基于 $\alpha-\beta$ 剪枝与位运算的反黑白棋

王宇航 12012208

I. 项目概述

人工智能是否真的拥有智能？这个话题从未停止过讨论。自人工智能诞生之始，就和游戏紧密结合在一起。人们通常认为，人类对游戏的决策过程是蕴含着人类的智能的。当人们创造出一个能够完成人类的某种游戏的程序，我们认为这个程序拥有了某种类人的“智能”。因此游戏领域会成为人工智能测试的良好环境。

相较于其他游戏，人工智能研究者更关注于棋类游戏，因为棋类游戏有着高度的形式化和受限的规则，大部分是一种典型的零和博弈问题。在人工智能领域，这种多个智能体相互竞争的环境被抽象为对抗搜索问题。黑白棋相比于其它棋类，规则简单，易于建模，因此成为了对抗搜索模型的典型代表。黑白棋又称奥赛罗棋。

图 1. 黑白棋

棋盘大小为 8 行 8 列。双方轮流走子，每次走子都必须要俘虏并翻转对方的棋子成为自己方棋子 (简称吃子)，若没有能吃子的落子点，则要弃权一次。双方都没有落子点时则游戏结束。最后以占据地盘多者为胜。

本项目以反黑白棋作为研究对象，对对抗搜索展开研究。本项目主要采用的算法是 Alpha-Beta 剪枝 [1]，优化策略为位运算 [2]，使用启发式函数作为评估函数。通过本项目的研究，笔者期望能对对抗搜索的概念有一个清晰的认知，与此同时，通过动手实现，也能够在理论模型与实际实现间构建桥梁，加深对棋类游戏搜索模型的理解。

II. 实验准备

在正式开始项目之前，笔者先对接下来将会用到的符号进行说明

符号说明

符号	符号意义
α	当前节点的孩子中棋局分数的最大值，对应下界
β	当前节点的孩子中棋局分数的最小值，对应上界
a	节点当前合法行动
a_{list}	节点当前合法行动列表
v	节点当前最优棋局分数
b	搜索树节点的最大分支数量
m	搜索树的最大深度

与此同时，本项目的棋盘由无符号整型数储存，bit 为 1 的位表示棋盘该位置有子，反之无子。因此需要特殊说明

棋盘说明

符号	意义	类型
board	棋盘当前状态	numpy.ndarray
board[0]	黑棋棋盘当前状态	numpy.uint64
board[1]	白棋棋盘当前状态	numpy.uint64
point	当前合法落子位置	numpy.uint64

需要注意的是：uint64 中，最高位表示的是棋盘左上角，而在 point 中，bit 为 1 的位表示该位置为合法落子区域。

III. 研究方法

1. 工作流程

程序启动，根据棋盘状态判断合法落子位置，并进入剪枝搜索。

剪枝搜索中，若满足终止条件，进入评估函数，将棋局得分回溯。否则递归地扫描决策树进行剪枝。剪枝过程中，每进入新的节点都需计算合法落子并更新棋盘状态。

其中，剪枝搜索对应 Algorithm1,Algorithm2 与 Algorithm3，寻找合法落子位置对应 Algorithm4，搜寻二进制串中 1 的个数方法对应 Algorithm5, 该算法基于 bit_count() 算法 [3] 实现。

2.Alpha-Beta pruning 算法

Algorithm 1 Alpha-Beta pruning

Input: input board

Output: output a

$v \leftarrow maxValue(board, -\infty, \infty)$ ▷ 最大化分数

return a which score is v

Algorithm 2 maxValue

Input: input board,α,β

Output: output v

if terminal(board) **then**

return eval(board)

end if

$v \leftarrow -\infty$

for all a in a_{list} **do**

$board \leftarrow result(board, a)$ ▷ 更新棋盘

$v \leftarrow max(v, minValue(board, \alpha, \beta))$

if $v \geq \beta$ **then**

return v ▷ 大于上界 β, 当前节点剩余的子节点不会更新父节点的 β, 剪枝

end if

$\alpha \leftarrow max(\alpha, v)$ ▷ 更新下界

end for

return v

Algorithm 3 minValue

Input: input board,α,β

Output: output v

if terminal(board) **then**

return eval(board)

end if

$v \leftarrow -\infty$

for all a in a_{list} **do**

$board \leftarrow result(board, a)$ ▷ 更新棋盘

$v \leftarrow min(v, maxValue(board, \alpha, \beta))$

if $v \leq \alpha$ **then**

return v ▷ 小于下界 α, 当前节点剩余的子节点不会更新父节点的 α, 剪枝

end if

$\beta \leftarrow min(\beta, v)$ ▷ 更新上界

end for

return v

Alpha-Beta 剪枝算法的伪代码如上。该算法基于 Minimax 算法进行改进，能基于已有信息判断子树有无搜索价值，并跳过无用的子树。

3.Alpha-Beta pruning 算法时间复杂度分析

Minimax 算法是基于深度优先的树搜索，其时间复杂度为 $O(\sum_{i=0}^{m} b^i)$=$O(b^m)$。

Alpha-Beta 剪枝算法带来的优化取决于节点孩子的遍历顺序。最坏情况与 MIN-MAX 情况相同，为 b^m。最优情况为：遍历到的第一个孩子即可剪枝，则需要评估的最大叶节点数目按层数奇偶性，分别约为 $O(b*1*b*1*...*b)$ 和 $O(b*1*b*1*...*1)$, 即 $O(b^{\frac{m}{2}})$, 在孩子节点无序情况下，为 $O(b^3)$。

本项目在遍历孩子节点前，首先对孩子节点依据权重进行了排序，以此提高剪枝的次数，优化效率。

4. 位运算加速

本项目在基于用 numpy.uint64 表示棋盘的基础上，实现了常数时间（与棋盘大小无关）的寻找合法落子位置的算法。

该算法无需遍历棋盘寻找合法位置，而是将棋盘整体移动，分为四个方向，横（棋盘移动 1 位），竖（棋盘移动 8 位），自左上至右下（棋盘移动 9 位），自右上至左下（棋盘移动 7 位）。每次将己方棋盘整体按某

个方向移动若干次，并与敌方棋盘进行按位与操作，最终可得到该方向上所有的合法落子位置。例如，寻找竖直向上方向合法落子位置伪代码如下，竖直向下只需将 << 改为 >> 即可。

Algorithm 4 findPoint

Input: input board

Output: output point

$curBoard, oppBoard \leftarrow board$
$tmp \leftarrow (curBoard << 8)\&oppBoard$
$tmp \leftarrow (tmp << 8)\&oppBoard|tmp$
$tmp \leftarrow (tmp << 16)\&oppBoard|tmp$
$tmp \leftarrow (tmp << 16)\&oppBoard|tmp$
$point \leftarrow tmp << 8$
$point \leftarrow point\& \ (curBoard|oppBoard)$
return point

5. 二进制串中 *1* 的个数

由于本项目采用位运算，故会频繁地搜索二进制串中 1 的个数，若采用遍历的方法，则时间复杂度为 O($size^2$), 其中 size 为棋盘长度。但是，由于整个棋局在大部分时间，棋盘均较为稀疏，所以，本项目采用了 bit_count() 算法实现该需求。

Algorithm 5 bitCount

Input: input board

Output: output cnt

while $board \neq 0$ **do**
  $cnt \leftarrow cnt + 1$
  $board \leftarrow board\&(board - 1)$
end while
return cnt

该算法实际上是每次迭代去除二进制串中的最后一个 1，二进制串中有多少个 1，便能迭代多少次，时间复杂度为 O(c),c 为二进制串中 1 的个数，在棋盘稀疏的情况下，能大幅提升搜索效率。

IV. 实验测试

1. 本地实验环境

硬件环境:

操作系统 Windows11

CPU: AMD Ryzen 5 4500U

内存: 16.0 GB (15.2 GB 可用)

软件环境:

python 解释器 3.9

numpy 1.22.2

2. 数据集

本项目使用的数据集均来自于对战服务器上的对局日志，通过分析每一步的行动力，决策时间，实际决策进行评估函数的调整。

同时，笔者在本地实现了一个自动对战的控制类，然而，这个控制类的主要作用是计算每个部分的时间花销。

3. 评估函数设计与分析

本项目的评估函数分为两类，一类为前 50 步使用的评估函数，一类为后 10 步使用的评估函数。后 10 步由于行动力锐减，本地测试可以直接搜到最后一层，于是采用的评估函数为直接比较敌我双方的棋子差值，这里不展开说明。

前一种评估函数本项目采用了启发式函数。评估指标分为：棋盘位置，行动力与翻转棋子数目。棋盘位置的权重为 w_1, 行动力，翻转棋子数目权重分别为 w_2, w_3，二者均可变。cnt 为当前步数,cnt_1 为行动力，cnt_2 为翻转棋子数目。于此同时，棋局分为三个阶段，开局，中局（第 5 步起）与终局（第 35 步起）。最终，本项目的评估函数 h 表达式为

$$h = \begin{cases} \mathrm{w}_1 board & 0 \leq cnt \leq 5 \\ \mathrm{w}_1 board + 8 * cnt_1 & 5 < \mathrm{cnt} \leq 15 \\ \mathrm{w}_1 board + 8 * cnt_1 - 8 * cnt_2 & 15 < \mathrm{cnt} \leq 20 \\ \mathrm{w}_1 board + 16 * cnt_1 - 8 * cnt_2 & 15 < \mathrm{cnt} \leq 35 \\ \mathrm{w}_1 board - 8 * cnt_2 & 35 < \mathrm{cnt} \leq 40 \\ \mathrm{w}_1 board - 16 * cnt2 & 40 < \mathrm{cnt} \leq 50 \end{cases}$$

其中，w_1 的值为 (空缺位置均为-8)

$$\begin{pmatrix} -1024 & 128 & \cdots & 128 & -1024 \\ 128 & \cdots & \cdots & \cdots & 128 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 128 & \cdots & \cdots & \cdots & 128 \\ -1024 & 128 & \cdots & 128 & -1024 \end{pmatrix}$$

该评估函数设计思路如下

黑白棋的本质是逼迫对手下出昏棋，若行动力匮乏，则很容易被对手利用而走出昏棋。由于开局和终局阶段行动力匮乏是正常情况，故行动力仅参与中局阶段评估函数计算，且中局的白热化阶段，进一步提高行动力权重。

黑白棋在开局乃至中局阶段前半部分，一般没有稳定子的存在，大部分棋子易受攻击，故评估函数未加入翻转子个数。当中局阶段过半时，开始加入翻转子个数权重，避免中局过半后大规模吃子。到了终局阶段，进一步加大惩罚，尽量少吃子。

由于本项目是反黑白棋，故需要保持己方棋子少于敌方方能获胜。本项目的棋盘权重采取了贪心的策略，绝大部分棋盘位置权重均为-8，所以棋子数越多，则局面评分越低。以此来贪心地减少己方棋子。然而，这个策略对棋盘的四个角并不适用。占据顶点之后，则己方所有连着该顶点的棋子均变成了稳定子，这是十分糟糕的，故权重设为-1024，有其余方案绝不下顶点。而顶点旁边的两个位置，是我们希望占据的，因为这两个位置能给对手提供下顶点的机会。故权重 128。

4. 速度测试与分析

本项目在时限为 5s 的条件下，搜索树的深度平均能达到六层（中局阶段部分步数为五层，第 50 步起，一律搜十层，其余为六层）。这与本项目在运行速度优化方面所做的努力是紧密相关的。除了将 0-63 的索引映射为对应的 tuple 加入 candidate_list 时，采用了除法，评估函数中采用了加法，其余情况下，本项目所有运算均为位运算。主要优化策略如下：

- 棋盘权重全部设置为 2 的次幂，便于位运算
- 计算棋盘权重时，先用 bit_count() 算法计算棋盘三种权重分别对应的棋子数，再采用位运算分别计算并相加，无需扫描棋盘
- 对于频繁使用的量，在程序开头全部提前声明
- 计算合法落子范围时，分别用三个掩模棋盘将棋盘 C 位，其余位置，四个顶点的合法位置对应提取出来，并以此顺序加入合法落子列表中，以提高剪枝率，加速程序运行

下面将分几个模块进行测试，针对结果进行对应的分析

位运算加速

下面对比了采用位运算与未采用位运算，进行 10000 次寻找合法落子范围函数计算的时间开销。实验数据均为多次运行的平均值，且硬件经过预热处理。

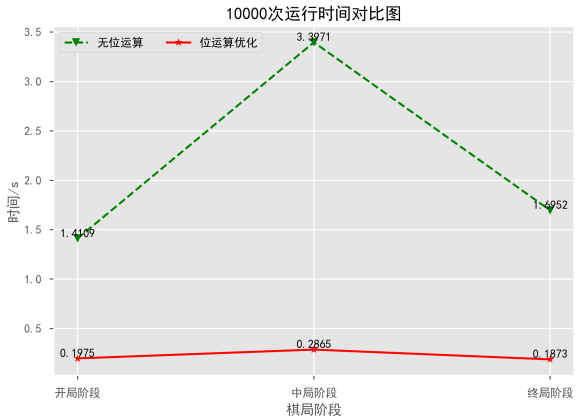

图 2. 位运算优化效果示意图

由于位运算主要针对棋盘进行加速，与剪枝算法无关，故可以单独进行速度测试。由图 2 得出结论，位运算效果十分显著，尤其在棋盘行动力较大时，对程序运行速度有着一个数量级显著提升。

bit_count 算法与节点排序加速

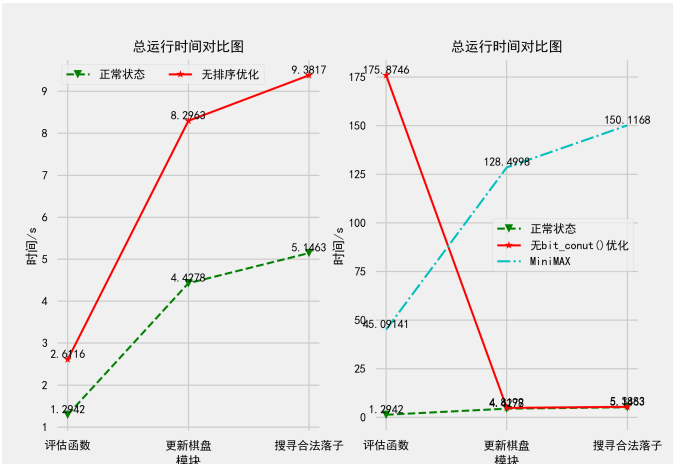

图 3. 搜索树深度为 4 情况下程序的优化效果示意图

图 3 表示的是程序一整局对局后，评估函数计算，更新棋盘，寻找合法位置三者分别消耗的总时间。不难发现，倘若计算评估函数时，不采用 bit_count() 算法，则计算时间完全超出了可接受的范围。因此，在计算稀

疏的二进制串中 1 的个数时，采用该算法，速度会有质的飞跃，这也验证了第三部分研究方法中的结论。下面分析 Alpha-Beta 剪枝算法与 Minimax 算法的差异。前文提及，Minimax 算法时间复杂度为 $O(b^m)$, 这里 m 为 4，即 $O(b^4)$, 同理，Alpha-Beta 剪枝理想时间复杂度为 $O(b^2)$，实际在孩子节点无序情况下，为 $O(b^3)$, 故本项目的剪枝算法在孩子节点经过排序后，时间复杂度应接近 $O(b^2)$，而未排序时应接近 $O(b^3)$，通过观察图示，结论为 Minmax 时间是无排序剪枝的 15 倍左右，是排序后剪枝的 30 倍左右。与理论时间（应呈等比数列）有较大差距。可能的原因是本项目仅对孩子节点进行了简单的分类，将有可能结果好的节点放前面，并没有真正意义上的排序。且孩子节点并不是完全无序的，在无人为排序的情况下顺序是棋盘中索引从大到小。

搜索树深度时间测试

图 4 表示的是根据搜索树深度不同，每步的时间差异。

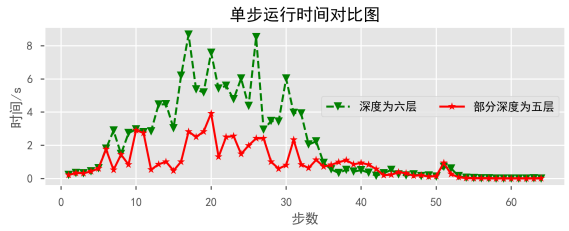

图 4. 搜索树深度恒为 6 和深度为 5 或 6 情况下程序单步执行时间示意图

由于本项目限定时间为 5s，倘若在搜索过程中程序被截断，那么 Alpha-Beta 剪枝的效果便会非常糟糕，因此，我们需要针对超时部分进行处理。本项目的解决方案为：在搜六层超时的情况下，改为搜五层。判断依据是合法落子范围的数量，实验发现，一旦合法落子数量达到 8 个甚至更多，在中局阶段很容易超时，对于此情况，更改搜索树深度为 5。由图可见，很好的解决了问题。

V. 实验结论与展望

1. 实验结论

通过本项目的实验，笔者得出的结论为

- $\alpha - \beta$ 剪枝算法的效果与搜索树的层数高度关联，搜索树的层数每增加一层，算法强度将有质的提高，但是，十分依赖于评估函数的实现，倘若程序有逻辑错误，搜的层数再深也无济于事
- 程序的效率由算法的时间复杂度决定，但也依赖于实现方式。巧妙使用位运算，能使程序的运行效率显著提升

2. 项目有待完善的地方

- 由于时间原因，未能很好地实现稳定子判断
- 对于反黑白棋先手后手的问题没有探究，导致本项目在后手的白棋表现较差
- 搜索时间利用率不足，只有极少数步时大于 2 秒

3. 项目展望

如今，AI 技术在棋类这一充分体现人类思考艺术的领域已经获得了巨大的成功。AI 也在逐渐的渗入我们的生活，改变我们的观念。但还是那句话，欲穷千里目，更上一层楼，有理由相信，AI 将在未来取得更大的发展，彻底改变人类社会。很高兴能在 CS303 这门课上有机会动手实现一个具有一定智能的 AI，相信这将是笔者在入门 AI 的路上的一笔不小的财富。

References

参考文献

[1] Alpha-Beta 剪枝. (n.d.). Retrieved from https://zh.wikipedia.org/wiki/Alpha-beta

[2] Fffasttime. (n.d.). Reversi-Bwcore1. Retrieved from https://github.com/fffasttime/Reversi-bwcore

[3] 剑指 Offer 15. 二进制中 1 的个数. (n.d.). Retrieved from https://leetcode.cn/problems/er-jin-zhi-zhong-1de-ge-shu-lcof/