《人工智能》课程系列

博弈树搜索技术*

武汉纺织大学数学与计算机学院

杜小勤

2018/10/23

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: November 1, 2019。

1 概述

在多智能体 (Agent) 环境中，每个智能体需要考虑其它智能体的行动及其对自身的影响，关系错综复杂。

本章要讨论的博弈树搜索技术，涉及到 2 个智能体之间的对抗或竞争博弈。此处的博弈，专指有完整信息的、确定性的、轮流行动的、双人的零和游戏，例如国际象棋、围棋等[1]。

在双人游戏中，一方先行，另一方后行，两人交替出招，直至游戏结束。在下面的讨论中，我们规定先行的一方为 MAX 方，后行的一方为 MIN 方。之所以这样规定，原因在于：

- 零和博弈

  一般而言，双人游戏的结果有胜、负或平。胜利的一方得分，失败的一方不得分或罚分，平局时双方各得一半分。例如，在 TicTacToe、国际象棋等棋类游戏中，胜负平可以分别赋予数值为 1、-1 和 0。在此情况下，双方的总收益维持为常量；

- 双方是一种竞争关系

  在双人游戏中，对弈的双方都力图使自身的利益最大化。每一次出招时，先全面分析对方“所有可能”的行棋，然后选择自身利益最大化的招数。为了定量地反映出这种关系，规定先出招的一方为 MAX，后出招的一方为 MIN 方，并从 MAX 方的视角来定义胜负平的得分——例如，对于 MAX 方而言，胜利分值为 +1，失败分值为-1，平局分值为 0；对于 MIN 方而言，胜利分值为-1，失败分值为 1，平局分值为 0；

实际上，可以将博弈游戏形式化为一个搜索问题：

- S_0：初始状态；
- Player(s)：在状态 s 下，返回出招者——MAX 方或 MIN 方；
- Actions(s)：在状态 s 下，返回合法的动作集合；

[1]博弈论是经济学的一个分支。一般而言，包含多智能体的系统被称为博弈系统，其中的每个智能体都会明显地受到其它智能体的影响，不论这些智能体之间是合作关系，还是竞争关系。此外，包含非常多智能体的系统被称为经济系统，而不是博弈系统。

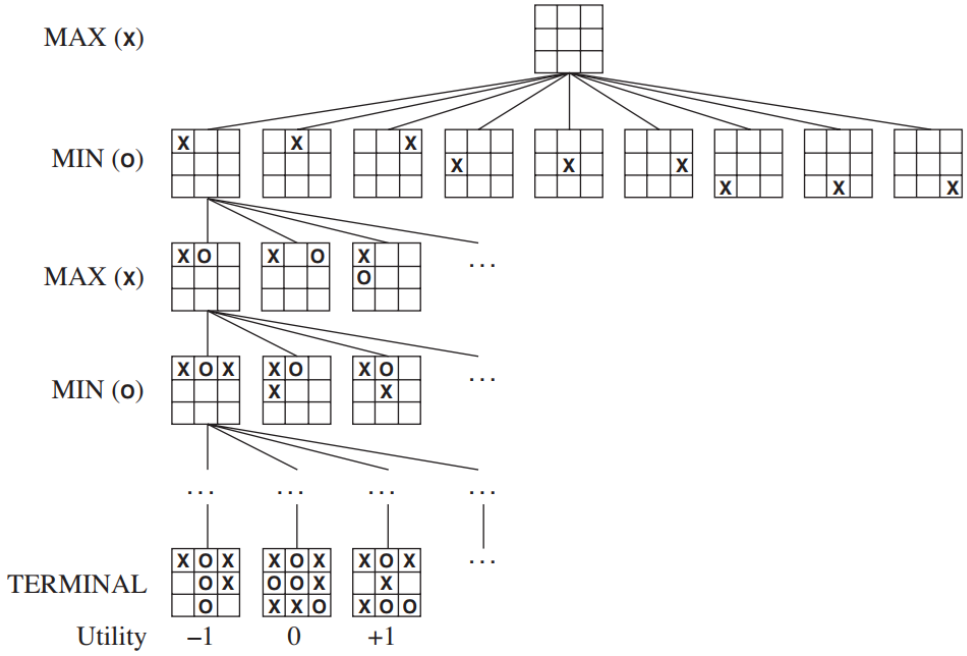

图 1-1: TicTacToe 的博弈树片段

- Result(s, a)：在状态 s 下，执行动作 a，将使状态发生转移；
- Terminal-Test(s)：返回游戏的状态——游戏是否结束；
- Utility(s, p)：在终局状态下，返回 MAX 或 MIN 的效用值 (即胜负平的收益值)；

其中，初始状态 S_0、Actions 函数、Result 函数定义了一棵博弈树 (Game Tree) 或对抗搜索树 (Adversarial Search Tree)。

在博弈树中，节点是状态，边表示动作或落子。如图1-1所示，展示了一棵 TicTacToe 博弈树的片段。可以看出，在初始状态下，MAX 先下，MIN 后下，双方轮流出招，直至棋局结束。在终局时，所有终局状态都被赋予了效用值——从 MAX 方的视角，定义了效用值——值越高，对 MAX 方越有利。

2 Minimax 算法

在一般的搜索问题中，搜索的目标是找到一条从起始节点到目标节点的最优路径。搜索完毕，智能体只需沿着最优路径一直走下去，最终必然会到达目标节点。

而在博弈树搜索中，双方要根据当前的状态，选择“‘最优”的动作执行，期望能够到达自己获胜的目标状态 (节点)。在理想情况下，可以把获胜的棋局状态看作是目标状态。这意味着，在搜索时，必须执行一次完整的搜索，即从当前状态出发，考虑对手的所有可能应对，然后针对每一个可能的应对，生成新的状态。上述过程循环往复，一直进行到棋局终局时为止。最后，智能体将根据所有可能的终局状态，依照某种算法 (例如 Minimax 算法) 做出在当前状态下对自己最有利的落子选择。

对于类似 TicTacToe 这样的双人棋类游戏，可以执行完全的搜索。然而，对于大多数棋类游戏而言，执行完全的搜索几乎是不可能的。例如，国际象棋博弈树的平均分支因子大约是 35，每盘棋双方的平均行棋步数各为 50 步，那么树的节点个数大约为 35^{100} 或者 10^{154} 个 (虽然不同的节点个数只有大约 10^{40} 个)。节点数目随搜索深度呈指数级增长，这对任何算法而言是一个巨大的灾难！

还有一个重要因素，即博弈树搜索不再是一个单纯的目标搜索问题——必须考虑对手的策略。因此，“最优解”并非真正的最优解，而是考虑双方策略时的最优解，它是双方妥协的结果。

因此，在博弈树搜索中，双方将始终依据“己方利益最大化，对方利益最小化”这一原则，选择对己方最有利的动作来执行[2]。由此，从当前状态出发，一直到棋局终局时，双方都将选择最有利己方的动作来执行，这将得到一个最优解。我们可以把在这种情况下得到的最优解称为“对抗最优解”或“博弈最优解”。

下面将给出一个能够体现上述策略的算法，即 Minimax 算法——在理想对手的假设下，MIN 方总是最小化 MAX 方的最大化努力，这是该算法名称的由来。Minimax 算法如下：

```
def Minimax(node, depth, player):
    if depth == 0 or node is a terminal node:
        return the heuristic value of node
    if player == True:
        bestValue = -ϖ
        for each child of node:
            v = Minimax(child, depth-1, False)
            bestValue = max(bestValue, v)
```

[2]在这种情况下，对弈的双方被称为理性或理想的对手。

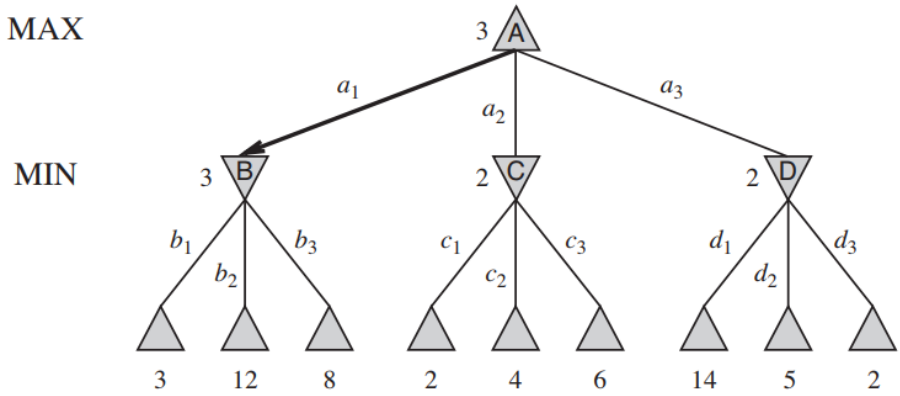

图 2-2: Two-Ply 博弈树示例

```
        return bestValue
    else:
        bestValue = +ϖ
        for each child of node:
            v = Minimax(child, depth-1, True)
            bestValue = min(bestValue, v)
        return bestValue
```

为便于理解算法的执行过程，我们将在一棵假想的简单博弈树上模拟执行一次 Minimax 算法。如图2-2所示，展示了一棵简单的 2 层 (Two-Ply) 博弈树[3]。

在该图中，根节点 MAX 的可能落子有 3 种选择：a_1、a_2 和 a_3。在此例中，博弈树的搜索目标是，确定 1 个最优落子。

首先，从整体上来理解 MAX 方和 MIN 方如何按照最优策略行棋。在该图中，第 3 层都是叶子节点，并且每个叶子节点都有一个反映其价值的效用值 (这些值是从 MAX 方的角度定义的，值越大，对 MAX 方越有利)。例如，从左到右，第 1 个叶子节点的效用值为 3，第 2 个叶子节点的效用值为 12。显然，MAX 方更喜欢第 2 个叶子节点的棋局，而 MIN 方更喜欢第 1 个叶子节点的棋局。因此，MAX 方偏爱效用值大的节点，MIN 方偏爱效用值小的节点：

- $Minimax(s) = Utility(s)$，如果 s 为终止状态；

[3]按照博弈的说法，单方的 1 次出招被称为 1 个 ply，2 个 ply 被称为 1 个回合，它对应着博弈树上的 1 个对弈深度。

- $Minimax(s) = max_{a \in Actions(s)} Minimax(Result(s,a))$，如果 s 为 MAX 节点；
- $Minimax(s) = min_{a \in Actions(s)} Minimax(Result(s,a))$，如果 s 为 MIN 节点；

对于节点 B 而言，它的值将由 MIN 方确定：MIN 方将从 b_1、b_2 和 b_3 中选择 b_1，并将 b_1 的效用值 3 作为节点 B 的值。对于节点 C 和节点 D，采用同样的方式来确定它们的值，得到的结果都是 2。

对于节点 A 而言，它的值将由 MAX 方确定：MAX 方将从 a_1、a_2 和 a_3 中选择 a_1，并将 a_1 的效用值 3 作为节点 A 的值。

因此，在状态 A 时，MAX 方的最优落子是 a_1。

在理解了算法的基本原理之后，我们来具体分析一下 Minimax 算法。它使用了简单的深度递归搜索技术来确定每个节点的值：它从根节点开始，一直前进到叶子节点，然后在递归回溯时，将叶子节点的效用值往上回传——对于 MAX 方，计算最大值，对于 MIN 方，计算最小值。在图2-2中，算法从节点 A 开始，一路深度递归到最左边的叶子节点，随后将值 3 回传给节点 B，然后又深度递归到第 2 个叶子节点，值 12 被回传并与值 3 比较，不占优势，B 节点的值维持 3 不变，接着又深度递归到第 3 个叶子节点，值 8 被回传并与值 3 比较，也不占优势，B 节点的值仍然维持 3 不变。对于其余节点的分析，可依此类推。

如果博弈树的最大深度是 m，每个节点的合法落子有 b 个，那么 Minimax 算法的时间复杂度是 $O(b^m)$。该算法一次生成一条递归路径上每个节点的所有后继节点，它的空间复杂度是 $O(bm)$。在实际应用中，该算法的时间复杂度完全不实用。但是，它可以作为数学分析的对象，并可作为设计实用算法的基本算法。

3 $\alpha - \beta$ 算法

在博弈树搜索中，节点数目的指数级增长趋势是无法消除的。但是，可以巧妙地修改基本的 Minimax 算法，使之有效地提高搜索效率。

让我们简单地回顾一下 A* 算法。在 A* 算法中，待搜索的节点按 (搜索代价) 优先级被插入到优先级队列 frontier 中。依据 A* 算法的原理，该算法从来不会扩展那些搜索代价大于 f^* 的节点。实际上，从树搜索的角度看，这就是一种剪枝 (Pruning)——这些节点及其分支没有得到扩展的机会，而被整个地剪掉了。

类似的思路，可以应用于 Minimax 算法，剪枝技术将剪掉那些不影响决策效果的分支。

为了理解 Minimax 算法中的剪枝技术，以图2-2中 MAX 方在根节点 A 处的决策为例：

$$\begin{aligned} Minimax(A) &= max_{a\in Actions(A)} Minimax(Result(A,a)) \\ &= max_{a\in Actions(A)}[min(3,12,8), min(2,4,6), min(14,5,2)] \\ &= max_{a\in Actions(A)}[3,2,2] = 3_{Action=a_1} \end{aligned} \tag{1}$$

下面，将对上式进行 2 种变形，可以得到改善 Minimax 算法的 2 种方法：

- 剪枝技术

$$\begin{aligned} Minimax(A) &= max_{a\in Actions(A)}[min(3,12,8), min(2,4,6), min(14,5,2)] \\ &= max_{a\in Actions(A)}[min(3,12,8), min(2,x,y), min(14,5,2)] \\ &= max_{a\in Actions(A)}[3,z,2] = 3_{Action=a_1} \end{aligned} \tag{2}$$

其中，$z = min(2,x,y) \leq 2$。易知，在满足一定条件的前提下，即便不关心或去掉某些分支 (例如，分别使用 x 和 y 替换 4 和 6)，也不会影响到最后的决策。我们把该条件称为剪枝条件，下文将描述；

- 改变子节点的访问顺序

在执行 Minimax 算法时，子节点的访问顺序对算法性能的影响也非常大。例如，在访问 D 节点时，假设算法以 d_3、d_2、d_1 的顺序依次访问子节点，那么可以对公式1进行如下的变形：

$$\begin{aligned} Minimax(A) &= max_{a\in Actions(A)}[min(3,12,8), min(2,4,6), min(2,5,14)] \\ &= max_{a\in Actions(A)}[min(3,12,8), min(2,x_1,y_1), min(2,x_2,y_2)] \\ &= max_{a\in Actions(A)}[3,z_1,z_2] = 3_{Action=a_1} \end{aligned} \tag{3}$$

其中，$z_1 = min(2,x_1,y_1) \leq 2$，$z_2 = min(2,x_2,y_2) \leq 2$。可见，标记 x_1、y_1、x_2 和 y_2 的分支都可以被剪掉。这给我们一个启示，算法应该优先访问那些对某方而言“更有前途的”子节点；

下面，从 Minimax 算法执行的角度来讨论剪枝技术，如图3-3所示。

在该图中，每一节点的左边标出了搜索进行时该节点的取值范围[4]。在图 (a) 中，A 为 MAX 方节点，下层还未有值回传，取值范围为初始范围 $[-\infty, +\infty]$，B

[4]每个节点存在取值范围的原因是，双方都需要考虑对方的招数，反映到数值上即为双方能够取得的效益范围，它是双方妥协的结果。

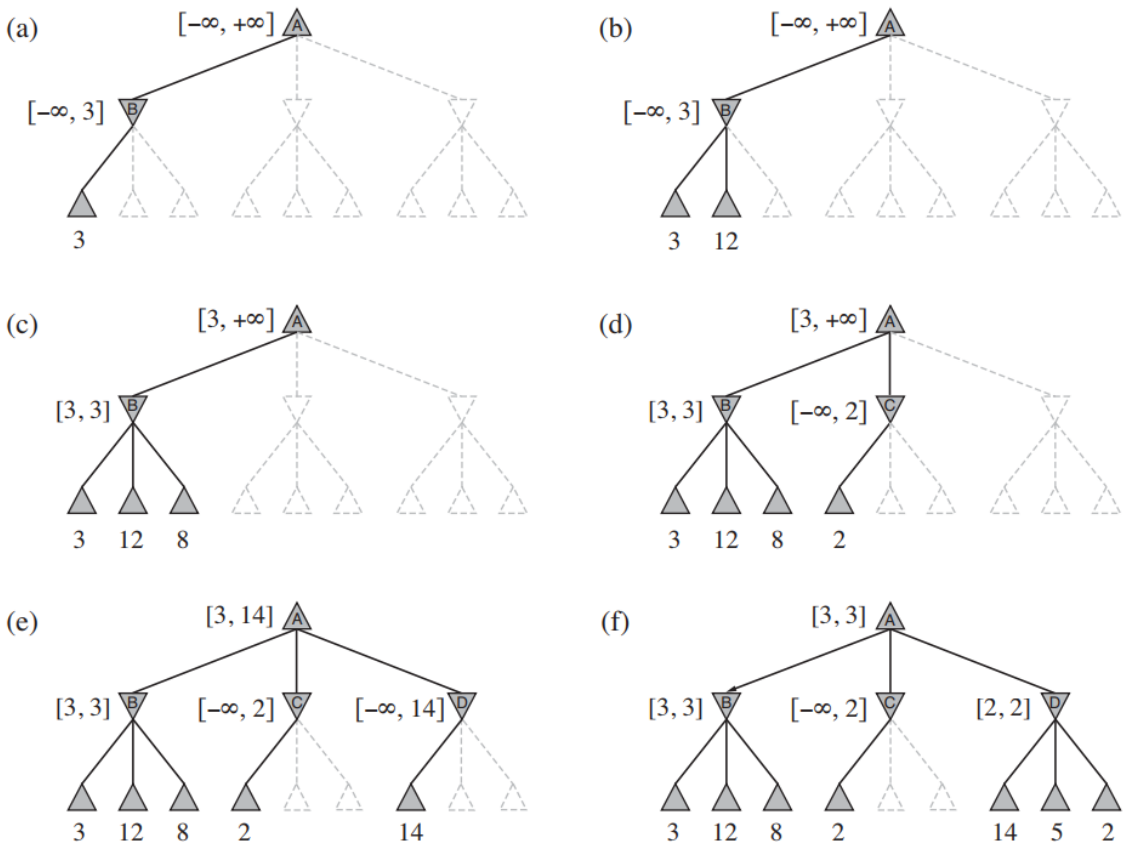

图 3-3: Two-Ply 博弈树的剪枝

为 MIN 方节点，已得到一个回传值 3，取值范围为 $[-\infty, 3]$。在图 (b) 中，B 节点已得到另一个回传值 12，对 B 而言，效益比 3 要差，B 不会接受，故 B 节点的取值范围不会更新。在图 (c) 中，B 节点得到最后一个回传值 8，基于同样的原因，B 也不会接受，故 B 的最终取值为 3，B 将 3 回传给 A，于是 A 更新自己的取值范围为 $[3, +\infty]$。在图 (d) 中，C 为 MIN 方节点，已得到一个回传值 2，取值范围为 $[-\infty, 2]$。然而，A 的取值范围在 $[3, +\infty]$，而 C 将要回传给 A 的值只能在 $[-\infty, 2]$ 范围内，故 A 不会接受。在此情况下，C 节点的 2 个还未访问的子节点，已无扩展必要，这就出现了剪枝。对图 (e) 和图 (f) 可以做类似的分析。

Minimax 算法是基于深度优先的搜索，因此，任何时候只需要考虑树中某条路径上的节点。$\alpha - \beta$ 剪枝算法的名称来源于该条路径上的 2 个回传值：

- α

  路径上发现的 MAX 方的最佳值；

- β

  路径上发现的 MIN 方的最佳值；

在搜索的过程中，$\alpha - \beta$ 算法不断地更新 MAX 方的 α 值和 MIN 方的 β 值，并且一旦条件成熟时即进行剪枝：MAX 方发现回传值低于自己的当前最佳值，即进行 β 剪枝；MIN 方发现回传值高于自己的当前最佳值，即进行 α 剪枝。

下面给出 $\alpha - \beta$ 算法[5]：

```
def alpha-beta(node, depth, alpha, beta, player):
    if depth == 0 or node is a terminal node:
        return the heuristic value of node
    if player:
        v = -∞
        for each child of node:
            v = max(v, alpha-beta(child, depth-1, alpha, beta, False))
            alpha = max(alpha, v)
            if beta <= alpha:
```

[5]在分析算法时，利用树的分形或自相似特点，只分析 Two-Ply 树结构或 1 个对弈深度的树结构即可。

```
                break #beta pruning
        return v
    else:
        v = +∞
        for each child of node:
            v = min(v, alpah-beta(child, depth-1, alpha, beta, True))
            beta = min(beta, v)
            if beta <= alpha:
                break #alpha pruning
        return v
```

4 Monte Carlo 树搜索

4.1 概述

基本的 Minimax 算法需要搜索一棵完整的博弈树，即树的叶子节点必须是终端棋局——具有明确结果 (胜负平) 的棋局。显然，对于那些具有高分支因子 (每步的可选落子数目较多) 的博弈树而言，这是非常不现实的——博弈树的节点数目随树的深度呈指数级增长，即满足 $O(b^d)$ 关系，其中 b 是分支因子，d 是树的深度。

一般情况下，可以采取 2 种基本方法来减轻该问题。第 1 种方法是使用剪枝方法，将那些不满足最优解或目标解条件的分支去掉，例如 $\alpha - \beta$ 算法。第 2 种方法是，使用启发函数 (Heuristic Function) 或评估函数 (Evaluation Function) 对非终局节点进行价值的估算。在这种情况下，算法只需要在博弈树上搜索到一定的深度即可——博弈树不需要完全扩展到终端棋局 (即叶子节点不是终端棋局)。如果评估函数能够准确地反映棋局的状态，那么这种方法将会非常有效。

一种可行的方法是，使用专家级的领域知识对棋局进行评估。但是，这种方式，对一些棋类游戏能够奏效，而对于另一些棋类游戏，要找到一种合理的棋局评估函数，是极其困难的。例如，在计算机国际象棋中，已经找到了能够战胜国际象棋大师的评估函数。1997 年 5 月，IBM 公司研制的深蓝 (DeepBlue) 对弈程序战胜了国际象棋大师卡斯帕洛夫 (Kasparov)。而在计算机围棋领域，同样的方式却难以发挥作用。直到 2016 年，使用了三大技术 (Monte Carlo 树搜索、强化学习与深度学习) 的对弈程序 AlphaGo，以 4:1 战胜了人类九段棋手李世石，取

得了前所未有的成就[6]。经过不断的改进，AlphaGo 的后续版本 AlphaZero，在与 AlphaGo 的比赛中，取得了 100:0 的绝对优势。

Monte Carlo 树搜索 (Monte Carlo Tree Search, MCTS) 是一种主要应用于 (但不限于) 寻找最优决策 (例如博弈树) 的技术。从整体上而言，该方法在状态空间上执行随机采样，并动态地构建一棵搜索树。它具有如下优点：

- 以随机采样的方式，对 (非终端棋局) 节点进行评估；
- 支持实时决策。但是，越多的运行时间，决策的效果越好；
- 无需或只需较少的领域知识；
- 可以有效地解决很难的决策问题，而这些问题不能被其它技术解决，例如计算机围棋；
- 易于实现，且方法具有普适性；

4.2 基本思想

MCTS 的基本思想非常简单——使用随机模拟的方式完成对节点价值的评估，进而为节点的选取决策提供统计依据。它的核心过程是节点的选择与评估，如图4-4所示。

在左图中，节点的选取路径使用粗线条表示，节点的随机模拟评估使用虚线条表示。右图表示随机模拟之后的树状态空间。

可以看出，树状态空间以增量与渐近的方式进行增长。图4-5展示的是一棵使用 MCTS 算法搜索到一定程度的博弈树。

为什么节点的选取也是一个核心过程呢？这涉及到最优决策过程中节点的利用 (Exploitation) 与探索 (Exploration) 问题，这是一个两难决策问题。“利用” 策略偏向于选取那些目前已知最好的节点，因为它认为这些节点能够让智能体获得最大的利益；而 “探索” 策略偏向于选取那些目前并非最好但从长远来看也许是最好的节点。一个好的策略必须在这两者之间取得平衡。

[6]在计算机围棋领域的发展历程中，评估函数一直是一个难以解决的问题。主要原因在于，孤立地评估单个棋子的价值是毫无意义的，必须从整体的、动态的角度来评估它们。这一点，与国际象棋中的情形完全不同。在国际象棋中，可以给不同的棋子指定不同的价值量。Monte Carlo 树搜索技术的出现，为计算机围棋的成功奠定了坚实的基础。

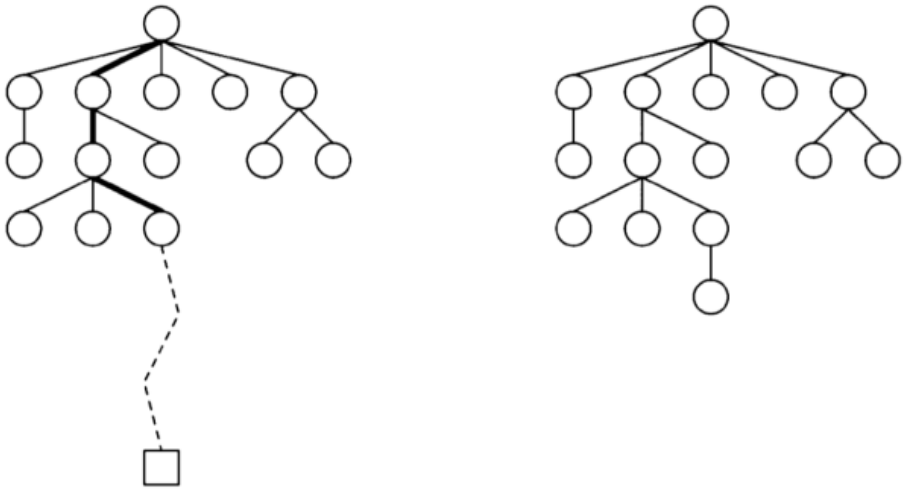

图 4-4: MCTS 的核心过程示意图

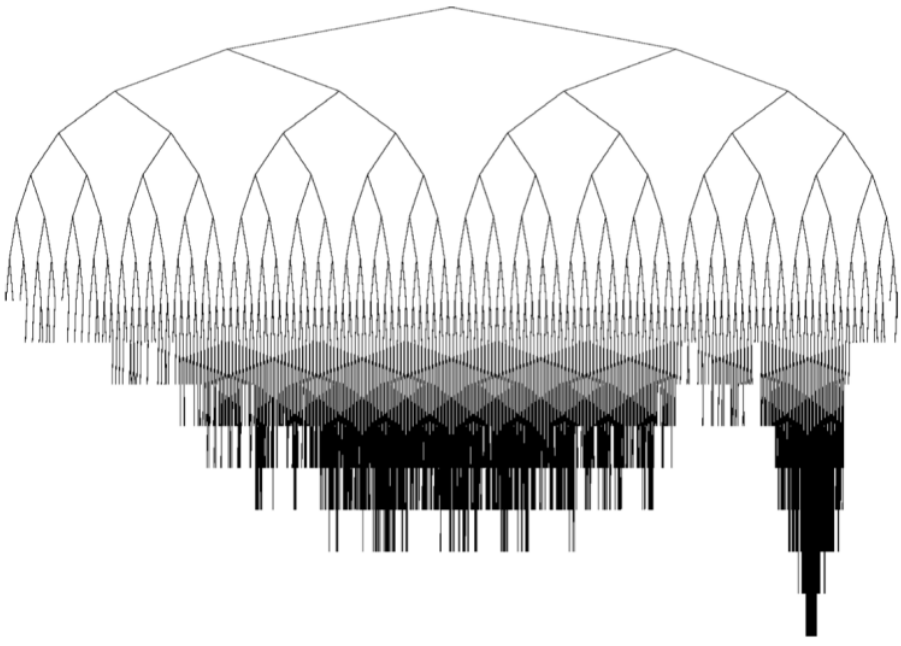

图 4-5: MCTS 搜索到一定程度的博弈树

在详细讨论 MCTS 算法之前，我们首先讨论一下该算法的主要理论基础。

4.3 理论基础

4.3.1 Monte Carlo 方法

Monte Carlo(MC) 方法也被称为统计模拟方法，属于一种基于概率统计的数值计算方法。20 世纪 40 年代，由 John von Neumann、Stanislaw Ulam 和 Nicholas Metropolis 在 Los Alamos 科学实验室为核武器计划工作时，发明了 MC 方法。为象征性地表明该方法的概率统计特征，特借用摩纳哥的赌城 Monte Carlo 为该方法命名。实际上，在此之前，MC 方法就已经存在。1777 年，法国数学家 Buffon 就提出使用投针实验来计算圆周率 π，这被认为是 MC 方法的起源。

对于许多现实的估算问题，很难找到一种合适的方法对其进行准确的推断，例如计算机围棋中的棋局评估。在这种情况下，可以采取近似推断的方法。于是，类似 MC 的随机算法就应运而生。

MC 方法是一类满足某种特征的随机算法的统称。这类方法的特点是，使用随机采样得到近似解，随着采样的增多，近似解越来越接近真实解[7]。

MC 方法通常遵循如下的求解步骤：

1. 定义与输入数据相关的空间；

2. 在该空间中，算法采用某种特定的概率分布，随机地产生一个输入数据；

3. 对该输入数据执行一次确定的计算；

4. 重复第 2-3 步，将生成一系列结果，最后执行统计分析，以得到最终的结果；

下面通过一个简单的例子来说明 MC 方法的基本思想。假设需要计算一个不规则图形的面积。易知，图形的不规则程度与分析性计算方法 (例如积分方法) 的复杂程度是成正比的。针对此问题，可以使用 MC 方法简单地求解。首先，在不规则图形的外围放置一个正方形，使正方形刚好围住不规则图形。然后，向正方形内随机地投掷 N 个小石子，并数出落在不规则图形内小石子的数目 M。最后，可以利用公式 $S\frac{M}{N}$ 计算出不规则图形面积的近似值，其中 S 为正方形的面积。注意，在使用 MC 方法求解问题时，随机数生成器与模拟次数都是非常重要的。

[7]还有一类被称为拉斯维加斯的随机算法，它的特点是，随着采样的增多，找到真实解的概率越大。一旦找到，即为真实解。详细内容，请阅读相关资料。

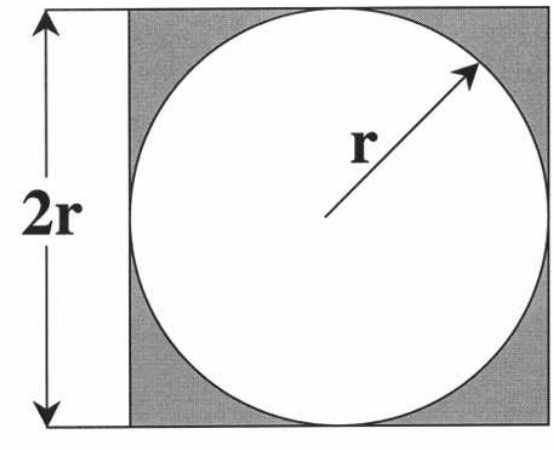

图 4-6: 圆周率的计算示意图 1

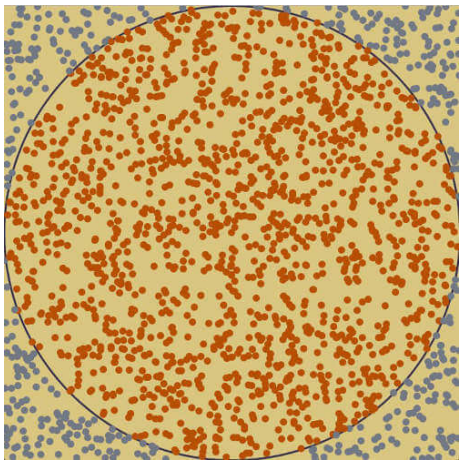

图 4-7: 圆周率的计算示意图 2

另一个有趣的例子是，使用 MC 方法计算圆周率 π。如图4-6所示，圆与正方形的面积之比为：

$$\frac{S_{circle}}{S_{square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} \tag{4}$$

然后，采取与上例同样的方式，随机地向正方形内投掷小石子。投掷结束后，可以得到一个比值关系，并依此计算出 π，示意图如图4-7所示。

MC 方法的应用实例非常多，其作用也相当明显。尤其是对一些采用传统方法难以解决的复杂问题，MC 方法能够很好地予以解决。例如，将 MC 方法与 UCB 算法结合的博弈树搜索算法 MCTS，使得计算机围棋的对弈水平有了一个质的飞跃。

图 4-8: 单臂老虎机

4.3.2 多臂老虎机模型

单臂老虎机 (Slot Machine/One-Armed Bandit) 是一种赌博机，如图4-8所示。单臂老虎机有一个拉杆，一个投币口，三个卷轴。在投下一定数量的硬币后，拉动拉杆，三个卷轴将会旋转，停止后将会获得一定的回报，这个回报可能是正值 (赢钱)、0 或者负值 (输钱)。

多臂老虎机 (Multi-Armed Bandit/N-Armed Bandit) 是具有多个拉杆的老虎机，拉动每个拉杆得到的回报是随机的、互不相关的，且它们可能分别遵循不同的概率分布。

在任一时刻，赌博者只能拉动一个拉杆。赌博者对于每个拉杆的回报信息是一无所知的，要想知道哪个拉杆的回报多，唯一的办法就是不断地去试探 (或试错，Try-and-Error[8])，以此推断出每个拉杆的回报信息。

赌博者需要确定拉动哪个拉杆、每个拉杆的拉动次数与拉动顺序，也需要确定是继续拉动当前的拉杆还是拉动其它的拉杆。赌博者的目的是，通过一系列的拉杆来赢得尽可能多的钱，获取最大的累积回报 (Accumulative Reward)。

多臂老虎机模型属于统计决策领域的经典模型，于 1952 年由 Robbins 提出，它是研究利用 (拉动当前已知回报最优的拉杆) 与探索 (拉动其它潜在最优的拉杆) 问题的理想模型。

[8]强化学习的典型特征。

下面给出多臂老虎机的形式化模型。设多臂老虎机的拉杆个数为 K，每个拉杆的回报具有概率分布 $R_i, 1 \leq i \leq K$，模型的回报概率分布为 $B = \{R_1, R_2, ..., R_K\}$，设 μ_i 为概率分布 R_i 的均值 (平均回报)。赌博者在每一时刻 t 拉动一个拉杆并获得一个回报 $\hat{r}_t$，其目标是最大化累积回报 $\sum_{t=1}^{T} \hat{r}_t$，其中 T 为赌博者拉动拉杆的总次数[9]。

为了衡量各种拉杆策略的优劣，我们使用后悔度 (Regret)ρ 来评价拉杆策略。拉杆策略的后悔度指的是使用该策略拉动拉杆后获得的实际回报与最优策略所获得的回报之间的差值：

$$\rho = T\mu^* - \sum_{t=1}^{T} \hat{r}_t \tag{5}$$

其中，$\mu^* = \max_i\{\mu_i\}$ 是最大的平均回报，$\hat{r}_t$ 是时刻 t 获得的实际回报。该公式还有另一种形式：

$$\rho = T\mu^* - \sum_{i=1}^{K} E[T_i]\mu_i \tag{6}$$

其中，$E[T_i]$ 表示拉杆 i 的平均拉动次数 (期望值)。

从上述公式可以看出，后悔度表示策略没有每次都拉动最好的拉杆所造成的损失。因此，拉杆策略应以减少后悔度为目标。

在拉杆回报分布未知的情况下，为了减少拉杆策略的后悔度，必须确保：

- 任一拉杆被选中的概率不能为 0——避免遗漏最佳拉杆；
- 在选择当前最佳拉杆与潜在最佳拉杆之间取得某种平衡 (折中)——提供利用与探索的平衡机制；

下面介绍几种常用的算法：

1. Naive 方法

   这是一种较为简单的算法。首先，算法执行 N 次尝试，然后统计每个拉杆的平均回报，最后始终选择平均回报最大的拉杆；

2. $\epsilon - greedy$ 算法

   该算法由强化学习之父 Sutton 和 Barto 提出，是一个简单且实用的算法。它以 $1-\epsilon$ 的概率选择当前最佳拉杆，而以 ϵ 的概率选择其它拉杆，其中 $\epsilon \in [0, 1]$ 是一个较小的值。

[9] 多臂老虎机模型等价于单状态 MDP(One-state Markov Decision Process)。

显然，该算法通过 ϵ 来控制“利用与探索”的平衡，ϵ 越小，探索的机会越少，算法偏向保守，稳定性好；反之，算法偏向冒险，稳定性差；

3. UCB 算法

UCB(Upper Confidence Bound，置信度上界) 算法包括 UCB1 和 UCB2 等一系列算法，用于解决利用与探索平衡的问题。下面讨论基本的 UCB1 算法，它由奥地利格拉茨技术大学的 Auer 于 2002 年提出。在对回报的分布没有任何先验知识的前提下，该算法可以将后悔度控制在 $O(\ln T)$ 级别 (拉杆的回报在 $[0,1]$ 区间)。

UCB1 算法计算每个拉杆的平均回报及其不确定量的和，公式如下：

$$I_i = \overline{x_i} + \sqrt{\frac{2\ln T}{T_i}} \tag{7}$$

其中，I_i 表示拉杆 i 的 UCB1 值，$\overline{x_i}$ 是拉杆 i 的平均回报，T 是拉动拉杆的总次数，T_i 是拉杆 i 的拉动次数。

公式的第 1 项反映了拉杆 i 获取回报的平均水平，第 2 项反映了拉杆 i 获取回报的不确定性 (本质上是均值的标准差)，2 项之和形成了拉杆 i 的置信度上界。当拉杆 i 的平均回报较大时，UCB1 值较大，每次选择时拉杆 i 有较大的优势；当拉杆 i 被选中的次数较少时，第 2 项比较大，UCB1 值也较大，每次选择时拉杆 i 也有较大的优势。因此，UCB1 算法利用公式7在利用与探索之间取得了某种折中。

UCB1 算法的常规做法如下：

(a) 对所有的拉杆尝试一次；

(b) 计算每个拉杆的 UCB1 值；

(c) 每次拉动 UCB1 值最高的拉杆，获得相应的回报，更新所有拉杆的 UCB1 值，此过程循环往复；

对于常见的棋类博弈树搜索，可以采用如下公式计算平均回报 $\overline{x_i}$：

$$\overline{x_i} = \frac{w_i}{T_i} \tag{8}$$

其中，w_i 表示节点 i 的获胜次数。

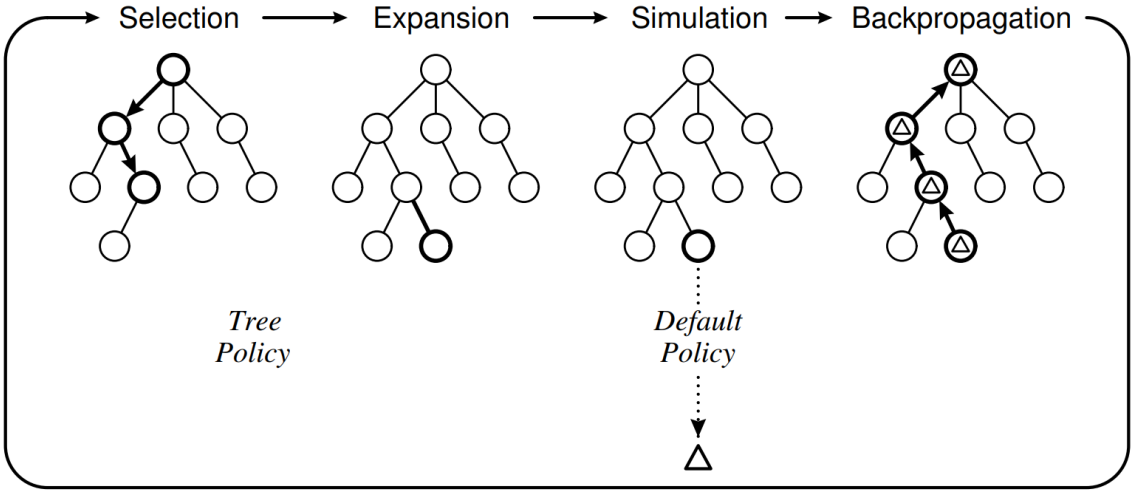

图 4-9: MCTS 算法的一次迭代过程

4.4 基本算法

MCTS 算法由 Remi Coulom 于 2006 年首次应用于围棋对弈程序 Crazy Stone，该对弈程序在随后的一系列比赛中取得了令人瞩目的成绩。

整体上而言，MCTS 算法随机模拟下棋若干次，完成对各个落子的评估，最后选择最佳落子进行对弈。该算法解决了 2 个主要问题：非终局棋局的评估、节点选取中的利用与探索问题。

具体来说，MCTS 算法分为 4 个阶段。在第 1 阶段 (Selection 阶段)，它从当前节点 i 开始，如果其孩子节点都已被至少访问过一次 (称节点为“被完全扩展-Fully Expanded”；反之，称节点为“未被完全扩展-Not Fully Expanded”)，依照 UCB 算法选取最佳孩子节点进行访问。重复上述过程，一直进行到节点未被完全扩展为止，设该节点为 j。在第 2 阶段 (Expansion 阶段)，从节点 j 的未被访问的孩子节点中随机选取一个，设节点为 k。在第 3 阶段 (Simulation 阶段)，从节点 k 开始，随机模拟下棋 (Playout/Simulation)，一直进行到棋局终局，得到明确的胜负平结果。在第 4 阶段 (Backpropagation 阶段)，将得分结果从节点 k 开始向上回传，一直到节点 i，更新每个节点的访问次数与胜率。循环往复执行上述 4 个阶段，直到决策 (思考) 时间用完为止。此时，对于节点 i 而言，一般情况下，选取访问次数最多的孩子节点作为最佳落子。

MCTS 算法的一次迭代过程如图4-9所示。MCTS 算法如下：

```
def MCTS(root):
    seed()
    decision_time = MAX_TIME
    for time in range(decision_time):
        path = [] #for backpropagation
        node = Select(root)
        simulation_node = Expand(node)
        simulation_result = Simulate(simulation_node)
        Backpropagate(simulation_result)
    retrun a child of root, with highest number of visits

    def Select(node):
        path.append(node)
        while node is nonterminal and node is fully expanded:
            node = a best UCT child of node
            path.append(node)
        return node

    def Expand(node):
        path.append(node)
        if node is nonterminal:
            child = a random unvisited child of node
            path.append(child)
            return child
        else:
            return node

    def Simulate(node):
        while node is nonterminal:
            node = a random child of node
        return result(node)
```

```
def Backpropagate(result):
    for node in path:
        update node's statistics with result
```

5 练习

1. 实现 TicTacToe 的 Minimax 对弈程序；
2. 实现 TicTacToe 的 $\alpha - \beta$ 对弈程序；
3. 实现 TicTacToe 的 Monte Carlo 对弈程序；

6 参考文献

1. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。
2. 马少平，朱小燕。《人工智能》，2004 年 8 月第 1 版，清华大学出版社。
3. Wikipedia: Minimax.
4. Wikipedia: Monte Carlo Method.
5. Wikipedia: Monte Carlo Tree Search.
6. Introduction to Monte Carlo Tree Search
7. 刘知青，李文峰。《现代计算机围棋基础》，北京邮电大学出版社，2011 年 4 月。
8. Browne, Cameron & Powley, Edward & Whitehouse, Daniel & Lucas, Simon & Cowling, Peter & Rohlfshagen, Philipp & Tavener, Stephen & Perez Liebana, Diego & Samothrakis, Spyridon & Colton, Simon. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games. 4:1. 1-43. 10.1109/TCIAIG.2012.2186810.

9. Coulom R. (2007) Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik H.J., Ciancarini P., Donkers H.H.L.M.. (eds) Computers and Games. CG 2006. Lecture Notes in Computer Science, vol 4630. Springer, Berlin, Heidelberg.

10. MC 方法入门.

11. Wikipedia: Multi-armed bandit.

12. Monte Carlo Tree Search - beginners guide.