当前 open 列表中评价值 F 最小的节点，将这个节点称为 S；

将 S 从 open 列表移除，然后添加 S 到 closed 列表中；

对于与 S 相邻的每一块可通行的方格节点 T：如果 T 在 closed 列表中，忽略；如果 T 不在 open 列表中，添加它然后计算出它的评价值 F；如果 T 已经在 open 列表中，当我们从 S 到达 T 时，检查是否能得到 更小的 F 值，如果是，更新它的 F 值和它的前继(parent = S)。

在此迷宫问题中，将使用 “曼哈顿距离”（也叫 “曼哈顿长” 或者 “城市街区距离” ）作为 H 值，计算出距离终点水平和垂直方向上的方格数量和，忽略障碍物。这里 H 作为一种搜索的启发信息，搜索过程为广度优先搜索+贪心策略。当 End 被加入到 open 列表作为待检验节点时，表示路径被找到，此时应终止循环；或者当 open 列表为空，表明已无可以添加的新节点，而已检验过的节点中没有终点节点，则意味着无路径可达终点，此时也终止循环。从终点开始沿父节点回溯到起点，记录整个遍历中得到的节点坐标，便得到了最优路径。

```
    # 查找路径的入口函数
        def find_path(self):
            # 构建开始节点
            p = Node_Elem(None, self.s_x, self.s_y, 0.0)
            while True:
                # 扩展节点
                self.extend_round(p)
                # 如果 open 表为空，则不存在路径，返回
                if not self.open:
                    return
                # 取 F 值最小的节点
                idx, p = self.get_best()
                # 到达终点，生成路径，返回
                if self.is_target(p):
                    self.make_path(p)
                    return
                # 把此节点加入 close 表，并从 open 表里删除
                self.close.append(p)
                del self.open[idx]
```

选取 F 值最小的节点作为扩展节点，其中 G 是实际移动量，H 是估计剩余移动量，我们取用曼哈顿距离：

```
    # 求距离
        def get_dist(self, i):
            # F = G + H
            return i.dist + math.sqrt((self.e_x-i.x)*(self.e_x-i.x))+ math.sqrt((self.e_y-i.y)*(self.e_y-i.y))
```

当我们遍历到终点节点的时候，对路径进行回溯，就能得到最优路径：

```
    # 生成路径
        def make_path(self, p):
            # 从终点回溯到起点，起点的 parent 为 None
            while p:
                self.path.append((p.x, p.y))
                p = p.parent
```

1.5 群智能算法：遗传算法、粒子群算法

1.5.1 遗传算法

1.5.1.1 基本原理

遗传算法（genetic algorithm，或 GA）是随机束搜索的一个变形，它通过把两个父状态结合来生成后继，而不是通过修改单一状态进行。这和随机剪枝搜索一样，与自然选择类似，除了我们现在处理的是有性繁殖而不是无性繁殖。

像束搜索一样，遗传算法也是从 k 个随机生成的状态开始，我们称之为种群。每个状态，或称个体，用一个有限长度的字符串表示————通常是 0、1 串。例如，八皇后问题的状态必须指明 8 个皇后的位置，每列有 8 个方格，所以需要 8×log28=24 比特来表示。换句话说，每个状态可以由 8 个数字表示，数字范围都是从 1 到 8（后面我们会看到这两种不同的编码形式表现是有差异的）。图 4.6（a）显示了 4 个表示 8 皇后状态的 8 位数字串组成的种群。

图 4.6（b）～4.6（e）显示了产生下一代状态的过程。在（b）中，每个状态都由它的目标函数或（用遗传算法术语）适应度函数给出评估值。对于好的状态，适应度函数应返回较高的值，所以在八皇后问题中，我们用不相互攻击的皇后对的数目来表示，最优解的适应度是 28。这四个状态的适应度分别是 24、23、20 和 11。在这个特定的遗传算法实现中，被选择进行繁殖的概率直接与个体的适应度成正比，其百分比标在旁边。

在图 4.6（c）中，按照（b）中的概率随机地选择两对进行繁殖。请注意其中一个个体被选中两次而有一个一次也没被选中。对于要配对的每对个体，在字符串中随机选择一个位置作为杂交点。图 4.6 中的杂交点在第一对的第三位数字之后和第二对的第五位数字之后。

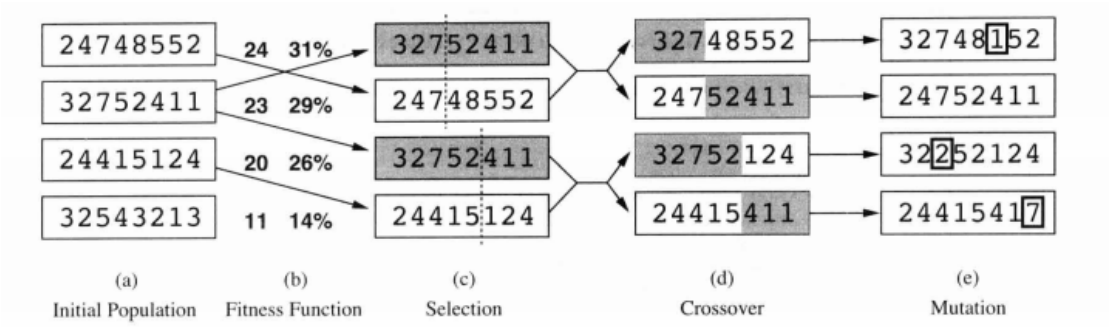


图 4.6　遗传算法，以八皇后问题为例

（a）是初始种群，（b）是适应度函数，（c）是配对结果。（d）是杂交产生的后代，（e）是变异的结果

在图 4.6（d）中，父串在杂交点上进行杂交而创造出后代。例如，第一对的第一个后代从第一个父串那里得到了前三位数字、从第二个父串那里得到了后五位数字，而第二个后代从第二个父串那里得到了前三位数字从第一个父串那里得到了后五位数字。这次繁殖过程中涉及的 8 皇后状态如图 4.7 所示。这个例子表明，如果两个父串差异很大，那么杂交产生的状态和每个父状态都相差很远。通常的情况是早期的种群是多样化的，因此杂交（类似于模拟退火）在搜索过程的早期阶段在状态空间中采用较大的步调，而在后来当大多数个体都很相似的时候采用较小的步调。

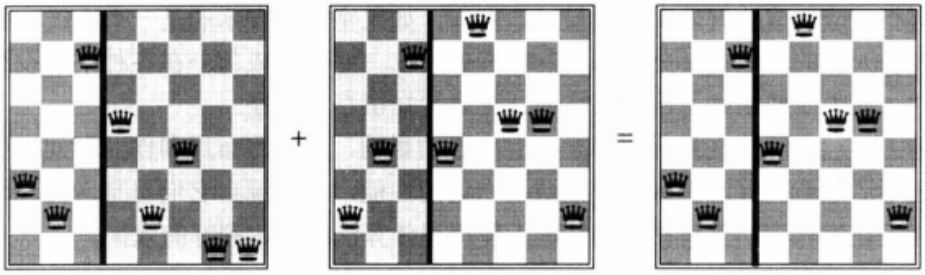


图 4.7　与图 4.6（c）中的前两个父串相对应的八皇后问题状态和与图 4.6（d）中的第一个后代相对应的状态。阴影部分的列在杂交过程中丢失，非阴影部分的列保留下来

最后，在图 4.6（e）中每个位置都会按照某个小的独立概率随机变异。在第 1、第 3 和第 4 个后代中都有一个数字发生了变异。在八皇后问题中，这相当于随机地选取一个皇后并把它随机地放到该列的某一个方格里。图 4.8 描述了实现所有这些步骤的算法。

像随机束搜索一样，遗传算法结合了上山趋势、随机探索和在并行搜索线程之间交换信息。遗传算法最主要的优点，如果算是，来自于杂交操作。然而可以在数学上证明，如果基因编码的位置在初始的时候就允许随机转换，杂交就没有优势了。直观上说，杂交的优势在于它能够将独立发展出来的能执行有用功能的字符区域结合起来，因此提高了搜索的粒度。例如，将前三个皇后分别放在位置 2、4 和 6（互不攻击）就组成了一个有用的区域，它可以和其他有用的区域组合起来形成问题的解。

遗传算法理论用模式（schema）思想来解释运转过程，模式是指其中某些位未确定的子串。例如，模式 246***描述了所有前三个皇后的位置分别是 2、4、6 的状态。能匹配这个模式的字符串（例如 24613578）称作该模式的实例。可以证明，如果某模式实例的平均适应度超过均值，那么种群内这个模式的实例数量就会随时间增长。显然，如果邻近位互不相关，效果就没有那么显著，因为只有很少的邻接区域能受益。遗传算法在模式具备真正与解相对应的成分时才工作得最好。例如，如果字符串表示的是一个天线，那么模式就应该表示天线的各组成部分，如反射器和偏转仪。好的组成部分在各种不同设计下可能都是好的。这说明要用好遗传算法需要认真对待知识表示工程。

实际上，遗传算法在最优化问题上有广泛的影响，如电路布局和作业车间调度问题。目前，还不清楚遗传算法的吸引力是源自它们的性能，还是源自它们出身进化理论。很多研究工作正在进行中，分析在什么情况下使用遗传算法能够达到好的效果。

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

图 4.8　遗传算法。算法和图 4.6 中算法一样，除了一点不同：更流行的做法是，两个父串的每次配对只产生一个后代而不是两个

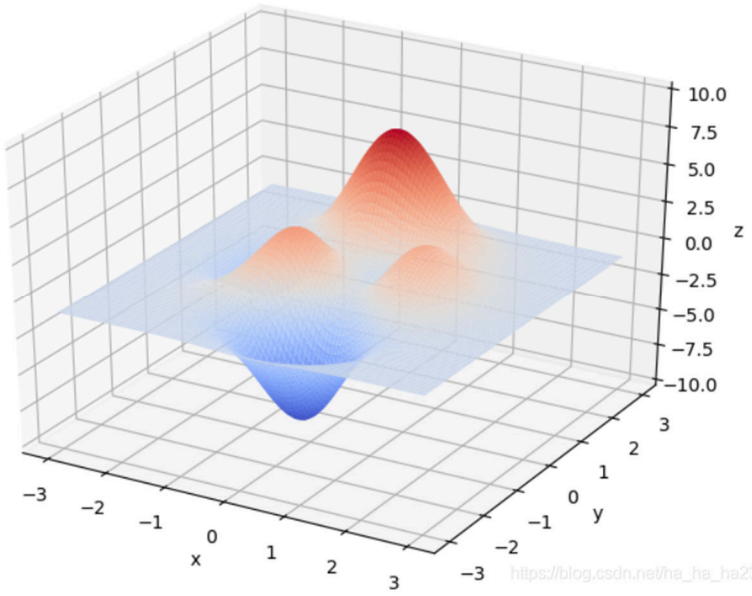
1.5.1.2 简单实战

https://blog.csdn.net/ha_ha_ha233/article/details/91364937

遗传算法是用来解决最优化问题的，下面我将以求二元函数：

```
def F(x, y):
    return 3*(1-x)**2*np.exp(-(x**2)-(y+1)**2)- 10*(x/5 - x**3 - y**5)*np.exp(-x**2-y**2)-
1/3**np.exp(-(x+1)**2 - y**2)
```

在 x ∈ [− 3 , 3] , y ∈ [− 3 , 3] x\in[-3, 3], y\in[-3, 3]x∈[−3,3],y∈[−3,3]范围里的最大值为例子来详细讲解遗传算法的每一步。该函数的图像如下图：



种群和个体的概念：

遗传算法启发自进化理论，而我们知道进化是由种群为单位的，种群是什么呢？维基百科上解释为：在生物学上，是在一定空间范围内同时生活着的同种生物的全部个体。显然要想理解种群的概念，又先得

理解个体的概念，在遗传算法里，个体通常为某个问题的一个解，并且该解在计算机中被编码为一个向量表示！ 我们的例子中要求最大值，所以该问题的解为一组可能的(x , y) (x, y)(x,y)的取值。比如(x = 2.1 , y = 0.8) , (x = − 1.5 , y = 2.3) . . . (x=2.1,y=0.8), (x=-1.5, y=2.3)...(x=2.1,y=0.8),(x=−1.5,y=2.3)...就是求最大值问题的一个可能解，也就是遗传算法里的个体，把这样的一组一组的可能解的集合就叫做种群 ，比如在这个问题中设置 100 个这样的 x , y x,yx,y 的可能的取值对，这 100 个个体就构成了种群。

编码、解码与染色体的概念：

在上面个体概念里提到个体（也就是一组可能解）在计算机程序中被编码为一个向量表示，而在我们这个问题中，个体是 x , y x,yx,y 的取值，是两个实数，所以问题就可以转化为如何将实数编码为一个向量表示，可能有些朋友有疑惑，实数在计算机里不是可以直接存储吗，为什么需要编码呢？这里编码是为了后续操作（交叉和变异）的方便。

生物的 DNA 有四种碱基对，分别是 ACGT，DNA 的编码可以看作是 DNA 上碱基对的不同排列，不同的排列使得基因的表现出来的性状也不同（如单眼皮双眼皮）。在计算机中，我们可以模仿这种编码，但是碱基对的种类只有两种，分别是 0，1。只要我们能够将不同的实数表示成不同的 0，1 二进制串表示就完成了编码，也就是说其实我们并不需要去了解一个实数对应的二进制具体是多少，我们只需要保证有一个映射。

$$y = f(x), x\ is\ decimal\ system, y\ is\ binary\ system$$

能够将十进制的数编码为二进制即可，至于这个映射是什么，其实可以不必关心。将个体（可能解）编码后的二进制串叫做染色体，染色体（或者有人叫 DNA）就是个体（可能解）的二进制编码表示。为什么可以不必关心映射 f (x) f(x)f(x)呢？因为其实我们在程序中操纵的都是二进制串，而二进制串生成时可以随机生成，如：

```
    #pop 表示种群矩阵，一行表示一个二进制编码表示的 DNA，矩阵的行数为种群数
目,DNA_SIZE 为编码长度
    pop = np.random.randint(2, size=(POP_SIZE, DNA_SIZE*2)) #matrix (POP_SIZE, DNA_SIZE*2)
```

实际上是没有需求需要将一个十进制数转化为一个二进制数，而在最后我们肯定要将编码后的二进制串转换为我们理解的十进制串，所以我们需要的是 y = f (x) y=f(x)y=f(x)的逆映射，也就是将二进制转化为十进制，这个过程叫做解码（很重要，感觉初学者不容易理解），理解了解码编码还难吗?先看具体的解码过程如下。

首先我们限制二进制串的长度为 10（长度自己指定即可，越长精度越高），例如我们有一个二进制串（在程序中用数组存储即可）$[0,1,0,1,1,1,0,1,0,1]$，这个二进制串如何转化为实数呢？不要忘记我们的$x,y\in[-3,3]$这个限制，我们目标是求一个逆映射将这个二进制串映射到$x,y\in[-3,3]$即可，为了更一般化我们将 x,y 的取值范围用一个变量表示，在程序中可以用 python 语言写到：

```
    X_BOUND = [-3, 3] #x 取值范围
    Y_BOUND = [-3, 3] #y 取值范围
```

为将二进制串映射到指定范围，首先先将二进制串按权展开，将二进制数转化为十进制数$0\times2^9+1\times2^8+0\times2^7+\cdots+0\times2^1+1\times2^0=373$,，然后将转换后的实数压缩到[0 , 1]之间的一个小数，$373\div(2^{10}-1)\approx0.36461388074$通过以上这些步骤所有二进制串表示都可以转换为[0 , 1] [0,1][0,1]之间的小数，现在只需要将[0 , 1]区间内的数映射到我们要的区间即可。假设区间[0 , 1]内的数称为 num，转换在 python 语言中可以写成：

```
    #X_BOUND,Y_BOUND 是 x，y 的取值范围  X_BOUND = [-3, 3], Y_BOUND = [-3, 3],
    x_ = num * (X_BOUND[1] - X_BOUND[0]) + X_BOUND[0] #映射为 x 范围内的数
```

通过以上这些步骤我们完成了将一个二进制串映射到指定范围内的任务（解码）。

现在再来看看编码过程。不难看出上面我们的 DNA（二进制串）长为 10，10 位二进制可以表示2^{10}种不同的状态，可以看成是将最后要转化为的十进制区间 $x,y\in[-3,3]$（下面讨论都时转化到这个区间）切分成2^{10} 份，显而易见，如果我们增加二进制串的长度，那么我们对区间的切分可以更加精细，转化后的十进制解也更加精确。例如，十位二进制全 1 按权展开为 1023，最后映射到[-3, 3]区间时为 3，而 1111111110（前面 9 个 1）按权展开为 1022，$1022/(2^{10}-1)\approx 0.999022$，$0.999022*(3-(-3))+(-3)\approx 2.994134$；如果我们将实数编码为 12 位二进制，111111111111（12 个 1）最后转化为 3，而 111111111110（前面 11 个 1）按权展开为 4094，$4094/(2^{12}-1=4095)\approx 0.999756$，$0.999755*(3-(-3))+(-3)\approx 2.998534$,可以看出用 10 位二进制编码划分区间后，每个二进制状态改变对应的实数大约改变 0.005866，而用 12 位二进制编码这个数字下降到 0.001466，所以 DNA 长度越长，解码为 10 进制的实数越精确。

以下为解码过程的 python 代码：

这里我设置 DNA_SIZE=24（一个实数 DNA 长度），两个实数 x , y x,yx,y 一共用 48 位二进制编码，我同时将 x 和 y 编码到同一个 48 位的二进制串里，每一个变量用 24 位表示，奇数 24 列为 x 的编码表示，偶数 24 列为 y 的编码表示。

```
    def translateDNA(pop):#pop 表示种群矩阵，一行表示一个二进制编码表示的 DNA，矩阵的行数为种群数目
        x_pop = pop[:,1::2]#奇数列表示 X
        y_pop = pop[:,::2] #偶数列表示 y
            #pop:(POP_SIZE,DNA_SIZE)*(DNA_SIZE,1) --> (POP_SIZE,1)完成解码
            x = x_pop.dot(2**np.arange(DNA_SIZE)[::-1])/float(2**DNA_SIZE-1)*(X_BOUND[1]-X_BOUND[0])+X_BOUND[0]
            y = y_pop.dot(2**np.arange(DNA_SIZE)[::-1])/float(2**DNA_SIZE-1)*(Y_BOUND[1]-Y_BOUND[0])+Y_BOUND[0]
            return x,yy_ = num * (Y_BOUND[1] - Y_BOUND[0]) + Y_BOUND[0] #映射为 y 范围内的数
```

适应度和选择

我们已经得到了一个种群，现在要根据适者生存规则把优秀的个体保存下来，同时淘汰掉那些不适应环境的个体。现在摆在我们面前的问题是如何评价一个个体对环境的适应度？在我们的求最大值的问题中可以直接用可能解（个体）对应的函数的函数值的大小来评估，这样可能解对应的函数值越大越有可能被保留

```
    def get_fitness(pop):
        x,y = translateDNA(pop)
     pred = F(x, y)
     return (pred - np.min(pred)) + 1e-3 #减去最小的适应度是为了防止适应度出现负数，通过这一步 fitness 的范围为[0, np.max(pred)-np.min(pred)],最后在加上一个很小的数防止出现为 0 的适应度
```

下来，以求解上面定义的函数 F 的最大值为例，python 代码如下：

pred 是将可能解带入函数 F 中得到的预测值，因为后面的选择过程需要根据个体适应度确定每个个体被保留下来的概率，而概率不能是负值，所以减去预测中的最小值把适应度值的最小区间提升到从 0 开始，但是如果适应度为 0，其对应的概率也为 0，表示该个体不可能在选择中保留下来，这不符合算法思想，遗传算法不绝对否定谁也不绝对肯定谁，所以最后加上了一个很小的正数。

有了求最大值的适应度函数求最小值适应度函数也就容易了，python 代码如下:

```
    def get_fitness(pop):
     x,y = translateDNA(pop)
     pred = F(x, y)
     return -(pred - np.max(pred)) + 1e-3
```

因为根据适者生存规则在求最小值问题上，函数值越小的可能解对应的适应度应该越大，同时适应度也不能为负值，先将适应度减去最大预测值，将适应度可能取值区间压缩为$[np.min(pred) - np.max(pred),0]$，然后添加个负号将适应度变为正数，同理为了不出现 0，最后在加上一个很小的正数。

有了评估的适应度函数，下面可以根据适者生存法则将优秀者保留下来了。选择则是根据新个体的适应度进行，但同时不意味着完全以适应度高低为导向（选择 top k 个适应度最高的个体，容易陷入局部最优解），因为单纯选择适应度高的个体将可能导致算法快速收敛到局部最优解而非全局最优解，我们称之为早熟。作为折中，遗传算法依据原则：适应度越高，被选择的机会越高，而适应度低的，被选择的机会就低。 在 python 中可以写做：

```
def select(pop, fitness):       # nature selection wrt pop's fitness
    idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, replace=True,
                           p=(fitness)/(fitness.sum()) )
    return pop[idx]
```

不熟悉 numpy 的朋友可以查阅一下这个函数，主要是使用了 choice 里的最后一个参数 p，参数 p 描述了从 np.arange(POP_SIZE)里选择每一个元素的概率，概率越高约有可能被选中，最后返回被选中的个体即可。

交叉、变异

通过选择我们得到了当前看来“还不错的基因”，但是这并不是最好的基因，我们需要通过繁殖后代（包含有交叉+变异过程）来产生比当前更好的基因，但是繁殖后代并不能保证每个后代个体的基因都比上一代优秀，这时需要继续通过选择过程来让试应环境的个体保留下来，从而完成进化，不断迭代上面这个过程种群中的个体就会一步一步地进化。

具体地繁殖后代过程包括交叉和变异两步。交叉是指每一个个体是由父亲和母亲两个个体繁殖产生，子代个体的 DNA（二进制串）获得了一半父亲的 DNA，一半母亲的 DNA，但是这里的一半并不是真正的一半，这个位置叫做交配点，是随机产生的，可以是染色体的任意位置。通过交叉子代获得了一半来自父亲一半来自母亲的 DNA，但是子代自身可能发生变异，使得其 DNA 即不来自父亲，也不来自母亲，在某个位置上发生随机改变，通常就是改变 DNA 的一个二进制位（0 变到 1，或者 1 变到 0）。

需要说明的是交叉和变异不是必然发生，而是有一定概率发生。先考虑交叉，最坏情况，交叉产生的子代的 DNA 都比父代要差（这样算法有可能朝着优化的反方向进行，不收敛），如果交叉是有一定概率不发生，那么就能保证子代有一部分基因和当前这一代基因水平一样；而变异本质上是让算法跳出局部最优解，如果变异时常发生，或发生概率太大，那么算法到了最优解时还会不稳定。交叉概率，范围一般是 0.6~1，突变常数（又称为变异概率），通常是 0.1 或者更小。

python 实现如下：

```
    def crossover_and_mutation(pop, CROSSOVER_RATE = 0.8):
     new_pop = []
     for father in pop:        #遍历种群中的每一个个体，将该个体作为父亲
          child = father            #孩子先得到父亲的全部基因（这里我把一串二进制串的那些 0，1
称为基因）
          if np.random.rand() < CROSSOVER_RATE:                #产生子代时不是必然发生交叉，
而是以一定的概率发生交叉
               mother = pop[np.random.randint(POP_SIZE)]#再种群中选择另一个个体，并将该个体
作为母亲
               cross_points = np.random.randint(low=0, high=DNA_SIZE*2) #随机产生交叉的点
               child[cross_points:] = mother[cross_points:]          #孩子得到位于交叉点后的母亲的
基因
          mutation(child)   #每个后代有一定的机率发生变异
          new_pop.append(child)
```

上面这些步骤即为遗传算法的核心模块，将这些模块在主函数中迭代起来，让种群去进化.

```
    pop = np.random.randint(2, size=(POP_SIZE, DNA_SIZE*2)) #生成种群  matrix (POP_SIZE,
DNA_SIZE)
    for _ in range(N_GENERATIONS):      #种群迭代进化 N_GENERATIONS 代
        crossover_and_mutation(pop, CROSSOVER_RATE)      #种群通过交叉变异产生后代
        fitness = get_fitness(pop)     #对种群中的每个个体进行评估
        pop = select(pop, fitness)     #选择生成新的种群
    return new_pop


   def mutation(child, MUTATION_RATE=0.003):
    if np.random.rand() < MUTATION_RATE:                          #以 MUTATION_RATE 的概率进行
变异
        mutate_point = np.random.randint(0, DNA_SIZE)   #随机产生一个实数，代表要变异基因的
位置
        child[mutate_point] = child[mutate_point]^1   #将变异点的二进制为反转
```

1.5.2 粒子群算法

1.5.2.1 基本原理

粒子群算法（Particle Swarm Optimization, PSO）是一种有效的全局寻优算法，最早由美国 的 Kenedy 和 Eberhart 于 1995 年提出，设想模拟鸟群觅食的过程，后来从这种模型中得到启示， 并将粒子群算法用于解决优化问题。与其他进化算法相类似，粒子群算法也是通过个体间的 协作与竞争，实现复杂空间中最优解的搜索。粒子群优化算法具有进化计算和群智能的特点， 是基于群体智能理论的优化算法，通过群体中粒子间的合作与竞争产生的群体智能指导优化 搜索。与传统的进化算法相比，粒子群算法保留了基于种群的全局搜索策略，但是其采用 的“速度-位移”模型操作简单，避免了复杂的遗传操作，它特有的记忆使其可以动态跟踪当前 的搜索情况调整搜索策略。由于每代种群中的解具有“自我”学习提高和向“他人”学习的双重 优点，从而能在较少的迭代次数内找到最优解。该算法目前已广泛应用于函数优化、数据挖 掘、神

经网络训练等应用领域。

粒子群算法中，每一个优化问题的解看做是搜索空间中的一只鸟，即“粒子”。首先生成初 始种群，即在可行解空间中随机初始化一群粒子，每个粒子都为优化问题的一个可行解，并由 目标函数评价其适应度值。每个粒子都在解空间中运动，并由一个速度决定其飞行方向和距 离，通常粒子追随当前的最优粒子在解空间中进行搜索。在每一次迭代过程中，粒子将跟踪两 个“极值”来更新自己，一个是粒子本身找到的最优解，另一个是整个种群目前找到粒子群算法可描述为：设粒子群在一个 n 维空间中搜索，由 N 个粒子组成种群 X={X1 ，X2 ， …，XN }，其中每个粒子所处的位置 Xi ={xi1 ， xi2 ，…，xin }都表示问题的一个解。粒子通过不断 调整自己的位置 xid 来搜索新解。每个粒子都能记住自己搜索到的最优解，记做 pid ，以及整个 粒子群经历过的最好的位置，即目前搜索到的最优解，记做 pgd 。此外每个粒子都有一个速度，记做 Vi={vi1 ，vi2 ，…，vin }，当两个最优解都找到后，每个粒子根据公式（5-1）来更新自己的速 度。的最优解， 这个极值即全局最优解。

粒子群算法可描述为：设粒子群在一个 n 维空间中搜索，由 N 个粒子组成种群X = {X1 ，X2 ，…，XN }，其中每个粒子所处的位置Xi = {X_{i1} ，X_{i2} ，…，X_{in} }都表示问题的一个解。粒子通过不断 调整自己的位置 xid 来搜索新解。每个粒子都能记住自己搜索到的最优解，记做p_{id} ，以及整个 粒子群经历过的最好的位置，即目前搜索到的最优解，记做p_{gd} 。此外每个粒子都有一个速度， 记做$Vi = \{v_{i1}$ ，v_{i2} ，…，$v_{in}\}$，当两个最优解都找到后，每个粒子根据公式（5-1）来更新自己的速 度。

$$v_{id}(t+1) = \omega v_{id}(t) + \eta_1 \text{rand}()(p_{id} - x_{id}(t)) + \eta_2 \text{rand}()(p_{gd} - x_{id}(t)) \qquad (5-1)$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1) \qquad (5-2)$$

$$v_{id}(t+1) = \omega v_{id}(t) + \eta_1 \text{rand}()(p_{id} - x_{id}(t)) + \eta_2 \text{rand}()\left(p_{gd} - x_{id}(t)\right)$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1)$$

式中，$v_{id}(t+1)$表示第 *i* 个粒子在$t+1$次迭代中第 d 维上的速度，ω 为惯性权重，η1 、η2 为加速常数，rand()为 0～1 之间的随机数。此外，为使粒子速度不致过大，可设置速度上限V_{max} ，即当式 （5-1）中$v_{id}(t+1) > V_{max}$ 时，$v_{id}(t+1) = V_{max}$；$v_{id}(t+1) < V_{max}$ 时，$v_{id}(t+1) = V_{max}$ 。 从式（5-1）和式（5-2）可以看出，粒子的移动 方向由三部分决定：自己原有的速度$v_{id}(t)$、与自己最佳经历的距离$p_{id} - x_{id}(t)$和与群体最佳经历的 距离$p_{gd} - x_{id}(t)$，并分别由权重系数 ω、η1 、η2 决定 其相对重要性。图 5-1 所示为 3 种移动方向的加权 求和示意图。

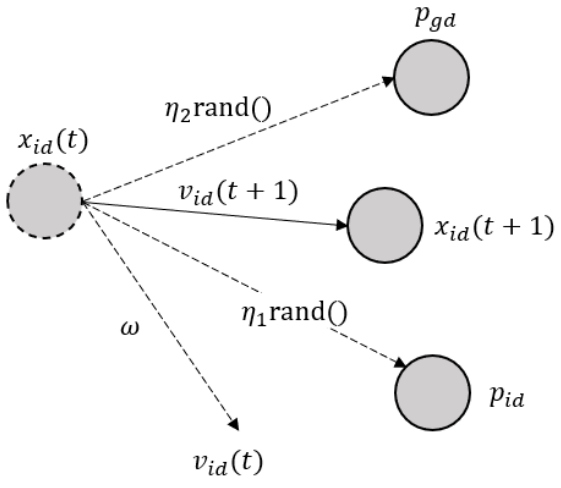


图 1.8　3 种类移动方向的加权求和示意图

1.5.2.2 术语介绍

（1）粒子 在任何一种组合优化问题中，问题的解都是以一定的形式给出的。在粒子群算法中，粒子即代表优化问题的可行解。所有的操作都是在粒子的基础上进行的。

（2）群体 一定数量的个体组合在一起构成了一个群体，粒子是群体的基本单位。

（3）群体规模 群体中个体数目的总和称为群体规模，又叫群体大小。

（4）适应度 个体对环境的适应程度叫适应度。适应度用来衡量每一个粒子解决问题的能力，代表个体在群体中的地位。

（5）全局模式 全局模式指每一个粒子的运动轨迹受粒子群中所有粒子的影响，粒子追随两个极值——自身极值和全局极值，通过这两个极值来不断调整粒子的位置。全局模式具有较快的收敛特 性，但鲁棒性较差。

（6）局部模式 局部模式指粒子的轨迹只受自身的认知和邻近粒子状态的影响，而不是被所有粒子的状 态影响。局部模式具有较高的鲁棒性，但是收敛速度相对较慢。

3. 基本流程 粒子群算法的具体步骤描述如下。

①初始化粒子群，即随机设定各粒子的初始位置 X 和初始速度 V。

②根据初始位置和初始速度产生各粒子的新的位置。

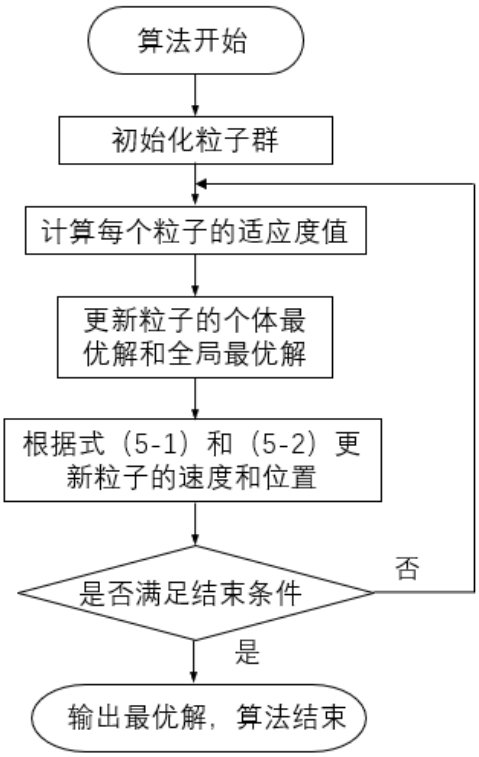
③计算每个粒子的适应度值。

④自身极值算子。 对每个粒子，比较它的适应度值和它经历过的最好位置 pid 的适应度值，并更新最好位置 pid 。

⑤全局极值算子。 对每个粒子，比较它的适应度值和群体所经历的最好位置 pgd 的适应度值，如果比 pgd 更 好，更新全局最优解 pgd 。

⑥速度-位移模型操作算子 在粒子群算法中，每个粒子都有一个速度，通过对粒子速度的改变来更新粒子的位置，根 据式（5-1）和式（5-2）调整粒子的速度和位置。

⑦如果达到结束条件（足够好的位置或最大迭代次数），则结束，否则转步骤③继续迭代。 粒子群算法的基本流程如图 5-2 所示。



1.5.2.3 粒子群算法的构成要素

粒子群算法的构成要素有自身极值算子、全局极值算子、速度-位移模型操作算子及搜索 模式。

1）自身极值算子 自身极值算子记录每个粒子到目前为止自己所搜索到的最优解，通过自身所经历的最优 解来不断地调整自身的位置，具有“自我”学习提高的能力。

2）全局极值算子 全局极值算子记录整个粒子群经历过的最好的位置，即目前搜索到的最优解，通过粒子 所经历的全局最好位置来更新自身位置，使得更新的解向着全局最优解的方向逼近，具有 向“他人”学习的优点，从而能在较少的迭代次数内找到最优解。

3）速度-位移模型操作算子 在粒子群中，每个粒子都有自己的速度，通过对自身速度的改变来更新当前的位置。速度

的改变主要是通过当前位置与自身极值的偏差和当前位置与全局极值的偏差，两个偏差都乘 以一个[0，1]的随机数加到原来的速度上实现的。虽然改变的步长是随机的，但是速度改变的 方向是向着最优解的方向逼近。位置的更新是在原来位置上加上改变的速度来实现的。使得 位置向着最优解的方位逼近。

4）搜索模式 搜索模式包括全局模式和局部模式。Kennedy 等人在对鸟群觅食的观察过程中发现，每只 鸟并不总是能看到鸟群中其他所有鸟的位置和运动方向，而往往只是看到相邻的鸟的位置和 运动方向。因此他提出了两种粒子群算法模式：全局模式（global version PSO）和局部模式 （local version PSO）。 全局模式是指每个粒子的运动轨迹受粒子群中所有粒子的状态影响，粒子追随两个极 值，自身极值 pid 和种群全局极值 pgd ，式（5-1）和式（5-2）的算法描述的就是全局算子。而在局 部模式中，粒子的轨迹只受自身的认知和邻近的粒子状态的影响，而不是被所有粒子的状态 所影响，粒子除了追随自身极值 pid 外，不追随全局极值 pgd ，而是追随邻近粒子中的局部极值 pnd。在该情况中，每个粒子需记录自己和它邻居的最优值，而不需要记录整个群体的最优值。 此时，速度更新过程可用式（5-3）表示。

$$v_{id}(t+1)=\omega v_{id}(t)+\eta_1\,\mathrm{rand}()(p_{id}-x_{id}(t))+\eta_2\,\mathrm{rand}()(p_{nd}-x_{id}(t)) \qquad (5-3)$$

$$v_{id}(t+1)=\omega v_{id}(t)+\eta_1\mathrm{rand}()(p_{id}-x_{id}(t))+\eta_2\mathrm{rand}()(p_{nd}-x_{id}(t))$$

全局模式具有较快的收敛速度，但是鲁棒性较差。相反，局部模式具有较高的鲁棒性而收敛速度相对较慢，因此在运用粒子群算法解决不同的优化问题时，应针对具体情况采用相应 的算子。本章所讨论的粒子群算法采用全局模式。

5. 控制参数选择 参数选取对算法的性能和效率有很大的影响。在粒子群算法中有 3 个重要参数：惯性权重 ω，速度调节参数 η1 、η2 。惯性权重 ω 使粒子保持运动惯性，速度调节参数 η1 、η2 表示粒子向 pid 和 pgd 位置的加速项权重。如果 ω=0，则粒子速度没有记忆性，粒子群将由当前的极值解位置决定，失去自我，会产生发散效应；如果 η1 =0，则粒子失去“认知”能力，只具有“社会”性，粒子群 收敛速度会更快，但是容易陷入局部极值；如果 η2 =0，则粒子只具有“认知”能力，而不具有“社 会”性，等价于多个粒子独立搜索，因此很难得到最优解。

实践证明没有绝对最优的参数，针对不同的问题选取合适的参数才能获得更好的收敛速度和鲁棒性，一般情况下 ω 取 0～1 之间的随机数，η1 、η2 分别取 2。

6. 粒子群算法群体智能搜索策略分析

1）个体行为及个体之间信息交互方法分析

粒子群算法和其他进化算法一样都是基于“种群”的概念，用来表示一组解空间的个体集合。它们随机初始化种群，使用适应度值来评价个体，粒子在进化过程中具有记忆功能，它能够记住自身所经历的最优位置（自身极值）和全局粒子所遍历的最优位置（全局极值），在全局 模式下通过自身极值和全局极值来不断地调整自身的位置，使得自己的位置不断地向着最优 解的方向逼近，因此所有的粒子可以更快地收敛于最优解。粒子群算法具有并行性，其搜索的 过程是从一个解集合开始的，而不是从单个个体开始，这样就有效避免了算法陷入局部极小 值，并且这种并行计算易于在并行计算机上实现，提高了算法的性能和效率。 粒子群算法中的个体指的就是粒子本身，粒子是构成粒子群算法的基础，一切操作都是 围绕着粒子

进行的。将粒子的最优位置作为求解优化问题的解，通过不断更新粒子的位置在 整个解空间中寻找最优的解。粒子的位置信息是通过其运动速度来改变的，它决定了粒子的 移动方向和距离。 在粒子群算法中，粒子个体之间的交互行为主要体现在两点：一是自身与自身的历史记录 交互，每个个体都能够利用自身的认知，根据自己所经历的最优位置调整自己，具有“自我”学 习提高的能力；二是全局记录与自身的交互，根据整个群体所遍历的最好位置的差距来更新自 己的信息，具有向“他人”学习的优点。这种找出自身与自身历史的差距和向优秀个体学习的 行为是生物进化的本能，也是个体进步的动力，这正是粒子群算法最值得借鉴之处，这种思想 可以嵌入到许多其他的智能算法之中。

2）群体进化分析

粒子群算法同其他群智能算法相同，也是模拟生物行为过程，在群体进化上有自己的特 点，并不像传统的生物进化算法那样通过选择机制使得群体的整体质量提高，而是通过找出 自身与自身历史的差距和向优秀个体学习的机制。个体之间既有差异，又有全局的引领，具有 协同搜索的特点，使得群体向着更好的方向发展。通过每一代群体的位置更新产生新的群体。 这样可以保证群体的优良性，并加快寻优速度，不断地搜寻全局的最优个体。

粒子群算法在迭代过程中，从纵向看，自身极值算子和全局极值算子两种方式历史性记 载了每个个体的最佳状态和全局所经历的最佳状态，从横向看，在每一次的迭代过程中，速 度-位移模型操作算子不考虑最佳状态出现的时机，而是根据当前状态与历史最佳状态的偏差 调整位置，下一次的迭代过程没有上一次最优解的参与，既没有充分利用在每一次迭代过程 中所产生的有用信息，也没有定量地掌握目标函数在自变量定义空间中的总体变化趋势。粒 子群算法作为一种模拟自然生物行为的随机搜索算法，有可能实现全局最优搜索，但同时也 存在出现过早收敛的情况。 搜索模式包括全局模式和局部模式。在这两种模式下的个体都具有记忆最优解的能力， 能够很好地实现最优解的寻找。全局模式充分利用所找到的全局最优解更新自己的位置，具 有协同搜索和很好的全局搜索能力，能够较快地搜索到最优解，但是其鲁棒性较差；局部模式 只是受邻近粒子状态的影响，其意图在当前最优解附近搜索，避免盲目操作，鲁棒性较好，但 收敛速度相对缓慢。在实际应用中，可以结合全局模式和局部模式来共同优化每个个体。

1.5.2.4 粒子群算法在聚类分析中的应用

1. 构造个体

设模式样品集为 X={Xi ，i=1，2，…，N}，其中 Xi 代表某一样品的总特征，为 D 维模式向 量，聚类问题就是要找到一个划分 C={C1 ，C2 ，…，Ck }，使得总的类内离散度和达到最小。

$$J_c = \sum_{j=1}^{k} \sum_{X_i \in C_j} d(X_i, C_j)$$

式中，Cj 为第 j 个聚类的中心，d（Xi ，Cj ）为样品到对应聚类中心的距离，聚类准则函数 Jc 即为各类样品到对应聚类中心距离的总和。 当聚类中心确定时，聚类的划分可由最近邻法则决定。即对样品 Xi ，若第 j 类的聚类中心 Cj 满足式（5-5），则 Xi 属于类 j。

$$d(X_i, C_j) = \min_{l=1,2,\cdots,k} d(X_i, C_l)$$

在粒子群算法求解聚类问题中，每个粒子作为一个可行解组成粒子群（即解集）。根据解 的含义不同，通常可以分为两种方法，一种是以聚类结果为解，一种是以聚类中心集合为解。 本节讨论的方法采用的是基于聚类中心集合作为粒子对应解，也就是每个粒子的位置是由 k 个 聚类中心组成的，k 为已知的聚类数目。

一个具有 k 个聚类中心，样品向量维数为 D 的聚类问题中，每个粒子 i 由三部分组成，即粒子位置、速度和适应度值。粒子结构 i 表示为：

$$\begin{aligned}\text{Particle}(i) = \{ \\ &\text{location}[\,], \\ &\text{velocity}[\,], \\ &\text{fitness} \\ \}\end{aligned} \tag{5-6}$$

$$\text{Paticle}(i).=\{$$

粒子的位置编码结构表示为：

$$\text{Paticle}(i).\text{location}[\quad] = [C_1, \ldots, C_j, \ldots, C_k]$$

式中，Cj 表示第 j 类的聚类中心，也是一个 D 维矢量。同时每个粒子还有一个速度，其编码结构为:

$$\text{Paticle}(i).\text{velocity}[\quad] = [V_1, \ldots, V_j, \ldots, V_k]$$

V_j 表示第j个聚类中心的速度值，可知V_j 也是一个 D 维矢量。

粒子适应度值 Particle.fitness 为一个实数，表示粒子的适应度。

以如图 5-3 所示的原始数据为例，介绍粒子结构。从图上可见样品分为 6 个类别。首先产生 N 个粒子，每个粒子对图 5-3 中的样品随机分类，并计算各个类中心 ，初始速度 Vi 为 0。粒子 编码如表 5-1 所示，一个个体最优解 Pid 编码如表 5-2 所示，全局最优解 Pgd 编码如表 5-3 所示。

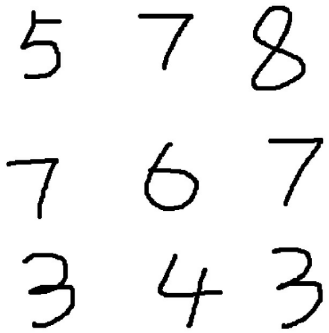


图 5-3 原始数据

表 5-1 粒子编码

类中心 w	$\overline{X^{(w_1)}}$	$\overline{X^{(w_2)}}$	$\overline{X^{(w_3)}}$	$\overline{X^{(w_4)}}$	$\overline{X^{(w_5)}}$	$\overline{X^{(w_6)}}$
每个类的中心速度 V	V_1	V_2	V_3	V_4	V_5	V_6
适应度	$\text{fitness} = \frac{1}{J} = 1/\sum_{j=1}^{M}\sum_{X_i \in w_j} d(X_i, \overline{X^{(w_j)}})$					

表 5-2 一个个体最优解 Pid 编码

粒子 i 最优解P_{id}	$\overline{X^{(w_1)}}$	$\overline{X^{(w_2)}}$	$\overline{X^{(w_3)}}$	$\overline{X^{(w_4)}}$	$\overline{X^{(w_5)}}$	$\overline{X^{(w_6)}}$

表 5-3 全局最优解 Pgd 编码

全局最优解P_{gd}	$\overline{X^{(w_1)}}$	$\overline{X^{(w_2)}}$	$\overline{X^{(w_3)}}$	$\overline{X^{(w_4)}}$	$\overline{X^{(w_5)}}$	$\overline{X^{(w_6)}}$

评价适应度

根据已定义好的粒子群结构，可以采用以下方法计算其适应度。

①按照最近邻法则式（5-5），确定该粒子的聚类划分。

②根据聚类划分，重新计算聚类中心，按照式（5-4）计算总的类内离散度 Jc 。

③粒子的适应度可表示为下式：

$$\text{Paticle.fitness} = \frac{1}{J_c}$$

式中，Jc 是总的类内离散度和，根据具体情况而定。即粒子所代表的聚类划分的总类间离散度越小，粒子的适应度越大。

此外，每个粒子在进化过程中还记忆一个个体最优解 P_id，表示该粒子经历的最优位置和 适应度值。整个粒子群存在一个全局最优解 P_gd，表示粒子群经历的最优位置和适应度值。

$$P\_id(i) = \{ \text{location}[\,], \text{fitness} \} \tag{5-10}$$

$$P\_gd = \{ \text{location}[\,], \text{fitness} \} \tag{5-11}$$

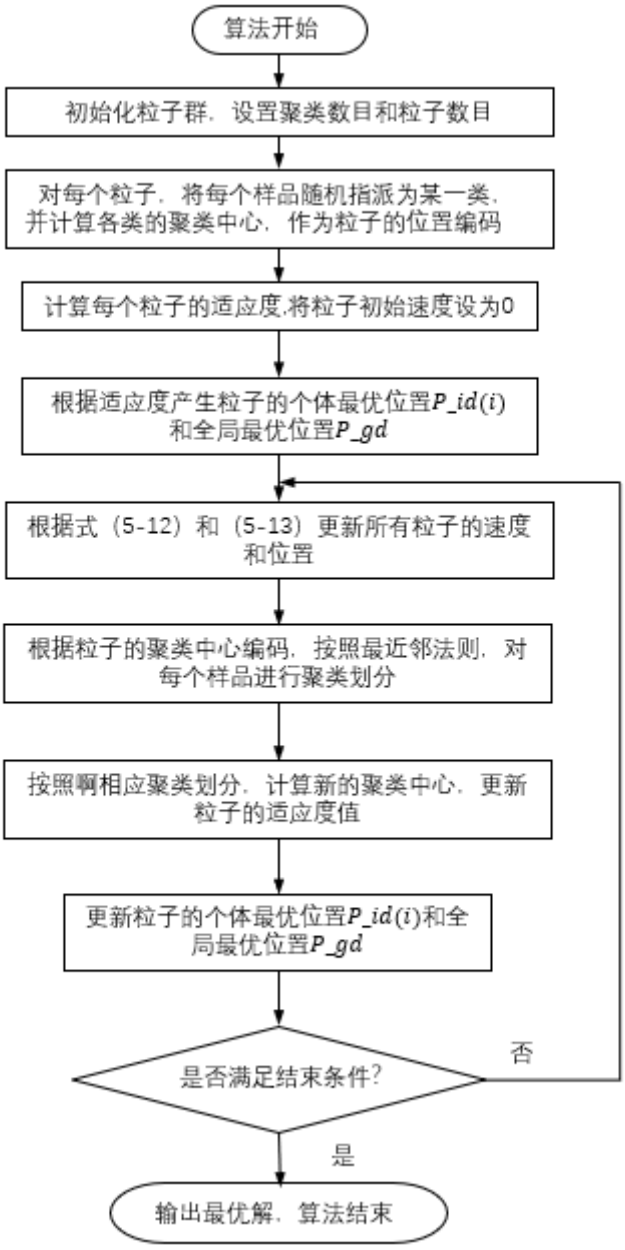
根据式（5-1）和式（5-2），可以得到粒子 i 的速度和位置更新公式。

$$\begin{aligned} \text{Particle}(i).\text{velocity}[\,]' = {} & \omega \text{Particle}(i).\text{velocity}[\,] + \\ & \eta_1 \text{rand}()(P\_id(i).\text{location}[\,] - \text{Particle}(i).\text{location}[\,]) + \\ & \eta_2 \text{rand}()(P\_gd.\text{location}[\,] - \text{Particle}(i).\text{location}[\,]) \end{aligned} \tag{5-12}$$

$$\text{Particle}(i).\text{location}[\,]' = \text{Particle}(i).\text{location}[\,] + \text{Particle}(i).\text{velocity}[\,]' \tag{5-13}$$

实现步骤

根据已定义好的粒子群结构，采用前面介绍的粒子群优化算法，可实现求解聚类问题的 最优解。粒子群聚类算法流程示意图如图 5-4 所示。



①粒子群的初始化。给定聚类数目 centerNum 和粒子数量 N，对于第 i 个粒子 Particle(i)，先 将每个样品随机指派为某一类，作为最初的聚类划分，并计算各类的聚类中心，作为粒子 i 的 位置编码 Particle(i).location[]，计算粒子的适应度 Particle(i).fitness，设置粒子 i 各中心的初始速 度为 0。反复进行 MaxIter 次。

②根据初始粒子群得到粒子的个体最优位置 P_id(i)，i=1，…，N 和全局最优位置 P_gd。

③根据式（5-12）和式（5-13）更新所有粒子的速度和位置。其中 η 1 =2，η2 =2，ω 按式（5-14） 取值，式中 iter 为当前迭代次数，MaxIter 为最大迭代次数，$\omega_{max}=1$，$\omega_{min}=0$。式（5-14）可描述为 ω 在迭代过程中由ω_{max}减少到ω_{min}

$$\omega=\omega_{max}-\text{ iter }\times\frac{\omega_{max}-\omega_{min}}{\text{MaxIter}}$$

④对每个样品，根据粒子的聚类中心编码，按照最近邻法则来确定该样品的聚类划分。

⑤对每个粒子，按照相应聚类划分，计算新的聚类中心，更新粒子的适应度值。

⑥对每个粒子 i，比较它的适应度值和它经历过的最好位置 P_id(i)的适应度值，如果前者 更好，更新 P_id(i)。

⑦对每个粒子 i，比较它的适应度值和群体所经历的最好位置 P_gd 的适应度值，如果前者 更好，更新 P_gd。

⑧如果达到结束条件（等到足够好的位置或最大迭代次数），则结束算法，输出全局最优 解；否则转步骤③继续迭代。

4. 编程代码 具体实现方法见作者撰写的《模式识别与智能计算——Matlab 技术实现》一书。

5. 效果图 粒子群聚类算法的效果图如图 5-5 所示。

（a）原始数据　（b）选择欧氏距离

（c）设定聚类数目和最大迭代次数　（d）聚类结果

图 5-5 粒子群聚类算法效果图

1.5.2.5 简单实战

https://www.jianshu.com/p/c0da3b3ab70a

我们查找下面这个函数的最优解。

$$f(x,y,m)\begin{cases}30x-y; & x<m,y<m\\ 30y-x; & x<m,y\geq m\\ x^2-\frac{y}{2}; & x\geq m,y<m\\ 20y^2-500x; & x\geq m,y\geq m\end{cases}$$

我们所要做的步骤，分别是初始化（init）、循环更新位置、速度（Update_position）与最大值记（Update_best），以及输出结果（info），并且将迭代部分的两步包在一起，形成一个函数（pso）

1.__init__分析和实现

前面已经反复提到过，__init__方法会作为初始化的函数，需要传递方法、参数，这在我们上面定义的函数签名上有很好的体现。

首先是 func 参数，按理来说，这里应该是函数（method），不过这是我的习惯，喜欢写方法（function），无伤大雅；这里用来传递函数引用，无论是 def 出来的函数，还是 lambda 表达式出来的函数都没问题。

bound 参数为边界，就是变量的取值范围，数据格式为((x_min, x_max), (y_min, y_max), z,...)，简单来说就是个容器，里面装着许多个二元容器或单个变量。例如：[(0, 60), (20, 70), (23, 45.5), 7]，其中二元容器代表变量的上下限，单个变量为变量只能取得唯一值。此处只支持这些，如果有类似：(min, None)这种

单边界或((min1, max1), (min2, max2))这种多边界的需求，可以自己去实现。

POP_SIZE 为粒子数，可以看成一个范围内觅食鸟的个数。

w、c1、c2 是后面更新公式所需要的参数，我们更新的时候再讲解。

v_max 为初始速度的最大值。

var_name 是变量名，None 为默认，前面加上*参数的意思是：想要给 var_name 赋值，必须用 var_name=...的形式。

```
    def __init__(self, func, bound, POP_SIZE, w=1, c1=0.2, c2=0.2, v_max=0.05, *, var_name=None):
        bounds = Counter([isinstance(a, Iterable) for a in bound])[True]
        Var_size = int(np.ceil(POP_SIZE ** (1/bounds)))
        vals = [np.linspace(var[0], var[1], Var_size) if isinstance(var, Iterable) else np.array([var]) for var
in bound]
        vals = np.array(list(map(lambda var: var.flatten(), np.meshgrid(*vals))))
        self.var_quantity, self.POP_SIZE = vals.shape
        self.func = func
        self.bound = bound
        self.w = w
        self.c1=c1
        self.c2=c2
        self.v_max = v_max
        self.var_name = var_name
```

第一行，用计数器数一个推导列表，这个列表由 bool 类型组成，当前元素含义是：bound 参数当前位置元素是否可迭代（既是二元组），然后查里面有多少个 True；因此第一行的含义是：看变量中有多少个变量是在范围内取值。

第二行实现的是这个公式：$\mathrm{var}_{\mathrm{size}} = \sqrt[bounds]{pop\_size}$。假如我们有$\mathrm{x,y}$两个变量，各取 10 个数字，那么$(\mathrm{x,y})$的元组是否就有 100 种可能的取值？以此推广，假如有$\mathrm{x,y,z}$三个变量，各取 5 个值，$(\mathrm{x,y,z})$元组就有 125 种可能取值；假如我们需要 100 个粒子，有三个变量，那么就，结果是 5，虽然能得到的是 125 个粒子，但结果近似；这里的算法不但是根据变量数，还要求变量不能是那种唯一取值的变量，因此开根号数由第一行得到。

第三行，确定所有变量的取值；如果是范围内取值，将会由其最小值到最大值，个数为上一行计算的进行填充；如果是单个值，就只计入一个值。例如上面的例题，、变量为范围取值，是单个取值，我们需要 100 个粒子，上面计算的，根据和的取值范围，和的取值列表都为[0, 6.667, 13.333, 20, 26.667, 33.333, 40, 46,667, 53,555, 60]，而的取值列表为[30]。这样能做到像网状一样的均匀取值。

第四行，np.meshgrid 函数就能做到找出所有的取值，我们将所有变量传入，它将返回“变量数+1”维的数组，最外面一层可以解包给各个变量，每个变量持有“变量数”维数组，我们可以将其看做“变量数-1”维数组，每一个维度为其他变量的取值个数，最内层的元素是原来的变量本身。听起来很绕口，自己试试就明白了，这个方法常用于多维作图铺开取值网格。不过多维的数组处理起来很头疼，因此我将它们都用 fatten 铺成了一维数组，原来位置的对应关系没有改变，现在 vals 成了二维数组，一维数的含义是变量数，二维数含义是最终粒子个数。这就是第五行做的事。

之后几行是参数赋值。velocity 是和粒子同样维度的随机序列，意思是每一个粒子的每一个变量都有速度。最后记录当前位置为粒子最优位置。

get_fitness 就是获得所有粒子的适应度，因此将粒子容器中的每一个粒子代入到函数 func 中计算即可。

```
    def get_fitness(self):
        return np.array([self.func(*particle) for particle in self.particles])
```

之前函数直接 return func(*self.particles)，这样能调用 numpy 中数组的矩阵运算，但这个函数中存在判断，因此不能简单送入方法，而选择用列表推导实现。

update_position 是更新位置和速度的函数。

```
def update_position(self):
    for index, particle in enumerate(self.particles):
        V_k_plus_1 = self.w * self.velocity[index] \
                     + self.c1*np.random.rand()*(self.person_best[index]-particle) \
                     + self.c2*np.random.rand()*(self.global_best-particle)

        self.particles[index] =  self.particles[index] + V_k_plus_1
        self.velocity[index] = V_k_plus_1

        for i, var in enumerate(particle):
            if isinstance(self.bound[i], Iterable):
                if var < self.bound[i][0]:
                    self.particles[index][i] = self.bound[i][0]
                elif var > self.bound[i][1]:
                    self.particles[index][i] = self.bound[i][1]
            elif var != self.bound[i]:
                self.particles[index][i] = self.bound[i]
```

上面一段先算出V^{K+1}的速度，然后改变位置和原有速度；下面一段是用于检查，变量是有取值范围的，如果超出范围，将重置为边界，如果是唯一取值，将让值不变。

update_best 更新最优记录。

```
def update_best(self):
    global_best_fitness = self.func(*self.global_best)
    person_best_value = np.array([self.func(*particle) for particle in self.person_best])
    for index, particle in enumerate(self.particles):
        current_particle_fitness = self.func(*particle)
        if current_particle_fitness > person_best_value[index]:
            person_best_value[index] = current_particle_fitness
            self.person_best[index] = particle
        if current_particle_fitness > global_best_fitness:
            global_best_fitness = current_particle_fitness
            self.global_best = particle
```

pso 执行两个更新。

```
def pso(self):
    self.update_position()
    self.update_best()
```

info 输出当前粒子信息，如果没传入变量名（var_name 为 None），就默认采用x_n这种形式。

```
    def info(self):
        result = pd.DataFrame(self.particles)
        if self.var_name == None:
            result.columns = [f'x{i}' for i in range(len(self.bound))]
        else:
            result.columns = self.var_name
        result['fitness'] = self.get_fitness()
        return result
```

我们准备好需要传递的参数，传入构造方法里面。然后不断迭代，由于粒子群算法属于群体进化，因此我们可以打印总体的加和方式，体现进化的效果，并打印出全体最优解及最优取值。

```
    func = lambda x, y, m: 30*x-y if x < m and y < m else 30*y-x if x<m and y>=m else x**2-y/2 if x>=m and y<m else 20*(y**2)-500*x
    bound = ((0, 60), (0, 60), 30)
    var_name = ['x', 'y', 'm']
    POP_SIZE = 100
    w = 1
    c1 = 0.2
    c2 = 0.2
    v_max = 0.05
    pso = PSO(func, bound, POP_SIZE, w, c1, c2, v_max, var_name=var_name)
    for _ in range(1000):
        pso.pso()
        print(pso.get_fitness().sum())

    print(pso.global_best, func(*pso.global_best))
    pso.info()   #调用 info 即可看到格式化的结果输出
```

1.6 对抗搜索：估值决策、最小最大值搜索、Alpha-Beta 剪枝搜索

本章讨论竞争环境，竞争环境中每个 Agent 的目标之间是有冲突的，这就引出了对抗搜索问题———通常被称为博弈。

数学中的博弈论，是经济学的一个分支，把多 Agent 环境看成是博弈，其中每个 Agent 都会受到其他 Agent 的"显著"影响，不论这些 Agent 间是合作的还是竞争的'。人工智能中"博弈"通常专指博弈论专家们称为有完整信息的、确定性的、轮流行动的、两个游戏者的零和游戏（如国际象棋）。术语中，这是指在确定的、完全可观察的环境中两个 Agent 必须轮流行动，在游戏结束时效用值总是相等并且符号相反。例如下国际象棋，一个棋手赢了，则对手一定是输了。正是 Agent 之间效用函数的对立导致了环境是对抗的。

从人类文明产生以来，博弈就和人类智慧如影随形——有时甚至到了令人担忧的程度。对于人工智能研究人员来说，博弈的抽象特性使得博弈成为感兴趣的研究对象。博弈游戏中的状态很容易表示，Agent 的行动数目通常受限，而行动的输出都有严谨的规则来定义。体育游戏如台球和冰球，则有复杂得多的描述，有更大范围的可能行动，也有不够严谨的规则来定义行动的合法性。所以除了足球机器人，体育游戏目前并没有吸引人工智能领域的很大兴趣。

与之前讨论的大多数玩具问题不同，博弈因为难于求解而更加令人感兴趣。例如国际象棋的平均分支因子大约是 35，一盘棋一般每个棋手走 50 步，所以搜索树大约有 35100 或者 10154 个结点（尽管搜索图

"只可能"有大约 1040 个不同的结点)。如同现实世界，博弈要求具备在无法计算出最优决策的情况下也要给出某种决策的能力。博弈对于低效率有严厉的惩罚。在其他条件相同的情况下，只有一半效率的 A*搜索意味着运行两倍长的时间，于是只能以一半效率利用可用时间的国际象棋程序就很可能被击败。所以，博弈在如何尽可能地利用好时间上产生了一些有趣的研究结果。

我们从最佳招数的定义和寻找它的搜索算法开始。接着讨论时间有限时如何选择好的招数。剪枝允许我们在搜索树中忽略那些不影响最后决定的部分，启发式的评估函数允许在不进行完全搜索的情况下估计某状态的真实效用值。5.5 节讨论诸如西洋双陆棋这类包含概率因素的游戏;我们也讨论桥牌，它包含不完整信息，桥牌中每个人都不能看到所有的牌。最后我们看看最高水平的博弈程序如何与人类对手抗衡以及未来的发展趋势。

首先考虑两人参与的游戏：MAX 和 MIN，马上就会讨论这样命名的原因。MAX 先行，两人轮流出招，直到游戏结束。游戏结束时给优胜者加分，给失败者罚分。游戏可以形式化成含有下列组成部分的一类搜索问题。

- So：初始状态，规范游戏开始时的情况。
- PLAYER（s）：定义此时该谁行动。
- ACTIONS（s）：返回此状态下的合法移动集合。
- RESULT（s，a）：转移模型，定义行动的结果。
- TERMINAL-TEST（s）：终止测试，游戏结束返回真，否则返回假。游戏结束的状态称为终止状态。
- UTLITY（s，p）：效用函数（也可称为目标函数或收益函数），定义游戏者 p 在终止状态 s 下的数值。在国际象棋中，结果是赢、输或平，分别赋予数值+1，0，或 1/2。有些游戏可能有更多的结果，例如双陆棋的结果是从 0 到+192。零和博弈是指在同样的棋局实例中所有棋手的总收益都一样的情况。国际象棋是零和博弈，棋局的收益是 0+1，1+0 或 1/2+1/2。"常量和"可能是更好的术语，但称为零和更传统，可以将这看成是下棋前每个棋手都被收了 1/2 的入场费。

初始状态、ACTIONS 函数和 RESULT 函数定义了游戏的博弈树———其中结点是状态，边是移动。图 5.1 给出了井字棋的部分博弈树。在初始状态 MAX 有九种可能的棋招。游戏轮流进行，MAX 下 X，MIN 下 O，直到到达了树的终止状态即一位棋手的标志占领一行、一列、一对角线或所有方格都被填满。叶结点上的数字是该终止状态对于 MAX 来说的效用值;值越高对 MAX 越有利，而对 MIN 则越不利（这也是棋手命名的原因）。

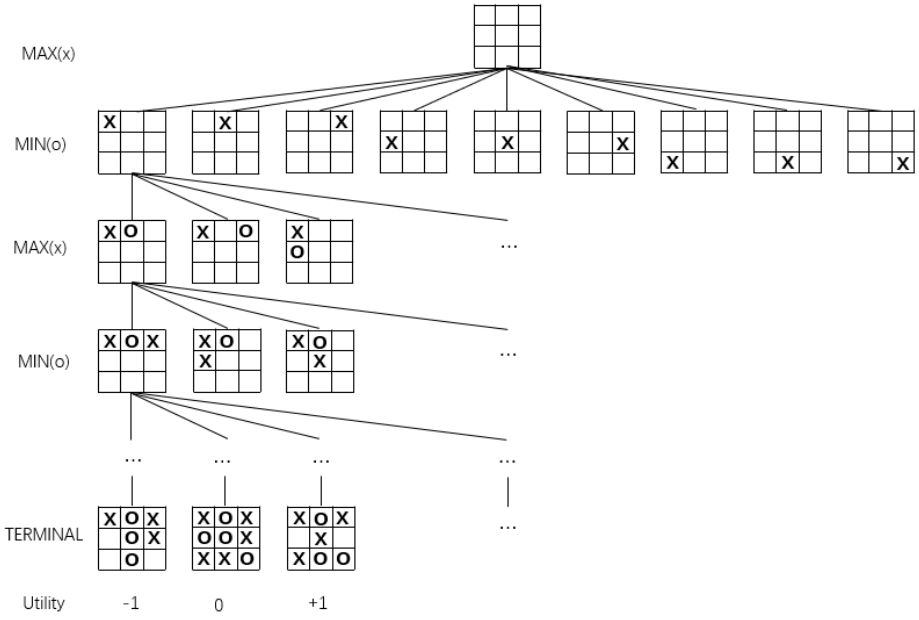


图 5.1 井字棋游戏的（部分）搜索树

最上面的结点是初始状态，MAX 先走棋，放置一个 X 在空位上。图显示了搜索树的一部分，给出 MIN 和 MAX 的轮流走棋过程，直到到达终止状态。所有终止状态都按照游戏规则被赋予了效用值

1.6.1 对抗搜索中的优化决策

在一般搜索问题中，最优解是到达目标状态的一系列行动——终止状态即为取胜。在对抗搜索中，MIN 在博弈中也有发言权。因此 MAX 必须找到应急策略，制定出 MAX 初始状态下应该采取的行动，接着是 MIN 行棋，MAX 再行棋时要考虑到 MIN 的每种可能的回应，依此类推。这有些类似于 AND-OR 搜索算法（图 4.11）MAX 类似于 OR 结点，MIN 类似于于 AND 结点。粗略地说，当对手不犯错误时最优策略能够得到至少不比任何其他策略差的结果。我们将从寻找最优策略开始。

即使是井字棋这样简单的游戏，也很难在一页画出它的整个博弈树，所以讨论如图 5.2 所示的更简单游戏。在根结点 MAX 的可能行棋为 a，a2 和 a。对于 a，MIN 可能的对策有 b，b2 和 b3，等等。这个特别的游戏在 MAX 和 MIN 各走一步后结束（按照博弈的说法，这棵博弈树的深度是一步，这包括两个单方招数，每个单方招数称为一层）。终止状态的效用值范围是从 2 到 14。

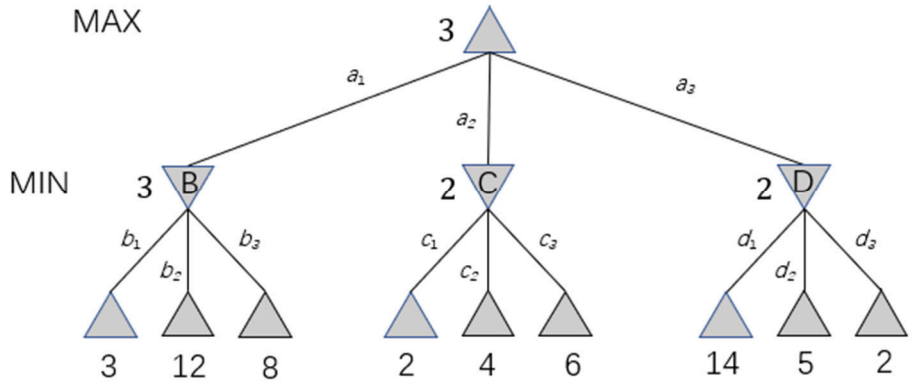


图 5.2 两层博弈树

△结点是“MAX”结点，代表轮到 MAX 走，▽结点是“MIN 结点”。终止结点显示 MAX 的效用值；其他结点标的是它们的极小极大值。MAX 在根节点的最佳行棋是a_1，因为它指向有最高的极小极大值的后继，而 MIN 此时的最佳行棋是b_1，因为它指向有最低的极小极大值的后继

给定一棵博弈树，最优策略可以通过检查每个结点的极小极大值来决定，记为 MINIMAX（n）。假设两个游戏者始终按照最优策略行棋，那么结点的极小极大值就是对应状态的效用值（对于 MAX 而言）。显然地，终止状态的极小极大值就是它的效用值自身。更进一步，对于给定的选择，MAX 喜欢移动到有极大值的状态，而 MIN 喜欢移动到有极小值的状态。所以得到如下公式：

$$\text{MINIMAX}(s)=\begin{cases}\text{UTILITY}(s) & s\text{为终止状态}\\ \max_{a\in Actions(s)}\text{MINIMAX}(\text{RESULT}(s,a)) & s\text{为}MAX\text{结点}\\ \min_{a\in Actions(s)}\text{MINIMAX}(\text{RESULT}(s,a)) & s\text{为}MIN\text{结点}\end{cases}$$

我们将这些定义应用于图 5.2 中的博弈树。底层终止结点的效用值即为它们的效用函数值。第一个 MIN 结点为 B，其三个后继的值分别是 3、12 和 8，所以它的极小极大值是 3。类似地，可以得出其他两个 MIN 结点的极小极大值都是 2。根是 MAX 结点，其后继结点分的极小极大值分别为 3、2 和 2，所以它的极小极大值是 3。可以确定在根结点的极小极大决策：对于 MAX 来说 a1 是最优选择，因为它指向有最高的极小极大值的终止状态。

对 MAX 的最挂行棋进行求解时做了 MIN 也按最佳行棋的假设——尽可能最大化 MAX 的最坏情况。如果 MIN 不按最佳行棋行动怎么办?这种情况下显然（习题 5.7）MAX 可以做得更好。可能有一些策略在对付非最优化对手方面做得比极小极大策略好，但是用这些策略对付最优化对手则会得到更差的结果。

1.6.2 估值决策

1.6.3 最小最大值搜索

极小极大算法（图 5.3）从当前状态计算极小极大决策。它使用了简单的递归算法计算每个后继的极小极大值，直接实现上面公式的定义。递归算法自上而下一直前进到树的叶结点，然后随着递归回溯通过搜索树把极小极大值回传。例如，在图 5.2 中，算法先递归到三个底层的叶结点，对它们调用 UTLITY 函数得到效用值分别是 3、12 和 8。然后它取最小值 3 作为回传值返回给结点 B。通过类似的过程可以分别得到 C 和 D 的回传值均为 2。最后在 3、2 和 2 中选取最大值 3 作为根结点的回传值。

极小极大算法对博弈树执行完整的深度优先探索。如果树的最大深度是 m，在每个结点合法的行棋有 b 个，那么极小极大算法的时间复杂度是 O（bm）。一次性生成所有的后继的算法，空间复杂度是 O（bm），而每次生成一个后继的算法（参见原书第 87 页），空间复杂度是 O（m）。当然对于真实的游戏，这样的时间开销完全不实用，不过此算法仍然可以作为对博弈进行数学分析和设计实用算法的基础。

算法：最小最大值搜索

1. 指定当前节点和搜索深度
2. 如果能得到确定的结果或者深度为零，使用评估函数返回局面得分
3. 如果轮到对手走棋，是极小节点，选择一个得分最小的走法 min(α, minimax(child, depth-1))
4. 如果轮到我们走棋，是极大节点，选择一个得分最大的走法 max(α, minimax(child, depth-1))
5. 结束

1.6.4 Alpha-Beta 剪枝搜索

极小极大值搜索的问题是必须检查的游戏状态的数目是随着博弈的进行呈指数级增长。不幸的是，指数增长无法消除，不过我们还是可以有效地将其减半。这里的技巧是可能不需要遍历博弈树中每一个结点就可以计算出正确的极小极大值。于是，借用第 3 章中的剪枝思想尽可能消除部分搜索树。这种特别技术称为$\boldsymbol{\alpha-\beta}$剪枝。将此技术应用到标准的极小极大搜索树上，会剪掉那些不可能影响决策的分支，仍然返回和极小极大算法同样的结果。

再来看图 5.2 中的两层博弈树。重新观察最优决策的计算过程，特别注意此过程中在每个结点的已知

信息。图 5.5 解释了每一步骤。结果发现可以在不计算评价其中两个叶结点的情况下就确定极小极大决策。

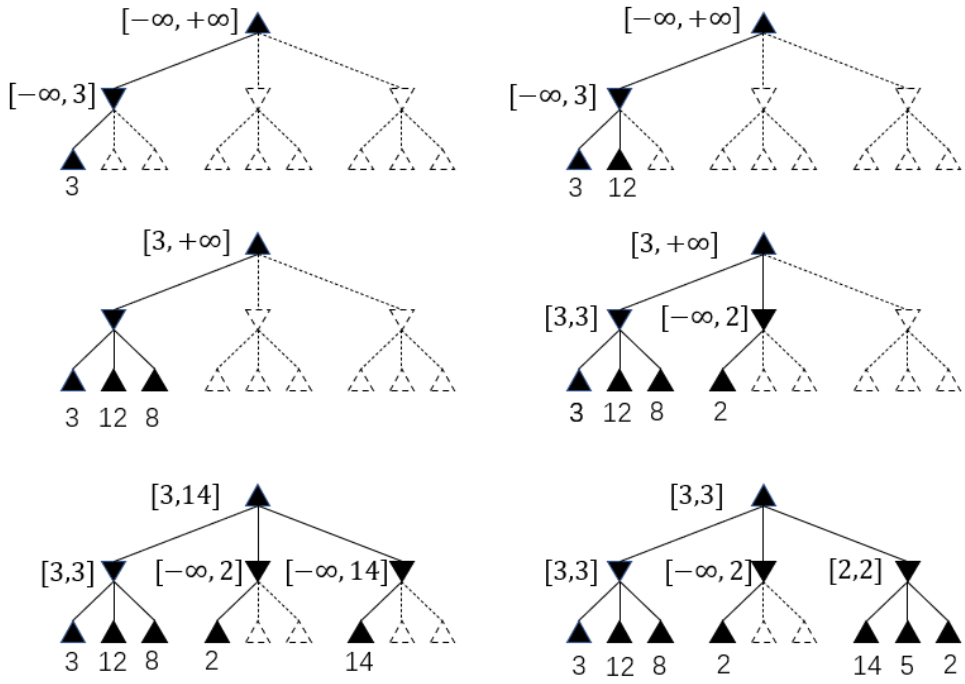


图 5.5 中博弈树的最优决策过程

每一结点上标出了可能的取值范围。(a）B 下面的第一个叶结点值为 3。因此作为 MIN 结点的 B 值至多为 3。(b）B 下面的第二个叶结点值为 12。MIN 不会用这招，所以 B 的值仍然至多为 3。(c）B 下面的第三个叶子值为 8：此时已经观察了 B 的所有后继，所以 B 的值就是 3。现在可以推断根结点的值至少为 3，因为 MAX 在根结点有值为 3 的后继。(d）C 下面的第一个叶结点值为 2。因此 C 这个 MIN 结点的值至多为 2。不过已经知道 B 的值是 3，所以 MAX 不会选择 C。这时再考察 C 的其他后继已经没有意义了。这就是$\boldsymbol{\alpha-\beta}$剪枝的实例。(e）D 下面的第一个叶结点值为 14，所以 D 的值至多为 14。这比 MAX 的最佳选择（即 3）要大，所以继续探索 D 的其他后继。还要注意现在知道根的取值范围，根结点的值至多为 14。(f）D 的第二个后继值为 5，所以我们又必须继续探索。第三个后继值为 2，所以 D 的值就是 2 了。最终 MAX 在根结点的决策是走到值为 3 的 B 结点

还可以把这个过程看作是对 MINIMAX 公式的简化。假设图 5.5 中的 C 结点的两个没有计算的子结点的值是 x 和 y。根结点的值计算如下：

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3,z,2) \\ &= 3\end{aligned}$$

即，根结点的值以及因此做出的极小极大决策与被剪枝的叶结点 x 和 y 无关。

$\boldsymbol{\alpha-\beta}$剪枝可以应用于任何深度的树，很多情况下可以剪裁整个子树，而不仅仅是剪裁叶结点。一般原则是：考虑在树中某处的结点 n（见图 5.6），选手选择移动到该结点。如果选手在 n 的父结点或者更上层的任何选择点有更好的选择 m，那么在实际的博弈中就永远不会到达 n。所以一旦发现关于 n 的足够信息（通过检查它的某些后代），能够得到上述结论，我们就可以剪裁它。

记住极小极大搜索是深度优先的，所以任何时候只需考虑树中某条单一路径上的结点。a-β 剪枝的名称取自描述这条路径上的回传值的两个参数：

$\boldsymbol{\alpha}$= 到目前为止路径上发现的 MAX 的最佳（即极大值）选择

$\boldsymbol{\beta}$= 到目前为止路径上发现的 MIN 的最佳（即极小值）选择

$\boldsymbol{\alpha-\beta}$搜索中不断更新$\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$的值，并且当某个结点的值分别比目前的 MAX 的$\boldsymbol{\alpha}$或者 MIN 的$\boldsymbol{\beta}$值更差的时候剪裁此结点剩下的分支（即终止递归调用）。

1.6.5 简单实战

AlphaBeta 剪枝算法是对 Minimax 方法的优化，能够极大提高搜索树的效率，如果对这个算法感兴趣的可以去参考相关资料。

当正确理解 AlphaBeta 剪枝算法后，还可以将它应用在象棋、围棋等一些高级游戏的算法搜索上，使得电脑寻找最优胜率的速度加快。

用如下的 9 个数字来表示棋盘的位置:

0 1 2
3 4 5
6 7 8

设定获胜的组合方式(横、竖、斜)

```
WINNING_TRIADS = ((0, 1, 2), (3, 4, 5), (6, 7, 8),
                  (0, 3, 6), (1, 4, 7),(2, 5, 8),
                  (0, 4, 8), (2, 4, 6))
```

设定棋盘按一行三个打印，用一维列表表示棋盘。

```
PRINTING_TRIADS = ((0, 1, 2), (3, 4, 5), (6, 7, 8))
SLOTS = (0, 1, 2, 3, 4, 5, 6, 7, 8)
```

用-1 表示 X 玩家 0 表示空位 1 表示 O 玩家.

```
X_token = -1
Open_token = 0
O_token = 1

MARKERS = ['_', 'O', 'X']
END_PHRASE = ('平局', '胜利', '失败')
```

运用 AlphaBeta 剪枝来计算当前局面的分值，因为搜索层数少，总能搜索到最终局面，估值结果为[-1,0,1]

```
def alpha_beta_valuation(board, player, next_player, alpha, beta):
    wnnr = winner(board)
    if wnnr != Open_token:
        # 有玩家获胜
        return wnnr
    elif not legal_move_left(board):
        # 没有空位,平局
        return 0
    # 检查当前玩家"player"的所有可落子点
    for move in SLOTS:
        if board[move] == Open_token:
            board[move] = player
            # 落子之后交换玩家，继续检验
            val = alpha_beta_valuation(board, next_player, player, alpha, beta)
            board[move] = Open_token
            if player == O_token:   # 当前玩家是 O,是 Max 玩家(记号是 1)
                if val > alpha:
                    alpha = val
                if alpha >= beta:
                    return beta    # 直接返回当前的最大可能取值 beta, 进行剪枝
            else:   # 当前玩家是 X,是 Min 玩家(记号是-1)
                if val < beta:
                    beta = val
                if beta <= alpha:
                    return alpha    # 直接返回当前的最小可能取值 alpha, 进行剪枝
    if player == O_token:
        retval = alpha
    else:
        retval = beta
    return retval
```

打印当前棋盘

```
def print_board(board):
   for row in PRINTING_TRIADS:
        r = ' '
        for hole in row:
            r += MARKERS[board[hole]] + ' '
        print(r)
```

判断棋盘上是否还有空位

```
def legal_move_left(board):
    for slot in SLOTS:
        if board[slot] == Open_token:
            return True
    return False
```

判断局面的胜者,返回值-1 表示 X 获胜,1 表示 O 获胜,0 表示平局或者未结束

```
def winner(board):
    for triad in WINNING_TRIADS:
        triad_sum = board[triad[0]] + board[triad[1]] + board[triad[2]]
        if triad_sum == 3 or triad_sum == -3:
            return board[triad[0]]    # 表示棋子的数值恰好也是-1:X,1:O
    return 0
```

决定电脑(玩家 O)的下一步棋,若估值相同则随机选取步数

```
def determine_move(board):
    best_val = -2    # 本程序估值结果只在[-1,0,1]中
    my_moves = []
    print("开始思考")
    for move in SLOTS:
        if board[move] == Open_token:
            board[move] = O_token
            val = alpha_beta_valuation(board, X_token, O_token, -2, 2)
            board[move] = Open_token
            print("Computer 如果下在", move, ",将导致", END_PHRASE[val])
            if val > best_val:
                best_val = val
                my_moves = [move]
            if val == best_val:
                my_moves.append(move)
    return random.choice(my_moves)
```

主函数,先决定谁是 X(先手方),再开始下棋

```
HUMAN = 1
COMPUTER = 0
def main():
    next_move = HUMAN
    opt = input("请选择先手方，输入 X 表示玩家先手，输入 O 表示电脑先手：")
    if opt == "X":
        next_move = HUMAN
    elif opt == "O":
        next_move = COMPUTER
    else:
        print("输入有误，默认玩家先手")
    # 初始化空棋盘
    board = [Open_token for i in range(9)]
    # 开始下棋
    while legal_move_left(board) and winner(board) == Open_token:
        print()
        print_board(board)
        if next_move == HUMAN and legal_move_left(board):
            try:
                humanmv = int(input("请输入你要落子的位置(0-8)："))
                if board[humanmv] != Open_token:
                    continue
                board[humanmv] = X_token
                next_move = COMPUTER
            except:
                print("输入有误，请重试")
                continue
        if next_move == COMPUTER and legal_move_left(board):
            mymv = determine_move(board)
            print("Computer 最终决定下在", mymv)
            board[mymv] = O_token
            next_move = HUMAN
    # 输出结果
    print_board(board)
    print(["平局", "Computer 赢了", "你赢了"][winner(board)])
```

1.7 习题

1. 解释为什么问题的形式化必须在目标的形式化之后。
2. 你的目标是让机器人走出迷宫。机器人面朝北，开始位置在迷宫中间。你可以让机器人转向面朝东、南、西或北。你可以让机器人向前走一段距离，在撞墙之前它会停步。

   a.将问题形式化。状态空间有多大？

   b.在迷宫中游走，在两条路或更多路交叉的路口可以转弯。重新形式化这个问题。现在状态空间有多大？

   c.从迷宫中的任一点出发，我们可以朝四个方向中的任一方向前进直到可以转弯的地方，而且我们只需

要这样做。重新对这个问题进行形式化。我们需要记录机器人的方向吗？

d.在我们对问题的最初描述中已经对现实世界进行了抽象，限制了机器人的行动并移除了细节。列出三个我们做的简化。

3. 两个朋友住在地图上的不同城市中。每一轮次，两个人都可以前进到地图上相邻的城市。从城市到相邻城市丿耗费的时间与城市间的距离相等，但是每一轮次先到达相邻城市的人要等另一人抵达他的相邻城市（到达后打手机),方可进入下一轮次。我们希望这两个朋友能尽快相遇。

a.请详细形式化此问题（你会发现在这定义一些形式符号很有帮助）。

b.用D(i, j)表示城市i和j之间的直线距离。下列启发式函数哪些是可采纳的？

（i）D(i, j)

（ii）2 D(i, j)；

(iii）D(i, j)/2。

c.是否存在完全联通图却没有解？

d.是否存在这样的地图．所有解都需要一个朋友访问同一城市两次？

4. 证明八数码问题的所有状态可以划分为两个不相交的子集，处在同一个子集中的状态之间可以相互到达，处在不同子集中的两个状态之间必不可达到。设计一个算法判断一个给定的状态属于哪个子集，并解释为什么这对于生成随机状态是有用的。

5. 对以下问题给出完整的形式化。选择的形式化万法要足够精确以便于实现．

a.只用四种颜色对半面地图着色，要求每两个相邻的地区不能具有相同的颜色。

b.屋子里有只 3 英尺高的猴子，离地 8 英尺的屋顶上挂着一串香蕉。猴子想吃香蕉。屋子里有两个可叠放、可移动、可攀爬的 3 英尺高的箱子。

c.有这样一个程序，当输入一个包含很多记录的文件时会输出消息“不合法的输入记录每个记录的处理都是独立的．请找出报错的是哪个记录。

d.有三个水壶，容量分别为 12 加仑、8 加仑和 3 仑，还有一个放液嘴。可以把水壶装满或者倒空，从一个壶倒进另一个壶或者倒在地上。请量出刚好 1 加仑水

6. 传教士和野人问题。三个传教士和三个野人在河的一岸，有一条能载一个人或者两个人的船。请设法使所有人都渡到河的另一岸，要求在任何地方野人数都不能多于传教士的人数。

a.请对该问题进行详细形式化，只描述确保该问题求解所必需的特性，画出完整的状态空间图到

b.应用合适的搜索算法求出该问题的最优解?对于这个问题检查重复状态是个好主意吗？

c.这个问题的状态空间很简单，你认为是什么导致人们求解它很凼难？

7. 对以下术语给出你自己的定义；状态，状态空间，搜索树，搜索结点，目标，行动，转移模型和分支因子。

8. 世界状态、状态描述和搜索结点有何不同？为什么要做这样的严格区分？

9. 判断对错并说明理由：

a.深度优先搜索至少要扩展与使用可采纳启发式的 A*一样多的结点。

b.h(n)=0 对于八数码问题是可采纳的启发式。

c.A*在机器人学中没有任何用处，原因是感知器、状态和行动都是连续的。

d.即使是通话零代价的情况下，宽度优先搜索依然是完备的。

e.假设车在棋盘上可以沿水平或垂直方向移动至任一方格中，但要注意不能跳过棋子。将车从方格 A 移到方格 B 的最小移动步数问题中，曼哈顿距离是可采纳的启发式。

10. 对随机产生的八数码问题（用曼哈顿距离）和 TSP（用 MST）问题，比较 A*算法和 RBFS 算法的性能。对结果进行讨论。八数码问题如果在启发式值上加上一个很小的随机数，会对 RBFS 的性能有何影响？

11. 跟踪 A*算法应用直线距离启发式求解从 Lugoj 到 Bucharest 问题的过程。给出结点扩展的顺序和每个结点的不和方值。

12. 设计一个状态空间，对其硬用 GRAPH-SEARCH 的 A*得到次优解，其中”）是可采纳的而不是一致的。
13. 设计一个启发函数，它在八数码问题中有时会估计过高，对某一特定问题它会求出次优解（可以用计算机编程找出).证明：如果被高估的部分不超过 c，A*算法返回的解代价比最优解代价多出的部分也不超过 c。
14. 证明如果启发式是一致的，它一定是可采纳的。构造一个非一致的可采纳启发式。
15. 旅行商问题（TSP)可以通过最小生成树(MST)启发式来解决，如果已经旅行已在进行中，MST 用于估计完成旅行的代价。一组城市的 MST 代价是连接所有城市的树的最小连接代价和。

    a.这个启发式是如何通过松地的 TSP 问题得到的。

    b.说明为何 MST 启发式比直线距离启发式有优势。

    c.编写 TSP 问题的实例生成器，城市的位置用在单位正方形内的随机点表示。

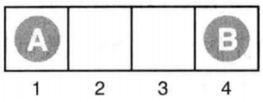
    d.在文献中找到构造 MST 的有效算法，并将之应用搜索来求解 TSP 问题实例。
16. 假设你有先知，OM(到，可以准确预测对手在任一状态下的行棋。利用这一点，给出博弈问题的形式化，并将之视为（单个 Agent)搜索问题。给出寻找最优解的算法。
17. 考虑两个八数码难题的问题求解。

    a.给出完整的问题形式化。

    b.可达的状态空间有多大？请给出精确的数字表达式。

    c.假设我们这样修改问题：两个选手轮流移动；用硬币来决定选手移动的是哪道题；第一个求解了某道题的就是赢家。在这样的假设下应该使用什么算法来确定移动？

    d.如果两位选手都是完美的，请说明最终总会有人获胜。
18. 考虑一个或多个如下的随机博弈：大富翁，拼字，已知定约的桥牌，Texas 扑克。请给出状态描述、行棋生成器、效用函数、评估函数。
19. 描述并实现一个实时的多人游戏环境，状态中包含时间，每个选手有固定的时间分配。
20. 讨论如何将标准博弈技术应用于连续物理状态空间的游戏，如网球、台球、门球。
21. 证明下面的断言：对于每棵博弈树，MAX 使用极小极大算法对抗次优招数的 MIN 得到的效用值不会比对抗最优招数的 MIN 得到的效用值低。你能否找出一棵博弈树，使得 MAX 用次优策略依然要好于次优 MIN 时的策略。
22. 考虑图中描述的两人游戏。



图一个简单游戏的初始棋局

选手 A 先走。两个选手轮流走棋，每个人必须把自己的棋子移动到任一方向上的相邻空位中．如果对方的棋子占据着相邻的位置，你可以跳过对方的棋子到下一个空位。（例如，A 在位置 3，B 在位置 2，那么 A 可以移回 1。）当一方的棋了移动到对方的端点时游戏结束。如果 A 先到达位置 4，A 的值为+1：如果 B 先到位置 1，A 的值为-1.

a.根据如下约定画出完整博弈树：

- 每个状态用，表示，其中和表示棋子的位置。
- 每个终止状态用方框画出，用圆圈写出它的博弈值。
- 把循环状态（在到根结点的路径上已经出现过的状态）画上双层方框。由于清楚他们的值，在圆圈里标记一个“？”。

b.给出每个结点倒推的极小极大值（也标记在圆圈里）。解释怎样处理“？”值和为什么这么处理。

c.解释标准的极小极大算法为什么在这棵博弈树中会失败，简要说明你将如何修正它，在(b)的图上画出你的答案。你修正后的算法对于所有包含循环的游戏都能给出最优决策吗？

d.这个 4-方格游戏可以推广到个方格，其中 n>2。证明如果 n 是偶数，A 一定能赢，而 n 是奇数则 A

一定会输。

23. 本题以井字棋（圈与十字游戏）为例练习博弈中的基本概念。定义X_n为恰好有 n 个 X 而没有 O 的行、列或者对角线的数目。同样O_n为正好有 n 个 O 的行、列或者对角线的数目。效用函数给$X_3 = 1$的棋局+1,给$XO_3 = 1$的棋局-1。所有其他终止状态效用值为 0。对于非终止状态，使用线性的评估函数定义为 $\text{Eval}(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$

a.估算可能的井字棋局数。

b.考虑对称性，给出从空棋盘开始的深度为 2 的完整博弈树（即，在棋盘上一个 X 一个 O 的棋局）。

c.标出深度为 2 的棋局的评估函数值。

d.使用极小极大算法标出深度为 1 和 0 的棋局的倒推值，并根据这些值选出最佳的起始行棋。

e.假设结点按对卩剪枝的最优顺序生成，圈出使用卩剪枝将被剪掉的深度为 2 的结点。

24. 如下定义井字棋家族。S 表示方格棋盘，W 表示赢的棋局。每个赢局是 S 的子集。例如，在标准井字棋中，S 是 9 格集合而 W 是 8 个子集：三行、三列和两个对角线。

在其他方面，这个游戏与标准井字棋相同。从窄棋盘开始，选手轮流在空格处画上自己的标记。如果选手画出了赢局，则赢得了比赛。如果棋盘上没有空格但没有人赢，则是和棋。

a.设 N=|S|到，即方格数。请给出井字棋博弈树中结点数上限，将之表示为关于 N 的函数。

b.给出博弈树在最坏情况下即 W={}时的下限。

c.请给出通用井字棋棋局的评估函数。该函数可能依赖于 S 和 W

d.假设可能在 100N 条机器指令内生成新棋局并检查它是否是赢局，假设是 2GHz 处理器。不计内存限制。利用你在 a 中的估算，在 1 秒的计算机时间内使用$\alpha - \beta$能够完全求解的博弈树是多大？1 分钟呢？1 小时呢？

25. 设计通用博弈程序，有能力完成多种博弈游戏。

a.实现下面一种或多种游戏的行棋生成器和评估函数：KaIah 游戏（美国播棋），翻转棋，西洋跳棋和国际象棋。

b.构造一个通用的$\alpha - \beta$博弈 Agent

c.比较增加搜索深度、改进行棋排序和改进评估函数对程序的影响。你的有效分支因子有多接近于完美行棋排序的理想情况呢？

|d.实现一个选择搜索算法，如 B*（Berliner,1979）对策数搜索(McAllester，1988)或 MGSS*（RusseII 和 WefaId,1989），并与 A*的性能作出比较。

26. 对于两人非零和游戏，每个人都有自己的效用函数并且相互知道，应如何修改极小极大算氵去和女．Y 枝算法进行求解？如果对终止效用值没有约束，别枝是否可能剪裁某一结点呢？如果每位选手任一状态的效用函数值至多相差常量呢？博弈是否成为合作了呢？